Scalable abstractions for general-purpose parallel computation

James W. Hanlon

A thesis submitted to the University of Bristol in accordance with the requirements of the degree Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science, March 2014.

70,000 words.

Abstract

Parallelism has become the principal means of sustaining growth in computational performance but there has been relatively little development in *general-purpose* computer architectures or programming models that can deal effectively with large amounts of it. A new general-purpose model of parallel computing would enable standardisation between architectures, high-volume production and software that is portable between different machines, now and as they develop with future technology. There is substantial opportunity to support this in emerging areas of embedded computing, where the problems of sensing, interaction and decision making can exploit large amounts of parallelism.

This thesis demonstrates the essential aspects of a *scalable* general-purpose model of parallel computation by proposing a *Universal Parallel Architecture* (UPA), based on a highly-connected communication network, and a high-level parallel programming language for it called *sire* that can be compiled using simple techniques. The design of sire combines the essential capabilities of shared-memory programming with the benefits of message passing to support a range of programming paradigms and to provide powerful capabilities for *abstraction* to build and compose subroutines and data structures in a distributed context. The design also enables program code to be distributed at run time to reuse memory and for processor allocation to be dealt with during compilation so that the overheads of using distributed parallelism are minimal.

To evaluate whether the UPA is *practical* to build, a high-level implementation model using current technologies is described. It demonstrates that the cost of generality is relatively small; for a system with 4,096 processors, an overall investment of around 25% of the system is required for the communication network. Executing on specific UPA implementations, sire's primitives for parallelism, communication and abstraction incur minimal overheads, demonstrating its close correspondence to the UPA and its scalability. Furthermore, as well as executing highly-parallel programs, the UPA can support sequential programming techniques by emulating large memories, allowing general sequential programs to be executed with a factor of 2 to 3 overhead when compared to contemporary sequential machines.

Acknowledgements

My PhD has been a fantastic experience in which I have had free reign to pursue a line of research that has been fascinating, even though at times, hard to define. I owe a great deal of thanks to my supervisor Simon Hollis who originally encouraged me to embark on it and who has been so supportive throughout, and to David May who has provided endless inspiration and guidance.

There have been a number of people who have contributed to this work in various ways, to whom I am also very grateful: Henk Muller and Simon McIntosh-Smith who have provided valuable advice and discussion throughout; Simon Knowles, Paul Winsor and Ian Stewart who have provided their expertise and helped me to put my work in context; Paul Kelly who was kind enough to host me for a year in his research group at Imperial College in London; and Richard Osborne for his help getting to grips with the XS1 architecture and for the fast emulator he developed.

Of course, success with my PhD has not only required academic and technical guidance, but also moral support and distractions from the intensity of working alone; thank you to my colleagues and friends, particularly Chris Jackson, Markus Jalsenius, Leon Atkins, Steve Kerrison and James Pallister for this.

Finally, I would like to thank the EPSRC for providing the financial support for my studies.

Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

Signed: ______________________________

Date: ______________________________

Contents

List of Figures

List of Tables

List of Processes

Chapter 1.

Introduction

The rate of development of computing in the last century is unmatched by any other technology; consequently, it has become an integral part of science, commerce, industry and everyday life. Its success can be attributed two key factors: the proliferation of a single basic model of computation and the development of sympathetic implementation technologies that have supported an exponential growth in performance.

The conventional basic model of computation has been the *von Neumann architecture* [von45], which can be seen as an efficient realisation of Alan Turing's work on *universal computing machines* [Tur37]. In Turing's work, a computing machine is universal in the sense that, as well as being able to execute any program that it may be supplied as an input, it can emulate any other machine by reading both the description of the machine to be emulated, as well as the input. Making this universality efficient means that the cost of the machine and the performance of an emulation are both proportional to that of a hypothetical specialised device that implements directly the machine being emulated. This has allowed von Neumann machines to take over from special-purpose devices in many application areas as a cheaper and more flexible alternative.

The von Neumann model is based on the abstraction of a single large *randomly-accessible memory* in which both the program and data are stored and accessed by a *processor* that performs computational operations. Arbitrarily large programs and data sets can be handled by scaling the size of the memory and performance can be scaled by increasing the rate at which operations are performed. *Complementary metal-oxide-semiconductor* (CMOS) technology has provided a synergistic implementation technique for the model. It is a technology for densely integrating electronic circuits onto semiconductor materials and ways of scaling it have provided exponential increases in the density and operating frequency of devices.[1] This has provided according increases in memory capacity and processing rate and, driven by huge demand in consumer markets, this has led to rapid iterations of hardware, providing immediate benefits to existing software.

Inevitably, the scaling of frequency in CMOS devices has not continued indefinitely due to fundamental limits in its power efficiency, and since this was the principal way by which the performance of von Neumann machines was scaled, their performance has stopped scaling in the same way. This occurred around 2005 and since then, there have only been very modest increases in sequential performance. However, although operating frequency has stalled, the density of CMOS devices has continued to scale and is predicted to do so for the next 15 to 20 years [Int12a].

1.1. Parallelism in computer architecture

It is now widely accepted that parallelism is the only known means of sustaining growth in computational performance [FM11, p. 109] and as the density of CMOS devices continues to scale, the microprocessor industry will produce designs with larger number of processors. Based on this, it looks increasingly certain that future systems will involve large numbers of processors,[2] potentially with thousands of cores per chip [Bor07]. There are many application areas that will be able to make

[1]This scaling was first observed by Gordon Moore at Intel in 1965 and has since been known as *Moore's law* [Moo65]. It has since become a significant driver for the industry as a self-fulfilling prophecy.

[2]This is reflected in the model used by the *International Technology Roadmap for Semiconductors* (ITRS) which was significantly revised in 2007, setting a basis of 4 processor cores per chip and projected a factor of 1.4x increase every technology generation, which is on average 2.5 years [Int12c, p. 14].

use of large amounts of parallelism. These include artificial intelligence, vision, graphics, natural language processing and human interaction. With continued scaling, these will increasingly be applied in *embedded systems* such as *robotics*, where there is enormous potential for the further application of computers.

The prevailing reaction from microprocessor designers since 2005 has been an evolutionary one to extend the von Neumann architecture with multiple processors that all access the same memory. These devices are known as *symmetric multiprocessors* (SMPs). They maintain compatibility with legacy code, which has perhaps come to be the most important driver in microprocessor design, but they exacerbate the problem of memory latency by increasing the competition for access bandwidth. This is known as the *von Neumann bottleneck* [Bac78]. The consequence is that this approach will likely face diminishing returns [ABC$^+$06] and it is expected that SMPs will not scale well beyond around 24 cores [RKB$^+$09]. They cannot therefore in the long term support large numbers of processors to provide the expected sustained growth in performance from parallelism.

Despite a preoccupation with the von Neumann model and frequency scaling in mainstream consumer devices, parallelism has been an active area of academic study and a practical means of scaling performance in *high-performance computing* (HPC) systems, which deliver performance far beyond that of a single processor. Typical state-of-the-art HPC systems employ hundreds of thousands of processors and have become essential in areas such as science and engineering for dealing with large-scale problems. Parallelism also has a long history in specialised devices such as *graphics processing units* (GPUs) and *digital-signal processors* (DSPs) where the problem domain is well defined and the markets large enough to support their production. In particular, GPUs have developed to support greater levels of programmability, and consequently have found applications in other areas that fit their restricted model of parallelism. In these cases, GPUs can provide significant improvements in performance, but they remain used typically as an accelerator in conjunction with a conventional SMP to handle sequential execution and more general forms of parallelism. This *heterogeneous* approach is at odds with a single basic model of computation since it essentially argues for a variety of special-purpose designs.

1.2. General-purpose parallel computers

Parallelism has been deployed in specific domains with great success but a more general model has not yet emerged. For parallelism to succeed sequential computing and become the *standard form*, a model analogous to the von Neumann general-purpose sequential computer is required that embodies the same concept of efficient universality [Val88, McC93]. Such a model would enable standardisation between architectures and high-volume production, leading to optimised manufacturing processes and low-cost devices, and the development of software that could not only be used on different parallel machines, but also on successive generations in order to benefit from new technologies to deliver increasing levels of performance [May94]. The emergence of large consumer-based markets, for instance robotics, could drive this.

Surprisingly, Valiant has already described the essential aspects of a basic model of parallel computation [Val90b] and it is based on parallel communication networks that can support arbitrary patterns of communication. These provide *architectural independence* by allowing a separation of the implementation of a program from the structure of a machine implementation, with both conforming to the basic computational model. Crucially, architectural independence simplifies the process of writing and compiling *efficient* programs because the machine structure does not need to be taken into account.

In the analogy with the von Neumann model, structural independence corresponds to the abstraction of the memory that supports arbitrary patterns of access but this depends on memory implementations having to invest heavily in the structure and interconnection to deliver this capability. In the 1950s, before the advent of magnetic storage technology that could deliver random access efficiently, *ultrasonic memories* such as *mercury delay lines* had non-uniform behaviour and a great deal of effort was invested to optimising accesses with *optimum codings* to reduce delay [Wil68]. This however significantly complicated the programming of systems with these memories and tied programs to specific memory implementations. Random-access memory facilitated a simple programming model for sequential processors and, up until the mid-1980s, memory latency corresponded closely to computational operations such as arithmetic and branching. In the C programming language [KR88] up to this time, each primitive operation corresponded to one machine operation (memory access or otherwise), providing the programmer with a simple performance model and efficient and predictable execution [Myc07]. However, as the performance of memory has deteriorated with respect to processor speeds, architectural optimisations such as caches have led to the need for programming and compilation approaches that can leverage these and consequently, C has lost its close correspondence to sequential machines and its utility as a simple abstraction.

The essential aspects of a *practical* general-purpose model of computation are therefore a universal machine that can be implemented efficiently with current technology, a high-level language that can express arbitrary programs and a compilation process that can transform programs written in the language to execute efficiently on the machine. In this, there are three key *abstractions*; first, the universal nature of the machine provides an *abstraction of its implementation*, second, the high-level programming language provides a further *abstraction of the machine* to provide the programmer with primitive constructs that are convenient to use to express computational structures, and third, the language also provides the ability to create *new abstractions* such as subroutines, modules and libraries, to simplify the expression of complex program structures.

Figure 1.1 below summarises the components of general-purpose computation and abstractions involved.

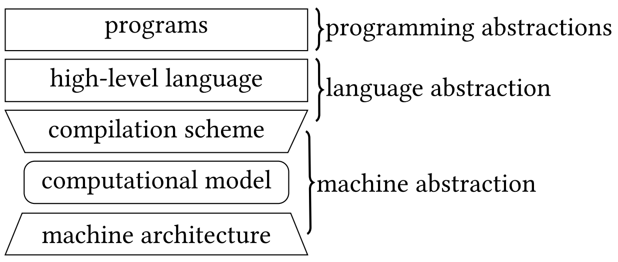


Figure 1.1.: Components of a general-purpose form of computation and the abstractions involved. The shape of the compilation-scheme component is tapered to represent a *reduction* from the high-level language to the computational model and the machine-architecture component is also tapered to reflect that the computational model captures only the essential features of the implementation.

1.3. This thesis

This thesis practically demonstrates the essential aspects of a general-purpose model of parallel computation by proposing a scalable *Universal Parallel Architecture* (UPA) based on a fast, highly-connected communication network and a high-level programming language called sire that corresponds closely to the UPA because it can be compiled using simple techniques. It evaluates both the implementation cost of the UPA using current technologies and its performance, focusing on specific sire programs as well as its ability to support sequential programming techniques.

1.3.1. Contributions

The UPA design

1. An architecture that can support the efficient execution of highly-parallel programming techniques as well as sequential ones, and that can switch between these during the execution of a program.
2. An architecture that allows implementations of it to be easily specialised by adjusting the network performance or balance of processing to memory to optimise for specific workloads.
3. The choice of a folded-Clos network for the interconnect topology, based on:
   a) its equivalence to the hypercube and fixed-degree variants and consequent universality as a communication network, and its ability to support two-phase randomised routing with no increase in the average distance that messages travel;
   b) its hierarchically recursive structure that allows it to be packaged with different technologies and for bandwidth to be flexibly provisioned at each level.
4. The use of a high-degree crossbar switching element to reduce the number of stages in the folded-Clos network and to connect multiple processors to each switch.
5. The use of wormhole switching and a remote memory access mechanism to efficiently support both arbitrary-sized messages and memory accesses.
6. The use of XMOS XS1 processor cores that provide direct support for communication and parallelism to minimise *communication startup latency* and the cost of creating parallel processes respectively, so that the degree of parallelism exploited from a particular problem can be maximised.

The sire programming language design

7. The ability to express and combine a wide range of *programming paradigms*, including message-passing and data-flow structures, task farms and event handlers, in particular, PRAM-style shared-memory parallel computations and RAM-style sequential computations.
8. The concept of a *server* as a language primitive:
   a) to provide a mechanism for *sharing* within a message-passing framework (i.e. combining the capabilities of shared memory programming with those of message passing), where sharing in this context relates to many-to-one patterns of communication;
   b) to provide a basis for distributed parallel subroutines, distributed representations of data (as a basis for distributed data structures) and the modular composition of a program;
   c) to deal with all aspects of abstraction that involve communication, allowing processes to be named (rather than having named communication channels as was the case in occam) in order to facilitate a simple distributed implementation;
   d) to provide a mechanism for explicitly moving pieces of program to the data on which they operate (i.e. between processors);
   e) with declaration syntax similar to standard variable declarations so that a program can be composed in a conventional way with a sequence of declarations followed by a a sequence of operations;
   f) with call syntax similar to local procedure calls to allow a programmer to move easily between local and remote forms of a call to employ parallelism or to distribute data.
9. Facilities for combining collections of servers and creating abstractions based on them.
10. The capability of sire to be compiled using simple techniques (i.e. with a non-optimising compiler) and therefore for the primitive aspects of the language to correspond closely to the operation of the UPA, providing a *transparent execution model*.

11. For sire to be used as a ‘system’ programming language because of its close correspondence to the UPA. This, for example, allows it to be used to build components for managing and optimising distributed systems such as memories with a particular model of consistency, caches to transparently improve locality and facilities for replication and combining to manage distributed data.

Compilation of sire programs to the UPA

12. For a large part of the sire compilation process to be dealt with using source-to-source program transformations into to a simplified *canonical form*, using algebraic properties of the language. This is beneficial because the output is understandable by the programmer and only the small canonical subset needs to be implemented.
13. A compile-time scheme to determine a run-time schedule for the allocation of processes to processors. This avoids the overheads of run-time processor allocation and thus minimises the overhead of using distributed parallelism.
14. Efficient run-time distribution of program code between the memories of different processors to reuse processor memory. This also decouples the compilation process and its run time from the number of processors in the target system and allows minimal program binaries to be produced (i.e. not one for each processor) that can be replicated rapidly over a system at boot time.
15. A compile-time scheme to prevent deadlock that can be caused by many-to-one server channel connections. This is based on queuing of client requests by servers to engage with them, so that client requests do not block other communication traffic in the network.

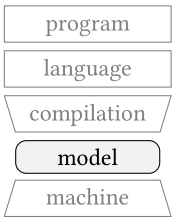
Performance evaluation of the UPA and sire

16. A high-level implementation model of the UPA, based on a packaging of a folded Clos using a H-tree layout and a simplified VLSI model with specific parameters to characterise current technology.
17. Evaluation of the cost and scaling of the UPA implementation, demonstrating that the overall investment in a universal interconnect for a system with up to 4,096 processors is around 20% to 30% of the total cost, compared to 5% with a non-universal 2D mesh network.
18. Evaluation of the performance of the primitive mechanisms for parallelism, communication and abstraction in sire, demonstrating that the overheads are very low:
    a) remote server calls, performed across the network, have an overhead of 8 to 20 times that of local calls;
    b) processes can be offloaded to remote processors if they exceed just a few tens of thousands of operations (around 10 μs to 60 μs at 1 GHz);
    c) the mechanism for process distribution can bring thousands of processors into action in a few thousand cycles (4,096 tiles in around 200 μs at 1 GHz).
19. A compiler for a simple sequential language that generates programs to execute on the UPA with a large emulated memory.
20. Evaluation of the ability of the UPA to support sequential programming techniques by emulating large random access memories. The results of this show that general sequential programs can be emulated with a factor of 2 to 3 slowdown when compared to conventional sequential architectures.

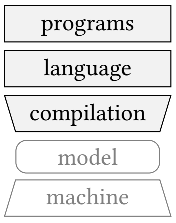
1.3.2. Overview

This thesis is divided into three parts: background, the description of the UPA and sire, and performance evaluation. Each chapter deals with different components of the 'stack' in Figure 1.1 and with each summary below, an illustration of the components it corresponds to is included.

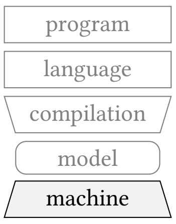
The *background part* consists of three chapters.



Chapter 2 considers the concept of parallel computation independently of any particular architecture or programming language. It focuses on the two main communication paradigms: *shared memory* and *message passing* and argues why conventional shared memory is insufficient for a general model of parallelism.

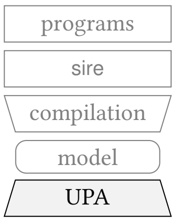


Chapter 3 discusses parallel programming, from high-level issues related to the formulation of an algorithm, to the issues that affect the effectiveness with which a particular language can be compiled. It outlines criteria for a general-purpose parallel language and presents a survey of existing programming approaches.

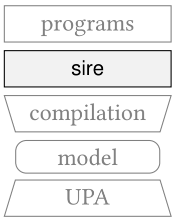


Chapter 4 describes the theoretical results concerning universal networks, describes the practical details of an implementation of an interconnection network and presents a short survey of existing parallel machines.

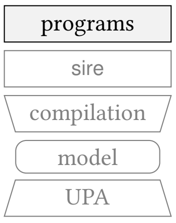
The *UPA and sire language part* consists of four chapters.



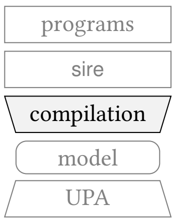
Chapter 5 presents a description of the UPA and an explanation of its main characteristics.



Chapter 6 presents a definition of the sire programming language that introduces it incrementally and, for each part, provides simple examples of its use.

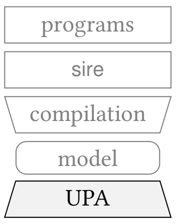


Chapter 7 demonstrates a range of high-level programming *structures* that can be expressed with sire and how they can be combined to form complex programs. The chapter is divided into message-passing *process structures* and *server structures*.

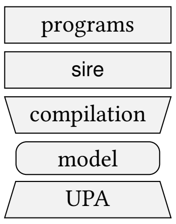


Chapter 8 describes how sire programs can be compiled. It is divided into a description of the sequence of *transformations* applied to convert a program into canonical form, and a description of the generation of executable code and its combination with run-time program components.

The *evaluation of implementation cost and performance part* consists of two chapters.



Chapter 9 describes a *realistic* implementation model for the UPA, based on current production technologies, and presents a range of hypothetical systems to show how the relative proportions of processing, interconnect and memory scale.

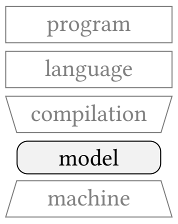


Chapter 10 presents the methodology and results for the empirical performance investigations based on the modelled implementation of the UPA from Chapter 9. The first set of experiments investigate the efficiency of the primitive mechanisms for parallelism and communication in sire and the second set of experiments investigate the ability of the UPA to support sequential programming techniques.

Finally, Chapter 11 provides a summary and conclusion.

Part I.

BACKGROUND



CHAPTER 2.

PARALLEL COMPUTATION

This chapter examines the concept of *parallel computation*, to provide a foundation for the later discussion of parallel programming and machine architecture in Chapters 3 and 4.

Since this thesis is concerned with *computers*, i.e. general purpose programmable devices to perform computations, and using parallelism to scale performance, in this context, a *parallel computation* is expressed in a high-level language as a *program* and *compiled* to a format for execution by a device consisting of a collection of sequential processors. These correspond to the components above and below the 'model' component of the above stack.

Section 2.1 discusses communication, the principal issue in parallel computation, and the characteristics of the *message-passing* and *shared-memory* paradigms; Section 2.2 discusses the two most practical models of parallel computation based on these paradigms; lastly, Section 2.3 provides a summary.

2.1. Parallelism and communication

Conceptually, a *parallel computation* consists of a collection of *processes* that independently operate sequentially and work cooperatively to perform a computational task, i.e. arithmetic and logic operations. Cooperation is achieved by transferring data between one process and another, which is called *communication*, and the *structure* of a parallel computation corresponds to interactions between processes.

It is evident that a parallel computation must contain some communication, otherwise it would be a set of independent tasks that produce separate results. Communication is therefore an intrinsic aspect of parallel computation and it is the principal way in which parallel computer architectures and programming languages differ from one another.

There are two main paradigms in parallel computation that characterise the way processes interact: message passing and shared memory.

- *Message passing* is based on two operations, *send* and *receive*. A communication occurs when one process performs a send operation and it is matched by a receive operation performed by another process. Message passing is *synchronous* when the sender waits until the receiver is ready before sending a message, and *asynchronous* when it does not.
- *Shared memory* is based on two operations, *read* and *write*, that provide access to an area of storage shared between a collection of processes. Processes cooperate with one another by accessing the shared space.

Message passing corresponds directly to the concept of communication and therefore to the wide variety of things that communicate. This includes, at one end of the spectrum, us as human beings that talk to each other, and at the other end, electronic devices that use electric or electromagnetic signals to convey data from one place to another.

Since shared memory has to be implemented with electronics, and hence a form of message passing, it can be seen to provide an abstraction of communication. This abstraction is important in parallel computation because it provides a natural way to formulate parallel algorithms that are based on access to shared data. Because of this and its similarity to the conventional model of sequential computation, shared memory has provided the basis for many parallel architectures and programming

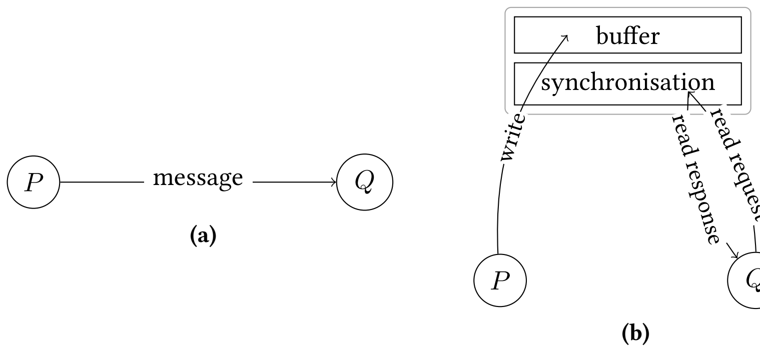


Figure 2.1.: An illustration of communication in message passing, where it occurs with a matching pair of send and receive operations, and in shared memory, where it must be emulated by setting a synchronisation flag to indicate a write has completed and the reader can access the message.

languages. However, a model of parallel computation based on shared memory neglects the intrinsic aspect of communication, making it inefficient to perform and furthermore, introduces problems with non-determinism due to *data-access races* and poses problems for distributed implementations because of *consistency*. These issues are discussed in the following sections.

2.1.1. Communicating with shared memory

When it is necessary to perform communication between two processes in a shared-memory model of computation, message passing must be emulated (it will be argued in §3.1.3 [p. 21] that many common paradigms in parallel algorithms are based on interactions). The problem with this is that it is not possible to do so efficiently. For a process P to send a message to another process Q in a shared memory, they require a synchronising location S and a buffer B, then:

1. P repeatedly reads the value at location S while it is 0;
2. when S is 1, P writes the value to be transmitted in B;
3. P writes the value 1 in S.

For Q to receive the message:

1. Q repeatedly reads the value at location S while it is 0;
2. when S is 1, Q reads the value of B, which is the message sent by P;
3. finally, Q resets S to 0.

This communication scheme is illustrated in Figure 2.1b. The source of the inefficiency in this is in the *polling* behaviour of the receiver since it amounts to wasted work. To avoid this, the receiver could be notified by an *interrupt* generated by the sender while it is performing other work. However, interrupts require *context switches* where execution of a process is suspended and all processor state associated with process is saved so the interrupt can be dealt with, which introduces further overhead and timing issues.

The above scheme for implementing message passing illustrates that there is an inherent difficulty in implementing interactions with shared memory. This can be summarised as whenever two or more processes access a particular storage location, a synchronisation mechanism is required to ensure data is written before it is read.

The inefficiencies of polling or interrupts can be reduced by augmenting the implementation of shared memory with primitives to support synchronisation. For example, an operation to synchronise two processes could be implemented by sending messages to deschedule the execution of a process waiting to receive a message and to schedule the execution when it has arrived [May94]. This however essentially amounts to the addition of message-passing functionality to the implementation.

2.1.2. Non-determinism and shared memory

Another important issue with shared memory is that it introduces the potential for *non-deterministic behaviour*. This occurs when the sequences of accesses to some shared state are dependent on the sequencing of events in the execution. For example, if a memory location is read before a corresponding write to it has completed, the value read will not be valid. To illustrate this, consider two following two parallel processes that both increment a count stored at a memory location `x`:

```
      process P                 process Q
          ⋮                         ⋮
  a := memory[x]            a := memory[x]
  a := a + 1                a := a + 1
  memory[x] := a            memory[x] := a
          ⋮                         ⋮
```

If the read operation from Q is executed before P has written back the incremented value, Q will write back the wrong value. The behaviour of the program is then dependent on the way in which the read and write are sequenced, rather than the semantics of the program. This is called a *race condition*. A program in which there are no race conditions is said to be *scheduling invariant* because it does not depend on the sequencing of events in the execution [Hoa85].

Race conditions pose significant problems in programming because their non-deterministic effects make them difficult to reproduce and correct. They also make both formal and empirical verification of program correctness difficult because the potential state space of the program grows exponentially with the number of processes accessing it. As such, there are no simple or widely used techniques for verifying the correctness of shared-memory parallel programs.

2.1.3. Consistency in distributed implementations

In implementations of shared memory that are not entirely centralised and therefore comprise more than one physical memory (for example with per-processor caches or distributed memory systems) the issue of *memory consistency* arises.

In a *sequential consistency model*, the order in which memory accesses are performed is fixed for any sequential ordering of the operations performed by a program; for a parallel program, this is a serialisation of the execution. Sequential consistency appears to be a sensible basis for the semantics of shared-memory programs but because there are significant opportunities for *relaxing* aspects of this model in the interests of improving performance, many shared-memory approaches have adopted *weak consistency models* [AG96].

Weaker models of consistency remove guarantees about the ordering of reads and writes. Consider a distributed memory system where the total memory capacity is divided into physically separate devices but they are addressed in one logically shared space, and the access latency between different processors and memories is variable.[1] If a processor P performs a write to a memory location M then synchronises with another processor Q that then reads from M, no guarantee can be given that the read will arrive after the write because network latency is variable (often referred to as *non-uniform memory access* (NUMA)). To provide sequential consistency in this situation, each processor must be informed about changes made to the memory by other processors so that a read always returns the value written by the most recent write [LH89]. The cost of such a scheme is significant and becomes complex when performance optimisations are employed. Moreover, choosing a good trade-off between the programming semantics and implementation complexity and performance is difficult to do for a general class of programs.

[1]A shared-memory multiprocessor with per processor caching is conceptually similar except each processor maintains a small subset of the global memory. Consistency issues arise because writes can invalidate cached values.

2.1.4. Message passing

Message passing has some clear benefits over shared memory as a basis for parallel computation:

- when it is performed synchronously, it combines transfers of data with synchronisation into a single primitive concept;
- it corresponds closely to the operation of electronic systems (and, in general, of things that communicate) so it is simple to implement;
- it allows programs to be expressed in terms of localised communications and processing, which makes an efficient use of resources in distributed implementations;
- non-determinism only occurs with many-to-one patterns of communication (multiple senders and one receiver) and can be dealt with explicitly;
- it can be formalised.

The main perceived difficulty with message passing is that it results in a programming model that requires all communication to be managed explicitly. The consequences of this are that:

- programs lack clarity since large portions of them deal with messaging;
- representations of data are fragmented between the processes that operate on it;
- it is difficult to express arbitrary patterns of data access.

This perception however is based on the relatively few message-passing approaches that exist, which in general provide little flexibility in the patterns of communication that can be expressed, primarily with *point-to-point* relationships between a pair of processes (in §3.3 [p. 28], the capabilities of existing message-passing approaches are surveyed). However, messages can also be passed *one-to-many* in a broadcast, *many-to-one* with client-server relationships or even *all-to-all* in a total exchange.

A key argument in this thesis is that many-to-one patterns of communication, in particular, are essential for dealing with the above shortcomings of current message-passing approaches. This is because it provides a basis for *sharing* and thus a way to integrate the benefits of shared memory with those of message passing.

2.2. Models

A *model of parallel computation* refines the general concept of a parallel computation that was given at the start of this chapter by restricting the way in which processes can evolve, are arranged and can communicate. A model defines neither a programming language nor machine architecture[2] but is said to be *practical* if:

- it is a convenient target for the compilation of *high-level programming languages*;
- it can be implemented efficiently with current technologies.

A practical model can therefore be used to define a relationship between programming languages and architecture. It is an essential aspect of a general-purpose approach because it provides a common basis for the design and analysis of algorithms, programming languages and machine architectures.

In this section, the two most practical existing models of parallel computation are discussed. These are *communicating sequential processes* and the *parallel random-access machine*.

[2]In this context it might also be referred to as an *abstract machine* or *bridging model* [Val90a].

2.2.1. Communicating sequential processes

Communicating sequential processes (CSP) [Hoa78, Hoa85] is a mathematical formalism for describing patterns of interaction in systems of concurrent processes. It is built on the primitive concepts of message passing, parallel composition of processes and guarded commands for dealing with non-determinism. These can be combined to express complex systems in a simple way. It is part of the family of *process calculi* that includes Petri Nets [Pet77], the *calculus of communicating systems* (CCS) [Mil82], which was a direct influence for CSP, the *Actor model* [Agh85] and the π*-calculus* [Mil99].

Although CSP is conceptually similar to other communicating process models, its choice of primitives make it practical both as a basis for a programming language and to implement.

- *Synchronous message passing* provides simple semantics and does not require buffering because a sender always waits for the receiver to be ready to receive a message. In contrast, *asynchronous message passing* (on which the Actor model is based, for example) requires buffering which complicates theoretical analysis, can cause deadlock, can introduce inefficiency when they grow large and can introduce latency in accessing the buffer [Hoa85, §7.3]. To note, when buffering is required, it can be implemented simply with synchronised message passing.
- *Bounded process creation* prohibits the dynamic creation of processes from either parallel recursive processes or unbounded arrays of processes. This is because the bounded case is simpler to analyse and because dynamic process creation is difficult to implement efficiently, particularly with distributed memory architectures.
- *Non-determinism* is permitted and managed explicitly with the use of a special *guarded alternative* operator.

In comparison, the π-calculus is much more dynamic; it provides mechanisms for *process mobility* and to communicate channels as first-class entities between processes. This allows it to express dynamically evolving systems, but it is faced with significant implementation challenges in providing a basis for a scalable parallel programming model.

The simplicity and practicality of CSP inspired the *occam* programming language [INM84] and the *INMOS transputer* microprocessor [INM88c], as well as numerous other programming languages. It has also found uses in many other areas of computer science [AJS05].

2.2.2. Parallel random-access machine

A *parallel random-access machine* (PRAM) is the parallel analogue of the RAM model and was first proposed to study the computational power of parallel machines with respect to serial ones [FW78]. A PRAM consists of an unbounded number of processors that operate synchronously in parallel and access an unbounded shared memory. This makes it particularly suited to *data-parallel* computations where collections of similar processes operate synchronously on shared data structures (the concept of data parallelism is explained in more detail in Chapter 3).

The PRAM model is *idealised* because it assumes that memory accesses and synchronisation are primitive operations that complete in the same (unit) time as regular computational operations.[3] Its simplicity has provided a concrete basis for the design and analysis of parallel algorithms and has generated significant work in the development of efficient PRAM algorithms and techniques; see [KR90] for a survey. The idealistic assumptions it makes with respect to memory performance pose significant problems for an efficient implementation because it ignores the practical issues of latency, access contention and synchronisation overheads. However, a significant research effort has generated ways that PRAMs can be emulated efficiently on practical parallel machines,

[3]In this respect, the PRAM is very similar to the RAM model in that it is assumes that a RAM provides accesses to memory in unit time. However, the access time of conventional *dynamic random-access memories* (DRAMs) can be several orders of magnitude larger than computational operations.

e.g. [KU88, Ran87, AHMP87]; the most prominent of these is Valiant's *bulk-synchronous parallel* (BSP) model [Val90a, Val90b].

The bulk-synchronous parallel machine model

The BSP model is capable of an efficient implementation on practical scalable parallel architectures and provides a compilation target for PRAM programs. It consists of a distributed-memory parallel computer with a communication network that delivers messages between arbitrary pairs of components that implement global read and write operations, and a (bulk) synchronisation mechanism that can perform synchronisation between processors. PRAM programs are compiled to a BSP machine with additional software components that deal with the distribution of memory accesses to avoid hotspots and scheduling of multiple processes per processor to hide communication latency.

The separation of the mechanisms required to support a PRAM emulation from the underlying architecture allows a BSP machine to be programmed directly so that it can be used with no emulation overheads [Val90a] and to be capable of a simple implementation. This is perhaps the main reason for its (limited) success [McC94]; other emulation schemes that include complex mechanisms directly, such as those mentioned above, do not provide the same flexibility.

Adoption of BSP and PRAM algorithms

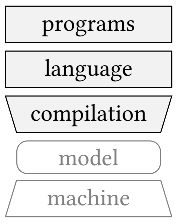
Despite the apparent ability of the BSP model to provide a efficient abstraction of a parallel computer and to support shared-memory programming approaches, which are closely related to the practices of conventional sequential programming, the BSP model has not yet received widespread adoption. One reason for this is that it suffers from the problems associated with shared-memory programming relating to communication, non-determinism and consistency that were explained in §2.1 [p. 11]. Another reason is that it is expensive to perform bulk synchronisation. Although the model allows synchronisation for subsets of processors, without an efficient method to implement pair-wise synchronisation, the model does not easily accommodate less bulk-synchronous approaches such as message passing.

2.3. Summary

The two main paradigms of interaction in parallel computation are shared memory and message passing. Message passing corresponds directly to the concept of communication and shared memory provides an abstraction of it.

Shared-memory is a useful tool in computations that are based on shared data with arbitrary access patterns but it introduces a number of problems. First, implementing communication is difficult and leads to inefficiency, second, concurrent accesses can cause non-determinism and therefore problems for programming and verification, and third, scalable implementations incur significant overheads to maintain consistency. Message passing, in contrast, is simple to implement, can be used to express efficient and scalable parallel programs and, with models such as CSP, can be formalised and deal explicitly with non-determinism. However, message-passing approaches are perceived to be low-level and burden the programmer with the explicit management of communication, making it difficult to express programs clearly.

The BSP model has demonstrated that it is possible to implement high-level programming approaches efficiently on practical scalable parallel architectures, but it has had limited success. However, the theory of universal communication networks, on which the BSP model is built, underpins the work in this thesis and will be explained in Chapter 4.



Chapter 3.

Parallel programming, languages and compilation

This chapter discusses *parallel programming*. It begins by outlining the general high-level issues of parallelism and the formulation of parallel algorithms. Then it discusses the issues associated with a parallel-programming language by outlining a set of criteria for an effective general-purpose parallel programming language. Lastly, based on the proposed criteria, it surveys existing programming approaches and their suitability for a general-purpose model of parallel computation.

The outcome of this chapter is to establish the requirements of a general purpose parallel-programming language, in order to motivate the proposed sire programming language and to put it in context with existing approaches.

3.1. Principles

This section discusses the two key principles that underpin the discipline of parallel programming: *problem decomposition* and *parallel efficiency*. These transcend the details of particular programming languages, compilation techniques and machine architectures, and are therefore foundational to the construction of efficient parallel computations.

3.1.1. Problem decomposition

The first step in developing a parallel algorithm is to identify how it may be *decomposed* into a number of independent parts that can be performed in parallel.

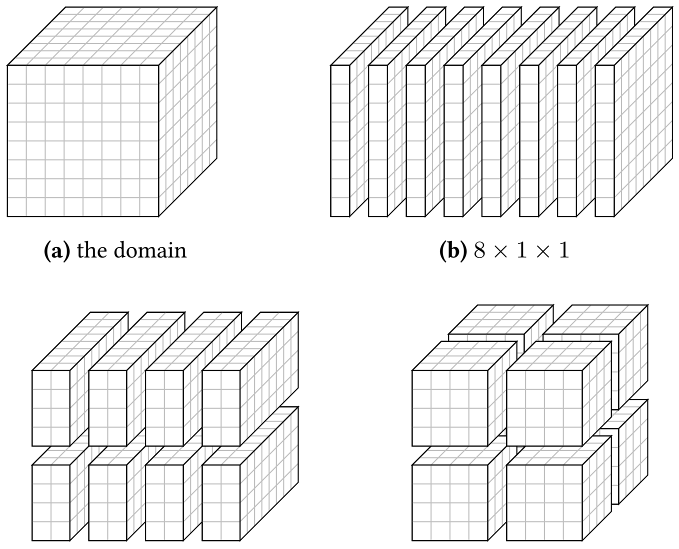
There are three main forms of problem decomposition [GKKG03]:

- *domain*, where the problem space is divided into a number of smaller parts that can be operated on independently;
- *recursive*, where the problem itself is divided into a number of smaller sub-problems of the same type;
- *functional*, where the problem itself is divided into a number of smaller sub-problems of different types.

In general, domain decompositions are referred to as *data parallel* [HSJ86], and ones that are not are referred to as *task parallel*.

Domain

With domain decomposition, the computational domain of a problem is partitioned into a number of sub-domains that can be operated on simultaneously by a number of similar processes [Gro92]. Typically, there will be dependencies between sub-domains and each parallel computation will proceed in phases where it performs some local computation, followed by some communication to resolve the dependencies. Dependencies often arise at the boundaries of a sub-domain. In the case where there are no dependencies between sub-domains, parallel computations can proceed independently with no communication.



(a) the domain

(b) $8 \times 1 \times 1$

(c) $4 \times 2 \times 1$

(d) $2 \times 2 \times 2$

Figure 3.1.: Decompositions of a 3D domain (a) into a number of sub-domains in 1 (b), 2 (c) and 3 (d) dimensions to be processed in parallel between 8 processes. The surface area of a sub-domain often relates directly to the amount of communication that must be performed. A decomposition in just one dimension produces an elongated sub-domain with a high surface area, as in (b), whereas in 3-dimensions the surface area is much lower, as in (d).

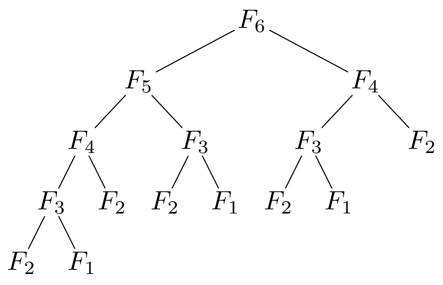
Domain decomposition is a natural way to employ parallelism since it separates communication and computation and it produces algorithms where the problem size can easily be scaled to increase the level of parallelism and performance. Figure 3.1 shows some examples decompositions of a 3-dimensional domain.

Many problems in HPC can be effectively parallelised with domain decomposition and due to the large nature of the problem they are trying to solve, they can be scaled up to operate on very large machines. For example, many simulation algorithms are structured as grids where each point on the grid updates according to the state of a small number of neighbouring points. These connections may have a regular pattern, for example in weather simulations where the atmosphere is partitioned into three dimensional blocks [LCD+08], or they may not, for example in *adaptive mesh refinement* computations where the domain is more finely discretised in areas of interest [BO84].

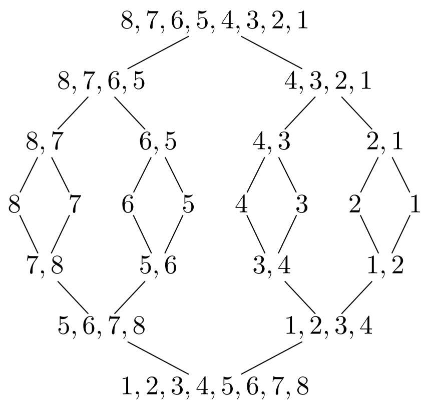
Recursive

With recursive decomposition, a problem is solved by dividing it into a number of independent sub-problems of the same type. Each of these is solved in parallel using the same approach, until a termination condition is reached. The results of each of the sub-problems are then combined to obtain a final result. This is also known as a *divide-and-conquer* approach.

As an example, consider an algorithm that calculates the n^{th} Fibonacci number F_n recursively. It begins by creating processes to calculate F_{n-1} and F_{n-2}. Each of these creates further processes to calculate their components. Figure 3.2a illustrates the structure of this decomposition. Although this algorithm is an inefficient way to perform the calculation since much of the computation is redundant, it highlights that recursion is a useful way to create large amounts of parallelism; *parallel recursion* will be exploited for this reason later in Chapter 8. A more-efficient recursive algorithm, for example, is *mergesort* which successively divides a list into smaller sub-lists until they contain only one element. Sub-lists are then combined in order to produce a single sorted list in a total number of steps, logarithmic in the size of the list. Figure 3.2b illustrates this.



(a) calculating the 6th Fibonacci number



(b) mergesort

Figure 3.2.: With a recursive decomposition of a problem it is successively divided into smaller independent sub-problems, until these can be solved directly. The solved sub-problems are then successively combined to solve the main problem. In (a) the $n + 1^{\text{st}}$ Fibonacci number is calculated by summing the n^{th} and $n - 1^{\text{th}}$ numbers and $F_1 = F_2 = 1$. In (b) a sequence of numbers can be sorted by successively diving it into smaller sub-sequences until they contain only one number. These are then successively *merged* to obtain a sorted sequence.

An important class of algorithms that are based on recursive decomposition are *branch-and-bound algorithms* [GC94]. These are used to solve combinatorial optimisation problems that have a large solution space. They explore this space looking for feasible solutions, but since the space is too large to exhaustively search they employ a means of identifying regions of the space that lie outside a range of best current feasible solutions so that they can be ignored.

Functional

With *functional decomposition* (or conversely *functional composition*), a problem is decomposed (respectively composed) into a number of components with different behaviours. Functional decomposition into a set of parallel processes will only yield performance benefits proportional to the number of components. It is however, an important tool in managing the complexity of a program by separating the functionality of different components and mediating their interactions with minimal interfaces. For example, when two different operations must be performed on overlapping portions of data, it is difficult to combine these sequentially but they can easily be separated to be executed in parallel [JG88].

The use of functional decomposition and the role of programming abstractions is discussed later in §3.2.1 [p. 25].

3.1.2. Parallel efficiency

A key reason to employ parallelism in execution is to reduce the running time. With p processors, one ideally wants a factor of p reduction in running time. This is the same as in sequential computation, where a factor f increase in processor clock speed can deliver up to a factor of f improvement in execution time. However, the movement of data between processors constituting communication, incurs a latency relating to its transmission. This can cause processors receiving data to become idle with the effect that the resulting speedup of the system moves away from p. For the effectiveness of a parallel program to be judged, it is therefore necessary to quantify these aspects of parallelism.

Kruskal et al. argue the performance of a *parallel algorithm* (which are the constituents of parallel programs) is judged with respect to the best-known sequential algorithm for the same problem [KRS90].

Definition 3.1 (Parallel speedup)**.** For a given problem with input size n, let $T_s(n)$ denote the sequential running time of the best known algorithm and $T(n)$ denote the parallel running time. Then the *parallel speedup* S is the ratio between T_s and T:

$$S(n) = \frac{T_s(n)}{T(n)}$$

The speedup indicates the improvement in performance due to parallelism. However, a reduction in running time requires an investment in processors. A speedup of S requires at least S processors, but due to inefficiencies it will be more. Speedup alone does not therefore capture how well the available processors are utilised.

Definition 3.2 (Parallel efficiency)**.** The *parallel efficiency*, E, of an algorithm is measured by how well each processor is utilised, the ratio of speedup to the number of processors used. Let $p(n)$ denote the number of processors, then

$$E(n) = \frac{S(n)}{p(n)} = \frac{T_s(n)}{p(n)T(n)}$$

With an *ideal* parallel algorithm, there is no inefficiency due to communication and idling overheads, and a maximal speedup is obtained with an efficiency of one. Since any parallel algorithm must necessarily perform some communication there will be some level of *inefficiency* with the consequence that $S(n) < p(n)$ and $E(n) < 1$. In general, an algorithm is considered efficient and therefore *scalable* when the efficiency is a constant independent of p and n.

It is interesting to note that parallel efficiency, as defined in Definition 3.2, implicitly assumes that processors are a scarce resource in a parallel system and the most effective use of such a system is with their full utilisation. This holds in general for large-scale systems such a supercomputers but as the cost of a processor diminishes, relative to other system components such as the interconnect or in terms of power, then alternative characterisations of efficiency will be required that are taken with respect to these factors.

Obtaining high efficiency

Each processor in a parallel machine may be performing some computation or have no useful work to do and be *idle*. A perfect speedup is obtained only when all processors are fully utilised and engaged in local computation. Therefore, to make a good utilisation of a parallel machine, the time in which processors are engaged in communication or spend idling must be minimised.

The following points outline several ways that machine utilisation can be improved. Each one could potentially be provided by a programming language automatically, or implemented as a programming approach.

- *Granularity*. Decomposition of a problem should aim to expose as much parallelism as possible, but for the resulting algorithm to execute efficiently it must make a good utilisation of the machine that it is running on. This is related to the program's *granularity*, the amount of work that each process performs, since creating and terminating parallel processes and performing communication adds overhead to a computation. If there is less work in a process than the cost of initiating it or the cost of communication outweighs the computation that takes place as a result, then it is more economical to execute it sequentially.

  The granularity of a parallel program can be adjusted to match that of a particular machine by *serialising* its execution. This can be done manually by expressing it as the combination of parallel and sequential components or automatically by the compiler by combining groups of processes. The amount of parallelism can then be decreased (respectively increased) and the amount of work performed by each sequential process increased (respectively decreased).

- *Latency hiding.* When a communication requires a response, the sending process will have to wait until the response is received. The duration of this wait is related to the *communication latency.* The processor that is executing this process will idle if it has no other work that it can perform. However, by maintaining a small set of processes it can choose a different one that is not blocked by a communication and execute it, thereby *hiding* the latency associated with the communication. This approach is often referred to as *overlapping computation and communication.* Asynchronous or non-blocking communication operations achieve the same effect.
- *Load balancing.* Computational load must be evenly distributed over a machine so that processors remain busy and do not idle. Many computations, such as BSP and data-parallel algorithms, proceed in a sequence of phases with synchronisation performed at the end of each phase. The length of the phase is determined by the slowest participant, and during the time taken for it to complete, every other process is idle. When this gap is large, there is significant scope for inefficiency. This can only be avoided if the execution time of each process is predictable and if it is not then a more-adaptive structure must be employed.

  Load balancing is also an important issue for computations that evolve in an unpredictable or irregular way. An effective and scalable method is *work stealing* where processes maintain a queue of work to perform and when this becomes empty, 'steal' work from queues of neighbouring processes, thereby diffusing work over the system.
- *Redundant recomputation.* The costs of communication can be avoided entirely if it is possible simply to recompute a particular result locally, rather than being sent it. This is only economical if the cost of recomputing is less than the communication.
- *Local operations on data.* Communication can also potentially be reduced significantly by moving a computation to operate locally on data, rather than moving or copying all of the data to be operated on. This is because, in general, the size of a computational procedure will be significantly less than the data on which it operates. In a *data-driven* computation the execution of a parallel operation over a dataset is scheduled on the basis of the data layout, rather than the ordering of parallel operations.

3.1.3. Algorithmic paradigms

An *algorithmic paradigm* is a concept that underpins a class of algorithms [Flo79]. These are commonly employed high-level methodologies that define the overall behaviour and pattern of communication in a collection of processes. There are a number of familiar paradigms in sequential programming, such as *object-orientated*, *functional* and *event-driven.* It has also been observed that a relatively small set of paradigms underpin many effective parallel programs by providing simple ways to express high degrees of parallelism. These include regular arrangements composed of similar processes, to express domain or recursive decompositions. Other paradigms characterise the natural flow of data internally or externally to a computation or the interface between different functional components.

A number of different characterisations of a set of parallel-programming paradigms have been proposed, for example, with Kung's computation models for systolic arrays [Kun88a], Cole's *Algorithmic Skeletons* [Col89], Brinch Hansen's *paradigms for computational science* [Han95b] and derived from Brinch Hansen's paradigms, the *Berkeley Dwarfs* [ABC+06]. Other work has built on these ideas to integrate them into a software engineering discipline, for example, in the shared-memory programming models of Mattson et al. [MSM04].

Enumeration of these paradigms is important since they represent a range of programming styles that provide both valuable guidance for the design of a programming language and benchmarks for a general-purpose parallel architecture. This section outlines the five general paradigms, that occur in a variety of parallel programs from the spectrum of computing applications from computational

Paradigm	General characteristics
shared memory	data decoupled from computation, arbitrary access patterns
process structures	synchronised point-to-point communication in fixed patterns
data flow structures	synchronised unidirectional point-to-point communication in fixed patterns
task farms	independent tasks with unpredictable run times
event handlers	dealing with events that occur at unpredictable times

Table 3.1.: A summary of the different parallel-programming paradigms.

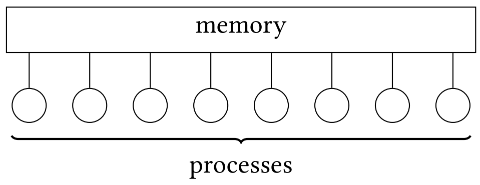


Figure 3.3.: An illustration of the shared-memory paradigm in which a collection of processes are able to randomly access a shared-memory space. This is similar to the PRAM model of computation.

science to embedded computing. They also serve to capture the existing characterisations mentioned above. The following five paradigms are summarised in Table 3.1.

Shared memory

With a *shared memory*, a number of processes are able to concurrently access an area of storage. There is no explicit communication, but data can be passed between processes by writing to and reading from shared locations. This is the form of the *parallel random access machine* (PRAM) model of computation, which was explained in §2.2.2 [p. 15]. Figure 3.3 illustrates the structure of a shared-memory computation.

Shared memory is particularly useful for expressing data-parallel computations where a collection of synchronised processes operate over a region of memory and for expressing computations in which the access pattern is arbitrary. Furthermore, shared memory provides an abstraction from the distribution and management of data over physical storage locations, and a natural separation of data from a computation. A consequence of this however is that there is no data locality.

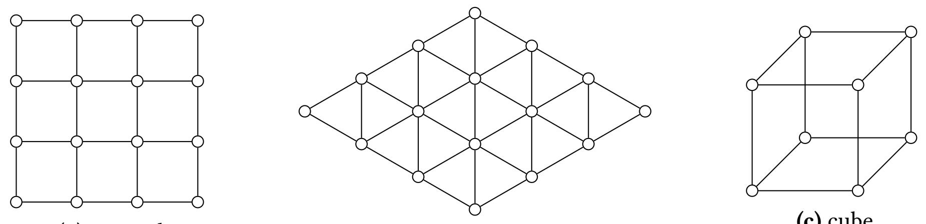
Process structures

A *process structure* is a collection of message-passing processes connected by communication channels. The operation of a process structure is based on localised processing and communication; the data for a problem may be distributed amongst the component processes and the computation proceeds as a sequence of local computations and locally synchronising message exchanges.

A *regular* communication structure is one where the connections follow a pattern. Common examples of regular process structures are pipelines, low-dimensional grids, trees and hypercubes [Lei92]. Figure 3.4 illustrates some of these. An *irregular* communication structure is one where the connection pattern is not defined by a pattern. For example, some problems in computational science are based on unstructured meshes. However, it is important to note that regular structures are essential in formulating large numbers of processes in a scalable way [MC80, Ch. 8], [May88].

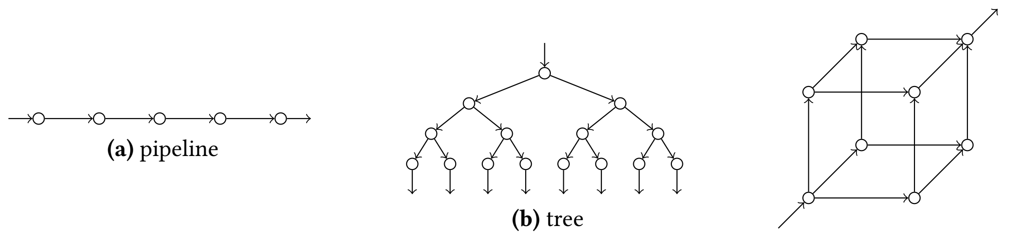
Data-flow structures

A *data-flow structure* is similar to a process structure except that communication channels are *directional* and data is 'streamed' from one end to another. Each component process receives some data on input channels, performs a local computation and outputs some data. Data-flow structures



(a) 2D grid **(b)** hexagonal array **(c)** cube

Figure 3.4.: Regular process structures where processes are connected with bidirectional message-passing channels according to a simple pattern.



(a) pipeline **(b)** tree **(c)** cube

Figure 3.5.: Regular data-flow structures similar to those in Figure 3.4, except that communication links are directed and additional links are needed to source and sink data to and from the network.

are suited to the implementation of data-parallel computations for signal, image and audio processing, especially in an 'on-line' fashion when data is being generated in real-time from an input such as a sensor. Figure 3.5 shows some example regular data-flow structures.

Regular data-flow structures where communication is globally synchronised are known as *systolic arrays* [Kun82], deriving from the similarity to the human circulatory system. They were originally developed for specialised *very large scale integration* (VLSI) architectures with global clocking and have proven to be an effective way to implement a broad class of algorithms algorithms [Kun88b]. When operating asynchronously, each process is driven by local pairwise communications. This is referred to as a *wavefront array* [Kun84] and this is the natural way to implement a systolic array with message passing.

Task farms

A *task farm* is used to solve problems that can be decomposed into a number of sub-problems that can be solved independently (so-called *embarrassingly parallel* problems). It consists of a *farmer* process and a number of *worker* processes. The farmer maintains a queue of outstanding work to be performed and assigns jobs to the workers when they are idle. Results from completed work might be returned back to the farmer or output to a shared structure. Figure 3.6 illustrates a simple task farm. For more complex problems, a number of farmers might be organised in a hierarchy to provide different processing stages [Hey90].

Task farms provide a simple mechanism for balancing the computational load between workers, which operates in a similar way to work stealing (see §3.1.2 [p. 21]). This allows task farms, in many cases, to obtain near optimal speedups (particularly when the amount of work is large with respect to the distribution of work and combination of results that is performed by the farmer) i.e. for a problem size n, when $E(n) \sim 1$. In computations where the time each sub-problem takes to solve can be unpredictable, using a synchronised algorithm will cause many of the component processes to idle between synchronisations, resulting in a poor processor utilisation.

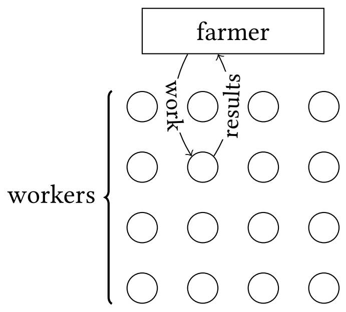


Figure 3.6.: In the task farm paradigm a 'farmer' process maintains a queue of outstanding items of (independent) work and distributes them amongst a number of workers based on their activity. This provides a natural mechanism for load balancing.

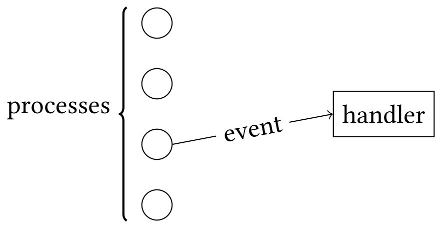


Figure 3.7.: An event handler is a process that reacts in response to a particular event or set of events, performing some action when they occur. Events may be caused by internal components of the program or from external interaction.

Event handlers

An *event handler* deals with a processes that interact at unpredictable moments. It may provide an interface with external input or be components of the program that have unpredictable behaviour. Figure 3.7 illustrates the event handler paradigm.

Composition

Many programs will naturally be expressed as a combination of the above five paradigms. The following outline potentially useful compositions.

- *Shared data structure.* A shared data structure implemented as a shared memory in composition with a computational program component. This could be a collection of independent processes, a task farm or a process structure. Furthermore, the program component could itself be composed a *sequence* of parallel and even sequential components.
- *Message-passing functional composition.* Message-passing process or data-flow structures can be combined in parallel in a functional composition to create more complex program structures.
- *Embeddings.* Any paradigm can be employed as part of a *parallel subroutine* by a process. Since a process might be a component of another structure, this can be seen as an *embedding*.

3.2. Criteria for a general-purpose parallel-programming language

A *programming language* is a notation for human beings to express a computation that can be compiled to execute on a machine, or class of machines. It provides to a programmer an *abstraction* of the underlying architecture, hiding details that are not directly relevant to the specification of a computation [Hoa73] (this is the language abstraction that was illustrated in Figure 1.1). This abstraction typically involves dealing with the complexities of resource management, such as memory and processors. The choice of this abstraction depends on obtaining a balance between what provides the programmer with the most expressive power and what can be effectively compiled to run efficiently.

This section proposes the following criteria for a general-purpose parallel programming language and discusses the requirements of each one. For clarity in the context of this thesis, the criteria do not include other potential criteria such as syntactic issues, formal verification, debugging or sequential features.

$$\left.\begin{array}{l}\text{1. support for programming abstraction;}\\ \text{2. support for data abstraction;}\\ \text{3. support for general parallelism;}\end{array}\right\}\text{expressiveness}$$

4. execution efficiency.

The criteria labelled with '*expressiveness*' enable the language to support the simple expression of programs and are at odds with the execution efficiency. These criteria provide a basis for the survey of programming approaches in §3.3 [p. 28] and motivate the design of sire and its compilation scheme.

3.2.1. Expressiveness

Support for programming abstraction

The ability of a programming language to support the construction of abstractions is fundamentally important [DDH72] since programs in general are complex and cannot be understood all at once. Programming abstraction is based on the principle of *compositionality* that states the meaning of a whole is determined by the meanings of its constituent parts and the rules used to combine them [Pel94]. With this, a program can be constructed as composition of components, in a layered hierarchy, where the *internal* behaviour of each component is simple enough to be understood in isolation and the combined effect of their *external* behaviour constitutes the whole [Han77].

The importance of abstraction in programming was recognised by Turing in the 1940s and his design for the Automatic Computing Engine (ACE), which included support for *subsidiary operations* [Tur46], or *subroutines* as they are now known [Whe50].

The concept of abstraction to reduce complexity depends on the representation chosen for something adequately characterising it. If it does not, then additional details of the object in question have to be considered, thereby breaking the principle of compositionality. When this happens, the abstraction provides no significant reduction in complexity. For example, if a program component behaves differently in combination with other components than it does in isolation, then the internal details of the component must be explicitly considered and the abstraction is lost.

It is also interesting to note that the machine abstraction provided by the basic computational model can also be broken if a programming language contains *machine-orientated* features. In this case, the behaviour of a program will not be understandable only in terms of the language, and therefore the details of the compiler and the machine also have to be considered [Han77].

Support for data abstraction

Data are simply values belonging to a set, but in a computation it is necessary to structure them in way that allows them to be accessed and manipulated. The storage of data items is typically based on a single contiguous and randomly-accessible memory; a high-level structuring of the data, such as a queue, list or tree, must be mapped to this. In a distributed machine, the mapping must also correspond to each distinct memory.

A *data structure* therefore consists of two components:

- a representation of the data;
- a set of *basic* operations that allow the representation to be manipulated.

Following Hoare's characterisation of a data structure [DDH72, Ch. 2], these operations are basic in the sense that their implementation is heavily dependent on the chosen representation, and a representation is generally chosen to minimise the amount of storage and permit efficient basic operations. The number of basic operations that should be provided by a data structure is arbitrary, but the guiding principle is that their capability should be sufficient that any other operation can be defined in terms of them.

It is generally regarded as good programming practice to separate the details of a data structure implementation from the abstract properties of the interface it presents to the program, sometimes called its *data type*.[1] This is due to the same reasons that were explained in the previous section (§3.2.1 [p. 25]), because it reduces complexity and allows improvements or even substitutions to be made to the data structure that respect the same external interface.

Data abstraction therefore relates to the way that data structures are separated from computational structures. For distributed systems (as opposed to von Neumann-based shared-memory ones) a data representation must specify a distribution over a collection of memories with mappings between each data element and processor and memory location. Finding a good trade-off between high-level notations to deal with this and efficiency of an implementation has proven to be difficult; various approaches are surveyed in §3.3 [p. 28].

Support for general parallelism

A general-purpose language must support the expression of a wide class of parallel programs. Given that a small set of paradigms underpin a broad class of programs (a characterisation of these were outlined in §3.1.3 [p. 21]), it is essential that the expression of these paradigms and their composition with one another is made convenient [Flo79].

3.2.2. Execution efficiency

The issues relating to the simple expression and comprehension of programs, captured in the previous criteria, compete to a large degree with the efficiency, in terms of the program size and execution time, that can be obtained with the corresponding compiled machine code. A language that ignores the issue of machine efficiency may rely on sophisticated compiler technology to deal with this automatically, but there are serious disadvantages to taking this approach [Hoa73]:

- the complexity of the compiler will require a significant period of time for its development and verification, and the resulting program will be large and execute slowly;
- the translations and optimisations employed by the compiler will likely exhibit pathological behaviours in some circumstances, which can not be predicted by the programmer without a knowledge of the compilation, and will result in inefficient execution;
- different compilers may vary in their approaches, with the consequence that the programmer has no control over the efficiency of their program.

A language that can be compiled in a straightforward manner to a target machine architecture will avoid these problems since it narrows the scope of implementation options for compilers and reduces their complexity. This allows compilation to produce compact and efficient executable programs, providing the programmer with a *transparent execution model* and responsibility for the performance of their program. Furthermore, if the language has clear, well-defined semantics, then it may be possible to implement particular features by translating them to a canonical form in terms of a smaller set of language features, or to apply optimisations with transformations. This can be performed independently of any machine with the results understandable to the programmer.

A balance must be struck between the competing aspects of the expressive power of a programming language and the use of a simple compilation strategy to produce efficient executable programs. It is highly unlikely that both can be achieved without support from the underlying machine architecture. A key argument in this thesis is that a parallel architecture must support an effective machine

[1]It can be argued that object-orientated programming is an exception to this, since a class is designed to encapsulate both the details of the implementation of a data type and computational procedures associated with it. However, this style of structuring is usually applied at a higher level, and standard data structures such as lists and tables are implemented as classes in their own right.

abstraction so that the implementation and use of programming languages is relatively simple, for all of the reasons outlined in this section.

Resource management

The key concern with the abstraction provided by sequential programming languages is to hide the management of memory but in general, the greater the degree of automatic memory management, the less efficient the execution because of the additional time spent executing the management routines.

The least dynamic schemes, where memory is allocated at compile time for static variables and stack frames, have the benefit of efficiency and determinism. Memory allocation performed at run time incurs significant performance overheads because the state of memory must be maintained with internal data structures. Automatic deallocation such as *garbage collection* causes further inefficiency that is not under the programmer's control.

The use of memory allocation mechanisms depends on the application and limitations of a machine. For 'fast' conventional sequential machines with large memories, the performance overhead of highly dynamic languages that leave everything to the run time can be considered an acceptable price to pay for their ease of use and resulting programmer productivity. In contrast, for embedded systems with slower processors, limited memory and applications with real-time requirements, predictable behaviour is essential.

In contrast with sequential computation, parallelism introduces a richer set of resources with collections of processors, memories and communication channels. A computation requires the creation and termination of threads of sequential execution and their activities to be coordinated with communication and synchronisation. This must be mapped to and scheduled on the target machine either explicitly by the programmer, implicitly by the compiler or at run time. Since a programmer is interested primarily in introducing parallelism to improve performance,[2] the management of these resources is central to the abstraction provided by a parallel-programming language and its ability to support programming abstractions.[3] The management of resources for parallelism is analogous to memory management in sequential machines.

The greater the level of abstraction provided by a parallel-programming language, the less a programmer is concerned with the low-level details of a parallel machine. There is a trade-off between the level of abstraction and the performance and predictability of programs expressed with it. As the level of resource management that is hidden from the programmer increases, then the extent to which it must be dealt with at run time increases accordingly.

3.2.3. Summary of criteria

The following summarise the criteria for a parallel programming language.

1. *Support for programming abstraction*: abstractions are fundamentally important to programming since they allow a program to be constructed as a composition of parts, where only the *external* behaviour needs to be considered, allowing the details of the implementation to be ignored. Parts may be subroutines, modules or libraries.
2. *Support for data abstraction*: separating the details of a data structure implementation from its abstract interface yields the same benefits as the previous criteria, but it depends on an ability to effectively separate a representation of data from other components.
3. *Support for general parallelism*: a general-purpose language must make convenient the expression of a wide class of parallel programs.

[2]They might also be concerned about reducing power consumption by executing more tasks in parallel at a slower clock frequency to exploit the quadratic relationship between voltage and power.

[3]It is interesting to note that dynamic resource allocation in parallel machines can introduce the opportunity for deadlock and non-determinism. Deadlock results from cyclic resource dependencies and non-determinism can result from the insufficient resources because it is dependent on the scheduling of processes and order of allocations and deallocations.

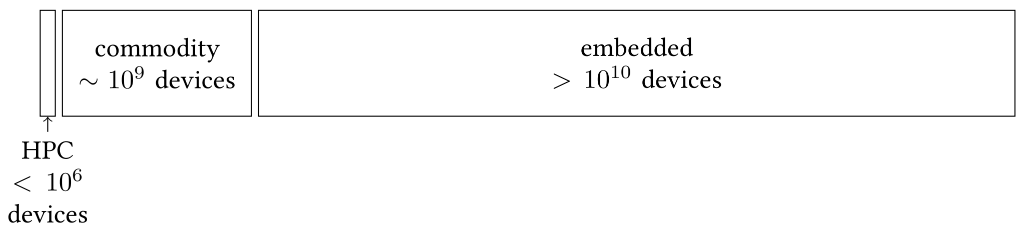


Figure 3.8.: The spectrum of computing with its three main application areas. A first-order approximation of the total number of devices worldwide is given for each one to emphasise their relative sizes.

4. *Execution efficiency*: a language that provides an effective abstraction of the machine architecture enables it to be compiled in a simple way to execute efficiently and predictably. This provides a transparent execution model, placing the programmer in control of the performance of their program. The issue of efficiency is related primarily to the management of resources such as processors, components for communication and memory.

3.3. Survey of programming and compilation approaches

This section surveys existing parallel-programming approaches and compilation techniques, with the main emphasis on those that are *scalable* i.e. distributed and non-von-Neumann. This excludes many *multithreaded* parallel languages since they target shared-memory multiprocessors and employ centralised resources; it is therefore unlikely that they can be implemented and executed efficiently with distributed memory since communication will become concentrated around a single processor. For example, Cilk [BJK+95] and OpenMP [CMD+00] are considered mainstream programming approaches but their implementations target shared memory and employ a centralised thread scheduler.

The survey is representative of the main approaches taken in scalable parallel programming, but does not attempt to be exhaustive. Table 3.2 provides a summary of the approaches discussed and their distinguishing features.

3.3.1. Application areas

It is useful to first outline the main areas of computing and their defining characteristics, since these have a large bearing on the design of the programming languages and compilation schemes employed for them. Figure 3.8 illustrates this *spectrum* of computing devices comprising at one end, *large-scale high performance systems*, of which there are of the order of hundreds of thousands worldwide, to *small-scale embedded systems*, of which there are billions worldwide. Lying between these are (of the order of hundreds of millions of) *commodity* devices such as desktops and servers.

High-performance computing systems

HPC is the domain of large-scale systems. HPC systems that are used to run single programs are referred to as *supercomputers*. These are at the forefront of absolute computing performance and have necessarily had to employ parallelism and high-performance interconnection networks to achieve this. Due to the scale of supercomputers systems, the only feasible physical memory architecture that they can use is distributed. Modern systems for instance, are vast arrangements of tens or hundreds of thousands of processors that occupy hundreds of square metres of space. Consequently, supercomputing has historically been the area in which distributed parallel programming languages have received the most attention.

Supercomputers are used for computationally intensive tasks such as simulation and modelling in science and industry. They share the following general characteristics:

- the amount of memory per processor is large;

- the communication bandwidth is high but, in general, the latency of establishing communications is also high, restricting communication to large messages.

There are also some general characteristics of supercomputer applications:

- the computation is highly parallel;
- the problem size is large and in some cases can be scaled arbitrarily, such as in simulations where further accuracy is always desirable;
- there is no external input or any real-time constraints, computations are launched and results are returned in a number of hours or days later.

Embedded systems

An *embedded computing* system is designed to perform a particular function as part of a larger system. They typically employ general-purpose processors, in the sense of being able to execute general-purpose programming languages, due to the expense of producing *application-specific integrated circuits* (ASICs) and the programming challenges associated with *field-programmable gate arrays* (FPGAs). Embedded systems are used, for example, for control tasks, signal processing, and controlling user interfaces in systems such as robots, consumer electronics and medical equipment. The main characteristics of these systems are:

- a small physical form factor;
- limited power, particularly when powered by a battery, and consequently a slow clock frequency;
- limited memory;
- a requirement to interface with external input or output, which necessitates real-time performance constraints.[4]

Commodity systems

Commodity computer systems are intended for general use and comprise the broad area of *consumer devices*, such as desktop and laptop personal computers, and *server devices* that are used to provide resources such as web pages, email or storage. Although server devices are typically deployed in large numbers in data centre-type facilities, they are used to run many independent jobs and therefore do not require the same degree of intraconnectivity that supercomputers do. Commodity computing systems employ shared-memory architectures with SMPs and increasingly, attached GPU devices where, in general

- the amount of shared memory and caching is large;
- processors are run at a fast clock speed.
- their real-time requirements are typically soft, of the order of milliseconds and related to responding to user input.

It is interesting to note that general-purpose ‘desktop’ applications, such as word processors, web browsers and games obtain an average speedup from parallelism of around 2, which has not changed significantly in the last 10 years, despite there being a great deal more parallelism available. In contrast, domain-specific applications such as video transcoding make a high utilisation of commodity devices [BDMF10]. However, these systems make significant use of task-level parallelism to provide multitasking when a number of (independent) programs execute concurrently. This approach increases the aggregate computational throughput.

[4] Real-time constraints may be *hard* if the behaviour of the system is heavily dependent on internal deadlines, where the system may crash or become dangerous if they are not met; *soft* constraints on the other hand relate to an acceptable degradation in the behaviour of the system.

Name	**Citation**	**Appeared**	**Paradigm**	**Area**	**Resource management**
			Communicating process		
Occam	[INM84, INM88b]	1983	message passing	Em/GP	compile time
Linda	[Gel85]	1985	shared memory	GP	dynamic
Joyce	[Han87]	1987	message passing	GP	dynamic
Strand	[FT90]	1989	declarative	GP	dynamic
CA	[CD90]	1990	OO, global name space	GP	dynamic
LUSTRE	[HCRP91]	1991	data flow	Em/GP	static
Ease	[Zen92b, Zen92a]	1992	shared memory	GP	dynamic
PCN	[FOT92]	1992	declarative	GP	dynamic
CC++	[CK93]	1993	OO, global name space	GP	dynamic
SuperPascal	[Han94]	1994	message passing	GP	dynamic
Fortran M	[FXA94]	1995	message passing	GP	dynamic
Charm	[KRSG94, RSSK94]	1995	OO, global name space	HPC	dynamic
StreamIt	[TKA02]	2002	data flow	Em/GP	static
Occam-π	[WB05]	2005	message passing	GP	dynamic
XC	[Wat09]	2005	message passing	Em	compile time
			Functional		
ZAPP	[BS81, MS87, MS88]	1981	functional	GP	dynamic
SISAL	[MSA$^+$83]	1983	functional	GP	dynamic
SCL	[DGTY95]	1993	functional	GP	dynamic
NESL	[Ble95]	1995	data parallel, functional	GP	dynamic
			Communication libraries		
PVM*	[BDG$^+$91]	1991	message passing	HPC	dynamic
SHMEM*	[BK94]	1994	distributed shared memory	HPC	static
MPI*	[WD96]	1996	message passing	HPC	static
GASNet*	[Bon02]	2002	distributed shared memory	HPC	static
			Data-parallel		
CM Fortran	[BHMS91]	1991	data parallel	HPC	static
Fortran D	[FHK$^+$90]	1991	data parallel	HPC	static
Vienna Fortran	[CMZ92]	1992	data parallel	HPC	static
HPF	[KLS93]	1993	data parallel	HPC	static
ZPL	[CLC$^+$98]	1998	data parallel	HPC	static
			PGAS		
Global Arrays*	[NHL94]	1994	data parallel	HPC	static
Co-array Fortran	[NR98]	1998	data parallel	HPC	static
UPC	[CDC$^+$99]	1999	data parallel	HPC	static
Titanium	[YSP$^+$98]	1998	data parallel	HPC	static
X10	[CGS$^+$05]	2004	shared memory	HPC	mixed
Fortress	[ACH$^+$05]	2005	shared memory	HPC	mixed
Chapel	[CCZ07]	2007	shared memory	HPC	mixed

Table 3.2.: Summary of parallel-programming languages and libraries (denoted by '*') discussed in the survey. The following abbreviations are used: general purpose (GP), embedded (Em) and object-orientated (OO).

3.3.2. Communicating process programming

This survey begins with programming models that are based on *communicating processes*, and in particular the occam programming language, because occam is the basis of the proposed sire language.

In general, a communicating-process model considers processes and interaction to be the most important abstractions in computation [Han90] and different approaches are characterised primarily by the way in which communication is performed, which can be message passing, data flow, remote procedure calls or through shared data.

Occam

Occam is a general-purpose programming language [May83, INM84] that was developed at INMOS in the 1980s. It was based on an earlier language *EPL* [MTWS78] and the principles of CSP. Occam takes a minimal approach with the smallest set of features that were adequate for its purpose.[5]

Occam programs are expressed as hierarchical collections of processes, which can be nested, composed in sequence or composed in parallel. Processes executing in parallel are connected by message-passing point-to-point channels, which can be named and used as procedure parameters, thereby providing a mechanism for abstraction. Its message-passing approach and *alternative operator* for dealing with non-determinism (based on that of CSP) allows the expression of a variety of forms of parallelism, in particular, process and data flow structures, task farms and event handlers, which can all be combined arbitrarily.

A key principle of occam, adopted from CSP, is *scheduling invariance*, or the absence of race conditions. An occam program (that does not make explicit use of non-determinism with alternatives) executes deterministically, no matter how the execution is scheduled. This is to simplify the construction of parallel programs. Scheduling invariance is enforced with *process disjointness* with the following rules:

1. by ensuring no shared variable can be updated or shared channel can be output to;
2. by synchronising channel communication so messages are always guaranteed to be received;
3. by allocating resources at compile time, which includes processors, communication channels and memory, thereby avoiding any run-time allocation.

The simplicity of occam, combined with well-defined semantics and control of side-effects, gives it algebraic properties [RH88], permitting formal verification of correctness and transformations to convert programs to more convenient equivalent forms. Furthermore, the economy of features and the careful choice of their behaviour allow it to be compiled in a simple manner to execute efficiently. It was developed alongside the INMOS *transputer* architecture [INM88c], which was intended to be a natural target for it, but it can also be implemented efficiently on different architectures and even used as a hardware description language [May91]. Later versions of occam preserved its main aspects but extended it with features, particularly in the type system, to support better software engineering practices.[6]

Occam provides good support for programming abstraction and general parallelism while maintaining a simple compilation process and efficient execution, these fulfilling most of the criteria outlined in §3.2 [p. 24]. However, because it does not provide any mechanisms for sharing, occam can not be used to express shared-memory structures which are important when the access patterns are unknown and, in general, sharing is essential for developing data abstractions and higher forms of abstraction.

[5]It employs the principle of *Ockham's razor* (after which it was also named) which was developed by William of Ockham, a 14th century Philosopher.

[6]Occam 2 [INM88b] and 2.1 [SGS95] were the main revisions, but another version, occam 3 [Bar92], which was actually developed shortly after occam 2 but never implemented, proposed a number of features for expressing program modules and libraries. This included a client-server type notation with *call channels* that provides a remote procedure-call mechanism.

Occam variants

Some of the main criticisms of occam are that it is too low-level and too restrictive, requiring the programmer to deal with resource allocation and communication. Apart from mechanisms for sharing, these were mainly related to an efficient distributed implementation. A number of languages have attempted to relax some of the restrictions, in particular, by adding recursion and permitting dynamic sizing of arrays or collections of processes. These include *Joyce* [Han87], *SuperPascal* [Han94], an experimental version of occam with recursion [Dav00], which was based on the same scheme used in the implementation for SuperPascal [Han95a], and *occam-π* [WB05], which adds further dynamism inspired by the π-calculus with process and channel mobility. However, these approaches are all based on a multithreaded implementation with shared memory and can therefore employ conventional techniques for dealing with this dynamism. Recursion and dynamic processor allocation remain difficult mechanisms to implement efficiently on a distributed architecture.[7]

The *XC* programming language [Wat09] is based heavily on occam with C-style syntax and designed to exploit the *XMOS*[8] *XS1 architecture* [May09], which is designed for use in embedded systems. XC and the XS1 architecture bear many similarities to occam and the transputer but the most significant difference is the ability to support efficient input and output on chip IO pins, and deterministic execution with high-level programming. This approach enables programs to be expressed that can implement behaviour usually associated with hardware, such as digital communication protocols, giving it an advantage over the conventional approach of using a hardware description language and an FPGA or custom ASIC.

Data-flow languages

A *data-flow* computation corresponds to the data-flow structure paradigm described in §3.1.3 [p. 22]. In the same way a data-flow structure is a special case of a process structure, a data-flow computation can be viewed as a special case of message passing when all communication channels are directed. Data-flow languages have a rich history [WP94] and have been employed, for example, in embedded systems for *digital signal processing* (DSP) and for hardware design. They bear similarities to functional languages due to their absence of side effects and consequent implicit parallelism. The languages *LUSTRE* [HCRP91] and *StreamIt* [TKA02] are described as exemplars of this programming approach.

LUSTRE is a declarative data-flow language designed both for embedded control systems that must react quickly to events and for describing hardware. A program is expressed as a composition of functions, called *nodes*. The input and outputs of the functions define the flows of data. LUSTRE is a *synchronous* data-flow language, meaning in that the input and output rate of each process is known at compile time and all computation and communication can be scheduled statically. This gives the programmer control over temporal aspects of the program, which is essential for real-time applications. Since LUSTRE programs are mainly used for control, the need for distributed execution arises from the coordination of events in different locations [CG95].

StreamIt is an imperative language in which asynchronous data-flow computations are expressed as a collection of sequential processes, called *filters*. Filters are joined by *streams* and are composed with constructs to *split*, *join*, *feedback* and *pipeline* streams. StreamIt programs are compiled by converting them into a data-flow graphs, which are then mapped to a target architecture to balance computational load [IWM+02].

[7] Parallel recursion using shared memory requires stack frames to be allocated dynamically according to a tree-structure, which is a significant overhead compared to conventional compile-time stack allocation. This is how the multithreaded programming languages Cilk [BJK+95] and Go [Goo13] are implemented. It also introduces the possibility of non-determinism since the behaviour of a program that exceeds the stack capacity will be dependent on the ordering of the stack frames. One way to avoid this problem is to distribute recursion over memories so each processor has at most one recursive process [Wel92].

[8] A company founded by a group of ex-INMOS employees around 2005.

The problem with data-flow languages in general is that they only support one parallel algorithmic paradigm, making them appropriate for only a narrow range of problems.

Languages supporting composability

It has been observed that there is a general lack of support for composability and abstraction in parallel programming [FKT90, Fos96]. *Strand* [FT90], *Program Composition Notation* (PCN) [CT89] and *Fortran M* [Fos95] have been developed to address this issue specifically (occam and its variants do also support composition and abstraction but have already been mentioned).

Strand has roots in parallel logic programming and consequently a declarative style with single assignment variables. These variables are globally accessible and are used as a means of communication and synchronisation between processes. This approach is taken to prevent race conditions from concurrent writes to the same variable, thereby enforcing process disjointness and determinism. It also supports distributed execution directly with language notations to specify the mapping of processes to processors, but this depends on the structure of the computation being known to the programmer.

PCN extends the basic ideas of Strand with a richer syntax including imperative elements and the ability to construct reusable modules, but does not make any substantial improvements [Fos93, Fos96].

Fortran M is an extension to Fortran that draws on the experience of Strand and PCN to support compositional (*M*odular) programming. It introduces processes, message passing, parallel composition, parallel 'loops' and an operator to deal with non-determinism in one-to-many channel connections. These additions are very similar to the core features of occam, although it permits their flexible use with recursion, dynamic process creation and dynamic channel connections. Fortran M also introduces notation to specify the mapping of processes on to processors, which can be performed explicitly or recursively by specifying ranges of processors.

These three languages are interesting because of their approaches to composability but because they are designed primarily for this aspect, they neglect the issues of general parallelism and execution efficiency. Although Fortran M in principle maintains the advantages of occam, its dynamism makes an efficient distributed implementation difficult.

Object-orientated languages

A different form of communicating process languages are those based on an *object-orientated* approach where an *object* defines a entity that encapsulates some state and provides *methods* that can be executed externally to act on it or to perform work. Object orientation provides a natural way to structure a program and build abstractions. It also corresponds well to parallel execution since objects can be mapped to particular processors and communication between objects is achieved with method calls, providing a way to execute a subroutine in a different address space. This mechanism is attractive because it can be implemented simply with a sequence of message exchanges, because it provides clean procedure-call semantics and because it provides the ability to move easily between the local and remote forms of a call. In general, method calls to remote objects are referred to as *remote procedure calls* (RPCs) [BN84] and can be viewed as a simple abstraction of message passing.

Implementation of an object-orientated language on a distributed memory architecture raises the question of how object references are maintained. The following languages all employ a *global name space* in which object references are visible. When the location of objects can be determined at compile time the references can be resolved at the same time, otherwise they must be distributed in some way at run time. It is crucial for performance that resolution of object locations can be performed in a scalable way. One way is to support global name spaces in hardware, the MIT J-Machine for example, did this to support the implementation of parallel object-orientated languages [DFK+92].

Concurrent Aggregates (CA) [CD90] introduces parallelism implicitly from concurrent method executions. It supports hierarchical program structuring and provides a mechanism for data abstraction, called *aggregates*. An aggregate is a structured collection of objects, referenced by a single name, that can be distributed between processors and accessed concurrently. In addition, members of an aggregate can access any other member and call methods on it. New objects can be created dynamically and are accessed in a global shared address space. The idea of distributed collections of objects was integrated with C++ in the programming language *pC++* [BBG+93].

Charm [KRSG94, RSSK94] is similar to CA, based on the dynamic creation of objects in a global, and potentially distributed, *object space*. The location of objects is maintained dynamically by the run-time system, providing global references. Remote method execution is asynchronous and completes in two phases so that latency hiding can be performed. Objects can be created individually or in arrays (like an aggregate) and the placement of objects can be specified or determined automatically by the run-time system, which is also able to perform load balancing. To support regular data-parallel computations, Charm provides a special type of object array that is distributed over the whole system, with one element per processor. This can be used as a simple mechanism for data abstraction.

Compositional C++ (CC++) [CK93] is an extension to C++ that differs to CA and Charm in that it introduces constructs to control parallelism with parallel composition, parallel 'loops' and asynchronous parallelism. Operators for synchronisation and atomic access are provided to facilitate interaction. Objects can be mapped onto specific processors and accessed via a 'global pointer' to execute methods remotely.

These three languages interpret the concept of object orientation in quite different ways and do not provide much support for general forms of parallelism. Moreover, their global name spaces and additional dynamism makes an efficient distributed implementation difficult.

Tuple spaces

A *tuple space* is a globally accessible store, in which tuples containing one or more values can be added removed and updated. Access to a tuple space is associative, depending on a key or pattern. The idea was introduced with the language *Linda* [Gel85], which provided just four operations: read and remove a tuple, read a tuple, write a tuple, and create a new process. Tuples are accessed according to matching rules that are based on the number of elements they contain and communication between processes corresponds to a write with a matching read.

Linda was designed as a *coordination language* to be used in combination with any sequential programming language, as a means of expressing the structure and evolution of the parallel components of the computation [GC92], with tuple spaces providing a simple way to express concurrently accessible data structures [CGL86].

A single global tuple space poses problems for composability since it is difficult to develop components that encapsulate communications. The naming and matching of tuples also makes it difficult to express structured representations of data and communication structures between collections of processes. The implementation of tuple spaces is difficult due to the complexity that arises from matching tuples. A number of proposals for distributed memory machines have been made, such as [Gel85, CJ87, Zen90, DWR95] but they depend on compile-time optimisations based on analysis of tuple types and accesses and hashing schemes with centralised components or broadcasting. The former can lead to unpredictable performance and the latter to poor scalability.

Ease [Zen92b, Zen92a] is a refinement of Linda's tuple space concept to produce a language that is intended to be simpler to implement. It is based on shared data structures called *contexts*, which can be declared and accessed (concurrently) according to block scoping rules. Its implementation is simplified since access to a context is dictated by whether it is ordered or unordered.[9] A special *call-reply* context provides an RPC-style mechanism that can be used to implement servers to provide access to resources. Although Ease allows hierarchical compositions of parallel components with

[9] An implementation of Ease integrated with C has been proposed [MD94] but the access to a context is directed through a single process that has been designated to own it. This makes it easy to implement ordered access, but competition for access to a large data structure will significantly reduce performance.

contexts, it is not made clear in the work how contexts can be used to express more complex data structures.

3.3.3. Functional programming

Functional or *applicative* programming languages are based on the application of functions as a means of computation. A function takes a set of parameters and can return a result but it cannot cause any *side effects*, changing any state of the computation. A computation therefore evolves as a tree-structured collection of processes and transfers of data occur only between parents and children.

The absence of side effects makes functional languages an excellent candidate for parallel execution since independent parts of the tree can be evaluated in parallel. Their execution with a von Neumann architecture is inherently inefficient as all memory accesses are channelled through a single point (the von Neumann bottleneck), despite a high level of independence between them [Bac78]. Functional languages can therefore be viewed as being *implicitly parallel*, hiding all resource allocation issues from the programmer [PJ89]. This creates a broad scope for compilation, in particular for the run-time components since nearly all of the management of resources must be dealt with dynamically.

A weakness of a functional approach is that transfers of data in control of the programmer occur only between callers and callees (parent and child nodes of the execution tree). This restricts the exploitation of locality in parallel computations, particularly computations that are naturally expressed as a process or systolic array, where each process iteratively computes on local data. To achieve a similar effect with a functional description, the state of the array must be passed up and down the execution tree as it expands between iterations.

Compilation of functional languages

The *Zero-Assignment Parallel Processor* (ZAPP) [BS81] architecture is a message-passing run-time system designed as a compilation target for the execution of functional programs. It is based on a simple distributed scheme where a single processor starts executing at the 'root' of the tree and other processors steal tree nodes from adjacent processors and *reducing* nodes that are not dependent on any children. Expansion of the process tree is conducted breadth-first when adjacent processes are underloaded and depth-first when they are loaded. When no work can be diffused into the network, expansion and reduction proceeds sequentially.

An implementation of ZAPP on an array of transputers demonstrated that the scheme can deliver near-optimal machine utilisation for algorithms expressed in a 'divide-and-conquer' form [MS87, MS88]. This depends crucially on a network topology that is both vertex-symmetric, where each processor has an identical view of the network, and well connected, thereby supporting the efficient diffusion of work at an exponential rate, bringing processors into action rapidly.

Similar schemes to ZAPP have also been described, e.g. [HG85], and are generally referred to as *parallel graph reduction* [PvE93] but in general it is difficult to deliver good performance for a broad class of functional programs.

Another way to implement functional languages is to convert them into a data-flow form. This is the approach for compiling the functional language *SISAL* (streams and iteration in a single assignment language) [MSA+83], which creates an intermediate form consisting of an acyclic data-flow graph where nodes denote operations and edges denote transfers of data [FCO90].

Functional paradigms for parallelism

A key observation made in the ZAPP work was that many problems can be expressed in a divide-and-conquer form (written in terms of mapping and combining functions), where they are recursively decomposed into a set of sub-problems and these are then successively combined into a solution. Additional 'functional forms' have been identified [Col89, DFH+93, THLPJ98], in the same sense as the paradigms described in §3.1.3 [p. 21], with common structures underlying different programs.

These forms also include a pipeline, task farm, a data-parallel ‘map’ and a function to perform all-to-all interactions.

By restricting the structure in functional programs to a small set of paradigms, the implementation of each one can be addressed explicitly so that resource management can be dealt with in an efficient way. This is potentially useful for a programmer since the paradigms can be used as ‘building blocks’ that can be combined in various ways to create complex program structures. The paradigms are also amenable to transformations can be used to simplify or convert between paradigms and may even be combined with side-effect free imperative code to define the behaviour of some functions. The programming language *Structured Composition Notation* (SCN) [DGTY95] developed as a means of doing this.

Most notably perhaps is Google’s *MapReduce framework* [DG08], which is essentially based on the divide-and-conquer paradigm. A programmer simply specifies the behaviour of the ‘map’ and ‘reduce’ functions; an implementation of the framework deals with all other aspects of execution and resource management.

A different approach to functional execution of parallel programs is to restrict the form of parallelism. The *Nested Data-parallel Language* (NESL) [Ble95] is a functional programming language that introduces parallelism with data-parallel operations on sequences, which are its only primitive data type. It allows arbitrary nesting of functions and composition in sequence and parallel. The implementation of NESL is simple and based on a number of program transformations to ‘flatten’ nested parallelism and sequences [BSC+93].

3.3.4. Communication libraries

Communication libraries are used with sequential languages such as C and Fortran to provide functions for communicating with messages. Despite their status as libraries, they impose a particular programming model in which a number of identical instances of the program are executed in parallel and communicate with one another, based on a unique identifier or *rank*. This model is known as a *single-program multiple-data* (SPMD) model and is due to the simplicity of compilation and execution, where a single binary is produced and replicated over a machine for execution.

The simplicity of communication libraries and a SPMD model makes programs written in this way simple to understand and compile. This provides efficient and predictable execution as well as portability between systems. However, this approach provides little or no abstraction from the underlying machine in terms of resource management, which is left to the programmer. In particular, there is only a single level of parallelism between programs and distributed data structures must be managed explicitly by allocating fragments to each of the program instances.

The *Message Passing Interface* (MPI) [WD96] defines a standard message-passing library interface with specific implementations for different machines, to provide a degree of portability for programs that use it. It is based on synchronised (or *two-sided*) message exchanges but also defines high-level collective operations such as broadcasts, reductions and all-to-all transfers of data, and features for the construction of modular components such as libraries with named groups of processes and scoping of operations within them [SDB93]. Despite its low-level SPMD approach, the resulting performance predictability and portability, and support for general parallelism from message passing that it provides, have resulted in MPI becoming the standard for programming in HPC.

MPI 2 [Mes09] introduced dynamic process creation but it does not address how resources are managed, leaving this to an implementation. In particular, a server construct to “address the need to support groups of reactive processes that accept connections from other groups” [Mes09, §10.4], or in other words, a mechanism for sharing. The problems with this are that the location of servers is not known at compile time and because MPI is a library they cannot be named. To quote the specification directly: “*Almost all of the complexity in MPI client/server routines addresses the question “how does the client find out how to contact the server?”*” This is due to the fact that MPI makes no assumptions about the mapping or scheduling of processes onto processors. The solution is a centralised scheme where servers publish their address to a *nameserver* third party, which clients then query to establish a connection with a particular server, but this will not scale well with large numbers of clients and servers.

One of the main reasons dynamic process creation was added to the MPI specification was to match the capabilities of the contemporary *Parallel Virtual Machine* (PVM) [BDG+91] library and run-time environment [GL95], which employs a message-passing model of communication and permits the dynamic creation of processes. PVM is based on the concept of managing a collection of computing resources with a single 'virtual' machine, and was originally intended for programming networks of workstations.

Lastly, the *shared memory* (SHMEM) [BK94], *Global Address Space Networking* (GASNet) [Bon02] communication interfaces to support shared-memory programming with, what are referred to as, *one-sided communication* that implement global read and write operations. These operations were also introduced in the MPI 2 specification. In general, libraries supporting one-sided communication are used as compilation targets for the translation of high-level data-parallel languages for HPC systems (which are surveyed in the next section) and are compiled according to a SPMD model.

3.3.5. Data-parallel programming

Data-parallel programming languages are ones in which operations are applied globally to arrays distributed over a collection of processors. These might include standard arithmetic operations such as addition or multiplication, or more general functions such as maps, sums, reductions and prefixes. Typically, the decomposition and placement of data is specified with a high-level syntax to hide the details of how data and computations are partitioned across processors, and to hide the management of global and local memory accesses, in order to provide the programmer with simple sequential semantics.

Essentially, a data-parallel approach can be viewed as a high-level notation for a shared-memory programming model and programs expressed in this way are compiled to a form in which processes synchronise in groups and perform remote memory accesses.

Data-parallel Fortran variants

A family of data-parallel programming languages are based on versions of sequential Fortran, enhancing them with syntax for data decompositions and data-parallel operations. These include *Connection machine Fortran* [BHMS91], *Fortran D* [FHK+90], *Vienna Fortran* [CMZ92] and *High Performance Fortran* (HPF) [KLS93]. These differ primarily by the way in which data decompositions and distributions are specified, but are all based on compilation schemes, which translate the program into a SPMD form. This restricts the available parallelism to a single level as operations cannot be composed in parallel or nested.

Since the data structures in these languages are distributed at compile time over the parallel program instances, the access patterns can be determined at this point. However, even for simple programs, access patterns can become complex and the process of determining them involves non-trivial analysis of the distribution of data and its dependencies between loop iterations and processes. The generation of efficient SPMD code depends on a number of optimisations based on the analysis of the resulting communication patterns; for example, by aggregating messages, overlapping communication with computation, reordering sequential operations and introducing additional parallelism [HKwT92b, HKwT92a, HBB+95]. The result is a complex sequence of communications nested in conditional control flow, bearing little relation to the original program and creates a large potential for inefficiency; the execution time of a data-parallel Fortran program compiled with different compilers can vary significantly and hence programs become 'tuned' to particular compilers to obtain good performance.

Partitioned global address space languages

Another family of data-parallel programming approaches are referred to as *Partitioned Global Address Space* (PGAS). They aim to address the poor efficiency of high-level data-parallel languages by logically partitioning a shared address space into disjoint regions that each process can access locally, thereby providing the opportunity for the programmer to exploit locality. Since PGAS languages essentially permit the programmer some level of control over local and remote accesses, they also introduce the possibility of race conditions and deadlock. They therefore include mechanisms for synchronisation to maintain consistency.

The *Global Arrays* (GAs) library [NHL94] and languages *Coarray Fortran* (CAF) [NR98], *Unified Parallel C* (UPC) [CDC+99], *Titanium* [YSP+98] and *Z-level Programming Language* (ZPL) [CLC+98] were amongst a first generation of PGAS languages. In addition to the logical partitioning of arrays, these provided functionality associated with the specification and distribution of arrays and operations on them, similar to that of existing data-parallel languages. A serious drawback of these approaches however is that they do not permit the composition of data-parallel operations or expression of any other forms of parallelism. This restricts their ability both to support abstraction and a wide class of parallel programs. This problem has persisted from previous data-parallel languages and stems from their basis on a SPMD programming and compilation model.

Chapel [CCZ07] and *X10* [ESS04] build on the approach of the first generation of PGAS languages. As well as providing partitioned data distributions, they support more general forms of parallelism as well as their arbitrary nesting and parallel and sequential composition.[10] However, since their approach remains firmly within a shared-memory model, they do not directly support the expression of communication with message passing. Pairwise interactions are produced with synchronised access to variables, which was explained in §2.1.1 [p. 12].

The main feature of Chapel is the integration of a range of data types, distributions and operations into the language syntax; less emphasis is placed on these features in X10 although it provides similar abilities. The basic operations in Chapel are similar to those in HPF and ZPL, but they also include associative and unstructured arrays (a type of index set referred to as a *domain*). They are implemented using the task-parallel features of the language [CCDN11, CCD+11] to provide some level of performance transparency and the opportunity to optimise or modify them. However, the compiler remains responsible for generating the global reads and writes to satisfy any dependencies. Chapel programs are compiled by transforming them into a SPMD form and data-parallel operations are compiled to operate over fixed collections of processors. In this sense their resources are *statically* allocated, in contrast to the creation of parallel tasks, which is managed *dynamically* by the run-time system with queues of work per processor.

Fortress [ACH+05] provides similar features to Chapel and X10 with shared-memory data and task parallelism, but with a high-level mathematical syntax geared towards scientific programs.

3.4. Summary

This chapter has discussed parallel programming. At its heart are two key principles, the decomposition of a problem into a collection of parallel tasks and the notion of speedup and parallel efficiency to quantify performance. A small set of parallel paradigms characterise simple concepts that underpin the structure of many parallel algorithms. Their simplicity is essential when dealing with large amounts of parallelism and they can be viewed as building blocks that can be combined to implement parallel algorithms and programs.

To define what constitutes a good parallel programming language and to provide a basis for the survey of existing languages, a set of criteria were proposed:

[10] These languages are included in this section as 'data-parallel' because although in principle they can be used to express task-parallel programs, in a recursive way for example, this is not their intended usage and will likely not deliver scalable performance.

1. support for programming abstraction to allow a program to be constructed as a composition of parts;
2. support for data abstraction to separate representations of data from the computational parts that operate on them;
3. support for general parallelism to make the expression and composition of the paradigms convenient;
4. high execution efficiency.

A balance must be struck between the expressive power of a language in fulfilling the first three criteria with the last, the execution efficiency.

The survey highlighted three main programming paradigms: message passing, shared memory and functional. In general, approaches based on shared memory correspond closely to data parallelism. These can summarised as follows:

paradigm	**communication**	**parallelism**	**forms of parallelism**
message passing	explicit	explicit	general
shared memory	implicit	explicit	limited
▷ data parallel	implicit	implicit	single
functional	implicit	implicit	limited

The following summarise the main observations from the survey.

- *Message passing.* There are relatively few message-passing approaches and, despite a large number of data-parallel languages, MPI has been an enduring standard. Occam and its close descendants are the only languages addressing the need for distributed execution in embedded and general-purpose systems.

  The reason for the success of message-passing approaches is that they can be compiled simply giving them a transparent execution model. Simple compilation enables efficient and predictable programs to be expressed, as well as providing portability between different compilers and architectures. It is for these same reasons that the use of C is so widespread.

  The main criticisms of message passing is that the programmer is left to deal explicitly with communication and parallelism at the expense of the clarity of their program. This makes it difficult to build abstractions such as subroutines, modules and libraries, and, in particular, representations of data are fragmented between processes so it is difficult to express shared data structures.

- *Data-parallel.* Data-parallel languages are based on a shared-memory abstraction of communication in which data-parallel operations are performed on large (and potentially distributed) data structures. Thus, they are amenable only to data-parallel problems.

  Early data-parallel languages were closely tied to the SPMD compilation model. This restricted programs to expressing a single level of parallelism with no composition or nesting of operations. The allocation of data between parallel instances of the program allows access patterns to be determined at compile time, but they tend to be complex, even for simple operations. This led to complex and specialised compilers with the performance of programs being dependent on a particular compiler, rather than the program itself.

  PGAS languages attempt to place some control of locality with the programmer by associating portions of shared data structures with the local memory of processes. The Chapel language, an exemplar PGAS language, tries to address the problems with compilation by defining its high-level approach in terms of primitive parallel constructs. However, the resolution of global accesses and generation of distributed reads and writes is a complex problem and leads to inefficient execution.

  Almost all of the data-parallel/shared-memory/PGAS approaches deal with resource allocation at compile time because performance depends so much on this.

- *Functional.* Functional languages appear to be ideal for dealing with parallelism because it is implicit in the evaluation of non-dependent functions, due to the absence of side-effects. Communication is also implicit in the transfer of arguments and results between parent and child functions. This allows the programmer to focus on the high-level behaviour and composition of a program.

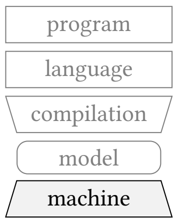
  The problem with a functional approach is that the correspondence to a machine architecture is weak and therefore the scope for an implementation is broad, requiring a great deal of run-time resource management. Although limited forms of functional parallelism can deliver good speedups, obtaining good speedups for general classes of functional programs is difficult, and especially so with distributed implementations. Functional languages also have some inherent limitations in dealing with external interaction and responding to unpredictable events.

In conclusion, existing message-passing approaches are best placed to express general forms of parallelism whilst maintaining efficient execution, but they offer little to the programmer for building abstractions. In contrast, data-parallel languages are designed to deal with data abstractions, but because this is based on hiding communication, communication patterns must be inferred (with difficulty) during compilation and they therefore do not provide a good machine abstraction.

It seems clear that these capabilities should be *combined*—shared memory *with* message passing—to produce a programming language that would be both expressive and efficient, fulfilling the proposed language criteria. This is the approach of the sire programming language described in Chapter 6.

CHAPTER 4.

GENERAL-PURPOSE PARALLEL ARCHITECTURE



This chapter describes how a parallel machine architecture can support a communicating-process model of computation in a way that provides an abstraction from specific implementations and for programs to be expressed in an architecture-independent way.

The chapter is divided into three main parts that correspond to different aspects of the architecture. First, the theoretical basis for a universal communication network with the choice topology and routing strategy; second, the practical details of an implementation of such a network with the mechanics of transmitting data around the network; and third, a discussion of the characteristics of the processors that the network connects and their affect on the nature of the machine. To contrast against this approach, a short survey of real machines is included at the end.

4.1. Universal communication networks

An *interconnection network* for a parallel computer is the hardware by which a collection of processors are able to communicate with each other. This might be to send messages in a distributed-memory system or to support memory accesses in a shared-memory system.

Ideally, an interconnection network connects every processor directly to every other processor to support efficient communication, as is illustrated in Figure 4.1. In practice however, this is not practical since the number of connections grows with the square of the size of the network and the number of connections to each node grows linearly. Even for small networks, the complexity involved in organising and connecting links in an implementation prohibits this approach and as such, it is inherently *non-scalable.*

A more practical approach is to use *sparsely-connected* networks where each network node is connected only to a small number of other nodes. When the number of connections per node is either fixed or grows slowly with the network size, this in principle allows it to connect an arbitrary number of nodes, limited only by the physical constraints of an implementation. Messages are sent between nodes in a sparse network by traversing a path from the source to the destination, potentially via intermediate nodes.

This section explains the surprising result due to Valiant that there exists a class of sparse *universal networks* that, in combination with a distributed routing strategy, can simulate any other communication network efficiently [VB81, Val90b]. This makes them suitable to support arbitrary communication patterns between a collection of processors. The remainder of the section first introduces some key concepts, before discussing the family of network topologies and the routing scheme.

4.1.1. Network model

A *network* is a collection of *nodes* that are connected by *links.* The purpose of a network in the context of this thesis is to perform a *computation.* Nodes can create and consume messages as well as forwarding them on to other nodes. Links are point-to-point connections between nodes that transport *messages* in one direction and a buffer is associated with each link to queue outstanding messages to be transmitted.

The structure of an interconnection network can be represented by a *graph* $G = (V, E)$ whose vertices V represent nodes and edges E that represent communication links. Each (undirected) edge

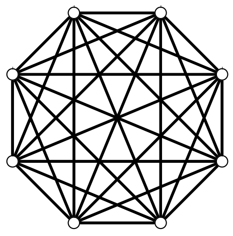


Figure 4.1.: A complete network. This is the ideal topology for an interconnect since every node can communicate directly with every other node, without interference from any other communications. However, an N-node complete network requires $N-1$ connections per node making it difficult to implement for all but the smallest networks.

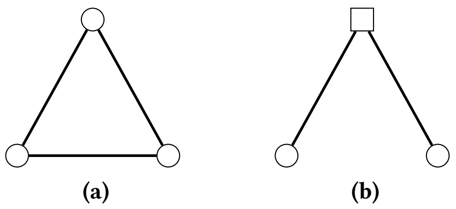


Figure 4.2.: Illustrations of the network model. (a) shows a network consisting of three terminal nodes connected with (bidirectional) links; and (b) shows a network with two terminal nodes connected by a non-terminal node, so that messages between the two terminals always traverse the intermediate non-terminal node.

represents two links, one in each direction. Each *terminal* node contains a *switch* that connects the incident links and a processor that can source and sink messages to and from the switch. A *non-terminal* node only contains a switch and a network that contains non-terminal nodes are called *indirect*. In a *direct* network, all nodes are terminal. The switch can establish connections between all of its inputs and outputs, and it supports simultaneous transmissions, but links can only transmit one message at a time. The transmission of a message along a link is taken to be one time unit. Figure 4.2 illustrates the main components of the network model.

Basic features

The following are definitions of some basic features of a network G:

- the *size* of G is given by the number of vertices, which is denoted by N and is generally an integral power of two, $N = 2^n$;
- a *path* $p(u, v)$ between two nodes u and v in G is an ordered sequence of adjacent edges that join them;
- the *distance* $d(u, v)$ is the length of the path between two nodes u and v in G, and a path is *minimal* if the distance is a minimum of all possible paths;
- the *degree* of G is the maximum number of edges incident upon a node;
- the *capacity* of G is the total number of edges.

4.1.2. Properties

Based on the above network model, the following two properties provide a strong characterisation of communication performance.

Definition 4.1 (Diameter)**.** The diameter D of G is the longest shortest path between any two nodes:

$$D(G) = \max_{u,v \in V} d(u, v).$$

The diameter gives a lower bound on the worst-case time required to deliver a message and hence should be small. The following theorem provides a lower bound

Theorem 1. *For any fixed d, the diameter of any N-node network of degree-d is at least $\Omega(\log N)$.*

Proof. Any node can in k steps reach at most $(d-1)^k$ other nodes. To be able to reach every other node it must travel at least the diameter of the network, so when $(d-1)^k \geq N$, the diameter is at least $\log_{d-1} N$. □

Definition 4.2 (Bisection width)**.** The bisection width B of G is the minimum number of vertices that connect any two equally-sized partitions (± 1 for odd sizes) of V. For every $U \in V$, let $\bar{U} = U \backslash V$ and $C(U, \bar{U})$ be the number of edges that have one end point in U and one end point in $\bar{U}$, then

$$B(G) = \min_{U \subset V, |U| = \lfloor |V|/2 \rfloor} C(U, \bar{U}).$$

The bisection width is a measure of how connected a network is. In a network with a large bisection width, communication can be performed efficiently across any partition of the network into two equally sized components. A network with a small bisection width will have a 'bottleneck' where many paths of communication share only a small set of edges.[1] Both of these metrics also provide a measure of the implementation complexity in terms of routing links between components.

The additional properties of *symmetry* and *hierarchical recursivity* relate to the structure of the network and affect both the implementation and capability of network. The symmetry of a graph can apply to both vertices and edges. A *vertex-symmetric* graph looks the same from any vertex, i.e. no vertex can be distinguished for any other based on the surrounding vertices. A similar definition is given for *edge-symmetric graphs.* Symmetry is important for parallel programming as it implies uniformity. The execution and evolution of a program is therefore independent of its location. If a graph is *hierarchically recursive* then it contains smaller sub-graphs of the same structure. This can be beneficial for an implementation based on a hierarchical integration using different technologies and for executing parallel programs with a recursive structure.

4.1.3. Simulations

An important ability of a network in the context of parallel communication is to be able to *simulate* other networks. Let H be the *host* network with p nodes and G be the *guest* network with v nodes. A simulation is then an *embedding* of G into H, which consists of a pair of mappings: one from each node of G to a node in H and the other from each *edge* in G to a *path* in H. The simulation of a communication pattern takes a number of time steps which is dependent on the distances messages have to travel and contention for edges due to intersecting paths in the embedding. If G is known in advance then a *static embedding* can be devised to obtain the best-possible mappings. A simulation performed using a static embedding is said to be *off-line.* If G is not known in advance, since the G may be generated or change dynamically, then a *dynamic embedding* is required where a mapping of the edges is performed *on-line* during the simulation.

Definition 4.3 (Simulation efficiency)**.** The *efficiency* by which H can simulate G is judged by the *slowdown* in the simulation compared to an direct implementation of G. If H can simulate t_G steps

[1]A more general form of bisection width is *expansion* which measures how well-connected subcomponents of a graph are to the rest of the graph in which they are contained. Graphs with high expansion are known as *expanders* [HLW06].

of G in time t_H, then the efficiency E is defined as

$$E = \frac{t_H N_H}{t_G N_G}. \tag{4.1.1}$$

This is based on the node-time product of the network to normalise the slowdown when $N_H \neq N_G$.

A simulation is said to be *optimal* when E is a constant that does not depend on p_G or p_H.[2]

There are four measures that bound the efficiency with which a host can simulate some guest [Lei92]:

- the *dilation* is the maximum path length in H of any edge in G;
- the *congestion* is the maximum number of edges in G that are mapped to any edge in H;
- the *load* is the maximum number of nodes in G mapped to any node in H;
- the *expansion* is the ratio between the number of vertices in H to the number of edges in G.

When all of these measures are small, it corresponds to an efficient simulation, and if all of them are constant, then the host can simulate the guest *optimally*.

The notion of a simulation gives rise to a class of *universal networks* that can simulate any other network. This can be defined in the following way.

Definition 4.4 (Universal network)**.** A *universal network* is a bounded-degree network that can simulate any other bounded-degree network of the same size with efficiency $O(\log N)$.

Although this efficiency is not optimal, it is the least possible since the diameter of any practical bounded-degree network has a diameter of at least $\Omega(\log N)$, due to Theorem 1. Furthermore, Leiserson has observed that non-universal networks exhibit a polynomial slowdown when simulating other networks and therefore have no theoretical advantage over a sequential machine, which can simulate any parallel machine also in polynomial time, since t steps of an N-node network can be simulated in tN steps sequentially [Lei85].

In the next section, a family of *hypercubic* networks are introduced that are universal for off-line simulations. §4.1.5 [p. 49] after that explains how, with the addition of a dynamic routing scheme, hypercubic networks can be universal for on-line simulations as well.

4.1.4. Example networks

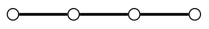
Before introducing the hypercube family of universal networks, the next section introduces some other simple sparse networks and discusses their properties. These simple networks relate to structure of many parallel algorithms; some have already appeared in the description of process and data-flow structure paradigms in §3.1.3 [p. 21].

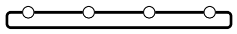
Common bounded-degree networks

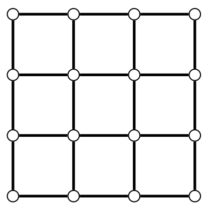
Arrays generally refer to *mesh* and *toroidal* networks. In a d-dimensional torus, if each node is labelled with a d-component coordinate, it is connected to $2d$ other nodes where each adjacent node differs in one coordinate component by one unit; Figure 4.3d shows two examples. In a d-dimensional mesh, nodes located at the 'edge' of a dimension with a coordinate component that is 0 or $\sqrt[d]{N} - 1$ do not connect to the adjacent node at the other end of the dimension (these are referred to as 'wrap around' links); Figure 4.3c shows two examples. Two special instances of arrays are 1-dimensional mesh, called a *line* (Figure 4.3a) and a 1-dimensional torus, called a *cycle* (Figure 4.3b).[3] In general, the

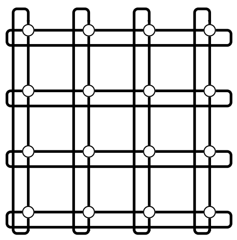
[2] It is interesting to note that this is similar to the definition of parallel efficiency given in Definition 3.2, with $S(n)$, specialised single processor machine that directly implements the algorithm, relating to a direct implementation of G.

[3] A generalisation in the definition for toroidal and mesh networks specifies the size of each dimension as k. They are thus referred to as k-ary d-cubes and k-ary d-meshes respectively [DT03].


(a) 1D mesh (line)


(b) 1D torus (cycle)


(c) 2D mesh


(d) 2D torus

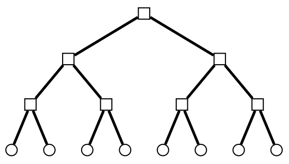

(e) Binary tree

Figure 4.3.: Examples of common bounded-degree networks.

diameter of any d-dimensional array is $\Theta(\sqrt[d]{N})$, the bisection size is $\Theta(N/\sqrt[d]{N})$ and the capacity is $\Theta(N)$.

Trees are structures where each node connects to a single parent node and one or more *child* nodes, except at the *root* which has no parent. In a k-ary tree, where each node has k children, with d levels contains $k^d - 1$ nodes. The number of nodes in a tree grows exponentially with respect to the depth and the diameter is $\log_k N$, but the root creates a 'bottleneck'. For a binary tree when $k = 2$ (see Figure 4.3e), one half of the nodes can only communicate with the other half through this point. A binary tree can therefore be bisected by removing just a single link from the root node. This makes it particularly unsuitable as a communications network.

Hypercubic networks

Hypercubes have been popular in parallel computing due to their rich communication properties and simple homogeneous structure that is both symmetric and recursive. A d-dimensional hypercube consists of 2^d nodes and, if each node is labelled with a d-bit identifier, then an edge exists between any two nodes u and v if and only if their identifiers differ by exactly one bit; hence each node is connected to d other nodes.[4] Conceptually, a 4-dimensional hypercube can be constructed by joining two cubes (3-dimensional hypercubes), each with 8 nodes, by adding 8 edges between corresponding vertices of each cube. This is construction is illustrated in Figure 4.4.

The drawback of hypercube networks is that their degree is logarithmically related to the network size. To scale the number of nodes, the node degree increases and consequently the complexity of the switch in an implementation. Furthermore, switches designed for an N-node hypercube cannot be used in a $2N$-node hypercube. Additionally, the wiring density increases rapidly with network size; for any bisection of the network, $N/2$ links between every pair of nodes pass through it, resulting in a complex pattern of connections.

There is a family of networks that are related to the hypercube that have constant degree nodes. The cost of this is a logarithmic factor increase in the number of nodes and diameter. Below, the common hypercube variants are surveyed because they are closely related to one another and they

[4]They are also known as *binary N-cubes*, an instance of a torus network in which the size of each dimension is two.

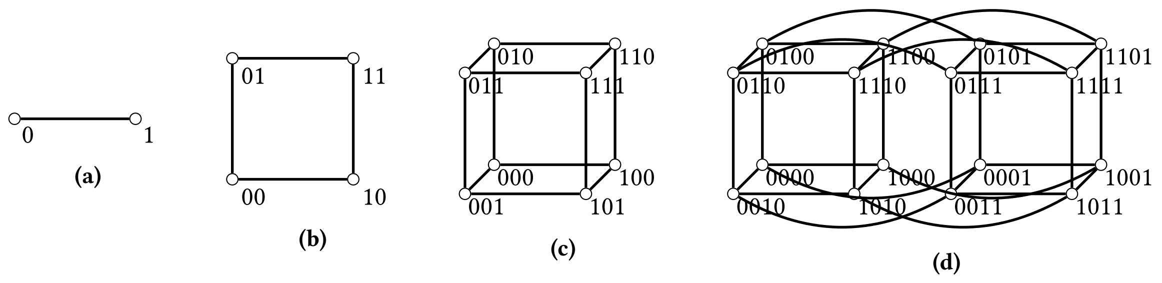


Figure 4.4.: Hypercubes networks in 1, 2, 3 and 4 dimensions. A $d + 1$-dimensional hypercube is constructed by creating two dimension d hypercubes and connecting corresponding nodes between them.

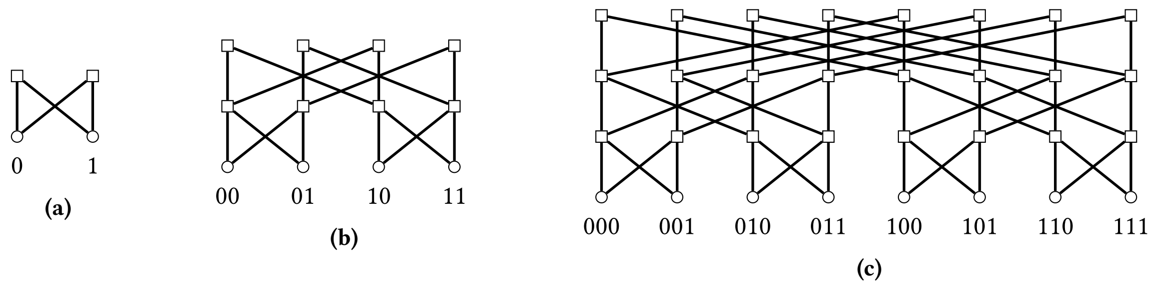


Figure 4.5.: Butterfly networks in 1, 2 and 3 dimensions. These correspond to the same dimension hypercubes in Figure 4.4, with the columns to each node of the hypercube and the crossing links between columns to the edges of the hypercube.

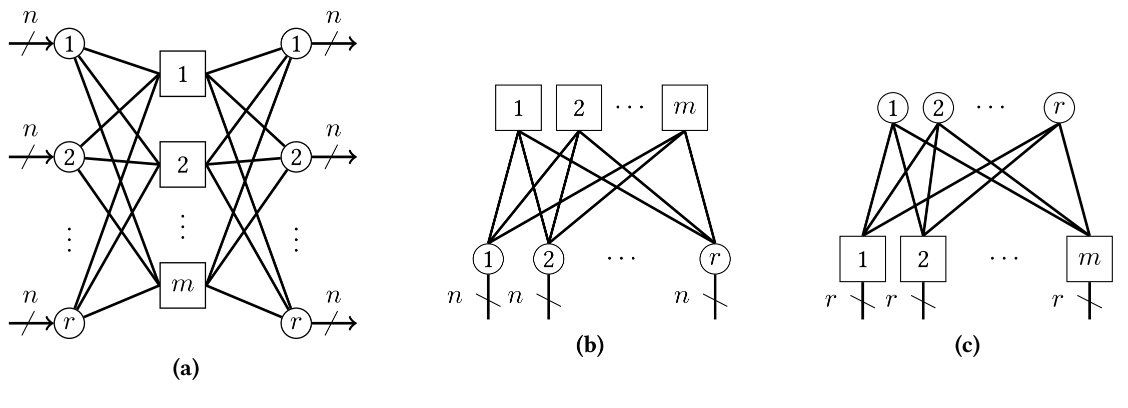


Figure 4.6.: Clos networks. (a) is a general three-stage (n, m, r)-Clos network, with r input and output nodes each with n inputs and outputs respectively, and m middle-stage nodes. Each input and output is connected by one link to every middle-stage node. Middle stage nodes can be replaced with a three-stage Clos, expanding the number of stages by two, to reduce the degree of nodes, and middle-stages can be removed to adjust the bisection bandwidth. (b) shows a folded Clos (along the middle-stage nodes), merging the input and output nodes and (c) shows it folded in the same way but with terminals connected to the middle-stage nodes, which can be more convenient for an implementation.

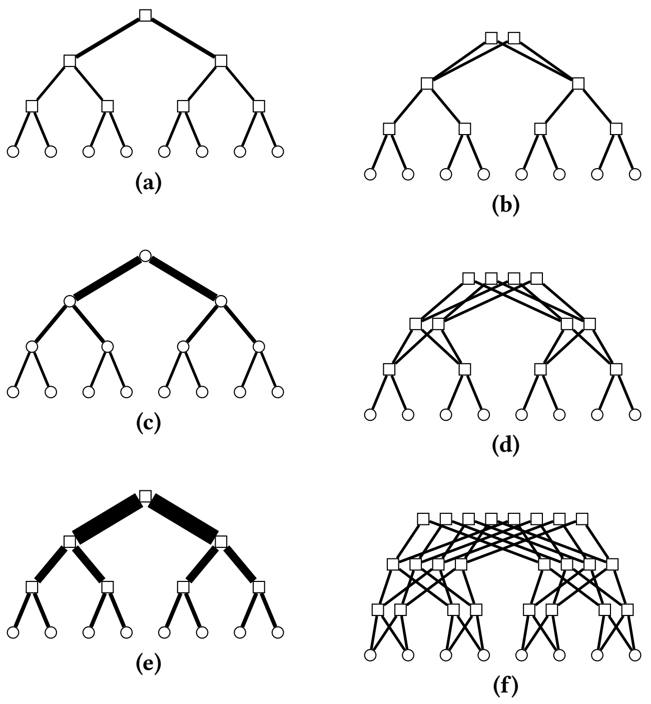


Figure 4.7.: Fat tree networks. (a), (c) and (e) are examples of binary fat tree networks. 8 terminal nodes (leaves) are connected by 3 stages of intermediate nodes. Fat tree (a) adds 2 additional links at the root of the tree to increase the bisection bandwidth to 2; (c) adds 4 links at the root and 4 at the second stage to increase the bisection bandwidth to 4; (e) adds 8 links to each stage, providing full bisection bandwidth. These can be built from fixed-degree nodes by expanding the nodes as illustrated in (b), (d) and (f). The stages containing split fat-tree nodes in (b) and (d), and the complete network in (f) correspond to butterfly networks of degree 1, 2 and 3, or folded-Clos constructions where $m = n = 2$.

appear widely in the literature, but their relations are often not made apparent. The survey highlights the individual properties of each variant, to explain how they are related and demonstrate that they are all (computationally) equivalent, up to constant factors. Establishing this equivalence is important since it means that theoretical results based on one variant will also apply to the others.

First, *butterflies* and *cube-connected cycles* (CCCs) [PV81] extrapolate in a simple way from the hypercube.

A butterfly network is a *multistage* network that is constructed by expanding each node of an n-dimensional hypercube into $n + 1$ degree-4 nodes (called a *column*). A stage consists of adjacent nodes in a *row*. There are two kinds of links: *straight links* that join nodes in a column and *crossing links* between columns, which are derived from the hypercube. Links between stage i and $i + 1$ are of stage i. Figure 4.5 shows 1, 2 and 3 dimensional butterflies.[5]

A butterfly is *indirect* in the sense that N inputs and N outputs are connected by $N \log N$ nodes and messages will travel further than in the hypercube. It is also has no *path diversity* in that a only a unique shortest path exists between any pair of terminal nodes (stage-0 nodes).

CCCs replace each node of an n-dimensional hypercube with a cycle of n nodes. Each node of the cycle is connected to a corresponding node in a different dimension. A CCC is closely related to the butterfly and is contained as a sub-graph [FU92]; a single cycle (node) of the cube corresponds to an entire column of the butterfly.

[5] A common variant of the butterfly is the *wrapped butterfly*, which is obtained by merging nodes in the first and last stages. The relationship between the butterfly and wrapped butterfly is similar to the line and ring networks; each can simulate the other with at most a factor of 2 slowdown. Another variant is the *multibutterfly* which increases the degree of nodes by a factor of 2 [LM89, Upf92].

Clos [Clo53] and *Beneš* [Ben65] networks are indirect, multistage networks that were originally conceived for use in telecommunications. This was due to their *non-blocking* properties that enables them to support circuit switching, where multiple simultaneous connections between the inputs and outputs are used for continuous transfers of data. These networks have separate input and output nodes and links are unidirectional.

A Clos network has three stages: input, middle and output, and is parameterised by the triple (m, n, r). The input stage consists of r lots of $n \times m$ nodes, the middle stage of m lots of $r \times r$ nodes and the output stage of r lots of $m \times n$ nodes. One link connects each of the middle-stage nodes to each of the input and output nodes. This therefore allows any input to be routed to any output via any middle-stage node. The general structure of an (m, n, r)-Clos network is illustrated in Figure 4.6a.

The number of switches in a particular stage can also be adjusted to increase or decrease the number of available paths between each input and output. With just one middle-stage switch, the network would be fully connected but only one path would exist between any pair of inputs and outputs. With m middle-stage switches, m paths are available. When $m \geq 2n - 1$ the network is *strictly non-blocking*. This means any permutation mapping the inputs to the outputs can be realised by link-disjoint paths; i.e. messages sent from each input to a unique output can be routed along a set of paths in which no two paths share the same link. When $m \geq n$, the network is *rearrangeably non-blocking*, meaning that existing paths can always be rearranged to provide a link-disjoint path.[6]

A Clos network can be generalised to any odd number of stages by replacing each middle-stage node with a 3-stage Clos network with the same number of inputs and outputs. Another way to see this is that a Clos network can be used to implement any middle-stage node. This can be done to reduce node degree or to increase the number of input and output connections.

Beneš networks are a special case of a Clos network where $m = n = 2$ and with additional stages added to reduce all nodes to degree-4. This structure is rearrangeably non-blocking. An N-input Beneš network can be constructed by placing two N-terminal butterflies back-to-back, merging the terminal nodes.

The symmetry of the Clos and Beneš networks allows them to be *folded* around the middle-stage nodes, merging the input and output networks and using bidirectional links [JDFJ97, Ch. 3]. This approach has two advantages: it makes the network more convenient to package when connecting terminals that produce both input and output, and messages can traverse shorter paths between terminals. Figure 4.6b shows the network in Figure 4.6a folded along the middle-stage switches. Figure 4.6c shows the same folded structure but with the terminal connections made to what were the middle-stage switches. This is an equivalent construction that can be more convenient to build since it follows a tree structure with terminal connections at the leaves and is related to the *fat tree*.

A *fat-tree* network is based on a binary tree structure but to remedy the 'bottleneck' at the root, links closer to the root become progressively 'fatter' by bundling collections of links to preserve the network bisection size, in the same way the branches of a tree are get progressively thinner towards the leaves. The structure of a fat tree is illustrated in Figure 4.7, where the network bisection bandwidth of the network increases between Figures 4.7a, 4.7c and 4.7e, which is indicated by thicker links.

The original proposal for fat trees was based on nodes of increasing degree that connect links of increasing capacity [Lei85]. This however, requires nodes of an exponentially growing degree and therefore poses problems for an implementation of these switching elements in terms of area and latency. A practical way to implement fat trees is by splitting each node into a number of independent fixed-degree nodes; for each fat tree in Figure 4.7 an equivalent construction with fixed-degree nodes is given (Figures 4.7b, 4.7d and 4.7f). In general, this is equivalent to a folded Clos, but other

[6]Although the non-blocking properties of Clos networks are well understood in telecommunications, they are not exploited in the same way in the context of computer communication networks [Yua11], instead being used for packet switching (see §4.2 [p. 54]).

generalisations of fat trees have also been proposed [OIDK95, PV97].[7]

At this point it is useful to summarise the relationships between the networks described thus far.

- (*hypercube* $\rightarrow$ *butterfly, CCC*) An n-dimensional butterfly is obtained by expanding each node of an n-dimensional hypercube, and this contains n-dimensional CCCs as a subgraph (shown in Figure 4.4 and Figure 4.5).
- (*fat tree* $\rightarrow$ *butterfly*) Expanding each node of a depth-n fat tree into a number of nodes, each with two links to the previous level and two links to the next level (unless they are a root or leaf node), yields an n-dimensional butterfly graph (shown in Figure 4.7).
- (*folded Clos* $\rightarrow$ *fat tree*) An n-stage folded-Clos network with terminal connections made with the middle-stage switches is equivalent to a fat tree where each node is expanded with the same degree nodes in each level (shown in Figure 4.6c).
- In both the folded-Clos and fixed-degree fat-tree networks, nodes of degree greater than 4 (that of the butterfly) can be used to reduce the number of levels or increase the bisection bandwidth, or a combination of both.

Lastly, the *shuffle-exchange* and *de Brujin* [dBE46] networks, although still related to the hypercube, are not derived as directly.

Shuffle-exchange networks have 2^n nodes with two kinds of links: *exchange links* that connect nodes differing in the least significant bit of the binary representation and *shuffle links* that connect nodes identified by a one-bit circular shift of the binary representation. They have been suggested as appropriate for use in parallel interconnection networks in several influential papers [Sto71, Sch80] and although they have constant-degree nodes, suffer from the similar wiring problems as hypercubes.

De Brujin networks closely relate to shuffle-exchange networks. An N-node de Brujin network can be obtained by merging all of the nodes connected by exchange edges in an $(N+1)$-node shuffle-exchange network.

The following theorem establishes a universality result for hypercubic networks.

Theorem 2 (Leighton [Lei92, Ch. 3])**.** *Any N-node bounded-degree network can be simulated off-line in an N-node butterfly network with a $O(\log N)$ slowdown.*

The butterfly is therefore a *universal* network since it can simulate any other network with a comparable amount of hardware (i.e. processors, switches and wires). The $O(\log N)$ overhead is the least possible overhead that can be achieved with practical bounded-degree networks. This result also applies to CCC, Beneš, Clos, shuffle-exchange and de Brujin networks since they are all computationally equivalent up to constant factors.[8]

4.1.5. Routing

So far only off-line simulations have been considered. These use static embeddings of one network into another, where full knowledge of the network structure can be used to choose paths to minimise dilation and congestion. This is the basis for the universality result of Theorem 2. A more difficult problem, and the one of a general parallel communication scheme, is efficient on-line simulations of arbitrary and potentially dynamic communication patterns. This requires a *distributed routing scheme* that makes decisions based on local information, because in a centralised scheme, it would be inefficient for each node to gather information about the whole network and would require an

[7] The terms folded Clos and fat tree are often used interchangeably in the literature, where fat tree typically refers to the construction with fixed-degree nodes rather than the original proposal [Lei85].

[8] For a class of *normal* hypercube algorithms where only one dimension of the hypercube is used for communication at a time, these networks can simulate the algorithm directly with no loss of efficiency [Ull84, Ch. 6]. The intuition for this is that each node of a hypercube is expanded into $\Theta(\log N)$ nodes, with each node being connected in one of the original dimensions.

Name	Degree	Diameter	Bisection size	Capacity per terminal
Ideal				
Complete	N	1	$(N/2)^2$	$N-1$
Simple				
1D array	2	$N-1$	1	$\Theta(1)$
2D array	4	$\Theta(\sqrt{N})$	$\Theta(\sqrt{N})$	$\Theta(1)$
3D array	6	$\Theta(\sqrt[3]{N})$	$\Theta(N/\sqrt[3]{N})$	$\Theta(1)$
Binary tree	3	$\log N$	1	$\Theta(1)$
Hypercubic				
Hypercube	$\log N$	$\log N$	$N/2$	$\Theta(\log N)$
Cube-connected cycles*	3	$\Theta(\log N)$	$\Theta(N/\log N)$	$\Theta(\log N)$
Butterfly*	4	$\Theta(\log N)$	$N/2$	$\Theta(\log N)$
Beneš*	4	$\Theta(\log N)$	N	$\Theta(\log N)$
Clos*	$\Theta(1)$	$\Theta(\log N)$	$\Theta(N)$	$\Theta(\log N)$
Folded Clos*	$\Theta(1)$	$\Theta(\log N)$	$\Theta(N)$	$\Theta(\log N)$
Fat tree*	$\Theta(1)$	$\Theta(\log N)$	$\Theta(N)$	$\Theta(\log N)$
Shuffle-exchange*	3	$2\log N$	$\Theta(N/\log N)$	$\Theta(\log N)$
De Brujin*	4	$\log N$	$\Theta(N/\log N)$	$\Theta(\log N)$

Table 4.1.: Summary of different sparse networks and their properties. '*' denotes the properties of these networks are expressed relative to the number of terminals they connect, N, rather than the number of nodes, $\Theta(N \log N)$.

amount of state proportional to the size of the network. A distributed scheme therefore provides performance that scales well with network size.

A communication pattern can be characterised by a *h-relation* where each node has at most h messages to send to various other nodes in the network, and each node is also due to receive at most h messages from other nodes in the network [Val90a]. A special case of a 1-relation is a *permutation* or one-to-one mapping where each node sends to a unique destination. A *routing scheme* for a network G determines the paths on which messages are sent between source and destination nodes, providing a dynamic embedding of paths for a h-relation into G.

An essential property of any routing scheme is that it should not cause *deadlock*. Deadlock is a property of the combination of the network and routing scheme. It is caused by a cyclic dependency between two or more messages that each wait for the other to release limited network resources whilst holding some themselves. Typically, this limited resource is buffer space.

Oblivious routing

The most simple routing schemes are *oblivious* where messages are routed according only to the address of the destination and possibly the source and not to any state of the network. Oblivious schemes will typically be *greedy* and only route along minimal paths.

For any particular network, adversarial permutations can be designed to maximise the congestion on particular links and hence the time taken to realise the communication. The following theorem provides a general lower-bound for deterministic oblivious schemes, representing a worst-case.

Theorem 3 (Borodin & Hopcroft [BH85]). *For any N-node degree-d network, no deterministic oblivious routing scheme can perform a 1-relation in less than $\Omega(\sqrt{N}/d)$ steps.*

Although these schemes perform poorly in the worst case, they do in fact perform well on average since adversarial permutations rarely occur. Consider a random routing problem called a *random*

h-mapping where each node has at most h packets to send, each packet has a destination chosen uniformly at random and each node receives an expected h packets. Then, the following theorem established this routing problem can be simulated efficiently on a butterfly.

Theorem 4 (Leighton [Lei92, Ch. 3]). *With high probability, a random h-mapping between the $\log N$ input nodes of an N-node butterfly can be realised in $O(\log N) + O(h)$ steps with a deterministic oblivious routing scheme.*

Therefore, the average case behaviour of deterministic oblivious routing is close to the best case, even when the network is heavily loaded.

For all d-dimensional arrays (i.e. mesh and toroidal networks that were defined in §4.1.4 [p. 44]), *dimension-order* routing can be used. When each node is labelled with an d-component coordinate this works by moving a packet through the first dimension until the corresponding coordinate is the same. It then proceeds in the same way for the remaining dimensions until the coordinates match and the destination is reached. For multistage indirect networks, a simple deterministic routing scheme operates in two phases where packets are first routed 'up' towards the middle-stage switches and then back 'down' towards the terminals.

There are a wide variety of oblivious routing algorithms for different types of networks and communication patterns, see [Lei92, §1.7] or [DT03, Ch. 9] for examples.

Adaptive routing

In contrast to oblivious schemes, *adaptive* schemes choose the available paths between a source and destination based on some criteria. This serves to balance communications more evenly over the network to reduce congestion. Adaptive schemes could take into account the state of the network, such as the buffer occupancy in neighbouring nodes, and are consequently more complex than oblivious ones.

Furthermore, adaptive schemes can create additional communication due to the exchange of information with other nodes, they can potentially choose longer routes for messages and can cause *livelock*. This is when messages are able traverse non-minimal paths and make no progress towards the destination. Adaptive schemes are therefore beyond rigorous theoretical analysis [Val82] and cannot practically be used to deliver bounded latency.

For examples of adaptive routing schemes see [Sch98, Ch. 8] and for further details see [DT03, Ch. 10].

Two-phase randomised routing

The bounds in Theorems 3 and 4 show that although oblivious routing schemes perform poorly in the worst case, they do on average perform well. Based on this in the 1980s, Valiant observed that efficient distributed routing could be achieved by reducing any routing problem into a sequence of two random routing problems. This technique has come to be known as *two-phase randomised routing* and has been widely studied [VB81, Val82, Upf84, Val90b].[9]

Two-phase randomised routing works by routing all messages to a randomly chosen intermediate node, before routing them towards their destination. An oblivious routing scheme used in either phase. The effect is that each phase corresponds to random 1-mappings and all communication patterns are reduced to a collective worst case. On average, messages travel twice the average path length, but load is evenly distributed across the network, making delivery time uniformly low and therefore bounded. Since the scheme involves randomisation the bound is not guaranteed but it holds with overwhelming probability; very occasionally messages will be held up but overall performance will not be noticeably affected. It also generalises to arbitrary h-relations.

[9] Randomisation has been employed in a different way for routing on fat-tree networks by randomly resending groups of messages [GL96]. However, it lacks the simplicity of two-phase randomised routing.

The following theorem summarises the main theoretical result from the study of two-phase randomised routing, establishing that hypercubic networks can deliver on-line simulations of arbitrary bounded-degree networks.

Theorem 5 (Valiant [Val90b]). *With high probability, in $O(\log N)$ steps, every h-relation can be realised on an N-node butterfly and every $(h \log N)$-relation can be performed on a N-node hypercube.*

Empirical evaluation of two-phase randomised routing on hypercubes, folded-Clos and shuffle-exchange networks strongly supports this result [VB81], [MTW93, Ch. 7].

There are two important practical implications of two-phase randomised routing:

- packets travel twice the average path length and in order to maintain throughput per node in general networks, the capacity must be doubled by replicating all links;
- to avoid deadlock occurring between the two routing phases, the network must be divided into two components, where one component performs the first routing phase and the other performs the second [MTW93, Ch. 7].

A special case is with multistage indirect networks since messages can pass though a randomly chosen middle-stage node with exactly the same effect as routing via an intermediate node. By doing this, there is no increase in the average path length, additional links are not required to scale throughput and it is also not possible for deadlocks to occur between the routing phases, since such a route cannot contain any cycles.

4.1.6. Characterisation

For a universal network, the latency scales logarithmically with the number of nodes and the throughput per node scales by at most a logarithmic factor. The combination of a universal routing scheme ensures that these bounds are delivered for all communication patterns. The consequence of this is that the structure of the network is *independent* from anything that it simulates and can thus be characterised by just two parameters, the *grain* (g) and the *latency* (ℓ) [Val90a, May94].

Definition 4.5 (Grain). The *grain* of a network is the *communication bandwidth*, the ratio between the rates of computation and communication, and defined as

$$g = \frac{1}{\text{time steps per unit communication}}$$

assuming a single computational operation can be performed in one time step.

Ideally, the rates of computation and communication are matched equally with $g = 1$ so that an operand can be communicated every step and a high rate of processing can be sustained. Thus, in t steps, t operations can be performed and t/g messages can be exchanged. To note, this definition of granularity differs from the *program granularity* defined in §3.1.2 [p. 19], which was the ratio of sequential work to parallelism.

Definition 4.6 (Latency). The *latency* of a network, measured in units of time, bounds the time for a message to be delivered and contributes to the value of g. It is determined by three components and defined as

$$\ell = \text{communication startup latency} + \text{network latency} + \frac{\text{message size}}{\text{channel throughput}}$$

The *startup latency* is the time taken at the source and destination to setup a connection. It is determined by the processor and its interface to the network; in particular, the time required to invoke the operation from a process and to move data from the process into the network, or vice versa (the components of this latency and the processor and its interface to the network are discussed in §4.3.1 [p. 58]). The *network latency* is the elapsed time after the head of the message enters the network to it first arriving at the destination and it depends on the distance travelled and switching delay at each node. The third component of ℓ is the time taken for the contents of the message to be transmitted through the network. This is determined by the *channel throughput*, the rate at which messages can be delivered along the channel of communication. The throughput is the *effective* bandwidth in the presence of other communication traffic in the network.

To sustain computational work in machines where ℓ is large and g is low, large messages have to be sent. This restricts processing to problems that involve large amounts of data transfer. By doubling g for a particular machine (bringing it closer to 1), the potential amount of concurrency that can be employed can be doubled. This is because a sequence of t operations on a message can be split into two processes with a message sent from one to the other, so that they both execute $t/2$ operations. A general-purpose machine should therefore support the efficient execution of a wide class of programs, including those with small problem sizes, by allowing parallelism to be employed to scale performance. In particular, a machine must support efficient communication with a low ℓ for small problem sizes.

4.1.7. Cost

The universality results discussed depend on the hypercubic family of networks, which are substantially more expensive to implement, in terms of the number of switching elements and links, than other less well-connected networks such as arrays and trees (see Table 4.1). It is important to ask *can less-expensive networks be used to obtain similar universal behaviour?* It is however easy to argue that this is not possible; the following argument is based on one made by Valiant in [Val88].

Since a universal machine is structurally independent from any it simulates, there is no locality in the simulation and messages will on average travel the diameter of the network d (to within a constant factor). For a network with N terminals and a communication grain g, then the number of messages in flight from a single node will be d/g. If there is no communications bottleneck, i.e. the per node throughput scales less quickly than N, in the system overall:

$$s = \frac{\text{total capacity}}{\text{number of terminals}} \geq d/g$$

Since g is essentially fixed and d increases with N, the capacity of a network has to exceed the number of components it connects by a factor of at least d and as the diameter of practical networks is at least $\Omega(\log N)$, s therefore has to be at least $\Omega(\log N)$. In the hypercube, each node provides the logarithmic connectivity and in the related networks it is provided by a logarithmic number of nodes per terminal.

It has also been argued by Dally [Dal90] that low-dimensional arrays (in 2 and 3 dimensions) have lower latency and higher throughput than high-dimensional networks such as hypercubes when the number of links is increased to match the bisection bandwidth in the hypercube. This is due to a larger number of paths in the permutation using links in a particular dimension and since there is a higher per link bandwidth, it permits greater sharing of the available network capacity.

If one were not to consider the theoretical limitations of low-dimensional networks, Dally's argument for their use appears to be convincing, but it is based on the assumption that it is acceptable to permit adversarial communication permutations to occur. Using low-dimensional networks will cause message latencies to be highly variable and programs will require careful mapping to the network, during compilation and/or at run time, to avoid congestion hot-spots.

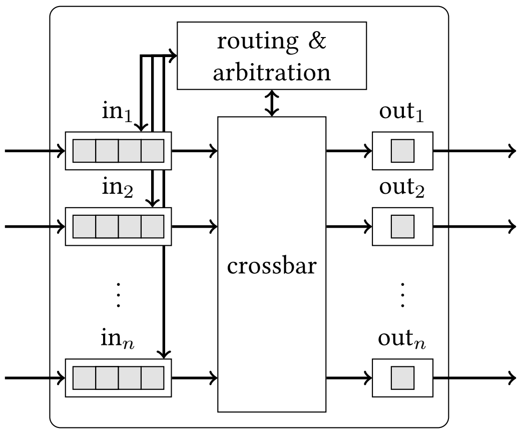


Figure 4.8.: Block architecture of an input-buffered $n \times n$ crossbar switch. With this, any input port can be connected to any output port according to the routing and arbitration control. Bidirectional communication links are connected to pairs of input and output ports.

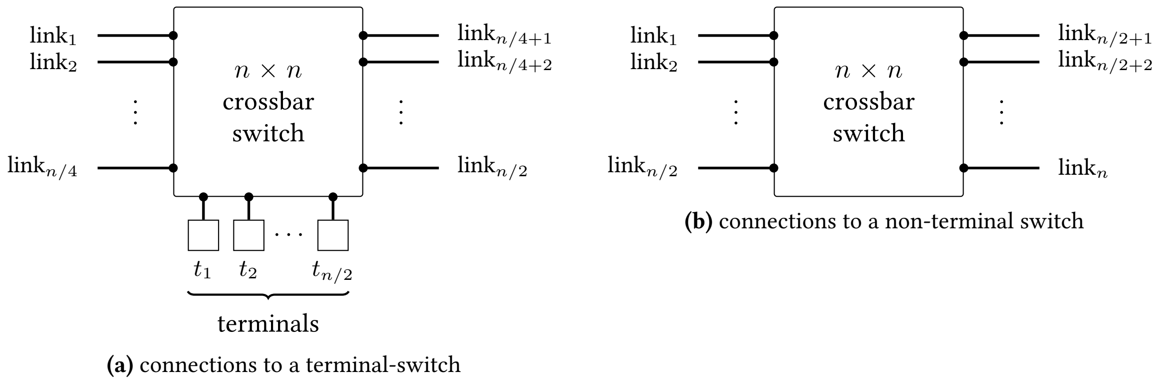


Figure 4.9.: Connectivity of a high-degree crossbar switch. A terminal switch (a) can use half of the available links of a switch to connect terminals and the other half to connect to the rest of the network. A non-terminal switch in a multistage network (b) uses all links to connect to the network.

4.2. Switching mechanics

Switching is concerned with the mechanics of transmitting messages between nodes in a network. This capability is implemented in a discrete *switch* component and one is included at each node. This section discusses the main issues relating to the operation of a switch with regard to building a universal communication network with two-phase randomised routing.

4.2.1. Crossbar switches

A *crossbar switch* allows all inputs to be simultaneously routed to their outputs. Switches are connected by links that connect an input port to the output port of another switch and vice versa. Figure 4.8 illustrates the basic architecture of a conventional input-buffered $n \times n$ crossbar switch.

It is beneficial to use high-degree switches to build networks because connecting multiple terminals to each one, the network diameter can be reduced. With a switch of degree-8 or greater, it is practical for half the links to connect terminals, leaving sufficient bandwidth and connectivity in the set of remaining links; this is illustrated in Figure 4.9.

The scaling of a switch is limited primarily by the delay and area requirements of the central crossbar component. A number of high-degree single-chip designs have been proposed and shown to be effective, such as the *INMOS C104 32×32 switch* [MTW93, Ch. 3], [JDFJ97, App. A], the 64×64

YARC switch [KDTG05, SAKD06], and the *Swizzle-Switch*, which scales up to 64×64 [SDTO+11, SDM+12a, SDM+12b].

Incidentally, with an increasing level of parallelism on-chip, various proposals for on-chip folded-Clos networks have been made with the stated aims of reducing latency, improving capacity and simplifying programming. There are proposals for electrical implementations, e.g. [BD06, LGM+09, KYAC11], and optical implementations, e.g. [JBK+09, ZGY10]. However, these only consider relatively small systems up to 64 processors and low-degree $4{\times}4$, $8{\times}8$, or $16{\times}16$ crossbar switches.

4.2.2. Switching schemes

A *switching scheme* deals with the allocation of resources for the movement of data around a network.

Store-and-forward switching

Store-and-forward or *packet-switching* splits a message into one or more fixed-length *packets*. Packets are individually routed to their destination and consist of data and routing information. Typically, the routing information is small and must have at least $\lceil \log N \rceil$ bits to specify the destination, it may also contain book-keeping and sequencing information. Every link is able to buffer at least one packet and a packet makes a *hop* from one node to another by moving the whole packet from the output buffer to the next input buffer. The transmission latency of store-and-forward switching is proportional to the product of the size of the packet and the distance.

Circuit switching

Circuit switching reserves a complete path between the source and destination before any data is sent. The circuit exists until it is closed either by the source or destination. This is advantageous for long messages as the cost of setting up the route is only paid once and, with an active circuit, data will experience no contention and can be streamed at a constant rate.

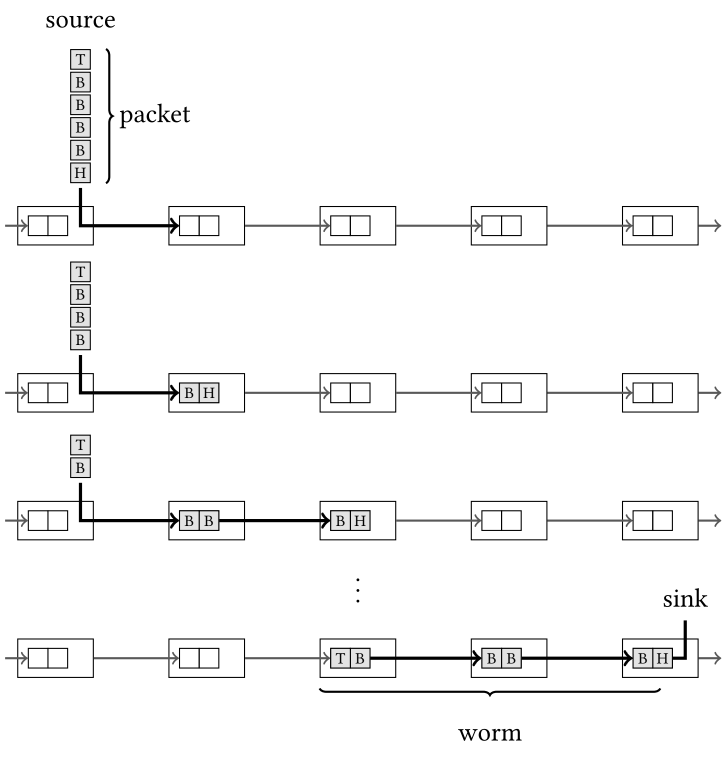
Wormhole switching

Wormhole switching is similar to store-and-forward switching but it reduces the latency and the per-link buffering requirement. Messages are again sent as a sequence of packets but each packet is transmitted as a sequence of *flits*, allowing the packet to be pipelined in the network, like a *worm* (typically, flits are a number of bytes, up to one word). A header flit opens a route through the network, subsequent flits follow it and experience no switching delay, then the final 'end of packet' tail flit closes the route. Data can be streamed in this way with an open route or packetised by closing the route after a payload has been sent. As such, wormhole switching can be seen as a form of dynamic *circuit-switching* where a complete route is established before messages are transmitted.

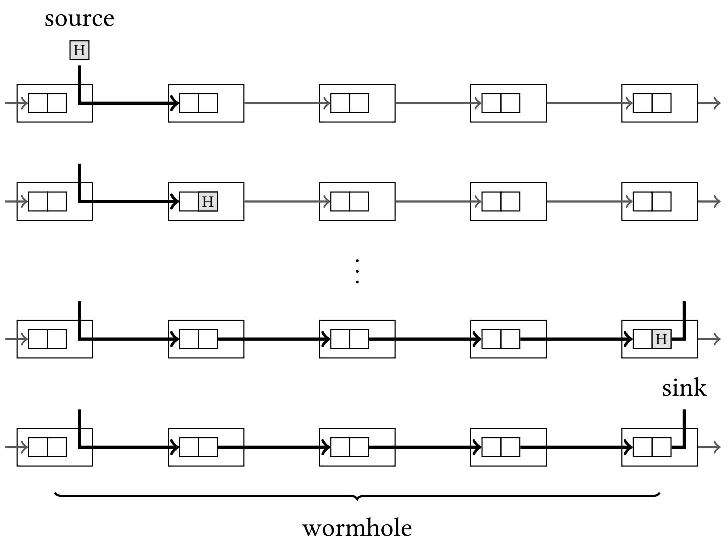
Packet latency is independent of the size of the payload since the header can progress immediately. This is an important ability in the context of a general-purpose system, since the size of link buffers can be chosen independently of packet sizes. The buffering requirement is also reduced as it is necessary to store only a small number of flits at each node.

Wormhole switching does however introduce the possibility of *deadlock* when a header becomes blocked by contention for a link and the rest of the packet also stalls. The remaining tail of the 'worm' will occupy buffers and potentially block other packets. To avoid this, it must be guaranteed that every packet must be always be able to make progress in the network. This can be done either with the use of *virtual channels* (explained below in §4.2.3 [p. 57]) or by employing a protocol that does not allow 'worms' to become blocked in the network. Such a protocol will ensure the receiver is ready before transmitting the payload.

Figure 4.10 illustrates the operation of wormhole switching and how it can be used to perform packet switching and circuit switching.



(a) wormhole packet switching



(b) wormhole circuit switching

Figure 4.10.: An illustration of the operation of wormhole switching. In (a), a packet consists of a sequences of flits: a header (H), followed by a number of body (B) flits, constituting the message data, and a tail flit (T). Each box represents a communication channel with input buffering for two flits and the horizontal sequences of boxes represent a path in the network. As the header flit traverses the network towards the destination and as it does this, establishes a route between input and output links in each switch. The proceeding flits then traverse this route, following the header. As the tail flit passes through, the route is closed. The sequence of flits contained in the channel buffers, spanning multiple switches is referred to as the *worm*. In (b) a single header flit is sent to establish a network route, along which data can be streamed.

4.2.3. Flow control

Flow control determines the allocation of network resources, such as buffer space and channel bandwidth [JDFJ97, §2.1.1], [DT03, Ch. 12]. This is necessary to make effective use of the available resources and to stop buffers from being over-run.

Credit-based flow control

Credit-based flow control is a simple way to manage the use of buffers to permit a node to output on a link only if there is sufficient buffer space on the corresponding input at the receiving node. This works by each output link maintaining a count of the available buffer space at the corresponding downstream node. When a packet (or flit) is sent the count is decremented; when the downstream node forwards the packet, it sends a credit message back upstream and the count is incremented.

In this presence of congestion, credit-based flow control provides a 'back pressure' from the busy link, initially causing packets that must also traverse this link to become blocked. As further packets become blocked, this eventually stops packets from being injected into the network and therefore allows the delivery of packets in transit.

Virtual channel flow control

Virtual channels are used to decouple resources associated with transmission on channels from the physical links [Dal92], [DT03, Ch. 16]. A single virtual channel contains a buffer and some state information, and a number of virtual channels are associated with each link. A particular virtual channel can be assigned to use a physical channel when the packet that has been allocated is able to progress, thereby allowing it to use a link when other packets using the same link are blocked. This provides benefits for communication latency and throughput.

Virtual channels therefore allow network resources to be better utilised when there is competition for their use and are most relevant to networks and routing strategies that may cause areas of high congestion. In particular, with low-dimensional array networks where the bisection size scales more slowly than the number of nodes. With high-capacity networks that provide constant throughput scaling per node, and when they operate in conjunction with a routing strategy that can evenly balance load, such as two-phase randomised routing, there will be relatively little link contention. In this situation, the utility of virtual channels is diminished (the reasons for this were explained in §4.1.7 [p. 53]).

The cost of virtual channels is the additional buffering required for each channel and the allocation and arbitration logic, which can all contribute significantly to the area of a switch. With high-dimensional networks, these costs are magnified. Additionally, the complexity introduced into the behaviour of a link can impact significantly on the overall predictability of the network. Another way to look at this argument is that a virtual channel switch invests more buffering in fewer links, whereas a high-degree switch invests in more links with less buffering, or a more highly connected networks invests more heavily in communication capacity.

Another use of virtual channels is to break deadlocks in routing schemes [DA93]. They can do this by separating the virtual channels of each link into a number of groups to obtain a partitioning of the network into independent sets of virtual links. Communication cycles are broken, by distributing them between the partitions to ensure deadlock cannot occur. The main disadvantage of using virtual channels in this way is that the number of virtual channels must be increased by a factor of the number of partitions required to guarantee deadlock freedom, with no performance benefit.

Buffering requirement

The theoretical analysis of two-phase randomised routing is based on store-and-forward packet routing and unbounded buffers, and the analysis of the run time implies that the size of a queue will grow to at most $O(\log N)$ with high probability since this is the time taken to deliver a random relation. Because of the bound holds with high probability, some links will experience high contention but with very low probability. In these cases there will be some additional communication delay.

Experience with using the C104 to build universal interconnects has demonstrated that little buffering is required with wormhole switching. For each inbound and outbound link around one packet (around 4 to 8 flits) is sufficient [JDFJ97, Appx. A].

4.2.4. Routing mechanics

Routing mechanics refers to the implementation of a routing scheme, i.e. the mechanism by which the output port of a switch is chosen to forward a packet if it has not reached its destination. In general, this is done either with a *routing table* at each node that records mappings of links to destinations, or by a direct calculation of the next destination according to a rule. Only the former is discussed here, see [DT03, Ch. 11] for details on the latter.

Interval routing is a table-based scheme that provides an efficient way to implement deterministic routing schemes [VLT87]. It works in the following way. In a network with N destinations, each is labelled with a unique value from the range 0 to $N-1$. Every link is then assigned a contiguous interval of labels, non-overlapping with other links at a given node. A packet is routed along a given link if its destination lies within the link's label range. Intervals are stored in a table and this requires $O(d)$ space where d is the degree of the node. Routing decisions can be made very quickly as they are based on table lookups, which maximises the benefit of wormhole switching.

Interval routing information can be assigned to links independently. This provides two key abilities [MTW93, Ch. 3]. First, with *grouped adaptive routing*, links can be bundled together between two switches and assigned the same intervals. Then, a link can be chosen adaptively based on which one becomes available first. Second, the network can be *partitioned* into independent components by assigning routing intervals to disjoint sets of links.

A simple way to implement two-phase randomised routing is to transmit packets with two headers, one to specify the randomly chosen intermediate node and the other to specify the final destination [MTW93, Ch. 1]. Both headers contain a flag to indicate whether the packet is in the first or second phase of routing. When the first header arrives at the intermediate node, it is recognised as being in the first phase and discarded. When the second header then arrives, it is used to initiate the second phase and it is sent towards the destination. In direct networks, the network must be partitioned to separate the routing of packets between the two phases and prevent deadlock. In indirect networks, links are partitioned into an 'up' component for the first phase with intervals corresponding to a labelling of the middle-stage switches and a 'down' component for the second phase with intervals corresponding to the labelling of the terminals.

4.3. Processing

The chapter so far has discussed the structure and operation of a universal interconnection network that is parameterised by just two variables: the communication latency ℓ and the communication grain g. This section discusses the general characteristics of the processors that are connected to it and how they affect the capability of the system.

4.3.1. Network interface

The *network interface* sits between the data paths of a processor and the network, providing a connection to a particular switch, thereby allowing it to engage in communication with other processors.

The network interface performs a translation of messages from the processor into the format required by the network switches, or vice versa, thereby implementing any *link-level* network

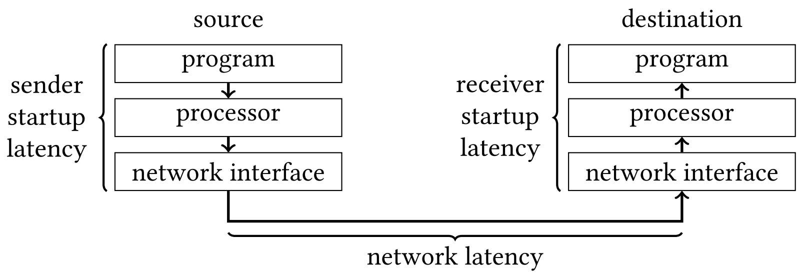


Figure 4.11.: An illustration of the components of communication latency, highlighting the components of the startup latency at both the sender and receiver.

protocols. Wormhole switching, for example, requires a message to be broken up into flits and additional flits added that contain routing information. This might be performed by the network interface. It will also implement link-level protocols, for example to perform credit-based flow control by generating additional messages, and the physical network layer, which is concerned with the encoding and transmission of bits onto the wires implementing the link itself.

Communication startup latency

The processor's communications system, including its network interface, contributes a fixed proportion of the message latency, with the rest determined by the interconnect. This is illustrated by Figure 4.11, which shows the contributing components.

The overhead in setting up a communication depends on the interface between a processor and the interconnect and the efficiency of the following three mechanisms, operating at either end of a communication channel. The latency they contribute is typically not dependent on the message size.

1. *Software latency.* The latency of dispatching or receiving a communication to or from a program to the hardware. This relates to the preparation and processing of a message as well as the traversal of software between the programs and hardware, including the overhead of additional control code, procedure calls and any associated memory accesses.
2. *Data movement latency.* The transfer of data from registers in the processor onto the link. When this involves main memory and the associated mechanisms of *direct memory access* (DMA) and interrupts, the overhead can be significant.
3. *Notification to receiver latency.* Notification of the receiving process that it is being sent a message. If this is an interrupting event, then it might be necessary to perform a context switch, again accessing memory, to execute the receiving process.

Communication latency can therefore be reduced by:

- closely integrating the network interface with, or in, the processor architecture to reduce the critical path between communication operations in a program to the data entering the network;
- by avoiding the use of main memory for messages and instead using fast dedicated memory.

Several processor architectures have employed these optimisations to reduce communication latency. The most prominent examples are the INMOS transputer [INM88c] and the descendent XMOS XS1 architecture [May09], which provide communication operations at an architectural level. This allows messages to be transferred directly from the processor pipeline into message buffers ready to be transmitted onto the network. In the XS1 architecture, the execution of a send operation will move the data in a single cycle from the processor registers into a message buffer. In the next cycle, the message can be transmitted on to the network [May11]. Additionally, the provision of hardware support for a number of processes with dedicated registers and a simple scheduler avoids context-switch overhead [May10].

A similar approach is taken in the *Message-Driven Processor* (MDP) of the *MIT J-Machine* [NWD93], which provides architectural integration of communication operations with dedicated data paths and hardware scheduling of processes based on message reception [DW89, DFK+92, DCC+98]. Avoiding main memory for messaging is also supported by experience with the *Intel Single-Chip Cloud Computer* (SCC), where a factor of 15 improvement in latency was observed by moving messages onto the network using a small 16 KB on-chip buffer memory, rather than using the main off-chip memory [HDH+10]. In contrast to these 'fine-grained' architectures, the setup cost of communication in the *BlueGene family of supercomputers*, for example in the Q system [CEH+11, HOF+12], is much higher at around 700 ns, which corresponds to around 1000 local computational operations. This is particularly surprising since the latency of one hop between two switches in the network is around 45 ns, a factor of 15 difference.

Remote memory access

Remote memory access (or *remote direct memory access* (RDMA) as it is also known) is a mechanism for accessing remote memories in distributed-memory systems. It is implemented in the network interface and bypasses the execution pipeline of the processor.

RDMA can be supported directly by the interconnect and network interface to avoid any software overheads at the target processor in servicing an access. To do this, a remote memory access is made by sending a message prefixed with a special header. Instead of the message being received by software process, it is intercepted by a mechanism in the network interface and connected to the memory system that performs the access and generates a response to complete the transaction.

RDMA is widely used in conventional computer networks and supercomputers where the complexity of the operating system and software stack would otherwise add significant latency. The *Quadrics interconnect* [PFH+02] and the *Cray Gemini interconnect* [ARK10] for example, both support RDMA. Similar schemes have also been proposed for chip-level networks such as [JM02].

4.3.2. Efficient support for parallelism

The finest level of parallelism is between individual operations and called *instruction-level parallelism*. Superscalar processors are designed to exploit this and require a large amount of specialised logic to identify when operations are independent and when to dispatch them to different execution units and combine their results. However, there is little inherent parallelism in typical sequential instruction streams and this type of parallelism can deliver at most a factor of 2 to 4 speedup; the sequential capabilities of processors is discussed in §9.6.1 [p. 200].

The other more general form of parallelism is between streams of instructions and called *task-level parallelism*. Exploitation of this requires the creation and termination of parallel processes. Clearly, for this to be economical, the cost of creating, initialising, and terminating a parallel task, at the very least, must be less than the amount of work it subsequently performs otherwise the work could more quickly be performed sequentially. Furthermore, communication and synchronisation are required to coordinate the activity of a collection of processes, and the overheads of these will limit the level of parallelism in the same way. It is only economical to communicate data when the cost of doing so is less than the resulting work performed by the receiving process, and it is only economical if the additional cost of synchronising between more processes is less than the benefits from increasing their number.

There is abundant parallelism in most problems (or conversely, very few problems are inherently sequential) but the ability to exploit parallelism depends on the time taken to create and terminate parallel processes and synchronisation and communication latency. Systems in which these mechanisms are inefficient can only be applied to problems that provide large amounts of work to obtain speedups from parallelism. With the MDP of the J-Machine, it is economical to create processes of a few hundred operations [NWD93]. With the transputer and XS1 architecture, a process can be created and initialised in a few tens of operations and processes executing on the same processor can synchronise in a single cycle (equivalent operations between processors can be implemented with message passing and their performance will depend on the network).

4.4. A short survey of real machine architectures

This section briefly surveys some of the overall trends in real parallel architectures. For brevity, this focuses on the interconnection topology and communication model, rather than the details of the processor and interconnect architecture.

Table 4.2 provides a chronological summary of a selected set of systems, both large multi-chip systems and more recent systems integrated onto a single chip.

4.4.1. High-performance computing

HPC/supercomputer systems have been, and remain, at the forefront of the large scale use of parallelism to improve performance. Distributed memory architectures have proven to be the most practical and economical way to scale the computational capacity with available technology to deliver the highest levels of performance.

Distributed memory machines with interconnection networks based on hypercubes were popular in the mid-1980s with systems such as the *nCUBE 10* [HMS+86], *Cosmic Cube* [Sei85], *Connection Machine 1* [TR88] and *Intel Personal Super Computer* (iPSC) [Bok90, Arl88]. However, this popularity shifted at the end of the 1980s to the use of 2- and 3-dimensional mesh- and torus-based interconnects, likely due to the implementation difficulties associated with hypercubes and influential work by Dally [Dal90]. Despite the popularity of low-dimensional networks, multistage networks based on folded-Clos/fat-tree topologies have also been popular, in systems such as the *Meiko Computing Surface 2* [BCM94], the *Connection Machine 5* [LAD+92] and the *IBM Roadrunner* [BDH+08]. There are several likely reasons for this popularity, one is that they are straightforward to build out of commodity networking equipment, whereas direct networks require more specialised switches. The other is that multistage networks are effective in distributed-memory systems that implement a global address space. Examples include SGI systems, such as the *Origin 3000*, with the *NUMALink interconnect* [WRRF05].

Typical modern high-performance systems employ a custom interconnect since the performance of commodity networking equipment is not satisfactory, but they generally use commodity components at the compute nodes. This includes multicore shared-memory processors, conventional DRAM and in some systems accelerator-devices, such as the *Nvidia Tesla K20* [NVI12] GPU, and more general-purpose devices such as the *Intel Xeon Phi* architecture [Int13]. The use of commodity components is due to the economies of scale gained from consumer markets weighed against the enormous cost of developing custom chips.

4.4.2. Single-chip systems

The need for scalable general-purpose systems has led to a number of proposals for explicitly parallel tiled architectures, where a system is integrated on-chip as a regular arrangement of processor-memory pairs connected by an interconnection network. This approach allows memories to be more tightly bound to processors and local and access latency to be reduced. Prominent examples include the *Stanford FLASH* [KOH+94], the *MIT RAW* [WTS+97] and its descendant *Tilera Tile architecture* [WGH+07], *Smart Memories* [MPJ+00], the *Ambric Massively Parallel Processor Array* [BJW07]. and the *Kalray Multi-Purpose Processor Array* (MPPA) [Kal12]. However, despite the low-diameter high-capacity networks required for general-purpose parallel computation, these systems all employ

System	**Citation**	**Appeared**	**Interconnect topology**	**Communication model**
		Multi-chip/high-performance systems		
Cray 1	[Rus78]	1976	custom	SIMD/SM
ALICE	[DR81]	1981	Delta*	DF
NYU Ultracomputer	[GGK$^+$82, Got86]	1982	Omega*	SM
CM-1	[TR88]	1983	hypercube	MP
Cosmic Cube	[Sei85]	1985	hypercube	MP
nCUBE 10	[HMS$^+$86]	1985	hypercube	MP
Intel iPSC	[Bok90]	1985	hypercube	SM
Manchester Data-flow computer	[GKW85]	1985	ring	DF
Meiko Computing Surface	[Mei88]	1986	2D mesh	MP
GRIP Machine	[PJCSH87]	1987	bus	DF
Data Diffusion Machine	[WH88]	1988	folded Clos	SM
CMU/Intel iWARP	[BCC$^+$88]	1988	2D torus	DF
Intel iPSC/2	[Arl88]	1988	hypercube	SM
MIT J-Machine	[NWD93]	1990	3D mesh	MP
MIT Alewife	[ACJ$^+$91]	1991	2D mesh	SM & MP
Stanford DASH	[LLG$^+$92]	1992	2D mesh	SM
Cray T3D	[KS93]	1993	3D torus	MP
Meiko CS-2	[BCM94]	1993	folded Clos	MP
CM-5	[LAD$^+$92]	1995	folded Clos	MP
SGI Origin 3000	[WRRF05]	2000	folded Clos	SM
IBM BlueGene/L	[GBC$^+$05]	2004	3D torus	MP
Maxeler MPC	[DFMP08]	2008	ring	DF
IBM Roadrunner	[BDH$^+$08]	2008	folded Clos	MP
XMOS XMP-64	[XMO10]	2010	hypercube	MP
Cray XK7	[ARK10]	2012	3D torus	MP & SM
		Single-chip systems		
Stanford FLASH	[KOH$^+$94]	1994	2D mesh	SM & MP
Berkeley IRAM	[PAC$^+$97]	1997	crossbar	SIMD/SM
MIT RAW	[WTS$^+$97]	1997	2D mesh	DF
Smart memories	[MPJ$^+$00]	2000	2D mesh	SM
PicoChip picoArray	[PTD$^+$06]	2000	2D mesh	MP
Clearspeed CSX	[Cle06]	2002	2D mesh	SM
UC Davis AsAP	[TCM$^+$09]	2005	2D mesh	DF
Ambric MPPA	[BJW07]	2006	2D mesh	MP
XMOS XS1-G4	[XMO12b]	2008	crossbar	MP
Intel SCC	[HDH$^+$10]	2009	2D mesh	MP
Tilera Tile	[WGH$^+$07]	2010	2D mesh	SM
IntellaSys SEAforth	[Int08]	2010	2D mesh	MP
Adapteva Epiphany	[Ada11]	2011	2D mesh	SM
Kalray MPPA	[Kal12]	2012	2D mesh	MP
Intel Xeon Phi	[Int13]	2013	ring	SM
Nvidia Tesla K20	[NVI12]	2013	custom	SIMD/SM

Table 4.2.: A short survey of real parallel machine architectures, divided into systems comprised of multiple chips, which are generally physically large and mostly built for HPC-type applications, and systems integrated on a single chip. The following abbreviations are used: data-flow (DF), message passing (MP), shared memory (SM) and single-instruction multiple-data (SIMD). The Delta and omega networks marked with a '*' are indirect multistage and closely related to a folded Clos.

a 2D mesh interconnect in which communication latency scales linearly with the number of tiles and programs must be carefully mapped to preserve locality to obtain good performance.

Indeed, where the class of programs is limited to a specific domain, low-dimensional array interconnects can be effective. Examples of such systems include the *UC Davis Asynchronous Array of Simple Processors* (AsAP) [YMA+08], the *PicoChip picoArray* [PTD+06] and the *Adapteva Epiphany* [Ada11], all for *digital signal processing* (DSP). In contrast, the INMOS transputer [INM88c] and descendant XMOS XS1 architecture [May09] were designed to support general-purpose parallelism with high-performance networks [MTW93, May10].

4.5. Summary

This chapter has described the essential ingredients of a scalable general-purpose parallel computer architecture. It first described the theoretical aspects of a communication network that can support arbitrary patterns of communication and then then practical aspects of implementing such a network.

A network is *universal* in the sense that it can simulate any other network efficiently. This means that when the nodes of a *guest* network to be simulated are mapped to the nodes of a *host* network of the same size, the host can support any pattern of communication in the guest network with at most a logarithmic-factor slowdown in the size of the host network (larger guest networks will require additional time proportional to the difference in size).

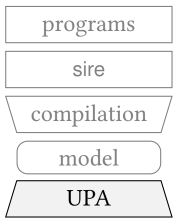
Hypercube networks are universal for *off-line* simulations when the pattern of communication to be simulated is known in advance, but for general parallel computations, the pattern will not always be known. By using Valiant's *two-phase randomised routing* scheme [VB81], simulations can be performed efficiently *on-line*. Hypercubes themselves pose problems to build because the degree of nodes is not fixed, but there are a number of fixed-degree hypercube variants that are better suited. Because these appear widely in the literature, they were surveyed and their relationships described to show that the theoretical results for one variant also apply (up to constant factors) to the others.

A universal communication network can be characterised by just two parameters, the *communication latency*, ℓ, and the *communication grain*, g, which is the ratio between the rates of computation to communication. The theoretical properties of the network hold asymptotically; it is the details of a specific implementation that determine their absolute values. In particular: a *high-degree switching element* allows multiple terminals to be attached to reduce the network diameter; *wormhole switching* minimises latency, requires only small amounts of buffering and supports both variable-sized packets and circuits for streaming; and *interval routing* can be used to implement two-phase routing and it enables the adaptive use of sets of links between the same nodes to distribute load.

Finally, the processor and its interface to the network determine the *startup latency*, a crucial component of both ℓ and g, and the ability of the processor to exploit all of the available parallelism in a particular problem depends on how quickly parallel processes (executing on the same processor or a different one) can be created, can synchronise and can be terminated.

Part II.

THE UPA AND THE SIRE LANGUAGE



Chapter 5.

The Universal Parallel Architecture

This chapter describes the *Universal Parallel Architecture* (UPA). It is *scalable*, *general purpose*, *simple* to implement and it provides the natural target for the compilation of sire programs.

5.1. Overview

The UPA is a programmable computer architecture that consists of a set of *tiles*, each containing a processor, a memory and a network interface. The tiles are connected by an *interconnect* that supports efficient communication between all pairs of tiles.

The UPA is *universal* in the sense that it can be *programmed* to perform the function of any parallel machine with a reasonable degree of efficiency. It can therefore execute arbitrary programs efficiently and this provides *independence* of the structure of any UPA implementation from that of any program; a particular machine is characterised only by the communication grain g and latency ℓ. This simplifies the compilation of programs and facilitates portability between different implementations.

The interconnect supports efficient packetised and streamed communication, which provides a firm basis for a range of programming paradigms (each with different *parallel structures*), including message-passing, PRAM-style shared memory and conventional sequential execution. These can also be *combined* in programs that allocate and deallocate processors dynamically.

It is possible to implement the UPA efficiently with current technologies. A single UPA chip can contain hundreds or thousands of tiles and multiple chips can be combined to create larger systems.

Figure 5.1 below illustrates the software-level view of the UPA.

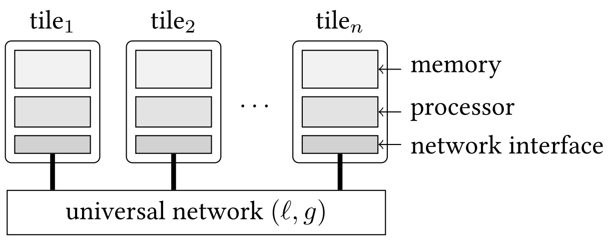


Figure 5.1.: A high-level view of the UPA. A collection of processor-memory tiles are connected by a universal network that provides efficient communication between all pairs of tiles with a latency ℓ and at a rate g.

5.2. Interconnect

The UPA interconnect combines a *universal network* and *universal routing strategy* to support arbitrary patterns of communication with scalable throughput and low bounded delay. A *Clos network* [Clo53] can support universal two-phase randomised routing [VB81] with no increase in the average path length and has a flexible structure that facilitates a simple and efficient implementation.

The implementation of the network follows the design of the C104 switch [MTW93, Ch. 3]:

- a deterministic routing scheme is implemented with *interval routing* [VLT87];

- *two-phase randomised routing* [Val82] is implemented by generating packets with two headers (one to specify a randomly-chosen middle-stage node and the other to specify the destination) and by partitioning the network into 'up' and 'down' components for the two phases;
- sets of links between two switches can be used adaptively with *grouped adaptive routing* (assigned the same intervals);
- data is transmitted on the network using *wormhole switching* because this supports both variable-sized packet switching and circuit switching, minimises latency and requires only small amounts of buffering.

5.3. Processing

The universal capability of the UPA stems from the combination of a *universal communication network* able to support arbitrary patterns of communication and *universal sequential processors* able to execute arbitrary sequential programs. The *XMOS XS1 architecture* [May09] is used for the processor cores because it is universal and provides direct support for communication and parallelism. However, the particular choice of architecture for the UPA is relatively unimportant but there are a number of high-level considerations.

At a minimum, it is essential that a UPA processor can support the concurrent execution of a collection of processes, concurrent input and output, and a rate of communication that matches the capability of the interconnect.

To maximise the level of parallelism that can be exploited from particular problems with limited sizes, it is essential to minimise the communication startup latency, because this contributes a fixed portion of the network latency, ℓ, and to minimise the overheads of process creation, synchronisation and termination.

Ideally, the execution of a UPA processor is *deterministic* to eliminate unexpected variations in timing called *jitter*. This is because the complexities of conventional sequential processors to improve performance, such as pipelining, superscalar execution and branch prediction reduce the predictability of execution timing, which can be magnified with high levels of parallelism and cause significant inefficiency.

Finally, reducing the complexity of a processor also has the advantage of reducing the area to maximise the number of processors that can be integrated on to a chip (this is discussed further at the end of Chapter 9).

5.4. Memory

Larger memories occupy more area and incur longer access latencies. When latency is significantly more than the time for basic processor operations, the use of caches is necessary, at the expense of predictability and area. The approach of the UPA is to provide each processor with a small memory that has an access latency close to the time for a basic processor operation but also with the ability to efficiently access the memory of other tiles. This allows tiles to be viewed as units of processing or storage and results in a system where the amount of memory per tile does not need to be over provisioned because larger memories can be *emulated* with collections of tiles (this capability is investigated in Chapter 10) and longer access latencies over the network can be dealt with in software, for example by implementing a cache or by moving program components to operate in-place on data.

Memory accesses from remote tiles is supported primarily by the interconnect with low-latency communication of arbitrary-sized messages, but is optimised with a mechanism for *remote memory access*. This is implemented in the network interface and avoids any software overheads in accessing memory. The remote memory access mechanism can be targeted directly at compilation to treat tiles as memories to effectively increase the ratio of memory to processing.

The size of a tile memory should be chosen so that the UPA delivers good performance over all programs, where at one end of this spectrum they are bound to the performance of the processor and at the other end they are bound to the performance of the memory. With efficient inter-tile accesses, this implies a balanced provision of memory and processing, and as such can never exhibit more than a factor of two inefficiency for any program, whether it is a large-memory sequential program or a highly-distributed parallel program.

For some cases it is possible that particular implementations of the UPA to be specialised towards computationally-intensive or memory-intensive workloads by altering the size of the tile memory.

5.5. Packaging

The regular and hierarchical structure of the UPA with a *folded Clos network* [JDFJ97, Ch. 3] allows it to be constructed from one or more identical smaller chips. A chip will contain a set of replicated tiles, switches and communication links and can be produced at a size to provide the best economic trade-off between performance and system cost, similar to the production and packaging model of commodity DRAM.[1] A particular number of chips are chosen to build a system to provide a certain memory or processing capacity.

A folded Clos network can be rearranged to enable a hierarchical multichip packaging with a complete sub-folded Clos integrated on a single chip and for multiple chips to be combined with connections to a new middle stage. This is achieved by connecting tiles to the original middle-stage switches so that the network can be recursively expanded and the 'longer' connections occur closer to the new middle-stage switches (see Figure 4.6c in relation to Figure 4.6a). A chip's middle-stage switches will have links connected to off-chip IO so that larger systems can be built with multiple chips by directly extending the network with electrical or optical connections.

The regular recursive structure of a folded Clos provides further flexibility in an implementation. First, it can be constructed from arbitrary fixed-degree switches, allowing high-degree ones to be used. With these, a good trade-off between the switch complexity and resulting network diameter can be obtained, and multiple tiles can be connected to each switch. Second, because it consists of a number of *levels*, each level can be constructed independently, allowing the use of different-degree switches and different provisioning of bandwidth between stages.

A packaging of the UPA based on this approach used for a detailed multichip implementation model using current technology is presented in Chapter 9.

[1] A low cost-performance ratio is the primary driver in the production of commodity DRAMs. The basic memory cells and supporting circuits are replicated millions of times on a chip and consequently the designs are highly optimised. DRAM chips are produced in high volumes and are produced at a size which makes the best trade-off between cost and performance, but run much slower than logic devices such as processors.

CHAPTER 6.

THE SIRE PROGRAMMING LANGUAGE

Sire[1] is a communicating-process programming language designed for the highly-parallel distributed-memory UPA. Sire provides facilities for sharing and abstraction that can be used to build distributed parallel subroutines and shared-access data structures; its design is based on the occam programming language and, in the occam philosophy, includes the smallest set of features necessary to this. Sire is capable of being compiled using simple techniques to produce efficient programs, allowing the programmer to directly exploit the UPA.

This chapter presents the definition of the sire, which introduces the language incrementally, with each section building on the last. It concludes with a discussion about its relationship with occam and other language proposals that have provided inspiration to its design.

6.1. The model of computation

A sire program consists of a collection of *processes*. Each process performs a sequence of commands, as well as being able to create additional processes. Processes can communicate with each other by using point-to-point message passing *channels*, which consist of a connected pair of *channel ends* local to either process. Communication also causes processes to synchronise; a sending process waits until the receiver is ready to receive a message. A special type of process called a *server* provides a means of abstraction and forms the basis of a parallel subroutine mechanism.

A sire program is executed by a collection of one or more *processors*. Each processor has a small private memory and can execute a number of *processes* simultaneously. The execution begins on a single processor and the computation progresses in *time* and *space* as processors are allocated and deallocated dynamically.

6.2. Notation

Since sire builds on occam, it is prudent to present the language in a similar way to the original occam specifications,[2] as its successors have also done [May83, Bar92, SGS95]. The presentation uses a modified version of the *Backus-Naur Form* (BNF) to specify the syntax of the language. In the BNF, a *production rule* defines a new *symbol* to consist of a choice of one or more sequences of other symbols, which are defined by other rules or are *terminal* and therefore elements occurring in the program. Taking part of the command syntax as an example, a production of the form

$$\textit{command} = \langle\text{input}\rangle \quad | \quad \langle\text{output}\rangle$$

specifies the symbol ⟨process⟩ to be the symbol ⟨input⟩ or ⟨output⟩. This can also be written with separate rules in the equivalent definition

[1]The word *sire* is a verb meaning to create or bear. The choice of this for the name of the language relates to the way in which a sire program executes, by allocating and deallocating processors dynamically.

[2]This style of specification has gained inspiration from a number of other languages including *Algol* [NBB$^+$63], BCPL (*Basic Combined Programming Language*) [Ric67], *Pascal* [Wir71] and the original proposal for CSP [Hoa78].

$$command = \langle\text{input}\rangle$$
$$command = \langle\text{output}\rangle$$

in order to incrementally introduce elements of the syntax. The symbols ⟨input⟩ and ⟨output⟩ are specified with the rules

$$input = \langle\text{chanend}\rangle \; \mathbf{?} \; \langle\text{variable}\rangle$$
$$output = \langle\text{chanend}\rangle \; \mathbf{!} \; \langle\text{expression}\rangle$$

The emboldened symbols '**?**' and '**!**' are terminals and ⟨chanend⟩, ⟨variable⟩ and ⟨expression⟩ are defined in terms of one or more additional production rules.

The notation $\{_c X\}$ specifies a list of c or more Xs and is equivalent to $X_1 X_2 \cdots X_c \cdots$, and $\{_c \oplus X\}$ specifies a list of c or more Xs separated by $\oplus$ and is equivalent to $X_1 \oplus X_2 \oplus \cdots \oplus X_c \oplus \cdots$.

As each part of the language is introduced, fragments of the BNF are given and the semantics of the syntax explained. The semantics of a small subset of the language are given informally, which allows the remaining portion to be defined algebraically, building on the subset. This approach is attractive since it provides clear semantics and allows simple transformations between different constructs. The full syntax of the language is included in Appendix A.

Where examples of sire syntax are given the notation $[\![\cdot]\!]$ is used to distinguish sire syntax from mathematical notations.

6.3. Overview

The definition is presented in the following sections.

6.4. Primitive commands

$$\begin{aligned} \textit{command} &= \langle\text{assignment}\rangle \\ &\mid \langle\text{input}\rangle \\ &\mid \langle\text{output}\rangle \\ &\mid \langle\text{connect}\rangle \\ &\mid \langle\text{skip}\rangle \\ &\mid \langle\text{stop}\rangle \end{aligned}$$

A program is built from commands and the simplest of these are *primitive* commands, which are *assignment*, *connect*, *input*, *output*, *skip* and *stop*.

6.4.1. Assignment

$$\textit{assignment} = \langle\text{variable}\rangle \texttt{ := } \langle\text{expression}\rangle$$

An *assignment command* evaluates the right hand side expression and changes the value of the variable to the result.

Example

```
v := 1
```

6.4.2. Input and output

$$\begin{aligned} \textit{input} &= \langle\text{chanend}\rangle \textbf{ ? } \langle\text{variable}\rangle \\ \textit{output} &= \langle\text{chanend}\rangle \textbf{ ! } \langle\text{expression}\rangle \end{aligned}$$

Values are passed between processes using *bidirectional communication channels*. These consist of two *channel ends* that are associated with either process communicating on a channel. Communication is *synchronised*, which means that an outputting process must wait for the inputting process to be ready before any data is sent. This prevents any data from being lost due to programming errors and precludes the use of buffering.

An *input command* receives a value from a channel and assigns it to a variable and an *output command* evaluates an expression and sends the result on a channel. Matching pairs of input and output are a distributed form of assignment.

Example

The command

```
s ! 3
```

outputs the value 3 on the channel end `s` and the command

```
t ? v
```

inputs a value from the channel end `t` and sets it to the variable `v`. When `s` and `t` appear in separate processes and are *connected* it is equivalent to the assignment

```
v := 3
```

6.4.3. Connect

$$\begin{aligned} \textit{connect} &= \texttt{\textbf{connect}}\ \langle\text{chanend}\rangle\ \texttt{\textbf{to}}\ \langle\text{chanend}\rangle \\ \textit{chanend} &= \langle\text{element}\rangle \end{aligned}$$

A *connection* between two channel ends is established with a matching pair of *connect* commands. Each connect command connects a *local* channel end to the corresponding *remote* channel end. The commands terminate when the connection has been established. The channel end itself is an element that specifies a name or field (see §6.10 [p. 90]).

Example

```
connect s to t
```

6.4.4. Skip and stop

$$\begin{aligned} \textit{skip} &= \texttt{\textbf{skip}} \\ \textit{stop} &= \texttt{\textbf{stop}} \end{aligned}$$

The command **skip** does nothing and terminates, the command **stop** never terminates, causing the process to execute no more commands. These commands are used primarily to explain the behaviour of other constructs in the language.

6.5. Structured commands

$$\begin{aligned} \textit{command} &= \langle\text{alternation}\rangle \\ &\mid \langle\text{conditional}\rangle \\ &\mid \langle\text{loop}\rangle \end{aligned}$$

The structured commands *conditional*, *alternation* and *loop* are control constructs that are used to combine commands.

6.5.1. Alternation

$$\begin{aligned} \textit{alternation} &= \texttt{alt \{ } \{_0 \texttt{ | } \langle\text{alternative}\rangle \} \texttt{ \}} \\ \textit{alternative} &= \langle\text{guarded-alternative}\rangle \\ &\mid \langle\text{alternation}\rangle \\ \textit{guarded-alternative} &= \langle\text{guard}\rangle \texttt{ : } \langle\text{command}\rangle \\ \textit{guard} &= \langle\text{input}\rangle \\ &\mid \langle\text{expression}\rangle \texttt{ \& } \langle\text{input}\rangle \\ &\mid \langle\text{expression}\rangle \texttt{ \& } \langle\text{skip}\rangle \end{aligned}$$

An *alternation command* is used to deal with non-determinism by allowing a process to wait for inputs on a number of different channels. It consists of a list of *guarded command alternatives.* A guard consists of an expression and an input on a channel or **skip** in the place of the input if none is required. When the value of the expression is **true**, the guard behaves like the input or **skip**, otherwise it behaves like **stop** and does not proceed. Conceptually, the guard expression *enables* or *disables* an alternative.

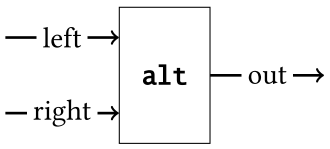
When an alternation command is performed, it behaves like one of the alternatives that can proceed. Without any alternatives, it behaves like **stop**. An alternative that is itself an alternation is ready when one of its alternatives is ready.

Examples

- The alternation command

  ```
  alt { left ? v: out ! v | right ? v: out ! v }
  ```

  merges data from the channel ends `left` and `right` onto a a single channel end `out`. This is illustrated by the diagram

- Guarding inputs prevents the input and its corresponding action when a Boolean condition is false; the command

  ```
  alt { enabl & left ? v: out ! v | enabr & right ? v: out ! v }
  ```

  includes Boolean guards on each input so that when `enabl` or `enabr` are false, a process outputting to the channel ends `left` or `right` (respectively) will be blocked.

- Alternatives in a nested alternation can be included in the parent alternation with the same effect:

  $$[\![\texttt{alt \{ alt c ? v: c ! v \}}]\!] = [\![\texttt{alt \{ c ? v: c ! v \}}]\!]$$

6.5.2. Conditional

$$
\begin{aligned}
\textit{conditional} &= \textbf{if \{ } \{_0\ \textbf{|}\ \langle\text{choice}\rangle\ \textbf{\}} \\
&\mid\ \textbf{if}\ \langle\text{expression}\rangle\ \textbf{then}\ \langle\text{command}\rangle\ \textbf{else}\ \langle\text{command}\rangle \\
\textit{choice} &= \langle\text{guarded-choice}\rangle \\
&\mid\ \langle\text{conditional}\rangle \\
&\mid\ \langle\text{specification}\rangle\ \textbf{:}\ \langle\text{choice}\rangle \\
\textit{guarded-choice} &= \langle\text{expression}\rangle\ \textbf{:}\ \langle\text{command}\rangle
\end{aligned}
$$

A *conditional command* consists of a list of expressions enumerating a set of choices, with a command associated with of each. When a conditional command is performed, each choice is tested in sequence and the command behaves like the first choice that evaluates **`true`**. Without any choices, it behaves like **`skip`**.

An alternative *if-then-else* form of the conditional command combines two choices, one of which is performed when the value of a condition expression is **`true`**, and the other when it is false. Let e be an expression and C and D be commands, then the alternative conditional form is defined as

$$[\![\textbf{if}\ e\ \textbf{then}\ C\ \textbf{else}\ D]\!] = [\![\textbf{if}\ \texttt{\{}\ e\texttt{:}\ C\ \texttt{|}\ \texttt{\textasciitilde}e\texttt{:}\ D\ \texttt{\}}]\!]$$

Examples

```
if { i=1: c ! 1 | i=2: c ! 2 | i>2: c ! 13 }
```

The following equivalences hold:

$$[\![\texttt{if i=0 then c ! 17 else c ! 19}]\!] = [\![\texttt{if \{ i=0: c ! 17 | i\textasciitilde=0: c ! 19 \}}]\!]$$

$$[\![\texttt{if i=0 then c ! 21 else skip}]\!] = [\![\texttt{if \{ i=0: c ! 21 \}}]\!]$$

$$[\![\texttt{if i=0 then skip else skip}]\!] = [\![\texttt{if \{ \}}\ ]\!] = [\![\ \texttt{skip}]\!]$$

6.5.3. Loop

$$\textit{loop} = \textbf{while}\ \langle\text{expression}\rangle\ \textbf{do}\ \langle\text{command}\rangle$$

A *looping command* repetitively evaluates a condition expression and executes a command if the value of the expression is **`true`**. When it is **`false`**, it terminates.

Example

The looping command

```
while true do { in ? v; out ! v }
```

acts as a buffer for a single value by repeatedly forwarding values from the channel end `in` to the channel end `out`.

6.6. Types, names and scope

6.6.1. Types

Primitive types

$$\begin{aligned} \textit{type} &= \langle\text{primitive-type}\rangle \\ \textit{primitive-type} &= \textbf{var} \end{aligned}$$

There is a single primitive *variable* type in sire. It holds a value that can be changed by input or assignment and it is specified by the keyword **var**. Variables are interpreted as signed integers and if no value has been assigned to it then the value is arbitrary (see Appendix A.4 for details of the representation of values).

Array types

$$\begin{aligned} \textit{type} &= \langle\text{array-type}\rangle \\ \textit{array-type} &= \langle\text{primitive-type}\rangle \\ &\mid \langle\text{array-type}\rangle \;\mathbf{[}\; \langle\text{expression}\rangle \;\mathbf{]} \end{aligned}$$

An *array variable* contains a number of component variables. Let T be a primitive type, then a 1-dimensional array of components of type T is specified by the type $T[e]$ where e is an expression specifying the length of the dimension. Arrays with additional dimensions can be declared by specifying additional lengths in the type. Let T be a primitive type, then $T[e_1][e_2]\cdots[e_d]$ specifies a d-dimensional array with $e_1 \times e_2 \times \cdots \times e_d$ components of type T.

The components of an array variable are accessed with integer-valued subscripts that can appear as an element in an expression or on the left hand side of an assignment or input. The *element* $u[i]$ selects the i^{th} component of a 1-dimensional array variable with the name u, and the subscript $v[i_1][i_2]\cdots[i_d]$ selects the i_1^{th} component of the first dimension and the i_2^{th} component of the second dimension etc. of a d-dimensional array variable with the name v.

6.6.2. Scope

A name has a context in which it is valid and can be used. This is called the *scope*. The scope of a name extends from the point at which it is declared to the end of the *block*. A block is enclosed by curly braces **{**$\cdots$**}** or a choice or alternative. The scope of a name also extends to any nested blocks.

6.6.3. Declarations

$$\textit{declaration} = \langle\text{type}\rangle \; \{_1 \,\mathbf{,}\; \langle\text{name}\rangle \;\}$$

A *declaration* $T\ n$ declares a variable of type T with the name n. A single declaration can specify multiple names of a particular type. Let $S(x)$ be a scope in which the name x is free, T be a type and $N_1, N_2, \cdots, N_n$ be names, then

$$[\![T \;\; N_1, N_2, \cdots, N_n \;\; \mathbf{:} \;\; S]\!] \;=\; [\![T \;\; N_1 \;\; \mathbf{:} \;\; T \;\; N_2 \;\; \mathbf{:} \;\; \cdots \;\; \mathbf{:} \;\; T \;\; N_n \;\; \mathbf{:} \;\; S]\!]$$

Examples

```
var u

var[23] u, v

var[29][31] u, v, w
```

6.6.4. Abbreviations

$$\begin{aligned}
\textit{abbreviation} &= \langle\text{specifier}\rangle\ \langle\text{name}\rangle\ \texttt{\textbf{is}}\ \langle\text{element}\rangle \\
&\mid\ \texttt{\textbf{val}}\ \langle\text{name}\rangle\ \texttt{\textbf{is}}\ \langle\text{expression}\rangle \\
\textit{specifier} &= \langle\text{primitive-type}\rangle \\
&\mid\ \langle\text{specifier}\rangle\ \texttt{\textbf{[ ]}} \\
&\mid\ \langle\text{specifier}\rangle\ \texttt{\textbf{[}}\ \langle\text{expression}\rangle\ \texttt{\textbf{]}} \\
\textit{element} &= \langle\text{element}\rangle\ \texttt{\textbf{[}}\ \langle\text{expression}\rangle\ \texttt{\textbf{]}} \\
&\mid\ \langle\text{field}\rangle \\
&\mid\ \langle\text{name}\rangle
\end{aligned}$$

An *abbreviation* can be used to specify new names for variables and values.

Variable abbreviation

A *variable abbreviation* specifies the name for an *element* that can either be a name or subscripted component of an array. Let T be a type, n be a name, e be an element and S be a scope. Then a variable abbreviation has the effect

$$[\![T\ \ n\ \ \texttt{\textbf{is}}\ \ e\ \ \texttt{\textbf{:}}\ \ S(n)]\!]\ \ =\ \ [\![S(e)]\!]$$

The length specifier for an array dimension can be omitted from an abbreviation to define a name of an array of components of the specified type of any length in that dimension. An array abbreviation is said to be *compatible* with an array variable if the lengths of all dimensions are equal or the abbreviation does not specify the length of a particular dimension.

Value abbreviation

A *value abbreviation* specifies a name for an expression. Let n be a name, e be an expression and S be a scope. Then a value abbreviation has the effect

$$[\![\texttt{\textbf{val}}\ \ n\ \ \texttt{\textbf{is}}\ \ e\ \ \texttt{\textbf{:}}\ \ S(n)]\!]\ \ =\ \ [\![S(e)]\!]$$

The value of the abbreviation must remain constant throughout its use. To ensure this, the abbreviated expression must not contain any variables that are updated by assignment or input in the scope of the abbreviation.

Examples

```
var n is v

val n is (m*x) + c

val n is m
```

6.6.5. Specifications

$$\begin{aligned}
\textit{specification} &= \langle\text{declaration}\rangle \\
&\mid\ \langle\text{abbreviation}\rangle \\
\textit{command} &= \langle\text{specification}\rangle\ \texttt{\textbf{:}}\ \langle\text{command}\rangle \\
\textit{alternative} &= \langle\text{specification}\rangle\ \texttt{\textbf{:}}\ \langle\text{alternative}\rangle
\end{aligned}$$

A *specification* introduces a new name into a scope and all names specified within a block must be unique, however, a block can introduce a name that is already in scope. In general, a free variable of a block is *bound* to the most recent name that is still in scope.

If $S(x)$ and $S(y)$ are scopes with the same behaviour, but $S(x)$ contains the free variable x wherever $S(y)$ contains the free variable y and vice versa, and $N(x)$ and $N(y)$ are identical specifications except that $N(x)$ specifies the name x and $N(y)$ specifies the name y, then

$$[\![N(x)\ \texttt{:}\ S(x)]\!] = [\![N(y)\ \texttt{:}\ S(y)]\!]$$

In other words, specifications can always be added to produce a block in which the free variables are distinct from any scope it may be inserted into. This provides a basis for the semantics of substitution for process definitions, server definitions and functions, which are all explained in §6.11 [p. 91].

Examples

```
var v: v := 73
```

The following equivalences with abbreviations hold:

$$[\![\texttt{var n is v: n := 37}]\!] = [\![\texttt{v := 37}]\!]$$

$$[\![\texttt{var n is v[41]: n := 43}]\!] = [\![\texttt{v[41] := 43}]\!]$$

$$[\![\texttt{var[] n is v: n[47] := 53}]\!] = [\![\texttt{v[47] := 53}]\!]$$

6.6.6. Rules

1. *Declarations.* A specification can only introduce declarations that have a primitive variable type.
2. *Valid abbreviations.* The specifier of the element specified in the abbreviation must be compatible with the type of the name. Compatibility requires the primitive types to be the same and, for array types, each specified length must be also be equal in the corresponding dimension (unspecified lengths can match any dimension).
3. *Abbreviation subscripts.* Any variables used in subscripts of an abbreviated variable cannot be changed in the scope of the abbreviation.
4. *Value abbreviations.* The value of an abbreviated variable may be changed but to ensure the abbreviation always refers to the same array component, then no variable in the subscript expression of that component can be changed by assignment or input.
5. *Array abbreviations.* Components of an array must be identified by a single name within a given scope to prevent aliasing. Therefore, components of an array that has one or more abbreviated components may also be referred to by abbreviations.

6.7. Composition

$$\begin{aligned} \textit{command} &= \texttt{\{}\ \langle\text{sequence}\rangle\ \texttt{\}} \\ &\mid\ \texttt{\{}\ \langle\text{parallel}\rangle\ \texttt{\}} \end{aligned}$$

6.7.1. Sequence

$$\textit{sequence} = \{_0\ \texttt{;}\ \langle\text{command}\rangle\ \}$$

A set of commands are composed in *sequence* with the '`;`' separator. Execution starts with the first component command and each subsequent command is executed if and when the preceding command terminates. The sequence terminates when the last component command terminates.

Example

```
a := 1; b := 3; c := 5
```

```
connect s to t; s ! 7
```

6.7.2. Parallel

$$\begin{aligned} \textit{parallel} &= \{_0\ \texttt{\&}\ \langle\text{parallel-component}\rangle\ \} \\ \textit{parallel-component} &= \langle\text{process}\rangle \end{aligned}$$

A *parallel command* creates new *processes* that execute in parallel, where a process is another command. A set of processes are composed in parallel with the '&' separator. It causes the component processes to be executed simultaneously and it terminates if and only if all of the component processes have terminated.

Process interfaces

$$\begin{aligned} \textit{process} &= \langle\text{interface}\rangle\ \texttt{:}\ \langle\text{command}\rangle \\ &\mid\ \langle\text{command}\rangle \\ \textit{interface} &= \textbf{interface}\ \texttt{(}\ \{_0\ \texttt{,}\ \langle\text{declaration}\rangle\ \}\ \texttt{)} \\ \textit{type} &= \langle\text{chanend-type}\rangle \\ \textit{chanend-type} &= \textbf{chanend} \end{aligned}$$

Processes executing in parallel can communicate via channels and a process can specify a number of local channel ends to do this. The set of *local* channel ends belonging to a process constitutes its *interface*, which is specified before the body of a process.

Let $N_1, N_2, \cdots, N_n$ be names, then

interface (**chanend** N_1, N_2, $\cdots$, N_n)

specifies an interface that declares n channel ends in the scope of the process. No other types may be declared in a process interface.

Process names

$$\begin{aligned} \textit{parallel-component} &= \langle\text{process-label}\rangle\ \langle\text{process}\rangle \\ \textit{process-label} &= \langle\text{name}\rangle\ \textbf{is} \end{aligned}$$

Channel connections are established between component processes of a parallel command by *naming* the processes. This is done by prefixing a process with a *label* of the form $[\![\langle\text{name}\rangle$ **is**$]\!]$. The name of a process acts as a prefix for a *compound name* for the channels in the interface, where each one can be selected as a *field*. Let P be a process and $N_1, N_2, \cdots, N_n$ be names, then

```
p is interface (chanend N₁, N₂, ···, Nₙ): P
```

is a process with the name p that specifies an interface with n components. Each component of the interface can be selected with the fields p.N_1, p.N_2, $\cdots$, p.N_n.

A process name is visible to all of the component processes in the parallel command and a process that is not named and does not engage in any channel communication is said to be *anonymous*. A process can make a connection to another parallel process by selecting the channel end name as a field using the process' name and using it as the target of a connect statement.

Process arrays

Arrays of processes can be constructed provided they have the same interface. Components of a process array are selected using integer-valued subscripts. Let $P_1, P_2, \ldots, P_n$ be processes with the same interface, then

```
p is {P₁, P₂, ···, Pₙ}
```

declares a *process array* containing n processes. The component processes are selected with the subscripts `p[0]` to `p[`$n-1$`]`. Process arrays can also be nested to create multidimensional arrays.

Examples

- The parallel command

  ```
  u := 1 & v := 2 & w := 3
  ```

  updates the values of the variables u, v and w simultaneously.
- The parallel command

  ```
  connect u to v & connect s to t
  ```

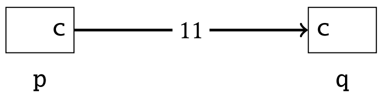
  connects the local channel ends u and s simultaneously, which might be necessary to avoid a connection deadlock.
- The process

  ```
  p is interface(chanend c): { connect c to t; c ! 7 }
  ```

  has an interface with one channel end c, which it connects to another channel end t then outputs the value 7 on it.
- The parallel command

  ```
  p is interface(chanend c): { connect c to q.c; c ! 11 } &
  q is interface(chanend c): { connect c to p.c; c ? v }
  ```

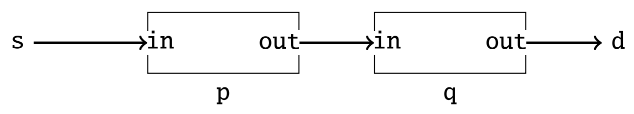
  is illustrated by the diagram:

- Combining two buffer processes in parallel allows two values to be buffered between an inputting channel s and outputting channel t:

```
p is interface(chanend in, out):
{ connect in to s; connect out to q.in;
  while true do { in ? v; out ! v } } &
q is interface(chanend in, out):
{ connect in to p.out; connect out to d;
  while true do { in ? v; out ! v } }
```

This is illustrated by the diagram:



- Let P, Q, R and S be processes with the same interface, then the process

  p **is** { P & Q & R }

  is a 1D array and the component process R, for example, is selected with the subscript `p[2]` and

  p **is** { { P & Q } & { R & S } }

  is a 2D array, this time R is selected with the subscript `p[1][0]` since it is the first component of the second array.

6.7.3. Rules

6. *Process disjointness.* Processes in a parallel command must be *disjoint* and only update disjoint sets of variables. This is to eliminate the possibility of unwanted race conditions, i.e. in the absence of alternation (which is explained in the next section), leaving alternation as the only means of introducing non-determinism.

   A variable or component of an array variable can be *changed* by assignment or input, or *read* if it occurs as an operand or in an expression. Disjointness is enforced by permitting *read-only access* to variables or components of array variables shared between components of a parallel command, and *exclusive write access* when a variable or array variable component appears in at most one component of a parallel command. With arrays, components of an array variable may only be changed in parallel if it can be determined at compile time that the array subscripts are used in a linear combination to select disjoint components of the array.
7. *Point-to-point channels.* No channel end can be connected to by more than one process. This is to ensure that no channel can be used for input or output by more than one process.
8. *Channel end subscripts.* If the target of a connect command is a component of a process array, the name will contain a subscript and the subscript must only use constants or replicator indices. This is so that the process to which it belongs can be identified at compile time.
9. *Nesting.* Channel connections can only be established between component processes of a parallel. Therefore, channel end names can only be used by child processes of the single parallel command and they cannot be passed to parallel commands nested in a process.
10. *Named process arrays.* Components of a named array must all present the same interface.
11. *Process interfaces.* A process interface can only declare channel end components.

6.8. Servers

A *server* is a special type of process that consists of special declarations that define its behaviour. It executes in parallel with the processes in its scope, is only active in response to *calls* and terminates when control flow leaves the block in which it is defined. *Client* processes of a server make calls that behave in the same way to a local procedure call.

Servers provide a mechanism for abstraction by building program components that provide a service or *subroutine*, and they can be combined into larger *server structures* to employ parallelism or distribute storage.

6.8.1. Specification

$$
\begin{aligned}
\textit{server} &= \langle\text{interface}\rangle \ \texttt{:} \ \langle\text{server-specification}\rangle \\
\textit{server-specification} &= \langle\text{declaration}\rangle \\
&\mid \ \texttt{\{} \ \{_1 \ \texttt{:} \ \langle\text{declaration}\rangle \ \} \ \texttt{\}}
\end{aligned}
$$

The specification of a server consists of an interface that defines means of external interaction (consisting of calls and channel ends) and set of declarations that define its behaviour. These are *initialisation commands* that are performed before it responds to anything, responses to each communication or call and *finalisation commands* that are performed when it terminates.

Interface

$$
\begin{aligned}
\textit{declaration} &= \texttt{call} \ \{_1 \ \texttt{,} \ \langle\text{name}\rangle \ \texttt{(} \ \{_0 \ \texttt{,} \ \langle\text{formal}\rangle \ \} \ \texttt{)} \ \} \\
\textit{abbreviation} &= \texttt{call} \ \langle\text{name}\rangle \ \texttt{is} \ \langle\text{name}\rangle
\end{aligned}
$$

A call declaration

$$\texttt{call} \ N \ \texttt{(}f_1, f_2, \cdots, f_n\texttt{)}$$

specifies N as a name for a call with the formal parameters $f_1, f_2, \cdots, f_n$. A single call declaration can specify multiple calls. Let F_k be a sequence of formals $f_1, f_2, \cdots, f_n$ where n is an integer defined for the sequence.

$$
\begin{aligned}
&[\![\texttt{call} \ N_1 \ \texttt{(}F_1\texttt{)}\texttt{,} \ N_2\texttt{,} \ \texttt{(}F_2\texttt{)}\texttt{,} \ \cdots\texttt{,} \ N_n \ \texttt{(}F_n\texttt{)}]\!] = \\
&\quad [\![\texttt{call} \ N_1 \ \texttt{(}F_1\texttt{)}\texttt{,} \ \texttt{call} \ N_2\texttt{,} \ \texttt{(}F_2\texttt{)}\texttt{,} \ \cdots\texttt{,} \ \texttt{call} \ N_n \ \texttt{(}F_n\texttt{)}]\!]
\end{aligned}
$$

Alternation and call guards

$$
\begin{aligned}
\textit{declaration} &= \langle\text{alternation}\rangle \\
\textit{guard} &= \texttt{accept} \ \langle\text{name}\rangle \ \texttt{(} \ \{_0 \ \texttt{,} \ \langle\text{formal}\rangle \ \} \ \texttt{)} \\
&\mid \ \langle\text{expression}\rangle \ \texttt{\&} \ \texttt{accept} \ \langle\text{name}\rangle \ \texttt{(} \ \{_0 \ \texttt{,} \ \langle\text{formal}\rangle \ \} \ \texttt{)}
\end{aligned}
$$

The main behaviour of a server is defined by an *alternation declaration* that specifies an alternation command (see §6.5.1 [p. 75] for the definition of alternation). This, as well as containing input alternatives, can contain *call* alternatives.

An *alternative* dealing with a call specifies the formal parameters that must match the list of formals in the specification in the interface of the corresponding call exactly, including the names, and a command to be executed when the call is invoked by a client. A call alternative may be guarded with an expression in the same way as an input can, so that the call is ready only when the guard is **`true`**. The effect is that the server will not respond to a call while the guard is false.

Initialisation and finalisation

$$\begin{aligned} \textit{declaration} &= \texttt{\textbf{initial}}\ \langle\text{command}\rangle \\ &\mid\ \texttt{\textbf{final}}\ \langle\text{command}\rangle \end{aligned}$$

An *initialisation declaration* specifies a command to be executed before the alternation. A *finalisation declaration* specifies a command to be executed after the alternation, when the scope of the server has terminated.

Examples

- The server

  ```
  interface(call inc(), dec()):
  { initial c := 0:
    alt { accept inc(): c := c+1
        | accept dec(): c := c-1 } }
  ```

  provides two calls to increment and decrement a count value that have no parameters, and the count is initialised to 0.

- The server

  ```
  interface(call inc(), dec()):
  { initial c := 0:
    alt { c < 100 & accept inc(): c := c+1
        | c > 0 & accept dec(): c := c-1 } }
  ```

  is a version of the previous server with guarded calls. When the server is shared by two or more processes that increment and decrement the count, no process will be able to increment the count above 100 and no process will be able to decrement the count below 0. If they attempt to either of these, they will become blocked until the count enters the valid range.

- The server

  ```
  interface(call read(var v), write(val v)):
    var x: alt { accept read(var v): v := x
               | accept write(val v): x := v }
  ```

  acts like a (potentially shared) variable that can be read from and written to.

6.8.2. Declarations

$$\begin{aligned} \textit{declaration} &= \langle\text{server-declaration}\rangle \\ &\mid\ \langle\text{hiding-declaration}\rangle \\ &\mid\ \langle\text{simultaneous-declaration}\rangle \end{aligned}$$

Server declarations

$$\begin{aligned} \textit{server-declaration} &= \langle\text{name}\rangle\ \texttt{\textbf{is}}\ \langle\text{server}\rangle \\ \textit{server} &= \langle\text{server-array}\rangle \\ \textit{server-array} &= \texttt{\textbf{[}}\ \{_1\ \texttt{\textbf{,}}\ \langle\text{server}\rangle\ \}\ \texttt{\textbf{]}} \end{aligned}$$

A *server declaration*

n **is** S

declares `n` as a name for the server S and `n` acts as a prefix for the compound names of the call and channel names specified in the interface of the server.

Arrays of servers can be constructed provided they have the same interface. Let $S_1, S_2, \ldots, S_n$ be servers with the same interface, then

$$\texttt{s}\ \textbf{is}\ [S_1,\ S_2,\ \cdots,\ S_n]$$

declares a server array containing n servers that are selected with the subscripts `s[0]` to `s[`$n-1$`]`. Server arrays can also be nested to create multidimensional arrays.

Hiding declarations

$$\textit{hiding-declaration} = \textbf{from \{}\ \{_1\ \texttt{:}\ \langle\text{declaration}\rangle\ \}\ \textbf{\}}\ \textbf{interface}\ \langle\text{name}\rangle$$

A *hiding declaration* encloses a set of declarations and specifies one name declared by these to be visible to the scope of the declaration. This is used to create minimal interfaces with collections of servers to perform abstraction.

Simultaneous declarations

$$\textit{simultaneous-declaration} = \{_0\ \textbf{\&}\ \langle\text{declaration}\rangle\ \}$$

Declarations separated by the symbol '&' occur *simultaneously* and have the same scope. They are therefore visible to one another. Simultaneous declarations are used to introduce mutually referential declarations, such as servers that communicate with each other.

Examples

Let P, Q, R and S be servers, and $X(y)$ be a server X in which y is free.

- If P, Q, R and S have the same interface then the declaration

  $$\texttt{n}\ \textbf{is}\ [\ P,\ Q,\ R,\ S\ ]$$

  declares a 1D array containing four servers and R, for example, is selected with the subscript `n[2]`. A 2D array with the same components is declared as

  $$\texttt{n}\ \textbf{is}\ [\ [P,\ Q],\ [R,\ S]\ ]$$

  and this time R is selected with the subscript `n[1][0]`.

- The hiding declaration

  $$\textbf{from}\ \{\ \texttt{m}\ \textbf{is}\ R\ :\ \texttt{n}\ \textbf{is}\ S(\texttt{m})\ \}\ \textbf{interface}\ \texttt{n}$$

  only introduces the name `n` to the scope of the declarations, but the name `m` is visible to the declaration of `n`.

- With a simultaneous declaration, the following holds:

  $$[\![\texttt{m}\ \textbf{is}\ R\ :\ \texttt{n}\ \textbf{is}\ S(\texttt{m})]\!] = [\![\texttt{n}\ \textbf{is}\ S(\texttt{m})\ \&\ \texttt{m}\ \textbf{is}\ R]\!]$$

- The simultaneous declarations

  $$\texttt{n}\ \textbf{is}\ S(\texttt{m})\ \&\ \texttt{m}\ \textbf{is}\ R(\texttt{n})$$

  introduce names for two servers that communicate with each other.

6.8.3. Calls

$$command \;=\; \langle\text{element}\rangle\ \texttt{(}\ \{_0\texttt{,}\ \langle\text{actual}\rangle\ \}\ \texttt{)}$$

A *server call* is a command that specifies the name of the server, name of the call and a list of actual parameters. The behaviour of a server call is defined in the following way. Let $f_1, f_2, \ldots, f_n$ be formals, $a_1, a_2, \ldots, a_n$ be actuals, and C be a command; then

```
s is interface (call c(f₁, f₂, ···, fₙ)):
 alt { accept c(f₁, f₂, ···, fₙ) C }:
s.c (a₁, a₂, ···, aₙ)
```

has the effect

```
f₁ is a₁ : f₂ is a₂ : ... : fₙ is aₙ : C
```

Examples

- The process

  ```
  s is interface(call read(var v), write(val v)):
    var x: alt { accept read(var v): v := x
               | accept write(val v): x := v }:
  ... s.write(59); ...; s.read(v); ...
  ```

  uses a server as a variable. At some point it assigns the value 59 to the variable held by the server, then later reads the value back.

- The processes

  ```
  s is interface(call inc(), dec()):
  { initial c := 0:
    alt { c < 100 & accept inc(): c := c+1
        | c > 0 & accept dec(): c := c-1 } }:
  { { ... s.inc(); ... } & { ... s.dec(); ... } }
  ```

  share a server that provides guarded calls to increment and decrement a count. This could be used, for example, if there is a producer-consumer relationship to ensure that the producer process which increments the count never produces more than 100 outstanding items.

6.8.4. Rules

12. *Server channels.* A server can use channels for communication and these are subject to the same rules as processes, except that connections can only be established with other servers with the same scope. Since channel inputs can only appear in the alternatives in an alternation declaration, output commands can only appear either in the command associated with an alternative or in the initialisation or finalisation commands.
13. *Server calls.* A server must provide call guards for each of the calls specified in its interface.
14. *Server interfaces.* A server interface can only contain channel end and call specifier components.
15. *Server disjointness.* Servers are subject to the same variable- and channel-disjointness rules as processes (see §6.7.3 [p. 82]). In particular, a server can change a variable or component of an array by assignment or input if no other server or process in its scope can read or change that variable, and, a server can read the value from a variable or component of an array only if no other server or process changes the variable.
16. *Named server arrays.* Components of a server array must all present the same interface.

6.9. Replication

$$
\begin{array}{rcl}
\textit{command} & = & \texttt{seq}\ \langle\text{replicator}\rangle\ \langle\text{command}\rangle \\
\textit{process} & = & \texttt{par}\ \langle\text{replicator}\rangle\ \langle\text{process}\rangle \\
\textit{conditional} & = & \texttt{if}\ \langle\text{replicator}\rangle\ \langle\text{choice}\rangle \\
\textit{alternation} & = & \texttt{alt}\ \langle\text{replicator}\rangle\ \langle\text{alternative}\rangle \\
\textit{server-declaration} & = & \langle\text{name}\rangle\ \texttt{is}\ \langle\text{replicator}\rangle\ \langle\text{server}\rangle \\
& | & \langle\text{name}\rangle\ \texttt{is [}\ \langle\text{expression}\rangle\ \texttt{]}\ \langle\text{server}\rangle \\
\textit{replicator} & = & \texttt{[}\ \{_1\ \texttt{,}\ \langle\text{index-range}\rangle\ \}\ \texttt{]} \\
\textit{index-range} & = & \langle\text{name}\rangle\ \texttt{=}\ \langle\text{expression}\rangle\ \texttt{for}\ \langle\text{expression}\rangle \\
& | & \langle\text{name}\rangle\ \texttt{=}\ \langle\text{expression}\rangle\ \texttt{for}\ \langle\text{expression}\rangle\ \texttt{step}\ \langle\text{expression}\rangle
\end{array}
$$

Replicators can be used to create a number of similar commands, components of a construct, processes or servers. A replicator consists of one or more *index ranges*. An index range declares the name of an *index* variable, a *base* expression, a *count* expression and, optionally, a *step* expression. The base, count and step define the range of values that an index variable takes. This name of the index variable is available to the replicated component and each replicated instance takes a unique value of the index.

6.9.1. Constructs

Let X be one of **if**, **alt**, **seq** or **par**; $Y(i)$ be a choice, alternative, command or process corresponding to X in which i is free; $\oplus$ be one of the separators '`|`', '`;`' or '`&`' corresponding to X; and b, c and s be integer values. Then, the behaviour of a *replicated conditional, alternative, sequence or parallel* with a single index range is defined by the following. If $c \geq 0$:

$$[\![X\ \texttt{[}i\texttt{=}b\ \texttt{for}\ c\ \texttt{step}\ s\texttt{]}\ Y(i)]\!] \ = \ [\![X\ \texttt{\{}\ Y(b)\ \oplus\ Y(b{+}s)\ \oplus\cdots\oplus\ Y(b{+}(c{-}1)s)\ \texttt{\}}]\!]$$

If $c = 0$:

$$[\![\texttt{if}\ \texttt{[}i\texttt{=}b\ \texttt{for}\ \texttt{0}\ \texttt{step}\ s\texttt{]}\ Y(i)]\!] \ = \ [\![\texttt{stop}]\!]$$

$$[\![\texttt{alt}\ \texttt{[}i\texttt{=}b\ \texttt{for}\ \texttt{0}\ \texttt{step}\ s\texttt{]}\ Y(i)]\!] \ = \ [\![\texttt{stop}]\!]$$

$$[\![\texttt{seq}\ \texttt{[}i\texttt{=}b\ \texttt{for}\ \texttt{0}\ \texttt{step}\ s\texttt{]}\ Y(i)]\!] \ = \ [\![\texttt{skip}]\!]$$

$$[\![\texttt{par}\ \texttt{[}i\texttt{=}b\ \texttt{for}\ \texttt{0}\ \texttt{step}\ s\texttt{]}\ Y(i)]\!] \ = \ [\![\texttt{skip}]\!]$$

If $c < 0$:

$$[\![X\ \texttt{[}i\texttt{=}b\ \texttt{for}\ c\ \texttt{step}\ s\texttt{]}\ Y(i)]\!] \ = \ [\![\texttt{stop}]\!]$$

For an index range without a step expression

$$[\![X\ \texttt{[}i\texttt{=}b\ \texttt{for}\ c\texttt{]}\ Y(i)]\!] \ = \ [\![X\ \texttt{[}i\texttt{=}b\ \texttt{for}\ c\ \texttt{step}\ \texttt{1]}\ Y(i)]\!]$$

Let I_k be a index range $i_k = b_k$ **for** c_k, where i_k is a name and b_k and c_k are integer values defined for the index range. Let b_i and c_i for $1 \leq i \leq d$ be integer values, then the behaviour of *replicators* with d index ranges is defined by

$$\begin{array}{l}[\![X\ \texttt{[}I_1, I_2, \cdots, I_d\texttt{]}\ Y(i_1, i_2, \cdots, i_d)]\!] \ = \\ \quad [\![X\ \texttt{[}I_1\texttt{]}\ \texttt{\{}\ X\ \texttt{[}I_2\texttt{]}\ \texttt{\{}\ \cdots\ \texttt{\{}\ X\ \texttt{[}I_d\texttt{]}\ Y(i_1, i_2, \cdots, i_d)\ \texttt{\}\}\}}]\!]\end{array}$$

Examples

- The conditional command with a nested replicated conditional command

```
if { if [i=0 for N] a[i] ~= b[i]: match := false
   | true: match := true }
```

  determines the equality of two arrays a and b of length N.

- The alternation command

```
alt [i=0 for N] c[i] ? v: out ! v
```

  merges data from N input channels onto a single channel end out.

- The replicated sequence with two index ranges

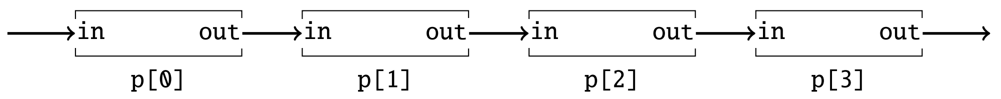
```
seq [i=0 for N, j=0 for N-i-1]
  if { a[j] > a[j+1]:
    { tmp := a[j];
      a[j] := a[j+1];
      a[j+1] := tmp } }
```

  implements the bubble sort algorithm over a length N array a.

- The replicated process

```
p is par [i=0 for 4] interface(chanend in, out):
{ if i=0 then
    connect in to s
  else
    connect in to p[i-1].out;
  if i=N-1 then
    connect out to d
  else
    connect out to p[i+1].in;
  var v: in ? v; out ! v+1 }
```

  creates a *pipeline* of processes, through which a value is passed and incremented by each process. This is illustrated by the diagram, where the subscript and channel end names are labelled for each component process:



6.9.2. Servers

Servers can also be replicated. Let $S(i)$ be a server specification in which i is free; n be a name; and b, c and s be integer values. Then, the behaviour of a *replicated server* with a single index range is defined by

$$[\![\texttt{[}i = b\ \mathbf{for}\ c\ \mathbf{step}\ s\texttt{]}\ S(i)]\!] \;=\; [\![\texttt{[}S(b)\texttt{,}\ S(b+s)\texttt{,}\ \cdots\texttt{,}\ S(b+(c-1)s)\texttt{]}]\!]$$

$$[\![\texttt{[}i = b\ \mathbf{for}\ c\texttt{]}\ S(i)]\!] \;=\; [\![\texttt{[}i = b\ \mathbf{for}\ c\ \mathbf{step}\ \texttt{1]}\ S(i)]\!]$$

Let I_k be a index range $i_k = b_k$ **for** c_k, where i_k is a name and b_k and c_k are integer values. Then, the behaviour of replicators with d index ranges is defined by

$$[\![\texttt{[}I_1\texttt{,}\ I_2\texttt{,}\ \cdots\texttt{,}\ I_d\texttt{]}\ N]\!] \;=\; [\![\texttt{n}\ \mathbf{is}\ \texttt{[}I_1\texttt{]}\ \texttt{[}\ \texttt{[}I_2\texttt{]}\ \texttt{[}\ \cdots\ \texttt{[}\ \texttt{[}I_d\texttt{]}\ N\ \texttt{]]]}]\!]$$

When replicated instances of a server do not require their unique indices, then a shorthand form of the replicator can be used. Let e be an expression, then

$$[\![\texttt{[}e\texttt{]}\ N]\!] \;=\; [\![\texttt{[}i = 0\ \textbf{for}\ e\texttt{]}\ N]\!]$$

Example

The process

```
n is [N] interface(call read(var v), write(val v)):
  var x: alt { accept read(var v): v := x
             | accept write(val v): x := v }:
... n[i].write(53); ... n[j].read(v); ...
```

uses an array of servers that act as variables that can be written to and read from. At some point the client process writes the value 53 to the `i`th server variable and then later on reads the value stored by the `j`th one.

6.9.3. Rules

17. *Replicator indices.* The value of any replicator index cannot be changed by an assignment or input.

6.10. Expressions and elements

An *expression* produces a value and is composed of operands and operators. They cannot cause any *side-effects* by changing any external state. An *operand* is either an element or literal or nested expression, which is enclosed by parentheses. There are no precedence rules, so nested expressions must be explicitly bracketed. An *operator* takes either one or two operands and produces a value. An *element* is either a name, subscripted name, field, or function call.

A full specification of expressions and elements is given in §A.3 [p. 263].

6.10.1. Valof expressions

$$\begin{aligned} \textit{valof} &= \textbf{valof} \ \langle\text{command}\rangle \ \textbf{result} \ \langle\text{expression}\rangle \\ &\mid \ \langle\text{specification}\rangle \ \textbf{:} \ \langle\text{valof}\rangle \\ \textit{expression} &= \textbf{(} \ \langle\text{valof}\rangle \ \textbf{)} \end{aligned}$$

A *valof* expression produces a value from a command. A valof expression **valof** C **result** e is a block and the scope of C extends to the **result** expression e. It is evaluated by performing the command C and then evaluating the expression e to produce the result.

Example

The valof expression

```
valof { var t, x, y: x := a; y := b;
        while y ~= 0 do
        { t := y; y := x rem y; x := t } }
result t
```

produces the greatest common divisor between two numbers a and b using Euclid's algorithm.

6.10.2. Rules

18. *Valof side-effects.* A valof expression cannot cause side-effects by changing any external state. It cannot therefore make calls using a free name or assign to a free variable.

6.11. Procedural abstraction

The details of a process, server or expression can be hidden in a component that presents an interface to allow the behaviour to be considered abstractly. These components are called *procedures* and *functions*.

6.11.1. Procedures

Definition

$$\begin{aligned}
\textit{definition} &= \langle\text{procedure}\rangle \\
\textit{procedure} &= \texttt{process}\ \langle\text{name}\rangle\ \texttt{(}\ \{_0\ \texttt{,}\ \langle\text{formal}\rangle\ \}\ \texttt{)}\ \texttt{is}\ \langle\text{process}\rangle \\
&\mid\ \texttt{server}\ \langle\text{name}\rangle\ \texttt{(}\ \{_0\ \texttt{,}\ \langle\text{formal}\rangle\ \}\ \texttt{)}\ \texttt{is}\ \langle\text{server}\rangle \\
\textit{formal} &= \langle\text{specifier}\rangle\ \{_1\ \texttt{,}\ \langle\text{name}\rangle\ \} \\
&\mid\ \texttt{val}\ \{_1\ \texttt{,}\ \langle\text{name}\rangle\ \}
\end{aligned}$$

A procedure specifies a name, a process or server and a set of *formal parameters.* Since there is no global scope in sire the set of formal parameters are the free elements of the process or server.

Let S be a specifier, then a formal parameter of the form $S\ n$ specifies a name n with type given by S. A single formal can also specify multiple names. Let S be a specifier and $N_1, N_2, \cdots, N_n$ be names, then

$$S\ \ N_1, N_2, \cdots, N_n\ =\ S\ \ N_1\texttt{,}\ \ S\ \ N_2\texttt{,}\ \ \cdots\texttt{,}\ \ S\ \ N_n$$

S may specify a variable or array variable of a primitive or channel end type, or a compound channel end or array of compound channel ends. For value specifications

$$\texttt{val}\ \ N_1, N_2, \cdots, N_n\ =\ \texttt{val}\ \ N_1\texttt{,}\ \ \texttt{val}\ \ N_2\texttt{,}\ \ \cdots\texttt{,}\ \ \texttt{val}\ \ N_n\texttt{.}$$

Let X be one of **process** or **server**, Y be a process or server corresponding to X and $f_1, f_2, \cdots, f_n$ be formal parameters. Then, the definition

$$X\ \ N\ \ \texttt{(}f_1, f_2, \cdots, f_n\texttt{)}\ \ \texttt{is}\ \ Y$$

defines the name N to be a procedure that behaves like Y. If f_i is an array type, it must specify the length of each dimension with an expression that consists of only of constant values or variables specified by other the other formals $f_1, f_2, \cdots, f_{i-1}, f_{i+1}, \cdots, f_n$.

Examples

- The procedure

  ```
  process Buffer(chanend cin, cout) is
    interface(chanend in, out):
    { connect in to cin; connect out to cout;
      while true do { in ? v; out ! v } }
  ```

  defines a buffer process with channel end parameters corresponding to the input and output processes it connects to.

- The procedure

  ```
  server Var() is
    interface(call read(var v), write(val v)):
        var x: alt { accept read(var v): v := x
                   | accept write(val v): x := v }:
  ```

  defines a server that behaves like a variable.

Channel end and server call parameters for procedures

$$\begin{aligned}
\textit{primitive-type} &= \langle\text{process-type}\rangle \\
&\mid \langle\text{server-type}\rangle \\
\textit{process-type} &= \textbf{process}\ \langle\text{name}\rangle \\
&\mid \textbf{process}\ \langle\text{interface}\rangle \\
\textit{server-type} &= \textbf{server}\ \langle\text{name}\rangle \\
&\mid \textbf{server}\ \langle\text{interface}\rangle
\end{aligned}$$

A procedure can specify parameters for channel ends and server calls. These may be specified directly or with a compound name from a process or server reference with the keywords **process** or **server** respectively, its interface and a name. The interface can either be specified *explicitly* by listing the components, or *implicitly* by giving the name of a procedure that defines a process or server with the interface.

Examples

- The procedure

  ```
  process Producer(process Buffer b) is
    interface (chanend c):
  { connect c to b.cin; ... c ! v; ... }
  ```

  has a procedure parameter of type `Buffer` (defined above) with the name `b`. This name is used to select components from the interface of `Buffer`.
- The procedure

  ```
  process P(server Var v) is
    ... v.write(101); ...
  ```

  has a server array parameter of the type `Var` (defined above) with the name `v`, allowing the process to read and write to each component server. Alternatively, it could be written by specifying the interface explicitly:

  ```
  process P(server interface(call read(var v), write(val v)) v) is
    ... v.write(101); ...
  ```

Passing procedures as parameters

$$\begin{aligned}
\textit{formal} &= \langle\text{call-type}\rangle\ \{_1\ \texttt{,}\ \langle\text{name}\rangle\ \texttt{(}\ \{_0\ \texttt{,}\ \langle\text{formal}\rangle\ \}\ \texttt{)}\ \} \\
\textit{abbreviation} &= \langle\text{call-type}\rangle\ \langle\text{name}\rangle\ \texttt{(}\ \{_0\ \texttt{,}\ \langle\text{formal}\rangle\ \}\ \texttt{)}\ \textbf{is}\ \langle\text{name}\rangle \\
\textit{call-type} &= \textbf{process} \\
&\ \textbf{function}
\end{aligned}$$

A procedure or server call can specify procedures as parameters. This is so that different implementations of a procedure with the same formals can be supplied to a procedure instance or server call. With a procedure parameter to a server call, the supplied procedure is executed remotely.

A formal parameter of the form

$$\textbf{process}\ N\ (f_1,\ f_2,\ \cdots,\ f_n)$$

specifies N as the name for a procedure parameter. A single procedure can specify multiple calls. Let F_k be a sequence of formals $f_1, f_2, \cdots, f_n$ where n is an integer defined for the sequence.

$$[\![\texttt{process}\ N_1\ \texttt{(}F_1\texttt{)},\ N_2,\ \texttt{(}F_2\texttt{)},\ \cdots,\ N_n\ \texttt{(}F_n\texttt{)}]\!] =$$
$$[\![\texttt{process}\ N_1\ \texttt{(}F_1\texttt{)},\ \texttt{process}\ N_2,\ \texttt{(}F_2\texttt{)},\ \cdots,\ \texttt{process}\ N_n\ \texttt{(}F_n\texttt{)}]\!]$$

Example

The procedure

```
process P(process sort(var[len] data, val len)) is
  ... sort(values, 997); ...
```

has a parameter specifying a procedure that implements a sorting algorithm.

6.11.2. Instances

$$\begin{aligned}
\textit{process} &= \langle\text{instance}\rangle \\
\textit{server} &= \langle\text{instance}\rangle \\
\textit{command} &= \langle\text{instance}\rangle \\
\textit{instance} &= \langle\text{name}\rangle\ \textbf{(}\ \{_0\ \textbf{,}\ \langle\text{actual}\rangle\ \}\ \textbf{)} \\
\textit{actual} &= \langle\text{element}\rangle \\
&\mid\ \langle\text{expression}\rangle
\end{aligned}$$

An *instance* of a procedure is created by specifying its name and a list of *actual parameters*. When an *instance* of the procedure is created, each formal serves as the left hand side of an abbreviation of an *actual parameter*, a, that is supplied to the instance. This provides a binding of each free variable to the scope in which it is instantiated. Each actual is an element or expression with a compatible type with the formal parameter.

The scoping rules mean that an instance of a process type can be substituted in-place for the process it names by inserting abbreviations of each formal with the actual parameter. This is defined in the following way. Let X be one of **process** or **server**, Y be a process or server corresponding to X, $f_1, f_2, \cdots, f_n$ be formals, $a_1, a_2, \cdots, a_n$ be actuals. Then, if X is a program in which no name is specified more than once and it contains the definition

$$X\ N\ \texttt{(}f_1, f_2, \cdots, f_n\texttt{)}\ \texttt{is}\ Y$$

then, within its scope

$$N\ \texttt{(}a_1, a_2, \ldots, a_n\texttt{)}\ =\ f_1\ \texttt{is}\ a_1\ \texttt{:}\ f_2\ \texttt{is}\ a_2\ \texttt{:}\ \ldots\ \texttt{:}\ f_n\ \texttt{is}\ a_n\ \texttt{:}\ P$$

provided each abbreviation is valid. This allows procedures to be compiled either as a closed subroutine or by substituting the body of the definition directly with the instance.

Examples

- The parallel processes

  ```
  b1 is Buffer(in, b.in) & b2 is Buffer(b1.out, out)
  ```

  are named instance of the buffer procedure that was defined in the previous section. The procedure `Buffer` can be substituted directly into this to obtain

  ```
  chanend out is b2.in:
  p is interface(chanend in, out):
  { connect in to s; connect out to q.in;
    while true do { in ? v; out ! v } } &
  chanend in is b1.out:
  ```

```
q is interface(chanend in, out):
{ connect in to p.out; connect out to d;
  while true do { in ? v; out ! v } }
```

where channel end abbreviations are inserted where the name of the actual differs from that of the formal.

- In the following, instances of `Producer` and `Consumer` processes are provided with calls from a server to insert and remove items from a shared data structure:

```
process Producer(call insert(val x)) is { ... insert(x); ... }:
process Consumer(call remove(var x)) is { ... remove(x); ... }:
s is interface(call insert(var v), remove(val v)):
  ... :
{ Producer(s.insert) & Consumer(s.remove) }
```

The definitions of the processes can be substituted in this to obtain the equivalent process

```
s is interface(call insert(val x), remove(var x)):
  ... :
{ call insert is s.insert: { ... insert(x); ... }
& call remove is s.remove: { ... remove(x); ... } }
```

where abbreviations are added for the call parameters where the name of the actual differs from that of the formal.

6.11.3. Hiding definitions

$$\textit{definition} = \textbf{server}\ \langle\text{name}\rangle\ \textbf{(}\ \{_0\textbf{,}\ \langle\text{formal}\rangle\ \}\ \textbf{)}\ \textbf{inherits}\ \langle\text{hiding-declaration}\rangle$$

A different way to define a server is to inherit an interface from a server in a set of declarations. This can be used to combine a number of servers into a single reusable module, exposing a single server declaration as an interface.

Let $i_1, i_2, \cdots, i_m$ be specifiers for components of an interface, then the definition

```
server N(f1, f2, ···, fn) inherits
 from
  ⋮
  s is interface (i1, i2, ···, im) to D
  ⋮
 interface s
```

defines N as the name of a server type who inherits the interface from the server with name `s` in the hiding declaration. Let X be a program in form where no name is specified more than once, then if X contains the above definition, in the scope of N

$$\begin{aligned}&[\![\texttt{x}\ \textbf{is}\ N(a_1, a_2, \cdots, a_n)\ \texttt{:}\ S]\!]\ =\\ &\quad[\![\ f_1\ \textbf{is}\ a_1\ \texttt{:}\ f_2\ \textbf{is}\ a_2\ \texttt{:}\ \ldots\ \texttt{:}\ f_n\ \textbf{is}\ a_n\ \texttt{:}\\ &\quad\ \texttt{s}\ \textbf{is interface}(i_1, i_2, \cdots, i_n)\texttt{:}\ D\ \texttt{:}\ \cdots\ \texttt{:}\ S]\!]\end{aligned}$$

where S is the scope of the declaration.

Example

The server definition

```
server Array() inherits
  from
    m is [N] Var():
    n is interface(call read(val i, var v), write(val i, val v)):
      alt { accept read(val i, var v): m[i].read(v)
          | accept write(val i, val v): m[i].write(v) }:
  interface n
```

defines a collection of servers consisting of an array of `N Var` servers that provide calls to read and write a value and a single server that provides access. An instance of the server `Array`

```
x is Array()
```

can be substituted for the definition to obtain

```
m is [N] Var():
n is interface(call read(val i, var v), write(val i, val v)):
  alt { accept read(val i, var v): m[i].read(v)
      | accept write(val i, val v): m[i].write(v) }:
server x is n
```

6.11.4. Functions

Definitions and instances

$$\begin{aligned} \textit{definition} &= \mathbf{function}\ \langle\text{name}\rangle\ \mathtt{(}\ \{_0\mathtt{,}\ \langle\text{formal}\rangle\ \}\ \mathtt{)}\ \mathbf{is}\ \langle\text{valof}\rangle \\ \textit{expression} &= \langle\text{instance}\rangle \end{aligned}$$

A valof expression can be reused by defining a *function*. Each formal parameter of a function must be a value abbreviation and functions cannot contain calls to processes. The definition

$$\mathbf{function}\ N\ \ (f_1, f_2, \cdots, f_n)\ \mathbf{is\ valof}\ C\ \mathbf{result}\ e$$

defines N as the name of a function with the valof expression **valof** C **result** e.

Let X be a program in a form where no name is specified more than once, then if X contains the above function definition, in the scope of N

$$\begin{aligned} &[\![N\ \ (a_0, a_1, \cdots, a_n)]\!]\ = \\ &\quad [\![(\mathbf{valof}\ f_0\ \mathbf{is}\ a_0\ :\ f_1\ \mathbf{is}\ a_1\ :\ \cdots\ :\ f_n\ \mathbf{is}\ a_n\ :\ C\ \mathbf{result}\ e)]\!] \end{aligned}$$

provided each abbreviation is valid. This allows functions to be compiled either as a closed subroutine or by substitution.

Functions as parameters

$$\textit{call-type} = \mathbf{function}$$

Functions can be passed as parameters in the same way as procedures.

Examples

- The function

```
function gcd(val a, val b) is
  valof { var t, x, y: x := a; y := b;
        while y ~= 0 do
        { t := y; y := x rem y; x := t } }
  result t
```

defines a valof expression to produce the greatest common divisor between two numbers `a` and `b`.

- The assignment

  ```
  divisor := gcd(243, 346)
  ```

  assigns the value produced of the function `gcd` to the variable `divisor`, which is 1. The function can be substituted directly to obtain

  ```
  divisor := (val a is 243: val b is 346:
      valof { var t, x, y: x := a; y := b;
              while y ~= 0 do
              { t := y; y := x rem y; x := t } }
      result t)
  ```

6.11.5. Rules

19. *Parameters.* The rules for procedures, hiding definitions and functions follow those for abbreviations (see §6.6.6 [p. 79]). The following two rules further define compatibility for interface, call and procedure types.
20. *Matching interface parameters.* A compound name that references a single process or server can only be supplied as an actual parameter if it is compatible. This requires the interface to be the same as the one specified by the formal parameter.
21. *Matching call and procedure parameters.* A call or procedure name can only be supplied as an actual parameter if it is compatible. This requires (1) the type of the formal parameter are the same and (2) each of the types of the formals in the definition of the call or procedure are the same as the formals specified in the parameter (which are specified in braces).
22. *Array parameters.* Each array parameter of a process or server type must specify its length by an expression that can contain only constant values or the names of other value parameters.
23. *Recursion.* Recursive procedures or functions are not permitted.

6.12. Program

$$
\begin{aligned}
\textit{program} &= \langle\text{program-specification}\rangle\ \mathbf{:}\ \langle\text{program}\rangle \\
&\mid\ \langle\text{sequence}\rangle \\
\textit{program-specification} &= \langle\text{specification}\rangle \\
&\mid\ \langle\text{definition}\rangle \\
\textit{definition} &= \langle\text{simultaneous-definition}\rangle \\
\textit{simultaneous-definition} &= \{_0\ \mathbf{\&}\ \langle\text{definition}\rangle\ \}
\end{aligned}
$$

A *program* is a single command sequence, or process, with a specification that can contain definitions of process types, server types and functions.

Simultaneous definitions

Definitions separated by the symbol **&** occur *simultaneously* and share the same scope; they are therefore visible to one another. Simultaneous definitions are used to introduce mutually referential definitions, such as processes or servers that communicate with each other.

Example

The program

```
val N is 4:
process Node(val i, process Source s, Sink d, Node[N] p) is ... &
process Source(process Node b) is ... &
process Sink(process Node e) is ... &
{ s is Source(p) & d is Sink(p) &
  p is par [i=0 for N] Node(i, s, t) }
```

consists of a collection of processes organised in a pipeline. The procedures `Node`, `Source` and `Sink` are introduced simultaneously since they communicate with each other. This is illustrated by the diagram

$$\begin{tikzcd}
\boxed{\texttt{Source}} \arrow[r] & \boxed{\texttt{Node}} \arrow[r] & \boxed{\texttt{Node}} \arrow[r] & \boxed{\texttt{Node}} \arrow[r] & \boxed{\texttt{Node}} \arrow[r] & \boxed{\texttt{Sink}} \\[-2em]
\texttt{s} & \texttt{p[0]} & \texttt{p[1]} & \texttt{p[2]} & \texttt{p[3]} & \texttt{d}
\end{tikzcd}$$

6.12.1. Rules

24. *No global scope.* A program specification can only contain definitions and abbreviations.

6.13. Discussion

The design of sire draws inspiration from a number of different areas and the following sections explain these relationships in more detail. The differences are summarised in the final section.

6.13.1. Occam

Sire is based on a subset of occam that includes elements of the syntax and their semantics. Those components of this subset, which were present in the original version of the language [INM84], are:

- the *primitive processes* assignment, input, output, skip and stop;
- the *constructs* sequence, parallel, conditional, alternation and loop;
- the *replicated* forms of sequence, parallel, conditional and alternation;
- the disjointness rules for parallel processes;
- declarations, abbreviations and the scoping rules;
- *procedures* and their substitution semantics.

Additionally, several components from occam 2 [INM88b] are used. These do not introduce any new concepts into the language, but they provide more convenient and expressive notations. These are:

- valof processes and functions;
- multidimensional arrays.

Another version of occam known as occam 3 [Bar92] was proposed as a successor to occam 2 but it was never implemented. It was influential in the design of occam 2.1 [SGS95] but several of its main features were not included. These features provided ways to structure programs and develop abstractions, and they have provided inspiration for some of the key concepts in sire. In particular:

- *remote call channels* provide procedure-call semantics for channel communication between processes and a mechanism for many-to-one communication patterns;
- *resource* and *server declarations* provide a way to formulate the behaviour of a program component that is used as a subroutine into a declaration consisting of an alternative construct;
- *interfaces* provide a way of hiding declarations from a scope (although they are different from the interface concept in sire);
- *modules* allow a sequence of declarations to be encapsulated with a single name;
- *libraries* provide a way of encapsulating definitions and data types with a single name and the ability to make a subset of these visible.

The core aspects of the server proposal in sire are based on the combination of remote call channels, server declarations, interfaces and modules. The proposal makes syntactic and behavioural changes to these, but differs principally by the way in which the concept of a server is integrated into the language. In sire a server is a primitive mechanism that provides the sole means of dealing with communication abstractions.

A final important point is that a key concern in the design of the sire language is that it can be implemented efficiently on a highly-parallel distributed-memory system. This is demonstrated by the description of the compilation process in Chapter 8 and the empirical evaluation in Chapter 10. Since occam 3 was never implemented, it is not known whether that proposal, as it stands, also has this capability.

6.13.2. Communicating sequential processes

The original proposal for occam [May83] was based on early work with CSP [Hoa78] as a pragmatic embodiment of its principles in a practical programming language. Some elements of CSP were not included, such as recursion and guarded outputs, due to the complexity arising from their distributed implementation; others were changed and this experience influenced later versions of CSP.

Process naming

Processes in the original CSP were named and they communicated by specifying the name in an input or output. This makes it difficult to create abstractions since names cannot be hidden. Occam developed the concept of named communication channels that provide a single logical address that can be inherited, thereby removing the need for naming processes. The effect of this is that programs can be expressed as hierarchical collections of processes where the behaviour of component sub-processes can be hidden behind a minimal interface consisting solely of sets of channels. Named channels was incorporated into the later 'book' version of CSP (see [Hoa85, Ch. 7] for a discussion), however, they pose the following problems both for the programmer and for a distributed implementation.

- For the programmer, expressing process structures typically involves using arrays of named channels and selecting components with subscripts. Even for relatively simple structures such as trees and hypercubes, the subscripts can become complicated and bear little relation to the structure they represent.
- The compiler is responsible for generating code that can establish connections and, with named channels, it must do this by resolving the locations of each channel end point. For fixed process structures, this can be done by statically determining the program mapping and generating the corresponding connections in the executable program. However, it is difficult to produce an executable version of the program with this approach that does not include binary images for every core; for large numbers of processors, the generated binaries can grow very large.

For these reasons, sire does not use named communication channels, and instead uses a mechanism similar to the original version of CSP with named processes. The problems associated with naming and abstraction are instead solved by using servers.

Subordination

CSP includes the concept of *subordination* where the actions of a process are determined completely by another process [Hoa78], [Hoa85, Ch. 4].[3] This is analogous to the concept of a subroutine and is intended to be used as a means of structuring and representing data. Communication with a subordinate process follows procedure call semantics. The *user* process outputs one or more arguments and receives one or more results. The behaviour of the subordinate is to repetitively wait for input on one more channels and service each one according to the call. It terminates when the user does.

This was clearly the inspiration for remote call channels, servers and hiding declarations in the proposal for occam 3. It is also underpins the server proposal in sire.

6.13.3. Other influences

There are several other key influences in the design of sire.

- *Remote procedure calls.* The concept of remote procedure calls stems from programming networks of computers with distributed storage and is now widely employed in distributed

[3]The concept of shared resources and abstraction in parallel programming languages was introduced in Hoare's earlier work in 1971 [Hoa71].

systems [BN84] and programming languages. The first, *Distributed Processes* [Han78], provided inspiration for later languages such as Ada [Ame83] that combined remote procedure calls with CSP-style alternative selection.

- *Object orientation.* Servers correspond directly to the concepts of a *class* and *object* in the paradigm of *object-orientated programming*, where a server definition relates to a class, a server instance relates to an object, and the set of calls relate to public methods that operate on state hidden in the object.
- *Simultaneous declarations.* BCPL [Ric67, RWS79] was a strong influence on the original design for occam and its ideas remain as relevant as ever. In particular, its notation for introducing declarations simultaneously to express of mutually recursive procedures was the basis for simultaneous definitions and declarations for mutually referential collections of processes.

6.13.4. The differentiating aspects of sire

The following points enumerate the main differentiating aspects of sire compared to other approaches.

- *No named communication channels.* There are no named communication channels and channels are only established between named components of a parallel command. This precludes channels from making hierarchical connections between parent processes and nested children.
- *Process interfaces.* A process specifies an *interface* consisting of a number of channel ends. These are the only means of communicating with other processes. The name of a process is a prefix for the compound names for each component of the interface; these are selected as fields and can be passed as parameters.
- *Connect commands.* A channel is connected using a *connect* command. It specifies a *local* channel end from a process' own interface and the target *remote* channel end. Remote channel ends are selected from the name of the target process.
- *Instances of processes.* A subtle difference in terminology is used to refer to the use of procedure calls: they are *instanced* rather than *called*. This is due to the natural substitution semantics of procedure as a named process or server. However, *call* terminology is still used to refer to interaction with servers since this is the intuition for the operation of a server call, even though the semantics are described with substitution.
- *Servers.* A server is a type of process that is active only in response and only for the duration of the processes in its scope. In particular, servers can specify channel ends and calls in their interface and are the only means for dealing with sharing and abstraction communication between processes.
- *Hiding declarations.* A *hiding declaration* is used to make one of a collection of server declarations visible to a scope, hiding all of the others. An *inheriting server definition* has the same effect, but defines a new name that can be declared independently.
- *Simultaneous definitions and declarations* are used to introduce mutually referential entities.
- *Multiple index ranges.* Replicators can specify multiple index ranges. These are used to introduce multidimensional arrays of processes and servers.
- *No global scope.* There are no global variables in the program scope since execution is distributed. However, a server can be used to provide global state.

CHAPTER 7.

SIRE PROGRAMMING STRUCTURES

This chapter demonstrates how the primitive features of sire can be used to express simple and reusable *programming structures* for many types of programs. These structures are divided into *process structures*, which are regular message-passing topologies and *server structures*, which are the different ways in which servers can be used and combined. The last section provides some general discussion.

7.1. Process structures

Simple regular process structures underpin a huge variety of parallel algorithms and it is important that these can be expressed concisely. The examples in this section are based on pipeline, grid, tree and hypercube process structures (see §4.1.4 [p. 44]) and each one is explained in terms of a simple parallel algorithm, whose operation is largely determined by the pattern of communication in the process structure itself. Particular attention is paid to the way in which channel connections are established to form the structures, since sire is unconventional in this respect.

To simplify the presentation of each example, only problems sizes of $N = 2^i$ for $i > 0$ are considered with each process storing only one element of the input; the *granularity* can be increased by refactoring to store additional elements in each process.

7.1.1. Pipeline: the Sieve of Eratosthenes

Perhaps the most simple process structure is a *pipeline* (or 1D array). Typically, this structure operates in a systolic manner where data are streamed in through the component processes, each of which iteratively computes and communicates.

A simple parallel algorithm based on a pipeline is a parallel version of the *Sieve of Eratosthenes* that generates prime numbers. The following example is based on one by Hoare in [Hoa78]. In this, one process generates a stream of numbers, which are directed through a pipeline of processes where each one filters out numbers that are a multiple of a particular prime. A final process consumes the prime numbers output from the pipeline. The input sequence starts at 2, then 3 and subsequent odd numbers. The first number that each process receives is the prime that it uses to filter.

Component processes

Process 7.1 shows the definition of the `Node` process. It specifies an interface containing two channel ends, one for input and the other for output, and references to the `Master` process and the array of `Node` processes (of which it is itself a part). The input channel end is connected to the preceding pipeline node, except for the first `Node` where it is connected to the output channel end of the master. This case is identified with the parameter `i`, corresponding to the `Node`'s position. The output channel end is connected in a similar way, with the last node connected to the master. The remaining `Node`s make their connections in parallel so that `Node` i does not need to wait for `Node` $i-1$ to connect channel `in` and the total time to establish connections is not related to the size of the pipeline. When the channel connections have been established, each `Node` waits to receive a prime number and then

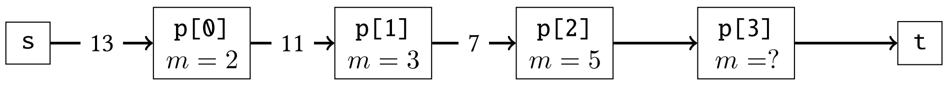


Figure 7.1.: A pipeline process structure that implements the Sieve of Eratosthenes. It consists of `Source` and `Sink` processes, and a replicated `Node` process, which are listed in Processes 7.2, 7.3 and 7.1 respectively. The labels show the values communicated and their direction of transmission at a particular point of execution.

repetitively receives a value and forwards it only if it is not divisible by the prime.[1]

The process `Source` in Process 7.2 generates a sequence of numbers. The process `Sink` in Process 7.3, with a single channel end, consumes the primes output from the pipeline and stores them in an array.

The program

The following is the complete process structure with 4 nodes to generate primes up to 16.[2]

```
val N is 16:
val SQRTN is 4:
process Node(val i, process Node[SQRTN] p, Source s, Sink t) is ... &
process Source(chanend in) is ... &
process Sink(chanend out) is ... :
{ s is Source(p) & t is Sink(p) &
  p is par [i=0 for SQRTN] Node(i, p, s, t) }
```

Since the `Node`, `Source` and `Sink` processes are mutually referential, their definitions are introduced simultaneously. In the parallel command, the `Source` and `Sink` are composed in parallel with an array of `Node`s. All three processes are labelled and their names are exchanged. The output of the pipeline will be 2, 3, 5, 7, 11 and 13. An illustration of this is given in Figure 7.1, where the source has just generated the number 13, the first three `Node`s have primes and the number 7 is about to pass through `Node` 3 and initialise `Node` 4.

Finally, it should be noted that this process structure will not terminate after the last input has been generated since the loop in each node has no termination condition. The `Node` and `Sink` processes could be easily modified to accept a termination token to halt the loop.

[1] The channel connections have been specified in such a way that there are no cycles. If this was not the case then the system would deadlock. Take for example the following process that creates a *ring* of processes.

```
p is par [i=1 for N] interface(chanend i, o):
{ connect i to p[i-1].o;
  connect o to p[i+1].i }
```

Since each process first connects its input channel end `i` to the output channel end `o` of the preceding process, no connection will complete and the ring will deadlock. One way to solve this is to execute the connections in parallel.

```
p is [i=1 for N] interface(chanend i, o):
{ connect i to p[i-1].o &
  connect o to p[i+1].i }
```

Similar care must be taken to ensure that channel communications do not cause cyclic dependencies and deadlock.

[2] To generate primes up to a number N, it is sufficient only to filter up to $\sqrt{N}$ since any factor greater than $\sqrt{N}$ has a matching factor less than $\sqrt{N}$.

```
process Node(val i, process Node[SQRTN] p, Source s, Sink t) is
  interface(chanend in, out):
{ if
  { i = 0:
    { connect in to s.out;
      connect out to p[i+1].in }
  | i = N-1:
    { connect in to p[i-1].out;
      connect out to t.in }
  | i > 0 and i < N-1:
    { connect out to p[i+1].in &
      connect in to p[i-1].out } };
  var p, m, mp:
  % receive a prime
  in ? p;
  % filter subsequent numbers
  mp := p;
  while true do
  { in ? m;
    while m > mp do mp := mp + p;
    if m < mp then out ! m else skip } }
```

Process 7.1: A pipeline node. This inputs a prime number from the previous node, then filters any subsequent numbers that have the prime as a factor.

```
process Source(chanend in) is
  interface(chanend c):
{ connect c to in;
  % generate the first prime, 2
  c ! 2;
  % then a stream of the next N odd numbers
  seq [i=3 for N step 2] c ! i }
```

Process 7.2: The pipeline source node, which generates a stream of values for the pipeline.

```
process Sink(chanend out) is
  interface(chanend c):
{ connect c to out;
  var i:
  var[N] primes:
  i := 0
  % receive and store each prime
  while true do
  { c ? primes[i];
    i := i + 1 } }
```

Process 7.3: The pipeline sink node, which receives the filtered prime values and stores them in an array.

7.1.2. Grid: systolic matrix multiplication

In a *grid* (or 2D array) each process is connected to four others. Matrix multiplication is a natural fit for execution in parallel on a grid and a systolic version, where one matrix is distributed on the grid and the other is streamed over the grid, has a particularly simple implementation.[3] The example here is based on the process structure given for the same problem by Hoare in [Hoa78], which computes the product $B \times A$ of two square matrices A and B.

Component processes

Process 7.4 gives the definition of a `Node` process. This has parameters for the column and row coordinates in the grid, references to the 2D grid array (of which it is a part) and references to each of the 1D border arrays. The initial connection phase is split into two parts, an east-west direction and a north-south direction. These operate in the same way as the primes pipeline.

When the connections have been established, each `Node` repetitively receives elements of the matrix B from its west neighbours and partial sums from the northern ones. It then forwards the elements of B to the east, computes the product of its element of the matrix A (stored in the variable `Aij`) with the element of B, adds this to the partial product and sends it south. A is initialised to the identity matrix.

The grid border processes `North`, `West`, `South` and `East` are given in Process 7.5. Each of these takes an index parameter and reference to the grid and establishes a single connection to a corresponding grid `Node`. The `North` sources input the value 0 (an initial partial sum); the j^{th} `West` source inputs elements from the j^{th} column of B (generating the identity matrix); the `East` sink processes discard the output; and the `South` sink processes receive elements of the output such that result $B \times A$ is distributed over the `South` processes, each holding a column of the matrix.

The program

The following is a complete process structure that implements parallel matrix multiplication for two 2×2 matrices.

```
val N is 3:
process North(val j, process Node[N][N] g) is ... &
process West(val j, process Node[N][N] g) is ... &
process South(val j, process Node[N][N] g) is ... &
process East(val j, process Node[N][N] g) is ... &
process Node(val i, val j, Node[N][N] g,
    process North[N] bn, process West[N] bw,
    process South[N] bs, process East[N] be) is ... :
{ n is par [i=0 for N] North(i, g) &
  w is par [j=0 for N] West(j, g) &
  s is par [i=0 for N] South(i, g) &
  e is par [j=0 for N] East(j, g) &
  g is par [i=0 for N, j=0 for N] Node(i, j, n, s, e, w) }
```

It consists of four 2-component arrays for each border and an 2×2 array of `Node`s. The process definitions are again introduced simultaneously since they are mutually referential as they were in the primes pipeline. An illustration of this process structure is given in Figure 7.2, where the communication links are annotated with the values being communicated at a point during the execution of the algorithm.

[3] Matrix multiplication can also be implemented in a non-systolic manner, for example with Cannon's algorithm [Can69], or using a pipeline process structure as is described by Brinch Hansen in [Han95b, Ch. 7].

Since the border processes make only a single connection to components of the grid, there are no cycles in the connection pattern and therefore it is deadlock free. Like the primes pipeline, the `Node`, `North` and `East` processes do not terminate; termination could be implemented with the `West` processes issuing a special token when they have completed inputting their column.

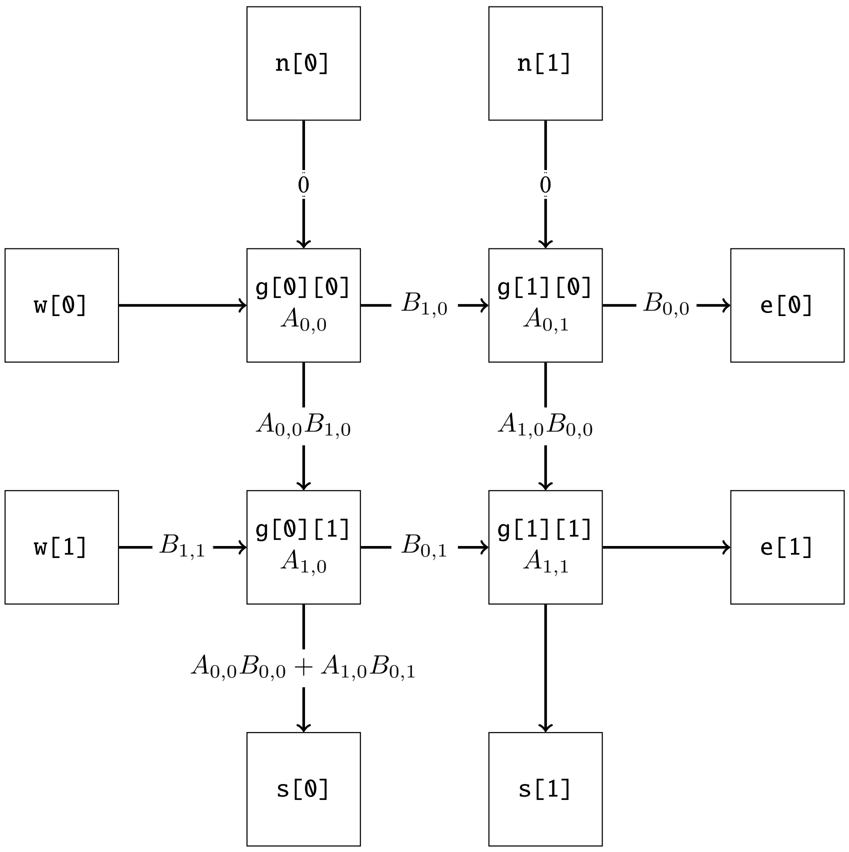


Figure 7.2.: A 2×2 grid process structure that implements matrix multiplication. It consists of a replicated `Node` process (Process 7.4). Four border processes `North`, `West`, `South` and `East` (Process 7.5), source and sink data from the grid. Links are labelled with the communicated values at a particular point of execution to illustrate their combined behaviour.

```
process Node(val i, val j, process Node[N][N] g,
    process North[N] bn, process West[N] bw,
    process South[N] bs, process East[N] be) is
  interface(chanend n, w, s, e):
{ if
  { i = 0:
    { connect w to bw[j];
      connect e to g[i+1][j] }
  | i = N-1:
    { connect w to g[i-1][j];
      connect e to be[j] }
  | i > 0 and i < N-1:
    { connect w to g[i-1][j] &
      connect e to g[i+1][j] } };
  if
  { j = 0:
    { connect n to bn[i];
      connect s to g[i][j+1] }
  | j = N-1:
    { connect n to g[i][j-1];
      connect s to bs[i] }
  | j > 0 and j < N-1:
    { connect n to g[i][j-1] &
      connect s to g[i][j+1] } };
  var Aij, Bi, j;
  % initialise the matrix element
  if i = j then Aij := 1 else Aij := 0;
  % communicate and compute
  while true do
  { n ? j;
    w ? Bi;
    e ! Bi;
    s ! Aij*Bi + j } }
```

Process 7.4: The replicated component of the grid process structure. This stores one element of the matrix A in the variable `Aij`. It repetitively receives elements a row of the matrix B from its west neighbour (`Bi`) and partial sums from the north. It then forwards the element of B eastwards, multiplies its element of A with the one from B, and sends this, added to the partial sum, south.

```
process North(val i, process Node[N][N] g) is
  interface(chanend c):
{ connect c to g[i][0];
  % input partial sums
  while true do c ! 0 }

process West(val j, process Node[N][N] g) is
  interface(chanend c):
{ connect c to g[0][j];
  % input elements from a column of B (the identity)
  seq [i=0 for N]
    if i = j then c ! 1 else c ! 0 }

process South(val i, process Node[N][N] g) is
  interface(chanend c):
{ connect c to g[i][N-1];
  % receive elements from the columns of the result
  var Bj[N]:
  for[k=0 for N] c ? B[k] }

process East(val j, process Node[N][N] g) is
  interface(chanend c):
{ connect c to g[N-1][j];
  % discard output
  var v:
  while true do c ? v }
```

Process 7.5: The border nodes to source and sink data to and from the grid of `Nodes` that perform the matrix multiplication.

7.1.3. Tree: prefix sum

A binary tree is a natural structure on which to implement a parallel *prefix sum*, a building block for many parallel algorithms. This operation takes a sequence of numbers $x_1, x_2, \cdots, x_n$ and produces a new sequence $y_1, y_2, \cdots, y_n$ where $y_1 = x_1$ and $y_i = x_i + y_{i-1}$ for $0 < i \leq n$. The method in the following example is based on the formulation described by Blelloch in [Ble90].

Component processes

The `Branch` process in Process 7.7 establishes channel connections in a similar way to the pipeline and grid examples. However, the channel connections between other `Branch` processes are embedded into a 1D array and are selected using more complex subscripts. Connections to the root and leaf arrays are performed based on the branch index. The `Root` process in Process 7.6 is connected to the first branch. Instances of the `Leaf` process in Process 7.8 are connected to each of the final level of branch nodes.

A complete binary tree with $N = 2^i$, for $i > 0$, leaf nodes has $N/2$ branch nodes connecting these and $N/2 - 1$ other branch nodes. A `Branch` node with array index i is connected to a root at $\lfloor(i+1)/2\rfloor + (i+1) \mod 2$ and children at $2i+1$ and $2i+2$ (these are the connection subscripts in Process 7.7. If the index is greater than $N/2 - 1$ then its children are `Leaf` nodes. The `Branch` process connects in sequence to the root node and children, then receives a value from the root node and forwards it to its children.

Initially, each `Leaf` has a value (its `id`) and the operation will compute the prefix sums of the values. The algorithm works by propagating partial sums to the `Leaf` nodes in a depth-first traversal of the tree. The result is stored at each of the `Leaf` nodes.

The program

The following is a complete process structure for a depth-3 binary tree implementing a prefix-sum operation.

```
val D is 3:
val N is 1 << D:
process Root(chanend root) is ... &
process Branch(val i, chanend m,
    process Branch[N-1] b, process Leaf[N] l) is ... &
process Leaf(val i, process Branch[N-1] b) is ... :
{ r is Root(b[0].root) &
  b is par [i=0 for N-1] Branch(i, m.root, b, l) &
  l is par [i=0 for N] Leaf(i, b) }
```

It consists of an array of `Branch` processes, an array of `Leaf` processes and a `Root` process composed in parallel. An illustration of this process structure is given in Figure 7.3. It shows the behaviour and state of the algorithm as it enters the right hand side of the tree.

```
process Root(chanend root) is
  interface(chanend c):
{ connect c to root;
  var v:
  root ! 0;
  root ? v }
```

Process 7.6: The tree root node. This inputs the initial partial sum, 0, then receives a value as the propagation passes from the left hand side of the tree to the right.

```
process Branch(val i, chanend m,
    process Branch[N-1] b, Leaf[N] l) is
  interface(chanend root, left, right):
{ if i = 0
  then connect root to m
  else if i rem 2 = 1
  then connect root to b[((i+1)/2)+((i+1) rem 2)].left
  else connect root to b[((i+1)/2)+((i+1) rem 2)].right;
  if i < N/2 then
  { connect left to b[(2*i)+1].root;
    connect right to b[(2*i)+2].root }
  else
  { connect left to l[i-(N/2)-1].root;
    connect right to l[i-(N/2)-1].root }
  var sum, lsum, rsum:
  root ? sum;                 % receive the partial sum
  left ! sum; left ? lsum;    % propagate sum to left-hand child
  right ! lsum; right ? rsum; % propagate sum to right-hand child
  root ! rsum                 % return the new sum
}
```

Process 7.7: The tree branch node. This propagates a partial sum first to the left hand child, then to the right hand one, and finally back to the root.

```
process Leaf(val i, process Branch[N-1] b) is
  interface(chanend root):
{ if i rem 2 = 1
  then connect root to b[N-(i/2)].left
  else connect root to b[N-(i/2)].right;
  var v, r, psum:
  v := i;
  root ? psum;   % receive the partial sum
  r := psum + v; % compute the local result
  root ! r       % return the updated partial sum
}
```

Process 7.8: The tree leaf node. This stores a value from the input sequence (in `v`), it then receives a partial sum to compute its own result, which is then returned for the remaining right-hand `Leaf` node.

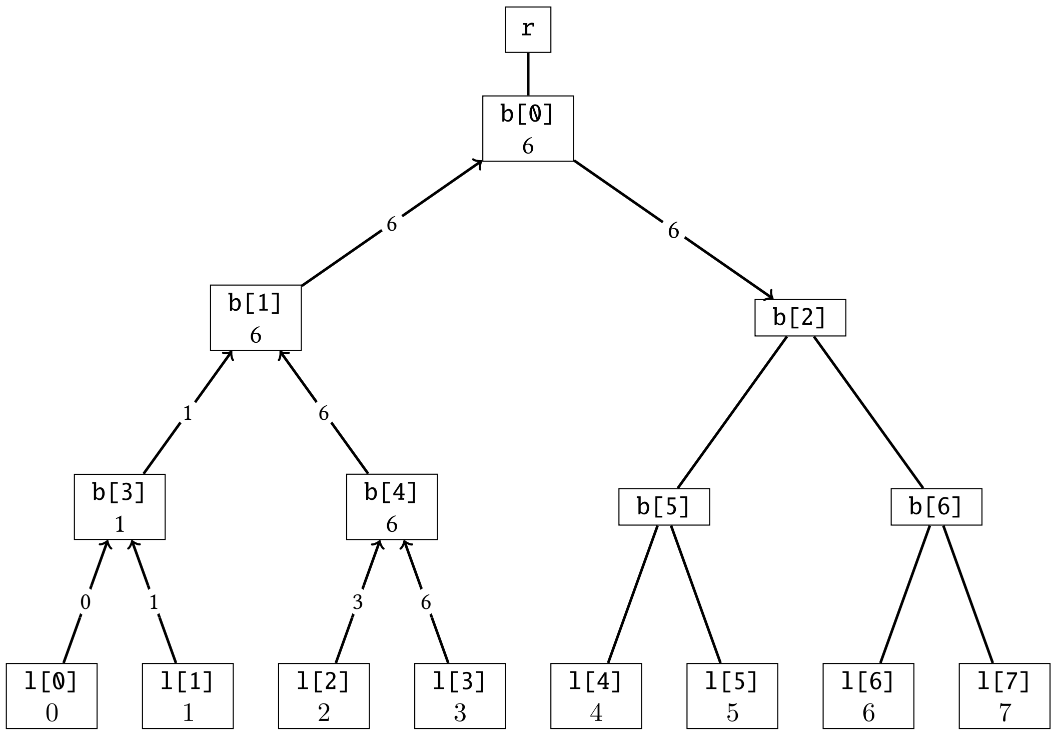


Figure 7.3.: A depth-3 binary tree process structure, consisting of `Root`, `Branch` and `Leaf` processes (Processes 7.6, 7.7 and 7.8 respectively). It implements a prefix-sum operation on the values stored at each `Leaf` and operates in a depth-first manner, propagating partial sums from the left hand side to the right. The labelled communication links show the child-parent communications as the partial sums enter the right hand side of the tree. The `Leaf` processes are labelled with the values of their variable `v` and `Branch` processes are labelled with the value of their `sum` variable.

7.1.4. Hypercube: sorting

Hypercubes have a powerful structure that allows all-to-all operations to be completed in n pairwise exchanges, where the number of nodes $N = 2^n$. This lends them as a structure for a variety of parallel algorithms, see Ranka and Sahni [RS90], and Foster [Fos95, Ch. 11] for examples.

The example here is a sorting algorithm based on a similar one described by Brinch Hansen in [Han95b, Ch. 10]. Sorting a sequence of items $x_1, x_2, \cdots, x_n$ produces a permutation of the sequence according to a total ordering of the elements, in this case $x_i \leq x_{i+1}$.

Component process

Process 7.12 lists a `Node` process that is replicated to form each node of the hypercube. The main operation of this is to, in sequence, establish the channel connections, distribute the input array, perform the sorting operation and then collect the output array. Since sorting is based on permuting the data, it would be relatively simple to integrate a sorting algorithm into the collection procedure, however, the sorting operation in this example could be replaced by any other hypercube algorithm and so it serves as a simple template.

The `distribute` and `collect` processes, listed in Process 7.9 treat the hypercube as a hierarchical structure by assigning each node to a level (which is computed by the `level` function, taken from the example in Brinch Hansen's computational paradigms [Han95b, Ch. 10]). A hypercube has $n + 1$ levels such that in 0 dimensions a hypercube consists of a single node at level 0 and in 1 dimension the 0th dimension is replicated and this is level one. In general, the nodes in the 0 to ith dimensions are replicated to form the $(i + 2)$th level. The nodes belonging to the first three levels of a hypercube are listed in the following table.

Level	**Nodes**
0	0
1	1
2	2, 3
3	4, 5, 6, 7

Each `Node` has n channels for each dimension of the hypercube (levels 1 to n) and an additional channel corresponding to level 0. This channel is used exclusively by `Node` 0 for external interaction. The n channels are connected in the sequence of dimensions to the corresponding channel end at the neighbouring `Node` in that dimension. This pattern produces a sequence of n pairs of interactions, which can be seen for $n = 2$ with the following table.

i	$i \oplus 1$	$i \oplus 2$
0	1	2
1	0	3
2	3	0
3	2	1

For example, `Node` 1 connects first to `Node` 0 in dimension 1 and then `Node` 3 in dimension 2. These correspond to the connections made by `Node` 0 and `Node` 3. Figure 7.4a illustrates this sequence of interactions procedure on a cube.

The distribution then proceeds one level at a time, with each `Node` first waiting to receive a portion of the input array on the channel in the dimension equal to its level, then distributing half of this in the channels in higher dimensions. This starts with `Node` 0 receiving input from the `Master` process (listed in Process 7.11). When this is complete, each `Node` holds a single value from the input array. Figure 7.4a illustrates the operation of `distribute` procedure on a cube.

The procedure to sort the array is listed in Process 7.10 and proceeds in n phases, following the same pattern of interactions as the sequence of connections. Each pair with labels a and b where $a < b$ exchange their values u and v. a keeps v if $v < u$ and b keeps u if $u > v$. After n exchanges, the values are sorted according to the labelling of `Node`s, with one element per `Node`.[4]

When the sorting is completed, the results are collected with the `collect` procedure listed in Process 7.9. This operates in the opposite way to the `distribute` procedure, with each of the 'leaf' `Node`s sending their results first.

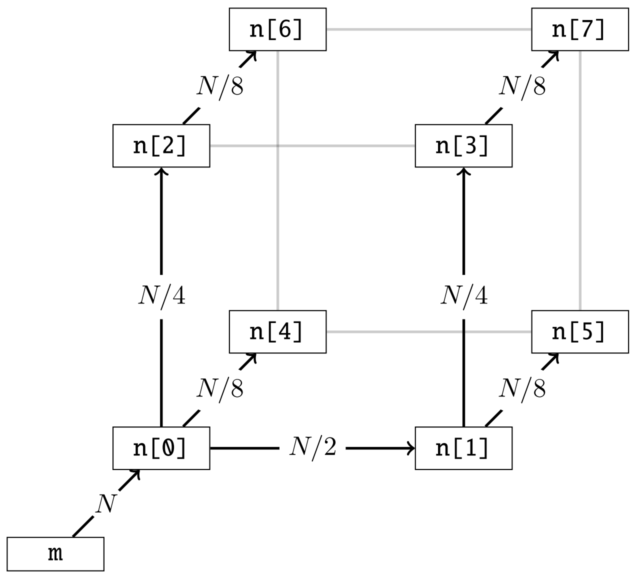
The program

The following is a complete process structure for a 3D hypercube (cube) to that executes the parallel sorting algorithm.

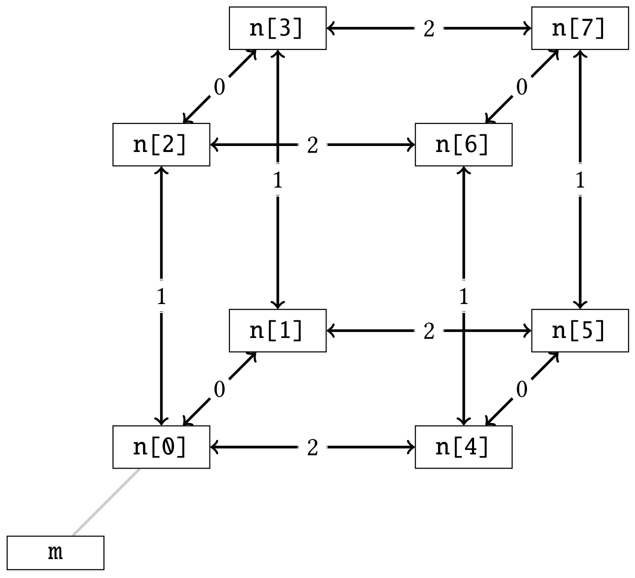
```
val D is 3:
val N is 1 << D:
function nbr(val id, val d) is ... :
function level(val id) is ... :
process distribute(val id, chanend[D+1] c, var[n] a, val n) is ... :
process collect(val id, chanend[D+1] c, var[n] a, val n) is
process sort(val id, chanend[D+1] c, var a) is ... :
process Master(process Node n) is ... :
process Node(val id, Node[N] c) is ... :
{ m is Master(n) &
  n is par [i=0 for N] Node(i, m, n) }
```

This structure is illustrated by the diagrams in Figure 7.4.

[4]For more than one value per `Node`, the merge sort algorithm could be employed in these exchanges. The pair would first sort their sequence locally, then exchange the sequences so they have both. One would then merge the two sequences to obtain a sorted sequence with the lowest values, and the other with the highest [Fos95, §11.4].



(a) distribution of input



(b) the sequence of pairwise interactions

Figure 7.4.: A 3D hypercube (cube) process structure to implement a parallel sort. In (a) the communication involved in the `distribute` procedure is indicated (the `collect` procedure operates in the opposite way). In (b) the sequence of pairwise interactions used to establish channel connections and perform the sort are indicated.

```
function level(val id) is
  % return a level for a hypercube node
  valof
  { var d, dmax, l:
    dmax := 0; l := 0;
    while dmax < id do
    { dmax := 2*dmax + 1;
      l := l + 1 } }
  result l

process distribute(val id, chanend[D+1] c, var[n] a, val n) is
{ var lvl, len, base:
  lvl := level(id);
  % receive $2^{level(id)}$ elements of 'a' from the parent
  c[lvl] ? len;
  seq [i=0 for len] c[lvl] ? a[i]
  % redistribute topmost portions of 'a' to each child
  base := len/2;
  seq [i=lvl+1 for D-lvl-1]
  { len := len / 2;
    c[i] ! len;
    seq [j=base for len] c[i] ! a[j];
    base := base-len } }

process collect(val id, chanend[D+1] c, var[n] a, val n) is
{ var lvl, len, lenPart:
  lvl := level(id);
  % receive portions of array from each child
  lenPart := 1;
  seq [i=D-1 for D-1-lvl step -1]
  { c[i] ? len;
    seq [j=len/2 for len] c[i] ? a[j];
    lenPart := lenPart * 2 };
  % send a complete portion to the parent
  c[lvl] ! lenPart;
  seq [i=0 for lenPart] c[lvl] ! a[i] }
```

Process 7.9: Procedures to distribute and collect an array variable, whose length is integral power of two. These treat the $N = 2^n$ node hypercube as a hierarchical structure with $n + 1$ levels.

```
function nbr(val id, val d) is
  % return the id of a neighbour for a hypercube node in dimension d
  valof result id xor (1<<d)

process sort(val id, chanend[D+1] c, var a) is
{ var swp:
  % perform D pairwise exchanges
  seq [i=0 for D]
    if id < nbr(id, i+1)
    then
    { c[i] ! a;
      c[i] ? swp;
      if { swp < a: a := swp } }
    else
    { c[i] ? swp;
      c[i] ! a;
      if { swp > a: a := swp } } }
```

Process 7.10: The hypercube sorting procedure, which is executed by each `Node`. It works by making `D`$= \log N$ pairwise exchanges so that the sequence distributed between them is sorted according to their IDs.

```
process Master(process Node n) is
  interface(chanend c):
{ connect c to n.c[0];
  var a[N];
  seq [i=0 for N] c ! N-i-1; % input the array to the hypercube
  seq [i=0 for N] c ? a[i] } % output the sorted array
```

Process 7.11: A master node to source and sink data from the hypercube.

```
process Node(val id, process Master m, Node[N] n) is
  interface(chanend[D+1] c):
{ seq [i=1 for D] connect c[i-1] to n[nbr(id, i)].c;
  if { id = 0: connect c[0] to m.c };
  var a[N]:
  distribute(id, c, a, N);
  sort(id, c, a[0]);
  collect(id, c, a, N) }
```

Process 7.12: The hypercube node. The instance of this process with `id` 0 initialises the input array. This is then distributed between the other nodes, then the sort is performed and the results collected.

7.2. Server structures

Abstractions are essential in managing the complexity of a program and in scalable parallel programming, abstractions necessarily involve distributed parallelism. In sire, abstraction of parallel program components is dealt with using servers. Their key ability is to allow the expression of a global state that can be shared. This gives rise to a subroutine mechanism where client connections can be rebound dynamically.

This section explains the ways in which servers can be used to express abstractions to structure a program. It also explores some of the ways in which the concept of a server is analogous to conventional language primitives such as variables, arrays and procedure calls.

7.2.1. Single servers

A server process can be used to encapsulate state and make this available to a scope through a set of calls.

Servers providing remote storage

Consider the server definition given below in Process 7.13 that provides remote storage by containing an array declaration and providing in its interface calls to read and write to particular locations in it.

```
server Store() is
  interface(call read(val i, var v), write(val i, val v)):
{ var[N] a:
  alt
  { accept read(val i, var v): v := a[i]
  | accept write(val i, val v): a[i] := v } }
```

Process 7.13: A server that provides access to an array.

In the following, an instance of a `Store` server is declared with name `s`.

```
server Store() is ... :
s is Store(): ... s.write(i, 23); ... s.read(i, v); ...
```

The server executes in parallel with the scope of the declaration, which makes calls to the server to read and write from it. This is equivalent to the following use of a array variable with infix notation:

```
var[N] a: ... a[i] := 23; ... v := a[i]; ...
```

Servers providing remote calls

A server also encapsulates behaviours that are invoked with the same semantics as conventional procedure calls. Based on the substitution semantics of server calls (see §6.8 [p. 83]), the following local call

```
process increment(var v) is v := v + 1:
... increment(v) ...
```

has the same effect as

```
s is interface(call increment(var v)):
  alt { accept increment(var v): v := v + 1 }:
... s.increment(v) ...
```

However, in the version with the server, the call is executed by a different parallel process, which may be executing remotely on a different processor. The server notation therefore makes it convenient to move between local process calls and remote server calls. This makes it easy to modify a program to offload the work in a procedure to another processor, potentially where the data that it operates on is actually stored, and to employ additional parallelism in the program's evaluation.

Moving program to servers

Since procedures and functions can be passed as parameters, server calls can be used as a mechanism to *move* program components to operate on data *locally* at the server. This is illustrated with the modified version of Process 7.13 below, which provides an additional call that applies a procedure parameter to the array.

```
server Store() is
  interface(call read(val i, var v), write(val i, val v),
      apply(process f(var[n] a, val n))):
{ var[N] a:
  alt
  { accept read(val i, var v): v := a[i]
  | accept write(val i, val v): a[i] := v }
  | accept apply(process f(var[n] a, val  n)): f(a, N) }
```

Process 7.14: A modified version of Process 7.13 that allows the internal array to be modified locally.

Servers as data structures

The encapsulation of both state and behaviour in an entity that is available throughout a conventional variable scope provides a basis for *abstraction*. This allows servers to be used to implement data structures.

Process 7.15 shows the definition of a server type that implements a stack data structure in which items are added and removed in a first-in last-out manner by a single client. It maintains an array `s` that it uses to store the values and a pointer `p` that records the index of the last element. Calls are also provided to query the occupancy of the stack in order that accesses can be made safely.

The following creates an instance of the `Stack` server type that is used by a single client process.

```
s is Stack():
{ ... s.full(f); if { ~f: s.push(0) }; ...
  ... s.empty(e); if { ~e: s.pop(v) }; ... }
```

7.2.2. Multiple servers

Replicated server declarations create arrays of servers. The following creates an array of `N` `Store` servers.

```
server Store() is ... :
s is [N] Store(): ... s[i].write(j, 19); ... s[i].read(j, v); ...
```

```
server Stack() is
  interface(call push(val v), pop(var v), full(var b), empty(var b)):
{ var[N] s:
  var p:
  initial p := 0:
  alt
  { accept write(val v):
    { s[p] := v;
      count := count + 1 }
  | accept read(var v):
    { v := s[p];
      count := count - 1 }
  | accept full(var b):
      b := p+1 = N
  | accept empty(var b):
      b := p = 0 } }
```

Process 7.15: A stack server.

The effect of this is to create a large *distributed store*. Specific regions of the store are accessed by subscripting component servers. This again is equivalent to a similar use of conventional array variables, for example:

```
var[N][N] s: ... s[i][j] := 19; ... v := s[i][j]; ...
```

The equivalences between conventional variables and procedures, and the corresponding server constructs, permit simple refactoring to move from the local (conventional) form to the remote (server) form. This provides the potential with a small amount of refactoring to distribute state or to employ parallelism. This idea will be explored further in later examples.

7.2.3. Shared servers

A server can be shared between a number of clients, which can make calls at arbitrary times, but a server is able only to service the calls in sequence.

A shared buffer

A simple shared data structure is a *bounded buffer* in which the order that items are read from it is the same as they are written. Process 7.16 shows the definition of a `Buffer` server, which is based on the buffer example in [INM88a, Ch. 1]. It maintains an array `b` with pointers `inp` and `outp` that are used to allocate storage at the beginning and end of the buffer in a circular manner.

```
server Buffer() is
  interface(call write(val v), read(var v)):
{ var[N] b:
  var inp, outp, count:
  initial { inp := 0; outp := 0; count := 0 }:
  alt
  { count+1 < N & accept write(val v):
    { b[inp] := v;
      inp := (inp + 1) rem N;
      count := count + 1 }
  | count > 0 & accept read(var v):
    { v := b[outp];
      outp := (outp+1) rem N;
      count := count - 1 } } }
```

Process 7.16: A buffer server.

The buffer server example also illustrates the use of guarded call alternatives. If the buffer is read when it is empty, then the call will block until something is written to it. Conversely, when the buffer is full, a write will block until it is read. This allows the buffer to be safely shared between one or more of parallel processes, for example, by a pair of *producer* and *consumer* processes:

```
server Buffer() is  ... :
process Producer(server Buffer b) is
  while true do b.write(0):
process Consumer(server Buffer b) is
  var v: while true do b.read(v):
b is Buffer(): { Producer(b) & Consumer(b) }
```

The following diagram illustrates this process structure:

$$\begin{tikzcd} \boxed{\texttt{Producer}} \arrow[r, "\texttt{write}(\cdots)"] & \boxed{\texttt{b}} \arrow[r, "\texttt{read}(\cdots)"] & \boxed{\texttt{Consumer}} \end{tikzcd}$$

A task farm

A *task farm* (see §3.1.3 [p. 23]) can be expressed concisely using a single shared server that acts a the *farmer* and a replicated *worker process*. Processes 7.17 and 7.18 give the definitions these respectively.

The Farmer maintains a list of outstanding work, represented by an integer value between 0 and $N-1$, and an array of results. Request calls do not block, but when there is no work remaining a special value NONE is sent to indicate this. Return calls also do not block and the supplied result is written to the array. The Worker process repetitively requests work from the farmer until there is no work remaining. For each piece of work, it performs a computation with a call to a compute process, and returns the result.

The following is a complete task farm with an array of Worker processes that share access to a Farmer server.

```
val NONE is 0:
server Farmer() is ... :
process Worker(server Farmer f) is ... :
f is Farmer():
par [i=0 for N] Worker(f)
```

```
server Farmer() is
  interface(call request(var w), result(val w, val r)):
{ var results[N]:
  var work:
  initial work := 0
  alt
  { % provide a worker with work if any remains or NONE if not
    accept request(var w):
      if work < N
      then
      { w := work;
        work := work + 1 }
      else w := NONE
    % accept a result from a worker
  | accept result(val w, val r):
      results[w] := r } }
```

Process 7.17: A task farm farmer server.

```
process Worker(server Farmer f) is
{ var running: running := true;
  while running do
  { var work, result:
    % request some work
    f.request(work);
    % if there is none then terminate
    if work = NONE
    then running := false
    else
    { % otherwise, perform the computation and return the result
      compute(work, result);
      f.result(work, result) } } }
```

Process 7.18: A task farm worker process.

7.2.4. Data abstractions

A central challenge associated with programming distributed memory parallel computers is that the working dataset of a particular problem will generally be larger than the capacity of any one memory. Therefore, data has to be distributed between processes, and as a computation progresses, processes have to access components of the data stored in remote memories.

When the dataset of a problem and its associated access patterns can be decomposed in a regular way, it can be expressed as a process structure with message-passing communication, such as the examples described in §3.1.3 [p. 22] and the examples earlier in this chapter in §7.1 [p. 101]. However, when a problem cannot, it is awkward to express it in terms of a fixed process structure. In these cases, collections of servers can be used to build distributed representations of data that can be accessed by collections of client processes.

The following examples demonstrate some simple ways this can be done. The examples also serve to illustrate the kinds of techniques employed to manage distributed data.

A large random access memory

A desirable quality of a distributed-memory architecture is to build larger random-access memories. This establishes a base-case for their use, providing the ability to run large memory sequential programs. The following example explores how a large random-access memory structure can be *expressed* using the features provided by sire (in Chapter 10, the efficiency with with the UPA can *execute* this structure is investigated).

A distributed memory can be expressed with two types of server, one that is replicated to provide the distributed storage and one that provides access by dealing with the structure of the storage array. Process 7.19 lists the definition for a server that implements a distributed random-access memory. In this, the Store server provides access to remote storage and the Access server holds a reference to an array of Store servers and provides calls to read and write locations over the array. Each access is converted into an integer server index ($\lfloor \texttt{addr}/\texttt{S} \rfloor$) and an index within the storage of that server ($\texttt{addr} \mod \texttt{S}$). For simplicity, accesses outside of the valid index range 0 to $\texttt{N}(\texttt{S}-1)$ are ignored. The RAM server definition combines the Store and Access servers so that a single declaration can be introduced that encapsulates the behaviour of the entire server structure. Since client processes will only interact with the Access server, the array of Store servers is hidden using a *hiding definition*, making only the interface m visible.

The following is an outline of a program that uses this composite server structure. With N storage servers and S words storage per server, it provides $\texttt{N} \times \texttt{S}$ words of storage to a single client process.

```
val S is 1024:
val N is 10:
server Store() is ... :
server Access(server Store[N] s) is ... :
server RAM() is ... :
m is RAM():
{ ... m.write(addr, 42); ... m.read(addr, v); ... }
```

Figure 7.5 shows the process diagram for this program, illustrating the internal structure of the RAM server type.

```
server Store() is
  interface(call read(val i, var v), write(val i, val v)):
{ var[S] a:
  alt
  { accept read(val i, var v):
      v := a[i]
  | accept write(val i, val v):
      a[i] := v } }

server Access(server Store[N] s) is
  interface(call read(val addr, var v), write(val addr, val v)):
  alt
  { accept read(val addr, var v):
      s[addr/N].read(addr rem S, v)
  | accept write(val addr, val v):
      s[addr/N].write(addr rem S, v) }

server RAM() inherits
  from
  { s is [N] Store():
    m is Access(s) }
  interface m
```

Process 7.19: A server providing a distributed random access memory.

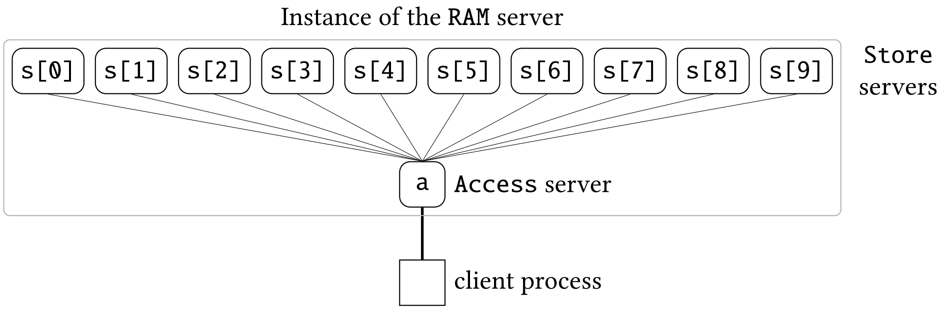


Figure 7.5.: A random access memory server structure (whose components are listed in Process 7.19) that executes in parallel with a single client process. The client performs reads and writes to arbitrary addresses via an Access server that dispatches the accesses to particular Store servers.

Caching accesses

A simple extension to the RAM server is to introduce caching to improve performance by reducing the number of accesses that are potentially dispatched to remote Store servers.

Process 7.20 extends the Access with a simple *write-through* caching scheme where every write is written to a random location in the cache and also to a Store server. A read causes the cache to be checked first with a linear search. The value is returned and the call completed if it is held in the cache, otherwise it is fetched from a server. Entries are inserted into the cache with a separate function h that computes the hash of a value.

```
server Access(server Store[N] s) is
  interface(call read(val i, var v), write(val i, val v)):
{ var[M] cache, indices:
  var k, hit:
  initial seq [i=0 for M] indices[0] := 0
  alt
  { accept read(val i, var v):
    { hit := false;
      if [j=0 for M]
      { i = indices[j]:
        { v := cache[j];
          hit := true } };
      if { ~hit:
      { k := h(i);
        s[i/N].read(i rem S, cache[k]);
        indices[k] := i;
        v := cache[k] } } }
  | accept write(val i, val v):
    { k := h(i);
      cache[k] := v;
      indices[k] := i;
      s[i/N].write(i rem S, v) } } }
```

Process 7.20: A modified version of the RAM Access server listed in Process 7.19 to implement a simple write-through caching scheme.

A large shared random access memory

It is natural to extend the distributed random access memory to support concurrent access for a set of parallel processes. This corresponds closely to emulating a PRAM and there are many techniques to do this in an efficient way (the PRAM model was explained in §2.2.2 [p. 15]). A parallel random-access structure is important to consider since it provides the natural basis on which to implement large distributed data structures, where read and write operations are generalised to arbitrary abstract operations.

The most simple variant of a PRAM is *exclusive-read, exclusive write* (EREW) where there are no access collisions. To ensure accesses are evenly distributed and there is not excessive contention at any particular processor, it is necessary to evenly distribute the logical address space over the set of physical address spaces provided by the processors that implement the memory. Each *logical address* $a \in \{1, 2 \cdots, m\}$ is mapped to a processor $p \in \{1, 2, \cdots, n\}$ where $m \gg n$, $p = h(x)$ and h is a suitable hash function. The effect is that the logical address space is partitioned into a number of disjoint components that are mapped to processors.[5]

The structure of an EREW-PRAM follows that of the RAM in the previous section; Process 7.21 lists the definitions of the server and its components. The `Store` server uses an additional hash table server `Table` to store addresses assigned to it[6] and the `Access` server uses a hash function `h` to select servers to dispatch read and write requests to. Therefore, all instances of the `Access` server compute the same address mappings.

Process 7.21 shows the definition of the complete server structure with the array of `N` hidden `Store` servers and array of `M` `Access` servers made visible through the `ParallelRAM` server type. The following program shows an instance of the `ParallelRAM` server type being used by a collection of parallel clients.

```
val S is 1024:
val N is 10:
val CLIENTS is 4:
val M is CLIENTS:
function h(val v) is ... :
server Table() is ... :
server Store() is ... :
server Access(Store[N] s) is ... :
server ParallelRAM() inherits ... :
m is ParallelRAM():
par [i=0 for CLIENTS]
{ ... m[i].write(addr, 42); ... m[i].read(addr, v); ... }
```

In the program, each client performs reads and writes through a particular `Access` server and the number of `Access` servers are matched to the number of clients. If the number of clients changes between different phases of execution, then the number of `Access` servers would be determined by the maximum number of client processes, and in phases with fewer clients, some `Access` servers would remain unused. Figure 7.6 shows the process diagram of this program, illustrating the internal structure of the `ParallelRAM` server, marking all of the potential client-server call interactions.

[5] A well known class of *universal hash functions* that can be computed in constant time are suitable to perform the addresses-processor mappings [CW79]. These employ randomisation to choose a hash function from a family of functions to guarantee few collisions in expectation. There is therefore a small probability that the chosen hash function will perform poorly; in this case, typically a new one is be chosen and the data rehashed. Using universal hashing for processor-address mappings, the maximum number of logical addresses placed on a single processor is at most $O\left(\frac{\log n}{\log\log n}\right)$ [Gon81] and this can be reduced to $O(\log\log n)$ by using two independent hash functions to distribute collisions between two locations [Mit91].

[6] A definition for `Table` is not given but it would follow the conventional behaviour of a hash table with calls to insert, update and search; see [Knu99, §6.4].

```
server Store() is
  interface(call read(val i, var v), write(val i, val v)):
{ t is Table(S):
  alt
  { accept read(val addr, var v):
      t.lookup(addr, v)
  | accept write(val addr, val v):
    { var s: t.search(addr, s);
      if s ~= NONE then t.update(addr, v) else t.insert(addr, v) } } }

server Access(server Store[N] s) is
  interface(call read(val addr, var v), write(val addr, val v)):
  alt
  { accept read(val addr, var v):
      s[h(addr)].read(addr, v)
  | accept write(val addr, val v):
      s[h(addr)].write(addr, v) }

server ParallelRAM() inherits
  from
  { s is [N] Store():
    a is [M] Access(s) }
  interface a
```

Process 7.21: A server providing a distributed parallel random access memory.

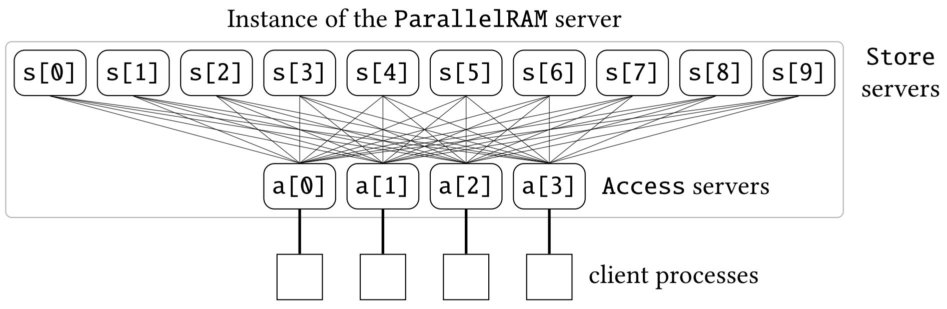


Figure 7.6.: The parallel random access memory server structure (whose components are listed in Process 7.21) in composition with a collection of client processes. Each client interacts with one Access server and performs concurrent reads and writes to the memory. Each Access server dispatches accesses to particular Store servers.

7.2.5. Embedding process structures

The previous examples in this section have demonstrated how servers can be used to build resource and data abstractions in which collections of servers communicate with calls. However, servers can also communicate with other servers through channels in the same way that processes do, and can therefore be used to express message passing structures. This is the mechanism by which process structures are *embedded* into other processes, to be used as subroutines.

A message-passing structure is expressed in this way by formulating its behaviour in terms of a reaction to some external stimulus, i.e. a call to perform an action. Each component process is a server that repetitively waits for input on a set of channels.

Embedding a pipeline

To demonstrate the use of servers to embed process structures, the Sieve of Eratosthenes example from §7.1.1 [p. 101] is modified to be expressed in this form. The behaviour of the original version was just to generate a set of primes in an array and terminate. This could be used as a subroutine in this form by writing to an output array in the scope of the parent process. However, using a subroutine in this way precludes it from storing any state and incurs the overheads associated with initialising and terminating a set of parallel processes each time it is used. In the modified version of the pipeline example, it provides a call to test whether a number is prime or not, using the pipeline structure to compute the answer.

Process 7.22 lists the definition of the pipeline server `Node` which, in its initialisation phase, connects the two channels and receives its prime. It then waits for incoming numbers to test. Numbers that are not divisible by its prime are forwarded on and numbers that are divisible are forwarded as the special value `NONE`. This means that the pipeline always produces a value. Process 7.23 lists the definition of the `Control` server, which source and sinks values from the pipeline. It first circulates a stream of integers to initialise the pipeline and then serves a call `isPrime` to test the primality of a particular number.

The following program outline shows a new server constructed from the `Node` and `Control` servers to implement a subroutine to test the primality of numbers up to `N`= 16. Figure 7.7 illustrates the process diagram of the program.

```
val SQRTN is 4:
val N is 16:
val NONE is false:
server Control(chanend pin, pout) is ... :
server Node(val i, server Control m, server Node[SQRTN] p) is ... :
server TestPrimes() inherits
  from
  { m is Control(p) &
    p is [i=0 for SQRTN] Node(i, m, p) }
  interface m:
p is TestPrimes():
... p.isPrime(11, r); ...
```

```
server Node(val i, server Control m, Node[SQRTN] p) is
  interface(chanend in, out):
{ var p, mp:
  initial
  { if i = 0
    then connect in to m.out
    else connect out to p[i+1].in;
    if i = N-1
    then connect in to p[i-1].out
    else connect out to m.in;
    p := NONE;
    while p ~= NONE do in ? p;
    mp := p }:
  alt { accept in ? m:
    { while m > mp do
        mp := mp + p;
      if m < mp then out ! m else out ! NONE } } }
```

Process 7.22: A pipeline node server.

```
server Control(chanend pin, pout) is
  interface(chanend out, in, call isPrime(val p, var r)):
  var i, v:
  initial
  { connect out to pin;
    connect in to pout;
    in ! 2;
    seq [i=3 for N step 2]
    { in ! i; out ? v } }:
  alt {
    accept isPrime(val p, var r):
    { in ! v;
      out ? r } }
```

Process 7.23: A pipeline control server.

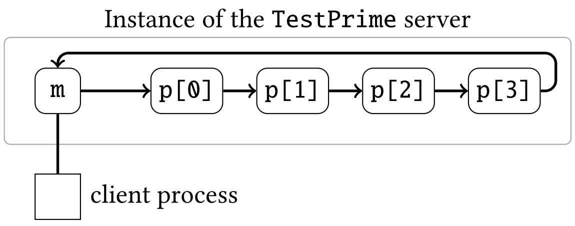


Figure 7.7.: A server structure that employs a pipeline internally (with the components listed in Processes 7.22 and 7.23) to implement a primality testing algorithm that can be used as a subroutine.

7.3. Discussion

7.3.1. Sequential program narrative

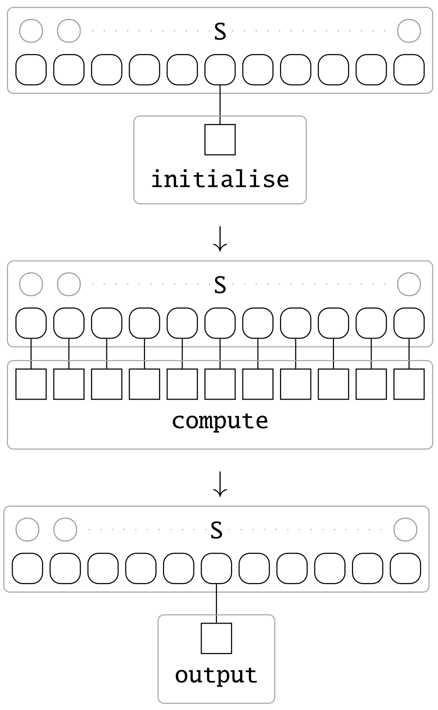
Servers provide a simple way to build and structure programs. They can encapsulate both behaviour and state (in fact arbitrary program components), and are introduced with conventional-style declarations and scoping rules. This allows a distributed parallel program to be expressed as a *sequence* of declarations and actions that operate using them.

A generic high-level program sequence, which is familiar to the imperative model of programming, might consist of the creation and initialisation of a data structure, a computation that operates on the data structure, then the output of any results. Sire allows this kind of program structure to be expressed in the following way:

```
s is S():
initialise(s);
compute(s);
output(s)
```

with the significant capability that each component can employ distributed parallelism.

The following diagrams illustrate the evolution of the above program in both *time*, from one computational component to the next, and in *space*, with the creation of new processes and dynamic rebinding of connections between clients and servers.



In contrast to this approach, program components in occam are composed in monolithic parallel blocks and it is difficult to discern the structure of the program. The situation is worse still with programming approaches such as MPI that do not support abstraction of nesting of parallel components. In this case, the different logical components of a program must be merged into a single parallel entity.

A further advantage from constructing a distributed parallel program using servers in a sequential way is related to the ability to perform testing. Since the interface to a server is a set of calls, they can be instantiated and tested in isolation. This approach is formalised in the *unit testing* methodology, where the smallest units of a program are subjected to a set of tests to establish their correct behaviour. This is generally considered to be good software-engineering practice.

7.3.2. Software resource management

Delivering performance in concurrently accessible distributed data structures relies on simple and established techniques such as caching to reduce latency, hashing to distribute load, combining and replication to reduce the effects of hotspots and the model of consistency.

Integrating specialised mechanisms into hardware is unattractive because it is difficult to choose a best general behaviour or way to parameterise a mechanism, such that it can be easily tuned to particular workloads. Moreover, a risk with integrating complex functionality into general-purpose systems is that they can exhibit pathological behaviours in certain unpredictable cases that can result in poor performance.

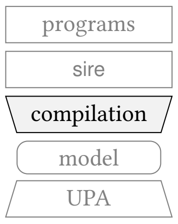
With a language that provides a small set of primitive mechanisms that relate closely to the operation of the underlying hardware, it is possible, and indeed convenient, to implement such mechanisms in software. This provides the additional advantage that the software mechanisms can be further specialised to the particular application, increasing performance and making the application's behaviour more predictable. Similar benefits were observed, for example, with a software caching scheme in the RAW microprocessor [MFLA99].

Resource management in PRAM implementations

Caching and hashing were discussed in §7.2.4 [p. 121] in the RAM and EREW-PRAM examples. For other less restrictive types of PRAM where reads can occur concurrently (a *concurrent-read, exclusive-write* (CREW) PRAM), or both reads and writes can occur concurrently (a *concurrent-read, concurrent-write* (CRCW) PRAM), an emulation must also deal with access contention as well as distributing the address space.

There are two established techniques for reducing access contention are relevant to any other shared distributed data structure similar to a PRAM, such as a database, file system or hash table. One is *replication* where data items are made available on multiple processors and accesses are distributed among these. When a replicated location is written to, each of the replicas must be updated to maintain consistency. This would require broadcast communications between either the access or storage servers. The other is *combining* where memory accesses are assumed to traverse a network and those accessing a particular location will traverse a tree-structured sub-network [Ran87]. The switches at the nodes of this tree can identify identical requests and permit only one to continue. When the corresponding response returns, responses are generated for each original request. A combining network could therefore be implemented with a network of servers between the client-facing access servers and the storage servers.

To note, the RAM and parallel RAM examples in §7.2.4 [p. 121] did not consider a consistency model. Consistency is a prime concern to these, or indeed any other concurrently accessible shared distributed data structure. The level and nature of the consistency required will depend on factors such as the programming model and the algorithms being expressed. The ability to express a consistency model in software is therefore a great advantage of the design of sire.



Chapter 8. Compilation of sire to the UPA

This chapter demonstrates that sire is capable of an efficient implementation. It does so by describing the process of compilation for each of the non-conventional aspects of the language that are related to parallelism and communication. The process is *minimal* in the sense that it is sufficient to deal effectively with all aspects of the language. There is however scope for optimisations to improve performance or resource usage; some potential ideas are discussed at the end of the chapter.

8.1. Overview

Compilation of a sire program is composed of the following stages:

1. lexing and parsing;
2. semantic checking;
3. source-to-source transformation of the program into a canonical form;
4. generating executable code.

Success of the first two stages establishes a valid input program that conforms to the syntax and rules of the language. The output of these stages is an abstract representation of the syntax. The abstract representation is transformed in stage 3 into a *canonical form* and finally emitted as executable code in stage 4.

The remainder of this chapter describes stage 3 and the non-conventional aspects of stage 4. Stages 1 and 2 are not described because they can be implemented with conventional compilation techniques.

8.1.1. Execution model

A compiled sire program consists of two binaries, a *master* that contains the compiled user program and the *run-time kernel*, and a *slave* that only contains the kernel. The run-time kernel is a collection of processes and routines that implement dynamic aspects of sire.

The system on which a sire program executes consists of a set of processors that are labelled 0 to $N-1$ and connected by a communication network. Each processor has a memory, a pool of communication *channel ends* and the ability to execute a collection of processes simultaneously. Two channel ends belonging to different processes are *connected* for the channel to be established and for the processes to communicate.

The system is initialised by loading the master binary onto processor 0 and replicating the slave binary over processors 1 to $N-1$. Execution begins with each processor creating a kernel *service process* that is active for the duration of the user program execution. Service processes are responsible for managing local channel ends and processes. After this, the service process on processor 0 initialises the user program to execute as a new process.

As the the user program runs and creates more processes, the execution proceeds in *time* and *space* as components of the program are *moved* to other processors and new processes are allocated and deallocated dynamically to execute them. Termination of requires all active processes to terminate and for the (potentially distributed) flow of control to return to processor 0 before the first process finally terminates.

8.1.2. Key aspects

Before describing the transformation and code generation stages, the following sections highlight several key aspects of the compilation process and their interplay with the design of the sire syntax.

Dynamic distribution of the program

A key ability of sire is to permit efficient dynamic distribution of program code, so that components of a program are *moved* at run-time to the processors on which they execute. This is perhaps the most unconventional aspect of the compilation process compared with other distributed programming languages that statically allocate program code to each processor. The are a number of reasons why this is a good approach to take, and each of the following reasons listed strengthen as the number of processors increases and the memory capacity per processor decreases.

- *Run-time reuse of resources.* Dynamic distribution makes a reuse of the available memory between phases of a program; a processor needs only store the processes it is currently executing, rather than having to store all the processes that it will execute in the duration of the program.
- *Minimal compilation time.* The compilation time should not depend on the number of processors in the target system because it only needs to produce two program binaries: a *master* one that contains the run-time kernel components (that support the execution of sire) and the user's program, and a *slave* one that only contains the run-time kernel components.
- *Minimal portable binaries.* For the same reason, the size of the resulting program binary is also independent of the number of processors in the target system. This will minimise the binary's size and potentially provide portability between different systems with varying numbers of processors.
- *Rapid booting.* A two-image binary format allows a system to be booted by loading the master image and *replicating* the slave image. Replication can be performed recursively and in a time logarithmically related to the number of processors. Binaries with separate images per processor must be loaded sequentially and in a time linearly related to the number of processors; even for relatively small systems the time taken to boot in this way can be significant.

The *single-program multiple-data* (SPMD) compilation model, which is used by MPI programs for example, shares the advantages with fast compilation, minimal binary size and fast booting but since the single binary is replicated for each processor, there will be a large amount of redundancy. For HPC-scale systems this might not be a serious issue with substantial amounts of memory per-processor, but with much smaller memories, e.g. less than 1 MB, efficient memory usage is essential.

Allocation of processing resources

A second key ability of sire is that processor allocation can be dealt with at compile time, rather than at run time. This is attractive because dynamic processor allocation, particularly in highly parallel systems, is problematic:

- it is inherently difficult to perform in a scalable way;
- it introduces performance overheads;
- performance is less predictable;
- it can potentially cause non-deterministic behaviour in programs that do not terminate properly or exhaust the number of available processors.

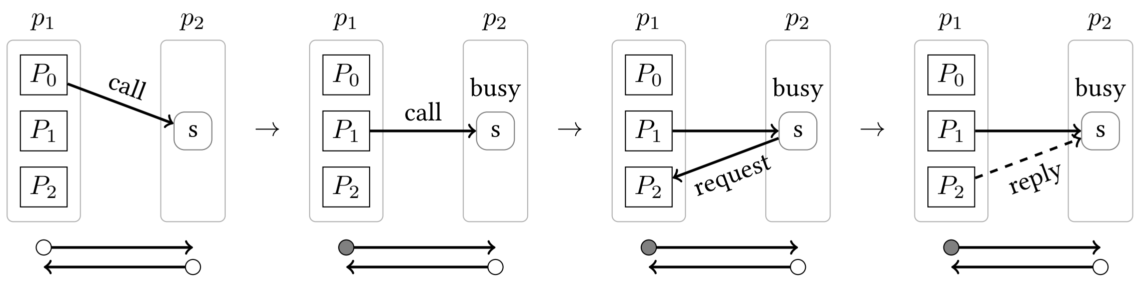


Figure 8.1.: An illustration of deadlock arising from many-to-one communications. In this, P_1, P_2 and P_3 are processes that execute on processor p_1 and s is a server that executes on p_2. A single bidirectional communication channel connects p_1 to p_2 that has space to buffer a single message in each direction. First, P_1 makes a call to s, which is accepted by s. Second, P_2 makes a call to s, which is blocked since s is busy. Third, s send a message to P_3 as part of the execution of the call P_1 made. This is received by P_3 because the outgoing channel is not in use, but P_3 is not able to reply since it is blocked by the call request from P_2.

Minimising the costs of processor management is essential to effectively use highly parallel architectures and to maximise the granularity of parallelism that can be exploited for a particular problem. A direct analogy can be drawn here between stack- and heap-based memory allocation in sequential programs. Stack-based memory allocation is *scheduled* at compile time and a program makes dynamic reuse of stack memory for procedure calls, whereas heap-based memory allocation is performed at run time at a much greater cost and with must less predictability.

The only restriction required to allow compile-time processor allocation for a sire program is that parallel replicators must have constant-valued counts. This however, is not an inherent restriction in the language and there is scope for an implementation that supports dynamically-sized replicators. Since the compilation process generates code that distributes itself dynamically it would be simple to modify the process to support dynamic processor allocation; this is discussed in §8.6.1 [p. 175].

Preventing deadlock with servers

A crucial issue with the compilation of sire programs is to prevent deadlock from arising in many-to-one patterns of communication with servers. This is a particularly important problem with systems that have limited memory and capacity for buffering.

When a message traverses the network from its source to a destination, at each stage it occupies buffering resources. If at some point the message becomes blocked by the presence of another message occupying resources in its path, then it will remain blocked. Deadlock occurs when messages are blocked by each other.

When a number of client processes attempt to interact with a server simultaneously, their requests to do so will become blocked if the server is busy since their route out of the network to the server is blocked. Then, if the server needs to engage in a communication to complete the current client request, there may be no available route to the destination and deadlock will occur. Figure 8.1 illustrates this with a simple example.

To remove the possibility of deadlock:

- either the server must not engage in any communication during the servicing of a request;
- or, the client requests must not be able to become blocked in the network, guaranteeing that routes will be available into the network.

In the compilation process described in this chapter, the former case holds for the *service processes* in the run-time kernel, which are introduced in §8.4.2 [p. 157], and the latter case is guaranteed for server calls with an interrupt mechanism to queue calls from all potential clients.

Implementing language constructs through transformations

A large part of the compilation process is performed by transforming language constructs into a simplified *canonical* form in terms of a small subset of sire. The advantages of this approach are that the output of the transformations is understandable by the programmer and only the canonical subset resulting from the transformations needs to be implemented.

8.2. Program transformations

In the first phase of compilation, a sire program is transformed into a *canonical form* such that:

- there are no server declarations or parallel replicators;
- procedure-call recursion is permitted;
- processes can include call specifications in their interface and call guards in alternative commands;
- *on clauses* are introduced to distribute execution and the command or process body of an on clause are instances of a process (these are explained in §8.2.3 [p. 139]);
- remote channel-end names are rewritten as absolute references to a particular process on a particular processor (this is explained in §8.2.6 [p. 144]).

Conversion into this form consists of a sequence of eight transformations (based on the properties of sire described in its definition):

1. server declarations are transformed into processes in parallel composition;
2. parallel process instances are substituted with the body of their definition;
3. processes in parallel composition are distributed with the insertion of on clauses and processes are enumerated to assign them identifiers (IDs);
4. interfaces are rewritten to specify the local processor location, process identifier and component index if it belongs to an array;
5. remote calls are rewritten to use local channel ends and connections are inserted to connect these with their remote channel ends implementing the call;
6. the remote channel ends specified as connection targets are rewritten to specify the target channel end absolutely with a processor location, process ID and component index if it belongs to an array;
7. component processes of parallel commands are transformed into instances of processes, thereby explicitly defining their free variables;
8. replicators are transformed into a recursive form.

The transformations can be performed recursively on a representation of the parse tree. Figure 8.2 illustrates the sequence of transformations and the following sections describes each transformation.

The following sections describe each transformation. Where new lower-level syntax is introduced, it is presented in the same way it was in the definition of sire in Chapter 6. Let X be a valid sire program that is produced by the first two stages of the compilation process and in which no name is specified more than once. To illustrate the operation of each phase, the transformations are applied to the program given in Process 8.1. This is a simplified version of the server pipeline from §7.2.5 [p. 126] with 4 `Nodes`.

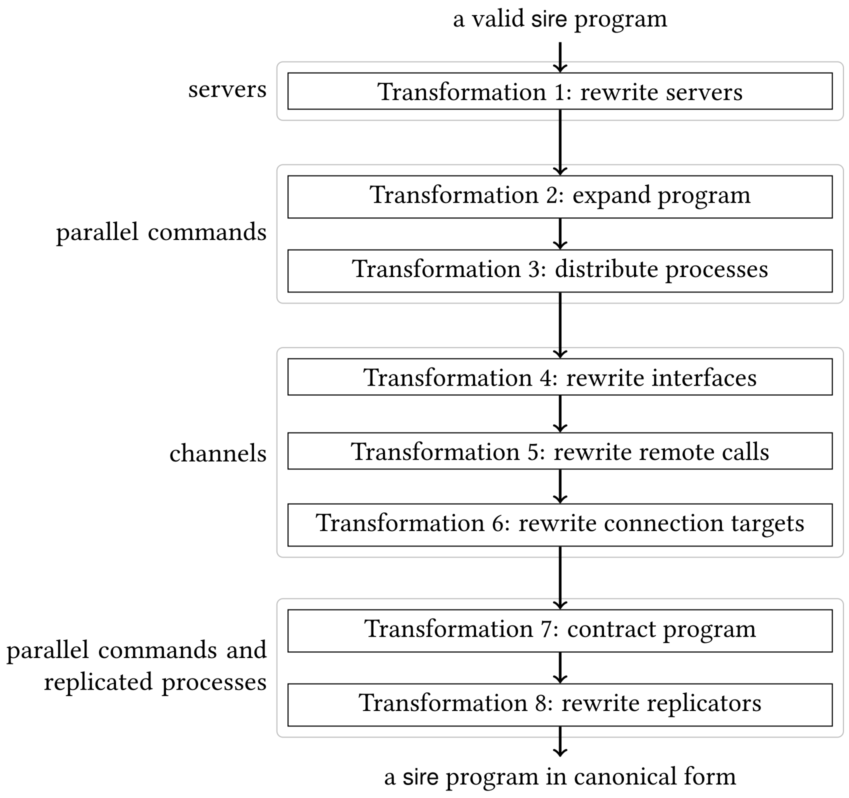

a valid sire program
servers
Transformation 1: rewrite servers
parallel commands
Transformation 2: expand program
Transformation 3: distribute processes
channels
Transformation 4: rewrite interfaces
Transformation 5: rewrite remote calls
Transformation 6: rewrite connection targets
parallel commands and replicated processes
Transformation 7: contract program
Transformation 8: rewrite replicators
a sire program in canonical form

Figure 8.2.: The sequence of program transformations that are applied to a valid sire program to produce a canonical form. This form does not include servers or parallel replicators, processes are distributed explicitly with the on clause and channel-end references are rewritten to specify the process to which they refer.

```
server Node(val i, process Control m, Node[4] p) is
  interface (chanend in, out) to
  initial
  { if i = 0
    then connect in to m.out
    else connect out to p[i+1].in;
    if i = 3
    then connect in to p[i-1].out
    else connect out to m.in }:
  alt
  { var v:
    in ? v;
    out ! v+1 }

server Control(chanend pin, pout) is
  interface (chanend in, out, call x(var v)) to
  initial
  { connect out to pin;
    connect in to pout }:
  alt
  { accept x(var v):
    { out ! 0;
      in ? v } }

server Pipeline() inherits
  from
    m is Control(p[0], p[3]):
    p is [4] Node(m, p)
  interface m

s is Pipeline():
var v: s.x(v)
```

Process 8.1: Compilation transformations example: input program.

8.2.1. Transformation 1: rewrite servers

In this stage, servers in X are transformed from a sequence of declarations into processes.

The transformation

Each server declaration in X is transformed into named processes in parallel composition with their scope with additional calls added to implement their termination. Let T be a specification of a server or instance of a server type where:

- $T_{\text{interface}} = \{F_1, F_2, \cdots, F_k\}$ for $k \geq 0$ is the set of interface specifications;
- $T_{\text{initial}} = \{I_1, I_2, \cdots, I_\ell\}$ for $\ell \geq 0$ is the set of initialisation commands;
- $T_{\text{final}} = \{F_1, F_2, \cdots, F_m\}$ for $m \geq 0$ is the set of finalisation commands;
- $T_{\text{alt}} = \{A_1, A_2, \cdots, A_n\}$ for $n \geq 0$ is the set of guarded alternatives

and $S(x)$ be its scope where x is free, then a server declaration

```
x is $T$ : $S$(x)
```

is transformed into

```
x is interface($F_1$, $F_2$, $\cdots$, $F_k$, call end()):
{ $I_1$; $I_2$; $\cdots$; $I_\ell$;
  var $c$: $c$ := true;
  while $c$ do
    alt
    { accept end() $c$ := false
    | $A_1$ | $A_2$ | $\cdots$ | $A_n$ };
  $F_1$; $F_2$; $\cdots$; $F_m$ }
& { $S$(x); x.end() }
```

where the name c is not specified in the transformed server process. A replicated server declaration

```
x is [i=$b$ for $c$] $T$ : $S(x)$
```

is transformed as above with additional termination calls to each of the component processes

```
x is [i=$b$ for $c$] interface ($F_1$, $F_2$, $\cdots$, $F_k$, call end()): $\cdots$
& { $S$(x); seq [i=0 for $c$] x[i].end() }
```

Example

Process 8.2 shows the server transformation applied to Process 8.1.

```
{ m is interface (chanend in, out, call x(var v), end()):
  { connect out to p[0].in;
    connect in to p[3].out;
    var continue: continue := true;
    while continue do
    alt
    { accept end():
        continue := false
    | accept x(var v):
      { out ! 0;
        in ? v } } }
& p is par [i=0 for 4]
    interface (chanend in, out, call end()):
    { if i = 0
      then connect in to m.out
      else connect out to p[i+1].in;
      if i = 4-1
      then connect in to p[i-1].out
      else connect out to m.in;
      var continue: continue := true;
      while continue do
      alt
      { accept end():
          continue := false
      | var v: in ? v:
          out ! v+1 } }
& { var v:
    m.x(v);
    m.end();
    seq [i=0 for 4] p[i].end() } }
```

Process 8.2: Transformation example: stage 1, rewritten servers.

8.2.2. Transformation 2: expand program

In this stage, instances of process instances in X that contain parallel commands are substituted with the body of their definition with the correct abbreviations of their actuals. This is performed according to the substitution semantics for procedures.

The output of this transformation is an *expanded* form of the program in which no parallel or parallel replicated commands are contained in any definitions. This allows the distribution of nested processes to differ between instances and precludes the need to deal with channel-end parameters and then remove them when they are no longer required in stage 4, where interfaces are removed.

The transformation

Let $f_1, f_2, \cdots, f_n$ be formal parameters and $a_1, a_2, \cdots, a_n$ be actual parameters, for $n \geq 0$. If X contains the definition

$$\texttt{\textbf{process}}\ N\ \texttt{(}f_1, f_2, \cdots, f_n\texttt{)}\ \texttt{\textbf{is}}\ P$$

and P contains a parallel command, then each instance of the type N

$$N\texttt{(}a_1, a_2, \cdots, a_n\texttt{)}$$

is substituted with

$$f_1\ \texttt{\textbf{is}}\ a_1\ \texttt{:}\ f_2\ \texttt{\textbf{is}}\ a_2\ \texttt{:}\ \cdots\ \texttt{:}\ f_n\ \texttt{\textbf{is}}\ a_n\ \texttt{:}\ P$$

Example

Since there are no process instances in Process 8.2, this stage leaves it unchanged.

8.2.3. Transformation 3: distribute processes

In this stage, processes in X are allocated to physical processors according to a static schedule and X is rewritten in a form to direct this allocation at run time. The effect of this is that during the execution of X, processes are allocated and deallocated to processors dynamically (according to the schedule), thereby reusing each processor's available memory.

The on clause

$$\textit{process} = \textbf{on}\ \langle\text{expression}\rangle\ \textbf{do}\ \langle\text{process}\rangle$$

The *on clause* provides a way to distribute execution. Let p be a value and P be a process then

$$\texttt{\textbf{on}}\ e\ \texttt{\textbf{do}}\ P$$

causes P to be executed remotely on processor p. The process terminates when P terminates and any variables changed by P are updated locally with the same effect as the local execution of P:

$$[\![P]\!]\ =\ [\![\texttt{\textbf{on}}\ e\ \texttt{\textbf{do}}\ P]\!]$$

The transformation

Processor allocation follows a simple scheme where each process in a parallel command is allocated to a processor and each process array is allocated to a contiguous block of processors. This allows each component process to be addressed with the processor ID corresponding to the *base* address of the array with an *offset* into it.

Let $|P|$ denote the number of component processes of an array P, such that for a single process Q, $|Q| = 1$, $|[i = b$ **for** c **step** $s]\ Q| = c$ and $|[i = b_1$ **for** c_1 **step** $s_1, i = b_2$ **for** c_2 **step** $s_2]Q| = c_1 \times c_2$ etc. where b, c and s are integer values (for an implementation of sire that permits dynamically-sized replicators, c would be an expression and the proposed approach would work in the same way). Taking X to be the input process P, b to be the *base processor* variable and i be a *process ID*, then the algorithm to allocate processors (and at the same time assign IDs to each process for use in later stages) proceeds in the following way.

(1) Set b to 0, i to 0 and allocate P to processor b (allocation to this processor is implicit since execution starts at processor 0).

(2) For each command C of P:

- If C is a parallel command with n anonymous component processes $Q_1, Q_2, \cdots, Q_n$

  ```
  { $Q_1$ & $Q_2$ & $\cdots$ & $Q_n$ }
  ```

  then each component is allocated to the processors $b, b + |Q_1|, b + |Q_1| + |Q_2|, \cdots, b + |Q_1| + |Q_2| + \cdots + |Q_{n-1}|$ and to have the IDs $i, i + 1, \cdots, i + n$ respectively. They are then rewritten with on clauses to perform the distribution:

  ```
  { $Q_1$
  & on $b + |Q_1|$ do $Q_2$
  & on $b + |Q_1| + |Q_2|$ do $Q_3$
  & $\cdots$
  & on $b + |Q_1| + |Q_2| + \cdots + |Q_{n-1}|$ do $Q_n$ }
  ```

  With named component processes, the on clause is inserted between the name label and process.
- Set b to $b + |Q_1| + |Q_2| + \cdots + |Q_n|$, i to $i + n$ and apply the allocation (starting from step 2) recursively to each component process, each time updating the values of b and i.

(3) For each remaining procedure or function in X, assign it a unique ID $i' > i$.

Process mappings

After this transformation, each process is assigned a unique identifier and can be associated with a particular processor. The following notation is used to denote these mappings; they represent the lookups that would be performed on internal compiler data structures.

Definition 8.1 (Process ID). The function $\mathrm{procid}(n)$ returns an integer value at compile time that uniquely identifies the process or process array name, n, from all other processes in the program.

Then, for any component processes p or process arrays q of X, $\mathrm{procid}(p) \neq \mathrm{procid}(q)$.

Definition 8.2 (Process location). The function $\mathrm{location}(n)$ returns the processor allocated to the process with name n.

Then, each process p is allocated to the processor $\mathrm{location}(p)$ and each process array q is allocated to the block of processors starting at $\mathrm{location}(q)$.

The processor allocation for components of process arrays is based on a linear combination of the replicator indices, defined by the following function that produces an expression consisting only of constants or replicator indices. This is used to transform parallel replicators in stage 8.

Definition 8.3 (Process array parameters). The functions $\mathrm{dims}(x)$ and $\mathrm{bases}(x)$ return the dimensions and bases respectively of each index range of a process array x. Let I_x be an index range $[\![i_x = b_x\ \texttt{for}\ c_x]\!]$, where i_x is a name and b_x and c_x are integer values, P be a process and

$$p = [\![\texttt{par [}I_1\texttt{, }I_2\texttt{, }\cdots\texttt{, }I_d\texttt{] }P]\!]$$

then $\text{dims}(p) = \{c_1, c_2, \cdots, c_n\}$ and $\text{bases}(p) = \{b_1, b_2, \cdots, b_n\}$.

Definition 8.4 (Component process offset)**.** A component process of a d-dimensional array p where $d \geq 1$ with dimensions of length $\ell_1, \ell_2, \cdots, \ell_d$ is allocated to the processor

$$\text{location}(p) + \text{index}(p, e_1, e_2, \cdots, e_d)$$

where $e_1, e_2, \cdots, e_d$ are the subscript expressions selecting the component and where

$$\text{index}(p, e_1, e_2, \cdots, e_d) = \sum_{i=2}^{d} \left((e_i - b_i) \prod_{j=i}^{d} l_j \right) + (e_1 - b_1)$$

where $\{\ell_1, \ell_2, \cdots, \ell_d\} = \text{dims}(n)$ and $\{b_1, b_2, \cdots, b_d\} = \text{bases}(n)$. For processes that are not a component of an array, when $d = 0$, then index() has the value -1. This is used to discern component processes.

As an example, given a 3-dimensional array a where $\text{dims}(a) = \{4, 3, 2\}$ and $\text{bases}(a) = \{0, 0, 0\}$, then the component with indices (i_3, i_2, i_1) has the offset $i_1 + (2 \times i_1) + (3 \times 2 \times i_3)$.

Definition 8.5 (Interface component ID)**.** The function $\text{intid}(n.c)$ returns an integer value that uniquely identifies the channel-end c within the interface of the process n.

Then, for any components $n.c$ and $n.d$ of the same interface, $\text{intid}(n.c) \neq \text{intid}(n.d)$.

Example

Process 8.3 shows the server transformation applied to Process 8.2, unchanged parts of the example are omitted for brevity. Table 8.1 shows the values of procid(), location() and index() for to each process, where 'main' is the anonymous client process. This corresponds to an internal data structure that would be maintained by the compiler.

Process	procid()	location()	index()	**Interface mappings**
`m`	0	0	-	$\text{intid}(m.in) = 0$, $\text{intid}(m.out) = 1$, $\text{intid}(m.x) = 2$, $\text{intid}(m.end) = 3$
`p`	1	1	i	$\text{intid}(p.in) = 0$, $\text{intid}(p.out) = 1$, $\text{intid}(p.end) = 2$
`p[0]`	-	1	0	-
`p[1]`	-	2	1	-
`p[2]`	-	3	2	-
`p[3]`	-	4	3	-
main	2	5	-	-

Table 8.1.: Values associated with each process in Process 8.3 after transformation 3.

```
{ m is interface (chanend in, out, call x(var v), end()) to
    ...
& p is on 1 do par [i=0 for 4]
    ... ||
& on 5 do
    ... }
```

Process 8.3: Transformation example: stage 3, process distribution and processor allocation.

8.2.4. Transformation 4: rewrite interfaces

In this stage, interfaces in X are rewritten to specify an *absolute reference* of the executing process, using '@' for syntax. An absolute reference specifies for a process, its location, its ID and index if it is a component of an array. For an interface, this reference is used to generate code to *publish* the channel-end IDs, to make them available to remote processors. The following two transformation stages rewrite server calls and connection targets using the same syntax. This is used to generate connections that make requests for remote channel-end IDs.

The details of publishing and requesting channel ends is explained in more detail in Sections 8.4.4 and 8.4.5.

The transformation

$$\begin{aligned} \textit{absolute-reference} &= \texttt{(}\ \langle\text{expression}\rangle\ \texttt{:}\ \langle\text{expression}\rangle\ \texttt{:}\ \langle\text{expression}\rangle\ \texttt{)} \\ \textit{interface} &= \texttt{interface}\ \texttt{(}\ \{_0\ \texttt{,}\ \langle\text{declaration}\rangle\ \}\ \texttt{)}\ \texttt{@}\ \langle\text{absolute-reference}\rangle \end{aligned}$$

Let R_d for $d > 0$ be a parallel replicator $[\![\textbf{par}\ [I_1, I_2, \cdots, I_d]]\!]$, where I_x is an index range $[\![i_x = b_x\ \texttt{for}\ c_x]\!]$, i_x is a name and b_x and c_x are integer values, or for $d = 0$ be null. Let S be a sequence and $s_1, s_2, \cdots, s_n$ be interface specifiers, then a process p in X

$$R_d\ \texttt{interface}\ \texttt{(}s_1, s_2, \cdots, s_n\texttt{):}\ S$$

is rewritten as

$$R_d\ \texttt{interface(}s_1, s_2, \cdots, s_n\texttt{)@(}\text{location}(p)\texttt{:}\text{procid}(p)\texttt{:}\text{index}(p, i_1, i_2, \cdots, i_d)\texttt{):}\ S$$

Example

Process 8.4 shows the server transformation applied to Process 8.2, unchanged parts of the example are omitted for brevity.

```
{ interface (chanend in, out, call x(var v), end())@(0:0:-1):
    ...
& on 1 do par [i=0 for 4]
    interface (chanend in, out, call end())@(1:1:i):
    ...
& on 5 do
    interface(chanend m, p)@(2:5:-1):
      ... }
```

Process 8.4: Transformation example: stage 4, rewritten interfaces to specify an absolute reference to its location.

8.2.5. Transformation 5: rewrite remote calls

In this stage, remote calls in X are rewritten as outputs on local channel ends and additional details of the sending process are included to identify itself to the server. Remote names are substituted with new local channel-end names and connections are inserted to connect the local name to the remote name for each *distinct* use of the name, i.e. those with different subscripts or with a different call.

The transformation

$$\begin{aligned} \textit{server-call} &= \langle\text{local-chanend}\rangle\ \texttt{!}\ \langle\text{element}\rangle\ \texttt{(}\ \{_0\ \texttt{,}\ \langle\text{actual}\rangle\ \}\ \texttt{)} \\ \textit{local-chanend} &= \langle\text{element}\rangle\ \texttt{@}\ \langle\text{absolute-reference}\rangle \end{aligned}$$

Let R_d for $d > 0$ be a parallel replicator $[\![\textbf{par}\ [I_1, I_2, \cdots, I_d]]\!]$, where I_x is an index range $[\![i_x = b_x\ \texttt{for}\ c_x]\!]$, i_x is a name and b_x and c_x are integer values, or for $d = 0$ the index range is null. Let S be a sequence that contains calls to processes with process names $N_1, N_2, \cdots, N_n$ for $n \geq 0$ and $s_1, s_2, \cdots, s_m$ for $m \geq 0$ be specifications of interface components. Then, a process p in X

R_d **interface**$(s_1, s_2, \cdots, s_m)$@$(x$:y:$z)$: S

is rewritten with additional local channel ends with names $c_1, c_2, \cdots, c_n$ that are unique to their scope as

R_d **interface**$(s_1, s_2, \cdots, s_m$, **chanend** $c_1, c_2, \cdots, c_n)$@$(x$:y:$z)$: S

and each call in S

$N_j.x(a_1, a_2, \cdots, a_k)$

is rewritten with a preceding connect command[1] and a local channel end that specifies the processor location and process ID of the executing process

connect N_j **to** $N_j.x$;
  c_j@(location(p):procid(p):index$(i_1, i_2, \cdots, i_d)$) ! $x(a_1, a_2, \cdots, a_k)$

Example

Process 8.5 shows the server transformation applied to Process 8.4.

```
{ m is interface (chanend in, out, call x(var v), end())@(0:0:-1):
    ...
& p is par [i=0 for 4]
    ...
& { interface(chanend m, p)@(1:1:-1):
      var v:
      connect p to m.x; m@(5:2:-1) ! x(v);
      connect m to m.end; m@(5:2:-1) ! end();
      seq [i=0 for 4]
      { connect p to @(1:1:i).2;
        p@(5:2:0) ! end() } } }
```

Process 8.5: Transformation example: stage 5, rewriting of remote calls and insertion of implicit server connections.

[1]By doing this every call is always preceded by a connection. The reason is to simplify the explanation of the transformation but it is unnecessary since in a sequence of calls the channel will already be connected. Therefore, for any call contained in a sequence of commands, a connect command should be inserted, if there is not already one, at the lowest position in the sequence such that no command between the connection and call:

- contains a replicator that defines an index used in a subscript for the call;
- contains another call to a different server in the same array or to a different call.

8.2.6. Transformation 6: rewrite connection targets

In this stage, remote channel ends in X are rewritten as an *absolute reference* to a specific channel end or call, on a specific processor, based on the allocation of processes to processors in stage 3. Since a subscript expression can consist only of constants or replicator indices, its value can be computed and the target process can be determined.

The transformation

$$\begin{aligned} \textit{chanend} &= \langle\text{remote-chanend}\rangle \\ \textit{remote-chanend} &= \texttt{@}\ \langle\text{absolute-reference}\rangle\ \texttt{.}\ \langle\text{expression}\rangle \end{aligned}$$

Remote channel ends and remote calls are rewritten to specify the processor, process ID, process index (if it is a component of an array) and interface component index. When $\text{index}()$ is used to produce an expression, an implicit mapping is assumed between the form of the mathematical expression and a corresponding sire expression.

Let p be a d-dimensional array for $d \geq 0$ and $e_1, e_2, \cdots, e_d$ be subscript expressions, then in X, a remote channel end of a component of this array that appears in a connect command

connect c **to** N.c

where N is a n name ($d = 0$) or subscripted name $n[i_1][i_2]\cdots[i_d]$ ($d > 0$), is rewritten as an absolute reference with a single subscript index

connect c **to** @($\text{location}(n)$:$\text{procid}(n)$:$\text{index}(n, i_1, i_2, \cdots, i_d)$).$\text{intid}(n.c)$

Example

Process 8.6 shows the server transformation applied to Process 8.3.

8.2.7. Transformation 7: contract program

This stage performs the opposite transformation to the program expansion in stage 2 by rewriting processes in X that are prefixed with an on clause or replicator as an instance of a procedure. This is to simplify the implementation of process distribution since a process type specifies the complete environment with the set of formal parameters, which correspond to its *free variables*.

Definition 8.6 (Free variables)**.** Let P be a process then the function $\text{free}(P)$ returns a set of names that are free in P, this includes the names of procedures and functions.

The transformation

Let P be a process, $\text{free}(P) = \{a_1, a_2, \cdots, a_n\}$ and $f_1, f_2, \cdots, f_n$ be formal parameters corresponding to the type of each a_i. Variables that are not changed by assignment or input and replicator indices are of type **val**. Then, each process in X prefixed by an on clause

on e P

is rewritten as

on e N(a_1, a_2, $\cdots$, a_n)

and the definition

process N (f_1, f_2, $\cdots$, f_n) **is** P

```
{ interface (chanend in, out, call x(var v), end())@(0:0:-1):
  { connect out to @(1:1:0).0;
    connect in to @(1:1:3).1;
    ... }
& on 1 do par [i=0 for 4]
    interface (chanend in, out, call end())@(1:1:i):
    { if i = 0
      then connect in to @(0:0:0).1
      else connect out to @(1:1:i+1).0;
      if i = N-1
      then connect in to @(1:1:i-1).1
      else connect out to @(0:0:0).0;
      ... }
& on 5 do
    interface(chanend m, p)@(2:5:-1):
    { var v:
      connect p to @(0:0:-1).2; m@(2:5:-1) ! x(v);
      connect m to @(0:0:-1).3; m@(2:5:-1) ! end();
      seq [i=0 for 4]
      { connect p to @1:1[i].2;
        p@(2:5:0) ! end() } } }
```

Process 8.6: Transformation example: stage 6, rewriting of remote channel ends to specify an absolute reference to the target process.

is introduced where N is not specified by any other definition. Let I_x is an index range $[\![i_x = b_x$ **for** $c_x]\!]$, i_x is a name and b_x and c_x are integer values, then each replicated process in X

par $[I_1, I_2, \cdots, I_d]$ P

where $d \geq 1$ is rewritten as

par $[I_1, I_2, \cdots, I_d]$ $N(a_1,\ a_2,\ \cdots,\ a_n)$

and the definition

process N $(f_1,\ f_2,\ \cdots,\ f_n)$ **is** P

is introduced where N is not specified by any other definition.

Example

Process 8.7 shows the server transformation applied to Process 8.6.

8.2.8. Transformation 8: rewrite replicators

In this stage, replicators in X are implemented by transforming them into processes that use recursion to allocate processors in parallel using the on clause to distribute execution.

Consider Process 8.8 below, which is based on an example proposed by May [May99]:

```
process d(val t, n) is
  if n = 1 then p(t)
  else { d(t, n/2) & on t+(n/2) do d(t+(n/2), n/2) }
```

Process 8.8: A process that employs parallel recursion to distribute execution.

```
process P1() is
  par [i=0 for 4] P3(i)

process P3(val i) is
  interface (chanend in, out, call end())@(1:1:i):
    ... :

process P2() is
  interface(chanend m, p)@(2:5:0):
    ... :

{ interface (chanend in, out, call x(var v))@(0:0:-1):
    ...
& on 1 do P1()
& on 5 do P2() }
```

Process 8.7: Transformation example: stage 7, program contraction with the creation of procedure instances.

It works by offloading a copy of itself to a remote processor each time it recurses. These offloaded processes then, themselves, continue this behaviour. Each level of recursion sees a doubling of the capacity to initiate computations and this follows the structure of a binary tree. The parameter `t` is the distribution *base* and `n` is the number of processes to distribute or the *interval*. At each stage of the recursion, `t` is moved to the middle of the interval and `n` is halved. When each instance of `d` executes with its parameter `n` equal to 1, it halts the recursion and executes the process `p` and `t` then indicates the index of each process.

As an example, the execution of `d(0, 8)`, which distributes `p` over 8 processors, $p_0, p_1, ..., p_7$, is illustrated by Table 8.2. The table shows at each time step the state of each processor, given by the process it is executing. After 4 steps in the recursion, each processor executes the process `p`.

Step	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7
0	d(0,8)							
1	d(0,4)				d(4,4)			
2	d(0,2)		d(2,2)		d(4,2)		d(6,2)	
3	d(0,1)	d(1,1)	d(2,1)	d(3,1)	d(4,1)	d(5,1)	d(6,1)	d(7,1)
4	p(0)	p(1)	p(2)	p(3)	p(4)	p(5)	p(6)	p(7)

Table 8.2.: Illustration of the behaviour of Process 8.8 to distribute the execution of the process `p`.

Generalised distribution of a replicated process

Process 8.8 can be generalised to distribute non-powers-of-two process arrays to arbitrary processor ranges, for replicators with multiple index ranges, and for processes with arbitrary parameters.

To distribute arbitrary sized process arrays, an additional parameter `m` can be added to record how much of the interval needs to be distributed and the recursion can be terminated when it exceeds this. Process 8.9 is a modified version of Process 8.8 that distributes the process `p` over the processors $p_{\mathtt{t}}, p_{\mathtt{t}+1}, \cdots, p_{\mathtt{t}+\mathtt{m}-1}$ where `n` is the next power-of-two larger than `m`.

For multidimensional replicators, the value of each replicator index can be produced from single array index by an arithmetic mapping.

Definition 8.7 (Replicator index map)**.** A d-dimensional array n with dimensions $\ell_1, \ell_2, \cdots, \ell_d$ contains $N = \prod_{i=1}^{d} \ell_i$ components. Let i be a component index in the range 0 to N, then the j^{th}

```
process d(val t, n, m, b) is
  if
  { n = 0: p(t)
  | m > n/2:
    { on b+t+(n/2) do d(t+(n/2), n/2, m-n/2, b) &
      d(t, n/2, n/2) }
  | m <= n/2: d(t, n/2, m, b) }
```

Process 8.9: A generalised version of Process 8.8 that distributes arbitrary sized arrays over arbitrary processor ranges.

replicator index for process i is

$$\mathrm{ri}(n,i,j) = b_j + \left(\frac{1}{i}\prod_{k=1}^{j-1} \ell_i\right) \bmod \ell_j$$

where $\{\ell_1, \ell_2, \cdots, \ell_d\} = \mathrm{dims}(n)$. and $\{b_1, b_2, \cdots, b_d\} = \mathrm{bases}(n)$.

As an example, given a 3-dimensional array a where $\mathrm{dims}(a) = \{4,3,2\}$ and $\mathrm{bases}(a) = \{0,0,0\}$, then the i^{th} component has the replicator index values $i \bmod 2$, $\frac{i}{3} \bmod 3$ and $\frac{i}{4\times 3} \bmod 4$. Since $\mathrm{ri}()$ maps a one-dimensional index into a multidimensional index and $\mathrm{index}()$ performs the opposite mapping, then for a d-dimensional array n they have the relationship

$$i = \mathrm{index}(n, \mathrm{ri}(n,i,1), \mathrm{ri}(n,i,2), \cdots, \mathrm{ri}(n,i,d)).$$

When $\mathrm{ri}()$ is used to produce an expression, in the following transformations an implicit mapping is assumed between the form of the mathematical expression and a corresponding sire expression.

Lastly, arbitrary sets of parameters can be appended to those of the distributing process and passed between instances.

The transformation

Let $f_1, f_2, \cdots, f_n$ be formals; $a_1, a_2, \cdots, a_n$ be actuals; $b_1, b_2, \cdots, b_d$ and $c_1, c_2, \cdots, c_d$ be integer values; $i_1, i_2, \cdots, i_d$ be names; P be a process; and $N(f_1, f_2, \cdots, f_n)\ P$ be the definition of a process with the name N. Then, in the scope of N, the d-dimensional replicator

par [$i_1 = b_1$ **for** c_1, $i_2 = b_2$ **for** c_2, $\cdots$, $i_d = b_d$ **for** c_d] $N(a_1, a_2, \cdots, a_n)$

which will be referred to as r, is rewritten as

$D(b,\ N,\ M)$

where $N = \prod_{j=1}^{n} c_j$ and $M = 2^j$ such that $2^{j-1} < N \leq 2^j$ and the definition

```
process $D$(val t, n, m, b,
    $f'_1$, $f'_2$, $\cdots$, $f'_m$) is
  if
  { n = 0: $N$($a'_1$, $a'_2$, $\cdots$, $a'_n$)
  | m > n/2:
    { on b+t+(n/2) do
        $D$(t+(n/2), n/2, m-n/2,
          $\mathrm{n}(f'_1)$, $\mathrm{n}(f'_2)$, $\cdots$, $\mathrm{n}(f'_m)$) &
      $D$(t, n/2, n/2,
        $\mathrm{n}(f'_1)$, $\mathrm{n}(f'_2)$, $\cdots$, $\mathrm{n}(f'_m)$) }
  | m <= n/2: $D$(t, n/2, m,
      $\mathrm{n}(f'_1)$, $\mathrm{n}(f'_2)$, $\cdots$, $\mathrm{n}(f'_m)$) }
```

is introduced where D is not specified by any other definition, the ordered set

$$\{f'_1, f'_2, \cdots, f'_m\} = \{x \mid x \in \{f_1, f_2, \cdots, f_n\}, x \text{ is not a replicator index}\}$$

which consists the formal parameters that are not replicator indices, $\mathrm{n}(f_i)$ is the name of the formal parameter f_i and $a'_1, a'_2, \cdots, a'_m$ are the actuals parameters of an instance of N where replicator indices are substituted with an expression converting from the index t, defined as

$$a'_j = \begin{cases} \mathrm{n}(f_k) & \text{if } f_k \text{ is not a replicator index } (\forall \ell, \mathrm{n}(f_k) \neq i_\ell) \\ ri(r, \mathtt{t}, \ell), & \text{if } f_k \text{ is the } \ell^{\text{th}} \text{ replicator index } (\mathrm{n}(f_k) = i_\ell \text{ for some } \ell). \end{cases}$$

Example

Process 8.10 shows the server transformation applied to Process 8.7.

```
process P1() is
  P4(1, 4, 4)

process P4(val t, n, m) is
{ var x: x := n / 2;
  if
  { n = 0: P3(t)
  | m > x:
    { on (1 + (t + x)) do P1(t + x, x, m - x) & P4(t, x, x) }
  | m <= x: P4(t, x, x) } }

process P3(val i) is
  interface (chanend in, out, call end())@(1:1:i):
  { if i = 0
    then connect in to @(0:0:0).1
    else connect out to @(1:1:i+1).0;
    if i = N-1
    then connect in to @(1:1:i-1).1
    else connect out to @(0:0:0).0;
    var continue: continue := true;
    while continue do
    alt
    { accept end():
        continue := false
    | var v: in ? v:
        out ! v+1 } }:

process P2() is
  interface(chanend m, p)@(2:5:-1):
  { var v:
    connect p to @(0:0:-1).2; m@(2:5:-1) ! x(v);
    connect m to @(0:0:-1).3; m@(2:5:-1) ! end();
    seq [i=0 for 4]
    { connect p to @(1:1:i).2;
      p@(2:5:0) ! end() } }:

{ interface (chanend in, out, call x(var v))@(0:0:-1):
  { var continue: continue := true;
    while continue do
    alt
    { accept end():
        continue := false
    | accept x(var v):
      { out ! 0;
        in ? v } } }
& on 1 do P1()
& on 5 do P2() }
```

Process 8.10: Transformation example: stage 8, transformation of replicators to a recursive form and the output program in canonical form.

8.3. Machine target

The generation of executable code for sire programs is described using the *XMOS XS1 architecture* [May09] as a target. There are a number of important reasons why targeting a real architecture is a good way to present this description.

1. The semantics of the instruction set are well defined (see [May09] for the definition). A description of the compilation process with a new instruction set or set of subroutines would require a complete specification of its semantics and, if it were not implemented, a belief that it would be capable of an efficient implementation.
2. A functioning implementation of the sire compiler and run-time kernel has been developed for the XS1 architecture. This provides a proof-of-concept and confidence that the proposal is practical. Moreover, the experience gained in doing this was an integral part of the development of the language.
3. The specification of XS1 implementations and measurements of their silicon area and performance provide a basis for a hypothetical implementation of a system employing these processors, in order to obtain an associated performance model, which is introduced in Chapter 9. The model is used in conjunction with a software simulation to evaluate the sire language on systems that employ large numbers of processors.
4. The sire notations are capable of being compiled into short sequences of operations and it is important that this is demonstrated.

Even with these advantages, if the XS1 architecture was a radical departure from contemporary parallel systems then a compilation process targeting it would not generalise to different processor architectures. This however is not the case. The XS1 architecture provides a simple set of operations in the ISA for communication, threading and synchronisation and, although the architecture is unconventional with the way that these are integrated in the ISA, they capture the essential aspects of a message-passing parallel computer.

For an implementation of the sire language targeting a different architecture, one way to retarget the compilation process in this chapter would be to translate each operation into an equivalent one provided by the target, or if there was no such operation, to implement it as a routine as part of a small run-time kernel. It is likely that for most architectures, an efficient implementation of these primitive operations would be possible since the behaviour of each one has been chosen such that it can be implemented efficiently in hardware and executed in a single cycle.[2]

8.3.1. The XS1 architecture

An XS1 *tile* contains a memory and a processor core. The core has hardware resources that support the simultaneous execution of a set of *threads*, including a scheduler and mechanisms for synchronisation and locking. It also has a set of *channel-end resources* that are the physical end-points of communication channels. These are multiplexed onto physical links that connect directly to a switch, connecting the tile to the network. Threads on the same core have symmetric access to the memory and can use channel ends to communicate locally on the same processor, or remotely, between different processors. Figure 8.3 shows a block diagram of these main components of an XS1 tile.

8.3.2. Presentation of instruction listings

The remainder of this section is presented in terms of XS1 assembly. There are two reasons for this, first, to specify precisely the process of compilation and, second, to demonstrate that all features

[2]For instance, the implementation of occam-π, which extended occam 2, targeted the conventional x86 architecture and it was noted that the implementation of occam primitives for concurrency were very lightweight when compared to similar constructs in conventional languages [WB05].

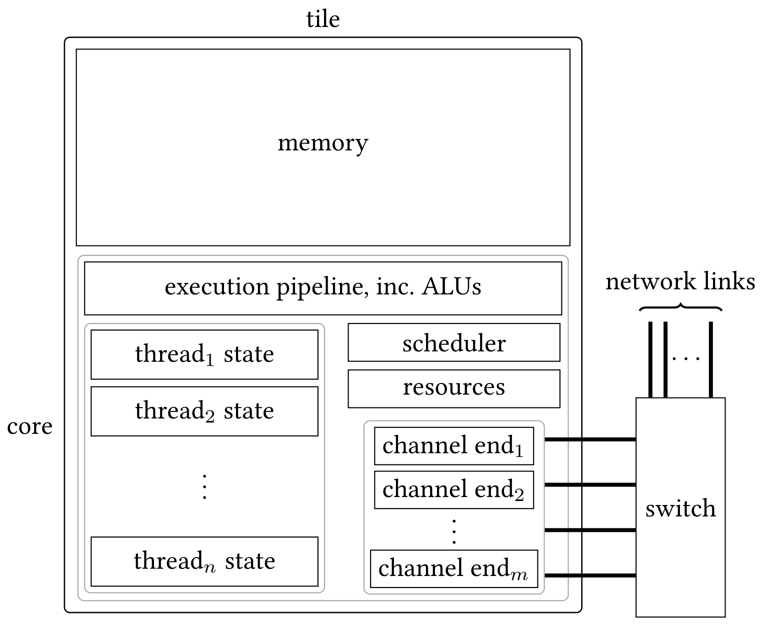


Figure 8.3.: A block diagram of the XS1 architecture. A tile consists of a processor core and memory. The processor core has a set of thread, channel end, synchroniser and lock *resources*.

of sire correspond to short sequences (i.e. tens of instructions). A simplified instruction set is used, which is described in Appendix B, but the reader will need to be familiar with the architecture and instruction set. They are referred to the architectural definition for full details [May09] and as a companion for this section.

Some additional notation is used to further simplify the assembly listings.

- Instruction sequences take two formats that are used interchangeably:
  - annotated assembly listings are used when sequences of instructions can be written concisely or special attention must be paid to the instructions;
  - numbered sequences of steps described in prose are used when instruction sequences are dominated by memory accesses and management of repetitions or control flow.
- The management of available registers, the allocation of stack storage for spilling, implementation of the calling convention and generation of arithmetic expressions are assumed to be performed by the compiler by inserting additional instructions. This is to give clarity to the behaviour of the sequences. The following notation provides this convenience:
  - mathematical variables are used in the following fragments of sire and instruction listings, in the place of channel ends, values and expressions;
  - let N be a label and $a_1, a_2, \cdots, a_n$ be variables, then the notation

    BL $N(a_1, a_2, \cdots, a_n)$

    denotes a procedure call to the address specified by N with parameters a_i passed according to the calling convention (defined below in §8.3.3 [p. 152]). Let t and i be integer values, then the notation

    BLT $t[i](a_1, a_2, \cdots, a_n)$

    similarly denotes a procedure call but at the memory address $t[i]$.
  - the notation $(f_1, f_2, \cdots, f_n)$ used at the beginning of a procedure definition, where f_i are variables, denotes the variable names used to represent the parameters that are passed in registers and on the stack according to the calling convention.
- The notation $x[i]$ denotes a memory address at byte address x with word offset i.

- The notation $x[i \cdots j]$ denotes a range of memory address $x[i], x[i+1], \cdots, x[j]$.
- The notation $x \leftarrow y$ denotes the value y being written to the storage location x, which could be a register or memory location
- The symbol 'Bpw' is the number of bytes per word and 'bpw' is the number of bits per word.

The following functions are also used. The first two correspond to lookups that a compiler would perform on its internal data structures.

Definition 8.8 (Array size)**.** The function '$\text{size}(x)$' returns the total size if an array. If x is the name of a d-dimensional array variable with dimensions of length $\ell_1, \ell_2, \cdots, \ell_d$ then $\text{size}(v) = \prod_{i=1}^{d} l_i$, is the total size of x.

Definition 8.9 (Label)**.** The function '$\text{label}(x)$' returns a label assigned to the compiled element x, such as a procedure or function.

Definition 8.10 (Channel end ID)**.** The function '$\text{chanid}(x, i)$' returns the channel-end ID with node ID x and resource index i by computing the value of the expression $(x \ll 16) \vee (i \ll 8) \vee$ `CHAN`, which corresponds to the format of a resource ID (see Appendix B.4).

Lastly, all communication in the run-time kernel is synchronised according to the sequences described in Appendix B; communication sequences that do not follow this are listed explicitly.

8.3.3. Calling convention

It is necessary at this point to define a *calling convention* since this is used throughout the remaining sections. A calling convention specifies the interactions between a *calling* process and a *callee* process, which occur through a set of parameter registers and the stack. Execution of a process activates a *stack frame* that is used to pass parameters, store saved registers and store variables local to the scope of the process.

The following describes a simple calling convention based on the XMOS *Application Binary Interface* [OW10, §2.6]. It is sufficient to implement procedure calling in sire and is used in the description of the run-time kernel and code generation.

Let P be a process, $f_1, f_2, \cdots, f_n$ be formal parameters and $a_1, a_2, \cdots a_n$ be actual parameters, then the procedure

process N(f_1, f_2, $\cdots$, f_n) **is** P

is compiled as the caller in the sequence:

(1) save the *lr* in *sp*[0] of frame before the *sp* is extended by n words (`ENTSP` n);

(2) push m required general purpose registers used in the execution of P in the set $\{r4, \cdots, r10\}$ onto stack from *sp*[p] to *sp*[$p+m-1$] where p is the offset of the preserved registers storage in the stack frame;

(3) the instructions of P;

(4) pop m saved general purpose registers back from *sp*[p] to *sp*[$p+m-1$] into $\{r4, \cdots, r10\}$;

(5) contract the stack by n words and restore the *pc* from the *lr* (`RETSP` n).

An instance of the procedure N

N(a_1, a_2, $\cdots$, a_n)

is compiled as the caller with the sequence:

(1) write the parameters $a_1, \cdots, a_4$ in the registers $r0$ to $r3$ respectively;

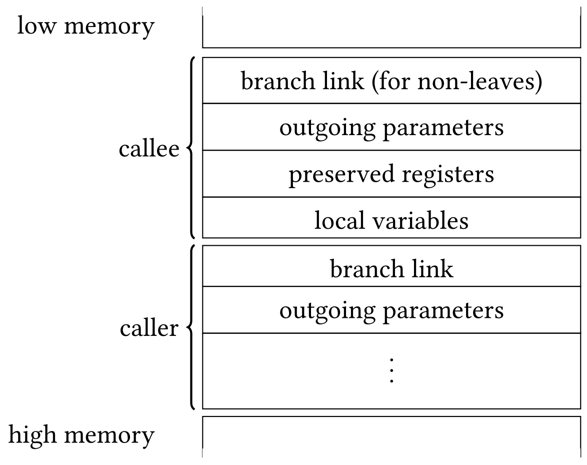


Figure 8.4.: Example stack frame layout illustrating the calling convention with a caller and callee. A stack frame provides space during the lifetime of a procedure call to store *lr* (for procedures that call other procedures), to pass parameters and for additional storage to the registers.

(2) write the remaining parameters a_{4+i} to *sp*$[1 + i]$;

(3) call the procedure and wait for it to return (`BL` $\text{label}(N)$).

This convention is summarised in Figure 8.4, which shows the layout of the adjacent caller and callee process stack frames.

8.4. Run-time kernel

8.4.1. Overview

A sire program is compiled with additional components that implement dynamic aspects of the language. They are collectively referred to as the *run-time kernel*. The kernel is divided into two components: a *service kernel* for a *service process* that executes using one thread on each processor (for the duration of the user program's execution); and a *program kernel* comprising a set of routines for the user program's processes to engage with the service processes.

The service process is responsible for initialisation of the processor it is executing on and allows:

- a remote process to execute a new process on the processor (the remaining threads on each processor that are used to execute program processes are called the *worker* threads);
- a remote process to obtain channel-end IDs that have been allocated on the processor by particular processes;
- a local process to *publish* a local channel-end ID it has been allocated.

The remainder of this section describes the service and program kernel routines. The following figures and tables summarise various aspects of these and can be referred to while reading.

- Figure 8.5 provides a high-level overview of the run-time kernel, illustrating the division between the service and program components, and listing the routines and the main data structures.
- Figure 8.6 shows the memory layouts of the master and slave nodes that is produced by compilation.
- Table 8.3 summarises the kernel routines.
- Table 8.4 lists the constant values used in the run-time kernel.
- Table 8.5 lists the storage requirements and labels for the components of these data structures as well as the other channel end and lock resources used by the kernel.

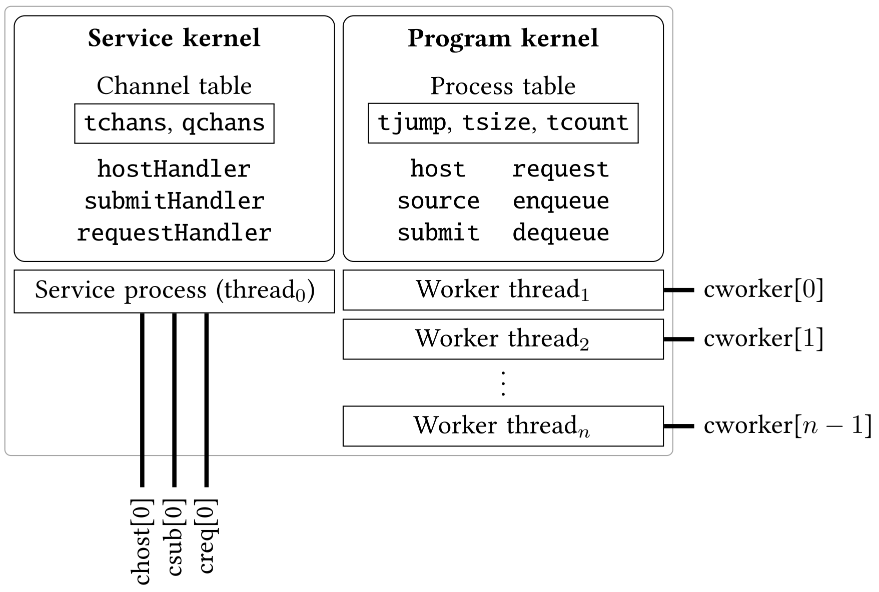


Figure 8.5.: Block diagram of the per-processor run-time kernel. The service component deals with remote requests and the program component deals with local execution of processes. Table 8.3 lists summarises the operation of the routines listed in each kernel and Table 8.5 summarises the data structures the channel and process tables (enclosed in boxes). The lines show the channel ends used to implement the run-time communication.

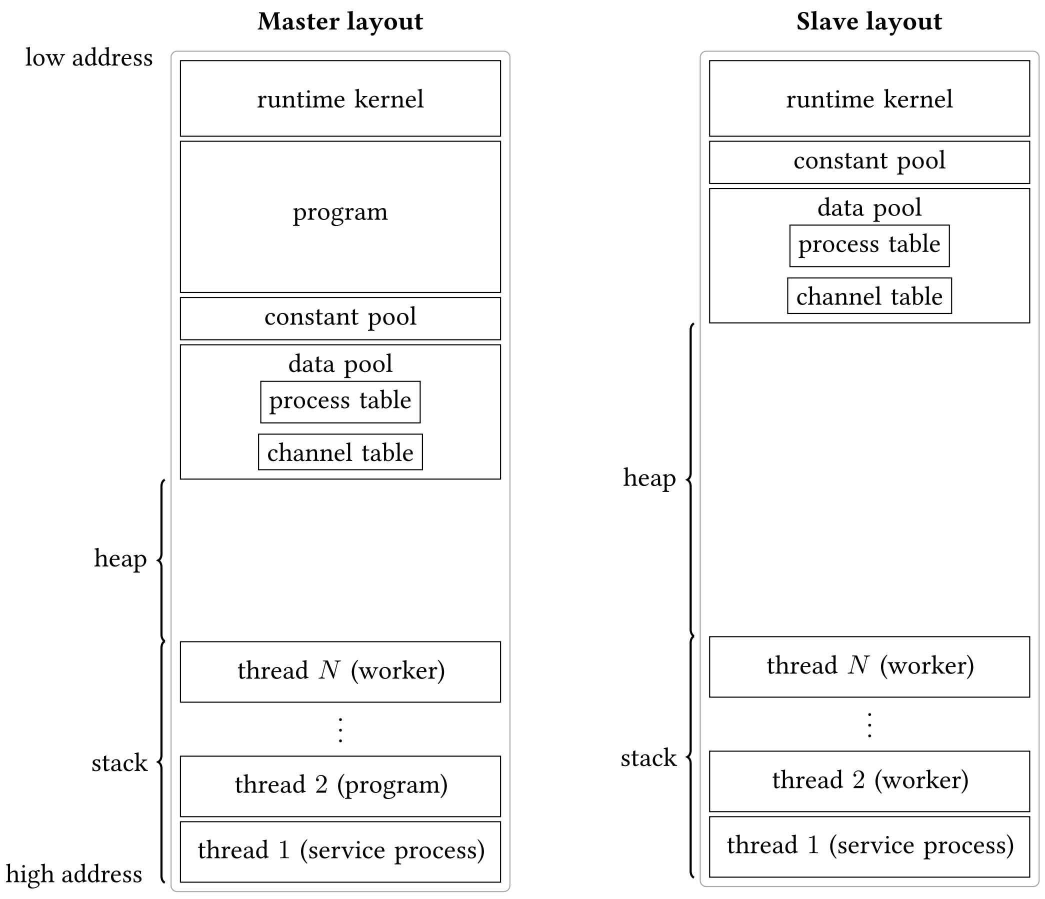


Figure 8.6.: Master and slave memory layouts produced by compilation. The master layout is executed on node 0 and contains the complete program, which starts executing on thread 2. The slave image is replicated on all other processors.

Routine	Pairing	Operation
		Service kernel
`hostHandler`	`source`	deal with a request to *host* and execute a process
`subHandler`	`submit`	deal with a local request from to submit a channel-end ID
`reqHandler`	`request`	deal with a request for a published channel-end ID
		Program kernel
`host`	`source`	host the execution of a process from a *source*
`source`	`hostHandler` `host`	cause a process to be executed remotely by a *host* processor
`submit`	`subHandler`	submit a channel-end ID the local service process to be published
`request`	`reqHandler`	request a channel-end ID from a remote service process
`enqueue`	-	enqueue a call request
`dequeue`	-	dequeue a call request

Table 8.3.: Overview of the kernel routines. A *pairing* indicates routines that communicate with each other.

Name	Description	Value
N_{threads}	number of threads per processor	architecture determined
N_{chans}	number of channels per processor	architecture determined
N_{stack}	stack space per thread	compilation determined
N_{procs}	maximum number of processes per processor	compilation determined
N_{params}	maximum number of parameters per process	compilation determined
N_{kcalls}	number of kernel calls	5
I_{host}	host jump table index/channel-end resource ID	0
I_{sub}	submit jump table index/channel-end resource ID	2
I_{req}	request jump table index/channel-end resource ID	4
I_{enq}	queue call request jump table index	6
I_{deq}	dequeue call request jump table index	7
I_{qsize}	call request queue size	8

Table 8.4.: Constant values used in the run-time kernel.

Name	Words	Description
		Resources
`chost`	1	`hostHandler` request channel-end ID
`csub`	2	`subHandler` request and service channel-end IDs
`creq`	2	`reqHandler` request and service channel-end IDs
`cworker`	$N_{\text{threads}}-1$	worker channel-end IDs for remote process creation
`lhost`	1	a lock ID to use for resource allocation in the 'host' routine
		Data structures
`tjump`	N_{procs}	a *process jump table* that maps process IDs to local addresses
`tsize`	N_{procs}	a process *size table* that records process sizes in bytes
`tcount`	N_{procs}	a process *count table* that records the number of times a process is used
`tchans`	$N_{\text{procs}}\times 4$	a *channel-end table* that records mappings of processes to local channel ends
`qchans`	$N_{\text{chans}}\times 4$	a *channel-end request queue* that records requests for local channel ends

Table 8.5.: Summary of the storage requirements per processor for the run-time kernel. These components are statically allocated in the data pool.

8.4.2. Structure and operation

The service process first initialises the system and then behaves like a server by repetitively servicing requests on three channels `chost`, `csub` and `creq` that correspond to hosting processes, publishing local channel ends and providing access to them.

Initialisation

Each service process performs the following initialisation of the system.

1. *Initialise registers.* Set the *cp*, *dp* and *sp* registers, and an entry point into the exception handler.
2. *Initialise thread registers.* Allocate all of the available threads and, for each thread, initialise the *cp*, *dp* and *sp* registers. Stack space is allocated to threads in blocks of size N_{stack}at the address

   $$\text{base } sp + N_{\text{stack}} \times \text{thread ID}$$

   where 'base *sp*' is the beginning of the stack region. Each thread is then directed to execute a short routine to set their exception handler, after which it is deallocated. Deallocation does not change any of their state, leaving them ready to be allocated as worker threads to execute processes.
3. *Allocate run-time kernel channel ends.* Allocate, in sequence, a request channel end for the `hostHandler`and request and service channel ends for the `submit` and `request` routines, and store them at `chost`[0], `creq`$[0\cdots1]$ and `csub`$[0\cdots1]$ respectively. Since no other channel ends are allocated at this point, they will have consecutive resource IDs from 0. This allows the channel-end IDs for these channels to be computed for any processor.
4. *Allocate worker channel ends.* Allocate a channel end for each of the worker threads and store them at addresses `cworker`$[0\cdots N_{\text{threads}}-1]$.
5. *Initialise the jump table.* Initialise the kernel calls in the jump table by setting `tjump`$[0\cdots2]$ to the addresses of the `source`, `submit` and `request` routines respectively. The master version containing the program must also initialise the addresses for each of the component processes. Since all of the processes are known at compile time, the entries in the master and slave jump tables can be statically initialised.
6. *Initialise the size table.* Initialise each entry in the size table `tsize`$[0\cdots N_{\text{procs}}-1]$ to 0.
7. *Initialise the request queue.* Initialise each entry in the request queue `qchans`$[0\cdots(N_{\text{chans}}\text{-}1)\times4]$ to 0.
8. *Allocate a lock.* Allocate a lock for the worker threads and store it at `lhost`[0].

Service 'loop'

The core of the run-time kernel is defined by the following sequence that initialises each of the `host`, `submit` and `request` routine channels to trigger events on the arrival of a message and the execution of the associated routine. When the handler returns, control is transferred back to the point 'serve', where events are re-enabled.

	B entry	branch to the entry point
hostHandler:	· · ·	host-handler instructions
subHandler:	· · ·	submit-handler instructions
reqHandler:	· · ·	request-handler instructions
entry:	LDW c_{host}, chost[0]	load the channel-end IDs
	LDW c_{sub}, csub[0]	
	LDW c_{req}, creq[0]	
	SETC c_{host}, MODE_EVENT	enable events on the channel ends
	SETC c_{sub}, MODE_EVENT	
	SETC c_{req}, MODE_EVENT	
	LDAW p, host	load the address of the handler routines
	LDAW p, submit	
	LDAW p, request	
	SETV c_{host}, p	set the event vectors
	SETV c_{sub}, p	
	SETV c_{req}, p	
serve:	SETSR EVENT_ENABLE	enable events
	WAITE	wait for an event

Since the behaviour of a service process is similar to that of a sire server process there is potential for it to cause deadlock, but because none of the service routines engage in any communication, it is safe for pending clients requests to block in the network. However, the extent to which they block, i.e. consume network resources, should be limited so that network congestion is minimised. This is achieved by clients sending short requests to service processes. These requests contain only the sender's channel-end ID and they close the route immediately.

To prevent any interleaving of messages from different clients (since they close the route) requests are received on a separate channel. The subHandler and reqHandler routines therefore are each permanently allocated two channels, one for requests and the other to service requests. The same effect is achieved in the hostHandler and host routines with the use of a worker channel end to service the request. These channels are stored at chost[0], csub[0], csub[1], creq[0] and creq[1] respectively.

The following sections define the behaviour and implementation of the three event handlers and the corresponding request routines for each one.

8.4.3. Hosting processes

Execution of an on clause causes the specified process to be executed on a remote *host* processor. The *source* process executing it first requests from the host processor a thread, and then communicates with that thread to transfer the required components and state, then wait to receive any updated state.

worker thread (processor s)		processor t	
source	$\rightarrow$	hostHandler	(service thread)
		host	(worker thread)

Process closures

Remote execution of a process P requires the instructions of P, any processes it uses, and its *environment* (its free variables) to be made available at the remote processor. These components are called the *closure* of the instance of P.

Since each process prefixed by an on clause is an instance of a type in the canonical program form (the effect of the program contraction transform, see §8.2.7 [p. 144]), the environment is defined explicitly by the formal parameters. The closure therefore needs only to contain the set of actual parameters to the instance of P and any nested (i.e. referenced) processes; it is defined in the following way.

Definition 8.11 (Process closure)**.** Let P be a process, $f_1, f_2, \cdots, f_n$ be formal parameters, $a_1, a_2, \cdots a_n$ be actual parameters and X be a program with a definition

process $N(f_1,\ f_2,\ \cdots,\ f_n)\ P$

then instances of N takes the form

$N(a_1,\ a_2,\ \cdots,\ a_n)$

and the *closure* of N consists of the values of the actual parameters $a_1, a_2, \cdots, a_n$ and the process P.

A process closure is *represented* by the following sequence of words, which corresponds to the communication required to transmit it between a *source* and *host*:

- the process ID;
- the size s in bytes;
- the instruction sequence in $\lceil s/\,\mathrm{Bpw}\rceil$ words.
- the number of parameters;
- for each parameter p:
  - the parameter type t (one of **val**, **var**, **array**, **process** or **function**);
  - if t =**val** or t =**var**, the value of the variable;
  - if t =**array**, the $\mathrm{size}(p)$ and the component values of p in a sequence;
  - if t =**process** or t =**function**:
    * the process ID;
    * the size s of the process code in bytes;
    * the instruction sequence in $\lceil s/\,\mathrm{Bpw}\rceil$ words.

Source routine

The `source` routine interacts first with the `hostHandler` routine to allocate a worker thread. The `hostHandler` then hands over to the worker, which executes the `host` routine to carry out the remote execution. The sequence of communication in the `source` routine therefore corresponds to those in the `hostHandler` and `host` routines.

`source:`	$(v_{\text{location}}, v_{\text{caddr}})$	host location, closure address
	`ENTSP` $N_{\text{framesize}}$	allocate a stack frame
	`GETID` i	get the (worker) thread ID
	`LDW` c, `cworker`$[i]$	load the worker channel-end ID
	`SETD` c, d	where $d = \mathrm{chanid}(v_{\text{location}}, I_{\text{host}})$ is the remote request channel end
	`OUT` c, c	send the local channel-end ID as the client identity
	`OUTCT` c, `END`	synchronise and close service process connection
	`CHKCT` c, `END`	
	`IN` c, d	receive the worker process channel-end ID
	`SETD` c, d	set this as the new destination
	`OUTCT` c, `END`	synchronise and close worker thread connection
	`CHKCT` c, `END`	

Next, the closure is transferred (according the above representation in memory). In the following sequence each consecutive read accesses a word from the next consecutive word address; i.e. `LDW` d, v_{caddr}`[0]`; `LDW` d, v_{caddr}`[1]`; `LDW` d, v_{caddr}`[2]` etc.

```
(1) read the process ID $p$ and send it;
(2) receive a flag $f$ to signal acceptance of the process;
(3) if $f = 1$ then, repeat for $j$ from 0 to tsize[$p$]:
      • read the word at address tjump[$p + i$] and send it.
(4) read the number of parameters $n$;
(5) for each $i \in \{0, 1, \cdots, n - 1\}$ read the parameter type $t$ then:
      • if $t =$ val then read the value $v$, send $t$ and send $v$;
      • if $t =$ var then read the value $v$, send $t$ and send $v$;
      • if $t =$ array then:
          • read the total array size $\ell$ and send it;
          • read a component value $v$ and send it, $\ell$ times.
      • if $t =$ process or $t =$ function:
          • read the process ID $p$ and send it;
          • receive a flag $f$ to signal acceptance of the process;
          • if $f = 1$ then: repeat for $j$ from 0 to tsize[$p$]:
              • read the word at address tjump[$p + i$] and send it.
(6) return from the routine (RETSP $N_{\text{framesize}}$).
```

Host-handler routine

The `hostHandler` routine deals with a request for the processor to execute a new process from a remote source process executing the `source` routine. It does this by creating a new asynchronous worker thread to execute the `host` routine, which deals with the remainder of the `source` interaction.

```
hostHandler:  IN c_src, c_host        receive the source channel-end ID
              SETD c_host, c_src      set the source as the local channel-end destination
              CHKCT c_host, END       } synchronise and close connection with source
              OUTCT c_host, END       }
              GETR t, THREAD          allocate an asynchronous thread
              LDAP p, host            load the address of the worker-host routine
              TSETR pc, p             set the program counter to this
              TSETR r0, c_src         set the first parameter to the source channel-end ID
              START t                 start execution of the worker thread
              B serve                 return to wait for events
```

Worker-host routine

The worker-`host` routine deals directly with the source process, interacting with the `source` routine, to receive a representation of a closure, initialise and execute the process contained in the closure and then transmit back any updated free variables. Since it makes changes to the jump and size tables, which are shared between all worker threads, a lock is used to obtain exclusive access. Table 8.6 shows the local storage allocated in the stack frame is used by the worker thread to record details of the closure.

In the following listings, routines to dynamically allocate and deallocate storage in the heap area of memory for processes and arrays are assumed.[3] It is also convenient to define a subroutine that

[3]A memory allocation routine is not described since there are many options that trade space efficiency for performance and predictability; see Wilson for a survey [WJNB95]. However, an attractive scheme would be where memory is allocated in fixed-sized blocks, because of the simplicity and because the block size can be chosen at compile time to best match the size of the (fixed-size) processes and arrays being allocated.

`recvProcess`(c):

(1) claim the lock (`LDW` ℓ, `lhost`[0]; `IN` ℓ);

(2) receive the process ID p and set `procs`[i]$\leftarrow p$;

(3) if `tsize`[p]$= 0$ then

- send the value 1 to accept the process;
- receive the process size s in bytes and update the size table (`tsize`[p]$\leftarrow s$);
- dynamically allocate $w = \lceil s/\text{Bpw}\rceil$ words of store for the process (address a);
- receive w words and write each one to $a[0]$ to $a[w-1]$ respectively;
- update the jump table with the new address (`tjump`[p]$\leftarrow a$).

(4) if `tsize`[p]> 0 then

- send the value 0 to decline process;
- increment the process count table (`tcount`[p]$\leftarrow$`tcount`[p]$+1$).

(5) release the lock (`LDW` ℓ, `lhost`[0]; `OUT` ℓ).

Process 8.11: A routine to receive a process on a channel c.

Name	Words	Description
`types`	$N_{\text{params}}/\text{Bpw}$	types of each parameter (**`val`**, **`var`**, **`array`**, **`process`** or **`function`**)
`values`	N_{params}	the values of each parameter
`sizes`	N_{params}	the size of each array parameter
`procs`	N_{procs}	the IDs of each process in the closure

Table 8.6.: Local storage allocated by a worker thread to host an incoming process.

receives the data describing a process to avoid repetition, this subroutine is listed in Process 8.11 and is also used in the compilation of alternation in §8.5.1 [p. 167].

The following sequence establishes a connection with the source process:

```
host:  (v_csrc)              source channel-end ID
       ENTSP N_framesize     allocate space on the stack where N_framesize ≥ N_params
       GETID i               get the thread ID
       LDW c, tchans[i]      load the worker thread channel-end ID
       SETD c, v_csrc        set the channel-end destination to the source
       OUT c, c              send the local channel end to the source
       OUTCT c, END        } synchronise and close the source connection
       CHKCT c, END        }
```

The source and host processes are now ready to engage in the transfer of the process closure, according to the representation of the closure.

The host first receives the process, then each of the actual parameters in sequence:

(1) call the receive process subroutine (`BL` `recvProcess`(c));

(2) receive the number of parameters N_{params} in the closure;

(3) receive each parameter, allocating storage for arrays:
repeat for i from 0 to $N_{\text{params}}-1$:

- receive the argument type and store in `types`[i];
- if `types`[i]=**`val`** then receive the value and write it to `values`[i];

- if `types`[i]=**`var`** then receive the value and write it to `values`[i];
- if `types`[i]=**`array`** then:
  - receive the array size and write it to `sizes`[i];
  - dynamically allocate `sizes`[i] words of store and write the address to `values`[i];
  - receive each element of the array and write to the allocated space.
- if `types`[i]=**`process`** or `types`[i]=**`function`** then
  - call the receive process subroutine (`BL recvProcess`(c)).

The process is then initialised and executed:

(4) release the host lock (`LDW` ℓ, `lhost`[0]; `OUT` ℓ);

(5) write the parameter values `values`[0] to `values`[3] in the registers *r*0 to *r*3 respectively;

(6) write the remaining parameters $4 + i$ to *sp*[$1 + i$];

(7) call the procedure through the jump table (`BLT tjump`[i]).

Once execution of the process has terminated, the values of referenced variables are sent back and the memory dynamically allocated to the processes and parameters is deallocated:

(8) send each variable and array variable back to the source:
repeat for i from 0 to $N_{\text{params}}-1$:
- if `types`[i]=**`var`** then send the value stored in P_i;
- if `types`[i]=**`array`** then
  - send `sizes`[i] words of the array at address `values`[i];
  - deallocate the array storage.

(9) claim the host lock (`LDW` ℓ, `lhost`[0]; `IN` ℓ);

(10) deallocate each process if it not used by any other worker:
repeat for i from 0 to $N_{\text{procs}}-1$:
- decrement the process count table (`tcount`[i]←`tcount`[i]−1);
- if `tsize`[i]= 0 then deallocate the process space at address `tjump`[i].

(11) release the host lock (`LDW` ℓ, `lhost`[0]; `OUT` ℓ);

(12) yield the worker thread (`FREET`).

8.4.4. Publishing local channel ends

Processes publish the IDs of channel ends they have allocated by submitting them to their local service process. They are then recorded in the channel table, referenced against the ID and index of the process, so they can be requested by remote processes.

worker thread (processor p)		service thread (processor p)
`submit`	$\rightarrow$	`subHandler`

Submit routine

The `submit` routine is called by a process executing on a worker thread to submit a local channel-end ID to be published in the channel-end table for the processor.

```
submit:  ($c_{\text{chanid}}$, $v_{\text{location}}$,
            $v_{\text{index}}$, $v_{\text{procid}}$)   the channel-end ID to submit and its process location,
                                                         index and ID
         ENTSP $N_{\text{framesize}}$                  allocate some space on the stack
         GETID $i$                                     get the (worker) thread ID
         LDW $c$, cworker[$i$]                         load the worker channel-end ID
         SETD $c$, $d$                                 where $d$ = chanid($e_{\text{location}}$,$I_{\text{sub}}$)
                                                       is the submit request channel-end ID
         OUT $c$, $c$                                  send the local channel-end ID as the client identity
         OUTCT $c$, END                              } synchronise and close the request connection
         CHKCT $c$, END                              }
         SETD $c$, $d$                                 where $d$ = chanid($e_{\text{location}}$,$I_{\text{sub}}+1$)
                                                       is the service channel-end ID
         OUT $c$, $e_{\text{location}}$                send the processor location
         OUT $c$, $e_{\text{procid}}$                  send the process ID
         OUT $c$, $e_{\text{index}}$                   send the process index
         OUTCT $c$, END                              } synchronise and close the service connection
         CHKCT $c$, END                              }
         RETSP $N_{\text{framesize}}$
```

Submit-handler routine

The `subHandler` is executed by the service thread to deal with the submission of channel end from local processes by the `submit` routine.

```
subHandler:  IN $c_{\text{sub}}$, $c_{\text{src}}$       receive the source channel-end ID
             SETD $c_{\text{sub}}$, $c_{\text{src}}$     set the source as the local channel-end destination
             CHKCT $c_{\text{sub}}$, END               } synchronise and close the request connection
             OUTCT $c_{\text{sub}}$, END               }
             LDW $c$, csub[1]                           load the service channel end
             SETD $c$, $c_{\text{src}}$                 set the source as the local channel-end destination
             IN $c$, $v_{\text{chanid}}$                receive the channel-end ID
             IN $c$, $v_{\text{procid}}$                receive the process ID
             IN $c$, $v_{\text{index}}$                 receive the process index
             CHKCT $c$, END                           } synchronise and close the service connection
             OUTCT $c$, END                           }
```

It then updates the channel table and completes any outstanding requests for this channel end.

(1) Update the j^{th} entry of the channel table, where $j = ((c \gg 8) \oplus \text{FF}_{\text{hex}}) \times 3$ is the channel-end resource ID:

```
    STW $v_{\text{chanid}}$, tchans[$j$]       }
    STW $v_{\text{procid}}$, tchans[$j+1$]     } update the channel-end table entry
    STW $v_{\text{index}}$, tchans[$j+2$]      }
```

(2) Complete any queued requests for this channel end:

```
    repeat for $i$ from 0 to $N_{\text{chans}}-1$:
      • if qchans[$i \times 4 + 1$]= $p$ and qchans[$i \times 4 + 2$]= $i$ then
          SETD $c_{\text{sub}}$, qchans[$i \times 4$]   set the destination to the queued requester
          OUT $c_{\text{req}}$, $c$                     send the local channel-end ID $c$
          OUTCT $c_{\text{req}}$, END                   close the outgoing channel
          STW 0, qchans[$i \times 4$]                   clear the entry
```

(3) Return to wait for events (`B` serve).

8.4.5. Remote channel-end requests

Any process can request from any processor a published local channel-end ID. It does this by submitting to the service process the process ID and index of the channel-end ID it wishes to obtain. If no channel-end ID corresponding to this criteria has been published, the service process queues the request until it is submitted by a local process, leaving the requesting process waiting.

worker thread (processor s)		service thread (processor t)
`request`	$\rightarrow$	`reqHandler`

Request routine

The `request` routine is called by a worker thread and it interacts with a remote service process (executing the `reqHandler` routine) to obtain a channel-end ID.

`request`:	$(v_{\text{location}}, v_{\text{index}}, v_{\text{procid}})$	process location, index and ID
	`ENTSP` $N_{\text{framesize}}$	allocate some space on the stack
	`GETID` i	get the (worker) thread ID
	`LDW` c, `cworker`$[i]$	load the worker channel-end ID
	`SETD` c, d	where $d = \text{chanid}(e_{\text{location}}, I_{\text{req}})$ is the request channel-end ID
	`OUT` c, c	send local channel-end ID as the client identity
	`OUTCT` c, `END`	$\left.\begin{array}{l}\\ \\ \end{array}\right\}$ synchronise and close the request connection
	`CHKCT` c, `END`	
	`SETD` c, d	where $d = \text{chanid}(e_{\text{location}}, I_{\text{req}}+1)$ is the submit service channel-end ID
	`OUT` c, e_{procid}	send the process ID
	`OUT` c, e_{index}	send the process index
	`OUT` c, v_{intid}	send the interface component ID
	`OUTCT` c, `END`	close the incoming connection
	`IN` c, d	wait to receive the remote channel-end ID
	`CHKCT` c, `END`	close the outgoing connection
	`SETD` c, d	set local channel end to remote channel-end ID
	`RETSP` $N_{\text{framesize}}$	deallocate the stack space and return

Request-handler routine

The `reqHandler` routine deals with remote processes that want to obtain a published local channel-end ID. If the channel end has already been published, the request is fulfilled immediately, otherwise it is queued and fulfilled when the channel-end ID is submitted.

`reqHandler`:	`IN` r, c_{src}	receive the source channel-end ID
	`SETD` c_{req}, c_{src}	set the source as the local channel-end destination
	`CHKCT` c_{req}, `END`	$\left.\begin{array}{l}\\ \\ \end{array}\right\}$ synchronise and close the request connection
	`OUTCT` c_{req}, `END`	
	`LDW` c, `csub`$[1]$	load the service channel end
	`SETD` c, c_{src}	set the source as the local channel-end destination
	`IN` c, v_{procid}	receive the process ID
	`IN` c, v_{index}	receive the process index
	`IN` c, v_{intid}	receive the interface component ID
	`CHKCT` c, `END`	$\left.\begin{array}{l}\\ \\ \end{array}\right\}$ synchronise and close the service connection
	`OUTCT` c, `END`	

It then looks up the local channel-end ID. If this channel end has already been submitted by a local process, it is returned, otherwise the request is queued and fulfilled by a subsequent local channel-end submission.

```
(1) Check the published local channel ends:
      repeat for $i$ from 0 to $N_{\text{chans}}-1$:
        • if tchans[$j \times 3+1$]$= v_{\text{procid}}$, tchans[$j \times 3+2$]$= v_{\text{index}}$ and tchans[$j \times 3+3$]$= v_{\text{intid}}$
          then:
            OUT $c$, tchans[$i \times 3$]     send the local channel-end ID
            OUTCT $c$, END             close connection
            B serve                    return to wait for events
(2) If the request could not be completed then queue it:
      repeat for $i$ from 0 to $N_{\text{chans}}-1$:
        • if qchans[$j \times 4$]$= 0$ then:
            STW $v_{\text{chanid}}$, qchans[$j \times 4$]      }
            STW $v_{\text{procid}}$, qchans[$j \times 4+1$]    } update request queue entry
            STW $v_{\text{index}}$, qchans[$j \times 4+2$]     }
            STW $v_{\text{intid}}$, qchans[$j \times 4+3$]     }
            B serve                              return to wait for events
```

8.4.6. Call-request queuing

The run-time provides three routines for a queue data structure; these are used in the implementation of alternative commands that contain call accepts (§8.5.1 [p. 167]). They operate on a representation of the queue that is stored locally to the alternative to provide calls to enqueue a request, dequeue a request and to return the number of queued requests.

This section describes the storage requirements and behaviour of the data structure but does not describe the details of an implementation.[4]

Queue representation

A bound on the number of clients accessing a server can be determined at compile time, but this can potentially be large when servers are shared by process arrays. To minimise the storage requirement per client, components of process arrays can be stored with one bit per process and a fixed overhead of two words per array to store the base location and base process ID. These details can be used by the server to perform a remote channel-end ID request from the client when it is ready to service the call. The channel-end ID of single client processes can be stored directly.

A call queue for an alternative can therefore be implemented with:

- 2 words per single process to store the process ID and channel-end ID;
- $2 + \lceil n/\,\text{bpw}\rceil$ words per process array, where n is the number of component processes, to store the process ID, location and outstanding component request bit set.

A single call queue can also contain requests on multiple channel ends. These can be stored with an additional 1 word per call channel end and an additional lookup.

[4]The proposal is intended to deal quickly with a large number of clients and trades some increase in lookup time for a reduction in space. There are various ways that the data structure could be implemented to make different trade-offs, which could, for example, be employed in cases where the number of clients is limited.

Call enqueue routine

The routine `enqueue` has six parameters that are passed according to the calling convention:

1. q, the address of the queue;
2. v_{ccall}, the call channel-end ID;
3. v_{cclient}, the client channel-end ID;
4. v_{procid}, the client process ID;
5. v_{location}, the client location;
6. v_{index}, the client process index.

It inserts a call request on the channel end v_{ccall} into the queue q from a single process or component of a process array.

```
enqueue($q$,$v_{\text{ccall}}$,$v_{\text{cclient}}$, $v_{\text{procid}}$, $v_{\text{location}}$, $v_{\text{index}}$):
  (1) if $v_{\text{index}} < 0$ (single process) then insert ($v_{\text{ccall}}$, $v_{\text{procid}}$, $v_{\text{cclient}}$) into the queue $q$;
  (2) if $v_{\text{index}} \geq 0$ (process array) then insert ($v_{\text{ccall}}$, $v_{\text{procid}}$, $v_{\text{location}}$ and $v_{\text{index}}$) into the queue $q$.
```

Call dequeue routine

The routine `dequeue` has six parameters that are passed according to the calling convention:

1. q, the address of the queue;
2. v_{ccall}, the call channel-end ID;
3. v_{cclient}, the client channel-end ID;
4. v_{procid}, the client process ID;
5. v_{location}, the client location;
6. v_{index}, the client process index.

It removes and returns a queued call request from the queue q.

```
dequeue($q$,$v_{\text{ccall}}$, $v_{\text{procid}}$,$v_{\text{cclient}}$, $v_{\text{location}}$, $v_{\text{index}}$):
  (1) if the queue is not empty:
        • remove a queued request $R$ from the queue $q$:
            • if it is a single process then:
                • update the variables:
                    • $v_{\text{ccall}} \leftarrow R_{\text{ccall}}$;
                    • $v_{\text{index}} \leftarrow -1$;
                    • $v_{\text{cclient}} \leftarrow R_{\text{cclient}}$.
                • and return.
            • if it is an process array then:
                • update the variables:
                    • $v_{\text{ccall}} \leftarrow R_{\text{ccall}}$;
                    • $v_{\text{procid}} \leftarrow R_{\text{procid}}$;
                    • $v_{\text{location}} \leftarrow R_{\text{location}}$;
                    • $v_{\text{index}} \leftarrow R_{\text{index}}$.
                • and return.
  (2) if the queue is empty: $v_{\text{ccall}} \leftarrow 0$.
```

8.5. Code generation

In this section, the generation of instruction sequences for communications and the constructs *alternative*, *connect*, *interface*, *on* and *parallel* are described. The remaining parts of the language can be compiled using conventional approaches.

8.5.1. Alternation

Let $A_1, A_2, \cdots, A_n$ be alternatives. Each alternative A_i is an input or call and specifies a command C_i to be performed on successful activation. Furthermore, A_i uses a channel end c_i and may be guarded by an expression g_i. Let $\{A'_1, A'_2, \cdots A'_m\}$ be a subset of $\{A_1, A_2, \cdots, A_n\}$ consisting of call accepts. Then, an alternation command

alt { A_1 | A_2 | $\cdots$ | A_n }

is compiled to the following sequence:

	`CLRE`	clear all events
	`GETR` c_{serve}, `CHANEND`	get a service channel end
`initialise`:	$\cdots$	instructions to perform initialisation
	`WAITE`	wait for an event
`alt`$_1$:	$\cdots$	
`alt`$_2$:	$\cdots$	for alternative A_i, complete input on c_i and execute C_i or complete call on c_i
	$\vdots$	
`alt`$_n$:	$\cdots$	
`setIntrs`:	$\cdots$	on activation of an alternative, set interrupts on all calls
`intrHandler`:	$\cdots$	called on reception of a call request when executing another alternative to save execution state and execute the `callHandler` routine to queue the request
`callHandler`$_1$:	$\cdots$	
`callHandler`$_2$:	$\cdots$	queue call request for A_i
	$\vdots$	
	$\cdots$	
`callHandler`$_m$:		
`exit`:	`FREER` c_{serve}	exit point for each alternative
`serveQueued`:	$\cdots$	

In this compiled alternation, the channel ends $c'_1, c'_2, \cdots, c'_m$ corresponding to each of the calls are used as request channels and the channel end c_{serve} is used to service the calls. Separating requests from the body of interactions avoids any interleaving of different call sequences, allowing short requests that will cause minimum congestion in the network. The same is done in the `hostHandler`, `subHandler` and `reqHandler` routines.

The following sections describe each of the components in the compiled alternation sequence.

Initialisation

For each unguarded or guarded alternative whose guard expression evaluates to the value `true`, the corresponding channel end is initialised to generate an event when a message is received that triggers the execution of the input or call and its command. Additionally, for call accept alternatives, the environment vector is set to specify an interrupt routine to be executed to queue the request when a call request is received but another alternative is being serviced.

```
initialise:
  (1) repeat for $i$ from 1 to $n$:
      • if $A_i$ is unguarded then:
          • if $A_i$ is an input then:
            LDAW $p$, label($C_i$)       load the address of the command $C_i$
            SETV $c_i$, $p$              set $p$ as the *event vector* of $c_i$
            SETC $c_i$, MODE_EVENT       set channel end to generate events
            EE $c_i$                     enable events on $c_i$
          • if $A_i$ is a call accept then:
            LDAW $p$, intrHandler        load the address of the interrupt handler
            SETEV $c_i$, $p$             set $p$ as as the *environment vector* of $c_i$
      • if $A_i$ is guarded then:
          • evaluate the expression $g_i$;
          • if $g_i$ =true then:
              • if $A_i$ is an input or call then:
                LDAW $p$, label($C_i$)       load the address of the command $C_i$
                SETV $c_i$, $p$              set $p$ as the *event vector* of $c_i$
                SETC $c_i$, MODE_EVENT       set channel end to generate events
                EE $c_i$                     enable events on $c_i$
              • if $A_i$ is a call accept then:
                LDAW $p$, intrHandler        load the address of the interrupt handler
                SETEV $c_i$, $p$             set $p$ as as the *environment vector* of $c_i$
      • if $A_i$ is just a guard with no input (skip) then:
          • if $g_i$ =true then execute $C_i$.
  (2) wait for an event (WAITE).
```

Alternatives

If A_i is an input

c_i `?` v`:` C_i

it is compiled to the sequence:

```
alt_i:  BL setIntrs()      call the set interrupts routine
        CHKCT c_i, END     } synchronise and close the connection
        OUTCT c_i, END     }
        IN c_i, v          receive the message data
        CHKCT c_i, END     } synchronise and close the connection
        OUTCT c_i, END     }
        ···                instructions of C_i
        B exit
```

If A_i is a call accept

accept c`(`$f_1, f_2, \cdots, f_k$`):` C_i

where c is a name and $f_1, f_2, \cdots, f_k$ are formal parameters, it is compiled to the sequence:

```
altᵢ:  BL setIntrs()   call the set interrupts routine
       IN cᵢ, d        receive the caller channel-end ID
       CHKCT cᵢ, END   } synchronise and close the connection
       OUTCT cᵢ, END   }
       SETD cᵢ, d      set the destination to the caller channel-end ID
       OUT cᵢ, 1       instruct the caller to continue
       OUTCT cᵢ, END   close the connection
```

Each parameter f_i in a call accept is allocated local storage a_i in the stack frame. If f_i is an array variable and the length is a specified constant c, then the storage $a_i[0]$ to $a_i[c-1]$ is allocated. If the length is unspecified or unknown, a single word a_i is allocated to f_i to store the address of a dynamically allocated region. The following sequence is libelled since the queued calls continue at this point. The parameter c is the channel end used for communication.

```
altContinueᵢ(c):
  (1) receive each parameter as a sequence of IN instructions:
        repeat for $j$ from 1 to $k$:
          • if $f_j$ is of type val or var, then receive the value and write it to $a_j$;
          • if $f_i$ is of type array, then receive size($f_i$) values and write them in sequence to the
            components of $a_j$;
          • if $f_i$ is of type process or function, then:
              • call the receive process subroutine (BL recvProcess($c$)).
  (2) synchronise and close the connection (CHKCT $c$, END; OUTCT $c$, END);
  (3) execute the call body, $C_i$;
  (4) send each updated parameter as a sequence of OUT instructions:
        repeat for $j$ from 1 to $k$:
          • if $f_j$ is of type var, then send the value $a_j$;
          • if $f_j$ is of type array, then send the components of $a_j$ in sequence.
  (5) synchronise and close the connection (OUTCT $c$, END; CHKCT $c$, END);
  (6) branch to the exit point (B exit).
```

Enabling call interrupts

On the successful activation of an alternative, the `setIntrs` routine is executed. This initialises interrupts on all of the channels for call alternatives and sets the event vectors to interrupt-handler routines that queue requests. This is so that the `intrHandler` routine in the next section can quickly activate the appropriate handler.

```
setIntrs:  LDAW p, intrHandler    load the address of the interrupt handler
           SETV p, c'₁            }
           SETV p, c'₂            }
           ⋮                      } set the event vectors to the interrupt handler
           SETV p, c'ₘ            }
           SETC c'₁, MODE_INTR    }
           SETC c'₂, MODE_INTR    }
           ⋮                      } set the channel ends to generate interrupts
           SETC c'ₘ, MODE_INTR    }
           SETSR INTR_ENABLE      enable interrupts
           RETSP 0                return to caller
```

Handling call interrupts

When a request arrives at a call channel end, an interrupt is generated and the following interrupt handler is executed. The handler first saves the execution state by writing *r*0 to *r*11 and *lr* to the stack, then invokes the call handler to deal with the request, then it restores the state and returns from the interrupt.

```
intrHandler:  EXTSP $N_{\text{framesize}}$        allocate stack space, where $N_{\text{framesize}} \geq 14$
              STW r0, sp[1]          ⎫
              STW r1, sp[2]          ⎪
              ⋮                      ⎬ save r0 to r11 and lr to the stack
              STW r11, sp[12]        ⎪
              STW lr, sp[13]         ⎭
              BL ed                  call the callHandler routine from the environment
                                     vector that was stored in register event data register
                                     when the interrupt occurred
              LDW r0, sp[1]          ⎫
              LDW r1, sp[2]          ⎪
              ⋮                      ⎬ restore r0 to r11 and lr from the stack
              LDW r11, sp[12]        ⎪
              LDW lr, sp[13]         ⎭
              EXTSP $-N_{\text{framesize}}$       deallocate stack space
              SETSR INTR_ENABLE      enable interrupts
              KRET                   return from the interrupt
```

Call-handler routine

The `callHandler` routine is called from the interrupt handler to deal with a call request that is received when another alternative is being serviced. It establishes a connection with the caller and receives its location, process ID and process index in order to queue the request to be dealt with after the processing for the active alternative. It assumes the queue data structure is allocated in memory at address q.

```
callHandler$_i$:  IN $c_i$, $d$                       receive the caller channel-end ID
                  CHKCT $c_i$, END                    ⎫ synchronise and close the connection
                  OUTCT $c_i$, END                    ⎭
                  SETD $c_{\text{serve}}$, $d$                  set the destination to the caller
                  OUT $c_{\text{serve}}$, 0                     instruct the caller to wait
                  OUT $c_{\text{serve}}$, $c_{\text{serve}}$              send the new service channel-end ID
                  OUTCT $c_{\text{serve}}$, END                 close the connection
                  IN $c_{\text{serve}}$, $v_{\text{location}}$            receive the process location
                  IN $c_{\text{serve}}$, $v_{\text{procid}}$              receive the process ID
                  IN $c_{\text{serve}}$, $v_{\text{index}}$               receive the process index
                  CHKCT $c_{\text{serve}}$, END                 ⎫ synchronise and close the connection
                  OUTCT $c_{\text{serve}}$, END                 ⎭
                  BLT tjump[enqueue]
                    $(q, d, v_{\text{location}} v_{\text{procid}}, v_{\text{index}})$   queue the request
                  RETSP 0                             return to the interrupt handler
```

Service queued call requests

On completion, each alternative branches to a single exit point where upon any queued requests are removed from the queue and serviced. Interrupts are re-enabled to service each call in order that incoming calls can continue to be queued and not block in the network. The queue size at this point may grow, but this process is repeated until there are no outstanding calls and the queue is empty.

`serveQueued`:

(1) `BLT tjump[dequeue]` $(q, v_{\text{ccall}}, v_{\text{procid}}, v_{\text{cclient}}, v_{\text{location}}, v_{\text{index}})$ — dequeue a request
    `BL setIntrs()` — set the call interrupts

(2) if $v_{\text{ccall}} \neq 0$ (a request was dequeued), then:

- if $v_{\text{index}} \geq 0$ (it was from a component process of an array), then:
  `BLT tjump[request]` $(v_{\text{location}}, v_{\text{index}}, v_{\text{procid}}, v_{\text{cclient}})$ — request the remote channel-end ID
  `SETD` $v_{\text{ccall}}, v_{\text{cclient}}$ — set this as the destination
- `OUTCT` c_{serve}, `END` — instruct the caller to continue
  `B altContinue`$_i(c_{\text{serve}})$ — branch to the remaining portion of the alternative

8.5.2. Communication

Input and output

Let c be a channel end and v be a value, then an output command

$$c \;!\; v$$

is compiled to the sequence

`OUTCT` c, `END`	synchronise and close the channel connection
`CHKCT` c, `END`	
`OUT` c, v	output the value of v
`OUTCT` c, `END`	synchronise and close the channel connection
`CHKCT` c, `END`	

Let v be a variable, then an input command

$$c \;?\; v$$

is compiled to the sequence

`CHKCT` c, `END`	wait to synchronise and close the channel connection
`OUTCT` c, `END`	
`IN` c, v	receive a value
`CHKCT` c, `END`	wait to synchronise and close the channel connection
`OUTCT` c, `END`	

Appendix B.5 explains the use of synchronised communication in these.

Calls

Let $a_1, a_2, \cdots, a_m$ be actuals, $f_1, f_2, \cdots, f_m$ be formals, c be the name of a local channel end, e_{location} be an expression for the processor ID, e_{index} be an expression for the process index, v_{procid} be a process ID value and n be a name of a call. Then, a remote call from a single process

c`@(`e_{location}`:`v_{procid}`:`e_{index}`) ! `n`(`a_1`, `a_2`, `$\cdots$`, `a_m`)`

where the call has the specifier $[\![$**`call`** n`(`f_1`, `f_2`, `$\cdots$`, `f_n`)`$]\!]$ is compiled to the sequence:

```
          OUT $c$, $c$                  send the local channel-end ID as the client identity
          OUTCT $c$, END             } close the connection
          CHKCT $c$, END             }
          IN $c$, $r$                   receive a response (0 to wait or 1 to continue)
          IN $c$, $d$                   receive the service channel-end ID
          SETD $c$, $d$                 set this as the new destination
          CHKCT $c$, END             close the incoming connection
          BT $r$, continue
wait:     OUT $c$, $e_{\text{location}}$         send the processor location
          OUT $c$, $v_{\text{procid}}$           send the process ID
          OUT $c$, $v_{\text{index}}$            send the process index
          OUTCT $c$, END             close the outgoing connection
          CHKCT $c$, END             wait to continue
continue:
```

(1) send each parameter as a sequence of OUT instructions:
 repeat for i from 0 to $m-1$:
 - if f_i is of type **`val`** or **`var`**, then send the value a_i;
 - if f_i is of type **`array`**, then send $\text{size}(f_i)$ components of the array;
 - if f_i is of type **`process`** or **`function`**, then:
   - send the ID of the procedure or function;
   - receive a flag f to signal acceptance of the process;
   - if $f = 1$ then, repeat for j from 0 to `tsize`$[p]$:
     - read the word at address `tjump`$[p + j]$ and send it.

(2) synchronise and close the connection (`OUTCT` c, `END`; `CHKCT` c, `END`);

(3) receive each updated parameter as a sequence of `IN` instructions:
 repeat for i from 0 to $m-1$:
 - if f_i is of type **`var`**, then receive the value and write it to the storage location for a_i;
 - if f_i is of type **`array`**, then receive $\text{size}(f_i)$ values and write them in sequence beginning at address a_i.

(4) synchronise and close the connection (`OUTCT` c, `END`; `CHKCT` c, `END`);

8.5.3. Connect

Let c be the name of a local channel end, e_{location} be an expression for the processor ID, e_{index} be an expression for the process index, v_{procid} be a process ID value and n be a name of a call. Then, a connect command

`connect` c **`to`** `@(`e_{location}`:`v_{procid}`:`e_{index}`).`v_{intid}

is compiled to the sequence

BLT `tjump[request]` $(e_{\text{location}}, e_{\text{index}}, v_{\text{procid}})$	call the request routine
SETD c, *r*0	set the channel-end destination to the remote channel-end ID

8.5.4. On

An instance of a procedure with an on clause is compiled into instructions that forms a representation of the closure. The closure is then passed as a parameter to a *source-handler* routine that performs the interaction with the remote processor to transfer the closure, perform the execution and receive any updated free variables.

Let $f_1, f_2, \cdots, f_n$ be formals and N be the name of a process

process N $(f_1, f_2, \cdots, f_n)$ P

then, an instance of the process prefixed with an on clause

on e **do** $N(a_1, a_2, \cdots, a_n)$

is compiled into the following sequence that constructs a parameter in memory that describes the components of the closure. In this, each consecutive write stores a word to the next consecutive word address (i.e. STW d, *sp*$[x]$; STW d, *sp*$[x+1]$, STW d, *sp*$[x+2]$ etc.); storage for the closure is allocated on stack and requires $m + n + 3$ words.

(1) write $\text{procid}(N)$;

(2) write n, the number of parameters;

(3) repeat for i from 0 to n:
- if f_i is of type **val**, then write **val** (0) and the value a_i;
- if f_i is of type **var**, then write **var** (1) and the value a_i;
- if f_i is of type **array**, then write **array** (2), the value $\text{size}(f_i)$ and the address a_i;
- if f_i is of type **process** or **function**, then write the process ID, $\text{procid}(a_i)$.

(4) call the source routine with the value of e, specifying the destination processor and the address of the closure (BLT `tjump`[source](e, *sp*$[x]$)).

8.5.5. Interface

Each component of an interface is allocated a channel end, which is then submitted to the service process to be made available to channel-end ID requests from remote processes.

Let $s_1, s_2, \cdots, s_n$ be interface specifiers, e_{location} be an expression for the processor ID, e_{index} be an expression for the process index and v_{procid} be a process ID value. Then an interface specification

interface $(s_1, s_2, \cdots, s_n)$@$(e_{\text{location}} : v_{\text{procid}} : e_{\text{index}})$

is compiled to:

(1) repeat for i from 1 to n:

GETR c, CHANEND	allocate a channel end
BLT `tjump[submit]`$(c, e_{\text{location}}, v_{\text{procid}}, e_{\text{index}})$	call the local submit routine

8.5.6. Parallel

A parallel command with n component processes

```
{ $P_1$ & $P_2$ & $\cdots$ & $P_n$ }
```

is compiled to create and initialise $n-1$ *slave* threads to execute processes $P_2, P_3, \cdots, P_n$. The *master* thread, which manages this process also executes P_1. Since each slave process is a process instance, the initialisation of each thread consists of initialising its registers and stack according to the calling convention, the program counter is set to the process entry point and the *lr* is set to an exit point.

Let $\text{actuals}(P)$ denote the set of actual parameters of a process instance P. Then, the following routine sequence defines the initialisation performed by the master thread.

(1) get a thread synchroniser resource (`GETR` s, `SYNC`);

(2) repeat for each component process i from 2 to n where $\{a_1, a_2, \cdots, a_m\} = \text{actuals}(P_i)$:

`GETST` t, s	get a synchronised thread t
`LDAP` p, $\text{label}(P_i)$	load the address of the slave entry point
`TSETR` *pc*, p	set the program counter to p
`LDAP` p, `slaveExit`	load the address of the slave exit point
`TSETR` *lr*, p	set the *lr* to p
- repeat for i from 1 to 4:
  write the value of v to register $i-1$ (`TSETR` *ri* $- 1$, t, v) where:
  - v is the value of a_i if f_i is of type **`val`**;
  - v is the address of a_i if f_i is of type **`var`**;
  - v is the value of a_i if f_i is of type **`array`**;
  - v is `tjump`$[\text{procid}(a_i)]$ if f_i is of type **`process`** or **`function`**.
- repeat for i from 5 to m:
  load the value v onto stack at offset $1 + (4 - i)$ subject to the type conditions of a_i in the previous step.

This is followed by the instruction sequence:

	`MSYNC` s	start execution of slave threads
$\text{label}(P_1)$:	$\vdots$	P_1 instructions
	`MJOIN`	synchronise and terminate with slave threads
	`B masterExit`	
`slaveExit`:	`SSYNC`	synchronise with the master thread
`masterExit`:	`FREER` s	free the synchroniser

8.5.7. Procedure calls

Let N be a name and $a_1, a_2, \cdots, a_n$ be actual parameters, then an instance of a procedure or function

```
$N$($a_1$, $a_2$, $\cdots$, $a_n$)
```

is compiled according to the calling convention, except that instead of using the conventional link-and-branch instruction, `BL`, it is performed via the jump table with the `BLT` instruction:

```
BLT tjump[$N$]($a_1$, $a_2$, $\cdots$, $a_n$)
```

This is because procedures and functions from the user program are moved between different processors and their locations in memory will vary. In contrast, the relative offsets between procedures in the run-time kernel will not change, and for inter-kernel calls, the `BL` instruction can be used.

8.6. Discussion

This section discusses ways in which the compilation process could be improved or extended.

8.6.1. Dynamic processor allocation

The proposed sire compilation scheme does not allow parallel replicators to have a dynamic size (although this is not an inherent restriction in sire). Allowing dynamic replicators would potentially make the expression of some program structures more convenient, at the cost of moving the allocation into the program execution.

To support dynamic replicators, additional dynamic allocation routines and a new transformation stage would be required. A *processor allocation routine* would be passed a required number of processors and return the location of a contiguously-addressed block of that size; a *processor deallocation routine* would be passed the location of an allocated block of processors to free them for reuse. These routines are analogous to `malloc` and `free` in the C programming language. The additional transformation would operate between stage 6 and 7, to insert calls to allocate processors before a dynamically-sized parallel command and to deallocate them after. The base processor location returned by the allocation routine can then be passed as a parameter to the component processes that reference channel ends of the component processes of the replicator.

8.6.2. Latency hiding

The proposed processor allocation scheme allocates one process to each processor. This approach was taken primarily to simplify the explanation of the subsequent transformations that deal with absolute references. However, processes waiting on communication will cause the processor to idle.

An implementation of the UPA is characterised by the *number of processors*, the *communication grain* and the *communication latency*. By allocating a set of processes to a single processor, processes not waiting on communication can be executed while others are, and by allocating enough processes to a single processor (based on the latency) the processor will, in principle, never idle. This is the approach taken by PRAM emulations with BSP machines to obtain *optimal* efficiency [Val90a, Val90b].

Latency hiding with PRAM programs is relatively simple since there is no locality, so processes can be allocated to any processor and all memory accesses will complete in a time directly proportional to the latency. Although latency hiding can in principle be applied to sire programs, it is complicated by two issues relating to the use of communication channels. Firstly, arrays of processes must be allocated on consecutively numbered processors in order for component processes to be addressed by a base location and an offset. One solution to maintain this, while increasing the number of processes per processor, is to allocate consecutive chunks of an array to consecutive processors. The addressing then works by dividing offsets by the chunk size. Secondly, the latency of server calls depends not only on the network but also by the time taken to execute the call. Latency hiding for collections of processes that access servers can only be performed when the execution time of a call can be determined. However, it is likely that in many cases, such as with the server-based random access memories described in §7.2 [p. 116], that the latency of a server call is dominated by the machine latency as it would be for memory accesses in a PRAM, thus allowing latency to be more easily hidden.

Several potential optimisations when performing latency hiding are discussed in the following section, described as *layering servers with clients* and *server reduction*.

8.6.3. Cases for optimising server communication

The compilation scheme described deals with server communication in a uniform way. Analogously to the special allocation cases of the previous section, the following special cases can be taken advantage of to simplify the implementation of communication.

1. *A single client.* One-to-one client-server relationships are likely to occur in programs that use process structures that provide an access abstraction with collection of servers to implement a data structure (server structures of this nature were explained in §7.2.4 [p. 121]).

   If the server only ever serves one client, it can either be allocated to the same processor, so the calls execute locally, or be compiled directly in into the client code as a sequence of instructions, to remove any inter-process overheads all together.
2. *Multiple single clients.* Communication with a server that deals with one client at a time can be implemented with a channel without any additional routines to deal with interrupts.
3. *A small number of clients.* If a server deals with a small number of clients (i.e. less than the number of available channel ends), then each client can be assigned a separate channel for each call so that the client does not need to identify itself every time it makes a call.
4. *Server reduction.* To perform latency hiding with an array of 'access' servers that map one-to-one to an array of client processes, each processor would be allocated a collection of client processes sufficient to hide the latency of the server. Since only a fraction of the servers will be active (proportional to $1/\ell$), the number of server processes can be reduced and shared between the clients to improve the use of available threads.
5. *Layering clients with servers.* When the latency of server calls makes it difficult or impossible to hide the latency, servers can be allocated with their clients. The effect of this is that the processor will either be executing the client, or a server call. With server and client arrays, provided the behaviour of client components was similar, a similar effect in the average case over the array should be observed.
6. *Memory access servers.* A server that provides direct access to a memory, such as the `Store` server listed in Process 7.21 in §7.2.4 [p. 121], could potentially be implemented directly by a remote memory access mechanism provided by the hardware. The performance benefits of doing this are investigated later in Chapter 10 and show a saving of around 20%.

These are just a few ideas, it is likely that there are many more opportunities for interesting optimisations.

8.6.4. Threading

Inefficient use of threads with parallel replicators

Compilation deals with the creation of new processes (from the parallel command and the on clause), by mapping them directly to thread resources provided by the hardware. This is sufficient when the number of available threads per processor is not exceeded, a condition that can be checked during compilation because of the design of sire. It is likely that a modest number of threads will suffice for most programs (current implementations of the XS1 architecture have 8 threads per processor [XMO12b]) but this poses a problem for one aspect of the proposed implementation of sire.

Replicated processes are transformed into a recursive form to distribute the component processes over a collection of processors rapidly. A consequence of this approach is that with synchronous on clauses, threads executing on clauses become idle for the duration of the execution of the process array. For an array of N processes, the processor executing the root node of the distribution tree will have $\log_2 N - 1$ idle threads. If, for example, $N = 1024$, then 9 threads will be idle at the root. This is clearly an inefficient use of threads and moderately sized parallel replicators would exceed realistic implementations of the processor core.

One solution to this would be to implement a simple thread virtualisation scheme whereby a scheduler manages the execution of an arbitrary number of *virtual* threads by sharing the available physical threads. Such a scheme could be specialised to deal only with replicators in order to reduce the complexity of a full-blown general virtualisation of the processor's resources.

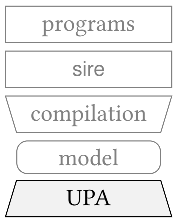
Stack allocation

Allocation of stack space to threads operates by dividing the available stack space into fixed-sized chunks. This scheme was chosen for simplicity, but since recursion is not permitted, the stack usage can be determined at compile time and threads can be allocated exact-size chunks. Allocation is this way, for example, how the XMOS XC language is compiled [Wat09]. Incorporating exact stack allocation into the implementation of sire would be straightforward. The stack requirement can be recorded in the process table and included in process closures for distribution as well as the process size.

It is worth noting here that it would be possible to relax the constraint on recursive processes to allow one recursive process per processor. This is because the recursive process can be placed at the outer-most position of the stack space so that it can grow into the remaining available memory. To deal with general recursion, stack frames could be allocated on the heap or recursive parallel programs could be transformed so that each processor contains at most one recursive process, a suggestion posited by Welch [Wel92].

Part III.

EVALUATION OF IMPLEMENTATION COST AND PERFORMANCE



Chapter 9. An implementation model for the UPA

This chapter presents an implementation model, based on current production technology, to investigate whether the abstraction it provides of the machine structure is efficient in terms of its implementation cost, when compared with an equivalent system with a non-universal network.

The proposed model is a *straw man* in the sense that it is based on general approximations of the capabilities of current VLSI and packaging technology to obtain *rough*, but not *unrealistic* designs. It is used to evaluate implementation cost and, in particular, how the cost of the interconnection network scales with respect to processing and memory. The model then forms the basis for a performance model which is used for the experimental performance evaluation in the next chapter.

9.1. Overview

The UPA is implemented with one or more chips, with each chip containing a complete sub-folded Clos network with replicated processing tiles, switches and communication links. For multichip systems, each chip contributes additional switches to form the core switching stage and communication links connect switches on different chips, in the same way they are connected on-chip. The size of the chip is chosen to trade-off between performance and system cost, similar to commodity DRAM.[1]

The most challenging aspect of the UPA is the density of connectivity between chips that is required to extend the universal network topology. A silicon interposer is used to do this as it is the only current production technology that provides the required level of connectivity between chips. However, in the future, other forms of 3D integration would apply naturally to package these chips in dense stacks; future technology is discussed briefly in §9.6.3 [p. 202].

To provide a baseline for the evaluation, an equivalent construction with a (non-universal) 2D mesh network is presented. This is due to its popularity in the literature and a number of implementations because it has a simple structure that is amenable to packaging onto a 2D surface. For brevity, in the remainder of this chapter and the next one, the folded Clos and 2D mesh networks are referred to as simply *Clos* and *mesh* respectively.

9.2. Background

The following sections provide background to some of the aspects of the model: the switch, the memory technology, the interposer and the wire characteristics.

9.2.1. Switch

The characteristics of the switch component are based on the *INMOS C104 chip* [MTW93, Ch. 3] because it was designed to be used as a component for universal communication networks [MTW93, Ch. 8] and it provides the capabilities required for the UPA including a degree of 32, wormhole switching and interval routing. Furthermore, its connectivity provides a good match with the available area of a chip and the size of the system components. A Clos with two stages can connect up to 256 tiles and this fits well into the target economical chip area. A 64×64 switch can connect up

[1] DRAM chip sizes have remained virtually constant between technology generations in order to maximise cost-effectiveness. Current DRAM chips are manufactured with an area of around 100 mm^2 with a capacity of 0.3 Gbits [Int12a, pp. 84-104].

to 1,024 tiles with two stages, but this far exceeds the target area; a smaller network would have to be built using this switch that would not fully exploit its connectivity. Figure 9.1 shows some examples network constructions with a 32×32 switching element for 64, 256 and 1,024 tiles.

9.2.2. Memory technology

The choice of tile memory technology and capacity depends on obtaining a good trade-off between access latency, area and its integration with microprocessor logic. Ideally, the access time of a memory should be matched with the time to execute one operation, but access time is in general inversely related to capacity, therefore faster memories require more silicon area.

There are two dominant forms of fast random access memory that can be integrated with CMOS logic: *static RAM* (SRAM) and *embedded dynamic RAM* (eDRAM):

- SRAM can be integrated directly, has fast access latency and good random access performance but it has a relatively low density since it uses six transistors per bit;
- eDRAM is integrated with CMOS logic by adding three to six additional steps to the fabrication process and is more dense than SRAM (by a factor of 2 to 3) since it uses a single transistor-capacitor pair to store a bit [Int12c].[2]

Contemporary commodity *dynamic RAM* (DRAM) in contrast, is produced on a different manufacturing process that is optimised for density and power, and is several technology nodes behind production logic devices. Consequently, DRAM chips operate at a lower clock frequency and have long access latencies.[3] DRAM is typically packaged on *printed circuit board* (PCB) modules, called a *dual-inline memory module* (DIMM), and connected with wire traces on another PCB, but this severely constrains the potential bandwidth between chips.

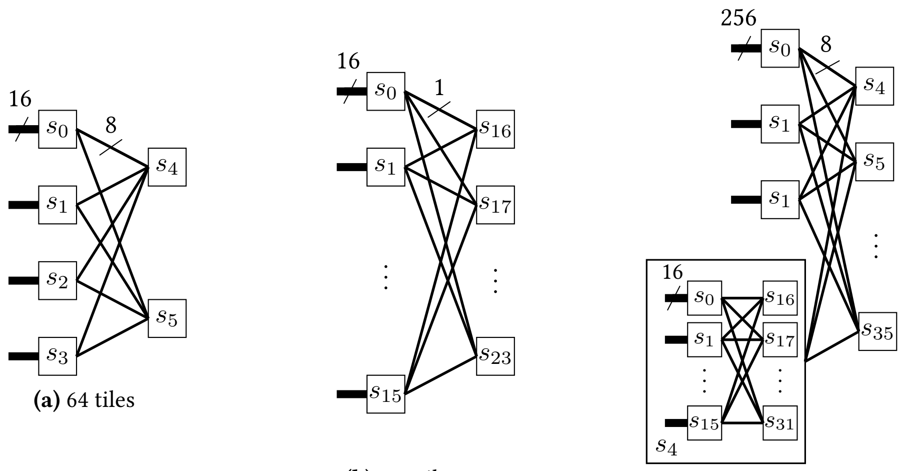
9.2.3. Interposer

Conventional packaging technology poses significant challenges for interconnecting collections of chips that contain large numbers of processors. A high density *ball grid array* (BGA) package will have around 2,000 pins [Int10a, Tab. AP11] and typically around 40% of these have to be used for ground and power, leaving 1,200 for signals. For bidirectional links with 8 wires in each direction, only 75 such links could be connected. PCBs offer a minimum wiring pitch of about 0.15 mm and a high density package would require around 6 wiring layers to route tracks to each of the pins [Pfe07] and the resulting density of wiring on the board is low; for instance, 16 links side-by-side would occupy 38.4 mm. Even for modest numbers of processors, this technology does not provide the connectivity between chips that is required by a universal communication network. It would require the number of processors on a single chip to be reduced to match the package pin-out and to connect them chip-to-chip in three-dimensional space, but this would have significantly increase power and latency.

It is clear that the construction of general-purpose parallel systems spanning multiple chips requires an interconnection technology that provides higher density connections. *Silicon interposers* are becoming an established technology [KPA$^+$06] that can be used as a substrate to interconnect multiple processing chips with high-density wiring. In essence, they can be thought of as a high-density PCB. The interposer can be manufactured on a larger process geometry than the processing chips since this provides a sufficient wiring density. Larger geometries are higher yield because the technology is better developed, so it is practical to produce larger die sizes.

[2] As the technology has matured, the access latency has approached that of SRAM [AGJP11], especially for large memories where the flight-time across the array can dominate access latency and the improved density of eDRAM can significantly reduce this. Its main application has been to build large on-chip caches, replacing SRAM, for example in the BlueGene/L [IBP$^+$05] and Power7 [BPN$^+$10] chips.

[3] Although DRAMs have long provided the highest density, it is projected that SRAM will reach the density of standard DRAM by 2020 and could potentially be a replacement [Hoe12], due to problems scaling capacitors.



(a) 64 tiles

(b) 256 tiles

(c) 1,024 tiles

Figure 9.1.: Example 32×32 crossbar switch topologies for various size Clos networks. Edge switches connect to 16 tiles and core switches use all 32 links to connect to the previous stage. The 1,024-tile network is constructed by replicating a 256-tile network four times and connecting it with 32 core switches, making it three-stage. This construction maintains capacity between stages and has a logarithmic diameter (2 or 3 for the examples) in the number of tiles.

Xilinx use a 65 nm 776 mm^2 silicon interposer in their Virtex-7 FPGA [Sab11], which was available in 2011, to connect four 28 nm FPGA 'slices'. This was the first production device to do so, with the principal reason to do this being an economic one. At a cutting edge 28 nm process technology[4] yields are lower, so producing a number of identical smaller devices and connecting them with an interposer to produce a single larger device provides a good trade-off between cost and performance. The same philosophy is taken with the proposed implementation model and the characteristics of the Xilinx interposer are used as a basis for it.

9.2.4. Wires

The logic and wiring resources provided by VLSI devices are plentiful and they continue to scale but their utilisation is significantly limited by two factors: one is the connectivity of packages that was discussed in the previous section and two, the performance of long wires (relative to the size of a chip) in terms of delay and power [HMH01].

There are techniques for implementing power-efficient global interconnections such as low-swing signalling schemes [HMH03] and optimisation of wire width and spacing [LMHL05], but an unavoidable effect of the scaling of VLSI devices is that wire delays exceed a single clock cycle. This makes modular parallel architectures that can tolerate these latencies an attractive way to build systems [Ho03], and adds weight to the general approach proposed in this thesis.

Wire delay is related to the square of the length of the distance a signal is transmitted. Long wires are therefore unattractive. A conventional technique to reduce wire delay is to insert *repeating elements* at regular intervals along a wire [IF99], [Ho03, Chap. 4]. This reduces the total delay to a multiple of the delay between repeaters, at *optimal* intervals, this produces a linear relationship between delay and length. The cost of this is the delay through the repeaters and the logic area and power required for them.

[4]The *process technology* or *process geometry* of a chip refers to the *minimum half pitch of contacted metal lines*, as defined by the ITRS [Int12a, pp. 5-6], which is the minimum feature size of a circuit. This has historically been an indicator of IC scaling and it has been DRAM that has exhibited the smallest metal pitch.

9.3. Implementation model

The following sections describe the components of the model and discuss the layouts of the *processing chip* for the Clos and mesh topologies, and the chip layout and wiring for the *interposer chip*. The layouts are used to estimate area requirements and wire delays. The parameters of the model are described in §9.4 [p. 186] and summarised in Table 9.1 and Table 9.2.

9.3.1. VLSI model

The following VLSI model is used to produce a floorplan for the chip and is based on those in Mead and Conway [MC80] and Ullman [Ull84].

Metal layers

- A chip consists of a number of *metal layers*. Metal layer 1 (M1) is used for logic components and the remaining metal layers M2, M3, M4 etc. are used to route wires.
- A particular metal layer contains wires orientated in one direction and adjacent metal layers contain wires in perpendicular directions. This is to reduce *crosstalk*, where a signal in one circuit affects the signal in another circuit.

Wires

- Wires carry signals in one direction and all wires are *half shielded* to further reduce crosstalk. This is where a ground wire is placed either side of a wire pair. The result is that minimum wire pitch increases by a factor of $\frac{3}{2}$.
- All wires have *repeaters* inserted to minimise latency.
- Wires with multicycle delays are *pipelined* by inserting flip-flops.
- For sets of wires, banks of repeaters and flip-flops are required. To simplify their insertion, interconnect wires are routed in dedicated channels.

Chip

- The area of a chip is taken to be the smallest rectangle in which the complete circuit fits.
- External connections are made to *pads* with a fixed *driver* circuit component. This is due to the higher capacitance of external wires.

9.3.2. Chip layout

A UPA processing chip contains a collection of tiles and switches that can be used by itself or as a sub-component to build a larger system. A chip presents a number of external links from its switches and multiple chips are connected by joining links between switches in different chips according to the interconnection topology, thereby extending the network.

VLSI layouts of the architecture with Clos and mesh networks are described. These are not specified rigorously since the explanations are relatively simple and the model aims only to produce ball-park estimates. Software models of these layouts are used to produce figures for the dimensions of chips and their components and wire lengths. These figures are then combined and compared to show the relative proportions of processors and memory to switches and interconnect wiring.

Assumptions and limitations

To avoid over-complicating the model, only the following essential aspects of the system are captured in the layout:

- the only components included in the layout are the processor core, memory, switch, pad driver circuitry and IO pads (internal component wiring is not considered);
- only the routing of the inter-switch and switch to chip IO wiring is considered since this is the most significant component of the system in terms of wire lengths and density;
- the remaining communication links between switches and processors are not explicitly accounted for in the area; it is assumed they can be routed over other resources, in the area not occupied by the interconnect wiring channels;
- links are connected between components and it is assumed they can be connected from any position within the footprint of a component;
- layout relating to wiring and IO for power and clock signals is not considered, but provision is made for them in the available metal layers and chip IO.

Clos layout

A UPA processing chip contains a complete Clos network connecting a collection of tiles with two or more stages, banks of unconnected switches that are contributed to the core-stage of a larger multi-chip network and driver circuitry for off-chip IO. The layout of the Clos network has a recursive structure and is based on a *H-tree*, an efficient embedding of a binary tree onto a 2D surface [MC80, Ch. 8]. An example layout for a 2-stage 256-tile Clos network relating to the topology of one of the 4 sub-Clos networks in Figure 9.1c is given in Figure 9.2a.

All of the links from the system core switches are routed to off-chip pads since the connections between the core switches and the chip's core switches depends on the size of a network. This is so that they can be connected to the correct corresponding switches for a particular network size, both on the same chip and in core switch banks of other chips. The core switches of the sub-Clos are placed in the centre of the chip and half of the links are routed to the off-chip IO. The next stage of switches in is divided into four groups and each are placed in the centre of a quadrant surrounding the core switches. Connections are made from all of the core switches to each of the next stage switches. This process continues recursively in each quadrant for each additional stage. The minimum separation of groups of tiles is constrained by the placement of switches and the width of the wiring channels between them.

Groups of switches are arranged in staggered sets to minimise the resulting size of their bounding box. Their placement relative to one another is constrained in two dimensions by the pitch of wires connecting horizontally and vertically. These are routed on a minimum of two metal layers, but additional layers can be used to reduce the dimensions of a group. Each group of switches has *branching* connections in both directions. This restricts both the horizontal and vertical placement of switches. A switch arrangement is chosen to minimise the width of the group subject to not exceeding the height of its quadrant since it can extend into the wiring channel. An example of the wiring pattern for a switch group is illustrated in Figure 9.2a. The total area of the interconnect is calculated as the sum of the area of the wiring channels and all of the switch groups.

The pads and driver circuitry for off-chip IO are positioned along one edge of the chip due to the wiring pattern on the interposer. A chip that contains N tiles requires IO for $2N$ links to extend the network, N from the core switches and N from the contributed bank of system core switches.

Mesh layout

A mesh network is laid out as an array of blocks of processors where each block of processors attaches to a single switch. These are separated by wiring channels accommodating the width and height of a switch, which is placed at the corner of its block. Adjacent horizontal and vertical links connect directly between switches. The layout for a 256 tile mesh network is given in Figure 9.2b.

The pads and driver circuitry for the off-chip IO are positioned at the edges of the chip so that the mesh can be extended directly between adjacent chips. A chip that contains N tiles requires IO for $4\sqrt{N} - 4$ links to extend the network.

9.3.3. Silicon interposer

The processing chips are mounted on a silicon interposer using a *flip-chip* assembly. This bonds the top side of the chip with an array of solder *microbumps*. The interposer provides wiring traces between the microbumps to connect links between switches on different chips. The interposer is connected to a package substrate with *through-silicon vias* (TSVs) that provide a connection through the interposer's substrate to *controlled collapse chip connection* (C4) flip-chip bumps on the package substrate. These connections only bridge the power, ground and clocking IOs. The package substrate carries these connections out of the package, to the BGA balls.

For a Clos topology, a set of chips are arranged in two rows, either side of a wiring channel on the interposer, orientated with their wiring area closest to this. Each wire in the channel is used to connect between pads on two chips with additional horizontal wires. The wiring channel contains sufficient wires so that every pair of connections between the chips can be made. The maximum height of the channel between two chips is therefore twice the total pitch of the wires connecting a chip. For a mesh topology, connections are provided on the interposer between the links of adjacent chips, directly extending the mesh. In both networks, wiring for power, ground and clock signals are not included, but there is sufficient space in other areas of the chip. Figure 9.4 illustrates the packaging of a set of processing chips stacked on the silicon interposer for both networks.

9.4. Model parameters

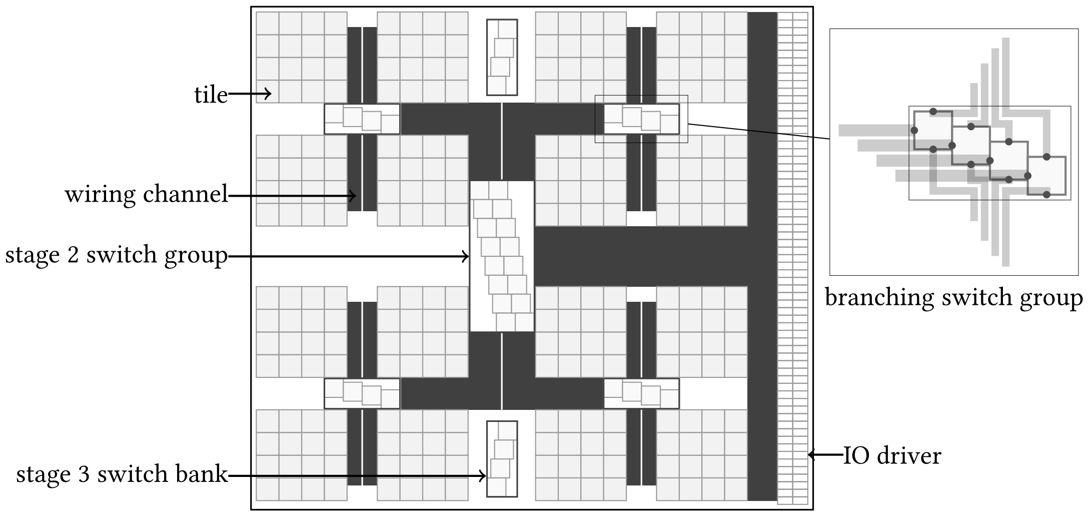
This section presents and explains the parameters used to characterise current production technology for the implementation model. The *International Technology Roadmap for Semiconductors* (ITRS), a set of documents produced annually by a broad collection of industry members since 1994, provides many of these; others are taken from elsewhere in the literature or estimated. Table 9.1 summarises the parameters for the processing chip and Table 9.2 summarises the parameters for the interposer.

The remainder of this section is divided into the parameters of the processing and interposer chips, the main VLSI components the memories.

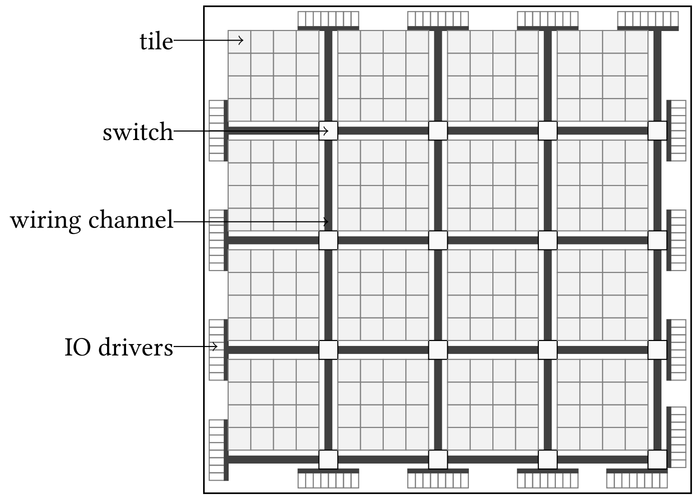
9.4.1. Processing chip and interposer

The parameters for the processing chip are based on a 28 nm logic process technology, which is representative of current production devices [Int12a, Tab. OCTC-2C]. The chip IO pad area includes the contact and driver circuitry and its dimensions are estimated based on a 1 to 4 ratio between width and height which is characteristic of conventional designs [WM05]. The width is taken to be the pitch of the interposer contacts (microbumps). An on-chip processor and interconnect clock rate of 1 GHz is chosen as this is typical of an embedded processor. The parameters for the silicon interposer and packaging are based on the *Xilinx Virtex 7 FPGA* [Jon10, Sab11, Ram11], however it is assumed that wires can be repeated.

All links between switch components on the processing chip have 9 wires in each direction, allowing up to 1 byte to be transmitted per cycle with an additional bit to signal control tokens. All off-chip links have 5 wires in each direction, allowing up to 1 byte to be transmitted every 2 cycles. This is to reduce the off-chip IO requirements and interposer wiring.



(a) Clos layout



(b) mesh layout

Figure 9.2.: Block diagrams of the 256-tile Clos and mesh network layouts for the UPA processing chip. In these, groups of 16 tiles connect to a single switch. In (a), the topology corresponds to one of the four sub-Clos networks in Figure 9.1c. The layout is based on a H-tree organisation with groups of switches at each node organised in staggered sets to minimise area subject to the constraints of their vertical and horizontal connections. Running along the right hand side of the chip is driver circuitry for each chip IO pads. Connections from switches to processors and IOs for power and ground are not included in the model. In (b), running around the edge of the chip is the diver circuitry for each of the chip IO pads. Connections from switches to processors and IOs for power and ground are not included in the model.

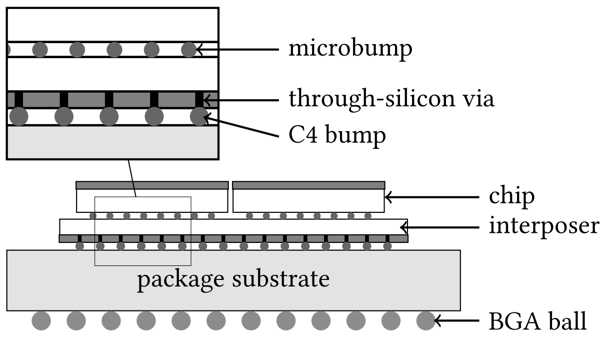
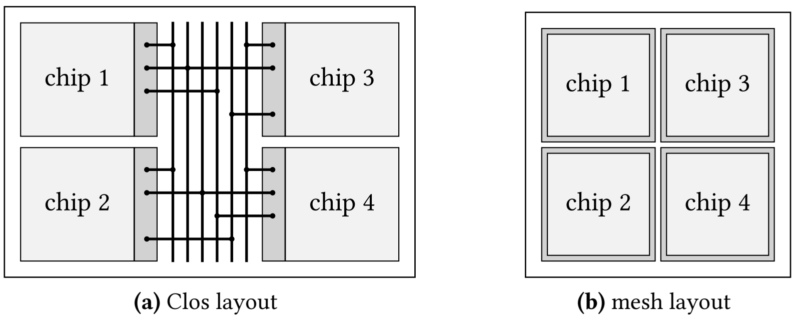


Figure 9.3.: Cut through view of the packaging. A collection of chips are stacked using a flip-chip assembly onto solder microbumps, connecting them to a silicon interposer which provides dense connectivity between them. The interposer contains TSVs to provide connectivity though its substrate and these are connected with C4 solder bumps to the package substrate.



(a) Clos layout **(b)** mesh layout

Figure 9.4.: Chip layouts on the interposer. For the Clos network, chips are located beside a wiring channel that contains a common wire for every connection between two chips. A connection is made by connecting horizontal wires from each chip to the common wire; diagram (a) shows a wiring pattern for connections from each chip to every other chip. For the mesh, in diagram (b), chips are arranged in a grid and the network is extended directly between chips.

The delay of an optimally repeated wire can be estimated with the equation

$$\tau = 1.47\sqrt{\text{FO4} \cdot \hat{R}\hat{C}}$$

where τ is the delay in picoseconds per unit length; $\hat{R}\hat{C}$ is a time constant equal to the product of the resistance per unit length and the capacitance per unit length; and FO4 (*fanout-of-4*) delay is a constant related to the particular process technology, defined as the delay of an inverter, driving four identical copies of itself (see [Ho03, §2.2] for more details). This approximation is based on the derivation by Bakoblu [Bak90] and follows the explanation by Ho [Ho03, §4.2]. The FO4 delay is estimated for a particular process geometry using the heuristic $\text{FO4} = 360 \cdot f$, where f is the feature size in μm, producing a delay in ps [HMH01].

The ITRS provide figures for RC delay, which are summarised for a range of process geometries in Table 9.3; their method of calculation for the RC delay is described in [Int12b, §7.3]. The M1 half pitches of 68 nm and 26.76 nm are the closest matching geometries of the processing chip and interposer respectively and the data for them are taken to estimate wire delay. Using these figures and the above formula for wire delay yields 155 ps/mm for the 28 nm processing chip and 89 ps/mm for the 65 nm interposer chip.

Parameter	Value	Note
Process geometry	28 nm	2013 prediction [Int12a, Tab. OCTC-2C]
FO4 delay	11 ps	see §9.4.1 [p. 186]
Economical chip sizes	80 mm to 140 mm^2	cost-performance processor [Int12a, Tab. ORTC-2C]
Metal layers	8	M1 logic; M2, 7 & 8 power & clock; M3-M6 wiring
Interconnect wire pitch	125 nm	for the global interconnect [Int12b, Tab. INTC6]
Repeated wire delay	155 ps/mm	see §9.4.1 [p. 188]
Processor area	0.10 mm^2	see §9.4.2 [p. 190]
Switch area	0.05 mm^2	see §9.4.2 [p. 190]
IO pad dimensions	45×225 μm	see §9.4.1 [p. 186]
Wires per link	18	1 control, 8 data per direction, see §9.4.1 [p. 188]
Clock rate	1 GHz	see §9.4.1 [p. 186]

Table 9.1.: Parameters for the processing chip model.

Parameter	Value	Note
Process geometry	65 nm	†
FO4 delay	24 ps	see §9.4.1 [p. 186]
Metal layers	4	†, M1 & M2 power & ground; M3 & M4 wiring
Interconnect wire pitch	2 μm	†, 333 half-shielded wires/mm
Repeated wire delay	89 ps/mm	see §9.4.1 [p. 188]
Microbump pitch	45 μm	†, 493.83 bumps/mm^2
TSV pitch	210 μm	†, 22 TSVs/mm^2
C4 bump pitch	210 μm	†, 22 bumps/mm^2
Wires per link	10	1 control, 4 data per direction, see §9.4.1 [p. 188]

Table 9.2.: Parameters for the interposer. † parameters based on Xilinx Virtex 7 FPGA package [Ram11], which integrates 4 28 nm FPGA ‘slices’ on a 775 mm^2 interposer.

Process geometry (M1 $\frac{1}{2}$ pitch)	Minimum global wire pitch (nm)	RC delay (ps/mm)	ITRS edition
150	670	not given	2001 [Int01]
90	300	96	2005 [Int05]
68*	210	168	2007 [Int07]
45	154	385	2010 [Int10b]
37.84	114	621	2011 [Int11]
26.76*	81	1,115	2012 [Int12b]

Table 9.3.: ITRS data for global wires. The rows marked with a * are the ones used to determine the wire delay for the processing chip and interposer.

Type	Typical capacity (MB)	Cell area factor (F^2)	Area efficiency	Current process geometry (nm)	Density (KB/mm^2)	Cycle time (ns)
SRAM	<8	140	70%	28	778.51	0.5
eDRAM	1 - 64	50	60%	28	1,868.42	1.3
Comm-DRAM	>64	6	60%	40[a]	7,629.39	30[b]

Table 9.4.: Comparison of contemporary memory technologies, with figures from the 2012 ITRS report [Int12c, Tab. SYSD3a]. The area factor is a multiple of square half pitch units, the number given by the process geometry. The area efficiency is the proportion of area in a memory array that is occupied by storage cells.

[a]Although not reported by the ITRS, high volume commodity DRAM lags commodity logic by several process geometries.
[b]Random cycle time (t_{RC}) from a 1Gb Micron DDR3 device [Mic12a].

Type	Capacity (KB)	Area (mm^2)
SRAM	64	0.082
SRAM	128	0.164
SRAM	256	0.329
SRAM	512	0.658
eDRAM	128	0.069
eDRAM	256	0.137
eDRAM	512	0.274
eDRAM	1024	0.548

Table 9.5.: Selected memory capacities of SRAM and eDRAM.

9.4.2. Processor and switch components

The area A of a component at a process geometry g is estimated for a different geometry h by applying a linear scaling $A_h = A_g/(g/h)^2$ where $g \geq h$. The area of the XCore processor on a 90 nm process is estimated to be 1 mm^2; on a 28 nm process it is estimated to be 0.10 mm^2. The area of the C104 switch on a 1 μm process was approximately 40 mm^2; on a 28 nm process, it is estimated to be 0.03 mm^2.

These figures are consistent with the ARM Cortex-M0 processor [ARM12b] and the 32×32 SWIFT switch [SDTO$^+$11]. The Cortex-M0 is a simpler processor than the XCore which has a 3-stage pipeline and supports only a single hardware thread. On a 40 nm process it has an area of 0.01 mm^2 and on 28 nm an estimated area of 0.003 mm^2. The SWIFT switch has an area of 0.35 mm^2 on a 65 nm process and an estimated area of 0.06 mm^2 on 28 nm.

9.4.3. Memory

The area and access latency of memory for processing tiles is estimated using requirements data from the ITRS [Int12c, Tab. SYSD3a]. Table 9.4 gives the key characteristics of the dominant forms of memory technology, of which SRAM and eDRAM are potential choices for an implementation. Figures for SRAM and eDRAM are given for 28 nm in line with the processing chip; 40 nm DRAM is included as a baseline, which is characteristic of contemporary commodity devices.

Although density, and in particular, cycle time (the time to perform a random access) will vary with capacity and process technology, typically, memory capacity increases with process technology and these remain roughly constant between generations. In general, eDRAM is 2 to 3 times the density of SRAM and 4 to 5 times less dense than commodity DRAM. SRAM cycle time is around three times faster than eDRAM and commodity DRAM cycle time is much higher because of the specialised process on which it is produced.

Table 9.5 shows several capacities of tile memory for SRAM and eDRAM. These are selected to be similar to the area of the processor, to investigate the relative trade-offs between them.

9.5. Cost and scaling

In this section, the implementation cost and scaling with the number of tiles for various system configurations is investigated. It first looks at the general properties of the Clos and mesh networks and then at the cost of differently sized systems and memory capacities. Finally, several potential economical (*cost-performance*) chip configurations are selected and systems comprising a collection of these are outlined.

The figures presented are produced from the calculation of specific instances of the chip layouts for the chosen technology parameters and over particular ranges.

9.5.1. Inherent scalability of the topologies

A mesh is much less well connected than a Clos, which reduces the cost of implementation but significantly limits its capability and scalability. Figures 9.5 and 9.6 demonstrate how these competing aspects scale for the two networks, over a range of system sizes appropriate for the implementation.

A measure of the implementation cost is obtained from the total number of switches and links in the network. The diameter and bisection size are also presented. For a universal communication network connecting N tiles, with diameter d and a universal routing strategy to evenly distribute communication traffic, the network capacity must be at least $\Omega(Nd)$ and the bisection size at least $\Omega(N)$, in order to provide scalable per-processor throughput (see §4.1.7 [p. 53] for the properties required of a universal network).

The number of switches and links grows linearly with the number of tiles for the *mesh* and the capacity therefore grows more slowly than the potential amount of communication traffic that can be generated by the nodes. This is reflected in the bisection size. As the size of the network increases, so to does the gap between potential traffic and capacity and this is exacerbated by a faster-growing $O(\sqrt{N})$ diameter as messages take longer to reach their destination.

The capacity per node scales logarithmically with the *Clos*, matching the diameter and the potential amount of evenly distributed communication traffic. Another way to look at this is that the provision for each tile in the mesh is a single switch and $\Theta(1)$ links, and the provision for each tile in the Clos is $\Theta(\log N)$ switches and $\Theta(\log N)$ links. For the systems presented, the diameter of the Clos scales from 4 in a 2-stage network at 64 tiles, to 6 in a 3-stage network at 8,192 tiles and the capacity and bisection bandwidth are maintained. The mesh in comparison, has a diameter that matches the Clos at 64 tiles, but grows to nearly ten times that of the Clos at 8,192 tiles with a capacity and bisection size both around 4% of that required for constant throughput per node.

9.5.2. Area and wire delay

Of the processing chip

Figure 9.7 shows how the total chip area scales with the number of tiles, with different capacities of SRAM and eDRAM. Horizontal lines are drawn to mark the range of cost-performance chip sizes (80 mm^2 to 140 mm^2). Table 9.6 lists the chips that fall in this range with their dimensions and proportions of processing, interconnect and IO. This highlights a 30% to 40% interconnect overhead in the Clos-based system compared to the mesh.

Tables 9.7 and 9.8 list the length of interconnect wires and their delays in both chips. For the Clos, these are listed for the different stages of the topology and they vary from less than 1 ns on 3 mm to

5 mm wires and between 1 ns and 2 ns on 5 mm to 10 mm wires. For the mesh, wire lengths between switches vary from 1.5 mm to 3.5 mm with delays less than 1 ns.

Of the components of the processing chip

Figure 9.8 shows the total area of the switch, wire and IO components of the chip as a percentage of the total area. This shows how they scale with respect to the number of components they connect, highlighting the difference between the Clos and the mesh in the rate network resources are invested with increasing size. Wire areas for systems with 64 KB and 512 KB SRAM memories are included to show the variation between the smallest and largest memory area.

The total switch area in the Clos is calculated as the total area occupied by the switch groups. Although the number of switches and links per-tile is proportional to the number of stages and the growth of additional components is logarithmic in the number of tiles, the area grows more quickly than this due to the increasing inefficiency of larger switch groups. The IO remains relatively constant per tile. Overall, for the cost-performance chips, the interconnect occupies between 5% and 10% of the die area.

In the mesh, switch area remains constant per tile and the wire grows slowly because of the decreasing ratio of switches on the edges of the mesh to those in the middle. The proportion of IO diminishes as the number of tiles increases since the number of external connections diminishes relative to the number of tiles. Overall, for the cost-performance chips, the mesh interconnect occupies 2% to 3% of the die area.

Of the interposer

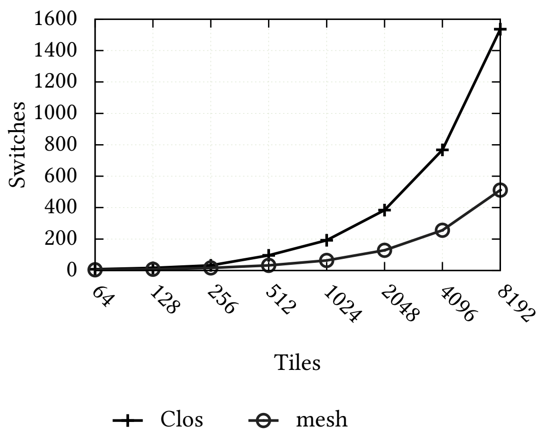
Tables 9.9 and 9.10 list potential systems that can be constructed by integrating the cost-performance chips listed in Table 9.6 on a silicon interposer. For the Clos system, this lists the percentage area occupied by the wiring channel (between 2% and 50%) and the minimum and maximum wire delays (1 ns to 10 ns). These correspond to the width and the width plus the height of the channel. For the mesh, the wire delay is a constant 0.09 ns for all systems since the chips are tiled and the distance between adjacent pads is fixed. The plots in Figure 9.9 summarise the systems presented in the tables with the total area of the interposer.

From the systems listed in Tables 9.9 and 9.10, one SRAM and one eDRAM configuration is selected for either network that provides a good trade-off between the number of processors and the total amount of memory. These are used for the performance evaluation in the next chapter.

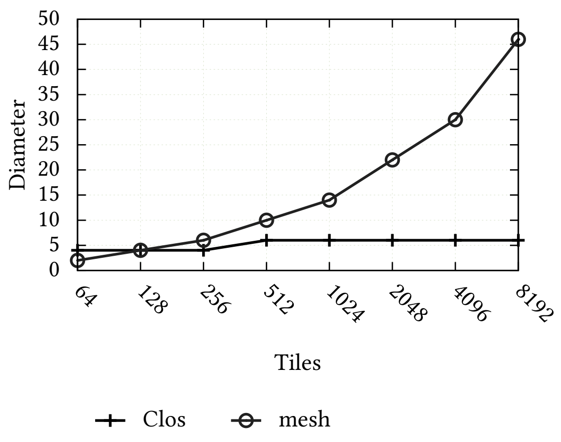
9.5.3. Summary

The cost of a universal Clos interconnect on-chip is around 10% of the available area, and off-chip, using silicon interposers to provide high density wiring, it is around 30% of the interposer area. In the system overall, this represents a 20% to 30% investment in a universal interconnect, compared to around 5% with a mesh network.

Is this then a reasonable price to pay? On the one hand, it is *unavoidable* since universal communication networks are essential to provide structural independence and therefore a general-purpose system and the results presented indicate that, even with current technology, these costs are not excessive. On the other hand, it seems almost obvious that a significant proportion of the operation of a large number of processors with *fine-grained* communication will be concerned with moving data between processors. Therefore, an investment of only 5% of a machine's resources towards this aspect does not correspond well to the demands placed on it.

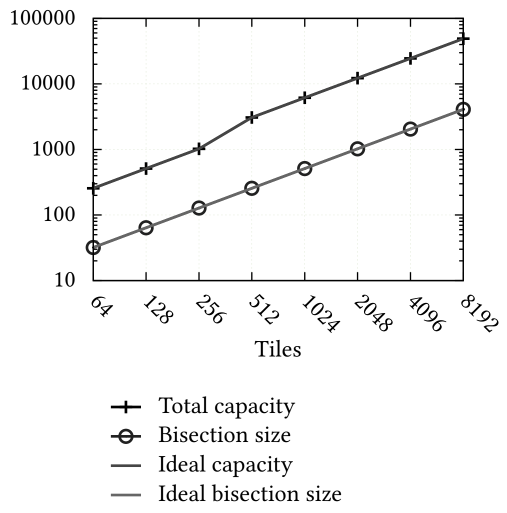


(a) switches

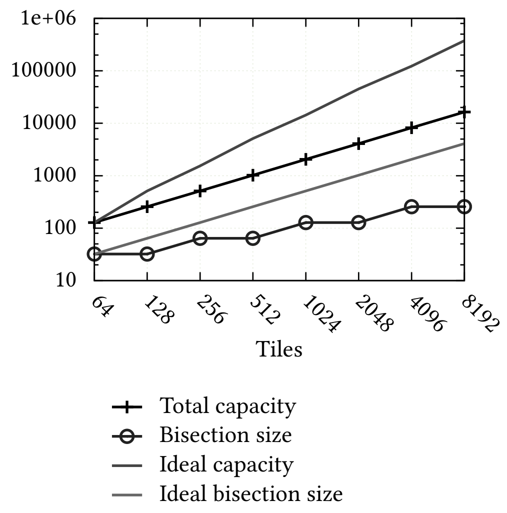


(b) diameter

Figure 9.5.: Log-linear plots showing how the number of switches (a) and network diameter (b) scales as system size increases for the Clos and mesh networks built with 32 × 32 crossbar switches.



(a) Clos



(b) mesh

Figure 9.6.: Log-log plots showing how the capacity (the total number of links in the network) and bisection size scales for Clos and mesh networks built with 32 × 32 crossbar switches. The 'ideal' values of these are the requirements of a universal network with constant throughput-scaling per node.

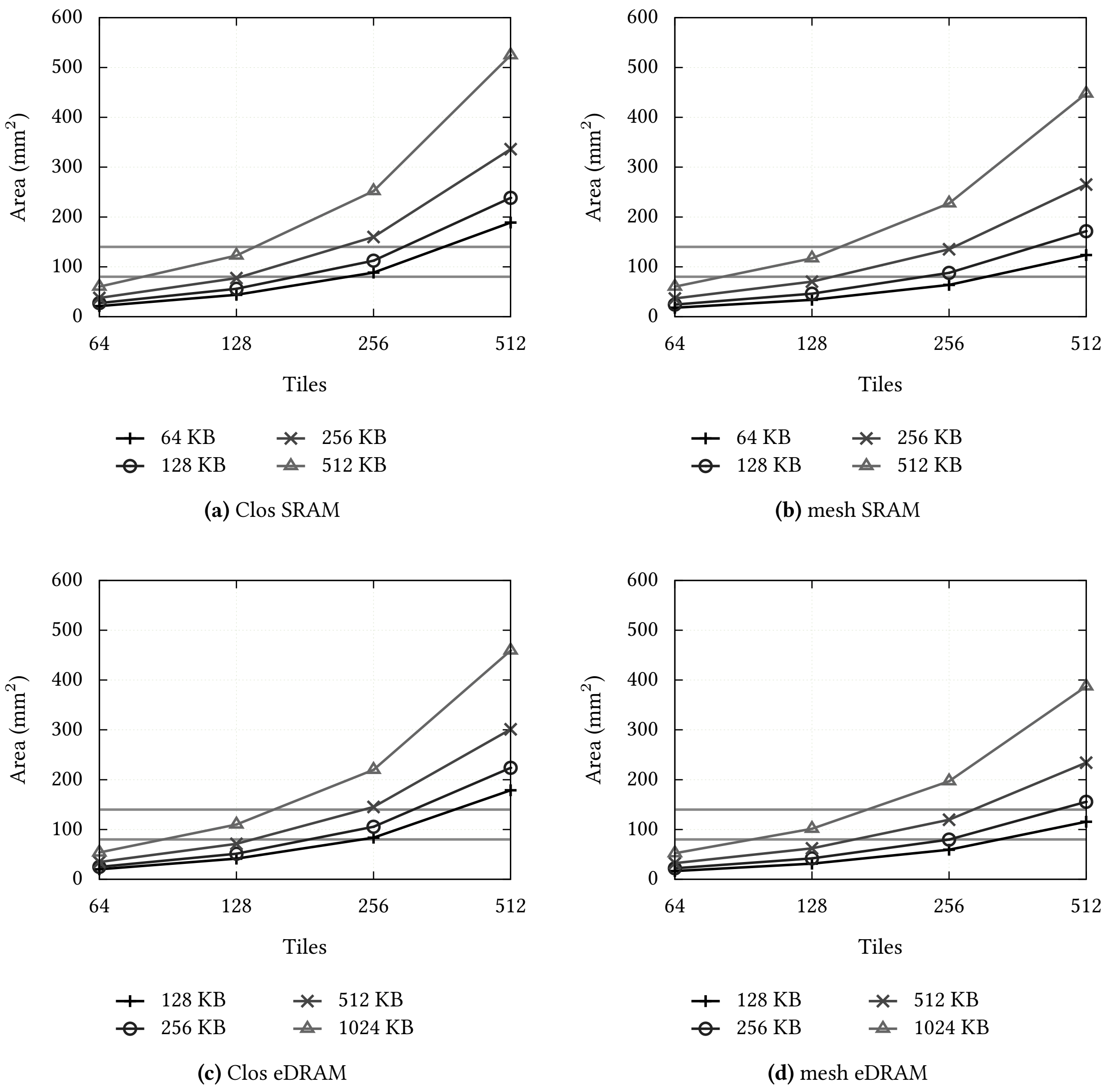
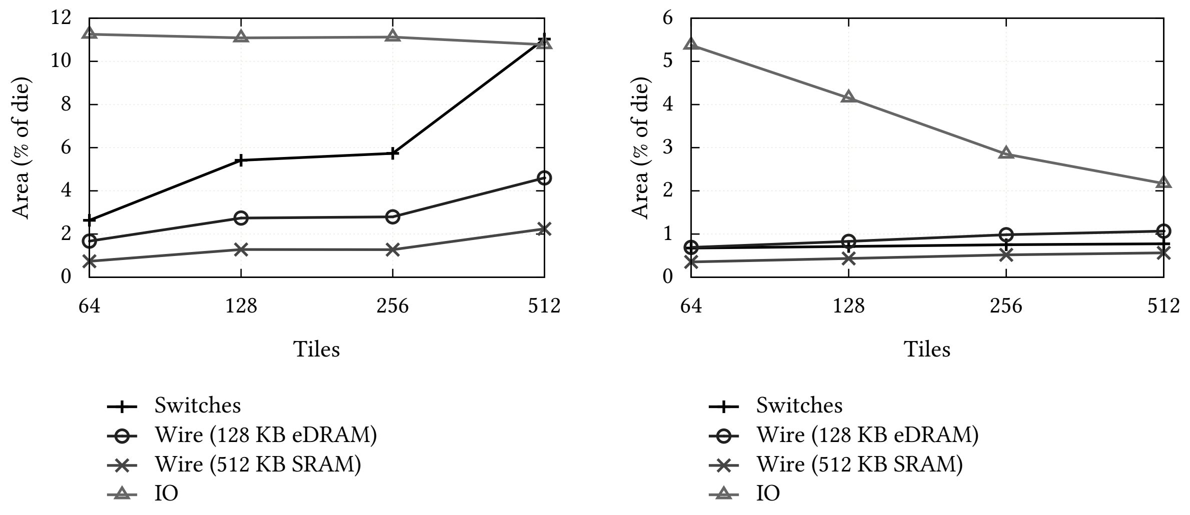


Figure 9.7.: Log-linear plots of the total chip area as a function of the numbers of tiles. The grey horizontal lines indicate the range of economical (*cost-performance*) chip sizes, from 80 mm^2 to 140 mm^2.



(a) Clos

(b) mesh

Figure 9.8.: Log-linear plots of the total area of the switches and wiring components as a percentage of the total die area. Switch area is calculated as the sum of the switch group area in the Clos. This adds an overhead to the otherwise logarithmic scaling of switch and wire area. Both components remain constant in the mesh, apart from a small convergent growth in the wire area due to a decreasing ratio of switches on the edges of the mesh to those in the middle.

Num. tiles	Mem. type	Tile mem. (KB)	Total mem. (MB)	Chip dims. (mm)	Processing area (%)	Intercon-nect area (%)	IO area (%)
				Clos			
256	SRAM	64	16	10.40 × 8.51	52.71	8.42	30.27
256	SRAM	128	32	11.34 × 9.91	60.21	7.64	23.78
128	SRAM	512	64	8.10 × 15.15	79.01	5.12	11.09
256	eDRAM	128	32	10.13 × 8.25	51.59	8.66	31.02
256	eDRAM	256	64	11.13 × 9.47	57.55	7.81	26.22
128	eDRAM	1024	128	7.80 × 14.11	75.37	5.34	14.40
				mesh			
512	SRAM	64	32	7.98 × 15.50	75.44	1.81	7.86
256	SRAM	128	32	9.38 × 9.38	77.00	1.37	7.37
256	SRAM	256	64	11.63 × 11.63	81.22	1.02	4.79
128	SRAM	512	64	7.77 × 15.07	82.84	0.64	4.15
512	eDRAM	128	64	7.72 × 14.98	74.65	1.90	8.41
256	eDRAM	512	128	10.93 × 10.93	80.10	1.11	5.42
128	eDRAM	1024	128	7.24 × 14.03	81.65	0.70	4.78

Table 9.6.: Chip configurations that lie in an economical (*cost-performance*) range of areas, as illustrated by Figure 9.7.

Num. tiles	Mem. type	Tile mem. (KB)	Tile to stage-1 wire	Stage 1 to 2 wire	Stage 2 to off-chip wire
256	SRAM	64	3.51 mm/0.57 ns	7.26 mm/1.18 ns	7.82 mm/1.27 ns
256	SRAM	128	4.21 mm/0.69 ns	8.65 mm/1.41 ns	9.22 mm/1.50 ns
128	SRAM	512	7.06 mm/1.15 ns	10.87 mm/1.77 ns	11.15 mm/1.81 ns
256	eDRAM	128	3.38 mm/0.55 ns	7.00 mm/1.14 ns	7.56 mm/1.23 ns
256	eDRAM	256	3.99 mm/0.65 ns	8.22 mm/1.34 ns	8.78 mm/1.43 ns
128	eDRAM	1024	6.54 mm/1.06 ns	10.08 mm/1.64 ns	10.36 mm/1.69 ns

Table 9.7.: Global wire lengths and their delays for the selected Clos chips.

Num. tiles	Mem. type	Tile mem. (KB)	Switch-to-switch wire
512	SRAM	64	1.71 mm/0.28 ns
256	SRAM	128	2.06 mm/0.33 ns
256	SRAM	256	2.62 mm/0.43 ns
128	SRAM	512	3.48 mm/0.57 ns
512	eDRAM	128	1.64 mm/0.27 ns
256	eDRAM	512	2.45 mm/0.40 ns
128	eDRAM	1024	3.22 mm/0.52 ns

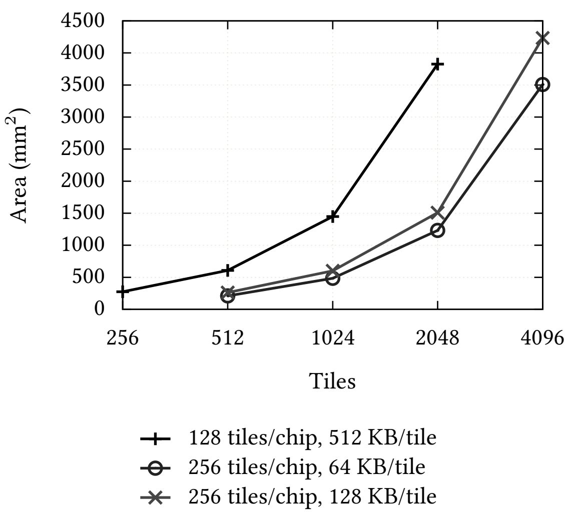
Table 9.8.: Global wire lengths and their delays for selected mesh chips.

Num. chips	Total procs.	Total mem. (MB)	Interposer dims. (mm)	Interposer wiring area (%)	Min. wire delay (ns)	Max. wire delay (ns)
256 tiles/chip, 64 KB SRAM/tile						
2	512	32	24.64 × 8.51	14.06	0.36	1.08
4	1024	64	28.48 × 17.02	24.34	0.72	2.16
8	2048	128	36.16 × 34.04	38.33	1.44	4.31
16	4096	256	51.52 × 68.08	53.81	2.88	8.63
256 tiles/chip, 128 KB SRAM/tile						
2	512	64	26.52 × 9.91	11.22	0.36	1.08
4	1024	128	30.36 × 19.82	19.60	0.72	2.16
8	2048	256	38.04 × 39.64	31.29	1.44	4.31
16	4096	512	53.40 × 79.28	44.58	2.88	8.63
128 tiles/chip, 512 KB SRAM/tile						
2	256	128	18.12 × 15.15	2.69	0.18	0.54
4	512	256	20.04 × 30.30	4.86	0.36	1.08
8	1024	512	23.88 × 60.60	8.15	0.72	2.16
16	2048	1024	31.56 × 121.20	12.34	1.44	4.31
256 tiles/chip, 128 KB eDRAM/tile						
2	512	64	24.10 × 8.25	14.83	0.36	1.08
4	1024	128	27.94 × 16.50	25.59	0.72	2.16
8	2048	256	35.62 × 33.00	40.14	1.44	4.31
16	4096	512	50.98 × 66.00	56.10	2.88	8.63
256 tiles/chip, 256 KB eDRAM/tile						
2	512	128	26.10 × 9.47	11.93	0.36	1.08
4	1024	256	29.94 × 18.94	20.80	0.72	2.16
8	2048	512	37.62 × 37.88	33.11	1.44	4.31
16	4096	1024	52.98 × 75.76	47.02	2.88	8.63
128 tiles/chip, 1024 KB eDRAM/tile						
2	256	256	17.52 × 14.11	2.98	0.18	0.54
4	512	512	19.44 × 28.22	5.38	0.36	1.08
8	1024	1024	23.28 × 56.44	8.98	0.72	2.16
16	2048	2048	30.96 × 112.88	13.50	1.44	4.31

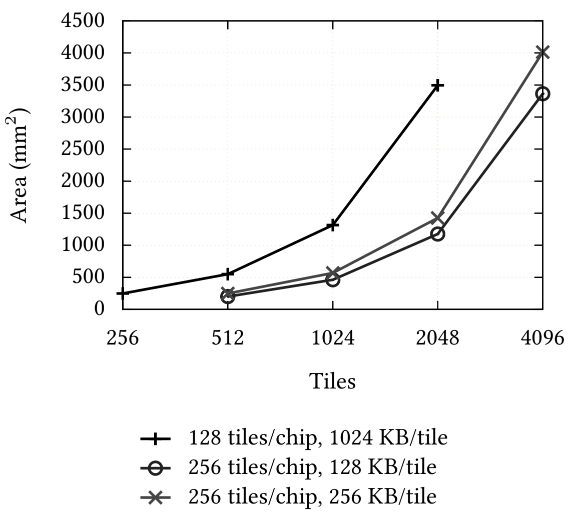
Table 9.9.: Potential multi-chip Clos system configurations integrated on a silicon interposer. The minimum and maximum wire delays are based on the width and the width plus the height respectively of the wiring channel. Delays are shared between different systems with the same number of tiles per chip, since the pitch of off-chip wires is greater than the dimension of the chip and so they are separated along the channel with gaps. The highlighted systems are used for performance evaluation in the next chapter.

Num. chips	Total procs.	Total mem. (MB)	Interposer dims. (mm)	Wire delay (ns)
512 tiles/chip, 64 KB SRAM/tile				
2	1024	64	7.98 × 31.00	0.09
4	2048	128	15.96 × 31.00	0.09
8	4096	256	15.96 × 62.00	0.09
16	8192	512	31.92 × 62.00	0.09
256 tiles/chip, 128 KB SRAM/tile				
2	512	64	9.38 × 18.76	0.09
4	1024	128	18.76 × 18.76	0.09
8	2048	256	18.76 × 37.52	0.09
16	4096	512	37.52 × 37.52	0.09
256 tiles/chip, 256 KB SRAM/tile				
2	512	128	11.63 × 23.26	0.09
4	1024	256	23.26 × 23.26	0.09
8	2048	512	23.26 × 46.52	0.09
16	4096	1024	46.52 × 46.52	0.09
128 tiles/chip, 512 KB SRAM/tile				
2	256	128	7.77 × 30.14	0.09
4	512	256	15.54 × 30.14	0.09
8	1024	512	15.54 × 60.28	0.09
16	2048	1024	31.08 × 60.28	0.09
512 tiles/chip, 128 KB eDRAM/tile				
2	1024	128	7.72 × 29.96	0.09
4	2048	256	15.44 × 29.96	0.09
8	4096	512	15.44 × 59.92	0.09
16	8192	1024	30.88 × 59.92	0.09
256 tiles/chip, 512 KB eDRAM/tile				
2	512	256	10.93 × 21.86	0.09
4	1024	512	21.86 × 21.86	0.09
8	2048	1024	21.86 × 43.72	0.09
16	4096	2048	43.72 × 43.72	0.09
128 tiles/chip, 1024 KB eDRAM/tile				
2	256	256	7.24 × 28.06	0.09
4	512	512	14.48 × 28.06	0.09
8	1024	1024	14.48 × 56.12	0.09
16	2048	2048	28.96 × 56.12	0.09

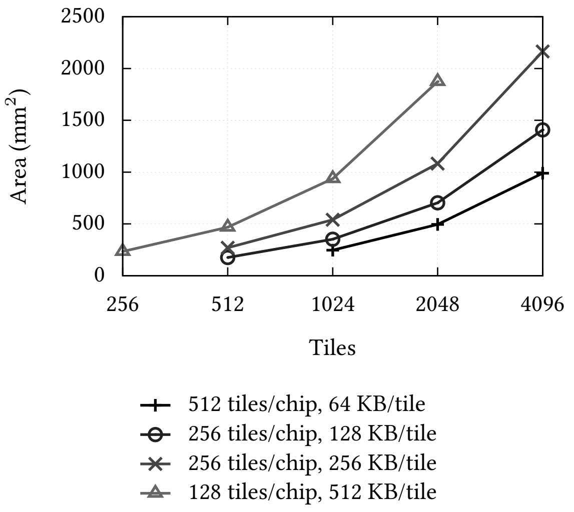
Table 9.10.: Potential multi-chip mesh system configurations integrated on a silicon interposer. In all of the configurations, a 1 mm separation between is assumed between the pads of adjacent chips, leading to a constant wire-delay value. The highlighted systems are used for performance evaluation in the next chapter.



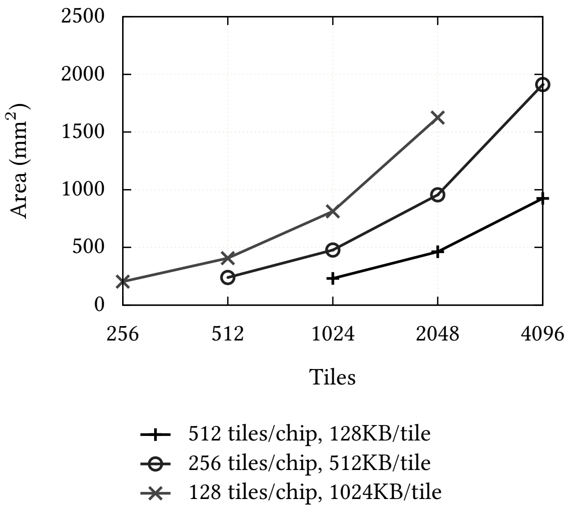
(a) Clos SRAM systems



(b) Clos eDRAM systems



(c) mesh SRAM systems



(d) mesh eDRAM systems

Figure 9.9.: Log-linear plots of the total area of the interposer for different types and numbers of processing chips for the Clos and mesh networks with both SRAM and eDRAM memories, summarising the systems listed in Tables 9.9 and 9.10.

Processor	**Dhrystone performance (DMIPS/MHz per core)**	**Process geometry (nm)**	**Area (mm^2)**	**Normalised area (at 40 nm)**
Intel Sandy Bridge	7.50	32	$\sim$15	$>$1,000
ARM Cortex A9	2.50	65	$<$1.5	57
XMOS XS1-L	0.96	90	$<$1	20
ARM Cortex M0	0.84	40	$<$0.01	1

Table 9.11.: A comparison of area and sequential capability of different processors. DMIPS are *Dhrystone MIPS*, the score obtained from the integer Dhrystone benchmark [Wei84] divided by 1,757, which is the score obtained by the VAX 11/780, a 1 MIP machine [Yor02]. Extrapolation to different process geometries is done with a linear scaling. The comparison does not consider the width of data paths or additional functionality such as vector units.

9.6. Discussion

9.6.1. Architectural choices

Sequential capability

Conventional sequential processors employ a range of techniques to increase instruction throughput on single streams of instructions, such as *superscalar execution* to dynamically exploit parallelism within the instruction stream to execute operations simultaneously, *pipelining* to increase the number of instructions that can be processed at once, *speculative execution* to reduce the costs of conditional branches and *out-of-order execution* to fill delays.

There is a question of what the level of sequential capability of a processor to use in a parallel system shall be, since this comes at the expense of silicon area and predictability.

The effectiveness of techniques to improve sequential performance has diminished in terms of area (and consequently power) as their complexity has grown. There are inherent limits to parallelism in instruction streams and the benefits of branch prediction, and to the depth and size of execution pipelines and reorder buffers. The trend of diminishing returns from these architectural optimisations is described by *Pollack's rule* [Pol99]. It states that performance increase is roughly proportional to the square root of increase in complexity, so that a doubling of area delivers only 40% more performance. This is evident in historical scaling of CPU technology and performance [DKM$^+$12]. Borkar observes that Pollack's rule suggests a better use of silicon area is with more cores as it has the potential to increase performance linearly with complexity and power [Bor07], adding further support for the approach proposed in this chapter.

The increased complexity of processors also results in less predicable performance and the likelihood of rare events such as branch mispredictions or cache misses increases exponentially with the number of processors and any delay is magnified when it affects a synchronisation.

A simple microprocessor architecture therefore allows large numbers of processors to be integrated together to deliver higher performance per unit area and the performance of each processor will be predictable or even deterministic. Moreover, with close proximity of memory, access latency is minimised close to that of regular arithmetic, logic and branching operations, then many of the techniques found in conventional sequential processors, in particular caching, are no longer necessary. Indeed, with the UPA and sire, when large memories and caching schemes are absolutely necessary, they can be expressed as *software components* and *emulated* with only a small overhead. This was discussed in Chapter 7 with some simple examples; the next chapter provides some evaluation.

A rough comparison of sequential capability

To illustrate the trade-off between sequential capability and silicon area, Table 9.11 compares several contemporary processors. The *Intel Sandy Bridge* core is 64-bit, floating-point capable, pipelined, superscalar and out-of-order. Estimates for its area are based on [YKM+11]. The *ARM Cortex-A9* core is 32-bit and also floating-point capable, pipelined, superscalar and out-of-order, but to a lesser extent than Sandy Bridge since it is designed as an embedded processor for low-power applications [ARM12a]. *ARM Cortex M0* core is a 32-bit minimal implementation of the ARM architecture [ARM12b]. The sequential capability of the *XMOS XS1-L* core is similar to the Cortex M0.

The figures indicate that a conventional complex sequential processor can deliver up to around a factor of 8 times the serial performance when compared to very simple processor, although in general the gain is likely to be less than this. The cost of additional sequential capability is around a factor of 50 to 1,000 in silicon area, less predictable execution and greater power requirements. An array of the simple processors therefore only needs to find an average parallel speedup of 4 to 8 to break even with the conventional processor. With N processors, there is potential for an N-fold performance improvement, far beyond what a conventional processor can deliver.

Heterogeneity

Another important issue to discuss here is *heterogeneity*. Heterogeneous architectures employ different types of processors to support different workloads. In the last 5 to 10 years there has been an emerging trend with heterogeneity, particularly in HPC with the use of accelerator devices. Predominantly, this has been concerned with GPUs as they have become more programmable, but FPGAs have also had some attention.

A heterogeneous system is, by its very nature, specialised to a particular set of applications (or even just one application) because its capabilities are apportioned in specific quantities. This is a good approach when the application of the machine is well defined, as it is in computer graphics with GPUs, but problems arise when the system is applied to different domains, as GPUs are in HPC.

Specialisation poses a significant problem for the portability of programs because programmers have to explicitly consider the details of a specific heterogeneous machine to obtain good performance. It is therefore unlikely that a program written for one heterogeneous machine will execute with the same efficiency on a different machine with a different specialisation. A further effect of specialisation is that it makes it difficult to develop a single programming model, making the development of programming languages challenging.

In general, a *homogeneous* architecture provides a better balance over changing workloads and maintains a single *architecture*. However, there is potential for heterogeneity to be employed in different *implementations* of a particular architecture. In the case of the UPA, a specific implementation could be specialised to support a particular subset of the programs that can be compiled to it. This subset could, for example, be sequential programs and the architecture could be specialised with one or a small number of processors that have good sequential performance, at the expense of some area (this kind of approach is motivated by the investigation into the sequential capabilities of a UPA implementation in the next chapter). Specialisation in this way will rely on extensions to a compiler that can exploit it, but by maintaining a single architecture, programming languages, compilers and programs remain portable and can develop with different implementations and technologies.

9.6.2. Modelling and estimates

Chip layout

The layout of the Clos allowed a simple VLSI model to be used to obtain estimates for area of the die, components of the system contributing to the area and lengths of wires. These estimates are likely to be conservative since the design does not aggressively utilise the available area. In particular, the switch groups and dedicated wiring channels introduce unused area into the designs and the inefficiency of these grows with system size.

A more efficient layout could be used, to make a better use the available area and metal layers for global interconnections. Theoretical results for VLSI layouts of fat tree variants that, similar to the Clos networks studied, for example those by Greenberg [GL88] and DeHon [DeH00], could potentially be applied or extended to do this.

The CACTI tool

CACTI [TMHAJ08] is a tool built by HP labs to estimate the area and performance of memories. It provides a range of configurable parameters making it useful for design-space explorations and relative studies and therefore, in principle, ideal to produce figures for area and delay of different memory configurations for UPA tiles. However, it was found in the course of this investigation to produce results that were highly inconsistent with the ITRS and other published data.

The area efficiency of CACTI designs in many cases were very low and varied significantly. For example, the area of 40 nm commodity DRAM memories from 1 MB to 1 GB increased from 4.60% to 31.44% area efficiency. This indicates that the model employed by CACTI is perhaps geared towards larger memory capacities or is ineffective in generalising between different technologies. Supporting this experience, a previous study by Agrawal and Sherwood on SRAMs found that CACTI overestimated area and delay by 20% to 60% [AS07]. The consequence of these issues were that CACTI was unsuitable to be used as part of the model. Instead, a simpler approach was taken based on density and area efficiency, which remain reasonably consistent between different implementations.

High-level design studies

The experience with the CACTI tool highlights an important aspect of the work presented in this chapter. There are a multitude of issues associated with VLSI systems, from high-level design to low-level implementation, and particularly as process technologies descend into deep sub-micron geometries. This makes it difficult to generalise and extrapolate between different geometries, especially with complex models that focus too much on particular issues, even though there is a great degree of continuity in the high-level capabilities of silicon devices.

A more robust approach in high-level design studies is to choose the simplest set of parameters to obtain a best generalisation, because despite the challenges faced by system designers and fabricators, ways continue to be developed to maintain broad scaling trends such as transistor density. The proposed implementation model takes this approach, for example, with the placement of components and wiring, and in the performance scaling of memory and global wires. The result is a definite design to evaluate, leaving others to refine.

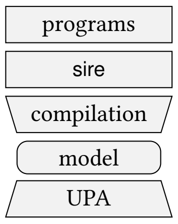
9.6.3. Future technology

The key strengths of the UPA is its simplicity and modularity. This chapter has demonstrated that it can be implemented effectively with current technologies but it also suggests that it will continue to be with future technologies, a crucial aspect of its general-purpose nature and longevity.

Of particular interest are two emerging technologies that could significantly increase the density of integration and connectivity in large multi-chip systems.

- *3D integration* allows chips to be stacked on top of one another and to be connected between chips with through-silicon vias to provide flexible high-density connections between chips. It is being developed already to increase bandwidth and density in memories. A stackable DRAM die is already commercially available from Tezzaron [Tez10] and Elpida are producing packages containing stacked DRAM chips [Elp11]. As well, standards are emerging for embedded systems with Wide I/O [JED11] and in the desktop and server space with the Hybrid Memory Cube [Hyb13].

- *Optical interconnections* can potentially be made with a silicon chip [Mil10] that would allow a large number of signals at high bandwidth to be carried down a single optical fibre. This has the potential to significantly improve the density of connections between chips, to move away from the enormous constraints of conventional electrical connections on PCBs. Moreover, if 3D integration was used to build stacks of chips of the nature described in this chapter, optical interconnections may be the only currently feasible technology able to source and sink the amount of data that would be consumed and produced by the large collection of processors contained in them.



CHAPTER 10. PERFORMANCE EVALUATION OF THE UPA AND SIRE

In this chapter, the performance of the implementation of the UPA proposed in Chapter 9 and its ability to support the execution of sire programs is explored. Evaluated is conducted with high-level software simulations of the UPA with a performance model for the proposed implementations. Because of the high-level nature of the models, the results produced by the simulation can be taken as only an *indication* of what could be expected from the systems examined. Therefore, a comprehensive evaluation and comparison against contemporary architectures is beyond of the scope of this investigation.

The evaluation focuses on the following specific aspects of the proposals and concentrates on the *scalability* of the performance:

1. the efficiency of primitive mechanisms for parallelism, communication and abstraction to demonstrate the scalability of the language and its implementation targeting the UPA;
2. the ability of the UPA to support sequential programming techniques since this provides a way to support legacy sequential software and a means to transition from existing sequential or non-scalable machines.

The chapter begins by describing the simulation and performance model used in the evaluation. Then, for the two above aspects, the experimental methodology performance results are presented. Finally, the chapter concludes with a discussion of the results.

10.1. Simulation model

10.1.1. A network performance model

The system model presented in Chapter 9 provides values for the latency of signal transmissions along links and their bandwidth. The performance of communication depends on some additional parameters related to the latency of the switching elements in the network. The following sections describe these parameters and formulate the simple network performance model that is used in the simulation.

Network latency

Given an interconnection network represented by a graph $G = (V, E)$ whose vertices V represent nodes containing a switch and zero or more tiles, and edges E that represent communication links (see §4.1.1 [p. 41] for a description of the network model) the latency of a message sent from processor $s \in V$ to processor $t \in V$ depends on:

- t_{tile}, the latency of the link between the tile and the switch;
- t_{switch}, the switch latency;
- t_{open}, the additional latency to open a route through the switch;
- $c_{\text{contention}}$, the switch contention factor;

- $d(s,t)$, the length of the path in the network ($d(s,t) = |p(s,t)|$);
- $t_{\text{link}}(u,v)$, the latency of the link between nodes $u, v \in V$;
- t_{serial}, the serialisation latency, which is determined by the message length and channel bandwidth, such that if s and t are on the same chip, then $t_{\text{serial}} = t_{\text{serial-intra}}$ and if they are on different chips, then $t_{\text{serial}} = t_{\text{serial-inter}}$.

When the route between s and t is not already open, message latency is calculated as

$$t_{\text{closed}}(s,t) = 2t_{\text{tile}} + t_{\text{serial}} + (d(s,t)+1)(t_{\text{open}} + t_{\text{switch}} \cdot c_{\text{contention}}) + \sum_{\ell \in p(s,d)} t_{\text{link}}(\ell)$$

and when the route is open it is calculated as

$$t_{\text{open}}(s,t) = 2t_{\text{tile}} + t_{\text{serial}} + (d(s,t)+1) \cdot t_{\text{switch}} \cdot c_{\text{contention}} + \sum_{\ell \in p(s,d)} t_{\text{link}}(\ell)$$

Both models are comprised of the sum of four latency components: tile-to-switch, serialisation, switch traversal and link traversal. When shortest-path oblivious routing is employed, $d(s,t)$ is the minimum distance between s and t; with two-phase randomised routing, $d(s,t)$ is twice the network diameter for a Clos network, and twice the average path distance for a mesh.

Values for the parameters of the above latency are summarised in Table 10.1. The estimates in Chapter 9 are used to calculate the link and serialisation latencies and the switch latencies are estimated by fitting measurements taken with a real XMOS device called the XMP-64 [XMO10] to the above latency model.[1] The XMP-64 consists of 16 XS1-G4 [XMO12b] chips that each contain 4 processing tiles and a switch. Measurements taken for the other parameters are included for comparison, with the following caveats: on-chip there is no serialisation latency since the tiles are connected to the switch with 8-bit links, and off-chip, the switches are connected in a 4D hypercube topology with two sets of 5-wire links that use a 1-in-5 coding scheme.[2] This uses 5 wires to deliver 1 byte every 4 cycles (the serialisation latency), and operating at 400 MHz, this provides a bandwidth of 400 Mb/s in each direction [XMO12a, §3.1].

Switch contention

When the network is under load, many communications occur simultaneously and in many cases these will compete for resources. This situation occurs at switches where messages contend for particular links, delaying one another and increasing the time taken to traverse a switch. With two-phase randomised routing the distribution of load on switches will be almost always uniform and even. This allows a simple probabilistic model to be used where all messages experience a particular level of contention and a uniform overhead in switch traversal.

The following model of switch contention is based on the model described in [MTW93, Ch. 6]. A switch with n input links and n output links attempts to route tokens from each of the inputs to one of the outputs, where output links are chosen uniformly at random. A subset of these will succeed immediately in the time to open the route and transmit the token. This is called a *time slot*, t_{slot}, and is equal to the time taken to open a connection, t_O, and transmit a token across the switch, t_S. All other tokens are discarded and a new time slot begins, proceeding in the same way. The switch does not actually discard any tokens, but since the destinations in next time slot of the model are chosen uniformly at random, it captures the delay experienced by tokens that could not be delivered.

[1] For a general discussion of the methodology for taking performance measurements and other specific results with the XMP-64, please refer to [Han09].

[2] An m*-of-*n *scheme* encodes an m bit *symbol* into an n-bit *codeword* such that the Hamming weight of all the symbols is constant. This property means they can be used to signal asynchronously, for example, across clock domains, because the transmission of each symbol will change a constant number of signals. This approach also reduces power consumption and improves signal integrity.

Parameter	**Symbol**	**Value**	**XMP-64 measurement (cycles)**
Thread-to-thread latency	-	1 cycle	4
Tile-to-switch latency	t_{tile}	see §9.5.2 [p. 191]	1
Switch latency	t_{switch}	2 cycles	2
Latency to open a route	t_{open}	5 cycles	5
Switch contention factor[a]	$c_{\text{contention}}$	1.567	not measured
Serialisation latency intra-chip	$t_{\text{serial-intra}}$	0 cycles	0
Serialisation latency inter-chip	$t_{\text{serial-inter}}$	2 cycles	4
Link latency	t_{link}	see §9.5.2 [p. 191]	2 on-chip, 3 off-chip

Table 10.1.: Parameters for the network performance model. The switch latencies are based on measurements made with the XMP-64; the other measurements are included for comparison.

[a]This is under uniform load (see §10.1.1 [p. 206]); when then network is uncongested, $c_{\text{contention}} = 1$.

The expected delay of a packet t_{delay} is given by

$$t_{\text{delay}} = t_{\text{slot}} \cdot \sum_{i=1}^{\infty} i \cdot p(i)$$

where i is the slot number and $p(i)$ is the probability of success on the i^{th} attempt. Since $p(i) = p(\text{failure})^{i-1} \cdot p(\text{success})$ then

$$t_{\text{delay}} = t_{\text{slot}} \cdot p(\text{success}) \cdot \sum_{i=1}^{\infty} i \cdot p(\text{failure})^{i-1} \tag{10.1.1}$$

At the start of a time slot, the probability of an output link being free is

$$P(\text{success}) = \left(1 - \frac{1}{n}\right)^n$$

and hence the probability of it being used is

$$P(\text{failure}) = 1 - P(\text{success}) = 1 - \left(1 - \frac{1}{n}\right)^n$$

Then summing the series in Equation 10.1.1 yields

$$t_{\text{delay}} = \frac{t_{\text{slot}}}{P(\text{failure})}.$$

The peak contention of a 32×32 crossbar switch, with tokens on all input links, is then $t_{\text{delay}} = t_{slot}/0.638$ and hence, under uniform load, $c_{\text{contention}} = 1/0.638 = 1.567$.

10.1.2. Simulation platform

A modified version of AXE (*An XCore Emulator*)[3] is used to simulate the behaviour and performance of the parallel systems. AXE is an event-based functional simulator for the XS1 architecture that provides approximate instruction execution timing. It simulates only the state of each processor and not the behaviour of the network; communication corresponds only to writes into 'remote'

[3]The original version of AXE is open source and available from `http://github.com/rlsosborne/tool_axe`; the version developed for this work is available from `http://github.com/jameshanlon/tool_axe`.

channel ends. It is fast compared to the cycle-accurate simulator provided with the standard XMOS tools, which makes it practical to run large simulations (up 4,096 processors) for short instruction sequences.

The main modifications to AXE were to attach a latency to messages passed between threads, based on the performance model described in the previous section and a latency to instructions that access memory to reflect the performance of SRAM and eDRAM. For message passing, the effect is to assign a fixed latency to communications between each source-destination pair. AXE was also modified to boot from an executable file of the format produced by the sire compilation that contains two binary images: a *master* image that is loaded on core 0 and a *slave* image that is replicated on all other cores.

10.1.3. Assumptions

The following points outline several assumptions made in the performance model and evaluation.

- *Chosen system configurations.* In the last chapter, four system configurations from the options listed in Tables 9.9 and 9.10 were highlighted as providing a good trade-off between their various capabilities. For the purposes of this evaluation, these particular selections are representative since the various configurations would not result in significant differences in performance. One further simplification is made by choosing only the systems with SRAM memory. This is because the additional cycle of latency incurred by the eDRAM would also only marginally affect affect performance.
- *Optimistic mesh performance.* In all experiments with parallel workloads, two-phase randomised routing is used and the network is assumed to be loaded with congested switches. The model of congestion assumes that communication traffic is distributed evenly over the network, which will be true with the Clos network because the capacity scales sufficiently with the number of terminals it connects. However, capacity in the 2D mesh does not scale with the number of terminals and the consequence of this is that traffic hot-spots will occur at some switches. Since the evaluation uses the same model for both networks, it assumes an optimistic case for the 2D mesh and in reality, performance is likely to be worse in most cases, and substantially worse in some.
- *Single thread XCore performance.* An XCore processor is intended to execute a collection of threads to perform latency hiding. It has a four-stage pipeline and, to simplify the implementation, at most one instruction from any thread can be present in the pipeline at any particular time. The result of this is that each thread executes at a rate of at most one quarter of the core clock frequency. Since the evaluation does not deal with latency hiding, it is assumed in the evaluation that a thread can operate at the core frequency.

10.2. Efficiency of sire primitives

In this section, the evaluation efficiency of the *primitive aspects* of the sire language, relating to parallelism and communication, is presented. These are primitive in the sense that they are not derived from any other feature (although they are present in both the canonical and standard forms of the language).

The evaluation is performed using an experimental implementation of sire,[4] which is based on an early and undocumented version of the language. It differs in a number of respects from the version proposed in Chapter 6 but it includes the features for processes, channel connections and run-time distribution of program. It is sufficient to perform the following evaluation but the limitations of using it are discussed at the end of this chapter in §10.4.1 [p. 224].

[4]The experimental implementation of sire is available from `http://github.com/xcore/tool_sire`.

10.2.1. Methodology

There are several reasons why evaluation of sire primitives is a good approach to characterising the efficiency of the language:

1. the canonical and standard forms of sire are based on a small set of primitives so it is essential that their operation is efficient;
2. the independence of programs from the underlying network means that the performance of different components is *compositional* and the performance of a given component remains consistent whether in isolation or in composition with other components;
3. the results are general and applicable to all programs;
4. simple benchmark programs will exhibit simple behaviours and therefore produce results that can be interpreted accurately and even used to model the performance of other programs.

The primitive aspects chosen for the evaluation are:

- *call primitives*[5] relating to the *on clause* of the canonical form and *server calls* that both have conventional procedure-call semantics and are used to implement program distribution, communication and abstraction;
- *structural primitives* relating to *process replication* and *channel connections* that provide the basis for expressing and creating parallel structures.

These are integrated in a set of *microbenchmark* programs and the evaluation is based on executing each one over a range of system parameters to demonstrate efficiency and scaling. The microbenchmarks are compiled according to the process described in Chapter 8 and no special optimisations, such as inlining, are used that would bias the results.

Microbenchmarks for call primitives

The call primitives are based on a set of similar procedures that vary in the number of parameters and whether they are passed by value or by reference. Each procedure applies a binary associative operation to all of the parameters. On the following examples this operation is logical 'and' and returns the result in a variable parameter. The *pass-by-value* procedures with n input parameters have the form

$$\textbf{process}\ P(\textbf{var}\ \texttt{r},\ \textbf{val}\ v_1,\ v_2,\ \cdots,\ v_n)\ \textbf{is}\ \texttt{r} := v_1 \oplus v_2 \oplus \cdots \oplus v_n$$

and the *pass-by-reference* procedures with n input parameters have the form

$$\textbf{process}\ P(\textbf{var}\ \texttt{r},\ v_1,\ v_2,\ \cdots,\ v_n)\ \textbf{is}\ \texttt{r} := v_1 \oplus v_2 \oplus \cdots \oplus v_n$$

These provide simple benchmarks in which the execution time of a procedure call will be correlated with the number and nature of parameters and serve to highlight the costs of the remote forms of calling. Although further variation could be introduced with array parameters, their effect would be a fixed overhead for the transfer of data that is not directly related to the calling mechanism.

Let X be one of **val** or **var**, then in the scope of the above definitions, local execution corresponds to

$$\textbf{var}\ r\texttt{:}\ X\ v_1,\ v_2,\ \cdots,\ v_n\texttt{:}\ P(r,\ v_1,\ v_2,\ \cdots,\ v_n)$$

where r is the name of a variable and $v_1, v_2, \cdots, v_n$ are values or names of variables depending on the type of the parameters. This is used as a performance baseline. In the scope of the above definitions, execution on a *remote* processor x corresponds to

[5]In the definition of sire in Chapter 6, procedure calls are referred to as *instances* because this terminology is consistent with named processes. However, for simplicity, these three mechanisms are all referred to as calls in this context.

```
var r: X v1, v2, ···, vn: on x do P(r, v1, v2, ···, vn)
```

The equivalent server call corresponds to the client process in the composition

```
var r: X v1, v2, ···, vn:
s is interface(call P(var r, X v1, v2, ···, vn)):
  alt { accept P(var r, X v1, v2, ···, vn):
    r := v1 and v2 and ··· and vn } :
s.P(r, v1, v2, ···, vn)
```

The three versions of procedure calling differ primarily by the amount of state that is to be communicated in the execution of a call. The experimental evaluation will quantify their relative costs. The effects of each version can be summarised as follows, from the least communication to the most.

- *Local call*: execute a procedure locally, providing it with local parameters from registers and the stack (according to the calling convention that is described in §8.3.3 [p. 152]).
- *Server call*: invoke the execution of a procedure remotely, providing the target server with just the parameter values.
- *Remote call*: execute a procedure remotely, providing the target processor with the procedure *closure*. The closure contains the parameter values and all of the necessary program for execution. When execution is complete, the values of any updated referenced variables are sent back.

For each benchmark, the time for the local, remote or server call to complete is measured.

Microbenchmarks for structural primitives

The structural primitives are based on the creation of process arrays and formation of process structures with communication channels. The replicator benchmark is the process array

```
par [i=0 for N] skip
```

which creates N instances of the process **skip** on N different processors. Although this is seemingly simple, it is the mechanism by which collections of processes, and in particular servers, are initiated and terminated, and therefore will underpin the performance of sire programs. The canonical form of this statement is a recursive procedure that employs on clauses to distribute execution (the process by which a replicator is transformed into a parallel recursive process is described in §8.2.8 [p. 145]). The performance is measured by timing the complete execution of the replicated process.

The process structures of §7.1 [p. 101] are taken as a representative set of regular structures to characterise the basic constructs of simple message-passing algorithms. They are a 1D mesh or *pipeline*, a 2D mesh or *grid*, a binary tree and a hypercube. Each structure is used as a benchmark to observe the overhead of establishing channel connections between component processes of an array. In general, this overhead of establishing connections will not be correlated with the number of channels per component process of the structures because the sequence of connections will cause some processes to have to wait. Although this is a consequence of the algorithm rather than the implementation of the language, it is the approach that must be taken when writing sire programs, so it is important to investigate the effect that it has.

For each structure, the total time for the creation and distribution of processes, connection of channels and termination and tear-down of the structure is measured and compared with just the distribution time to quantify the channel connection cost.

The replicator and process structure benchmarks can be seen as minimal examples of highly parallel components since they only trigger the setup and teardown mechanisms, and do not perform any real

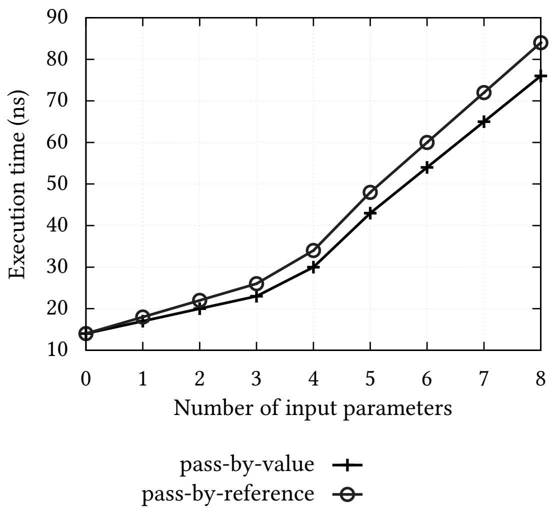


Figure 10.1.: Execution time for the local procedure-call benchmark.

computation. Their execution time therefore relates to the cost of employing parallel subroutines in a program. In a sequential analogue of procedure calling, this corresponds to measuring the overhead of parameter passing in registers and the stack, branching and linking, stack initialisation and the corresponding uninitialisation and return of results on completion.

10.2.2. Results

The following sections present the results for the call and structural primitives.

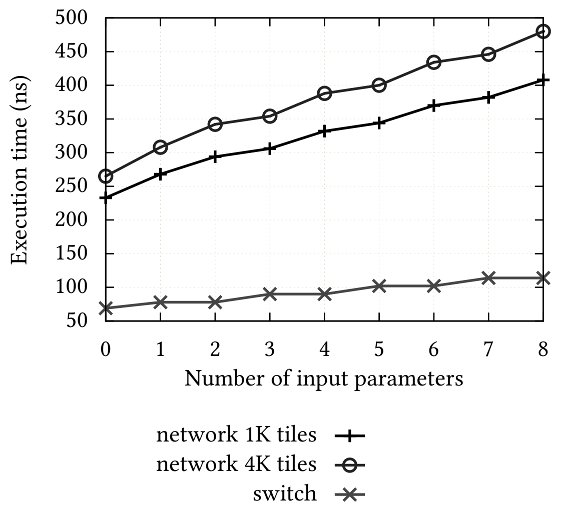
Call primitives

Figure 10.1 shows the execution time as a function of the number of parameters for the procedure-call benchmark executed locally. In this, pass-by-reference has a greater overhead because all of the values must be loaded from memory by the callee from the addresses passed to them, whereas they are available directly from registers when passed-by-value. The change in line gradient shows that after four parameters, there is an additional store instruction required to pass them using the stack.

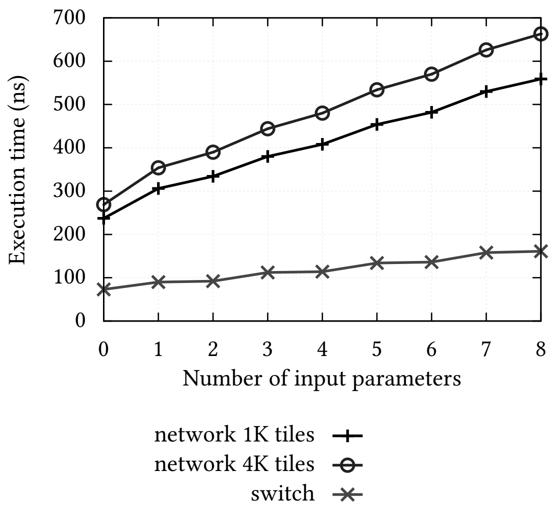
Figure 10.2 shows the performance of the server versions of the procedure-call benchmarks and Figure 10.3 shows the remote versions. In each plot, the execution time is given for both intra-switch and inter-switch communication (in the largest and smallest networks) since inter-switch latency is fixed in network of a given size due to two-phase randomised routing.

A server call is a message-passing exchange between a client and a server that consists of an initialisation where the client identifies itself, then transmission of parameter values, execution of the call by the server while the client waits, then finally transmission of any updated referenced parameters. Overall, the execution time grows linearly with the number of parameters since they are all effectively sent by value. There is an additional overhead of around 20% to 30% when they are passed by reference because the value of each referenced variable is sent back to the caller after the call is executed (the client and server sides of a server call are described in §8.5.2 [p. 172] and §8.5.1 [p. 167]).

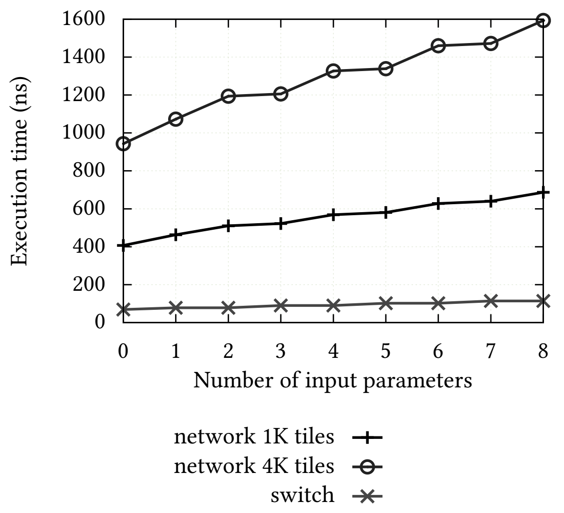
The execution time of server calls between tiles attached to the same switch is between 2 and 5 times that of a local procedure call at 70 ns to 160 ns. When the call is performed over the Clos network, this increases to between 8 to 20 times that of local calls, but in all cases, it does not exceed one microsecond, or equivalently, 1,000 local operations. When the call is performed over the 2D mesh network, the costs are much higher at between 25 and 70 times that of local calls (1 μs to 2 μs) because of the longer path lengths and consequent additional latency.



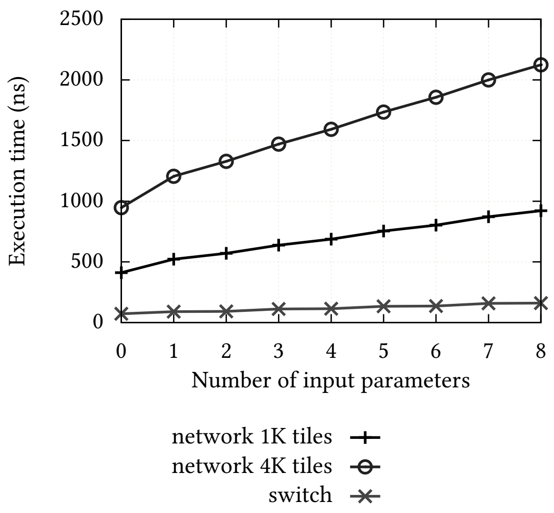
(a) Clos network, pass-by-value parameters



(b) Clos network, pass-by-reference parameters

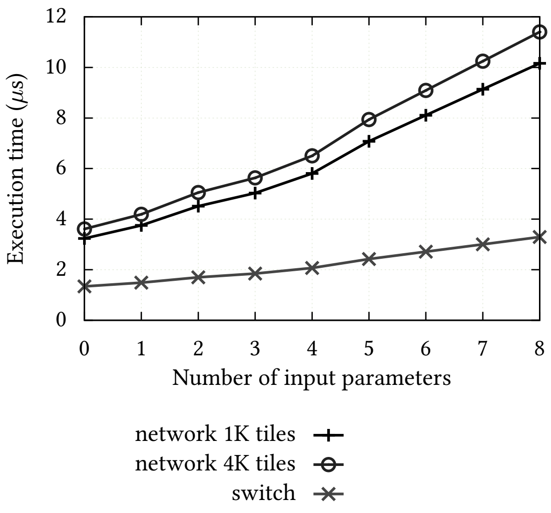


(c) mesh network, pass-by-value parameters

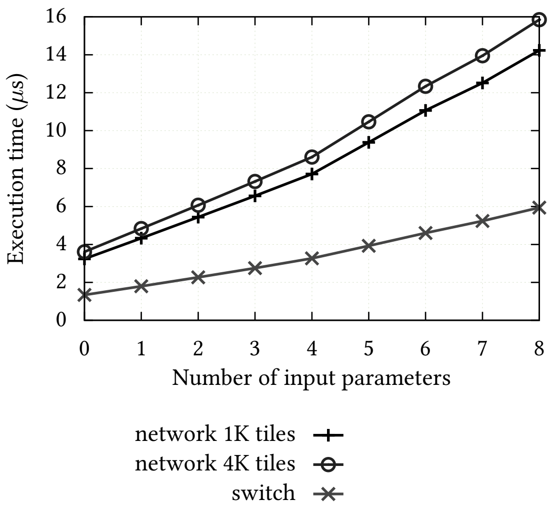


(d) mesh network, pass-by-reference parameters

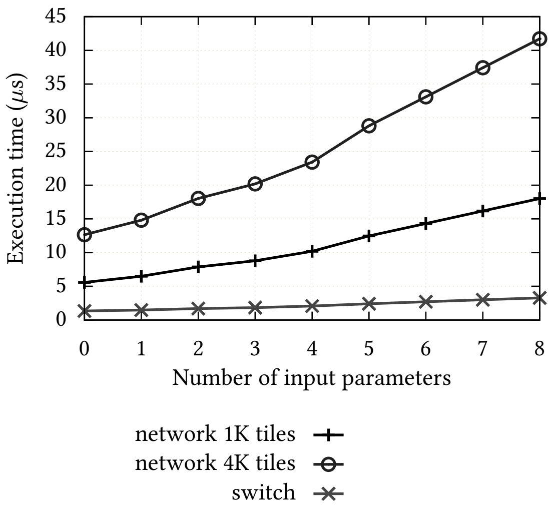
Figure 10.2.: Execution time for the server versions of the procedure-call benchmarks. In each plot, results are given for intra-switch communication and inter-switch communication in large and small networks.



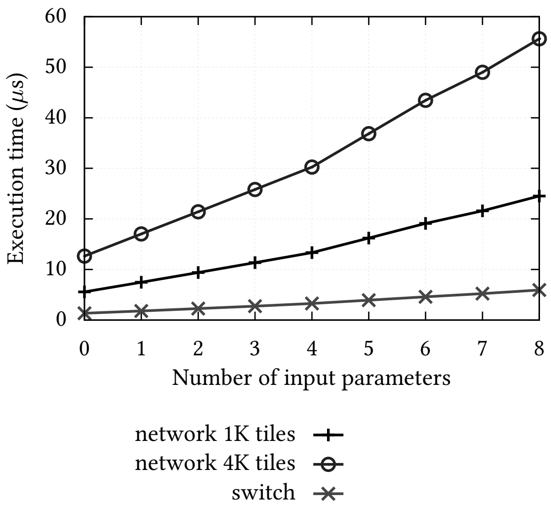
(a) Clos network, pass-by-value parameters



(b) Clos network, pass-by-reference parameters



(c) 2D mesh network, pass-by-value parameters



(d) 2D mesh network, pass-by-reference parameters

Figure 10.3.: Time for the remote execution of the procedure-call benchmarks. In each plot, results are given for intra-switch communication and inter-switch communication in large and small networks.

A remote call is similar to a server call except that the procedure (and any child procedures) to be executed are transmitted to the *host* processor as the parameters of the call. There is also overhead in initialising a new thread of execution and its associated state such as the stack (the two sides of the 'on' protocol are described in §8.4.3 [p. 158] and §8.5.4 [p. 173]). The trend is therefore similar to that of server calls, with a linear relationship effectively between the size of the call closure and the execution time.

The execution time of a remote call between tiles attached to the same switch is 40 to 100 times that of a local call at around 1 μs to 6 μs. When the call is performed over the Clos network, this increases to between 150 and 250 times that of a local call (around 4 μs to 16 μs), and when it is performed over the 2D mesh this is much higher at 500 to 1,000 times that of a local call (around 10 μs to 60 μs).

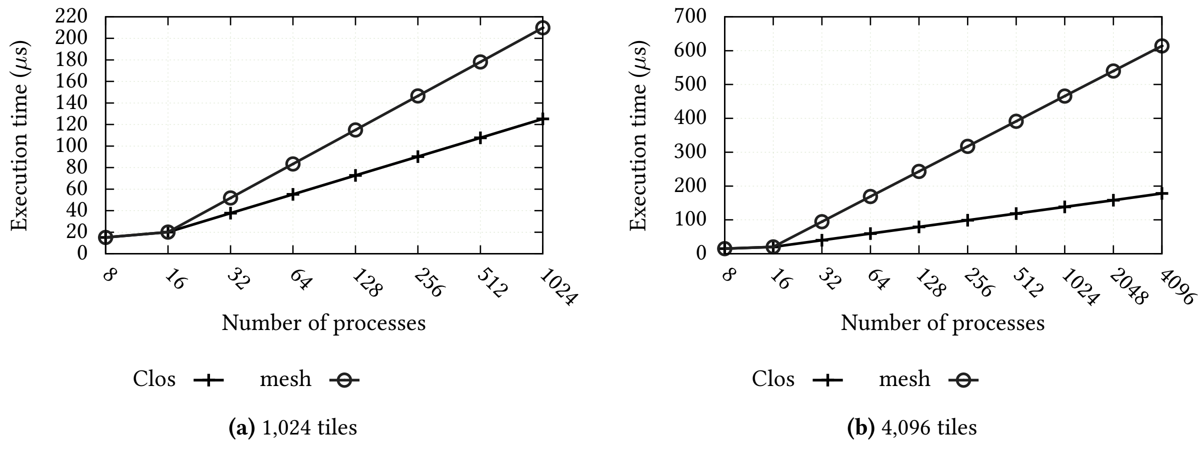


Figure 10.4.: Log-linear plots of the performance of process distribution.

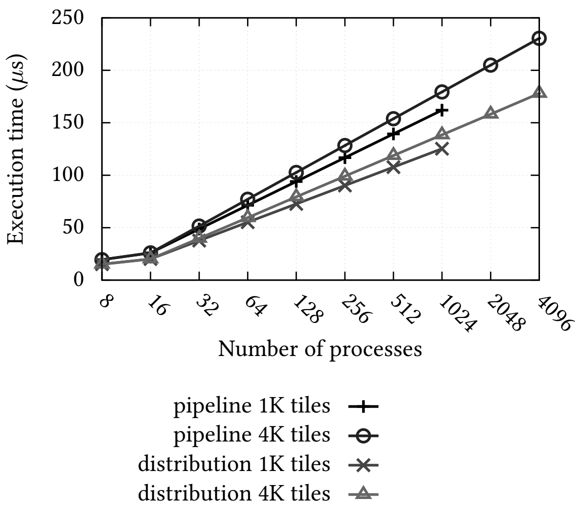
Structural primitives

Figure 10.4 shows how the execution time of a replicated parallel process scales as the number of component processes increases. For 8 and 16 processes, communication does not incur the network latency. Beyond this, the latency increases logarithmically with the size of the replicator because the distribution expands exponentially over the system. Since the execution time presented in the plots corresponds to both the creation and termination of the array of processes, the time to bring the component processes into action will be around half of the total time. This is because the transfers of data between processors are small (few parameters and short programs), so the time will be dominated by the allocation and initialisation (respectively termination and deallocation) of threads, memory and channels etc.

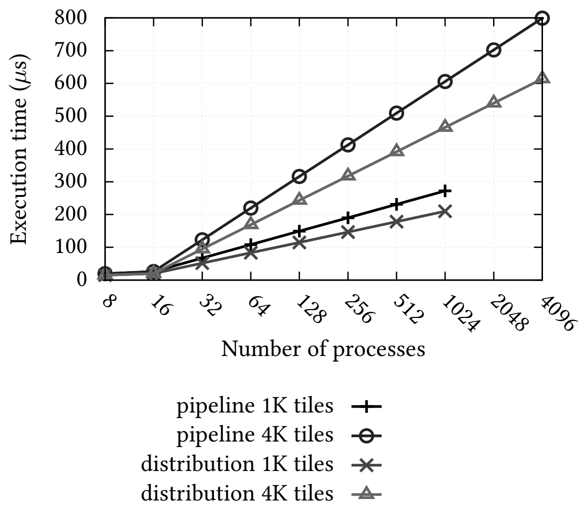
Figure 10.6 shows the performance of the pipeline, 2D grid, hypercube and binary tree process structure benchmarks. In each plot, the execution time for the creation and termination of the structure is given. This comprises the distribution time, which will be similar to that of Figure 10.4 since the overhead of multidimensional replicators and multiple replicators in composition is negligible, and the time to establish the channel connections. The execution time of each of the programs without communication channels is included to indicate the overhead of channel connections.

To help interpret these results, it is useful to look at the time it takes to establish a connection. Between two ready processes, it is 620 ns when they are both connected to the same switch, and when they are not, it takes between around 900 ns and 1,000 ns in Clos networks of 1,024 and 4,096 tiles, and between around 900 ns and 2 μs in 2D mesh networks of the same sizes. These times are small compared to the distribution overhead, but the sequence of connections and the relative timing for each process in the structure introduces further overhead. To see this, consider the parallel recursion in the execution of a replicator; when it branches, one branch will execute locally and the other remotely, causing the two branches to diverge in time. Therefore, processes connecting across branches close to the root will wait the longest; the benchmarks highlight this. In particular: the connection sequences of the pipeline and 2D grid have a fixed number of steps (2 for the pipeline and 4 for the 2D grid) but as the size of the array increases, so does the variation in timing between components, causing longer waits; and the connection sequences for the hypercube and binary tree both have a logarithmic number of steps in N (the number of processes) which for N greater than 8 is more than the pipeline and grid, but they execute faster because the connection they make are better matched with the timing of the distribution.

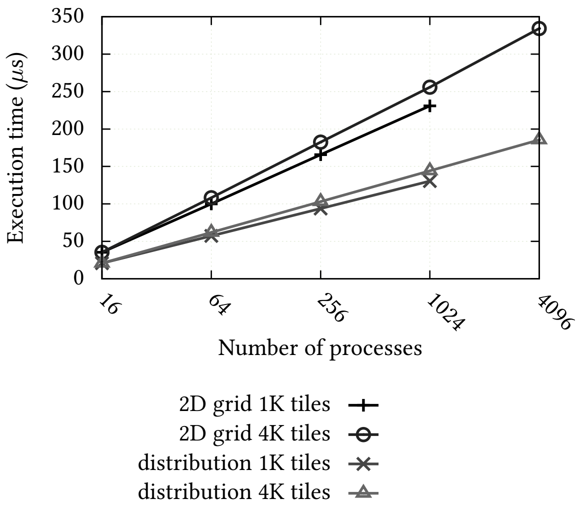
Overall, these results demonstrate a 10% to 25% connection overhead for the pipeline, hypercube and tree process structures, and 40% to 50% for the 2D grid. These structures can be initiated and terminated over 1,024 tiles in 300 μs to 400 μs and over 4,096 tiles in 800 μs to 1,200 μs.



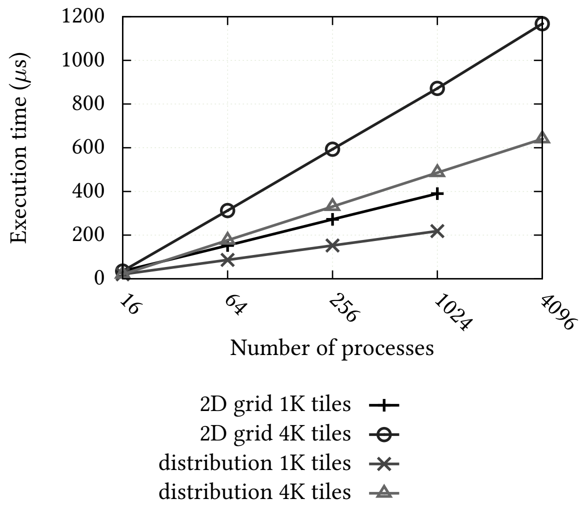
(a) pipeline process structure, Clos network



(b) pipeline process structure, mesh network

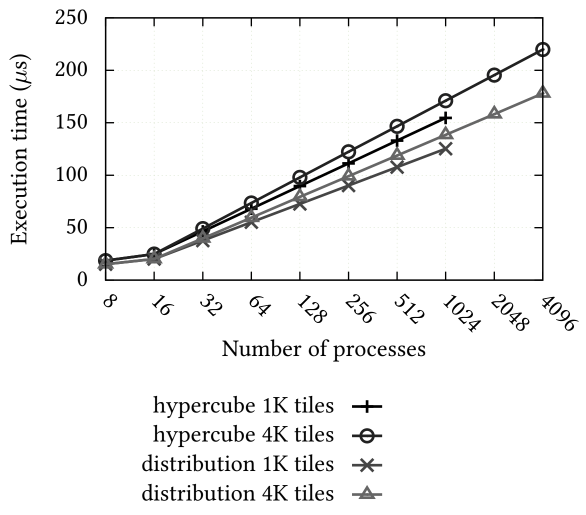


(c) 2D grid process structure, Clos network

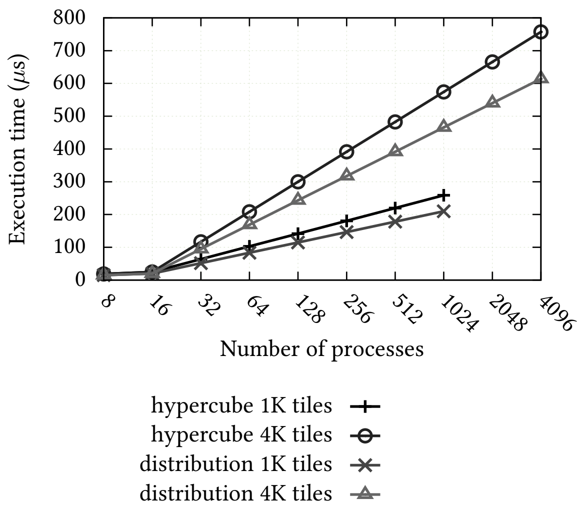


(d) 2D grid process structure, mesh network

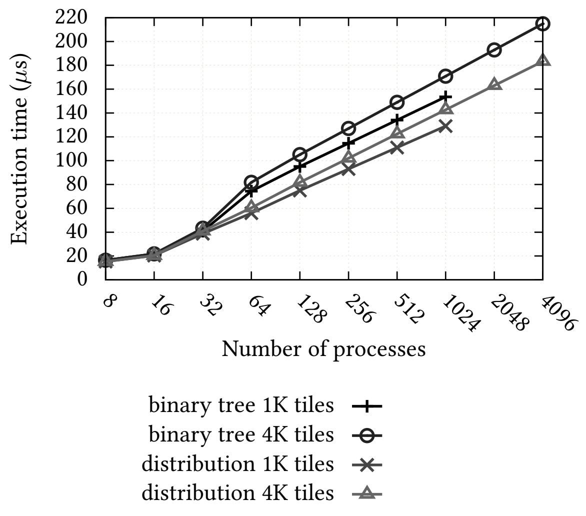
Figure 10.5.: Log-linear plots for performance of the initialisation and termination of pipeline, 2D grid process structures in sire. Each plot also shows the distribution time, which is without any channel connections. There are fewer results in (c) and (d) because they are based on square grids.



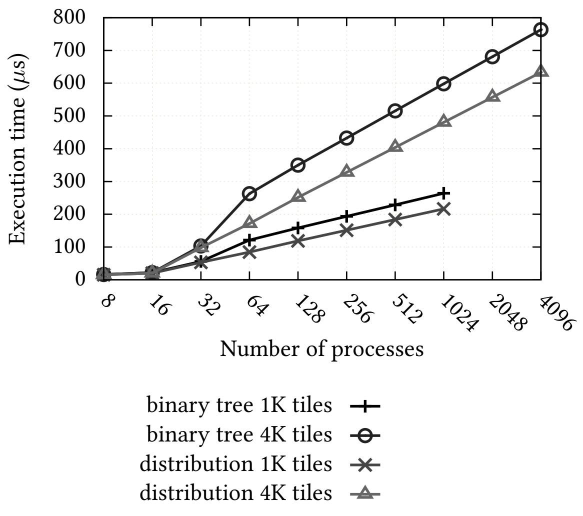
(a) hypercube process structure, Clos network



(b) hypercube process structure, 2D mesh network



(c) binary tree process structure, Clos network



(d) binary tree process structure, 2D mesh network

Figure 10.6.: Log-linear plots for performance of the initialisation and termination of the hypercube and binary tree process structures in sire. Each plot also shows the distribution time, which is without any channel connections.

Summary

Server calls are a mechanism for implementing distributed abstractions. They provide conventional procedure-call semantics but are implemented with sequences of message-passing exchanges. The performance results show that server calls can be executed efficiently, for example, between nodes in a network of 4,096 nodes in less than 1 μs; an overhead of around just 20 times that of a local call. In contrast, remote calls, which are used to distribute state around a system have an overhead of around 150 to 250 times that of a local call. This might sound a lot, but it suggests that it is economical to offload work from a particular processor, either to a nearby processor attached to the same switch when the amount of work exceeds just 2,000 to 12,000 operations, or to any other processor in the system when the work exceeds 8,000 to 32,000 operations.[6]

When remote calls are combined with recursion they can be used to rapidly initiate distributed parallel processes. This forms the basis for the implementation of parallel replicators, which are the principal means of introducing large amounts of parallelism with the sire language. With these, a process array can be distributed over 128 tiles in less than 100 μs, and over 4,096 tiles in less than 200 μs. Regular communication structures can be established dynamically over process arrays with sequences of channel connections. The results demonstrate that for regular structures, the overhead of this is generally small, at around 30% of the distribution time.

Since the evaluation is based on two-phased randomised routing and congested switches, the results represent the performance of the system under load (a particularly favourable case for the mesh) and therefore, these mechanisms can be composed arbitrarily while maintaining the same performance.

10.3. Emulation of large sequential memories

In this section, the ability of the UPA to support sequential programming approaches is explored. Although each processor is capable of executing sequential programs, to be able to execute arbitrary ones, it must also support those with large memory requirements. This can be done by *emulating* a large memory with a collection of smaller ones.

The main component of a conventional sequential machine is a large monolithic memory that provides a large uniform address space. These are typically implemented with a collection of DRAM arrays that are integrated in one or more chips and connected with an interconnect specialised to transmit control, data and address information, to provide efficient sequential random access.[7] The architecture of a DRAM system is therefore inherently *distributed* and in this sense, it is similar to the UPA, which can been seen as a more general system that provides processing capability at each sub-memory as well as an interconnect that can support parallel communication patterns.

This similarity leads to the question, and the subject of this section, *how well can an implementation of the UPA emulate a DRAM system?* The most important factor in this is latency. DRAM systems have been subject to around 40 years of optimisation in their architecture and manufacturing process, reducing access latency to a minimum (the scaling of latency and bandwidth in DRAMs is discussed at the end of this chapter in §10.4.2 [p. 225]). It is therefore determined primarily by the transmission delay on wires, the dimensions of the component array cores and the critical delay of control components such as the memory controller.

[6]Let N be the amount of work to do and D be the cost to distribute it to another processor, then it is economical to divide the work in two when $\frac{N}{2} + D < N$ and therefore when $2D < N$. When D is the cost just to create the parallelism and does not include the cost of any data movement, then this provides a lower bound.

[7]The central component of a DRAM is an *array core*. This is a two-dimensional array of cells and associated peripheral circuitry. The array core is sized to trade-off well between density and delay and energy per activation and refresh. Array cores are limited to a modest size that grows very slowly with respect to technology scaling due to their intrinsic capacitance. For more details on the architecture of DRAM chips see Itoh [Ito01] and for general information on DRAM memory systems see Jacob, Spencer and Wang [JNW07].

An implementation of the UPA will inevitably introduce additional overheads because of its generality, but the hypothesis underpinning this investigation is that, in practice, the overheads are not more than an order of magnitude and that, for conventional sequential programs with a mix of memory and computational operations, it can deliver an efficient emulation with only a small constant factor overhead.

10.3.1. Methodology

Emulation scheme

Typically, the requirements of *local storage*, which includes the program, constant values and the stack, are small and can fit into the local memory of a processor. The remaining *global storage*, which is used for the data pool and heap, for statically- and dynamically-allocated items respectively, can be stored in an emulated memory. The emulation is performed with a collection of tiles that are managed by a memory controller process. The controller receives access requests over a contiguous address range from a sequential *client* program and distributes them over the tiles by sending read and write messages to them. Each tile could respond to these requests by either a server process (similar to the `Store` server in §7.2.4 [p. 121]) or with the remote memory access mechanism. The latter is employed in the evaluation to reduce latency to a minimum.

In the sequential (client) program, accesses to the emulated memory are written as communication sequences, where a read has the correspondence

		`OUTCT` c, `READ4`
`LDW` destination, address	$\rightarrow$	`OUT` c, address
		`IN` c, destination

and a write has the correspondence

		`OUTCT` c, `WRITE4`
`STW` value, address	$\rightarrow$	`OUT` c, address
		`OUT` c, value

In both, c is a channel end that is connected to a memory controller process that will map accesses over the distributed address space.

If the requirements of local storage exceeded the capacity of local memory, there are two options to meet this requirement. Either, the program or stack could be split between processors, and control would transfer from one processor to the other when using either portions of the stack. Alternatively, a second emulated memory could be used. For the purposes of this investigation, only the simple case is considered where local data does not exceed the local memory capacity.

In contrast with parallel programs, execution of sequential programs will not induce any concurrent communication traffic in the network, and unless additional processes are run in parallel, each message will travel without contention. When this is the case, the interconnect needs only to provide low latency and high bandwidth at zero-load, and there is no requirement for two-phase randomised routing. Therefore, messages can be routed obliviously along shortest paths.

Sequential machine model

The performance of a *specialised* sequential machine is used to provide a baseline for the emulation. Performance is presented primarily by the relative *slowdown* of the emulation compared to the sequential machine executing the same program. The modelled performance is based on latency since it is the most difficult aspect to scale; schemes for scaling bandwidth generally involve replication and hence may in principle be applied to both systems. This is discussed in §10.4.2 [p. 225].

The sequential machine is modelled as an instance of the UPA with a single processor that is attached to a DRAM memory. A cache is not modelled but memory accesses to areas that are stored in local memory in the parallel emulation incur the same latency. The effect of this is similar to providing the sequential system with a fast cache memory with an 80% to 90% hit rate since global memory accesses constitute between 10% to 20% of executed instructions in the benchmarks used in the evaluation. The clock frequency of both systems is held at 1 GHz and it is assumed the sequential DRAM can operate at this speed (typical DDR3 operates well in excess of this).

The access latency is estimated for a modern DRAM by simulation with *DRAMSim2* [RCBJ11]. Performance is measured by performing read and write accesses to addresses chosen uniformly at random over the address range and the fixed latency is calculated as the average of these accesses.[8] Latency measurements are based on random accesses and accesses are issued only once the last has completed to restrict the memory controller to processing a single transaction at a time. For a system with 1 Gbit DDR3 chips [Mic12a], average random-access latency is measured at 35 ns for a single *rank*[9] with a 1 GB capacity. For multi-rank system with 2 GB to 16 GB capacities, this increases to 36 ns due to a small overhead in switching between ranks. This choice is not important since the empirical analysis is concerned with obtaining results to within small factors and to demonstrate scaling behaviour.

Typically, DRAMs are packaged in DIMM cards but production processes are moving towards stacked packages integrated with through-silicon vias and it is expected that stacks with multiple DRAM chips will be available in the next few years e.g. with *Wide IO* [JED11] and the *Hybrid Memory Cube* [Hyb13]. Since DRAMSim2 does not model a particular packaging or wire delay, the model is inclusive of stacked DRAM.

Benchmarks

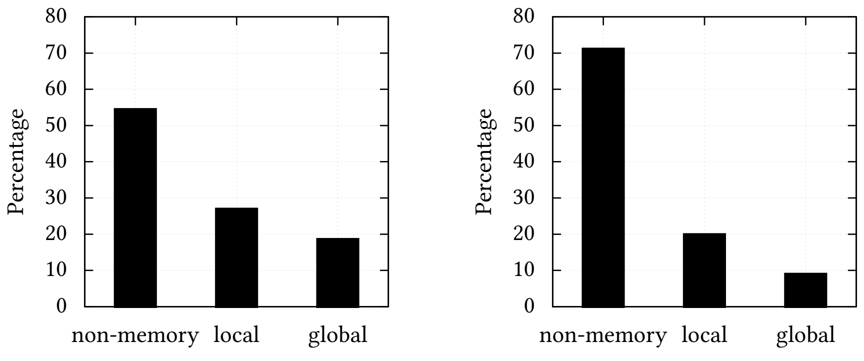
The analysis is based on the following two benchmarks.

1. *Synthetic instructions sequences* that contain a particular ratio of global memory accesses to local memory and non-memory operations, to characterise conventional sequential programs. The ratios of these are chosen based on the instruction mix of the *Dhrystone benchmark* [Wei84], which characterises integer general-purpose sequential programs, and at points over the range of potential ratios demonstrate the effect they have on performance.
2. *A compiler for a simple sequential language.* This provides both an example of a realistic general-purpose application and a means of compiling sequential programs to the UPA. A modified version of it emits message-passing sequences in the place of global memory accesses in order to generate sequential programs that interact with an emulated global memory. Furthermore, this allows the compiler to bootstrap itself, using the emulated memory in which its largest data structures are stored: the parse tree, name table, label table and instruction buffer. The compiler is itself written in the language it compiles and for the experiments, it performs bootstrapping runs by compiling itself.

Figure 10.7 shows the proportions of different types of instructions executed for the Dhrystone and compiler benchmarks, that were measured from a simulated sequential execution. These are *non-memory* instructions, such as arithmetic and branching, *local memory* instructions that include

[8] There is an overhead in opening a row in a DRAM and successive accesses to the same row exhibit lower latency. Consequently, measurement of DRAM latency is based on two main components. First, the *column Address Strobe* (t_{CL}) latency, which is the time between specifying a column address and receiving the data in response, given that the row being accessed is already open. Second, the *row cycle time* (t_{RC}), which is the minimum time between the activation of one row and another; there is a minimum period a row must be active for to perform a refresh and a non-active row must be precharged before it can be read from.

[9] A DRAM *rank* is a set of one or more DRAM chips that are accessed simultaneously and provide a particular data width. For JEDEC-standard (*Joint Electron Device Engineering Council*) DRAMs, a channel typically has a 64-bit data bus and with *error-correcting code* (ECC) memory, this is expanded to 72-bits.



(a) Dhrystone instruction mix **(b)** compiler instruction mix

Figure 10.7.: Instruction mix of the Dhrystone and compiler benchmarks, showing the proportions of executed non-memory, local memory and global memory instructions.

all accesses to the program, stack and constant pool and *global memory* instructions to access the data pool and heap regions. Synthetic sequences are used with a varying proportion of global access and these have a fixed 20% proportion of local memory accesses, based on the benchmarks.

In all experiments, the size of the emulation is scaled by distributing a particular address range over a collection of tiles.

10.3.2. Results

The following sections present the results for the absolute performance of the emulation and its performance with the set of benchmarks.

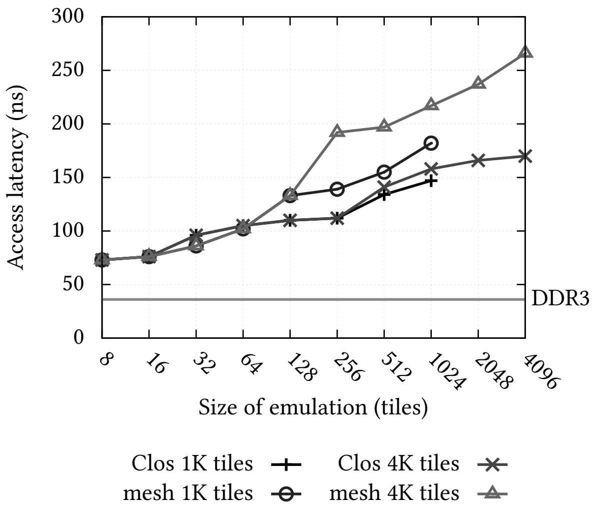
Absolute latency

Figure 10.8 shows how the average access latency of random reads and writes in the emulated memory scales as the number of tiles is increased in the emulation. The baseline latency measured from the simulated DDR3 memory is included for comparison.

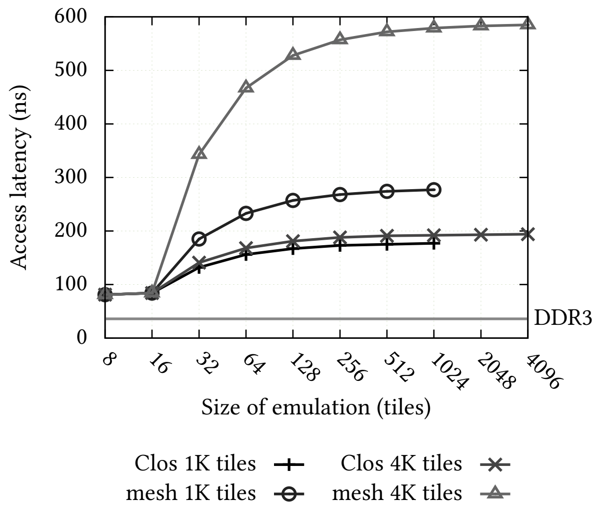
Figure 10.8a shows the performance with oblivious shortest-path routing in an unloaded network, when there is no switch contention. Performance of the Clos network clearly reflects the logarithmic growth of the network diameter and the latency incurred by additional stage in systems larger than 256 tiles can be seen in Figure 10.8a. Latency in the 2D mesh increases linearly with the size of the emulation, with a change of gradient as communications traverse between chips. Overall, the Clos delivers latency with a factor of around 2 to 5 compared to the DDR3 memory. The performance of the two networks is similar on-chip but the 2D mesh incurs a 30% to 40% overhead relative to the Clos for larger multi-chip emulations.

The situation is more pronounced in Figure 10.8b, which shows the performance with two-phase randomised routing in a loaded network. The Clos maintains a similar level of performance, up to a factor of 5.5 compared to the DDR3 memory, but latency in the mesh increases significantly since messages must travel twice the average path length.

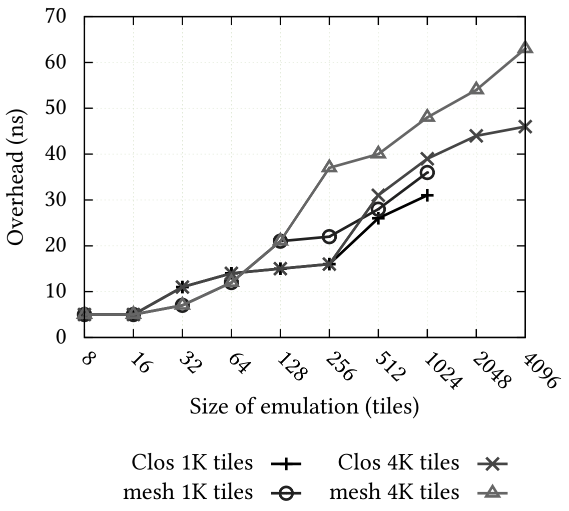
Figure 10.8c and Figure 10.8d show the *additional* latency that is introduced in an identical emulation except now server processes deal with the memory accesses at each of the component tiles instead of the remote memory access mechanism. This is therefore the performance of the emulation when it is expressed purely using the sire notations, and no special optimisations are applied in the compilation. These results show around a 25% overhead and give a further indication of the efficiency of the language.



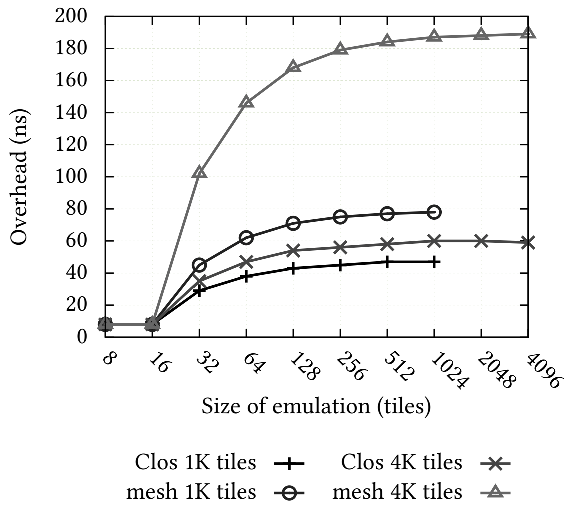
(a) shortest-path routing, unloaded network



(b) two-phase routing, loaded network



(c) shortest-path routing, unloaded network, no RMA



(d) two-phase routing, loaded network, no RMA

Figure 10.8.: Log-linear plots showing how the memory latency scales for 1,024- and 4,096-tile systems with SRAM memory as the number of tiles in the emulation increases. (a) and (b) show the absolute performance with remote memory access to the component tiles; (c) and (d) show the additional software overhead of using server processes at each tile dealing with the memory accesses.

The results for absolute latency are based on the sequential system and the UPA having the same clock speed. An increase in clock speed for the sequential system relative to the UPA would only improve bandwidth because the inherent latency of a DRAM cannot be improved. However an increase in clock speed for the UPA would improve latency because the network would operate faster.

Benchmark performance

Figure 10.9 shows the performance of the synthetic Dhrystone and compiler benchmarks under a range of different system parameters. The general behaviour reflects that of Figure 10.8, but with a mix of emulated global accesses, local accesses and non-memory operations, the overhead of the emulation is lower. The systems with a Clos interconnect can deliver an emulation with a slowdown of between 2 to 3 up to 4,096 tiles over the sequential machine. The performance of the 2D mesh with

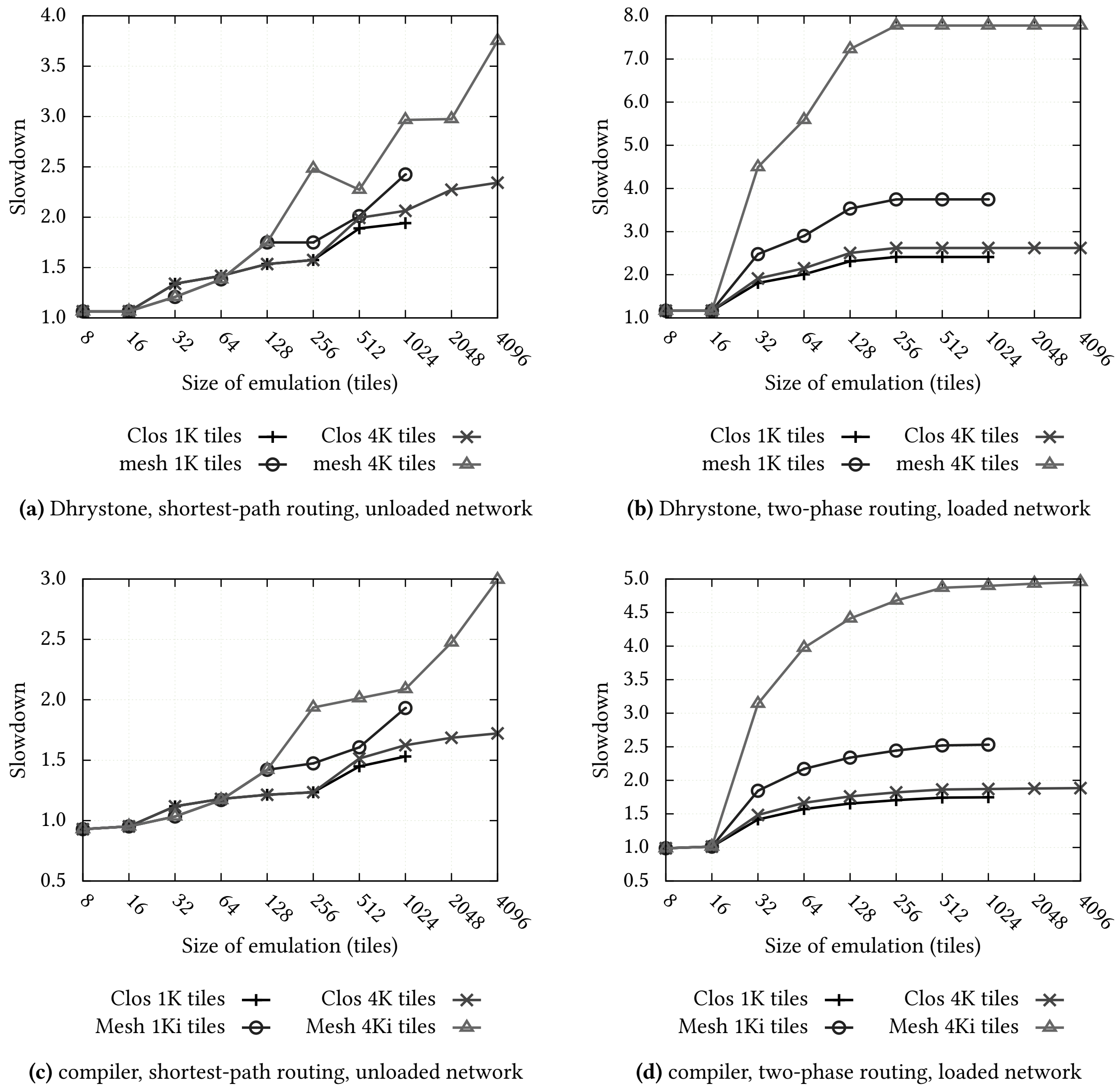


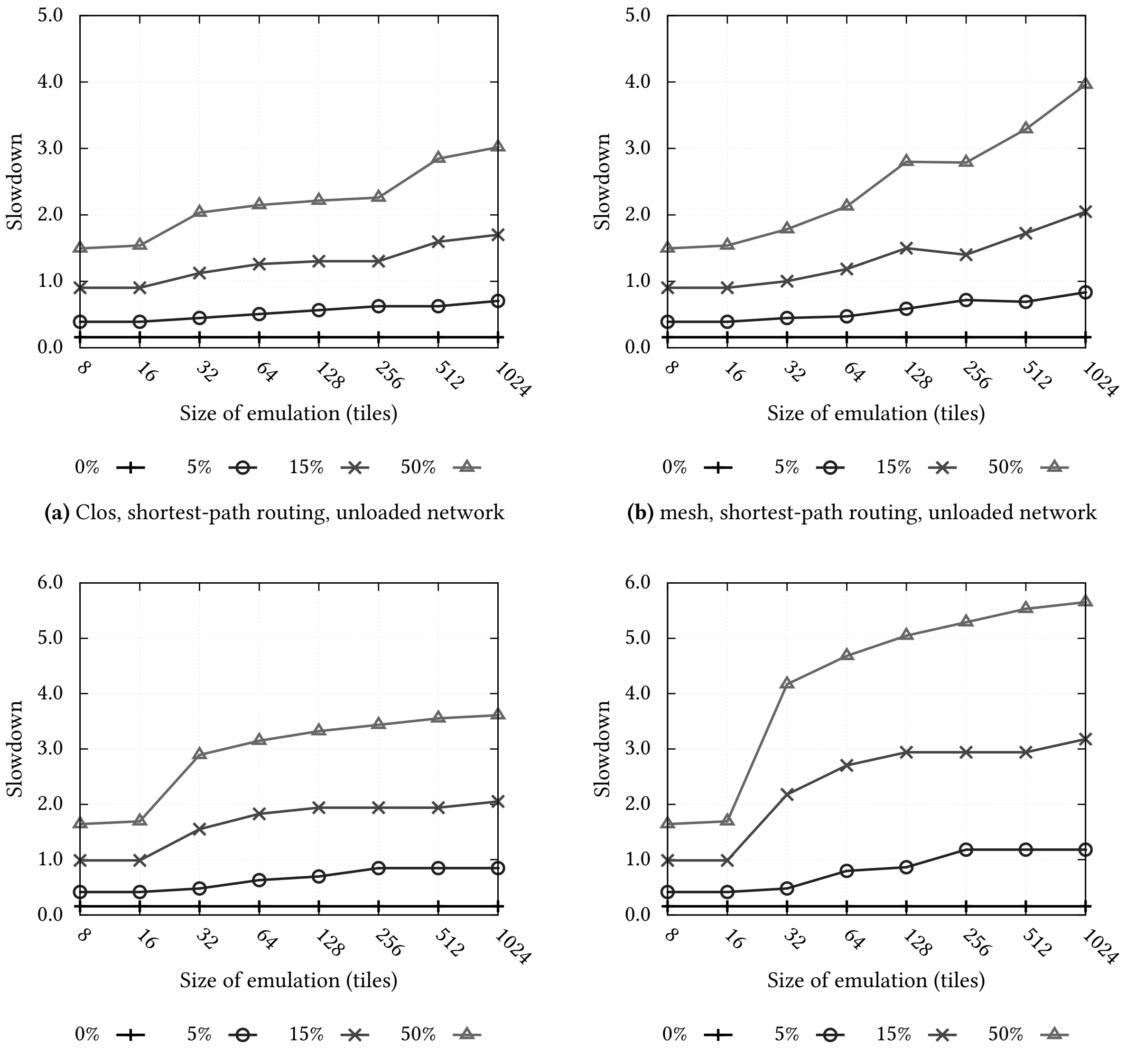
Figure 10.9.: Log-linear plots of the performance of the synthetic Dhrystone and compiler benchmarks, relative to the sequential machine, using an emulated memory on 1,024- and 4,096-tile systems.

shortest-path routing is similar to the Clos up to around 1,024 tiles, but the performance degrades substantially with two-phase randomised routing. In both, the execution of Dhyrstone is less efficient due to the higher proportion of global accesses.

In general, as the ratio of global memory operations to local and non-memory operations decreases, the slowdown does also, converging to a worst case. This is the ratio between the sequential machine with the DDR3 memory and the parallel emulation shown in Figure 10.8. This trend can be seen in Figure 10.10, which shows the emulation performance for different proportions of global memory accesses (between 0% and 50%) in the synthetic benchmark.

Program binary size

Since each memory reference is written as a communication sequence (listed in §10.3.1 [p. 218]), the size of the program binary increases. For loads, there is an overhead of two instructions and for stores, there is an overhead of three. For the version of the compiler that uses the emulated memory,



(a) Clos, shortest-path routing, unloaded network

(b) mesh, shortest-path routing, unloaded network

(c) Clos, two-phase routing, loaded network

(d) mesh, two-phase routing, loaded network

Figure 10.10.: Log-linear plots showing the emulation slowdown, relative to the sequential machine, over a range of instruction mixes, with proportions of global accesses varying between 0% to 50%, for 1,024- and 4,096-tile systems. The proportion of local memory access is fixed at 20%, based on the Dhrystone and compiler instruction mixes

the size of its executable binary increased by 8%. This however is a small price compared to the amount of extra memory that can be provided.

Summary

The absolute access latency of the emulated memory is high, a factor of 3 to 4 times that of a specialised sequential machine for a Clos system, but the effect of this is diluted by fast local accesses and other non-memory operations. For general sequential programs where there is a mix of operations and 10% to 20% global accesses, the Clos systems can deliver to within a factor of 1.5 to 2.5 of the sequential machine in an unloaded network with shortest path routing. When the network is loaded and with two-phase randomised routing, this increases to a factor of 2 to 3.

The performance of the 2D mesh is comparable up to around 1,024 tiles when shortest-path routing is used (a switch diameter of 8), suggesting that it may be practical to use a mesh for systems up to this size with high-degree switches; or equally that this approach could be applied to existing mesh-based systems. However, when two-phase randomised routing is used for general parallel workloads, the performance of the mesh reduces significantly. Furthermore, since the contention model assumes a uniform distribution of communication traffic, which will only occur with certain sympathetic workloads, the real performance will likely be worse.

10.4. Discussion

10.4.1. Evaluation of sire

The evaluation of the primitive mechanisms for parallelism and communication in sire establishes a firm basis for the language by proving their efficiency. In combination with the demonstration in Chapter 7 of the structures that can be expressed and composed with one another, the results suggest that the choice of features in the language and their implementation provide a good trade-off between the simplicity and expressiveness of the language, and the efficiency of its implementation.

However, there were many interesting comparisons drawn, and there are further topics that warrant explanation. The following points outline several ideas.

- *Call request queuing* is a primitive aspect of the implementation of sire but it was not considered in the evaluation. The reason for this is that it was not implemented in the version of the language used for the evaluation due to time constraints. It would however be interesting to evaluate because its behaviour depends on the dynamic behaviour of a collection of clients.
- *Application to 'real' problems.* Ultimately, the main reason to employ parallelism is to scale computational performance. Extending the evaluation to demonstrate the performance of the sire language and UPA when applied to real problems *and* demonstrating good parallel speedups, would be convincing evidence supporting the capability of the approach.

  An evaluation of this nature would however depend either on a real implementation of the UPA or a scalable simulation. It was possible to scale the simulations in this chapter up to 4,096 tiles because the benchmark programs have short execution times (hundreds of thousands of cycles), but at that size the simulation time for some programs was in excess of 10 minutes. For 'real' problems, the working datasets will necessarily be large and the execution time will span much longer. One approach to constructing scalable simulations is to employ parallelism with *distributed discrete-event simulation* [Mis86].
- *Network simulation.* The performance model used for the evaluation provides an approximation to shortest-path routing with zero-communication traffic and two-phase randomised routing with uniform congestion. It could both be improved and validated by directly modelling the network. This would involve modelling the behaviour of the switching component and the network interface to the processor.

- *Latency hiding*. To obtain a high utilisation of a system it will be essential to hide communication latency by allocating multiple processes to each processor. This was discussed at the end of the last chapter in §8.6.2 [p. 175] but a specific scheme was not described in the compilation chapter because, based on the existing implementation and evaluation, the best way to do this was not clear.

  On the basis of realistic application benchmarks, an evaluation of the effects of latency hiding would be essential to establish a good approach to it as well as demonstrating the full capability of the UPA in executing sire programs.

10.4.2. Emulating large memories with the UPA

Bandwidth scaling

The evaluation uses latency as the primary performance metric and simplified models of the parallel and sequential systems capture the essential aspects of it. Latency is chosen because it is subject to fundamental physical limits and is inherently difficult to scale, whereas bandwidth can, in general, be scaled by replicating components at the cost of area, or increasing transmission rates at the cost of power.

This is the case with modern DRAMs where bandwidth has been scaled aggressively, driven by higher transistor densities, whilst latency has scaled very slowly [Pat04]. Consequently, many of the architectural optimisations employed by modern devices are designed to scale bandwidth involve replication to exploit parallelism in accesses; the following are the main examples.

- *Double data rate* (DDR) DRAMs allow two data items to be transferred every cycle, one on each edge of the clock. This is achieved by replicating a memory and parallelising access to it.[10]
- *Bursting*. There is a cost associated with accessing a row in a DRAM and successive reads from an open page will exhibit lower latency. Memory controllers can issue commands based on address, scheduling and queuing policies and dynamically reorder outstanding transactions to best exploit this.
- *Multiple channels*. Memory systems can be configured with multiple channels where entire ranks of DRAM are replicated and accessed in parallel, providing a data rate increased up to a factor of the number of channels.

Figure 10.11a shows the average latency in nanoseconds of random reads and writes in successive generations of DRAM devices, from *single data rate* (SDR) with a 100 MHz interface to DDR3 (that was used as a baseline in the evaluation) with a 1.8 GHz interface [Mic12d, Mic12b, Mic12c, Mic12a]. Over these generations, bandwidth has increased by around a factor of 40 while latency has decreased by around only a factor of 2. This is due mainly to the relatively constant scaling and delay of the array cores and interconnections. When the access time is measured in cycles (Figure 10.11b) it increases with respect to operating frequency. This has resulted in modern DRAMs being reliant on complex memory controllers to exploit reduced latency to accesses in the same row.[11]

Bandwidth was not considered in the evaluation since architectural optimisations can in principle, be applied to the parallel system in either the UPA implementation, the emulation, or both. For example:

[10]DDR DRAM replicates the core array twice so a transaction can be sent to both simultaneously and data is returned from one of them on the positive edge of the clock and the other on the negative. DDR2 and DDR3 memory employs the same idea with four arrays, returning values on each edge two/four clock cycles.

[11] A modern DDR3 memory controller maintains a buffer of outstanding transactions for look-ahead to prepare DRAM for an access, and reordering to mask the latency of accesses to different memory pages [Gre11]. This requires the stream of transactions to contain correlated bank accesses in the same way super-scalar processors require independence between sequences of instructions to exploit ILP. With the higher clock rates of DDR4, the situation will continue to worsen and the benefits of optimisations implemented in the memory controller will diminish, in the same way they have done in sequential processors; this was discussed in §9.6.1 [p. 200].

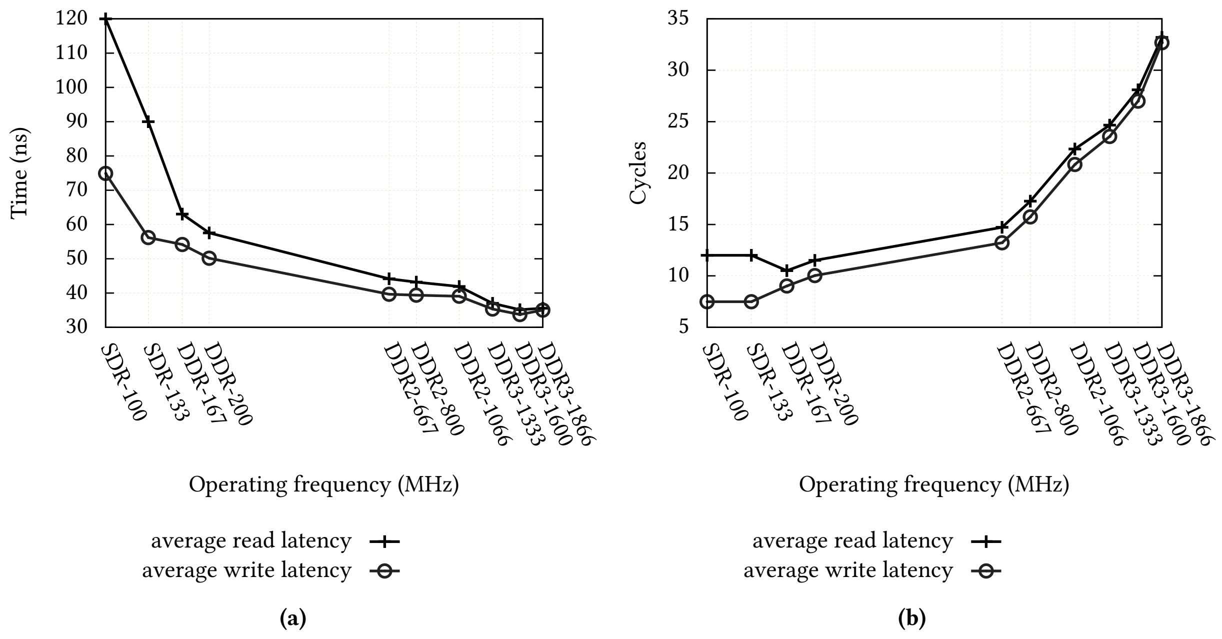


Figure 10.11.: Log-linear plots of the average random read and write latency for monolithic DRAM systems as a function of operating frequency. The data points are specific devices that span the generations of DRAM technology [Mic12d, Mic12b, Mic12c, Mic12a] and the latency was measured with DRAMSim2. The devices are labelled on the horizontal axis with the notation ⟨technology⟩-⟨frequency⟩. Plot (a) shows the latency in nanoseconds and plot (b) the latency in cycles.

- the interconnect could employ DDR transfers or the frequency could be increased to achieve the same effect;
- additional functionality could be added to the memory controller process to implement bursting and transaction reordering;
- replication could be employed to further utilise the additional capacity in the interconnect.

This choice depends on the desired capabilities of the system and the balance between sequential and parallel workloads.

Reducing latency in memory emulations

The UPA is conceptually similar to a modern DRAM system, both having a collection of interconnected memories. The main additions are processors, a more complex switch between link connections and a greater provisioning of interconnect resources to support general concurrent communications.

Access to a DRAM system consists of request and reply components sent over the interconnect. Messages experience delay through components such as the processor and memory controllers, on connections between these from crossing between different clock domains and in transmission on long busses, and from the DRAM itself. The interface to the memory is integrated into the processor instruction set and a program simply executes a load or store instruction to perform an access. DRAM memory systems have been heavily optimised for this specific case. An access to an emulated memory also consists of request and reply components and the latency experienced by these will be dependent on software and hardware delays through the processor, time domain crossing and transmission on links. Additional latency is contributed by serialisation of data on to narrow links, the link between the memory and the interconnect and the switches along the path through the network.

Access latency could be further reduced in the UPA emulation, to approach that of conventional DRAMs. The following improvements could provide significant benefits.

- *Switch latency.* The latency model was calibrated against measurements made with the XMP-64, but the implementation of the switch in the XS1 G4 chips was synthesised automatically, which potentially adds a factor of 2 to 3 times the latency in the critical path over a manual layout. This could reduce the switch latency to 1 cycle and overhead to open a route to 2 cycles; other designs can achieve this kind of performance, for an example see Kumary et al. [KKS+07] or Mullins, West and Moore [MWM04].
- *Hardware generation of memory references.* The emulation scheme described in this chapter employs a software memory controller. This could be implemented alternatively in hardware as part of the ISA, in order that load and store instructions accessing particular address ranges are converted into messages and sent on the interconnect in a single cycle.

Performance trade-offs

A balance must be struck between the sequential and parallel capabilities of an *implementation* of the UPA. The extent to which the system is geared towards sequential programming by using powerful sequential processors or an interconnect that is specialised to deal with memory accesses depends on the workloads for which it is intended.

One potential scenario is that early generations of the architecture could be specialised in these ways in order to facilitate a transition away from conventional shared-memory systems. Then, as workloads change and become more parallel, the capability of later generations would change accordingly. Because the specialisations are applied to the implementation (in a homogeneous *or* heterogeneous way), the same architecture is maintained and so it does not impact the programming model. This is an essential feature of a general-purpose design.

Extensions

There is great potential for ways in which the UPA scheme for sequential execution could be extended to further exploit the underlying architecture. These possibilities stem from the significant capability of the interconnect and programmability of the system.

The following points outline several ideas. For the direct compilation of sequential programs on the architecture, most of these could be integrated into the compilation process transparently to the programmer; the resulting performance benefits could surpass the performance of conventional systems.

- *On-the-fly processing.* With a processor associated with and mediating access to each memory in the system, data read and written in remote memories could be processed on-the-fly. In particular, compression could be applied before sending messages to increase throughput and reduce latency.
- *In-place processing.* Overheads of data movement could be significantly reduced by moving processes to the processor storing particular data to operate on it in-place in local memory, rather than transferring it back and forth as is the case with conventional memory systems.
- *Concurrent access.* Unlike a conventional DRAM, an emulated memory can easily be extended to support concurrent access (see the `ParallelRAM` server in §7.2.4 [p. 123]). A sequence of reads from distinct locations, which might occur as terms in an expression, could be issued simultaneously. The effective cost of this sequence would then be similar to that of a read.
- *Debugging and profiling.* Since all memory requests are issued as messages sent on the interconnect, they could be intercepted and inspected to provide debugging or profiling information.

- *Caching.* The memory emulation could be extended to implement a caching scheme, similar to the `CachedRAM` server in §7.2.4 [p. 121]. Furthermore, the caching scheme employed could be optimised by the programmer, or even in the compilation process, to match the characteristics of the program that it serves.
- *Low-power states.* With single large memories it is likely that portions of the address space will be inactive for long periods. During these periods, the memories and processors of individual tiles could either be switched off or placed in a low-power state to reduce overall power consumption.
- *Reducing system granularity.* Multiple instances of the memory emulation scheme could be run in parallel to emulate a parallel machine with a larger memory capacity per tile, effectively reducing the granularity of the system.

Chapter 11.

Summary and conclusions

11.1. Background

The use of *general-purpose programmable machines* called *computers*, since their inception in the early 20th century, has become almost ubiquitous. Their success has stemmed from the ability to be *programmed* according to a simple computational model and to be applied to any problem with a reasonable degree of efficiency. This, coupled with large markets and sympathetic implementation technologies, has provided economies of scale and continued substantial improvements in performance and reduction in size.

The conventional universal *sequential* model of computation, efficiently embodied by the *von Neumann architecture*, has underpinned these advances but it has exhausted the capabilities of known technologies to scale performance. Now, parallelism is the only known means to sustain performance but the prevailing reaction has been evolutionary by extending the von Neumann machines with additional processors. This approach maintains the conventional abstraction of a large randomly accessible memory (and even compatibility with legacy programs) but it does not scale.

There are known techniques for building scalable parallel machines that have a single *shared* address space, but even so, shared memory has proven to be ineffective in dealing with large amounts of parallelism. This is evident from the fact that the only programming approach that has and can endure, is MPI. The reason for this is clear; shared memory is fundamentally at odds with parallelism because it hides communication, with the result that communication is both difficult to express and inefficient. These problems restrict shared memory to limited forms of parallelism and complicates the process of compilation for (scalable) distributed architectures, and these problems are exacerbated by the programming issues of non-determinism and formal verification.

In contrast to shared memory, message passing is simple, general, efficient and can be formalised; but surprisingly, it has not received a great deal of attention. The reason for this can be attributed largely to preoccupation with sequential microprocessor performance and that also, in the relatively small HPC community, MPI has sufficed. However, as implementation technologies continue to improve, computers can be increasingly *embedded* to facilitate the advancement of exciting new areas such as medical equipment, clothing, the home environment and robotics. In these applications, computers will be applied to the problems of sensing, interaction and decision making, and so will need to exploit a variety of forms of parallelism.

Embedded computing applications will continue to demand the development of a wide variety of devices with complex capabilities. Because much of the software has yet to be written, there is a substantial opportunity for a new *standard form* of parallel computing to provide this, analogous to that of general-purpose sequential computing with the von Neumann architecture and languages like C. The utility of a such a model is clear: it enables high-volume production and optimised manufacturing processes and supports standardisation across programming languages, tools and machine implementations, providing portability of programs between current machines and future generations as they develop with future technology.

11.2. Contributions

This thesis has demonstrated the essential aspects of a standard model of parallel computation by proposing a scalable universal parallel architecture, the UPA, a high-level programming language designed for it called sire and a compilation scheme to transform sire programs to execute efficiently on the UPA.

The crucial advantage of this model over others (in particular the PRAM and BSP [Val90a]) is that it allows a wide class of parallel programs to be expressed. It does this by using message passing as a basis and *combining* mechanisms for *sharing* with it to support both programming structures with fixed communication patterns such as process networks, and structures that do not, such as shared data structures.

11.2.1. The UPA design

The UPA was described in Chapter 5. It is a distributed-memory parallel computer architecture with a high-performance universal interconnection network that connects an array of processor-memory tiles that use the *XMOS XS1 processor core* [May09]. The UPA design draws on Valiant's theory of *universal parallel communication networks* [Val90b] and known approaches to building parallel computers, particularly on the design of the C104 switch [MTW93, Ch. 3].

The specific contributions of the UPA design are the combination of network topology, switching and processor architecture that produce a *balanced* general-purpose system. The result is an architecture that can support a range of programming techniques and the execution of highly-parallel programs as well as sequential ones with large memory requirements.

The folded Clos network (closely related to a *fat tree*) provides the necessary properties for universality, it supports universal two-phase randomised routing efficiently and its structure offers a significant degree of flexibility in an implementation by allowing the use of arbitrary fixed-degree switches, the bisection bandwidth to be adjusted, and a hierarchical packaging using different technologies. The network implementation supports low-latency communication of arbitrary-sized messages and memory accesses using wormhole routing. XS1 processor cores further minimise communication latency with an optimised network interface and they provide low-level support for parallelism with efficient mechanisms for creating, synchronising and terminating processes.

The UPA is simple to implement and it allows particular implementations to be easily *specialised* whilst maintaining the same programming model by adjusting the network bandwidth to meet the limitations of an implementation or by optimising the system for particular workloads by changing the balance of processing to memory.

11.2.2. The sire programming language design

The sire programming language was defined in Chapter 6, complete syntax given in Appendix A and illustrative examples of its use were given in Chapter 7. Sire builds on the formalism of CSP [Hoa78] and the approach of occam [May83], and is designed to be the natural language for programming the UPA, with which it is easy to express programs that deal with large amounts of distributed parallelism.

Sire builds on occam principally by integrating a mechanism for *sharing* on a message-passing framework by introducing the concept of a *server* as a primitive in the language. This is significant because it combines the essential capabilities of (conventional) shared-memory programming with the benefits of message passing. With mechanisms for combining collections of servers, sire provides powerful capabilities for abstraction with distributed parallel programs that are fundamentally important to a discipline of programming:

- to separate distributed representations of data from the computational components of a program, to build *scalable data structures*;

- for distributed parallelism to be employed freely and embedded in sequence, giving rise to a mechanism for *parallel subroutines*;
- for the arbitrary composition of program components to create *modular hierarchical program structures*.

Servers provide some additional abilities and benefits for the programmer:

- as way of dealing explicitly with data locality and movement of program code to the data on which it operates;
- with declaration syntax similar to standard variables, a program can be composed in a conventional way as a sequence of declarations followed by a sequence of operations or subroutines;
- with call syntax similar to local procedure calls, programs can be easily refactored to employ servers to introduce parallelism or data distribution.

In terms of the language design, servers provide an elegant way to deal with all aspects of abstraction involving communication. This removes the need for named communication channels that occam and the later version of CSP have, instead a process-naming mechanism can be used, which significantly simplifies the distributed implementation.

The overall design of sire allows it to be compiled using simple techniques and therefore for its primitive operations to correspond closely to the operation of the UPA. Apart from the immediate benefits of efficiency and predictability for the programmer, this additionally places the language in a position where it can be used to implement low-level distributed 'system' functionality. For example, it can be used to build components usually realised in hardware such as memories with a particular consistency model, caches to improve locality and facilities for hashing, replication and combining to manage distributed data; or even used as a basis to implement other programming languages.

11.2.3. Compilation of sire programs to the UPA

The sire compilation scheme was described in Chapter 8. The compilation is divided into two parts; the first is a set of program transformations that *reduce* a sire program into a simplified *canonical form*. This approach is beneficial because the output is understandable by the programmer and only the small canonical subset needs to be implemented. The second is the generation of machine code and description of the run-time kernel. This was presented with detailed instruction listings for each (unconventional) feature of sire to demonstrate that it is capable of being compiled into short instruction sequences and that there is no hidden complexity.

In addition to the direct mapping of sire notations to machine operations, its design facilitates two enhancements to compilation and program execution. First, sire processes can be mapped to processors according to a static schedule that is determined during compilation, so that processors do not need to be allocated and deallocated at run time and the overhead of using distributed parallelism is minimal. Second, the distribution of program code can be performed at run time, which creates a number of benefits. For the compilation scheme, this decouples the compilation process (and its execution time) from the number of processors in the target system and allows a minimal program binary to be produced. For the execution of a program, it allows a system to be booted rapidly by replicating a single binary and during execution it makes a run-time reuse of processor memory, which is essential when memory is limited.

A crucial issue with the compilation of sire programs is the prevention of deadlock from many-to-one server channels, caused by client requests preventing a server from engaging in communication. This is dealt with by servers queuing client requests locally and causing the client to wait, and only completing a call when the server is ready.

11.2.4. Performance evaluation of the UPA and sire

Chapter 9 presented a high-level model for an implementation of the UPA, which integrates a complete sub-folded Clos network onto a single chip and uses a silicon interposer to build larger systems with multiple chips. The model demonstrates that despite the apparently high costs of universal communication networks, even with current production technology, there is only a moderate overhead. In a system that connects between up to 4,096 processors, the overall investment is around 20% to 30% of total cost, compared to around 5% with the popular 2D mesh topology (which is not universal and therefore unsuitable for general-purpose systems).

Chapter 10 used the implementation model to inform a simple performance model for the UPA. The performance model was integrated into a software simulation of the UPA and used to evaluate the performance of specific programs. Results for a set of *microbenchmarks* that exercise the primitive mechanisms for parallelism, communication and abstraction in sire confirm they are all very low cost; remote server calls, performed over the network have an overhead of 8 to 20 times that of local calls, processes can be offloaded to remote processors if they exceed several tens of thousands of basic operations (around 10 μs to 60 μs at 1 GHz) and the mechanism for process distribution, which underpins the basic subroutine mechanism, can bring thousands of processors into action in a period of several hundred thousand operations (4,096 tiles in around 200 μs at 1 GHz). Furthermore, these results are *compositional* in that they are independent of the parallel workload being executed because the network delivers bounded latency for all communication patterns.

The final part of the empirical investigation looked at the ability of the UPA to support sequential programming techniques by emulating large memories with collections of tiles. This demonstrated that general sequential programs can be executed with a reasonable degree of efficiency (a factor of 2 to 3 overhead) when compared to contemporary sequential machines. The ability to execute sequential programs efficiently is significant because, despite parallelism being the primary means of scaling performance, many problems exhibit little or no parallelism and many existing formulations are sequential. It could therefore facilitate a transition from sequential machines by allowing existing programs to be compiled directly to the UPA and then for their performance to be improved either through new compiler optimisations or by changing the program to use parallelism.

11.3. Conclusions and future work

A preoccupation with sequential performance has meant that there has been relatively little development of general-purpose models of parallel computation, and indeed, of scalable parallel architectures outside of HPC. Consequently, there are remarkably few scalable (but non-cluster-based) parallel architectures that are intended for general-purpose use. This is reflected in the variety of existing parallel programming models that in general are heavily biased towards shared memory; of these, only a few message-passing approaches have proven to be simple to use and to scale. There is both a huge potential for innovation in parallel architectures and programming models that can support large amounts of parallelism and a significant opportunity for them to advance unexplored areas of embedded computing.

The work in this thesis has investigated the potential for scalable general-purpose parallel computing by developing practical proposals for an architecture, a programming language and a compilation scheme. In doing so, it has had to work from the gritty details of a hypothetical implementation, all the way to the high-level issues of program structuring. To manage this, the approach has been to use Ockham's razor wherever possible to produce a minimal design, and the approach of the empirical work was to consider as far as possible the high-level trade-offs and scaling issues. This has both left, and led to, a host of opportunities for extension and further questions; the discussions at the end of each chapter have provided starting points for future work.

Bibliography

[ABC+06] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[ACH+05] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. W. Maessen, S. Ryu, G. L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, et al. The Fortress language specification. *Sun Microsystems*, 139:140, 2005.

[ACJ+91] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, and D. Nussbaum. The MIT Alewife machine: A large-scale distributed memory multiprocessor. Technical Report 454, MIT/LCS, 1991. Also in Scalable Shared Memory Multiprocessors, Kluwer Academic Publishers.

[Ada11] Adapteva Inc. 1024-core 70GFLOP/W floating-point manycore microprocessor, October 2011. Whitepaper.

[AG96] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[Agh85] G. A. Agha. Actors: a model of concurrent computation in distributed systems. Technical Report AITR-844, MIT, June 1985.

[AGJP11] Darren Anand, Kevin Gorman, Mark Jacunski, and Adrian Paparelli. Embedded DRAM in 45-nm technology and beyond. *IEEE Design & Test of Computers*, 28:14–21, January 2011.

[AHMP87] H. Alt, T. Hagerup, K. Mehlhorn, and F. P. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal on Computing*, 16(5):808–835, 1987.

[AJS05] Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders. *Communicating Sequential Processes. The First 25 Years: Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004. Revised Invited Papers*, volume 3525. Springer, 2005.

[Ame83] American National Standards Institute. *Reference Manual for the ADA Programming Language.* Silicon Press, 1983.

[ARK10] R. Alverson, D. Roweth, and L. Kaplan. The Gemini system interconnect. In *IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, pages 83–87. IEEE, 2010.

[Arl88] R. Arlauskas. iPSC/2 system: a second generation hypercube. In *Proceedings of the third conference on Hypercube concurrent computers and applications: Architecture, software, computer systems, and general issues-Volume 1*, pages 38–42. ACM, 1988.

[ARM12a] ARM Ltd. Cortex-A9 processor, 2012. Technical specification.

[ARM12b] ARM Ltd. Cortex-M0 processor, 2012. Technical specification.

[AS07] B. Agrawal and T. Sherwood. Guiding architectural SRAM models. In *International Conference on Computer Design*, pages 376–382. IEEE, 2007.

[Bac78] John Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bak90] H. Buman Bakoglu. *Circuits, Interconnections, and Packaging for VLSI.* Addison-Wesley, 1990.

[Bar92] Geoff Barrett. *occam 3 reference manual.* INMOS Ltd., Bristol, UK, March 1992.

[BBG$^+$93] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang. Distributed pC++ basic ideas for an object parallel language. *Scientific Programming*, 2(3):7–22, 1993.

[BCC$^+$88] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: an integrated solution to high-speed parallel computing. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 330–339, November 1988.

[BCM94] E. Barton, J. Cownie, and M. McLaren. Message passing on the Meiko CS-2. *Parallel Computing*, 20(4):497–507, 1994.

[BD06] J. Balfour and W. J. Dally. Design tradeoffs for tiled CMP on-chip networks. In *Proceedings of the ACM/IEEE conference on Supercomputing*, ICS '06, pages 187–198. ACM, 2006.

[BDG$^+$91] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A users' guide to PVM (parallel virtual machine). Technical report, Oak Ridge National Laboratory, TN, 1991.

[BDH$^+$08] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: The architecture and performance of Roadrunner. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2008.

[BDMF10] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. *ACM SIGARCH Computer Architecture News*, 38(3):302–313, 2010.

[Ben65] V. E. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic.* Mathematics in Science and Engineering. Academic Press, 1965.

[BH85] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30(1):130–145, 1985.

[BHMS91] Mark Bromley, Steven Heller, Tim McNerney, and Guy L. Steele, Jr. Fortran at ten gigaflops: the Connection Machine convolution compiler. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI '91, pages 145–156. ACM, 1991.

[BJK$^+$95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8):207–216, 1995.

[BJW07] M. Butts, A. M. Jones, and P. Wasson. A structural object programming model, architecture, chip and tools for reconfigurable computing. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 55–64, April 2007.

[BK94] Ray Barriuso and Allan Knies. SHMEM user's guide for C. Technical report, Cray Research Inc., 1994.

[Ble90] G. E. Blelloch. Prefix sums and their applications. In John H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan-Kaufmann, 1990.

[Ble95] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-95-170, Carnegie Mellon University, September 1995. Version 3.1.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[BO84] Marsha J Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484 – 512, 1984.

[Bok90] Shahid H. Bokhari. Communication overhead on the Intel iPSC-860 hypercube. Technical report, Institute for Computer Applications in Science and Engineering, NASA, 1990. ICASE interim report 10.

[Bon02] Dan Bonachea. GASNet specification. Technical Report CSD-02-1207, UC Berkeley, October 2002. Version 1.1.

[Bor07] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749. ACM, 2007.

[BPN$^+$10] J. Barth, D. Plass, E. Nelson, C. Hwang, G. Fredeman, M. Sperling, A. Mathews, W. Reohr, K. Nair, and N. Cao. A 45nm SOI embedded DRAM macro for POWER7 32MB on-chip L3 cache. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 342–343. IEEE, 2010.

[BS81] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 187–194. ACM, 1981.

[BSC$^+$93] G. E. Blelloch, Chatterjee S., Hardwick J. C., Sipelstein J., and Zagha M. Implementation of a portable nested data-parallel language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, May 1993.

[Can69] Lynn Elliot Cannon. *A cellular computer to implement the Kalman Filter algorithm*. PhD thesis, Montana State University Bozeman Engineering Research Laboratories, 1969.

[CCD$^+$11] B. L. Chamberlain, S. E. Choi, S. J. Deitz, D. Iten, and V. Litvinov. Authoring user-defined domain maps in Chapel. In *Cray Users Group Conference (CUG)*, 2011.

[CCDN11] B. L. Chamberlain, S. E. Choi, S. J. Deitz, and A. Navarro. User-defined parallel zippered iterators in Chapel. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models*, 2011.

[CCZ07] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[CD90] A. A. Chien and W. J. Dally. Concurrent aggregates (CA). In *ACM Sigplan Notices*, volume 25 of *3*, pages 187–196. ACM, 1990.

[CDC$^+$99] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[CEH+11] Dong Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 1–10, November 2011.

[CG95] P. Caspi and A. Girault. Execution of distributed reactive systems. In *Proceedings of the 1st International Euro-Par conference on Parallel Processing*, pages 13–26. Springer, 1995.

[CGL86] N. Carriero, D. Gelernter, and J. Leichter. Distributed data structures in Linda. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 236–242. ACM, 1986.

[CGS+05] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Notices*, volume 40, pages 519–538. ACM, 2005.

[CJ87] N. J. Carriero Jr. Implementation of tuple space machines. Technical Report Research Report YALEU/DSC/RR-567, Yale University, December 1987.

[CK93] K. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. *Languages and Compilers for Parallel Computing*, 1:124–144, 1993.

[CLC+98] B. L. Chamberlain, C. Lin, S. E. Choi, L. Snyder, EC Lewis, and W. D. Weathersby. ZPL's WYSIWYG performance model. In *Proceedings of the third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 50–61. IEEE, 1998.

[Cle06] Clearspeed Ltd. Clearspeed CSX processor architecture. Technical Report PN-1110-0306, 2006.

[Clo53] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, March 1953.

[CMD+00] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan Kaufmann, 2000.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[Col89] M. I. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press & Pitman, 1989.

[CT89] K. M. Chandy and S. Taylor. The composition of concurrent programs. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 557–561. ACM, 1989.

[CW79] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[DA93] W. J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):466–475, 1993.

[Dal90] W. J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, 1990.

[Dal92] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194 –205, March 1992.

[Dav00] C. David. An experiment with recursion in occam. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures WoTUG-23: Proceedings of the 23rd World occam and Transputer User Group Technical Meeting*, volume 30, page 193. IOS Press, 2000.

[dBE46] N. G. de Bruijn and P. Erdos. A combinatorial problem. *Koninklijke Netherlands: Academe Van Wetenschappen*, 49:758–764, 1946.

[DCC$^+$98] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a message-driven processor. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 337–344. ACM, 1998.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured programming*. Academic Press Ltd., 1972.

[DeH00] André DeHon. Compact, multilayer layout for butterfly fat-tree. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, SPAA ’00, pages 206–215. ACM, 2000.

[DFH$^+$93] J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In *Parallel Architectures and Languages Europe (PARLE)*, pages 146–160. Springer, 1993.

[DFK$^+$92] W. J. Dally, J. A.S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, 1992.

[DFMP08] R. Dimond, M. J. Flynn, O. Mencer, and O. Pell. MAXware: acceleration in HPC. In *Proceedings of 20th IEEE HOT CHIPS conference*, 2008.

[DG08] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DGTY95] J. Darlington, Y. Guo, H. W. To, and J. Yang. Parallel skeletons for structured composition. *ACM SIGPLAN Notices*, 30(8):19–28, 1995.

[DKM$^+$12] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: Recording microprocessor history. *Communications of the ACM*, 55(4):55–63, April 2012.

[DR81] J. Darlington and M. Reeve. ALICE: a multi-processor reduction machine for the parallel evaluation CF applicative languages. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA ’81, pages 65–76. ACM, 1981.

[DT03] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.

[DW89] W. J. Dally and D. Wills. Universal mechanisms for concurrency. In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *Parallel Architectures and Languages Europe (PARLE)*, volume 365 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 1989.

[DWR95] A. Douglas, A. Wood, and A. Rowstron. Linda implementation revisited. *Transputer and Occam developments*, pages 125–138, 1995.

[Elp11] Elpida Memory, Inc. Elpida begins sample shipments of DDR3 SDRAM (x32) based on TSV stacking technology, June 2011. Press release.

[ESS04] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: Programming for hierarchical parallelism and non-uniform data access (extended abstract). In *Language Runtimes Workshop: Impact of Next Generation Processor Architectures on Virtual Machines (colocated with OOPSLA 2004)*, October 2004.

[FCO90] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[FHK^{+}90] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical report, Center for Research on Parallel Computation, Rice University, Houston, 1990.

[FKT90] I. Foster, C. Kesselman, and S. Taylor. Concurrency: Simple concepts and powerful tools. *The Computer Journal*, 33(6):501–507, 1990.

[Flo79] R. W. Floyd. The paradigms of programming. *Communications of the ACM*, 22(8):455–460, 1979.

[FM11] Samuel H. Fuller and Lynette I. Millett, editors. *The Future of Computing Performance: Game Over or Next Level?* National Acadamies Press, 2011.

[Fos93] I. Foster. Strand and PCN: Two generations of compositional programming languages. Technical Report CRPC-TR93446, Center for Research on Parallel Computation, Rice University, 1993.

[Fos95] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing, 1995.

[Fos96] I. Foster. Compositional parallel programming languages. *ACM Transactions on programming languages and systems*, 18(4):454–476, July 1996.

[FOT92] I. Foster, R. Olson, and S. Tuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1(1):51–66, 1992.

[FT90] Ian Foster and Stephen Taylor. *Strand: new concepts in parallel programming*. Prentice-Hall, Inc., 1990.

[FU92] R. Feldmann and W. Unger. The cube-connected cycles network is a subgraph of the butterfly network. *Parallel Processing Letters*, 02(01):13–19, 1992.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118. ACM, 1978.

[FXA94] I. Foster, M. Xu, and B. Avalani. A compilation system that integrates High Performance Fortran and Fortran M. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 293–300. IEEE, 1994.

[GBC^{+}05] A. Gara, M. A. Blumrich, D. Chen, L-T G. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2):195–212, March 2005.

[GC92] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, 1992.

[GC94] B. Gendron and T. G. Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42(6):1042–1066, 1994.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GGK$^+$82] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - designing a MIMD, shared-memory parallel machine (extended abstract). *SIGARCH Computer Architecture news*, 10(3):27–42, April 1982.

[GKKG03] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing*. Addison-Wesley, second edition, January 2003.

[GKW85] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

[GL88] Ronald I. Greenberg and Charles E. Leiserson. A compact layout for the three-dimensional tree of meshes. *Applied Mathematics Letters*, 1(2):171–176, 1988.

[GL95] W. D. Gropp and E. Lusk. Dynamic process management in an MPI setting. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, pages 530–533. IEEE, 1995.

[GL96] Ronald I. Greenberg and Charles E. Leiserson. Randomized routing on fat-trees. In *Advances in Computing Research*, pages 345–374. JAI Press, 1996.

[Gon81] G. H. Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM*, 28(2):289–304, 1981.

[Goo13] Google Inc. The official Go Language specification, November 2013.

[Got86] A. Gottlieb. An overview of the NYU Ultracomputer project. *Experimental Parallel Computing Architectures*, pages 25–95, 1986.

[Gre11] M. Greenberg. DDR4, higher speeds and larger SoCs: Why external memory latency is getting worse, and what to do about it, November 2011. ARM TechCon 2011.

[Gro92] W. D. Gropp. Parallel computing and domain decomposition. In *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations, Philadelphia, PA*, 1992.

[Han77] P. B. Hansen. Design principles. In *The Architecture of Concurrent Programs*, pages 3–14. Prentice Hall, July 1977.

[Han78] P. B. Hansen. Distributed processes: A concurrent programming concept. *Communications of the ACM*, 21(11):934–941, 1978.

[Han87] P. B. Hansen. Joyce—A programming language for distributed systems. *Software: Practice and Experience*, 17(1):29–50, 1987.

[Han90] P. B. Hansen. The nature of parallel programming. *Natural and Artifical Parallel Computation, (Arbib, M. A. and Ribinson J. A., Eds.)*, pages 31–46, 1990. The MIT Press, Cambridge, MA.

[Han94] P. B. Hansen. SuperPascal: A publication language for parallel scientific computing. In *The origin of concurrent programming: from semaphores to remote procedure calls*, pages 495–524. Springer-Verlag, 1994.

[Han95a] P. B. Hansen. Efficient parallel recursion. *ACM SIGPLAN Notices*, 30(12):9–16, 1995.

[Han95b] P. B. Hansen. *Studies in computational science: parallel programming paradigms*. John Wiley and Sons, Ltd., 1995.

[Han09] James Hanlon. XMP-64 performance measurements. XMOS Ltd., Bristol, UK, 2009.

[HBB^{+}95] J. Harris, J. A. Bircsak, M. R. Bolduc, J. A. Diewald, I. Gale, N. W. Johnson, S. Lee, C. A. Nelson, and C. D. Offner. Compiling High Performance Fortran for distributed-memory systems. *Digital Technical Journal*, 7(3):5–23, 1995.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305 –1320, September 1991.

[HDH^{+}10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, Feburary 2010.

[Hey90] A. Hey. Experiments in MIMD parallelism. *Future Generation Computer Systems*, 6(3):185–196, December 1990.

[HG85] P. Hudak and B. Goldberg. Distributed execution of functional programs using serial combinators. *IEEE Transactions on Computers*, 100(10):881–891, 1985.

[HKwT92a] Seema Hiranandani, Ken Kennedy, and Chau wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35:66–80, 1992.

[HKwT92b] Seema Hiranandani, Ken Kennedy, and Chau wen Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 6th international conference on Supercomputing*, pages 1–14. ACM, 1992.

[HLW06] S. Hoory, N. Linial, and A. Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–562, 2006.

[HMH01] Ron Ho, Ken Mai, and Mark Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[HMH03] Ron Ho, Ken Mai, and Mark Horowitz. Efficient on-chip global interconnects. In *Symposium on VLSI Circuits, Digest of Technical Papers*, pages 271–274. IEEE, 2003.

[HMS^{+}86] John P. Hayes, Trevor Mudge, Quentin F. Stout, Stephen Colley, and John Palmer. A microprocessor-based hypercube supercomputer. *IEEE Micro*, 6(5):6–17, October 1986.

[Ho03] R. Ho. *On-chip wires: scaling and efficiency*. PhD thesis, Stanford, 2003.

[Hoa71] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *In Operating Systems Techniques, Proceedings of a Seminar at Queen's University, Belfast, Northern Ireland.* Academic Press, August–September 1971. Article 5.

[Hoa73] C. A.R. Hoare. Hints on programming language design. Technical Report STAN-CS-73-403, Stanford University, 1973.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall International (UK) Ltd., 1985.

[Hoe12] Bernd Hoefflinger. Towards terabit memories. In B. Hoefflinger, editor, *Chips 2020*, chapter 11. Springer, 2012.

[HOF$^+$12] R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, A. Gara, G. L.-T. Chiu, P. A. Boyle, N. H. Chist, and Changhoan Kim. The IBM Blue Gene/Q compute chip. *IEEE Micro*, 32(2):48–60, March 2012.

[HSJ86] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[Hyb13] Hybrid Memory Cube Consortium. Hybrid Memory Cube Specification 1.0, 2013.

[IBP$^+$05] S. S. Iyer, J. E. Barth, P. C. Parries, J. P. Norum, J. P. Rice, L. R. Logan, and D. Hoyniak. Embedded DRAM: technology platform for the Blue Gene/L chip. *IBM Journal of Research and Development*, 49:333–350, March 2005.

[IF99] Yehea I. Ismail and Eby G. Friedman. Repeater insertion in RLC lines for minimum propagation delay. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 6, pages 404–407. IEEE, 1999.

[INM84] INMOS Ltd. *Occam Programming Manual.* Prentice-Hall International (UK) Ltd., 1984.

[INM88a] INMOS Ltd. *Communicating Process Architecture.* Prentice-Hall International (UK) Ltd., 1988.

[INM88b] INMOS Ltd. *occam 2 Reference Manual.* Prentice-Hall International (UK) Ltd., 1988.

[INM88c] INMOS Ltd. *Transputer Databook.* INMOS Ltd., Bristol, UK, 1988. First Edition.

[Int01] International Technology Roadmap for Semiconductors. Interconnect, 2001.

[Int05] International Technology Roadmap for Semiconductors. Interconnect, 2005.

[Int07] International Technology Roadmap for Semiconductors. Interconnect, 2007.

[Int08] IntellaSys. SEAforth 40C18, scalable embedded array processor, 2008. Datasheet.

[Int10a] International Technology Roadmap for Semiconductors. Assembly and packaging, 2010.

[Int10b] International Technology Roadmap for Semiconductors. Interconnect, 2010.

[Int11] International Technology Roadmap for Semiconductors. Interconnect, 2011.

[Int12a] International Technology Roadmap for Semiconductors. Executive summary, 2012.

[Int12b] International Technology Roadmap for Semiconductors. Interconnect, 2012.

[Int12c] International Technology Roadmap for Semiconductors. System drivers, 2012.

[Int13] Intel Corporation. Intel Xeon Phi coprocessor, June 2013. Datasheet 328209-002EN.

[Ito01] Kiyoo Itoh. *VLSI Memory Chip Design.* Springer, 2001.

[IWM$^+$02] Gordon M. I., Thies W., Karczmarek M., Wong J., Hoffman H., Maze D. Z., and Amarasinghe S. A stream compiler for communication-exposed architectures. Technical Report MIT/LCS Technical Memo LCS-TM-627, Massachusetts Institute of Technology, May 2002.

[JBK$^+$09] Ajay Joshi, Christopher Batten, Yong-Jin Kwon, Scott Beamer, Imran Shamim, Krste Asanovic, and Vladimir Stojanovic. Silicon-photonic Clos networks for global on-chip communication. In *Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '09, pages 124–133. IEEE, 2009.

[JDFJ97] A. M. Jones, N. J. Davies, M. A. Firth, and Wright C. J. *The Network Designer's Handbook*. IOS Press, 1st edition, 1997.

[JED11] JEDEC Solid State Technology Association. Wide I/O single data rate (Wide I/O SDR), December 2011. JESD229.

[JG88] G. Jones and M. Goldsmith. *Programming in occam 2. International series in computer science.* Prentice-Hall International (UK) Ltd., 1988.

[JM02] A. M. Jones and M. D. May. Microcomputer with packet translation for event packets and memory access packets, May 2002. Patent, STMicroelectonics Ltd., number US 6,397,325 B1.

[JNW07] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann, 2007.

[Jon10] Handel H. Jones. Technical viability of stacked silicon interconnect technology. Technical report, IBS Inc., October 2010.

[Kal12] Kalray. Kalray MPPA Manycore processors, September 2012. Product brief.

[KDTG05] J. Kim, W. J. Dally, B. Towles, and A. Gupta. Microarchitecture of a high-radix router. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, pages 420–431. IEEE, 2005.

[KKS$^+$07] A Kumary, P Kunduz, AP Singhx, L-S Pehy, and NK Jhay. A 4.6 Tbits/s 3.6 GHz single-cycle NoC router with a novel switch allocator in 65nm CMOS. In *25th International Conference on Computer Design (ICCD)*, pages 63–70. IEEE, 2007.

[KLS93] C. H. Koelbel, D. B. Loveman, and R. S. Schreiber. *The High Performance Fortran handbook.* MIT press, 1993.

[Knu99] Donald E. Knuth. *The art of computer programming*, volume 3, Sorting and searching. Addison-Wesley, 1999.

[KOH$^+$94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, et al. The Stanford FLASH multiprocessor. In *Proceedings the 21st Annual International Symposium on Computer Architecture*, pages 302–313. IEEE, 1994.

[KPA$^+$06] J. U. Knickerbocker, C. S. Patel, P. S. Andry, C. K. Tsang, L. P. Buchwalter, E. J. Sprogis, H. Gan, R. R. Horton, R. J. Polastre, S. L. Wright, et al. 3-D silicon integration and silicon packaging technology using silicon through-vias. *IEEE Journal of Solid-State Circuits*, 41(8):1718–1725, 2006.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice-Hall, Inc., 2nd edition, 1988.

[KR90] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In Jan van Leeuwen, editor, *Handbook of theoretical computer science (vol. A)*, pages 869–941. MIT Press, 1990.

[KRS90] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95 – 132, 1990.

[KRSG94] LV Kalé, B. Ramkumar, AB Sinha, and A. Gursoy. The Charm parallel programming language and system: Part I–description of language features. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[KS93] R. E. Kessler and J. L. Schwarzmeier. CRAY T3D: A new dimension for Cray Research. In *Compcon, Digest of Papers.*, pages 176–182. IEEE, 1993.

[KU88] A. R. Karlin and E. Upfal. Parallel hashing: An efficient implementation of shared memory. *Journal of the ACM*, 35(4):876–892, 1988.

[Kun82] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, 1982.

[Kun84] S. Y. Kung. On supercomputing with systolic/wavefront array processors. *Proceedings of the IEEE*, 72(7):867–884, July 1984.

[Kun88a] H. T. Kung. Computational models for parallel computers. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 326(1591):357–371, 1988.

[Kun88b] S. Y. Kung. *VLSI array processors.* Prentice-Hall, Inc., 1988.

[KYAC11] Yu-Hsiang Kao, Ming Yang, N. S. Artan, and H. J. Chao. Cnoc: High-radix clos network-on-chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(12):1897 –1910, December 2011.

[LAD$^+$92] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong-Chan, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 272–285. ACM, 1992.

[LCD$^+$08] H. W. Lean, P. A. Clark, M. Dixon, N. M. Roberts, A. Fitch, R. Forbes, and C. Halliwell. Characteristics of high-resolution versions of the Met Office unified model for forecasting convection over the United Kingdom. *Monthly Weather Review*, 136(9):3408–3424, 2008.

[Lei85] Charles E. Leiserson. Fat trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, October 1985.

[Lei92] Frank T. Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes.* Morgan Kaufmann, 1992.

[LGM$^+$09] D. Ludovici, F. Gilabert, S. Medardoni, C. Gómez, M. E. Gómez, P. López, G. N. Gaydadjiev, and D. Bertozzi. Assessing fat-tree topologies for regular network-on-chip design under nanoscale technology constraints. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 562–565. European Design and Automation Association, 2009.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.

[LM89] Frank T. Leighton and Bruce Maggs. Expanders might be practical: Fast algorithms for routing around faults on multibutterflies. In *30th Annual Symposium on the Foundations of Computer Science*, pages 384–389. IEEE, 1989.

[LMHL05] X. C. Li, J. F. Mao, H. F. Huang, and Y. Liu. Global interconnect width and spacing optimization for latency, bandwidth and power dissipation. *IEEE Transactions on Electron Devices*, 52(10):2272–2279, 2005.

[May83] D. May. OCCAM. *ACM SIGPLAN Notices*, 18(4):69–79, 1983.

[May88] D. May. The influence of VLSI technology on computer architecture. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 326(1591):377–393, 1988.

[May91] D. May. Compiling occam into silicon. In *Developments in concurrency and communication*, pages 87–106. Addison-Wesley Longman Publishing Co., Inc., 1991.

[May94] D. May. How to design a parallel computer. In *A classical mind: essays in honour of C. A. R. Hoare*, pages 275–294. Prentice Hall, 1994.

[May99] D. May. The transputer revisited. In *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*, pages 215–246. Palgrave Macmillan, 1999.

[May09] D. May. *The XMOS XS1 Architecture.* XMOS Ltd., October 2009.

[May10] D. May. Communicating process architecture for multicores. *Concurrency: Practice and Experience*, 22(8):935–948, 2010.

[May11] D. May. XMOS Architecture: XS1 chips. *IEEE Micro*, 32:28–37, August 2011.

[MC80] Carver Mead and Lynn Conway. *Introduction to VLSI systems.* Addison-Wesley series in computer science. Addison-Wesley, 1980.

[McC93] W. F. McColl. General purpose parallel computing. *Lectures on parallel computation*, 4:337–391, 1993.

[McC94] W. F. McColl. BSP programming. In *Proceedings of the DIMACS Workshop*, volume 18, pages 21–35, 1994.

[MD94] T. Mackenzie and T. Dix. A distributed memory multiprocessor implementation of C-with-Ease. In *International Conference on Parallel and Distributed Systems*, pages 250 –257, December 1994.

[Mei88] Meiko Ltd. The Meiko computing surface: an example of a massively parallel system. In *Proceedings of the third conference on Hypercube concurrent computers and applications: Architecture, software, computer systems, and general issues*, volume 1 of *C^3P*, pages 852–859. ACM, 1988.

[Mes09] Message Passing Interface Forum. *MPI: A message-passing interface standard*, September 2009.

[MFLA99] C. A. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot pages: Software caching for RAW microprocessors. Technical Report LCS-TM-599, MIT,Cambridge, MA, August 1999.

[Mic12a] Micron. 1Gb: x4, x8, x16 DDR3 SDRAM: MT41J128M8JP-125 features, 2012. Datasheet.

[Mic12b] Micron. 512Mb: x4, x8, x16 DDR SDRAM: MT46V128M4BN-5B features, 2012. Datasheet.

[Mic12c] Micron. 512Mb: x4, x8, x16 DDR2 SDRAM: MT47H128M4CF-25E features, 2012. Datasheet.

[Mic12d] Micron. 512Mb: x4, x8, x16 SDR SDRAM: MT48LC128M4A2P-75 features, 2012. Datasheet.

[Mil82] Robin Milner. *A calculus of communicating systems.* Springer-Verlag, 1982.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, June 1999.

[Mil10] David A. B. Miller. Optical interconnects to electronic chips. *Applied optics*, 49(25):F59–F70, September 2010.

[Mis86] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, 1986.

[Mit91] M. D. Mitzenmacher. *The power of two choices in randomized load balancing.* PhD thesis, Harvard University, 1991.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[MPJ$^+$00] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: a modular reconfigurable architecture. *SIGARCH Computer Architecture News*, 28(2):161–171, 2000.

[MS87] D. McBurney and M. R. Sleep. Transputer-based experiments with the ZAPP architecture. In *Parallel Architectures and Languages Europe (PARLE)*, pages 242–259. Springer, 1987.

[MS88] D. McBurney and M. R. Sleep. Transputers + virtual tree kernel = real speedups. In *Proceedings of the third conference on Hypercube concurrent computers and applications: Architecture, software, computer systems, and general issues*, volume 1, pages 128–137. ACM, 1988.

[MSA$^+$83] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee. SISAL: streams and iteration in a single-assignment language. Technical report, Lawrence Livermore National Laboratory, CA (USA), 1983. Language reference manual, Version 1.1.

[MSM04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming.* Addison-Wesley Professional, 2004.

[MTW93] D. May, P. W. Thompson, and P. H. Welch, editors. *Networks, Routers and Transputers: Function, Performance and Applications.* IOS Press, 1st edition, 1993.

[MTWS78] D. May, R. J.B. Taylor, and C. Whitby-Strevens. EPL: An experimental language for distributed computing. In *Proceedings of Trends and Applications: Distributed Processing, Natlonal Bureau of Standards*, pages 69–71, May 1978.

[MWM04] R. Mullins, A. West, and S. Moore. Low-latency virtual-channel routers for on-chip networks. In *ACM SIGARCH Computer Architecture News*, volume 32, page 188. IEEE, 2004.

[Myc07] A. Mycroft. Programming language design and analysis motivated by hardware evolution (invited presentation). In *Proceedings of SAS'07*, volume 3634, pages 18–33. Springer-Verlag, August 2007.

[NBB$^+$63] P. Naur, J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. Mc-Carthy, A. J Perlis, H. Rutishauser, K. Samelson, B. Vauquois, et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[NHL94] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A portable shared-memory programming model for distributed memory computers. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 340–349. ACM, 1994.

[NR98] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[NVI12] NVIDIA Corporation. NVIDIA K-Series Tesla GPU accelerators, 2012. Datasheet.

[NWD93] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-machine multicomputer: an architectural evaluation. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 224–235. ACM, 1993.

[OIDK95] S. R. Ohring, M. Ibel, S. K. Das, and M. J. Kumar. On generalized fat trees. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 37–44, April 1995.

[OW10] R. Osborne and D. Watt. *Tools Developer Guide*. XMOS Ltd., Bristol, UK, May 2010.

[PAC$^+$97] D. A. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997.

[Pat04] D. A. Patterson. Latency lags bandwith. *Communications of the ACM*, 47:71–75, October 2004.

[Pel94] F. J. Pelletier. The principle of semantic compositionality. *Topoi*, 13(1):11–24, 1994.

[Pet77] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[Pfe07] Charles Pfeil. BGA breakout challenges. *OnBoard Technology*, pages 10–13, October 2007.

[PFH$^+$02] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, 2002.

[PJ89] S. L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[PJCSH87] S. L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP—a high-performance architecture for parallel graph reduction. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 1987.

[Pol99] Fred J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32. IEEE, 1999.

[PTD$^+$06] Gajinder Panesar, Daniel Towner, Andrew Duller, Alan Gray, and Will Robbins. Deterministic parallel processing. *International Journal of Parallel Programming*, 34(4):323–341, August 2006.

[PV81] Franco P. Preparata and Jean Vuillemin. The cube-connected cycles: a versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[PV97] F. Petrini and M. Vanneschi. k-ary n-trees: High performance networks for massively parallel architectures. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 87–93. IEEE, 1997.

[PvE93] Rinus Plasmeijer and Marko van Eekelen. *Functional programming and parallel graph rewriting*. Addison-Wesley, 1993.

[Ram11] Suresh Ramalingam. Stacked silicon interconnect technology (SSIT) qualification – requirements and tools, July 2011. Presentation given at the 3D Stress Workshop.

[Ran87] Abhiram G. Ranade. How to emulate shared memory. In *Proceedings of the 28th annual IEEE symposium on Foundations of Computer Science*, pages 185–194, 1987.

[RCBJ11] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters*, 10(1):16–19, January 2011.

[RH88] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theoretical Computer Science*, 60(2):177–229, 1988.

[Ric67] Martin Richards. The BCPL reference manual. Technical report, Project MAC, Massachusetts Institute of Technology, July 1967. Memorandum M-352.

[RKB$^+$09] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: challenges in and avenues for CMP scaling. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 371–382. ACM, 2009.

[RS90] Sanjay Ranka and Sartaj Sahni. *Hypercube algorithms for image processing and pattern recognition*. Springer-Verlag, 1990.

[RSSK94] B. Ramkumar, AB Sinha, VA Saletore, and LV Kale. The Charm parallel programming language and system: Part II–the runtime system. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[Rus78] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[RWS79] Martin Richards and Colin Whitby-Strevens. *BCPL - the language and its compiler*. Cambridge University Press, 1979.

[Sab11] Kirk Saban. Xilinx stacked silicon interconnect technology delivers breakthrough FPGA capacity, bandwidth, and power efficiency. Technical report, Xilinx Inc., October 2011.

[SAKD06] S. Scott, D. Abts, J. Kim, and W. J. Dally. The BlackWidow high-radix Clos network. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 16–28. IEEE, 2006.

[Sch80] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, October 1980.

[Sch98] Christian Scheideler. *Universal routing strategies for interconnection networks*, volume 1390. Springer-Verlag, 1998.

[SDB93] A. Skjellum, N. E. Doss, and P. V. Bangalore. Writing libraries in MPI. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 166–173, October 1993.

[SDM+12a] S. Satpathy, R. Dreslinski, T. Manville, D. Sylvester, T. Mudge, and D. Blaauw. A 4.5Tb/s 3.4Tb/s/W 64x64 switch fabric with self-updating least recently granted priority and quality of service arbitration in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, pages 478–479, February 2012.

[SDM+12b] K. Sewell, R. G. Dreslinski, T. Manville, S. Satpathy, N. Pinckney, G. Blake, M. Cieslak, R. Das, T. F. Wenisch, D. Sylvester, et al. Swizzle-Switch networks for many-core systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2(2):278–294, 2012.

[SDTO+11] S. Satpathy, R. Dreslinski, D. Sylvester T. Ou, T. Mudge, and D. Blaauw. SWIFT: A 2.1tb/s 32x32 self-arbitrating manycore interconnect fabric. In *Symposia on VLSI Technology and Circuits, Koyoto, Japan*, pages 138–139, June 2011.

[Sei85] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, 1985.

[SGS95] SGS-Thompson Microelectronics Ltd., Bristol, UK. *occam 2.1 reference manual*, May 1995.

[Sto71] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, 20(2):153–161, Feburary 1971.

[TCM+09] D. N. Truong, W. H. Cheng, T. Mohsenin, Z. Yu, A. T. Jacobson, G. Landge, M. J. Meeuwsen, C. Watnik, A. T. Tran, Z. Xiao, et al. A 167-processor computational platform in 65nm CMOS. *IEEE Journal of Solid-State Circuits*, 44(4):1130–1144, 2009.

[Tez10] Tezzaron Semiconductor. Octopus 8-port DRAM for die-stack applications, 2010. Datasheet.

[THLPJ98] P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *Journal of functional programming*, 8(1):23–60, 1998.

[TKA02] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*, pages 49–84. Springer, 2002.

[TMHAJ08] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. CACTI 5.1. Technical report, HP Laboratories, 2008.

[TR88] L. W. Tucker and G. G. Robertson. Architecture and applications of the Connection Machine. *IEEE Computer*, 21(8):26–38, 1988.

[Tur37] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, January 1937.

[Tur46] A. M. Turing. Proposals for development in the mathematics division of an automatic computing engine (ACE). Technical Report Report E882, Executive Committee, National Physical Laboratory (NPL) in Teddington, England, Feburary 1946. Reprinted April 1972 as NPL Report Com. Sci 57.

[Ull84] J. D. Ullman. *Computational Aspects of VLSI.* W. H. Freeman & Co., 1984.

[Upf84] E. Upfal. Efficient schemes for parallel communication. *Journal of the ACM*, 31(3):507–517, June 1984.

[Upf92] E. Upfal. An O(log N) deterministic packet-routing scheme. *Journal of the ACM*, 39(1):55–70, 1992.

[Val82] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, 1982.

[Val88] L. G. Valiant. Optimally universal parallel computers. In *Scientific applications of multiprocessors*, pages 17–20. Prentice Hall, 1988.

[Val90a] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Val90b] L. G. Valiant. General purpose parallel architectures. In *Handbook of theoretical computer science (vol. A): algorithms and complexity*, pages 943–973. MIT Press, 1990.

[VB81] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 263–277. ACM, 1981.

[VLT87] J. Van Leeuwen and R. B. Tan. Interval routing. *The Computer Journal*, 30(4):298–307, 1987.

[von45] J. von Neumann. First draft of a report on the EDVAC. Technical Report Contract No. W-670-ORD-4926, Moore School of Electical Engineering, University of Pennsylvania, June 1945. Reprinted in *Annals of the History of Computing, IEEE*, 15(4), 1993, pages 27-75.

[Wat09] Douglas Watt. *Programming XC on XMOS Devices.* XMOS Ltd., September 2009.

[WB05] P. H. Welch and F. R.M. Barnes. Communicating mobile processes: introducing occam-pi. In A. E. Abdallah, C. B. Jones, and J. W. Sanders, editors, *25 Years of CSP*, Lecture Notes in Computer Science, pages 175–210. Springer-Verlag, April 2005.

[WD96] David W. Walker and Jack J. Dongarra. MPI: A standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[Wei84] Reinhold P Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[Wel92] P. H. Welch. The role and future of occam. In *Transputer Applications – Progress and Prospects (Proceedings of the Closing Symposium of the SERC/DTI Initiative in the Engineering Applications of Transputers), pub. IOS Press, Amsterdam (ISBN 90-5199-079-0)*, pages 152–169, 1992.

[WGH$^+$07] D. Wentzlaff, P. Griffin, H. Hoffmann, Liewei Bao, B. Edwards, C. Ramey, M. Mattina, Chyi-Chang Miao, J. F. Brown, and A. Agarwal. On-chip interconnection architecture of the Tile processor. *IEEE Micro*, 27(5):15–31, September 2007.

[WH88] D. H. D. Warren and S. Haridi. Data diffusion machine – a scalable shared virtual memory multiprocessor. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 943–952, 1988.

[Whe50] D. J. Wheeler. Programme organization and initial orders for the EDSAC. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 202(1071):573–589, 1950.

[Wil68] M. V. Wilkes. Computers then and now. *Journal of the ACM*, 15(1):1–7, 1968.

[Wir71] Niklaus Wirth. The programming language Pascal. *Acta informatica*, 1(1):35–63, 1971.

[WJNB95] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Memory Management*, pages 1–116. Springer, 1995.

[WM05] Neil H. E. Weste and David Money. *CMOS VLSI Design.* Pearson/Addison-Wesley, fourth edition, 2005.

[WP94] P. G. Whiting and R. S.V. Pascoe. A history of data-flow languages. *IEEE Annals of the History of Computing*, 16(4):38–59, 1994.

[WRRF05] M. Woodacre, D. Robb, D. Roe, and K. Feind. The SGI AltixTM 3000 global shared-memory architecture, 2005. Silicon Graphics International.

[WTS+97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: RAW machines. *IEEE Computer*, 30(9):86–93, September 1997.

[XMO10] XMOS Ltd., Bristol, UK. *XK-XMP-64 Hardware Manual*, Feburary 2010.

[XMO12a] XMOS Ltd., Bristol, UK. *XS1-G System Specification*, March 2012.

[XMO12b] XMOS Ltd., Bristol, UK. *XS1-G04B-FB512 Datasheet*, October 2012.

[YKM+11] Marcelo Yuffe, Ernest Knoll, Moty Mehalel, Joseph Shor, and Tsvika Kurts. A fully integrated multi-CPU, GPU and memory controller 32nm processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 264–266. IEEE, 2011.

[YMA+08] Z. Yu, M. J. Meeuwsen, R. W. Apperson, O. Sattari, M. Lai, J. W. Webb, E. W. Work, D. Truong, T. Mohsenin, and B. M. Baas. AsAP: An asynchronous array of simple processors. *IEEE Journal of Solid-State Circuits*, 43(3):695–705, March 2008.

[Yor02] Richard York. Benchmarking in context: Dhrystone. White Paper, ARM Ltd., Cambridge, UK, 2002.

[YSP+98] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, et al. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.

[Yua11] Xin Yuan. On nonblocking folded-Clos networks in computer communication environments. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 188–196, May 2011.

[Zen90] S. E. Zenith. Linda coordination language: Subsystem kernel architecture (on transputers). Technical Report Research Report YALEU/DSC/RR-794, Yale University, May 1990.

[Zen92a] S. E. Zenith. Ease: the model and its implementation. In *Proceeding of a Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Multiprocessors*, September 1992. Appeared as SIGPLAN Not. 28, 1 (Jan.), 1993, 87.

[Zen92b] Steven E. Zenith. A rationale for programming with Ease. *Reasearch Directions in High-Level Parallel Programming Languages*, pages 147–156, 1992.

[ZGY10] Jing Zhang, Huaxi Gu, and Yintang Yang. A high performance optical network on chip based on Clos topology. In *2nd International Conference on Future Computer and Communication (ICFCC)*, volume 2, pages V2–63 –V2–68, May 2010.

APPENDICES

APPENDIX A.

SIRE SYNTAX

A.1. Collected syntax

The following sections list the sire syntax, with each section collecting related to a particular feature, similar to the presentation in Chapter 6.

A.1.1. Program

$$\begin{aligned}
\textit{program} &= \langle\text{program-specification}\rangle\ \texttt{:}\ \langle\text{program}\rangle \\
&\mid \langle\text{sequence}\rangle \\
\textit{program-specification} &= \langle\text{specification}\rangle \\
&\mid \langle\text{definition}\rangle \\
\textit{definition} &= \langle\text{simultaneous-definition}\rangle \\
\textit{simultaneous-definition} &= \{_0\ \texttt{\&}\ \langle\text{definition}\rangle\ \}
\end{aligned}$$

A.1.2. Specifications

$$\begin{aligned}
\textit{specification} &= \langle\text{declaration}\rangle \\
&\mid \langle\text{abbreviation}\rangle
\end{aligned}$$

Declarations

$$\begin{aligned}
\textit{declaration} &= \langle\text{type}\rangle\ \{_1\ \texttt{,}\ \langle\text{name}\rangle\ \} \\
\textit{type} &= \langle\text{primitive-type}\rangle \\
&\mid \langle\text{array-type}\rangle \\
\textit{primitive-type} &= \texttt{var} \\
\textit{array-type} &= \langle\text{primitive-type}\rangle \\
&\mid \langle\text{array-type}\rangle\ \texttt{[}\ \langle\text{expression}\rangle\ \texttt{]}
\end{aligned}$$

Abbreviations

$$\begin{aligned}
\textit{abbreviation} &= \langle\text{specifier}\rangle\ \langle\text{name}\rangle\ \texttt{is}\ \langle\text{element}\rangle \\
&\mid \texttt{val}\ \langle\text{name}\rangle\ \texttt{is}\ \langle\text{expression}\rangle \\
\textit{specifier} &= \langle\text{primitive-type}\rangle \\
&\mid \langle\text{specifier}\rangle\ \texttt{[ ]} \\
&\mid \langle\text{specifier}\rangle\ \texttt{[}\ \langle\text{expression}\rangle\ \texttt{]}
\end{aligned}$$

A.1.3. Sequence

$$\begin{aligned}
\textit{sequence} &= \{_0\ \texttt{;}\ \langle\text{command}\rangle\ \} \\
\textit{command} &= \langle\text{primitive-command}\rangle \\
&\mid \langle\text{structured-command}\rangle
\end{aligned}$$

	|	⟨specification⟩ **:** ⟨command⟩

Primitive commands

primitive-command	=	⟨skip⟩
	|	⟨stop⟩
	|	⟨assingment⟩
	|	⟨input⟩
	|	⟨output⟩
	|	⟨connect⟩
skip	=	**skip**
stop	=	**stop**
assignment	=	⟨variable⟩ **:=** ⟨expression⟩
input	=	⟨chanend⟩ **?** ⟨variable⟩
output	=	⟨chanend⟩ **!** ⟨expression⟩
connect	=	**connect** ⟨chanend⟩ **to** ⟨chanend⟩

Structured commands

structured-command	=	⟨alternation⟩
	|	⟨conditional⟩
	|	⟨loop⟩
	|	**{** ⟨sequence⟩ **}**
	|	**{** ⟨parallel⟩ **}**
conditional	=	**if {** $\{_0$ **|** ⟨choice⟩ **}**
	|	**if** ⟨expression⟩ **then** ⟨command⟩ **else** ⟨command⟩
choice	=	⟨guarded-choice⟩
	|	⟨conditional⟩
	|	⟨specification⟩ **:** ⟨choice⟩
guarded-choice	=	⟨expression⟩ **:** ⟨command⟩
alternation	=	**alt {** $\{_0$ **|** ⟨alternative⟩ $\}$ **}**
alternative	=	⟨guarded-alternative⟩
	|	⟨alternation⟩
	|	⟨specification⟩ **:** ⟨alternative⟩
guarded-alternative	=	⟨guard⟩ **:** ⟨command⟩
guard	=	⟨input⟩
	|	⟨expression⟩ **&** ⟨input⟩
	|	⟨expression⟩ **&** ⟨skip⟩
loop	=	**while** ⟨expression⟩ **do** ⟨command⟩
parallel	=	$\{_0$ **&** ⟨parallel-component⟩ $\}$
parallel-component	=	⟨process-label⟩ ⟨process⟩
	|	⟨process⟩
process-label	=	⟨name⟩ **is**

A.1.4. Process

process = ⟨interface⟩ **:** ⟨command⟩
| ⟨command⟩
interface = **interface (** $\{_0$ **,** ⟨declaration⟩ $\}$ **)**
primitive-type = ⟨chanend-type⟩
chanend-type = **chanend**

A.1.5. Server

Specification

server = ⟨interface⟩ **:** ⟨server-specification⟩
primitive-type = ⟨call-type⟩
call-type = **call**
declaration = ⟨type⟩ $\{_1$ **,** ⟨name⟩ **(** $\{_0$ **,** ⟨formal⟩ $\}$ **)** $\}$
server-specification = ⟨declaration⟩
| **{** $\{_1$ **:** ⟨declaration⟩ $\}$ **}**
declaration = **initial** ⟨command⟩
| **final** ⟨command⟩
| ⟨alternation⟩
guard = **accept** ⟨name⟩ **(** $\{_0$ **,** ⟨formal⟩ $\}$ **)**
| ⟨expression⟩ **& accept** ⟨name⟩ **(** $\{_0$ **,** ⟨formal⟩ $\}$ **)**

Declarations

declaration = ⟨server-declaration⟩
| ⟨hiding-declaration⟩
| ⟨simultaneous-declaration⟩
server-declaration = ⟨name⟩ **is** ⟨server⟩
server = ⟨server-array⟩
server-array = **[** $\{_1$ **,** ⟨server⟩ $\}$ **]**
hiding-declaration = **from {** $\{_1$ **:** ⟨declaration⟩ $\}$ **} interface** ⟨name⟩
simultaneous-declaration = $\{_0$ **&** ⟨declaration⟩ $\}$

Call

command = ⟨server-call⟩
server-call = ⟨element⟩ **(** $\{_0$ **,** ⟨actual⟩ $\}$ **)**

A.1.6. Replication

command = **seq** ⟨replicator⟩ ⟨command⟩
process = **par** ⟨replicator⟩ ⟨process⟩
conditional = **if** ⟨replicator⟩ ⟨choice⟩
alternation = **alt** ⟨replicator⟩ ⟨alternative⟩
server-declaration = ⟨name⟩ **is** ⟨replicator⟩ ⟨server⟩

	|	⟨name⟩ **`is [`** ⟨expression⟩ **`]`** ⟨server⟩
declaration	=	**`alt`** ⟨replicator⟩ ⟨alternative⟩
replicator	=	**`[`** $\{_1$ **`,`** ⟨index-range⟩ $\}$ **`]`**
index-range	=	⟨name⟩ **`=`** ⟨expression⟩ **`for`** ⟨expression⟩
	|	⟨name⟩ **`=`** ⟨expression⟩ **`for`** ⟨expression⟩ **`step`** ⟨expression⟩

A.1.7. Expressions

expression	=	⟨unary-operator⟩ ⟨operand⟩
	|	⟨operand⟩ ⟨binary-operator⟩ ⟨operand⟩
	|	⟨operand⟩
operand	=	⟨element⟩
	|	⟨literal⟩
	|	**`(`** ⟨expression⟩ **`)`**

Valof

valof	=	**`valof`** ⟨process⟩ **`result`** ⟨expression⟩
	|	⟨specification⟩ **`:`** ⟨valof⟩
expression	=	**`(`** ⟨valof⟩ **`)`**

A.1.8. Elements

chanend	=	⟨element⟩
element	=	⟨element⟩ **`[`** ⟨expression⟩ **`]`**
	|	⟨field⟩
	|	⟨name⟩
field	=	⟨element⟩ **`.`** ⟨name⟩
variable	=	⟨element⟩
literal	=	⟨decimal-integer⟩
	|	**`#`** ⟨hexdecimal-integer⟩
	|	⟨byte⟩
	|	**`true`**
	|	**`false`**
hexdecimal-integer	=	⟨hexdecimal-digit⟩
	|	⟨hexdecimal-digit⟩ ⟨hexdecimal-integer⟩
decimal-integer	=	⟨digit⟩
	|	⟨digit⟩ ⟨decimal-integer⟩
byte	=	**`'`** ⟨character⟩ **`'`**

A.1.9. Abstraction

Definitions

definition	=	⟨procedure⟩
	|	⟨function⟩

procedure = **process** ⟨name⟩ **(** $\{_0$ **,** ⟨formal⟩ $\}$ **)** **is** ⟨process⟩
| **server** ⟨name⟩ **(** $\{_0$ **,** ⟨formal⟩ $\}$ **)** **is** ⟨server⟩
| **server** ⟨name⟩ **(** $\{_0$ **,** ⟨formal⟩ $\}$ **)** **inherits** ⟨hiding-declaration⟩

function = **function** ⟨name⟩ **(** $\{_0$ **,** ⟨formal⟩ $\}$ **)** **is** ⟨valof⟩

formal = ⟨specifier⟩ $\{_1$ **,** ⟨name⟩ $\}$
| ⟨call-type⟩ $\{_1$ **,** ⟨name⟩ **(** $\{_0$ **,** ⟨formal⟩ $\}$ **)** $\}$
| **val** $\{_1$ **,** ⟨name⟩ $\}$

abbreviation = ⟨specifier⟩ ⟨name⟩ **(** $\{_0$ **,** ⟨formal⟩ $\}$ **)** **is**
| ⟨call-type⟩ ⟨name⟩ **(** $\{_0$ **,** ⟨formal⟩ $\}$ **)** **is** ⟨name⟩

primitive-type = ⟨process-type⟩
| ⟨server-type⟩

process-type = **process** ⟨name⟩
| **process** ⟨interface⟩

server-type = **server** ⟨name⟩
| **server** ⟨interface⟩

call-type = **process**
| **function**

Instances

process = ⟨instance⟩

server = ⟨instance⟩

command = ⟨instance⟩

expression = ⟨instance⟩

instance = ⟨name⟩ **(** $\{_0$ **,** ⟨actual⟩ $\}$ **)**

actual = ⟨element⟩
| ⟨expression⟩

A.1.10. Compiler transformations

Process distribution

process = **on** ⟨expression⟩ **do** ⟨process⟩

Channel end references

absolute-reference = **(** ⟨expression⟩ **:** ⟨expression⟩ **:** ⟨expression⟩ **)**

interface = **interface** **(** $\{_0$ **,** ⟨declaration⟩ $\}$ **)** **@** ⟨absolute-reference⟩

server-call = ⟨local-chanend⟩ **!** ⟨element⟩ **(** $\{_0$ **,** ⟨actual⟩ $\}$ **)**

local-chanend = ⟨element⟩ **@** ⟨absolute-reference⟩

chanend = ⟨remote-chanend⟩

remote-chanend = **@** ⟨absolute-reference⟩ **.** ⟨expression⟩

A.2. Ordered syntax

The following lists each element of the sire syntax in alphabetical order.

abbreviation = **val** ⟨name⟩ **is** ⟨expression⟩
| ⟨call-type⟩ ⟨name⟩ **(** {$_0$ **,** ⟨formal⟩ } **)** **is** ⟨name⟩
| ⟨specifier⟩ ⟨name⟩ **(** {$_0$ **,** ⟨formal⟩ } **)** **is**
| ⟨specifier⟩ ⟨name⟩ **is** ⟨element⟩

absolute-reference = **(** ⟨expression⟩ **:** ⟨expression⟩ **:** ⟨expression⟩ **)**

actual = ⟨element⟩
| ⟨expression⟩

alternation = **alt** **{** {$_0$ **|** ⟨alternative⟩ } **}**
| **alt** ⟨replicator⟩ ⟨alternative⟩

alternative = ⟨alternation⟩
| ⟨guarded-alternative⟩
| ⟨specification⟩ **:** ⟨alternative⟩

array-type = ⟨array-type⟩ **[** ⟨expression⟩ **]**
| ⟨primitive-type⟩

assignment = ⟨variable⟩ **:=** ⟨expression⟩

byte = **'** ⟨character⟩ **'**

call-type = **call**
| **function**
| **process**

chanend = ⟨element⟩
| ⟨remote-chanend⟩

chanend-type = **chanend**

choice = ⟨conditional⟩
| ⟨guarded-choice⟩
| ⟨specification⟩ **:** ⟨choice⟩

command = **seq** ⟨replicator⟩ ⟨command⟩
| ⟨instance⟩
| ⟨primitive-command⟩
| ⟨server-call⟩
| ⟨specification⟩ **:** ⟨command⟩
| ⟨structured-command⟩

conditional = **if** **{** {$_0$ **|** ⟨choice⟩ } **}**

| **`if`** ⟨expression⟩ **`then`** ⟨command⟩ **`else`** ⟨command⟩
| **`if`** ⟨replicator⟩ ⟨choice⟩

connect = **`connect`** ⟨chanend⟩ **`to`** ⟨chanend⟩

decimal-integer = ⟨digit⟩
| ⟨digit⟩ ⟨decimal-integer⟩

declaration = **`alt`** ⟨replicator⟩ ⟨alternative⟩
| **`final`** ⟨command⟩
| **`initial`** ⟨command⟩
| ⟨alternation⟩
| ⟨hiding-declaration⟩
| ⟨server-declaration⟩
| ⟨simultaneous-declaration⟩
| ⟨type⟩ {$_1$ **`,`** ⟨name⟩ **`(`** {$_0$ **`,`** ⟨formal⟩ } **`)`** }
| ⟨type⟩ {$_1$ **`,`** ⟨name⟩ }

definition = ⟨function⟩
| ⟨procedure⟩
| ⟨simultaneous-definition⟩

element = ⟨element⟩ **`[`** ⟨expression⟩ **`]`**
| ⟨field⟩
| ⟨name⟩

expression = **`(`** ⟨valof⟩ **`)`**
| ⟨instance⟩
| ⟨operand⟩
| ⟨operand⟩ ⟨binary-operator⟩ ⟨operand⟩
| ⟨unary-operator⟩ ⟨operand⟩

field = ⟨element⟩ **`.`** ⟨name⟩

formal = **`val`** {$_1$ **`,`** ⟨name⟩ }
| ⟨call-type⟩ {$_1$ **`,`** ⟨name⟩ **`(`** {$_0$ **`,`** ⟨formal⟩ } **`)`** }
| ⟨specifier⟩ {$_1$ **`,`** ⟨name⟩ }

function = **`function`** ⟨name⟩ **`(`** {$_0$ **`,`** ⟨formal⟩ } **`)`** **`is`** ⟨valof⟩

guard = **`accept`** ⟨name⟩ **`(`** {$_0$ **`,`** ⟨formal⟩ } **`)`**
| ⟨expression⟩ **`&`** **`accept`** ⟨name⟩ **`(`** {$_0$ **`,`** ⟨formal⟩ } **`)`**
| ⟨expression⟩ **`&`** ⟨input⟩
| ⟨expression⟩ **`&`** ⟨skip⟩
| ⟨input⟩

guarded-alternative = ⟨guard⟩ **`:`** ⟨command⟩

guarded-choice = ⟨expression⟩ **:** ⟨command⟩

hexdecimal-integer = ⟨hexdecimal-digit⟩
| ⟨hexdecimal-digit⟩ ⟨hexdecimal-integer⟩

hiding-declaration = **from {** $\{_1$ **:** ⟨declaration⟩ $\}$ **}** **interface** ⟨name⟩

index-range = ⟨name⟩ **=** ⟨expression⟩ **for** ⟨expression⟩
| ⟨name⟩ **=** ⟨expression⟩ **for** ⟨expression⟩ **step** ⟨expression⟩

input = ⟨chanend⟩ **?** ⟨variable⟩

instance = ⟨name⟩ **(** $\{_0$ **,** ⟨actual⟩ $\}$ **)**

interface = **interface (** $\{_0$ **,** ⟨declaration⟩ $\}$ **)**
| **interface (** $\{_0$ **,** ⟨declaration⟩ $\}$ **)** **@** ⟨absolute-reference⟩

literal = **#** ⟨hexdecimal-integer⟩
| **false**
| **true**
| ⟨byte⟩
| ⟨decimal-integer⟩

local-chanend = ⟨element⟩ **@** ⟨absolute-reference⟩

loop = **while** ⟨expression⟩ **do** ⟨command⟩

operand = **(** ⟨expression⟩ **)**
| ⟨element⟩
| ⟨literal⟩

output = ⟨chanend⟩ **!** ⟨expression⟩

parallel = $\{_0$ **&** ⟨parallel-component⟩ $\}$

parallel-component = ⟨process-label⟩ ⟨process⟩
| ⟨process⟩

primitive-command = ⟨assingment⟩
| ⟨connect⟩
| ⟨input⟩
| ⟨output⟩
| ⟨skip⟩
| ⟨stop⟩

primitive-type = **var**
| ⟨call-type⟩
| ⟨chanend-type⟩
| ⟨process-type⟩

| ⟨server-type⟩

procedure = **process** ⟨name⟩ **(** {$_0$ **,** ⟨formal⟩ } **)** **is** ⟨process⟩
| **server** ⟨name⟩ **(** {$_0$ **,** ⟨formal⟩ } **)** **inherits** ⟨hiding-declaration⟩
| **server** ⟨name⟩ **(** {$_0$ **,** ⟨formal⟩ } **)** **is** ⟨server⟩

process = **on** ⟨expression⟩ **do** ⟨process⟩
| **par** ⟨replicator⟩ ⟨process⟩
| ⟨command⟩
| ⟨instance⟩
| ⟨interface⟩ **:** ⟨command⟩

process-label = ⟨name⟩ **is**

process-type = **process** ⟨interface⟩
| **process** ⟨name⟩

program = ⟨program-specification⟩ **:** ⟨program⟩
| ⟨sequence⟩

program-specification = ⟨definition⟩
| ⟨specification⟩

remote-chanend = **@** ⟨absolute-reference⟩ **.** ⟨expression⟩

replicator = **[** {$_1$ **,** ⟨index-range⟩ } **]**

sequence = {$_0$ **;** ⟨command⟩ }

server = ⟨instance⟩
| ⟨interface⟩ **:** ⟨server-specification⟩
| ⟨server-array⟩

server-array = **[** {$_1$ **,** ⟨server⟩ } **]**

server-call = ⟨element⟩ **(** {$_0$ **,** ⟨actual⟩ } **)**
| ⟨local-chanend⟩ **!** ⟨element⟩ **(** {$_0$ **,** ⟨actual⟩ } **)**

server-declaration = ⟨name⟩ **is** **[** ⟨expression⟩ **]** ⟨server⟩
| ⟨name⟩ **is** ⟨replicator⟩ ⟨server⟩
| ⟨name⟩ **is** ⟨server⟩

server-specification = **{** {$_1$ **:** ⟨declaration⟩ } **}**
| ⟨declaration⟩

server-type = **server** ⟨interface⟩
| **server** ⟨name⟩

simultaneous-declaration = {$_0$ **&** ⟨declaration⟩ }

simultaneous-definition = $\{_0$ **`&`** ⟨definition⟩ $\}$

skip = **`skip`**

specification = ⟨abbreviation⟩
| ⟨declaration⟩

specifier = ⟨primitive-type⟩
| ⟨specifier⟩ **`[ ]`**
| ⟨specifier⟩ **`[`** ⟨expression⟩ **`]`**

stop = **`stop`**

structured-command = **`{`** ⟨parallel⟩ **`}`**
| **`{`** ⟨sequence⟩ **`}`**
| ⟨alternation⟩
| ⟨conditional⟩
| ⟨loop⟩

type = ⟨array-type⟩
| ⟨primitive-type⟩

valof = **`valof`** ⟨process⟩ **`result`** ⟨expression⟩
| ⟨specification⟩ **`:`** ⟨valof⟩

variable = ⟨element⟩

A.3. Operators

A ⟨binary-operator⟩ $\oplus$ takes two operands a and b and produces a value $a \oplus b$. A binary *arithmetic operators* takes signed integer operands and produces a signed integer result.

Symbol	Meaning	Definition
`+`	sum	$a + b$
`-`	difference	$a - b$
`*`	produce	$a \times b$
`/`	quotient	a/b
`rem`	remainder	$a \mod b$

A binary *relational operator* takes two signed integer values and produces the value `true` or `false`.

Symbol	Meaning	Definition
`=`	equality	$a = b$
`~=`	inequality	$a \neq b$
`<`	less than	$a < b$
`<=`	less than or equal	$a \leq b$
`>`	greater than	$a > b$
`>=`	greater than or equal	$a \geq b$

A binary *logical operator* takes two operands and produces a bitwise result $b_i = a_i \oplus b_i$ for $0 \leq i < n$.

Symbol	Meaning	Definition
`or`	bitwise or	b `or` $0 = b$, b `or` $1 = 1$
`and`	bitwise and	b `and` $0 = 0$, b `and` $1 = b$
`xor`	bitwise exclusive or	b `xor` $0 = b$, b `xor` $1 = 1$, 1 `xor` $1 = 0$,
`<<`	left bitwise shift	$b_i = b_{i+1}$, $b_0 = 0$
`>>`	right bitwise shift	$b_i = b_{i-1}$, $b_n - 1 = 0$

A ⟨unary-operator⟩ $\oplus$ takes a single operand a and produces the value $\oplus a$.

Symbol	Meaning	Definition
`-`	negation	$0 - a$
`~`	bitwise not	~$0 = 1$, ~$1 = 0$

A.4. Representation of values

Signed values are represented as a two's complement bit-pattern. With a word size of n, values in the range $-2^{n-1} \leq n < 2^{n-1}$ can be represented.

The literal value `true` represents an all-1 bit pattern (the value -1) and the literal value `false` represents an all-0 bit pattern (the value 0). The effect of this is consistent with the logical not operation: `not false` = `true` and `not true` = `false`.

A.5. Character set

A ⟨character⟩ can be any alphabetical character

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

or any special character

```
_ + - = , .  :  ; ?  { } [ ] ( ) # & !  * @ | " '
```

A ⟨digit⟩ can be any numeric character

```
0 1 2 3 4 5 6 7 8 9
```

A ⟨hex-digit⟩ can be any numeric character or

```
A B C D E F a b c d e f
```

A ⟨name⟩ consists of a sequence of alphanumeric characters and underscores.

A.6. Comments

A comment is introduced with a '`%`' symbol and continues until the end of the line.

A.7. Keywords

The following are keywords in sire and cannot be used for names.

`accept`	`function`	`server`
`alt`	`if`	`skip`
`call`	`inherits`	`step`
`chanend`	`initial`	`stop`
`connect`	`interface`	`then`
`do`	`is`	`to`
`else`	`on`	`true`
`false`	`par`	`val`
`final`	`process`	`valof`
`for`	`result`	`var`
`from`	`seq`	`while`

Appendix B.

An overview of the XS1 architecture

This appendix gives a brief overview of the XS1 architecture as a basis for the description of the run-time functionality and code generation given in Chapter 8. It presents a simplified subset of the instruction set, omitting details irrelevant to this work; for full details please refer to its definition [May09].

B.1. Overview

A system consists of a collection of *processor-memory tiles* connected by a network (see Figure 8.3).

A processor has access to a memory, *mem*, and contains a number of *physical resources*:

- A set of *threads*, each with its own set of registers:
  - *r*0 to *r*11, general purpose registers;
  - *pc*, the *program counter*;
  - *sp*, the *stack pointer*;
  - *dp*, the *data pointer*;
  - *cp*, the *constant pool*;
  - *lr*, the *link register*;
  - *sr*, the *status register*;
  - *ed*, the *event data* register.

  All threads have symmetric access to the tile memory.
- A set of *channel ends* for communication, where each channel end represents the resources associated with the physical end point of a communication channel, including:
  - *d*, a *destination channel end* register that specifies where messages are sent;
  - *e*, an *event vector* register;
  - *ev*, an *environment vector* register;

  where *e* and *ev* are used to service events and interrupts.
- A set of *locks* to perform mutual exclusion.
- A *scheduler* that dynamically selects which thread to execute. Threads are not runnable when they are waiting either to synchronise with another thread, to input on a channel, to output on a channel because there is no room to buffer the data, or for an *event*.

B.1.1. Simplifications

The following simplifications are used in the presentation of the instruction subset.

1. *Variants.* Some instructions represent a number of variants of the same operations, such as forward and backward references and memory accesses using the constant pool, data pool or stack pointer as a base. These are all marked with an asterisk.
2. *Exceptions.* Unexpected instructions, invalid system state, invalid use of resources or invalid memory accesses will all cause the processor enters an exception state. However, the details of this and the exception types are not presented since exception handling is not discussed.

B.1.2. Notation

The following notation is used:

- '$x[i]$' denotes a memory address at byte address x with word offset i;
- '$x \leftarrow y$' denotes the value y being written to the storage location x, which could be a register or memory location;
- '$P; Q$' denotes a sequence in which P is performed and then Q;
- 'Bpw' is the number of bytes per word;
- 'bpw' is the number of bits per word.

B.2. Memory access

The memory is accessed with a base and an offset.

`LDAW`* d, b, i	$d \leftarrow b + i \times \text{Bpw}$	load the word address at base b with word offset i
`LDW`* d, b, i	$d \leftarrow \text{mem}[b + i \times \text{Bpw}]$	load word at address b with word offset i into d
`STW`* s, b, i	$\text{mem}[b + i \times \text{Bpw}] \leftarrow s$	store s address b with word offset i

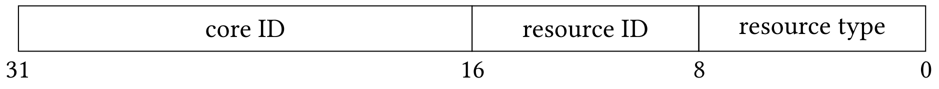
B.3. Branching and procedure calls

The following instructions support branches and procedures.

`B` s	*pc* $\leftarrow s$	branch to address s
`BL`* s	*lr* $\leftarrow$ *pc*; *pc* $\leftarrow s$	link then branch to s
`BLT`* s, u	*lr* $\leftarrow$ *pc*; *pc* $\leftarrow s[u]$	link then branch via table
`ENTSP` n	*sp*[0] $\leftarrow$ *lr*; *sp* $\leftarrow$ *sp*-$n \times$ Bpw	save *lr* then extend the stack by n words
`EXTSP` n	*sp* $\leftarrow$ *sp*+$n \times$ Bpw	extend the stack by n words
`RETSP` n	*sp* $\leftarrow$ *sp*+$n \times$ Bpw; *lr* $\leftarrow$ *sp*[0]	contract the stack by n words then branch to saved *lr*

B.4. Resources

A resource can be a thread, synchroniser, channel end or lock. Resource types are specified with an immediate value. A *resource identifier* is a 32-bit word in which the top 16 bits are the *core identifier* (0 for a synchroniser), the next 8 bits are the *resource identifier* and the bottom 8 bits are the *resource type.*



Resource type values are denoted by the constants `THREAD`, `SYNC`, `CHANEND`, and `LOCK`.

For channel end IDs, this allows them to be uniquely identified within a system and used as an end point by any other core, or indeed a thread on the same core.

`GETR` d, u	allocate a resource of type u and set d to its ID, otherwise set d to 0
`FREER` r	deallocate a resource with ID r

`GETR` allocates resources from a pool and returns the next available resource with the lowest ID value, otherwise it returns 0.

B.5. Communication

Channel ends are allocated and deallocated using `GETR` and `FREER`. Each channel end has a destination register *d* that is set to the ID of the receiving channel end. This must be set by a thread before data is sent, but it is not necessary for data to be received. This provides the ability for a number of threads to send messages to a single channel end.

`SETD` r, s	set the destination of channel end r to the channel end ID s
`OUT` r, s	output word s to the channel end r
`OUTCT` r, s	output a control token s to the channel end r
`IN` d, r	input a word from r to d if r is a channel end
`CHKCT` r, s	if next token is the control token s discard it, otherwise raise an exception

The interconnect is *wormhole switched* such that when a message is output on a channel, it establishes a route through the network (if it is not already open) so that subsequent messages can travel without any setup overheads. A special 8-bit *control token* `END` is used to close a route, allowing transfers of data to be packetised or streamed. Other control tokens can be used to control network resources and encode communication protocols.

However, this flexibility requires routes to be opened and closed in such a way that they cannot cause deadlock by holding network resources. One way to do this is with *synchronised communication* to ensure receivers are always ready to receive a message before it is send. A synchronisation between two parties, a and b, can be implemented with the sequence

thread a		thread b
`OUTCT` c, `END`	$\rightarrow$	`CHKCT` c, `END`
`CHKCT` c, `END`	$\leftarrow$	`OUTCT` c, `END`

Then, this synchronisation can be used to ensure the destination received a message before anything else is sent:

thread a		thread b
`OUT` c, v	$\rightarrow$	`IN` c, v
`OUTCT` c, `END`	$\rightarrow$	`CHKCT` c, `END`
`CHKCT` c, `END`	$\leftarrow$	`OUTCT` c, `END`

And, if the sender does not know whether the destination is ready to receive a message, a synchronisation can be also be applied beforehand:

thread a		thread b
`OUTCT` c, `END`	$\rightarrow$	`CHKCT` c, `END`
`CHKCT` c, `END`	$\leftarrow$	`OUTCT` c, `END`
`OUT` c, v	$\rightarrow$	`IN` c, v
`OUTCT` c, `END`	$\rightarrow$	`CHKCT` c, `END`
`CHKCT` c, `END`	$\leftarrow$	`OUTCT` c, `END`

B.6. Events and interrupts

Channel ends can be configured to trigger *events* or *interrupts* on receipt of a message. An event causes control to be transferred from a waiting thread to the channel's event vector and for an *event data* register to be set to the channel's *environment vector*, which is used to store specific data for the handler. Interrupts, in contrast, are unexpected events that cause the same transfer of control and access to the environment vector, but additionally, the *pc*, *sr* and *ed* registers are saved.

SETSR u	*sr*←*sr*$\vee u$	set the status register
GETSR d ,u	*sr*←*sr*$\wedge u$	set the status register
SETV r, s		set event vector to s on channel end r
SETEV r, s		set event environment to s on channel end r
SETC r, s		set the control register to s of channel end r
EE r		enable events on channel end r
ED r		disable events on channel end r
WAITE		wait for an event
CLRE		disable the generation of any events by the current thread

Status register bits corresponding to events and interrupts are set with the constants EVENT_ENABLE and INTR_ENABLE, and the control register bits are set with the constants MODE_EVENT and MODE_INTR.

B.7. Threading

The following instructions support the allocation, initialisation, synchronisation and termination of local threads.

GETID d	get the resource ID of the executing thread
GETST d, r	get a thread, bind it to a synchroniser r and set d to the thread ID.
TSETR* d, s, t	set register d of thread t to s
MSYNC r	master synchronise on synchroniser r
SSYNC r	slave synchronise on synchroniser r
MJOIN r	master synchronise on synchroniser r and free slave threads
FREET	stop the executing asynchronous thread and free it

Collections of threads can be *synchronised* or they can operate *asynchronously*. A *master* thread can allocate one or more synchronised *slave* threads with GETST, binding them to a synchroniser (itself allocated and deallocated with GETR and FREER). This allows the master and slaves to barrier synchronise with MSYNC and SSYNC. When the master executes an MJOIN, the slaves are terminated and deallocated. The master can use TSETR to perform initialisation by directly setting slave registers.

Asynchronous threads are allocated and deallocated with GETR and FREER and are responsible for their own termination with FREET.

B.8. Locks

Locks are allocated and deallocated using GETR and FREER, and are *claimed* and *released* using input and output instructions.

OUT r	release the lock r
IN r	claim the lock r