XMP-64 performance measurements

James Hanlon[*]

22 Feburary 2010

1 Introduction

This document presents performance measurements for the XMP-64 device. For each measurement, it describes the method taken and the results obtained. Section 2 starts by introducing the hypercube interconnection topology, Section 3 describes a technique for global clock synchronisation and gives timings for barrier synchronisation, and finally Section 4 describes performance and behaviour with synthetic traffic patterns.

2 Topology

The XMP-64 connects together 64 XCore processors in 16 XS1-G4 devices. These are arranged as a 4-dimensional *hypercube* using 5b XMOS links on a single PCB. A hypercube is a generalisation of a regular cube structure into an arbitrary number of dimensions. A d-dimensional hypercube is a special case of a k-ary n-cube (torus network) when $k = 2$, and has $N = 2^d$ nodes and $d2^{d-1}$ edges. Each node in the network can be labeled with a d-bit binary identifier, and an edge exists between two nodes x and y if and only if their identifiers differ by exactly one bit, i.e. for some integer $k \geq 0$

$$x \oplus y = 2^k.$$

Hence, each node has $d = \log N$ edges. An edge is called a *dimension* e *edge* if it links two nodes whose identifiers differ in the e^{th} bit position [2].

Intuitively, a 4-dimensional hypercube can be constructed by joining two cube structures (each with 8 nodes), by adding edges between corresponding vertexes. Figure 1 illustrates this. Incidentally, a 4-ary 2-cube is equivalent to a 4-dimensional hypercube, and this *flat* structure is used to package the hypercube network between the 16 chips in the XMP-64. As each chip contains 4 cores, it is convenient to view the network as a hypercube with 6 dimensions.

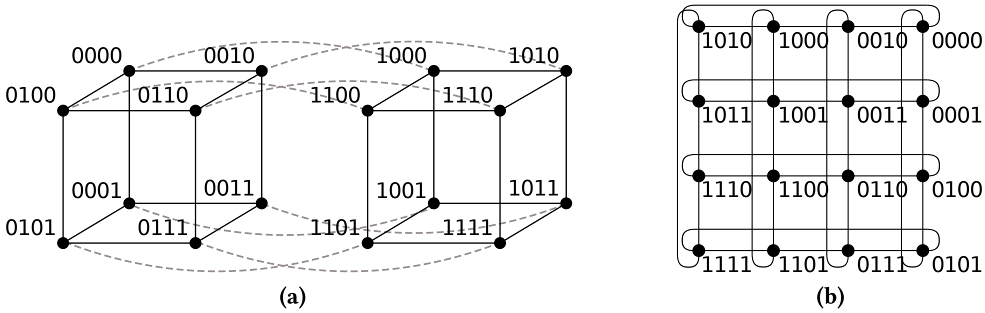

Figure 1: Representations of a 4-dimensional hypercube. (a) shows an intuitive construction and (b) shows an equivalent 4-ary 2-cube, or torus network, which is used to package the hypercube on the XMP-64 board.

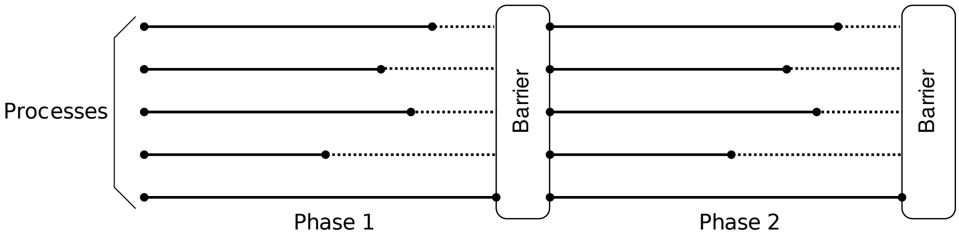

Figure 2: Operation of a barrier. Some processes may complete a phase more quickly than others, but the barrier ensures that all processes enter the next phase synchronously.

3 Synchronisation

Some programming techniques for parallel computers rely on efficient synchronisation between some or all of the processes, so that they may operate in unison. Synchronisation may be needed to detect termination, or to ensure that all running processes have completed updates to a global state before proceeding to the next stage of a computation.

Barrier synchronisation is a key operation in many parallel algorithms. It is used to ensure that a set of processes enter a new phase of computation at the same time. Any global communication such as a reduction (an operation such as a sum or multiply performed over all elements of a distributed data set) or scatter (to distribute data from one process to many processes) may imply the use of a barrier. Figure 2 illustrates the operation of a barrier. Processes may enter a barrier at any time, but may leave only once all other processes have entered.

Clock synchronisation is another form of synchronisation, necessary when each node in a distributed system has access to its own clock, but no guarantees can be made of the agreement with the clocks kept by other nodes. Synchronisation can be performed globally so that there is a consensus on a single time in the network. Clock synchronisation is key to making accurate measurements for message latencies, barrier synchronisations and traffic patterns.

3.1 Barrier synchronisation

With a hypercube network, it is possible to perform a barrier synchronisation in $O(\log N)$ communication steps, where N is the number of nodes in the network. In other network topologies such as meshes or irregular structures, barrier synchronisation is typically implemented using tree structures which incur a far higher cost.

The barrier synchronisation scheme for a hypercube works by synchronising with neighbouring nodes, in each dimension in turn. To begin, each node exchanges a single message with its neighbouring node. The first neighbour of node i is $i \oplus 1$, the second is $i \oplus 2$ and $i \oplus 2^d$ for dimension d. In the first exchange, all pairs of nodes connected in the first dimension become synchronised with each other and we have $N/2$ synchronised pairs. In the second exchange, nodes connected in second dimension become synchronised with those in the first, producing $N/4$ synchronised groups. After d iterations, all nodes become synchronised in a single group. In this scheme, no node can leave the barrier before all nodes have entered it. Algorithm 1 gives pseudo-code that each node executes to perform a barrier synchronisation.

Algorithm 1 Barrier synchronisation executed by each node m in the network.

```
for $i = 1$ to $d$ do
    Neighbour node $n = m \oplus 2^i$
    Send message to $n$
    Receive message from $n$
end for
```

This approach of iteratively exchanging messages in each dimension is a general and useful communication pattern that can be applied to many other problems. These include finding minimum and maximum values over the set of nodes and to calculate the average of the values held by each node. In particular though, global clock synchronisation can also be achieved in this way.

3.2 Global clock synchronisation

The aim of clock synchronisation is for each node to learn an offset value to some reference clock in the network. This could be the average clock or that of a specific node.

Synchronisation of clocks to a specific node for a hypercube works in the same way as the barrier. Let c_n denote the clock of node n. Initially, all pairs of nodes connected in the first dimension exchange messages to determine the offsets between their clocks. If synchronisation is based on c_0, then a adjustments are applied in each exchange to the node with the largest identifier value. If it is based on c_{N-1}, then adjustments are applied to the lowest.

Using a similar inductive argument as the one used in Section 3.1, we can show this will approach will result in global synchronisation. After the first exchange, each pair of nodes are synchronised and conceptually the network contains $N/2$ different clocks. In the second phase, nodes exchange in the second dimension, but *include* the offset they learned in the first, leaving $N/4$ different clocks. This continues until all nodes have learned their offset from the reference clock.

Pseudo-code for this process is given in algorithm Algorithm 2. The functions $\mathrm{clkSyncMaster}()$ and $\mathrm{clkSyncSlave}()$ perform communication between neighbours in a particular dimension to determine the difference Δ, between the two clocks such that

$$\Delta = c_{n_u} - c_{n_v}.$$

Algorithm 2 Clock synchronisation pseudo-code for node m.

```
$offset \leftarrow 0$
for $i = 1$ to $d$ do
    Neighbour node $n = n \oplus 2^i$
    if $m > n$ then
        $\Delta \leftarrow \mathrm{clkSyncMaster}()$
        $offset \leftarrow offset + \Delta$
    else
        $\mathrm{clkSyncSlave}()$
    end if
end for
```

3.2.1 Determining the clock offset between two cores

Accurately determining the value of Δ is key to the final synchronisation between nodes. This can be performed with two *ping-pong* exchanges between the *master* and *slave* processes. The master learns from this two time values t_0 and t_2 recorded by the slave, and t_1, recorded itself in the middle. The exchange is initialised by the slave node which sends a message to the master. When the master receives this, it replies. On receiving this, the slave measures its time (t_0) and send this back to the master. When the master receives it, it measures its own time (t_1), then pings the slave again for another clock measurement (t_2) which is measured and sent back in the same way. Figure 3 illustrates this exchange.

Using these values of t_0, t_1 and t_2, the following equations can then be setup, where h is a single hop time and ϵ is an error term.

$$\begin{aligned} t_1 - t_0 &= \Delta + h + \epsilon & (1) \\ t_2 - t_1 &= -\Delta + h + \epsilon & (2) \end{aligned}$$

Subtracting 2 from 1 we can then obtain the value of Δ:

$$\Delta = t_1 - t_0/2 - t_2/2.$$

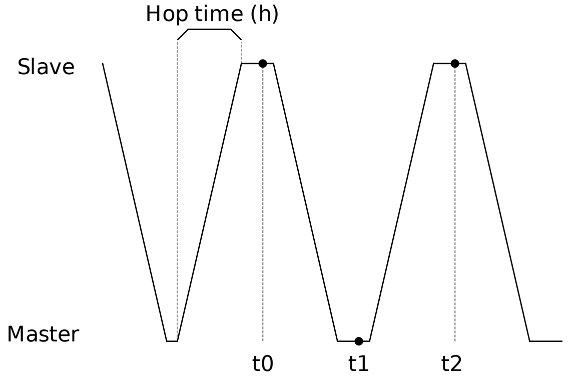

Figure 3: A diagram illustrating the exchanges made between the master and slave nodes to obtain the values t_0, t_1 and t_2.

To reduce to a minimum any error in measurement, it is important for these measurement operations to minimise the number of instructions and ensure the exchanges are symmetric. As a result, the functions $\text{clkSyncMaster}()$ and $\text{clkSyncSlave}()$ were implemented directly in assembly.

3.2.2 Reducing Error in Δ

In practice, the true value of Δ cannot always be learned, and instead the calculation may yield $\Delta + \epsilon$, where ϵ is some small error value. This could be caused by non-determinism at the hardware level. We know that the calculation of Δ in a given dimension d' should be the same for all of the nodes that have already synchronised in the previous dimensions, of which there will be $2^{d'-1}$.

Using this invariant, we can reduce the effect of this error as we propagate an offset through the cube by averaging over previous dimensions. Each node m computes its Δ_m offset as the average over the other Δs calculated in previous dimensions $0, \ldots, d' - 1$:

$$\Delta_m(d') = \frac{1}{2^{d'-1}} \sum_{n \in A(n,d')} \Delta_n$$

where

$$A(m, d) = \{n \mid n = m \oplus 2^i \text{ for } 0 \leq i < d\}$$

is the set of neighbouring nodes of m in dimensions $0, \ldots, d-1$. The calculation of the average at each node can be completed in $\log(2^{d-1}) = d - 1$ steps using the same dimension-ordered exchange procedure.

Core-to-core journey	Time (ns)
On-chip	70
Off-chip (1 hop)	200
Off-chip (2 hops)	290
Off-chip (3 hops)	390
Off-chip (4 hops)	480

Table 1: Timings of core-to-core journeys, both on and off-chip. Note 4 is the diameter of the hypercube; the maximum distance between any two nodes.

3.3 Timing a barrier synchronisation

3.3.1 Estimated time

A simple estimate of the time for a barrier synchronisation to complete can be made by considering the single-hop times between cores. Since we can view the XMP-64 network as a 6-dimensional hypercube, where the first two dimensions are contained in-chip, the single hop time in-chip h_{in} and off-chip h_{off} form this estimate. The time to run a barrier synchronisation (the operation given in Algorithm 1) is then $2h_{in} + 4h_{off}$. These times are simple to measure and are presented in table 1. Using them, an estimate of 940ns for the barrier to complete can be made.

3.3.2 Measured time

To make a precise measurement of the time taken for a barrier to complete, where all nodes minimise their time in the barrier, i.e. that they enter at precisely the same point in time (an assumption made by the estimate), it is necessary to use globally synchronise clocks.

If nodes enter the barrier unsynchronised, some will enter before others, completing exchanges in as many dimensions as they can but not completing until all have entered. For those nodes entering late, they will complete much faster then normal as messages will be waiting for them in one or more dimensions. In one extreme, nodes n_1 to n_{63} enter the barrier early, followed much later by n_0. In this case, it takes n_0 just 280ns to complete the barrier synchronisation.

For nodes to enter the barrier simultaneously, they must synchronise their clocks against, for instance, node n_0 to obtain an offset to c_{n_0} and enter at a time in the future specified by n_0 adjusted by the offset to c_{n_0}. If the synchronisation is perfect, then each node should spend exactly the same amount of time in the barrier.

Using the above method, the elapsed time through the barrier was recorded for each node. The results varied by a range of around 150ns each run, with minimum and maximum times of approximately 930ns and 1100ns respectively, but with a consistent average of 990ns, which is in-line with the estimate made by considering single hop times.

Although the measurement error in the Δ term was reduced by averaging over synchronised nodes, it still effects the synchronisation, evident in the resulting times through the barrier. To ensure this error was not systematic in the program code, the precise clock offsets were inspected by analysing signal output from pins on the board. This revealed that offsets after synchronisation between nodes n_1 to n_{63} and n_0 varied between runs and hence could not be caused by some bias in the measurement for example.

4 Traffic patterns

In order to evaluate the performance of interconnection networks, *synthetic workloads* can be generated. These are a simplification of real execution workloads, but they capture the important *spatial* and *temporal* elements of them. With the XMP-64, we are interested in the temporal characteristics of different traffic patterns and the congestion that they induce over the network.

4.1 Permutation patterns

Synthetic traffic patterns are commonly considered as a permutation π, which provides a one-to-one mapping from source addresses s to destination addresses d:

$$d = \pi(s).$$

Because permutation traffic concentrates load on individual source-destination pairs, they provide good stress-testing [1].

Bit permutations calculate each bit of the destination address d_i as a function of one bit of the source address s_i such that

$$d_i = s_{f(i) \oplus g(i)}.$$

The following bit permutations were used to evaluate performance. In all cases, b is the number of bits in the pattern, in the case of hypercube identifiers $b = d$.

- **Shuffle.** A Fast Fourier Transform or sorting algorithm will demonstrate communications characteristic of the shuffle permutation:

  $$d_i = s_{i-1 \mod b}.$$

  Equivalently, the identifier is circularly shifted by 1-bit.

- **Transpose.** Matrix transpose or corner-turn operations induce the transpose permutation:

  $$d_i = s_{i+\frac{b}{2} \mod b}.$$

  This is equivalent to a circular shift of an b-bit identifier by $b/2$. The transpose permutation is a worst case for a hypercube network since it causes all source-destination pairs to be separated by the full diameter of the network, and hence all nodes to be maximally loaded. For the XMP-64, the transpose relates to a circular shift of two, performed on the four most significant bits.

- **Bit-complement.** Each bit is negated:

$$d_i = \bar{s_i}.$$

- **Bit-reverse.** The binary representation is reversed:

$$d_i = s_{b-i-1}.$$

- **Random.** Random permutations were also used to provide an average-case. These differ slightly to random traffic patterns, where each node is equally likely to send to each destination, possibly resulting in many sources sending to a single destination.

4.2 Method

As we are interested in the spatial locality of the traffic permutations, measurements can be taken from a single burst of traffic between all source-destination pairs. If this is performed in unison by all nodes, i.e. they begin sending at the same instance, then maximum congestion will occur.

To do this it is necessary to perform a global clock synchronisation, so that they can synchronise their entry into the permutation and calculate the latencies of messages sent. Measurements are taken over 10,000 runs of the permutation to ensure values are representative of the underlying process.

We will look at two important elements of the traffic patterns: distribution of message latencies and average latencies. To look at the latency distribution, each node records the latency of each message in a set of frequency bins. To determine the bin ranges, the traffic pattern is simulated for a number of runs so that all nodes can share a maximum latency value, from which the bin range is determined. At the end of the experiment, a master node collates the frequency distributions from all other nodes. To determine average latency, again each node records total latency and then calculated average latency on completion, passing values back to the master node for collation into a global average.

For random permutations, each iteration of the experiment is conducted with a new permutation so that the measurements are unbiased towards some particular configuration. This is achieved by each node, each iteration, re-shuffling the permutation, achieving pseudo-randomness using a cyclic redundancy check (CRC) instruction. An initial global seed value is distributed to all nodes so they generate the same sequence of random numbers. According to the permutation, channel end destinations are manually configured during execution.

With regards to the software implementation, each network node consists of two threads; one sender and one receiver. This is necessary for message lengths greater that the buffering

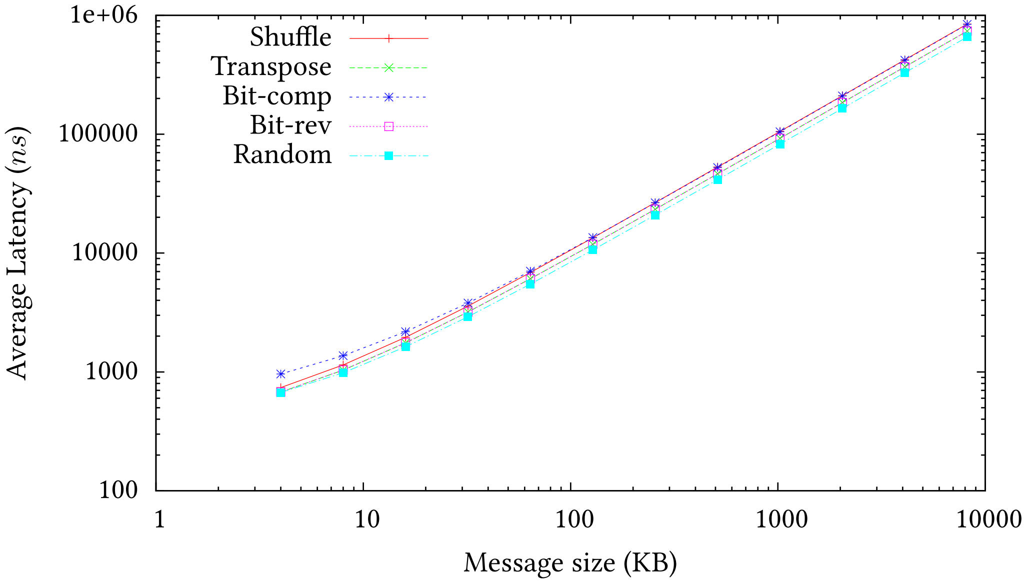

Figure 4: Log-log plot of average latency as a function of message size for a 64 nodes.

between nodes (16 Bytes). As each dimension of the hypercube is connected by four links, traffic congestion will be highest when every link is fully utilised. This can be achieved by running 4 pairs of send and receive processes on each core. Alternatively, the number of available links between each processor can be altered by modifying the XN mapping file.

4.3 Average latency

Figure 4 shows the average latency of messages over all nodes, for varying message lengths. These results were obtained from all 64 nodes, with each core running a single pair of send and receive threads. Processors are connected with a single link in each dimension to maximise congestion. Note that there is very little, or even no penalty for sending short messages.

4.4 Latency Distributions

Figures 5, 6, 7, 8 and 9 show the latency distributions for a message length of 32 bytes, with 64 cores and single wire interconnects.

The latency distribution for the random permutation in Figure 9 clearly shows distributions around each of the 1, 2, 3 and 4 node hops. The distributions are asymmetric because a hop must always take at least some period of time, but a message can be delayed in a network for any amount of time.

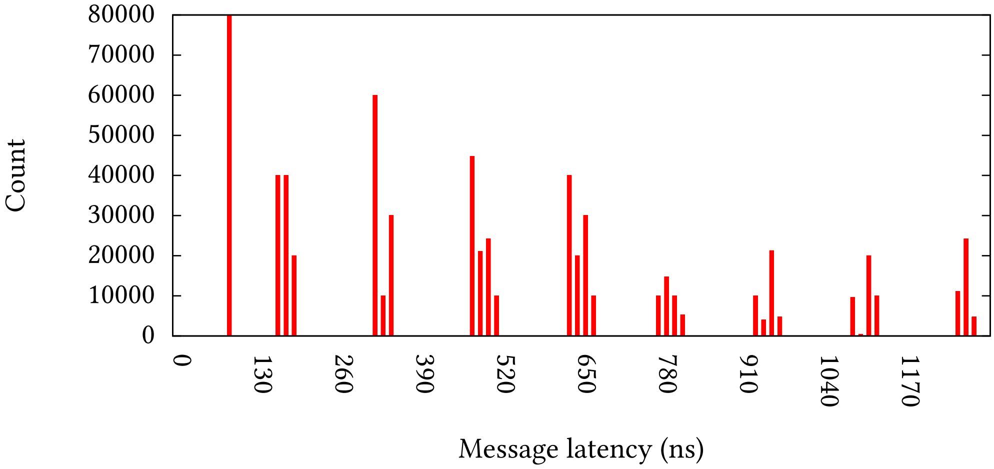

Figure 5: Latency distribution for a shuffle permutation

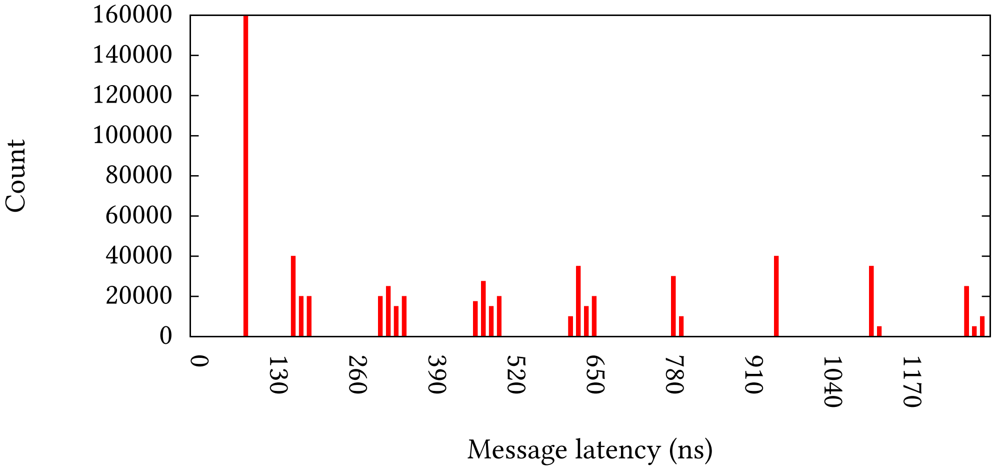

Figure 6: Latency distribution for a transpose permutation

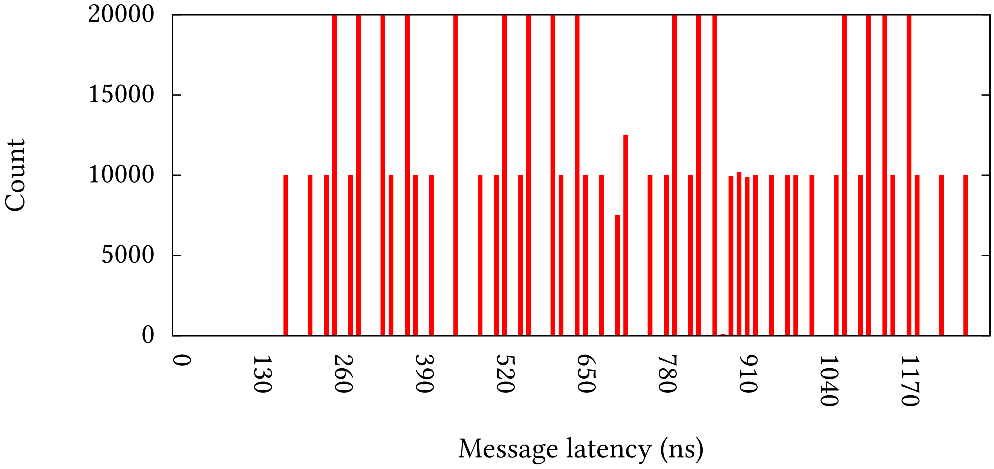

Figure 7: Latency distribution for a bit-complement permutation

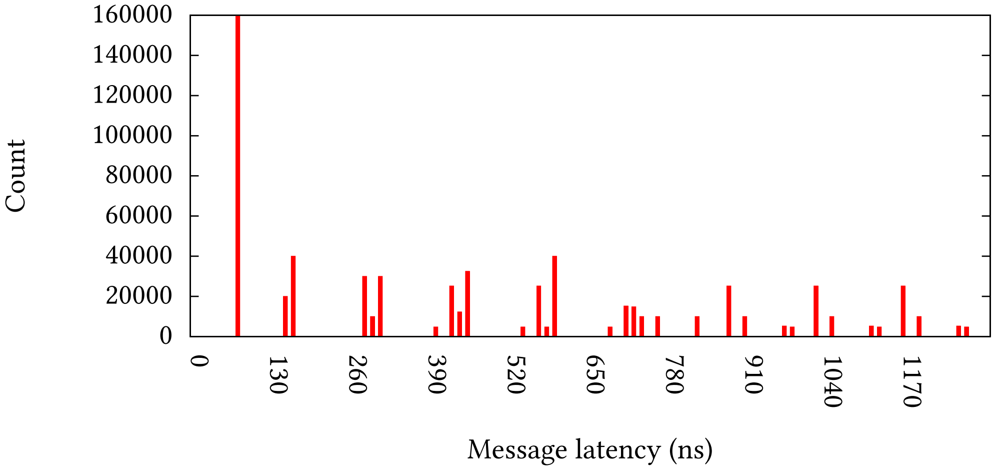

Figure 8: Latency distribution for a bit-reverse permutation

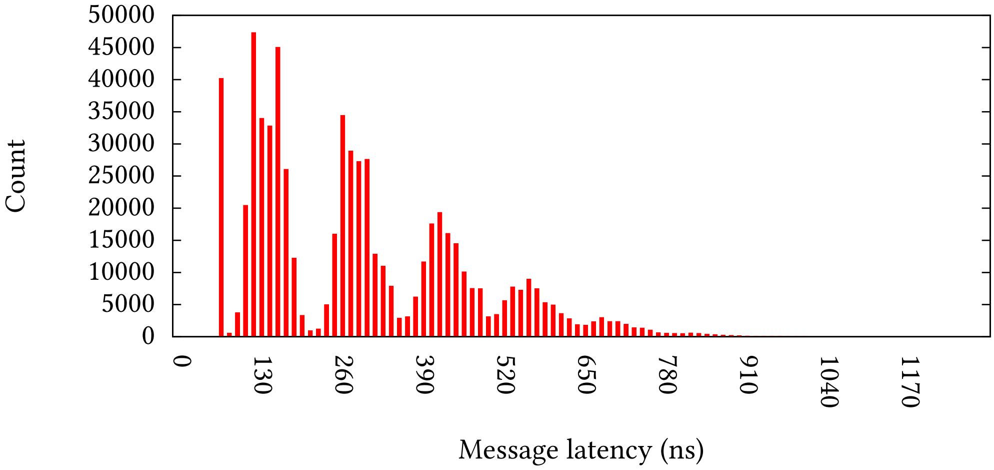

Figure 9: Latency distribution for random permutations

References

[1] W. J Dally, B. Towles *Principles and practices of interconnection networks* Morgan Kauffman Pub, 2004

[2] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures : Arrays, Trees, Hypercubes* Morgan Kauffman Pub, 1992