XK-XMP-64 Performance Measurements

(VERSION 1.1)

2010/03/15

Authors:

JAMIE HANLON

1 Hypercube Topology

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 if their identifiers differ by exactly one bit. Hence, each node has $d = logN$ edges. An edge is called a *dimension e edge* if it links two nodes whose identifiers differ in the eth 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, the hypercube can be viewed as having 64 nodes, in 6 dimensions.

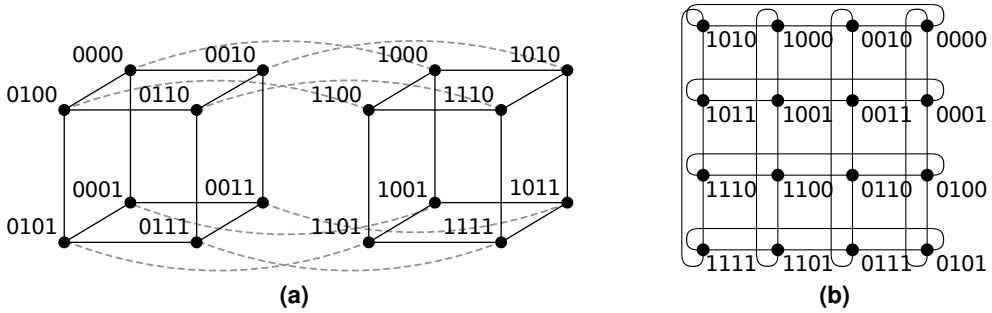

Figure 1 *Representations of a 4-dimensional hypercube. (a) shows an intuitive construction and (b) shows an equivalent 4-ary 2-cube, or torus network, which is used to package the hypercube on the XMP-64 board.*

2 Node Synchronisation

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

Barrier synchronisation is used to ensure that a set of threads/processes enter a new phase of computation at the same time. More generally, any global communication such as a reduction (an operation such as a sum or multiply performed on distributed data) 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.

Clock synchronisation is another form of synchronisation, necessary when each node in a network 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 the

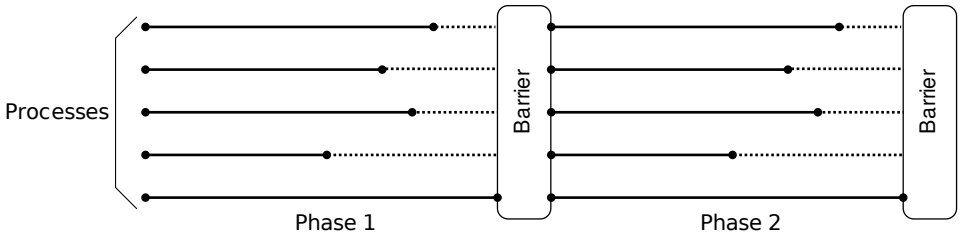

Figure 2 *Operation of a barrier. Some processes may complete a phase more quickly than others, but the barrier ensures that all processes enter the next phase synchronously.*

time in the network. Doing this, for example, would allow the transit times of messages between nodes to be calculated.

2.1 Barrier Synchronisation

The communication channels of the XMP-64 are arranged in a *hypercube* topology, and it is possible to implement a barrier synchronisation in $O(logN)$ 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 in the following way. In dimensional order, each node exchanges a single message with its neighbouring node. For example the first neighbour of node i is $i \oplus 0x1$, the second is $i \oplus 0x2$ and in general for dimension d: $i \oplus (0x1 \ll d)$.

In the first exchange, all pairs of nodes connected in the first dimension become synchronised with each other. In the second exchange, nodes connected in second dimension also become synchronised. After d iterations, all nodes become synchronised with each other. The critical feature of these exchanges is that no node can leave the barrier before all nodes have entered it. Algorithm 1 gives the barrier pseudo-code that each node executes to synchronise.

Algorithm 1 Barrier synchronisation pseudo-code executed by each node in the network.

```
for i = 1 to d do
    Neighbour node n = i ⊕ (0x1 ≪ j)
    Send message to n
    Receive message from n
end for
```

This approach of iteratively exchanging messages in each dimension can be applied to other problems, such as finding minimum and maximum values over the set of nodes, or to calculate the average of the values held by each node. In particular though, clock synchronisation can also be achieved this way.

2.2 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 by iteratively synchronising dimensions 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 synchronising to c_{n_0}, an adjustment is then applied to the node with the largest identifier, and if to node $c_{n_{N-1}}$'s clock, then to the lowest. At this point the two nodes in each pair of nodes are synchronised, and conceptually the network now contains $N/2$ different clocks. In the second phase, nodes exchange in the second dimension, but *include* the offset they learnt in the first, leaving $N/4$ different clocks. This continues until all nodes have learnt their offset from the reference clock.

The pseudo-code for this process is given in algorithm 2. The functions clkSyncMaster() and clkSyncSlave() communicate in order to determine the Δ term between the two clocks such that $\Delta = c_{n_u} - c_{n_v}$.

Algorithm 2 Clock synchronisation pseudo-code for node u.

```
offset ← 0
for i = 1 to d do
  Neighbour node v = i ⊕ (0x1 ≪ j)
  if u > v then
    Δ ← clkSyncMaster()
    offset ← offset + Δ
  else
    clkSyncSlave()
  end if
end for
```

Accurately determining the value of Δ is key to the final synchronisation between nodes. The following describes a way by which this can be done.

The algorithm essentially works with two *ping-pong* exchanges between the master and slave processes from which the master learns 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 sends one back. On receiving this, the slave measures its time (t_0) and send it back to the master. The master receives it and 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, the following equations can then be setup, where h is a single hop time and ϵ is an error term.

$$t_1 - t_0 = \Delta + h + \epsilon \quad (1)$$

$$t_2 - t_1 = -\Delta + h + \epsilon \quad (2)$$

Then, subtracting 2 from 1 we have

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

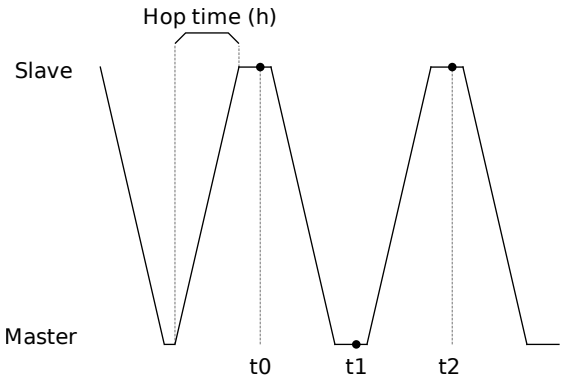

Figure 3 *A diagram illustrating the exchanges made between the master and slave nodes to obtain the values t_0, t_1 and t_2.*

In order to reduce to a minimum any error in measurement by minimising the number of instructions and ensuring the exchanges are symmetric, the functions clkSyncMaster() and clkSyncSlave() were implemented in assembly code.

In practice, the true value of Δ cannot always be learnt, and instead the calculation may yield $\Delta+\epsilon$, where ϵ is some small error. 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}. Hence, to reduce the effect of this error, each node u computes its Δ_u offset as the average over the other Δs calculated in previous dimensions:

$$\Delta_u(d) = \frac{1}{2^{d-1}} \sum_{v \in \text{nbs}_d(v)} \Delta_{n_v}(d)$$

where

$$\text{nbs}_d(v) = \{n \,|\, (n \text{ AND } (1 \ll dd)) = (v \text{ AND } (1 \ll d))\}$$

is the set of neighbouring nodes of v in dimension d, of which will always be of size 2. The calculation of the average at each node can be completed in $d - 1 = log(2^{d-1})$ steps using the same approach of exchanging values in each dimension.

3 Timing a Barrier Synchronisation

As barrier synchronisation is key to the operation of many parallel algorithms, it is useful to know how long it takes. An simple estimate can be made by considering the single-hop times between cores. In the case of the XMP-64, we can view its 64 cores as being arranged in a 6-dimensional hypercube, where the first two dimensions are in the chip. If h_{in} is a single hop time in-chip and

h_{off} is the time for a single hop off-chip, then the time to run a barrier synchronisation should be $2 \times h_{in} + 4 \times h_{off}$. These times are straight-forward to measure and are presented in table 1. Using these values, an estimate of 940ns for the barrier to complete can be made.

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.*

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 above estimate), it is necessary to use the clock synchronisation to achieve this.

If nodes enter the barrier synchronised, 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 example, 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 Traffic Patterns

Synthetic traffic patterns are commonly considered as a permutation π, which provides a one-to-one mapping of source addresses to destination addresses; $d = \pi(s)$. 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 of the XMP-64.

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

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

  Where b is the number of bits in the pattern. 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 n-bit identifier by $n/2$. The transpose permutation is a worst case for a hypercube network as 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 as are interested in the four dimensions of the hypercube, the transpose relates to a circular shift of two, performed on the four most significant bits.

- **Bit Complement.**

  $$d_i = \bar{s}_i.$$

- **Bit Reverse.**

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

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 between all nodes, 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. Technically, 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 between nodes (16 Bytes). As each dimension of the hypercube is connected by 4 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.

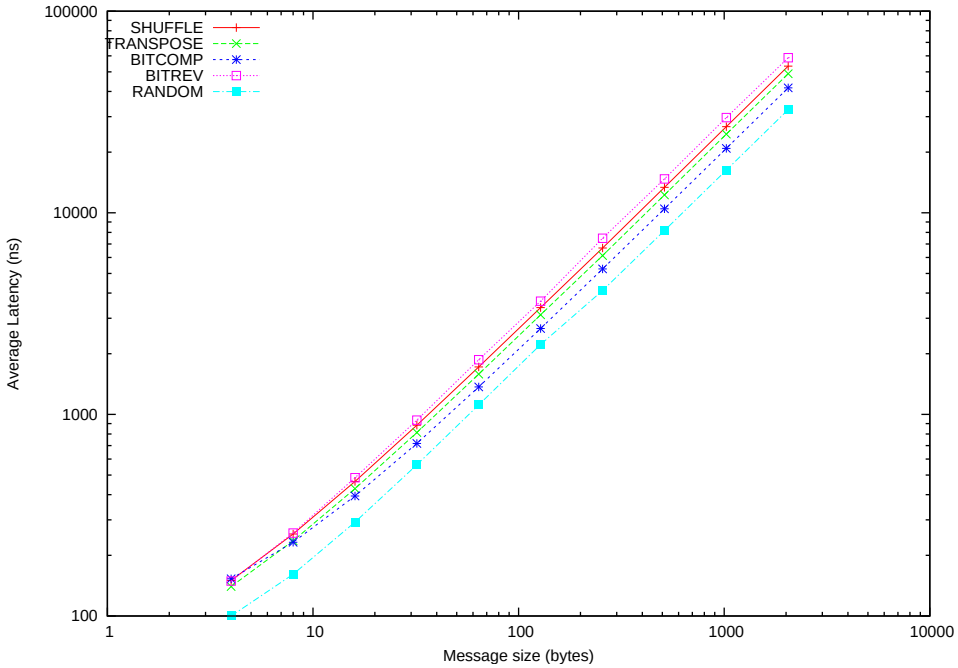

Figure 4 *Log-log plot of average latency as a function of message size for a 64 nodes with single wire interconnects.*

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 asymmetric 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.

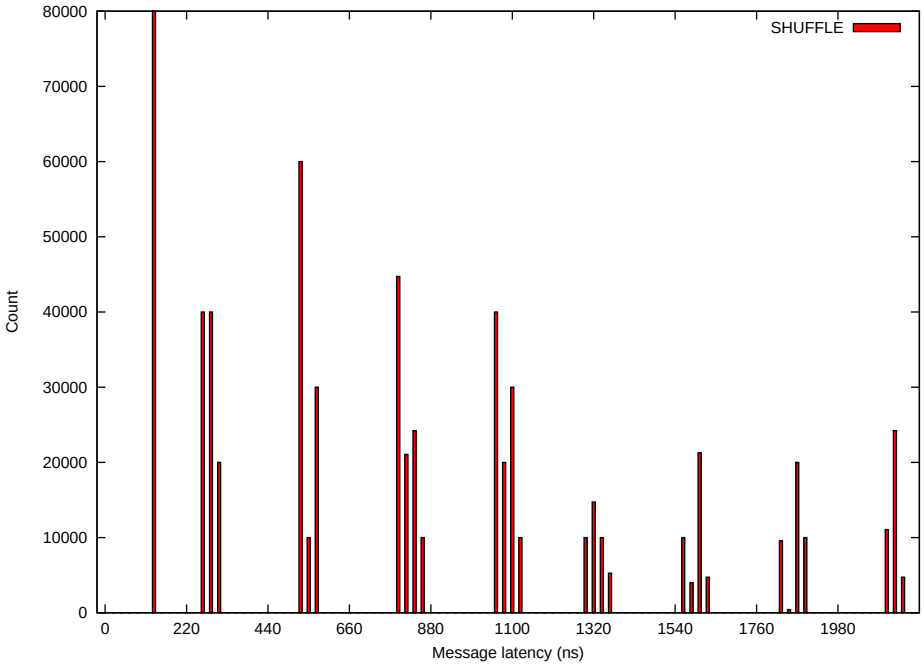

Figure 5 *Shuffle permutation*

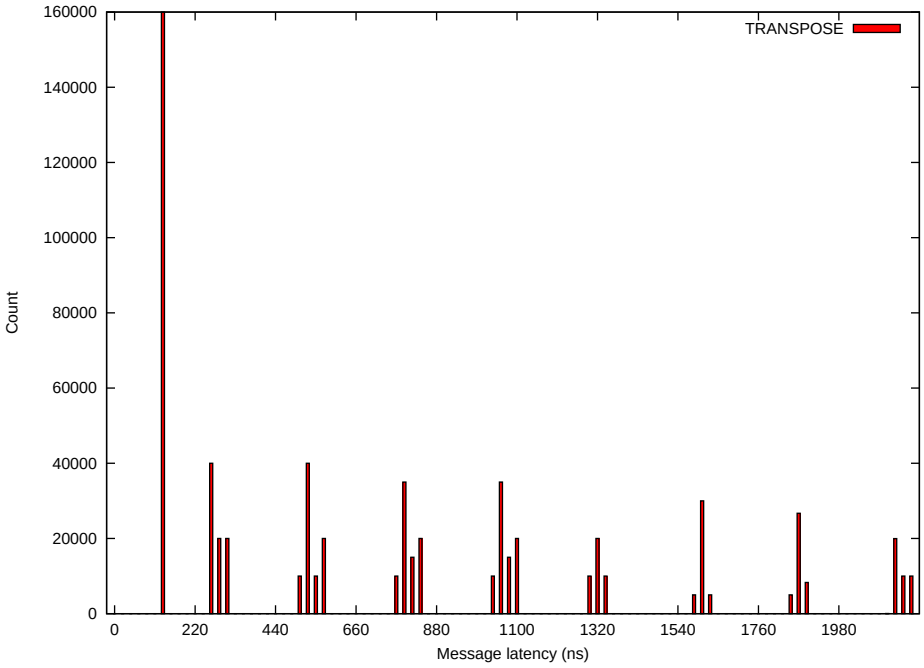

Figure 6 *Transpose permutation*

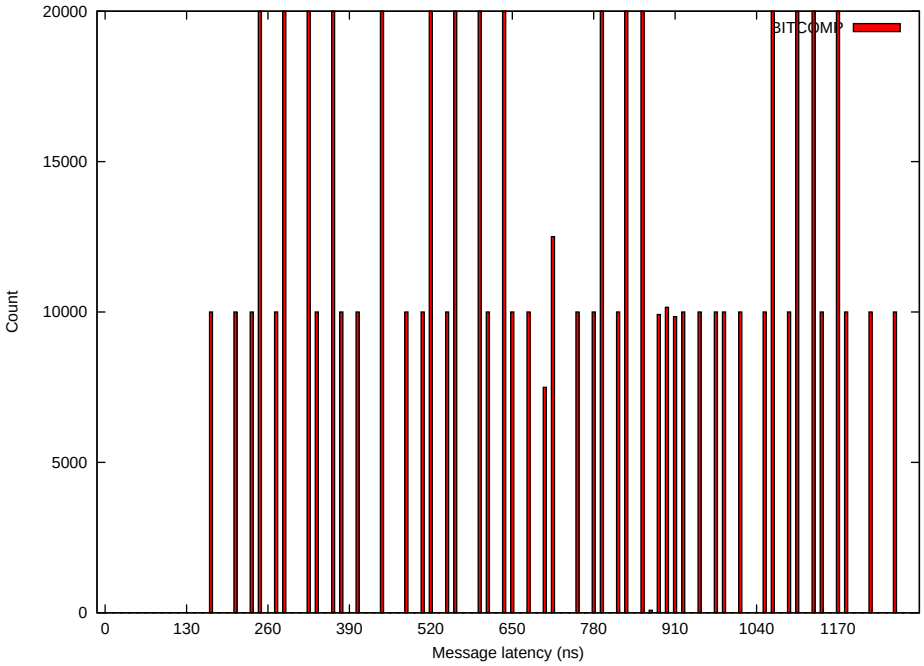

Figure 7 *Bit-complement permutation*

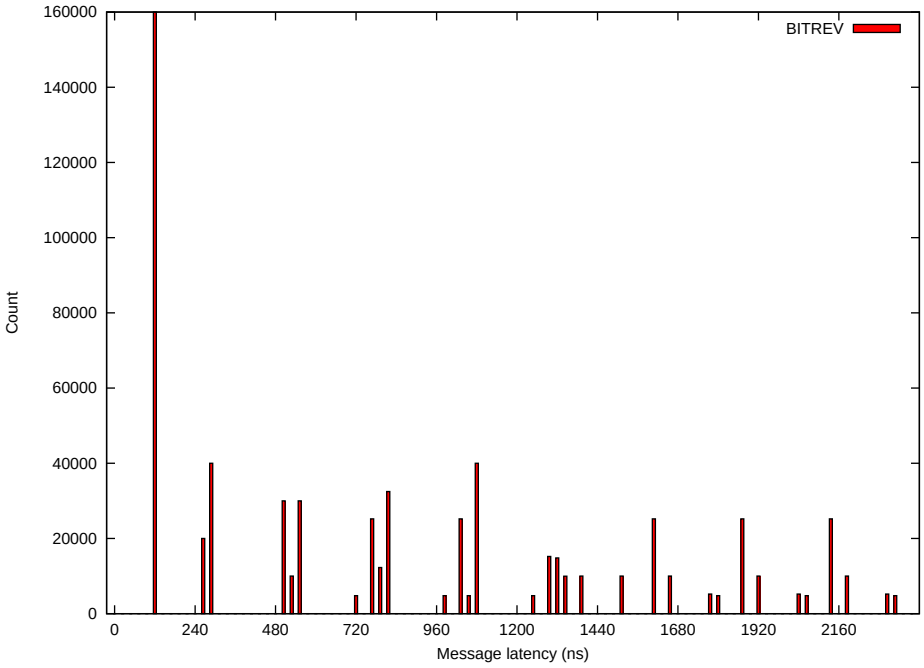

Figure 8 *Bit-reverse permutation*

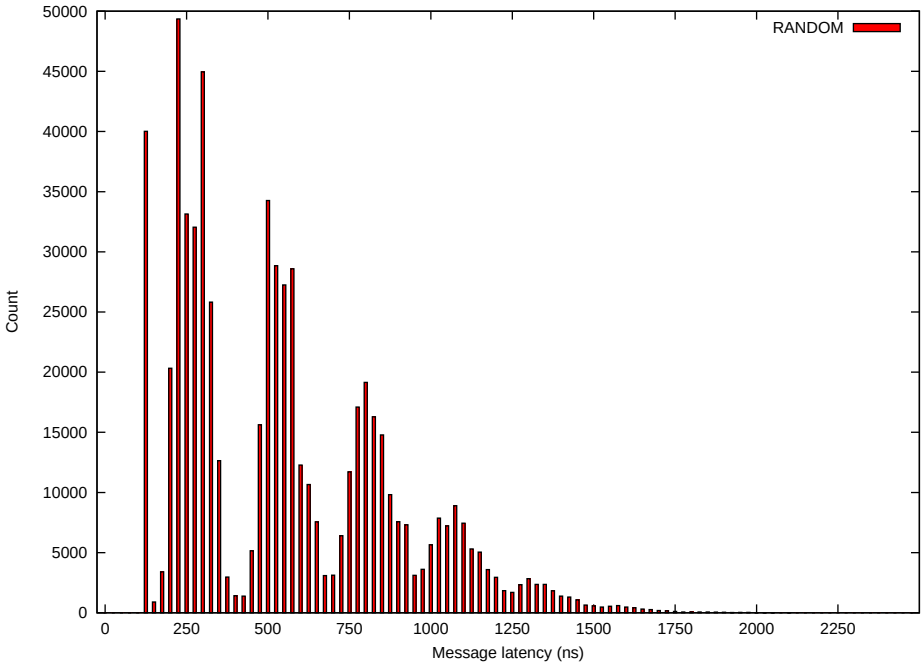

Figure 9 *Random permutation*

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

5 Document History

Date	Release	Comment
2010/02/22	1.0	First release.
2010/03/15	1.1	Latency histograms updated.