SIMULATION OF LARGE SCALE ARCHITECTURES ON HIGH PERFORMANCE COMPUTERS

A Dissertation
Submitted In Partial Fulfilment Of
The Requirements For The Degree Of

MASTER OF SCIENCE

In

NETWORK CENTERED COMPUTING,
HIGH PERFORMANCE COMPUTING AND
COMMUNICATION

in the

FACULTY OF SCIENCE

UNIVERSITY OF READING

by

Ian Jones

22nd October 2010

Dr. Christian Engelmann

Acknowledgements

I would like to extend my thanks to my supervisor, Dr. Christian Engelmann and my course director, Prof. Vassil Alexandrov for providing me with the opportunity to do this thesis. I wish to thank Al Geist, Stephen Scott, Thomas Naughton, Geoffroy Vallee and my other colleagues within the ORNL Network and Cluster Computing Group. I also give thanks to Jack Dongarra and the many staff and post-grads who I was associated with during my time at the University of Tennessee. Finally I want to give thanks to the University of Reading ACET staff for providing a thoroughly enjoyable degree course.

Abstract

Powerful supercomputers often need to be simulated for the purposes of testing the scalability of various applications. This thesis endeavours to further develop the existing simulator, XSIM, and implement the functionality to simulate real-world networks and the latency which might be encountered by messages travelling through that network. The upgraded simulator will then be tested at the Oak Ridge National Laboratory. The work completed herein should provide a solid foundation for further improvements to XSIM; it simulates a variety of basic network topologies, calculating the shortest path for any given message and generates a transmission time.

Contents

1. Introduction

1.1 Overview

Since the dawn of the modern computer, there have existed many problems which are considered exceptionally large when compared to the computational demand of the average application. Such “Grand Challenge” problems (for example, accurate climate prediction modelling, highly reliable weather forecasting, simulation of the human brain, etc.) require resources which are often far beyond that available in the average commercially available computer. Therefore, a need exists to have computers which can at least attempt to solve some of these problems in a reasonable amount of time. Hence this has become the driving force behind the development of what is known as the 'supercomputer'; machines developed specifically for applications requiring “exceptionally high-speed computations” [1], and today mostly for scientific applications [15].

The growth in performance of such systems has been approximately exponential, in terms of the number of operations per second which can be performed, and this trend looks set to continue, at least for another decade. However, the means of achieving this growth have evolved over time; whereas in the past there was a focus on improvements in component engineering and integrating new hardware technologies into the design of the system, the modern approach is to increase the parallelism of such high performance computer (HPC) systems so there exist more nodes working concurrently on a problem. With the focus on parallelism, there are a few issues which now need to be considered in HPC design which were previously unnecessary. Probably the most important of these considerations is the inter-node (or inter-core) communication, which was either non-existent or unimportant in early supercomputers, since they contained relatively few cores.

Modern HPC systems can have millions of cores which are all connected together in a network. The communication overhead between cores can take a

significant percentage of execution time, and so it may become necessary to develop new algorithms for previously optimised programs, to optimise communication and thus execution. In the same manner that sequential programs often need to be largely rewritten to account for parallel systems, parallel programs may need to be restructured for massively parallel HPC architectures. Furthermore these optimisations may rely on the topology of the network and communication architecture of specific systems.

Often it is neither practical nor possible to use actual HPC systems when attempting to design, test and implement these algorithms. It may also be beneficial to study how the devised solutions cope not only with existing architectures, but with possible future HPC architectures. Within the last decade several attempts have been made to develop HPC simulators, which simulate the execution of millions or billions of processing cores running in parallel. Running programs on these simulators can then be done to gather experimental data on the programs' performance and scalability. This information can then be used to possibly either alter a program's algorithm or gain insight into new techniques to exploit massively parallel architectures.

The purpose of this project is to develop and upgrade an existing HPC simulator to account for network topologies and thus inter-node communication. The simulator in this project upon which development is taking place is 'XSIM', designed and implemented at Oak Ridge National Laboratory (ORNL) in 2009 by student Frank Lauer under the supervision of Dr. Christian Engelmann, ORNL researcher. The Oak Ridge National Laboratory in Tennessee, United States, contains what is currently the most powerful supercomputer in the world, named the Jaguar, as well as a number of other notable powerful HPCs and cluster configurations, making it an ideal place to work on projects which are related to the study of large scale architectures.

1.2 A Brief History of Supercomputers

It is difficult to determine which system could be said to be the world's first supercomputer, as the term is very loosely defined. One consensus is that this title belongs to the Electronic Numerical Integrator and Calculator (ENIAC) [14], often heralded as the world's first large-scale supercomputer [2]. ENIAC was developed in the mid 1940's as a military machine and its first task was to work on the Manhattan Project [5] during World War II. It could perform 1,000 floating point operations per second (FLOPS), or 1 KiloFLOPS (10^3 FLOPS) [6], which admittedly pales in comparison to modern standards [12]. FLOPS is a widely used standard for measuring and contrasting the performance of supercomputers.

In 1958, Seymour Cray (a long-time producer of HPC machines) developed the CDC 1604, the world's first completely transistor based system [8]. By the 1960's, the focus had shifted to utilising supercomputers for non-militaristic purposes, and they began to become used instead for scientific applications. In 1964, Cray then released the CDC 6600, which could produce up to 1 MegaFLOP (10^6 FLOPS) [11]. This was a significant landmark in terms of sheer processing power and remained the fastest HPC architecture in existence for 5 years, until surpassed by the CDC 7600 which could produce around 10 MegaFLOPS [4].

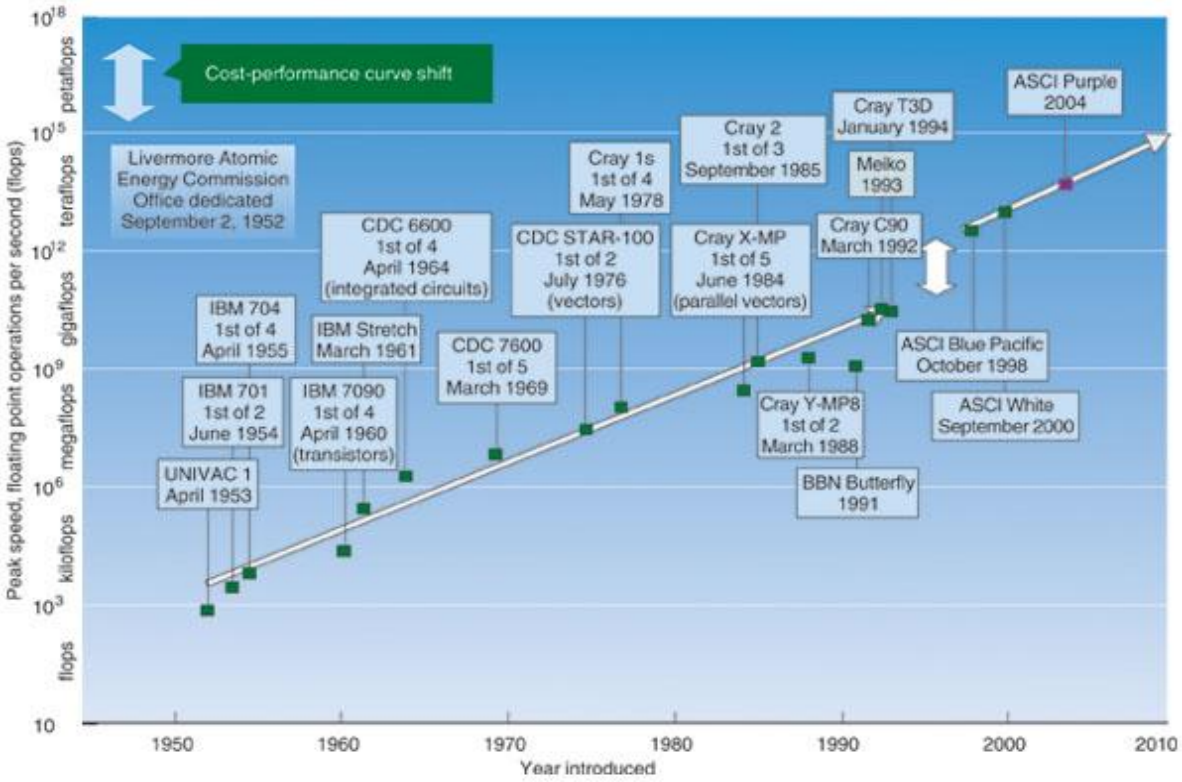

Figure 1: A time-line of the development of supercomputers [17]

In 1976 Cray then produced the Cray-1 system, capable of 160 MegaFLOPS, and costing around $8,800,000. Then, in 1985, came the Cray-2 which could produce 1.9 GigaFLOPS (10^9 FLOPS) [10]. This growth trend continued into the mid-nineties when supercomputers started to perform on the TeraFLOP scale (10^12 FLOPS) [13]. In 2008, the IBM HPC system nicknamed Roadrunner reached a then record performance of 1 PetaFLOPS (10^15 FLOPS), and the Cray Jaguar HPC system at Oak Ridge reached 1.75 PetaFLOPS in 2009, and currently at the time of writing is the fastest known supercomputer in the world. The Jaguar utilises a Cray XT5 system and approximately 2.24*10^5 processing cores. Assuming the current rate of growth in processing cores continues at the same rate, it would not be unreasonable to expect HPC machines with 10^8 cores by the end of the decade [8, 18].

Since the early nineties, there has been a collaborative effort to maintain a list which ranks the 500 fastest machines in the world. This list is compiled twice a year,

and the peak performance of an HPC system is defined by running the Linpack benchmark. Introduced by Jack Dongarra, the Linpack benchmark solves a dense system of linear equations [7]. The peak performance achieved over different sizes of the problem, as well as the theoretical peak performance, are then included within the top 500 list.

1.3 Related Work

There exist a number of HPC simulators developed in various scientific and academic institutions around the globe. They each have their own advantages and disadvantages as creating such a simulator is a very complex process. There are many considerations to make in the design and implementation stages depending on what purpose the simulator is going to be used for. Because supercomputers themselves often have large and complex architectures it is often not practical to simulate the entirety of its features, but rather only those which have an impact on the reasons for which the simulator is being created.

One such simulator of note is the Java Cellular Architecture Simulator (JCAS), developed by Dr. Christian Engelmann at Oak Ridge National Laboratory during 2002. Unsurprisingly given the acronym, JCAS is implemented in the Java programming language. It uses a TCP/IP protocol to simulate the communication between nodes, but these approaches have a few drawbacks, namely because both TCP/IP and Java are not supported on all systems, minimising portability. Nonetheless, JCAS is a robust simulator and one of its primary advantages is that it maintains a minimal operational overhead and has low resource requirements compared with other simulators, despite being written in Java. It can also simulate up to 10^6 simultaneous processes on just 10 actual cores. The simulator which is the focus of this thesis, XSIM, was originally based upon JCAS [25].

μπ is another HPC simulator, also designed and developed at ORNL, but created independently of JCAS by Dr. Kalyan Perumalla. It utilises the μsik Parallel Discrete Event Simulation (PDES) engine, and supports C, C++ and FORTRAN, as well as an extended range of implemented Message Passing Interface (MPI) commands. μπ can simulate up to 10^5 simultaneous processes and due to its methods of implementation is highly portable, able to run on a variety of common operating systems such as Windows, Linux and Mac, as well as cluster configurations and modern supercomputers such as the Cray XT5 architecture [20].

BigSim is also a HPC simulator which operates using Charm++, an object-oriented portable parallel programming language built on C++ [9]. Charm++ is machine independent and was built specifically to target the issues of portability, latency tolerance, dynamic load balancing, reuse and modularity. It is based around the idea of processor virtualisation where the user specifies the interaction between multiple virtual processes, and then directly maps the virtual processes to their physical counterparts. BigSim also uses the Adaptive Message Passing Interface (AMPI), a modified version of MPI, written specifically to work in conjunction with Charm++ [22]. BigSim was developed at the University of Illinois, by a research group headed by Gengbin Zheng [21]. Each virtual process in BigSim is implemented as a thread embedded within a Charm++ object.

1.4 Related Technologies

Before exploring the current state of the XSIM HPC simulator, it would be helpful to introduce at a few of the features which are central to its operation. One such feature is the use of the Message Passing Interface (MPI), which is a protocol that specifies the communication procedure between processes running in parallel. MPI consists of a number of routines which can then be called from various languages such as FORTRAN and C/C++. Some such routines are MPI_SEND and MPI_RECEIVE, which are used when a point-to-point message is passed from a sender process to a receiver process [16].

Other routines include MPI_SCATTER and MPI_GATHER which are for collective communication between multiple nodes, where MPI_SCATTER is used by the sender process to send data to multiple receiver nodes, and MPI_GATHER is used by the receiver process to receive data from multiple sender nodes. MPI also allows for specification and manipulation of communication domains, such that nodes can be grouped into categories if there exist communication patterns between different subsets. XSIM only employs a relatively small subset of the MPI routine library; the complete protocol is large and extensive allowing for advanced communication, which is unnecessary at this stage in the simulator's development.

Some of the features of note which XSIM does utilise include both blocking and non-blocking communication. MPI_SEND and MPI_RECEIVE are examples of blocking communication. When this approach is used, a node which calls one of these commands cannot proceed with other instructions until they have successfully completed. MPI_ISEND and MPI_IRECEIVE are examples of non-blocking communication. As the term implies, these MPI calls allow the node upon which they are executing to proceed even though the send or receive may not have completed. The orders in which instructions complete is often vital to a program producing the correct result, and so it is necessary to be mindful of data dependencies when using non-blocking communication and so to handle these manually [26].

Another related topic is Parallel Discrete Event Simulation (PDES). PDES refers to the simulation of a discrete event or program on a parallel computer. A program can be considered to be 'discrete event' when it consists of a series of sequential time-steps, each with a specific configuration of data held in memory. At any given point in the execution sequence, there is a fixed image which represents the entire program and indicates its current state. When the program is rolled forward a single time-step and data is modified, the image changes to reflect this. An MPI call is an example of an event which would alter the system between one step and the next. In a parallel computer, each node or process maintains an active queue of events which it has been allocated, and as time progresses these events are carried out.

In order to maintain reproducible results, a PDES system must be deterministic. That is to say, events must be guaranteed to execute in the correct order if they have dependencies between one another. To overcome this problem, each event is given a virtual time-stamp which indicates when that event was issued. Ideally, dependent events can then be executed in the correct order. It should also be noted that the system monitors the event-queues of all nodes to determine the lowest current time-stamp within the system. This is known as the global virtual time.

There exist two main approaches to ensure the deterministic nature of a PDES system. The first is the conservative approach. This relies on making absolutely certain that the next event, which is chosen from the various input queues belonging to a given node, possesses the lowest virtual time-stamp that it is possible for that node to receive. This decision is made after examining all the local time-stamps available, along with the global virtual time (GVT). The conservative approach can lead to a deadlock situation, where a sub-network of inter-communicating nodes are waiting on one another for an event with a lower time-stamp than any of the others currently within their respective local queues. This issue can be resolved if a 'lookahead' strategy is used, which foresees such deadlock situations by allowing neighbouring nodes to communicate.

The second is the optimistic approach. Some causality errors (when dependent events are executed out of sequence) may be permitted to occur because each node executes the event with the lowest virtual time-stamp that it can find locally, even if this is not guaranteed to be the lowest virtual time-stamp it will receive from an event which has not yet arrived. If a causality error does arise, the system can be corrected using 'roll-back' and 'time-warp' mechanisms which carefully reset the system to a prior state before the data was corrupted by the out of order execution.

Sometimes not all changes committed to the system since the error occurred need to be reversed. Some changes may have been outside the chain of dependent events and thus would have happened regardless of the error. Using this information, only the changes which causally followed from the error need to be rolled back. The large downside of the optimistic approach is that there are added memory costs of storing data about previous system states in case a roll-back is necessary. Such costs though are usually deemed acceptable when contrasted with the time wasted in conservative approaches when nodes idle unnecessarily waiting for neighbours to complete tasks when they may in fact have tasks of their own which could be completed without affecting the deterministic outcome of the system [23].

1.5 XSIM – An Overview

The XSIM simulator, originally developed by Frank Lauer, utilises a C++ core based upon the Java core of the JCAS simulator, and has an MPI layer running over the top to allow communication between nodes, modified from the TCP/IP layer of JCAS. The simulator has been implemented using a conservative PDES approach, meaning that there is no requirement for roll-back as it is impossible for causality errors to occur. However, deadlock is an issue that has to be handled, and there is no current 'lookahead' approach used to deal with this problem. Using the MPI_ANY_SOURCE instruction within a send or receive is also not recommended as it can lead to out of order execution.

XSIM utilises an algorithm to allocate each logical process a virtual MPI rank (a unique identifier for each virtual node). These ranks are then assigned a real MPI rank (a unique identifier for each real node), such that each real MPI rank has an equal number of virtual MPI ranks associated with it. This ensures that when running the simulator on a multi-core configuration, each core can be given a theoretically equal workload, and thus a simulation of 10,000 cores would quite happily run on a machine with 10 real cores, giving each real core 1,000 virtual cores to simulate. This same algorithm can then be used to determine the real MPI rank destination of messages for a given virtual MPI rank.

One issue in HPC simulator design is to ensure that inter-node communication is as realistic as possible. Variables belonging to one virtual node should not be readable by others without some form of communication, even if those virtual nodes reside on the same real node within the simulator. If this were the case, it would defeat the purpose of MPI communication between virtual nodes. To further this concept, the simulator contains no global variables. All data is locally created, stored and shared via communication, as would be the case in a real massively parallel modern HPC.

Furthermore, CPU contention can cause a few problems. Each virtual node

effectively emulates a single-threaded CPU, and thus only one logical process (or event, such as an MPI call) is allowed to be active at any given time. Any logical process that is not currently active is placed in a self-contained loop so that it has no concept of inactivity and believes itself to be always running. When the CPU switches from one logical process to another, it must save the variables and parameters of the outgoing process and load those belonging to the incoming process.

Each virtual node has been given two queues, an incoming queue and an outgoing queue. All outgoing queues are redirected to a special purpose CPU which has the task of overseeing inter-node communication, redirecting all received messages to the corresponding incoming queues of their destination nodes. Each message possesses two virtual time-stamps, one which indicates the time that the message left the source node, and the second which is an estimate for the time of receipt at the destination node. This information is used to determine in which order the messages should be processed, where higher priority is given to those messages which have lower time-stamps.

The root node keeps track of the global virtual time (GVT), indicating the lowest time-stamped event currently active in the simulator. The value of the GVT is useful for determining which currently queued events are safe for their respective nodes to execute. In optimistic systems the value of the GVT can also be used as an indication of where to roll-back to should a causality error arise. In order to maintain the correct GVT, it is necessary for each node to periodically communicate its local virtual time (LVT) to the root node.

The LVT of a node is determined by taking the lowest time-stamp of all events within its incoming and outgoing queues, in a similar fashion to the way that the GVT is determined by by taking the lowest LVT of all nodes. This gathering of all LVT data is done using a tree-like data structure. Neighbouring nodes communicate their LVT to a local root node, which then calculates the lowest LVT from those it receives (including itself), and passes this on to the local root at the next tier up the tree. This

divide and conquer approach eventually results in the minimum LVT (the GVT) being passed to the root node. The strategy is implemented asynchronously, as there is no specified time interval when the entire simulation pauses in order to update the GVT. Instead, each node updates its LVT whenever it is altered, thus the GVT is assured to be as up to date and accurate as possible [24].

1.6 Proposed Changes

Initially there was a list of proposed changes to XSIM, however throughout the development of the project these have evolved. XSIM is effectively a work in progress, and there are many directions which could be explored depending on what goals the simulator should be aiming to achieve. Originally the main focus points chosen included a concept of real-time, useful for gathering performance metrics on the run-time of the simulator which could then be used for statistical purposes to monitor running the simulator with different parameters such as the number of virtual nodes.

The second point considered was a concept of fault injection, such that node failures could be added to the system, both by manually specifying the point of failure, or by doing so in a stochastic fashion such that nodes would fail at random; information that would be passed to the simulator via command line arguments. This would allow for the study of fault tolerance, to monitor the simulator and investigate the maximum percentage of single points of failure that the system could handle before total failure ensued.

A few other minor alterations were also discussed. For example, extending the range of supported MPI calls and libraries, as XSIM currently only supports what is essentially the minimum number of calls necessary for performing a basic simulation, including both point-to-point and collective communication models. It is also possible that by exploring the inclusion of additional MPI calls there may be more efficient ways to execute some of the existing implemented methods. Before the start of the report, the most recent feature to be implemented was virtual time, and this may need some tweaking as the original method of implementing virtual time was inefficient and not very scalable.

One implementation feature that was decided against was to upgrade the simulator to handle an optimistic PDES approach. This is a large and complex task beyond the scope of the project, and implementing such would likely require the

dedication of an entire project to itself. One of the problems which leads to complications when developing an optimistic PDES system is the need to constantly allocate memory for the check-pointing process which holds the state of the system in case of a roll-back. Another feature considered but ultimately not selected was logical process migration; moving a logical process from one virtual node to another in order to optimise the speedup due to parallelism by ensuring all nodes are kept as active as possible. Once again a feature such as this would probably need an entire project dedicated to its development.

Ultimately, a single main objective was chosen on which to focus the project. This is to implement a concept of network latency within the simulator. At present, there is no concept of a network model. Messages sent and received between two virtual nodes are transmitted instantly, with no consideration for the time that the message would take to travel from the source to the destination. The objectives of the project have therefore been laid out as follows:

1. Simulate network latency.
    - 1.1 Develop network topology model.
    - 1.2 Implement latency cost calculations.
    - 1.3 Implement bandwidth considerations.
    - 1.4 Develop hierarchical structure within topology.

2. Implement fault injection.
    - 2.1 Enable determinate failure of node(s).
    - 2.2 Enable indeterminate/random failure of node(s).

As is evident in the objectives above, implementing fault injection has become a secondary objective and its completion will depend upon the complexity of the primary objective. Simulating network latency has several key components which

must be completed. Firstly, a concept of a network topology must be defined so that it can be determined exactly how the virtual nodes are connected together. Secondly, there must be some kind of relation between the network topology and the latency of traversing that network between the source and the destination; this latency must then be calculated. Thirdly, there must be some consideration of bandwidth which determines how quickly data can pass through the network. And finally, it may be necessary to design a hierarchical network topology.

The secondary objective of fault injection will then involve developing the functionality to turn off virtual nodes, either allowing the user to manually specify which nodes should be shut off and when, or allowing the user to specify a rate of failure and then select random nodes to shut off periodically throughout the program's execution.

2. Design

2.1 Assumptions

The first objective is to identify any necessary assumptions made about the system. For the purposes of simplicity is was decided that all elements of the network shall be uniform, meaning that the bandwidth of all links is identical, all nodes are identical (routers and processing nodes have no functional difference when routing messages), each link takes the same amount of time for a message to traverse across and the topology across the network is consistently identical.

One of the issues with the implementation of a network class, is that there are so many different possible levels of complexity which can be added to the simulator to provide greater functionality and more customisability for the user's, or the application's, needs. However, these are considerations which can be added later in the simulator's development; the main focal point for the implementation is to construct a working network class. However, as will be shown in the following sections, the ability to customise certain features of the network and its available topologies has been implemented should these be required.

2.2 Strategy

In order to create a meaningful concept of latency within the simulator, every communication between two nodes should have an associated cost, which reflects the time taken to send the message from the source to the destination. Latency is proportional to the network distance between the sender and the receiver, and thus also proportional to the number of intermediary nodes that must be visited en-route to the destination.

There are many different ways in which the nodes may be connected and thus many different forms of network topology which must be considered. The simulator must therefore be able to differentiate between topologies and produce an accurate latency value between any two given nodes, A and B. However, depending upon network configuration and furthermore, depending upon which two nodes A and B happen to be, this value may vary greatly within a single topology. For example, within a ring network, the latency between two adjacent nodes is going to be less than the latency between two distant nodes which occupy opposite sides of the ring. It may also be the case in certain networks that sending a message from A to B may not have the same cost as sending a message from B to A (for example, in a ring network where traffic is only permitted in one direction).

There are two possible ways to overcome this issue and calculate latency between source and destination. The first is to use data structures to simulate an actual network topology. For example, there may exist a node class, and when the simulator is executed, an object of this class is created for each virtual core. The class might then have a private array member titled 'links' which contains a dynamic list of all other nodes directly connected to that node; considered its neighbours. This idea though was quickly dismissed as a viable option purely due to both the computational and memory overhead that would be incurred from instantiating an object for every single core, particularly if XSIM is simulating millions of cores. In addition, it may be that the simulator is being used to execute a program which only utilises a small subset of the

total number of cores, making it a waste of time to define a large number of objects at start-up which will never be accessed.

Another problem with this approach is that calculating the latency between two nodes will require accessing the source node object and analysing its neighbours. Because for distantly located nodes there is no method of determining travelling to which neighbour will place the message closer to the destination, all neighbours must then be recursively accessed and analysed. The result can be, that if two nodes are located at the maximum network distance from one another, then every single core object is accessed and, once again, with millions of cores this is a significantly unacceptable amount of wasted overhead.

The second approach is not to create an actual topology simulation by using objects but instead to simply emulate the result of traversing the network by using mathematics to calculate the relative network distance between source and destination. For example, suppose there exists a simple bi-directional bus network (i.e.: a one dimensional mesh) where node zero connects to node one, which in turn connects to node two, two connects to three, three to four, and so on. Suppose for the moment that latency is purely based upon the number of intermediary links that the message must pass through when travelling from source to destination.

So passing a message from node two to node three would have a relative latency of 1. Passing a message from node two to node four (or from node four to node two) would have a relative latency of 2. Latency can then simply be defined as the absolute difference between the source rank and destination rank. Larger latencies may then be obtained by multiplying the relative distance obtained by a pre-determined 'multiplier', such that for example, traversing node two to three may have a latency of 10 and traversing node two to four may have a latency of 20. This mathematical approach is obviously far simpler than recursively accessing and analysing data structures for all objects as it can be performed instantly with one mathematical formula, although more complex topologies will need more complex formulae.

2.3 Function Design

It was decided that the simulator should at least implement some of the more common topologies in use today in order to be pertinent to modern HPC design. These should include a star network, a ring network, a mesh, a torus, a twisted torus and a tree. Time permitting, some of the more popular networks in existence within modern supercomputers may also be implemented such as the dragonfly topology, used within the Jaguar supercomputer at Oak Ridge, and the butterfly topology. The first step in design is to analyse each of these network types and calculate the mathematical formula for determining the quickest route between any source and any destination.

The star network is the easiest of all topologies to calculate, as the latency is constant regardless of the source and the destination. Ultimately, the latency function for a star network will simply echo back the predetermined latency multiplier and double this value to simulate the cost of travelling firstly from the source to the central switch, and secondly from the switch to the destination.

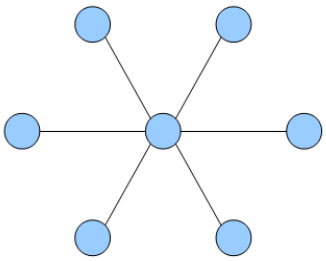

Figure 2: Star network

A ring network is also relatively straightforward to implement, because it requires calculating the integer difference between the source rank and the destination rank, and returning this value. However, there is the added complication that the destination rank may be lower than the source rank. As a ring is uni-directional, in this instance the message must travel to the final node in the ring, which then links back to the first node, essentially looping around the network. In this instance the latency will

instead be equivalent the total number of nodes in the ring, minus the difference between the source rank and the destination rank.

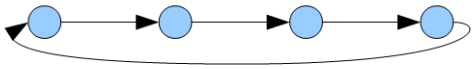

Figure 3: Ring network

A mesh network is both a simplification and an extension of the same concept used in a ring topology, except that it is legal to travel in both directions, and there may be more than one dimension to consider. In this case, calculating the distance between source and destination is simply a case of traversing each dimension one at a time, effectively aligning each dimension in turn from the source to the destination. Because the dimensions of a mesh do not loop around there is no need to consider whether the source rank is lower or higher than the destination rank as is the case with a ring, but simply to calculate the absolute difference between both source and destination. Once each dimension has been successively aligned, the total number of steps needed to do this in each dimension can be totalled to work out the latency.

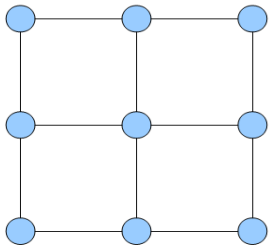

Figure 4: Mesh network

A torus network works in exactly the same way as with a mesh, except that like a ring, each dimension can loop around on itself, so the same technique for doing this with a ring is re-used to calculate the difference should a loop around become necessary. However, unlike a ring, a torus is bi-directional and messages can travel in either direction. The implication of this is that if the destination rank happens to be lower than the source rank (and thus would indicate a definite need for a loop-around

in a ring), it does not necessarily mean that looping around will produce the fastest result. In fact, for every dimensional alignment, regardless of source rank and destination rank, it must be checked whether it would be quicker to travel directly between source and destination, or to loop around. Once again, after all dimensions are aligned in this way, the total number of steps required yields the latency.

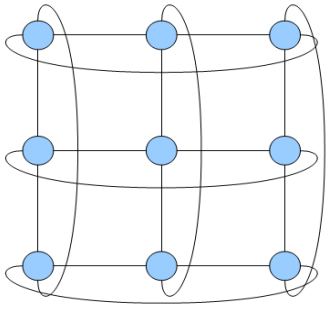

Figure 5: Torus network

A twisted torus is perhaps the most challenging topology of all those listed above. In a twisted torus, once the last element of a given dimension is reached, the structure loops around to the first element of that dimension as with a standard torus. However, unlike the standard torus, other dimensions may also be incremented (or decremented depending on the direction of travel) whenever a dimension loops around. Ultimately this means that the dimensions are smeared (or twisted) into one another, and it is no longer a case of traversing each dimension independently of the next because dimensions now in fact depend on one another. The quickest route between source and destination becomes less obvious. The issues of a twisted torus will be further addressed in the implementation section of this report.

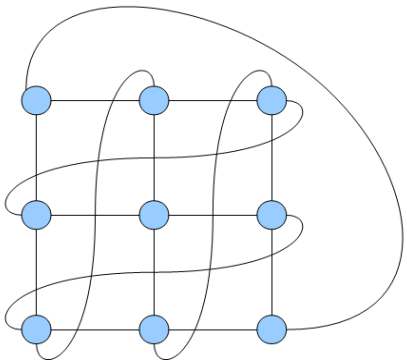

Figure 6: Twisted torus network

Finally there is the tree topology. The latency calculation must allow for trees of varying degrees, not just the obvious binary tree. Latency calculation will involve using the source rank and destination rank to calculate how far up the tree the two nodes share a common ancestor, and then working the number of steps it takes to get up to that ancestor from the source rank and back down to the destination node from the ancestor. The results obtained from the mathematical formulae described above are then multiplied by the latency multiplier to calculate the base latency.

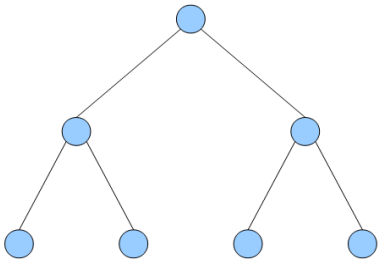

Figure 7: Tree network

The time that a message takes to travel from source to destination is of course not only dependent upon the route it takes through a given network topology, but also upon the size of the message being transmitted, as well as the bandwidth. XSIM currently already contains inbuilt information regarding message size, and this will need to be factored into the latency calculation. There is no correlation between the

message size and the route that the message takes through the network. So an additional parameter of bandwidth will need to be specified which will then be used in conjunction with the message size to work out how long it will take to get a message through a given point in the network. This result will then be added to the base latency calculated previously to determine the total latency for getting the message through the selected network route.

Then, there is the issue of hierarchy. The latency between all nodes is not identical; the cost of traversing a link between neighbouring nodes is not consistent across the entire network. Although there may be various reasons why this is not the case, in this project it has been chosen to largely ignore the more minor variations, but to instead consider the larger implication of core hierarchy. Thus far, it has been assumed that all cores operate within the same local environment, however in reality, it may be that the HPC consists of many processors, each with its own subset of cores.

The cost of travelling from one core to another located within the same processor will be less (possibly far less) than the cost of travelling from one core to another located on different processors. A method needs to be calculated to determine which cores are located on which processors, and thus whenever a message is sent between two nodes, it can be recognised if they are on the same processor or not. It would then also be necessary to specify two different network topologies, one type of topology that indicates how each processor's cores are connected, and another that indicates how all processors are connected together within the network. Using this information, the latency calculations can then be applied for either inter-core communication, or inter-processor communication using the appropriate parameters for each to determine the correct base latency.

The secondary objective of the project is to attempt to implement some method of fault injection. The two stages of this are deterministic fault injection and statistical fault injection. In deterministic fault injection, the user can specify the failure of specific nodes at will via the console during execution. The simulator then shuts these

nodes down such that no message may pass through them, and must find alternate routes. In doing so, the mathematical models of the aforementioned latency calculation will have to be altered to account for this. There may be some kind of limitations which specify if node [x,y,z] is offline, then there is a specific set of inhibitory rules which come into play and limits the freedom of messages to move in those dimensions which violate that position.

For statistical fault injection, the user specifies a likelihood of a node being shut-down. This is a percentage value, which can either indicate how many nodes will randomly fail in any given time-step, or how many nodes should fail over the course of the simulator's total execution time; the second alternative might be problematic as it is not known how long a given application will take to run until it completes. The exact nodes which go offline, and when they go offline, is entirely random and, once again, the mathematical model for latency calculation will have to be updated to account for this phenomena in a similar way to before.

For both of these varieties of fault injection, it may also be possible to recover nodes. In deterministic fault injection, the user may specify a node which has come back online that was previously offline, and in statistical fault injection, it may be that after a given number of time-steps, any nodes which have entered failure are restored to a working state. This would require a dynamic management of the mathematical latency model so that it is kept in constant awareness of which nodes are working and which are not and finds the fastest route regarding only those nodes which are presently operational.

3. Implementation

3.1 The Network Class

The first step taken during implementation was to develop an entirely new object class dedicated to the network model. This network class is responsible for performing all manner of calculations relating to the network topology and any associated latency calculations. It has two main responsibilities within the simulator; firstly, when the simulator is initialised, the network class is instantiated as an object and various arguments are passed to it (taken from the command line, specified by the user) which determine the nature of the network. Secondly, every time a virtual node receives an MPI message, a method of this network class is called to determine the time the message took to travel through the network from the source to the destination, and this 'latency' value is returned.

The simulator was then modified such that the user is able to specify a single argument for the network class which in itself may contain a number of different parameters. This is the argument which is then passed to the network object when it is first instantiated, and the network constructor breaks down this often lengthy parameter using delimiters to extract its various elements which are then stored in private variables belonging to the network object. Arguably the most important such parameter is the network type, for instance mesh or torus, which defines the topology of the network. If no network type is specified, a default star topology is assumed. The next argument is the network-latency multiplier, which determines the latency or traversal time of a single link between two neighbouring nodes.

Later on in the implementation, when the hierarchical element was being implemented, a processor-latency multiplier was also added which determines the latency or traversal time of a single link between two neighbouring cores within a processor. An additional argument was added to supply the number of cores that each

processor has. Then, when calculations are being done to work out the latency of a message during execution, and it can be determined whether the source and destination are on the same processor or different processors, either the network-latency or the processor-latency can be called upon depending upon the result.

Most of the remaining parameters have duplicates as with the network-latency and processor-latency, because it is necessary for defining both the structure of the network as a whole and the structure of the processors. There is also network/processor degree which indicates the number of dimensions that the relative topology has, and network/processor dimensions which specifies the values of those dimensions. For torus and twisted torus topologies, there also exists the ability to handle network/core toroidal properties which specify which dimensions, if any, are toroidal (wrap around from the final element to the first). For twisted toruses there are two further parameters; toroidal jump is an integer array and toroidal degree is a single integer. The significance and use of both of these will be explained when the development of the twisted torus is explored.

Finally, there exists the network/core bandwidth which determines the rate at which data can pass through the links between nodes and this is used to calculate the additional latency overhead relative to message size. Bandwidth is specified in Megabits per second (Mbps). An example argument passed to the network constructor may appear as follows:

ntype=mesh,nlatency=100,ndegree=2,ndimensions=3*3,nbandwidth=10,cores=8,ptype=twisted,platency=10,pdegree=3,pdimensions=2*2*2,pt_connectedness=101,pt_jump=1*1*2,pt_degree=1,pbandwidth=100

Most of these values are assigned defaults if they are not specified by the user. There also exists some error checking to ensure that various parameters that rely on each other are compatible. For instance, the product of the processor dimensions must equal the number of cores, and number of cores multiplied by the product of the

network dimensions must equal the number of virtual nodes, found within a separate argument specified by the user. This error checking is done in the constructor after initialisation to ensure that there will be no problems and conflicts later when attempting to calculate message latencies.

3.2 The Apply Method

When an MPI message is received by a node, the apply method of the network object is called to calculate how long the message took to travel through the network. The first check that is made is to determine whether the source and the destination are on the same processor or on different processors. In a similar manner to the way in which the virtual network topology is not actually stored in a data structure but calculated mathematically, it is not known by any external process which virtual cores are on which processors. Instead, this is calculated every time the apply method is called.

This is simply determined by taking the source rank and the destination rank of the MPI message, and dividing them both by the number of processors (calculated by dividing the total number of virtual nodes by the number of cores per processor). If the results are equal, then they exist on the same processor, otherwise they are on different processors. This considered, one of two things may then happen. Firstly, if the nodes are seen to be on different processors, then the function proceeds to the next step. However, if the nodes are on the same processor, the function is recursively called, this time passing a different set of parameters.

The reason for using different parameters becomes apparent when calculating the latency is considered. If the nodes are on different processors, the network latency and network bandwidth will be needed. However, if the nodes are on the same processor, the processor latency and processor bandwidth will be needed. Furthermore, the topology of the network and the topology of the processors may not be identical. It may be the case for example, that each processor operates a 2*2*2 mesh of 8 cores, and all processors are then connected in a binary tree; the number of possibilities is vast. Even if the topologies are identical, it is necessary to calculate the relative position of the source and destination within the network, or within the processor.

The source and destination ranks with respect to the network can be calculated

by identifying the ranks of the processors that they are each located on. The processor rank is determined as before, by dividing both source and destination by the total number of processors. The source and destination ranks with respect to the processor are only necessary if they are located on the same processor. In this case, the modulus function is used instead of divide, to obtain the remainder after calculating the processor rank. This remainder then determines the 'core rank' of both source and destination, within the processor. For example, suppose the source rank is 14 and the destination rank is 20, operating on a network composed of processors with 8 cores each. The source rank is on processor 14/8 = 1 and the destination rank is on processor 20/8 = 2. They are on processors 1 and 2 respectively.

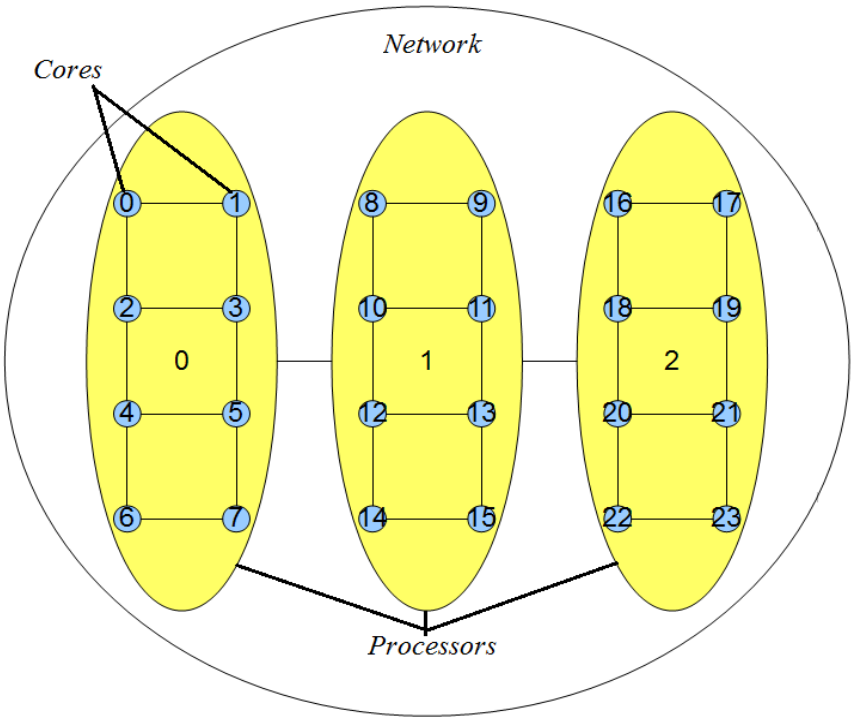

Figure 8: Example of node partitioning into a network with 3 processors and 8 cores per processor

Alternatively, suppose the destination rank is now 15. Now, both ranks are on processor 1. So we then determine their respective ranks on the processor. The source core-rank is now 14%8 = 6 and the destination core-rank is now 15%8 = 7. So the

core-ranks are 6 and 7 within processor 1. Depending on whether or not source and destination are on the same processor, either network or processor-specific arguments are passed to the next function, which calculates the latency.

In order to do this, the topology has to first be identified, which is done via a large case statement. If the source and destination ranks are on different processors, the network topology is needed; however if the ranks are on the same processor, the processor topology is needed. A topology-specific function is then called, to which various arguments are passed for calculating the latency of the MPI message. This function will always take the arguments for source, destination, and the latency-multiplier (which as explained previously will all vary depending on the original source and destination ranks). Additional arguments may be required depending on the network type, and these will be discussed in the following section.

3.3 Topology

3.3.1 Star and Ring

The star topology was as straightforward to implement as anticipated. It is the only network topology for which there exists no function for calculating latency; since source and destination rank have no bearing on the latency, there is no need for one. The star case simply echoes back double the latency-multiplier value. This doubling is done to account for the two steps in sending a message through a star network. First the message must travel from source to the central node, then from the central node to the destination, as explained previously.

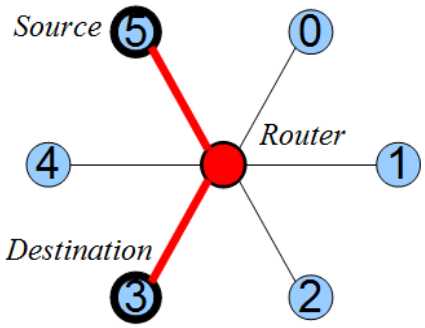

Figure 9: A message will always take 2 steps in a star network

The ring topology does need a separate function because the latency is dependent upon the source and destination. Again, as discussed in the design section, this is rather straightforward; all that must be done is to find the product of the latency-multiplier and the number of intermediary links that the message must travel through to get there. If the destination rank is ahead of the source, then this is destination minus source, and if the source rank is ahead of the destination, then this is node count minus source, minus destination, as illustrated below.

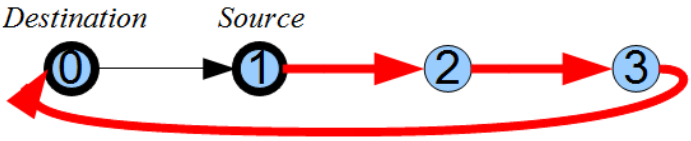

Figure 10: A message 'looping around' a ring to reach its destination

3.3.2 Mesh

Implementation of the mesh topology becomes a little more complex as various extra parameters are now necessary. First, it is necessary specify the degree of the mesh, or the number of dimensions that it has, as well as exactly what these dimensions are. There might be a square mesh of degree 2 with dimensions 5*5, or a mesh of degree 4 with dimensions 8*8*3*2. If a mesh is thought of as a vector space, a node's position within that space can be defined as a Euclidean vector. So in order to calculate the positions of the source and destination ranks, these must be broken down into their Euclidean counterparts.

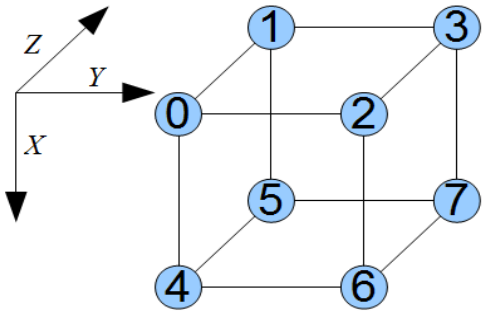

Figure 11: All dimensional representations hereafter are laid out in this fashion

The initial ranks are divided by the first dimension, and the modulus of this divisor gives the location in this dimension where the node can be found. Then, the divisor is divided by the second dimension, and once again the modulus of this second divisor specifies the location of the node in the second dimension. This procedure is done repeatedly depending upon the degree of the mesh until both source and destination have been translated into two Euclidean co-ordinates. For instance suppose the source co-ordinate is 6, and there exists a mesh of degree 3, consisting of dimensions 2*2*2. Dividing the source rank by the first dimension; 6/2 = 3 remainder 0. So the z dimension is 0. Taking the divisor 3 and dividing by the second dimension, 3/2 = 1 remainder 1. So the y dimension is 1. Taking the divisor 1 and dividing by the third dimension, 1/2 = 0 remainder 1. So the x dimension is 1. By reversing the order of these results, it can be seen that node 6 is located at co-ordinates (1,1,0).

The following illustrations demonstrate some possible mesh configurations and the resultant co-ordinates that are obtained by performing this calculation on each node. This is merely to give an idea of the relative sorts of positions of each node within mesh networks, and the general relationship that exists between location and rank. These guides can be extrapolated for networks of any degree and size. It should be noted from Figure 11, that unlike the conventional labelling of axes, in the subsequent diagrams the vertical is the x-axis and the horizontal is the y-axis. There is no particular reason things were done this way, it became personal preference when making diagrams throughout the project.

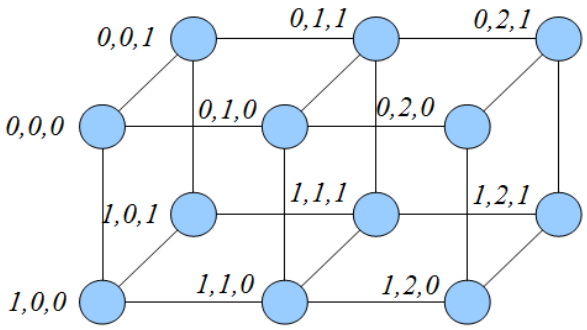

*Figure 12: The Euclidean co-ordinates in a 2*3*2 mesh*

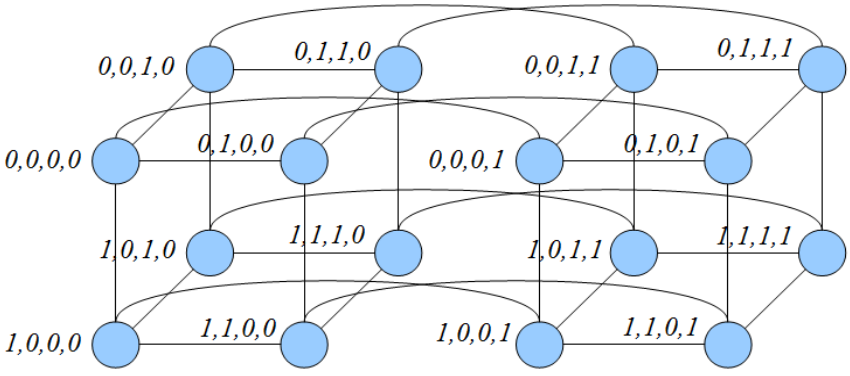

*Figure 13: The Euclidean co-ordinates in a 2*2*2*2 mesh*

Once the Euclidean co-ordinates for source and destination have been

calculated, the quickest route can be determined by considering each dimension in turn, and the respective co-ordinate in both the source and destination rank, and summing the absolute differences. Suppose that there exists a 4*4 mesh, the source rank is 5 and the destination rank is 15. The respective Euclidean co-ordinates are [1,1] and [3,3]. So the latency can be calculated by summing the differences in the x dimension and the y dimension; (3-1) + (3-1) = 4, as illustrated below. By finding the product of this result and the specified latency multiplier, the latency can be calculated.

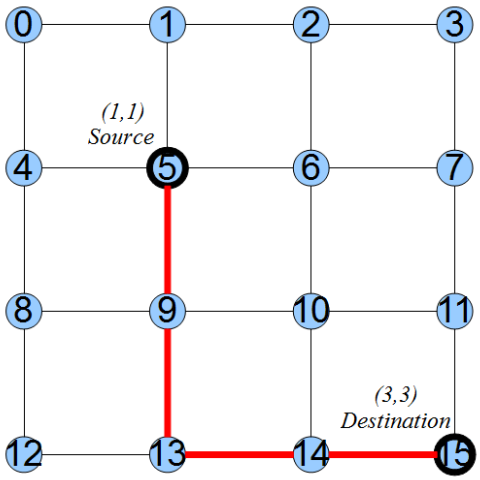

Figure 14: One possible shortest path that the message may take

3.3.3 Torus

The latency in a torus network is calculated in a similar fashion. A torus network is almost identical to a mesh, with the added consideration that each dimension wraps around from the last element to the first, in a similar manner to a ring (except of course that it is possible to travel in both directions). So calculating the latency is done in almost exactly the same way. Euclidean co-ordinates are extracted for source and destination and the differences are then summed. However, it is now possible to make the transition in each dimension in one of two directions; either as in a mesh, where the message travels through the centre of the mesh between the source and the destination, or by looping round in that dimension and crossing the link which joins the first and last elements.

The toroidal connectedness is an additional parameter which is passed to the torus latency function. This binary string specifies which, if any, dimensions are toroidal. The second option of travelling around a dimension via the link connecting the first and last nodes is only available in dimensions which are toroidal. For example, [1,0,1] indicates that the first and third dimensions are toroidal but the second is not. Non-toroidal dimensions are therefore treated exactly like they are in a mesh, by calculating the absolute difference in source and destination co-ordinates. A torus with no toroidal dimensions is therefore simply a mesh.

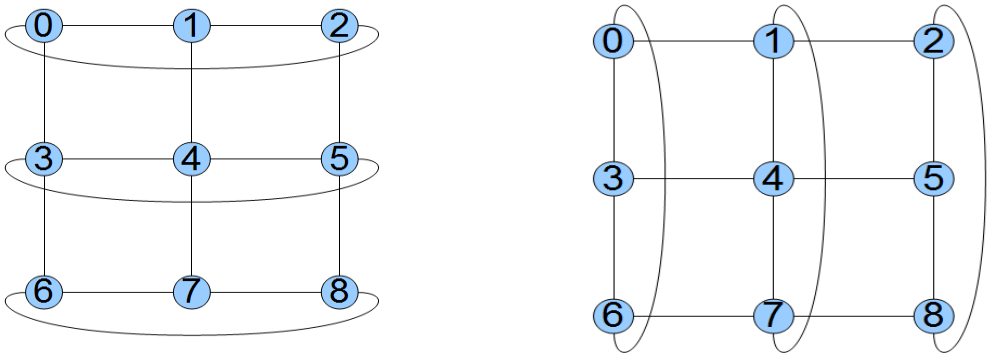

*Figure 15: A 3*3 torus with toroidal connectedness vector of [0 1] (left) and [1 0] (right)*

Obviously for toroidal dimensions, it becomes necessary to calculate which of these methods is fastest for each dimension. This is a fairly straightforward operation; the respective co-ordinates of a dimension are extracted from the source and destination, and their absolute difference is returned as in the mesh function. This value is then compared with the size of that dimension. If the absolute difference is greater than half the size of the dimension, it is evident that it would be quicker (in the sense that the message passes through less intermediary nodes) to traverse the dimension by looping around via the link connecting the first and last nodes.

For example, suppose that the Euclidean co-ordinates of the source and destination are [1,0] and [3,3] in a 4*4 torus. The differences between each dimension are x = (3-1) = 2 and y = (3-0) = 3. Considering the x dimension; 2 is not greater than half of the size of the x dimension, 4, so it would not be any quicker to traverse that dimension by 'looping around' between the first and last nodes. In fact in this instance, because 2 is exactly half of 4, it wouldn't make any difference in which direction the message travels – both methods would have to travel across 2 links. Now considering the y dimension; since 3 is greater than half of the size of the y dimension, which is again 4, it would be quicker to traverse this dimension by looping around between the first and last nodes. The following illustration demonstrates this route, along with a few other examples.

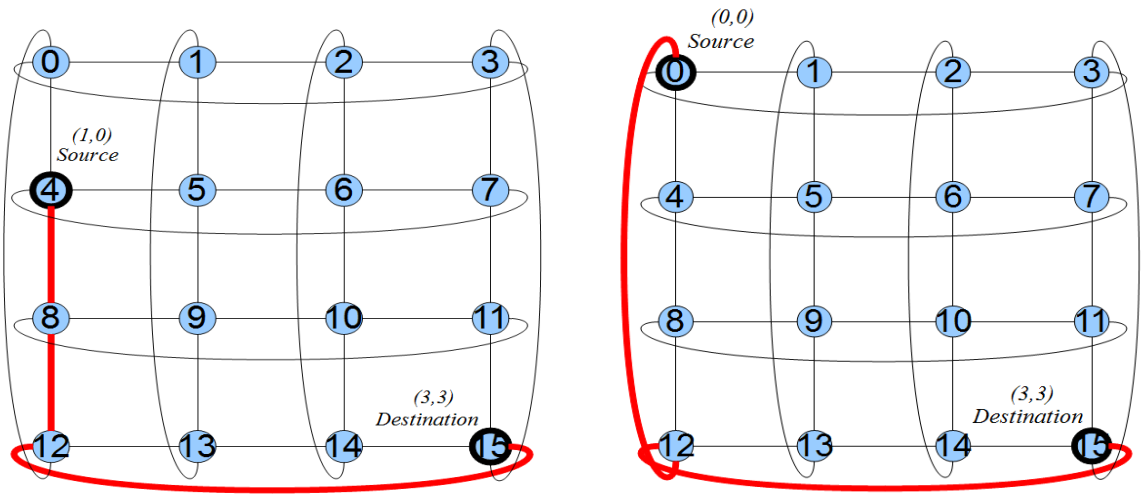

*Figure 16: A 4*4 torus with possible shortest paths for a couple of communications*

3.3.4 Twisted Torus

The twisted torus function is the most complex of all, comprising a large number of loops which perform various checks and comparisons between co-ordinate values. As with mesh and torus, the first task is to calculate the Euclidean co-ordinates of both source and destination. The big difference in a twisted torus is that dimensions are smeared together. In a mesh or torus, traversing along one dimension has no bearing on the position of the message in regards to the other dimensions. All dimensions essentially exist independent of one another and can be treated as such. However, the nature of a twisted torus, as briefly outlined in the Design section, means that when a toroidal dimension links between the first and last nodes, it also changes the value of another dimension.

Considering this fact, there are a further two parameters which are passed to the twisted torus function, as well as all the previous parameters which are used for a torus (source, destination, latency, degree, dimensions and toroidal connectedness). The first such parameter is the toroidal degree; this determines which degree is affected by travelling toroidally in a given dimension. Suppose there exists a twisted torus network with degree 3 and toroidal degree 1. Assuming that all dimensions are toroidal, this means that the x dimension (first) loops around into the y dimension (second), the y dimension loops around into the z dimension, and the z dimension loops back into the x dimension.

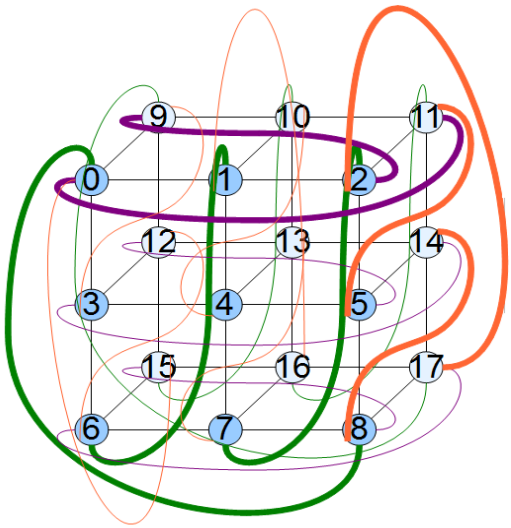

*Figure 17: A 3*3*2 twisted torus with toroidal degree of 1, twists of the frontal nodes are highlighted*

Now assume the toroidal degree is 2. The x dimension (first) loops around into the z dimension (third), the y dimension loops around into the x dimension, and the z dimension loops around into the y dimension. Simply put, the toroidal degree specifies how many dimensions 'ahead' of the current dimension must be counted in order to find the dimension which is changed by the twist; it determines which dimensions smear into which.

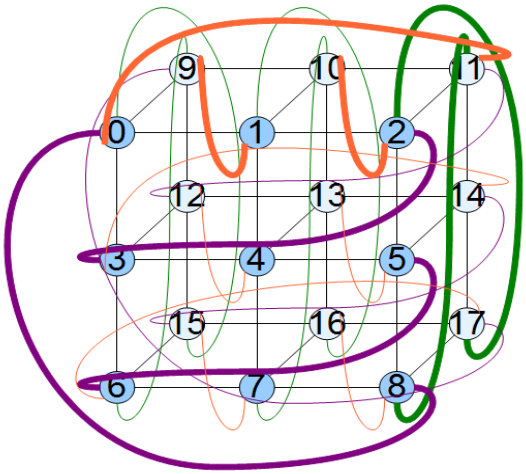

*Figure 18: A 3*3*2 twisted torus with toroidal degree of 2, twists of the frontal nodes are highlighted*

The second parameter is toroidal jump; this is a vector which determines by how much the related dimension is incremented when the dimension being operated on loops around. Assuming a twisted torus of degree 3, where all dimensions are toroidal, a toroidal jump vector of [2,3,1] and a toroidal degree of 1 would imply that looping around in the x dimension would increment the y dimension by 2, looping around in the y dimension would increment the z dimension by 3, and looping around in the z dimension would increment the x dimension by 1. Again, the following diagrams are provided to demonstrate variations of the toroidal jump.

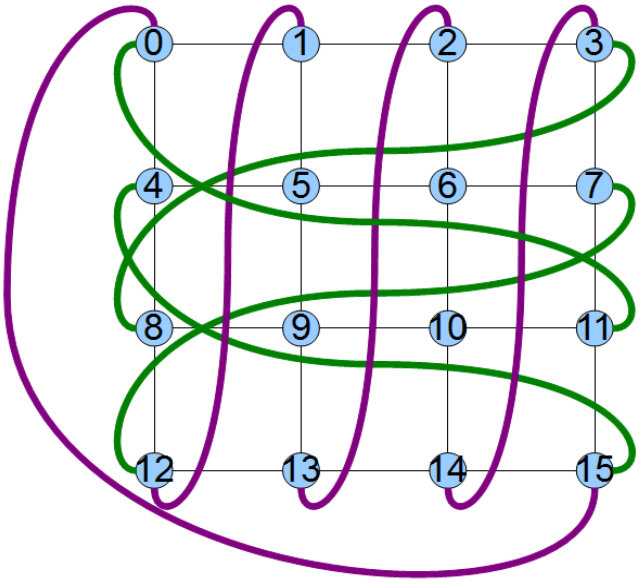

*Figure 19: A 4*4 twisted torus with a toroidal jump vector of [1 2]*

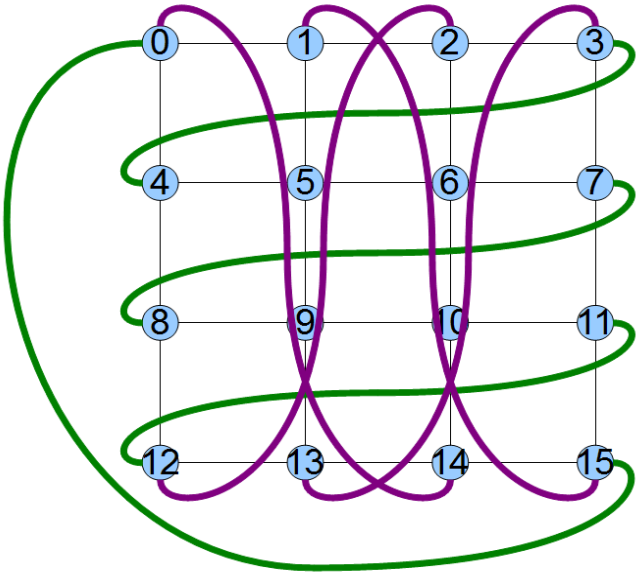

*Figure 20: A 4*4 twisted torus with a toroidal jump vector of [2 1]*

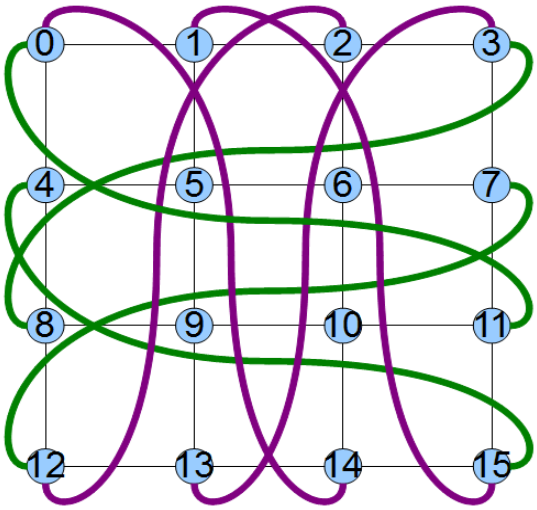

*Figure 21: A 4*4 twisted torus with a toroidal jump vector of [2 2]*

After calculating the Euclidean co-ordinates, the twisted torus function then proceeds to examine each dimension in turn. Two vector variables are used to hold the positions after traversing that dimension directly and by traversing that dimension by looping around, as was done in the torus function. The direct vector simply involves replacing the corresponding co-ordinate in the source with the one in the destination. For example if the source were [0,2] and the destination were [3,1], then examining the

x dimension would give us the new position [3,2], equally examining the y position would instead yield [0,1].

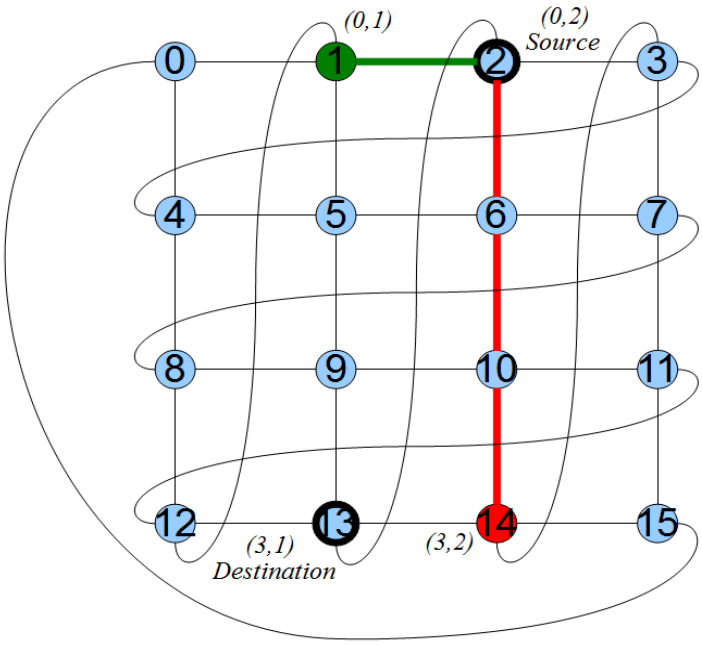

Figure 22: Testing direct traversal in the x (red) and y (green) dimensions

If the dimension is toroidal, the loop around vector is calculated too. This involves replacing the corresponding source co-ordinate with its destination counterpart as before, however now it must be taken into consideration how the other dimensions are affected by this loop. Firstly, it needs to be determined which secondary dimension is affected by traversing the current one. The toroidal degree parameter comes into use here, however it needs to be considered that the current dimension plus the toroidal degree may be greater than the number of total dimensions. For example, in a twisted torus of degree 3, with a toroidal degree of 1, there is no fourth dimension for the third to affect, and instead it affects the first.

Similarly the secondary dimension needs to be examined to see which value it currently holds. The toroidal jump vector is also needed to see how by how much this value is to be incremented. Once again, it may be the case that the current value plus the value extracted from the toroidal jump vector, exceeds the size of that dimension.

For example, in a 4*4 twisted torus, if the toroidal jump is [2,2] and the x co-ordinate is looping around, affecting the y co-ordinate, then the y co-ordinate needs to be incremented by 2. Suppose the Euclidean vector is currently [0,3]; incrementing 3 by 2 will give 5, but the dimensions have already been defined as 4*4 and so this value is clearly 'outside' of the mesh boundaries. As with the toroidal degree, it becomes necessary to perform a check such that this does not happen, and thus the new value of the y co-ordinate in this case would be 1.

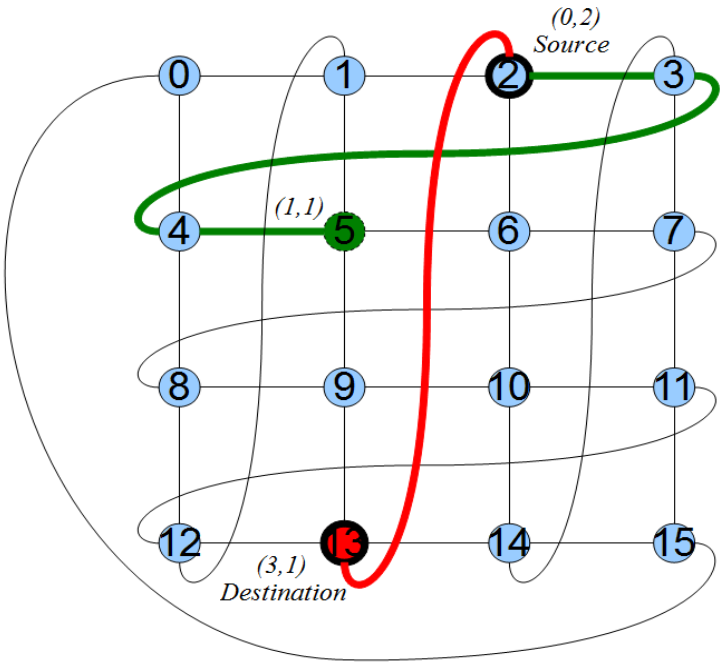

Figure 23: Testing loop-around traversal in the x (red) and y (green) dimensions

After the function has calculated the new position of the message after both direct traversal and looping around, in every dimension, it proceeds to compare all results. Each possible option is given an associated cost. This cost is comprised of two parts; the first is the distance that the new message position is from the destination. This is measured in the same way as distance in a torus, by taking the sum differences of all co-ordinates. The second part consists of measuring how far the message has travelled to get from the source to its current location. This is done by measuring the difference of the value of the dimension which was being traversed in that step.

Finally, the direction of travel with the lowest associated cost is chosen and the message advances to this point in the network. The source is updated with this new position, and that dimension is marked as being traversed. The cost of this chosen direction is also added to a cumulative total. Then the entire process of testing all dimensions starts over, but only dimensions which have not yet been marked as traversed are tested. Of course, if the message happens to have already reached the destination before traversing all dimensions, then the algorithm stops, as is the case in the illustrated example above. One way or another, the message will eventually reach its destination, and the product of the cumulative total and the latency multiplier again yields the final latency. Whilst this is not a one hundred percent guaranteed method of finding the shortest path to the destination, it is reliably accurate in the vast majority of situations, and so for the purposes of the simulator the small margin of error which sometimes arises will suffice. These errors will be investigated further in the conclusion section.

3.3.5 Tree

The final network type to be implemented was tree. Initially, it was thought that every node at every level on the tree would hold a processing core. However this assumption was found to be incorrect as in reality, tree networks only hold cores on the leaf nodes. Nodes on higher levels act as simple routers which forward data. With this in mind, the tree function is relatively straightforward. The source and destination nodes are both recursively divided by the degree of the tree (for example, 2 in a binary tree). Every division identifies the parent of that node, so when the source and destination are divided to find the same result, this identifies their common ancestor in the tree.

The number of divisions made is summed and multiplied by two to account for travelling up the tree from the source to the common ancestor, and then back down to the destination. As with all network types, this product of this value and the latency multiplier yields the final latency result which is returned to the apply method.

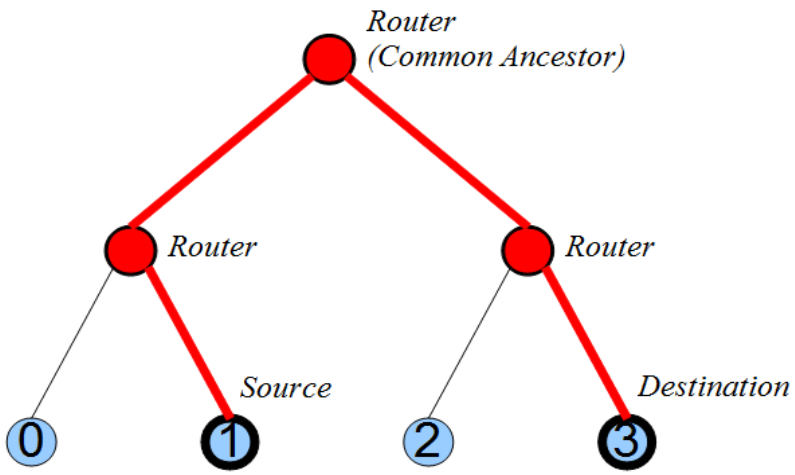

Figure 24: Traversing a binary tree after finding the common ancestor

3.3.6 Other Considerations

Once the latency cost of traversing the network has been established, the apply method then needs to factor in the bandwidth. Depending upon whether the source and destination are located on the same processor or different processors, either the processor bandwidth or network bandwidth will be used for this operation. The message has an attribute which determines its size in bites, so this is divided by the product of the bandwidth value and a constant. The bandwidth value is specified in Mbps, so the result of this calculation gives the number of seconds which the message takes to pass into and out of the network.

Towards the end of implementation, the entire simulator was converted from C++ to C, so it was necessary to also convert the network class to C. This did not take too long, but required changing a few memory management statements, and restructuring the methods as functions, which also had implications for accessing data as variables no longer belonged to any sort of object, but had to be defined in a different way. An enum type was used to store the network/processor topology parameters, and these parameters were defined upon the simulator's creation. The apply method was transformed into a function which is called as before, when an MPI Receive takes place.

The secondary objective of implementing fault tolerance was never completed, as it was realised this would require an additional re-working of the mathematical network model.

4. Testing

4.1 General Performance Analysis

The parameters that can be examined during testing are: number of real nodes (NP), number of virtual nodes (VP), network topology, network latency multiplier, network bandwidth, network degree, network dimensions, network toroidal connectedness, network toroidal jump, network toroidal degree, number of processors/cores per processor, processor topology, processor latency multiplier, processor bandwidth, processor degree, processor network dimensions, processor toroidal connectedness, processor toroidal jump and processor toroidal degree.

Obviously it would be impractical to test all possible variations of these parameters, as the number of permutations in doing so is huge, even after considering the fact that various parameters are inter-dependent upon one another in some scenarios. Therefore, test cases will be made for each parameter, with all others remaining constant, where possible. This should at least give basic insight into how altering each parameter, and thus altering the network topology, can impact the performance of the simulator.

Varying NP and VP has already been covered in Frank Lauer's thesis, however considering the simulator in its altered state with new functionality, it would be useful to re-examine the performance of scaling both of these variables once again. There exist several programs included with the simulator which might be useful for testing performance. Two were chosen for use in testing; these are Ring, which passes MPI messages around the network as if it were a ring, from one neighbour to the next, and Random, which is very similar to the Ring function because it passes the same number of MPI messages, except that each message travels from a randomly selected source to a randomly selected destination, as opposed to the pattern seen in ring communication.

The Random program should be useful in obtaining average latencies which suggest the general usefulness of a given network setup when used for a wide variety of applications. The Ring program is used to give an example of a specific type of communication pattern and how this can affect the results and efficiency of various network types. The system on which tests were carried out is a cluster utilising 128 cores, based at the Oak Ridge Laboratory.

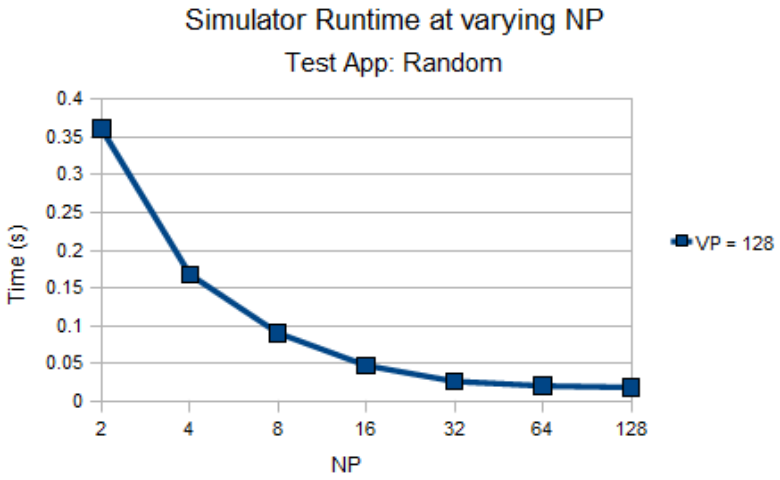

Figure 25

The first test undertaken was to vary NP and check the resultant performance time. The topology used was an 8*16 mesh. As might be expected, when NP increases the execution time decreases. Compared with the results found by Lauer, there is a notable difference in the execution time at each stage. However, it should be noted that this is most likely the result of using a different test application, namely Random as opposed to Heat Transfer. This can be explained by the fact that the Heat Transfer application utilises a far greater number of MPI communications. Despite this difference, the general trend appears to be similar, as would be expected, because there are more cores to split the workload. Evidently as NP continually doubles, the increase in performance slows down exponentially, as communication becomes a predominant factor in execution time.

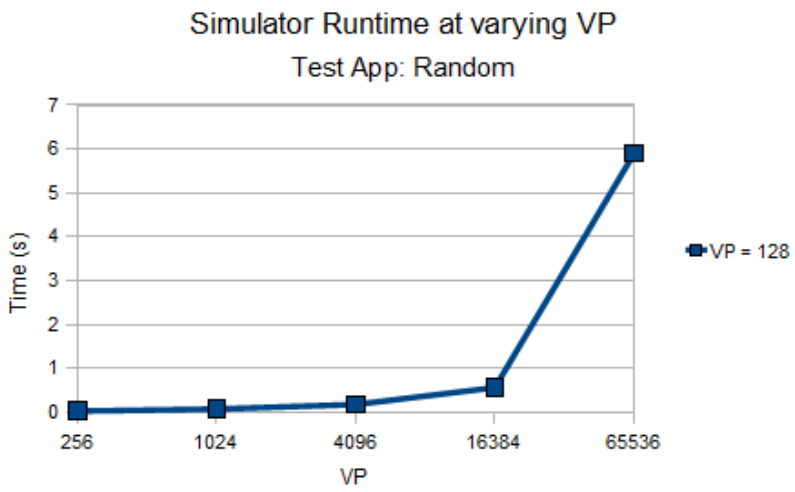

Figure 26

VP was then tested, and varied up to a maximum of 65,536 nodes while NP was kept at 128 nodes throughout. The number of MPI messages in the Random and Ring applications is of the order n^2 as each of the n nodes must pass on n messages, one from itself and one from every other. With this in mind, it is clear that the rate of increase in execution time is likely a result of both the increased MPI message count from a higher value of VP, and the increased communication overhead that exists between virtual nodes as VP is increased. Once again this test used an 8*16 mesh, although the choice of topology is an irrelevant factor in these early tests.

Before analysing the individual components of each topology, it would be useful to examine how basic topology configurations compare against one another during the execution of the test applications. Figure 27 demonstrates the results after executing the Random application using a generic version of every topology that has been developed. It should be noted NP remains at 4, and VP is varied up a maximum value of 2048. The numbers in parenthesis on the x-axis indicate the dimensions if the topology chosen were a mesh, torus or twisted torus. In all the following tests, the network latency multiplier is given a default value of 1, and the number of cores per processor is assumed to be 1 unless otherwise stated (such that there is no hierarchical topology to consider). Finally, in all subsequent tests, unless otherwise stated, all tori

and twisted tori are completely toroidal in every dimension (their toroidal connectedness values are always [1 1 … 1]), and any twisted tori have a toroidal degree of 1, and a toroidal jump vector of [1 1 … 1].

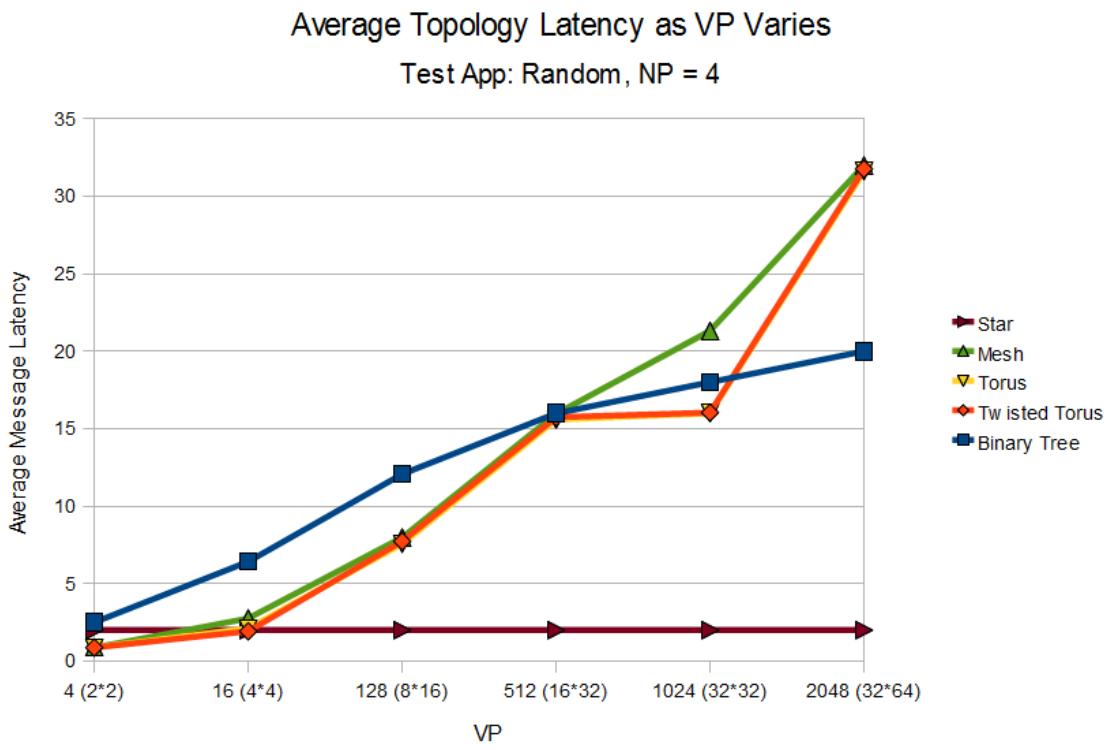

Figure 27

What is almost immediately evident is that as VP is scaled upwards, the star topology very quickly begins to outperform every other by a significant percentage. In fact, the performance of a star network will remain constant, because the latency between any source and any destination is a constant value (two times the network latency multiplier, in this case 2), regardless of the size of VP. However, despite the obvious attractiveness of this prospect, there are some limitations and considerations related to the bottleneck phenomenon, which were not within the scope of this project's implementation, but mean in reality that the star is actually a rather unfavourable choice. This, and other drawbacks, will be covered in the conclusion section.

The binary tree seems to be the best topology for coping with larger numbers of

processors. The reason for this is that every time VP is doubled, an extra layer is added to the tree, effectively adding a value of 2 to the average latency. This is evidenced in the graph where the average latency changes from roughly 18 at 1024 nodes, (10 levels) to 20 at 2048 nodes (11 levels); hence the tree would be ideal for systems with a larger processor count because of such scalability. Depending upon the application, as with the star topology, there may exist some limitations which mean the tree is an impractical solution in certain circumstances.

The mesh, torus and twisted torus all show very similar results, with one particular point of interest; when a balanced topology is being used (n*n rather than m*n), the torus and twisted torus improve in their performance, but the mesh does not. The reason for this is that in a network of unequal dimensions, the average source and destination picked at random are statistically more likely to be further from the edges, meaning the advantage of having the edges wrap around into one another becomes less significant. In a network of equal dimensions, this effect is reversed and hence in this situation the advantages of those extra links within a torus/twisted torus are evident.

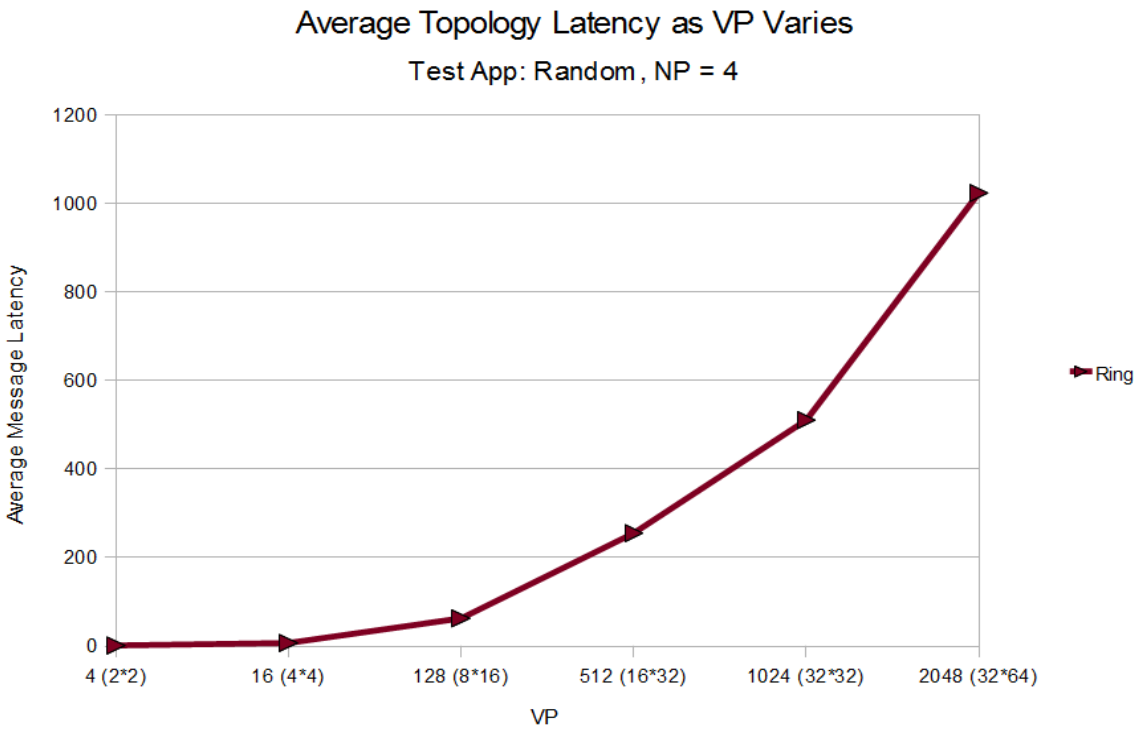

Figure 28

The performance of the ring has been included on a separate graph because of its significant difference in average latency. Although initially acceptable, the latency grows at an exponential rate. The average latency at any given point on the graph is equal to approximately half the number of nodes. This is because a randomly selected source and destination are going to require anything from a latency of 1 hops, up to a latency of n-1 hops (travelling all the way around the ring to previous node. Hence, since every selection is equally likely, this average latency will tend towards ½ of n. This is the case as can be seen in the graph where the average latency with 2048 nodes is approximately 1024, etc.

If the test application is now changed to Ring, differences can be immediately observed. Figure 29 demonstrates the new average latencies across the same VP range if the topologies are reapplied. Every MPI message that is now being transmitted is sent to the destination rank that immediately follows the source. As can be seen, the ring topology has an obvious advantage in being structured in same fashion as the

communication pattern of the Ring application, giving it an optimal average latency of 1.

The torus and twisted torus approximate the ring closely in terms of this performance. The twisted torus algorithm should in fact approximate the ring exactly (and thus have a constant average latency of 1) as it contains all of the links which are found in a ring. However, due to the minor errors found within the twisted torus implementation, this is not quite the case, and there are a few discrepancies where the average latency is slightly higher than it theoretically should be, as a result of these errors. The torus topology performs as expected; the loop around in each dimension does not take the message to the first node on the next row, but rather to the node above it, meaning that every nth message in an n*n torus will require 2 latency hops to pass it from the last element in the current row to the first, and then down to the next row. The twisted torus and ring take care of this traversal in a single link. Hence as VP and thus n grows, the significance of this difference should decrease as it becomes a less frequent occurrence when compared to the total number of messages sent.

Once again it can be seen that the star has a constant average latency of 2. The mesh starts off with an efficient average latency, however as VP increases, this value converges towards that of the star. This is because, as with the torus and twisted torus, every nth message in an n*n mesh must travel from the end of one row, to the start of the next. Since there is no toroidal link to do this quickly, the message must travel all the way back across the body of the mesh. Essentially these communications account for half of the entire latency as for every n messages, n-1 must travel a distance of 1 (hopping from neighbour to neighbour along the same row), and 1 must travel a distance of n (travelling all the way back horizontally along the row and then vertically down 1 step to the next row).

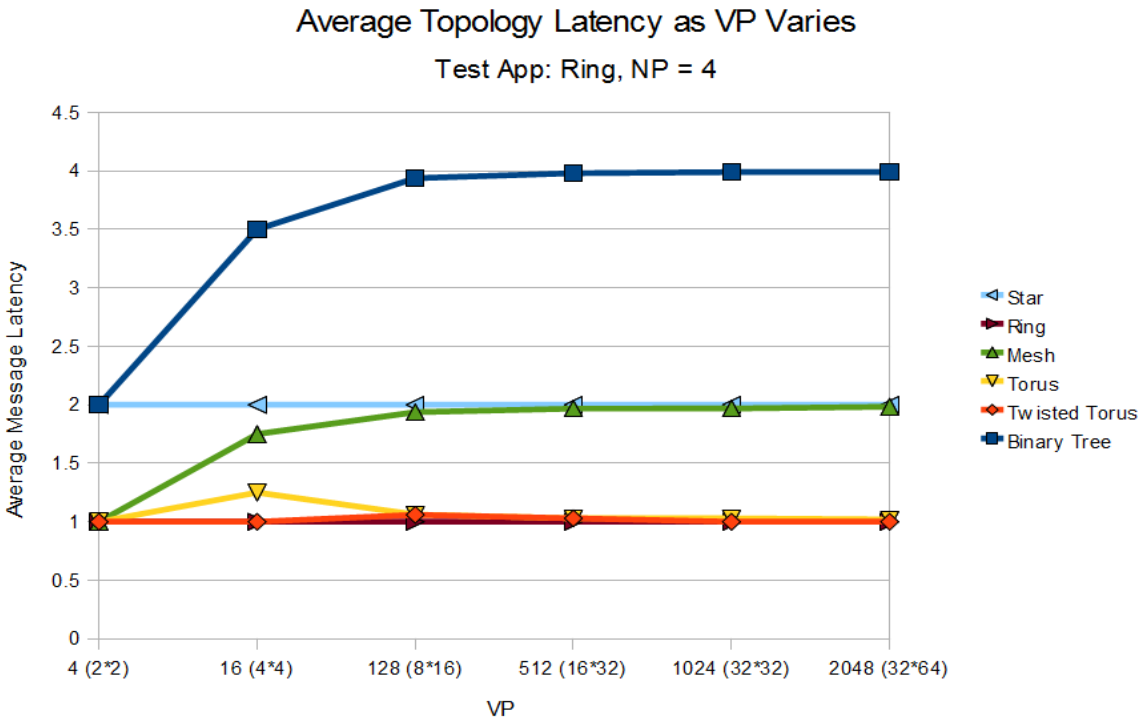

Figure 29

The binary tree converges to an average latency of 4. Consider that, since every message is being sent to its neighbour immediately to the right, then every second message passed along the tree, must only go up one link to find a common ancestor. Every other message must go up two links to find a common ancestor. As more and more nodes are added to the tree, it becomes apparent that every fourth message must go up for 3 links to find its common ancestor, every eighth message 4 links, every sixteenth message 5 links, and so on. As the size of VP increases, the frequency with which a new layer to the tree is added exponentially decreases. A binary tree of infinite size would give an average latency of 4, because the result of summing these rarer and rarer occurrences of having to travel further up the tree to find a common ancestor would exactly cancel out the number of times when only 1 step up the tree is required, leaving the mean number of steps at 2 (a latency value of 4, accounting for travelling up to the common ancestor and then back down).

4.2 Variable Tuning

Now that the general performance of the topologies has been briefly introduced and explained, it would be useful to examine how tuning their variables may affect their performance. Star and ring topologies have no such variables (other than latency which is an arbitrary multiplier for the purposes of this implementation and thus does not justify investigation). If VP is kept constant, the mesh, torus and twisted torus can be compared against varying dimension sizes to see the effect. As is evidenced in Figure 30, mesh is constantly outperformed because of the inconvenience of not having its dimensions connected or 'wrapped around'.

Torus and twisted torus perform roughly equally, although in most cases twisted torus has slightly better performance, likely a result of the advantage of dimensions smeared together resulting in traversing two dimensions in a single step in some instances. The various anomalies where twisted torus performs slightly worse can be attributed to the errors within the algorithmic implementation.

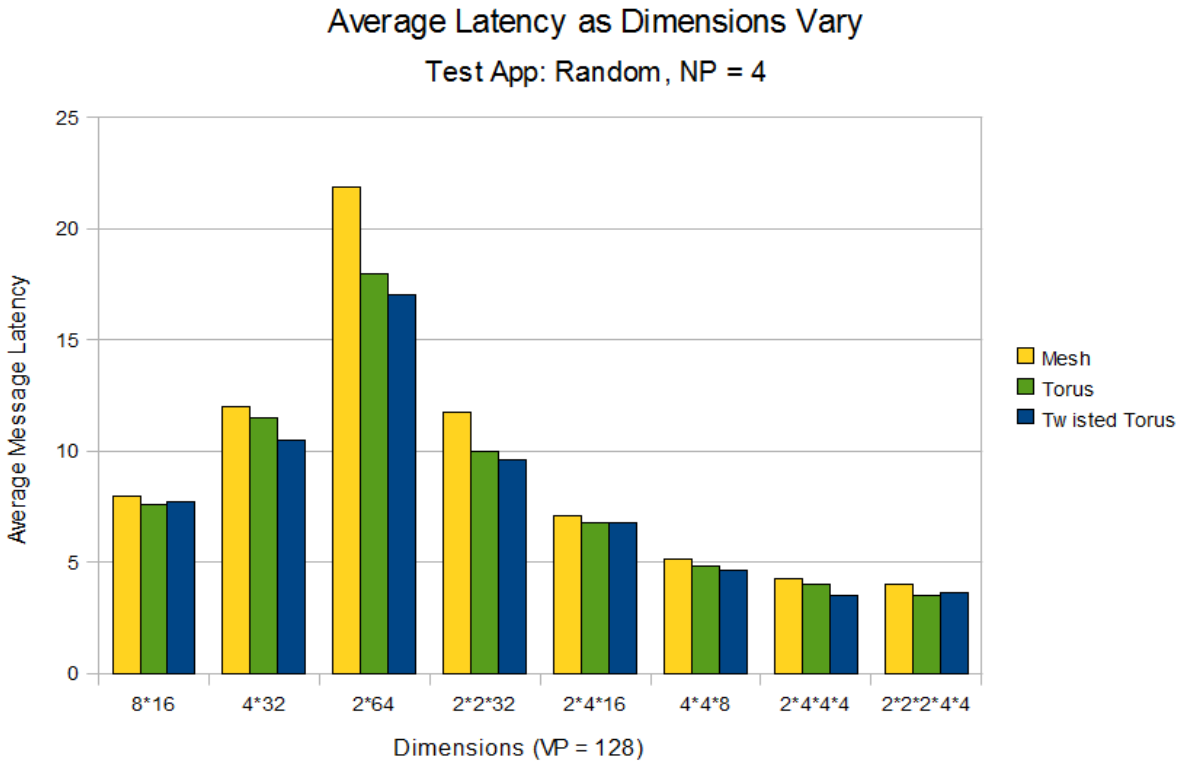

Figure 30

The variations also demonstrate that breaking a large number of nodes up into more dimensions gives better performance, however this is incidental as the higher the network degree, the more links that are required in composing the network. At higher values of VP, it would therefore become impractical to use a high degree as the number of links necessary would become very large. There is a balance to be struck between maximising performance, and minimising the number of links.

Toroidal connectedness within a torus can be analysed to more closely identify the relationship between a torus and a mesh. A toroidal connectedness value of [0 0 0] is essentially a mesh as no dimensions wrap around. A toroidal connectedness value of [1 1 1] is a completely connected torus (as has been used so far in all examples). While increasing the number of toroidal dimensions is shown to generally decrease the average message latency, this is not always the case. It is seen that making the x dimension toroidal has no effect on the average latency, a surprising result as it would

be assumed that the extra links between the first and last nodes in the x dimension would benefit those randomly selected nodes which happen to be at opposite ends of that dimension. This result then is most likely a bug with the implementation as there is no logical reason for a zero increase in performance.

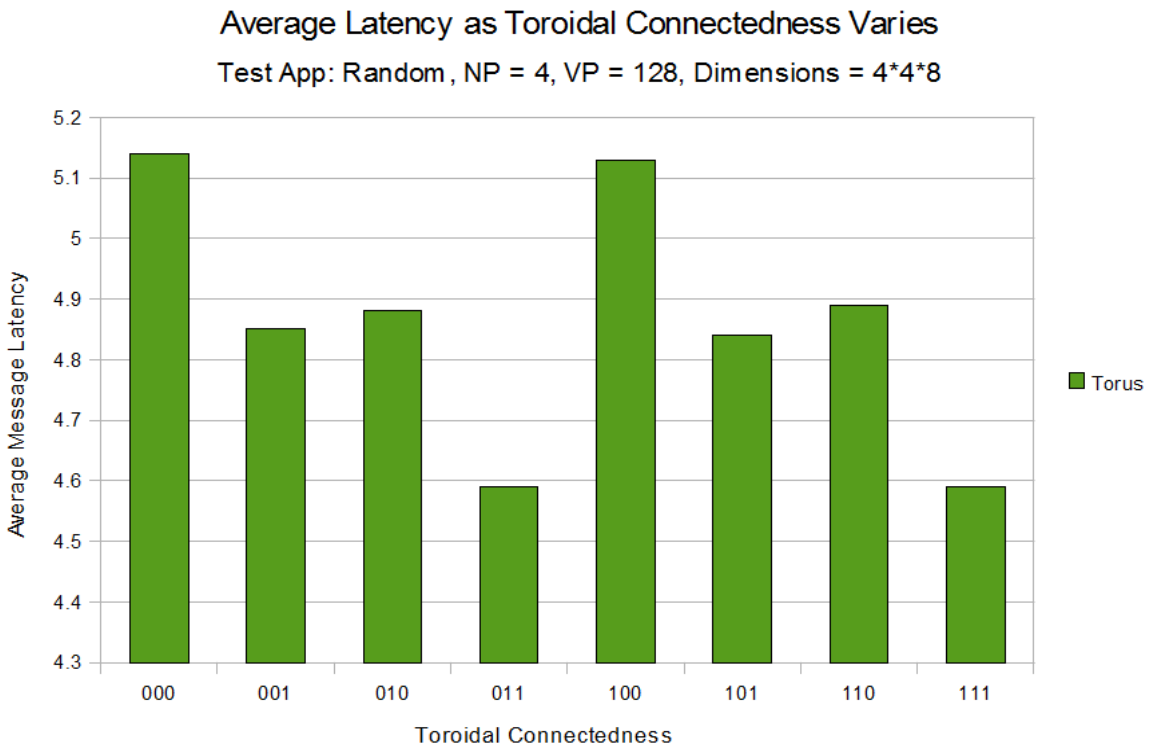

Figure 31

The other results are generally as expected; making the y dimension toroidal is advantageous to performance, but not as advantageous as doing so to the z dimension, because of the difference in dimension size. Making both the x and y dimensions toroidal evidently sums the extra performance gains of both and yields the optimum results.

The same can be done to a twisted torus to identify any similar trends that are likely to exist. There are some anomalous results here however, which can be seen in Figure 32. As expected, the twisted torus that is fully toroidal has the best performance, although there are some surprising results, such as the twisted torus with

connectedness of [0 0 1], which actually has a worse performance than that with [0 0 0]; adding these extra links to a topology should certainly not reduce performance, either an identical or increased performance should be observed. These anomalies are likely closely related to those found in the torus, and the function probably suffers from a similar bug.

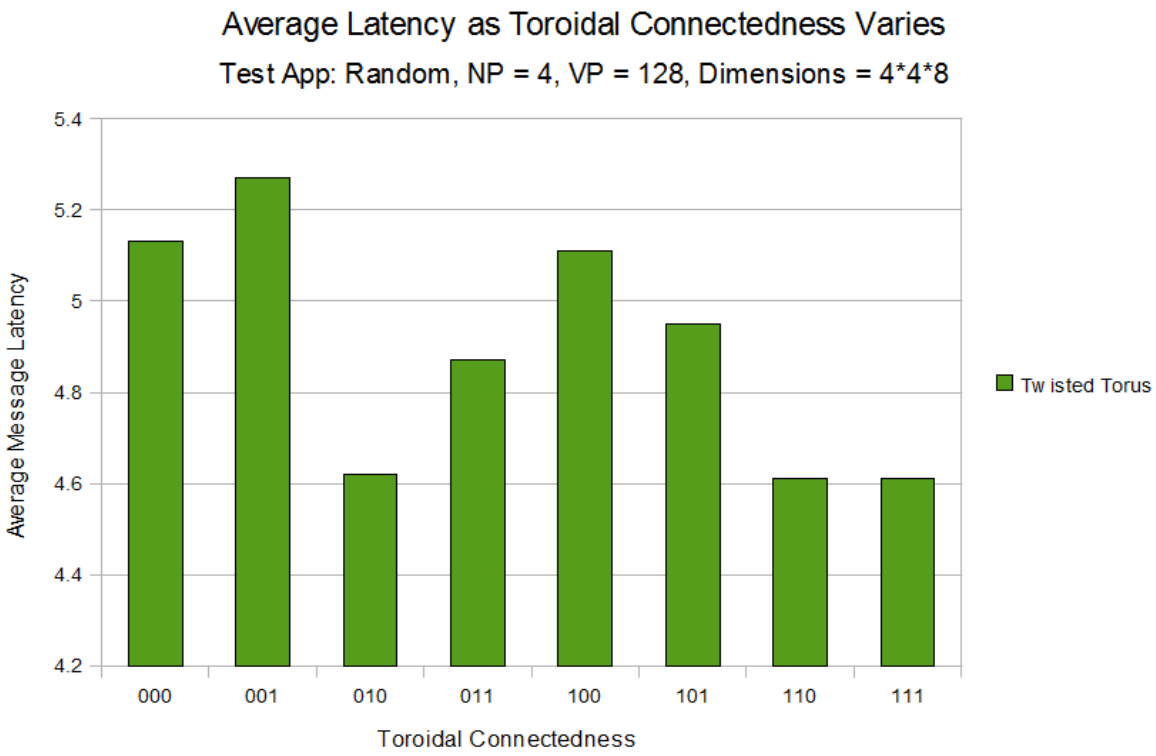

Figure 32

Toroidal degree was then varied, and tests were carried out using two different sets of dimensions. A toroidal degree of 0 defines the topology into an ordinary torus, as no additional dimension is incremented when a loop around occurs. Hence as might be expected, this yields the worst performance results. In the 4*4*8 twisted torus, a toroidal degree of 3 is the same as that of 0, as it wraps around 3 dimensions ahead of the current one; since there are only 3 dimensions, that sends it back to itself. It was intended that error handling be implemented to ensure that the user could not specify a toroidal degree greater than or equal to the topology degree, and the graph demonstrably shows that there are erroneous results in those instances. In the 4*4*8

network, the toroidal degree = 3 performance shows a similar latency to that of toroidal degree = 0 as expected. However, as should similarly happen, the toroidal degree = 4 performance does not show a similar latency to that of toroidal degree = 1.

Ignoring these results and examining the meaningful data, namely where toroidal degree is between 0-2 in the case of the network of degree 3, the only notable thing to be extrapolated is that varying the toroidal degree seems to have little to no effect on the average message latency within the network.

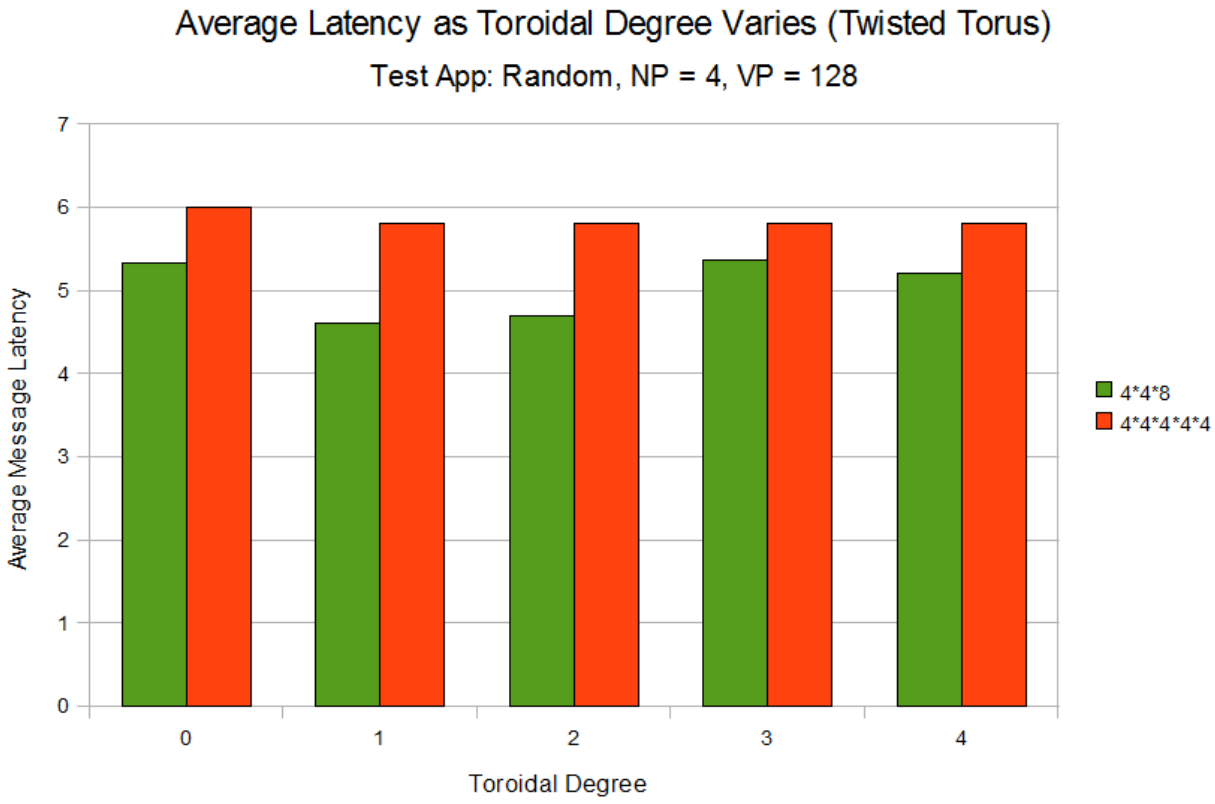

Figure 33

The final twisted torus parameter not yet examined is the toroidal jump vector. The following graphs can be analysed to see how the average latency varies. Figure 34 shows that there are a few performance changes observable by altering the toroidal jump, although the number of tests done is too small to gain any insight into what exactly is causing the various improvements. It would seem that altering the toroidal

jump in different dimensions does produce variously different results, some improved and some worsened. Figure 34 therefore was an attempt to try and look a bit more in-depth at any relationships that might exist.

To lower the number of factors involved and attempt to simplify things, the network degree was reduced to 2, but VP was increased to create a 16*16 twisted torus. The optimum performance can be found, when either one or both of the dimensions has a toroidal jump value of 8, which is exactly half the size of the dimension. As the toroidal jump values start to move further away from 8 on either side, the performance decreases. This can be explained by the fact that the toroidal jump essentially acts as a 'dimensional short-cut' to another part of the network. If this short-cut happens to take a message to a the opposite side of the network, it means that the average message will have fewer links to traverse to reach its destination.

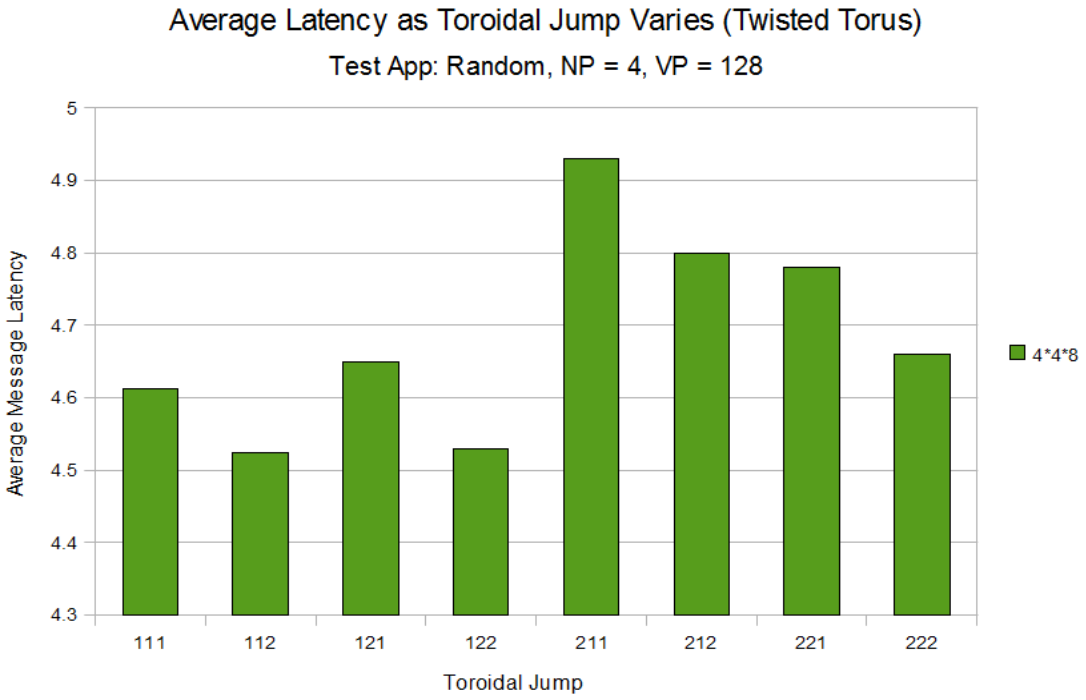

Figure 34

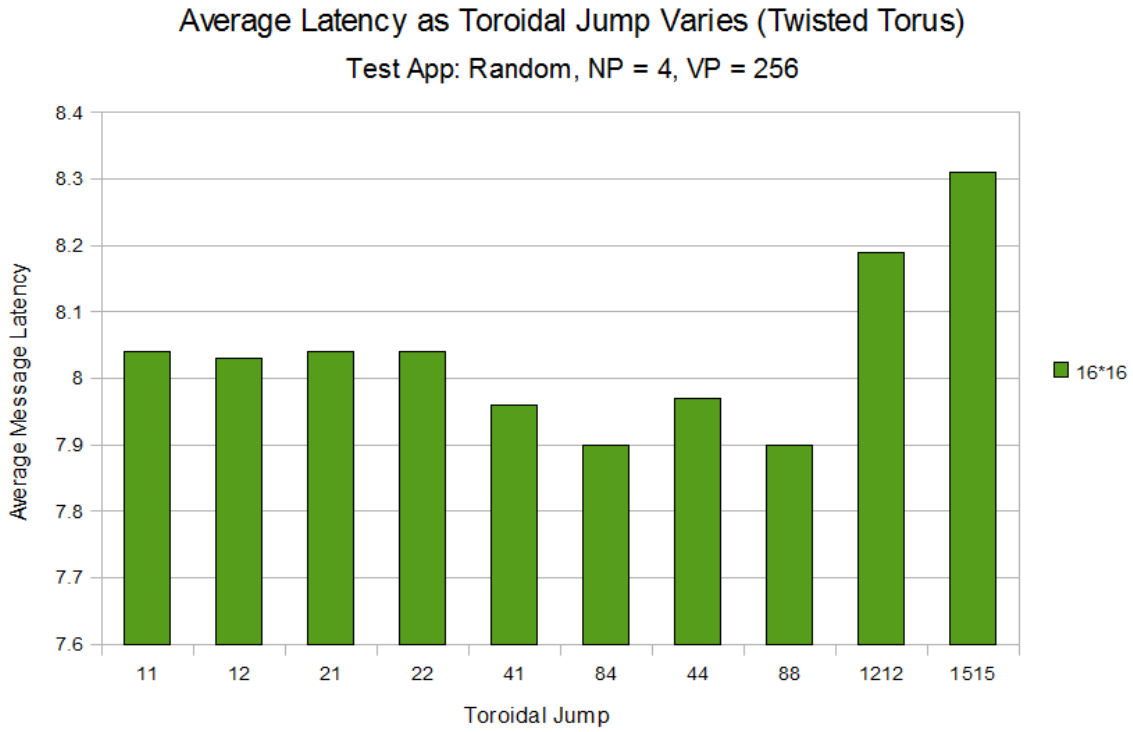

Figure 35

Finally the tree can be examined and its only unique variable, degree, can be analysed. The degree of the tree indicates how many children each node has. Higher degree trees tend to perform better, however they also suffer from bottleneck issues covered in the conclusion section of this report, which mean they may not be entirely practical; a balance is necessary. As degree increases, the performance improvement decreases because, in a very similar way in which binary tree performance converged to 4 on the ring as VP varied, the difference in the number of levels on the tree when increasing the degree begins to thin out.

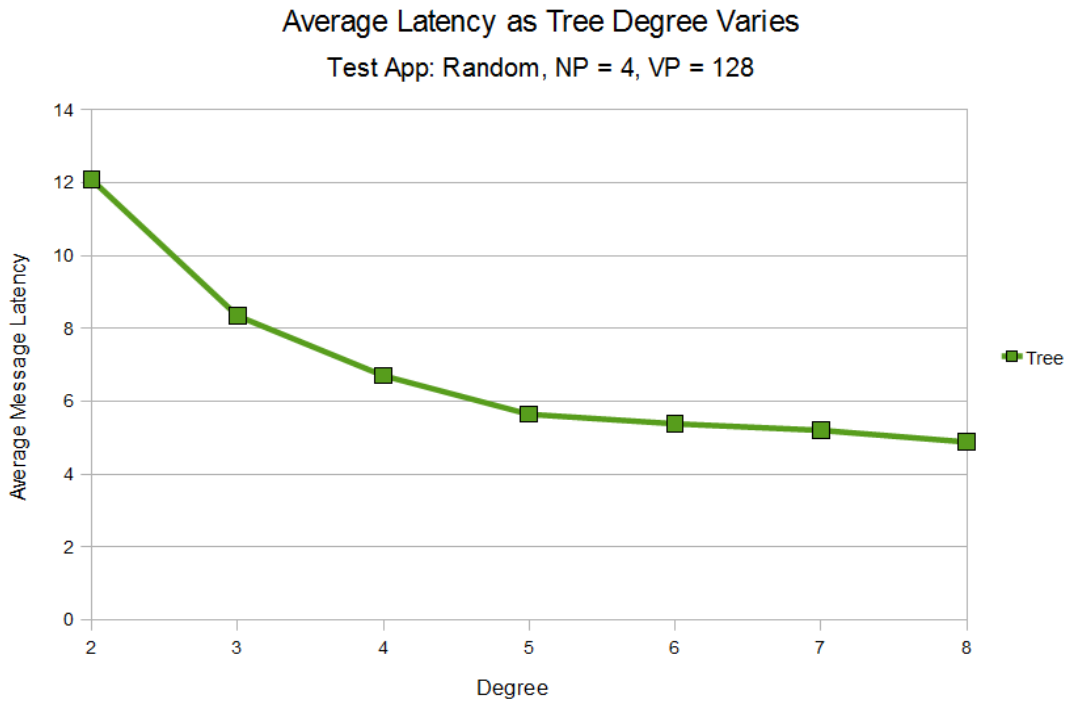

Figure 36

The final parameter to adjust is bandwidth. Whilst not strictly "tuning" because obviously larger bandwidth desirable in all cases, it is interesting to note the change as bandwidth is decreased such it starts to have an impact on the latency. Throughout all other tests, network bandwidth (and later, processor bandwidth) has been kept at a constant value of 1 Mbps. Due to the nature of the applications being tested, this is always large enough such that it has no bearing on the latency result. This is important because bandwidth is, for the purposes of this implementation, an unrelated consideration which is taken into account after the network latency has been calculated.

As Figure 37 shows, lowering the bandwidth starts to have an impact on the latency at around 0.00001 Mbps, indicating that most of the MPI messages are of a relatively small size. As the bandwidth is lowered further and further, messages take longer to pass through the network, eventually every communication becoming a bottleneck in the simulator. Evidently by the time bandwidth has been lowered to 0.0000001 Mbps, it is the dominating factor in latency calculation, as opposed to the actual message path. The bandwidth was only tested with a mesh network. It may be

interesting to examine other networks and the effect of reducing bandwidth, however as the network topology type is unrelated to bandwidth, it is almost certain the results would be similar, with bandwidth becoming predominant at the same figure.

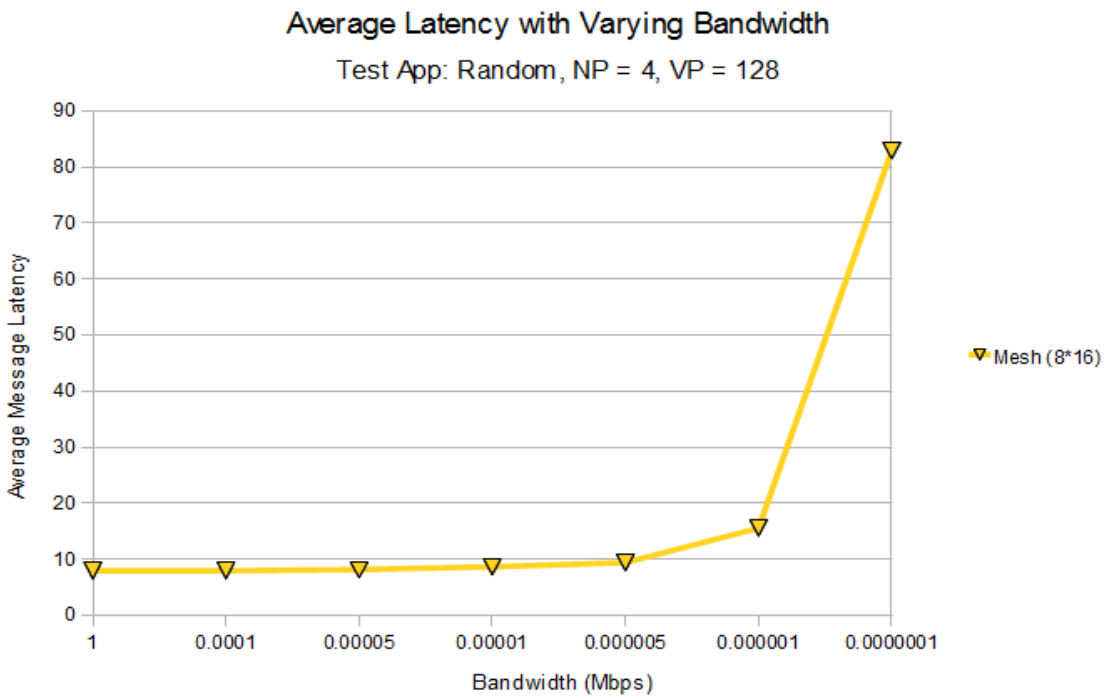

Figure 37

4.3 Hybrid Topologies

The final phase of testing is to examine hybrid topologies; that is where the number of cores per processor is greater than one, and topologies can be mixed and matched together such that the processors are using topologies which are different to that of the network. The first tests involved using nested topologies which were the same, e.g.: a mesh within a mesh, or a tree within a tree. Generally the differences seen would be expected to be similar to that of a single topology simulation, such as the results shown earlier in Figure 27. For the following tests, whilst the network latency multiplier remains 1, the processor latency multiplier has been sent to 0.1, as it would be expected that messages would travel faster from core to core, than from processor to processor.

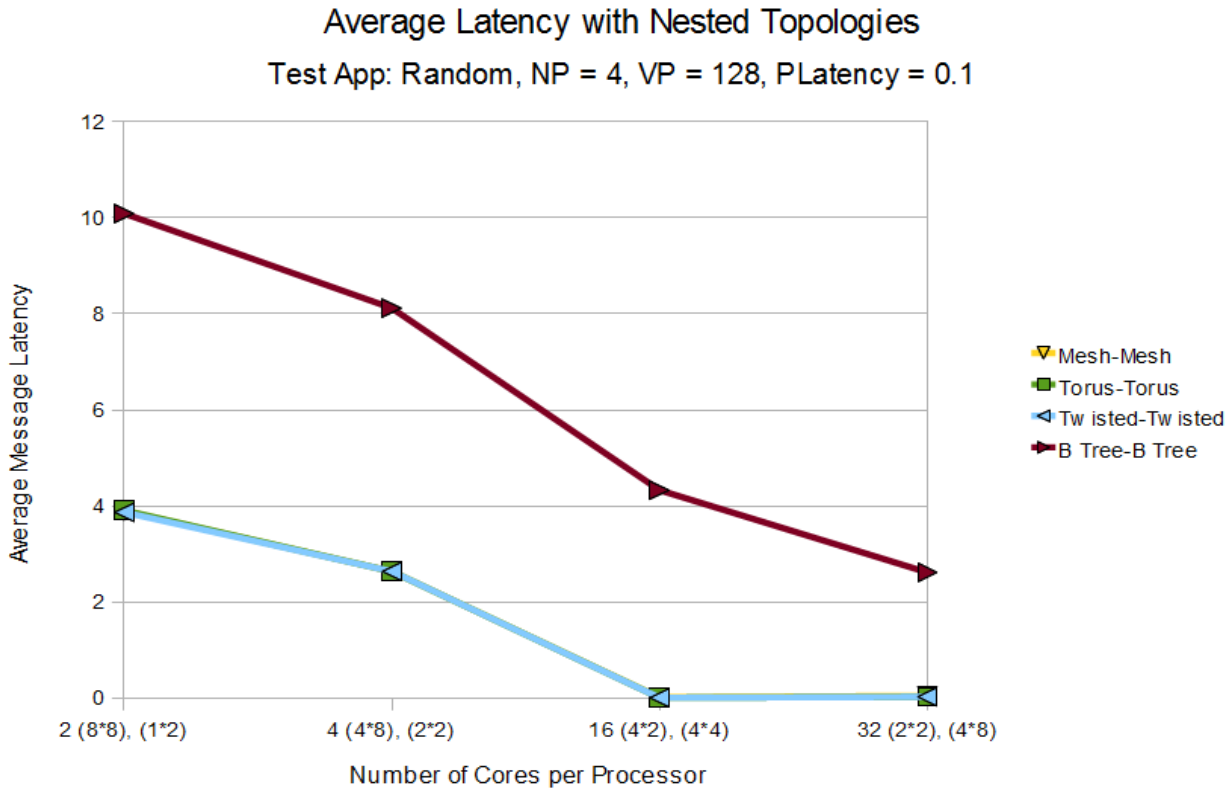

Figure 38

It should be noted that the x axis lists the number of cores per processor, and

the two sets of parentheses indicate firstly, the dimensions of the network, and secondly, the dimensions of each processor. The mesh, torus and twisted torus have almost identical sets of results; as the number of processors is reduced, and the number of cores per processor is thus increased, communication takes place predominantly within each processor, and thus the average latency converges to the processor latency multiplier, 0.1. The tree performs the worst of all the topologies here, which is expected considering that VP = 128, as seen in Figure 27. However, the effect is worsened here, because VP is divided into processors which are then divided into cores; when VP is large the tree has an advantage because adding a new layer to the tree becomes more and more infrequent as was discussed previously. Since VP now consists of smaller trees, rather than one larger tree, it would take a larger VP count before the tree's advantage began to make an impact.

The following graphs show some of the tests which were undertaken involving the mixing of different topologies. This was not done extensively, as an entire project may be dedicated to such a task, analysing the various topological combinations and their results, but some meaning can nevertheless be extracted from the data at hand. One point of interest is that when the processor topology is a mesh, torus or twisted torus there tends to be almost no difference in performance. With a processor latency multiplier of 0.1, the differences that do exist in the data are too small to be seen on the graphs as displayed.

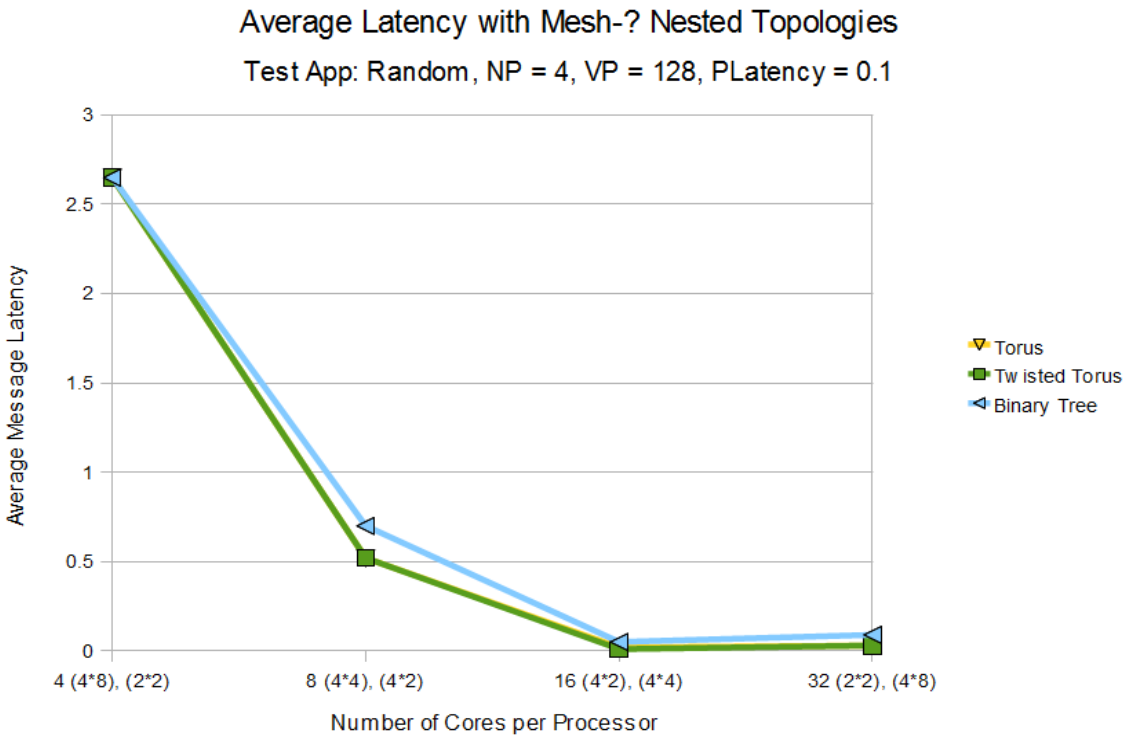

Figure 39

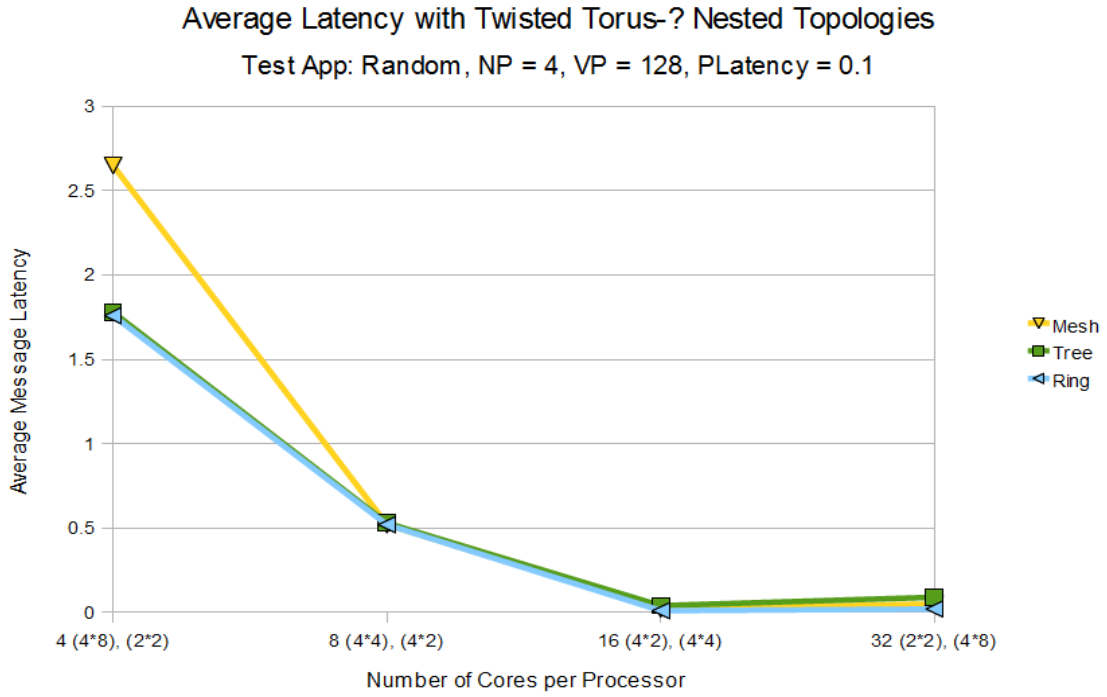

Figure 40

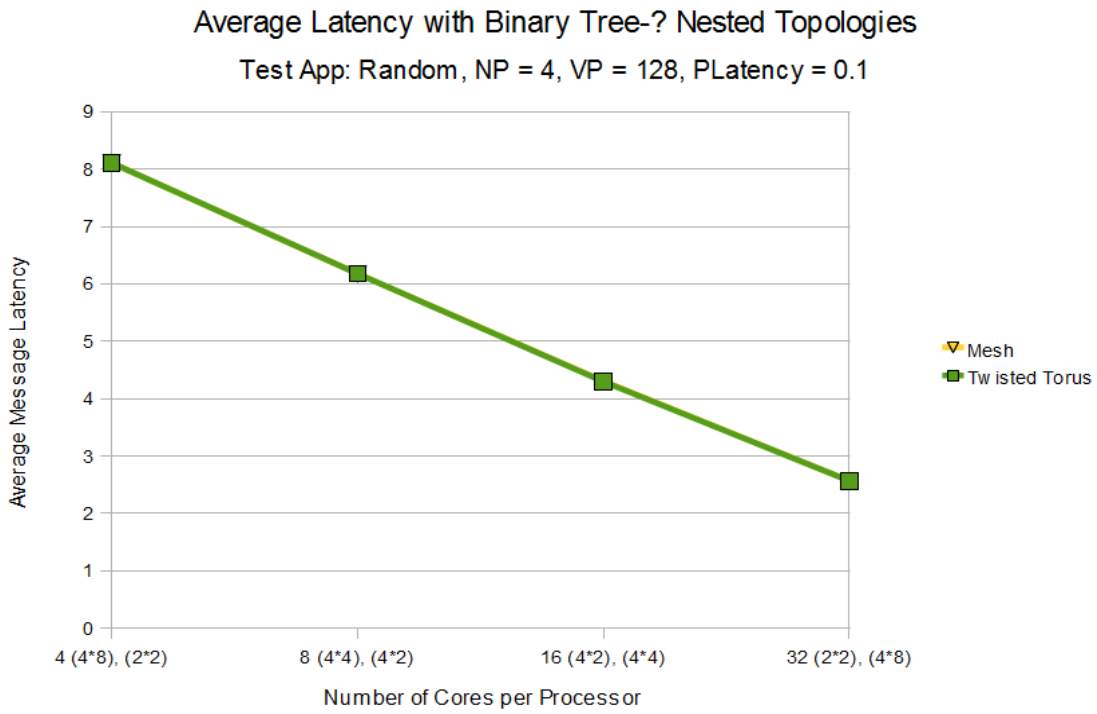

Figure 41

There is clearly an optimum split between mesh or twisted torus, and tree topologies as can be seen in Figures 38 and 39; although on both graphs they look as though they will converge to 0.1, when the processor dimensions become large and the network dimensions become small, the performance actually decreases. The reason for this is difficult to accurately speculate upon. The most likely reason is that the tree is under-performing in relation to the alternate topology, so when the number of cores per processor grows, the tree becomes the predominant network used in communication and as seen previously, the tree performs worse than the . However it must also be considered that the The exact point of this optimum will be variable depending on the defined ratio of the network and processor latency multipliers, in this case a ratio of 10:1. However by adjusting this figure the optimum point of balance between the topologies would shift.

Given more time, it would be beneficial to study how various topological combinations perform when other values are scaled such as VP, and how this affects

parameter tuning explored in the previous section. Additionally some of the results are unexpected, for example the twisted torus-ring topology mix seems to converge to the optimum of 0.1 as the number of cores per processor grows. However, as seen in earlier tests, this is most certainly not would be expected of a ring; generally the ring's performance degrades linearly as the number of cores increases.

5. Conclusions

5.1 Limitations and Critique

There are a few limitations to the project which need to be addressed in a future version of the simulator. Most of these limitations either arose at, or around, the end of the project and thus correcting these problems was difficult to do in the remaining time. Some of these are simple to fix and others may require a more in depth evaluation of the network structure and possibly some radical changes.

The first issue is with the twisted torus function. It would be fair to say that constructing this function was a task which took up a decent percentage of the total time spent working on the simulator. Although the topology was tested as it was being developed to ensure that the mathematics and logic behind the processes were sound, there are a few bugs which still exist when calculating the fastest route between specific nodes within a twisted torus. These bugs only occur in situations where nodes on the opposite sides of the network are attempting to communicate by looping around in multiple dimensions (traversing the link between the first and last nodes in the primary dimension), and furthermore the bugs only appear to occur in twisted tori of specific dimension values, and not others.

The nature of these errors tends arise due to the way in which the quickest route is analysed. In each phase of the function, when all remaining dimensions are tested culate the traversal cost, it is not accounted for that traversing one dimension to end up in a position which is actually further away from the destination, may in fact lead to the discovery of the shortest path, due to the smearing effect and some subsequent “short-cut”. It should be noted that when the cost of each dimension traversal is calculated, the network function only takes into account the physical differences in the x, y, z, etc., values of each dimension. At this stage, the algorithm has no knowledge of the twists that exist within the network, which surely plays a part in the erroneous data

sometimes obtained. Due to this, the twisted torus function is not always entirely accurate in its calculations, and leads to some of the anomalies which were seen during the testing section, and these are circumstantial. The algorithm is probably not the most efficient either, as when the number of dimensions, n, increases, the number of tests which must be carried out rises at the rate of (2*n!) which can become very large very quickly.

Additionally there were some instances during the testing phase in which a deadlock situation was reached where the simulator effectively froze up because various virtual nodes were each waiting on one another for messages with lower time stamps to arrive than any of them currently had (using an optimistic PDES would avoid this issue). Curiously, this would sometimes only happen with specific configurations of network parameters. Furthermore, some parameter configurations would execute to completion on some occasions, but not on others. And other configurations would run into deadlock every single time. These problems lay outside the scope of this project, as they are a feature of the underlying MPI functionality which was implemented before the project was started. This part of the simulator will need to be adjusted to account for these deadlock bugs. This could have been done as part of this project if there had been time.

Further issues arise when considering elements of the implementation which were disregarded for the sake of simplicity, but which are ultimately necessary in the future, if the simulator is to be useful when simulating real networks. One such issue is that the total number of links within a network is not considered. It might be all very well to link every node to every other node within a network and thus minimise the shortest path between nodes (it would thus always be 1 times the latency multiplier; half the time of a star network), however, this would be a massive waste of resources and highly impractical in all but the rarest of situations. As the size of this network is scaled up to that of supercomputer proportions, the number of links needed grows at an incredible rate and the overwhelming majority of these links will be in a state of disuse at all times.

A further element of real world networking which was not implemented is variation. All links in the network were assumed to have identical bandwidth, and all links were assumed to have equal latency multiplier values. The same assumption was made about processor bandwidth and processor latency multiplier values. In actuality, it may be the case that different links within a network have different bandwidth capacities, and some links, for various reasons, may take longer to traverse than others. If this were to be considered in a future version of the simulator, it would probably be that a default value for both bandwidth and latency-multiplier would be specified, as is the case now, and then any subsequent variations would be specified individually, including the specific link ID, along with the value of the specific bandwidth and the specific latency-multiplier which that link possesses.

Likewise, processors connected together in a single network may have different internal topologies to one another; for example, some may have their cores organised in tree structures, and others arranged into a mesh or torus. As before, a default topology could be specified for the majority of processors, and those with different topologies to that might then be defined with individual parameters.

Testing is an area which requires more in depth-analysis. Due to the limited time-frame of the project, it was impossible to test all combinations of data. Some key elements that need to be looked at include the performance of the simulator when using other test applications, such as Heat Transfer and Pi. These applications utilise different communication patterns and it would be useful to see how the various topologies cope with each of these as they are perhaps more closely representative of some of the common communication patterns that would be seen in real world applications. Also, as mentioned in the testing section, it would be ideal to more rigorously test the hybrid topology configurations, and then test how they perform when varying the parameters explored in the fine-tuning section, as well as investigating the results obtained when using a much larger value for VP, such as that used in the very first test.

One part of the network model which was never included when the simulator was converted from C++ to C is parameter validation. In the present working version of the simulator, it is therefore very easy for the user to specify invalid arguments and then receive error messages when attempting to execute an application. In the C++ version, as explained at the start of the implementation section, parameters were checked against each other to ensure integrity, such that for example, the dimensions specified in a mesh, had to equal the virtual processor count, VP. This would not be too difficult to add to the C version, although the code would have to be modified slightly because of some C++ exclusive commands and functionality, hence the reason it wasn't simply copied across.

For testing purposes, the simulator was briefly edited to output a few performance metrics. A global variable was used to keep track of the number of times that the latency of a message had to be calculated. This was incremented every time the apply function was called, and another global variable was used to sum the latency values which were returned. By dividing the sum of the latency values by the total message count, the average message latency could be calculated, which is the figure that was then used in the majority of the tests for the y-axis. These variables were then simply printed to the standard output just before the simulator terminated. This functionality is no longer in the simulator, although re-adding this to the code would take a matter of minutes if desired.

In regards to the project's objectives, the minimum requirements were achieved, although the expected requirements were not; network latency was simulated at least to a certain extent, but fault injection wasn't. The main reasons for this were the unforeseen complications encountered when creating the functionality for calculating the shortest message path, specifically for the twisted torus. As a result of this, the plans to incorporate a few extra network topologies were also not achieved. Earlier in the design section it was proposed to possibly implement the dragonfly topology, which itself is used in the Jaguar supercomputer at ORNL. This should be a high

priority for future versions of XSIM, as the dragonfly is a very common topology in the field of supercomputing and its inclusion in the simulator would be an invaluable asset to future research.

The problems in the twisted torus were not entirely resolved by the completion of the project. As mentioned in the testing section, there exist some anomalies when calculating the shortest path in certain circumstances. The algorithm for calculating the shortest path may therefore need to be reassessed or redesigned. However, the current algorithm operates as a good approximation in most cases and so will suffice in the short term. One such problem at an earlier stage in the implementation was when considering whether to wrap around in a dimension or not. Some nested if-statements are used to decide, when a wrap-around is tested, whether the so-called “secondary” affected dimension (see the implementation section) should be incremented by the dimensional jump value, or decremented by this value.

Since messages can travel in both directions along a given dimension, this varies depending upon whether or not the source co-ordinate for the primary dimension is greater than or less than the destination co-ordinate for the primary dimension. For example, suppose there exists a 3*3 twisted torus, where the source is [2,0] and the destination is [3,3]. Testing the x dimension, the wrap-around would have the effect of decrementing the secondary dimension (the y dimension), because the x value in the source is lower than that in the destination, and so the 'direct' traversal would go forward from 2->3, however the 'wrap-around' would go down from 2->0, loop around, and reach 3. As noted in Figure 22 for example, decrementing the x dimension to 0, and following the wrap-around link at the top, actually decrements the y dimension (although this example is a 3*3 torus, the same concept applies) However, once this is decremented by the toroidal jump value (be that 1 or greater), then it has to be accounted for the fact that the y dimension is already at 0, (y = 0) and so must flip back to the other side of the network (y = 2 or less). Figure 22 demonstrates this with the link between [0,0] and [3,3].

Although these problems seem trivial in hindsight, when first encountering the implementation of a twisted torus it becomes fairly problematic to envision exactly how traversing dimensions which are smeared together will work in all cases. There does not exist a great deal of readily accessible material on the subject of traversing twisted tori, however it was recognised that there almost definitely exists an algorithm out there which perfectly traverses twisted tori in an optimum fashion, and it would be beneficial to incorporate this into the simulator in a future upgrade. The current implementation, as with all other topology traversing functions, was implemented without external help.

5.2 Future Work

One piece of functionality which was considered for implementation towards the end of the project is the simulation of overlay networks. Ultimately this idea was never followed through to completion. The basic concept is to specify the physical topological structure of the nodes within a network/processor but then to incorporate the ability to place a 'virtual' network over this underlying topology, and force the nodes to communicate as if the overlay network was real. Thus, the mapping of the overlay network would need to be translated onto the underlying network, and the latency cost calculated for traversing the underlying network according to the rules of the overlay network. One such overlay network which was investigated was a binomial tree, which is optimal in situations requiring messages to be broadcast. However, this was not done, as there is currently no support in XSIM for MPI Broadcast procedures.

Possibly the biggest requirement in the current implementation, specifically pertaining to the newly developed network function, is for some consideration for network traffic. Since the latency is calculated as a mathematical cost rather than an actual virtual path through the network, the nodes and links which a message passes through as it goes from source to destination are not listed or identified. As soon as network traffic is taken into account it becomes necessary to determine the exact path which each message takes through the network. One way of doing this would be to restructure the entire network model such that it uses the data structure approach which was discussed in the design section. In this way, every link that a message passes through can be recorded, and held in a database, which then expires after a given period of time.

This information can be used to determine how 'busy' a particular link, node or router is at any given time, and if bandwidth is factored into the equation, a penalty can be applied to subsequent messages which traverse those links which are already fully saturated. It is for this reason that the star and tree networks in particular, were highlighted as possibly being impractical solutions in reality in some parts of the

testing section. In a star network, every message in the network must pass through the central router, which will quickly become overloaded and congested. So, once traffic hotspots and congestion are taken into account, it may be the case that the star network is very impractical for all but the smallest networks.

Similarly, although increasing the degree of a tree network was seen to increase the performance in terms of latency, it ultimately may suffer from the same issues, as the routing nodes begin to get very quickly outnumbered by the processing nodes at a faster rate as VP increases, and each routing node 'parent' becomes responsible for more processing nodes, and handling all messages sent to and from those nodes. As the degree increases, parents become bottlenecks in the same fashion as in the star network, although the effect is considerably dampened due to the divide and conquer structure of a tree. In some applications this may not be an issue, however in many there will be some implication which becomes apparent as the network is scaled.

Considering the alternate data structure approach to network latency once more; if this were implemented, and a particular link was saturated to an unacceptable extent, the message may then be re-routed through the network. Depending upon the topology of the network and the specific ranks of the source and destination nodes, this new route may have an equally short path, or it may be longer. No doubt some function could be written which determines the next shortest-path given the fact that a particular link (or links) is marked as unavailable. The latency penalty of taking this alternate route could then be weighed up against the latency penalty of queueing to traverse the congested route, and the result which has the lowest latency could then be taken.

If fault injection were implemented, it may have again been beneficial to use the data structure approach to designing the network model, discussed in the design section, rather than the mathematical one. In addition to the database suggested for holding messages which are currently traversing a given link, a secondary database could be created for maintaining a list of links which are unavailable. The user might then be able to specify if a given link fails by indicating the nodes which the link

connects, and this would then be entered into the database. They would also have the ability to indicate if a previously failed link is then rebooted and available for use again, removing the entry from the database. Similarly, a function could be written as part of the network class, which is fed in a variety of parameters by the user, namely the failure rate, and then this could be used to autonomously generate random link failures either throughout the entire network, or in a specific area of the network.

Taking this idea further still, it could be programmed such that links which are more congested are more likely to fail, increasing the probability of that link failing over any others by a scaled percentage relative to the amount of traffic experienced. Nodes could also be programmed to fail (again, either determined by the user or randomly generated), and then any links which connect to that node would have to be found and included in the database of unavailable links. If a message is then being passed from node to node en route to the destination, a function would then be made that checks that each and every link or node which is passed through is not currently unavailable. In the same way that message routes are recalculated if a link is over-congested, the message would be directed down an alternate path (if one exists) to avoid the failed links/nodes.

Of course it might be that the amount and locations of multiple failures effectively splits the network into two or more disjoint parts, each unable to communicate with the other. This would need to be accounted for. It might be that such a situation would be prevented from ever occurring by using an algorithm which performs some check on any new failures. Or the entire network may freeze, messages would pile up on either side of the divide, and then when one of the links is restored, the network traffic begins to flow again. It would be interesting to see how the different applications and different topologies cope with fault injection.

The simulator is still very much a work in progress, there are many elements which can be added to increase the realism and accuracy in representing a real supercomputer, and hence the reason why there are various other simulators which

exist, each one different and unique, as they try to focus on certain aspects of the process of simulation, depending upon the desired use. Future areas to develop in XSIM may include, but are not limited to, the ability to handle broadcast messages, a greater number of supported MPI procedures, an optimistic PDES implementation, complete with roll-back, and the output of performance metrics.

6. Bibliography

6.1 References

1) Encyclopaedia Britannica Supercomputer Article, Encyclopaedia Britannica Online – http://www.britannica.com/EBchecked/topic/574200/supercomputer, 2010

2) ENIAC – The Press Conference that Shook the World, Dr. C. Dianne Martin, 1995

3) Getting up to Speed – The Future of Supercomputing, Susan L Graham, 2004

4) A Brief History of Supercomputing: “the Crays”, Clusters and Beowulfs, Centers. What Next?, Microsoft Research, Gordon Bell, 2008

5) ENIAC – The Army Sponsored Revolution, William T. Moye, 1996

6) Hybrid and Manycore Architectures, Jeff Broughton, 2010

7) The Linpack Benchmark, Top500.org – http://www.top500.org/project/linpack, 2010

8) Overview of Supercomputers, Matt Lockwood and Mohammad Haghkar, 2010

9) Charm, Parallel Programming Laboratory, Department of Computer Science, University of Illinois - http://charm.cs.uiuc.edu/research/charm/, 2010

10) The Cray 2 Computer System, Cray Research Inc. - http://archive.computerhistory.org/resources/text/Cray/Cray.Cray2.1985.102646185.pdf, 2010

11) CDC6600, Fact Index, http://www.fact-index.com/c/cd/cdc_6600.html, 2010

12) ENIAC – The World's First Supercomputer, associatedcontent.com, http://www.associatedcontent.com/article/376650/eniac_the_worlds_first_super_computer.html?cat=15, 2010

13) World's First Teraflop Computer Decomissioned, Teraflop News, http://www.physorg.com/news70844486.html, 2010

14) ENIAC, Ron Avery – http://www.ushistory.org/oddities/eniac.htm, 2010

15) A Brief History of Supercomputers, Jan Matlis - http://www.cio.com.au/article/132504/brief_history_supercomputers/, 2010

16) Parallel Programming in C with MPI and OpenMP, Michael J. Quinn, 2010

17) Projected Performance Development, Top500.org – http://www.sausageandorangeslicepizza.com/index.php?/archives/64-Supercomputers-Perpetual-Motion-Machine.html, 2010

18) Time-line of Supercomputers, Lawrence Livermore National Laboratory - https://www.llnl.gov/str/June03/gifs/McCoy1.jpg, 2010

19) Supercomputers With 100 Million Cores Coming by 2018, Patrick Thibodeau - http://www.computerworld.com/s/article/9140928/Supercomputers_with_100_million_cores_coming_by_2018, 2009

20) μπ, Kalyan Perumalla – http://kalper.net/kp/software/mupi/index.php, 2009

21) Charm Manual, Department of Computer Science, University of Illinois - http://charm.cs.uiuc.edu/manuals/html/bigsim/manual.html, 2010

22) Charm AMPI, Department of Computer Science, University of Illinois - http://charm.cs.uiuc.edu/research/ampi/, 2010

23) Parallel Discrete Event Simulation, Richard M. Fujimoto, 1990

24) Simulation of Advanced Large-Scale HPC Architectures, Frank Lauer, 2010

25) Super-Scalable Algorithms for Computing on 100,000 Processors, Christian Engelmann and Al Geist, 2005

26) MPI: The Complete Reference 2E, Jack Dongarra et al. - http://www.netlib.org/utk/papers/mpi-book/node1.html, 2001

7. Appendices

7.1 Source Code

7.1.1 *xsim_nm.h*

```
/*************************************************************************//**
 *
 * @file xsim_nm.h
 *
 * Header file for the xsim library network model (NM) module.
 * Copyright (c) 2009-2010 Oak Ridge National Laboratory.
 *
 * For more information see the following files in the source distribution top-
 * level directory or package data directory (usually /usr/local/share/package):
 *
 * - README    for general package information.
 * - INSTALL   for package install information.
 * - COPYING   for package license information and copying conditions.
 * - AUTHORS   for package authors information.
 * - ChangeLog for package changes information.
 *
 *************************************************************************/

/* Avoid to include the content of this header file twice. */
#ifndef XSIM_NM_H
#define XSIM_NM_H

/*************************************************************************
 *
 * Macros
 *
 *************************************************************************/

/*************************************************************************
 *
 * Includes
 *
 *************************************************************************/

/*************************************************************************
 *
 * Data Types
 *
 *************************************************************************/

/** The network type data type. */
typedef enum xsim_nm_type_e {
  XSIM_NM_STAR = 0,           /**< The star network type. */
```

```
  XSIM_NM_RING = 1,           /**< The ring network type. */
  XSIM_NM_MESH = 2,           /**< The mesh network. type. */
  XSIM_NM_TORUS = 3,          /**< The torus network. type. */
  XSIM_NM_TWISTED_TORUS = 4,  /**< The twisted torus network type. */
  XSIM_NM_TREE = 5,           /**< The tree network type. */
} xsim_nm_type_t;             /**< The network type data type. */

/** The network model data type. */
typedef struct xsim_nm_s {
  xsim_nm_type_t type;              /**< The network type. */
  double    latency;          /**< The network latency in microseconds. */
  double          bandwidth;        /**< The network bandwidth in Mbps. */
  unsigned int    degree;           /**< The network degree. */
  unsigned int    t_degree;         /**< The network toroidal degree. */
  unsigned int    *t_connectedness; /**< The network toroidal connectedness. */
  unsigned int    *t_jump;          /**< The network toroidal jump. */
  unsigned int    *dimensions;      /**< The network dimensions. */
} xsim_nm_t;                        /**< The network model data type. */

/*****************************************************************************
 *
 * Function Prototypes
 *
 *****************************************************************************/

/**
 * Initializes the network model module.
 *
 * @param  nmcfg  The network model command line configuration argument (IN).
 * @return        MPI_SUCCESS for success, or MPI_ERR_OTHER for error with
 *                errno set appropriately.
 */
int xsim_nm_init (char *nmcfg);

/**
 * Finalizes the network model module.
 *
 * @return  MPI_SUCCESS for success, or MPI_ERR_OTHER for error with errno set
 *          appropriately.
 */
int xsim_nm_fini ();

/**
 * Applies the network model to the receive time of a point-to-point message.
 *
 * @param  source  The source rank in MPI_COMM_WORLD (IN).
 * @param  dest    The destination rank in MPI_COMM_WORLD (IN).
 * @param  bytes   The buffer byte count (IN).
 * @param  send    The send time in microseconds (IN).
 * @param  recv    The receive time in microseconds (OUT).
 * @return         MPI_SUCCESS for success, or MPI_ERR_OTHER for error with
 *                 errno set appropriately.
 */
int xsim_nm_p2p_apply (unsigned int        source,
                       unsigned int        dest  ,
                       unsigned int        bytes ,
                       unsigned long long  send  ,
```

```
                      unsigned long long *recv  );

/**
 * Applies the network model to the receive time of a broadcast message.
 *
 * @param  root    The root rank (IN).
 * @param  dest    The destination rank (IN).
 * @param  comm    The communicator (IN).
 * @param  bytes   The buffer byte count (IN).
 * @param  send    The send time in microseconds (IN).
 * @param  recv    The receive time in microseconds (OUT).
 * @return         MPI_SUCCESS for success, or MPI_ERR_OTHER for error with
 *                 errno set appropriately.
 */
int xsim_nm_bcast_apply (unsigned int        root ,
                         unsigned int        dest ,
                         MPI_Comm            comm ,
                         unsigned int        bytes,
                         unsigned long long  send ,
                         unsigned long long *recv );

/**
 * Get the positive difference between two integers.
 *
 * @param one         The first node
 * @param two         The second node
 */
int get_absolute(int one, int two);

/**
 * Checks the network type and calls the appropriate function to calculate
 * latency.
 *
 * @param one         The heirarchy level
 * @param two         The network type
 * @param three       The number of nodes
 * @param four        The source rank
 * @param five        The destination rank
 * @param six         The latnecy
 * @param seven       The dimensions
 * @param eight       The connectedness
 * @param nine        The bandwidth
 * @param ten         The number of cores
 * @param eleven      The toroidal degree
 * @param twelve      The toroidal jump
 */
double get_latency (int srank, int drank, int count, int level, xsim_nm_type_t
type, int size, int src, int dst, double latency, unsigned int degree,
                    unsigned int *dimensions, unsigned int *connectedness, int
cores, int t_degree, unsigned int *t_jump);

/**
 * Get the latency between two nodes in a ring.
 *
 * @param one         The number of nodes
 * @param two         The source rank
 * @param three       The destination rank
```

```
 * @param four        The latency
 */
double get_ring_latency(int size, int src, int dst, double latency);

/**
 * Get the latency between two nodes in a mesh.
 *
 * @param one         The number of nodes
 * @param two         The source rank
 * @param three       The destination rank
 * @param four        The latency
 * @param five        The degree
 * @param six         The dimensions
 */
double get_mesh_latency(int size, int src, int dst, double latency, unsigned int
degree, unsigned int *dimensions);

/**
 * Get the latency between two nodes in a torus.
 *
 * @param one         The number of nodes
 * @param two         The source rank
 * @param three       The destination rank
 * @param four        The latency
 * @param five        The degree
 * @param six         The dimensions
 * @param seven       The connectedness
 */
double get_torus_latency(int size, int src, int dst, double latency, unsigned
int degree, unsigned int *dimensions, unsigned int *connectedness);

/**
 * Get the latency between two nodes in a twisted torus.
 *
 * @param one         The number of nodes
 * @param two         The source rank
 * @param three       The destination rank
 * @param four        The latency
 * @param five        The degree
 * @param six         The dimensions
 * @param seven       The connectedness
 * @param eight       The toroidal jump index
 * @param nine        The toroidal degree index
 */
double get_twisted_latency(int size, int src, int dst, double latency, unsigned
int degree, unsigned int *dimensions, unsigned int *connectedness,
    int tdegree, unsigned int *tjump);

/**
 * Get the latency between two nodes in a tree.
 *
 * @param one         The number of nodes
 * @param two         The source rank
 * @param three       The destination rank
 * @param four        The latency
 * @param five        The degree
 */
```

```
double get_tree_latency(int size, int src, int dst, double latency, unsigned int
degree);

/*****************************************************************************
 *
 * Data Exports
 *
 *****************************************************************************/

/** The network model. */
extern xsim_nm_t xsim_net;

/** The processor model. */
extern xsim_nm_t xsim_processor;

/** The total latency. */
extern double total_latency;

/** The number of receives. */
extern double total_receives;

#endif /* XSIM_NM_H */

/*****************************************************************************
 *
 * END OF FILE
 *
 *****************************************************************************/
```

7.1.2 *xsim_nm.c*

```
/*****************************************************************************//**
 *
 * @file xsim_nm.c
 *
 * Source file for the xsim library network model (NM) module.
 * Copyright (c) 2009-2010 Oak Ridge National Laboratory.
 *
 * For more information see the following files in the source distribution top-
 * level directory or package data directory (usually /usr/local/share/package):
 *
 * - README    for general package information.
 * - INSTALL   for package install information.
 * - COPYING   for package license information and copying conditions.
 * - AUTHORS   for package authors information.
 * - ChangeLog for package changes information.
 *
 *****************************************************************************/

/*****************************************************************************
 *
 * Macros
 *
 *****************************************************************************/

/*****************************************************************************
 *
 * Includes
 *
 *****************************************************************************/

/*************************
 * Include library header.
 *************************/
#include "xsim.h"

/*****************************************************************************
 *
 * Data Types
 *
 *****************************************************************************/

/*****************************************************************************
 *
 * Function Prototypes
 *
 *****************************************************************************/

/*****************************************************************************
 *
 * Data
 *
 *****************************************************************************/
```

```
/** The network model. */
xsim_nm_t xsim_net;

/** The processor model. */
xsim_nm_t xsim_processor;

/** The total latency. */
double total_latency;

/** The number of receives. */
double total_receives;

/** The number of cores per processor. */
unsigned int cores;

/** Index used for loops. */
unsigned int loop_index;

/*****************************************************************************
 *
 * Functions
 *
 *****************************************************************************/

/**
 * Initializes the network model module.
 *
 * @param  nmcfg  The network model command line configuration argument (IN).
 * @return        MPI_SUCCESS for success, or MPI_ERR_OTHER for error with
 *                errno set appropriately.
 */
int xsim_nm_init (char *nmcfg) {
  /* The parameter name. */
  char *name;
  /* The parameter value. */
  char *value;
  /* The parameter delimiter. */
  char *delimiter;
  /* The dimension delimiters. */
  char *delimiterA;
  char *delimiterB;
  /* Check the nmcfg parameter. */
  if (NULL == nmcfg) {
    /* Set errno. */
    errno = EINVAL;
    /* Log error. */
    XSIM_LOG_ERROR(The nmcfg parameter is null)
    /* Return error. */
    return MPI_ERR_OTHER;
  }
  /* Initialize the total latency. */
  total_latency = 0;
  /* Initialize the number of receives. */
  total_receives = 0;
  /* Initialize the number of cores. */
  cores = 1;
  /* Initialize the network type. */
```

```
xsim_net.type = XSIM_NM_STAR;
/* Initialize the network latency. */
xsim_net.latency = 1;
/* Initialize the network bandwidth. */
xsim_net.bandwidth = 1;
/* Initialize the network degree. */
xsim_net.degree = 1;
/* Initialize the network toroidal degree. */
xsim_net.t_degree = 1;
/* Initialise the network toroidal jump. */
xsim_net.t_jump = malloc(xsim_net.degree * sizeof(unsigned int));
for (loop_index = 0; loop_index < xsim_net.degree; loop_index++) {
  xsim_net.t_jump[loop_index] = 1;
}
/* Initialise the network toroidal connectedness. */
xsim_net.t_connectedness = malloc(xsim_net.degree * sizeof(unsigned int));
for (loop_index = 0; loop_index < xsim_net.degree; loop_index++) {
  xsim_net.t_connectedness[loop_index] = 1;
}
/* Initialise the network dimensions. */
xsim_net.dimensions = malloc(xsim_net.degree * sizeof(unsigned int));
for (loop_index = 0; loop_index < xsim_net.degree; loop_index++) {
  xsim_net.dimensions[loop_index] = 1;
}
/* Parse the command line configuration argument. */
for (name  = nmcfg;
     NULL != (delimiter = strchr(name, '='));
     name  = delimiter + 1) {
  /* Set the value pointer. */
  value = delimiter + 1;
  /* Search for the end delimiter. */
  if (NULL == (delimiter = strchr(value, ','))) {
    delimiter = value + strlen(value) + 1;
  }
  /* Check for the cores parameter name. */
  if (0 == strncmp(name, "cores", 5)) {
    /* Set the number of cores. */
    cores = atoi(value);
  }
  /* Check for the network type parameter name. */
  if ((0 == strncmp(name, "type", 4)) || (0 == strncmp(name, "ntype", 5))) {
    /* Check for the star network type parameter value. */
    if (0 == strncmp(value, "star", 4)) {
      /* Set the network type. */
      xsim_net.type = XSIM_NM_STAR;
    }
    /* Check for the ring network type parameter value. */
    else if (0 == strncmp(value, "ring", 4)) {
      /* Set the network type. */
      xsim_net.type = XSIM_NM_RING;
    }
    /* Check for the mesh network type parameter value. */
    else if (0 == strncmp(value, "mesh", 4)) {
      /* Set the network type. */
      xsim_net.type = XSIM_NM_MESH;
    }
    /* Check for the torus network type parameter value. */
```

```
      else if (0 == strncmp(value, "torus", 5)) {
        /* Set the network type. */
        xsim_net.type = XSIM_NM_TORUS;
      }
      /* Check for the twisted torus network type parameter value. */
      else if (0 == strncmp(value, "twistedtorus", 12)) {
        /* Set the network type. */
        xsim_net.type = XSIM_NM_TWISTED_TORUS;
      }
      /* Check for the tree network type parameter value. */
      else if (0 == strncmp(value, "tree", 4)) {
        /* Set the network type. */
        xsim_net.type = XSIM_NM_TREE;
      }
    }
    /* Check for the network latency parameter name. */
    else if ((0 == strncmp(name, "latency", 7)) || (0 == strncmp(name,
"nlatency", 8))) {
      /* Set the network latency. */
      xsim_net.latency = atof(value);
    }
    /* Check for the network bandwidth parameter name. */
    else if ((0 == strncmp(name, "bandwidth", 9)) || (0 == strncmp(name,
"nbandwidth", 10))) {
      /* Set the network bandwidth. */
      xsim_net.bandwidth = atof(value);
    }
    /* Check for network degree parameter name. */
    else if ((0 == strncmp(name, "degree", 6)) || (0 == strncmp(name, "ndegree",
7))) {
      /* Set the network degree. */
      xsim_net.degree = atoi(value);
    }
    /* Check for network toroidal degree parameter name. */
    else if ((0 == strncmp(name, "t_degree", 8)) || (0 == strncmp(name,
"nt_degree", 9))) {
      /* Set the network toroidal degree. */
      xsim_net.t_degree = atoi(value);
    }
    /* Check for network toroidal connectedness parameter name. */
    else if ((0 == strncmp(name, "t_connectedness", 15)) || (0 == strncmp(name,
"nt_connectedness", 16))) {
      /* Initialise the network toroidal connectedness. */
      xsim_net.t_connectedness = malloc(xsim_net.degree * sizeof(unsigned int));
      /* Set the network toroidal connectedness. */
      for (loop_index = 0; loop_index < xsim_net.degree; loop_index++) {
        xsim_net.t_connectedness[loop_index] = (unsigned int)value[loop_index]-
48;
      }
    }
    /* Check for network toroidal jump parameter name. */
    else if ((0 == strncmp(name, "t_jump", 6)) || (0 == strncmp(name, "nt_jump",
7))) {
      /* Initialise the network toroidal jump. */
      xsim_net.t_jump = malloc(xsim_net.degree * sizeof(unsigned int));
      /* Set the starting values for the delimiters. */
      delimiterA = strchr(value,'*');
```

```
      delimiterB = value;
      /* Holds the extracted substring containing a single dimension. */
      char *value_str;
      /* Holds the current location in the dimension array. */
      unsigned int dimension_index = 0;
      /* Process each dimension. */
      while ((delimiterA != NULL) && (dimension_index < xsim_net.degree)) {
        /* Allocate enough space to hold the ASCII string of the next dimension.
*/
        value_str = malloc((delimiterA - delimiterB + 1) * sizeof(char));
        /* Read the substring. */
        for (loop_index = 0; loop_index < (int)strlen(value_str); loop_index++)
{
          value_str[loop_index] = value[delimiterB - value + loop_index];
        }
        /* Convert substring to integer. */
        xsim_net.t_jump[xsim_net.degree - dimension_index - 1] = atoi(value_str);
        dimension_index++;
        /* Locate the next delimiter. */
        delimiterB = delimiterA + 1;
        delimiterA = strchr(delimiterA + 1, '*');
      }
      /* Check for the special case where only one dimension is defined. */
      if (dimension_index == 0) {
        /* Set all dimensions equal to this value. */
        for (loop_index = 1; loop_index < xsim_net.degree; loop_index++) {
          xsim_net.t_jump[loop_index] = xsim_net.t_jump[0];
        }
      }
    }
    /* Check for network dimensions parameter name. */
    else if ((0 == strncmp(name, "dimensions", 10)) || (0 == strncmp(name,
"ndimensions", 11))) {
      /* Initialise the network dimensions. */
      xsim_net.dimensions = malloc(xsim_net.degree * sizeof(unsigned int));
      /* Set the starting values for the delimiters. */
      delimiterA = strchr(value,'*');
      delimiterB = value;
      /* Holds the extracted substring containing a single dimension. */
      char *value_str;
      /* Holds the current location in the dimension array. */
      unsigned int dimension_index = 0;
      /* Process each dimension. */
      while ((delimiterA != NULL) && (dimension_index < xsim_net.degree)) {
        /* Allocate enough space to hold the ASCII string of the next dimension.
*/
        value_str = malloc((delimiterA - delimiterB + 1) * sizeof(char));
        /* Read the substring. */
        for (loop_index = 0; loop_index < (int)strlen(value_str); loop_index++)
{
          value_str[loop_index] = value[delimiterB - value + loop_index];
        }
        /* Convert substring to integer. */
        xsim_net.dimensions[xsim_net.degree - dimension_index - 1] =
atoi(value_str);
        dimension_index++;
        /* Locate the next delimiter. */
```

```
    delimiterB = delimiterA + 1;
    delimiterA = strchr(delimiterA + 1, '*');
  }
  /* Check for the special case where only one dimension is defined. */
  if (dimension_index == 0) {
    /* Set all dimensions equal to this value. */
    for (loop_index = 1; loop_index < xsim_net.degree; loop_index++) {
      xsim_net.dimensions[loop_index] = xsim_net.dimensions[0];
    }
  }
}
/* Check for the processor type parameter name. */
else if (0 == strncmp(name, "ptype", 5)) {
  /* Check for the star network type parameter value. */
  if (0 == strncmp(value, "star", 4)) {
    /* Set the network type. */
    xsim_processor.type = XSIM_NM_STAR;
  }
  /* Check for the ring network type parameter value. */
  else if (0 == strncmp(value, "ring", 4)) {
    /* Set the network type. */
    xsim_processor.type = XSIM_NM_RING;
  }
  /* Check for the mesh network type parameter value. */
  else if (0 == strncmp(value, "mesh", 4)) {
    /* Set the network type. */
    xsim_processor.type = XSIM_NM_MESH;
  }
  /* Check for the torus network type parameter value. */
  else if (0 == strncmp(value, "torus", 5)) {
    /* Set the network type. */
    xsim_processor.type = XSIM_NM_TORUS;
  }
  /* Check for the twisted torus network type parameter value. */
  else if (0 == strncmp(value, "twistedtorus", 12)) {
    /* Set the network type. */
    xsim_processor.type = XSIM_NM_TWISTED_TORUS;
  }
  /* Check for the tree network type parameter value. */
  else if (0 == strncmp(value, "tree", 4)) {
    /* Set the network type. */
    xsim_processor.type = XSIM_NM_TREE;
  }
}
/* Check for the network latency parameter name. */
else if (0 == strncmp(name, "platency", 8)) {
  /* Set the network latency. */
  xsim_processor.latency = atof(value);
}
/* Check for the network bandwidth parameter name. */
else if (0 == strncmp(name, "pbandwidth", 10)) {
  /* Set the network bandwidth. */
  xsim_processor.bandwidth = atof(value);
}
/* Check for network degree parameter name. */
else if (0 == strncmp(name, "pdegree", 7)) {
  /* Set the network degree. */
```

```
      xsim_processor.degree = atoi(value);
    }
    /* Check for network toroidal degree parameter name. */
    else if (0 == strncmp(name, "pt_degree", 9)) {
      /* Set the network toroidal degree. */
      xsim_processor.t_degree = atoi(value);
    }
    /* Check for network toroidal connectedness parameter name. */
    else if (0 == strncmp(name, "pt_connectedness", 16)) {
      /* Initialise the network toroidal connectedness. */
      xsim_processor.t_connectedness = malloc(xsim_processor.degree *
sizeof(unsigned int));
      /* Set the network toroidal connectedness. */
      for (loop_index = 0; loop_index < xsim_processor.degree; loop_index++) {
        xsim_processor.t_connectedness[loop_index] = (unsigned
int)value[loop_index]-48;
      }
    }
    /* Check for network toroidal jump parameter name. */
    else if (0 == strncmp(name, "pt_jump", 7)) {
      /* Initialise the network toroidal jump. */
      xsim_processor.t_jump = malloc(xsim_processor.degree * sizeof(unsigned
int));
      /* Set the starting values for the delimiters. */
      delimiterA = strchr(value,'*');
      delimiterB = value;
      /* Holds the extracted substring containing a single dimension. */
      char *value_str;
      /* Holds the current location in the dimension array. */
      unsigned int dimension_index = 0;
      /* Process each dimension. */
      while ((delimiterA != NULL) && (dimension_index < xsim_processor.degree))
{
        /* Allocate enough space to hold the ASCII string of the next dimension.
*/
        value_str = malloc((delimiterA - delimiterB + 1) * sizeof(char));
        /* Read the substring. */
        for (loop_index = 0; loop_index < (int)strlen(value_str); loop_index++)
{
          value_str[loop_index] = value[delimiterB - value + loop_index];
        }
        /* Convert substring to integer. */
        xsim_processor.t_jump[xsim_processor.degree - dimension_index - 1] =
atoi(value_str);
        dimension_index++;
        /* Locate the next delimiter. */
        delimiterB = delimiterA + 1;
        delimiterA = strchr(delimiterA + 1, '*');
      }
      /* Check for the special case where only one dimension is defined. */
      if (dimension_index == 0) {
        /* Set all dimensions equal to this value. */
        for (loop_index = 1; loop_index < xsim_processor.degree; loop_index++) {
          xsim_processor.t_jump[loop_index] = xsim_processor.t_jump[0];
        }
      }
    }
```

```
    /* Check for network dimensions parameter name. */
    else if (0 == strncmp(name, "pdimensions", 11)) {
      /* Initialise the network dimensions. */
      xsim_processor.dimensions = malloc(xsim_processor.degree * sizeof(unsigned
int));
      /* Set the starting values for the delimiters. */
      delimiterA = strchr(value,'*');
      delimiterB = value;
      /* Holds the extracted substring containing a single dimension. */
      char *value_str;
      /* Holds the current location in the dimension array. */
      unsigned int dimension_index = 0;
      /* Process each dimension. */
      while ((delimiterA != NULL) && (dimension_index < xsim_processor.degree))
{
        /* Allocate enough space to hold the ASCII string of the next dimension.
*/
        value_str = malloc((delimiterA - delimiterB + 1) * sizeof(char));
        /* Read the substring. */
        for (loop_index = 0; loop_index < (int)strlen(value_str); loop_index++)
{
          value_str[loop_index] = value[delimiterB - value + loop_index];
        }
        /* Convert substring to integer. */
        xsim_processor.dimensions[xsim_processor.degree - dimension_index - 1] =
atoi(value_str);
        dimension_index++;
        /* Locate the next delimiter. */
        delimiterB = delimiterA + 1;
        delimiterA = strchr(delimiterA + 1, '*');
      }
      /* Check for the special case where only one dimension is defined. */
      if (dimension_index == 0) {
        /* Set all dimensions equal to this value. */
        for (loop_index = 1; loop_index < xsim_processor.degree; loop_index++) {
          xsim_processor.dimensions[loop_index] = xsim_processor.dimensions[0];
        }
      }
    }
  }
  /* Return success. */
  return MPI_SUCCESS;
}

/**
 * Finalizes the network model module.
 *
 * @return  MPI_SUCCESS for success, or MPI_ERR_OTHER for error with errno set
 *          appropriately.
 */
int xsim_nm_fini () {
  /* Return success. */
  return MPI_SUCCESS;
}

/**
 * Applies the network model to the receive time of a point-to-point message.
```

```
 *
 * @param  source  The source rank in MPI_COMM_WORLD (IN).
 * @param  dest    The destination rank in MPI_COMM_WORLD (IN).
 * @param  bytes   The buffer byte count (IN).
 * @param  send    The send time in microseconds (IN).
 * @param  recv    The receive time in microseconds (OUT).
 * @return         MPI_SUCCESS for success, or MPI_ERR_OTHER for error with
 *                 errno set appropriately.
 */
int xsim_nm_p2p_apply (unsigned int        source,
                       unsigned int        dest  ,
                       unsigned int        bytes ,
                       unsigned long long  send  ,
                       unsigned long long *recv  ) {
total_receives ++;
  total_latency +=
          get_latency(source, dest, xsim_vps.count, 1, xsim_net.type,
xsim_sim.count, source/cores, dest/cores,
          xsim_net.latency, xsim_net.degree, xsim_net.dimensions,
xsim_net.t_connectedness, cores, xsim_net.t_degree, xsim_net.t_jump);
  /* Calculate the receive time. */
  *recv = send +
          get_latency(source, dest, xsim_vps.count, 1, xsim_net.type,
xsim_sim.count, source/cores, dest/cores,
          xsim_net.latency, xsim_net.degree, xsim_net.dimensions,
xsim_net.t_connectedness, cores, xsim_net.t_degree, xsim_net.t_jump);
  if ((a/(xsim_vps.count/cores)) == (b/(xsim_vps.count/cores))) {
    total_latency += ((0 == xsim_net.bandwidth)?0:
                (unsigned long long)
                ((bytes/(131072.0 * xsim_net.bandwidth)) + 0.5));
    *recv += ((0 == xsim_net.bandwidth)?0:
                (unsigned long long)
                ((bytes/(131072.0 * xsim_net.bandwidth)) + 0.5));
  }
  else {
    total_latency += ((0 == xsim_processor.bandwidth)?0:
                (unsigned long long)
                ((bytes/(131072.0 * xsim_processor.bandwidth)) + -.5));
    *recv += ((0 == xsim_processor.bandwidth)?0:
                (unsigned long long)
                ((bytes/(131072.0 * xsim_processor.bandwidth)) + -.5));
  }

  /* Return success. */
  return MPI_SUCCESS;
}

/**
 * Applies the network model to the receive time of a broadcast message.
 *
 * @param  root    The root rank (IN).
 * @param  dest    The destination rank (IN).
 * @param  comm    The communicator (IN).
 * @param  bytes   The buffer byte count (IN).
 * @param  send    The send time in microseconds (IN).
 * @param  recv    The receive time in microseconds (OUT).
 * @return         MPI_SUCCESS for success, or MPI_ERR_OTHER for error with
```

```
 *                 errno set appropriately.
 */
int xsim_nm_bcast_apply (unsigned int        root ,
                         unsigned int        dest ,
                         MPI_Comm            comm ,
                         unsigned int        bytes,
                         unsigned long long  send ,
                         unsigned long long *recv ) {
  /* Return success. */
  return MPI_SUCCESS;
}

/**
 * Checks the network type and calls the appropriate function to calculate
 * latency.
 *
 * @param one         The heirarchy level
 * @param two         The network type
 * @param three       The number of nodes
 * @param four        The source rank
 * @param five        The destination rank
 * @param six         The latnecy
 * @param seven       The dimensions
 * @param eight       The connectedness
 * @param nine        The number of cores
 * @param ten         The toroidal degree
 * @param eleven      The toroidal jump
 */
double get_latency (int srank, int drank, int count, int level, xsim_nm_type_t
type, int size, int src, int dst, double latency, unsigned int degree,
                    unsigned int *dimensions, unsigned int *connectedness, int
cores, int tdegree, unsigned int *tjump) {
  /* Check if the nodes are on the same processor. */
  if ((level == 1) && (cores > 1) && (src == dst)) {
       /* Recursively find the latency considering a processor topology */
    return get_latency(srank, drank, count, 2, xsim_processor.type, cores,
srank%(cores), drank%(cores), xsim_processor.latency,
                       xsim_processor.degree, xsim_processor.dimensions,
xsim_processor.t_connectedness, cores, xsim_processor.t_degree,
                       xsim_processor.t_jump);
  }
  /* Check if the nodes are on different processors. */
  else {
    /* Check the network type. */
    switch (type) {
      /* Check for star network. */
      case XSIM_NM_STAR: {
        return 2*latency;
      }
      /* Check for ring network. */
      case XSIM_NM_RING: {
        return get_ring_latency(size, src, dst, latency);
      }
      /* Check for mesh network. */
      case XSIM_NM_MESH: {
        return get_mesh_latency(size, src, dst, latency, degree, dimensions);
      }
```

```
      /* Check for torus networkk. */
      case XSIM_NM_TORUS: {
        return get_torus_latency(size, src, dst, latency, degree, dimensions,
connectedness);
      }
      /* Check for twisted torus network. */
      case XSIM_NM_TWISTED_TORUS: {
        return get_twisted_latency(size, src, dst, latency, degree, dimensions,
connectedness, tdegree, tjump);
      }
      /* Check for tree network. */
      case XSIM_NM_TREE: {
        return get_tree_latency(size, src, dst, latency, degree);
      }
      /* Check for unsupported network. */
      default: {
        /* Log error. */
        XSIM_LOG_ERROR(Unsupported network type)
        /* Return error. */
        return -1;
      }
    }
  }
}

/**
 * Get the positive difference between two integers.
 *
 * @param one         The first node
 * @param two         The second node
 */
int get_absolute(int one, int two) {
  /* Check if the first number is larger. */
  if (one >= two) {
    /* Return the positive difference. */
    return one - two;
  }
  /* Check if the second number is larger. */
  else {
    /* Return the positive difference. */
    return two - one;
  }
}

/**
 * Get ring latency.
 *
 * @param one         The number of nodes
 * @param two         The source rank
 * @param three       The destination rank
 * @param four        The latency
 */
double get_ring_latency(int size, int src, int dst, double latency) {
  /* Check if source is ahead of destination. */
  if (dst < src) {
    /* Return the latency cost to loop around the ring. */
    return latency*(size - get_absolute(src, dst));
```

```
  }
  /* Check if the destination is ahead of the source. */
  else {
    /* Return the latency cost to travel directly from source to destination. */
    return latency*get_absolute(src, dst);
  }
}

/**
 * Get mesh latency.
 *
 * @param one         The number of nodes
 * @param two         The source rank
 * @param three       The destination rank
 * @param four        The latency
 * @param five        The degree
 * @param six         The dimensions
 */
double get_mesh_latency(int size, int src, int dst, double latency, unsigned int
degree, unsigned int *dimensions) {
  /* The total mesh distance between the source and the destination. */
  int mesh_distance = 0;
  /* Array to hold mesh co-ordinates of source. */
  unsigned int cartesian_src[degree];
  /* Array to hold mesh co-ordinates of destination. */
  unsigned int cartesian_dst[degree];
  /* The number of remaining processors in x dimensions. */
  int remaining_count = size;
  int remaining_src = src;
  int remaining_dst = dst;
  /* Iterate through each dimension. */
  for (loop_index = 1; loop_index <= degree; loop_index++) {
    /* Calculate the processors in the remaining dimensions. */
    remaining_count /= dimensions[degree-loop_index];
    /* Calculate the position of source within the current dimension. */
    cartesian_src[loop_index-1] = remaining_src/remaining_count;
    remaining_src %= remaining_count;
    /* Calculate the position of destination within the current dimension. */
    cartesian_dst[loop_index-1] = remaining_dst/remaining_count;
    remaining_dst %= remaining_count;
    /* Get the latency for traversing the current dimension. */
    mesh_distance += get_absolute(cartesian_src[loop_index-1],
cartesian_dst[loop_index-1]);
  }
  return latency*mesh_distance;
}

/**
 * Get torus latency.
 *
 * @param one         The number of nodes
 * @param two         The source rank
 * @param three       The destination rank
 * @param four        The latency
 * @param five        The degree
 * @param six         The dimensions
 * @param seven       The connectedness
```

```
 */
double get_torus_latency(int size, int src, int dst, double latency, unsigned
int degree,
                                      unsigned int *dimensions, unsigned int
*connectedness) {
  /* The total mesh distance between the source and the destination. */
  int torus_distance = 0;
  /* Array to hold mesh co-ordinates of source. */
  unsigned int cartesian_src[degree];
  /* Array to hold mesh co-ordinates of destination. */
  unsigned int cartesian_dst[degree];
  /* The number of remaining processors in x dimensions. */
  int remaining_count = size;
  int remaining_src = src;
  int remaining_dst = dst;
  /* Iterate through each dimension. */
  for (loop_index = 1; loop_index <= degree; loop_index++) {
    /* Calculate the processors in the remaining dimensions. */
    remaining_count /= dimensions[degree-loop_index];
    /* Calculate the position of source within the current dimension. */
    cartesian_src[loop_index-1] = remaining_src/remaining_count;
    remaining_src %= remaining_count;
    /* Calculate the position of destination within the current dimension. */
    cartesian_dst[loop_index-1] = remaining_dst/remaining_count;
    remaining_dst %= remaining_count;
    /* Check if current dimension is toroidal. */
    if (connectedness[loop_index-1] == 1) {
      /* Check if nodes are more closely connected directly. */
      if (get_absolute(cartesian_src[loop_index-1], cartesian_dst[loop_index-1])
<= (int)(dimensions[loop_index-1]/2)) {
        /* Get the latency for directly traversing the current dimension. */
        torus_distance += get_absolute(cartesian_src[loop_index-1],
cartesian_dst[loop_index-1]);
      }
      else {
        /* Get the latency for toroidally traversing the current dimension. */
        torus_distance += (dimensions[degree-loop_index] -
get_absolute(cartesian_src[loop_index-1], cartesian_dst[loop_index-1]));
      }
    }
    /* Check if the current dimension is not toroidal. */
    else {
      /* Get the latency for directly traversing the current dimension. */
      torus_distance += get_absolute(cartesian_src[loop_index-1],
cartesian_dst[loop_index-1]);
    }
  }
  return latency*torus_distance;
}

/**
 * Get twisted torus latency.
 *
 * @param one         The number of nodes
 * @param two         The source rank
 * @param three       The destination rank
 * @param four        The latency
```

```
 * @param five        The degree
 * @param six         The dimensions
 * @param seven       The connectedness
 * @param eight       The toroidal jump index
 * @param nine        The toroidal degree index
 */
double get_twisted_latency(int size, int src, int dst, double latency, unsigned
int degree, unsigned int *dimensions, unsigned int *connectedness,
    int tdegree, unsigned int *tjump) {

/* Switch source and dest so dest is always greater - avoids some issues. */
if (src > dst) {
  int tmp = src;
  src = dst;
  dst = tmp;
}
  /* The total twisted torus distance between source and destination. */
  int twisted_distance = 0;
  /* Array to hold mesh co-ordinates of source. */
  unsigned int cartesian_src[degree];
  /* Array to hold mesh co-ordinates of destination. */
  unsigned int cartesian_dst[degree];
  /* The number of remaining processors in x dimensions. */
  int remaining_count = size;
  int remaining_src = src;
  int remaining_dst = dst;
  /* Iterate through each dimension. */
  for (loop_index = 1; loop_index <= degree; loop_index++) {
    /*Calculate the processors in the remaining dimensions. */
    remaining_count /= dimensions[degree-loop_index];
    /* Calculate the psoition of source within the current dimension. */
    cartesian_src[loop_index-1] = remaining_src/remaining_count;
    remaining_src %= remaining_count;
    /* Calculate the position of destination within the current dimension. */
    cartesian_dst[loop_index-1] = remaining_dst/remaining_count;
    remaining_dst %= remaining_count;
  }
  /* Array to hold record of dimensions traversed. */
  unsigned int record[degree][degree+3];
  /* Assign all dimensions as not yet traversed. */
  for (loop_index = 0; loop_index < degree; loop_index++) {
    /* Set as 0 to indicate not done. */
    record[loop_index][0] = 0;
  }
  /* Loop variables. */
  unsigned int trav = 0;
  unsigned int test = 0;
  unsigned int source = 0;
  /* Traverse each dimension. */
  for (trav = 0; trav < degree; trav++) {
    /* Test each dimension. */
    for (test = 0; test < degree; test++) {
      /* Check if current dimension has not been traversed. */
      if (record[test][0] == 0) {
        /* Holds the possible position after dimension has been traversed
directly. */
        unsigned int direct_location[degree];
```

```
        /* Holds the possible position after dimension has been traversed by
looping. */
        unsigned int loop_location[degree];
        /* Use the source array as the starting point. */
        for (source = 0; source < degree; source++) {
          /* Copy the source into the direct/loop arrays. */
          direct_location[source] = cartesian_src[source];
          loop_location[source] = cartesian_src[source];
        }
        /* Calculate the new position from direct traversal. */
        direct_location[test] = cartesian_dst[test];
        /* Check if current dimension is toroidal. */
        if (connectedness[test] == 1) {
          /* Check if the source node is 'ahead' of the destination node in the
current dimension. */
          if (cartesian_src[test] > cartesian_dst[test]) {
            /* Check if the next dimension is the first. */
            if (test+tdegree >= degree) {
              /* Check if the first dimension is at the last level. */
              if (loop_location[(test+tdegree)-degree] + tjump[test] >=
dimensions[(test+tdegree)-degree]) {
                /* Loop around in the positive direction to the start of the
first dimension. */
                loop_location[(test+tdegree)-degree] = tjump[test] -
(dimensions[(test+tdegree)-degree] - loop_location[(test+tdegree)-degree]);
              }
              else {
                /* Loop around in the positive direction in the first dimension.
*/
                loop_location[(test+tdegree)-degree] += tjump[test];
              }
            }
            else {
              /* Check if the next dimension is at the last level. */
              if (loop_location[test+tdegree] + tjump[test] >=
dimensions[degree-test-tdegree-1]-1) {
                /* Loop around in the positive direction to the start of the
next dimension. */
                loop_location[test+tdegree] = tjump[test] -
(dimensions[test+tdegree] - loop_location[test+tdegree]);
              }
              else {
                /* Loop around in the positive direction in the next dimension.
*/
                loop_location[test+tdegree] += tjump[test];
              }
            }
          }
          /* Check if the destination node is 'ahead' of the source node in the
current dimension. */
          else {
            /* Check if the next dimension is the first. */
            if (test+tdegree >= degree) {
              /* Check if the first dimension is at the first level. */
              if ((int)(loop_location[(test+tdegree)-degree] - tjump[test]) < 0)
{
                /* Loop around in the negative direction to the end of the first
```

```
dimension. */
                loop_location[(test+tdegree)-degree] =
dimensions[(test+tdegree)-degree] + loop_location[(test+degree)-degree] -
tjump[test]-1;
              }
              else {
                /* Loop around in the negative direction in the first dimension.
*/
                loop_location[(test+tdegree)-degree] -= tjump[test];
              }
            }
            else {
              /* Check if the next dimension is at the first level. */
              if ((int)(loop_location[test+tdegree] - tjump[test]) < 0) {
                /* Loop around in the negative direction to the end of the next
dimension. */
                loop_location[test+tdegree] = dimensions[test+tdegree] +
loop_location[test+tdegree]- tjump[test];
              }
              else {
                /* Loop around in the negative direction in the next dimension.
*/
                loop_location[test+tdegree] -= tjump[test];
              }
            }
          }
          /* Calculate the toroidal distance between the source and the
destination in the current dimension. */
          loop_location[test] = cartesian_dst[test];
          /* Calculate the cost of traversing both directly and by loop. */
          int direct_cost = 0;
          int loop_cost = 0;
          /* Check how far from the destination in each dimension sequentially.
*/
          for (loop_index = 0; loop_index < degree; loop_index++) {
            /* Calculate the cost for direct traversal in this dimension. */
            direct_cost += get_absolute(direct_location[loop_index],
cartesian_dst[loop_index]);
            /* Calculate the cost for loop traversal in this dimension. */
            loop_cost += get_absolute(loop_location[loop_index],
cartesian_dst[loop_index]);
          }
          /* Factor the cost for traversing the current dimension. */
          direct_cost += get_absolute(cartesian_src[test],
direct_location[test]);
          /* Factor the cost for traversing the current dimension. */
          loop_cost += (dimensions[test]-1 - get_absolute(cartesian_src[test],
loop_location[test]));
          /* Check if the direct method is best. */
          if (direct_cost <= loop_cost) {
            /* Save the cost of traversing this dimension. */
            record[test][1] = get_absolute(cartesian_src[test],
direct_location[test]);
            record[test][2] = direct_cost;
            /* Save the new position should this dimension be traversed. */
            for (loop_index = 0; loop_index < degree; loop_index++) {
              /* Save the loop_index of each dimension. */
```

```
              record[test][loop_index+3] = direct_location[loop_index];
            }
          }
          /* Check if the loop method is best. */
          else {
            /* Save the cost of traversing this dimension. */
            record[test][1] = (dimensions[degree-test-1] -
get_absolute(cartesian_src[test], direct_location[test]));
            record[test][2] = loop_cost;
            /* Save the new position should this dimension be traversed. */
            for (loop_index = 0; loop_index < degree; loop_index++) {
              /* Save the position of each dimension. */
              record[test][loop_index+3] = loop_location[loop_index];
            }
          }
        }
        /* If not toroidal use direct by default. */
        else {
          int direct_cost = 0;
          /* Check how far from the destination in each dimension sequentially.
*/
          for (loop_index = 0; loop_index < degree; loop_index++) {
            /* Calculate the cost for direct traversal in this dimension. */
            direct_cost += get_absolute(direct_location[loop_index],
cartesian_dst[loop_index]);
          }
          /* Factor the cost for traversing the current dimension. */
          direct_cost += get_absolute(cartesian_src[test],
direct_location[test]);
          /* Save the cost of traversing this dimension. */
          record[test][1] = get_absolute(cartesian_src[test],
direct_location[test]);
          record[test][2] = direct_cost;
          /* Save the new position should this dimension be traversed. */
          for (loop_index = 0; loop_index < degree; loop_index++) {
            /* Save the position of each dimension. */
            record[test][loop_index+3] = direct_location[loop_index];
          }
        }
      }
    }
    /* Find the dimension which is best to traverse next. */
    int min_cost = 0;
    /* Find the first dimension which has NOT yet been traversed as a minimum.
*/
    while (record[min_cost][0] != 0) {
      min_cost++;
    }
    /* Check each dimension to find the minimum cost. */
    for (loop_index = 0; loop_index < degree; loop_index++) {
      /* Check if this dimension is better than the current best. */
      if ((record[loop_index][2] < record[min_cost][2]) &&
(record[loop_index][0] != 1)) {
        /* Set this dimension as the best. */
        min_cost = loop_index;
      }
    }
```

```
    /* Add the traversal cost to the total cost. */
    twisted_distance += record[min_cost][1];
    /* Commit the changes by traversing the best dimension. */
    for (loop_index = 0; loop_index < degree; loop_index++) {
      /* Update the position of each dimension. */
      cartesian_src[loop_index] = record[min_cost][loop_index+3];
    }
    /* Mark dimension as traversed. */
    record[min_cost][0] = 1;
    /* Check if destination has already been reached. */
    int difference = 0;
    /* Check every dimension. */
    for (loop_index = 0; loop_index < degree; loop_index++) {
      /* Check for equality. */
      if (cartesian_src[loop_index] != cartesian_dst[loop_index]) {
        difference++;
      }
    }
    /* Check if there is no need to continue traversing dimensions. */
    if (difference == 0) {
      /* Add the latency of traversing the twisted torus to the message VP
source time. */
      return latency*twisted_distance;
    }
  }
  return latency*twisted_distance;

}
/**
 *
 * Get tree latency.
 *
 * @param one       The number of nodes
 * @param two       The source rank
 * @param three     The destination rank
 * @param four      The latency
 * @param five      The degree
 */
double get_tree_latency(int size, int src, int dst, double latency, unsigned int
degree) {
  /* The total tree distance between the source and the destination. */
  int tree_distance = 0;
  /* The parent of the source node. */
  unsigned int src_parent = src;
  /* The parent of the destination node. */
  unsigned int dst_parent = dst;
  /* Determine if the source and destination nodes share a common link on this
level. */
  while (src_parent != dst_parent) {
    /* Add the cost of traversing the current level to the total distance. */
    tree_distance += 2;
    /* Calculate the parent of the current source. */
    src_parent /= degree;
    /* Calculate the parent of the current destination. */
    dst_parent /= degree;
  }
  return latency*tree_distance;
```

```
}

/*****************************************************************************
 *
 * END OF FILE
 *
 *****************************************************************************/
```