LDRD 98-0260 Final Report
High performance commodity interconnects for clustered scientific and engineering computing

Helen Chen
Sandia National Laboratories, MS 9011
P.O. Box 969, Livermore, CA 94551, USA
Phone 925 294 2991 FAX 925 294 1225
hycsw@ca.sandia.gov

Pete Wyckoff
Sandia National Laboratories, MS 9011
P.O. Box 969, Livermore, CA 94551, USA
Phone 925 294 3503 FAX 925 294 1225
wyckoff@ca.sandia.gov

Abstract

The Computational Plant (CPlant) project will run distributed parallel applications on a large cluster of commodity PCs. Because properly load-balanced distributed parallel applications tend to send messages synchronously, minimizing blocking is as crucial a requirement as are high bandwidth and low latency. Therefore, we consider the selection of an optimal, commodity-based, interconnect network technology and topology to provide high bandwidth, low latency, and reliable delivery to be an important design consideration.

Since our network design goal is to facilitate the performance of real applications, we evaluated the performance of myrinet and gigabit ethernet technologies in the context of working algorithms using modeling and simulation tools developed in this project. In addition to latency and bandwidth, we evaluated performance enhancements to parallel algorithms using hardware-based multicast and cut-through routing.

Our simulation results show that myrinet behaves well in the absence of congestion. Under heavy load, its latency suffers due to blocking in wormhole routing. Also, because of its severe cable length constraint, it limits myrinet's ability to scale. The simplicity in the myrinet switch results in low per-connection cost; however, it is also the reason for its lack of manageability and robustness in large systems.

Conventional gigabit ethernet switches can not scale to support more than 64 gigabit ethernet ports. Therefore, in order to build large parallel systems, switches must be cascaded. Several limitations arise as a result of this constraint. The ethernet spanning tree routing algorithm precludes a mesh topology. Without diverse paths, interswitch links become bandwidth bottlenecks. These switches store and then forward packets at each hop, inducing additional end-to-end latency among cascaded switches. However, conventional gigabit ethernet switches delivery the best multicast performance because they implemented multicast in hardware

The Avici terabit switch router (TSR) uses six 40 Gb/s internal links to interconnect individual switching nodes in a 3D toroidal mesh. Its switch fabric uses wormhole routing to provide cut-through latency to the gigabit ethernet hosts connected via its network IO cards. Our simulation studies show that it performed as well as myrinet when messages are smaller than 256 bytes, and progressively better when message sizes increased beyond 512 bytes. It also proved to be highly scaleable and robust. Riding on the ethernet popularity current, we expect the Avici solution to become cost efficient.

1 Introduction

As background information to explain the motivation behind the project, we provide an overview of cluster computing, technological developments which have brought us to this situation, and the current state of the art in high-performance networking protocols and hardware.

1.1 Cluster computing

A community of enthusiasts of commodity high-performance computing platforms has lived at the fringes of the computing community for many years now, and this bunch is seeing its numbers grow following the general demise of the massively parallel processor (MPP) manufacturers. Even the government research laboratories are joining the fray. Sandia and Ames National Laboratories are two United States Department of Energy research facilities which can no longer satisfy their thirst for FLOPS by buying monolithic multi-million dollar machines, as there is not sufficient market demand to keep vendors in business.

The idea of cluster computing is to aggregate machine rooms full of relatively cheap hardware, connected with some sort of network, and apply the combined force of the individual machines on a single calculation. Problems arise, though, in attempting to operate this set of machines as a single unit. As it is not feasible to run a single operating system on the entire cluster, the alternate paradigm of message passing is used instead. Each processor (which could also be a small shared-memory multiprocessor) maintains a disjoint address space, and messages are passed between machines as driven by the requirements of each application.

The hardware employed in a cluster is generally the most readily available in the volume personal computing market, so as to leverage the cost advantages of buying commodity hardware. The down-side to this is that some critical pieces of hardware for cluster computing are completely irrelevant for the mass market, namely the interconnect. The advent of shared 10 Mb/s ethernet [3] was a giant step, and remains the basis of the standard "fast" networking infrastructure, as it has been for the last 15 years. Within the last few years, though, the price of 100 Mb/s ethernet cards have approached the reach of most users, and commodity gigabit network components are on their way. More esoteric networking components, such as myrinet [9] and HiPPi [10] are available to those willing to incur the additional costs, and are also improving with time.

The remainder of the report discusses the technologies we simulated, and methods to that end which involve a mix of "artificial" basic tests and simulations of core algorithsm from real parallel applications. Our results tally the positive and negative aspects of each technology.

2 Interconnect technologies

Parallel computers using a large number of commodity PCs are opening the doors to providing cost-effective teraflop computing powers, where thousands of processors cooperate to solve a single large problem. Co-operating processes from parallel computers synchronize and communicate by passing explicit messages, and therefore require a very high-performance interconnection network to avoid communication bottlenecks. This study compares the performance of myrinet and gigabit ethernet as cluster-interconnect using simulation methods. In addition we evaluate their scalability, incremental expandability, physical limitations, robustness, and cost constraints.

2.1 Myrinet

Sandia chose Myricom's myrinet to interconnect its Alpha cluster because it is a cost-effective, high-performance communication and switching technology. It interconnects hosts and switches using 1.28 Gb/s full duplex links. The myrinet PCI host adapter can be programmed to interact directly with the host processors for low-latency communications, and with the network to send, receive, and buffer packets.

Myricom supplies open source software that runs on common hosts and operating systems. This software maps the network periodically to find available paths between communicating hosts. All myrinet packets carry a source-based routing header to provide intermediate switches with forwarding directions. Therefore, myrinet switches do not need to run routing algorithms and maintain a routing table. Because myrinet does not impose a size limitation on its packets, it can easily encapsulate any protocol's packet format (e.g. TCP [8], IP, etc.), thereby providing interoperability. While the simplicity in the myrinet switch offers a low per-port cost, it lacks management capability to maintain robustness in large clusters.

The current myrinet switch is a 16-port crossbar. These ports can be assigned to interconnect either switches or processors, thereby allowing arbitrary network topologies. Depending on the network topology, only a subset of the 16 ports is available to connect compute nodes. Normally, more interswitch connections

means more diverse paths, which can reduce blocking within the switching fabric. However, there will then be fewer ports available to interconnect processing nodes.

The severe cable length restriction is the greatest impediment to creating complex topologies. Myricom has plans of developing an optical interconnect of their switches, but these are not yet available. On the small scale, one can easily build hypercubes and large-dimensional tori. However, our cluster has eight compute nodes and associated myrinet and ethernet switches per standard rack, consuming an area of 2' by 3'. The smallest aggregation of 256 nodes (32 racks), including area for servicibility and to satisfy the fire marshall, requires an area of 16' by 21'. Our goal of scaling to ten thousand compute nodes in the next five years will further increase this area while packaging densities are not expected to increase significantly in that timescale. Cost is another issue to consider—highly connected networks (many interswitch connections) require more switches. Given all this, a two dimensional mesh (or torus) seems to be the best tradeoff in terms of area and cost. It scales in two physical dimensions just as our hardware scales in two dimensions. Myrinet provides incremental growth by adding one or more switches at a time.

The range of workloads that myrinet can support is also of concern. Wormhole routing used by myrinet leads to buffering of messages in the network itself. Thus, large messages can cause blocking and degrade the overall latency performance. Sections 3 and 4 describe our simulation study on the effects of blocking on the performance of parallel applications. Results are presented in Section 5.

2.2 Gigabit ethernet

The most popular Local Area Network (LAN) technology is ethernet. Ethernet has evolved from the 3 Mbps technology, invented by Bob Metcalf in 1973, to the 10-, 100- (or fast), and 1000-Mb/s (or gigabit) ethernet standards of today [7]. Riding on the ethernet popularity current, gigabit ethernet is fast becoming a commodity item and therefore, we believe it can be a cost-effective alternative to interconnect parallel computers. Moreover, there are already discussions of 10- and even 100-gigabit per second ethernet [4], which could provide the next generation parallel computers with a smooth upgrade path to their communication subsystem.

The half-duplex variant of the gigabit ethernet uses the same medium access control protocol (MAC) as the 10- and 100-Mb/s standards. While the carrier sense, multiple access, and collision detect (CSMA/CD) protocol provides adequate performance to ethernet's LAN applications, it incurs unacceptable latency to response sensitive parallel algorithms. Therefore, we only consider point-to-point, full-duplex gigabit ethernet in LAN switches, where packets can be sent without delay. In this mode, a pause protocol (802.3x) is defined to throttle traffic coming from the opposite end of a congested link. Congestion occurs as a result of offered load exceeding the switch service rate, thereby exhausting buffer resource at the congested port.

Conventional routers use switch fabrics that are based on backplane bus and crossbar switch. Busses are not scaleable to higher bit rates because a bus is a shared medium. Crossbars also lack scalability since the cost will grow as the square of the number of nodes. Today, the largest non-blocking switch has 64 ports. Thus, we need to cascade these switches in order to build a cluster larger than 64 nodes. Conventional ethernet switches exchange topological information using the spanning tree algorithm. This algorithm precludes the use of mesh topologies, because it calculates a loop-free tree that configures a single path for each destination. Spanning tree treats redundant links as hot stand-by links rather than as diverse paths. Without diverse paths, cascaded switches will suffer performance bottlenecks due to output port contention. Although this problem can be mitigated by trunking multiple physical links to create a larger pipe, it depletes the already limited port resources and will therefore inflate the per-connection cost.

Ethernet's common frame format provides compatibility between gigabit ethernet and the legacy 10- and 100-Mb/s ethernets. As the backbone to legacy ethernets, gigabit ethernet provides a large pipe to satisfy the growing demands of inter subnet traffic. Because legacy ethernet is shared medium, it will inevitably pass along remnants of collided packets. Therefore, a gigabit ethernet switch must store received packets first in order to prevent forwarding corrupted packets. The store-and-forward processes will affect the end-to-end latency of parallel applications, and its impact would likely worsen as the number of cascaded switches increases.

For these reasons, we believe the applications of a conventional gigabit ethernet switch fabric are limited to small parallel systems. We decided to conduct a simulation study of a 256-node cluster, nevertheless, in

order to evaluate the effects of ethernet's packet framing, inter-frame gap, maximum and minimum packet size, as well as the store-and-forward switching mechanism on the performance of parallel applications.

2.3 Avici terabit switch router

The Avici router uses two direct connection networks [2] as its switching fabric to achieve high-performance, economical scalability, and robustness. The dual fabric connects switching nodes (or line cards) using twelve 20-Gbps full duplex links to form two 3-D toroidal meshes [1]. Figure 1 depicts the packaging of one of the toroidal networks within the Avici router. As shown, line cards of a quadrant are connected via backplane conductors in a 5-cycle in the z-dimension. The x- and y-dimension channels from the line cards connect to the corresponding line card on an adjacent backplane. Figure 1(b) shows how adjacent backplanes are connected to from a folded torus. A folded torus allows uniformly short wires to be used for all connections, thereby lowering wiring costs as well as latency variations. With this arrangement, an Avici router can be incrementally expanded to include up to $14 \times 16 \times 5 = 1120$ line cards. At 16 gigabit ethernet ports per line cards, this configuration represents a parallel systems of $1120 \times 16 = 17\,920$ compute processors.

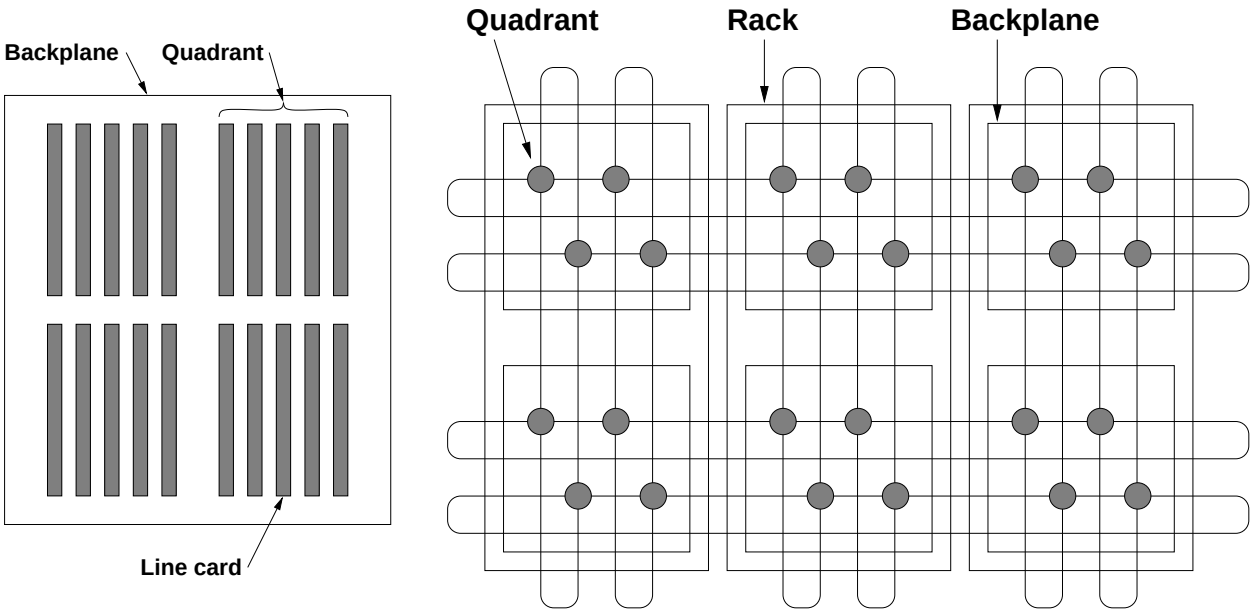

Figure 1. Avici switch router packaging: (a) backplane and (b) system.

Figure 2 lists the formulae for path diversity and other characteristics of several popular network topologies. As shown, a 3-D torus offers the highest path diversity, which is good for load balancing and fault tolerance. In addition, the 3-D torus network offers the smallest average hop-count (D_{avg}) and thus lowest end-to-end latency and jitter between nodes in the network. The 3-D torus also reduces the worst-case channel load (γ_{max}) because of its high switch channel bandwidth to input-line bandwidth ratio over all the channels in the network.

Similar to myrinet, the Avici router uses wormhole routing inside the fabric to achieve low latency. Unlike myrinet, however, rather than buffering the entire message inside the network, the Avici router segments its messages into 72-byte scheduling units or "flits" and exercises a stringent flow control to prevent flit loss. Together with its per- connection buffer management, and the 4-times speedup of its fabric links relative to the line card I/O demands, the Avici router implements an output-buffered virtual crossbar to eliminate the blocking problem in wormhole routing. Because of the huge speed mismatch between gigabit ethernet and the fabric link (1:20), the Avici router will store incoming gigabit ethernet packets before forwarding to prevent buffer underrun. Unlike conventional switches, however, the number of store-and-forward operations in the Avici reaches an upper bound of two, once at the incoming and the other at the outgoing gigabit ethernet port. Moreover, because the Avici router is designed for telecommunications applications, it is

	D_{avg}	$\gamma_{\max}$	Path discovery
2-D torus	$\frac{1}{2}\sqrt{n}$	$\frac{1}{8}\sqrt{n}$	$\binom{\sqrt{n}/2}{\sqrt{n}/4}$
3-D torus	$\frac{3}{4}\sqrt[3]{n}$	$\frac{1}{8}\sqrt[3]{n}$	$\binom{3\sqrt[3]{n}/4}{\sqrt[3]{n}/2}\binom{\sqrt[3]{n}/2}{\sqrt[3]{n}/4}$
Hypercube	$\frac{\log_2 n}{2}$	$\frac{1}{2}$	$\frac{\log_2 n}{2}$
Benes	$2\log_k n + 1$	1	n
Crossbar	2	1	1
Butterfly	$\log_k n + 1$	1	1

Figure 2. Distance, channel load, and path diversity for several network topologies.

extremely robust and has extensive SNMP-based management capabilities, a feature that is essential to building reliable large parallel systems.

3 Simulation methodologies

Simulation methodologies have long been accepted by network researchers to study the intricate dynamics of network traffic and protocols, offering in-depth understanding of performance issues that are otherwise unobtainable through experimental measurements. We adapt and extend simulation packages that are commercially available and in the public domain. We designed our simulation study to capture the characteristics of network protocols and the switch architectures that can affect the performance of working parallel algorithms. Once we understand the parallel computing requirements on the interconnection network, we plan to address the performance issues involving the host adapter, its device driver, and the OS bypass mechanisms in a continuation study.

All three of our simulation codes use an identical interface layer which serves to isolate the parallel algorithms from the details of packet transmission. This allows us to keep one set of algorithms, and simply relink with a different message passing library for each different simulator. The interface layers also produce identical diagnostic output which we parse with a single set of scripts to extract data and to produce plots.

Because many users share the use of a parallel computer system, each will allocate a number of processors from the pooled resources to solving his or her problems. Therefore, we chose to conduct our simulation study using a 256-node network to represent a typical user's problem solving environment. The following subsections discuss the intermediate interface layers and the details of the low-level simulators.

3.1 MIL3 Opnet

MIL 3's Optimized Engineering Tools (Opnet) [11] is a comprehensive engineering system capable of simulating large communication networks with detailed protocol modeling and performance analysis. Its features include graphical specification of models, event-scheduled simulation kernel, and hierarchical object-based modeling. We selected Opnet to simulate the conventional gigabit ethernet switch, because Opnet has an existing model that simulates the gigabit ethernet protocol. Furthermore, Sandia has a runtime license purchased for previous projects. The following paragraphs describe our hierarchical object-based modeling effort to evaluate the performance of conventional gigabit ethernet switch as cluster-interconnect.

On the top level, we used Opnet's network editor to compose our network using components such as switches, nodes, and links. Figure 3 depicts the 256-node simulation network that we constructed for this study. As shown, this network consists of five conventional gigabit ethernet switches, 256 compute nodes, and full-duplex links to interconnect them. Since the largest conventional switch today can support up to 64 ports without blocking, we populated four switches each with 64 end-nodes. They are in turn connected via a fifth switch. We choose a star topology because it offers the lowest hop count (3 in this case) between the farthest nodes in the network, thereby minimizing the overall end-to-end latency.

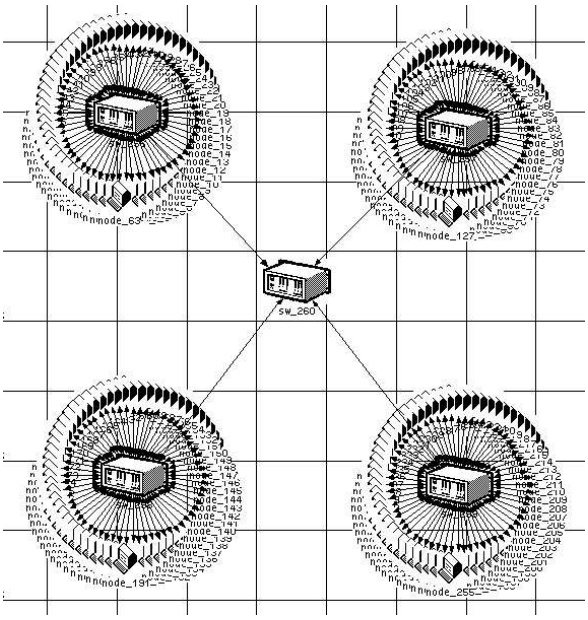

Figure 3. Opnet top-level network domain: the network topology.

At the next level, we used Opnet's node editor to construct our compute and switch nodes. As shown in Figure 4(a), a compute node consists of a module to run the parallel applications models that we wrote, a gigabit ethernet protocol entity, a transmitter, and a receiver. Figure 4(b) depicts the components of a 16-port switch. At the center of the switch is an existing Opnet model that can be instantiated to simulate popular conventional switch architectures. We derived a shared memory switch model by selecting a service rate to be able to handle incoming packets from all ports without blocking. As shown, a gigabit-ethernet switch port is built using a gigabit ethernet MAC, a transmitter, and a receiver module. We constructed 65 such ports for our 65-port switches that we used in our study.

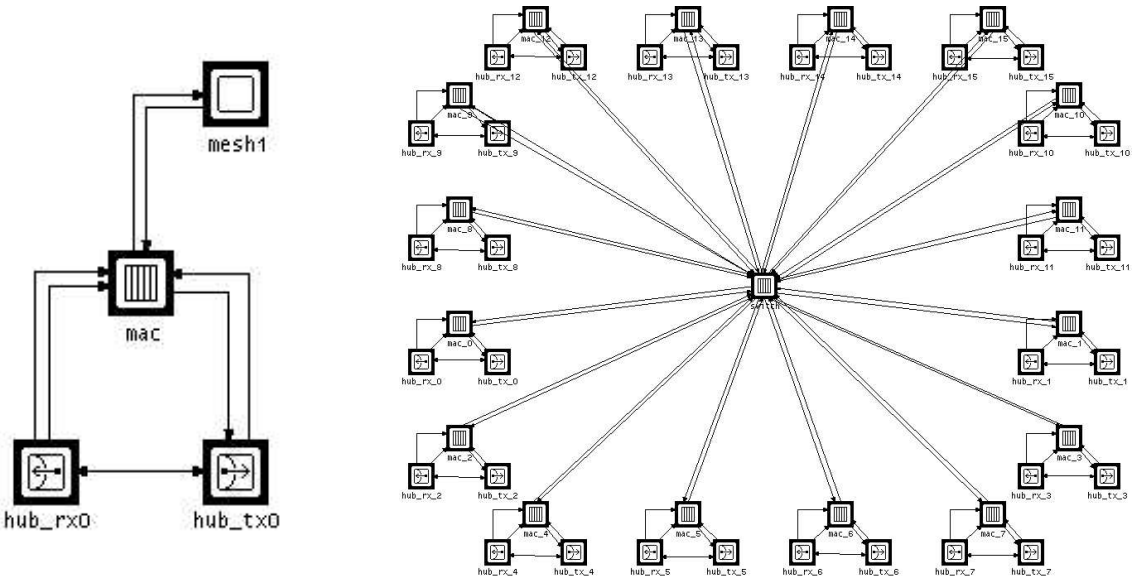

Figure 4. The Opnet node domain: (a) end system, (b) 16-port switch.

The process editor is at the bottom of the object hierarchy. This GUI allows us to design our parallel applications using state transition diagram (see Figure 5). As shown, after initialization, a process running

the parallel code can be in one of the following three states depending on the interrupting event. In XMT-state, a process sends messages to all of its neighbors and then enters into a WAIT, where the process will sleep until interrupted by a packet-arrival event. The packet-arrival event advances the process into the RCV-state, where state information will be updated and diagnostics logged. When the process has received a message from all of its neighbors (as indicated by the process's state information), it enters the compute phase. We simulate the compute phase by putting the process to sleep for a random amount of time. At the end of the compute phase, a self- interrupt wakes the process in order to proceed with sending another round of packets to its neighbors.

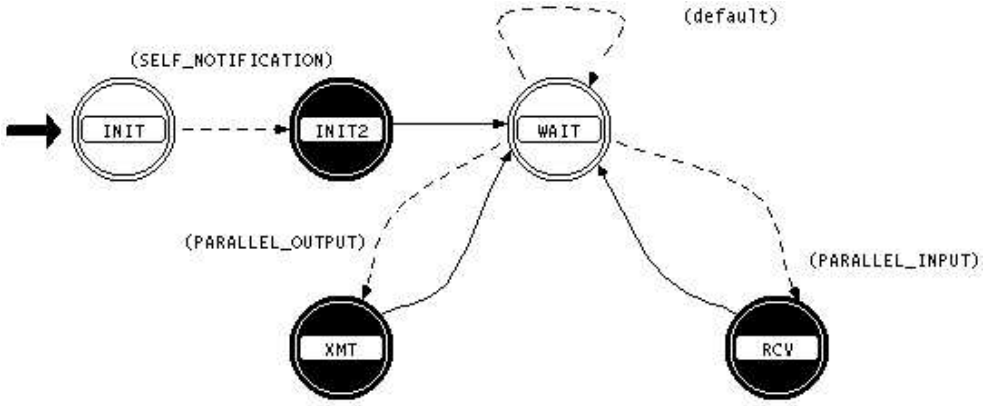

Figure 5. The Opnet process domain: state transmission diagram.

We coded the behaviors of the parallel algorithm for each state using a text editor provided by Opnet (see Figure 6). Our parallel models access Opnet's packet manipulation, event scheduling, and interrupts handling framework via a set of kernel procedure calls. Instead of using Opnet's statistical analysis package, we generated our own diagnostic output such that we can use the same parsing and plotting tools for the myrinet and TSR studies as well. We developed five parallel algorithms for this study: token passing, 2-D mesh, 2-D torus, `fan_in`, and `fan_out`. Details of these code algorithms are presented in Section 4.

3.2 Avici simulator

Allen King and coworkers at Avici wrote a simulator [5] to be used in planning the switch hardware they built. The first version of that code is accurate for their "generation one" product, while the second version we used is almost correct for the generation two hardware we bought at Sandia. "Almost" because Avici never got around to updating their simulator to reflect a very fundamental hardware redesign. The actual hardware incorporates two parallel fabrics, while the simulator posits the existence of only a single fabric. The impact of this on the simulations is that message traffic has fewer routes from which to choose to avoid congestion, and latencies induced by heavy traffic will likely be longer in the simulator than in real life.

Their simulator has the capability to generate random traffic at line cards to probe the response of the switching fabric to different network conditions. We, however, chose not to use that, but instead inserted hooks into the simulator by which we could feed our own traffic patters into the switch. Allen suggested the points to add a very few calls: sendPacket() inserts a packet into the fabric, while receivedPacket() is called up from the fabric to notify our modules of the receipt of a packet at a destination line card. The function nextFlitTime() notifies our code that the simulator has advanced in time a fundamental flit unit (30 ns in our hardware), and we use that notification to fire any pending timed events. Various other calls are used by the fabric and the application code modules to notify each other of initialization, completion, and to acquire or change fabric parameters.

One major lacking point in the Avici simulator is the modeling of the network beyond the line card. In particular, if the line card contains 16 gigabit ethernet ports (as in our case), each port has limitations on the rate at which data can be sent through it, and the ethernet protocol specifies a frame gap, and minimum and maximum packet sizes which must be enforced. As this choice of medium is very important to the overall accuracy of the simulation, we model the line card interfaces ourselves.

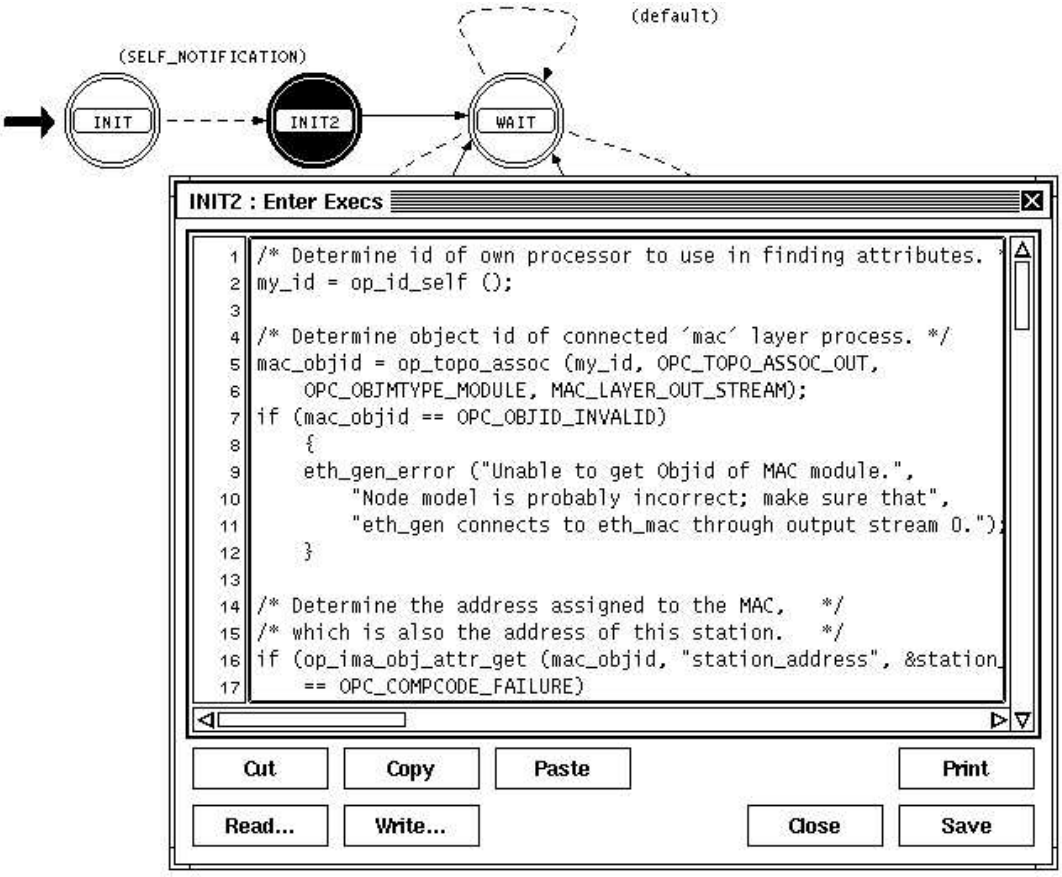

Figure 6. The Opnet process editor: model code development.

The fundamental data transfer unit modeled in the code is called a "packet," which is the maximum unit which can fit on the external interface. In the case of gigabit ethernet, that is 1500 bytes (or 9000 depending on how liberal your host interface vendor decides to be). The upper application layers do not see packets, they only speak in terms of a "message," which is the full amount of data a code writer wants to send at once. The `send()` routine is called from, say, `MPI_Send()`, and fragments the message into a series of packets. These packets are scheduled to be sent from the host interface across the gigabit ethernet cable to the Avici switch according to a global array describing when the line is in use in that direction. The line is marked busy until a time which is calculated from physically-induced parameters: frame header, frame gap, line bit rate, and packet length.

The `send()` routine schedules each of these fragments to enter the Avici fabric as it appears at the other end of the ethernet cable, by registering an event for each packet. Our interface will fire the event when it sees that the simulator time has advanced to the delay specified in the event. Here a timestamp is marked on the packet and it is injected into the fabric by calling `sendPacket()`. This routine returns a unique identifier we can use as packets are received to match them up with the ones we sent. If zero is returned, however, the input buffer on the Avici line card is full due to fabric congestion and we delay the packet until the next flit, continually trying until there is room. This simulates an infinite buffer in the network, which is required for lossless transfers. In reality the buffer will not be infinite, but the application will be put to sleep by the network driver when it receives a gigabit ethernet flow control packet until the network congestion has passed. This should give the same timing results as an infinite buffer.

As each packet is received at a line card inside the Avici router fabric, their simulator calls a routine `receivedPacket()` in our interface code which is given the identifier of the packet. The routine searches through the list of packets which have been sent out to find the matching structure. Then it ensures that the flit which was received from the simulator is indeed the next one that was expected for in-order deliver of

the entire packet. If the flit is the final one in the packet, a receive statistic is printed (for just the fabric part of the transfer), and delivery of the full packet to the compute node is scheduled in the line card's queue for that node. A delayed event will signal the packet arrival at the node, printing another transfer statistic, this time for the full path from the sending node. There the packet is deleted from the list of pending transfers.

We also added a multicast capability to their simulator by copying a packet and injecting it into the fabric multiple times to different destinations, as if the copying had happened at the line card, not at the source host. Avici has a similar, but more efficient, multicast capability in their hardware that is not implemented in their simulator.

3.3 Myrinet simulator

The myrinet simulator [6] was initially developed by Chen-Chi Kuo as a graduate student in the Computer Science Department at the University of Utah to be used in their Paint simulator. Paint is a detailed component-level simulator for HP-PA processors and associated caches, memory, and interface modules. We adapted just the myrinet part of the simulator by cleaning up the code so that it would compile cleanly on Posix-compliant systems using either 32- or 64-bit processors. The simulator does not include a task or event handling mechanism, so we wrote a naive scheduler based on a linear list sorted by next event time. Much of the packet tracking and upper layer framework written for the Avici simulator interface could be reused in the myrinet simulator.

The myrinet simulator requires three data files to describe the modeled hardware. A parameter file gives the basic numerical constants: number of switches, number of nodes, number of processing elements, and so on. A full table is given in Figure 7, along with the values used for our 256-processor cluster simulations. The units for time are all "flits," which is the length of time it takes for one byte to progress through the Myrinet fabric. Initially the Paint simulator used the system clock period as the basic unit, then expressed the myrinet clock units as a multiple of that basic system clock period. We have set the parameter SpeedFactor equal to unity as our model does not consider a processor or system bus, but only network components. Hence the myrinet flit is a more basic unit.

```
# topology
numOfProcessor 256
maxNumOfPorts 16
numOfSwitch 32

# assume 0.6c, 25m cables, so 138 ns delay
propDelay 22

# assume all SAN cables, 100 ns delay through each switch
fallThruDelay16 16

# do not confuse "host" speed versus myrinet speed.  Only conversion
# that happens is in the interface, using the 160 MHz clock rate.
SpeedFactor 1

# assume 44 symbols (= 2 * 160 MHz * 138 ns) outstanding in the round
# trip path, and a hysteresis factor of 32 symbols (it was 16
# in version 1).  Round up to likely multiples of 2.
buffer_kg 64
buffer_ks 64
buffer_h  32
```

Figure 7. Myrinet parameter file example.

Our current CPlant myrinet hardware is their second generation product, which has a theoretical bandwidth of 1.28 Gb/s in each direction per link. This comes from a 160 MHz clock period and an 8-bit wide

data path. Inverting the clock rate gives the value for a flit: 6.25 ns. The other hardware parameters are expressed in flit units, and are derived from Myricom's documentation or empirical tests. We assume only SAN links in the entire cluster, which are faster to switch than the LAN links we actually use in the system itself, as the simulator only allows for homogeneous switches.

The second file is the topology specification showing how the myrinet interface cards are connected to the switches, and how the switches are connected to each other to form a network. An abridged topology is given in Figure 8 for a 256-processor cluster. Each switch is listed on one line, followed by a list of connected elements, denoted by `P`⟨*processor*⟩ for compute nodes, `S`⟨*switch*⟩`.`⟨*port*⟩ for a connection to another switch, and `D` for dangling (unconnected) ports. The 256-processor version of the file was created by a few-line awk script to reproduce our connection pattern.

```
S0: P0 P1 P2 P3 P4 P5 P6 P7 S7.10 S7.11 S1.8 S1.9 S24.14 S24.15 S8.12 S8.13
S1: P8 P9 P10 P11 P12 P13 P14 P15 S0.10 S0.11 S2.8 S2.9 S25.14 S25.15 S9.12
    S9.13
S2: P16 P17 P18 P19 P20 P21 P22 P23 S1.10 S1.11 S3.8 S3.9 S26.14 S26.15 S10.12
    S10.13
S3: P24 P25 P26 P27 P28 P29 P30 P31 S2.10 S2.11 S4.8 S4.9 S27.14 S27.15 S11.12
    S11.13
 ⋮
```

Figure 8. Myrinet topology file example.

We investigated a few topologies for the Myrinet topology. All have in common the feature that eight compute nodes are attached to each 16-node switch. This leaves eight more ports for interswitch links. The first topology connected physically neighboring pairs of switches with a single link, requiring thus 31 more cables for the entire cluster. Our system administrator built this for us and we quickly discarded it as being inadequate. The next attempt was to create a five-dimensional torus of switches, as pictured in Figure 9(a), where each vertex represents a switch to which eight compute nodes are further attached. Ignoring the dotted line for the moment, each line represents a connection between switches, and all together they use up only five of our extra eight ports per switch. The dotted line, then, is one example of the sixth link we added to the switches. Showing them all would muddle the figure too much. The dotted lines connect "opposite" corners of the hypercube (by identifying binary switch number x with $\bar{x}$) and reduce the maximum hop count between any two switches from five to three. The routing we use is the bitwise variant of the standard x-then-y.

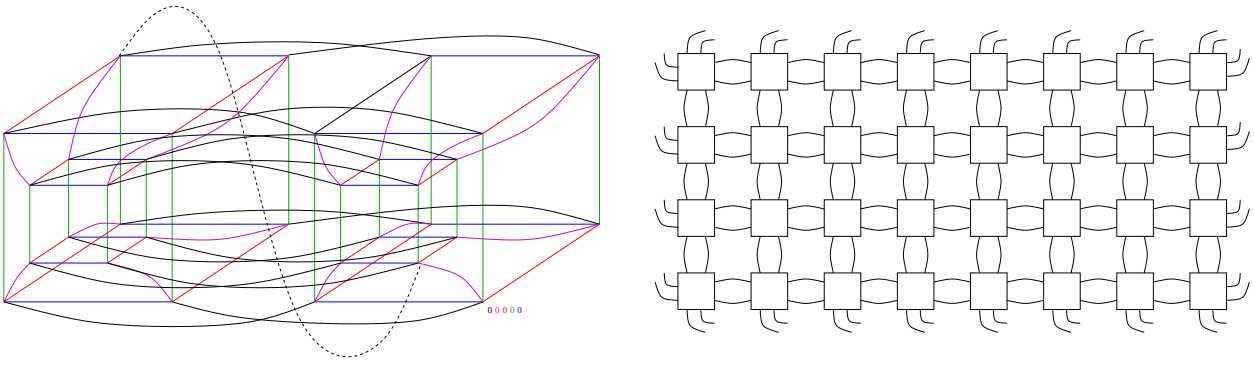

Figure 9. Myrinet toroidal topologies: (a) 5d, (b) 2d.

We did away also with this hypercube topology due to its non-scalability. A sixteen port switch runs out of connections at 1024 nodes, and the physical link distance limits the size before the port limit is reached, as discussed in Section 2.1. Thus we use only a two-dimensional torus to connect the nodes to generate the results presented in this paper. A diagram of the topology is given in Figure 9(b). Each box is a switch with eight compute nodes, and the remaining eight ports are consumed by two (bidirectional) links in each

of four directions to neighbors on the torus, giving a hop-to-hop bandwidth of 2.56 Gb/s. Notice there is no physical constraint arising from the chopped links appearing at the edges of the diagram since we use the same wrapped torus links as were used in the Avici layout.

The final file needed by the simulator specifies the routing used by each processor to send a message to each other processor. The 256-processor simulation version has 65536 lines, each of which is a route for a source/destination pair of nodes. The route is given as a list of single-character port numbers specifying the output port to choose at each switch encountered along the way from the source to the destination. A few sample entries are shown in Figure 10. The chosen routes attempt to balance traffic across the parallel links by choosing a link based on the even or odd character of the sum of the destination node's x and y positions in the mesh.

```
p0 p14 b6
p0 p40 9990
p14 p0 80
p14 p40 9990
p40 p0 aaa0
p40 p14 99996
```

Figure 10. Myrinet route file example entries.

As the topology and route information is much too tedious to enter manually, we adapted a program written by us earlier for California's CPlant to generate the routes used in the system. Important parts of the configuration for that program are shown in Figure 11.

The same application codes designed for use with the Avici simulator couple directly to the interface we wrote to communicate with Utah's flit-level myrinet simulator, and similar parsing tools can be used to deduce statistics from the output of simulation runs.

4 Parallel code algorithms

Accurate characterization of network performance is a complex task. Simple numbers such as minimum latency or maximum bandwidth are not sufficient metrics to enable cross-technology comparisons. We augment these basic numbers with results from computational core algorithms from real parallel codes in use at Sandia. Results from the tests are deferred until the following section.

4.1 Basic diagnostics

First we present the *de rigeur* analyses of network performance. These are useful not only for the baseline information they provide, but also as debugging checks of the simulation methodologies presented in the previous section.

A code entitled `token_pass` is our simplest test. It arranges the participating processors in a virtual loop than iterates the passing of a "token" around the loop a certain number of times. Each processor awaits a message from its neighbor to the left, then delays a bit to simulate processing time, then sends a message to its neighbor on the right.

The code is shown in Figure 12, as it is the smallest example of an algorithm which uses the simulator interface written for this project. By changing the size of the token (variable `payload` in the code) to be large, we can perform accurate bandwidth measurements. By setting the payload to zero, we find the minimum message latency. Since only two processors at a time are ever involved in a communication, there are no contention effects to filter out from the results.

The codes `fan_in` and `fan_out` are generated from the same source file with different `#define` settings, as they perform quite similar functions. The former simulates a global reduction whereby each processor sends a message to the "host" processor (number 0 in our case). The sends are staggered slightly ($\lesssim$ 10ns) to avoid odd synchronization effects in the switches, and to simulate real life in which it is impossible to do clock-synchronized sends on a distributed machine. This test is good for ensuring that no messages are

```
#
# connectivity description file, SU1--SU16, 256 nodes total
# Using the new M-2LM SW16 S1aN & LAN switches
#
sus 16
nodes 16  # per su
# this defines an ordering on switches
switches 32 1a 1b 2a 2b 3a 3b 4a 4b 5a 5b 6a 6b 7a 7b 8a 8b 9a 9b 10a 10b
11a 11b 12a 12b 13a 13b 14a 14b 15a 15b 16a 16b

# where the nodes are connected
su 1 node  1 switch 1a port 0
su 1 node  2 switch 1a port 1
 ⋮
su 1 node 16 switch 1b port 7

su 2 node  1 switch 2a port 0
su 2 node  2 switch 2a port 1
 ⋮

# how the switches are connected to each other
switch 1a port 8 switch 1b port 8
switch 2a port 8 switch 2b port 8
 ⋮
switch 16a port 8 switch 16b port 8

switch  1b port 9 switch  2a port 9
switch  2b port 9 switch  3a port 9
 ⋮

# routing for things that aren't directly connected, just provide
# the next hop here.  Means any switch beyond 1b gets there by
# going through 1b, for example.
route 1a using > 1b
route 1b using > 2a

route 2a using < 1b
route 2a using > 2b
route 2b using < 2a
route 2b using > 3a
⋮
```

Figure 11. Myrinet route configuration file example.

dropped or corrupted under such an extreme congestion level. In the reverse mode, `fan_out` has the host processor sending staggered messages to all the other processors. This tests the flow control mechanism in the other direction. Performance numbers from `fan_in` and `fan_out` models the startup and shutdown events of parallel applications, which often include global broadcast and reduction phase.

The code `mesh` simulates a computational kernel from a two-dimensional finite element calculation. This class of structured grid codes are very common among the large ASCI-scale calculations being performed today at the laboratories. The processors are laid out in a virtual two-dimensional mesh, and each processor

```
/*
 * token_pass.c - implements a simple token-passing algo around the
 * ring.  It stops after the specified number of iterations (LOOPS).
 * <wyckoff@ca.sandia.gov>
 */
#include <stdio.h>
#include <stdlib.h>
#include "interface.h"

/*
 * Time each processor holds the token, in absolute time (sec).
 */
#define DELAY 30.0e-6

/*
 * Size of token to pass, in bytes.
 */
static const int payload = 1;  /* 128kB should take 819.2 us */
static int numproc;

/*
 * Number of times to go around, and max processors to use.
 */
#define LOOPS 3
#define MAX_NUMPROC 256

/*
 * Local variables, keep track of value in token, and who's got it.
 */
static int loops;

/* data for a callback event */
struct cb_data {
    int src;
};
static struct cb_data *new_data(int src);
static void schedule(void *data);

/*
 * Initialize the algorithm, maybe sending the first packet.
 */
void
parallel_init(void)
{
    struct cb_data *data;

    loops = 0;
    numproc = get_numproc();
    if (numproc > MAX_NUMPROC)
        numproc = MAX_NUMPROC;
    data = new_data(0);
    schedule(data);
}
```

```
/*
 * Timer fires to do something.
 */
static void
schedule(void *vdata)
{
    int to;
    struct cb_data *data = (struct cb_data *) vdata;

    if (data->src == 0)
        if (loops++ == LOOPS) {
            printf("schedule: loops = %d, done\n", loops);
            return;
        }
    to = (data->src + 1) % numproc;
    printf("%d: parallel_schedule: send out to %d\n", data->src, to);
    send(data->src, to, payload);
    free(data);
}

/*
 * Accept a packet from the network.
 */
void
parallel_input(int src, int dst, int len)
{
    struct cb_data *data;

    printf("%d: parallel_input: from %d, len %d\n", dst, src, len);
    data = new_data(dst);  /* new source is one who just received it */
    event_add(DELAY, schedule, data);
}

static struct cb_data *
new_data(int src)
{
    struct cb_data *data;

    if (!(data = (struct cb_data *) malloc(sizeof(struct cb_data))))
        error("new_data: alloc cb_data");
    data->src = src;
    return data;
}
```

Figure 12. Source code of `token_pass` algorithm.

will communicate with its immediate neighbors in both the x and y directions. Most processors thus have four neighbors while those on an edge have three, and the four corner processors each have only two neighbors. The code performs a number of iterations of computation and communication cycles, which represents the real code's explicit time stepping algorithm as it solves the generalized set of equations:

$$\frac{\partial \mathbf{f}}{\partial t} = F\left(t, \mathbf{f}, \frac{\partial \mathbf{f}}{\partial x}, \frac{\partial \mathbf{f}}{\partial y}\right).$$

This is converted into the discrete form

$$\mathbf{f}^{n+1} = \mathbf{f}^n + \Delta t\ F\left(t^n, \mathbf{f}^n, \frac{\partial \mathbf{f}^n}{\partial x}, \frac{\partial \mathbf{f}^n}{\partial y}\right),$$

where $\mathbf{f}$ is represented as an array of values, decomposed across the participating processors in the system. The derivatives above are written as only first-order, but could be of higher order as well in the real code. Also some values on the right hand side might be taken at timestep $n + 1$ in which case the algorithm is implicit. This does not change the basic structure of the problem, but increases the storage requirements and communication of each processor. Each processor calculates the function F for the processors in its domain, which we simulate by a time delay. The communication phase is where the processors trade information about the cells along shared boundaries to propagate recently computed data to neighboring cells. We simulate this by the transfer of point-to-point messages in the network fabric. The effects of load imbalance on the algorithm are not simulated, as good codes strive to avoid such imbalances, and the effect would be to lessen the importance of network performance on the overall algorithm. We also do not model startup and shutdown events which might include global broadcast and reduction phases, as these are not standardized across codes, do not constitute the core of the algorithm, and would only serve to muddle the results.

We have made some modifcations to the `mesh` code to model a `torus` topology, which is necessary for a code simulating periodic boundary conditions and arises in calculations on a spherical domain or in free space, for instance. In the toroidal topology, each processor always has four neighbors. Another modification is to change the dimensionality from two to three. This has the effect of increasing the number of neighbors from four to six on every processor (except boundaries in the mesh case), and is more representative of advanced codes which simulate complex solid structures.

5 Results

Our results are presented in order of the algorithms we used to test the networks, followed by a summary of all the tests.

5.1 Technology characterization

We ran the `token_pass` code with a one-byte payload to determine the minimum message latency between neighboring nodes in a 256-member virtual ring for all three technologies. As mentioned earlier, since only two processors at a time are involved in communication, there are no contention effects to filter out. We compiled our results and listed the minimum, maximum, average, and standard deviation values for each study in Figure 13. As shown, the myrinet technology delivered very good latency and jitter (latency variation). Jitter in the absence of congestion is a function of network topology; it reflects the difference in distance between `token_pass` neighbors in the network.

	Min	Avg	Max	Stdev
Myrinet	0.388	0.427	0.869	0.093
Avici fabric	0.180	0.186	0.330	0.024
Avici GigE	1.380	1.386	1.530	0.024
Conventional GigE	1.564	1.595	3.532	0.244

Figure 13. Minimum message latency results. Units are in μs.

As a result of the Avici's higher fabric-speed and path diversity in the 3D-torus topology, myrinet's performance is still inferior to Avici. We had to configure a 2D torus for myrinet due to its physical constraints. As expected, since the Avici gigabit ethernet routes its packet through the Avici fabric, it had inherited the fabric's low jitter. The increases in its latency amounts to the sum of two transmission delays, when the packet arrives at the input and when it reaches the output of the Avici gigabit ethernet line cards. The store-and-forward switching is necessary here in order to prevent buffer underrun at the outgoing line card switch due to the large speed mismatch; a fabric link is 20 times that of the gigabit

ethernet speed. Furthermore, because the ethernet standard imposes a minimum packet size of 64-byte, the original one-byte message was padded before transmission. Therefore, each of the transmission delays is actually $64 * 8/1000 = 0.512$ μs. Two times this value is roughly the increase seen in comparing to the fabric's latency.

The existing Opnet switch model does not emulate switch processing-delay, consequently, the latency values that we obtained through simulation (Figure 13 row 4) are better than measured statistics. In our star topology, a packet will traverse either one or three hops depending on whether the immediate neighbor is on the same switch or not. Since switches today typically incurs ~ 10 μs of processing delay, the values listed in Figure 13 would have an additional latency of 10μs at the lower bound, and 30μs at the upper bound, making the performance of conventional switch the least favorable in comparison.

Using `token_pass` and a 15 MB message, we measured throughput for each technology to verify the correctness of our simulation code. We choose that message size because it is large enough to fill the end-to-end communication pipe, a criterion necessary for throughput measurements. The end-to-end communication pipe is the product of the theoretical bandwidth and the round-trip-time (RTT). The simulation throughput values and their corresponding theoretical bandwidth are listed in Figure 14. As shown, they are in close agreement.

	Theoretical	Simulation
Myrinet	1280.000	1279.947
Avici GigE	980.000	975.610
Conventional GigE	980.000	974.301

Figure 14. Throughput results. Units are in Mb/s.

5.2 Fan-in and Fan-out

Figure 15 plots the maximum, minimum, average, and standard deviation latency results against message sizes from our `fan_in` study. The simulations had 255 sources each sending one message to a single destination, thereby causing contention at the destination host machine.

As shown, myrinet performance is roughly 20% better than both the conventional and the Avici gigabit ethernet, because it has an effective bandwidth of 1.28 Gb/s as opposed to the 0.98 Gb/s of gigabit ethernet. The simulation ends when the last message is fully received, and is dictated by the bandwidth in the channel from the network to the receiving host, thus we can compare the maximum latency values from the plots to see the bandwidth effect:

$$\begin{array}{lrcll} \text{Bandwidth ratio} & 0.98\ Gb/s & / & 1.28\ Gb/s & = 0.77 \\ \text{Maximum latency ratio} & 3300\ \mu\text{s} & / & 4300\ \mu\text{s} & = 0.77 \end{array}$$

at large message sizes.

Figure 15(d) demonstrates that the performance of the Avici fabric, considered without the gigabit ethernet line cards, is over an order of magnitude better than other technologies, again because of its high fabric speed and path diversity. Therefore, we believe that subscribing the bandwidth of one switch node to support only 16 gigabit ethernet ports is overkill. This is a point worth pursuing in a future study, because more network ports per Avici switching node would mean increased scalability to beyond the current 17 920 processing node limitation.

Figure 16 depicts the latency results of the `fan_out` studies in a similar fashion, latency on the y-axis versus message sizes on the x-axis. In the `fan_out` study, we sent different size of messages from a source to all destinations. As shown in Figure 16(b), the conventional gigabit ethernet switches offer the best end-to-end latency by far, because these switches implement multicast in hardware, where a multicast packet is referenced and sent simultaneously to all multicast members. On the other hand, wormhole routing emulates multicast in software; this mechanism requires a source host to send a multicast packet multiple times, one for each multicast destination. Therefore, myrinet exhibited the worst latency performance. The Avici fabric fared better because of its higher aggregate bandwidth and diverse paths. Consequently, gigabit ethernet based processors node on the Avici cluster also performed better than myrinet.

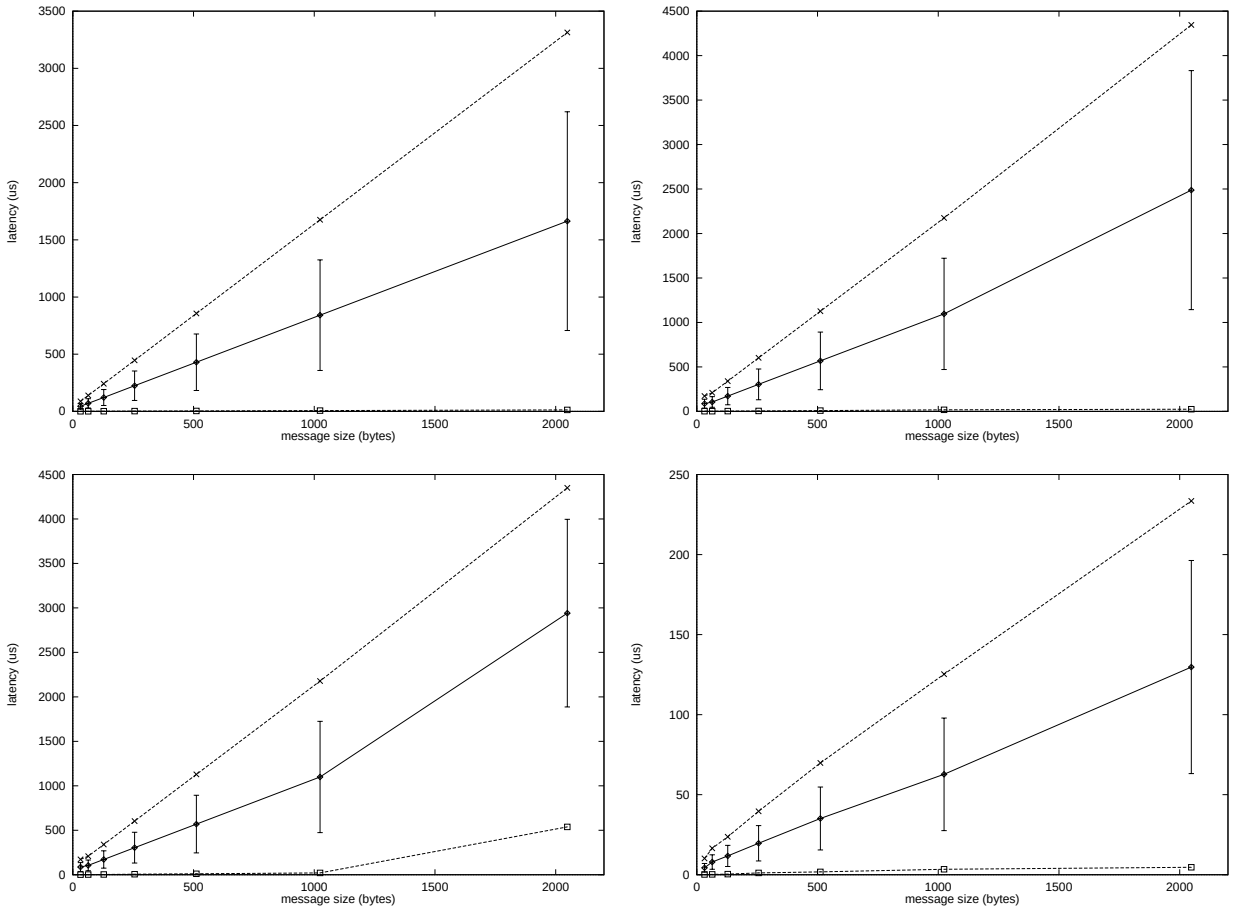

Figure 15. `fan_in` latency statistics for: (a) myrinet, (b) conventional gigabit ethernet, (c) TSR gigabit ethernet, and (d) TSR internal fabric.

5.3 Mesh and torus

The results for both `mesh` and `torus` algorithsm are included in this section as the codes are identical save for the extra edge connections in `torus`. The same results will also be seen in both.

Message latency

The first data we present is a message latency. All messages sent (and received) are recorded with timestamps across all the iterations of the algorithm. The data in Figure 17 show the average time for a message to pass through the respective network, with bars to show one standard deviation. Also shown are the maximum and minimum times. The values used to generate these plots are shown in Figure 18 as well. From the left-hand column of the plots, it is easily seen that the maximum message transfer time can be up to an order of magnitude more than the average in the case of myrinet and conventional gigabit ethernet. Only with the Avici switch are the numbers more reasonable. These maximum numbers tend to pull up the averages.

Also from the close-ups in the right-hand column, the average transfer times for both the Avici and for myrinet are seen to be similar, while the conventional ethernet is larger due to the bottleneck at the second-stage switch in the center of the topology. It would be reasonable to use trunked links from the first-stage switches to the central switch to provide improved bandwidth and alleviate the bottleneck, but this type of scalability will not go too far as commodity switch vendors only provide small numbers of ports per switch. A fat tree using multiple switches is a possible, but expensive, alternative, and may also not reach high node counts as the switches directly connected to hosts will run out of ports in that case.

Up to a message size of 256 bytes, the myrinet network delivers average latencies about 1 μs lower than does the Avici ethernet. The y-axis intercept of the myrinet average line is about 1 μs while that for the

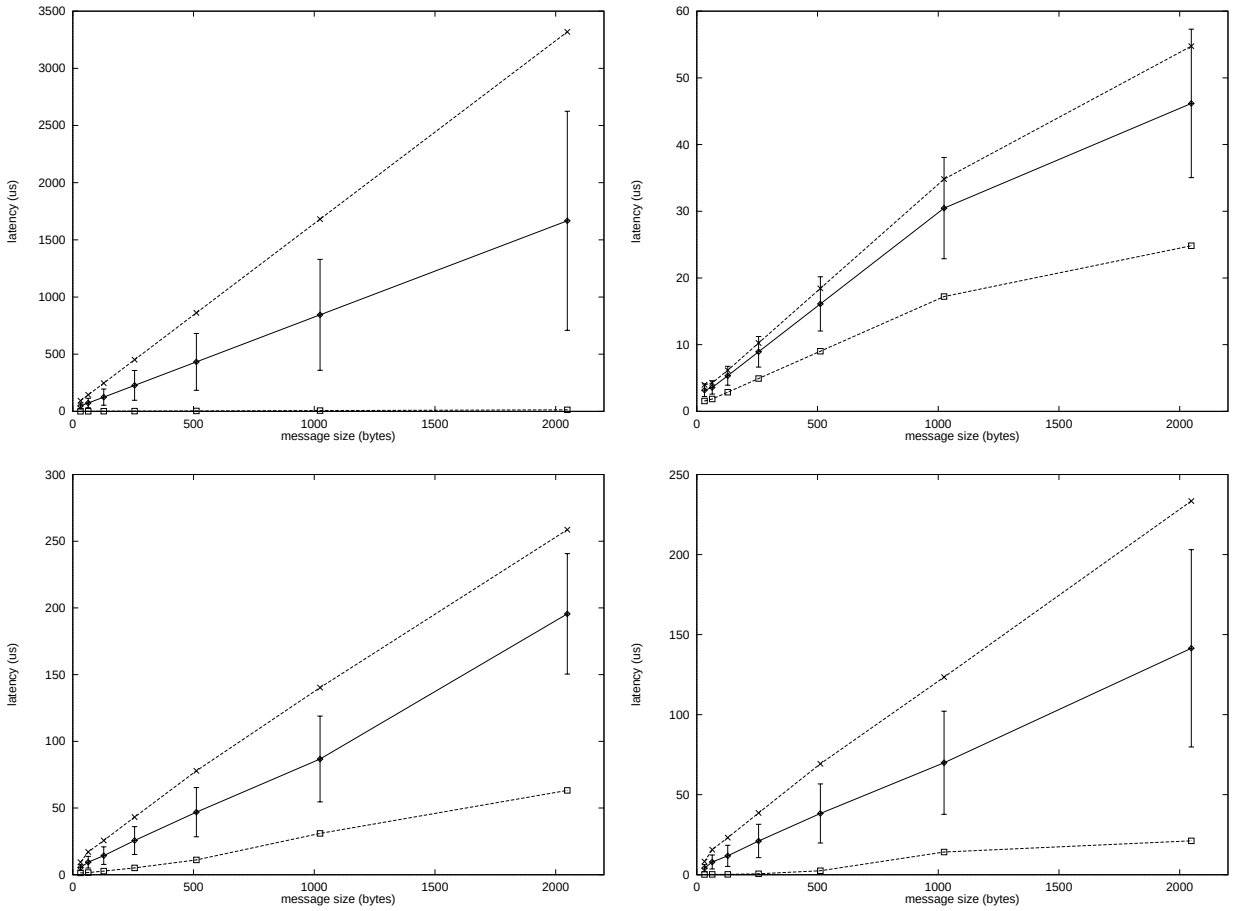

Figure 16. `fan_out` latency statistics for: (a) myrinet, (b) conventional gigabit ethernet, (c) TSR gigabit ethernet, and (d) TSR internal fabric.

Avici is 2 μs. This is due to the latency induced in the Avici switch core itself, which we measure to be about 1 μs longer than myrinet for a shortest-path message. Myrinet switches add about 300 ns per hop, with an average of 3 hops per route in a 8×4 two-dimensional torus, to give the y-intercept seen there.

In the large message extreme, a 2 kB packet takes on average 59 μs to transfer through the Avici ethernet network, or 76 μs to transfer through the myrinet network; however, the worse case transfer time is a factor of eight greater for the myrinet, almost as bad as in the conventional gigabit ethernet network. Note that these are not raw transfer times, but the result of the interactions with transfers between other pairs of nodes on the network. This leads us to conclude that the effect is from the blocking induced by obstructing messages in the network traffic. The Avici switch is configured to be non-blocking by its extreme path reduncancy and the fact that we do not overload the ports on each line card, so any difference between the maximum message transfer time and the average is due to output port contention, *i.e.,* when multiple messages are waiting to enter a single destination host. In the case of the conventional gigabit ethernet, messages may be blocked at the output ports of each of the up to three switches in the path from the source to the destination. The case for Myrinet involves up to six switches, but the bottleneck is not as great as in ethernet due to the multiple routes and full crossbar behavior of the switches.

The same types of plots for the related algorithm, `torus`, are shown in Figure 19, with the raw data in Figure 20. No solid conclusions can be drawn from the differences, which are well inside of one standard deviation. The algorithmic difference is that slightly more communication is occuring, and it is becoming more regular in that each processor talks to exactly four neighbors in `torus`. The underlying network is identical in both algorithms. Perhaps this regularity is of benefit to the myrinet network, which shows a decrease in average and maximum latencies and their standard deviations for all message sizes. Similarly

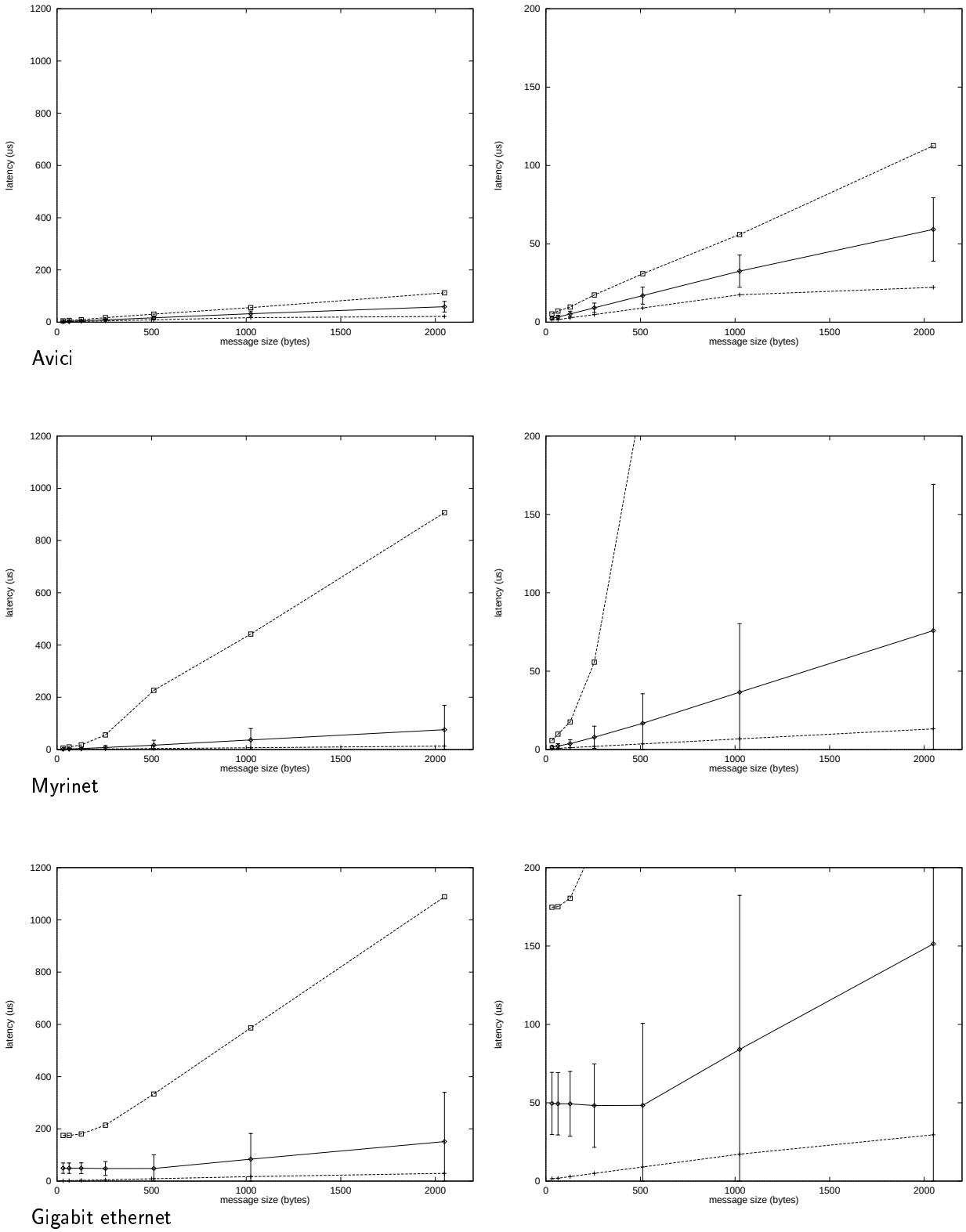

Figure 17. Mesh simulation message latencies as a function of message size, with standard deviation around the average. Maximum and minimum values are plotted as well. Large scale in left column, close-up in right.

Avici

Size	Min	Avg	Max	Stdev
32	1.38	2.57	5.28	0.87
64	1.65	3.27	7.11	1.10
128	2.76	5.34	9.72	1.75
256	4.86	9.18	17.37	3.02
512	9.09	16.97	30.93	5.44
1024	17.49	32.56	55.95	10.25
2048	22.20	59.20	112.59	20.21

Myrinet

Size	Min	Avg	Max	Stdev
32	0.58	1.57	5.74	0.90
64	0.78	2.23	9.85	1.44
128	1.18	3.74	17.58	2.54
256	1.98	7.85	55.75	7.07
512	3.58	16.73	226.71	18.83
1024	6.78	36.54	441.90	43.75
2048	13.18	75.89	906.86	93.33

Ethernet

Size	Min	Avg	Max	Stdev
32	1.56	49.55	174.77	19.82
64	1.85	49.35	175.06	19.90
128	2.88	49.30	190.47	20.65
256	4.92	48.18	214.20	26.61
512	9.02	48.32	333.28	52.31
1024	17.21	84.04	586.44	98.29
2048	29.52	151.36	1088.28	188.40

Figure 18. Simulation results from `mesh` algorithm used to generate Figure 17. Size is in bytes, all other fields are in microseconds (μs).

the increased traffic may hurt both the Avici and conventional gigabit ethernet networks as the offered load to the centralized switch(es) is raised, causing the average number of packets queued at the output ports to grow.

Avici

Size	Min	Avg	Max	Stdev
32	1.38	2.66	5.04	0.87
64	1.65	3.45	6.60	1.17
128	2.76	5.51	10.11	1.86
256	4.86	9.60	17.22	3.21
512	9.09	17.53	30.66	5.75
1024	17.49	33.49	55.68	10.72
2048	22.20	59.92	103.14	20.85

Myrinet

Size	Min	Avg	Max	Stdev
32	0.58	1.71	5.89	1.01
64	0.78	2.51	9.85	1.60
128	1.18	4.12	18.19	2.86
256	1.98	8.91	65.84	8.10
512	3.58	20.92	154.19	22.29
1024	6.78	42.92	327.98	47.49
2048	13.18	88.30	591.33	99.29

Ethernet

Size	Min	Avg	Max	Stdev
32	1.56	55.07	174.02	20.94
64	1.85	55.12	174.30	20.99
128	2.88	54.91	175.33	22.19
256	4.92	55.71	232.28	28.53
512	9.02	49.32	390.97	60.36
1024	17.21	84.73	718.95	112.50
2048	29.52	161.71	1331.21	224.29

Figure 20. Simulation results for the `torus` algorithm used to generate Figure 19. Size is in bytes, all other fields are microseconds (μs).

Completion times

The second data analysis we perform takes into account more of the details of the algorithm. Figures 21, 22, and 23 show six plots for a series of message sizes across three pages. We arranged these plots in three rows,

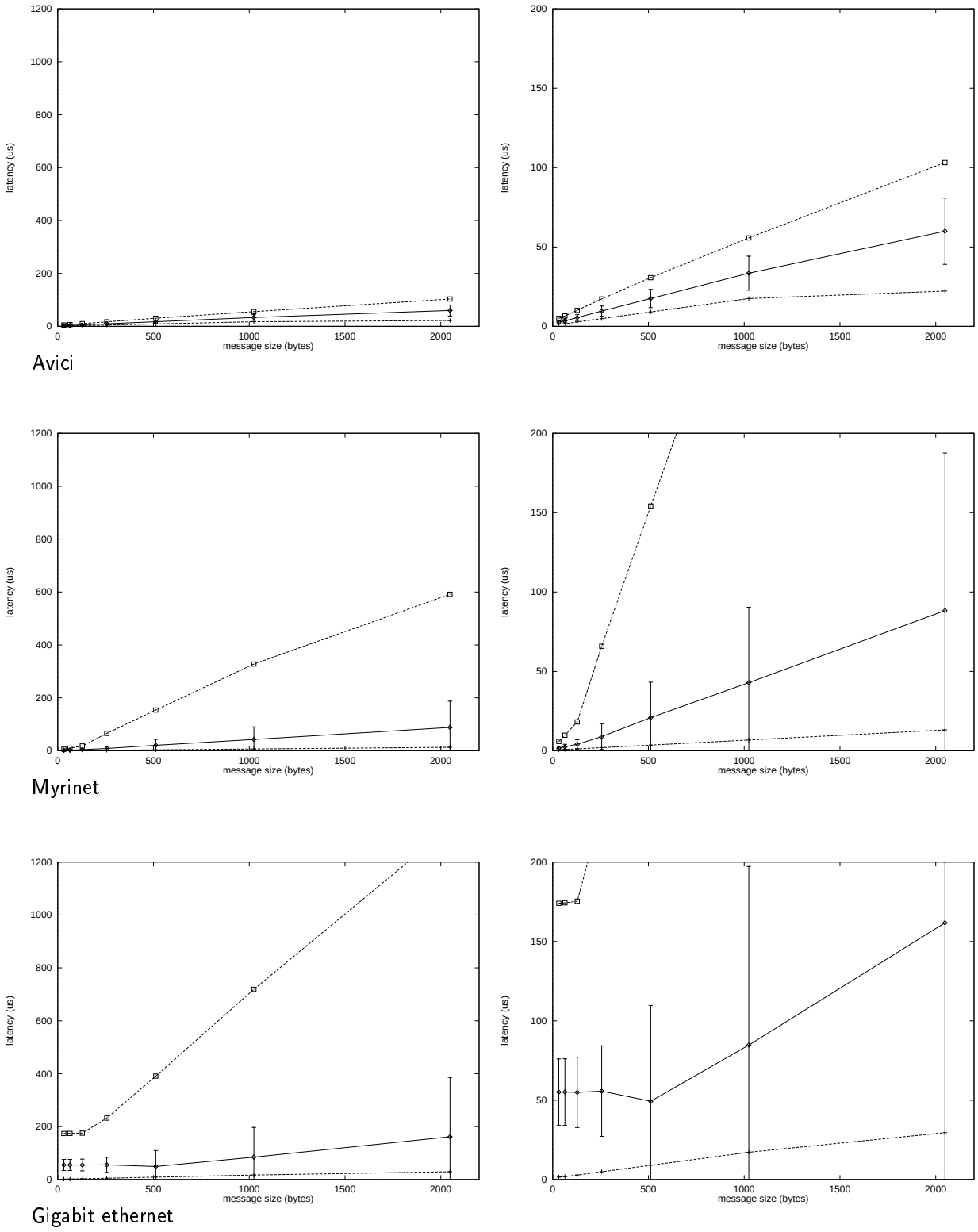

Figure 19. Torus simulation message latencies as a function of message size, with standard deviation around the average. Maximum and minimum values are plotted as well. Large scale in left column, close-up in right, axes are identical to those in Figure 17.

each representing a network technology: Avici, myrinet, and gigabit ethernet. Plots of six message sized are placed in a row, in increasing order from left to right and across three pages.

Each plot shows, for each iteration, and for each processor, the time when that processor completed that iteration. The unlabelled vertical axis is the iteration number of the algorithm, from 1 to 10. The horizontal axis is the global time, in microseconds, and varies from plot to plot as the completion times are quite different with respect to both message size and to network technology. Plots for the `torus` case are in Figures 24, 25, and 26.

Each integral band of y-axis is broken up into 256 points, one for each processor, and a dot is placed in a processor's strip in a given iteration number at the time that processor has sent and received all messages necessary to proceed with the calculation of that timestep, or equivalently, when the processor has received the results of the previous iteration from all his neighbors. For example, in Figure 21 in the top-most plot, there is a dot toward the lower-left of the grouping between 5 and 6 on the y-axis at (68.31, 5.00) representing that processor 0 completed the communications after its fourth iteration at a time 68.31 μs into the execution time of the simulation, and its closest neighbor to the right is (70.35, 5.125) which says processor 32 (since $32/256 = 0.0125$) completed a bit over 2 μs later.

One thing to notice in the plots is that some processors always complete much earlier than the others. For the `mesh` case, these are usually the ones on the corner which have fewer messages to exchange with their neighbors, as there are fewer neighbors. In the `torus` case, this is not true, and the individual bunches of dots tend to be more even, as the corner and edge processors can not advance too far ahead of the rest in the middle.

For the columns where the message size is fairly small, the iteration bunches are well separated from each other as most of the time to completion of each iteration is taken up by computation time, represented in our simulation by a sleep time of 10 μs. An ideal network which used no time to transfer messages would show perfectly vertical lines at each iteration, with the last line (between 9 and 10) at 90 μs. Anything more than this is the effect of waiting for communications.

Scanning across the top row of the figures to look at just the Avici results, it can be seen that the total time to completion is gradually increasing, from 140 μs to 900 μs at 2 kilobyte messages, and that each iteration bunch is becoming separated into individual stripes, with the edge processors finishing earlier than the bulk in the center. For the `torus` case there is no obvious striping.

In the middle row lie the results for the myrinet network where the groups are fuzzier as the effect of the larger maximum communication times shown in Figures 17 and 18. The apparent patterns in the large message size plots show the discrepency in transfer time between nearby nodes and distant nodes in the mesh (or torus) as messages sent farther through the network are subject to more potential points of blocking. Total time to completion for these simulations are the same as for Avici at small message sizes, to about three times longer in the large message case.

The bottom row of the six pages show the results for the conventional gigabit ethernet cascade of switches. The x-axis range on these plots are consistently three to six times larger than those for the Avici plots. Great multi-millisecond stripes can be seen in the large message size plots for the ethernet where whole regions of the two dimensional mesh proceed into later iterations while other regions are still working on the communications associated with earlier iterations. In the `torus` case this horizontal striping is more pronounced but the iterations are forced to be more temporally bunched as the added toroidal communication patterns introduce more dependencies between processors.

This spread in iteration number is allowed to occur since there is no global synchronization step between iterations—each processor is permitted to proceed to the next iteration as long as it has received results from the previous iteration from all its neighbors. Following this thought, it can be seen that a certain processor can get up to two iterations in time away from those processors which are neighbors of its immediate neighbors. This continues up to the boundaries of the mesh, which for our 16×16 case means that the spread can proceed up to eight iterations apart.

It is interesting to notice the rate of degradation of network performance with increasing message size. This is shown in Figures 27 and 28 for the two algorithms. A linear fit of the largest two points for each technology and each algorithm gives:

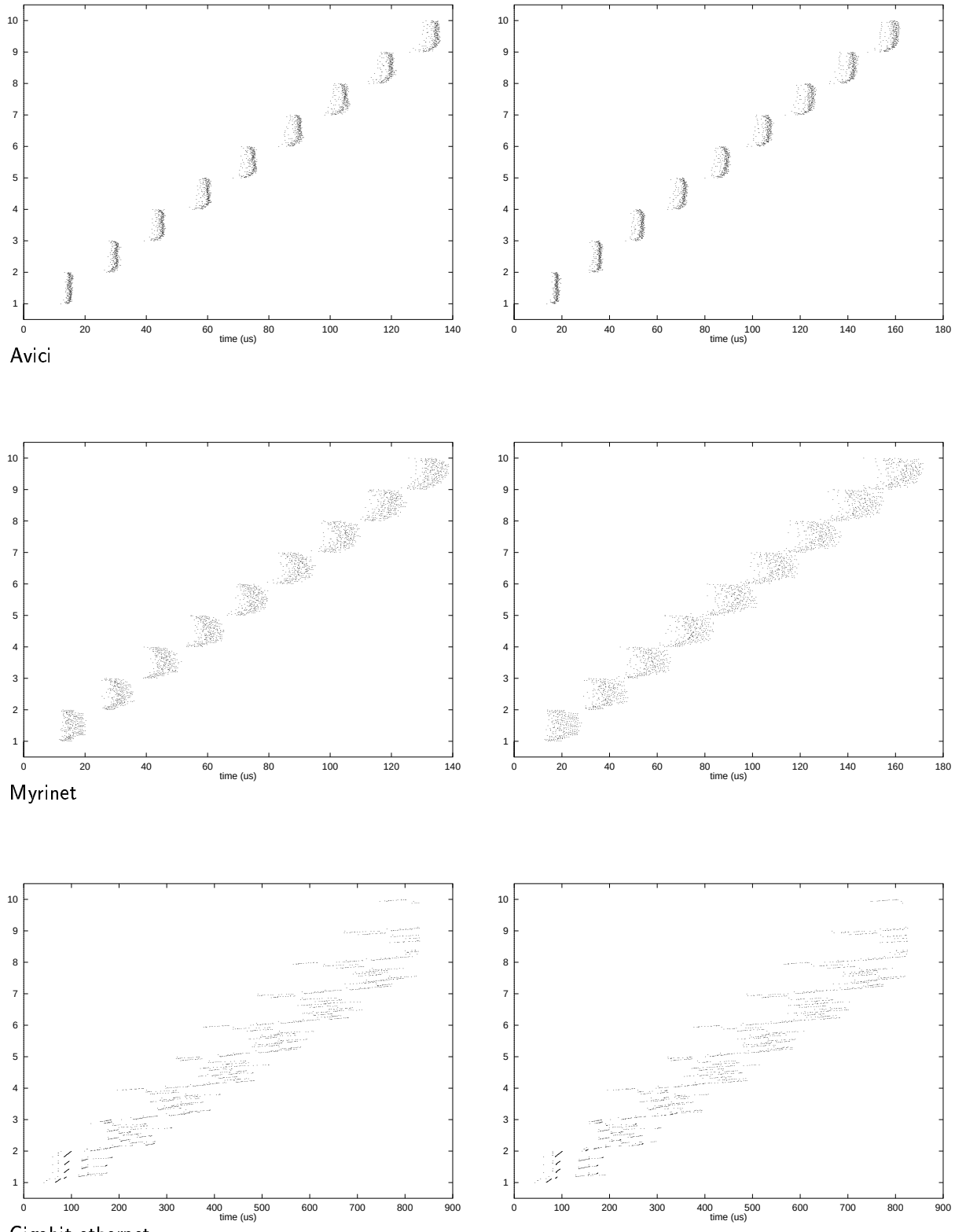

Figure 21. Mesh completion times, 64 (left) and 128 (right) byte messages.

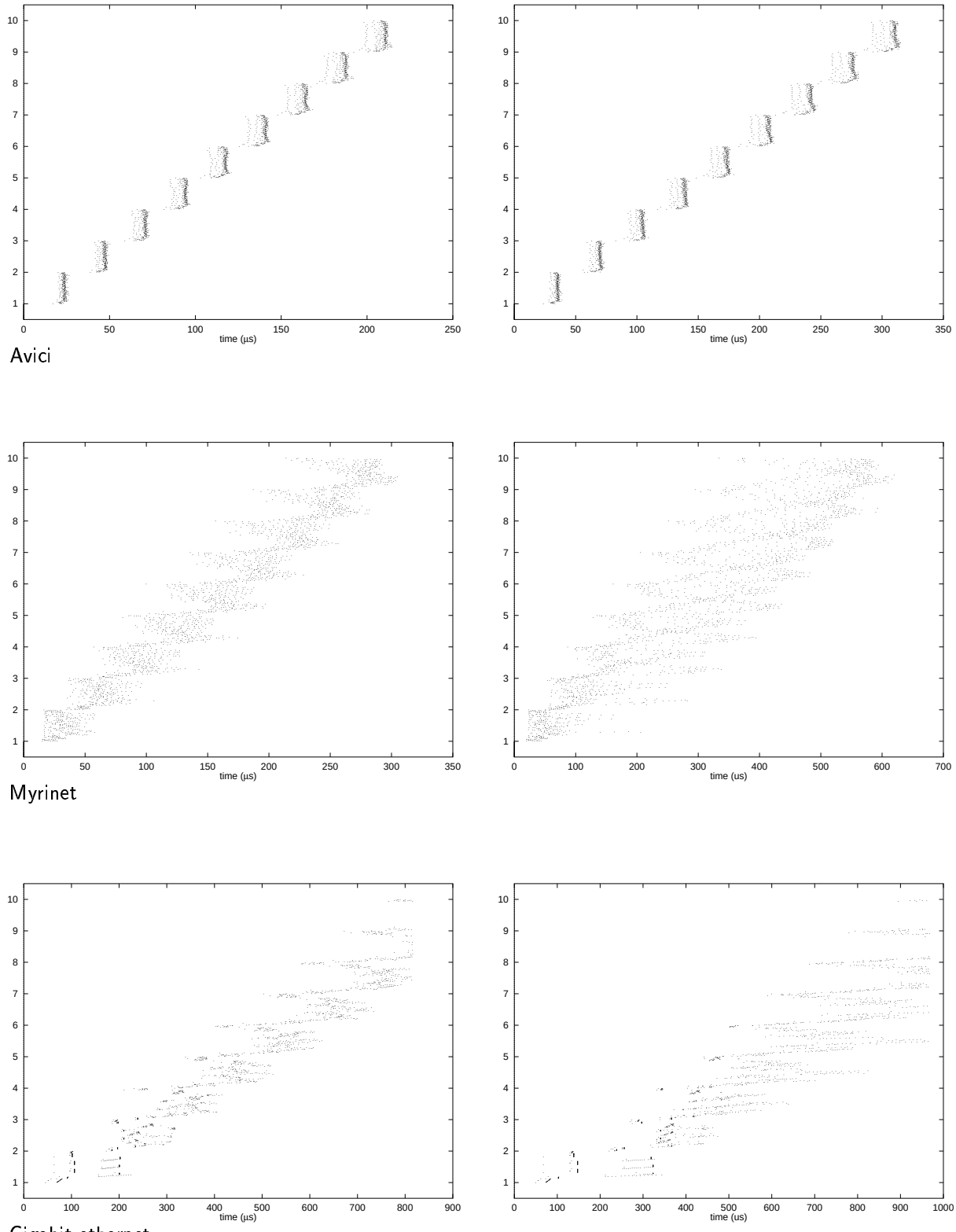

Figure 22. Mesh completion times, 256 (left) and 512 (right) byte messages.

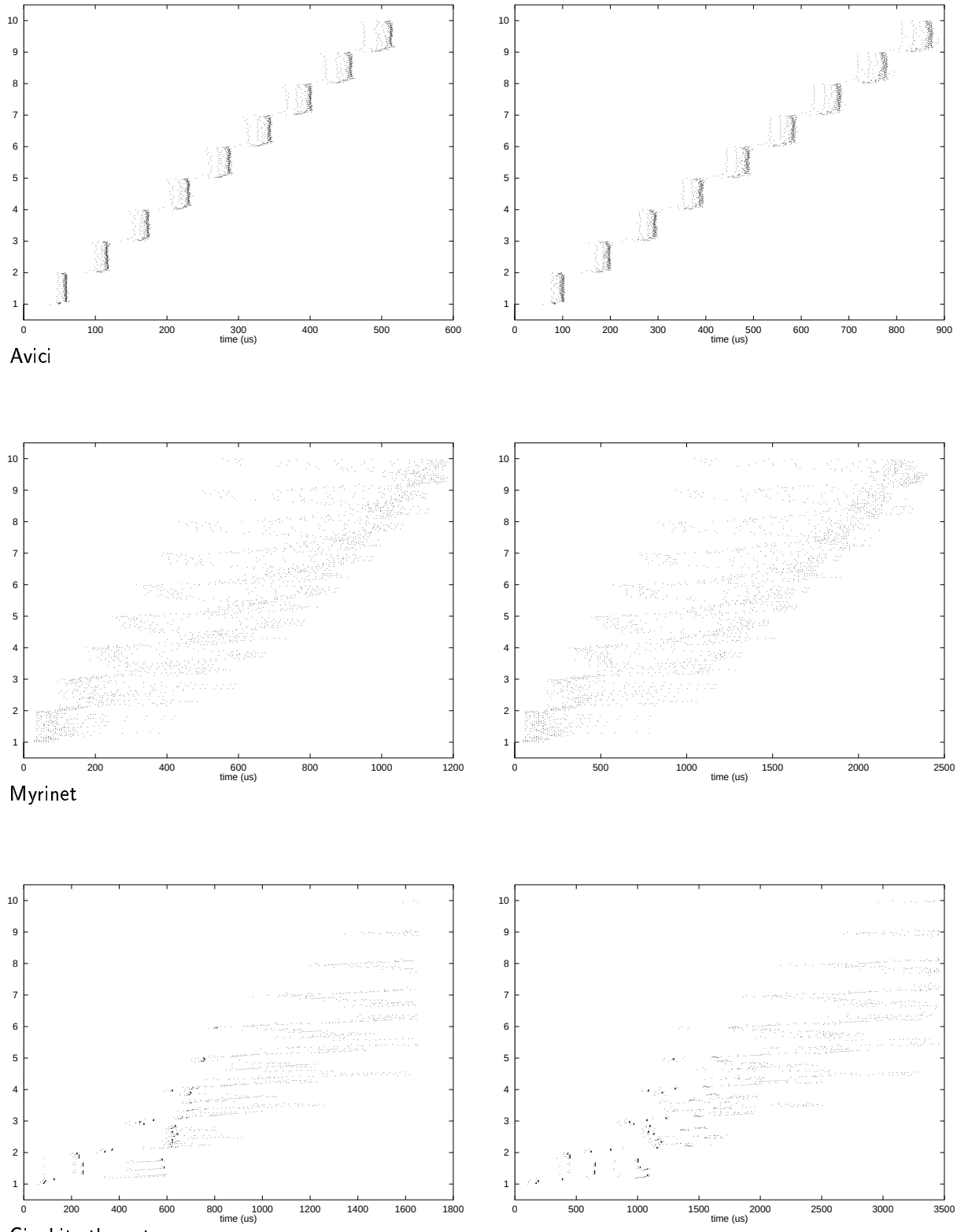

Figure 23. Mesh completion times, 1024 (left) and 2048 (right) byte messages.

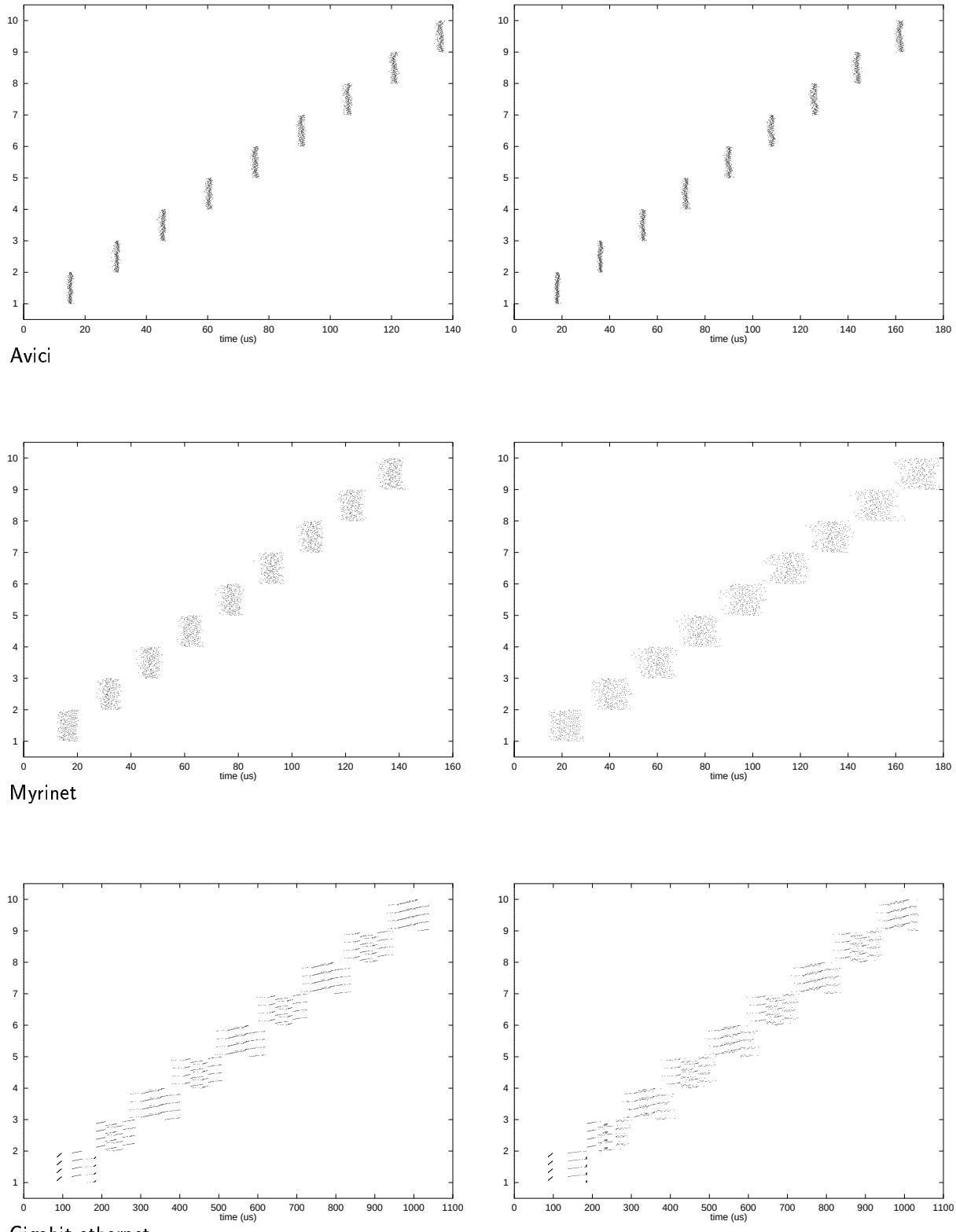

Figure 24. Torus completion times, 64 (left) and 128 (right) byte messages.

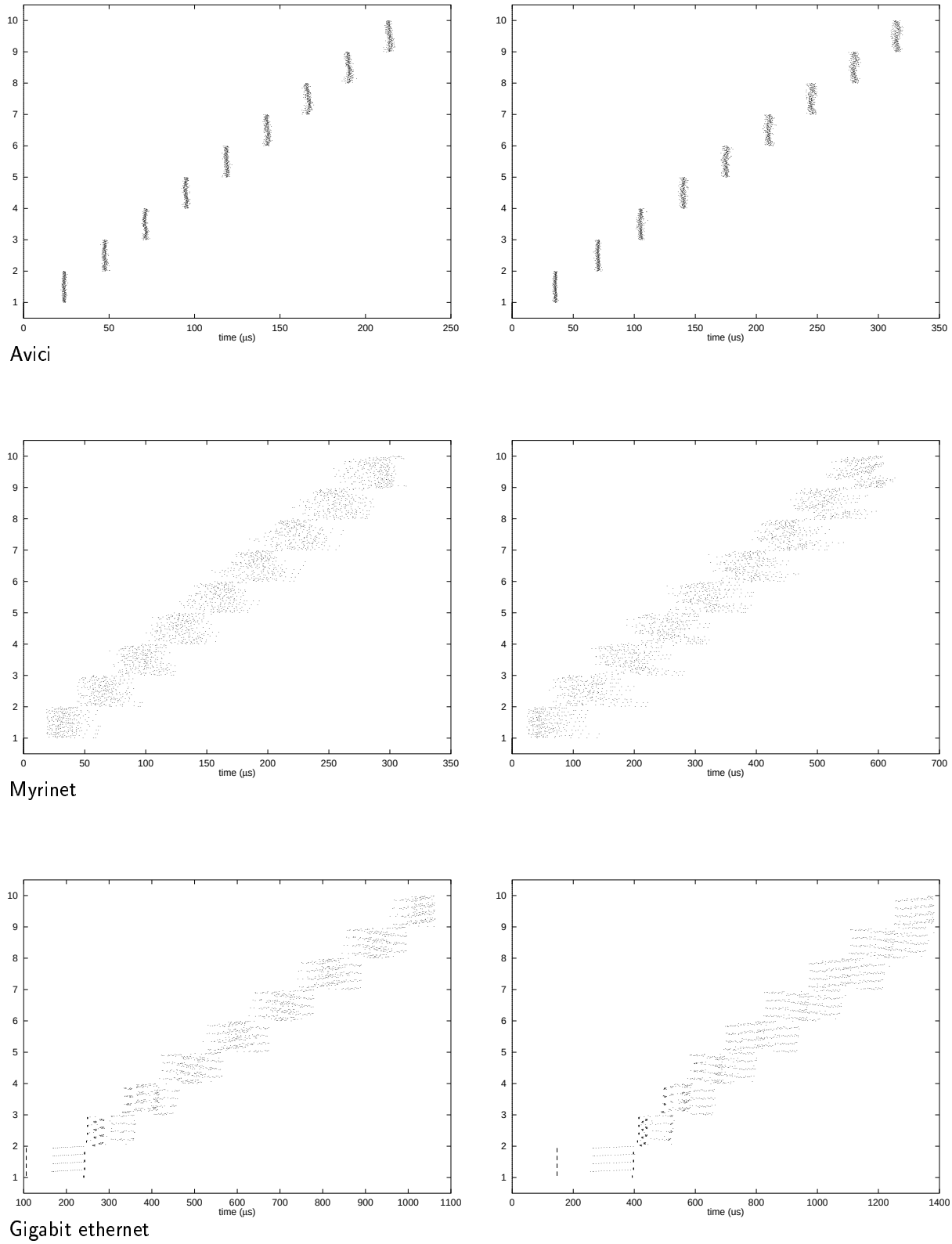

Figure 25. Torus completion times, 256 (left) and 512 (right) byte messages.

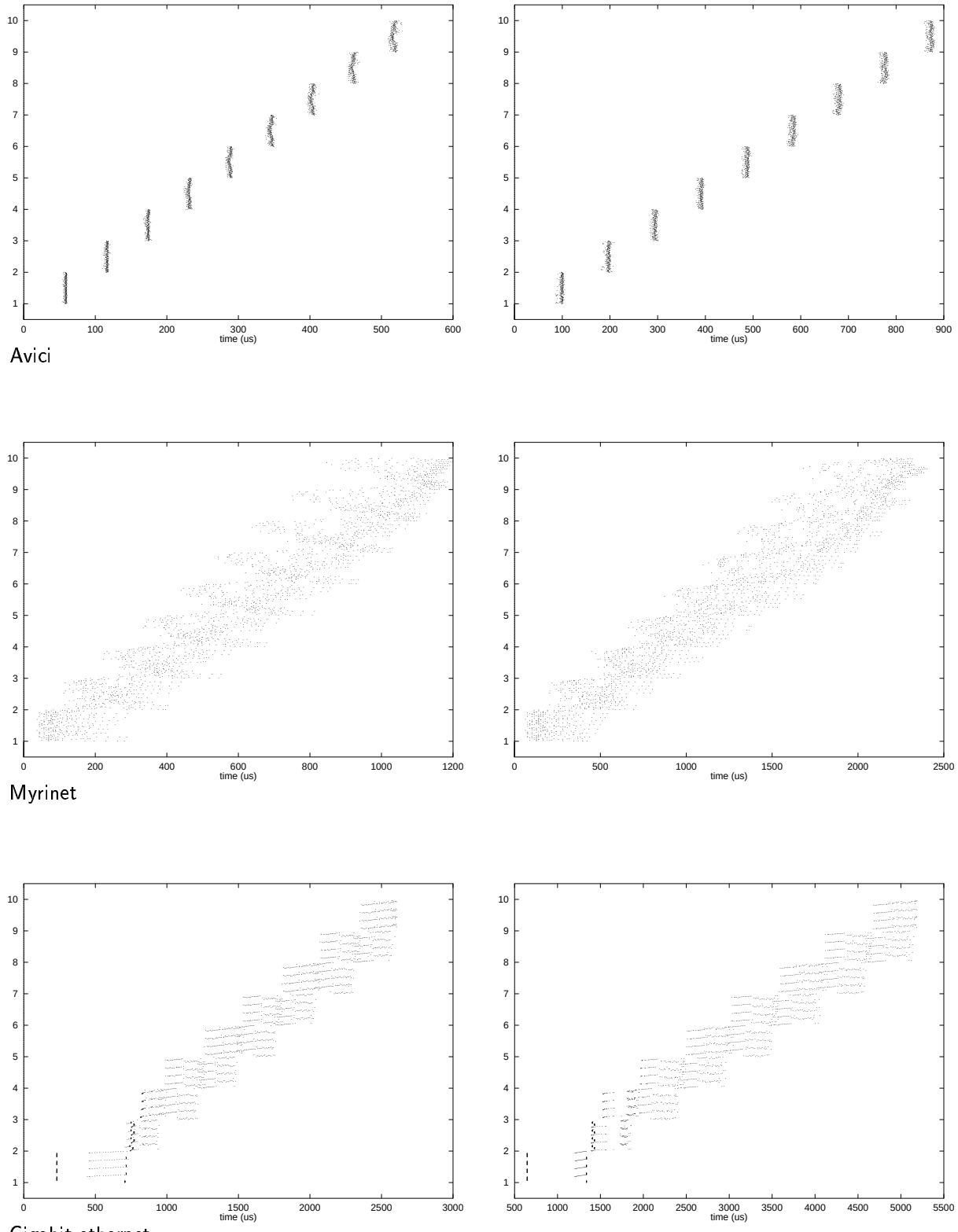

Figure 26. Torus completion times, 1024 (left) and 2048 (right) byte messages.

	mesh	torus
Avici	0.36	0.34
Myrinet	1.19	1.18
Ethernet	1.77	2.52

where the individual entries are in μs/byte and represent the scalability of the given network technology to the two algorithms under increasing message size. Both codes give the same scalability performance on the first two networks, but for conventional gigabit ethernet, the increased offered load seen in the `torus` algorithm renders the network less scalable as message sizes grow.

Summary

The average message latency delivered by the three network technologies is affected both by available bandwidth and the presence of bottlenecks. The conventional cascaded gigabit ethernet without trunking is affected severely by both these factors. Fabric blocking is seen to be bad for parallel algorithms in that it increases the maximum latency seen by any particular message, and since all messages must eventually reach their destination before the code can complete, that maximum latency value is crucual to the wall-clock performance of a code. The Avici switch is seen to have the smallest amount of fabric blocking, while the myrinet fabric offers potential blocking points at every switch along the path of a message.

The algorithm we tested was chosen due to its ubiquity in parallel scientific computing. It emphasizes the nature of locality in many algorithms in production use today, but points out in the results above that not all communications will physically be local even though in the virtual topology they may appear to be nearest neighbor. This mapping of algorithmic topology to physical topology is crucial for application performance. We did not model an algorithm which involved a global synchronization, which are also common especially for those that do disk input and output. The effects of this communication pattern can be discerned by looking at the `fan_in` and `fan_out` results of Section 5.2.

6 Conclusions

We have presented the results of analysis of three different major network architectures for parallel commodity computing. It is important to choose the network correctly as it can have a large impact on all but the most embarassingly parallel applications, and may be the source of up to half of the cost of the entire machine. Important factors to consider are raw performance figures such as bandwidth and latency, as well as more complex parameters such as jitter, routing, multicast support, and distribution of blocking in the fabric.

Since our network design goal is to facilitate the performance of real applications, we evaluated the performance of the three network technologies when applied to specific application cores important to our users. In this context we analyzed timing results gathered from the networks and drew conclusions from our knowledge of the network about its effect on performance of the application.

Our simulation results show that myrinet behaves well in the absence of congestion. Under heavy load, its latency suffers due to blocking in wormhole routing. Also myrinet is limited from scaling too far due to the short cable length problem. Future development by Myricom may alleviate that constraint, although the cost to latency or budgets is unknown. The simplicity in the myrinet switch results in low per-connection cost; however, the non-commodity nature of the host network interface cards keep that side of the connection expensive.

Conventional gigabit ethernet switches can not scale to support more than 64 gigabit ethernet ports, which leads to the introduction of a topology which involves cascading multiple stages of small switches. The presence of multiple hops in a path between hosts, and the store-and-forward nature of legacy ethernet leads to unacceptable message delays. Bandwidth bottlenecks at the topmost switch in the cascade are also a problem.

The Avici terabit switch router has an internal fabric which is quite similar to myrinet, in that it is a very high-bandwidth three-dimensional torus using source routing and simple non-buffering switches. The line cards present a standard gigabit ethernet connections to hosts, though, in keeping with the current commodity favorite. Our simulations show that the Avici switch outperformed myrinet on large messages

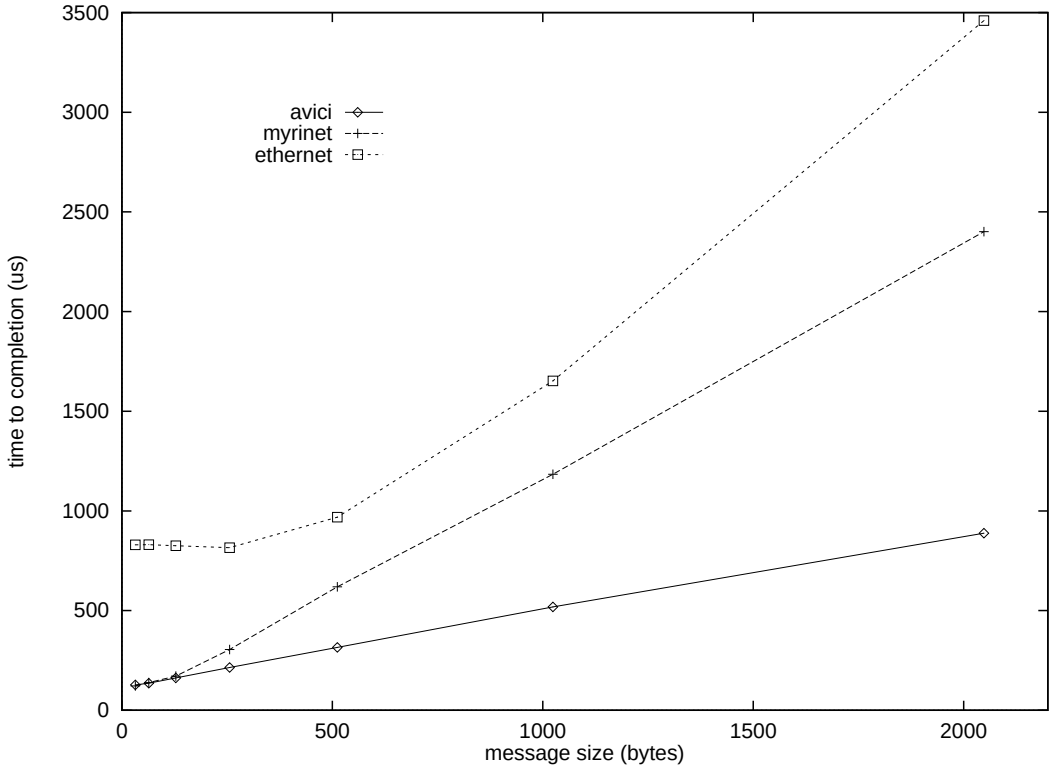

Figure 27. Total time for completion, `mesh` topology.

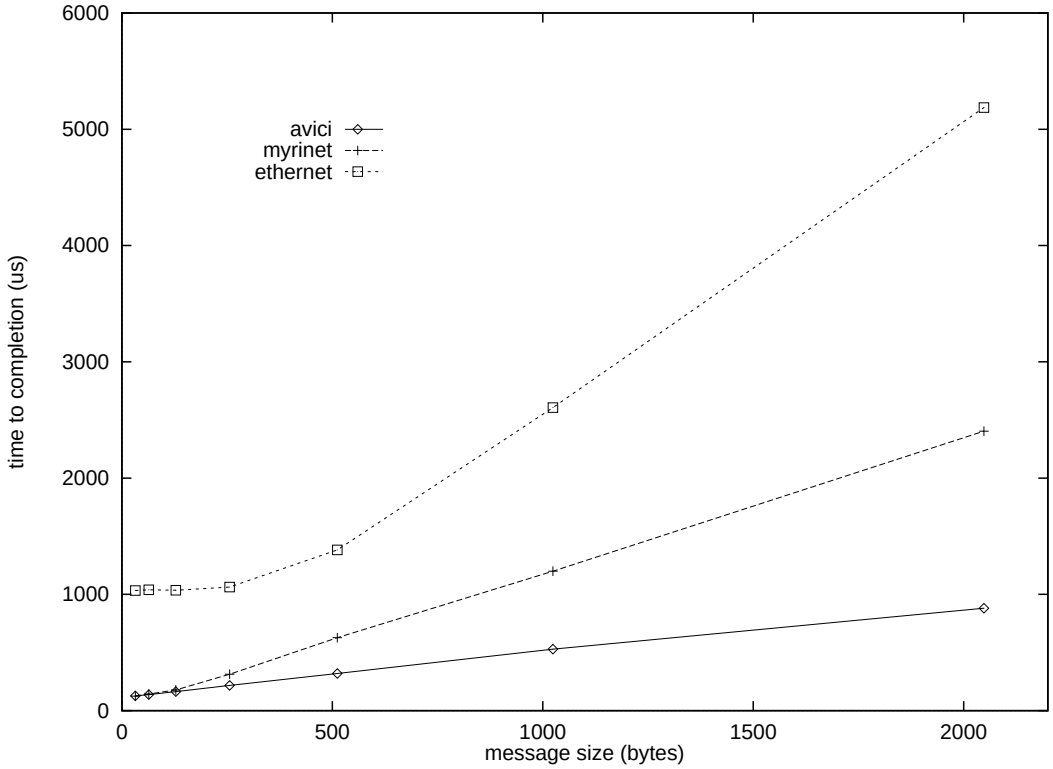

Figure 28. Total time for completion, `torus` topology.

(above 512 bytes), and was comparable in the small-message regime. From a cost standpoint, Avici is only slightly cheaper than myrinet for a comparable topology, and is expected to reduce in cost with further penetration of gigabit ethernet into the market.

7 Future Work

Our preliminary results demonstrate that the Avici terabit switch router met our performance criteria. Moreover, because the tremendous speedup in its fabric, it is way over subscribed to support 16-port gigabit-ethernet line cards. Avici has plans to manufacture a 63-port gigabit ethernet line card, which would not only increase the cluster's processor count but also drive the per-port cost down from $3000 to $600.

We have obtained fund from Sandia's Research Foundation to prototype a high performance cluster using the Avici router to interconnect existing commodity components such as the Alpha processors, their gigabit ethernet adapters, and the Linux operating system. We intend to extend the low latency environment to the cluster's compute nodes. We describe the proposed research study in the following paragraphs.

Using its stringent flow control protocol, the Avici router achieves reliable end-to-end deliveries within its fabric. We propose to extend the Avici flow control to reach end-hosts, thereby eliminating their needs for TCP/IP and allowing users a direct access to network IO. This will involve collaborations with researchers at the Avici Systems in order to design the flow control extension that interoperates. We propose to conduct simulation studies to evaluate and refine our design prior to the actual implementation.

With efficient end-to-end flow control in place, we can bypass the TCP/IP kernel and replace its multiplexing and de-multiplexing functionality inside the adapter's device driver. We plan to evaluate and adapt either Sandia's portals or VIA to accomplish this.

8 References

[1] Dally, W. "Scaleable Switching Fabrics for Internet Routers." Computer Systems Laboratory, Stanford University and Avici Systems. July 1999.

[2] Duato, J., Yalmanchili, S., and Ni, L. "Interconnection Networks: an Engineering Approach." *IEEE Computer Society Press,* 1997, pp. 11–16.

[3] Held, G. *Ethernet Networks.* John Wiley & Sons, Inc., 1996, pp. 78–95.

[4] Jacobsen, O. J., ed. "From the editor." *The Internet Protocol Journal,* **2**, 3, 1999, pp. 1ff.

[5] King, A. "Terasim: the Simulator for Avici TSR." Avici Systems, Inc., 1997.

[6] Kuo, C. C. "The Avalanche Myrinet Simulation Package: User Manual for V2.0." Department of Computer Science. University of Utah. May 1997.

[7] Seifert, R. *Gigabit Ethernet: Technology and Applications for High-Speed LANs.* Addison-Wesley, 1998, pp. 141–280.

[8] Stevens, W. R. *TCP/IP Illustrated.* Addison-Wesley, 1994–1996.

[9] `http://www.myri.com/myrinet/overview/index.html`.

[10] `http://www.hnf.org`.

[11] `http://www.mil3.com/products/modeler/home.html`.