



Performance of Voltaire InfiniBand in IBM 64-Bit Commodity HPC Clusters

*Dr. Douglas M. Pase
IBM xSeries Performance Development and Analysis
3039 Cornwallis Rd.
Research Triangle Park, NC 27709-2195
pase@us.ibm.com*

Abstract

In this paper we examine the performance of Voltaire InfiniBand using 100 MHz PCI-X, 133 MHz PCI-X and PCI-Express slots. To do this we use the Pallas MPI Benchmarks and the NAS Parallel Benchmarks in two eight-node Linux clusters of IBM® eServer™ 326 (e326) and IBM eServer xSeries® 336 (x336) servers. In our experimentation we find that InfiniBand exhibits high throughput (1,735 MB/s) and low latency (4 microseconds) when used with PCI-Express. With 133 MHz PCI-X it also shows low latency (5 microseconds) and high bandwidth (788 MB/s), considering the limitations of its slot. InfiniBand shows consistent performance over a range of message sizes. We also find that the e326 cluster with InfiniBand performs especially well on the NAS Parallel Benchmarks, with superior performance on six of the eight benchmarks. This result was surprising because we thought InfiniBand's superior performance on the x336 cluster would have greater impact than it did.

1. Introduction

High Performance Computing (HPC) has gravitated in recent years to clusters of Commodity Off-The-Shelf (COTS) servers and components. Large clusters built from inexpensive commodity processors and memory replace the expensive, monolithic compute servers of the past. The glue for such clusters, the component that allows the many commodity servers to productively work together, is the network, or “interconnect fabric,” as it is sometimes called. To solve large HPC problems, the application must divide the work and data into smaller components and distribute them appropriately to each of the individual compute nodes. Each server then works on its own portion of the problem for a period of time. But each node must also communicate with other nodes in the cluster to properly coordinate the work and to share intermediate results with each other. How this communication takes place is dependent on the type of problem being solved and the algorithm is chosen to solve it, but the point is that the communication must take place, and must take place quickly. A single slow node can significantly degrade the performance of the rest of the cluster, and slow communication has the same effect as a slow processor.

Fast commodity networks are seen as an important component of any high performance cluster. There has been a lot of development in recent years toward very high performance commodity interconnect fabrics, from Gigabit Ethernet on the low end, to Myrinet, to Quadrics on the high end. InfiniBand is a recent addition to that list, and it is important to understand how well it does or does not perform. In considering performance, we only use end-to-end performance, that is, performance that would be experienced by an application sitting on top of the network, rather than a lower-level measure such as link bandwidth or latency. Lower-level measures are valuable in their own right, but they can also be misleading, so we do not use them here. The measures we *do* use are the Pallas MPI Benchmarks, to measure the performance of individual MPI subroutines over InfiniBand, and the NAS Parallel Benchmarks, a suite of kernels and pseudo-applications designed to have performance characteristics similar to common Computational Fluid Dynamics (CFD) codes.

The remainder of this paper is organized as follows. Section 2 describes InfiniBand design and implementation in general and Voltaire InfiniBand in particular. Section 3 on page 7 describes the benchmarks we used, what they measure and how they work. Section 4 on page 11 describes MPI in some detail, including factors that affect MPI performance. Server performance is described in Section 5 on page 17, to lay a proper foundation for understanding the results of the benchmark experiments. Section 6 on page 19 presents the results and analysis of our experiments with the Pallas MPI Benchmarks, and Section 7 on page 28 presents the results and analysis of our experiments with the NAS Parallel Benchmarks. Section 8 on page 35 gives a summary of our conclusions. Acknowledgements and References are in Section 9 and Section 10.

2. Voltaire InfiniBand

2.1 General Design

InfiniBand is an open standard for interprocessor and storage communication. It builds on the lessons learned from Ethernet, while incorporating features useful to enterprise-class computing and storage networks. It provides a flexible, manageable, high-speed communication infrastructure [1][2].

InfiniBand connections support three types of ports, namely a backplane port, a copper cable port, and a fiber optic port. Only the copper cable port was tested. Ports support three speeds as well—1x, 4x and 12x, named for the number of full-duplex data paths, or lanes, in the port. (Full-duplex paths allow data to be transmitted in both directions simultaneously.) Each lane is clocked at 2.5 GHz, and uses 8-bit/10-bit encoding. This gives a bandwidth of 250 MB/s per direction, per lane. This means a 4x device is capable of 1,000 MB/s, in each direction, and a 12x device is capable of 3,000 MB/s.

Although the devices and switches are capable of 1 GB/s in each direction, the adapter slots into which they are plugged are not always capable of such speeds. The e326 supports two 100 MHz or one 133 MHz PCI-X adapter slots. Each slot is 64-bits wide and uses a half-duplex connection. (Half-duplex allows data to be transmitted in only one direction at a time.) As such, a 100 MHz slot is capable of, at most, 800 MB/s total, alternating directions. A 133 MHz PCI-X slot is capable of 1,067 MB/s total.

A new type of adapter slot, called PCI-Express, or PCI-E, is available in the x336 systems. PCI-Express is supported in 1x, 2x, 4x, 8x and 16x widths. Each PCI-E lane operates at 2.5 GHz in much the same manner as an InfiniBand lane, that is, full-duplex and 8-bit/10-bit encoded. (The x336 supports a single 8x PCI-E slot or a 133 MHz PCI-X slot, or two 100 MHz PCI-X slots.) A 4x PCI-E adapter slot therefore seems ideally suited for a single 4x InfiniBand port.

Our tests exercised only 4x copper cable devices. The devices themselves are dual-port, where each port is capable of 1,000 MB/s each direction, but their actual performance may be less depending on whether they are placed in 100 MHz PCI-X, 133 MHz PCI-X, or 8x PCI-E adapter slots. We used both 100 MHz and 133 MHz PCI-X on the e326 systems, and 133 MHz PCI-X and 8x PCI-E on the x336 systems.

InfiniBand packet headers at the lowest level include Local Routing Header (8 bytes) and Base Transport Header (12 bytes). Trailers include ICRC (4 bytes) and VCRC (2 bytes) integrity checks. InfiniBand allows the Maximum Transfer Unit (MTU) to be specified as 512, 1,024, 2,048 or 4,096 bytes. Voltaire uses an MTU of 1,500 bytes, which implies the data payload cannot be more than about 98% of the total bandwidth. On a 100 MHz PCI-X adapter slot this gives a peak bandwidth of around 785 MB/s. Note that this does not include losses from higher-level protocols, such as MPI, or from the transmission of data over the PCI-X bus. On a 133 MHz slot the peak is around 1,047 MB/s, and for PCI-E the peak is 1,965 MB/s. This doesn't mean we can achieve such performance, only that we cannot exceed it. It should be considered an upper bound rather than an expectation.

The switch design is another point to consider. Even the fastest switch adds latency and reduces bandwidth. Voltaire offers switches with 24, 96 and 288 ports. In our test bed we have a model ISR 9024 with 24 ports. The switch uses a multistage design that increases the number of stages by one each time the number of ports doubles. For example, a 24-port switch uses three stages, a 96-port switch uses five stages (see Figure 1), and a 288-port switch uses seven stages. When a larger cluster is needed, switches can be cascaded together to form a larger network, as illustrated in Figure 2. Unloaded latency through the switching fabric for the current generation of switch is 144 nanoseconds per stage. So, for example, worst case latency through a 288-port switch is just over one microsecond. Latency through an ISR 9024 switch is just under 1/2 microsecond.

2.2 Network Operation

A goal of this paper is to make clear not only *how* InfiniBand performs, but also *why* it performs as it does. To this end we start with a description of what occurs when an application sends and receives a message. In this way we can identify where potential bottlenecks may occur within a system, and the circumstances under which they can affect performance. To aid in understanding we contrast the design of MPI over InfiniBand with that of MPI built on Ethernet.

When an application sends a message, it prepares the message buffer and places a call to a library. For our purposes the library will be an implementation of MPI, though the same principles apply whether the message is passed via some other interface, such as PVM or BSD sockets. If the MPI library is implemented on top of a standard Ethernet connection that uses TCP/IP, it will pass the message to the TCP/IP stack, first making the transition from user mode into kernel mode. The TCP/IP stack examines the message and determines the number of IP packets needed to transmit

the message. It then allocates enough space for the message packets, copies the data into the packet payload areas, fills in the TCP and IP headers, and hands the packets to the Ethernet driver. The driver copies the packets to the Ethernet adapter, either by direct copy or by DMA, and the packets are transmitted over the network. If other packets have arrived in the interim, the driver transmits those packets, too, until there are no remaining packets to be transmitted. When the transmission completes, the message buffers are released and the sending process is notified that the operation has completed. This is illustrated in Figure 3.

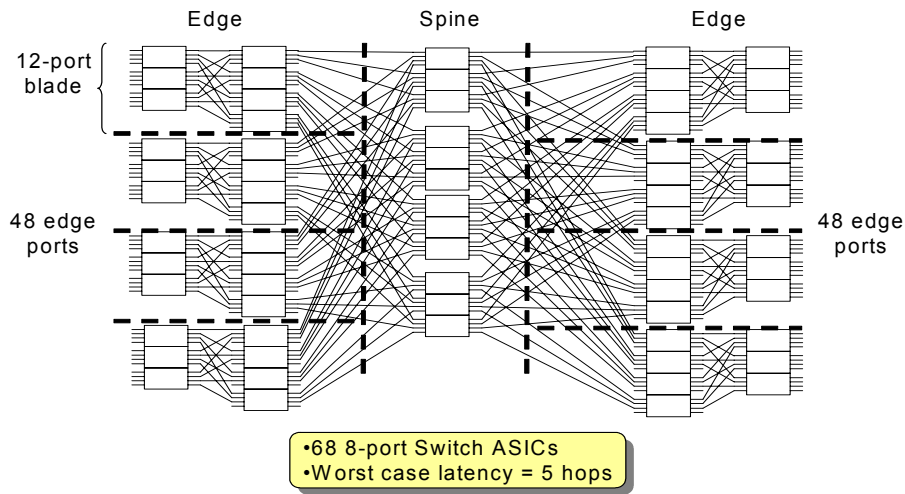


Figure 1. Block Diagram of the Voltaire ISR 9600 (96-Port) Switch

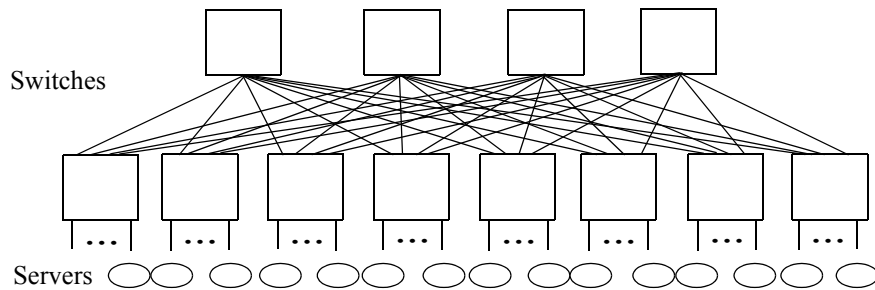


Figure 2. Combining Switches to Form a Larger Network

Once a packet is sent from an adapter, it is passed on to a switch. Ethernet switches use a mechanism called *store and forward*. This means the entire packet is received by the switch, processed, and then forwarded out the appropriate port. The outbound port may be connected to the final destination, or it may be connected to another switch.

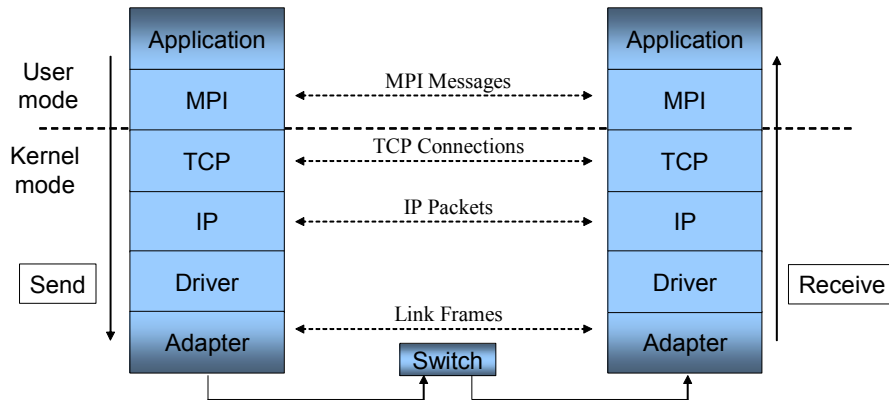


Figure 3. Ethernet Control and Data Flow

Receive operations at the final destination begin when an incoming packet arrives at the adapter. Packet buffers may be pre-allocated or they may be allocated on demand, that is, when the packet arrives, or a combination of both. If they are allocated on demand, the processor must be interrupted to allocate a packet buffer. Once the buffer is allocated, the incoming data is copied from the adapter to the buffer, again, by direct copy or by DMA. The packet is then forwarded to the IP layer, and afterwards to the TCP layer, to strip off the packet headers and reassemble the message. If some portion of the packet is corrupted, if packets arrive out of order, or if a packet is missing, the TCP layer requests the sender to retransmit the offending packet. The correctly assembled message is then held in a queue until the receiving process requests the message. When the user process issues the request, the message is then copied from the kernel buffer to a user buffer.

Notice that receiving a message may require several interrupts for each packet that arrives. Processing an interrupt is expensive, at the least because it is expensive to save the current state of whatever is running and figure out to whom the interrupt must go. A good driver will reduce the number of interrupts by accepting the first interrupt, then disabling future interrupts from that device. When the driver completes its processing of a packet, it checks whether there are any more incoming packets. If so, it processes the next packet. The driver continues to process incoming packets until there are no more to process, after which it re-enables interrupts and exits.

The above description presents a fairly standard design for message passing within a network. As can be seen from the description, a lot of work must be done to send a message and to retrieve it using this method. Multiple transitions must be made, from user mode to kernel mode and back again. Each incoming packet can require a separate interrupt to be processed. The data must be copied multiple times to and from user buffers, kernel buffers, and the network device itself.

MPI over InfiniBand attempts to use a more efficient model. Instead of frequent transitions between user and kernel modes, InfiniBand attempts to minimize the transitions by keeping

critical operations in user mode. It also attempts to reduce the number of times a message must be copied between buffers. Figure 4 illustrates this model.

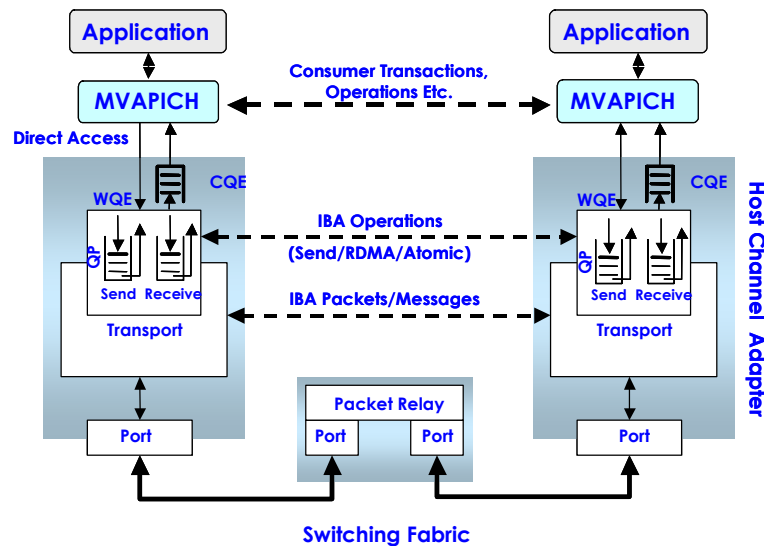


Figure 4. InfiniBand Control and Data Flow

To send a message in this model the MPI layer, called MVAPICH, creates a Work Queue Element (WQE) that contains a message description, start addresses, element lengths, and other information. Certain addresses within the InfiniBand Host Channel Adapter (HCA) are mapped into user-addressable locations, so MVAPICH is able to notify the HCA that new work is available without ever making a transition into the kernel. The HCA reads the WQE and transmits the data directly from memory, blocking it into packets as necessary. If a transmission failure occurs, the HCA retransmits any offending frames. Blocking operations must use a spin lock to delay the user process until the HCA has completed the transmission. Non-blocking operations return control immediately to the application.

Some messages, such as scatter and gather operations, are not contiguous, but MVAPICH avoids an extra copy by leaving pointer references to the data in the WQE. When the HCA transmits the message, it recognizes the pointer and copies the data to the HCA at that time. This eliminates a copy operation that would otherwise have to be done.

The switch does not buffer any messages in order to keep the performance high. As each packet enters the switch, its destination is read and mapped to an output port. Each byte is passed immediately to the output port as it enters the switch. If a problem with the packet is detected, a field in the packet trailer is marked, because by the time a problem is detected, some or all of the packet has already left the switch.

The receiving HCA accepts the incoming frames, validates them and reassembles them into the original message. When the message arrives before the receive operation has been issued, the

message is moved into a pre-allocated buffer, and later copied into the application buffers. If the call to MPI receive (or other suitable routine) has already taken place, the data is copied directly into the application buffers, bypassing the secondary buffers. Finally, the HCA registers a completion notification entry, called a CQE, that is accepted by MVAPICH to indicate that the message transmission is complete.

All data is kept in user memory, whether managed directly by the application or by the MVAPICH library. The main latency reduction on the receiver side stems from the fact the MPI receive queue is in hardware, and the only thing MPI needs to do in order to read a packet is to look at its own memory. To see if another message arrived, it looks into the next memory structure. No interrupts are required to force execution into kernel mode, and no kernel calls are needed for normal operation. This provides a faster mechanism than what is available through TCP/IP and Ethernet.

Ethernet presents another penalty on the switching. Because Ethernet is store and forward, most Ethernet switches have around 10-microsecond latency per hop, whereas InfiniBand switch latencies are closer to 0.14-microsecond latency per hop. A large 5-hop configuration in Ethernet can add 30 to 50 microseconds just for the switching, while a similar InfiniBand configuration would be less than 1 microsecond. Ethernet performance is also more strongly affected by congestion.

3. Benchmarks

Four benchmark suites are used in these tests: Linpack HPL [3][4], STREAM [5], Pallas MPI Benchmarks [6] and NAS Parallel Benchmarks [7]. The first two, Linpack and STREAM, are single-node benchmarks. Linpack measures processor floating-point performance. STREAM measures memory throughput. These two benchmarks do not engage or make use of the interconnect in any way. But they *do* help us understand the performance of other benchmarks, particularly the NAS Parallel Benchmarks, and for this reason, we include them here. Pallas measures the performance of each of the core MPI operations, giving latencies, and where appropriate, throughput measurements, for a wide range of message sizes. In some sense the NAS Parallel Benchmarks are the most interesting of the benchmarks we have chosen, because they come closest to measuring performance as it is seen by the users. The first three benchmarks measure a specific subsystem, whereas the last measures the performance of the entire system.

3.1 Linpack

Linpack HPL (High Performance Linpack) is a benchmark that measures the performance of a node or cluster solving a system of linear equations. The system of equations is represented as a matrix that has been divided into smaller pieces, called tiles, and distributed across the available processors. Performance is reported as the number of floating-point operations per second achieved by the system. The user is able to vary the size of the problem and a number of parameters that describe how the problem is to be solved. Some of the parameters describe how the problem is decomposed into tiles, and how those tiles are distributed.

Linpack itself is a collection of subroutines that analyze and solve linear equations and linear least-squares problems [8][9]. Originally designed for supercomputers in the 1970s and early 1980s, Linpack solves linear systems whose matrices are general, banded, symmetric indefinite,

symmetric positive definite, triangular, and tridiagonal square. Linpack is built upon the Basic Linear Algebra Subroutine package, or BLAS. The Linpack benchmark uses the Linpack library to solve a general dense system of linear equations using LU factorization [10]. LU factorization is similar to a technique commonly taught in college Linear Algebra courses, called *Gaussian elimination*, but significantly improved to increase numerical stability and performance. The Linpack library has largely been replaced by Lapack, which extends and improves upon the routines [11]. In Lapack, the routines have been carefully rewritten and tuned to take advantage of processor cache, and relatively few references actually go to memory. For our tests, we use the double-precision High Performance Linpack (HPL) benchmark over a wide range of problem sizes. Our benchmark implementation uses the high-performance BLAS created by Kazushige Goto [12].

Linpack HPL is primarily a processor-core-intensive benchmark. It heavily exercises the floating-point unit of each processor core. The faster the processor core and the more cores available, the better the benchmark performance. Data is moved from memory into cache and used heavily from cache, so larger caches do help. With current cache sizes of 1MB or more, most of the current working data set already fits into cache, so memory performance is not seen as a major factor in determining Linpack performance.

A factor that *does* affect benchmark performance is how the tiles are defined and distributed. Linpack alternates between computation and communication when solving a system of equations. The work to be done in the computation phases depends on the number and size of the tiles to be solved. How the system of equations is tiled affects how the workload is balanced across the system, and how efficiently each tile can be processed. These factors can be significant. Tile definition and distribution also affect how much time is spent in communication. Time spent in communication depends on the size of the edges of the tiles and the speed of communication. As problem sizes increase, the amount of computation grows faster than the cost of communication, and the impact of communication performance diminishes accordingly.

For these tests the computation was distributed across processors within a single node and *not* across nodes in cluster. Communication takes place within shared memory, so communication speed is not a factor. However, how the tiles are created and used still has a major impact on Linpack performance that is not easy to predict. For that reason many tests were run, but only the best outcome for each problem size is reported.

3.2 STREAM

The STREAM benchmark was created to measure memory throughput. STREAM is a simple benchmark of four loops, as shown in Figure 5. The first loop, COPY, copies data from one vector into another; the second, SCALE, multiplies a vector by a scalar value; the third loop, ADD, adds two vectors; and the fourth loop, TRIAD, combines a scale with an addition operation. This last loop is very similar to a DAXPY operation (Double-precision A times X Plus Y). The arrays used in these four loops are required to be much larger than the largest processor cache, so operands are always retrieved from memory. All operations use 64-bit operands and memory accesses are stride-one (that is, $a[i]$, $a[i+1]$, $a[i+2]$, ...).

```

for (j=0; j<N; j++)      // COPY
    c[j] = a[j];
for (j=0; j<N; j++)      // SCALE
    c[j] = scalar * a[j];
for (j=0; j<N; j++)      // ADD
    c[j] = a[j] + b[j];
for (j=0; j<N; j++)      // TRIAD
    c[j] = a[j] + scalar * b[j];

```

Figure 5. STREAM Loops

STREAM and Linpack are both kernel benchmarks; that is, they focus on measuring the performance of a small, relatively simple mathematical kernel. In so doing they give an accurate measure of the performance of a single subsystem of the computer, though not the system as a whole.

3.3 Pallas MPI Benchmark Suite

The Pallas MPI Benchmark suite measures the raw performance of routines within the MPI library with minimal interpretation. Each test measures latency across a wide range of message sizes. Message sizes are integral powers of two, ranging from zero bytes up to about 4MB. Effective bandwidth (throughput) values are also provided where such measurements are well defined, but that is generally restricted to point-to-point and parallel operations. Collective operations do not report bandwidth measurements.

This suite is important because it gives the raw performance of some of the major building blocks used to construct a cluster application, and faster communication often translates to faster application performance. However, caution must be exercised when interpreting these measurements or comparing two communication subsystems. First, speeding up a communication subsystem by some amount does not mean the application will be faster by the same amount. If, for example, an application spends 30% of its time in communication, doubling the speed of the subsystem would improve the application performance by no more than 15%.

Furthermore, it is important to understand that some applications are limited by the latency of small messages, whereas others are constrained by the throughput of large messages. Whether message latency or throughput is more important to performance in a specific case depends on how the application is organized. To make predictions based on raw MPI performance requires a fairly sound understanding of the inner workings of the specific application.

For this report we have used one or more tests from each performance class. For point-to-point communication, we used the PingPong test. For one-to-many operations, Bcast was used. For many-to-one operations, we used Reduce. And for many-to-many operations, we used both Allreduce and Barrier.

3.4 NAS Parallel Benchmarks

The NAS Parallel Benchmarks were developed at the Numerical Aerospace Simulation facility at NASA Ames Research Center. The benchmarks were developed for the performance evaluation of highly parallel supercomputers. They consist of five parallel kernels and three simulated application benchmarks. Together they roughly approximate the computation and data movement characteristics of large-scale Computational Fluid Dynamics (CFD) applications. The kernel benchmarks are CG, EP, FT, IS, and MG. The application benchmarks are BT, LU and SP.

BT is a simulated CFD application that uses a 3-dimensional block-tridiagonal solver. It is a highly efficient, implicit method of solving CFD problems that performs multiple sweeps across the problem space in different dimensions. This does not allow data to reside in cache for long and requires frequent accesses to memory. It is restricted to clusters where the number of processors is an integer square, such as 16 or 25.

CG is a kernel that uses the conjugate gradient method to compute approximate eigenvalues. This kernel is typical of algorithms that use unstructured grids. Such algorithms are often dominated by the performance of unstructured matrix-vector multiplication and irregular long distance communication. It is restricted to clusters where the number of processors is an integer power of 2, such as 16 or 32.

EP is an “embarrassingly parallel” kernel. It calculates a sequence of random numbers using Gaussian deviates. This is a common method for generating random numbers, which uses a very high proportion of scalar floating-point operations. The benchmark is very processor core (ALU) intensive, and all data are either in registers or in L1 cache. It requires almost no communication or memory performance. It can be run on a cluster with any number of processors.

FT is a 3-D partial differential equation solver that uses Fast Fourier Transforms (FFTs) at its core. Its performance is dominated by long distance communication, and its behavior is similar to many CFD “spectral” codes. It is restricted to clusters where the number of processors is an integer power of 2.

IS is a kernel that sorts a large set of integers. This operation is important to “particle method” codes. It is dominated by both integer and communication performance. It is restricted to clusters where the number of processors is an integer power of 2.

LU is a simulated CFD application that uses an LU diagonalization method. This is a version of the well known *applu* benchmark [13][14]. It is restricted to clusters where the number of processors is an integer power of 2.

MG is a simplified multigrid kernel. Its performance is dominated by highly structured short and long distance communication. Multigrid algorithms use Cartesian (rectangular) grids to represent the data space because of their simplicity. But Cartesian grids can be inefficient when there are large differences in the amount of detail that needs to be represented throughout the grid. To avoid this problem, Multigrid methods use a hierarchy of Cartesian grids to represent the data space, using finer grids where more detail is needed, and coarser grids where it is not. It is restricted to clusters where the number of processors is an integer power of 2.

SP is a simulated CFD application that uses an Alternating Direction Implicit (ADI), scalar pentadiagonal solver. It is restricted to clusters where the number of processors is an integer square.

Each kernel or application supports multiple classes of input data. The classes are labeled, from smallest to largest, A, B, C, D and W. For these tests we use only the class C data sets.

4. MPI Operations

4.1 Functional View of MPI

The Message Passing Interface (MPI) [15] is a general-purpose interface for passing structured, typed messages across network interconnects within a cluster. It was designed primarily to meet the needs of the scientific and technical community. It supports many of the basic operations needed to write Single Program, Multiple Data (SPMD) and Multiple Program, Multiple Data (MPMD) applications. Operations include point-to-point, parallel and collective communication operations.

MPI is an abstract standard. There are many different implementations of MPI, both public and proprietary. Popular examples of freely available implementations for Ethernet include MPICH [16] and LAM [17]. Specialized interconnects, such as InfiniBand, usually have a vendor-supported implementation. For these tests we used Voltaire's release 3.0.0-16, which is based on MVAPICH from the Network-Based Computing Laboratory of The Ohio State University [18].

Point-to-point operations are those that communicate between two single nodes. The most common and best understood example of that is the send/receive pair. When a single node within a cluster wishes to communicate with another single node, the first sends a message using the send operation. The second node receives the message when it is ready by executing a receive operation. In this way any two nodes may communicate.

Parallel and collective operations are built up from point-to-point operations. Parallel operations are essentially simple collections of point-to-point operations that occur across the cluster in a coordinated fashion. For example, nodes within a cluster might be arranged (logically) within a grid, where each node within the interior of the grid has a neighbor to the north, south, east and west. At some point in the computation it may be necessary for each node to exchange data with each of its neighbors. This can be done using a series of send/receive pairs, but doing so can be tricky and error-prone. Parallel operations are designed to accomplish the same work with fewer problems. Figure 6 shows an example of a parallel send/receive operation. Each message is independent of every other message, therefore the entire operation can be done completely in parallel.

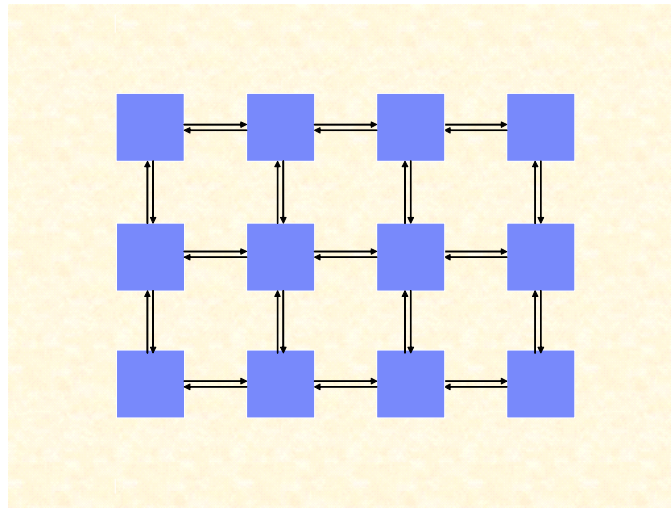


Figure 6. Parallel Data Exchange As Done By `MPI_Sendrecv`

Collective operations are similar in some respects to parallel operations, but there is a degree of interaction among the messages. They are considered collective because all nodes must participate in the operation. Perhaps the most common collective operation is the broadcast. In a broadcast operation a single node sends data to every other node. This allows all nodes to make decisions, such as whether to continue computing, based on the same information. One way to broadcast data would be for the sender to send separate messages to every node in the cluster. A more efficient approach would be to send the message to another node, and then the sender and receiver would send the data to two more nodes, and so forth, until all nodes have the data. This forms a tree with the sender at the root. Even more efficient techniques can be constructed when other network components, such as intelligent switches, are used.

Another common collective operation is known as a reduction. A reduction operation takes a collection of data and reduces it to a single value. For example, adding up a series of numbers is considered a sum reduction—it reduces a vector to a scalar by summing the values. Reductions may also take multidimensional objects, such as a matrix, and perform the operation across only one dimension. If the initial object has n dimensions, the result after the reduction will have $n-1$ dimensions.

It is very common for the data that is to be reduced to be spread across many or all of the nodes in the system. A naive programmer attempting to code a reduction using point-to-point communication could easily use inefficient techniques. One obvious way is for all nodes to send the input data to a single node, which receives and sums the data. For small clusters this may work fine, but for large clusters the receiving node would quickly become overwhelmed. A better technique would be to have each node send its data to another, which then performs a partial reduction. The partially reduced data would then be sent to another node, which would reduce the data further, and so on, until all data were reduced. The resulting pattern looks like a tree, where the root node has the final result of the reduction. Even more efficient techniques can be constructed depending on how the network is organized.

A final example of collective operation combines a reduction operation with a broadcast. It is called the all-to-all reduction. Suppose that the application is attempting to approximate some calculation that is too difficult to compute directly. One technique is to divide the work and data among all nodes, allow them to compute for a while, and then check the error to see whether it is appropriate to continue computing. The key is that the error must be checked across all nodes, and all nodes must make the same decision whether to continue or to stop. An intuitive way to do this would be to allow each node to compute its own error locally, use a reduction to add up all of the local errors to form a global error value, and then broadcast the global error back to all of the nodes. A faster way is to perform the reduction in such a way that all nodes are the root of a reduction tree; therefore, all nodes have the result at the end of the reduction.

So, a common taxonomy of MPI operations organizes them as point-to-point, parallel and collective operations. For the sake of describing performance, a more appropriate taxonomy would organize the operations by their communication patterns. Routines that may have very different behaviors from a functional standpoint can perform similarly when their communication patterns are similar. The four communication patterns that emerge from a careful examination of MPI operations are point-to-point, one-to-many, many-to-one, and many-to-many operations. A send/receive pair is an example of point-to-point communication. Broadcast operations are an example of one-to-many operations. Simple reductions are an example of many-to-one operations. All-to-all reductions are an example of many-to-many operations. Although there are occasional differences, operations within a category do show similar performance, so organizing MPI in this way allows us to simplify the data we must compare to understand MPI performance.

4.2 One-to-One Operations

Point-to-point operations have two parts to consider: a message must be sent and that message must be received. In many ways send and receive are mirror images of each other. How well they perform depends on how they are implemented. In the most generic sense, a send operation must package the data, or payload, to be sent into a message. The data is collected into a message by an MPI library such as MPICH. The message is handed to the adapter and shipped across the network.

Each packet moves through the network through a series of one or more switches until it arrives at its final destination. Each switch reads and modifies the contents of the packet headers to move the packet closer to its destination. When a packet arrives, it must be processed and possibly combined with other packets to recreate the original message. The message is held in a queue until a process asks for it.

When an application is ready for the data, it calls a receive operation in the MPI library. The MPI library asks the operating system for the data; the operating system must, in turn, request the data from the protocol layer (for example, TCP). If the data has not yet arrived, the operating system may suspend the process execution until the data arrives, or it may return a status indicating the data is not yet available. The former action is considered a blocking receive, while the latter is a non-blocking receive. If the data is ready and available, the data is passed to the MPI library. The MPI library extracts the data and hands it to the application.

4.3 One-to-Many Operations

One-to-many operations propagate data from a single node to a collection of nodes. The target nodes may be the entire cluster or it may be a subset of the cluster. They are sometimes called fan-out operations because of the way data fans out from the source. Broadcast is the best known example of this type of operation. In MPI this is implemented as the `MPI_Bcast` routine. Other examples are the scatter operations `MPI_Scatter` and `MPI_Scatterv`. In each case there is a root node, where the data originates, and leaf or target nodes, to which the data is sent. The difference between fan-out operations is primarily in determining what data is sent to each leaf node.

Fan-out operations may be implemented in different ways. As mentioned previously, an especially naive and inefficient method is for the root to send the data directly to each of the leaves. Although it is especially easy to code, it scales poorly to large clusters because the root node must send a sequence of $n-1$ messages in a cluster of size n . The root becomes the bottleneck. Another way of saying it is that the number of messages is of order n , or $O(n)$, and the number of messaging stages is also $O(n)$.

A faster, binary method is illustrated in Figure 7. In the first stage of the operation, the root sends its message to another node, in this example node 4. In the next stage, the root and node 4 both send the message to other nodes. In this way the number of nodes that have received their data doubles at each stage. The same number of messages, $O(n)$, is sent, but because the messages are sent in parallel, the number of messaging stages is reduced to $O(\log_2 n)$, which is much faster than $O(n)$.

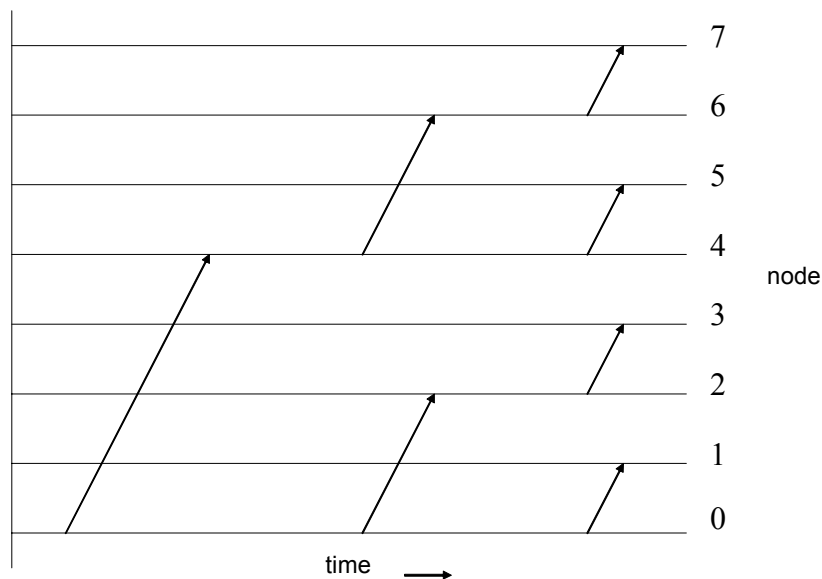


Figure 7. Representative Message Traffic Pattern for One-to-Many MPI Operations

Something worthy of note here is that, while this is a very fast and very general method, there may be other implementations that are faster still. For example, one might take advantage of special hardware built into switches or adapters, or implement parts of the operation within the

device drivers. In this way one might reduce the size or number of messages to be sent, or reduce the overhead associated with each message. Each implementation of the MPI software and hardware stack attempts to provide the best possible performance, and each may deviate, perhaps even substantially, from this binary method. What is important is that these operations form an equivalence class. Because they can be implemented in similar ways, we need only to examine one of the members of the equivalence class to understand the performance of a communication stack.

4.4 Many-to-One Operations

In some sense the many-to-one operations, also known as fan-in operations, are the inverse of the one-to-many operations. Examples of these operations are `MPI_Reduce`, `MPI_Gather` and `MPI_Gatherv`. In these operations the data is gathered together from across the cluster and accumulated on a single node. The data may be processed along the way, as occurs in reduction operations, or accumulated in a large buffer at the destination point, as occurs in gather operations.

Terminology is similar to the one-to-many operations; that is, data is gathered from multiple leaf nodes to a single root node. Also similar is the fact that the messages can be arranged in a tree-like structure by reversing the direction of the message transmission, as illustrated in Figure 8. As with fan-out operations, the number of messages is $O(n)$ and the number of messaging stages is $O(\log_2 n)$.

With reductions, MPI implementers also have additional opportunities to take advantage of special hardware and software because the MPI stack is required to perform additional processing on the data as it is accumulated.

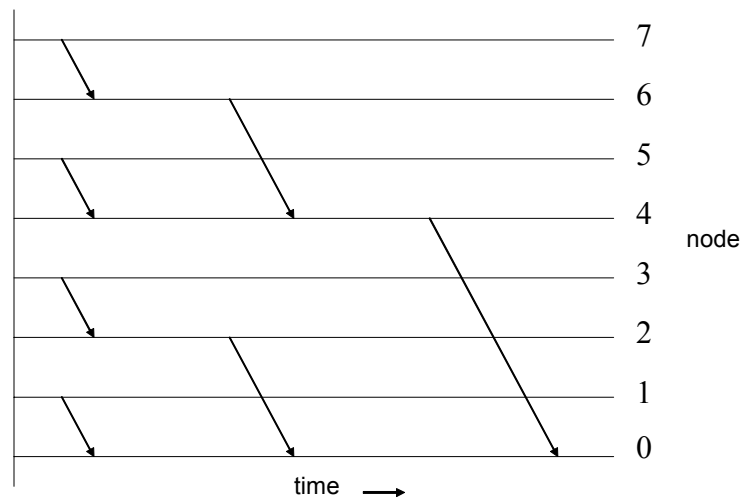


Figure 8. Representative Message Traffic Pattern for Many-to-One MPI Operations

4.5 Many-to-Many Operations

Many-to-many operations are similar to a combination of many-to-one and one-to-many operations. Examples include all-to-all gather, scatter, broadcast and reduce operations. Barriers

are also an example of a many-to-many operation. In each of these, every node is a leaf node, and every node is a root. Every node contributes data to the operation, and every node receives a copy of the final result. Each node sends and receives a message in every stage, and, as before, there are $\log_2 n$ stages. That means the number of messages is $O(n \log_2 n)$ and the number of messaging stages is $O(\log_2 n)$, assuming the switch has sufficient bandwidth.

An efficient binary algorithm for many-to-many operations maps message transmissions to a butterfly network, as shown in Figure 9. In essence, each node is identified as a unique, consecutive integer beginning at zero. The destination node for each stage is determined by inverting bits in the binary representation of the source node identifier. Starting from either the least significant bit or most significant bit in the node identifier, the first bit is inverted to determine the target node in the first stage, the second bit is inverted in the second stage, and so forth.

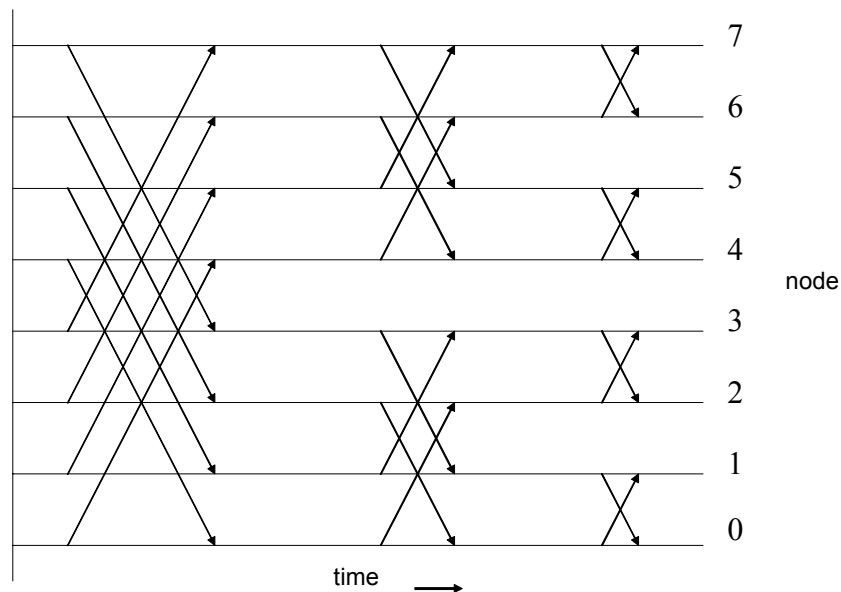


Figure 9. Representative Message Traffic Pattern for Many-to-Many MPI Operations

Without a formal framework, it may be difficult to see the intuition behind how this method works, but it may help to observe that there is a path from every node to every other node in the time line. For example, consider an arbitrary node, say node 3, in Figure 9. Node 3 sends to node 7 in the first stage. Node 3 sends to node 1, and node 7 sends to node 5 in the second stage, so after the second stage completes, nodes 1, 3, 5 and 7 have the data. In the third stage, node 1 sends to node 0, node 3 sends to node 2, node 5 sends to node 4, and node 7 sends to node 6. After that stage completes, all nodes have access to the data from node 3. Similar coverage can be shown for each node in the figure.

As mentioned before, those who implement the MPI software and hardware stack may be able to improve on this method, but such improvement would apply to all operations within this class.

5. Server Performance

The intention of this paper is to measure and document the interconnect performance. However, interconnects, such as InfiniBand, do not operate in isolation from the rest of the system. They use system resources and therefore can run only as fast as those subsystems allow. Furthermore, the benchmarks used to measure interconnect performance also use resources, even those not specifically required to operate the interconnect. For this reason, we briefly review server performance. A more complete description of each system can be found elsewhere [19].

5.1 Processor Core Performance

The processor core represents the brains of the system. It contains multiple functional units, each of which implements some key function needed for operation. Commonly considered units include simple integer units for implementing addition, subtraction and multiplication; complex integer units for implementing integer division; floating-point units; load and store units; and others. Of the cluster benchmarks we use, several are processor-core-limited. They are primarily limited by their floating-point performance. We measure core floating-point performance with the Linpack benchmark, the results of which are shown in Figure 10.

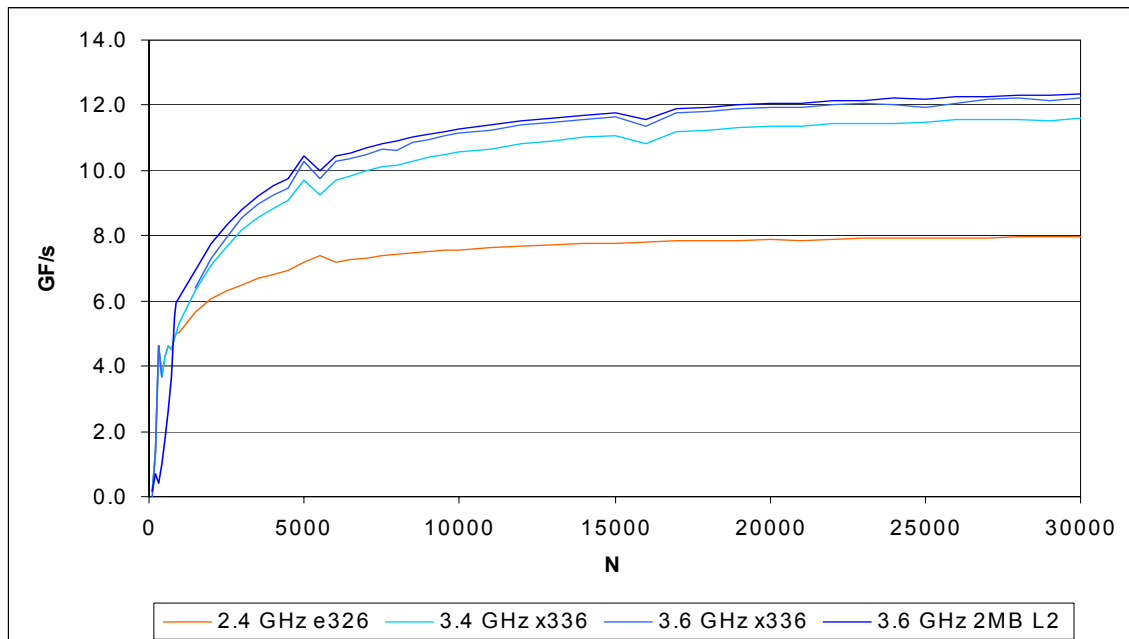


Figure 10. Linpack Performance

It is evident from this chart and other experiments that floating-point performance is proportional to the processor clock speed (and the number of cores available) for any given size of problem, though cache size also has a minor effect. The reason is that the processors have the same number of floating-point units, in this case one SSE2 unit per core. The architectures are similar enough that they all achieve nearly the same efficiency, around 80%, which reduces the performance problem to one of which system can produce more clock cycles per second. In this case the Xeon processors win with their higher processor clock speed. Note that Linpack measures only vector floating-point performance, so scalar performance may be different.

5.2 Memory Performance

Memory performance is only a little more complicated. In this case we are interested in memory throughput rather than latency. It depends on the number of channels to memory, the clock frequency of the memory DIMMs, and the type of memory. Opteron processor-based systems support two memory channels per processor. A two-socket system, such as the e326, supports two or four channels of 333 MHz or 400 MHz DDR memory. A Xeon processor-based system, such as the x336, requires a shared front-side bus and a separate memory controller. This memory controller supports only two channels of 400 MHz DDR2 memory. From Figure 11 one can see that performance is very nearly proportional to the number of channels and the speed of memory, and also that DDR memory, clock for clock, is slightly faster than DDR2 memory.

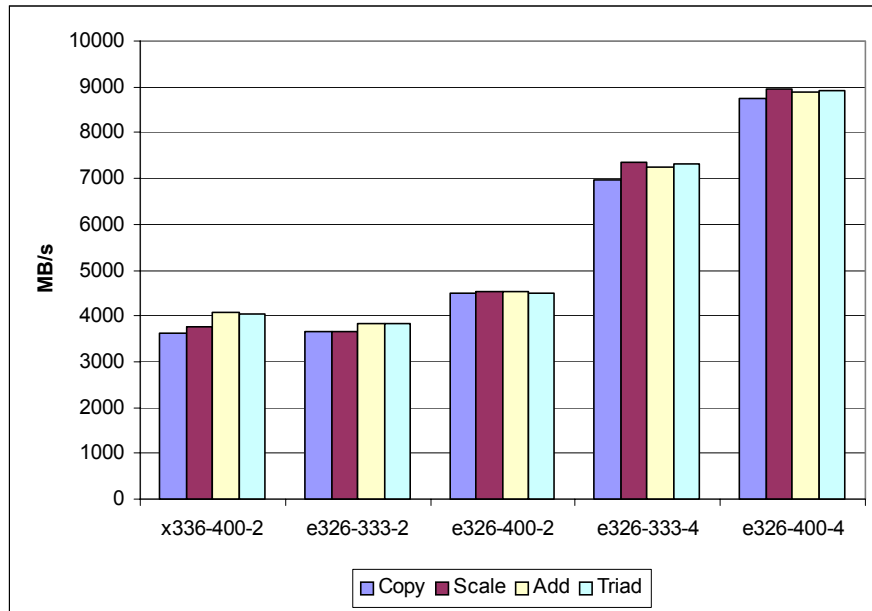


Figure 11. Memory Bandwidth (STREAM)

5.3 I/O Slot Performance

A third subsystem, for which we have no benchmark, is the I/O slots used to connect the InfiniBand adapters to the system. There are two types of slots available to us for these tests, PCI-X and PCI-Express. PCI-X is a 64-bit-wide, half-duplex bus protocol that operates at 66, 100 or 133 MHz bus frequencies. As a bus protocol, PCI-X can support multiple devices sharing the bus, but in our tests there was only the single adapter. All PCI-X is half-duplex, so the bus is only capable of transmitting data in one direction at a time. To transmit in the opposite direction the bus must become idle for a short period before transmission can begin in the opposite direction.

The PCI-X InfiniBand adapters are capable of operating at speeds of up to 133 MHz. The InfiniBand interface and internal data paths of the adapter are actually much faster than this, so they are necessarily limited by the speed of the PCI-X bus. The PCI-X protocol allows the bus and the adapter to negotiate the speed, so when the bus is limited to 100 MHz, as is the case in some of our tests, the adapter automatically reduces its speed to match the bus. At 100 MHz the bus supports 800 MB/s bandwidth. At 133 MHz it supports 1067 MB/s bandwidth.

PCI-Express, or PCI-E, is a full-duplex, point-to-point bus that operates at 2.5 GHz. PCI-E supports multiple sizes from 1 to 16 lanes in width. Since it is bidirectional, an 8x bus, as we have in the x336, has 8 data lanes running in each direction. The bus is 8/10 bit encoded, giving the bus 1000 MB/s bandwidth in each direction.

6. Interconnect Performance

Before we examine the performance, a brief explanation of the charts will make them more understandable. The data label of each system configuration is encoded in what we hope is an obvious way. The labels start with the system type, followed by the processor frequency, then by the memory frequency, and last of all, the I/O slot frequency. So “e326 2.4 400 133” represents a cluster of eight nodes of dual processor, 2.4 GHz Opteron processor, e326 systems using 400 MHz DDR memory and 133 MHz PCI-X I/O slots. The x336 clusters use 64-bit Intel Xeon processors. Xeon processors are assumed to be Nocona processors with 1MB L2 cache, unless the processor frequency is followed by an “i” (for Irwindale). Irwindale processors are very similar to Nocona processors, but Irwindale processors have a 2MB L2 cache.

6.1 Unidirectional Point-to-Point Communication

There are generally two aspects to consider in point-to-point communication, namely latency and throughput. An application that sends many small messages over long distances may be latency-limited, whereas one that sends many large messages is likely to be throughput-limited.

The PingPong benchmark measures both latency and throughput for point-to-point communication. It consists of a pair of processes that exchange messages of various sizes. The first process sends a message to the second process. Upon receiving the message, the second process resends the message back to the first. The results of this test are shown in Figure 12.

All systems are shown in this chart; several of these systems have very similar performance. For example, both x336 systems with 133 MHz PCI-X are similar, as are the e326 systems with 133 MHz PCI-X, and all three x336 systems with PCI-Express. In later charts we may not show all examples from each group, to reduce chart clutter, unless there is a good reason to do so.

Several interesting observations can be made from this chart:

1. It is the x336 with 133 MHz PCI-X that has the greatest latency, and not an e326 with the (slower) 100 MHz PCI-X.
2. The e326 with 100 MHz PCI-X is next fastest.
3. The e326 with 133 MHz PCI-X is about one to two microseconds slower than an x336 with 8x PCI-Express.
4. Systems with DDR400 are only marginally faster than those with DDR333, which implies that InfiniBand performance is not gated by the speed of memory.
5. Faster processor clock speed makes no difference on Xeon with PCI-X or PCI-Express, although a larger cache seems to slow communication slightly.

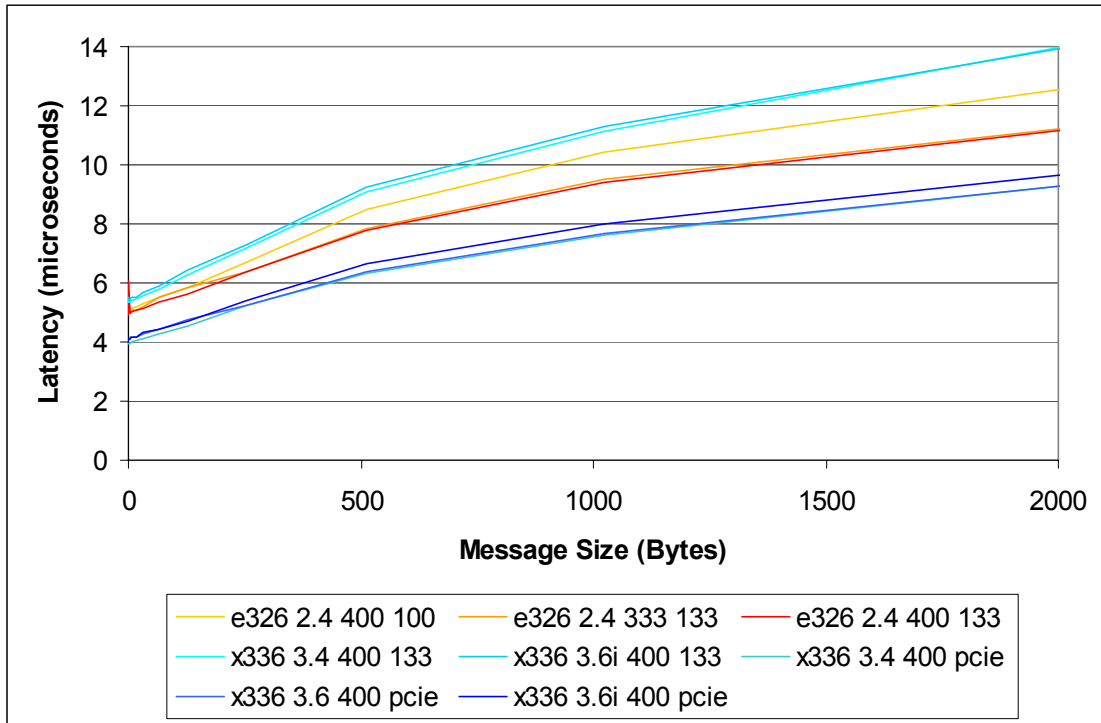


Figure 12. Unidirectional Point-to-Point Latency (PingPong)

The PingPong benchmark can also be used to measure throughput. Just as it measures unidirectional latency, it also measures unidirectional throughput. This is an issue because of how the PCI-X and PCI-E busses operate. The PCI-X bus is half-duplex, so the unidirectional throughput should be approximately the same as the bidirectional throughput. In contrast, the PCI-E bus is bidirectional, so the unidirectional throughput should be half of the bidirectional throughput.

We can see from Figure 13 that the throughput performance approximates the latency performance in that the x336 with PCI-X has the worst latency and the worst throughput. The x336 with PCI-E has both the best latency and the best throughput, as it does for the e326.

The efficiency of each system is also important. It indicates, among other things, how much improvement may be available if drivers are improved. The results are in Table 1.

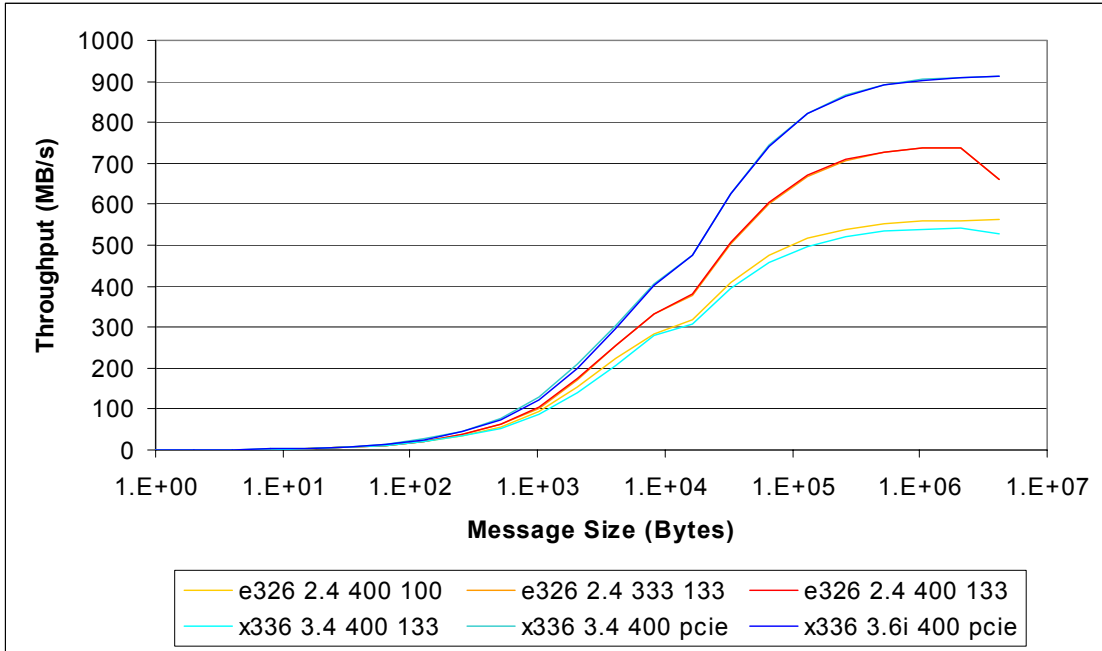


Figure 13. Unidirectional Point-to-Point Throughput (PingPong)

Table 1: One-Way Latency and Throughput (PingPong)

Server				PingPong		
System	CPU (GHz)	Memory (MHz)	Slot Speed	Latency (μ sec)	Throughput (MB/s)	Efficiency
e326	2.4	400	100 MHz	5.13	562	70%
e326	2.4	333	133 MHz	4.99	738	69%
e326	2.4	400	133 MHz	4.93	739	69%
x336	3.4	400	133 MHz	5.36	541	51%
x336	3.6 2M	400	133 MHz	5.47	539	51%
x336	3.4	400	PCI-E 8x	3.95	914	91%
x336	3.6	400	PCI-E 8x	4.05	914	91%
x336	3.6 2M	400	PCI-E 8x	4.05	914	91%

6.2 Bidirectional Point-to-Point Communication

Two methods are available to exchange data between two processes. The first is where one process sends to another, and when the transmission is complete, the second process sends to the first. This is the method used in the Exchange benchmark. Processes pair up to exchange messages. The first process in each pair sends the message, the second process receives it, and then the reverse occurs.¹ It is different from PingPong in that Exchange has many pairs of processes sending and receiving simultaneously. The second method is to use the `MPI_Sendrecv` subroutine. This routine coordinates the exchange of messages efficiently and reliably, reducing the likelihood of programming error.

From the experiments we can see that `MPI_Sendrecv` is only partially successful. (See Figure 14 through Figure 17. The results are also summarized in Table 2.) The `MPI_Sendrecv` latencies are indeed smaller, as one would expect, and the maximum throughput values are very similar. But the throughput results favor manually exchanging messages over using `MPI_Sendrecv` for all message sizes until the messages become very large, around 1 MB. The most visible disparities occur with message sizes from 1,000 to 100,000 bytes.

Other things worthy of note are that the systems with PCI-E have the lowest latencies and highest bandwidths, as expected. But among those systems using PCI-X, the latency behaviors are quite complex. Once again, the fastest PCI-X slots do not always give the lowest latencies. Also, higher memory bandwidth (for example, 333 MHz DDR versus 400 MHz DDR) shows a slight advantage.

Table 2: Two-Way Latency and Throughput

Server				Exchange			Sendrecv		
System	CPU (GHz)	Memory (MHz)	Slot Speed	Latency (μ sec)	Throughput (MB/s)	Efficiency	Latency (μ sec)	Throughput (MB/s)	Efficiency
e326	2.4	400	100	9.82	589	74%	7.16	591	74%
e326	2.4	333	133	9.71	710	67%	7.09	719	67%
e326	2.4	400	133	9.72	789	74%	7.32	788	74%
x336	3.4	400	133	10.1	628	59%	7.33	628	59%
x336	3.6 2M	400	133	10.4	627	59%	7.34	627	59%
x336	3.4	400	PCI-E	7.97	1737	87%	5.74	1735	87%
x336	3.6	400	PCI-E	8.2	1737	87%	5.79	1734	87%
x336	3.6 2M	400	PCI-E	8.3	1737	87%	5.81	1735	87%

1. Exchanging messages in this manner is reliable, but another method commonly used is not. It is common for both processes to send data, and then for both to receive it. This works for small messages, but a deadlock may occur in some cases if there is not enough buffering within the system to fully queue all messages.

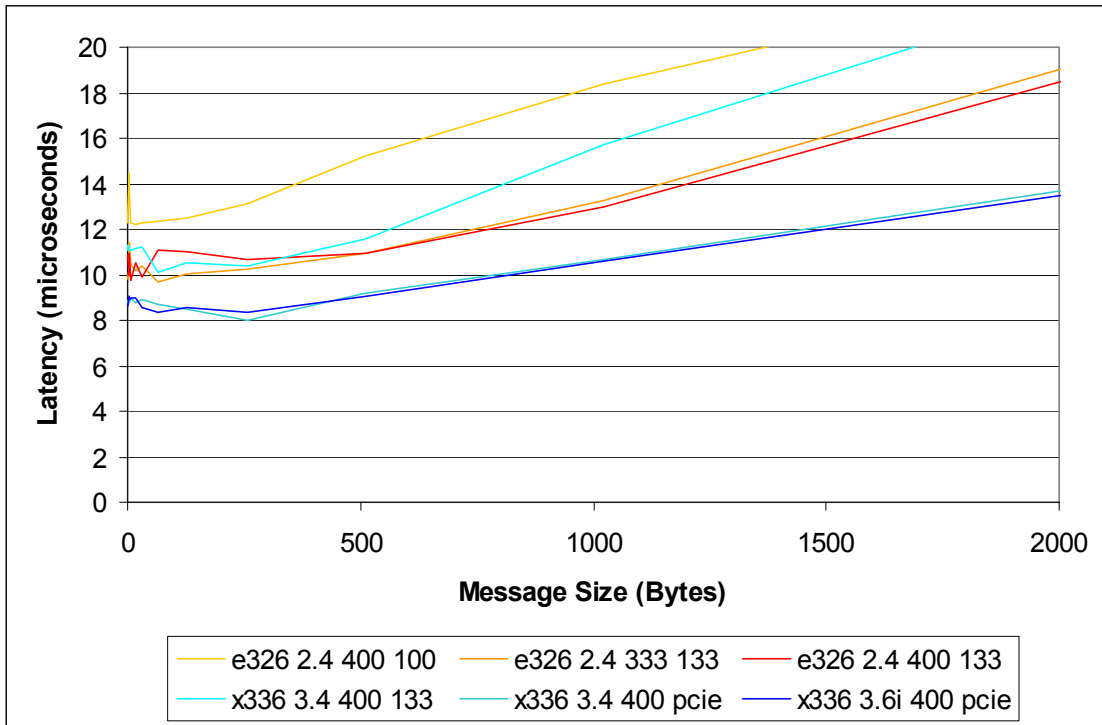


Figure 14. Bidirectional Point-to-Point Latency (Exchange)

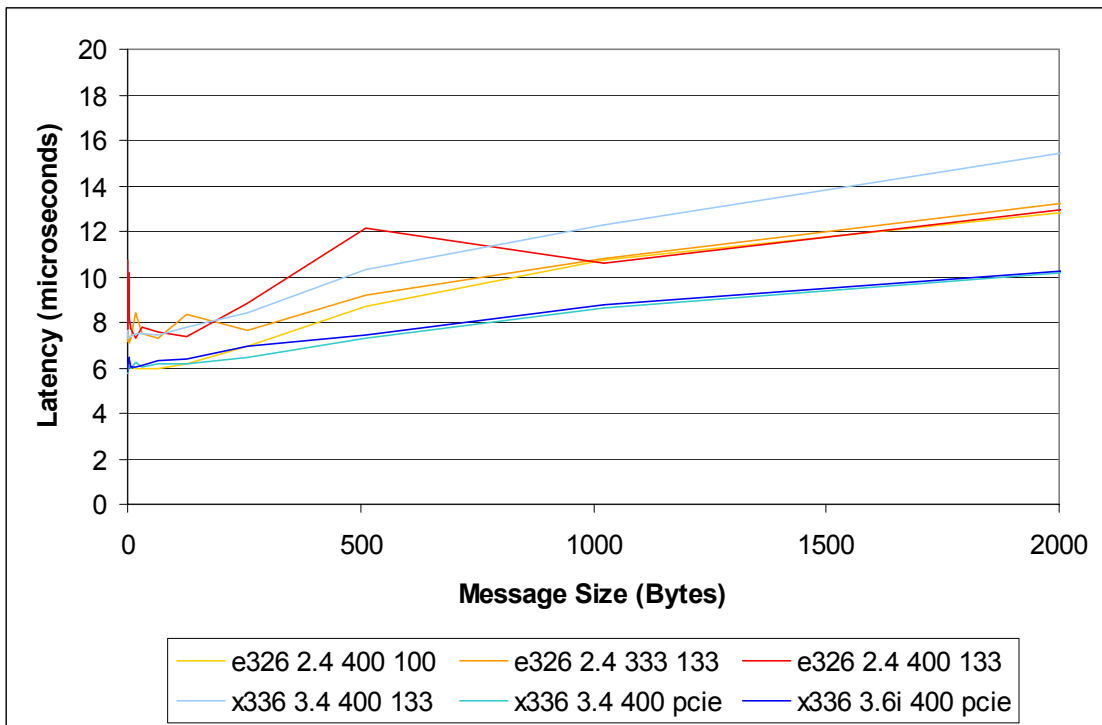


Figure 15. Bidirectional Point-to-Point Latency (Sendrecv)

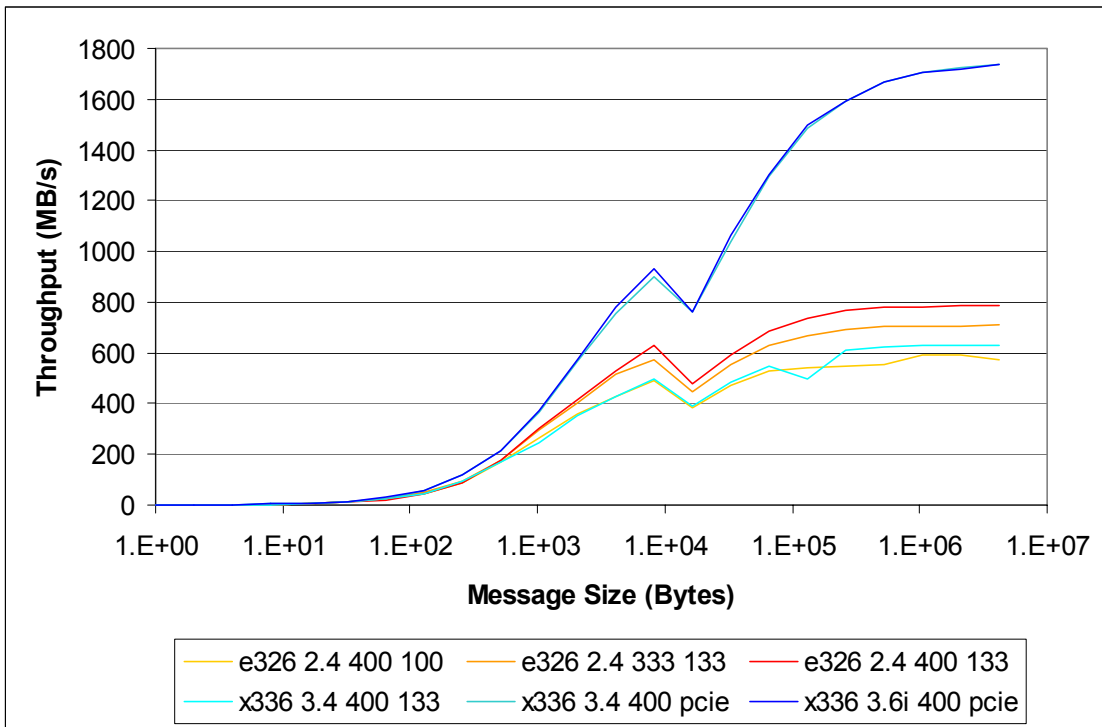


Figure 16. Bidirectional Point-to-Point Throughput (Exchange)

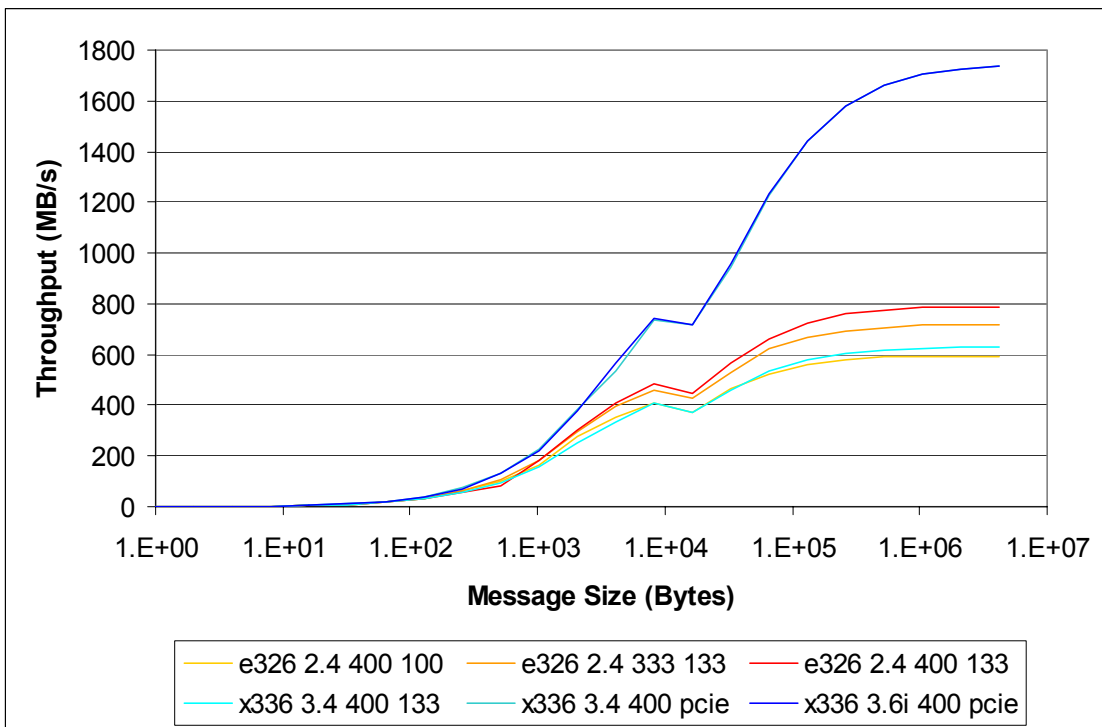


Figure 17. Bidirectional Point-to-Point Throughput (Sendrecv)

6.3 One-to-Many and Many-to-One Performance

Unlike point-to-point operations, bandwidth and throughput measures are not well-defined for one-to-many operations, nor are they well-defined for many-to-one or many-to-many operations. For this reason we will focus on latency measures in this section and the next.

As we can see with Bcast (Figure 18) and Reduce (Figure 19), the initial pattern established with PingPong continues. (See Section 6.1 on page 19.) The x336 with PCI-E provides the fastest communication, followed by the e326 using 133 MHz PCI-X, then the e326 using 100 MHz PCI-X, and last, the x336 using 133 MHz PCI-X.

In Bcast, PCI-Express is a full 20% faster than its nearest competitor. However, unlike the PingPong tests, the effects of memory bandwidth are becoming more pronounced. At 2KB, the difference in latencies between 333 MHz DDR and 400 MHz DDR for the PingPong test was about 1 microsecond in 19, or about 5%. The same size message with Bcast shows a difference of about 2 microseconds in 19, or about 10%.

In Reduce, PCI-Express also shows a healthy advantage over its nearest competitor, but only by about 12%, and the effects of memory bandwidth are not pronounced. But in both tests Bcast and Reduce, there is a pronounced difference between 133 MHz PCI-X on the x336 and on the e326, of about 20%.

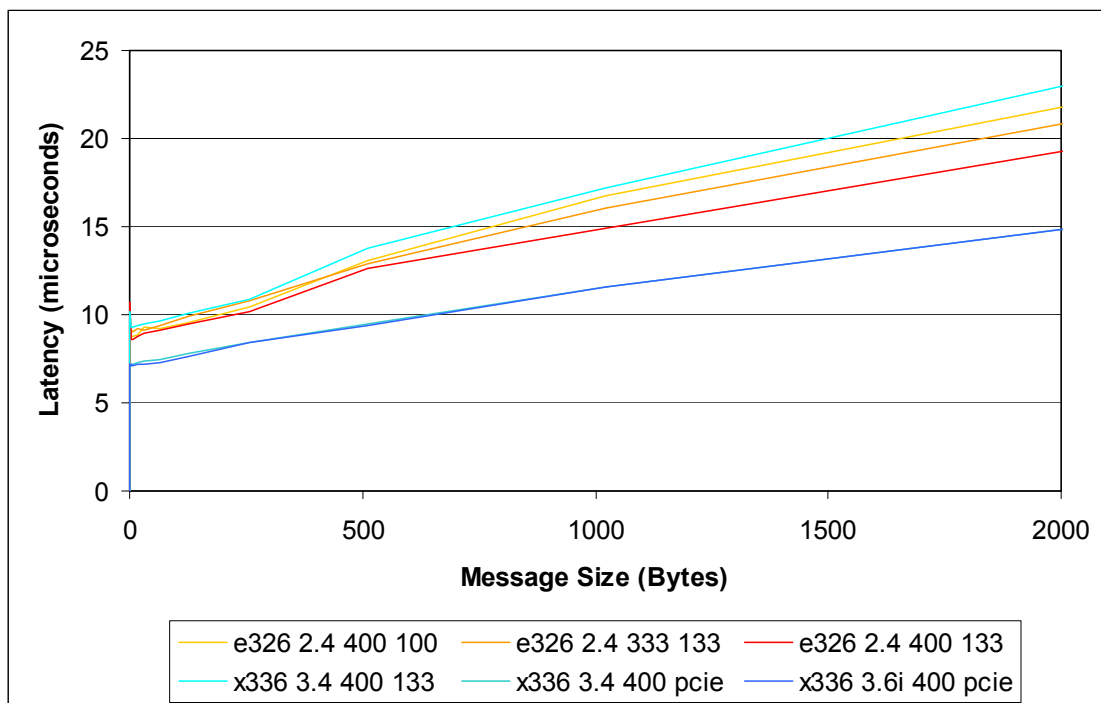


Figure 18. One-to-Many Latency (Bcast)

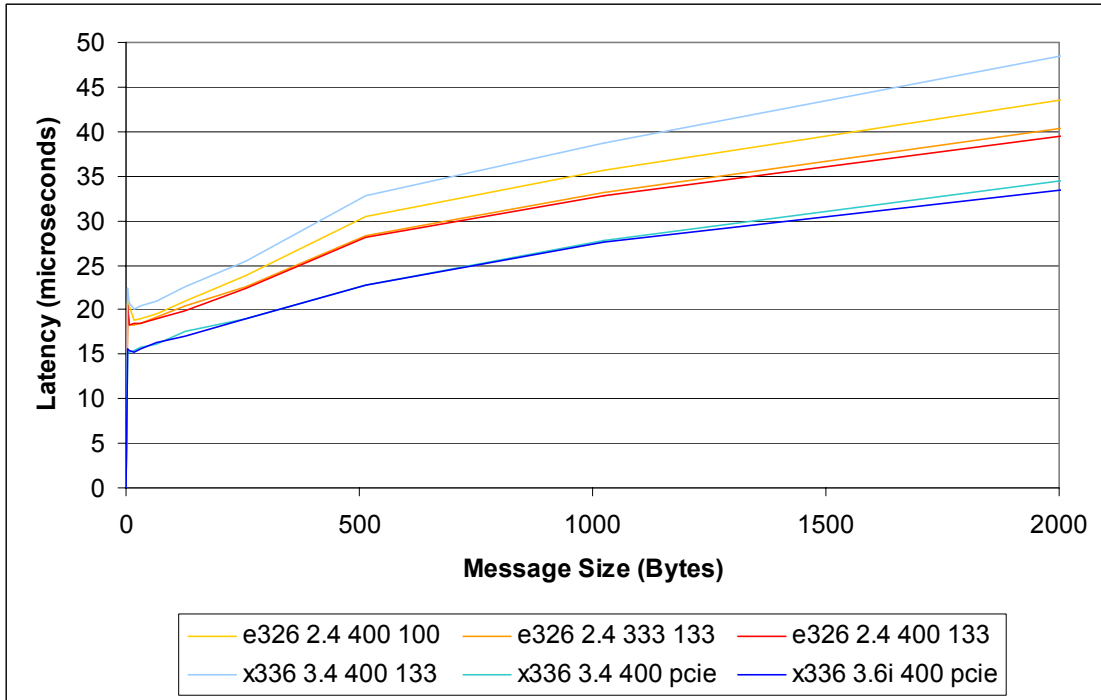


Figure 19. Many-to-One Latency (Reduce)

6.4 Many-to-Many Performance

Once again we can see the effect of slot performance on network performance, see Figure 20. Those systems with PCI-E outperform all others. The e326 systems using 133 MHz PCI-X outperform the e326 with 100 MHz PCI-X. The x336 with 133 MHz PCI-X shows the highest latencies.

A barrier operation is a special case of the `MPI_Allreduce` operation. `MPI_Barrier` is, in essence, an `MPI_Allreduce` using a message that contains no data. The reason is that a barrier provides the same synchronization that is implicitly provided in the `MPI_Allreduce` operation.

Barrier performance is shown in Figure 21. The chart shows that there are no clear differentiators here except the difference between PCI-X and PCI-E. The PCI-E performance is always about 25% faster than the PCI-X solution.

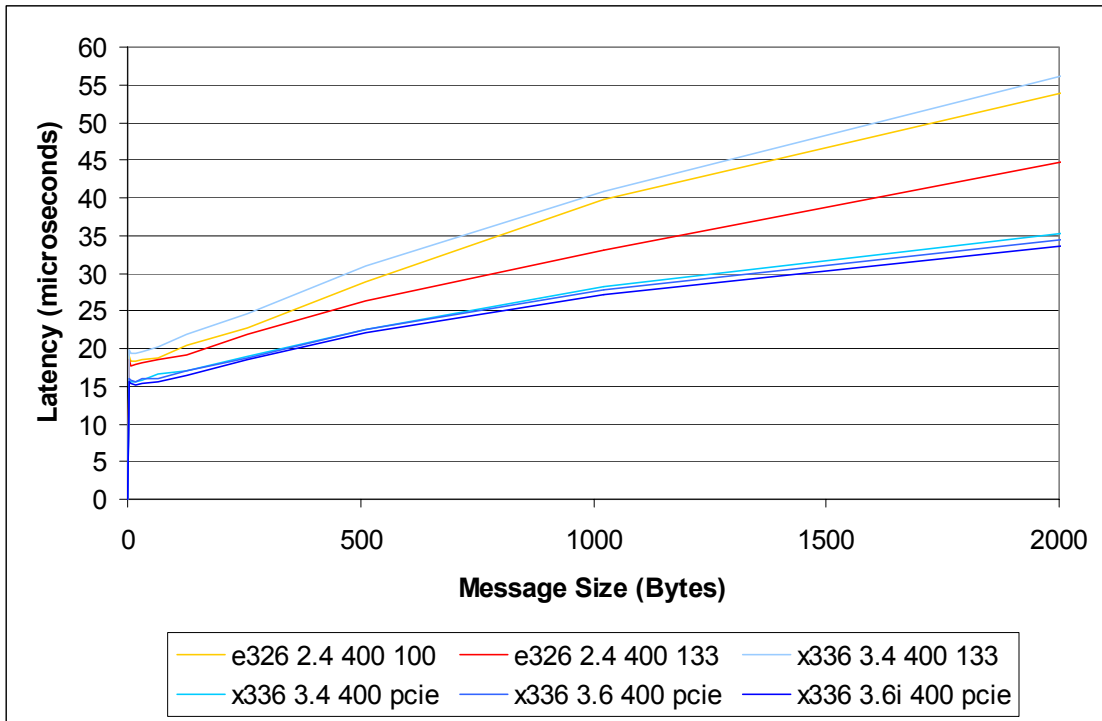


Figure 20. Many-to-Many Latency (Allreduce)

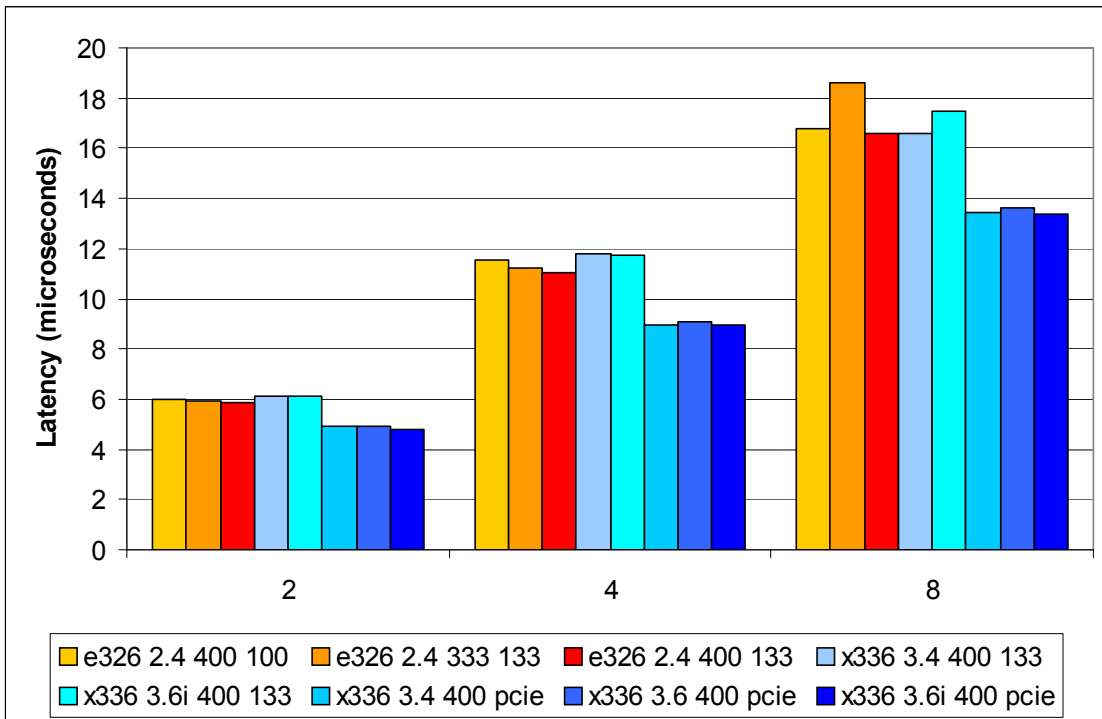


Figure 21. Barrier Latency

7. NAS Parallel Benchmarks

In this section we examine the performance of the NAS Parallel Benchmarks from several different perspectives. We want to understand how well each server performs, to be sure, but we also want to understand what portion of the system each benchmark is stressing. Each piece of information is important, but for different reasons. Knowing that one system is faster than another on one of the benchmarks tells us that applications that behave similarly to the benchmark will also run faster on that system. For example, by knowing that BT runs faster on an e326, we also know that applications that behave in ways that are similar to BT will also run faster on an e326.

But we also want to know what subsystems each of the benchmarks use, so that when something changes, for example, new servers or interconnects arrive, we have some idea what the effect of the changes will be.

As explained in the previous section, the data label of each system configuration is encoded with the configuration. The labels start with the system type, followed by the processor frequency, then by the memory frequency, and last of all, the I/O slot frequency. So “e326 2.4 400 133” represents a 2.4 GHz e326 using 400 MHz DDR memory and 133 MHz PCI-X I/O. Xeon processors are assumed to be Nocona processors with 1MB L2 cache, unless the processor frequency is followed by an “i” (for Irwindale). Irwindale processors have a 2MB L2 cache.

7.1 BT (Block Tridiagonal) Performance

From a description of the benchmark and a cursory examination of the code, one can reason that memory performance is important to this benchmark. Multiple sweeps across different dimensions of a three-dimensional array suggest that data will not stay in cache long. And from the measurements shown in Figure 22, we see this is so.

In our experiments we vary processor clock speed, cache size, memory bandwidth and slot speed. The results clearly show that *only* memory performance has a significant effect. The memory performance of a 400 MHz DDR e326 is 8.9 GB/s, whereas the same system with 333 MHz DDR memory has 7.3 GB/s. The 333 MHz DDR memory is about 20% slower than 400 MHz DDR memory. The BT performance for the same systems is 10,171 and 8,153, respectively, or about 20% slower, which shows a strong correlation.

A similar comparison with the x336 also holds, but not as well. The memory performance of the x336 is slightly more than 4 GB/s. In other words, the e326 memory performance is about 2.2x that of the x336, whereas the benchmark performance is only about 1.7x. There is a strong correlation between memory performance and benchmark performance, but it isn't exact. However, other obvious choices, such as cache size, processor speed and network performance, show no correlation at all.

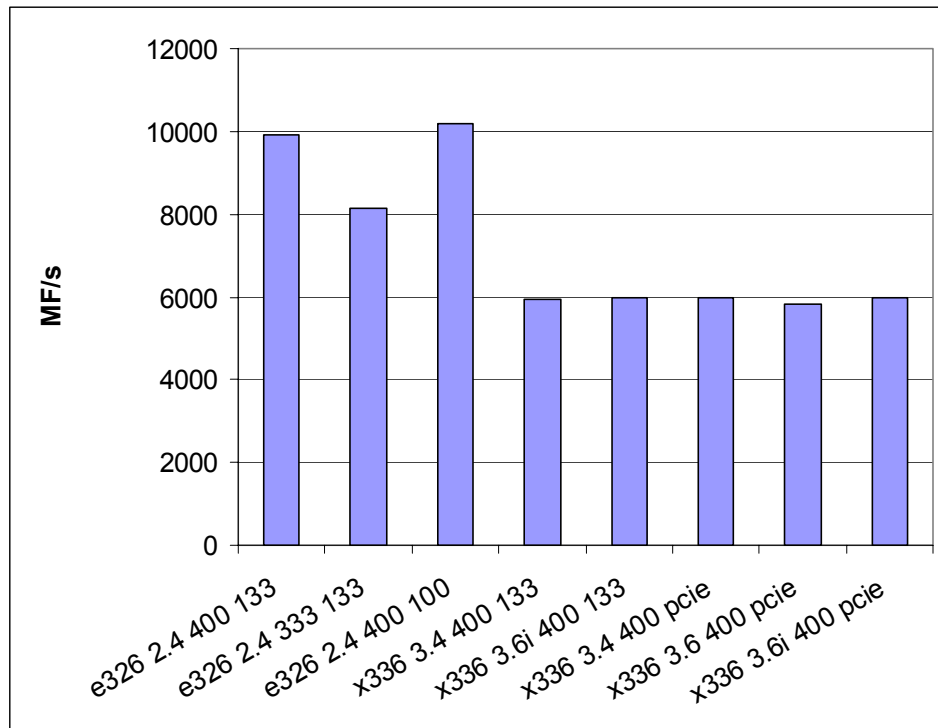


Figure 22. BT Performance

7.2 CG (Conjugate Gradient) Performance

The CG benchmark behaves very differently from BT, as Figure 23 shows. Whereas BT favors Opteron because of its high memory performance, CG favors Xeon for its interconnect performance.

Clearly, the most important aspect of the cluster to CG is the interconnect performance. All of the systems with PCI-E outperform all of the systems with PCI-X. Furthermore, all systems with 133 MHz PCI-X outperform the system with 100 MHz PCI-X. By inspection it is evident that processor performance and cache size are not limitations, as a 3.4 GHz Nocona processor-based cluster outperforms two similar clusters, one with 3.6 GHz Nocona processors, the other with 3.6 GHz Irwindale processors. It is also apparent that memory performance is not a limitation, because all of the Xeon processor-based systems outperformed all of the Opteron processor-based systems.

What is most striking is the apparent contradiction CG presents. In all of the Pallas benchmark tests, the e326 using 133 MHz PCI-X outperformed the x336 using 133 MHz PCI-X. Yet in CG, all of the x336 systems with 133 MHz PCI-X outperformed all of the e326 systems, and no other subsystem seems to be significant. The solution may lie in the specific pattern of communication used, that is, the size of messages sent and how they are distributed around the cluster. We have no answer to this puzzle, but we do find it peculiar and believe it deserves further investigation.

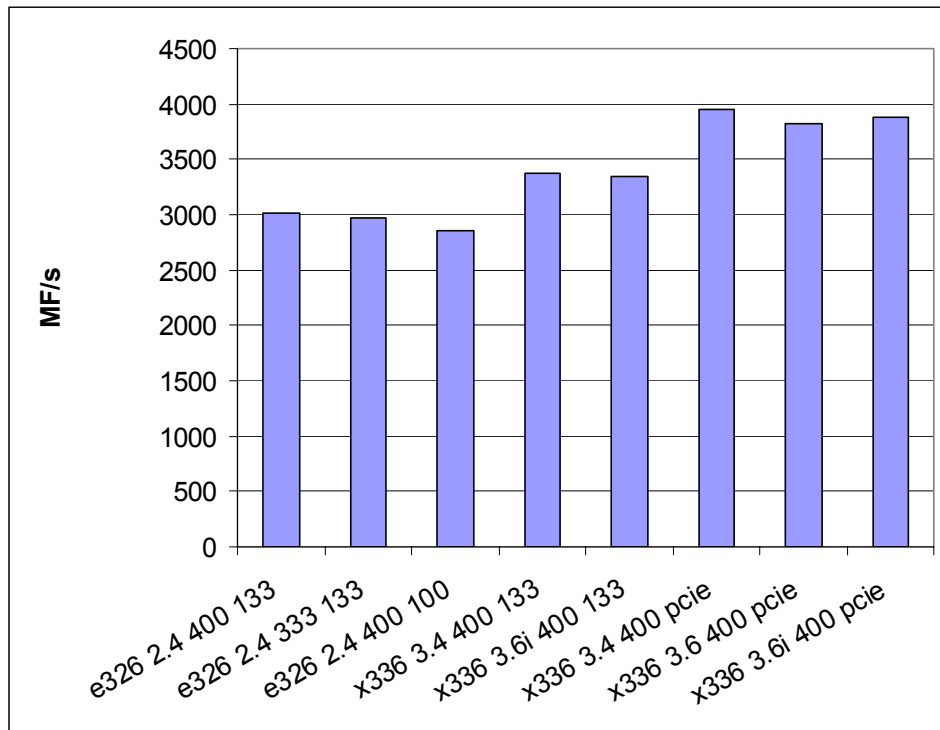


Figure 23. CG Performance

7.3 EP (Embarrassingly Parallel) Performance

The EP benchmark exercises the processor core almost exclusively. The only communication that takes place is the initial barrier that starts the program, and the final barrier that ends it. There are only a few variables, few enough that most or all will fit in the available registers, and any that don't will certainly fit within even a small L1 cache. The performance of this benchmark is shown in Figure 24.

It is quite evident from the figure that all Opteron processor-based systems outperformed all Xeon processor-based systems. This would be no surprise if the benchmark played to one of Opteron's well-known strengths, such as its memory bandwidth, but memory performance is not a factor. This is evident from inspecting the code, and because the system with 333 MHz DDR memory performs at the same level as one with 400 MHz DDR memory. L2 cache size is not a factor, because the e326 cluster outperformed the Xeon cluster, even with Irwindale processors. Network performance is also not a factor, which is evident because some systems with PCI-E were bested by systems using PCI-X.

The answer lies in the type of instructions used. Xeon is well-known for having a floating-point core that is faster than Opteron. Both have Streaming SIMD Extension (SSE2) units that are capable of operating in vector mode at a rate of two results per clock. Since Xeon has the faster clock, its core is perceived as faster. But in this case the benchmark does not use vector operations. Instead the vast majority of its operations are scalar floating-point operations. Opteron

issues scalar instructions on every clock, while Xeon issues them on every other clock. This gives Opteron an advantage of about 33%, which is what we see reflected in the benchmark performance.

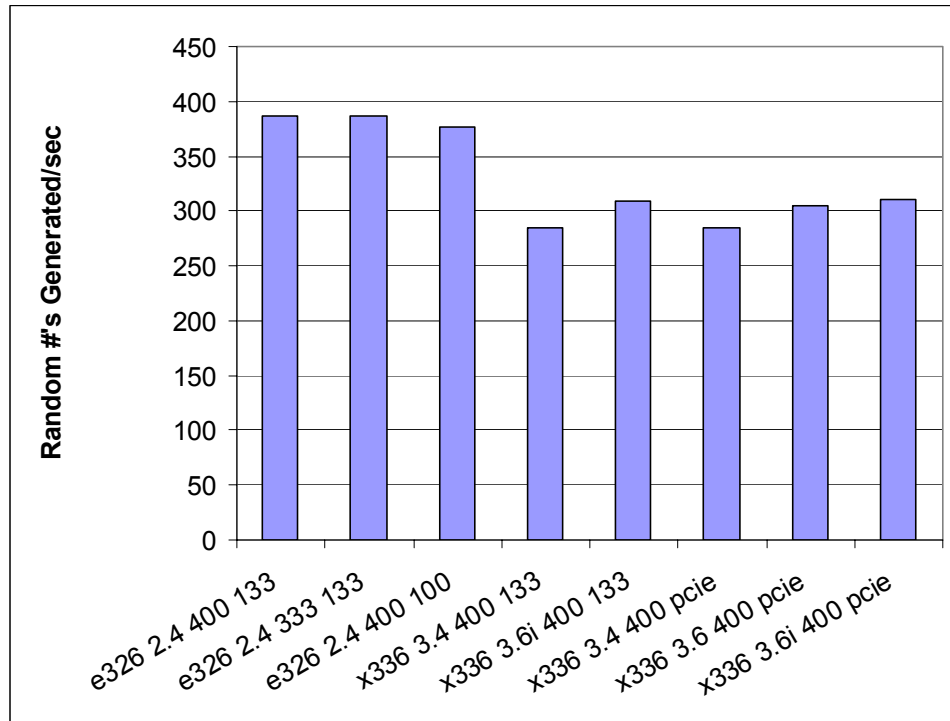


Figure 24. EP Performance

7.4 FT (Fourier Transform) Performance

More so than any of the other NAS Parallel Benchmarks, FT exercises the entire system. Benchmark performance, shown in Figure 25, is affected by memory bandwidth, cache size, interconnect performance and processor performance, in roughly that order. Because it is affected by so many subsystems, it is a well-balanced benchmark. No single subsystem is a bottleneck.

We can see from the figure that the fastest performance goes to the e326 with the fastest memory and the fastest interconnect. In fact, all of the e326 clusters outperformed all of the x336 clusters. This tells us that the most significant component of this benchmark is the memory bandwidth. An interesting observation is that the system with 333 MHz DDR memory and 133 MHz PCI-X performed at about the same level as the system with 400 MHz DDR memory and 100 MHz PCI-X. The interesting part is that 400 MHz DDR is faster than 333 MHz DDR by the same proportion, 33%, as 133 MHz PCI-X is faster than 100 MHz PCI-X.

The Xeon processor-based systems expose even more relationships. As with the Opteron processor-based systems, the clusters with faster networks outperform similarly configured clusters with slower networks. (The communication routine used in this benchmark is MPI_Alltoall.) Clusters with faster processors outperformed similarly configured clusters

that have slower processors, and clusters with larger caches outperformed similar clusters with smaller caches.

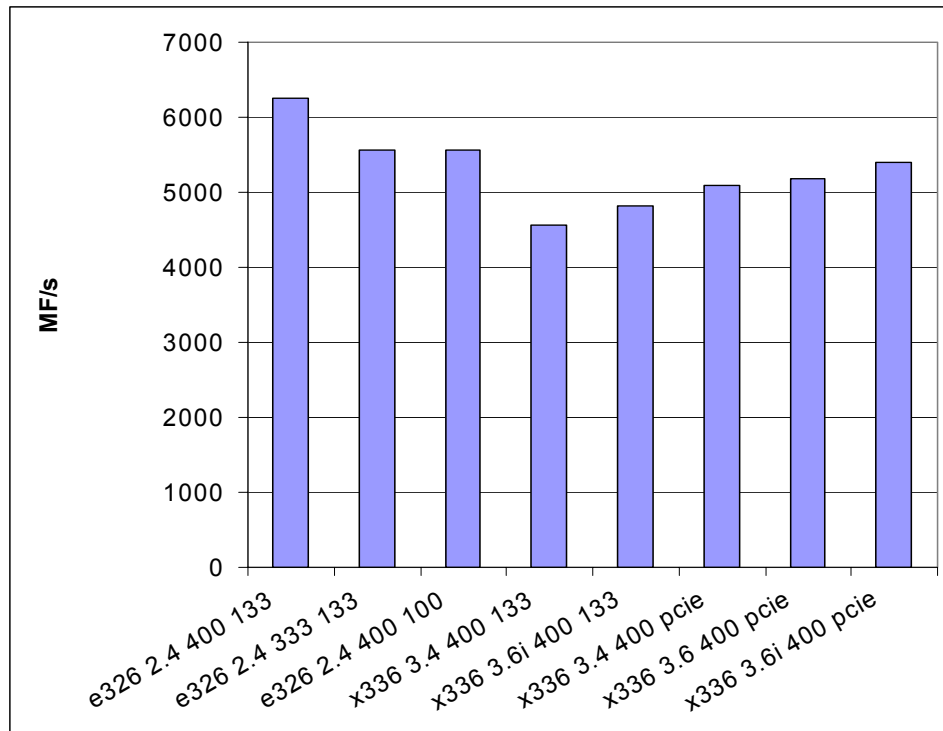


Figure 25. FT Performance

To quantify, a 33% increase in memory or network performance resulted in a 12% benchmark gain. Increasing cache from 1MB to 2MB increased performance by 4%.

7.5 IS (Integer Sort) Performance

The IS benchmark exercises a combination of interconnect, processor and memory performance, as Figure 26 shows. It is a well balanced integer benchmark.

For this benchmark it is clear that performance is affected by memory, processor and network performance. Increasing the memory speed on the e326 by 33% increases benchmark performance by 7%. Increasing the processor performance by 6% on the x336 increases benchmark performance by 3%. Network performance is harder because we don't know whether communication is dominated by small messages or large, half-duplex or full-duplex. But using the e326 as a gauge, increasing network performance by 33% increases benchmark performance by 16.7%. Using the x336, increasing the network performance by 100% increases benchmark performance by 33%. This tells us it is most sensitive to processor performance, then network performance, then to memory performance.

Someone might complain that if the IS benchmark is most sensitive to processor performance, why doesn't the Xeon lead? After all, it has the faster clock. Doesn't that make it a faster processor? Linpack results seem to suggest so!

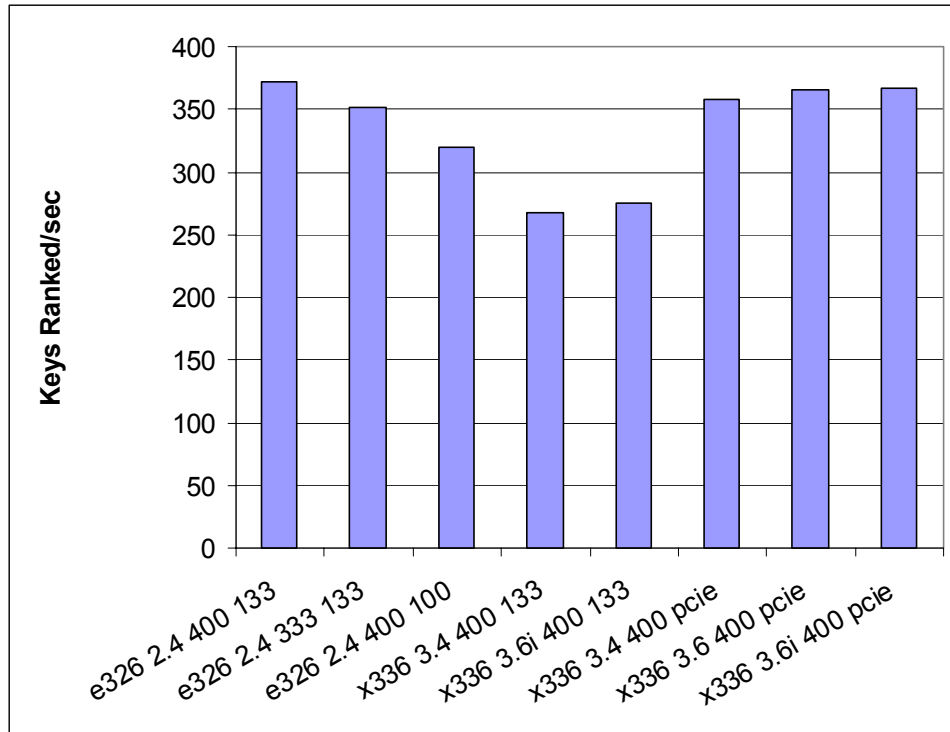


Figure 26. IS Performance

The answer lies in the fact that Xeon and Opteron processors have differences in their core architectures. Xeon processors have two simple integer units and a 31-stage pipeline, while Opteron processors have three units with only 12 stages. The net effect is that a 2.4 GHz Opteron is roughly the equal to a 3.6 GHz Xeon in core integer performance. So the fact that the best e326 closely approximates the best x336 performance supports the conclusion and doesn't refute it.

7.6 LU (LU Factorization) Performance

Although the BT and LU benchmarks use different techniques to solve CFD problems, they behave in similar ways. The data in Figure 27 looks remarkably similar to the performance data from the BT benchmark. The scales are even similar. The obvious implication is that if BT performs well on a given server, LU will also perform well on that server. It also suggests that BT and LU are redundant, that the inclusion of both in a benchmark suite does not add any more information than including only one of them. Looking forward, there are also strong similarities with MG and SP, so any conclusions about BT and LU also apply, to some degree, to them.

The statement about BT and LU being redundant is a little overstated because there are *some* differences in performance. For example, raising the L2 cache size from 1MB to 2MB does increase LU performance by 7%. Increasing the cache size raised BT performance only slightly, if at all. The sensitivity to memory performance is also a little less. Increasing memory bandwidth by 33% improves LU performance by 15%, whereas it improves BT performance by 20%.

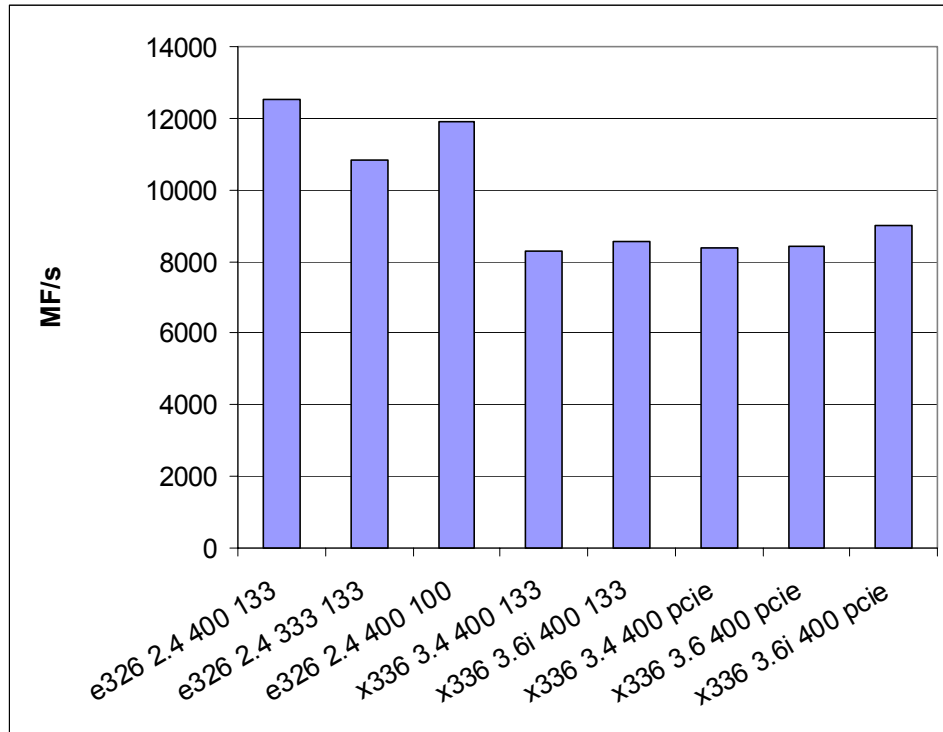


Figure 27. LU Performance

7.7 MG (Multi-Grid) Performance

Like BT and LU, MG performance is gated primarily by memory bandwidth. Figure 28 shows the performance. It also shows an unexpected behavior. The e326 cluster with 100 MHz PCI-X shows 22% better performance than a similarly configured cluster with 133 MHz PCI-X. This is the best performing configuration of all configurations we used.

Memory bandwidth scaling for MG is moderate. Increasing the memory from 333 MHz to 400 MHz, or 33%, increases the benchmark performance from 8,508 to 9,605, or about 13%. Cache scaling for 1MB L2 to 2MB L2 is very small, less than 3%.

The performance of the e326 cluster with 100 MHz PCI-X is most puzzling. We're not aware of anything that should cause this effect. The measurements for this cluster and the e326 cluster with 133 MHz PCI-X were done with exactly the same software stack, and even the same hardware. Only a jumper was changed in the chassis to switch the slot from 133 MHz to 100 MHz. The experiments have been double-checked to ensure they were not mislabeled in any way. The experiments were run 10 times to ensure it was not a statistical anomaly, and all experiments show consistent results. It is especially puzzling because communication performance, for example, using 133 MHz PCI-X in place of PCI-E on the x336, does not show *any* affect on performance.

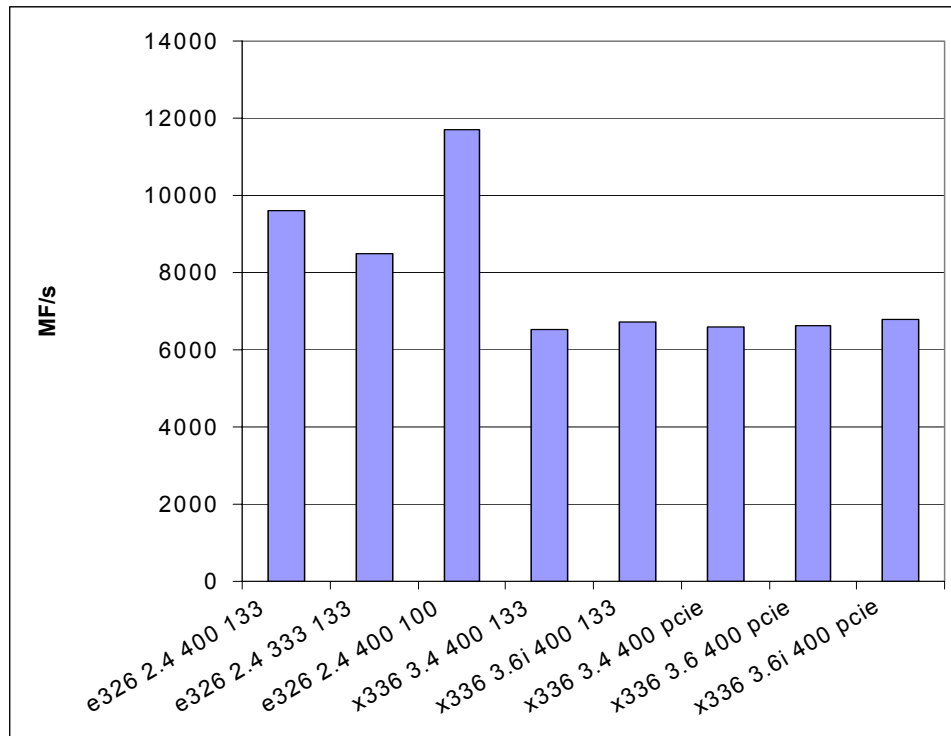


Figure 28. MG Performance

7.8 SP (Scalar Pentadiagonal) Performance

The final benchmark is SP. As Figure 29 shows and as mentioned in earlier sections, it has much the same characteristics as BT, LU and MG. Its performance is correlated very strongly with memory performance.

Increasing memory performance from 333 MHz DDR to 400 MHz DDR, about 33%, takes the benchmark performance from 5915 to 7405, or about 25%. This is nearly perfect scaling. In contrast, cache scaling from 1MB L2 to 2MB L2 is minimal, barely 2%. Interconnect and processor performance have almost no impact on benchmark performance, so there is effectively no scaling for them.

8. Conclusions

Voltaire InfiniBand has proven itself to be a fast interconnect. The PCI-Express performance is especially high. InfiniBand with both PCI-X and PCI-E achieved high efficiencies after considering the limitations of each slot in which it was used. We tested InfiniBand on eight separate configurations of an eight-node cluster, using the Pallas MPI Benchmarks and the NAS Parallel Benchmarks, and it performed well, except possibly on the x336 using 133 MHz PCI-X.

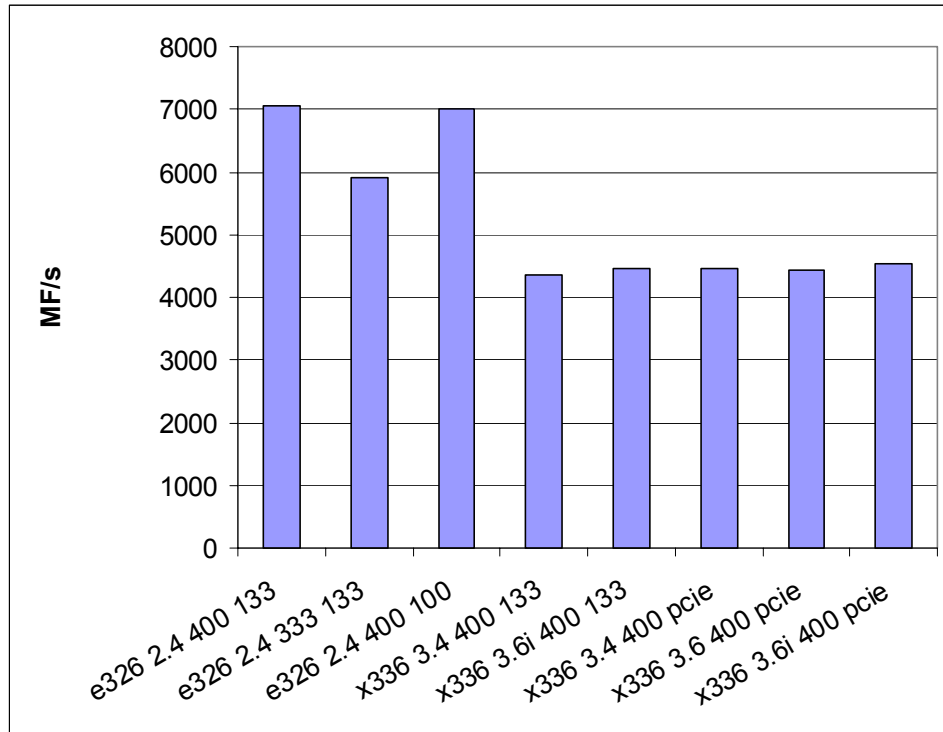


Figure 29. SP Performance

8.1 Pallas MPI Benchmarks

1. Using PCI-Express, Voltaire InfiniBand achieved latencies below 4 microseconds, 914 MB/s unidirectional throughput, and 1,735 MB/s bidirectional throughput.
2. On the e326 using 133 MHz PCI-X slots it achieved latencies below 5 microseconds and throughput of 739 MB/s (both uni- and bidirectional throughput).
3. Using PCI-Express, maximum throughput is achieved with message sizes of about 1 MB, and half of the maximum throughput is achieved with message sizes of about 8 KB.
4. Using 133 MHz PCI-X, maximum throughput is achieved with message sizes of about 500 KB, and half of that is achieved with message sizes of 2 KB.
5. Collective operations of all types showed stable performance without excessive congestion.
6. InfiniBand shows stable performance across most ranges of message sizes, but there is an expected dip in performance where a message must fill a complete packet and a few bytes of a second packet, just above 1500 bytes.
7. The network performance of the x336 using 133 MHz PCI-X was unexpectedly slower than the e326 with 100 MHz PCI-X in all but two of the experiments, although it outperformed even the e326 with 133 MHz PCI-X in the NAS Parallel Benchmark CG.

8.2 NAS Parallel Benchmarks

Only two of the eight NAS Parallel Benchmarks, CG and IS, are sensitive to network performance. CG is most sensitive, as demonstrated by the fact that any cluster with InfiniBand over PCI-E outperformed any cluster using InfiniBand over PCI-X, regardless of other

characteristics of the cluster. IS is also quite sensitive since increasing the InfiniBand network performance by some amount x increased the benchmark performance by about one half of x . Even so, IS performance is not dominated solely by network performance as CG is.

All other benchmarks depend on some other aspect of the system for performance, something unrelated to network performance. This may hold true for these tests because InfiniBand is very fast, and a cluster with a slower network might show greater sensitivity to network performance. Larger clusters or different data sets may also change the fundamental relationships between communication and performance, and increase their sensitivity to network performance.

To give a brief summary of all of the benchmarks:

1. Opteron processor-based clusters are faster on six of the eight benchmarks (BT, EP, FT, LU, MG and SP). Xeon processor-based clusters are faster on one (CG), and one benchmark (IS) shows no clear-cut winner.
2. BT, LU, MG and SP all have nearly identical performance characteristics. All are almost exclusively dependent upon memory performance, although LU and SP are affected slightly by cache size. A processor with high bandwidth to memory, such as Opteron, has a strong advantage on these benchmarks.
3. CG favors Xeon processor-based clusters because they support PCI-E with its high interconnect performance.
4. The EP benchmark is dominated by double-precision floating-point scalar operations. Once again, this is an area where Opteron processors excel. Opteron processors are capable of issuing a scalar operation with every clock, whereas Xeon processors are only able to issue scalar operations every other clock.
5. FT is a well-balanced floating-point benchmark in that it depends on all subsystems and not on only one or two to achieve good performance. FT performance is most sensitive to memory performance, then to interconnect performance, then processor core and L2 cache size.
6. IS is a well-balanced integer benchmark. Its performance depends on a mix of factors, including processor performance, interconnect performance and memory performance.

From these experiments it is clear that clusters based on the AMD Opteron processor have a significant advantage when running the NAS Parallel Benchmarks. They were only challenged by Xeon processor-based clusters in the CG and IS benchmarks. Future Opteron processor-based systems that support PCI-E and dual-core processors will likely be able to win on those two benchmarks as well.

9. Acknowledgements

The author would like to extend his appreciation to all those who helped make this paper happen. To start, I would like to thank the Voltaire team for their support and insight. Special thanks go to Tia Howell for her patience and persistence in providing outstanding technical support, and Yaron Haviv, who provided comments, technical insight, and graphics. I would also like to thank the IBM technical staff, especially Bill Wilmoth and Matt Eckl, for their significant contributions in helping with all of the many configurations and experiments that went into this paper.

10. References

- [1] William T. Futral, *InfiniBand Architecture: Development and Deployment*, Intel Press, 2001.
- [2] Tom Shanley and Joe Winkles, *InfiniBand Network Architecture*, Addison-Wesley, 2002.
- [3] Jack J. Dongarra, Piotr Luszczek, and Antoinet Petit, “Linpack Benchmark: Past, Present, and Future,” <http://www.cs.utk.edu/~luszczek/articles/hplpaper.pdf>.
- [4] J. J. Dongarra, “The Linpack benchmark: An explanation,” in A. J. van der Steen, editor, *Evaluating Supercomputers*, pages 1-21. Chapman and Hall, London, 1990.
- [5] STREAM: Sustainable Memory Bandwidth in High Performance Computers, www.cs.virginia.edu/stream/.
- [6] “Pallas MPI Benchmarks—PMB, Part MPI-1,” <http://www.pallas.com/pub/PALLAS/PMB/PMB-MPI1.pdf>.
- [7] David Bailey, *et al*, “The NAS Parallel Benchmarks,” <http://www.nas.nasa.gov/Research/Reports/Techreports/1994/PDF/RNR-94-007.pdf>, 1994.
- [8] J.J Dongarra, Cleve B. Moler, G. W. Stewart, *Linpack User's Guide*, Society for Industrial & Applied Mathematics, June 1979.
- [9] Linpack, www.netlib.org/linpack/index.html.
- [10] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, “LU Decomposition and Its Applications,” §2.3 in *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, 2nd ed. Cambridge, England: Cambridge University Press, pp. 34-42, 1992.
- [11] E. Anderson, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, Sorenson D., Zhaojun Bai, Christian H. Bischof, *Lapack User's Guide*, 3rd edition, Society for Industrial & Applied Mathematics, 2000.
- [12] Kazushige Goto, “High-Performance BLAS,” www.cs.utexas.edu/users/flame/goto/.
- [13] 173.applu, <http://www.spec.org/osg/cpu2000/CFP2000/173.applu/docs/173.applu.html>
- [14] E. Barszcz, R. Fatoohi, V. Venkatkrishnan and S. Weeratunga “Solution of Regular Sparse Triangular Systems on Vector and Distributed-Memory Multiprocessors”, Rept. No: RNR-93-007, NASA Ames Research Center, 1993. Also available at <http://www.nas.nasa.gov/Pubs/TechReports/ebarszcz/RNR-93-007/RNR-93-007.html>
- [15] Marc Snir, Steve Otto, Seven Huss-Lederman, David Walker, and Jack Dongarra, *MPI—The Complete Reference*, Vol. 1, 2nd Ed., MIT Press, 1996.
- [16] “MPICH—A Portable MPI Implementation,” <http://www-unix.mcs.anl.gov/mpi/mpich/>.
- [17] “LAM/MPI Parallel Computing,” <http://www.lam-mpi.org/>.
- [18] MPI over InfiniBand Project, <http://nowlab.cis.ohio-state.edu/projects/mpi-iba/>.
- [19] Douglas M. Pase and James Stephens, “Performance of Two-Way Opteron and Xeon Processor-Based Servers for Scientific and Technical Applications,” ftp://ftp.software.ibm.com/eserver/benchmarks/wp_server_performance_030705.pdf, IBM, March 2005. Also published under 2005 LCI Conference, http://www.linuxclustersinstitute.org/Linux-HPC-Revolution/Archive/PDF05/14-Pase_D.pdf.



© IBM Corporation 2005

IBM Systems and Technology Group

Department MX5A

Research Triangle Park NC 27709

Produced in the USA.

5-05

All rights reserved.

IBM, the IBM logo, the eServer logo, eServer and xSeries are trademarks or registered trademarks of IBM Corporation in the United States and/or other countries.

Intel and Xeon are trademarks or registered trademarks of Intel Corporation.

InfiniBand is a registered trademark of the InfiniBand Trade Association.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

AMD and Opteron are trademarks or registered trademarks of Advanced Micro Devices, Inc.

Other company, product, and service names may be trademarks or service marks of others.

IBM reserves the right to change specifications or other product information without notice. References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates. IBM PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This publication may contain links to third party sites that are not under the control of or maintained by IBM. Access to any such third party site is at the user's own risk and IBM is not responsible for the accuracy or reliability of any information, data, opinions, advice or statements made on these sites. IBM provides these links merely as a convenience and the inclusion of such links does not imply an endorsement.