Contention of Communications in Switched Networks with Applications to Parallel Sorting

REU Site: Interdisciplinary Program in High Performance Computing

Nil Mistry[1], Jordan Ramsey[2], Benjamin Wiley[3], and Jackie Yanchuck[4],
Graduate RA: Xuan Huang[5], Faculty Mentor: Matthias K. Gobbert[5],
Clients: Christopher Mineo[6] and David Mountain[6]

[1]Department of Mathematics and Statistics, University of Connecticut
[2]Department of Computer Science and Electrical Engineering, UMBC
[3]Department of Mathematics and Statistics, University of New Mexico
[4]Department of Mathematics, Seton Hill University
[5]Department of Mathematics and Statistics, UMBC
[6]Advanced Computing Systems Research Program

Technical Report HPCF–2013–13, `www.umbc.edu/hpcf` > Publications

Abstract

Contention of communications across a switched network that connects multiple compute nodes in a distributed-memory cluster may seriously degrade performance of parallel code. The InfiniBand network is the most popular interconnect for compute clusters. While one may correctly assume that increased resource contention leads to decreased application performance, alternate methods such as virtual channels and adaptive routing have obscured the point at which inter-job interference becomes a major issue. This contention is maximized when communicating large blocks of data among all parallel processes simultaneously. This communication pattern arises in many important algorithms such as parallel sorting. We use the cluster tara in the UMBC High Performance Computing Facility (HPCF) with a quad-data rate InfiniBand network which provides an opportunity to test the case if the capacity of a switched network is a limiting factor in algorithmic performance.

1 Introduction

Consider a standard InfiniBand network in which there are multiple parallel processes being run on various computation nodes freely connected to one another. Information transferred among nodes may stress communication across the network, both in relation to the size of the data being sent and the number of nodes being considered. As communication increases, contention along the system will stress the network due to the mass transfer of data among compute nodes. At significantly high levels of contention, the network eventually will fail to process inter-node communication. Studying network contention by varying the size of the data being sent along the network, our group has effectively studied the use of All-to-All communications in parallel jobs performance considering large blocks of data.

To accomplish sufficient inter-node stress within the network, our team implemented a parallel integer sorting function which transfers a user-defined number of n integers across p

processes. Hence, each node receives subsections of an array of data, sorted locally per group though unsorted globally, thereby allowing all delays in computation times to be the result of communication stress and not computational sorting time. In an effort to increase network contention, the array of integers were converted from data type `int` to `double`, which requires more memory to send across nodes. Memory selection was also considered such that integer arrays too large to store on nodes within the network would not be considered in our case as this leads to memory leaks.

Using All-to-All commands, each individual node sent its received integer array data across all other nodes in the network, maximizing inter-node communication stress. Our results state that, for constant global memory, as the number of processes increase, speed improves as the network contention decreases under All-to-All communication. Alternately, for varying local memory, as the number of processes increase, speed deteriorates as the network contention increases under All-to-All communication.

2 Background

2.1 Computational Environment and InfiniBand Interconnect

The studies were performed on the cluster tara in the UMBC High Performance Computing Facility (HPCF). All details of the cluster tara and in particular about its InfiniBand interconnect are posted on the webpage `www.umbc.edu/hpcf`. Various performance studies using tara are available as technical reports, for instance [2] that compares performance by two implementations of MPI. Following [2], we use MVAPICH2.

The cluster tara has 86 nodes, comprising 82 compute nodes, 2 develop nodes, 1 user node, and 1 management node. Each node has two quad-core Intel Nehalem X5550 processors (2.66 GHz, 8192kB cache) and 24 GB of local memory. All components of tara are connected by a quad-data rate InfiniBand interconnect.

InfiniBand is a serial connection that allows for high-speed data transfers from computers to input/output devices [3]. It is a switched fabric communication link, meaning that it connects the nodes to each other via switches. In computer networks, switches receive data sent from one device and direct the data to only the device(s) which were meant to receive the data [3]. This allows for more secure and potentially faster data transfers between multiple devices. Using the InfiniBand communication network, there is very low latency (1.2 microseconds to transfer a message between two nodes), and wide bandwidth up to 3.5 GB (28 Gb) per second.

It is intuitive to hypothesize that as the number of processes on which a job is run increases, the communication between processes will become slower and may bottleneck because more processes need to communicate with each other than when the number of processes is small. However, many times, commercial manufacturers attempt to avoid this occurrence by using methods such as virtual channels and adaptive routing. Adaptive routing, as apposed to merely routing, allows nodes to reroute the path that data is sent based on network fluctuations, such as congestion at one node. When a problem is encountered

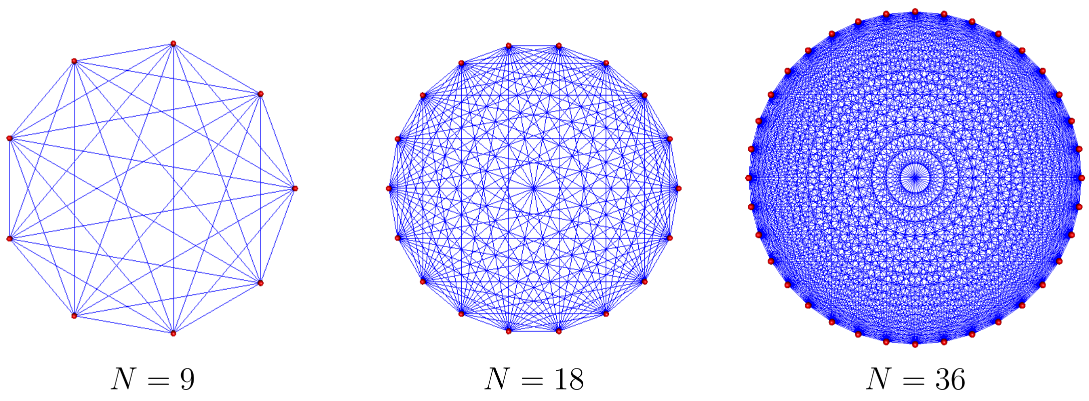

Figure 2.1: Network schematics for All-to-All communication between $9, 18, 36$ nodes, respectively

while transferring data, information is sent to the appropriate nodes, and new paths to send data that avoid problem areas are created [3]. Virtual channels were created in order to alleviate the deadlock issue, and also decrease network latency and throughput. Though these methods are commonly used in parallel computing technology to solve many communication issues that arise, their effects on performance have not been rigorously studied. Therefore, it is difficult to determine when inter-job communication will become a performance issue. Our experiments attempted to study this issue using a variety of different network setups. In order to study the effects of inter-job communication on job performance, our team implemented an sort algorithm which requires communication between all nodes. As the set of numbers to be sorted increases, more communications will be required.

2.2 All-to-All Communications

An All-to-All communication simultaneously sends and receives data between all parallel processes in one call. Since is it eventually not possible to have physical cable connections between all possible pairs of ports in the InfiniBand switch and its leaf modules, All-to-All commands necessarily lead to contention between all required pairwise communications. The network schematics in Figure 2.1 gives an impression of how many cables would be needed to connect $N = 9, 18, 36$ nodes, respectively. An All-to-All communication command sends the j^{th} block of its input array from Process i to Process j and receives it into the i^{th} block of the output array on Process j. MPI has two All-to-All communication commands: `MPI_Alltoall` and `MPI_Alltoallv`. The former command sends the same amount of data between all processes, while the latter one can send variable (hence the letter "v" at the end of the command name) amounts of data between all processes [1]. To test the InfiniBand network, we will maximize the contention by communicating the largest block sizes possible. Thus, also the variable version `MPI_Alltoallv` will be programmed to send the same amount of data between all processes, since that maximizes contention between messages.

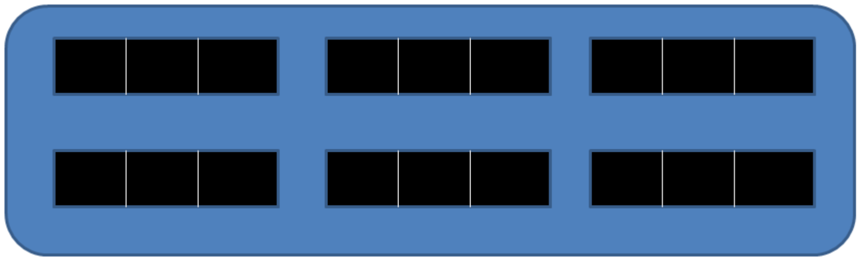

Figure 3.1: Leaf Module with Eighteen Nodes

3 Numerical Methods

3.1 Leaf modules

The cluster tara has 6 leaf modules, of which two currently have complete sets of 18 compute nodes attached to them. Specifically, one leaf module connects the nodes n37 through n54, while another leaf module connects nodes n55 through n72. We can control the choice of leaf module by explicitly requesting nodes for our jobs by name. The remaining leaf modules contain other nodes that are not part of the partition of compute nodes (such as the develop nodes or components of the storage system) or have a failed node among its connections. Therefore, in this study, our team focuses on how contention is effected both within a leaf module and contention over two leaf modules. Considering this network contention provides insight into whether parallel algorithms that send large blocks of data via All-to-All communications result in contention first over two leaf modules or whether there is contention using nodes located within just one leaf module. More importantly, our conclusions answer the question regarding whether implementation of parallel code requiring All-to-All communications of large data seriously degrades performance.

Nodes are distributed evenly between two rows on the leaf module; that is, nine nodes are located in the first row and nine nodes are located in the second row. Each of the nine nodes are separated in groups of three pairs of three nodes, as shown in the schematic picture in Figure 3.1. Effectively studying contention on the InfiniBand system relies on analyzing performance on one node for control purposes. Furthermore, our team runs several tests on the InfiniBand network by requesting specific nodes, starting with three nodes, then testing nine nodes or one row in the leaf module. Finally, this process is extended to the whole leaf module or 18 nodes, and then across two leaf modules or 36 nodes. This setup allows us to see if contention problems can be linked to communication within the leaf module or communication between leaf modules of parallel code.

3.2 Sorting Function

In order to effectively stress inter-job communication, our team implemented a sorting function which transfers data of type `struct` across specified nodes within the InfiniBand network utilizing MPI commands, mainly `MPI_AlltoAll` and `MPI_AlltoAllv`. Since each node on the tara cluster contains 24 GB of memory, integers were created in the form of `int` and `double` with the hypothesis that sending large amounts of double-precision floating point numbers (i.e., doubles) will force `MPI_AlltoAllv` to test the contention of the InfiniBand network. In this algorithm, doubles are stored within an array, of MPI data type structure, thereby causing increased contention along the network during communication. This test case for contention of communication emulates data structures that is common is many parallel programs. Mainly, a global array of n vectors, each comprising m double-precision numbers, is split onto the p parallel processes. This results in local arrays of $l_n = n/p$ vectors, each comprising m double-precision numbers. In an algorithmic context, the goal may be to sort all m-vectors in the global array. More specifically, two vectors, *sorted* and *unsorted*, are defined such that the design of the size of the integers are determined by

$$n = 2^{\nu} \cdot (p \cdot N \cdot l_{m_1} \cdot l_{m_2})^2 \tag{3.1}$$

where p is the number of processors on one node, ν controls the size of the integer, N is the number of nodes, l_{m_1} is leaf module 1, and l_{m_2} is leaf module 2. By defining ν as a small integer, in this case $\nu = 1$, one obtains an integer size

$$n = 2 \cdot (18 \cdot 8 \cdot 3 \cdot 2)^2 = 1,492,992.$$

Local integers on each process, $l_n = \frac{n}{p}$, are defined as the total size of the integers, n, divided by the number of processes, p. Our team created two variables, $send_{count}$ and $send_{displacement}$, to keep track of how many integers are being sent to each processor. It is important to note that the unsorted vector is assumed to contain integers in leaf module 1 defined as counting from 1 up to n; that is, for all processes p, the unsorted vector $\vec{l}_{u_{\texttt{int}}}$ is defined such that

$$\vec{l}_{u_{\texttt{int}}} = \begin{bmatrix} j_0 \\ j_1 \\ \vdots \\ j_{p-1} \end{bmatrix}, \ \forall j_i \in \{1, \ldots, l_n\}, \tag{3.2}$$

where the set $\{1, \ldots, l_n\}$ only contains data of type `int`, and $\vec{l}_{u_{\texttt{int}}}$ has dimensions $l_n \times 1$. Our team developed an algorithm which defines

$$send_{count} = send_{count}\left(\frac{\vec{l}_{u_{\texttt{int}}}}{l_n}\right) + 1, \tag{3.3}$$

if the amount of integers unsorted on the process is less than the local number of the unsorted integers.

Similarly, our team constructed an additional unsorted vector, $\vec{l}_{u_{\texttt{double}}}$, per process such that

$$\vec{l}_{u_{\texttt{double}}} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{p-1} \end{bmatrix}, \ \forall v_i \in \{m, \ldots, l_n m\}, \tag{3.4}$$

where the set $\{m, \ldots, l_n m\}$ only contains data of type `double`, and $\vec{l}_{u_{\texttt{double}}}$ has dimensions $l_n m \times 1$. One should note that the double-precision floating point definition of $\vec{l}_{u_{\texttt{double}}}$ is unique and separate from the integer characterization of (3.3).

To establish this definition, our team declared `MPI_Datatype` as `MPI_DOUBLE` which enables communication between the network of nm doubles being sent from process p_i, for some $i \in \{1, \ldots, p\}$, to all processes p_j, $\forall j \in \{1, \ldots, p\}$, as shown by $l_{n_{np}} = \frac{l_n}{p} = \frac{n}{p^2}$.

As above, a similar algorithm

$$send_{count} = send_{count} \left(\frac{\vec{l}_{u_{\texttt{double}}}}{l_n} \right) + 1, \tag{3.5}$$

is defined if the amount of unsorted doubles on the process is less than the local number of the unsorted doubles.

The `MPI_Datatype` call sends information from the $send_{count}$ and $recv_{count}$ vectors to each of the processes. The purpose of this command is to send the same amount of data, namely the $send_{count}$ and $recv_{count}$, to each of the processes. Alternately, the `MPI_AlltoAllv` call sends specific information regarding which process needs which integers of type `int`, since this is a small size data, the run times will be hard to compare. Thus, `MPI_AlltoAllv` is used again, this time to send the structure of doubles.

3.3 Experiment Design

In order to effectively study network contention, our team designed an experiment to maximize communication by having no local sorting before and after the double-precision floating point numbers are distributed across processes. Implementing the following equation,

$$\vec{l}_{u_{\texttt{int}}}[i] = 1 + l_{n_{np}} \cdot id + l_n \cdot \frac{i}{l_{n_{np}}} + i \mod l_{n_{np}}, \tag{3.6}$$

creates an unsorted vector $\vec{l}_{u_{\texttt{int}}}$ that is size $l_n \times 1$, where $l_n = \frac{n}{p}$, $l_{n_{np}} = \frac{l_n}{p} = \frac{n}{p^2}$, and i is an index running from 0 through l_n (See section 3.2 for details). Equation 3.7 generates integers of size l_n on each process p.

Constructing $\vec{l}_{u_{\texttt{double}}}$,

$$\vec{l}_{u_{\texttt{double}}}[im + j] = \vec{l}_{u_{\texttt{int}}}[i] + 0.0001j, \tag{3.7}$$

the vector $\vec{l}_{u_{\texttt{double}}}$ stores $\vec{l}_{u_{\texttt{int}}}[i]$ at the current step i added by $0.0001j$, for all $i \in \{0, 1, 2, \ldots, l_n\}$

and $j \in \{0, 1, 2, \ldots, p\}$. Accordingly, each process p of size l_n contains m double precision floating point numbers.

The algorithmic scheme in (3.8) can be generalized as displayed in the following matrix,

$$Unsorted = \begin{pmatrix} 1\ ,2\ ,3 & 13, 14, 15 & 25, 26, 27 & 37, 38, 39 \\ 4\ ,5\ ,6 & 16, 17, 18 & 28, 29, 30 & 40, 41, 42 \\ 7\ ,8\ ,9 & 19, 20, 21 & 31, 32, 33 & 43, 44, 45 \\ 10, 11, 12 & 22, 23, 24 & 34, 35, 36 & 46, 47, 48 \end{pmatrix}.$$

which removes any local sorting when $N = 4$, $l_n = 12$, and $n = 48$. The rows in this matrix represent those processes which contain the specific doubles, which are already locally sorted from equations (3.7) and (3.8) above.

3.4 Memory Selection

In order to stress communication, the two vectors, *sorted* and *unsorted* (See section 3.6), are sent between nodes on a network. Since each node on the tara cluster contains 24 GB of memory, total memory must be less than 24 GB per node. The characterization of our sorting algorithm requires proper intake so that memory may be distributed evenly on every utilized node. Controlling the size of m allows the maximum amount of memory to be transferred onto each node.

Hence, the magnitude of m is easily controlled by differing either the number of nodes N on the network, or changing the total number of integers n being communicated along the network. Controlling the size of the integer n such that m is defined as an integer value enables the two vectors, *sorted* and *unsorted*, to be passed through the 24 GB memory on tara. Therefore, each of the vectors must be less than 10 GB to insure that memory leakage does not occur. Also, since contention arises due to the simultaneous nature of the All-to-All pairwise communications, we maximize contention by designing the test case to have all blocks to be communicated to be of the same maximum size which is $\frac{l_n}{p}$. In Table 3.1, we generalize our memory calculations to use the maximum possible number of 8 parallel processes on each compute node, which maximizes contention on each node for the All-to-All communications among its local processes and contention when all local processes access the InfiniBand cable at the same time.

Table 3.1 provides equations for node computations regarding a global unsorted vector, where the size of the integers n, contained in the vector, is constant at $n = 2 \cdot (8 \cdot 18 \cdot 2 \cdot 3)^2 = 1,492,992$. The global vector is then divided into p sub-vectors of length $\vec{l_n}$ such that the size of the integers contained within each locally sorted vector is constant per number of processes p, equal to $\frac{1,492,992}{p}$. In addition, the total memory being received per process is equal to $m \cdot \frac{n}{p}$. Finally, communication between process p_i, where $i \in \{0, \ldots, p\}$, to all processes p_j, $\forall j \in \{0, \ldots, p\}$, is equal in magnitude to to the number of integers on that particular process, scattered to all other processes; that is, $\frac{l_n}{p} = \frac{n}{p^2}$. Also, the amount of memory being sent across the network per process is equal to $m \cdot \frac{n}{p^2}$. One should note that memory is allowed to vary as the number of processes p increases, where doubles are being

Table 3.1: Equations for memory predictions.

Nodes N	1	3	9	18	36
Processes p	8	24	72	144	288
Dimension m	m	m	m	m	m
Length n of global array of m-vectors and their size in elements:					
Length n	n	n	n	n	n
Size	$m\,n$	$m\,n$	$m\,n$	$m\,n$	$m\,n$
Length $l_n = n/p$ of local arrays of m-vectors and their size in elements:					
Length l_n	$\frac{n}{p}$	$\frac{n}{p}$	$\frac{n}{p}$	$\frac{n}{p}$	$\frac{n}{p}$
Size	$m\,\frac{n}{p}$	$m\,\frac{n}{p}$	$m\,\frac{n}{p}$	$m\,\frac{n}{p}$	$m\,\frac{n}{p}$
Length l_n/p of block size of m-vectors in All-to-All and their size in elements:					
Length l_n/p	$\frac{n}{p^2}$	$\frac{n}{p^2}$	$\frac{n}{p^2}$	$\frac{n}{p^2}$	$\frac{n}{p^2}$
Size	$m\,\frac{n}{p^2}$	$m\,\frac{n}{p^2}$	$m\,\frac{n}{p^2}$	$m\,\frac{n}{p^2}$	$m\,\frac{n}{p^2}$

sent from process p_i to all other processes p_j decreases by a factor of p. This is because the number of parallel MPI processes $p = 8N$ in our tests are $p = 8, 24, 72, 144, 288$. The final values were then converted to megabytes (MB) for easier interpretation.

4 Results

4.1 Design of Experiment with Constant Global Memory

To effectively test the contention of the InfiniBand network, our team conducted a performance study with a constant global memory value to see how the InfiniBand handles the All-to-All commands. This was done by having constant $m = 512$ and the size of the total integer $n = 1,492,992$ being shuffled around the InfiniBand. Notably, the local memory of l_n decreases as the p increases and as N increases. This is amplified when we wish to transmit memory from $l_{n_{np}}$ namely p_i to p_j and decreases by another factor of p. The following is a memory distribution table where m is constant at $m = 512$.

To ensure memory leak is not an issue the total size of the global n is kept constant at $1,492,992$ in the experiments, so that $l_n = \frac{n}{p}$ and $\frac{l_n}{p} = \frac{n}{p^2}$ are integers for all possible p under consideration. Furthermore, the number of double-precision numbers communicated between pairs of processes in our experiments is the block size in the $All - to - All$ communications given by $m\frac{l_n}{p}$. As p increases, each process holds $\frac{n}{p}$ n and thus the memory decreases locally at $\frac{5832}{p}$. Finally, the n being sent from p_i to p_j decreases by $\frac{n}{p^2}$ and the memory decreases to $\frac{5832}{p^2}$ as this shows the communication between processes. The following results were obtained,

As expected, run times decreased as N increased. Interestingly, `All-to-Allv(int)}` increased as N increased and `MPI_Alltoallv(double)` decreased as N increased. From this we can conclude that our `MPI_Alltoallv(double)` is efficiently handled over the InfiniBand

Table 4.1: Constant global memory for $m = 512$: predicted memory usage for one array.

Nodes N	1	3	9	18	36
Processes p	8	24	72	144	288
$m = 512$	512	512	512	512	512
Length n of global array of m-vectors and their memory in GB:					
Length n	1,492,992	1,492,992	1,492,992	1,492,992	1,492,992
Memory	6 GB	6 GB	6 GB	6 GB	6 GB
Length $l_n = n/p$ of local arrays of m-vectors and their memory in MB:					
Length l_n	186,624	62,208	20,736	10,368	5,184
Memory	729 MB	243 MB	81 MB	41 MB	20 MB
Length l_n/p of block size of m-vectors in All-to-All and their memory in kB:					
Length l_n/p	23,328	2,592	288	72	18
Memory	93,312 kB	10,368 kB	1,152 kB	288 kB	72 kB

Table 4.2: Constant global memory for $m = 512$: wall clock time in seconds.

Nodes N	1	3	9	18	36
Processes p	8	24	72	144	288
$m = 512$	1.14	0.57	0.25	0.15	0.11

interconnect. Also, we obtain no bottlenecks with a constant global memory.

4.2 Design of Experiment with Constant Local Memory

From our result in 4.1 we are only interested in `MPI_Alltoallv(double)` as this is handled efficiently with InfiniBand. In order to keep the block size in the All-to-All communications as large as possible, the vector length m is designed to increase with increasing $p = 8N$ in the tests reported below.

With varying $m * N$, the amount of total memory increases by a factor of N. Thus, when this is distributed to l_n, the memory is held constant regardless of the N we choose to use. The following table shows run time as we vary mn,

This increase is limited by the fact that 8 local arrays must fit in the local memory of each node, since we use 8 processes per node. Specifically, the $m = 512$ N represents a choice which comfortably fits in the local memory, as shown in Figure 4.1. With the local memory held constant, the run times steadily increase as we increase mn with increasing N. Thus the contention on the network is maximized using our parallel integer algorithm of doubles.

Table 4.3: Constant local memory for $m = 512\,N$: predicted memory usage for one array.

Nodes N	1	3	9	18	36
Processes p	8	24	72	144	288
$m = 512\,N$	512	1,536	4,608	9,216	18,432
Length n of global array of m-vectors and their memory in GB:					
Length n	1,492,992	1,492,992	1,492,992	1,492,992	1,492,992
Memory	6 GB	17 GB	51 GB	103 GB	205 GB
Length $l_n = n/p$ of local arrays of m-vectors and their memory in MB:					
Length l_n	186,624	62,208	20,736	10,368	5,184
Memory	729 MB	729 MB	729 MB	729 MB	729 MB
Length l_n/p of block size of m-vectors in All-to-All and their memory in kB:					
Length l_n/p	23,328	2,592	288	72	18
Memory	93,312 kB	31,104 kB	10,368 kB	5,184 kB	2,592 kB

Table 4.4: Remark 1: performance close to local memory limit for All-to-All communication of $m\,(n/p^2)$ numbers. Wall clock time in seconds. The notation ERR indicates a memory error.

Nodes N	1	3	9	18	36
Processes p	8	24	72	144	288
$m = 512\,N$	0.60	1.64	2.09	2.28	2.30
$m = 800\,N$	1.79	3.05	3.73	5.01	6.73
$m = 810\,N$	1.80	2.83	3.30	5.54	ERR
$m = 1024\,N$	85.00	170.62	ERR	ERR	ERR

5 Conclusions

With local memory constant and contention on the network maximized, the run times grow with the number of processes. We can conclude that this test case creates stress on the InfiniBand network and that its performance will limit the scalability of parallel algorithms that use All-to-All communications. Furthermore, for cases with larger memory requirement, we encounter excessive run times and eventually memory errors as indicated by the notation

Table 4.5: Remark 2: relative performance of All-to-All commands for constant global memory with $m = 512$. Wall clock time in seconds.

Nodes N	1	3	9	18	36
Processes p	8	24	72	144	288
A. `MPI_Alltoall(int)`	<0.01	<0.01	<0.01	<0.01	<0.01
B. `MPI_Alltoallv(int)`	<0.01	0.01	0.10	0.32	0.59
C. `MPI_Alltoallv(double)`	1.14	0.57	0.25	0.15	0.11

ERR in Table 4.4

Acknowledgments

These results were obtained as part of the REU Site: Interdisciplinary Program in High Performance Computing (`www.umbc.edu/hpcreu`) in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County (UMBC) in Summer 2013. This program is funded jointly by the National Science Foundation and the National Security Agency (NSF grant no. DMS–1156976), with additional support from UMBC, the Department of Mathematics and Statistics, the Center for Interdisciplinary Research and Consulting (CIRC), and the UMBC High Performance Computing Facility (HPCF). HPCF (`www.umbc.edu/hpcf`) is supported by the National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from UMBC. Co-author Jordan Ramsey was supported, in part, by the UMBC National Security Agency (NSA) Scholars Program though a contract with the NSA. Graduate RA Xuan Huang was supported by UMBC as HPCF RA.

References

[1] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

[2] Andrew M. Raim and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the cluster tara. Technical Report HPCF–2010–2, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.

[3] Curt M. White. *Data Communications and Computer Networks: A Business User's Approach.* Course Technology, 2013.