

Throughput Studies on an InfiniBand Interconnect via All-to-All Communications

Nil Mistry
Department of Mathematics
and Statistics, University of
Connecticut
nil.mistry@uconn.edu

Jackie Yanchuck
Department of Mathematics,
Seton Hill University
yan6374@setonhill.edu

Jordan Ramsey
Department of Computer
Science and Electrical
Engineering, UMBC
jramsey3@umbc.edu

Xuan Huang
Department of Mathematics
and Statistics, UMBC
hu6@umbc.edu

Benjamin Wiley
Department of Mathematics
and Statistics, University of
New Mexico
bwiley01@unm.edu

Matthias K. Gobbert
Department of Mathematics
and Statistics, UMBC
gobbert@umbc.edu

ABSTRACT

Distributed-memory clusters are the most important type of parallel computer today, and they dominate the TOP500 list. The InfiniBand interconnect is the most popular network for distributed-memory compute clusters. Contention of communications across a switched network that connects multiple compute nodes in a distributed-memory cluster may seriously degrade performance of parallel code. 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. The cluster tara in the UMBC High Performance Computing Facility (HPCF) with a quad-data rate InfiniBand interconnect provides an opportunity to test if the capacity of a switched network can become a limiting factor in algorithmic performance. We find that we can design a test case of a problem involving increasing usage of memory that does not scale any more on the InfiniBand interconnect, thus becoming a limiting factor for parallel scalability. However, for the case of stable memory usage of the problem, the InfiniBand communications get faster and will not inhibit parallel scalability. The tests in this paper are designed to involve only basic MPI commands for wide reproducibility, and the paper provides the detailed motivation of the design of the memory usage needed for the tests.

Author Keywords

InfiniBand interconnect, All-to-All communications, network contention, scalability studies, MPI

ACM Classification Keywords

I.6.1 SIMULATION AND MODELING (e.g. Model Development). : Performance

INTRODUCTION

The TOP500 list at www.top500.org of the world's most powerful supercomputers has been dominated by distributed-memory clusters for many years by now.

Distributed-memory clusters require a network for communication among all nodes. The high-performance InfiniBand interconnect is currently the most popular network to communicate among the parallel processes on several nodes in distributed-memory compute clusters.

Information transferred among nodes may stress communication across the InfiniBand network, both in relation to the size of the data being sent and the number of nodes being considered. As communication increases, contention among the parallel processes will stress the network due to the massive transfer of data among compute nodes. At extremely high levels of contention, the network is expected to eventually fail to process inter-node communication efficiently. This work studies this effect by creating the maximum possible contention by simultaneous communication among all processes, created by All-to-All communication commands. The tests in this paper are designed to involve only basic MPI commands for wide reproducibility, and the paper provides the detailed motivation of the design of the memory usage needed.

To accomplish sufficient inter-node stress within the network in an algorithmically realistic way, we implemented a parallel sorting function which transfers the local portion of a user-defined number of n pieces of data among the p processes. Before the communications, each node holds a portion of an array of data that is sorted locally, but may contain portions that should reside on other processes. Using All-to-All commands, each individual node sends the appropriate portions of data to all other nodes in the network simultaneously, thus maximizing inter-node communication stress. Our results show that, for constant *global* memory, as the number of processes increase, the runtime in fact improves as the network contention decreases under All-to-All communication. Alternately, as global memory usage is increased maximally, that is, for constant *local* memory, as the number of processes increase, the runtime deteriorates as the network contention increases under All-to-All communication. This test demonstrates that stress on the InfiniBand network can be created that will eventually

limit the scalability of parallel algorithms that use All-to-All communications as building blocks. The situation of constant local memory occurs in weak scalability studies, while the situation of constant global memory occurs in strong scalability studies. Therefore, both are relevant in practice, and the conclusion whether the InfiniBand interconnect can become a limiting factor to parallel scalability will depend on the algorithmic structure of the code.

The sorting example used as prototypical basis for our experiments is inspired by [8, Chapter 10] that uses it to introduce the `MPI_Alltoall` and `MPI_Alltoallv` commands. Other works that test the usage of these commands over InfiniBand are [1, 3–6, 11]. Experiments designed to stress the network data transfer over InfiniBand are described in [2, 13], and [10] reports using FFT and RandomAccess to stress the bandwidth and latency of the Infiniband.

BACKGROUND

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 [9] that compares performance by two implementations of MPI. Following [9], we use the MPI implementation 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 connection that allows for high-speed data transfers from computers to input/output devices [12]. 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 is (are) meant to receive the data [12]. 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 μ s 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 parallel 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 while transferring data, information is sent to the appropriate nodes, and new paths to send data that avoid problem areas are created [12]. Virtual channels were created in order to alleviate the issue of deadlock, 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 study this issue. In order to study the effects of inter-job communication on job performance, our team implemented a sorting algorithm which requires communication between all parallel processes.

Leaf Modules

Each leaf module in the InfiniBand switch in the cluster has 18 ports. Two leaf modules currently have complete sets of 18 compute nodes attached to them. 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 defective node among its connections. We can control the choice of leaf module by explicitly requesting nodes for our jobs by name. Therefore, in this study, our team focuses on how contention is effected both within one 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.

The 18 ports in one leaf module are arranged evenly in two rows of ports; that is, nine ports are located in the first row and nine nodes are located in a second row. Each of the nine ports are separated in three groups of three ports. Our team studied contention by running several tests by requesting specific nodes, starting with three nodes that form one group on the leaf module, then testing nine nodes or one row in the leaf module, and finally extending this process to the whole leaf module with 18 nodes, and then across two leaf modules with 36 nodes. By keeping the node assignments as tightly together as possible in this way, we give the InfiniBand interconnect the best opportunity to overcome contention, since the more local communications can profit from the most complete set of interconnection.

METHODOLOGY

All-to-All Communications

An All-to-All communication simultaneously sends and receives data between all parallel processes in one call.

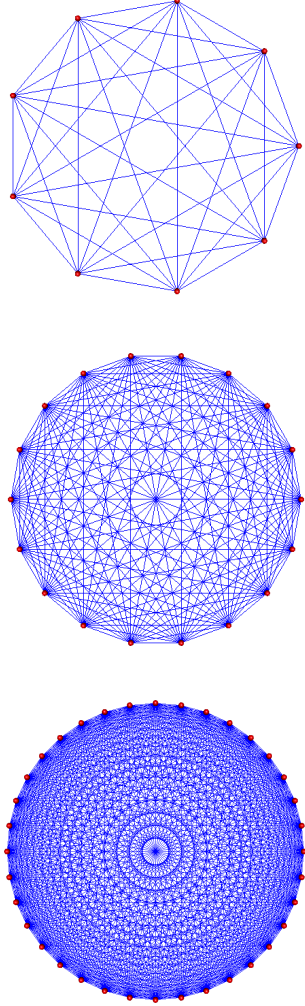


Figure 1. Network schematics for All-to-All communication between $N = 9, 18, 36$ nodes.

Since it is 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 1 give a visual 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 [8]. To test the InfiniBand network, we will maximize the contention by communicating the largest block sizes possible. Thus, in our studies, also the variable version `MPI_Alltoallv` will send (by choosing an appropriate example data set) the

same amount of data between all processes, since that maximizes contention between messages.

Experimental Design

In order to effectively stress inter-job communication, our team implemented a sorting function which transfers data among all nodes within the InfiniBand network utilizing the MPI commands `MPI_Alltoall` and `MPI_Alltoallv`. The idea of the algorithm is inspired by [8, Chapter 10] that uses a similar example to introduce these MPI commands. The data structure is given by n numbers, which are distributed onto the p parallel processes. Only local arrays of length $l_n := n/p$ are stored, never a global array of length n . Only the minimum number of arrays are used in the algorithm, namely one vector *unsorted* that holds the unsorted data originally and one vector *sorted* that holds the sorted data at the end of the algorithm. These two vectors have length l_n on each parallel MPI process. In [8], the algorithm has four steps: (i) The data in *unsorted* is sorted locally on each process (by any serial method of choice); while the numbers in *unsorted* are now sorted, they may contain components that need to be sent to the other process, thus creating the need for All-to-All communications. (ii) An `MPI_Alltoall` call communicates a single integer among all process pairs that informs the process pairs, how many pieces of data need to be sent and received among them in the next step. (iii) An `MPI_Alltoallv` call communicates the appropriate portions of the local *unsorted* vector on each process to the appropriate portions of the local *sorted* vector on each process. (iv) The numbers in the received *sorted* vector then still need local sorting (by any serial method) to obtain the final result of the algorithm, in which the *sorted* vectors — if they were concatenated from all processes — are globally sorted.

To focus entirely on the effect of the communications on the timings, we choose a sample dataset, in which neither of the local sort algorithms in steps (i) or (iv) above are needed. Moreover, since the goal is to stress the network by having as much simultaneous parallel communication as possible, we design the dataset in the initial *unsorted* vector to have an equal number of components that need to be sent to all other processes. That is concretely, out of the $l_n = n/p$ numbers in *unsorted* on one parallel process, the same block length of l_n/p components needs to be sent to each of the p processes.

The idea is best understood by a concrete example of $n = 48$ numbers, given as numbers $1, 2, \dots, 48$, distributed to $p = 4$ processes, with ID numbers $0, 1, 2, 3$ in MPI counting, displayed in the matrix *unsorted* =

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

The $p = 4$ rows in this matrix list the $l_n = n/p = 48/4 = 12$ numbers each that are initially on the Pro-

cesses 0, 1, 2, 3, respectively. We note that the numbers in each row above are already locally sorted, thus not requiring a local sort of step (i) of the algorithm. To achieve a globally sorted vector, stored in local vector *sorted* on each process, requires for this sample dataset the communication of a block length $l_n/p = 12/4 = 3$ of numbers among all pairs of processes. For example for Process 0 (data in first row of the matrix), the group of numbers 13, 14, 15 needs to be sent to Process 1, which — coming from Process 0 — will show up as the first numbers in vector *sorted* on Process 1. This communication gives the result that can be summarized in the matrix *sorted* =

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

which lists in each row the numbers in *sorted* on the Processes 0, 1, 2, 3, respectively; notice the group 13, 14, 15 at the start of the second row for Process 1. We observe that the numbers in each row of this matrix are sorted and no local sort of the vectors *sorted* on each process in step (iv) of the algorithm is needed.

Memory Predictions

To stress the network as much as possible, we need to make the amount of data communicated between each pair of parallel processes as large as possible. In the example dataset designed so far, this amount of data is simply a block length l_n/p of numbers, which is thus given indirectly by choosing n and p . We introduce now another independent variable m that allows to control the amount of this data independently from n and p . Namely, in place of each number in the example for the arrays *unsorted* and *sorted* we use a struct that contains an array of m double-precision numbers. We can now think of the numbers in *unsorted* and *sorted* as indices into an array of structs, and communicating each struct requires the sending and receiving of m doubles. In other words, in place of communicating l_n/p numbers between process pairs, we communicate l_n/p many vectors of m double-precision numbers, called a block size l_n/p of m -vectors for short.

The benefit of introducing m is that we can now explicitly control the memory requirement of the arrays by choosing m . The two local vectors, *unsorted* and *sorted*, are the only major variables in memory. Since each node on the cluster tara has 24 GB of memory, total local memory must be less than 24 GB per node. To comfortably stay within this memory also on one node, the vectors are chosen as less than 10 GB each to insure that memory does not become a problem. In Table 1, we specialize 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 1 provides the formulas for

Table 3. 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

memory predictions for a global array of length n consisting of vectors of m doubles. The global vector is then divided into p local arrays of block length $l_n = n/p$ of m -vectors, such that the size of the local vectors is constant per number of processes p . The block size of the portions in the arrays *unsorted* and *sorted* that need to be communicated between process pairs is then the block size l_n/p of m doubles.

As explained in Section **Background**, we wish to use $N = 1, 3, 9, 18, 36$ nodes, which we choose by name so as to ensure their optimal connectivity in the leaf modules in the InfiniBand interconnect. Running then 8 processes on each node with two quad-core CPUs for most contention of network traffic, we have $p = 8N = 8, 24, 72, 144, 288$ parallel processes in a job. These numbers are listed in the first two rows of Table 1.

In all following experiments, we fix the length of the global array at $n = 2 \cdot (8 \cdot 18 \cdot 2 \cdot 3)^2 = 1,492,992$. This number is designed to ensure that all desired values of the block length l_n/p divide n without remainder. That is, n needs to be divisible without remainder not just by p , but by p^2 , since $l_n/p = n/p^2$. We had originally planned to consider also some other values of p , hence some additional factors are contained in the choice of n that are not strictly needed going forward; this generality does not impact the conclusions.

RESULTS

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, by fixing $m = 512$ as constant, which makes the global memory an estimated 6 GB for each of the two arrays *unsorted* and *sorted* in Table 2 for $n = 1,492,992$. The local memory of $l_n = n/p$ decreases from 729 MB to 20 MB, as the numbers of processes p and nodes N increase. This effect of decreasing memory is amplified for the block size l_n/p , namely from process to process it decreases by another factor of p , so that 93,312 kB decrease dramatically to 72 kB eventually.

Table 3 and Figure 2 both display the runtimes in seconds for the call to the `MPI_Alltoallv` command sending and receiving $m l_n/p$ doubles between processes for the choices of parameters in Table 2. The results show that the communication speed of the All-to-All command in fact decreases with additional nodes in the parallel job. The plot brings out how dramatic the decrease is.

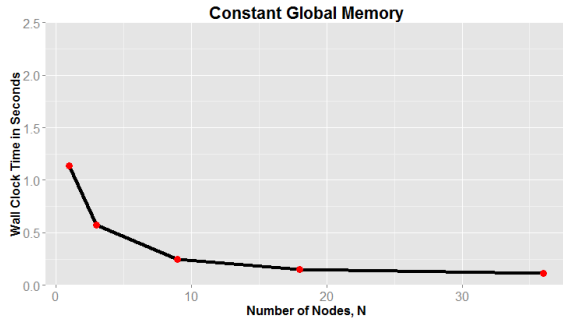


Figure 2. Constant global memory for $m = 512$: wall clock time in seconds vs. number of nodes.

Table 5. Constant local memory for $m = 512 N$: wall clock time in seconds.

Nodes N	1	3	9	18	36
Processes p	8	24	72	144	288
$m = 512 N$	1.14	1.64	2.09	2.28	2.30

This is remarkable and demonstrates that the high-performance InfiniBand interconnect can handle the stress of contention in All-to-All communications successfully, provided the overall problem does not increase in size, that is, for constant global memory. In the context of a larger algorithm that uses All-to-All communications, this communication will not be a bottleneck.

Experiment with Constant Local Memory

The results up to this point used a constant *global* memory with m constant for any number of p parallel processes, which leads to a rapidly decreasing block size l_n/p between pairs of processes. Now, in order to keep the block size in the All-to-All communications as large as possible, the vector length m will be designed to increase with increasing $p = 8N$, as reported in Table 4. The goal is to keep the block size l_n/p as large as possible, while p increases. This is limited by the requirement that the arrays *unsorted* and *sorted* need to fit in memory on each node. This implies that we cannot keep the block size l_n/p itself constant, but only the local memory controlled by l_n ; thus we pick the function $m = 512 N$, so that the local memory of each array *unsorted* and *sorted* is 729 MB for all values of p ; we call this the case of constant *local* memory. The block size l_n/p will then still decrease with increasing p , but less dramatically than before. This is seen in Table 4 in a decrease from 93,312 kB to 2,592 kB, which is a much larger final value than the 72 kB in Table 2. Notice the size of the global array increasing to a total of 205 GB on 36 nodes with increasing m , showing what significant problem size is eventually considered in this experiment.

The results displayed in Table 5 and Figure 3 present the runtimes in seconds, as we increase the number of processes $p = 8N$, while holding local memory on N nodes constant using $m = 512 N$. With the local memory held constant, the run times steadily increase as we increase

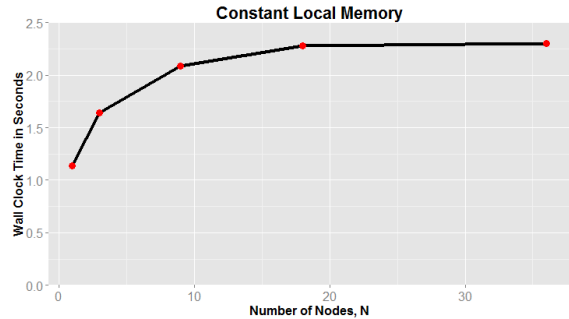


Figure 3. Constant local memory for $m = 512 N$: wall clock time in seconds vs. number of nodes.

N . The plot in Figure 3 brings the increase out very well, in particular compared to the decreasing line in Figure 2, which started from the same initial data point. Thus, in this case of maximum contention on the network, it is apparent that the run times increase with the numbers of processes, and the InfiniBand interconnect is eventually overcome by the All-to-All contention. For the use of All-to-All communication commands as building blocks in larger algorithms, this means that parallel scalability studies cannot succeed, since communication time worsens as the number of processes p increases.

CONCLUSIONS

As the results in Section Experiment with Constant Local Memory show, with local memory constant and contention on the network maximized, the run times for `MPI_Alltoallv` grow with the number of processes. This test case demonstrates that stress on the InfiniBand network can be created and will eventually limit the scalability of parallel algorithms that use All-to-All communications as building blocks. This is contrasted by the results in Experiment with Constant Global Memory that prove efficient behavior of the All-to-All communications, as long as the global memory stays constant, because this implies a dramatic decrease of the block size of the pairwise communications in the `MPI_Alltoallv` command. Both situations can occur in practice: The case of constant global memory occurs in strong scalability studies, where a fixed problem is divided onto the parallel processes, while the case of constant local memory appears in weak scalability studies, where the problem size is increased such that the local memory of each node is used at a constant level. Therefore, it depends on the structure of the code and its used, whether the InfiniBand interconnect becomes a limiting factor in parallel performance.

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, where they

were originally reported in the tech. rep. [7]. 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 through a contract with the NSA. Graduate RA Xuan Huang was supported by UMBC as HPCF RA.

REFERENCES

1. Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Kumar, S., Lusk, E., Thakur, R., and Träff, J. L. MPI on a million processors. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2009, 20–30.
2. Balaji, P., Narravula, S., Vaidyanathan, K., Krishnamoorthy, S., Wu, J., and Panda, D. K. Sockets direct protocol over InfiniBand in clusters: Is it beneficial? In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, IEEE (2004), 28–35.
3. Hoefer, T., Lumsdaine, A., and Rehm, W. Implementation and performance analysis of non-blocking collective operations for MPI. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, IEEE (2007), 1–10.
4. Kandalla, K., Mancini, E. P., Sur, S., and Panda, D. K. Designing power-aware collective communication algorithms for InfiniBand clusters. In *Parallel Processing (ICPP), 2010 39th International Conference on*, IEEE (2010), 218–227.
5. Kandalla, K., Subramoni, H., Tomko, K., Pekurovsky, D., and Panda, D. A novel functional partitioning approach to design high-performance MPI-3 non-blocking Alltoallv collective on multi-core systems. In *42nd International Conference on Parallel Processing (ICPP) 2013*, IEEE (2013), 611–620.
6. Kandalla, K., Subramoni, H., Tomko, K., Pekurovsky, D., Sur, S., and Panda, D. K. High-performance and scalable non-blocking all-to-all with collective offload on InfiniBand clusters: a study with parallel 3D FFT. *Computer Science-Research and Development* 26, 3-4 (2011), 237–246.
7. Mistry, N., Ramsey, J., Wiley, B., Yanchuck, J., Huang, X., Gobbert, M. K., Mineo, C., and Mountain, D. Contention of communications in switched networks with applications to parallel sorting. Tech. Rep. HPCF-2013-13, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2013.
8. Pacheco, P. S. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.
9. Raim, A. M., and Gobbert, M. K. Parallel performance studies for an elliptic test problem on the cluster tara. Tech. Rep. HPCF-2010-2, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.
10. Reuther, A., Funk, A., Kepner, J., McCabe, A., Arcand, W., Currie, T., Hubbell, M., and Michaleas, P. Benchmarking the MIT LL HPCMP DHPI system. In *DoD High Performance Computing Modernization Program Users Group Conference, 2007*, IEEE (2007), 310–316.
11. Thibeault, C. M., Minkovich, K., O'Brien, M. J., Harris Jr, F. C., and Srinivasa, N. Efficiently passing messages in distributed spiking neural network simulation. *Frontiers in Computational Neuroscience* 7 (2013).
12. White, C. M. *Data Communications and Computer Networks: A Business User's Approach*. Course Technology, 2013.
13. Wu, J., Wyckoff, P., and Panda, D. PVFS over InfiniBand: Design and performance evaluation. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, IEEE (2003), 125–132.

Table 1. Formulas 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	mn	mn	mn	mn	mn
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	$m\frac{n}{p^2}$	$m\frac{n}{p^2}$	$m\frac{n}{p^2}$	$m\frac{n}{p^2}$	$m\frac{n}{p^2}$

Table 2. 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. Constant local memory for $m = 512N$: predicted memory usage for one array.

Nodes N	1	3	9	18	36
Processes p	8	24	72	144	288
$m = 512N$	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