A Study of Process Arrival Patterns for MPI Collective Operations

Ahmad Faraj
Blue Gene Software Development
IBM Corporation
Rochester, MN 55901
faraja@us.ibm.com

Pitch Patarasuk Xin Yuan*
Department of Computer Science
Florida State University
Tallahassee, FL 32306
{patarasu, xyuan}@cs.fsu.edu

Abstract

Process arrival pattern, which denotes the timing when different processes arrive at an MPI collective operation, can have a significant impact on the performance of the operation. In this work, we characterize the process arrival patterns in a set of MPI programs on two common cluster platforms, use a micro-benchmark to study the process arrival patterns in MPI programs with balanced loads, and investigate the impacts of different process arrival patterns on collective algorithms. Our results show that (1) the differences between the times when different processes arrive at a collective operation are usually sufficiently large to affect the performance; (2) application developers in general cannot effectively control the process arrival patterns in their MPI programs in the cluster environment: balancing loads at the application level does not balance the process arrival patterns; and (3) the performance of collective communication algorithms is sensitive to process arrival patterns. These results indicate that process arrival pattern is an important factor that must be taken into consideration in developing and optimizing MPI collective routines. We propose a scheme that achieves high performance with different process arrival patterns, and demonstrate that by explicitly considering process arrival pattern, more efficient MPI collective routines than the current ones can be obtained.

Keywords: MPI, collective communication, process arrival pattern, communication algorithm.

1 Introduction

MPI collective operations are used in most MPI applications and they account for a significant portion of the communication time in some applications [25]. Yet, compared to their

*Contact Author: Xin Yuan, xyuan@cs.fsu.edu, phone: (850)644-9133, fax: (850)644-0058.

point-to-point counterparts, MPI collective operations have received less attention. Some fundamental issues in collective operations are still not well understood [11].

The term *process arrival pattern* denotes the timing when different processes arrive at an MPI collective operation (the call site of the collective routine). A process arrival pattern is said to be *balanced* when all processes arrive at the call site roughly at the same time such that the arrival timing does not dramatically affect the performance of the operation. Otherwise, it is said to be *imbalanced*. The terms, balanced and imbalanced arrival patterns, are quantified in Section 3.

The process arrival pattern can have a profound impact on the performance because it decides the time when each process can start participating in an operation. Unfortunately, this important factor has been largely overlooked by the MPI developers community. We are not aware of any study that characterizes process arrival patterns in application programs. MPI developers routinely make the implicit assumption that all processes arrive at the same time (a balanced process arrival pattern) when developing and analyzing algorithms for MPI collective operations [9, 10, 11, 31]. However, as will be shown in this paper, the process arrival patterns in MPI programs, even well designed programs with balanced loads, are more likely to be sufficiently imbalanced to affect the performance.

The imbalanced process arrival pattern problem is closely related to the application load balancing problem. MPI practitioners who have used a performance tool such as Jumpshot to visually see the process arrival times for their collectives should have noticed the imbalanced process arrival pattern problem. However, these two problems are significantly distinct in their time scales: the time differences that cause load imbalance at the application level are usually orders of magnitude larger than those causing imbalanced process arrival patterns. It is often possible to "balance" application loads by applying some load balancing techniques. However, as will be shown later, it is virtually impossible to balance the process arrival patterns in typical cluster environments: even programs with perfectly balanced loads tend to have imbalanced process arrival patterns.

This work is concerned about efficient implementations of MPI collective routines. Application load balancing, although important, requires techniques at the application level and is beyond the scope of this paper. In order for applications with balanced loads to achieve high performance, it is essential that the MPI library can deliver high performance with different (balanced and imbalanced) process arrival patterns. Hence, from the library implementer point of view, it is crucial to know (1) the process arrival pattern characteristics that summarize how MPI collective routines are invoked; (2) whether the process arrival pattern can cause performance problems in the library routines; and (3) how to deal with the problem and make the library most efficient in practice. These are the questions that we try to answer in this paper. Note that an MPI library does not differentiate applications with different load balancing characteristics. It should try to deliver the best performance to applications with or without balanced loads.

We study the process arrival patterns of a set of MPI benchmarks on two commercial off-the-shelf (COTS) clusters: a high-end Alphaserver cluster and a low-end Beowulf cluster. These two clusters are representative and our results can apply to a wide range of practical clusters. We characterize the process arrival patterns in MPI programs, use a micro-benchmark to examine the process arrival patterns in applications with balanced loads and to study the causes of the imbalanced process arrival patterns, and investigate the impacts of different process arrival patterns on some commonly used algorithms for MPI collective operations. The findings include:

- The process arrival patterns for MPI collective operations are usually imbalanced. Even in a micro-benchmark with a perfectly balanced load, the process arrival patterns are still imbalanced.
- In cluster environments, it is virtually impossible for application developers to control the process arrival patterns in their programs without explicitly invoking a global

synchronization operation. Many factors that can cause imbalance in computation and communication are beyond the control of the developers. Balancing the loads at the application level is insufficient to balance the process arrival patterns.

- The performance of MPI collective algorithms is sensitive to the process arrival pattern. In particular, the algorithms that perform better with a balanced arrival pattern tend to perform worse when the arrival pattern becomes more imbalanced.

These findings indicate that for an MPI collective routine to be efficient in practice, it must be able to achieve high performance with different (balanced and imbalanced) process arrival patterns. Hence, MPI library implementers must take process arrival pattern into consideration when developing and optimizing MPI collective routines. We propose a scheme that uses a dynamic adaptive mechanism to deal with the imbalanced process arrival pattern problem, and demonstrate that by explicitly considering process arrival pattern, more robust MPI collective routines than the current ones can be developed.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 formally describes the process arrival pattern and the parameters we use to characterize it. Section 4 presents the statistics of process arrival patterns in a set of benchmark programs. In Section 5, we study a micro-benchmark that has a perfectly balanced load and investigate the causes for such a program to have imbalanced process arrival patterns. In Section 6, we evaluate the impacts of process arrival patterns on some common algorithms for MPI collective operations. In Section 7, we propose and evaluate a potential solution to the imbalanced process arrival pattern problem. Finally, Section 8 concludes the paper.

2 Related Work

Understanding the application/system behavior is critical for developing an efficient MPI library. Due to its importance, there are numerous research efforts focusing on analyzing MPI communication behavior. In [36], the performance of parallel applications is analyzed using a technique that automatically classifies inefficiencies in point-to-point communications. The study analyzes the usage of MPI collective communication routines and their elapsed times. The studies in [6, 14] performed quantitative measurements of the static and dynamic communication routines in parallel applications. Work in [29] performed statistical analysis of all-to-all elapsed communication time on the IBM SP2 machine to understand the causes of the performance drop as the number of processors increases. The researchers in [5, 30] examined the NAS parallel benchmarks [20] to quantitatively describe the MPI routine usage and the distribution of message sizes. The analysis performed on parallel applications in these studies (and other similar studies) often involves the investigation of communication attributes such as the type of MPI routines, the message size, the message volume, the message interval, the bandwidth requirement, and the communication elapsed time. Our study focuses on a specific communication attribute for collective operations, the process arrival pattern, which has not been studied before. It must be noted that the process arrival pattern is affected not only by the application, but also by the operating system, the system hardware, and the communication library. Understanding the characteristics and the impacts of process arrival pattern is important for developing advanced communication schemes [7, 8, 12, 34, 37] that maximize the MPI collective communication performance.

Researchers are aware of the imbalanced arrival pattern problem and have developed adaptive collective communication algorithms that can automatically change the logical topologies used in the algorithms based on various factors including the process arrival pattern. Example algorithms include the all-reduce and barrier algorithms in [17] and the broadcast and reduce algorithms in [28]. Our work advocates further development of such algorithms as well as other mechanisms to handle the imbalanced process arrival pattern problem.

3 Process arrival pattern

Let n processes, p_0, p_1, ..., p_{n-1}, participate in a collective operation. Let a_i be the time when process p_i arrives at the collective operation. The *process arrival pattern* can be represented by the tuple $(a_0, a_1, ..., a_{n-1})$. The average process arrival time is $\bar{a} = \frac{a_0+a_1+...+a_{n-1}}{n}$. Let f_i be the time when process p_i finishes the operation. The *process exit pattern* can be represented by the tuple $(f_0, f_1, ..., f_{n-1})$. The elapsed time that process p_i spends in the operation is thus $e_i = f_i - a_i$; the total time is $e_0 + e_1 + ... + e_{n-1}$; and the average per node time is $\bar{e} = \frac{e_0+e_1+...+e_{n-1}}{n}$. In an application, the total time or the average per node time accurately reflects the time that the program spends on the operation. We will use the average per node time ($\bar{e}$) to denote the performance of an operation (or an algorithm).

We will use the term **imbalance** in the process arrival pattern to signify the differences in the process arrival times at a collective communication call site. Let δ_i be the time difference between p_i's arrival time a_i and the average arrival time $\bar{a}$, $\delta_i = |a_i - \bar{a}|$. The imbalance in the process arrival pattern can be characterized by the *average case imbalance time*, $\bar{\delta} = \frac{\delta_0+\delta_1+...+\delta_{n-1}}{n}$, and the *worst case imbalance time*, $\omega = max_i\{a_i\} - min_i\{a_i\}$. Figure 1 depicts the described parameters in a process arrival pattern.

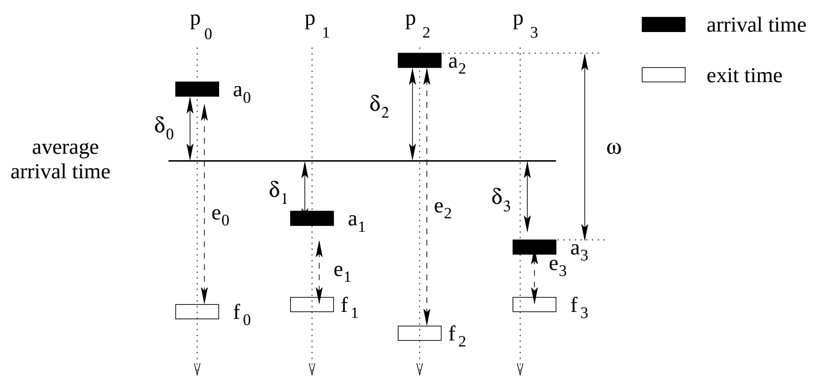


Figure 1: Process arrival pattern

An MPI collective operation typically requires each process to send multiple messages. A collective algorithm organizes the messages in the operation in a certain manner. For example, in the *pair* algorithm for *MPI_Alltoall* [31], the messages in the all-to-all operation are organized in $n - 1$ phases: in phase i, $0 \leq i \leq n - 1$, process p_j exchanges a message with process $p_{j \oplus i}$ ($\oplus$ is the exclusive or operator). The impact of an imbalanced process arrival pattern is mainly caused by early completions or late starts of some messages in the operation. In the *pair* algorithm, early arrivals of some processes will cause some processes to complete a phase and start the next phase while other processes are still in the previous phase, which may cause system contention and degrade the performance. Hence, the impacts of an imbalanced process arrival pattern can be better characterized by the number of messages that can be sent during the period when some processes arrive while others do not. To capture this notion, we normalize the worst case and average case imbalance times by the time to communicate one message. The normalized results are called the *average/worst case imbalance factor*. Let T be the time to communicate one message in the operation. The *average case imbalance factor* equals to $\frac{\bar{\delta}}{T}$ and the *worst case imbalance factor* equals to $\frac{\omega}{T}$. A worst case imbalance factor of 20 means that by the time the last process arrives at the operation, the first process may have sent twenty messages. In general, a process arrival pattern is **balanced** if the worst case imbalance factor is less than 1 (all processes arrive within a message time) and **imbalanced**, otherwise.

4 Process arrival patterns in MPI programs

4.1 Platforms

The process arrival pattern statistics are collected on two representative platforms. The first is the Lemieux machine located in Pittsburgh Supercomputing Center (PSC) [24]. The machine consists of 750 Compaq Alphaserver ES45 nodes connected by Quadrics, each of the nodes includes four 1GHz SMP processors with 4GB of memory. The system runs Tru64 Unix operating system. All benchmarks are compiled with the native *mpicc* and linked with the native MPI and ELAN libraries. ELAN is a low-level internode communication library for Quadrics. On Lemieux, the experiments are conducted with a batch partition of 32, 64, and 128 processors (4 processors per node). The second platform is a 16-node Beowulf cluster, whose nodes are Dell Dimension 2400, each with a 2.8GHz P4 processor and 128MB of memory. All machines run Linux (Fedora) with the 2.6.5-1.358 kernel. These machines are connected by a Dell Powerconnect 2624 1Gbps Ethernet switch. This system uses MPICH 2-1.0.1 for communication. All programs are compiled with the *mpicc* that comes with the MPICH package. Some of the times and the corresponding bandwidths (BW) for one way point-to-point communications with different message sizes on the two platforms are summarized in Table 1. These numbers, which are obtained using a ping-pong program, are used to compute imbalance factors.

Table 1: One way point-to-point communication time and bandwidth on Lemieux and Beowulf

message	Lemieux		Beowulf	
size	time (ms)	BW (MB/s)	time (ms)	BW (MB/s)
4B	0.008	0.50	0.056	0.07
256B	0.008	32.0	0.063	4.10
1KB	0.021	49.5	0.088	11.6
4KB	0.029	141	0.150	27.3
16KB	0.079	207	0.277	59.1
32KB	0.150	218	0.470	69.7
64KB	0.291	225	0.846	77.5
128KB	0.575	228	1.571	83.4

4.2 Benchmarks

Table 2 summarizes the seven benchmarks. For reference, we show the code size as well as the execution and collective communication elapsed times for running the programs on $n = 64$ processors on Lemieux. Table 3 shows the major collective communication routines in the benchmarks and their dynamic counts and message sizes (assuming $n = 64$). There are significant collective operations in all programs. Next, we briefly describe each benchmark and the related parameters/settings used in the experiments.

FT (Fast-Fourier Transform) is one of the parallel kernels included in NAS parallel benchmarks [20]. FT solves a partial differential equation using forward and inverse FFTs. The collective communication routines used in FT include *MPI_Alltoall*, *MPI_Barrier*, *MPI_Bcast*,

Table 2: Summary of benchmarks (times are measured on Lemieux with 64 processors)

benchmark	description	#lines	exec. time	comm. time
FT	solves PDE with forward and inverse FFTs	2234	13.4s	8.3s
IS	sorts integer keys in parallel	1091	2.2s	1.6s
LAMMPS	simulates dynamics of molecules in different states	23510	286.7s	36.1s
PARADYN	sim. dynamics of metals and metal alloys molecules	6252	36.6s	33.1s
NBODY	simulates effects of gravitational forces on N bodies	256	59.5s	1.5s
NTUBE 1	performs molecular dynamics calculations of diamond	4480	894.4s	32.3s
NTUBE 2	performs molecular dynamics calculations of diamond	4570	852.9s	414.1s

and *MPI_Reduce* with most communications being carried out by *MPI_Alltoall*. We used the class B problem size supplied by the NAS benchmark suite in the evaluation.

Table 3: The dynamic counts of major collective communication routines in the benchmarks ($n = 64$)

benchmark	routine	msg size (byte)	dyn. count
FT	alltoall	131076	22
	reduce	16	20
IS	alltoallv	33193*	11
	allreduce	4166	11
	alltoall	4	11
LAMMPS	allreduce	42392	2012
	bcast	4-704	48779
	barrier		4055
PARADYN	allgatherv	6-1290*	16188
	allreduce	4-48	13405
NBODY	allgather	5000	300
NTUBE 1	allgatherv	16000*	1000
NTUBE 2	allreduce	8	1000

* For a v-version routine, the number is the average of all message sizes in the routine. A range indicates there are multiple routines with different (average) message sizes.

IS (Integer Sort) is a parallel kernel from NAS parallel benchmarks. It uses bucket sort to order a list of integers. The MPI collective routines in IS are *MPI_Alltoall*, *MPI_Alltoallv*, *MPI_Allreduce*, and *MPI_Barrier* with most communications carried out by the *MPI_Alltoallv* routine. The class B problem size is used in the experiments.

LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [16] is a classical parallel molecular dynamics code. It models the assembly of particles in a liquid, solid, or gaseous state. The code uses *MPI_Allreduce*, *MPI_Bcast*, and *MPI_Barrier*. We ran the program with 1720 copper atoms for 3000 iterations.

PARADYN (Parallel Dynamo) [22] is a molecular dynamics simulation. It utilizes the embedded atom method potentials to model metals and metal alloys. The program uses *MPI_Allgather*, *MPI_Allgatherv*, *MPI_Allreduce*, *MPI_Bcast*, and *MPI_Barrier*. In the experiments, we simulated 6750 atoms of liquid crystals in 1000 time steps.

NBODY [21] simulates over time steps the interaction, in terms of movements, positions and other attributes, among the bodies as a result of the net gravitational forces exerted on one another. The code is a naive implementation of the nbody method and uses *MPI_Allgather* and *MPI_Gather* collective communications. We ran the code with 8000 bodies and for 300 time steps.

NTUBE 1 performs molecular dynamics calculations of thermal properties of diamond [26]. This version of the code uses *MPI_Allgatherv* and *MPI_Reduce*. In the evaluation, the program ran for 1000 steps and each processor maintained 100 atoms.

NTUBE 2 is a different implementation of the Nanotube program. The functionality of NTUBE 2 is exactly the same as NTUBE 1. The collective communication routines used in this program are *MPI_Allreduce* and *MPI_Reduce*. In the evaluation, the program ran for 1000 steps with each processor maintaining 100 atoms.

4.3 Data collection

To investigate process arrival patterns and other statistics of MPI collective communications, we develop an MPI wrapper library. The wrapper records an event at each MPI process for each entrance and exit of each MPI collective communication routine. An event records information about the timing, the operation, the message size, etc. The times are measured using the *MPI_Wtime* routine. Events are stored in memory during the program execution until *MPI_Finalize* is called, when all processors write the events to log files for post-mortem analysis. The functionality of our wrapper is similar to that of the standard MPI Profiling interface (PMPI), we use our own wrapper for its flexibility and future extension. Accurately measuring the times on different machines requires a globally synchronized clock. On Lemieux, such a synchronized clock is available. On the Beowulf cluster, the times on different machines are not synchronized. We resolve the problem by calling an *MPI_Barrier* after *MPI_Init* and having all measured times normalized with respect to the exit time of the *MPI_Barrier*. Basically, we are assuming that all (16) machines exit a barrier operation at the same time. This introduces inaccuracy that is roughly equal to the time to transfer several small messages.

4.4 Process arrival pattern statistics

In this sub-section, we focus on presenting the process arrival pattern statistics. The causes for MPI applications to have such behavior will be investigated in the next section. Table 4 shows the average of the worst/average case imbalance factors among all collective routines in each benchmark on Lemieux and the Beowulf cluster. The table reveals several notable observations. First, the averages of the worst case imbalance factors for all programs on both clusters are quite large, even for FT, whose computation is fairly balanced. Second, the process arrival pattern depends heavily on the system architecture. For example, the imbalance factors for NTUBE 1 and NTUBE 2 are much larger on Lemieux than on the Beowulf cluster. This is because these two programs were designed for single CPU systems. When running them on Lemieux, an SMP cluster, the process arrival patterns become extremely imbalanced. Overall, the imbalance factors for all programs on both platforms are large: the best average worst case imbalance factor is 19 for Lemieux (LAMMPS) and 17 for Beowulf (NTUBE 1).

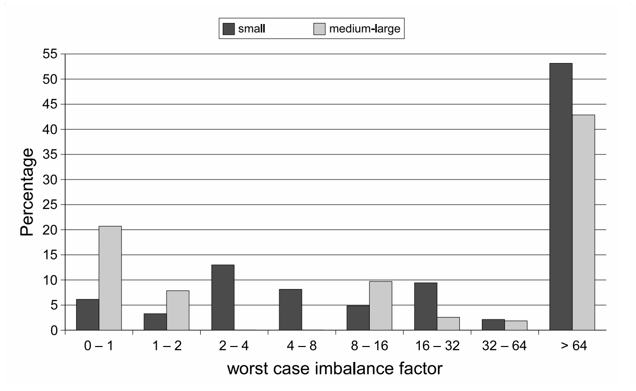
Operations that account for most of the communication times typically have large message sizes. In Figure 2, we distinguish operations with medium/large message sizes (> 1000B) from those with small message sizes (≤ 1000B). Part (a) of Figure 2 shows the distribution of the worst case imbalance factors for the results on Lemieux (128 processors) while part

Table 4: The average of worst case ($\frac{\bar{\omega}}{T}$) and average case ($\frac{\bar{\delta}}{T}$) imbalance factors among all collective routines on two the platforms

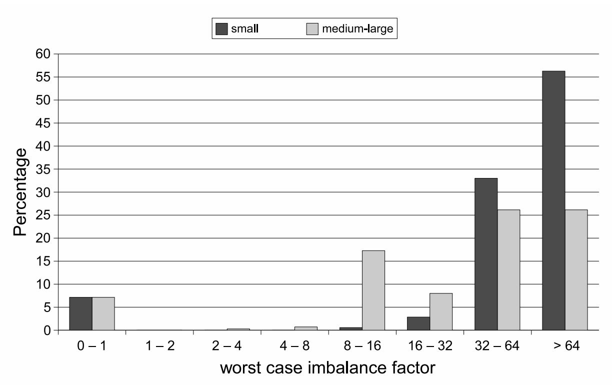
benchmark	imbalance factor			
	Lemieux (n = 128)		Beowulf	
	average	worst	average	worst
FT	91.0	652	278	1.2K
IS	61.0	358	1.4K	11K
LAMMPS	4.00	19.0	273	630
PARADYN	9.10	46.0	12.0	79.0
NBODY	13.0	132	12.0	50.0
NTUBE 1	4.8K	38K	4.30	17.0
NTUBE 2	85K	347K	9.00	39.0

(b) shows the results on the Beowulf cluster. All benchmarks are equally weighted when computing the distribution. As expected, arrival patterns for operations with large messages are in general less imbalanced than those for operations with small messages. This is mainly due to the way the imbalance factors are computed: larger messages mean larger per message time (T). However, there is a significant portion of operations with both small and medium/large sizes having large imbalance factors and only a small fraction of the operations are balanced. In particular, for operations with medium/large messages, only a small percentage (21% on Lemieux and 6% on Beowulf) have balanced process arrival patterns (a worst case imbalance factor less than 1). The percentage is smaller for operations with small messages. This indicates that imbalanced process arrival patterns are much more common than balanced process arrival patterns.

In Table 5, we further narrow our focus on the imbalance factors for the collective operations that are important in the benchmarks. These are the operations that appear in the main loop and account for a significant amount of application time. Compared with the imbalance factors shown in Table 4, we can see that the process arrival patterns for these important routines are generally less imbalanced than the average of all routines in the applications, which reflects the fact that programmers are more careful about the load



(a) Lemieux (128 processors)



(b) Beowulf cluster

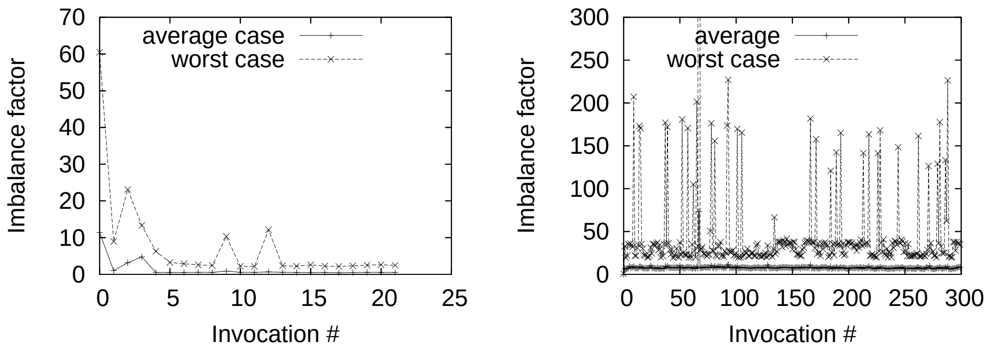
Figure 2: The distribution of worst case imbalance factors ($\frac{\bar{\omega}}{T}$)

balancing issue in the main loop. However, the process arrival patterns for these important routines are still quite imbalanced. On both platforms, only the *MPI_Alltoallv* in IS can be classified as having balanced process arrival patterns. Examining the source code reveals that this routine is called right after another MPI collective routine.

Table 5: The imbalance factor for major routines

	major routine	imbalance factor			
		Lemieux ($n = 128$)		Beowulf	
		ave.	worst	ave.	worst
FT	alltoall	2.90	24.0	26.0	124
IS	alltoallv	0.00	0.20	0.20	0.80
	allreduce	145	756	4.4K	34K
LAMMPS	bcast	0.20	3.40	299	671
	allreduce	16.3	91.3	24	132
	barrier	28.6	157.3	106	442
PARADYN	allgatherv	0.80	6.50	10.0	66.5
	allreduce	15.7	73.3	14.0	93.0
NBODY	allgather	13.0	132	12.0	50.0
NTUBE 1	allgatherv	78.8	345	3.50	14.0
NTUBE 2	allreduce	83K	323K	9.00	39.0

Another interesting statistics is the characteristics of process arrival patterns for each individual call site. If the process arrival patterns for each call site in different invocations exhibit heavy fluctuation, the MPI routine for this call site must achieve high performance for all different types of process arrival patterns to be effective. On the other hand, if the process arrival patterns for the same call site is statistically similar, the MPI implementation will only need to optimize for the particular type of process arrival patterns. In the experiments, we observe that the process arrival patterns for different invocations of the same call site exhibit a *phased* behavior: the process arrival patterns are statistically similar for a period of time before they change. In some cases, the process arrival patterns for the same call site are statistically similar in the whole program. Figure 3 depicts two representative cases: the imbalance factors for the *MPI_Alltoall* in FT and the *MPI_Allgather* in NBODY. As can be seen from the figure, the majority of the calls have similar worst case and average case imbalance factors despite some large spikes that occur once in a while. This indicates that it might be feasible to customize the routine for each MPI call site and get good performance.



(a) FT: *MPI_Alltoall* (b) NBODY: *MPI_Allgather*

Figure 3: The imbalance factors for *MPI_Alltoall* in FT and *MPI_Allgather* in NBODY on Lemieux ($n = 128$)

4.5 Summary

While we expect to see some imbalance process arrival patterns in MPI programs, it is surprising to see the very low percentage of balanced process arrival patterns. The low percentage applies to applications whose loads are fairly balanced, to collective operations in the main loops where load balancing is critical for the performance of the applications, and to operations with all different message sizes.

5 Process arrival patterns in a micro-benchmark

Since a well designed MPI program typically has a balanced computation load, understanding the process arrival patterns in such programs is particularly important. One surprising result in the previous section is that even programs with evenly distributed computation loads have very imbalanced process arrival patterns. However, these programs are too complex to determine what exactly is causing the imbalanced process arrival patterns. In this section, we study a simple micro-benchmark, shown in Figure 4, where all processes perform exactly the same computation and communication (perfectly balanced load). The goals are (1) to determine whether application programmers can control the critical process arrival patterns in their MPI programs by balancing the load at the application level, and (2) to investigate the causes of the imbalanced process arrival patterns. In this micro-benchmark, a barrier is called before the main loop that is executed 1000 times. The loop body consists the simulated computation (lines (4)-(6)) and the *MPI_Alltoall()* routine (line (8)). The computation time can be adjusted by changing the parameter $XTIME$.

```
(1) MPI_Barrier(...);
(2) for (i=0; i<1000; i++) {
(3)    /* compute for roughly X milliseconds */
(4)    for (m=0; m< XTIME; m++)
(5)       for (k=1, k<1000; k++)
(6)          a[k] = b[k+1] - a[k-1] * 2;
(7)    arrive[i] = MPI_Wtime();
(8)    MPI_Alltoall(...);
(9)    leave[i] = MPI_Wtime()
(10)}
```

Figure 4: Code segment for a micro-benchmark

We measured the process arrival patterns for the all-to-all operation. We will report results for message size 64KB. Smaller message sizes result in larger imbalance factors. The average computation time in each node is set to 200ms for both clusters. Figure 5 shows the worst and average case imbalance factors in each invocation in a typical execution on each of the two platforms. In both clusters, the process arrival patterns are quite imbalanced even though all processors perform exactly the same operations. The imbalance factors on Lemieux are larger than those on the Beowulf cluster for several reasons. First, Lemieux has more processes and thus has a higher chance to be imbalanced. Second, on Lemieux, different jobs share the network in the system, the uncertainty in messaging can cause the imbalance. Third, Lemieux has a faster network, the same imbalance time results in a larger imbalanced factor.

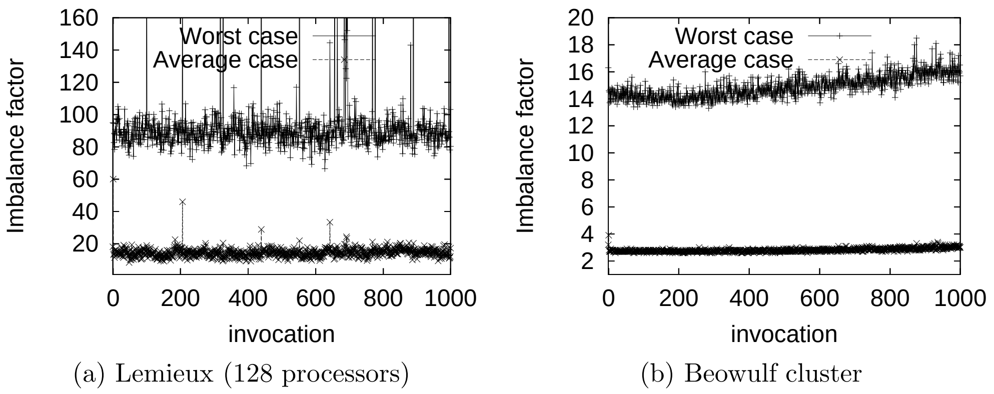


Figure 5: Process arrival patterns in the micro-benchmark (64KB message size, 200ms computation time) on the two platforms

We further investigate the causes of the imbalanced process arrival patterns in this simple benchmark. For the *MPI_Alltoall* routine to have the imbalanced process arrival patterns shown in Figure 5, there can be only two potential causes. First, it might take different processors different times to run the (same) computation. An earlier study [23] has shown that this is indeed happening in some clusters and has attributed this phenomenon to the asynchronous operating system events. Second, it might take different processors different times to perform the communication (*MPI_Alltoall*). This imbalance in the communication is reflected in the process exit patterns. In the following, we study the relationship among the imbalance in process arrival patterns, computation times, and process exit patterns in the micro-benchmark. The worst case imbalance factor for a process exit pattern is defined similarly to that of a process arrival pattern. The *computation imbalance time* is defined as the maximum time among all processes to execute the computation minus the minimum time among all processes. To be consistent, we use the imbalance factor in the comparison, which is equal to the imbalance time divided by the time to send one message (64KB).

We change the $XTIME$ parameter such that the average computation time lasts for 50, 100, 200, 400, and 800ms. Due to the nondeterministic nature in the imbalance, we repeat each experiment 5 times, each on a different day. In each experiment, we collect data from the 1000 invocations of the all-to-all routine. We then use the data from the 5000 samples (5 experiments, 1000 samples per experiment) to compute the average values and the 95% confidence intervals of the imbalance factors for process arrival patterns, process exit patterns, and computation.

Tables 6 and 7 show the worst case imbalance factors for exit patterns, computation, and arrival patterns in the micro-benchmark for different computation times on the two platforms. In the tables, for each worst case (exit, computation, arrival) imbalance factor, we show the average value along with the confidence interval in the format of $ave \pm \frac{interval}{2}$, which denotes that the 95% confidence interval is $[ave - \frac{interval}{2}, ave + \frac{interval}{2}]$. There are a number of observations in the tables. First, when changing the computation time from 50ms to 800ms, the computation imbalance in both clusters increases almost linearly. Such imbalance in computation is inherent to the system and is impossible for application developers to overcome. This explains why in our benchmark study of the previous section, we only observe balanced process arrival patterns in consecutive collective routine calls. Second, the worst case imbalance factors for process arrival patterns are consistently larger than the computation imbalance factors, which indicates that the imbalances in both computation and communication are contributing to the imbalance in the process arrival patterns. Third, on Lemieux, the imbalance factors for process exit patterns are almost the same with different process arrival patterns while on the Beowulf cluster, the imbalance factors for process exit patterns are quite different. This is because different algorithms are used to implement *MPI_Alltoall* on the two clusters. On the Beowulf cluster, since the imbalance factors for process exit patterns are somewhat related to those for process arrival patterns, the imbalance effect may be accumulated as the simple benchmark executes. This explains the slight upward trend in the worst case imbalance factor in Figure 5 (b) (the worst case imbalanced factors are about 14.5 in the first few iterations and about 16.5 in the last few iterations). Nonetheless, the imbalance in communication, which is directly affected by the library implementation, is beyond the control of application developers.

Table 6: Effects of process exit patterns and computation imbalance on process arrival patterns on Lemieux (32 processors)

comp. time	worst case imbalance factor($\frac{\bar{\omega}}{T}$)		
	exit	computation	arrival
50ms	15.2 ± 0.7	23.4 ± 0.3	32.5 ± 0.8
100ms	15.2 ± 0.6	46.8 ± 1.6	54.5 ± 1.9
200ms	15.0 ± 0.3	87.4 ± 1.8	92.7 ± 1.9
400ms	15.1 ± 0.8	160 ± 1.9	164 ± 2.0
800ms	15.0 ± 0.3	320 ± 3.6	322 ± 3.6

Table 7: Effects of process exit patterns and computation imbalance on process arrival patterns on the Beowulf cluster

comp. time	worst case imbalance factor ($\frac{\bar{\omega}}{T}$)		
	exit	computation	arrival
50ms	5.07 ± 1.29	3.16 ± 0.02	7.02 ± 1.29
100ms	4.32 ± 1.00	7.52 ± 0.02	9.53 ± 0.99
200ms	3.71 ± 0.11	14.18 ± 0.02	15.17 ± 0.06
400ms	6.22 ± 0.23	31.41 ± 0.30	33.17 ± 0.35
800ms	11.62 ± 0.41	56.24 ± 0.05	56.29 ± 0.20

5.1 Summary

The way a program is coded is only one of many factors that can affect process arrival patterns. Other factors, such as system characteristics and library implementation schemes that can introduce the inherent imbalance in computation and communication, are beyond the control of application developers. Hence, it is unrealistic to assume that application programmers can balance the load at the application level to make the process arrival patterns balanced. The process arrival patterns in MPI programs are and will be imbalanced in most cases in a cluster environment.

6 Impacts of imbalanced process arrival patterns

We study the impact of the process arrival pattern on commonly used algorithms for *MPI_Alltoall* and *MPI_Bcast*. *MPI_Alltoall* and *MPI_Bcast* represent two types of MPI collective operations: *MPI_Alltoall* is an inherently synchronized operation, that is, a process can complete this operation only after all processes arrive; while *MPI_Bcast* is not an inherently synchronized operation.

The impacts of imbalanced process arrival patterns depend on the communication algorithms. For example, some communication algorithms such as the binomial tree *MPI_Bcast* algorithm are able to tolerate some degrees of imbalanced process arrival patterns since some nodes start the communication later than other nodes in these algorithms. As shown in the previous sections, the process arrival patterns are likely to be random and imbalanced: it is difficult to have a process arrival pattern that matches the algorithm. Hence, the impacts of imbalanced (random) process arrival patterns on the algorithms are not clear. This section tries to systematically study the impacts of imbalanced process arrival patterns on different types of algorithms.

The evaluated *MPI_Alltoall* algorithms include the *simple*, *Bruck*, *pair*, and *ring* algorithms. The *simple* algorithm basically posts all receives and all sends, starts the communications, and waits for all communications to finish. The *Bruck* algorithm [4] is a $lg(n)$-step algorithm that is designed for achieving efficient all-to-all with small messages. The *pair* algorithm only works when the number of processes, n, is a power of two. It partitions the all-to-all communication into $n-1$ steps. In step i, process p_j exchanges a message with process $p_{j \oplus i}$. The *ring* algorithm also partitions the all-to-all communication into $n-1$ steps. In step i, process p_j sends a messages to process $p_{(j+i) \ mod \ n}$ and receives a message from process $p_{(j-i) \ mod \ n}$. A more detailed description of these algorithms can be found in [31]. We also consider the native algorithm used in *MPI_Alltoall* on Lemieux, which is unknown to us.

The evaluated *MPI_Bcast* algorithms include the *flat tree*, *binomial tree*, *scatter-allgather*, and native algorithm on Lemieux, which is unknown to us. In the flat tree algorithm, the root sequentially sends the broadcast message to each of the receivers. In the binomial tree algorithm [19], the broadcast follows a hypercube communication pattern and the total number of messages that the root sends is $lg(p)$. The scatter-allgather algorithm, used for broadcasting large messages in MPICH [19], first distributes the *msize*-byte message to all nodes by a scatter operation (each node gets $\frac{msize}{p}$ bytes), and then performs an all-gather operation to combine the scattered messages to all nodes.

```
r = rand() % MAX_IF;
for (i=0; i<ITER; i++) {
  MPI_Barrier (...);
  for (j=0; j<r; j++) {
     ... /* computation time equal to one msg time */
  }
  t0 = MPI_Wtime();
  MPI_Alltoall(...);
  elapse += MPI_Wtime() - t0;
}
```

Figure 6: Code segment for measuring the impacts of imbalanced process arrival patterns

Figure 6 outlines the code segment we use to measure the performance with a controlled imbalance factor in the random process arrival patterns. The worst case imbalance factor is controlled by a variable *MAX_IF* (maximum imbalance factor). Line (1) generates a random number r that is bounded by *MAX_IF*. Before the all–to–all routine (or broadcast) is measured (lines 7-9), the controlled imbalanced process arrival pattern is created by first calling a barrier (line (3)) and then introducing some computation between the barrier and all-to-all routines. The time to complete the computation is controlled by r. The time spent in the loop body in line (5) is made roughly equal to the time for sending one message (see Table 1), and the total time for the computation is roughly equal to the time to send r messages. Hence, the larger the value of *MAX_IF* is, the more imbalanced the process arrival pattern becomes. Note that the actual worst case imbalance factor, especially for small message sizes, may not be bounded by *MAX_IF* since the process exit patterns of *MPI_Barrier* may not be balanced.

For each process arrival pattern, the routine is measured 100 times ($ITER = 100$) and the average elapsed time on each node is recorded. For each *MAX_IF* value, we perform 32 experiments (32 random process arrival patterns with the same value of *MAX_IF*). The communication time is reported by the confidence interval with a 95% confidence level, computed from the results of the 32 experiments.

Figure 7 (a) shows the results for 1B all-to-all communication on Lemieux (32 processors). When $MAX\_IF \leq 9$, the Bruck algorithm performs better than the ring and pair algorithms, and all three algorithms perform significantly better than the simple algorithm. However, when the imbalance factor is larger ($16 \leq MAX\_IF \leq 129$), the simple algorithm shows better results. The native algorithm performs much better than all algorithms in the case when $MAX\_IF \leq 129$. When $MAX\_IF = 257$, the native algorithm performs worse than the ring and simple algorithms. These results show that under different process arrival patterns with different worst case imbalance factors, the algorithms have different performance. When the imbalance factor increases, one would expect that the communication time should increase. While this applies to the Bruck, ring, pair, and native algorithms, it is not the case for the simple algorithm: the communication time actually decreases as *MAX_IF* increases when $MAX\_IF \leq 17$. In this cluster, 4 processors share the network interface card. With moderate imbalance in a process arrival pattern, different processors initiate their communications at different times, which reduces the resource contention and improves communication efficiency.

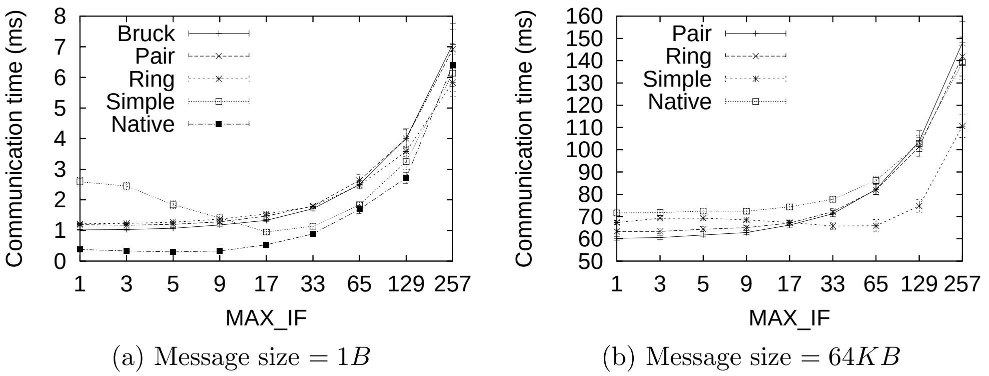
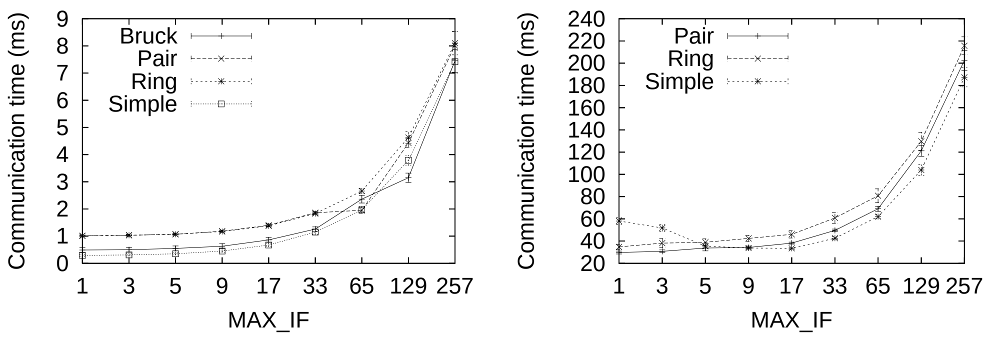


Figure 7: 1B and 64KB *MPI_Alltoall* on Lemieux (32 processors)

Figure 7 (b) shows the performance when the message size is 64KB. When $MAX\_IF \leq 9$, the pair algorithm is noticeably more efficient than the ring algorithm, which in turn is faster than the simple algorithm. However, the simple algorithm offers the best performance when $MAX\_IF \geq 33$. For this message size, the native algorithm performs worse than all three algorithms when $MAX\_IF \leq 65$. The figure also shows that each algorithm performs very differently under process arrival patterns with different imbalance factors. The trend observed in Lemieux is also seen in the Beowulf cluster, which is captured in Figure 8.

Since *MPI_Alltoall* is an inherently synchronized operation, when the imbalance factor is very large, all algorithms should have a similar performance. This is shown in all experiments except for the 64KB case on Lemieux where $MAX\_IF = 257$ is not sufficiently large. However, from the experiments, we can see that algorithms that perform better with a balanced process arrival pattern tend to perform worse when the process arrival pattern



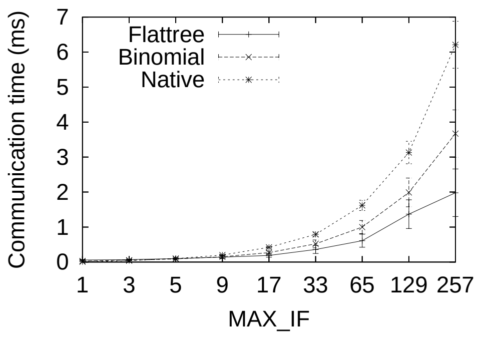
(a) Message size = $1B$ (b) Message size = $64KB$

Figure 8: 1B and 64KB *MPI_Alltoall* on Beowulf

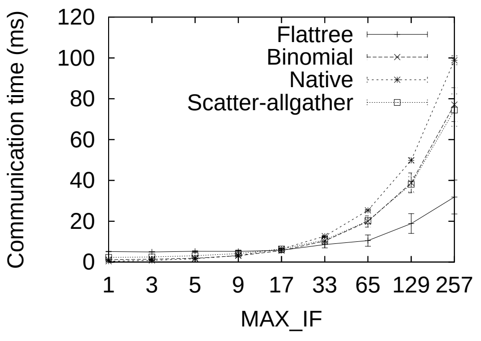
becomes more imbalanced.

Figure 9 (a) shows the results for 1B broadcast on Lemieux (32 processors). When $MAX\_IF \leq 8$, all algorithms perform similarly. When $MAX\_IF > 8$, the flat tree algorithm performs considerably better than the other algorithms. Part (b) of the figure shows the results for broadcasting 64KB messages. When $MAX\_IF < 8$, native, binomial, and scatter-allgather algorithms perform similarly and better than the flat tree algorithm. However, when $MAX\_IF > 16$, the flat tree algorithm performs better than all other algorithms. Moreover, the performance advantage of the flat tree algorithm increases as the imbalance factor increases. Figure 10 shows the results on the Beowulf cluster. The trend for large message sizes is similar. For $1B$ broadcast, the flat-tree algorithm is consistently better for different imbalance factors.

The algorithms for *MPI_Bcast* that perform better under a balanced process arrival pattern perform worse when the arrival pattern becomes imbalanced. In contrast to the results for *MPI_Alltoall*, the performance difference for different broadcast algorithms widens as the imbalance factor increases. Due to the implicit synchronization in *MPI_Alltoall*, there is a limit on the impacts of an imbalanced pattern (all algorithms will have a similar performance when the imbalance factor is very large). However, for the *MPI_Bcast* type of operations that are not inherently synchronized, the impacts can potentially be unlimited.

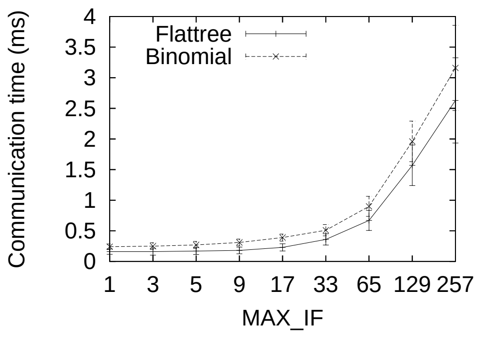


(a) Message size = $1B$

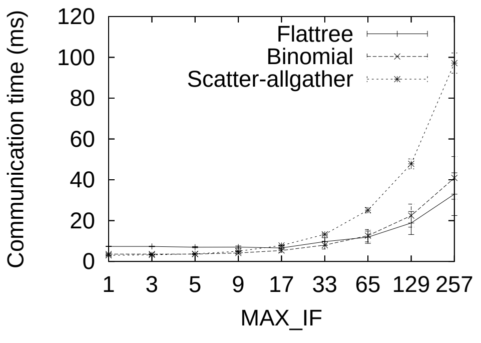


(b) Message size = $64KB$

Figure 9: 1B and 64KB *MPI_Bcast* on Lemieux (32 processors)



(a) Message size = $1B$



(b) Message size = $64KB$

Figure 10: 1B and 64KB *MPI_Bcast* on Beowulf cluster

6.1 Summary

The common observation in the experiments in this section is that collective communication algorithms respond differently to different process arrival patterns. The algorithm that performs better with a balanced process arrival pattern tends to perform worse when the process arrival pattern becomes more imbalanced. Moreover, depending on the type of collective operations, the impact of an imbalanced process arrival pattern can be large.

7 A Potential Solution

Our proposed solution is based on two key observations. First, while the process arrival pattern for a collective operation is nondeterministic, the process arrival patterns for each individual call site tend to exhibit a phased behavior as discussed in Section 4, that is, the process arrival patterns are statistically similar for an extended period of time before they change (See Figure 3). Hence, if the library routine can find an algorithm that can provide good performance, it is likely that the algorithm will provide good performance for an extended period of time. Second, while different collective algorithms may perform best for different process arrival patterns, the performance of a given algorithm changes slowly as the maximum imbalance factor changes, as shown by the small 95% confidence intervals and the smooth curves for all algorithms in Section 6. This indicates that when an algorithm gives the best performance for a particular process arrival pattern, it tends to give a reasonable performance for other process arrival patterns that are not drastically different. Hence, to get a reasonable performance, we do not need to find all best algorithms for different process arrival patterns. Instead, we just need to find *some* best algorithms for *some* representative points in the process arrival pattern space.

These two observations strongly suggest that it might be possible to develop a collective routine that performs well for different process arrival patterns by (1) identifying good algorithms for different process arrival patterns and (2) using a dynamic adaptive mechanism that selects the best performing algorithm at run-time. The STAR-MPI that we developed [8] provides such a dynamic adaptive mechanism. We apply the STAR-MPI idea to develop a robust *MPI_Alltoall* routine by incorporating process arrival pattern aware all-to-all algorithms. This solution takes a coarse-grain approach to achieve high performance with different (balanced and imbalanced) process arrival patterns: the routine maintains a set of all-to-all algorithms that perform well for different process arrival patterns; at runtime, each communication algorithm is probed to decide the best algorithm for a given call site. As shown in the performance evaluation, our routine consistently achieves higher performance for different platforms and applications (different process arrival patterns) than the native MPI implementations. This demonstrates that, by explicitly considering the process arrival pattern, it is possible to develop more efficient MPI collective communication routines than the current ones. Next, we will describe the process arrival pattern aware all-to-all algorithms included in our robust all-to-all routine. Details about the dynamic adaptive mechanism can be found in [8].

7.1 Process arrival pattern aware all-to-all algorithms

To identify good algorithms for different process arrival patterns, we empirically test an extensive set of algorithms that we implemented [7] on different platforms. We will describe the selected algorithms and give rationale about why they provide good performance in different situations.

Pair/Ring algorithms. The *pair* and *ring* algorithms described in Section 6 provide good performance when the process arrival pattern is balanced. In both algorithms, the all-to-all operation is partitioned into $n - 1$ phases with each process performing exactly the same amount of operations in each phase. When the process arrival pattern is perfectly balanced, the phases may be executed in a lock-step manner and the communication is efficient.

While the ring and pair algorithms are efficient when the process arrival pattern is balanced, they do not perform well when the imbalanced factor is larger. In particular, when the worst case imbalanced factor is larger than 1, early arrivals of some processes in the pair/ring algorithms will cause some processes to complete a phase and start the next phase while other processes are still in the previous phase. This may destroy the phase structure, cause system contention, and degrade the performance. This problem can be resolved in two ways, each resulting in a different type of efficient algorithms.

Ring/Pair + one MPI barrier. One solution is to prevent the imbalanced arrival patterns from happening when the ring/pair algorithm is invoked. By adding a barrier operation, this scheme guarantees to have a balanced process arrival pattern for the ring/pair algorithm. The barrier operation forces processes that arrive at the operation early to idle. It provides good performance when the worst case imbalance factor is small, but not 0.

Ring/Pair + light barrier. The *ring/pair + one MPI barrier* algorithm forces processes that arrive at the operation early to idle. This may not be efficient when a large number of processes arrive at the operation significantly earlier than others since processes that arrive early could have used the idle time to perform some useful operations. The *ring/pair+light barrier* is another solution to the problem caused by the imbalanced process arrival patterns. The idea is (1) to allow the phases to proceed in an asynchronous manner and (2) to use a mechanism (light barrier) to minimize the impact of the imbalanced process arrival pattern. Basically, whenever there is a possibility that two messages in different phases can be sent to the same processes at the same time and cause contention, a light barrier that performs pair-wise synchronization is added to sequentialize the two messages. By introducing light barriers, the algorithm sequentializes all messages that can potentially cause contention and the impact of imbalanced process arrival patterns is minimized.

Simple. All of the above algorithms are based on the concept of *phase*, which requires processes to coordinate. In the case when the imbalance factor is large, the coordination among processes may actually hinder the communication performance. The *simple* algorithm, described in Section 6, performs all communications in a single phase (step), eliminating the coordination among processes. As a result, this algorithm performs well for sufficiently imbalanced process arrival patterns.

Besides these algorithms, our routine also includes the native *MPI_Alltoall*, which is selected in the native MPI library for a reason. Hence, there are a total of 8 algorithms that are included in our robust *MPI_Alltoall* routine. As shown in the performance evaluation section, our routine performs better than the native *MPI_Alltoall* in most cases, which indicates that the native *MPI_Alltoall* implementation is not the best performing algorithm among the algorithms in many practical cases. Notice that some of these algorithms, such as *pair/ring* and *simple*, are included in MPICH, where they are used to realize the all-to-all operation with different message sizes. In our routine, all the algorithms can be selected to realize the operation with the same message size, but different process arrival patterns.

7.2 Performance results

We evaluate the performance of the robust *MPI_Alltoall* routine on the following high-end clusters: the Lemieux cluster at Pittsburgh Supercomputing Center [24], the UC/ANL Teragrid cluster at Argonne [33], the AURORA cluster at the University of Technology at Vienna [1], and the AVIDD-T cluster at Indiana University [2]. Table 8 summarizes the configurations of all clusters besides Lemieux, whose configuration is described in Section 4. The benchmarks were compiled with the native *mpicc* or *mpif90* installed on the systems and linked with the native MPI library. We use a micro-benchmark and a set of application benchmarks in the evaluation. In presenting the results, we denote our robust routine as ROBUST and the native routine as NATIVE. The software used in this section, including our robust all-to-all routine and all benchmarks, are available to the public at http://www.cs.fsu.edu/~xyuan/MPI/STAR-ALLTOALL.

Table 8: Clusters used other than Lemieux

cluster	UC-TG [33]	Aurora [1]	Avidd-T [2]
node	two 2.4GHz Xeon	two 3.6GHz Nocona	four 1.3 GHz Itanium II
memory	4GB	4GB	6GB
interconn.	Myrinet	Infiniband	Myrinet
MPI	MPICH-GM 1.2.7	MVAPICH 0.9.5	MPICH-GM 1.2.7

7.3 Micro-benchmark results

The micro-benchmark used in the study is shown in Figure 11. This benchmark can emulate balanced applications with different computation times as well as imbalanced applications with different computation times and different degrees of computation imbalance. There are four components inside the loop. First, a barrier is called in line (3) to synchronize all processes. Then, line (4) is a loop emulating the balanced computation. In this loop, all processes perform the same computation. The duration of this loop is controlled by the parameter $XTIME$. After that, line (5) is an imbalanced loop that artificially introduces an imbalanced computation load. The duration of this loop is controlled by the variable r, which is a random variable with a maximum value of MAX_IF. This loop runs for roughly the time to send r messages of size equal to the message size in *MPI_Alltoall*. Note that r is not the same across all processes: r is a random number (with different seeds in different processors) bounded by the parameter MAX_IF. Thus, the larger the value MAX_IF is, the more imbalanced the process arrival pattern becomes. At the end, the *MPI_Alltoall* routine is called and measured.

In the evaluation, $XTIME$ will be set such that the computations in line (4) last for 50ms, 100ms, 200ms, and 400ms while the MAX_IF takes the values of 1, 10, 50, and 100. As shown in Section 5, even without the imbalanced loop in line (5), the balanced loop in line (4) introduces imbalanced process arrival patterns since different processors take different times to complete the loop. The micro-benchmark runs for 200 iterations. Our proposed all-to-all routine consumes 80 iterations examining different algorithms before landing on

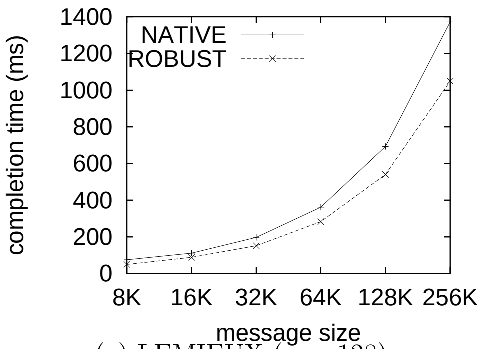
```
(1) r = rand() % $MAX\_IF$;
(2) for (i=0; i<ITER; i++) {
(3)    MPI_Barrier (...);
       /* compute for roughly X milliseconds */
(4)    for (j=0; j< $XTIME$; j++) COMP;
       /* compute for roughly a time equal to sending $r$ messages */
(5)    for (j=0; j<r; j++) COMP_FOR_ONE_MSG_TIME;
(6)    $t_0$ = MPI_Wtime();
(7)    MPI_Alltoall(...);
(8)    elapse += MPI_Wtime() - $t_0$;
(9)}
```

Figure 11: Code segment to measure performance of *MPI_Alltoall*

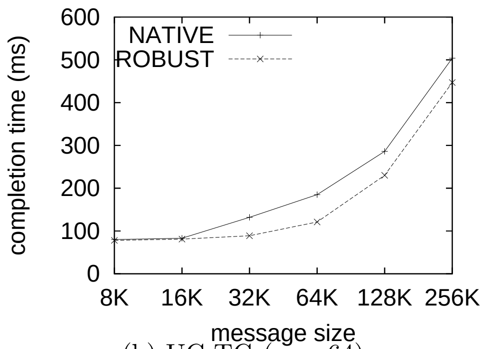
the best algorithm. Since we are more interested in the final algorithm selected and since the algorithm selection overhead is amortized over all invocations in a program, we only compute and report the per invocation communication completion time of *MPI_Alltoall* by averaging the communication completion time of the last 120 iterations.

Parts (a-d) of Figure 12 show the micro-benchmark performance results for using NATIVE and ROBUST *MPI_Alltoall* routines with different message sizes in a perfectly load-balanced micro-benchmark on the different clusters. The results shown are based on using the micro-benchmark with balanced computations (line (4)) set to consume roughly 400ms and no execution of the imbalanced computations (line (5)) as $MAX\_IF = 1$. There are two major observations across the different clusters. First, ROBUST never performs worse than NATIVE. Second, for a wide range of message sizes, the speed-up achieved by ROBUST over NATIVE is significant. For example, when the message size is 64KB, ROBUST improves the performance of the all-to-all operation across all machines with a speed-up of 28% on LEMIEUX, 53% on UC-TG, 24% on AVIDD-T, and 19% on AURORA.

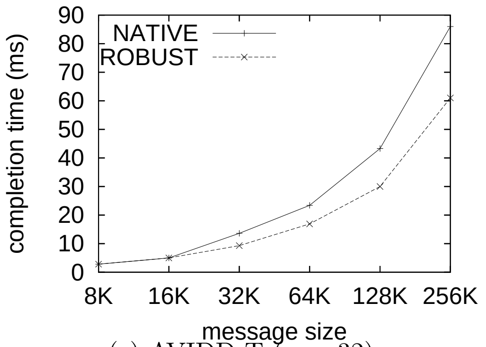
Table 9 shows the results for using NATIVE and ROBUST *MPI_Alltoall* routines with 64KB message size in the micro-benchmark with perfectly load-balanced computations of various computation times (different $XTIME$). From Section 5, the computation imbalance increases as the computation time increases from 50ms to 400ms. ROBUST is able to sustain its substantial speed-ups NATIVE across the different clusters and under different computation loads.



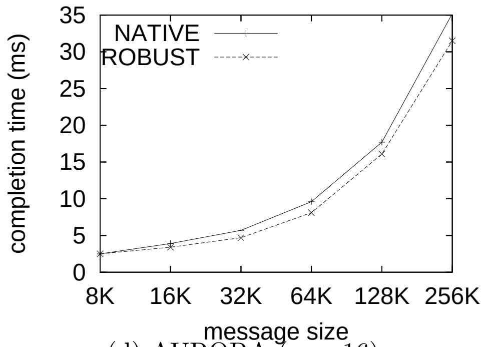
(a) LEMIEUX ($n = 128$)



(b) UC-TG ($n = 64$)



(c) AVIDD-T ($n = 32$)



(d) AURORA ($n = 16$)

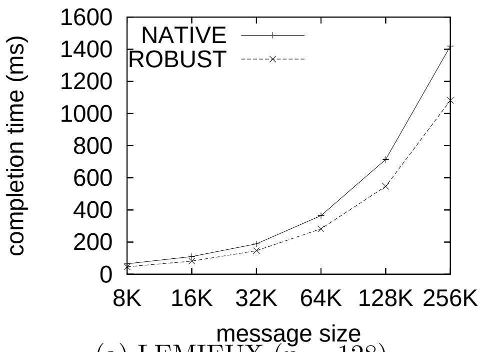
Figure 12: Per-invocation comm. time (400ms balanced computation, $MAX\_IF = 1$)

Parts (a-d) of Figure 13 show the performance results for using NATIVE and ROBUST *MPI_Alltoall* routines with different message sizes under an imbalanced computation load (and thus an imbalanced process arrival pattern). The balanced computations in the benchmark are set to consume roughly 200ms and the imbalanced arrival pattern is generated with MAX_IF set to 100. Using the proposed algorithms in our all-to-all routine for imbalanced arrival patterns, the results shown in Figure 13 demonstrate the robustness of our scheme in achieving significant improvements over the native one. For example, ROBUST speeds up over NATIVE for 64KB messages by 29% on LEMIEUX, 28% on UC-TG, 38% on AVIDD-T, and 19% on AURORA.

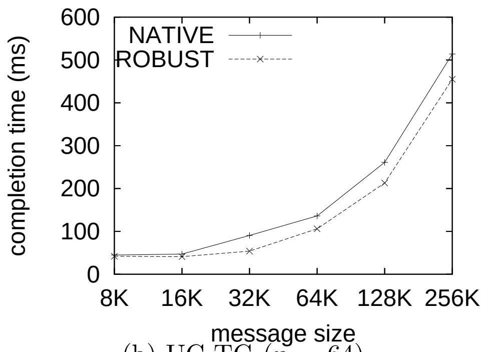
Table 10 shows the performance results for using NATIVE and ROBUST *MPI_Alltoall*

Table 9: Performance of NATIVE and ROBUST MPI_Alltoall of 64KB under different computation times on different machines ($MAX\_IF = 1$)

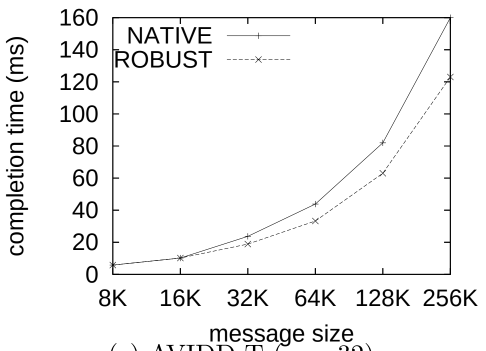
machine	implem-entation	computation time 50	100	200	400
LEMIEUX ($n = 128$)	NATIVE	346ms	348ms	352ms	362ms
	ROBUST	263ms	266ms	273ms	283ms
	speed-up	31.6%	30.8%	28.9%	27.9%
UC-TG ($n = 64$)	NATIVE	117ms	108ms	147ms	185ms
	ROBUST	105ms	93.0ms	122ms	121ms
	speed-up	11.4%	16.1%	20.5%	52.9%
AVIDD-T ($n = 32$)	NATIVE	76.0ms	77.5ms	77.5ms	80.5ms
	ROBUST	63.3ms	64.7ms	66.7ms	64.9ms
	speed-up	20.1%	19.8%	16.2%	24.0%
AURORA ($n = 16$)	NATIVE	8.90ms	9.20ms	9.20ms	9.60ms
	ROBUST	8.20ms	8.10ms	8.50ms	8.10ms
	speed-up	8.50%	13.6%	8.30%	18.5%



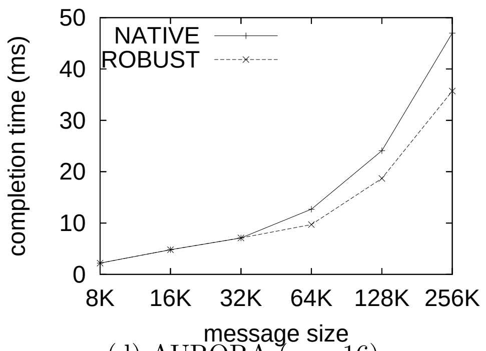
(a) LEMIEUX ($n = 128$)



(b) UC-TG ($n = 64$)



(c) AVIDD-T ($n = 32$)



(d) AURORA ($n = 16$)

Figure 13: Per-invocation comm. time (200ms balanced computation, $MAX\_IF = 100$)

routines with 64KB message size and under different imbalanced process arrival patterns generated by various values of MAX_IF. As shown in the table, whether the imbalance in process arrival patterns is modest ($MAX\_IF \leq 10$) or large ($MAX\_IF > 10$), ROBUST is still able to achieve substantially better performance than NATIVE across the different platforms. For instance, when $MAX\_IF = 10$, ROBUST achieves a speed-up of 30.6% on LEMIEUX, 26.1% on UC-TG, 17.7% on AVIDD-T, and 13.6% on AURORA.

Table 10: Performance of NATIVE and ROBUST MPI_Alltoall of 64KB under different imbalanced arrival patterns on different machines (200ms balanced computation)

machine	implem-entation	MAX_IF			
		1	10	50	100
LEMIEUX ($n = 128$)	NATIVE	353ms	354ms	357ms	365ms
	ROBUST	271ms	271ms	276ms	283ms
	speed-up	30.1%	30.6%	29.4%	29.0%
UC-TG ($n = 64$)	NATIVE	143ms	145ms	139ms	136ms
	ROBUST	115ms	103ms	122ms	106ms
	speed-up	24.4%	26.1%	35.0%	28.3%
AVIDD-T ($n = 32$)	NATIVE	78.2ms	78.5ms	84.0ms	94.0ms
	ROBUST	63.0ms	66.7ms	70.4ms	68.0ms
	speed-up	24.1%	17.7%	19.3%	38.2%
AURORA ($n = 16$)	NATIVE	9.20ms	10.30ms	10.31ms	12.43ms
	ROBUST	8.50ms	8.06ms	8.43ms	9.63ms
	speed-up	8.30%	27.8%	22.3%	29.1%

7.4 Results for application benchmarks

The previous section shows that our routine (ROBUST) achieves high performance in the micro-benchmark with different message sizes, different computation loads, and different degrees of computation imbalance. In this section, we evaluate the performance of ROBUST using four MPI application benchmarks: FT, VH-1, MT, and FFT-2D. The FT (Fast-Fourier Transform) benchmark, a parallel kernel from the NAS parallel benchmarks [20], solves a partial differential equation using forward and inverse FFTs. In the evaluation, the class C problem size is used and the program runs for 400 iterations (or steps). The VH-1 (Virginia Hydrodynamics) [35] is a multidimensional ideal compressible hydrodynamics code based on the Lagrangian remap version of the Piecewise Parabolic Method. The code uses

MPI_Alltoall to perform a transpose on a matrix of size $i \times i$. The value of i is 16K on LEMIEUX, 8K on UC-TG, 4K on AVIDD-T, and 2K on AURORA. The code executes for 500 steps. The FFT-2D [32] program performs a two-dimensional Fast Fourier Transform on a 4K×4K complex matrix. In the evaluation, the code executes for 300 steps. Finally, the MT (Matrix Transpose) [18] is a simple program that uses *MPI_Alltoall* to perform matrix transpositions on a $4K \times 4K$ matrix. The code executes for 300 steps. Table 11 summarizes the message sizes of the *MPI_Alltoall* routine used in the application programs across different clusters. The process arrival patterns in these four benchmarks on different platforms are very different. Hence, the performance of these benchmarks on different platforms gives good indications about the performance of ROBUST in practical situations.

Table 11: Message sizes of *MPI_Alltoall* in application benchmarks on different clusters

program	LEMIEUX ($n = 128$)	UC-TG ($n = 64$)	AVIDD-T ($n = 32$)	AURORA ($n = 16$)
FT	128KB	512KB	2048KB	8192KB
VH-1	320KB	320KB	320KB	320KB
FFT-2D	16KB	64KB	256KB	1024KB
MT	8KB	32KB	128KB	512KB

Table 12 shows the results for application benchmarks with the NATIVE and ROBUST *MPI_Alltoall* routines. For each application, we show the communication time for the *MPI_Alltoall* routine (left column) as well as the total application time (right column), the speed-up by using ROBUST over NATIVE, and the algorithm selected by ROBUST on each platform. We can clearly see that ROBUST significantly improves the communication time over NATIVE across the different applications on different platforms. To illustrate, ROBUST achieves a communication time speed-up of 55.74% and 31.45% for the VH-1 benchmark on LEMIEUX and AVIDD-T clusters, respectively. Similar gains are also seen for other programs; e.g. a speed-up of 31.00% for the FFT-2D benchmark on UC-TG and a speed-up of 45.28% for the FT benchmark on AURORA. For the overall application time, the speed-up depends on several factors, including (1) the percentage of all-to-all time in the total application time, and (2) how the all-to-all operation interacts with computation and other (collective) communications. In particular, the interaction among the all-to-all operation, the computation, and other collective communications can either offset or enhance the performance improvement: the improvement in the communication time may or may not transfer into the improvement in the total application time. For example, for VH-1 on Lemieux, the communication time is improved by 895 seconds, but the improvement in the total application time is only 250 seconds. On the other hand, for VH-1 on UC-TG, the communication time is improved by 175 seconds while the total application time is improved by 212 seconds. Analyzing the program execution traces reveals that while ROBUST minimizes the total communication time among all processes (the sum of the times in all processes), it does not uniformly decrease the communication time in each individual process. Thus, the reduction of the total execution time between two collective routines, which can serve as synchronization points, depends not only on the reduction of the communication time, but also on the amount of computation between the two collective routines and the shape of the process exit pattern of the first collective routine. This is why the reduction in total communication time does not always transfer to the reduction in total application time. Overall, minimizing the total communication time is effective and ROBUST achieves noticeable improvement over NATIVE for both communication time and total benchmark time, which demonstrates the robustness of our implementation. Moreover, we can see from the table that, in many cases, the algorithms used by ROBUST are different across different programs on different platforms. This shows the importance of having multiple process arrival pattern aware algorithms to deal with different applications of different arrival patterns, which our all-to-all routine does.

7.5 Summary

Although the native implementations of the *MPI_Alltoall* routine across the different platforms exploit features of the underlying network architecture, these routines do not perform as good as ROBUST in many cases. This can mainly be attributed to the fact that the native

Table 12: Performance of application benchmarks

program	implem.	LEMIEUX ($n = 128$)		UC-TG ($n = 64$)		A VIDD-T ($n = 32$)		AURORA ($n = 16$)	
		comm.	total	comm.	total	comm.	total	comm.	total
FT	NATIVE	265.0s	501.3s	6182s	10107s	1069s	1720s	616.0s	1690s
	ROBUST	221.0s	450.6s	5917s	9832s	865.0s	1583s	424.0s	1500s
	speed-up	19.9%	11.3%	4.9%	2.8%	23.6%	8.7%	45.3%	12.7%
	algorithm	pair+one barrier		simple		pair+light barrier		pair+light barrier	
VH-1	NATIVE	2495s	3679s	836.0s	5489s	443.0s	1600s	45.50s	457.0s
	ROBUST	1602s	3429s	661.0s	5277s	337.0s	1506s	39.50s	451.0s
	speed-up	55.7%	7.3%	26.5%	3.9%	31.5%	6.2%	15.2%	1.3%
	algorithm	simple		simple		simple		simple	
FFT-2D	NATIVE	178.3s	403.0s	78.60s	594.0s	91.20s	190.5s	47.30s	255.0s
	ROBUST	165.0s	399.0s	60.00s	576.0s	79.80s	180.3s	38.10s	244.0s
	speed-up	8.1%	1.0%	31.0%	3.1%	14.3%	5.7%	24.2%	4.5%
	algorithm	simple		simple		ring+light barrier		simple	
MT	NATIVE	14.46s	15.97s	15.30s	16.50s	44.10s	47.40s	22.50s	27.74s
	ROBUST	12.62s	13.96s	14.70s	16.20s	37.80s	41.10s	21.93s	27.22s
	speed-up	14.5%	14.3%	4.1%	1.9%	16.7%	14.8%	2.6%	1.9%
	algorithm	pair+one barrier		pair+light barrier		pair+light barrier		ring+light barrier	

routines were designed without taking process arrival pattern into consideration. As such, they do not provide high performance for many practical cases. By explicitly considering process arrival pattern and employing a dynamic adaptive technique, more robust collective routines than the current ones can be developed.

8 Conclusion

In this paper, we investigate the process arrival patterns in a set of MPI benchmarks on two representative cluster platforms. We show that in such environments, it is virtually impossible for application developers to control process arrival patterns in their applications without explicitly invoking global synchronization operations and that process arrival patterns are likely to be imbalanced. Since the process arrival pattern has a significant impact on the performance of collective communication algorithms, we conclude that MPI developers must take the process arrival pattern characteristics into consideration when developing MPI collective communication routines that can provide high performance in practical clusters. This study advocates further investigation for understanding the impact of process arrival patterns on different MPI collective operations and different collective communication algorithms and for identifying efficient process arrival pattern aware algorithms. The current understanding of MPI collective algorithms, which assumes a balanced process arrival pattern, is insufficient for developing routines that are efficient in practice. We demonstrate that when process arrival pattern aware algorithms are identified, a dynamic adaptive scheme can be used to implement robust collective routines that provide high performance across different applications and platforms.

Acknowledgment

This research is supported in part by National Science Foundation (NSF) grants CCF-0342540, CCF-0541096, and CCF-0551555. The experiments on the Lemieux cluster at Pittsburgh Supercomputing Center are sponsored through an NSF Teragrid grant CCR-050010T.

References

[1] The AURORA cluster of University of Technology at Vienna, http://aurora.tuwien.ac.at.

[2] The AVIDD-T cluster, http://rac.uits.iu.edu/rats/research/avidd-t/hardware.shtml.

[3] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, "Pipelined Broadcasts on Heterogeneous Platforms." *IEEE Trans. on Parallel and Distributed Systems*, 16(4):300-313, 2005.

[4] J. Bruck, C. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient Algorithms for All-to-all Communications in Multiport Message-Passing Systems." *IEEE Trans. on Parallel and Distributed Systems*, 8(11):1143-1156, Nov. 1997.

[5] W. E. Cohen and B. A. Mahafzah, "Statistical Analysis of Message Passing Programs to Guide Computer Design," In *Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences*, volume 7, pages 544-553, 1998.

[6] A. Faraj and X. Yuan, "Communication Characteristics in the NAS Parallel Benchmarks," In *Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, pages 724-729, November 2002.

[7] A. Faraj and X. Yuan, "Automatic Generation and Tuning of MPI Collective Communication Routines," The *19th ACM International Conference on Supercomputing* (ICS'05), pages 393-402, Cambridge, MA, June 20-22, 2005.

[8] A. Faraj, X. Yuan, and D. K. Lowenthal, "STAR-MPI: Self Tuned Adaptive Routines for MPI Collective Operations," *The 20th ACM International Conference on Supercomputing (ICS'06)*, pages 199-208, Queensland, Australia, June 28-July 1, 2006.

[9] A. Faraj, X. Yuan, and Pitch Patarasuk, "A Message Scheduling Scheme for All-to-all Personalized Communication on Ethernet Switched Clusters," *IEEE Transactions on Parallel and Distributed Systems*, 18(2):264-276, Feb. 2007.

[10] A. Faraj, P. Patarasuk, and X. Yuan, "Bandwidth Efficient All-to-all Broadcast on Switched Clusters," *International Journal of Parallel Programming*, accepted.

[11] J.Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. Dongarra, "Performance Analysis of MPI Collective Operations," *IEEE IPDPS*, CDROM proceedings, 2005.

[12] A. Karwande, X. Yuan, and D. K. Lowenthal, "An MPI Prototype for Compiled Communication on Ethernet Switched Clusters," *Journal of Parallel and Distributed Computing* 65(10):1123-1133, October 2005.

[13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, 22(6):789-828, 1996.

[14] D. Lahaut and C. Germain, "Static Communications in Parallel Scientific Propgrams," In *Proceedings of the 6th International PARLE Conference on Parallel Architectures and Languages*, pages 262-276, 1994.

[15] LAM/MPI Parallel Computing. http://www.lam-mpi.org.

[16] LAMMPS: Molecular Dynamics Simulator, Available at http://www.cs.sandia.gov/ sjplimp/lammps.html.

[17] A. Mamidala, J. Liu, D. Panda. "Efficient Barrier and Allreduce on InfiniBand Clusters using Hardware Multicast and Adaptive Algorithms." *IEEE International Conference on Cluster Computing*, pages 135-144, Sept. 2004.

[18] Matrix Transposition example, http://www.sara.nl/userinfo/reservoir/mpi/mpi-intro.

[19] MPICH - A Portable Implementation of MPI, Available at http://www.mcs.anl.gov/mpi/mpich.

[20] NASA Parallel Benchmarks, Available at http://www.nas.nasa.gov/NAS/NPB.

[21] Parallel NBody Simulations, Available at http://www.cs.cmu.edu/ scandal/alg/nbody.html.

[22] ParaDyn: Parallel Molecular Dynamics With the Embedded Atom Method, Available at http://www.cs.sandia.gov/ sjplimp/download.html.

[23] F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8192 Processors of ASCI Q", *IEEE/ACM SC2003 Conference*, 2003.

[24] Pittsburg Supercomputing Center, Available at http://www.psc.edu/machines/tcs/lemieux.html.

[25] R. Rabenseinfner, "Automatic MPI counter profiling of all users: First results on CRAY T3E900-512," In *Proceedings of the Message Passing Interface Developer's and User's Conference*, pages 77-85, 1999.

[26] I. Rosenblum, J. Adler, and S. Brandon, "Multi-processor molecular dynamics using the Brenner potential: Parallelization of an implicit multi-body potential," *International Journal of Modern Physics*, 10(1): 189-203, February, 1999.

[27] P. Patarasuk, A. Faraj, and X. Yuan, "Pipelined Broadcast on Ethernet Switched Clusters." The *20th IEEE International Parallel & Distributed Processing Symposium* (IPDPS), CDROM proceedings, April 25-29, 2006.

[28] P. Sanders and J.F. Sibeyn, "A Bandwidth Latency Tradeoff for Broadcast and Reduction." *Information Processing Letters*, 86(1):33-38, 2003.

[29] T.B. Tabe, J.P. Hardwick, and Q.F. Stout, "Statistical analysis of communication time on the IBM SP2," *Computing Science and Statistics*, 27: 347-351, 1995.

[30] T. Tabe and Q. Stout, "The use of the MPI communication library in the NAS Parallel Benchmark," *Technical Report CSE-TR-386-99*, Department of Computer Science, University of Michigan, Nov 1999.

[31] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimizing of Collective Communication Operations in MPICH," *International Journal of High Performance Computing Applications,* 19(1):49-66, Spring 2005.

[32] Two-D FFT, http://www.mhpcc.edu/training/workshop/parallel_develop.

[33] The UC/ANL Teragrid cluster, http://www.uc.teragrid.org/tg-docs/user-guide.html.

[34] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Automatically Tuned Collective Communications," In *Proceedings of SC'00: High Performance Networking and Computing* (CDROM proceeding), 2000.

[35] The Virginia Numerical Bull Session Ideal Hydrodynamics, http://wonka.physics.ncsu.edu/pub/VH-1.

[36] J. S. Vetter and A. Yoo, "An empirical performance evaluation of scalable scientific applications," In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1-18, 2002.

[37] X. Yuan, R. Melhem and R. Gupta, "Algorithms for Supporting Compiled Communication," *IEEE Transactions on Parallel and Distributed Systems*, 14(2):107-118, February 2003.