The Use of the MPI Communication Library in the NAS Parallel Benchmarks

Theodore B. Tabe and Quentin F. Stout

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109
{*tabe*, *qstout*}@eecs.umich.edu

Abstract

The statistical analysis of traces taken from the NAS Parallel Benchmarks can tell one much about the type of network traffic that can be expected from scientific applications run on distributed memory parallel computers. For instance, such applications utilize a relatively few number of communication library functions, the length of their messages is widely varying, they use many more short messages than long ones, and within a single application the messages tend to follow relatively simple patterns. Hardware and software designers can use information such as this to optimize their systems for the highest possible performance. This paper presents specific data on how these generally known characteristics about distributed memory applications are exhibited in the NAS Parallel Benchmarks.

1 Introduction

Parallel computing is a computer paradigm where multiple processors attempt to co-operate in the completion of a single task. Within the parallel computing paradigm, there are two memory models: shared-memory and distributed memory. The shared-memory model distinguishes itself by presenting the programmer with the illusion of a single memory space. The distributed-memory model, on the other hand, presents the programmer with a separate memory space for each processor. Processors, therefore, have to share information by sending messages to each other. To send these messages, usually applications call a standard communication library. The communication library is usually MPI (Message Passing Interface) [7] or PVM (Parallel Virtual Machine) [2], with MPI rapidly becoming the norm.

An important component in the performance of a distributed-memory parallel computing application is the performance of the communication library the application uses. Therefore, the hardware and software systems providing these communication functions must be tuned to the highest degree possible. An important class of information that would aid in the tuning of a communication library is an understanding of the communication patterns that occur within applications. This includes information such as the relative frequency with which the various functions within the communication library are called, the lengths of the messages involved, and the ordering of the messages.

Since it is not realistic to examine all the distributed-memory parallel applications in existence, one looks to find a small set of applications that reasonably represents the entire field. The representative set of applications that was chosen was the widely used NAS Parallel Benchmarks (NPB) [1]. The rest of this paper describes in further detail the NPB and the results obtained from analyzing the frequency and type

of message calls which occur within the NPB. This area of research is not new (see Section 6). However, the NAS Parallel Benchmarks have not been quantified in this manner before.

Section 2 of the paper describes the NPB. Section 3 describes the instrumentation methodology used on the NPB. Following that is Section 4, which describes the assumptions made about the manner in which the MPI message-passing library was implemented. Section 5 gives a summary of the data gathered from the traces. Section 6 discusses related work on the NAS Parallel Benchmarks and on the characterization of the communication patterns of distributed memory parallel applications. Section 7 provides an explanation for the patterns observed, in terms of the nature of the communication patterns of the NPB. Section 8 provides some final conclusions.

2 NAS Parallel Benchmarks Description

The NAS Parallel Benchmarks are a set of scientific benchmarks issued by the Numerical Aerodynamic Simulation (NAS) program located at the NASA Ames Research Center. The benchmarks have become widely accepted as a reliable indicator of supercomputer performance on scientific applications. The benchmarks are largely derived from computational fluid dynamics code and are currently on version 2.2. The NAS Parallel Benchmarks 2.2 includes implementations of 7 of the 8 benchmarks in the NAS Parallel Benchmarks 1.0 suite. The eighth benchmark shall be implemented in a later version of the NAS Parallel Benchmarks. The benchmarks implemented are:

BT: a block tridiagonal matrix solver.

EP: Embarrassingly Parallel, an application where there is very minimal communication amongst the processes

FT: a 3-D FFT PDE solver benchmark.

IS: integer sort

LU: an LU solver.

MG: a multigrid benchmark.

SP: a pentadiagonal matrix solver.

The benchmark codes are written in Fortran with MPI function calls, except for the IS benchmark which is written in C with MPI function calls. The NAS Parallel Benchmarks can be compiled into three problem sizes known as classes A, B, and C. The class A benchmarks are tailored to run on moderately powerful workstations. Class B benchmarks are meant to run on high-end workstations or small parallel systems. Class C benchmarks are meant for high-end supercomputing.

3 Trace Gathering

3.1 Instrumenting the Benchmarks

The source code for the NPB was instrumented by preprocessing the source code with a filter written in Perl. Since all MPI function calls have the form:

```
call MPI_function_name(parameter1,parameter2,...)
```

the Perl preprocessor simply uses pattern matching to find the MPI function calls in the program and inserts code just before the MPI function call to print out the relevant parameters of the MPI function call. The static frequency of MPI function calls was determined by using the **grep** utility to search for "MPI_" in all source files.

3.2 Running the Benchmarks

The benchmarks were compiled as class B benchmarks and traced on machines at the University of Michigan's Center for Parallel Computing. When the dynamic frequencies of the MPI function calls were calculated from the traces, the **MPI_COMM_*** functions and the **MPI_WTIME** function were not included because the beforementioned functions are not inherently involved in the transfer of data between processors. Also, many of the benchmarks run an iteration of the code before the running and timing of the main loop. This is done to minimize variations in performance that would be caused by cache misses, TLB misses, and page faults. The portions of the traces that correspond to this extra iteration were not included in the dynamic results due to the fact that we wished the results to be as close as possible to those which would be found if the benchmarks were used in real-world situations.

The BT and SP benchmarks were run with 4, 9, and 16 processors. The FT, IS, LU, and MG benchmarks were run with 2, 4, 8, and 16 processors. As an example of the type of information gathered by the traces, for an MPI_SEND function call, the following information is output to stdout:

1. MPI_SEND
2. the ID of the sending process
3. number of data items being transferred
4. datatype of the transferred data (i.e. INTEGER, DOUBLE, or DOUBLE_COMPLEX)
5. the ID of the receiving process
6. the type tag for this interprocess communication

4 MPI Implementation Model

Given the fact that the low-level implementation of the MPI library is platform and vendor dependent, there are then widely varying answers to the question of how many messages it takes to implement any given MPI function. Therefore, for this paper, we created a limited MPI implementation model that we feel reasonably represents how the MPI library functions found within the NAS Parallel Benchmark 2.2 could be implemented. Then, for each function call, we determined how many messages the processor with pid equal to one would send. The processor with pid equal to one was chosen as the one to model over the more typical processor zero is because many times the processor with pid equal to zero is used to broadcast information, causing its call frequency to be unrepresentative of that of a typical node.

Our model is as follows:

MPI_ALLREDUCE The MPI_ALLREDUCE function call is modeled as a reduction followed by a broadcast. The reduction and broadcast are each modeled as occurring via binary trees. In the binary tree

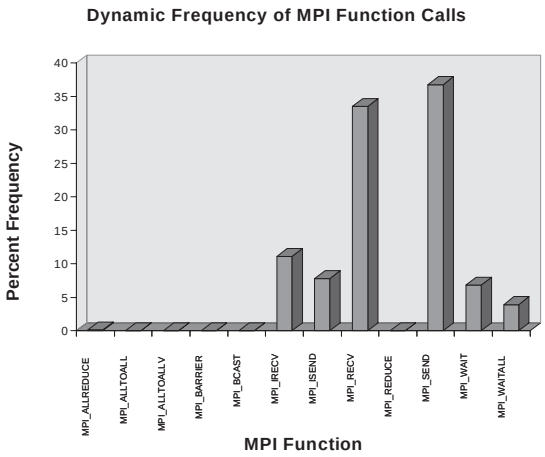

Figure 1: Dynamic Frequency of MPI Function Calls in the NAS Benchmarks

model, one message travels over each edge. Also, in a binary tree of p vertices, there are $p-1$ edges. Therefore, $(p-1) \times 2$ total messages are sent globally for each MPI_ALLREDUCE function call. In this model, processor one sends two messages.

MPI_ALLTOALL The straightforward method of having each node send out $p-1$ messages is the model used for the MPI_ALLTOALL function call. Globally, this implies that $p(p-1)$ messages are sent for each MPI_ALLTOALL function call. For processor one, the model implies $p-1$ messages.

MPI_ALLTOALLV It is modeled in the same manner as the MPI_ALLTOALL function call is modeled.

MPI_BARRIER It is treated as an MPI_ALLREDUCE where the message length is 4 bytes.

MPI_BCAST It is modeled as occurring via a binary tree-based algorithm. Therefore, $p-1$ messages are sent globally. Processor one sends one message.

MPI_IRECV Since a message is received, no messages are sent by processor one.

MPI_ISEND Processor one sends one message.

MPI_RECV Since a message is received, no messages are sent by processor one.

MPI_REDUCE The model assumes a binary tree-based algorithm, implying $p-1$ total messages globally. Processor one sends one message.

MPI_SEND Processor one sends one message.

5 Results

A plethora of conclusions can be drawn from the trace data:

MPI Function	Percent Frequency
MPI_IRECV	14.4%
MPI_SEND	10.6%
MPI_ISEND	10.2%
MPI_BCAST	9.7%
MPI_WAIT	9.7%
MPI_ALLREDUCE	7.2%
MPI_BARRIER	7.2%
MPI_ABORT	4.7%
MPI_COMM_SIZE	4.2%
MPI_WAITALL	3.4%
MPI_FINALIZE	3.0%
MPI_COMM_RANK	2.5%
MPI_INIT	2.5%
MPI_REDUCE	2.5%
MPI_ALLTOALL	1.7%
MPI_COMM_DUP	1.7%
MPI_COMM_SPLIT	1.7%
MPI_RECV	1.7%
MPI_WTIME	0.8%
MPI_ALLTOALLV	0.4%

Table 1: Static Frequency of MPI Function Calls in the NAS Parallel Benchmarks 2.2

- One of the most interesting results reached by examining the traces was that relatively few of the functions in the MPI library are used by the NAS Parallel Benchmarks. Table 1 shows the static frequency in percent of MPI function calls in the NAS Parallel Benchmarks 2.2. Of the 125 functions in the MPI communication library, only 20 were actually used in the NAS Parallel Benchmarks. The functions used are:

  1. MPI_ABORT (abort from MPI)
  2. MPI_ALLREDUCE (reduction plus a broadcast)
  3. MPI_ALLTOALL (all-to-all communication where all the messages are the same length)
  4. MPI_ALLTOALLV (all-to-all communication where the messages can be of different lengths)
  5. MPI_BARRIER (barrier synch)
  6. MPI_BCAST (broadcast)
  7. MPI_COMM_DUP (duplicate a communication group pointer)
  8. MPI_COMM_RANK (find processor id)
  9. MPI_COMM_SIZE (find number of processors in group)
  10. MPI_COMM_SPLIT (split current communication group into two groups)
  11. MPI_FINALIZE (shut MPI down cleanly)

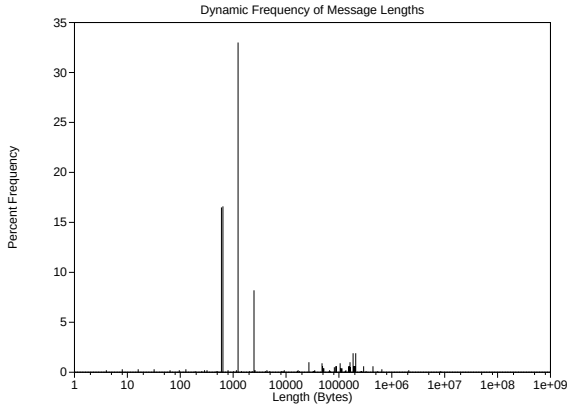

Figure 2: Length of MPI Messages in the NAS Parallel Benchmarks

	Static	**Dynamic**
MPI_ALLREDUCE	17	268
MPI_ALLTOALL	4	128
MPI_ALLTOALLV	1	32
MPI_BARRIER	17	4
MPI_BCAST	23	86
MPI_IRECV	34	45,604
MPI_ISEND	24	32,436
MPI_RECV	4	139,500
MPI_REDUCE	6	98
MPI_SEND	25	152,628
MPI_WAIT	23	27,568
MPI_WAITALL	8	16,206

Table 2: Dynamic versus Static MPI Function Calls

12. MPI_INIT (initialize MPI)
13. MPI_IRECV (non-blocking receive)
14. MPI_ISEND (non-blocking send)
15. MPI_RECV (blocking receive)
16. MPI_REDUCE (reduction)
17. MPI_SEND (blocking send)
18. MPI_WAIT (wait for a non-blocking communication to complete)
19. MPI_WAITALL (wait for a list of non-blocking communications to complete)
20. MPI_WTIME (system time)

This phenomena is not unique to the NPB, as others have noted that relatively few MPI commands are needed for most programs. For example, in the tutorial by Gropp [4], he states "MPI is small (6 functions) — many parallel programs can be written with just 6 basic functions."

MPI Function	IS Percent Frequency	non-IS Percent Frequency
MPI_ALLREDUCE	30.5%	0.1%
MPI_ALLTOALL	30.5%	0.0%
MPI_ALLTOALLV	24.4%	0.0%
MPI_BARRIER	0.0%	0.0%
MPI_BCAST	0.0%	0.0%
MPI_IRECV	3.1%	11.0%
MPI_ISEND	0.0%	7.8%
MPI_RECV	0.0%	33.7%
MPI_REDUCE	6.1%	0.0%
MPI_SEND	2.3%	36.8%
MPI_WAIT	3.1%	6.7%
MPI_WAITALL	0.0%	3.9%

Table 3: Dynamic Frequency of MPI Function Calls in the IS Benchmark vs. the Other Benchmarks

- When looking at the dynamic frequency of the MPI functions in the NAS Parallel Benchmarks, as shown in Figure 1, another interesting result is noted. Fully 89% of the MPI functions calls are blocking or non-blocking sends and receives. None of the more complex MPI communication functions nor even more complex sends or receives such as buffered sends and buffered receives are used.

- Another interesting result is shown in Figure 2. It shows that 74.8% of the messages have a message length of 600, 640, 1240, or 2480 bytes, implying that short messages dominate NPB network traffic. However, the mean message length is 73,447 bytes, the median message length is 1240 bytes, and the standard deviation is a rather large 1,594,623 bytes. Thus there was wide variation in the message lengths. The largest message length is 128 MB when two nodes exchange 128 MB messages using MPI_ALLTOALL in the 2-processor version of the FT benchmark. The shortest message length is 4 bytes which is mainly used by **MPI_BCAST** synchronization messages.

  Overall, the conclusion that can be drawn is that interprocessor communication hardware and software must be optimized for both short and long messages. It is the norm now for parallel machines to realize their full bandwidth for extremely long messages. These traces show that a large percentage of messages are not extremely long. These short messages should also be experiencing the full bandwidth of the interconnect. If this occurred, then many applications would notice increased performance from their distributed environment. Note, however, that "short" is not a few bytes, but hundreds of bytes.

- The IS benchmark is the only NAS parallel benchmark which performs computations on primarily integer datatypes. The others operate primarily on floating point datatypes. The inclusion of the IS benchmark within the NAS Parallel Benchmarks implies that the authors of the NPB felt that the IS benchmark is representative of distributed memory parallel computer applications that operate primarily on integer datatypes. Therefore, contrasting the IS benchmark against the other NAS Parallel Benchmarks may indicate some differences between applications which operate primarily on integer datatypes versus those which operate primarily on floating point datatypes. Table 3 indicates that the IS code is dominated by reductions and all-to-all communication as opposed to the floating point code which is dominated

	IS	non-IS
MPI_ALLREDUCE	40	228
MPI_ALLTOALL	40	88
MPI_ALLTOALLV	32	0
MPI_BARRIER	0	4
MPI_BCAST	0	86
MPI_IRECV	0	45,600
MPI_ISEND	0	32,436
MPI_RECV	0	139,500
MPI_REDUCE	8	90
MPI_SEND	3	152,625
MPI_WAIT	4	27,564
MPI_WAITALL	0	16,206

Table 4: Absolute Count of MPI Function Calls in the IS Benchmark vs. the Other Benchmarks

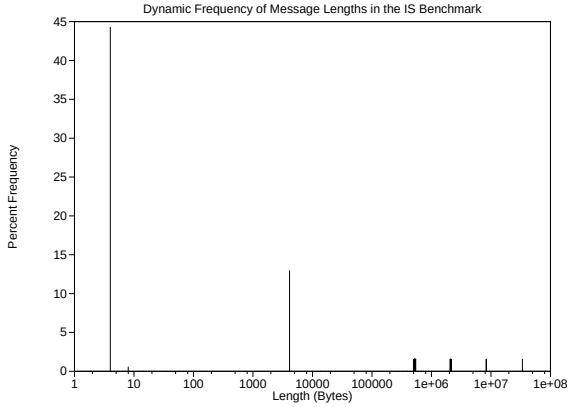

Figure 3: Length of MPI Messages in the IS Benchmark

by sends and receives. Figure 3 shows the distribution of message lengths in the IS benchmark and Table 5 compares the message length statistics of the IS benchmark versus the rest of the NAS Parallel Benchmarks. It shows that the mean of the IS benchmark is two orders of magnitude larger than the mean of the rest of the NPB, and the standard deviation of the IS benchmark is three times larger than the standard deviation of the rest of the NPB. Thus the IS benchmark has a wider variation in message lengths than the rest of the NPB, a fact which may be indicative of some differences between data-intensive and computation-intensive applications. However, it should not be taken to be indicative of general differences between codes dominated by integer operations versus floating point operations.

- Just from comparing the total number of dynamic versus static MPI function calls, as shown in Table 2, one can immediately conclude that some of the function calls in the NAS application codes are being visited a large number of times. This would lead one to believe that the communication patterns of the NPB are dominated by their behavior within loops. Section 7 explores this issue further.

  The reason that the number of dynamic MPI_BARRIER function calls is less than the number of static

	IS Benchmark (bytes)	non-IS Benchmarks (bytes)
Mean	1,332,196	69,317
Median	4116	1240
Minimum	4	4
Maximum	33,716,496	134,217,728
Standard Deviation	4,568,524	1,574,036

Table 5: Message Length Statistics for the IS Benchmark vs. the Rest of the NAS Parallel Benchmarks 2.2

MPI_BARRIER function calls in Table 2 is because we did not in the dynamic frequency count include MPI function calls that were included specifically to aid in the timing of the NAS Parallel Benchmarks. The reason for this is that we wanted the dynamic function call counts to be as representative as possible of the counts one would find when using the applications in a production environment. Those added MPI function calls were MPI_BARRIER calls. However, the MPI_BARRIER function calls were included in the static MPI function call count.

The eleven function call difference in the total static count between (non-blocking and blocking) sends and (non-blocking and blocking) receives (see Table 2) is entirely due to the MG benchmark which contains one MPI_IRECV call and twelve MPI_SEND calls. In all the other benchmarks, the number of static non-blocking and blocking sends equals the number of static non-blocking and blocking receives. This example also illustrates a simple fact about the MPI communication library: a blocking send can transmit data to a non-blocking receive and the opposite can also be true. This leads to the fact that there is not an equivalence, for example, between the dynamic frequency of blocking sends and the dynamic frequency of blocking receives. The IS, LU, and MG benchmarks mix the use of blocking and non-blocking MPI function calls. The other benchmarks do not.

6 Related Work

The NAS Parallel Benchmarks have been extensively analyzed. Strohmaier [10] presented a methodology for characterizing the performance of the NAS Parallel Benchmarks without a detailed knowledge of the source code. White *et al.* [11] measured the performance and communication bandwidth needed to run the NAS Parallel Benchmarks using PVM as the communication library. Simon and Stohmaier [9] used Amdahl's Law to analyze the performance of the NAS Parallel Benchmarks.

Many studies have been done that characterize the communication patterns of distributed memory parallel computers. Cypher *et al.* [3] characterized the amount and type of communication in non-NPB distributed memory codes. They also obtained results indicating that there is a wide variation in the size of messages. Kim and Lilja [5] analyzed some benchmarks to using some locality metrics. They established that most nodes only communicate with a few other nodes using only a few message sizes. O'Hallaron and Shewchuk [8] analyzed the communication characteristics of a family of unstructured 3D finite element simulations and also found wide variations in the message sizes.

7 NAS Parallel Benchmark Communication Kernels

Within the NAS Parallel Benchmarks there is a very simple structure to the communication that occurs between nodes. The Appendix contains pseudocode delineating the communication structure of the NPB codes. As the Appendix shows, the communication is dominated by a loop-based repetition of a few simple communication calls.

Tables 6, 7 and 8 are another view of this simple communication structure. It contains equations that describe the relationship between the frequency of an MPI function being called and the number of processors a benchmark is run on for the various NAS Parallel Benchmarks.

The loop-based patterns mean that the communication structure is static for long periods of time, leading one to believe that simple strategies can detect the pattern. Most likely, there are hardware and software optimizations that can be used to take advantage of these communication patterns. This is an ongoing area of research for us.

8 Conclusion

The fact that only the knowledge of a few MPI functions are really necessary in order to create applications is emphasized to programmers even as early as the first MPI tutorial [6]. This paper, however, goes one step further and quantitatively describes which MPI functions are important. We considered both their static frequency, i.e., how often they were written, and their dynamic frequency, i.e., how often they were executed.

Our statistical analysis of traces taken from the NAS Parallel Benchmarks can tell one much about the type of network traffic to be expected from parallel distributed-memory scientific applications. For instance, these applications will utilize a relatively few number of communication library functions, and the length of these messages will be widely varying. Further, for an application, the majority of the messages are issued within loops having a simple communication pattern.

Since communications is a critical component of distributed-memory parallel computing, it is important that it be carefully optimized. Hardware and software designers to tune their communications systems to increase the performance of real applications can use studies such as those in this paper. This in turn should enable users to achieve higher performance and increased scalability of their codes.

Acknowledgments

Computing services were provided by the University of Michigan's Center for Parallel Computing.

References

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[2] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, "A User's Guide to PVM (Parallel Virtual Machine)," Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, Oak Ridge, TN, July 1991.

[3] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina, "Architectural Requirements of Parallel Scientific Applications with Explicit Communication," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 2–13, 1993.

[4] W. Gropp. *Tutorial on MPI: The Message-Passing Interface.* http://www.mcs.anl.gov/mpi/tutorial/gropp/talk.html #Node0.

[5] J. Kim and D. J. Lilja, "Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs," Technical Report HPPC-97-10, University of Minnesota, October 1997.

[6] Maui High Performance Computing Center. *MPI SP Parallel Computing Workshop.* http://www.mhpcc.edu/ training/workshop/html/mpi/MPIIntro.html.

[7] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," Technical report, University of Tennessee at Knoxville, May 1994.

[8] D. R. O'Hallaron and J. R. Shewchuk, "Properties of a Family of Parallel Finite Element Simulations," Technical Report CMU-CS-96-141, Carnegie Mellon University, December 1996.

[9] H. D. Simon and E. Strohmaier, "Amdahl's Law and the Statistical Content of the NAS Parallel Benchmarks," *Supercomputer*, vol. 11, no. 4, pp. 75–88, September 1995.

[10] E. Strohmaier, "Statistical Performance Modeling: Case Study of the NPB 2.1 Results," Technical Report UTK-CS-97-354, University of Tennessee at Knoxville, March 1997.

[11] S. White, A. Ålund, and V. S. Sunderam, "Performance of the NAS Parallel Benchmarks on PVM Based Networks," *Journal of Parallel and Distributed Computing*, vol. 26, no. 1, pp. 61–71, April 1995.

MPI Function	**BT**	**FT**	**IS**
MPI_ALLREDUCE	2	0	10
MPI_ALLTOALL	0	22	10
MPI_ALLTOALLV	0	0	if $(processors = 2)$ then 5 if $(processors = 4)$ then 8 if $(processors = 8)$ then 9 if $(processors = 16)$ then 10
MPI_BARRIER	0	0	0
MPI_BCAST	3	2	0
MPI_IRECV	$1200\sqrt{processors} + 6$	0	0
MPI_ISEND	$1200\sqrt{processors} + 6$	0	0
MPI_RECV	0	0	0
MPI_REDUCE	1	20	2
MPI_SEND	0	0	if $(processors = 2)$ then 0 if $(processors > 2)$ then 1
MPI_WAIT	$2400\sqrt{processors} - 2400$	0	0
MPI_WAITALL	201	0	0

Table 6: Expressions Describing the MPI Dynamic Function Call Frequency in the NAS Parallel Benchmarks 2.2 as a Function of the Number of Processors, Part 1

MPI Function	**LU**
MPI_ALLREDUCE	8
MPI_ALLTOALL	0
MPI_ALLTOALLV	0
MPI_BARRIER	0
MPI_BCAST	9
MPI_IRECV	if $(processors = 2)$ then 252 if $(processors = 4)$ then 506 if $(processors = 8)$ then 759 if $(processors = 16)$ then 759
MPI_ISEND	0
MPI_RECV	if $(processors < 16)$ then $15500 \log_2 (processors)$ if $(processors = 16)$ then 46500
MPI_REDUCE	0
MPI_SEND	if $(processors = 2)$ then 15755 if $(processors = 4)$ then 31506 if $(processors = 8)$ then 47258 if $(processors = 16)$ then 47258
MPI_WAIT	if $(processors = 2)$ then 252 if $(processors = 4)$ then 506 if $(processors = 8)$ then 759 if $(processors = 16)$ then 759
MPI_WAITALL	0

Table 7: Expressions Describing the MPI Dynamic Function Call Frequency in the NAS Parallel Benchmarks 2.2 as a Function of the Number of Processors, Part 2

MPI Function	**MG**	**SP**
MPI_ALLREDUCE	46	2
MPI_ALLTOALL	0	0
MPI_ALLTOALLV	0	0
MPI_BARRIER	1	0
MPI_BCAST	6	3
MPI_IRECV	if $(processors < 16)$ then 2772 if $(processors = 16)$ then 2572	$2400\sqrt{processors} + 6$
MPI_ISEND	0	$2400\sqrt{processors} + 6$
MPI_RECV	0	0
MPI_REDUCE	1	1
MPI_SEND	if $(processors < 16)$ then 2772 if $(processors = 16)$ then 2532	0
MPI_WAIT	if $(processors < 16)$ then 2772 if $(processors = 16)$ then 2572	0
MPI_WAITALL	0	$2400\sqrt{processors} - 1999$

Table 8: Expressions Describing the MPI Dynamic Function Call Frequency in the NAS Parallel Benchmarks 2.2 as a Function of the Number of Processors, Part 3

This appendix contains the pseudocode which exposes the structure of interprocessor communication within the various NAS Parallel Benchmarks. The pseudocode was formulated from the gathered communication traces and is, therefore, from the viewpoint of the processor with its id equal to 1 running a class B compilation of the benchmark. The pseudocode contains the variable **number_of_processors** which represents the total number of processors utilized during an execution of the application. Within the pseudocode, the variable **pid** is a number between 0 and **number_of_processors**-1. Each processor's **pid** variable is assigned a unique value within the beforementioned range. Furthermore, all reduction operations have the processor with pid equal to 0 as the root.

The MPI function calls within the pseudocode have been abbreviated for simplicity. A short description of the meaning of the fields within the pseudocode MPI function calls is, therefore, necessary:

1. **mpi_allreduce**(*number of items*, *datatype of items*, *reduction operation*)
2. **mpi_alltoall**(*number of items*, *datatype of items*)
3. **mpi_bcast**(*number of items*, *datatype of items*)
4. **mpi_irecv**(*number of items*, *datatype of items*, *destination*)
5. **mpi_reduce**(*number of items*, *datatype of items*, *reduction operation*)
6. **mpi_send**(*number of items*, *datatype of items*, *destination*)

For function calls where the source, destination, or message length cannot be succinctly mathematically expressed, the function call does not contain any fields.

Finally, the pseudocode for the all benchmarks is appropriate for an arbitrary number of processors except the pseudocode for the MG benchmark. Within the MG benchmark, multiple grids of varying resolution are used to find the solution to an equation. The resolution of these fields depends on a variety of factors such as the number of processors, number of grids, and problem size. As the number of processors increases, then, the communication structure of the benchmark changes. To reduce the complexity of the pseudocode, we have only given a version appropriate for sixteen or fewer processors.

A BT Benchmark Code

```
mpi_bcast(1,MPI_INTEGER)
mpi_bcast(1,MPI_DOUBLE_PRECISION)
mpi_bcast(3,MPI_INTEGER)

do i=1,200
   loop()
end do

mpi_allreduce(5,MPI_DOUBLE_PRECISION,MPI_SUM)

mpi_irecv()
mpi_irecv()
mpi_irecv()
mpi_irecv()
mpi_irecv()
mpi_irecv()
mpi_isend()
mpi_isend()
mpi_isend()
mpi_isend()
mpi_isend()
mpi_waitall()

mpi_allreduce(5,MPI_DOUBLE_PRECISION,MPI_SUM)

mpi_reduce(1,MPI_DOUBLE_PRECISION,MPI_MAX)

---------------

loop()
{
 mpi_irecv()
 mpi_irecv()
 mpi_irecv()
 mpi_irecv()
 mpi_irecv()
 mpi_irecv()
 mpi_isend()
 mpi_isend()
 mpi_isend()
 mpi_isend()
 mpi_isend()
 mpi_isend()
 mpi_waitall()

 /* x-axis */
 do i=1,sqrt(number_of_processors)
    mpi_isend()
    mpi_irecv()
    mpi_wait()
    mpi_wait()
 end do

 /* x-axis */
 do i=1,sqrt(number_of_processors)
    mpi_isend()
    mpi_irecv()
    mpi_wait()
    mpi_wait()
 end do

 /* y-axis */
 do i=1,sqrt(number_of_processors)
    mpi_isend()
    mpi_irecv()
    mpi_wait()
    mpi_wait()
 end do

 /* y-axis */
 do i=1,sqrt(number_of_processors)
    mpi_isend()
    mpi_irecv()
    mpi_wait()
    mpi_wait()
 end do

 /* z-axis */
 do i=1,sqrt(number_of_processors)
    mpi_isend()
    mpi_irecv()
    mpi_wait()
    mpi_wait()
 end do

 /* z-axis */
 do i=1,sqrt(number_of_processors)
    mpi_isend()
```

```
    mpi_irecv()
    mpi_wait()
    mpi_wait()
 end do
}
```

B FT Benchmark Code

```
count=33554432/(number_of_processors*number_of_processors)

mpi_bcast(3,MPI_INTEGER)
mpi_bcast(1,MPI_INTEGER)

mpi_alltoall(count,MPI_DOUBLE_COMPLEX)

do i=1,20
   mpi_alltoall(sendbuf,count,MPI_DOUBLE_COMPLEX)
   mpi_reduce(1,MPI_DOUBLE_COMPLEX,MPI_SUM)
enddo
```

C IS Benchmark Code

```
do i=1,10
   loop()
end do

mpi_reduce(1,MPI_DOUBLE_PRECISION,MPI_MAX)

if (pid > 0)
 mpi_irecv(1,MPI_INTEGER,pid-1)
end if

if (pid < number_of_processors-1)
 mpi_send(1,MPI_INTEGER,pid+1);
end if

mpi_wait()

mpi_reduce(1,MPI_INTEGER,MPI_SUM)

---------------------------------

loop()
{
 mpi_allreduce(1029,MPI_INTEGER,MPI_SUM)

 mpi_alltoall(1,MPI_INTEGER)

 mpi_alltoallv()
}
```

D LU Benchmark Code

```
mpi_bcast(1,MPI_INTEGER)
mpi_bcast(1,MPI_INTEGER)
mpi_bcast(1,MPI_INTEGER)
mpi_bcast(1,MPI_DOUBLE_PRECISION)
mpi_bcast(1,MPI_DOUBLE_PRECISION)
mpi_bcast(5,MPI_DOUBLE_PRECISION)
mpi_bcast(1,MPI_DOUBLE_PRECISION)
mpi_bcast(1,MPI_DOUBLE_PRECISION)
mpi_bcast(1,MPI_DOUBLE_PRECISION)

do i=1,2
  mpi_irecv()
  if (number_of_processors > 4)
     mpi_send()
  end if
  mpi_wait()

  if (number_of_processors > 4)
    mpi_irecv()
  end if

  mpi_send()

  if (number_of_processors > 4)
     mpi_wait()
  end if

  if (number_of_processors > 2)
     mpi_send()
     mpi_irecv()
     mpi_wait()
  end if
end do

mpi_allreduce(5,MPI_DOUBLE_PRECISION,MPI_SUM)

mpi_barrier()

do i=1,249
  j_loop()
  k_loop()
  mpi_irecv()
  if (number_of_processors > 4)
     mpi_send()
  end if
  mpi_wait()
  if (number_of_processors > 4)
     mpi_irecv()
  end if
  mpi_send()
  if (number_of_processors > 4)
     mpi_wait()
  end if
  if (number_of_processors > 2)
     mpi_send()
     mpi_irecv()
     mpi_wait()
  end if
end do

j_loop()
k_loop()

mpi_allreduce(5,MPI_DOUBLE_PRECISION,MPI_SUM)
mpi_irecv()

if (number_of_processors > 4)
   mpi_send()
end if

mpi_wait()
mpi_send()

if (number_of_processors > 4)
   mpi_wait()
end if

if (number_of_processors > 2)
   mpi_send()
   mpi_irecv()
   mpi_wait()
end if

mpi_allreduce(5,MPI_DOUBLE_PRECISION,MPI_SUM)
mpi_allreduce(1,MPI_DOUBLE_PRECISION,MPI_MAX)
mpi_allreduce(5,MPI_DOUBLE_PRECISION,MPI_SUM)

if (number_of_processors > 2)
   mpi_irecv(()
```

```
   mpi_wait()
end if

if (number_of_processors > 4)
   mpi_irecv()
   mpi_wait()
end if

mpi_send()
mpi_allreduce(1,MPI_DOUBLE_PRECISION,MPI_SUM)

if (number_of_processors > 4)
   mpi_irecv()
   mpi_wait()
end if

mpi_send()
mpi_allreduce(1,MPI_DOUBLE_PRECISION,MPI_SUM)

if (number_of_processors == 4)
   mpi_irecv()
   mpi_wait()
end if

mpi_allreduce(1,MPI_DOUBLE_PRECISION,MPI_SUM)

-------------------------------------------

j_loop()
{
 do j=1,62
    mpi_recv()
  if (number_of_processors > 4)
    mpi_send()
  end if
  if (number_of_processors > 2)
    mpi_send()
  end if
 end do
}

-------------------------------------------

k_loop()
{
 do k=1,62
```

```
   if (number_of_processors > 4)
      mpi_recv()
   end if
   if (number_of_processors > 2)
      mpi_recv()
   end if
   mpi_send()
 end do
}
```

E MG Benchmark Code

```
if (number_of_processors < 16)
{
 mpi_bcast(1,MPI_INTEGER)
 mpi_bcast(1,MPI_INTEGER)
 mpi_bcast(1,MPI_INTEGER)
 mpi_bcast(1,MPI_INTEGER)
 mpi_bcast(1,MPI_INTEGER)
 mpi_bcast(8,MPI_INTEGER)
 mpi_barrier()

 do i=1,10
    loop3()
 end do

 mpi_barrier()

 do i=1,2
    loop1()
 end do

 do i=1,23
    loop2()
 end do

 mpi_barrier()

 do i=1,10
    loop3()
 end do

 mpi_barrier()

 loop2()

 mpi_barrier()

 loop1()

 do i=1,459
    loop2()
 end do

 loop1()

 mpi_reduce(1,MPI_DOUBLE_PRECISION,MPI_MAX)
}
else
{
 mpi_bcast(1,MPI_INTEGER)
 mpi_bcast(1,MPI_INTEGER)
 mpi_bcast(1,MPI_INTEGER)
 mpi_bcast(1,MPI_INTEGER)
 mpi_bcast(1,MPI_INTEGER)
 mpi_bcast(8,MPI_INTEGER)
 mpi_barrier()

 do i=1,10
    loop3()
 end do

 mpi_barrier()

 do i=1,2
    loop1()
 end do

 do i=1,6
    loop2()
 end do

 mpi_irecv()
 mpi_irecv()
 mpi_wait()
 mpi_wait()

 do i=1,5
    loop2()
 end do

 mpi_barrier()

 do i=1,10
    loop3()
 end do

 mpi_barrier()

 loop2()
```

```
 mpi_barrier()

 loop1()

 do i=1,6
    loop2()
 end do

 do i=1,19
   mpi_irecv()
   mpi_irecv()
   mpi_wait()
   mpi_wait()
   do j=1,21
     loop2()
   end do
 end do

 mpi_irecv()
 mpi_irecv()
 mpi_wait()
 mpi_wait()

  do j=1,14
    loop2()
  end do

  mpi_reduce(1,MPI_DOUBLE_PRECISION,MPI_MAX)
}

-------------------------------------------

loop1()
{
 loop2()
 mpi_allreduce(1,MPI_DOUBLE_PRECISION,MPI_MAX)
 mpi_allreduce(1,MPI_DOUBLE_PRECISION,MPI_SUM)
}

-------------------------------------------

loop2()
{
 mpi_irecv()
 mpi_irecv()
 mpi_send()
 mpi_send()
 mpi_wait()
 mpi_wait()
 mpi_irecv()
 mpi_irecv()
 mpi_send()
 mpi_send()
 mpi_wait()
 mpi_wait()
}

-------------------------------------------

loop3()
{
 mpi_allreduce(1,MPI_DOUBLE_PRECISION,MPI_MAX)
 mpi_allreduce(1,MPI_DOUBLE_PRECISION,MPI_MAX)
 mpi_allreduce(1,MPI_DOUBLE_PRECISION,MPI_MIN)
 mpi_allreduce(4,MPI_INTEGER,MPI_MAX)
}
```

F SP Benchmark Code

```
mpi_bcast(1,MPI_INTEGER)
mpi_bcast(1,MPI_INTEGER)
mpi_bcast(3,MPI_INTEGER)

loop()
mpi_barrier()

do i=1,400
   loop()
end do

mpi_allreduce(5,MPI_DOUBLE_PRECISION,MPI_SUM)

mpi_irecv()
mpi_irecv()
mpi_irecv()
mpi_irecv()
mpi_irecv()
mpi_irecv()
mpi_isend()
mpi_isend()
mpi_isend()
mpi_isend()
mpi_isend()
mpi_isend()
mpi_waitall()

mpi_allreduce(5,MPI_DOUBLE_PRECISION,MPI_SUM)
mpi_reduce(1,MPI_DOUBLE_PRECISION,MPI_MAX)

----------------------------------------

loop()
{
 mpi_irecv()
 mpi_irecv()
 mpi_irecv()
 mpi_irecv()
 mpi_irecv()
 mpi_irecv()
 mpi_isend()
 mpi_isend()
 mpi_isend()
 mpi_isend()
 mpi_isend()
 mpi_isend()
 mpi_waitall()

 /* x-axis */
 do i=1,sqrt(number_of_processors)-1
    mpi_isend()
    mpi_irecv()
    mpi_waitall()
 end do

 /* x-axis */
 do i=1,sqrt(number_of_processors)-1
    mpi_isend()
    mpi_irecv()
    mpi_waitall()
 end do

 /* y-axis */
 do i=1,sqrt(number_of_processors)-1
    mpi_isend()
    mpi_irecv()
    mpi_waitall()
 end do

 /* y-axis */
 do i=1,sqrt(number_of_processors)-1
    mpi_isend()
    mpi_irecv()
    mpi_waitall()
 end do

 /* z-axis */
 do i=1,sqrt(number_of_processors)-1
    mpi_isend()
    mpi_irecv()
    mpi_waitall()
 end do

 /* z-axis */
 do i=1,sqrt(number_of_processors)-1
    mpi_isend()
    mpi_irecv()
    mpi_waitall()
 end do
}
```