Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 03-032

MPI-based Adaptive Parallel Grid Services

Lakshman Abburi Rao and Jon Weissman

August 26, 2003

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
26 AUG 2003		**-**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
MPI-based Adaptive Parallel Grid Services	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Army Research Laboratory,2800 Powder Mill Road,Adelphi,MD,20783-1197**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**13**	

MPI-based Adaptive Parallel Grid Services[1]

Lakshman Aburri Rao and Jon B. Weissman

Department of Computer Science and Engineering
University of Minnesota, Twin Cities
(jon@cs.umn.edu)

Abstract

This report presents the design and implementation of an adaptive MPI implmentation (adaptive-MPI) that allows an MPI application to adapt to respond to changing CPU availability. An adaptive MPI application can start sooner with fewer processors, opportunistically add processors later should they become available, and release processors to avoid suspension should the resource owner take them back. The behavior of adaptive-MPI is well-suited to the unpredictable and dynamic nature of the Grid. We presents results that indicate the systems overhead of adaptive MPI is small, and that performance benefits in terms of reduced waiting time and reduced completion time can be achieved relative to traditional MPI.

1.0 Introduction

The Message Passing Interface (MPI) is perhaps the most widely adopted parallel programming standard and has been implemented on a large variety of parallel machines from clusters to parallel super-computers [10]. The central abstraction within MPI is the *communicator*, a structure that maintains the set of (potentially) communicating **processes**. In standard MPI, the communicator is static and established when the MPI program is launched on a fixed set of processors. The static nature of MPI programs places two limitations on them: (1) when a node fails, the communicator fails, and the application must be aborted, and (2) resource allocation is set at the beginning of the program and cannot be changed. In retro-spect, these properties are reasonable on reliable, dedicated, parallel machine platforms. In non-dedicated computing environments with more dynamic resource sharing, e.g. Computational Grids [6][7], or desktop cycle stealing [9], a more dynamic programming model is beneficial. In this paper, we present the design and implementation of a more dynamic MPI that does not suffer from these limitations. Our dynamic MPI

1. This work was funded in part by the Department of Energy DE-FG02-03ER25554 and National Science Foundation NSF-0305641.

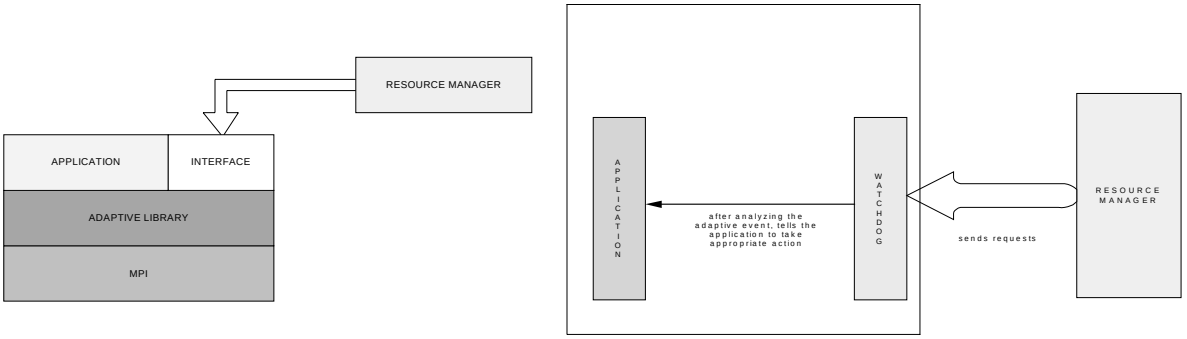
maintains the standard MPI library interface, but allows MPI programs to be more adaptive to their environment. In particular, they can recover from node failure and allow processors to be dynamically added or removed while they are running. The benefits of adaptive-MPI are three-fold: (1) the ability to opportunistically add resources if they become available, (2) the ability to release resources to prevent preemption should a higher priority job require a subset of its resources, and (3) the ability to continue running in the face of node failure. The ability to acquire resources later also has the side-effect of allowing jobs to start earlier since they need not wait for a "full allocation" at the start. We could have opted to design a brand new parallel programming model that supported adaptivity as a core abstraction and ended up with a cleaner model. However, the development of an adaptive MPI can immediately impact the large number of legacy MPI applications already written. An adaptive MPI can also promote more efficient scheduling of multiple MPI jobs [12], and more efficient execution of MPI-based parallel network services [8]. Other approaches to adaptive MPI have been proposed such as AMPI [2], but these schemes generally refer to dynamic load balancing within a set of static processes or are limited to thread-level adaptation [3], or are limited only to fault recovery [1][4]. We are unaware of an MPI implementation that can dynamically add or remove resources at run-time. **In prior work, we have established the costs and benefits of adaptation for non-MPI parallel applications [11].**

In this paper, we describe the design and implementation of adaptive-MPI, and performance results for an illustrative MPI application, a Jacobi iterative solver. In addition, in a Clustered-Grid environment, MPI may not be running by default, and may require on-demand launching when an MPI application is submitted for execution. We presents results that compare the cost of on-demand vs. pre-instantiation of our MPI infrastructure. We also present results for the cost of MPI adaptivity at the system-level, i.e. adding and removing processors from the **MPI ring**. Overall, the results demonstrate that adaptive MPI can promote low latency (i.e. small wait time) and high performance (i.e. reduced finishing time) at acceptable cost.

2.0 Adaptive MPI Architecture

We have developed an adaptive MPI architecture that allows MPI applications to have resources dynamically added and removed (Figure 1). We have started with an implementation of MPI called Fault Tolerant MPI (FT-MPI) [12] as its more modular design enabled our changes to be easily integrated. In particular, adding support for dynamic communicators was very straightforward. Our adaptive-MPI introduces several new components, one internal to MPI called the *watchdog*, and the other external, called the *resource manager*. The *resource manager* makes decisions about resource availability. How the resource manager makes such decisions, discovers new resources, takes away resources, is outside the MPI model. At present, it generates two adaptive events, `add_resource` and `release_resource`. These events

are propagated to the watchdog process that is started when the application is submitted. The watchdog is a separate process that receives adaptive events along a TCP connection and delivers them to the application via Unix signals[2].



(a) Layered View (b) MPI components

Figure 1: Adaptive MPI Architecture**.**

When an adaptive event is generated to the application, the MPI application must detect the event and respond in a manner that is specific to the application. In particular, the application must perform any necessary data distribution to rebalance the application. However, the adaptive-MPI automatically spawns new processes (if the event is `add_resource`), and dynamically adjusts the communicator to account for newly arriving or departing nodes (Figure 2).

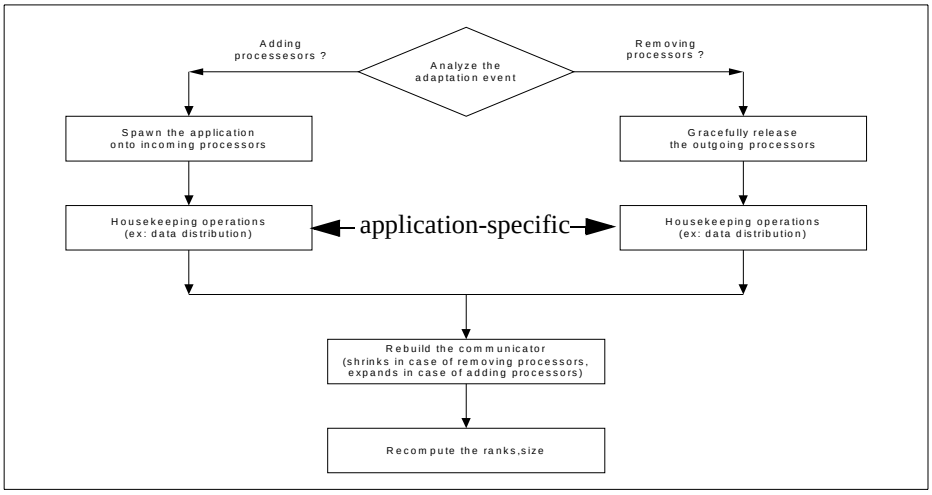


Figure 2: Adaptation logic.

2. Future optimizations include a threaded watchdog and the use of shared memory communication.

The dynamics of the default communicator `MPI_COMM_WORLD` is shown for processor addition and removal (Figure 3). Before the communicator is modified, the application must respond to the adaptive event and redistribute any data for load balance.

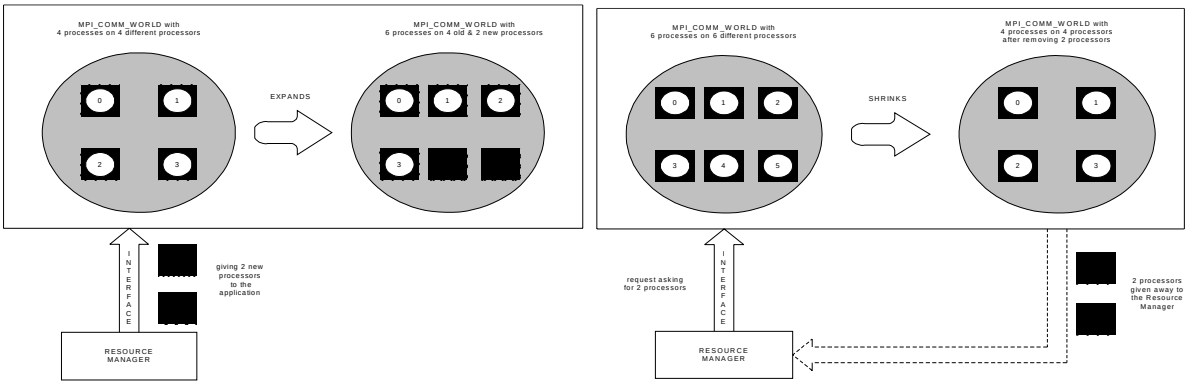


Figure 3: Dynamics of the Communicator.

To illustrate Adaptive-MPI in action, we use a canonical Jacobi solver in which the grid is decomposed by row to the selected processors. Initially the grid is on disk and decomposed across 4 processors as shown (Figure 4). When an adaptive event occurs, all processes write back their updated partition of the grid to disk in a synchronous fashion at the same iteration (Figure 5). After the communicator is repaired (in this case to add a processor), the slaves restart by retrieving their newly updated partition of the grid from disk reflecting the new number of processors (in this case, 5). **An alternative and more optimized adaptation mechanism would allow the slaves to communicate the updated partition directly using messages, rather than through the disk.** However, we show that even with a less optimized implementation, the overheads of adaptivity are easily managed.

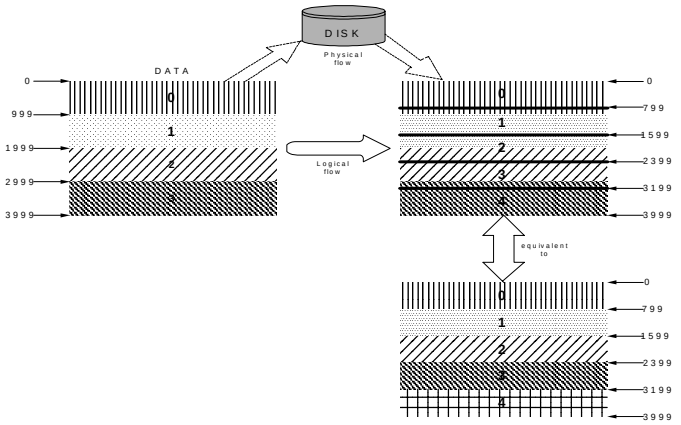


Figure 4: Jacobi Example. Data distribution.

Parameters: number of processes, application size, required number of iterations.
1. Initialize MPI. Compute Ranks & Size of the communicator.
2. Check if this process is created by an adaptation event.
   2a. If yes, receive piece of the problem from the neighboring processes.
       Else receive piece of the problem as input from the user.
3. Iterate until done
   4. Check for adaptation events from the Watchdog or if adaptation has to be done in this iteration.
      4a. If yes, go to step-5 Else go to step-15.
   5. Check for adaptation events from Watchdog.
      5a. If yes, communicate with neighbors. Find next possible nearest iteration in future for adaptation.
          Go to step-15.
          Else this is adaptive iteration. Check if removing processors. If yes, go to step-6. Else go to step-11.
   6. Write piece of the problem that has to be distributed (if any) onto the disk.
   7. Check if I (this process) have to exit. If yes, Exit my Watchdog then Exit myself.
   8. Check if I am the top process (rank = 0). If yes, call **`MPI_Remove_Processors`** `(number)`.
   9. Rebuild the communicator. Recompute Ranks & Size.
   10. Get my new piece of the problem from the disk. Go to step-15.
   11. Write the piece of the problem that has to be distributed onto the disk. /*Adding Processors */
   12. Check if I am the top process (rank = 0). If yes, call **`MPI_Add_Processors`** `(number).`
   13. Rebuild the communicator. Recompute Ranks & Size: `MPI_Comm_Dup/Rank/Size/ (…);`
   14. Get my new piece of the problem (if any) from the disk.
   15. Communicate to/from neighbors. /* From here usual Jacobi */
   16. Compute local data domain.
   17. Perform convergence check periodically.
18. End iterate.

Figure 5: Pseudo-code for adaptive-MPI version of Jacobi. Calls provided by adaptive-MPI shown in courier.

3.0 Results

Our adaptive-MPI infrastructure was deployed on a shared network of 10 UltraSPARCs (IIe - 502 Mhz, 512 MB RAM, and IIi - 333 Mhz, 128 MB RAM) all running Solaris 5.8**,** connected by 100 Mb ethernet. The first question we examined was the basic overheads inherent in adaptive MPI at the system level. This includes the adding and removing of processors and processes and rebuilding internal data-structures such as the communicator (Table 1). In these experiments, we assume that the MPI infrastructure is already running. Rebuilding the communicator can be expensive as it requires network communication. Adding a process has an additional cost, an application-level process must be created via `fork-exec`. The current system is not fully optimized as the addition or removal of multiple processes at the same time is essentially serialized resulting in added overhead. That is, as each process is added or removed the communicator is rebuilt. In spite of this, the benefits of adaptation we still be demonstrated.

One of the motivating environments for future experimentation is the Grid and the ability to launch MPI “on-demand” is something that we also support (Figure 6). For on-demand, we show the cost of bringing up MPI (but not any application processes). This cost consists of bringing up the underlying MPI server daemon via `ssh`. The cost of on-demand instantiation of MPI is low because the startup procedure brings up the daemons in parallel and there is no need to rebuild the communicator (Table 2). The cost of

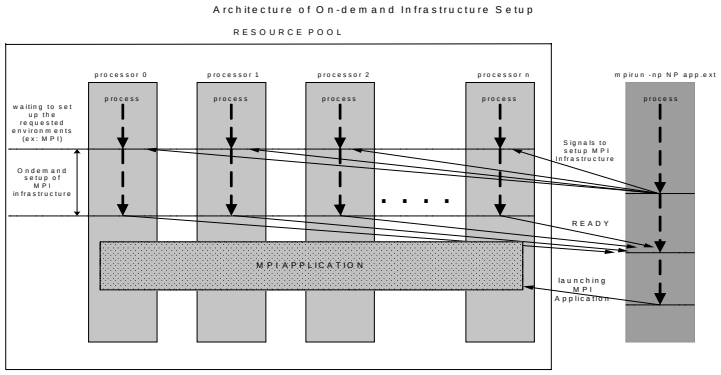


Figure 6: On demand Infrastructure.A barrier is formed to ensure that all MPI daemons are launched an running before the application is initiated.

on-demand adaptation can be approximated by simply adding up corresponding cells from both tables. For example, the system cost of an on-demand addition of 8 processors to a running MPI application would be approximately 27.7 + 1.5 = 29.2 sec. With most parallel jobs running in the 1000s range or more, on-demand execution is definitely viable.

To evaluate the performance of adaptive-MPI from the perspective of the application, we designed two adaptation models and applied them to the modified Jacobi application. The first model *model-1* is largely synthetic and used to measure the costs and benefits of opportunistic adaptation (only adding processors) to an MPI application. The second model *model-2* is more realistic and uses measured workstation traces to drive to placement of adaptation events, both addition and removal. Using these models, we then compare adaptive-MPI to a static MPI that is unable to adapt to changing resource availability.

Table 1: Cost of Adding and Removing **Processors**

# of processors	cost of adding (sec)	cost of removing (sec)
1	1.7	.8
2	3.6	2.0
3	5.9	3.3
4	8.7	4.6
5	11.7	6.5
6	15.4	8.0
7	19.7	10.6
8	27.7	13.7

Table 2: Cost of On-Demand Execution

# of processors	cost (sec)
1	.32
2	.44
3	.64
4	.83
5	.99
6	1.2
7	1.4
8	1.5
9	1.8
10	1.9

In *model-1*, we assume that a fixed set of resources becomes available at regular intervals throughout the execution of the application. In this paper we set that interval to every 20 Jacobi iterations. At the outset, two processors are available and an additional processor becomes available every 20 iterations. With adaptive-MPI, the application is able to start immediately and expand as resources become available. With static-MPI, the application must either start and finish with a smaller number of resources (2 in this case) or wait until a more desireable number becomes available. The results indicate that adaptive-MPI can provide benefits for the application (Figure 7). The x-axis refers to the total number of processors (*TP*) that will eventually become available to the application. For example, P=2 at iter=0, P=3 at iter=20, ... P=*TP* at iter=20*(*TP*-2)). At the first x-axis point, P=2 for the entire experiment, and at the next x-axis point P=2 up to iter=20 and P=3 for the rest of the run, and so on. P=2 is a baseline for this problem instance and processor availability pattern, and represents the worst-case performance of static-MPI. For the other points,

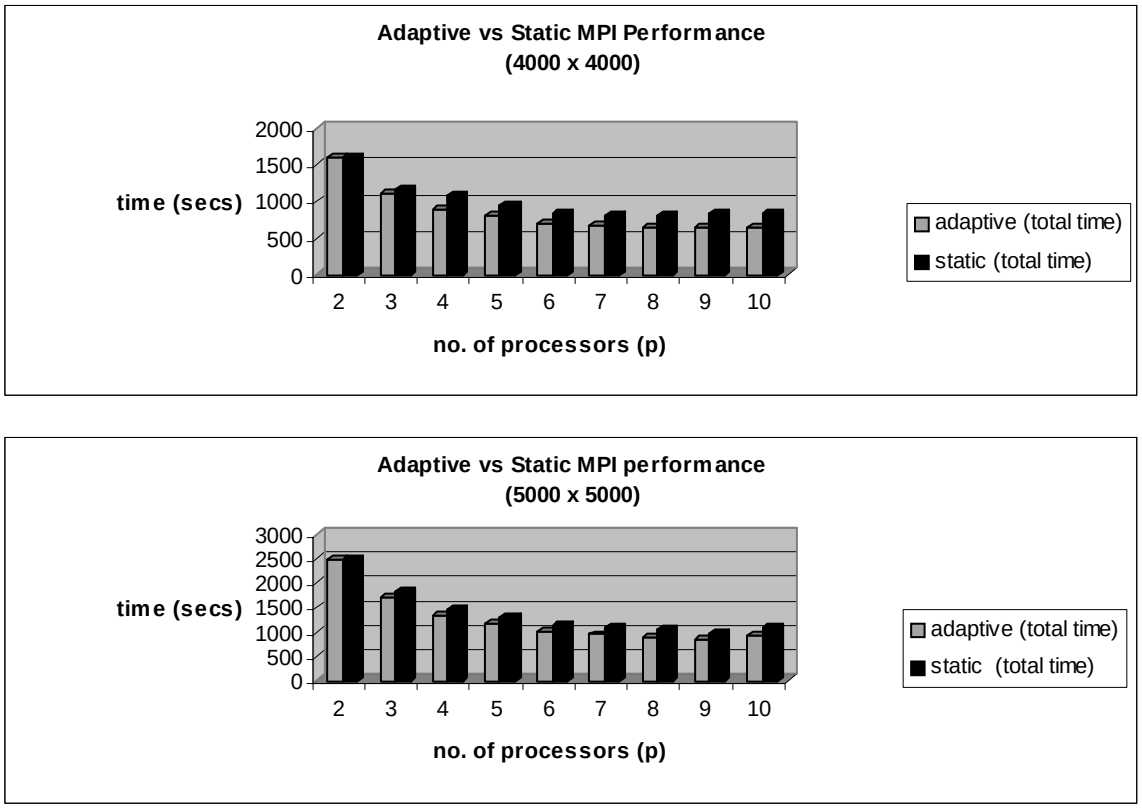


Figure 7: Comparative Performance for *model-1*: Adaptive vs. Static MPI. For Adaptive-MPI results include adaptation overhead. For Static-MPI results include wait time.

static-MPI will wait for the number of processors that will eventually be available. So for P=3, static-MPI will wait for 20 iterations (of real time) and then run with P=3 for the entire run, and so on. So as the X-axis increases: (1) static-MPI will incur greater wait time and (2) adaptive-MPI will incur greater adaptation overhead, due the number of adaptation events. This is the fundamental trade-off. These results

depend, of course, on the model of resource availability and the application, but if the overhead of adaptation is smaller than the wait time, or if the benefit of adaptation outweighs the "cost" of running sooner with fewer resources, then adaptation is a winner. In many parallel computing environments, wait times will dominate any adaptation overhead [12].

In *model-2*, we consider a more realistic network environment in which the MPI application shares resources dynamically with other applications. This model is most typical of a cluster or workstation network environment. In prior work, we used availability data from supercomputer workloads [5] but applied to the problem of job scheduling adaptive applications but in a simulation study only. We have analyzed 4-day traces from our 10 workstations. A workstation was considered to be available if the average load was below a certain threshold (0.3) for duration of 5 minutes. In 3 of the 4 traces, the initial pool size was smaller since long-running CPU-intensive local jobs occupied 1 or 2 machines. The results of the four traces is shown (Table 3).

Table 3: Availability

Time (min)	**# Available (trace-1) pool size = 9**	**# Available (trace-2) pool size = 9**	**# Available (trace-3) pool size = 8**	**# Available (trace-4) pool size = 10**
T	4	2	8	7
T+5	5	5	6	8
T+10	5	6	7	10
T+15	7	5	6	8
T+20	7	7	7	9
T+25	9	9	7	9
T+30	8	6	8	9
T+35 ...	9	9	8	10

We assume that the value measured at T+35 holds for the remainder of the application. The adaptive-MPI application is run at time T and adapts according to the resource availability in the traces above. For example, in trace-2, it would add 3 processors at T+5 (2->5), but at T+25 it would release 3 processors (9->6). In other words, we treat the background jobs as having greater priority in *model-2*. We also provide a more realistic model for application resource requirements. We benchmarked Jacobi off-line to determine the ideal of processors within our testbed of 10 machines. We also set a minimum number below which the application must be suspended. A minimum is normally required to enforce other resource constraints such as memory. In our experiments, we determined the ideal number to be 9 for most problem sizes, and we set the minimum to be 2. Static-MPI must wait for the ideal number of resources to begin execution and if the available resources fall below the ideal number, it would be *suspended* until the ideal number is available again. For example, in trace-3, the application was wait until T+10 (when 10 are available), is suspended at

T+15 (when 8 is available), and then resumes at T+20 (when 9 are available) and runs until completion. This captures the notion that static-MPI runs on a fixed, typically pre-specified, ideal number of resources.

The results indicate that adaptive-MPI far outperforms static-MPI in *model-2* (Figure 8). We have

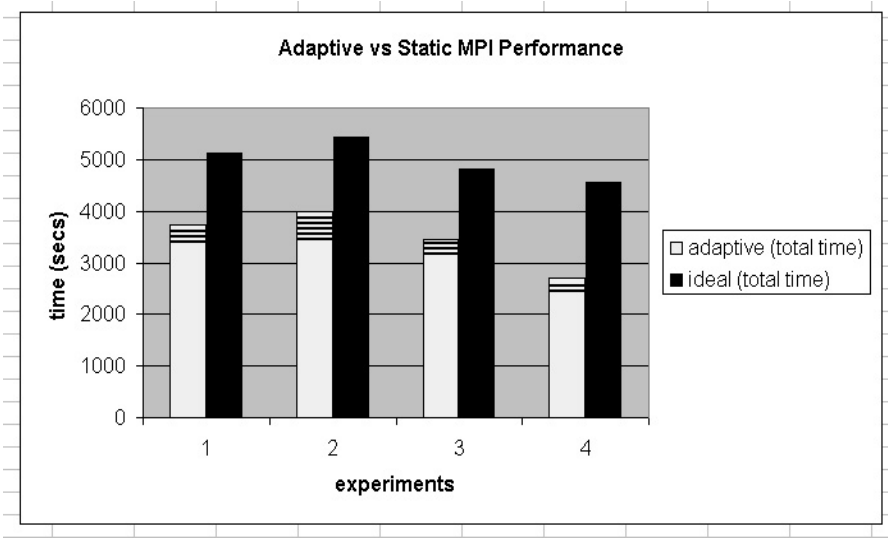


Figure 8: Comparative Performance for model-2: Adaptive vs. Static MPI. For Adaptive-MPI results include adaptation overhead. For Static-MPI results include wait and suspension time. Adaptation overhead is dashed box at the top of the adaptive bar graphs.

boosted the number of iterations to allow the application to run during a trace window (25+ minutes) to enable adaptations to occur. The principle reason that adaptive-MPI is superior in this environment is that in a shared network, the wait and suspend times far exceed the cost of adaptation. For instance, consider trace-1. In the case of static-MPI, only 4 free processors are available at T and 9 free processors (ideal number) are not available for the next 25 minutes. So the job must wait for 25 minutes until 9 are available. Once the job starts at time T + 25, the ideal number of processors are available only for 5 minutes. So the job has to suspend again at time T + 30, until the ideal number of processors become available once again. When the ideal number of processors becomes available at time T + 35, the job resumes its execution, and finishes.With adaptive-MPI, the job need not wait or suspend if the ideal number of processors are not available. It can start with the available processors if the available number of processors is greater than the minimum number of processors. It can consume the processors later as they become available. There is an overhead in adaptation, but we observed the overhead to be very small compared to the high wait and suspend times in static-MPI for this application and these traces. It should be noted that adaptive-MPI incurs minimal additional overhead over static-MPI unless adaptations are needed. A single conditional statement is added to the main loop and an initial extra process is added at the outset (this will be converted to a thread in the next version). Given the granularity of MPI applications that we expect to benefit from adap-

tive MPI (100's of seconds or more), this overhead is not noticeable. Because adaptation is an orderly process (unlike fault recovery) no checkpoints or data movement are needed unless adaptations occur.

In general, it is better to run on fewer resources (and make progress) then to wait for an ideal number of resources which may be sporadically available. This kind of adaptation becomes even more important in a Grid. We believe that as programmers move onto the Grid, they would like to take their MPI programs with them. Even for space-shared parallel machines, there is a large benefit to adaptation [12]. Certainly, the hidden cost is the need to provide adaptivity within the application and we are ultimately interested in ways to reduce this burden. However, the benefits are clear: the ability to add resources opportunistically has obvious benefits for high performance computing, and the ability to release resources to avoid suspension, also has merit.

4.0 Acknowledgements

This work was supported in part by the Minnesota Supercomputing Institute and the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAD19-01-2-0014, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred.

5.0 Summary and Future Work

We presented the design and implementation of an adaptive MPI library that allows MPI applications to dynamically adapt to changing resource availability in a shared environment, e.g. clusters or Grids. The results established that the basic costs of adaptation are manageable and that real performance benefits can be provided to MPI applications. Adaptive MPI applications need not wait for an ideal or specified number of resources as in traditional MPI, and can therefore achieved reduced latency or wait time. Such applications can also obtain additional resources at run-time further boosting performance. Finally, adaptive MPI applications can release releases to prevent suspension if higher priority users or jobs enter the system. This is particularly important for Grid computing - local site autonomy may require that resources be released to satisfy local users.

Future work includes optimizations to adaptive MPI when multiple processors are added or removed. Currently these are done serially in the run-time library. We are also investigating techniques to "insert" adaptivity into MPI applications automatically. Compiler-generated adaptive code and pre-built libraries for common parallel data-structures are two avenues we are exploring. Finally, we are exploring the use of adaptive MPI as a basis for constructing adaptive parallel network services. Such services can adapt to

concurrent user demand by sharing resources between competing requests. The ability to dynamically add and remove resources is needed to support dynamic resource sharing in this environment.

6.0 Bibliography

[1] FT-MPI: `http://icl.cs.utk.edu/iclprojects/source/ftmpi.html`, 2002.

[2] AMPI: `http://charm.cs.uiuc.edu/papers/AmpiSC02.html`, 2002.

[3] K. Shen, H. Tang, and T. Yang, “Adaptive Two-level Thread Management for Fast MPI Execution on Shared Memory Machines,” *Proceedings of ACM/IEEE SC'99, 1999*.

[4] G. Bosilca et al., “MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes,” *Proceedings of ACM/IEEE SC'02, 2002*.

[5] D. Feitelson, Parallel Workload Archive, `http://www.cs.huji.ac.il/labs/parallel/workload/logs.htm`, 1999.

[6] I. Foster and C. Kesselman, “Globus: A Metacomputing Infrastructure Toolkit,” *International Journal of Supercomputing Applications*, 11(2), 1997.

[7] A.S. Grimshaw and W. A. Wulf, “The Legion Vision of a Worldwide Virtual Computer,” *Communications of the ACM*, Vol. 40(1), 1997.

[8] B. Lee and J.B. Weissman, “Adaptive Resource Scheduling for Network Services," to appear in the *IEEE 3rd International Workshop on Grid Computing*, 2002.

[9] M.J. Litzkow et al., “Condor - a hunter of idle workstations,” In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

[10] MPI 2.0 Standard: `http://www-unix.mcs.anl.gov/mpi`, 2002.

[11] J.B. Weissman, “Predicting and Cost and Benefit of Adapting Data Parallel Applications in Clusters,”*Journal of Parallel and Distributed Computing*, 62(8), August 2002.

[12] J.B. Weissman, D. Velegaleti, D. England, and L. Rao, “Integrated Scheduling: The Best of Both Worlds,“in review for the *Journal of Parallel and Distributed Computing*, 2002.