Building MPI for Multi-Programming Systems using Implicit Information

Frederick C. Wong[1], Andrea C. Arpaci-Dusseau[2], and David E. Culler[1]

[1] Computer Science Division, University of California, Berkeley
{fredwong, culler}@CS.Berkeley.EDU
[2] Computer Systems Laboratory, Stanford University
dusseau@CS.Stanford.EDU

Abstract. With the growing importance of fast system area networks in the parallel community, it is becoming common for message passing programs to run in multi-programming environments. Competing sequential and parallel jobs can distort the global coordination of communicating processes. In this paper, we describe our implementation of MPI using implicit information for global co-scheduling. Our results show that MPI program performance is, indeed, sensitive to local scheduling variations. Further, the integration of implicit co-scheduling with the MPI runtime system achieves robust performance in a multi-programming environment, without compromising performance in dedicated use.

1 Introduction

With the emergence of fast system area networks and low-overhead communication interfaces [6], it is becoming common for parallel MPI programs to run in cluster environments that offer both high performance communication and multi-programming. Even if only one parallel program is run at a time, these systems utilize independent operating systems and may schedule sequential processes interleaved with the parallel program. The core question addressed in this paper is how to design the MPI runtime system so that realistic applications have good performance that is robust to multi-programming.

Several studies have shown that shared address space programs tend to be very sensitive to scheduling and in many cases only perform well when co-scheduled [1]. Fortunately, this work has also shown an interesting way for programs to co-ordinate their scheduling implicitly by making simple observations and reacting by either spinning or sleeping [2].

The first question to answer is whether or not MPI program performance is sensitive to multi-programming. It is commonly believed that message passing should tolerate scheduling variations, because the programming model is inherently loosely coupled and programs typically send large messages infrequently. It is used, after all, in distributed systems and PVM-style environments. However, local scheduling variations may cause communication events to become *out-of-sync* and impact the global

schedule [1].

In this paper, we show that MPI performance is, indeed, sensitive to scheduling; the common intuition is misplaced. Two or three programs running together may take 10 times longer than running each in sequence; competing with sequential applications has a similar slowdown. This result is demonstrated on the NAS Parallel Benchmarks [3] using a fast, robust MPI layer that we have developed over Active Messages [6] on a large high-speed cluster. We then show that simple techniques for implicit co-scheduling can be integrated into the MPI runtime library to make application performance very robust to multi-programming with little loss of dedicated performance.

This paper is organized as follows. We briefly describe our experimental environment and our initial MPI implementation in Section 2. In Section 3, we examine the sensitivity of message passing programs to multi-programming and show the influence of global uncoordination on application execution time. Section 4 describes our solution to this problem and gives a detailed description of our MPI implementation using implicit co-scheduling. Application performance results are discussed in Section 5.

2 Background

This section briefly describes our experimental environment and our initial implementation of the MPI Standard that serves as a basis for our study of sensitivity to multi-programming.

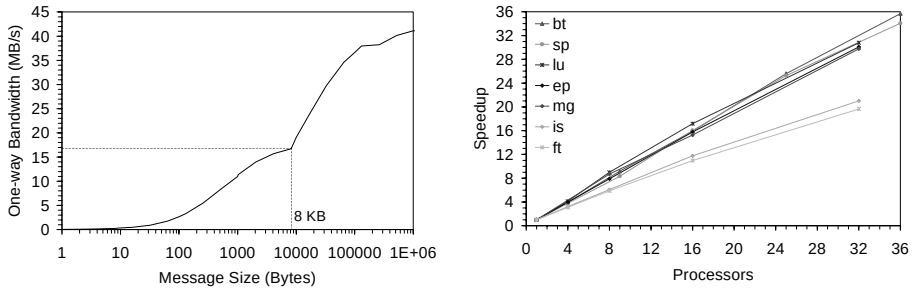
2.1 Experimental Environment

The measurements in this paper are performed on the U. C. Berkeley NOW cluster, which contains 105 UltraSPARC I model 170 workstations connected with 16-port Myrinet [4] switches. Each UltraSPARC has 512 KB of unified L2 cache and 128 MB of main memory. Each of the nodes is running Solaris 2.6, Active Messages v5.6 firmware, and GLUnix [7], a parallel execution tool used to start the processes across the cluster.

2.2 MPI-AM

Our MPI implementation is based on the MPICH [8] (v1.0.12) reference implementation by realizing the Abstract Device Interface through Active Message operations. This approach achieves good performance and yet is portable across Active Message platforms.

Active Messages. Active Messages [6] (AM) is a widely used low-level communication abstraction that closely resembles the network transactions that underlie modern parallel programming models. AM constitute *request* and *response* transactions which form restricted remote procedure calls. Our implementation of the Active Messages



(a) (b)

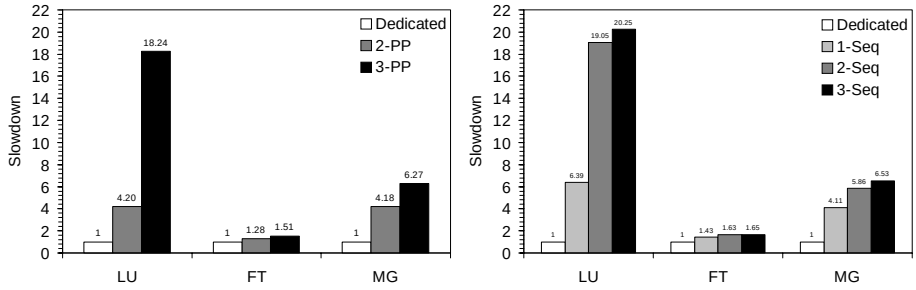
Fig. 1. These figures show the micro-benchmark and application performance of MPI-AM in a dedicated environment. Figure *a* shows the *one-way bandwidth* of MPI-AM with *message sizes* up to 1 MB. Figure *b* shows the *speedup* of the NAS benchmarks (v2.2) on up to 36 processors

API [9] supports a multi-user, multi-programming environment, and yet provides efficient user-level communication access.

Implementation. Our current implementation of MPI uses the *eager* protocol to transport messages in the abstract device. Each MPI message is transferred by a medium AM request (up to 8 KB) with communication group and rank, tag, message size, and other ADI specific information passed as request arguments. Messages larger than 8 KB are fragmented and transferred by multiple Active Messages. Instead of specifying the destination buffer, the sender transfers all message fragments to the destination process, which dynamically reorders and copies the message fragments into the appropriate buffer. A temporary buffer is allocated if the corresponding receive has not been posted. An MPI message is considered delivered when all associated AM requests are handled at the receiving node.

Performance in Dedicated Environment. Figure 1a shows the one-way bandwidth of MPI-AM using Dongarra's echo test [5]. The one-way bandwidth is calculated with the reciprocal of the one-way message latency, which is half of the average round-trip time. The start-up cost of MPI-AM is 19 μs (3 μs above that of the raw AM performance) and the maximum bandwidth achievable is 41 MB/sec. The kink at 8 KB shows the performance of a single medium request. The increase in bandwidth with message sizes larger than 8 KB is due to streaming multiple medium AM requests.

Figure 1b shows the speedup of the NAS benchmarks on class A data sizes on up to 36 dedicated processors in the cluster. Except FT and IS, for which the performance is limited by aggregate bandwidth, the benchmarks obtain near perfect speedup. Indeed, for a few benchmarks, speedup is slightly super-linear for certain machine sizes due to cache effects. To evaluate the impact of multi-program scheduling, we chose three applications (LU, FT, and MG) representing a broad range of communication and computation patterns that we might encounter in real workloads.



(a) (b)

Fig. 2. These figures show the *slowdown* of the NAS benchmarks in a multi-programming environment. Figure *a* shows the *slowdown* of *LU*, *FT*, and *MG* when one copy (*Dedicated*), two copies (*2-PP*) and three copies (*3-PP*) of the same benchmark are running on the same cluster. Figure *b* shows the *slowdown* of the same benchmarks competing with one copy (*1-Seq*), two copies (*2-Seq*) and three copies (*3-Seq*) of a sequential job

3 Sensitivity to Multi-programming

In a multi-programming environment, a local operating system scheduler is responsible for multiplexing processes onto the physical resources. The execution time of a sequential process is inversely proportional to the percentage of CPU time allocated, plus possible context switching overheads. Parallel applications spend additional communication and waiting time in the message passing library. For a parallel application running in a non-dedicated environment, the waiting time may increase if the processes are not scheduled simultaneously. In this section, we examine the sensitivity of parallel applications to multi-programming. In particular, we investigate the performance of the NAS benchmarks when multiple copies of parallel jobs are competing against each other, and when multiple sequential jobs are competing with a parallel job.

Figure 2a shows the slowdown of our three benchmarks when multiple copies of the same program are running. The rest of our experiments are performed on a set of 16 workstations. Slowdown is calculated by dividing the running time of the program in a multi-programming environment by the execution time of the program run sequentially in a dedicated environment. The slowdown of a workload with two jobs is equal to one if the workload runs twice as long as a single job in the dedicated environment. LU has the most significant drop in performance, with a slowdown of 18.2 when three copies of LU are competing for execution. MG and FT have a slowdown of 6.3 and 1.5 respectively. This shows that the performance of the NAS benchmarks is noticeably worse when they are not co-scheduled.

Figure 2b shows the effect on the NAS benchmarks of competing with sequential processes. The benchmark LU has a slowdown of 20.2, FT has a slowdown of 1.7 and MG has a slowdown of 6.5, when three copies of the sequential job are run. As when competing with parallel jobs, the performance of the benchmarks drops significantly when they are actively sharing with sequential jobs.

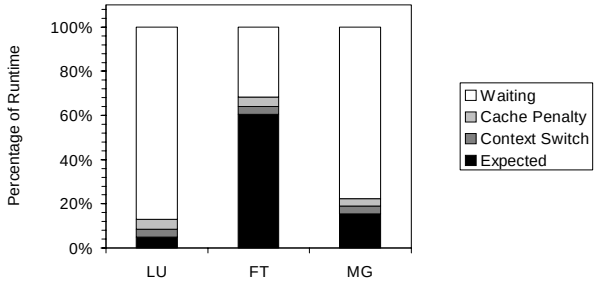


Fig. 3. This figure shows the execution time breakdown of the NAS benchmarks when three copies of a sequential process running

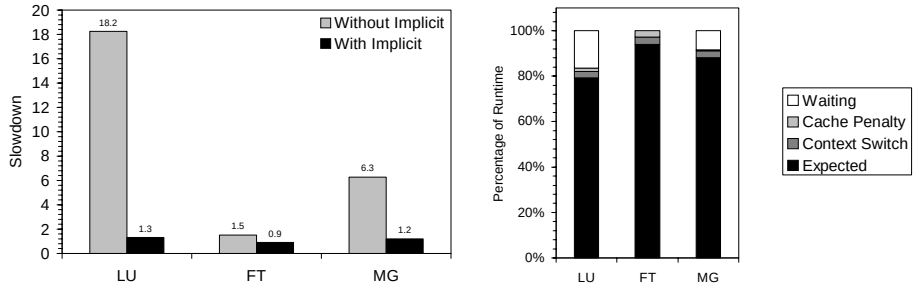
To further understand the performance impact of a multi-programming environment on parallel jobs, we profile the execution time of each benchmark. Figure 3 shows the execution time breakdown of the NAS benchmarks sharing with three copies of a sequential job. The *expected time* is the execution time of the benchmark in a dedicated environment. The *context switch time* is the overhead of context switching to other processes. The *cache penalty* is the extra time spent in the memory hierarchy due to cache misses caused by other processes. The *waiting time* is the additional time spent by processes waiting on communication events in the multi-programming environment.

Although the context switching cost and cache penalty are substantial, they constitute less than 10 percent of the execution time. The excessive slowdown of LU and MG is explained by a tremendous increase in waiting on communication events. For example, only 5 percent of the LU execution time is spent performing useful work while 85 percent of the time is spent on spin-waiting on message sends and receives.

As shown in these figures, parallel jobs are highly sensitive to global co-scheduling. The traditional solution to this problem in MPPs is explicit gang scheduling by the operating system. This is unattractive in general-purpose clusters and mixed workloads. In the following sections, we present an elegant solution obtained by modifying our implementation of MPI to use local information to achieve global co-scheduling.

4 Incorporating Implicit Co-scheduling

The idea of implicit co-scheduling is to use simple observations of communication events to keep coordinated processes running, and to block the execution of un-coordinated processes so as to release resources for other processes. If the round-trip time of a message is significantly higher than that expected in a dedicated environment, the sending process can infer that the receiving process is currently not scheduled. Therefore, by relinquishing the processor of the sending process, other local processes can proceed. On the other hand, a timely message response means the sender is probably scheduled currently, and consequently, the receiving process should remain scheduled.



(a) (b)

Fig. 4. Figure *a* compares the *slowdown* of the NAS benchmarks with and without implicit co-scheduling on three copies of the same benchmark. Figure *b* shows the *execution time breakdown* of the benchmarks when three copies of a sequential process are time-shared with each of the benchmarks using implicit co-scheduling

Two-phase spin-blocking is a mechanism that embodies these ideas. It consists of an active spinning phase and a block-on-message phase. In the active spinning phase, the sender actively polls the network for a reply for a fixed amount of time. In the block-on-message phase, the caller blocks until an incoming message is received. In order to keep active processes co-scheduled, the amount of time the sender needs to spin wait has to be at least the sum of the round-trip time plus the cost of a process context switch. On the other hand, the receiver should wait for the time of a one-way message latency under perfect co-scheduling.

We utilize these ideas and modify our MPI-AM runtime library to incorporate the two-phase spin-block mechanism wherever spin-waiting may occur. The first two places are trivial; two-phase spin-block is used when the sender is waiting for a reply from the receiver to confirm the delivery of the message (`MPID_Complete_send`), and when the receiver is waiting for the delivery of a message (`MPID_Complete_recv`).

Spin-waiting can also occur when posting a receive (`MPID_Post_recv`). Every MPI receive contains a *receive handle* that carries the receive information. When posting a receive by the application, the abstract device needs to ensure that the network device has not received the corresponding message by checking the *unexpected receive queue*. If no match is found, the *receive handle* is then posted to the *expected receive queue*. Otherwise, the message has been received by the network device and an *unexpected handle* is returned. Since large messages are fragmented, it is possible that the message is in the middle of transmission when the receiving process retrieves the *unexpected handle*. Our implementation spin waits until the entire message is buffered. Thus, the two-phase spin-block mechanism is needed to relinquish the processor if the sender is not delivering the message fragments fast enough.

Finally, we need to ensure that every layer in the message passing system does not spin wait. In our Active Messages implementation, an AM request operation might be blocked due to running out of flow control credits. In order to avoid the spin wait in the AM layer, MPI-AM keeps a outstanding requests counter. Two-phase spin-block is used whenever the counter reaches the pre-defined flow control limit of the AM layer.

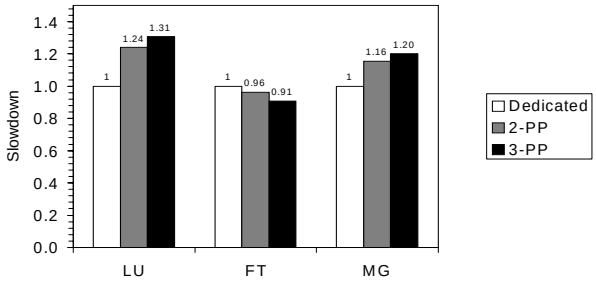


Fig. 5. This figure shows the slowdown of the NAS benchmarks when one copy (*Dedicated*), two copies (*2-PP*), and three copies (*3-PP*) of the same benchmark are run

5 Results

The extra complexity introduced by implicit co-scheduling to the MPI library only increases the one-way latency of small message by 1 μs on the echo test, whereas the execution time of a single copy of the NAS benchmarks is essentially unchanged.

Figure 4a compares the slowdown of running three copies of the NAS benchmarks with and without implicit co-scheduling. LU has the most significant improvement, from a slowdown of 18.2 down to 1.3. Figure 4b shows the execution time breakdown of the benchmarks when running with three copies of a sequential process. In LU the waiting time is reduced from 9900 seconds (Figure 3) to 160 seconds, in MG it is reduced from 180 seconds to 3 seconds and in FT, it is almost completely eliminated. The two-phase spin-block mechanism effectively reduces the spin-waiting time in the benchmarks without substantially increasing the context-switch overhead.

Figure 5 repeats the study of Figure 2a using implicit co-scheduling to demonstrate the scalability with different workload sizes. Both LU and MG experience a moderate increase in slowdown when multiple copies are run. This increase occurs because both application perform very frequent fine-grained communications.

FT, on the other hand, experiences a speedup when more copies are run. Because FT sends relatively large messages, its performance is limited by the aggregate bandwidth of the network and processes often spin wait for the messages to drain in and out of the network. In a dedicated environment, FT spends as much as 25 percent of its execution time spin waiting in communication events. When multiple copies of FT are implicitly co-scheduled, a waiting process eventually blocks, allowing other processes to continue. The total execution time of all processes is therefore reduced by interleaving the computation and communication across competing processes. Speedup is achieved when the benefit of interleaving outweighs the cost of multi-programming.

6 Conclusion

Previous studies have shown that implicit co-scheduling is useful for building parallel

applications using a fine-grain shared memory programming model. Our study indicates that loosely coupled message passing programs are highly sensitive to multi-programming as well, even on computationally intensive, bulk synchronous programs. Application slowdown caused by global uncoordination can be as high as 20 times. Without coordination, processes may waste up to 85 percent of their execution time spin-waiting for communication.

In this paper, we have presented an implementation of the MPI Standard for a distributed multi-programming environment. By leveraging implicit information for global co-scheduling, we effectively reduce the communication waiting time in the message passing applications caused by communication uncoordination. The performance of message passing applications is improved by as much as a factor of 10, reducing the application slowdown to 1.5 in the worst case studied. In one case, implicit co-scheduling helps a coarse grain message passing application yield better performance by interleaving communication with computation across parallel processes.

Acknowledgments

We would like to thank Shirley Chiu, Brent Chun, Richard Martin, Alan Mainwaring, and Remzi Arpaci-Dusseau for their many helpful comments and discussions on this work. This research has been supported in part by the DARPA (F30602-95-C0014), the NSF (CDA 94-01156), the DOE (ASCI 2DJB705), and the California MICRO.

Reference

1. R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, D. Patterson: The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In Proceedings of ACM Joint Intr. Conf. on Measurement and Modeling of Computer Systems, pp. 267-278, May 1995.
2. A. Arpaci-Dusseau, D. Culler, A. Mainwaring: Scheduling with Implicit Information in Distributed Systems. ACM SIGMETRICS'98/ PERFORMANCE'98.
3. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga: The NAS Parallel Benchmarks. Intr. J. of Supercomputer Applications. 5(3):66-73, 1991.
4. N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su: Myrinet - A Gigabet-per-Second Local-Area Network. IEEE Micro, 15(1):29-38, Feb. 1995.
5. J. Dongarra and T. Dunnigan: Message Passing Performance of Various Computers. University of Tennessee Technical Report CS-95-299, May 1995.
6. T. von Eicken, D. Culler, S. Goldstein, and K. Schauser: Active Messages: a Mechanism for Integrated Communication and Computation. In Proc. of the 19th ISCA, 1992, pp.256-266.
7. D. Ghormley, D. Petrou, S. Rodrigues, A. Vahdat, and T. Anderson: GLUnix: A Global Layer Unix for a Network of Workstations. Software Practice and Experience, 1989.
8. 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, Sept. 1996.
9. A. Mainwaring: Active Message Application Programming Interface and Communication Subsystem Organization. University of California at Berkeley, T.R. UCB CSD-96-918, 1996.