Automatic and Transparent Optimization of an Application's MPI Communication

Thorvald Natvig and Anne C. Elster

Norwegian University of Science and Technology, Sem Sælands vei 7-9, NO-7491 Trondheim, NORWAY,
thorvan@idi.ntnu.no,
WWW home page: http://www.idi.ntnu.no/ ~thorvan/

Abstract. HPC users frequently develop and run their MPI programs without optimizing communication, leading to poor performance on the cheaper clusters. Unfortunately, optimizing communication patterns will often decrease the clearness and ease of modification of the code, and users desire to focus on the application problem and not the tool used to solve it.
To this end, we present a new method for automatically optimizing any application's communication. By protecting the memory associated with MPI requests, we can let the request continue in the background as MPI_Isend or MPI_Irecv while the application is allowed to continue in the belief the request is finished. Once the data is accessed by the application, our protection will ensure we wait for the background transfer to finish before allowing the application to continue.
We have observed performance close to that of manual optimization on our testcases when run on normal clusters.

1 Introduction

This paper describes our method for automatic optimization of MPI [1] applications.

The specific task we have chosen to focus on is turning synchronous sends (send and wait for completion) into asynchronous sends (start sending in the background). While it sounds easy, it's not just a matter of replacing the synchronous sends with asynchronous ones. What happens if the original code first received into a buffer and then performed computation on this data? If the reception is started in the background, the data will not have arrived by the time computation starts, and hence the results will be wrong.

A more detailed description of the implementation and benchmarking may be found in [2].

1.1 Previous work

Faraj and Yuan [3] have presented such a method for automatically optimizing the MPI Collective subroutines, and Østvold has presented numerous ways of timing collective communication [4].

Ogawa and Matsuoka [5] use compiler modifications to optimize MPI. The compiler will recognize the MPI calls in a program, do a static analysis to find out what arguments are static and then create specialized MPI functions for that program. With the introduction of interprocedural optimizations such as is available in the Intel C++ Compiler [6], such optimizations can be extended to all function calls and not just MPI Calls.

2 Design and Implementation

As it was a design goal that the users should not have to change their code when using this new method, the first challenge was placing ourselves between the application and MPI. Our implementation therefore allows for two modes of program injection: Runtime (which works only if MPI is dynamically linked) or compile time (which works by overriding mpi.h).

2.1 Marking memory

When a overridden synchronous MPI function is called and turned into an asynchronous one, we need to track the memory area in use. When a pagefault occurs, we need to know what request we should wait for, and more importantly we need to make sure we don't have two active requests to the same memory area. Allowing two write requests to the same addresses would violate the dataflow of the program.

There are two problems in tracking memory. Overlapping elements and overlapping pages. Overlapping elements deals with two requests needing to access the exact same memory location, while overlapping pages deal with two requests needing access to the same page.

Figure 1 shows a theoretical example of all the requests for a classic Jacobi PDE solver with a 8x8 local grid, 1 layer of shadow cells, and a theoretical page size of 128 bytes. Each row represents one page in memory, and each color is a separate request, with bright colors being sends and dark colors being receives. No receive requests have overlapping elements, but the sends do for the corners. Additionally, the only requests that do not share pages are the ones for the top and bottom.

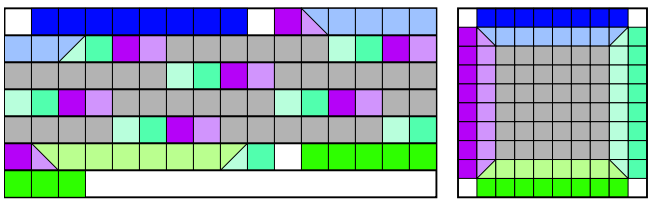

Fig. 1. Left: Memory cells used by border exchange in Jacobi PDE Solver. Right: Cells shown in structured form.

Our implementation accurately handles both requests with overlapping elements and requests with overlapping pages (but no overlapping or shared elements), as a simple 'if page is used, then wait' approach yielded no performance increases.

2.2 Chains of requests

A chain of requests is any phase of the program that is pure communication. For example, in our Red-Black SOR PDE Solver, the exchange of red borders is a chain of requests.

By recognizing such chains, and the knowledge that the majority of communication improvements are in communicating with multiple neighbors simultaneously, we can avoid the overhead of page protection by allowing the requests to start background transmission and wait for all outstanding requests at the end.

It is important that chains be remembered, as it's only once we have great confidence that a chain is identical for every iteration that we can perform this trick; otherwise the "end of chain" might never happen and the program might read or write unavailable data.

Our implementation generates a signature of all MPI calls, and once a series of specific signatures have been observed repeatedly, this is treated as part of a inner loop. No memory protection will be done, and we instead wait for all communication to finish at the last request in the chain.

3 Experiments and Results

We have implemented a Red-Black SOR 2D PDE Solver as it's a good representation of 2D communication patterns. Additionally, it's alternating red and black cells stresses our method by forcing it to operate under less-than-ideal conditions.

We have 3 versions implemented. The first version simply uses MPI_Sendrecv of the 4 borders on each red or black phase of each iteration. The second version starts sending and receiving all 4 borders in parallel using MPI_Isend and MPI_Irecv. The final version is optimized even further by first computing the updated boundary cells, then exchanging those in the background while computing interior cells.

These 3 versions are then compared with the automatically tuned program (using just MPI_Sendrecv).

We ran these benchmarks on numerous machines, and presented here are the numbers from our cluster (3.4 Ghz Pentium 4 with Gigabit Ethernet).

4 Conclusion

We have implemented, tested and verified a method for automatic runtime optimization of communication patterns. Our method requires little or no user intervention and, with paging only, cannot break data flow.

Method	$n = 128$	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$	$n = 4096$
MPI_Sendrecv	1.14	3.01	3.01	3.49	6.05	16.98
MPI_Isend	0.36	0.99	1.00	1.97	4.48	14.09
Full overlap	0.35	0.99	1.00	1.15	3.87	13.84
Paging	0.92	1.13	1.32	2.94	6.34	22.01
Chaining	0.43	1.07	1.06	2.04	4.55	14.17

Table 1. Average iteration execution time in milliseconds of automatically optimized program compared to manually optimized on 16 nodes on a cluster.

It is fully transparent, so a system administrator might install the static injection as part of the mpicc system, and users would not notice anything but a small speedup of their programs.

The improvements make normal applications based on MPI_Sendrecv rival those written with MPI_Isend. This allows users to think and write using simple communication patterns which leads to greater productivity and faster application development or them, and it lets us focus on the optimization part at runtime. We hope this focus on optimizing "average" applications is something we can inspire others to follow.

References

1. M. P. I. Forum, Technical Report No. UT-CS-94-230 (unpublished).
2. T. Natvig, Automatic Optimization of MPI Applications: Turning Synchronous Calls Into Asynchronous, 2006.
3. A. Faraj and X. Yuan, in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing* (ACM Press, New York, NY, USA, 2005), pp. 393–402.
4. Åsmund Østvold, Timing and Measurement Techniques for MPI Collective Communication Operations, 2003.
5. H. Ogawa and S. Matsuoka, in *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)* (IEEE Computer Society, Washington, DC, USA, 1996), p. 37.
6. C. Dulong *et al.*, Intel Technology Journal 15 (1999).