Run-Time Analysis and Instrumentation for Communication Overlap Potential

Conference Paper · September 2010

DOI: 10.1007/978-3-642-15646-5_5 · Source: DBLP

CITATION
1

READS
18

2 authors:

Thorvald Natvig

7 PUBLICATIONS **15** CITATIONS

SEE PROFILE

Anne C. Elster

Norwegian University of Science and Technology

80 PUBLICATIONS **856** CITATIONS

SEE PROFILE

Run-Time Analysis and Instrumentation for Communication Overlap Potential

Thorvald Natvig and Anne C. Elster

Norwegian University of Science and Technology(NTNU)
Sem Sælands vei 9, NO-7491 Trondheim, Norway
{thorvan,elster}@idi.ntnu.no

Abstract. Blocking communication can be runtime optimized into non-blocking communication using memory protection and replacement of MPI functions. All such optimizations come with overhead, meaning no automatic optimization can reach the performance level of hand-optimized code.In this paper, we present a method for using previously published runtime optimizers to instrument a program, including measured speedup gains and overhead.The results are connected with the program symbol table and presented to the user as a series of source code transformations. Each series indicates which optimizations were performed and what the expected saving in wallclock time is if the optimization is done by hand.

Keywords: MPI Overlap Communication Instrumentation Analysis.

1 Introduction

For point-to-point communication, MPI [1] offers two methods of communication; blocking and non-blocking. Blocking communication is the easiest to use, as communication is complete by the time the function call returns. When using blocking communication, all communication time is overhead, and this limits effective speedup.

Non-blocking communication offers an alternative, where control immediately returns to the program, while the communication operation is performed asynchronously. The program must not reference the data area until communication is complete, which makes this method of communication harder to use correctly. If the programmer forgets to explicitly wait for communication to finish before accessing data, the program may read unreceived data or write to unsent data, causing data corruption. Often, these problems do not appear in small test runs, but only appear when the application is scaled to larger problem sizes on a larger number of nodes. However, non-blocking communication allows overlap both between inidividual communication requests and with computation, greatly reducing the effective overhead of parallelization.

1.1 Previous Work

We have previously shown that it is possible to run-time optimize applications by turning blocking communication into non-blocking communication, using

R. Keller et al. (Eds.): EuroMPI 2010, LNCS 6305, pp. 42–49, 2010.

memory protection to ensure data integrity [2]. We have also demonstrated how real-time network performance can be modeled and how the tradeoff between optimization savings and optimization overhead can be quickly decided at run-time [3]. The work presented here builds directly on these, extendig the run-time analysis of function calls and presenting the result to the user as potential source code transformations.

Itzkovitz and Schuster [4] introduce the idea of mapping the same physical memory multiple times into the virtual address space of a process in order to reduce the number of page protection changes.

Keller *et.al* [5] have integrated Valgrind memory checking with Open MPI. This does datatype analysis and parameter validation, but has higher overhead than our method presented here. It also does no direct overlap analysis.

Danalis *et.al* have developed ASPhALT [6], a tool that does automatic transformation at the compiler level, but only for specific recognized problems.

Breakpad [7] is a crash reporter utility, which includes the tools we use for extracting uniform symbol tables from many different platforms.

1.2 Outline

Section 2 details our run-time method for communication analysis and performance measurements. Section 3 shows how this information is post-processed to predict wallclock savings from manual optimization. Section 4 shows an example of performance optmization results. Sections 5 and 6 present our future work and conclusions.

2 Instrumentation Method

We inject a small library into the application using *LD_PRELOAD*, which intercepts all MPI and memory allocation function calls the application performs. When the application allocates memory, it is allocated from a shared pool which is physically mapped twice in memory. This allows us to manipulate the memory protection seen by the application, while simultaneously having a private, non-protected view of the same physical memory.

For all function calls, we record a *context*. The context is the calling address, the parameters for the function and the three previous stack return addresses. This ensures that the *context* is fairly unique to a single execution point in the application, even if the MPI functions are wrapped inside other functions in the application. A series of related calls (such as multiple send/recieve in a border exchange phase) are called a *chain*. Our injected library keeps track of *contexts* and *chains* in memory.

2.1 Startup

On application startup, we perform a number of quick benchmarks. We time memory protection overhead for various memory sizes. On some architectures this is very cheap, while on others it can be prohibitively expensive. However, we need to know the magnitude of this overhead to give accurate speedup predictions.

We also benchmark the point-to-point bandwidth of all the nodes for transfers up to $512kB$ to have a baseline prediction of request transfer time between any two nodes. The transfer time is linear with transfer size beyond $512kB$ for most systems.

2.2 Sends and Receives

When an MPI send request is issued by the application, our library will intercept the call and protect the application-visible memory of the request with *read-only* protection. We then start the request as a non-blocking request, operating on our private, non-protected view of the memory. The memory protection will ensure that the application does not alter the request until the transfer is finished. We note the *context* of the call and append it to the current *chain*.

When an MPI receive request is issued by the application, the call is likewise intercepted, and the application-visible memory is protected *no-access*. The request is then started as a non-blocking request on our private view of the memory and, exactly as for read requests, we note the *context* and append it to the current *chain*.

Any number of send requests are allowed to operate in parallel. When a receive request is issued on pages which already have active requests, the request buffer address and data type are analyzed. If the requests only have overlapping pages, but no actual overlapping cells, the call is started non-blocking as above. This is especially common for non-contigous datatypes, which can have interleaving cells without actually sharing any. When requests do overlap, we mark this as the end of the current *chain*. This means that we wait for all non-blocking requests to finish, unprotect the associated memory and start this receive request as a non-blocking request. The request will begin a new *chain*.

Note that we also intercept collective communication functions. As long as their buffer area does not overlap a non-blocking request, they are performed but otherwise ignored. Non-blocking collective communication will unfortunately not be available until MPI 3.0.

2.3 Page Faults

If the application accesses memory a page fault will occur. This typically occurs at the end of a communication phase when the application accesses data to compute. We handle this page fault, checking if the faulting address is one we have protected. If it is not, we restore the original page fault handler and allow the crash mechanism of the application to handle it. If it is, we mark the end of the current *chain*, wait for all requests to finish and unprotect the memory before allowing the application to finish.

2.4 Other Function Calls

We also intercept OS function calls, such as file reads, socket sends etc. While no non-blocking improvement is made, we need to ensure that their buffer area does not overlap any request we have made non-blocking.

Similarily, we intercept functions such as *MPI_Pack()*, which operate on buffers with datatypes. This allows us to analyze if the actual memory cells overlap or just the pages. As in the above, overlapping cells indicate the end of the *chain*, whereas overlapping pages without overlapping cells are ignored.

2.5 Performance Measurements

If the same *chain* of *contexts* is observed multiple times, we start performing speedup measurements. This is done by alternating function calls between two modes.

For both modes, the memory for the entire *chain* is protected when the first request in the chain is seen, with the most restrictive access of any request in the *chain*. This ensures that there will, ideally, be only two page protection calls; at the start of the chain and at the end. If we have mis-detected the *chain* and the application does not follow the pattern we expect, our library will identify this on the next function call or page fault. It will then unprotect the memory and mark the *chain* as broken.

In the first mode, MPI function calls are performed exactly as they are originally written. This gives us the original communication time for a chain of requests. In the second mode, all functions are optimized to their non-blocking versions. This gives a real-world measurement of the effect of overlapping multiple communication requests. Both values are recorded as part of the *chain*.

We also measure the time between the end of a chain and the start of the next request to have an estimate of the compute time between each chain of requests.

2.6 Non-blocking Verification

If the application already uses non-blocking MPI function calls, the memory areas for these functions are protected similarly to the blocking calls. The memory is not unprotected until all the non-blocking operations have been *MPI_Wait()*ed for.

This allows our method to verify race conditions from non-blocking communication, which is important if our recommendations are implemented in the original application.

3 Post-processing and Presentation

Once the application calls MPI_Finalize, we start our analysis pass. The list of *contexts* and *chains* are stored in shared memory, and an external application is started to analyze them. The use of an external application allows the injected library to stay lightweight.

3.1 Chain Merges

We analyze *chains* which follow each other with little or no appliction computation between them. If the first chain ended because of a receive request that overlaps memory areas, we scan the second chain for any requests that do not overlap the terminating receive request's memory. If found, these are noted as code that could be reordered and moved into the first chain.

3.2 Expected Savings

All *chains* are analyzed for their expected wallclock time savings, which our measurements have hopefully revealed. For *chains* which have not been seen sufficiently many times, we have no relevant performance measurements, so we use the network measurements to estimate the non-blocking performance.

Each *chain*'s savings is then multiplied by the number of times it is called, and results are presented in the order of most savings first. By default, transformations need to yield at least a 5% overall speedup to be reported.

3.3 Code Parser

Our analysis tool uses the symbol files for the application to find the correlation between calling address and program source code line. A source code parser will then open the relvant source code file and read the corresponding lines.

With the exception of the request pointer parameter, most blocking to non-blocking transformations have identical parameters. Our analysis tool will declare an array of requests at the start, rewrite the blocking communication functions to non-blocking, and use *MPI_Waitall()* at the end of the *chain*.

It should be noted that our code parser is not a full syntax tree parser, and fails if it encounters macros that expand to commas or other unusual code constructs. In this case, it simply reports the speedup it achieved.

The "before" and "after" results are presented to the user along with an explanation of what the transform does.

3.4 Scalability Analysis

Our analysis tool will dump the *chains* to disc when it is done. It can optionally be informed about the base problem size, which will be noted in the file. A number of such analysis passes, with varying problem sizes, constitute an analysis set.

Once a sufficiently large set has been obtained, the analysis can be switched to projection mode. In this mode, the analyzer will work on theoretical problem sizes. This is done by curve fitting the data size of the various requests to the problem size, and similarily curve fitting the execution time of each compute phase to the problem size.

We can extrapolate the expected wallclock time for any problem size, and our analysis tool supports prioritizing improvement areas based on the extrapolated results rather than actual measurements. This allows quick benchmarking on small problem sizes in a few minutes, with the goal to improve wallclock time for large problem sizes which take hours or days to compute.

4 Results

The majority of development of the instrumentation was done on 2D and 3D code, but we've chosen to include a 1D border exchange here so the output will fit in the paper. Here is an example of output when applied to such an application:

Example of Analysis Output

```
/* Chain #2, seen 11713 times: 60.3us per chain, 0.7 sec total savings.
 * Please change line 72-78 from
 */

MPI_Sendrecv(& local[1 * g], g, MPI_FLOAT, prev, 0,
             & local[(1+1) * g], g, MPI_FLOAT, next, 0,
             MPI_COMM_WORLD, &a);

MPI_Sendrecv(& local[1 * g], g, MPI_FLOAT, next, 0,
             & local[0 * g], g, MPI_FLOAT, prev, 0,
             MPI_COMM_WORLD, &b);

/* to */

MPI_Request req_72[4];
MPI_Status status_72[4];
MPI_Isend(& local[1 * g], g, MPI_FLOAT, prev, 0, MPI_COMM_WORLD, &req_72[0]);
MPI_Irecv(& local[(1+1) * g], g, MPI_FLOAT, next, 0, MPI_COMM_WORLD, &req_72[1]);
MPI_Isend(& local[1 * g], g, MPI_FLOAT, next, 0, MPI_COMM_WORLD, &req_72[2]);
MPI_Irecv(& local[0 * g], g, MPI_FLOAT, prev, 0, MPI_COMM_WORLD, &req_72[3]);
MPI_Waitall(4, req_72, status_72);
*(&a) = status_72[1];
*(&b) = status_72[3];
```

We've trimmed the output slightly, removing the per-transfer scalability analysis as well as the marker for the code-line which triggered the end of chain. The

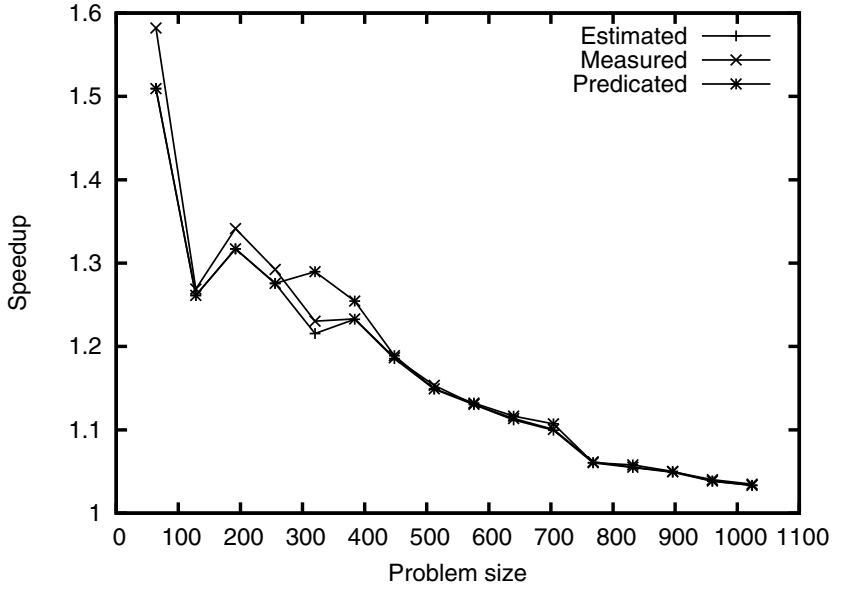

Fig. 1. Speedup predictions for the optimized code, measured optimized code and scalability-extrapolated results

transformed code aims to be functionally identical to the original code, including the assignment of MPI status variables.

Figure 1 shows the speedup of the original test program on various problem sizes, comparing the estimated and real speedup of the transformations. Also shown is the speedup for large problem sizes, estimated using only problem sizes $n \leq 256$. What is not clear from the graph is that the wallclock time predictor mispredicts the actual completion time for sizes of $n > 768$. These sizes are large enough that the computation no longer fits in L2 cache, which makes the computation phase more expensive. As this affects all implementations equally, the effect is not clear when looking only at speedup.

Please note that speedup of 2D and 3D cases are better, as there are more simultaneous requests which potentially reduce the effect of latency even more. Please see our previously published papers for optimization results of the underlying technique.

5 Current and Future Work

Our overlap analysis currently only does analysis for the problem size actually tried. There might be rounding or interpolation errors which cause overlap states to change as the problem size changes. Hence, while we know that the transformations presented to the user are safe for the specific problem size, we cannot guarantee they are safe for *any* problem size.

It would be interesting to do the entire analysis as a Valgrind module. This should enable more suggestions for code reordering. It would also allow us to switch from *chain* termination to waiting for only the request that is needed. This should allow larger overlap of computation and communication.

As it is, our code parser only works with C code. It would be interesting to extend this to use Open64 and hence work with a lot more languages.

The scalability predictor currently ignores the number of nodes, and assumes it will remain constant with the problem size as the only variable. This is naturally not the case for most applications, and it should be extended to cover runs both with varying number of nodes and varying problem sizes.

6 Conclusion

We have developed a method for run-time analysis of potential communication overlap improvements, with presentation of these to the user as transformations that are easily applied to their application source code. Each transformation includes the potential speedup of the application, an analysis of scalability, and also includes suggestions to replace the original code.

Acknowledgments. Thanks to NTNU and NOTUR for access to the computational clusters we have performed this work on. Thanks to Jan Christian Meyer and Dr. John Ryan for constructive feedback.

References

1. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, UT-CS-94-230 (1994)
2. Natvig, T., Elster, A.C.: Automatic and transparent optimizations of an application's MPI communication. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 208–217. Springer, Heidelberg (2007)
3. Natvig, T., Elster, A.C.: Using context-sensitive transmission statistics to predict communication time. In: PARA (2008)
4. Itzkovitz, A., Schuster, A.: MultiView and MilliPage – fine-grain sharing in page-based DSMs. In: Proceedings of the third USENIX symposium on operating system design and implementation (1999)
5. Keller, R., Fan, S., Resch, M.: Memory debugging of MPI-parallel Applications in Open MPI. In: Proceedings of ParCo 2007 (2007)
6. Danalis, A., Pollock, L., Swany, M.: Automatic MPI application transformation with ASPhALT. In: Parallel and Distributed Processing Symposium (2007)
7. Google: Breakpad - An open-source multi-platform crash reporting system, `http://code.google.com/p/google-breakpad/`