Thorvald Natvig

Automatic Optimization of MPI Applications

Turning Synchronous Calls Into Asynchronous

Trondheim, January 2006

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

Master's thesis
Programme of study: Master of Science in Computer Science

Supervisor: Anne Cathrine Elster, IDI

Innovation and Creativity

Thesis Goal

The introduction of MPI software for supercomputing clusters of regular computers has increased the number of HPC users. These users often have only rudimentary knowledge of parallel optimization techniques.

This thesis will focus on developing techniques for automatically optimizing MPI communication in users' applications. In particular, it will focus on turning synchronous communication calls into asynchronous calls which may allow overlapping communication with computations. The code developed should allow for this overlap without changing the dataflow of the original application.

Testcase codes comparing the automatic optimizations with manual optimizations using different optimization techniques will be developed and results presented.

Abstract

The availability of cheap computers with outstanding single-processor performance coupled with Ethernet and the development of open MPI implementations has led to a drastic increase in the number of HPC clusters. This, in turn, has led to many new HPC users.

Ideally, all users are proficient programmers that always optimize their programs for the specific architecture they are running on. In practice, users only invest enough effort that their program runs correctly. While we would like to teach all HPC users how to be better programmers, we realize most users consider HPC a tool and would like to focus on their application problem.

To this end, we present a new method for automatically optimizing any application's communication. By protecting the memory associated with MPI_Send, MPI_Recv and MPI_Sendrecv 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.

Also presented is an alternate method with less overhead based on recognizing series of requests made between computation phases. We allow the requests in such a chain to overlap with each other, and once the end of such a chain of requests is reached, we wait for all the requests to complete. All of this is done without any user intervention at all. The method can be dynamically injected at runtime, which makes it applicable to any MPI program in binary form.

We have implemented a 2D parallel red-black SOR PDE solver, which due to its alternating red and black cell transfers represents a "worst case" communication pattern for MPI programs with 2D data domain decomposition. We show that our new method will greatly improve the efficiency of this application on a cluster, yielding performance close to that of manual optimization.

Acknowledgements

Thanks to Dr. Anne C. Elster for being the primary advisor for this thesis. Without her continued pushing and encouragement, this thesis would have ended once the coding was complete and before a single page was written.

Thanks to Dr. Lloyd Clark for spelling and grammar advice.

Thanks to NTNU for providing me with sufficient hardware resources to do my experiments.

And a special (and honest) thanks to all those out there who are HPC users and not programming experts. Without being frustrated over the underutilization of resources your programs represent, we would have never gotten this idea. Hopefully, we now have at least a partial solution we are all happy with.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

When working as a computer scientist in the HPC field, we frequently work with and share resources with scientists from other disciplines. Quite often, such scientists will be more interested in the results of their computation than the efficiency and beauty of their code. The code may be written using only basic methods of communication and with a "works – don't touch" mentality that prevents further optimization once it works. While most code that enters production is eventually optimized by either the original scientists or a computer engineer assigned to the task, such optimization does not happen for code that only uses a smaller amount of time on the HPC machines and it almost never happens during the development stage of the code.

As computer scientists that are somewhat annoyed with this abuse of our marvelous HPC machines, we are left with the options of either teaching all other scientists in the world how to properly optimize their HPC code, or we can do the job for them. There are a lot of scientists out there, and most of them rightfully consider HPC just a tool that they already feel works well enough, so we have chosen to go with the second option. As optimizing all other code in the world by hand would be an impossible task, we have further chosen to go with automatic optimizations, specifically the reduction of communication overhead in parallel code.

1.1.1 Thesis goal

It is the goal of this thesis work to achieve a solution that automatically optimizes the communication of MPI programs. The solution should not require any user intervention at all to be enabled, and should ideally be fully usable on all MPI based parallel architectures.

It is important that the solution does not in any way alter the result of the MPI program, so we have to make sure any optimizations done do not alter the data flow of the program, only the communication. The only difference the user should notice should be somewhat better wallclock running time.

In this thesis, we present a solution that has been implemented and tested which satisfies these criteria and also serve as a base for further work for the Ph.D.

1.2 Problem statement

Optimizing for parallel machines requires extensive knowledge of both general optimization techniques and the specific machine. Any parallelization will add some overhead for communication. For example, when parallelizing Game of Life, an iterative 2D partial differential equation solver (PDE) or any other kind of grid-based domain, each iteration requires the borders of its subdomain to be exchanged with the "neighbors".

1.2.1 Parallel efficiency

The efficiency of a parallel implementation is measured by its speedup.

$$Sp = \frac{T_{sequential}}{T_{ParallelCompute} + T_{communication}}$$

where $T_{sequential}$ is the wallclock time the serial program used, $T_{ParallelCompute}$ is the wallclock time the parallel version uses for computation and $T_{communication}$ is the time it spends communicating. In most cases, $T_{ParallelCompute}$ will never be better than $T_{sequential}/P$, where P is the number of processors. Ideally, we'd like $Sp = P$, but that requires $T_{communication}$ to be zero. This is not possible in practice, but a lot can be done to make it smaller.

As shown in 3.3, a substantial reduction of communication time can be achieved by using overlapping communication to multiple neighbors at the same time.

1.3 Method overview

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.

1.3.1 Memory protection of program buffers

By using the memory protection features of the machine, it is possible to start the communication in the background while having the memory marked inaccessible to the program. Hence, if the program uses the memory in any way (such as computing some value), a fault will occur that can be intercepted, and at that time we wait for communication to complete before allowing the program to continue. While simple in theory, in practice there are a lot of challenges as the memory protection on most machines has a lot of shortcomings that need to be worked around, and there is also the overhead of the memory protection itself.

1.3.2 Detecting communication chains

With the basis of the information learned through the memory protection method, it's possible to build information about communication chains. A chain is defined as any series of communications that is followed by computation. For example, for 2D domain problems, the usual iteration is exchange north, exchange east, exchange south, exchange west, compute.

Once a chain has been detected and a high confidence established in its validity, we can avoid the overhead of memory protection by simply waiting for the entire chain to complete. This reduces the overhead drastically, as no tricks or copying have to be used to avoid buffer problems. While the memory protection alone provides good results, and indeed is the basis on which this and other optimization ideas are built, it is this chaining optimization that fully shows the potential of this method. As shown in chapter 6, wallclock times close to that of a manually optimized program are possible.

1.4 Thesis outline

Chapter 1 has been this brief introduction.

Chapter 2 will detail the background material for the project, including a brief background on MPI (2.1 and a very short summary of memory page protection (2.2). This chapter also has the list of related work in Section 2.3.

The test application to be optimized is described in Chapter 3, with detailed timings in Section 3.3.

Chapter 4 details the basic framework of memory protected asynchronous optimizations. It will explain how to inject into the target application (4.2), how to mark memory properly 4.3 and how to add and remove background requests (4.4). The handling of program page faults, which occur when the program accesses data that's still in transit, is explained in Section 4.5. The importance of keeping MPI_Status intact and a method to allow it with background transfers is explained in Section 4.6. Finally, Section 4.7 lists the functions we chose to override.

Chapter 5 shows how confidence in communications patterns allows us to build chains of requests (5.2). There are two major parts to this: Recognizing a chain (5.3) and using it (5.4). It is important to note that unlike basic paging, the chaining idea may break program data flow, and a theoretical example of this is shown and discussed in Section 5.5.

Chapter 6 shows results of running our method on the test program. Explanations for results on Snehvit, a highly parallel SMP machine is detailed in Section 6.2, and results for Norgrid, a Ethernet connected cluster, is detailed in Section 6.3. Section 6.5 has the discussion, showing strengths and weaknesses of our program based on the results and the development experience.

Chapter 7 contains all the future work we'd like to see done with this idea. In Section 7.1 we have the conclusion.

In Appendix A is a short installation and enduser's guide.

Most of the code in the text of this thesis is pseudocode, intended to show the method. The actual implementation is detailed in Appendix B, C and D, but the code in the implementation (Appendix E) is written for efficiency and not for easy reading.

Chapter 2

Background material

2.1 MPI

> MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.

MPI is the de-facto parallel message-passing environment, available on all modern HPC machines. The first version of the standard (1.0 [1]) was released in June 1994. Version 2.0 was finalized in July 1997, but most implementations still only contain partial support for all the features of MPI 2.0. MPI contains official bindings for C, C++ and Fortran, but people have ported the bindings to numerous other languages.

MPI focuses on messaging and contains both point-to-point and collective communication. Point-to-point is the simple "send this data to another processor", while collective communication covers communication between groups of processors, such as "find the minimum value", "distribute this array among processors" and so on.

In general, one would like to keep the amount of collective communication to a minimum as it usually scales as $O(logP)$, meaning the more processors one has the longer it takes to complete. Most iterations of 2D problems use only point-to-point communication with the neighbors. The background information here focuses on point-to-point communication, 2D cartesian groups and datatypes. For more information, please refer to Peter Pacheco's excellent "Parallel Programming with MPI" [2].

2.1.1 Cartesian groups

Many parallel programs are parallelized by decomposing the problem domain into subdomains and having each processor work on a specific subdomain. Between iterations, processors usually need to exchange some values with the processors handling the subdomains next to it's own ("neighbor" processors) to make sure the values along the borders of the domain are correct.

Many highly optimized HPC machines have a communication network organized as a 3D torus. Each node has a dedicated physical link to its 6 closest neighbors. Even if such a dedicated network isn't present, the MPI functions dealing with cartesian decomposition are highly flexible and help ensure that "neighbors" are as close as possible. For example, on a cluster of SMP machines, the MPI cartesian functions will help ensure that as many neighboring nodes as possible are on the same physical machine to minimize the amount of communication that has to pass over the communications network.

MPI makes using cartesian groups very simple, as it has methods for finding ideal decomposition of a n-dimensional problem of p nodes (MPI_Dims_create) as well as creation of cartesian communicators (MPI_Cart_create).

2.1.2 Datatypes

Quite often, the data to be sent is not simply contiguous chunks of memory, but data that is interleaved in some way. For example, for the red-black SOR, we would like to send "only red nodes" which is every 2nd memory location when sending the top or bottom rows, and every $2n$th memory location for the left and right borders.

To avoid having to copy the data in and out of temporary buffers, MPI provides the concept of datatypes which can describe any organization of data in memory.

2.1.2.1 Datatype extents

To use our idea of memory protection, we need to know precisely what area in memory a datatype occupies. For contiguous chunks of memory, this is simple, but for more advanced datatypes the problem becomes harder.

As an example, consider a plain $n \times n$ 2D area in memory where we want to send columns of data. The datatype will simply be a vector with stride n, defined using MPI_Type_vector. However, if we want to use this datatype to send many columns at once, we have to tell MPI that the datatype really is only 1 element large, so that if column 1 starts at position x, column two would start at position $x + 1$ and after the last element of column 1, which would be at $x + n(n - 1) + 1$. We do this by setting the upper bound of the datatype.

Unfortunately, this flexibility and these tricks make it much harder for us to determine the actual memory used by a datatype, and as explained in Section 4.3.1 we need this to find the minimum area to protect.

2.1.3 Point-to-point communication

MPI provides two basic facilities for point-to-point communication, sending and receiving, implemented as MPI_Send and MPI_Recv. Both of these take arguments for a pointer to the start of the area to send, a datatype, the number of elements, the node to send/receive to/from, a message tag and the communicator to use as parameters. In addition, MPI_Recv takes a pointer to a status variable that can be queried.

While there is only one MPI_Recv, there are many variations of MPI_Send. Most MPI implementations feature a buffer of some kind, and if there is enough room in the buffer, MPI_Send will usually return immediately after copying the data to the buffer without waiting for the receiving end to issue a MPI_Recv. To avoid this, you can use MPI_Ssend which always waits for the receiver to accept the data. If you know the other end has already posted a receive, you can use MPI_Rsend (ready send). To use your own buffering, use MPI_Bsend.

In addition to this, all of the above are also available in asynchronous modes, so called "immediate" mode (as it returns immediately to the caller without waiting). So there is MPI_Isend, MPI_Issend, MPI_Irsend, MPI_Ibsend as well as MPI_Irecv. The immediate mode versions return a request variable which can later be queried for completion (MPI_Test) and waited for (MPI_Wait family of functions). The idea behind asynchronous communication is simple; allow the program to continue computing while communication takes place in the background.

If this wasn't convoluted enough, there is also the problem of deadlock. Let's say you have two processes, A and B, which first send a bit of data to the other processor and then receive. Using basic MPI_Send, this works as long as the buffer is large enough. However, if the datasize is too large for the buffer, both processes will hang waiting for the other to finish its send and start receiving. To make it easier to avoid this problem, MPI provides MPI_Sendrecv, which is MPI_Send and MPI_Recv merged into one to make it easier to avoid deadlocks.

2.2 Memory and page faults

Page Fault: An interrupt that occurs when a program requests data that is not currently in real memory. The interrupt triggers the operating system to fetch the data from a virtual memory and load it into RAM.

An invalid page fault or page fault error occurs when the operating system cannot find the data in virtual memory. This usually happens when the virtual memory area, or the table that maps virtual addresses to real addresses, becomes corrupt.

– Webopedia on "Page fault"

All modern machines which implement virtual memory have some kind of page protection. The basic mechanism is to divide the address space into evenly sized pages, and map each virtual page to a corresponding page in physical memory. A page may also be marked as "not available" in some form, causing an interrupt to occur. Operating systems use this to fake having more memory than is actually present, by moving infrequently used pages of memory to disk and marking these as inaccessible. If the program accesses the page, an interrupt occurs and the OS copies the data back info physical memory and then resumes program execution.

When a program accesses memory, the virtual to physical mapping happens through page tables. In its most basic form, a page table is a linear set of values, so that the nth index in the table contains the page number of the physical page for virtual page n. On modern OSes and CPUs which often use discontinuous memory allocation (so that pages 37, 100 and 192 might be used, but none in between), a tree structure of pages is used instead. Basically, each level in the tree correspond to a certain number of bits of the virtual address, and if a tree node is missing, then the program is accessing memory it didn't allocate.

Different architectures support different page sizes. For example, the x86 architecture supports 4KiB and 4MiB page sizes while the Itanium2 defaults to 16KiB. Smaller page sizes mean better granularity, but also imply larger page tables which take more space and have a larger overhead when they need to be updated for large chunks of memory.

2.2.1 Manipulating page tables

Page tables can be used for other things than just virtually extending memory. It is also used to map files into memory (mmap'ing) and making sure a program only modifies its own memory. On a machine without any page protection, a program can intentionally or unintentionally (through bugs) modify the memory of other programs, causing them to fault.

Modern operating systems allow a user to manipulate the page table for any pages that are allocated. On UNIX, this is done through the mprotect() system call, which takes a pointer to the area to protect, a size and set of flags of allowable access. If used to remove privileges to a page that is subsequently accessed, a segment violation fault will occur. This can be intercepted by writing a signal handler to handle the signal. This method is fully portable to all POSIX compliant OSes.

2.2.2 Shortcomings of page protection

The smallest amount of memory that can be protected is a single page, meaning 4KiB on the most favorable architectures. Unfortunately, this is way too large to protect a single variable or a even a small data structure – one will unavoidably end up protecting a lot of other data as well.

Furthermore, while the standard specifies individual protection access for read, write and execute, it is frequently seen that read privilege grants execute, and there are very few architectures which allow write-only pages. It is important to be aware of these shortcomings, as a mprotect() call to create a write-only page will succeed, but might end up giving you a writable, readable and executable page.

2.2.3 Alternate methods of page manipulation

A problem encountered in our implementation is the fact that you can't just mprotect() the memory from the user program. The MPI implementation runs in the same executable context as the user program, so if you change the access parameters, the MPI implementation will not be able to access the data either and hence cannot complete its background data handling.

In this implementation, we have solved this with manual buffering of data, but a more elegant solution is to run the MPI background transmission in a separate thread, and modify the kernel to allow per-thread pagetable manipulations. Unfortunately, this requires some non-trivial changes to the kernel and is limited to kernels for which we have the source code. It is also quite difficult to get acceptance for installing such changes on production machines, defeating our implementation's purpose of being generally applicable.

2.3 Previous and related Work

From our research, we cannot see that the idea of runtime tuning MPI requests using page protection has ever been tried before. As such, there is no direct previous work using this method. There are, however, various related works that have inspired us and we will detail those here.

2.3.1 MPICH and LAM

MPICH [3, 4, 5] and LAM [6, 7] are freely available MPI implementations that enable anybody to build their own "supercomputer" using a group of machines connected with simple Ethernet or low latency dedicated interconnect.

The availability of these free implementations has been used as a starting point for many research projects, as they allow ideas for improvements to be tried without having to write an entire MPI implementation.

2.3.2 Jumpshot and TAU

Jumpshot [8] is a classic postmortem performance analysis tool and comes with MPICH. By compiling a program with MPI trace enabled, a file will be generated containing the call trace of the program. This can be used to visualize the communication pattern of the entire program, and one can easily find the areas of the program where optimization effort has the most impact.

Jumpshot lets the user see the time period which is "wasted" between the start of a MPI_Recv call and when data actually starts arriving, and it was this wasted time that inspired us to investigate the effects of using MPI_Isend on clusters.

TAU (Tuning and Analysis Utilities [9]) is another and more comprehensive analysis package, covering much more than basic MPI. Instead of using the trace file, TAU inserts itself into the program (which need not be an MPI program) and generates extensive profiling which can be easily visualized.

However, all of these analysis tools have one weakness: They assume the user will understand and care about the information they provide. While the benefits of checking the analysis and implementing its suggestions would be immense, there is often an equally large benefit in using profile-feedback compilation, and this is also not done.

2.3.3 D.U.M.A.

D.U.M.A. (Detect Unintended Memory Access [10]) is the successor to *Electric Fence - efence*. It is intended to detect buffer overruns on the heap, and does this by allocating an extra page for all memory requests and protect this page. It can work in one of two modes:

- underrun protection, where the page before the returned memory pointer is protected and the returned pointer is aligned to the page boundary.
- overrun protection, where the page after the returned memory is protected and the returned pointer is such that $pointer + size$ is at a page boundary.

Note that efence cannot detect both underruns and overruns at the same time unless the requested allocation size divides evenly with the pagesize.

Electric Fence was one of the first programs to use the same dynamic injection techniques we use, which allows it to be used to debug binaries for which no source is available.

Lately Valgrind [11] has become more popular than D.U.M.A., since it can detect both heap and stack overruns and underruns along with numerous other memory problems. Valgrind does this by interpreting the code as it runs, meaning it results in a massive slowdown. So while Valgrind provides better results for debugging, its techniques are of little use for us.

2.3.4 Internal MPI Optimizations

Numerous work has been done on optimizing the individual MPI functions. Some of these are inspired by the ATLAS (Automatically Tuned Linear Algebra Software [12]) idea of taking a basic routine and trying thousands of small variations until you find the specific set of parameters that is perfect for the architecture you are compiling on. Faraj and Yuan [13] have presented such a method for automatically optimizing the MPI Collective subroutines, and Østvold has presented numerous ways of timing collective communication [14].

Ogawa and Matsuoka [15] use compiler modifications to optimize the 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 [16], such optimizations can be extended to all function calls and not just MPI Calls.

There are numerous other approaches, but they all assume the actual application is already optimized, while we assume it is just a regular application.

Chapter 3

Test program

During the development of this thesis, we have used a simple test program solving a 2 dimensional partial differential equation (2D PDE) using the parallel red-black successive over-relaxation algorithm. In this chapter, we will present the necessary background material for this test program, as well as test results.

3.1 PDE Problem

We have chosen a simple Laplacian problem on the unit square. To visualize this type of problem, think of a sheet of metal kept at $0\,°\mathrm{K}$ on 3 edges, and the last edge is kept at $300\,°\mathrm{K}$. After the system stabilizes, what will be the temperature in the middle of the sheet?

Our specific problem domain is as follows:

$$\begin{array}{ll}
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 & \\
u(x,1) = 1 & 0 \leq x \leq 1 \\
u(x,0) = -1 & 0 \leq x \leq 1 \\
u(0,y) = u(1,y) = -1 & 0 \leq y \leq 1
\end{array}$$

3.1.1 Discretizing

To solve this iteratively, the system is discretized with m points in the x and y direction, and for each iteration the approximate value of $u_{i,j}$ is computed as

$$u_{i,j}^{n+1} = \frac{1}{4}(u_{i-1,j}^{n} + u_{i+1,j}^{n} + u_{i,j-1}^{n} + u_{i,j+1}^{n}), i = 1..m, j = 1..m$$

This is the Jacobi iteration. To speed up convergence, it is possible to use the new values of $u_{i,j}$ as soon as they are available, which gives us the Gauss-Seidell method. Finally, by observing that each iteration brings us a small step closer to the solution, it is possible to over-relax to speed up convergence. This method is called Successive Over-Relaxation, and can be written as

$$u_{i,j}^{n+1} = \frac{\omega}{4}(u_{i-1,j}^{n+1} + u_{i+1,j}^{n} + u_{i,j-1}^{n+1} + u_{i,j+1}^{n}) - (1-\omega)u_{i,j}^{n}$$

where ω is the amount of over-relaxation. A good default value of ω is 1.8, but an ideal implementation will have ω vary with the size of the problem and the iteration number using Chebyshev acceleration [17].

3.1.2 Red-black SOR

In SOR with a 4-point stencil, each point requires the updated value of the point above and to the left, meaning the lower right point in an iteration recursively depends on all other points. This makes it very hard to parallelize.

An alternate, but equally effective algorithm was therefore developed. If one marks each point either red or black after a checkerboard pattern, it is easy to see that each red point depends only on black points, and each black point depends only on red points. This is illustrated in Figure 3.1.

Each iteration now computes all the red nodes, then uses these updated values to compute all the black nodes.

3.2 Test program

Our test implementation is of red-black SOR using 2D decomposition. The problem is stated in 3.1.

We chose the SOR PDE solver as it's a typical usage of a iterative 2D algorithm, and the same method is easily extended to 3D. It also has stresses the datatypes of MPI, as it needs to exchange red and black nodes separately.

We have chosen a 2D domain decomposition among the processors, meaning the data points are spread over processors organized as a 2D grid. As long as P can be factored into two numbers, this needs less bandwidth than the 1D case ($4\frac{n}{\sqrt{P}}$ vs $2n$). 1D would also avoid the problem of requests with overlapping memory areas (the left/right edges), a case we need to both test and validate as working.

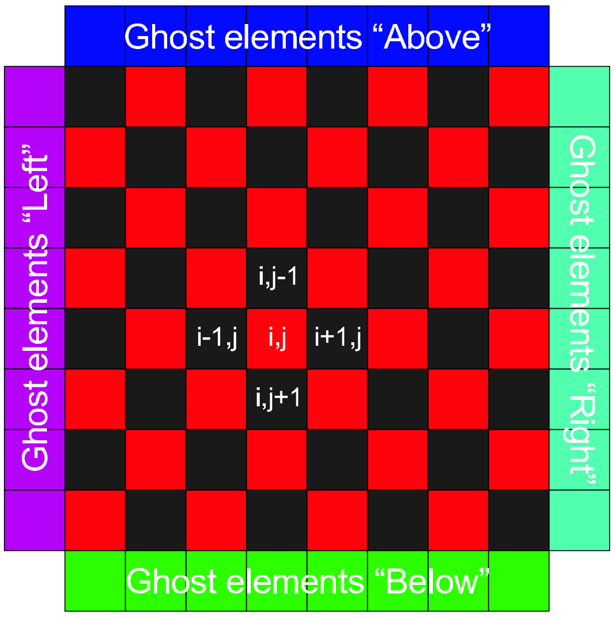

Figure 3.1: Red-black SOR domain marking and ghost elements.

Each node in the computation has allocated memory for it’s own subdomain along with a layer of "ghost cells" along the edges. These ghost cells will receive the values from the neighboring nodes, as updating the elements on the edges requires the values from the subdomains belonging to the neighbors.

The test program contains a serial implementation as well as 3 parallel implementations. In the following discussion and algorithm descriptions, we have focused on the communication patterns as that is the focus of this thesis and the computation part is mostly unchanged from the serial code. For a complete example, see the code accompanying the thesis.

The most basic version uses just MPI_Sendrecv, and can be seen in Algorithm 1. For each phase of red or black, the elements along the edges of the other color are exchanged, and then all elements of the same color are updated. It will not deadlock as there is always a receiver for every send, and MPI takes care of sends/receives going to non-existent neighbors becoming no-ops (for the nodes that have subdomains along the edges of the whole domain). This algorithm is the simplest and easiest to write, but also has the worst performance. It does 8 MPI_Sendrecv pairs, but each pair has to wait for the previous to complete before the next can start.

A more advanced version using MPI_Isend and MPI_Irecv can be seen in Algorithm 2. It communicates with all neighbors in parallel, but does not overlap any

Algorithm 1: Basic parallel red-black SOR PDE Solver.

```
1  begin
2      for each iteration do
           // Red phase
3          MPI_Sendrecv(Send top black elements to "above" node, receive
           bottom black ghost elements from "below" node) ;
4          MPI_Sendrecv(Send bottom black elements to "below" node,
           receive top black ghost elements from "above" node) ;
5          MPI_Sendrecv(Send left black elements to "left" node, receive right
           black ghost elements from "right" node) ;
6          MPI_Sendrecv(Send right black elements to "right" node, receive
           left black ghost elements from "left" node) ;
7          Update all red elements ;
           // Black phase
8          MPI_Sendrecv(Send top red elements to "above" node, receive
           bottom red ghost elements from "below" node) ;
9          MPI_Sendrecv(Send bottom red elements to "below" node, receive
           top red ghost elements from "above" node) ;
10         MPI_Sendrecv(Send left red elements to "left" node, receive right
           red ghost elements from "right" node) ;
11         MPI_Sendrecv(Send right red elements to "right" node, receive left
           red ghost elements from "left" node) ;
12         Update all black elements ;
13 end
```

communication with computation. As can be seen, this algorithm is slightly more complex and requires use of more advanced MPI functions.

Algorithm 2: Parallel red-black SOR PDE Solver using overlapping communication with multiple neighbors.

```
begin
    for each iteration do
        // Red phase
        MPI_Irecv(top black ghost elements from "above" node) ;
        repeat for bottom, left and right ;
        MPI_Isend(top black elements to "above" node) ;
        repeat for bottom, left and right ;
        MPI_Waitall(all 8 requests) ;
        Update all red elements ;
        // Black phase
        MPI_Irecv(top red ghost elements from "above" node) ;
        repeat for bottom, left and right ;
        MPI_Isend(top red elements to "above" node) ;
        repeat for bottom, left and right ;
        MPI_Waitall(all 8 requests) ;
        Update all black elements ;
end
```

The "perfect" version, seen in Algorithm 3, overlaps communication and computation, and should in theory be "as good as it gets".

3.3 Test results

The results of running the test program on the Norgrid cluster (Dell 750, 3.4 Ghz Pentium 4, Gigabit Ethernet) and on Snehvit (Altix 350, 1.4Ghz Itanium 2, NUMAlink) are found in Table 3.1 and Table 3.2 respectively. The Dell cluster has a high speed CPU which does very well on the semi-unstructured access patterns of the red-black SOR, but it has a very slow interconnect. On the Altix, which is a shared-memory machine, copying data to another processor is about as fast as copying it to memory. It therefore has excellent latency, but still suffers from limited bandwidth when all processors communicate at once.

Algorithm 3: Parallel red-black SOR PDE Solver using overlapping communication and computation.

```
begin
    MPI_Irecv(top black ghost elements from "above" node) ;
    repeat for bottom, left and right ;
    MPI_Isend(top black elements to "above" node) ;
    repeat for bottom, left and right ;
    for each iteration do
        // Red phase
        MPI_Waitall(black border exchanges) ;
        Update red elements along the edges ;
        MPI_Irecv(top red ghost elements from "above" node) ;
        repeat for bottom, left and right ;
        MPI_Isend(top red elements to "above" node) ;
        repeat for bottom, left and right ;
        Update all interior red elements ;
        // Black phase
        MPI_Waitall(red border exchanges) ;
        Update black elements along the edges ;
        MPI_Irecv(top black ghost elements from "above" node) ;
        repeat for bottom, left and right ;
        MPI_Isend(top black elements to "above" node) ;
        repeat for bottom, left and right ;
        Update all interior black elements ;
    MPI_Waitall(black border exchanges) ;
end
```

Method	Timing	$n = 128$	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$	$n = 4096$
MPI_Sendrecv	Wallclock	1.14	3.01	3.01	3.49	6.05	16.98
	Compute	0.01	0.05	0.19	0.80	3.27	12.97
	Communicate	1.12	2.96	2.82	2.70	2.76	4.01
MPI_Isend	Wallclock	0.36	0.99	1.00	1.97	4.48	14.09
	Compute	0.02	0.05	0.19	0.80	3.32	12.92
	Communicate	0.35	0.94	0.81	1.16	1.17	1.19
Full overlapping	Wallclock	0.35	0.99	1.00	1.15	3.87	13.84
	Compute	0.02	0.05	0.20	0.86	3.53	13.32
	Communicate	0.34	0.94	0.80	0.28	0.34	0.52

Table 3.1: Average iteration execution time in milliseconds of regular Sendrecv, Isend and overlapping communication and computation on 16 nodes of the Norgrid(Dell) cluster.

Method	Timing	$n = 128$	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$	$n = 4096$
MPI_Sendrecv	Wallclock	0.12	0.22	0.59	1.92	8.00	106.68
	Compute	0.03	0.10	0.39	1.60	6.61	56.22
	Communicate	0.10	0.12	0.20	0.32	1.38	50.45
MPI_Isend	Wallclock	0.11	0.21	0.55	1.87	7.80	106.18
	Compute	0.03	0.10	0.39	1.60	6.60	56.03
	Communicate	0.09	0.11	0.16	0.27	1.21	50.15
Full overlapping	Wallclock	0.11	0.20	0.54	1.80	7.36	105.63
	Compute	0.03	0.10	0.38	1.60	6.61	56.97
	Communicate	0.08	0.10	0.15	0.21	0.75	48.67

Table 3.2: Overrate iteration execution time in milliseconds of regular Sendrecv, Isend and overlapping communication and computation on 16 nodes on the Snehvit(Altix) machine.

3.3.1 Test methodology

The tests were run such that each method had 20 iterations, the execution times of the iterations sorted and the median picked. This avoids the outlier values that will always be present. For example, it seems most MPI implementations delay allocation of buffers and final commit of datatypes until they are actually used, meaning the first iteration may be as much as 20 times slower than the median one.

All tests are run on 16 nodes. This is to make sure there are some nodes that have to communicate in all 4 directions. For example, with just 4 nodes, all nodes will have 2 edges that it are not shared with another node, so the amount of communication is halved. With 16 nodes, the 4 nodes in the middle all have to communicate with 4 different neighbors.

The most challenging part of the tests were making sure no external factors influenced our timings. On the Altix machine, this meant running in the middle of the night when no interactive jobs from other users could influence the execution, and on Norgrid it meant repeated runs until we had a run where other users didn't use the shared switch.

3.3.2 Analysis of Norgrid results

Norgrid has an interconnect with very high latency. For small grid sizes, the nodes spend more time waiting for communication than computing.

On the cluster, programs will get a substantial performance boost by using MPI_Isend to multiple neighbors. As each node just has a single Ethernet cable to connect to the backbone, this is a somewhat unexpected result. It is quite logical though; the high latency means that the program has to wait for the 'receive' part of MPI_Sendrecv before it can start the next exchange. By using 4 Isends and Ireceives, we only have to wait once, effectively quartering the overhead of the network latency.

Another interesting observation is that, while the fully overlapping version does overlap communication and computation, the actual computation time is increased somewhat as the CPU now has to service network interrupts while computing. However, the overhead of servicing the network card is more than compensated for by the decrease in communication wait time. Especially noteworthy is the change from $n = 512$ to $n = 1024$. Here the increase in computation time is enough to overlap the majority of the communication, meaning less time is spent purely waiting for $n = 1024$ than for $n = 512$.

3.3.3 Analysis of Snehvit results

Snehvit is a SMP machine, as has very low latency on messages. This is clearly seen for the cases where n is low. For example, at $n = 128$, it is 10 times faster than Norgrid using basic MPI_Sendrecv.

Something that is quickly observed is that Snehvit doesn't benefit much from overlapping communication with multiple neighbors or from overlapping communication and computation. The reason is simple; sending and receiving messages are just memory copy operations, and the bottleneck becomes the CPU bus. The single CPU can either copy memory or compute, but not both. The small gain that is seen, results from the CPU just copying data to cache, and the cache will be synchronized in the background.

Snehvit uses more time for computation than Norgrid does. Without carefully tuned algorithms written by experts on the Itanium2, the machine is unable to deliver the performance it is capable of. Indeed, when n becomes large enough that the problem no longer fits in L1 cache, execution time rises drastically, to the point that Norgrid is nearly 10 times faster at $n = 4096$.

3.3.4 Expectations of new method

Figure 3.2 shows the efficiency of the different methods on both machines, where efficiency is measured simply as $1 - T_{Wall}/T_{Communicate}$, so a high efficiency means less time wasted waiting for communication. Figure 3.3 shows the speedup of switching to different methods than MPI_Sendrecv on the two architectures.

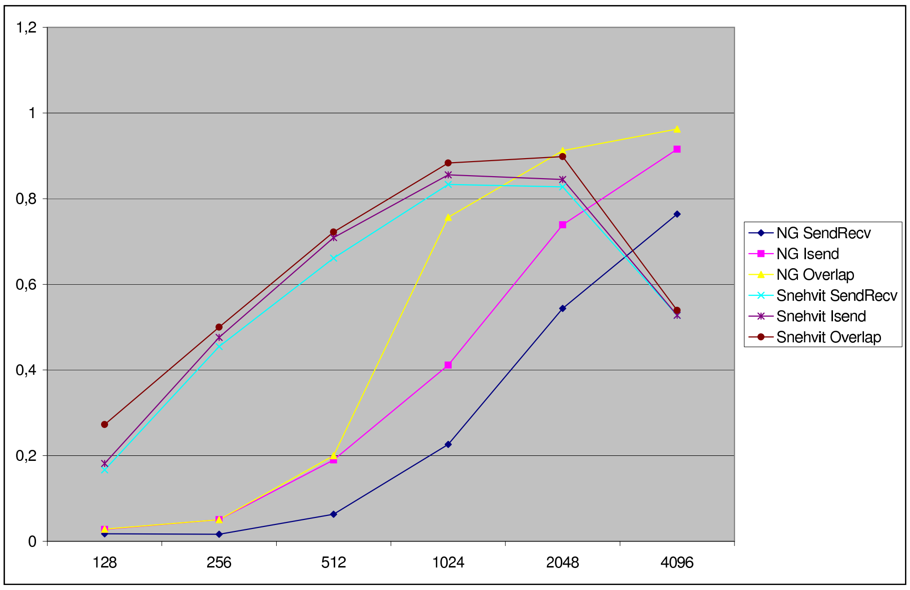

Figure 3.2: Efficiency of MPI_Sendrecv, MPI_ISend and full overlapping on both Norgrid and Snehvit with different grid sizes.

Based on the timing from the two very different HPC machines, we can conclude that overlapping communication with multiple neighbors results in a negligible to substantial improvement, but it is always an improvement.

Overlapping communication with computation in a perfect way results in even better results, but the improvement will be less than that seen from the multiple-neighbor case.

The perfect overlapping requires the data flow of the program to change, hence it requires either a rewrite of the application or a method capable of analyzing the application and determine its goal. Such a method is called "analysis and

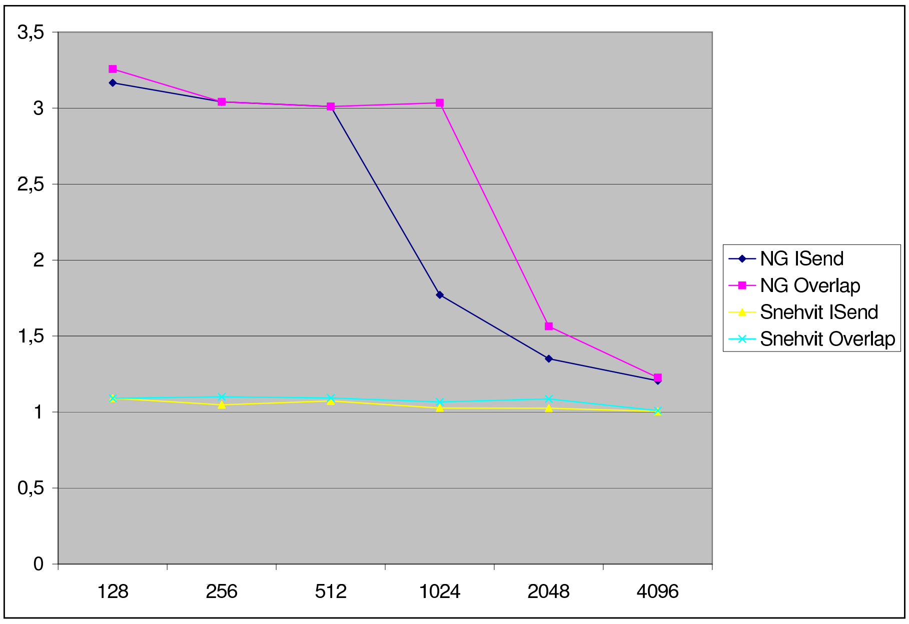

Figure 3.3: Speedup of using MPI_ISend or full overlapping instead of just MPI_Sendrecv on both Norgrid and Snehvit with different grid sizes.

understanding" and is, for the time being, something only a human mind can do. While we can use a lot of tricks to get closer to the timing results of this method, it is practically impossible to automatically tune the simple MPI_Sendrecv program into this.

It is fully possible to tune the basic MPI_Sendrecv into something close to the MPI_Isend case. A method to achieve this is presented in the following chapters.

Chapter 4

Pagefault-assisted request tracking

4.1 Overview

The pagefault assisted request tracking is the backbone on which this method builds. The idea is to mark pages that are in use by background processes as inaccessible to the program, and then deal with it when the program accesses one of them.

The first problem is getting our method inserted into the program, and this is discussed in Section 4.2. Then comes the problem of requests with overlapping memory areas as well as requests with non-overlapping memory but which still share the same pages in memory, and this is discussed in Section 4.3. To avoid page-protecting data initialized outside the targeted application, it's necessary to track memory allocation, and a method for this is explained in 4.3.3. Section 4.4 explains the methods used to add and wait for requests as necessary, and section 4.5 explains how the pagefault signal handler works. Finally, section 4.7 details the MPI functions we have chosen to override and how.

4.2 Injecting into the process

There are two separate methods for a program to link to the MPI library. It can use static linking, in which parts of the MPI library are included in the application binary, or it can use dynamic linking, where the program loads and links to the MPI functions at runtime.

There are also two separate methods for injecting our method into a program. It may be done at compile time, in a way transparent to the user but still requiring a compile, or it may be done at runtime by changing things "on the fly".

4.2.1 Compile time injection

Compile time injection is facilitated by including mpii.h instead of mpi.h, and linking to injlib.a. For this thesis project this is done manually, but it is a trivial task to change the mpicc wrappers used to compile mpi programs to enable this for an entire machine.

When using static injection, the header defines the functions we want to override,such as MPI_Init, to be aliases for function names such as Inj_MPI_Init. That way, only code compiled with our new header will use the overridden functions. To find the address of the original function which we need to chain to, it’s just to take the address of the original function name. So taking the address of the symbol MPI_Init will indeed return a function pointer to the original.

For compile time injection it doesn’t matter what way the MPI library is included.

4.2.2 Runtime injection

Runtime injection is only possible if the MPI library is included dynamically. If it’s included statically, the address of the MPI functions might not be resolvable. Attempts were made to work around this by analyzing the binary and looking for function signatures identical to the MPI library, but advanced compilers with cross-object inlining ruins this idea, and there was also the chance of misdetection meaning we would end up replacing the programs fopen() with our MPI_Init, which was catastrophic.

When using runtime injection on a dynamically linked process, we use the LD_PRELOAD method. LD_PRELOAD is an environment variable which specifies a library that is to be loaded for every process, and it is put first in the search chain of functions to resolve. We therefore name our overridden functions the same as the functions they should override. The problem then becomes finding the original function, since taking the address of the function name will return our own function. We solve this by using dlsym() and the RTLD_NEXT parameter, which means "resolve this somewhere down the chain, but not here". This means the dynamic loader will go down the list of libraries and resolve the function name in the original MPI library (which hopefully is also linked against).

4.3 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.

See Figure 4.1 for an example used in the following discussion. This is 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.

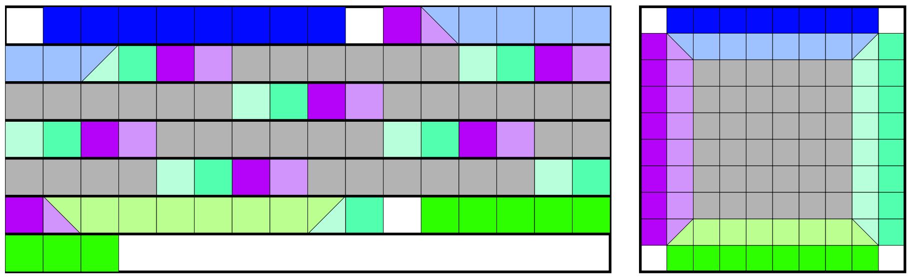

Figure 4.1: Left: Memory cells used by border exchange in Jacobi PDE Solver. Right: Cells shown in structured form.

4.3.1 Overlapping elements

Accurately detecting overlapping elements is hard. At the moment, we use a simple approach of testing if the lower and upper bound of the requests overlap. This has a major shortcoming though, as requests that do not share a single actual address might still have their extents overlap. Consider the two columns to be sent to the left and right side respectively; they have overlapping extents but do not have any addresses in common.

Since MPI datatypes can be arbitrarily complex, the only sure way to detect if two operations overlap is to track all calls to the MPI_Type family of functions and build a memory stencil. When a request starts, the stencil is repeated *count* times, and then compared to the stencils of other requests. Unfortunately, this requires $ub - lb$ bits of memory for both requests as well as $(ub - lb)r$ comparisons (where r is the number of outstanding requests). As such, the overhead becomes larger than the gain, and we stay with the simple solution of checking extents.

4.3.2 Overlapping pages

Detecting the overlapping pages is considerably easier, and unfortunately the detection is frequently true. Unless the application took care and aligned every row of data on a page boundary, there will be major page overlapping. As a goal of this new method is to be fully user transparent, page-alignment does not occur in practice.

If a new request is added which uses pages already protected from use, it can temporarily unprotect them, copy the data it needs, and then reprotect. The problem is actually in waiting for requests to complete, as a page cannot be unprotected before all requests dealing with it are finished. Failing to do this means the program could read data before it was actually there or write to data that hasn't been sent yet, violating data flow.

4.3.3 Overriding malloc() and free()

A problem that was discovered on the SGI implementation of MPI is that it frequently dynamically allocates memory, and does so by just placing data at the top of the program heap. If this data is placed in an area that we have protected, we have a logical deadlock, as our method will not release the page until the MPI call is complete, and the MPI call itself needs that data. The result is a program crash.

To avoid this problem, we override malloc() so that they always return page-aligned addresses, and we increase the size of the allocated area so it fills the data page. This way we can ensure that the MPI library's use of the heap does not interfere with that of the application or our method.

free() is overridden too, as it needs the original allocated memory address and not our page-aligned one.

Allocated memory pages are marked in a map, and the overridden MPI functions will only optimize the call if the memory area starts in a marked page. This avoids the problems of programs that use buffers allocated on the stack or as static data, which again would cause logical deadlocks (especially the stack).

This method of ensuring page-granularity of malloc() requests can cause massive increase of memory usage if there are hundreds of small malloc()s. For this reason, the overridden malloc() and free() functions are not used before the call to MPI_Init. This is especially important for runtime injection, as we will override malloc() in all programs, including mpiexec and the shell itself, and we do not want to bloat memory allocation in these programs. Most scientific MPI programs and development code we have looked at start with the call to MPI_Init and then allocate memory in large chunks, so the bloat should be a minor problem.

4.4 Adding and waiting for requests

When a new request is made, it is added to the set of active requests. The algorithmic overview of this process is shown in Algorithm 4. The accompanying algorithm for waiting for requests is shown in Algorithm 5.

Algorithm 4: Turning synchronous request into asynchronous and adding to set of active requests.

```
   input: $is_send$ signifying send or receive request
1  begin
2      overlap ← false ;
3      for each other request r do
4          if r has overlapping elements with this request then
5              Wait for r ;
6          if $is\_send$ and r overlaps our pages then
7              overlap ← true ;
8      if overlap then
9          Unprotect memory area ;
10     Allocate temporary buffer large enough to hold data ;
11     if $is\_send$ then
12         Copy data from memory into buffer ;
13     Mark memory inaccessible ;
14     Start background transfer ;
15     Add request to set of active transfers ;
16 end
```

Algorithm 5: Waiting for requests to finish.

```
   input: A request w to wait for
 1 begin
 2     Remove w from set of active transfers. ;
 3     Call MPI_Wait for request of w. ;
 4     for each other request r do
 5         if r overlaps pages of w then
 6             Wait for r. ;
 7     Mark memory area of w accessible;
 8     if w is a receive request then
 9         Copy data from buffer into memory.;
10 end
```

There is one complication not shown in the algorithm, which is pairs of requests such as those from MPI_Sendrecv. For these, we mark the accompanying request, and we do not wait for that request to finish even if our primitive test for overlapping elements succeed. The MPI specification clearly states that the send and receive buffer should not have overlapping elements, so any violation of data flow would also occur without optimization and as such we can ignore the test. Without this added logic, calls to MPI_Sendrecv might deadlock when it shouldn't.

4.5 Page fault handling

The OS mechanism of page fault signal handlers is simple and effective. Upon a page fault, the handler is called with a list of parameters, including the memory address at which access was attempted. When the signal handler returns, the instruction that caused the fault is attempted again and program flow continues.

Our method handles page faults as explained in Algorithm 6. The hardest part here is gracefully handling "real" page faults caused by bugs in the application. If that happens, we need to finish all our outstanding requests and restore the original handler. When we return, the original instruction will immediately cause a page fault again, but this time the original handler will handle it and the application will crash properly.

We do not modify the page tables at all in the page handler, the function to wait for requests handle that for us.

Algorithm 6: Handling page faults.

```
   input: address which caused fault
1  begin
2      found ← false ;
3      for each active request r do
4          if r pages include address then
5              Wait for r. ;
6              found ← true ;
7      if not found then
8          Wait for all active requests ;
9          Restore original handler ;
10 end
```

4.6 Tracking MPI_Status

A problem that became apparent after a bit of testing was that the original idea violated the concept of the status parameter to MPI_Recv. As the request is most likely not finished yet, trying to figure out the number of bytes received becomes meaningless.

The solution to this problem was unexpectedly simple. As MPI defines all querying of status through functions, we simply override these functions. When a receive request is added, we map the address of the status variable to the request, and if a call is made to query the status variable, we wait for the request to finish before chaining to the original function.

With this added logic to handle MPI_Status, our program ensures that the logical data flow of the program remains exactly the same as without our optimizations, meaning that our optimizations cannot cause the program to misbehave.

4.7 Overridden functions

We will here provide a short detail of the functions we override as well as the ones we leave unmodified.

4.7.1 MPI_Init

MPI_Init is used as our entry point into the program. It is here that we initialize the function pointers to the original functions, as well as initialize our data tables. In addition, it is only after this initialization that the page-alignment of malloc() is enabled.

After chaining to the original MPI_Init, we initialize our signal handler for page faults (or segment violations which they are still archaically called). It is necessary to call the original MPI_Init first, as most MPI implementations add their own signal handler which simply returns an error to the user, but does not check for any previous signal handlers. This MPI-specific signal handler would then replace our own, meaning the program will page fault and crash on the first access to the memory of a request.

4.7.2 MPI_Send and MPI_Recv

MPI_Send adds a new send request and immediately returns to the caller, and MPI_Recv similarly adds a new receive requests.

If the environment variable *INJ_IGNORE* evaluates to true, both functions will simply chain to the original MPI implementation without using the page tricks and immediately wait for it's completion. This enables applications to easily see the improvements added by our method.

4.7.3 MPI_Sendrecv

MPI_Sendrecv adds both a receive and a send request, making use of the added logic to disable overlapping tests between the two requests.

MPI_Sendrecv also honors the *INJ_IGNORE* flag.

4.7.4 MPI_Ssend, MPI_Rsend, MPI_Bsend and the MPI_Isend family

Calls to any of these functions are not optimized in any way and will go though completely untouched.

Our method is aimed at optimizing unoptimized applications, and applications using any of these functions already have some degree of optimization in them.

It is unnecessary to check the memory areas of the requests against the list of active requests and protected pages. As the MPI implementation runs in the context of the application, any page faults will be handled by our normal signal handler. This way, an application may use both simple MPI_Send, MPI_Recv and MPI_Sendrecv which will be optimized as well as it's own optimizations.

4.7.5 MPI_Get_count and MPI_Get_elements

These functions will check the mapping between status variables and requests. If such a mapping exists, they will wait for the request and chain to the original function. As waiting for a request will remove the mapping, this solution incurs practically no overhead.

Chapter 5

Tracking request chains

5.1 Overview

The pagetable manipulation tricks we propose are able to give us good improvements on some architectures, but on architectures where MPI operations are essentially memory operations it will cause a slowdown. The trick gives us something else though, it gives us a accurate way of determining when the results of an MPI request is actually needed. By remembering this information, it is possible to turn single MPI commands into chains of them and avoid the page protection overhead.

The first stage of such a technique is explaining what a chain is, and this is done in Section 5.2. Recognizing chains from program and data flow is explained in Section 5.3.

5.2 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.

5.3 Recognizing chains of requests

Recognizing a chain requires both a method for recognizing identical MPI calls and a way to recognize the sequence of MPI calls.

5.3.1 MPI call signatures

To recognize MPI calls, a signature of the call is made. This signature contains all the parameters passed to the MPI function, such as base address, datatype, count, communicator, tag and receiving/sending rank. Two signatures are identical only if all parameters are identical.

For each maintained signature, a small record is kept with the following data:

- *Seen count* - the number of times this exact signature has been used by the application.
- *Previous* and *Next* - what we believe to be the next and previous requests in the chain to be.
- *Confidence* of *Previous* and *Next* - how many times we've seen these requests in the same sequence.
- *PageBreak* pointer - the address that caused a pagebreak when this was the last active request.
- *WaitPoint* - if this request has been used as a end-of-chain for another request.

5.3.2 Confidence of chains

The confidence of chains is used to validate sequences that will always repeat. High confidence reduces the risk of data flow violation. It might still occur, but the higher the confidence the lower the chance.

Every time a request is made from the application, it's signature is looked up and the matching record found. Our method keeps track of the last signature seen, and checks if *LastSignature.Next* equals the new request. If it does, the *Confidence* of *LastSignature.Next* as well as *ThisSignaure.Prev* is increased. If it isn't, the pointers are updated and *Confidence* is set to 0.

5.3.3 Terminating chain

When a page fault occurs, the signal handler checks if there is a *LastSignature*. If there is, the *PageBreak* pointer is set in the matching record, indicating that no request in a chain should be allowed to proceed after this point in the application.

5.4 Using request chains

To make use of the request chains, the algorithm to add new requests is amended according to 7. It is only if we have a high confidence of chains all the way to a record with *PageBreak* set that we attempt using chains.

Algorithm 7: Testing for and adding a recognized request from part of a chain.

```
   input: R
1  New request begin
2      Compute signature for $R$ ;
3      Lookup record record for signature ;
4      Update record confidence. ;
5      recognized ← true ;
6      recptr ← record ;
7      while recognized and recptr $.Next$ and not recptr $.PageBreak$ do
8          if recptr $.Next$ confidence low then
9              recognized ← false ;
10     if recptr $neq$ signature and recognized then
11         recptr $.waitpoint$ ← true ;
12         Submit request asynchronously with no parameter change. ;
13     else
14         Submit request using normal paging method. ;
15     if recognized $.waitpoint$ then
16         Wait for all outstanding requests. ;
17 end
```

It is important to note that all communication in a recognized chained is done in-place, so there is no overhead of buffering and no overhead of paging.

5.4.1 Interoperability with paging

The most troublesome part of the chaining implementation is its interoperability with the paging. As chaining request are done in-place, they cannot overlap with the page-protected ones. Unfortunately, chains tend to grow gradually as larger confidence is built. At the start it might be just the terminating request and it's partner send or receive request, and once confidence builds up it will include communication with the other neighbors as well.

During this transition period, a recognized request in a chain might still be demoted to a page-protected request if it overlaps a request which need page protection. After all, the other request is already in progress and we can't change it. Similarly, if a paging request is starting that overlaps a recognized request, we have no choice but to wait for the recognized request to finish. Otherwise the MPI implementation would try to background write or read to data we just protected.

As seen in the results chapter, this transition period incurs a substantial overhead as we get the overhead of both methods and in addition might have to wait for more requests to finish.

5.5 Failure points of chaining

Chaining is not perfect. Even with a very high confidence in the chain it might break. Consider the small program given in 8. After a few iterations, the chain will be

$$SendA \rightarrow RecvB \rightarrow SendC \rightarrow RecvD \rightarrow Terminate$$

This comes from the fact that the only time the program accesses data is between the communication exchanges, and so that will become the end of chain. As the number of iterations increase, the confidence that $SendC$ always follows $RecvB$ becomes very high.

Unfortunately, this means the wrong value of B will be printed, as we never wait for that request to finish. There is no easy workaround for this that will enable us to maintain the same amount of improvement, but thankfully it only applies to

Algorithm 8: Example program that terminates in the middle of a recognized chain, violating data flow.

```
begin
    a ← 1 ;
    c ← 2 ;
    for each iteration do
        Send c to $Rank + 1$ ;
        Recv d from $Rank + 1$ ;
        c ← a ;
        a ← c + d + b ;
        Send a to $Rank - 1$ ;
        Recv b from $Rank - 1$ ;
    Print b;
end
```

programs with rather illogical communication patterns such as the example provided.

Chapter 6

Results and Discussion

This chapter will show and discuss the results of running our new method on the test program.

6.1 Test results

The same testing methodology was used as explained in 3.3.1. We ran tests with the confidence threshold set higher than the number of iterations to get results for just the paging method, and set it to just 2 to get results for the chaining. We also only measured the improvements on the "classic" part of the test program, the one using just MPI_Sendrecv.

It is now very hard to split timing into communication and computation parts, as a lot of the "waiting for communication" will happen in the pagefault handler, which is called during the test programs computation phase. While we note that in general the communication time has decreased drastically, the computation time has also increased noticeably. The results therefore focus on wallclock time, which after all is the final measurement of a program's execution time.

To validate that the program still computes the same results, we ran several test runs both with and without optimization and compared the results, and they were in all cases identical. Since the paging method does not allow for data flow violations, we have confidence the method should not alter execution results of any program.

Method	$n = 128$	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$	$n = 4096$
MPI_Sendrecv	0.12	0.22	0.59	1.92	8.00	106.68
MPI_Isend	0.11	0.21	0.55	1.87	7.80	106.18
Full overlap	0.11	0.20	0.54	1.80	7.36	105.63
Paging	0.44	0.64	0.96	2.58	8.60	102.45
Chaining	0.21	0.31	0.66	1.96	7.91	106.29

Table 6.1: Average iteration execution time in milliseconds of automatically optimized program compared to manually optimized on 16 nodes on the Snehvit(Altix) machine.

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 6.2: Average iteration execution time in milliseconds of automatically optimized program compared to manually optimized on 16 nodes on the Norgrid(Dell) cluster.

6.2 Snehvit results

The results on Snehvit are detailed in Table 6.1, and the speedup of the different methods are shown in Figure 6.1. Snehvit is a SMP machine with a highly optimized MPI engine, and already before we started the difference between a manually optimized and an unoptimized application was small.

6.2.1 Paging

On Snehvit, the overhead of paging far outweighs the benefits. Each time requests overlap, something they frequently do for the smaller data sizes, pages have to be unprotected and reprotected frequently.

As seen, paging consistently leads to a slowdown, with the notable exception that when $n = 4096$, the paging program has better results than any of the others. We ran the test numerous times and always got the same result. Part of the reason may be the fact that the paging version always optimizes memory to be page aligned, leading all requests to be aligned to the start of a cache line and fewer cache misses.

Additionally, with large data sets, computation is allowed to start at an earlier point. As soon as the top, left and right borders are done, all nodes except the

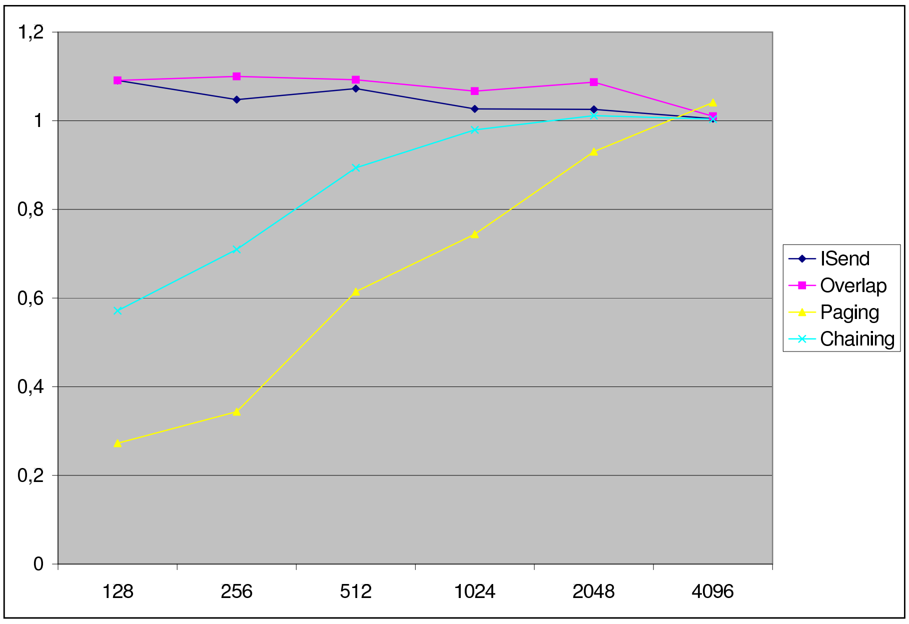

Figure 6.1: Speedup of automatic optimization and manual optimization on Snehvit.

ones at the bottom can be updated, meaning computation for interior nodes start while communication is still in progress, which leads to a very high degree of overlapping.

6.2.2 Chaining

Once chains are recognized and executed without page protection, there is a pretty consistent overhead of 10 milliseconds. This overhead comes mainly from looking up the request signature in the map and iterating over "active requests" to check for overlapping.

For the small request sizes, this overhead still leads to a slowdown, but once $n > 2048$, the overhead is small enough that there's a small gain. Since chaining aims to be equal to the MPI_Isend case, it will always wait at the end of the chain for all communication, so the benefit of overlapping interior computation and communication will not happen here. So for the $n = 4096$ case, paging is faster than chaining, a result which was totally unexpected when we started development.

6.3 Norgrid results

The results on Norgrid are detailed in Table 6.2, and the speedup of the different methods are shown in Figure 6.2. Norgrid is a regular Cluster of Dell rack-mounted Pentium4 PCs with Gigabit Ethernet. MPI operations are quite expensive, as they will ultimately be passed over Ethernet after being encapsulated in TCP/IP. As our method mainly deals with reducing the effect of latency, expectations for this architecture were high.

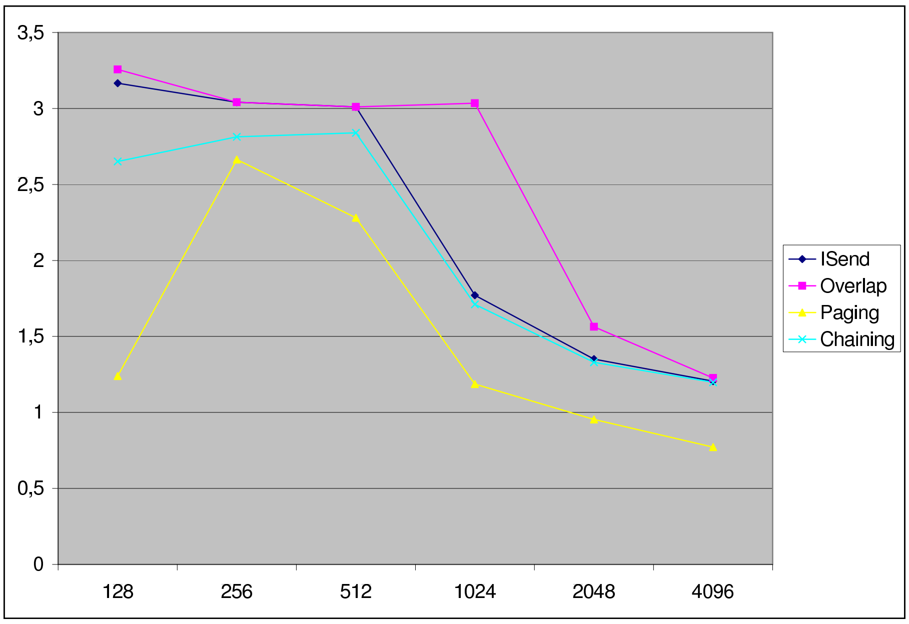

Figure 6.2: Speedup of automatic optimization and manual optimization on Norgrid.

6.3.1 Paging

On this architecture the overhead of a page fault is small, but it seems pages which get their access attributes changed are flushed from cache. For small data sizes, this still gives a decent speedup compared to regular MPI_Sendrecv, but as the data size grows, the overhead of reloading data from main memory outweighs the benefits, and there's a noticeable slowdown once $n > 2048$.

The sweet spot for paging seems to be in the $256 \leq n \leq 512$ range, where the running time was reduced by over 60%.

6.3.2 Chaining

As it was on Snehvit, chaining has pretty much a constant overhead which may be added to the MPI_Isend case. However, on this architecture, that overhead is smaller, so for any size of n, chaining leads to a substantial speedup over MPI_Sendrecv, getting close to the MPI_Isend case.

In short, this means that as long as the program doesn't have any communication patterns that tricks the chain confidence tests, there will always be a speedup when applying our new method on this architecture.

6.4 Projections for other architectures

Our new method deals primarily with reducing latency. On any kind of cluster or SMP machine where latency is a dominating factor, the method may provide substantial improvements.

On machines with dedicated low-latency MPI hardware where writing to a remote machine is almost as fast as writing to local memory, the method currently does not offer any improvements. However, we have a few ideas on how to achieve speedup even on these architectures, and this is discussed in Chapter 7.

6.5 Discussion

Our new method deals primarily with automatically optimizing MPI programs for which no previous optimization has been done. The first point of discussion becomes whether or not it's worth putting effort into optimizing for these kinds of applications?

6.5.1 Why optimize naive code?

While we have no studies or data to prove it, our own observations of other MPI programs running on the machines we have access to lead to the conclusion that optimizations are frequently not done at all. Indeed, previous work at NTNU with the Unified Model [18] shows that even production code can have horrible code quality, and that the major point of interest for the users of the code is just the results.

At the same time, there is a growing trend for Desktop applications that optimizations aren't necessary. By the time the application is developed, faster processors and more memory will be available for the end user to buy. At the same time, the fact that program libraries like the Java Foundation Classes, the Standard Template Library (C++) and the .NET Framework provide optimized sort functions is the only reason bubblesort is no longer the most commonly implemented sorting algorithm.

As computer scientists, we can either optimize our own code, going from 99% efficiency to 99.5%, or we can take a stab at optimizing all the other horrible code out there, going from 50% efficiency to 80%. While the glamor and fame of the former might be higher, the real world savings in computer time is much higher for the latter.

6.5.2 Portability

The current version of the code is made with portability in mind. Indeed, we use the same code on both Norgrid and Snehvit, which are completely different machines. The code also compiles and runs on Gridur, an Origin 3800, but the instability in timing on this machine precludes measuring improvements with any confidence.

It is possible to achieve better results by using machine specific functions and hardware features. For example, having per-thread page protection attributes would avoid the overhead of copying to a temporary buffer for the paging method, and SGIs implementation of memory allocation pools would allow us to avoid the page-alignment of malloc.

Likewise, it's possible to use a specific feature of certain compilers. For example, it's easy to use the inter-procedural optimizations of the Intel C Compiler to inline our code at compile time in the target application.

However, any such optimization would limit the programs usability. We would lock it to a specific set of platforms, and hence limit the users choices. As it is our goal to make this method available and useful to everyone, we are willing to sacrifice a small bit of theoretical peak performance in favor of having it available on more platforms.

6.5.3 3D PDE Solvers

Our current tests are done with a 2D PDE Solver. While it has not been tested, the performance increase for a 3D PDE Solver should be just as good.

In the 3D PDE solver, surfaces are exchanged instead of borders, so the amount of data to transfer will be larger. However, it will also be necessary to transfer to 6 neighbors instead of 4, and with chaining they will all be done at once meaning the latency overhead is even further reduced than in the 2D case.

Chapter 7

Conclusion and Future work

7.1 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.

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.

Our implementation demonstrates unconventional use of hardware resources to aid optimization, and also demonstrates dynamic changing of a precompiled program we have no source for, something that's previously been restricted to interpreted or Just-in-Time compiled languages.

With chaining enabled, our method has very little overhead, and will improve the running time of most applications running on regular clusters.

7.2 Future work

Our new method is mostly a proof-of-concept at this point, enough to show that the idea works in real life on a given test case. There are still a lot of things that can be done that may or may not improve the runtime even further, but unfortunately we didn't have time to implement them all. We will list some of these ideas here.

7.2.1 Caller address in signature

It would be beneficial to add the call-return address to the signature of the request, to make sure the request originated from the exact same point in the source code as well. Currently, if two completely different places in the source code transfer the exact same data, the signatures would be identical. Most likely the communication pattern will vary slightly between the two places, and this will preclude chains from building up enough confidence.

Getting the call-return address means peeking at the stack, which is very architecture specific, and can change depending on what compiler options are used (-fomit-frame-pointer being a good example). The best solution is probably to make a small macro to fetch the return address on architectures that support it and just return 0 on the others, so unsupported architectures just fall back to the same signature we use today.

7.2.2 Overlapping tests

At the moment, we use a very simple test for overlapping regions of data; if the upper and lower bounds intersect in any way, it's an overlap. However, as was explained in Section 2.1.2.1, this gives us a lot of false positives. Unfortunately, proper analysis of overlapping is computationally expensive and is likely to generate more overhead than we have to gain.

An idea is to use the fast method during the paging phase, but if enough confidence is achieved that we move into chaining, we can do the expensive analysis once. As long as the signatures stay identical, two requests that didn't overlap the first time will not overlap later on. By delaying until we reach the chaining phase, we also ensure that the added overhead will be spread over many following iterations.

7.2.3 Modeling of transfer time

As was seen in Chapter 6, the potential improvement in runtime varies with the size of messages. If better modeling was added to the method, one could predict when no speedup was possible, and then fall back to just using the original function. Unfortunately, the classic model of $T_C = T_S + \beta n$ is not accurate enough to cover our needs here, as it does not fit well for very small data packets, nor does it model multiple concurrent communications.

If a better model was available that was very fast to compute and update, it could be updated on the fly based on the time transfers actually took, and then continually be used to make proper decisions on what to do.

7.2.4 Per-thread page protection

While it will be limited to just one OS (Linux kernel based ones), a kernel module that enables per-thread page manipulations would enable us to find the "best possible" time, and more accurately measure how much we actually lose by our current copying scheme.

7.2.5 Partial communication

Based on the results for the $n = 4096$ case on Snehvit, we see a potential for allowing partial unlocking of pages. For the left and right sides, unlocking half the pages once half the transfer is complete would allow half the computation to finish while we finish the last half of communication. While such a method would add overhead, it is better than simply waiting for communication to finish.

Unfortunately, there is no good way to measure the progress of a MPI transfer, as it's either done or not done. So such a method would require some modification of the underlying MPI library.

7.2.6 Communication priority

Once n is high, communication is not bound by latency, but by bandwidth. At that point the current method fails to offer much of an improvement.

By delaying the parts of communication that won't be needed until the end of the computation phase until the end of the communication phase, some speedup will be possible here too. For our test program, this would work roughly as follows:

- The application submits exchanges of top, left, right and bottom borders.
- The improved method protects all the pages, but only starts transferring the top and left borders before returning.
- The application will page fault at the very first element, belonging to the left and top borders.
- The improved method waits for top and left to finish, then starts the transfer for right border. Pages for top and left are unprotected.
- The application computes the top row of elements, page faulting at the last one.
- The improved method waits for right border to finish, then starts the transfer for bottom border. Pages for right are unprotected.
- The application finishes computation for all rows but the last one, which will be done by the time it gets there.

In theory, this should provide a 25% decrease in communication time even on machines with very good interconnect. It is absolutely essential that the communication model be accurate though, as otherwise this method will just add overhead.

7.2.7 Ready sends

On SMP machines, it is customary to use either an "eager" or a "rendezvous" communication mode. In "eager" mode, the data is copied to a buffer by the sending processor and copied out of the buffer again by the receiver. This is used for very small messages. In "rendezvous" mode, the sender waits for the receiver to post a receive, then copies the data directly to the target memory.

Normally, eager is used for short messages and rendezvous for larger ones. eager is "faster" as it doesn't require the two processes to be synchronized, but rendezvous avoids the extra buffer copy. The MPI implementation switches between these based on a heuristic of the transfer size.

However, sometimes it makes sense to use a different method than what the MPI implementation heuristic wants. For example, if we know the receive is ready at the other end, we would improve runtime by forcing rendezvous. And if we know it will not be ready for some time, we should force eager.

Knowing whether or not a receive is ready ahead of time is very hard. However, by expanding our request record associated with a signature a bit, we can try forcing a different method each time we see the request and measure the time taken. One of the two methods will give better results, and as our heuristic is based on program-specific request history it should give a better heuristic than the built-in one.

7.2.8 Safe breaking of chains

If a program breaks a recognized chain as demonstrated in Section 5.5, there is currently nothing we can do.

The problem can be worked around by setting program execution mark points at each start of chain, and using copy-on-write semantics to preserve the execution image at that point, but this would add far more overhead than we could save.

What we can do, however, is warn the user that a chain broke. Each time a MPI function is called, we check if it came unexpectedly. If it does, program execution flow changed and we broke the chain. At this point, we would inform the user of the problem.

7.2.9 Application database

The ideas of *Better modeling*, *Ready sends* and *Safe breaking of chains* would benefit immensely from having their data stored between runs. It should be fairly trivial to store the database once MPI_Finalize is called and reload it in MPI_Init. While this won't help much during development, where communication and execution patterns change all the time, it will help a lot once the application reaches the "verification" stage in which numerous runs with different data sets are used, and it will also help for unoptimized applications that are in production.

7.2.10 Instrumentation

While our method is primarily targeted at unoptimized programs for normal users, it's idea of chains and confidence, it's analysis of *Ready Sends* and *Communication priority* would be a very good starting point for manual optimization.

If the *Application database* is implemented along with *Caller address*, it would be possible to read the debug data of the application and print out the source code line for each MPI function call and a recommendation on what to do.

Bibliography

[1] M. P. I. Forum, Technical Report No. UT-CS-94-230 (unpublished). 5

[2] P. S. Pacheco, *Parallel programming with MPI* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996). 5

[3] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, Parallel Computing **22**, 789 (1996). 10

[4] W. Gropp and E. Lusk, Parallel Computing **22**, 1513 (1997). 10

[5] W. Gropp and E. Lusk, The International Journal of Supercomputer Applications and High Performance Computing **11**, 103 (1997). 10

[6] G. Burns, R. Daoud, and J. Vaigl, in *Proceedings of Supercomputing Symposium* (PUBLISHER, ADDRESS, 1994), pp. 379–386. 10

[7] J. M. Squyres and A. Lumsdaine, in *Proceedings, 10th European PVM/MPI Users' Group Meeting*, No. 2840 in *Lecture Notes in Computer Science* (Springer-Verlag, Venice, Italy, 2003), pp. 379–387. 10

[8] O. Zaki, E. Lusk, W. Gropp, and D. Swider, The International Journal of High Performance Computing Applications **13**, 277 (1999). 10

[9] S. Shende *et al.*, Portable Profiling and Tracing for Parallel Scientific Applications using C, 1998. 10

[10] D.U.M.A. - Detect Unintended Memory Access. 11

[11] N. Nethercote and J. Seward, Valgrind: A Program Supervision Framework, 2003. 11

[12] R. C. Whaley, A. Petitet, and J. J. Dongarra, Parallel Computing **27**, 3 (2001), also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`). 11

[13] 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. 11

[14] Åsmund Østvold, Timing and Measurement Techniques for MPI Collective Communication Operations, 2003. 11

[15] 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. 11

[16] C. Dulong *et al.*, Intel Technology Journal 15 (1999). 11

[17] L. Hageman and D. Young, *Applied Iterative Methods* (Academic Press, New York, 1981). 14

[18] L. E. Eilertsen and J. A. Amundsen, Unified Model Optimization, Personal Communication, 2005. 43

Appendix A

Installation and User's Guide

This chapter contains a brief introduction on how to compile and install our new method as well as a short end user guide on how to use it.

A.1 Compilation

The only file that needs to be compiled is *layer.cpp*, but it has to be compiled both as a static library and as a dynamic injectable one.

Note that the method of dynamic injection doesn't apply unless you also have a dynamically loaded MPI library. The easiest way to test this, is to compile a simple MPI program, and run

```
ldd programname
```

If the output contains a line with something like *libmpi.so* then you have a dynamically loaded MPI library.

A.1.1 Static library

Static compilation should be just like compiling any other object file on your architecture.

```
g++ -O3 -DINJECT_STATIC -o layer.o layer.cpp
```

This syntax works with the Intel C Compiler and MIPSPro as well. The *-DINJECT_STATIC* is just a flag to include the initialization options necessary for the static library.

Next, the object file needs to be linked into a static library. On most UNIX architectures, this can be achieved as follows:

```
ar r libinjlib.a layer.o
ranlib libinjlib.a
```

ar archives the object files (in this case just one) into an archive, and ranlib generates an index of symbol names in the archive to make it suitable for linking.

A.1.2 Dynamic library

Compiling a dynamic library is often slightly more complicated, and even more so in this case as the library has to be fully dynamically relocatable. To compile the object file with GCC or Intel C, use the following:

```
g++ -O3 -DINJECT_DYNAMIC -rdynamic -fPIC -o layer.ro layer.cpp
```

-DINJECT_DYNAMIC is just a flag for the source to include the parts necessary for dynamic injection. *-fPIC* produces Position Independent Code in which most variable references are made relative to the program counter, and constant addresses use the GOT (Global Offset Table). To be able to be injected into a precompiled process, we need to ensure a GOT exists, and *-rdynamic* forces the compiler to generate one.

To link the object into a library, the following is used:

```
g++ -shared -Wl,-soname,injlib.so.1 -o injlib.so.1.0.1 layer.ro -l
```

-shared specifies a shared library, the *-soname* specifies the internal linker name (so we can make inlib.so.1 a symlink to injlib.so.1.0.1 and later just update the symlink). *-lc -ldl -lmpi -lmpi++* include the necessary libraries for linking.

A.2 Using the library

Depending on whether you use static or dynamic injection, you'll have to modify either the compilation or running phase of your program. *Do not do both*, as that would make our new method try to optimize our new method, and the added overhead will decrease performance somewhat.

A.2.1 Compiling with static library

To compile with the static library, include *mpii.h* instead of *mpi.h* in your program source. Modify your final linking stage to include *-linjlib* before *-lmpi*, or if you use a *mpicc* wrapper, just include *-linjlib* in options to the final linker.

For example, if your original program was linked by

```
gcc -o hello hello.o -lmpi
```

then you should instead use

```
gcc -o hello hello.o -linjlib -lmpi
```

If you didn't put libinjlib.a in the library path, add a options for *-L/path/to/injlib/directory*.

A.2.2 Injection dynamic library

To run the program with dynamic injection, set the environment variable *LD_PRELOAD* to */full/path/to/injlib.so.1.0.1*. It is safest not to set this variable globally, as that would make injlib be inserted into every subsequently run program. The safest is to use it on the line of *mpirun* as follows:

```
LD_PRELOAD=./injlib.so.1.0.1 mpirun -np 4 hello
```

A.2.3 Controlling library use

If the library has been successfully inserted, it will print out a line saying something like

```
MPII Init (debug -1, ignore 0)
```

from each process.

To disable the library, set the environment variable *INJ_IGNORE* to 1. This will make the overridden functions wait for results to complete, and will disable both the paging and chaining mechanisms. This option is very useful if you suspect a fault in the library.

To enable extended debugging, set the environment variable *INJ_DEBUG*. A value of -1 (the default) disables debugging. -2 will enable debugging output from all processes, and any other value will enable debugging only from the process whose rank matches *INJ_DEBUG*.

A.3 Global installation

A global installation enables our new method transparently for all users, using the static injection method on all compiled MPI programs.

Most cluster architectures provide a *mpicc* wrapper to compile MPI programs. You will need to modify this wrapper to add *-linjlib* everywhere it adds *-lmpi*. In the MPICH distribution, this is accomplished by adding

```
mpilibs="-linjlib $mpilibs"
```

just before the linker is run.

Modify *mpii.h* to include *mpi_orig.h* instead of *mpi.h*. Rename the original *mpi.h* to *mpi_orig.h* and then rename *mpii.h* to *mpi.h* and put it in the same directory as *mpi_orig.h*. This will make sure the symbols are right for all loaded programs.

Finally, we recommend modifying *mpirun* to set *INJ_IGNORE* to 1 unless the environment variable already exists. Most users react negatively to forced changes, and this change will change the timing of any benchmarks or comparisons. However, once all the above is done, users just have to

```
export INJ_IGNORE=0
```

to enable our new method in their programs.

Appendix B

MPII Class Documentation

B.1 MemReq Struct Reference

A memory area in use by an active MPI request.

Public Member Functions

- MemReq (bool is_send, unsigned char ∗base, int count, MPI_Datatype datatype, MPI_Comm comm, MPI_Status ∗status)

  Construct a new active request.

- ∼MemReq ()

  Destructor.

Public Attributes

- bool is_send

  Indicates if is a MPI_∗send or MPI_∗recv request.

- bool is_protected

  Is the memory area mprotect()ed?

- bool is_waitall

Is this request the end of a chain?

- Range reqmem

  Memory range (in user program) of request.

- Range pages

  Memory range of request, page-aligned for.

- void ∗ base

  Base address used in request, identical to reqmem.start.

- int count

  Number of elements in request.

- MPI_Datatype datatype

  Datatype used in request.

- MPI_Request req

  Request handle.

- MPI_Comm comm

  Communicator used in request.

- MPI_Status ∗ status

  Status (if any).

- int bufsize

  Size of buffer for MPI_Pack and MPI_Unpack.

- unsigned char ∗ buffer

  Allocated buffer for packing/unpacking data.

B.1.1 Detailed Description

A memory area in use by an active MPI request.

This struct holds all the information about an active request.

Definition at line 187 of file layer.cpp.

B.1.2 Constructor & Destructor Documentation

B.1.2.1 MemReq::MemReq (bool *is_send*, unsigned char ∗ *base*, int *count*, MPI_Datatype *datatype*, MPI_Comm *comm*, MPI_Status ∗ *status*) `[inline]`

Construct a new active request.

Parameters:

is_send Is this request a MPI_∗send?

base Buffer to send/recv to.

count Number of elements.

datatype Datatype.

comm Communicator to use.

status Pointer to status variable. This will initialize reqmem and pages based on the datatype and count, and will allocate a buffer for packing and unpacking.

Definition at line 236 of file layer.cpp.

References buffer, bufsize, is_protected, is_waitall, Range::pageAdjust(), pages, reqmem, Range::start, and Range::stop.

B.1.2.2 MemReq::∼MemReq () `[inline]`

Destructor.

Deallocates memory for buffer.

Definition at line 267 of file layer.cpp.

References buffer.

B.1.3 Member Data Documentation

B.1.3.1 bool MemReq::is_waitall

Is this request the end of a chain?

If this is true, this is the end of a chain and we should wait for all outstanding requests to finish before moving along.

Definition at line 199 of file layer.cpp.

Referenced by add_req(), and MemReq().

The documentation for this struct was generated from the following file:

- layer.cpp

B.2 Range Struct Reference

A range of memory.

Public Member Functions

- Range ()

  Null constructor.

- Range (void ∗start, void ∗stop)

  Basic contructor.

- bool operator== (const Range &o) const

  Compare two ranges.

- bool overlaps (const Range &o) const

  Check if two ranges overlap (intersect).

- bool overlaps (void ∗ptr) const

  Check if address is inside range.

- Range pageAdjust () const

  Adjust range to page boundries.

- size_t size () const

  Compute size of range.

Public Attributes

- void ∗ start

  Start of range.

- void ∗ stop

  End of range.

B.2.1 Detailed Description

A range of memory.

This is a convenience class to work with an manipulate a range of memory (such as the buf variable of most MPI calls).

As many compilers don't like pointer arithmetics with void $*$ pointers, this makes the rest of the codemuch cleaner.

Definition at line 117 of file layer.cpp.

B.2.2 Constructor & Destructor Documentation

B.2.2.1 Range::Range (void $*$ *start*, void $*$ *stop*) `[inline]`

Basic contructor.

Parameters:

start Start of range.

stop End of range.

Definition at line 133 of file layer.cpp.

B.2.3 Member Function Documentation

B.2.3.1 bool Range::operator== (const Range & *o*) const `[inline]`

Compare two ranges.

Parameters:

o Other range to compare with.

Returns:

true if ranges are identical.

Definition at line 142 of file layer.cpp.

References start, and stop.

B.2.3.2 bool Range::overlaps (void ∗ *ptr*) const `[inline]`

Check if address is inside range.

Parameters:

ptr Address to check.

Returns:

true if address is in range.

Definition at line 159 of file layer.cpp.

References start, and stop.

B.2.3.3 bool Range::overlaps (const Range & *o*) const `[inline]`

Check if two ranges overlap (intersect).

Parameters:

o Other range to compare with.

Returns:

true if ranges overlap.

Definition at line 150 of file layer.cpp.

References start, and stop.

B.2.3.4 Range Range::pageAdjust () const `[inline]`

Adjust range to page boundries.

Returns:

New adjusted range.

start is adjusted downwards to the start of page boundry, and stop is likewise adjusted upwards.

Definition at line 169 of file layer.cpp.

References PAGE_MASK, PAGE_SIZE, start, and stop.

Referenced by MemReq::MemReq().

B.2.3.5 size_t Range::size () const `[inline]`

Compute size of range.

Returns:

Size of range in bytes.

Definition at line 177 of file layer.cpp.

References start, and stop.

Referenced by add_req(), and wait_req().

The documentation for this struct was generated from the following file:

- layer.cpp

B.3 ReqInfo Struct Reference

Information about a request we've seen.

Public Attributes

- int seen

  How many times this exact request has been seen.

- ReqInfo $*$ prev

  Next and previous request in chain.

- ReqInfo $*$ next

  Next and previous request in chain.

- int prevcount

  How many times have we seen the same next and previous requests.

- int nextcount

  How many times have we seen the same next and previous requests.

- void $*$ faultat

  Address of page fault while this request was last active request.

- bool waitall

  If true, some earlier request specified this as chain end.

B.3.1 Detailed Description

Information about a request we've seen.

The library tracks and records the most recent requests done by the user program. This is used to build information about request chains; requests that always follow each other without any computation inbetween them. Such chains can then be started with their original buffer area, removing the overhead of copying, and we simply wait at the end of the chain for all requests to finish.

Chains are recognized in add_req(), and are simply a series of requests where prevcount and nextcount are fairly high, and the chain terminates in the request which has faultat set.

Definition at line 365 of file layer.cpp.

The documentation for this struct was generated from the following file:

- layer.cpp

B.4 ReqSig Struct Reference

Signature of MPI Request.

Public Member Functions

- ReqSig (bool is_send, void $*$base, int count, MPI_Datatype datatype, int partner, int tag, MPI_Comm comm)

  Construct a new signature.

- bool operator== (const ReqSig &o) const

  Compare two signatures.

- bool operator< (const ReqSig &o) const

  Compare two signatures.

Public Attributes

- bool is_send

  Is this is a send request?

- void $*$ base

  Base memory address of request.

- int count

  Number of elements.

- MPI_Datatype datatype

  Datatype.

- int partner

  Source or destination rank.

- int tag

  Message tag.

- MPI_Comm comm

  Communicator handle.

B.4.1 Detailed Description

Signature of MPI Request.

This is the signature of an MPI request. If two signatures are identical, they should transfer the same range of memory in the same way to the same destination. This is used to recognize requests we've seen before on subsequent iterations in the user program.

The member variables here will be the parameters seen to the MPI_Send or MPI_Recv calls.

This struct has no virtual functions, no parent and should not have any children as it relies on being "just a chunk of memory" for its assignment and comparison operators.

Definition at line 287 of file layer.cpp.

B.4.2 Constructor & Destructor Documentation

B.4.2.1 ReqSig::ReqSig (bool *is_send*, void $*$ *base*, int *count*, MPI_Datatype *datatype*, int *partner*, int *tag*, MPI_Comm *comm*) `[inline]`

Construct a new signature.

Parameters:

is_send Is this a send requst?

base The initial address of the send/receive buffer.

count Number of elements in buffer.

datatype Datatype of each element.

partner Rank of source or destination.

tag Message tag.

comm Communicator handle.

This constructor just does assignments, so it should be reasonably fast. C++ will provide copy and assignment constructors for us (which will basically be memcpy() versions).

Definition at line 317 of file layer.cpp.

B.4.3 Member Function Documentation

B.4.3.1 bool ReqSig::operator< (const ReqSig & *o*) const `[inline]`

Compare two signatures.

Parameters:
o Other signature to compare with.

Returns:
true if this signature is "less than" o.

See also:
operator==()

This is primarily to allow use as key in std::map<>.

Definition at line 344 of file layer.cpp.

B.4.3.2 bool ReqSig::operator== (const ReqSig & *o*) const `[inline]`

Compare two signatures.

Parameters:
o Other signature to compare with.

Returns:
true if signatures are identical.

See also:
operator<()

Definition at line 333 of file layer.cpp.

The documentation for this struct was generated from the following file:

- layer.cpp

Appendix C

MPII File Documentation

C.1 layer.cpp File Reference

Main library for the pagefault assisted automatic tuning.

```
#include <mpi.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <dlfcn.h>
#include <limits.h>
#include <signal.h>
#include <sys/mman.h>
#include <errno.h>
#include <set>
#include <map>
#include "layer.h"
```

Namespaces

- namespace **std**

Classes

- struct Range

  A range of memory.

- struct MemReq

  A memory area in use by an active MPI request.

- struct ReqSig

  Signature of MPI Request.

- struct ReqInfo

  Information about a request we've seen.

Defines

- #define PAGE_SIZE 16384

  Size of a hardware page, in bytes.

- #define PAGE_MASK 0xffffffffffffc000

  And-mask for aligning an address to the start of page.

- #define CHAIN_MIN_ITER 3

  Minimum number of stable iterations before recognizing chain.

- #define INJ_METHOD extern "C"

  Calling convention and name mangling of exported symbols.

- #define MEM_VALID_TEST(x) (mallocpages.find((unsigned long int)x / PAGE_SIZE) != mallocpages.end())

  Check if x is inside our overriden malloc()ed pages.

Functions

- void dbgout (const char ∗format,...)

  Output debug string.

- void wait_req (MemReq ∗mr)

  Wait for request and release memory.

- void wait_all_req ()

  Wait for all outstanding requests.

- MemReq ∗ add_req (bool is_send, void ∗base, int count, MPI_Datatype datatype, int rank, int tag, MPI_Comm comm, MPI_Status ∗status, MemReq ∗ignore=NULL)

  Start new request.

- void sigact (int signal, siginfo_t ∗siginfo, void ∗ucontext)

  Segment violation signal handler.

- void Inj_init_chain ()

  Initialize injection.

- INJ_METHOD int Inj_MPI_Init (int ∗argc, char ∗∗∗argv)

  Initialize MPI and injection.

- INJ_METHOD int **Inj_MPI_Send** (void ∗buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- INJ_METHOD int **Inj_MPI_Recv** (void ∗buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status ∗status)
- INJ_METHOD int Inj_MPI_Sendrecv (void ∗sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void ∗recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status ∗status)

  Overridden MPI_Sendrecv().

- INJ_METHOD int Inj_MPI_Get_count (MPI_Status ∗status, MPI_Datatype datatype, int ∗count)

  Overridden MPI_Get_count().

- INJ_METHOD void ∗ Inj_malloc (size_t size)

  Overridden malloc().

- INJ_METHOD void Inj_free (void ∗ptr)

  Overridden free().

Variables

- int do_debug

  Holds the rank of the process that should output debug information.

- int do_nothing

  If nonzero, the user has requested no "messing" should take place.

- int rank

  Rank of current process.

- int(∗ Orig_MPI_Init)(int ∗, char ∗∗∗)

  Pointer to original MPI_Init() function.

- int(∗ Orig_MPI_Isend)(void ∗, int, MPI_Datatype, int, int, MPI_Comm, MPI_Request ∗)

  Pointer to original MPI_Isend() function.

- int(∗ Orig_MPI_Irecv)(void ∗, int, MPI_Datatype, int, int, MPI_Comm, MPI_Request ∗)

  Pointer to original MPI_Irecv() function.

- int(∗ Orig_MPI_Wait)(MPI_Request ∗, MPI_Status ∗)

  Pointer to original MPI_Wait() function.

- int(∗ Orig_MPI_Get_count)(MPI_Status ∗, MPI_Datatype, int ∗)

  Pointer to original MPI_Get_count() function.

- void ∗(∗ Orig_malloc)(size_t)

Pointer to original malloc() function.

- void(∗ Orig_free)(void ∗)

  Pointer to original free() function.

- sigaction sa

  Information about new and old signal handler.

- sigaction **oldsa**
- set< MemReq ∗ > requests

  Set of active requests.

- map< MPI_Status ∗, MemReq ∗ > statuses

  Map of statuses we might wait for.

- map< void ∗, void ∗ > mallocmap

  Map between returned page-aligned address and "true" address.

- map< void ∗, size_t > mallocsizes

  Map of malloc sizes (number of bytes).

- set< unsigned long int > mallocpages

  Set of pages we've pre-aligned (page number, not address).

- map< ReqSig, ReqInfo ∗ > reqhist

  Map of signatures and matching information.

- ReqInfo ∗ last_req = NULL

  Last active request.

C.1.1 Detailed Description

Main library for the pagefault assisted automatic tuning.

This is the main portion of the library which handles the automatic tuning of MPI code. The library can be injected either dynamically (via LD_PRELOAD) or statically (by including mpii.h and linking with the generated injlib.a.

Note that most variables, functions and classes in this file are limited to the file-scope (static), to avoid polluting the function namespace of the process we're injecting into, and also allow the compiler as much flexibility with inlining as it cares to do.

Todo

Use MPI_Irsend for MPI_Rsend. Or maybe not, as programs which use MPI_Rsend are kinda optimized already.
If seen>10, make it a persistant request.
Runtime option to mprotect() entire area of a chain.
LD_PRELOAD oveeride for operator new.

Definition in file layer.cpp.

C.1.2 Function Documentation

C.1.2.1 MemReq∗ add_req (bool *is_send*, void ∗ *base*, int *count*, MPI_Datatype *datatype*, int *rank*, int *tag*, MPI_Comm *comm*, MPI_Status ∗ *status*, MemReq ∗ *ignore* = NULL) `[static]`

Start new request.

Parameters:

is_send Is this a send request?

base Base address of request.

count Number of elements.

datatype Datatype of elements.

rank Source or destination rank.

tag Message tag.

comm Communicator handle.

status Pointer to status receptor.

ignore Address of MemReq for which we ignore overlapping. (SendRecv pairs)

Returns:

Address of MemReq

This adds a new tracked request. First, it updates chain information, and if there is sufficient confidence this is part of a chain, skip memory protection alltogether. If this is the end of a chain, wait for all outstanding requests to finish before returning.

For new requests, or requests not part of a chain, make sure any requests which use the same memory addresses are already done (unless they were a send, and this is also a send).

Definition at line 535 of file layer.cpp.

References MemReq::buffer, MemReq::bufsize, CHAIN_MIN_ITER, dbgout(), ReqInfo::faultat, MemReq::is_protected, MemReq::is_waitall, last_req, ReqInfo::next, ReqInfo::nextcount, Orig_MPI_Irecv, Orig_MPI_Isend, MemReq::pages, ReqInfo::prev, ReqInfo::prevcount, rank, MemReq::req, reqhist, requests, ReqInfo::seen, Range::size(), Range::start, statuses, wait_all_req(), wait_req(), and ReqInfo::waitall.

Referenced by Inj_MPI_Sendrecv().

C.1.2.2 void dbgout (const char $*$ *format*, ...) `[static]`

Output debug string.

Parameters:

format format of string (printf format).

This outputs a debug string to stderr if the environment variable INJ_DEBUG equals the rank of the calling process.

Definition at line 411 of file layer.cpp.

References do_debug, and rank.

Referenced by add_req(), Inj_MPI_Get_count(), Inj_MPI_Init(), Inj_MPI_Sendrecv(), sigact(), wait_all_req(), and wait_req().

C.1.2.3 INJ_METHOD void Inj_free (void $*$ *ptr*)

Overridden free().

This is an overridden free() which replaces the pointer with the "orignal" if this memory slab was allocated through our overridden Inj_malloc().

Definition at line 937 of file layer.cpp.

References INJ_METHOD, mallocmap, mallocpages, mallocsizes, Orig_free, Orig_MPI_Init, and PAGE_SIZE.

C.1.2.4 void Inj_init_chain () `[static]`

Initialize injection.

The guts of this function will differ betweenthe static and dynamic injection types, but their function is to initialize Orig_MPI_Init and related functions.

Definition at line 764 of file layer.cpp.

References Orig_free, Orig_malloc, Orig_MPI_Get_count, Orig_MPI_Init, Orig_MPI_Irecv, Orig_MPI_Isend, and Orig_MPI_Wait.

Referenced by Inj_MPI_Init().

C.1.2.5 INJ_METHOD void∗ Inj_malloc (size_t *size*)

Overridden malloc().

This is an overridden malloc() which ensures the allocated area is to page boundries. Without this, small dynamic allocations (such as those done by the MPI library itself) could end up in the same page as the original buffer, meaning they too would be inaccessible once we mprotect().

However, if MPI_Init hasn't been called yet, this just chains through immediately. If called via LD_PRELOAD, we would also be injected into mpirun, and it doesn't make sense to pagealign that.

Definition at line 913 of file layer.cpp.

References INJ_METHOD, mallocmap, mallocpages, mallocsizes, Orig_malloc, Orig_MPI_Init, PAGE_MASK, and PAGE_SIZE.

C.1.2.6 INJ_METHOD int Inj_MPI_Get_count (MPI_Status ∗ *status*, MPI_Datatype *datatype*, int ∗ *count*)

Overridden MPI_Get_count().

This waits for the request to actually finish, then chains through.

Definition at line 889 of file layer.cpp.

References dbgout(), INJ_METHOD, Orig_MPI_Get_count, statuses, and wait_req().

C.1.2.7 INJ_METHOD int Inj_MPI_Init (int ∗ *argc*, char ∗∗∗ *argv*)

Initialize MPI and injection.

This is the overridden versionof MPI_Init(), and will initialize the signal handler and status variables in addition to calling the original MPI_Init().

Definition at line 784 of file layer.cpp.

References dbgout(), do_debug, do_nothing, Inj_init_chain(), INJ_METHOD, Orig_MPI_Init, rank, sa, and sigact().

C.1.2.8 INJ_METHOD int Inj_MPI_Sendrecv (void ∗ *sendbuf*, int *sendcount*, MPI_Datatype *sendtype*, int *dest*, int *sendtag*, void ∗ *recvbuf*, int *recvcount*, MPI_Datatype *recvtype*, int *source*, int *recvtag*, MPI_Comm *comm*, MPI_Status ∗ *status*)

Overridden MPI_Sendrecv().

If do_nothing is true, this just transforms the request into a Irecv and Isend, but waits on them immediately before returning.

During normal operation, add_req() is called for both the send and receive parts of the request before we return.

Definition at line 862 of file layer.cpp.

References add_req(), dbgout(), do_nothing, INJ_METHOD, MEM_VALID_TEST, Orig_MPI_Irecv, and Orig_MPI_Isend.

C.1.2.9 void sigact (int *signal*, siginfo_t ∗ *siginfo*, void ∗ *ucontext*) `[static]`

Segment violation signal handler.

Parameters:

signal Signal number.

siginfo Information about signal.

ucontext User context.

This handles page faults, which most likely are caused by the user program accessing protected pages.

We simply iterate the active requests, wait for all requests that overlap the faulted address to finish. wait_req() will unprotect the pages again, so we simply return to userspace when done.

Definition at line 689 of file layer.cpp.

References dbgout(), do_debug, ReqInfo::faultat, last_req, rank, requests, sa, wait_all_req(), and wait_req().

Referenced by Inj_MPI_Init().

C.1.2.10 void wait_all_req ()

Wait for all outstanding requests.

This waits for all outstanding requests, used at the end of chains and in the case of "true" segment faults.

Definition at line 504 of file layer.cpp.

References dbgout(), requests, and wait_req().

Referenced by add_req(), and sigact().

C.1.2.11 void wait_req (MemReq $*$ *mr*) `[static]`

Wait for request and release memory.

Parameters:

mr Request to wait for.

This will wait for an MPI request to finish, then wait for any requests using the same pages it has locked, and finally unprotect the pages so they can be accessed by the application again. If the request was a receive, the data will be copied from the buffer to the user buffer.

If the request was part of chain and not protected, just wait for it finish.

Definition at line 441 of file layer.cpp.

References MemReq::base, MemReq::buffer, MemReq::comm, MemReq::count, MemReq::datatype, dbgout(), MemReq::is_protected, MemReq::is_send, Orig_MPI_Get_count, MemReq::pages, MemReq::req, requests, Range::size(), Range::start, MemReq::status, and statuses.

Referenced by add_req(), Inj_MPI_Get_count(), sigact(), and wait_all_req().

C.2 mpii.h File Reference

Header to use insted of mpi.h.

```
#include <mpi.h>
```

Defines

- #define **MPI_Init** Inj_MPI_Init
- #define **MPI_Send** Inj_MPI_Send
- #define **MPI_Recv** Inj_MPI_Recv
- #define **MPI_Sendrecv** Inj_MPI_Sendrecv
- #define **MPI_Get_count** Inj_MPI_Get_count
- #define **malloc** Inj_malloc
- #define **free** Inj_free

Functions

- int Inj_MPI_Init (int ∗, char ∗∗∗)

  Initialize MPI and injection.

- int **Inj_MPI_Send** (void ∗, int, MPI_Datatype, int, int, MPI_Comm)
- int **Inj_MPI_Recv** (void ∗, int, MPI_Datatype, int, int, MPI_Comm, MPI_Status ∗)
- int Inj_MPI_Sendrecv (void ∗, int, MPI_Datatype, int, int, void ∗, int, MPI_Datatype, int, int, MPI_Comm, MPI_Status ∗)

  Overridden MPI_Sendrecv().

- int Inj_MPI_Get_count (MPI_Status ∗, MPI_Datatype, int ∗)

  Overridden MPI_Get_count().

- void ∗ Inj_malloc (size_t size)

  Overridden malloc().

- void Inj_free (void ∗ptr)

  Overridden free().

C.2.1 Detailed Description

Header to use insted of mpi.h.

This header, when used instead of (or after) mpi.h will override the functions we replace with their auto-optimized variants.

This is only necesarry for static injection.

Definition in file mpii.h.

C.2.2 Function Documentation

C.2.2.1 void Inj_free (void ∗ *ptr*)

Overridden free().

This is an overridden free() which replaces the pointer with the "orignal" if this memory slab was allocated through our overridden Inj_malloc().

Definition at line 937 of file layer.cpp.

References INJ_METHOD, mallocmap, mallocpages, mallocsizes, Orig_free, Orig_MPI_Init, and PAGE_SIZE.

C.2.2.2 void∗ Inj_malloc (size_t *size*)

Overridden malloc().

This is an overridden malloc() which ensures the allocated area is to page boundries. Without this, small dynamic allocations (such as those done by the MPI library itself) could end up in the same page as the original buffer, meaning they too would be inaccessible once we mprotect().

However, if MPI_Init hasn't been called yet, this just chains through immediately. If called via LD_PRELOAD, we would also be injected into mpirun, and it doesn't make sense to pagealign that.

Definition at line 913 of file layer.cpp.

References INJ_METHOD, mallocmap, mallocpages, mallocsizes, Orig_malloc, Orig_MPI_Init, PAGE_MASK, and PAGE_SIZE.

C.2.2.3 int Inj_MPI_Get_count (MPI_Status $*$ *status*, MPI_Datatype *datatype*, int $*$ *count*)

Overridden MPI_Get_count().

This waits for the request to actually finish, then chains through.

Definition at line 889 of file layer.cpp.

References dbgout(), INJ_METHOD, Orig_MPI_Get_count, statuses, and wait_req().

C.2.2.4 int Inj_MPI_Init (int $*$ *argc*, char $***$ *argv*)

Initialize MPI and injection.

This is the overridden versionof MPI_Init(), and will initialize the signal handler and status variables in addition to calling the original MPI_Init().

Definition at line 784 of file layer.cpp.

References dbgout(), do_debug, do_nothing, Inj_init_chain(), INJ_METHOD, Orig_MPI_Init, rank, sa, and sigact().

C.2.2.5 int Inj_MPI_Sendrecv (void $*$ *sendbuf*, int *sendcount*, MPI_Datatype *sendtype*, int *dest*, int *sendtag*, void $*$ *recvbuf*, int *recvcount*, MPI_Datatype *recvtype*, int *source*, int *recvtag*, MPI_Comm *comm*, MPI_Status $*$ *status*)

Overridden MPI_Sendrecv().

If do_nothing is true, this just transforms the request into a Irecv and Isend, but waits on them immediately before returning.

During normal operation, add_req() is called for both the send and receive parts of the request before we return.

Definition at line 862 of file layer.cpp.

References add_req(), dbgout(), do_nothing, INJ_METHOD, MEM_VALID_TEST, Orig_MPI_Irecv, and Orig_MPI_Isend.

C.3 stattest.cxx File Reference

Test MPI_Get_count.

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#include "mpii.h"
```

Functions

- **main** (int argc, char **argv)

Variables

- int **rank**
- int **nump**

C.3.1 Detailed Description

Test MPI_Get_count.

This tests the chaining of MPI_Get_count in a simple one-to-one transfer.

Definition in file stattest.cxx.

Appendix D

MPII Page Documentation

D.1 Todo List

File layer.cpp Use MPI_Irsend for MPI_Rsend. Or maybe not, as programs which use MPI_Rsend are kinda optimized already.

If seen>10, make it a persistant request.

Runtime option to mprotect() entire area of a chain.

LD_PRELOAD oveeride for operator new.

Appendix E

Program Code

E.1 Red-Black 2D SOR Solver

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#include <string.h>

// Static injection test
#include "mpii.h"

int rank;
int nump;
bool show = false;

#define DO_SIMPLE
#undef DO_ISEND
#undef DO_OVERLAP


void printnum(double v, double min=-1.0, double max=1.0) {
  if (v < min)
    v = min;
  if (v > max)
    v = max;
  double r = (v-min)/(max-min);

  bool bold = false;
  char c = 0;

  // Mørkblå, blå, mørk cyan, cyan, gul, mørk gul, rød, mørk rød, hvit

  if (v == 0.0) {
  } else if (r == 0.0) {
    c = 30;
    bold = true;
  } else if (r == 1.0) {
    c = 37;
    bold = true;
  } else if (r < 0.125) {
    c = 34;
  } else if (r < 0.250) {
    c = 34;
    bold = true;
  } else if (r < 0.375) {
    c = 36;
  } else if (r < 0.5) {
    c = 36;
    bold = true;
  } else if (r < 0.625) {
    c = 33;
    bold = true;
  } else if (r < 0.750) {
    c = 33;
  } else if (r < 0.875) {
    c = 31;
    bold = true;
  } else {
    c = 31;
  }

  fprintf(stderr,"%c[%dm", 27, c);
  if (bold)
    fprintf(stderr,"%c[1m",27);
  fprintf(stderr,"%6.3f␣", v);
  fprintf(stderr,"%c[0m", 27);
}

struct Vector {
  friend class Matrix;
protected:
  double *vector;
  int dim;
public:
  Vector(int dim) {
    this->dim=dim;
    vector=(double *) malloc(dim * sizeof(double));
    memset(vector, sizeof(double) * dim, 0);
  }
  ~Vector() {
    free(vector);
  }
  operator double *() {
    return vector;
  }
  operator const double *() const {
    return vector;
  }
};

struct Matrix {
protected:
  Vector v;
  int _n, _m;
public:
  Matrix(int m, int n) : v(n*m) {
    _n = n;
    _m = m;
  }
  int n() const {
    return _n;
  }
  int m() const {
    return _m;
  }

  operator double *() {
    return v;
  }
  double *operator[](int i) {
    return v+_n*i;
  }
  const double * operator [](int i) const {
    return v+_n*i;
  }
};

/*
 * Mode of operation:
 * If f == NULL, assume the case where f(x,y) = 1
 * If u == NULL, return umax
 * If u != NULL, compare the estimated u with the
 * provided u and report emaxx
 */

typedef double (*PoissonFunc) (double, double);

double fone(double, double) {
  return 1.0L;
}

#define UPDATE(j,i) u[j][i]=(s/4.0) * (u[j-1][i]+u[j+1][i]+u[j][i-1]+u[j][i+1]) + (1.0 - s) * u[j][i]

void sorsolve(int n, int steps = 10, PoissonFunc ffunc = NULL, PoissonFunc ufunc = NULL)
```

```
{
  double h;
  int i, j, t;
  double iv, jv;

  const double s= 1.8;

  Matrix u(n,n);

  h    = 1.0L/(double)(n-1);

  if (ffunc == NULL)
    ffunc = fone;

  for (j=0; j < n; j++) {
    for (i=0; i < n; i++) {
      iv = (double) (i) * h;
      jv = (double) (j) * h;
      u[j][i] = ffunc(iv,jv);
    }
  }

  for(t=1;t<=steps;t++) {
    printf("Iter␣%d␣(omega␣=␣%f)\n", t, s);

    for(j=1;j< n-1; j++) {
      for(i=(j & 1) ? 1 : 2;i<n-1;i+=2) {
        UPDATE(j, i);
      }
    }
    for(j=1;j< n-1; j++) {
      for(i=(j & 1) ? 2 : 1;i<n-1;i+=2) {
        UPDATE(j, i);
      }
    }

    for(j=0;j< n; j++) {
      for(i=0;i<n;i++) {
        printnum(u[j][i]);
      }
      printf("\n");
    }
  }
}

void reptiming(const int steps, double *elapsed, double *com_elapsed, const char *what) {
  double *min, *max, *avg, *com_min, *com_max, *com_avg;
  min = new double[steps];
  max = new double[steps];
  avg = new double[steps];
  com_min = new double[steps];
  com_max = new double[steps];
  com_avg = new double[steps];
  int i;

  MPI_Allreduce(elapsed, min, steps, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
  MPI_Allreduce(elapsed, max, steps, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
  MPI_Allreduce(elapsed, avg, steps, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  for(i=0;i<steps;i++)
        avg[i] /= nump;

  MPI_Allreduce(com_elapsed, com_min, steps, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
  MPI_Allreduce(com_elapsed, com_max, steps, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
  MPI_Allreduce(com_elapsed, com_avg, steps, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  for(i=0;i<steps;i++)
        com_avg[i] /= nump;

  if (rank == 0) {
    printf("%s\n",what);
    for(i=0;i<steps;i++)
        printf("Iter␣%3d␣%f␣->␣%f␣[%f]␣(%fcpu␣%fcom)\n", i, min[i], max[i], avg[i], avg[i]-com_avg[i],com_avg[i]);
  }

  delete [] min;
  delete [] max;
  delete [] avg;
  delete [] com_min;
  delete [] com_max;
  delete [] com_avg;
}

// Upper left is red
```

```
void parasorsolve(const int n, const int steps = 10, PoissonFunc ffunc = NULL, PoissonFunc ufunc = NULL)
{
  double h;
  int lx, ly;
  int i, j, t;
  double iv, jv;
  double umax;
  const double s= 1.4;

  const int showrank=show ? ((nump==1) ? 0 : nump-1) : -1;

  int dims[2];
  int periods[2] = {0, 0};
  MPI_Comm comm_cart;
  MPI_Group group_cart;
  int rank_cart;
  int coord_cart[2];
  MPI_Datatype type_column, type_row;
  MPI_Status status;
  int rank_left, rank_right, rank_up, rank_down;

  double start, stop, com_start, com_stop;

  double *elapsed = new double[steps];
  double *com_elapsed = new double[steps];

  dims[0] = dims[1] = 0;
  MPI_Dims_create(nump, 2, dims);

  if ((n % dims[0]) || (n % dims[1])) {
    if (rank==0)
      printf("Failed␣to␣divide␣%dx%d␣grid␣into␣%dx%d␣nodes\n", n, n, dims[0],dims[1]);
    return;
  }

  lx = n / dims[0];
  ly = n / dims[1];

  if ((lx %2) || (ly % 2)) {
    if (rank==0)
        printf("Can't␣create␣datatypes␣with␣uneven␣red/black␣lengths\n");
    return;
  }


  if (rank == 0)
    printf("Organized␣into␣%dx%d␣nodes␣(%dx%d␣points,␣%dx%d␣total)\n",dims[0],dims[1],lx,ly,n,n);

  MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, true, &comm_cart);
  MPI_Comm_group(comm_cart, &group_cart);
  MPI_Group_rank(group_cart, &rank_cart);
  MPI_Cart_coords(comm_cart, rank_cart, 2, coord_cart);

  MPI_Type_vector(ly / 2, 1, (lx+2) * 2, MPI_DOUBLE, &type_column);
  MPI_Type_commit(&type_column);

  MPI_Type_vector(lx / 2, 1, 2, MPI_DOUBLE, &type_row);
  MPI_Type_commit(&type_row);

  MPI_Cart_shift(comm_cart, 0, 1, &rank_left, &rank_right);
  MPI_Cart_shift(comm_cart, 1, 1, &rank_up, &rank_down);

  Matrix u(ly+2,lx+2);

  h     = 1.0L/(double)(n+1);

  if (ffunc == NULL)
    ffunc = fone;

  for (j=0; j <= ly+1; j++) {
    for (i=0; i <= lx+1; i++) {
      iv = (double) (i+coord_cart[0]*lx) * h;
      jv = (double) (j+coord_cart[1]*ly) * h;
      u[j][i] = ffunc(iv,jv);
    }
  }

#ifdef DO_SIMPLE
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Barrier(comm_cart);

  for(t=0;t<steps;t++) {
    if (rank==showrank) {
```

```
                fflush(stdout);
                fflush(stderr);
            fprintf(stderr,"Iter␣%d\n", t);
        }

    com_elapsed[t] = 0.0;
    start = MPI_Wtime();

    // Exchange black borders

    com_start = MPI_Wtime();

    MPI_Sendrecv(&u[1][2], 1, type_row, rank_up, 0, &u[ly+1][2], 1, type_row, rank_down, 0, comm_cart, &status);
    MPI_Sendrecv(&u[ly][1], 1, type_row, rank_down, 0, &u[0][1], 1, type_row, rank_up, 0, comm_cart, &status);

    MPI_Sendrecv(&u[2][1], 1, type_column, rank_left, 0, &u[2][lx+1], 1, type_column, rank_right, 0, comm_cart, &status);
    MPI_Sendrecv(&u[1][lx], 1, type_column, rank_right, 0, &u[1][0], 1, type_column, rank_left, 0, comm_cart, &status);

    com_stop = MPI_Wtime();
    com_elapsed[t] += com_stop - com_start;

    // Compute red
    for(j=1;j<=ly; j++) {
      for(i=(j&1)?1:2;i<=lx;i+=2) {
        UPDATE(j, i);
      }
    }

    // Exchange red borders

    com_start = MPI_Wtime();

    MPI_Sendrecv(&u[1][1], 1, type_row, rank_up, 0, &u[ly+1][1], 1, type_row, rank_down, 0, comm_cart, &status);
    MPI_Sendrecv(&u[ly][2], 1, type_row, rank_down, 0, &u[0][2], 1, type_row, rank_up, 0, comm_cart, &status);

    MPI_Sendrecv(&u[1][1], 1, type_column, rank_left, 0, &u[1][lx+1], 1, type_column, rank_right, 0, comm_cart, &status);
    MPI_Sendrecv(&u[2][lx], 1, type_column, rank_right, 0, &u[2][0], 1, type_column, rank_left, 0, comm_cart, &status);

    com_stop = MPI_Wtime();
    com_elapsed[t] += com_stop - com_start;

    // Compute black
    for(j=1;j<=ly; j++) {
      for(i=(j&1)?2:1;i<=lx;i+=2) {
        UPDATE(j, i);
      }
    }

    stop = MPI_Wtime();
    elapsed[t] = stop-start;

    if (rank==showrank)
    for(j=0;j<=ly+1; j++) {
      for(i=0;i<=lx+1;i++) {
        printnum(u[j][i]);
      }
      fprintf(stderr,"\n");
    }
  }

  reptiming(steps, elapsed, com_elapsed, "Regular␣sendrecv");

#endif

  MPI_Request req[8];
  MPI_Status statuses[8];

#ifdef DO_ISEND

  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Barrier(comm_cart);

  for(t=0;t<steps;t++) {
    if (rank==showrank)
            printf("Iter␣%d␣(ISend)\n", t);

    com_elapsed[t] = 0.0;
    start = MPI_Wtime();

    // Exchange black borders

    com_start = MPI_Wtime();
    MPI_Irecv(&u[ly+1][2], 1, type_row, rank_down, 0, comm_cart, &req[0]);
    MPI_Irecv(&u[0][1], 1, type_row, rank_up, 0, comm_cart, &req[1]);
    MPI_Irecv(&u[2][lx+1], 1, type_column, rank_right, 0, comm_cart, &req[2]);
    MPI_Irecv(&u[1][0], 1, type_column, rank_left, 0, comm_cart, &req[3]);

    MPI_Isend(&u[1][2], 1, type_row, rank_up, 0, comm_cart, &req[4]);
    MPI_Isend(&u[ly][1], 1, type_row, rank_down, 0, comm_cart, &req[5]);
    MPI_Isend(&u[2][1], 1, type_column, rank_left, 0, comm_cart, &req[6]);
    MPI_Isend(&u[1][lx], 1, type_column, rank_right, 0, comm_cart, &req[7]);

    MPI_Waitall(8, req, statuses);

    com_stop = MPI_Wtime();
    com_elapsed[t] += com_stop - com_start;

    // Compute red
    for(j=1;j<=ly; j++) {
      for(i=(j&1)?1:2;i<=lx;i+=2) {
        UPDATE(j, i);
      }
    }

    // Exchange red borders

    com_start = MPI_Wtime();

    MPI_Irecv(&u[ly+1][1], 1, type_row, rank_down, 0, comm_cart, &req[0]);
    MPI_Irecv(&u[0][2], 1, type_row, rank_up, 0, comm_cart, &req[1]);
    MPI_Irecv(&u[1][lx+1], 1, type_column, rank_right, 0, comm_cart, &req[2]);
    MPI_Irecv(&u[2][0], 1, type_column, rank_left, 0, comm_cart, &req[3]);

    MPI_Isend(&u[1][1], 1, type_row, rank_up, 0, comm_cart, &req[4]);
    MPI_Isend(&u[ly][2], 1, type_row, rank_down, 0, comm_cart, &req[5]);
    MPI_Isend(&u[1][1], 1, type_column, rank_left, 0, comm_cart, &req[6]);
    MPI_Isend(&u[2][lx], 1, type_column, rank_right, 0, comm_cart, &req[7]);

    MPI_Waitall(8, req, statuses);

    com_stop = MPI_Wtime();
    com_elapsed[t] += com_stop - com_start;

    // Compute black
    for(j=1;j<=ly; j++) {
      for(i=(j&1)?2:1;i<=lx;i+=2) {
        UPDATE(j, i);
      }
    }

    stop = MPI_Wtime();
    elapsed[t] = stop-start;


    if (rank==showrank)
    for(j=0;j<=ly+1; j++) {
      for(i=0;i<=lx+1;i++) {
        printnum(u[j][i]);
      }
      printf("\n");
    }
  }

  reptiming(steps, elapsed, com_elapsed, "ISend/IRecv");

#endif

#ifdef DO_OVERLAP

  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Barrier(comm_cart);

  // Exchange black borders
  MPI_Irecv(&u[ly+1][2], 1, type_row, rank_down, 0, comm_cart, &req[0]);
  MPI_Irecv(&u[0][1], 1, type_row, rank_up, 0, comm_cart, &req[1]);
  MPI_Irecv(&u[2][lx+1], 1, type_column, rank_right, 0, comm_cart, &req[2]);
  MPI_Irecv(&u[1][0], 1, type_column, rank_left, 0, comm_cart, &req[3]);

  MPI_Isend(&u[1][2], 1, type_row, rank_up, 0, comm_cart, &req[4]);
  MPI_Isend(&u[ly][1], 1, type_row, rank_down, 0, comm_cart, &req[5]);
  MPI_Isend(&u[2][1], 1, type_column, rank_left, 0, comm_cart, &req[6]);
  MPI_Isend(&u[1][lx], 1, type_column, rank_right, 0, comm_cart, &req[7]);

  for(t=0;t<steps;t++) {
    if (rank==showrank)
```

```
        printf("Iter␣%d␣(ISend␣overlap)\n", t);

com_elapsed[t] = 0.0;
start = MPI_Wtime();

// Wait for black borders
com_start = MPI_Wtime();
MPI_Waitall(8, req, statuses);
com_stop = MPI_Wtime();
com_elapsed[t] += com_stop - com_start;

// Compute red borders
for(i=1;i<=lx;i+=2) {
   UPDATE(1, i);
   UPDATE(ly, i+1);
}

for(j=2;j<ly;j+=2) {
   UPDATE(j+1, 1);
   UPDATE(j, lx);
}

// Exchange red borders
com_start = MPI_Wtime();
MPI_Irecv(&u[ly+1][1], 1, type_row, rank_down, 0, comm_cart, &req[0]);
MPI_Irecv(&u[0][2], 1, type_row, rank_up, 0, comm_cart, &req[1]);
MPI_Irecv(&u[1][lx+1], 1, type_column, rank_right, 0, comm_cart, &req[2]);
MPI_Irecv(&u[2][0], 1, type_column, rank_left, 0, comm_cart, &req[3]);

MPI_Isend(&u[1][1], 1, type_row, rank_up, 0, comm_cart, &req[4]);
MPI_Isend(&u[ly][2], 1, type_row, rank_down, 0, comm_cart, &req[5]);
MPI_Isend(&u[1][1], 1, type_column, rank_left, 0, comm_cart, &req[6]);
MPI_Isend(&u[2][lx], 1, type_column, rank_right, 0, comm_cart, &req[7]);
com_stop = MPI_Wtime();
com_elapsed[t] += com_stop - com_start;

// Compute red interior
for(j=2;j<ly; j++) {
  for(i=(j&1)?3:2;i<lx;i+=2) {
    UPDATE(j, i);
  }
}


// Wait for red borders
com_start = MPI_Wtime();
MPI_Waitall(8, req, statuses);
com_stop = MPI_Wtime();
com_elapsed[t] += com_stop - com_start;

// Compute black borders
for(i=1;i<=lx;i+=2) {
   UPDATE(1, i+1);
   UPDATE(ly, i);
}

for(j=2;j<ly;j+=2) {
   UPDATE(j, 1);
   UPDATE(j+1, lx);
}

// Exchange black borders
com_start = MPI_Wtime();
MPI_Irecv(&u[ly+1][2], 1, type_row, rank_down, 0, comm_cart, &req[0]);
MPI_Irecv(&u[0][1], 1, type_row, rank_up, 0, comm_cart, &req[1]);
MPI_Irecv(&u[2][lx+1], 1, type_column, rank_right, 0, comm_cart, &req[2]);
MPI_Irecv(&u[1][0], 1, type_column, rank_left, 0, comm_cart, &req[3]);

MPI_Isend(&u[1][2], 1, type_row, rank_up, 0, comm_cart, &req[4]);
MPI_Isend(&u[ly][1], 1, type_row, rank_down, 0, comm_cart, &req[5]);
MPI_Isend(&u[2][1], 1, type_column, rank_left, 0, comm_cart, &req[6]);
MPI_Isend(&u[1][lx], 1, type_column, rank_right, 0, comm_cart, &req[7]);
com_stop = MPI_Wtime();
com_elapsed[t] += com_stop - com_start;

// Compute black interior
for(j=2;j<ly; j++) {
  for(i=(j&1)?2:3;i<lx;i+=2) {
    UPDATE(j, i);
  }
}

stop = MPI_Wtime();
```

```
    elapsed[t] = stop-start;

    if (rank==showrank)
    for(j=0;j<=ly+1; j++) {
      for(i=0;i<=lx+1;i++) {
        printnum(u[j][i]);
      }
      printf("\n");
    }
  }

  reptiming(steps, elapsed, com_elapsed, "ISend/IRecv␣overlapped");

#endif

  delete [] elapsed;
  delete [] com_elapsed;
}

double examplef(double x, double y) {
/*
  if (x == 0.5 && y == 0.5)
        return 1.0;
  return 0.0;
*/
  // Wiley coyote problem
  if (x == 0.0 || x == 1.0)
        return -1.0;
  if (y == 0.0)
        return -1.0;
  if (y == 1.0)
        return 1.0;
  return 0.0;
}

double exampleu(double x, double y) {
  return sin(M_PI * x) * sin(2.0L * M_PI * y);
}


main(int argc, char **argv )
{
  int n = 8;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nump);

  if ((rank == 0) && (argc == 2))
    n = atoi(argv[1]);

  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

  if (n < 20)
        show = true;

//  sorsolve(n, 40, examplef, exampleu);

  parasorsolve(n, 20, examplef, exampleu);

  MPI_Finalize();
}
```

E.2 Status validation test

```
/*! \file stattest.cxx
 * \brief Test MPI_Get_count
 *
 * This tests the chaining of MPI_Get_count in a simple
 * one-to-one transfer.
 */
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>

// Static injection test
```

```
#include "mpii.h"

int rank;
int nump;

main(int argc, char **argv )
{
  int n = 8;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nump);

  if (nump != 2) {
        fprintf(stderr, "nump␣!=␣2\n");
        exit(-1);
  }

  unsigned char *buffer=(unsigned char *) malloc(128);
  unsigned char *rbuffer=(unsigned char *) malloc(128);
  for(int i=0;i<128;i++)
        buffer[i]=rank*128+i;

  fprintf(stderr,"Rank␣%d:Alloc␣%p␣%p\n", rank,buffer,rbuffer);

  int other_rank = (rank + 1) % 2;

  MPI_Status status;
  MPI_Sendrecv(buffer, 128, MPI_UNSIGNED_CHAR, other_rank, 0, rbuffer, 128, MPI_UNSIGNED_CHAR, other_rank, 0, MPI_COMM_WORLD, &status);
  fprintf(stderr,"Rank␣%d:␣Oooout\n",rank);

  int size;
  MPI_Get_count(&status, MPI_UNSIGNED_CHAR, &size);
  fprintf(stderr,"Rank␣%d:␣Size␣%d\n", rank, size);

  int sum = 0;
  int valid = 0;
  for(int i=0;i<128;i++) {
        sum += buffer[i];
        sum += rbuffer[i];
        valid += i;
        valid += (i+128);
  }
  fprintf(stderr,"Rank␣%d:␣Sum␣%d␣Valid␣%d\n", rank, sum, valid);

  MPI_Finalize();
}
```

E.3 MPI replacement header

```
/*! \file mpii.h
 * \brief Header to use insted of mpi.h
 *
 * This header, when used instead of (or after) mpi.h will override the functions
 * we replace with their auto-optimized variants.
 *
 * This is only necesarry for static injection.
 */

#ifndef _INJ_MPII_H
#define _INJ_MPII_H

#include <mpi.h>

/* Make sure the include works in both C and C++ programs! */

#if defined(__cplusplus)
extern "C" {
#endif

/* Prototypes for our injected versions, identical to the originals. */

int Inj_MPI_Init(int *, char ***);
int Inj_MPI_Send(void *, int, MPI_Datatype, int, int, MPI_Comm);
int Inj_MPI_Recv(void *, int, MPI_Datatype, int, int, MPI_Comm, MPI_Status *);
int Inj_MPI_Sendrecv(void *, int, MPI_Datatype, int, int, void *, int, MPI_Datatype, int, int, MPI_Comm, MPI_Status *);
int Inj_MPI_Get_count(MPI_Status *, MPI_Datatype, int *);
void *Inj_malloc(size_t size);
void Inj_free(void *ptr);

#if defined(__cplusplus)
}
#endif

/* Use preprocessor trick to use our version instead of the original. */

#define MPI_Init Inj_MPI_Init
#define MPI_Send Inj_MPI_Send
#define MPI_Recv Inj_MPI_Recv
#define MPI_Sendrecv Inj_MPI_Sendrecv
#define MPI_Get_count Inj_MPI_Get_count

#define malloc Inj_malloc
#define free Inj_free

#endif
```

E.4 Injected library

```
/*! \file layer.cpp
 * \brief Main library for the pagefault assisted automatic tuning.
 *
 * This is the main portion of the library which handles the automatic
 * tuning of MPI code. The library can be injected either dynamically
 * (via LD_PRELOAD) or statically (by including mpii.h and linking
 * with the generated injlib.a.
 *
 * Note that most variables, functions and classes in this file are
 * limited to the file-scope (static), to avoid polluting the
 * function namespace of the process we're injecting into, and
 * also allow the compiler as much flexibility with inlining as
 * it cares to do.
 *
 * \todo Use MPI_Irsend for MPI_Rsend. Or maybe not, as programs
 *       which use MPI_Rsend are kinda optimized already.
 * \todo If seen>10, make it a persistant request.
 * \todo Runtime option to mprotect() entire area of a chain.
 * \todo LD_PRELOAD oveeride for operator new.
 */

#include <mpi.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <dlfcn.h>
#include <limits.h>
#include <signal.h>
#include <sys/mman.h>
#include <errno.h>
#include <set>
#include <map>

using namespace std;

//! \brief Size of a hardware page, in bytes.
#define PAGE_SIZE 16384

//! \brief And-mask for aligning an address to the start of page.
#define PAGE_MASK 0xffffffffffffc000

//! \brief Minimum number of stable iterations before recognizing chain
#define CHAIN_MIN_ITER 3

// Make sure either INJECT_STATIC or INJECT_DYNAMIC is set, but not both
#ifndef INJECT_STATIC
#ifndef INJECT_DYNAMIC
#error Must set injection type
#endif
#else
#ifdef INJECT_DYNAMIC
#error Can not have both injection types
#endif
#endif

// If we're dynamically injecting, don't use a two-stage wrapper, just
// replace the function directly. We'll use dynamic loader tricks to
// find the right address.

#ifdef INJECT_DYNAMIC
#define Inj_MPI_Init MPI_Init
#define Inj_MPI_Send MPI_Send
#define Inj_MPI_Recv MPI_Recv
#define Inj_MPI_Sendrecv MPI_Sendrecv
#define Inj_MPI_Get_count MPI_Get_count
#define Inj_malloc malloc
#define Inj_free free
#endif

//! \brief Calling convention and name mangling of exported symbols.
#define INJ_METHOD extern "C"

//! \brief Holds the rank of the process that should output debug information.
static int do_debug;

//! \brief If nonzero, the user has requested no "messing" should take place.
static int do_nothing;

//! \brief Rank of current process.
static int rank;

//! \brief Pointer to original MPI_Init() function.
static int (*Orig_MPI_Init) (int *, char ***);

//! \brief Pointer to original MPI_Isend() function.
static int (*Orig_MPI_Isend) (void *, int, MPI_Datatype, int, int, MPI_Comm, MPI_Request *);

//! \brief Pointer to original MPI_Irecv() function.
static int (*Orig_MPI_Irecv) (void *, int, MPI_Datatype, int, int, MPI_Comm, MPI_Request *);

//! \brief Pointer to original MPI_Wait() function.
static int (*Orig_MPI_Wait) (MPI_Request *, MPI_Status *);

//! \brief Pointer to original MPI_Get_count() function.
static int (*Orig_MPI_Get_count) (MPI_Status *, MPI_Datatype, int *);

//! \brief Pointer to original malloc() function.
static void *(*Orig_malloc)(size_t);

//! \brief Pointer to original free() function.
static void (*Orig_free)(void *);

//! \brief Information about new and old signal handler.
static struct sigaction sa;
static struct sigaction oldsa;

/*! \brief A range of memory.
 *
 * This is a convenience class to work with an manipulate a range
 * of memory (such as the buf variable of most MPI calls).
 *
 * As many compilers don't like pointer arithmetics with void * pointers,
 * this makes the rest of the codemuch cleaner.
 */
struct Range {
        //! \brief Start of range.
        void *start;
        //! \brief End of range.
        void *stop;

        //! \brief Null constructor.
        Range() {
                start = stop = NULL;
        }

        /*! \brief Basic contructor.
         * \param start Start of range.
         * \param stop End of range.
         */

        Range(void *start, void *stop) {
                this->start = start;
                this->stop = stop;
        }

        /*! \brief Compare two ranges
         * \param o Other range to compare with.
         * \return true if ranges are identical.
         */
        bool operator ==(const Range &o) const {
                return ((o.start == start) && (o.stop==stop));
        }

        /*! \brief Check if two ranges overlap (intersect).
         * \param o Other range to compare with.
         * \return true if ranges overlap.
         */
        bool overlaps(const Range &o) const {
                return ( ((o.start >= start) && (o.start <= stop)) ||
                         ((o.stop >= start) && (o.stop <= stop)));
        }

        /*! \brief Check if address is inside range.
         * \param ptr Address to check.
         * \return true if address is in range.
         */
        bool overlaps(void *ptr) const {
                return ((ptr >= start) && (ptr <= stop));
        }

        /*! \brief Adjust range to page boundries.
         * \return New adjusted range.
         *
         * start is adjusted downwards to the start of page boundry, and
         * stop is likewise adjusted upwards.
         */
        Range pageAdjust() const {
                Range o((void *)((long int)start & PAGE_MASK), (void *)(((long int)stop & PAGE_MASK) + (PAGE_SIZE - 1)));
                return o;
        }

        /*! \brief Compute size of range.
         * \return Size of range in bytes.
         */
        size_t size() const {
                return (long int)stop-(long int)start;
        }
};

/*! \brief A memory area in use by an active MPI request.
 *
 * This struct holds all the information about an active request.
 */

struct MemReq {
        //! \brief Indicates if is a MPI_*send or MPI_*recv request.
        bool is_send;

        //! \brief Is the memory area mprotect()ed?
        bool is_protected;
        /*! \brief Is this request the end of a chain?
         *
         * If this is true, this is the end of a chain and we should
         * wait for all outstanding requests to finish before
         * moving along.
         */
        bool is_waitall;

        //! \brief Memory range (in user program) of request.
        Range reqmem;
        //! \brief Memory range of request, page-aligned for
        // fast access.
        Range pages;

        //! \brief Base address used in request, identical to reqmem.start.
        void *base;
        //! \brief Number of elements in request.
        int count;
        //! \brief Datatype used in request.
        MPI_Datatype datatype;
        //! \brief Request handle.
        MPI_Request req;
        //! \brief Communicator used in request.
        MPI_Comm comm;
        //! \brief Status (if any)
        MPI_Status *status;

        //! \brief Size of buffer for MPI_Pack and MPI_Unpack
        int bufsize;
        //! \brief Allocated buffer for packing/unpacking data.
        unsigned char *buffer;
        /*! \brief Construct a new active request.
         * \param is_send Is this request a MPI_*send?
         * \param base Buffer to send/recv to.
         * \param count Number of elements.
         * \param datatype Datatype.
         * \param comm Communicator to use.
         * \param status Pointer to status variable.
         * This will initialize reqmem and pages based on the datatype and
         * count, and will allocate a buffer for packing and unpacking.
         */

        MemReq(bool is_send, unsigned char *base, int count, MPI_Datatype datatype, MPI_Comm comm, MPI_Status *status) {
                MPI_Aint extent, lb, ub;

                this->base = base;
                this->count = count;
                this->datatype = datatype;
                this->comm = comm;
                this->is_send = is_send;
                this->status = status;

                MPI_Type_lb(datatype, &lb);
                MPI_Type_ub(datatype, &ub);
                MPI_Type_extent(datatype, &extent);
                MPI_Pack_size(count, datatype, comm, &bufsize);

                MPI_Aint size = (count-1) * ub + extent;

                reqmem.start = base + lb;
                reqmem.stop = base + size;
                pages = reqmem.pageAdjust();

                buffer=new unsigned char[bufsize];

                is_protected = true;
                is_waitall = false;
        }

        /*! \brief Destructor.
         *
         * Deallocates memory for buffer.
         */
        ~MemReq() {
                delete [] buffer;
        }
};

/*! \brief Signature of MPI Request.
 *
 * This is the signature of an MPI request. If two signatures are identical,
 * they should transfer the same range of memory in the same way to the same
 * destination. This is used to recognize requests we've seen before on
 * subsequent iterations in the user program.
 *
 * The member variables here will be the parameters seen to the MPI_Send
 * or MPI_Recv calls.
 *
 * This struct has no virtual functions, no parent and should not have
 * any children as it relies on being "just a chunk of memory" for
 * its assignment and comparison operators.
 */

struct ReqSig {
        //! \brief Is this is a send request?
        bool is_send;
        //! \brief Base memory address of request.
        void *base;
        //! \brief Number of elements.
        int count;
        //! \brief Datatype
        MPI_Datatype datatype;
        //! \brief Source or destination rank.
        int partner;
        //! \brief Message tag.
        int tag;
        //! \brief Communicator handle.
        MPI_Comm comm;

        /*! \brief Construct a new signature.
         * \param is_send Is this a send requst?
         * \param base The initial address of the send/receive buffer.
         * \param count Number of elements in buffer.
         * \param datatype Datatype of each element.
         * \param partner Rank of source or destination.
         * \param tag Message tag.
         * \param comm Communicator handle.
         *
         * This constructor just does assignments, so it should be
         * reasonably fast. C++ will provide copy and assignment
         * constructors for us (which will basically be memcpy()
         * versions).
         */
        ReqSig(bool is_send, void *base, int count, MPI_Datatype datatype, int partner, int tag, MPI_Comm comm) {
                this->is_send=is_send;
                this->base=base;
                this->count=count;
                this->datatype=datatype;
                this->partner=partner;
                this->tag=tag;
                this->comm=comm;
        }

        /*! \brief Compare two signatures.
         * \param o Other signature to compare with.
         * \return true if signatures are identical.
         * \see operator<()
         *
         */
        bool operator==(const ReqSig &o) const {
                return(memcmp(this, &o, sizeof(ReqSig))==0);
        }

        /*! \brief Compare two signatures.
         * \param o Other signature to compare with.
         * \return true if this signature is "less than" o.
         * \see operator==()
         *
         * This is primarily to allow use as key in std::map<>.
         */
        bool operator<(const ReqSig &o) const {
                return(memcmp(this, &o, sizeof(ReqSig))==-1);
        }

};

/*! \brief Information about a request we've seen.
 *
 * The library tracks and records the most recent requests done
 * by the user program. This is used to build information
 * about request chains; requests that always follow each other
 * without any computation inbetween them. Such chains can
 * then be started with their original buffer area, removing
 * the overhead of copying, and we simply wait at the end of
 * the chain for all requests to finish.
 *
 * Chains are recognized in add_req(), and are simply a series
 * of requests where prevcount and nextcount are fairly high,
 * and the chain terminates in the request which has faultat
 * set.
 */
struct ReqInfo {
        //! \brief How many times this exact request has been seen.
        int seen;
        //! \brief Next and previous request in chain.
        struct ReqInfo *prev, *next;
        //! \brief How many times have we seen the same next and previous requests.
        int prevcount, nextcount;
        //! \brief Address of page fault while this request was last active request.
        void *faultat;
        //! \brief If true, some earlier request specified this as chain end.
        bool waitall;
        ReqInfo() {
                seen=0;
                prev = next = NULL;
                prevcount=nextcount = 0;
                faultat = NULL;
                waitall = false;
        }
};

//! \brief Set of active requests.
static set<MemReq *> requests;

//! \brief Map of statuses we might wait for.
static map<MPI_Status *, MemReq *> statuses;
```

```
//! \brief Map between returned page-aligned address and "true" address.
static map<void *, void *> mallocmap;
//! \brief Map of malloc sizes (number of bytes)
static map<void *, size_t> mallocsizes;
//! \brief Set of pages we've pre-aligned (page number, not address).
static set<unsigned long int> mallocpages;

//! \brief Map of signatures and matching information.
static map<ReqSig, ReqInfo *> reqhist;

//! \brief Last active request.
static ReqInfo *last_req = NULL;

/*! \brief Output debug string
 * \param format format of string (printf format).
 *
 * This outputs a debug string to stderr if the environment
 * variable INJ_DEBUG equals the rank of the calling process.
 */

static void dbgout(const char *format, ...)
{
        static int line = 1;

    if ((do_debug == rank) || (do_debug==-2)) {
        va_list args;


        fprintf(stderr, "%d[%d]:␣", rank,line++);
        va_start(args, format);
        vfprintf(stderr, format, args);
        va_end(args);
        fprintf(stderr, "\n");
    }
}

/*! \brief Wait for request and release memory.
 * \param mr Request to wait for.
 *
 * This will wait for an MPI request to finish, then
 * wait for any requests using the same pages it has
 * locked, and finally unprotect the pages so
 * they can be accessed by the application again.
 * If the request was a receive, the data will be
 * copied from the buffer to the user buffer.
 *
 * If the request was part of chain and not
 * protected, just wait for it finish.
 */

static void wait_req(MemReq *mr) {
        MPI_Status status;
        if (mr->status)
                statuses.erase(mr->status);
        else
                mr->status = &status;

        dbgout("MPI_Wait␣%p␣..␣status␣%p",mr,mr->status);
        MPI_Wait(&mr->req, mr->status);
        requests.erase(mr);

        // Memory ranges cannot overlap, but pages might, and we don't want
        // to unprotect the pages of requests that aren't done, so wait
        // for them too.

        set<MemReq *>::const_iterator iter;
        bool ok;

        do {
                ok = true;
                for(iter=requests.begin(); iter != requests.end(); ++iter) {
                        if ((*iter)->pages.overlaps(mr->pages) && (*iter)->is_protected) {
                                dbgout("Found␣partner␣%p␣to␣wait␣for", *iter);
                                wait_req(*iter);
                                ok = false;
                                break;
                        }
                }
        } while (!ok);

        // Is this a unprotected request? (Part of chain)
        if (! mr->is_protected) {
                dbgout("Unprotected,␣early␣exit");
                delete mr;
                return;
        }

        // Get number of bytes transferred
        int bsize = 0;
        Orig_MPI_Get_count(mr->status, MPI_PACKED, &bsize);


        // Unprotect pages
        mprotect(mr->pages.start, mr->pages.size(), PROT_READ | PROT_WRITE);

        // Copy from buffer to userprogram if this was a receive.
        if (! mr->is_send) {
                dbgout("Going␣to␣copy");
                if (!bsize || bsize == MPI_UNDEFINED)
                        return;
                int pos = 0;
                MPI_Unpack(mr->buffer, bsize, &pos, mr->base, mr->count, mr->datatype, mr->comm);
        }

        dbgout("Goint␣to␣delete␣and␣return");
        delete mr;
}

/*! \brief Wait for all outstanding requests.
 *
 * This waits for all outstanding requests, used at the end of chains and
 * in the case of "true" segment faults.
 */
void wait_all_req() {
        set<MemReq *>::iterator iter;

        dbgout("Wait␣all␣req");

        while((iter=requests.begin())!=requests.end())
                wait_req(*iter);
}

/*! \brief Start new request.
 * \param is_send Is this a send request?
 * \param base Base address of request.
 * \param count Number of elements.
 * \param datatype Datatype of elements.
 * \param rank Source or destination rank.
 * \param tag Message tag.
 * \param comm Communicator handle.
 * \param status Pointer to status receptor.
 * \param ignore Address of MemReq for which we ignore overlapping. (SendRecv pairs)
 * \return Address of MemReq
 *
 * This adds a new tracked request. First, it updates chain information, and
 * if there is sufficient confidence this is part of a chain, skip
 * memory protection alltogether. If this is the end of a chain, wait
 * for all outstanding requests to finish before returning.
 *
 * For new requests, or requests not part of a chain, make sure any requests
 * which use the same memory addresses are already done (unless they were
 * a send, and this is also a send).
 */

static MemReq *add_req(bool is_send, void *base, int count, MPI_Datatype datatype, int rank, int tag, MPI_Comm comm, MPI_Status *sta
        if (rank == MPI_PROC_NULL)
                return NULL;

        ReqSig rs(is_send,base,count,datatype,rank,tag,comm);
        ReqInfo *ri;
        bool nopage;

        // Find previous ReqInfo for this signature, or make a new one.
        ri = reqhist[rs];
        if (! ri) {
                ri = new ReqInfo;
                reqhist[rs] = ri;
        }
        ri->seen++;

        // Was the previous request the same as last time?
        if (ri->prev == last_req) {
                ri->prevcount++;
        } else {
                if (ri->prev) {
                        ri->prev->next = NULL;
                ri->prev->nextcount = 0;
        }
        ri->prev = last_req;
        ri->prevcount=1;
}

// Was this the "next" request of the previous
// request lasttime?
if (last_req) {
        if (last_req->next == ri) {
                last_req->nextcount++;
        } else {
                last_req->next = ri;
                last_req->nextcount++;
        }
} else if (ri->prev) {
        ri->prev = NULL;
        ri->prevcount = 0;
}

last_req = ri;

// Test confidence of chain. If it is a chain,
// we have high nextcounts.
ReqInfo *rptr = ri;
int steps = 1;
while (rptr->next && rptr->nextcount>=CHAIN_MIN_ITER) {
        rptr = rptr->next;
        steps++;
}

// Did we end up at something that pagefaulted?
if (rptr && rptr->faultat && rptr != ri && steps>=2) {
        rptr->waitall = true;
        nopage = true;
} else {
        nopage = false;
}

dbgout("ReqInfo␣%p:␣Next␣%p␣prev␣%p␣Ncnt␣%d␣Pcnt␣%d␣steps␣%d␣Waitall␣%d␣Npage␣%d",ri, ri->next, ri->prev, ri->nextcount, ri->prevcount,

// Allocate new active request.
MemReq *mr = new MemReq(is_send,(unsigned char *)base,count,datatype,comm,status);

if (nopage || ri->waitall)
        mr->is_protected = false;
if (ri->waitall)
        mr->is_waitall = true;

bool memlap = false;

// If memory ranges overlap, wait for request to finish
set<MemReq *>::const_iterator iter;
bool ok;
do {
        ok = true;

        for(iter=requests.begin();ok && iter!=requests.end();++iter) {
                if ((*iter)->pages.overlaps(mr->pages) && ((mr->is_protected || (*iter)->is_protected))) {
                        if (mr->is_protected && (*iter)->is_protected)
                                memlap = true;
                        else if (*iter == ignore) {
                                // This is a sendrecv pair and they disagree, which
                                // is troublesome. No choice but to equate them, possibly waiting.
                                mr->is_protected = (*iter)->is_protected;
                                if (! mr->is_protected)
                                        mr->is_waitall = true;
                                ok = false;
                                break;
                        } else {
                                // Overlapping and different types.
                                wait_req(*iter);
                                ok = false;
                                break;
                        }
                }
        }
} while (!ok);

// If it is a send request with protected memory, copy into sending buffer
int pos = 0;

if (memlap && is_send)
```

```
                mprotect(mr->pages.start, mr->pages.size(), PROT_READ | PROT_WRITE);

        if (is_send && mr->is_protected)
                        MPI_Pack(base, count, datatype, mr->buffer, mr->bufsize, &pos, comm);

        if (mr->is_protected) {
                // Mark pages
                if (mprotect(mr->pages.start, mr->pages.size(), PROT_NONE) != 0) {
                        perror("MPII␣failed␣to␣access␣protect␣pages!");
                        abort();
                }

                if (is_send)
                        Orig_MPI_Isend(mr->buffer, pos, MPI_PACKED, rank, tag, comm, &mr->req);
                else
                        Orig_MPI_Irecv(mr->buffer, mr->bufsize, MPI_PACKED, rank, tag, comm, &mr->req);
        } else {
                if (is_send)
                        Orig_MPI_Isend(base, count, datatype, rank, tag, comm, &mr->req);
                else
                        Orig_MPI_Irecv(base, count, datatype, rank, tag, comm, &mr->req);
        }

        requests.insert(mr);
        if (status)
                statuses[status] = mr;

        // If this was the end of a chain, wait for everything to be done.
        if (mr->is_waitall) {
                wait_all_req();
                last_req = NULL;
        }

        return mr;
}

/*! \brief Segment violation signal handler.
 * \param signal Signal number.
 * \param siginfo Information about signal.
 * \param ucontext User context.
 *
 * This handles page faults, which most likely are caused by the user
 * program accessing protected pages.
 *
 * We simply iterate the active requests, wait for all requests that
 * overlap the faulted address to finish. wait_req() will unprotect
 * the pages again, so we simply return to userspace when done.
 */

static void sigact(int signal, siginfo_t *siginfo, void *ucontext) {
        void *addr = siginfo->si_addr;

        bool found = false;

        dbgout("Sigaction");

        // If memory ranges overlap, wait for request to finish
        set<MemReq *>::const_iterator iter;
        bool ok;
        do {
                ok = true;
                for(iter=requests.begin();iter!=requests.end();++iter) {
                        dbgout("Testing␣%p", *iter);
                        if((*iter)->pages.overlaps(addr)) {
                                found = true;
                                ok = false;
                                dbgout("Found.␣Waiting.");
                                wait_req(*iter);
                                break;
                        }
                }
        } while (!ok);

        dbgout("Did␣iter");

        // If we didn't find anything, this is a "real" segment violation,
        // so we should abort.
        if (! found) {
                do_debug = rank;
                dbgout("MPII:␣SIGSEGV␣--␣trying␣to␣chain\n");

                if (sigaction(SIGSEGV, &oldsa, &sa) != 0) {
                        perror("MPII:␣restore␣sigaction");
                }

                wait_all_req();
        }

        dbgout("Did␣check");

        // Mark the last active request as a end−of−chain candidate.
        if (last_req) {
                last_req−>faultat = addr;
                last_req = NULL;
        }
        dbgout("Sigact␣ret");
}

/∗! \brief Initialize injection.
 ∗
 ∗ The guts of this function will differ betweenthe static and dynamic
 ∗ injection types, but their function is to initialize Orig_MPI_Init
 ∗ and related functions.
 ∗/

#ifdef INJECT_DYNAMIC
static void Inj_init_chain()
{
    // Use RTLD_NEXT to resolve in "the next library down the line"
    // which is either the original MPI library or other MPI tracer
    // code. This way we're as friendly as possible.
    Orig_MPI_Init = (int (∗)(int∗, char∗∗∗))dlsym(RTLD_NEXT, "MPI_Init");
    Orig_MPI_Isend = (int (∗)(void∗, int, unsigned int, int, int, unsigned int, MPI_Request∗))dlsym(RTLD_NEXT, "MPI_Isend");
    Orig_MPI_Irecv = (int (∗)(void∗, int, unsigned int, int, int, unsigned int, MPI_Request∗))dlsym(RTLD_NEXT, "MPI_Irecv");
    Orig_MPI_Wait = (int (∗)(MPI_Request∗, MPI_Status∗))dlsym(RTLD_NEXT, "MPI_Wait");
    Orig_MPI_Get_count = (int (∗)(MPI_Status ∗, MPI_Datatype, int ∗))dlsym(RTLD_NEXT, "MPI_Get_count");
    if (!Orig_MPI_Init) {
        fprintf(stderr, "MPII␣failed␣to␣find␣original␣MPI_Init\n");
        abort();
    } else {
        fprintf(stderr, "MPII␣.so␣injection\n");
    }
}
#else
static void Inj_init_chain()
{
    // Simply resolve the symbols at link time.
    Orig_MPI_Init = MPI_Init;
    Orig_MPI_Isend = MPI_Isend;
    Orig_MPI_Irecv = MPI_Irecv;
    Orig_MPI_Wait = MPI_Wait;
    Orig_MPI_Get_count = MPI_Get_count;
    Orig_malloc = malloc;
    Orig_free = free;
}
#endif

/∗! \brief Initialize MPI and injection.
 ∗
 ∗ This is the overridden versionof MPI_Init(), and will
 ∗ initialize the signal handler and status variables
 ∗ in addition to calling the original MPI_Init().
 ∗/

INJ_METHOD int Inj_MPI_Init(int ∗argc, char ∗∗∗argv)
{

    if (getenv("INJ_DEBUG"))
        do_debug = atoi(getenv("INJ_DEBUG"));
    else
        do_debug = −1;

    if (getenv("INJ_IGNORE") && atoi(getenv("INJ_IGNORE")))
        do_nothing = 1;
    else
        do_nothing = 0;

    Inj_init_chain();


  sa.sa_sigaction = sigact;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = SA_SIGINFO;


    int ret=Orig_MPI_Init(argc, argv);
```

```
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        MPI_Bcast(&do_debug, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(&do_nothing, 1, MPI_INT, 0, MPI_COMM_WORLD);

  if (sigaction(SIGSEGV, &sa, &oldsa) != 0) {
    perror("MPII:␣sigaction");
  }

    dbgout("MPII:␣All␣is␣in␣order.");

    fprintf(stderr, "MPII␣Init␣(debug␣%d,␣ignore␣%d)\n", do_debug, do_nothing);

        return ret;
}

//! \brief Check if x is inside our overriden malloc()ed pages.
#define MEM_VALID_TEST(x) (mallocpages.find((unsigned long int)x / PAGE_SIZE) != mallocpages.end())

INJ_METHOD int Inj_MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
{
    if (do_nothing || !MEM_VALID_TEST(buf)) {
            MPI_Request req;
            MPI_Status stat;

            Orig_MPI_Isend(buf, count, datatype, dest, tag, comm, &req);
            return Orig_MPI_Wait(&req, &stat);
    }

    dbgout("MPI_Send␣chained␣to");
        add_req(true, buf, count, datatype, dest, tag, comm, NULL);
        return MPI_SUCCESS;
}

INJ_METHOD int Inj_MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status * status)
{
    if (do_nothing || !MEM_VALID_TEST(buf)) {
            MPI_Request req;
            Orig_MPI_Irecv(buf, count, datatype, source, tag, comm, &req);
            return Orig_MPI_Wait(&req, status);
    }

    dbgout("MPI_Recv␣chained␣to");
    add_req(false, buf, count, datatype, source, tag, comm, status);
    return MPI_SUCCESS;
}

/*! \brief Overridden MPI_Sendrecv()
 *
 * If do_nothing is true, this just transforms the request into a Irecv and
 * Isend, but waits on them immediately before returning.
 *
 * During normal operation, add_req() is called for both the send and
 * receive parts of the request before we return.
 */
INJ_METHOD int Inj_MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvc
{
    if (do_nothing || !MEM_VALID_TEST(sendbuf) || !MEM_VALID_TEST(recvbuf)) {
            MPI_Request req[2];
        MPI_Status stat[2];
        Orig_MPI_Irecv(recvbuf, recvcount, recvtype, source, recvtag, comm, &req[0]);
        Orig_MPI_Isend(sendbuf, sendcount, sendtype, dest, sendtag, comm, &req[1]);

        MPI_Wait(&req[1], &stat[1]);
        return MPI_Wait(&req[0], status);
    }

    dbgout("MPI_Sendrecv␣chained␣to");

    MemReq *sendreq = add_req(true, sendbuf, sendcount, sendtype, dest, sendtag, comm, NULL);
    add_req(false, recvbuf, recvcount, recvtype, source, recvtag, comm, status, sendreq);

    dbgout("MPI_Sendrecv␣chained␣out");

    return MPI_SUCCESS;
}

/*! \brief Overridden MPI_Get_count()
 *
 * This waits for the request to actually finish, then chains through.
 */
```

```
INJ_METHOD int Inj_MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count) {
        dbgout("MPI_Get_count␣chained");

        map<MPI_Status *, MemReq *>::const_iterator iter;

        iter=statuses.find(status);
        if (iter != statuses.end())
                wait_req((*iter).second);

        return Orig_MPI_Get_count(status, datatype, count);
}

/*! \brief Overridden malloc().
 *
 * This is an overridden malloc() which ensures the allocated area is
 * to page boundries. Without this, small dynamic allocations (such as those
 * done by the MPI library itself) could end up in the same page as the
 * original buffer, meaning they too would be inaccessible once we
 * mprotect().
 *
 * However, if MPI_Init hasn't been called yet, this just chains through
 * immediately. If called via LD_PRELOAD, we would also be injected
 * into mpirun, and it doesn't make sense to pagealign that.
 */
INJ_METHOD void *Inj_malloc(size_t size) {
        if (! Orig_malloc)
                    Orig_malloc = (void *(*)(size_t)) dlsym(RTLD_NEXT, "malloc");
        if (! Orig_MPI_Init)
                return Orig_malloc(size);

        void *base = Orig_malloc(size + 2*PAGE_SIZE);
        unsigned long int adjusted = ((long int)base + PAGE_SIZE - 1) & PAGE_MASK;
        mallocmap[(void *)adjusted] = base;
        mallocsizes[(void *)adjusted] = size;

        for(unsigned long int i=adjusted / PAGE_SIZE;i<=(adjusted+size)/PAGE_SIZE;i++)
                mallocpages.insert(i);

        return (void *) adjusted;
}

/*! \brief Overridden free().
 *
 * This is an overridden free() which replaces the pointer
 * with the "orignal" if this memory slab was allocated
 * through our overridden Inj_malloc().
 */

INJ_METHOD void Inj_free(void *ptr) {
        if (! Orig_free)
            Orig_free = (void (*)(void *)) dlsym(RTLD_NEXT, "free");
        if (! Orig_MPI_Init)
                return Orig_free(ptr);

        void *base=mallocmap[ptr];
        if (base) {
                Orig_free(base);
                unsigned long int adjusted=(long int)ptr;
                for(unsigned long int i=adjusted / PAGE_SIZE;i<=(adjusted+mallocsizes[ptr])/PAGE_SIZE;i++)
                        mallocpages.erase(i);
                mallocsizes.erase(ptr);
        }
        else
                Orig_free(ptr);
        mallocmap.erase(ptr);
}
```

Index