

Run-time Optimization of Sends, Receives and File I/O

Thorvald Natvig

Norwegian University of Science and Technology(NTNU),
Sem Sælands vei 9, NO-7491 Trondheim, NORWAY
Email: Thorvald.Natvig@ntnu.no

Anne C. Elster

Norwegian University of Science and Technology(NTNU),
Sem Sælands vei 9, NO-7491 Trondheim, NORWAY
Email: elster@idi.ntnu.no

Abstract—On today’s commodity clusters, achieving near-optimal speedup is very hard. We have previously shown that parallel applications using synchronous sequential MPI calls can be optimized by runtime replacement with corresponding asynchronous MPI operations. This may be achieved by protecting memory used by the MPI call from application writes until the operation is complete. However, changing the protection bits of memory pages, which alters the page table and flushes the CPU caches, may introduce a significant overhead. In the case of overlapping requests, which are common for applications with domain decompositions using border exchanges of ghost cells, this would have to be done multiple times for each communications phase.

In this paper, we overcome this problem by mapping the same memory twice into the virtual address space, but with different protection bits set for each view. This allows us to optimize overlapping MPI requests without changing the page protection bits until all the requests are finished. The same method is also extended to cover basic file I/O, overlapping file operations with computation without rewriting the original application. We also add distributed locking to I/O operations, which allows aggressive read-ahead and write-merging for parallel applications, reducing the wall-clock time of the file I/O phases that commonly surround the computation of the solution.

I. INTRODUCTION

On a modern cluster using commodity interconnects, latency can seriously impact on application speedup. Passing through the MPI [1] layer, operating system kernel call, networking layers adds considerable overhead, as does traversing the wire itself. For problems where each node requires frequent, small data exchanges, such as 2D decomposed iterative solvers, straight-forward implementations using MPI will spend more time waiting for data than computing.

If the application is rewritten to use asynchronous communication, the effect of latency is reduced. For a 2D case border exchange, each node has to send and receive 4 borders. By doing this in parallel using asynchronous communication instead of sequential, synchronous communication, you effectively wait for the latency in parallel, potentially reducing the effect of latency by a factor 4.

Unfortunately, using asynchronous communication is harder to do correctly than using synchronous communication, and the asynchronous nature means bugs in the program can be intermittent in nature and may not be revealed for small test runs.

A related problem is file I/O. A common pattern for parallel applications is to start by reading a file to get input data, perform computation until a result is received and then write to file. High-performance parallel file operations exist, and in order to take advantage the MPI-2 standard defined the MPI-IO API [2]. However, many existing and new applications still use regular file I/O and use them in a suboptimal manner.

A. Motivation

Parallel applications should perform with optimal efficiency in order to avoid wasting resources on large, shared installations. A common source of inefficiency is using synchronous, sequential communication on each node instead of asynchronous and parallel communication.

We have previously shown that it is possible to runtime optimize synchronous communication into asynchronous communication [3], but the overhead necessary to ensure correct application execution was shown to be substantial. As shown in [4], it is possible to model the overhead and predict when optimizations pay off, which ensures that slowdowns are avoided. This paper focuses on reducing the overhead itself and increasing the speedup.

For some real world applications, a large portion of the execution time is spent in I/O operations, using suboptimal I/O access. If a single node is doing I/O while the others wait, resources are wasted.

B. Related Work

Ogawa and Matsuoka [5] have used compiler modifications to optimize MPI, doing static analysis to create specialized MPI programs. Karwande et.al. [6] have presented a method for compiled communication (CC-MPI), which applies more aggressive optimizations to communications whose parameters are known at compile time. Preissl et.al [7] have demonstrated how to use program communication traces to allow a compiler to optimize synchronous communication into asynchronous. Common for these is that they require the program to be recompiled in a new environment, and often requires traces of previous program runs.

Memory protection bits have been used before to achieve interesting effects. Pakin et.al have, with their JumboMem method [8], shown that the memory of a single program can be distributed among a large number of nodes. The

same underlying principle has been used by Hu et.al [9] to implement OpenMP shared memory across networks of distributed memory machines.

Itzkovitz and Schuster [10] introduce the idea of mapping the same physical memory multiple times into the virtual address space of a process in order to reduce the number of page protection changes.

II. AUTOMATIC ASYNCHRONOUS COMMUNICATION

When a synchronous MPI operation is performed, it is started as an asynchronous operation, and the memory area is protected from application access. If the application tries to access the memory, a memory access violation will occur. We intercept this signal, wait for the asynchronous communication to finish, then unprotect the memory pages and continue application execution.

For send requests, the memory is *read-only* protected. For receive requests, the memory is *no-access* protected and the data is received to a temporary buffer, as *no-access* is also applicable to the MPI library.

Communication frequently uses MPI data types and have buffers that overlap their start and end addresses, but with stride or holes in the layout such that they do not actually touch the same data. The coherency of operations (that is, avoiding sending data that has not been fully received yet) is done by data type overlap analysis.

A. Page re-protection problem

When using page protection, protection can only be set at page granularity, which typically is 4096 bytes. As shown in Figure 1, common conditions exist where all the borders overlap, and there is not a single request that does not share a page with another. This leads to a series of minor problems.

First, if a page has an asynchronous send request in progress when we wish to start a receive operation; we cannot change the protection on the page until the send request is finished. If we unprotect the page, the application might write to it before the data is actually sent. We cannot *no-access* protect it, as that would cause the send operation to crash. Second, if a page has an asynchronous receive request in progress and we wish to send; we cannot read the data we wish to send, as the page is *no-access* protected. We have to unprotect the page, copy the data to a temporary buffer, then reprotect the page. The allocation of temporary buffers and the copying of data is undesirable overhead.

A related problem is that all changes to the page table are expensive as they flush the translation lookaside buffer (TLB) and parts of the cache. Measurements on modern machines reveal little execution time difference between changing 1 and 10 pages; the overhead of changing anything at all dominates.

Using Figure 1 again, we see that pages would be protected and unprotected multiple times as the different calls are made. If the application issues the border exchanges as a series of MPI_Sendrecv calls, the actions performed would be:

1) *Application issues MPI_Sendrecv for top border.*

- 2) Page #1 needs to be marked *no-access*, to ensure the application does not read it. Page #2 would need to be marked *read-only* to avoid the application writing it, but since it shares partial data with page #1, we have to copy it to a temporary buffer. Page #2 is not marked.
- 3) *Application issues MPI_Sendrecv for left border.*
- 4) Page #1 needs to be unprotected, data copied out to a temporary send buffer, and then reprotected. Pages #2, #3 and #4 need to be *no-access* protected.
- 5) *Application issues MPI_Sendrecv for right border.*
- 6) The pages need to be re-protected again.
- 7) *Application issues MPI_Sendrecv for bottom border.*
- 8) Page #4 needs to be unprotected and data copied out. Pages #4 and #5 need to be *no-access* protected.
- 9) *Application accesses data in page #1 and causes a page fault.*
- 10) MPI_Waitany for all requests. Repeat until all requests involving page #1 are finished.
- 11) Unprotect all pages that no longer have active requests.

B. Multi mapped memory

To avoid the multiple calls to change page protections, we replace the application's memory allocation functions. The replacement functions returns memory allocated from a memory pool we map twice in the address space, and allocates new such pools as needed. For memory requests larger than a page, we align the memory on a new page to minimize protection overlap.

This gives the effect that we have two views of the applications memory, both of which are available to our optimization method, but only one of which is readily available to the application. On modern hardware, the protection bits for these two views can be different.

The automatic asynchronous operations can now be done by protecting the application's view of the memory while doing the MPI operation on our private view. This avoids the copying overhead, and also avoids the unprotect-copy-reprotect necessary when issuing a send after a receive operation on the same page. The only overhead of this method is an enlarged page table, which needs 8 bytes per page of multi mapped memory (0.2% of allocated memory).

C. Pre-emptive page protection

When a request is issued by the application, we note the call offset in the application, as well as the previous stack return addresses, the data type, memory offset and transmission size. When a request is executed, we record it, and link it bidirectionally with the previous request to execute.

This will give us a chain of related requests with a communication phase between iterations of computation. Once the same chain has been executed a set number of times (currently 3), we change the method of page protection slightly.

When a request is executed, the memory for that request and all requests following it in the same chain is protected. If all the requests are send requests, we use *read-only* protection,

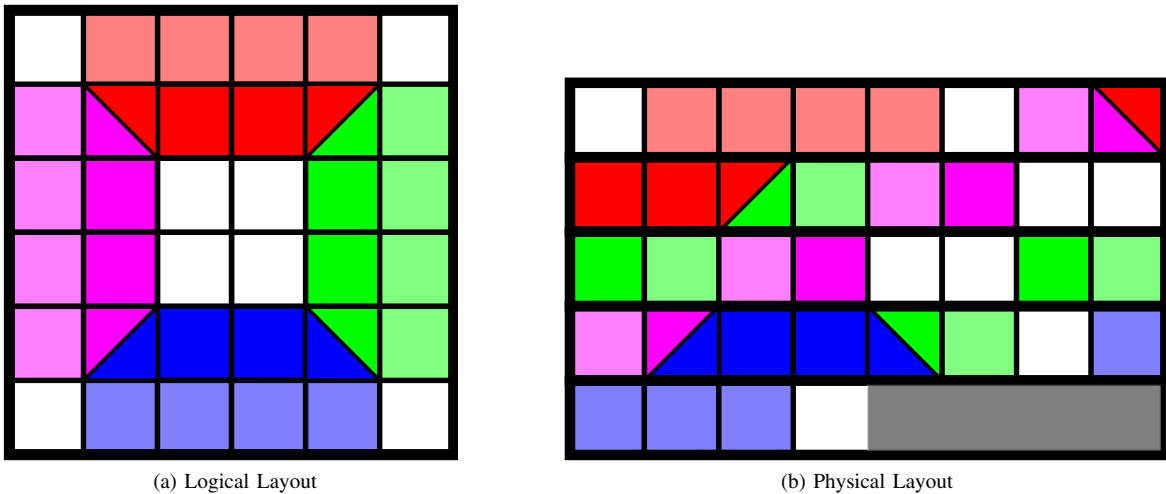


Fig. 1. Logical and physical layout of a 4x4 local domain with shadow cells on memory pages with space for 8 elements per page

but if any of them are receive requests, we use *no-access* protection. The pages are subsequently marked as speculatively protected, so our method knows not to protect them again. If this is a correct prediction, we will have turned multiple page protection calls into a single call. If the prediction is wrong, the only downside is that we'll have protected a few pages too many, increasing the time of the kernel call slightly.

This reduces the program flow to:

- 1) *Application issues MPI_Sendrecv for top border.*
- 2) Based on request history, pages #1 to #5 are protected *no-access* from the application. Request started asynchronously from mirror view.
- 3) *Application issues MPI_Sendrecv for left border.*
- 4) Pages are already speculatively protected, so mark pages as in use by request, but do not change page protection bits. Request started asynchronously.
- 5) *Application issues MPI_Sendrecv for right border.*
- 6) Request started asynchronously.
- 7) *Application issues MPI_Sendrecv for bottom border.*
- 8) Request started asynchronously.
- 9) *Application accesses data in page #1 and causes a page fault.*
- 10) `MPI_Waitall` for the entire request chain.
- 11) Unprotect all pages associated with the chain and allow application to continue.

III. OPTIMIZING FILE ACCESS

Many parallel applications follow a simple pattern for execution:

- 1) Read input data from filesystem
- 2) Compute result
- 3) Write result to filesystem

If point #1 and #3 are done sequentially and synchronously, Amdahl's law clearly states that, even if point #2 is infinitely parallelizable, maximum speedup is limited to $[Sp = \frac{T_{total}}{T_{IO}}]$

A secondary problem is that many users write data in small blocks, often as little as a single element at a time. Whereas

this works for small datasets run on their personal machines, it doesn't scale to datasets that are several gigabytes large and run on parallel filesystems.

A. Overlapping File I/O with computation

Using the same mechanism as for communication, we can mark pages associated with read or write calls as inaccessible and perform the I/O requests in the background while allowing the application to continue. Unlike communication, File I/O can be orders of magnitude slower, and hence the overhead tradeoffs are different.

A parallel application might communicate over the filesystem, using communication to indicate when each process should write. It is possible to let each node writes to the file system in turn, passing a token in a ring with `MPI_Sendrecv` to ensure only one process writes at a time. If we do the IO asynchronously, we risk processes overwriting each other. It is therefore necessary to lock regions of the file that we are currently accessing.

For read requests, we use `fcntl()` with read locking the region. This requires that the underlying shared filesystem implements POSIX locks, which most current remote filesystems do, including NFS on modern Linux kernels. Like page protection, an application can have only one type of lock for a given region at a time, but unlike page protection it operates on a byte granularity.

B. MPI-IO/AIO

MPI-IO, which is implemented on top of AIO on Linux, has asynchronous functions which work with the same logic as its communication counterparts; you issue the I/O operation and get a request object you can wait for.

MPI-IO itself uses file locking, which limits which MPI-IO functions we can use. Indeed, finding which functions use what locking requires knowledge of the underlying MPI library's implementation, and a process can have only one set of locks for any given file, even if the file is opened twice.

We therefore risk that MPI-IO replaces our lock, which can cause data corruption in the output file.

If two large read requests are performed asynchronously at the same time, the file system has to satisfy both in parallel. It does this by alternately reading a large block of data from each file, satisfying part of each request. For a single filesystem on rotational storage, such as a single central filesystem for a cluster, this causes the drive head to move back and forth for each large block. This means two parallel requests can be slower than the same two requests executed sequentially.

If an application accesses memory where a request is in progress, a page fault will occur. At this point, we want to prioritize the request whose memory the application wishes to access. MPI-IO and AIO offer no functionality for prioritization of requests.

C. Worker thread

As an alternative to AIO, we have implemented a set of worker threads to do file I/O, with one thread per filesystem in each MPI process. This allows local and remote file operations to be performed concurrently.

1) *Read operations*: When a new file read request is issued (which will write to memory), we first examine if the memory is already in use by another request. If a file write request currently uses the memory, we copy the remainder of the data the existing write request needs to a temporary buffer. If the memory is already in use by a file read operation, we notify the existing operation that it should skip the memory area for the new request.

Once overlap analysis has been done, the file region is locked to ensure we do not overlap I/O operations with other nodes working on the same file. The read operation is then attempted using non-blocking I/O. If this succeeds, the request has been satisfied from filesystem buffer cache, and we release the lock and allow the application to continue. If the non-blocking read fails, we advise the filesystem that we expect to read the complete area of the request. We then protect the application's view of the memory with *no-access* permissions and add the read request to the worker thread's queue of operations.

2) *Write operations*: When a new file write request is issued (which will read from memory), we first examine if the memory is in use by another request. If it is in use by another write request, we do nothing special. If it is in use by a file read request, we mark the new file write request as depending on the file read request to have finished. We then write-lock the file region and make sure the application's view of the memory is *no-access* protected.

3) *Reordering*: If a page fault occurs in memory associated with a file write request (because the application wants to write to it), we copy the remainder of the file write operation to a temporary buffer, and have the worker thread continue writing from the buffer.

If a page fault occurs in memory associated with a file read request, we move the request to the head of the queue. We also tell the worker thread to change the order in which the

file is read, starting at the offset that caused the page fault to allow the application to continue as early as possible. This ensures that we get a good speedup even for applications that iterate bottom-to-top instead of top-to-bottom.

The worker thread on each filesystem reads and writes data with a predefined block size (1 MB in our tests), and completes each request in the order it was started. File operations are first attempted in non-blocking mode, meaning read operations are satisfied only from the operating system buffer cache, and write operations are only written to the buffer cache; we do not wait for data to be passed to or from the physical disc. When the first such non-blocking operation fails, we unprotect the pages for data that has already been read or written and reissue the operation in a blocking fashion. This ensures that the worker thread reads or writes as large continuous blocks of data that is possible without incurring the overhead of changing page protection bits.

Note that memory copying for overlapping read/write requests is performed only if allocating the buffer for the request would not allocate more memory than is physically available to the process. In this case, we fall back to waiting for requests to finish using incremental page release in the worker thread.

With this set of operations, an application is free to start computing on the start of the dataset even while the remainder of it is still being read from storage. Similarly, for applications which write intermediate results to file, computation can continue, overwriting the working area while the file operation is completed in the background.

D. Merging small operations

If file requests are small, such as when reading a large matrix a row (or an element) at a time, remote filesystem operations become very slow. This can be overcome by using user-space buffering I/O routines (such as that offered by *fread()*, *fwrite()* etc), but it is then necessary to flush your data manually to ensure synchronization.

If multiple, sequential file accesses to the same file is detected, we start aggressive caching for the application. The current heuristic for this is 3 subsequent operations of the same type on the same file descriptor.

For file read operations, we extend the read lock to twice the amount of data that has been read thus far, and advise the filesystem to start caching it to the filesystem buffer cache. Each read request is extended to cover at least a filesystem block (typically 4096 bytes), and subsequent read operations by the application are redirected to this cache.

For file write operations, we also extend the lock to twice the amount of data that has been written thus far. Requests are cached in memory, and are not written to the filesystem until any of the following are true:

- The requests total length cover the worker thread block size.
- 50ms has passed since the last write request.
- The application exits.

It's important that requests are flushed frequently, since other processes might be waiting for the lock.

If the write requests have holes between them, we either write them as individual requests (if there are less than 2 per filesystem block), or we read the filesystem block, copy in the write requests and write the block back to disc.

The above tricks of read-ahead and write-merging are similar methods applied by the kernel when scheduling I/O operations. However, by doing it in user-space, we save expensive kernel context switches, and by ensuring the regions are locked in the filesystem, we allow this to work for parallel systems working on remote filesystems.

If the application issues any *fsync()* or *fdatasync()* calls, these are delayed until the write operation is completed and converted to a single *fdatasync()* call.

IV. PERFORMANCE PREDICTIONS

For most clusters, communication is much faster than file I/O. For file-read followed by a scatter or a series of sends, or for gather or receives followed by a file-write, we can hence assume that the communication time is less than the I/O time.

In general, the expected wallclock time for such an operation is

$$T = T_{I/O} + T_{Comm} \quad (1)$$

In an equally general sense, the expected wallclock time for this operation to complete, when I/O and communication are automatically overlapped, is

$$T = T_{I/O} + (1 - \alpha)T_{Comm} \quad (2)$$

where α signifies the ratio of communication that can be satisfied with partially completed I/O.

For a 2D problem with size n (and complete data area of n^2), we have chosen to examine the expected speedup of three different data distributions.

A. Row distribution

For row distribution on p nodes, data distribution can be done in-order as it is read. Communication and I/O therefore overlap almost perfectly, and the expected total time is

$$T_{rowdist} = T_{I/O} + \frac{1}{p}T_{Comm} \quad (3)$$

B. Column distribution

For column distribution on p nodes, each node requires data both from the beginning and the end of the file. Due to the way rotational storage works, no performance can be gained by reading the data a column at a time, so no attempt is made to optimize this. Time savings is therefore impossible in this scenario, and the speedup is the same as the base case.

C. 2D Cartesian distribution

To minimize the amount of communication that has to take place between each iteration, the optimal distribution is 2D Cartesian, as this reduces the per-iteration communication from 2 send-receives of n data to 4 send-receives of n/\sqrt{p} . This allows communication to scale to a much larger set of nodes.

Only the bottom row of nodes in the 2D Cartesian distribution has to wait for the complete row to be read. The expected total time becomes

$$T_{2D} = T_{I/O} + \frac{1}{\sqrt{p}}T_{Comm} \quad (4)$$

V. RESULTS

We have developed and benchmarked these approaches on the ClustIS3 cluster at NTNU, a small cluster with gigabit network cards, all connected to the same switch. The frontend node acts as the central filesystem, and compute nodes mount the central filesystem over NFS.

For verification, we use a 2D SoR PDE solver [11] with a 2D decomposition with ghost cells as shown in Figure 1, but with problem sizes from $n = 128$ to $n = 32768$. At $n = 32768$, the local data size is 8 GB, the largest that can fit in the cluster's 9 GB of memory per node. On startup, rank #0 reads the input file data and distributes this with a series of MPI_Send. Border exchange is done with MPI_Sendrecv on a Cartesian communicator. Every 1000th iteration, the data is gathered back to rank #0 and written to disc.

As an optimal reference, we changed the border exchange to persistent asynchronous communications, and the I/O operation to use MPI-IO directly on each node.

All these tests were performed on 9 nodes, with the frontend as rank #0. This ensures at least one node does border exchanges in all 4 directions. Filesystem buffer caches were cleared before each test.

A. Communication

Figure 2 shows speedup for the communication phase done between iterations of computation, all results relative to the speed of the original application. This compares the original code, the optimal version of the code and three different optimization strategies.

The first optimization shown is the original page protection method. As can be seen, the overhead of page protection is considerable, yet still gives good results for small problem sizes where the communication latency is dominating. The second optimization is page protection with multiview. This eliminates the unprotect-reprotect cycles, and is the speedup that can be expected for the first iterations. Overhead is still significant, especially for the left and right borders that may involve many pages.

Third and last is optimization with pre-emptive page protection. This reduces the overhead of page protection to two page protection calls per set of related communications, and yields good speedup for a much larger set of problem sizes. However, even this method has overhead, and dips into a slowdown.

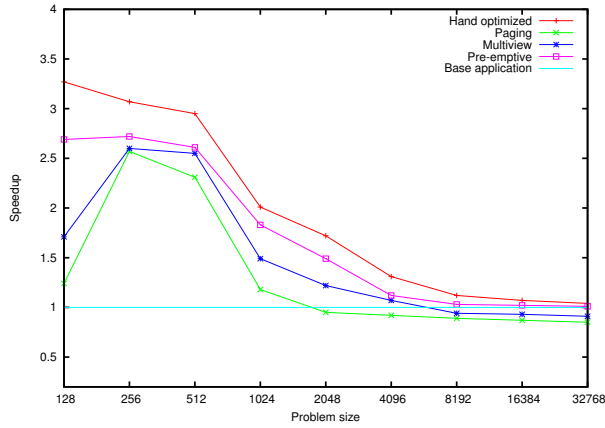


Fig. 2. Communication speedup on 9-node gigabit Ethernet cluster using 2D border exchange with different automatic optimization strategies.

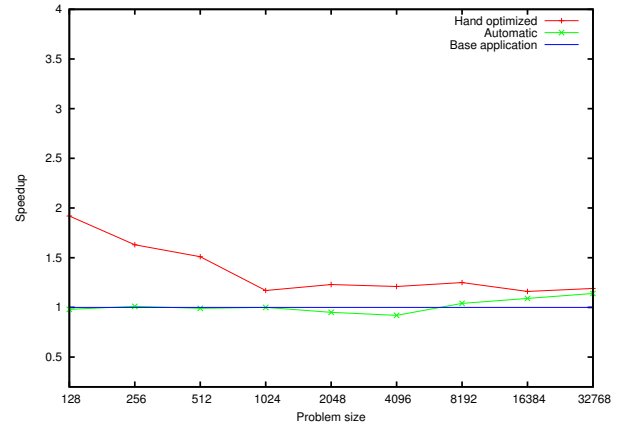


Fig. 4. Data gather and file write speedup before and after optimizations.

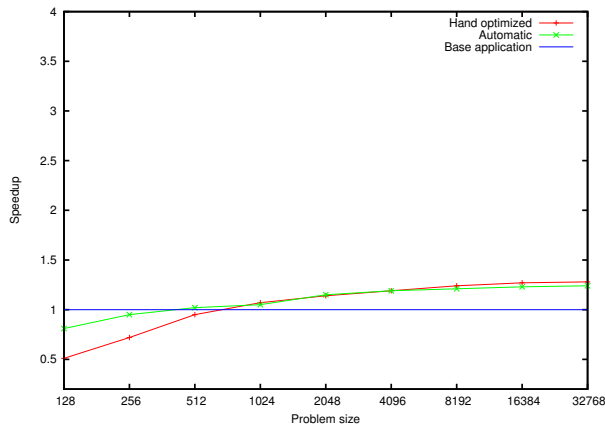


Fig. 3. File read and data distribute speedup before and after optimizations.

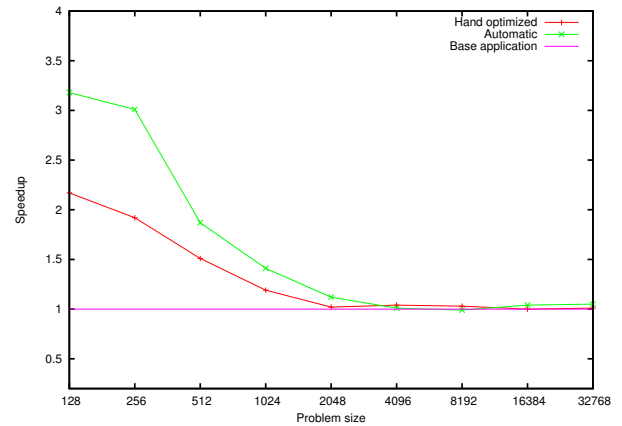


Fig. 5. Distributed read using regular file I/O over NFS.

Note that the optimization predictor shown in [4] ensures an actual slowdown never takes place, but we have disabled the predictor here to clearly show the crossover points where overhead outweighs potential gains.

B. File I/O

Figure 3 shows speedup for the startup of the application (file read and scatter), relative to the speed of the original application. For this test, we have the MPI optimizations turned off, as we wish to analyze the file I/O separately.

For small problem sizes, we see a significant slowdown for both hand-optimized and automatically optimized code. For a problem with size $n = 128$, the complete file is only 131kB large, which is already optimized by the kernel’s filesystem level read-ahead. For the hand-optimized version, the overhead is the NFS traffic of sending the request for the data, and for our optimized version the overhead is page protection.

The speedup achieved from our optimization method is due the fact that the distribution phase of the data can begin before the file has completed reading. As soon as the top third of the data has been read, the MPI calls to send data to rank #1 and #2 can be allowed to continue. The cluster has approximately

a 2 : 1 communication to file-read speed ratio. When the 2/3 of the communication can happen while file I/O is still progressing, we reduce the effect of communication overhead by 2/3, which reduces total wallclock time by 2/9. This in turn should yield a theoretical speedup of $Sp \leq \frac{9}{7}$ which is almost identical with our measured results.

Figure 4 shows the speedup for the gather and file write operation the application performs every 1000th iteration. Since rank #0 is the frontend, the filesystem is local and writes are simply cached by the filesystem buffer cache, meaning we have very little speedup and sometimes a slowdown. It is only when we increase the problem size so that the write operation does not fit in buffer cache that a small speedup is observed.

In Figure 5 we show initialization speedup when each node reads directly over NFS, seeking to and reading a local row of data at a time. The graph includes the time for the first iteration of computation. For small problem sizes, the read-ahead gives a very good speedup, since latency when fetching small data blocks over NFS can be quite high. As the problem size grows, speedup of the I/O is reduced, but since we allow local computation to begin while the bottom of the local file is read, and this meager computational gain means we still deliver wallclock time improvement. The comparison is unfair

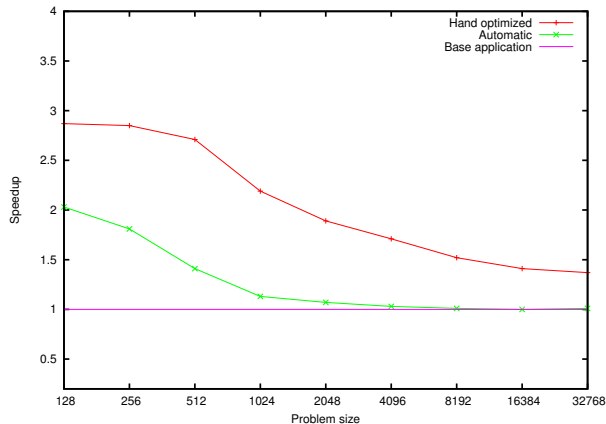


Fig. 6. Distributed write using regular file I/O over NFS.

to the hand-optimized version, as this version could achieve a similar speedup if it was rewritten to interleave the first computation with multiple asynchronous read requests.

Figure 6 shows the speedup of the distributed writes when each application writes directly over NFS. Here the locking and write-merging effectively serializes access, meaning each node will submit its data as a few large NFS requests instead of several small ones. The locking means that the last node to be allowed access to the lock has to wait for the others to finish before its write phase can begin. We still cannot reach the performance of hand-optimized code which uses collective file I/O operations.

If we allow the worker thread to copy the data to be written when the application tries to access it, the application-seen delay for the write operation is reduced to a memory copy. For large problem sizes, this gives file I/O speedup in the 20-30 range, effectively removing the overhead of file writes. We have therefore devised a user-space workaround for some of NFS’ performance problems.

To verify that file integrity was maintained, we modified our solver to write and synchronize local data back to the filesystem for every iteration, and re-reading border cells from neighbors, using only barriers in MPI to synchronize processes. For this case, our method gave a wallclock speedup of 1.2 for the worst case and also yielded identical results to the un-optimized application as well as the MPI-based application.

Furthermore, we’ve run our optimizer on several MPI applications currently in development by fellow researchers. In each case, we gained speedup without any alteration of the results.

VI. DISCUSSION

The communications optimizations we have presented perform well on small problem sizes. When the size increases, communication is no longer bound by latency, and the overhead of changing the page protection bits is greater than any gain we could have.

File optimization works best on large problem sizes, and less so on the very small ones. When combined, the two optimizations give a speedup to most programs.

All of the optimizations presented here assume the original application is unoptimized. For optimized applications that already use asynchronous operations, our injected method only ensures consistency with any synchronous operations it has performed, and otherwise does nothing to the request. This minimal checking overhead is less than 80 cycles on our cluster, and is less than the error of measurement for the complete communications operations.

There are cases where our optimizations fail completely. For example, a program may receive data over MPI or read data from a file. If it then calls a kernel function we do not intercept with this data as a parameter, the kernel function will fail with an error the original application cannot handle. We’ve covered most basic operations in our implementation, but it is not future proof for new kernel function calls.

It is our hope that the massive speedups seen for distributed file write operations will allow researchers to write their temporary results back to the filesystem more often without any significant delay in wallclock execution time. This should make it easier to make jobs that are restartable and hence able to squeeze in a bit of computation now and then during unused cluster hours.

VII. FUTURE WORK

It would be interesting to analyze the data dependency between File I/O operations and MPI operations. For example, whereas we already have good speedup on the *Read File, then MPI_Scatter* pattern of operations, it should be possible to delay the entire file operation until the data is needed and then replace the *MPI_Scatter* call on each node with a *MPI_File_read_ordered* call. This should lead to good speedup on clusters which have parallel filesystems (such as PVFS [12]) installed. Currently, both these methods are part of the same library. Since the filesystem accelerator part is applicable to more than just parallel MPI applications, it can be split out into a separate project.

”In flight” data used in kernel calls while still in a protected memory page is a problem that needs to be addressed. One solution to this could be spawning a separate thread which uses *ptrace()* or similar to trace the main thread, intercepting all kernel calls and watching for addresses that are covered by current asynchronous file requests. If such a request is found, we hold the process until the worker thread has finished.

VIII. CONCLUSION

We have developed, implemented and tested a method for automatically turning synchronous communication into asynchronous communication, with the application and the MPI library operating on separate virtual-memory views of the same physical memory. Each view has separate access protection, ensuring the data dependency of the application is not altered in any way.

We have similarly accelerated file I/O using memory protection and replacement of functions, and using aggressive caching with file locking to increase application performance without altering data dependency flow.

This is done at run-time, without requiring the original application to be recompiled or altered in any way, and works without access to source code.

We have demonstrated that significant speedup can be gained with these optimizations, thus giving application authors and researchers speedup close to that of manually optimized asynchronous code while maintaining the simplicity of sequential, synchronous operations.

ACKNOWLEDGMENT

The authors would like to thank NTNU for access to clusters and filesystems to perform our testing on, and numerous colleagues and students for access to their applications to validate our results. We would also like to thank Jan Christian Meyer and John Ryan for critical and constructive feedback on this paper.

REFERENCES

- [1] M. P. I. Forum, "MPI: A message-passing interface standard," Tech. Rep. UT-CS-94-230, 1994. [Online]. Available: citeseer.ist.psu.edu/519858.html
- [2] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA, USA: MIT Press, 1999.
- [3] T. Natvig and A. C. Elster, "Automatic and transparent optimizations of an application's MPI communication," in *PARA*, ser. Lecture Notes in Computer Science, B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, Eds., vol. 4699. Springer, 2006, pp. 208–217.
- [4] T. Natvig, A. C. Elster, and J. Balla, "Using context-sensitive transmission statistics to predict communication time," in *PARA 2008 (to appear in Lecture Notes in Computer Science 6126/6127)*. Springer, 2010.
- [5] H. Ogawa and S. Matsuoka, "OMPI: optimizing MPI programs using partial evaluation," in *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 1996, p. 37.
- [6] A. Karwande, X. Yuan, and D. K. Lowenthal, "An mpi prototype for compiled communication on ethernet switched clusters," *J. Parallel Distrib. Comput.*, vol. 65, no. 10, pp. 1123–1133, 2005.
- [7] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan, "Transforming mpi source code based on communication patterns," *Future Gener. Comput. Syst.*, vol. 26, no. 1, pp. 147–154, 2010.
- [8] S. Pakin and G. Johnson, "Performance analysis of a user-level memory server," in *CLUSTER '07: Proceedings of the 2007 IEEE International Conference on Cluster Computing*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 249–258.
- [9] Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel, "Openmp for networks of smps," *J. Parallel Distrib. Comput.*, vol. 60, no. 12, pp. 1512–1530, 2000.
- [10] A. Itzkovitz and A. Schuster, "Multiview and millipage – fine-grain sharing in page-based dsms," in *In proceedings of the third USENIX symposium on operating system design and implementation*, 1999, pp. 215–228.
- [11] D. M. Young, *Iterative solution of large linear systems [by] David M. Young*. Academic Press, New York., 1971.
- [12] W. B. I. Ligon and R. B. Ross, "Implementation and performance of a parallel file system for high performance distributed applications," in *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 1996, p. 471.