

# Automatic Run-time Parallelization and Transformation of I/O

Thorvald Natvig, Anne C. Elster and Jan Christian Meyer  
Norwegian University of Science and Technology(NTNU),  
Sem Sælands vei 9, NO-7491 Trondheim, NORWAY  
Email: Thorvald.Natvig@ntnu.no, elster@idi.ntnu.no and janchris@idi.ntnu.no

**Abstract**—As the size of computational clusters grows, one can expect that I/O will consume an increasing portion of wall-clock time as the problem and node sizes are scaled up, unless parallel I/O is introduced. Unfortunately, using parallel I/O is non-trivial, so few applications developed by individual researchers enjoy its benefits.

In this paper, we describe our novel method for analyzing I/O and communication operations at run-time. When nodes perform I/O or communication operations, our technique protects the memory associated with the requests from the application. Subsequent operations are analyzed for overlap between communication and I/O operations. When found, the I/O operation is automatically transformed, by our injected library, from an individual operation to a collective and shared MPI I/O operation.

This allows users to benefit from parallel file systems without redesigning or recompiling their applications, and we demonstrate speedup for common usage patterns.

## I. INTRODUCTION

Most parallel programs work with large files, and large, high-performance I/O systems with matching filesystems have always followed large, high-performance processing power. It is easy to see the advantage of using a parallel environment, such as MPI [1] or OpenMP [2] to gain increased processing power, and you can quickly gain speedup for the computational phase of your program. Such optimizations are frequently implemented, tested and benchmarked on smaller problems running on fewer nodes, and then scaled up to "real" problem sizes on a commodity cluster or custom supercomputer.

In this paper, we attempt to address the problem of I/O on an increasingly large number of nodes, allowing regular, non-optimized code to scale to a substantially larger number of nodes.

### A. Motivation

Amdahl's law states that the maximum speedup achievable is

$$Sp_{max} \leq \frac{p}{1 + f(p - 1)}$$

where  $f$  is the fraction that cannot be parallelized and  $p$  is the expected speedup of the part that is parallelized. For small problems running on a small number of nodes, speedup can be seen even if  $f$  is relatively high. Assuming  $f = 0.1$  and perfect speedup of the computation, a speedup of 4.7 is achievable on 8 nodes. However, the same problem, when run on 128 nodes, achieves a speedup of only 9.3. Hence, the problem does not scale to a larger set of nodes.

One definition of scalability is that speedup should remain constant with a number of nodes linearly proportional to the problem size. For this definition to hold, the  $f$  fraction has to be zero, otherwise it will dominate the denominator for sufficiently high  $p$ .

A common source of sequential and non-parallel code in a program is file I/O. I/O operations are often performed on a single node, before distributing data to the other nodes. Much work has been done to address this common inefficiency, and a portable solution is to use the MPI I/O functions available in MPI 2 [3]. Unfortunately, such functions are hard to use correctly. Since the problems of scalability are only revealed at high node counts, researchers seldom use this advanced functionality when developing their applications.

Clusters continuously become larger, and jobs need to parallelize to a larger number of nodes to maintain resource utilization. This means that scalability problems are becoming a limiting factor for the computational speed of many applications. Frequently, the problem is no longer to get a large enough machine to run on, but to effectively use the large machine you have. While production code which is expected to run for millions of CPU hours can justify the cost of being rewritten by an expert, this is not the case for code under development. Such code still has to be able to scale to run on large, modern machines.

It is our goal to transform an application to use the distributed, parallel I/O functions available in MPI automatically, without the application author having to alter their own code. This should reduce the impact of one of the major sequential portions of the application, thus enabling the application to scale much better.

### B. Related Work

Prost *et.al* have implemented MPI I/O on GPFS [4], and Thakur *et.al* implemented ROMIO [5], which is in use on both OpenMPI [6] and MPICH [7] based clusters.

Liao *et.al* have implemented node-local caching for MPI I/O, which greatly speeds up reads and writes [8].

Itzkovitz and Schuster [9] 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.

We have previously shown that it is possible to automatically convert synchronous communication to asynchronous

communication at run-time, applied to both static and dynamically compiled applications, without altering the application or having access to the application’s source code [10].

### C. Outline

A summary of our previous work on using page protection to improve communication time as well as work on merging I/O requests on clusters with NFS filesystems is presented in Section II. In Section III we present our method for transforming common unoptimized I/O patterns into optimal ones, and Section IV presents performance predictions and measurements of optimal reference cases. Section V presents results when applying our method to the reference cases, and Section VI presents future work which should be performed to perfect this method. Section VII offers our conclusion.

## II. PAGE PROTECTED ASYNCHRONOUS COMMUNICATION

Our method builds on previous methods for altering the communication and basic I/O functions of a program while ensuring the integrity and sequence of computations. We inject a library into the application which replaces several functions, including MPI communications functions, memory management and file I/O. For a dynamic application, we use LD\_PRELOAD to override the functions. For a static application, we scan the memory for static library signatures and replace their start with a trampoline calling our code.

When the application issues a synchronous MPI communication or regular file I/O request, we start the request asynchronously, protect the memory pages to deny the application access to it, and allow the application to continue execution. For a MPI receive or file read request, the memory pages are protected *no-access*, and for a MPI send or file write request, the memory pages are protected *read-only*. This ensures that the application does not alter data before it has been written, and does not read data before it has been received.

If the application accesses the data, it will generate a page fault. Our page fault handler will check if the fault happened in a page we protected. If it did not, regular page fault mechanisms take over, usually causing the application to crash. If it did, we wait for the requests in the corresponding page to finish, unprotect the pages, and allow the application to continue.

### A. Multimapped Memory

There may be multiple requests that share the same memory page without sharing the same memory addresses. For MPI applications with border exchanges, this is almost always the case for the left and right borders. When a new receive request is issued to an already protected page, protection levels may need to be changed (receive follows send) or the data copied (send follows receive).

To keep the number of page protection alterations to a minimum, we map the physical memory of the application twice in the virtual address space. One view is accessible to the application, and the second view is only accessible to our method. This allows us to protect the memory for the

first request. The communication or file operation and any subsequent operation inside the same set of pages will operate on our private view of the memory.

If two requests overlap on physical addresses, we examine the requests. Multiple memory read (MPI send or file write) requests are allowed to continue concurrently, but a memory write (MPI receive or file read) signifies a barrier such that all requests before it must finish, and no more requests may begin on the same address until the request is finished.

### B. Operation Tracking

For each operation we intercept, we note the calling address of the application, the parameters for the operation, and the three previous stack return addresses in the application. This will, in most cases, refer to a unique position in the application, even if the application should abstract its operations inside wrapper functions. We refer to this as the call’s *context*.

The *context* can be used to track repeated sequences issued by the application, and can be used both to predict communication time (as we have done in [11]) and to preemptively protect pages. As computing the *context* is relatively cheap, the tracking has negligible overhead.

### C. Memory Allocation

To decrease the probability of two non-related requests sharing the same memory page, we have replaced the basic memory allocation functions. Our previously published allocators aligned memory on a page barrier. For applications that allocate and release many small memory buffers, this has noticeable overhead on physical memory allocation. It also has the side-effect of increasing the cache contention for cache lines that correspond to the lower entries of a page.

We have experimented with allowing multiple allocations per page once a certain allocation threshold is reached, but setting the threshold becomes an application-specific parameter, and usually requires a benchmarking run to complete.

A new, alternative memory allocator has been developed for the work presented here. First, we allocate memory from a memory pool using a regular allocator. We then create a new mapping of the allocated memory, and return a pointer in this new view to the application. Thus, if the application requests three small chunks of memory *A*, *B* and *C*, the three are allowed to share the same physical page, but the addresses returned to the application will be in three different virtual address pages. This allows us to have different page protection bits for all three requests.

Note that creating additional views is computationally expensive, and adds significant overhead to the memory allocation. Our method looks at the *context* of the allocations; if the allocation has been repeated frequently and no optimization was possible, we revert back to regular memory allocation for that request. Thus we ensure that the overhead cost is only paid if we gain performance from it.

Also note that overrunning the area returned by the allocator will overwrite other arrays, even if they do not appear to be adjacent in the virtual address space. However, any buffer

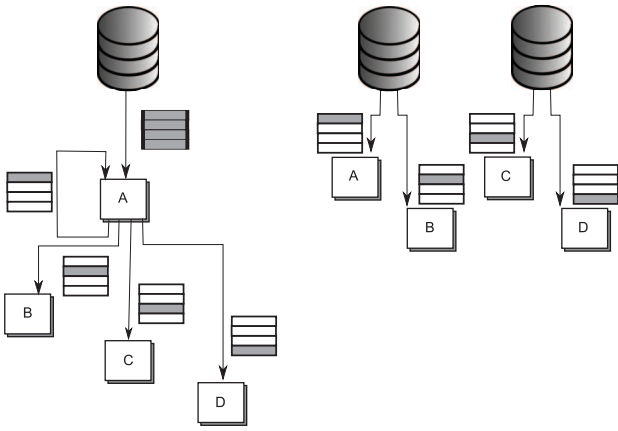


Fig. 1. Read and Scatter (row distribution). Left: Original data flow (Single-node I/O). A reads whole file, then scatters data. Right: Transformed (Parallel I/O).

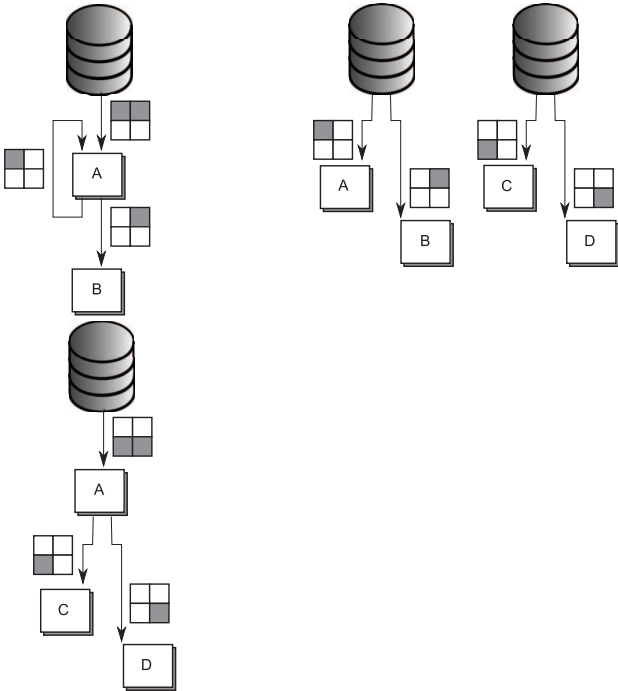


Fig. 2. Read and Send (2D block distribution). Left: Original data flow (Single-node I/O). A reads top half and distributes to A and B, then reads bottom half and distributes to C and D. Right: Transformed (Parallel I/O)

overrun is an application error, so no attempt is made to correct this.

### III. I/O TRANSFORMATIONS

Applications which perform single-node sequential and synchronous file I/O can be automatically transformed at run time to benefit from parallel file I/O. We investigate optimizations for four common patterns, specifically, *read and scatter*, *read and send*, *gather and write*, and *receive and write*. *Read and scatter* is a common method to evenly distribute a file as a set of contiguous blocks, while *read and send* tends to be favored in 2D and 3D domain decomposition problems or

other cases where the distributed data requires a more complex access pattern. *Gather and write* and *receive and write* are the respective inverse operations, which often appear as the natural conclusion of a computation which has used one of the former patterns to distribute data. The read patterns and their appropriate transformations are illustrated in Figs. 1, 2. Note that the matching write pattern is just the reverse of the data flow of the read pattern.

To implement these transformations at run time, we examine file I/O and communication requests as they happen. If a relationship between a file I/O request and a communication request can be established, the communication request is replaced with a MPI I/O request directly to the file. This optimization permits a node which did not originally perform file I/O to be redirected to the file system, allowing parallel file systems to concurrently fulfil requests from multiple nodes.

In order to perform these transformations transparently, we require out-of-band communication to inform participating nodes of the appropriate file, and we must intercept the regular file access functions, such as *read()*, *write()*, and *seek()*. Finally, if the application already uses MPI I/O functions, we must intercept them to ensure that transformations do not interfere with their operation.

Out-of-band communication is implemented using a separate communicator, to handle all control messages. This communicator is a copy of `MPI_COMM_WORLD`. Greater flexibility could be attained by implementing control signals directly over TCP sockets, or by altering the MPI envelope. To ensure that our method is portable to any system where the application would run unmodified, however, we rely only on MPI functions for communication. MPI I/O functions with individual file pointers are treated identically to their regular file I/O counterparts. To handle direct calls to MPI I/O, file pointers are saved at completion, and restored at the next call from the application. This is necessary since our method may have altered the file pointer state between application calls.

The following sections describe optimizations for our four patterns, and discuss some additional considerations to ensure safe and efficient operation.

#### A. Read and Scatter

To find related read and scatter operations, it is necessary to intercept both, and retain the state of the read operation until we can determine whether it should be transformed or not. To do this, we initially protect the application's view of read request memory as *no-access*. If the memory is later scattered, it can be recognized by the intercepted scatter operation. Other access causes page faults which can be trapped, signaling that the read request is not part of a pattern.

When intercepting `MPI_Scatter()`, we must check if the buffer area corresponds to an outstanding file request. If it does, all nodes involved are notified of the file name and base offset on the private communicator. Otherwise, they are signaled to expect a no-op. All recipients in a scatter operation (root included) check the private communicator. When a file message is found, the file is opened and read

with `MPI_File_read_ordered_begin()`, using parameters from the transformed scatter operation. The file descriptor is cached for later use. Using the asynchronous version with matching page protection permits overlapping I/O with computation and unrelated communication.

If a page fault occurs before any related MPI function has been called, we know that the read operation must be issued and we have to wait for it to finish. Page faults resulting from a transformed MPI call are handled by awaiting the completion of the MPI I/O function. The node which issued the read operation keeps the file open, in the event that instructions we do not intercept will cause page faults which require us to read the corresponding pages.

Although this method transparently rewrites the *read and scatter* pattern, the added manipulation of page protection bits causes overhead. Therefore, if the *context* of a scatter call is seen 3 times without optimization taking place, the root node requests a reduction of the other nodes' count. If this is also 3 or greater, the *context* is marked as unoptimizable, avoiding further overhead to other scatter operations.

### B. Read and Send

This pattern is treated similarly to the *read and scatter* case, by protecting the request as *no-access*, intercepting MPI send calls, and sending messages about filename and offset on the private communicator. On the receiving end, MPI receive calls are intercepted and started asynchronously. If a message is received on the private communicator before completion of the asynchronous call, the communication request is cancelled and replaced with a matching MPI I/O request instead. When the send message corresponds to an I/O request, the original communication request is never sent. Thus, the information message always arrives before the receive request is finished. This method involves very little overhead, and no additional messages for non-transformed function calls.

If the node in I/O reuses the same memory area for multiple reads and sends, and calls originate from the same *context*, we do not start read requests asynchronously. Instead, memory is protected *no-access*, and the I/O operation is delayed until a page fault is seen. This allows us to quickly dispatch redirection messages to receiving nodes, without causing the overhead of multiple cancelled I/O requests and page reprotctions.

### C. Gather and Write

Because the collective operation happens before it is known whether or not a file operation follows, this pattern is hard to handle correctly. Our method is to protect memory on participating nodes, and await a message from the root, using a timeout to ensure that it will arrive. The root protects memory, and notifies each node in either of these events:

- An I/O write operation is performed
- The application causes a page fault
- An I/O read or MPI receive operation overwrites the memory area
- 10ms of time pass

If an I/O write operation matches the pattern we are detecting, the root will notify each node of filename and offset. Each node then replaces the gather with `MPI_File_write_ordered_begin()`. Page faults indicate that the gather is part of the application's computation, so `MPI_Gather()` is executed as it was called. An I/O read or receive operation which overwrites the memory area is a symptom of wasteful resource management in the application code. As we have nevertheless observed it in practice, we detect it, and signal each node to ignore the `MPI_Gather()` call at no extra overhead.

If time passes without either event occurring, no clear choice can be made. The application may have issued a superfluous gather operation, or may simply not have reached a stage where the result is needed. In this case, we rewrite the `MPI_Gather()` into a series of asynchronous sends and receives, which proceed in the background. In case one of the other conditions apply before this is completed, the appropriate response can still be taken by either waiting for the transfers, cancelling and transforming, or just cancelling them. If the operation finishes, the memory is unprotected, and subsequent I/O operations are not optimized.

### D. Receive and Write

This pattern is handled by transforming send requests into their asynchronous counterparts, and protecting the memory. The sender does not complete any send until notified by the receiver. Receiving nodes protect associated memory, and start asynchronous receives into their private view of the same area.

If the receiving node sees a page fault, transfers are completed as asynchronous operations using the optimizations in section II, and each node tracks that no optimization was performed. As before, 3 such cases for the same *context* will disable further optimizations. If the receiving node starts an I/O write using the relevant memory area, the receives are cancelled, and the private communicator is used to notify nodes about file name and offsets. The operation is then started as an asynchronous file operation on each node.

As in the *read and send* pattern, we detect if the same memory area is reused. If it is, we delay starting the asynchronous receive operation until a page fault occurs. If no page fault occurs, we save the overhead of starting and cancelling the asynchronous request.

### E. Request Overlapping

For requests involving file write operations, the writing node will analyze data overlap. When data from multiple nodes share cells in the data file, we attempt to modify the write requests to exclude overwritten parts, giving priority to the last received request. If the resulting data structure is non-trivial (i.e. it is not a vector type), the requests are serialized. This modification is necessary to ensure that our method does not introduce any race conditions for which data are eventually written to file.

## F. Merged Operations

When new communications requests are started from the same *context* while optimized I/O requests are in progress, the application might be doing row-at-a-time file I/O. For this reason, each node will only participate in one optimized I/O request at a time. If multiple outstanding requests are detected, we attempt to merge them into one, large file I/O request. This does not reduce the number of communication requests, but it greatly reduces the number of file requests, giving favorable speedup when file system latency is high.

## G. Local to Shared Operations

If the cluster has a high performance parallel filesystem and a node issues a large file read or write request, we protect memory as usual, but also immediately notify all other nodes about the filename and offset. When a pagefault occurs and/or 10ms of time has passed, we check if other nodes perform I/O to the same file, which permits substitution with a collective file request. Collective operations can also be issued asynchronously, so we can protect the memory, and let the application continue with I/O operating in the background.

## H. Local Adaptations

As some implementations of *MPI\_Cancel()* are expensive, our method issues, and immediately cancels, a small request on startup. The time required to complete this is used to decide whether communication will be performed in the background, or delayed until the data is needed. For small requests, MPI I/O implementations operating on NFS perform considerably worse using collective operations on shared file pointers. We therefore check the type of filesystem files are opened on, and for NFS, we will subsequently use *MPI\_File\_read\_at()* instead of *MPI\_File\_read\_ordered()* (and similar for write operations).

## IV. PERFORMANCE PREDICTIONS

We can analyze and predict the expected performance speedups of different patterns, and based on these predictions, our method can dynamically turn optimizations on and off.

### A. Read Operations

For the *read and send* pattern, our only overhead is the page protection. The original cost is highest at the root node;

$$T = T_{I/O}(m, 1) + pT_{Comm}\left(\frac{m}{p}\right)$$

where  $m$  is the total problem size,  $p$  is the number of nodes, and  $T_{Comm}(x)$  is the time to transfer one point-to-point message of  $x$  bytes.

$T_{I/O}(x, y)$  is the time to read  $x$  bytes from the filesystem on each of  $y$  nodes. As an approximation,

$$T_{I/O}(m, p) = \frac{1}{\min(i, p)} T_{I/O}(mp, 1)$$

where  $i$  is the number of dedicated I/O nodes.

When transformed, the cost for this becomes

$$T = 2P(m) + pT_{Comm}(h) + \frac{p}{\min(i, p)} T_{I/O}\left(\frac{m}{p}, p\right)$$

where  $i$  is the maximum I/O speedup of the system. For systems with identical bandwidth to I/O and compute nodes,  $i$  is the number of I/O nodes.  $P(x)$  is the overhead of changing page protection bits of  $x$  bytes of data.  $h$  is the size of a hint message with the filename and offset, and is expected to be small, at most a few hundred bytes. As can be seen, we expect significant speedup when  $i > 1$ , or when  $P(x)$  is low.

If the complete file is too large to fit in memory and 2D distribution is used, a common solution is to read a row of nodes at a time, distribute the data and then read the next row. The cost for this is

$$T = \sqrt{p}T_{I/O}\left(\frac{m}{\sqrt{p}}, 1\right) + pT_{Comm}\left(\frac{m}{p}\right)$$

which is transformed to

$$T = 2P\left(\frac{m}{p}\right) + pT_{Comm}(h) + T_{I/O}\left(\frac{m}{p}, p\right)$$

The *read and scatter* pattern has identical performance predictions to *read and send*, since the scatter call is effectively replaced by the sending of the hint message.

### B. Write Operations

Write operations have the same expected performance as a read operation, but here the  $T_{I/O}$  function will be different, as the filesystem may cache write operations. If asynchronous I/O works, and sufficient memory is available to buffer a copy of data, the expected cost seen by the application becomes

$$T = pT_{Comm}(h) + C\left(\frac{m}{p}\right)$$

where  $C(x)$  is the time to copy  $x$  bytes of memory. In other words, the speedup should be substantial, as our transformation allows I/O to complete asynchronously while the application continues computation.

## V. RESULTS

A problem with automatic optimization is that most existing benchmarks and published applications are already heavily optimized, and several papers are written on how to optimize them even further. It has been a challenge for us to find published and available code that is not optimized, as researchers seem reluctant to publish unoptimized applications. Our unoptimized base codes are therefore copied from the research applications of other local researchers at our university, who have asked not to be named.

To assess the efficiency of automatic transformation, we have run the unoptimized codes with and without automatic transformations enabled, and also compared them to manually optimized versions that use MPI I/O directly. All tests have been run on multiple architectures.

1) *Njord*: The first system is Njord, a Power 5+ cluster with 186 compute nodes, each with 8 Dual-core 1.9Ghz processors and 4 I/O nodes. All I/O operations are performed over GPFS. Nodes are connected with IBM HPS Switches (Federation), giving 3.77 GB/s unidirectional bandwidth. Each node has 32 GB of local memory, but due to job scheduling limitations, only 13 GB are available to normal processes. This system runs IBM AIX 5L 5.3.

2) *Kongull*: The second system is Kongull, a system with dual hex-core Opteron 2431 nodes, gigabit Ethernet connectivity to the compute nodes and 2 I/O nodes, each with 10 Gbit/s connectivity. This system also uses GPFS, but compute and I/O node bandwidth differs, and  $T_{I/O}$  is less predictable than on Njord. Each node has 24 GB of local memory, all of which is available to user applications. This system runs Rocks 5.2 on Linux 2.6.18.

3) *Clustis3*: The third system is Clustis3, with dual quad-core 2Ghz Xeon E5405 nodes and gigabit Ethernet connectivity. This system has no dedicated I/O nodes, but uses the frontend as a filesystem, with NFS on the compute nodes and Ext3 on the frontend. Each node has 8 GB of local memory. Without dedicated I/O nodes, speedup here is lower than on Kongull, but estimating I/O cost is relatively easy. This system runs Rocks 5.3 on Linux 2.6.33.2. Originally, the front-end of Clustis3 had a single regular harddrive. For these tests, the frontend has been equipped with a SSD drive, which moves the bottleneck to the network link.

All systems have a page size of 4096 bytes.

#### A. Testing procedure

Test runs on all systems are limited to 16 compute nodes, as this was enough to fully saturate the available I/O bandwidth, and no additional speedup was seen beyond this point. Additional I/O nodes would be needed to see continued speedup with a growing number of compute nodes. The problem size on each system is limited so that the unoptimized *read* and *scatter* pattern can complete without swapping. This ensures a fair speedup comparison with the unoptimized application. Unfortunately, this limits the total problem size to  $\frac{p}{p+1}$  of available memory. Tests involving *read* and *scatter* or *gather* and *write* patterns are performed with a row distribution, so that each node is responsible for a contiguous area of the file. Tests for the *read* and *send* or *receive* and *write* patterns are performed with a 2D Cartesian distribution, so that each node is responsible for a subrectangle of the global problem domain, and file areas overlap. The unoptimized application reads data for an entire row of nodes at a time and then distributes these. While this may use more memory, it has higher performance than reading a node at a time (which involves  $n$  reads and seeks per node) before sending, and it also has higher performance than reading a row at a time and distributing (which involves  $np$  sends from the I/O node).

All tests are repeated and averaged. A unique file is used for each phase of each repetition of each test. This avoids filesystem caching, both on the compute and the I/O nodes. To focus tests on interconnect and I/O, rather than memory copy, tests are run with one MPI process per node.

For Njord, we had to increase the number of repetitions substantially and filter outliers. It is a large production system which is nearly constantly under full load. I/O from a single compute node usually has repeatable and predictable performance since only a single I/O node is needed, but parallel I/O depends on all I/O nodes being available for the job, something they seldom are. Best- and worst-case performance

differ by up to a factor 4, depending on the number of I/O nodes available at the time of testing.

We also see occasional outliers with a factor of 20 or more, meaning that all I/O nodes were busy and had to be shared. Kongull is a new cluster without any load, and we were easily able to get repeatable results. For Clustis3 we allocated the entire cluster for exclusive use.

#### B. Basic Performance Data

Figure 3 shows the measured overhead of changing page protection bits for various memory sizes on Njord and Kongull, compared to the communication and I/O cost for each size. These data are quick to measure, and are used as the base values for our performance predictions.

It should be noted that, on Njord, the overhead of changing page protection bits is very high compared to Kongull. Wall-clock time for *mprotect()* of a data area is comparable to the time for *MPI\_Send()* of the same data. The optimizations we have earlier performed on communication do not work on this machine; all potential gain is completely overshadowed by the overhead of page protection. Similar high cost is seen for other memory manipulation functions such as *mmap()* itself.

#### C. Transformed Reads

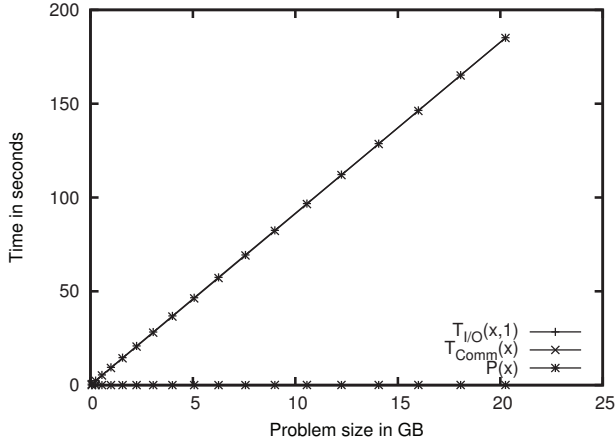
Figure 4 shows the speedup of our automated optimization when applied to a *read* and *scatter* pattern, compared to manually using *MPI\_File\_read\_ordered()*.

On Kongull, we achieve speedup almost identical to that of manual optimization. This is as expected, since the overhead of page protection is very low, and the transmission of control messages is fast. Each I/O node is able to fully saturate the network link of a compute node. This means that the original code uses as much time for data distribution as it does for I/O read. Since our optimization eliminates the data distribution, and parallelizes the I/O read, overall speedup is good. For small problem sizes (not shown in the graph), where each node's domain is only 1MB or less, both the manual optimization and our automatic one result in a slowdown, since the overhead of parallel I/O outweighs the data distribution time.

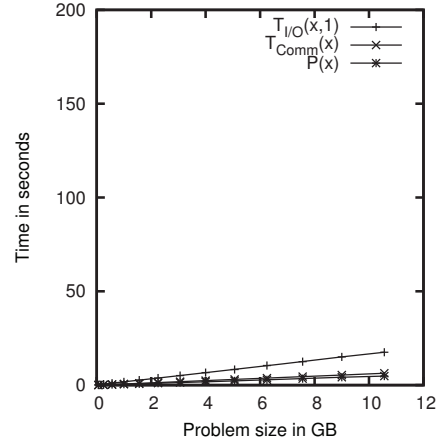
Our predictions on Kongull do not match our measured results. Indeed, while single-node I/O is very predictable, parallel I/O time varies greatly with different data sizes and the number of nodes, and a larger speedup is seen the larger the dataset is. The block size of the filesystem is 1MB, so in theory speedup should have been constant from problem sizes of 16MB and up. However, we see a clear trend that speedup improves with problem size.

On Njord, speedup is substantially lower. Since we need to switch page protection bits twice, this takes longer time than the original scatter call did, meaning we are only getting a fraction of the speedup of manual optimization. Here our predictions match our measurements much more closely.

Figure 5 shows the speedup when applied to the *read* and *send* pattern, using "row-of-nodes" reading, compared to

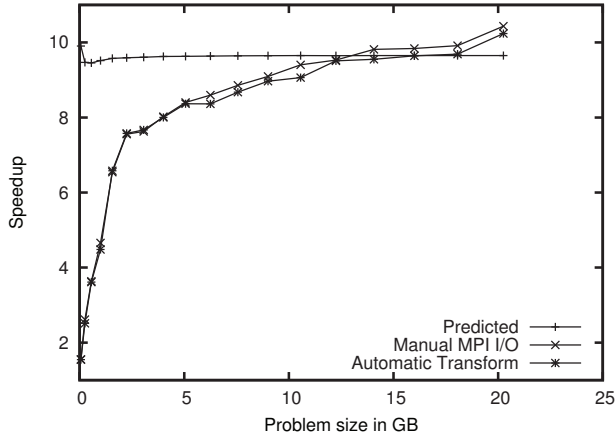


(a) Kongull

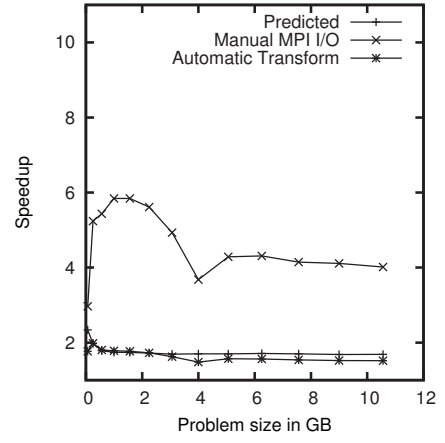


(b) Njord

Fig. 3. Base performance of I/O, communication and memory protection for Kongull and Njord.

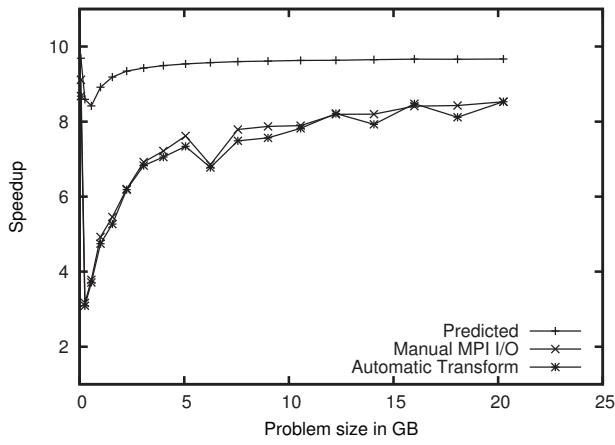


(a) Kongull

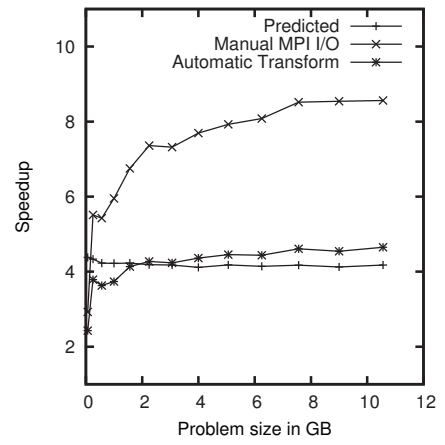


(b) Njord

Fig. 4. Speedup of read and scatter with row distribution.



(a) Kongull



(b) Njord

Fig. 5. Speedup of read and send with 2D Cartesian distribution

manual optimization using `MPI_File_read_all()` with a view on a subarray datatype.

On Kongull, automatic optimization is still almost identical to that of manual optimization. Speedup is still not as predictable as desired, and dips slightly when data distributions do not evenly match block sizes.

On Njord, our predictions match our measurements, and speedup here is considerably better than for *read and scatter*. The protected memory area is  $\frac{1}{4}$  as large (each read is 4 nodes worth of data), and since the memory area is reused without interim protection, overhead is reduced to  $\frac{1}{4}$ . However, overhead is still substantial, and we are unable to match manual optimization.

#### D. Transformed Writes

Figure 6 shows the speedup of transformed write operations from the *gather and write* pattern. Kongull does not match our predictions here, and it is clear that the overhead of parallel writes are considerably higher than parallel reads. This is to be expected, as metadata and RAID parity has to be updated, and shared blocks that don't match filesystem block boundaries have to be locked and updated sequentially. We still see that automatic optimizations are almost identical to manual optimizations.

On Njord, predictions match measurements much more clearly, and it seems GPFS and MPI I/O scales better on this machine. The automatic speedup is significantly worse than on Kongull, going as low as 1.02.

If our buffer copy optimization is enabled, the write operation is done asynchronously on a memory copy of the buffer. This effectively reduces the wallclock time of the gather and write operation to just the sending of control messages, leading to inflated speedup numbers. However, the I/O operation is performed in the background, and will increase the time for subsequent communication operations, so the effective overall speedup depends on the communication pattern used. We have therefore turned this optimization off for these measurements.

Figure 7 shows the speedup of the transformed *receive and write* operations. Kongull shows a slightly more predictable speedup than for the 1D case. For the smallest problem size, the baseline code does 4 sequential writes, which leads to waiting. The runtime transformation results in a single write on each node, with data that fit in the GPFS system cache. This leads to an inflated speedup number which is not representative for larger problems, as can be clearly seen on the graph. Once the problem size reaches a sufficient size that write caches are not a performance measurement issue, speedup is almost constant.

Again, speedup on Njord is primarily due to the reduced memory area in use, which reduces the overhead from large *mprotect()* calls.

#### E. Local to Global Transformations

Figure 8 shows the speedup when transforming *read and scatter* into either individual MPI I/O on all nodes in parallel or collective MPI I/O functions. It is clear that for certain

problem sizes on GPFS, and always on NFS, individual I/O read functions should be preferred over the collective ones.

#### F. Overcommit

As a side-effect of our page protected optimizations, it is possible to use the *read and gather* pattern (or read-whole-file and send pattern) on files that do not fit in main memory on the node performing file I/O.

For these cases, speedup comparisons are impossible, since the original application will crash due to lack of memory if attempted with the largest problem sizes. We did, however, observe wallclock time usage similar to that already seen for the compared cases.

This means our method can be used to help move applications from large shared-memory machines to commodity clusters.

#### G. Performance over NFS

Figure 9 shows the speedup for the different patterns when run on Clustis3, which has only a single I/O node and, on the compute nodes, mounts the filesystem over NFS. As can be seen, small problem sizes exhibit enough overhead that we see a small slowdown, but as the problem size grows, the elimination of the data distribution stage means we get speedup, even if it is worse than the speedup we observe on the clusters with multiple I/O nodes.

The exception to this is 2D cartesian read. In this mode, each node issues one read and one seek operation for every single row of data it needs, and the OS does readahead. Unfortunately, this operational overhead is enough to negate any speedup we could have, and our method results in a slowdown instead.

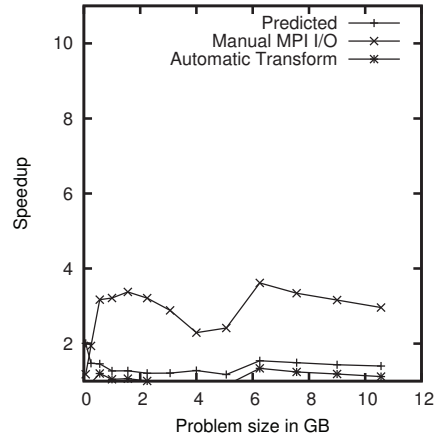
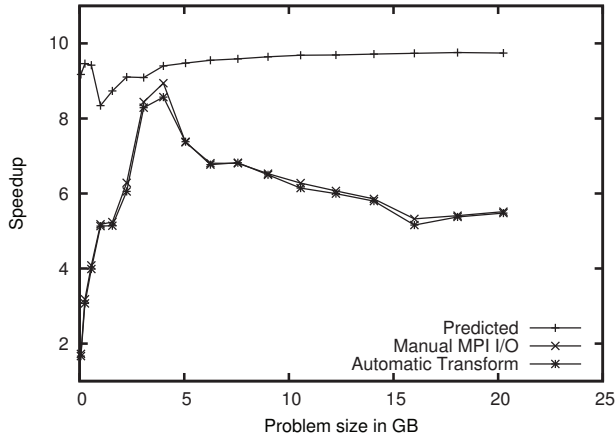
## VI. CURRENT AND FUTURE WORK

To avoid serializing requests for non-trivial datatypes that may overlap, better datatype analysis is needed. Such analysis has already been done for overlapping communication requests, and this needs to be ported to the I/O space as well.

Scatter operations currently send a hint message to each node, since this fits well with the generalization of routines done internally. However, further performance gain can be achieved by distributing the hint message as a broadcast instead of as a series of point-to-point messages.

It is possible to replace page protection with runtime code emulation. By switching the main execution thread to emulate instructions during asynchronous operations, we avoid the overhead of large page protection operations. Chances are that either the data will be used or a new optimizable operation will be performed within a few hundred instructions. An emulation scheme would have no overhead when switching back to regular execution. We can also scan past the blocking point for future operations that we can start while waiting for I/O or communication to finish.

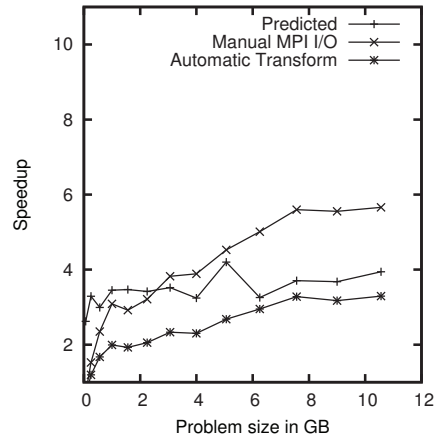
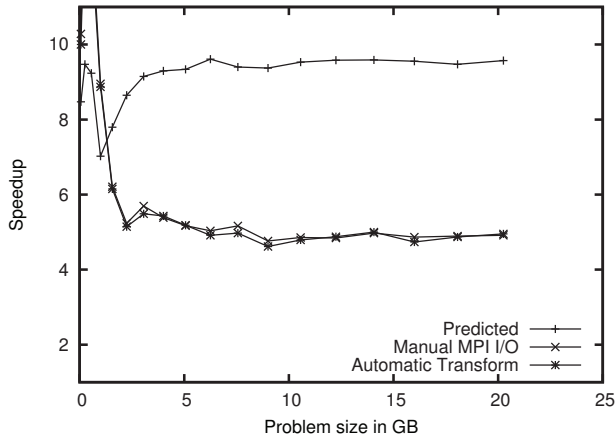
The context knowledge could be saved between application executions, so that overhead is minimized for non-optimizable calls, and mergeable operations are detected sooner.



(a) Kongull

(b) Njord

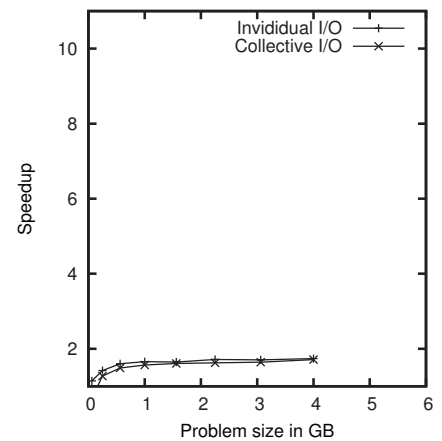
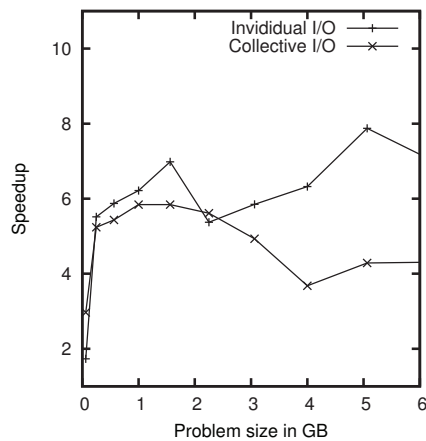
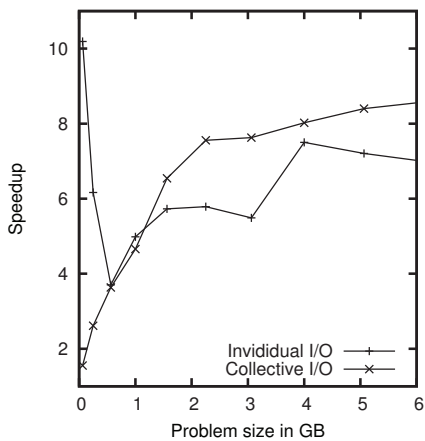
Fig. 6. Speedup of *gather and write* with row distribution



(a) Kongull

(b) Njord

Fig. 7. Speedup of *receive and write* with 2D Cartesian distribution



(a) Kongull

(b) Njord

(c) Clustis3

Fig. 8. Speedup of individual to collective transformation

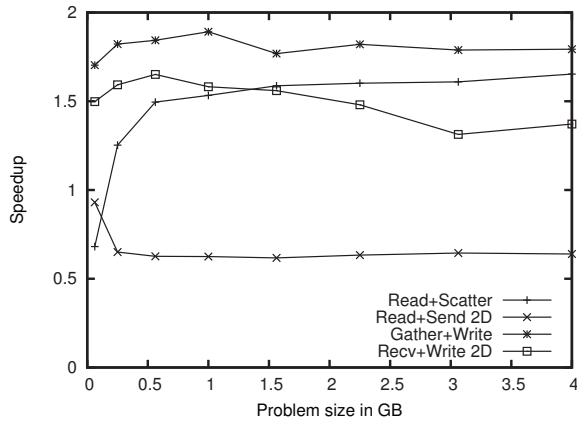


Fig. 9. Speedup on Clustis3

The method could be made aware of the number of the I/O node configuration of the cluster so that the optimal optimization method can be chosen automatically. A solution for this could be to maintain a database of previous performance numbers, and choose the best one based on previous knowledge. At random intervals, the non-optimal method is tested to see if the results of that method have improved.

For some data sizes and distributions, individual file I/O is much faster than collective I/O. If we had previously benchmarked different I/O layouts and sizes, and made the result of this available to our method, it could automatically choose the most efficient method for any given optimization case.

For small data sizes, collective I/O is slower than gather or scatter from a single node with sequential I/O. We should therefore also perform the reverse optimization; transforming collective, shared I/O to single-node I/O if we can predict it is faster.

The slowdown observed over NFS for 2D and 3D distributed data can be addressed by having one node in each row of nodes perform the I/O, then distribute the data to the rest of its row. While the speedup will not be as good as for row-distributed data, it will at least no longer give a slowdown.

We are eagerly expecting the introduction of asynchronous collective communication in MPI 3.0, which will greatly extend the operations we can optimize. We also believe we have demonstrated a need for the ability to cancel asynchronous collective MPI I/O operations, and hope to see this in a future MPI revision.

## VII. CONCLUSIONS

We have implemented an automatic run-time parallelization of sequential I/O for MPI applications. Requests are transformed from sequential and individual I/O to collective I/O operations at run-time, without the need for any knowledge about the application. Our method therefore works with any parallel program, even without access to source code. We show speedup on most common use cases, and that we can scale and distribute the I/O load on clusters with multiple I/O nodes.

These changes allow existing applications to scale better when problem size, node-count and I/O size increases. It also allows application authors to develop and test their applications on small problem sizes on smaller clusters, where I/O may not be a large concern, and see the same wallclock speedup when moving them to a large cluster.

As an added benefit, our method allows applications to read and distribute larger amounts of data than fit in main memory without swapping, and with significant speedup.

## VIII. ACKNOWLEDGMENTS

The authors would like to thank NTNU and NOTUR for access to Njord and Kongull to perform our testing on, and especially for being allowed to far exceed our filesystem quotas. We thank the Department of Computer and Information Science for allowing us to upgrade, reconfigure and allocate for exclusive use the Clustis3 cluster.

We would also like to thank Dr. John Ryan for critical and constructive feedback on this paper, and our other colleagues and students for access to their applications which we used for validation of our results.

## 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] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.
- [3] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA, USA: MIT Press, 1999.
- [4] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges, "Mpio/gpfs, an optimized implementation of mpi-io on top of gpfs," in *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM, 2001, pp. 17–17.
- [5] R. Thakur, E. Lusk, and W. Gropp, "Users guide for ROMIO: A high-performance, portable MPI-IO implementation," Mathematics and Computer Science Division, Argonne National Laboratory, Tech. Rep. ANL/MCS-TM-234, October 1997. [Online]. Available: ftp://ftp.mcs.anl.gov/pub/thakur/romio/users-guide.ps.gz
- [6] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadar, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.
- [7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, September 1996.
- [8] W.-k. Liao, A. Ching, K. Coloma, A. Nisar, A. Choudhary, J. Chen, R. Sankaran, and S. Klasky, "Using mpi file caching to improve parallel write performance for large-scale scientific applications," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–11.
- [9] 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.
- [10] T. Natvig and A. C. Elster, "Automatic and transparent optimizations of an application's MPI communication," in *PARA 2006*, ser. Lecture Notes in Computer Science, B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, Eds., vol. 4699. Springer, 2007, pp. 208–217.
- [11] T. Natvig and A. C. Elster, "Using context-sensitive transmission statistics to predict communication time," in *PARA 2008 (to appear in Lecture Notes in Computer Science 6126/6127)*. Springer, 2010.