Overlapping Data Transfer With Application Execution on Clusters

Karen L. Reid and Michael Stumm
reid@cs.toronto.edu stumm@eecg.toronto.edu

Department of Computer Science
Department of Electrical and Computer Engineering
University of Toronto

1 Introduction

When scientific applications run on clusters (or other supercomputers), they often operate on and/or produce large data sets, which are usually stored on a separate storage system. To allow efficient I/O during application execution, the data sets are transfered to and from disks local to the cluster. When an application receives its allocation of processors from the scheduler, it is typically run in 3 phases: i) an input phase, where a script copies its input files from the remote storage site to the cluster, ii) a computation phase where the actual application runs and accesses local files, and iii) an output phase where a script transfers the output files back to the remote storage (Figure 1 a).

In this mode of operation, transferring large files takes a significant amount of time, during which the CPUs are largely unutilized. For example, with a Linear Accelerator simulation (Linac) that runs at LANL [3, 4], snapshots of the data set saved for later visualization can be produced such that 7% of the elapsed time is spent on the final I/O phase. The I/O phases do not typically overlap the computation phases of other applications, primarily for accounting and billing reasons. Conventional wisdom suggests that an application's performance suffers if it must share its processors with other jobs' I/O operations.

In this paper, we show that this conventional wisdom is largely unfounded, and that if we can overlap the I/O phase of one application with the computation phase of another (Figure 1 b), there is potential for significant overall performance improvement. For example, for the Linac application, concurrent I/O does not slow down the computation phase, so another application's I/O may be completely overlapped with the computation phase of Linac, with no performance loss to Linac, and an improvement in overall throughput.

To successfully overlap the I/O phases with computation, there must be sufficient idle resources during the computation phase, and it must be possible to exploit these idle resources with minimal performance loss to the application. The data transfer program may interfere with the application in complex ways as the two compete for CPU, network, memory bandwidth, and disk bandwidth.

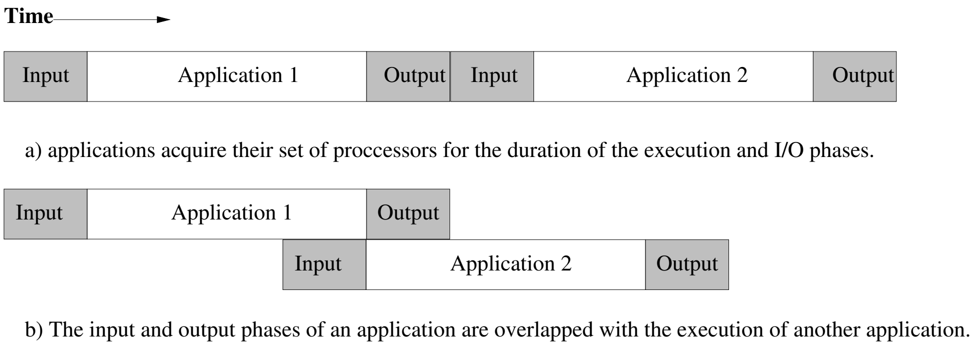

Figure 1: Scheduling applications on a cluster

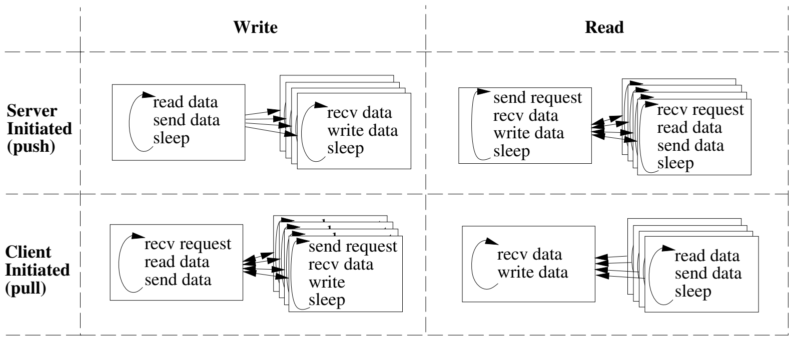

Figure 2: Models for data transfer

The architecture itself imposes upper limits on the achievable data transfer rate and therefore on the resource requirements of the data transfer program. The bandwidth out of the server determines the maximum rate at which data can be transferred between the server and the cluster. For example, suppose one server sends and receives data from 32 nodes. If a switched 100 Mbps Ethernet is used to carry the data, the maximum data transfer rate per client is only about 2.8 Mbps. If Myrinet is used, then the limiting factor is not the network, but the speed at which the server can process requests to get data from disk and send it to all the nodes. In our example of one server and 32 clients, the maximum data transfer rate per client is 5 Mbps.

Section 2 presents two versions of a data transfer program we call Datamover, which we use to analyze the effect of concurrent data transfers on application performance. Section 3 describes a synthetic benchmark application, MPbench, that we developed to investigate how resource usage characteristics of the application influence the performance impact of Datamover. Experimental results in Section 4 show how the different data transfer programs affect the performance of Linac and MPbench.

2 Datamover

Our bulk data transfer program is written as a user-level client-server application. The server runs on a machine outside the cluster, most likely the file-server or an administrative node on the cluster. One client runs on each node of the cluster.

Two versions of the Datamover program have been written. The first uses MPI and a push model (see Figure 2). The server periodically broadcasts blocks to all clients until N Mbytes are sent to each client. Each client writes the blocks it receives to local disk. The total number of bytes transfered, the interval between broadcasts, and the size of the blocks are parameters to the program. We chose to use MPI for this version because it provided a simple user-level library for communication, promising portability and optimized communication libraries.

A problem we found with the MPICH [2] implementation of the MPI standard was that the receive call busy-waits until the message is received, using far more CPU resources than necessary. To mitigate this effect, the client can estimate the time until the next receive is expected, and sleep an appropriate amount before calling receive. This solution still incurs extra CPU overhead because the estimates are not perfect. Also, if the server sends too much data before the client wakes up, there is a danger that the client's network buffers overflow. A blocking receive call in MPICH would reduce this type of overhead.

If each node receives the same data set from the server, it is possible to use the broadcast call in MPICH (`MPI_Bcast`). `MPI_Bcast` is implemented using a tree algorithm, which results in a faster completion time for the broadcast. However, this means that some client nodes require more CPU cycles to transmit the messages to other clients. To minimize the interference on the client side, Datamover has the server send every block to each client separately using `MPI_Send`. This method has a longer total completion time than when using `MPI_Bcast`, but since we expect the computation phase to dominate I/O time, the time spent on I/O is not a critical factor. (Section 4 shows how the use of `MPI_Bcast` compares to `MPI_Send`.)

Because the many layers of the MPI library made it difficult to determine exactly how the data transfer program was using resources and interfering with the application, we wrote another version using TCP sockets. The TCP version required substantially less CPU time than the MPI version, the primary reason being that the TCP Datamover receives

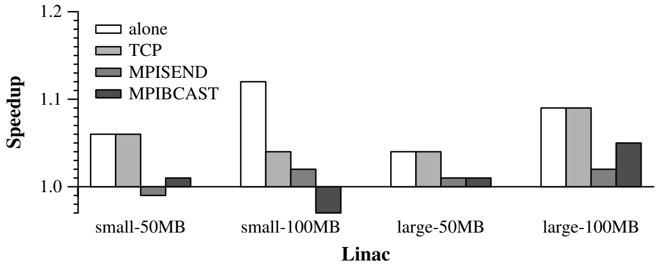

Figure 3: Speedup obtained when Datamover is run concurrently with Linac.

are implemented using `select` and rely on interrupts rather than polling to notify the program when a packet has arrived. The TCP Datamover uses the pull model. While this approach requires more messages, it gives the clients the ability to control the rate of data transfer. In practice, the extra messages do not appear to affect the overall performance.

3 MPbench

The characteristics of the application probably affect the extent to which Datamover interferes with the application. To study this, we constructed a parameterized synthetic benchmark application that allowed us to vary the application's resource requirements.

MPbench is an MPI program that iterates over several large arrays. Several parameters control its behaviour: memory size, cache hit rate, message broadcast frequency and size, file read frequency and size, and file write frequency and size. For our tests, we select a memory size that is small enough to avoid swapping. To modify the cache hit rate, MPbench iterates a variable number of times over a block of data that is smaller than the cache. We vary the number of iterations between reads, writes or broadcasts to change their frequency.

4 Experiments

The experiments were conducted on a 64-node Linux cluster connected by 100Mbps switched Ethernet and Myrinet. Each node contains two Pentium II 333MHz processors with 512MB of memory. The nodes run the Linux kernel version 2.2.12. One node of the cluster is used as the Datamover server.

The experiments presented here all have the same structure. We measure the execution time of the computation phase of an application (either Linac or MPbench) first when it is run on an otherwise idle set of 16 nodes (T_{comp}). Then we measure the time needed to transfer the required data to each of the 16 nodes (T_{io}). This gives us the time for the I/O phase of the application. Finally, we measure the execution time of the computation phase of the application while (one of the versions of) the Datamover is run concurrently (T_{conc}). Both the application and Datamover use Ethernet for communication. In tests where the application used Myrinet, the results were similar.

4.1 Linac results

We compare the computation time of the Linear accelerator code when run by itself (T_{comp}) to the execution time when it is run concurrently with each of the Datamover programs (T_{conc}). The runs shown in Figure 3 illustrate the effect each version of the Datamover has on the concurrently running computation. All of the times are normalized to T_{comp}. The graph shows results using two different sets of parameters for Linac: the "small" parameter set took approximately 9 minutes to execute by itself on 32 processors (16 nodes), and the "large" parameter set took approximately 12 minutes on 32 processors. For each parameter set we look at the results when Datamover transfers two different amounts of data: 50 and 100 Mbytes per node. The white bars represent the optimal performance improvement ($\frac{T_{comp}+T_{io}}{T_{comp}}$) when there is no interference from concurrently running I/O. The remaining bars give the performance improvement/loss when Linac is run with the various Datamover implementations ($\frac{T_{comp}+T_{io}}{T_{conc}}$).

The results show that the TCP Datamover is quite successful at overlapping I/O with the computation of the application. Only in the second case, where the time to transfer data is a larger portion of the total time, does it fail to

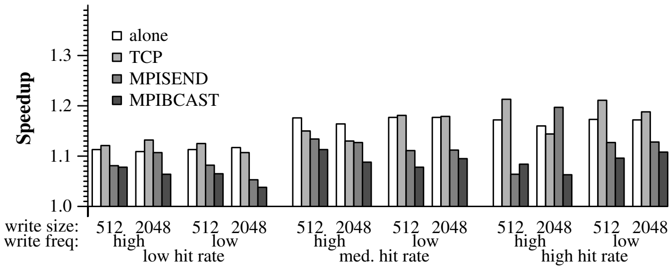

Figure 4: Speedup obtained when Datamover is run concurrently with MPbench-write.

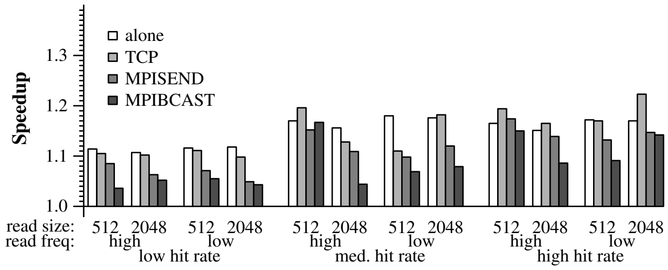

Figure 5: Speedup obtained when Datamover is run concurrently with MPbench-read.

achieve the maximum possible speedup. On the other hand, the overhead of the MPI Datamover is such that there is little benefit to running it concurrently with Linac. In two of the cases, it takes longer to run Linac concurrently with MPI Datamover than it does to run the two separately. The difference between MPISEND and MPIBCAST in the far right case is interesting. Although MPIBCAST has higher CPU requirements, the distribution of network traffic and the shorter completion time for Datamover leads to better performance than with MPISEND.

We conclude that there are idle resources available to be exploited by Datamover. However, the lower CPU requirements of the TCP Datamover were necessary to achieve the maximum performance benefits.

4.2 MPbench results

Using our synthetic benchmark, we investigate the effect of application characteristics on the overhead experienced when MPbench is run concurrently with the Datamover. Figures 4, 5, and 6 show the results of running MPbench using a variety of parameter settings. In the examples shown in Figure 4, each process of MPbench periodically writes to a local file. In Figure 5, each process periodically reads from a local file, and in Figure 6, MPbench periodically broadcasts messages to all nodes.

The cases with low cache hit rates yield performance similar to what we saw with Linac. TCP Datamover caused almost no interference. We also see performance gains from overlapping the MPI Datamover with MPbench. As expected, MPIBCAST generally does not do as well as MPISEND. We see little performance difference when varying the size and frequency of file activity in the low cache hit rate cases. When MPbench is broadcasting messages (Figure 6), there is a larger performance improvement when the size and frequency of the broadcast increases. This is not surprising, since nodes will inevitably need to wait for each other while synchronizing, leaving more opportunity for Datamover to use the idle CPU.

As the cache hit rate increases, a strange phenomenon arises. In several cases, the time to run MPbench by itself is greater than the time to run MPbench concurrently with the Datamovers, and hence the performance improvement seems to exceed the optimal. In other words, T_{comp} is greater than T_{conc}. We are currently investigating the reasons for this.

The results of Figures 4–6 show that the characteristics of the application do not appear to play much of a role in the amount of speed up that is obtained. This indicates that CPU time is the biggest factor in the overhead to the application, and is why TCP Datamover always wins over MPI Datamover.

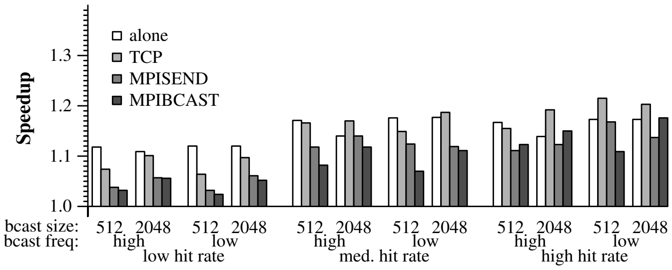

Figure 6: Speedup obtained when Datamover is run concurrently with MPbench-bcast.

5 Conclusions

Our results indicate that it is beneficial to overlap data transfer with application execution. We consistently get a speedup by running Datamover concurrently with the application. Often, there is virtually no overhead to the application.

We are continuing to investigate how applications use the node resources. If we can identify phases of an application that do not fully utilize the node, we can decide at runtime when it is best to transfer data concurrent to application execution.

There are also interesting questions about whether we can improve the per node scheduling behaviour. For example, lowering the priority of Datamover did not help the performance, because the priority of our application was also lowered as it acquired more CPU time. We need to reexamine the scheduling algorithms in the context of having one long-running high-priority job, and other low-priority system-level tasks.

The next step would be to integrate Datamover into a batch scheduler such as PBS [5]. This will require changes to the scheduling algorithms, since we will need to predict both the set of processors, and the start time of a job far enough in advance so that the jobs input data set may be loaded.

The InterMezzo [1] file system allows the automatic migration of user files to caches on the local disks, thereby relieving the user of the need to explicitly transfer files. By integrating the ideas from Datamover with prefetching hints from the scheduler, we could use the file system to ensure that the data files have been moved to the local disks when the application is ready to start.

Bulk data transfers also occur inside the cluster. For example, when a checkpointed job is restarted it may receive a different set of processors than when it was previously run. The checkpoint files need to be transferred from the previous set of nodes. A mechanism similar to Datamover can be used, but determining the optimal rate of transfer, and avoiding load imbalances will be important.

References

[1] Peter Braam, Michael J.Callahan, and Philip I. Schwan. The InterMezzo filesystem. In *O'Reilly Perl Conference 3.0*. O'Reilly, 1999.

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[3] William Humphrey and Susan Coghlan. Using a Linux cluster for linear accelerator modeling. In *Extreme Linux Workshop*. USENIX, 1999.

[4] William Humphrey, Robert Ryne, Timothy Cleland, Julian Cummings, Salman Habib, Graham Mark, and Ji Qiang. Particle beam dynamics simulations using the POOMA framework. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[5] The Portable Batch System. http://pbs.mrj.com/.