Enhancing Data Migration Performance via Parallel Data Compression

Jonghyun Lee, Marianne Winslett, Xiaosong Ma, Shengke Yu
Department of Computer Science, University of Illinois, Urbana, IL 61801 USA
{jlee17, winslett, xma1, syu3}@uiuc.edu

Abstract

Scientific simulations often produce large volumes of output that are moved to another platform for visualization or storage. This long-distance migration is slow due to the data size and slow network. Compression can improve migration performance by reducing the data size, but compression is computation-intensive and so can raise costs. In this work, we show how to reduce data migration cost by incorporating compression into migration. We analyze eight scientific data sets, and propose three approaches for parallel compression of scientific data. Our results show that with reasonably fast processors and typical parallel configurations, the compression cost for large scientific data is outweighed by the performance gain obtained by migrating less data. We found that a client-side compression approach (CC) can improve I/O and migration performance by an order of magnitude. In our experiments, CC always matches or outperforms migration without compression when we overlap migration with computation, even for not very compressible dense floating point data. We also present a variant of CC that is well suited for use with implementations of two-phase I/O.

1 Introduction

Large-scale scientific simulation codes produce copious amounts of output, which is placed on secondary storage for fault-tolerance purposes or future time-based analysis. This analysis is usually conducted on a workstation that is geographically separated from the machine where the simulation ran, such as a scientist’s local workstation. To reduce application turnaround time including data migration, one can overlap computation and migration [5, 10], but with typical network bandwidths of less than 1 MB/s from supercomputers to the outside world, often migration is still the longest part of a simulation run. Compression can help by reducing data size, but is very computation-intensive for the relatively incompressible dense floating point data that predominate in simulation codes. Our goal is to incorporate rate compression into migration in a way that reliably reduces application turnaround time on today’s popular parallel platforms.

The idea of using data compression in scientific computing is not new. Generic, data-specific, and parallel compression algorithms can improve file I/O performance and apparent Internet bandwidth [1, 3, 7, 8, 11, 13, 16, 17, 19]. However, this paper is the first to address the issues that arise when integrating compression with long-distance transport of data from today’s parallel simulations. These issues include: what kind of compression ratios can we expect? Will they fluctuate over the lifetime of the application? If so, how should we make the decision whether to compress? Will the compression ratios differ with the degree of parallelism? If so, how can we handle the resulting load imbalance during migration? Are special compression algorithms needed? Can we exploit the time-series nature of simulation snapshots and checkpoints, to further improve compression performance? What kind of performance can we expect on today’s supercomputers and internet? In this paper, we answer as many of these questions as space permits.

Multidimensional arrays are the most common output from scientific simulations. Simulation codes may assign separate arrays to each processor or may divide large arrays into disjoint subarrays (chunks), each of which is assigned to a processor. Many visualization tools can only read array data in traditional row-major or column-major order, while the simulation uses a different distribution. In this case, reorganization of array data between its memory and disk distributions is required (Figure 1).

Simulations typically perform output operations at regular intervals. The two most important output operations are snapshots and checkpoints. A snapshot stores the current “image” of simulation data, for future visualization or analysis. A checkpoint saves enough simulation data for computation to restart from the most recent checkpoint if the system crashes. These I/O operations bracket each computation phase. Similarly, an I/O phase is defined as the period between two consecutive computation phases.

The processors in a parallel run can be divided into
two possibly overlapping groups, compute processors (also known as (I/O) clients) and I/O processors (I/O servers). Compute processors perform the simulation code’s computation, and I/O processors do the file I/O. Dedicated I/O processors are only used for I/O, so are usually idle when compute processors are busy. Non-dedicated I/O processors act as compute processors during computation phases, and as I/O processors at I/O time. Often, the I/O operations of a simulation are collective, where all processors cooperate to carry out I/O tasks.

As proposed in [10], we use I/O processors to stage the output to the local file system, then migrate it. This local staging prevents compute processors from stalling while data is migrated. We can also migrate output immediately without local staging, but this does not allow overlap between data transfer and other application activities, and performs poorly in current typical hardware configurations, so we do not consider it further. Figure 2 shows the data flow in a simulation run with I/O and migration, along with three possible spots for performing compression.

Client-side Compression during an I/O Phase (CC). Under CC, each client compresses each of its output chunks and sends them to a server, along with metadata such as the compressed chunk size and the compression method. I/O servers receive compressed chunks from clients and stage them to disk. CC’s advantage is its high degree of parallelism in compression. However, CC’s compression cost is fully visible to clients. Further, if the array distributions in memory and on disk are different, servers must decompress the chunks, reorganize the data, and recompress the new chunks. Therefore, we assume the same array distribution in memory and on disk for CC, with clients’ array chunks assigned to servers in a round-robin manner. Codes with independent arrays on each processor, such as codes for irregular problems, fit this model.

Server-side Compression during an I/O Phase (SC). In SC, servers receive output data from clients during an I/O phase, compress them, and stage them to disk. SC allows the array to be reorganized to any target distribution before compression, and thus is more flexible than CC. However, some or all of the compression cost will not be hidden with SC, if the servers start to stage the data before all of it has arrived from the clients, and force clients to wait during compression. Further, scientists usually use far fewer dedicated I/O processors than compute processors, so aggregate compression performance with SC will be worse than CC. To keep the flexibility of SC, but also exploit as many processors as possible like CC, we can choose to use all the processors as non-dedicated I/O processors, and perform SC using them, as discussed further in section 3.

Server-side Compression on Already-Staged Outputs (SC2). Before being transferred to a remote machine, a staged output needs to be read into memory from the local file system. SC2 reads and compresses the staged output, and then migrates it. This overlaps the compression with computation, so the visible I/O cost may be shorter than CC and SC. However, concurrent compression and computation will slow down the simulation so much that SC2 is only suitable for dedicated I/O processors, thus limiting the degree of parallelism during compression. Further, more time will be spent in file I/O, due to the uncompressed data.

2 Experimental Results

Data Sets. Floating point arrays are the most common data type for scientific simulations. Typical floating point arrays are dense, i.e. most of the array elements contain important numbers, but also can be sparse, and therefore highly compressible. Sparse output arrays are not unusual near the beginning and end of a simulation run. Integer arrays are also widely used: for example, floating point data often have an accompanying integer array describing the coordinate system. Simulations typically include text annotations and textual formatting information in their HDF (Hierarchical Data Format) [7] output. To reflect this wide spectrum of data, we used the eight data sets in Table 1. Astrophysics, Cactus, ZEUS-MP, and Flash are simulation results from four different astrophysics codes. Gen1 is the output of a rocket simulation code. SCAR-B and AVHRR are direct observation data from an airborne scanning spectrometer and a satellite. Bible contains the whole Bible. The compression ratios in Table 1 were calculated as the compressed size using UNIX gzip divided by the uncompressed size.
pressed size. The compression will be discussed in the next section.

**Experimental Setup.** Our experiments migrate data from Blue Horizon, an IBM SP2 at San Diego Supercomputing Center (SDSC), to a workstation at the University of Illinois. The SDSC SP comprises 144 SMP nodes, each equipped with 8 375 MHz Power3 CPUs, 4 GB of memory and 8 GB of scratch space on local disk, running AIX 4.3.3. The UIUC workstation is a Sun SPARCstation 10 running Solaris 2.7.

We built microbenchmarks that measure the I/O and network performance on the SDSC SP (Table 2). The file system and HDF write performance were obtained by writing out 128 MB of data with sequential 1 MB write requests. We used the scratch space local to each processor and 1 processor per node for the write, so the write performance scales perfectly as the number of processors increases. Note that for data staging, there is no motivation to use a shared file system if faster local disks are available. The network bandwidth was measured by using TCP/IP to send a series of messages from concurrent senders to a single recipient. For consistency of results, we conducted the experiments late at night and averaged the numbers over 5 or more runs. The numbers in parentheses in Table 2 are the 95% confidence interval for the averages.

In Table 2, aggregate network bandwidth increases with the number of concurrent senders, but not linearly. The low bandwidth, though measured at one moment in time, is typical. We never obtained more than 1 MB/s from Blue Horizon to the National Center for Supercomputing Applications (NCSA) or UIUC. Our previous experiments [10] obtained a transfer rate of up to 1 MB/s using multiple concurrent streams in a regular internetwork environment between UIUC and Argonne National Laboratory (ANL).\(^1\)

We believe that the compression routine used for migration should be a tunable parameter, in case an appropriate application- or data-specific compression routine is available [1, 3, 17, 19]. Previous work in the HDF group [4] and our own group did not identify any home-grown or freely available compression routine that consistently produced superior compression ratios or faster compression rates on typical simulation data. For that reason, we use UNIX `gzip` 1.2.4 modified to perform memory-to-memory compression, instead of disk-to-disk compression.

The current experiments all focus on compression, rather than decompression. Because scientific data sets are so large, scientists often store the data compressed, and decompress only at read time. Decompression is quick; for example, the decompression rate with unaltered `gzip` and an unflushed file cache on the workstation described earlier ranges from 2.22 MB/s for Gen1 to 3.72 MB/s for ZEUS-MP, as compared to data copy rates with Unix `cp` on the same workstation of 4.38 MB/s. With performance this good for decompression, our experiments can focus on the compression costs.

**Data Set Analysis.** Because compression is highly data dependent, we performed parallel compression experiments on each data set. We used from 1 to 64 processors, in powers of 2, and distributed the arrays evenly over the processors, using an HPF-style distribution whose first directive is `BLOCK` and the others are all `*`. An exception was Gen1, which consists of many small independent arrays, and it was distributed by assigning each block to a compute processor in a round-robin manner.

Figure 3 shows the results. The X-axis of the graph represents the number of processors used, and the Y-axis represents the aggregate compression rate, calculated as the size of the uncompressed data divided by the elapsed time to compress it. This graph shows two important compression characteristics of each data set. First, there is a wide range in compression ratios across the data sets (0.055 - 0.84), and a wide range of single-processor compression rates, from 1.40 MB/s to 8.00 MB/s. The highly compressible arrays had the best aggregate compression rates (ZEUS-MP, Flash, AVHRR). The text data had one of the lowest compression rates, although it can be compressed to 32% of the original size. Experiments with postscript files, not shown in Figure 3, also showed a low compression rate.

Second, Figure 3 shows that except for the highly com-

---

\(^1\)We also migrated data on a special backbone network connecting ANL and NCSA, and obtained up to 13 MB/s. Space constraints prohibit further discussion of this environment.

**Table 1.** The description of the array data sets used in the experiments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Rank</th>
<th>Shape</th>
<th>Element Size</th>
<th>Total Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astrophysics</td>
<td>3</td>
<td>704 × 704 × 9</td>
<td>32-bit float</td>
<td>17.01 MB</td>
<td>14.26 MB</td>
<td>0.84</td>
</tr>
<tr>
<td>Gen1</td>
<td>3</td>
<td>64 blocks (645 × 23 × 17, 748 × 26 × 20 arrays / block)</td>
<td>64-bit float</td>
<td>133.88 MB</td>
<td>90.37 MB</td>
<td>0.66</td>
</tr>
<tr>
<td>Cactus</td>
<td>3</td>
<td>192 × 192 × 192</td>
<td>64-bit float</td>
<td>54 MB</td>
<td>34.69 MB</td>
<td>0.64</td>
</tr>
<tr>
<td>ZEUS-MP</td>
<td>3</td>
<td>256 × 256 × 256</td>
<td>32-bit float</td>
<td>64 MB</td>
<td>3.55 MB</td>
<td>0.055</td>
</tr>
<tr>
<td>Flash</td>
<td>4</td>
<td>161,793 × 9 × 9 × 9</td>
<td>64-bit float</td>
<td>91.53 MB</td>
<td>7.36 MB</td>
<td>0.083</td>
</tr>
<tr>
<td>SCAR-B</td>
<td>3</td>
<td>1525 × 50 × 716</td>
<td>16-bit integer</td>
<td>104.13 MB</td>
<td>62.69 MB</td>
<td>0.60</td>
</tr>
<tr>
<td>AVHRR</td>
<td>2</td>
<td>5004 × 2168</td>
<td>16-bit integer</td>
<td>20.69 MB</td>
<td>3.57 MB</td>
<td>0.17</td>
</tr>
<tr>
<td>Bible</td>
<td>1</td>
<td>4370977</td>
<td>8-bit character</td>
<td>4.17 MB</td>
<td>1.32 MB</td>
<td>0.32</td>
</tr>
</tbody>
</table>
Table 2. The performance measured on the SDSC SP by the microbenchmarks.

<table>
<thead>
<tr>
<th></th>
<th>1 processors</th>
<th>2 processors</th>
<th>4 processors</th>
<th>8 processors</th>
<th>16 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>file system write throughput</td>
<td>14.05(±0.099) MB/s per processor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HDF write throughput</td>
<td>6.24(±0.54) MB/s per processor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aggregate network</td>
<td>(0.10 ± 0.001) MB/s</td>
<td>(0.20 ± 0.003) MB/s</td>
<td>(0.33 ± 0.011) MB/s</td>
<td>(0.40 ± 0.016) MB/s</td>
<td>(0.46 ± 0.021) MB/s</td>
</tr>
<tr>
<td>bandwidth to the workstation</td>
<td>MB/s</td>
<td>MB/s</td>
<td>MB/s</td>
<td>MB/s</td>
<td>MB/s</td>
</tr>
</tbody>
</table>

Pressible data sets, compression rates scale up linearly as the number of processors increases. The most compressible data sets show lower scalability because different chunks have different compression ratios and therefore different compression rates. For example, the compression ratios of ZEUS-MP chunk data ranged from 0.001 to 0.394 with 64 processors. For the less compressible arrays, the compression ratio was almost uniform across all processors. Since the overall performance of parallel compression is determined by the last processor to finish, these results raise potential load imbalance issues for highly compressible data sets, as discussed later.

We also measured the total size of compressed chunks when multiple processors are used, and compared it to the compressed size when one processor is used. The difference is less than 1%, except for the Bible, where the total size of 64 compressed chunks is about 5% larger than the size of the compressed array using one processor. However, the postscript data set mentioned earlier did not show this problem.

To see the correlation between compression ratio and compression rate, Figure 4 plots the compression rate and ratio of each chunk used in Figure 3. The plots in the graph form different clusters with different types of data. Within a cluster, the more compressible a chunk is, the higher the compression rate is [16]. However, the more compressible data sets do not always have higher compression rates. Especially, Gen1 and Bible are far from this trend. Therefore, it is not safe to directly relate compression rate and compression ratio across data sets.

Comparison. We implemented each compression approach as an extension to Panda [15], a high-performance portable parallel I/O library developed at UIUC. Panda’s API allows users to read and write subarrays of multidimensional arrays on distributed and distributed-shared memory machines. Panda supports both regular and irregular distributions of arrays as well as independent arrays, and it performs automatic reorganization of array data between the two distributions in memory and on disk during I/O. It also supports both dedicated and non-dedicated I/O servers. In the experiments, Panda writes out data in HDF 4.1.

We measured the I/O and migration performance (i.e., a single output period) of the extended Panda library for each approach and without compression at all (NC). We used 30 compute processors and 2 dedicated I/O processors, and distributed the arrays over the compute processors using a (BLOCK, ..., BLOCK) distribution and a mesh appropriate for the array rank. For the approaches that allow data reorganization, we reorganized the arrays to traditional row-major order on disk. For Gen1, each I/O processor took care of half the data blocks. For CC, chunks were assigned to I/O servers in a round-robin manner. For migration, we used eight data streams (allowing multithreaded streams from each I/O processor), for better migration per-
formance. Figure 5 shows the results averaged over 5 or more runs, with error bars representing 95% confidence intervals. In CC and SC, data compression is included in the I/O cost, and in SC2, compression cost is explicitly shown. The HDF version of the output is slightly larger than the original size as it contains metadata and is structured differently, so the migration of HDF output took longer than the migration of binary data.

Considering the total cost of I/O and migration, migration with compression always outperforms NC, even for a dense floating point array which only compresses to 84% of the original size (Astrophysics). With highly compressible data, the benefit is tremendous, showing up to 11 times better performance than NC. Compression can increase I/O cost for CC and SC, but the migration performance improves from reduced data transfer time.

The migration cost may or may not be visible to the application, depending on the length of its computation periods, which is a tunable parameter. However, the application will always see the amount of time spent in a collective I/O call. If we compare apparent I/O costs only, CC always outperforms SC, as it uses 15 times more processors for compression in our experimental setting. The same was true when we experimented with a 1:4 ratio of servers to clients. Since the total size of compressed data will be similar no matter which compression method is used, CC performs better than the other two approaches.

For highly compressible arrays, the compressed data size at each I/O processor can be quite different, causing a load imbalance problem in migration. With CC, this problem seems less serious than the other approaches, as it compresses smaller chunks than SC and SC2, and distributes them to I/O processors in a round-robin manner, evening out the load. Also, for four data sets (Cactus, ZEUS-MP, Flash, and SCAR-B), CC’s I/O cost was actually less than NC/SC2’s I/O cost, because compression was fast, the compressed data were very small, and HDF write performance was relatively slow. In this case, no matter how fast the network is, NC cannot outperform CC for a single migration, as long as the decompression at the other side does not become a bottleneck. This will be discussed further below.

Although SC performs worse than CC during I/O, when compared to the long migration after the I/O, the difference between the two approaches seems relatively small in many cases. This makes SC attractive, because it is more flexible than CC. However, we also performed experiments not shown here on a much slower SP2 at ANL, with a faster path to the local workstation. From this work, we know that with compression on slower processors, or a network bandwidth much faster than in Table 2, the performance gap between CC and SC will increase significantly. Moreover, CC was the only approach that outperformed NC even with dense floating point data in that experiment. The same arguments can be applied to SC2. The only advantage of SC2 over CC is the reduced apparent I/O cost, but in our experiments, the I/O cost of CC is close to that of SC2, or even better thanks to fast compression and relatively slow HDF writes.

Figure 5 considered the cost of a single output phase. When data migration overlaps computation, all or part of migration cost may be hidden. Then CC or SC may become less attractive than NC, because migration costs may all be hidden, while increased I/O phase costs are still visible. We performed experiments to analyze this effect using three data sets whose time series snapshots were available to us: Cactus, Gen1, and ZEUS-MP, using the same machine configuration as in Figure 5, and a sequence of floating point computations. The length of a computation phase is an important factor which can affect the overall turnaround time, because its length determines how much of the migration cost can be hidden. Because the frequency of output is a tunable parameter in simulation codes, we varied the length of the computation phases. Figure 6 shows the application turnaround time, measured from the beginning of a run to the end of the migration of the last output, of each approach with 10 computation and I/O phases. The data in Figure 6 are evolving as simulation time goes on, therefore the compression rate/ratio is different for each snapshot.

In Figure 6, each performance curve can be divided into two parts by the “transition point” where the slope of the curve changes. This is where a computation phase becomes longer than a migration, so that a migration can be completely hidden by the overlap. Our results show that CC and SC2 always beat NC in all three data sets. Before the
transition point of NC, the performance gap between NC and the other two is significant, but not after the transition point, where a computation becomes longer than a migration. This is because CC’s I/O cost including compression is at most slightly larger than NC’s I/O cost, and in SC2, the cost for both compression and migration of compressed data is always shorter than the migration cost of uncompressed data. However, with slower machines relative to disk speed and/or a faster network relative to processor speed, CC’s I/O cost can become much larger than NC’s I/O cost, and SC2’s compression and migration can become longer than NC’s migration. In that case, NC will eventually outperform the other approaches for long computation phases. When comparing CC and SC2, CC always performed better than SC2.

We discussed earlier that if the I/O cost of CC or SC including data compression becomes shorter than the I/O cost of NC, NC cannot beat CC in a single output migration. The same argument holds when we overlap computation with migration, or even if the I/O cost of CC or SC is close enough to that of NC. For example, in Figure 5, the I/O cost of CC for Gen1 is just 0.22 seconds longer than NC’s I/O cost. For NC to perform better than CC in a single migration for Gen1, we need a network bandwidth of at least 206.91 MB/s, which cannot be reached in today’s slow internet. Even for SC, whose I/O time is 38.4 seconds longer than NC’s, a bandwidth of 1.19 MB/s is needed, which is often hard to achieve in our experience. Therefore, it is crucial to decrease the visible I/O cost as much as possible to achieve improved migration performance regardless of the network bandwidth available to us, making CC even more attractive.

From these observations and analysis, we conclude that CC gives the best performance overall, and it will be likely to dominate the NC approach either in a single migration or a real simulation run in a current typical parallel environment, with reasonable processing power. CC has no significant drawbacks when we do not have to reorganize the data, as in Gen1. When we really need reorganization and have dedicated I/O processors available, SC2 is best. Although SC2 performs worse than SC for shorter computation periods, it will eventually outperform SC with longer computation phases. As discussed in section 3, a variant of SC will be the winning approach under certain other conditions.

3 Discussion

Application to Other Parallel I/O Systems. Perhaps the most popular parallel I/O library in use today is the ROMIO [18] implementation of two-phase I/O (TPIO). With TPIO, all processors who will make I/O calls cooperate to reorganize the output data in their memories, and then once reorganization is complete, each processor writes a contiguous portion of an output file. TPIO’s separation of
reorganization and file I/O distinguishes it from the server-directed I/O used in Panda [15].

Because TPIO’s “I/O servers” are not dedicated to I/O, we can use a large number of servers to reorganize the data and then immediately compress it and write it out, in the manner of SC. However, our previous work showed that it is not a good idea for non-dedicated I/O servers to migrate staged data [10]. While the migration code itself is minimal, file system read operations on staged data can slow down the application’s computation by up to 27% on the SP2. Further, compression performance will benefit from having many servers, but migration performance will drop if the number of senders is too high. For these reasons, TPIO implementations that add migration support should write staged data to a shared file system, and use one or two additional dedicated processors to migrate the data. The same approach should be taken if SC or SC2 is used with server-directed I/O and a large number of non-dedicated I/O processors.

**Predicting Compression Performance.** In the experiments reported in this paper, the addition of compression never increases application turnaround time. However, for example, if we speed up the SDSC SP’s file system to 30 MB/s (aggregate), speed the internet bandwidth up to 1 MB/s, and increase the number of iterations in a simulation run to 50, NC would be faster than CC in Figure 6. If we know the compression characteristics of the output beforehand, we can decide in advance whether to perform compression for data migration. However, when we run a simulation, we often do not know the nature of output that the simulation will generate, and since simulation data is evolving, the compression characteristics of the data are also changing. Thus how can we predict the compression performance of a data set without compressing it first, and how fast can we do that while retaining some level of accuracy?

The most obvious solution for this problem is sampling, only taking a small portion of the data and compressing it for prediction. If we know that the data are uniform, i.e., each chunk shows almost the same data compression characteristics, this task can be done efficiently by taking one reasonably sized chunk and compressing it. However, this is not always true for scientific data, as our experiments revealed. Some chunks may be highly compressible, while the others are moderately or not very compressible. Further, compression of a single chunk cannot be parallelized.

A better alternative is for processors to compress their first (large) buffer of output data, examine the compression ratio, and decide whether to output/send the compressed data or the uncompressed version. Under this approach, the processors can communicate to arrive at a global decision as to whether to compress, or they can make the decision separately for each chunk.

Since the data are evolving over time, another possibility is to base the compression decision on the compression ratio from the previous output phase. For example, we can compress the data from the first few output phases, and based on this result, decide whether to compress the next few output phases. Periodically we perform compression again, and correct the predicted values. We designed a performance-model-based approach to decide whether or not to compress, but it is not included here due to the space limit.

**Delta Compression.** In general, simulations begin their runs with a given initial state, and the simulation data evolve from there as the simulation time goes on. Therefore, when we take intermediate snapshots periodically, it is likely that the difference between two consecutive arrays, or the “temporal difference”, will contain only relatively small numbers. This delta array might be more compressible than the original data, leading us to the idea that migrating compressed delta arrays might be faster than migrating the whole (either compressed or uncompressed) output. Similar techniques have been applied to enhance HTTP and video compression performance [11, 12], but our problem deals with much larger data sets with different characteristics.

Unfortunately, the use of delta arrays faces several challenges. First, since the output data can be as large as the size of main memory, and the delta has to be calculated while the simulation is still running, it is not obvious to calculate deltas efficiently in the general case. Efficiently calculating the original output from deltas and a previous output is another challenge. However, the most serious challenge is whether deltas are actually more compressible than the original output. We experimented with the three data sets for which we had time series snapshots (Gen1, Cactus, and ZEUS-MP). Of these three, only ZEUS-MP had smaller compressed delta arrays than regular compressed arrays. Since ZEUS-MP is already highly compressible, the overhead of creating and managing the deltas is unlikely to be offset by reduced migration costs.

However, the poor compression ratios we obtained for delta arrays may be overcome by the use of a compression algorithm specifically crafted for use with delta arrays containing small values. Engelson et al. [3] experimented with a variety of such approaches, including wavelet-based and home-grown methods. The compression ratios they obtained are better than those we obtained for dense data sets, but their best compression ratios were obtained by looking at the differences between up to ten consecutive snapshots simultaneously. It is not clear how to translate their approaches into on-line algorithms for use during a simulation run, and the compression rates of their algorithms are unknown. Our future work will address these issues.
4 Related Work

There are numerous data-specific and parallel compression techniques for scientific data and file I/O [1, 3, 7, 8, 14, 16, 19]. Among them, No et al. [14] present a runtime parallel I/O system for irregular applications based on collective I/O techniques and optimized by chunking and on-line compression mechanisms. HDF [7] supports various compression methods at the array and chunk level for its data sets. Compression has also been introduced for efficient web transmission [9, 11, 13], mainly focusing on HTTP. Today, the potential performance gain from compressed migration is much higher than that for compressed file I/O.

The FTP specification [6] defines a “compressed mode” of data transfer as well as stream and block mode, although it is not supported by most implementations, with the notable exception of BBFTP [2], which also supports parallel client streams, a special case of migration where migration cost is not hidden at all. Our compression approaches can be added to BBFTP to support parallel servers.

5 Conclusions

In this paper, we examined how data migration performance can be improved by using compression to reduce the amount of data to be migrated. We proposed three different compression approaches using parallel processors: CC performed compression at the I/O clients, while SC and SC2 compressed data on the server side before or after, respectively, they staged the data to disk for migration. Each approach was implemented using the Panda parallel I/O library and HDF. We used eight different kinds of real-world data sets with different data types (floating point, integer, text), different sizes (17 to 134 MB), and different compression characteristics (5.5% to 84% of the original size). Our results show that, first, with reasonably fast processors and a degree of parallelism typical in a parallel environment, good compression performance can be obtained even for dense floating point data with an off-the-shelf compression utility. Further, compression performance scaled well with the numbers of processors, for all but the most compressible data sets. Second, the increase in I/O cost due to compression in CC and SC was relatively small compared to the migration performance gain, even for dense data sets, making compressed migration attractive for most computation phase lengths.

CC is the best-performing approach with Panda, primarily because it utilizes many more processors for compression. In our experiments, the cost of an I/O phase at most increases slightly when CC is used. However, CC does not allow data to be reorganized as it is moved between compute processors and I/O processors. Thus CC will work well with parallel codes that solve irregular problems, and do not use distributed arrays. When data reorganization is required, or two-phase I/O is used, our recommended approach is SC with a large number of I/O servers, a shared file system, and a small number of extra processors dedicated to migration.

References

[4] Personal communication with Mike Folk, head of the HDF project at NCSA.