Runtime Compression of MPI Messages to Improve the Performance and Scalability of Parallel Applications

Jian Ke and Martin Burtscher
Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University, Ithaca, NY 14853
{jke, burtscher}@csl.cornell.edu

Evan Speight
Future Systems Group
IBM Austin Research Lab
Austin, TX 78758
speight@us.ibm.com

ABSTRACT

Communication-intensive parallel applications spend a significant amount of their total execution time exchanging data between processes, which leads to poor performance in many cases. In this paper, we investigate message compression in the context of large-scale parallel message-passing systems to reduce the communication time of individual messages and to improve the bandwidth of the overall system. We implement and evaluate the cMPI message-passing library, which quickly compresses messages on-the-fly with a low enough overhead that a net execution time reduction is obtained. Our results on six large-scale benchmark applications show that their execution speed improves by up to 98% when message compression is enabled.

1. INTRODUCTION

Parallel computation on clusters of inexpensive workstations has become the standard method for constructing supercomputers out of commodity parts. Pairing industry-standard SMP or uniprocessor nodes with high-speed interconnection networks provides a computing platform that can achieve reasonable performance on a wide range of applications from databases to scientific algorithms.

In order to hide many of the implementation-specific details of the underlying network protocol, several portable message-passing libraries have been designed that allow a "write once, run anywhere" paradigm for large-scale computing needs. The Message Passing Interface (MPI) [13] is perhaps the most widely used of these libraries. MPI provides a rich set of interfaces for operations such as point-to-point communication, collective communication, and synchronization operations.

There has been much work on improving the performance of MPI runtime libraries. Some libraries, such as TMPI [15] and TOMPI [4], provide fast messaging between processes co-located on the same node via shared memory semantics that are completely hidden from the application writer. Other implementations [12, 14] take advantage of user-level networks such as VIA [5] or InfiniBand [11] to drastically reduce the amount of overhead associated with sending messages, reducing small message latency. Still other researchers have investigated ways to improve the performance of collective communication operations in MPI [16].

While reducing the latency of small messages can be beneficial, there has been little work on improving the achievable bandwidth of large messages because with large message sizes the utilization of most networks is relatively good in comparison. However, our research indicates that for many MPI applications large messages dominate the overall message makeup. This paper investigates the idea of employing a fast compression algorithm to improve the overall bandwidth achievable by the system during periods of heavy communication.

The latency to send a message to another process comprises the message setup overhead and the time for the message to pass through the network, the latter of which roughly equals to the message size divided by the network bandwidth. The setup overhead can be expressed as a fixed cost plus a term that is proportional to the message size. Therefore, the total latency L for a message of size S is

$$L \approx l_0 + l_1 S + \frac{S}{BW}, \qquad (1)$$

where l_0 is the constant setup overhead, l_1 is the per byte overhead, and BW is the network bandwidth. When compressing messages before they are sent and decompressing them at the receiving end, the latency becomes

$$L' \approx l_0 + l_1 S + l_0' + l_1' S + \frac{S/R}{BW}, \qquad (2)$$

where $l_0' + l_1' S$ is the overhead incurred by the compression and the decompression and R is the compression rate. For the compression to reduce the communication latency, $L' < L$ must hold. Using the above two equations, the inequality can be rewritten as

$$(\frac{1}{BW}\frac{R-1}{R} - l_1')S - l_0' > 0. \qquad (3)$$

Since l_0' and S cannot be negative, the term in parentheses must be sufficiently greater than zero for the above inequality to hold. Hence, the compression overhead per message byte must at least satisfy

$$\frac{1}{BW}\frac{R-1}{R} > l_1'. \qquad (4)$$

0-7695-2153-3/04 $20.00 (c)2004 IEEE

In Table 1, we tabulate the maximum available CPU cycles to compress each message byte for various compression rates assuming a platform with a 3GHz processor and a 1Gbps network bandwidth.

Table 1: Compression speed requirements.

compression rate	1.2	1.5	2.0	4.0
max. cycles per byte	4	8	12	18

For instance, with a compression rate of 1.5, the CPU needs to compress and decompress one byte every eight cycles. Since CPUs operate on four or eight bytes at a time, there are actually 32 cycles available per word on, for example, a Pentium-style machine. This corresponds to roughly one hundred machine instructions (assuming no stalls), as Pentiums can execute multiple instructions per cycle, which is sufficient to run a low-overhead compression algorithm. Finally, the compression and decompression can be overlapped as will be discussed in Section 2.4.

This paper introduces cMPI, a library that automatically compresses and decompresses MPI messages at runtime without any application-level source code modifications. cMPI currently provides the forty most commonly-used MPI functions, which is enough to cover the vast majority of MPI applications. We evaluate cMPI on a set of benchmarks from the NAS Parallel Benchmark Suite [1] and the ASCI Purple Benchmark Suite [8]. Our results show that cMPI can improve parallel application scaling beyond the point of an MPI library that does not employ a compression scheme, resulting in up to 98% reduction in overall execution time.

The rest of this paper is organized as follows. Section 2 describes the design of the cMPI library. Section 3 presents the experimental evaluation methodology used. Section 4 discusses results of the cMPI library on the Velocity+ supercomputing cluster at the Cornell Theory Center. Section 5 presents conclusions and avenues for future work.

2. IMPLEMENTATION

In this section we describe the design of our cMPI library, the compression algorithm that allows cMPI to make better use of available network bandwidth, and several performance-enhancing optimizations.

2.1 The cMPI Library

We have implemented a commonly used subset of forty MPI functions in our cMPI library, covering most point-to-point communications, collective communications, and communicator creation APIs in the MPI specification [13]. The library is written in C and provides an interface for linking with Fortran applications. cMPI utilizes TCP as the underlying network protocol and creates one TCP connection between every two communicating MPI processes. Each process creates a message thread to handle sending to and receiving from all communication channels. This thread also compresses and decompresses appropriate messages if the corresponding environment variable is set, that is, if compression is enabled. A flag in the cMPI message header marks whether or not a particular message has been compressed so that the receiver can interpret the message correctly.

When calling a send function in MPI, the application must specify the message data type to the underlying MPI library. Based on this type, an appropriate compression method can be selected. Since the majority of the messages in numeric applications consist of arrays of MPI_DOUBLEs, in the initial implementation presented in this paper we only compress messages that consist of the type MPI_DOUBLE. Choosing a suitable compression algorithm for different MPI data types is the subject of ongoing work.

2.2 Compression Scheme

Our compression technique employs a value predictor to forecast message entries based on earlier entries. The compression is performed one MPI_DOUBLE at a time. To compress an MPI_DOUBLE, we predict its value and then encode the difference between the predicted and the true value. If the prediction is close to the true value, the difference can be encoded in just a few bits.

Figure 1 illustrates how the fourth value D_4 in a message of 64-bit MPI_DOUBLEs is compressed. First, the DFCM value predictor (see Section 2.3) produces a guess D'_4. Then we *xor* D_4 and D'_4 to obtain the difference $Diff_4$. $Diff_4$ has many leading zero bits if the prediction D'_4 is close to D_4. The leading zeros are then encoded using a leading zero count (LZC). The remaining bits (EBits) are not compressed.

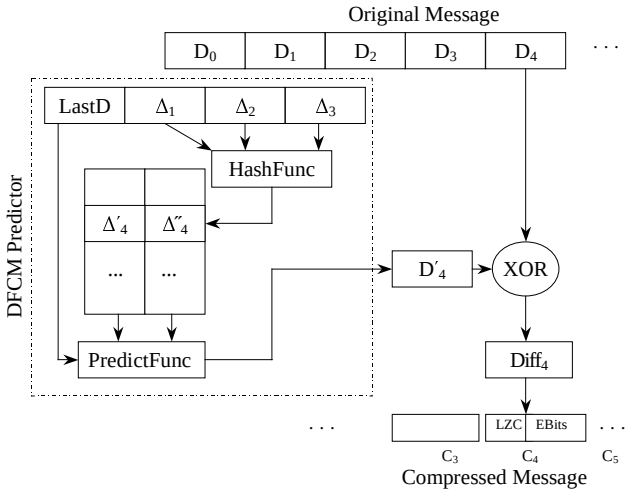

Figure 1: The compression algorithm.

In our compression scheme, we use four bits for the LZC, which encodes *4*LZC* leading zeros. Note that this

scheme provides the same average code length as a six-bit LZC if the leading zero counts are evenly distributed. For maximum speed, we wrote the leading zero counter in inline assembly code, where we take advantage of the Pentium's leading-zero-count instruction [7].

We chose not to use a more sophisticated compression scheme because the (de)compression time lies on the critical path for message transmission and reception. Therefore, this code's execution time needs to be kept very short so that reductions in message latency are not lost due to the (de)compression overhead.

At the receiver side, the messaging thread first reads the four-bit LZC and then *64-4*LZC* effective bits to regenerate the difference $Diff_4$. The predictor at the receiving end is kept consistent with the sender's predictor by always updating both predictors with the same values, i.e., the previously seen MPI_DOUBLEs. Thus both predictors are guaranteed to produce the same prediction D'_4. The true value D_4 can therefore trivially be regenerated by *xor*ing $Diff_4$ with D'_4.

2.3 The DFCM Predictor

The differential-finite-context-method predictor (DFCM) [6] computes a hash out of the *n* most recently encountered differences between consecutive values in the original message, where *n* is referred to as the order of the predictor. Figure 1 shows the third-order DFCM predictor we use. It performs a table lookup using the hash as an index to retrieve the differences that followed the last two times the same hash was encountered, i.e., the last two times that same sequence of last three differences was observed. The retrieved differences are used to predict the next value by adding them to the previous value in the message as explained below. Once the prediction has been made, the predictor is updated with the true difference and value.

The DFCM predictor exploits both spatial and temporal locality in MPI messages. Scientific applications often communicate data of adjacent simulation points in the same message. Each simulation point typically consists of multiple physical properties. The property lists of adjacent simulation points all exhibit the same structure and the values of the properties of two adjacent simulation points are often numerically close. For instance, each simulation point in a weather forecast application may include properties such as the pressure and the temperature. The temperatures of two spatially adjacent simulation points should differ only slightly. Hence, such data patterns can readily be captured by the predictors and looked up when similar patterns repeat in the same or a subsequent message.

The DFCM predictor was originally proposed as a micro-architectural enhancement to predict the content of CPU registers [6]. Recently, it has been modified and successfully used to compress program traces [2, 3]. We found the DFCM predictor with the following modifications to predict and compress floating-point messages well.

Hash function: For sequences of floating-point values, the chance of an exact 64-bit prediction match is low. Moreover, it is desirable that, for example, the decimal difference sequence (0.6001, 0.9001) be hashed to the same index as the sequence (0.6000, 0.9000) in a second-order DFCM predictor. For this reason, our hash function uses only the *m* most significant bits and ignores the remaining bits. Our experiments show that hashing only the first fourteen bits (the sign bit, eleven exponent bits, and two mantissa bits) results in the best average prediction accuracy. We use the following hash function.

$$hash(\Delta_0, \Delta_1, \Delta_2) = lsb_{0..14}(\Delta_2 \otimes (\Delta_1 << 5) \otimes (\Delta_0 << 10))$$

In this function, $\otimes$ denotes bit-wise *xor*, $<<$ denotes bit-wise left shift with zero insertion, and the Δ_i stand for the most significant fourteen bits of the difference between consecutive MPI_DOUBLE values in the MPI message. The lowest five bits of the Δ_i consist of three exponent and two mantissa bits and thus contain the most frequently changing bits. Shifting each Δ_i by five bits before *xor*ing them moves the frequently changing bits of the three Δ_i into non-overlapping positions, which we found to decrease the chance of detrimental aliasing in the hash table. Note that we only need the fifteen least-significant bits of the *xor* result for the index.

Prediction function: Instead of keeping just the latest Δ in the hash-table, we keep two deltas, Δ" and Δ'. These represent the most and the second-most recent difference values. Each Δ is a full 64-bit difference value. The predicted difference Δ_p is set to Δ" if Δ" and Δ' are not close to each other, i.e., the first fourteen bits are not the same. Otherwise, Δ_p is set to Δ"+(Δ"-Δ'). In other words, we use a conventional DFCM predictor except if the two Δs are almost the same, in which case we add Δ"-Δ' to account for the drift in the difference values, which we found to improve the prediction accuracy and thus the resulting compression rate.

2.4 Optimizations

To reduce the (de)compression overhead, cMPI posts a send whenever the compressed message size reaches one or two times the Maximum Transmission Unit (MTU) of the network interface, which is typically 1500 bytes for Ethernet. This allows the receiver thread to start decompressing the message as early as possible, hiding some of the compression overhead.

For small messages, the setup overhead dominates the total messaging time. Our experiments show that compressing messages below a certain threshold yields no performance improvement due to the overhead introduced by the compression algorithm. In fact, for small messages the compression overhead may easily exceed the improved message bandwidth, as is evident from Equation 3. Hence, we only invoke compression for messages that are at least 128 MPI_DOUBLEs (one kilobyte in our system) in size.

Since processes usually exchange large messages with only a small number of other processes and a predictor is only created when the first message of at least one kilobyte is seen, employing such a cutoff also reduces the number of predictors needed in each process, which in turn reduces the memory requirement.

3. EVALUATION METHODS

In this section we describe the system we use to generate our results as well as the benchmark applications used in each experiment.

3.1 System

We performed all measurements on the Velocity+ cluster at the Cornell Theory Center [10]. Velocity+ runs Microsoft Windows 2000 Advanced Server and consists of 64 dual-processor nodes with 733 MHz Intel Pentium III processors and two gigabytes of RAM per node. The network we used is 100Mb/s Ethernet, interconnected by 3Com 3300 24-port switches. Note that although we use Fast Ethernet in our experiments due to resource constraints, our processor speed is also correspondingly lower than current state-of-the-art microprocessors. We run large enough problem sizes that our chosen applications scale reasonably well, even in the baseline cases (see Table 3). We are currently migrating our experiments to a cluster with faster processors and network.

Table 2: Benchmark program information.

Program	Lang	Problem Size	Description
LU	F77	Class C	Simulated CFD application that uses symmetric successive over-relaxation (SSOR) to solve a block lower triangular-block upper triangular system of equations
BT	F77	Class C	Simulated CFD applications that solve systems of equations resulting from an approximately factored implicit finite-difference discretization of the Navier-Stokes equations
SP	F77	Class C	
sPPM	F77	384x384x384	3-D gas dynamics problem on a uniform Cartesian mesh, using a simplified version of the PPM (Piecewise Parabolic Method)
Sweep3D	F77	256x256x256	Solver for the 3-D, time-independent, particle transport equation
AZTEC	C/F77	31855013	Parallel iterative library for solving linear systems

3.2 Benchmarks

Our benchmark suite consists of three applications from the NAS Parallel Benchmark Suite (NPB) [1] and three applications from the ASCI Purple Benchmarks [8]. The NAS Parallel Benchmarks are a set of eight programs derived from computational fluid dynamics (CFD) applications consisting of five kernels and three pseudo-applications. We use the three pseudo-applications LU, BT, and SP. In addition, we use sPPM, Sweep3D, and AZTEC from the ASCI Purple Benchmark suite (Sweep3D is no longer included in the current release of the Purple benchmarks but its source code is still available [9]). Table 2 provides an overview of these applications. BT and SP require the number of processes to be a square number. Hence, we run them on 36, 64, and 100 processors. LU requires the process count to be a power of two, so we run it on 32, 64, and 128 processors. The three remaining applications, sPPM, Sweep3D, and AZTEC, are also run on 32, 64, and 128 processors.

3.3 Predictor Configuration

Our experiments show that third-order DFCM predictors with a hash-table size of 2^{15} lines work well for all six applications. Higher order predictors do not improve the compression rate. Larger hash tables increase the compression rate slightly, but are not worthwhile because of their much larger memory requirement. Hence, we use hash tables with 2^{15} lines for all experiments.

Each line in the hash table requires 16 bytes to store two MPI_DOUBLEs in our system. Thus, the total table size is 512 kilobytes (2^{15} * 16 bytes). Due to the minimum message-length requirement introduced in Section 2.4, only four to twelve predictors are created in each process. Hence, no more than six megabytes of memory are allocated for the predictor tables.

4. RESULTS

Section 4.1 compares the runtime of all applications with different numbers of processes on MPI-Pro, the commercial MPI implementation used on Velocity+, and on cMPI with message compression turned off. This is done to ensure that any speedups we obtain with compressed messages are not due to a poor baseline implementation. Section 4.2 studies the performance improvement when message compression is turned on. We conduct measurements for different numbers of processes to evaluate the effect of message compression on the scalability of the six applications. Section 4.3 presents message information and compression rates. Section 4.4 investigates the time spent compressing and decompressing messages.

4.1 cMPI Baseline Performance

We first compare our MPI library with MPI-Pro, a widely used commercial MPI implementation. Figure 2 plots the ratio of the baseline cMPI's execution time normalized to that of MPI-Pro for our six benchmark applications and various numbers of processes. Results below one indicate that cMPI (without compression) is faster than MPI-Pro. The absolute runtimes (in seconds) are given in Table 3. Note that for improved readability, most of the figures in this paper are not zero based.

When BT is run on 100 and SP on 64 or 100 processors, cMPI outperforms MPI-Pro by almost 20%. On the other hand, MPI-Pro is faster than cMPI for some of the other programs and configurations, though never by more than 7.2%. The two MPI implementations perform within about 5% of each other in the majority of the cases. The results

clearly show cMPI (without compression) to be competitive with MPI-Pro. All remaining experiments use cMPI without compression as their baseline.

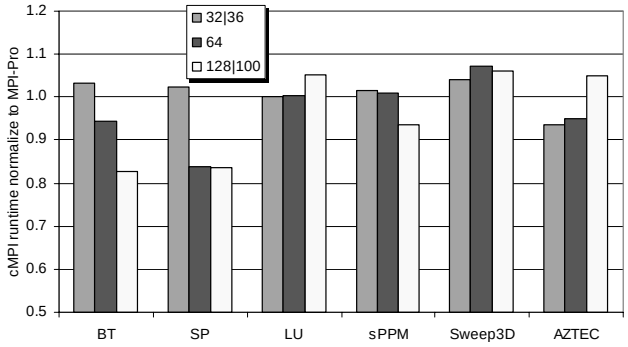

Figure 2: Runtime of cMPI normalized to MPI-Pro.

Table 3: Benchmark runtimes in seconds.

program	# procs	MPI-Pro	cMPI baseline	cMPI with compres.
BT	36	3079	3176	3160
	64	2207	2086	1986
	100	1861	1537	1399
SP	36	2821	2885	2863
	64	2811	2361	2001
	100	2552	2130	1692
LU	32	3122	3128	3120
	64	1690	1696	1665
	128	1074	1130	1098
sPPM	32	2173	2203	2173
	64	1184	1195	1108
	128	1283	1200	607
Sweep3D	32	523	544	535
	64	278	298	289
	128	195	207	198
AZTEC	32	2340	2191	2019
	64	1606	1526	1343
	128	1470	1543	1198

4.2 Speedup with Message Compression

The rightmost column in Table 3 shows the runtime of cMPI when message compression is enabled. The corresponding speedup over no compression is depicted in Figure 3. Numbers above one indicate that an application runs faster with compression than without.

The figure shows that all applications show improved performance when message compression is turned on. sPPM improves by 98 percent on the 128-process run. As can be seen in Table 3, sPPM does not scale to more than 64 processors on our baseline system. However, our compression approach allows this application to scale almost perfectly to 128 processors. AZTEC's speed improves by up to 29% in the 128-process run, which also does not scale on the baseline system. For this application, message compression improves the overall performance at each processor-point in addition to increasing the scalability. The other four applications achieve a performance improvement of 3% to 26% in 128-process runs (100-process runs for BT and SP). Overall, these results clearly demonstrate the improved scalability that can result from utilizing our message compression technique in MPI.

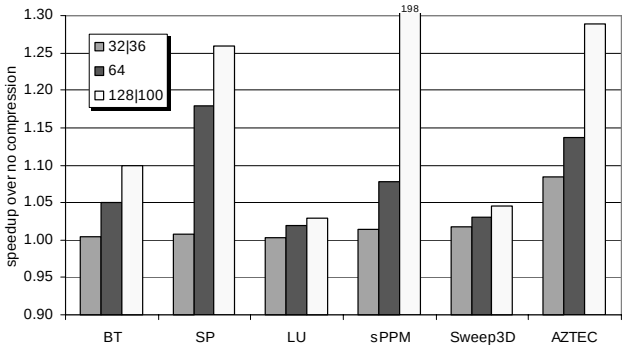

Figure 3: Speedups due to message compression.

4.3 Message Information and Compression Rate

The net saving in communication time and the fraction of the total runtime that is communication time determine the performance improvement due to message compression. The former is in turn determined by the compression rate and the latter is interchangeable with the communication to computation ratio. The higher the compression rate and the larger the communication to computation ratio, the greater the message-compression effect.

Table 4: Statistics about large messages (above 1kB).

program	# of procs	message count	average size (kB)	predictors per process
BT	64	617,856	80.9	12
SP	64	1,232,256	71.8	12
LU	64	56,476	256.2	7
sPPM	64	74,048	654.4	11
Sweep3D	64	286,720	24.0	7
AZTEC	64	207,522	785.1	3.9
BT	100	1,206,600	54.1	12
SP	100	2,406,600	47.3	12
LU	128	116,972	194.4	7.3
sPPM	128	156,480	428.2	11.5
Sweep3D	128	593,920	18.2	7.3
AZTEC	128	418,338	785.1	4

Table 4 summarizes the message information for each application on 64- and 128|100-process runs. The *message count* is the sum of all messages on all processes that are larger than one kilobyte. As we can see, the message count roughly doubles when the number of processes is doubled, so the message count per process remains about the same. However, the computation per process is usually halved, as problem sizes remain the same across varying numbers of processors. At the same time, the average message size decreases by between zero and 35%, meaning that the communication to computation ratio increases substantially

as the number of processes doubles. Thus, assuming a constant compression rate, we expect higher speedups due to message compression as the number of processes increases, which the results in Figure 3 confirm. In other words, the compression scheme we employ allows for an overall improvement in the application's use of available network bandwidth, reducing overall communication time and improving performance.

The *predictors per process* numbers in Table 4 record the average number of (de)compression predictors created by each process. Our experiments show that the maximum number of predictors in any process is twelve. Since one compression predictor and one decompression predictor are created for each channel that has large (above one kilobyte) messages, the number of *predictors per process* divided by two yields the average number of major communicating neighbors of each process.

While a high communication to computation ratio provides opportunity for speedup due to message compression, the compression rate dictates the final success. The compression rate is the size of the original message divided by the size of the compressed message. Applications with highly predictable message values will demonstrate higher compressibility as described in Section 2.2. Interestingly, we found the compression rate to be rather constant for different numbers of processes with each application. This appears to be an indication that the message compressibility is program dependent but independent of the degree of parallelization. Hence, we only list the compression rates for two problem sizes for each application in Table 5.

Table 5: Compression rates.

# processes	BT	SP	LU	sPPM	Sweep3D	AZTEC
64	1.35	1.30	1.24	4.39	1.39	1.46
128|100	1.36	1.29	1.24	4.63	1.40	1.46

A compression rate of over four for sPPM and 1.46 for AZTEC, together with their large average message sizes, leads to the excellent runtime reductions shown in Figure 3. The other four applications exchange shorter messages and have a compression rate between 1.24 and 1.4, which is why they exhibit smaller speedups. sPPM's messages are highly compressible because they contain large chunks of non-zero values that only differ in the last few bits.

Note that we excluded the kernel benchmark applications from our study because their message data patterns, and hence the compressibility, may not be representative of real applications. For the reasons discussed in Section 2.3, we believe that the message compressibility demonstrated in the applications we did investigate in this paper is characteristic of many real applications.

4.4 Compression Overhead

The average compression and decompression times in the 128-process runs (100-process runs for BT and SP) are plotted in Figure 4 as a percentage of the total runtime. They lie in the range of 0.4 to 1.8 percent of the runtime for most applications except AZTEC, where they represent 5.7 percent of the runtime. Compressing the messages takes approximately the same time as decompressing them in all six applications. Note that even though the (de)compression overhead is non-negligible, we obtain an overall performance gain. This is mainly due to the larger benefit of the reduced communication time. In addition, some of the (de)compression may have been overlapped with message completion, reducing the overhead imposed by our compression scheme.

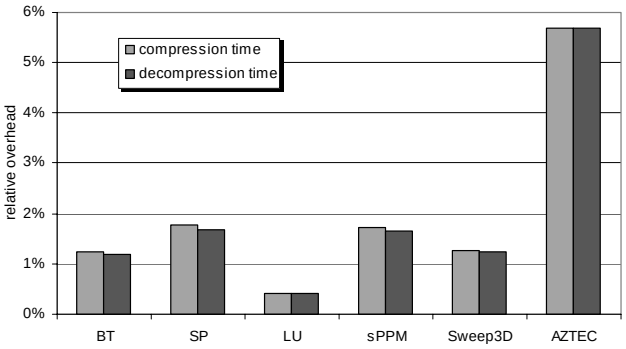

Figure 4: Average (de)compression overhead (128|100-process runs).

5. CONCLUSIONS AND FUTURE WORK

This paper shows that the messages of large-scale parallel scientific applications are compressible and introduces message compression as part of an MPI library to reduce the messaging overhead. Our novel compression algorithm is based on value prediction and encodes the difference between the true value and the predicted value to save bits. The compression algorithm is fast and provides good compression rates for all applications we have investigated. The saved messaging overhead outweighs the compression and decompression overhead in all applications, resulting in an overall runtime reduction. We observed speedups on 128-process runs of at least 3% for all benchmarks and up to 98% in one case.

The compression is handled by the MPI library and is therefore completely transparent to user applications. MPI-library providers can easily add our compression scheme to their implementation, which will immediately benefit a wide range of parallel programs without any source-code changes at the application level.

We are planning to evaluate the benefits of message compression on other parallel machines with various networking speeds. The relative speed of communication to computation will have a significant effect on the performance gain achievable through message compression. In particular, we would like to investigate the potential benefits of message compression on future architectures and computing grids.

Another direction for exploration is the possibility of off-loading the compression and decompression to the network interface card. Due to the simplicity of our compression approach, the (de)compression can be performed directly by a NIC processor and may even be implementable in the NIC hardware.

Finally, since the compression rate is crucial to the ultimate performance, we are also investigating other compression algorithms that better exploit the unique characteristics of MPI messages. Adaptively choosing the best compression algorithm for a particular message type may yield more performance gains on applications in areas other than scientific computing.

6. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under Grants No. 0125987, 0208567, and 0312966. This research was conducted using the resources of the Cornell Theory Center, which receives funding from Cornell University, New York State, federal agencies, foundations, and corporate partners.

7. REFERENCES

[1] D. Bailey, T. Harris, W. Saphir, R. v.d. Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," *Technical Report NAS-95-020, NASA Ames Research Center*, December 1995.

[2] M. Burtscher and M. Jeeradit, "Compressing Extended Program Traces Using Value Predictors," *International Conference on Parallel Architectures and Compilation Techniques*, September 2003, pp. 159-169.

[3] M. Burtscher, "VPC3: A Fast and Effective Trace-Compression Algorithm," *Joint International Conference on Measurement and Modeling of Computer Systems*, June 2004, pp. 167-176.

[4] E. D. Demaine, "A Threads-Only MPI Implementation for the Development of Parallel Programs," *International Symposium on High Performance Computing Systems*, July 1997, pp. 153-163.

[5] D. Dunning, G. Regnier, G. McApline, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd, "The Virtual Interface Architecture," *IEEE Micro*, March/April 1998, pp. 66-76.

[6] B. Goeman, H. Vandierendonck, and K. Bosschere, "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency," *Seventh International Symposium on High Performance Computer Architecture*, January 2001, pp. 207-216.

[7] http://www.intel.com/design/pentium/MANUALS/24319101.PDF

[8] http://www.llnl.gov/asci/purple/

[9] http://www.llnl.gov/asci_benchmarks/

[10] http://www.tc.cornell.edu/

[11] Infiniband Trade Association, *Infiniband Architecture Specification, Release 1.0*, October 2000.

[12] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand," *International Conference on Supercomputing*, June 2003, pp. 295-304.

[13] MPI Forum, "MPI: A Message-Passing Interface Standard," *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):165-414, 1994.

[14] E. Speight, H. Abdel-Shafi, and J. K. Bennett, "Realizing the Performance Potential of the Virtual Interface Architecture," *International Conference on Supercomputing*, June 1999, pp. 184-192.

[15] H. Tang and T. Yang, "Optimizing Threaded MPI Execution on SMP Clusters," *International Conference on Supercomputing*, June 2001, pp. 381-392.

[16] R. Thakur and W. Gropp, "Improving the Performance of Collective Operations in MPICH," *European PVM/MPI Users' Group Conference*, September 2003, pp. 257-267.