DE-FG02-06ER25722
Final Report, March 2009

Portable and Transparent Message Compression in MPI Libraries to Improve the Performance and Scalability of Parallel Applications

David Albonesi, School of Electrical and Computer Engineering, Cornell University
Martin Burtscher, Computational Engineering and Sciences, The University of Texas at Austin

Signature: David A Albonesi Date: 4/17/09

OVERVIEW

The goal of this project has been to develop a lossless compression algorithm for message-passing libraries that can accelerate HPC systems by reducing the communication time. Because both compression and decompression have to be performed in software in real time, the algorithm has to be extremely fast while still delivering a good compression ratio.

During the first half of this project, we designed a new compression algorithm called FPC for scientific double-precision data, made the source code available on the web, and published two papers describing its operation, the first in the proceedings of the Data Compression Conference and the second in the IEEE Transactions on Computers. At comparable average compression ratios, this algorithm compresses and decompresses 10 to 100 times faster than BZIP2, DFCM, FSD, GZIP, and PLMI on the three architectures we tested. With prediction tables that fit into the CPU's L1 data cache, FPC delivers a guaranteed throughput of six gigabits per second on a 1.6 GHz Itanium 2 system. The C source code and documentation of FPC are posted on-line and have already been downloaded hundreds of times.

To evaluate FPC, we gathered 13 real-world scientific datasets from around the globe, including satellite data, crash-simulation data, and messages from HPC systems. Based on the large number of requests we received, we also made these datasets available to the community (with permission of the original sources).

While FPC represents a great step forward, it soon became clear that its throughput was too slow for the emerging 10 gigabits per second networks. Hence, no speedup can be gained by including this algorithm in an MPI library. We therefore changed the aim of the second half of the project (in coordination with the program manager Fred Johnson). Instead of implementing FPC in an MPI library, we re-focused our efforts to develop a parallel compression algorithm to further boost the throughput. After all, all modern high-end microprocessors contain multiple CPUs on a single die and will undoubtedly contain even larger numbers of CPUs in the future. Hence, a compute node in a parallel system is likely to have multiple processors per interconnection link. Thus, we investigated how to best exploit multiple CPUs to compress and decompress data for a single link, i.e., we developed a parallel version of the FPC algorithm called pFPC.

On a 4-core 3 GHz Xeon system, pFPC exceeds 10 gigabits per second both in compression and decompression throughput while still reaching a compression ratio of 18%, which is reasonable for our double-precision floating-point datasets. At half this throughput, it achieves a 41% compression ratio. Moreover, the algorithm scales well to four cores and promises even higher throughputs on larger machines. Additional results and a description of the pFPC algorithm have recently been published in the proceedings of the Data Compression Conference. The C source code of pFPC is also publicly available.

In summary, we achieved all goals of this project, in particular the design of a very fast compression algorithm for floating-point data and a software implementation thereof with a throughput above 10 gigabits per second. There were no problems or delays, no absence or changes of key personnel, and we have used the funds as budgeted.

The project has resulted in the following products:

1) The sequential FPC compression algorithm
2) The parallel pFPC compression algorithm
3) 13 real-world scientific datasets
4) Three peer-reviewed publications
   a) A DCC'07 publication describing FPC
      http://users.ices.utexas.edu/~burtscher/papers/dcc07a.pdf
   b) A TC'09 journal article describing FPC in much more detail
      http://users.ices.utexas.edu/~burtscher/papers/tc09.pdf
   c) A DCC'09 publication describing pFPC
      http://users.ices.utexas.edu/~burtscher/papers/dcc09.pdf
5) One non-refereed technical report describing the source code of FPC line by line
   http://users.ices.utexas.edu/~burtscher/papers/tr08.pdf
6) Three web sites for the dissemination of the results of this project
   a) A download site for the FPC source code and documentation
      http://www.csl.cornell.edu/~burtscher/research/FPC/
   b) A download site for the 13 datasets
      http://www.csl.cornell.edu/~burtscher/research/FPC/datasets.html
   c) A download site for the pFPC source code and documentation
      http://users.ices.utexas.edu/~burtscher/research/pFPC/

The remainder of this document describes the operation and performance of FPC and pFPC.

THE FPC ALGORITHM

FPC compresses linear sequences of IEEE 754 double-precision floating-point values by sequentially predicting each value, xoring the true value with the predicted value, and leading-zero compressing the result. As illustrated in Figure 1, it uses variants of an *fcm* [13] and a *dfcm* [7] value predictor to predict the doubles. Both predictors are effectively hash tables. The more accurate of the two predictions, i.e., the one that shares more common most significant bits with the true value, is xored with the true value. The xor operation turns identical bits into zeros. Hence, if the predicted and the true value are close, the xor result has many leading zeros. FPC then counts the number of leading zero bytes, encodes the count in a three-bit value, and concatenates it with a single bit that specifies which of the two predictions was used. The resulting four-bit code and the nonzero residual bytes are written to the output. The latter are emitted verbatim without any encoding.

FPC outputs the compressed data in blocks. Each block starts with a header that specifies how many doubles the block encodes and how long it is (in bytes). The header is followed by the stream of four-bit codes, which in turn is followed by the stream of residual bytes. To maintain byte granularity, which is more efficient than bit granularity, a pair of doubles is always processed together and the corresponding two four-bit codes are packed into a byte. In case an odd number of doubles needs to be compressed, a spurious double is encoded at the end. This

spurious value is later eliminated using the count information from the header. For speed reasons, FPC keeps the four-bit codes and the residual bytes separate instead of interleaving them.

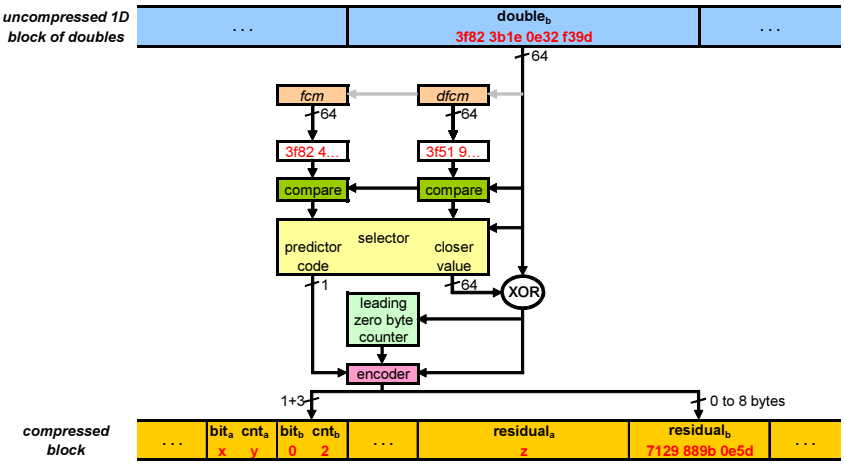

Figure 1: FPC compression algorithm overview

Decompression starts by reading the current four-bit code, decoding the three-bit field, reading the specified number of residual bytes, and zero-extending them to a full 64-bit number. Based on the one-bit field, this number is xored with either the 64-bit *fcm* or *dfcm* prediction to recreate the original double. This lossless reconstruction is possible because xor is reversible.

For performance reasons, FPC interprets all doubles as 64-bit integers and uses only integer arithmetic. Since there can be between zero and eight leading zero bytes, i.e., nine possibilities, not all of them can be encoded with a three-bit value. We decided not to support a leading zero count of four because it occurs only rarely [4]. Consequently, all xor results with four leading zero bytes are treated like values with only three leading zero bytes and the fourth zero byte is emitted as part of the residual.

Before compression and decompression, both predictor tables are initialized with zeros. After each prediction, they are updated with the true double value to ensure that they generate the same sequence of predictions during compression as they do during decompression. The following pseudo code demonstrates the operation of the *fcm* predictor. The `table_size` has to be a power of two. `fcm` is the hash table.

```
unsigned long long true_value, fcm_prediction, fcm_hash, fcm[table_size];
...
fcm_prediction = fcm[fcm_hash]; // prediction: read hash table entry
fcm[fcm_hash] = true_value; // update: write hash table entry
fcm_hash = ((fcm_hash << 6) ^ (true_value >> 48)) & (table_size - 1);
```

Right shifting the `true_value` (i.e., the current double expressed as a 64-bit integer) by 48 bits eliminates the often random mantissa bits. The remaining 16 bits are xored with the previous hash value to produce the new hash. However, the previous hash is first shifted by six bits to the left to gradually phase out bits from older values. The hash value (`fcm_hash`) therefore represents the sequence of most recently encountered doubles, and the hash table stores the double that follows this sequence. Hence, making an *fcm* prediction is tantamount to performing a table lookup to determine which value followed the last time a similar sequence of previous doubles was seen.

The *dfcm* predictor operates in the same way. However, it predicts integer differences between consecutive values rather than absolute values, and the shift amounts in the hash function are different.

```
unsigned long long last_value, dfcm_prediction, dfcm_hash, dfcm[table_size];
...
dfcm_prediction = dfcm[dfcm_hash] + last_value;
dfcm[dfcm_hash] = true_value - last_value;
dfcm_hash = ((dfcm_hash << 2) ^ ((true_value - last_value) >> 40)) &
 (table_size - 1);
last_value = true_value;
```

The complete C source code and a brief description of how to compile and use it are available at http://www.csl.cornell.edu/~burtscher/research/FPC/. The web site also contains a detailed discussion of the code and some of the optimization techniques it employs. Additional information on FPC is available elsewhere [3, 4].

THE pFPC ALGORITHM

We have investigated two main techniques to parallelizing FPC. The first approach is to divide the data into chunks and have multiple instances of FPC losslessly compress or decompress the chunks in parallel. The number of threads (instances) and the chunk size are user selectable. The second approach is to split the algorithm into multiple components and execute the components in parallel. This is typically done in a pipelined fashion where every stage performs part of the processing and streams its (intermediate) results to the next stage. The number of stages, and therefore the number of parallel threads, is determined by the programmer, making this the less flexible of the two approaches. Nevertheless, the two parallelization approaches are independent and can be combined.

Because FPC is memory and not compute bound [4], the pipelined approach, which introduces a lot of extra memory traffic to stream the intermediate data from one core to the next, does not work well. All our attempts to pipeline FPC have failed in the sense that the parallel code was always slower than the sequential code. Hence, we focused on the chunked approach.

There are two key parameters in the chunked approach: the chunk size and the thread count. The chunk size determines the number of consecutive doubles that make up a chunk. Depending on the size of the data to be processed, the last chunk may not be a full-size chunk. The thread count determines the number of parallel compression or decompression threads that work together. Depending on the data size and thread count, some threads may be assigned one fewer chunk than others. Chunks are assigned in a round-robin fashion to the threads. If c is the chunk size (in number of doubles), n is the thread count, and v_i is the i^{th} value in the sequential data stream, then the k^{th} chunk consists of values v_{k*c} through $v_{(k+1)*c-1}$ and is assigned to the $k \bmod n^{th}$ thread. Figure 2 illustrates this process with two threads.

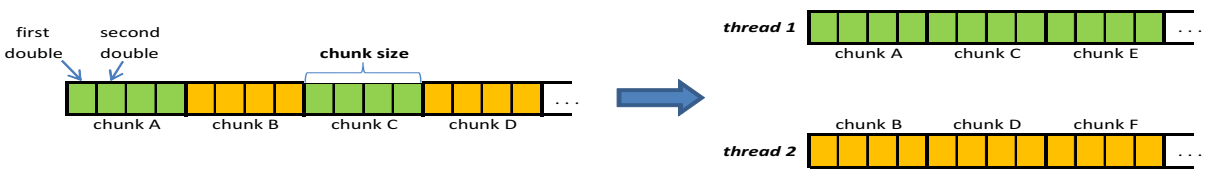

Figure 2: pFPC data chunking and thread assignment

Note that pFPC is a reimplementation of the FPC algorithm that runs slower with one thread than the original FPC implementation on our Itanium 2 system but faster on x86-based systems. Moreover, pFPC is easier to read and simpler to integrate with other code. The C source code is available at http://users.ices.utexas.edu/~burtscher/research/pFPC/. More detailed information about pFPC can be found elsewhere [5].

EVALUATION METHODOLOGY

System and Compiler

We compiled and evaluated FPC as well as other compressors on a 64-bit system with a 1.6 GHz Itanium 2 CPU, which has a 16 kB L1 data cache, a 256 kB unified L2 cache, a 3 MB L3 cache (on chip), and 3 GB of main memory. The operating system is Red Hat Enterprise Linux AS4 and the compiler is the Intel C Itanium Compiler version 9.1. We used the "-O3 -mcpu=itanium2 -static" compiler flags for each compressor.

We compiled and evaluated pFPC on a 64-bit system with two dual-core 3 GHz Xeon CPUs. Each of the four cores has a 32 kB L1 data cache. Each dual-core module contains a shared 4 MB unified L2 cache. The system has 4 GB of shared main memory. The operating system is Red Hat Linux 2.6.23.15. We used gcc version 4.1.2 with the "-O3 -march=core2 -static -pthread -std=c99" compiler flags.

For brevity, we only show results for these two systems in this report. Results for additional systems can be found elsewhere [4, 5].

Datasets

We used thirteen datasets from various scientific domains for our evaluation. Each dataset consists of a binary sequence of IEEE 754 double-precision floating-point numbers and belongs to one of the following categories.

Observational data: These datasets comprise measurements from scientific instruments.

- *obs_error*: observation input values specifying brightness temperature errors of a weather satellite
- *obs_info*: latitude and longitude of the observation points of a weather satellite
- *obs_spitzer*: observational data from the Spitzer Space Telescope showing a slight darkening as an extrasolar planet disappears behinds its star
- *obs_temp*: observational data from a weather satellite denoting how much the observed temperature differs from the actual contiguous analysis temperature field

Numeric simulations: These datasets are the results of numeric simulations.

- *num_brain*: simulation of the velocity field of a human brain during a head impact
- *num_comet*: simulation of the comet Shoemaker-Levy 9 entering Jupiter's atmosphere
- *num_control*: control vector output between two minimization steps in weather-satellite data assimilation
- *num_plasma*: simulated plasma temperature of a wire array z-pinch experiment

Parallel messages: These datasets capture the MPI messages sent by a node in a parallel system running NAS Parallel Benchmark (NPB) [1] and ASCI Purple [10] applications.

- *msg_bt*: NPB computational fluid dynamics pseudo-application bt
- *msg_lu*: NPB computational fluid dynamics pseudo-application lu
- *msg_sp*: NPB computational fluid dynamics pseudo-application sp
- *msg_sppm*: ASCI Purple solver sppm
- *msg_sweep3d*: ASCI Purple solver sweep3d

Table 1 summarizes information about each dataset. The first two data columns list the size in megabytes and in millions of double-precision values. The middle column shows the percentage of doubles in each dataset that are unique, i.e., appear exactly once. The fourth column displays the first-order entropy in bits. The last column expresses the randomness of the datasets in percent, that is, it reflects how close the first-order entropy is to that of a truly random dataset with the same number of doubles.

Table 1: Statistical information about each dataset

	size (megabytes)	doubles (millions)	unique values (percent)	1st order entropy (bits)	randomness (percent)
msg_bt	254.0	33.30	92.9	23.67	94.7
msg_lu	185.1	24.26	99.2	24.47	99.7
msg_sp	276.7	36.26	98.9	25.03	99.7
msg_sppm	266.1	34.87	10.2	11.24	44.9
msg_sweep3d	119.9	15.72	89.8	23.41	97.9
num_brain	135.3	17.73	94.9	23.97	99.5
num_comet	102.4	13.42	88.9	22.04	93.1
num_control	152.1	19.94	98.5	24.14	99.6
num_plasma	33.5	4.39	0.3	13.65	61.9
obs_error	59.3	7.77	18.0	17.80	77.8
obs_info	18.1	2.37	23.9	18.07	85.3
obs_spitzer	189.0	24.77	5.7	17.36	70.7
obs_temp	38.1	4.99	100.0	22.25	100.0

We observe that all datasets contain several million doubles. What is striking is that the datasets from all three categories appear to largely consist of unique values. Moreover, they are highly random from an entropy perspective, even the ones that do not contain many unique values (e.g., *num_plasma*).

Based on these statistics, it is unlikely that a pure entropy-based compression approach will work well. Note that the higher-order entropies are also close to random because of the large percentage of unique values. Clearly, we have to use a good data model or subdivide the doubles into smaller entities (e.g., bytes), some of which may exhibit less randomness, to compress these datasets well. FPC and pFPC incorporate both approaches.

Other Compressors

This subsection describes the compression schemes with which we compare FPC. GZIP and BZIP2 are lossless, general-purpose algorithms that can be used to compress any kind of data. The remaining algorithms represent our implementations of special-purpose floating-point compressors from the literature. They are all single-pass, lossless compression schemes that "know" about the format of double-precision values. We compiled the C source code of each algorithm described in this section with the same compiler and optimization flags as FPC.

BZIP2: BZIP2 [8] is a general-purpose compressor that operates at byte granularity. It implements a variant of the block-sorting algorithm described by Burrows and Wheeler [2]. It applies a reversible transformation to a block of inputs, uses sorting to group bytes with similar contexts together, and then compresses them with a Huffman coder. The block size is adjustable. We evaluate all supported block sizes, i.e., one through nine.

DFCM: Our previously proposed DFCM scheme [12] maps each encountered floating-point value to an unsigned integer and predicts it with a modified *dfcm* predictor. This predictor computes a hash out of the three most recently encountered differences between consecutive values in the input. Next, it performs a hash table lookup to retrieve the differences that followed the last two times the same hash was encountered, and one of the two differences is used to predict

the next value. A residual is generated by xoring the predicted value with the true value. This residual is encoded using a four-bit leading zero bit count. We evaluate predictor sizes between 16 B and 512 MB.

FSD: The FSD compressor implements the fixed step delta-algorithm proposed by Engelson et al. [6]. As it reads in a stream of doubles, it iteratively generates difference sequences from the original sequence. The order determines the number of iterations. A zero suppress algorithm is then used to encode the final difference sequence, where each value is expected to have many leading zeroes. Generally, gradually changing data tend to benefit from higher difference orders whereas rapidly changing data compress better with lower orders. We evaluate orders one through five (higher orders perform worse).

GZIP: GZIP [9] is a general-purpose compression utility that operates at byte granularity and implements a variant of the LZ77 algorithm [14]. It looks for repeating sequences of bytes (strings) within a 32 kB sliding window. The length of the string is limited to 256 bytes, which corresponds to the lookahead buffer size. GZIP uses two Huffman trees, one to compress the distances in the sliding window and another to compress the lengths of the strings as well as the individual bytes that were not part of any matched sequence. The algorithm finds duplicated strings using a chained hash table. A command-line argument determines the maximum length of the hash chains and whether lazy evaluation should be used. We evaluate all supported levels, i.e., one through nine.

PLMI: The PLMI scheme proposed by Lindstrom and Isenberg [11] uses a Lorenzo predictor in the front-end to predict 2D and 3D geometry data for rendering. In case of general linear data, as is the case with our datasets, the Lorenzo predictor reverts to a delta predictor, which processes data similar to the first-order FSD algorithm. Our implementation of PLMI employs a modified *dfcm* predictor (see above), which we found to compress linear data better. The predicted and true floating-point values are mapped to an unsigned integer from which a residual is computed by a difference process. The final step involves encoding the residual with range coding based on Schindler's quasi-static probability model. We evaluate predictor sizes between 16 B and 512 MB.

RESULTS

Compression Ratio

This subsection investigates the highest compression ratio that the six algorithms achieve on each dataset. Note that we individually optimized the size, level, or order for each algorithm and dataset to obtain the results shown in Table 2. The numbers in bold print reflect the best compression ratio for each dataset.

FPC achieves the highest geometric mean compression ratio because on four datasets it exceeds the performance of the other five algorithms by a large margin. The other algorithms substantially outperform FPC on only two datasets, *msg_sppm* and *obs_spitzer*.

No algorithm performs best on all datasets. In fact, no algorithm is best on more than five of the thirteen datasets. There is also no best algorithm within the three dataset categories. Even GZIP and BZIP2, the general-purpose compressors that have no knowledge of the format of double-precision floating-point values, provide the highest compression ratio on some of the datasets. Only FSD is outperformed on all of our datasets.

None of our datasets are highly compressible with any of the algorithms we studied. Only *msg_sppm* can be compressed by at least a factor of two with all six algorithms. Two datasets, *num_control* and *obs_temp*, cannot even be compressed by ten percent. These results are consis-

tent with the randomness information presented in Table 1, based on which we would expect *msg_sppm* to be the most and *obs_temp* the least compressible dataset. The highest overall compression ratio of 15.05 is obtained on *num_plasma*, which exhibits the second lowest randomness and the lowest percentage of unique values.

On some datasets, most notably *msg_sweep3d*, *num_plasma*, *obs_error*, and *obs_info*, and to a lesser extent *obs_spitzer*, one algorithm performs much better than the others. With the exception of *msg_sweep3d*, these datasets all have relatively few unique values and low randomness. The five datasets with above 99% randomness cannot be compressed by more than 26% by any of the algorithms we investigated.

Table 2: Maximum compression ratio of the six algorithms on each dataset

	BZIP2	DFCM	FPC	FSD	GZIP	PLMI
msg_bt	1.10	**1.36**	1.29	1.07	1.13	1.24
msg_lu	1.02	**1.24**	1.17	1.00	1.06	1.19
msg_sp	1.08	1.25	**1.26**	0.99	1.11	1.19
msg_sppm	6.93	4.23	5.30	2.35	**7.43**	5.02
msg_sweep3d	1.29	1.56	**3.09**	1.21	1.09	1.21
num_brain	1.04	**1.23**	1.16	1.10	1.06	1.12
num_comet	1.17	1.17	1.16	1.11	1.16	**1.18**
num_control	1.03	**1.07**	1.05	0.99	1.06	1.06
num_plasma	5.79	1.30	**15.05**	1.00	1.61	1.26
obs_error	1.34	1.52	**3.60**	1.16	1.45	1.26
obs_info	1.22	1.23	**2.27**	1.00	1.15	1.16
obs_spitzer	**1.75**	1.00	1.03	0.96	1.23	1.08
obs_temp	1.02	1.01	1.02	0.97	**1.04**	1.04
geo_mean	*1.52*	*1.36*	***1.95***	*1.11*	*1.35*	*1.30*

FPC Throughput

This subsection examines the compression and decompression throughput of the six algorithms (i.e., the raw dataset size divided by the runtime). Figure 3 plots the throughput in gigabits per second versus the geometric mean compression ratio on the Itanium 2 system. The individual data points correspond to different sizes, levels, or orders.

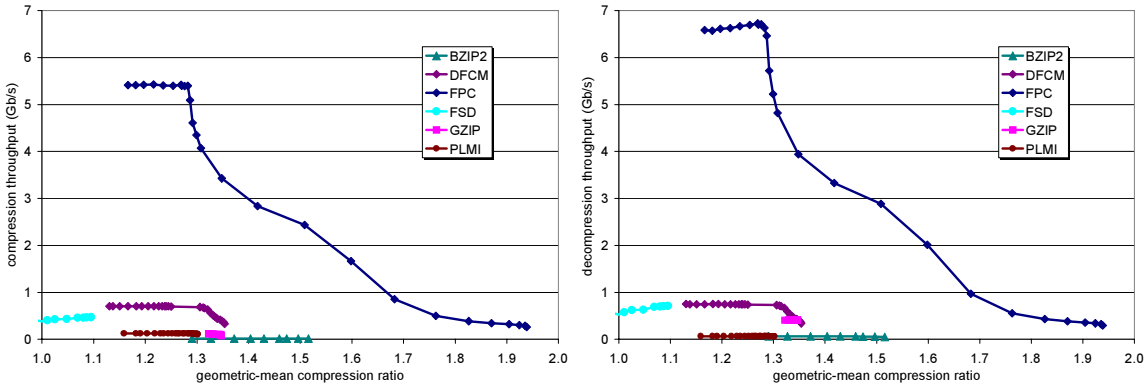

Figure 3: Compression (left) and decompression (right) throughput vs. compression ratio

For a given compression ratio, FPC compresses our datasets 8 to 300 times faster and decompresses them 9 to 100 times faster than the other algorithms. DFCM has the next highest throughput though GZIP's decompression throughput is similar. FSD is third, but it delivers the lowest compression ratios on our datasets. PLMI compresses the datasets faster than GZIP but decompresses them more slowly. BZIP2 is the slowest algorithm but reaches the second highest

compression ratio. All algorithms except our implementation of PLMI decompress faster than they compress. FPC compresses at up to 5.43 Gb/s and decompresses at up to 6.73 Gb/s. More results for FPC are available elsewhere [4].

pFPC Throughput

This subsection studies the compression and decompression throughput of pFPC. Figure 4 plots the throughput in megabytes per second versus the chunk size with a small and a large predictor on a four-core Xeon system. The left panel shows the compression throughput and the right panel the decompression throughput.

We see that the large predictor yields a lower throughput than the small predictor. This is because the small predictor's tables fit in the L1 data cache. Therefore, accessing them almost always results in L1 hits, which are fast and do not increase the memory traffic beyond the L1 cache. The large predictor's tables are 256 times as large. Accessing them causes many slow L1 misses and increases the memory bandwidth demand. We further observe that compression is slower than decompression as it requires more operations.

Small chunk sizes yield the lowest throughputs, especially for decompression. There are two reasons for this behavior. First, the inner loop in the compressor and decompressor iterates over the elements of a single chunk and the outer loop iterates over the chunks. For smaller chunk sizes, the outer loop has to perform more iterations, each of which involves running the inner loop. This results in higher looping overhead compared to large chunks, which only require a few iterations of the outer loop. Hence, larger chunks perform better, even with a single thread. Second, for chunk sizes up to and including 4, multiple threads access the same cache line, which may cause false sharing and coherence traffic. Decompression suffers more because in the decompressor the output buffer, which is written by the threads, is shared, whereas in the compressor only the input buffer, which is merely read by the threads, is shared. We notice another jump in performance at a chunk size of 512. 512 doubles require 4 kB of storage, which is the page size of the Xeon system. Hence, we believe this jump is due to improved TLB performance. At this threshold, the number of pages touched by each thread in the multi-threaded runs is halved, which reduces the pressure on the TLB.

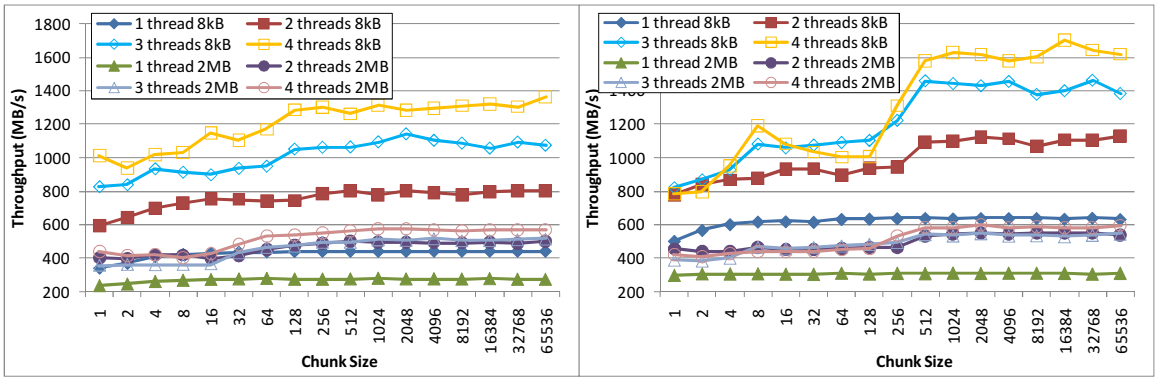

Figure 4: Harmonic-mean compression (left) and decompression (right) throughput versus the chunk size for two predictor sizes and various thread counts on a four-core Xeon system

Finally, we find that the throughput increases with the number of threads. Clearly, parallelization helps, especially for larger chunk sizes. For both predictor sizes, the average speedup with two threads is about 1.8 for compression and 1.75 for decompression. The corresponding spee-

dups with the small predictor are 2.5 and 2.25 for three threads and 3.0 and 2.6 for four threads. With the large predictor, the speedups are 1.85 and 1.75 for three threads and 2.05 and 1.9 for four threads. Clearly, the throughput scales sublinearly, and scaling is worse for decompression. There are two factors that limit scaling, the load balance and the memory bandwidth. Because the CPUs in our systems share the main memory, the memory bandwidth demand increases with the thread count and limits how many parallel threads can usefully be employed. Moreover, there are three sources of load imbalance: an unequal number of chunks per thread, a partial last chunk, and a variable processing speed. Because some predictions cause cache misses in the predictor tables, some doubles take longer to (de)compress than others.

To quantify the load imbalance, we wrote a load-balanced version of pFPC in which the threads grab chunks whenever they are ready for the next chunk. Because threads may grab chunks out-of-order, the predictor has to be reinitialized before each chunk, which takes time and lowers the compression ratio. Hence, this approach only works for large chunk sizes and small predictors that require infrequent zeroing out. The highest throughputs measured with this version of pFPC are 10% higher, but only for compression with the small predictor. There is almost no difference for decompression or for compression with the large predictor. Overall, the load imbalance in pFPC does not seem high and mostly affects compression.

To quantify the memory bandwidth, we wrote yet another version of pFPC that does not perform any compression but simply copies the values from the input buffer to the output buffer. This version does not cause any memory traffic for accessing predictors. It yields a throughput of 2.05 GB/s. With the small predictor, the "compressing" version of pFPC reaches a throughput of 1.7 GB/s, which is reasonably close to the maximum the memory subsystem can deliver, especially when accounting for the extra memory accesses to the predictor tables. Hence, the memory bandwidth fundamentally limits the scaling of pFPC. Nevertheless, pFPC reaches 10.9 and 13.6 Gb/s throughput with a compression ratio of 1.18 on a 4-core 3 GHz Xeon and is therefore fast enough to feed a 10 Gb/s network link in real time. More results are available elsewhere [5].

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*. 1995.

[2] M. Burrows and D. J. Wheeler. "A Block-Sorting Lossless Data Compression Algorithm." *Digital SRC Research Report 124*. May 1994.

[3] M. Burtscher and P. Ratanaworabhan. "High Throughput Compression of Double-Precision Floating-Point Data." *Data Compression Conference*, pp. 293-302. 2007.

[4] M. Burtscher and P. Ratanaworabhan. "FPC: A High-Speed Compressor for Double-Precision Floating-Point Data." *IEEE Transactions on Computers*, Vol. 58, No. 1, pp. 18-31. 2009.

[5] M. Burtscher and P. Ratanaworabhan. "pFPC: A Parallel Compressor for Floating-Point Data." *Data Compression Conference*, pp. 43-52. 2009.

[6] V. Engelson, D. Fritzson and P. Fritzson. "Lossless Compression of High-Volume Numerical Data from Simulations." *Data Compression Conference*, pp. 574-586. 2000.

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

[8] http://www.bzip.org/, 2009.

[9] http://www.gzip.org/, 2009.

[10] https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/, 2009.

[11] P. Lindstrom and M. Isenburg. "Fast and Efficient Compression of Floating-Point Data." *IEEE Transactions on Visualization and Computer Graphics*, Vol. 12, No. 5. 2006.

[12] P. Ratanaworabhan, J. Ke and M. Burtscher. "Fast Lossless Compression of Scientific Floating-Point Data." *Data Compression Conference*, pp. 133-142. 2006.

[13] Y. Sazeides and J. E. Smith. "The Predictability of Data Values." *30th International Symposium on Microarchitecture*, pp. 248-258. December 1997.

[14] J. Ziv and A. Lempel. "A Universal Algorithm for Data Compression." *IEEE Transaction on Information Theory*, Vol. 23, No. 3, pp. 337-343. May 1977.