pFPC: A Parallel Compressor for Floating-Point Data

Martin Burtscher
The University of Texas at Austin
burtscher@ices.utexas.edu

Paruj Ratanaworabhan
Cornell University
paruj@csl.cornell.edu

Abstract

This paper describes and evaluates pFPC, a parallel implementation of the lossless FPC compression algorithm for 64-bit floating-point data. pFPC can trade off compression ratio for throughput. For example, on a 4-core 3 GHz Xeon system, it compresses our nine datasets by 18% at a throughput of 1.36 gigabytes per second and by 41% at a throughput of 570 megabytes per second. Decompression is even faster. Our experiments show that the thread count should match or be a small multiple of the data's dimensionality to maximize the compression ratio and the chunk size should be at least equal to the system's page size to maximize the throughput.

1. Introduction

FPC is a lossless compression algorithm for double-precision floating-point data [3, 4]. It has been designed to compress and decompress numeric data from scientific environments with an emphasis on throughput. Possible application domains include compressing program output and checkpoints before storing them to disk as well as compressing data that need to be transferred between compute nodes or to another machine.

The computers used for data-intensive scientific calculations often comprise a number of nodes. Each node typically contains multiple shared-memory processors but only one communication link to access the network or disk subsystem. To maximize the throughput of this link, we need an algorithm whose operation can be overlapped with the link access and whose speed is high enough to compress and decompress the data to be transferred in real time. For example, 10-gigabit-per-second links, which are common in high-end systems, require a compressor with a throughput of over one gigabyte per second.

This paper studies and evaluates different approaches to parallelize the FPC algorithm to reach such throughputs. In particular, it investigates how to best divide the work among the processors, i.e., which data elements should be processed by which core. The results show, for example, that small and large chunks of data yield better compression ratios than medium-sized chunks. However, small chunks can result in poor cache and TLB performance and therefore a low throughput. Multidimensional datasets often prefer a number of parallel threads that is a small multiple of the dimensionality. Finally, scaling of our parallel algorithm, called pFPC, is limited by the memory bandwidth. On our four-core Xeon system, it compresses our nine scientific numeric datasets by 18% at a speed of 1.36 gigabytes per second and decompresses them at a speed of 1.7 gigabytes per second. With a larger and therefore more accurate but slower predictor, pFPC compresses the datasets by 41% at 570 megabytes per second and decompresses them at 600 megabytes per second. These results are harmonic means over the nine datasets.

The rest of this paper is organized as follows. Section 2 provides an overview of FPC and describes our parallelization approach. Section 3 summarizes related work. Section 4 presents the evaluation methods. Section 5 discusses the results. Section 6 concludes the paper with a summary of our findings.

2. The FPC Algorithm

2.1 Sequential FPC

FPC [4] 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* [19] and a *dfcm* [8] value predictor to predict the doubles. Both predictors are fast hash tables. The more accurate of the two predictions, i.e., the one that shares more common most significant bytes 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 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.

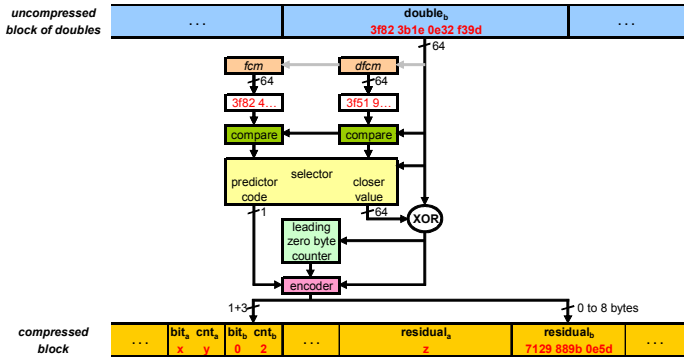

Figure 1: The sequential FPC compression algorithm

Decompression works as follows. It 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 *fcm* or *dfcm* prediction to recreate the original double.

For performance reasons, FPC interprets all doubles as 64-bit integers and exclusively uses integer arithmetic. Both predictor tables are initialized with zeros. After each prediction, they are updated with the true double value. They generate the same sequence of predictions during compression as they do during decompression. The predictors work by performing a hash-table lookup to determine which value followed the last time a similar sequence of preceding values, called the history, was seen. The *fcm* predictor uses the actual values for this purpose whereas the *dfcm* predictor uses the differences between consecutive values. The exact operation of the two predictors is described elsewhere [4].

2.2 Parallel FPC

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 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 de-

termined by the programmer, making this the less flexible of the two approaches. Note that the two parallelization approaches are independent and can be combined.

Because FPC is memory and not compute bound (Section 5.2), 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. As we did not consider hardware support [24], the rest of this paper focuses 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.

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/.

3. Related Work

Lossy parallel compression is of great interest to the image and video processing community. A large amount of work exists in this area, e.g., Shen et al. [20] and Okumura et al. [17]. Our work targets the scientific community where lossless compression is often a requirement. There is some related work on parallel lossless compression, but none about parallel floating-point compression, which is the focus of this paper.

Smith and Storer [9] proposed two parallel algorithms for data compression based on textual substitution. These algorithms target systolic array architectures. One algorithm uses a static dictionary and the other a sliding dictionary. Stauffer and Hirschberg [22] proposed another parallel static dictionary algorithm that uses longest fragment first and greedy parsing strategies. Their algorithm is geared towards a theoretical Parallel Random Access Memory (PRAM) machine. Agostino [1] parallelized a 2D compression algorithm that is based on a sequential version by Storer [23]. He proposed a work-optimal parallel algorithm using a rectangular greedy matching technique that also targets PRAM.

Stassen and Tjalkens [21] implemented a parallel version of the Context-Tree Weighting (CTW) algorithm. They parallelized the compressor by splitting the CTW tree into subtrees and assigning a group of subtrees to each processor. Klien and Wiseman [15] show how the Lempel-Ziv compression algorithm can be parallelized. They partition the input file into n blocks (or chunks), one per processor, and use a hierarchical tree structure to synchronize the processing of the n blocks. Their experiments with LZSS and LZW yielded a speedup of two on a four-processor machine. Franaszek et al. [6] proposed another LZSS parallel compression scheme where the parallel compressors operate on partitioned data but jointly construct a global dictionary. Howard and Vitter [10] describe parallel algorithms for Huffman and arithmetic coding. These algorithms use the PRAM computational model and are optimized for the Connection Machine Architecture.

Gilchrist [7] parallelized BZIP2 by targeting the Burrows Wheeler Transform in the

compressor, realizing that it can benefit from parallel sorting and the concurrent processing of multiple blocks. His experiments show scaling up to 22 processors with close to linear speedup on a 1.83 gigabyte dataset of an elliptic curve. GZIP has also been parallelized [14]. Note that both of these algorithms are computation bound and therefore scale better than our memory-bound compressor. Moreover, our previous results [4] show that the sequential FPC implementation is more than 22 times faster than BZIP2.

4. Evaluation Methodology

4.1 Systems and Compilers

We compiled and evaluated pFPC on the following three 64-bit shared-memory systems.

• The **Itanium** system has 3 GB of memory and two 1.6 GHz Itanium 2 CPUs, each with a 16 kB L1 data cache, a 256 kB unified L2 cache, and a 3 MB L3 cache. The operating system is Red Hat Enterprise Linux AS4. We used the Intel C Itanium Compiler version 9.1 with the "-O3 -mcpu=itanium2 -restrict -static -pthread" compiler flags.

• The **Opteron** system has 2 GB of memory and a dual-core 1.8 GHz Opteron processor. Each core has a 64 kB L1 data cache and a 1 MB unified L2 cache. The operating system is Red Hat Linux 2.6.23.17 and the compiler is gcc version 4.1.2. The compiler flags on this system are "-O3 -march=opteron -static -pthread -std=c99".

• The **Xeon** system has 4 GB of memory and 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 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.

4.2 Measurements

All timing measurements are performed by instrumenting the source code, i.e., by adding code to read the timer before and after the measured code section and printing the difference. We measure the runtime of the compression and decompression code only, excluding the time it takes to preload the input buffers with data. Each experiment was conducted three times in a row and the shortest runtime is reported. Averaged compression ratios and throughput numbers refer to the harmonic mean throughout this paper. We used two predictor configurations, the smaller of which requires a total of 8 kilobytes of table space and the larger of which requires 2 megabytes of table space. Note that all predictor tables and buffers are aligned to page boundaries.

4.3 Datasets

We used the following nine[1] datasets from three scientific domains for our evaluation.

Parallel messages: These datasets capture the messages sent by a node in a parallel system running NAS Parallel Benchmark (NPB) [2] and ASCI Purple [13] applications.

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

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_plasma*: simulated plasma temperature of a wire array z-pinch experiment

[1] To improve the presentation, we removed the 4 least interesting datasets that we had used in our previous studies [3, 4].

Observational data: These datasets comprise measurements from scientific instruments.

- *obs_error*: data values specifying brightness temperature errors of a weather satellite
- *obs_info*: latitude and longitude of the observation points of a weather satellite
- *obs_spitzer*: data from the Spitzer Space Telescope

Table 1 summarizes information about each dataset. The first two data columns list the size in megabytes and in millions of double-precision floating-point values. The middle column shows the percentage of values in each dataset that are unique. The fourth column displays the first-order entropy of the doubles in bits. The last column expresses the randomness of the datasets in percent, i.e., it reflects how close the first-order entropy is to that of a truly random dataset with the same number of unique 64-bit values.

Table 1: Statistical information about the datasets

	size (megabytes)	doubles (millions)	unique values (percent)	first order entropy (bits)	randomness (percent)
msg_bt	254.0	33.30	92.9	23.67	95.1
msg_sp	276.7	36.26	98.9	25.03	99.7
msg_sweep3d	119.9	15.72	89.8	23.41	98.6
num_brain	135.3	17.73	94.9	23.97	99.9
num_comet	102.4	13.42	88.9	22.04	93.8
num_plasma	33.5	4.39	0.3	13.65	99.4
obs_error	59.3	7.77	18.0	17.80	87.2
obs_info	18.1	2.37	23.9	18.07	94.5
obs_spitzer	189.0	24.77	5.7	17.36	85.0

5. Results

5.1 Compression Ratio

We first study the effect of the thread count and chunk size on the compression ratio. These results are architecture independent. Figure 2 shows the compression ratio of pFPC on the nine datasets for 1 through 16 threads with a chunk size of 1 double and a predictor size of 2 megabytes per thread. The results for the small predictor size are not shown because they exhibit the same trends.

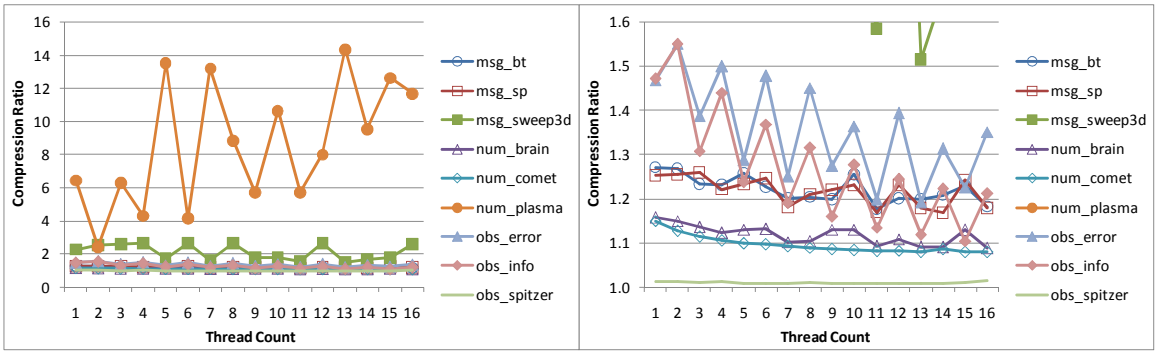

Figure 2: Compression ratio for different thread counts with a chunk size of 1 double and a predictor size of 2 megabytes (the right panel is a zoomed in version of the left panel)

We find that pFPC is ineffective on *obs_spitzer*. It compresses *msg_sweep3d* and especially *num_plasma* quite well and the remaining six datasets by 15% to 55%. While these compression ratios are not high, they are in the expected range for lossless compression of somewhat random data. In previous work [3, 4], we have shown that the single-threaded FPC algorithm compresses five of the nine datasets better than BZIP2 [11],

DFCM [18], FSD [5], GZIP [12], and PLMI [16] and also delivers a higher harmonic-mean compression ratio on these datasets. DFCM works best on *msg_bt* (1.35), PLMI on *num_brain* (1.24) and *num_comet* (1.26), and BZIP2 on *obs_spitzer* (1.75).

Increasing the number of threads results in interesting fluctuations in the compression ratios. More pronounced examples include *obs_error* and *obs_info*, both of which compress best with two threads and generally compress better with even numbers of threads than with odd numbers. We believe this indicates that they are 2D datasets where the data from the two dimensions are interleaved. Thus, with two threads, each thread only observes values from one dimension. Since the values within a dimension correlate more than between dimensions, two threads yield a better compression ratio than one thread does. In fact, for any even number of threads each thread will only see data from one dimension, which explains the better compression ratios. The compression ratios drop for higher (even) thread counts because the threads do not get to see every value. As this gap grows, important history is lost and the compressor does not perform as well anymore.

Interestingly, the remaining datasets follow similar patterns. *msg_sp* exhibits small peaks at thread counts that are multiples of three. This dataset appears to be 3D. *obs_spitzer* has small peaks that hint at a 4D dataset. *msg_bt* and, less clearly, *num_brain* may be 5D datasets. *num_comet* reaches its highest compression ratio with 42 threads and has another peak at 84 threads (not shown). Hence, it looks like a 42D dataset. The last two datasets seem to be combinations of different dimensionalities, i.e., they probably contain data records of different sizes; *msg_sweep3d* appears to be a combination of 4D and 6D records, and *num_plasma* has peaks at multiples of 5, 7, and 13.

Some datasets exhibit significantly more correlation between the values within a dimension than across the dimensions. As a consequence, the compression ratios differ substantially for different thread counts. For other datasets, the correlation between the dimensions is almost the same as within a dimension, which is why the fluctuations are small. In fact, two of the nine datasets (*msg_bt* and *num_brain*) compress best with just one thread and should, for practical purposes, be considered 1D datasets. Nevertheless, we conclude that for *n*-dimensional data, a thread count of *n* is probably a good choice, as are thread counts that are small integer multiples of *n*.

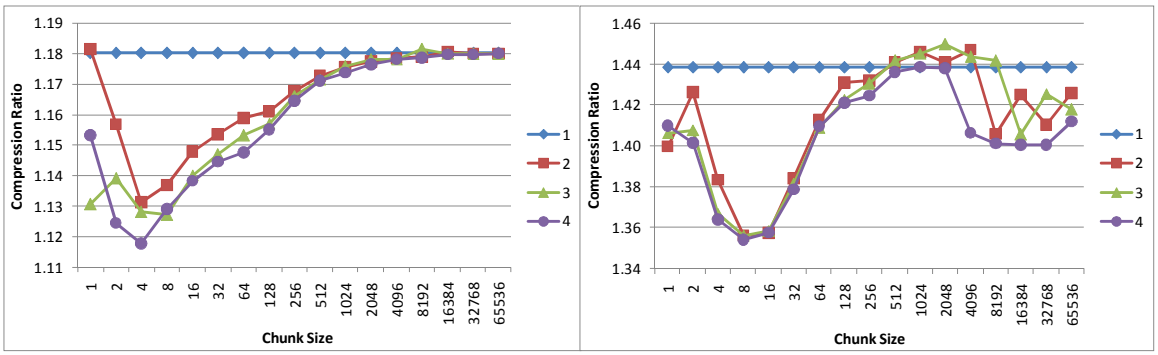

Figure 3: Harmonic-mean compression ratio for different chunk sizes and 1, 2, 3 and 4 threads with predictor sizes of 8 kilobytes (left panel) and 2 megabytes (right panel)

Thus far, we have only investigated chunk sizes of one double. Figure 3 shows the effect of increasing the chunk size on the harmonic-mean compression ratio for up to four threads with the small and large predictors. The chunk size does not affect the compres-

sion ratio when one thread is used. With larger thread counts, each thread processes some but not all of the data (Section 2.2).

Figure 3 reveals that very small chunks result in reasonable compression ratios, medium-sized chunks perform poorly, and large chunks are best, but only up to a limit. The reason for this behavior is twofold. On the one hand, due to their low dimensionality, most of our datasets favor very small chunks. As the chunks get larger, data from multiple dimensions are compressed by the same thread, and the compression ratio decreases. On the other hand, small chunks result in frequent small gaps in the data that each thread observes. After all, with small chunks, a thread sees a few data values, then a gap, then another few data values, then another gap, and so on. This makes it hard to build a meaningful history in the predictors, and the prediction accuracy suffers. Larger chunks result in larger gaps, but each thread observes a long sequence of consecutive values (i.e., the chunk size) before encountering a gap, which is better for the history-based predictors in pFPC. Hence, the compression ratio increases with the chunk size.

In some instances, multiple threads result in better compression ratios than a single thread. This is because each thread uses the same predictor size, so for higher thread counts the cumulative predictor size is larger, which can boost the compression ratio.

For the large predictor, the drop for large chunk sizes in the harmonic-mean compression ratio is mostly due to the two best compressing datasets, *msg_sweep3d* and *num_plasma*. Just like these two datasets show the largest fluctuations in Figure 2, they also fluctuate the most when changing the chunk size (not shown). Coincidentally, both of these datasets perform relatively poorly with the largest chunk sizes we investigated and thus lower the harmonic mean. The small predictor, however, does not compress these two datasets significantly better than the other datasets. Consequently, their behavior only has a small impact on the mean compression ratio. We conclude that large chunk sizes typically yield the best compression ratios for our history-based compressor.

5.2 Throughput

This subsection studies the compression and decompression throughput of pFPC (i.e., the raw dataset size divided by the runtime). These results are hardware dependent. Figure 4 plots the throughput in megabytes per second versus the chunk size with the small and large predictors. The left panels show the compression throughput and the right panels the decompression throughput. The top panels show the results from the Itanium system, the middle panels from the Opteron system, and the bottom panels from the Xeon system.

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.

On all systems, 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. In some cases, we notice another jump in performance at a chunk size of 512 or 2048. 512 doubles require 4 kB of storage, which is the page size of the Opteron and Xeon systems. 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. Note that this transition happens at a four times larger chunk size on the Itanium because the page size on this system is four times larger. We conclude that the chunk size should be at least equal to the page size (and aligned to a page boundary, which also eliminates the potential for false sharing), to maximize the throughput.

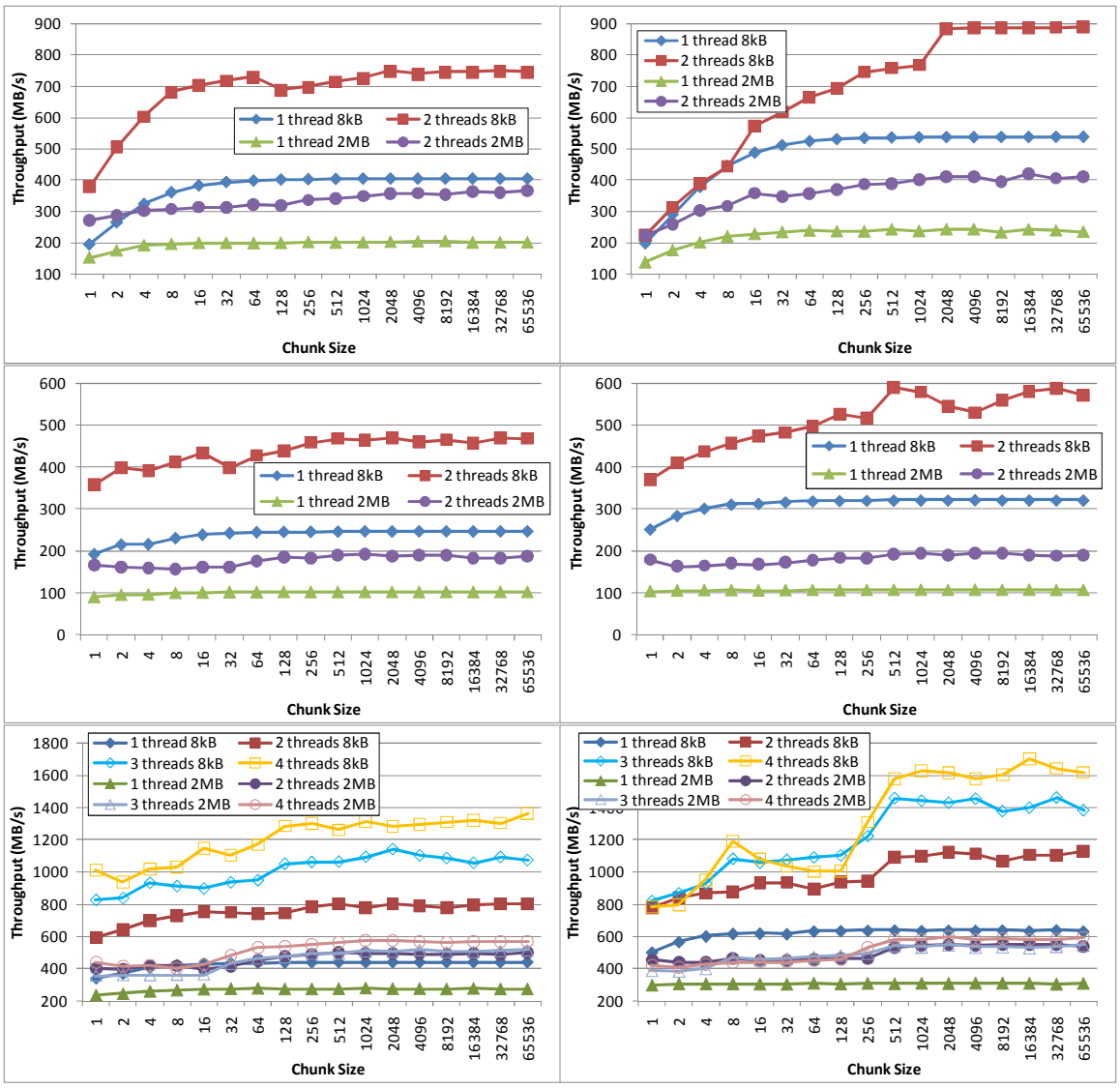

Figure 4: Harmonic-mean compression (left) and decompression (right) throughput versus the chunk size for the two predictor sizes and various thread counts on the Itanium (top), Opteron (middle), and Xeon (bottom) systems; some y-axes are not zero based

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 speedups with two threads are about 1.85 (compression) and 1.75 (decompression) on the Itanium, 1.9 and 1.8 on the Opteron, and 1.8 and 1.75 on the Xeon. On the Xeon, the speedups 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 (Section 2.2), a partial last chunk (Section 2.2), 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 that require infrequent zeroing out of the predictor tables. The highest throughputs measured with this version of pFPC are 10% higher on the x86 machines and 30% higher on the Itanium, but only for compression with the small predictor. There is almost no difference for decompression or for compression with the large predictor, except on the Itanium, where compression with the large predictor is about 10% faster. 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 throughputs of 1.4 gigabytes per second on the Itanium, 1.1 GB/s on the Opteron, and 2.05 GB/s on the Xeon. With the small predictor, the "compressing" version of pFPC reaches throughputs of 0.9 GB/s on the Itanium, 0.6 GB/s on the Opteron, and 1.7 GB/s on the Xeon, which are 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.

6. Summary and Conclusions

This paper presents a parallel implementation of the lossless FPC compression algorithm for double-precision floating-point values. pFPC works by breaking up the data into chunks and distributing them among the threads, which led to the following observations.

- For *n*-dimensional data, a thread count of *n* or a small multiple thereof is usually a good choice as far as the compression ratio is concerned.
- The chunks should be at least as large as the page size of the underlying system (and aligned to a page boundary) for throughput reasons.
- Large chunk sizes result in the best compression ratio for our history-based predictors.
- Scaling is limited by the system's memory bandwidth (and the load imbalance).

While the last two observations are specific to history-based and high-speed compression algorithms, respectively, we believe the first two observations are general and apply to other compressors as well. We conclude that future improvements to pFPC should fo-

cus on increasing the compression ratio at the cost of more computation but not at the cost of additional memory traffic.

7. Acknowledgments

The authors and this project are supported by DOE award DE-FG02-06ER25722. Martin Burtscher is further supported by NSF grants 0833162, 0719966, 0702353, 0615240, 0541193 as well as grants and gifts from IBM, Intel, and Microsoft.

8. References

[1] S. Agostino. "A Work-Optimal Parallel Implementation of Lossless Image Compression by Block Matching." *Nordic Journal of Computing*, Vol. 10, No. 1, pp. 13-20. 2003.

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

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

[6] P. Franaszek, J. Robinson and J. Thomas. "Parallel Compression with Cooperative Dictionary Construction." *Data Compression Conference*, p. 200. 1996.

[7] J. Gilchrist. "Parallel Data Compression with BZIP2." *IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 559-564. 2004.

[8] 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.

[9] M. Gonzalez and J. Storer. "Parallel Algorithms for Data Compression." *Journal of the ACM*, Vol. 32, No. 2, pp. 344-373. 1985.

[10] P. Howard and J. Vitter. "Parallel Lossless Image Compression using Huffman and Arithmetic Coding." *Information Processing Letters*, Vol. 59, No. 2, pp. 65-73. 1996.

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

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

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

[14] http://zlib.net/pigz/, 2009.

[15] S. Klien and Y. Wiseman. "Parallel Lempel Ziv Coding." *Discrete Applied Mathematics*, Vol. 146, No. 2, pp. 180-191. 2005.

[16] P. Lindstrom and M. Isenburg. "Fast and Efficient Compression of Floating-Point Data." *IEEE Transactions on Visualization and Computer Graphics*, Vol. 12, No. 5, pp. 1245-1250. 2006.

[17] Y. Okumura, K. Irie and R. Kishimoto. "Multiprocessor DSP with Multistage Switching Network for Video Coding." *IEEE Transactions on Communications*, Vol. 39, No. 6, pp. 938-946. 1991.

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

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

[20] K. Shen and E. J. Delp. "A Parallel Implementation of an MPEG Encoder: Faster than Real-time!" *SPIE Conference on Digital Video Compression: Algorithms and Technologies*, pp. 407-418. 1995.

[21] M. Stassen and T. Tjalkens. "A Parallel Implementation of the CTW Compression Algorithm." *22nd Symposium on Information Theory in the Benelux*, pp. 85-92. 2001.

[22] L. Stauffer and D. Hirschberg. "Dictionary Compression on the PRAM." *UC Irvine TR 94-7*. 1994.

[23] J. Storer. "Lossless Image Compression Using Generalized LZ1-Type Methods." *Data Compression Conference*, p. 290. 1996.

[24] Y. Wiseman. "A Pipeline Chip for Quasi Arithmetic Coding." *IEICE Journal - Trans. Fundamentals*, Vol. E84-A, No. 4, pp. 1034-1041. 2001.