— Parallel Data Compression with BZIP2 —

Jeff Gilchrist
Systems and Computer Engineering
Carleton University
Ottawa, Canada K1S 5B6
jeff@gilchrist.ca
http://gilchrist.ca/jeff/

December 16, 2003

Abstract

A parallel implementation of the bzip2 block-sorting lossless compression program is described. The performance of the parallel implementation is compared to the sequential bzip2 program running on various shared memory parallel architectures.

The parallel bzip2 algorithm works by taking the blocks of input data and running them through the Burrows-Wheeler Transform (BWT) simultaneously on multiple processors using pthreads. The output of the algorithm is fully compatible with the sequential version of bzip2 which is in wide use today.

The results show that a significant, near-linear speedup is achieved by using the parallel bzip2 program on systems with multiple processors. This will greatly reduce the time it takes to compress large amounts of data while remaining fully compatible with the sequential version of bzip2.

1 Introduction

Parallel computing allows software to run faster by using multiple processors to perform several computations simultaneously whenever possible. The majority of general purpose software is designed for sequential machines and does not take advantage of parallel computing. Lossless data compression software like bzip2, gzip, and zip are designed for sequential machines. The data compression algorithms require a great deal of processing power to analyze and then encode data into a smaller form. Creating a general purpose compressor that can take advantage of parallel computing should greatly reduce the amount of time it requires to compress files, especially large ones.

This paper explores several ways of implementing a parallel version of the Burrows-Wheeler Transform (BWT) and describes one implementation that modifies the popular bzip2 compression program. The limited research papers available on parallel data compression use theoretical algorithms designed to run on impractical parallel machines that do not exist or are very rare in the real world. The purpose of a parallel bzip2 compression program is to make a practical parallel compression utility that works with real machines. The bzip2 program was chosen because it achieves very good compression and is available in library and source form, free under an open source license.

One method of parallelizing the BWT algorithm has been implemented and tested on several shared memory parallel machines such as dual processor Intel Pentium3 and AMD

Athlon MP systems, a Pentium4 Hyperthreaded system, and a 24 processor SunFire 6800 system. The results show that near-linear speedup is achieved on the multiple-processor machines and a small speedup is achieved on the Hyperthreaded machine.

The remainder of the paper is organized as follows. In Section 2, the relevant literature on data compression is reviewed. Section 3, shows several possible ways to parallelize the BWT algorithm. Section 4, presents the parallel bzip2 algorithm. Section 5 gives the performance results using the algorithm on several parallel architectures. Section 6 concludes the paper.

2 Literature Review

Lossless data compression is used to compact files or data into a smaller form. It is often used to package up software before it is sent over the Internet or downloaded from a web site to reduce the amount of time and bandwidth required to transmit the data. Lossless data compression has the constraint that when data is uncompressed, it must be identical to the original data that was compressed. Graphics, audio, and video compression such as JPEG, MP3, and MPEG on the other hand use lossy compression schemes which throw away some of the original data to compress the files even further. We will be focusing only on the lossless kind. There are generally two classes of lossless compressors: dictionary compressors and statistical compressors. Dictionary compressors (such as Lempel-Ziv based algorithms) build dictionaries of strings and replace entire groups of symbols. The statistical compressors develop models of the statistics of the input data and use those models to control the final output [6].

2.1 Dictionary Algorithms

In 1977, Jacob Ziv and Abraham Lempel created the first popular universal compression algorithm for data when no a priori knowledge of the source was available. The LZ77 algorithm (and variants) are still used in many popular compression programs today such as ZIP and GZIP. The compression algorithm is a dictionary based compressor that consists of a rule for parsing strings of symbols from a finite alphabet into substrings whose lengths do not exceed a predetermined size, and a coding scheme which maps these substrings into decipherable code-words of fixed length over the same alphabet [13]. This was a big improvement for people at the time that didn't have much other than Shannon and Huffman coding to use for compression.

Many variants and improvements to the LZ algorithm were proposed and implemented since its initial release. One such improvement is LZW (Lempel-Ziv-Welch) which is an adaptive technique. As the algorithm runs, a dictionary of the strings which have appeared is updated and maintained. The dictionary is pre-loaded with the 256 possible bit sequences that can appear in a byte. For example, if "fire" and "man" are two strings in the dictionary then the sequence of "fireman" in the data would be converted into the index of "fire" followed by the index of "man" in the dictionary. The algorithm is adaptive because it will add new strings to the dictionary. Once the dictionary has filled up after using all its space, the algorithm becomes non-adaptive only using the codes already found in the dictionary, unable to add any new ones. Uncompression with LZW is faster that compression since string searching in the dictionary is not necessary [8].

Some investigation into creating parallel dictionary compression has been performed. Once the dictionary has been created in an LZ based algorithm, a greedy parsing is per-

formed to determine which substrings will be replaced by pointers into the dictionary. The longest match step of the greedy parsing algorithm can be executed in parallel on a number of processors. For a dictionary of size N, $2N$-1 processors configured as a binary tree can be used to find the longest match in $O(\log_N)$ time. Each leaf processor performs comparisons for a different dictionary entry. The remaining N-1 processors coordinate the results along the tree in $O(\log_N)$ time. Another option is to break the input data into blocks and compress each block on a different processor [11]. Compressors generally work better with larger blocks of data so while overall speed may be increased by splitting up the input data into blocks, the compression ratio may be decreased at the same time. A careful balance would have to be selected between the two. Stauffer a year later expands on previous parallel dictionary compression schemes on the PRAM by using a parallel longest fragment first (LFF) algorithm to parse the input instead of the greedy algorithm [12]. The LFF algorithm parses the input by continuously locating the longest substring in the uncoded section of the input data which matches a dictionary entry. It then replaces the substring with the corresponding dictionary reference. The LFF algorithm in general performs better than the greedy algorithm but is not often used in sequential implementations because it requires two passes over the input. With the PRAM, using the LFF parsing can be performed over the entire input in one step. They assume a static dictionary which is stored as a suffix tree, and that dictionary references are of a fixed length. A processor is assigned to each position of the input string which then computes the list of lengths of matches between the dictionary and the input beginning at its assigned position. With a maximum dictionary length of M, the LFF parsing can be done in $O(M(\log_M + \log_n))$ time with $O(n)$ processors. They later refine the algorithm to improve the number of processors required to $O(\frac{n}{\log_n})$. In practice, this algorithm would never work since very few people if anyone would have a machine with enough processors to satisfy the requirements.

2.2 Statistical Algorithms

Statistical compressors traditionally combine a modeling stage, followed by a coding stage. The model is constructed from the known input and used to facilitate efficient compression in the coder. Good compressors will use a multiple of three basic modeling techniques. Symbol frequency associates expected frequencies with the possible symbols allowing the coder to use shorter codes for the more frequent symbols. Symbol context has the dependency between adjacent symbols of data usually expressed as a Markov model. This gives the probability of a specific symbol occurring being expressed as a function of the previous n symbols. Symbol ranking occurs when a symbol "predictor" chooses a probable next symbol, which may be accepted or rejected [6].

Arithmetic coding is a statistical compression technique that uses estimates of the probabilities of events to assign code words. Ideally, short code words are assigned to more probable events and longer code words are assigned to less probable events. Theoretically, arithmetic codes assign one "code word" to each possible data set. The arithmetic coder must work together with a modeler that estimates the probabilities of the events in the coding. To obtain good compression, a good probability model and efficient way of representing the probability model are required. The models can be adaptive, semi-adaptive, or non-adaptive. Adaptive models dynamically estimate the probably of each event based on preceding events. Semi-adaptive models use a preliminary pass of the data to gather some statistics, and non-adaptive models use fixed probabilities for all data. An advantage of arithmetic coding is the separation of coding and modeling since it allows the complexity

of the modeler to change without having to modify the coder. The disadvantage is that is runs more slowly and is more complex to implement than LZ based algorithms [9].

2.3 Burrows-Wheeler Transform

The Burrows-Wheeler transform [1] is a block-sorting, lossless data compression algorithm that works by applying a reversible transformation to a block of input data. The transform does not perform any compression but modifies the data in a way to make it easy to compress with a secondary algorithm such as "move-to-front" coding and then Huffman, or arithmetic coding. The BWT algorithm achieves compression performance within a few percent of statistical compressors but at speeds comparable to the LZ based algorithms. The BWT algorithm does not process data sequentially but takes blocks of data as a single unit. The transformed block contains the same characters as the original block but in a form that is easy to compress by simple algorithms. Same characters in the original block are often grouped together in the transformed block.

The algorithm works by transforming a string S of N characters by forming the N rotations (cyclic shifts) of S, sorting them lexicographically, and extracting the last character of each of the rotations. A string L is then formed from these extracted characters, where the ith character of L is the last character of the ith sorted rotation. The algorithm also computes the index I of the original string S in the sorted list of rotations. With only L and I there is an efficient algorithm to compute the original string S when undoing the transformation for decompression. They also explain that to achieve good compression, a block size of sufficient value must be chosen, at least 2 kilobytes. Increasing the block size also increases the effectiveness of the algorithm at least up to sizes of several megabytes.

The BWT can be seen as a sequence of three stages: the initial sorting stage which permutes the input text so similar contexts are grouped together, the Move-To-Front stage which converts the local symbol groups into a single global structure, and the final compression stage which takes advantage of the transformed data to produce efficient compressed output [6]. Present implementations of BWT require about 9 bytes of memory for each byte of data, plus a constant 700 kilobytes. For example, to compress 1 megabyte of data would require about 10 megabytes of memory using this algorithm.

Before the data in BWT is compressed, it is run through the Move-To-Front coder. The choice of MTF coder is important and can affect the compression rate of the BWT algorithm. The order of sorting determines which contexts are close to each other in the output and thus the sort order and ordering of source alphabet can be important. Many people consider the MTF coder to be fixed, but any reversible transformation can be used for that phase. The dependence of BWT on input alphabet encoding is a relatively unique characteristic for general lossless data compression algorithms. Algorithms such as PPM and LZ are based on pattern matching which is independent of source alphabet encoding. As a test, a sample image (LENA) was run through a randomly chosen alphabet permutation and the resulting compressed file using BWT was 15 percent larger than the original image compressed. Using different methods of ordering for the input alphabet can result in smaller compressed files for some data and larger compressed sizes for others so the ordering must be carefully selected and may not be appropriate for all data [2].

It was discovered that with sequences of length n taken from a finite-memory source that the performance of BWT based algorithms converge to the optimal performance at a rate of $O(\frac{\log_n}{n})$ surpassing that of LZ77 which is $O(\log_{\frac{\log_n}{logn}})$ and LZ78 which is $O(\frac{1}{\log_n})$ [5]. They also note that while many algorithms use sequential codes on the BWT output,

the overall data compression algorithms are non-sequential since the transform requires simultaneous access to all symbols of the data string (or blocks of the data string if the block size is smaller than the entire data). This is something that can be taken advantage of in a parallel implementation.

2.4 Prediction by Partial Match

The PPM (Prediction by Partial Match) algorithm is currently the best lossless data compression algorithm for textual data. It was first published in 1984 by Cleary and Witten [4]. A number of variations and improvements have been made since then. While the PPM algorithms have superior compression, other lossless algorithms such as LZ and BWT based algorithms are more commonly used since they require less resources than PPM and are usually much faster [7].

PPM is a finite-context statistical modeling technique which combines several fixed-order context models to predict the next character in the input sequence. The prediction probabilities for each context are adaptively updated from frequency counts. The maximum context length is a fixed constant and has been found that increasing the length beyond 6 generally does not improve compression. The basic premise of PPM is to use the previous bytes in the input stream to predict the following one. Models that make their predictions on several immediately preceding symbols are finite-context model of order k, where k is the amount of preceding symbols used. PPM uses several fixed-order context models with different values starting at 0 to some maximum value. From each model, a separate probability distribution is obtained which are effectively combined into a single one where arithmetic coding is used to encode the actual character relative to that distribution. Escape probabilities are used for this combination such that if a context cannot be used for encoding a value, an escape symbol is transmitted and the model with the next smaller value of k is used instead. Cleary's latest paper improves on his original PPM algorithm with PPM* which allows for unbounded length context with PPM [3]. Instead of imposing a fixed maximum upper bound on context length, the context length is allowed to vary depending on the coding situation. This method stores the model in such a way that rapid access to predictions can be achieved based on any context. The PPM* algorithm gives a 5.6 percent performance increase over the older PPM algorithm which places it ahead of the BWT algorithm.

2.5 BZIP2

The BZIP2 program compresses files using the Burrows-Wheeler block-sorting text compression algorithm, and Huffman coding. Compression is generally considerably better than that achieved by more conventional LZ77/LZ78-based compressors, and approaches the performance of the PPM family of statistical compressors while being much faster [10]. The first popular compression algorithms used for universal lossless data compression were the Lempel-Ziv (LZ) algorithms used in such programs as ZIP, GZIP, ARJ, and many others. Arithmetic coding was discovered later and had improved compression results. Even better still was the Burrows-Wheeler Transform (BWT) which achieved improved compression performance over previous algorithms at quick speeds. Finally, the best universal compression algorithms today are variants of the PPM (Prediction by Partial Matching) statistical algorithm which have superior compression performance but at the cost of speed and memory usage.

3 Parallelizing BWT

Three ways to parallelize the BWT algorithm in a shared memory architecture are considered.

3.1 Parallel Sort

Burrows and Wheeler explain that much of the time, the BWT algorithm is sorting [1]. The initial stage of the BWT algorithm creates a matrix from the input data which needs to be sorted lexographically. Implementing a parallel sorting algorithm to perform the lexicographical sort may increase the speed of the BWT on a parallel machine.

3.2 Multiple Blocks

The BWT algorithm in most implementations requires about 10 times the amount of memory as input data. In order to cope with large amounts of data, the BWT algorithm will naturally split the data into independent blocks of a predetermined size before the data is compressed. The blocks are then processed through the BWT algorithm. Since each block is independent, it should be possible to run the BWT algorithm on multiple blocks of data simultaneously and achieve speedup on a parallel machine. The separate blocks are then concatenated back together again to form the final compressed file. The sequential version of BWT will process the blocks in order so does not need to worry about order when writing the output to disk. A parallel implementation will need to keep track of block ordering and write the compressed blocks back to disk in the correct order.

3.3 Combination

Depending on the number of processors on the parallel machine, it may be useful to combine processing multiple blocks of data simultaneously with using a parallel implementation of the lexicographical sorting algorithm. Tests would have to be performed in order to determine if using a combination is better than either of the solutions alone.

4 Parallel BZIP2

The method of processing multiple blocks of data simultaneously with the BWT algorithm was implemented. The popular bzip2 program by Julian Seward uses the BWT algorithm for compression. It was chosen because it is available in library and source form, free under an open source license. This gives the benefit of not having to implement the BWT from scratch and allows the compressed files generated from the parallel version to be compatible. Anyone with the sequential version of bzip2 would be able to uncompress files created with the parallel version and vice-versa. The parallel version of bzip2 has been named pbzip2.

4.1 Sequential bzip2

The bzip2 program processes data in blocks ranging from 100,000 bytes to 900,000 bytes in size, depending on command line switches. The default block size is 900,000 bytes. It reads in data 5,000 bytes at a time, until enough data to fill a block is obtained. It will then process the block with the BWT algorithm and write the compressed output to disk. This continues until the entire set of data is processed; see Figure 1. Since bzip2 is a

sequential program, it will only start processing the next block after the previous one has been completed.

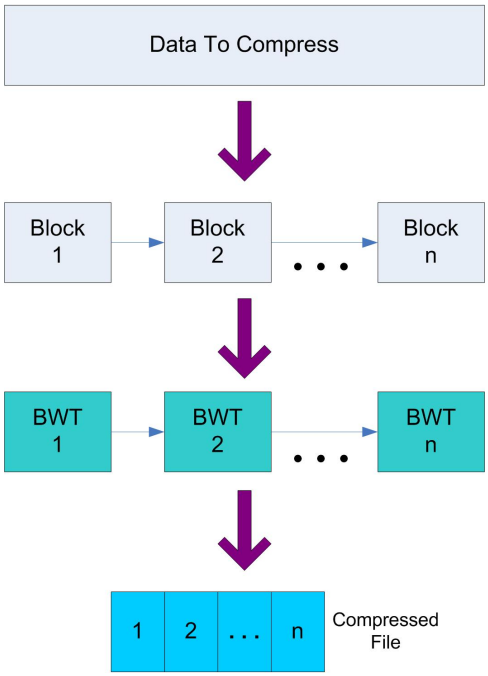

Figure 1: BZIP2 flow diagram

4.2 pbzip2

For pbzip2 to achieve speedup on parallel machines, a multi-threaded design was created using pthreads in C++. The code is generic enough that it should compile on any C++ compiler that supports pthreads. The code is linked with the libbzip2 library to access the BWT algorithm (http://sources.redhat.com/bzip2/).

The pbzip2 program also works by splitting the data into blocks of equal size, configurable by the user on the command line. Instead of reading in 5,000 bytes at a time, pbzip2 reads in an entire block at once which increases performance slightly, even with a single processor. The user can also specify the number of processors for pbzip2 to use during compression (default is two). The final argument pbzip2 needs on the command line is the file to compress. It supports compressing single files just as the bzip2 program does. If multiple files are needed to be compressed together, they should first be processed with TAR to create a single archive, and then the .tar file compressed using pbzip2.

A FIFO (first in, first out) queue system is used in pbzip2 with a producer/consumer model. The size of the queue is set to the number of processors pbzip2 was configured to use. This gives a good balance between speed and amount of memory required to run. Setting the queue size larger than the number of processors did not result in any speedup but significantly increased the amount of memory required to run. If the default block size of 900,000 bytes is used and the number of processors requested was two, pbzip2 will read the first two blocks of 900,000 bytes from the input file and insert the pointers to those buffers into the FIFO queue. Since pbzip2 is using two processors, two consumer threads

are created. Mutexes are used to protect the queue so only one thread can read from or modify the queue at a time. As soon as the queue is populated, the consumer threads will remove items from the queue and process the data using the BWT algorithm. Once the data is processed, the memory for the original block is released and the pointer to the compressed block along with the block number is stored in a global table. A file writing thread is also created which will go through the global table and write the compressed blocks to the output file in the correct order. Once free space in the queue is available, more blocks of data will be read from the file and added to the queue. This process continues until all blocks have been read, at which point a global variable is set to notify all the threads that file has been read in its entirety. The consumer threads will continue to process data blocks until the queue is empty and there are no more blocks going to be added to the queue. The pbzip2 program finishes when the file writing thread has finished writing all the compressed blocks to the output file in their correct order; see Figure 2.

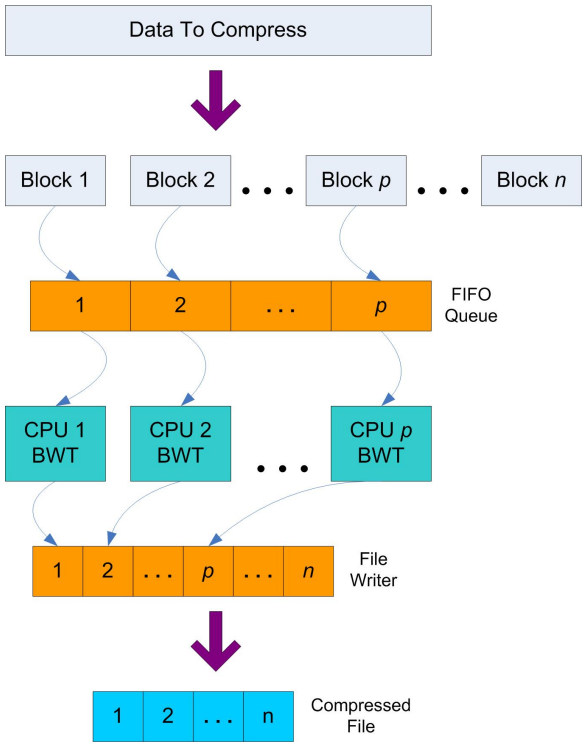

Figure 2: PBZIP2 flow diagram

5 Performance Results

The pbzip2 program was compiled on several different parallel platforms using the gcc compiler and the libbzip2 1.0.2 library. The input file for testing the performance of bzip2 and pbzip2 is the Linux kernel source tarball (linux-2.4.23.tar) which is 166,604,800 bytes (158 MB) uncompressed.

All experiments were measured as wall clock times in seconds. This includes the time taken to read from input files and write to output files. Each experiment was carried out three times and the average of the three results was recorded.

5.1 Athlon-MP 2600+

The platform used in this experimentation was an AMD Athlon-MP 2600+ machine with two 2.1 GHz processors and 1 GB of RAM. The software was compiled with gcc v3.3.1 under cygwin 1.5.5-1 on Windows XP Pro.

Since bzip2 just supports running on 1 processor, the results are only listed once given that the running time is the same no matter how many processors the system has. With pbzip2, the program was run once with the -p1 switch for 1 processor, and tested again with the -p2 switch for 2 processors. The results can be found in Table 1.

	Processors	Wall Clock Time	Compressed Size
bzip2 -9	1	101 sec	29,832,609 bytes
pbzip2 -p1	1	100 sec	29,897,521 bytes
pbzip2 -p2	2	54 sec	29,897,521 bytes

Table 1: Athlon-MP 2600+ Results

The bzip2 program establishes a baseline for performance. Running pbzip2 on one processor showed that it is at least as fast as bzip2. It was observed that when running pbzip2 on two processors, a near optimal speedup was achieved. The slightly larger compressed file size with pbzip2 is an artifact of how the compressed BWT blocks are concatenated together, but remains fully compatible with bzip2. A graph of the wall clock time as a function of the number of processors is shown in Figure 3.

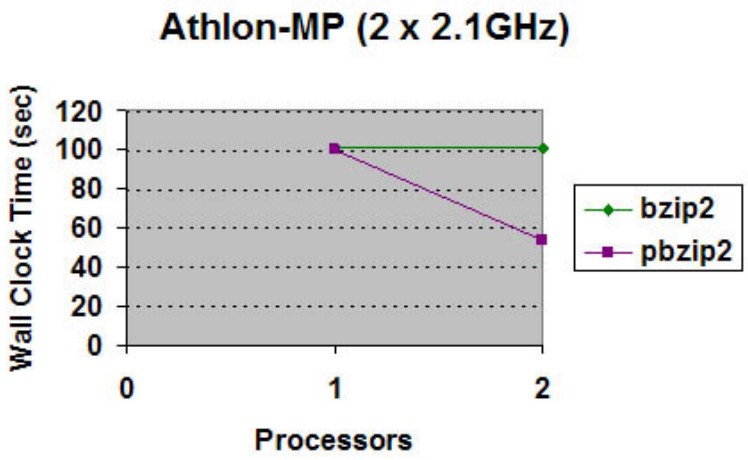

Figure 3: Wall clock time in seconds as a function of the nmber of processors to compress 158 MB on an Athlon-MP 2600+ system.

5.2 Intel P3 866MHz

The platform used in this experimentation was an Intel Pentium 3 machine with two 866 MHz processors and 1 GB of RAM. The software was compiled with gcc v3.2.2 and run on RedHat Linux 9.0 with the 2.4.20-20.9 SMP kernel.

Since bzip2 just supports running on 1 processor, the results are only listed once given that the running time is the same no matter how many processors the system has. With pbzip2, the program was run once with the -p1 switch for 1 processor, and tested again with the -p2 switch for 2 processors. The results can be found in Table 2.

	Processors	Wall Clock Time	Compressed Size
bzip2 -9	1	172 sec	29,832,609 bytes
pbzip2 -p1	1	158 sec	29,897,521 bytes
pbzip2 -p2	2	102 sec	29,897,521 bytes

Table 2: Pentium3 866MHz Results

The bzip2 program establishes a baseline for performance. Running pbzip2 on one processor showed that it was slightly faster than bzip2. This can probably be attributed to the fact that pbzip2 will read larger chunks of data at a time than bzip2, reducing the number of reads required. It was observed that when running pbzip2 on two processors, a 50% speedup was achieved. The slightly larger compressed file size with pbzip2 is an artifact of how the compressed BWT blocks are concatenated together, but remains fully compatible with bzip2. A graph of the wall clock time as a function of the number of processors is shown in Figure 4.

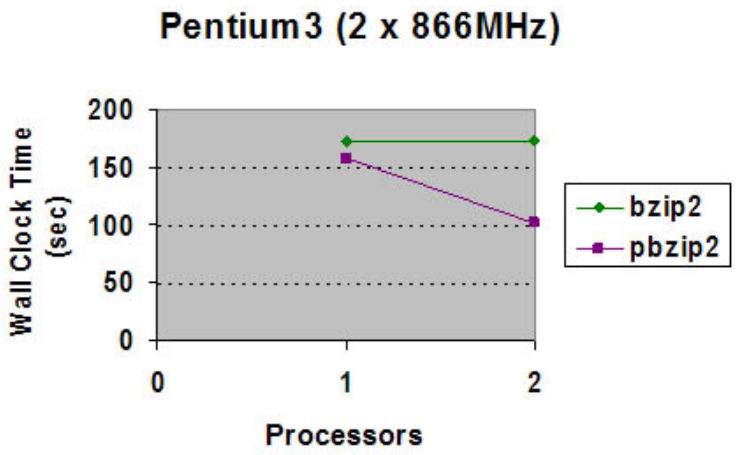

Figure 4: Wall clock time in seconds as a function of the nmber of processors to compress 158 MB on a Pentium3 866MHz system.

5.3 Intel P4 3.2GHz

The platform used in this experimentation was an Intel Pentium 4 machine with one 3.2 GHz processor (Hyperthreading enabled) and 2 GB of RAM. The software was compiled with gcc and run on Debian Linux 3.0r0 with the 2.4.23 SMP kernel.

Since bzip2 just supports running on 1 processor, the results are only listed once given that the running time is the same no matter how many processors the system has. With pbzip2, the program was run once with the -p1 switch for 1 processor, and tested again

with the -p2 switch for 2 processors to see if the hyperthreading would improve results at all. The results can be found in Table 3.

	Processors	Wall Clock Time	Compressed Size
bzip2 -9	1	57 sec	29,832,609 bytes
pbzip2 -p1	1	52 sec	29,897,521 bytes
pbzip2 -p2	2	47 sec	29,897,521 bytes

Table 3: Pentium4 3.2GHz Results

The bzip2 program establishes a baseline for performance. Running pbzip2 on one processor showed that it was slightly faster than bzip2. This can probably be attributed to the fact that pbzip2 will read larger chunks of data at a time than bzip2, reducing the number of reads required. It was observed that when running pbzip2 on one physical processor and one virtual "hyperthreaded" processor, a 10% speedup was achieved. The type of calculations that the BWT algorithm performs must keep a majority of the P4 processor busy so that hyperthreading does not provide much of a benefit. The slightly larger compressed file size with pbzip2 is an artifact of how the compressed BWT blocks are concatenated together, but remains fully compatible with bzip2. A graph of the wall clock time as a function of the number of processors is shown in Figure 5.

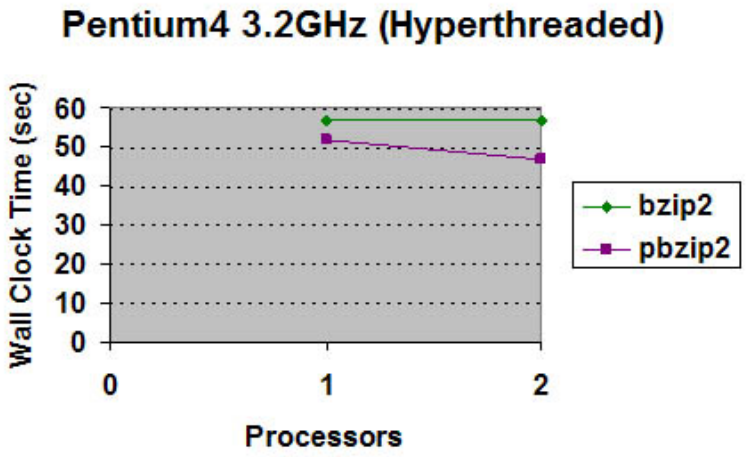

Figure 5: Wall clock time in seconds as a function of the nmber of processors to compress 158 MB on a Pentium4 3.2GHz system.

5.4 SunFire 6800

The platform used in this experimentation was a SunFire 6800 machine with twenty four 900 MHz UltraSPARC-III processors and 96 GB of RAM. The software was compiled with gcc v3.2.3 and run on SunOS 5.9.

Since bzip2 just supports running on 1 processor, the results are only listed once given that the running time is the same no matter how many processors the system has. With pbzip2, the program was run once with the -p1 switch for 1 processor, and tested again with

the appropriate command line switches for 2, 4, 6, 8, 10, 12, 14, 16, 18 and 20 processors. The SunFire machine is shared amongst various researchers so it was not possible to get exclusive access to more than 20 processors. The results can be found in Table 4.

	Processors	Wall Clock Time	Compressed Size
bzip2 -9	1	105 sec	29,832,609 bytes
pbzip2 -p1	1	106 sec	29,897,521 bytes
pbzip2 -p2	2	53 sec	29,897,521 bytes
pbzip2 -p4	4	27 sec	29,897,521 bytes
pbzip2 -p6	6	18 sec	29,897,521 bytes
pbzip2 -p8	8	14 sec	29,897,521 bytes
pbzip2 -p10	10	11 sec	29,897,521 bytes
pbzip2 -p12	12	10 sec	29,897,521 bytes
pbzip2 -p14	14	8 sec	29,897,521 bytes
pbzip2 -p16	16	7 sec	29,897,521 bytes
pbzip2 -p18	18	7 sec	29,897,521 bytes
pbzip2 -p20	20	6 sec	29,897,521 bytes

Table 4: SunFire 6800 Results

The bzip2 program establishes a baseline for performance. Running pbzip2 on one processor showed that it was approximately the same speed as bzip2. It was observed that when running pbzip2 on multiple processors, near-optimal speedup was achieved. The largest variation was seen with 18 processors where pbzip2 was at 84% of optimal speedup, and with 20 processors it was at 88% of optimal speedup. This variation can most likely be attributed to the timing being done in seconds and not a smaller unit of time. The slightly larger compressed file size with pbzip2 is an artifact of how the compressed BWT blocks are concatenated together, but remains fully compatible with bzip2. A graph of the wall clock time as a function of the number of processors is shown in Figure 6. The relative speedup as a function of the number of processors can be found in Figure 7.

6 Conclusion

In this paper, several methods of parallelizing the Burrows-Wheeler Transform are described. One method, the simultaneous processing of BWT blocks, was implemented and the performance of the algorithm was evaluated on several shared memory parallel architectures. The results showed that a significant, near-optimal speedup was achieved by using the pbzip2 algorithm on systems with multiple processors. This will greatly reduce the time it takes to compress large amounts of data while remaining fully compatible with the sequential version of bzip2. Future work may consist of implementing pbzip2 on a distributed memory cluster architecture, and exploring the parallel lexicographical sort method of parallelizing the BWT algorithm.

7 Acknowledgements

Special thanks to Dr. Frank Dehne and John Saalwaechter for their suggestions, and Ken Hines and HPCVL (http://www.hpcvl.org/) for use of their computing resources.

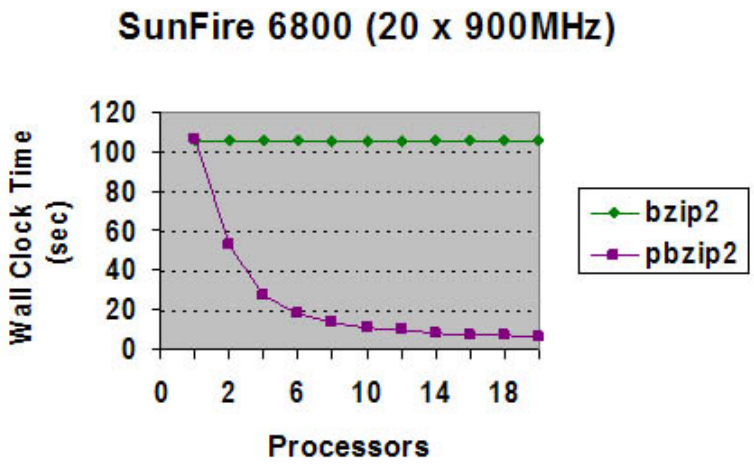

Figure 6: Wall clock time in seconds as a function of the nmber of processors to compress 158 MB on a SunFire 6800 system.

References

[1] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[2] Brenton Chapin and Stephen R. Tate. Higher compression from the burrows-wheeler transform by modified sorting. In *Data Compression Conference*, page 532, 1998.

[3] J. G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *The Computer Journal*, 40(2/3):67–??, ???? 1997.

[4] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32(4):396–402, April 1984.

[5] Michelle Effros. Universal lossless source coding with the burrows wheeler transform. In *Data Compression Conference*, pages 178–187, 1999.

[6] P. Fenwick. Block-sorting text compression — final report, 1996.

[7] Jeff Gilchrist. Archive comparison test (http://compression.ca). Technical report, 2002.

[8] R. Nigel Horspool. Improving LZW. In *Data Compression Conference*, pages 332–341, 1991.

[9] Paul G. Howard and Jeffery Scott Vitter. Arithmetic coding for data compression. Technical Report Technical report DUKE–TR–1994–09, 1994.

[10] Julian Seward. *The bzip2 and libbzip2 official home page (http://sources.redhat.com/bzip2/)*, 2002.

[11] Lynn M. Stauffer and Daniel S. Hirschberg. Parallel text compression - REVISED. Technical Report ICS-TR-91-44, 1993.

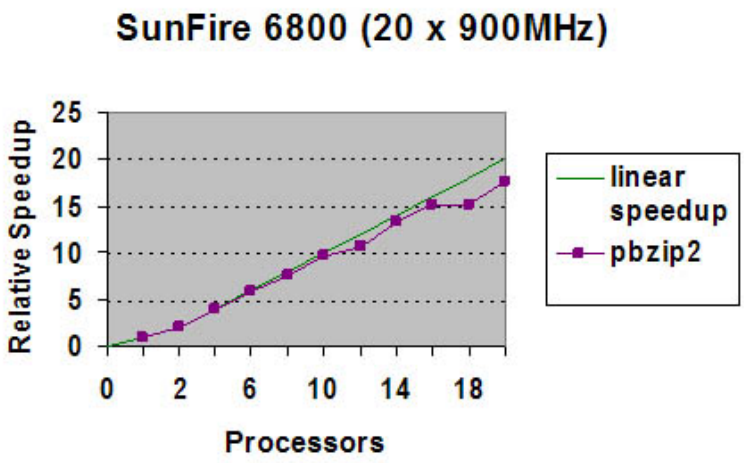

Figure 7: Relative speedup as a function of the nmber of processors to compress 158 MB on a SunFire 6800 system.

[12] Lynn M. Stauffer and Daniel S. Hirschberg. Dictionary compression on the PRAM. Technical Report ICS-TR-94-07, 1994.

[13] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.