Parallel Lossless Data Compression Based on the Burrows-Wheeler Transform

by

Jeffrey S. Gilchrist

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfillment of the requirements for the degree of

Master of Applied Science

in Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

September 2006

Your file Votre référence
ISBN: 978-0-494-23338-2
Our file Notre référence
ISBN: 978-0-494-23338-2

To my parents, who taught me everything I needed to succeed in life.

To my wife Johanne, for her invaluable support and encouragement.

And to our son Sébastien, who makes all the hard work worthwhile.

Acknowledgments

I would like to thank my thesis supervisor, Dr. Aysegul Cuhadar, for her experience, guidance, and patience during this research work. Her high standards have given me great insight and significantly improved the final outcome.

I would also like to thank Carl Bond and Elytra Enterprises Inc. for their financial support, Dr. Frank Dehne for inspiring me to pursue this research topic, Geoffrey Green for his advice, and HPCVL and SHARCNET for use of their computing resources. Many thanks to Blazenka Power for keeping me on track.

Abstract

This thesis presents parallel algorithms for lossless data compression based on the Burrows-Wheeler Transform (BWT) block-sorting technique. We investigate the performance of using data parallelism and task parallelism with both multi-threaded and message-passing programming. The output produced by the parallel algorithms is fully compatible with their sequential counterparts. To balance the workload among processors we develop a task scheduling strategy. An extensive set of experiments is performed with a shared memory NUMA system using up to 120 processors and on a distributed memory cluster using up to 100 processors. Our experimental results show that significant speedup can be achieved with both data parallel and task parallel methodologies. These algorithms will greatly reduce the amount of time it takes to compress large amounts of data while the compressed data remains in a form that users without access to multiple processor systems can still use.

Contents

List of Tables

List of Figures

List of Acronyms

AMD	Advanced Micro Devices
BWT	Burrows-Wheeler Transform
CPU	Central Processing Unit
FIFO	First In First Out
FPGA	Field Programmable Gate Array
HP	Hewlett-Packard
IO	Input Output
JPEG	Joint Photographic Experts Group
KB	Kilobyte
LFF	Longest Fragment First
LQN	Layered Queuing Network
LZ	Lempel-Ziv
LZ77	Lempel-Ziv from year 1977
LZW	Lempel-Ziv-Welch
MB	Megabyte

MPEG	Moving Picture Experts Group
MPI	Message Passing Interface
MTF	Move-To-Front
mutex	Mutual Exclusion
MVA	Mean Value Analysis
NUMA	Nonuniform Memory Access
PC	Personal Computer
POSIX	Portable Operating System Interface
PPM	Prediction by Partial Match
PRAM	Parallel Random Access Machine
SGI	Silicon Graphics Inc.
SMP	Symmetric Multiprocessing
TCP/IP	Transmission Control Protocol over Internet Protocol
TIFF	Tagged Image File Format
ZLE	Zero Length Encoding

Chapter 1

Introduction

1.1 Motivation

Data compression takes advantage of redundancy to compact files or data into a smaller form. It is often used to package software and data to reduce storage requirements, and to shorten the amount of time and bandwidth required to transmit the data over a network. Lossless data compression has the constraint that when data is uncompressed, it must be identical to the original data that was compressed. This type of compression is essential for some applications such as text, executables, databases, and medical images. Graphics and video compression such as baseline JPEG [40] and MPEG4 [26] on the other hand use lossy compression schemes which discard some of the original data. This leads to better compression than lossless algorithms, but at the expense of not being able to perfectly reconstruct the original data.

The majority of general-purpose software is designed for sequential processing and does not take advantage of parallel computing. Lossless data compression software like bzip2, gzip, and zip are designed for sequential processing. The data compression algorithms require a great deal of processing power to analyze and then encode data

into a more compact form. Furthermore, as the size of software and data continues to increase, faster data compression rates will be needed. 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.

AMD and Intel processors are now being designed with multiple cores, giving even the general public the ability to take advantage of multi-threaded and message-passing software. The Message Passing Interface (MPI) library is free and can be used for developing parallel applications using message-passing code and a POSIX thread library is freely available in most compilers for developing multi-threaded code. With the increasing availability of systems with multiple processors and multiple cores, parallel lossless compression algorithms can be of great use to both advanced and novice users.

1.2 Thesis Objectives

The main objective of this thesis is to develop parallel algorithms for the Burrows-Wheeler Transform (BWT) lossless data compression algorithm. The BWT was chosen because it produces smaller compressed data than the common ZIP format but is significantly slower. It achieves 30% better compression on text data but is 1.5 times slower than ZIP and obtains 21% better compression on the Canterbury Corpus data set and runs 1.2 times slower than ZIP [17]. The BWT uses less computational resources and performs within 8% compression on average of the Prediction by Partial Match (PPM) statistical algorithms, the current best at compressing text data. A parallel version would allow the BWT to be more competitive speed-wise with ZIP while remaining competitive compression-wise with the best lossless algorithms.

The methods we propose are based on data parallel and task parallel methodologies. We provide algorithm designs for both shared memory systems using multiple

threads and for distributed memory systems using message-passing. It is anticipated that the parallel algorithms will achieve significant speedup compared to the sequential BWT compression algorithm, allowing the compression of large data sets that were not feasible before. A secondary goal is to provide practical algorithm solutions that can be used in real systems owned by the general population, not just expensive research systems or theoretical ones. To increase the usefulness and adoption of these parallel algorithms, we propose a design whose output is compatible with the popular sequential bzip2 compressor in wide use today on UNIX systems.

1.3 Thesis Contributions

The following is a summary of the main contributions from this thesis:

- A parallel multi-threaded version of the BWT lossless data compression algorithm was developed for shared memory systems using a data parallel strategy. The output of this algorithm is fully compatible with the popular sequential bzip2 software, allowing either one to work with the same data.

- We also developed parallel message-passing versions of the BWT lossless data compression algorithm for clusters using both data parallel and task parallel strategies. The output of these algorithms is fully compatible with the sequential versions of the software, allowing either one to work with the same data.

- The performance of the data parallel and task parallel BWT algorithms were evaluated. Our experimental results showed significant speedup, near linear in some cases, for the multi-threaded and message-passing versions of the algorithms.

- A software package using the data parallel multi-threaded BWT algorithm was created and made available free to users, including the source code (*http://compression.ca/pbzip2/*). The pbzip2 software can also be found bun-

dled in the Debian, FreeBSD, Gentoo, NetBSD, RedHat, and Solaris UNIX distributions.

1.4 Thesis Outline

In Chapter 2, we present background information on lossless data compression. The two main classes of lossless compressors, dictionaries and statistics, are introduced. The Lempel-Ziv (LZ) dictionary compression algorithm is discussed as well as the arithmetic coding and prediction by partial match (PPM) statistical algorithms. Finally, a detailed description of the Burrows-Wheeler Transform (BWT) block-sorting algorithm is given.

In Chapter 3, we outline background information on parallel programming and parallelization techniques. We look at the multi-threaded and message-passing approaches to parallel programming. A brief overview of the data and task parallel methodologies for parallelizing algorithms is also given.

Chapter 4 describes the parallel BWT algorithms developed for this thesis. The sequential version of the bzip2 BWT compressor is first addressed. A description of the data parallel multi-threaded version of bzip2 is then given. This algorithm is designed to run on shared memory SMP and NUMA computer systems. A data parallel message-passing version of the bzip2 algorithm, designed to run on distributed memory cluster systems is then introduced. This is followed by a description of the sequential version of the bwtzip BWT compressor. Lastly, a task parallel message-passing version of the bwtzip algorithm, designed to run on distributed memory systems is presented.

In Chapter 5, the experimental results of the parallel algorithms are reported and analysed. The data and task parallel algorithms were tested on both shared memory and distributed memory systems.

Lastly, Chapter 6 summarizes our findings for the parallel BWT algorithms and discusses some improvements that can be evaluated in future research.

Chapter 2

Background – Lossless Data Compression

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 [12]. The most popular lossless data compression techniques in use today are Lempel-Ziv (LZ), Prediction by Partial Match (PPM), and BWT-based algorithms.

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) is 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 [42].

Many variants and improvements to the LZ algorithm were proposed and implemented since its initial release. One such improvement is Lempel-Ziv-Welch (LZW) 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. Decompression with LZW is faster than compression since string searching in the dictionary is not necessary [19].

Some research on creating parallel dictionary compression is available. Once the dictionary has been created in an LZ based algorithm, a greedy parsing is performed 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 [35]. Stauffer a year later expands on previous parallel dictionary compression schemes on the Parallel Random Access Machine (PRAM) by using a parallel longest fragment first (LFF) algorithm to parse the input instead of the greedy algorithm [36]. 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 $O(\frac{n}{\log n})$. This algorithm is not practical since very few people if anyone would have a machine with enough processors to satisfy the requirements. Splitting data into smaller blocks for parallel compression with LZ77 yields speedup but poor compression performance because of the smaller dictionary sizes. A cooperative method for building a dictionary achieves speedup while keeping similar compression performance [15]. The data is divided into sub-blocks and assigned to multiple processors. A dynamic dictionary is constructed with the cooperation of all processors involved in the compression. Since the dictionary is shared amongst the processors, it allows for better compression of the data than using smaller independent dictionaries on each sub-block.

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 [12].

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 probability 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 [20].

The Prediction by Partial Match (PPM) algorithm is currently the best lossless data compression algorithm for textual data. It was first published in 1984 by Cleary and Witten [10]. 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 memory and CPU resources than PPM and are usually much faster [17]. Compressing TIFF image data, the LZ based gzip software is 2.8 times faster and BWT based bzip2 is 1.4 times faster. 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 [9]. 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 yields a 5.6% compression performance increase over the older PPM algorithm which places it 3.7% ahead of the BWT algorithm on the Calgary Corpus data set.

2.3 Burrows-Wheeler Transform

The Burrows-Wheeler Transform [5] is a block-sorting, lossless data compression algorithm that works by applying a reversible transformation to a block of input data first and then compressing it. The sorting stage of the algorithm does not perform any compression but modifies the data in a way to make it easy to compress with

the other stages of the algorithm such as "move-to-front" coding and then 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.

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 [12]. 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.

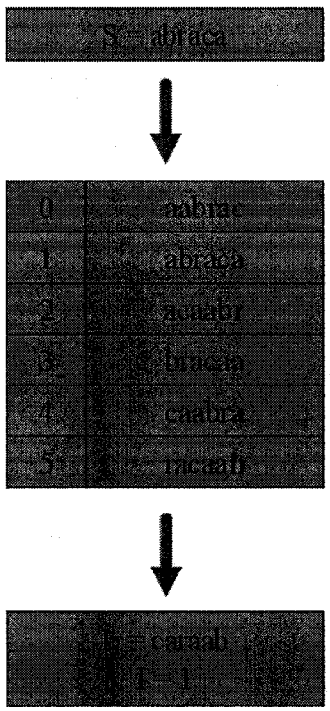

Figure 2.1: Sorting stage of BWT algorithm

The first stage sorts the input data so that similar contexts are grouped together. Figure 2.1 illustrates how the stage works, transforming the string "abraca" (S) of 6 characters by forming the 6 rotations (cyclic shifts) of S, sorting them lexicographically. The last character from each of the 6 rotations is used to form the string "caraab" (L), 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 which occurs after the second rotation in our example. With only L and I there is an efficient algorithm to compute the original string S when performing inverse BWT for decompression. 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. Bzip2 uses two different sorting algorithms for its BWT implementation [31]. The direct-comparison algorithm has a best-case complexity of $O(N \log N)$ and a worst-case complexity of $O(N^2 \log N)$. The doubling algorithm has a worst case complexity of $O(N \log N)$. Even though the doubling algorithm has a better worst time complexity, the direct-comparison algorithm outperforms the doubling algorithm for average match lengths less than 100. Therefore direct-comparison is used first but can fall back to the doubling algorithm when deep comparisons are encountered. This sorting stage is the most computationally complex stage of the BWT algorithm.

In the second stage, before the data 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. With similar contexts together, any region of the sorted data should contain few symbols that are efficiently captured by the move-to-front compressor [12]. 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 [8]. 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.

Finally, the compression stage uses an order-0 arithmetic coder to take advantage of the skewed transformed data producing effective smaller output sizes. To decompress the data, the three stages are performed in opposite order, first using an arithmetic decoder, then move-to-front decoder and finally computing the original string by reversing the transformation, which can be done in an efficient manner. Since the computationally complex sorting does not need to be performed to reverse the transformation, decompression using BWT is significantly faster than compression.

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 $O(\log \frac{\log n}{\log n})$ and LZ78 which is $O(\frac{1}{\log n})$ [11]. 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).

2.4 Summary

In this chapter we reviewed some dictionary and statistical based lossless com-

pression algorithms. Dictionary algorithms such as Lempel-Ziv are fast and still in wide use today by programs such as ZIP. They trade compression performance for speed. The statistical algorithms such as PPM and BWT obtain better compression than dictionary algorithms but are more computationally intensive and slower. These algorithms trade speed for better compression performance.

In the next chapter, we provide some background information on multi-threaded and message-passing programming as well as parallelization strategies for creating parallel applications.

Chapter 3

Background – Parallel Programming

This chapter provides some background information on parallel programming and parallelization techniques. A brief overview of strategies for parallelizing algorithms is given. We also look at the multi-threaded and message-passing approaches to parallel programming.

3.1 Parallelization Strategies

Parallel applications can be designed using several different parallel programming strategies. A small number of strategies are used in the majority of parallel applications [7]. A specific strategy is chosen based on the underlying parallelism in the problem being solved and the parallel computing resources available. Parallelism can be achieved from the structure of the data by running identical code on different sections of data simultaneously (data parallelism). It can also be achieved from the structure of the application, where the program can perform different tasks concurrently (task parallelism).

Four main steps can be used in designing parallel algorithms [14]. First, the

problem is partitioned into small tasks, then the communication needed for acquiring the data to solve the problem is organised. Next agglomeration can be used to decrease communication costs and improve efficiency. Finally, tasks are mapped to processors.

3.1.1 Partitioning

Partitioning is used to break up the problem at hand into small tasks that can be executed in parallel [14]. This partitioning can be achieved by splitting the data into pieces that can be processed separately, ideally of similar size. The most frequently accessed or largest data structures are good candidates for partitioning. Partitioning can also be accomplished by decomposing the computation required to process the data into separate tasks. Some opportunities for parallelization in breaking up the functional sections of the problem may exist that are not possible with partitioning the data alone. Both methods should be explored to produce as many tasks as possible without regards for the type of parallel computing platform being used.

Data Parallelization

Data parallelization is a simple strategy used in parallel programming to increase the performance of an application using multiple processors by partitioning data into blocks and distributing it amongst the processors [29]. The individual processors execute the same set of code on their portions of the data. The results are then collected from each processor. Parallelism is achieved from the partitioned data being processed simultaneously rather than in sequential order. Communications between processors is most often highly structured and predictable [7]. This strategy can be used on both shared-memory systems with techniques such as multi-threading and also on distributed memory systems with message-passing. If the time it takes to partition and manage the data on multiple processors is less than the computation

time required to process the data, the application should obtain some form of speedup.

An integer summing program can be implemented using data parallelization. The integer series 1 through 10 is partitioned into two blocks of data, 1 through 5 and 6 through 10. Different processors run the identical set of code on the partitioned data concurrently, providing parallelism to the software and increasing its speed. Instead of partitioning the data into two blocks, it could be broken into any arbitrary number of blocks to suit the load balancing required.

Task Pipelining

Pipelining is another strategy to reduce the execution time of an application with multiple processors. Instead of partitioning data, the application is partitioned into independent tasks. Each task can be made into a different stage of a software pipeline, which may also run in parallel [13]. Software pipelines are ideal for continuous-flow systems where data is always available to be inserted into the pipeline. They can be synchronous where the previous stage must be complete before the next can begin, asynchronous where each stage is independent and can run in any order, or a mixture of both.

Pipelining can increase the throughput and reduce latency of software by overlapping multiple stages. Throughput is the amount of data that can be processed by an application per unit of time [23]. Latency on the other hand is the amount of time it takes for the application to start and finish processing data, so the minimum amount of time required. A sequential algorithm cannot start processing the next block of data until it as finished the current one [13]. When a stage in the pipeline has completed and passed the data onto the next, it can start processing the next block immediately without having to wait for the entire algorithm to finish. The performance of a simple software pipeline that sequentially divides the software into tasks that run on different processors will be constrained by the slowest task.

Software pipelines can take advantage of several types of parallelism. Temporal multiplexing allows a task to be processed in parallel [13]. While the task will not complete any faster, the average throughput should increase with the number of processors being used. Geometric multiplexing, also called data parallelization, provides parallelism by partitioning the data so it can be run on multiple processors concurrently. Algorithmic multiplexing is the partitioning of a task so that the subtasks can run on different processors. Temporal, geometric, and algorithmic multiplexing can be combined in the software pipeline as required.

Divide and Conquer

The divide and conquer strategy takes a problem and divides it into two or more sub-problems which are solved independently and then combined for a final result. This technique is also commonly used in sequential programming to recursively break down a problem until it becomes simple enough to solve directly [7]. The overhead introduced using this strategy includes the processing required to split the problem into sub-problems, distribute the sub-problems to multiple processors, and then join the results. Since each sub-problem is distinct, there is no communication required between the processors solving the sub-problems.

The Master/Slave strategy is similar to divide and conquer except that the problem is split into tasks beforehand and join operations are performed only by the master. The work is statically partitioned and dynamically distributed. With divide and conquer, tasks can be split during runtime and any process in the parallel system can perform split and join operations. In this case, the work is both dynamically partitioned and distributed.

Hybrid Models

Some problems maybe able to take advantage of both data and task parallelism [7]. Multiple strategies can be used within the same application, either simultaneously or in different sections of the code. For example, an application may be split into separate tasks to form a software pipeline. The speed of a software pipeline will be limited by its slowest stage. If that stage of the pipeline can also take advantage of data parallelization and the data partitioned onto multiple processors, it will improve the overall speed of the pipeline using multiple strategies.

3.1.2 Communication

Partitioned tasks are designed to operate in parallel but may not be independent, requiring data from another task in order to complete the processing [14]. Even independent tasks will need to obtain data to perform its computations. Communications will therefore be needed to move data to and from the tasks. Local communications are used to relay data from a small number of tasks with communications channels that are easy to define. The task can communicate directly with the other tasks required to obtain the data to process. Global communications however are more complicated with multiple tasks required to cooperate. A task may require data from other tasks, which in turn require data from another set of tasks in order to proceed. Communication can be static in that the data follows a regular pattern or dynamic where it can change during execution. Synchronous communication occurs when one task explicitly sends data to another task that is waiting to receive it. Computation does not continue until the communication is complete. Asynchronous communication is also possible where a task will send data to another task without knowing if it is ready to receive the data, and then continue with its computation without blocking. This also adds overhead to the receiving side, which needs to poll to determine if data has been received.

3.1.3 Agglomeration

Agglomeration is the process of combining tasks together. Since the partitioning and communication design steps did not take into account specific parallel architectures, the resulting algorithm may be inefficient for a particular parallel computer system [14]. Some tasks may be combined to form larger but more efficient tasks with less communication on a distributed memory system for example. Reducing the time spent communicating and number of messages sent can improve performance. If communication is being used between a small number of tasks, agglomerating those tasks may reduce the volume enough to improve performance. Combining tasks is also useful when concurrency between them is not possible. Agglomeration should be used to modify the parallel algorithm to work more efficiently with the parallel architecture being considered.

3.1.4 Mapping

After the tasks have been partitioned, communications strategies devised, and agglomeration complete, the tasks must be mapped to a processor to execute the algorithm [14]. The objective of mapping is to minimize the total execution time of the algorithm. On shared memory systems using threads, the operating system can in many cases automatically assign tasks to processors based on the load. In general however, mapping is difficult with no known polynomial time algorithm for determining the minimum execution time. In practice, several task-scheduling strategies have been devised to help with mapping.

Task scheduling is used to maximize processor utilization and minimize the amount of communications between processors [30]. Processor utilization will be maximized if the workload is balanced. If the time to process all tasks is identical, the parallel application should be computationally balanced. If the processing time varies depending on the task, certain processors will receive less work than others leaving

them idle while waiting for new work, creating a system that is not evenly balanced. A load-balancing algorithm will need to be devised in order to best assign tasks so the work load amongst processors is as evenly balanced as possible. Static load balancing occurs at run time to determine the best way to partition the tasks for the available processors and is fixed for the execution of the application. Dynamic load balancing can be used periodically during the execution of the application to analyze the current tasks and change the partitioning if required. Task scheduling algorithms also require a method to determine and signal when processing is complete so tasks do not continue requesting work indefinitely [14].

Master/Slave

The Master/Slave strategy comprises a master and one or more slaves. The master is in charge of assigning and distributing small tasks to the slaves. It might also collect the results from the slaves to produce the final result [7]. The slaves receive data from the master, process the data, and return the results, usually only communicating with the master. The Master/Slave strategy can be used with both data and task parallelism to achieve speedup. When a large number of processors are used, the master eventually becomes a bottleneck since it is one entity communicating with many slaves. A hierarchical Master/Slave model extends the strategy by dividing slaves into disjoint sets managed by their own sub-master [14]. The master assigns tasks to the sub-masters, which will then assign tasks to their group of slaves.

The integer summing program described previously can be designed to use a Master/Slave strategy. A master process splits the integer series 1 through 10 into two blocks of data, 1 through 5 and 6 through 10. It then sends the data to two different slave processes. Each slave sums the integer series it was given, and returns their sum to the master. The master collects the data from the slaves to produce the final sum.

Decentralized

Task pools can be maintained on each processor in a decentralized manner with no central manager [14]. When a task requires more work, it will contact another processor requesting data to process in an asynchronous manner. Tasks can either be configured to contact other processors at random for work requests or from a list of specific processors. With no central manager to become a bottleneck, a decentralized strategy may scale higher at the cost of being more complex to implement.

3.2 Multi-threaded Programming

Multi-threaded programming is a parallel programming technique that is often used on shared-memory systems to reduce the execution time of applications. The two common shared-memory systems in use today are symmetric multiprocessing (SMP) systems in which all processors have direct access to the same physical memory and non-uniform memory access (NUMA) systems [29]. With NUMA, groups of processors have direct access to a portion of physical memory. If a processor needs to access data in the physical memory of another group, the data must be fetched over a communications link, which is slower than reading from its own memory pool. Large shared-memory systems use custom designed hardware and are extremely expensive.

A thread is a lightweight process that contains its own stack and executes code. It is a subset of a process and shares its memory space with that process and all other threads within the process [6]. Because of this, computer systems can switch between threads in a process much faster than between different processes. The use of threads also allows for simple and high-bandwidth communication since all threads have access to the same memory space and do not need to use other methods such as message-passing, sockets, or pipes to communicate. The POSIX threading standard (pthreads) allows developers to create and manage threads in software. Threads within a process

can be created and destroyed dynamically at the beginning or during execution of the program. They allow for multiple computations to be performed simultaneously on systems with more than one processor. This will give programs with independent sections of code the ability to achieve speedup when used with multiple processors. The use of multi-threaded programming also introduces computing overhead. There are costs for the time needed to manage and synchronize the threads within a process including the need to protect common memory locations from being accessed by more than one thread at the same time. Depending on the application, the added overhead from managing threads may be more than the speed gained and not provide any true performance benefits.

Since the memory space in a process is shared amongst threads, certain portions of code that access common variables or memory locations need to be protected from multiple threads trying to access them at the same time. Any code that can only be executed by one thread at any given time is called a critical section [29]. This problem is handled in the pthreads library by allowing the use of a mutual exclusion (mutex). When a thread tries to execute code in a critical section it first checks to see if the mutex for that code is locked. If not, it will lock the mutex, execute the code, and then unlock the mutex when finished. If any other thread tries to execute code in the critical section at the same time, it will see that the mutex is locked, and block until the other thread has unlocked it again. Careful planning is required to ensure that threads do not spend an excessive amount of time waiting for access to critical sections. It is also important that a thread releases a mutex lock when it is complete so that other threads will not block indefinitely.

Figure 3.1 shows an example of a simple multi-threaded program that calculates the sum of the integers 1 through 10 using pseudocode. In a sequential program, the numbers would be summed one at a time. With the parallel multi-threaded version, several numbers in the series could be summed simultaneously. If two threads were

being used, the integer series would be split in two and stored in a local array for each thread. The results would be stored in a global variable called *sumGlobal*.

```
BEGIN Main_Program
     SET sumGlobal to 0
     SET array1 to 1,2,3,4,5
     SET array2 to 6,7,8,9,10
     CREATE sumThread with array1
     CREATE sumThread with array2
     CALL waitForThreadsToFinish
     PRINT sumGlobal
END

BEGIN sumThread( arrayLocal )
     SET sumLocal to 0
     FOR all elements in arrayLocal
          ADD arrayLocal.element to sumLocal
     ENDFOR
     LOCK mutex variableMutex
          ADD sumLocal to sumGlobal
     UNLOCK mutex variableMutex
END
```

Figure 3.1: Pseudocode for integer summing program using multi-threaded programming

The use of a mutex is required to protect the *sumGlobal* variable from being accessed by both threads at the same time. Each thread can sum half the numbers in the series simultaneously making the multi-threaded version theoretically twice as fast. In practice, the overhead of creating the threads and managing the mutex locks will reduce the performance somewhat. While only a simple example, it illustrates how multi-threaded programming can be used to parallelize software.

3.3 Message-passing Programming

Message-passing is the most common parallel programming technique that is used on distributed-memory systems to achieve speedup in applications [29]. A typical system includes a cluster of PCs that communicate via network interconnect such as

Ethernet, Myrinet, or Quadrics. Clusters can be built with standard PC components and use commonly available Ethernet networking for the interconnect making them much less expensive than large shared-memory systems.

The nodes in the cluster can be connected in several different configurations ranging from a simple linear array to fully connected networks. In a fully connected network, each node has a direct communications path to every other node so can quickly send data to another node even when any other communications is taking place [29]. The cost to build a fully connected network is very high for anything but small clusters making it impractical. Instead, a more practical choice is the hypercube, torus, or mesh. Clusters using Ethernet interconnect are typically connected to a high-bandwidth network switch allowing each node to communicate with another requiring only 1 hop through the switch. Enterprise level switches such as the 6500 Series from Cisco Systems scales from 16 ports up to 576 ports with Gigabit Ethernet. This would allow a cluster with quad-processor nodes to have 2304 processors connected via Gigabit Ethernet interconnect in a near-fully connected network configuration.

With message-passing, processes communicate and synchronize their work by sending and receiving messages over the interconnect. The Message-Passing Interface (MPI) [24] is a popular standard for implementing message-passing software on clusters. Unlike multi-threaded programming, processes are not dynamically created or destroyed during runtime but statically set when the program first executes [29]. Each process is assigned a unique rank from 0 up to the number of processes minus 1. The same code is run on each processor so the rank integer value can be used to programmatically execute different sections of the code to change the behaviour of certain nodes. A process with rank 0 could perform a different function than the process with rank 1. MPI supports both blocking and non-blocking communication. The standard send and receive commands in MPI use blocking communication. If rank 0

tries to send data to rank 1, the rank 0 process will block until the rank 1 process calls the corresponding receive command. Both processes will also block until the transfer of data is complete. MPI also has a non-blocking send and receive command which means that rank 0 can send data to rank 1 before rank 1 is ready to receive the data. The message is stored in a system-controlled buffer until rank 1 calls the receive command to retrieve it. Once the non-blocking send is complete on rank 0, control is returned to the process immediately and it can continue. If there is other work rank 1 can do while waiting for messages, a non-blocking receive can be used to return control to the process. The status of the non-blocking receive may be periodically checked to see if a message has actually been received yet and then act upon it. Since each rank is a separate process and no memory is shared between ranks, there are no critical sections to be protected and no mutual exclusions required.

Figure 3.2 describes a message-passing version of the example program (Figure 3.1) that calculates the sum of the integers 1 through 10 using pseudocode. With the parallel message-passing version, several numbers in the series could be summed simultaneously. A master/slave model is used, one master rank coordinating work for slave ranks to perform. If two slave processors were being used, the integer series would be split in two on the master rank and sent to slave ranks in the cluster to be processed. Once each rank was finished summing their portion of the series, they would send the result back to the master rank. The results would then be summed together and stored in variable called *sumTotal*.

Each slave rank can sum half the numbers in the series simultaneously making the message-passing version theoretically twice as fast. In practice, the overhead of message communications and managing buffers will reduce the performance somewhat. While only a simple example, it illustrates how message-passing programming can be used to parallelize software.

```
BEGIN Main_Program
     SET masterRank to 0
     CALL GetMyRankID RETURNING myRank
     IF myRank is masterRank THEN
          SET array1 to 1,2,3,4,5
          SET array2 to 6,7,8,9,10
          SET sumTotal to 0
          SET sumLocal to 0
          CALL SendBlockToSlave with array1, slaveID1
          CALL SendBlockToSlave with array2, slaveID2
          FOR all slave ranks
               CALL ReceiveBlockFromSlave RETURNING sumLocal
               ADD sumLocal to sumTotal
          ENDFOR
          PRINT sumTotal
     ELSE
          SET sumLocal to 0
          CALL ReceiveBlockFromMaster RETURNING arrayLocal
          FOR all elements in arrayLocal
               ADD arrayLocal.element to sumLocal
               CALL SendBlockToMaster with sumLocal, masterRank
          ENDFOR
     ENDIF
END
```

Figure 3.2: Pseudocode for integer summing program using message-passing programming

3.4 Summary

In this chapter we described several strategies for creating parallel applications using data and task parallelism. We also discussed multi-threaded programming on shared-memory systems and included an example of an integer-summing program using pseudocode. Finally, we described how message-passing programming on distributed memory systems can increase speedup of applications with a pseudocode example.

In the next chapter, we describe data parallel and task parallel strategies for parallelizing the BWT compression algorithm using both multi-threaded and message-passing programming.

Chapter 4

Parallelizing BWT Compression Algorithm

In this chapter we review other parallel implementations of the Burrows-Wheeler Transform (BWT) block-sorting compression algorithm. Two new techniques are also described for parallelizing the BWT. The first involves parallelizing the data stream with multi-threaded and message-passing algorithms. The second technique separates the BWT algorithm into multiple tasks to create a software pipeline with a message-passing algorithm.

4.1 Current Parallel BWT Implementations

It seems very little research has been performed on parallel BWT algorithms to date. Only two papers related to parallel BWT were found during a literature review; a software implementation using a parallel sorting stage [21], and a Field Programmable Gate Array (FPGA) hardware implementation [28]. Until recently with the introduction of multiple core processors, parallel computing hardware was not commonly found outside of large businesses and academic institutions. Even though the BWT can compress significantly better than the ZIP algorithm, the majority of people still

use ZIP for its speed and near universal compatibility amongst systems. The BWT achieves 30% better compression on text data and 21% better compression on the Canterbury Corpus than the common LZ77 algorithm used in ZIP [17]. These reasons along with the complexity of the BWT algorithm may explain why few people have chosen to research parallel implementations of the BWT algorithm.

The parallel message-passing BWT implementation investigated using a parallel suffix sort for the BWT algorithm [21]. A linear time algorithm is used to distribute suffices to each processor to sort in parallel. The load is balanced based on the size of data for each suffice. The results from each processor are collected and then sorted serially to form the final sorted data. The remaining stages of the BWT algorithm are performed sequentially. Using a message-passing implementation on a cluster, the best results achieved were a speedup of 10 using 30 processors compressing 30 MB of human chromosome data. When a 3 MB text file was tested, the maximum speedup obtained was 5.5 using 15 processors. The use of parallel suffix sorting to parallelize the BWT does not seem to scale well for large number of processors.

The parallel FPGA hardware BWT implementation also uses a parallel sorting technique to obtain speedup [28]. A sequential FPGA implementation of BWT was completed using a wave sorting algorithm for the suffix sorting stage of the BWT based on shift registers. The parallel version added a second level of shift registers for comparisons. All comparisons and swaps during the sort are performed in parallel. To test the performance of the algorithms, 9 random strings of 127 characters were extracted from a research paper. The results showed a speedup of 1.5 using the parallel FGPA algorithm compared to the sequential wave sort FPGA algorithm; not a significant increase.

Rather than performing further research on the parallel sorting aspect of the BWT algorithm, we chose to investigate different areas to parallelize the algorithm. We explore the use of data parallel and task parallel techniques with the BWT algorithm

to try and obtain speedup from parallel implementations.

4.2 Data Parallel BWT Compression Algorithm

Bzip2 is a lossless data compressor that uses the BWT compression algorithm. It is available as open-source software and is popular on Unix systems for compressing software distributions and source code such as the Linux kernel. With 8% lower compression on average, bzip2 also approaches the performance of the PPM statistical algorithms, the current best at compressing text data [9]. We use the BWT algorithm implementation in the libbzip2 library [32]. This approach gave access to well tested optimized code and allowed the output data to be compatible with the sequential version of bzip2. As a result, data compressed by using the data parallel algorithms presented in this paper can be decompressed using the original bzip2.

The BWT algorithm requires approximately 10 bytes of memory for every byte of data that is processed [33]. In order to compress large amounts of data, it first needs to be split up into manageable sizes to fit within the memory limitations of the system being used. This provides a natural separation of data into independent blocks that can be taken advantage of when parallelizing the BWT algorithm. Multiple blocks of data can then be processed with the BWT simultaneously to achieve speedup. Each block will have a variable processing time depending on the contents of the data being compressed. Finally, the compressed blocks can be concatenated together in the correct order to produce output data compatible with the sequential version of bzip2.

4.2.1 Sequential bzip2

Figure 4.1 shows how bzip2 compresses data. The sequential version of bzip2 by default splits the input data into multiple blocks of 900 kB each [33]. The block size can range from 100 kB to 900 kB. A block is filled 5 kB at a time until the block

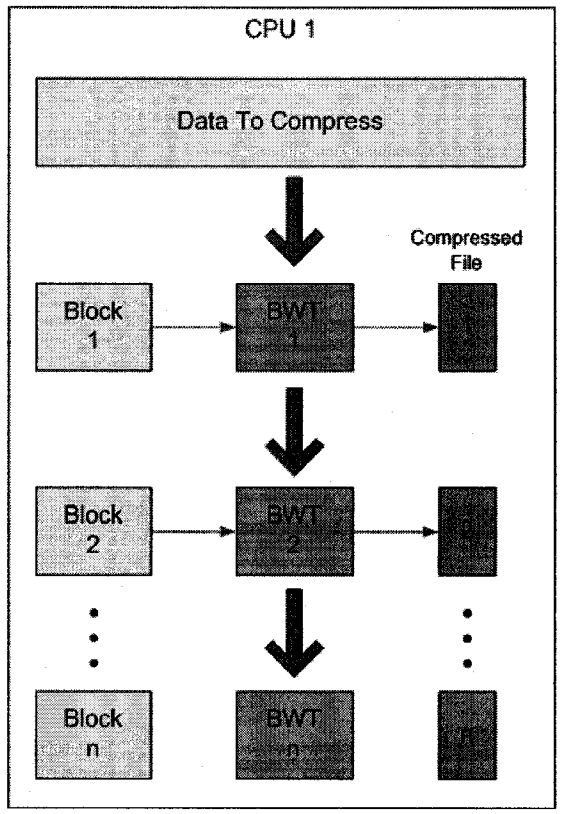

Figure 4.1: Bzip2 flow diagram

size or end of data is reached. That block of data is then processed with the BWT algorithm and the compressed results are output. These steps are repeated until all data has been read. Only one block of data is processed by bzip2 at a time.

4.2.2 Multi-threaded bzip2

We developed a data parallel multi-threaded version of bzip2, called pbzip2, designed to work on shared memory machines [18]. Similar to bzip2, input data is split into blocks of equal size, configurable by the user. Unlike bzip2 which reads 5 kB of data at a time, pbzip2 will read an entire block at once, reducing the amount of read calls required and increasing the performance slightly. The sequential bzip2 most likely uses such a small read buffer to better handle the ability to pipe data into the software from an external program. The pbzip2 software does not support pipes but

reads files directly from disk so can take advantage of using a larger read buffer to increase performance.

Task scheduling is achieved using a master/slave model; one master thread is created to read the input data. Figure 4.2 illustrates how the master thread assigns blocks of data to the slaves. Once a block of data is read, it is inserted into a FIFO (first in, first out) queue. The size of the queue is equal to the number of slave threads created. This size configuration gave a good balance between performance and memory requirements. A smaller queue size reduced performance as the slave threads had to wait for the queue to be populated. We also tested queue sizes of twice the number of processors and four times the number of processors. The larger queue sizes did not result in any performance increase but greatly increased the amount of memory required. A mutex is used to protect the queue so only one thread can modify the queue at a time.

```
WHILE inputFileBlock is not end-of-file
     READ inputFileBlock
     LOCK mutex FIFO_Queue
     WHILE FIFO_Queue is full THEN
          UNLOCK mutex FIFO_Queue
          CALL WaitForNotFull with FIFO_Queue
          LOCK mutex FIFO_Queue
     ENDWHILE
     PUT inputFileBlock into FIFO_Queue
     UNLOCK mutex FIFO_Queue
ENDWHILE
```

Figure 4.2: Pseudocode describing how the master thread assigns blocks of data to the slaves

Figure 4.3 illustrates the multi-threaded bzip2 algorithm. The dashed arrows indicate communication between separate processors while the solid arrows signify communication on the same processor. Once blocks start populating the queue, slave threads will remove one block of data and process it using the BWT compression algorithm. One master thread is created to manage input and one file writing thread

is created to manage output. A slave thread for each processor on the system is created unless otherwise specified by the user. After a slave thread has finished processing the data, the compressed block is stored in a global table along with a block id. A file writer thread takes the compressed data in the global table and uses the block id to write the output in the correct order. Since file writing is performed in a separate thread, the master thread does not need to wait for the write to finish and will continue to populate the FIFO queue until all input data has been read. When the slave threads have finished processing all blocks in the queue and the file writer thread has output all data in the correct order, the program terminates.

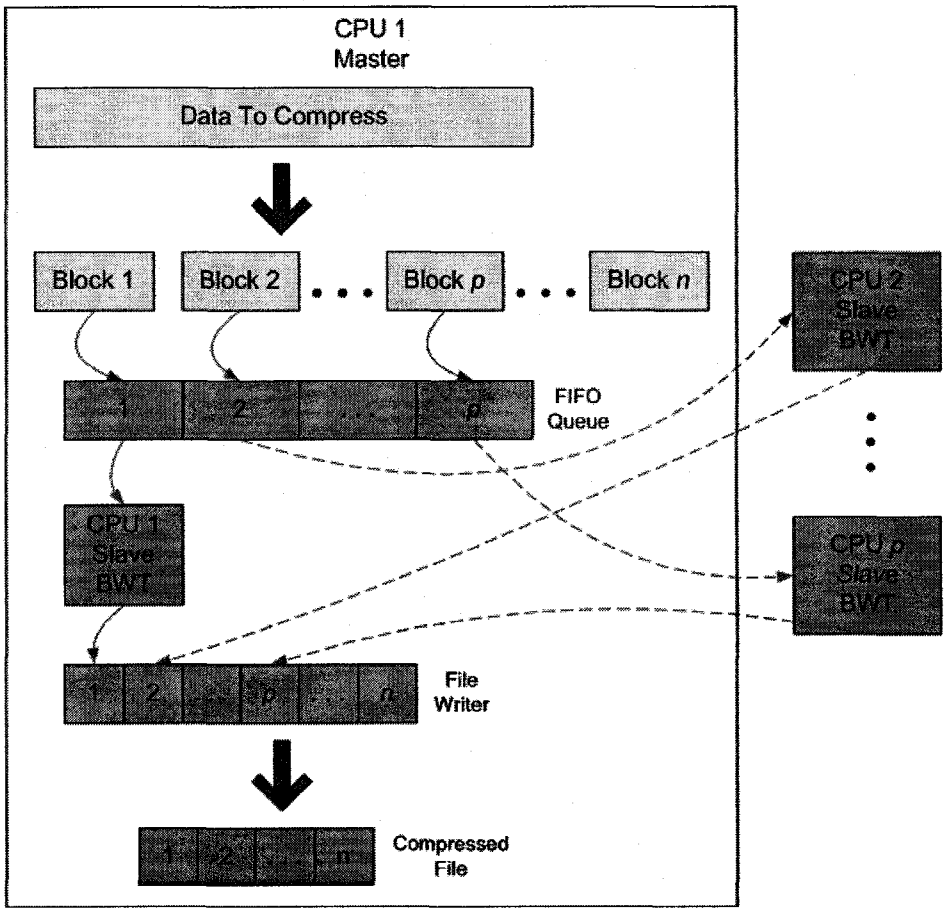

Figure 4.3: Multi-threaded bzip2 flow diagram

Each data block processed by pbzip2 is independent and requires very little com-

munication between processors; only to signal that the FIFO queue is empty or populated. Figure 4.4 shows how the slave threads retrieve a block of data to compress from the FIFO queue in memory and then write the compressed block into the file-writing queue in memory.

```
LOCK mutex FIFO_Queue
IF FIFO_Queue is not empty THEN
     GET next inputFileBlock from FIFO_Queue
     UNLOCK mutex FIFO_Queue
     CALL compressBWT with inputFileBlock RETURNING comprBlock
     LOCK mutex FileWriter_Table
     PUT comprBlock into FileWriter_Table
     UNLOCK mutex FileWriter_Table
ELSE
     UNLOCK mutex FIFO_Queue
ENDIF
```

Figure 4.4: Pseudocode showing how the slave threads process a block of data

4.2.3 Message-passing bzip2

Next, we developed a data parallel message-passing version of bzip2, called mpibzip2, designed to work on cluster machines. It uses the Message Passing Interface (MPI) standard [24] and links to the libbzip2 library [32] for access to the BWT algorithms. Similar to bzip2, input data is split into blocks of equal size, configurable by the user.

For task scheduling, a master/slave model is adapted; one master processor is used to read the input data. Figure 4.5 illustrates the flow of the message-passing bzip2 algorithm. The dashed arrows indicate communication between separate processors while the solid arrows signify communication on the same processor. The pseudocode in Figure 4.6 shows once a block of data is read, the master processor will send it to the first available idle slave processor. When the master receives a compressed block from a slave, it is stored in a global table along with a block id. A file writer thread running on the master processor takes the compressed data in the global table

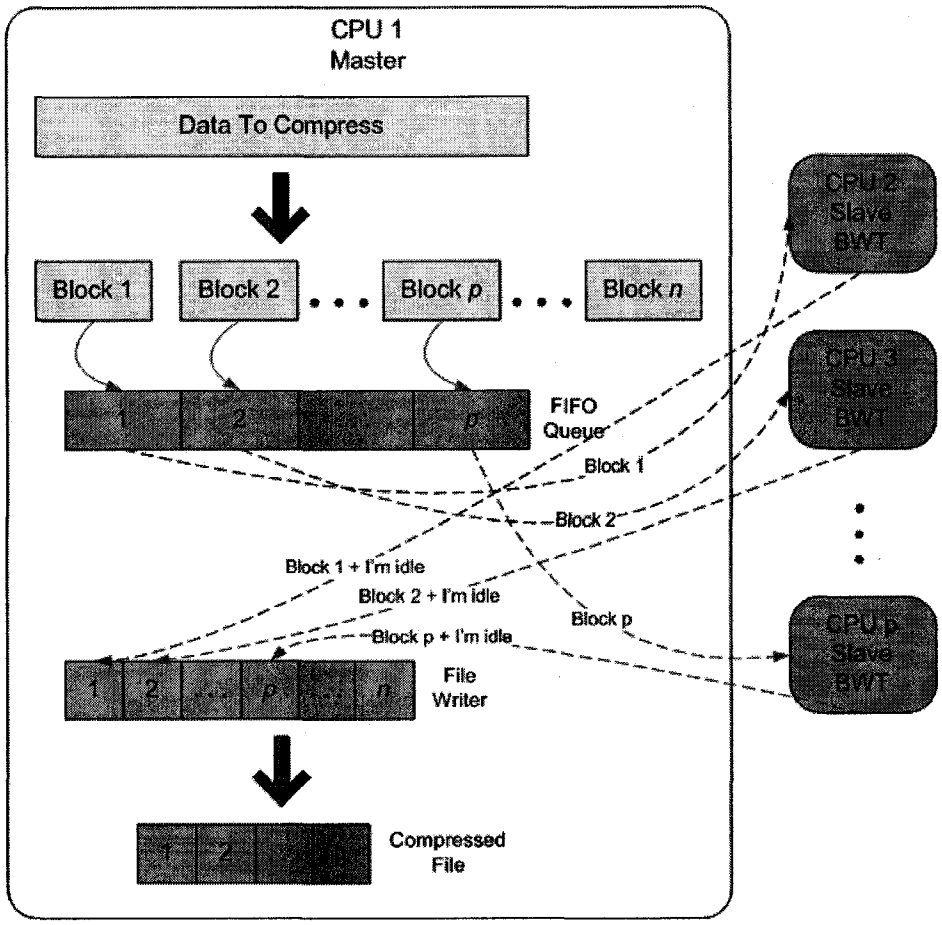

Figure 4.5: Message-passing bzip2 flow diagram

and uses the block id to write the output in the correct order. Since file writing is performed in a separate thread, the master processor does not need to wait for the write to finish and will continue to prepare blocks until all input data has been read. Figure 4.7 describes how slaves will process data using the BWT compression algorithm and send the results back to the master, signaling that it is now idle. When the slaves have finished processing all blocks received from the master and the file writer thread on the master has output all data in the correct order, the program terminates.

Each data block processed by mpibzip2 is independent but requires both the uncompressed and compressed data to be transmitted over the network from the

```
WHILE intputFileBlock is not end-of-file
     READ inputFileBlock
     CALL WaitForIdleSlave RETURNING comprBlock, blkSize, blkID, slaveID
     IF blkSize is not 0 THEN
          LOCK mutex FileWriter_Table
          PUT comprBlock into FileWriter_Table
          UNLOCK mutex FileWriter_Table
     ENDIF
     CALL SendBlock with inputFileBlock, blkSize, blkID, slaveID
     SET slaveID in slaveBusyArray to busy
ENDWHILE
WHILE slaveBusyArray contains a busy slave
     CALL WaitForIdleSlave RETURNING comprBlock, blkSize, blkID, slaveID
     IF blkSize is not 0 THEN
          LOCK mutex FileWriter_Table
          PUT comprBlock into FileWriter_Table
          UNLOCK mutex FileWriter_Table
     ENDIF
     SET slaveID in slaveBusyArray to not busy
     SET blkSize to 0
     CALL SendBlock with inputFileBlock, blkSize, blkID, slaveID
ENDWHILE
```

Figure 4.6: Pseudocode describing how the master assigns blocks to slaves and receives processed data

master to slaves and back. Since the block processing time is variable, the software was benchmarked using various data block sizes to see how it affected load balancing.

4.3 Task Parallel BWT Compression Algorithm

The bzip2 implementation of the BWT compression algorithm is not modular but designed for single data streams. In order to break apart the different components of the algorithm to use in a software pipeline, the bzip2 library would have to be completely re-written. To test the performance of a task parallel version of the BWT compression algorithm, another implementation was chosen instead. Bwtzip [25] is a lossless data compressor that uses the BWT compression algorithm. It is a highly modular open-source implementation of the BWT algorithm designed to be used for

```
SET allDone to 0
SET blockSize to 0
CALL SendBlock with comprBlock, blockSize, blockID, masterID
WHILE allDone is not 1
     CALL WaitForBlock RETURNING inputFileBlock, blockSize, blockID
     IF blockSize is 0 THEN
          SET allDone to 1
     ELSE
          CALL compressBWT with inputFileBlock RETURNING comprBlock
          CALL SendBlock with comprBlock, blockID, masterID
     ENDIF
ENDWHILE
```

Figure 4.7: Pseudocode describing how the master assigns blocks to slaves and receives processed data

research purposes. While compressed output sizes are similar, bwtzip has not been as widely adopted as bzip2 since it is less optimized and runs approximately 5 times slower. The modular design of bwtzip software allowed us independent access to the various stages of the BWT which we required to design the software pipeline. As with bzip2, each block will have a variable processing time depending on the contents of the data being compressed.

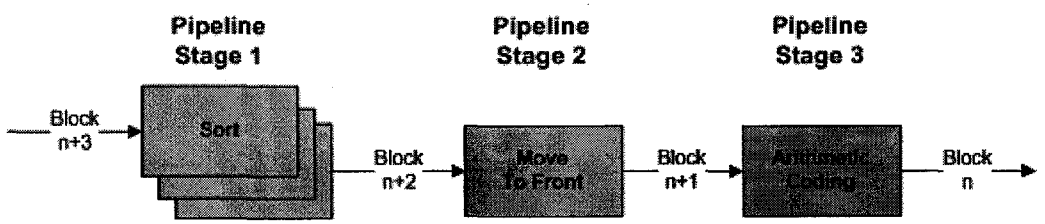

Figure 4.8: BWT pipeline implementation

Recall that the Burrows-Wheeler-Transform compression algorithm consists of three steps: sorting, move to front, and arithmetic compression. Figure 4.8 shows these three steps can be implemented in a software pipeline. Algorithmic multiplexing is used to separate the BWT algorithm into three stages. Since stage 1 (*sorting*) takes the most time, the performance of a simple pipeline would be limited by the

time it takes for the sorting stage to complete. To reduce this latency, geometric multiplexing can be applied to run multiple sorting stages simultaneously. When a stage 1 processor finishes sorting the data, it will send the data to stage 2 (*move to front*) and immediately start processing the next block. While stage 1 processes the second block, stage 2 is using temporal multiplexing to process the first block at the same time. After this data is processed it is passed on to stage 3 (*arithmetic coding*). The stage 2 processor will then start a new block of data while the stage 3 processor is working on the block of data it just received. The software pipeline implementation of the BWT algorithm allows multiple processors to be used which should increase throughput and achieve speedup compared to the sequential version.

4.3.1 Sequential bwtzip

The sequential version of bwtzip splits the input data into multiple blocks of user defined size. A block of data is read from the input file and then processed with the BWT algorithm and the compressed results are output. These steps are repeated until all data has been read. Only one block of data is processed by bwtzip at a time.

The BWT portion of the bwtzip software can be split into four distinct tasks as shown in Figure 4.9. The first is the sorting stage of the BWT which is implemented using an $O(n)$ time Ukkonen suffix tree sort [38]. The second task implements the move to front (MTF) stage of the BWT. The third task is a zero length encoder (ZLE) that compresses the repeated sequence of bytes containing 0 which commonly occur after the sorting and MTF stages of BWT. Finally, an adaptive arithmetic encoder task implements the compression stage of the BWT. The MTF, ZLE, and arithmetic coding algorithms used in bwtzip have linear run times [25].

When comparing the two sequential implementations of the BWT algorithm, the compression performance of bwtzip is similar to bzip2. A 159 MB test file was compressed to 30 MB with bwtzip and 29 MB with bzip2. The processing time however is

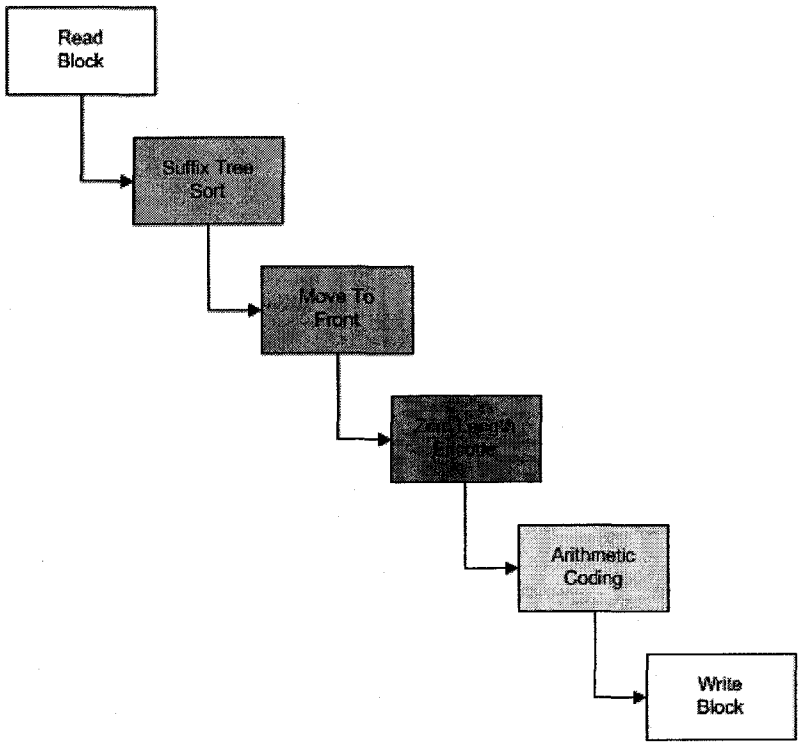

Figure 4.9: Sequential bwtzip task flow diagram

considerably slower. The same test file took bwtzip 3 minutes 18 seconds to process while bzip2 took only 38 seconds. Memory requirements are also higher for bwtzip, needing 85 MB of memory to process the test file when bzip2 used 14 MB. The less optimized and more modular nature of bwtzip allows it to be modified more easily at the expense of speed and memory usage.

4.3.2 Parallel bwtzip

We developed a task parallel message-passing version of bwtzip, called mpibwtzip, designed to work on cluster machines. It uses the Message Passing Interface (MPI) standard [24] and is based on the bwtzip source code. The parallel algorithm uses the four distinct tasks in the BWT implementation to create a software pipeline. Based on profiling results with the sequential bwtzip software, two different pipeline designs were created. In the first design, the four tasks were grouped into a three stage

pipeline. The first stage contained the suffix tree task, the second stage contained both the MTF and ZLE tasks, and the third stage contained the arithmetic coding task. The suffix tree stage uses approximately 84%, the MTF+ZLE stage 8%, and the arithmetic coding stage 8% of the time required to process a block of data. When the parallel algorithm is initialized, one processor will be reserved for the master, 8% of the processors assigned to perform the MTF+ZLE stage, 8% available for the arithmetic coding stage, and the remaining processors designated for suffix tree stage. Figure 4.10 illustrates the 3-stage pipeline message-passing bwtzip algorithm using 30 processors with 1 reserved for the master, 2 assigned to the MTF+ZLE task, 2 assigned to the arithmetic coding task, and the remaining 25 will be used for the suffix tree task. The dashed arrows indicate communication between separate processors while the solid arrows signify communication on the same processor. The 25 suffix tree processors (CPU 2 - CPU 26) will contact the master processor (CPU 1), asking for work. Half of those stage 1 processors will send their processed blocks to one of the two stage 2 processors (CPU 27), and the rest to the other stage 2 processor (CPU 28). The first stage 2 processor (CPU 27) will send its results to the first stage 3 processor (CPU 29) while the second stage 2 processor (CPU 28) will send its results to the second stage 3 processor (CPU 30). Both stage 3 processors will send the final results back to the master processor (CPU 1). Each block processed by mpibwtzip requires the data to be transmitted over the network from the master to each of the three stages of the pipeline and back; a total of 4 hops.

For the second design, a two stage pipeline was used. This reduces the amount of network usage to process data at the cost of having less overlapping between tasks. The first stage contained the suffix tree task and the second stage contained the MTF, ZLE, and arithmetic coding tasks. The suffix tree stage uses approximately 84% and the remaining stage 16% of the processing time for a block. Similar to the first design, one processor is reserved for the master while the remaining processors are divided

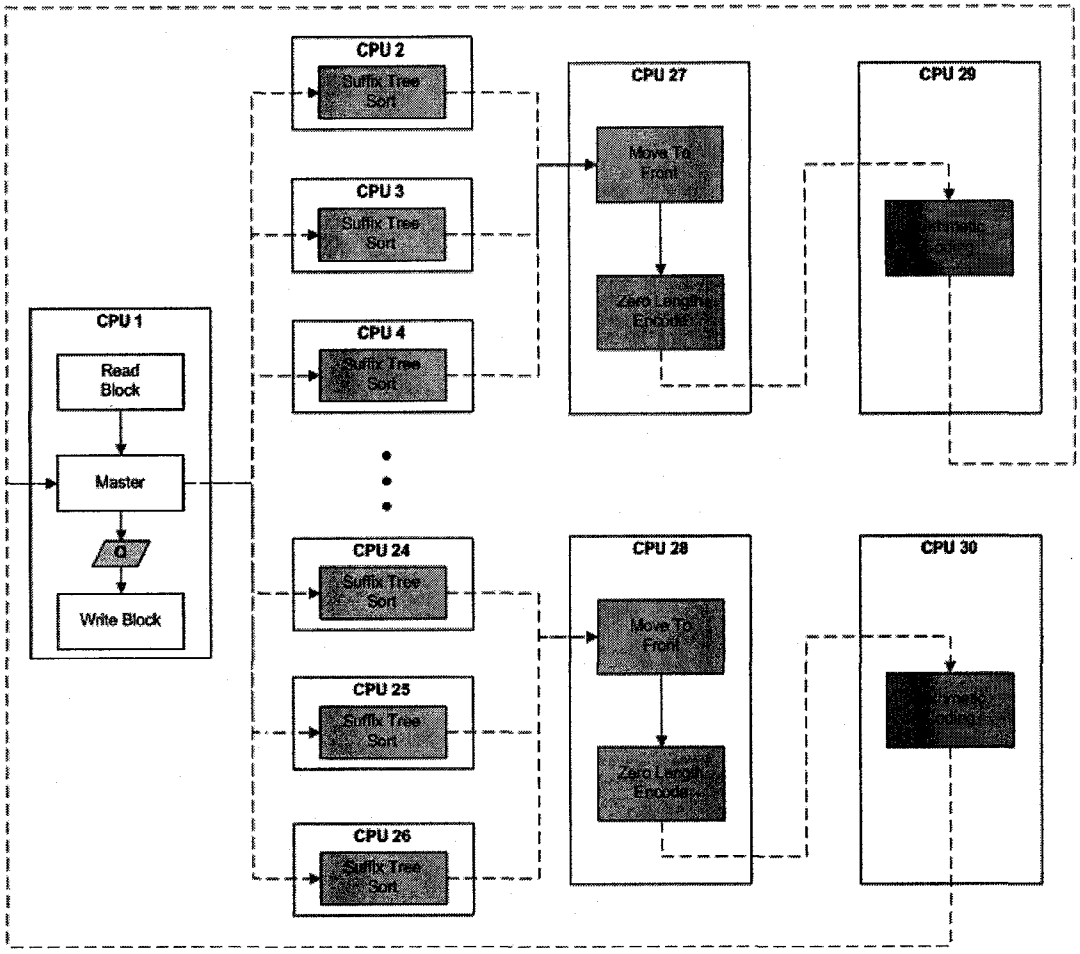

Figure 4.10: Message-passing bwtzip 3-stage pipeline flow diagram with 30 processors

between the stages using the 84% and 16% split.

With the sequential bwtzip algorithm, all four tasks had to complete processing the current block of data before the next block could start being processed. With the task parallel software pipeline approach, the stages can be overlapped. Once a stage in the pipeline completes a block, it passes it on to the succeeding stage and can start processing the next block immediately. This should allow the algorithm to achieve speedup with multiple processors.

Similar to bwtzip, input data is split into blocks of equal size, configurable by the user. For task scheduling, a master/slave model is adapted; one master processor is used to read the input data. Once a block of data is read, the master processor will

put the block into the software pipeline by sending it to the first available idle suffix tree slave processor. That slave will process the data using the suffix tree sorting algorithm and send the results on to the next stage in the pipeline. Since the suffix tree stage is now complete, it will contact the master, signaling that it is now idle and ready for more work. With the 3-stage pipeline design, once the block of data in the MTF+ZLE stage is complete, it will be sent to the arithmetic coding stage which is the final stage in the pipeline. The MTF+ZLE stage is now available to process the next block. After the arithmetic coding stage is complete, the final compressed block is sent back to the master processor, making the stage available again for the next block from the MTF+ZLE stage. In the 2-stage design, once the block of data in the MTF+ZLE+Arithmetic coding stage is complete, the final compressed block is sent back to the master processor, making the stage available for the next block from the suffix tree stage.

When the master receives a compressed block, it stores it in a global table along with a block id. A file writer thread running on the master processor takes the compressed data in the global table and uses the block id to write the output in the correct order. Since file writing is performed in a separate thread, the master processor does not need to wait for the write to finish and will continue to prepare blocks until all input data has been read. When the pipeline has finished processing all blocks received from the master and the file writer thread on the master has output all data in the correct order, the program terminates. Figure 4.11 shows the 2-stage pipeline message-passing bwtzip algorithm using 15 processors with 1 reserved for the master, 2 assigned to the MTF+ZLE+Arithmetic coding task, and the remaining 12 will be used for the suffix tree task. The dashed arrows indicate communication between separate processors while the solid arrows signify communication on the same processor. The 12 suffix tree processors (CPU 2 - CPU 13) will contact the master processor (CPU 1), asking for work. Half of those stage 1 processors will send their

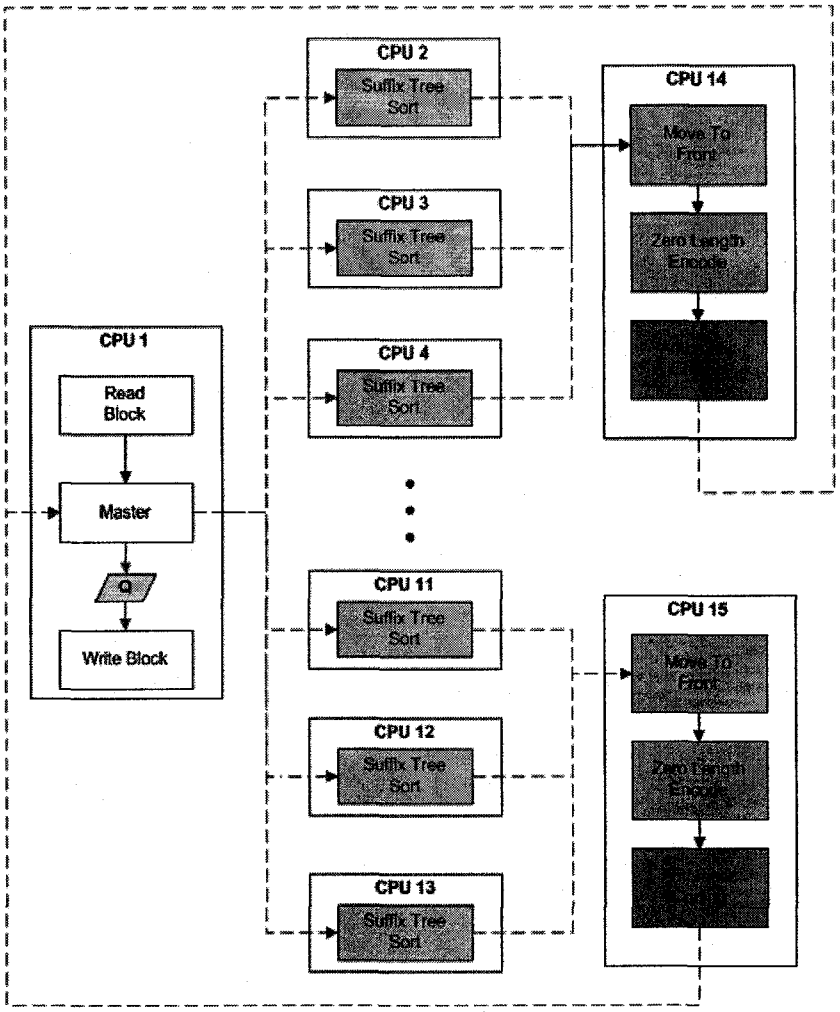

Figure 4.11: Message-passing bwtzip 2-stage pipeline flow diagram with 15 processors

processed blocks to the first stage 2 processor (CPU 14), and the rest to the second stage 2 processor (CPU 15). Both stage 2 processors will send the final results back to the master processor (CPU 1). With the shorter pipeline version, the data only requires 3 hops on the network for processing.

4.4 Summary

In this chapter we reviewed the current literature on parallel BWT implementations. We also described two different designs for parallelizing the BWT compression

algorithm. The first involves parallelizing the data stream with multi-threaded and message-passing algorithms based on bzip2. The second technique decomposes the BWT algorithm into multiple tasks to create a software pipeline with a message-passing algorithm based on bwtzip.

In the next chapter, we present the experimental results and analysis for both the data parallel and task parallel implementations of the BWT data compression algorithm.

Chapter 5

Parallel BWT Performance Evaluation

In this chapter, we present the performance evaluation and experimental results for both the data parallel and task parallel implementations of the Burrows-Wheeler Transform (BWT) lossless data compression algorithm.

5.1 Experimental Setup

All experiments measured the wall clock times including the time taken to read from input files and write to output files. The only NUMA and cluster systems available for testing were shared amongst many researchers, which resulted in the presence of unrelated tasks on the same machine such as other user research software processes and system processes. While exclusive access to the systems was not possible, a queue is used to schedule jobs so that each job is given exclusive access to the number of processors they require on the system. Each experiment was carried out 40 times, using the average as the result to help negate any effects of unrelated tasks on the system. The variance of the results was small indicating that unrelated tasks did not have a major impact on the experimental results.

The multi-threaded parallel algorithms were tested on a 128 processor shared memory system, while the message-passing parallel algorithms were tested on a 100 processor cluster. Since the block processing time is variable, block sizes of 100 kB, 900 kB, 1800 kB, and 3600 kB were used in the experiments to determine how block size affected load balancing and performance. The 100 kB block size is the smallest supported by bzip2, while 900 kB is the default block size. Two (1800 kB) and four (3600 kB) times the default size was chosen to compare the difference when using a smaller number of blocks with less reads and writes. Typically with the BWT compression algorithm, increases in block size provide smaller compressed output files with diminishing returns. Extremely large block sizes would also use extremely large amounts of memory that could saturate the memory bus and cause the operating system to start using the swap file on disk, greatly slowing down the performance. Due to diminishing returns, the test data compressed using 3600 kB blocks was only 1% smaller than using 900 kB blocks.

Two different input files were used for testing the performance of the multi-threaded and message-passing versions of bzip2. The first is the Linux 2.4.23 kernel source code [37] totaling 159 MB uncompressed. The bzip2 software can compress this test file using 900 kB blocks down to 28 MB, only 18% of its original size. The second is a binary database of elliptic curve distinguished points [3] that is 1875 MB uncompressed. This data set contains less redundant data than text, so bzip2 can only compress this test file using 900 kB blocks down to 1571 MB, 84% of its original size. These two data sets are a good barometer for how the parallel bzip2 algorithms should perform in general. One contains text which the BWT algorithm is optimized for and the other is binary data. The two file sizes are significantly different and each one exercises different parts of the BWT algorithm.

5.2 Experimental Results for Data Parallel Algorithms

The multi-threaded pbzip2 program was compiled using ICC 9.0 on the SGI Altix 3700 system with the libbzip2 1.0.2 library [32]. The message-passing mpibzip2 program was compiled using PathScale 2.2.1 on the HP Opteron 275 cluster with the libbzip2 1.0.2 library.

5.2.1 Multi-threaded bzip2

The pbzip2 software was tested on an SGI Altix 3700 Bx2 shared memory system with 128 Itanium2 processors running at 1.6 GHz, 6 MB L3 cache, and 256 GB system RAM. The space overhead due to parallelization on SMP systems ranges between 2 MB per processor with 100 kB blocks to approximately 70 MB per processor with 3600 kB blocks. Considering that single processor systems commonly have 512 MB to 1 GB of memory, and multi-processors systems contain more memory to handle the larger workloads, the overhead imposed is reasonable. For example, using all 128 processors on the SGI Altix system with 3600 kB blocks, the multi-threaded algorithm uses a maximum of 8.75 GB of memory out of the available 256 GB.

SGI Altix 3700

Table 5.1 shows the processing times for compressing the 159 MB Linux kernel source test data on the Altix 3700 system. The 159 MB data was compressed using pbzip2 with 1 processor and 100 kB blocks in 34.77 seconds, 900 kB blocks in 37.54 seconds, 1800 kB blocks in 37.65 seconds, and 3600 kB blocks in 37.66 seconds. Using 100 kB blocks, maximum performance was achieved using 40 processors in 1.74 seconds. With 900 kB blocks, performance peaked at 40 processors needing only 1.35 seconds. The fastest performance for 1800 kB blocks was obtained with 40 processors in 1.97 seconds and for 3600 kB blocks with 30 processors in 2.02 seconds. Using more than

40 processors, the performance leveled off and no further improvements were seen.

Processors	159 MB Data Set Time (sec)			
	100 kB	900 kB	1800 kB	3600 kB
1	34.77	37.54	37.65	37.66
2	17.56	18.89	19.04	19.10
5	7.17	7.79	7.75	7.85
10	3.70	3.96	4.06	4.02
15	2.59	2.77	2.86	3.17
20	2.08	2.12	2.37	2.57
25	1.80	1.83	2.20	2.45
30	1.79	1.55	2.04	2.02
40	1.74	1.35	1.97	2.06
50	2.33	1.38	2.13	2.08

Table 5.1: Processing times with 159 MB data for multi-threaded pbzip2 on SGI Altix 3700

Figure 5.1 shows speedup as a function of the number of processors. Speedup (S_p) is the ratio of the runtime of the sequential BWT algorithm (T_1) using 1 processor to that of the parallel BWT algorithm (T_p) using p processors on the same system such that $S_p = \frac{T_1}{T_p}$ [29]. When the 159 MB input file was tested, significant speedup was observed compared to the sequential version of bzip2. The best overall result for the 159 MB data was obtained with 900 kB blocks. For the other block sizes, speedup was similar but lower than with 900 kB blocks. Furthermore, with 3600 kB blocks, the 159 MB data set can only be split into 47 blocks. Speedup peaks before 50 processors because extra threads would not have any data to work on and sit idle. All block sizes performed fairly equally with less than 10 processors and speedup levelled off at 40 processors. This is due to a combination of the FIFO queue becoming a bottleneck and most likely slave threads blocking more often while waiting for memory buffers to be allocated. Any thread within a POSIX multi-threaded process that requires new memory will block while another is allocating memory using *malloc* or *new*. One possible way to improve performance might be to design the parallel version

of the algorithm so that it does not require dynamic memory allocation for buffers but instead allocated the required memory at the beginning so threads will not have to block. While this might improve performance, it would also increase the memory requirements of the algorithm as well. Careful planning and testing would be required to balance performance with system resources.

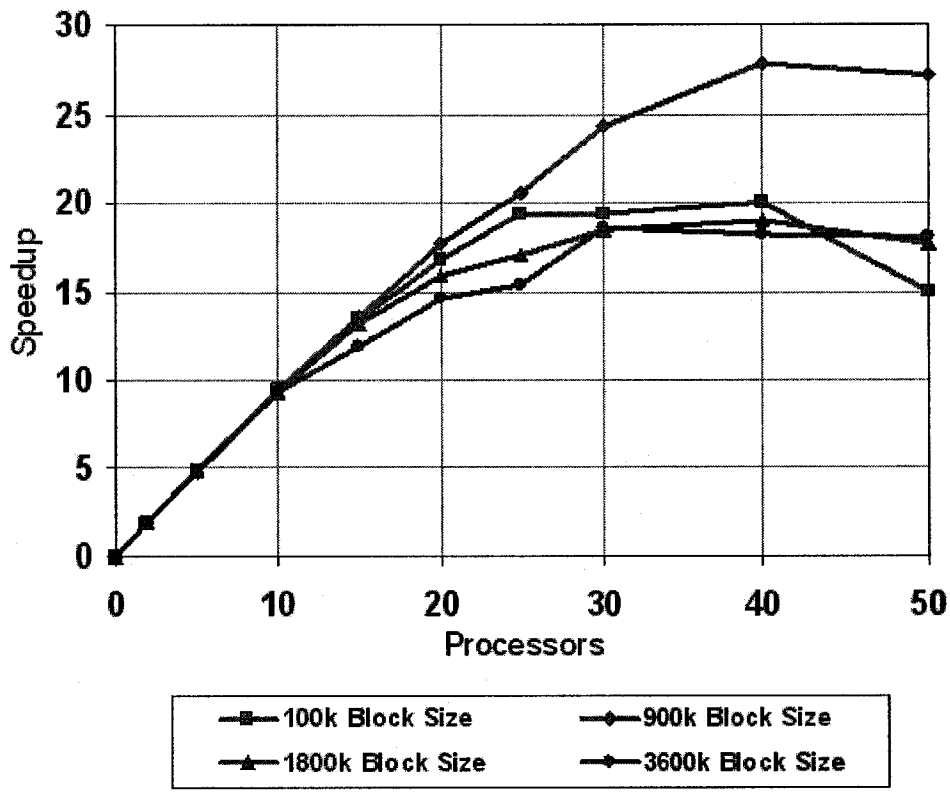

Figure 5.1: Multi-threaded pbzip2 speedup with 159 MB data on SGI Altix 3700

Table 5.2 shows the processing times for compressing the 1875 MB test data. The performance gains with the larger test file was even more impressive due to the longer processing time required. Pbzip2 with 1 processor and 100 kB blocks compressed the data in 952.71 seconds, 900 kB blocks in 955.05 seconds, 1800 kB blocks in 957.34 seconds, and 3600 kB blocks in 954.52 seconds. With 100 kB blocks, maximum performance was achieved using 70 processors in 23.03 seconds. Speedup leveled off at 60 processors being only 1.17 seconds slower than 100 processors. With 900 kB

blocks, performance peaked at 110 processors requiring only 12.77 seconds. The best processing time was obtained for 1800 kB blocks with 110 processors in 10.26 seconds and for 3600 kB blocks with 110 processors in 10.28 seconds.

Processors	1875 MB Data Set Time (sec)			
	100 kB	900 kB	1800 kB	3600 kB
1	952.71	955.05	957.34	954.52
2	477.23	480.53	480.25	477.87
5	191.57	193.00	193.03	192.59
10	96.65	96.63	97.41	97.02
15	65.18	65.13	65.07	65.39
20	51.99	48.76	49.03	49.61
25	40.01	39.50	39.37	40.26
30	32.87	32.87	33.24	33.82
40	25.75	25.26	25.40	25.35
50	26.01	20.30	20.47	20.46
60	24.02	17.14	17.51	18.05
70	23.03	16.44	15.20	15.22
80	24.88	15.03	13.26	13.78
90	24.42	13.73	11.98	12.69
100	24.26	13.40	10.98	11.46
110	26.19	12.77	10.26	10.28
120	25.81	12.91	10.79	10.32

Table 5.2: Processing times with 1875 MB data for multi-threaded pbzip2 on SGI Altix 3700

Figure 5.2 shows a significant speedup was also observed when the 1875 MB database file was tested. The best overall results for the 1875 MB data was with 1800 kB blocks and 3600 kB blocks. The pbzip2 results for this data shows near-linear speedup, staying within 90% of linear speedup with 100 kB blocks up until 40 processors, 900 kB blocks up until 60 processors, 1800 kB blocks up until 80 processors, and 3600 kB blocks up until 70 processors. The lower performance of the 100 kB block size is due to the faster processing blocks which cause the FIFO queue to become a bottleneck more quickly than the larger block sizes. Blocking for allocation of memory as discussed previously may also be a factor for the lower performance.

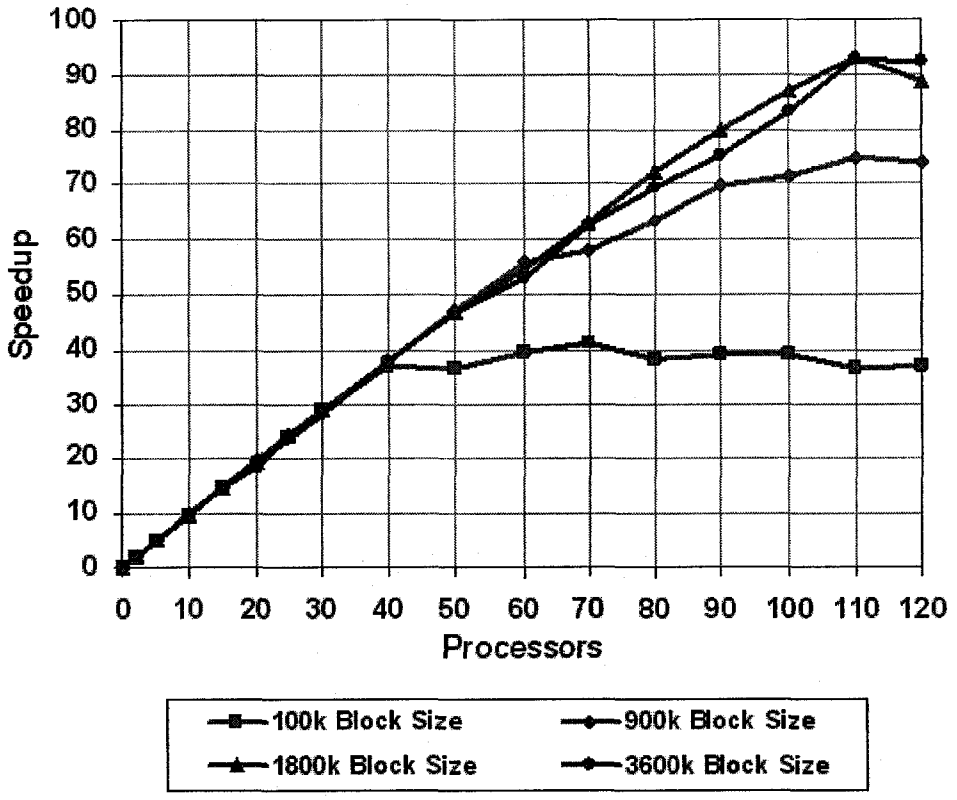

Figure 5.2: Multi-threaded pbzip2 speedup with 1875 MB data on SGI Altix 3700

The multi-threaded pbzip2 software achieved greater speedup with the 1875 MB database than with the 159 MB kernel source. Table 5.3 shows the processing time per block for each of the block sizes with the two data sets. The longer processing time of the larger test file allowed the speedup to scale to a higher number of processors. The contents of the data also differed greatly between the two test files. When the average processing time per block was measured, blocks of data in the 159 MB test file were found to be compressed by the BWT algorithm approximately 2 times faster than with blocks from the 1875 MB database. The variance in block processing times is significantly larger with the 159 MB data set compared to same block size with the 1875 MB data set.

As the number of processors increased, requests for data from the FIFO queue occurred at greater frequency with the 159 MB data causing more threads to block

Data Set	Block Size (kB)	Number Blocks	Time Per Block (ms)			
			Max	Min	Avg	Variance
159 MB	100	1667	73.74	0.19	20.07	18.83
	900	186	732.20	19.30	201.55	4639.89
	1800	93	1258.88	205.47	398.31	13882.52
	3600	47	1601.55	206.10	787.88	39936.61
1875 MB	100	19668	48.72	10.90	47.86	0.10
	900	2186	441.82	102.34	437.37	52.77
	1800	1093	893.03	538.51	873.11	105.83
	3600	547	1777.89	542.02	1742.65	2665.76

Table 5.3: Processing times per block for multi-threaded pbzip2 on SGI Altix 3700

while waiting for access to the queue. This led to the FIFO queue becoming a bottleneck faster with the 159 MB data set than with the 1875 MB one. Figure 5.3 shows the effects of the FIFO queue wait time for threads as the number of processors are increased. The increase in wait times for the 159 MB file is significantly higher than the 1875 MB test file since the bottleneck occurs more quickly. The queue wait times for the 159 MB data increase rapidly above 40 processors for all block sizes showing the FIFO queue is the bottleneck when the speedup levels off. The wait time for the 3600 kB block size with the 159 MB data set levels off above 50 processors as there are only 47 blocks to process and the extra processors sit idle and are unused. The larger block sizes show longer queue wait times than the smaller block sizes. This is expected as the larger blocks take longer to read, write, and manage, the slave threads wait longer for the master to prepare the data and place it into the queue.

The data from the input file is read in sequentially from disk and fed into the FIFO queue, which would lead to another bottleneck with a large number of processors. Finally, once the data has been compressed it is merged in the correct order and written to disk sequentially. Since disk reads and writes are counted in the benchmark time, the faster compressing 159 MB kernel source would have created a larger backlog of data to be written to disk than with the 1875 MB database test file as the number of processors increased.

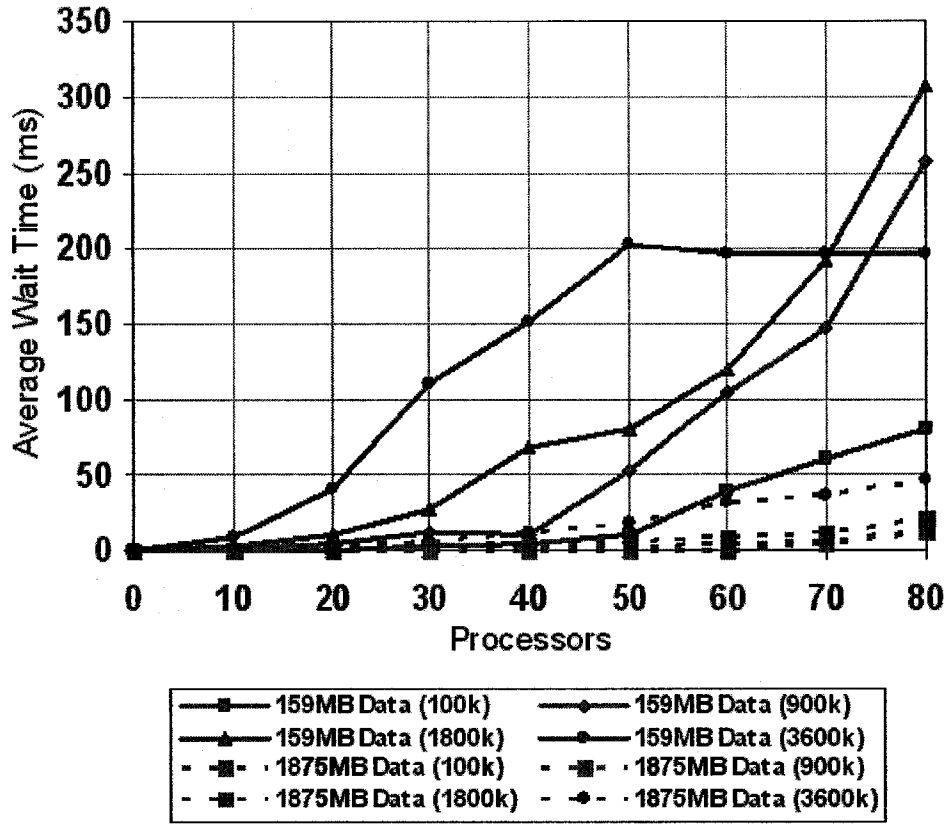

Figure 5.3: Multi-threaded pbzip2 average FIFO queue wait time on SGI Altix 3700

With the SGI Atlix system, processor utilization can be obtained from the job scheduling software. Figure 5.4 shows processor utilization for the 159 MB and 1875 MB data sets as the number of processors increase. The utilization of the 159 MB data set decreases more rapidly than with the 1875 MB data set which results in the lower speedup obtained. The longer block processing times of the 1875 MB data set allow for more efficient processor utilization resulting in better maximum speedup.

Inexpensive SMP Systems

One of the goals of this thesis was to provide a parallel algorithm that would be useful on computer systems that the general population own. Now that dual-core processors are becoming common place and quad-core processors manufactured in the near future, an inexpensive system could contain up to 4 processors. In order to

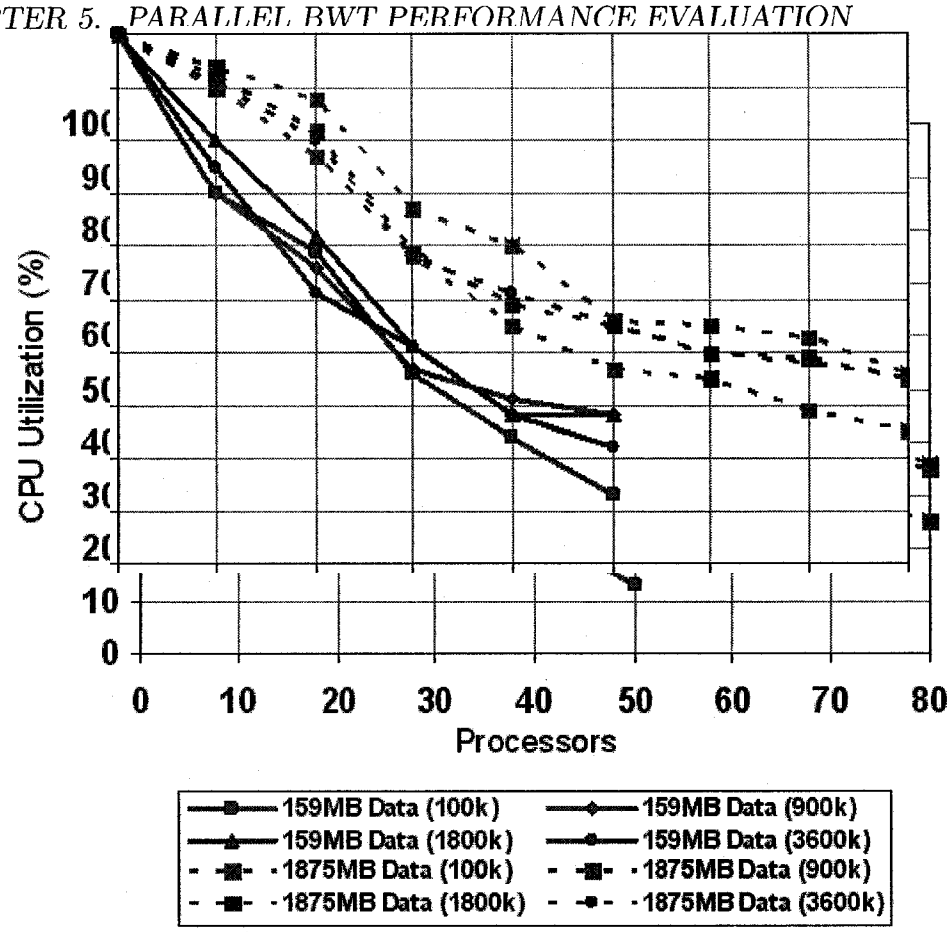

Figure 5.4: Multi-threaded pbzip2 processor utilization on SGI Altix 3700

determine if the multi-threaded algorithm would be useful for consumer-level systems, the pbzip2 software was tested on three relatively inexpensive systems. The test systems include an AMD Athlon-MP2600+ system with two 2.1 GHz processors and 1 GB of system RAM running Windows XP, an Intel Pentium4 Xeon system with two 2.8 GHz processors and 1 GB of system RAM running Linux 2.6.9 and finally an AMD Opteron 275 system with four 2.2 GHz processors (two dual-core CPUs) and 8 GB of system RAM running Linux 2.6.9. The pbzip2 software was compiled using the GCC version 3 compiler. The 159 MB Linux kernel source test file was used for benchmarking, as this type of file is more likely to be processed on a consumer-level machine. The processing times for compressing the test data on each system are shown in Table 5.4.

Processors	Pentium4 Xeon Time (sec)	Athlon MP-2600+ Time (sec)	Opteron 275 Time (sec)
1	67.6	98.6	39.0
2	40.7	55.9	19.8
3	-	-	13.6
4	-	-	10.6

Table 5.4: Processing times with 159 MB data for multi-threaded pbzip2 on inexpensive systems

Figure 5.5 shows that even on consumer-level systems, near-linear speedup is achieved using the multi-threaded BWT algorithm. The Opteron 275 with its superior memory bus had the best performance, followed closely by the older Athlon MP-2600 system and the Pentium4 Xeon system. The AMD systems use a HyperTransport bus which allows it to scale better than the Intel Pentium4 system with multiple processors.

5.2.2 Message-passing bzip2

The message-passing mpibzip2 software was tested using up to 100 processors on a cluster of 25 HP Opteron 275 dual-core dual-processor systems running at 2.2 GHz with 1 MB L2 cache and 8 GB system RAM. The nodes were connected with Myrinet interconnect and the processors within a node are connected with AMD's HyperTransport bus technology. Since the message-passing algorithm uses a master/slave model a minimum of two processors are required, one for the master to manage the work, and at least one slave to process the work. With a cluster environment, the memory requirements are spread out amongst the nodes in the cluster. The space overhead due to parallelization is still 70 MB per processor with 3600 kB blocks like the multi-threaded algorithm. Each node in the cluster contains 4 processors so requires only 280 MB out of the available 8 GB of system memory.

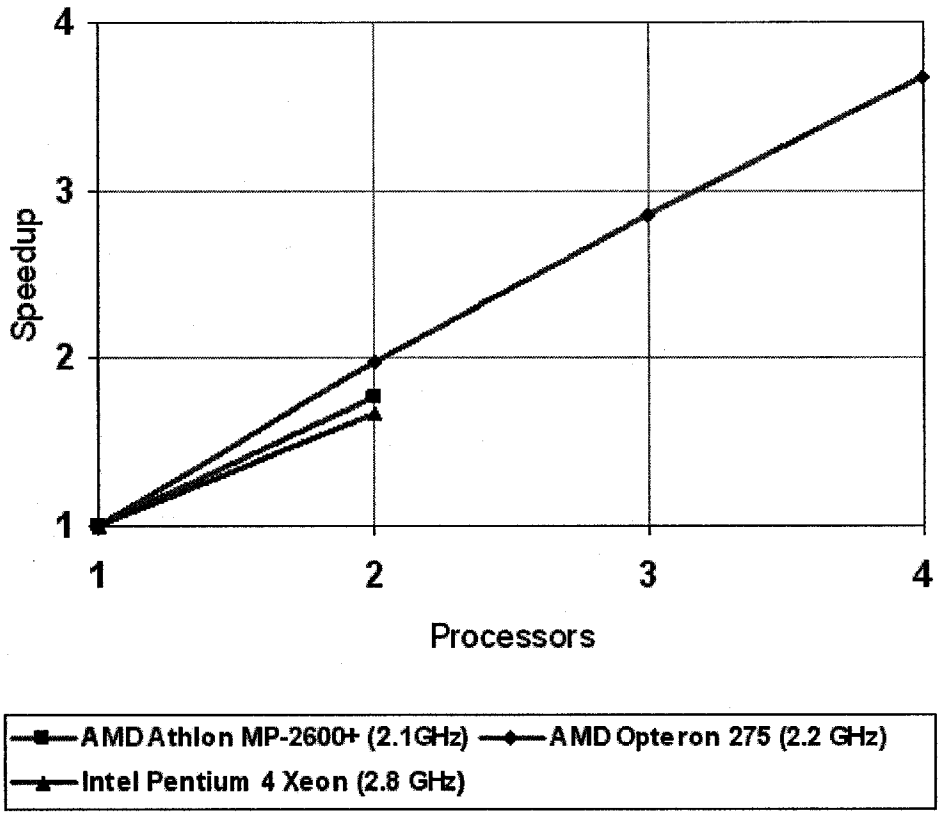

Figure 5.5: Multi-threaded pbzip2 speedup with 159 MB data on inexpensive SMP systems

HP Opteron 275 Cluster

Processing times for the 159 MB test data on the HP Opteron cluster are shown in Table 5.5. The 159 MB data was compressed using mpibzip2 with 2 processors (one master, one slave) and 100 kB blocks in 29.88 seconds, 900 kB blocks in 38.14 seconds, 1800 kB blocks in 37.97 seconds, and 3600 kB blocks in 37.64 seconds. The fastest processing time was achieved with mpibzip2 taking only 1.18 seconds using 50 processors with 100 kB blocks, 1.48 seconds using 50 processors with 900 kB blocks, 2.29 seconds using 50 processors with 1800 kB blocks, and 2.57 seconds using 40 processors with 3600 kB blocks. Using more than 40 processors, the performance leveled off and no further improvements were seen. With all block sizes, using only 20 processors was less than 1 second slower than the maximum performance.

Processors	159 MB Data Set Time (sec)			
	100 kB	900 kB	1800 kB	3600 kB
2	29.88	38.14	37.97	37.64
5	7.68	9.84	9.83	9.87
10	3.68	4.53	4.55	4.67
15	2.33	3.07	3.41	3.43
20	1.90	2.29	2.80	3.10
25	1.55	2.01	2.76	2.99
30	1.36	1.70	2.50	2.60
40	1.20	1.55	2.36	2.57
50	1.18	1.48	2.29	2.59

Table 5.5: Processing times with 159 MB data for message-passing mpibzip2 on HP Opteron cluster

The speedup as a function of the number of processors is shown in Figure 5.6. When the 159 MB input file was tested, significant speedup was observed compared to the sequential version of bzip2. The best overall performance for the 159 MB data was with 100 kB and 900 kB blocks. All block sizes performed fairly equally up to 10 processors where the 100 kB and 900 kB block sizes outperformed the 1800 kB and 3600 kB blocks. The 1800 kB blocks achieved slightly better results than the 3600 kB blocks. Using the smaller block sizes allowed for better load balancing with the faster compression of each block and the shorter disk read and network transfer times. The slaves had less time waiting for the master to assign them work than with the larger and slower processing 1800 kB and 3600 kB block sizes. With the smaller block sizes there are also more blocks to process and less variance in processing time per block which contributes to improved load balancing. While the load balancing was better with the 100 kB and 900 kB blocks, the master eventually became the bottleneck with a large number of processors resulting in slaves waiting to be serviced.

Processing times for the 1875 MB test data on the HP Opteron cluster are shown in Table 5.6. The 1875 MB data was compressed in 580.42 seconds using mpibzip2 with 2 processors (one master, one slave) and 100 kB blocks, 675.34 seconds using

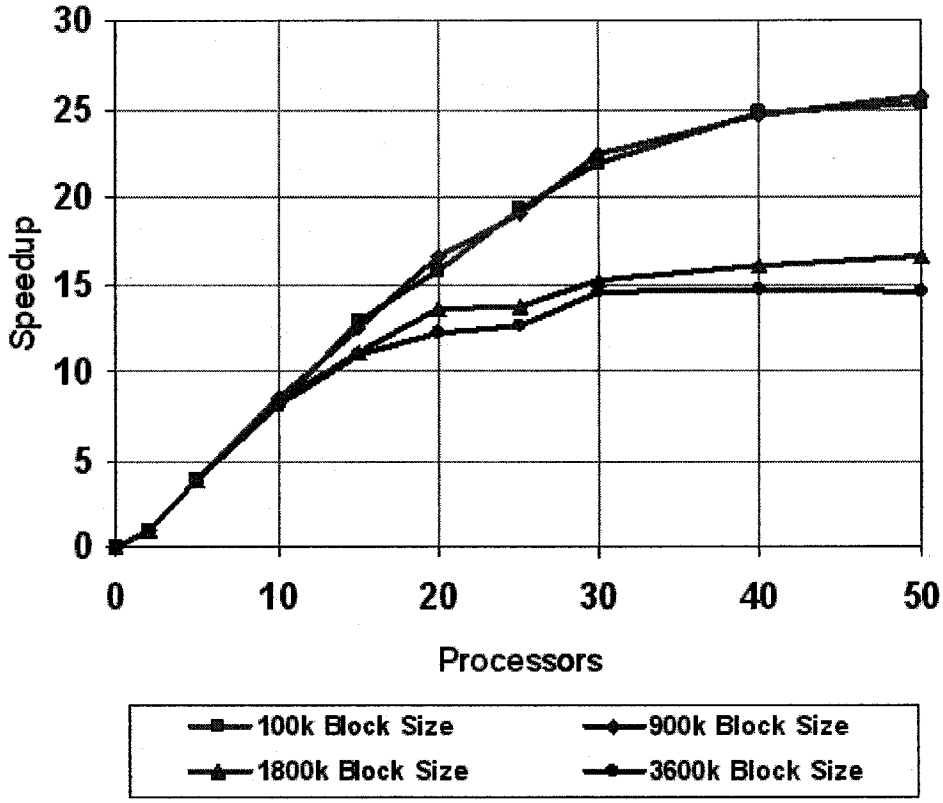

Figure 5.6: Message-passing mpibzip2 speedup for 159 MB data set on HP Opteron cluster

900 kB blocks, 671.97 seconds using 1800 kB blocks, and 669.98 seconds using 3600 kB blocks. The message-passing version of bzip2 obtained maximum performance with 100 kB blocks in 26.78 seconds using 30 processors, with 900 kB blocks in 23.80 seconds using 40 processors, with 1800 kB blocks in 23.28 seconds using 40 processors, and 3600 kB blocks in 23.60 seconds using 50 processors.

The speedup as a function of the number of processors is shown in Figure 5.7. Considerable speedup was also seen when the 1875 MB database input file was tested compared to the sequential version of bzip2. The speedup of mpibzip2 using the 900 kB, 1800 kB, and 3600 kB block sizes was comparable and superior to 100 kB blocks. Greater speedup was obtained using the larger 1875 MB data set with more blocks to process compared to the 159 MB data set. Since the data takes longer to compress compared to the 159 MB data set and compressed output is larger, the dynamics

Processors	1875 MB Data Set Time (sec)			
	100 kB	900 kB	1800 kB	3600 kB
2	580.42	675.34	671.97	669.98
5	148.65	177.28	175.63	173.66
10	66.96	81.12	80.79	80.30
15	44.59	55.09	55.25	55.92
20	33.72	40.31	39.91	39.95
25	30.30	33.20	32.83	32.54
30	26.78	27.77	26.88	27.07
40	27.44	23.80	23.28	23.77
50	27.13	24.05	24.69	23.60

Table 5.6: Processing times with 1875 MB data for message-passing mpibzip2 on HP Opteron cluster

of network usage and lower frequency the slaves contacting the master, allowed for better speedup with the larger data set.

The message-passing mpibzip2 software achieved greater speedup with the 1875 MB database than with the 159 MB kernel source. Table 5.7 shows the processing time per block for each of the block sizes with the two data sets. The longer processing time of the larger test file allowed the speedup to scale to a higher number of processors. When the average processing time per block was measured, blocks of data in the 159 MB test file were found to be compressed by the BWT algorithm approximately 1.5 times faster than with blocks from the 1875 MB database. The variance in block processing times is significantly larger with the 159 MB data set compared to same block size with the 1875 MB data set.

In order to further analyse the performance differences between test files we determined the computation to communication ratio with tests performed on the HP cluster to measure the average time it takes to send a block of data over the Myrinet network between processors, and time taken to compress a block. The data being returned from the slave processor will be compressed and likely smaller than the original block size. In a worst-case scenario, the block will not be compressible and the

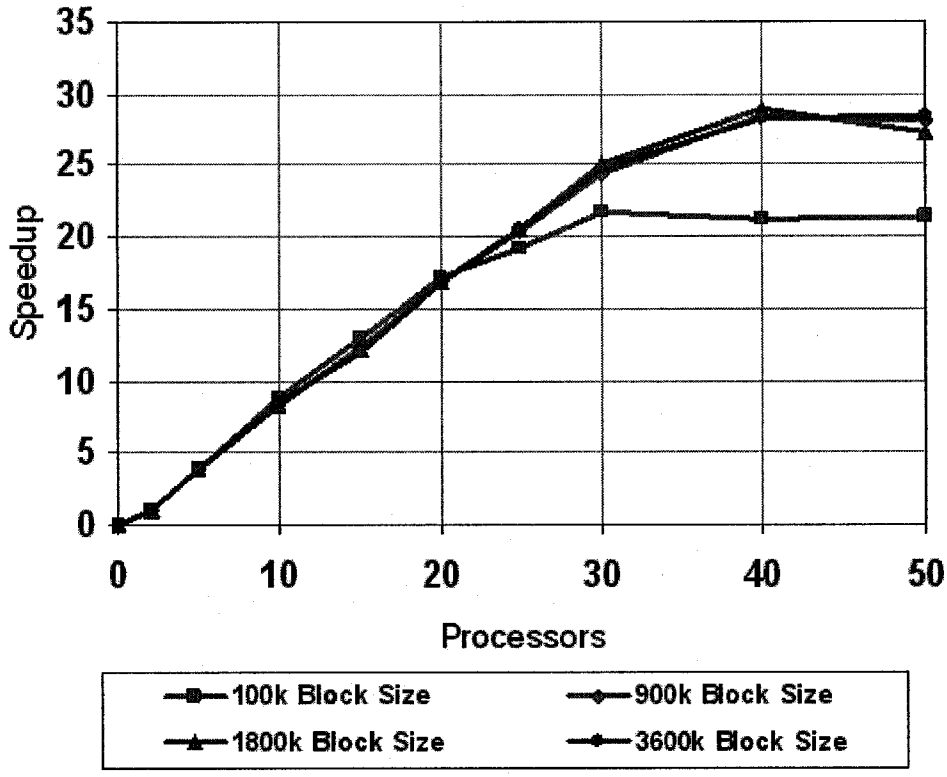

Figure 5.7: Message-passing mpibzip2 speedup for 1875 MB data set on HP Opteron cluster

same size data will need to be sent back. In our calculations we will assume the worst-case scenario where the same amount of data must be sent to the slave processor and back, which is almost the case for the 1875 MB database test file. A graph showing the average computation and communication times per block for the test files is given in Figure 5.8. With the 159 MB test file, sending and receiving blocks of data is 15.3 times faster with 100 kB blocks, 27.0 times faster with 900 kB blocks, 26.3 times faster with 1800 kB blocks, and 25.8 times faster with 3600 kB blocks than the computation required for compression. The computation to communication ratio for the 100 kB block size is smaller than the rest with the others being almost equal. The difference in ratios does not seem to effect the speedup obtained between the block sizes. With the 1875 MB test file, sending and receiving blocks of data is 25.1 times faster with 100 kB blocks, 40.7 times faster with 900 kB blocks, 41.3

Data Set	Block Size (kB)	Number Blocks	Time Per Block (ms)			
			Max	Min	Avg	Variance
159 MB	100	1667	76.95	0.12	17.87	0.03
	900	186	1057.56	15.61	205.59	10.61
	1800	93	1750.33	197.27	406.05	32.10
	3600	47	2037.69	199.14	799.28	78.38
1875 MB	100	19668	35.07	7.53	29.17	0.00
	900	2186	337.69	63.29	307.66	0.03
	1800	1093	618.95	371.72	613.58	0.06
	3600	547	1243.30	373.83	1217.10	1.33

Table 5.7: Processing times per block for message-passing pbzip2 on HP Opteron cluster

times faster with 1800 kB blocks, and 39.4 times faster with 3600 kB blocks than the computation required for compression. As with the 159 MB test file, the ratio for the 100 kB blocks is less than the other block sizes which have similar ratios. However, with the 1875 MB test file, the ratios are 1.5 times higher than with the 159 MB data set allowing it to scale better with a larger number of processors due to the longer processing time. The speedup obtained with the 1875 MB test file follows a similar pattern to the ratios with the 100 kB blocks having lower speedup than the other 3 block sizes which all perform equally.

With the message-passing version of bzip2, one processor is designated as the master to send out data to be compressed and receive the compressed data from the slave ranks. This processor does not perform any compression at all. In the case of our test files, doubling the block size also doubled the time it took to compress each block and send over the network. The master reads in the data sequentially from disk block by block and prepares the data to be sent to the next available slave rank. All slave ranks that are idle must wait their turn on a first-come, first-served basis for the master to give them more work which will cause a bottleneck as the number of processors increases. In order to see how long the slaves had to wait to receive blocks from the master as the number of processors increased, timing code was added to the slaves and measurements were obtained. Figure 5.9 shows the average slave

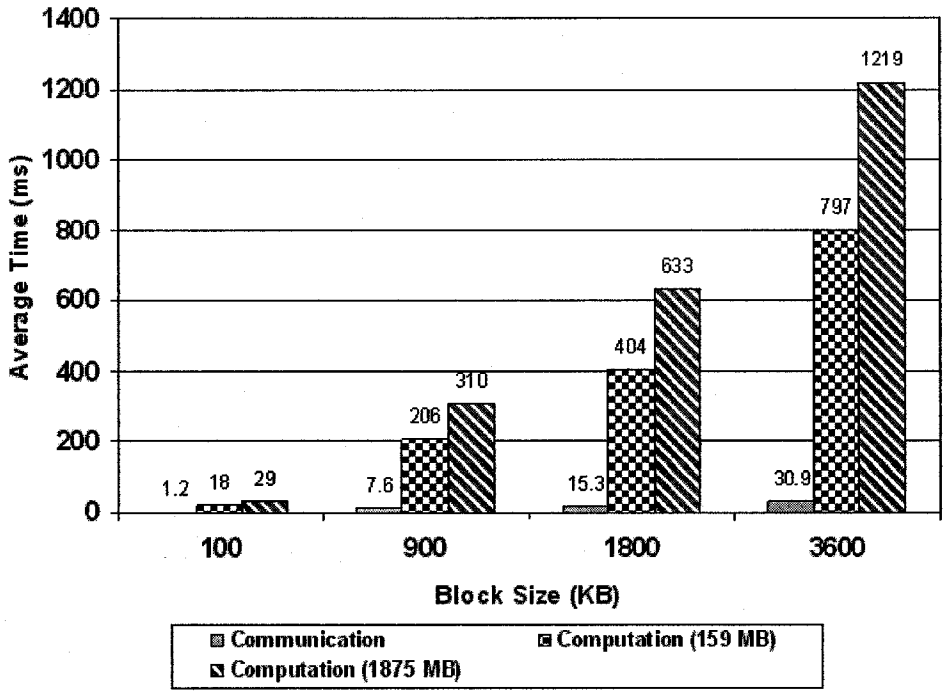

Figure 5.8: Average computation and communications time per block for mpibzip2 with various block sizes

wait time for data blocks increasing more rapidly with the 1875 MB test file as the number of processors are increased than with the 159 MB test file. The data in the 1875 MB test file is less compressible (16% smaller) than the 159 MB test file (82% smaller) which means the data received by the master from the slaves is significantly larger. This extra time to receive the larger compressed data over the network and place it into the file writing queue causes the slaves to wait longer for the master to respond with more work when processing the larger data set. For both test files, the wait times increase rapidly above 40 processors indicating that the master becomes a bottleneck when the speedup levels off. The larger block sizes show longer wait times than the smaller block sizes. Larger blocks take longer to read, write, and send over the network so slaves wait longer for the master to prepare the data and make it available for processing. Since the master reads input data serially from disk to prepare blocks for the slave tasks to process and writes compressed blocks received

from the slaves, the reading and writing from the disk would also become a bottleneck.

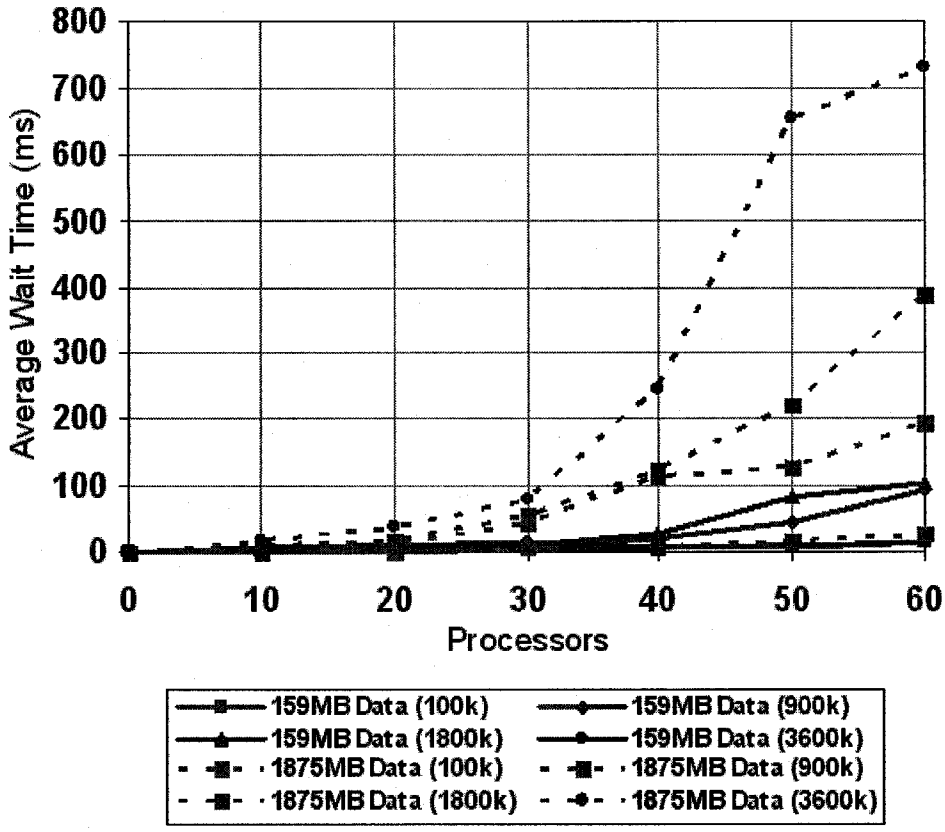

Figure 5.9: Message-passing mpibzip2 average data block wait time for slave on HP Opteron cluster

There was no direct way to measure the network load on the cluster system used. In order to determine how data transmission speeds were affected by the number of processors used, the average time it took to send a block of data from the master to an available slave was measured. These results are shown in Figure 5.10. Since each node in the cluster has 4 processors, the block send times using 2 processors is much faster than the rest since the data is only sent locally on the memory bus. The average block transfer times increase with the number of processors. There is a larger increase in transfer time as the number of processors increase with the larger block sizes than the smaller block sizes. The higher number of slaves trying to contact the master for work and sending processed results increases the network usage as the

number of processors goes up.

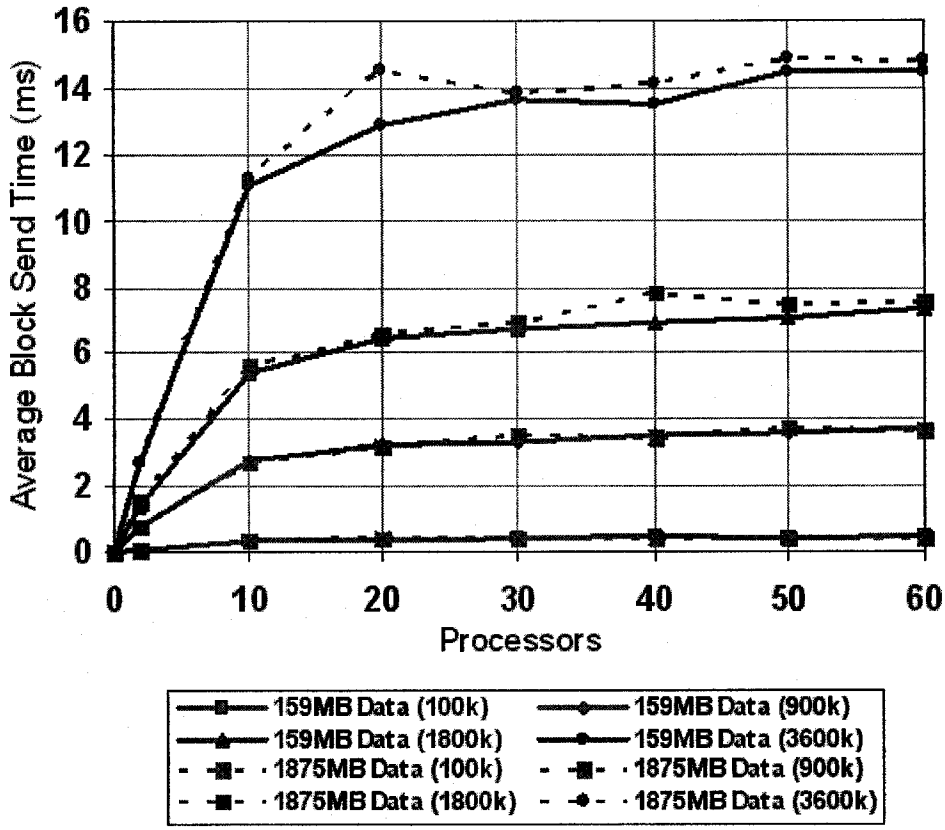

Figure 5.10: Message-passing mpibzip2 average data block send time for master on HP Opteron cluster

5.2.3 Multi-threaded vs Message-passing bzip2

The multi-threaded algorithm provides better performance than the message-passing algorithm in general. The shared-memory and cluster systems have similar performance with sequential reading of files with both systems able to read the data from disk at approximately 1 GB/sec. The shared memory system was able to write data to disk at a rate of 367 MB/sec while the cluster was slower at 89 MB/sec. The NUMA local memory bus of the SGI Altix also has much higher bandwidth (6.4 GB/sec) than the Myrinet network (0.25 GB/sec) on the cluster. The higher bandwidth and lower latency of the shared-memory bus allows the FIFO queue wait times in the multi-

threaded algorithm to be significantly less than the slave data block wait times in the message-passing algorithm. These characteristics allow the multi-threaded BWT compression algorithm to outperform the message-passing version.

The multi-threaded algorithm does have some disadvantages however. It requires more memory than the message-passing algorithm since all the required memory must be located on the one system whereas the memory with the message-passing version is spread out among each of the systems in the cluster. The memory allocation of data buffers in the multi-threaded algorithm cause threads to block while another is performing such operations. If enough memory blocking occurs while processing data, the reduced performance can lead to the multi-threaded algorithm having worse performance than the message-passing algorithm. In this case, the faster memory bus and disk write speed does not help the multi-threaded algorithm gain an advantage. Each slave in the message-passing algorithm is a separate process that does not share memory and thus not blocked by the allocation of memory in other slaves. Shared memory systems are much more expensive than clusters so it may be more cost effective to use the message-passing algorithm to compress data.

Both the multi-threaded and message-passing parallel bzip2 algorithms provide considerable speedup over the sequential version. As the output from the parallel algorithms is compatible with the popular sequential bzip2 software, everyone with access to a shared or distributed memory system with multiple processors can take advantage of faster compression. People who only have access to the sequential version do not require any special hardware or software to decompress the data for themselves.

5.3 Experimental Results for Task Parallel Algorithms

The BWT compression algorithm can be separated into distinct tasks, therefore a

task parallel strategy is also possible to achieve speedup. The software pipeline design of the task parallel BWT algorithm uses several different types of parallelism including algorithmic, geometric, and temporal multiplexing. We performed experiments using task parallelism to determine how the performance compares to and if it has any advantages to the data parallel design. With the overlapping of tasks in the pipeline, we expect the latency to be lower so the data can be compressed in less time. Since bzip2 could not be decomposed into separate tasks, bwtzip was chosen to evaluate task parallelism.

The message-passing mpibwtzip program was compiled using PathScale 2.2.1 on the HP Opteron 275 cluster with the bwtzip source code base [25]. The same two data sets were used for testing. The bwtzip software can compress the 159 MB test file using 900 kB blocks down to 29 MB, only 18% of its original size. The 1875 MB test file can be compressed using 900 kB blocks down to 1612 MB, 86% of its original size.

5.3.1 Profiling

Since the message-passing bwtzip algorithm has no way of knowing before-hand what type of data it will be processing, a generic set of parameters for the software pipeline tasks must be chosen that will give decent performance with different types of input. In order to choose these parameters, 10 different sets of data were profiled to determine what percentage of each task was required to process the data. High precision timing routines were added to the bwtzip software to calculate the percentage of time spent in each of the tasks. Recall that the bwtzip software was separated into three tasks, the first consisting of the suffix tree sorting routine, the second task contained both the move to front (MTF) and zero length encoding (ZLE) routines, and the third task was the arithmetic encoding.

The 10 data sets that were profiled include the Canterbury Corpus which contains a mixture of text, executables, and binary data [2]. The Calgary Corpus con-

tains mostly text and some binary and image data [41]. The WAV set contains uncompressed audio data [16]. The TIF set contains uncompressed images from the Waterloo BragZone Colour Set [22]. The Worms2 set contains binary, audio, video, and image data from the Worms2 video game [1]. The Maximum Compression set contains a mixture of image, text, binary, and executable data [4]. The Large Text 8 and 9 sets contain text data from the English version of Wikipedia [27]. The Linux Kernel Source set contains source code for the Linux 2.4.23 kernel [37]. Finally, the Elliptic Curve Database set contains binary elliptic curve distinguished points [3]. Results from profiling the data sets are found in Table 5.8.

Data Set	**Suffix Tree (%)**	**MTF +ZLE (%)**	**Arithmetic Coding (%)**	**Entropy (bits/Byte)**
Calgary Corpus *(2.7 MB)*	86.85	5.77	7.38	4.71
Canterbury Corpus *(3.1 MB)*	88.03	5.88	6.08	5.35
Elliptic Curve DB *(1875 MB)*	77.82	11.77	10.40	7.54
Large Text 8 *(95 MB)*	86.99	5.53	7.48	7.07
Large Text 9 *(954 MB)*	88.00	5.16	6.85	7.02
Linux Kernel Source *(159 MB)*	89.84	4.63	5.53	6.95
Maximum Compress *(51 MB)*	87.32	6.24	6.43	5.08
TIF *(12 MB)*	86.10	6.17	7.73	5.55
WAV *(8.3 MB)*	76.99	11.60	11.42	5.16
Worms2 *(16 MB)*	81.54	8.88	9.59	7.07
Average:	**84.95**	**7.16**	**7.89**	**6.151**
Variance:	**20.37**	**6.94**	**3.79**	**1.138**

Table 5.8: Profiling results with bwtzip on multiple data sets

From the profiling results, the suffix tree task takes the majority of the time, ranging from 77% to 90% of the processing time. The MTF+ZLE task ranges between 5% and 12%, while the Arithmetic coding task ranges from 5% to 11%. Taking the average profile results from each task for the 10 data sets, we get an approximate split of 84% for the first task, 8% for the second task, and 8% for the third task. This work load split will be used as the parameters for the 3-stage pipeline message-passing bwtzip algorithm. Since a master/slave model approach is used, one processor

will be reserved for the master to manage the data and the task parallel software pipeline will be load balanced with 8% of the processors assigned to perform the MTF+ZLE task, 8% available for the arithmetic coding task, and the remaining processors designated for suffix tree task. Each stage in the pipeline is a master/slave configuration providing data to the subsequent stage after it has been processed. For example, if 50 processors are available for the message-passing bwtzip algorithm, 1 will be reserved for the master, 4 will be assigned to the MTF+ZLE task, 4 will be assigned to the arithmetic coding task, and the remaining 41 will be used for the suffix tree task. With the 2-stage pipeline design, the MTF+ZLE task and the arithmetic coding task are combined to form the second stage in the pipeline. Now one processor is reserved for the master, 16% of the processors are assigned to the second stage, and the remaining processors are assigned suffix tree stage.

Entropy is a measure of the uncertainty associated with an outcome such that $H = -\sum_{i=1}^{n} p_i \log_2 p_i$ bits per symbol where n is the number of symbols and p_i is the probability of the i^{th} symbol [34]. The entropy of each data set was measured using the *ent* [39] pseudorandom number sequence test software. If data has high entropy (close to 8 bits per byte) then it will not compress very well as it is essentially random. In theory, lower entropy leads to better compression. Table 5.8 shows that the data sets with highest entropy such as the WAV and Elliptic Curve DB require almost twice as much processing time in the MTF+ZLE and arithmetic coding tasks as those with the lowest entropy. The longer processing time for higher entropy data sets may lead to better scaling than with lower entropy data.

There was not enough computing resources available to fully test all 10 data sets with all block sizes so two data sets at opposite ends of the profiling results were chosen. The 159 MB Linux kernel source data set with the highest usage of the suffix tree task and the 1875 MB elliptic curve database set with the lowest usage of the suffix tree task were chosen to represent the benchmark results of the parallel bwtzip

algorithm.

5.3.2 HP Opteron 275 Cluster

Experiments using both 3-stage pipeline and 2-stage pipeline task parallel BWT algorithms were performed using the HP Opteron cluster.

Three Stage Pipeline

Recall that the 3-stage pipeline is load balanced with one processor reserved for the master, 8% of the processors assigned to the third stage (*Arithmetic Coding*), 8% assigned to the second stage (*MTF+ZLE*), and the remaining processors assigned to the first stage (*Sorting*). Table 5.9 shows the actual number of processors assigned to each stage of the pipeline for the experiments carried out using the 3-stage pipeline algorithm.

Total	**Assigned Processors**			
Processors	Master	Stage 1	Stage 2	Stage 3
5	1	2	1	1
10	1	7	1	1
15	1	12	1	1
20	1	17	1	1
25	1	20	2	2
30	1	25	2	2
40	1	33	3	3
50	1	41	4	4
60	1	51	4	4
70	1	59	5	5
80	1	67	6	6
90	1	75	7	7
100	1	83	8	8

Table 5.9: 3-stage pipeline load balancing processor assignment

Processing times for the 159 MB test data using the 3-stage pipeline design on the HP Opteron cluster are shown in Table 5.10. The 159 MB data was compressed using

the sequential bwtzip with 1 processor and 100 kB blocks in 169.49 seconds, 900 kB blocks in 209.73 seconds, 1800 kB blocks in 225.41 seconds, and 3600 kB blocks in 233.90 seconds. The fastest processing time was achieved with mpibwtzip taking only 7.11 seconds using 50 processors with 100 kB blocks, 8.53 seconds using 60 processors with 900 kB blocks, 9.80 seconds using 60 processors with 1800 kB blocks, and 13.57 seconds using 60 processors with 3600 kB blocks. Using more than 60 processors, the performance leveled off and no further improvements were seen.

Processors	**159 MB Data Set Time (sec)**			
	100 kB	900 kB	1800 kB	3600 kB
1	169.49	209.73	225.41	233.90
5	89.24	108.11	116.38	121.96
10	26.84	33.35	35.95	41.41
15	19.10	24.89	26.08	30.03
20	16.12	16.40	17.97	22.09
25	11.91	14.67	16.68	19.75
30	9.01	12.66	14.42	18.35
40	8.01	9.42	10.89	14.79
50	7.11	8.86	10.43	13.65
60	7.57	8.53	9.80	13.57

Table 5.10: Processing times for 3-stage message-passing mpibwtzip with 159 MB data on HP Opteron cluster

The speedup as a function of the number of processors is shown in Figure 5.11. When the 159 MB Linux kernel input file was tested, significant speedup was observed compared to the sequential version of bwtzip. The 100 kB, 900 kB, and 1800 kB block sizes obtained similar speedup while 3600 kB blocks had lower performance. All block sizes performed equally well up to 20 processors. Speedup peaks above 50 processors with 3600 kB blocks as the data is only split into 47 blocks.

Processing times for the 1875 MB test data on the HP Opteron cluster are shown in Table 5.11. The 1875 MB data was compressed in 3573 seconds using bwtzip with 1 processor and 100 kB blocks, 5086 seconds using 900 kB blocks, 5098 seconds

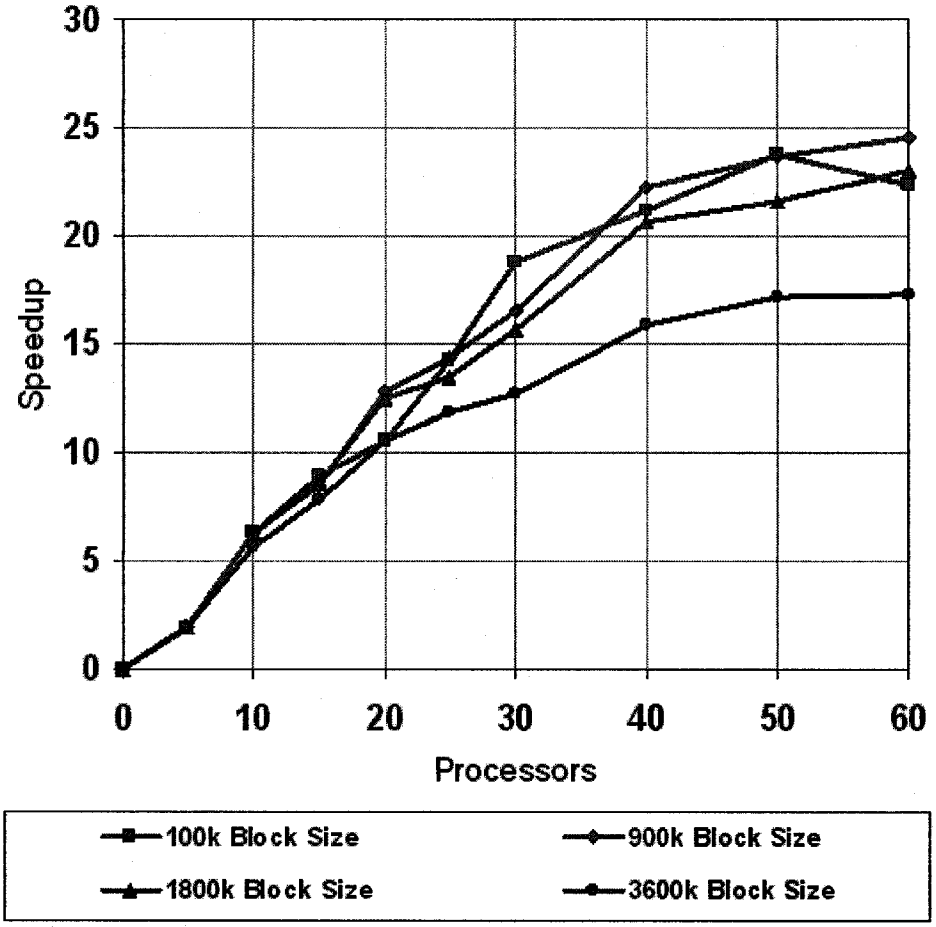

Figure 5.11: Message-passing 3-stage mpibwtzip speedup for 159 MB data set on HP Opteron cluster

using 1800 kB blocks, and 5418 seconds using 3600 kB blocks. The message-passing version of bzip2 obtained maximum performance with 100 kB blocks in 127.59 seconds using 100 processors, with 900 kB blocks in 125.20 seconds using 100 processors, with 1800 kB blocks in 136.91 seconds using 100 processors, and 3600 kB blocks in 149.56 seconds using 100 processors.

The speedup as a function of the number of processors is shown in Figure 5.12. Considerable speedup was also seen when the 1875 MB input file was tested compared to the sequential version of bwtzip. The best overall speedup for the 1875 MB data was using 900 kB blocks, with 1800 kB and 3600 kB blocks obtaining similar but slightly lower results. The performance of the 100 kB blocks was lower than the

Processors	1875 MB Data Set Time (sec)			
	100 kB	900 kB	1800 kB	3600 kB
1	3572.83	5086.00	5098.00	5417.50
5	1598.75	2275.11	2403.72	2391.82
10	880.15	805.66	853.08	922.40
15	816.15	817.84	816.56	815.15
20	823.97	828.04	822.89	813.02
25	435.29	438.75	450.30	455.23
30	437.09	442.38	448.31	454.03
40	307.07	317.79	318.39	334.29
50	209.70	203.15	213.35	223.65
60	208.69	204.27	210.43	223.82
70	170.46	164.70	174.74	184.70
80	151.11	146.52	157.98	165.18
90	132.47	136.15	142.69	155.84
100	127.59	125.20	136.91	149.56

Table 5.11: Processing times for 3-stage message-passing mpibwtzip with 1875 MB data on HP Opteron cluster

rest. With the faster processing 100 kB blocks and shorter network transfer times, the master became a bottleneck sooner than with the larger block sizes. The slave tasks returned to the master for work more quickly than the master could keep up. Speedup continued to increase up to 100 processors which indicates that if more processors were available on the cluster, even higher speedup might be obtained. The speedup levels off at certain points on the graph, between 10 and 20 processors, 25 and 30 processors, and 50 and 60 processors. This effect is due to the pipeline load balancing where the same number of processors is being assigned to the MTF+ZLE and arithmetic coding stages of the pipeline. Recall that 8% of the processors are assigned to these stages but with 10, 15, and 20 processors, only 1 processor is assigned to each of the MTF+ZLE and arithmetic coding stages. While the sorting stage is assigned 17 processors when using 20 CPUs compared with 7 processors when using 10 CPUs, the speedup is limited by the other stages. The data cannot be processed any faster since there are no extra MTF+ZLE and arithmetic coding stages to handle

the increased load which results in similar speedup.

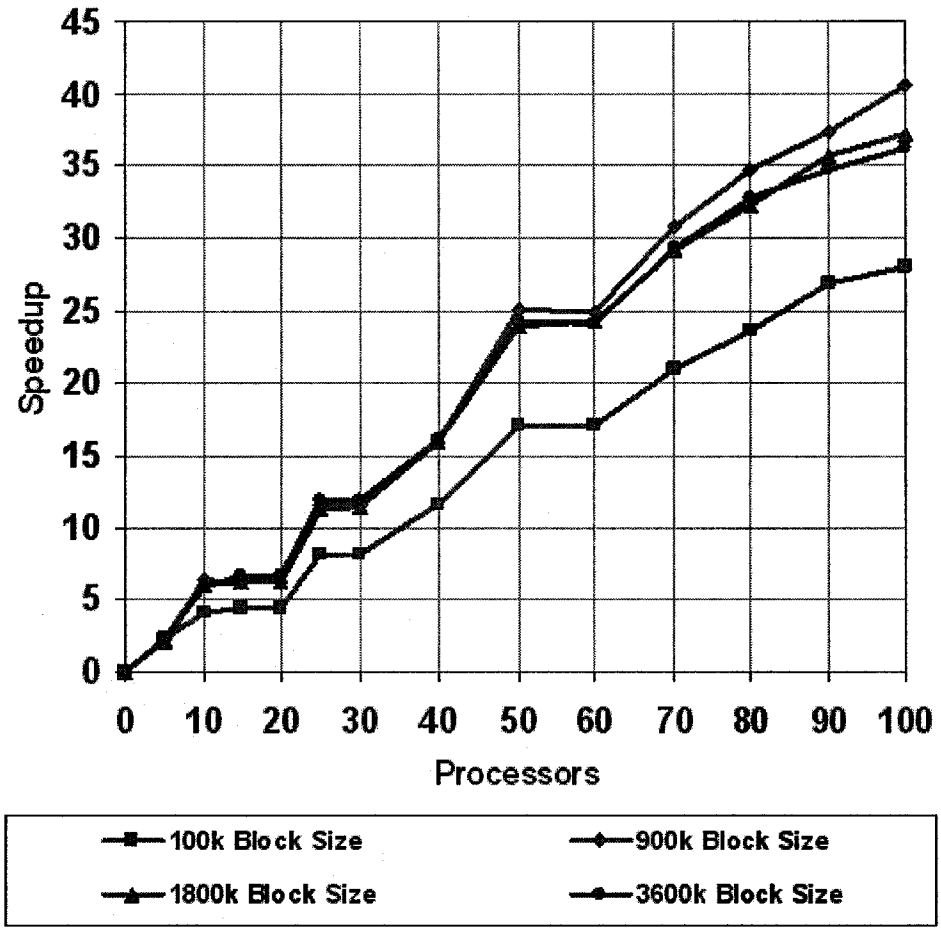

Figure 5.12: Message-passing 3-stage mpibwtzip speedup for 1875 MB data set on HP Opteron cluster

There was no direct way to measure the network load on the cluster system used. In order to determine how data transmission speeds were affected by the number of processors used, the average time it took to send a block of data from the master to an available slave in stage 1 of the pipeline was measured. These results are shown in Figure 5.13. The average block transfer times increase with the number of processors. There is a larger increase in transfer time as the number of processors increase with the larger block sizes than the smaller block sizes. The higher number of slaves trying to contact the master for work and sending processed results increases the network

usage as the number of processors goes up.

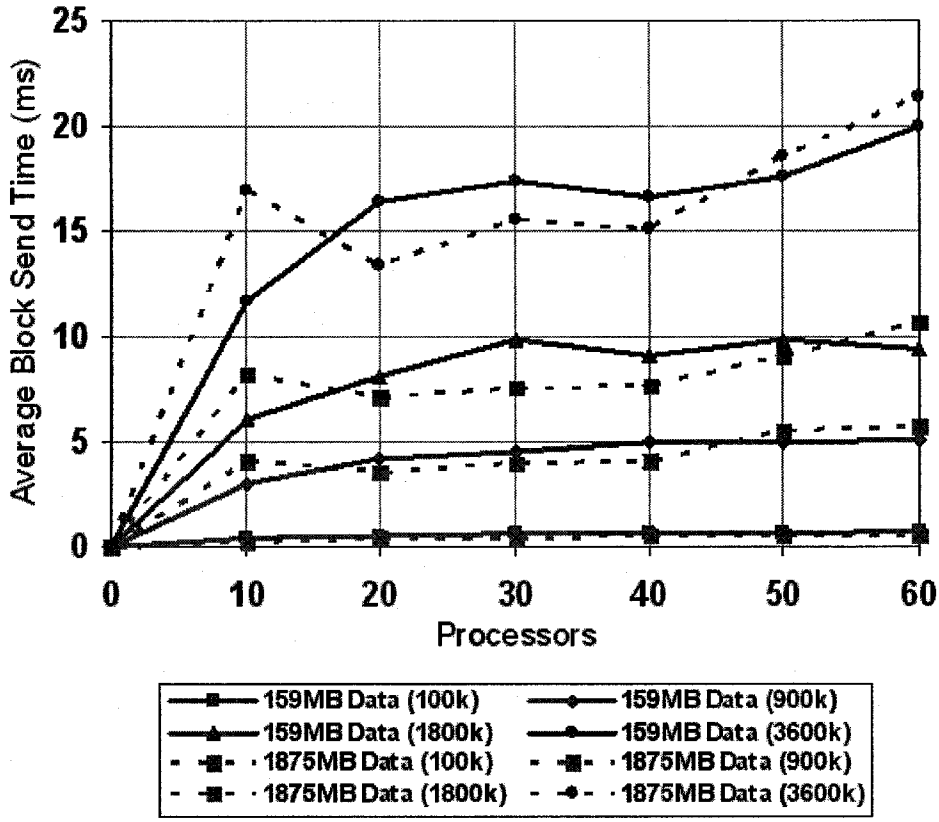

Figure 5.13: Message-passing 3-stage mpibwtzip average data block send time for master on HP Opteron cluster

Two Stage Pipeline

Recall that the 2-stage pipeline is load balanced with one processor reserved for the master, 16% of the processors assigned to the second stage (*MTF+ZLE+Arithmetic Coding*), and the remaining processors assigned to the first stage (*Sorting*). Table 5.12 shows the actual number of processors assigned to each stage of the pipeline for the experiments carried out using the 2-stage pipeline algorithm.

Processing times for the 159 MB test data using the 2-stage pipeline design on the HP Opteron cluster are shown in Table 5.13. The 159 MB data was compressed using the sequential bwtzip with 1 processor and 100 kB blocks in 169.49 seconds, 900

Total Processors	Assigned Processors		
	Master	Stage 1	Stage 2
5	1	2	2
10	1	7	2
15	1	12	2
20	1	17	2
25	1	20	4
30	1	25	4
40	1	33	6
50	1	41	8
60	1	51	8
70	1	59	10
80	1	67	12
90	1	75	14
100	1	83	16

Table 5.12: 2-stage pipeline load balancing processor assignment

kB blocks in 209.73 seconds, 1800 kB blocks in 225.41 seconds, and 3600 kB blocks in 233.90 seconds. The fastest processing time was achieved with mpibwtzip taking only 7.60 seconds using 60 processors with 100 kB blocks although using 40 processors was only 0.5 seconds slower. With 900 kB blocks performance peaked at 8.44 seconds using 60 processors, 10.01 seconds using 60 processors with 1800 kB blocks, and 14.00 seconds using 60 processors with 3600 kB blocks. Using more than 60 processors, the performance leveled off and no further improvements were seen.

The speedup as a function of the number of processors is shown in Figure 5.14. When the 159 MB Linux kernel input file was tested, significant speedup was observed compared to the sequential version of bwtzip. The 100 kB, 900 kB, and 1800 kB blocks obtained similar speedup while the 3600 kB block speedup was lower. As with the 3-stage pipeline, speedup starts to level off above 50 processors with 3600 kB blocks as the data is only split into 47 blocks. All block sizes performed fairly equally up to 20 processors.

Processing times for the 1875 MB test data on the HP Opteron cluster are shown in Table 5.14. The 1875 MB data was compressed in 3573 seconds using bwtzip

Processors	159 MB Data Set Time (sec)			
	100 kB	900 kB	1800 kB	3600 kB
1	169.49	209.73	225.41	233.90
5	63.46	72.41	75.59	81.75
10	27.55	33.17	35.23	40.88
15	18.45	23.05	26.46	32.25
20	14.05	17.36	20.97	25.36
25	11.17	14.53	16.56	20.03
30	10.35	12.28	14.46	17.97
40	8.12	10.21	12.34	17.42
50	7.98	8.87	10.87	14.78
60	7.60	8.44	10.01	14.00

Table 5.13: Processing times for 2-stage message-passing mpibwtzip with 159 MB data on HP Opteron cluster

with 1 processor and 100 kB blocks, 5086 seconds using 900 kB blocks, 5098 seconds using 1800 kB blocks, and 5418 seconds using 3600 kB blocks. The message-passing version of bzip2 obtained fastest processing times with 100 kB blocks in 94.51 seconds using 100 processors, with 900 kB blocks in 104.89 seconds using 100 processors, with 1800 kB blocks in 111.29 seconds using 100 processors, and 3600 kB blocks in 116.05 seconds using 100 processors.

Figure 5.15 shows the speedup as a function of the number of processors. Considerable speedup was also seen when the 1875 MB database input file was tested compared to the sequential version of bwtzip. The best overall speedup for the 1875 MB data was with 900 kB, 1800 kB, and 3600 kB blocks. The performance of the 100 kB blocks was lower than the rest. With the faster processing 100 kB blocks and shorter network transfer times, the master became a bottleneck sooner than with the larger block sizes. The slave tasks returned to the master for work more quickly than the master could keep up. The maximum speedup for all blocks sizes was 100 processors which indicate that even higher speedups could be achieved if the cluster had more processors available. With the 2-stage pipeline, the effects due to the load

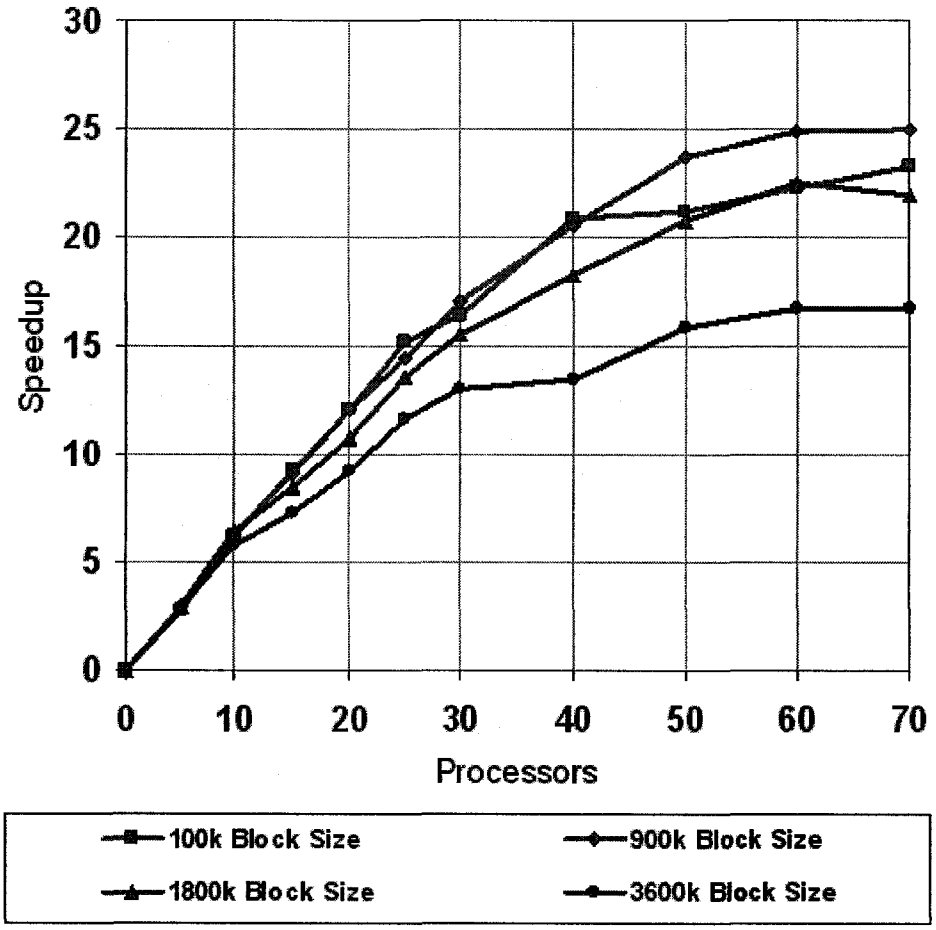

Figure 5.14: Message-passing 2-stage mpibwtzip speedup for 159 MB data set on HP Opteron cluster

balancing can also been seen but to a lesser extent than the 3-stage pipeline. In this case 16% of the processors are assigned to the combined MTF+ZLE+arithmetic coding stage. While 5, 10, 15, and 20 processors all have 2 processors assigned to the second stage, the increase in processors assigned to the first stage have an impact on speedup. With two stage 2 tasks available, a larger number of stage 1 processors can be assigned before the pipeline becomes saturated unlike the 3-stage pipeline which only had one stage 2 and 3 tasks available.

There was no direct way to measure the network load on the cluster system used. In order to determine how data transmission speeds were affected by the number of

Processors	1875 MB Data Set Time (sec)			
	100 kB	900 kB	1800 kB	3600 kB
1	3572.83	5086.00	5098.00	5417.50
5	1250.01	1423.08	1592.64	1665.73
10	1301.68	1326.76	1338.89	1314.33
15	684.59	626.18	652.71	635.04
20	466.38	423.22	471.30	477.64
25	391.03	338.94	356.35	376.03
30	313.68	312.21	324.04	329.09
40	219.40	226.55	238.16	249.64
50	171.20	181.26	191.03	201.27
60	157.88	158.77	167.78	178.55
70	130.01	143.34	147.67	154.96
80	124.53	131.27	140.74	145.84
90	115.92	120.69	113.99	136.79
100	94.51	104.89	111.29	116.05

Table 5.14: Processing times for 2-stage message-passing mpibwtzip with 1875 MB data on HP Opteron cluster

processors used, the average time it took to send a block of data from the master to an available slave in stage 1 of the pipeline was measured. These results are shown in Figure 5.16. The average block transfer times increase with the number of processors. There is a larger increase in transfer time as the number of processors increase with the larger block sizes than the smaller block sizes. The higher number of slaves trying to contact the master for work and sending processed results increases the network usage as the number of processors goes up.

The message-passing mpibwtzip software achieved greater speedup with the 1875 MB database than with the 159 MB kernel source. Table 5.15 shows the processing time per block for each of the block sizes with the two data sets. The longer processing time of the larger test file allowed the speedup to scale to a higher number of processors. When the average processing time per block was measured, blocks of data in the 159 MB test file were found to be compressed by the BWT algorithm approximately 2 times faster than with blocks from the 1875 MB database. The variance in

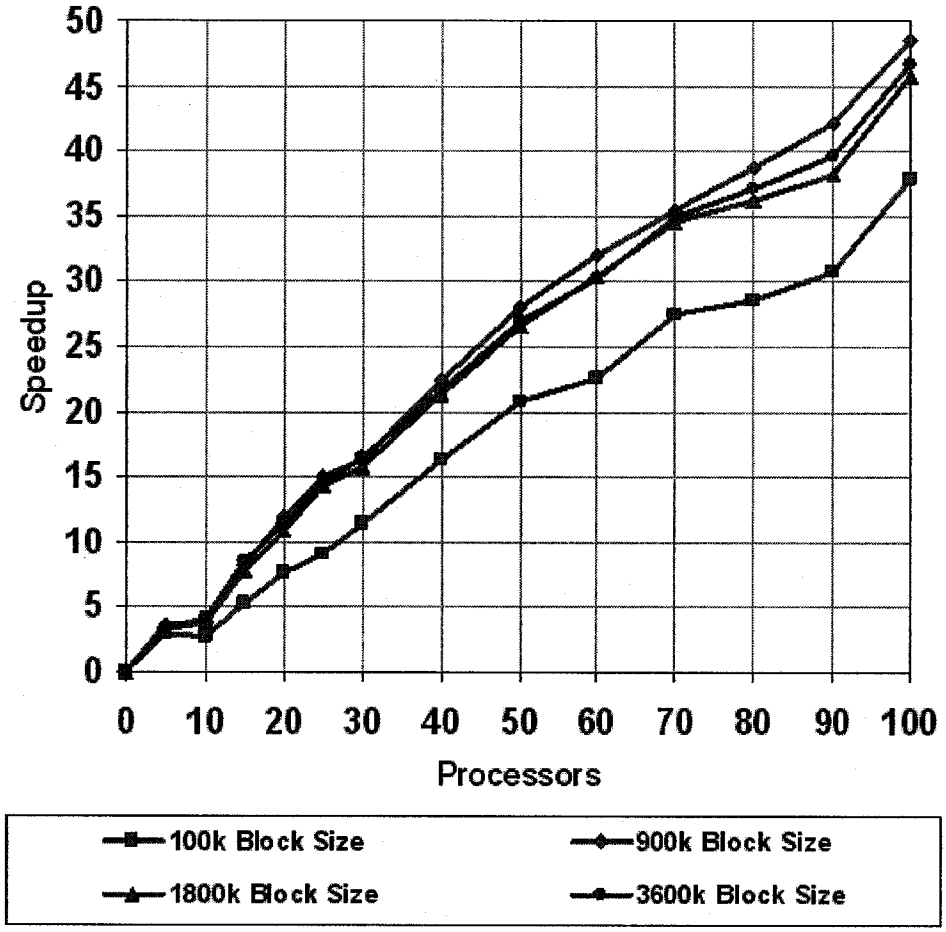

Figure 5.15: Message-passing 2-stage mpibwtzip speedup for 1875 MB data set on HP Opteron cluster

block processing times is significantly larger with the 159 MB data set compared to same block size with the 1875 MB data set. The overall speedup with 3600 kB blocks from the 159 MB data set is lower than the other block sizes using both 3-stage and 2-stage pipelines. The variance for the 3600 kB block processing time is 14 times higher than 1800 kB blocks and 36 times higher than 900 kB blocks. With a smaller number of blocks and much higher variance, the load balancing becomes less efficient in the pipeline than the other block sizes. Faster processing blocks stall while waiting for the stage 2 and stage 3 processors to become free from the slower blocks. This increased variability is enough to reduce the performance of the software pipeline to

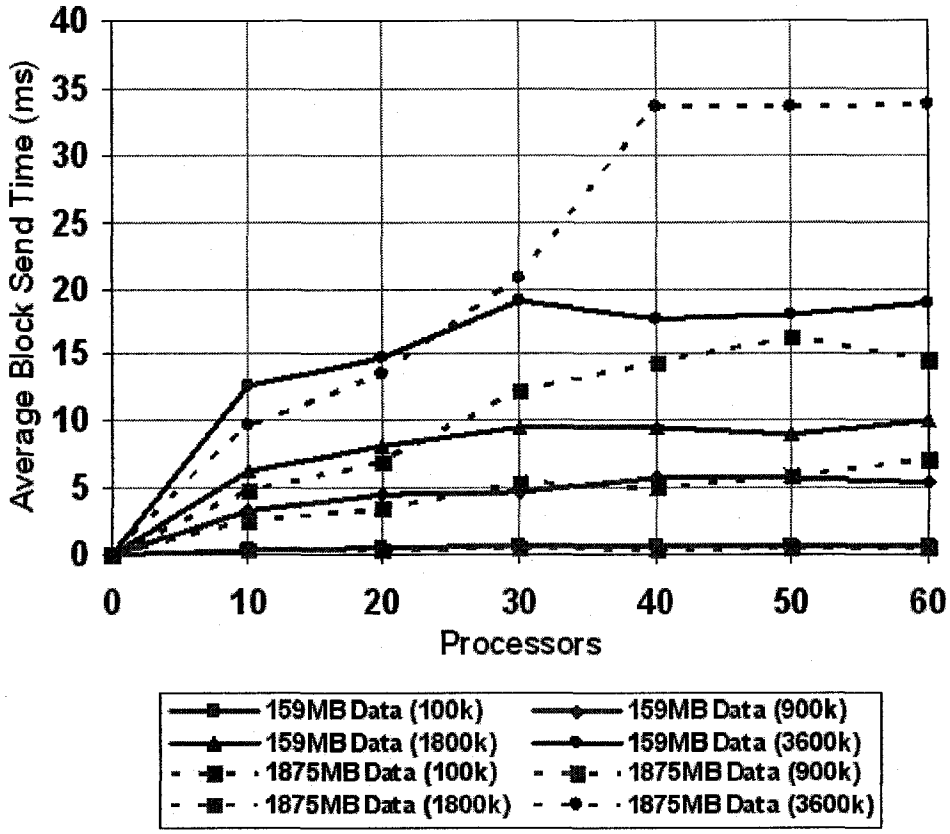

Figure 5.16: Message-passing 2-stage mpibwtzip average data block send time for master on HP Opteron cluster

the point where it performs worse with 3600 kB blocks than the rest.

In order to further analyse the performance differences between test files we determined the computation to communication ratio with tests performed on the HP cluster to measure the average time it takes to send a block of data over the Myrinet network between processors, and time taken to compress a block. With the 3-stage message-passing bwtzip algorithm, the block of data is sent from the master to a slave that will perform the suffix tree task. After the data is processed it will then be sent to the designated slave that will perform the MTF+ZLE task and once completed will be forwarded over the network again to the designated final slave that will perform the arithmetic coding task. Once the block has been processed by all tasks, the arithmetic coding task slave will send the results back to the master to be written

Data Set	Block Size (kB)	Number Blocks	Time Per Block (ms)			
			Max	Min	Avg	Variance
159 MB	100	1667	133.98	6.31	119.96	0.06
	900	186	1339.47	120.24	1228.55	10.10
	1800	93	2706.22	1415.02	2534.61	25.64
	3600	47	5821.50	1421.59	5299.55	362.66
1875 MB	100	19668	217.83	23.75	207.92	0.01
	900	2186	2468.44	540.98	2442.50	1.68
	1800	1093	4981.83	3011.66	4918.25	3.48
	3600	547	10346.00	2992.78	10110.01	95.72

Table 5.15: Processing times per block for message-passing mpibwtzip on HP Opteron cluster

to disk. This means a block of data will be sent over the network four times when it returns to the master again. For the 2-stage algorithm, the data is sent over the network three times.

The data being sent between tasks and returned to the master will likely be smaller than the original block size after being processed by the MTF+ZLE and arithmetic coding tasks. In a worst-case scenario, the block will not be compressible and the same size data will need to be sent back. In our calculations we will assume the worst-case scenario where the same amount of data must be sent between all task processors and back to the master, which is almost the case for the binary database test file. A graph showing the average computation and communications time per block of the 3-stage pipeline for the test files is shown in Figure 5.17. With the 159 MB test file, sending the blocks of data between the master and slaves is 49.0 times faster with 100 kB blocks, 78.1 times faster with 900 kB blocks, 83.6 times faster with 1800 kB blocks, and 80.9 times faster with 3600 kB blocks than the computation required for compression. The ratio for the 100 kB block size is smaller than the rest with the others being almost equal. With the 1875 MB test file, sending and receiving the blocks of data is 91.3 times faster with 100 kB blocks, 169.7 times faster with 900 kB blocks, 179.2 times faster with 1800 kB blocks, and 168.6 times faster with 3600 kB

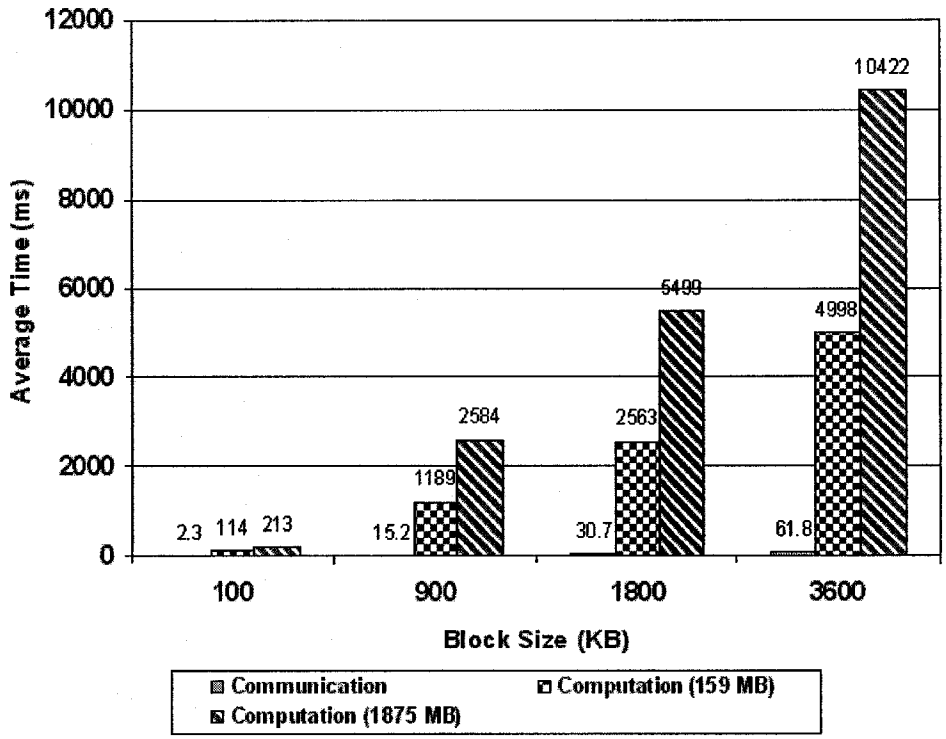

Figure 5.17: Average computation and communications time per block for 3-stage mpib-wtzip with various block sizes

blocks than the computation required for compression. As with the 159 MB test file, the ratio for the 100 kB blocks is less than the other block sizes which have similar ratios. Since the 1875 MB test file has a computation to communication ratio that is twice as high per block as the 159 MB test file, the longer processing time from the higher amount of entropy allows it to scale better with a larger number of processors. The speedup obtained with the 1875 MB data follows a similar pattern to the ratios with the 100 kB blocks having lower speedup than the other 3 block sizes which all perform equally.

With the message-passing version of bwtzip, one processor is designated as the master to send out data to be processed by the suffix tree task and receive the final compressed data from the arithmetic coding task. This processor does not perform any compression at all. In the case of our test files, doubling the block size also doubled the time it took to compress each block and send over the network. The

master reads in the data sequentially from disk block by block and prepares the data to be sent to the next available suffix tree slave rank. All slave ranks that are idle must wait their turn on a first-come, first-served basis for the master to give them more work which will cause a bottleneck as the number of processors increases. The suffix tree tasks then send completed work to the MTF+ZLE task which after processing sends the data to the Arithmetic coding task for the final compression. In order to see how the long the suffix-tree slaves had to wait to receive blocks from the master as the number of processors increased, timing code was added to the slaves and measurements were obtained. Figure 5.18 shows the average slave wait time for data blocks with the 3-stage pipeline increasing more rapidly with the 159 MB test file as the number of processors are increased than with the 1875 MB test file. Using 60 processors, the average wait time for slaves to receive data from the master is approximately 6 times longer with the 159 MB data set compared to the 1875 MB data set. Above 60 processors, the speedup levels off with the 159 MB test file due to the master becoming a bottleneck. Similar results in slave wait times are seen with the 2-stage pipeline algorithm. Since the master reads input data serially from disk to prepare blocks for the slave tasks to process and writes compressed blocks received from the slaves, the reading and writing from the disk would also become a bottleneck.

The message-passing parallel bwtzip algorithm provides considerable speedup over the sequential version. The performance of the 2 and 3-stage pipeline algorithms with the 159 MB test file are almost identical. The data in the 2-stage pipeline is sent one less time on the network than with the 3-stage pipeline. This reduced network usage did not compensate for the decreased overlapping of tasks with one less stage in the pipeline causing the performance to be similar. Using the 1875 MB test file, the 2-stage pipeline provided more speedup than the 3-stage design when looking at both the speedup with an equal number of processors and maximum speedup. Since

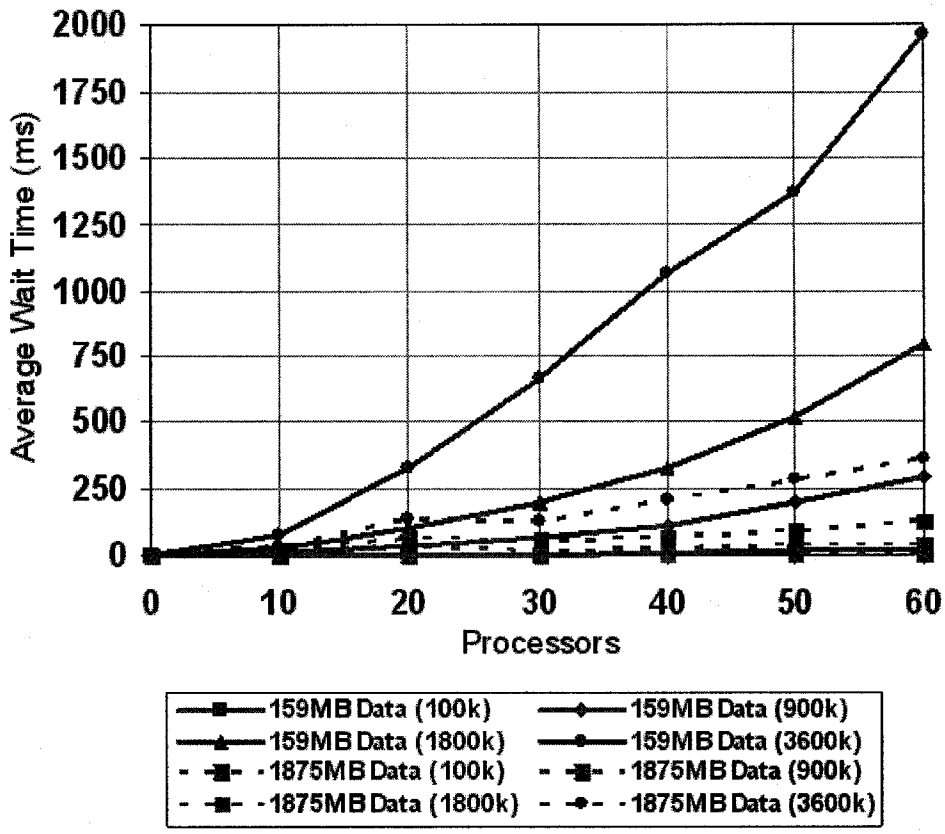

Figure 5.18: Message-passing 3-stage mpibwtzip average data block wait time for slave on HP Opteron cluster

the data is less compressible with the larger test file and thus more data is being sent over the network, reducing the number of network hops was a greater advantage than having a longer pipeline. The 2-stage pipeline provides better performance and is simpler to implement.

Since the output from the parallel algorithm is compatible with the sequential bwtzip software, everyone with access to a shared or distributed memory system with multiple processors can take advantage of faster compression. People who only have access to the sequential version do not require any special hardware or software to decompress the data.

5.3.3 SGI Altix 3700

Initially, a multi-threaded version of the parallel bwtzip algorithm was implemented and tested on an SGI Altix 3700 Bx2 shared memory system with 128 Itanium2 processors running at 1.6 GHz, 6 MB L3 cache, and 256 GB system RAM. This implementation used the same 84%/8%/8% split between tasks as the message-passing version. Enough threads of each task were created to process the data, one thread to read the input data and populate a FIFO queue, and one thread to write the compressed processed data to disk. Unfortunately it was quickly discovered that using multiple processors did not increase speedup but actually decreased it. With the 159 MB data set, using 2 processors was twice as slow and 4 processors was three times slower than the sequential version of bwtzip. With the 1875 MB data set, using 2 and 4 processors was slightly slower than the sequential version of bwtzip.

The implementation was first checked to ensure it was correct. Each thread and task was confirmed to be operating properly. Further investigation found that the suffix tree task was to blame. The suffix tree sorting routine is written in object oriented C++ with many objects being created and destroyed during the sort for various things such as the nodes of the suffix tree. This causes a large amount of memory to be allocated and released. This technique works fine for the sequential version, but when multiple-threads in a process attempt to perform a suffix tree sort simultaneously, each thread spends a large amount of time blocking on memory requests. Memory allocation within a process is an atomic operation so only one thread at a time can allocate new memory. The end result is a net loss of speed from the overhead of managing the shared memory amongst the threads. In order to overcome this problem, the suffix tree code would have to be entirely re-written to better manage memory. This is a good reminder for software developers who may want to parallelize their code in the future to be careful with memory management and minimize the amount of memory that is allocated and released dynamically.

5.4 Data Parallel vs Task Parallel BWT

We examined the experimental results from both the data parallel and task parallel Burrows-Wheeler Transform compression algorithms to determine which parallelization technique was superior. While the bzip2 and bwtzip programs both implement the BWT algorithm, bzip2 is more optimized and runs faster than bwtzip. In order to be able to make a direct comparison between the data parallel and task parallel techniques using the same implementation, experiments were conducted with the message-passing mpibwtzip program modified to use a data parallel technique. The 3 stages of the pipeline were combined to form 1 slave task which becomes the same design as the data parallel mpibzip2 algorithm. Figure 5.19 shows the speedup as a function of the number of processors using a data parallel technique with both 159 MB and 1875 MB data sets.

The data parallel and task parallel message-passing techniques with the 159 MB data set provided a similar best maximum speedup of approximately 25. The multi-threaded bzip2 based data parallel BWT performed somewhat better with a speedup of 28. The master became a bottleneck preventing any further speedup. With the 1875 MB data set, the multi-threaded bzip2 based data parallel BWT had the best performance with a maximum speedup of 93, followed by the message-passing task parallel 2-stage pipeline algorithm with a speedup of 48, the message-passing bwtzip based data parallel algorithm with a speedup of 47, message-passing task parallel 3-stage pipeline algorithm with a speedup of 41, and finally the message-passing bzip2 based data parallel algorithm with a speedup of 29.

The data parallel BWT algorithm achieves better speedup with the same number of processors as the task parallel algorithm due to reduced network usage and better load balancing. The task parallel version creates a software pipeline with 3 hops on the network for the 2-stage pipeline and 4 hops for the 3-stage pipeline. The data parallel algorithm requires only 2 hops on the network, one to send data from

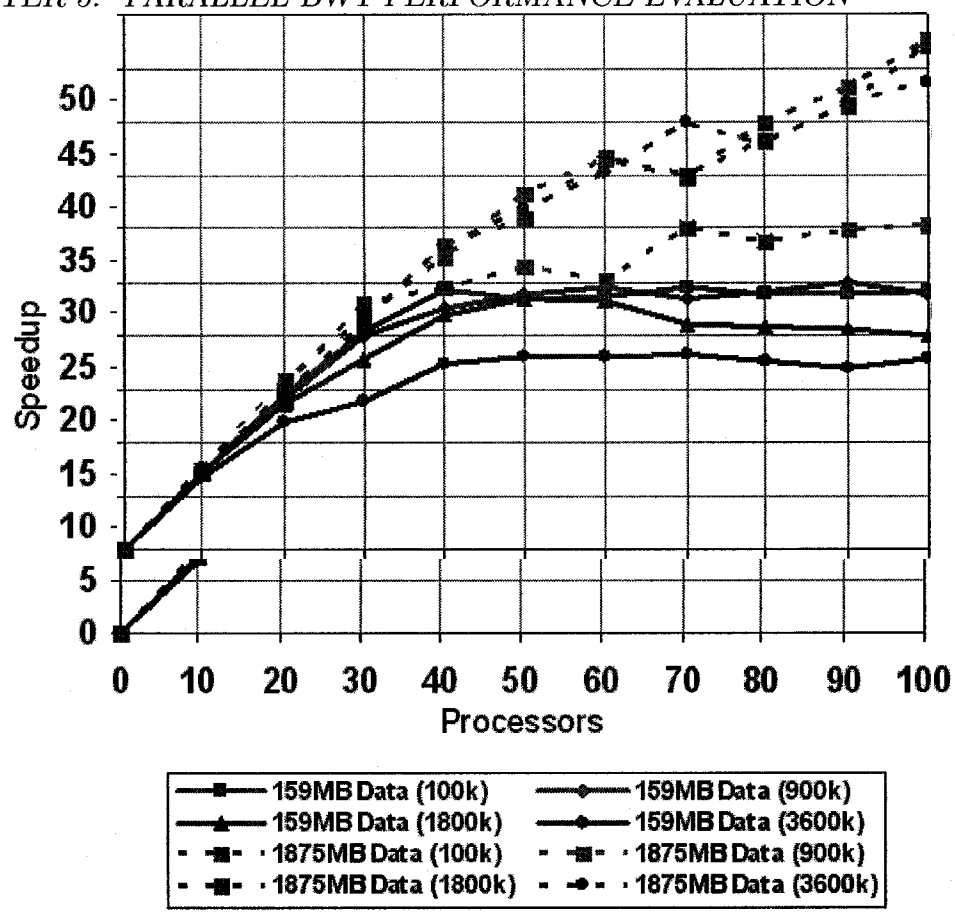

Figure 5.19: Message-passing mpibwtzip speedup using data parallel algorithm on HP Opteron cluster

the master to slave, and one for the slave to return the processed data to the master again. In order to balance the task parallel software pipeline, a generic load balancing split of 84%/8%/8% or 84%/16% was chosen based on profiling 10 data sets. This generic split is less optimal than the data parallel design which provides data to each slave as it is needed, working equally well for all types of data. The greater performance and simpler design of the data parallel algorithm makes it superior to the task parallel algorithm. The bzip2 BWT data format is much more popular and used on many more systems than the bwtzip format which was designed for research purposes. The data parallel bzip2 based multi-threaded or message-passing version of the BWT algorithm would therefore be a better choice if compatibility with others

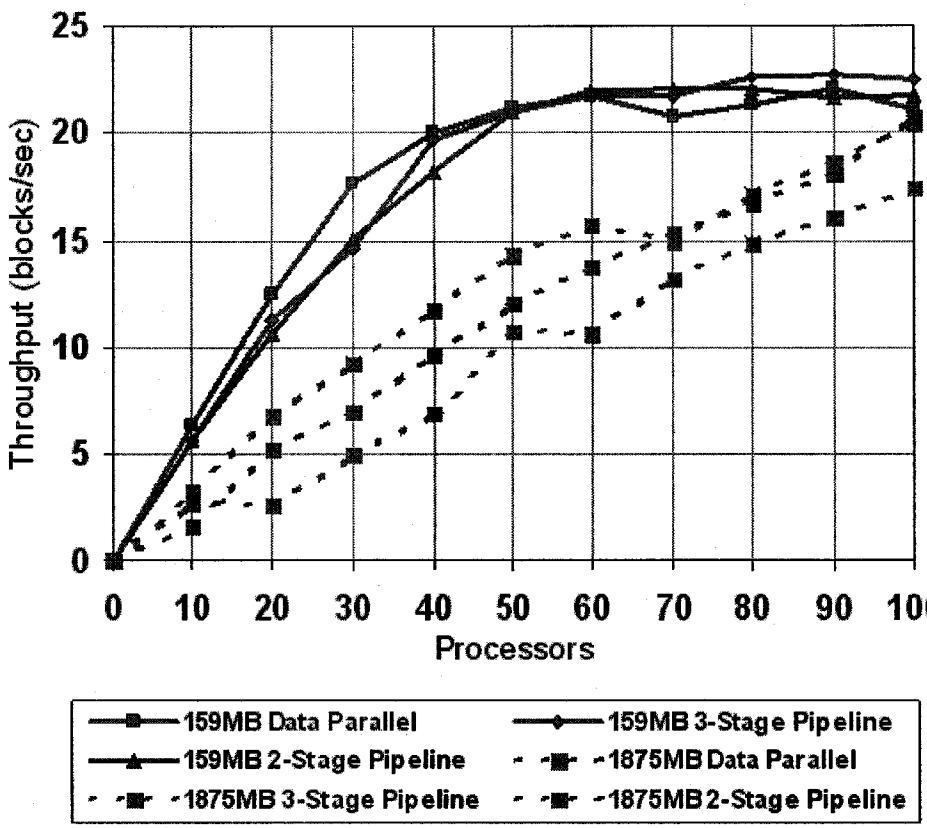

Figure 5.20: Message-passing mpibwtzip average throughput using 900 kB blocks on HP Opteron cluster

is important.

We also wanted to see how throughput and latency differed between the data parallel and task parallel mpibwtzip algorithms. Each block size showed similar differences between the parallelism strategies used. Figure 5.20 shows the average throughput (number of 900 kB blocks processed per second) as a function of the number of processors with both 159 MB and 1875 MB data sets. With the 159 MB data set, the data parallel algorithm achieved the highest throughput up to 40 processors where the throughput levels off and is similar for all algorithms. The 3-stage and 2-stage pipeline algorithms had lower but similar performance. The data parallel algorithm obtained the highest throughput with the 1875 MB data set up to 70 processors where the 2-stage pipeline algorithm begins to achieve similar levels. The 3-stage pipeline

algorithm has better throughput than the 2-stage design up to 10 processors after which its throughput is lower than the rest. The higher throughput attained with the data parallel algorithm can be attributed to the better load balancing achieved where each processor has similar amount of work to perform compared to the generic split used by the task parallel pipeline algorithms. With its more optimal load balancing, the data parallel algorithm can process more data per second than the 3-stage and 2-stage pipelines.

The task parallel software pipeline algorithms overlap the tasks of the BWT compression algorithm which we expected to provide lower latency than the data parallel algorithm. With the 159 MB test file, the data parallel mpibwtzip using 1 processor for the BWT task had an average latency of 1.25 seconds per 900 kB block. Using the 3-stage pipeline mpibwtzip with 1 processor for each of the 3 pipeline stages resulted in an average latency of 1.22 seconds per 900 kB block once the pipeline was filled. The average latency for the 2-stage pipeline once filled was 1.17 seconds per 900 kB block. The reduced network usage of the 2-stage pipeline allowed for lower latency and greater throughput compared to the 3-stage pipeline. When 20 processors were used, the data parallel bwtzip continued to have an average latency of 1.25 seconds per 900 kB block. The 2-stage pipeline however was able to finish processing a 900 kB block of data every 0.27 seconds on average due to the overlapping of tasks. With the 3-stage pipeline, the latency was reduced even further processing a 900 kB block on average every 0.08 seconds. Looking at the results for the 1875 MB test file, the average latency for the data parallel bwtzip with 1 processor for the BWT task was 2.5 seconds per 900 kB block. The 3-stage pipeline when filled provided an average latency of 1.90 seconds per 900 kB block and the 2-stage pipeline an average of 1.93 seconds per 900 kB block using 1 processor for each stage in the pipeline. With the 1875 MB data set, the reduced network usage of the 2-stage pipeline did not provide lower latency. Instead the reduced amount of task overlapping resulted in higher

latency than the 3-stage pipeline when 1 processor for each task was used. Using 20 processors, latency for the data parallel algorithm remained the same, while the 2-stage algorithm completed a block every 0.57 seconds and the 3-stage algorithm every 0.35 seconds on average. As expected, latency was lower for the task parallel algorithms compared to the data parallel one. In the end, the lower latency did not provide a benefit for the task parallel algorithms as the throughput was greater for the data parallel algorithm which allowed the total execution time to be lower. The higher throughput from the better load balancing of the data parallel algorithm contributed more to reducing the execution time of the application than the lower latency of the task parallel algorithms.

5.5 Adaptive Parameter Selection

Currently, the system parameters of the parallel BWT data compression algorithms are fixed or chosen manually by the user at run time. These include the number of processors to use, the block size, and pipeline load balancing. A user may not be sure which block size or number of processors would be best for the data they need to compress. Several techniques can be explored for adaptively selecting the parameters at run time or during execution.

5.5.1 Pipeline Load Balancing

With the task parallel strategy, the assignment of processors to balance the load in the pipeline is fixed for all data and not configurable by the user. Ten different data sets were profiled and the average task usage was used to obtain load balancing parameters with 84% of the processors assigned to stage 1 (*sorting*) of the pipeline, 8% assigned to stage 2 (*MTF+ZLE*), and 8% assigned to stage 3 (*arithmetic coding*). This assignment will not be optimal for most files. When the entropy was calculated for each of the 10 data sets, a pattern emerged where higher entropy generally led to higher usage of the MTF+ZLE and arithmetic coding stages of the pipeline. The

range between the lowest and highest usage of the arithmetic coding stage is not large, from 5.5% to 11.4% but an adaptive technique should provided better load balancing. It would take too much time to fully profile the data beforehand but calculating the entropy first would be feasible.

The entropy of the profiled data sets ranged from 4.7 to 7.5 bits per byte. This information could be used to select the load balancing processor assignments at run time based on the data being compressed. While not perfect, it should provide improved load balancing over the generic 8% selection that adapts to the specific data being compressed. A large number of data sets could be profiled to increase the accuracy of the selection. In the case of our ten data sets, a range of entropy values would be configured to select a processor assignment between 6% and 11%. For example, when the parallel BWT compressor executes, it would first measure the entropy of the data to be compressed. If the entropy was less than 5.2 bits per byte, the load balancing parameters might be configured to assign 6% of the processors to stage 3, 6% to stage 2, and 88% of the processors to stage 1. If the entropy was above 7.2 bits per byte, the selection instead might be 11% assigned to stage 3, 11% assigned to stage 2, and 78% of the processors assigned to stage 1. This parameter selection would happen automatically without user intervention and provide better load balancing for a wider range of data types compared to the current fixed system.

5.5.2 Block Size Selection

The selection of block size is configurable by the user and defaults to the standard 900 kB blocks used by the bzip2 compressor. In order to remain compatible with the sequential version of the BWT algorithms, the block size must be chosen at run time and remain the same for each block of the data being compressed. The choice of block size will affect the size of the compressed output. In general, a larger block size will lead to more compression with diminishing returns. With the 159 MB data

set, the 3600 kB block size achieves only 1.2% better compression than using 900 kB blocks. However, better compression is not always guaranteed. With the 1875 MB data set, the 3600 kB block size results in less compression (0.001%) than with 900 kB blocks.

Processing time with the BWT varies per block depending on the contents of the data. From the experimental results, we observed that the variance in block processing time increases with block size. With the 159 MB data set, the variance in block processing time using 3600 kB blocks was more than 2000 times higher than with 100 kB blocks. A higher variance means that the processor load could be less balanced which may result in lower performance. While smaller block sizes might theoretically lead to better load balancing, the faster processing time of the smaller blocks will cause the slaves to contact the master more frequently and could lead to a bottleneck more quickly giving lower performance in practice.

The size of block also determines the amount of memory required for the parallel BWT algorithm. The larger the block size, the more memory on the system is required. With a large number of processors, a significant amount of memory could be required. An adaptive system would have to take into account the current available memory as well as the file size of the data to be compressed to ensure that the BWT algorithm would not use more memory than acceptable to prevent using the swap file. If the file was small in relation to the number of processors available, the adaptive system might favour a smaller block size in order to increase the number of blocks to process. A smaller block size which partitioned the data into 10 blocks should compress faster on a system with 10 processors than a larger block size that had only 1 or 2 blocks available. From the experimental results, the default 900 kB block size that bzip2 uses provided good performance for all the parallel BWT algorithms.

5.5.3 Processor Selection

The user must select the number of processors they would like to use for the BWT compression algorithm or by default all available processors are selected at run time. If the system is shared amongst users or the system load high, this may not be the desired parameter to use. An adaptive system could first determine the number of processors available. It could be configured to either use up to all available processors or only idle processors based on the current system load. It would then look at the size of the file to be compressed along with the selected block size to calculate the number of blocks to be compressed.

If only one file will be compressed, the adaptive system would assume the user is interested in fastest possible compression time. If the number of blocks is less than the available processors, one processor for each block could be selected for the data parallel algorithm or enough processors to assign a block of data to each of the stage 1 processors for the task parallel pipeline at run time. If the number of blocks is larger than the available processors, the maximum number of available processors would be assigned for the parallel BWT algorithm at run time. As seen in the experimental results, a bottleneck will occur when the number of processors gets to a certain level. Using more than that number of processors would be wasting resources as the file would not be compressed any faster. If the adaptive system could detect that processor utilization was lower than some configured threshold, it could kill slave processes that were not currently processing blocks until the utilization was at or above the threshold. This would occur during execution of the program in order to free up unnecessary resources for other uses.

If multiple files have been selected to be compressed, the adaptive system would assume that the user is interested in shortest time for compressing all files instead of the fastest time for each individual file. This means that instead of using the maximum number of available processors to compress each file in sequence, multi-

ple files would be compressed concurrently. Processor utilization decreases with the parallel BWT algorithm as the number of processors increase. The adaptive system would then use the number of available processors to split between the files to be compressed. If 50 processors were available and 10 files of similar size needed to be compressed, the adaptive system would assign 5 processors to compress each of the 10 files. This should result in shorter overall processing time than assigning 50 processors to compress each of the 10 files in sequence since the processor utilization would lower with the parallel BWT algorithm using 50 processors compared with 5. Larger files generally take longer to compress so the system would have to take into account different file sizes when allocating processors for each file at run time. Even files of similar size have different processing times, depending on the contents of the data. The average processing time per block was found to be longer with higher entropy. With the Linux kernel source data set (5.6 bits per byte of entropy), the block processing time on average was 50% faster than with the binary database (7.1 bits per byte of entropy). The adaptive system could also look at the entropy of the data in each file to better estimate the relative processing times and improve the assignment of processors between files.

5.6 Summary

In this chapter we presented the experimental results and analysis for both the data parallel and task parallel versions of the BWT data compression algorithm. The data parallel strategy achieved more speedup than the task parallel strategy and had a simpler design. The multi-threaded implementation of the data parallel strategy obtained significant speedup and performed better than the message-passing strategy. All message-passing implementations obtained significant speedup compared to the sequential BWT.

In the next chapter, we provide some concluding remarks and discuss possible future work.

Chapter 6

Conclusion and Future Work

6.1 Summary

In this thesis, we described multi-threaded and message-passing implementations of the lossless Burrows-Wheeler Transform data compression algorithm using both data parallel and task parallel strategies. The parallel BWT algorithms were implemented and evaluated on shared memory and distributed memory systems.

The results showed significant speedup for both the multi-threaded and message-passing versions of the BWT using a data parallel approach. Because the compressed data output from the parallel algorithm is in a form that can be decompressed using the sequential bzip2 algorithm, users do not need access to a parallel machine to recover the data. The results also showed a significant speedup for the message-passing version of the BWT using a task parallel approach. The output from the task parallel algorithm is compatible with the sequential version of bwtzip. Due to the large amount of memory management required, the multi-threaded task parallel version did not obtain any speedup and ran slower than the sequential. This is not inherent in the design of the BWT algorithm but in the implementation. Software developers who may want to parallelize their code in the future on shared memory

systems should be careful to minimize the amount of memory that is reserved and released dynamically. Due to the modular nature of the BWT algorithm and the high computation to communication ratio, both data parallel and task parallel strategies for parallelizing the algorithm were successful.

Now that mainstream processors are being designed with multiple cores, the general public will be able to take advantage of multi-threaded software. Since the parallel bzip2 software provides linear speedup for workstations and small servers, and is fully compatible with the sequential bzip2, there has already been much interest in the algorithms. A number of companies such as Debian, FreeBSD, Gentoo, NetBSD, RedHat, and Solaris have packaged the parallel BWT algorithms for their UNIX distributions. We have decided to make the software available to the public and the source code for the multi-threaded pbzip2 can be downloaded at: **http://compression.ca/pbzip2/**

6.2 Future Work

- **Multiple Queues**

  In this thesis, the multi-threaded algorithm used one FIFO queue for task scheduling. When a large number of processors are used, threads waiting for work from the queue become a bottleneck. A different task scheduling strategy using multiple-queues could be investigated. The processors would be partitioned into groups and each group would have its own queue, reducing the amount of time that a particular processor would have to wait for access to data. With less competition for each of the queues, the processors should be able to obtain new work more quickly and thus improve overall performance.

  Similarly with the message-passing algorithms, when many processors are being used on the cluster, there are slave processors waiting for the master to respond with more work. A strategy where slave processors are partitioned into groups

and each group communicates with a different master processor could be used. With all slave processors not competing for the same master, performance may be improved.

- **Slave Thread for Master Processor**

  The data parallel message-passing version of the BWT algorithm uses a master/slave model where the master processor hands out work for the slave processors to perform. The master reads and writes data but does not do any compression itself. The master processor performs many disk operations but does not require much computational CPU time. It might be possible to increase speedup further by adding a slave thread to the master processor so it can also compress data with the leftover CPU cycles available after the read and write operations.

- **Separate File Writing Processor**

  The task parallel message-passing version of the BWT algorithm uses a master/slave model with a software pipeline. The master processor both reads the input data and writes the final compressed data. Since the last stage in the pipeline does not need to receive any communication back from the master processor when it sends a completed block of data, a separate file writing processor could be used instead. A processor dedicated to writing the compressed data to disk might offload enough processing on the master to increase performance with a large number of processors.

Bibliography

[1] Team 17. *Worms2 Data Set*, 1998. ftp://ftp.team17.com/pub/t17/worms2/demo/Worms2Demo10OctA.zip (accessed June 2, 2006).

[2] Ross Arnold and Tim Bell. *Canterbury Corpus Data Set*, 1997. http://corpus.canterbury.ac.nz/descriptions/ (accessed June 2, 2006).

[3] Jay Berg. *Elliptic Curve Database*, 2004. http://eCompute.org (accessed February 23, 2006).

[4] Werner Bergmans. *Maximum Compression Data Set*, 2003. http://www.maximumcompression.com/data/files/ (accessed June 2, 2006).

[5] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Systems Research Center, 1994.

[6] David R. Butenhof. *Programming with POSIX Threads*, pages 1–4. Addison-Wesley, 1997.

[7] Rajkumar Buyya. *High Performance Cluster Computing: Programming and Applications*, volume 2, pages 15–23. Prentice Hall PTR, 1999.

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

[9] J. G. Cleary and W. J. Teahan. Unbounded length contexts for ppm. *The Computer Journal*, 40(2/3):67–75, 1997.

[10] 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.

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

[12] P. Fenwick. Block-sorting text compression — final report. Technical Report 130, The University of Auckland, Department of Computer Science, 1996.

[13] Martin Fleury and Andrew Downton. *Pipelined Processor Farms - Structured Design for Embedded Parallel Systems*, pages 6–29. John Wiley and Sons Inc., 2001.

[14] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, chapter 2. Addison-Wesley, 1995.

[15] P. Franaszek, J. Robinson, and J. Thomas. Parallel compression with cooperative dictionary construction. U.S. Patent No. 5,729,228, 1998.

[16] Jeff Gilchrist. *WAV Audio Data Set*, 1995. http://compression.ca/act/act-files.html (accessed June 2, 2006).

[17] Jeff Gilchrist. Archive comparison test. Technical report, 2002. http://compression.ca (accessed January 10, 2006).

[18] Jeff Gilchrist. Parallel compression with bzip2. In *16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*, pages 559–564, 2004.

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

[20] Paul G. Howard and Jeffery Scott Vitter. Arithmetic coding for data compression. Technical Report CS–1994–09, Duke University, 1994.

[21] Jacob Kitzman and Guilherme Issao Fujiwara. Parallel file compression. Technical Report 18.337J, Massachusetts Institute of Technology, May 2005.

[22] John Kominek. *Waterloo BragZone Colour Data Set*, 1995. http://links.uwaterloo.ca/colorset.base.html (accessed June 2, 2006).

[23] Charles M. Kozierok. *The TCP/IP Guide*, pages 35–36. No Starch Press, 2005.

[24] Argonne National Laboratory. *The Message Passing Interface (MPI) standard*, 2006. http://www-unix.mcs.anl.gov/mpi/ (accessed May 18, 2006).

[25] Stephan T. Lavavej. *The bwtzip home page*, 2003. http://nuwen.net/bwtzip.html (accessed May 9, 2006).

[26] Jie Liang. New trends in multimedia standards: Mpeg4 and jpeg2000. *Informing Science*, 2(4):101–106, 1999.

[27] Matt Mahoney. *Large Text Compression Data Set*, 2006. http://cs.fit.edu/ mmahoney/compression/text.html (accessed May 23, 2006).

[28] Jose Martinez, Rene Cumplido, and Claudia Feregrino. An fpga-based parallel sorting architecture for the burrows wheeler transform. In *IEEE International Conference on Reconfigurable Computing and FPGAs*, page 17, September 2005.

[29] Peter S. Pacheco. *Parallel Porgramming with MPI*, pages 11–37. Morgan Kaufmann Publishers Inc., 1997.

[30] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*, pages 70–72. McGraw-Hill Professional, 2003.

[31] J. Seward. On the performance of bwt sorting algorithms. In *IEEE Data Compression Conference*, pages 173–182, 2000.

[32] Julian Seward. *The bzip2 and libbzip2 official home page*, 2005. http://www.bzip.org (accessed March 15, 2006).

[33] Julian Seward. *The bzip2 manual*, 2005. http://www.bzip.org/1.0.3/bzip2-manual-1.0.3.html (accessed March 15, 2006).

[34] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948. http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html (accessed September 17, 2006).

[35] Lynn M. Stauffer and Daniel S. Hirschberg. Parallel text compression - REVISED. Technical Report ICS-TR-91-44, University of California, Irvine, 1993.

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

[37] Linus Torvalds. *Linux 2.4.23 Kernel Source Data Set*, 2003. http://www.kernel.org/pub/linux/kernel/v2.4/ (accessed February 23, 2006).

[38] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[39] John Walker. *ENT - Pseudorandom Number Sequence Test*, 1998. http://www.fourmilab.ch/random/ (accessed September 16, 2006).

[40] Gregory K. Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44, April 1991.

[41] Ian Witten and Tim Bell. *Calgary Corpus Data Set*, 1990. http://links.uwaterloo.ca/calgary.corpus.html (accessed June 2, 2006).

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