LITERATURE REVIEW: — Parallel Data Compression —

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

October 8, 2003

1 Introduction

Parallel computing allows software to run faster by using multiple processors and performing multiple computations simultaneously when possible. The majority of general purpose software is designed for sequential machines and does not take advantage of parallel computing. Lossless data compression software like WinZip, gzip, and bzip2, used for compressing any files found on a computer, are designed for sequential machines. The data compression algorithms require a lot of processing power to analyze and then encode the files in a smaller form. Writing 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. A parallel version of bzip2 shall be created and tested on a dual-processor machine and parallel cluster to determine if and how much speedup is achieved. The bzip2 program was chosen because it achieves very good compression and is available in library and source form, free under a BSD-style license. The limited research papers available on parallel data compression use theoretical algorithms designed to run on impractical parallel machines that do not exist or are very rare in the real world. The idea with the parallel bzip2 compression program is to make a practical parallel compression utility that works with real machines.

2 Literature Review

Lossless data compression is used to compact files or data into a smaller form. It is often used to package up software before it is sent over the Internet or downloaded from a web site to reduce the amount of time and bandwidth required to transmit the data. Lossless data compression has the constraint that when data is uncompressed, it must be identical to the original data that was compressed. Graphics, audio, and video compression such as JPG, MP3, and MPEG on the other hand use lossy compression schemes which throw away some of the original data to compress the files even further. We will be focusing on the lossless kind. There are generally two classes of lossless compressors: dictionary compressors and statistical compressors. Dictionary compressors (such as Lempel-Ziv based algorithms) build dictionaries of strings and replace entire groups of symbols. The statistical compressors develop models of the statistics of the input data and use those models to control the final output [6].
2.1 BZIP2

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

2.2 Dictionary Algorithms

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

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

Some investigation into creating parallel dictionary compression has been performed. Once the dictionary has been created in an LZ based algorithm, a greedy parsing is 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. Another option is to break the input data into blocks and compress each block on a different processor [11]. Compressors generally work better with larger blocks of data so while overall speed may be increased by splitting up the input data into blocks, the compression ratio may be decreased at the same time. A careful balance would have to be selected between the two. Stauffer a year later expands on previous paral-
lel dictionary compression schemes on the PRAM by using a parallel longest fragment first (LFF) algorithm to parse the input instead of the greedy algorithm [12]. The LFF algorithm parses the input by continuously locating the longest substring in the uncoded section of the input data which matches a dictionary entry. It then replaces the substring with the corresponding dictionary reference. The LFF algorithm in general performs better than the greedy algorithm but is not often used in sequential implementations because it requires two passes over the input. With the PRAM, using the LFF parsing can be performed over the entire input in one step. They assume a static dictionary which is stored as a suffix tree, and that dictionary references are of a fixed length. A processor is assigned to each position of the input string which then computes the list of lengths of matches between the dictionary and the input beginning at its assigned position. With a maximum dictionary length of \( M \), the LFF parsing can be done in \( O(M \log M + \log n) \) time with \( O(n) \) processors. They later refine the algorithm to improve the number of processors required to \( O(n/\log n) \). In practice, this algorithm would never work since very few people if anyone would have a machine with enough processors to satisfy the requirements.

2.3 Statistical Algorithms

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

Arithmetic coding is a statistical compression technique that uses estimates of the probabilities of events to assign code words. Ideally, short code words are assigned to more probable events and longer code words are assigned to less probable events. Theoretically, arithmetic codes assign one ”code word” to each possible data set. The arithmetic coder must work together with a modeler that estimates the probabilities of the events in the coding. To obtain good compression, a good probability model and efficient way of representing the probability model are required. The models can be adaptive, semi-adaptive, or non-adaptive. Adaptive models dynamically estimate the probably of each event based on preceding events. Semi-adaptive models use a preliminary pass of the data to gather some statistics, and non-adaptive models use fixed probabilities for all data. An advantage of arithmetic coding is the separation of coding and modeling since it allows the complexity of the modeler to change without having to modify the coder. The disadvantage is that is runs more slowly and is more complex to implement than LZ based algorithms [9].

2.4 Burrows-Wheeler Transform

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

The algorithm works by transforming a string $S$ of $N$ characters by forming the $N$ rotations (cyclic shifts) of $S$, sorting them lexicographically, and extracting the last character of each of the rotations. A string $L$ is then formed from these extracted characters, where the $i$th character of $L$ is the last character of the $i$th sorted rotation. The algorithm also computes the index $I$ of the original string $S$ in the sorted list of rotations. With only $L$ and $I$ there is an efficient algorithm to compute the original string $S$ when undoing the transformation for decompression. Burrows and Wheeler explain that much of the time, the algorithm is performing sorts which may be another area where using a parallel sorting algorithm may increase the speed of BWT. They also explain that to achieve good compression, a block size of sufficient value must be chosen, at least 2 kilobytes. Increasing the block size also increases the effectiveness of the algorithm at least up size of several megabytes. If enough memory is present, the block size could also equal the size of the file to be compressed (i.e., one large block) which would be good for compression but reduce the parallelism of the algorithm.

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

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

It was discovered that with sequences of length $n$ taken from a finite-memory source that the performance of BWT based algorithms converge to the optimal performance at a rate of $O(\log n / n)$ surpassing that of LZ77 which is $O(\log \log n / \log n)$ and LZ78 which is $O(1 / \log n)$ [5]. They also note that while many algorithms use sequential codes on the BWT output, the overall data compression algorithms are non-sequential since the transform requires simultaneous access to all symbols of the data string (or blocks of the data string if the block size is small than the entire data). This is something that can be taken advantage of in a parallel implementation.
2.5 Prediction by Partial Match

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

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

References


