Empirical Speedup Study of Truly Parallel Data Compression

James A. Edwards
University of Maryland
College Park, Maryland
jedward5@umd.edu

Uzi Vishkin
University of Maryland
College Park, Maryland
vishkin@umiacs.umd.edu

ABSTRACT

We present an empirical study of novel work-optimal parallel algorithms for Burrows-Wheeler compression and decompression of strings over a constant alphabet. To validate these theoretical algorithms, we implement them on the experimental XMT computing platform developed especially for supporting parallel algorithms at the University of Maryland. We show speedups of up to 25x for compression, and 13x for decompression, versus bzip2, the de facto standard implementation of Burrows-Wheeler compression. Unlike existing approaches, which assign an entire (e.g., 900KB) block to a processor that processes the block serially, our approach is "truly parallel" as it processes in parallel the entire input. Besides the theoretical interest in solving the "right" problem, the importance of data compression speed for small inputs even at great expense of quality (compressed size of data) is demonstrated by the introduction of Google's Snappy for MapReduce. Perhaps surprisingly, we show feasibility of holding on to quality, while even beating Snappy on speed.

In turn, this work adds new evidence in support of the XMT/PRAM thesis: that an XMT-like many-core hardware/software platform may be necessary for enabling general-purpose parallel computing. Comparison of our results to recently published work suggests 70x improvement over what current commercial parallel hardware can achieve.

Keywords

parallel, PRAM, Burrows-Wheeler, lossless compression

1. INTRODUCTION

A *lossless compression function* is an invertible function $C(\cdot)$ that accepts as input a string S of length n over some alphabet Σ and returns a string of length $\Theta(n)$ over some alphabet Σ' where, on average, fewer bits are required to represent $C(S)$ than S. A *lossless compression algorithm* for a given lossless compression function is an algorithm that accepts S as input and produces $C(S)$ as output; the corresponding *lossless decompression algorithm* accepts $C(S)$ for some S as input and produces S as output.

In [5], Burrows and Wheeler describe their eponymous lossless compression algorithm and corresponding decompression algorithm; it has been shown [2, 3] to be among the best such algorithms, and its operation is reviewed in this paper. The *Burrows-Wheeler (BW) Compression problem* is to compute the lossless compression function defined by the algorithm of [5], and the *Burrows-Wheeler (BW) Decompression problem* is to compute its inverse. The algorithm of [5] solves the BW Compression problem in $O(n \log^2 n)$ serial time and solves BW Decompression problem in $O(n)$ serial time. Later work reduced a critical step of the compression algorithm to the problem of computing the suffix array of S, for which linear-time algorithms are now known, so both problems can now be solved in $O(n)$ optimal serial time.

We propose an $O(\log^2 n)$-time, $O(n)$-work PRAM algorithm for solving the BW Compression problem and a $O(\log n)$-time, $O(n)$-work PRAM algorithm for solving the BW Decompression problem. These algorithms appear to be the first polylogarithmic-time work-optimal parallel algorithms for any standard lossless compression scheme.

We implement our parallel algorithm and experimentally validate it. A parallel-algorithmic approach to BW compression may not have been seriously considered in the past because the fine-grained parallelism provided by such an approach is difficult for existing computing hardware to exploit. However, the Explicit Multi-Threading (XMT)[1] architecture developed at the University of Maryland was designed specifically to provide good performance on such algorithms. Using our parallel algorithm in conjunction with XMT, we obtain speedups of up to 25x for compression and 13x for decompression for small inputs (say, up to 1MB) where no speedup was possible before. This is especially important for real-time applications, where single-task completion time is more important than throughput.

In passing, we note that commonly-used compression programs divide the input into uniformly-sized blocks and apply a serial implementation of BW compression to each block independently. The blocks can be compressed in parallel; however, this does not solve the BW Compression problem for the original input and thus is not a parallel algo-

[1]Not to be confused with the Cray XMT

rithm for solving it. It is worth noting that our parallel-algorithmic approach is orthogonal to the foregoing block-based approach, and the two approaches could conceivably be combined to obtain better speedups than either alone.

One application where our implementation shows a distinct advantage over existing compression libraries is in compressing data that is sent over a network. In warehouse-scale computers such as those found in data warehouses, the bandwidth available between various pairs of nodes can be extremely different, and for pairs where the bandwidth is low can be debilitating [17]. A way to mitigate this is to compress data before it is transmitted over the network and decompress it on the other side. This approach is taken, for example, by Google via their Snappy [15] library. The goal of Snappy is to compress data very quickly, even at the expense of less compression, providing a larger increase in effective network bandwidth than other libraries for all but very low-bandwidth networks. As shown in Section 5.2, our implementation outperforms Snappy and similar libraries for point-to-point bandwidths of up to 1 Gbps.

This technical report augments the theory results of [11] with experimental speedups. For an extended description of the algorithms, please see the companion report [11].

1.1 Related Work

There are applications where BW compression would be useful but is not currently used because of performance. One such application is JPEG image compression. JPEG compression consists of a lossy compression stage followed by a lossless stage. The work [38] considered replacing the currently-used lossless stage with the BW compression algorithm. For high-quality compression of "real-world" images such as photographs, this yielded up to a 10% improvement, and for the compression of "synthetic" images such as company logos, the improvement was up to 30%. The author cites execution time as the main deficiency of this approach.

A commonly-used, serial implementation of the block-based approach noted above is bzip2 [31]; the algorithm it applies to each block is based on the original BW compression algorithm of [5]. There are also variants of bzip2, such as pipeline bzip [14], that compress multiple blocks simultaneously. However, these variants do not achieve speedup on single blocks while our approach does. There exists at least one implementation of a linear time serial algorithm for BW compression, bwtzip [24]. However, bwtzip is a research-grade implementation that emphasizes modularity over performance, unlike the focus of this paper.

The survey paper [12] articulates some of the issues involved in parallelizing BW for a GPU; decompression is not discussed. The author gives an outline of an approach for making some parts of the algorithm parallel and claims that the remaining parts would not work well on GPUs due to exhibiting poor locality. [28] reports such parallelization, and indeed was unable to demonstrate a speedup for compression using the GPU, instead obtaining a slowdown of 2.78x. Note that our results reflect a speedup of 70x over [28]. Parallelization of decompression was left as future work, and no speedups or slowdowns are reported. Furthermore, no asymptotic complexity analysis is given, and our own analysis shows their algorithm to be non-work-optimal. To their credit, they appear to be the first to formulate MTF encoding (Section 2.1.2) in terms of a binary associative operator. However, two challenges, (i) work-optimal parallelization of BW and (ii) feasibility of speedups on buildable hardware, remained unmet.

A parallel algorithm for Huffman decoding is given in [23]. However, the algorithm is not analyzed therein as a PRAM algorithm, and its worst case run time is $O(n)$. Our PRAM algorithm for Huffman decoding runs in $O(\log n)$ time.

The rest of the paper is organized as follows: Section 2 gives an overview of the serial BW compression and decompression algorithms and Section 3 describes our parallel algorithms for the same along with their complexity analysis, ending the theoretical part of the paper. The remainder of the paper is devoted to experimental validation of the algorithms. Section 4 describes the experimental comparison to bzip2, Section 5 contains a discussion of the results we obtained, and Section 6 concludes.

2. SERIAL ALGORITHM

In their original paper, Burrows and Wheeler [5] describe a lossless data compression algorithm consisting of three stages in the following order: a reversible block-sorting transform (BST)[2], move-to-front (MTF) encoding, and Huffman coding. The corresponding decompression algorithm performs the inverses of these stages in reverse order: Huffman decoding, MTF decoding and inverse BST (IBST). See Figure 1.

Given an input string of length n, their original decompression algorithm runs in $O(n)$ serial time, as do all stages of their compression algorithm except the (forward) BST, which requires $O(n \log^2 n)$ serial time [32]. More recently, linear-time serial algorithms [19, 26] have been developed to compute suffix arrays, and the problem of finding the BST of a string can be reduced to that of computing its suffix array, so Burrows-Wheeler (BW) compression and decompression can be performed in $O(n)$ serial time. The linear-time BST algorithms are relatively involved, so we refrain from describing them here and instead refer interested readers to the cited papers.

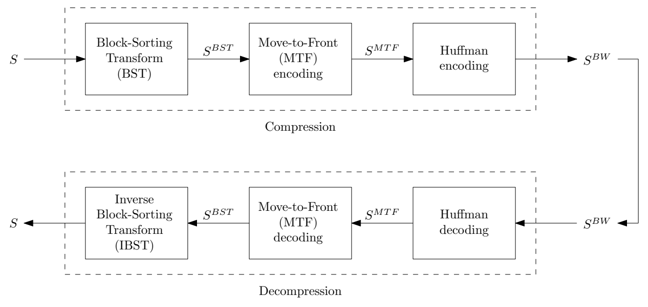

S
Block-Sorting Transform (BST)
S^{BST}
Move-to-Front (MTF) encoding
S^{MTF}
Huffman encoding
S^{BW}
Compression
S
Inverse Block-Sorting Transform (IBST)
S^{BST}
Move-to-Front (MTF) decoding
S^{MTF}
Huffman decoding
S^{BW}
Decompression

Figure 1: Stages of BW compression and decompression.

[2]This transform is also known as the Burrows-Wheeler Transform (BWT). We refrain from using this name to avoid confusion with the similarly-named Burrows-Wheeler compression algorithm which employs it as a stage.

2.1 Compression

Given a string S of length n from an alphabet Σ, where $|\Sigma|$ is constant with respect to n, the compression algorithm proceeds in three stages as follows.

2.1.1 *Block-Sorting Transform (BST)*

The BST stage takes S as input and produces as its output S^{BST}, a permutation of S. S^{BST} is formed by making a list of all the rotations of S (each of which is also a string of length n), sorting the list of rotations lexicographically, and outputting the last character of each rotation in the sorted list starting with the first. See Figure 2. The BST has two properties that make it useful for lossless compression: (1) it has an inverse and (2) S^{BST} tends to have many occurrences of any given character in close proximity, even when S does not. Property (1) ensures that the decompressor can reconstruct S given only S^{BST} and Property (2) allows the following stages to work effectively.

$$\text{banana\$} \xrightarrow{\text{rotate}} \begin{array}{l} \text{banana\$} \\ \text{anana\$b} \\ \text{nana\$ba} \\ \text{ana\$ban} \\ \text{na\$bana} \\ \text{a\$banan} \\ \text{\$banana} \end{array} \xrightarrow{\text{sort}} \underbrace{\begin{array}{l|l|} \text{\$banan} & \text{a} \\ \text{a\$bana} & \text{n} \\ \text{ana\$ba} & \text{n} \\ \text{anana\$} & \text{b} \\ \text{banana} & \text{\$} \\ \text{na\$ban} & \text{a} \\ \text{nana\$b} & \text{a} \end{array}}_{M} \xrightarrow{\text{output}} \text{annb\$aa}$$

Figure 2: BST of the string "banana\$". The sorted list labeled M can be viewed as a matrix of characters.

The critical step in the BST algorithm is the sorting of the list of rotations of S. The BST algorithm given in [5] is actually a combination of two algorithms: direct comparison and doubling [32]. The direct comparison algorithm sorts the list of rotations of S using a comparison-based sorting algorithm that compares rotations in the list character-by-character. Therefore, it requires $O(n \log n)$ string comparisons, and since comparing two strings of length n requires $O(n)$ comparisons in the worst case, the direct comparison algorithm has a worst-case running time of $O(n^2 \log n)$.

The doubling algorithm works in $O(\log n)$ iterations. Each iteration applies comparison-based sorting to the current list of rotations of S as in the direct comparison algorithm. However, each comparison is limited to the first d characters of the rotations being compared, with $d \geq 2$ a constant, so the list will not be completely sorted if two rotations begin with the same sequence of d characters. Afterwards, a new string S' is constructed over the alphabet $[0, n-1]$ such that S_i, $0 \leq i < n$, is the rank of the ith rotation in the partially-sorted list. To complete the iteration, S is replaced by S'. During the next iteration, up to d character comparisons are made again as in the first iteration, but now each character in S' gives the rank of d consecutive characters in S, so the character comparisons are spaced d characters apart to give a new partially-sorted list based on the ranks of the first d^2 characters. The spacing increases by a factor of d each iteration, so after $\lceil \log_d n \rceil$ iterations, all comparisons are guaranteed to reach the end of the string. Since d is constant, each comparison takes constant time, so each partial sort takes $O(n \log n)$ time. Because the partially-sorted list is in non-decreasing order, ranking its elements can be done in $O(n)$ time. Therefore, each iteration takes $(n \log n)$ time, and the overall doubling algorithm takes $O(n \log^2 n)$ time.

The two algorithms can be combined as follows: begin by using direct comparison, keeping track of cumulative number of character comparisons that exceed a depth of d. If the cumulative number exceeds some constant value, switch to the doubling algorithm. This heuristic ensures that the direct-comparison algorithm never performs more than $O(n \log n)$ character comparisons before it either completes or is abandoned in favor of the doubling algorithm. Therefore, the overall BST algorithm takes at most $O(n \log^2 n)$ time.

2.1.2 *Move-to-Front (MTF) Coding*

The input to the MTF coding stage is the string S^{BST}. Given an initial list L_0 of the characters in Σ in arbitrarily-defined order, the output, denoted by S^{MTF}, is a string of length n over the alphabet of integers $\Sigma' = [0, |\Sigma| - 1]$. See Figure 3. MTF coding exploits property (2) of the BST to produce a string that can be readily compressed by an entropy coding technique such as Huffman coding. MTF coding is performed by scanning the characters of S^{BST} in order of increasing index. For each character $c = S_i^{BST}$, the output character S_i^{MTF} is set to the index of c in L_i, and then c is moved to the front of L_i to produce L_{i+1}. That is, L_{i+1} is set to L_i, then c is removed from L_{i+1} and reinserted as the first element of L_{i+1}. Because $|\Sigma|$ is constant, the size of L is constant as well, and each update to L takes $O(1)$ time. The list is updated n times, so MTF coding takes $O(n)$ time. See Figure 4.

The purpose of MTF coding is to maintain L as a list of the characters seen so far in most-recently used (MRU) order. As a consequence, c will now have an index in L of zero, and if c is immediately followed by more repetitions of c, then the MTF coder will output zero for each of those subsequent repetitions. Each run of m repetitions of any character in S^{BST} will be converted to a nonzero integer followed by $m - 1$ zeros in S^{MTF}. Even if c is not immediately followed by another occurrence of c, there will likely be one nearby. In that case, since only a few characters are moved ahead of c between the two occurrences of c in S', c will be assigned an index close to zero.

2.1.3 *Huffman Coding*

The input to the Huffman coding stage is the string S^{MTF}, and it produces as output (1) the string S^{BW}, a binary string (i.e., a string over the alphabet $\{0, 1\}$) whose length is $\Theta(n)$ and (2) a coding table T, whose size is constant given that $|\Sigma|$ is constant. In S^{MTF}, smaller integers tend to occur more frequently than larger integers, even if the characters in Σ occur an equal number of times in S. Therefore, S^{MTF} is amenable to entropy coding, even when S is not. This means that S^{BW} is typically shorter than any fixed-length encoding of S (e.g., the way it was originally stored on disk).

Huffman coding proceeds in three steps. In step 1, S^{MTF} is scanned once to build a frequency table F indicating how many times each character in Σ occurs in S^{MTF}; this takes $O(n)$ time. In step 2, the coding table T is constructed using a heap-based algorithm that takes only F as input. Since $|\Sigma|$ is constant, the size of F is also constant, so this takes $O(1)$ time. In step 3, S^{MTF} is scanned once more, and for each

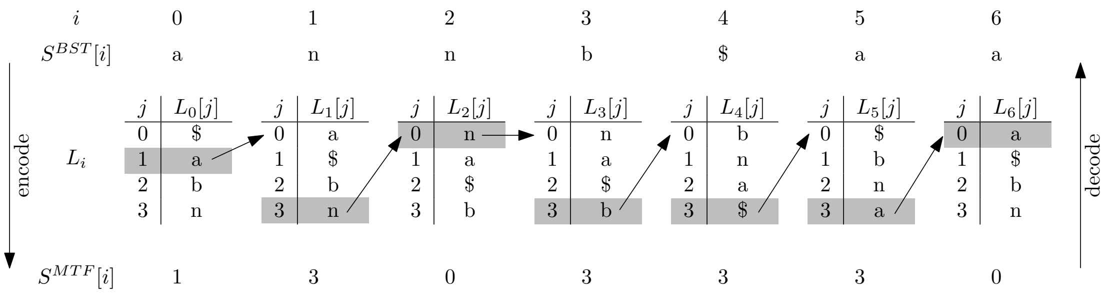
$$\Sigma = \{\$, a, b, n\}$$
$$S^{BST} = (a, n, n, b, \$, a, a)$$

assumed prefix: $i = 0$–3

i	0	1	2	3	4	5	6	7	8	9	10
$S^{BST}[i]$	n	b	a	\$	a	n	n	b	\$	a	a
$prev[i]$	-	-	-	-	2	0	5	1	3	4	9
$C[i]$	-	-	-	-	{\$}	{\$,a,b}	{}	{\$,a,n}	{a,b,n}	{\$,b,n}	{}
$\lvert C[i]\rvert$	-	-	-	-	1	3	0	3	3	3	0

$$S^{MTF} = (1, 3, 0, 3, 3, 3, 0)$$

Figure 3: MTF of the string "annb\$aa". $C[i]$ is the set of characters between $S^{BST}[i]$ and its previous occurrence.



i 0 1 2 3 4 5 6
S^BST[i] a n n b $ a a
encode
L_i
j L_0[j]: 0 $, 1 a, 2 b, 3 n
j L_1[j]: 0 a, 1 $, 2 b, 3 n
j L_2[j]: 0 n, 1 a, 2 $, 3 b
j L_3[j]: 0 n, 1 a, 2 $, 3 b
j L_4[j]: 0 b, 1 n, 2 a, 3 $
j L_5[j]: 0 $, 1 b, 2 n, 3 a
j L_6[j]: 0 a, 1 $, 2 b, 3 n
decode
S^MTF[i] 1 3 0 3 3 3 0

Figure 4: MTF encoding and decoding. Observe that $S^{BST}[i] = L_i[S^{MTF}[i]]$. In both the encoder and the decoder, the shaded elements are moved to the front of the list according to the arrows. In the encoder, the shaded element is identified by searching the list L_i for the character $S^{BST}[i]$. In the decoder, the shaded element is chosen to be the one whose index is $j = S^{MTF}[i]$; no searching is necessary.

character S_i^{MTF}, the corresponding codeword $T(S_i^{MTF})$ is written to S_{BW}; this takes $O(n)$ time. See Figure 5. Overall, Huffman coding takes $O(n)$ time.

$$T = \begin{array}{l} 0 \rightarrow 10 \\ 1 \rightarrow 11 \\ 3 \rightarrow \ 0 \end{array}$$

$$S^{MTF} = (1, 3, 0, 3, 3, 3, 0)$$

$$S^{BW} = 11\ 0\ 10\ 0\ 0\ 0\ 10$$

Figure 5: Huffman table and encoding of S^{MTF} (spaces added for clarity). Recall that this is, in fact, the compression of the original string "banana\$".

The output of the entire BW compression algorithm has size $\Theta(n)$ and consists of S_{BW} and T. The overall run time is dominated by the BST stage. If a linear-time suffix array algorithm is used to compute the BST, the overall runtime is $O(n)$. If the BST algorithm described herein is used instead, the overall runtime is $O(n \log^2 n)$.

2.2 Decompression

With the exception of the IBST, the decompression algorithm is simply the reverse of the compression algorithm: given S_{BW} and T, S_{BST} can be constructed in $O(n)$ time by applying the respective algorithms with the lookup tables inverted. The major difference is the IBST, which is simpler than the BST and consists of two steps. In step 1, the individual characters of S_{BST} are sorted using stable integer sorting, which takes $O(n)$ time. The resulting list of ranks is equivalent to a linked ring (a linked list whose tail points back to its head) of the characters in S_{BST} in the order they appear in S; see [5] or [11] for an explanation of why this is true. In step 2, the linked ring is traversed once, beginning from the character \$, to produce the characters of S in reverse order; this traversal takes $O(n)$ time. Therefore, the IBST, and thus the overall BW decompression algorithm, has a runtime of $O(n)$.

3. PARALLEL ALGORITHM

The parallel BW compression and decompression algorithms follow the same sequence of stages as the foregoing serial algorithms, but the sequential algorithm of each stage is replaced by an equivalent PRAM algorithm. As is the case in the serial algorithm, the dominant stage of the compression algorithm is the BST stage. Our PRAM algorithm for the BST stage, described below, requires the same work as the serial BST algorithm described above. If an $O(n)$-work compression algorithm is desired, the work-optimal algorithm of [30] can be used to compute the BST in $O(\log^2 n)$ time.

3.1 Compression

As in the serial algorithm, the input is a string S of length n over an alphabet Σ, where $|\Sigma|$ is constant with respect to n. The overall PRAM compression algorithm consists of the following three steps.

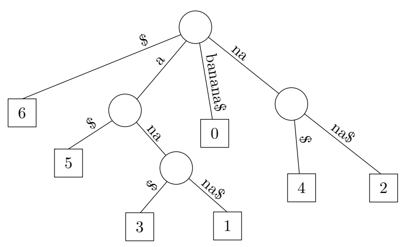
3.1.1 Block-Sorting Transform (BST)

The BST of a string S of length n can be computed as follows. Add a character \$ to the end of S that does not appear elsewhere in S. Sorting all rotations of S is equivalent to sorting all suffixes of S, as \$ never compares equal to any other character in S. Such sorting is equivalent to computing the suffix array of S, which can be derived from a depth-first search (DFS) traversal of the suffix tree of S (see

Figure 6). The suffix tree of S can be computed in $O(\log^2 n)$ time and $O(n)$ work using the algorithm of [30]. The order that leaves are visited in a DFS traversal of the suffix tree can be computed using the Euler tour technique [34] within the same complexity bounds, yielding the suffix array of S. Given the suffix array SA of S, we derive S^{BST} from S in $O(1)$ time and $O(n)$ work as follows:

$$S^{BST}[i] = S[(SA[i] - 1)_{\bmod n}], 0 \leq i < n$$

Overall, computing the BST takes $O(\log^2 n)$ time using $O(n)$ work.



i	0	1	2	3	4	5	6
$S[i]$	b	a	n	a	n	a	\$
$SA[i]$	6	5	3	1	0	4	2
$S[SA[i]-1]$	a	n	n	b	\$	a	a

Figure 6: Suffix tree and suffix array (SA) for the string S = "banana\$".

3.1.2 Move-to-Front (MTF) Coding

Let $S^{BST}_{i,j}$, $0 \leq i \leq j \leq n$ be the substring $[S^{BST}_i, ..., S^{BST}_{j-1}]$; $S^{BST}_{i,j}$ is defined to be the null string when $i = j$. Let $\sigma_{i,j}$ be the set of characters contained within $S^{BST}_{i,j}$ and $M_{i,j}$ be the listing of the characters in $\sigma_{i,j}$ in order of last occurrence in $S^{BST}_{i,j}$ (i.e., in MRU order); this is the empty list when $i = j$. Denote by $x \oplus y$ the list formed by concatenating to the end of y the list formed by removing from x all elements that are contained in y. The key idea behind the PRAM algorithm for MTF coding is the observation, noted in the discussion of the serial MTF algorithm, that L_i is the MRU listing of the characters of $S^{BST}_{0,i}$ followed by the remaining characters of Σ in their originally defined order. That is, $L_i = L_0 \oplus M_{0,i}$.

Observe that $M_{i,j} = M_{i,k} \oplus M_{k+1,j}$ for all k, $i \leq k < j$. This implies that $M_{i,j} = \oplus^{j-1}_{k=i} M_{k,k+1}$. By definition, $M_{k,k+1}$ is simply the list $[S^{BST}_k]$. Furthermore, $\oplus$ is associative, and by the assumption that $|\Sigma|$ is constant, takes $O(1)$ time and work to compute. Therefore, $M_{0,i}$, and thus L_i, for $0 \leq i < n$ can be computed in $O(\log n)$ time using $O(n)$ work by the standard PRAM algorithm for computing all prefix-sums with respect to the operation $\oplus$. The prefix sums algorithm works in two phases:

1. Adjacent pairs of MTF lists are combined using $\oplus$ in a balanced binary tree approach until only one list remains (see Figure 7).
2. Working back down the tree, the prefix sums corresponding to the rightmost leaves of each subtree are computed using the lists computed in phase 1 (see Figure 8).

Given L_i, S^{MTF}_i is simply the index in L_i of S^{BST}_i, which

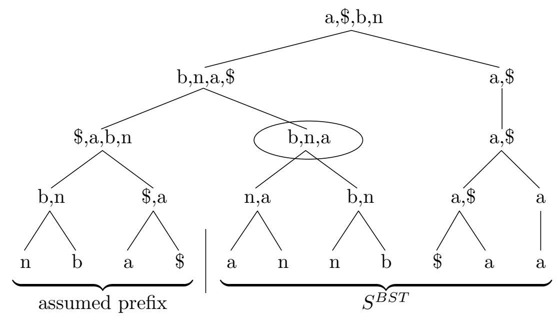


Figure 7: Phase 1 of prefix sums: Computing local MTF lists for "annb\$aa" using the operator $\oplus$. Each node in the tree is the $\oplus$-sum of its children. For example, the circled node is (n, a) $\oplus$ (b, n).

can be found for all characters independently in $O(1)$ time and $O(n)$ work. Therefore, MTF coding can be performed in $O(\log n)$ time using $O(n)$ work.

3.1.3 Huffman Coding

The PRAM algorithm for Huffman coding follows readily from the serial algorithm. In step 1, F is constructed using the integer sorting algorithm outlined in [7], which sorts a list of n integers in the range $[0, r-1]$ in $O(r + \log n)$ time using $O(n)$ work. Because $r = |\Sigma|$ is constant, step 1 runs in $O(\log n)$ time and $O(n)$ work. Step 2 of the serial algorithm runs in $O(1)$ serial time, so the same algorithm can be used to construct T from F in $O(1)$ time and work. Step 3 is performed in as follows. First, the prefix-sums of the code lengths $|T(S^{MTF}_i)|$ are computed into the array U in $O(\log n)$ time and $O(n)$ work. Then, in parallel for all i, $0 \leq i < n$, $T(S^{MTF}_i)$ is written to S^{BW} starting at position U_i in $O(1)$ time using $O(n)$ work. Therefore, the overall Huffman coding stage runs in $O(\log n)$ time using $O(n)$ work.

The above discussion proves the following theorem:

THEOREM 1. *The above algorithm solves the Burrows-Wheeler Compression problem in $O(\log^2 n)$ time using $O(n)$ work.*

3.2 Decompression

3.2.1 Huffman Decoding

The main obstacle to decoding S^{BW} in parallel is that, because Huffman codes are variable-length codes, we do not know where the boundaries between codewords in S^{BW} lie. We cannot simply begin decoding from any position, as the result will be incorrect if we begin decoding in the middle of a codeword. Thus, we must first identify a set of valid starting positions for decoding. Then, we can trivially decode the substrings of S^{BW} corresponding to those starting positions in parallel.

Our algorithm for locating valid starting positions for Huffman decoding is as follows. Let l be the length of the longest codeword in T, the Huffman table used to produce S^{BW}; l is constant because $|\Sigma|$ is. Without loss of generality, we assume that $|S^{BW}|$ is divisible by l. Divide S^{BW} into partitions of size l. Our goal is to identify one bit in each

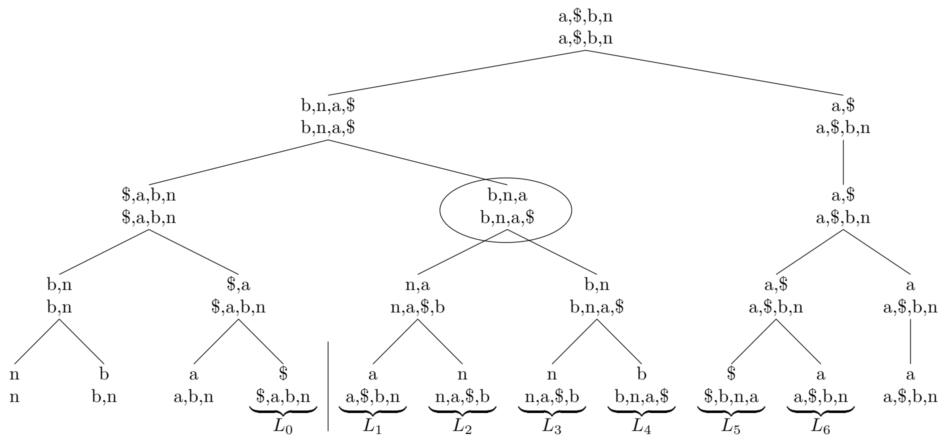


Figure 8: Computing the prefix sums of the output of the BST stage, "annb$aa", with respect to the associative binary operator $\oplus$. The top line of each node is copied from the tree in Figure 7. The bottom line of a node V is the cumulative $\oplus$-sum of the leaf nodes starting at the leftmost leaf in the entire tree and ending at the rightmost child of V (i.e., the prefix sum up to the rightmost leaf under V). For example, the circled node contains the sum of leaves corresponding to the prefix "nba$annb". Observe the correspondence of the labeled lists with Figure 4.

partition as a valid starting position. The computation will proceed in two steps: (1) initialization and (2) prefix sums computation.

For the initialization stage, we consider every bit i, $0 \leq i < |S^{BW}|$, in S^{BW} as if it were the first bit in a string to be decoded, henceforth S_i^{BW}. In parallel for all i, we decode S_i^{BW} (using the standard serial algorithm) until we cross a partition boundary, at which point we record a pointer from bit i to the stopping point. Now, every bit i has a pointer $i \to j$ to a bit j in the immediately following partition, and if i happens to be a valid starting position, then so is j. See Figure 9(a).

For the prefix sums stage, we define the associative binary operator $\oplus$ to be the merging of adjacent pointers (that is, $\oplus$ merges $A \to B$ and $B \to C$ to produce $A \to C$). See Figure 9(b). The result is that there are now pointers from each bit in the *first* partition to a bit in every other partition. Finally, we identify all bits with pointers from bit 0 as valid starting positions for Huffman decoding (see Figure 9(c)); we refer to this set of positions as V. All this takes $O(\log n)$ time and $O(n)$ work.

The actual decoding is straightforward and proceeds as follows.

1. Employ $|S^{BW}|/l$ (which is $O(n)$) processors, assign each one a different starting position from the set V, and have each processor run the serial Huffman decoding algorithm until it reaches another position in V in order to find the number of decoded characters. Do not actually write the decoded output to memory yet. This takes $O(1)$ time because the partitions are of size $O(1)$.
2. Use prefix sums to allocate space in S^{MTF} for the output of each processor. ($O(\log n)$ time, $O(n)$ work)
3. Repeat step (1) to actually write the output to S^{MTF}. ($O(1)$ time, $O(n)$ work)

These three steps, and thus the entire Huffman decoding algorithm, take $O(\log n)$ time and $O(n)$ work.

3.2.2 Move-to-Front (MTF) Decoding

The parallel MTF decoding algorithm is similar to the parallel MTF encoding algorithm but uses a different operator for the prefix sums step. In contrast to MTF encoding, MTF decoding uses the characters of S^{MTF} directly as indices into the MTF list. Therefore, S_i^{MTF} defines a fixed permutation function that maps L_i to L_{i+1}. Denote by $P_{i,j}$ the permutation mapping L_i to L_j. Then, $P_{0,j}$ can be computed for all j, $0 \leq j < n$, using prefix sums with function composition as the associative operator. See Figure 10. A permutation function for a list of constant size can be represented by another list of constant size, so composing two permutation functions takes $O(1)$ time and work. Therefore, the prefix sums, and the overall MTF decoding algorithm, take $O(\log n)$ time and $O(n)$ work.

3.2.3 Inverse Block-Sorting Transform (IBST)

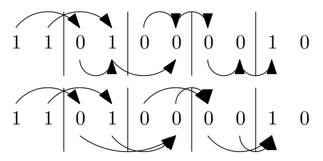
The parallel IBST algorithm proceeds in two steps, analogous to the serial algorithm. In step 1, the integer sorting algorithm of [7] is used to sort the characters of S^{BST}. Because $|\Sigma|$ is constant, the characters have a constant range, and so this step takes $O(\log n)$ time and $O(n)$ work. In step 2, and the list ranking algorithm of [8] is used to rank the linked list in $O(\log n)$ time and $O(n)$ work. Finally, the characters of S^{BWT} are written to S according to their rank in the linked list; this takes $O(1)$ time and $O(n)$ work. Overall, the IBST takes $O(\log n)$ time and $O(n)$ work.

The above discussion proves the following theorem:

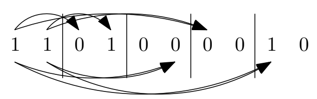
THEOREM 2. *The above algorithm solves the Burrows-Wheeler Decompression problem in $O(\log n)$ time using $O(n)$ work.*

4. EXPERIMENTAL VALIDATION

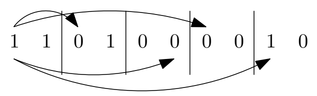
4.1 The XMT Platform



(a) Step 1: initialization.



(b) Step 2: prefix sums.



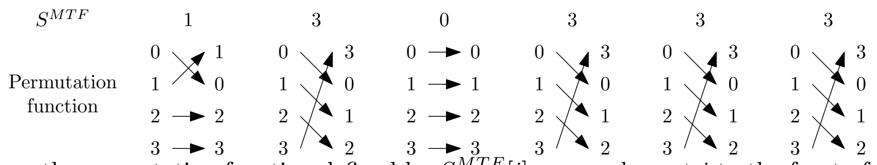
S^{BW}	$\underline{1}1$	$\underline{0}$ 10	$\underline{0}$	$\underline{0}$ 0	$\underline{1}0$
S^{MTF}	1	3, 0	3	3, 3	0

(c) Pointers from bit 0, corresponding to valid staring positions in S^{BW} (underlined).

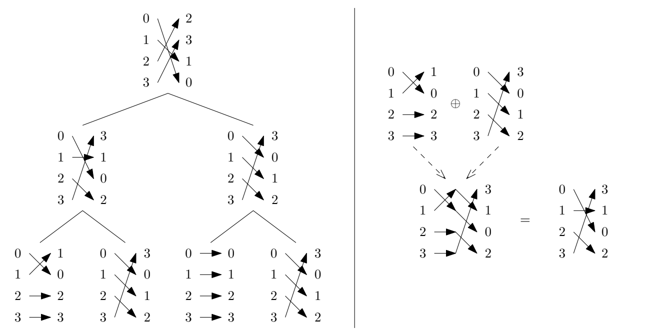
Figure 9: Huffman decoding of S^{BW} (from Figure 5).

1 3 0 3 3 3 0

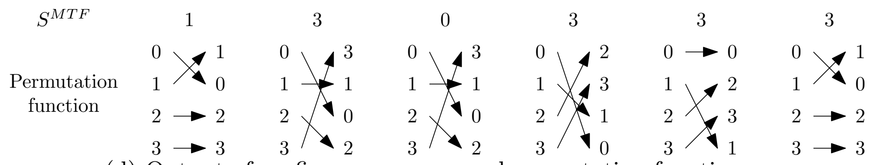
(a) S^{MTF} (from Figure 9).



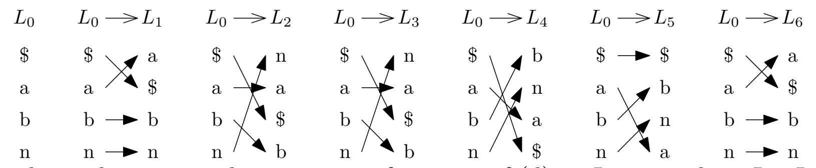
(b) Initialization: the permutation function defined by $S^{MTF}[i]$ moves element i to the front of its input list.



(c) (Left) Prefix sums: composition of permutation functions using a balanced binary tree (here, we show the tree for the first four elements).
(Right) Computing the $\oplus$-sum of the leftmost two leaves of the tree. The result is the parent of the two leaves.



(d) Output of prefix sums: composed permutation functions.



(e) Applying the composed permutation functions of (d) to L_0 to produce L_1, L_2, etc.

Figure 10: MTF decoding of S^{MTF} from Figure 9: construction of L_i in parallel using composed permutation functions. The last character of S^{MTF} is not used in this construction because the corresponding list L_7 is not needed. Observe the correspondence of the labeled lists in (e) with Figure 4.

The Explicit Multi-Threading (XMT) general-purpose computer architecture is designed to improve single-task completion time. It does so by supporting programs based on Parallel Random-Access Machine (PRAM) algorithms but relaxing the synchrony required by the PRAM model. The XMT programming model differs from the strict PRAM model in two ways:

1. The PRAM model requires specifying the instruction that will be executed by each processor at each point in time, but XMT uses the work-depth methodology ([33, 35]), which allows the programmer to specify all of the operations that can be performed at each point in time while leaving to the runtime environment the assignment of those operations to processors.
2. The PRAM model requires instructions to be executed in lockstep by all processors at once, but XMT programs follow independence-of-order semantics: parallel sections of code are delimited by spawn-join instruction pairs, and threads only synchronize when they reach the join instruction at the end of the parallel section.

The XMT architecture consists of the following: a number of lightweight cores (TCUs) grouped into clusters, a single core (master TCU or MTCU) with its own local cache, a number of mutually-exclusive cache modules shared by the TCUs and MTCU, an interconnection network connecting the TCUs to the cache modules, and a number of DRAM controllers connecting the cache modules to off-chip memory. Each TCU has a register file, a program counter, an execution pipeline, and a lightweight ALU. Each TCU also contains prefetch buffers, which can be used by the compiler to prefetch data from memory before it is needed, reducing the length of the sequence of round trips to memory (LSRTM) and improving performance [36]. Each cluster has one or more multiply/divide units (MDUs), floating-point units (FPUs), and a compiler-managed read-only cache, all of which are shared by the TCUs within the cluster. When a parallel section of code is reached, the MTCU broadcasts the instructions in that section to all of the TCUs, and each TCU stores the instructions in a buffer. Virtual threads are assigned to TCUs using a dedicated prefix-sum network.

An overview of XMT with details relevant to work on application can be found in [6].

4.2 Evaluated Configurations

Because XMT is an experimental platform, we establish that XMT is competitive with single-chip multi-cores and many-cores currently available on the market by choosing a configuration of XMT that would use resources comparable to one such commercially-available chip. The most recent comparison of XMT with existing chips is [6], in which a 1024-TCU configuration of XMT with 4 MB[3] shared cache (herein called XMT-1024) is shown to use a comparable silicon area to the NVIDIA GTX 280 GPGPU, which uses 576 mm^2 of silicon in 65 nm technology. In [21], XMT-1024 was shown to remain in the same power envelope as the GTX 280 as well. Since then, silicon technology has improved, and the current successor to the GTX 280, the GTX 680, is manufactured in 28 nm technology, with a die size of 294 mm^2, just over half that of the GTX 280; however, recall that nominally 294 mm^2 in 28 nm technology offers more than twice the device capacity of 576 mm^2 in 65 nm technology.

[3]1 MB = 2^{20} bytes

File	Description	Size (bytes)
bible.txt	The King James version of the bible	4,047,392
E.coli	Complete genome of the E. Coli bacterium	4,638,690
world192.txt	The CIA world fact book	2,473,400

Table 1: Files in the Large Corpus

We compare our parallel implementation of Burrows-Wheeler[4] compression running on a 64-TCU FPGA prototype of XMT [37], and also on XMT-1024, against bzip2 running on one core of the Intel Core i5-2500K CPU with 6 MB of L3 cache. To obtain results for the XMT-1024 configuration, we used XMTSim, the cycle-accurate simulator of the XMT architecture. XMTSim and the XMTC compiler are described in [22] and have already been the basis for several publications including [6]. The most recent validation of the cycle-accuracy of the simulator is [20], which shows that the simulator cycle counts match those of the FPGA except in a minority of cases, where the discrepancy may be up to 33%, due in part to interconnect and DRAM technology limitations in the FPGA prototype that would not exist in an ASIC product. For BW compression, the difference due to these limitations is 15%.

4.3 Data Sets

We perform our comparison using the Large Corpus from the Canterbury Corpus [29], a standard set of files used to evaluate compression algorithms. We use a block size of 900,000 bytes for both bzip2 and our parallel implementation, and we obtain speedup results for each block separately since both implementations compress one block at a time. We use the notation *file.i* to denote block i of a file named *file*. Because the file sizes are not evenly divisible by the block size, the last block of each file is smaller than 900,000 bytes, and such blocks are denoted using parentheses. For comparison purposes with bzip2 implementations, our experimental results are reported with respect to blocks. It should also be noted that our fine-grained approach is orthogonal to existing coarse-grained ones allowing one to benefit from both in a single implementation.

4.4 Implementation Details

4.4.1 Prefix Sums

We use a k-ary tree to implement prefix sums operations. To improve the performance of these operations, we cluster threads in the spawn block immediately preceding each prefix sums operation into groups of size c and merge them with the first iteration of the prefix sums operation. Similarly, we merge the last iteration with the following spawn block. In our code, we fixed c at 256, corresponding to the 8-bit character set used by bzip2.

4.4.2 Run-Length Encoding (RLE)

[4]Available at `http://www.umiacs.umd.edu/users/vishkin/XMT/OPEN_SOURCE_ALGS/`

To reduce the size of its output, bzip2 adds two run-length encoding (RLE) stages to the basic BW compression algorithm. We added these stages to our implementation as well. Since this enhancement is not part of the core compression algorithm and thus not covered in the theoretical portion of this paper, we state without proof that the RLE algorithm we implemented runs in $O(\log n)$ time and $O(n)$ work.

4.4.3 Multiple Huffman Tables

Bzip2 also implements a heuristic that switches among multiple Huffman tables to possibly reduce the size of its output. We are not aware of a parallel algorithm that can decode data encoded using this heuristic within the bounds given by Theorem 2. To enable a fair comparison with our implementation, we modified bzip2 to only use a single Huffman table. For the inputs we used in our comparison, this caused the average size of the compressed output to increase by 2.75% relative to that of the unmodified bzip2. On average, the modified bzip2 compression ran 1.5% faster than the unmodified version, and the decompression ran 7.5% faster.

4.4.4 Block-Sorting Transform (BST)

To provide better practical performance for small inputs, we use a randomized, recursive variant of shared-memory sample sort to compute the BST. Although a serial recursive sample sort algorithm is described and analyzed in [4], there appears to be no prior polylogarithmic-time PRAM analog of such an algorithm. We describe the sorting algorithm below; after the rotations of S are sorted, S^{BST} can be derived in $O(1)$ time using n processors by having each processor i, $0 \leq i < n$, output the last character of string i in the list of sorted strings.

The initial list of the n rotations of S is passed to a procedure called $SAMPLESORT$. Let T_c be the time, and W_c be the work, required to compare two strings. Given a list L as input, $SAMPLESORT(L)$ proceeds in five steps. (1) A subset of n/k splitters, with $k > 1$ a constant, is randomly selected from L and placed in the list L'. (2) If L' contains more than one element, $SAMPLESORT(L')$ is called recursively. (3) Each element of L is ranked within L' using binary search. (4) The elements of L are partitioned according to their rank in L'. Because the ranks are integers in the range $[0, n-1]$, this partitioning can be done using, e.g., the integer sorting algorithm of [16], which runs in $O(\log n)$ time and $O(n\sqrt{\log n})$ work. (5) The partitions are sorted in parallel, with a serial comparison-based sort applied to each partition.

5. RESULTS

5.1 Comparison with bzip2

Speedups for the 64-TCU FPGA prototype are shown in Figure 11 and are in the range 1.8-2.8x for compression and 0.8-1.1x for decompression. Speedups for the simulated XMT-1024 configuration are shown in Figure 12 and are in the range 12x-25x for compression and 11x-13x for decompression. The main reason that speedups for 64 TCUs are low, but then scale up nicely for 1024 TCUs, is the extra work that our parallel algorithms do beyond the original serial algorithm.

On the FPGA, speedups for partial blocks are higher than

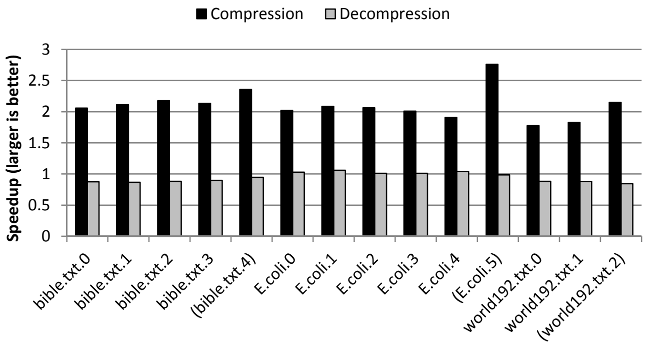


Figure 11: Speedups obtained using the 64-TCU FPGA prototype. Partial blocks at the ends of files are indicated with parentheses.

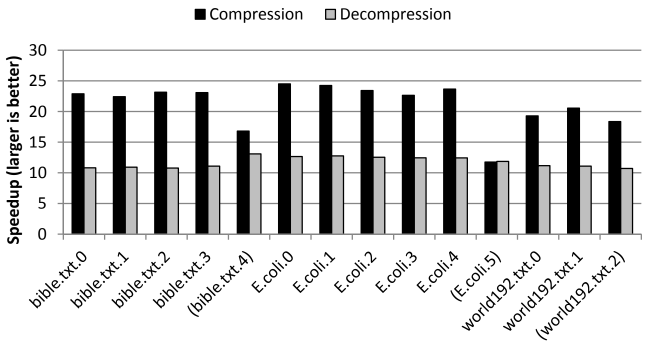


Figure 12: Speedups obtained using the XMT-1024 configuration. Partial blocks at the ends of files are indicated with parentheses.

for the preceding full blocks in the same file. This is because the partial blocks fit better than full blocks into the limited cache size (256 KB[5]) of the FPGA. The situation is reversed for XMT-1024, where the partial blocks bible.txt.4 and E.coli.5 exhibit lower speedups than full blocks in the same file. This is because we tuned our code to provide optimal performance on 900 KB blocks. For smaller inputs, performance can be improved by tuning the code to spread the work among a larger number of threads, decreasing granularity (at the cost of higher overhead). For example, decreasing the factor k in $SAMPLESORT$ and the clustering factor c provides up to 1.3x higher speedup for partial blocks.

Of all the stages in the parallel implementation, the BST in the compression routine is the most time consuming, and the corresponding inverse BST (IBST) in the decompression routine is the second most time-consuming step. This is equally true for bzip2 compression; it may be true for bzip2 decompression as well, but the stages in the bzip2 decompressor are interleaved, so we could not separate out IBST. Therefore, improving the performance of these stages has the greatest effect on overall runtime.

The aforementioned BST and IBST stages have irregular parallelism and memory access patterns. In addition, all of the stages in the compression and decompression routines employ fine-grained parallelism. In contrast to many parallel computing platforms, which have difficulty running such

[5]1 KB = 2^{10} bytes

algorithms efficiently, the XMT platform is designed with such algorithms in mind. This is perhaps one reason that others have overlooked a parallel-algorithmic approach.

In addition to allowing parallelism to be exploited within a block, our approach has the advantage that it only requires working space for a single block, as blocks are processed one at a time. Therefore, all working data fits in cache, and DRAM is only accessed to read input blocks and write output blocks. In contrast, if we were to compress multiple blocks simultaneously using a single XMT chip, we would only be able to process a few blocks in parallel without spilling working data to DRAM. Therefore, our approach may have more efficient cache utilization than block-parallel approaches.

The current embodiments of the XMT platform have the limitation that memory can only be addressed in terms of 32-bit words; threads cannot write to individual bytes without overwriting all bytes within a word. Therefore, if multiple threads need to be able to write to arbitrary elements of an array, the elements of that array must be stored as 32-bit words even if they could otherwise be stored as single bytes. Commercial-grade platforms, such as the Intel processor we compare against, do not have this limitation. This means that our results are conservative relative to a more complete version of XMT with this restriction removed.

For XMT-1024, our parallel decompression implementation performs better on E.coli.5 than on any other input. This is because it is smaller than every other input (by at least a factor of 3), and thus the working set fits into cache for this input alone. To verify this, we tested a variant of XMT-1024 with 16 MB of cache and found that the minimum speedup increased from 10x to 12x. This suggests that it may be worthwhile to take advantage of improvements in silicon technology noted earlier to increase the size of the shared cache of XMT.

5.2 Using Compression to Increase Bandwidth

We compare our implementation (henceforth xmt-bw) to a number of other compression libraries by providing as input the entire (11 MB) Large corpus[6] and measuring the compression ratio and speed; except for xmt-bw and pbzip2, all libraries are serial. Figure 13 shows our results. Each library has two regions. (1) As long as the effective bandwidth does not exceed the maximum compression speed, the effective bandwidth is limited only by the compression ratio (sloped portion, bandwidth-limited). (2) Once the maximum compression speed is reached, no further increase is possible (horizontal portion, compute-limited).

We compare xmt-bw against two classes of compression libraries:

- High compression, low speed: zlib [13], bzip2 [31], pbzip2[7] [14], and xz [9].
- Low compression, high speed: Snappy [15], LZO [27], QuickLZ [1], liblzf [25], and FastLZ [18].

[6]All of the implementations tested here (including our own) subsequently divide the input into blocks.
[7]4 cores

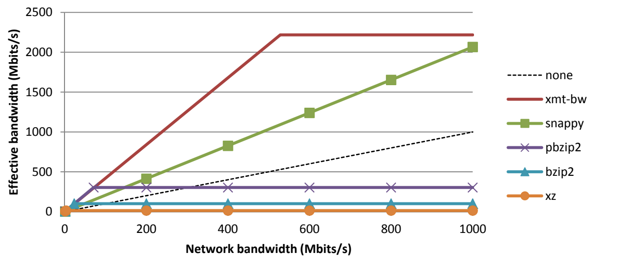


Figure 13: Comparison of maximum transfer rates over a bandwidth-limited network using BW compression on XMT (xmt-bw) and existing compression libraries. The dotted diagonal line represents the baseline of no compression.

Of these, QuickLZ, liblzf and FastLZ are dominated by Snappy, and zlib is dominated by pbzip2. Snappy outperforms LZO up to 1.46 Gbits/s, beyond which LZO provides 6% more effective bandwidth. Results for the remaining libraries are shown in Figure 13.

For network bandwidths up to 3 Mbits/s, xz outperforms xmt-bw by 4% due to its slightly higher compression ratio. Beyond that, for network bandwidths up to 1 Gbit/s, xmt-bw is dominant; it is only outperformed by Snappy, LZO, and QuickLZ at higher bandwidths. Remarkably, this breakpoint coincides with the peak bandwidth of Gigabit Ethernet, which is commonly used on commodity systems. Even on networks with a higher peak bandwidth, the point-to-point bandwidth depends on network load and may fall into the range where xmt-bw provides an advantage. Finally, beyond 3.1 Gbits/s, it is more efficient to transmit data uncompressed.

6. CONCLUSION

This paper is the first to demonstrate work-optimal algorithmic and empirical feasibility of parallel compression which compromises neither speed nor compression quality. For small inputs, it provides speedups where no other approach does. For transmission of data over a network, it provides a larger increase in effective bandwidth than other approaches over a wide range of network bandwidths.

Today's parallel architectures allow good speedups on regular dense-matrix type programs, but are basically unable to match this success outside this, including for: 1. irregular problems/programs; and, 2. strong scaling. Extending parallel hardware to address these domains could potentially lead to phenomenal growth in supercomputing: 1. Nearly all serial algorithms in the CS curriculum are irregular; how many more programmers and applications will migrate to parallel computing if such parallel algorithms will deliver good speedups? 2. Publication of slow-down results, as in [28], are extremely rare and reflect an unusual level of interest in a problem. How many more will become interested if commercial hardware allows speedups?

7. REFERENCES

[1] QuickLZ: Fast compression library for C, C# and Java. `http://www.quicklz.com/index.php`.
[2] J. Abel. Improvements to the Burrows-Wheeler compression algorithm: After BWT stages. 2003.

[3] J. Abel. Post BWT stages of the Burrows-Wheeler compression algorithm. *Software: Practice and Experience*, 40(9):751–777, 2010.

[4] P. M. G. Apers. Recursive samplesort. *BIT Numerical Mathematics*, 18:125–132, 1978.

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

[6] G. C. Caragea, F. Keceli, A. Tzannes, and U. Vishkin. General-purpose vs. GPU: Comparison of many-cores on irregular workloads. In *HotPar '10: Proceedings of the 2nd Workshop on Hot Topics in Parallelism*. USENIX, June 2010.

[7] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Inf. Control*, 70(1):32–53, July 1986.

[8] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81(3):334 – 352, 1989.

[9] L. Collin. XZ utils. `http://tukaani.org/xz/`.

[10] J. A. Edwards and U. Vishkin. Brief announcement: Truly parallel Burrows-Wheeler compression and decompression. In *SPAA 2013,* to appear.

[11] J. A. Edwards and U. Vishkin. Parallel algorithms for Burrows-Wheeler compression and decompression. Technical report, University of Maryland, College Park, MD, November 12, 2012. `http://hdl.handle.net/1903/13299`.

[12] A. Eirola. Lossless data compression on GPGPU architectures. *ArXiv e-prints*, 2011.

[13] J.-L. Gailly and M. Adler. zlib compression library. `http://www.dspace.cam.ac.uk/handle/1810/3486`.

[14] J. Gilchrist and A. Cuhadar. Parallel lossless data compression based on the Burrows-Wheeler transform. In *Proc. Advanced Information Networking and Applications*, pages 877 –884, May 2007.

[15] Google, Inc. Snappy, a fast compressor/decompressor. `http://code.google.com/p/snappy/`, Released March 2011.

[16] Y. Han and X. Shen. Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs. *SIAM Journal on Computing*, 31(6):1852–1878, 2002.

[17] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, fifth edition, 2011.

[18] A. Hidayat. Fastlz - lightning-fast compression library. `http://fastlz.org/`.

[19] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, Nov. 2006.

[20] F. Keceli. *Power and Performance Studies of the Explicit Multi-Threading (XMT) Architecture*. PhD thesis, University of Maryland, 2011. Chapter 4. `http://hdl.handle.net/1903/11926`.

[21] F. Keceli, T. Moreshet, and U. Vishkin. Power-performance comparison of single-task driven many-cores. In *Proc. ICPADS*, 2011.

[22] F. Keceli, A. Tzannes, G. Caragea, R. Barua, and U. Vishkin. Toolchain for programming, simulating and studying the XMT many-core architecture. In *Proc. IPDPSW*, pages 1282–1291, 2011.

[23] S. T. Klein and Y. Wiseman. Parallel Huffman decoding with applications to JPEG files. *The Computer Journal*, 46(5):487–497, 2003.

[24] S. T. Lavavej. bwtzip: A linear-time portable research-grade universal data compressor. `http://nuwen.net/bwtzip.html`.

[25] M. Lehmann. liblzf. `http://software.schmorp.de/pkg/liblzf.html`.

[26] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *Proc. Data Compression Conference*, pages 193–202. IEEE, 2009.

[27] M. F. Oberhumer. LZO real-time data compression library. `http://www.oberhumer.com/opensource/lzo/`.

[28] R. Patel, Y. Zhang, J. Mak, A. Davidson, and J. Owens. Parallel lossless data compression on the gpu. In *Innovative Parallel Computing (InPar), 2012*, pages 1–9, 2012.

[29] M. Powell. The Canterbury corpus. `http://corpus.canterbury.ac.nz/`, 2001.

[30] S. C. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, STOC '94, pages 300–309, New York, NY, USA, 1994. ACM.

[31] J. Seward. bzip2, a program and library for data compression. `http://www.bzip.org/`.

[32] J. Seward. On the performance of BWT sorting algorithms. In *Data Compression Conference, 2000. Proceedings. DCC 2000*, pages 173 –182, 2000.

[33] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms*, 3(2):128–146, February 1982.

[34] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, pages 12–20, 1984.

[35] U. Vishkin. Using simple abstraction to reinvent computing for parallelism. *Communications of the ACM (CACM)*, 54:75–85, Jan. 2011.

[36] U. Vishkin, G. Caragea, and B. Lee. Models for advancing PRAM and other algorithms into parallel programs for a PRAM-on-chip platform. In S. Rajasekaran and J. Reif, editors, *Handbook on Parallel Computing: Models, Algorithms, and Applications*, chapter 5. Chapman and Hall/CRC Press, 2008.

[37] X. Wen and U. Vishkin. FPGA-based prototype of a PRAM-on-chip processor. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 55–66, New York, NY, USA, 2008. ACM.

[38] Y. Wiseman. Burrows-wheeler based JPEG. *Data Science Journal*, 6, 2007.