Parallel File Compression

Jacob Kitzman and Guilherme Issao Fujiwara

May 12, 2005

Abstract

Suffix Sorting is a common, but computationally intensive algorithmic problem with applications in bioinformatics, searching, and data compression. Here we present a parallel Suffix Sorting implementation and apply it to improve the performance of a common class of data compression programs.

1 Introduction

Suffix Arrays were introduced by Myers and Manber [1] to solve large text-searching problems in bioinformatics. Their construction is equivalent to the problem of Suffix Sorting, a computationally-intensive algorithm with many applications. Many incremental improvements to the time and space requirements of this algorithm have recently been reported in the literature, enabling new applications from bioinformatic sequence analysis to file compression. Even so, further improvements are needed to keep pace with the burgeoning sizes of today's inputs such as multimedia content and genomic sequence. Paralleized implementations such as that described by Futamura and colleages [3] can bring the increasingly ubiquitous power of cluster computing to bear on Suffix Array-based problems.

Here we present a implementation of Futumura's parallel Suffix Sorting approach and apply it to the problem of data compression. The application to data compression is based on the Burrows-Wheeler Transform, a permutation transform on strings. It has an inverse transform that is simple to compute and it has the property that equal characters tend to appear together, making it is useful for preprocessing data to be compressed, increasing the compressing ratio.

File Compressors that use this transform (e.g. bzip2) show very good compression rates but are more time-consuming than other common compression methods like gzip and zip. We hypothesized that a BWT-based compressor like bzip2 would be an excellent candidate for parallelization.

We implement a Parallel file compression software based on `bwtzip` [4], a research-grade file compressor that uses the Burrows-Wheeler Transform. We implemented our own Parallel Burrows-Wheeler Trasnform and intergrated it with `bwtzip`.

We obtained timing data points for our File Compressor and we observed very good scalability in most inputs.

2 Suffix Sorting

Given a string $a_1a_2\ldots a_n$, we define $s_i = a_ia_{i+1}\ldots a_n$. Then the problem of Suffix Sorting is to lexicographically sort all s_i for $i = 1, \ldots, n$. This yields the Suffix Array [1], a permutation of $1, \ldots, n$: $SA_1, \ldots, SA_n$ such that $SA_i = j$ iff s_i is the j-th lexicographically smallest suffix of s.

A naive approach might be to use a standard sorting procedure such as `quicksort` or `radixsort`. Running a comparison-based sort such as `quicksort` on $O(n)$ suffices, each of length $O(n)$, will grow in runtime as $O(n^2 * \log(n))$. Similarly, the runtime of `radixsort` on n suffices grows as $O(n^2)$ time. Instead, many suffix sorting algorithms take advantage of the underlying structure common to a set of suffices.

Applying the methods of Weiner [5] and Ukkonen [6], it is possible to construct a suffix tree, and from that tree derive the sorted suffix array by a lexicographic preorder traversal in both linear time and space. Gusfield [7] gives a very thorough treatment of this topic.

2.1 Parallelizing Suffix Sort

There are many algorithms for Suffix Sorting that make use of the fact that s_i is a suffix of a string and not just any string and therefore perform better than general sorting algorithms.

In particular, there is a parallel algorithm [3] performing to do just that, making use of the fact that s_i is a suffix to distribute in linear time consecutive suffices to each processor to sort in parallel, so that after that we just need to concatenate the results of each processor.

This distribution works as follows: Let n be the size of the string, s be the size of our alphabet and p be the number of processors. We choose a window-size w and we separate the suffices into buckets according to its first w characters. At first, this seems to take $O(ws^w)$. However, we make use of the fact that we are working with the suffices of a string: we index the buckets with integer, ordering them in lexicographic order. Then we compute the index f_1 of the bucket for the first suffix. To compute the index of the bucket for the next, we use that $f_2 = (f_1 * s + a_{w+1}) (\text{mod } s^w)$, which can be computed in linear time. We proceed this way using $f_{k+1} = (f_k * s + a_{k+w}) (\text{mod } s^w)$ to compute the index of the bucket of each suffix in linear time. Moreover, we do this computation in parallel, assigning the distribution of the suffices $a_{\frac{nk}{p}}$ through $a_{\frac{n(k+1)}{p}-1}$ into buckets to processor k, for all $k = 0, 1, \ldots p - 1$.

Now we need to distribute the buckets among the nodes. To do so, we compute in each processor how many suffices are there in each bucket. Then we use the MPI `Allreduce` routine to compute the total number of suffices in each bucket.

Then we compute the cummulative sum of the vector of suffices per bucket and we do our cuts for load balancing in the entries closest to $\frac{n*k}{p}$ for $k = 1, 2, \ldots p-1$.

The last step of the algorithm is to (in serial) sort the suffices in each bucket. Because we are sorting a subset of the suffices, we cannot apply linear-time suffix-sorting algorithms. Rather, we adapted and used the ternary search tree-based general string sorting algorithm described by Bentley and Sedgewick [2].

3 Burrows-Wheeler Transform

One interesting application of Suffix Sorting is in computing the Burrows-Wheeler Trasform: Given a string $w = w_1 w_2 \ldots w_n$, we define $c_i = w_i w_{i+1} \ldots w_n w_1 \ldots w_{i-1}$ the i-th circular shift of the original string w. The Burrows-Wheeler Transform of the string w is given by reading the last character of each c_i in lexicographical order. For example, if we are given $w =$ `parallel`, then:

c_1	`parallel`
c_2	`arallelp`
c_3	`rallelpa`
c_4	`allelpar`
c_5	`llelpara`
c_6	`lelparal`
c_7	`elparall`
c_8	`lparalle`

And after sorting these circular shifts we obtain:

c_4	`allelpar`
c_2	`arallelp`
c_7	`elparall`
c_6	`lelparal`
c_5	`llelpara`
c_8	`lparalle`
c_1	`parallel`
c_3	`rallelpa`

And therefore the Burrow-Wheeler Transform is `rpllaela`, reading the last column.

3.1 Why is this useful for Compression?

This transform has two useful properties from a compression point of view. First, its inverse is computable and there are algorithms that do it in a small time compared to the forward transform. For a description of an algorithm for the inverse transform, see [8].

The other useful property is that this Transform tends to gather groups of a single character (in case the original string has recurring patterns, such as words). The presence of continous sequences of a repeted character in the original sequence is good for most compression algorithms, generating good compression rates. In particular, using a Huffman code [9] in the transformed string results in very good compression rates in most common data.

Many Compression Algorithms use this transform, in particular the famous open source `bzip2` and the proprietary formats `RAR` and `ACE`.

3.2 BWT can be Found by Suffix Sorting

It turns out that BWT is reduced to the problem of Suffix Sorting for the class of strings having the sentinel character at the end. The sentinel is simply a character that does not occur anywhere else in the string; it is often denoted $.

Then soring the suffices is equivalent to sort the shifts because any two circular shifts will differ in a character that appears before any of the strings wrap around (i.e., before the character w_1), so comparing these two circular shifts is equivalent to compare the corresponding suffices that start at the same position.

In order to always use sentineled for BWT, we add a symbol to our alphabet and always insert this symbol in the end of our original string before computing this transform, remembering to remove it when we compute the inverse.

4 Implementation of Parallel Suffix Sorting

We provide an implementation in MPI/C++ of an adaptation of the parallel suffix sorting algorithm described by Futamura and colleagues [3]. Although they mention their own implementation and indeed cite performance experiments, we were unable to find either their code or any other parallelized suffix sorting implementation on the Internet.

The implementation and headers for our parallel suffix sorting are contained, respectively, in the files `ss_par.cc` and `ss_par.hh`. The main sorting routine, `ss_par_main` accepts the following inputs:

- `vBuf`, a string in the form of an STL `vector` of type `short` elements,
- `w`, the desired window-size,
- `a`, the size of the alphabet such that for every element s_i in `wBuf`, $s_i <$ `w`
- `_myRank`, the rank of the processor on which the code is running,
- `_np`, the total number of processors over which the code is running, and
- `ppvSortedSufIdxs`, a vector of indices into which the output suffix array is stored.

Our adaptation of the (sequential) Bentley-Sedgewick quicksort is contained in the files `ser_suf_sort.cc` and `ser_suf_sort.hh`. These files are minor adaptations of the publicly available implementions provided by Robert Sedgewick at http://www.cs.princeton.edu/ rs/strings/. We modified their implementation to take only one input string but also take an array of indices into that string corresponding to the subset of suffices to be sorted.

5 Integration with BWTZIP and Structure of code

BWTZIP is a compression software developed by Stephan T. Lavavej and Joergen Ibsen that uses the Burrows-Wheeler Transform and a adaptive Huffman encoder [9].

Their code is very modular and implements many different algorithms. The author describe it as research-based, meaning it is very good for experimenting different algorithms, but not very optimized. This fits very well with our purpose, so we based ourselves on their code, writing a parallel extension.

The BWTZIP package contains implementations of a few different algorithms for the BWT transform. We add two more, a parallel version (`bwtparallel`) implemented with MPI that uses the Suffix Sorting algorithm by Futamura [3] describe earlier in this paper, and a non MPI serial version (`bwtserial`) of the same code.

To introduce the `bwtparallel` module, we created a new main file `main_bwtparallel.cc` that initializes MPI and calls the function `bwtzipMain` in `bwtzipp.hh`, which is a modification of the original `bwtzip.h` to use MPI primitives to make sure that work that is supposed to be done in serial is only executed in node 0 and that BWT is executed in all nodes.

We create `bwtzip_parallel.cc` based on `bwtzip_suffixarray.cc` to make the call to the parallel suffix sorting algorithm that we implemented. The window size for the parallel suffix sorting is defined in `main_bwtparallel.cc` and is currently set to 2.

Also, the `Makefile` was editted to use mpic++ and the proper flags and to accept the arguments `bwtparallel` and `bwtserial` in order to compile our new modules.

For further reference about code structure, compiling and running the software, refer to the `README` file under the tar ball.

6 Performance

We measure our performance in terms of the time spent in the Burrows-Wheeler transform. We measured this running time for three different inputs: a text file with the book "War and Peace" (3MB), a text file with the first 10MB of the bases of the human chromossome 19 and a binary file containing a picture in bitmap format (18MB).

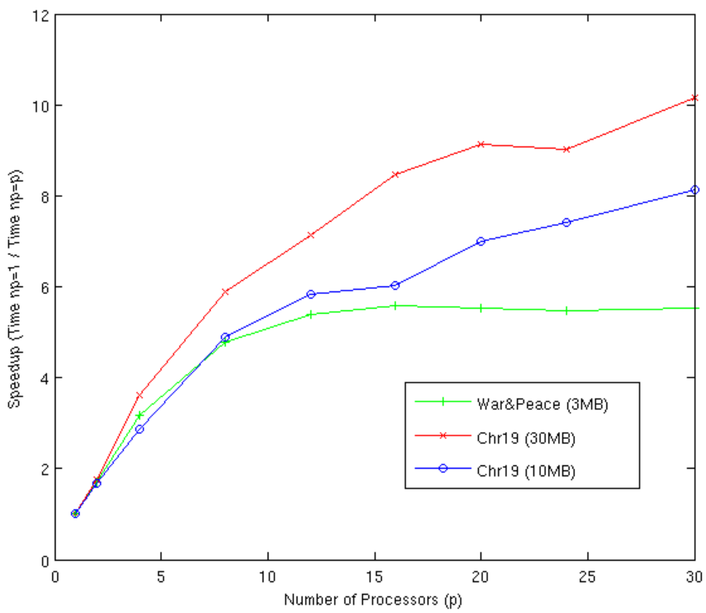

Figure 1: Results of the timing experiments

For each of these inputs, we timed our programs with number of processors 1, 2, 4, 8, 12, 16, 20, 24 and 30.

In all of the timing experiments we used a window size of 2 for the parallel sufix sorting. The result of using a window size of 3 was a much slower bucket distribution due to handling an array of size $|\Sigma|^3$, where $|\Sigma| = 257$, and it was not worth the benefit in load balance.

The results are shown in figure 6 and show a speedup close to linear, specially in the input files that generate a better load balance.

7 Load Balance

A crutial part of our suffix sorting code is the distribution of the buckets to the different processors. The way we do this distribution is the following. Let b_i be the number of suffices in bucket i. Define $t_i = \sum_{k=1}^{i} s_k$ as be the cumulative sum. If m is the number of buckets and p the number of processors, then we let i_k be such that

$$|\frac{m}{p}k - t_{i_k}|$$

is minimal. Then we assign buckets $i_k, i_k + 1, \ldots, i_{k+1} - 1$ to processor k, for $k = 0, 1, \ldots, p - 1$. This is the optimal greedy algorithm for load balancing.

In the ideal case all buckets would have roughly the same amount of suffices so if the number of buckets is much bigger than the number of processors we would expect a very even load distribution. How ever, in common inputs different characters have very different frequencies, therefore different buckets will have very different amounts of suffices. For example, in the input file containing the chromosome, the alphabet is very limited, so the suffices end up being concentrade in very few buckets, creating a bad load balance. This can be observed comparing figures 7 and 7.

Also, an increase in the windows size causes a greater diversity of windows, therefore a smaller granulation of the suffices leading to a better load distribution among processors. This can be observed comparing the two graphs in each of the figures 7 and 7.

8 Compression Rate

We can compare the performance and the compression rate of our parallel compressor with the most popular compressors in the market. For all cases we use the 3MB text input with the book "War and Peace".

	Running time (sec)	Compression rate
bwtparallel (p=27)	5.95 (0.87 + overhead)	76%
bwtparallel (p=1)	9.71 (4.65 + overhead)	76%
bzip2	1.8	73%
gzip	0.82	63%
zip	0.88	63%

As you can see, `bwtparallel` achives the best compression rate. The running time is considerably slower than the other compressors, but that is reasonable since we are working with a research-grade code not very fine tuned.

9 Conclusion and Future directions

We presented a parallel implementation of a Suffix Sorting algorithm and demonstrated an application to the BWT and data compression. Subsituting our BWT into the freely-available compression program bwtzip, we observed linear speedup in cpu count for sample inputs while still achieving compression rates better than and speed on the order of the popular compressor bzip2.

References

[1] U. Manber and G. Myeres, Suffix arrays: A new method for on-line string searches," SIAM Journal of Computing, vol. 22, no. 5, pp. 935-948, 1993.

[2] Bentley, J. L., and Sedgewick, R. 1997. Fast algorithms for sorting and searching strings. In Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, pp. 360-369.

[3] N. Futamura, S. Aluru and S. Kurtz, A Practical Algorithm for Parallel Suffix Sorting, 9th International Conference on Advanced Computing and Communications (2001)

[4] Stephan T. Lavavej and Joergen Ibsen, BWTZIP, http://nuwen.net/bwtzip.html

[5] E. Ukkonnen. Approximate string-matching over suffix trees. Algorithmica, 14:249-60 (1995).

[6] P. Weiner. Linear pattern matching algorithms. Proc 4th IEEE Symp. on Switching and Automata Theory. pp. 1-11 (1973).

[7] D. Gusfield. Algorithms on Strings, Trees and Sequences. Cambridge University Press, New York. (1997)

[8] M. Burrows and D. J. Wheeler, "A Block-Sorting Lossless Data Compression Algorithm", Digital Systems Research Center Research Report 124 (1994)

[9] Bierbrauer, Juergen. Introduction to coding theory / Boca Raton, Fla. : Chapman & Hall/CRC, c2005.

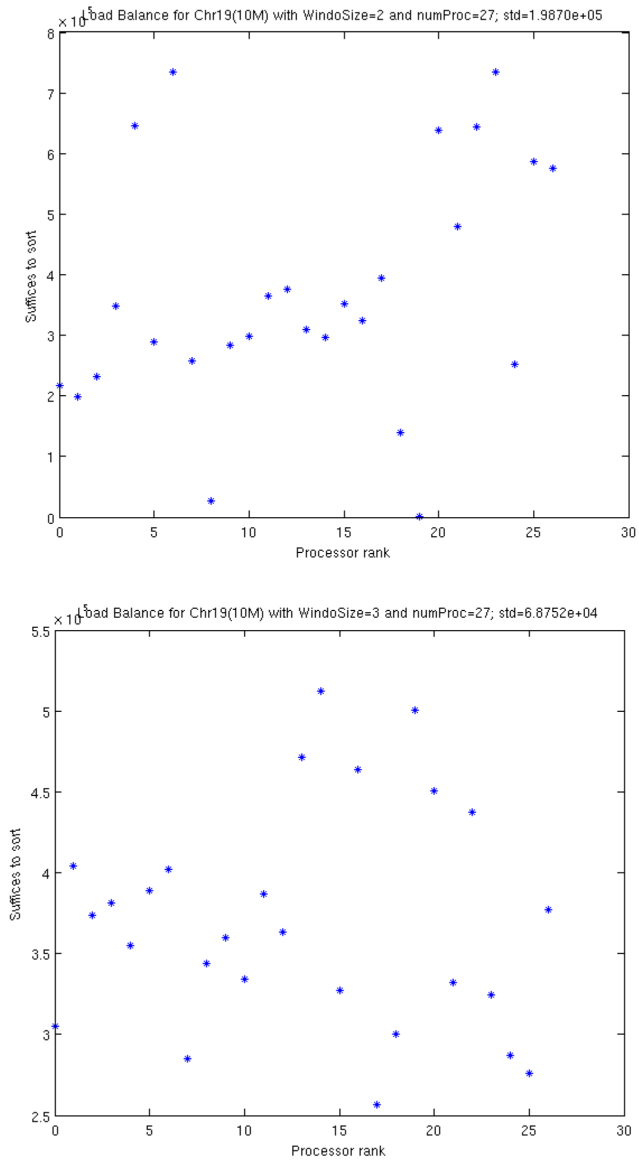

Figure 2: Load balance for input Chromossome 19 (10M) and window size 2(top) and 3(bottom)

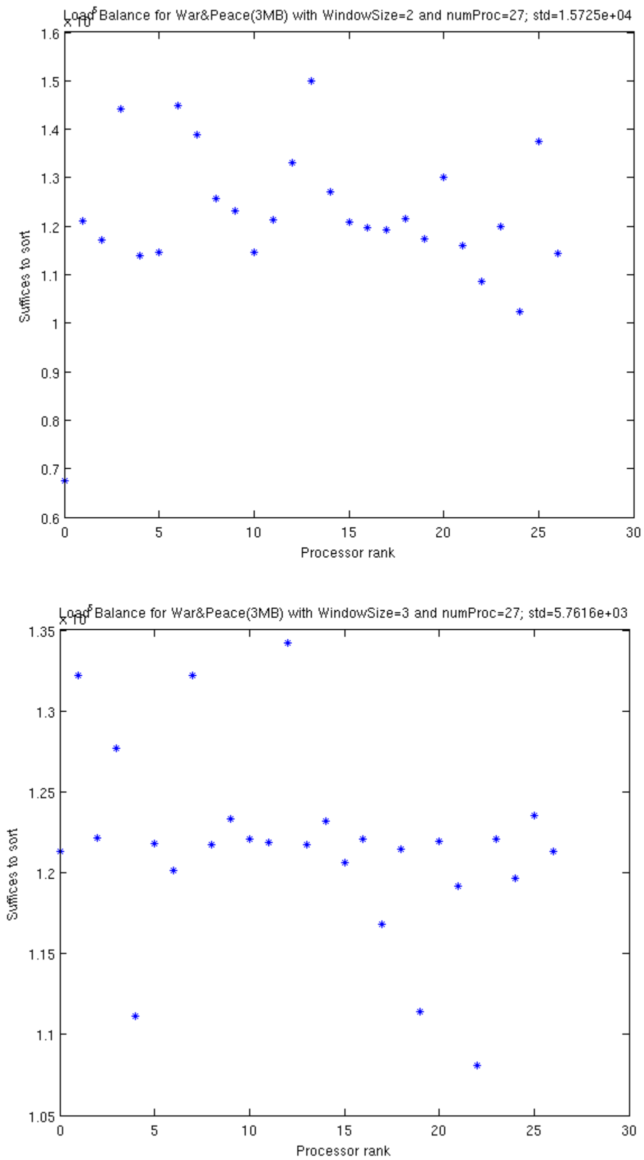

Figure 3: Load balance for text input "War and Peace" (10M) and window size 2(top) and 3(bottom)