6.338 Final Paper: Parallel Huffman Encoding and Move to Front Encoding in Julia

Gil Goldshlager

December 2015

1 Introduction

1.1 Background

The Burrows-Wheeler transform (BWT) is a string transform used for data compression and string indexing. It achieves high rates of compression, even better than common tools like gzip, but at the cost of taking more time to compute. Though the transform is fundamentally difficult to parallelize, in 2013 and 2014, Edwards et al discovered effective PRAM algorithms to perform the transform, and implemented the algorithms on their XMT platform to produce speedups up to 70 time faster than previous work [1],[2]. Their algorithms involve using fine-grained parallelism difficult to achieve in many standard programming languages and difficult to take advantage of on standard processors. Therefore, for my project, I endeavoured to implement their algorithms in Julia, on a standard 6 core machine, in order to see how easy it would be to write the code, and whether speedups could in fact be achieved.

1.2 Specific Aims

The BWT consists of three parts: the block sorting transform, move-to-front encoding (MTF), and Huffman encoding. I initially wanted to implement all three portions of the algorithm. However, it became obvious with further research that block sorting transform algorithms, essentially equivalent to suffix array construction, are extremely complicated (even in serial), and that it would be more than a class project to develop a parallel algorithm for that portion of the transform. Thus, I focused on adapting the MTF and Huffman encoding and decoding algorithms for a standard machine architecture, and implementing them in julia. Unfortunately, the time bottleneck of the overall BWT is in fact the block sorting transform, so speedups on the two parts I studied will not give significant speedups for a full BWT compression scheme. However, MTF and Huffman encoding are of independent interest, and will still serve as a good case study of implementing non-embarrassingly parallel algorithms in Julia. All code for this project can be found on GitHub at https://github.com/ggoldsh/ParallelBWT.

2 Definitions

Definition 1. *Let Σ be the alphabet over which the strings that we wish to encode are defined. Let $|\Sigma| = A$.*

Definition 2. *Let Π be the set of integers $\{0, 1, \cdots, A-1\}$.*

Definition 3. *Let B be the set $\{0, 1\}$.*

Definition 4. *For any set X, let $S(X)$ be the set of all sequences of elements of X. For example, $S(\Sigma)$ is the set of all strings that we can encode.*

Definition 5. *For any sequence S, let S_i be the ith element of S. Let $S_{i,j}$ be the subsequence of S starting at S_i and ending at S_j.*

3 Serial Algorithms

In this section, I describe the four component algorithms that I implemented: MTF encoding, MTF decoding, Huffman encoding, and Huffman decoding. The full details of all of these algorithms are described in [1], so I will only provide an overview.

3.1 MTF Encoding

The MTF encoding algorithm maps an element of $S(\Sigma)$, called the input string, to an element $T = MTFE(S) \in S(\Pi)$, called the encoding. We calculate the encoding as follows. We initialize an auxiliary array L to contain the alphabet in order. Then, for every character S_i of S, we find S_i in L, and write out the index j so that $L_j = S_i$. We then move L_j to the front of L (hence move-to-front encoding), and push every character that preceded L_j one position back in the array. For example, we could move 4 to the front of $[2, 3, 4, 1]$ to get $[4, 2, 3, 1]$. After doing this for every character in S, we obtain an output string $T \in S(\Pi)$ which is the MTF encoding of S.

3.2 MTF Decoding

MTF decoding takes in an array $T \in S(\Pi)$, called the encoded string, and returns the a string $S = MTFD(T) \in S(\Sigma)$, called the decoding of T. Of course, MTF decoding is defined so that, for any $S \in S(\Sigma)$, $MTFD(MTFE(S)) = S$. To calculate this decoding, we initialize an auxiliary array L to contain the alphabet in order. Then, for each integer T_i in T, we first write down the character L_{T_i} to our decoding array. We then move L_{T_i} to the front of L, as in the encoding algorithm, and then proceed to the next character of T. This produces a decoding $S \in S(\Sigma)$ and can be easily proven to be the inverse of the MTF encoding algorithm.

3.3 Huffman encoding

The Huffman encoding algorithm is very straightforward. It takes in an array $S \in S(\Pi)$, called the input array, and outputs an array $T = HE(S) \in S(B)$, called the Huffman encoding of S. To calculate T, we first count the occurrences of each element of Π in S. We then build up an encoding tree for the array which maps each integer in Π to a bit string such that no one encoding is a substring of another. For example, one encoding of $\{0, 1, 2, 3\}$ would be $0 \to 0$, $1 \to 10$, $2 \to 111$, $3 \to 110$. The key insight of Huffman encoding is that it is possible to find the optimal such encoding tree very quickly; however, this part of the algorithm is not important to my project, so I won't discuss it here. Finally, given the coding, we iterate over every element S_i, and write its bitstring code to our output array. This produces our Huffman encoding T.

3.4 Huffman decoding

The Huffman decoding algorithm takes as input an encoding tree and an input string $T \in S(B)$, and produces as output $S = HD(T) \in S(\Pi)$, which is the Huffman decoding of T. To calculate S, we first initialize an auxiliary bitstring C to be the empty string. We then traverse T, adding one bit at a time from T to C. If, when we add T_i to C, C matches a code for some integer x in our encoding tree, we clear C and write x to our output array. When we reach the end of T, we should have an empty array C (otherwise the string we are trying to decode was not properly encoded), and our output array will contain $HD(T)$.

4 Parallel Algorithms in Theory

Just as in the last section, we will separately describe the parallel algorithm for each step that I implemented.

4.1 MTF Encoding

To parallelize the MTF encoding algorithm, we cleverly interpret it as a parallel prefix over an appropriate binary operator. First, let $L(i)$ be the array L after processing character S_i of the input string. Then, our first goal is to compute $L(i)$ for all i, which we can do as a parallel prefix. Define the binary associative operation C, which works as follows. Given two character arrays, x and y, C returns a character array containing the elements of y followed by all characters in x which do not also occur in y. As it turns out, we can calculate $L(i)$ by prepending the alphabet in order to S, thinking about each character S_i in S as the array $[S_i]$, and then computing a parallel prefix using operation C over S [1]. For details and proof, see the original paper. Now, once we have $L(i)$ for all i, we can produce the output at any position i by simply finding the integer j such that $L(I)_j = S_i$, and writing that integer to our output array. This is a purely local operation and thus directly parallelizable.

4.2 MTF Decoding

The parallelization of the MTF decoding algorithm is similar, but uses a different binary associative operator. We begin with an array $T \in S(\Pi)$ of integers, and want to produce the decoding string $S \in S(\Sigma)$ of characters. Our first goal is again to compute $L(i)$ for all i. In this case, we realize that, in our decoding algrithm, we permute L the same way every time that we see the same integer $x = T_i$. Namely, we always move L_x to the front of L. As such, we can calculate $L(i)$ for all i as a parallel prefix over the permutations defined by each integer in T, with permutation composition as the binary associative operator. Then, given $L(i)$ for all i, we can produce the decoding by simply writing out $L(i)_{T_i}$ for all i, which we can directly parallelize.

4.3 Huffman encoding

The parallel Huffman encoding algorithm is fairly straightforward, but there is some added complexity due to the fact that the encoding is not one-to-one. That is, the characters of $HE(S)$ do not correspond bijectively to the characters of S, so, when writing our encoding, we need to do some extra calculations to determine to where each coded character should be written. First, to build our coding tree, we need the counts of all characters in the array. This we calculate directly with a parallel prefix computation. We then build up the optimal encoding tree on the master processor, since this computation is on the order of the size of the alphabet, rather than the size of the input string. Now comes the extra step. Given the length of each code from the coding tree, and the cumulative counts at each point, we can compute the position P_i to which the encoding of S_i should be written in the output string, using another parallel prefix. Finally, given P_i for all i, we can write the encodings of each character in parallel to the output array.

4.4 Huffman decoding

The Huffman decoding algorithm is the most involved of the four algorithms. Here the problem is that only certain positions in the input bit string are valid start positions for decoding. Thus, our first goal is to identify a set of such valid starting positions. Let M be the maximal length of a bit encoding of any integer in Π. Then, we first segment the array into bins of length M. In parallel, we calculate a pointer starting at every position p in the input array, which points to a bit in the next bin after that point. To calculate this pointer, we start decoding at p, and the first time we land in the next bin, we record a pointer to the position at which we landed. A further discussion of this computation can be found in [1], but the intuition is that each pointer tells us that, if its start point is a valid decoding start position, so is its end point. Clearly, these pointers can all be calculated in parallel. Armed with these pointers, we can run a parallel prefix computation over the bins, with the operation that combines every pair of pointers $(a \to b), (b \to c)$, into one longer pointer $(a \to c)$. This gives us pointers from every point in the first bin to points in every other bin. If we take the pointers from the first point of the first bin, they point to one valid decoding start position in every bin. Given these start positions, it is possible to decode S in parallel. We first calculate in parallel how many decoding characters separate each of the start points that we have calculated, and use a parallel prefix over these numbers to determine to where each bin maps in the output array. Given this, we can decode each bin directly in parallel.

5 General Implementation Details

5.1 General differences between parallel algorithms in theory and practice

In translating the parallel algorithms described above into code, there was one common pattern that arose. In almost every case, the parallel prefix interpretation of the algorithm is outperformed by the serial algorithm. For example, interpretting MTF decoding as a composition of permutations is less efficient than simply scanning over the array and moving each character successively to the front of L. Thus, rather than running a full parallel prefix over the input, where the distinction between parallel and local operations could be masked from the programmer, the most efficient way to implement the algorithms was to write a serial version to be run on each local process, and then use the parallel prefix operation only to combine the results of the local computations into seeds for the decoding. For example, the composition of permutations idea is necessary to determine the starting state of L for each local process given the output of the serial MTF decoding algorithm on each local portion of the array. However, the local MTF decoding algorithm is faster written in its standard form than in terms of permutations.

5.2 Infrastructure for Parallelism

To implement these algorithms, I used Julia's SharedArray framework. The code written for these algorithms can run on any number of processors, using only a very small number of functions meant to parallelize operations over a SharedArray. With p processors, we always divide a SharedArray S into p equally sized blocks over which we can parallelize operations. Of course, every processor has access to the whole array, so this partitioning constraint is entirely self-imposed. However, it does mean that the algorithms could easily be transformed to work on DArrays.

The primary operations I implemented for parallelization are $fanin$ and $fanout$. $fanin$ takes in a function f and a SharedArray S, runs f on every local portion of S, and returns the results of the computation as an array of length p. $fanout$ takes in an array *seed* of length p, a function f, a SharedArray S, and a SharedArray T. *seed* is used to distribute information calculated in a $fanin$, and possibly processed locally, back to the local processes. f then takes a single element of the *seed*, and the arrays S and T, and runs locally on each processor. I used two arrays so that I could encode the array S into the array T without overwriting S, and so that it would work even if the output needs to be a different type of array than the input. It is nice to note that the star topology described by Eka is exactly a $fanin$ operation, followed by a local computation, followed by a $fanout$. While I initially implemented the star operation directly, I later found it more intuitive for me to split up the components into three stages, rather than write them in one command, leading to the $fanin$ and $fanout$ structure I described.

Throughout this project, I saw the utility of such a framework for hiding the parallelism used when writing up each of the four algorithms I implemented. The *star*, or $fanin/fanout$ structure, is just the right level of abstraction, where the programmer is separated from low level parallelizatoin calls, but still has the power to write code that will perform well. For example, a simple parallel prefix structure which used the same algorithm to combine results locally and globally would be even easier to work with, but would not allow the user to distinguish between using algorithms optimized in serial on each local process, and only using the parallel prefix framework to combine the results globally.

As far as the specific choice of which functions to use as a baseline, I think if I were to start from here, I would implement one basic function, called fan. fan would fundamentally represent one computation performed on each processor on its local portion of the array. It would optionally take an array of seeds, and if seeds were provided, then the function given for the distributed computation would be passed the appropriate element of the seed array. After performing the computation, each processor would optionally fan back in a result from its computation, or return nothing if all that was desired was a mutation. Using this framework, $fanin$ and $fanout$ would simply be special cases of fan, and the star operation would be two fan operations. The advantage would be that simpler computations, like a simple cum sum, or a command to sort each portion of the array, could also be naturally written as a single fan operation without unnecessarily using two parallel calls. Of course, there are surely benefits and drawbacks to each way of doing this, but for me, for this project, putting the three steps of the computation into a single *star* operation was less intuitive than splitting it up into several function calls.

One final interesting implementation detail with respect to using parallel structures that take in functions is that I found I had to be careful about how I created the functions. There were some cases in which each function needed a seed that would be calculated at run time, but the seed was the same across all the portions of the array. For example, in Huffman decoding, each local processor needs the decoding array, but I originally preferred not to put that in the seed of each function, because it was the same for every process. Thus it was appealing to imagine that I could build the function on the local process, by calling a function builder which took the decoding array, and then pass the resulting function to each processor to be run. In hindsight, this doesn't make any sense, because the decoding array will inevitably need to reach each processor, so it might as well be sent to each one as part of the seed. However, it took me a while in debugging to realize that that was significantly slowing down my code (roughly by a factor of 10). While I still don't know what exactly Julia does in this case, or whether this is really an issue of parallelization, or just an issue of trying to encode too much information in a function, it was interesting to see how much of a performance effect that had and how easily I made such a costly mistake.

6 Particular Implementation Details

6.1 MTF encoding and decoding

MTF decoding and encoding worked just about exactly as described above. The parallel prefix operations are implemented by using the serial algorithms within each local portion of the array, and using the parallel prefix framework only to combine these results on the master node into seeds for the next step of the computation.

6.2 Huffman encoding

Relative to the algorithm described above, and the previously made implementation comments, there is only one further modification I made to the Huffman encoding algorithm. That modification is that, instead of using one parallel prefix operation to count the characters, and then another one to calculate starting points for each local process, we can in fact avoid making the second computation altogether. This is because the first parallel prefix gives us the counts of each character up to the starting point of each array. On the master process, we can then use the determined codes to calculate how many bits each local portion's encoding should take, giving us the starting points of each process without another parallel operation. This takes out the only extra step performed by the parallel algorithm over the serial algorithm, giving us near perfect parallel speedups.

6.3 Huffman decoding

The Huffman decoding algorithm is, again, a little more involved, and the most interesting part of my project. Recall that the parallel algorithm described above first bins the input into bins of length M, where M is the maximum code length. It then calculates pointers from each bin into the next bin showing where a decoder would end up in the next bin if it started decoding at each point in one bin. To modify this procedure for an array divided into p blocks, the most natural method is to have each processor locally decode, starting at each of the first M positions in its portion of the array, and create a pointer from each of those M start points into the next processor's portion of the array. We can then combine these pointers with a parallel prefix computation on the master process, calculating one valid decoding start position at the beginning of every local portion of the array. We can simultaneously fan in the number of characters used for each pointer calculated, allowing us to also calculate positions in the output array coresponding to each valid start point. Finally, we can *fanout* the startpoints and the local decoding computations.

The problem with this, and the algorithm described in the paper, is that they both require roughly M times as much total work as the serial decoding algorithm, since they require decoding every position in the array up to M separate times. In the original paper, this was overcame by parallelizing over many, many processors. In my project, I had access to a machine with 6 processors, so the factor of M more than overwhelmed the speedup I could hope to obtain from parallelization. Luckily, there is an algorithmic solution.

The solution relies on an insight about Huffman decoding from different points in the input string. The insight is that, if we decode an input string from M consecutive start locations, it is almost impossibly unlikely that the decoding paths found by the M start points stay distinct. That is, the paths almost certainly converge very quickly into a single path. This is for the simple reason that, every time two separate paths starting near each other decode a character, there is some chance that they converge. On the other hand, once they converge, they can never separate. In practice, this means that the total amount of computation we need to do should have some component based on the size of the alphabet, representing the computation from the beginning of the bin until the convergence point of the M paths, and then another, additive component that is proportional to the sequence length, representing the decoding of the single converged path from where it converges until the start of the next bin. This is much better than before, when the factor of M was multiplied into the computation on the length.

To implement this effectively in practice, I divided each processor's local portion of the array into medium-sized chunks. By medium-sized, I mean that I wanted the chunks to be big enough that the computation of decoding over a chunk would be much larger than the bookkeeping needed to check for the convergence of paths at the end of the chunk. On the other hand, I wanted them to be small enough that I could afford to process the whole first chunk naively before checking for redundant paths. Given such properly sized chunks, the algorithm starts off with all the start points in the first chunk, and decodes each of them until it reaches the second chunk. At the beginning of each chunk, it then checks for paths that converged, and only continues decoding once from each distinct start point reached. This results in an extremely efficient computation which has no significant speed loss over simply decoding a single path of the same length as the local portion. In every case I saw, all paths converged within the first chunk of the local portions, which were designed to be at least 10000 bits each, and if possible, at most 100 times smaller than the local portion of the array. Importantly, while my algorithm takes advantage of convergence, it doesn't assume convergence; thus, a pathological input could cause it to run slowly, but could not cause it to return an incorrect result. Overall, using this modified algorithm, I was able to achieve significant parallel speedups, even with only 6 cores at my disposal. In particular, using the parallel algorithm on one processor resulting in a factor of 2 increase in runtime, which was then translated

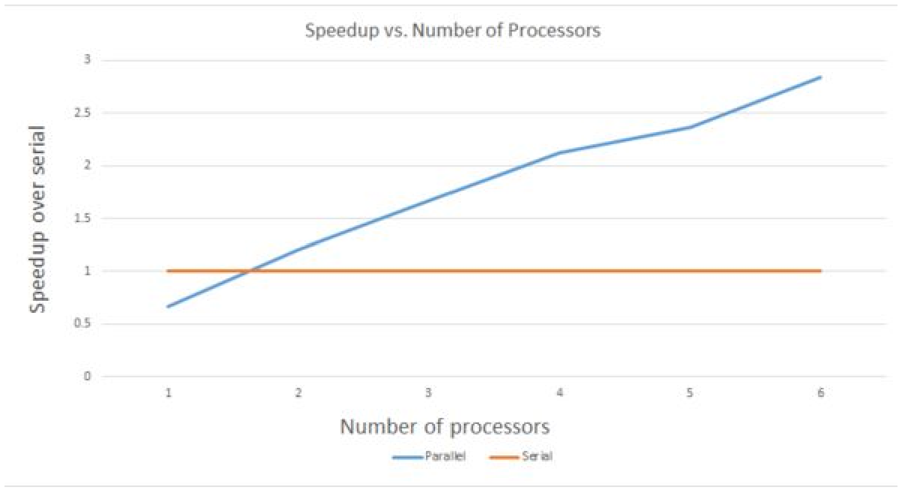


Figure 1: Runtime speedup of entire encoding and decoding process on between one and six processors

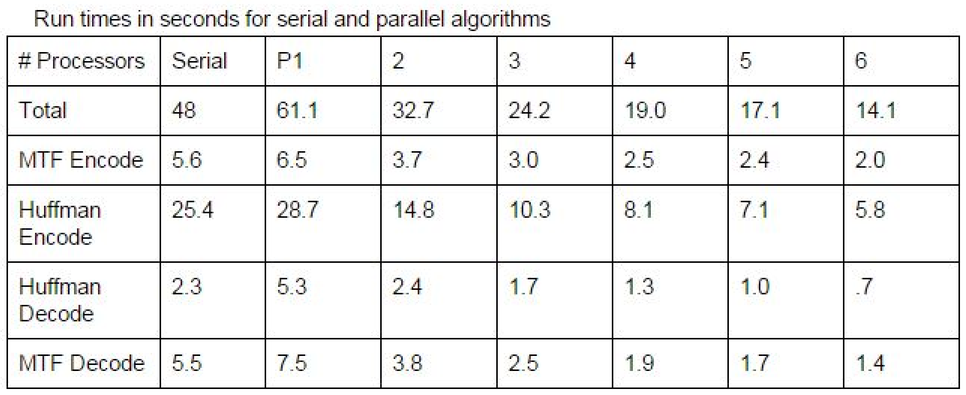
into a 3 times speedup for running the parallel algorithm on 6 processors.

7 Results

The results of my implementation of these algorithms were very encouraging. Once the kinks of the parallelization were worked out, I obtained significant speedups on every part of the algorithm, as shown in Figures 1 and 2. Overall, the full process takes about twice as long to run in its parallel form as in its serial form on a single processor. Thereafter, as the number of processors used increases, at least up to 6 (and on a single machine), the speedup scales nearly linearly with each processor added, resulting in a speedup of almost 3x for 6 processors.

8 Conclusions

Overall, I learned a lot about designing and implementing parallel algorithms throughout the course of this project. Julia provided a relatively convenient base on which to build parallel functionality, although it was very confusing at first to figure out how to handle modules on multiple processors. As an overall implementation strategy, I first played around with implementing various very simple parallel operations (doubling the entries of an array, or setting them, or calculating a cumulative sum) in order to ensure that the parallelization structure I was using was effective and could obtain speedups on the architecture I had. Thereafter, once that was tested, I never had to worry about it again,



Run times in seconds for serial and parallel algorithms

# Processors	Serial	P1	2	3	4	5	6
Total	48	61.1	32.7	24.2	19.0	17.1	14.1
MTF Encode	5.6	6.5	3.7	3.0	2.5	2.4	2.0
Huffman Encode	25.4	28.7	14.8	10.3	8.1	7.1	5.8
Huffman Decode	2.3	5.3	2.4	1.7	1.3	1.0	.7
MTF Decode	5.5	7.5	3.8	2.5	1.9	1.7	1.4

Figure 2: Times in seconds for each portion of the algorithm as a function of the number of processors.

which made the whole process much easier from a debugging and engineering standpoint.

Finally, in terms of the algorithms I implemented, the conclusions are very positive: Huffman encoding and MTF encoding can both be effectively parallelized on standard machines in Julia. An interesting further project would be to design a full BWT in parallel, and see if it is still possible to get such effective speedups. It would also be interesting to implement my algorithms for DArrays and then try running them on more processors and over machine clusters to see if the parallel speedups continue to be linear.

References

[1] Edwards, J. A., and Vishkin, U. (2014). Parallel algorithms for Burrows–Wheeler compression and decompression. Theoretical Computer Science, 525, 10-22. Chicago

[2] Edwards, James A., and Uzi Vishkin. "Empirical speedup study of truly parallel data compression." (2013).