Engineering a Blockwise Suffix Array Builder for the Burrows-Wheeler Transform

Benjamin Langmead
CMSC858P Final Project, Spring 2008

The Burrows-Wheeler Transform (BWT) is a critical tool for searching and indexing biological sequences, particularly when the sequence in question is very long, as in the case of the 3-billion-character human genome. Unfortunately, computing the BWT of very long sequences can be difficult in practice. This is because most methods for computing the BWT require that the sequence's suffix array (SA) be computed in its entirety first, which requires a very large amount of memory, even exceeding the available virtual memory on a computer. Recent work by Kärkkäinen [1] proposes an alternate BWT-building technique that reduces the memory requirement dramatically by building the suffix array "block-by-block" and discarding blocks after calculating the corresponding sections of the BWT. The goal of this work is to implement an efficient version of Kärkkäinen's algorithm in a way that enables time- and memory-efficient calculation of BWTs and similar suffix-array-derived data from DNA inputs. We also seek to characterize how the parameters of the algorithm affect runtime and memory performance. This paper present results from using the blockwise algorithm to calculate the EBWT (an indexed version of the BWT similar to the FM Index [6]) over the entire human genome.

Algorithm

The chief insight behind Kärkkäinen's technique is that the value of any given element of the BWT depends only on a *single* element of the suffix array, according to the following equation.

$$BWT[i] = \begin{cases} T[SA[i]-1] & SA[i] \neq 0 \\ \$ & SA[i] = 0 \end{cases}$$

Equation relating elements of the suffix array to characters in the Burrows-Wheeler text.

For this reason, constructing the entire SA at once is not necessary. Instead, we can divide the suffix array into blocks, compute each SA block separately, compute the corresponding block of the BWT, then discard the SA block and reclaim its memory. BWT blocks are much smaller than SA blocks (especially for bit-packed DNA texts) and can often be written directly to disk instead of occupying memory.

The overall blockwise BWT construction algorithm is as follows:

1. Choose a random sample of "splitter" suffixes, which must include the least and greatest suffixes
2. Sort the splitter suffixes lexicographically
3. For each consecutive pair of splitters, starting with the lexicographically smallest:
    a. For each suffix of the input text, add it to a list if it falls lexicographically between the splitters at either end of the current block
    b. Sort all suffixes in the list accumulated in 3a, yielding a block of the suffix array

c. Compute the Burrows-Wheeler transform corresponding to the block and immediately discard the block

Care must be taken to choose splitters that result in a relatively even distribution of block sizes across the suffix array. My implementation integrates something akin to the heuristic suggested by Kärkkäinen on page 10 of [1] of choosing more splitters than are needed, determining the sizes of the resulting buckets using Myers-and-Manber binary search [4], then doing a round of splitting and merging and, if any blocks were still too big, iterating again.

Kärkkäinen's goal is an efficient (better-than-quadratic) algorithm using sublinear space. The potential performance bottlenecks of the algorithm are steps 2, 3a, and 3b. Kärkkäinen proposes an algorithm based on Z boxes for step 3a that is linear in the length of the input text. However, we are left with two problems. First, the Z-box-based algorithm requires space linear in the length of the text in order to store the Z values. Second, neither of the suffix sorting steps (2 and 3b) can straightforwardly give sub-quadratic time and sublinear space, though one or the other are both possible. A doubling algorithm like Manber-Myers [4] or Larsson-Sadakane [5] yields O(n log n) running time but runs in linear space. A traditional sorting algorithm such as multikey quicksort [9] yields O(log n) space but runs in quadratic time.

Kärkkäinen's solution to both problems is the difference cover (DC) sample, a concept first applied to suffix sorting by Burkhardt and Kärkkäinen [2]. The idea behind DC is to sort a relatively small subset (a "sample") of the suffixes in the overall string up front, then use that ordering to help determine orderings in later stages. If the sample is small and sufficiently cleverly chosen, it can both fit in sub-linear space and improve the problematic time and space bounds by answering certain critical "tie-breaking" queries in constant time. See [1] and [2] for details about how tie breakers are used to improve the problematic time and space bounds from steps 2, 3a and 3b, and why it is possible to answer tie-breaking queries in constant time using the difference cover.

Implementation

I have implemented all parts of the algorithm described in [1] in C++, including building and using the difference-cover sample, though I make calls into the SeqAn [3] library in places where a traditional suffix-array sorter is needed (e.g., in step 5.1.3 of calculating the difference cover, as described in [2]). My implementation is highly parameterized and configurable from the command line and supports a variety of input formats.

This project is implemented within the framework of the SeqAn library for C++ [3]. To use the library, users must place a copy the provided code (which includes a slightly modified version of the March 2008 snapshot of the SeqAn library) in their include path and #include the "blockwise_sa.h" header in their code. The KarkkainenBlockwiseSA template class encapsulates all relevant functionality. Because the implementation adopts SeqAn's Sequence "concept" (family of template classes and functions), users of the KarkkainenBlockwiseSA class must provide the input text in the form of a SeqAn String<>. String<>'s can easily be built from C char*'s or C++ strings, and SeqAn supplies a variety of ways of reading Strings<> directly from Fasta, Genbank, Embl, and raw files. SeqAn also makes it easy to design applications that work with either packed or unpacked representations (though that facility of SeqAn is flawed, requiring a few ugly workarounds [10]).

The implementation strives to be economical in its use of memory. SA blocks are discarded

as soon as possible and BWT blocks are written to disk instead of being kept in memory needlessly. As explained below, the implementation is capable of producing an EBWT for the entire human genome in less than 10 GB of memory in just over 3 hours.

The project submission includes the source file "bwt.cpp" which is a simple example of how to use the library and the KarkkainenBlockwiseSA class to produce a stream of sorted suffixes. The sample application uses the stream to calculate and print a Burrows-Wheeler transform. Here is the (short) code listing for that program:

```
#include "blockwise_sa.h"
#include <seqan/file.h>

/**
 * Example application showing how to use the KarkkainenBlockwiseSA
 * class to obtain a stream ordered suffixes of an input text.  The
 * KarkkainenBlockwiseSA class is working behind-the-scenes to divide
 * the suffix array into blocks and calculate each block as needed.
 */
int main(int argc, char** argv) {
    typedef String<Dna> TStr;
    TStr input;
    if(argc < 2) {
        cerr << "Must specify Fasta input file as first argument" << endl;
        return 1;
    }

    // Read a DNA sequence in from a Fasta file
    std::ifstream in(argv[1], std::ios_base::in);
    read(in, input, Fasta());
    if(empty(input)) {
        cerr << "Bad or empty input file" << endl;
        return 1;
    }

    // Construct a blockwise SA builder.  First argument is the input
    // text, second argument is the difference-cover periodicity, and
    // third argument is the maximum block size.
    KarkkainenBlockwiseSA<TStr> sa(input, length(input)/5, 1024);
    while(sa.hasMoreSuffixes()) {
        // Get the offset of the next greatest suffix
        uint32_t suf = sa.nextSuffix();
        // Output BWT char
        cout << ((suf == 0)? input[length(input)-1] : input[suf-1]);
    }
    cout << endl;
    return 0;
}
```

Results

In all cases, sequences used were taken from the "Contig" section of Genbank's human genome build 36.3. "Contig" versions were used instead of "Assembled" versions because the extremely long stretches of Ns present in the assemblies pose a problem to the current version of the ebwt_search application, which I intend to use to search the resulting EBWT indexes in future work. All experiments except for those involving the whole human genome were performed on sycamore.umiacs.umd.edu. Sycamore has 32 GB of RAM and has 4 x dual-core 2.2 Ghz AMD Opteron 875 Processors.

All runtimes are taken from the "user time" result returned by the getrusage() function [7]. In no case was the "system time" a significant component of runtime. Memory footprints were calculated from snapshots taken once per second by top [8]. (Note that this is not a totally accurate measure of peak footprint, since peaks may occur between samples, and thus go unnoticed. sycamore.umiacs.umd.edu seems not to maintain the "VmPeak" statistic in /proc/(pid)/status, which is the more well known and reliable way of measuring peak usages.)

In the column headers of the tables below, "VM" stands for virtual memory and "RES" stands for resident (non-swapped) physical memory. These are both measured by top.

Block Size

The following table shows the impact of maximum block size on the time- and memory-performance of the Blockwise EBWT Builder. These results are shown for three texts of different sizes: human chromosomes 21 (25 Mbases) 11 (131 Mbases) and 1 (226 Mbases).

Run type	**Runtime**	**Peak VM footprint**	**Avg. VM footprint**	**Peak RES footprint**	**Avg. RES footprint**
Chromosome 21 (25 Mbases)					
Unpacked Max block sz: **32M** (2 blocks) Diff cover periodicity: 1024	0m:51s	329 MB	188 MB	176 MB	136 MB
Unpacked Max block sz: **16M** (3 blocks) Diff cover periodicity: 1024	0m:56s	192 MB	142 MB	144 MB	107 MB
Unpacked Max block sz: **8M** (6 blocks) Diff cover periodicity: 1024	0m:58s	124 MB	69 MB	93 MB	74 MB
Unpacked Max block sz: **4M** (13	1m:19s	N/A	N/A	68 MB	56 MB

blocks) Diff cover periodicity: 1024					
Unpacked Max block sz: **2M** (25 blocks) Diff cover periodicity: 1024	1m:47s	N/A	N/A	57 MB	48 MB
Unpacked Max block sz: **1M** (49 blocks) Diff cover periodicity: 1024	2m:51s	N/A	N/A	59 MB	44 MB
Chromosome 11 (131 Mbases)					
Unpacked Max block sz: **128M** (2 blocks) Diff cover periodicity: 1024	3m:39s	1370 MB	662 MB	977 MB	553 MB
Unpacked Max block sz: **64M** (4 blocks) Diff cover periodicity: 1024	4m:18s	697 MB	526 MB	562 MB	393 MB
Unpacked Max block sz: **32M** (5 blocks) Diff cover periodicity: 1024	3m:56s	458 MB	375 MB	381 MB	298 MB
Unpacked Max block sz: **16M** (10 blocks) Diff cover periodicity: 1024	5m:55s	319 MB	269 MB	264 MB	214 MB
Unpacked Max block sz: **8M** (22 blocks) Diff cover periodicity: 1024	7m:11s	246 MB	230 MB	206 MB	185 MB
Unpacked Max block sz: **4M** (44 blocks) Diff cover periodicity: 1024	9m:54s	320 MB	194 MB	185 MB	161 MB
Chromosome 1 (226 Mbases)					

Unpacked Max block sz: **128M** (3 blocks) Diff cover periodicity: 1024	10m:07s	1155 MB	936 MB	863 MB	659 MB
Unpacked Max block sz: **64M** (5 blocks) Diff cover periodicity: 1024	11m:17s	870 MB	709 MB	706 MB	557 MB
Unpacked Max block sz: **32M** (9 blocks) Diff cover periodicity: 1024	11m:26s	543 MB	484 MB	484 MB	399 MB
Unpacked Max block sz: **16M** (19 blocks) Diff cover periodicity: 1024	13m:28s	434 MB	404 MB	368 MB	328 MB
Unpacked Max block sz: **8M** (38 blocks) Diff cover periodicity: 1024	18m:13s	387 MB	344 MB	318 MB	285 MB
Unpacked Max block sz: **4M** (77 blocks) Diff cover periodicity: 1024	27m:25s	374 MB	315 MB	318 MB	269 MB

There is consistent direct relation between the number of blocks and the running time of the algorithm, which is not surprising since the construction of each block requires a separate pass over the entire input text (step 3a of Kärkkäinen's algorithm). On the other hand, there is also a consistent inverse relation between the number of blocks and both peak and average memory usage. This too is expected since fewer and larger blocks clearly lead to more peak and average memory usage for storing and sorting those blocks.

For each chromosome, there seems to be a point of "diminishing returns" as the number of blocks increases. After that point peak and average memory usage decrease very little or none at all while runtime increases substantially. This implies that for a given input text there will generally be a "sweet spot" where the blockwise construction yields a substantial space savings without affecting runtime too badly. For chromosomes 1, 11 and 21, all other things being equal, that points seem lie at roughly 10 blocks.

Entries marked as "N/A" in the VM columns for the smaller block sizes of Chromosome 21 are instances where top produced an incongruously small result (much less than the result for RES).

Note that the number of blocks for a particular run does not correspond cleanly to the

maximum block size (e.g. halving the block size from 64M to 32M leads to only 1 additional block for Chromosome 11). This is because of the randomness inherent in the process of selecting samples, as well as the tendency of the implementation heuristic to settle quickly on a set of bucket boundaries that is "good enough."

Difference Cover

The difference-cover sample can be constructed in sub-quadratic time and sub-linear space, and can break lexicographical "ties" between two suffixes in constant time when those suffixes share a sufficiently long prefix. In Kärkkäinen's algorithm, the difference cover sample is constructed up-front and then used to aid sorting (in steps 2 and 3b) and block accumulation (step 3a). Without the difference cover sample, sorting would be quadratic in the worst case, where the worst case is a pathologically repetitive input sequence. Nonetheless, it is not obvious whether the quadratic worst case manifests itself in practice, and thus it is not obvious whether the difference cover is a "win" overall.

Whether to use a difference-cover sample and, if so, what period to use, are both configurable parameters to the KarkkainenBlockwiseSA class. Like Kärkkäinen's, my implementation constrains the periodicity to be a power of 2 in order to avoid slow division and modulus operations. The following table shows how adjusting those parameters affect runtime and memory usage for Chromosomes 1, 11 and 21.

Run type	**Runtime**	**Peak VM footprint**	**Avg. VM footprint**	**Peak RES footprint**	**Avg. RES footprint**
Chromosome 21 (25 Mbases)					
Unpacked Max block sz: 16M (3 blocks) **No difference cover**	1m:00s	154 MB	107 MB	105 MB	83 MB
Unpacked Max block sz: 16M (3 blocks) **Diff cover periodicity: 2048**	1m:01s	157 MB	116 MB	117 MB	90 MB
Unpacked Max block sz: 16M (3 blocks) **Diff cover periodicity: 1024**	0m:56s	192 MB	142 MB	144 MB	107 MB
Unpacked Max block sz: 16M (3 blocks) **Diff cover periodicity: 512**	1m:00s	168 MB	121 MB	143 MB	96 MB
Chromosome 11 (131 Mbases)					
Unpacked Max block sz: 32M	4m:32s	404 MB	337 MB	325 MB	254 MB

(6blocks) **No difference cover**					
Unpacked Max block sz: 32M (6 blocks) **Diff cover periodicity: 2048**	5m:18s	458 MB	348 MB	367 MB	270 MB
Unpacked Max block sz: 32M (6 blocks) **Diff cover periodicity: 1024**	3m:56s	458 MB	375 MB	381 MB	297 MB
Unpacked Max block sz: 32M (6 blocks) **Diff cover periodicity: 512**	4m:46s	473 MB	379 MB	381 MB	287 MB
Chromosome 1 (226 Mbases)					
Unpacked Max block sz: 64M (5 blocks) **No difference cover**	23m:00s	830 MB	653 MB	687 MB	542 MB
Unpacked Max block sz: 64M (5 blocks) **Diff cover periodicity: 2048**	12m:27s	836 MB	695 MB	685 MB	541 MB
Unpacked Max block sz: 64M (5 blocks) **Diff cover periodicity: 1024**	11m:17s	870 MB	709 MB	706 MB	557 MB
Unpacked Max block sz: 64M (5 blocks) **Diff cover periodicity: 512**	10m:47s	833 MB	703 MB	689 MB	530 MB

The only effect that seems significant here is that use of the difference cover seems to substantially improve runtime for Chromosome 1, and that improvement grows as the period decreases (as expected). That this effect is observed only for Chromosome 1 is likely because that chromosome has longes repetitive regions compared to 11 or 21. In no experiment did the construction and use of the difference cover have an obvious effect on memory footprint, though such an effect likely would be visible for periodicities less than 512.

More experimentation is needed to come to firm conclusions, but use of difference cover seems to be critical in keeping the runtime of highly repetitive sequences under control, and

a periodicity in the neighborhood of 512 seems to be a reasonable trade between added memory overhead and runtime improvement.

Packed versus Unpacked Sequence

The following table contrasts runtimes and memory usages of runs over chromosomes 1, 11 and 21 with unpacked (one-base-per-byte) and packed (2-bits-per-base) representations for the sequence.

Run type	Runtime	Peak VM footprint	Avg. VM footprint	Peak RES footprint	Avg. RES footprint
Chromosome 21 (25 Mbases)					
Unpacked Max block sz: 16M (3 blocks) Diff cover periodicity: 1024	0m:56s	192 MB	142 MB	144 MB	107 MB
Packed Max block sz: 16M (3 blocks) Diff cover periodicity: 1024	3m:20s	167 MB	91 MB	119 MB	80 MB
Chromosome 11 (131 Mbases)					
Unpacked Max block sz: 32M (5 blocks) Diff cover periodicity: 1024	3m:56s	458 MB	375 MB	381 MB	298 MB
Packed Max block sz: 32M (5 blocks) Diff cover periodicity: 1024	10m:31s	364 MB	269 MB	287 MB	200 MB
Chromosome 1 (226 Mbases)					
Unpacked Max block sz: 64M (5 blocks) Diff cover periodicity: 1024	11m:17s	870 MB	709 MB	706 MB	557 MB
Packed Max block sz: 64M (5 blocks) Diff cover periodicity: 1024	27m:14s	710 MB	508 MB	546 MB	362 MB

These results show that use of the packed representation imposes a significant runtime penalty - more than doubling the runtime of the unpacked version in all cases. This is roughly as expected, since the packed representation requires a few additional bit operations (bitwise shifts and ANDs) every time the EBWT builder accesses a character in the original text. The payoff in decreased memory footprint is relatively small, especially in relation to cost in terms of slowdown.

These results, together with the results presented in "Block Size" above, suggest the following rule of thumb: if the user wishes to reduce memory footprint and has control over both the block size and whether to use a packed or unpacked representation, the best option is to leave the representation unpacked and reduce the block size until the desired threshold is met. If the desired threshold cannot be met by adjusting block size alone (e.g. if the unpacked input string itself is greater than the threshold), and the user can afford to increase runtime by 2-3x, then using the packed representation is a good option. An instance of the latter scenario will be seen in the next section when we attempt to build an EBWT for the whole human genome within a 2 GB virtual memory budget.

Whole Human Genome

Run type	**Runtime**	**Peak VM footprint**	**Avg. VM footprint**	**Peak RES footprint**	**Avg. RES footprint**
Human Genome (2.87 Gbases)					
Unpacked Max block sz: **1,280M** (3 blocks) Diff cover periodicity: 1024	3h:09m	9,675 MB	4,048 MB	11,264 MB (?)	8,190 MB (?)
Unpacked Max block sz: **128M** (33 blocks) Diff cover periodicity: 4096	15h:39m	2,206 MB	1,960 MB	8,192 MB (?)	5,116 MB (?)

A good test for this blockwise implementation is whether it can build an EBWT of the whole human genome without exhausting virtual memory on a typical server. As shown below, using the implemented blockwise SA algorithm accomplishes this goal with as few as 3 blocks in just over 3 hours and in under 10 GB of virtual memory at peak. These are reasonable time and space budgets for researcher using a server.

A more challenging test is whether the tool enables whole-human EBWT building on a typical *workstation.* Workstation processors are generally comparable to server processors, but workstations generally have much less physical memory than servers. Dell Precision T3400 workstations, for example, come equipped with 1 GB of RAM by default, but can be upgraded to 4 GB of RAM for a total system price of less than $1,000 on Dell's website.

The final row of the table above shows an attempt to demonstrating how this implementation can be made to work on such a workstation. My goal was to bring peak VM footprint in below 2 GB. While it averaged less than 2 GB, the run spent a good deal of time at 2206m VM usage in the block sorting phase. Accomplishing this footprint required using

a packed representation for the input sequence, increasing the difference-cover periodicity to 4K, and decreasing the max block size to 128 million (which yields 33 blocks). It is plausible that further tweaks for these parameters could allow it to fit in less than 2 GB of virtual memory at peak, thereby allowing it to run with little or no thrashing on many workstations. The runtime, while not fast, is probably fast enough for overnight runs.

These whole-human tests were performed on privet.umiacs.umd.edu, which has 32 GB of RAM and 4 x 2.4 Ghz AMD Opteron 850 Processors.

I do not understand why the reported peak and average RES footprint are greater than the reported peak and average VM footprints in both cases. The reported RES footprints are almost certainly inaccurate.

Algorithm Bottlenecks

Three components of the blockwise EBWT-construction algorithm as implemented stand out as taking significant time :

1. Sorting blocks (step 3b)
2. Accumulating suffixes into a block (step 3a)
3. Binary sort for calculating bucket sizes (step 1)

Sorting blocks dominates the runtime of every long experiment shown above. When the number of blocks becomes particularly large, then accumulating suffixes begins to take a greater share of runtime, but never more than about 20%. For instance for the 3-block whole-genome run spends about 80% of its time sorting blocks, while the 33-block whole-genome run spends about 65% of its time sorting blocks.

Other Performance Factors

Page faults were not a major issue on sycamore because of its huge supply of phyiscal memory (32 GB). One a handful of "major" (slow, I/O-triggering) page faults occurred across all off the experiments described above. Major and minor faults were measured via calls to getrusage().

In all of the above experiments, only a tiny fraction of the "real" runtime is "sys time" (time spent in the kernel because of a system call; e.g., a major page fault).

I did not have a chance to measure finer-grain effects such as cache misses.

Possible Future Work

1. Improve multikey quicksort with common optimizations (median-of-3 pivot selection; pseudo-median-of-9 for large subproblems; selection sort for small subproblems)
2. Parallelize; the blockwise algorithm makes this easier!
3. Produce results for a wider range of difference-cover periodicities
4. Implement something other than the BWT transformation (e.g., LCPs) using the blockwise suffix-array construction
5. Measure fine-grain performance features such as cache misses

References

[1] Kärkkäinen, J. 2007. Fast BWT in small space by blockwise suffix sorting. *Theor. Comp. Sci.* 387, 3 Nov. 2007, 249-257
[2] Burkhardt, S. and Kärkkäinen, J., Fast lightweight suffix array construction and checking. In: LNCS, vol. 2676. Springer.
[3] A. Döring, D. Weese, T. Rausch, K. Reinert, SeqAn - An efficient, generic C++ library for sequence analysis, BMC Bioinformatics 2008, 9:11
[4] Manber, U. and Myers, G., Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing. v22 i5. 935-948.
[5] Larsson, N. J. and Sadakane, K. 2007. Faster suffix sorting. *Theor. Comput. Sci.* 387, 3 (Nov. 2007), 258-272.
[6] Ferragina, P. and Manzini, G. 2000. Opportunistic data structures with applications. In Proceedings of the 41st Annual Symposium on Foundations of Computer Science (November 12 - 14, 2000). FOCS. IEEE Computer Society, Washington, DC, 390.
[7] man getrusage
[8] man top
[9] Jon L. Bentley, Robert Sedgewick, Fast algorithms for sorting and searching strings, in: Proceedings of the eighth Annual ACM SIAM Symposium on Discrete Algorithms, 1997, pp. 360-369
[10] http://www.seqan.de/trac/ticket/20