Accelerating DNA Read Mapping and Alignment

Spring 2016

Sunny Nahar

April 17, 2016

Contents

1 Introduction

Mapper speed remains a bottleneck in DNA read mapping. Processing and aligning billions of reads takes a substantial amount of time, and there is a quest to build faster and better algorithms to improve runtime and throughput. Single processor compute capabilities have not improved significantly, and with the advent of multi-core and multi-threaded machines, parallelism is important in achieving the next generation of performance. Many aspects of DNA read mapping are inherently parallel, so this makes it a viable approach for achieving improvements in runtime.

2 Research Question and Goal

This project focuses on optimizing and parallelizing a component of DNA read mapping pipeline, seed selection. We want to determine how efficiently we can distribute work over cores to achieve a large speedup.

2.1 Machine Specs

The machine used for testing is a quad-socket 40 core machine with Intel Hyperthreading and 1000GB of RAM. It has a 64KB L1, 256KB L2, and 24MB L3.

3 HashTree

3.1 Introduction

The HashTree is a hashtable of suffix tries used to store the genome. It allows for searches with complexity L, where L is the length of the query. More specificly, a search requires L cache misses. To search for a given read in the HashTree, the first 10 base-pairs of the read are used to index into the hashtable, and the remaining base-pairs are used to index into the trie.

3.2 Greedy Scheduling and Dynamic Work Assignment

The HashTree is segmented into multiple files onto disk. This makes it simpler for multithreaded loads, as each thread can grab a file to load and operate on. The files encode multiple tries in the HashTree, and each trie is independent, so each thread can independent construct its respective trie.

Currently the HashTree is split into 256 parts. There is large variation in file and hence trie sizes, so it is important to distribute work equally among threads. This is done through OpenMP's dynamic scheduling and a initial greedy distribution of work. OpenMP creates a thread scheduler which dynamicly allocates work to available threads: whenever a thread is done with its task, the scheduler gives it the next available piece of work to do. However a problem arises if the sizes of the work in the list is not distributed evenly. For example, if the last piece of work requires a large compute time, then this will likely be sequentially running at the end, as other threads will not have any work to do.

To combat this, we first estimate the size of the work need to load each trie into memory. This is computed by using the file size. We sort the work list of tries from largest to smallest. Therefore the initial batch of work the threads receive are the largest pieces. This is within a factor of 2 to the optimal schedule of work to threads, per the greedy work scheduling algorithm.

3.3 Memory Mapped IO

Reading from disk is inherently slow. Reading roughly 80GB from disk is monumentally worse, especially since it must be sequentially. When reading from disk, we need each thread to read its file completely. This is to prevent disk thrashing, since if each thread only reads a smaller number of bytes from its file, then the disk will keep seeking between various files. Hence reading is sequential. In addition, it is highly inefficient to operate on data while its being read. In that case, we have essentially sequentialized the entire process. So data must be copied from the file to memory before it is used.

This takes significant time and disk bandwidth. A better approach is using Memory Mapped IO. The file is not copied to memory, but mapped to pages in the kernel page cache. Hence we do not need to copy it to user space. Therefore accessing pages is much more efficient. In addition, we can advise the kernel about the page accesses, which in this case is linear.

This allows the kernel to prefetch and optimize accordingly. Use memory mapped IO gives a significant boost, as the time to read the file is almost removed.

3.4 Memory Allocation

A critical component of generating the HashTree is allocation of memory. Using the traditional new operator causes thread stalls, as new contains a lock, and threads have to sequentially request memory. Constructing the HashTree is a memory intensive operation, with 250GB of memory being requested. Hence contention over memory allocation creates a significant drawback. A custom per thread memory allocator solves this problem. It requests a large chunk of memory using new, and facilitates smaller allocations through its own data structure. This allows for contention free memory access. In addition, free memory is much simpler, as only the large chunks need to be free, not the individual requested nodes. So the HashTree keeps a centralized pool of memory requested from all the allocators, which it frees

3.5 Speedup

The performance caps off at 16 threads, where the speedup is fairly close to linear. This is likely due to the overhead of TLB misses when multiple threads are memory mapping files.

Threads	Runtime (s)	Speedup
1	479.09	1.00
2	232.33	2.06
4	118.89	4.03
8	62.16	7.71
16	35.54	13.48
32	32.23	14.86
64	31.9	15.02
80	31.6	15.16

Table 1: Speedup of building the HashTree across 80 threads

3.6 Future Work

3.6.1 Minimizing Disk Thrashing through Reader Thread

Since there are multiple threads each reading from a respective file, the files are read in arbitrary order compared to the other stored on disk. Hence there is still some disk overhead due to the random access. A solution is to have one reader thread which performs the read operation and notifies other threads when the read has completed.

4 Efficient Space Representations

4.1 HashTree

The HashTree is implemented using TreeNode and LeafNode classes. These contain extra information used during creation from the genome, so their sizes are large: 48 bytes for the TreeNode and 64 bytes for the LeafNode. This implementation has a size of 297GB.

4.2 Compressed HashTree

For search queries, we only need to keep track of frequency. Therefore we can create a compressed tree which only encodes this information. Storing the frequency is only 4 bytes, but storing virtual function pointers and struct padding causes the TreeNode size to be 48 bytes and LeafNode 16 bytes. The TreeNode needs to store 4 pointers to each of the subtrees, so this is 32 out of the 48 bytes. This implementation has a size of 186GB.

4.2.1 Speedup

Threads	Runtime (s)	Improvement over HashTree
80	28.47	11%

Table 2: Compressed HashTree across 80 threads

4.3 Compressed HashTree (C structs)

Using structs, we can reduce the space to 4 bytes for LeafNodes and 40 bytes for the TreeNode, since we no longer virtual function pointers. This reduces the size to 147GB.

4.3.1 Speedup

Threads	Runtime (s)	Improvement over HashTree
80	26.12	21%

Table 3: Compressed Struct HashTree across 80 threads

4.4 Future Work

4.4.1 Bit level compression

We observe that the frequency requires at most 22 bits. Hence the extra 10 bits in the 4 byte integer used to store the frequency is wasted. This can potentially be used to store additional information to aid with compression. This is detailed below.

4.4.2 Reducing pointer space

Storing 4 8-byte pointers requires a lot of space. We detail a scheme which can reduce this to storing 1 pointer. Given a TreeNode, if we store all child nodes successively, then only the start pointer to the first node is required. The other 3 can be indexed using the first pointer. Therefore we need an 8-byte pointer to point to the start. We need to keep track of which children (ACGT) exist, so this requires 4 indicator bits. Hence to calculate the position of the child, we add the corresponding indicator bits and multiply by 8 to get the offset. However, the child node sizes are not uniform, since the LeafNode is 4 bytes, but the TreeNode is at least 16 bytes (8 bytes for pointer, 4 bytes for frequency and indicator bits, and 4 bytes for padding). So we also need to keep track of whether the child is a LeafNode or TreeNode. This takes an additional 4 bits. Hence we are able to use 16 bytes (with 4 bytes and 2 bits free) to store the TreeNode and 4 bytes for the LeafNode.

4.4.3 Succinct and Compact data structures

There has been considerable research in building very compact representations of binary trees which use 2-4 bits per node in addition to the data and still allow for a searchable representation. It is probably possible to reduce the pointer to perhaps an integer or less.

4.4.4 FM-Indexing and BWT

5 Bidirectional Frequency Predictor

5.1 Introduction

The Bidirectional Frequency Predictor is an important data structure used to offset the cost of accesses to the HashTree. It gives a rough estimate of the frequency of the in O(1) time, and 1 cache miss.

5.2 Building across the HashTree

It is built by traversing the HashTree and updating a global table at each iteration. Each trie of the HashTree can be traversed in parallel by a thread, so this makes the build operation highly parallel.

5.3 Synchronization

A simple method to update the global table would be to grab a lock or grab a per-element lock. Acquiring a lock can potentially take time and requires memory to store all the lock data structures. A simpler method is using OpenMP's atomic update. This essentially performs a CAS to update the table.

Since the table is huge and we have at most 80 threads concurrently modifying it, it is unlikely that two threads will access the same element in the same. Therefore the CAS should succeed in almost all cases.

It was observed that removing the atomic write improved the performance by 30% and still produced the correct table output, but this is still unsafe in the general case.

5.4 Speedup

The speedup is linear up to 16 threads, but slowly degrades to about half the performance at 80 threads. The use of atomic writes is a contributing factor, as removing the atomic requirement substantially increases speed, but at the cost of errors. However no errors were observed.

Threads	Runtime (s)	Speedup
1	2930	1.00
2	1470	2.00
4	727	4.03
8	364	8.07
16	190	15.42
32	109	26.88
64	82	35.73
80	71	41.27

Table 4: Speedup of building the Predictor across 80 threads

6 Seed Selection

6.1 Introduction

Seed selection is culmination of building the HashTree and the Bidirectional Frequency predictor. Given a large set of reads of DNA, the goal is to output a set of non-overlapping substrings (seeds) with the least overall frequency. We use the predictor and HashTree to query for frequency of substrings to help construct the set of seeds. Finding the optimal set of seeds is time-consuming, and it is done by the OSS seed selection algorithm. We want to reduce the runtime and output close to optimal set of seeds.

6.2 Parallelization

This problem is almost embarrassingly parallel. We can simply parallelize over reads in the set, as there are millions of such reads, instead of trying to parallelize the selection algorithms themselves, which do not exhibit much parallelism.

The seed selection algorithms access the HashTree and Predictor concurrently. Since these are static data structures, multiple threads can concurrently read the data. However there will be overhead due to cache coherence traffic from the reads propagating between caches which will bring down the overall performance.

In addition, the HashTree is large, so it is spread across multiple NUMA nodes. This implies that there will be effects due to nonuniform memory accesses. Threads located closer to the core whose memory node contains the HashTree will perform much faster than threads located faraway. So we will definitely expect to see an impact due to NUMA on the performance.

The frequency predictor is likely stored across 1 node, so threads on other nodes have higher memory latencies.

6.3 Speedup

Almost every selector was able to get high speedup across the multicore system. The Uniform Seed Selector performed poorly, mainly due to extremely low arithmetic intensity, so the overhead of multicore multithreading exceeded the benefits. The Centered Bidirectional Uniform Selector, which only accessed the frequency predictor, had a 43x speedup over 80 threads. The remaining seed selectors which accessed both the predictor and the HashTree had speedups between 60x-67x over 80 threads, which is reasonably good given the numerous limitations, most notably the NUMA architecture.

Threads	Runtime (s)	Speedup
1	0.647	1.00
2	1.078	0.60
4	0.751	0.86
8	0.132	5.26
16	0.321	2.02
32	0.279	2.32
64	0.396	1.63
80	0.648	1.00

Table 5: Speedup of Uniform Selector (4M read set)

Threads	Runtime (s)	Speedup
1	73.49	1.00
2	39.90	1.84
4	21.14	3.48
8	10.42	7.05
16	5.34	13.76
32	3.34	22.00
64	1.85	39.73
80	1.70	43.23

Table 6: Speedup of Centered Bidirectional Uniform Selector (4M read set)

Threads	Runtime (s)	Speedup
1	621.69	1.00
2	341.45	1.82
4	169.80	3.66
8	82.08	7.57
16	40.99	15.17
32	20.34	30.56
64	10.91	56.98
80	9.27	67.06

Table 7: Speedup of Exact Centered Bidirectional Uniform Selector (4M read set)

Threads	Runtime (s)	Speedup
1	80.85	1.00
2	40.61	1.99
4	21.36	3.79
8	10.94	7.39
16	6.08	13.30
32	4.92	16.43
64	1.59	50.85
80	1.33	60.79

Table 8: Speedup of Hobbes Selector (4M read set)

Threads	Runtime (s)	Speedup
1	107.69	1.00
2	54.21	1.98
4	39.09	2.75
8	13.17	8.18
16	6.72	16.02
32	3.44	31.31
64	2.09	51.53
80	1.78	60.50

Table 9: Speedup of Bidirectional Hobbes Selector (4M read set)

Threads	Runtime (s)	Speedup
1	190.36	1.00
2	94.70	2.01
4	53.85	3.54
8	23.37	8.15
16	11.77	16.17
32	6.22	30.60
64	3.85	49.44
80	3.17	60.05

Table 10: Speedup of Centered Bidirectional Hobbes Selector (4M read set)

Threads	Runtime (s)	Speedup
1	189.35	1.00
2	95.89	1.97
4	55.45	3.41
8	24.48	7.73
16	12.00	15.78
32	7.68	24.66
64	3.61	52.45
80	2.95	64.19

Table 11: Speedup of Exact Bidirectional Hobbes Selector (4M read set)

Threads	Runtime (s)	Speedup
1	605.79	1.00
2	358.56	1.69
4	169.77	3.57
8	81.20	7.46
16	40.60	14.92
32	20.63	29.36
64	11.23	53.84
80	9.30	65.14

Table 12: Speedup of Exact Centered Bidirectional Hobbes Selector (4M read set)

7 Conclusions

We were able to improve the runtime and effectively parallelize the entire Seed Selection pipeline, from building the HashTree, constructing the predictor, and performing Seed Selection using various algorithms.

There was a 15x improvement to loading the HashTree from disk. We described potential ways to decrease space and improve the runtime, further lowering the overhead of the HashTree load.

There was about a 42x improvement to constructing the Bidirectional Frequency predictor. We can build larger predictors much more efficiently and even on the fly, since the small build time likely offsets the cost of storing on disk.

We achieved a very high efficiency in parallelizing seed selection, averaging about a 60x speedup over 80 threads. There is room for further improvement if we analyze and optimize for NUMA accesses using thread affinity.

Reducing the runtime for seed selection speeds the overall mapping process, and allows for more complex, expensive, but accurate seed selection algorithms for the same cost as running basic algorithms sequentially.