Optimizing Smith-Waterman for Increased Throughput Using Traditional Multithreading APIs

Frank Nothaft, Karthika Periyathambi, Michel Aoun, Jaclyn Chen
Stanford University, Department of Electrical Engineering
{fnothaft,karthipd,jaclync,mcaoun}@stanford.edu

Abstract

Current implementations of the Smith-Waterman Sequence Alignment Algorithm use SIMD architectures and non-commodity hardware acceleration to achieve top throughput, but do not scale well to commodity parallel computers. This paper proposes an implementation of the Smith-Waterman Algorithm that demonstrates throughput of up to 588 billion cell updates per second (GCUPS) and over 65 GCUPS in typical cases using a commodity server and the POSIX thread library demonstrating an approximately 4x improvement in throughput over the current leading software implementation of Smith-Waterman.[1]

Introduction

In bioinformatics, the objective of DNA and protein sequencing is to explore the relationship according to the similarity between a query sequence and multiple subject sequences. Biologists and scientists are dedicated to developing methods to align two sequences. The most widely used algorithm is Smith –Waterman algorithm[23], which is one of the slowest sequence search algorithms but guarantees the best accuracy. Given two sequences of lengths m and n, the computation time is of O(mn), which is very time consuming. BLAST, among the other most competitive algorithms, works more efficiently but the alignment outcomes are not optimal.

In light of the huge amount of repetitive computations of Smith-Waterman algorithm, research effort has been put in speeding up the algorithm with both parallel architectures and hardware accelerators. Implementations of Smith-Waterman algorithm on FPGAs, Cell/Bes and GPUs demonstrated the opportunities for speed gain by utilizing more parallel computation-oriented architectures.[145] However, the parallelism used in these implementations is mostly on the diagonal elements in the matrix, which results in idle processors for the diagonals near boundary.

In order to utilize the parallelism more efficiently, we proposed two parallelization approaches: a) Thread-pool scheduling and b) Block-level parallelism. In Section 2, the operation Smith-Waterman algorithm is briefly addressed. An introduction of the two proposed parallelization approaches is presented in Section 3, followed by the detailed implementation of thread-pool scheduling and block-level parallelism in Section 4 and 5, respectively. The results are shown in Section 6, and we bring up the related work and draw our conclusions in the last two sections.

Smith-Waterman Algorithm

Given the query sequence a and subject sequence b of lengths m and n, respectively, the algorithm is based on the m by n matrix H with the

match/mismatch penalty function w(c, d). There are three steps to obtain the final optimally aligned sequences as follows.

a) Initialization

b) Matrix update by dynamic programming

$$H(i,j)=\max\left\{\begin{array}{ll}0 & \\ H(i-1,j-1)+w(a_i,b_j) & \text{Match/Mismatch}\\ H(i-1,j)+w(a_i,-) & \text{Deletion}\\ H(i,j-1)+w(-,b_j) & \text{Insertion}\end{array}\right\},\ 1\le i\le m, 1\le j\le n$$

c) Path tracking
Starting from the maximum value in matrix H(i,j), the path tracks back to one of the three elements from top to left at the positions (i,j-1), (i-1,j-1) and (i-1,j) that has the maximum value. The output with optimal alignment is updated according to the type of sequence relationship (match/mismatch, deletion or insertion) and the sequence data of a(i) and b(j). This process stops when reaching the boundary element or back at position (0,0).

Data Dependencies

As indicated in the previous paragraph, each element in H matrix is updated by the knowledge of three elements on the top left region. This data dependency prevents the algorithm from the straightforward parallelism on all elements, but on each 45 degree diagonal.

Diagonal parallelism has been clearly depicted in the figure below:

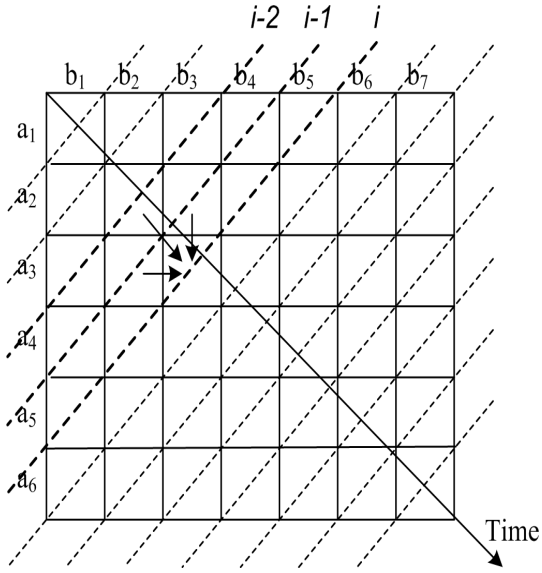


Figure 1: Diagonal parallelism depicted in Smith Waterman Algorithm

Sequential Trackback

In the tracking process, the next position depends entirely on the current position and the output sequences are updated only when the position is known. Therefore, path tracking is a completely sequential process and there is no efficient way for parallelism.

Parallelization Approaches

In order to enhance the efficiency in using parallelism, we proposed two approaches. The first approach is to create pools for threads to be utilized at any convenience, and the second one is to divide the data set into blocks.

Thread-Pool

Similar to the dynamic scheduling, thread-pool scheduling is to maintain three thread pools that represent three levels of readiness. Each thread in the ready pool is executed by any available processor, and the execution will update

the three threads that require the data of current position and move them to the next ready pool.

By using the thread-pool scheduling, we have the advantage of keeping all resources busy and thus reducing the total execution time. In the meantime, there must be synchronization mechanism of the thread pools for all processors. Also, the overhead of searching, retrieving, moving and deleting any thread in the pool needs to be taken into consideration. The choice of data structure becomes important in this case.

Blocked

Although Smith-Waterman can be scaled across multiple processors using the thread-pool methods described above, this particular implementation does not achieve throughput due to the amount of inter-process communication and scheduling that must occur. By migrating to a system that divides the matrix calculations into larger blocks, the scheduling overhead can be significantly reduced. SIMD instructions can be used within the blocks for further improvement of throughput.[6]

The implementation of Blocked Smith-Waterman also led to the optimization of the scheduling system to reduce queue lock contention and boost performance by 20%. These parallelization and lock-contention reduction techniques led to a 7x speedup on a 16-core commodity server.

Implementation of Thread-Pool Smith-Waterman

The first attempt at parallelizing the Smith waterman from its sequential version was to assign each row/ column to a Pthread.

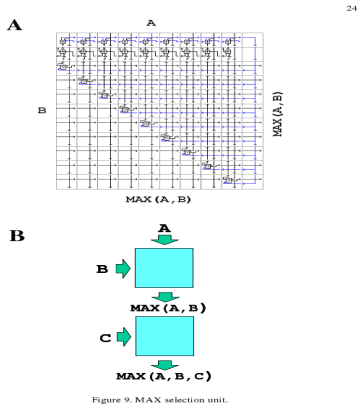


Figure 9. MAX selection unit.

Figure 2: First step towards parallelelism with static scheduling

This version was slower than the normalized sequential version owing to heavy locking between each row and the Pthreads remaining idle for one half of the iteration. Though pipelining could have been of use, still they did not benefit for the timings considered.

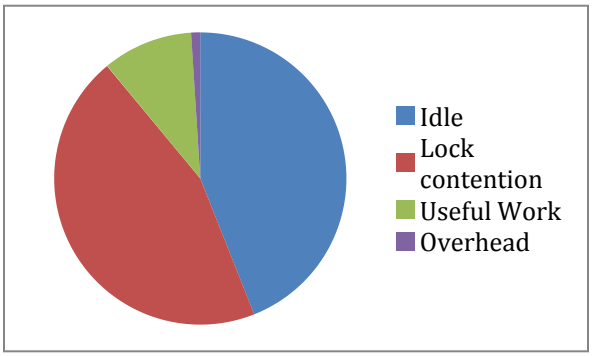


Figure 3: Fraction of time spent by a Pthread during its entire lifetime for a large query size of 1024.

This clearly shows that some improvement is needed in terms of dynamic scheduling over static scheduling. Hence, we move onto the

next approach that depicts thread pool and workers concept.

The next step is the presence of three maps/queues which contain the elements depending on the progress of their input entries.

Map1: only one of prev_hori / prev_vert / prev_diag is ready

Map2: any two of [prev_hori , prev_vert , prev_diag] are available

Queue: all three inputs are available

The workers pull in elements from the queue, apply the equations, calculate the logic, then update the other elements in the corresponding maps and promote them. To avoid contention , we used “Hashed Locks” which were hashed in the anti-diagonal fashion (i-j=constant) to minimize contention between the diagonally parallel algorithm. Furthermore, locking has to be in “Nested style” as one thread might pull the element while other pushes it elsewhere. This increased the overhead of locking tremendously.

Despite the hard work, there was effectively no speedup owing to the fact that computation/locking ratio was very poor as indicated in the figure 4 below. But the scenario isn’t that bad, as the fact that executing computation though around 11.8 % in parallel does offer a speed up which will be compared in overall timings. Despite that, thread pool version is slow owing the shift to blocked version to increase computation burden on the threads. The figure 4 compares single threaded version against 16 threads one.

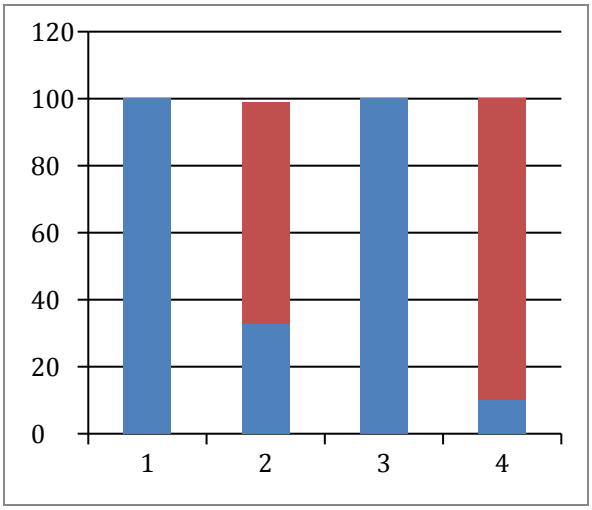


Figure 4: Comparison between sequential and thread pool version for a single Pthread independent of total execution time

Implementation of Blocked Smith-Waterman

In order to improve the efficiency of a parallel application, the application developer must strive to minimize the amount of time spent stalling due to communication and maximize the percent of time spent performing computation. While the Smith-Waterman algorithm can be statically scheduled in order to reduce scheduling complexity, static scheduling is inefficient, as processes must still communicate in order to check dependencies. Additionally, static scheduling frequently leads to load imbalances in a massively parallel system.

While the static scheduler profiled in the implementation of Thread-Pool Smith-Waterman did not provide performance improvements over the sequential version of the Smith-Waterman algorithm, we propose an extension that operates on square blocks of data from

the matrix, instead of single matrix entries. By performing more computation per scheduling decision, we are able to minimize the impact of communication latency and lock contention. Larger block sizes also allow for improved cache performance due to fewer false misses and improved computational throughput using SIMD instructions, as seen in previous high-performance implementations of the Smith-Waterman algorithms.

Implementation

Although adapting to a system where larger blocks are scheduled at a specific time reduces communication, communication still occurs in order to track the data dependencies present within the Smith-Waterman cell matrix. At application launch, all cells at the immediate left and top of the matrix are enqueued in the scheduler, and multiple threads are launched. Each thread repeatedly queries the scheduler until assigned a block to compute.

Once assigned a block by the scheduler, the receiving thread computes all matrix values within the assigned section of the matrix in a method equivalent to the sequential Smith-Waterman algorithm. Upon completion, the thread contacts the scheduler to either schedule the immediate right and lower neighbors of the completed block, or to declare that all blocks within the matrix have been computed. When all blocks have been computed, all threads finish, and the sequential trackback is completed.

Scheduler Implementation

Although a dynamically scheduled parallel application has the potential to greatly improve the balance of work between multiple worker threads, important tradeoffs have to be considered in order to minimize the impacts of lock contention and queue starvation. We propose a two-tier dynamic scheduler using multiple queues for the purpose of efficiently scheduling work for the parallel Blocked Smith-Waterman algorithm. The developed scheduler is shown in figure 5 below, and uses hashes and random variables in order to load balance between multiple queues, as well as a back-off-and-retry methodology to minimize lock contention within the scheduler.

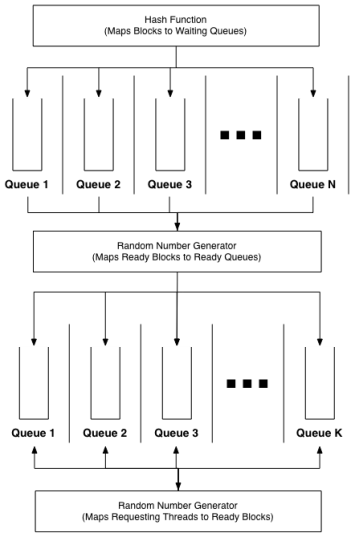


Figure 5: Queuing System for Blocked Smith-Waterman

Due to the need to monitor the data dependencies within the Smith-Waterman system, the scheduler must track all blocks and only schedule blocks for execution when they have been submitted for scheduling twice, necessitating the use of both waiting and ready queues. When a block is added to the scheduler, the scheduler must check to make certain that it has not already been enqueued by checking to see if an entry for the specific block

exists currently in any ready queues. Although this solution is correct, it is inefficient, as it requires the scheduler to either only have one ready queue or to check all ready queues. This solution can be streamlined through the use of a hash to determine the assignment of a block to a ready queue. Since we are not worried about collisions with our hash, we can use the simple hash ($x + y$) % N, where x and y are the block coordinates and N is the number of queues. If a block is not in the assigned ready queue, the lock for the queue is obtained, and the block is added; else, the lock is obtained and the block removed and scheduled in a ready queue.

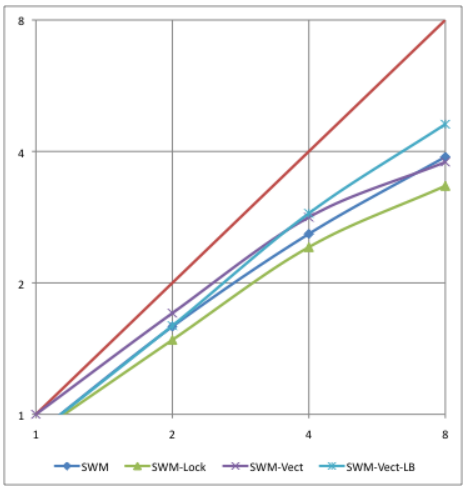
Within the ready queues, there are no data dependencies, allowing us to use multiple queues and a fully random scheduling discipline to reduce lock contention and maintain the appearance of one single ready queue to improve load balance. Requests to add or remove blocks from a queue are served to and from randomly chosen queues, which balances the requests across multiple queues, serving to reduce the likelihood of multiple scheduling operations causing mutual lock contention. Additionally, the use of pthread_mutex_trylock and queue reselection instead of pthread_mutex_lock led to a 7-15% performance improvement due to a significant decrease in lock contention.

SIMD Extensions

As noted above, the use of blocks of increased size allowed us to efficiently vectorize the Smith-Waterman algorithm using ICC. Although we did not see a six-fold improvement in execution time as seen in Rognes and Seeburg[7], we achieved a performance improvement of approximately ten percent over non-vectorized code. We believe that further optimization of the current implementation could lead to larger throughput improvements.

Results

We executed the Blocked Smith-Waterman application on both Hotbox, a server with support for eight hardware threads, and Octomom, a server with eight Intel Xeon processors supporting SSE4 and 12GB of RAM. The speedups achieved on Hotbox can be seen in graph 1, below.

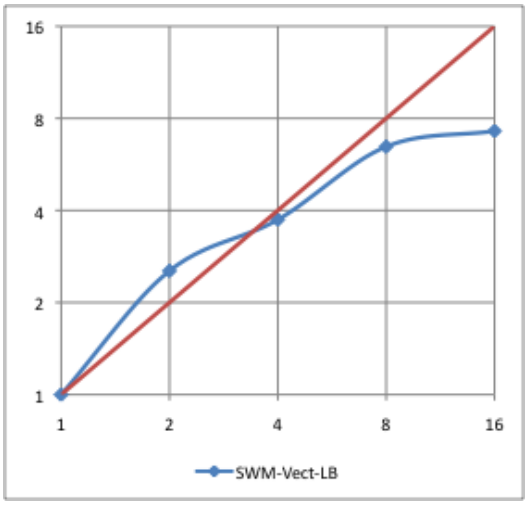


Graph 1: Speedup of Blocked Smith-Waterman on Hotbox

For testing, we ran the basic blocked Smith-Waterman algorithm (SWM), the blocked Smith-Waterman algorithm without lock back-off-and-retry (SWM-Lock), blocked Smith-Waterman with vector optimizations (SWM-Vect), and blocked Smith-Waterman with vector optimizations and large blocks (SWM-Vect-LB). Although the application has strong data dependencies that prevent the easy and direct parallelization of the application, large amounts of communication for scheduling, and a

significant sequential section (trackback), we still achieved significant speedup on Hotbox.

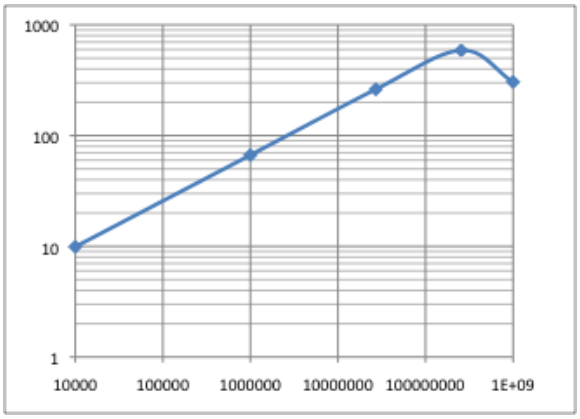
We achieved even better results on Octomom, as seen in graph 2 below. We attribute this largely to the increased amount of memory available and larger on-chip caches.



Graph 2: Speedup of Blocked Smith-Waterman on Octomom

While it was expected that the Blocked Smith-Waterman would allow for improved results, the superlinear speedup that was achieved at two processors was unexpected. We attribute this to improved caching of data as well as low communication latency and scheduling complexity between two threads.

While the speedup achieved by Blocked Smith-Waterman was impressive, the throughput achieved was a much more significant result, and can be seen below in graph 3.



Graph 3: Throughput of SWM-Vect-LB on Octomom (GCUPS)

We tested our implementation of the Blocked Smith-Waterman algorithm on matrices varying between 100x100 elements (10000 cells) and 32000x32000 elements (~10^9 cells) with 16 threads. Our application reached a peak throughput of 588 GCUPS on a 16000x16000 (256 million cells) matrix, and sustained throughputs over 250 GCUPS from 5200x5200 (27 million) cells to 32000x32000 (~10^9 cells) matricies.

Related Work

There is similar work going on the hardware side in a company named TimeLogic which is trying to provide hardware solutions for the same. Furthermore, intense research is going on in striped waterman related approaches[vi] . FPGA and ASIC implementations are focused on improving the speedup using hardwird coding in Verilog. More information can be seen in the references.

Conclusion

As discussed throughout the paper, Smith Waterman is of extreme importance owing to its high accuracy, but is extremely tough to parallelize. The various parallelized versions using thread pools and static scheduling fail owing to heavy locking overhead and poor computational intensity. On the other hand, the blocked vectorized version offers a hope owing to decent speedup, which when combined with embarrassingly parallel parts (implemented through SPMD clusters/GPUs) can revolutionize the bio-gene world. Hence, our novel proposal of blocked version of Smith Waterman with heavier chunks can accelerate the mutation location and benefit the gene research.

Acknowledgements

We are highly grateful to Professor Kunle Olukotun for offering us this wonderful chance to work on a research project and learn more about parallelism related techniques. Also, we would like to thank Lawrence McAfee for his support and help in setting up the project requirements.

[1] Michael Farrar, "Optimizing Smith-Waterman for the Cell Broadband Engine," White Paper, 2008.

[2] Smith T and Waterman M, "Identification of Common Molecular Subsequences," J Molecular Biology, 1981.

[3] Gotoh O, "An improved algorithm for matching biological sequences," J Mol Biol, 1982.

[4] Jeff Allred, Jack Coyne, Joseph Grecco, Joel Morrissette, "Optimized Smith-Waterman Implementation on a FSB-FPGA module using the Intel Accelerator Abstraction Layer," Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, 2009.

[5] Peiheng Zhang, Guangming Tan, Guang R. Gao, "Implementation of the Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform," Conference on High Performance Networking and Computing, 2007.

[6] Michael Farrar. "Striped Smith-Waterman speeds database searches six times over other SIMD implementations," Oxford Journal on Bioinformatics, V. 23, n. 2.

[7] Torbjørn Rognes, Erling Seeberg. "Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors," Oxford Journal on Bioinformatics, V. 16, n. 8.