A PAIRWISE COMPARISON OF DNA SEQUENCE ALIGNMENT USING AN OPENMP IMPLEMENTATION OF THE SWAMP PARALLEL SMITH-WATERMAN ALGORITHM

A thesis submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Masters of Computer Science

by
Tristan Cuevas

May, 2015

© Copyright
All rights reserved
Except for previously published materials
Thesis written by

Tristan Cuevas
B.S., Kent State University, 2009
M.S., Kent State University, 2015

Approved by

______________________________
Johnnie W. Baker, Professor Emeritus, Ph.D., Computer Science, Masters Advisor

______________________________
Javed Khan, Professor, Ph.D., Chair, Department of Computer Science

______________________________
James L. Blank, Dean, Ph.D., Dean, College of Arts and Sciences
# TABLE OF CONTENTS

LIST OF FIGURES .................................................................................. V

DEDICATION.......................................................................................... VI

ACKNOWLEDGEMENTS ........................................................................ VII

CHAPTER 1 INTRODUCTION .................................................................. 1

CHAPTER 2 BACKGROUND AND RELATED WORK ............................... 3
  2.1 Smith- Waterman algorithm background .................................... 3
  2.2 ClearSpeed background ............................................................... 5

CHAPTER 3 IMPLEMENTATION ............................................................... 7
  3.1 Smith-Waterman Implementation ............................................... 7
  3.2 SWAMP+ Extension ................................................................. 11
  3.3 Implementation Challenges ..................................................... 12

CHAPTER 4 EXPERIMENT ................................................................. 15
  4.1 Testing Environment ................................................................. 15
  4.2 Parameters ............................................................................. 15
  4.3 Test Data .............................................................................. 16
  4.4 Algorithm Correctness ............................................................. 16

CHAPTER 5 RESULTS .......................................................................... 18
  5.1 Results ..................................................................................... 18

CHAPTER 6 DISCUSSION AND FUTURE WORK ................................. 25
  6.1 Summary of Work .................................................................... 25
  6.2 Future Work. ......................................................................... 25
LIST OF FIGURES

Figure 1 Smith-Waterman Equations................................................................. 4
Figure 2 Matrix wavefront traversal ................................................................. 5
Figure 3 Fasta Example ....................................................................................... 8
Figure 4 Diagonal Wavefront Traversal .............................................................. 8
Figure 5 Scoring Matrix ..................................................................................... 10
Figure 6 Traceback Matrix ................................................................................ 10
Figure 7 Traceback Pairings ............................................................................. 10
Figure 8 Matrix Class ....................................................................................... 13
Figure 9 Parallel Race Condition Example ....................................................... 17
Figure 10 Speedup of the OpenMP stack implementation ................................. 19
Figure 11 Speedup of the OpenMP heap implementation .................................. 19
Figure 12 Original OpenMP Bar Graph ............................................................. 20
Figure 13 Original OpenMP/ClearSpeed Comparison ......................................... 21
Figure 14 Internal Run Time of the OpenMP Heap Implementation .................. 23
Figure 15 OpenMP Heap Implementation Run Times in Microseconds .......... 24
DEDICATION

I would like to dedicate this thesis to my wife Brittany and my two daughters Emma and Addsion.
ACKNOWLEDGEMENTS

I want to thank my thesis advisors Dr. Baker and Dr. Steinfadt for their help, insight, direction, and support through this process. Dr. Steinfadt has dedicated many nights to meetings to discuss this work and I want to thank her for that. I would also like to thank the other committee members, Dr. Bansal and Dr. Ruttan, for making themselves available for the thesis defense and taking the time to read over and evaluate my thesis. Lastly I would like to thank Marcy Curtiss for taking the time to work with and advise me on many decisions throughout my master’s program.

Tristan Cuevas

04/06/2015, Kent, Ohio
CHAPTER 1

Introduction

With the increasing interest in sequence alignment and approximate string matching to solve problems like DNA sequencing [Steinfadt10, Junid08], text searching [Esenwein97], natural language processing [Salehi13], and plagiarism detection [Irving04], there is a demand for faster, more efficient solutions to solve these problems. Use of parallel computing is one way to accelerate the sequence matching algorithms. Parallel computing that was historically done on large single instruction, multiple data (SIMD) and multiple instruction, multiple data (MIMD) machines can now be run on common-off-the-shelf (COTS) machines with multiple core processors to distribute the workload.

This Smith-Waterman using OpenMP (SWOMP) work stems off of the previous parallel sequence alignment work Smith-Waterman using Associative Massive Parallelism or SWAMP research [Steinfadt10]. That work first implemented the Smith-Waterman algorithm on the ASC model [Potter94] and then later on the ClearSpeed CSX600 accelerator. Then was extended further into SWAMP+ [Steinfadt12] by improving performance and adding unique extensions to the Smith-Waterman algorithm to handle multiple subsequence alignments. The initial motivation for this work involved implementing an OpenMP shared memory model version of Dr. Steinfadt’s extended Smith-Waterman algorithm in hopes of being able to show the potential of a theoretical larger ClearSpeed board that does not exist today. ClearSpeed has since then declined as a leading edge company in the development of SIMD acceleration boards. They haven't released a new board since 2008 and have discontinued distribution of the Cn Compiler that is used for development on their boards.
The first version of our OpenMP implementation was developed over the summer of 2013. We then submitted a paper to the 2014 Workshop on Large-Scale Parallel Processing or LSPP that compared the timings we recorded with this version of the application against the ClearSpeed timings from Dr. Steinfadt’s earlier work. With the feedback we received from the LSPP committee we had some new directions for the application. ClearSpeed was not seen as a company that was going to be continuing to produce new SIMD boards and we had limitations on the problem size our algorithm could handle at the time. With the feedback from LSPP we decided to work on improving the OpenMP implementation to be able to handle much larger problem sizes. This will allow us to show how the running time of our implementation scales as the problem size increases on a broader range of problem sizes.

In Chapter 2, the background and related work for the Smith-Waterman algorithm and the ClearSpeed accelerator are discussed. In Chapter 3 the implementation details of the algorithm are given. In Chapter 4 we describe the testing environment including hardware, test parameters, the data we used in our tests, confirming that the algorithm is working correctly, and we discuss the results of our tests. In Chapter 5 we go over the conclusion and we explore future work possibilities.
CHAPTER 2

Background and related work

2.1 Smith- Waterman algorithm background

The Smith-Waterman algorithm is very important in the field of bioinformatics because given two strings of RNA, DNA, or amino acids, it will find the best local alignment that is possible between them. Finding all of the possible alignments is a very computation intensive process. Given the broad applicability of the Smith-Waterman algorithm as a “gold-standard” of alignment as well as its usefulness for pairwise, detailed alignments without heuristics, and its use as a benchmark in parallel bioinformatics, there are many implementations to speed up the Smith-Waterman algorithm. Some examples of other parallel implementations include ClearSpeed [Steinfadt10,12,13], FastFlow [Aldinucci10], FPGA’s [Junid08], GPU using CUDA [Ligowski09, Manavski08], OpenMP [Noorian09], MPI [Noorian09], and Hybrid OpenMP + MPI in SMP clusters [Noorian09].

This work uses the Smith-Waterman algorithm and its SWAMP [Steinfadt13] extensions as a benchmark to show the potential for the larger scale ClearSpeed or other SIMD accelerator/computer.

The Smith-Waterman algorithm is a greedy algorithm that computes a matrix to determine a score for every possible alignment, including all possible insertions and deletions of a character within the alignment. Each cell has a data dependency on its Northwest, North, and West neighboring cells. Due to the data dependencies, computation along a diagonal wavefront can be used to parallelize the algorithm. This matrix generation step that allows parallelization is
what makes it a popular algorithm of bioinformatics and HPC benchmarks. While diagonally traversing the matrix, each cell’s score $C_{i,j}$ is calculated as the max of $D_{i,j}$ a deletion, $I_{i,j}$ an insertion, $C_{i-1,j-1} + d(S1_i, S2_j)$ a match or miss, and 0. Where “$\sigma$” is the gap extension cost and “$g$“ is the gap insertion cost.

$$D_{i,j} = \max \left\{ C_{i-1,j} - g, D_{i-1,j} \right\} - \sigma$$

$$I_{i,j} = \max \left\{ C_{i,j-1} - g, I_{i,j-1} \right\} - \sigma$$

$$C_{i,j} = \max \left\{ D_{i,j}, I_{i,j}, C_{i-1,j-1} + d(S1_i, S2_j), 0 \right\}$$

$$d(S1_i, S2_j) = \begin{cases} \text{match\_cost} & \text{if } S1_i = S2_j \\ \text{miss\_cost} & \text{if } S1_i \neq S2_j \end{cases}$$

Figure 1 Smith-Waterman Equations.

Figure 2 shows an example of a Smith-Waterman scoring matrix, the wavefront diagonal traversal, and the data dependencies on the North West, North, and West neighbors in calculating each cell’s score.
2.2 ClearSpeed background

The ClearSpeed CSX600 accelerator [ClearSpeed14] that we are using for our baseline is a SIMD-like accelerator board. It uses a distributed memory model and has a ring network topology that can be used to quickly pass data between neighboring processing elements (PEs). The CSX600 has 96 PEs in a chip that all execute a single instruction stream in parallel on their own dataset.

The SWAMP algorithm that we are comparing uses an associative SIMD library ASC on top of the ClearSpeed language to bring associative functions [Potter94] to the ClearSpeed accelerator.

Figure 2 Matrix wavefront traversal.
The SWAMP algorithm is written in Cn which is a parallel C language extension library that ships with the ClearSpeed accelerator. The Cn language adds poly variables to the C programming language. A poly variable is a variable where each PE has its own instance of in that PE’s local memory. When an instruction is executed on a poly variable, each active PE will run the instruction on its own instance of the variable in parallel. For example if you created a poly char each PE would have its own copy of that char within its local memory. The char in each PE could have its own unique value and be operated on in parallel across all PE’s or a subset of PE’s. Dr. Steinfadt used this architecture to create the SWAMP and SWAMP+ parallel implementations of the Smith-Waterman Algorithm.
CHAPTER 3

Implementation

3.1 Smith-Waterman Implementation

Our OpenMP [Park01] Smith-Waterman implementation was written in C++ using Microsoft Visual Studio 2013 and OpenMP 2.0 language extension. The SWOMP application reads in a data file that stores the input in FASTA format (as shown in Figure 3). FASTA is an industry standard format for representing RNA, DNA, and amino acids. All new sequences start with a ‘>’ symbol. The first word after the ‘>’ symbol is known as the identifier and the rest of the line is used as a description. All lines after that are considered part of the sequence until another line starting with a ‘>’ symbol is found, which starts the next sequence in the file. For our experiment we compared the first two FASTA strings in the file but the application can be configured to compare the first string in the file to all other strings in the file.
Our algorithm traverses the matrix with a diagonal wavefront traversal (as shown in Figure 4). This is done because of the natural dependency of each cell on its Northwest, West, and North neighbors.

In this method of traversal, each diagonal is isolated in the outer loop. Then an inner loop traverses the cells in that diagonal. By computing the diagonals in order from top-left to bottom-right we guarantee that the dependencies for each cell will already have been computed in past iterations of the outer loop. This allows us to process every cell in a given diagonal in parallel.

To parallelize the inner loop of our traversal we used OpenMP. First we must tell OpenMP how many threads we want to use to parallelize a section of code. To do this we call the OpenMP function “omp_set_num_threads()”, passing it the desired number of threads. Then we must tell it what variables will be shared between those threads. To do this we call the OpenMP
function “omp parallel shared()”, passing each variable that will be shared between the threads. Lastly we use the OpenMP “omp for” pragma to wrap our sequential loop. The OpenMP “omp for” call parallelizes the for-loop that it wraps. It handles task scheduling and splits up the iterations of the for-loop to be worked on by multiple threads in parallel.

In each iteration of the inner loop these steps are taken to calculate the scoring and log the routes for traceback.

a. Calculate score coming from the Northwest neighbor.
b. Calculate score coming from the West neighbor.
c. Calculate score coming from the North neighbor.
d. Store North and West scores for later use in gap extension calculations.
e. Store the max cell score comparing the Northwest, West, and North scores.
f. If the max score stored for the cell is < 0 replace it with a value of 0.
g. Store the traceback route based on the max score stored.
h. Check if this cells score is greater than the global max score. If it is set it as the new global max score.

After the whole matrix is traversed and every cell has its score calculated, the saved value for global max is located and a traceback is done by backtracking from that cell along the path that led to that score until a cell with a score of 0 is encountered. The application will then output its findings in several formats including a scoring matrix (Figure 5), a traceback matrix (Figure 6), and a traceback pairing (Figure 7)
Figure 5 Scoring Matrix

Figure 6 Traceback Matrix

Figure 7 Traceback Pairings
3.2 SWAMP+ Extension

Now that we have implemented the base Smith-Waterman algorithm with traceback, we will look at the Multiple Subsequence Alignment feature that was first implemented as part of SWAMP+ [Streinfadt12]. It takes in a parameter for the number of alignments to find and a parameter for the degrade factor to prevent searching for scores that are vastly lower than the original best score. It uses the number of alignments parameter to continue searching for additional non-intersecting alignments after the best alignment is found. When the best local alignment is found the algorithm will block out the characters in the query and database strings that were part of that alignment by changing all instances of the characters used in the query string to ‘Z’ and all instances of the characters used in the database string to ‘O’. It then searches again in the remaining parts of the query and database strings to look for another best local alignment in the remaining parts of the query and database strings. It continues this until the number of alignments that the applications was configured for are achieved or until a score that is lower than the original best score times the degrade factor parameter is reached.

Since our OpenMP implementation of the Smith-Waterman algorithm is based on the SWAMP+ algorithm that the multiple subsequence alignment feature was developed for, this is the only OpenMP implementation of the Smith-Waterman algorithm that has implemented this feature.
3.3 Implementation Challenges

During development we faced several challenges including design modifications that were necessary to use the OpenMP shared memory model, task scheduling, and memory limitations.

The first challenge was that this algorithm was based on code written and optimized for the distributed memory model and swazzle network used by the ClearSpeed CSX600 board we described earlier. The code had to be modified for the shared memory model used by the OpenMP implementation. Much of the code had to be written very differently because of the different memory architectures. For example the matrices used to store data, all instances of writing or reading data, and the diagonal wavefront traversal were all major parts of the algorithm that had to be rewritten to optimally use the shared memory architecture.

Next the data structure that the data was stored in had to be changed to reflect our new memory model. In the distributed memory model, the data was stored across multiple PE’s which was not a good fit for our shared memory model. We modified it to be stored in matrix array variables. This matrix data structure used the stack to store the scoring matrices that was in finding the best possible alignment. What we discovered is that the stack is very limited when working with values of this scale. With the stack data structure we were limited to running comparisons with string sizes of up to 256 characters. Any larger and we would overflow the stack and crash. After receiving the feedback from our submission to LSPP it was clear that this was a problem that needed to be addressed. To solve this problem a Matrix class was created to initialize the matrix in heap memory (Shown in Figure 8). This new implementation allowed us to extend our max problem size to reach comparisons of up to 8192 characters.
Task scheduling was also a challenge we encountered implementing this algorithm in OpenMP. The original implementation of this algorithm had a custom written task scheduler that would divide the cells in a given diagonal by the number of threads available and assign the tasks in N chunks where N is the number of threads. In the case where the number of chunks does not evenly divide the amount of work the 0\textsuperscript{th} thread would take on the remainder in addition to its assigned chunk. Then the “#pragma omp parallel” call was used for each thread to process its chunk. During testing it was found that this scheduler worked correctly for comparisons where the query string length was equal to the database string length e.g. when the matrix was square. But it was miscalculating values for cases where the strings being compared were not of equivalent length. To remedy this problem we restructured our loop to use the built in OpenMP task scheduler. This uses the “#pragma omp for” instead of “#pragma omp parallel” to automatically split the work load of the iterations of the loop between multiple threads. This method is optimized for OpenMP parallelization. The original task scheduler was designed to
emulate a number of PE’s and was trying to force the ClearSpeed architecture. Using the “#pragma omp for” along with the “#pragma omp shared” takes full advantage of the shared memory scope and is a better implementation for this architecture.

The last problem encountered was that our initial test was to compare two strings both of length 96 and, as will be seen in our results, at this problem size the scheduling overhead offsets the speedup gained by OpenMP’s multithreading and it appeared that the OpenMP only created one thread because of its relatively equal running time to the sequential version of the algorithm. After much additional testing and confirmation that the OpenMP task scheduler was indeed breaking up the work, we decided to try a larger problem size and discovered that it was indeed just the overhead of the scheduler that made it appear to not be running in parallel.
CHAPTER 4

Experiment

4.1 Testing Environment.

Our OpenMP application was run on a Windows 8 computer containing an Intel Core i7-3600QM CPU @ 2.6GHz and 16.0 GB of RAM. The compiled application was run using a batch script with the “start /realtime” command to start the process with real time priority. Running the application with real time priority makes sure that the OS does not get priority to the system threads and gives our application the best possible performance running in Windows.

4.2 Parameters

Our application takes several parameters that it uses when it runs. These are broken down into system parameters for the run time environment and Smith-Waterman sequence alignment parameters. The system thread parameters are as follows. Thread count sets the number of threads to run the application with; Simulation times sets the number of times the application will run before averaging the output timings; And maximum string length sets the systems max string size that it can take as input and is used to initialize a large enough matrix to handle the input. Our original limit for max string size was 256 due to stack limitations. After discovering that limit, we improved the algorithm by moving the arrays into heap memory. This allows us to run the algorithm on much larger string sizes. Our new limit is strings of up to 8,192 characters.
The Smith-Waterman parameters are as follows: Match is the score that is added when a match is found; Miss is the score that is subtracted when two characters do not match; Gap Insert is the score that is subtracted when starting a new gap; Gap Extend is the score that is subtracted both when a new gap is opened and when a gap is extended; Number of alignments is the number of alignments to attempt for multiple subsequence alignments; And degrade factor is the percentage of the best score to stop searching after if the application is running multiple subsequence alignments.

4.3 Test Data

The test data that we used for our experiment consisted of two strings of length $n$ where $n$ is our problem size. Each smaller problem size string is a subset of the next larger problem size. The data is input in FASTA file format. Only the first two strings in the file are compared. We ran the application comparing the two strings using the Smith-Waterman algorithm and recorded the running time using the built-in C++ chrono library.

4.4 Algorithm Correctness

To confirm that the algorithm was producing the correct results we tested the same data sets against a known working Python version of the Smith-Waterman algorithm. We did this for each of the different thread counts we ran to assure that the multi-threaded algorithm produced the same results as the sequential algorithm. There were rare cases where the parallel version of the algorithm we implemented did not produce the exact same result as the sequential python version of the algorithm. After much testing it was determined that these inconsistencies were
due to a race condition on any given diagonal. If two cells on the same diagonal have the same global best score then the parallel version of the algorithm will select whichever cell happens to be processed first (As seen in Figure 9). Because we are processing all cells in a given diagonal in parallel there is no way for us to predict which cell will be processed first. Where in the sequential version of the algorithm the cells are processed in a predetermined order and will always execute in that order. This is a known risk with parallelizing the Smith-Waterman algorithm and is widely considered an acceptable risk as it is a rare case and the best value is still selected.

<table>
<thead>
<tr>
<th></th>
<th>@</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>C</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>@</td>
<td>@</td>
<td>@</td>
<td>@</td>
<td>@</td>
<td>@</td>
<td>@</td>
<td>@</td>
<td>@</td>
</tr>
<tr>
<td>A</td>
<td>@</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>@</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>@</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>@</td>
<td>5</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>@</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>3</td>
<td>15</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>@</td>
<td>5</td>
<td>0</td>
<td>15</td>
<td>3</td>
<td>20</td>
<td>8</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>C</td>
<td>@</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>20</td>
<td>8</td>
<td>25</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>A</td>
<td>@</td>
<td>5</td>
<td>0</td>
<td>15</td>
<td>8</td>
<td>25</td>
<td>13</td>
<td>21</td>
<td>18</td>
</tr>
</tbody>
</table>

**Figure 9 Parallel Race Condition Example**
CHAPTER 5

Results

5.1 Results

We originally ran the Smith-Waterman algorithm on a problem size of powers of two from 64 to 256. We also ran the algorithm with problem sizes 96 and 192 because that is the number of PE’s that the two ClearSpeed accelerators available. After the algorithm was improved to handle larger problem sizes we re-ran the algorithm on the original problem sizes and expanded on powers of two up to 8,192.

Figures 10 and 11 show the speedup we achieved with our OpenMP implementations. The low speedups values for small problem sizes is due to the overhead of creating the threads and the task scheduler dividing the work. At very small problem sizes we even see a negative speedup. But as our problem size grows we start to get a good speedup at problem sizes of 512 characters or more.
Figures 12 and 13 show our original stack implementation of the algorithm running on one, two, and four threads over the different problem sizes. The inner loop timings are the timings for just the matrix calculations for that run. Taking the timings on the inner loop allow us
to see the running time of the algorithm without the overhead of memory initialization. Note that for very small problem sizes that the algorithm actually runs faster on a single thread. This is once again due to the scheduling overhead suffered by shared memory parallel computation. As the problem size increases the runs with more threads become more efficient and quickly overtake the single thread timings.

Figure 12 Original OpenMP Bar Graph
Figure 13 shows the comparison of our original OpenMP implementation to the ClearSpeed solution. And while at this small problem size the OpenMP implementation has better timings, we predict that as the problem size grows to a more real world scale the Associative SIMD ClearSpeed implementation will overtake it and have the best performance of the grouping.

The second set of results we gathered from our improved algorithm show that a larger datasets can be run on an OpenMP solution. But initializing the memory in heap comes at a cost
of a large constant in the running time. This is only an issue at small problem sizes since as the problem size increases, the running time becomes large enough to make the cost of initializing the array in the heap negligible.

<table>
<thead>
<tr>
<th></th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>538,473</td>
<td>538,203</td>
<td>559,977</td>
<td>561,606</td>
</tr>
<tr>
<td>64i</td>
<td>939</td>
<td>1,745</td>
<td>1,050</td>
<td>2,031</td>
</tr>
<tr>
<td>96</td>
<td>537,862</td>
<td>539,228</td>
<td>534,099</td>
<td>550,288</td>
</tr>
<tr>
<td>96i</td>
<td>1,093</td>
<td>2,483</td>
<td>1,021</td>
<td>1,434</td>
</tr>
<tr>
<td>196</td>
<td>532,933</td>
<td>581,016</td>
<td>536,861</td>
<td>578,521</td>
</tr>
<tr>
<td>196i</td>
<td>3,749</td>
<td>1,438</td>
<td>1,750</td>
<td>1,867</td>
</tr>
<tr>
<td>256</td>
<td>539,547</td>
<td>553,659</td>
<td>545,075</td>
<td>528,336</td>
</tr>
<tr>
<td>256i</td>
<td>4,936</td>
<td>3,765</td>
<td>1,985</td>
<td>4,853</td>
</tr>
<tr>
<td>512</td>
<td>577,654</td>
<td>565,599</td>
<td>584,090</td>
<td>559,357</td>
</tr>
<tr>
<td>512i</td>
<td>23,251</td>
<td>11,599</td>
<td>7,175</td>
<td>6,646</td>
</tr>
<tr>
<td>1024</td>
<td>652,850</td>
<td>599,683</td>
<td>573,670</td>
<td>580,995</td>
</tr>
<tr>
<td>1024i</td>
<td>101,232</td>
<td>53,500</td>
<td>35,325</td>
<td>29,149</td>
</tr>
<tr>
<td>2048</td>
<td>989,845</td>
<td>763,391</td>
<td>700,780</td>
<td>705,623</td>
</tr>
<tr>
<td>2048i</td>
<td>419,388</td>
<td>226,624</td>
<td>171,528</td>
<td>136,796</td>
</tr>
<tr>
<td>4096</td>
<td>2,227,663</td>
<td>1,439,040</td>
<td>1,265,800</td>
<td>1,050,160</td>
</tr>
<tr>
<td>4096i</td>
<td>1,680,876</td>
<td>910,736</td>
<td>717,994</td>
<td>530,338</td>
</tr>
<tr>
<td>8192</td>
<td>8,797,632</td>
<td>4,582,761</td>
<td>3,607,464</td>
<td>3,119,630</td>
</tr>
<tr>
<td>8192i</td>
<td>8,240,088</td>
<td>4,031,327</td>
<td>3,075,432</td>
<td>2,564,127</td>
</tr>
</tbody>
</table>

Table 1 Improved Algorithm Timings in Microseconds

Table 1 shows the timings from the improved OpenMP algorithm using the heap implementation. The timings are in microseconds and the rows with an ‘i’ suffix are to mark timings that exclude the memory initialization. This data shows a rough constant of about 500,000 microseconds for memory initialization. The constant becomes smaller in comparison to
the running time the larger the problem size grows. The improved algorithm was also run on 8 threads to see if hyper threading had any effect on performance. What we found was that there is an increase in performance but it is not as significant as the increase in performance with the true core to thread ratio.

![Internal Thread Performance](image)

**Figure 14 Internal Run Time of the OpenMP Heap Implementation**

Figure 14 shows that the running time of the internal algorithm is higher than that of the original stack implementation (Figure 12). We believe this to be caused by the use of heap memory instead of the stack memory. The higher performance of the original algorithm is hindered by the problem size limit of 256 character strings. So the later algorithm is still a better choice in the long run unless you are applying it to a very specific problem that has a small string size throughout.
Figure 15 shows the running times we captured against the full range of problem sizes. With this version of the algorithm we were able to increase the maximum problem size we could handle from 256 characters to 8192 characters. With this broader range we are able to clearly see that the running time of the application is growing exponentially as the problem size is increasing. Because of this we can show that a theoretical larger ClearSpeed machine would in fact overtake the OpenMP implementation of this algorithm as Dr. Steinfadt has shown the SWAMP+ algorithm runs in O(m) time with a small constant of 3 when m = n [Steinfadt12].
CHAPTER 6

Discussion and Future Work

6.1 Summary of Work.

In summary we developed an OpenMP implementation of the extended Smith-Waterman algorithm based on the ClearSpeed SWAMP+ implementation. We overcame several development challenges including a shift in architectures that required a much different development approach, memory limitations using a stack implementation, and handling task scheduling to parallelize the work of the algorithm. We implemented the multiple subsequence alignments unique to the ClearSpeed SWAMP+ algorithm in OpenMP making it the only OpenMP implementation to include this feature. And lastly we showed that the OpenMP implementation’s running time is increasing exponentially as the problem size grows. This shows that the SWAMP+ implemented on a larger ClearSpeed or other SIMD computer would quickly overtake the OpenMP implementation as the problem size grows.

6.2 Future Work.

This work is the initial proof of concept work. In the future the baseline could be expanded by taking SWAMP timings using the newer CSX700 series accelerator. The OpenMP algorithm could also be improved by experimenting with dynamic scheduling, fine tuning the scheduling chunk size, and comparing how different chunk sizes perform at different problem sizes. We saw the impact of the scheduling overhead especially on small problem sizes. The task scheduling could be improved by moving the parallel code wrapper outside of the outer loop. This would allow us to generate the threads and do task scheduling once instead of at every
diagonal of the matrix and would reduce threading and task scheduling overhead. We also believe that larger problem sizes could be achieved by splitting the problem into smaller more manageable pieces. Those pieces could be processed separately and then the results could be stitched back together. Once the OpenMP algorithm has been optimized, it could also be used in an MPI + OpenMP hybrid solution that would allow it to tackle even larger problem sizes and compare those results to our OpenMP and ClearSpeed solutions. Lastly, a GPU implementation of the Smith-Waterman algorithm could be used to compare to the ClearSpeed, OpenMP, and MPI + OpenMP solutions.
REFERENCES


Software for High-End Computing (P2S2) held in conjunction with ICPP 2012: The 41st International Conference on Parallel Processing, Pittsburgh, Pennsylvania, September 10, 2012.
