# SWAMP: Smith-Waterman using Associative Massive Parallelism

Shannon Steinfadt Dr. Johnnie W. Baker
*Department of Computer Science, Kent State University, Kent, Ohio 44242 USA*
*ssteinfa@cs.kent.edu* *jbaker@cs.kent.edu*

## Abstract

*One of the most commonly used tools by computational biologists is some form of sequence alignment. Heuristic alignment algorithms developed for speed and their multiple results such as BLAST [1] and FASTA [2] are not a total replacement for the more rigorous but slower algorithms like Smith-Waterman [3]. The different techniques complement one another. A heuristic can filter dissimilar sequences from a large database such as GenBank [4] and the Smith-Waterman algorithm performs more detailed, in-depth alignment in a way not adequately handled by heuristic methods.*

*An associative parallel Smith-Waterman algorithm has been improved and further parallelized. Analysis between different algorithms, different types of file input, and different input sizes have been performed and are reported here. The newly developed associative algorithm reduces the running time for rigorous pairwise local sequence alignment.*

Index Terms—*Associative computing, SIMD computing, Parallel algorithms, Sequence alignment*

## 1. Introduction

A strong alignment between two sequences indicates sequence similarity. Similarity between a novel sequence and a studied sequence or gene reveals clues about the evolution, structure, and function of the novel sequence via the characterized sequence or gene.

Proposed in 1970, the Needleman-Wunsch algorithm [5] attempts to globally align one entire sequence against another using dynamic programming. A variation by Smith and Waterman allows for local alignment [3]. Local alignment does not force entire sequences to be positioned against one another. Instead it tries to find local region similarity, or sub-sequence homology. A minor adjustment by Gotoh [6] greatly improved the running time from $O(m^2n)$ to $O(mn)$ where $m$ and $n$ are the sequence sizes being compared. It is this algorithm that is often referred to as the "Smith-Waterman algorithm" [7-9]

These dynamic programming algorithms are rigorous in that they will always find the single best alignment. The drawback to these powerful methods is that they are time consuming and that they only return a single result. In this context, heuristic algorithms such as BLAST and FASTA have gained popularity for performing local sequence alignment quickly while revealing multiple regions of local similarity. While the heuristic methods are valuable, they may fail to report hits or report false positives that the Smith-Waterman algorithm would not. When comparing key sequences in depth, the rigorous Smith-Waterman is desired and necessary.

To speed up the algorithm and to extend it to the ASsociative Computing model (ASC) [10], we developed an associative parallel sequence alignment algorithm called SWAMP. Any solution that uses the ASC model to solve local sequence alignment has been dubbed Smith-Waterman using Associative Massive Parallelism (SWAMP). The SWAMP algorithm presented here is based on an earlier associative sequence alignment algorithm [11]. It has been further developed and parallelized to reduce its running time.

Some of the changes from [11] to the work presented here are:

- Parallel input (usually a bottleneck in parallel machines) has been greatly reduced.
- Data initialization of the matrix has been parallelized
- Comparative analysis between the different parallel versions
- Comparative analysis between different "worst-case" file sizes

The paper is organized as follows: Section 2

discusses the Smith-Waterman algorithm in more detail, Section 3 briefly describes the ASC model and Section 4 describes the improved algorithm. The final sections deal with comparative performance analysis, future work, and conclusions.

## 2. The Smith-Waterman Algorithm

The Smith-Waterman algorithm is a widely used local sequence alignment algorithm. The dynamic programming method aligns characters from two sequences, S1 and S2. The goal is to maximize the substring alignment of similar residues, indicating evolution similarity and lineage.

To this end, gaps may be introduced into the original sequences to allow for a better substring alignment. A gap indicates an insertion to or a deletion from one string to the other, also known as an *indel*. This example alignment has a single indel (a deletion from S1 to S2) between the two strings `CATTG` and `CTTG`:

```
CATTG
C-TTG
```

The Smith-Waterman algorithm is rigorous, checking every possible alignment. Part of the calculation is to compute the effect of indels as well as direct character alignment.

To create biologically relevant results, affine linear gap penalties are used. Since the length of the gap impacts the overall score, it is a linear gap penalty. The penalty of opening up a gap ($g$) is different than the cost of continuing a gap ($\sigma$), making the penalty affine. The likelihood of gap clusters is controlled by these parameters, where $g$'s value is usually larger than $\sigma$'s to encourage fewer gap openings.

To track the scores in the sequential algorithm for deletions, insertions, matches, and mismatches an $m$ x $n$ scoring matrix is calculated where $m$ is the length of S1 and $n$ is the length of S2. A matrix element at location $i,j$ in the matrix represents the score of an alignment ending at residue $i$ of S1 and residue $j$ of S2, where $0 \le i \le$ m and $0 \le j \le$ n.

A deletion from $S1_i$ at $S2_j$ is calculated using the recurrence relationship originally defined in [3]. A new gap opening for a deletion at $C_{i,j}$ as well as a continued deletion (gap extension) are calculated and the maximum value is stored in $D_{i,j}$.

$$D_{i,j} = \max\begin{Bmatrix} C_{i-1,j} - g \\ D_{i-1,j} \end{Bmatrix} - \sigma \qquad (1)$$

An insertion occurs when one or more characters are in string $S2_j$ but not at $S1_i$'s corresponding location. This is calculated using a recurrence relationship similar to $D_{i,j}$. Again, both the cost of opening a new gap as well as extending an already open insertion gap at locations $i$ and $j$ are calculated and the maximum value is retained in $I_{i,j}$.

$$I_{i,j} = \max\begin{Bmatrix} C_{i,j-1} - g \\ I_{i,j-1} \end{Bmatrix} - \sigma \qquad (2)$$

The final component of a matrix cell computation begins with the scoring function for a match or mismatch of the two residues $S1_i$ and $S2_j$, shown in (3).

$$\mathrm{d}(S1_i, S2_j) = \begin{cases} match\_cost & \text{if } S1_i = S2_j \\ miss\_cost & \text{if } S1_i \neq S2_j \end{cases} \qquad (3)$$

Once calculated, the previous score from the northwest neighbor $C_{i-1,\,j-1}$ is added to $\mathrm{d}(S1_i, S2_j)$. The maximum of zero and the three calculated values is assigned to $C_{i,j}$ as shown in (4).

$$C_{i,j} = \max\begin{Bmatrix} D_{i,j} \\ I_{i,j} \\ C_{i-1,j-1} + \mathrm{d}(S1_i, S2_j) \\ 0 \end{Bmatrix} \qquad (4)$$

$C_{i,j}$ is the score of the best local alignment ending at the position $S1_i$, $S2_j$.

Each matrix element relies upon previously computed scores as shown in (1), (2) and (4). Matrix element ($i$,$j$) uses element ($i$-1) to the north, the element ($j$-1) to the west, and the northwest ($i$-1, $j$-1) corner diagonal element.

Figure 1 shows the data dependency for a single matrix element. The shaded matrix element at $i$ = 3 and $j$ = 2 has a computed $C$ value of 16. $C_{3,2}$ depends upon the previously computed matrix elements indicated by the arrows. This forces a particular order in the calculations, since no element can be computed prior to the elements to its north, west, and northwest. This is a limiting factor in the level of parallelization possible in the algorithm.

The sequential Smith-Waterman algorithm [6] makes a second pass over the matrix to find the maximum $C$ value and to perform a traceback from the largest value back to the beginning of that substring

alignment. The traceback process is inherently sequential and it is not parallelized as part of the SWAMP algorithm. It is, however, considered as Future Work in Section 6.

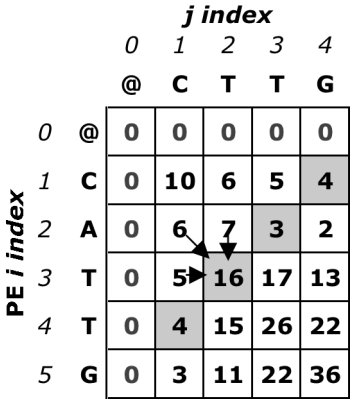


Fig. 1. Sequential Smith-Waterman matrix showing dependencies of cell (3, 2) via the arrows. The calculated *C* values are shown. The shaded anti-diagonal (where all *i+j* values are equal) can be calculated in parallel since its computations are independent of one another. This example's parameters are $g$=3, $\sigma$=1, $d(S1_i, S2_j)$=10 when $S1_i$=$S2_j$ and -3 when $S1_i \neq S2_j$ and the resulting alignment is shown above with the single indel.

## 3. The ASC Model of Computation

When discussing a parallel algorithm, a description of the parallel model of computation is vital for understanding the solution. The model used for this solution is known as ASC. The ASsocative Computing (ASC) model [10][12] is based on the STARAN associative SIMD computer, designed by Dr. Kenneth Batcher at Goodyear Aerospace in the early 1970's, and its heavily Navy-utilized successor, the ASPRO.

The ASC model is a SIMD with a few additional properties, and will be referred to as an associative SIMD. SIMD models, including ASC avoid both multi-tasking and asynchronous point-to-point communication routing schemes, and the inherent problems that result from the use of these techniques. In particular, ASC programmers avoid problems involving load balancing, synchronization, and dynamic task scheduling. These issues must be handled in MPI and other MIMD cluster paradigms.

The ASC model does not employ associative memory. The reference to the word "associative" is related to the use of searching to locate data by *content* rather than memory address. The data remains in place in the responding processing elements (PEs) and the PEs are subsequently processed in parallel.

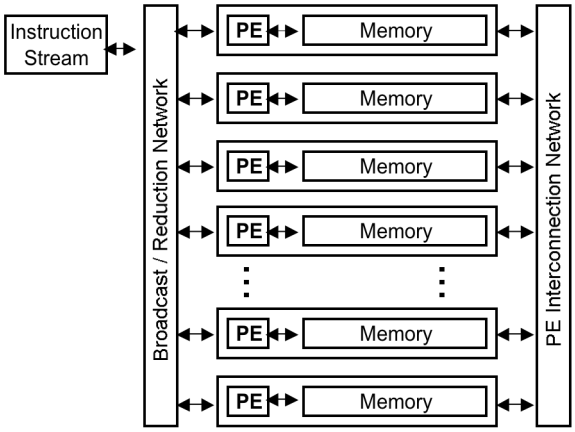


Fig. 2. A high-level view of the ASC model of parallel computation.

As shown in Figure 2, the ASC model consists of an array of PEs each with its own local memory. Every PE is capable of performing local arithmetic and logical operations, as well as the usual functions of a sequential processor other than issuing instructions.

Instead, instructions are issued via the single instruction stream (IS). The IS broadcasts instructions to all PEs. Only active PEs execute the instructions and receive data (also broadcast by the IS). An inactive PE will receive instructions but not execute them. Idle PEs are currently inactive with no essential program data but they can be reassigned as an active PE if needed (i.e. dynamic memory allocation).

The ASC model has fast searching and comparison capabilities. Based on the results of a search, active PEs are partitioned into responder and non-responder PEs. The IS is able to broadcast data to all of its active PEs, flip the responders and the non-responders, as well as select a single PE from the set of responder PEs. An IS may compute the global AND, OR, MAX or MIN of data in all active PEs in constant time [13][14]. The MAX reduction is particularly useful for this local alignment algorithm since the maximum *C* value is returned and used for the traceback.

SIMD computers are notably efficient in data movement. Data is moved in lock step through the network under the control of the compiled program, thereby producing predictable timings. For parallel data communication between PEs, ASC has a PE Interconnection Network. The ASC model allows various types of interconnection networks. This algorithm only requires a linear array network without wraparound, therefore that is what is assumed.

SWAMP and its predecessor [11] are the only known sequence alignment algorithms for an associative SIMD model. The tabular nature of the ASC parallel memory is well suited to the dynamic programming method used by the associative Smith-Waterman algorithm. SWAMP and other ASC algorithms can be efficiently implemented on other systems that are closely related to SIMDs. Two such systems are ClearSpeed Advance X620 and NVIDIA Tesla C870 parallel processing boards. This is discussed in Future Work in Section 6.

## 4. SWAMP Algorithm

The development environment used is the ASC emulator. The parallel programming language shares the name of the model in that it too is called ASC. Both the compiler and emulator are available for download at http://www.cs.kent.edu/~parallel under the "Software" link.

Throughout the SWAMP description, the required ASC convention to include [$] after the name of all parallel variables is used, as seen in Figure 3.

### 4.1 SWAMP Algorithm: Data Setup

SWAMP retains the dynamic programming approach of [6] and also the two-dimensional matrix. Instead of working on one element at a time, an entire matrix column is executed in parallel. However, it is not a direct sequential-to-parallel conversion. Due to the data dependencies, all north, west and northwest neighbors need to be computed before that matrix element can be computed. If directly mapped onto ASC, the data dependencies would force a completely sequential execution of the algorithm.

One of the challenges this algorithm presented was how to store an entire anti-diagonal, such as the one highlighted in Figure 1, in a single ASC variable (column). The second challenge was to organize the north, west, and northwest neighbors to be the same uniform "distance" away from each location for every

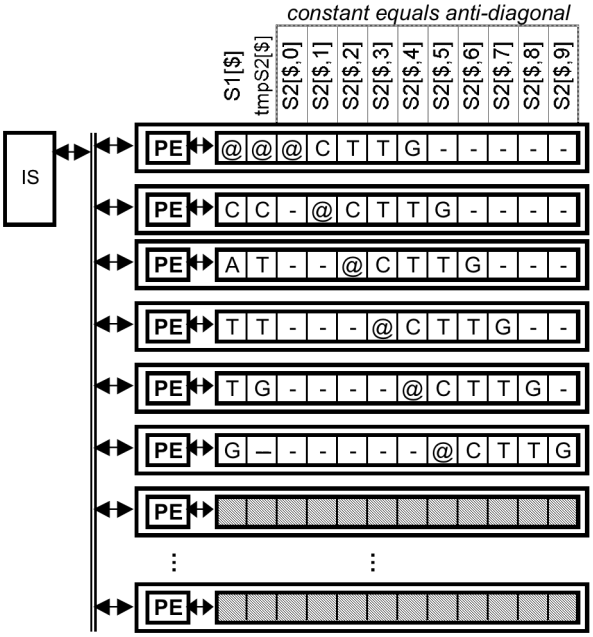


Fig. 3. Mapping the "shifted" data on to the ASC model. Every S2[$] column stores one full anti-diagonal from the original matrix. Here the number of PEs > *m* and the unused (idle) PEs are grayed out. When the # PEs < m, the PEs are virtualized and one PE will process [m/# PEs] worth of work.

D, I, and C value.

To align the values along an anti-diagonal the data is shifted in parallel memory so that the anti-diagonals become columns. This shift allows for the data-independent values along each anti-diagonal to be processed in parallel, from left to right. First the two strings S1 and S2 are read in as input into S1[$] and tempS2[$]. The tempS2[$] data is what gets shifted via a temporary parallel variable and copied into the S2[$] array so that it is arranged in the manner shown in Figure 3.

Instead of a matrix that is *m* x *n*, the new two-dimension ASC "matrix" has the dimensions *m* x (*m*+*n*). There are *m* PEs used, each requiring (*m*+*n*) memory elements for each variable D, I, and C.

When working in an associative SIMD environment, recall that the parallel memory is tabular in nature and all data movement on our linear PE Interconnection Network is synchronous (i.e. parallel movements occur in lock step). The same is true of arithmetic and logical operations. When adding two parallel variables and storing the sum into a third

parallel variable, C[$]=A[$]+B[$], every active PE will add its local value of A[$] to its local value of B[$] and sets its own local value of C[$] to this sum.

A specific example of the data shifting is shown in Figure 4. Here, the fourth anti-diagonal is initialized. To initialize this single column of the two-dimension array, S2[$,4], the temporary parallel variable shiftS2[$] acts as a stack. All active PEs replicate their copy of the 1-D shiftS2[$] variable down to their neighboring PE in a single ASC step utilizing the linear PE Interconnection Network (Step 1). Any data elements in shiftS2[$] that are "out of range" and have no corresponding S2 value are set to the placeholder value "-" (Step 2). The remaining character of S2 that is stored in tmpS2[$] is "pushed" on top (copied) to the first PE's value for shiftS2[$] (Step 3). Then all active PEs perform a parallel copy of shiftS2[$] into their local copy of the ASC 2-D array S2[$, 4] (Step 4).

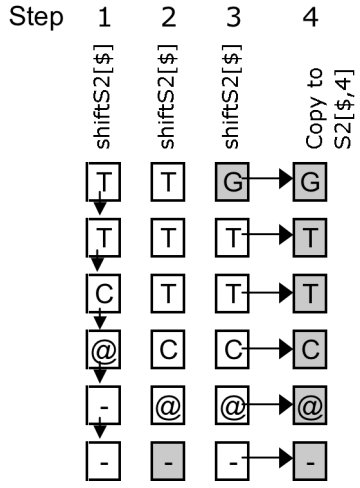


Fig. 4. Showing ($i$+$j$=4) step-by-step iteration of the $m$+$n$ loop to shift S2. This loop stores each anti-diagonal in a single variable of the ASC array S2[$] so that it can be processed in parallel.

Again, this parallel shifting of S2 aligns every anti-diagonal within the parallel memory so that an entire anti-diagonal can be concurrently computed. In addition, the shifting of S2 removes the parallel I/O bottleneck from algorithm in [11]. This new algorithm only reads in the two strings, S1 and S2 instead of reading the entire $m$ x ($m$+$n$) matrix in as input. From there, the setup of the matrix is done completely in parallel inside the ASC program, instead of being created sequentially outside of the ASC program as was done for [11].

## 4.2 Associative Functions for SWAMP

The parallel initialization described in Section 4.1 shifts S2 throughout the matrix. The algorithm now needs to iterate through each of the anti-diagonals to compute the matrix values of *D*, *I* and *C*. As it does this, the algorithm also finds the index and the value of the local (column) maximum using the ASC MAXDEX function.

## 4.3 Algorithm Outline

This SWAMP pseudocode is based on a working ASC language program. Since there are $m$+$n$ anti-diagonals, we assume that they are numbered 0 through ($m$+$n$+1), from the upper left to the lower right. The notation `[$,a_d]` indicates that all active PEs in a given anti-diagonal (`a_d`), process their array data in parallel.

SWAMP Local Alignment Algorithm:
1) Read in S1 and S2
*In Active PEs (those with data values for S1 or S2):*
2) Initialize the two-dimension variables *D[$]*, *I[$]*, *C[$]* to zeros.
3) Shift string S2 as described in Section 4.1
4) For every *a_d* from 2 to *m+n-1* do in parallel {
5) If S2[$,*a_d*] ≠ "@" and S2[$,*a_d*] ≠ "-" then {
6.1) Calculate score for deletion for *D*[$,*a_d*]
6.2) Calculate score for a insertion for *I*[$,*a_d*]
6.3) Calculate matrix score for *C*[$,*a_d*] }
7) local_maxPE=MAXDEX(C[$, *a_d*])
8) if C[local_maxPE, a_d] > max_Val then {
9.1) max_PE = local_maxPE
9.2) max_Val = *C*[local_maxPE, *a_d*]) } }
10) return max_Val, max_PE

Step 2 and 3 iterate through every anti-diagonal from zero through (m+n-1). Step 4 controls the iterations for the computations of *D*, *I*, and *C* from every anti-diagonal numbered 2 through (m+n-1). Step 5 masks off any non-responders including the first "buffer" row and column in the matrix. Steps 6.1-6.3 are based on the recurrence relationships defined in (1-3), respectively. Step 7 uses the ASC MAXDEX function to track the value and location of the maximum value in Steps 9.1 and 9.2.

## 5. Performance Analysis

### 5.1 Asymptotic Analysis

Based on an analysis of the pseduocode from Section 4.3, there are three loops that execute for each anti-diagonal O($m$+$n$) times in Steps 2-4. Step 3 and each substep of 6 require communication between PEs. The communication is with direct neighbors, at most one PE to the north. Using a linear array without wraparound, this can be done in constant time. Step 7 finds the PE index of the maximum value or MAXDEX in constant time as described in Section 3.

Given this analysis, the overall time complexity is O($m$+$n$) using $m$+$1$ PEs. The extra PE handles the border placeholder (the "@" in our example in Figure 3).

This is asymptotically the same as the algorithm presented in [11].

### 5.2 Performance Monitor Result Analysis

Where the performance diverges is through comparisons based on the number of actual operations.

Performance is measured by using ASC's built in performance monitor. It tracks the number of parallel and sequential operations. The only exception is that input and output operations are not counted.

Improvements to the code include the parallelization of the data shifting setup discussed in Section 4.1, the move of the initialization of $D$, $I$, and $C$ outside of a nested loop, and changes in the order of matrix calculations for $C$'s value when finding its max among $D$, $I$ and itself.

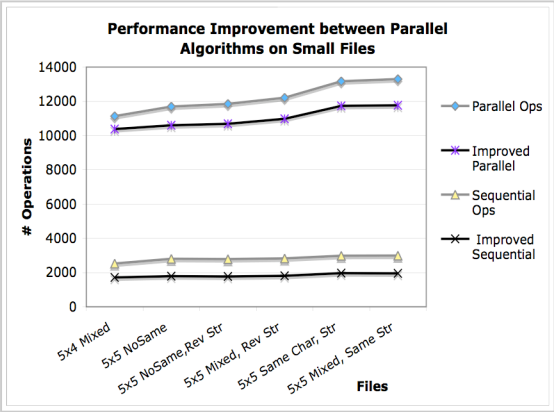


Fig. 5. Reduction in the number of operations through further parallelization of the SWAMP algorithm.

The files used in the evaluation are all very small with most sizes of S1 and S2 equal to five. Even with the small file size, an average speedup factor of 1.08 for the parallel operations and an average 1.54 speedup factor for sequential operations was achieved. The impact of these improvements is greater as the size of the input strings grows.

The type of data in the input files also impacts the overall performance. For instance, the "5x4 Mixed" file has the two strings `CATTG` and `CTTG`. This input creates the least amount of work of any of the files, partly due to its smaller size ($m$=5 and $n$=4) but also because not all of the characters are the same, nor do they all align with one another. The file that used the highest number of parallel operations is the "5x5 Mixed, Same Str." This file has the input string `CATTG` twice. This had slightly higher number of parallel operations than the two strings of `AAAAA` from "5x5 Same Char, Str" file.

The lower factor speedup of 1.08 in parallel operations is due to the matrix computations. This is the most compute-intensive section of the code and no parallelization changes were made to that section of code. Its domination can be seen in Figure 5, even with these unrealistically small files sizes.

The improvement for parallelizing the setup of the parallel data (i.e. the "shift" into the 2-D ASC array) is shown in Figure 5.

What is not apparent and not shown in Figure 5 is the huge reduction in parallel I/O. This is because the performance monitor is automatically suspended for I/O operations. The $m$ x ($m$+$n$) shifted S2 data values are no longer read in. Instead, only the character strings of S1 and S2 are input from a file. When working on actual hardware as well will our future work, I/O is a major concern as a bottleneck. This algorithm greatly reduces the parallel input from $m(m+n)$ or O($m^2$) to O(max($m$, $n$)).

### 5.3 Predicted Performance as S1 and S2 Grow

The level of impact of the different types of input was unexpected. After making the improvements to the algorithm and the code, performance was measured using the worst-case input: two identical strings of mixed characters. The two strings within a file were made the same length and were a subset of a GenBank nucleotide entry DQ328812 (Ursus arctos haplotype). SWAMP was tested with $m$ and $n$ set to lengths 3, 4, 8, 16, 32, 64, 128 and 256. We could not go beyond 256 due to the emulator constraints.

String lengths larger than 256 are performance predictions obtained using linear regression and the least squares method. These predictions are indicated with a dashed line in Figure 6.

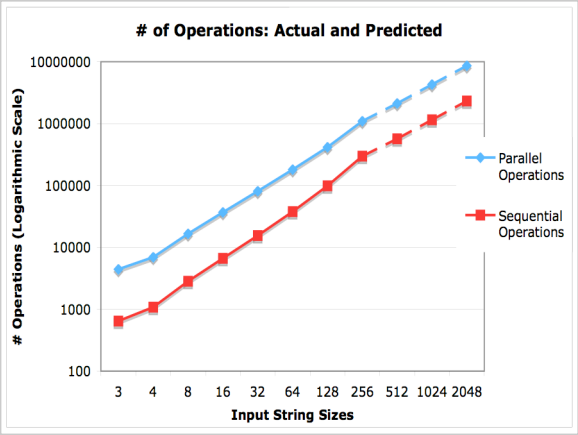


Fig. 6. Actual and predicted performance measurements using ASC's performance monitor. Predictions were obtained using linear regression and the least squares method and are shown with a dashed line.

Figure 6 demonstrates that as the size of the strings increases the number of operations' growth is linear, matching our asymptotic analysis. Note that the y-axis scale is logarithmic since the file sizes are doubling at each data point beyond size 4.

These predictions assume that there are |S1| or $m$ PEs available. If this is not the case, as mentioned in Figure 3's caption, there would be a slowdown coefficient of $m$ / (# PEs).

## 6. Future Work

In looking at the difference in the number of operations based on the type of input in Figure 5, it would be interesting to run a brief survey on the nature of the input strings. Since highly similar strings are likely the most common input, further improvements should be made to reduce the number of operations for this current worst case. Rearranging a section of the code would not change the worst-case number of operations, but it would change when the worst-case occurs.

Another consideration is to combine the three main loops in the Steps 2-4 of this algorithm. Instead of subroutine calls for the separate steps (initialization, shifting S2, computing *D, I* and *C*), they can be combined into a single loop and the performance measures re-run.

A non-trivial future goal is to provide a rigorous local alignment algorithm that provides multiple local non-overlapping, non-intersecting regions of similarity. Currently, only one subsequence alignment is found in SWAMP.

SIM [7] and LALIGN [15] are rigorous algorithms that provide multiple regions of similarity, but they are sequential with slow running times similar to the sequential Smith-Waterman.

Our approach would be to provide a parallel algorithm that yields the same results as SIM and LALIGN but with a reduced run time. This would require extending SWAMP. This extension (SWAMP+) will focus on the traceback section of the algorithm, utilizing ASC's powerful content searching capabilities.

Current and future work also includes moving SWAMP from an emulator to hardware using the ClearSpeed Advance X620 PCI board. The commercially available COTS board is a SIMD accelerator. In order to run ASC algorithms, the additional functions specific to ASC need to be implemented for the ClearSpeed board. This necessary associative functionality is currently under development at Kent State University.

A similar situation exists with the two NVIDIA Tesla C870 computing boards recently obtained through an equipment grant from NVIDIA. Equivalent associative functions will need to be developed for these GPU computing processors.

The associative functionality will allow for the migration of ASC algorithms onto these computing platforms, including SWAMP, and allow for practical testing with full-length sequence data.

Another ASC algorithm of special interest is an efficient pattern-matching algorithm [16]. Preliminary work shows that [16] could be a strong basis for an associative parallel version of a nucleotide search tool that uses spaced seeds to perform hit detection similar to MEGABLAST [17] and PatternHunter [18].

## 7. Conclusions

Further parallelization helped to reduce the overall number of operations and improve performance. The average number of parallel operations improved by a factor of 1.08, and the sequential operations by an average factor of 1.53 with *extremely* small file sizes of only 5 characters in each string. The greater impact of the speedup will be obvious when using string sizes

that are several hundred or several thousands of characters long.

Awareness about the impact of the different file inputs was raised through the different tests. The difference in the number of operations for such small file sizes was unexpected. In all likelihood, the pairwise comparisons are between highly similar (biologically homologous) sequences and therefore the inputs are highly similar. This prompts further investigation of how to modify the algorithm structure to change when worst-case number of operations occurs. It may prove beneficial to switch the worst case from happening when the input strings are highly similar to when the strings are highly dissimilar, a more unlikely data set for SWAMP.

Parallel input was greatly reduced to avoid bottlenecks and performance degradation. This is important as part of the future work to migrate the SWAMP algorithm onto the ClearSpeed Advance X620 board and the NVIDIA Tesla C870 boards.

Overall, the algorithm and implementation is better designed and faster running than the earlier ASC alignment algorithm. In addition, this stronger algorithm makes for a better transition to the ClearSpeed and NVIDIA parallel acceleration hardware.

## 8. References


[1] Altschul, S.F., T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, *Nucl. Acids Res*., *25(*17*)*, 1997, 3389-3402.

[2] FASTA Sequence Comparison Suite. Available for remote running at http://fasta.bioch.virginia.edu/fasta_www/home.html or for download at ftp://ftp.virginia.edu/pub/fasta. Last accessed on July 31, 2006.

[3] Smith, T.F. and Waterman, M.S. "Comparison of Biosequences," Adv. Appl. Math, 1981, 2:482-489.

[4] NIH GenBank genetic sequence database. http://www.ncbi.nlm.nih.gov/Genbank/index.html. Last accessed on January 25, 2008.

[5] Needleman, S.B. and C.D. Wunsch. "A General Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins." *Journal of Molecular Biology*, 1970, 48:443--453.

[6] Gotoh, O. "An Improved Algorithm for Matching Biological Sequences," *Journal of Molecular Biology, 162,* 1982, 705-708.

[7] Huang, X. and W. Miller, "A Time-Efficient, Linear-Space Local Similarity Algorithm." *Advances in Applied Mathematics*, 12, 1991, 337-357.

[8] Camerson, M., and H. Williams. " Comparing Compressed Sequences for Faster Nucleotide BLAST Searches." I*EEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(3) July-Sept. 2007, pp 349-364.

[9] Frey, J. D. *The Use of the Smith-Waterman Algorithm in Melodic Song Identification.* Preliminary copy of Master's Thesis from Kent State University to appear May, 2008.

[10] Potter, J.L., J. Baker, A. Bansal, S. Scott, C. Leangsuksun, and C. Asthagiri. "ASC: An Associative Computing Paradigm." *IEEE Computer*, 27(11) Nov. 1994, 19-25.

[11] Steinfadt, S., M. Scherger, J.W. Baker. "A Local Sequence Alignment Algorithm Using an Associative Model of Parallel Computation", *IASTED Computational and Systems Biology (CASB 2006)*. November 13-14, 2006, Dallas, TX, 38-43.

[12] Potter, J.L., Associative Computing - A Programming Paradigm for Massively Parallel Computers (NY, NY: Plenum Publishing, 1992).

[13] A. D. Falkoff, Algorithms for Parallel-Search Memories, Journal of the ACM, 9(4), October 1962, 488-511.

[14] M. Jin, J. Baker, & K. Batcher, Timings for Associative Operations on the MASC Model, Workshop on Massively Parallel Processing. Proc. 15th IEEE International Parallel and Distributed Processing Symposium, San Francisco, CA, 2001, 193.

[15] Pearson, W. and D. Lipman. "Improved tools for biological sequence comparison", Academy of Sciences USA 85(8), 2444–2448.

[16] Esenwein, M. J. Baker. "VLCD String Matching for Associative Computing and Multiple Broadcast Mesh." *Proc. of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 69-74. 1997.

[17] Zhang, Z., Schwartz, S., Wagner, L. & Miller, W. "A greedy algorithm for aligning DNA sequences." *Journal of Computational Biology* 7(1–2), 203–214, 2000.

[18] Ma B, Tromp J, Li M,. "PatternHunter: Faster and more sensitive homology search," *Bioinformatics* 18(3):440–445, March 2002.