Speeding Up Sequence Alignment Algorithms via Parallel Programming: Dividing Work into Chunks

Olger Calderón Achío*, Wilberth Castro Fuentes*, Irene Gamboa Padilla*,
Andrés Morales Esquivel*, Diego Pérez Arroyo*
*Instituto Tecnológico de Costa Rica

***Abstract*—There are several problem contexts (scientific, financial) where data intensive applications might highly benefit from high performance computing techniques. Probably the most common mean is to exploit the parallelism intrinsic to the problem using the available parallel hardware. Modern SOs provide the means to exploit this approach from a programming perspective (via multi-threading services). We characterize a type of dynamic programming algorithms as chunk-based and introduce a solution for parallelizing them. This solution has the property of equally distributing work among different threads, even if they remain idle at certain times. We plan to implement and compare, specifically, parallel versions of some sequence alignment algorithms (Needleman-Wunsch, Smith-Waterman), which are definitely not embarrassingly parallel and might benefit from our workload distribution strategy. Preliminary results suggest that our algorithm and modeling works, and with more tuning to our solution even better speedups could be obtained.**

***Index Terms*—Parallel Programming, Synchronization Techniques, Operating Systems, Dynamic Programming, High Performance Computing, Sequence Alignment.**

I. Introduction

Sequence Alignment algorithms helps us compare different strands of DNA, RNA or proteins, different parallelization methodologies has been researched to increase the speedup of the different algorithms, this is because the strands of DNA, RNA and proteins become bigger and bigger with more biology research going on.

With much more data to analyze the algorithms used need to handle not only a bigger set of data but also return results in a reasonable time. That's why we propose a parallelization methodology that exploits hardware resources (in this case multi-threaded processors) with the help of Operating Systems that provide access to these resources. We use the Needleman-Wunsch and the Smith-Watermam dynamic programming algorithms and propose a "Chunk-based" workload division strategy for speeding up those algorithms.

Dynamic Programming algorithms are hard to parallelize due to the fact that they need previous data to continue computing the results. So it can't be trivially partitioned for different threads for executing work in parallel. There have been different approaches to parallelize the Needleman-Wunsch and Smith-Waterman algorithms [1]–[4].

In our approach we give each thread a chunk of work to do, and they are blocked with semaphores until there is enough data for they to work with. We propose an experiment to provide enough information to verify our hypothesis that this approach can achieve a speedup that is considerably good enough to continue researching and comparing the results with other approaches.

This paper is organized in the following way: The section 2 explains the background about parallel hardware and different algorithms. The parallel dynamic programming architecture it's found in the section 3. Section 4 talks about our Chunk-Based Solution. The proposed experiments can be seen in the section 5. The section 6 and 7 have the Related Work and Future Work respectively. And finally we will write our conclusions in the last section.

II. Background

Since the molecular biology began, it has been necessary to develop sequence alignment methods, the development of these methods has taken many years and it's not as simple as matching the length of the sequences and then make an exhaustive comparison. Just find a way to match is a hard work, there are many possibilities, but the most convenient are needed. For this, an additive model was developed, a points-accumulation model, valuing the matches of symbols, the model has been commonly called "+1-1-2".

The dynamic programming algorithms essentially divides a large problem (e.g. the full sequence) into a series of smaller problems and uses the optimal solutions to the smaller problems to reconstruct an optimal solution to the larger problem.

In 1970, Saul B. Needleman y Christian D. Wunsch published [5], a method for global sequence alignment with dynamic programming (one table), and in 1981, Temple F. Smith y Michael S. Waterman published [6], a method for local alignment of sequences consisting of a Needleman-Wunsch variant method. However, these methods did not have mechanisms for compaction of the sequences.

With semi-global alignment techniques this lack was satisfied, leaving the gaps penalization. In general, the technique was useful for aligning long sequences with short sequences and to find similar fragments.

The problem with these methods is that with long sequences are very expensive. As a result of an effort to reduce these costs, knowing that two sequences to align are similar (commonly correspond to the same gene or protein), the k-Band technique was developed [7], with which can be used the Needleman-Wunsch method. The technique focuses on a set of central diagonals in the table, called band.

But, still missing the stimuli of the gaps continuity, for achieve this, penalty rules were changed. Then the penalty is

defined in function of the number of consecutive gaps, block of gaps. For complexity reasons, it's recommended that let this function as linear function, and must penalize more the first gap over another [8].

To achieve this it's necessary to know whether a block is spreading or is the first gap. The most common solution requires three tables. The first registers the best score by align symbol with symbol. The second registers the best score by align a gap with a symbol in the first sequence. The third registers the best score by align a gap with a symbol in the second sequence. The optimal alignment path may be in the three tables [8].

However, this algorithm is not always better than previous versions, it's good when you want to stimulate the gaps continuity, in other cases can generate overhead. The spatial and temporal complexity of this algorithm is quadratic although uses three tables, this if the penalty function is linear.

It's possible to decrease the time complexity with the k-Band technique, and the spatial complexity, if only the score of the alignment is needed, it can be computed in linear space using only a vector. Also, it's possible to write a function that requires only linear space to find the optimal alignment score of an arbitrary sequence and the suffixes of another sequence. In this way it's possible to obtain quadratic spatial complexity, but could increase the time complexity [9].

Speaking about information quantity, there may be matches more significant, also there may be differences between mismatches, thus, the model "+1-1-2" could be very simple, so, with amino-acid sequences, the use tables for obtain the scores of matches and mismatches is recommended.

The amino-acid substitution matrices are tables with scores for all possible pairs of symbols, the score depends on properties of each pair as hydrophobicity, polarity, charge, molecular weight, number and type of codons, frequency, etc. Margaret Oakley Dayhoff designed one of the first substitution matrices, the PAM (Point Accepted Mutation) matrices, based on specific mutations that come to be accepted by nature [10].

The BLOSUM (BLOcks of Amino Acid SUbstitution Matrix) matrices are another alternative, these are used to assess sequence alignments of evolutionarily divergent proteins. Are used in local alignments, introduced in 1992 in [11] by Henikoff and Henikoff.

Finally, High-performance computing (HPC) is applied to speedup long-running scientific applications, for instance the simulation of computational fluid dynamics (CFD), or in sequence alignment algorithms. Today's supercomputers often base on commodity processors, but also have different facets: from clusters over (large) shared-memory systems to accelerators (e.g. GPUs). Leveraging these systems, parallel computing with e.g. MPI, OpenMP or CUDA must be applied.

III. Parallel Dynamic Programming Architecture

An important aspect that we consider when implementing more than one dynamic programming parallelization approach is to be able to separate or decompose the solutions into different layers of processing. In particular, we have structured our solutions in 3 layers on processing:

1) Cell filling logic: is the one at the lowest level and the one used to calculate each individual cell's value. In other words, the implementation of Bellman equation. When we execute the logic to fill a cell, it is a assumed that all required dependencies of this cell have been previously calculated.
2) Filling patterns: dynamic programming algorithms are usually characterized by filling the matrix (or matrices) using different patterns or shapes. It is common to say: we are filling this matrix by subsquares o by diagonals. A filling pattern algorithm fills a group of cells following a certain order. Such an order should be compatible with each cell individual dependencies.
3) Matrix filling: this layer constitutes the high-level approach for filling the whole matrix. It is at this level where we decide how exactly the parallelization effort is implemented. We could decide, for example, to fill certain different shapes in parallel. Being $N \times N$ the dimensions of a dynamic programming matrix, at the simplest case, a matrix filling algorithm could be to fill the subsquare which has its upper-left corner at cell $(0, 0)$ and has height and width equal to N. In this case, it turns out that the given subsquare is equal to the whole matrix, but it doesn't have to be the case. More elaborate matrix filling algorithms should make use of the filling primitives in the layer below.

The previously described software architecture facilitates the development of different versions of the same algorithm. We are decoupling the logic needed to satisfy Bellman equation at each individual cell (the lowest most layer), from the patterns for filling subparts of the matrix (the layer in the middle), from the high-level logic used to fill the whole matrix (which could involve parallel programming logic if desired). Changing the parallel programming strategy should not affect the logic in other layers, as they act as primitives for the layer above them.

It should be also noted that the idea behind this layering, is in fact to expose parallelism in dynamic programming applications. By having different groups of cells that could be filled independently, and by having filling pattern primitives that could be applied to each one of them, we are greatly simplifying the writing process of dynamic programming parallel programs.

We will describe how each of these layers plays a role in our implementation. A prototype that makes use of these concepts can be found at https://github.com/diepe28/Proyecto2-BMC-Alineamientos.

A. Cell filling logic layer

We have devised a simple `fill` primitive for our sequence alignment algorithms. This routine will apply the respective Bellman equation to one cell and store the calculated results in it. It should be noted, that in our implementation we are using a traditional C language matrix for storing the values. Given that we are using the affine-function versions of the sequence alignment algorithms for penalizing gaps, we will store three values on each cell instead of one. Other authors

refer to three distinct matrices, but we will refer to just one for the sake of simpleness. To fill one cell in our matrix will be equivalent to filling each one of the three cells of the three mentioned matrices at the same positions.

Our `fill` primitive will differentiate between two cases: filling a corner cell and filling an interior cell. For our purposes, an interior cell is one that needs all dependencies to satisfy its Bellman equation (one cell to the north, one to northwest and the last one to the west). A corner cell is any cell which filling logic is different than interior cells just because at least one of the dependencies is non existent (cells at row or column 0 in our case). The `fill` primitive looks like:

```
void fill(Cell*** matrix, ScoringOptions*
    options, char* seq1, char* seq2, int
   x, int y)
{
   if (x == 0 || y == 0)
      fill_corner (matrix, options, x, y);
   else
      fill_interior (matrix, options, x,
   y, seq1, seq2);
}
```

Here `x` and `y` are the row and column (respectively) of the cell that is being filled. Each one of the functions `fill_corner` and `fill_interior`, have the appropriate logic to satisfy the Bellman equation of the alignment algorithm.

B. Filling patterns layer

As mentioned previously, there could be potentially different types of filling patterns that we would like to support. For our purposes it suffices to have a pattern for filling a submatrix. Later on, we will use the terms submatrix and chunk interchangeably. Our submatrix filling primitive is defined as follows:

```
void fill_submatrix(Cell*** matrix,
    ScoringOptions* options, int startX,
    int startY, int height, int width,
    char* seq1, char* seq2)
{
   int i, j = 0;
   int xLimit = startX + height;
   int yLimit = startY + width;
   for (i = startX; i < xLimit; i++) {
      for (j = startY; j < yLimit; j++)
         fill (matrix, options, seq1,
   seq2, i, j);
   }
}
```

Notice that we are making use of the previously described `fill` function, and also that we are filling the submatrix from up to bottom, and from left to right. The filling directions were not picked deliberatively, as this is required by the studied sequence alignment algorithms. This filling primitive could be enhanced to support different filling directions, but that is outside of the scope of this paper.

Just for illustration purposes, in a k-band version of Needleman-Wunsch algorithm (where conceptually the filling is done by antidiagonals instead of submatrices) we could define a filling primitive with the following function signature:

```
void fill_antidiagonal(Cell*** matrix,
    int n, char* seq1, char* seq2, int
    seq1Length, int seq2Length, int k,
    ScoringOptions* scoringOptions);
```

where n is the number of antidiagonal in the matrix (if they were numbered from top to bottom) and k is the width of the band. The idea is that the programmer specifies his/her own primitives in a way that makes sense for the specific dynamic programming problem they are currently addressing. In this modeling is important to decompose the whole task (filling the matrix) into subtasks that could be potentially executed in parallel.

C. Matrix filling layer

In case we want a sequential implementation for one of our sequence alignment algorithms, then we just need to apply our previously defined primitive in the following way:

```
void fill_similarity_matrix_full(Cell***
    matrix, char* seq1, char* seq2, int
    seq1Length, int seq2Length,
    ScoringOptions* scoringOptions)
{
    fill_matrix (matrix, scoringOptions,
   0, 0, seq1Length + 1, seq2Length + 1,
   seq1, seq2);
}
```

That was kind of simple. But for our parallel implementation we will have more complex schemes at this particular layer. And that's what we will describe in the following sections.

IV. Chunk-based Solution

We claim that Needleman-Wunsch and Smith-Waterman algorithms are **chunk-based**. For dynamic programming algorithms we define **chunk-based** as follows. Given a dynamic programming matrix A of dimensions N and M:

- An arbitrary chunk size $S = (n, m)$ could be chosen for the algorithm, where $n \geq 1 \wedge n \leq N$ and $m \geq 1 \wedge m \leq M$.
- If chunk size $S = (n, m)$ is chosen, then each chunk is a submatrix of A composed of n adjacent rows and m adjacent columns of A. If n is not divisible by N or m is not divisible by M then some chunks will have sizes different to S. We will call these "leftover" chunks. For simplicity we will assume that n is divisible by N and m is divisible by M.
- There are no cells shared between chunks.
- A chunk is said to be filled if all its cells have been already calculated.
- A chunk could have at most 8 adjacent chunks (some of which could depend on). Assuming the set of directions relative to a chunk are: D =

$\{N, W, E, S, NW, NE, SW, SE\}$, the set of chunk dependencies directions C_D is a subset of D.
- The chunk dependencies directions should be always the same for any chosen chunk size.
- We also define the granularity of a chunk of size S as $G(S) = \frac{n+m}{2}$.

The previous definitions tell us intuitively, that Bellman Equation for a (**chunk-based**) algorithm defines C_D, and conceptually, chunks are thought as having $S = (1, 1)$ (chunks are individual cells).

For our studied algorithms we have that $C_D = \{N, NW, W\}$. Each chunk depends at most of three other chunks: the one located to the north, the one located at the north-west and the one located to the west. To fill a chunk we need to be certain that its dependent chunks have been previously filled.

A. Workload Distribution

Next, we will detail how we distribute the work in our Chunk-based Solution among the different processing units (which are threads in our case, but this could be generalized). In particular, we wish that the algorithm will scale to the number of provided threads of execution. Our solution guarantees that each thread will have an equal amount of work and each one remains idle approximately the same amount of time.

Our solution is designed to receive as an input T (the number of threads of execution that will be used). Then we perform automatically the workload distribution among the threads in a way we consider appropriate. We use the following algorithm for distributing the work given a dynamic programming matrix A with dimensions $N \times M$:

1) We choose chunk size width $m = M/T$ [1].
2) We choose chunk size height $n = N/(2 * T)$.
3) We divide the dynamic programming matrix in $2T$ equal strips. A strip is an horizontal concatenation of chunks that goes from column 0 to column $M-1$. That is, one strip spans all matrix columns.
4) Strips are all labeled with a number. We will denote these $S_0, S_1, \cdots, S_{2T}$.
5) For each thread t_i, $0 \leq i \leq T-1$, assign the strips S_i and S_{i+T} to be filled by t_i. Given this, it means that each thread will be responsible of filling two strips which are $T-1$ strips apart. Thread t_i will first fill strip S_i from left to right (chunk by chunk), and then will jump to fill strip S_{i+T}. See figure 1. There is no particular reason of choosing two strips per thread other than reducing the percentage of idle time for each thread (this will be explained later). We don't provide any specific formula, but we claim that more strips could be assigned to each thread as the number of computer cores goes up.
6) We can think of each strip S_k as a set of chunks. That is $S_k = \{C_{k,0}, C_{k,1}, C_{k,2}, \cdots, C_{k,T-1}\}$.

To achieve significant speedup the number of threads T should be chosen carefully. In one extreme, if we choose to use a lot of threads, then chunk granularity will be really small. There will be higher parallelism, but too much synchronization overhead (see section below). In the other extreme, if we choose a small number of threads then we will have a big chunk granularity that will result in a smaller degree of parallelism, but synchronization overhead will be small. Experimental results suggests that choosing T to be equal to the number of processor cores in the machine might be the best option. A similar distribution of work via strips was proposed in [12].

[1]We will assume that both m and n are integer numbers. Extending the algorithm for handling leftover chunks is not difficult but it is outside the scope of this paper.

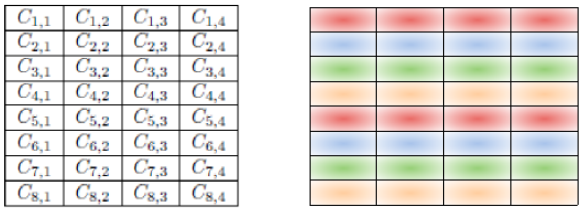

Fig. 1. An example of a workload distribution for 4 threads according to our solution. Chunks of the same color are assigned to the same thread. In this case, each strip is composed of four chunks.

B. Matrix Filling Strategy

Given the nature of the algorithms, it is not possible to start the execution with all worker threads filling their strips at the same time. Instead, threads are added and removed incrementally to fill strips as needed. To better understand this, we should divide the execution of the algorithm in phases.

Let's denote P_i as the i-th phase of the algorithm. The computation executed by all threads at phase P_i should give as a result the filling of an antidiagonal of chunks. Each phase should take roughly the same amount of time.

- On P_0 the only available chunk to fill is the upper-left most, that is $C_{0,0}$. The thread t_0 is the one assigned to this chunk so it will be the only one that is active.
- On P_1 we can calculate chunks $C_{0,1}$ and $C_{1,0}$. Thread t_0 calculates $C_{0,1}$ and thread t_1 starts doing its work at $C_{1,0}$.
- On P_2 we can calculate chunks $C_{0,2}$, $C_{1,1}$ and $C_{2,0}$. The chunk $C_{0,2}$ is filled by t_0, $C_{1,1}$ is filled by t_1, and $C_{2,0}$ by t_2 (that just got added to the set of active threads).

C. CPU Utilization

The number of antidiagonals in a simple matrix A of dimensions $N \times M$ is $N + M - 1$. Given that our algorithm partitions the matrix in $2T \times T$ chunks, it can be safely assumed that the matrix has $2T + T - 1 = 3T - 1$ chunk antidiagonals. So there is a total of $3T - 1$ phases in the algorithm. A thread will be idle $T-1$ phases. Thread t_i needs to wait until phase i to start filling its first strip (it will be idle from P_0 to P_{i-1}, which constitutes i phases). Thread t_i will also remain idle for the last $T-1-i$ phases (this compensates for threads that start early which also finish early). So in total each thread will be idle $i + (T - 1 - i) = T - 1$ phases. We can calculate a ratio of the time that a thread remains active during the execution of the algorithm as $1 - \frac{T-1}{3T-1}$.

For two threads ($T = 2$) we have that the CPU utilization ratio is 80%. For $T = 3$ the utilization ratio drops to 75%. It appears that the higher the number of threads then the utilization ratio will be lower. However, the function seems to stabilize at very high values of T, where utilization remains fixed at 66%. Assigning more strips to threads will increase the overall number of phases in the algorithm while keeping the number of idle phases constant. This could be used if overall CPU utilization wants to be increased, however there is a risk that synchronization overhead dominates computation if too many strips are assigned to threads.

D. Semaphore-based Synchronization

For achieving correct synchronization between threads we used POSIX semaphores. Our solution makes use of T semaphores (there is one semaphore associated with each thread). The idea is pretty simple and is based widely on the producer/consumer kind of problems. Thread t_i will act as a producer for thread $t_{i+1\%T}$.

All semaphores are initialized with their values set to 0. Every thread, except thread t_0 when filling S_0 executes a `sem_wait` on its own semaphore before trying to fill any chunk. After receiving the proper notification from thread $t_{i-1+T\%T}$, it proceeds to fill its next chunk, then it executes a `sem_post` on thread's $t_{i+1\%T}$ semaphore. This creates a ring kind of communication structure where each thread notifies the next that the chunk just above the one they are waiting to fill is ready.

Following the previous scheme it is guaranteed that a chunk, before being filled, will always have its chunk dependencies calculated. Suppose we are in phase P_i of the algorithm. A thread waits for a semaphore notification before filling a chunk c, which indicates that the chunk located to the north c_N has been filled. Chunk c also depends on the chunk located to the west c_W and the one located to the north-west c_{NW}. Given that threads fill chunks inside a strip in west to east sense, then it can be safely assumed that the chunk located to the west of c has been already been filled in phase P_{i-1}, and the same to the chunk located to the west of c_N which is in fact c_{NW}.

See Figure 2 for an example of how the algorithm progresses among phases.

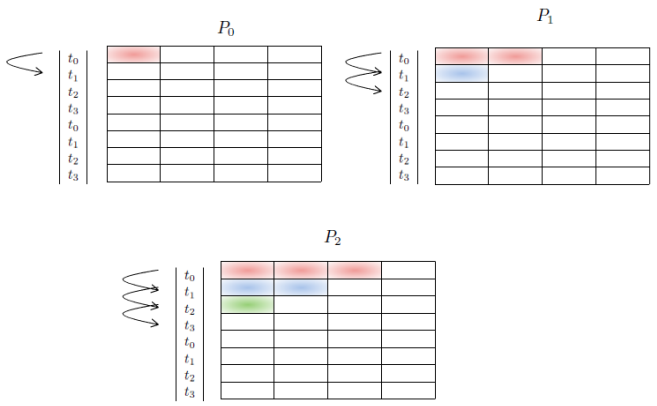

Fig. 2. An example of how the algorithm progresses among phases. Each image illustrates the chunks that have being already calculated at the end of each phase, along with the semaphore notifications between processors.

V. Proposed Experiments

A. Concepts

For the Proposed Experiment we are going to analyze the use of our parallelization approach applied to the Smith Waterman and Needleman-Wunsch algorithms.

The response variable, which is the variable that is going to be the one used to measure the results of the experiment will be the Speedup. The Speedup is defined as the ratio of the best sequential time over the parallel time with p threads or $t(1)/t(p)$ where t is the function that calculates the time used by p threads.

The parameters of the experiment, the invariant part of the experiments, the inputs that won't change between executions of the experiment, would be that we are going to use our approach (chunk-based), and the hardware in which it will run.

The hardware in which the experiment will run consists of a Intel core i7 processor, with 8 processing cores, 16GB of RAM running Ubuntu on a Virtualbox Virtual Machine.

Another parameter is the match value, when the bases to compare are equal, this value will be equal to $+1$, the miss-match value, for bases that do not match, will be -1 and the gap value, the value that we use when we prefer to insert a gap in between the sequence rather than do a match or miss-match is a function of the form $n+Km$, where $n = -2$, the penalty of opening a gap block, $m = 0$, the penalty of continuing a gap block and k is the k-th gap in the block; this function has the purpose of opening any gap block at the penalty of -2. This values are used when comparing DNA sequences.

In the case of Protein sequences we are going to use the Blosom45 value table, when matching amino acids and the same gap block function just described.

B. Factors

The factors are the variable inputs of the experiment we are going to perform. This variables are also the ones we want to study for effects in our proposed approach.

In the case of our experiments we are going to have 3 factors:

- Number of Threads: This factor is the number of threads that are going to be executed concurrently. This factor will have 8 different levels, from 1 thread to 8 threads, since this is our maximum hardware capacity .
- Sequences: These are the input for the algorithm itself, we are going to have 2 pairs of DNA sequences, the first pair is the first chromosome of a Macaca monkey versus the first chromosome of a Gorilla, which consists of 8.632 base pairs and 7.655 base pairs respectively. For the other Sequence pair we are going to use a sequence of amino-acids of Myosin-4 (Myosin is a family of proteins which are involved in the muscle contraction) of Trichinella Sp (a parasite) with 1966 amino-acids and the Myosin-3 (a

different class of the same protein family) of the same parasite with 1970 amino-acids [2].
- Algorithm: This variable describes the algorithm that is going to be used with our approach, this factor is going to have 2 levels: SW (Smith-Waterman) algorithm and NW (Needleman-Wunsch) algorithm.

We are obtaining the data of the sequences that we are going to use from the from the National Center of Biotechnology Information (NCBI), which is part of the United States National Library of Medicine, a branch of National Institutes of Health [13].

C. Experimental Design

The experiments will take into account all the defined above, so in total we are going to have 32 configurations of the experiment, which are defined below:

Number of Threads	*Sequences*	*Algorithm*
1 to 8	Macaca Monkey vs Gorilla	SW
1 to 8	Macaca Monkey vs Gorilla	NW
1 to 8	Myosin-4 vs Myosin-3	SW
1 to 8	Myosin-4 vs Myosin-3	NW

We also propose running each configuration 5 times, this to eliminate noise from other processes and get more concrete results from each configuration, this causes a total of 80 executions for the NW algorithm, 40 with the Macaca vs Gorilla sequence and 40 with the Myosin-4 vs Myosin-3 sequence.

Similarly we get 80 executions for the SW algorithm, for a total of 160 executions with results to be analyzed.

D. Analysis and Expected Results

For the correct analysis of our experiment we are going to need ANOVA (Analysis of Variance) which will help us to compare the different speedup that we get from our executions. We need to use ANOVA because our experiment takes into account 3 factors: Number of threads, Sequences and Algorithm used, this makes it a candidate to be analyzed with ANOVA.

Our null hypothesis (H_0) is that there will be no considerable speedup using our Chunk-based approach, while our alternate hypothesis (H_1) would be that there is a considerable speedup of the algorithms by using a Chunk-based approach to parallelize the algorithms.

The results we expect that will happen by running our experiments would be that we will achieve some considerable speedup in both algorithms as the thread counts goes up (with 8 the maximum of concurrent threads we can achieve with our current hardware platform).

[2]For this experiments the result of the aligned sequences is not what matters to us, but the execution time of the algorithms

E. Preliminary Results

In this sub-section we present the result of the experiments just described. In figure 3 we see the speedup obtained with our chunk-based parallelization approach of the Needleman-Wunsch algorithm, when comparing the first chromosome of a Macaca Mulata Monkey, of 8632 base pairs, against the first chromosome of a Gorilla, of 7665 base pairs; we ran the algorithm using from 1 to 8 cores (threads) and we replicate the experiment 5 times in order to obtain mean values.

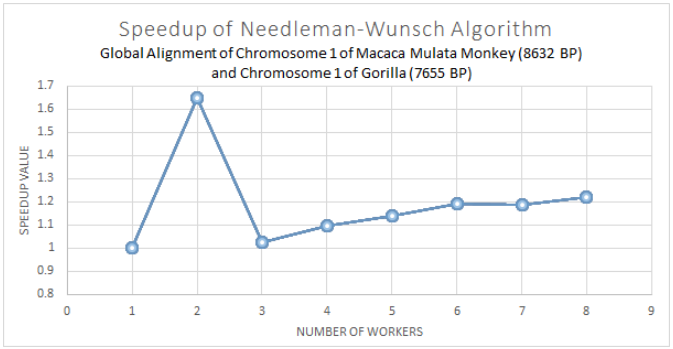

Fig. 3. Speedup of NW algorithm aligning the Chromosome 1 of a Macaca Mulata Monkey with the Chromosome 1 of a Gorilla.

It is interesting to observe that even though the results for all executions of the parallel algorithm are better than the original approach, the best result is obtained when using just 2 cores; in that case we obtain a speedup of about 1.6.

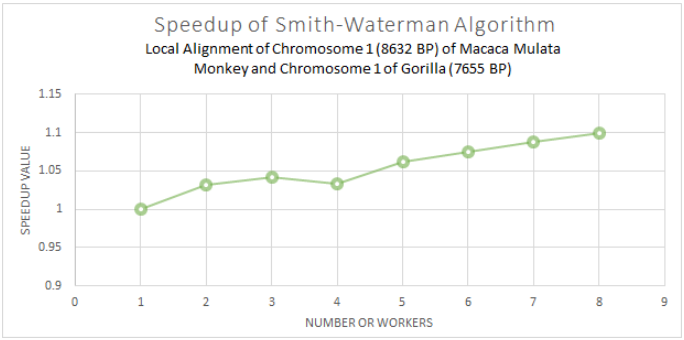

Fig. 4. Speedup of SW algorithm aligning the Chromosome 1 of a Macaca Mulata Monkey with the Chromosome 1 of a Gorilla.

In figure 4 we present the speedup obtained aligning the same two sequences of the last case, but instead of performing a global alignment we apply a local alignment using the Smith-Waterman algorithm. In this case we have a more expected result, where as the number of cores is increased so is the speedup; sadly the best result in this scenario is a very low speedup of about 1.1.

The next two images correspond to a different experiment. In this next case what we align is not DNA but protein sequences, wich are given in amino acids instead of nitrogenous bases. The sequences aligned are two classes of the same family of proteins called Myosin, which are involved in the muscular contraction needed for movement. They are

proteins of the same organism, called Trichiniella Spiralis, a parasite. Respectively the Myosin-4 has 1966 amino acids and the Myosin-3 has 1970 amino acids. The image 5 shows the speedup obtained when globally aligning the sequences.

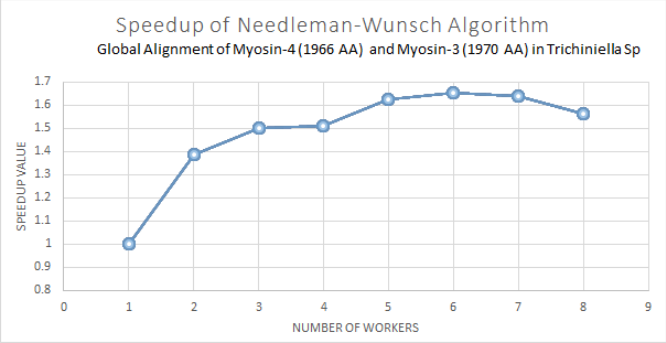

Fig. 5. Speedup of NW algorithm aligning the protein Myosin-3 of a parasite with the Myosin-4 protein of the same parasite.

It is here we we get the best results overall. When using 6 cores with our chunk-based approach we obtain about 1.65 of speedup in relation to the sequential algorithm. When using 7 and 8 cores the overhead of managing the threads (specially the semaphore synchronization) starts to overshadow the benefits of parallelizing, it is possible than with larger sequences this would tend to disappear.

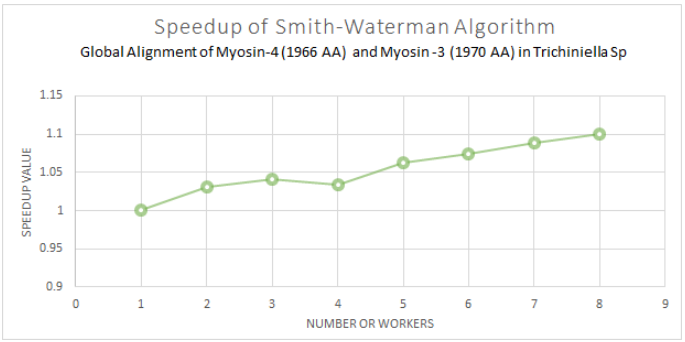

Fig. 6. Speedup of SW algorithm aligning the protein Myosin-3 of a parasite with the Myosin-4 protein of the same parasite.

The image 6 shows the speedup values when aligning the same two sequences of protein using the Smith-Waterman algorithm. Here, as in the last SW case as the cores are increased so is the speedup, but it still yields a very low value of about 1.1.

Even though the results of the experiments did not produced the performance improvements we were hoping, in all comparisons of the parallel algorithm vs the original sequential algorithm, the former was faster. We believe that exploiting parallelism in other parts of the method or using different ways for synchronizing the threads could yield better results.

VI. Related Work

There have been previous researches with the development of parallel processing of Biological Sequence, this researches are about comparison of the algorithms. In [14] the comparison is based in sequences that provides insight into molecular structure function, and homology, and it is increasingly important as the available data bases become larger and more numerous.

There are also forms of generation parallel programs from the Wavefront Design Pattern, this forms allowed to reduce the complexity of sequential programming, [15] describe a pattern-based parallel programming system that generates parallel programs from parallel design patterns.

In previous research using a general-purpose graphics processing unit, have helped develop accelerated version of the popular NCI-Blast like GPU-Blast, the speedup of this solution is range mostly between 3 and 4 [16].

By means of a dynamic programming method it is performed a parallel similarity search [17], this is a goal to implemented alignment of biological sequence data using multiple processors operating in parallel. Also the dynamic programming is used to a multi threaded parallel implementation that allowed the sequence comparison [12]. Fine-grain multi threading permits efficient parallelism exploitation in this application both by taking advantage of asynchronous point-to-point synchronizations and communication with low overheads and by effectively tolerating latency.

Within a benchmark is evaluated an ULTRA SPARC running at 167 MHz show a speedup factor of two compared to the same algorithm implemented with integer instructions on the same machine and the results show the performance reaches over 18 million matrix cells per second on a single processor, giving to our knowledge the fastest implementation of the Smith-Waterman algorithm on a workstation [18]

An implementation of the Smith-Waterman sequence-alignment algorithm using Single-Instruction, this implementation is based on the MMX and Streaming SIMD, for this in [19] the research shows a speed of more than 150 million cell updates per second was obtained on a single Intel Pentium III 500 MHz microprocessor. This had been probably the fastest implementation of this algorithm on a single general-purpose microprocessor.

VII. Future Work

As future work we propose implementing other approaches and running the same experiment as we did, to be able to compare statistically our results. One of this other approaches, that has not escapes our mind, is trying different ways of synchronization in our chunk-based solution. Currently, the thread management is mainly done via semaphores. Another option would be to use a *yield*-like solution. Meaning that a thread relinquish its CPU time directly, without having the semaphores as intermediary. For that there are multiple c primitives to aid the programmer in such circumstances such as: *pthread_yield* or *sched_yield* that causes the calling thread to relinquish the CPU to another thread; also we could use something such as *sleep* which makes sure that the calling thread will not run on the CPU for a given amount of time.

A question that immediately pops in our head is determining the right time to yield the processor? Is it when there is no

more work to do? Or should it be better to give up the CPU when a certain percentage of work has been completed? and therefore trying to buy time for dependent tasks to finish; we think that comparing this kind of strategies could, at the end, improve the algorithm's performance.

Replicating our experiments with other DNA, RNA or protein strands to double check our results.

There is also other researches that can be performed about parallelizing other parts of the algorithms such as the step for island searching in the Smith Waterman algorithm.

VIII. Conclusions

Our work indicates that is possible to obtain speedup on dynamic programming algorithms via parallel programming, even when there are clear dependencies among tasks. It is necessary to design an algorithm which tries to distribute work equally among threads and that has as little synchronization overhead as possible. Classifying different dynamic programming algorithms as chunk-based has helped us to develop our parallelization strategy. Both Needleman-Wunsch and Smith-Waterman algorithms fall into this category and both are relevant in the field of bioinformatics.

Preliminary results seem to suggest that tuning of the algorithm is required to achieve higher speedups. We think that assigning more strips to the threads (not just two) might improve the speedup measures, as that would reduce the threads idle times.

References

[1] Y. Liu, A. Wirawan, and B. Schmidt, "Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions," *BMC bioinformatics*, vol. 14, no. 1, p. 117, 2013.

[2] T. Rognes, "Faster smith-waterman database searches with inter-sequence simd parallelisation," *BMC bioinformatics*, vol. 12, no. 1, p. 221, 2011.

[3] S. A. Manavski and G. Valle, "Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment," *BMC bioinformatics*, vol. 9, no. 2, p. 1, 2008.

[4] S. A. Tahir Naveed, Imitaz Saeed Siddiqui, "Parallel needleman-wunsch algorithm for grid," pp. 1–6, 2005.

[5] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J. Mol. Biol.*, vol. 48, pp. 443–453, 1970.

[6] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[7] K.-M. Chao, W. R. Pearson, and W. Miller, "Aligning two sequences within a specified diagonal band," *Computer applications in the biosciences : CABIOS*, vol. 8, no. 5, pp. 481–487, 1992. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/8/5/481.abstract

[8] O. Gotoh, "Optimal alignment between groups of sequences and its application to multiple sequence alignment," *Computer applications in the biosciences : CABIOS*, vol. 9, no. 3, pp. 361–370, 1993. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/9/3/361.abstract

[9] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Commun. ACM*, vol. 18, no. 6, pp. 341–343, Jun. 1975.

[10] M. Dayhoff, R. Schwartz, and B. Orcutt, "A model of evolutionary change in proteins," in *Atlas of Protein Sequence and Structure*, M. Dayhoff, Ed. Washington, D. C.: National Biomedical Research Foundation, 1978, vol. 5, pp. 345–352.

[11] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks," *Proceedings of the National Academy of Sciences*, vol. 89, no. 22, pp. 10 915–10 919, Nov. 1992. [Online]. Available: http://dx.doi.org/10.1073/pnas.89.22.10915

[12] W. Martins, J. del Cuvillo, F. Useche, K. B. Theobald, and G. R. Gao, "A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison." in *Pacific Symposium on Biocomputing*, vol. 6, 2001, pp. 311–322.

[13] N. C. of Biotechnology Information. [Online]. Available: http://www.ncbi.nlm.nih.gov/

[14] E. W. Edmiston, N. G. Core, J. H. Saltz, and R. M. Smith, "Parallel processing of biological sequence comparison algorithms," *International Journal of Parallel Programming*, vol. 17, no. 3, pp. 259–275, 1988.

[15] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, and K. Tan, "Generating parallel programs from the wavefront design pattern," in *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*. IEEE, 2001, pp. 8–pp.

[16] R. Hughey, "Parallel hardware for sequence comparison and alignment," *Computer applications in the biosciences: CABIOS*, vol. 12, no. 6, pp. 473–479, 1996.

[17] A. R. Galper and D. L. Brutlag, *Parallel similarity search and alignment with the dynamic programming method*. Knowledge Systems Laboratory, Medical Computer Science, Stanford University, 1990.

[18] A. Wozniak, "Using video-oriented instructions to speed up sequence comparison," *Computer applications in the biosciences: CABIOS*, vol. 13, no. 2, pp. 145–150, 1997.

[19] T. Rognes and E. Seeberg, "Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors," *Bioinformatics*, vol. 16, no. 8, pp. 699–706, 2000.