Fast Biological Sequence Comparison on Hybrid Platforms

Safia Kedad-Sidhoum, Fernando Mendonca, Florence Monna, Gregory Mounié, Denis Trystram

Fast Biological Sequence comparison on Hybrid Platforms

Safia Kedad-Sidhoum[1], Fernando Mendonca[2], Florence Monna[1,2], Gregory Mounié[2], Denis Trystram[2, 3]

[1]*Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France,*
[2]*University of Grenoble-Alpes,*
[3]*Institut Universitaire de France*

safia.kedad-sidhoum@lip6.fr, fernando.machado-mendonca@inria.fr,
{florence.monna,gregory.mounie,denis.trystram}@imag.fr

***Abstract*—Today, many high performance computing platforms use hybrid architectures combining multi-core processors and hardware accelerators like GPUs (Graphic Processing Units). This paper presents a new method for scheduling tasks for biological sequence comparison applications with CPUs and GPUs. This strategy is called SWDUAL and is based on a dual approximation scheme for determining which tasks are most suitable to be executed on the GPUs. The objective is to obtain fast execution time and minimize the idle time on each PE (Processing Element). It is implemented using a master-slave model. Results obtained when sequences were compared to five public genomic databases show that this method allows to reduce the execution time on hybrid platforms when compared to other public available implementations.**

***Keywords*-bioinformatics; sequence comparison; hybrid platforms; task scheduling; dual approximation**

I. Introduction

We are interested in this paper in an efficient implementation of the classical problem of comparing biological sequences in a parallel multi-core platform with hardware accelerators.

Once a new biological sequence is discovered, its functional/structural characteristics must be established. In order to do that, the newly discovered sequence is compared against other sequences, looking for similarities. Sequence comparison is, therefore, one of the most crucial operations in Bioinformatics [1]. The most accurate algorithm to execute pairwise comparisons is the one proposed by Smith-Waterman (denoted by SW in short) [2], which is based on dynamic programming and run in quadratic time and space complexity in the length of the sequences. This can easily lead to very large execution times and huge memory requirements, since the size of biological databases is growing exponentially.

Parallel implementations can be used to compute results faster, reducing significantly the time needed to obtain results with the SW algorithm. Indeed, many proposals do exist to execute SW on clusters [3], [4] and computational grids [5]. More recently, hardware accelerators such as GPUs (Graphics Processing Units) and FPGAs (Field Programmable Gate Arrays) have been explored to speed-up the SW algorithm [6]–[8]. In addition to that, SIMD extensions of general-purpose processors, such as the Intel SSE, have also been explored to accelerate SW implementations [9].

Since accelerators are usually connected to a multi-core host, the idea is to use both the accelerators and the CPUs to execute SW in parallel. There are some approaches in the literature that explore such idea [10]–[12]. In order to distribute work among the hybrid processing elements, these approaches usually assume that multi-cores and accelerators have the same processing power [11], distribute work proportionally, considering the theoretical computing power of each processing element [12] or assign one work unit at the time [10] in a Self-Scheduling strategy.

In this paper, we propose SWDUAL, a new implementation of the Smith-Waterman algorithm on hybrid platforms composed of multiple processors and multiple GPUs. SWDUAL is based on a fast dual approximation scheduling algorithm that selects the most suitable tasks to be run on the GPUs while keeping a good balance of the computational load over the whole platform [13].

Given a set of query sequences and a biological database, our strategy uses a one round master-slave approach to assign tasks to the processing elements according to the dual approximation scheduling algorithm being used.

The remainder of this paper is organized as follows. Section II presents the sequence comparison problem and recalls the principle of the classical SW algorithm. The strategy and its implementation for executing SW on hybrid platforms are proposed respectively in Sections III and IV. Section V presents the experimental results. Finally, Section VI concludes the paper.

II. Biological Sequence Comparison

A. Presentation of the core problem

A biological sequence is a structure composed of nucleic acids or proteins. It is represented by an ordered list of residues, which are nucleotide bases (for DNA or RNA sequences) or amino acids (for protein sequences).

DNA and RNA sequences are treated as strings composed of elements of the alphabets $\sum = \{A, T, G, C\}$ and $\sum = \{A, U, G, C\}$, respectively. Protein sequences are also

treated as strings which elements belong to an alphabet with, normally, 20 amino acids.

Since two biological sequences are rarely identical, the sequence comparison problem corresponds to approximate pattern matching. To compare two sequences, a good alignment between each other should be determined. This corresponds to place one sequence above the other making clear the correspondence between similar characters [1]. In an alignment, some gaps (space characters) can be inserted in arbitrary locations such that the sequences end up with the same size.

Given an alignment between sequences s and t, a score is associated to it as follows. For each two bases in the same column:

- a punctuation *ma* is associated if both characters are identical (*match*);
- a penalty *mi*, if the characters are different (*mismatch*);
- a penalty *g*, if one of the characters is a gap.

The score is obtained by the addition of all these values. The maximal score is called the similarity between the sequences. Figure 1 presents one possible global alignment between two DNA sequences and its associated score. In this example, $ma = +1$, $mi = -1$ and $g = -2$.

A	C	T	T	G	T	C	C	G
A	$-$	T	T	G	T	C	A	G
$+1$	-2	$+1$	$+1$	$+1$	$+1$	$+1$	-1	$+1$

$score = 4$

Figure 1. Example of an alignment and score

B. Smith-Waterman (SW) Algorithm

The SW algorithm [2] is an exact method based on dynamic programming to obtain the optimal pairwise local alignment in quadratic time and space in the length of the sequences.

The first phase of the SW algorithm starts by two input sequences s and t, with $|s| = m$ and $|t| = n$, where $|s|$ is the size of sequence s. The similarity matrix is denoted by $H_{m+1,n+1}$, where $H_{i,j}$ contains the score between prefixes $s[1..i]$ and $t[1..j]$. At the beginning, the first row and column are filled with zeros. The remaining elements of H are obtained from Equation (1). In addition, each cell $H_{i,j}$ contains the information about the cell that was used to produce the value. $S_{i,j}$ is a similarity score for the elements i and j

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + S_{i,j} \\ H_{i,j-1} + \mathrm{g} \\ H_{i-1,j} + \mathrm{g} \\ 0 \end{cases} \quad (1)$$

The SW algorithm assigns a constant cost to gaps. Nevertheless, in nature, gaps tend to appear in groups. For this reason, a higher penalty is usually associated to the first gap and a lower penalty is given to the following ones (this is known as the affine-gap model). Gotoh [14] proposed an algorithm based on SW that implements the affine-gap model by calculating three Dynamic Programming (DP) matrices, namely H, E and F, where E and F keep track of gaps in each of the sequences. The gap penalties for starting and extending a gap are G_s and G_e, respectively. This recursion formulas are given by Equations (2), (3) and (4).

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + S_{i,j} \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} \quad (2)$$

$$E_{i,j} = -G_e + \max \begin{cases} E_{i,j-1} \\ H_{i,j-1} - G_s \end{cases} \quad (3)$$

$$F_{i,j} = -G_e + \max \begin{cases} F_{i-1,j} \\ H_{i-1,j} - G_s \end{cases} \quad (4)$$

C. Parallelizing SW

There are several ways to parallelize the SW algorithm. The following paragraph describes the comparison of a set q of m query sequences $(q_1, q_2, ..., q_m)$ to a set d of n database sequences $(d_1, d_2, ..., d_n)$. It is assumed that the size of the database is much larger than the set of query sequences ($m \ll n$).

In the fine-grained approach, the comparison of one query sequence and one database sequence (i.e. a single SW execution) is done by several Processing Elements (PEs). The data dependency in the matrix calculation is non-uniform, and the calculations that can be done in parallel evolve as waves on diagonals (according to Equation (2)). Figure 2 illustrates a fine-grained column-based block partition technique with four PEs. At the beginning, only p_0 is computing. When p_0 finishes calculating the values of a block of matrix cells, it sends its border column to p_1, that can start calculating and so on. Note that this solution may be unbalanced: very close to the end of the matrix computation, only p_3 is calculating. When the PEs finish to compare q_1 to d_1, they start comparing q_1 to d_2 and so on, until the comparison of q_m to d_n is completed.

In the very coarse-grained approach, each PE compares a different query sequence to the whole database (see Figure 3). For instance, p_0 compares q_1 to d, p_1 compares q_2 to d and so on. Note that, in this case, the number of SW comparisons executed by each processing element is big and this approach can easily lead to load imbalance.

The SWDUAL implementation uses both the fine-grained and very coarse-grained approach. Each one of the workers

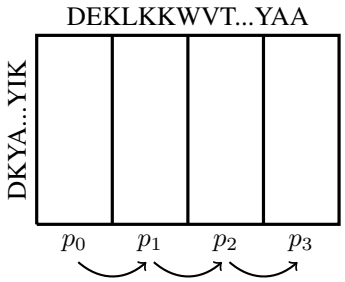

Figure 2. Fine-grained strategy to parallelize the SW algorithm

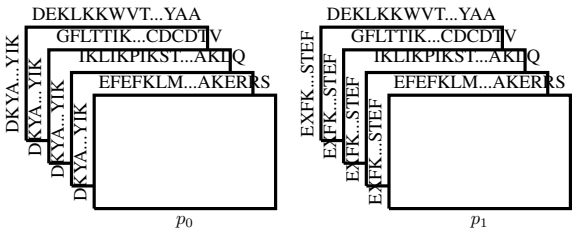

Figure 3. Very coarse-grained strategy to parallelize the SW algorithm

uses fine-grained approach to accelerate the execution of the SW algorithm for a particular comparison. That approach is dependent on the type of worker and the techniques being used to optimize each comparison. At the same time, other workers are comparing other sequences of the query set to the database in the same way. In our case the master uses the scheduling algorithm to allocate tasks to the workers. Each task is equivalent to the comparison of one task of the query set to the whole database.

In the following section, we describe the scheduling algorithm used by the master to allocate the tasks to the workers.

III. Scheduling Algorithm with Dual Approximation

In the implementation targeted in this paper, the tasks are pairwise comparisons of two sequences. The problem is to determine an allocation of the tasks to the GPUs that minimizes the global completion time (called *makespan*).

The principle of the proposed scheduling algorithm is to use the dual approximation technique introduced in [15] and which is recalled as follows. A g-dual approximation algorithm for any minimization problem takes a real number λ (called the guess) as an input and either delivers a schedule whose makespan is at most $g\lambda$ or answers correctly that there exists no schedule of length at most λ.

We target $g = 2$. Let λ be the current real number input for the dual approximation. In the following, we assert that there exists a schedule of length lower than λ. Then, we have to show how it is possible to build a schedule of length at most 2λ.

We introduce an allocation function $\pi(j)$ of a task T_j which corresponds to the processor where the task is processed. The set $\mathcal{C}$ (resp. $\mathcal{G}$) is the set of all the CPUs (resp. GPUs). Therefore, if a task T_j is assigned to a CPU, we can write $\pi(j) \in \mathcal{C}$. Each task T_j has two processing times, $\overline{p_j}$ if it is processed on a CPU, $\underline{p_j}$ if it is processed on a GPU. We define W_C as being the computational area of the CPUs on the Gantt chart representation of a schedule, i.e. the sum of all the processing times of the tasks allocated to the CPUs: $W_C = \sum_{j \,/\, \pi(j) \in \mathcal{C}} \overline{p_j}$.

To take advantage of the dual approximation paradigm, we have to make explicit the consequences of the assumption that there exists a schedule of length at most λ. We state below some basic properties of such a schedule:

- The execution time of each task is at most λ.
- The computational area on the CPUs is at most $m\lambda$.
- The computational area on the GPUs is at most $k\lambda$.

We are looking for an assignment of the tasks to either a CPU or a GPU satisfying the following two constraints:

- (C1) The total computational area W_C on the CPUs is at most $m\lambda$.
- (C2) The total computational area on the GPUs is lower than $k\lambda$.

We define for each task T_j a binary variable x_j such that $x_j = 1$ if T_j is assigned to a CPU or 0 if T_j is assigned to a GPU. Determining if an assignment satisfying (C_1) and (C_2) exists corresponds to solving a minimization knapsack problem [16] that can be formulated as follows:

$$W_C^* = \min \sum_{j=1}^{n} \overline{p_j} x_j \tag{5}$$

$$\text{s.t.} \sum_{j=1}^{n} \underline{p_j} (1 - x_j) \leqslant k\lambda \tag{6}$$

$$x_j \in \{0, 1\} \quad \forall j = 1, \ldots, n \tag{7}$$

Equation (5) represents the minimal workload on all the CPUs. Constraint (6) imposes an upper bound on the computational area of the GPUs which is $k\lambda$ (cf. (C2)).

The knapsack is solved by a greedy algorithm. Usually the knapsack is a maximization problem. Here we consider the opposite minimization version. The tasks are sorted by decreasing order of the ratio $\frac{\overline{p_j}}{\underline{p_j}}$. Thus, the most prioritary tasks are those with the best relative processing times on GPUs. Figure 4 depicts the principle of the greedy knapsack. The knapsack allocates the first tasks to the the GPUs until the computational area on the GPUs is roughly equal to $k\lambda$.

The result of the knapsack leads to a solution with a computational area on GPUs larger than $k\lambda$. All remaining tasks are scheduled on CPUs. If the value of the computational area on the CPUs is greater than $m\lambda$, then there exists

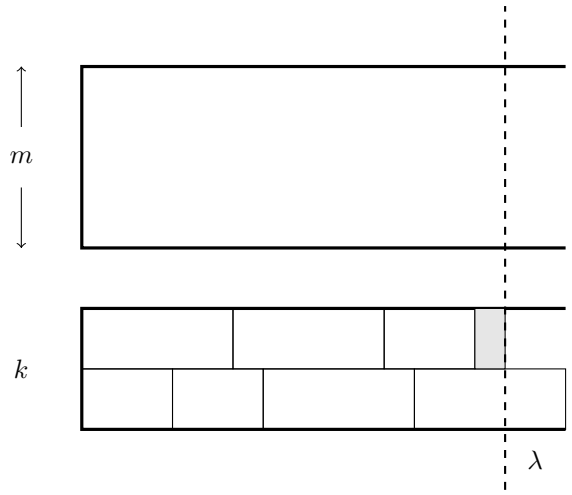

Figure 4. Greedy knapsack fills the GPUs with tasks up to getting a computational area larger than $k\lambda$ on the GPUs

no solution with a makespan at most λ, and the algorithm answers "NO" to the dual approximation.

The scheduling on the CPUs after the allocation of the greedy knapsack is done with a list scheduling algorithm assigning the tasks on an available processor of the corresponding type in the assignment (cf. Figure 5).

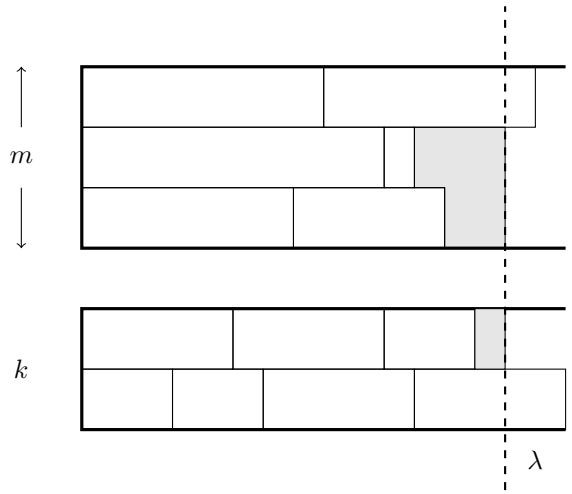

Figure 5. List scheduling fills the CPUs with remaining tasks. The computational area is smaller than $m\lambda$, otherwise λ is smaller than C^*_{max}

Proposition 1. *If W_C is lower than $m\lambda$, there exists a feasible solution with a makespan at most 2λ.*

Proof: The makespan on the CPUs, C^{CPU}_{max}, is bounded by the following inequality:

$$C^{CPU}_{max} \leqslant \max_{1\leqslant j\leqslant n}(\overline{p_j}x_j) + \frac{\sum_{j=1}^{n}\overline{p_j}x_j}{\sum_{j=1}^{n}x_j} \tag{8}$$

All the tasks assigned to the CPUs have a processing time lower than λ, therefore $\max_{1\leqslant j\leqslant n}\overline{p_j}x_j \leqslant \lambda$ and $\sum_{j=1}^{n}\overline{p_j}x_j \leqslant m\lambda$ with the hypothesis that W_C is lower than $m\lambda$. We obtain

$$C^{CPU}_{max} \leqslant \left(1 + \frac{m}{\sum_{j=1}^{n}x_j}\right)\lambda \tag{9}$$

Moreover, we can assume $\sum_{j=1}^{n}x_j > m$, otherwise the optimal solution is straightforward (one task per CPU), thus

$$C^{CPU}_{max} \leqslant 2\lambda \tag{10}$$

Let us turn now to the GPU side. Let $jlast$ be the index of the last task selected by the knapsack. The task $jlast$ is thus the last task scheduled by the greedy knapsack on the GPUs. Hence, task $jlast$ has no influence at all on the scheduling of all the other tasks.

Two cases hold (cf. Equation (11)): either the $jlast$ task is not the last to be completed or it is. On the first hand, $jlast$ can be removed from the schedule instance without changing the makespan. The computational area of all tasks except $jlast$ is smaller than $k\lambda$ thus the guarantee is the same as the one derived for the CPU schedule. On the second hand, the computational area of all tasks save $jlast$ is also smaller than $k\lambda$ thus, when the list algorithm schedules the $jlast$ task, the least loaded of the k GPUs is loaded less than λ. Hence the $jlast$ task ends before 2λ.

$$C^{GPU}_{max} \leqslant \begin{cases} \max_{1\leq j\leq n|j\neq jlast}\left(\underline{p_j}(1-x_j)\right) + \frac{\sum_{j=1}^{n}\underline{p_j}(1-x_j)-\underline{p_{jlast}}}{k} \leqslant 2\lambda \\ \left(\underline{p_j}(1-x_j)\right) + \frac{\sum_{j=1}^{n}\underline{p_j}(1-x_j)-\underline{p_{jlast}}}{k} \leqslant 2\lambda \end{cases} \tag{11}$$

Since the makespan of the schedule is the maximum of the makespans on the CPUs and on the GPUs, we get

$$C_{max} \leqslant 2\lambda \tag{12}$$

■

We have described one step of the dual-approximation algorithm, with a fixed guess. A binary search will be used to try different guesses to approach the optimal makespan as follows.

Binary Search. We first take an initial lower bound B_{min} and an initial upper bound B_{max} of our optimal makespan.

We start by solving the problem with λ equal to the average of these two bounds and then we adjust the bounds:

- If the previous algorithm returns "NO", then λ becomes the new lower bound.
- If the algorithm returns a schedule of makespan at most 2λ, then λ becomes the new upper bound.

The number of iterations of this binary search can be bounded by $\log(B_{max} - B_{min})$.

Cost Analysis. The greedy algorithm used to fill the GPUs only requires a sorting of the tasks, whereas the list scheduling used for the CPUs is linear. Therefore, the time complexity of each step of the binary search is $\mathcal{O}(n \log(n))$.

The algorithm described above returns a schedule with a makespan equal to at most twice the optimal makespan. Some constraints on the number of tasks with processing times larger than $\frac{2}{3}\lambda$, λ being the current guess, in the algorithm can be added to the original problem. The resolution of the knapsack problem with these additional constraints via dynamic programming can reduce the makespan of the schedule returned by the algorithm to $\frac{3}{2}OPT$, where OPT is the optimal makespan. This method is described in [13], and has a time complexity in $\mathcal{O}\left(n^2mk^2\right)$ per step of the binary search. This time complexity is important, but it can be lowered with special instances where all the considered tasks are accelerated when assigned to a GPU, which is the case for the sequence comparison problem addressed in this paper. In this special case, the time complexity reduces to $\mathcal{O}(mn \log(n))$, which is satisfactory for real implementations.

IV. Designing the SWDUAL implementation

Our implementation is designed using the master-slave model. The master is responsible for receiving commands from the user, reading the sequences from disk, generating a list of tasks and allocating them to the workers (slaves), receiving and presenting the results back to the user. The workers first have to register themselves with the master. Then, acquire the same sequences that master received as parameters from the user, receive tasks from the master, execute them and return the results. Both the master and workers convert the format of the sequences if necessary. Figure 6 shows the different steps taken during execution by the master and the workers.

First, the master processes the command line arguments entered by the user. Then, it loads the sequences, converts the format if necessary and waits for the workers to connect. The workers are started either manually or automatically, connect to the master, load the sequences and also if necessary convert the format.

Now, the master can use the information gathered from the workers and the allocation policy or scheduling algorithm to allocate tasks to the workers, after which the workers start executing them. That can be done only once at the beginning of the execution or iteratively until all tasks are executed. Finally, the workers send the results to the master that merge and present them to the user.

Sequence database files created using the Fasta [17] format are in fact text files, with sequences placed one after the other. For that reason, it is not feasible to read specific sequences contained in the file, which is important for implementations like SWDUAL.

To improve this reading process, a simple binary format was created with a few additional fields. Using this format, both the master and workers are able to read sequences in any position inside the file, directly. Additionally, the memory allocation process is simplified due to the fact that the all the sequences sizes are known beforehand.

V. Experimental Results

In this section, the experimental results of the method implementation are presented and compared to the state-of-the-art.

The method proposed in Section III was implemented in C++ with SSE extensions and CUDA.

The strategy was implemented in C with SSE extensions and CUDA, and it integrates CUDASW++ 2.0 [7] and SWIPE [9] into the code. That code was compiled with the CUDA SDK 4.2.9 and gcc 4.5.2. The operating system used was Linux 3.0.0-15 Ubuntu 64 bits. The tests were conducted with 40 real query sequences of minimum size 100 and maximum size 5,000 amino acids, which were compared to 5 real genomic databases: Uniprot with 537,505 sequences, Enbembl Dog with 25,160 sequences and Rat with 32,971 sequences and RefSeq Human with 34,705 sequences and Mouse 29,437 sequences.

The tests were executed in the Idgraf high performance computer located at Inria Grenoble. It contains 2 Intel Xeon 2.67GHz processors with 4 cores each, 74GB of RAM and 8 Nvidia Tesla C2050 GPUs. The machine was reserved for exclusive use for the duration of the test to ensure that no other major process was running concurrently. All the sequences used were available locally to minimize the influence of the network and file reading time. All combinations of programs, number of workers, query and database sequences were executed twenty-five times and the average total wall-clock execution time was recorded. Also, processor affinity was used to ensure that each process stayed in the same processor during the whole execution.

A. Comparison to other implementations

Table I shows the state-of-the-art implementations that were compared to SWDUAL, as well as their version number and command line options. For the commands, the variables were $T for the number of threads, $Q query sequence and $D database sequence.

The SWDUAL implementation was compared against SWIPE, STRIPED, SWPS3 and CUDASW++.

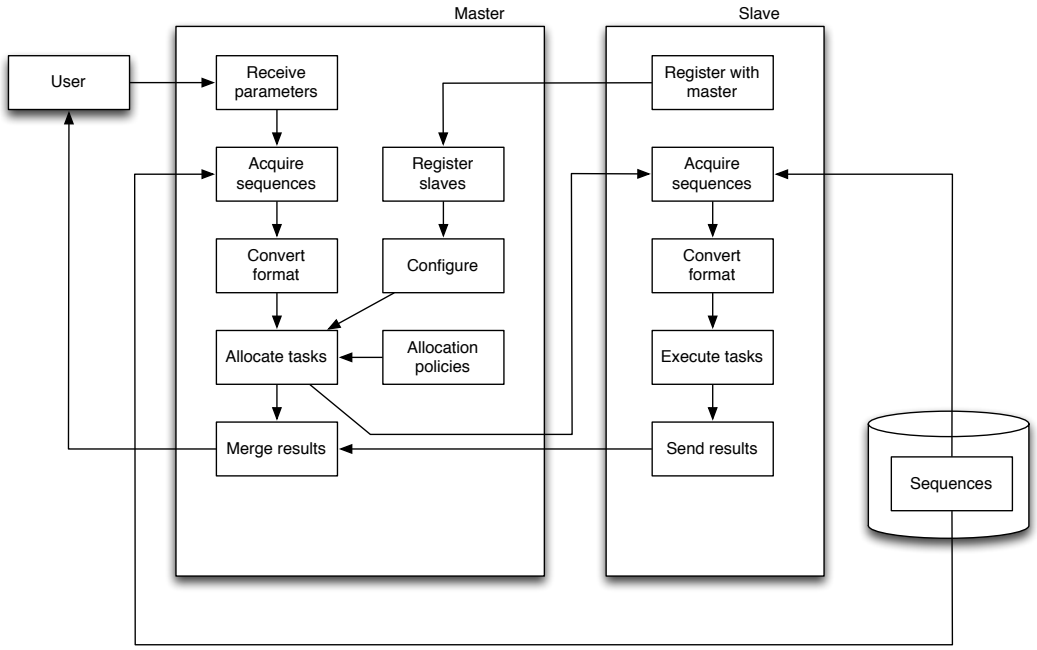

Figure 6. SWDUAL master-slave model

Table I
APPLICATIONS INCLUDED IN THE COMPARISON

Application	Version	Command line
SWIPE	1.0	./swipe -a $T -i $Q -d $D
STRIPED		./striped -T $T $Q $D
SWPS3	20080605	./swps3 -j $T $Q $D
CUDASW++	2.0	./cudasw -use_gpus $T -query $Q -db $D

SWIPE [9] was written mostly in C++ with some parts hand coded in assembly. It was compiled using the provided Makefile.

The source code for the Farrar's STRIPED implementation of the SW algorithm [18] was compiled using the provided Makefile. It was written mainly in C with some parts also coded in assembly or Intel intrinsics.

SWPS3 [19] was downloaded from the author's website and was written in C. It was compiled using the provided Makefile.

CUDASW++ 2.0 [7] was also downloaded from the author's website and was written in C++ and CUDA. It was compiled using the provided Makefile. CUDA 4.1 was used in the compilation.

The tests were conducted using the UniProt database (*www.uniprot.org*) and 40 query sequences taken from it. Also, were used on the test up to four CPUs and four GPUs. For that reason the considered applications were executed with up to four workers, while SWDUAL, that uses both CPUs and GPUs as workers was executed with workers between two and eight. In this case, the first four workers used on the SWDUAL execution were GPUs and the last four workers were CPUs.

The reason why only four CPUs and four GPUs were used in this test although eight CPUs and eight GPUs were available is that each GPU worker actually needs some CPU time to execute as fast as it can. As a consequence, using more CPUs and GPUs than that number impacts on the overall performance of the applications and the speedup is considerably worst. Thus for the applications that only used CPUs or GPUs up to four workers were used. The exception was our case that was executed with four GPUs and four CPUs for a total of eight workers. In this case, since our implementation needs at least one CPU and one GPU to execute, we start with two workers. For three workers, two are GPUs and one is a CPU. Finally, an execution of four workers uses three GPUs and one CPU.

The SWDUAL implementation was able to significantly reduce the execution time of the sequence database searches using the Smith-Waterman algorithm compared to earlier proposals that use only one type of processing element. As can be seen on Figure 7 and Table II, the combination of CPUs and GPUs leaded to very good results. When executing with two workers, SWDUAL showed a reduction of 54.7%, 85% and 98% when compared to the same execution on SWIPE, STRIPED and SWPS3, respectively. When executing with four workers, a reduction of 55.3% was obtained when compared to the execution on SWIPE, 73.5% when compared to STRIPED and 98.6% on SWPS3.

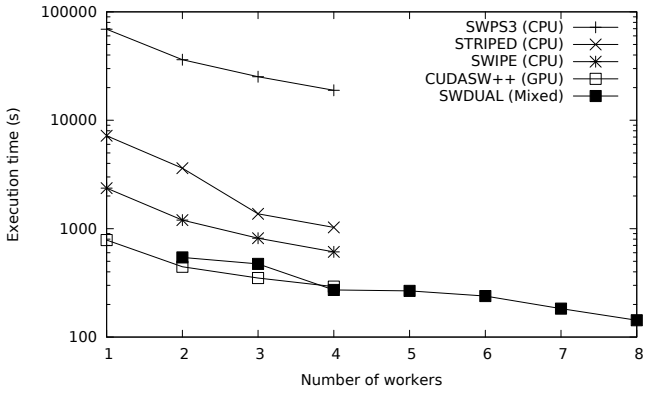

Figure 7. Execution times in seconds for the compared implementations

Also, due to the implementation of the dual approximation scheduling algorithm, the execution on each of the processing elements finished with almost no idle time.

Table II
EXECUTION TIMES FOR THE COMPARED IMPLEMENTATIONS

Application	Number of workers			
	1	2	3	4
SWPS3	69208.2	36174.09	25206.563	18904.31
STRIPED	7190	3615.38	1369.33	1027.28
SWIPE	2367.24	1199.47	816.61	610.23
CUDASW++	785.26	445.611	350.09	292.157
SWDUAL		543.28	472.84	271.98
Application	Number of workers			
	5	6	7	8
SWDUAL	266.69	239.04	183.12	142.98

B. Comparison to 5 genomic databases

In this case, the tests were conducted with 40 real query sequences of minimum size 100 and maximum size 5,000 amino acids, which were compared to 5 real genomic databases, as in [7]: Uniprot with 537,505 sequences (*www.uniprot.org*), Enbembl (*www.ensembl.org*) Dog with 25,160 sequences and Rat with 32,971 sequences and RefSeq (*www.ncbi.nlm.nih.gov/RefSeq*) Human with 34,705 sequences and Mouse 29,437 sequences as listed in Table III.

Table III
GENOMIC DATABASES USED ON THE TESTS

Database	Number of database seqs	Smallest query seq	Longest query seq
Ensembl Dog Proteins	25,160	100	4,996
Ensembl Rat Proteins	32,971	100	4,992
RefSeq Human Proteins	34,705	100	4,981
RefSeq Mouse Proteins	29,437	100	5,000
UniProt	537,505	100	4,998

In order to measure the benefits of using a hybrid platform, the wall-clock execution time and GCUPs (billion cell updates per second) obtained were measured when comparing 40 query sequences to the five genomic databases.

As can be seen on Table IV, SWDUAL was able to obtain good speedups while combining CPUs and GPUs, reducing the execution time repeatedly while adding processing elements. For the Uniprot database the execution time was reduced from 543 seconds (approximately 10 minutes) to 86 seconds when executing on eight CPUs and eight GPUs. Figure 8 shows the execution times obtained when comparing the databases.

Table IV
RESULTS RUNNING ON GPUS AND CPUS

Workers	2 Time (s) GCUPS	4 Time (s) GCUPS	8 Time (s) GCUPS
Ensembl Dog	78.36 18.91	39.63 37.39	20.45 72.45
Ensembl Rat	75.85 22.97	37.97 45.89	20.17 86.38
RefSeq Mouse	84.40 18.99	46.25 34.66	23.59 67.95
RefSeq Human	95.09 20.70	48.01 41.00	24.82 79.31
Uniprot	543.28 35.81	271.98 71.53	142.98 136.06

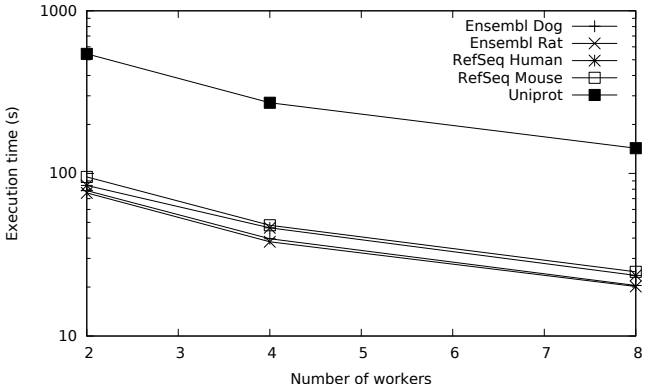

Figure 8. Execution times for the compared databases

C. Comparison of homogeneous and heterogeneous sets

For this test, two additional query sets were created from the Uniprot database. Each query set have, like in the previous tests, 40 sequences. In this case, the sequences in the homogeneous set range in size from 4500 to 5000 and the ones in the heterogeneous set have sizes between 4 (the smallest sequence in the database) and 35213 (the largest sequence in the database).

The idea is to verify that the allocation strategy and the application as a whole is equally able to work with

sequences, and therefore tasks, that are similar in terms of size as well as tasks with very different sizes.

Table V shows the execution times and the GCUPs obtained when comparing these two sets to the UniProt database. In this case, SWDUAL was able to achieve good performance on both sets. Figure 9 also shows the results obtained in these comparisons.

Table V
RESULTS RUNNING THE HOMOGENEOUS AND THE HETEROGENEOUS SETS

Sets	2 Time (s) GCUPS	4 Time (s) GCUPS	8 Time (s) GCUPS
Heterogeneous	3554.36 37.55	1785.73 74.74	908.45 146.92
Homogeneous	998.27 36.3	484.74 74.76	249.69 145.14

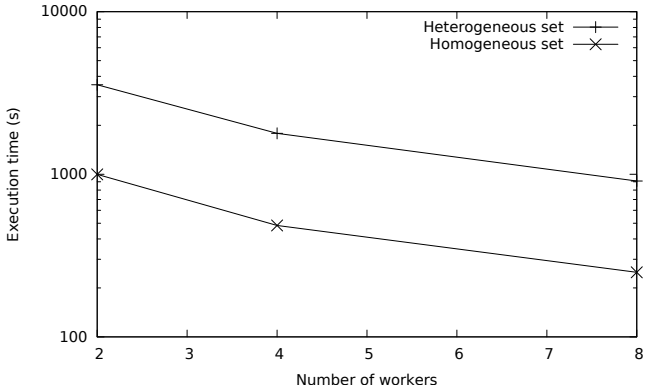

Figure 9. Execution times for the heterogeneous and homogeneous sets

VI. CONCLUSION

Efficient parallelization of the Smith-Waterman algorithm using SIMD and SIMT on standard hardware enables sequence database searches to be performed much faster than before. Also, scheduling algorithms like the dual approximation and the master-slave model allow for better utilization of the resources by combining the processing elements available in hybrid platforms.

In our new implementation, the comparison of a given database to a set of query sequences is divided amongst any number of workers. The dual approximation algorithm was used to decide which tasks to execute on the GPUs. The objective is to achieve good execution times and have as little idle time on the processing elements as possible.

When comparing 40 query sequences to the UniProt database a speed of 225 billion cell updates per second (GCUPS) was achieved on a dual Intel Xeon processor system with Nvidia Tesla GPUs, reducing the execution time from 543 seconds to 86 seconds. In addition to that, the combination of GPUs and CPUs was responsible for reducing the execution time to a total of 142 seconds for that database, which is faster than all the compared implementations.

Finally, we showed that SWDUAL is able to reduce execution times in sequence database comparisons both when tasks have similar sizes and sizes that are very different between tasks.

ACKNOWLEDGMENT

This paper was partly funded by CAPES Foundation, Ministry of Education, Brazil.

REFERENCES

[1] D. W. Mount, "Sequence and genome analysis," *Bioinformatics: Cold Spring Harbour Laboratory Press: Cold Spring Harbour*, vol. 2, 2004.

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

[3] S. Rajko and S. Aluru, "Space and time optimal parallel sequence alignments," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 15, no. 12, pp. 1070–1081, 2004.

[4] A. Boukerche, A. C. M. A. de Melo, E. F. de Oliveira Sandes, and M. Ayala-Rincon, "An exact parallel algorithm to compare very long biological sequences in clusters of workstations," *Cluster Computing*, vol. 10, no. 2, pp. 187–202, 2007.

[5] C. Chen and B. Schmidt, "An adaptive grid implementation of dna sequence alignment," *Future Generation Computer Systems*, vol. 21, no. 7, pp. 988–1003, 2005.

[6] E. F. de O Sandes and A. C. M. A. de Melo, "Smith-waterman alignment of huge sequences with gpu in linear space," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 1199–1211.

[7] Y. Liu, B. Schmidt, and D. L. Maskell, "Cudasw++ 2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions," *BMC research notes*, vol. 3, no. 1, p. 93, 2010.

[8] X. Jiang, X. Liu, L. Xu, P. Zhang, and N. Sun, "A reconfigurable accelerator for smith–waterman algorithm," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 54, no. 12, pp. 1077–1081, 2007.

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

[10] A. Singh, C. Chen, W. Liu, W. Mitchell, and B. Schmidt, "A hybrid computational grid architecture for comparative genomics," *Information Technology in Biomedicine, IEEE Transactions on*, vol. 12, no. 2, pp. 218–225, 2008.

[11] J. Singh and I. Aruni, "Accelerating smith-waterman on heterogeneous cpu-gpu systems," in *Bioinformatics and Biomedical Engineering,(iCBBE) 2011 5th International Conference on*. IEEE, 2011, pp. 1–4.

[12] X. Meng and V. Chaudhary, "A high-performance heterogeneous computing platform for biological sequence analysis," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, no. 9, pp. 1267–1280, 2010.

[13] S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram, "Scheduling independent tasks on multi-cores with gpu accelerators," in *In 11th HeteroPar 2013, International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms, in conjuction with the Euro-Par 2013 conference*, Aachen, Germany, Aug 2013.

[14] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of molecular biology*, vol. 162, no. 3, pp. 705–708, 1982.

[15] D. S. Hochbaum and D. B. Shmoys, "Using dual approximation algorithms for scheduling problems theoretical and practical results," *J. ACM*, vol. 34, no. 1, pp. 144–162, 1987.

[16] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, 1st ed. John Wiley & Sons, 1990, wiley Series in Discrete Mathematics and Optimization.

[17] W. R. Pearson, "Rapid and sensitive sequence comparison with fastp and fasta," *Methods in enzymology*, vol. 183, pp. 63–98, 1990.

[18] M. Farrar, "Striped smith–waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[19] A. Szalkowski, C. Ledergerber, P. Krähenbühl, and C. Dessimoz, "Swps3–fast multi-threaded vectorized smith-waterman for ibm cell/be and× 86/sse2," *BMC Research Notes*, vol. 1, no. 1, p. 107, 2008.