2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

Biological Sequence Comparison on Hybrid Platforms with Dynamic Workload Adjustment

Fernando Machado Mendonca
Department of Computer Science
University of Brasilia, UnB
Brasilia, Brazil
fmendonca@cic.unb.br

Alba Cristina Magalhaes Alves de Melo
Department of Computer Science
University of Brasilia, UnB
Brasilia, Brazil
albamm@cic.unb.br

***Abstract*—This paper proposes and evaluates a strategy to run Biological Sequence Comparison applications on hybrid platforms composed of GPUs and multicores with SIMD extensions. Our strategy provides multiple task allocation policies and the user can choose the one which is more appropriate to his/her problem. We also propose a workload adjustment mechanism that tackles situations that arise when slow nodes receive the last tasks. The results obtained comparing query sequences to 5 public genomic databases in a platform composed of 4 GPUs and 2 multicores show that we are able to reduce the execution time with hybrid platforms, when compared to the GPU-only solution. We also show that our workload adjustment technique can provide significant performance gains in our target platforms.**

***Keywords*-bioinformatics; smith-waterman; GPUs; multicores;**

I. Introduction

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 basic operations in Bioinformatics [1]. The most accurate algorithm to execute pairwise comparisons is the one proposed by Smith-Waterman (SW) [2], which is based on dynamic programming, with quadratic time and space complexity. This can easily lead to very high execution times and huge memory requirements, since biological databases are growing exponentially.

Parallel processing can be used to produce results faster, reducing significantly the time needed to obtain results with the SW algorithm. Indeed, many proposals do exist to execute SW on clusters [9] [3] and grids [7]. More recently, accelerators such as GPUs (Graphics Processing Units) and FPGAs (Field Programmable Gate Arrays) have been explored to execute SW [4] [6] [8]. In addition to that, the SIMD extensions of general-purpose processors, such as the Intel SSE, have also been explored to accelerate SW applications [17].

Since accelerators are normally connected to a multicore host, the idea to use both the accelerators and the SIMD extensions of multicores to execute SW came naturally and there are some approaches in the literature that explore this idea [16] [10] [13]. In order to distribute work among the hybrid processing elements, these approaches either (a) assume that multicores and accelerators have the same processing power [10]; (b) distribute work proportionally, considering the theoretical computing power of each processing element [13]; or (c) assign one work unit at the time [16]. As far as we know, there is no work in the literature that executes Smith-Waterman on hybrid platforms, distributing work according to the observed performance of each processing element.

This paper proposes and evaluates a strategy that uses workload adjustment to execute Smith-Waterman on hybrid platforms composed of multi-cores and accelerators. Our proposal assumes that the computational platform can be either dedicated or non-dedicated, being composed of one or more GPUs and one or more multi-cores with SIMD extensions. In the GPUs, CUDASW++ 2.0 [6] is used, since it is the state-of-the art program to compare query sequences to a biological database with SW in GPUs. In order to execute SW in multicores with SSE extensions, we implemented a modified version of the Farrar algorithm [18].

Having a set of query sequences and a biological database, our approach uses a master-slave strategy to assign work units to the processing elements in the following way. In the first allocation, the master assigns one work unit for each slave. The slaves execute the query x database comparison and ask for more work. Based on the observed performance, the master adjusts the number of work units to be assigned to each slave. Nevertheless, if a slow node receives one of the last tasks, the end of the computation can be significantly retarded. To avoid this situation, we propose a workload adjustment mechanism that allows idle nodes to execute tasks which have been assigned to other nodes which have not yet finished

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.28

the execution.

Our strategy was implemented in C with SSE extensions and CUDA, and it integrates CUDASW++ 2.0 into our code. 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: SwissProt/UniprotKB (available at *www.uniprot.org*, with 537,505 sequences), Enbembl (*www.ensembl.org*) Dog (25,160 sequences) and Rat (32,971 sequences) and RefSeq (*www.ncbi.nlm.nih.gov/RefSeq*) Human (34,705 sequences) and Mouse (29,437 sequences). The tests conducted in a hybrid platform composed of 4 NVidia GPUs and 2 Intel i7 (4 SSE real cores each) show that 172.82 GCUPS (Billions of Cells Updates per Second) can be attained, reducing the execution time from 7,190 seconds (one SSE core) to 112 seconds (4 GPUs + 4 Intel SSE Cores). Also, we show that our workload adjustment mechanism is able to reduce the total execution time in 57.2%.

The remainder of this paper is organized as follows. Section 2 presents the sequence comparison problem and the SW algorithm. Related work is discussed in Section 3. Our strategy for executing SW on hybrid platforms is proposed in Section 4. Section 5 presents experimental results. Finally, Section 6 concludes the paper and suggests future work.

II. Biological Sequence Comparison

A biological sequence is a molecule 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 ($\sum = \{A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y\}$).

Since two biological sequences are rarely identical, sequence comparison is in fact a problem of approximate pattern matching [1]. To compare two sequences, we need to find one alignment between them, which is to place one sequence above the other making clear the correspondence between similar characters [1]. In an alignment, spaces can be inserted in arbitrary locations so 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, we associate (a) a punctuation *ma*, if both characters are identical (*match*); or (b) a penalty *mi*, if the characters are different (*mismatch*); or (c) a penalty g, if one of the characters is a space (*gap*). The score is 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 figure, $ma = +1$, $mi = -1$ 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 alignment and score

A. Smith-Waterman (SW) Algorithm

The algorithm SW [2] is an exact method based on dynamic programming to obtain the optimal pairwise local alignment in quadratic time and space. It is divided in two phases: create the similarity matrix and obtain the alignment.

1) Phase 1: Create the similarity matrix: The first phase of the SW algorithm receives as input sequences s and t, with $|s| = m$ and $|t| = n$, where $|s|$ represents the size of sequence s. The similarity matrix is denoted $H_{m+1,n+1}$, 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 zeroes. The remaining elements of H are obtained from equation 1. In addition, each cell $H_{i,j}$ contains information about the cell that was used to produce the value.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + (\text{if } s[i] = s[j] \text{ then } ma \text{ else } mi) \\ H_{i,j-1} + \text{g} \\ H_{i-1,j} + \text{g} \\ 0 \end{cases} \quad (1)$$

2) Phase 2: Obtain the optimal alignment: In order to obtain the optimal local alignment, the algorithm starts from the cell that has the highest value in $H_{i,j}$, following the arrows until the value zero is reached. A left arrow in $H_{i,j}$ is the alignment of $s[i]$ with a gap in t. An up arrow represents the alignment of $t[j]$ with a gap in s. Finally, an arrow on the diagonal indicates that $s[i]$ is aligned with $t[j]$. Figure 2 presents the similarity matrix to obtain the local alignment between two sequences, with $score = 3$.

3) Affine-gap Model: The SW algorithm assigns a linear cost to gaps. Nevertheless, in nature, gaps tend to be together. For, this reason a higher penalty is usually associated to the first gap and a lower penalty is given to the following ones (affine-gap model). Gotoh [20] proposed an algorithm based on SW that implements the affine-gap model by calculating 3 Dynamic Programming (DP) matrices: H, E and F, where E and F keep track of gaps in each of the sequences.

	*	G	A	A	G	C	T	A
*	0	0	0	**0**	0	0	0	0
G	0	1	0	0	**1**	0	0	0
C	0	0	0	0	0	**2**	0	0
T	0	0	0	0	0	0	**3**	1
G	0	1	0	0	1	0	1	2
A	0	0	2	1	0	0	0	2
C	0	0	0	0	0	1	0	0
C	0	0	0	0	0	1	0	0
T	0	0	0	0	0	0	2	0

Figure 2. Similarity matrix for sequences s and t

B. SW in Parallel

There are several ways to parallelize the SW algorithm. In the following paragraphs, we assume that a set of x query sequences $(q_1, q_2, ..., q_x)$ will be compared to a set of y database sequences $(d_1, d_2, ..., d_y)$ and that $x \ll y$.

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 (Equation 1). Figure 3.a illustrates a fine-grained column-based block partition technique with four PEs. At the beginning, only $P0$ is computing. When $P0$ finishes calculating the values of a block of matrix cells, it sends its border column to $P1$, that can start calculating and so on. Note that, very close to the end of the matrix computation, only $P3$ is calculating. When the PEs finish to compare d_1 to q_1, they start computing SW for q_1 x d_2 and so on, until the comparison q_x x d_y is done.

In the coarse-grained parallelization, each PE receives the query sequence q_1 and a subset of d. The PEs calculate the SW algorithm for q_1 x subset of d, without communication, as shown in Figure 3.b. After that, the PEs compute the SW algorithm for q_2 x d, q_3 x d, and so on, until the computation of q_x and d is finished.

In the very coarse-grained approach, each PE compares a different query sequence to the whole database (Figure 3.c). For instance, $P0$ compares q_1 to d, $P1$ 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.

III. Related Work

Blazwicz et al. [12] propose a strategy to run SW (first and second phases) in multiple GPUs. In order to retrieve the alignment, the quadratic space procedure (Section II.A.2) was used. Therefore, only short sequences were compared. In this proposal, coarse-grained parallelization is used in the top level. The algorithm executes as follows. Each idle GPU retrieves one window (set of tasks which can be executed in one kernel invocation) from the top of the queue until the queue is empty, in a Self-Scheduling (SS) basis.

Ino, Kotani and Hagihara [14] propose the use of a non-dedicated grid composed of multiple GPUs to execute SW. When the screensaver is activated, coarse-grained SW tasks are executed. The authors propose a master/slave architecture where the resource manager server is the master and the grid resources are the slaves. Tasks are distributed using a self-scheduling policy.

The work of [14] was extended by [15] in order to take advantage of shorter idle periods of time so, in this case, the screensaver is not used. As in [14], a master/slave architecture is used and idle resources execute coarse-grained SW tasks. SW tasks are distributed to the workers in a modified self-scheduling fashion, where resources with longer idle periods receive tasks first. SW tasks can be canceled if the resource changes its state to busy. In this case, the task is re-assigned to another idle resource.

Rognes [17] proposes a multi-threaded version of an optimized SSE code to compare a query sequence to a database, with the SW algorithm. Work is assigned in an SS basis, where a task is defined as one query sequence and a chunk of x database sequences (coarse-grained allocation).

Meng and Chaudhary [13] propose the execution of SW in a platform composed of CPU and FPGA. A master/slave architecture is proposed where the master prepares the data, assigns tasks to the workers, merges the results and presents the final results to the user. The workers execute the SW algorithm. A configuration file is used that specifies the number of PEs (CPU w/o SSE, CPU with SSE, FPGA) that the node contains. This file is used to distribute work proportionally (WFixed) among the nodes. Since the FPGA imposes restrictions on the size of the sequences, long database sequences are compared in the CPU and long query sequences are segmented (with overlap). Depending on the degree of overlapping, the sensitivity of the SW algorithm is reduced.

Singh et al. [16] propose the use of desktop grids composed of CPUs and GPUs to execute SW comparisons for comparative genomics. In this case, all possible pairwise comparisons between n sequences are calculated ($(n * (n-1)/2)$ comparisons). The application is divided into work units of approximately the same number of residues. The processing of the working units is managed with BOINC (*boinc.berkeley.edu*), that allocates tasks in an SS manner.

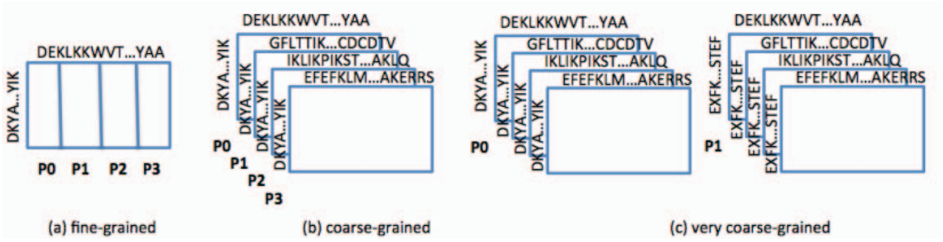


Figure 3. Strategies to Parallelize the SW Algorithm.

Table I
SMITH-WATERMAN APPROACHES ON HETEROGENEOUS PLATFORMS

Ref	Platform	Alloc	Ded.	Reassign	GCUPs
[12]	GPUs	SS	Yes	No	**11.00** 4 GPUs
[14]	GPUs	SS	No	No	**3.09** 8 GPUs
[15]	GPUs	SS	No	Yes	**64.00** 8 GPUs
[17]	CPUs	SS	Yes	No	**106.00** 12 SSE
[13]	CPU FPGA	WFixed	Yes	No	**11.30** 1 FPGA, 20 SSE
[16]	CPU GPU	SS (BOINC)	No	No	**4.40** 10 GPUs
[10]	CPU GPU	Fixed	Yes	No	**27.00** 1 GPU, 4 SSE

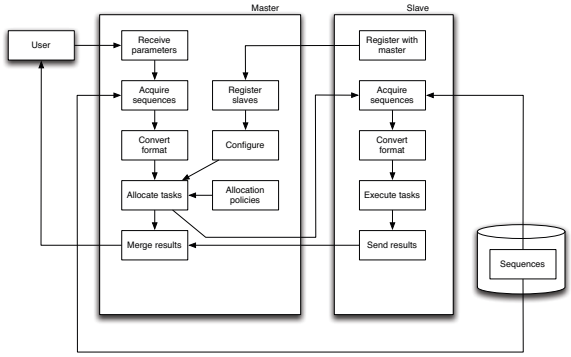


Figure 4. Overview of the Architecture

Singh and Aruni [10] proposed a strategy for executing the first phase of SW in a system composed of CPU and GPU. In the CPU, a modified version of SWPS3 [11] was executed and a modified version of CUDASW++ 2.0 [6] was executed in the GPU. The authors assumed that the performance of the CPU (4 SSE cores) and the GPU are the same, so the work was distributed evenly between both platforms in a coarse-grained way.

Table I presents a comparative view of the approaches discussed in this section. In column 2, we can see the platform targeted by each paper. As can be seen, most of the papers treat multiple heterogeneous GPUs or CPUs ([12], [14], [15], [17]). Two of them ([16] [10]) consider GPUs and CPUs and one of them ([13]) allocates tasks to CPUs and FPGAs. In order to allocate tasks to the PEs (Aloc. column), most of the papers use a self-scheduling-based policy, where an idle node asks for more tasks. All the papers that propose the execution on a non-dedicated environment use the SS policy (Ded. column). Only paper [15] allows task reassignment (Reassign column). In the last column, we can see the GCUPs provided by each paper. This information cannot be used for direct comparison since the GPUs, FPGAs and multi-cores used in the papers vary a lot. However, it can be used to give an idea of the potential of these approaches. The best GCUPs was obtained with 12 SSE cores.

IV. DESIGN OF THE SW TASK EXECUTION ENVIRONMENT FOR HYBRID PLATFORMS

In order to execute SW on hybrid platforms composed of different types of PEs, we propose a master/slave architecture provides user-selectable task allocation policies with dynamic workload adjustment capabilities. An overview of the proposed architecture is given in Figure 4.

First, the master process starts the execution and waits for the slaves to register. After that, the slaves can register themselves in the master. In order to execute SW comparisons, the user provides the names of two files (query and database). The master then reads the sequences (acquire sequences) and converts the format to a more suitable one. Using information about the number of slaves, their processing speed and the allocation policy, the master assigns a subset of tasks to the slaves. In our system, a task defined to be the comparison of one query sequence to one genomic database (very coarse-grained approach). When the slaves finish the execution of the assigned tasks, they communicate with the master, asking for more tasks. At this moment, the slaves implicitly give information to the master about their processing speed. When there are no more tasks left, the slaves send their results to the master, that merges them and sends the result to the user.

A. Task Allocation Module

Considering the hybrid and dynamic characteristics of our environment, we opted to provide two important functionalities in our task allocation module: multiple allocation policies and workload adjustment mechanism.

We do not believe that there is a single task allocation policy that is best suited for all databases and query sequences sizes as well as for all the hybrid platforms that we are targeting. For this reason, we claim that the user must be able to select the allocation policy which is more appropriate for his/her platform and sequence files. Therefore, when requesting one application execution, the user provides also the task allocation policy. So far, we have implemented the Self-Scheduling (SS) policy and the Package Weighted Adaptive Self-Scheduling (PSS) policies.

1) Self-Scheduling (SS): Having a set of N tasks and P slave PEs, the SS policy always allocates one task to each slave $p_i(Allocate(N,p_i)=1)$. When the slave finishes the execution of this task, it communicates with the master, asking for another task. Using this policy, the maximum idle time a set of slaves could wait for is limited by the processing time of one task in the slowest slave. Note that the SS policy incurs in considerable communication, since each task retrieved by a slave node requires at least one interaction with the master node.

2) Weighted Adaptive Task Allocation Strategy (PSS): Having a set of N tasks and P slave PEs, *PSS* allocates tasks to the slave p_i as shown in Equation (2). In our case, $Allocate(N,p_i)$ is the SS policy and $\Phi(p_i,P)$ represents the weight calculated by *PSS*.

$$PSS(p_i,N,P) = Allocate(N,p_i) * \Phi(p_i,P) \quad (2)$$

To distribute tasks to PEs, the master analyzes periodic notifications sent by the slave PEs, reporting the progress in processing tasks. It then calculates the weighted mean from the last Ω notifications sent by each p_i slave PE. A small Ω indicates that only very recent histories will be considered to calculate the weight. On the other hand, high values for Ω indicate that not only recent histories will be considered but also older ones. The weighted means calculated for each p_i slave PE are used to produce the $\Phi(p_i,P)$ PSS weight. A more detailed explanation of PSS can be found in [19].

3) Workload Adjustment Mechanism: On hybrid platforms, such as the ones composed of GPUs and SSE cores, load imbalance can have a great impact on the total execution time. For instance, if the last tasks are assigned to the slower nodes, the execution time of the whole application can be significantly augmented.

To tackle this problem, we propose a workload adjustment mechanism that works as follows. Each task can be in one of three states: *ready, executing* or *finished*. The slave PEs request tasks to the master node, using a given task allocation policy. As long as there are *ready* tasks, they are allocated to the slave PEs. When a slave PE requests tasks and there are no more *ready* tasks, the workload adjustment mechanism assigns tasks in the *executing* state to the idle PE. Note that, in this case, there can be more than one node executing the same task. When a slave PE finishes executing a task, the task state changes to *finished* and its results can be collected.

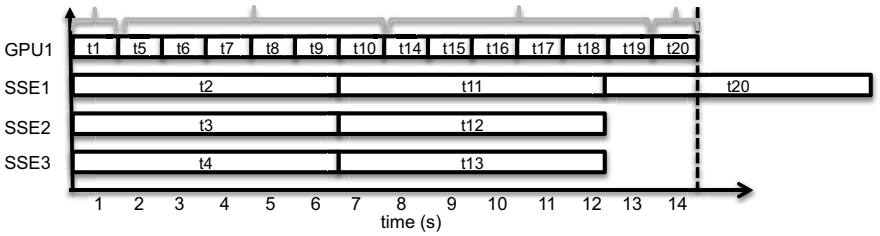
In order to illustrate the benefit of our workload adjustment mechanism, assume a system with 4 PEs (1 GPU and 3 SSE cores) and 20 tasks that take 1s to execute in the GPU, allocated by the PSS policy. In this system, the GPU is 6 times faster than the SSE core and the communication time between the master and the slave is negligible. Figure 5 illustrates this execution.

In Figure 5.a, the workload adjustment mechanism is used. At the beginning, one task is assigned to each PE. When GPU1 finishes processing task $t1$ (1s), it contacts the master, asking for more tasks. The master then calculates the processing power of GPU1 and assigns 6 tasks to it ($t5$ to $t10$). SSEs 1, 2 and 3 finish to process their task at 6s and contact the master, which assigns one task to each SSE ($t11$, $t12$ and $t13$, respectively). At time 7s, the GPU finishes to process its tasks and asks for more tasks. At this moment, it receives tasks $t14$ to $t19$. At time 12s, the SSEs finish executing and ask for more tasks. Since there is only one task left ($t20$), this task is allocated to SSE1. At time 13s, the GPU finishes processing its tasks and asks for more work. At this moment, there are no more idle tasks and tasks $t1$ to $t19$ are finished. Nevertheless, task $t20$ is at the *executing* state. Therefore, our workload adjustment mechanism assigns $t20$ to GPU1. At time 14s, GPU1 finishes executing $t20$ and the application execution is completed.

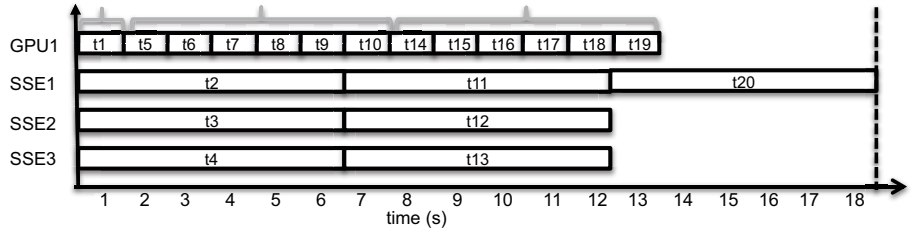
Note that, if we do not use the load adjustment mechanism (Figure 5.b), the execution is exactly the same until time $t13$. When GPU1 finishes the execution of tasks $t14$ to $t19$ and there are no more tasks in the ready state, the master replies saying that there are no more tasks to execute and waits for $SSE1$ to finish execution. In this case, the application execution is completed in 18s.

B. Indexed format for sequence files

Even though many biological files are called databases, they are in fact huge flat files where the sequences are put together. In our mechanism, the tasks compare one sequence to the whole database. In this case, the execution modules in the PEs will process the database files sequentially so this will



(a) Task allocation with the load adjustment mechanism. Total execution time is 14s.



(b) Task allocation without the load adjustment mechanism. Total execution time is 18s.

Figure 5. Comparison between the mechanism with and without the load adjustment mechanism

should not be a problem. Nevertheless, we also have a flat file that contains the query sequences which will be compared to the database.

In order to retrieve quickly a subset of query sequences, we propose an indexed format that keeps track of the total number of sequences, the size of the biggest sequence and the offset that marks the beginning of each sequence in the file. Using the offsets, we can quickly retrieve the beginning of a sequence that is in the middle of the file.

C. SW execution in the PEs

In order to execute SW on GPUs, we opted to use CUDASW++ 2.0 [6]. Differently from [10], we did not need to modify CUDASW++ 2.0, since we defined that each task will compare one query sequence with the whole genomic database. Therefore, CUDASW++ 2.0 was encapsulated and easily integrated to our tool.

In order to execute SW on SSE cores, we implemented the Farrar algorithm [18] , generating an adapted Farrar version. Basically, our version uses signed integers instead of unsigned ones to store the values of the SW DP matrices, augmenting the maximum score to 255 (8 bits) and 65536 (16 bits).

V. Experimental Results

We implemented the architecture proposed in section IV in C with SSE extensions and CUDA. The 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 executed in a hybrid platform composed of 2 hosts interconnected by Gigabit Ethernet. Each host contained 2 GPUs NVidia GTX580 and one Intel Core i7 3.4GHz, 8GB RAM. The PSS policy was used in all the tests and, unless otherwise stated, the workload adjustment mechanism was always activated.

In our tests, we compared 40 query sequences to five genomic databases (Table II). We chose 40 query sequences from each genomic database, with equally distributed sizes, ranging from 100 amino acids to approximately 5,000 amino acids, as in [6].

Table II
Genomic Databases in our 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
UniProtDB/SwissProt	537,505	100	4,998

A. Execution Times and GCUPs

1) SSE Execution: First, we measured the time needed for the multicores to compare these 40 amino acid sequences to each database. Our modified Farrar implementation (Section IV-C) was used in up to 4 cores in each Intel Core i7. The wallclock execution times and GCUPs (Billions of Cells Updated per Second) are shown in Table III. In this table, we can see that speedups close to linear are obtained for all databases.

2) GPU Execution: We also measured the execution times/GCUPs for 1, 2 and 4 GPUs (Table IV). Note that the results obtained for 4 GPUs were collected using both hosts. As in the SSE execution, the GPU execution presents speedups close to linear for

Table III
RESULTS FOR THE SSE CORES

Database	1 SSE Time(s) GCUPs	2 SSEs Time(s) GCUPs	4 SSEs Time(s) GCUPs	8 SSEs Time(s) GCUPs
Ensembl Dog	553.03 2.68	283.27 5.23	157.13 9.43	76.74 19.31
Ensembl Rat	629.08 2.68	337.27 5.17	181.31 9.61	88.89 19.60
RefSeq Human	673.55 2.92	353.79 5.56	197.81 9.95	96.02 20.49
RefSeq Mouse	572.47 2.80	305.92 5.24	169.54 9.46	82.01 19.55
UniProtDB/SwissProt	7,190.60 2.80	3,615.38 5.38	2,020.68 9.63	1,027.28 18.94

all databases. Differently form the SSE execution, the GPUs obtain much better GCUPs and, consequently, better execution times, for huge databases such as the UniProtDB/SwissProt. For this comparison, we were able to obtain 163.93 GCUPs, using 4 GPUs, which is approximately the double of GCUPs obtaining when using the other databases.

Table IV
RESULTS FOR THE GPUS

Database	1 GPU Time(s) GCUPs	2 GPUs Time(s) GCUPs	4 GPUs Time(s) GCUPs
Ensembl Dog	72.54 20.43	39.44 37.58	23.75 62.41
Ensembl Rat	93.34 18.62	52.85 32.96	25.90 67.25
RefSeq Human	89.16 22.07	46.52 42.30	24.75 79.52
RefSeq Mouse	69.99 22.91	36.82 43.54	22.95 69.81
UniProtDB/SwissProt	439.15 44.3	222.85 87.3	118.67 163.93

3) GPU and SSE Execution: In order to measure the benefits of using a hybrid platform, we measured the wallclock execution time and GCUPs obtained when comparing 40 query sequences to the five genomic databases (Table V).

Table V
RESULTS FOR THE GPUS AND SSES

Database	1 GPU +1 SSE Time(s) GCUPs	1 GPU +2 SSE Time(s) GCUPs	1 GPU +4 SSE Time(s) GCUPs	2 GPU +4 SSE Time(s) GCUPs	4 GPU +4 SSE Time(s) GCUPs
Ensembl Dog	65.02 22.79	60.87 24.35	53.82 27.53	33.47 44.28	24.93 59.43
Ensembl Rat	84.64 20.58	79.79 21.83	68.33 25.49	47.99 36.3	28.83 60.41
RefSeq Human	81.096 24.27	78.67 25.01	65.47 30.06	46.87 41.98	25.16 71.77
RefSeq Mouse	65.44 24.50	61.01 26.28	50.59 31.69	34.38 46.63	25.16 63.67
UniProtDB/ SwissProt	425.19 45.76	400.64 48.56	393.17 49.48	211.85 91.83	112.43 172.82

We can see that the combined GPU + SSE execution (Table V) provides better times and GCUPs for 1 and 2 GPUs (Table IV). However, better results are obtained with the 4 GPUs execution for the first four databases, when compared to the 4 GPUs + 4 SSEs execution. This happens because these databases are relatively small (Table II) and most of the work assigned for the SSEs is actually done by the GPUs, using the workload adjustment mechanism. For the UniProtDB/SwissProt, we observe that better results are obtained by the 4 GPUs + 4 SSE platform, indicating that this configuration is best suited for bigger databases.

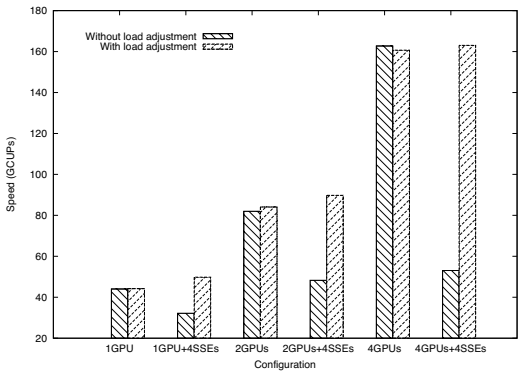


Figure 6. GCUPs for comparing the databases with and without the workload adjustment mechanism

B. Impact of the Workload Adjustment Mechanism

In order to evatuate the benefits of our workload adjustment mechanism, we compared the 40 query sequences with and without the mechanism, for the UniProtKD/SwissProt database (Figure 6).

In this figure, we first show that the load adjustment mechanism has a negligible impact when the PEs are homogeneous (1, 2 and 4 GPUs). Without using the workload adjustment mechanism, the GCUPs rate drops a lot when we add SSEs to the execution configuration (1GPU+4SSEs, 2GPUs+4SSEs and 4GPUs+4SSEs). Nevertheless, when the load adjustment mechanism is activated, much better performance is obtained. For instance, for the 2GPUs+4SSEs platform, we obtained 48.288 GCUPs without the mechanism and 89.768 GCUPs with the mechanism, yielding a performance gain of 85,9%. The same behavior was also obtained for the 4GPUs+4SSE case. The workload adjustment mechanism was able to achieve a performance gain of 207,2%, when compared to the execution without the mechanism. Also, we can see in this figure that using GPUs combined with SSEs gives a better performance than the GPU-only solution, showing the appropriateness of the hybrid platform.

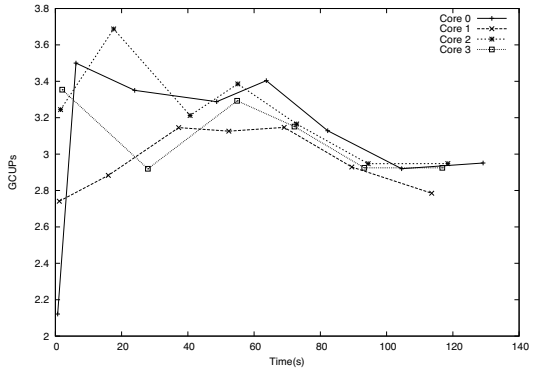


Figure 7. Dedicated execution with 4 cores

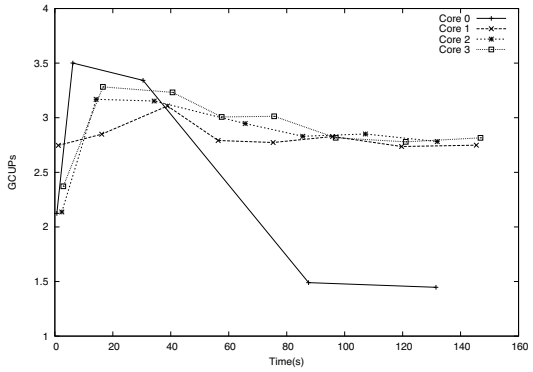


Figure 8. Non-dedicated execution with 4 cores, with local load at core 0

C. Non-dedicated Execution

In this test, we wanted to evaluate if our PSS policy is able to adjust the PE weights if local load is introduced. In order to do that, we compared the Ensembl Dog database with 40 query sequences, using 4 SEEs.

First, we executed the Smith-Waterman application in a dedicated environment (Figure 7). In this figure, each mark corresponds to a task allocation interaction with the master node. With this execution, we noticed that, even without other application running in the machine, there is a small variation in the GCUPs of each core, probably due to some operating system's services. The wallclock execution time was 129.13s.

Second, we re-executed the SW application and introduced local load at core 0, after 60 seconds of execution time (Figure 8). The local load was caused by the execution of the compute-intensive benchmark *superpi* (*www.superpi.net*). As can be seen in Figure 8, after about 60 seconds, the GCUPs rate for core 0 is reduced to less than a half, since now our SW application competes with *superpi* for the CPU. The wallclock execution time was 146.86s, i.e., an augmentation of 12.1% in the execution time. This is a very good result since the computing power of the 4-core system was reduced in approximately 15%, from 60s to 146.86s.

VI. Conclusion

In this paper, we proposed and evaluated a master/slave architecture to execute SW applications on hybrid platforms composed of SSE cores and GPUs. Our architecture is flexible in such a way that multiple task allocation policies can be incorporated to it. Also, we propose a workload adjustment technique that is able to cope with load balancing problems that arise when slow nodes receive some of the last tasks.

The results obtained when comparing 40 query sequences with 5 different genomic databases show that our architecture and mechanism can be used with SSE cores, GPUs and GPUs+SSEs, with very good performance benefits. We also show that the workload adjustment technique can reduce drastically the execution time when GPUs and SSE cores are used. Without this mechanism, many of the hybrid executions would not be better than the GPU-only executions. Finally, we also evaluated our PSS allocation policy and showed that it is able to adapt the number of tasks assigned to a PE when local load occurs.

As future work, we intend to integrate FPGAs to our architecture. Also, we want to extend our solution to tackle situations where nodes join/leave the platform while an SW application is executing. Finally, we want to adapt our architecture to run other Bioinformatics applications, such as Multiple Sequence Alignment and DNA Assembly/Scaffolding, among others.

Acknowledgment

This work is partially supported by CNPq/Brazil (Alba C. M. A. Melo's fellowship) and Capes/Brazil (Fernando M. Mendonca's graduate scholarship and the GPU machines).

References

[1] D.W. Mount. *Bioinformatics: sequence and genome analysis*. Cold Spring Harbor Laboratory Press, 2004.

[2] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal Mol. Biol.*, 147:195-197, 1981.

[3] A. Boukerche, A. C. Melo, E. F. Sandes, and M. Ayala-Rincon. An exact parallel algorithm to compare very long biological sequences in clusters of workstations. *Cluster Computing*, 10(2):187–202, 2007.

[4] E. F. de O. Sandes and A. C. M. A. de Melo. Smith-Waterman Alignment of Huge Sequences with GPU in Linear Space. In *IPDPS*, pages 1199–1211, 2011.

[5] D. Hains, Z. Cashero, M. Ottenberg, W. Bohm, and S. Rajopadhye. Improving CUDASW++, a Parallelization of Smith-Waterman for CUDA Enabled Devices. *HICOMB 2011*, 490–501, 2011.

[6] Y. Liu, B. Schmidt, and D. Maskell. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, 3(1):93, 2010.

[7] C. Chen and B. Schmidt. An adaptive grid implementation of DNA sequence alignment. *Future Generation Comp. Syst.*, 21(7):988–1003, 2005.

[8] X. Liu L. Xu P. Zhang N. Sun X. Jiang. "A Reconfigurable Accelerator for Smith-Waterman Algorithm". *"IEEE Trans. on Circuits and Syst. II"*, 54(12):1077–1081, December 2007.

[9] S. Rajko and S. Aluru. Space and time optimal parallel sequence alignments. *IEEE Trans. Parallel Dist. Syst.*, 15(12):1070–1081, 2004.

[10] J. Singh, I. Aruni. Accelerating Smith-Waterman on Heterogeneous CPU-GPU Systems. *5th ICBBE*, 2011.

[11] A. Szalkowski, C. Ledergerber, P. Krahenbuhl and C. Dessimoz. SWPS3 fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. *BMC Research Notes*, 1:107, 2008.

[12] J. Blazewicz, W. Frohmberg, M. Kierzynka, E. Pesch and P. Wojciechowski. Protein alignment algorithms with an efficient backtracking routine on multiple GPUs. *BMC Bioinformatics*, 12:181, 2011.

[13] X. Meng and V. Chaudhary. A High-Performance Heterogeneous Computing Platform for Biological Sequence Analysis. *IEEE Trans. Parallel Dist. Syst.*, 21(9):1267-1280, 2010.

[14] F. Ino, Y. Kotani, Y. Munekawa, K. Hagihara. Harnessing the Power of Idle GPUs for Acceleration of Biological Sequence Alignment. *Parallel Processing Letters*, 19(4):513, 2009.

[15] F. Ino, Y. Munekawa, K. Hagihara. Sequence Homology Search Using Fine Grained Cycle Sharing of Idle GPUs. *IEEE Trans. Parallel Dist. Syst.*, 23(4):751-759, 2012.

[16] A. Singh, C. Chen, W. Liu, W. Mitchel, B. Schmidt. A Hybrid Computational Grid Architecture for Comparative Genomics. *IEEE Trans. Inf. Tech. in Biomed.*, 12(2):218-225, 2008.

[17] T. Rognes. Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics*, 12:221, 2011.

[18] Farrar M. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156-161, 2007.

[19] M. S. Sousa, A. C. M. A. Melo, A. Boukerche. An adaptive multi-policy grid service for biological sequence comparison *J. Par. Dist. Comp.*, 70(2): 160-172, 2010.

[20] O. Gotoh. An improved algorithm for matching biological sequences. *J. of Mol. Biology*,162:705-708, 1982.