MASA: A Multiplatform Architecture for Sequence Aligners with Block Pruning

EDANS F. DE O. SANDES, Department of Computer Science, University of Brasilia, Brazil
GUILLERMO MIRANDA, Barcelona Supercomputing Center, Spain
XAVIER MARTORELL and EDUARD AYGUADE, Barcelona Supercomputing Center and Universitat Politecnica de Catalunya, Spain
GEORGE TEODORO and ALBA C. M. A. DE MELO, Department of Computer Science, University of Brasilia, Brazil

Biological sequence alignment is a very popular application in Bioinformatics, used routinely worldwide. Many implementations of biological sequence alignment algorithms have been proposed for multicores, GPUs, FPGAs and CellBEs. These implementations are platform-specific; porting them to other systems requires considerable programming effort. This article proposes and evaluates MASA, a flexible and customizable software architecture that enables the execution of biological sequence alignment applications with three variants (local, global, and semiglobal) in multiple hardware/software platforms with block pruning, which is able to reduce significantly the amount of data processed. To attain our flexibility goals, we also propose a generic version of block pruning and developed multiple parallelization strategies as building blocks, including a new asynchronous dataflow-based parallelization, which may be combined to implement efficient aligners in different platforms. We provide four MASA aligner implementations for multicores (OmpSs and OpenMP), GPU (CUDA), and Intel Phi (OpenMP), showing that MASA is very flexible. The evaluation of our generic block pruning strategy shows that it significantly outperforms the previously proposed block pruning, being able to prune up to 66.5% of the cells when using the new dataflow-based parallelization strategy.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Parallel Programming

General Terms: Algorithms

Additional Key Words and Phrases: Biological sequence alignment, parallel algorithms, GPU, multicores, Intel Phi

ACM Reference Format:
Edans F. De O. Sandes, Guillermo Miranda, Xavier Martorell, Eduard Ayguade, George Teodoro, and Alba C. M. A. De Melo. 2016. MASA: A multiplatform architecture for sequence aligners with block pruning. ACM Trans. Parallel Comput. 2, 4, Article 28 (February 2016), 31 pages.
DOI: http://dx.doi.org/10.1145/2858656

This work was supported by grant no. SEV-2011-00067 from the Severo Ochoa Program, awarded by the Spanish Government, grant no. TIN2012-34557 of the Spanish Ministry of Science and Technology, and grant no. 2014-SGR1051 of the Generalitat de Catalunya. This work is also partially supported by CNPq/Brazil (grant nos. 242800/2012-2, 211456/2013-6, 313931/2013-5, 305208/2014-4, and 446297/2014-3).
Authors' addresses: E. F. de O. Sandes, G. Teodoro, and A. C. M. A. de Melo, Predio CIC/EST, Campus UNB Asa Norte, PO Box 4466, University of Brasilia (UNB), Brasilia-DF, Brazil, CEP: 70910-900; emails: {edans, george, albamm}@cic.unb.br; G. Miranda, X. Martorell, and E. Ayguade, Carrer de Jordi Girona, 29, 08034 Barcelona, Spain; emails: {guillermo.miranda, xavier.martorell, eduard.ayguade}@bsc.es.

DOI: http://dx.doi.org/10.1145/2858656

1. INTRODUCTION

The astonishing evolution of DNA sequencing techniques is producing an overwhelming number of new biological sequence data to be analyzed. Also, the number of laboratories that are analyzing the sequences is quickly increasing. To cope with this, Life Sciences laboratories are facing the challenge of producing accurate results in a very short time.

Once a new biological sequence is produced, its functional/structural characteristics must be established. In order to do that, the newly discovered sequences are compared against other sequences, looking for similarities. This is done on a daily basis all over the world.

Biological sequence comparison is, therefore, a very important operation in Bioinformatics [Mount 2004]. It produces a score, indicating the similarity between the sequences and, optionally, an alignment, which highlights the regions of similarities/differences between the sequences.

Biological sequences can be DNA, RNA, or protein sequences. Protein and RNA sequences are rather small; their sizes range from hundreds to tens of thousands of residues (amino acids and nucleotide bases, respectively). On the other hand, DNA sequences can be very long, often composed of millions of base pairs (Mbp).

There are three types of comparisons: (1) global, in which all the characters of the sequences belong to the alignment; (2) local, in which a subset of the characters belongs to the alignment; and (3) semiglobal, in which the head/tail of the sequences is discarded. Depending on the analysis, the biologists may choose among the types of the sequences (DNA, RNA, or protein), the comparison type (local, global, semiglobal) and the output produced (score, score, and alignment).

The Needleman-Wunsh (NW) algorithm [Needleman and Wunsch 1970] is an exact algorithm that retrieves the optimal score/alignment of a global pairwise sequence comparison. It is based on Dynamic Programming (DP) and calculates a DP matrix of size $n \times m$, where n and m are the sizes of the sequences. NW has quadratic time and space complexity $O(nm)$.

In 1981, the Smith-Waterman (SW) algorithm [Smith and Waterman 1981] was proposed. It calculates the optimal score/alignment of a local sequence comparison by slightly modifying the recurrence relation used in the computation of the NW DP matrix. As in NW, SW has time and space complexity $O(nm)$.

In order to reduce the execution time of the NW and SW exact methods, heuristic methods such as BLAST [Altschul et al. 1990] were proposed. These methods combine exact pattern matching with dynamic programming in order to produce good solutions faster. BLAST can align sequences in a very short time, still producing good results. Nevertheless, its accuracy is expected to be worse than the accuracy of the exact methods. In this article, we focus on the exact algorithms.

In the last years, many parallel variants of SW have been proposed for CPUs with Single Instruction Multiple Data (SIMD) vector instructions [Farrar 2007], clusters [Rajko and Aluru 2004; Chen and Schmidt 2003], Field Programmable Gate Arrays (FPGAs) [Yamagutchi et al. 2011; Maskell et al. 2005], Cell Broadband Engines (CellBEs) [Sanchez et al. 2010; Sarje and Aluru 2009; Wirawan et al. 2009], Graphics Processing Units (GPUs) [Sandes and Melo 2013a; Liu et al. 2013; Manavski and Valle 2008; Xiao et al. 2009; Korpar and Sikic 2013] and, more recently, for the Intel Phi [Liu and Schmidt 2014b]. Even though these SW proposals commonly share some similar features or code blocks that could be reused, they usually have a design that is highly platform-dependent. Therefore, porting them to other high-performance computing platforms tends to involve a significant programming effort.

CUDAlign [Sandes and Melo 2013a] is an optimized parallel variant of SW that is able to compare Megabase DNA sequences (>1Mbp) using GPU. In contrast to

other variants of SW, which calculate the whole DP matrix, CUDAlign proposed Block Pruning (BP) optimization, which is able to reduce the number of cells calculated while still guaranteeing that the optimal local alignment will be produced. BP is suitable for regular processing patterns (e.g., diagonal by diagonal) and it was able to reduce computation in more than 50%, when the sequences have high similarity. Since the code of CUDAlign is highly dependent on CUDA, it only runs on NVIDIA GPUs. Nevertheless, we identified that more than 90% of the CUDAlign code was platform-independent; thus, the full reimplementation to a new platform would require only a small amount of coding effort in order to rewrite the 10% of the code related to the platform-specific section.

The alignment of biological sequences longer than 10Mbp with exact DP algorithms such as SW and NW is still considered unfeasible by most of the researchers, and the use of these algorithms in chromosome-wide scale has not been fully explored. Still, in the last three years, there have been proposals to use these exact methods in Megabase sequences. Besides CUDAlign [Sandes and Melo 2013a], in the recent literature, we can find SW# [Korpar and Sikic 2013], which executes SW comparisons of 33Mbp $\times$ 46Mbp on GPUs; and SWAPHI-LS [Liu et al. 2014], which presents comparisons of 42Mbp $\times$ 50Mbp on the Intel Phi. This shows that other researchers are also putting effort into chromosome-wide SW/NW sequence comparisons. Hence, we claim that local/global/semiglobal optimal chromosome-wide alignments can be very useful since they can reveal information that is not obtained with other tools/techniques.

This article proposes and evaluates MASA (**M**ultiplatform **A**rchitecture for **S**equence **A**ligners), a software architecture for sequence aligners based on CUDAlign. The main contributions of MASA are: (1) a flexible and customizable architecture for deployment of pairwise biological sequence aligners in different hardware/software platforms, (2) a generic version of the BP optimization, (3) a novel asynchronous parallelization strategy based on the dataflow model, and (4) four parallel MASA aligner implementations for multicores and accelerators.

In order to design MASA, we faced the challenge of modifying CUDAlign in order to decouple the small platform-specific code from the platform-independent one, creating a simple integration API without losing performance. To accomplish this, a generic version of BP needed to be proposed to support the processing of the DP matrix in an asynchronous way, for which the regular diagonal processing pattern is not guaranteed.

In addition to proposing the MASA architecture, in this article, we design and implement four MASA aligners, which consists of rewriting the platform-specific code to four distinct hardware/software platforms. MASA-CUDAlign runs on NVIDIA GPUs. MASA-OpenMP/Phi is a sequence aligner that runs on the Intel Phi accelerator, programmed with OpenMP. MASA-OpenMP/CPU is the OpenMP version that runs on multicores. Finally, MASA-OmpSs/CPU is an OmpSs [Duran et al. 2011] version that also runs on multicores. With these 4 implementations, we show that multiple hardware platforms can be integrated to MASA and that high-level parallel programming environments such as OmpSs and OpenMP can also be integrated.

In addition, we included in MASA not only the local sequence alignment exact algorithm (SW), but also the global (NW) and semiglobal variants. Finally, a new BP strategy was created in the MASA portable code to allow the computation of the DP matrix in a generic execution order, not restricted to antidiagonals, as it is implemented in CUDAlign. The source code of MASA is freely available and can currently be found at https://github.com/edanssandes/MASA-Core.

Experimental results obtained with real DNA sequences show that our multicore implementations (MASA-OmpSs/CPU and MASA-OpenMP/CPU) outperform MASA-CUDAlign and MASA-OpenMP/Phi for small sequences. For longer sequences, MASA-CUDAlign presents the best execution times. We also show that generic BP combined

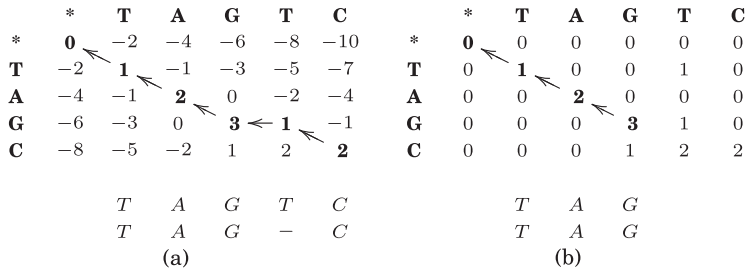


Fig. 1. DP matrices and alignments for S_0 and S_1 (mi=−1, ma=+1, g=−2) using NW and SW.

with the OmpSs dataflow model is able to prune more cells than the original BP and, as a consequence, MASA-OmpSs/CPU outperforms MASA-OpenMP/CPU in most cases. Finally, we present the alignment results for two strains of the *Amycolaptosis mediterranei* bacteria (10Mbp x 10Mbp) and the human x chimpanzee homologous chromosomes 21 (47Mbp x 32Mbp).

The rest of this article is organized as follows. Section 2 presents the biological sequence alignment problem. Section 3 presents related work in the area of biological sequence aligners for HPC platforms. Section 4 gives a brief overview of CUDAlign. The design of MASA is presented in Section 5. Section 6 discusses the Generic Block Pruning optimization proposed in this article. Section 7 presents and discusses experimental results. Section 8 presents our conclusions.

2. BIOLOGICAL SEQUENCE ALIGNMENT

DNA biological sequences are treated as strings composed of elements of the alphabet $\Sigma = \{A, T, G, C\}$. To compare two sequences, it is necessary to find an alignment between them, placing one sequence above the other and making clear the correspondence between the characters [Mount 2004]. In the alignment, spaces (*gaps*) can be introduced in one of the sequences in order to improve the alignment quality. Each alignment has a score, which measures the similarity between the sequences. Our goal is to obtain the alignment with the highest score.

2.1. NW and SW Algorithms

The NW algorithm [Needleman and Wunsch 1970] is based on DP and obtains the optimal global alignment in quadratic time and space. It takes as input sequences S_0 and S_1, with sizes m and n, respectively. The DP matrix H is calculated as follows. The first row and column of H are filled with $-g \cdot i$, where i is the size of the nonempty sequence and g is the gap penalty. The remaining cells are calculated with Equation (1), where *ma* and *mi* are the punctuations for match ($S_0(i) = S_1(j)$) and mismatch ($S_0(i) \neq S_1(j)$), respectively. Each cell keeps an indication of the cell that was used to produce the value (arrows in Figure 1). The optimal global score is the value contained in $H_{m,n}$. To retrieve the alignment, a traceback procedure is executed from the bottom right cell, following the arrows until the top left cell is attained (Figure 1 (a)).

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + (\text{if } S_0[i] = S_1[j] \text{ then } ma \text{ else } mi) \\ H_{i,j-1} - g \\ H_{i-1,j} - g \end{cases} \quad (1)$$

The SW algorithm [Smith and Waterman 1981] is used to obtain the optimal local alignment. It is similar to NW, with three differences. First, the initial row and column are filled with zeroes. Second, Equation (2) is used to compute the cells. Third, the traceback starts in the cell that has the optimal local score (highest value in H) and

stops when a zero-valued cell is reached (Figure 1 (b)).

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + (\text{if } S_0[i] = S_1[j] \text{ then } ma \text{ else } mi) \\ H_{i,j-1} - g \\ H_{i-1,j} - g \\ 0 \end{cases} \quad (2)$$

NW and SW assign a constant cost g to gaps. However, gaps tend to occur together. Thus, a higher penalty is associated to the gap opening and a lower penalty to the gap extensions. This is called the *affine-gap* model, which calculates 3 DP matrices: H, E, and F (Equations (3), (4), and (5)), where E and F keep track of gaps in each sequence [Gotoh 1982].

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + (\text{if } S_0[i] = S_1[j] \text{ then } ma \text{ else } mi) \\ E_{i,j} \\ F_{i,j} \\ 0 \end{cases} \quad (3)$$

$$E_{i,j} = \max \begin{cases} E_{i,j-1} - G_{ext} \\ H_{i,j-1} - G_{open} \end{cases} \quad (4)$$

$$F_{i,j} = \max \begin{cases} F_{i-1,j} - G_{ext} \\ H_{i-1,j} - G_{open} \end{cases} \quad (5)$$

2.2. Semiglobal Alignments

The algorithm for semiglobal alignment mixes the matrix initialization and NW/SW recurrence relations in order to ignore gap penalties in the head and/or tail of the sequences. Durbin et al. [2002] define the overlap alignment as a special semiglobal case in which the alignment starts in the first row or first column and ends in the last row of last column of the matrix. Liu and Schmidt [2014a] also define semiglobal alignment as an alignment that starts in the first row and ends in the last row, or, symmetrically, starts in the first column and ends in the last column.

In general terms, an alignment type may combine other possibilities, using different initialization formulae and optimal score retrieval positions. Thus, we can classify the alignment types considering where the alignment may start and where it may end. The following define symbols for each edge possibility and state the changes that must be done in the alignment algorithm.

Regarding the beginning of the alignment:

(+) *in the first cell:* initializes the first row and column with gap penalties and uses the NW recurrence relation;
(1) *in the first row:* initializes the first row with zeroes and the first column with gap penalties and uses the NW recurrence relation;
(2) *in the first column:* initializes the first row with gap penalties and the first column with zeroes and uses the NW recurrence relation;
(3) *in the first row or column:* initializes the first row and column with zeroes and uses the NW recurrence relation;
(*) *anywhere in the matrix:* initializes the first row and column with zeroes and uses the SW recurrence relation.

Regarding the end of the alignment:

(+) *in the last cell:* optimal score resides exactly at the last cell;
(1) *in the last row:* seeks for the optimal score in the last row;
(2) *in the last column:* seeks for the optimal score in the last column;

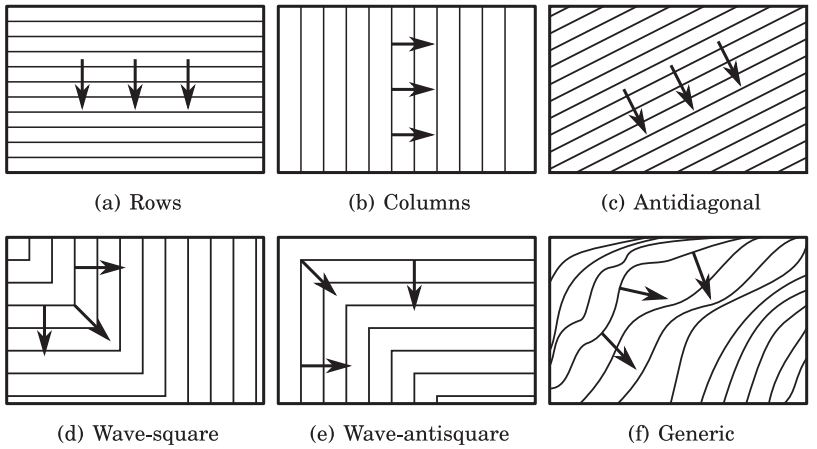


Fig. 2. Different orders to calculate the DP matrix.

(3) *in the last row or column:* seeks for the optimal score in the last row or column;
(*) *anywhere in the matrix:* seeks for the optimal score in any cell of the matrix.

In this way, we may have $5 \times 5 = 25$ possible alignment types. We will represent the alignment type with the symbol (b/e) when beginning at position b and ending at position e. For instance, type (2/$*$) starts in the first column and ends anywhere in the matrix. Some combinations are already defined as a global alignment (+/+) and local alignment (*/*). All other combinations may be classified as specific semiglobal alignment types, for instance, the overlap alignment (3/3).

2.3. Linear Space Algorithms

Many SW/NW applications retrieve only the score. In this case, the traceback procedure is not executed and space complexity is linear. To retrieve the alignment with SW/NW for huge sequences, quadratic space is required; for this reason, several Petabytes of memory may be needed. Therefore, another approach must be used.

The algorithm proposed by Hirschberg [1975] retrieves alignments in linear space. It was further adapted by Myers-Miller (MM) [Myers and Miller 1988] to the affine-gap model. MM is an NW variant that retrieves alignments in linear space with affine-gaps. It uses a divide-and-conquer technique with a matching procedure to find one point where the optimal alignment occurs and recursively splits the DP matrix to obtain the alignment. This approach can double the execution time, in the worst-case scenario [Myers and Miller 1988], when compared with NW.

In order to apply MM to local alignments, the beginning and end positions of the local alignment must be previously found with the SW algorithm. With these positions, the problem can be treated as a global alignment; thus, the MM algorithm can be applied. Liu et al. [2009] is one of the first proposals that apply the MM algorithm for local alignments in GPUs.

2.4. Parallel SW/NW

In SW/NW and its variants, most of the time is spent calculating the DP matrices; this is the part that is usually parallelized. Each cell (i, j) of the DP matrix depends on three other cells: $(i-1, j)$, $(i-1, j-1)$, and $(i, j-1)$ (Equations (1) to (5)). Respecting these data dependencies, the DP matrix can be processed in different ways (Figure 2):

by row (a), by column (b), by antidiagonal (c), by wave-square (d), by wave-antisquare (e) or in a generic order (f).

Considering the execution orders presented in Figure 2, the antidiagonal strategy, also called *wavefront* [Pfister 1995], is a very straightforward and commonly used method that permits parallel computation of the cells. Since the cells in the same antidiagonal do not have direct dependencies among them, all cells in each diagonal can be computed in parallel. The linear strategies presented in Figures 2(a) (rows) and 2(b) (columns) are also suited for parallel execution using prefix computations [Aluru et al. 2003; Rajko and Aluru 2004]. Square wave strategies (Figures 2(d) and 2(e)) alternate the computation of rows and columns in the same execution. Finally, the DP matrix can be processed in a generic order, as presented in Figure 2(f), allowing cells in different rows, columns or antidiagonals to be processed simultaneously. This generic execution order is well suited for dynamic schedulers that resolve task dependencies at runtime. This strategy may generate irregular computation patterns with reduced synchronization points, while still respecting the data dependencies.

3. RELATED WORK

In past decades, several strategies have been proposed the efficient execution of biological sequence comparison algorithms in HPC platforms. Currently, HPC platforms are usually composed of CPUs and accelerators such as Field Programmable Gate Arrays (FPGAs), Cell Broadband Engines (CellBEs), Graphics Processing Units (GPUs) and, more recently, Intel Phis. In the literature, we can find more than 50 papers in this research field; it is not our intention to discuss extensively all these works. Therefore, we will discuss solutions that: (1) calculate more than one type of alignment; (2) use multiple hardware/software platforms; and (3) compare Megabase sequences.

A parallel CellBE approach to retrieve optimal global, syntenic, and spliced alignments is proposed in Sarje and Aluru [2009]. The MM algorithm (Section 2.3), combined with Parallel Prefix (PP) computations and antidiagonal parallelizations (Figure 2(c)), is used to retrieve these 3 types of alignments. Sarkar et al. [2010] designed an Application-Specific Integrated Circuit (ASIC) composed of tiny Processing Elements (PEs) integrated through an NoC (Network-on-Chip) to retrieve global, local, and semiglobal alignments. A unidirectional systolic array was designed that can calculate alignments with either the original SW/NW equations (Section 2.1) or with PP. Each PE was implemented in Register-Transfer Level (RTL) and the switches were designed with Candence Spectra Tools, generating a custom hardware. Maleki et al. [2014] used linear algebra to transform DP problems, including local, global and sequence-profile comparisons, into independent sets of matrix-vector multiplications (MVM) and predecessor products, using the concept of rank convergence to obtain optimal alignments in CPU. The algorithm executes iteratively, stage by stage, where each stage is executed in parallel. To compute local alignments, Farrar's algorithm [Farrar 2007], which uses Streaming SIMD Extension (SSE) vector instructions that process according to the striped pattern with query profile, was used inside each stage. Liu and Schmidt [2014a] proposed GSWABE, a GPU strategy that computes global, semiglobal, and local alignments with the Gotoh algorithm (Section 2.1) that compares short DNA reads with whole genomes. In this strategy, the DP matrix is divided into tiles of size 4x4 and each thread executes a different comparison, using antidiagonal parallelism (Figure 2(c)).

In Aldinucci et al. [2010], the Farrar SSE implementation [Farrar 2007] was reimplemented according to the dataflow model (Figure 2(f)) and ported to multiple high-level programming environments for shared memory multicores (OpenMP, Cilk, TBB, and FastFlow). Benkrid et al. [2012] create different SW codes to locally compare protein sequences that are highly optimized for different hardware platforms (FPGA,

GPU, and CellBE) programmed in Handel-C, CUDA, and Cell SDK, respectively. The FPGA and GPU implementations are fine-grained and explore the antidiagonal parallelism (Figure 2(c)), whereas the CellBE implementation is coarse-grained, that is, each thread compares the same query sequence to a different sequence. Liu et al. [2013] proposed CUDASW++ 3.0, which uses simultaneously CPUs and GPUs to compare protein sequences. Static load distribution based on clock frequencies, number of cores (CPUs), and number of SMs (GPUs) is used to assign work to the CPUs or GPUs. The CPU code is based on SWIPE [Rognes 2011], and the GPU code uses CUDA PTX SIMD instructions combined with the ideas of Farrar [2007] to accelerate the computation. Hamidouche et al. [2013] propose the use of a high-level parallel programming library called BSP++ [Hamidouche et al. 2010] to execute local comparisons in several HPC platforms. A fine-grained parallel implementation for a cluster of CellBEs and multicores is proposed that uses a block-cyclic approach to distribute square partitions of the DP matrix among the processing units.

A parallel exact algorithm that generates global alignments for Megabase sequences was proposed in Rajko and Aluru [2004] for a cluster of CPUs. The key idea of this algorithm is to use Hirshberg's algorithm (Section 2.3) combined with PP computations to find a partial balanced partition between subsequences of S_0 and S_1. This partition allows the subdivision of the original problem in independent subproblems that can be solved in parallel. CUDAlign 2.1 was proposed by Sandes and Melo [2013b] and is a combination of the Gotoh (Section 2.1) and MM algorithms (Section 2.3), retrieving optimal local alignments in linear space for Megabase sequences in the GPU. It computes optimal local alignments in 5 stages. Stage 1 obtains the optimal score with antidiagonal parallelism (Figure 2(c)) and BP optimization, saving some rows to disk. Stages 2 to 5 implement the traceback, executing a modified version of MM, retrieving the coordinates of the points that belong to the optimal local alignment in a divide-and-conquer way. Korpar and Sikic [2013] proposed SW#, which is an approach that implements the MM algorithm (Section 2.3) with the parallelization strategy and the BP optimization proposed in CUDAlign 2.1 [Sandes and Melo 2013b] for retrieving the local alignment between Megabase DNA sequences in the GPU. Liu et al. [2014] proposed SWAPHI-LS, a strategy to execute local comparisons of Megabase sequences with one or more Intel Phis. As output, the optimal score is provided. The DP matrix is divided into blocks that are processed by antidiagonals (Figure 2(c)). Each block is further divided into small tiles of fixed size, which are distributed to a team of threads. Each thread calculates its tile in a vectorized way, using SIMD instructions.

Table I presents a comparative view of the papers discussed in this section. In this table, we present the paper, the maximum query sequence size used in the experimental results, the supported alignment types, the output produced and, finally, the processing units/programming environments used in each paper. In Table I, the cited papers are grouped into three different classes. In the "Multiple Type of Alignments" group, the papers were able to obtain the alignments for local, global, semiglobal, synthenic and sequence profile comparison types. Although they used different types of hardware (CellBE, ASIC, CPU, and GPU), these papers restricted the length of query sequences to some thousands of characters. In the "Multiple HW/SW Platforms" group, the papers used many platforms simultaneously (CPU, FPGA, CellBE, GPU) and different frameworks (OpenMP, Cilk, TBB, FastFlow, Handle-C, Cell SDK, CUDA, BSP++) to obtain the score of the optimal local alignment, without retrieving the full alignment. The maximum sequence sizes were greater than the previous group, increasing to sequences of more than a million base pairs. The group "Comparison of Megabase Sequences" contains papers that aligned sequences larger than 1Mbp. They all retrieve the optimal local score and many were able to retrieve the full alignment. The target platforms were CPU (using MPI), CUDA GPUs and the Intel Phi.

Table I. Comparative View of HPC Biological Sequence Comparison Papers

Paper	Max Size	Alignment Type	Output	Processing Units (Programming)
Multiple Types of Alignments				
Sarje and Aluru [2009]	10^3	local, spliced, synt	score, align	CellBE (Cell SDK)
Sarkar et al. [2010]	10^3	local, global, semig	score, align	ASIC (RTL *and* Cadence)
Maleki et al. [2014]	10^2	local, global, seq-prof	score, align	CPU (SSE *and* MPI)
Liu and Schmidt [2014a]	10^2	local, global, semig	score, align	GPU (CUDA)
Multiple HW/SW Platforms				
	10^4	local	score	CPU (OpenMP *or* Cilk *or* TBB *or* FastFlow)
Benkrid et al. [2012]	10^3	local	score	FPGA (Handel-C) CellBE (Cell SDK) GPU (CUDA)
	10^3	local	score	GPU (CUDA) CPU (SSE)
Hamidouche et al. [2013]	10^3 10^7	local	score	CellBE (BSP++) CPU (BSP++)
Comparison of Megabase Sequences				
Rajko and Aluru [2004]	10^6	local	score, align	CPU (MPI)
Sandes and Melo [2013b]	10^7	local	score, align	GPU (CUDA)
Korpar and Sikic [2013]	10^7	local	score, align	GPU (CUDA)
Liu et al. [2014]	10^7	local	score	Phi (IMCI)
This work (MASA)	10^7	local, global, semi-g	score, align	GPU (MASA *and* CUDA) Phi (MASA *and* OpenMP) CPU (MASA *and* OpenMP) CPU (MASA *and* OmpSs)

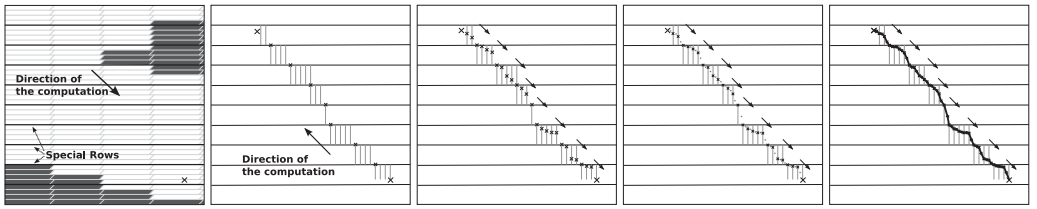
Our work (last row in Table I) is able to compare sequences with more than 10Mbp, as do the papers in the third group, generating global, local, or semiglobal alignments, as do the papers in the first group. In contrast to other works, in MASA, we are able to reuse the features/optimizations that belong to the platform-independent code in different implementations, enhancing the portability to other HPC platforms and potentially reducing the time to produce the codes.

4. THE CUDALIGN ALGORITHM

In this section, we give an overview of CUDAlign 2.1, which was used to generate the MASA code. CUDAlign 2.1 [Sandes and Melo 2013a] is a GPU algorithm that retrieves the optimal local alignment between two Megabase DNA sequences with affine gaps in linear space. In order to find the alignment, CUDAlign combines Gotoh and MM (Sections 2.1 and 2.3) and iteratively obtains coordinates of the optimal alignment, which are incrementally refined until the full alignment is obtained.

CUDAlign 2.1 is computed in 6 stages (Figure 3). The first three stages execute in the GPU and Stages 4, 5, and 6 execute in the CPU. In general terms, the algorithm calculates an increasing number of crosspoints in Stages 1 to 4, and Stage 5 is employed to align and concatenate results from all partitions. Further, Stage 6 is optionally used for visualization of the alignment.

Stage 1 is the most compute-intensive phase of CUDAlign since it calculates the Gotoh's matrix in linear space. Its goal is to retrieve the optimal score and the position in the DP matrix where it occurs. This position marks the end of the optimal alignment. The optimization BP [Sandes and Melo 2013a] is executed in this stage to reduce the



(a) Stage 1 (b) Stage 2 (c) Stage 3 (d) Stage 4 (e) Stage 5

Fig. 3. The CUDAlign Algorithm. (a) Stage 1: finds the optimal score and its position. Special rows are saved and some blocks are pruned (*gray*); (b) Stage 2: computes crosspoints between optimal alignment and special rows. Special columns are saved; (c) Stage 3: finds more crosspoints inside partitions; (d) Stage 4: executes Myers-Miller (MM) algorithm between successive crosspoints; (e) Stage 5: aligns partitions and obtains full alignment. Stage 6 is not represented in the figure since it is optional and creates only graphical and textual versions of the alignment.

number of cells calculated. A pruned block has such a small departing score that it is mathematically impossible to have an optimal alignment that crosses it. Thus, a pruned block does not contribute to the optimal score and therefore does not need to be calculated. Also in this stage, some rows (called special rows) are saved to disk in order to accelerate the computation in the next stages. This strategy is derived from FastLSA [Driga et al. 2006]. The number of special rows is a parameter of CUDAlign, chosen by the user. As the number of special rows increases, the area processed by Stages 2 and 3 is reduced. Figure 3(a) illustrates Stage 1.

Stage 2 computes the semiglobal recurrence relation using the MM matching procedure (Section 2.3) in the reverse direction of Stage 1, starting at the end point of the alignment obtained in Stage 1. The main goal of Stage 2 is to find all the points that belong to the optimal alignment and cross the special rows (crosspoints), including the start point. The orthogonal execution optimization [Sandes and Melo 2011] is used to reduce the area calculated by the MM matching procedure. Special columns are saved to disk. Figure 3(b) presents Stage 2.

Stage 3 is very similar to Stage 2 (Figure 3(c)). The main difference is that Stage 3 uses the special columns saved to disk in the previous stage to compute more crosspoints. These new crosspoints are found inside the partitions (area delimited by consecutive crosspoints) obtained in Stage 2.

Stage 4 is a multithreaded version of MM (Section 2.3) that computes the points that belong to the alignment inside partitions found in Stage 3. It adds more crosspoints to the solution until very small partitions are obtained (Figure 3(d)).

Stage 5 executes the NW algorithm (Section 2.1) to align the partitions formed by crosspoints found in Stage 4 and concatenates the results to obtain the full optimal alignment (Figure 3(e)).

Stage 6 is an optional stage used only for visualization purposes.

5. DESIGN OF MASA

The main goal of MASA is to provide a flexible and customizable infrastructure to develop sequence aligners in multiple hardware/software platforms. It proposes and implements a set of platform-independent modules that may be reused by multiple platform-specific implementations. It also allows platform-independent optimizations to be deployed to many platform-specific implementations.

In order to design MASA, we analyzed the code of CUDAlign 2.1 (Section 4) to determine which parts of the code are platform-independent or platform-specific. Most of the execution time of CUDAlign is spent in CUDA kernels, calculating the recurrence relation (SW/NW); this part is platform-specific. There are other parts

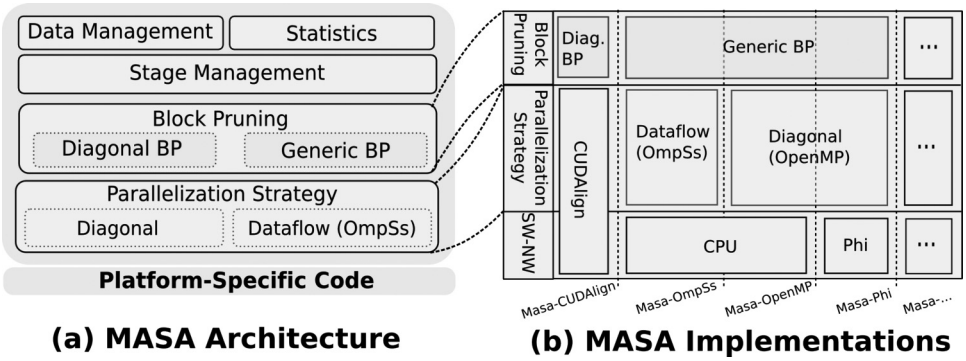


Fig. 4. Overview of MASA: (a) presents the MASA modules, including platform-independent optimizations that are selected by the developer, such as Block Pruning and Parallelization strategy, as well as the platform-specific code executed on the target processor to compute the NW/SW recurrences; (b) shows multiple Aligners we developed on top of MASA with their choice of block pruning and parallelization strategies.

that are platform-independent, such as input/output operations, profiling, and stage coordination.

We also analyzed BP and the parallelization strategy. Since BP (Section 4) is the most relevant CUDAlign optimization, we decided to implement it in a platform-independent way, with sufficient flexibility to be used even when a generic block execution order is used (Figure 2(f)). In the current version of MASA, BP works for local alignments, as in the previous versions of CUDAlign. The parallelization strategy employed by CUDA depends on the GPU hardware; thus it is platform-specific. Nevertheless, other implementations may benefit from parallelization strategies. Thus, as with BP, we decided that the parallelization strategy should be implemented in a platform-independent way. We call "MASA implementation" the union of the platform-specific and platform-independent codes, with selected customizations. Each implementation creates a single binary file that can be executed in a given platform.

Although the MASA code was originally based on the CUDAlign 2.1 code, new features were also included in a platform-independent way. For instance, CUDAlign 2.1 stores special rows only in disk, but MASA can store rows in memory and disk, with simultaneous usage, if necessary. Multinode support was also included in Stage 1, with the ideas presented in Sandes et al. [2014b], with heterogeneous platform support [Sandes et al. 2014a]. Stage 3 was also redesigned in order to iterate many times if the partitions are still too large for Stage 4. Furthermore, MASA can also produce the optimal alignment based on any of the 25 types defined in Section 2.2. With all these features, we can note the benefit of using the MASA platform, since a new feature can be easily delivered to all different implementations.

5.1. MASA Architecture

The architecture of MASA is divided in 5 modules, as shown in Figure 4(a). *Data Management*, *Statistics*, and *Stage Management* are modules used by all implementations. *Block Pruning* and *Parallelization Strategy* are customizable modules; the developer can choose the strategies provided by MASA or provide one's own strategy. The platform-specific code calculates the SW/NW equations.

Data Management: This module is responsible for managing data such as input sequences, user parameters, special rows/columns, optimal alignment, and score. Once a MASA implementation is dispatched for execution, it queries this module to retrieve the data used as input as well as to store results.

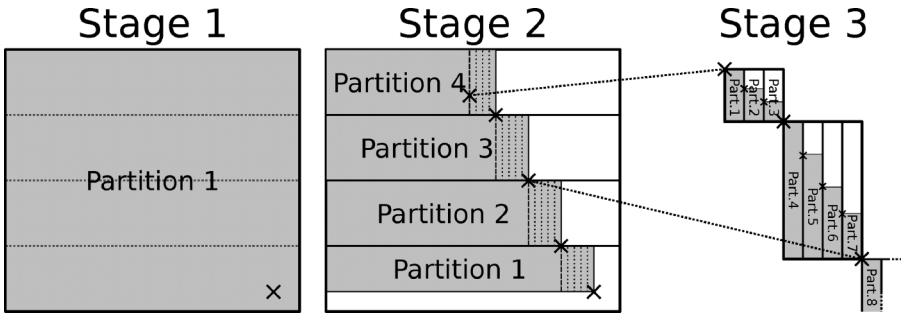
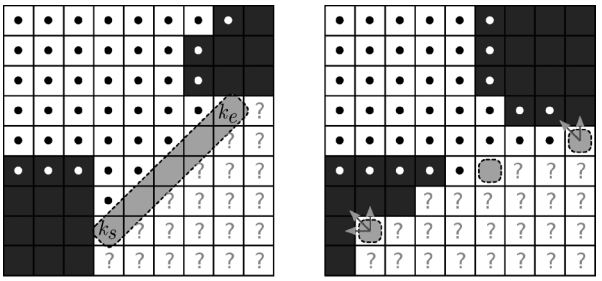


Fig. 5. MASA partition processing.



(a) Diagonal BP (b) Generic BP

Fig. 6. Block Pruning Strategies. Blocks with a circle are fully processed, black blocks are prunable, dashed blocks are the ones being processed, and question marks indicate blocks that were not processed yet. Diagonal BP processes blocks of the same diagonal in parallel. Generic BP processes blocks of different diagonals in parallel.

Statistics: This module provides information about the execution such as execution time of each stage, amount of memory/disk used and percentage of blocks pruned, among others.

Stage Management: This module is responsible for coordinating the execution of Stages 1 to 3 and for executing Stages 4 to 6 in the CPU. During the execution of Stages 1 to 3, it divides the DP matrix into partitions (Figure 5) and dispatches each partition to be computed by the platform-dependent code. This code receives the first row and first column of a partition and provides as output the last row and last column computed, as well as the highest score found and the special rows/columns. In Stage 1, there is a single partition whose size is the whole matrix; in the other stages, there are several smaller partitions, creating boundaries over the special rows/columns. When this module receives the border cells of a partition, it starts to search for crosspoints with the MM matching procedure with the orthogonal execution optimization (Section 4). As soon as it finds them, the computation of the partition stops and the next partition is dispatched.

Block Pruning (BP): We redesigned the BP optimization proposed in CUDAlign 2.1 in a platform-independent way and called it Diagonal BP in MASA. To perform the pruning, Diagonal BP must keep track of a nonprunable window $[k_s..k_e]$ containing the blocks that must be computed in each diagonal. Thus, two pointers k_s and k_e are stored in memory, being updated after each diagonal computation (Figure 6(a)). Since MASA aims to support multiple implementations of aligners, including generic

dataflow parallelization, we have extended Diagonal BP and created a new Generic BP strategy (Figure 6(b)). In Generic BP, we cannot maintain only two pointers as in Diagonal BP because many blocks from different diagonals can be calculated in parallel in a considerable number of concurrent scenarios. Thus, Generic BP maintains a matrix containing the prunable state of each block. If the neighbors of a block are all prunable, we infer that this block is also prunable, thus expanding the prunable area (Figure 6(b)). This implementation uses $O(B_h \times B_v)$ of memory, where B_h is the number of horizontal blocks in the DP matrix and B_v is the number of vertical blocks. A detailed description of Generic BP is presented in Section 6.

Parallelization strategy: SW, NW, and Gotoh's recurrence relations have the same dependency pattern, that is, the computation of a cell (i, j) depends on cells $(i - 1, j)$, $(i - 1, j - 1)$, and $(i, j - 1)$ (Section 2.4). To achieve better results, cells are grouped in blocks, maintaining the same dependency pattern between the blocks. Considering this, MASA provides two strategies to exploit parallelism: Diagonal and Dataflow.

In the Diagonal method, computations start at the top-left corner block of cells and propagate diagonally (Figure 2(c)). Blocks composing the same diagonal can be computed in parallel. The main limitation here is the frequent synchronization points at the end of each diagonal computation.

Dataflow parallelization is proposed to reduce the synchronization steps of the Diagonal method. In this case, the generic method (Figure 2(f)) is implemented as a dataflow in which each node of the dataflow is a block of cells. When the dependencies of a block are resolved, this block is ready for execution. Data dependencies are resolved during the execution, reducing synchronization overheads of the Diagonal method.

5.1.1. MASA-API. The MASA architecture was developed based on the object-oriented paradigm, using the C++ programming language. The MASA-API is presented as a class hierarchy in Figure 7. The IAligner class is the interface point between MASA stages and Aligners implementations. Each MASA implementation must create its own *Aligner* class that implements the IAligner virtual methods. The Aligner implementation communicates with the stages using an IManager implementation, which contains 4 types of methods: get methods, which return alignment parameters; must methods, which dictate runtime behavior for the Aligner; receive methods, which transfer to the Aligner initial rows and columns of the matrix; and dispatch methods, which send matrix cells to the MASA stage modules. Some of the IManager methods are listed in Table II.

Each IAligner implementation must implement some methods in order to be instantiated. The initialize and finalize methods are called only once during the alignment life cyle; they must be used to initialize and finalize any required resource (e.g., memory and accelerator hardware) for the alignment execution. Then, each stage calls the IAligner::setSequence and IAligner::unsetSequence to define the sequence interval and direction that will be used during the stage. During the stage execution, one or more partitions are aligned. The aligner executes this job in the IAligner::alignPartition method.

In order to simplify the creation of an IAligner subclass, there is a class hierarchy with different types of aligners, which are represented in Figure 7 as yellow classes. The AbstractAligner class encapsulates the IManager methods and initializes the grid and the BP operations. BP is handled by the AbstractBlockPruning class and its subclasses, shown in Figure 7 as blue classes. There are two other types of aligners: AbstractBlockAligner and AbstractDiagonalAligner.

AbstractBlockAligner calculates the matrix on a block basis, using the GenericBP BP strategy. Each AbstractBlockAligner subclass (OpenMP and OmpSs Aligners) must implement its own scheduler mechanism and each block is processed by a

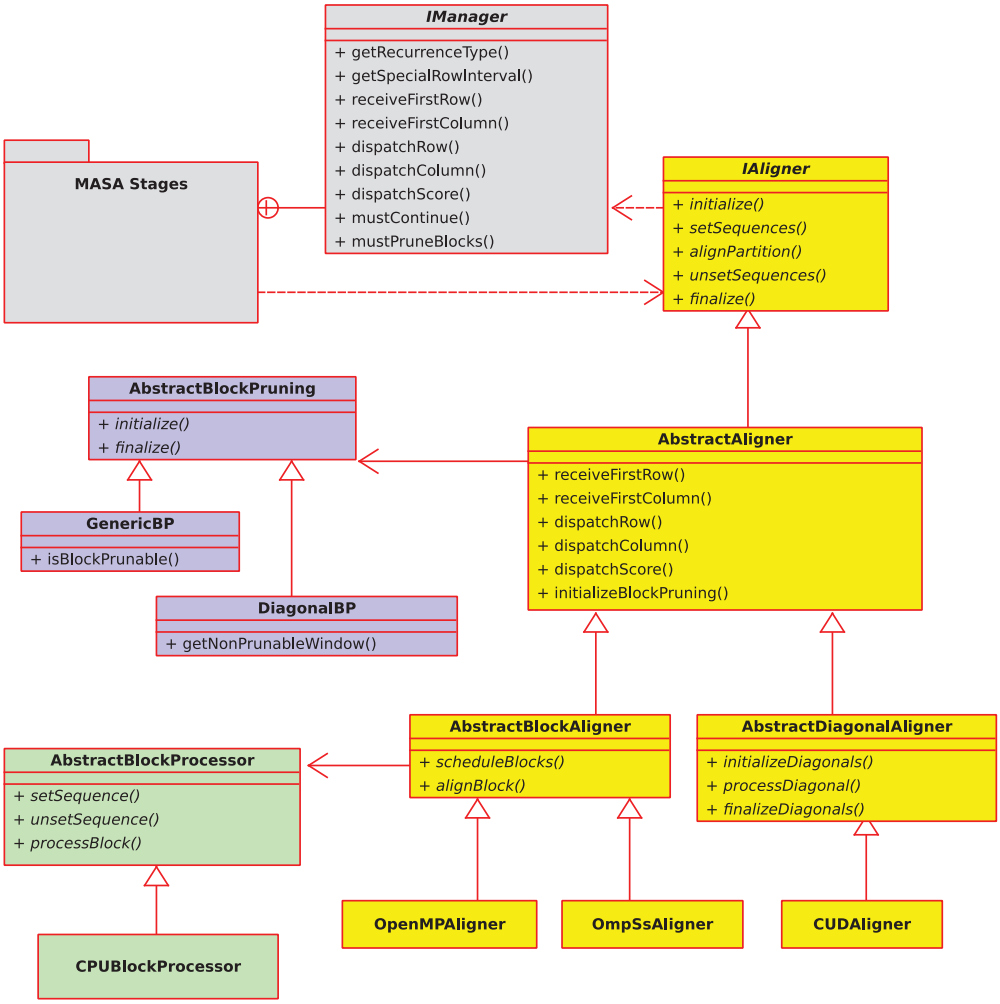


Fig. 7. MASA-API: Class Diagram.

Table II. Some IManager Methods

Name	Description
getRecurrenceType	Recurrence type: SW or NW
getSpecialRowInterval	Minimum distance between special rows
receiveFirstRow	Receives the first row of a partition
receiveFirstColumn	Receives the first column of a partition
dispatchRow	Outputs special rows of the partition
dispatchColumn	Outputs special columns of the partition
dispatchScore	Outputs the best score found in each block
mustContinue	If *false*, the aligner must stop execution
mustPruneBlocks	Defines if the aligner can prune blocks

subclass of an AbstractBlockProcessor, represented in Figure 7 as green classes. The CPUBlockProcessor is a subclass that processes blocks using the conventional SW/NW code in CPU. Other Block Processors could, for instance, use FPGA or SSE instructions to process the blocks.

The AbstractDiagonalAligner class calculates the DP matrix with the wavefront strategy, using the DiagonalBP block pruning strategy. The CUDAligner subclass extends this subclass and, whenever the processDiagonal method is called, it executes a new diagonal of blocks in the GPU. Although the AbstractBlockAligner can also implement a scheduler mechanism for diagonal processing, the AbstractDiagonalAligner allows the entire diagonal to be executed at once in the GPU. Thus, the CUDA architecture is able to schedule the blocks using its own scheduler.

5.2. Creating a MASA Implementation

To create a new MASA implementation, the programmer develops an aligner class (which must implement the IAligner interface or extend one of the abstract aligner classes) and supplies one instance of this class to the MASA entry point. Then, the platform-independent code processes the arguments, reads the sequences, and coordinates the stage execution. When the SW/NW equation needs to be computed, the aligner object is invoked (`alignPartition` procedure). It receives the boundary coordinates of the partition, the first column/row, and computes the SW/NW equation. Then, the aligner uses MASA functions to inform (dispatch) the best score, the special rows, and the last column of the partition.

Algorithm 1 illustrates a simplified MASA implementation based on the AbstractBlockAligner. The concrete aligner class will simply be called Aligner. The basic memory initialization, the grid partitioning, and the BP initialization are transparently done by the AbstractBlockAligner initialization. Then, the specific `scheduleBlocks` method (Lines 2–6) iterates through the grid per diagonal and calls `alignBlock` for each block (Line 4). The `alignBlock` procedure receives from MASA the first row (Line 10) or column (Line 11) of the neighbor blocks. The BP test is made by MASA (Line 12) and, if this block is not pruned, the aligner calls `processBlock` (Line 13) in order to compute the SW/NW recurrence relation. The best score found is dispatched to MASA (Line 14). Special rows (Line 16) and the last column (line 17) are also dispatched to MASA. The program entry point (Lines 20–23) creates the aligner object and passes it to the MASA entry point (Line 22) using the default CPUBlockProcessor (Line 21).

ALGORITHM 1: Aligner Pseudocode—Block Based

```
1: procedure Aligner::scheduleBlocks
2:     for each diagonal do
3:         for each block in diagonal do
4:             alignBlock(block)
5:         end for
6:     end for
7: end procedure
8:
9: procedure Aligner::alignBlock(block)
10:     if block is in first row then block.row ← receiveFirstRow
11:     if block is in first column then block.col ← receiveFirstColumn
12:     if Not isBlockPruned(block) then
13:         block.score := processBlock(block)
14:         dispatchScore(block.score)
15:     end if
16:     if isSpecialRow(block.row) then dispatchRow(block.row)
17:     if block is in last column then dispatchColumn(block.col)
18: end procedure
19:
20: procedure main(args)
21:     processor = new CPUBlockProcessor()
22:     MASA::entryPoint(args, new Aligner(processor))
23: end procedure
```

The `processBlock` method is implemented in the AbstractBlockAligner class; it essentially reads the first row/column from the `block.row` and `block.col` arrays, computes the SW/NW recurrence relation, and stores the last row/column into the same `block.row` and `block.col` arrays. Rows and columns of the blocks are chained so that the last row/column of a block is the first row/column of the next block. The `processBlock` procedure is compute-intensive, and this code is very suitable for platform-specific optimization. Furthermore, other third-party tools that execute optimized DNA sequence comparison may be adapted to MASA. This can be done with the reimplementation of the `processBlock` method using the recurrence relation source code of the third-party tool, with some modifications to fit the `processBlock` input/output parameters.

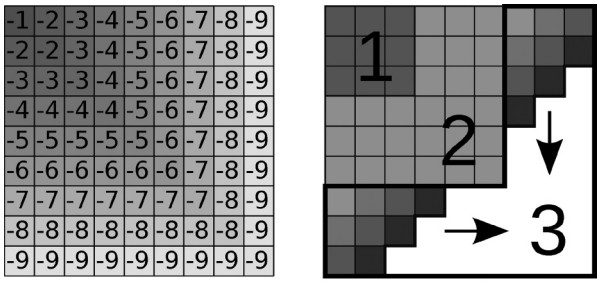
5.3. MASA Implementations

In this section, we present our four MASA implementations, which use different programming models/tools (OpenMP, OmpSs, and CUDA) and target different hardware platforms (multicore, GPU, Intel Phi). Each one of our MASA implementations (Figure 4(b)) used the affine gap model (Section 2.1) and present some modification in Algorithm 1. To give an idea in terms of number of code lines (excluding blank lines and comments), the platform-specific code contains 116 (OpenMP), 187 (OmpSs), and 1493 (CUDAlign) lines, whereas the platform-independent source code contains more than 15,000 lines.

5.3.1. MASA-OpenMP/CPU. MASA-OpenMP/CPU uses OpenMP to compute blocks that belong to the same diagonal in parallel. In Algorithm 1, "`#pragma omp parallel for schedule(dynamic,1)`" is inserted before Line 3. Dynamic scheduling is used because the size of the parallel loop is usually larger than the number of threads. The `processBlock` function is a CPU implementation of SW/NW, without vectorized instructions (SIMD).

5.3.2. MASA-OpenMP/Phi. MASA-OpenMP/Phi employs the same parallelization strategy as MASA-OpenMP/CPU, in which independent threads process each diagonal in parallel. It also uses the dynamic OpenMP scheduler, since this option leads to better performance. Our implementation uses the Intel Phi native execution mode, that is, the entire application (including MASA) runs within the coprocessor. This was possible because Intel Phi runs a specialized Linux kernel that provides the necessary OS-level services.

Because Intel Phi is equipped with 512b wide SIMD instructions, we have tried to modify the align partition code, as compared to the MASA-OpenMP/CPU, so that computation of cells in a diagonal are an internal loop in the computation. With this modification, we were able to leverage the Intel compiler tools and vectorize this operation. However, the vectorized version of our Phi-based code did not attain significant gains on top of the nonvectorized code that was generated from a cross-compilation of the MASA-OpenMP/CPU code. This occurred because the modified code used for vectorization has a larger number of instructions and branch statements in the inner loop as compared to the original code. In particular, the branch statements may strongly limit improvements with vectorization [Tian et al. 2013]. This finding is similar to that of Farrar [2007], which proposed a striped version of SW with no branching in the inner computation loop in order to maximize improvements with vectorization. Still, our Intel Phi-based aligner attains good performance and compares well to other CPU-based implementations. Therefore, we use this implementation to demonstrate the MASA flexibility. As a future work, we intend to port the Farrar striped NW algorithm [Farrar 2007] to Intel Phi, such as proposed in Liu and Schmidt [2014b], and include it in MASA.



(a) Priorities (b) Creation Order

Fig. 8. Task priorities and task creation order in OmpSs.

5.3.3. MASA-OmpSs/CPU. MASA-OmpSs/CPU uses the OmpSs parallel programming environment [Duran et al. 2011] and therefore its dataflow-based parallelization.

OmpSs proposes a unified programming model for heterogeneous systems, incorporating ideas of OpenMP and adding support for irregular and asynchronous parallelism. OmpSs applications are described as dataflows and use several optimizations, including efficient and automatic data movement, performed at compiler and runtime system levels. Parallelization of an application in OmpSs is carried out with a set of "`#pragma`" directives. The programmer defines portions of code that should be executed as tasks in the dataflow. The dependencies among tasks are derived from pragma options defining the task variables, which can be used as input and/or output. OmpSs applications also benefit from several task-scheduling policies.

In MASA-OmpSs/CPU, the code sections that invoke `processBlock`, `dispatchRow`, and `dispatchColumn` methods are annotated as OmpSs tasks, so that the OmpSs compiler can create the appropriate structures to dispatch the tasks. The same annotation also includes hints about the type of data used (input or output). With this, data dependencies among tasks are identified and used by the runtime system to guarantee correctness in the execution.

MASA-OmpSs/CPU uses the dataflow parallelization strategy (Figure 4(b)). Each call to `alignBlock` (Algorithm 1, Lines 9–18) creates up to three tasks: (1) processBlock task: Lines 12 to 15; (2) dispatchRow task: Line 16; and (3) dispatchColumn task: Line 17. Task dependencies are created considering the `block.row` and `block.col` vectors. Since OmpSs is able to execute blocks in generic order (Figure 2(f)), we set the block priorities to create a preferable execution order in square waves (Figure 8(a)). Considering block (bx, by), its priority is $min(-bx, -by)$. The OmpSs hysteresis throttle mechanism was enabled to limit the number of tasks in the task graph, reducing the amount of memory used by OmpSs. When the number of tasks per thread reaches a limit, task creation stops until it drops below a given value. Our goal is to keep a sufficient level of parallelism while reducing the amount of memory used by OmpSs. To prevent loss of parallelism during the hysteresis drop-down phase, the task creation order was changed to follow the priority order. Thus, tasks are created in lanes (Figure 8(b)). The width of the lanes is the number of parallel threads, which is equal to the number of cores in the running environment. Inside each lane, the block-creation order follows a diagonal creation (third lane in Figure 8(b)).

5.3.4. MASA-CUDAlign. Since CUDAlign was created before MASA, the original CUDAlign 2.1 code was basically kept the same as in Sandes and Melo [2013a], with some modifications to fit in the aligner-class hierarchy. Instead of using the AbstractBlockAligner, MASA-CUDAlign uses the AbstractDiagonalAligner as a base class. Thus, a `processDiagonal` method is called for each diagonal and the CUDAligner class launches

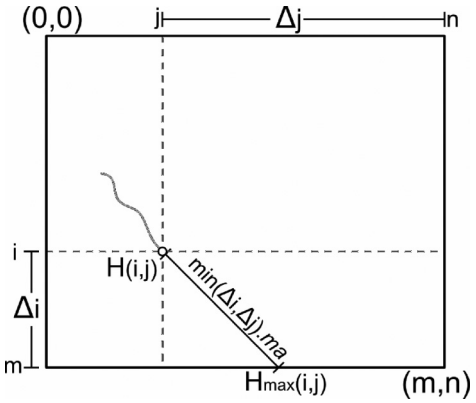


Fig. 9. Representation of the cell-based pruning definitions.

a kernel in the GPU. All memory allocation and transfers are made by the aligner class; thus, much more control is needed by this class. Calls to MASA functions (i.e., dispatch and receive methods) are always made by the CPU, usually preceded/followed by `cudaMemcpy` calls to transfer data from/to the GPU.

6. GENERIC BLOCK PRUNING

The BP optimization was designed to avoid the calculation of blocks of cells that do not contribute to the optimal local alignment. Sandes and Melo [2013a] have proposed a block pruning technique for antidiagonal execution order; in this article, we propose and implement BP for a generic execution order.

The antidiagonal BP must keep track of a nonprunable window $[k_s..k_e]$ containing the blocks that must be computed in the current diagonal. As such, the pointers k_s and k_e are stored in memory, and are updated after each diagonal computation (Figure 6(a)). In the generic execution order, blocks from multiple diagonals may be computed concurrently; as a consequence, keeping those two pointers is not sufficient (Figure 6(b)). Therefore, we have created a more elaborate BP strategy to work with the generic execution order, which is required by the novel parallelization strategies proposed and implemented in MASA.

6.1. Definitions

Let S_0 and S_1 be the sequences being aligned, with sizes $|S_0| = m$ and $|S_1| = n$. The punctuation for matches and mismatches are ma and mi, respectively. Suppose that cell (i, j) of the DP matrix has a score $H(i, j)$. The maximum score of any alignment that passes through cell (i, j) is defined in Equation 6:

$$H_{max}(i, j) = H(i, j) + H_{inc}(i, j), \tag{6}$$

where $H_{inc}(i, j)$ is the incremental score considering a matching of all remaining characters from subsequences $S_0[i..m]$ and $S_1[j..n]$. Additionally, the H_{inc} function may be calculated using Equation (7) for local alignments; it is also presented in Figure 9.

$$H_{inc}(i, j) = min(m - i, n - j).ma \tag{7}$$

We also define two special types of cells that are used in the pruning algorithms: *prunable* and *pruned* cells. A *prunable cell* is a cell that cannot be part of an alignment with a score higher than the maximum score currently known (H_{best}). In other words, a cell (i, j) is prunable if $H_{max}(i, j) \leq H_{best}$, since its computation will have no impact on the final optimal score. If cells $(i - 1, j)$, $(i - 1, j - 1)$, and $(i, j - 1)$ are already known

to be prunable, then it may be inferred that cell (i, j) is also prunable. In this case, cell (i, j) is called a *pruned cell*. Note that the *pruned cell* is considered prunable even before its computation.

Pruning conditions for the block-based execution can be similarly derived. Assume that the input DP matrix is divided into $B_h \times B_w$ blocks. Each block $b_{i,j}$ depends on blocks $b_{i-1,j}$, $b_{i,j-1}$, and $b_{i-1,j-1}$ (when they exist). For each block, we define (i', j') as the top-left cell coordinates and (i'', j'') as the bottom-right cell coordinates. The height and width of a block $b_{i,j}$ are given by Δ_i and Δ_j, respectively. Therefore, the maximum possible score of any alignment that passes through block (i, j) can be defined as shown in Equation (8):

$$H'_{max}(b_{i,j}) = H'(b_{i,j}) + H'_{inc}(b_{i,j}), \tag{8}$$

where $H'(b_{i,j})$ is the best score of that block and $H'_{inc}(b_{i,j})$ is the highest $H_{inc}(i, j)$ from all cells (i, j) inside the block. An upper bound for $H'_{inc}(b_{i,j})$ may be calculated as follows: considering that the best score of the block is in the top-left cell (i', j'), we can define the H'_{inc} function as presented in Equation (9):

$$H'_{inc}(b_{i,j}) = H_{inc}(i', j').ma. \tag{9}$$

In a similar way to the cells case, a *prunable block* is a block whose cells cannot generate an alignment with a score higher than the currently best score found, that is, $H'_{max} \leq H'_{best}$. Therefore, all the cells of a prunable block will not contribute to the optimal score. A block $(b_{i,j})$ is a *pruned block* if blocks $(b_{i-1,j})$, $(b_{i-1,j-1})$, and $(b_{i,j-1})$ are already known to be prunable. The difference between a prunable and a pruned block is that we need to calculate the score of the first in order to know that the block status is prunable. The pruned block, on the other hand, is known prior to its execution by inference from the status of neighboring blocks.

6.2. Generic Block Pruning Procedure

The generic BP procedure maintains a matrix k with an entry per block containing the prunable state of each block. Cell (i, j) of block k $(k_{i,j})$ is true if block (i, j) is prunable. We assume block coordinates starting from 1, that is, the first block is $(1, 1)$. For the sake of simplicity, we created an additional row/column in k (row/column 0) that is initialized with `true`, to represent pruned blocks, except for $k_{0,0}$, which is set to `false` in order to force the computation of the first block $(1, 1)$. The remaining matrix cells are set to `false`. The matrix k is initialized with Equation (10):

$$k[0..B_h][0..B_v] \leftarrow \begin{pmatrix} \text{false} & \textbf{true} & \textbf{true} & \cdots & \textbf{true} \\ \textbf{true} & \text{false} & \text{false} & \cdots & \text{false} \\ \textbf{true} & \text{false} & \text{false} & \cdots & \text{false} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \textbf{true} & \text{false} & \text{false} & \cdots & \text{false} \end{pmatrix}. \tag{10}$$

The matrix initialization is the first step of the generic pruning procedure, as presented in Algorithm 2.

The `isPruned` function (Lines 2 to 8) is a pretest invoked before the execution of each block to identify if that block is pruned, which would avoid the block computation. It simply infers the status of the current block (i, j) from its neighbors, and returns true if $k_{i-1,j}$, $k_{i-1,j-1}$, and $k_{i,j-1}$ are true.

If the current block is not pruned, its computation is carried out normally and the `pruningUpdate` method (Lines 14 to 19) is executed after the block calculation to update the status of that block in matrix k. The `pruningUpdate` method tries to update the current maximum score (H'_{best}) with the score just obtained as the result of the computation of block $b_{i,j}$.

ALGORITHM 2: Pruning Strategy—Generic

```
1: k[0..B_h][0..B_v] ← Equation (10)

2: function isPruned(b_{i,j})
3:     if (k[i − 1][j] is true) and (k[i][j − 1] is true) and (k[i − 1][j − 1] is true) then
4:         return true;
5:     else
6:         return false;
7:     end if
8: end function

9: function IsPrunable(b_{i,j}, H′, H′_best)
10:    (i′, j′) ← GetMinCoord(b_i, b_j)
11:    H′_max ← H′ + H′_inc(b_{i,j})
12:    return H′_max < H′_best
13: end function

14: procedure pruningUpdate(b_{i,j}, H′)
15:    H′_best ← max(H′_best, H′)
16:    if IsPrunable(b_{i,j}, H′, H′_best) then
17:        k[i][j] ← true
18:    end if
19: end procedure
```

Further, *IsPrunable* (Lines 9 to 13) is executed to evaluate if block $b_{i,j}$ could still contribute to the optimal score. In other words, it computes whether block $b_{i,j}$ is prunable or not, considering its own maximum score H' and the current overall H'_{best} score. The H'_{inc} function must be defined according to Equation (9).

The generic BP algorithm presented here uses $O(B_h \times B_w)$ of memory to keep the pruning status of the blocks (size of k). Considering that the best performance of the algorithm is attained with a number of blocks proportional to the logarithmic of the sequence sizes (i.e., $B_h = O(log(m))$ and $B_w = O(log(n))$), in practice, the memory complexity would be $O(log(m) \times log(n))$. Therefore, it results in a moderate memory complexity increase, as compared to the Diagonal BP, with the advantage of allowing a general block execution order.

7. EXPERIMENTAL RESULTS

MASA was implemented in C/C++, and MASA implementations (Section 5.3) were programmed in CUDA 4.1, OpenMP 3.0, and OmpSs 1.0. The Minotauro GPU cluster, hosted in the Barcelona Supercomputing Center, was used in our tests. Minotauro is composed of 128 nodes; each node has two 6-core Intel Xeon E5649 and two NVIDIA Tesla M2090 boards. In our tests, we used only one Minotauro node. For the MASA-OpenMP/CPU and the MASA-OmpSs/CPU executions, we used 12 cores. MASA-CUDAlign used one GPU NVIDIA Tesla M2090. For the MASA-Phi executions, we used the Intel Xeon Phi SE10P coprocessor.

7.1. Sequences Used in the Tests

We compared real DNA sequences (Table III) retrieved from the National Center for Biotechnology Information (NCBI) at *www.ncbi.nlm.nih.gov*. The sequence sizes vary from 10Kbp (Thousand base pairs) to 47Mbp (Million base pairs). The SW/NW parameters used were: *ma* (match) +1; *mi* (mismatch) −3; G_{open}: −5; G_{ext}: −2. For validation purposes, the optimal scores obtained during our tests are also shown for local, overlap, semiglobal, and global comparisons. In our experiments, the semiglobal alignments are defined as the ones that contain all characters of sequence 2 (type (2/2) in Section 2.2), and overlap alignments are defined as in Durbin et al. [2002] (type (3/3)).

Table III. Sequences Used in the Tests

Cmp.	Sequence 1		Sequence 2		Optimal Score			
	Accession	Size	Accession	Size	Local	Overlap	Semiglobal	Global
10K	AF133821.1	10K	AY352275.1	10K	5091	4279	3594	2981
50K	NC_001715.1	57K	AF494279.1	57K	52	2	−60479	−63880
150K	NC_000898.1	162K	NC_007605.1	172K	18	1	−201600	−208667
500K	NC_003064.2	543K	NC_000914.1	536K	48	1	−554606	−585725
1M	CP000051.1	1M	AE002160.2	1M	88353	62525	−588728	−1189459
3M	BA000035.2	3M	BX927147.1	3M	4226	0	−2655752	−2662376
5M	AE016879.1	5M	AE017225.1	5M	5220960	5220960	5220955	5220950
7M	NC_005027.1	7M	NC_003997.3	5M	172	2	−6020449	−8201748
10M	NC_017186.1	10M	NC_014318.1	10M	10235188	10235188	10235188	10235188
23M	NT_033779.4	23M	NT_037436.3	25M	9063	0	−26746584	−27446770
47M	NC_000021.7	47M	BA000046.3	32M	27206434	27179500	−484675	−572719

7.2. Execution Times for Local Alignments

In our tests, the default block size for MASA-OpenMP/CPU and MASA-OmpSs/CPU was set to 1024×1024, but this size was automatically reduced if the number of blocks in a diagonal is less than $2 \times p$, where p is the number of cores. For MASA-CUDAlign, we used $T = 128$ threads, $B = 512$ blocks, and $\alpha = 4$, resulting in a block height of $\alpha \cdot T \times \frac{n}{B} = 512 \times \frac{n}{512}$, where n is the width of the partition. For MASA-OmpSs/Phi, we empirically defined the optimal block size for each comparison by increasing the size of the block until it does not have any effect in the performance.

Table IV presents the Special Rows Area (SRA) used, the time spent in each stage, the overall runtime, GCUPS, the percentage of pruned blocks, and the block size in Stage 1. The overall runtime is the wall-clock time, including all initializations and I/O times outside the stages. GCUPS are calculated with $\frac{m \cdot n}{t} \cdot 10^9$, where m and n are the sizes of the sequences and t is the overall runtime. Some implementations did not execute all comparisons due to time constraints. The SRA was set to be stored in the RAM memory up to 4GB; the remaining bytes were saved into the disk.

MASA-CUDAlign presented the best performance for sequences ≥50K, with GCUPS ranging from 2.31 up to 54.74. These results are comparable with those presented in Sandes and Melo [2013a], observing that the M2090 GPU is slightly faster than the GTX 560 Ti GPU. This first set of results shows that the MASA architecture added negligible overheads to the GPU execution. For the smaller sequence (10K), MASA-CUDAlign is outperformed by other implementations, mainly because of the lack of parallelism to fully utilize the GPU.

The GCUPS varied from 0.57 to 4.81 (MASA-OpenMP/CPU), 0.41 to 5.95 (MASA-OmpSs/CPU) and 0.06 to 4.03 (for MASA-OpenMP/Phi). It is important to note that the MASA-OmpSs/CPU, MASA-OpenMP/CPU, and MASA-OpenMP/Phi implementations may be optimized with the use of a striped version of the code that could efficiently leverage optimizations for the vector instructions [Farrar 2007]. Thus, in future work, we intend to improve these versions by taking advantage of this kind of parallelism.

As expected, comparisons with higher similarity scores (Table III) present higher block pruning efficiency. Sequences 10K, 5M, 10M, and 47M are very similar, with block pruning rates above 40%. Sequences 1M present medium similarity, with pruning above 10%; other sequences have very few pruned blocks (below 1%).

The comparison of MASA-OmpSs/CPU (generic BP + dataflow parallel strategy) and MASA-OpenMP/CPU (generic BP + diagonal parallel strategy) execution times shows that the first is more efficient in most cases. The better performance of the MASA-OmpSs/CPU aligner is due mainly to the higher pruning rates combined with

Table IV. Execution Times, GCUPS, and Blocks Pruned for the MASA Implementations (Local Alignment

Cmp.	SRA	Ext	Stages (s)					Total		Pruned	Block Size
			1	2	3	4	5+6	Time	GCUPS		
10K	1M	CUDAlign	∼	∼	∼	0.2	∼	0.6	0.16	27.3%	512×321
		OmpSs/CPU	0.1	∼	∼	∼	∼	0.3	0.41	39.0%	418×428
		OpenMP/CPU	0.2	0.1	∼	∼	∼	**0.2**	**0.57**	37.8%	418×428
		OpenMP/Phi	0.7	0.4	0.2	0.2	∼	1.6	0.06	**42.8%**	129×129
50K	3M	CUDAlign	0.2	∼	∼	∼	∼	**1.4**	**2.31**	0.0%	512×271
		OmpSs/CPU	1.7	0.2	∼	∼	∼	1.9	1.72	0.0%	1024×1024
		OpenMP/CPU	1.8	0.2	∼	∼	∼	2.1	1.58	0.0%	1024×1024
		OpenMP/Phi	3.2	0.8	∼	∼	∼	4.1	0.78	0.0%	129×129
150K	5M	CUDAlign	1.3	∼	∼	∼	∼	**1.9**	**14.39**	0.0%	512×335
		OmpSs/CPU	13.0	1.3	∼	∼	∼	14.4	1.93	0.0%	1024×1024
		OpenMP/CPU	13.8	1.4	∼	∼	∼	15.2	1.83	0.0%	1024×1024
		OpenMP/Phi	17.8	1.8	∼	∼	∼	19.8	1.41	0.0%	168×159
500K	50M	CUDAlign	10.3	∼	∼	∼	∼	**11.6**	**25.00**	0.0%	512×1047
		OmpSs/CPU	134.1	0.9	∼	∼	∼	135.2	2.15	0.0%	1024×1024
		OpenMP/CPU	136.0	1.0	∼	∼	∼	137.1	2.12	0.0%	1024×1024
		OpenMP/Phi	160.1	12.9	∼	∼	∼	173.3	1.68	0.0%	266×263
1M	250M	CUDAlign	34.4	1.2	0.9	4.7	0.1	**41.9**	**26.75**	11.0%	512×2095
		OmpSs/CPU	453.8	22.4	1.8	3.3	0.1	481.7	2.33	**12.6%**	1024×1024
		OpenMP/CPU	462.2	23.4	1.1	1.0	0.1	488.1	2.30	11.9%	1024×1024
		OpenMP/Phi	538.1	33.8	11.1	21.6	1.0	605.9	1.85	12.0%	511×525
3M	1G	CUDAlign	331	∼	0.3	∼	∼	**333**	**31.04**	0.1%	512×6411
		OmpSs/CPU	4748	46.9	∼	0.1	∼	4796	2.15	**0.2%**	1024×1024
		OpenMP/CPU	4724	48.7	∼	0.1	∼	4773	2.16	**0.2%**	1024×1024
		OpenMP/Phi	5275	79.9	0.5	1.1	∼	5357	1.93	**0.2%**	1024×1024
5M	3G	CUDAlign	455	16.7	9.6	51.2	1.5	**536**	**51.02**	53.7%	512×10212
		OmpSs/CPU	4296	213.6	46.8	36.8	1.6	4595	5.95	**66.5%**	1024×1024
		OpenMP/CPU	5449	213.8	11.3	9.5	1.2	5685	4.81	57.5%	1024×1024
		OpenMP/Phi	6053	448.1	117.3	154.5	11.2	6785	4.03	57.5%	1024×1024
7M	3G	CUDAlign	1190	∼	∼	∼	∼	**1191**	**31.35**	0.0%	512×10209
		OmpSs/CPU	17080	109.4	∼	∼	∼	17190	2.17	0.0%	1024×1024
		OpenMP/CPU	17044	111.3	∼	∼	∼	17157	2.18	0.0%	1024×1024
		OpenMP/Phi	18504	176.1	∼	∼	∼	18682	2.00	0.0%	1024×1024
10M	5G	CUDAlign	1730	57.8	18.9	102.7	3.1	**1914**	**54.74**	53.3%	512×19993
		OmpSs/CPU	16572	903.3	182.2	163.0	2.2	17824	5.88	**66.5%**	1024×1024
		OpenMP/CPU	20838	928.9	36.7	41.6	3.1	21849	4.80	57.5%	1024×1024
23M	10G	CUDAlign	17785	∼	∼	0.2	∼	**17787**	**31.75**	0.0%	512×47936
47M	50G	CUDAlign	28029	288	61.2	341.6	15.6	**28738**	**53.58**	**48.1%**	512×91688

Note: Values in bold are the best values in each comparison. The symbol "∼" means negligible time.

the flexible parallel execution achieved by the dataflow strategy. For instance, for the 5M and 10M comparisons, MASA-OmpSs/CPU processed 21% less blocks than MASA-OpenMP/CPU, reducing the Stage 1 execution time in the same proportion. For the 1M comparison, the GCUPS rate of the MASA-OmpSs/CPU is also better than MASA-OpenMP/CPU, but in only 1% since the sequences are not so similar and the pruning rates are almost the same.

MASA-OmpSs/CPU presented the best pruning rate because the block priorities conduct the execution order to the square wave shape that, in the case of very similar sequences, obtains the highest scores near the main diagonal much earlier than the traditional diagonal wavefront method. The execution order of the matrix calculation

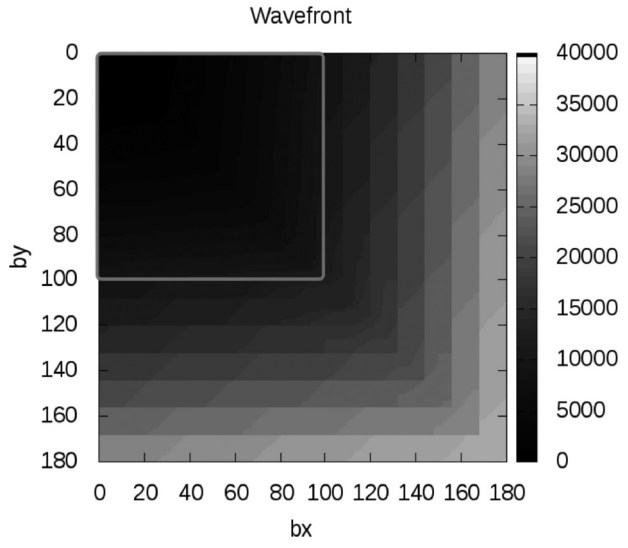
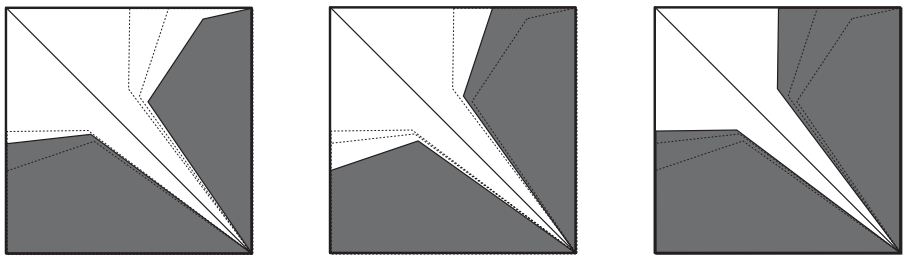


Fig. 10. MASA-OmpSs/CPU processing pattern.



(a) MASA-CUDAlign (b) MASA-OpenMP/CPU (c) MASA-OmpSs/CPU

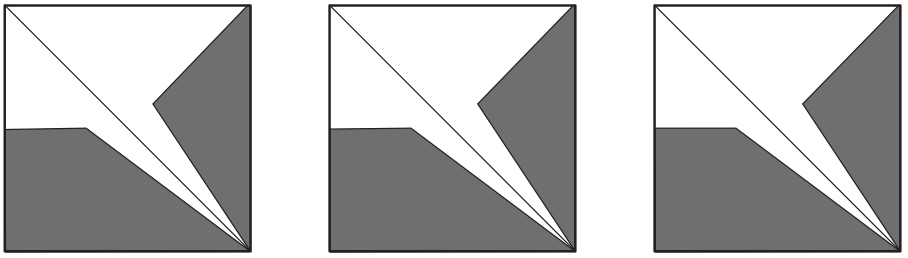
Fig. 11. Pruned areas (gray) when comparing sequence CP000051.1 (1Mbp) with itself (perfect match).

is shown in Figure 10, in which we see every 1000 executed blocks in a different shade of gray. The first 100×100 blocks (marked in a square in the figure) were executed similar to square wave, respecting the priorities set to each block. After the 100×100 blocks calculation, we can see that the lanes used during the task creation are much more noticeable. This happens because the hysteresis throttle pauses the task creation whenever the number of queued tasks reaches a maximum value, forcing threads to process nearby blocks, respecting the wavefront inside each lane. As soon as the threads are processing blocks in a diagonal wave inside each lane, this shape tends to persist until the end of the computation.

Although the block pruning and parallelization strategies in the MASA-OpenMP/CPU and MASA-OpenMP/Phi are the same, the percentage of pruned blocks varies slightly in some cases. This is due to the different block sizes used in both strategies. Specially in the 10K comparison, MASA-OpenMP/Phi achieved the best pruning efficiency compared with the other implementations since the blocks were much smaller.

7.3. Pruning Results for Perfect Match

This section presents the pruning results for each of the parallelization and pruning strategies used for the perfect match case (identical sequences) using local alignments.



(a) 23Mx23M comparison (b) 32Mx32M comparison (c) 47Mx47M comparison

Fig. 12. Other pruned areas (gray) with perfect match, using MASA-CUDAlign.

Figures 11(a), 11(b), and 11(c) show the pruned areas for MASA-CUDAlign (55.2%), MASA-OpenMP/CPU (57.3%), and MASA-OmpSs/CPU (66.2%), respectively, when comparing sequence CP000051.1 (1Mbp) with itself (perfect match). The line in the diagonal represents the optimal alignment and divides the pruned area into left and right sides. MASA-OmpSs/CPU presents the biggest pruned area in both sides. Both MASA-CUDAlign and MASA-OpenMP/CPU process the matrix by diagonals, but MASA-CUDAlign has wider blocks, leading to a less inclined wavefront and a different pruned area. Comparing MASA-CUDAlign and MASA-OpenMP/CPU, MASA-CUDAlign has a bigger pruned area in the left side and MASA-OpenMP/CPU in the right side. Nevertheless, the difference in the right side area is more significant than in the left side, leading to a better pruning performance in the MASA-OpenMP/CPU.

Figure 12 presents the MASA-CUDAlign pruned area for perfect matches with larger sequences (23Mbp, 32Mbp, and 47Mbp), where the pruning shape is visibly identical, with 53% of pruned blocks. The shapes shown in Figure 12 (23Mbp, 32Mbp, and 47Mbp comparisons) are slightly different from the shape shown in Figure 11(a) (1Mbp comparison), especially in the right upper border and in the middle row. This happens because the block heights are proportionally larger as we reduce the sequence size, increasing the diagonal wavefront angle.

7.4. Execution Times for Global, Semiglobal and Overlap Alignments

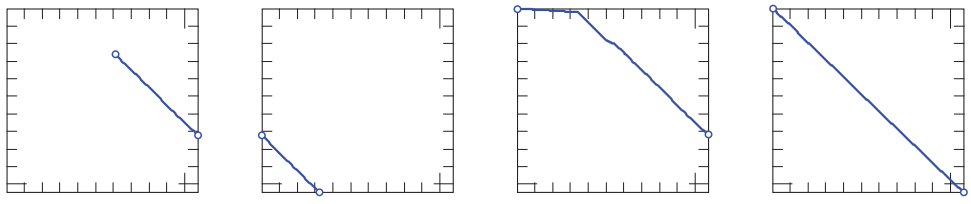
In this section, we compare the global, semiglobal, and overlap alignments against the local alignment, considering the MASA-CUDAlign implementation. For a fair comparison, BP was disabled and a new execution was made for the four types of comparisons. In some cases, the optimal overlap alignment can be very short and resides very near the corners of the matrix, limiting the results to positive values.

Table V presents the execution times and GCUPS for the MASA-CUDAlign without BP for the four types of alignment (local, overlap, semiglobal, and global). Since the global, semiglobal, and overlap recurrence relation (NW) does not have an if clause to avoid negative values, its Stage 1 performance is slightly better than the local recurrence relation (SW). This difference can be seen in Table V, in which the local alignment comparisons are around 3% slower than the other alignment types, in most cases.

Stages 2 to 6 execution times depend on the size of the alignment. The global alignment usually resides in the main diagonal of the matrix; its length is at least the size of the largest sequence. Thus, the global alignment is often larger than the other types of alignment, usually leading to a longer traceback time. The 47M local alignment (Figure 14(b)) and the overlap alignment reside in a shifted diagonal, shorter than the global and semiglobal alignments that inserted many gaps before the beginning of

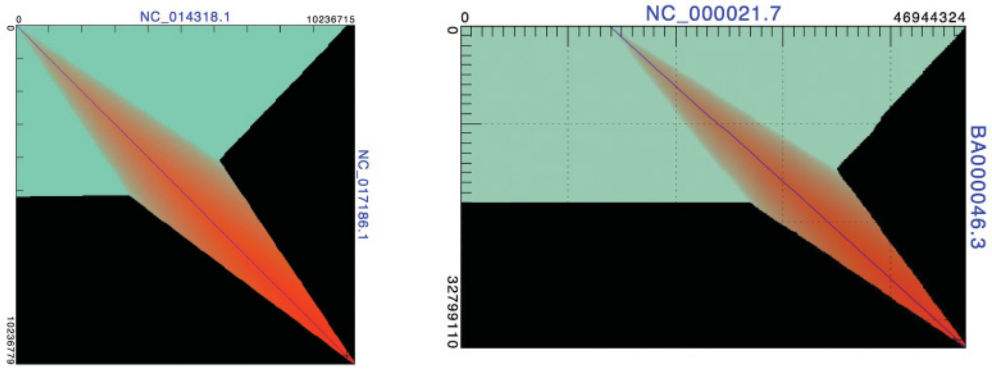
Table V. Execution Times and GCUPS for the MASA-CUDAlign without Block Pruning (Local, Overlap, Semiglobal and Global Alignments

Cmp.	SRA	Type	Stages (s)					Total	
			1	2	3	4	5+6	Time	GCUPS
10K	1M	Local	∼	∼	∼	0.2	∼	1.3	0.1
		Overlap	∼	∼	∼	0.2	∼	0.6	0.2
		Semig	∼	∼	∼	0.2	∼	1.6	0.1
		Global	∼	∼	∼	0.2	∼	1.5	0.1
50K	3M	Local	0.2	∼	∼	∼	∼	0.4	7.1
		Overlao	0.2	∼	∼	∼	∼	1.2	2.7
		Semig	0.2	∼	∼	0.5	∼	2.0	1.6
		Global	0.2	0.1	∼	0.7	∼	2.3	1.4
150K	5M	Local	1.2	∼	∼	∼	∼	2.0	14.2
		Overlap	1.2	∼	∼	∼	∼	2.6	10.6
		Semig	1.2	0.4	0.5	2.2	∼	5.8	4.8
		Global	1.2	0.4	0.6	1.6	∼	5.4	5.2
500K	50M	Local	10.3	∼	∼	∼	∼	10.7	27.2
		Overlap	10.0	∼	∼	∼	∼	11.3	25.8
		Semig	10.0	1.3	0.9	4.6	∼	18.0	16.2
		Global	9.9	1.4	1.0	5.6	0.1	19.1	15.3
1M	250M	Local	37.5	1.2	0.9	4.7	0.1	45.2	24.8
		Overlap	36.5	0.8	0.6	3.3	0.1	43.0	26.1
		Semig	36.4	2.2	1.8	10.2	0.2	51.7	21.7
		Global	36.2	2.5	2.0	10.5	0.3	52.9	21.2
3M	1G	Local	332	∼	∼	0.3	∼	333	31.0
		Overlap	321	∼	∼	∼	∼	322	32.0
		Semig	321	11.2	6.0	37.7	1.1	378	27.3
		Global	320	11.1	6.0	37.7	0.9	377	27.4
5M	3G	Local	871	17.5	9.6	51.3	1.6	952	28.7
		Overlap	846	17.1	9.6	51.3	4.0	928	29.4
		Semig	846	17.2	9.7	51.4	2.8	928	29.5
		Global	844	16.9	9.6	51.5	1.7	925	29.6
7M	3G	Local	1189	0.2	∼	∼	∼	1191	31.4
		Overlap	1152	0.2	∼	∼	∼	1154	32.4
		Semig	1152	19.6	9.2	45.2	4.0	1232	30.3
		Global	1150	22.0	10.9	72.7	2.1	1259	29.7
10M	4G	Local	3317	113.5	18.7	103.0	3.4	3557	29.5
		Overlap	3210	113.8	18.8	105.8	11.6	3463	30.3
		Semig	3211	114.9	18.9	102.1	5.4	3454	30.3
		Global	3208	114.8	18.9	102.2	3.4	3449	30.4
23M	10G	Local	17793	0.2	∼	0.3	∼	17796	31.7
		Overlap	17259	0.2	∼	∼	∼	17263	32.7
		Semig	17260	392.7	42.8	203.4	6.6	17908	31.5
		Global	17253	407.7	44.7	245.6	8.6	17962	31.4
47M	50G	Local	48498	445.1	61.2	339.5	10.8	49357	31.2
		Overlap	47043	440.3	61.2	339.8	10.9	47898	32.1
		Semig	47042	477.7	62.9	371.2	13.5	47970	32.1
		Global	47026	475.5	62.6	373.4	10.9	47950	32.1



(a) Local (b) Overlap (c) Semiglobal (d) Global

Fig. 13. 1M optimal alignments.



(a) 10M Comparison - *A. mediterranei* (b) human x chimpanzee chromosome 21 comparison

Fig. 14. Optimal local alignment of some comparisons. Green area means low-score region, red area means high-score region and black area means pruned blocks.

Sequence 1. Nevertheless, in comparisons 5M and 10M, all alignment types produce almost the same results, producing similar traceback time.

Figure 13 presents the shapes of the local, overlap, semiglobal, and global optimal alignments of the 1M comparison. In this figure, the different types of alignment reside in distinct edges of the matrix. The local alignment (Figure 13(a)) starts in the middle of the matrix and ends in the left edge. The overlap alignment (Figure 13(b)) starts in the left edge and ends in the bottom edge. The semiglobal alignment (Figure 13(c)) starts in the left edge and ends in the right edge, representing all the characters of the second sequence. The global alignment (Figure 13(d)) begins and ends in the upper-left and bottom-right corners, larger than the other alignment types, although with a lower optimal score. These differences in the edges of the alignment produce different scores (Table III), that is, 88353 (local), 62525 (overlap), −588728 (semiglobal) and −1189459 (global).

7.5. Alignment Results

7.5.1. Amycolaptosis Mediterranei. In this section, we show the results obtained with the comparison of the strains S699 (NC_017186.1) and U32 (NC_014318.1) of *Amycotopsis mediterranei*. *A. mediterranei* is a Gram-positive actinomycete that produces an important antibiotic (ricamycin) and is extensively studied in the literature [Verma et al. 2011; Zhao et al. 2010].

Figure 14(a) shows the optimal local alignment between these two sequences. In this figure, the green and orange parts represent the cells of the matrix that were calculated; the black part represents the pruned cells. As can be seen, it is almost a

```
Query:         1 GAGCTCTATTGTTCCATTTTCATTGGCTTCTTCAGGTGCTCAGAAAAAGAAAAGACCCTG       60
                 ||||| |||||||||||||||||||||||||||||||||||| ||| ||||| |||||| [44/44]
Sbjct: 13841681 GAGCTGTATTGTTCCATTTTCATTGGCTTCTTCAGGTGCTCAGTAAAGGAAAATACCCTG 13841740

Query:        61 TTTTTCATGGAGAAAACATTGGGCAAATTGATTCAATCAAGAACTTTCTTAGTATATAAG      120
                 |||||||||||||||||||||||||||||||||||||||||||| ||||||||||||||| [56/100]
Sbjct: 13841741 TTTTTCATGGAGAAAACATTGGGCAAATTGATTCAATCAAGAAATTTCTTAGTATATAAG 13841800

Query:       121 ATGAAAGTGGTGGAACACATGAACTCTGAACCTGTTTCTTACCTAAATCCATTCTTTTGT      180
                 || |||||||| ||||||||||||||||||||||||||||||||||||||||||| | [48/148]
Sbjct: 13841801 ATTAAAGTGGTAGAACACATGAACTCTGAACCTGTTTCTTACCTAAATCCATTCTTTTTT 13841860

Query:       181 GGTTCATTTACTTTATATAAATTTAAATGCCTGCTTTTCCCCAGGCACTGTGCTATGCAC      240
                  ||||| ||||||||||||||||||||||||||||||||||||||||||||||||||| [52/200]
Sbjct: 13841861 AGTTCACTTACTTTATATAAATTTAAATGCCTGCTTTTCCCCAGGCACTGTGCTATGCAC 13841920

Query:       241 TGGTCATGAGGCAGTTAGGTTTATAACTTGTCTCCCTAAAAATTATTTGCTAATATTAAT      300
                 |||| ||||||||||||||||||||||||||||||||||||||||||||||||||||| [56/256]
Sbjct: 13841921 TGGTGATGAGGCAGTTAGGTTTATAACTTGTCTCCCTAAAAATTATTTGCTAATATTAAT 13841980
```

Fig. 15. Beginning of the textual file that contains the human x chimpanzee chromosome 21 alignment.

perfect match. The percentages of matches, mismatches and gaps are 99.996%, 0.003%, and 0.001%, respectively. This is consistent with the result reported in Verma et al. [2011]. The optimal score is 10,235,188.

7.5.2. Human x Chimpanzee Chromosomes 21. In this section, we discuss results obtained with MASA-CUDAlign for the human x chimpanzee chromosome 21 comparison. The analysis of these two chromosomes is a very active research area; new genes and transcripts are being discovered at a high rate [Scarpato et al. 2014]. Very recently, a new genetic analysis of human chromosome 21 [Letourneau et al. 2014] further advanced the knowledge needed to fully understand Down's syndrome.

Figure 14(b) presents the plot of optimal local alignment between these two whole chromosomes and Figure 15 shows a small part of the text file that contains the alignment. In Figure 14(b), the human chromosome 21 (NC_000021.7) is in the x axis and the chimpanzee chromosome 21 (BA000046.3) is in the y axis. The optimal alignment is shown as a blue line, starting in position 13,841,681 (NC_000021.7) and 1 (BA000046.3). The percentages of matches, mismatches, and gaps are 94.380%, 1.537%, and 4.082%, respectively.

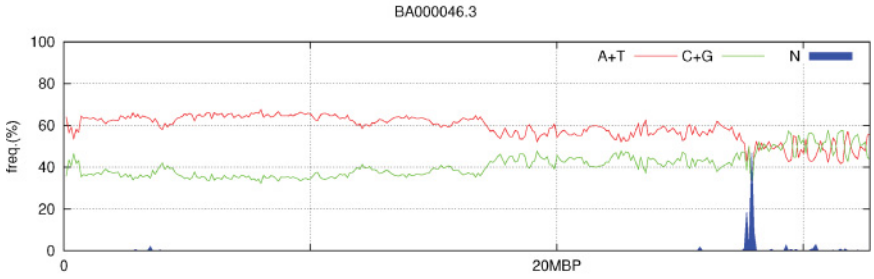
In order to further analyze the optimal alignment, we calculated the percentages of A-T and C-G for both sequences. The results are shown in Figure 16. Note that the human chromosome 21 has a high concentration of NNs in the beginning of the sequence. This is probably the reason why the optimal local alignment is shifted.

We also determined regions in the alignment that present a high rate of mismatches and gaps. These regions were detected by sliding a window (size = 50000) throughout the alignment and marking the areas with less than 70% matches. In each of these areas, the regions with the lowest score are selected as an unmatched region and the regions that overlap are joined together (Figure 17). The sizes of the regions and the location inside the optimal alignment are shown in Table VI. As can be seen, we were able to identify 10 regions of interest; detailed biological analysis of these regions is highly recommended.

8. CONCLUSION AND FUTURE WORK

This article presented Multiplatform Architecture for Sequence Aligners (MASA), a software architecture for the implementation of aligners in different hardware/software platforms. In MASA, the platform-independent code was decoupled from the CUDAlign 2.1 code [Sandes and Melo 2013a] and was successfully reused in 4 implementations. The MASA-CUDAlign implementation presented results comparable with CUDAlign 2.1, indicating that the MASA architecture did not insert considerable overhead. The

(a) Human chromosome 21 ATGCN frequency



(b) Chimpanzee chromosome 21 ATGCN frequency

Fig. 16. ATGCN frequency of chromosomes 21 (human and chimpanzee).

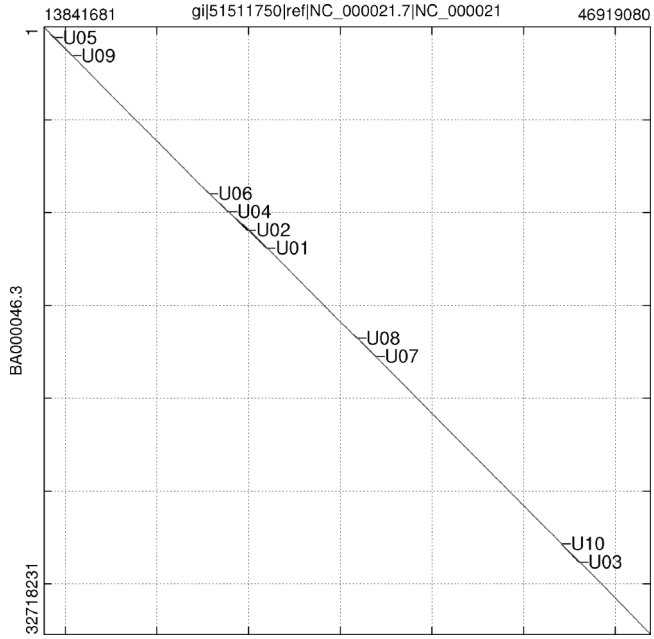


Fig. 17. Largest unmatched regions in the human x chimpanzee chromosome 21 optimal alignment.

Table VI. Sizes and Location of the Largest Unmatched Regions

Name	Size	Loc. S_0	Loc. S_1
U01	59072	10630194-10631376	24654791-24713863
U02	56421	10280845-10288485	24240387-24296799
U03	55857	27853079-27908936	42082089-42129406
U04	46315	9515853-9518073	23422491-23468803
U05	39683	337151-376789	14172065-14193379
U06	26872	8777783-8777920	22640592-22667464
U07	25757	16668297-16668303	30834341-30860098
U08	24390	16569464-16569473	30703011-30727401
U09	24251	408670-408670	14224960-14249211
U10	18456	27632628-27651084	41875088-41879528

MASA-OmpSs/CPU implementation used a dataflow parallelization strategy able to compute blocks of the DP matrix in a generic order, respecting the data dependencies of the SW/NW algorithm.

In order to allow the execution of BP in the dataflow parallelization strategy, a Generic BP algorithm was proposed and implemented. Generic BP was used in MASA-OmpSs/CPU, MASA-OpenMP/CPU and MASA-OpenMP/Phi, with the difference being that the MASA-OmpSs/CPU processed blocks in the dataflow strategy and the others in the diagonal strategy. The experimental results showed that Generic BP was more efficient using the dataflow strategy, since the MASA-OmpSs/CPU increased the GCUPS up to 23.7% compared with MASA-OpenMP/CPU.

As future work, we intend to use vector instructions for the Intel Phi and CPU MASA implementations. We also aim to create a MASA network interconnecting heterogeneous implementations to process the same DP matrix. Considering that the processing nodes may be shared with other processes, we intend to propose a dynamic load balancing mechanism for this MASA network. Finally, we plan to analyze in detail the unmatched regions found in the human x chimpanzee chromosome 21 optimal local alignment and to implement the BP strategy for other recurrence relations.

ACKNOWLEDGMENTS

The authors would also like to thank the BSC Minotauro team for help in setting up the environment for the multicore and GPU executions. Finally, the authors would like to thank the NSF through XSEDE resources provided by the XSEDE Science Gateways program that provided access to the Intel Phi coprocessor.

REFERENCES

M. Aldinucci, M. Meneghin, and M. Torquati. 2010. Efficient Smith-Waterman on multi-core with fastflow. In *Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 195–199.

S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. 1990. Basic local alignment search tool. *Journal of Molecular Biology* 215, 3, 403–410.

S. Aluru, N. Futamura, and K. Mehrotra. 2003. Parallel biological sequence comparison using prefix computations. *Journal of Parallel and Distributed Computing* 63, 3, 264–272.

K. Benkrid, A. Akoglu, C. Ling, Y. Song, Y. Liu, and X. Tian. 2012. High performance biological pairwise sequence alignment: FPGA versus GPU versus CellBE versus GPP. *International Journal of Reconfigurable Computing* 2012, 15 pages, Article ID 752910.

C. Chen and B. Schmidt. 2003. Computing large-scale alignments on a multi-cluster. *IEEE Cluster Computing Conference,* 38–45.

A. Driga, P. Lu, J. Schaeffer, D. Szafron, K. Charter, and I. Parsons. 2006. FastLSA: A fast, linear-space, parallel and sequential algorithm for sequence alignment. *Algorithmica* 45, 3, 337–375.

A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. 2011. OmpSs: A proposal for programming heterogeneous multicore architectures. *Parallel Processing Letters* 21, 2, 173–193.

R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. 2002. *Biological Sequence Analysis*. Cambridge University Press, New York, NY.

M. Farrar. 2007. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics* 23, 2, 156–161.

O. Gotoh. 1982. An improved algorithm for matching biological sequences. *Journal of Molecular Biology* 162, 3, 705–708.

K. Hamidouche, J. Falcou, and D. Etiemble. 2010. Hybrid bulk synchronous parallelism library for clustered SMP architectures. In *4th International Workshop on High-Level Parallel Programming and Applications (HLPP'10)*. 55–62.

K. Hamidouche, F. M. Mendonca, J. Falcou, A. C. M. A. Melo, and D. Etiemble. 2013. Parallel Smith-Waterman comparison on multicore and manycore computing platforms with BSP++. *International Journal of Parallel Programming* 41, 1, 111–136.

D. S. Hirschberg. 1975. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM* 18, 6, 341–343. DOI:http://dx.doi.org/10.1145/360825.360861

M. Korpar and M. Sikic. 2013. SW#-GPU-enabled exact alignments on genome scale. *Bioinformatics* 29, 19, 2494–2495.

A. Letourneau, F. A. Santoni, X. Bonilla, M. R. Sailani, D. Gonzalez, J. Kind, and C. Chevalier. 2014. Domains of genome-wide gene expression dysregulation in Down syndrome. *Nature* 508, 345–350.

Y. Liu and B. Schmidt. 2014a. GSWABE: Faster GPU-accelerated sequence alignment with optimal alignment retrieval for short DNA sequences. *Concurrency and Computation: Practice and Experience* 27, 4, 958–972.

Y. Liu and B. Schmidt. 2014b. SWAPHI: Smith-Waterman protein database search on Xeon Phi coprocessors. In *25th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'14)*. 184–185.

Y. Liu, B. Schmidt, and D. L. Maskell. 2009. MSA-CUDA: Multiple sequence alignment on graphics processing units with CUDA. In *20th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'09)*. 121–128.

Y. Liu, T. Tam, F. Lauenroth, and B. Schmidt. 2014. SWAPHI-LS: Smith-Waterman algorithm on Xeon Phi coprocessors for long DNA sequences. In *IEEE International Conference on Cluster Computing—CLUSTER*. 257–265.

Y. Liu, A. Wirawan, and B. Schmidt. 2013. CUDASW++ 3.0: Accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics* 14, 117.

S. Maleki, M. Musuvathi, and T. Mytcowicz. 2014. Parallelizing dynamic programming through rank convergence. In *19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, New York, NY, 219–232.

S. Manavski and G. Valle. 2008. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics* 9, Suppl 2.

D. W. Mount. 2004. *Bioinformatics: Sequence and Genome Analysis*. CSHL Press, Long Island, NY.

E. W. Myers and W. Miller. 1988. Optimal alignments in linear space. *Computer Applications in the Biosciences* 4, 1, 11–17.

S. B. Needleman and C. D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3, 443–453.

Gregory F. Pfister. 1995. *In search of Clusters: the Coming Battle in Lowly Parallel Computing*. Prentice-Hall, Inc., Upper Saddle, NJ.

S. Rajko and S. Aluru. 2004. Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel Distributed Systems* 15, 12, 1070–1081.

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

F. Sanchez, F. Cabarcas, A. Ramirez, and M. Valero. 2010. Long DNA sequence comparison on multicore architectures. In *Euro-Par 2010—Parallel Processing*. 247–259.

E. F. de O. Sandes, G. Miranda, A. C. M. A. de Melo, X. Martorell, and E. Ayguadé. 2014a. Fine-grain parallel megabase sequence comparison with multiple heterogeneous GPUs. In *PPOPP'14*. 383–384.

E. F. de O. Sandes, G. Miranda, A. C. M. A. Melo, X. Martorell, and E. Ayguade. 2014b. CUDAlign 3.0: Parallel biological sequence comparison in large GPU clusters. In *IEEE/ACM Symposium on Cluster, Cloud and Grid Computing (CCGrid'14)*. 160–169.

E. F. O. Sandes and A. C. M. A. Melo. 2011. Smith-Waterman alignment of huge sequences with GPU in linear space. In *IEEE International Parallel and Distributed Processing Symposium*. 1199–1211.

E. F. O. Sandes and A. C. M. A. Melo. 2013a. Retrieving Smith-Waterman alignments with optimizations for megabase biological sequences using GPU. *IEEE Transactions on Parallel and Distributed Systems* 24, 5, 1009–1021. DOI:http://dx.doi.org/10.1109/TPDS.2012.194

E. F. O. Sandes and A. C. M. A. Melo. 2013b. Retrieving Smith-Waterman alignments with optimizations for megabase biological sequences using GPU. *IEEE Transactions on Parallel and Distributed Systems* 24, 5, 1009–1021. DOI:http://dx.doi.org/10.1109/TPDS.2012.194

A. Sarje and S. Aluru. 2009. Parallel genomic alignments on the cell broadband engine. *IEEE Transactions on Parallel and Distributed Systems* 20, 11, 1600–1610.

S. Sarkar, G. R. Kulkarni, P. P. Pande, and A. Kalyanaraman. 2010. Network-on-chip hardware accelerators for biological sequence alignment. *IEEE Transactions on Computers* 59, 1, 29–41.

M. Scarpato, R. Esposito, D. Evangelista, M. Aprile, M. R. Ambrosio, A. Ciccodicola C. Angelini, and V. Costa. 2014. Analysis of expression on human chromosome 21, ALE-HSA21: A pilot integrated web resource. *Database—Journal of Biological Databases and Curation* 2014, Article ID bau009.

T. F. Smith and M. S. Waterman. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* 147, 1, 195–197.

D. Maskell, T. F. Oliver, and B. Schmidt. 2005. Hyper customized processors for bio-sequence database scanning on FPGAs. In *ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*. 229–237.

Xinmin Tian, H. Saito, S. V. Preis, E. N. Garcia, S. S. Kozhukhov, M. Masten, A. G. Cherkasov, and N. Panchenko. 2013. Practical SIMD vectorization techniques for Intel Xeon Phi coprocessors. In *2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW'13)*. 1149–1158. DOI:http://dx.doi.org/10.1109/IPDPSW.2013.245

M. Verma and others. 2011. Whole genome sequence of the rifamycin b-producing strain *Amycolatopsis mediterranei*. *Journal of Bacteriology* 193, 19, 5562–5563.

A. Wirawan, B. Schmidt, H. Zhang, and C. K. Kwoh. 2009. High performance protein sequence database scanning on the cell broadband engine. *Scientific Programming* 17, 97–111.

S. Xiao, A. M. Aji, and W. Feng. 2009. On the robust mapping of dynamic programming onto a graphics processing unit. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS'09)*.

Y. Yamagutchi, H. K. Tsoi, and W. Luk. 2011. FPGA-Based Smith-Waterman algorithm: Analysis and novel design. In *International Conference on Applied Reconfigurable Computing (ARC'11)*. 181–192.

W. Zhao and others. 2010. Complete genome sequence of the rifamycin SV-producing *Amycolatopsis mediterranei* U32 revealed its genetic characteristics in phylogeny and metabolism. *Cell Research* 10, 1096–1108.

Received November 2014; revised May 2015; accepted August 2015