Parallel Computing 37 (2011) 244–259

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Parallelizing and optimizing a bioinformatics pairwise sequence alignment algorithm for many-core architecture

David Díaz [a], Francisco José Esteban [b], Pilar Hernández [c], Juan Antonio Caballero [d], Gabriel Dorado [e,1], Sergio Gálvez [a,*,1]

[a] Dep. Lenguajes y Ciencias de la Computación, ETSI Informática, Campus de Teatinos, Universidad de Málaga, Bulevar Louis Pasteur 35, 29071 Málaga, Spain
[b] Servicio de Informática, Edificio Ramón y Cajal, Campus Rabanales, Universidad de Córdoba, 14071 Córdoba, Spain
[c] Instituto de Agricultura Sostenible (IAS-CSIC), Alameda del Obispo s/n, 14080 Córdoba, Spain
[d] Dep. Estadística, Campus Rabanales C2-20N, Universidad de Córdoba, 14071 Córdoba, Spain
[e] Dep. Bioquímica y Biología Molecular, Campus Rabanales C6-1-E17, Universidad de Córdoba, 14071 Córdoba, Spain

A R T I C L E   I N F O

Article history:
Received 3 May 2010
Received in revised form 11 January 2011
Accepted 18 March 2011
Available online 25 March 2011

Keywords:
Tile64 processor
Multi-core parallelization
System-on-chip
Chip multiprocessor architecture
Needleman–Wunsch
Smith–Waterman
High performance optimization
Multithreading

A B S T R A C T

Current computer engineering evolves at an accelerated pace, with hardware advancing towards new chip multiprocessors (CMP) architectures and with supporting software gearing towards new programming and abstraction paradigms, to obtain the maximum efficiency of the hardware at a low cost. In this context, Tilera Corporation has developed a brand new CMP architecture with 64 cores (tiles) called Tile64, and has launched several Peripheral Component Interconnect Express (PCIe) cards to be used and monitored from a host Personal Computer (PC). These cards may execute parallel applications built in C/C++ and compiled with the Tile-GCC compiler. We have previously demonstrated the usefulness of the Tile64 architecture for bioinformatics [S. Gálvez, D. Díaz, P. Hernández, F.J. Esteban, J.A. Caballero, G. Dorado, Next-generation bioinformatics: using many-core processor architecture to develop a web service for sequence alignment, Bioinformatics, 26 (2010) 683–686]. We have chosen a bioinformatics algorithm to test this many-core Tile64 architecture because of actual bioinformatics challenging needs: data-intensive workloads, space and time-consuming requirements and massive calculation. This algorithm, known as Needleman–Wunsch/Smith–Waterman (NW/SW), obtains an optimal sequence alignment in quadratic time and space cost, yet requires to be optimized to take full advantage of computing parallelization. In this paper we redesign, implement and fine-tune this algorithm, introducing key optimizations and changes that take advantage of specific Tile64 characteristics: RISC architecture, local tile's cache, length of memory word, shared memory usage, RAM file system, tile's intercommunication and job selection from a pool. The resulting algorithm – named MC64-NW/SW for Multicore64 Needleman–Wunsch/Smith–Waterman – achieves a gain of $\sim$1000% when compared with the same algorithm on a $\times$86 multi-core architecture. As far as we know, our NW/SW implementation is the fastest ever published for a standalone PC when aligning a pair of sequences larger than 20 kb.

* Corresponding author. Tel.: +34 952133312; fax: +34 952131397.
E-mail addresses: david.diaz@lcc.uma.es (D. Díaz), fjesteban@uco.es (F.J. Esteban), phernandez@ias.csic.es (P. Hernández), ma1camoj@uco.es (J.A. Caballero), bb1dopeg@uco.es (G. Dorado), galvez@lcc.uma.es, galvez@uma.es (S. Gálvez).
[1] Authors who contributed to the project leadership.

0167-8191/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2011.03.003

1. Introduction

Evolution of computer engineering is leading towards a new era of high-performance processing, in which new chip multiprocessors (CMP) are the cornerstone. Some companies and manufacturers have focused their efforts on the development of new and scalable architectures which will gradually replace previous approaches. Some of the new technologies are already in use in areas like the Graphics Processing Units (GPU) with hundreds of cores [1] or the Sony–IBM–Toshiba Cell Broadband Engine (CBE) [2], exploited in video games [3], and supercomputing blade systems like the IBM BladeCenter server platform [4]. All of them require detailed programming methodologies and architecture optimizations far from the ones of the Central Processing Units (CPU) [5,6]. Other companies are working in real CMP and multithreaded many-core processors, like the Intel Single-Chip Cloud Computer (SCC) with 48 cores and the Terascale Processor with 80 cores [7], the Sun Microsystems UltraSPARC T2 Pro with eight cores and eight threads per core [8] and the Tilera Tile64 processor http://www.tilera.com. The products of the latter company are available as System-on-Chip (SoC) many-core Peripheral Component Interconnect Express (PCIe) cards, where each of the 64 cores (tiles) integrated into a single Tile64 processor die is capable of running a full operating system, managing several independent threads [9]. Each card includes the many-core processor, RAM memory and communication ports. Besides, being provided as integrated PCI-Express boards, any parallel job can be executed independently of the host Personal Computer (PC).

We have previously demonstrated the usefulness of the Tile64 architecture for bioinformatics [10]. We have developed bioinformatics algorithms with TilExpress-20G cards, which include the many-core Tile64 processor (64 cores at 866 MHz), 8 GB of RAM and two 10 Gb/s communications ports. These cards focus on networking applications and streaming broadcasts, thanks to their very high communication bandwidth and scalability, to create a cluster of connected cards. The Tile64 has been also evaluated in other research fields as video encoding [11]. However, our research focuses on testing and exploring the suitability of Tile64 in a very different area: bioinformatics. We follow a similar methodology of other works, that have used intensive processing bioinformatics algorithms on platforms like GPU [12] or the mentioned CBE [13], obtaining better performance than its single CPU implementation, as a result of their parallelization factor.

As technology developments are allowing sequencing full genomes at an affordable cost, bioinformatics is gaining momentum, moving towards a data- and computational-massive science, which requires huge memory quantities and computational power. For this reason, actual life science problems require new and adapted algorithms to deal with full genomes. In this context, one of the most important tasks in bioinformatics is the alignments of nucleic acid (DNA or RNA) and peptide (e.g., protein) sequences, which can take several hours and even days of processing in standard PC, in order to optimally align a pair of sequences of many thousands of bases. Furthermore, the used algorithms require huge amounts of memory to run optimally.

To test the performance of Tile64 in the field of bioinformatics, we have developed parallel versions of widely used pairwise alignment algorithms: Needleman–Wunsch (NW) [14] and Smith–Waterman (SW) [15]. Our code has been written from scratch, based on the Fast Linear-Space Alignment (FastLSA) approach [16], introducing many variations and optimizations to exploit the Tile64 parallelism and to improve the performance. The project has been coded, tested and deployed with the Tilera's Multicore Development Environment (MDE) 1.3.5, and can be found as an open-source project at http://www.sicuma.uma.es/manycore. Furthermore, in this site, the algorithm can be executed remotely through a web service on our server with a dedicated Tile64 card. Hence, any life scientist can test the quality of the resulting alignments and the application performance [10].

Once developed this brand new algorithm, called MultiCore64-NW/SW, we have run an extensive benchmarking test to compare the results with other implementations of the same algorithms that can be run inside a standalone PC, Even GPU-based algorithms have been considered, but they are not comparable. Finally, we discuss the potential of the Tile64 many-core processor for bioinformatics.

2. Software and hardware architecture

Before starting with the algorithm, it is important to take a look at the Tile64 processor architecture and the software development environment that Tilera provides. As with any other specific technology, the code must be implemented following the manufacturer's guidelines and using the resources that the company recommends, as the algorithm will be compiled for and run in the TilExpress card.

2.1. Tile64 many-core processor

The Tile64 is a 90 nm RISC-based processor which contains 64 general-purpose processor cores, communicating over a so-called iMesh architecture, with a bandwidth of 31 Tbps. Each tile is capable of running a customized Linux in it, with a clock frequency of 500–866 MHz and efficient power usage (15–22 W at 700 MHz, with all cores active), reaching a maximum global of 0.166 Tera-Instructions Per Second (TIPS) with 32 bits operations, thanks to its 3-way Very Long Instruction Word (VLIW) pipeline for instruction level parallelism. Each tile contains 8 KB of L1 instruction and data caches accessed by a Translation Lookaside Buffer (TLB) and 64 KB of L2 cache. Additionally, the L2 caches of all tiles are grouped as a 4 L3 common cache [17]. Dedicated and shared tiles may be needed to manage some of these capabilities, reducing the total amount of effective available tiles to run applications. Our development has been carried out with a TILExpress-20G on-boarded with 8 GB of RAM and 40 Gbps of I/O bandwidth through its two XAUI ports, two 4-lane PCIe interfaces and two GbE MAC interfaces. The provided RAM can be used as a Solid State Disk (SSD), as well as main memory, with empirically tested limits: 7.8 GB for SSD, 2.8 GB for local memory and 1.9 GB for shared memory.

Aside from this, the power consumption is maintained at low levels: between 10.8 W with core power at 1 V and 750 MHz (85 °C) [17] and 35 W as reported by Intel [7] (or 15–22 W as published by Tilera in its home page http://www.tilera.com). However, the Tile64 processor is described as a chip with 0.192 TFLOPS [7], which is astonishing because such processor lacks floating point calculus (the 0.192 value is a peak when using 16 bits operations at 866 MHz). This handicap is not a problem for bioinformatics algorithms where mainly integer values are used, yet prevents the Tile64 to be exploited in some supercomputation areas.

2.2. Development environment

Developments for the Tile64 processor can be carried out with the Eclipse-based Tilera's Multicore Development Environment (TileMDE), which wraps a command-line set of tools. This Integrated Development Environment (IDE) provides a C/C++ cross-compiler, as well as a visual environment to deploy applications into the tiles of the card. A hardware simulator allows to program, compile and run without a physical card installed into the developing computer, simulating the behavior of each tile. However, the simulator is much slower than the physical card, and thus the performance when testing real applications is very low. When executing a program either in the simulator or in the physical TilExpress board, the IDE visually shows a real-time dynamic image of its tiles and the process that they are executing, as can be seen at http://www.sicuma.uma.es/manycore/index.jsp?seccion=m64nw/simulation. Basically, the method to feed the card with a parallel workload is as follows:

(i) Start the TilExpress board and load a vmlinux into each tile. The I/O dedicated tiles and other hardware configurations are set as well in this startup step.
(ii) Load the host files into the card file system. This includes uploading the job-required files, as well as executable files, cross compiled with the Tile-GCC compiler.
(iii) Launch an executable into a tile. Usually, this executable (the ''controller program'') will be in charge of scheduling the execution of code into other tiles, coordinating the results and providing new jobs. Another possibility is to replicate the application itself to be launched in as many cores as needed, running in parallel from this point on.
(iv) When the whole job is finished, the result files are downloaded to the host and used as required.

This approach is highly customizable and allows the programmer to control the process management and memory usage:

- The card can access directly the host file system.
- Many TilExpress boards can be physically linked and the workload can be distributed among them.
- Several tiles can be grouped by functional affinity.
- Coordination and communication can be established by shared memory, message passing or other strategies.

All these possibilities must be considered when programming using the Tile64 architecture‘s specific resources and C libraries. In particular, the main library for parallel execution and communication between processes (iLib) defines an Application Programming Interface (API) for these tasks. This library makes easier to port an existing parallel application to the iLib, in order to exploit the parallelism of the Tile64 processor.

3. Methodology

As we have stated above, the algorithms to align (i.e., compare) sequences are the cornerstone when comparing DNA, RNA, and peptide sequences, used to identify similar and different regions inside them. In order to port the basic global aligners Needleman–Wunsch and Smith–Waterman to the Tile64 architecture, different methodologies can be used, depending on the requirements for efficiency, time and effort required to program the new algorithms.

For comparison purposes, our first implementation is a direct translation, with minor changes, from the non affine-gap Needleman–Wunsch ANSI C source code to a native Tile64 C source: this creates an executable for the TilExpress card, which runs the Needleman-Wunsch algorithm into a single tile, as the Tile-GCC compiler does not automatically parallelize standard sequential programs. As a tile clock frequency and resources are below the ones of a $\times$86 CPU, this option should be discarded, due to efficiency leaks. Notwithstanding, this scenario does reveal some interesting facts (Fig. 1) when benchmarking a standard NW execution on a $\times$86 CPU vs a standard NW execution on Tile64 single tile vs a EMBOSS' needle tool execution on the $\times$86 CPU. For example, we can observe the EMBOSS' lower performance against the NW standard algorithm, because it uses affine gap penalties [18]. The main conclusion of this analysis is that the NW algorithm in a single tile of the Tile64 processor is very inefficient, being severely limited by local memory resources. Furthermore, the Tile64 single

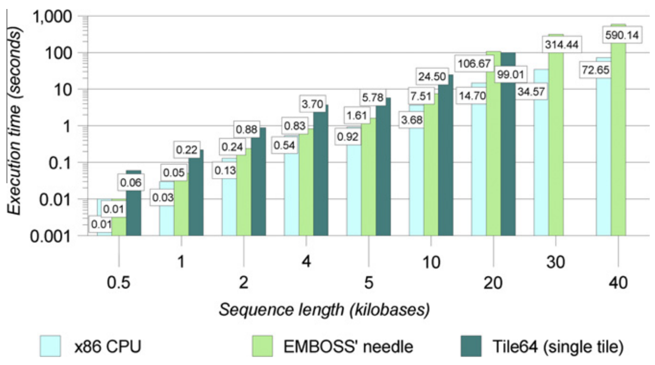

Fig. 1. Needleman–Wunsch benchmarking. The Needleman–Wunsch algorithm execution time comparison for execution in a single core of a ×86 CPU (Intel Xeon Quad Core 2.0 GHz PC with 8 GB of DDR2 memory in quad-channel) vs EMBOSS vs execution in a single tile of the Tile64 processor (Tilera TilExpress-20G card) is shown. Only EMBOSS uses the affine gap approach, but as Tile64 tile local memory is limited to 2.8 GB, the 30 kb and 40 kb executions could not be completed.

tile execution is slower in most cases than the EMBOSS' needle one, even though the latter produces a more accurate result. Therefore, the direct porting (the least effort approach) should be discarded.

Though the next step could be to use a compiler with abilities to automatically generate parallel code, the Tile-GCC compiler in MDE 1.3.5 does not support such feature. Computer parallelization is one of the most complex and researched fields in Computer Sciences nowadays, because of emerging technologies requiring high computer performance (like next-generation sequencing platforms) [10], being also one of the compilers' classic challenges [19]. However, in our case, it is a programmer's task to manually parallelize the execution and distribute it among the tiles.

For manual parallelization, the less-effort approach is to look for the bottleneck(s) of the algorithm and parallelize such part(s) of the code to distribute the workload among the available tiles. This methodology is explained in [20] and used in [13], where the *gprof* tool helps to determine the most time-consuming functions in the test applications. With little additional coding, an important speedup may be achieved, because a few most time-consuming functions usually take more than the 60% of execution time. In NW, obtaining the full Dynamic Programming Matrix (DPM) alignment is the most time- and memory-consuming task, which is a whole stage of the algorithm instead of a single function. As our approach tries to achieve the best performance possible in the Tile64 processor, the workload to be distributed must be the DPM calculus.

However, in order to exploit the real power of the Tile64 architecture, we will not develop a simple abstract parallelization of the DPM calculus, but the complete algorithm, taking into consideration the hardware architecture characteristics, to optimize access time and overall usage of memory and network, both for the forward and backward stages.

4. Sequence alignment

An efficient implementation of Needleman–Wunsch for the Tile64 architecture requires an intensive study of the algorithm. Furthermore, there are many improvements for the original algorithm which can be applied as well to the Smith–Waterman algorithm, which needs minor changes regarding NW to obtain a local instead of global alignment. We have started with a parallel implementation of one of these algorithms (FastLSA), which has been progressively improved and adapted to the Tile64 iLib library and hardware characteristics to optimize performance. With each innovation, new versions are compared against previous ones to determine the overall gain.

4.1. Overview

Sequence alignments are the most used operations in bioinformatics, as they may reveal useful information about the relationship between the compared sequences. An alignment of two sequences (where a sequence from a programmer's point of view is a string of letters from a reduced alphabet) is a matching between them without any permutation on their elements (checking possible base changes), taking also into consideration inserted and deleted zones, represented through the insertion of gaps in any sequence. Such insertions and deletions are also known as "indels". These are named pairwise alignments and can be classified into two subclasses, depending on the alignment length: global alignments if the sequences are aligned from beginning through end, and local alignments if only the slice where the two sequences present their maximum similarity is aligned. The number of possible alignments is exponential and the interest is to focus on the "best alignment". An alignment shows the zones with similarities and differences, which a life scientist can use for downstream applications. This way, phylogenetic trees (dendrograms) can be generated to show the evolutionary history of the sequences. Likewise, the polymorphisms among the sequences can be identified and appropriate molecular markers can be designed. Such is the case of the Single Nucleotide Polymorphism (SNP) that we have exploited for quality control and fraud prevention of olive oil [21].

There are several methods to calculate both local and global pairwise alignments, but the first and most extended one is the global aligner Needleman–Wunsch algorithm [14], which receives its name from their creators back in 1970. Starting from it, the local alignment concept was introduced by Smith and Waterman in 1981 [15]. Subsequently, many improvements and optimizations over the original global aligner were made, like Gotoh's [22], Hirschberg's [23] and FastLSA [16]. Thus, to align the A sequence of length *n* (query sequence) with the B sequence of length *m* (subject sequence), the Needleman–Wunsch algorithm uses dynamic programming to maximize scores in a *nxm* matrix – the previously defined DPM – with all the possible base changes and indels. The position [*i*,*j*] of this DPM stores the best score to align the first *i* elements of A with the first *j* of B, considering a reference table (called substitution or scoring matrix) and a gap insertion and deletion penalty cost. These penalty costs and the matrix quantify the concept of similarity. The substitution matrix determines the scoring for any two bases to align.

There are many standard matrices experimentally obtained, like the Point Accepted Mutation (PAM) or BLOck SUbstitution Matrix (BLOSUM) families, each one being used for specific applications. The first row and column of the DPM are initialized and each other cell is determined from its three neighbors: upper, left and upper-left. The score to store in the cell [*i*,*j*] is the maximum of: (i) inserting a gap in the sequence A, so that B[*j*] matches with a gap {upper cell's score [$i,j-1$] plus the insert penalty}; (ii) inserting a gap in the sequence B so A[*i*] matches with a gap {left cell's score [$i-1,j$] plus the delete penalty}; and (iii) matching the two bases A[*i*] and B[*j*] {upper-left cell's score [$i-1,j-1$] plus the match value given by the scoring matrix}.

When all the cells have been calculated through this process, the alignment is obtained in a backward stage, from the bottom right corner cell that contains the final score in a global alignment (which may be or not the maximum total value). In this stage, we track the path described in the first stage; i.e., moving from position [*i*,*j*] to the previous cell for which a maximum score was obtained from {[$i-1,j$], [$i,j-1$] or [$i-1,j-1$]}, as described before. Thus, coming from the upper position [$i,j-1$], it will insert a gap in the aligned sequence A and the base in the aligned sequence B; coming from the left position [$i-1,j$], it will insert the base in the aligned sequence A and a gap in the aligned sequence B; and coming from the upper left position [$i-1,j-1$], it will insert the correspondent base in both aligned sequences. This process continues until the upper right position in the DPM is reached. The algorithm is shown below (Algorithm 1).

Algorithm 1: Needleman–Wunsch pairwise global aligner algorithm.

```
Input
  $S_1$: query sequence of length $n$ to be aligned;    $S_2$: subject sequence of length $m$ to be aligned
  $g_i$: gap insertion penalty cost; $g_d$: gap deletion penalty cost
  $W$: substitution matrix, were $W[a,b]$ gives the score of replacing the symbol $a$ by $b$
  $F$: $n \times m$ internal alignment matrix to save scores
Output
  $A_1$: aligned query sequence; $A_2$: aligned subject sequence
Begin
$F_{0,0} = 0$; // Initialization
$F_{i,0} = F_{i-1,0} - g_d$;
$F_{0,j} = F_{0,j-1} - g_i$;
for (i = 1; i <= n; i ++) // Forward stage
    for (j = 1; j <= m; j ++)
        $F_{i,j} = \max(F_{i-1,j} - g_d, F_{i,j-1} - g_i, F_{i-1,j-1} + W[S_{1i}, S_{2j}])$
while (i > 0 || j > 0) // Backward stage
    if (($F_{i,j} == F_{i,j-1} - g_i$) || (i == 0)) // Coming from upper: insert
        gap # $A_1$; $S_{2j}$ # $A_2$; j –;
    else if (($F_{i,j} == F_{i-1,j} - g_d$) || (j == 0)) // Coming from left: delete
        $S_{1i}$# $A_1$; gap # $A_2$; i --;
    else // Coming from upper-left: replace ($F_{i,j} == F_{i-1,j-1} + W[S_{1i}, S_{2j}]$)
        $S_{1i}$# $A_1$; $S_{2j}$# $A_2$; i --; j --;

Where # represents the String concatenation operation

End
```

As may be seen, this algorithm uses dynamic programming to generate the calculation matrix (DPM), and both the time and space costs are quadratic. So, for very long sequences (>100,000 bases), this is on the edge of the current personal computer memory capabilities.

4.2. Improvements

The original Needleman–Wunsch algorithm [14] represented an unprecedented hit for bioinformatics, because it allowed life scientists to find similarities and differences between two sequences computationally, instead of manually as previously done. However, the original algorithm lacked precision, because long gaps caused by deletions (or insertions on the other strand) were severely penalized, yet they may arise from mutation events and are therefore found in nature. Sellers [24] released a study demonstrating that the evolutionary distances satisfied specific metric conditions, which were subsequently used by Waterman et al. in 1976 [25]. In 1981, Smith and Waterman extended this idea to give birth to the concept of the local alignment [15]: "a pair of segments, one from each of two long sequences, such that there is no other pair of segments with greater similarity (homology)".

These improvements introduced new changes in the original algorithm: for each position, an open gap penalty must be recomputed from the already calculated DPM part. The initialization does not use negative values (only zero), and the local alignment saves the highest score achieved during the first phase calculation. Besides, the backward phase starts from this highest score point until it reaches the first zero value cell, which will determine the beginning of the segment. This lead the Smith–Waterman algorithm to a higher time-complexity of $n^2 \times m$. Later, these changes were optimized in $n \times m$ time complexity by Gotoh [22], using two additional matrices instead of recalculating for each cell the gap penalization until that point. Thus, we will use Gotoh's adjustments with three matrices for an optimal alignment in both Needleman–Wunsch and Smith–Waterman aligners.

Furthermore, our objective is to allow NW to find the so-called semiglobal or *glocal* alignment [26], which is much more useful to life scientists. To achieve that, the initialization of the first row and column in NW must be changed, in order to match the sub-sequences if one is part of the other, and not necessarily both begin and/or end simultaneously. Thus, including zeros in the first row and column, the alignment can start with a long gap without any penalty cost in NW. For the same reasons, the backward stage's starting point in NW is the maximum value among all the values in the last row and column of the DPM. A brief description about the algorithm's new behavior is shown below (Algorithm 2).

Algorithm 2: Needleman–Wunsch pairwise semiglobal aligner algorithm with Gotoh's affine gap.

```
Input
  $S_1$: query sequence of length $n$ to be aligned; $S_2$: subject sequence of length $m$ to be aligned
  $g_i$: gap insertion open penalty cost; $g_d$: gap deletion open penalty cost; $g_e$: gap extension penalty cost
  $W$: substitution matrix, were $W[a,b]$ gives the score of replacing the symbol $a$ by $b$
  $F$: $n \times m$ internal alignment matrix to save scores
  D, I: Auxiliary $n \times m$ matrices to save scores for affine gap
Output
  $A_1$: aligned query sequence; $A_2$: aligned subject sequence
Begin
$F_{i,0} = F_{0,j} = 0$; // Initialization
$D_{i,0} = -\infty$;
$I_{0,j} = -\infty$;
for (i = 1; i <= n; i ++) // Forward stage
    for (j = 1; j <= m; j ++) {
        $D_{i,j} = \max(D_{i-1,j}, F_{i-1,j} - g_d) - g_e$ // Keep actual open gap or open a new one
        $I_{i,j} = \max(I_{i,j-1}, F_{i,j-1} - g_i) - g_e$ // Keep actual open gap or open a new one
        $F_{i,j} = \max(D_{i,j}, I_{i,j}, F_{i-1,j-1} + W[S_{1i}, S_{2j}])$
    }
(i, j, $A_1$,$A_2$) = trailing_gaps ();
while (i > 0 & & j > 0) // Backward stage, semiglobal
    if ($F_{i,j}$ == $I_{i,j}$) // Coming from upper: insert
        gap # $A_1$; $S_{2j}$ # $A_2$; j --;
    else if ($F_{i,j}$ == $D_{i,j}$) // Coming from left: delete
        $S_{1i}$# $A_1$; gap # $A_2$; i --;
    else // Coming from upper-left: replace ($F_{i,j} == F_{i-1,j-1} + W[S_{1i}, S_{2j}]$)
        $S_{1i}$# $A_1$; $S_{2j}$# $A_2$; i --; j --;
leading_gaps () # ($A_1$, $A_2$);
Where:
  • # represents the String concatenation operation.
  • trailing_gaps () returns [i,j], where $F_{i,j}$ is the maximum value in the last row or column, and $A_1$, $A_2$ are two Strings containing the aligned sequences from $F_{i,j}$ up to $F_{n,m}$ (excluding the $F_{i,j}$ cell).
  • leading_gaps () returns two Strings containing the aligned sequences from $F_{0,0}$ up to $F_{i,j}$.
End
```

Due to the quadratic space complexity of this algorithm, Hirschberg [23] presented a "divide and conquer" implementation that overcame such handicap, obtaining the results in linear space and double quadratic time. Myers and Miller [27] proposed a new implementation, which merged the Hirschberg's idea of linear-space with the Gotoh's affine gap matrix, later extended by Huang et al. [28] for subsequence matching. On the other hand, the methods proposed by Aluru et al. [29] using prefix computations for a cluster of processors, and Martins' fine-grain multi-threaded execution model [30], try to fine-tune the parallelization factor in linear space sequence alignment with the rise of multi-core and many-core technologies.

To avoid the problems of memory and time requirements, other authors developed heuristics for non-optimal alignments, like the Basic Local Alignment Search Tool (BLAST) [31] and the Fast Alignment Sequence Tools (FAST)-All (FASTA) [32], mostly oriented to align very fast relatively short sequences against many sequences stored in a database. However, our interest here mainly focuses on long genome alignments, which requires a high degree of accuracy, so we are not interested in such non-optimal approaches. The rationale is that the comparison of very long sequences may be very useful for some researches, as we have described for food quality control and fraud prevention [21]. On the other hand, the large set of data can be analyzed in detail with plugins like the Ω-JalView, being integrated in intuitive tools like the Ω-Brigid workflow that we have developed for life scientists, not requiring knowledge of code programming [33].

In any case, optimal is a relative concept from a biological point of view, so aligning algorithms take into account the so-called optimal string alignment (pattern matching). To align long sequences, the best option is FastLSA [16], an improved version of Hirschberg [23], which is efficient in space cost but is not linear: it takes advantage of the available memory to lower time execution. As previously mentioned, the Hirschberg's algorithm uses a divide and conquer approach, in which the DPM must be calculated twice to achieve a linear space complexity. Thus, it executes approximately the double of operations than a standard Needleman-Wunsch approach would do. On the other hand, the FastLSA proposes to save some temporal values from first stage in cache: this grid will contain columns and rows from the complete DPM saved in the forward stage. Therefore, when the backward stage is executed, only a few sub-matrices must be recomputed (the ones in which the optimal path passes through). With this approach, the problem is reduced to calculate many sub-matrices; each one being obtained by a different processing unit and, thus, capable of parallelization.

Hence, the FastLSA can be implemented through a wavefront parallelism as seen in Fig. 2. The first stage (Fig. 2, left) computes the complete DPM, with several sub-processes in task of calculating sub-matrices progressively. Once both the initial row and column for a sub-matrix are ready (obtained from previous processes' completion or from the global initial row and column), its complete calculus may start with independence of any other running jobs. At the time that a sub-matrix has been completely calculated, its final row and column are stored in the cache, and the sub-matrix is discarded to free memory. Such row and column allow the immediately next sub-matrices to be computed using the resulting row and column as initial row or column for the next job (depending on its placement, if left or down). For this reason, if k is the size of a job, the actual size of each sub-matrix will be $(k+1)*(k+1)$; and k will determine as well the number of sub-processes, $ceil(n/k)*ceil(m/k)$, taking into consideration that if n or m is not multiple of k, all sub-matrices in the DPM's final row and column will have a smaller size.

When the last job finishes, the second stage starts, as shown (Fig. 2, right). Starting from the maximum value obtained in the DPM's final row and column in the case of a semiglobal alignment (the local alignment would start in the maximum

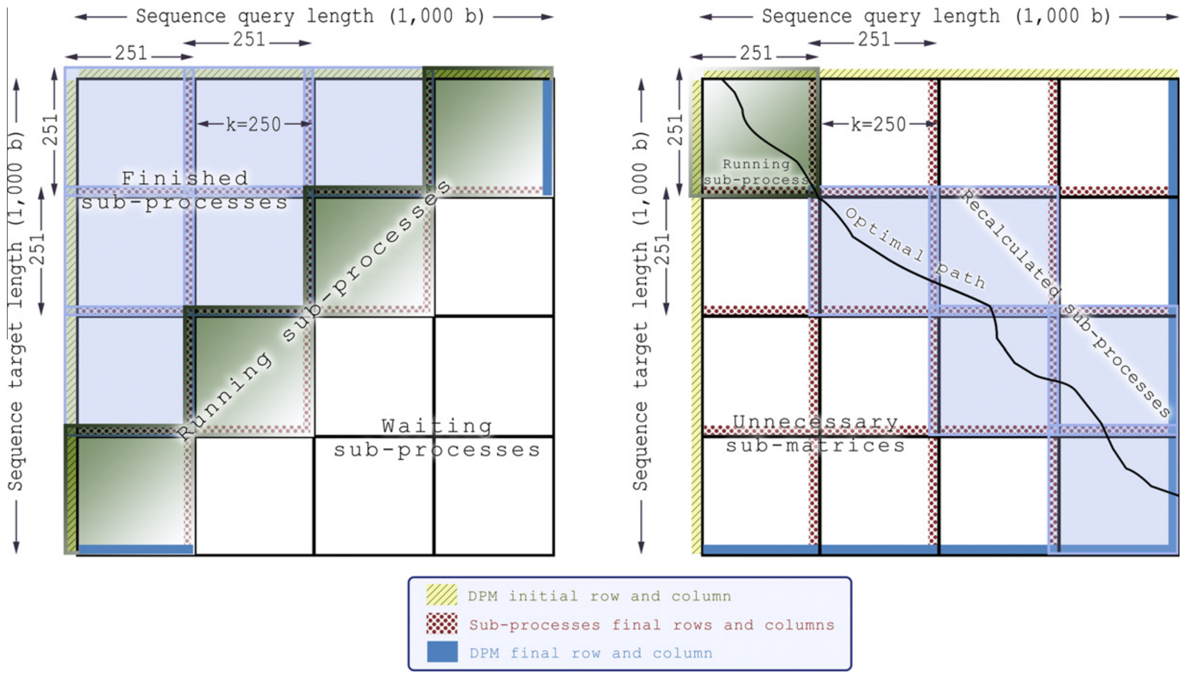

Fig. 2. FastLSA execution in its two stages. The FastLSA execution is shown subdividing the DPM in jobs of $k = 250$ in length. The first stage calculates the DPM using a wavefront parallelism strategy for the jobs, storing the resulting rows and columns in the grid cache. The second stage uses the stored data to re-compute only the necessary sub-matrices, in order to track the optimal path.

value cell all over the DPM), only the required sub-matrices are recalculated using the grid cache values previously stored, so that the optimal path (which defines the alignment) is obtained by backtracking, from where such value came from. At the time a boundary of the sub-matrix is reached, the adjacent sub-matrix is recalculated, and the process continues until it reaches the top or left DPM boundaries (in the case of a local alignment, the process ends when the first zero value is reached). In this second stage, only a few sub-matrices must be recalculated, as shown (Fig. 2, right), with $max(n,m)$ in the best case scenario, and $n+m-1$ in the worst case scenario. Thus, no remarkable parallelization improvement can be obtained in this linear time–cost stage, which is nevertheless much faster than the quadratic time–cost of the first stage.

4.3. Implementation

To estimate the potential of the Tile64 many-core processor, the aim of our work was to: (i) implement a parallel version of the efficient space cost FastLSA in the Tile64 architecture, both for the global Needleman–Wunsch aligner and the local Smith–Waterman aligner (that is, the MC64-NW/SW); (ii) run several comparative and benchmarking tests; and (iii) optimize the performance of the algorithm, taking into account the Tile64 behavior.

First of all, we decided to write the MC64-NW/SW from scratch instead of porting an existing C implementation, to adapt and optimize the communication between sub-processes to the Tile64 iLib. This approach facilitates to update and optimize code fragments. Furthermore, the source code of the original FastLSA was not available, and the only version we were able to obtain was an Intel C++ compiler-based Parallel Linear Space Algorithm (PLSA) implementation [34] which makes use of the Message Passing Interface (MPI).

The MC64-NW/SW requires saving the grid of rows and columns into a specific cache available for all the tiles of the Tile64 processor, which are computing sub-matrices in parallel. For this reason, it is mandatory to define a communication strategy to coordinate the parallel first stage of the FastLSA. To do this, we propose a task scheduling with a Controller encapsulating and managing the so-called Job Table structure (the grid cache), as shown in Fig. 3. The Controller initializes the Job Table and launches all the available Worker tiles (the number of effectively available tiles is reduced from 64 to 59, because one tile is used as Controller and four tiles are reserved for internal communication purposes, one dedicated and three shared), and keeps record of the idle tiles and the outstanding jobs. While there are queued jobs to be executed and idle Workers available, the Controller wakes up any idle Worker and sends it the job to be carried out. If neither jobs nor Workers are available, the Controller waits to receive results from any Worker and, then, stores the resulting data (row and column) and updates the list of free Workers and outstanding jobs. This cycle is repeated until the last job (bottom right one) is finished. From the Worker's point of view, it awaits to receive a new job from the Controller with all the data information required about the job to be done. Once received, the Worker computes the right column and bottom row of the sub-matrix using only two columns, instead of a $(k+1)*(k+1)$ two-dimensional array [23], and then sends the results back to the Controller: the final row and column, as well as transactional information.

The Job Table structure must be accessed not only by the Controller, but also by the Workers, since the latter will need to read and write the initial and final rows and columns of their assigned job. To share data between the Controller and the Workers, several strategies can be adopted, depending on memory restrictions and performance. In the first algorithm

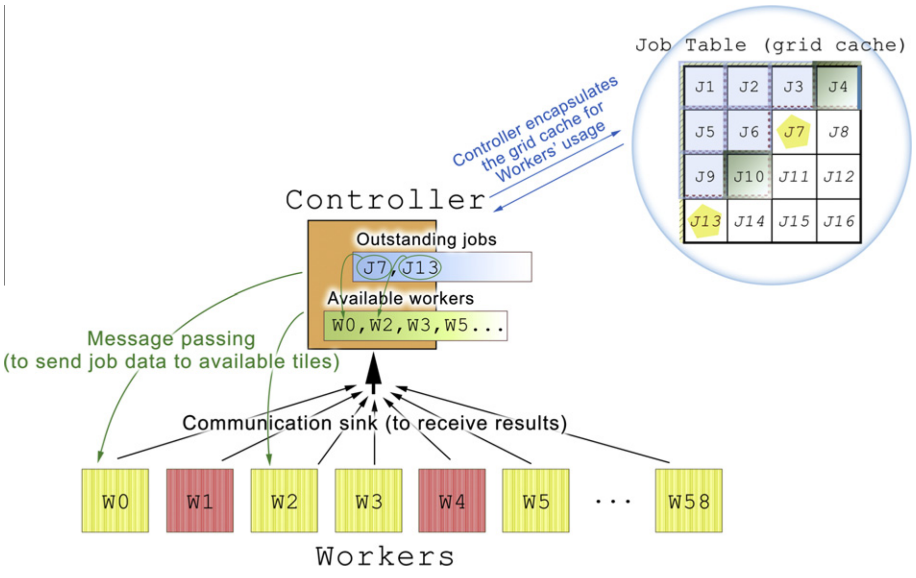

Fig. 3. Parallel FastLSA implementation with task scheduling. A Controller encapsulates the grid cache (Job Table structure), and manages the lists of jobs and available Workers, to send the necessary data of the Job Table to perform the task. Once a Worker finishes its work, it sends the results back to the Controller through a common communication sink.

version, the MC64-NW/SW uses shared memory for the Job Table access, which is the most intuitive approach, but not the optimal one. This Job Table, as well as many other structures and concepts, has been implemented using a highly modular approach, to easily allow further optimizations. For example, the Job Table uses what we have called Job Fragment, which is a data structure representing the sub-matrix coordinates for both row and column. This simplifies programming, as now the Job Table is a matrix of Job Fragments, so a Worker processes Job Fragments as in/out data.

Parallelizing tasks in the Tile64 is similar to fork processes in a multiprocessing system: each parallel task has its own independent memory for data and code. The iLib library allows parallelizing the code that is being executed, replicating code and data to the new processes using a fork-like iLib function. As our strategy is based on the Controller and Workers, the former uses a different executable than the latter, so the Controller spawns the parallel tasks using a specific iLib function parameterized with the executable file to be processed by each Worker. With this approach, smaller programs are generated that can be fully stored inside each tile's local code cache. Anyway, global resources like the substitution matrix and configuration parameters must be accessed by all the Workers, so they have to be stored in shared mode when allocating memory.

Two different communication strategies have been used to schedule processing: message passing and communication sink. The former is used when the Controller sends outstanding jobs to an available Worker. To send and receive a message, appropriate iLib functions must be parameterized with a specific *tag* identifying the communication, as well as the Worker's identity by means of its rank and its process group. The identity of each tile involved is known as follows: (i) the Controller knows the rank of the available tiles in the set of Workers through an special group structure created when spawning them; and (ii) any Worker knows its parent's (Controller's) group through an iLib-specific constant and its rank, which is always 0, as the Controller is the only tile in its group. This is a blocking communication model, which forces the idle Workers to wait for new jobs to be sent by the Controller. On the other hand, to receive the results from any Worker, the Controller uses a non-blocking communication sink, which receives results indiscriminately from all the senders. This sink is a special iLib raw channel, which takes advantage of the high-performance network hardware to communicate two tiles with very low latency, albeit with the use of a very small queue buffer.

When a Worker finishes its work, it saves the results into shared memory and sends back to the Controller only the pointer (i.e., a word) to this shared memory region, which will include not only the data but also the Worker's rank. The receiving port buffer can store up to 96 words, so it is big enough to allow proper communications. A sink channel uses one port from the Controller and multiple sending ports from the Workers, storing in the buffer all the incoming data. Once the communication channel has been open synchronously on both receiver and senders' sides, the Controller and Workers can use the iLib raw channel functions to receive/send the pointer, as previously stated. Unlike message passing operations, sending data in sink raw channel is a non-blocking function, so a Worker will send its results and will wait for receiving the next job by message passing. Receiving data is a blocking function, so the Controller will be forced to wait until a Worker sends a finished job, in case of the channel's buffer being empty. This represents the starting version of the MC64-NW/SW, which can be further improved and optimized, taking into account empirical results and benchmarks, as described below.

4.4. Optimization

Following the guidelines of a spiral software development and optimization process, we have gathered in this section the consecutive key improvements to the algorithm and their severe impact on the overall performance. This evolution shows the architecture bottlenecks, and how different communication/memory management strategies can avoid them.

Our first approach of parallel FastLSA for Tile64 architecture offers impressive results, due to its high parallelization factor. Furthermore, many enhancements can be added in order to improve the memory access, give more functionality and allow larger sequence alignments. The comparison chart of Fig. 4 illustrates the gained efficiency of changes introduced in consecutive versions of the MC64-NW/SW. In this benchmarking comparative, all the versions have been evaluated in the same context and conditions: (i) same parameter configuration (complete semiglobal alignment with $k = 1000$); (ii) same compiling options; (iii) same hardware environment; (iv) same Tile64 boot ROM using 60 tiles (from 64; three are shared and one is dedicated, as commented above); and v) same sequences to align of approximately 360 kb; namely, the mitochondrial genome of the thale cress (*Arabidopsis thaliana*; GenBank GI number: 49256807; Accession number: Y08501) and the mitochondrial DNA of the field beet (*Beta vulgaris* subsp. *vulgaris*; GenBank GI number: 47118321; Accession number: BA000009). Eqs. (1) and (2) are particularly useful to study and optimize the memory usage, which is a key performance factor. Eq. (1) shows the size of a row from a grid cache, which only depends on the structure data to save and the length n of the query sequence (it is important to notice that the k value has no influence on the memory used to store a grid cache row). On the other hand, Eq. (2) shows the complete grid cache size, which depends on the previously calculated grid cache row size, the length m of the subject sequence and the k parameter.

$$grid\ cache\ row\ size = (k+k)^{*}n/k^{*}(struct\ size) = 2n^{*}(struct\ size) \tag{1}$$

$$grid\ cache\ size = (grid\ cache\ row\ size)^{*}m/k = 2nm/k^{*}(struct\ size) \tag{2}$$

In the first version of the MC64-NW/SW (v1.0 in Fig. 4), the Needleman–Wunsch parameters are set as internal constants, not requiring to share data between the Controller and the Workers. This makes programming easier, but lacks algorithm configuration flexibility for the life scientists. In addition, the match/replace calculation function is inefficient, because it

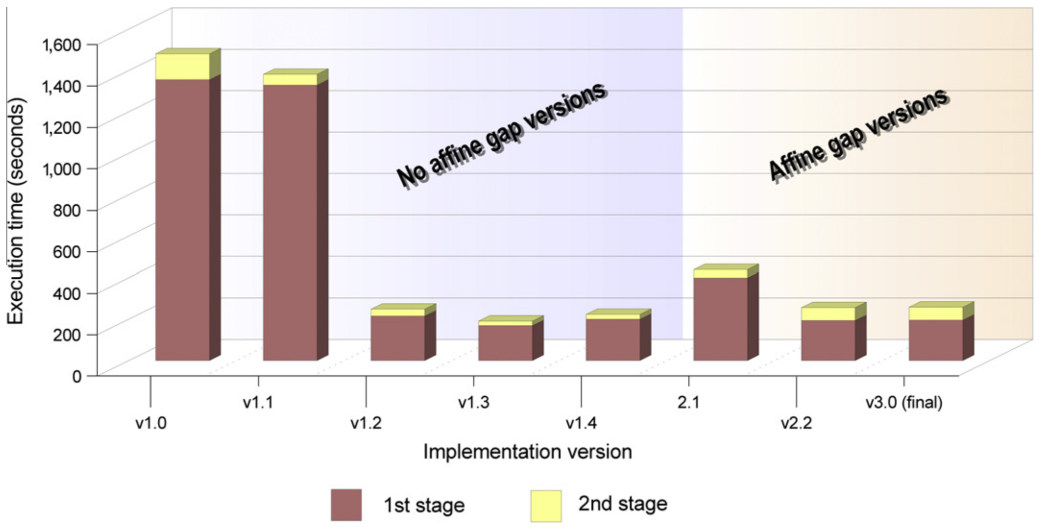

Fig. 4. Execution time comparison of MC64-NW/SW consecutive versions for 360 kb sequences. The original Needleman–Wunsch without affine gap implementation (version 1.0 to 1.4 update) is shown: improvements, optimizations of memory access, and usage and storing data in local worker's memory space. The Needleman–Wunsch with Gotoh's affine gap implementation (version 2.0 to 2.2 update) is also shown: grid cache memory usage optimization and memory swapping. In the latest version (3.0), the local alignment (Smith–Waterman) is fully optimized, with extra added functionalities.

looks for bases' indexes in the substitution matrix and then obtains the value (linear cost). As this calculus is the core of the pairwise alignment, its improvement is especially relevant.

To allow usage of a standard substitution matrix, the first update of the MC64-NW/SW (v1.1 in Fig. 4) shares this structure as a variable between the Controller and the Workers. Regarding the match/replace linear cost function, an auxiliary array saves indexes for each base, which takes part in the selected matrix. Since bases are represented as ASCII characters, an array of 128 byte values (8 bits) is enough to reference an index, taking into account the ASCII character encoding value (7 bits). Thus, this allows the substitution matrix to be evaluated in an optimal two-single-steps access approach (indirect access).

In Tile64, every tile runs faster when data is located in its own local memory space, so the next relevant optimization deals with the memory allocation of the required data. In the second update (v1.2 in Fig. 4), the subsequences for a specific job and the matrix structure are copied from shared memory into the tile's local memory. Finally, in the third update (v1.3 in Fig. 4), the initial Job Fragment is cloned into the local space. The consequence of these optimizations is an amazing speedup factor of more than a 680% in relation to the non-optimized one. This is due to the specific characteristics of the Tile64: the shared memory allocated can be accessed by all process, but it is homed on the tile which makes the allocation call. This forces the other tiles to send a request to the home tile's cache, thus consuming tens of cycles instead of the just two cycles required to access the local private memory.

Once the original basic algorithm has been improved in time cost, it is mandatory to improve the accuracy and tune up the results, using the Gotoh's affine gap implementation [22]. For the latest update of the base algorithm (v1.4 in Fig. 4), the DPM of *long* values has been replaced by a DPM of structures containing only the *long* value, in order to add in the future the two more values that are needed for affine gap implementation (instead of having three different matrices, the best memory access performance is obtained this way). In addition, the algorithm uses parameters instead of constants, and they are shared between the Controller and the Workers the same way as with the substitution matrix. These changes slow slightly the algorithm, but are needed to make the algorithm useful for life scientists, in order to tune up algorithm parameters.

From this point on, MC64-NW algorithm will be able to use the affine gap extension, in order to obtain more accurate results from a biological point of view. As the algorithm changes its behavior, we start a new version numbering; in fact, this is not a time/space optimization, but adds a more time- and memory-consuming functionality. The first implementation (v2.0 not shown in Fig. 4) follows the guidelines exposed in the Algorithm 2, including three *long* values into the DPM structure (one for each matrix), which in fact consumes triple of the memory stored in the cache. For this reason, for the 360 kb sequence example (with a *k* value of 1000), the grid cache size follows the Eq. (3).

$$grid\ cache\ size = (grid\ cache\ row\ size)^{*}m/k = 2nm^{*}(struct\ size)/k = 2nm^{*}(3^{*}4\ bytes)/k = 2.9\ GB \qquad (3)$$

As the grid cache is stored in the Controller's local memory and Tile64 has a 2.8 GB limit of local space, the grid cache cannot be stored and the algorithm halts its execution. A new optimization can be included, which allows the DPM to save only the actual value, and the differences of the vertical and horizontal open gaps, instead of saving their own complete values [16]. According to this, the DPM can use a structure of one *long* value and two *bytes*, with a total size of 6 bytes. However, the Tilera's C compiler aligns structures considering the largest member, as the GCC compiler does, using in this case 8 bytes and wasting the last 2 ones with zeroed-values (which mean nearly 500 MB wasted in the complete grid cache for a 360 kb sequence). To avoid this, a *pragma* pack compiler directive has been used. With these new changes, the memory requirements halve in the first update of the second version (v2.1 in Fig. 4).

However, for larger sequences, the same issue will arise, even using the *pragma* directive and having a total memory of 8 GB in the TilExpress-20G card. To overcome this situation, a second update of version 2 is proposed (v2.2 in Fig. 4), which makes the Controller to swap the grid cache to the file system in the SSD. This new version is a little slower, due to the Controller overload when saving data into files, but expands the available grid cache memory up to 7.8 GB. This allows accurately aligning sequences of more than one million bases. Due to the internal TilExpress card factors, the memory controller efficiency drastically degrades at increasing common memory usage, but this does not happen when using the file system capacity. This allows the new file system swapping version (v2.2) to gain a 100% speedup in relation to the previous update (v2.1), due of the common memory local accession, as the empirical results show. Therefore, the Controller saves a row of Job Fragments when it has been completely calculated, loads it from the file system in the backtracking stage and, when used, deletes it (reloading files during the second stage markedly slows its execution). As shown previously in Eq. (1), the number of files to be saved depends on the k value and the subject sequence's length, but the size of each file is k-independent.

In the latest version (v3.0 in Fig. 4), the FastLSA zero-score padding gap penalties are implemented and the NW is extended for local alignment (Smith-Waterman algorithm), which justifies a new version numbering. Also, now it is possible as well to run only the first stage to obtain the alignment score.

To summarize, v3.0 includes the next main optimizations: full parameterization, faster match/replace calculus, optimal use of local memory instead of shared memory, affine gap functionality with cell size reduction, maximization of the Tile64 memory usage when swapping in the SSD and semiglobal/local implementation. From this point on, the MC64-NW/SW is complete and allows running both semiglobal and local optimal alignments at high speed, thanks to its implementation from scratch and architecture optimization.

4.5. Benchmarking

To objectively evaluate the performance of the MC64-NW/SW, it must be tested against similar applications executed in several different platforms, in the context of a single PC. We have focused on running an accurate pairwise alignment algorithm of large sequences, which obtains results in a reasonable small amount of time in a standalone PC (i.e., domestic hardware). Several applications have been developed with the same goal, but they are usually geared to align a large number of very short sequences instead of aligning large genomes. The GPU-based algorithms have been tested, but they are out of scope in this comparison chart, since they only support relatively short sequence alignments. For example, the CUDA

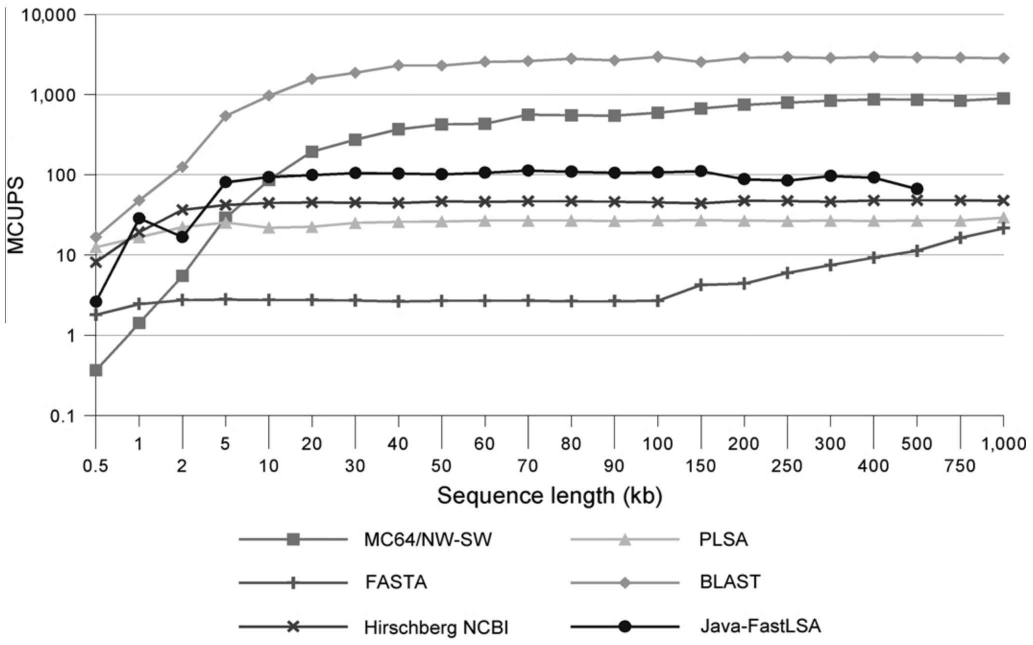

Fig. 5. Benchmark of several pairwise alignment algorithms with pseudo-random generated sequences of different sizes. The performance is expressed in MCUPS. MCUPS values have been estimated in order to allow a speed comparison with heuristics (BLAST and FASTA) which do not use a DPM. The MC64-NW/SW algorithm has been executed in a TilExpress-20G board with optimal k-value. The BioBench PLSA, Java FastLSA (unable to align sequences larger than 500 kb), NCBI C++ toolkit, Hirschberg and heuristics BLAST and FASTA have been executed in the host machine with optimal parameterizations (BLAST and FASTA are the only non-parallel algorithms, and use *task* = *blastn* and *ktup* = 3 as parameter configurations, respectively).

Smith–Waterman [35] has a limit of 2050 b, and CUDASW++ [36] does not support sequences longer than 59,000 b. Other methods are also oriented to align small sequences in databases, like Farrar's [37], where the DPM cell score is limited to 16 bits. We include, however, not only optimal aligners but also heuristic algorithms, like the widely extended BLAST [31] and FASTA [32]. These algorithms rely on heuristic methods based in partial aligning comparisons of sequence similarities and *k*-tuples, respectively, as they are optimized to find the most similar sequences in huge databases. The BLAST is the fastest and most popular method, but its results are not very accurate [38], as demonstrated when using the Pearson's "missed@equivalence" point method [39,40].

The latest version of the MultiCore64-NW/SW is thus compared in Fig. 5 to the PLSA parallel implementation of FastLSA, used for benchmarking in BioBench [41], both being parameterized with the best suitable *k* value in each case. As a performance comparison between fine-tuned C and Java programs may be considered well-balanced [42], we have included in this comparison a reference multi-threaded parallel FastLSA implementation in Java (an easier to use programming language). In addition, the linear space Hirschberg's algorithm from the NCBI C++ toolkit implementation [43] has been added, using a multi-threaded configuration. The heuristics BLAST and FASTA are evaluated as well in the comparison chart using *task* = *blastn* and *ktup* = 3 as parameter configuration, respectively. The FASTA does not allow multi thread execution. Therefore, although the BLAST allows it by means of the *num_threads* parameter, both heuristics are executed in a single thread to obtain a more homogeneous comparison. On the other hand, the Z-align is a parallel multi-core strategy method which can align sequences larger than 3 megabases (Mb) [44] that was originally considered [10], but it has been discarded from the chart due to its worse out-of-scale results. These applications have been executed in an Intel Xeon Quad Core 2.0 GHz PC with 8 GB of DDR2 memory in quad-channel. None of these algorithms have been executed in the Tile64 architecture, because porting them to the iLib is out of the scope of this work. The MC64-NW/SW has been executed in a TilExpress-20G board using 60 effective tiles (59 Workers and one Controller).

5. Results and discussion

To our knowledge, the MC64-NW/SW is the fastest implementation of a pairwise alignment algorithm for relatively large sequences in a standalone PC, which follows the concept of 'optimal alignment', taking into account the optimal string alignment with affine gap. This is why nowadays an optimal alignment definition does not exist, either in biology or in bioinformatics. Thus, it is widely accepted that this definition must rely by now on the Computer Science string alignment concept. Heuristics like the BLAST or the FASTA do not follow this guideline, so their results are bind to their speed and fine-grained behavior, being demonstrated that their results do not surpass the optimal aligners [38]. Aligning large sequences (e.g., more than 100 kb), may not be addressed by quadratic-space algorithms, with only efficient- and linear-space approaches being able to manage such large sequences, which are still very small when compared to full genomes having billions of bases [45].

The results from Fig. 5 reveal the efficient performance of the MC64-NW/SW against other efficient space implementations. Thus, the PLSA algorithm does not show a remarkable speedup against Hirschberg as [16] determines (the original FastLSA implementation in ChromaTool sequence analysis suite is no longer available). In spite of that, the FastLSA is the best choice to parallelize an alignment algorithm in many-core microprocessors, as it allows a bigger parallelization factor and decreases the Hirschberg's number of operations, through a better use of available memory. In order to have a real efficiency comparison, we developed our reference implementation in the $\times 86$ architecture (the multi-threaded Java FastLSA), which indeed achieves more Mega Cell Updates Per Second (MCUPS) than the Hirschberg and the PLSA algorithms. This Java FastLSA is a better non-heuristic implementation, though some memory usage problems appear for very long sequences with small *k* values, due to the language innate object-oriented approach. For this reason, the optimization factor decreases considerably for larger sequences, being unable to align sequences larger than 500 kb. Besides, the heuristic algorithms' MCUPS performance are very different, as well as their results. As above mentioned, the BLAST results are the fastest, but not the most accurate ones. On the other hand, the FASTA results are slightly more accurate than the ones of the BLAST, yet the MCUPS performance is even worse than the one obtained with the efficient space algorithms. Anyway, only linear or efficient space implementations can align very long sequences accurately. The FastLSA has the best performance from a scalability point of view, and thus, the MC64-NW/SW is the best algorithm for aligning large sequences in a standalone PC, with results being one order of magnitude faster.

The MC64-NW/SW is based in work distribution among several processes. As Fig. 6 shows, the MC64-NW/SW is an algorithm with a high scalability, being therefore a good example to illustrate the advantage of appropriately parallelized algorithms on many-core chips *versus* single-core implementations. The scalability only degrades when the sequences are very short and most of the tiles are not really needed. On the other hand, the scalability is uniquely limited by hardware resources (number of tiles, available memory and sink channel buffer size), but not by the algorithm's behavior.

Furthermore, the more tiles are working, the less is the impact of the Controller's delay for saving a complete row of Job Fragments, because only a Worker waits, so a higher percentage of Workers keep doing their jobs. The theoretical execution time for the first stage is calculated from Eq. (4), where c is the number of cores, w is the number of sub-matrices to be processed in a row/column (in a square matrix), and t is the time required to process a sub-matrix by a core:

$$\text{Theoretical execution time} = t\left(\frac{w^2}{c} + c - 1\right) \tag{4}$$

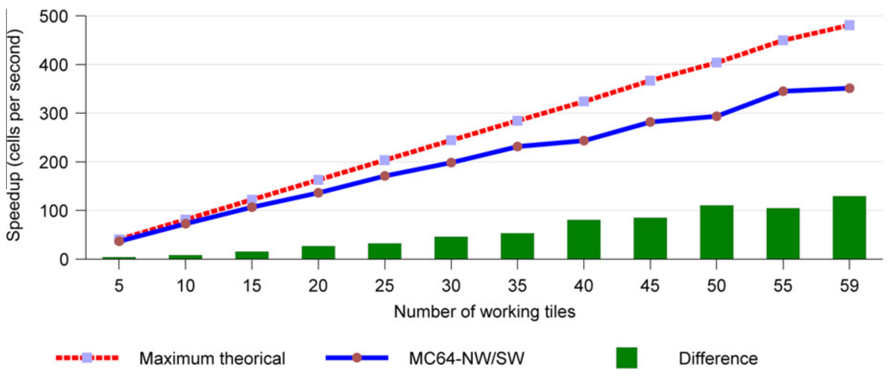

Fig. 6. Scalability speedup in the first stage of the MC64-NW/SW algorithm. The processing time of the first stage of the MC64-NW/SW is shown for sequences of 250 kb, with an increasing number of active executing tiles.

Eq. (4) takes into account the time required by the wavefront algorithm to achieve a state in which all the tiles are working, as well as the time consumed at the end of the wavefront (when the execution begins and ends, respectively). The difference between the real execution time and the theoretical time is caused by the Controller communication with Workers and the delay of saving grid cache rows, being displayed for each case in Fig. 6.

The MC64-NW/SW has been developed from scratch and meticulously optimized, taking advantage of the Tile64 architecture characteristics, being able to align very large sequences and exploiting parallelism in order to decrease execution time. To our knowledge, our algorithm in the TilExpress-20G card can align a pair of sequences of up to 1 Mb 15 times faster than any other standalone PC implementation. Of course, a proper parameterization of the algorithm is essential to achieve the optimal usage; i.e., tuning up the *k*-value influences in: (i) the dimensions of a sub-process' matrix; (ii) the average number of working tiles; and (iii) the memory required to save Job Fragments. As the Controller scheduling time and the memory access time through iMesh are much lower when compared to processing time, the optimal *k* values should maximize the number of processing tiles, but avoiding an excessive fragmentation of the matrix. In fact, the *k* value is limited by memory resources: with very long sequences (larger than 750 kb), *k* must be larger than with shorter sequences, in order to decrease the amount of the SSD memory required, as seen in Fig. 7. This certainly leads to a decrease in performance, but it is mandatory since the memory is limited. Even so, performance is still greatly superior to any other optimal alignment algorithm in a standalone system. The performance speedup factor increases from 1.22 to 9.91 for sequences from 20 kb (small size) up

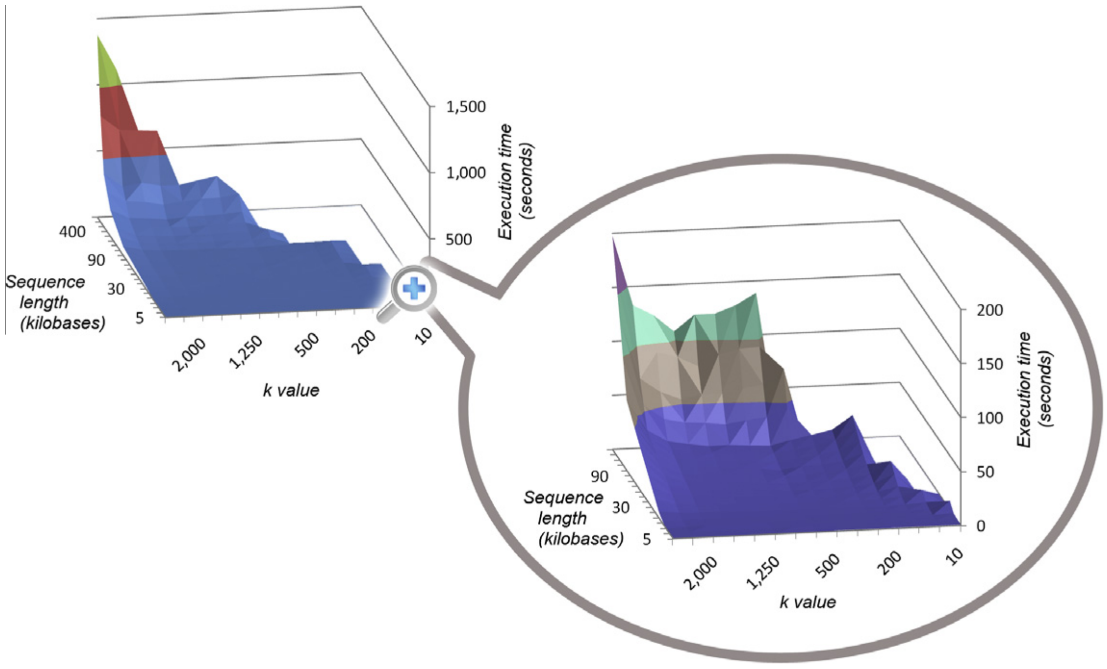

Fig. 7. Relationship between the *k* size, sequence length and processing time in the MC64-NW/SW algorithm. The analysis of the MC64-NW/SW relationship level between the sequence length (both have the same length), *k* value and processing time is shown. The undefined values are not possible: short sequences with higher *k* values and large sequences with comparatively small *k* values.

to 500 kb (medium size), respectively, against the same algorithm on a $\times 86$ multi-core architecture (which is also the fastest, but unable to align large sequences). The performance speedup factor increases even up to 15.00 for larger sequences of one gigabase (Gb) against the best $\times 86$ algorithm able to align them (NCBI Hirschberg).

6. Conclusions and further work

We have previously demonstrated the usefulness of the Tile64 architecture for bioinformatics [10]. To our knowledge, the MC64-NW/SW is the fastest implementation of a pairwise alignment algorithm for large sequences in a standalone PC. Taking into account the Tile64 architecture characteristics and parallelization factor of its 64 cores, it has been proved that for algorithms with no need of floating point calculi, the TilExpress card has a great bioinformatics potential. Although such hardware would be less effective for supercomputation usage, the Tile64 architecture and other many-core chips running parallel applications are capable of driving the next-generation bioinformatics that is required for the massive and exponentially growing data generated by the so-called next-generation sequencing platforms.

The effectiveness of parallelizing bioinformatics algorithms and methods has been demonstrated as well: alignment algorithms are space-limited and very time-consuming for large sequences, but such drawbacks can be drastically reduced when distributing work among the available cores. The MC64-NW/SW is an algorithm with a high scalability, which in turn depends on the Tile64 scalability as well. The Tile64 scalar behavior is excellent for the MC64-NW/SW, while taking into consideration the *k* value and memory usage, albeit the maximum number of working tiles is able to use is 60 (59 Workers and one Controller) for a correct usage of the processor, as explained above.

On the other hand, the described performance can be further escalated by connecting two or more cards by means of a Tile64 board cluster (work in progress). For this interconnection level approach, the TilExpress-20G provides two 10 Gigabit Ethernet ports using the standard 10GBase-CX4 connectors. The wire and connectors are similar to the InfiniBand's ones and they are principally oriented to connect servers in computer room environments, allowing the use of up to 15 meter long wires. This standard has been built for cross-connectivity, so it is possible to connect two cards with two wires obtaining a cluster with a speed of 20 Gb/s or three cards in a ring, with each card having 10 Gb/s to the others without any other hardware needs. For further degrees of aggregation, a 10GBase-CX4 switch should be used.

In the software level, in order to use a Tile64 on-boarded card in a cluster way, Tilera provides the API NetIO to use its Ethernet ports at maximum speed. Hence, the MC64-NW/SW would keep using the iLib communication mechanisms with the tiles in the same card through the iMesh network, and now would use the NetIO to communicate with the tiles in the others cards of the cluster through Ethernet.

The Tilera's iLib is close to the standard C communication API, but lacks an MPI library implementation, which may make more difficult to port some parallel applications. In addition, only a C/C++ compiler is available in the Tilera MDE, with no other development tools and programming languages like Java (one of the most attractive ones, from a programmer's point of view). Nevertheless, programming for the Tile64 processor is easier than for other many-core architectures like CUDA, which demands changing structural programming paradigms and great parallelization-exploiting efforts [46]: applications are designed as common multi-threaded programs using tasks for cores, instead of using threads or processes. In this sense, as the Tile MDE programs are written in C/C++, the door is open for application porting. Thus, several parallel bioinformatics algorithms have been coded in such languages, like ABySS [47] or mpiBLAST [48]. Many of them use the MPI [49], which provides the most widely spread distributed communication interface.

In summary, the MC64-NW/SW has proved to be much faster than other widespread implementations of NW and SW, generating better results in less time than some of the most used heuristic pairwise aligners. Thus, our algorithm in the TilExpress-20G card can align a pair of sequences of up to one Mb 15 times faster than any other standalone PC implementation. The BLAST algorithm provides results faster, but they are far from accurate [38]. In a practical perspective, we have demonstrated that the MC64-NW/SW is capable to aligning sequences of more than one Gb, which we have used to develop methodologies for quality control and fraud prevention of olive oil [21]. Such developments lay the groundwork for future multiple alignment algorithm parallelization, like ClustalW [50] or PRANK [51] that would be able to align a complete genome in much less time and with more accuracy than current implementations with heuristics. Usually, a phylogenetic tree is the basis of progressive Multiple Sequence Alignments (MSA), and this can be derived from a triangular matrix of distances obtained from full pairwise alignments between the sequences, using clustering methods like the Neighbor-Joining [52] or the Unweighted Pair Group Method, using Arithmetic Averages (UPGMA) [53]. For example, for 10 sequences of approximately 100 kb, the MC64-NW/SW can calculate the matrix with 55 NW or SW pairwise alignments in 765 s (about 13 min) [10].

Hence, the MC64-NW/SW opens a door for parallel implementations in emerging many-core technologies, in order to support the massive computational needs of research fields like bioinformatics in a cost effective way. These and similar developments will then have a huge impact on bioinformatics, allowing to fully exploit genomics in general and the current and future so-called "next-generation" sequencing methodologies in particular.

Acknowledgements

We are grateful to Tilera for providing hardware and software tools http://www.tilera.com. This work was supported by "Ministerio de Ciencia e Innovación" (AGL2010-17316 and BIO2009-07443-E grants); "Consejería de Agricultura y Pesca" of

“Junta de Andalucía” (041/C/2007, 75/C/2009 and 56/C/2010); “Grupo PAI” (AGR-248); and “Universidad de Córdoba” (“Ayuda a Grupos”), Spain.

References

[1] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, T.J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (2007) 80–113.
[2] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, D. Shippy, Introduction to the cell multiprocessor, IBM Journal of Research and Development 49 (2005).
[3] T. Chen, R. Raghavan, J.N. Dale, E. Iwata, Cell broadband engine architecture and its first implementation: a performance view, IBM Journal of Research and Development 51 (2007) 559–572.
[4] A.K. Nanda, J.R. Moulic, R.E. Hanson, G. Goldrian, M.N. Day, B.D. D’Amora, S. Kesavarapu, Cell/B.E. blades: building blocks for scalable, real-time, interactive, and digital media servers, IBM Journal of Research and Development 51 (2007) 573–582.
[5] A. Ruiz, N. Guil, M. Ujaldon, Recognition of circular patterns on GPUs: performance analysis and contributions, Journal of Parallel and Distributed Computing 68 (2008) 1329–1338.
[6] J. Kurzak, W. Alvaro, J. Dongarra, Optimizing matrix multiplication for a short-vector SIMD architecture – CELL processor, Parallel Computing 35 (2009) 138–150.
[7] T.G. Mattson, R.V.D. Wijngaart, M. Frumkin, Programming the Intel 80-core network-on-a-chip Terascale processor, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE Press, Austin, Texas, 2008, pp. 1–11.
[8] M. Shah, J. Barreh, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Saha, D. Sheahan, L. Spracklen, A. Wynn, in: UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC Asian Solid-State Circuits Conference (ASSCC’07), 2007, pp. 22–25.
[9] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, J. Zook, TILE64 - Processor: A 64-Core SoC with mesh interconnect, in: Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International, 2008, pp. 88–598.
[10] S. Gálvez, D. Díaz, P. Hernández, F.J. Esteban, J.A. Caballero, G. Dorado, Next-generation bioinformatics: using many-core processor architecture to develop a web service for sequence alignment, Bioinformatics 26 (2010) 683–686.
[11] W. Flohr, Implementation of and MPEG Codec on the Tilera 64 processor, Department of Electrical and Systems Engineering, Washington University, St. Louis, 2008.
[12] C. Trapnell, M.C. Schatz, Optimizing data intensive GPGPU computations for DNA sequence alignment, Parallel Computing 35 (2009) 429–440.
[13] V. Sachdeva, M. Kistler, E. Speight, T.H.K. Tzeng, Exploring the viability of the cell broadband engine for bioinformatics applications, Parallel Computing 34 (2008) 616–626.
[14] S.B. Needleman, C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, Journal of Molecular Biology 48 (1970) 443–453.
[15] T.F. Smith, M.S. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology 147 (1981) 195–197.
[16] A. Driga, P. Lu, J. Schaeffer, D. Szafron, K. Charter, I. Parsons, FastLSA: a fast linear-space, parallel and sequential algorithm for sequence alignment, Algorithmica 45 (2006) 337–375.
[17] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, A. Agarwal, J.F. Brown III, On-chip interconnection architecture of the tile processor, IEEE Micro 27 (2007) 15–31.
[18] P. Rice, I. Longden, A. Bleasby, EMBOSS: the European Molecular Biology Open Software Suite, Trends in Genetics 16 (2000) 276–277.
[19] F.E. Allen, J. Cocke, A catalogue of optimizing transformations, in: Design and Optimization of Compilers, Prentice-Hall, Englewood Cliffs, NJ, 1972, pp. 1–30.
[20] R. Eigenmann, Toward a methodology of optimizing programs for high-performance computers, in: International Conference on Supercomputing, 1993, pp. 27–36.
[21] A. Castillo, P. Pascual, E. Rodríguez, D. Díaz, M.G. Claros, J. Falgueras, S. Gálvez, G. Dorado, P. Hernández, Genomic approaches for olive oil quality control, in: Plant Genomics European Meetings (Plant GEM 6), Tenerife, Spain, 2007.
[22] O. Gotoh, An improved algorithm for matching biological sequences, Journal of Molecular Biology 162 (1982) 705–708.
[23] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Communiations of the ACM 18 (1975) 341–343.
[24] P.H. Sellers, An algorithm for the distance between two finite sequences, Journal of Combinatorial Theory, Series A 16 (1974) 253–258.
[25] M.S. Waterman, T.F. Smith, W.A. Beyer, Some biological sequence metrics, Advances in Mathematics 20 (1976) 367–387.
[26] J.C. Setubal, J. Meidanis, Introduction to computational molecular biology, PWS Pub., Boston, 1997.
[27] E.W. Myers, W. Miller, Optimal alignments in linear space, Computer Applications in the Biosciences 4 (1988) 11–17.
[28] X.Q. Huang, R.C. Hardison, W. Miller, A space-efficient algorithm for local similarities, Computer Applications in the Biosciences 6 (1990) 373–381.
[29] S. Aluru, N. Futamura, K. Mehrotra, Parallel biological sequence comparison using prefix computations, in: Parallel Processing Symposium, International 0, 1999, p. 653.
[30] S.M. Wellington, C. Juan, C. Wenwu, R.G. Guang, Whole genome alignment using a multithreaded parallel implementation, in: Symposium on Computer Architecture and High Performance Computing, Pirenopolis, Brazil, 2001, pp. 1–8.
[31] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, D.J. Lipman, Basic local alignment search tool, Journal of Molecular Biology 215 (1990) 403–410.
[32] W.R. Pearson, D.J. Lipman, Improved tools for biological sequence comparison, Proceedings of the National Academy of Sciences of the United States of America 85 (1988) 2444–2448.
[33] D. Díaz, S. Gálvez, J. Falgueras, J.A. Caballero, P. Hernández, M.G. Claros, G. Dorado, Intuitive bioinformatics for genomics applications: omega-brigid workflow framework, in: International Work-Conference on Artificial Neural Networks (IWANN), Springer, Salamanca, Spain, 2009, pp. 1084–1091.
[34] E. Li, C. Xu, T. Wang, L. Jin, Y. Zhang, Parallel linear space algorithm for large-scale sequence alignment, in: Euro-Par, Springer, 2005, pp. 1207–1216.
[35] S.A. Manavski, G. Valle, CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment, BMC Bioinformatics 9 (Suppl 2) (2008) S10.
[36] Y. Liu, D.L. Maskell, B. Schmidt, CUDASW++: optimizing Smith–Waterman sequence database searches for CUDA-enabled graphics processing units, BMC Research Notes 2 (2009) 73.
[37] M. Farrar, Striped Smith–Waterman speeds database searches six times over other SIMD implementations, Bioinformatics 23 (2007) 156–161.
[38] E.G. Shpaer, M. Robinson, D. Yee, J.D. Candlin, R. Mines, T. Hunkapiller, Sensitivity and selectivity in protein similarity searches: a comparison of Smith-Waterman in hardware to BLAST and FASTA, Genomics 38 (1996) 179–191.
[39] W.R. Pearson, Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith–Waterman and FASTA algorithms, Genomics 11 (1991) 635–650.
[40] W.R. Pearson, Comparison of methods for searching protein sequence databases, Protein Science 4 (1995) 1145–1160.
[41] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, D. Yeung, BioBench: A benchmark suite of bioinformatics applications, in: ISPASS, IEEE Computer Society, 2005, pp. 2–9.
[42] A. Shafi, B. Carpenter, M. Baker, A. Hussain, A comparative study of Java and C performance in two large-scale parallel applications, Concurrency and Computation: Practice and Experience 21 (2009) 1882–1906.
[43] D. Vakatov, K. Siyan, J. Ostell, The NCBI C++ Toolkit, National Center for Biotechnology Information, 2003.

[44] R.B. Batista, A. Boukerche, A.C.M.A. de Melo, A parallel strategy for biological sequence alignment in restricted memory space, Journal of Parallel and Distributed Computing 68 (2008) 548–561.
[45] E. Bortiri, D. Coleman-Derr, G.R. Lazo, O.D. Anderson, Y.Q. Gu, The complete chloroplast genome sequence of Brachypodium distachyon: sequence comparison and phylogenetic analysis of eight grass plastomes, BMC Research Notes 1 (2008) 61.
[46] J.F. Goetzmann, Massively parallel contact simulation on graphics hardware using NVIDIA CUDA, Institute for Computer Science, Universität Mainz, Mainz, 2007.
[47] J.T. Simpson, K. Wong, S.D. Jackman, J.E. Schein, S.J.M. Jones, I. Birol, ABySS: a parallel assembler for short read sequence data, Genome Research 19 (2009) 1117–1123.
[48] A.E. Darling, L. Carey, W. Feng, The design, implementation, and evaluation of mpiBLAST, San Jose, California, 2003.
[49] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, MIT Press, 1994.
[50] J.D. Thompson, D.G. Higgins, T.J. Gibson, CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice, Nucleic Acids Research 22 (1994) 4673–4680.
[51] A. Löytynoja, N. Goldman, An algorithm for progressive multiple alignment of sequences with insertions, Proceedings of the National Academy of Scienceas USA 102 (2005) 10557–10562.
[52] N. Saitou, M. Nei, The neighbor-joining method: a new method for reconstructing phylogenetic trees, Molecular Biology and Evolution 4 (1987) 406–425.
[53] P.H.A. Sneath, R.R. Sokal, Numerical Taxonomy, The Principles and Practice of Numerical Classification (1973).