A parallel programming framework for multi-core DNA sequence alignment

Tiago Almeida
INESC-ID / IST / TU Lisbon
Lisboa, Portugal
Email: tiago.barreiros.almeida@ist.utl.pt

Nuno Roma
INESC-ID / IST / TU Lisbon
Lisboa, Portugal
Email: Nuno.Roma@inesc-id.pt

***Abstract*—A new parallel programming framework for DNA sequence alignment in homogeneous multi-core processor architectures is proposed. Contrasting with traditional coarse-grained parallel approaches, that divide the considered database in several smaller subsets of complete sequences to be aligned with the query sequence, the presented methodology is based on a slicing procedure of both the query and the database sequence under consideration in several tiles/chunks that are concurrently processed by the several cores available in the multi-core processor. The obtained experimental results have proven that significant accelerations of traditional biological sequence alignment algorithms can be obtained, reaching a speedup that is linear with the number of available processing cores and very close to the theoretical maximum.**

***Keywords*-Multi-core processor; Parallel programming; Computational biology framework**

I. Introduction

In the past few years, the amount of available biological data resulting from sequencing of several genomes has had an exponential growth. In particular, the number of bases included in the GenBank database of nucleotides has doubled approximately every 18 months, reaching a size of 100×10^9 base pairs (bp) in January 2009 [1].

The computation of the optimal pairwise alignment between an individual *query sequence* and each sequence of a broader set of *database sequences* using current Dynamic Programming (DP) methods requires a significant amount of time. As a consequence, other sub-optimal heuristic based algorithms, such as BLAST [2] and FASTA [3], have been developed in order to substantially reduce the time required to compute the alignments. However, the acceleration that is obtained with such methods is often provided at the cost of a non-negligible reduction of the algorithm sensitivity, which means that they are prone to miss some alignments.

To overcome the severe computation time of the alignment algorithms, several parallel implementations have been presented. On one hand, some dedicated hardware solutions were proposed [4], [5], but the provided speedup is obtained at an expensive cost and they are often limited in terms of the offered flexibility level. To overcome such drawbacks, several parallel implementations based on general-purpose computers were also developed. Such solutions are usually based on the application of the set of techniques corresponding to one or more of the following two parallelism levels:

- *Coarse-grained parallelism*, where the sequence database is divided in subsets composed by several complete sequences, to be assigned to the different processors of the system. In this scope, several parallel MPI implementations supported on clusters of workstations have been proposed [6]. However, these solutions often present a difficult tradeoff, due to the significant time penalty that is imposed by the communication stages that are required to transfer the database sequences among the several nodes.
- *Fine-grained parallelism*, where several neighboring partial scores of the alignment procedure between the *query sequence* and the *database sequence* under consideration are simultaneously evaluated. With the inclusion of Single Instruction Multiple Data (SIMD) vector instructions in most processor architectures (e.g.: MMX/SSE2 from Intel, 3DNow! from AMD, etc.), several proposals have been implemented on current general purpose processors [7], [8]. By following a similar SIMD fine-grained approach, other recent proposals were also presented by making use of the streaming processing flow available on current Graphics Processing Units (GPUs) [9], [10]. However, it has been observed that for large databases the limited bandwidth that is available to transfer the data to the GPU tends to be a significant bottleneck.

Meanwhile, with the advent and widespread availability of multi-core general purpose processors on today's commodity computers, a third *intermediate-grained parallelism* level is now worth exploiting. With the new parallel programming framework for homogeneous multi-core sequence alignment that is now proposed in this paper, the biological data of each sequence pair alignment procedure is tiled in several chunks that are individually processed by the several cores available in the multi-core processor. Moreover, since such multi-core processing structures are usually based on shared-memory hierarchies, where all processing cores share a single view of the biologic sequence under processing, data communications between the several workers can be efficiently handled without any significant penalty. The experimental results that were obtained have proven that significant accelerations of traditional biological sequence alignment algorithms can be obtained, reaching a speedup that is linear with the number of cores that are available in the multi-core processing system.

II. DNA Sequence Alignment

The information that is obtained from Deoxyribonucleic Acid (DNA) sequence analysis is only useful if its biological function can be determined. Therefore, whenever a new DNA sequence is acquired, it is matched against a database of other already known sequences to find similarities. Sequence alignment is the process of organizing the sequences in order to find similarity regions between them that will most likely reveal a functional and/or evolutionary relationship. This alignment may include gaps in either of the sequences, as well as nucleotides substitutions arising as a result of mutations. The alignment itself can be a *global alignment*, in which the complete sequences take part in the alignment, or a *local alignment*, in which only certain regions of each sequence that optimally align are considered.

A. Dynamic Programming (DP) Algorithms

Several methods have been proposed to determine the best pairwise alignment between any two sequences. The least complex are based on the DP methods proposed by Needleman & Wunsch (NW) and Smith & Waterman (SW) [11], with a time complexity proportional to $\mathcal{O}(nm)$, where n and m are the lengths of the sequences. While the NW algorithm is applied to determine the best global alignment, the SW algorithm is used to look for the optimum local alignment. Both these algorithms are based on a pre-defined cost function with a biological significance that determines the scores for matches, substitutions and gaps. In particular, the inserted gaps can be modeled in two distinct ways: a *linear* gap penalty model (the opening and extension of a gap have the same cost) and an *affine* gap penalty model (opening and extending a gap have different costs).

B. Local Alignment

Two local alignment methods adopted in some of the most widely used toolboxes (such as BLAST) will be considered in the following description: the original and most often used SW algorithm and a fast alignment procedure based on an optimized reduction of the DP matrix.

1) Smith-Waterman (SW):

The SW algorithm determines the best alignment between any two sequences P and T, by building a DP matrix H and determining the location of the highest score within it. Let $P = p_1 p_2 \ldots p_n$ and $T = t_1 t_2 \ldots t_m$ be the two sequences to be aligned, with lengths n and m respectively. The algorithm starts by building the matrix H, with size $(n+1) \times (m+1)$, by setting $H_{i0} = H_{0j} = 0$ for $0 \leq i \leq n$ and $0 \leq j \leq m$.

The remaining values of the matrix are recursively obtained using the following equation:

$$H_{ij} = \max \begin{cases} H_{(i-1)(j-1)} + Sbt(p_i, t_j), \\ H_{(i-1)j} - gapcost, \\ H_{i(j-1)} - gapcost, \\ 0 \end{cases} \tag{1}$$

for $1 \leq i \leq n$ and $1 \leq j \leq m$. $Sbt(x, y)$ denotes the symbol substitution cost function. In the case of DNA sequences, $Sbt(x, y)$ has a positive value when $x = y$ and a negative value when $x \neq y$. The *gapcost* value represents the cost for opening and extending a gap in the sequences (linear gap penalty model).

After filling the entire matrix, the cell with the highest value represents the local alignment score between the two sequences and its location in matrix H represents the indexes of the last symbol of the subsequences that optimally align. Afterwards, a traceback from this position is performed in order to obtain the alignment between the sequences [11].

2) Reduced Matrix Optimizations:

Due to the significant length of the DNA sequences involved in real world applications, several sub-optimal strategies have been developed. Looking at the ranges commonly adopted by most alignment software packages (such as BLAST), one realizes that many of the used algorithms are inspired in the classic ones but with the application of ambitious heuristics that limit the number of processed cells, in order to make them faster at the cost of optimality. Despite the good results provided by many of such heuristics, they frequently create lots of additional data dependencies, thus making parallelization very hard to implement. An example of such an algorithm with additional data dependencies is shown in Fig. 1. This algorithm does not allow the current score (computed from the DP matrix) to drop more than an amount X from the best score evaluated up to that moment. When such situation occurs, the index of the first column considered in the next iteration is advanced, effectively reducing the number of DP matrix cells to be processed.

Just like this algorithm, there are many others that dynamically change the limits for the first and/or last column

```
bestScore=-INF

foreach line in 1,M {
  foreach column in minCol,N {
    score=maximizeScoreForCell(line,column,score,
                               dpMatrixLine, ...)
    if bestScore - score > X {
      minCol++
    }
    bestScore=max(bestScore, score)
  }
}
```

Figure 1. Alignment with dynamic adaptation of the first column index.

of each line. From a data dependency perspective, these heuristics create an additional dependency between the last cell of a line and the first cell of the following line, which prevents the application of a conventional wavefront approach without additional effort. Fig. 2(c) shows a possible scenario where the processed matrix cells are shown in grey.

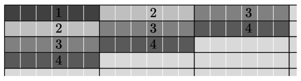
III. Parallel Programming Framework

The focus of the presented work was the development of a parallel programming framework sufficiently generic to cope with a wide set of different DP sequence alignment algorithms to be implemented in current homogeneous multi-core architectures. By analyzing what the several different sequence alignment algorithms that have been proposed [11] have in common, it is clear that most sequential implementations compute the DP matrix cells by iterating over the columns for each line. Also, all algorithms use some specific equation to compute the value of the DP matrix cell from its previous neighbors.

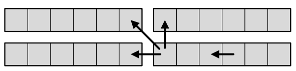
The presented approach started by creating a model that provides a minimum common denominator for most algorithms, abstracting the differences between them. Such common model assumes that all sequence alignment algorithms have at least the same dependencies as the SW algorithm.

A common way to parallelize algorithms with such a data dependencies pattern is to simultaneously compute the cells in the matrix anti-diagonals, by following a wavefront approach. Nevertheless, to improve the efficiency of inter-core communications, the proposed scheme introduces the application of traditional tiling techniques to this class of algorithms, by grouping together the neighboring cells in *chunks/tiles*. The chunks in each anti-diagonal are then processed in parallel, as illustrated in Fig. 2(a). Fig. 2(b) shows, in detail, some of the data dependencies that have to be dealt with: a chunk can only be processed when both the left (in the same line) and the directly above neighbor chunks are completely processed.

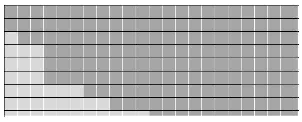
All distributed computations of the DP matrix cells are done in private copies of the data. This allows the framework



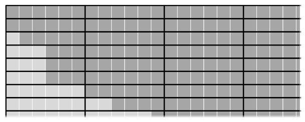
(a) The numbers define the relative order in which chunks are processed.



(b) Data dependencies between neighboring chunks.



(c) Processed DP matrix cells are shown in darker tone.



(d) Dynamic adaptation of the first-column index.

Figure 2. Proposed application of traditional tiling techniques in sequence alignment algorithms.

to apply speculative wavefront processing techniques (commonly applied in Software Transactional Memory (STM) schemes and similar to those presented in [12]) to the computation of the chunks, even when such chunks can not be computed at that time due to unresolved data dependencies. With such technique, the component responsible for controlling the parallel execution speculatively issues the computation of as many chunks as possible, maintaining a window of W lines to limit the amount of memory occupied by the outstanding chunks. Whenever a chunk is computed, it tries to issue the computation of both the chunk directly below and the chunk directly following it (on the right). Because all computations are done in private copies of the data, their effect on the running state of the algorithm can be selectively applied when it is possible to guarantee that such computations are correct. Hence, a speculatively executed chunk is said to be *valid* if its results are correct. When such situation occurs, it is possible to permanently change the running *state* of the algorithm, to reflect the computation of that chunk, in an operation denominated as *committing* a chunk. Although the chunks processing tasks are issued using the wavefront technique, the chunks are *validated* and *committed* in sequential order, i.e. a chunk is only *committed* if it is *valid* and all chunks from the previous lines were *committed*. Both the *validation* and the *commit* operations, as well as the algorithm *state* are only dependent on the sequence alignment algorithm that is being implemented.

The performance of the described scheme depends on two crucial factors: (*i*) the considered size of the chunk: smaller chunks imply greater coordination overhead but increase parallelization opportunities; as a rule of thumb, an equal partition of the line length (N) into the p available CPUs will be applied: $\text{Chunk Size} \approx N/p$; (*ii*) the amount of extra dependencies the algorithm has: if none of the speculatively executed chunks can be used, the running time will be equal to or slightly greater than that of the sequential algorithm.

From the perspective of the framework user, implementing a new parallel sequence alignment algorithm is a matter of implementing a set of functions with a very specific interface and semantics. Such functions belong to an abstract data type called `SequenceAlignmentAlgorithm`. In Fig. 3 it is illustrated the application of this framework in the processing of a chunk C from its neighbors A and B. The diagonal dependency presented by the first cell of C can be satisfied if, when processing A, the last value of this chunk is copied into A's output structure before updating chunk A.

A. *SequenceAlignmentAlgorithm*

The most important operations are the following:

- `process()`: Processes a chunk. Its single argument is an instance of `SeqAlignAlgInput`, which encapsulates all the needed data. Its result is an instance of `SeqAlignAlgOutput`, which encapsulates all the data resulting from the processing of the chunk.

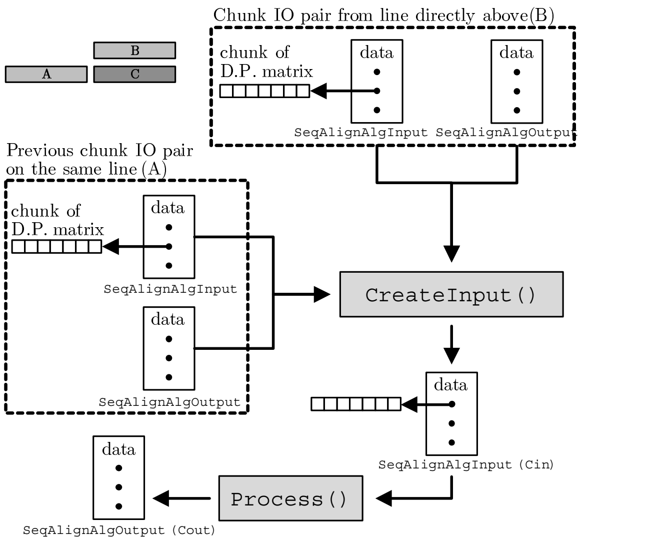


Figure 3. Chunk C processing using the data from neighbors A & B.

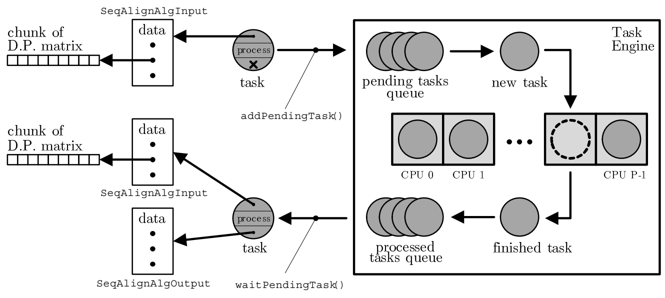


Figure 4. During the processing of a chunk, the *Tasks* use as input the data structures given from the `createInput()` function.

- `createInput()`: Creates the input structure containing all the needed data. The arguments are an *IO pair* associated with the previous chunk in the same line and the *IO pair* for the chunk above (if they exist).
- `isValid()`: Responsible for validating a processed chunk. The validation should be done using the information contained in the *IO pair* for that chunk (received as argument) and the information contained at that moment in the algorithm *state*.
- `commit()`: The *commit* function takes the information contained in the *IO pair* of a specific chunk and permanently changes the *state* of the algorithm.

B. Parallelizer

This component is responsible for controlling the parallel execution of the algorithm and for calling the `SequenceAlignmentAlgorithm` functions. All these functions are called in mutual exclusion, except for the `process()` function, which is concurrently called by the set of worker threads that incorporate a thread pool, controlled by the *Parallelizer*. The *Parallelizer* executes in its own thread, issuing the wavefront execution of the chunks and validating and committing them in sequential order. To issue a chunk, this component creates a *Task* entity, defined as a tuple $\{$*input data*; *processing function*$\}$, and passes it to the *TaskEngine*. The *TaskEngine* maintains a pool of worker threads that continuously grab a *Task* from the pending tasks queue, execute the processing function with that task's input data and save the resulting output in the same *Task* structure. Fig. 4 illustrates how this mechanism is integrated with the *SequenceAlignmentAlgorithm*. It has the following interface:

- `addPendingTask()`: Adds a task to the pending tasks queue. The execution of a *Task* produces an output data.
- `waitPendingTask()`: Blocks the execution of the caller until any issued *Task* is concluded.

The presented framework was fully implemented in C using the *pthreads* library, so that the most lower-level task management mechanisms could be efficiently used. One example of such mechanisms is the selective and preemptive termination of the processing of a given thread (coresponding to a chunk whose input data is later known to be incorrect).

IV. Case Studies

To demonstrate the application of the proposed framework, two different DP sequence alignment algorithms were implemented: the traditional SW algorithm, described in section II-B1, and the sub-optimal algorithm based on reduced matrix optimization heuristics, similar to the one described in section II-B2.

A. Smith-Waterman Algorithm

With respect to the proposed model, the SW algorithm does not have any additional data dependencies. As such, since the chunk computation is never invalidated, in the implementation of this algorithm the input structure for the processing of a given chunk contains a pointer to the corresponding tile of the DP matrix, instead of a local copy of the same data. It also contains the coordinate (in the DP matrix) of that chunk, needed to correctly address the sequences and to compute the corresponding match score.

The `process()` function iterates over each chunk in the input structure, updating every cell. At the same time, it also maintains the best *local score* found in that chunk, as well as the corresponding traceback data. At the end, the obtained values are copied into the output structure.

The global *state* of this algorithm contains the *global best score* and the corresponding traceback information. The *commit* operation updates the global best score, according to the value of the local best score for each committed chunk, and adds the traceback information corresponding to that chunk to the global traceback information.

B. Reduced Matrix Optimization

The algorithm described in section II-B2 presents two additional data dependencies: the *best score*, which is read at every cell, and the *initial column* considered for each line.

In the implementation of this algorithm, the `SequenceAlignmentAlgorithm` functions, as well as their input/output structures, were properly adapted in order to address the following aspects: the input structure contains the *column index* corresponding to the first cell of that chunk (see Fig. 2(d)) as well as the *global best score*, considered when such structure was created. This makes it possible to decide if a speculatively processed chunk used the correct values for these parameters. Hence, the `isValid()` function will only validate those chunks that adopted, at the moment they were processed, the correct values for the first column index and best score.

The implemented algorithm does not preclude any aggressive approaches where, for instance, a chunk that used incorrect values could still be considered *valid* if such errors were constrained in certain limits.

V. Experimental Results

To evaluate the performance of the programmed algorithms on the proposed parallel framework, two different platforms based on multi-core processors were considered:

- Intel Core 2 Quad-Core Q9550 running at 2.83GHz, with two unified 6 MB L2 caches and 4 GB RAM;
- Sun SPARC Enterprise T5120 Server, equipped with an UltraSPARC T2 processor (Niagara) running at 1.6 GHz and with 128 GB RAM. This processor is composed by eight CPU cores with a 4 MB integrated L2 cache. Each core is able to handle eight concurrent threads, making this processor capable of executing up to 64 concurrent threads.

To conduct this assessment, a representative set of real DNA sequences was considered, characterized by a broad range of different lengths (see Table I). Due to its universal application in most toolboxes that guarantee the optimal alignment, the SW algorithm was used in the characterization of the framework regarding its configuration parameters, such as the considered chunk size. After the best parameters were determined, the speedup ranges provided by the proposed framework in the implementation of the two considered algorithms were evaluated.

A. Evaluation of the Optimal Chunk Size

The top chart of Fig. 5 represents the variation of the processing time to align sequence S4 against S2 with the SW algorithm in Intel Core 2 Quad platform, by considering several different sizes of the chunk/tile. The obtained values clearly show that there is a delimited range of this parameter that leads to the best results. As it was predicted before, such range is defined around the size corresponding to a balanced partition of the line length (N) by the p available workers (cores) of the processor: $\text{Chunk Size} \approx N/p$. Entirely similar results were obtained by considering the remaining sequences of the adopted sequence set.

The bottom chart of Fig. 5 represents the variation of the speedup with the chunk size, obtained with Niagara platform for a fixed number of workers ($p = 8$). As before, the best results were reached when the size of each tile is close to N/p, corresponding to the best compromise between the processing time required for the score calculation of each chunk (by each worker) and the overhead imposed by the master thread to coordinate the parallelization (*Parallelizer*).

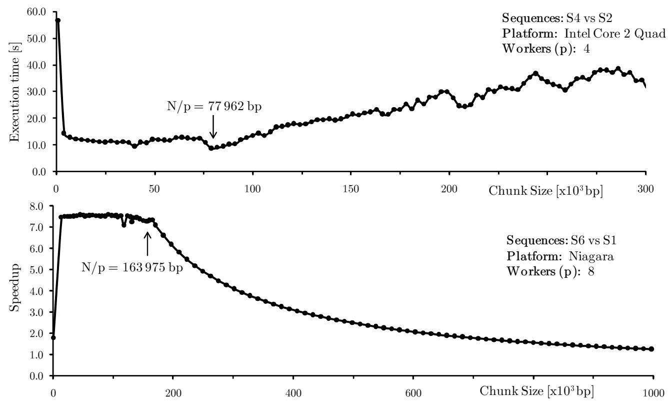


Figure 5. Variation of the execution time and of the obtained speedup with the size of the chunk that is assigned to each worker.

B. Performance Evaluation

The top chart of Fig. 6 plots the variation of the speedup achieved with the execution of the SW algorithm in Intel Core 2 Quad platform, when the number of workers is varied between 1 and 4. The obtained results show that this algorithm, when implemented with the proposed parallel framework, scales linearly with the number of workers. In fact, the real speedup is quite close to the optimal theoretical speedup, thus resulting in an efficiency greater than 88%.

Similar results were obtained with the Niagara platform when the number of workers is varied between 1 and 64, as presented in the bottom chart of Fig. 6. Contrasting to what happens in the Intel platform, this chart clearly defines two regions with distinct characteristics. When the number of workers is not greater than the total number of cores available at the processor (8), the obtained speedup presents a variation entirely similar to the one that was observed with the Intel platform, with an efficiency greater than 89% and a linear scaling with the number of workers. As soon as the number of workers increases beyond the number of available cores, more than one single thread is executed at each CPU, leading to a clear degradation of the scaling but

Table I
Considered DNA sequence set[1].

Sequence	S1	S2	S3	S4	S5	S6
Length [bp]	2230	25410	212160	311850	634690	1311800

[1]GenBank [1]: *ref—NC_000002.11—:121095352-122407052 Homo sapiens chromosome 2, GRCh37 primary reference assembly.*

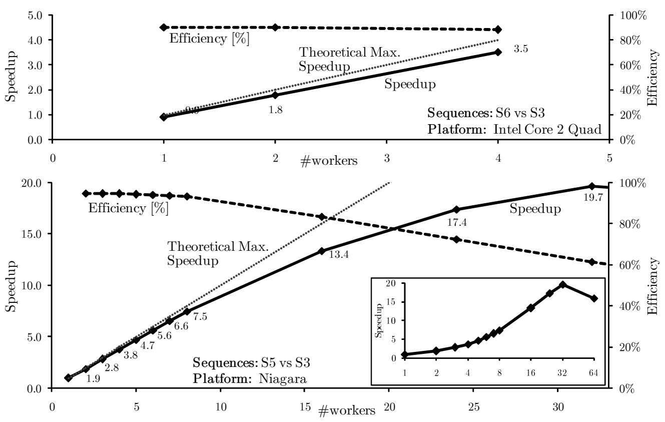


Figure 6. Resulting speedup obtained for the parallel execution of the SW algorithm with the considered platforms.

still providing a moderate increase of the speedup. Such expected degradation arises not only from a significative increase of structural conflicts between the several threads that concurrently execute at each CPU, but also from a presumably increase of invalidations at the shared cache levels.

In the same trend, the implementation of the considered sub-optimal algorithm based on reduced matrix optimization heuristics by using the proposed parallel framework has shown to provide quite satisfactory results. Contrasting with the SW algorithm, the fewer amount of computations resulting from the adopted reduced matrix optimization, as well as the inherently irregular data dependencies presented by this faster algorithm imposes more strict difficulties to reach the theoretical maximum speedup. Even so, the conducted experiments have shown the ability to obtain speedups of about 1.3 and 1.7 when the algorithm was implemented using 2 and 4 cores of the Intel platform, respectively.

VI. Conclusion

A new parallel programming framework for DNA sequence alignment in homogeneous multi-core processor architectures was proposed. This framework is based on the application of tiling techniques that divide the query sequence and each database sequence in smaller chunks that are concurrently processed by the several CPUs. The implementation of the presented framework was conducted in order to confer a highly flexible and generic nature, allowing it to cope with a wide set of different DP sequence alignment algorithms. Besides the classical and regular algorithms (e.g. SW and NW), the framework was designed to also support a broad range of sub-optimal heuristics with additional and irregular data dependencies that would, otherwise, be very hard to implement in parallel architectures.

The conducted experimental procedures have proven that significant accelerations of the considered sequence alignment algorithms can be achieved. While for the optimal SW algorithm the observed speedup is linear with the number of available processing cores and very close to the theoretical maximum, for the faster and sub-optimal method the proposed platform has proved to be capable of still providing a further reduction of the inherent processing time.

Acknowledgment

The authors would like to thank SUN Portugal for granting access to the Sun SPARC Enterprise T5120 Server used in the experimental procedures.

References

[1] D. Benson, I. Karsch-Mizrachi, D. Lipman, J. Ostell, and E. Sayers, "GenBank," *Nucleic Acids Research*, vol. 37, pp. D26–D31, Jan. 2009.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[3] W. Pearson and D. Lipman, "Improved tools for biological sequence comparison," *Proc. Nat. Acad. Sciences of the U.S.A.*, vol. 85, no. 8, pp. 2444–2448, 1988.

[4] K. Benkrid, Y. Liu, and A. Benkrid, "A highly parameterized and efficient FPGA-based skeleton for pairwise biological sequence alignment," *Very Large Scale Integration (VLSI) Systems, IEEE Trans. on*, vol. 17, no. 4, pp. 561–570, April 2009.

[5] T. Oliver, B. Schmidt, and D. Maskell, "Reconfigurable architectures for bio-sequence database scanning on FPGAs," *Circuits and Systems II: Express Briefs, IEEE Trans. on*, vol. 52, no. 12, pp. 851–855, Dec. 2005.

[6] A. Darling, L. Carey, and W. Feng, "The design, implementation, and evaluation of mpiBLAST," in *Int. Conf. on Linux Clusters: The HPC Revolution*, 2003.

[7] M. Farrar, "Striped Smith-Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[8] X. Meng and V. Chaudhary, "Exploiting multi-level parallelism for homology search using general purpose processors," in *Parallel and Distributed Systems. Proc. Int. Conf. on*, vol. 2, Jul. 2005, pp. 331–335.

[9] Y. Liu, D. Maskell, and B. Schmidt, "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units," *BMC Research Notes*, vol. 2, no. 1, p. 73, 2009.

[10] L. Ligowski and W. Rudnicki, "An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases," in *Parallel & Distributed Processing (IPDPS). IEEE Int. Symp.*, May 2009, pp. 1–8.

[11] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: computer science and computational biology*. Cambridge Univ. Press, 2007, ch. 11, pp. 215–253.

[12] A. F. Donaldson, C. Riley, A. Lokhmotov, and A. Cook, "Auto-parallelisation of sieve C++ programs," in *Euro-Par Workshops*, 2007, pp. 18–27.