Profiling biological applications for parallel implementation in multicore computers

Pedro Matos Geraldes Monteiro
Instituto Superior Técnico
Av. Rovisco Pais, 1049-001 Lisboa, Portugal
pedro.m.monteiro@ist.utl.pt

Abstract—Biological sequence databases keep growing every day, which represents a constant challenge in the context of the hardware and software architectures specially developed for high performance sequence alignment procedures. The main objective of this work is to propose a set of parallelization models for a restricted set of dynamic programming sequence alignment algorithms. A simulation environment is also proposed, making it possible to study the solution’s behavior on a common GPP multi-core architecture. The Smith-Waterman algorithm was implemented in the context of the presented framework. The proposed parallelization model, based both on fine and coarse-grained parallelization approaches, obtained a rather flat performance curve in several different platforms, scaling almost linearly with the number of cores in the architecture. The algorithm’s parallel implementation reached speedup values of over 480 times faster than the vanilla version of the same algorithm, which corresponds to a performance of more than 77 GCUPS obtained by using 32 parallel worker implementations typically adopt a master/worker approach, where a single processor acts as master and the remaining processors as workers.

This type of solution is generally adopted in the scope of large clusters. Several workstations are typically connected by an Ethernet network, which usually supports an MPI library based implementation for data exchange between processors and for processing coordination [2][3].

Solutions based on this implementation end up having good results [2]. However, they often present a difficult tradeoff, due to the significant time penalty imposed by the communication stages required to transfer data among several distributed nodes [3].

B. Fine-grained Parallelism

Fine-grained parallelism refers to a set of solutions that exploit parallelism at a very low computational level: the processor’s instruction level. With the inclusion of SIMD vector instructions in the instruction set of most processor architectures, several parallel sequence alignment implementations were proposed. These solutions were typically based on current GPPs [4][5][6][7].

Fine-grained parallelism mostly concentrates on parallelization within a single alignment, which is known as intra-task parallelization. The opposite approach is know as inter-task parallelization, where multiple database sequences are processed simultaneously.

The main advantage of an inter-task parallelization approach, is that it simply avoids all data dependencies within the alignment matrix [8]. This approach doesn’t seem to
have been explored much until T. Rognes presented his latest work [7].

The aim of his study was to explore the use of an inter-task approach, using SIMD on ordinary CPUs. His model is implemented on common Intel processors with SSSE3 [9], using parallelization over multiple database sequences. Basically, instead of aligning a single database sequence against the query sequence at a time, four residues from up to sixteen different database sequences are retrieved and processed in parallel. Rapid extraction and organization of data from the database reduce the number of memory accesses needed, making this approach feasible. The described model has been implemented in a tool called SWIPE.

The speed of SWIPE was heavily tested on a dual Intel Xeon X5650 six-core processor system, using a wide range of query lengths. The tool proved to be about six times faster than Farrar’s own solution, which runs on an heterogeneous multi-core system based on Cell’s architecture [10]. SWIPE achieved performances of over 9 BCUPS for a single thread and up to 106 BCUPS using 24 parallel threads.

C. Intermediate-grained Parallelism

With the advent and widespread availability of multicore architectures based on GPPs or on other heterogeneous designs on today’s computers, a new intermediate-grained parallelism level became worth of exploiting. This intermediate level corresponds to thread-level parallelism, which allows an algorithm to run concurrently in the context of several threads or processes.

Farrar proposed a version of the Smith-Waterman algorithm based on the Cell Broadband Engine heterogeneous architecture [10]. The solution is based on a multi-threaded approach, where each thread independently handles the alignment of multiple query sequence chunks. Since the implementation is based on Cell’s main architecture, only six of the eight SPEs are made available, which means that up to six independent sequence chunks can be aligned in parallel.

Despite SPEs being extremely fast and efficient in terms of performance, they are strongly limited in terms of memory access. Cell’s SPEs can manage only their own independent cache units. The adapted Smith-Waterman implementation running on a single 3.2 GHz Cell Broadband Engine achieved performance levels over 16 BCUPS [10].

Following a similar thread-level approach, other proposals were also presented by making use of stream processing flows available on a Graphics Processing Unit (GPU) [2][11]. Unfortunately, GPU’s global memory access is extremely limited when compared with its processing capacity. Still, speeds of more than 3.5 BCUPS are achieved on a workstation running two NVidia GeForce 8800 GPUs [11].

T. Rognes also implemented a thread-level processing strategy in his tool, SWIPE, which focus on the distribution of several independent database chunks across multiple threads [7]. The described approach basically changes the way how data is managed, not the way how the algorithm is able to perform the alignment procedure itself, which greatly simplifies this model’s implementation.

II. THREAD-LEVEL SOLUTION

State of the art commodity CPUs contain several processors on a single die, making it possible to run multiple parallel processes on a single machine. Each processor has the ability to run its own independent program, or more specifically, its corresponding process.

However, it is possible that all these processes belong to the execution of a single program. In this case, the term thread is used to describe processes which are able to operate in the context of a single program. Thus, in order to fully take advantage of an MIMD multiprocessor architecture composed by n processors or cores, at least n parallel threads should be considered [1].

Coarse-grained parallel solutions of sequence alignment algorithms typically suggest the possibility to simultaneously align multiple sequences with a single query sequence, by simply distributing several database chunks across different computing nodes. If a generic multicore architecture is considered, it is possible to adapt this model to a thread-level based solution, by considering each one of the available cores as an independent computing node. Thus, communication between different nodes becomes local, which improves scalability and flexibility. However, the model focus its attention on the distribution of several database chunks across different computing nodes, which means that each independent alignment will still depend on the size of the query sequence. Thus, the speed is gradually reduced for very long query sequences [7]. In order to avoid that, the presented solution focus on distributing data across several different nodes, in the context of a single alignment.

A. Wavefront Approach

A. Wozniak presented one of the first parallel implementations for the Smith-Waterman algorithm [4]. His solution was based on a video-oriented instruction set extension named VIS (Visual Instruction Set), which made it possible to process up to four 16-bit integers in parallel. By shifting each matrix row one place to the right, a pipeline processing scheme is created, which means that several independent cells may be processed in parallel.

The improvement that may be achieved by a thread-level parallel solution strongly depends on the way data is partitioned. Multi-core architectures, containing several dedicated processors, typically run a single thread on each of its cores. Thus, completely independent data chunks should be considered in the context of each thread. By following the previously described approach, multiple independent matrix cells are able to be processed simultaneously. Hence, we adapt Wozniak’s model to a thread-level parallel approach,
where multiple cell blocks are considered, instead of single unit cells.

In this context, each block corresponds to the alignment of two sub-sequences, corresponding to fractions of the original full sequences being processed. Thus, each thread processes an independent block, part of the algorithm’s global score matrix.

The sub-sequences must be aligned by a single specific order, due to the dependences beyond diagonal lines, which still remain. Thus, the number of available independent blocks will increase until the main anti-diagonal line is reached, and then decrease until the end of both sequences (Figure 1).

![Figure 1. Wavefront processing approach.](image1)

B. Master/Worker Model

Sharing data between threads running on different cores is inefficient. Thus, completely independent matrix blocks should be considered by each independent thread. However, some dependences remain, which means that a scheduling strategy is necessary in order to guarantee that every block is processed only after its top and left neighbors.

If each block is considered as a sub-part of the algorithm’s global score matrix, then the algorithm’s general equations can be independently applied to it by each thread. Since multiple diagonal lines are considered, only two frontier dependencies will remain for each block. Thus, only two block length arrays are kept in memory for each block.

This information has to be correctly managed to guarantee the correct alignment process. Evidently, this type of management implies handling several data structures. Although these structures are completely independent from the alignment procedure itself, the information they keep should be accessible to every thread.

To keep data independency between each thread, a master/worker model is adopted. By following this approach it is possible to completely separate the wavefront model logic from the alignment procedure itself.

The master process is responsible for the management and delivery of the arrays containing each block’s frontiers. The remaining threads act as workers: they receive both frontiers and apply the alignment algorithm. When the alignment is concluded, both arrays are sent back to the master.

By adopting a straight wavefront processing approach, threads remain blocked until all anti-diagonal line results are obtained. Thus, some of those threads will remain idle, while a set of new independent blocks can already be processed.

The master should be able to trigger any block processing as soon as both dependency frontiers have already been solved by previous worker processing. Thus, every single thread is kept constantly working until the full alignment process is completed (Figure 2).

![Figure 2. Modified wavefront processing approach.](image2)

To get maximum efficiency, the second option should be adopted. However, communication between a single master and several worker threads may represent a problem. If the information is delivered directly to a specific worker, it means that the remaining threads will have to wait for the master to finish its data processing and block delivery.

By considering an indirect block delivery system, it is possible to avoid a bottleneck situation introduced by the master process. The solution is to implement two First In, First Out (FIFO) lists through which the master is able to communicate asynchronously with all the workers and vice versa. Thus, messages containing both frontiers are placed by the master in the first list. A free worker accesses that message, processes the alignment, and places the results on the second list. The master accesses the results in the second list and repeats the whole process for each new block (Figure 3).

![Figure 3. Master/Worker communication model.](image3)

III. Instruction-Level Solution

The described thread-level model focuses on the distribution of independent data blocks across multiple worker threads. In order to achieve that, a minor diagonal approach was followed. Thus, each block becomes an independent subpart of the algorithm’s global score matrix. This means that it adopts an intra-task parallelization method, where parallelism occurs within a single alignment procedure.

An approach where parallelization is carried out across multiple database sequences is possible too. This technique is known as inter-task parallelization. In fact, it is possible to exploit both parallelization techniques by considering an inter-task approach in the context of each worker thread. As a consequence, this section presents an inter-task SIMD solution, which extends the previously presented thread-level parallelization model.
The described solution follows T. Rognes’ approach [7], which uses Intel’s SSSE3 instructions in order to simultaneously process multiple database sequences.

A. SIMD Model Adaptation

By exclusively focusing on the distribution of several database chunks across different nodes, SWIPE’s thread-level processing strategy becomes strongly dependent on the size of the query sequence. Each database sequence is partitioned and aligned along the entire query sequence, which means that each SIMD iteration will directly depend on the query length (see Figure 4 - A). Thus, the speed is gradually reduced for long query sequences when using multiple threads [7].

![Figure 4. Multi-sequence SIMD vectorization approaches for the Smith-Waterman algorithm. For simplicity, vectors of only four elements are shown, while sixteen elements would normally be used. (A) SWIPE, which follows a parallel to the query sequence processing approach, described by T. Rognes [7]. (B) Proposed parallel processing approach, which adopts SWIPE’s SIMD model on multiple thread-level processing blocks.](image)

However, despite the described issue, SWIPE can still achieve extremely high performance levels, reaching up to 9 GCUPS, in the context of a single processor [7]. This is mainly possible due to the SIMD part of the solution, which implements the core procedures of the algorithm itself. In fact, the described problem mostly arises from the absence of an intra-task processing approach. In order to maximize the efficiency and improve the performance, an overall solution should be as complete as possible. Thus, it is desirable to exploit both intra-task and inter-task parallelism in the context of a single solution.

Hence, the proposed approach is to adapt SWIPE’s highly efficient SIMD core to the previously described thread-level distribution model. As a consequence, both intra-task and inter-task parallelization techniques will be considered in the context of a single parallel solution.

First of all, in the presented solution the SWIPE’s core instructions are included in the context of each thread, which means that each worker can handle up to sixteen database sequences at a time. As a consequence, each block now includes up to sixteen independent matrix sub-parts as well.

SWIPE follows a processing approach parallel to the database sequence, which means that the query sequence will remain intact during all the processing procedure. Thus, every column will be processed in the context of a single step. As a consequence, each new processing step will exclusively depend on the previous line. This means that by adopting a parallel minor diagonal approach, each block will only depend on its left and upper frontiers.

In the modified solution, the final scores stored in the $E$ vector are saved in memory, in order to be used in the context of the algorithm’s next processing step. In a same way, in order to adapt Rogne’s solution to this new approach, the final scores in $F$ should be also kept in memory.

All processing vectors, except $N$, are correctly initialized prior to the actual processing code segment. Thus, if the $F$ vector is initialized with the previously computed scores, the values will be then correctly computed by each worker, in the context of an independent block. The same applies to $E$, which already happens in the original SIMD implementation.

In order to achieve that, some parts of the original code segment hand coded by Rognes in inline assembler had to be modified. Some new lines also had to be added in order to store the correct values in memory. However, all these modifications took place outside of the algorithm’s critical code section. Hence, no major overhead is introduced by this new set of instructions.

In the original solution, the residues are directly fed into sixteen independent channels. Then, when the first of these database sequences ends, the first residue of the next database sequence is loaded into the empty channel. The concept of channel does not exist in this new solution. In fact, in the context of a wavefront processing approach, blocks must be aligned by a single specific order. This happens because each single block corresponds to the alignment of several database sub-sequences with a single query sub-sequence, both corresponding to parts of the original full sequences being processed.

As a consequence, each sequence is now considered in the context of a batch, which can support up to sixteen different database sequences. Thus, each batch will depend on the alignment of its longest database sequence. Since the database sequences will typically have very different sizes, the system may end up by spending a greater amount of time on a single larger alignment procedure. This means that up to fifteen processing slots may remain unused for several processing blocks.

This problem could be solved by sorting the whole database by length, which would homogenize the sequences’ size in each batch. However, the sorting procedure would represent an even greater overhead in the overall solution. Then, in order to keep reading database sequences in the
same order as they are found in the original database file and still improve the system’s overall efficiency, several SIMD iterations were included in each thread-level processing block. By following this approach, it is possible to optimize the total amount of blocks per batch, thus minimizing the total number of blocks containing fewer database sequences than possible.

Just like the original SWIPE implementation, the presented solution also uses several bit score ranges. In Rogne’s solution, as soon as a potential overflow is detected, the alignment score for that particular database sequence is recalculated by using the next wider score range available. Thus, if a single potential overflow is detected, the corresponding database sequence is processed alone, which means that up to seven processing slots would remain empty during the necessary computations.

In order to avoid that, the new solution keeps a list, which contains the sequence ID of each detected potential overflow. Then, when the global database processing is over, the alignment for each sequence in that list is calculated again by using the next wider score range. Thus, several profile sequences can be simultaneously aligned in the context of a new batch, by considering a different and wider score range.

Hence, although there are some major differences, SWIPE’s core instructions remain basically the same. The developed work on this area focus on the adaptation of Rogne’s instruction-level solution to a completely new thread-level parallel approach.

In conclusion, it is possible to take full advantage of an highly optimized implementation in the context of a whole new thread-level approach. The result is a new and extremely flexible solution, which exploits completely different techniques and different parallelization levels, in order to achieve the system’s maximum performance.

IV. IMPLEMENTATION

The solution uses two FIFO lists through which the master is able to asynchronously communicate with all the workers and vice versa. The messages are placed by the master in the first list. A free worker accesses that message, processes the alignment, and places the results on the second list. The master accesses the results in the second list and repeats the whole process for each new block (see Figure 3).

Each message represents an independent block, which is considered to be as a sub-part of the algorithm’s global score matrices being processed. Since we are considering multiple anti-diagonal lines, two frontier dependencies (left and top) remain for each alignment, in the context of each block. Thus, each message will contain pointers (addresses) for two large memory arrays, containing the correct values for E and F (see Figure 2). These fields are used by the master, in order to specify the address of the values to be processed by any worker thread, or used by any worker, to return the results back to the master.

These arrays have a fixed size, which determines the dimension of each processing block. However, the type of data that they keep will vary. The computations are initially performed using a 7-bit score range. Hence, the arrays initially hold several byte units. Bytes are implemented using the unsigned char data type. In this case, up to sixteen database sequences are processed in the context of a single algorithm iteration.

When a potential overflow condition is detected in computations using a 7-bit score range, the alignment scores for those particular database sequences are recalculated using a 16-bit score range. In this case, the arrays hold several word units, which are implemented using the unsigned short data type. This means that up to eight different database sequences can be simultaneously processed in the context of a single processing iteration.

Finally, if no instruction-level parallelism is considered, which means that a single cell comparison is processed at a time, each array holds multiple long type elements. The type of data being used is indicated to the workers by a specific field in the message.

In all cases, the exchanged messages include pointers for both the query sequence and the database sequences being aligned. The message also includes their corresponding offsets. This information allows each worker thread to correctly access all the necessary sequences.

If several database sequences are considered in the context of a single block, a mask is used in order to keep track of which alignment slots are still active. Hence, each message includes a small array where each position represents an independent processing slot. All its values are initially set to TRUE, which means that all the slots are available and should be processed in the context of the alignment code segment. Each time a database sequence ends, its corresponding slot value is set to FALSE. Thereafter, those slots should be ignored by the alignment code segment, in order to avoid incorrect score updates.

Partial score values are computed in the context of each independent processing block. These values are sent by each worker to the master thread, which evaluates them in order to keep the alignment’s maximum scores. Thus, the exchanged messages must also include an array containing the partial scores computed in the context of their particular blocks.

Finally, each message also includes a tableID, which basically defines the scope of each block (see Figure 2). This table consists of an array containing multiple structures referred to as border tables.

A border table consists of an array structure keeping a node for each matrix block. Each node contains two FIFO lists. These lists basically keep track of the available frontier dependencies during the entire alignment process (see Figure
Each frontier dependency is calculated during the processing of the neighboring block belonging to the previous anti-diagonal line. Thus, each time a message reaches the master thread, a new set of results is pushed into two different nodes, corresponding to the block’s bottom and right neighbors. Whenever this happens, both node’s lists are checked. If none of them is empty, it means that both frontier dependencies are available. Then, the master is able to create a new message and place it in the outgoing list, where it will wait for processing.

The number of processing structures should be equal to the number of available worker threads in the system. This number usually matches the number of available cores, which means that we end up by having a single border table structure per available core. Each time a complete alignment procedure reaches its end, the default values in its border table are totally restored. Then, a new set of database sequences is loaded and the whole process is repeated using the same structure.

Finally, each time a potential overflow is detected, an ID tag is added to a list. This list contains the sequence ID of all the overflow cases detected during a database alignment process. When the global database processing is over, the alignment for each sequence in that list is recalculated by using the next wider score range. Thus, several profile sequences can be simultaneously aligned, by using a different and wider score range. This list is managed by the master thread, just like all the previously described data structures.

V. SIMULATOR ARCHITECTURE

The described parallel processing model is being developed in the context of a preliminary version of the HELIX project architecture. This architecture conceptually implements a master/worker model, where a single master processor manages a set of independent and dedicated processors, which will act as workers. Each worker processor exclusively processes data in its own local memory unit. Thus, a global memory system will be shared by a single bus being accessed by all the existing worker processors, which will be permanently required for multiple data exchanges between the global shared memory and the worker’s local independent cache units.

In order to study the several aspects of the proposed solution, a set of GPP based multi-core architectures is considered to simulate the described HELIX architecture model. Thus, if a n-core GPP architecture is considered, the n available cores will act as the previously described dedicated worker processors. To ensure that each worker thread remains attached to a single specific core, CPU pinning was enabled.

Processor affinity or CPU pinning techniques take advantage of the fact that some remain of a process or thread are fixed to the processor’s state (more particularly, to its cache) since the last time the process ran [12]. By scheduling each worker to always run in the same processor results in more efficiency, since it allows to minimize part of the performance-degrading situations, such as cache misses. Furthermore, this also better approximates to the original scenario, where each worker represents a real physical core, with exclusive access to its own cache and memory unit. Hence, each independent processing block should be completely computed by a single specific core.

In contrast, the master thread does not need to be attached to any specific physical core. Each available core will actually interleave its operations with the master’s activity. Thus, this particular thread will be executed by all the available cores in the system. Since the load introduced by the master in the system is typically very small, this does not represent any problem.

GPPs implement a considerably different memory model, when compared with the previously described one, in the HELIX architecture. In a GPP based architecture, the global memory is shared by all the available cores, which means that each core is able to directly access it and manipulate it. Thus, in order to correctly simulate the previously described HELIX memory model, memcpy operations are performed by each thread every time some data is required by a worker core. Furthermore, a mutex mechanism is used in order to ensure that a single thread is able to access the global memory data structures at a time, which simulates a common global memory access bus with arbitration (see Figure 6).

Hence, each thread performs its own independent memory access operations at a time. Thus, each worker thread is responsible for reading its corresponding sub-sequences, as well as the two frontier dependencies, on which its values will depend. After the processing, the worker threads must write the resulting dependencies back to the main memory. To correctly implement these operations, some
\textit{master/worker} synchronization mechanisms are also necessary. The described solution considers two FIFO lists through which the \textit{master} is able to asynchronously communicate with all the \textit{workers} and vice versa. This scenario describes a typical \textit{producer/consumer} problem, where one thread (the \textit{producer}) generates data items and another thread (the \textit{consumer}) receives those items and uses them [13]. In this particular case, both types of thread will act as \textit{producers} and \textit{consumers}. Thus, two different \textit{counting semaphores} are used, in order to synchronize the access to both lists.

The first \textit{semaphore} ensures that each \textit{worker} waits until new \textit{messages} are placed by the \textit{master} in the first list. The second one ensures that the \textit{master} thread waits until new results are placed in the second list by any of the available \textit{workers}. As a consequence, each time a new \textit{message} is placed in one of the lists, its corresponding \textit{semaphore} should be posted. Furthermore, two different \textit{mutex} variables are used, in order to ensure that each list is accessed by a single thread at a time.

It is important to note that the original HELIX architecture will not have an operating system installed. The architecture will basically implement a shared-memory system with exclusive and atomic access policies defined by an arbiter [14]. Thus, each one of the previously described synchronization mechanisms will end up by being implemented on hardware.

VI. RESULTS

The performance of the conceived parallel implementation of the Smith-Waterman algorithm was extensively evaluated using different processing parameters. This section presents the evaluation methodology that was followed in order to correctly study the results obtained by the described solution.

The presented model simulates the operation of an heterogeneous computing environment in any generic GPP based computer. Thus, in order to correctly evaluate the performance of the proposed solution, two different multi-core platforms were considered:

- 4-Core Intel i7 950 processor running at 3.07 GHz with six 2 GB DDR3 RAM modules running at 2.0 GHz.
- Dell PowerEdge R810 platform, equipped with four 8-Core Intel Xeon E7-4830 processors running at 2.13 GHz and with thirty two 8 GB DDR3 RAM modules running at 1066 MHz. This system adopts the Intel’s Smart Cache technology [15], which means that each core is equipped with an individual 24 MB L2 cache. Each of these cores is able to handle up to eight concurrent threads, making this processor capable of executing up to 32 SMP threads.

The code was compiled for 64-bit Linux operating system using the Intel C compiler version 12.1. The source code corresponding to Rogne’s SWIPE software was downloaded from the author’s website and compiled using Intel’s compiler as well, as specified in the supplied Makefile [7].

The Zv9 zebrafish genome sequence was used in the tests [16]. Sequences with different lengths were extracted from this sequence and used as queries. These sequences have lengths ranging from 16 to 524 288 amino acid residues, which made it possible to study the solution’s overall performance against multiple different query sizes.

For the database sequences, the \textit{nr.07} database, released on 31 July 2012, was used in all the tests [16]. The \textit{nr/nt} database corresponds to the largest database available through NCBI BLAST. It generally includes all GenBank, PDB (Protein Data Bank) and SwissProt sequences. This specific version of the database consists of 455 432 sequences with a total of 159 236 774 amino acid residues. All these sequences are aligned against a single query sequence in the context of each test.

Due to its universal application in most toolboxes that guarantee an optimal alignment, the Smith-Waterman algorithm was implemented in the context of the presented framework. The described solution supports multiple score matrices and gap penalties. However, all the tests were performed with the BLOSUM62 score matrix and gap open and extension penalties of 11 and 1, respectively, which correspond to the Basic Local Alignment Search Tool (BLAST) default values [17].

The results obtained with the proposed solution were compared with the values obtained using SWIPE [7]. Each alignment was performed using both tools and the obtained results were compared in order to verify that they were the same.

The variation of the algorithm’s performance to align a query sequence against the entire \textit{nr.07} database by considering several different processing block sizes had to be evaluated. For such purpose, a query sequence composed by 16 384 amino acid residues was used in these tests. Figures 7 and 8 present the obtained results.

The Dell PowerEdge R810 processing platform was used for this test, providing a total number of 32 \textit{worker} threads. In this case, it was observed that the block size corresponding to 4 096 x 16 384 amino acid residues should be considered, in order to optimize the system’s performance.
optimal size depends on the total number of worker threads being used. A closer look shows that the query block size remains the same in both architectures. However, the number of database residues per block is doubled. As a consequence, the number of worker iterations per block is also doubled. This happens because the number of worker threads trying to simultaneously access the global memory is considerably smaller. Thus, the risk of collision is considerably lower, making it better to reduce the total number of memory accesses by loading greater amounts of data at a time.

By considering the previously obtained values for the optimal block size in each configuration, the solution was re-evaluated by varying the number of workers. A query sequence composed by 16 384 amino acid residues was used for these tests. Figures 11 and 12 show the obtained results in both architectures.

It is possible to conclude that the proposed implementation scales almost linearly with the number of available worker threads. However, the obtained results should be carefully analyzed.

Firstly, the vanilla version of the algorithm runs on a single processing core, which means that a single cache memory unit is available. On the contrary, in the proposed parallel solution, each worker runs on an independent processing core, which means that if 32 workers are available, then 32 cache memory units are available, one per each available core. This difference ends up by increasing the solution’s overall performance due to the increase in the total number of cache hits per processing operation on a single core. Furthermore, recent computer architectures use multiple cache levels. Hence, each core has its own dedicated
L1 and L2 cache units. However, the L3 cache is usually shared by a set of independent cores. In this case, each processor is able to take advantage of memory loaded into cache by other processing units in the same set, which is impossible when a single worker core is being used.

Secondly, the algorithm is processed in a slightly different way when the SWIPE vectorial parallelization model is adopted, which is the case. As a consequence, the total number of memory accesses is reduced and the cache memory usage is optimized, which increases the algorithm’s overall performance, making it possible to achieve super-linear speedup results.

The obtained speedup should approximately correspond to the value obtained by multiplying the total number of database sequences being simultaneously processed in each SIMD register for the total number of available workers. However, there is still a significant difference, due to the described aspects. Hence, despite some observed variances in the expected values, it is possible to conclude that the described solution scales almost linearly with the number of available workers.

The variation of the achieved speedup was also measured by using different query lengths. The tests were conducted in both processing platforms. Figures 13 and 14 present the obtained results.

As expected, the proposed solution presents highly satisfactory results in both architectures. In fact, it is possible to observe that significantly smaller losses of performance occur for long query sequences, which makes this an highly scalable solution.

The previously described tests were also performed using Rogne’s parallel implementation [7], in order to directly compare both parallelization models.

In fact, besides the inter-task SIMD model, SWIPE also implements a thread based inter-task approach, where multiple threads can simultaneously work on different parts of the sequence database. However, Rogne’s tool adopts a rather simplistic shared-memory architecture. To obtain a comparison as fair as possible, a shared-memory version of the proposed solution was also considered, in order to compare both parallelization models. The tests were conducted by using the Dell PowerEdge R810 processing platform. Figure 15 presents the results for the achieved processing speed (in GCUPS), obtained with the proposed parallelization model, using both shared and distributed-memory architectures (blue and red, respectively), and SWIPE (black).

As it can be observed, SWIPE presents better results for query sequences shorter than approximately 32,768 amino acid residues. However, the attained speed suffers a significant degradation for larger query sequences in a massive multi-threaded environment.

SWIPE adopts an inter-task processing approach focused on the distribution of the database across different parallel computing nodes [7]. In this case, it is possible to simultaneously align multiple database sequences with a single query sequence. However, the size of the query sequence remains the same in the context of each independent alignment procedure. As a consequence, the approach will strongly depend on the query length.

On the contrary, the results obtained by the proposed solution remain constant for very long query sequences. This happens in both shared and distributed-memory architectures because a thread-level intra-task approach is adopted, which makes it possible to circumvent the described problem by splitting both the profile and the query sequences into multiple independent chunks (see Figure ??).

It is also important to note that slightly different results are obtained by implementing the proposed parallel approach on different memory architectures. By observing the obtained results, it is possible to conclude that the shared-memory version is specially faster, when small query sequences are considered. In fact, when a distributed-memory architecture
is adopted, each worker thread has to explicitly copy the data from the global memory system. Thus, the total amount of data loaded in a single memory access has to be optimized in order to avoid collisions. In this particular case, this value corresponds to the optimal size of each independent processing block, which has been studied in the context of several processing platforms.

On the other hand, when a shared-memory architecture model is considered, it is no longer needed to explicitly copy the sequence data from the global memory to each worker’s local memory, thus assuming a traditional transparent data transfer mechanism provided by the memory and cache controller systems. Hence, the obtained results will be more stable for any query sequence length.

To conclude, the shared-memory version of the proposed application attained performance levels of more than 77 GCUPS by using 32 parallel worker threads, which is almost three times faster than the software based on Rognes’ approach for long query sequences. The presented implementation also obtained a performance of more than 71 GCUPS using 32 parallel worker threads on a distributed-memory architecture, which is nearly 2.5 times faster than SWIPE, running on a different memory architecture.

VII. CONCLUSIONS

The main outcome of the presented work is the proposal of the most suited set of parallel models to be applied in the implementation of a set of bioinformatic algorithms in the context of the HELIX architecture.

In the previous sections, a completely new thread-level parallel approach implementation was proposed. An instruction-level model was also proposed, based on Rognes’ SIMD processing approach. The proposed implementation considers both parallelization models, exploring both intratask and inter-task parallelism in the context of a single solution.

A simulation environment was also presented, which allowed the solution to be tested on several common GPP based multicore architectures. The proposed model was tested using the Smith-Waterman algorithm, in order to align a single query sequence against an entire database.

The proposed parallelization model presented a rather flat performance curve in several different platforms, scaling almost linearly with the number of available cores in the architecture. In a 32-core GPP architecture, the algorithm’s parallel SIMD based implementation reached speedup values of over 480 times faster than the vanilla version of the same algorithm.

The achieved performance corresponds to more than 77 GCUPS using 32 parallel worker threads, which is almost three times faster than SWIPE implementation running in the exact same conditions.

The described model is also extremely flexible in terms of architecture, which mean that it can easily be adapted to any platform, pushing its overall performance to the limit.

REFERENCES