PA-Star: A disk-assisted parallel A-Star strategy with locality-sensitive hash for multiple sequence alignment

Article *in* Journal of Parallel and Distributed Computing · May 2017

DOI: 10.1016/j.jpdc.2017.04.014

George Teodoro

Federal University of Minas Gerais

116 PUBLICATIONS **1,042** CITATIONS

SEE PROFILE

Alba Melo

University of Brasília

140 PUBLICATIONS **803** CITATIONS

SEE PROFILE

PA-Star: a Disk-Assisted Parallel A-Star Strategy with Locality-Sensitive Hash for Multiple Sequence Alignment

Daniel Sundfeld[a], Caina Razzolini[a], George Teodoro[a], Azzedine Boukerche[b], Alba Cristina Magalhaes Alves de Melo[a]

[a]*Department of Computer Science, University of Brasilia (UnB), Brazil*
[b]*School of Information Technology and Engineering (SITE), University of Ottawa, Canada*

Abstract

Multiple Sequence Alignment (MSA) is a basic operation in Bioinformatics, used to highlight the similarities among a set of sequences. The MSA problem was proven NP-Hard and it requires a high amount of memory and computing power. This problem can be modeled as a search for the path with minimum cost in a graph and the A-Star algorithm has been adapted to solve it sequentially and in parallel. The main challenges of designing a parallel version for MSA with A-Star are the irregular data dependency pattern and huge memory requirements. In this paper, we propose PA-Star, a locality-sensitive multithreaded strategy based on A-Star which computes optimal MSAs using both RAM memory and disk to store A-Star nodes. The experimental results obtained in 3 different machines show that the optimizations used in PA-Star can achieve a speedup of 1.88× in the serial execution and that the parallel execution can attain a speedup of 5.52× in 8 cores. We also show that PA-Star outperforms a state-of-the-art MSA A-Star based tool, executing up to 4.77× faster. Finally, we show that our disk assisted strategy is able to retrieve the optimal alignment when other tools fail.

Keywords: multiple sequence alignment, locality-sensitive hash, A-Star, parallel algorithms

Email addresses: `sund@unb.br` (Daniel Sundfeld), `cfbrazzolini@gmail.com` (Caina Razzolini), `teodoro@cic.unb.br` (George Teodoro), `boukerch@site.uottawa.ca` (Azzedine Boukerche), `alves@unb.br` (Alba Cristina Magalhaes Alves de Melo)

1. Introduction

Bioinformatics is an interdisciplinary field that involves computer science, biology, mathematics and statistics [1]. One of its main goals is to analyze biological sequence and genome data in order to obtain the function/structure of the sequences as well as evolutionary information.

Multiple Sequence Alignment (MSA) is a basic operation in Bioinformatics in which similar characters among a set of n biological sequences ($n \geq 3$) are aligned together to highlight the similarity among the sequences. MSAs are often used as a building block to solve important and complex problems, such as the definition of phylogenetic relationships and 2D structure prediction, among others. In all these cases, the quality of the solutions relies heavily on the quality of the underlying MSAs.

The MSA problem has been proven NP-Hard [2] and, for this reason, obtaining the optimal solution requires high computing power and huge amount of memory space. There are two main strategies to reduce the search space of the optimal MSA problem. The first one was proposed by Carrillo-Lipman [3] and it defines lower and upper bounds in the search space composed of an n-dimensional dynamic programming matrix, in which n is the number of sequences. The second strategy transforms the MSA problem in the problem of searching the path with minimum cost in a graph and applying the A-Star (A*) algorithm [4] to solve it. In this paper, we employ the A-Star based strategies.

A-Star is a best-first search algorithm that works mainly with two lists (*OpenList* and *ClosedList*) in which the *OpenList* contains the nodes yet to be analyzed and the *ClosedList* contains the nodes which have already been analyzed. A-Star has been used to solve the MSA problem in [5, 6, 7] and, more recently, parallel variants of these solutions have been proposed [8, 9].

Parallelizing A-Star involves two main challenges. Firstly, it presents an irregular data access pattern that requires sophisticated strategies to assign the computation of A-Star nodes to threads. Secondly, it requires a huge amount of memory, which often limits the lengths and number of sequences compared.

The solution proposed in [8] is an MPI based implementation that reduces the use of RAM memory by eliminating the *ClosedList* at the expense of increasing computing time by reprocessing parts of the graph. In [9], a multithreaded solution is proposed where disk and RAM memory are used in order to increase the memory space also at the expense of increased execution time. So far, designing an A-Star based MSA strategy with good performance and reasonable use of memory is still an open problem.

In this paper, we propose and evaluate PA-Star, a parallel solution for the MSA problem based on A-Star. In our solution, we divide the A-Star search space using a hash function with locality preserving characteristic, aiming to reduce the overhead of threads communication and, as a consequence, improve the scalability of the parallelization. In order to deal with the high memory demands of the algorithm, PA-Star saves nodes of the *ClosedList* to disk when the usage of RAM memory is close to the maximum.

The results obtained in three different machines with up to 32 cores and 1TB of RAM memory by comparing real and synthetic sets of sequences show that PA-Star is able to drastically reduce the execution time. We show that our parallel approach is able to outperform one of the state-of-the art solutions [8], with a speedup of 2.89× and 4.77× for the BAliBASE glg and 2ack sets of sequences.

The rest of this paper is organized as follows. We present an overview of the MSA problem in Section 2. Section 3 discusses related work in the area of exact MSA with A-Star. In Section 4, we present the design of PA-Star and the experimental results are shown in Section 5. Finally, we conclude the paper in Section 6.

2. Multiple Sequence Alignment (MSA)

2.1. Overview

Biological sequences are an ordered set of characters, corresponding to DNA, RNA or protein sequences. DNA and RNA sequences are composed of four nucleotides which are, respectively, $\{A, T, C, G\}$ and $\{A, U, G, C\}$. There are 20 amino acids in nature, represented by $\{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$. The protein sequences are composed of these amino acids.

A global Multiple Sequence Alignment (MSA) of $n \geq 3$ sequences $S = S_1, S_2, ..., S_n$ is obtained in such a way that spaces (gaps) are inserted into each of the n sequences so that the resulting sequences have the same length l. Then, the sequences are placed one above the other, arranged in n rows of l columns each [10]. Figure 1 shows an example of an MSA of protein sequences $S_1 = NLFVALYD$, $S_2 = KVIYALWD$ and $S_3 = GYALWDQY$. In this figure, we show that the search space for the alignment of 3 sequences is a cube where the sequences are placed at the x, y and z axes. The alignment can be represented as a path in the graph which connects the upper left corner of the cube (beginning of the sequences) to its bottom right corner (end of the sequences).

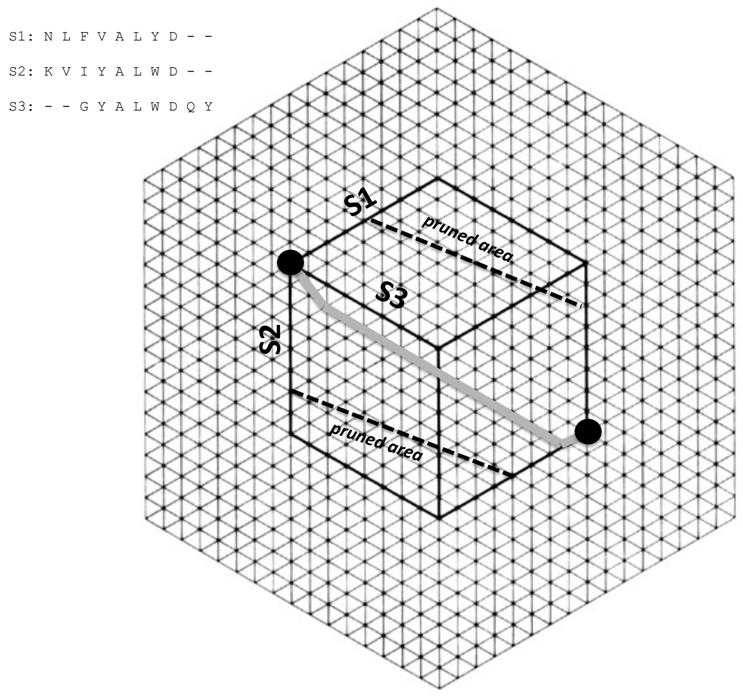


Figure 1: MSA with 3 sequences. The textual alignment is represented at the upper left side and the gray line inside the cube illustrates the alignment.

The goal of an MSA algorithm is to minimize a cost function, generating the alignment with the lowest score. If $n > 2$ sequences are compared, an n-dimensional cube is used. Usually, MSAs are scored with the Sum-of-Pairs (SP) function and the exact SP MSA problem is known to be NP-hard [2]. The SP score indicates the alignment distance (minimum number of insertions, deletions and substitutions) among the sequences. Every pair of bases is scored with the pairwise scoring function and the final score is the addition of the pairwise scores. For instance, consider the protein sequences in Figure 1 and assume that the punctuation for characters in the same column are given by the substitution matrix PAM250 [11] adjusted to calculate the minimum cost [1] and that the penalty for gaps is 24. In this case, the SP score for the alignments would be $122 + 145 + 114 = 381$.

The optimal MSA among n sequences can be calculated by extending the exact dynamic programming (DP) algorithm for pairwise comparison [12]. Without loss of generality, assume that there are 3 sequences S_1, S_2 and S_3, of lengths l_1, l_2 and l_3, respectively. A 3-dimensional DP matrix D is calculated, in which $D_{i,j,k}$ is the optimal alignment of prefixes $S_1[1..i]$, $S_2[1..j]$ and $S_3[1..k]$.

Algorithm 1 Naïve_MSA_3seq (S_1, S_2, S_3)

```
1: for $i = 1 \to l_1$ do
2:     for $j = 1 \to l_2$ do
3:         for $k = 1 \to l_3$ do
4:             $c_{ij} = score(S_1(i), S_2(j))$;
5:             $c_{ik} = score(S_1(i), S_3(k))$;
6:             $c_{jk} = score(S_2(k), S_3(j))$;
7:             $d_1 = D(i-1, j-1, k-1) + c_{ij} + c_{ik} + c_{jk}$
8:             $d_2 = D(i-1, j-1, k) + c_{ij} + gap$
9:             $d_3 = D(i-1, j, k-1) + c_{ik} + gap$
10:            $d_4 = D(i, j-1, k-1) + c_{jk} + gap$
11:            $d_5 = D(i-1, j, k) + 2 * gap$
12:            $d_6 = D(i, j-1, k) + 2 * gap$
13:            $d_7 = D(i, j, k-1) + 2 * gap$
14:            $D(i, j, k) = Min[d_1, d_2, d_3, d_4, d_5, d_6, d_7]$
15:        end for
16:    end for
17: end for
18: return $D(l_1, l_2, l_3)$
```

The naïve algorithm that computes the optimal MSA score is depicted in Algorithm 1. There are three *for* loops (lines 1 to 3), one loop for each sequence. The scores for matches/mismatches are calculated in lines 4 to 6. After that (lines 7 to 13), 7 values are calculated which correspond to the 7 neighbor cells of $D_{i,j,k}$. In line 14, $D_{i,j,k}$ receives the minimum value of those calculated in lines 7 to 13. If there are n sequences to be compared, there will be n loops in this algorithm and $2^n - 1$ cells will be used to calculate each cell of matrix D. At the end of the computation, the optimal MSA score is the value of cell $D(l_1, l_2, l_3)$. In the naïve algorithm, all cells of D are

[1]Value 17 is added to each cell as in the MSA 2.1 tool (*xylian.igh.cnrs.fr/msa/msa.html*)

computed.

In the literature, that are some proposals which reduce the number of cells calculated to obtain the optimal MSA. The most popular ones are Carrillo-Lipman and A-Star.

2.2. Carrillo-Lipman (CL)

Carrillo-Lipman (CL) [3] defined a lower and an upper bound to confine the region which contains the optimal alignment and, thus, to restrict the area of the n-dimensional matrix to be calculated.

The lower and upper bounds have a value that is based on the projection of a heuristic alignment subtracted from the scores of the pairwise alignments. It is guaranteed that cells outside those bounds do not contribute to the calculation of the optimal MSA. Carrillo-Lipman [3] have proven that the sum of all pairwise optimal alignments is a lower bound and that an heuristic alignment can be used as an upper bound to the optimal MSA. Since the optimal alignment is confined in the region defined by the upper and lower bounds, the number of cells computed can be reduced. In figure 1, the pruned area would be defined by the upper and lower CL bounds.

2.3. A-Star (A)*

In the A-Star (A*) [4] approach, the MSA problem is modeled as a graph and the problem of finding the optimal MSA is equivalent to the problem of finding the shortest path in this graph between the initial Node with coordinate $N_i = (0, 0, ..., 0)$ and the final Node with coordinate $N_f = (l_1, l_2, ..., l_n)$, where l_i is the length of the sequence i, and n is the number of sequences.

To reduce the search space, an admissible heuristic function is used. In the A-Star algorithm, the cost function f (Equation 1) describes the order in which the nodes should be visited, while searching for the optimal result.

$$f(N) = g(N) + h(N) \tag{1}$$

In Equation 1, $g(N)$ is the cost from the origin to node N, $h(N)$ is an estimated cost from node N to the final node, and $f(N)$ is the estimated optimal cost through N.

The A-Star algorithm is shown in Algorithm 2. The *OpenList* contains the nodes to be expanded, and the *ClosedList* contains the nodes that have been expanded. Initially, the *OpenList* contains only the initial node N_i (line 1), and the *ClosedList* is empty. In the **find** step, the node with the best f value is selected (line 2). Next, in the **stop** step, if the selected node is in

Algorithm 2 *A-Star(N_i, N_f)*

```
1: $OpenList \leftarrow N_i$
2: while $OpenList.lowest\_f() \notin N_f$ and $OpenList \neq \emptyset$ do
3:     $current \leftarrow OpenList.get\_lowest\_f()$
4:     if $current.g \leq ClosedList.find\_g(current)$ then
5:         $ClosedList \leftarrow ClosedList \cup current$
6:         for $neighbor$ in $neigh(current)$ do
7:             if $neighbor.g < OpenList.find\_g(neighbor)$ and $neighbor.g < ClosedList.find\_g(neighbor)$ then
8:                 $OpenList \leftarrow OpenList \cup neighbor$
9:             end if
10:        end for
11:    end if
12: end while
13: if $OpenList = \emptyset$ then
14:     $n_e \leftarrow NULL$
15: else
16:     $n_e \leftarrow OpenList.get\_lowest\_f()$
17: end if
18: return $n_e$
```

the N_f set, or if this node does not exist because the *OpenList* is empty, the algorithm stops (line 2). Otherwise, the node with best f value is removed from the *OpenList* (line 3). If a better node has been previously found (line 4), the current node cannot be part of the optimal path and it is discarded. Otherwise this node is moved to the *ClosedList* (line 5). In **expand and reconciliation** step (lines 6-10), all neighbors from the node are added to the *OpenList* if another node with better f was not found yet. Further, the algorithm continues the search process until the stop condition is true.

When the algorithm stops, if the *OpenList* is empty, it does not exist a path from N_i to N_f (line 14). Otherwise, the node with the best $f(N_f)$, has a g value that is the optimal cost from N_i to N_f, and it is possible to find the best path using a backtrace function in the *ClosedList*. In Figure 1, the pruned area would be defined by the way the nodes are expanded in the A-Star algorithm.

Spouge [5] showed that the Carrillo-Lipman bound and the A-Star algorithm are closely related by demonstrating that the Carrillo-Lipman lower bound is an admissible heuristic for the $h(n)$ cost of the A-Star algorithm. This heuristic is often called $h_{2,all}$ heuristic.

In order to apply the A-Star algorithm to the MSA problem, we assume that the $h_{2,all}$ heuristic will be used. In order to do that, the reverse of the n sequences are first pairwise aligned with the Needleman-Wunsh algorithm [12]. For instance, if three sequences S_1, S_2 and S_3 are compared, 3 optimal pairwise alignments are generated for the following pairs of sequences: (S_1, S_2), (S_1, S_3) and (S_2, S_3) and the entire dynamic programming matrix is kept in memory for each pairwise alignment. To calculate $h(n)$ for node with coordinates x, y, z, we add the matrix cells (x, y), (x, z) and (y, z) from the (S_1, S_2), (S_1, S_3) and (S_2, S_3) matrices. Then, function $g(N)$ is calculated with the SP function, representing the cost from the origin node to the node N being evaluated and function $f(N)$ is the addition of both functions (Equation 1).

3. Related Work

There are many works in the literature that treat the MSA problem with a graph formulation and solve it using the A-Star algorithm [5] [6] [7] [13] [14]. However, few works aim to solve MSA using A-Star with a parallel strategy.

PE2A* was proposed by [9]. It is a multithreaded strategy that uses external memory (disk) to augment the memory space that can be used by the algorithm. PE2A* combines two previously proposed techniques: Partial Expansion A* (PEA*) [15] and Hash Based Delayed Duplicate Detection (HBDDD)[16]. PEA* does not place all successor nodes in the *OpenList*. Instead, it prunes successor nodes which do not appear to be promising thus potentially accelerating the computation. In this case, however, additional runtime may be necessary due to node re-expansion. HBDDD places newly expanded nodes of the *OpenList* into several disk files and processes them later, using a hash function to allocate the files to the processing nodes. In a machine with 12 cores, PE2A* was capable of solving one of the hardest sets of sequences in BAliBASE [17] reference set (arp) 3.27× faster than the multithreaded strategy that uses only DDD [18].

Niewiadomski et. al. [8] proposed PFA*-DDD, a parallel strategy using Delayed Duplicate Detection (DDD) and Frontier-Search [19], which is a workload distribution strategy based on intervals. Since Frontier-Search does not use a *ClosedList*, the authors proposed a divide-and-conquer strategy to retrieve the alignment. This strategy creates $3n$ processes, each one executing a different step in A-Star. In the experimental results, a cluster of 32 dual-core machines was used, each one with 3 processes. PFA*-DDD was able to solve one of the hardest sets of instances in BAliBASE reference set 1 (gal4) 21.0× faster than the single node strategy, taking 1 hour and 10 minutes (64 cores). Since this is one of the most efficient and scalable solutions to the problem and its code has been made available to us by the authors, we have compared A-Star to PFA*-DDD in Section 5.

4. Design of Parallel A-Star

4.1. Overview

PA-Star is a multithreaded A-Star tool which retrieves optimal MSAs using augmented memory space. The goals of PA-Star are twofold. First, it aims to reduce the execution time by using a locality-sensitive hash function to assign A-Star nodes to threads. Second, it aims to solve difficult sets of sequences which require a huge amount of memory by combining RAM memory and disk. These two goals are in conflict with each other since the use of disk will clearly reduce the performance of the solution. Therefore, if there is enough RAM memory, PA-Star does not use disk. Nevertheless, if

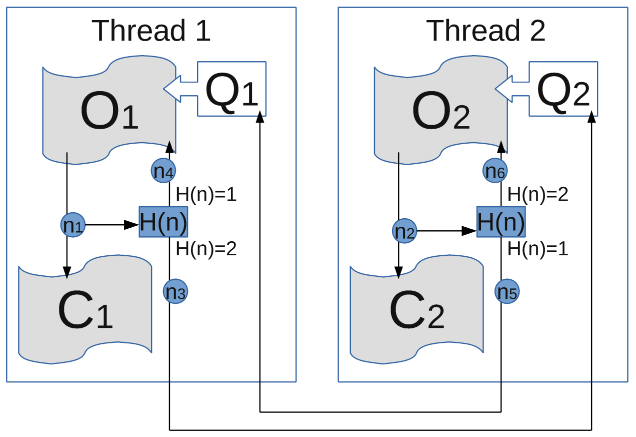


Figure 2: PA-Star Design with 2 threads. Each thread i has an *OpenList* (O_i), a *ClosedList* (C_i) and a queue (Q_i). The threads expand the nodes in parallel (n_1 and n_2), removing nodes from its OpenList and adding them in its ClosedList. During the expansion, the locality sensitive hash function (H) (Equation 2) is used to determine in which *OpenList* the neighbor node should be inserted. Nodes that belong to the same thread, are directly inserted in the thread's *OpenList* whereas nodes that belong to thread j are written in queue Q_j.

PA-Star notices that it will run out of RAM memory, it starts to use both RAM memory and disk.

In PA-Star, the nodes contained in the *OpenList* are expanded in parallel and the A-Star lists are managed locally by t threads. Therefore, each thread t_i has its own $OpenList_i$, $ClosedList_i$ and a queue q_i (Figure 2). The nodes that compose the search space are distributed among the threads using a locality-sensitive hash function.

Every node in the lists has a coordinate that uniquely identifies it. The initial node is $N_i = (0, 0, ..., 0)$ and the final node is $N_f = (l_1, l_2, ..., l_n)$. The mapping between the lists of thread t_i ($OpenList_i$ and $ClosedList_i$) and the nodes is computed by Equation 2. In this equation, H is a hash function, $(N_1, N_2, ..., N_n)$ are the node coordinates, S is a constant shift-right factor, t is the total number of threads and t_i is the thread id. The mod operation is used to map the hash result to $0 \leq t_i < t$. Final nodes are an exception for this rule and are written in all *OpenLists*, as explained at the end of this section.

$$(H(N_1, N_2, ..., N_k) \gg S) \bmod t = t_i \tag{2}$$

If thread t_i expands node k, it is desirable that node k is added to $OpenList_i$, reducing the overhead of communication and providing a better use of the caches. To achieve this, we propose the use of (a) hash functions with locality preserving characteristic, in order to map to the same thread the neighbors of the expanded node, and (b) shift-right operations, where the least significant bits of the hash are discarded. In this work, we used two H functions: the *SUM* function, that simply adds all numbers in the coordinate; and the *ZORDER* function, which is based on Z-Order curves [20]. The *ZORDER* function was selected because it preserves locality and is inexpensive, since it consists of shift operations to interleave the bits of numbers in the coordinate values.

4.2. PA-Star Algorithm

As shown in Algorithm 3, PA-Star is divided in two alternate steps, executed in parallel: *search_step* and *verify_end_condition* (lines 6, 7). *Search_step* implements Algorithm 2 lines 3 to 11 and *verify_end_condition* implements line 2 in the same algorithm.

PA-Star executes as follows. First, thread t_0 executes lines 1 to 4 (Algorithm 3). The hash function H is applied to the initial node N_i and the thread id h is obtained (Line 1). The initial node N_i is placed in the OpenList of thread h (Line 2). Then, the end condition and the value of the cost function f for the final node (n) are initialized as false and ∞ (lines 3 and 4). Further, t threads execute in parallel the two steps of the algorithm (lines 5 to 8).

Algorithm 3 *PA-Star*(N_i, N_f)

```
1: $h \leftarrow hash(N_i)$
2: $OpenList_h \leftarrow N_i$
3: $end\_condition \leftarrow false$
4: $n \leftarrow \infty$
5: parallel_do
6:     $n \leftarrow search\_step(t_i, N_f)$
7:     $end\_condition \leftarrow verify\_end\_condition(t_i)$
8: while $end\_condition == false$
9: return $n$
```

Algorithm 4 *search_step*(i, N_f)

```
1: $c_t \leftarrow 0$
2: $final\_node \leftarrow \infty$
3: while $c_t < threads\_num$ do
4:     /* beginning of the search step */
5:     $consume\_queue(q_i)$
6:     $current \leftarrow OpenList_i.get\_lowest\_f()$
7:     if $current.g \leq ClosedList_i.find\_g(current)$ then
8:         $ClosedList_i \leftarrow ClosedList_i \cup current$
9:         if $current \in N_f$ then
10:            $c_t++$
11:            $final\_node \leftarrow current$
12:            $process\_final\_node(current, c_t)$
13:        else
14:            for $neighbor$ in $neigh(current)$ do
15:                $h \leftarrow hash(current)$
16:                $q_h \leftarrow q_h \cup current$
17:            end for
18:        end if
19:    end if
20:    /*end of the search step */
21: end while
22: return $final\_node$
```

In the *search_step* (Algorithm 4), all threads execute a modified version of the A-Star algorithm. While the threads counter, c_t, is smaller than the total number of threads (line 3), thread t_i consumes queue q_i, removing all nodes in q_i and adding them in $OpenList_i$ (line 5). After this, the node with lowest f is removed from $OpenList_i$ (line 6).

If the node removed from the list is not final, it is added to the $ClosedList_i$ (line 8) and expanded (lines 14 to 17). During the expand phase, the locality-aware hash function is used to determine in which list the neighbor node should be inserted (line 15). Nodes that belong to thread t_h are inserted in queue q_h.

When the expanded node is a final node N_f (line 9), this node is the optimal result only if it has the lowest f among all the nodes in the *Open Lists* of all threads. We optimized this verification by dividing this end condition test in two phases (asynchronous and synchronous) aiming to reduce the synchronization overhead. In the asynchronous phase, the threads counter, c_t, is increased and the function $process\_final\_node$ (line 12) is called. In this function, if the value of f for the final node is lower than the current lowest value, node N_f and the value of f are inserted in all the other queues.

All threads keep executing the *search_step* and when thread t_i expands node N_f again, this means that N_f has the lowest f value among all nodes in $OpenList_i$, and c_t is incremented (line 10). When c_t is equal to $threads\_num$, all threads have expanded N_f, so this node is returned as a possible result (line 22). But some other nodes with lower f may exist in other queues q_i, or may have been inserted in some *Open List* after the thread has expanded N_f. To guarantee that N_f is the optimal result, all threads move to the next step.

In *verify_end_condition* (Algorithm 3, line 7), the threads do not expand nodes anymore, a shared memory boolean variable b_s is set to true and all threads are synchronized. All threads consume their queues and every thread t_i verifies if $f(N_f) < OpenList_i.lowest\_f()$ is true. If not, b_s is set to false. All threads are synchronized again. After this, if b_s is true, node N_f is the optimal result, the *verify_end_condition* returns true and the algorithm returns N_f (Algorithm 3, line 9). Otherwise, some nodes with lower f do exist in some *OpenList* and they must be expanded before it is possible to guarantee that N_f is the optimal result. To restart the search process, node N_f is inserted in the *OpenList* of the thread that first expanded it, the *verify_step* function returns false and all threads go back to the *search_step*.

In addition, few special conditions must be checked. First, when thread t_i calls *consume_queue* but $OpenList_i$ is empty, it must wait for new nodes in q_i. Another special condition is in the $process\_final\_node$ function. If one thread reaches the final node N_f and $f(N_f)$ is lower that the current lowest f value, it means that a better result was found while other threads were still consuming their *OpenLists*. The verification of N_f as a possible result must be canceled and newest f value must be verified as the optimal result. Thus, c_t is reset to 1, and the newest f value is written in all other queues.

With this two step verification phase, we reduce the overhead by synchronizing threads only in the second phase, i.e., if a possible optimal result is found.

4.3. Design of the OpenList data structure

One of the challenges of the A-Star approach is the design of the *OpenList* data structure. The nodes in the *OpenList* have two different fields that must be used as a key: the f value, represented by one integer, and the coordinates, represented by n integers, where n is the number of sequences. The A-Star algorithm must verify and possibly remove the node with lowest f (Algorithm 2, line 3). This operation is usually implemented with priority queues. It must also find a node using its coordinates (Algorithm 4, line 6), which is commonly done with the aid of hash functions or tables.

Niewiadomski et. al. [8] implemented the *OpenList* as a dictionary, using two data structures: one priority queue and one hash table. The obvious problem of this operation is that the memory used to represent the nodes is duplicated.

In PA-Star, we propose a new solution to the implementation of the *OpenList*: the use of a Multi-Index. The Boost C++ Library (*www.boost.org*) provides a multi-index class template which enables the construction of data structures having more than one index with different sorting and access semantics. Our Boost C++ based Multi-Index *OpenList* has two indexes: (a) the value of function f and (b) coordinates. We also designed an *OpenList* with the dictionary approach using the STL library [21]. For comparison reasons, we use an STL priority_queue to operate in the f field, and an STL map to operate in the coordinate field. In both implementations, the *ClosedList* data structure is implemented as an STL map, since it does not requires operations in the f field.

4.4. Use of Templates

Every node in the search space has a unique coordinate, which is represented by n integers, where n is the number of sequences. Changing the number of sequences has a great impact on the performance and, for this reason, some solutions in the literature are optimized for a specific number of sequences. One common technique is to create different codes and data

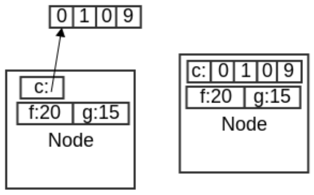


Figure 3: Dynamic Memory (Left) and Template Design (Right)

structures, each one for a given number of sequences, commonly between 4 and 8 sequences [8].

Clearly, it is inappropriate to create and maintain distinct sets of functions and data structures for every desired number of sequences. A possible solution for this issue is to use dynamic memory, where the node allocates memory according to the number of sequences. Figure 3 shows a node representation for four sequences with dynamic memory (left) and with templates (right).

The main disadvantage of using dynamic memory is the increase on memory usage. As an example, using 64-bit pointers and 16 bits integer coordinates, the node representation with dynamic memory requires about 2× more space than its templates counterpart.

Therefore, we opted to use templates. Templates allow for the compiler to create variants of data structures, classes and functions, using the same specification. In order to use templates, we created a macro to specify to the compiler which numbers of sequences will be used and the compiler uses this macro to create a code specialized for each variant. For the sake of performance comparison, we have also developed an implementation using dynamic memory allocation.

4.5. Disk-Assisted PA-Star Module (DAPA)

As stated in Section 2, one of the challenges of the A-Star algorithm is that it often requires a huge memory space. Our Disk-Assisted Module aims to overcome the limitation in size of the RAM memory by using disk space as soon as the RAM memory usage is higher than a user-defined threshold. Since empirical tests showed that the *ClosedList* is much bigger than the *OpenList* in most executions, we opted to place part of the *ClosedList* in disk, whenever the threshold is attained.

In order to define which areas of the *ClosedList* should be transferred to disk, we propose the concept of *region*. An active region is a subset of nodes which is being processed by thread t_i in a given iteration of the PA-Star algorithm. The active region of thread t_i contains the nodes with highest priority amongst the regions of thread t_i. The other regions of this thread are called inactive regions. Therefore, nodes placed in inactive regions may be placed into disk. In the next iteration, the inactive regions in disk may become active. In this case, the region is read from disk to RAM memory.

In the basic design of PA-Star (Section 4.2), each thread t_i processes exactly one region and has an $OpenList_i$ and a $ClosedList_i$. In DAPA, if thread t_i processes r regions, $OpenList_i$ and $ClosedList_i$ are partitioned into r $OpenList_{ij}$ and r $ClosedList_{ij}$, in which $j = 1..r$. Thus, one thread may process more than one region. For this reason, the hash function shown in Equation 2 was modified as shown in Equation 3.

$$(H(N_{f1}, N_{f2}, ..., N_{fk}) \gg S) \bmod (t * r) = t_i \qquad (3)$$

Since DAPA will run only when the RAM memory occupation is high, we can assume the sizes of the *OpenLists* and *ClosedLists* are considerable. For this reason, we included a new type of list, called *HoldingList*. The *HoldingList* contains nodes which are generated in the current iteration and would be otherwise placed into the *OpenList*.

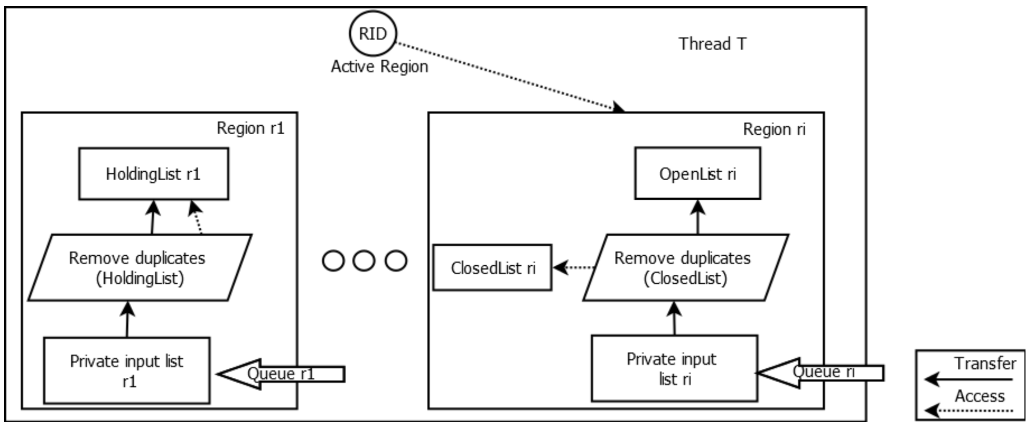


Figure 4: Design of DAPA

Figure 4 illustrates the design of DAPA. At the beginning of one iteration of Algorithm 3, each node which is in thread's t_i queue and belongs to region

j is either added to the $OpenList_{ij}$, if it belongs to the active region, or to the $HoldingList_{ij}$, otherwise. It is worth noticing that a node is only added to the $OpenList_{ij}$ if it is not already in the $ClosedList_{ij}$. If there are duplicates, the node with highest priority is kept. Then, the threads check all their regions and the region that contains the node with highest priority is set as active region. If the $ClosedList_{ij}$ of the active region for this iteration is in disk, it is transferred back to RAM memory. After, the $HoldingList_{ij}$ of the newly active region is moved to the $OpenList_{ij}$. If the RAM memory occupation is above the threshold, the region in which the node with highest priority has the lowest priority among all regions is transferred to disk iteratively until the memory occupation is below the threshold. Next, the *search_step* and *verify_end_condition* of the PA-Star are executed (Algorithm 3).

Data movement to/from disk is illustrated in Figure 5. In this case, the threshold is above the user-defined memory limit. So, the inactive region r_1 is moved to disk (1). If the inactive region r_i becomes active, it is moved back to memory (2).

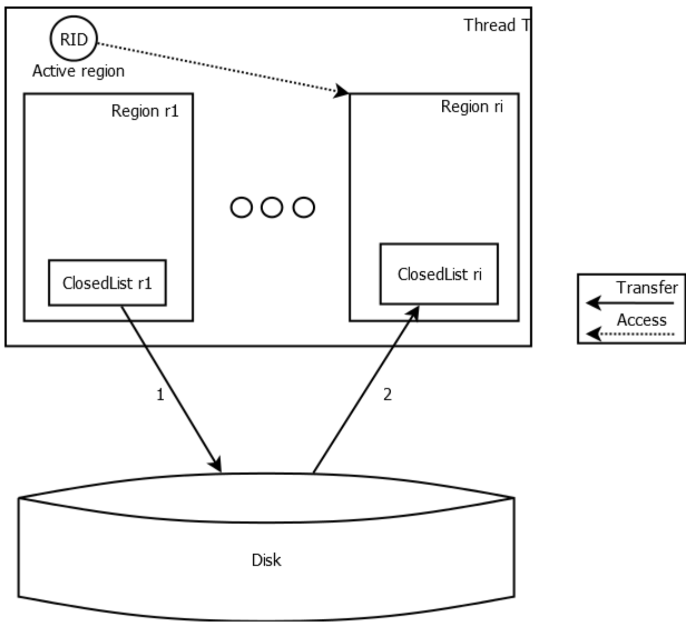


Figure 5: Data movement to/from disk in DAPA

During the implementation of DAPA, we were not able to limit the amount of RAM memory allocated to the PA-Star process due to the Linux OS restrictions. Therefore, we opted to limit the size of the *ClosedList*, which is the data structure that has the greatest impact on the amount of memory allocated to PA-Star.

5. Experimental Results

5.1. Experimental Setup

PA-Star was written in C++ with Boost C++ libraries, compiled with g++ -O3. In our tests, three different machines were used, as shown in Table 1. All machines run the operating system Linux CentOS. The first two machines (Machine1 and Machine2) are desktops available at our laboratories. Machine3 is a server machine located at Texas Advanced Computing Center (TACC) accessed through XSEDE (*www.xsede.org*).

Name	Processor	Cores	RAM (GB)	Disk
Machine1	1 Intel i7-37700 3.5GHz	4	8	1 TB
Machine2	1 Intel i7-4790 3.6GHz	4	32	200 GB
Machine3	4 Intel Xeon E5-2680 2.7GHz	32	1024	250 GB - local 14 PB - Lustre

Table 1: Machines used in the tests.

The following PA-Star parameters were used in our tests. The substitution matrix PAM250 was used to score the matches/mismatches, adjusted for the minimization problem (i.e. the values were added to 17), and the penalty for gaps was set to 30. Otherwise stated, the number of regions per thread was set to 1 and DAPA (Section 4.5) was disabled.

The experiments were executed using sets of sequences retrieved from the BAliBASE [17] benchmark from *http://labs.bio.unc.edu/Vision/private/Documentation/VisionLab/doc/BAli BASE/align_index.html*. BAliBASE is a widely used database composed of sets of sequences with their corresponding MSAs, which were manually defined by biologists in order to evaluate and compare MSA tools. In its current version, there are 9 reference sets. In our tests, we used all the 82 sets of sequences that belong to reference set 1, which contains sets of less than 7 sequences of similar length. We also used one sequence retrieved from the

PFAM database available at *http://pfam.xfam.org* and synthetic sequences. Synthetic sequences 1 to 3 were generated empirically by moving blocks of characters in a way that favors the addition of gaps in the optimal alignment as in [22]. Sequence synth4 used the BAliBASE sequence set 2ack as a basis and was generated by moving blocks of characters among the sequences in this set, generating a hard alignment pattern. The characteristics of the sequences mentioned in this section are shown in Table 2.

Base	Reference	# Seq	Shortest	Longest
PFAM	PF07708	22	17	17
BAliBASE	1gdoA	4	235	265
BAliBASE	1dlc	4	568	590
BAliBASE	3pmg	4	540	567
BAliBASE	1wit	5	90	106
BAliBASE	1hva	5	137	199
BAliBASE	arp	5	380	418
BAliBASE	1sesA	5	417	442
BAliBASE	glg	5	438	486
BAliBASE	2ack	5	452	482
BAliBASE	1gpb	5	796	828
BAliBASE	1pamA	5	435	572
BAliBASE	1taq	5	806	928
BAliBASE	1lcf	6	662	691
Synthetic	synth1	3	231	416
Synthetic	synth2	5	80	98
Synthetic	synth3	5	619	619
Synthetic	synth4	5	600	600

Table 2: Sequences used in the tests

5.2. Multi-index vs Dictionary

The goal of this experiment is to evaluate the gains obtained with the multi-index data structure. Machine2 (Table 1) was used in this test. Two different approaches for the implementation of the *OpenList* in PA-Star were considered: multi-index and Dictionary.

Table 3 presents the results obtained with four sets of sequences: PF07708, 3pmg, synth1 and synth2 (Table 2). The execution times and the RAM memory were obtained with the command *time*. In this table, we can notice that, for the sequences compared, the multi-index approach consumes less memory and has better execution times than its dictionary counterpart.

In this test, the best result was obtained for the sequence set PF07708, in which the execution time was reduced in 28.5% and the RAM memory in 57.7%. For this sequence, RAM memory usage was decreased from 28.8 GB to 11.97 GB. The execution of the synth2 set with multi-index achieved 0.5% reduction in the execution time and 21.4% in memory usage whereas the synth1 execution had a 23.2% reduction in execution time and 2.09% in memory usage. These two sequences were manually produced with complex character configurations. Even in this cases, multi-index showed that it is a better choice than dictionary.

Sequences (k)	Data Structure	Time (s)	RAM (GB)
PF07708 (22)	Dictionary	452.91	28.28
	Multi-Index	324.05	11.97
3pmg (4)	Dictionary	75.20	0.73
	Multi-Index	66.35	0.59
synth1 (3)	Dictionary	41.14	0.48
	Multi-Index	31.58	0.47
synth2 (5)	Dictionary	585.21	3.17
	Multi-Index	582.72	2.49

Table 3: Execution times and memory usage for multi-index vs dictionary

5.3. Templates vs Dynamic Memory

In this experiment, we used Machine2 to compare the PA-Star implementation using dynamic memory allocation with its implementation with templates. In this case, the multi-index *OpenList* was used. We executed PA-Star with the same sets of sequences shown in Section 5.2.

Table 4 presents the execution time and memory usage obtained in this experiment. It can be seen that the execution time and memory usage are greatly reduced when templates are used. We observed a reduction in the execution time from 18.7% (PF07708) to 43.42% (synth1) and a reduction from 30.58% (PF07708) to 53.20% (synth2) in memory usage when using templates. This great reduction can be explained by the usage of more compiler optimizations, which can be applied when using templates. In this case, the compiler knows exactly how many sequences are used in all functions and data structures, allowing it to make optimized memory allocations.

Figure 6 presents the output of PA-Star for the PF07780 sequence set. Even though there are 22 sequences in this set, PA-Star is able to retrieve the optimal alignment in less than five minutes since the sequences are very similar.

Sequences (k)	Technique	Time (s)	RAM (GB)
PF07708 (22)	Dynamic Memory	324.05	11.97
	Templates	263.49	8.31
3pmg (4)	Dynamic Memory	66.35	0.59
	Templates	39.97	0.29
synth1 (3)	Dynamic Memory	31.58	0.47
	Templates	17.87	0.22
synth2 (5)	Dynamic Memory	582.72	2.49
	Templates	358.69	1.22

Table 4: Execution times and memory usage for dynamic memory allocation vs templates

5.4. Optimizations Overview

Tables 5 and 6 present the impact of the proposed optimizations (multi-index data structure and templates) as well as multithreading for the 3pmg and synth5 sequence sets, in Machine2 with hyperthreading (HT) enabled.

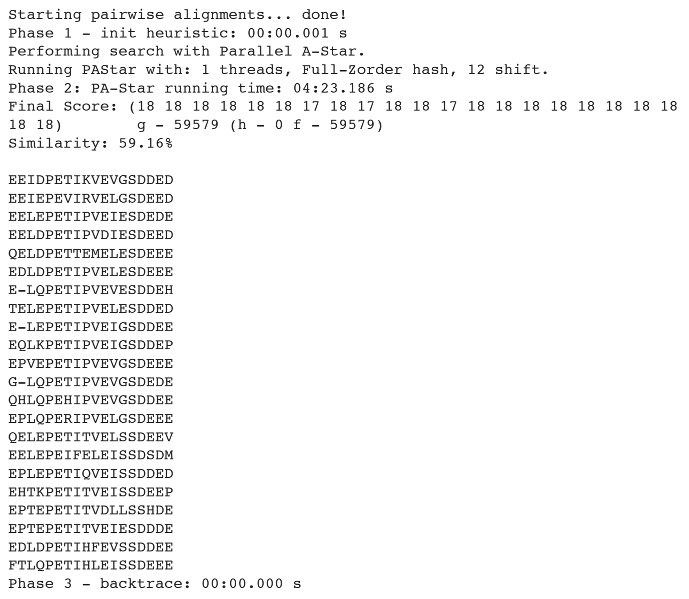
Data Structure	Technique	Threads	Time (s)	RAM (GB)
Dictionary	Dynamic Mem.	1	75.20	0.73
Multi-Index	Dynamic Mem.	1	66.35	0.59
Multi-Index	Template	1	39.97	0.29
Multi-Index	Template	2	20.03	0.29
Multi-Index	Template	4	10.88	0.31
Multi-Index	Template	4+4HT	07.23	0.32

Table 5: Impact of the proposed optimizations and multithreading for 3pmg

In the 3mpg comparison, the use of multi-index and templates reduced the elapsed time 1.88×. With 8 threads, the time is reduced from 39.97s to 7.23s (5.52×). With the optimizations and multithreading, the total time is reduced from 75.20s to 7.23s, with a total 10.40× speedup using the optimizations and 4-HT cores.

In the synth2 comparison, we observed the same behavior, with a speedup of time 1.63× when the optimizations multi-index and templates are applied. With multithreading, the execution time is reduced from 358.69s (1 thread) to 74.94s (8 threads), with a speedup of 4.78×. Applying the optimizations and multithreading, PA-Star was able to reduce the execution time from 585.21 to 74.94, achieving a speedup of 7.80×.

There was also a great reduction in the use of memory with the use of the proposed optimizations: 2.51× and 2.26× for the execution with sets 3pmg



```
Starting pairwise alignments... done!
Phase 1 - init heuristic: 00:00.001 s
Performing search with Parallel A-Star.
Running PAStar with: 1 threads, Full-Zorder hash, 12 shift.
Phase 2: PA-Star running time: 04:23.186 s
Final Score: (18 18 18 18 18 18 17 18 17 18 18 17 18 18 18 18 18 18 18 18
18 18)       g - 59579 (h - 0 f - 59579)
Similarity: 59.16%

EEIDPETIKVEVGSDDED
EEIEPEVIRVELGSDEED
EELEPETIPVEIESDEDE
EELDPETIPVDIESDEED
QELDPETTEMELESDEEE
EDLDPETIPVELESDEEE
E-LQPETIPVEVESDDEH
TELEPETIPVELESDDED
E-LEPETIPVEIGSDDEE
EQLKPETIPVEIGSDDEP
EPVEPETIPVEVGSDEEE
G-LQPETIPVEVGSDEDE
QHLQPEHIPVEVGSDDEE
EPLQPERIPVELGSDEEE
QELEPETITVELSSDEEV
EELEPEIFELEISSDSDM
EPLEPETIQVEISSDDED
EHTKPETITVEISSDEEP
EPTEPETITVDLLSSHDE
EPTEPETITVEIESDDDE
EDLDPETIHFEVSSDDEE
FTLQPETIHLEISSDEEE
Phase 3 - backtrace: 00:00.000 s
```

Figure 6: Output produced by PA-Star for the PF07780 sequence set

and synth2, respectively. With multithreading, as expected, memory usage slightly increases due to synchronization data.

5.5. Hash Function Comparison

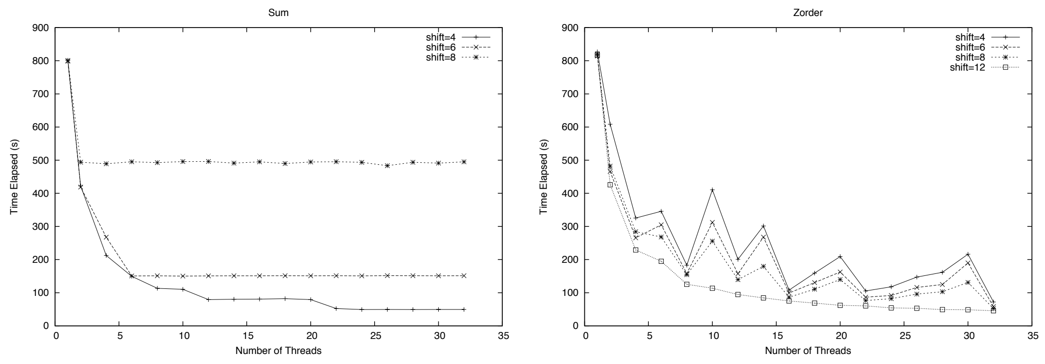
Three factors are used to decide in which *OpenList* queue the node should be written (Equations 2 and 3): (a) The hash function, SUM or $ZORDER$, (b) S, the constant right-shift factor, and (c) the total number of threads and regions managed by each thread.

If the hash function always decides that the result of expansion should be inserted in queues that belong to other threads, then the overhead of communication is high and the performance is affected. On the other hand, if the hash function decides that the result of expansion should never be inserted in other threads queues, then load imbalance is very high and the execution is almost serialized. In this experiment, we used Machine3 (Table 1) to test different combinations of hash functions, threads and shift-right factors.

Figure 9 shows the elapsed time to compare the synth2 set with the ZORDER (left) and SUM (right) hash functions. It can be seen that the hash function and the number of bits (shift) has a great impact on performance.

Data Structure	Technique	Threads	Time (s)	RAM (GB)
Dictionary	Dynamic Mem.	1	585.21	3.17
Multi-Index	Dynamic Mem.	1	582.72	2.42
Multi-Index	Template	1	358.69	1.22
Multi-Index	Template	2	191.41	1.24
Multi-Index	Template	4	109.48	1.30
Multi-Index	Template	4+4HT	74.94	1.40

Table 6: Impact of the proposed optimizations and multithreading for synth2



(a) SUM (b) ZORDER

Figure 7: Hash function SUM and ZORDER for sequence set synth2 in Machine3 (32 cores)

In the best case, the execution with ZORDER hash has a minimum elapsed time of 45s (17.9×) and the SUM hash 49s (16.2×). The graphic also shows that the ZORDER has more stable results as the shift values are varied, while the performance with SUM hash quickly degrades for values higher than 4. In other words, SUM seems to be much less scalable than ZORDER. For both cases, the number of re-expanded nodes and the overhead of communication increases with the number of threads. The graph also shows a local minimum with 4, 8, 16 and 32 threads, because the Equation 2 mod operation is optimized to a quick shift operation. With these results, we decided to use as default the *ZORDER* hash, with 12 as a shift factor.

5.6. BAliBASE Comparison

With the *ZORDER* hash, 12 shift factor and 16 threads, we decided to run PA-Star for all 82 sequence sets in BAliBASE Reference 1, in Machine3. In this experiment, we used 16 cores since empirically we noticed that the execution times for 16 cores were slightly better than the execution times for 32 cores.

Most of the sequence sets of BAliBASE reference set 1 of were quickly solved. The total time, including backtrace time, of the 76 easiest sequence set executions was 51 minutes. The PA-Star execution for three hard BAliBASE sequences (1sesA, arp and 1gpb) (Table 2) took almost 4 hours to finish and 3 executions (1pamA, 1lcf, 1taq) ran out of memory in Machine3 which has 1TB of RAM memory.

Sequences (k)	Avg Length	Time (hh:mm:ss)	Size *ClosedList*	Size Lists (Total)
1sesA (5)	427.8	00:30:01	368,609,668	455,761,987
arp (5)	397.0	00:56:43	603,275,186	797,852,355
1gpb (5)	809.8	02:39:27	1,615,316,096	2,009,496,349

Table 7: Elapsed time and number of page faults for 3 hard instances of Balibase 1 (16 cores)

Table 7 presents the execution time for the sequence sets 1sesA, arp and 1gpb, which are 3 of the most difficult BAliBASE reference 1 sequence sets. In this table, it can be seem that most of the list space of PA-Star is used with the *ClosedList* (75.61% and 80.87%) and that the number of nodes in this list is huge for these sequences ($> 300,000$).

5.7. DAPA Results

In order to execute DAPA, we first performed a profiling of the application to define the number of threads and the number of regions per thread. We chose sequences 1*gdoA*, 1*dlc* and 1*wit* and run PA-Star with DAPA enabled in Machine1 (Table 1), varying the number of threads from 2 to 8 and the number of regions per thread from 32 to 1024. In this evaluation, the threshold was set in a way that disk was not used. The results for these three sequence sets were quite similar and, for this reason, we only show the results for sequence 1*wit* in Figure 8.

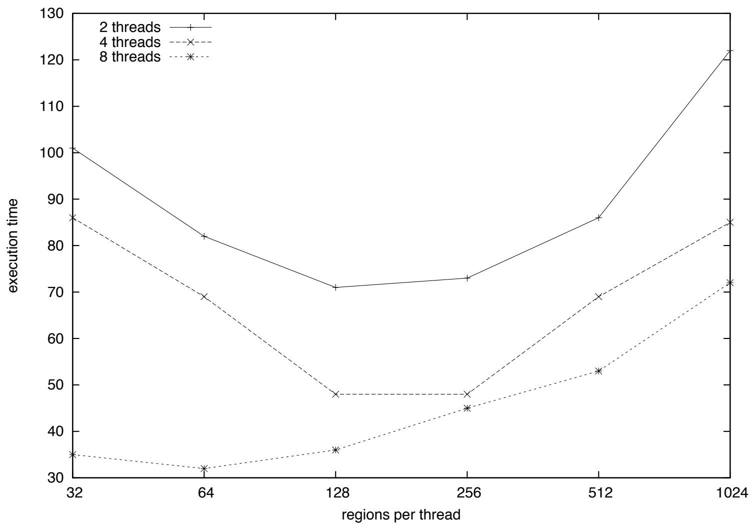


Figure 8: Execution times for 2, 4 and 8 threads for the 1*wit* comparison in Machine1

As shown in Figure 8, the execution time decreases when we augment the number of regions up to a certain number of regions and then it increases again. The lowest execution times for Machine1 were obtained with 8 threads and 64 regions per thread. Therefore, we chose this configuration to run the experiment on the impact of the available RAM memory.

In this experiment, we executed PA-Star with DAPA enabled for the 1*wit* comparison using 8 threads and 64 regions per thread with decreasing values of available RAM memory. In all cases, DAPA started to execute when 90% of the maximum size of the *ClosedList* was attained. We varied the amount of RAM memory allocated to the *ClosedList* from 140 MB to 17.5 MB, as shown in Table 8. In this table, the execution time and RAM memory were obtained with the *time* command. The available memory to the *ClosedList* is a parameter of PA-Star and the maximum occupation of the *ClosedList* was measured inside PA-Star.

As expected, Table 8 shows that reducing the available memory has a great impact on the overall execution time. When the whole *ClosedList* fits into memory (140MB), no accesses to disk are made and the PA-Star execution time is 15.16s. However, when the disk is necessary, an overhead is introduced and the execution times increase.

In the 1*wit* comparison, when the available memory to the *ClosedList*

Available Memory *ClosedList* (MB)	Execution Time (s)	Max Occupation *ClosedList* (MB)	RAM Memory (MB)
140.0	15.16	70.59	413.46
70.0	32.77	63.30	399.60
56.0	78.07	51.39	379.29
35.0	436.07	39.94	315.95
17.5	790.58	17.6	272.84

Table 8: Impact of the available memory in the execution times

is reduced from 140 MB to 17.5 MB (8×), the execution time increases from 15.16 s to 790.58 s (52×). This is indeed a considerable increase in the execution time but DAPA, in this case, allows PA-Star to complete its execution and return the optimal result to the user.

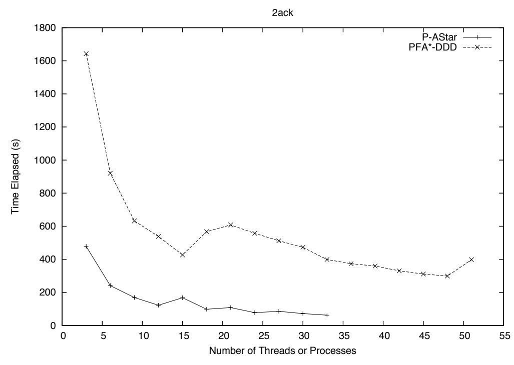
When PA-Star is executed with DAPA disabled and there is not enough RAM memory, the execution would not complete, terminating with a memory exception. In order to verify this, we compared sequence $1hvA$ (Table 2) with and without DAPA enabled. This comparison needs about 10 GB to complete and the total RAM memory in Machine1 is 8 GB. So, as expected, the PA-Star execution with DAPA disabled ran out of memory.

In the DAPA-enabled test, we first tried to set the parameters as 8 threads and 64 regions per thread. However, the execution was very slow so we empirically augmented the number of regions per thread to 256. With these parameters, PA-Star took 7 hours, 14 minutes and 27 seconds to align the sequence set $1hvA$, showing that, with DAPA, PA-Star is able to execute MSAs which require more than the total RAM memory of the machine.

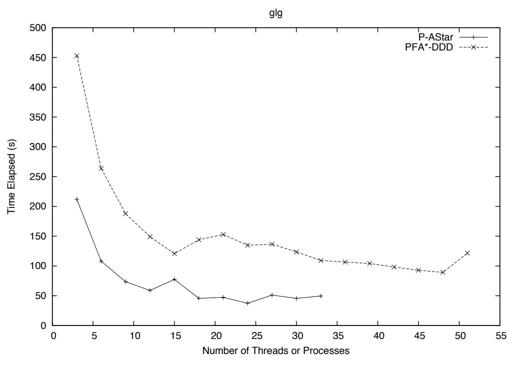
5.8. Comparison with PFA-DDD*

In this test, we used Machine3 to compare our implementation to one of the state-of-the-art solutions, PFA*-DDD (Section 3). To do this comparison, the authors kindly provided us the source code. PFA*-DDD requires an MPI library and in this test we used Intel IMPI version 5.0.1.035. PFA*-DDD requires multiples of 3 processes to run, while our solution can be executed with any number of threads.

Figure 9 shows the elapsed time for running the 2ack and glg sequence sets (Table 1). In this test, the sequences were pruned to have all the same size. It is possible to notice that our solution always has better execution



(a) 2ack



(b) glg

Figure 9: Comparison between PA-Star and PFA*-DDD in Machine3 (32 cores)

times, and the best result is achieved with 32 threads. PFA*-DDD minimizes execution time with 48 processes.

Sequence	PA-Star	PFA*-DDD
	Time (s)	Time (s)
2ack	62.64	299.21
glg	30.76	89.15

Table 9: Comparison between our Parallel A-Star strategy (with 32 threads) and PFA*-DDD (with 48 processes).

Table 9 shows the best running time for both solutions. As can be seen, PA-Star compares the 2ack instance in 1 minute and 2 seconds whereas PFA*-DDD takes almost 5 minutes to do the same comparison. For the glg sequences, PA-Star running time is 0:30.76, while the PFA*-DDD total executing time is 1:29.15.

5.9. PA-Star-DAPA, PA-Star and PFA-DDD comparison*

In this experiment, we compared PFA*-DDD (Section 3) with PA-Star without DAPA and PA-Star with DAPA. We used the Machine1, running PFA*-DDD with Intel IMPI version 5.0.1.

Table 10 compares PA-Star with DAPA, PA-Star without DAPA and PFA*-DDD, when executing the sequence set synth4. As can be seen, PA-

Sequence (k)	PA-Star-DAPA (s)	PA-Star (s)	PFA*-DDD (s)
synth4 (5)	5604	out of memory	out of memory

Table 10: Comparison between our Parallel A-Star strategy (with 8 threads), DAPA (with 4 threads and 256 regions), and PFA*-DDD (with 12 processes).

Star with DAPA is able to generate the optimal alignment for synth4 in 1 hour and 33 minutes whereas PFA*-DDD and PA-Star did not finish the execution, because there is not enough RAM memory.

In this experiment, the PA-Star created 8 threads and run out of memory in 8 minutes. PFA*-DDD created 12 processes and ran out of memory after 42 minutes and 49 seconds. PA-Star with DAPA created 4 threads and 256 regions per thread, with the available RAM memory to the *ClosedList* set to 1.5 GB. This test shows that DAPA is able to execute alignments which require more than the total available RAM memory required while other two solutions terminate with memory exception.

6. Conclusion and Future Work

In this paper, we proposed and evaluated PA-Star, a parallel solution which is able to retrieve optimal MSAs in multiple cores. We proposed many optimizations such as the usage of a multi-index data structure, templates and a locality-sensitive hash function. In PA-Star parallel execution, we also proposed to execute part of the synchronization among the threads asynchronously, accelerating the execution.

The results obtained with the sequence sets from the BAliBASE reference set 1 shows that PA-Star is able to reduce the elapsed time from 75.20s to 7.23s, by adding the multi-index data structure, templates and parallel execution using 8 cores. Using PA-Star, we were able to compare 79 BAliBASE sequence sets out of 82 in 4 hours and 56 minutes, producing the optimal results for all these sets. We also showed that PA-Star outperforms with the state-of-the-art tool PFA*-DDD, executing up to 4.77× faster. Finally, we showed that our disk-assisted strategy DAPA is able to retrieve optimal alignments when PFA*-DDD and PA-Star without the disk module run out of memory.

As future work, we intend to extend PA-Star to run in NUMA architectures, modifying the locality-sensitive hash function to take into account the distance between the cores. In addition, we intend to investigate other admissible heuristics, rather than the $h_{2,all}$, which is used in PA-Star. We also want to thoroughly investigate the impact of parameters such as number of threads, number of regions, hash function, amount of RAM memory, number of sequences, size of sequences and similarity among the sequences in the execution times.

Acknowledgment

We would like to thank Prof. Jose Nelson Amaral and Robert Niewiadomski for providing us the PFA*-DDD source code. This work was partially supported by Capes/Brazil and CNPq/Brazil.

References

[1] R. Durbin, S. R. Eddy, A. Krogh, G. Mitchison, Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids, Cambridge Univ. Press, 1999.

[2] L. Wang, T. Jiang, On the complexity of multiple sequence alignment, Journal of Computational Biology 1 (4) (1994) 337–348.

[3] H. Carrillo, D. Lipman, The multiple sequence alignment problem in biology, SIAM Journal of Applied Mathematics 48 (1988) 1073–1082. doi:10.1137/0148063.

[4] P. Hart, N. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans Sys Sci and Cybernetics SSC-4(2) (1968) 100–107.

[5] J. L. Spouge, Speeding up dynamic-programming algorithms for finding optimal lattice paths, SIAM J. Applied Mathematics 49 (5) (1989) 1552–1566.

[6] K. R. K., J. S. J., T. Will, Combining divide-and-conquer, the a-algorithm, and successive realignment approaches to speed multiple sequence alignment., in: German Conference on Bioinformatics, 1999, pp. 17–24.

[7] M. Lermen, K. Reinert, The practical use of the A* algorithm for exact multiple sequence alignment, Journal of Computational Biology 7 (2000) 655–673.

[8] R. Niewiadomski, J. N. Amaral, R. Holte, Sequential and parallel algorithms for frontier A* with delayed duplicate detection, in: Proc. of AAAI, 2006, pp. 1039–1044.

[9] M. Hatem, W. Ruml, External memory best-first search for multiple sequence alignment, in: Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, 2013, pp. 409–416.

[10] D. W. Mount, Bioinformatics: Sequence and Genome Analysis, 1st Edition, Cold Spring Harbor Laboratory Press, 2001.

[11] M. O. Dayhoff, R. M. Schwartz, B. C. Orcutt, A model of evolutionary change in proteins, Atlas of Protein Sequence and Structure 5 (suppl 3) (1978) 345–351.

[12] S. Needleman, C. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins., Journal of Molecular Biology (1970) 443–453.

[13] H. Kobayashi, H. Imai, Improvement of the A* algorithm for multiple sequence alignment, in: Proceedings of the 9th Workshop on Genome Informatics, 1998, pp. 120–130.

[14] R. Zhou, E. A. Hansen, Sweep A*: Space-efficient heuristic search in partially ordered graphs, in: In Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence, 2003, pp. 427–434.

[15] T. Yoshizumi, T. Miura, T. Ishida, A* with partial expansion for large branching factor problems, in: Proceedings of the American Association for Artificial Intelligence (AAAI/IAAI), 2000, pp. 923–929.

[16] R. E. Korf, Linear-time disk-based implicit graph search, Journal of the ACM 55 (6) (2008) 1–26.

[17] J. D. Thompson, F. Plewniak, O. Poch, BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs., Bioinformatics 15 (1) (1999) 87–88.

[18] R. E. Korf, Delayed duplicate detection: Extended abstract., in: IJCAI, 2003, pp. 1539–1541.

[19] R. E. Korf, P. Schultze, Large-scale parallel breadth-first search, in: Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3, AAAI'05, AAAI Press, 2005, pp. 1380–1385.

[20] Morton, A computer oriented geodetic data base and a new technique in file sequencing, Tech. rep., IBM Ltd. (1966).

[21] D. R. Musser, G. J. Derge, A. Saini, STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Addison-Wesley Professional, 2001.

[22] D. Sundfeld, A. C. M. A. de Melo, MSA-GPU: exact multiple sequence alignment using GPU., in: J. C. Setubal, N. F. Almeida (Eds.), BSB, Vol. 8213 of Lecture Notes in Computer Science, Springer, 2013, pp. 47–58.