Multi-GPU Approach for Large-Scale Multiple Sequence Alignment

Rodrigo A. de O. Siqueira[1], Marco A. Stefanes[1], Luiz C. S. Rozante[2], David C. Martins-Jr[2], Jorge E. S. de Souza[3], and Eloi Araujo[1,2](✉)

[1] Faculty of Computing, UFMS, Campo Grande, Brazil
rodrigo.siqueira@ufms.br, {marco,feloi}@facom.ufms.br
[2] Center of Mathematics, Computing and Cognition, UFABC, São Paulo, Brazil
{luiz.rozante,david.martins}@ufabc.edu.br
[3] Bioinformatics Multidisciplinary Environment (BioME), Metrópole Digital Institute, UFRN, Natal, Brazil
jorge@imd.ufrn.br

Abstract. Multiple sequence alignment is an important tool to represent similarities among biological sequences and it allows obtaining relevant information such as evolutionary history, among others. Due to its importance, several methods have been proposed to the problem. However, the inherent complexity of the problem allows only non-exact solutions and further for small length sequences or few sequences. Hence, the scenario of rapid increment of the sequence databases leads to prohibitive runtimes for large-scale sequence datasets. In this work we describe a Multi-GPU approach for the three stages of the Progressive Alignment method which allow to address a large number of lengthy sequence alignments in reasonable time. We compare our results with two popular aligners ClustalW-MPI and ClustalΩ and with CUDA NW module of the Rodinia Suite. Our proposal with 8 GPUs achieved speedups ranging from 28.5 to 282.6 with regard to ClustalW-MPI with 32 CPUs considering NCBI and synthetic datasets. When compared to ClustalΩ with 32 CPUs for NCBI and synthetic datasets we had speedups between 3.3 and 32. In comparison with CUDA NW_Rodinia the speedups range from 155 to 830 considering all scenarios.

Keywords: Multiple sequence alignment · MSA · Hybrid Parallel Algorithms · Multi-GPU Algorithms · Large sequence alignment

1 Introduction

A very relevant task in bioinformatics is sequence alignment, which is routinely employed in many situations, such as comparing a query sequence with databases, comparative genome and sequence similarity searching. There are many important real world objectives which can be pursued by applying sequence alignment, such as paternity test, criminal forensics, drug discovery, personalized medicine, species evolution studies (phylogeny), just to mention a few. In fact, there is a myriad of techniques and tools proposed with this aim, including BLAST [1], S-W [2] and N-W [3] as the most popular ones. In particular, multiple sequence alignment (MSA) is one of the important formulations of the problem whose objective is to align multiple sequences at once. It is usually involved in phylogeny, molecular (2D and 3D) structure predictions such as proteins and RNAs, among other applications. Due to the COVID-19 pandemic, researchers are keen to reveal SARS-CoV-2 strains phylogeny hoping to understand the implications of the emerging strains in public health. Currently there are about a million strain sequences deposited (https://www.gisaid.org).

Several methods have been proposed for MSA, such as MAFFT [4], ClustalW [5], Kalign [6] and DeepMSA [7]. Yet, the assembly of optimal MSAs is highly computationally demanding considering both processing and memory requisites, since the problem is considered NP-Hard [8]. Dynamic programming based approaches retrieve an MSA with k sequences of length n in $O(2^k k^2 n^k)$. These techniques usually present important limitations regarding both length and number of input sequences to be computationally feasible [9].

Due to the aforementioned limitations and the fact that both the number of biological sequences and sequence lengths are continuously growing, finding fast solutions have led to employment of high performance computing techniques to achieve MSA as the popular aligners ClustalW_MPI [10] and ClustalΩ [11]. Hybrid parallel implementations of MSA have been recently proposed [12,13]. The former addresses only the first stage and the latter implements the three stages of the progressive alignment method, namely: i) pairwise alignment on Multi-GPUs with MPI-based communication among processes; ii) Neighbor Joining [14] implementation in a single GPU to build the guide tree; and iii) CUDA-GPU cluster implementation of the parallel progressive alignment algorithm similar to the implementation done by Truong et al. [15].

In this work we improved the three stages of the method proposed in [13] by addressing lengthy sequences during stage i), developing a scalable Neighbor Joining using Multi-GPU and paralleling Myers-Miller [16] algorithm. To the best of our knowledge this is the first Multi-GPU Neighbor Joining method in the literature. In fact, when comparing the results obtained by our method with the ClustalW-MPI and with the CUDA NW module of the Rodinia Suite [17], our proposal with 8 GPUs achieved speedups ranging from 28.5 to 282.6 with regard to ClustalW-MPI with 32 CPUs considering three NCBI datasets and three synthetic datasets. In comparison with CUDA NW_Rodinia the speedups range from 155 to 830. When compared to ClustalΩ with 32 CPUs in NCBI and synthetic datasets, we had speedups between 3.3 and 32. Regarding accuracy and quality of solution, the proposed method had a performance similar to Clustal-W and ClustalΩ, considering the benchmark BAliBASE [18].

This text is structured as follows: Sect. 2 introduces some basic concepts; Sect. 3 describes the computational model and some details about the parallel algorithms; Sect. 4 shows experimental results, including a comparative analysis. Finally, Sect. 5 presents the final remarks and future work.

2 Preliminaries

A *sequence over a finite alphabet Σ* is a finite enumerated collection of elements in Σ. The *length of a sequence s*, denoted by $|s|$, is the number of symbols of s and the *j-th element of s* is denoted by $s(j)$. Thus, $s = s(1) \ldots s(|s|)$. The set of all sequences over Σ is denoted by Σ^*. Let $S = \{s_0, s_1, \ldots, s_{k-1}\} \subseteq \Sigma^*$. An *alignment* of S is a set $A = \{s'_0, \ldots, s'_{k-1}\} \subseteq \Sigma^*_{␣}\Big(= (\Sigma \cup \{␣\})^*\Big)$, where: (i) $␣ \notin \Sigma$ is a new symbol called *space*; (ii) $|s'_h| = |s'_i|$; (iii) s'_i is obtained by inserting spaces in s_i; (iv) there is no j such that $s'_i(j) = ␣$ for every i, $0 \leq i < k$. Note that a sequence can be seen as an alignment for $k = 1$, i.e., $\{s\}$ is the single alignment of s and, hence, we sometimes refer to a sequence as an alignment. The *length of alignment A* is $|s'_i|$ and it is denoted by $|A|$. We denote by $\mathcal{A}_S$ the set of all alignments of S.

An alignment is an important tool for comparison of sequences obtained from organisms that have the same kind of relationship. It shows which part of each sequence should be compared to the other, thus suggesting how to transform one sequence into another by substitution, insertion or deletion of symbols. An alignment can be visualized placing each sequence above another as showed in the following figure with two different alignments of (*abacb*, *bacb*, *aacc*). The left part represents the alignment (*abacb*, *␣bacb*, *a␣acc*) and the right one represents the alignment (*aba␣cb␣*, *␣b␣acb␣*, *a␣a␣c␣c*). Notice that the first suggests that the last c in the third sequence comes from substitution operation and the second alignment suggests that it comes from insertion operation.

$$\begin{array}{ccccc} a & b & a & c & b \\ ␣ & b & a & c & b \\ a & ␣ & a & c & c \end{array} \qquad \begin{array}{ccccccc} a & b & a & ␣ & c & b & ␣ \\ ␣ & b & ␣ & a & c & b & ␣ \\ a & ␣ & a & ␣ & c & ␣ & c \end{array}$$

An *optimal alignment* in $\mathcal{A}_S$ is one which maximizes a given objective function whose value is also called *similarity* of S. The problem of finding an optimal alignment or even only its similarity is NP-hard for many objective functions.

This work deals with a polynomial method known as *progressive alignment* [19], which is described in 3 stages. This organization is extremely convenient because the algorithms for each stage are studied and improved independently in this work. The first stage corresponds to the *PairWise alignment* (PW) of all pairs of sequences, which builds a similarity matrix D. Using D as input, the next stage Neighbor Joining (NJ) consists in generate of a rooted binary tree T according to the similarity of each pair of sequences, which means that the more similar two nodes are, the closer they are. Each node represents an *operational taxonomic unit* (OTU): the leaves represent the sequences given in the PW step and the internal nodes represent hypothetical ancestor of theiR descendant nodes. We can refer a vertex set of a subtree of T as a set of OTUs. This tree T is known as a *guide tree* and each subtree of T corresponds to a group of closest related OTUs. The third stage, known as *Progressive Alignment* (PA), receives T as input, builds a profile for each node u of T and returns a profile of the root of T. The *profile* of the root of any subtree T' is a multiple alignment of the sequences that are leaves of T' and it is according to T' topology.

Pairwise Alignment (PW). A *scoring matrix* γ for Σ is a function $\gamma : \Sigma_{\text{␣}} \times \Sigma_{\text{␣}} \rightarrow \mathbb{R}$, such that $\gamma(\text{␣}, \text{␣}) = 0$, $\gamma(a, \text{␣}) = g$ for some constant $g \in \mathbb{R}$ and $\gamma(a, b) = \gamma(b, a)$ for each pair $a, b \in \Sigma_{\text{␣}}$. Function γ is used to attribute value for each pair of aligned sequences.

Given sequences s and t, we define a matrix H:

$$H[i, j] = \max \begin{cases} H[i-1, j-1] + \gamma(s(i), t(j)), \\ H[i-1, j] + g, H[i, j-1] + g \end{cases} \tag{1}$$

where $H[0, 0] = 0$. The value of the optimal alignment of s, t is $H[|s|, |t|]$. If $|s|$ and $|t|$ are $O(n)$, then H can be computed in $O(n^2)$ time and, once H is calculated, an optimum alignment (s', t') such that $\sum_i \gamma(s'(i), t'(i))$ is maximum can be found in $O(n)$ time. Matrix H is called *alignment matrix.*

A matrix D indexed by sequences and called *similarity matrix* is generated, where $D[s_i, s_h]$ is the value of the optimum alignment of s_i and s_h. As a consequence of γ definition, D is a symmetric matrix, which implies that we can represent D as a lower triangular matrix. Since there are $O(k^2)$ entries of D and each entry spends $O(n^2)$ time to be computed, the overall time spent in this step is $O(k^2n^2)$. Matrix D is the input to the next step as follows.

Neighbor Joining (NJ). This stage creates *guide phylogenetic tree* T that is a binary phylogenetic tree from the similarity matrix D computed in the previous stage and it is the input of the next stage. This stage is implemented using the NJ algorithm [14]. The building begins with a star tree initially given by the set S of k sequences representing the k leaves that are the indices of D and a virtual node c in the center. In each iteration, if $|S| = 2$, it deletes the node c and connect directly the two vertices in S. Otherwise, pick u, v for which

$$(k-2)D[u, v] - \sum_{w \in S - \{u,v\}} \Big(D[u, w] + D[w, v] \Big) \tag{2}$$

is the largest. Then it deletes edges (u, c) and (c, v), creates a new vertex w (new OTU) and edges $(u, w), (v, w), (w, c)$, updates $S = (S - \{u, v\}) \cup \{w\}$ and sets

$$D[w, z] = \frac{1}{2} \Big(D[u, z] + D[v, z] - D[u, v] \Big) \tag{3}$$

for each vertex $z \neq w$ in S. It is performed in $k-2$ iterations. The expression (2) runs in time $O(k^3)$ in the first iteration and, if the computed values are stored, it spends $O(k^2)$ time in each of the next iterations. Expression (3) spends $O(1)$ time in each iteration. Thus, the total running time spent in stage 2 is $O(k^3) + (k-3) \cdot O(k^2) + (k-2) \cdot O(1) = O(k^3)$.

Progressive Alignment (PA). This stage buids an MSA by combining pairwise profiles and the guide tree described in the previous sections. For convenience, let us assume that the guide tree obtained is rooted. This assumption is not a special constraint. A rooted binary tree can be obtained from a rootless binary tree with the same set of leaves by subdividing some edge of the tree. By doing this, each internal node has exactly two children.

Given two sets S and S' of sequences, $S' \subseteq S$, and two alignments $C \in \mathcal{A}_S$ and $A \in \mathcal{A}_{S'}$, C is *compatible* with A if the elements of S' are aligned in C in the same way as in A. Feng and Doolittle [20] describe, from the alignments A and B, how to build a new alignment C that is compatible with A and B. Next, we show an example of two alignments A and B, and the respective compatible alignment C.

$$\begin{array}{rl} & a\;c\;\llcorner\;a\;b \\ A = & b\;b\;b\;a\;\llcorner \\ & a\;c\;a\;a\;a \\ \hline B = & \begin{array}{l} c\;a\;b\;b\;a\;a \\ c\;a\;\llcorner\;\llcorner\;c\;a \end{array} \end{array} \qquad \begin{array}{rl} & a\;c\;\llcorner\;\llcorner\;a\;b\;\llcorner \\ & b\;b\;\llcorner\;b\;a\;\llcorner\;\llcorner \\ C = & a\;c\;\llcorner\;a\;a\;a\;\llcorner \\ & c\;a\;b\;\llcorner\;b\;a\;a \\ & c\;a\;\llcorner\;\llcorner\;\llcorner\;c\;a \end{array}$$

The algorithm traverses the set of internal nodes of the rooted guide tree in post-order and it defines an profile for each visited OTU such that it is compatible with its children's profiles (and by transitivity, compatible with each of its descendants, including leaves). The root profile is the final MSA of the method.

Given an alignment $A = \{s'_0, \dots, s'_{k-1}\}$, a column j of A is denoted by $A(j)$ and define $\Gamma(A(j)) = \sum_{i<h} \gamma(s'_i(j), s'_h(j))$.

Now, we describe how to get the rooted profile A of a set S. First of all, consider the corresponding profiles A_1 and A_2 of (two) root children. Suppose that $|A_1| = n_1$ and $|A_2| = n_2$, A_1 and A_2 with k_1, k_2 rows and i, j be indexes. Denote by $A_1(i) \cdot A_2(j)$ the concatenation of columns $A_1(i)$ and $A_2(j)$, i.e., the sequence with $k_1 + k_2$ elements

$$A_1[0][i], \dots, A_1[k_1 - 1][i], A_2[0][j], \dots, A_2[k_2 - 1][j].$$

Also, denote by $\llcorner^{\ell}$ the sequence of ℓ symbols equals to $\llcorner$ and suppose that $n_1, n_2 = O(N)$ for some N.

We compute the matrix also called *alignment matrix*

$$M[i,j] = \max \begin{cases} M[i-1, j-1] + \Gamma(A_1(i) \cdot A_2(j)), \\ M[i-1, j] + \Gamma(A_1(i) \cdot \llcorner^{k_2}), M[i, j-1] + \Gamma(\llcorner^{k_1} \cdot A_2(j)) \end{cases} \tag{4}$$

for each $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$ with $M[0,0] = 0$. Considering $n = n_1 + n_2$ and that each entry of M can be computed in constant time, we spend $O(n^2)$ time for compute all entries of M and since the guide tree has $O(k)$ nodes, the entire procedure spends $O(kn^2)$ time. Once M is calculated, using a similar strategy to the trace back method by Needleman-Wunsch [3], we spend $O(n)$ to obtain each profile (compatible alignment) and $O(kn)$ to obtain all profiles.

3 Parallel Algorithms

3.1 Homogeneous Hybrid Parallel Platform

In this paper, we developed an MSA algorithm based on progressive alignment strategy to execute on a hybrid parallel computing platform. That platform is based on the joint use of CPUs and GPUs in order to obtain high performance systems. Hybrid parallel platform is a two-level parallel computing model. Below we provide some features of the target hardware related to this architecture.

Suppose we have p compute nodes (**CN** for short) and each of them contains a GPU: At the upper level, we use a coarse-grained model based on Beowulf cluster, which is scalable and based on an inexpensive hardware infrastructure composed by private and dedicated interconnection network coordinated by MPI [21]. See Fig. 1. At this level a special process called master node runs in CN 0, managing the tasks of the computing nodes. On the other hand, at the lower level, we use a fine-grained model through the CUDA-enabled GPUs [22]. CUDA (Compute Unified Device Architecture) is a parallel architecture based on many-core paradigm for NVIDIA GPUs. CUDA enables programmers to write a source code and execute it on the GPU. Each GPU can have several streaming multiprocessors (SM) and each SM contains dozens or even hundreds of Single Processors (SP). All SMs access a same device global memory.

The code runs on CPU or GPU and the tasks of the computing nodes are managed by MPI. CPU creates multi-thread kernels for the GPU. GPU has its own scheduler that assigns a thread block to any SM dynamically during the execution, and the SPs within SM run the threads.

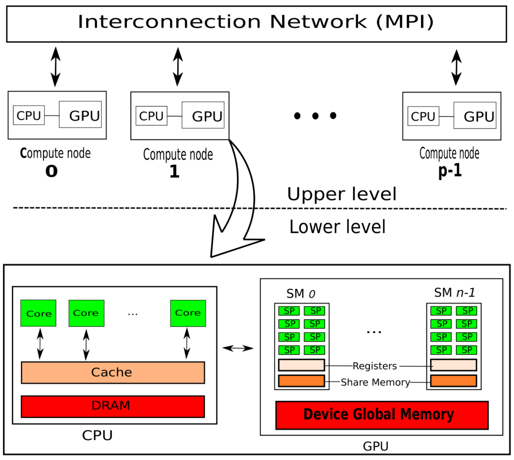


Fig. 1. Homogeneous hybrid parallel model

On CUDA programming environment we need define size of thread blocks and size of the grid, which is an abstraction for a group of thread blocks. Each thread block is assigned to an SM, and threads in a same block access a same shared memory. Moreover, the hierarchical memory consists in global memory, texture memory, shared memory and registers, where global memory is the slowest and registers are the fastest.

3.2 Parallel Algorithm

Our implementation of the MSA problem explores the two level of parallelization along the three stages of the progressive alignment. During the processing the job coordination on a CN is done by CPU which can execute local operations using CPU and GPU accordingly.

At the upper level, we use coarse-grained parallelism which manages whole process and control the distribution of the jobs to the CNs. In this level, initially the master node reads the input sequences from the disk and replicates them into the CPU memory of all CNs. Our algorithm uses a simple and efficient task allocation strategy that performs uniform load balancing between the CNs.

Let k be the number of sequences. In order to save space since we are primarily interested in computing large amount of lengthy sequences, we represent the similarity matrix D (lower triangular) by an array V of size $N = k(k-1)/2$. Each position in V represents a sequence pair to be aligned, considering the positions of the triangular matrix in lexicographic order (see Fig. 2). Assume that N is divisible by p. The array V is partitioned into p segments of size N/p, which we denote by $v_0, v_1, \ldots, v_{p-1}$. The CN i, $0 \leq i \leq p-1$, will process the elements of the segment v_i, whose positions in V are in the range $[(i)\frac{N}{p} .. (i+1)\frac{N}{p} - 1]$. Each CN i identifies the sequence pairs that will be sent to its GPU by computing (in CPU) the mapping of the elements from v_i to D, as follows:

$$l = \left\lfloor \sqrt{2(iN/p + 1)} + \tfrac{1}{2} \right\rfloor \quad \text{and} \quad c = iN/p - l(l-1)/2, \tag{5}$$

where l and c correspond, respectively, to the row and column of the element in D which is represented by the first element of v_i. Clearly, the Eq. 5 can be calculated in $O(1)$ time.

As all sequence characters belong to a small and well-defined alphabet, they can be mapped into numerical identifiers, allowing the representation of each character symbol with only 5 bits of memory (which allows to represent up to $2^5 = 32$ symbols). Hence, we can store 3 distinct sequence elements in a single 16-bit integer type using simple bit-wise operations. Effectively, this reduces by up to a third the total amount of memory required to store each sequence. Nevertheless, each sequence character can still be randomly accessed in constant time complexity with negligible overhead.

At the lower level, we use fine-grained parallelism whose jobs are computed on the CUDA-enabled GPUs. In the subsection below we detail the used approaches to perform the three stages using hybrid parallel computing.

Matrix Similarity on Multi-GPU Platform. In this stage we calculate the similarity matrix aligning pairwise sequences using a GPU version of the N-W

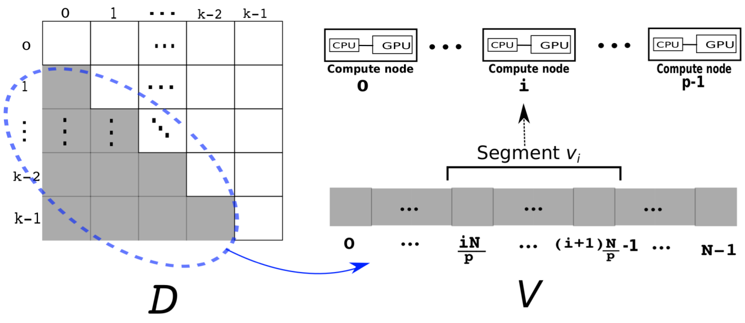


Fig. 2. Load balancing: the array V is evenly partitioned into p segments of size N/p among p computer nodes.

algorithm [3]. We align each pair of sequences in parallel on the GPU of each CN using an approach based on intra-task parallelization [23].

Each CN i receives from the master node the whole set of sequences, together with the scoring matrix γ, and identifies (through Eq. 5) the pairs v_i whose similarity value it will calculate. These data are sent to the GPU. The matrix γ is stored only once in the shared memory since it is the same for all the pairs. However, the segment v_i is sent to the GPU global memory in waves. Each element of the matrix D is computed using dynamic programming based on the Eq. 1. After aligned a set of pairs another wave of data is sent to GPU.

In this stage we save memory and gain performance by concurrently calculating (in GPUs) the similarity values using a combination of the techniques of the algorithms *DScan-mNW* and *LazyRScan-mNW* [15], as follows (see Fig. 3):

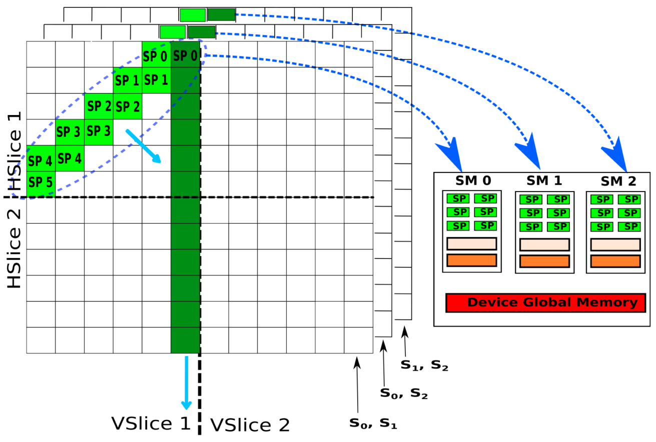


Fig. 3. Flow for calculating similarity values

Each SM stores in its shared memory its respective sequence slice and performs a diagonal scan. Each alignment matrix is iterated over the sequence in vertical slices which are computed by a thread block. Each thread acts over a row at a time. The vertical slice size is calculated so that it fits into shared memory and according to available SPs. When the thread reaches the end of the slice its value is transferred to global memory (dark green column). Then, for the next slice to resume the processing, the value of this cell is sent again to shared memory. Since our approach each thread needs only three cells at a time, instead of maintaining two contiguous rows in the shared memory we use sliding window strategy where old cells are removed from the shared memory. The similarity value is sent to the global memory as soon as they is computed.

For long sequences, we can split vertical slices in horizontal slices by forming a square mesh. As CUDA limits the number of threads per block, this improvement allows shared memory is used by more threads and hence to speedup the processing. Both combined techniques allows us to hide the access latency to global memory. However the last optimization limits the use of threads per block. On the other hand, it reduces the amount of necessary shared memory to store the similarity values being calculated. In our implementation a good equilibrium was 480 threads by block. After to compute all similarity values on GPUs these values are sent to master node where the whole matrix D is assembled.

Guide Phylogenetic Tree. Taking matrix D (previous stage) as input, we perform this stage according to the NJ algorithm [14], whose goal is to build the guide tree. Basically the NJ algorithm starts with a star tree where each leaf vertex corresponds to an OTU and iterates over the following three steps in order to joining the most similar pair of OTUs until reaches all leafs of the tree:

1. From the matrix D we have to compute each OTU pair in D by using the Formula 2. This processing generates a derived matrix in order to maintain the evolution relationship among all the OTUs.
2. By current derived matrix we choose the maximum value representing the most similar pair of OTUs. These pairs are joined to a newly created vertex which is joined the rest tree such that they form a new branch in the phylogenetic tree.
3. After joining the pair of OTUs we have to update the similarity matrix D according to Eq. 3. New row and column are created to store the similarity between the joined OTU and the remaining OTUs. Then, row and column correspondent to chosen OTUs in Step 2 are removed from the matrix D.

From parallel point of view this is a difficult task due to the data dependency among the 3 steps above, in addition there is dependency among each iteration. Despite of the large data dependency required by the method, we face the challenge to implement this stage of the alignment on Multi-GPU platform. Figure 4 illustrates the overall flow of an iteration.

Initially, the master node broadcasts the matrix D to all CNs which is copied to the GPU global memory. As soon as the CNs receive D they compute their

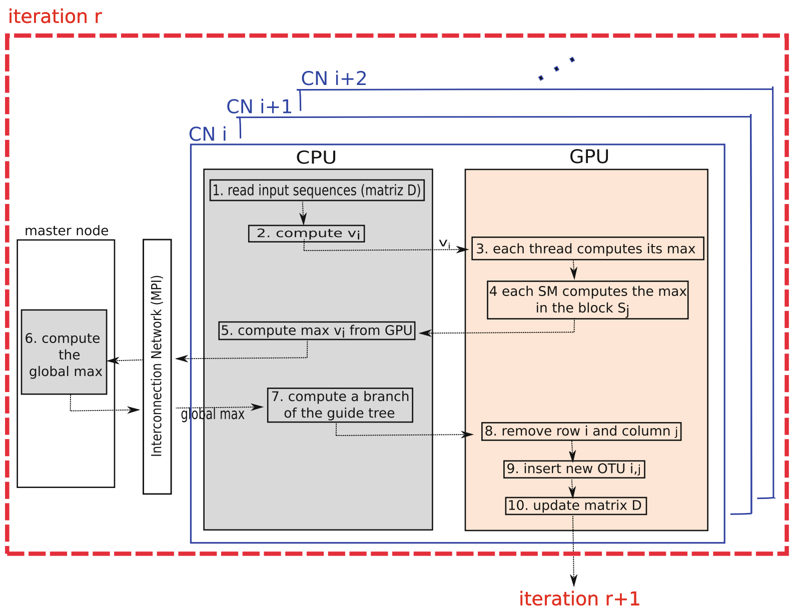

iteration r
CN i+2
CN i+1
CN i
CPU
GPU
master node
1. read input sequences (matriz D)
2. compute vi
vi
3. each thread computes its max
4 each SM computes the max in the block Sj
5. compute max vi from GPU
6. compute the global max
Interconnection Network (MPI)
7. compute a branch of the guide tree
global max
8. remove row i and column j
9. insert new OTU i,j
10. update matrix D
iteration r+1

Fig. 4. MPI-CPU-GPU flow in stage two (construction of the guide tree)

respective chunks in the same scheme that of the first stage and send to GPU. So, each GPU have to compute the range $[(i)\frac{N}{p}\ldots(i+1)\frac{N}{p}-1]$.

For the Step 1, computing Eq. 2 can be done independently within each GPU. We avoid wasting time with redundant operations by calculating the sum of the rows and columns attributed to GPU in advance since this values are used for all pairs. This values are kept in the shared memory and updated at the end of each iteration. Due to memory limit and the large amount of pairs to process we have to launch multiple kernels to compute the derived matrix, where the exact number of kernels depends on the number of sequences k, device global memory and the number of SMs available. The number of pairs are evenly distributed among the blocks.

In Step 2, choosing the largest value in D is accomplished as follows. Each thread computes its largest value and keeps it locally. After, each thread block applies a reduction and obtains the largest value of the block. Then, each block sends its values to CPU and each CPU computes its largest value. Finally MPI applies a reduction to obtain the global largest value. This value and its respective pair (i, j) are then known by all CNs. The pair (i, j) is used in each CPU to create the new node and the tree is updated on CPU.

In Step 3, we perform the computation of the branch sizes for the new node, the similarity value from the new node to the other sequences and update D. For this, each GPU receives the largest value and the pair (i, j) and has to remove row i and column j from D. In fact, these row and column are only marked as removed. We also have to calculate the similarity of newly create OTU with the remaining OTUs according to Eq. 3. Again, the Eq. 5 allows threads to access each OTU quickly. Hence we update matrix D and recalculate the sum for each of rest row/column to be used in next iteration. Note that a sum and a subtraction is enough to recalculate row/column sums instead of applying reduction.

An important improvement we implement is that of a threshold. At the end of each iteration every GPU calculates whether its workload is less a threshold. If so, we choose the last GPU to become idle and its task is redistributed.

Progressive Alignment. In this stage, we perform the alignment of the sequences according to the order provided by the guide tree. Starting from the leaves, the method works in a bottom-up way operating in parallel level by level and aligning ever larger groups of sequences until reaching the root (see Fig. 5).

Our approach keeps the guide tree in the master node's memory, which coordinates the alignments among its CNs. Each CN receives from the master node a node of the guide tree and sends its sequences to GPU. Thereafter, the GPU accomplishes a profile or sequence alignment and transfer it to its CPU which send it to the master node which stores it into CPU memory.

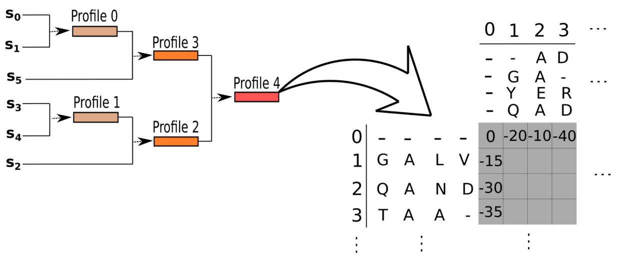


Fig. 5. Parallel progressive alignment algorithm. The guide tree lead the order of the pairwise alignments.

For all of the guide tree's nodes with height 1, several CNs run concurrently in its GPU our parallel version of the algorithm Myers and Miller [16]. Since we have to compute the alignment, not only the score, this method enables to address long sequences. In this case, we use only horizontal slices.

For the remaining nodes, note that the computation of $M[i, j]$ is based on the sum of all evolutionary distances between all possible combinations of amino acids. Hence, as we move up the tree, memory will be a bottleneck. To overcome this bottleneck we transfer only a profile and the other is sent as the slicing window technique in horizontal slices. Each GPU trhead calculates a score concurrently, for each column of the profile. These scores are projected as initialization values for the matrix M, enabling us to align the profile pair. After we join together all of their sequences into a single alignment profile.

4 Experimental Results and Discussion

We have compared our implementation, called Museqa (acronym for Multiple Sequence Aligner), with the popular aligners ClustalW-MPI and ClustalΩ. Besides, we compared the Museqa PW stage with PW_R, a CUDA N-W implementation included in the Rodinia Suite [17]. In our experiments we considered sequence datasets from NCBI and OrthoDB [24]. We have also considered three randomly generated synthetic sequence datasets called Syn20k, Syn30k and Syn40k. Table 1 presents the number of sequences and the sequence length of each dataset.

Table 1. Datasets considered in the experiments.

Datasets	Zika	Dengue	SARS-CoV2	OrthoDB	Syn20k	Syn30k	Syn40k
# of sequences	700	6,123	7,631	10,000	20,000	30,000	40,000
Average length	3,423	3,392	7,096	3,150	200	200	200

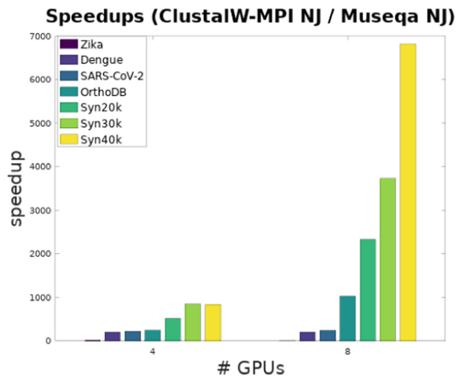
We run our experiments in two different platforms: Server A with Intel(R) Xeon(R) CPU E5-1620 3.50 GHz, 4 cores(8 threads), 64 GB RAM, 4x GeForce GTX 1080-11 GB and Server B with 2 x Intel(R) Xeon(R) CPU E5-2683 2.10 GHz, 16 cores (32 threads), 512 GB RAM, 8 x Tesla V100-16 GB.

Table 2. Comparison of Museqa run in GPUs with ClustalW-MPI run in CPUs and comparison of the stage PW of the Museqa with PW_R of the Suite Rodinia.

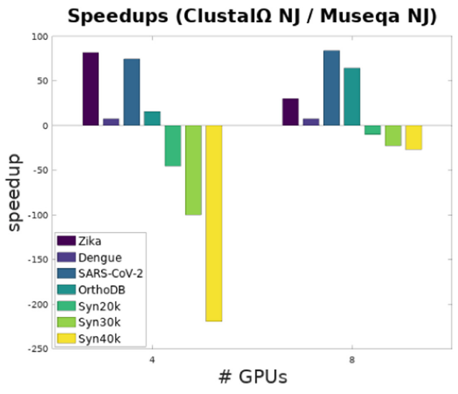
		a) ClustalW-MPI		b) ClustalΩ		c) R_S	d) Museqa		Speedups		
		#CPUs	#CPUs	#CPUs	#CPUs		#GPUs	#GPUs	a/d	b/d	c/d
Dataset	Stage	8	32	8	32	1	4	8			1
Zika	PW	2 h 10	52 m 03	31 s	15 s	30 m 58	14 s	3 s	1041	5	619
	NJ	15 s	2 s	14 s	9 s	–	0.11 s	0.3 s	6.67	30	–
	PA	2 m 53	2 m 20	16 m 34	27 m 27	–	1 m 15	58 s	2.41	17.1	–
Dengue	PW	196 h 07	64 h 17	8 m 43	3 m 40	14 h 28	18 m	5 m 37	687	0.65	155
	NJ	37 m 18	36 m 56	2 m 21	1 m 23	–	11 s	11 s	201	7.55	–
	PA	28 m 59	20 m 01	2 h 27	3 h 57	–	12 m 54	10 m 11	1.97	14,5	–
Sars-Cov2	PW	597 h 16	179 h 54	1 h 27	35 m 42	92 h 57	1 h 53	35 m 45	302	1	156
	NJ	1 h 08	1 h 05	40 m 41	22 m 19	–	18 s	16 s	244	84	–
	PA	28 m 58	26 m 09	14 h 24	19 h 59	–	15 m 34	14 m 18	1.83	60.8	–
OrthoDB	PW	489 h	163 h 01	44 m 45	22 m 22	108 h	24 m 27	7 m 48	1253	2.86	830
	NJ	2 h 45	2 h 33	10 m 30	7 m 30	–	29 s	7 s	1031	64.3	–
	PA	129 h 20	46 h 11	47 h 15	94 h 30	–	5 m 22	2 m 36	1065	1090	–
Syn20k	PW	2 h 51	1 h 18	32 s	14 s	3 h 20	1 m 55	32 s	146	0.44	375
	NJ	35 h 10	32 h 24	7 s	5 s	–	3 m 44	50 s	2333	0.1	–
	PA	8 h 55	3 h 11	8 h 5	9 h 4	–	2 h 17	1 h 06	2.89	7.35	–
Syn30k	PW	6 h 24	2 h 03 m	51 s	21 s	5 h 22	4 m 36	1 m 17	96	0.27	251
	NJ	165 h 35	163 h 46	11 s	7 s	–	11 m 35	2 m 38	3731	0.04	–
	PA	30 h 01	9 h 45	10 h 45	20 h 23	–	8 h 11	3 h 37	2.69	5.63	–
Syn40k	PW	11 h 28	4 h 34	1 m 12	21 s	7 h 25	8 m 54	2 m 23	115	0.15	187
	NJ	385 h	356 h	15 s	7 s	–	25 m 33	3 m 08	6817	0.04	–
	PA	90 h	27 h 12	16 h	29 h 30	–	15 h 29	10 h 56	2.49	2.70	–

Table 2 shows the runtimes of the three stages (PW, NJ and PA) using Server A and Server B by ClustalW-MPI, ClustalΩ and our Museqa. In addition, it shows the runtimes obtained by NW_R using Server A. For those tests we use as input Zika Virus, Dengue Virus, SARS-CoV2, OrthoDB, Syn20k, Syn30k and Syn40k datasets. We observe that ClustalΩ implements different strategies for each step when compared to Museqa and ClustalW-MPI. We calculated the average time over 3 executions for each test. Comparing with ClustalW-MPI, the first relevant fact is that the stage PW dominates at least 95% of the time in all NCBI dataset scenarios, while for synthetic datasets the opposite occurs: the time spent by NJ becomes significant, surpassing the PW runtime.

The scalability of Museqa for all stages combined in terms of number of GPUs were almost linear for all datasets considered. Note that, even though NJ stage did not achieve linear speedups it still performed better with multiple GPUs, even for synthetic datasets where the time spent in NJ stage was larger than in PW stage according to Table 2. Figure 6 illustrates the speedups between ClustalW-MPI and Museqa and between ClustalΩ and Museqa regarding NJ stage only for 4 and 8 GPUs (32 CPUs for ClustalW-MPI and ClustalΩ), highlighting the remarkable performance of Museqa compared to ClustalW-MPI for all seven datasets, and specially for the synthetic datasets for which the time consumption of NJ is much more significant than for NCBI datasets. Considering the comparison between ClustalΩ NJ and Museqa NJ, Museqa NJ performed better for real datasets (NCBI and OrthoDB), but ClustalΩ NJ had superior performance for synthetic datasets (only marginally for Museqa with 8 GPUs).



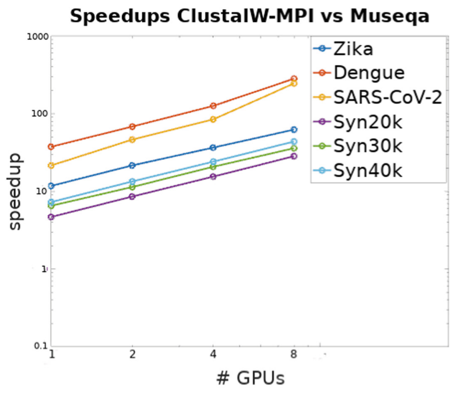
(a) ClustalW-MPI NJ / Museqa NJ



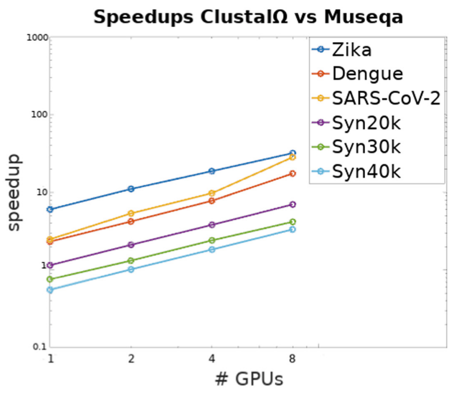
(b) ClustalΩ NJ / Museqa NJ

Fig. 6. Speedups between: a) ClustalW-MPI NJ and Museqa NJ; b) ClustalΩ NJ and Museqa NJ; according to Table 2. Each negative value ($-X$) in (b) means that ClustalΩ NJ was X times faster than Museqa NJ.

Considering the total time spent of all three stages combined, as shown in Fig. 7, Museqa with 8 GPUs was between 28.5 to 283 times faster than ClustalW-MPI with 32 CPUs, while Museqa with 8 GPUs had speedups ranging from 2.3



(a) ClustalW-MPI / Museqa



(b) ClustalΩ / Museqa

Fig. 7. Overall speedups (log-log scale) between ClustalW-MPI vs Museqa and ClustalΩ vs Museqa, considering 32 CPUs for ClustalW-MPI and ClustalΩ.

to 32 when compared to ClustalΩ with 32 CPUs. In addition, Fig. 7 highlights that all speedups increased almost linearly with the number of GPUs. Finally, the comparison between PW_R, which is also implemented in GPU, and Museqa PW stage performed with 8 GPUs revealed that Museqa was about two orders of magnitude faster than PW_R (speedups between 155 and 830).

Alignment Accuracy: In order to measure the accuracy of the alignments produced by Museqa, we use BAliBASE [18], which is the most widely used benchmark test sets of reference alignments. We compute BAli scores (SP and TC, which measure the alignment accuracy, ranging from 0 to 1, where 1 indicates the best possible accuracy) for Museqa, ClustalΩ and ClustalW, considering 386 alignments, which are organized in 6 BAli families covering six different situations (RV11, RV12, RV20, RV30, RV40, and RV50).

Table 3. SP and TC score average results for ClustalΩ, our proposed method (Museqa), and ClustalW, for the six considered BAli families.

	RV11		RV12		RV20		RV30		RV40		RV50	
	SP	TC	SP	TC	SP	TC	SP	TC	SP	TC	SP	TC
ClustalΩ	0.48	0.27	0.83	0.68	0.82	0.34	0.69	0.38	0.76	0.43	0.70	0.35
Museqa	0.41	0.21	0.79	0.61	0.78	0.24	0.57	0.16	0.61	0.28	0.56	0.19
ClustalW	0.48	0.24	0.80	0.64	0,79	0.26	0.62	0.25	0.65	0.30	0.62	0.27

As can be seen in Table 3, the obtained accuracy is very similar, proving the reliability of the alignments obtained by Museqa when compared to ClustalΩ and

CrustalW. However, mean values can suppress nuances regarding the differences found in each alignment. In this sense, we comparison of the measurements of scores between the three algorithms in each of the 386 alignments, and we clustered into three groups: group1 - when Museqa obtained a better score; group2 - when there was a tie between the scores; group3 - when the score obtained by Museqa was significantly lower. The group2 formed by comparisons in which the fold change between the scores is less than or equal to 1.3. And the group3 by fold change in scores greater than 1.3.

Comparing Museqa and ClaustralΩ, we notice that the SP-score possesses 18.6%, 73.8%, and 7.5% of the comparisons in group1, group2, and group3, respectively. And the TC-score holds 16.3%, 68.9%, and 14.7% of the comparisons in group1, group2, and group3, respectively. Comparing Museqa and ClustalW, the SP-score holds 36.5%, 60.3%, and 3.1% in group1, group2, and group3, respectively. And the TC-score keeps 25.3%, 70.4%, and 4.1% of the comparisons in group1, group2, and group3, respectively. These results demonstrate a high degree of congruity between the three programs.

5 Final Remarks

In this paper we described a Multi-GPU solution for the Multiple Sequence Alignment problem which implements the three stages of the progressive alignment method. All stages of the method achieved significant speedup when compared to some popular tools that are currently available for the same task.

As future work, we intend to look into three directions: i) to use a modified NJ which produces a complete binary tree, reducing the interdependence of the computations in stages 2 and 3; ii) to implement the aligments of the method using new biological information iii) to make improvements in the proposed method so that it can operate in heterogeneous hybrid parallel platforms.

Acknowledgments. We thank the High Performance Computing Center (NPAD/ UFRN) and CTEI/UFMS for providing computational resources, and grants #2018/18560-6, #2018/21934-5, São Paulo Research Foundation (FAPESP) for financial support.

References

1. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. J. Mol. Biol. **215**(3), 403–410 (1990)
2. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. J. Mol. Biol. **147**(1), 195–197 (1981)
3. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. **48**(3), 443–453 (1970)
4. Katoh, K., Misawa, K., Kuma, K., Miyata, T.: MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform. Nucleic Acids Res. **30**(14), 3059–3066 (2002)

5. Larkin, M.A., et al.: Clustal W and Clustal X version 2.0. Bioinformatics **23**(21), 2947–2948 (2007)
6. Lassmann, T.: Kalign 3: multiple sequence alignment of large datasets. Bioinformatics **36**(6), 1928–1929 (2020)
7. Zhang, C., Zheng, W., Mortuza, S.M., Li, Y., Zhang, Y.: DeepMSA: constructing deep multiple sequence alignment to improve contact prediction and fold-recognition for distant-homology proteins. Bioinformatics **36**(7), 2105–2112 (2020)
8. Bonizzoni, P., Della Vedova, G.: The complexity of multiple sequence alignment with SP-score that is a metric. Theoret. Comput. Sci. **259**(1), 63–79 (2001)
9. Thompson, J.D., Linard, B., Lecompte, O., Poch, O.: A comprehensive benchmark study of multiple sequence alignment methods: current challenges and future perspectives. PloS One **6**, e18093 (2011)
10. Li, K.-B.: ClustalW-MPI: ClustalW analysis using distributed and parallel computing. Bioinformatics **19**(12), 1585–1586 (2003)
11. Sievers, F., et al.: Fast, scalable generation of high-quality protein multiple sequence alignments using Clustal Omega. Mol. Syst. Biol. **7**, 539 (2011)
12. Alawneh, L., Shehab, M.A., Al-Ayyoub, M., Jararweh, Y., Al-Sharif, A.Z.: A scalable multiple pairwise protein sequence alignment acceleration using hybrid CPU-GPU approach. Cluster Comput. **23**, 2677–2688 (2020)
13. Araujo, E., Stefanes, M.A., Ferlete, V.O., Rozante, L.C.S.: Multiple sequence alignment using hybrid parallel computing. In: 17th IEEE International Conference on Bioinformatics and Bioengineering, pp. 175–180 (2017)
14. Saitou, N., Nei, M.: The neighbor-joining method: a new method for reconstructing phylogenetic trees. Mol. Biol. Evol. **4**(4), 406–425 (1987)
15. Truong, H., Li, D., Sajjapongse, K., Conant, G., Becchi, M.: Large-scale pairwise alignments on GPU clusters: Exploring the implementation space. J. Sig. Process. Syst. **77**(1–2), 131–149 (2014)
16. Myers, E.W., Miller, W.: Optimal alignments in linear space. Comput. Appl. Biosci. CABIOS **4**(1), 11–17 (1988)
17. Che, S., et al.: Rodinia: a benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC), pp. 44–54 (2009)
18. Thompson, J.D., Koehl, P., Ripp, R., Poch, O.: BAliBASE 3.0: latest developments of the multiple sequence alignment benchmark. Proteins Struct. Funct. Bioinf. **61**(1), 127–136 (2005)
19. Hogeweg, P., Hesper, B.: The alignment of sets of sequences and the construction of phyletic trees: an integrated method. J. Mol. Evol. **20**(2), 175–186 (1984)
20. Feng, D.-F., Doolittle, R.F.: Progressive sequence alignment as a prerequisite to correct phylogenetic trees. J. Mol. Evol. **25**(4), 351–360 (1987)
21. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press (1999)
22. Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs. Elsevier (2012)
23. Liu, Y., Schmidt, B., Maskell, D.L.: MSA-CUDA: multiple sequence alignment on graphics processing units with CUDA. In: 20th IEEE ASAP, pp. 121–128 (2009)
24. Zdobnov, E.M., et al.: OrthoDB in 2020: evolutionary and functional annotations of orthologs. Nucleic Acids Res. **49**, D389–D393 (2021)