2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Constructing Similarity Graphs from Large-scale Biological Sequence Collections

Jaroslaw Zola
Rutgers Discovery Informatics Institute
Rutgers University
Piscataway, NJ 08854, USA
Email: jaroslaw.zola@rutgers.edu

***Abstract*—Detecting similar pairs in large biological sequence collections is one of the most commonly performed tasks in computational biology. With the advent of high throughput sequencing technologies the problem regained significance as data sets with millions of sequences became ubiquitous. This paper is an initial report on our parallel, distributed memory and sketching-based approach to constructing large-scale sequence similarity graphs. We develop load balancing techniques, derived from multi-way number partitioning and work stealing, to manage computational imbalance and ensure scalability on thousands of processors. Our experimental results show that the method is efficient, and can be used to analyze data sets with millions of DNA sequences in acceptable time limits.**

***Keywords*-sequence similarity; min-wise independent permutations; load balancing; parallel computational biology;**

I. Introduction

Detecting similar pairs in large biological sequence collections is arguably one of the most frequently performed tasks in computational biology. It is an important prerequisite to multiple sequence alignment [1], distance based phylogenetic inference [2], biological databases management [3], and metagenomic clustering [4], among many others. In recent years the problem gained a new momentum due to the data deluge triggered by the next generation sequencing technologies. With the increasing sequencing throughput and the decreasing cost, data sets with millions of reads became routinely available. This renders many traditional methods, that depend on the exhaustive all pairs comparison, computationally infeasible.

In this paper, we provide an initial report on our new parallel method for large-scale sequence similarity graphs construction. The method uses approach in which candidate pairs are generated by the sketching technique [5], [6], and the final graph is obtained by validating candidate pairs. To guarantee scalability on large distributed memory machines we propose two different strategies to address load balancing challenges in different phases of the algorithm. These strategies include the multi-way number partitioning [7] and work stealing [8], and scale to problems with millions of sequences on thousands of processors. To demonstrate efficiency of the resulting software we provide experimental results with the actual biological data from the Human Microbiome Project.

The reminder of this paper is organized as follows. In Section II, we provide a formal definition of the similarity graph construction problem. In Section III, we briefly introduce a sequential solution that uses sketching technique to accelerate candidate sequence pairs enumeration. We follow up with a detailed presentation of our parallel approach in Section IV, and experimental results in Section V. We conclude the paper with a brief outlook in Section VI.

II. Problem Formulation

The problem of constructing a similarity graph from a large set of sequences can be formulated as follows: Consider a set $S = \{s_0, s_1, \ldots, s_{n-1}\}$ of DNA or protein sequences, and a symmetric pairwise similarity function $F : S \times S \to \mathbb{R}^+$. We want to find graph $G = (V, E)$, where $V = \{v_0, v_1, \ldots, v_{n-1}\}$ is a set of nodes such that $v_i \in V$ corresponds to sequence $s_i \in S$, and pair $e = (v_i, v_j)$ becomes an edge in E only if $F(s_i, s_j) > t$. Here, t is a predefined threshold above which two sequences are considered similar. The function F is usually derived from a pairwise sequence alignment, for example it can be a fraction of identities shared by two sequences, or it can be based on one of alignment-free methods [9]. Finally, we are focusing on cases where n is of order $10^5 - 10^6$.

For a given S we can construct graph G by first comparing all $\binom{n}{2}$ sequence pairs, and then pruning those with similarity below t. This direct approach is well studied, and has several implementations targeting especially accelerator architectures [10]. Unfortunately, it is computationally infeasible for any but relatively small data sets: suppose that to compute F we could use a highly optimized dynamic programming algorithm capable of 10×10^9 dynamic programming matrix cell updates per second. If our input consisted of 1M sequences with average length of 250 characters (nucleotides or amino acids), then it would take roughly 36 days to perform all pairs comparison. To put it in the context, data sets that commonly appear in metagenomics and metaproteomics can easily contain more than 5M much longer sequences (see for example the CAMERA database [11]), and often have to be analyzed instantly, e.g. in applications related to biological threat detection. This clearly shows that a better strategy is needed.

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.63

III. Sketching-Based Approach

Although the cost of directly constructing a similarity graph is prohibitive, in a typical scenario the graph is very sparse even for small values of t. In other words, for a given set S and threshold t, the majority of function F evaluations do not introduce edges to the resulting similarity graph. Consequently, by evaluating F only for those sequence pairs that have a high probability of being similar it is possible to significantly reduce the overall cost of graph construction. Of course, this requires an efficient strategy to identify candidate pairs, which at the same time has to be sensitive so that no similar sequences are missed. Several different variants of this approach have been pursued by researchers. For example, Holm and Sander [3] introduced the "decapeptide filter" to identify redundant proteins in large databases, and Kalyanaraman *et al.* [12] used suffix trees to efficiently enumerate candidate pairs for the ESTs clustering. Both these approaches exploit the fact that similar sequences must share a common substring of length k, a k-mer, where k is a parameter.

In this paper, we built on top of the approach developed by Yang *et al.* for metagenomic clustering [4]. Instead of identifying pairs that share just a single k-mer, the method efficiently approximates the fraction of shared k-mers between two sequences by means of min-wise independent permutations [13]. The fraction of shared k-mers, which we will call *k-mer similarity*, has been demonstrated to closely reflect the evolutionary distance between sequences [9], [14]. Consequently, in a wide range of sequence homology it can be used to bound alignment-based similarity functions. Let $\mathcal{S}_i$ be a k-mer spectrum of sequence $s_i \in S$, i.e. a set of all k-mers in s_i. Then, the fraction of shared k-mers between sequences s_i and s_j is equal to the Jaccard index between their k-mer spectra: $J(\mathcal{S}_i, \mathcal{S}_j) = \frac{|\mathcal{S}_i \cap \mathcal{S}_j|}{|\mathcal{S}_i \cup \mathcal{S}_j|}$. To capture containment it is often more convenient to consider a slightly modified measure $\mathcal{C}(\mathcal{S}_i, \mathcal{S}_j) = \frac{|\mathcal{S}_i \cap \mathcal{S}_j|}{\min(|\mathcal{S}_i|, |\mathcal{S}_j|)}$. In [5] Broder proposed an unbiased estimator of Jaccard index, as well as containment, that in the essence can be written as $J^*(\mathcal{S}_i, \mathcal{S}_j) = \frac{|H_i \cap H_j|}{|H_i \cup H_j|} = J(H_i, H_j)$ and $\mathcal{C}^*(\mathcal{S}_i, \mathcal{S}_j) = \mathcal{C}(H_i, H_j)$. Here, H_i is a set of sketches extracted from $\mathcal{S}_i$, and it is defined as follows. Let h be a hashing function that maps a k-mer into an integer, and let X_i be a set of all hashed k-mers from sequence s_i, i.e. $X_i = \{x_j \mid \forall_{s_j \in \mathcal{S}_i}\ x_j = h(s_j)\}$. For a given integer parameter M, H_i contains all hashes from X_i that are $0 \mod M$: $H_i = \{x \in X_i \mid x \mod M = 0\}$. It is easy to see that a set of sketches must be significantly smaller than the original set for any reasonable choice of M. Consequently, sketches can be used to sort the original sets such that sets that share sketches fall into the same bin. Then, Jaccard index can be estimated while avoiding the $O(n^2)$ cost of comparing all original sets against each other. This in turn translates directly into an efficient method for identifying pairs of sequences with high *k-mer similarity*. The resulting approach is summarized in Algorithm 1.

Algorithm 1 Sketching-based similarity graph generation

Input: S, F, t and M, k, t_{min}
Output: G

1: For each $s_i \in S$ generate spectrum $\mathcal{S}_i$
2: For each $\mathcal{S}_i$ extract sketches H_i
3: Group sequences with the same sketch
4: For each pair (s_i, s_j) sharing sketch compute $J(H_i, H_j)$
5: If $J(H_i, H_j) > t_{min}$ compute $F(s_i, s_j)$
6: If $F(s_i, s_j) > t$ add edge (v_i, v_j) to G

Note: All symbols are explained in the main text.

The algorithm proceeds in two stages: first, candidate pairs are identified using the sketching technique (lines 1–4). Two sequences form a candidate pair if approximated Jaccard index of their k-mer spectra is above the threshold t_{min}. Next, candidate pairs are validated by computing function F, and checking whether the resulting similarity is above the target threshold t (lines 5–6). Candidate pairs that pass this test are added as edges to G. Yang *et al.* [4] demonstrated that the sketching-based approach can reduce the computational work to only few percent of all pairs comparisons, while maintaining 99% sensitivity or higher.

The trade-off between computational cost and sensitivity can be controlled by two additional parameters required by the method: M and t_{min}. The parameter M decides how many sketches will be generated for each sequence. Small values of M result with many sketches extracted per sequence, and thus better *k-mer similarity* estimates (hence improved sensitivity). However, this comes at the cost of generating many candidate pairs rejected during validation. The parameter t_{min} is tightly related to the threshold t. Because we use *k-mer similarity* as a bound for F, t_{min} can be set by, for example, experimentally checking a relation between *k-mer similarity* and F. Of course, if we use *k-mer similarity* as F, then t_{min} must be less than t. In either case t_{min} can be tuned to compensate for a possible underestimation of the Jaccard index by the sketching technique. The parameter k is well understood, e.g. in the context of alignment-free sequence comparison methods, and typically $k \geq 15$ for DNA, and $k = 4$ or $k = 5$ for protein sequences. When set to one of these common values, this parameter has a minor impact on the performance of the method.

Finally, we note that to further increase sensitivity, candidate pairs can be identified in several iterations, where in iteration $j = 0 \ldots M-1$ the set of sketches for s_i becomes $H_i = \{x \in X_i \mid x \mod M = j\}$.

IV. Parallel Approach

While the sketching technique significantly reduces the cost of similarity graph generation, the problem remains

challenging. This is because the number of candidate pairs that have to be validated can easily exceed hundreds of millions, which poses nontrivial memory and computational requirements. Moreover, the problem is highly irregular owing to varying length of input sequences, and an unpredictable structure of the output graph.

To enable processing of data sets with millions of sequences in reasonable time limits, we developed a new scalable MPI-based approach. Our method proceeds in two tightly coupled steps that correspond to candidate pairs generation and validation stages in Algorithm 1. To ensure scalability we completely distribute both input and output data, and we develop load balancing approaches based on the multi-way number partitioning and efficient work stealing.

A. Candidate Pairs Generation

The process of candidate pairs generation can be summarized as follows: first, we extract sketches and pair them with sequences from which they have been extracted. Then, we sort such obtained pairs using sketch as a key. We follow with enumerating all pairs of sequences that share a sketch. Finally, we perform reduction on this list to obtain the final list of candidate pairs.

1) Extracting sketches: Let p be the number of processors. We start by distributing S such that processor $i = 0, \ldots, p-1$ receives a subset of sequences $S^i = \left\{ s_{\frac{in}{p}}, \ldots, s_{\frac{(i+1)n}{p}-1} \right\}$. Each sequence is assigned a unique identifier, which is its index in S. This identifier will enable us to instantly find a processor on which given sequence is stored. Next, we proceed to deriving sketches. This involves a linear scan of all sequences and can be performed independently by each processor. As a result each processor creates a list of all extracted sketches, in which every sketch forms a tuple (x, r, d). Here, x is the actual sketch represented by a 64-bit hash computed using function h (in our case Murmur2 [15]), r is an identifier of a sequence from which x has been extracted, and d is the total number of sketches extracted from sequence s_r. Note that we store d to quickly compute containment, which is a preferred measure in the majority of real-life applications. Because the length of sequences stored on different processors can vary, we expect a slight imbalance between processors during sketching. However, this step is computationally extremely fast, and hence the resulting imbalance can be safely ignored.

2) Grouping sketches: Given sketch-sequence pairings we can now proceed to identifying sequences that share sketches. To do this we first perform all-to-all data exchange such that processor i receives all tuples for which $x \mod p$ is i. Let L^i be a list of all tuples assigned to the processor i and sorted using sketch x as a key. At this stage each processor could simply enumerate all sequence pairs prescribed by every sketch it stores. This approach however would result in great computational imbalance, and additionally could quickly exhaust the main memory. To see why observe that it is possible that some k-mers will be shared by all sequences, while some others will be uniquely present in just one sequence. This bias will be reflected in the distribution of sketches, with some sketches common to many sequences, and some to only a few. Consequently, it could happen that for a single sketch we would enumerate all $\binom{n}{2}$ sequence pairs, thus defying the purpose of the sketching procedure. To eliminate such a possibility each processor removes from L^i all sketches shared by more than C_{max} sequences, and stores them in an auxiliary list A^i, this time however as a pair (r, x). Once L^i is pruned we can redistribute it between processors to balance computational load. Let $L^i(x) \subseteq L^i$, $|L^i(x)| \leq C_{max}$, be a set of all tuples in L^i with sketch x. Notice that for sketch x we have to enumerate $l = \binom{|L^i(x)|}{2}$ corresponding sequence pairs. Hence, we can think about $L^i(x)$ as an input data to *pair enumeration task* with cost l. Here, cost refers not only to computations but also to the memory, as each enumerated pair has to be stored. To achieve the ideal load balancing the total cost of *pair enumeration tasks* should be the same on all processors. This is equivalent to the multi-way number partitioning problem in which one wants to partition a set of integers such that the sum of numbers in the resulting subsets is equal [7]. Therefore, we developed the following load balancing procedure. Each processor sends information about its *pair enumeration tasks* to processor 0, which runs a simple greedy strategy to find the best assignment of tasks to processors (the largest tasks are iteratively assigned to the least loaded processors). It then broadcasts information about which processors should swap which tasks, and all processors exchange parts of L^i accordingly, which concludes the procedure. Note that although the multi-way number partitioning problem is $\mathcal{NP}$-hard the greedy approach, which is often used in scheduling, performs very well in practice, especially when the number of processors p is large. It has the additional advantage that it can be very efficiently implemented using e.g. a Fibonacci heap.

Figure 1 shows an example of how our procedure performs in practice. In the middle plot, we can see that the total cost of tasks on the most loaded processor is reduced approximately by a factor of two. In spite of this, processors remain in a slight but significant imbalance. This imbalance is triggered by an unavoidable skew in the tasks cost distribution, which becomes more visible with the increasing number of processors. To mitigate this effect once *pair enumeration tasks* are redistributed we decompose large tasks into a set of smaller but manageable subtasks. Notice that we can think about sequence pairs enumeration for sketch x as filling in an $|L^i(x)| \times |L^i(x)|$ upper (lower) triangular matrix, or equivalently two upper (lower) triangular matrices and one square matrix, each of size $\frac{|L^i(x)|}{2} \times \frac{|L^i(x)|}{2}$. Following this intuition we decompose $L^i(x)$ into three subsets, each corresponding to one of the

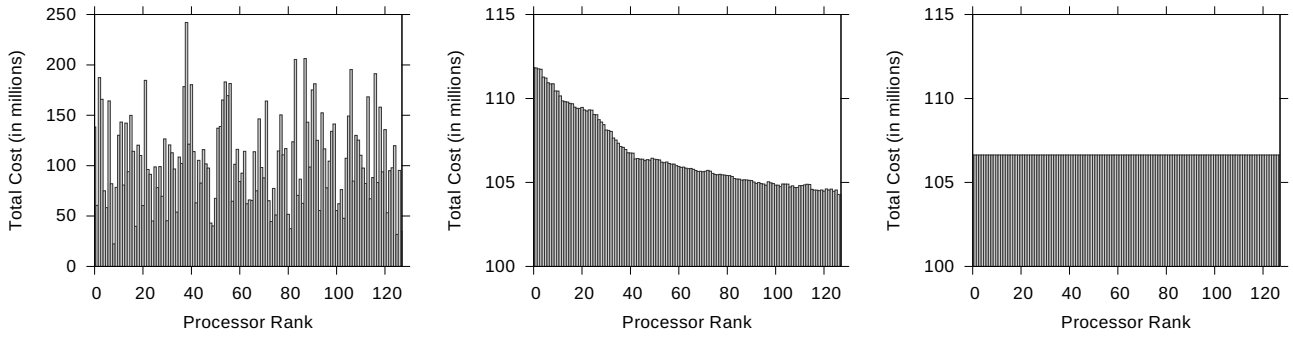

Figure 1. The total cost of *pair enumeration tasks* on different processors before load balancing (left), after load balancing (middle), and after tasks decomposition (right). $p = 128$, $n = 1{,}250{,}000$ and $C_{max} = 10{,}000$. Note that y-axis in the first plot has different scale.

three matrices. A subset related to the square matrix contains the same elements as the original set but its associated cost is $\frac{|L^i(x)|^2}{4}$. Two other subsets simply contain half of the elements of the original set. Newly created tasks can be recursively divided until certain assumed size is reached, with the exception that tasks related to square matrices have to be split into four subtasks. Once tasks decomposition is completed we repeat balancing via the multi-way number partitioning. The right plot in Figure 1 shows that with this additional step our procedure is able to achieve a near perfect load balance. Because tasks decomposition is performed after the initial balancing, we avoid memory congestion by newly generated subtasks. Finally, in all our tests the load balancing procedure took significantly less than 1% of the entire candidate pairs enumeration step, which confirms its feasibility.

3) Enumerating candidate pairs: With L^i balanced we proceed to generating candidate pairs. The general idea is to first enumerate sequence pairs prescribed by every sketch, then to perform a reduction to count how many sketches are shared by each pair, and then to compute containment $\mathcal{C}$ for each pair. The first step can be carried out independently by each processor. Consider two tuples (x, r_a, d_a) and (x, r_b, d_b) from $L^i(x)$, $r_a \neq r_b$. Since s_{r_a} and s_{r_b} share x they form a sequence pair that we represent by a tuple $(r_a, r_b, d_{ab} = \min(d_a, d_b))$ if $r_a \leq r_b$, and (r_b, r_a, d_{ab}) otherwise. Let W^i be a list of all sequence pairs enumerated on processor i. To locally count how many sketches are shared by each pair, it is sufficient to sort W^i, and perform a linear scan to merge the same pairs while counting how many times given pair occurred in W^i. Having W^i locally reduced we can perform a global reduction, to obtain the total sketch count for each pair. To achieve this, processors first perform all-to-all exchange such that tuple (r_a, r_b, d_{ab}) is assigned to processor $i = r_a \oplus r_b \mod p$, and then repeat the local reduction step. Here, using a combination of XOR and modulo operations ensures a fairly even redistribution of sequence pairs. Note that although this step induces processors synchronization, this is not a problem since L^i is balanced. The last operation we have to perform is to update the sketch count for each sequence pair. Recall that initially we discarded information about sketches contained in more than C_{max} sequences. These sketches however should still contribute to the final containment score for each sequence pair. Therefore, each processor broadcasts its sorted auxiliary list A^i, and then for every tuple in W^i it runs a binary search over the aggregate of all auxiliary lists. In practice this requires parallel sorting followed by all-to-all exchange of A^i. The update to the sketch count for tuple (r_a, r_b, d_{ab}) is equal to the size of the intersection between the subset of the auxiliary list containing r_a and the subset containing r_b. In other words, we check how many times r_a and r_b share a sketch in the auxiliary list. Let c_{ab} be the updated sketch count. Then, $\mathcal{C}(H_{r_a}, H_{r_b}) = \frac{c_{ab}}{d_{ab}}$ and the tuple becomes a candidate pair if $\mathcal{C}(H_{r_a}, H_{r_b}) > t_{min}$. Using this condition each processor prunes its list W^i, which now becomes the list of candidate pairs.

B. Candidate Pairs Validation

To construct the final graph G we have to validate all candidate pairs. Recall, that the set of input sequences S is evenly distributed between p processors: $S = S^0 \cup S^1 \cup \ldots \cup S^{p-1}, S^i \cap S^j = \emptyset, i \neq j$. Let $g(r)$ be the function returning index of the processor with sequence s_r, i.e. $s_r \in S^{g(r)}$. Consider now a candidate pair $(r_a, r_b) \in W^i$ on processor i (we drop the third element of the tuple for convenience). In order to validate this pair we have to compute $F(s_{r_a}, s_{r_b})$, which means that we have to access both sequences. This leads to three possible cases:

1) $g(r_a) = i$ and $g(r_b) = i$, hence F can be evaluated directly by the processor i.
2) $g(r_a) = i$ and $g(r_b) \neq i$, or $g(r_a) \neq i$ and $g(r_b) = i$, which means that to compute F one of the sequences has to be fetched from a remote processor.
3) $g(r_a) \neq i$ and $g(r_b) \neq i$, in which case either both sequences have to be fetched, or the candidate pair has to be sent to the processor $g(r_a)$ leading to Case 1 or Case 2.

Among these three cases, Case 1 is clearly the most desired, while Case 3 would require a significant communication overhead. Unfortunately, as the number of processors grows, Case 1 becomes less probable. Keeping this in mind we designed the following strategy. We redistribute W^i such that the candidate pair (r_a, r_b) is assigned to the processor $g(r_a)$ if $r_a + r_b \mod 2 = 0$, and to $g(r_b)$ otherwise. In this way we eliminate Case 3 as now each candidate pair is stored on the same processor as at least one of its component sequences. For each candidate pair in W^i we create a corresponding task that is placed in a local FIFO queue on processor i. Tasks are inserted such that in the front of the queue are tasks falling into Case 1, and in the back are tasks matching Case 2. Additionally, Case 2 tasks that have to fetch a sequence from the same processor are not interleaved with any other tasks. Note that by organizing computations in this way we are able to efficiently use work stealing, which we explain next. Once all tasks are placed in the respective queues, processors start execution immediately. When processor i encounters a task matching Case 2 for which input sequence has not been prefetched, it looks up a target processor storing that sequence. Because tasks in the queue are organized with respect to their target processor, i can execute one communication to obtain all sequences from a given target. As a result, the remaining tasks can be processed as if they were Case 1 tasks. Although the above procedure is sufficient to complete candidate pairs validation, it is not scalable. This is again due to the computational imbalance coming from the varying size of the task queue on each processor, and the varying cost of each task, which depends on the length of input sequences. To address this challenge we extended our queuing system with a work stealing capability.

1) Work stealing protocol: Work stealing techniques have been demonstrated as a scalable and efficient way to perform dynamic load balancing in both shared and distributed memory regimes [8], [16]. In our case work stealing is particularly well suited as all tasks are independent, and once the task queue is initialized no new tasks are generated. This simplifies the queue management and the termination detection procedure.

At the very high level our work stealing mechanism can be explained as follows. As soon as processor i is done with executing tasks in its local queue it starts work stealing. It randomly identifies a victim processor and initiates communication to check weather the victim's task queue in not empty. If this is the case, i extracts a set of tasks from the back of the queue together with associated input sequences that are local to the victim. It then executes these tasks locally as Case 1 or Case 2 tasks. If the victim's queue is empty, i removes that victim from its list of potential victims, and selects another target. This process is repeated until all processors report empty queue, which is our termination condition.

One challenging aspect of any work staling protocol is its implementation. The data prefetching required by Case 2 tasks can be easily implemented using MPI one-sided communication (each processor exposes its set S^i for RMA access). Similarly, to implement work stealing each processor could expose its task queue for one-sided communication. This however would require an RMA lock-free queue implementation to support concurrent updates by local and remote processors, especially when two different processors select the same victim. Although new MPI-3 standard introduced necessary API (e.g. MPI_COMPARE_AND_SWAP function), the majority of vendors do not support it yet. Therefore, we implemented a slightly more sophisticated solution on top of MPI-2.

Before starting tasks execution each processor initializes a work stealing request handler, by issuing a non-blocking receive from any source. After each completed task, processor calls the handler to test weather a message indicating a steal request arrived. If such a message arrived the processor answers by sending either a location in its one-sided communication buffer from which the thief can obtain tasks, or the empty queue flag if no tasks remain. Once the answer is sent the processor again issues a non-blocking receive from any source. Note that this procedure inherently serializes the access to the processor's local queue, hence eliminating race conditions and locks. To steal tasks processor initiates a non-blocking send that caries its steal request, and then switches between testing weather this message has been delivered, and calling its work stealing request handler to answer potential incoming steal requests (which then would be answered with the empty queue flag). Once the request message is delivered, the processor receives information about the status of the victim's queue, and if there are tasks to steal it proceeds with one-sided communication to obtain them. To terminate, each processor checks whether it sent $p-1$ empty queue answers. If that is the case it cancels the last non-blocking receive, and finalizes the validation stage.

As a final remark, we note that by interleaving tasks execution with calls to the work stealing request handler we induce progress of non-blocking messages in the main MPI thread. Consequently, the message exchange overhead becomes negligible and does not hinder the overall scalability.

V. Experimental Results

We implemented our parallel approach in the ELaSTIC package. This tool supports several popular sequence similarity functions, including global and local alignment, and can be used to analyze both DNA and protein sequences. All tools in the package are written in C++ and MPI-2, and are freely available from http://www.jzola.org/elastic.

We tested ELaSTIC's scalability using a collection of 16S rRNA sequences from the Human Microbiome Project. From the SRP002395-7514 data set [17] we selected all

Table I
SUMMARY OF DATA SETS USED IN EXPERIMENTS.

Data set	Total Nucleotides	Candidate Pairs	Final Edges
625K	295,492,320	624,603,560	306,775,041
1250K	588,126,956	1,102,288,912	601,086,466
2500K	1,174,628,604	1,942,549,831	1,072,468,284

non-redundant 454 reads covering V3-V5 region. Next, by random sampling we created three subsets of size 650K, 1250K and 2500K sequences, such that a smaller set is completely contained in a larger one. Properties of all three subsets, including the number of candidate pairs generated and the number of edges in the final graph, are summarized in Table I. Note that in our experiments we concentrated on scalability only, since sensitivity of the method has been analyzed by Yang *et al.* [4]. Therefore, in all tests we set F to be *k-mer similarity*, $t = 0.75$, $M = 25$, $k = 15$, $t_{min} = 0.5$, and finally $C_{max} = 10{,}000$. These can be considered default parameters for the 16S rRNA metagenomic analysis.

We executed ELaSTIC on the two-rack IBM Blue Gene/P system. This machine provides 2,048 nodes running at 850 MHz, each with 2 GB of RAM. Although each node has four cores, in our experiments we run only one MPI process per node, owing to the limited memory per node. To build ELaSTIC we used the GCC 4.7 compiler and the IBM BG/P MPI library. For each input data set we ran ELaSTIC on different number of processors recording the time taken by the pair generation stage, and the pair validation stage executed with and without work stealing. The obtained results are summarized in Tables II and III, and Figures 2–4.

We focus first on the candidate pairs generation stage. Table II shows that the sketching-based approach is very fast, and candidate pairs can be identified in just a matter of seconds, even for the largest data set. The time taken to identify candidate pairs is typically less than 10% of the total run time. Here, we should keep in mind that in our tests we executed only one iteration of the sketching procedure, and we employed *k-mer similarity* that can be computed in a linear time. In many production runs we would expect the sketching step to be repeated several times, and a similarity function to use alignment with the dynamic programming in quadratic time. Consequently, the candidate pairs generation would take even smaller fraction of the total run time, which then would be dominated by the validation stage. Figure 2 shows that our approach maintains near linear speedup up to 512 processors. Then the efficiency slightly deteriorates, and is around 60% on 2,048 processors. This performance drop can be attributed to the parallel sorting routine required to distribute the auxiliary list A^i, that is difficult to balance. We are currently investigating how to counter this problem.

To analyze performance of the candidate pairs validation step we measured the total throughput expressed as the number of candidate pairs over the time required to validate them. The obtained throughput was close to 8M pairs per second on 2,048 processors, irrespective of the input data set. To asses the speedup we first extracted a sample of 10K sequences. Then, we analyzed all pairs of sequences using a single processor, and measured the throughput which was 4650 pairs per second. Note that the sequence length distribution was well preserved in the sample, and hence the resulting estimate can be assumed accurate. Figure 3 shows almost perfect scalability of our approach. On 2,048 processors we achieve over 80% efficiency irrespective of the input data set. To demonstrate how important for the scalability is our work stealing procedure, we executed the validation step without the work stealing component. In this case, each processor after finishing tasks in its local queue waited idle for all other processors to complete their work. Figure 4 shows that disabling work stealing tremendously decreases performance. The efficiency does not exceed 40%, and the run time jumps from 240 seconds, for 2500K sequences on 2,048 processors, to 807 seconds – more than a three-fold increase. To conclude, we would like to point out that the scalability of the validation stage is critical as it dominates the total run time of our method. For example, in one of our production runs we analyzed the 2500K set with $t = 0.80$, $M = 25$, $k = 15$, $t_{min} = 0.70$, $C_{max} = 10{,}000$, 9 iterations of the sketching stage, and validation via the global pairwise alignment executed on 1,024 processors of our IBM Blue Gene/P. In this case the entire candidate pairs generation step was completed in 400 seconds while the validation of the resulting 4,157,303,313 candidate edges required 72,425 seconds.

Table II
RUN TIME IN SECONDS OF THE CANDIDATE PAIRS GENERATION STEP FOR THE VARYING NUMBER OF PROCESSORS.

	64	128	256	512	1024	2048
625K	168	87	44	24	14	9
1250K	–	151	76	42	25	16
2500K	–	–	138	76	44	28

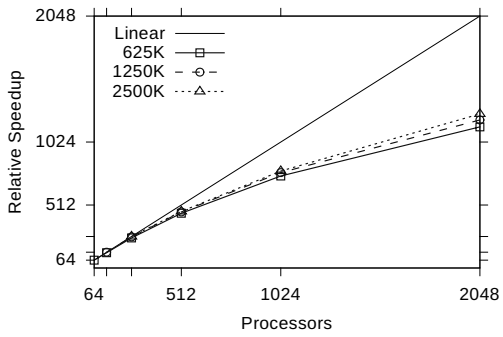

Figure 2. Relative speedup of the candidate pairs generation step.

Table III
RUN TIME IN SECONDS OF THE CANDIDATE PAIRS VALIDATION STEP FOR THE VARYING NUMBER OF PROCESSORS.

	64	128	256	512	1024	2048
625K	2264	1162	596	313	165	79
1250K	–	1958	992	525	285	140
2500K	–	–	1739	908	487	243

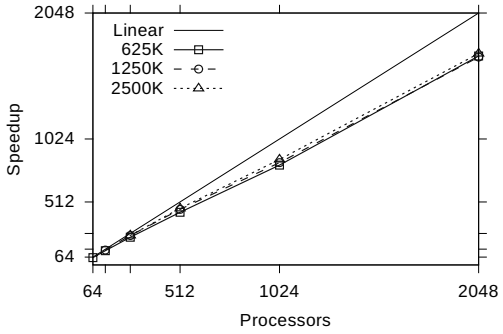

Figure 3. Speedup of the candidate pairs validation step with work stealing.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a distributed memory parallelization of the sketching-based similarity graph generation method. The obtained results demonstrate excellent performance of our load balancing strategies, great scalability of the entire framework, and its ability to handle data sets with millions of sequences. The resulting software provides a production quality solution that supports both DNA and protein sequences, and offers the choice of several similarity functions, including alignment-based and alignment-free functions.

Our current effort is focused on further improving scalability of the candidate pairs generation stage, and reducing memory footprint of the method. Additionally, our approach can be easily extended with a hybrid parallelism, e.g. by including OpenMP in all iterative steps, such as sketches extraction or sequence pairs enumeration, and by accelerating computations of similarity functions. In [6] Yang *et al.* described a MapReduce parallelization of the sketching-based similarity graph generation, and in [18] Wu and Kalyanaraman presented a protein-oriented solution based on suffix trees and the distributed producer/consumer model. A comparison of all three approaches, taking into consideration both scalability and sensitivity criteria, would provide many interesting insights into performance of different parallel programming models as well as similarity detection techniques. We hope to run such a comparison in the near future.

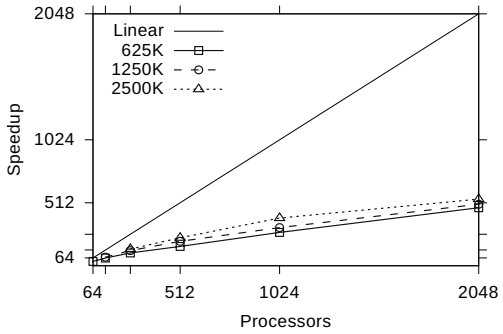

Figure 4. Speedup of the candidate pairs validation step without work stealing.

REFERENCES

[1] G. Parmentier, D. Trystram, and J. Zola, "Large scale multiple sequence alignment with simultaneous phylogeny inference," *Journal of Parallel and Distributed Computing*, vol. 66, no. 12, pp. 1534–1545, 2006.

[2] O. Gascuel, "BIONJ: an improved version of the NJ algorithm based on a simple model of sequence data." *Molecular Biology and Evolution*, vol. 14, no. 7, pp. 685–695, 1997.

[3] L. Holm and C. Sander, "Removing near-neighbour redundancy from large protein sequence collections." *Bioinformatics*, vol. 14, no. 5, pp. 423–429, 1998.

[4] X. Yang, J. Zola, and S. Aluru, "Large-scale metagenomic clustering on map-reduce clusters," *Journal of Bioinformatics and Computational Biology*, vol. 11, no. 1, p. 1340001, 2013.

[5] A. Broder, "On the resemblance and containment of documents," in *Proc. of the Compression and Complexity of Sequences*, 1997.

[6] X. Yang, J. Zola, and S. Aluru, "Parallel metagenomic sequence clustering via sketching and maximal quasi-clique enumeration on map-reduce clouds," in *Proc. IEEE Int. Parallel and Distributed Processing Symposium*, 2011, pp. 1223–1233.

[7] R. Korf, "Multi-way number partitioning." in *Proc. of the International Joint Conference on Artificial Intelligence*, 2009, pp. 538–543.

[8] J. Dinan, D. Larkins, P. Sadayappan *et al.*, "Scalable work stealing," in *Proc. of Supercomputing*, 2009, pp. 1–11.

[9] S. Vinga and J. Almeida, "Alignment-free sequence comparison – a review," *Bioinformatics*, vol. 19, no. 4, pp. 513–523, 2003.

[10] A. Sarje, J. Zola, and S. Aluru, *Scientific Computing with Multicore and Accelerators*. Chapman and Hall/CRC, 2010, ch. Pairwise Computations on the Cell Processor with Applications in Computational Biology.

[11] S. Sun, J. Chen, W. Li *et al.*, "Community cyberinfrastructure for advanced microbial ecology research and analysis: the CAMERA resource," *Nucleic Acids Research*, vol. 39, no. suppl 1, pp. D546–D551, 2011.

[12] A. Kalyanaraman, S. Aluru, V. Brendel, and S. Kothari, "Space and time efficient parallel algorithms and software for EST clustering," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 12, pp. 1209–1221, 2003.

[13] A. Broder, M. Charikar, A. Frieze *et al.*, "Min-wise independent permutations," *Journal of Computer and System Sciences*, vol. 60, no. 3, pp. 630–659, 2000.

[14] R. Edgar, "Local homology recognition and distance measures in linear time using compressed amino acid alphabets," *Nucleic Acids Research*, vol. 32, no. 1, pp. 380–385, 2004.

[15] "MurmurHash," http://sites.google.com/site/murmurhash/.

[16] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[17] "Human microbiome project data set documentation," http://www.hmpdacc.org/resources/dataset_documentation.php.

[18] C. Wu, A. Kalyanaraman, and W. Cannon, "pGraph: Efficient parallel construction of large-scale protein sequence homology graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 10, pp. 1923–1933, 2012.