pGraph: Constructing Large-Scale Protein Sequence Homology Graphs Efficiently on Distributed Memory Machines

Changjun Wu, Ananth Kalyanaraman, and William R. Cannon

Abstract—Protein sequence homology detection is a fundamental problem in computational molecular biology, with a pervasive application in nearly all analyses that aim to structurally and functionally characterize protein molecules. While detecting homology between two protein sequences is computationally inexpensive, detecting pairwise homology at a large-scale becomes prohibitive, requiring millions of CPU hours. Yet, there is currently no efficient method available to parallelize this kernel. In this paper, we present the key characteristics that make this problem particularly hard to parallelize, and then propose a new parallel algorithm that is suited for large-scale protein sequence data. Our method, called *pGraph*, is designed using a hierarchical multiple-master multiple-worker model, where the processor space is partitioned into subgroups and the hierarchy helps in ensuring the workload is load balanced fashion despite the inherent irregularity that may originate in the input. Experimental evaluation demonstrates that our method scales linearly on all input sizes tested (up to 640K sequences) on a 1,024 node supercomputer. In addition to demonstrating strong scaling, we present an extensive study of the various components of the system and related parametric studies.

Index Terms—Parallel protein sequence homology detection; parallel sequence graph construction; hierarchical master-worker paradigm.

1 Introduction

Protein sequence homology detection is a fundamental problem in computational molecular biology, where given a set of protein sequences, the goal is to identify *highly similar pairs of sequences*. In graph-theoretic terms, if we were to represent the input protein sequences as vertices and pairwise sequence similarity as edges, then the problem of pairwise homology detection is equivalent of constructing the graph.

Homology detection is widely used in nearly all analyses targeted at functional and structural characterization of protein molecules [8]. Most notably, the operation is heavily used in clustering applications, where the problem is to partition the input sequences such that all proteins that are "related" to one another by a pre-defined degree of sequence homology are grouped together. Clustering has become highly significant of late because of its potential to uncover thousands of previously unknown proteins from metagenomic data sets. Metagenomics [5], which is a rapidly emerging sub-field, involves the study of environmental microbial communities using novel genomic tools. In 2007, a single study that surveyed an ocean microbiota [16] resulted in the discovery of nearly 4×10^3 previously unknown protein families, significantly expanding the protein universe as we know it. As protein families are defined as groups of functionally related proteins, homology detection and clustering play a central role during family identification.

While there are numerous software options available for protein sequence clustering (e.g., [2], [3], [4], [8], [11]), all of them assume that the graph is already constructed and available as input. However, modern-day use-cases such as the ocean metagenomic sequence clustering suggest that this is not the case. This is because these large-scale projects generate sequence information of their own and hence will have to contend with detecting the sequence homology among all new sequences and against sequence information generated from previous projects. For example, the ocean metagenomic project alone generated more than 17 million new protein (ORF) sequences and this set was analyzed alongside over 11 million sequences downloaded from public protein sequence databanks (for a total of 28.6 million sequences). Consequently, the most dominant phase of computation in the entire analysis was the detection of pairwise sequence homology, which alone took 10^6 CPU hours, even after using heuristic approaches to compute homology [16]. Our own experience with the homology detection phase [15] is further confirmation for the challenges that confront this problem.

In this paper, we propose a new parallel algorithm for carrying out sequence homology detection of large-scale protein sequence data. Through detection, the resulting output is the sequence graph which can be directly used as input for the subsequent clustering step.

- *C. Wu and A. Kalyanaraman are with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, 99164.*
  E-mail: {cwu2, ananth}@eecs.wsu.edu
- *W.R Cannon is with Pacific Northwest National Laboratory, Richland, WA. 99352.*
  E-mail: william.cannon@pnl.gov

Developing a scalable solution for this problem using parallel processing is essential for this emerging application domain as it can help reduce the time to solution and enable larger data sets to be analyzed. The problem also has several attributes that make it interesting from the standpoint of parallel algorithm development. (i) **Input size:** Firstly, the problem is data-intensive. Tens of millions of protein sequences are already available from public repositories (e.g., CAMERA http://camera.calit2.net/). (ii) **Work processing rate:** The rate at which work is processed could be highly irregular. Detecting pairwise sequence homology is equivalent to the problem of finding optimal alignment [13], the time for which is proportional to the product of lengths of the strings being aligned. However, due to the large variance in the input protein sequence length, the time to process each unit of work could also vary significantly, as will be shown in Section 4. (iii) **Work generation rate:** To avoid a brute-force all-against-all sequence comparison, a string index such as suffix tree [9] or look-up table [1] is used so that only pairs satisfying an exact matching criterion need to be further evaluated [15], [16]. However, the rate at which the pairs are generated could be irregular. For instance, same sized portions of the suffix tree index could result in the generation of drastically different number of sequence pairs for alignment, as will be shown in Section 4. *A priori* stocking of pairs that require alignment is also not an option because of a worst-case quadratic explosion of work. (iv) **Local availability of data:** Finally, the ready in-memory availability of sequence data during alignment processing cannot be guaranteed under distributed memory machine setting because of large input sizes. Alternatively, moving computation to data is also virtually impossible because a pair listed for alignment work could involve any two input sequence. The algorithm proposed in this paper addresses all these challenges.

1.1 Contributions

In this paper, we present a novel algorithm for carrying out large-scale protein sequence homology detection. Our algorithm, called *pGraph*[1], is designed to take advantage of the large-scale memory and compute power available from distributed memory parallel machines. The method uses a hierarchical multiple-master multiple-worker model to dynamically distribute tasks corresponding to both work generation and work processing in a load balanced fashion. The processor space is organized into subgroups with each subgroup consisting of a producer (for work generation), a master (for work distribution) and a fixed number of workers acting as consumers of work. This producer-consumer model of organizing a subgroup helps decouple work generation from work processing. In addition, a dedicated supermaster is tasked with the responsibility of ensuring that the tasks are evenly shared among subgroups. The multiple-master model also helps avoid single point bottlenecks.

Experimental results show that this new approach achieves linear scaling on 1,024 nodes for the range of input tested (up to 640,000 sequences). More notably, the method was able to maintain parallel efficiency at more than 90% over all processor size tested. In addition to scalability results, we also present a thorough report on the system behavior by components. Though presented in the context of protein sequence graph construction, our method could be extended to other data-intensive applications where there is irregularity in work generation and/or in work processing.

The paper is organized as follows. Section 2 presents the current state of art for parallel sequence homology detection. Section 3 presents our proposed method and implementation details. Experimental results are presented and discussed in Section 4, and Section 5 concludes the paper.

2 Related Work

Sequence homology between two protein sequences can be evaluated using rigorous optimal alignment algorithms [10], [13] or heuristic alignment methods such as BLAST [1]. Protein sequence clustering is a well researched topic with numerous algorithms and software tools (e.g., [2], [3], [4], [8], [11]). While sequence homology is a fundamental computational kernel within clustering applications, there are currently no efficient parallel algorithms. In order to accommodate for the 10^6 CPU hours, the ocean metagenomics project [16] used an *ad hoc* parallel strategy, where an all-against-all sequence comparison using BLAST was manually partitioned across 125 dual processors systems and 128 16-processor nodes each containing between 16GB-64GB of RAM.

Recently, we proposed a parallel method [15] for the problem of protein clustering. The main contribution of this method was that it showed how to break a single large graph problem into multiple disjoint subproblems. This was achieved by first enumerating all connected components so that the individual components can be post-processed independently for dense subgraph detection. However, detecting connected components also involves enumerating all the edges of the graph through sequence homology detection. While performance tests demonstrated linear scaling up to 128 processors for 160,000 sequences, the phase for pairwise sequence homology detection failed to scale linearly for larger number of processors [15]. The cause for the slowdown was primarily the irregularity that was observed between pair generation and alignment computation. Interestingly, the same scheme had demonstrated linear scaling on DNA sequence clustering problems earlier [7]. This is the motivation behind our newly proposed design which decouples the work generation (producer) from work processing (consumer).

1. stands for "protein sequence homology Graph construction"

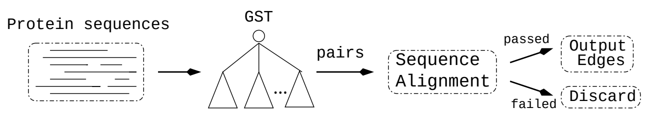


Fig. 1. Tree-based filtering scheme used by our approach for protein sequence homology detection. GST stands for Generalized Suffix Tree.

Our observations of a higher complexity for metagenomic protein data when compared to DNA data are consistent with other previous studies. For example, in the human genome assembly project [14], the all-against-all sequence homology detection of roughly 28 million DNA sequences consumed only 10^4 CPU hours. Contrast this with the 10^6 CPU hours observed for analyzing roughly the same number of protein sequences in the ocean metagenomic project [16].

3 Methods

Notation: Let $S = \{s_1, s_2, \ldots s_n\}$ denote the set of n input protein sequences, and Let p denote the set of processors. Let $G = (V, E)$ denote a graph defined as $V = S$ and $E = \{(s_i, s_j) \mid s_i$ and s_j have a significant sequence similarity$\}$.

Problem statement: Given a set S of n protein sequences and p processors, the protein sequence graph construction problem is to construct G in parallel.

Given S, the primary question is to detect if there exists an edge between any two vertices. While it is computationally inexpensive to determine an optimal alignment between two average-length protein sequences, performing hundreds of millions to billions of alignment computations could be highly prohibitive (e.g., [15], [16]). There are two independent ways to reduce the computational burden — one is to use algorithmic heuristics (see Section 3.0.1) and another is to use high performance computing (see Section 3.1).

3.0.1 Generating pairs

A brute-force approach to detect the presence of an edge is to enumerate all possible pairs of sequences and retain only those as edges which pass the alignment test. Such an approach would evaluate $\binom{n}{2}$ pairs for alignment, and hence is not a scalable solution. Alternatively, since alignments represent approximate matching, the presence of long exact matches can be used as a necessary but not sufficient condition [7]. This approach can filter out a significant fraction of poor quality pairs and thereby reduce the number of pairs aligned significantly (e.g., by $> 70\%$ [15]). Figure 1 illustrates the tree based filtering scheme.

To implement exact matching, we use the maximal match detection algorithm described in [7]. This method generates only those pairs that show high promise for passing the alignment test. It first builds a Generalized Suffix Tree (GST) data structure [9] as a string index for the strings in S. The tree index is generated as a forest of subtrees and then the individual subtrees are traversed to generate pairs. The pair generation process may exhibit nonuniformity in the sense that subtrees with the similar size could produce drastically different number of pairs and/or at different rates (as shown in Section 4). This is because the composition of a subtree is purely data-dependent and if a section of subtree receives a highly repetitive fraction of the input sequences then it is bound to generate a disproportionately large number of pairs. The tree construction code outputs the GST as a forest of subtrees on to the file system, with a small fixed number of sub-trees in each file.

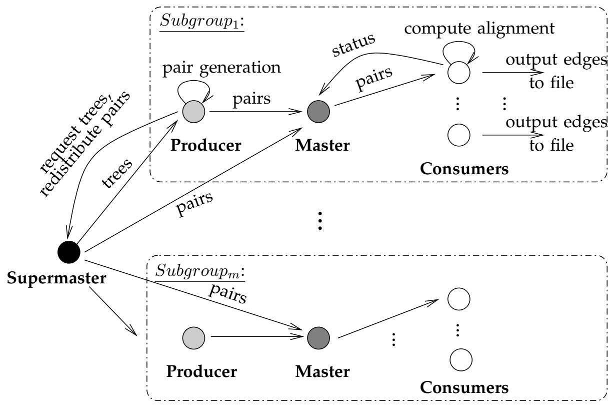


Fig. 2. The hierarchical multiple-master multiple-worker design of the *pGraph* approach showing the interaction of the individual components within and outside the subgroups.

3.1 *pGraph*: Parallel graph construction

In this section, we present a novel and efficient parallel algorithm to compute sequence alignments on the pairs generated from the tree index and output edges of the graph G. Our method uses a hierarchical multiple-master multiple-worker model to counter the challenges posed by inherent irregularities of pair generation and alignment rates.

The system architecture is illustrated in Figure 2. The inputs include the sequence set S and the tree index T. The tree index is available as a forest of k subtrees, which we denote as $T = \{t_1, t_2, \ldots t_k\}$. In both theory and practice, the value of k tends to be of the order of n, which is good for parallel distribution for large values of n. The output of *pGraph* is the set of all edges of the form (s_i, s_j) s.t., the sequences s_i and s_j pass the alignment test based on user-defined cutoffs. Given p processors and a small number $q \geq 3$, the parallel system is partitioned as follows: i) one processor is designated to act as the *supermaster* for the entire system; and ii) the remaining $p - 1$ processors are partitioned into m subgroups such that each subgroup has exactly q processors[2]. Furthermore, a subgroup is internally organized with one processor designated to the role of a *producer*, another to the role of a *master*, and the remaining $q-2$ processors to the role of a *consumer*.

At a high level, the producers are responsible for pair generation, the masters for distributing the alignment workload within their respective subgroups, and the consumers for computing alignments. The supermaster plays a supervisory role to ensure load is distributed evenly among subgroups. Nevertheless, there are several design considerations that need to be taken into account. In what follows, we explain these factors and present algorithms and protocols for each component in the system.

Notation: Let:
$P_{buf} \leftarrow$ a fixed sized pair buffer at the producer;
$M_{buf} \leftarrow$ a fixed sized pair buffer at the master;
$C_{buf} \leftarrow$ a fixed sized pair buffer at the consumer;
$S_{buf} \leftarrow$ a fixed sized pair buffer at the supermaster;
$b_1 \leftarrow$ batch size (for pairs) from producer or supermaster to master;
$b_2 \leftarrow$ batch size (for pairs) from master to consumer;

Producer: The primary responsibility of a producer is to load a subset of subtrees in T and generate pairs using the maximal matching algorithm in [7]. Pairs could be allocated for alignment computation by communicating them to the local master in the subgroup and have the master assign pairs to its consumers. However, such an approach runs the risk of a potential bottleneck situation where a producer receives a subtree that generates a significantly large volume of pairs and/or generate pairs that take significantly long alignment times. Another issue is the timing of communicating the pairs for alignment allocation. The memory limitation at the producer limits the size of P_{buf} used for temporary pair storage. On the other hand, immediately dispatching the pairs as they are generated may increase communication overhead or may overrun M_{buf}. Assigning subtrees to producers will also have to be done dynamically at a fine granular level as otherwise it may result in nonuniform distribution of pairs across subgroups.

To overcome the above challenges, the algorithm shown in Algorithm 1 is followed. Initially, a producer fetches a batch of subtrees (available as a single file) from the supermaster. The producer then starts to generate and enqueue pairs into P_{buf}. Subsequently, the producer dequeues and sends b_1 pairs to the master. This is implemented using a nonblocking send so that when the master is not yet ready to accept pairs, the producer can continue to generate pairs, thereby allowing masking of communication. After processing the current batch of subtrees, the producer requests another batch from the supermaster. Once there are no more subtrees available, the producers dispatch pairs to both master *and* supermaster, depending on whoever is responsive to their nonblocking sends. This strategy gives the producer an option of redistributing its pairs to other subgroups (via supermaster) if the local master is busy. In fact, we show in the experimental section that the strategy of using the supermaster route pays off significantly and ensures the system is load balanced.

2. With the possible exception of one subgroup which may obtain less than q processors if $(p-1)\%q \neq 0$.

Algorithm 1 Producer

```
1.  Request a batch of subtrees from supermaster
2.  while true do
3.    $T_i \leftarrow$ received subtrees from supermaster
4.    if $T_i = \emptyset$ then
5.      break while loop
6.    end if
7.    repeat
8.      if $P_{buf}$ is not FULL then
9.        Generate at most $b_1$ pairs from $T_i$
10.       Insert new pairs into $P_{buf}$
11.     end if
12.     if $send_{P\rightarrow M}$ completed then
13.       Extract at most $b_1$ pairs from $P_{buf}$
14.       $send_{P\rightarrow M} \leftarrow$ Isend extracted pairs to master
15.     end if
16.   until $T_i = \emptyset$
17.   Request a batch of subtrees from supermaster
18. end while
19. /* Flush remaining pairs */
20. while $P_{buf} \neq \emptyset$ do
21.   Extract at most $b_1$ pairs from $P_{buf}$
22.   if $send_{P\rightarrow M}$ completed then
23.     $send_{P\rightarrow M} \leftarrow$ Isend extracted pairs to master
24.   end if
25.   if $send_{P\rightarrow S}$ completed then
26.     $send_{P\rightarrow S} \leftarrow$ Isend extracted pairs to supermaster
27.   end if
28. end while
29. Send END signal to supermaster
```

Master: The primary responsibility of a master is to ensure all consumers in its subgroup are always busy with alignment computation. The main challenge in this setup is to ensure that a master's local buffer for storing pairs (M_{buf}) is not overrun by an overactive producer or is starved due to a slow producer. Either of these could happen because the pair generation rate is data-dependent. The above challenge is overcome as follows (see Algorithm 2).

Initially, to ensure that there is a steady supply and dispatch of pairs, the master listens for messages from both its producer and consumers. However, once $|M_{buf}|$ reaches a preset limit called τ, the master realizes that its producer has been more active than the rate at which pairs are processed at its consumers, and therefore shuts off listening to its producer, while only dispatching pairs to its consumers until $|M_{buf}| \leq \tau$. The rationale for this strategy is the practical expectation that pair generation tends to happen much faster than pair alignment. More importantly, this strategy helps to keep the consumers always busy with alignment computation. Since consumers are the majority in the system, this has a direct scalability implication. When the producer has exhausted sending all its pairs, the master can fallback on the supermaster to provide pairs.

As for serving consumers, the master maintains a priority queue, which keeps track of each of its consumers based on the latter's most recent status report to the master. Priority is defined based on the number of pairs left to be processed at the consumer's C_{buf}. Priority is implemented in the master as follows: at any given iteration, pairs are allocated in batches of size b_2 and send to consumers in the decreasing (or, nonincreasing) order of priority. While frequent updates from consumers could help the master to better assess the situation on each consumer, such a scheme will also increase communication overhead. As a tradeoff, we implement a priority queue by maintaining only three levels of priority depending on the condition of a consumer's C_{buf}: $\frac{1}{2}$-empty, $\frac{3}{4}$-empty, and completely empty. This also implies that the master, instead of pushing pairs on to consumers, waits for consumers to take the initiative in requesting pairs, while reacting in the order of their current workload status.

Consumer: The primary responsibility of the consumer is to compute optimal alignments using the Smith-Waterman algorithm [13] for the pairs allocated to it by its master and output results. The main challenge is to ensure that a consumer does not starve for work. The consumer follows Algorithm 3. The consumer maintains a fixed size pair buffer C_{buf}. When the master sends a new batch of b_2 pairs, it starts processing them one at a time. When C_{buf} reaches half size, the consumer sends out a message to the master updating its new buffer status, and continues processing of the remaining pairs in C_{buf}. At this stage, it also posts a nonblocking receive to accept new pairs from master while it is computing alignments. The send is also implemented as nonblocking to allow for further communication masking. Another message is sent out at the $\frac{1}{4}$ stage, but only after checking the status of the previous receive. If the master had sent pairs in the meantime, then the pairs are inserted into C_{buf} and the processing continues. Alternatively, if there were no messages from the master and C_{buf} becomes empty, the consumer sends another message to inform the master that it is starving and waits for the master to reply.

Before aligning a batch of pairs, the consumer has to ensure that the sequences needed are available in the local memory. While the local memory on a consumer may not be always sufficient to store the entire set of input sequences (S), it could be used to cache many strings. We use a parameter $\psi \leq n$ for this purpose. At initialization, all consumers load an arbitrary collection ψ sequences from I/O. This statically allocated buffer is then used as a string cache during alignment computation. Only strings which are not in the local cache are fetched from I/O.

Algorithm 2 Master
```
1.  $\tau$: predetermined cutoff for the size of $M_{buf}$
2.  $Q$: priority queue for consumers
3.  while true do
4.    /* Recv messages */
5.    if $|M_{buf}| > \tau$ then
6.      $msg \leftarrow$ post Recv for consumers
7.    else
8.      $msg \leftarrow$ post open Recv
9.      if $msg \equiv$ pairs then
10.       Insert pairs into $M_{buf}$
11.       if $msg \equiv$ END signal from supermaster then
12.         break while loop
13.       end if
14.     else if $msg \equiv$ request from consumer then
15.       Place consumer in the appropriate priority
          queue
16.     end if
17.   end if
18.   /* Process consumer requests */
19.   while $|M_{buf}| > 0$ and $|Q| > 0$ do
20.     Extract a highest priority consumer, and send
        appropriate amount of pairs
21.   end while
22. end while
23. /* Flush remaining pairs to consumers */
24. while $|M_{buf}| > 0$ do
25.   if $|Q| > 0$ then
26.     Extract a highest priority consumer, and send
        appropriate amount of pairs
27.   else
28.     Waiting consumer requests
29.   end if
30. end while
31. Send END signal to consumers
```

Supermaster: The primary responsibility of the supermaster is to ensure both the pair generation workload and pair alignment workload are balanced across subgroups. To achieve this, the supermaster follows Algorithm 4. At any given iteration, the supermaster is either serving a producer or a master. For managing the pair generation workload, the supermaster assumes the responsibility of distributing subtrees (in batches) to individual producers. The supermaster, instead of pushing subtree batches to producers, waits for producers to request for the next batch. This approach guarantees that the run-time among producers, and not the number of subtrees processed, is balanced at program completion.

The second task of the supermaster is to serve as a conduit for pairs to be redistributed across subgroup

Algorithm 3 Consumer

```
1. $g$: number of consumers in the same subgroup
2. $\psi = \frac{n}{g}$: number of sequences to be cached statically
3. EMPTY, HALF, QUARTER: 0, $\frac{b_2}{2}$ and $\frac{b_2}{4}$ buffer status
4. $S_{cache} \leftarrow$ load $\psi$ sequences from I/O
5. $Recv \leftarrow$post nonblocking receive
6. while true do
7.   if $Recv$ completed then
8.     if Sequence request from consumer $c_k$ then
9.       Pack sequences and send them out to $c_k$
10.      $Recv \leftarrow$post nonblocking receive
11.    else if Sequences from other consumer then
12.      Unpack sequences to dynamically cache
13.      $Recv \leftarrow$post nonblocking receive
14.    else if Pairs from master then
15.      Insert pairs into $C_{buf}$
16.      Prepare sequence requests for each consumer
17.      Send sequence requests to other consumers
18.      $Recv \leftarrow$post nonblocking receive
19.    end if
20.  else
21.    if $C_{buf} > 0$ then
22.      /* Align a pair if their seqs. are ready */
23.      extract next pair $(i, j)$ from $C_{buf}$
24.      if both $s_i$ and $s_j$ are ready then
25.        Align sequences $s_i$ and $s_j$
26.      else
27.        Append pair $(i, j)$ at the end of the $C_{buf}$
28.      end if
29.      /* Report $C_{buf}$ status back to master */
30.      if $C_{buf}$=$\frac{b_2}{2}$ then
31.        Send HALF status to master
32.      else if $C_{buf}$=$\frac{b_2}{4}$ then
33.        Send QUARTER status to master
34.      else if $C_{buf} = 0$ then
35.        Send EMPTY status to master
36.      end if
37.    end if
38.  end if
39. end while
```

boundaries. To achieve this, the supermaster maintains a local buffer, S_{buf}. Producers can choose to send pairs to supermaster if their respective subgroups are saturated with alignment work. The supermaster then decides to push the pairs (in batches of size b_1) to masters of other subgroups, depending on their respective response rate (dictated by their current workload). This functionality is expected to be brought into effect at the ending stages of producers' pair generation, when there could be a few producers that are still churning out pairs in numbers while other producers have completed generating pairs. As a further step toward ensuring load balanced distribution at the producers' ending stages, the supermaster sends out batches of a reduced size, $\frac{b_1}{2}$, in order to compensate for the deficiency in pair supply. Correspondingly, the masters also reduce their batchsizes proportionately at this stage. As will shown in our experimental section, the supermaster plays a key role in load balancing of the entire system.

Algorithm 4 Supermaster

```
1. Let $P = \{p_1, p_2, ...\}$ be the set of active producers
2. $Recv_{S \leftarrow P} \leftarrow$ Post a nonblocking receive for producers
3. while $|P| \neq 0$ do
4.   /* Serve the masters*/
5.   if $|S_{buf}| > 0$ then
6.     $m_i \leftarrow$ Select master for pairs allocation
7.     Extract and Isend $b_1$ pairs to $m_i$
8.   end if
9.   /* Serve the producers*/
10.  if $Recv_{S \leftarrow P}$ completed then
11.    if $msg \equiv$ subtree request then
12.      Send a batch of subtrees ($T_i$) to corresponding producer
13.    else if $msg \equiv$ pairs then
14.      Insert pairs in $S_{buf}$
15.    end if
16.    $Recv_{S \leftarrow P} \leftarrow$ Post a nonblocking receive for producers
17.  end if
18. end while
19. Distribute remaining pairs to all masters in a round-robin way
20. Send END signal to all masters
```

3.2 Implementation

The *pGraph* code was implemented in C/MPI. All parameters described in the algorithm section were set to values based on preliminary empirical tests. The default settings are as follows: b_1 =30,000; b_2 =2,000; $|P_{buf}| = 5 \times 10^7$; $|M_{buf}| = 6 \times 10^4$; $|C_{buf}| = 6 \times 10^3$; $|S_{buf}| = 4 \times 10^6$.

4 Experimental Results & Discussion

4.1 Experimental setup

Input data: The *pGraph* implementations were tested using an arbitrary collection of 2.56×10^6 (n) amino acid sequences representing an ocean metagenomic data set available at the CAMERA metagenomics data archive []. The sum of the length of the sequences (m) in this set is 390,345,218, and the mean$\pm\sigma$ is 152.48 ± 167.25; the smallest sequence has 1 residues and longest 32,794 residues. Smaller size subsets containing 20K, 40K, 80K, . . ., 1.28×10^6 were derived and used for scalability tests.
Experimental platform: All tests were performed on the *Chinook* supercomputer at the EMSL facility in Pacific Northwest National Laboratory. This is a 160 TF supercomputer running Red Hat Linux and consists of

2,310 HP DL185 nodes with dual socket, 64-bit, Quad-core AMD 2.2 GHz Opteron processors with an upper limit of 4 GB RAM per core. The network interconnect is Infiniband. A global 297 TB distributed LUSTRE file system is available to all nodes.

pGraph-specific settings: Even though the Chinook platform supports access to 4 GB RAM per core, in all our runs, we assumed a memory upper bound of $O(\frac{m}{p_c})$ per MPI process, where p_c is the number of consumers in a subgroup. This was done to emulate a more realistic use-case on any distributed memory machine. At the start of execution, all consumers in a subgroup load the input sequences in a distributed even fashion such that each consumer receives a unique $O(\frac{m}{p_c})$ fraction of the input. The locally available set of sequences is referred to as the "static cache". Any additional sequence that is temporarily fetched into local memory during alignments is treated as part of a constant space "dynamic cache" buffer.

In order to generate the suffix tree index required for all input sets, a suffix tree construction code from one of our earlier developments [7] was used. The tree index statistics on the different input sets are shown in Table **??**. Because the construction was quick, trees were generated just using a single CPU. Note that there are parallel implementations [7], [**?**] already available that can be used for larger inputs. For all our runs, we assume that the tree index is already built using any method of choice and stored in the disk.

For all the performance results presented in Sections 4.2 and 4.3, we set the subgroup size to 16 and the number of producers per subgroup to 2 (to approximate a producer:consumer ratio of 1:8 within each subgroup). The effect of changing these parameters are later studied in Section 4.4.

4.2 Comparative evaluation: $pGraph_{I/O}$ vs. $pGraph_{nb}$

At first, we compare the two versions of our software, $pGraph_{I/O}$ and $pGraph_{nb}$, which use I/O and non-blocking communication, respectively, for fetching sequences not in the local string cache during alignment at consumers. Figure 3 shows the runtime breakdown of an average consumer under each implementation, on varying number of processors for the 640K input. Both implementations scale linearly with increasing processor size. However, in $pGraph_{I/O}$, alignment time accounted only for $\sim 80\%$ of the total run-time, and the remaining 20% of the time is dominated primarily by I/O, for all processor sizes. In contrast, for $pGraph_{nb}$ nearly all of the run-time was spent performing alignments leaving the overhead associated with non-blocking communication negligible. Consequently, the non-blocking version is 20% faster than the I/O version. The trends observed hold for other data sets tested as well (data not shown). The results show the effectiveness of the masking strategies used in the non-blocking implementation and more importantly, its ability to effectively eliminate overheads associated with dynamic sequence fetches through the network. This coupled with the linear scaling behavior observed for $pGraph_{nb}$ makes it the implementation of choice. Note that the linear scaling behavior of $pGraph_{I/O}$ can primarily be attributed to the availability of a fast, parallel I/O system such as Lustre. For systems which do not have such a sophisticated I/O system in place, the I/O overheads are expected to become even more pronounced and could negatively impact speedup.

In what follows, we present all of our performance evaluation using only $pGraph_{nb}$ as our default implementation.

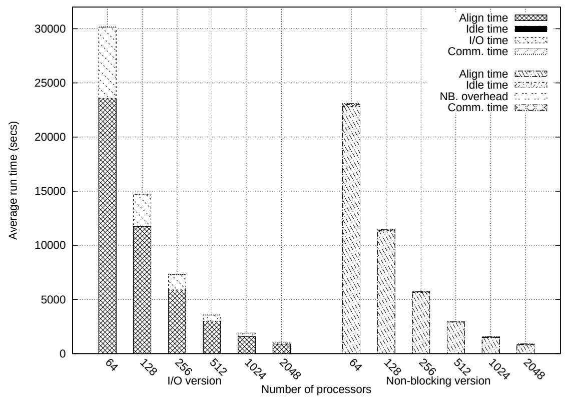


Fig. 3. Comparison of the I/O and non-blocking communication versions of *pGraph*. Shown are the runtime breakdown for an average consumer between the two versions. All runs were performed on the $640K$ input sequence set. The results show the effectiveness of the non-blocking communication version in eliminating sequence fetch overhead.

4.3 Performance evaluation for $pGraph_{nb}$

Table 2 shows the total parallel runtime for a range of input sizes (20K ... 2,560K) and processor sizes (16 ... 2,048). The large input sizes scale linearly up to 2,048 processors and more notably, inputs even as small as 20K scale linearly up to 512 processors. The speedup chart is shown in Figure 4a. All speedups are calculated relative to the least processor size run corresponding to each input. The smallest run had 16 processors because it is the subgroup size. The highest speedup $(2,004\times)$ was achieved for the 2,560K data on 2,048 processors. Figure 4b shows the parallel efficiency of the system. As shown, the system is able to maintain an efficiency above 90% for most inputs. Also note that for several inputs, parallel efficiency slightly *increases* with processor size for smaller number of processors (e.g., 80K on p : $32 \rightarrow 64$). This superlinear behavior can be attributed to the minor increase in the number of consumers (relative to the whole system size) — i.e., owing to the way in which the processor space is partitioned, the number of consumers more than doubles when the whole system size is doubled (e.g., when p increases from 16 to 32, the number of consumers increases from 12 to 25). And this

No. input sequences	Total sequence length	No. subtrees in the forest	No. tree nodes	Construction time (in secs; single CPU)
20K	3,852,622	133,639	5,721,111	3
40K	8,251,063	149,501	12,318,567	6
80K	20,600,384	158,207	30,952,989	26
160K	43,480,130	159,596	66,272,332	56
320K	86,281,743	159,991	128,766,176	108
640K	160,393,750	160,016	237,865,379	205
1,280K	222,785,671	160,000	306,132,294	300
2,560K	392,905,218	160,000	533,746,500	520

TABLE 1
Sequence and suffix tree index statistics for different input sets.

Input number	Number of processors (p)								Number of pairs
of sequences(n)	16	32	64	128	256	512	1,024	2048	(in millions)
20K	398	192	94	49	26	14	9	-	6.5
40K	1,217	583	286	143	73	37	20	-	16.9
80K	19,421	9,260	4,481	2,243	1,146	616	373	-	48.5
160K	-	-	7,666	3,837	1,978	1,011	574	356	125.6
320K	-	-	16,283	8,056	4,061	2,082	1,060	623	365.7
640K	-	-	23,102	11,481	5,739	2,942	1,561	893	590.1
1,280K	-	-	-	32,113	16,042	8,014	4,031	2,066	2,410.4
2,560K	-	-	-	124,884	62,222	31,103	15,639	7,975	5,258.3

TABLE 2
The run-time (in seconds) for $pGraph_{nb}$ on various input and processor sizes. An entry '-' means that the corresponding run was not performed. The last column shows the number of pairs aligned (in millions) for each input as a measure of work.

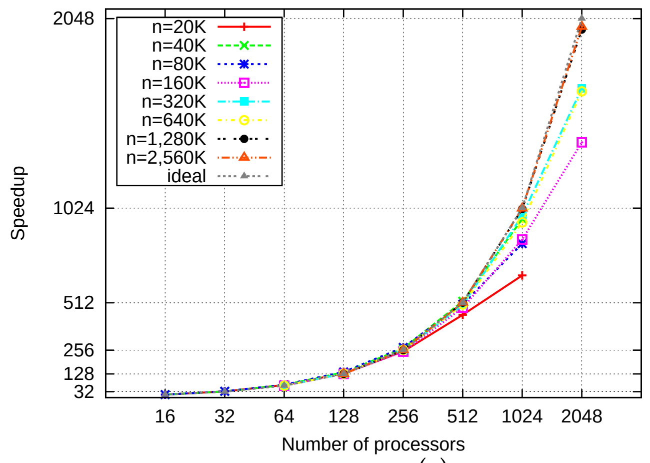
increased availability contributes more significantly for smaller system sizes — e.g., when p increases from 16 to 32, the one extra consumer adds 4% more consumer power to the system. This effect however diminishes for larger system sizes.

Table 2 also shows run-time increase as a function of input number of sequences. Although this function cannot be analytically determined because of its input-dependency, the number of alignments needed to be performed can serve as a good indicator. However, Table 2 shows that in some cases the run-time increase is not necessarily proportional to the number of pairs aligned — e.g., note that a $3\times$ increase in alignment load results in as much as a $16\times$ increase in run-time, when n increases from 40K to 80K. Upon further investigation, we found the cause to be the difference in the sequence lengths between both these data sets — both mean and standard deviation of the sequence lengths increased from 205±118 for the 40K input to 256±273 for the 80K input, thereby implying an increased cost for computing an average unit of alignment.

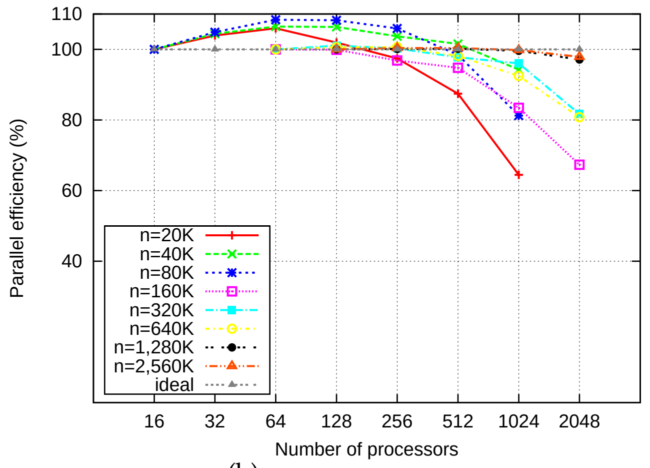
To better understand the overall system's linear scaling behavior and identify potential improvements, we conducted a thorough system-wide component-by-component study using $n = 640K$ as a case study.

Consumer behavior: At any given point of time, a consumer in $pGraph_{nb}$ is in one of the following states: i) *(align)* compute sequence alignment; or ii) *(comm)* communicate to fetch sequences or serve other consumers, or send pair request to master; or iii) *(idle)* wait for master to allocate pairs. As shown in Figure 3, an average consumer in $pGraph_{nb}$ spends well over 98% of the total time computing alignments. This desired behavior can be attributed to the combined effectiveness of our masking strategies, communication protocols and the local sequence buffer management strategy. The fact that the idle time is negligible demonstrates the merits of sending timely requests to the master depending on the state of the local pair buffer. Given that the sequence request patterns are completely random and sequence fetches are done asynchronously between consumers, the fact that the contribution from such communication is negligible indicates the effectiveness our strategy to overlap communication with alignment work. Keeping a small subgroup size (16 in our experiments) is also a notable contributor to the reason why the overhead due to sequence fetches, both at the requester and sender, is negligible. For larger subgroup sizes, this asynchronous wait times can increase.

The local sequence management strategy also plays an important role. Note that each consumer only stores $O(\frac{m}{p_c})$ characters of the input in the static cache. Figure 5 shows the statistics relating to sequence fetches carried out at every step as the algorithm proceeds at a con-



(a)



(b)

Fig. 4. (a) Speedup and (b) Parallel efficiency of *pGraph*. The speedup and efficiency computed are relative, and because the code was not run on smaller processor sizes for larger inputs, the reference speedups at the beginning processor size were assumed at linear rate — e.g., a relative speedup of 64 was assumed for 160K on 64 processors. This assumption is valid because it is consistent with the linear speedup trends observed at that processor size for smaller inputs.

sumer. As the top chart shows, the probability of finding a sequence in the local static cache is generally low, thereby implying that most of the sequences required for alignment computation need to be fetched over network. While the middle chart confirms this high need for communication, it can be noted that the peaks and valleys in this chart do not necessarily correspond to that of the top chart. This is because of the temporary availability of sequences in the fixed size dynamic sequence buffer (bottom chart), which serves to reduce the overall number of sequences fetched from other consumers.

Master behavior: The master within a subgroup is in one of the following states at any given point of execution: i) *(idle)* waiting for consumer requests or new pairs from the local producer(s); or ii) *(comm)* sending pairs to a consumer; or iii) *(comp)* performing local operations to manage subgroup. Figure 6 shows that the master is available (i.e., idle) to serve its local subgroup nearly all of its time. This shows the merit of maintaining manageably small subgroups in our design. The effectiveness of the master to provide pairs in a timely fashion to its consumers is also important. Figure 7 shows the status of a master's pair buffer during the course of the program's execution. As can be seen, the master is able to maintain the size of its pair buffer steadily despite the nonuniformity between the rates at which the pairs are generated at producers and processed in consumers. The sawtooth pattern is because of the master's receiving protocol which is to listen to only its consumers when the buffer size exceeds a fixed threshold.

Producer behavior: The primary responsibility of producers is to keep the system saturated with work by generating sequence pairs from trees and sending them to the local master (or the supermaster) in fixed size batches. Figure 8 shows the number of trees processed

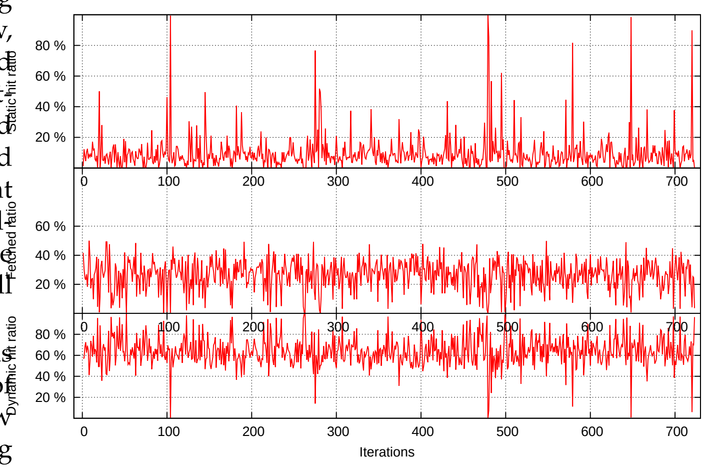


Fig. 5. Statistics of sequence use (and fetch) on an average consumer ($n = 640K$, $p = 1,024$). The topmost chart shows number of sequences successfully found in the local static cache during any iteration. *(Andy: to change this to show #seqs found in static cache/#seqs needed at that iteration — show as percentage)* The next chart shows the number of sequences actually fetched over the network at a given iteration. The bottom chart shows the number of sequences stored in the dynamic cache at a given iteration.

at each producer and the number of pairs generated from those set of trees. Although there is a visible correlation between the number of trees and the number of pairs generated for this run, the correlation no longer holds if the sizes of the trees were to be taken into account (data not shown). *(Andy to verify)* Despite this variability, our implementation is able to balance the workload devoted to pair generation across producers,

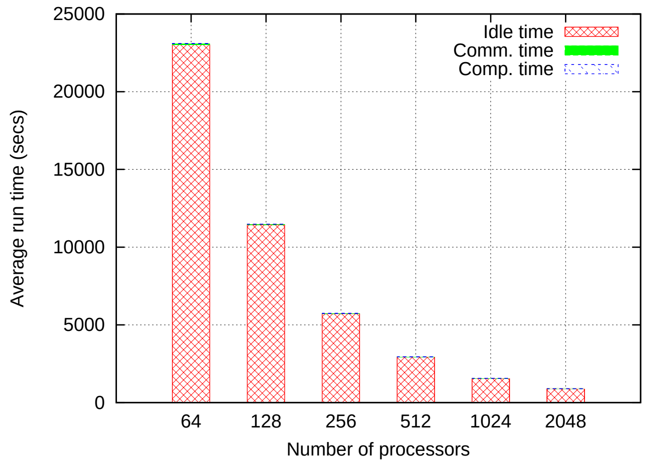


Fig. 6. Run-time breakdown for an average master ($n = 640K$).

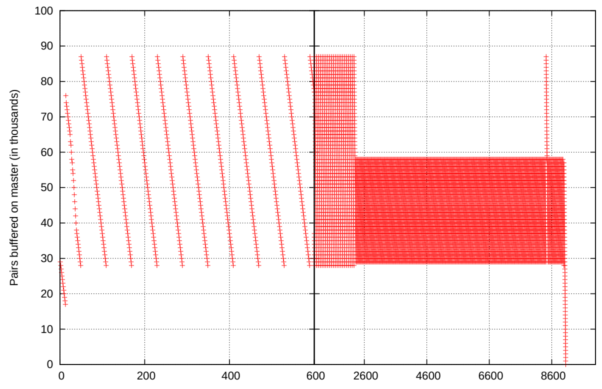


Fig. 7. The status of M_{buf} on a typical master as execution progresses (subgroup size 16).

as can be observed from the run-time chart in Figure 8. This demonstrates the effectiveness of our dynamic tree distribution scheme.

Note that, even with two producers per subgroup, the pair generation time for all producers is ~400s, which is roughly about 25% of the total execution time for the 640K input. This implies that pair generation is still a substantial part of the run-time that could make the merging of the roles of master and producers not so attractive.

Andy: I have assumed here that the fraction of the total time (400s) that a producer is idle is negligible. Is this right? On a related note, I can't understand the following: If the idle time for producers during pair generation is negligible AND since there are two producers per subgroup, it implies that almost 50% of the alignment time is pair generation time. Since there are 13 consumers per subgroup, this implies a ratio of 1:26 for pair gen time: pair alignment time. This seems contradicting to our 1:8 assumption. Can you clarify? Does your 1:8 ratio account for time to communicate and if so, as part of which (align or gen)?

Supermaster behavior: At any given point of time, the system's supermaster is in one of the following states: i)

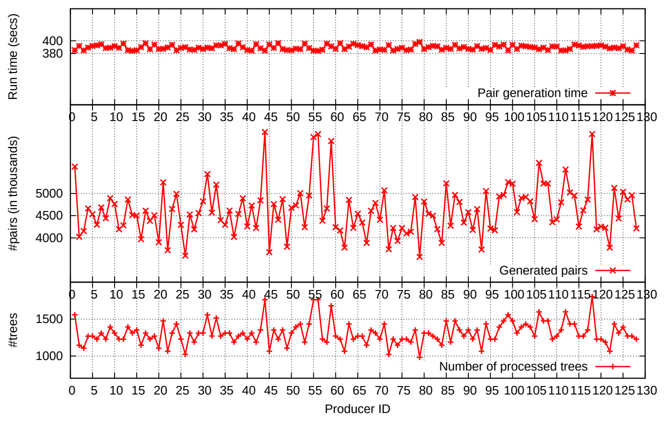


Fig. 8. Plots showing producer statistics on the number of trees processed, the number of pairs generated and the run-time of each of the 128 producers (i.e., 64 subgroups) for the 640K input.

(producer polling) checking for messages from producers, to either receive tree request or pairs for redistribution; ii) *(master polling)* checking status of masters to redistribute pairs. Figure 9 shows that the supermaster spends roughly about 25-30% of its time the polling the producers and the remainder of the time polling the masters. This is consistent with our empirical observations, as producers finish roughly in the first 10%-15% of the program's execution time, and the remainder is spent on simply distributing and computing the alignment workload.

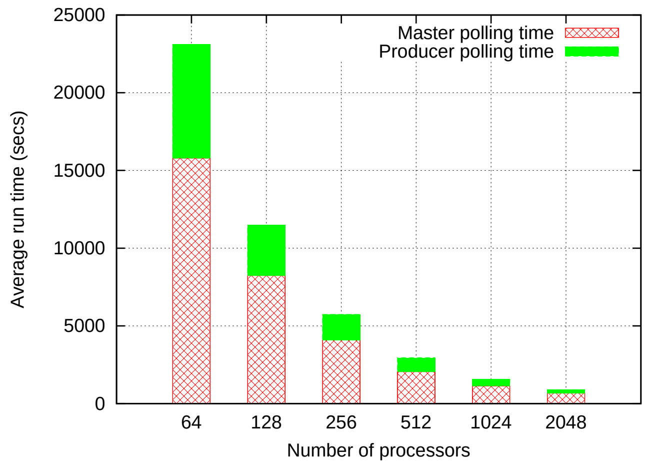


Fig. 9. Run-time breakdown for the supermaster ($n = 640K$).

Does the supermaster's role of redistributing pairs for alignment across subgroups help? To answer this question, we implemented a modified version — one that uses supermaster only for distributing trees to producers but *not* for redistributing pairs generated across groups. This modified implementation was compared against the default implementation, and the results are shown in Figure 10. As is evident, the scheme without pair redistribution creates skewed run-times across subgroups and introduces bottleneck subgroups that slow down the system by up to 40%. This is expected because a subgroup without support for redistributing its pairs may get overloaded with more pairs and/or pairs that need more alignment time, and this combined variability could easily generate nonuniform workload. This shows that the supermaster is a necessary intermediary among subgroups for maintaining overall balance in both pair generation and alignment.

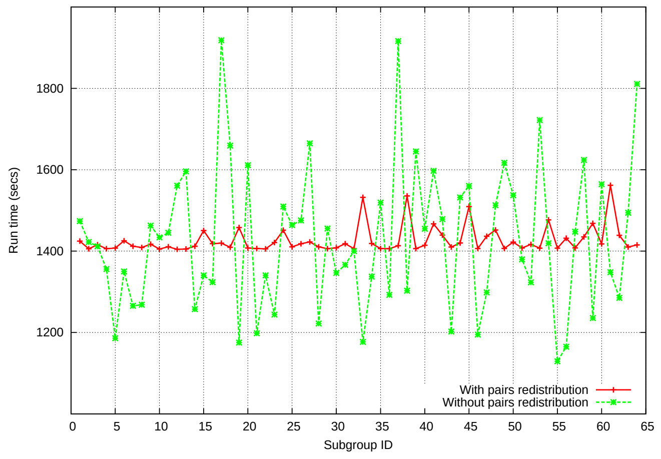


Fig. 10. The distribution of run-time over 64 subgroups (i.e., $p = 1,024$) for the 640K input, with and without the supermaster's role in pair redistribution. The chart demonstrates that the merits of the supermaster's intervention.

4.4 Parameter studies

Andy: to give charts/tables for i) subgroup size study; and ii) multiple producers study.

5 Conclusions

Acknowledgment

We would like to thank the staff at EMSL, PNNL for granting us access to their supercomputer. This research was supported by NSF grant 0916463.

References

[1] S.F. Altschul, W. Gish, and W. Miller *et al.* Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] R. Apweiler, A. Bairoch, and C.H. Wu. Protein sequence databases. *Current Opinion in Chemical Biology*, 8(1):76–80, 2004.

[3] A. Bateman *et al.* The Pfam protein families database. *Nucleic Acids Research*, 32:D138–141, 2004.

[4] A.J. Enright, S. Van Dongen, and S.A. Ouzounis An efficient algorithm for large-Scale detection of protein families. *Nucleic Acids Research*, 30(7):1575–1584, 2002.

[5] J. Handelsman. Metagenomics: Application of genomics to uncultured microorganisms. *Microbiology and Molecular Biology Reviews*, 68(4):669–685, 2004.

[6] S. Henikoff and J.G. Henikoff Amino acid substitution matrices from protein blocks. *Proc. National Academy of Sciences*, 89:10915-10919, 1992.

[7] A. Kalyanaraman, S.J. Emrich, P.S. Schnable, and S. Aluru. Assembling genomes on large-scale parallel computers. *Journal of Parallel and Distributed Computing*, 67:1240–1255, 2007.

[8] E.V. Kriventseva, M. Biswas, and R. Apweiler. Clustering and analysis of protein families *Current Opinion in Structural Biology*, 11(3):334-339, 2001.

[9] P. Weiner. Linear pattern matching algorithm. *Proc. IEEE Symposium on Switching and Automata Theory*, pp. 1–11, 1973.

[10] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[11] V. Olman, F. Mao, H. Wu, and Y. Xu. A parallel clustering algorithm for very large data sets. *IEEE/ACM Transaction on Computational Biology and Bioinformatics*, 5(2):344–352, 2007.

[12] P. Pipenbacher *et al.* ProClust: improved clustering of protein sequences with an extended graph-based approach. *Bioinformatics*, 18(S2):S182–S191, 2002.

[13] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[14] J.C. Venter *et al.* The sequence of the human genome. *Science*, 291(5507):1304–1351, 2001.

[15] C. Wu, and A. Kalyanaraman. An efficient parallel approach for identifying protein families in large-scale metagenomic data sets In *Proc. ACM/IEEE conference on Supercomputing*, pp. 1–10, 2008.

[16] S. Yooseph *et al.* The Sorcerer II Global Ocean Sampling Expedition: Expanding the Universe of Protein Families. *PLoS Biology*, 5(3):e16 doi:10.1371/journal.pbio.0050016, 2007.