pGraph: a scalable parallel algorithm for large-scale protein sequence homology detection

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 [**?**]. 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 predefined 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 [**?**], 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 [**?**] 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., [**?**], [**?**], [**?**], [**?**], [**?**]), 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 [**?**]. Our own experience with the homology detection phase [**?**] 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.

Developing a scalable solution for this problem using

- *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

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 [**?**], 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 **??**. (iii) **Work generation rate:** To avoid a brute-force all-against-all sequence comparison, a string index such as suffix tree [**?**] or look-up table [**?**] is used so that only pairs satisfying an exact matching criterion need to be further evaluated [**?**], [**?**]. 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 **??**. *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 **??** presents the current state of art for parallel sequence homology detection. Section **??** presents our proposed method and implementation details. Experimental results are presented and discussed in Section **??**, and Section **??** concludes the paper.

2 Related Work

Sequence homology between two protein sequences can be evaluated using rigorous optimal alignment algorithms [**?**], [**?**] or heuristic alignment methods such as BLAST [**?**]. Protein sequence clustering is a well researched topic with numerous algorithms and software tools (e.g., [**?**], [**?**], [**?**], [**?**], [**?**]). 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 [**?**] 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 [**?**] 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 [**?**]. 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 [**?**]. This is the motivation behind our newly proposed design which decouples the work generation (producer) from work processing (consumer).

Our observations of a higher complexity for metagenomic protein data when compared to DNA data are

1. stands for "protein sequence homology Graph construction"

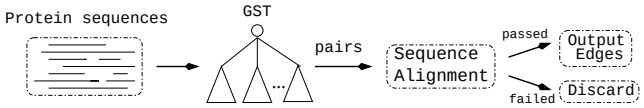

Fig. 1. Tree-based filtering scheme used by our approach for protein sequence homology detection. GST stands for Generalized Suffix Tree.

consistent with other previous studies. For example, in the human genome assembly project [**?**], 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 [**?**].

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., [**?**], [**?**]). There are two independent ways to reduce the computational burden — one is to use algorithmic heuristics (see Section **??**) and another is to use high performance computing (see Section **??**).

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 [**?**]. This approach can filter out a significant fraction of poor quality pairs and thereby reduce the number of pairs aligned significantly (e.g., by $> 70\%$ [**?**]). Figure **??** illustrates the tree based filtering scheme.

To implement exact matching, we use the maximal match detection algorithm described in [**?**]. 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 [**?**] as a string index for the strings in S. The tree index is generated as a forest of

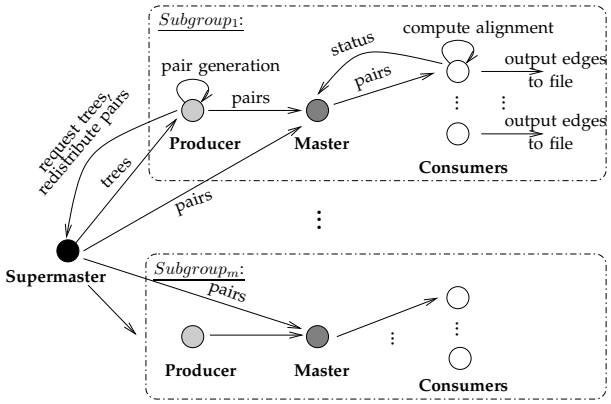

Fig. 2. The hierarchical multiple-master multiple-worker design of the *pGraph* approach showing the interaction of the individual components within and outside the subgroups.

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 **??**). 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.

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 **??**. 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

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

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 [**?**]. 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 **??** 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.

Algorithm 1 Producer

```
1.  Request a batch of subtrees from supermaster
2.  while true do
3.    T_i ← received subtrees from supermaster
4.    if T_i = ∅ 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→M} completed then
13.       Extract at most b_1 pairs from P_buf
14.       send_{P→M} ← Isend extracted pairs to master
15.     end if
16.   until T_i = ∅
17.   Request a batch of subtrees from supermaster
18. end while
19. /* Flush remaining pairs */
20. while P_buf ≠ ∅ do
21.   Extract at most b_1 pairs from P_buf
22.   if send_{P→M} completed then
23.     send_{P→M} ← Isend extracted pairs to master
24.   end if
25.   if send_{P→S} completed then
26.     send_{P→S} ← 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 **??**).

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 **[?]** 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 **??**. 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.  τ: 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| > τ then
6.      msg ← post Recv for consumers
7.    else
8.      msg ← post open Recv
9.      if msg ≡ pairs then
10.       Insert pairs into M_buf
11.       if msg ≡ END signal from supermaster then
12.         break while loop
13.       end if
14.     else if msg ≡ 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 **??**. 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 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

Algorithm 3 Consumer

1. ψ: number of sequences to be cached statically
2. EMPTY, HALF, QUARTER: 0, $\frac{b_2}{2}$ and $\frac{b_2}{4}$ buffer status
3. $S_{cache} \leftarrow$ load ψ sequences from I/O
4. $Recv_{C \leftarrow M} \leftarrow$post nonblocking receive for master
5. **while true do**
6. **if** $|C_{buf}| > 0$ **then**
7. Prefetch sequences $\notin S_{cache}$ for next batch of pairs
8. **if** END signal from master **then**
9. break while loop
10. **else**
11. Extract and align next pair in C_{buf}
12. **end if**
13. **end if**
14. **if** $|C_{buf}| = 0$ **and** $Recv_{C \leftarrow M}$ *not* completed **then**
15. Send **EMPTY** status to master
16. Wait for pairs from master
17. **end if**
18. **if** $Recv_{C \leftarrow M}$ *not* completed **then**
19. **if** $|C_{buf}| = \frac{b_2}{2}$ **then**
20. Send **HALF** status to master
21. **else if** $|C_{buf}| = \frac{b_2}{4}$ **then**
22. Send **QUARTER** status to master
23. **end if**
24. **else**
25. Insert received pairs into C_{buf}
26. $Recv_{C \leftarrow M} \leftarrow$post nonblocking receive for master
27. **end if**
28. **end while**

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.

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; $|P_{buf}| = 5 \times 10^7$; $|M_{buf}| = 6 \times 10^4$; $|C_{buf}| = 6 \times 10^3$; $|S_{buf}| = 4 \times 10^6$.

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

4 Experimental Results & Discussion

4.1 Experimental setup

Our parallel algorithm, *pGraph*, was tested on a set of 640,000 randomly sampled protein sequences from the ocean metagenomic data set (downloaded from the CAMERA portal). This sequence data set has a total of 1.6×10^8 amino acid residues. The mean$\pm\sigma$ sequence length is 255 ± 195 residues; smallest sequence length is 28 and the longest is 5,290. Subsets of smaller size ranging from 20K, 40K, ..., 320K were extracted from the 640K input for scalability tests. The platform used for scalability test was the *Chinook* supercomputer at the EMSL facility in Pacific Northwest National Laboratory. The supercomputer is a 160 TFlops Red Hat Linux system consisting of 2,310 HP DL185 nodes dual socket, 64-bit, Quad-core AMD 2.2 GHz Opteron processors. Each core has access to 4 GB RAM. The network interconnect is Infiniband.

In order to generate the tree index required for all input sets, we used our suffix tree construction code. On the 640K input, the program output a forest containing 167,870 subtrees. The construction was so quick that even a single processor run took only under 7 minutes on the 640K input. For readers interested in the scalability results for the suffix tree construction method, please refer to [**?**].

Even though 4 GB RAM was available to each core, to simulate a larger input scenario, each consumer was allowed to buffer only $\frac{n}{2}$ randomly sampled sequences for all runs reported. In all our experiments, we fixed

the subgroup size to 16 (i.e., 1 producer, 1 master, 14 consumers) based on preliminary empirical tests.

4.2 Performance Evaluation

Table **??** shows the total parallel run-time of *pGraph* both as a function of n and p. As can be observed, the run-time roughly halves each time the number of processors is doubled from $p = 16$ to $1,024$, for all inputs tested. The speedup chart in Figure **??**a confirms this linear scaling behavior, as all speedups were close to ideal. For instance, the speedups on all inputs were well above 900 on 1,024 processors (e.g., 947 for 640K). It is noteworthy that the linear speedup is achieved even on an input as small as 20K on up to 512 processors. These results demonstrate the effectiveness of load distribution strategies used in the proposed algorithm.

Figure **??**b shows the parallel efficiency of the system. As shown, the system is able to maintain an efficiency well over 90% for nearly all inputs and processor sizes, implying a strong scaling behavior. For certain input and processor size combinations, $n = 40K$ and $p = 64$, the efficiency observed was greater than 100% due to superlinear speedups at those points relative to the previous processor size. This could happen when the system does a slightly better job at load balancing when increased from a processor size of $\frac{p}{2}$ to a size of p.

Table **??** also shows the growth of run-time as a function of the input size. The growth in run-time with input size cannot be analytically determined as it is strictly input dependent. For the inputs tested, we observed the run-time to typically increase 2-4 times with doubling of n. Table **??** also shows the number of pairs aligned as a function of the input size. Again, the growth of this term is input-dependent, while observations show that the increase is roughly 2-4 times with doubling of n. There are a few anomalous cases where the run-time growth is disproportionately higher than the growth in the number of pairs aligned — e.g., observe rates of increase from 40K to 80K. We found the cause to be the result of a sudden increase in the input sequence length and standard deviation — e.g., the mean$\pm\sigma$ of input sequence length increased from 205 ± 118 in the 40K input to 256 ± 273 in the 80K input, thereby implying increased computation costs for an average unit of alignment computation.

To better understand the overall system's linear scaling behavior, we conducted a thorough study of the behavior of every component within our system. While we present below the results using $n = 160K$ as a case study, the trends hold for all other inputs.

Consumer behavior: At any given point of time, a consumer is in one of the following states: i) *(align)* compute sequence alignment; or ii) *(I/O)* load sequences for alignment; or iii) *(comm)* send request to master; or iv) *(idle)* wait for master to allocate pairs. Figure **??** shows the breakdown of an average consumer as a function of p. The results show that an average consumer spends more than 95% of its time in alignment computation regardless of the value of p, indicating a healthy sign for scalability. Performing I/O consumes only about 4.5% of the total time even though only half the number of sequences were locally cached. Time spent on all other calls is negligible. As a future improvement, we plan to totally eliminate I/O on consumers by having all the input sequences in S loaded in a distributed fashion among all consumers within each subgroup, and using MPI one-sided communication to fetch the sequences over the network.

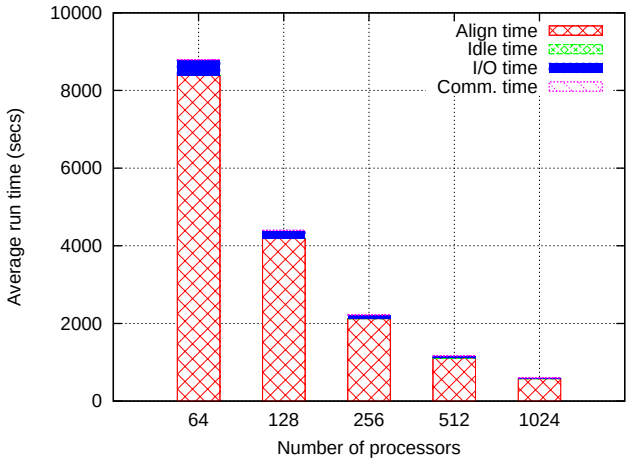

Fig. 4. Run-time breakdown for an average consumer ($n = 160K$).

Master behavior: At any given point of time, a master within a subgroup is in one of the following states: i) *(idle)* waiting for consumer requests; or ii) *(comm)* sending pairs to a consumer; or iii) *(comp)* performing queue and pair buffer operations. Ideally, in the interest of master's availability, one would expect the master to be idle most of the time. Figure **??** shows that this is indeed the case, with each master spending nearly all its time idle. This shows the merit of maintaining manageably small subgroups in our design.

Figure **??** 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. This shows that the priority protocol implemented by the master is highly effective. Maintaining a steady size at the master's pair buffer is critical to ensure that the inflow and outflow of work are regulated. The result is significant especially considering the nonuniformity in the rates at which pairs are generated from the producer, and pairs are aligned at the consumers.

Producer behavior: The primary responsibility of a producer is to ensure that the master is fed with new batch of pairs whenever the latter needs it. Figure **??** shows the number of trees each producer processes within a system of 64 subgroups, and the number of pairs generated locally from those set of trees. The figure validates one aspect of the metagenomic protein

Input number of sequences(n)	Number of processors (p)								Number of pairs (in millions)
	16	32	64	128	256	512	1,024	2048	
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 1
The run-time (in seconds) of *pGraph* 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) as a function of input size.

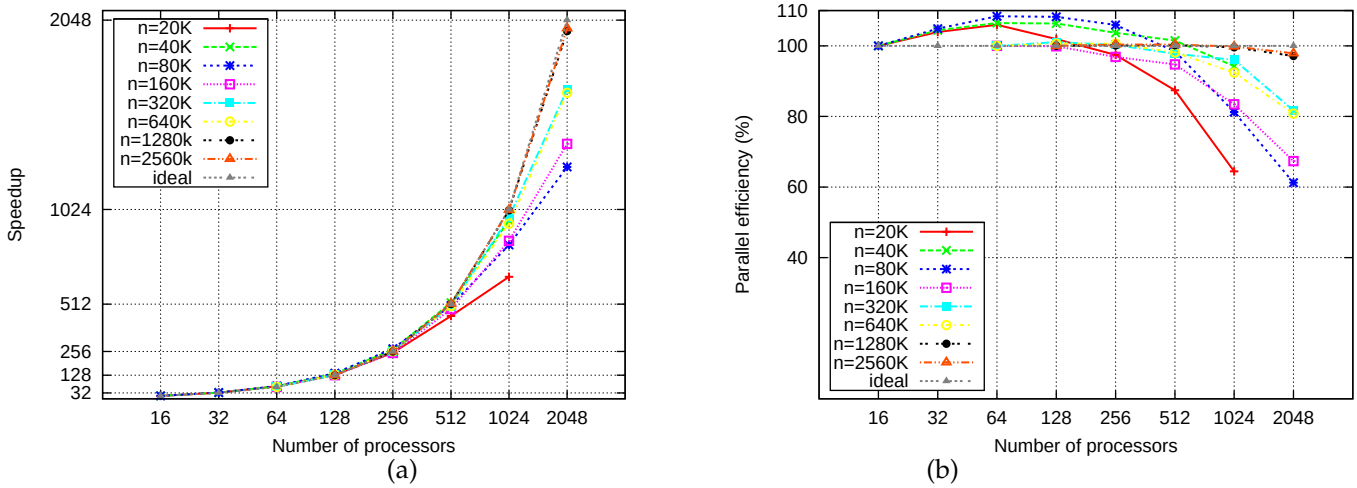

Fig. 3. (a) Speedup and (b) Parallel efficiency of $pGraph$. The speedup computed are relative, and because the code was not run smaller processor sizes for larger inputs, the reference speedups at the beginning processor size were assumed at linear rate — e.g., a relative speedup of 32 was assumed for 160K on 32 processors. This assumption is valid because it is consistent with the speedups observed at the processor size for smaller inputs.

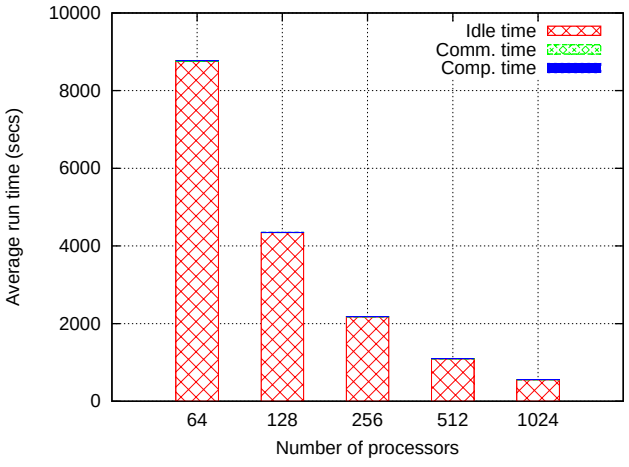

Fig. 5. Run-time breakdown for an average master ($n = 160K$).

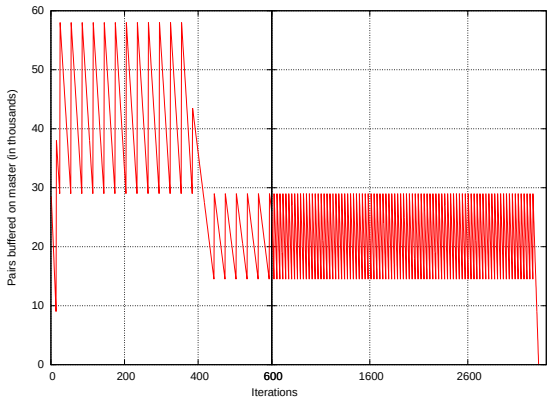

Fig. 6. The status of M_{buf} on a master as execution progresses. The trend holds for all masters tested.

sequence input that the number of trees does not necessarily correlate to the volume of pairs generated, as this term is tree-dependent. However, the uniformity in the pair generation time of all the producers (as shown in the top chart of Figure **??**) demonstrates the effectiveness of our dynamic tree distribution scheme. Note that the

pair generation time for all producers is only $\sim 11\%$ of the total run-time. This observation, coupled with the fact that the masters are idle for more than 99% of their time, indicates a potential for exploring an alternative design where producers and masters can be collapsed. However, such an approach runs the risk of the master becoming less responsive to its consumers and therefore will have to be more carefully investigated.

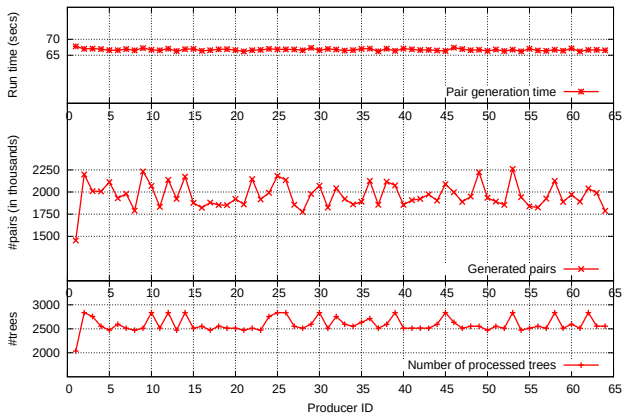

Fig. 7. (a) Plots showing the number of trees processed, the number of pairs generated and the run-time of the producer in each of the 64 subgroups for the 160K input.

Supermaster behavior: At any given point of time, the system's supermaster is in one of the following states: i) *(producer polling)* checking for messages from producers; ii) *(master polling)* checking status of masters. Figure **??** shows that the supermaster spends roughly about 10% to 15% 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.

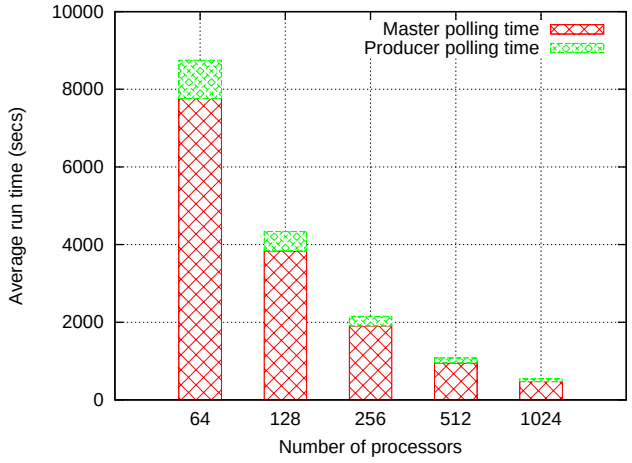

Fig. 8. Run-time breakdown for the supermaster ($n = 160K$).

Does the supermaster really help in ensuring load is balanced across the subgroup boundaries? To answer this question, we made a modified implementation — one that uses supermaster only for distributing trees to producers but *not* for redistributing pairs generated across groups. This modified implementation was tested against the default implementation where the supermaster is allowed to redistribute pairs as well. The results are shown in the plot in the lower half of Figure **??**. As is evident, the scheme without the supermaster's redistribution mechanism pays a heavy penalty in performance as one particular subgroup (ID: 25) acts as a bottleneck for the entire system, delaying the program's completion time by at least 50%. This is expected because a subgroup without support for redistributing its pairs is forced to get all its locally generated pairs aligned by its local consumers, and the combined variability in pair generation and alignment computation is likely to generate nonuniform workload and therefore create parallel bottlenecks. In contrast, the scheme with supermaster's involvement in redistribution would have helped out such bottleneck subgroups by redistributing their pairs to other subgroups (as corroborated by Figure **??**).

The top portion of the Figure **??** shows the difference in the number of pairs generated within a subgroup vs. the number of pairs aligned by that subgroup. The large difference is a manifestation of the supermaster's redistribution mechanism.

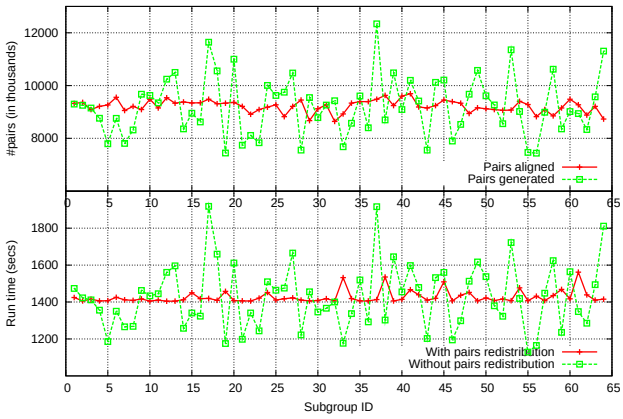

Fig. 9. The distribution of run-time and alignment computation over 64 subgroups (i.e., $p = 1,024$) for the 160K input. The plot below shows the benefit of redistributing pairs across subgroups using the supermaster, and the plot above shows the difference in the number of pairs generated and aligned within each group.

4.3 Other parametric studies

Effect of string caching: To check the effect of caching strings locally at each consumer, we repeated all our tests but with each consumer caching all n sequences in its local memory[3]. As expected there was a marginal improvement in consumer's run-times — e.g., the new run

3. Table not shown due to lack of space.

on 160K using 1,204 processors took 563 s to complete, when compared to 600 s in the code that cached only $\frac{n}{2}$ strings (see Table **??**). However, no significant differences were observed in the speedup or efficiency trends.

Effect of batchsize: We studies the effect of changing the master to consumer batchsize (b_2) on performance. smaller the value of b_2, the better granularity the master has to avoid load imbalance situations within its group (given the variance in alignment computation times). On the other hand, a larger value may prove better at decreasing the total number of communication rounds. In fact, if the batchsize is doubled, the master would communicate with each consumer half as less. To assess this trade-off, we varied batchsize from 500 through 8K and p from 128 to 1,024, and measured the run-time of the system for $n = 160K$ (under the setting of full string cache at the consumers). We observed that the break-even point for this trade-off appeared when batchsize was small (500 to 2K). In the interest of space, we show the results only for $p = 1,024$ in Table **??**, but the trend holds for other processor sizes.

p	Batchsize				
	0.5K	1K	2K	4K	8K
1,024	672	563	583	698	838

TABLE 2
Run-time (in sec) as a result of changing the batchsize from master to consumer on the 160K input.

5 Conclusions

Protein sequence homology is a fundamental problem in protein bioinformatics, one that is becoming increasingly important owing to high potential for discovery and increasingly time consuming (millions of CPU hours) for large-scale data. In this paper, we presented a novel parallel algorithm for parallel protein sequence homology detection. Our approach, *pGraph*, is built on a hierarchical multiple-master multiple-worker model. The strengths of the approach lies in its ability to negotiate effectively between irregularity in the work generation phase and work processing phase, its ability to mask all overheads introduced due to data movement, and its ability to maintain a balanced system. Experimental results demonstrate linear scaling behavior virtually on all inputs tested up to 1,024 processors. Our findings are further corroborated through extensive parametric and system behavior studies. Though developed in the context of protein sequence clustering, the ideas in our method could be applied to a broader range of data-intensive applications where irregularity in work generation and/or work processing could pose serious problems in scalability.

Further improvements such as I/O elimination on the consumers have been planned. More parametric studies such as the effect of changing subgroup size and buffer sizes are required before scaling to larger parallel systems. Moving forward, we plan to test our implementation on much larger metagenomic data sets (e.g., on the 28.6 million ocean metagenomic data), and using more than 1,024 processors.

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.