pGraph: Efficient Parallel Construction of Large-Scale Protein Sequence Homology Graphs

Changjun Wu, Ananth Kalyanaraman, and William R. Cannon

Abstract—Detecting sequence homology between protein sequences 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 the homology between two protein sequences is relatively inexpensive, detecting pairwise homology for a large number of protein sequences can become computationally prohibitive for modern inputs, often requiring millions of CPU hours. Yet, there is currently no robust support to parallelize this kernel. In this paper, we identify the key characteristics that make this problem particularly hard to parallelize, and then propose a new parallel algorithm that is suited for detecting homology on large data sets using distributed memory parallel computers. Our method, called *pGraph*, is a novel hybrid between the hierarchical multiple-master/worker model and producer-consumer model, and is designed to break the irregularities imposed by alignment computation and work generation. Experimental results show that *pGraph* achieves linear scaling on a 2,048 processor distributed memory cluster for a wide range of inputs ranging from as small as 20,000 sequences to 2,560,000 sequences. In addition to demonstrating strong scaling, we present an extensive report on the performance of the various system components and related parametric studies.

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

1 Introduction

Protein sequence homology detection is a fundamental problem in computational molecular biology. Given a set of protein sequences, the goal is to identify all highly "similar" pairs of sequences, where similarity constraints are typically defined using an alignment model (e.g., [18], [26]). In graph-theoretic terms, the protein sequence homology detection problem can be thought of as constructing an undirected graph $G(V, E)$, where V is the set of input protein sequences and E is the set of edges (v_i, v_j) such that the sequences corresponding to v_i and v_j are highly similar.

Homology detection is widely used in nearly all analyses targeted at functional and structural characterization of protein molecules [15]. Most notably, it is used as a precursor step to clustering, which aims at partitioning sequences into closely-knit groups of functionally and/or structurally related proteins called "families". In graph-theoretic terms, this is equivalent of finding variable-sized maximal cliques. However, in practice, owing to errors in sequence data and other biological considerations (e.g., functionally related proteins could differ at the sequence level), the problem becomes one of finding densely connected subgraphs [20], [30], [31]. Protein sequence clustering is gaining importance of late because of its potential to uncover and functionally annotate environmental microbial communities (aka. metagenomic data) [9]. For instance, a single study in 2007 that surveyed an ocean microbiota [31] resulted in the discovery of nearly 4×10^3 previously unknown protein families, significantly expanding the protein universe as we know it.

While there are a number of programs available for protein sequence clustering (e.g., [2], [3], [7], [15], [20]), all of them assume that the graph can be easily built or is readily available as input. However, modern-day use-cases suggest otherwise. Large-scale projects generate millions of *new* sequences that need to be matched against themselves and against sequences already available from previous sequencing projects. For example, the ocean metagenomic project generated more than 17 million new sequences and this set was analyzed alongside 11 million sequences in public protein sequence databanks (for a total of 28.6 million sequences). Consequently, the most time consuming step during analysis was homology detection, which alone accounted for 10^6 CPU hours despite the use of faster approximation heuristics such as BLAST [1] to determine homology. Ideally, dynamic programming algorithms [18], [26] that guarantee alignment optimality should be the method of choice as they are generally more sensitive but the associated high cost of computation coupled with a lack of support in software for coarse-level parallelism have impeded their application under large-scale settings.

- *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

In this paper, we address the problem of parallelizing homology graph construction on massive protein sequence data sets, and one that will enable the deployment of the optimality-guaranteeing dynamic programming algorithms as the basis for pairwise homology detection (or equivalently, edge detection). Although at the offset the problem may appear embarrassingly parallel (because the evaluation of each edge is an independent task), several practical considerations and our own experience [30] suggest it is a non-trivial problem.

Firstly, the problem is data-intensive, even more so than its DNA counterpart. While the known protein universe is relatively small, modern use-cases particularly in metagenomics, in an attempt to find new proteins and families, generate millions of DNA sequences first and then convert them into amino acid sequences corresponding to all six reading frames as protein candidates for evaluation, resulting in a $6\times$ increase in data volume for analysis[1]. Tens of millions of such amino acid sequences are already available from public repositories (e.g., CAMERA [4], IMG/M [10]). Large data size creates two complications.

i) A brute-force all-against-all sequence comparison strategy to detect the presence of edges becomes practically unfeasible due to the quadratic explosion of alignment work. Instead, a filtering based strategy, one that short-lists a smaller subset of sequence pairs based on their potential to pass the alignment criteria prior to actually computing the alignments, needs to be used. In practice, exact matching filters that deploy string data structures such as suffix trees [29] are highly effective in reducing alignment work without compromising on quality [13], [14]. While the time consumed by these advanced filters for pair generation is relatively less when compared to alignment computation, it is certainly *not negligible*. From a parallel implementation standpoint, this means that we could not use a standard work distribution tool — in order to take advantage of these effective sophisticated filters, work generation also needs to be parallelized dynamically alongside work processing.

ii) A large data size also means that the local availability of sequences during alignment processing cannot be guaranteed under the distributed memory machine setting. Alternatively, moving computation to data is also virtually impossible because a pair identified for alignment work could involve arbitrary sequences and could appear in an arbitrary order during generation, both of which are totally data-dependent.

Secondly, handling amino acid sequence data gives rise to some unique *irregularity* issues that need to be contended with during parallelization. i) Assuming "work" refers to a pair of sequences designated for alignment computation, *the time to process each unit of work could be highly variable*. This is because the time for aligning two sequences using dynamic programming takes time proportional to the product of the lengths of the two sequences [18], [26]. And, amino acid sequences tend to have a substantial variability in their lengths, as we will also shown in Section 4. ii) For amino acid data, *the rate at which work is generated could also be non-uniform*. For instance, similar sized portions of the suffix tree index could yield drastically different number of sequence pairs, as the composition of the index is data dependent. *A priori* stocking of pairs that require alignment is simply not an option because of a worst-case quadratic requirement.

Note that these challenges do not typically arise when dealing with DNA. For instance, in genome sequencing projects the lengths of raw DNA sequences derived from modern day DNA sequencers are typically uniform. This coupled with the nature of sampling typically leads to predictable work during generation and processing. In case of metagenomics protein data, the higher variability in sequence lengths is a result of the translation done on the assembled products of DNA assembly (i.e., not raw DNA sequences). Because of this variability, analysis of protein data tends to take longer time and more difficult to parallelize. For example, in the human genome assembly project [28], 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 despite the use of much faster hardware [31].

1.1 Contributions

In this paper, we present a new algorithm to carry out large-scale protein sequence homology detection. Our algorithm, called *pGraph*[2], is designed to take advantage of the large-scale memory and compute power available from distributed memory parallel machines. The output is the set of edges in the sequence homology graph which can be readily used as input for subsequent post-processing steps such as clustering.

Our parallel approach represents a hybrid variant between hierarchical multiple-master/worker and producer-consumer models. The processor space is organized into fixed-size subgroups; each subgroup comprising of possibly multiple "producers" (for pair generation), a local master (for local pair distribution) and a fixed number of "consumers" (for alignment computation). A dedicated global master ("supermaster") manages the workload across subgroups through dynamic load balancing and task reallocation across the subgroups. The producer-consumer division, inspired by the Map Reduce parallel paradigm [6], helps decouple

1. Henceforth for simplicity of exposition, we will use the terms "amino acid sequences" and "protein sequences" interchangeably; although in practice an amino acid sequence need not represent a complete or real protein sequence.

2. stands for "parallel construction of protein sequence homology Graph"

the two major operations in the code, while also providing the flexibility and user-level control to configure the system resources as per input demands. These techniques combined with other base principles in parallel program design for distributed memory computers have allowed us to accommodate the use of quality-enhancing dynamic programming algorithms for evaluating alignments and determining homology at a massive scale.

Experimental results show that *pGraph* achieves linear scaling on a 2,048 processor distributed memory cluster for a wide range of inputs ranging from as small as 20,000 sequences to 2,560,000 sequences. Furthermore, the implementation is able to maintain more than 90% parallel efficiency despite the considerable volume of data movement and the dedication of resources to the hierarchy. In addition to these strong scaling results, we present a thorough anatomical study of the system-wide behavior by its different components. We also comparatively evaluate two models of our algorithm, one that uses I/O and another that uses interprocess communication, for fetching sequences required for alignment computation.

Though presented in the context of protein sequence graph construction, we expect that the basic design principles of our approach can extend to other similar data-intensive scientific applications that involve an element of irregularity or unpredictability concerning work generation, work processing and input data movement. 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 Background and Related Work

Sequence homology between two biomolecular sequences can be evaluated either using rigorous optimal alignment algorithms in time proportional to product of the sequence lengths [18], [26], or using faster, approximation heuristic methods such as BLAST [1] and FASTA [22]. Detecting the presence or absence of pairwise homology for a *set* of protein/amino acid sequences, which is the subject of this article, can be modeled as a homology graph construction problem with numerous applications (e.g., [2], [3], [7], [15], [20]). The rapid adoption of cost-effective, high throughput sequencing technologies is contributing millions of new sequences into sequence repositories [4], [10], [11]. As a result, detection of pairwise homology over these large data sets is becoming a daunting computational task.

An indirect option for implementing homology detection is to use the BLAST program [1], which is a method originally designed for performing sequence database search (query vs. database). This can be done by setting both the query and database sets to the input set of sequences. For instance, the ocean metagenomics survey project [31] used BLAST to perform all-against-all sequence comparison. This took 10^6 CPU hours — a task that was parallelized, albeit in an *ad hoc* manner, by manually partitioning across 125 dual processors systems and 128 16-processor nodes each containing between 16GB-64GB of RAM. Several parallel tools are available for BLAST — the most notable tools being mpiBLAST [5] and ScalaBLAST [19]. These methods run the serial version of NCBI BLAST at their cores, while offering a high degree of coarse-level parallelism and have demonstrated scaling to high-end parallel machines. In addition to being relatively quicker, BLAST also provides a statistical score of significance for comparing a query sequence against a database of sequences.

The use of BLAST based techniques however comes with reduced sensitivity [21], [25] as the underlying algorithm is really an approximation heuristic for computing alignments. Comparatively, the dynamic programming algorithms offer alignment optimality but are generally an order of magnitude slower. Nevertheless, sensitivity is becoming a significant concern of late, especially when dealing with metagenomics data processing because of its highly fragmented nature of sampling. Another less desirable side effect of using BLAST for protein sequence data is that it uses the lookup table data structure which is limited to detection of only short, fixed-length matches between pairs of sequences. This could result in more pairs to be evaluated. Other string data structures such as suffix trees provide more specificity when it comes to the choice of pairs to evaluate due to their ability to detect longer, variable-length matches.

Due to the advantages in using dynamic programming, there has been a flurry of efforts for implementing hardware-level acceleration for optimal pairwise sequence alignment computation on different architectures (reviewed in [24]). However, there is a dearth in research that has targeted at achieving coarse-level parallelism for carrying out millions of such alignment computations. There have been a few efforts for DNA sequence analysis [12], [14], but carrying out protein sequence homology detection at a large-scale has not been addressed to the best of our knowledge.

The purpose of this paper is to investigate the development of a new parallel library that supports large-scale homology detection based on optimal alignment computation. As part of one of our earlier efforts to implement parallel protein clustering [30], we implemented a master-worker framework for homology detection based on optimal alignment computation. Performance evaluation showed that the pairwise sequence homology detection phase, which accounted for more than 90% of the total runtime, failed to scale linearly beyond 128 processors [30]. The cause for the slowdown was primarily the irregular rates at which pairs were generated and processed. Interestingly, the same scheme had demonstrated linear scaling on DNA sequence clustering problems earlier [12], [14], corroborating the higher complexity in analyzing protein sequences.

While there are dynamic load balancing schemes in parallel processing to mitigate the effects of variability in work processing rates, such techniques have traditionally suited compute-intensive applications. The data-intensive characteristic of the homology detection application is additionally challenged with data movement and availability issues. Recently, the mpiBLAST team proposed a highly scalable hierarchical master-worker framework for parallelizing the sequence search operation of BLAST [16]. However, the challenges posed by this problem are different from ours. In addition to being a query-to-database search operation, the unpredictability in BLAST is a result of the variability in query processing times; some queries can take more time the others. On the other hand, the number of task units are predictable as each query sequence is compared against the entire database (i.e., against all its fragments). In our problem, these task units (i.e., pairs to be aligned) are also detemined dynamically and in no predictable order. This results in an explicit task-level separation between work generation (i.e., pair generation from the suffix tree index) and work processing (i.e., alignment), and variability could be expected in both phases. This led us to investigate a different version of a hierarchical master-worker model and combine it with a producer-consumer model inspired by the Map Reduce paradigm.

3 METHODS

Notation: Let $S = \{s_1, s_2, \ldots s_n\}$ denote the set of n input protein sequences. Assuming $|s|$ denotes the length of sequence s and $m = \Sigma_{i=1}^{n}|s_i|$ denotes the total length of all sequences in S. Let $G = (V, E)$ denote a graph defined as $V = S$ and $E = \{(s_i, s_j) \mid s_i$ and s_j are "similar", defined as per pre-defined alignment cutoffs$\}$. We use the term "pair" in this paper to refer to an arbitrary pair of sequences (s_i, s_j).

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

3.1 Pair generation

A brute-force approach to detect the presence of an edge is to enumerate all possible pairs of sequences ($\binom{n}{2}$) and retain only those as edges which pass the alignment test. 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 to be aligned significantly. Suffix tree based filters provide one of the best filters — for instance, anywhere between 67% to over 99% savings for our experiments shown later in the results section (Table 2).

To implement exact matching using suffix trees, we use the optimal pair generation algorithm described in [12], which detects and reports all pairs that share a maximal match of a minimum length ψ. The algorithm first builds a Generalized Suffix Tree (GST) data structure [29] as a string index for the strings in S and then traverses the tree in a bottom-up fashion to generate pairs from different nodes. Suffix tree construction is a well studied problem in both serial and parallel, and any of the standard, serial linear-time construction methods [17], [27], [29] can be used, or efficient distributed memory codes can be used for parallelism [14], [8]. Either way, the tree index can be generated in one preprocessing step and stored in the disk[3].

For our purpose, we generate the tree index as a forest of disjoint subtrees emerging at a specified depth $\leq \psi$, so that the individual subtrees can be independently traversed in parallel to generate pairs. Given that the value of ψ is typically a small user-specified constant, the choice for the cutting depth is restricted too. This implies that the size distribution of the resulting subtrees can be *nonuniform* and is input dependent. It is also to be noted that, even though the pair generation algorithm runs in time bound by the number of output pairs, the process of generation itself could also be *nonuniform* — in that, subtrees of similar size could produce different number of pairs and/or at different rates, and the behavior is input-dependent. For instance, 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. Encouragingly, a small value for the cutting depth is *not* a limiting factor when it comes to the number of subtrees and is sufficient to support a high degree of parallelism. This is because the number of subtrees is expected to grow exponentially with the cutting depth; for instance, a cutting depth as small as 4 on a tree built out of protein sequences (alphabet size 20) could produce around $160K$ trees (as shown in experimental results).

3.2 *pGraph*: Parallel graph construction

We present here an efficient parallel algorithm to construct the homology graph G using the suffix tree index constructed in the previous step and the input sequence set S. 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 $T = \{t_1, t_2, \ldots t_k\}$. The output of *pGraph* should be 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. There

3. Note that there are other, more space-efficient alternatives to suffix trees such as suffix arrays and enhanced suffix arrays, which can also be equivalently used to generate these pairs with some appropriate changes to the pair generation code. We omit those details from this article as pair generation is not the focus of this paper. That said the type of challenges dealt with the tree during parallel pair generation and the solutions proposed would still carry over to these other representations.

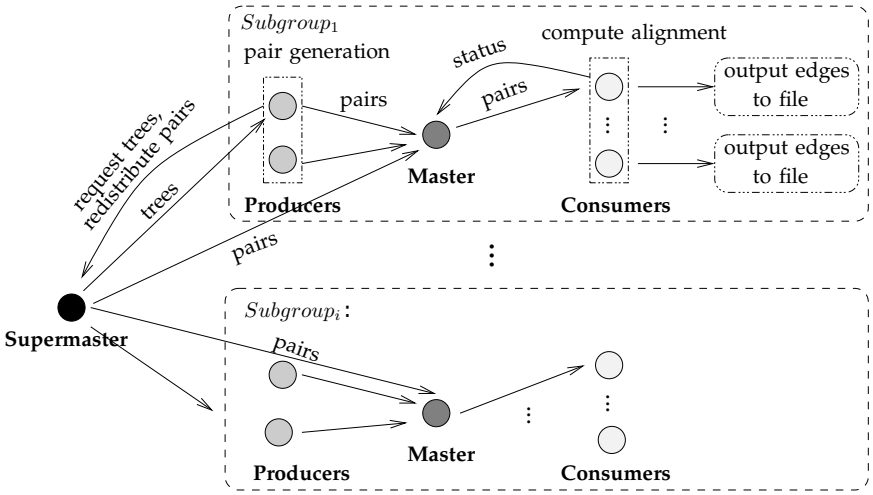

Fig. 1. The overall system architecture for *pGraph*.

are two major operations that need to be performed in parallel: i) generate pairs from the tree index; and ii) compute sequence alignments and output edges if they pass the predefined cutoffs.

Our method uses a hybrid variant between the hierarchical multiple-master/worker model and producer-consumer model to counter the challenges posed by the irregularities in pair generation and alignment rates. The overall system architecture is illustrated in Figure 1.

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 subgroups of size q processors each[4]. Furthermore, each subgroup is internally organized with r processors designated to the role of a *producer*, one processor to the role of a *master*, and c processors to the role of a *consumer*, where $c = q - r - 1$.

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. Unlike traditional models, the overall data flow is from supermaster to the subgroups and also back (for redistribution). In what follows, we describe the various design factors and present algorithms and protocols for each component in the system.

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;

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

3.2.1 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 [12]. The main challenge here is that trees allocated at a producer could result in generation of pairs at a variable rate, although this generation rate is virtually guaranteed to be faster than the rate of consumption (alignment). This is because the pair generation is a simple cross product of sets at any given tree node. To tackle an overactive producer, we maintain a fixed-size pair buffer at each producer ($\sim 80MB$ in our current implementation) and pause the generation process when the buffer is full. This is possible because the pair generation algorithm in [12] is an on-demand method. Furthermore, the tree allocation is left to the supermaster and pair allocation from the producer is left to the local master in our design.

More specifically, we follow the algorithm shown in Algorithm 1. 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 repeats the process by requesting another batch of subtrees from the supermaster. Once there are no more subtrees available, the producers dispatch the rest of 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 group is busy. We show in the experimental section that this 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.   else
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 if
19. end while
20. /* Flush remaining pairs */
21. while P_buf ≠ ∅ do
22.   Extract at most b_1 pairs from P_buf
23.   if send_{P→M} completed then
24.     send_{P→M} ← Isend extracted pairs to master
25.   end if
26.   if send_{P→S} completed then
27.     send_{P→S} ← Isend extracted pairs to supermaster
28.   end if
29. end while
30. Send an END signal to Supermaster
```

3.2.2 Master

The primary responsibility of a master is to ensure all consumers in its subgroup are always busy with alignment computation. Given that pairs could take varying time for alignment, it is more desirable to have the local consumers request for pairs from the local master, than have the master push pairs to its local consumers. Furthermore, to prevent work starvation at the consumers, it is important the master responds in a timely fashion to consumer requests. The hierarchical strategy of maintaining small subgroups helps alleviate this to a certain extent. Another challenge for the master is to accommodate the irregular rate at which its local producers are supplying new pairs. A fast supply rate could overrun the local pair buffer. Ideally, we could store as many pairs as can be stored at the fixed size M_{buf} at the master; however, assuming a protocol where the pairs stored on a local master cannot be redistributed to other subgroups, pushing all pairs into a master node may introduce parallel bottlenecks during the ending stages. The above challenges are 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 producers and consumers. However, once $|M_{buf}|$ reaches a preset limit called τ, the master realizes that its suppliers (could be producers or supermaster) have been overactive, and therefore shuts off listening to its suppliers, while only dispatching pairs to its consumers until $|M_{buf}| \leq \tau$. This way, priority is given to consumer response as long as there are sufficient pairs in M_{buf} for distribution, while at the same time, preventing buffer overruns from happening due to an aggressive producer. On the other hand, when the local set of producers cannot provide pairs in a timely fashion, which could happen at the ending stages when the subtree list has been exhausted, the supermaster could help provide pairs from other subgroups. To allow for this feature, the master opens its listening port to the supermaster as well, whenever it does it to the local producers.

As for serving consumers, the master maintains a priority queue, which keeps track of the states of the work buffers at its consumers based on the latter's most recent status report. The priority represents the criticality of the requests sent from consumers, and is defined based on the number of the pairs left at the consumers' C_{buf}. Accordingly the master dispatches work to the consumers. 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. 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} size: $\frac{1}{2}$-empty, $\frac{3}{4}$-empty, and completely empty, in increasing order of priority.

3.2.3 Consumer

The primary responsibility of a consumer is to compute optimal alignments using the Smith-Waterman algorithm [26] for the pairs allocated to it by its master and output edges for pairs that succeed the alignment test. One of the main challenges in consumer design to ensure the availability of sequences for which alignment is being performed, as the entire sequence set S cannot be expected to fit in local memory for large inputs. To fetch sequences not available in local memory, we considered two options: one is to use I/O (assuming all consumers have access to a shared file system with the sequence file); and the second option is to fetch them over the network intraconnect from other processors that have them. Intuitively, the strategy of using I/O to fetch unavailable sequences can be expected to incur large latency because the batch of sequences to be aligned at any given time could be arbitrary, thereby implying random I/O calls. Unless there is access to a efficient parallel I/O system, such a strategy is not likely to scale to larger system sizes. On the other hand, using the

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** $|M_{buf}| > \tau$ **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 signals to all consumers

intraconnect network could also potentially introduce network latencies, although the associated magnitude of such latencies can be expected to be much less when compared to I/O latencies in practice. In addition, if implemented carefully network related latencies can be effectively masked out in practice (as will be shown in the experimental results).

To test and compare these two models, we implemented both two versions: $pGraph_{nb}$ that uses nonblocking communication calls and $pGraph_{I/O}$ that uses I/O to do sequence fetches. As a third alternative option one can also think of using one-sided communicatoins (instead of nonblocking calls), particularly since the sequence fetches are read-only operations and therefore it becomes unnecessary to involve the remote processor during fetch. However, with one-sided communications, the problem lies in arranging these calls. Performing a separate one-sided call for every sequence that needs to be fetched at any given time is not a scalable option because that would mean that the number of calls is proportional to the number of pairs aligned in the worst case. On the other hand, aggregating the sequence requests by their source remote processor and issuing a single one-sided call to each such processor runs the disadvantage of fetching more sequence information than necessary. This is because one-sided calls can only fetch in windows of contiguously placed sequences and will therefore bring in unwanted sequences that could be between two required sequences. Due to these constraints, we did not implement a one-sided version. In what follows, we present the consumer algorithm that uses network for sequence fetching. The details for the I/O version should immediate follow from the description for $pGraph_{nb}$ and are omitted.

The consumer for $pGraph_{nb}$ follows Algorithm 3. Each consumer maintains a fixed size pair buffer C_{buf} and a sequence cache $\mathcal{S}_c$. The sequence cache ($\mathcal{S}_c$) is divided into two parts: (i) a static sequence cache $\mathcal{S}_c^s$ of size $O(\frac{m}{c})$ (preloaded from I/O); and (2) a fixed-size dynamic sequence cache $\mathcal{S}_c^d$ — a transient buffer to store dynamically fetched sequences from other consumers. During initialization, the consumers within *each subgroup* load the input sequence set S into their respective $\mathcal{S}_c^s$ in a distributed manner such that each consumer gets a unique contiguous $O(\frac{m}{c})$ fraction of input bytes. The assumption that the collective memory of all the c consumers in a subgroup is sufficient to load S is without loss of generality because the subgroup size can be increased proportional to the input size. This strategy of storing the entire sequence set within each subgroup also has the advantage that communications related to sequence fetches can be kept local to a subgroup, thereby reducing hotspot occurrences.

When a consumer receives a batch of new pairs from its master, it first identifies the sequences which are not present in $\mathcal{S}_c^s$ and $\mathcal{S}_c^d$, and subsequently sends out sequence requests to those consumers in the same subgroup that contains those sequences. When a consumer receives a batch of requests from another consumer, it packs the related sequences and dispatch them using a nonblocking send. When the remote sequences arrive, the receiving consumer unpacks the sequences into $\mathcal{S}_c^d$. A separate counter is maintained with each sequence entry in $\mathcal{S}_c^d$ to keep track of the number of pairs in C_{buf} requiring that sequence at any time. If the counter becomes zero at any stage, then the memory allocated for the sequence is released. The dynamic cache is intended to serve as a virtual window of sequences required in the recent past, and could help reduce the net communication volume. In fact we observed that about 60% savings (as will be shown in the results section). Furthermore, the worst-case dynamic sequence cache size is proportional to $2 \times |C_{buf}|$.

The consumer also sends reports of its C_{buf} size to its local master in a timely fashion. The states are $\frac{1}{2}$-empty, $\frac{3}{4}$-empty, and completely empty. Once a status is sent, the consumer continues to process the remaining pairs in C_{buf}. If C_{buf} becomes empty, the consumer sends an

empty message to inform master that it is starving and waits for the master to reply.

Algorithm 3 Consumer

1. $\Delta = \{0, \frac{1}{4}, \frac{1}{2}\}|C_{buf}|$: **empty, quarter, half** buffer status
2. n_s: number of sequences to be cached statically
3. $\mathcal{S}_c^s$: static sequence cache
4. $\mathcal{S}_c^d$: dynamic sequence cache
5. $recv \leftarrow$post nonblocking receive
6. $\mathcal{S}_c^s \leftarrow$ load n_s sequences from I/O
7. **while true do**
8. **if** $recv$ completed **then**
9. **if** Sequence request from consumer c_k **then**
10. Pack sequences and send them out to c_k
11. $recv \leftarrow$post nonblocking receive
12. **else if** Sequences from other consumer **then**
13. $\mathcal{S}_c^d \leftarrow$ unpack received sequences
14. $recv \leftarrow$post nonblocking receive
15. **else if** Pairs from master **then**
16. Insert pairs into C_{buf}
17. Identify sequences to fetch from others
18. Send sequence requests to other consumers
19. $recv \leftarrow$post nonblocking receive
20. **end if**
21. **else**
22. **if** $|C_{buf}| > 0$ **then**
23. Extract next pair (i, j) from C_{buf}
24. **if** $s_i, s_j \in \mathcal{S}_c^s \cup \mathcal{S}_c^d$ **then**
25. Align sequences s_i and s_j
26. Output edges (s_i, s_j) if they pass cutoffs
27. **else**
28. Append pair (i, j) at the end of the C_{buf}
29. **end if**
30. **if** $|C_{buf}| \in \Delta$ **then**
31. Report $|C_{buf}|$ status to master
32. **end if**
33. **end if**
34. **end if**
35. **end while**

3.2.4 Supermaster

The primary responsibility of the supermaster is to ensure that 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 fixed size 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 runtime of the producers (and not necessarily 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 to redirect 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 batch sizes proportionately at this stage. As shown in the 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 signals to all masters

3.3 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; $|P_{buf}| = 1 \times 10^7$; $|M_{buf}| = 6 \times 10^4$; $|C_{buf}| = 6 \times 10^3$; $|S_{buf}| = 4 \times 10^6$. Two sequences are said to be "homologous", if they share a local alignment with a minimum 40% identity

and if the alignment covers at least 80% of the longer sequence.

The software and related documentation is freely available as open source and can be obtained by contacting the authors.

4 Experimental Results

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 [4]. 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 residue 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 4 GB RAM is available at each core, for all runs we set a strict memory upper limit for usage to $O(\frac{m}{c})$ per MPI process, where c is the number of consumers in a subgroup. This was done to emulate a generic use-case on any distributed memory machine including those with limited memory per core. 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 $O(\frac{m}{c})$ fraction of the input. The locally available set of sequences is referred to as the "static sequence cache". Any additional sequence that is temporarily fetched into local memory during alignments is treated as part of a fixed size "dynamic sequence cache".

To generate the suffix tree index required for all input sets, a construction code from one of our earlier developments [14] was used. The suffix tree index for each input is generated as a forest of subtrees, one for each unique $k-mer$ in the input. We used $k = 4$ for all trees. The tree index statistics for the different input sets are shown in Table 1. A single CPU was used to generate the trees for all our experiments because the tree construction is quick and expected to scale linearly with input size, as shown in the table. For larger inputs, any of the already available parallel implementations can be used [8], [14]. Table 1 also shows the number of subtrees generated for each input set. As k was used, the total 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 1:7 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 either of the local sequence caches during alignment at consumers. Figure 2 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. Notably, 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 be primarily attributed to the availability of a fast, parallel I/O system such as Lustre. Such scaling cannot be expected for systems that do not have a parallel I/O system in place.

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

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 3a. All speedups are calculated relative to the least processor size run corresponding to that 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 3b 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

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	160,000	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 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 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.

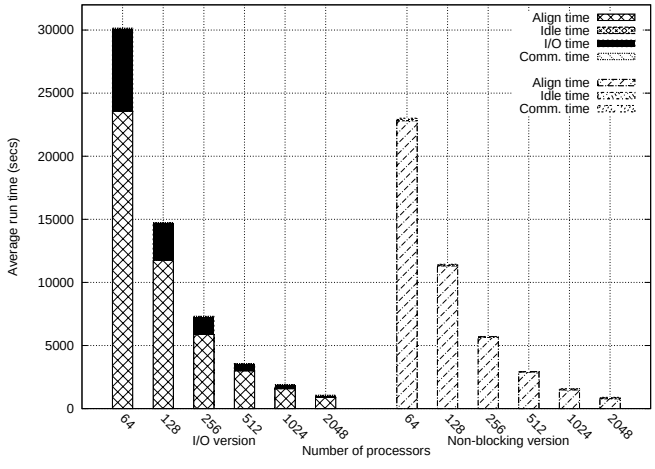

Fig. 2. Comparison of the I/O and non-blocking communication versions of $pGraph$. Shown are the runtime breakdown for an average consumer between the two versions. All runs were performed on the $640K$ input sequence set. The results show the effectiveness of the non-blocking communication version in eliminating sequence fetch overhead.

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 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

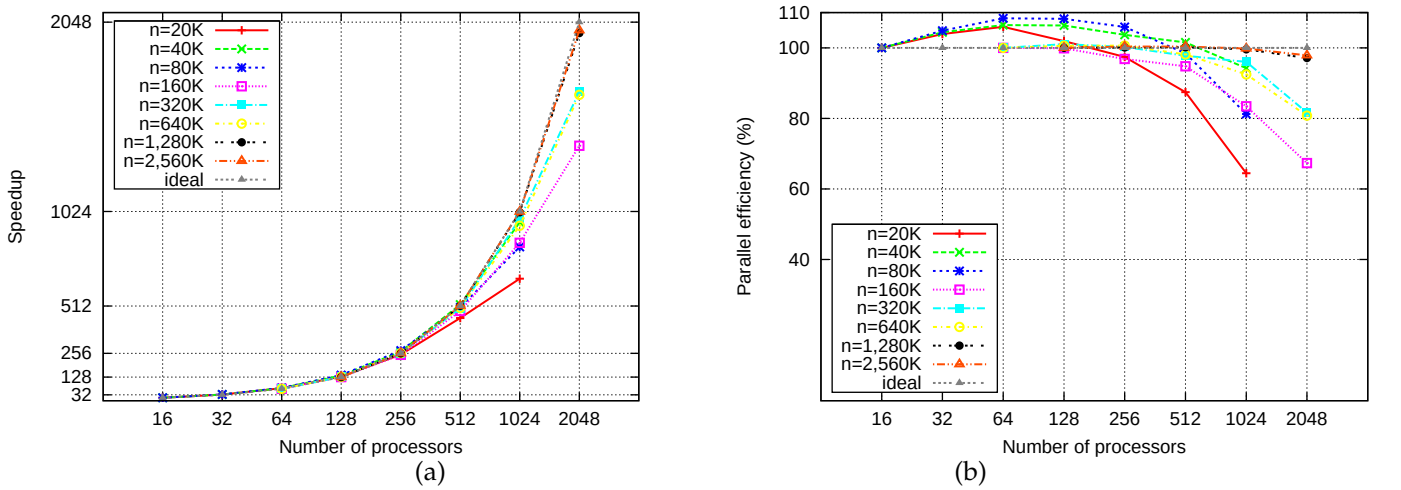

Fig. 3. (a) Speedup and (b) Parallel efficiency of *pGraph*. The speedup and efficiency computed are relative, and because the code was not run on smaller processor sizes for larger inputs, the reference speedups at the beginning processor size were assumed at linear rate — e.g., a relative speedup of 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.

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 2, 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 cache 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. Despite the fact that sequence requests are random and are done asynchronously, the contribution due to communication is negligible both at the senders and receivers. Keeping a small subgroup size (16 in our experiments) is also a notable contributor to the reason why the overhead due to sequence fetches 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 $O(\frac{m}{c})$ characters of the input in the static cache. Figure 4 shows the statistics relating to sequence fetches carried out at every step as the algorithm proceeds at an arbitrarily chosen consumer. 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 needed to be fetched over network. While the middle chart confirms this high volume of 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 cache (bottom chart), which

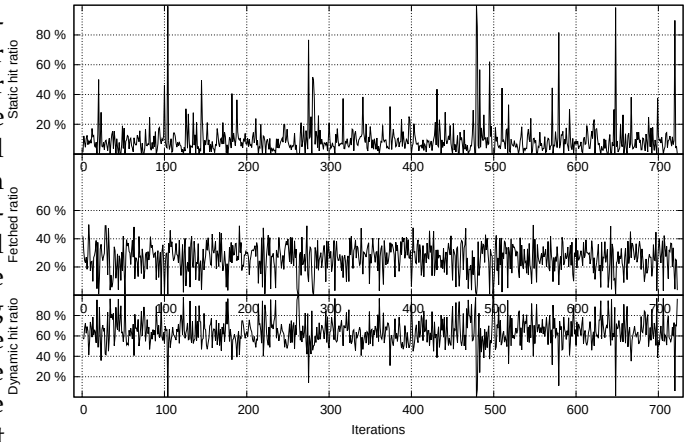

Fig. 4. Statistics of sequence use (and fetch) on an average consumer ($n = 640K$, $p = 1,024$). The topmost chart shows the percentage of sequences successfully found locally in the static cache during any iteration. The next two charts show the corresponding percentages of sequences that needed to be feteched (communicated) from other consumers, and found locally in the dynamic cache, respectively.

serves to reduce the overall number of sequences fetched from other consumers by about 60%.

Master behavior: The master within any 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 the supermaster; or ii) *(comm)* sending pairs to a consumer; or iii) *(comp)* performing local operations to manage subgroup. Figure 5 shows that the master is available (i.e., idle) to serve its local subgroup nearly all of its time. This shows the

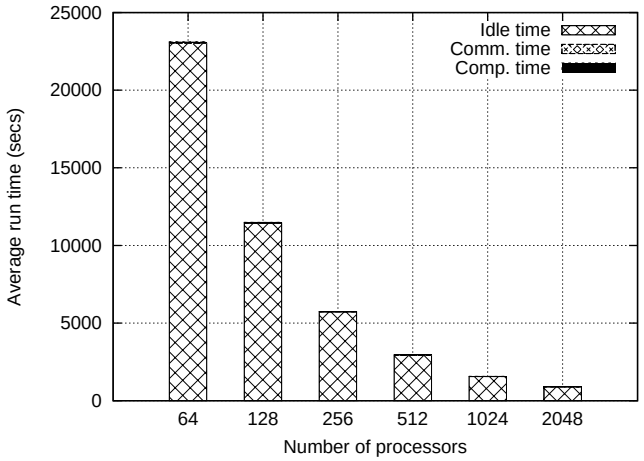

Fig. 5. Run-time breakdown for an average master ($n = 640K$).

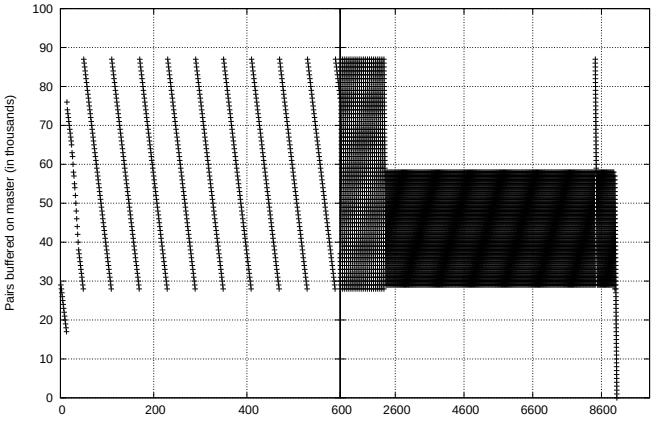

Fig. 6. The status of M_{buf} on a typical master as execution progresses (subgroup size 16).

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 6 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 7 shows the number of trees processed 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

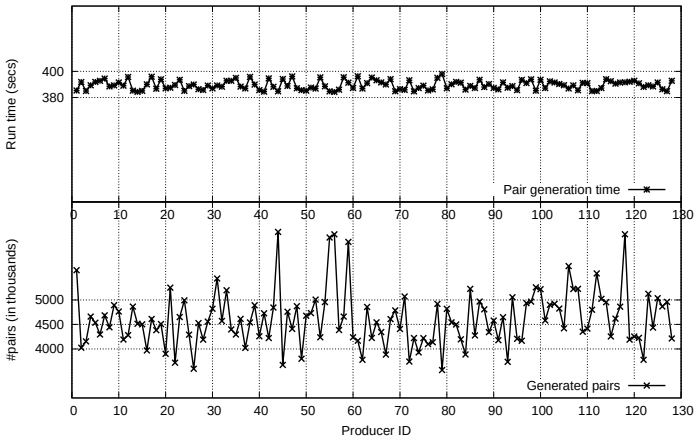

Fig. 7. Plots showing producer statistics on the number of trees processed, the number of pairs generated and the run-time of each of the 128 producers (i.e., 64 subgroups) for the 640K input.

account (data not shown). **(Andy to verify)** Despite this variability, our implementation is able to balance the workload devoted to pair generation across producers, as can be observed from the run-time chart in Figure 7. 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 $\sim$400s, which is roughly about 25% of the total execution time for the 640K input. In general, pair generation occupies a substantial enough part of the run-time and this warrants against merging the roles of master and producers. The increased memory capacity to stock pairs that are pending alignment computation further supports a decoupled design.

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, to either receive tree request or pairs for redistribution; ii) *(master polling)* checking status of masters to redistribute pairs. Figure 8 shows that the supermaster spends roughly about 25% 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 25% of the program's execution time, and the remainder is spent on simply distributing and computing the alignment workload.

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 9. As is evident, the scheme without pair redistribution creates skewed run-times across subgroups and introduces bottleneck subgroups that slow down

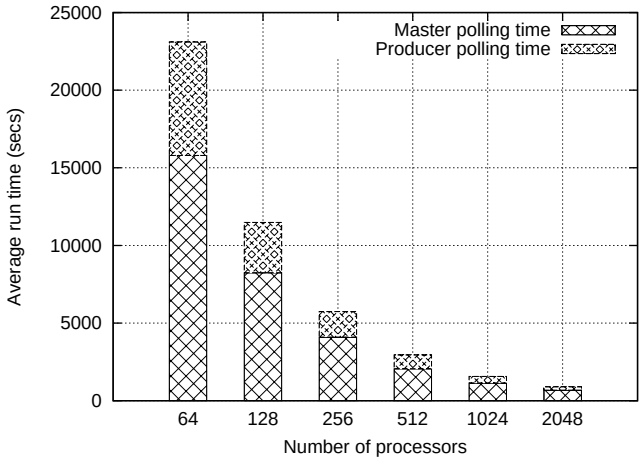

Fig. 8. Run-time breakdown for the supermaster ($n = 640K$).

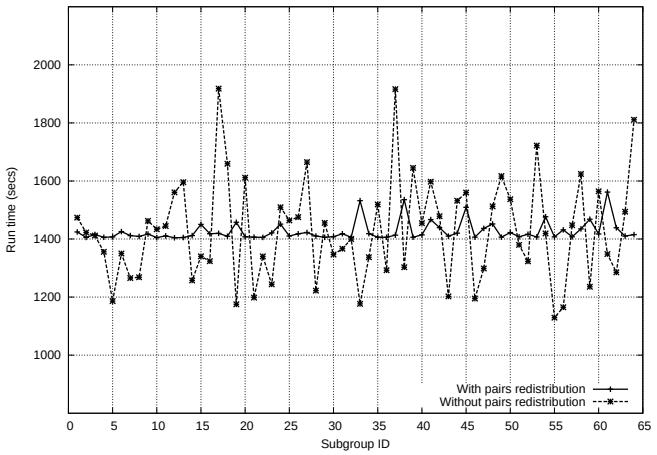

Fig. 9. The distribution of run-time over 64 subgroups (i.e., $p = 1,024$) for the 640K input, with and without the supermaster's role in pair redistribution. The chart demonstrates that the merits of the supermaster's intervention.

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.

4.4 Other parametric studies

We studied the effect of subgroup size on $pGraph_{nb}$'s performance by varying the subgroup sizes from 8, 16, 32, ... to 512, and keeping the total processor size fixed at 1,024 on the 640K input. In all our experiments, a producer:consumer ratio of 1:7 ratio was approximately maintained within each subgroup to reflect the average pair generation to alignment cost ratio. For example, a subgroup with 8 processors will contain 1 producer, 1 master and 6 consumers; whereas a subgroup with 512 processors will contain 64 producers, 1 master and 447 consumers. Note that a larger group size implies less number of subgroups to manage for the supermaster and also more importantly, more number of consumers to contribute to alignment computation. However, as the number of consumers per subgroup increase, the overheads associated with the local master response time and for sequence fetches from other consumers also increase. Therefore, it is increasingly possible that a consumer spends more time waiting (or idle) for data. Figure 10 shows the parallel run-time and the portion of it that an average consumer spends idle waiting either for pairs from the local master or for sequences from other consumers. As expected, we find that the total time reduces initially due to faster alignment computation, before starting to increase again due to increased consumer idle time. The figure also shows an empirically optimal run-time is achieved when the subgroup size is between 16 and 32. Even though this optimal breakeven point is data dependent, the general trend should hold for other inputs as well.

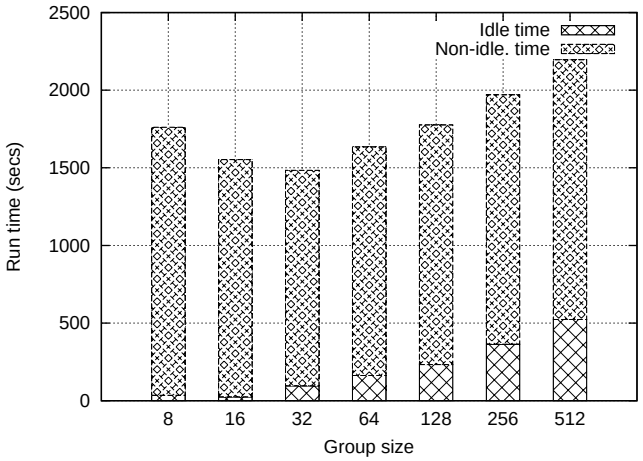

Fig. 10. Chart showing the effect of changing the group size on performance. All runs were performed on the 640K input, keeping the total number of processors fixed at 1,024.

5 Conclusions

In this paper, we presented a novel parallel algorithm and implementation to efficiently parallelize the construction of sequence homology graphs on large-scale protein sequence data sets using distributed memory computers. Coarse-level parallelism for this problem has been lacking in practice. The proposed parallel design is a hybrid of multiple-master/worker and producer-consumer models, which effectively addresses the unique set of irregular computation issues and input data availability issues. The new implementation demonstrates linear scaling on up to 2,048 processors that were tested, for a wide range of input sets tested up to 2.56×10^6 metagenomic amino acid sequences. A

thorough system-wide study by its components further confirms that the trends observed are likely to hold for larger data sets and for larger processor sizes. A key significance of our new implementation is that it enables users to evaluate large collections of protein sequences using the highly sensitive alignment computation algorithms.

To put these results in perspective, consider the following comparison with the ocean metagenomics results [31], which is the largest exercise in protein sequence homology detection to date. The $pGraph_{nb}$ implementation took 7,795 s on 2,048 processors for analyzing a 2.56×10^6 sequence subset of the ocean data set. Based on this, even assuming an absolute worst-case of quadratic explosion of work to 28.6×10^6, we conservatively estimate that $pGraph_{nb}$ would take 566,260 CPU hours. Compare this to the 10^6 CPU hours consumed in [31] despite the use of the faster albeit less-sensitive BLAST heuristic for evaluating homology.

The performance of our current implementation can be further enhanced by augmenting fine-grain parallelism to compute the individual alignments. This can be achieved by substituting the serial alignment code with hardware accelerated alignment computation kernels based on the accelerating platform available at disposal. Such an extension would make the alignment computation much faster and the effect of that along with the possibility of accelerating the pair generation routine needs to be studied in tandem.

The techniques proposed in this paper could also be extended to other data-intensive scientific applications which are posed with similar challenges in the work generation and work processing. The functions for pair generation at the producer and sequence alignment at the consumer could in principle be substituted with application-specific work generation and processing code (similar to specifying mapper() and reducer() functions in Map Reduce). We plan to incorporate this feature and make it available as a generic parallel library that can be plugged into any other data-intensive scientific computing applications.

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 IIS-0916463.

REFERENCES

[1] S.F. Altschul, W. Gish, W. Miller *et al.* Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):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, L. Coin, R. Durbin *et al.* The Pfam protein families database. *Nucleic Acids Research*, 32:D138–D141, 2004.

[4] CAMERA - Community Cyberinfrastructure for Advanced Microbial Ecology Research & Analysis. http://camera.calit2.net. Last date accessed (1/6/2011).

[5] A. Darling, L. Carey and W. Feng. The design, implementation, and evaluation of mpiBLAST. In *Proc. 4th International Conference on Linux Clusters*, 2003.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1):107–113, 2008

[7] 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.

[8] A. Ghoting and K. Makarychev. Indexing genomic sequences on the IBM Blue Gene. In *Proc. ACM/IEEE conference on Supercomputing*, pp. 1–11, 2009.

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

[10] V.M. Markowitz, N.N. Ivanova, E. Szeto, K. Palaniappan, *et al.* IMG/M: a data management and analysis system for metagenomes. *Nucleic Acids Research*, 36(suppl 1):D534–D538, 2008.

[11] The National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/genbank/. Last date accessed (1/6/2011).

[12] A. Kalyanaraman, S. Aluru, V. Brendel, and S. Kothari. Space and time efficient parallel algorithms and software for EST clustering. *IEEE Transactions on Parallel and Distributed Systems*, 14(12):1209–1221, 2003

[13] A. Kalyanaraman, S. Aluru, S. Kothari, and V. Brendel. Efficient clustering of large EST data sets on parallel computers. *Nucleic Acids Research*, 31(11):2963–2974, 2003

[14] 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(12):1240–1255, 2007.

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

[16] H. Lin, X. Ma, W. Feng and N.F. Samatova. Coordinating Computation and I/O in Massively Parallel Sequence Search. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2010.

[17] E. McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976

[18] 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(3):443–453, 1970.

[19] C. Oehmen and J. Nieplocha. ScalaBLAST: A scalable implementation of BLAST for high-performance data-intensive bioinformatics analysis. *IEEE Transactions on Parallel & Distributed Systems*, 17(8):740–749, 2006.

[20] 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.

[21] W.R. Pearson. Searching protein sequence libraries: Comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms. *Genomics*, 11(3):635–650, 1991.

[22] W.R. Pearson, and D.J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, 85(8):2444–2448, 1988.

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

[24] S. Sarkar, T. Majumder, P. Pande, and A. Kalyanaraman. Hardware accelerators for biocomputing: A survey. In *Proc. IEEE International Symposium on Circuits and Systems*, pp. 3789–3792, 2010.

[25] E.G. Shpaer, M. Robinson, D. Yee *et al.* Sensitivity and selectivity in protein similarity searches: a comparison of Smith-Waterman in hardware to BLAST and FASTA. *Genomics*, 38(2):179–191, 1996

[26] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[27] E. Ukkonen A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(1):313–323, 1990

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

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

[30] 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.

[31] S. Yooseph *et al.* The Sorcerer II Global Ocean Sampling expedition: expanding the universe of protein families. *PLoS Biology*, 5(3):432-466, 2007.