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

Changjun Wu, Ananth Kalyanaraman, *Member*, *IEEE*, 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., [31], [41]). 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 [27]. 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." (See Fig. S1 in supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.19, for an illustrative example.) 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 [33], [47], [48]. Protein sequence clustering is gaining importance of late because of its potential to uncover and functionally annotate environmental microbial communities (aka. metagenomic data) [18]. For instance, a single study in 2007 that surveyed an ocean microbiota [48] resulted in the discovery of nearly 4×10^3 previously unknown protein families, significantly expanding the protein universe as we know it. (For more information on metagenomics projects is provided in Section 1.1 of the online supplementary file.)

While there are a number of programs available for protein sequence clustering (e.g., [3], [4], [12], [27], [33]), 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 already available sequences. As a concrete example, the ocean microbiota survey project conducted in 2007 [48] 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 [2] to determine homology. Ideally, dynamic programming algorithms [31], [41] that guarantee alignment optimality should be the method of choice as they are generally more sensitive [34], [40], the associated high cost of computation coupled with a

- *C. Wu is with the Xerox Research Center, 2103 East Ave Apt M, Rochester, Webster, NY 14610. E-mail: changjun.wu@xerox.com.*
- *A. Kalyanaraman is with the School of Electrical Engineering and Computer Science, Washington State University, PO Box 642752, Pullman, WA 99164-2752. E-mail: ananth@eecs.wsu.edu.*
- *W.R. Cannon is with the Pacific Northwest National Laboratory, PO Box 999, J4-33, Richland, WA 99352. E-mail: william.cannon@pnnl.gov.*

Manuscript received 9 Jan. 2011; revised 8 Oct. 2011; accepted 19 Dec. 2011; published online 5 Jan. 2012.
Recommended for acceptance by S. Aluru.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-01-0018.
Digital Object Identifier no. 10.1109/TPDS.2012.19.

lack of support in software for coarse-level parallelism have impeded their application under large-scale settings.

1.1 The Problem and Its Challenges

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 outset the problem may appear embarrassingly parallel (because the evaluation of each edge is an independent task), several practical considerations and our own experience [47] suggest that it needs a nontrivial parallel implementation.

First, 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 [6], IMG/M [20]). Large data size creates two complications.

1. A brute-force solution would be to perform all-against-all pairs comparison—if n denotes the number of sequences and l denotes the average sequence length, then the overall cost in time is $\Theta(n^2l^2)$ using dynamic programming for comparison. Leaving aside the issue of parallelizing distribution of this workload, such a method is simply not scalable for large values of n ($\times 10^6$-10^8), and quite unnecessary, as only a minute fraction is expected to be homologous. That is where the need for sophisticated string indices such as suffix trees [46] arises, as they could effectively prune the search space [25], [26]. 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—instead, work generation also needs to be parallelized dynamically alongside work processing, in order to take advantage of these sophisticated filters.
2. 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.

Second, handling amino acid sequence data derived from metagenomic data sets gives rise to some unique *irregularity* issues that need to be contended with during parallelization.

1. 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 [31], [41]. And, amino acid sequences tend to have a substantial variability in their lengths, as seen in public repositories [6], [21], reported by metagenomic gene finders [30], and confirmed in our experiments (Section 4).
2. For amino acid data, *the rate at which work is generated could also be nonuniform*. In the experimental results section (Section 4), we will show that the time to generate each unit work (pair) using a data structure like suffix tree could be highly variable as the composition of the suffix tree 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 sequencers are typically uniform (e.g., 400 bp for pyrosequencing, 100 bp for Illumina). This coupled with the nature of sampling typically leads to predictable workload 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 [45], 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 [48].

1.2 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 postprocessing 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

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. *pGraph* stands for "*p*arallel construction of protein sequence homology *Graph*."

through dynamic load balancing and task reallocation across the subgroups. The producer-consumer task separation helps decouple the two major operations in the code, while 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 alignment algorithms in determining homology at a massive scale—a task that is deemed impracticable using existing approaches.*

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 percent 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.

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 [31], [41], or using faster, approximation heuristic methods such as BLAST [2], FASTA [35], or USEARCH [11]. Detecting the presence or absence of pairwise homology for a *set* of protein/amino acid sequences, which is the subject of this paper, can be modeled as a homology graph construction problem with numerous applications (e.g., [3], [4], [12], [27], [33]).

An indirect option for implementing homology detection is to use the NCBI BLAST program [2], which is a method originally designed for performing sequence database search (query versus database). However, BLAST is an alignment heuristic which does not guarantee alignment optimality. (See Section 1.2 in the online supplementary file for a more detailed discussion on BLAST-based homology detection.) Dynamic programming algorithms [31], [41] are therefore generally preferred from a sensitivity point of view [34], [40].

The purpose of this paper is to investigate the development of a new parallel library that enables large-scale homology detection based on optimal alignment computation. As part of one of our earlier efforts to implement parallel protein clustering [47], 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 percent of the total runtime, failed to scale linearly beyond 128 processors [47]. 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 [24], [26], corroborating the higher complexity in analyzing protein sequences. This led us to investigate the development of a new parallel tool capable of tackling the irregularities in work generation and processing rates encountered while analyzing large-scale metagenomic amino acid data. Our approach uses a hierarchical master-worker design (reviewed in Section 1.3 of the online supplementary file) in combination with a producer-consumer model.

3 Methods

Notation. Let $S = \{s_1, s_2, \ldots, s_n\}$ denote the set of n input protein sequences. Let $|s|$ denote the length of a sequence s, and let $m = \Sigma_{i=1}^{n}|s_i|$ denote the sum of the length of all sequences in S. Let $G = (V, E)$ denote a graph defined as $V = S$ and $E = \{(s_i, s_j)|s_i$ and s_j are "similar," defined as per predefined 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. Suffix tree based filters provide one of the most effective filters—for instance, anywhere between 67 percent to over 99 percent savings for our experiments (see Table 1).

To implement exact matching using suffix trees, we use the optimal pair generation algorithm described in [24], which detects and reports all pairs that share a maximal match of a minimum length ψ. For our purpose, we generate the tree 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. Implementation level details and discussion are provided in the online supplementary file.

3.2 *pGraph*: Parallel Graph Construction

Given a suffix tree constructed for the input S, we present an efficient parallel algorithm, *pGraph*, to build the corresponding homology graph G. The inputs include the sequence set S and the tree T. The tree is available as a forest of k subtrees, which we denote as $T = \{t_1, t_2, \ldots, t_k\}$. The output is a set of edges which correspond to sequence pairs that pass the alignment test based on user-defined cutoffs. There are two major operations that need to be performed in parallel: 1) generate pairs from T based on the presence of maximal exact matches; and 2) compute alignments and output edges.

TABLE 1
The Runtime (in Seconds) for $pGraph_{nb}$ on Various Input and Processor Sizes

Input number of sequences(n)	Number of processors (p)								Number of pairs aligned (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

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.

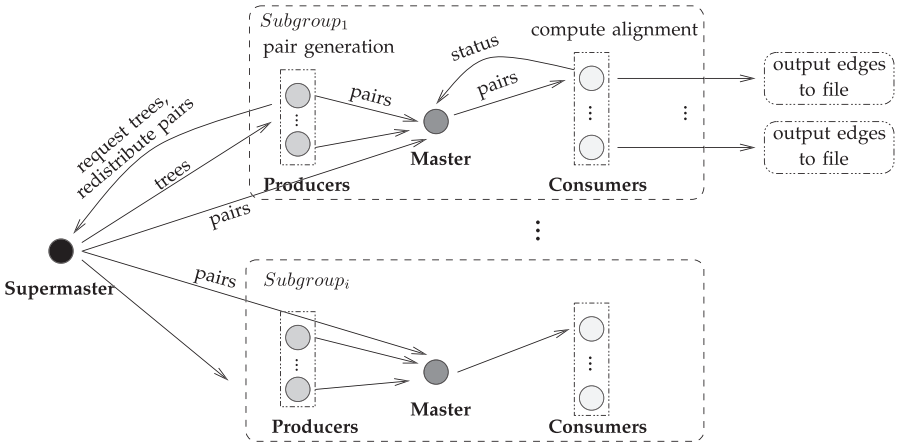


Fig. 1. The overall system architecture for *pGraph*.

Our method is a hybrid 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 Fig. 1.

Given p processors and a small number $q \geq 3$, the parallel system is partitioned as follows: 1) one processor is designated to act as the *supermaster* for the entire system; and 2) the remaining $p-1$ processors are partitioned into subgroups of size q processors each.[3] Furthermore, each subgroup is internally organized as follows: r processors designated to the role of *producers*, one processor to the role of the *master*, and c processors to the role of *consumers*, where $c = q - r - 1$. The ratio of the numbers of producers to consumers can be empirically predetermined through preliminary tests aimed at calculating average time requirements for pair generation to alignment computation.

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). Table S1 in the online supplementary file describes the different buffers and constants used in our design.

3. 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 [24]. 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 (P_{buf}) at each producer and pause the generation process when the buffer is full. This is possible because the pair generation algorithm in [24] is an on-demand method.

The algorithm for a producer is shown in Algorithm 1 (available in the online supplementary file). 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 a fixed-size batch (b_1) of pairs to the master. This is implemented using a nonblocking send so that when the master is not accepting 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.

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 to 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. Overactive generation should be moderated so as to eliminate the risk of overrunning the local pair buffer. Ideally, we could store as many pairs as can be stored at a fixed size buffer (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. All the above challenges are overcome as follows (see Algorithm 2 in the online supplementary file).

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 (say, τ pairs), 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 determined dynamically based on the number of the pairs left to be aligned at the consumer. Accordingly the master dispatches work to the consumers. This is an *on demand* system in which the master 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 three levels of priority (based on 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 align pairs using the Smith-Waterman algorithm [41] 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 to be performed, as it not always realistic to assume that the entire sequence set S can fit into local memory. To fetch sequences not available in local memory, we explored two options: one is to use I/O; 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. On the other hand, using the 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.

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. In what follows, we present the consumer algorithm that uses network for sequence fetching. The details for the I/O version should be evident from the description for $pGraph_{nb}$ and are omitted.

Each consumer maintains a fixed-size pair buffer (C_{buf}) and a sequence cache $\mathcal{S}_c$. The algorithm for each consumer in $pGraph_{nb}$ Algorithm 3 and the important details about the buffer management protocol pertaining to sequence fetches are given in the online supplementary file.

The consumer also reports the number of pairs left in its C_{buf} to its local master in a timely fashion. 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.

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 in the online supplementary file. 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.

3.3 Implementation and Availability

The *pGraph* code was implemented in C/MPI. All parameters described in the algorithm section were set to values based on preliminary empirical tests. Two sequences are said to be "homologous," if they share a local alignment with a minimum 40 percent identity and if the alignment covers at least 80 percent 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 [6]. The sum of the length of the sequences (m) in this set is 3,90,345,218, and the mean $\pm\sigma$ is 152.48 ± 167.25; the smallest sequence has 1 amino acid residue and longest 32,794 amino acid residues. Smaller size subsets containing 20K, 40K, $80K, \ldots$, 1280K were derived and used for scalability tests. Refer to Table S2 in the online supplementary file for more input statistics.

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 (i.e., 8 cores per node) 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. This set of sequences at each consumer is referred to as its "static sequence cache" ($\mathcal{S}_c^s$). Any additional sequence that is temporarily fetched into local memory during alignments is treated as part of a fixed size "dynamic sequence cache" ($\mathcal{S}_c^d$).

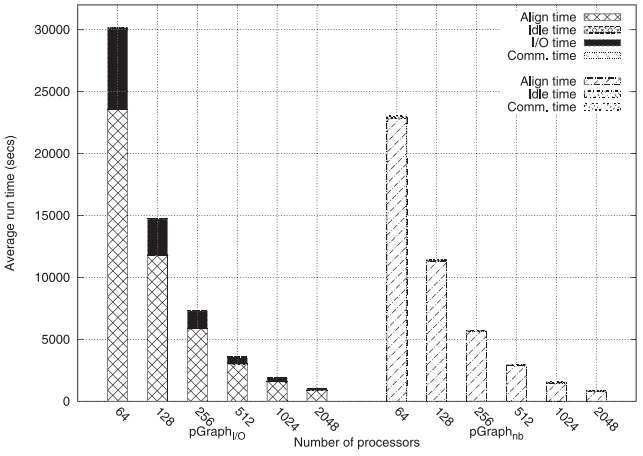


Fig. 2. Comparison of the I/O and nonblocking communication versions of *pGraph*. Shown are the runtime breakdown for an average consumer between the two versions. All runs were performed on the 640K input sequence set. The results show the effectiveness of the nonblocking communication version in eliminating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

4.2 Comparative Evaluation: $pGraph_{I/O}$ versus $pGraph_{nb}$

At first, we compare the two versions of our software, $pGraph_{I/O}$ and $pGraph_{nb}$, which use I/O and nonblocking communication at the consumers, respectively, for dynamically fetching sequences not in the local sequence cache. Fig. 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 percent of the total runtime, and the remaining 20 percent of the time is primarily due to I/O. In contrast, for $pGraph_{nb}$ nearly all of a consumer's runtime was spent performing alignments, with negligible overhead due to nonblocking communication. Consequently, the nonblocking version is 20 percent 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 nonblocking 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 Effect of Changing Subgroup Size

Next, we studied the effect of changing subgroup size on $pGraph_{nb}$'s performance. Subgroup sizes were varied from 8, 16, $32, \ldots$ to 512, while keeping the total number of processors fixed at 1,024 and the input fixed at 640K. In all our experiments, an approximate producer:consumer ratio of 1 : 7 ratio was maintained within each subgroup to reflect the ratio of the average cost of generating a pair to the cost of aligning a pair. For example, a subgroup with eight

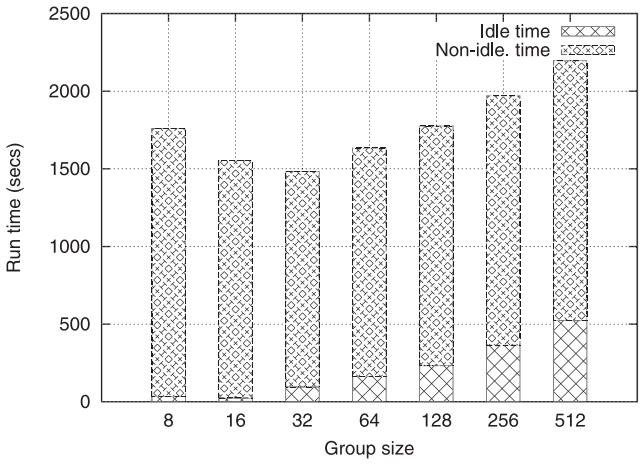


Fig. 3. Chart showing the effect of changing the subgroup size on performance. All runs were performed on the 640K input, keeping the total number of processors fixed at 1,024.

processors will contain one producer, one master and six 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. Fig. 3 shows the parallel runtime 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 runtime 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.

Consequently, in all our experiments that follow, we set the default subgroup size to 16. Each subgroup has 2 producers, 1 master, and 13 consumers.

4.4 Performance Evaluation for $pGraph_{nb}$

Table 1 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 Fig. 4a. 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×) was achieved for the 2,560K data on 2,048 processors. Fig. 4b shows the parallel efficiency of the system. As shown, the system is able to maintain an efficiency above 90 percent for most inputs. Also note that for several inputs, parallel efficiency slightly *increases* with processor size for smaller number of processors (e.g., 80K on $p : 32 \rightarrow 64$). This superlinear behavior can be attributed to the minor increase in the number of consumers (relative to the whole system size)—i.e., owing to the way in which the processor space is partitioned, the number of consumers more than doubles when the whole system size is doubled (e.g., when p increases from 16 to 32, the number of consumers increases from 12 to 25). And this increased availability contributes more significantly for smaller system sizes—e.g., when p increases from 16 to 32, the one extra consumer adds 4 percent more consumer power to the system. The effect, however, diminishes for larger system sizes.

Table 1 also shows runtime 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 1 shows that in some cases the runtime 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 runtime, when n increases from 40K to 80K. Upon further investigation, we found the cause to be the difference in the

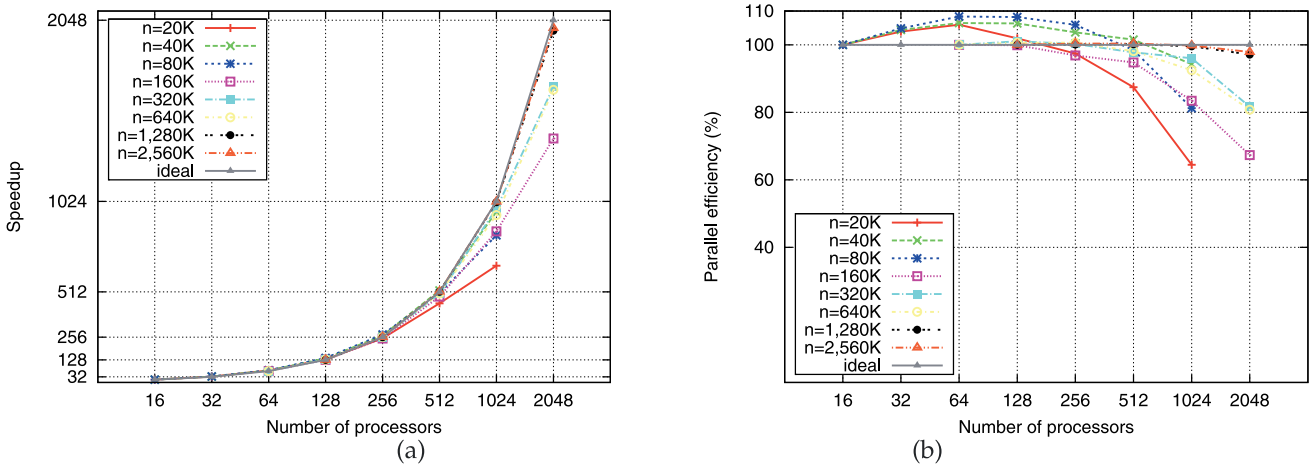


Fig. 4. (a) Speedup and (b) Parallel efficiency of *pGraph*. The speedup and efficiency computed are relative, and because the code was not run on smaller processor sizes for larger inputs, the reference speedups at the beginning processor size were assumed at linear rate—e.g., a relative speedup of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

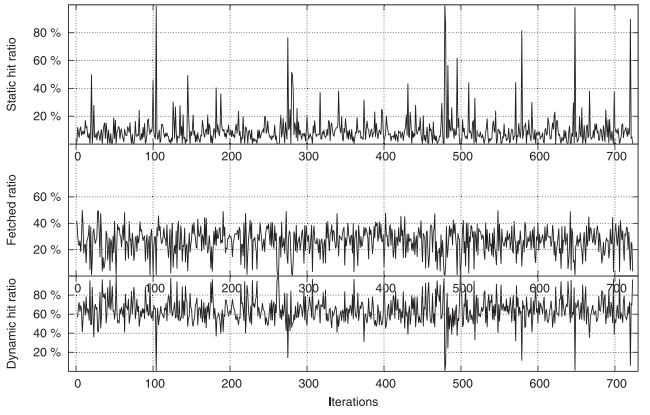


Fig. 5. Statistics of sequence use (and fetch) on an average consumer ($n = 640$K, $p = 1{,}024$). The topmost chart shows the percentage of sequences successfully found locally in $\mathcal{S}_c^s$ during any iteration. The next two charts show the corresponding percentages of sequences that needed to be fetched (communicated) from other consumers, and found locally in $\mathcal{S}_c^d$, respectively.

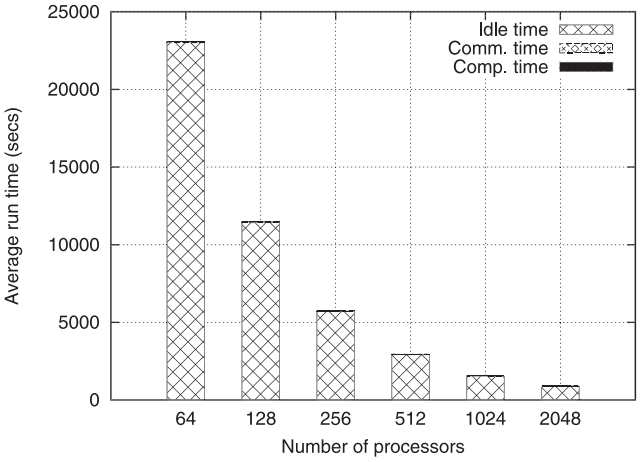


Fig. 6. Runtime breakdown for an average master ($n = 640$K).

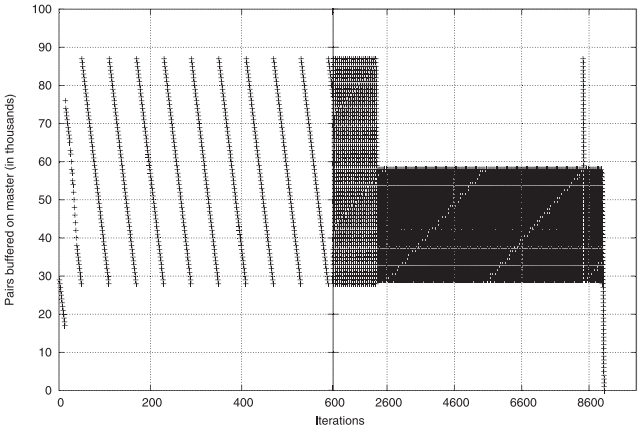
sequence lengths between both these data sets—both mean and standard deviation of the sequence lengths increased from 205 ± 118 for the 40K input to 256 ± 273 for the 80K input, thereby implying an increased cost for computing an average unit of alignment.

To better understand the overall system's linear scaling behavior and identify potential improvements, we conducted a thorough system-wide study. All runs were performing using the $n = 640$K input, as it represents a medium-sized input suitable for a case study.

Consumer behavior. At any given point of time, a consumer in $pGraph_{nb}$ is in one of the following states: 1) *(align)* compute sequence alignment; or 2) *(comm)* communicate to fetch sequences or serve other consumers, or send pair request to master; or 3) *(idle)* wait for master to allocate pairs. As shown in Fig. 2, an average consumer in $pGraph_{nb}$ spends well over 98 percent 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 (as shown in Section 4.3).

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. Fig. 5 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 $\mathcal{S}_c^s$, 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 between the middle and top charts do *not* necessarily correlate to one another. This is because of the temporary availability of sequences in the fixed size dynamic cache $\mathcal{S}_c^s$ (bottom chart), which serves to reduce the overall number of sequences fetched from other consumers by at least 60 percent in most iterations.

Master behavior. The master within any subgroup is in one of the following states at any given point of execution: 1) *(idle)* waiting for consumer requests or new pairs from the local producer(s) or the supermaster; or 2) *(comm)* sending pairs to a consumer; or 3) *(comp)* performing local operations to manage subgroup. Fig. 6 shows that the master is available (i.e., idle) to serve its local subgroup nearly all of its time. This shows the merit of maintaining small subgroups in our design. The effectiveness of the master to provide pairs in a timely fashion to its consumers is also important. Fig. 7 shows the status of a master's pair buffer during the course of the program's execution. As can be seen, the master is able to maintain the size of its pair buffer steadily despite the nonuniformity between the rates at which the pairs are generated at producers and processed in consumers. The sawtooth pattern is a result of the master's



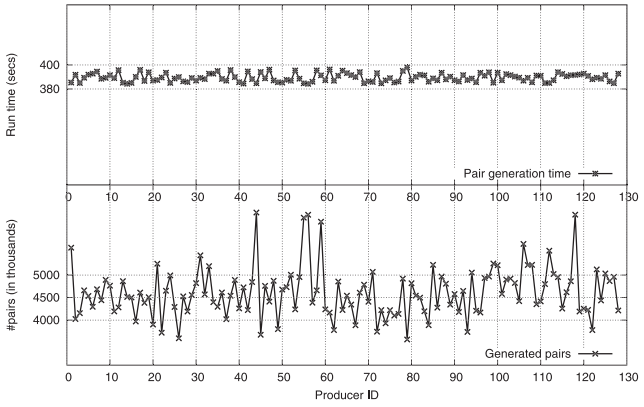


Fig. 8. Plots showing producer statistics on the number of trees processed, the number of pairs generated and the Runtime of each of the 128 producers (i.e., 64 subgroups) for the 640K input.

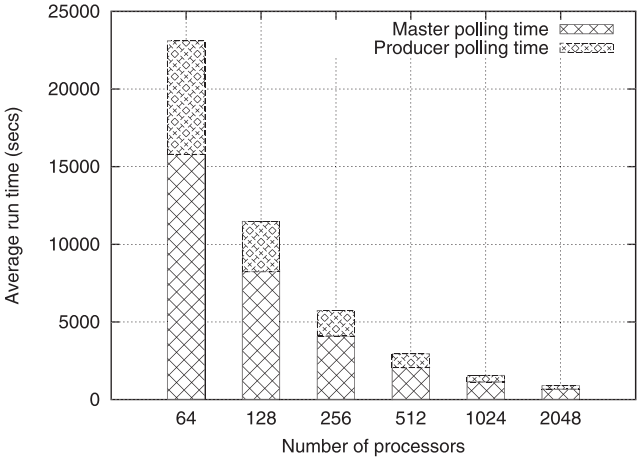


Fig. 9. Runtime breakdown for the supermaster ($n = 640$K).

receiving protocol which is to listen to only its consumers when the buffer size exceeds a fixed threshold τ (set at 60K pairs initially and then reduced to 30K after the local set of producers exhaust pairs).

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. Fig. 8 shows the Runtime and number of pairs generated at each producer. As can be observed, there is considerable variability in the number of pairs supplied by each producer, although all producers finish roughly at the same time. This confirms the irregular behavior of the pair generation phase, which is a result of the irregular overheads associated with tree processing. The results also shows the effectiveness of the dynamic tree distribution strategy deployed by the supermaster.

Note that, even with two producers per subgroup, the pair generation time for all producers is ~ 400 s, which is roughly about 25 percent of the total execution time for the 640K input. For larger data sets, pair generation could consume a substantial part of the Runtime and therefore keeping the roles of the master and producers separate is essential for scalability. Also, the increased memory capacity through using multiple producers 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: 1) *(producer polling)* checking for messages from producers, to either receive tree request or pairs for redistribution; 2) *(master polling)* checking status of masters to redistribute pairs. Fig. 9 shows that the supermaster spends roughly about 25 percent of its time the polling the producers and the remainder of the time polling the masters. This is consistent with other empirical observations, as producers finish roughly in the first 25 percent 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 Fig. 10. As is evident, the scheme without pair redistribution creates skewed Runtimes across subgroups and introduces bottleneck subgroups that slow down the system by up to 40 percent. 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.

4.5 Discussion and Comparison with Other Existing Methods

To put these results in perspective, consider the following comparison with the ocean metagenomics results [48], 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 5,66,260

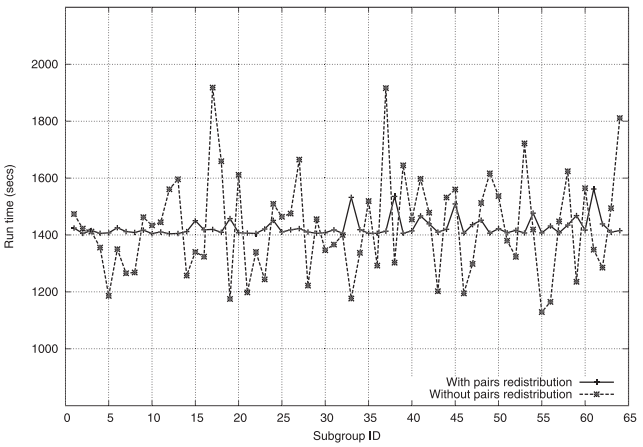


Fig. 10. The distribution of Runtime over 64 subgroups (i.e., $p = 1{,}024$) for the 640K input, with and without the supermaster's role in pair redistribution. The chart demonstrates that the merits of the supermaster's intervention.

CPU hours. Compare this to the 10^6 CPU hours consumed in [48] despite the use of the faster albeit suboptimal BLAST heuristic for evaluating homology.

In a more direct comparison, we compared the results of *pGraph* against the results obtained by running a parallel version of BLAST. The results of this comparative study, presented in Section 3.2 in the online supplementary file in more detail, show that *pGraph* is able to scale to bigger problem sizes, provide better sensitivity, and do so at comparable speeds to a BLAST-based solution despite calculating optimal alignments using dynamic programming.

5 Conclusions

In this paper, we presented a novel parallel algorithm and implementation called *pGraph* to efficiently parallelize the construction of sequence homology graphs from large-scale amino acid sequence data sets based on dynamic programming alignment computation. 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.

Using dynamic programming guarantees optimality of the computed alignments but deploying them at a large-scale has traditionally been deemed infeasible. The results presented in this paper show that building homology graphs based on dynamic programming alignment computation, even for inputs with certain irregular characteristics as exhibited by metagenomic amino acid sequences, may no longer need to be considered a prohibitive operation; and that if effectively combined with smarter exact matching filters and parallelization techniques, such as an endeavor may even become faster and more scalable than existing BLAST-based approaches.

Various extensions and experimental studies have been planned. These include:

1. exploring the use of hardware accelerators to achieve fine-grain parallelism at consumers;
2. developing a hybrid MPI/OpenMP version of the algorithm that can take advantage of multiple core parallelism and shared memory available at the node level;
3. studying the effects of using shared memory and storing all sequences at each consumer (if feasible);
4. testing out on much larger inputs ($n \approx 10^7 - 10^9$).

The online supplementary file contains more details about these planned extensions.

Acknowledgments

The authors thank the anonymous reviewers for the thorough and insightful comments in an earlier version of this manuscript. The research was supported by US National Science Foundation (NSF) grant IIS-0916463. WRC was supported by DOE funding under contracts 57,271 and 54,976. A portion of the research was performed using EMSL, a national scientific user facility sponsored by the Department of Energy's Office of Biological and Environmental Research and located at Pacific Northwest National Laboratory.

References

[1] K. Aida, W. Natsume, and Y. Futakata, "Distributed Computing with Hierarchical Master-Worker Paradigm for Parallel Branch and Bound Algorithm," *Proc. IEEE/ACM Int'l Symp. Cluster Computing and the Grid*, pp. 156-163, 2003.

[2] S.F. Altschul et al., "Basic Local Alignment Search Tool," *J. Molecular Biology*, vol. 215, no. 3, pp. 403-410, 1990.

[3] R. Apweiler, A. Bairoch, and C.H. Wu, "Protein Sequence Databases," *Current Opinion in Chemical Biology*, vol. 8, no. 1, pp. 76-80, 2004.

[4] A. Bateman et al., "The Pfam Protein Families Database," *Nucleic Acids Research*, vol. 32, pp. D138-D141, 2004.

[5] J. Berthold, M. Dieterle, R. Loogen, and S. Priebe, "Hierarchical Master-Worker Skeletons," *Proc. 10th Int'l Conf. Practical Aspects of Declarative Languages*, pp. 248-264, 2008.

[6] CAMERA—Community Cyberinfrastructure for Advanced Microbial Ecology Research & Analysis. http://camera.calit2.net, 2011.

[7] E. Cantú-Paz, "A Survey of Parallel Genetic Algorithms," *Calculateurs Parallèles, Réseaux et Systèmes Répartis*, vol. 10, no. 2, pp. 141-171, 1998.

[8] A. Darling, L. Carey, and W. Feng, "The Design Implementation, and Evaluation of mpiBLAST," *Proc. Fourth Int'l Conf. Linux Clusters*, 2003.

[9] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Comm. ACM*, vol. 51, no. 1, pp. 107-113, 2008.

[10] F.E. Dewhirst et al., "The Human Oral Microbiome," *J. Bacteriology*, vol. 192, no. 19, pp. 5002-5017, 2010.

[11] R.C. Edgar, "Search and Clustering Orders of Magnitude Faster Than BLAST," *Bioinformatics*, vol. 26, no. 19, pp. 2460-2461, 2010.

[12] A.J. Enright, S. VanDongen, and S.A. Ouzounis, "An Efficient Algorithm for Large-Scale Detection of Protein Families," *Nucleic Acids Research*, vol. 30, no. 7, pp. 1575-1584, 2002.

[13] A. Ghoting and K. Makarychev, "Indexing Genomic Sequences on the IBM Blue Gene," *Proc. ACM/IEEE Conf. Supercomputing*, pp. 1-11, 2009.

[14] GOLD "Genomes OnLine Database," http://www.genomesonline.org/, Sept. 2011.

[15] J. Gough, K. Karplus, R. Hughey, and C. Chothia, "Assignment of Homology to Genome Sequences Using a Library of Hidden Markov Models that Represent All Proteins of Known Structure," *J. Molecular Biology*, vol. 313, no. 4, pp. 903-919, 2001.

[16] K. Liolios et al., "The Genomes on Line Database (GOLD) in 2009: Status of Genomic and Metagenomic Projects and Their Associated Metadata," *Nucleic Acids Research*, vol. 38, pp. D346-D354, Nov. 2009.

[17] D.G. Feitelson and L. Rudolph, "Distributed Hierarchical Control for Parallel Processing," *Computer*, vol. 23, no. 5, pp. 65-77, 1990.

[18] J. Handelsman, "Metagenomics: Application of Genomics to Uncultured Microorganisms," *Microbiology and Molecular Biology Rev.*, vol. 68, no. 4, pp. 669-685, 2004.

[19] J. He et al., "A Hierarchical Parallel Scheme for Global Parameter Estimation in Systems Biology," *Proc. 18th Int'l Parallel and Distributed Processing Symp.*, p. 42b, 2004.

[20] V.M. Markowitz et al., "IMG/M: A Data Management and Analysis System for Metagenomes," *Nucleic Acids Research*, vol. 36, no. (suppl 1), pp. D534-D538, 2008.

[21] V.M. Markowitz et al., "The integrated Microbial Genomes System: An Expanding Comparative Analysis Resource," *Nucleic Acids Research*, vol 38, pp. D382-D390, 2010.

[22] Marine Microbial Initiative—Gordon and Betty Moore Foundation, http://www.moore.org/marine-micro.aspx, Sept. 2011.

[23] The Nat'l Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov/genbank/, Sept. 2011.

[24] A. Kalyanaraman, S. Aluru, V. Brendel, and S. Kothari, "Space and Time Efficient Parallel Algorithms and Software for EST Clustering," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 12, pp. 1209-1221, Dec. 2003.

[25] A. Kalyanaraman, S. Aluru, S. Kothari, and V. Brendel, "Efficient Clustering of Large EST Data Sets on Parallel Computers," *Nucleic Acids Research,* vol. 31, no. 11, pp. 2963-2974, 2003.

[26] A. Kalyanaraman, S.J. Emrich, P.S. Schnable, and S. Aluru, "Assembling Genomes on Large-Scale Parallel Computers," *J. Parallel and Distributed Computing,* vol. 67, no. 12, pp. 1240-1255, 2007.

[27] E.V. Kriventseva, M. Biswas, and R. Apweiler, "Clustering and Analysis of Protein Families," *Current Opinion in Structural Biology,* vol. 11, no. 3, pp. 334-339, 2001.

[28] H. Lin, X. Ma, W. Feng, and N.F. Samatova, "Coordinating Computation and I/O in Massively Parallel Sequence Search," *IEEE Trans. Parallel and Distributed Systems,* vol. 22, no. 4, pp. 529-543, Apr. 2011.

[29] E. McCreight, "A Space Economical Suffix Tree Construction Algorithm," *J. ACM,* vol. 23, no. 2, pp. 262-272, 1976.

[30] H. Noguchi, J. Park, and T. Takagi, "MetaGene: Prokaryotic Gene Finding from Environmental Genome Shotgun Sequences," *Nucleic Acids Research,* vol. 34, no. 19, pp. 5623-5630, 2006.

[31] S.B. Needleman and C.D. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *J. Molecular Biology,* vol. 48, no. 3, pp. 443-453, 1970.

[32] C. Oehmen and J. Nieplocha, "ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-intensive Bioinformatics Analysis," *IEEE Trans. Parallel and Distributed Systems,* vol. 17, no. 8, pp. 740-749, Aug. 2006.

[33] V. Olman, F. Mao, H. Wu, and Y. Xu, "A Parallel Clustering Algorithm for Very Large Data Sets," *IEEE/ACM Trans Computational Biology and Bioinformatics,* vol. 5, no. 2, pp. 344-352, Apr.-June 2007.

[34] W.R. Pearson, "Searching Protein Sequence Libraries: Comparison of the Sensitivity and Selectivity of the Smith-Waterman and FASTA Algorithms," *Genomics,* vol. 11, no. 3, pp. 635-650, 1991.

[35] W.R. Pearson and D.J. Lipman, "Improved Tools for Biological Sequence Comparison," *Proc. Nat'l Academy of Sciences of USA,* vol. 85, no. 8, pp. 2444-2448, 1988.

[36] P. Pipenbacher et al., "ProClust: Improved Clustering of Protein Sequences with an Extended Graph-Based Approach," *Bioinformatics,* vol. 18, no. S2, pp. S182-S191, 2002.

[37] N. Ronaldo and E. Zimeo, "A Transparent Framework for Hierarchical Master-Slave Grid Computing," *Proc. CoreGRID,* 2006.

[38] S. Sarkar, T. Majumder, P. Pande, and A. Kalyanaraman, "Hardware Accelerators for Biocomputing: A Survey," *Proc. IEEE Int'l Symp. Circuits and Systems,* pp. 3789-3792, 2010.

[39] R. Seshadri et al., "CAMERA: A Community Resource for Metagenomics," *PLoS Biology,* vol. 5, p. e75, 2007.

[40] E.G. Shpaer et al., "Sensitivity and Selectivity in Protein Similarity Searches: A Comparison of Smith-Waterman in Hardware to BLAST and FASTA," *Genomics,* vol. 38, no. 2, pp. 179-191, 1996.

[41] T.F. Smith and M.S. Waterman, "Identification of Common Molecular Subsequences," *J. Molecular Biology,* vol. 147, no. 1, pp. 195-197, 1981.

[42] E. Talbi and H. Meunier, "Hierarchical Parallel Approach for GSM Mobile Network Design," *J. Parallel and Distributed Computing,* vol. 66, no. 2, pp. 274-290, 2006.

[43] P.J. Turnbaugh et al., "The Human Microbiome Project," *Nature,* vol. 449, no. 18, pp. 804-810, 2007.

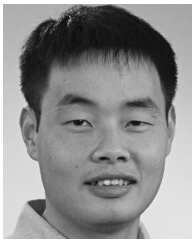
[44] E. Ukkonen, "A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings," *Algorithmica,* vol. 5, no. 1, pp. 313-323, 1990.

[45] J.C. Venter et al., "The Sequence of the Human Genome," *Science,* vol. 291, no. 5507, pp. 1304-1351, 2001.

[46] P. Weiner, "Linear Pattern Matching Algorithm," *Proc. IEEE Symp. Switching and Automata Theory,* pp. 1-11, 1973.

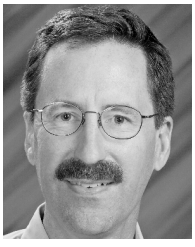
[47] C. Wu and A. Kalyanaraman, "An Efficient Parallel Approach for Identifying Protein Families in Large-Scale Metagenomic Data Sets," *Proc. ACM/IEEE Conf. Supercomputing,* pp. 1-10, 2008.

[48] S. Yooseph et al., "The Sorcerer II Global Ocean Sampling Expedition: Expanding the Universe of Protein Families," *PLoS Biology,* vol. 5, no. 3, pp. 432-466, 2007.



Changjun Wu received the PhD degree in computer science from Washington State University, Pullman, in 2011. Currently, he is working as a researcher at the Xerox Research Center, Webster, New York. His research interests include high-performance computing, graph algorithms, string algorithms, and computational biology. The focus of his dissertation research was on solving large-scale metagenomics problems using parallel computing techniques. He is interested in parallel algorithm design for distributed and shared memory environments.

Ananth Kalyanaraman received the bachelor's degree from Visvesvaraya National Institute of Technology, Nagpur, India, in 1998, and the MS and PhD degrees from Iowa State University, Ames, in 2002 and 2006, respectively. Currently, he is working as an assistant professor at the School of Electrical Engineering and Computer Science in Washington State University, Pullman. He is also an affiliate faculty in the WSU Molecular Plant Sciences graduate program and in the Center for Integrated Biotechnology at WSU. His research interests include high-performance computational biology. The primary focus of his work has been on developing algorithms that use high-performance computing for data-intensive problems originating from the areas of computational genomics and metagenomics. He is a recipient of a 2011 DOE Early Career Award, and two conference best paper awards. He was the program chair for the IEEE HiCOMB 2011 workshop and regularly serves on a number of conference program committees. He is a member of the ACM, IEEE, IEEE Computer Society, and ISCB.



William R. Cannon received the bachelor's degree from UC Santa Cruz in chemistry and the doctorate degree under J.A. McCammon and B.M. Pettitt in Biophysics and Biochemistry at the University of Houston. Currently, he is working as a senior staff scientist in the Computational Biology and Bioinformatics Group, Computational Science and Mathematics Division of the Fundamental and Computational Sciences Directorate at the Pacific Northwest National Laboratory in Richland, Washington and is an adjunct faculty in the School of Electrical Engineering and Computer Science in Washington State University. As a postdoctoral scholar, he worked in the laboratory of S.J. Benkovic and B.J. Garrison on theory and simulation of enzymatic reactions. Before joining PNNL in 2000, he developed data analytic methods for gene expression assays at Monsanto. At PNNL, he works at the intersection of statistical thermodynamics and data analysis to develop methods for understanding and modeling microbial cells, combining data driven and mechanistic approaches.

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

Changjun Wu, Ananth Kalyanaraman, *Member, IEEE,* and William R. Cannon

1 Additional Literature Review

1.1 Microbial sequencing and Metagenomics

Microbial sequencing efforts have been on the rise over the recent years [14], [16], which has led to more than 5,000 genome projects either in complete or draft stages, and with data from several large initiatives (340 metagenomics projects as of September 2011) expected to be available in the next 2-3 years. Notable examples of such large-scale projects include the Human Microbiome Project [43], the Human Oral Microbiome [10] and the Marine microbial initiative by the Moore Foundation [22].

Even a single metagenomics survey project could contribute tens of millions of raw reads and they need to be annotated against sequences from existing gene / protein clusters (>50 million as of September 2011) in community resources such as JGI's Integrated Microbial Genome (IMG / M) [20], [21] and CAMERA [39].

The rapid adoption of cost-effective, high throughput sequencing technologies is contributing millions of new sequences into sequence repositories [6], [20], [23]. As a result, detection of pairwise homology over these large data sets is becoming a daunting computational task. For instance, the ocean metagenomics survey project [48] 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.

1.2 Methods for homology detection

NCBI BLAST program [2] is a method originally designed for performing sequence database search (query vs. database). It can be used for the homology detection problem addressed in this paper, by setting both the query and database sets to the input set of sequences. Several mature parallel tools are available for BLAST — the most notable tools being mpiBLAST [8] and ScalaBLAST [32]. 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 [34], [40], as the underlying algorithm is really an approximation heuristic for computing alignments. Comparatively, the dynamic programming algorithms offer alignment optimality but are generally a couple of orders of magnitude slower. Nevertheless, sensitivity could become a significant concern for metagenomics data because of its highly fragmented nature of sampling. Another limitation of BLAST is its use of the lookup table data structure which is limited to detection of only short, fixed-length matches between pairs of sequences. In a large database of sequences, where short matches could be frequent, the lookup table-based approach could lead to more sequence pairs being evaluated. Other string data structures such as suffix trees [46] 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 optimality guarantee property of dynamic programming algorithms [31], [41], there has been a flurry of efforts for implementing hardware-level acceleration for optimal pairwise sequence alignment computation on different architectures (reviewed in [38]). 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 [24], [26], but carrying out protein / amino acid sequence homology detection at a large-scale has not been addressed to the best of our knowledge.

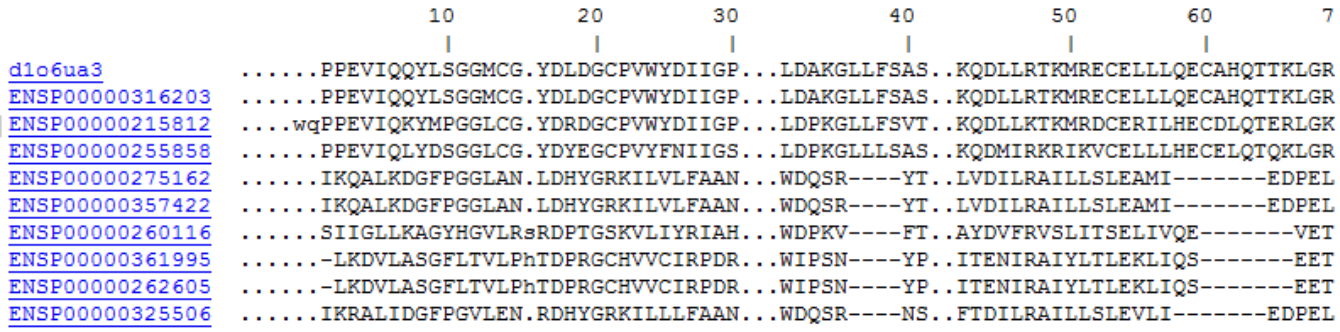


Fig. S1. A partial alignment of the CRAL/TRIO domain family of proteins, which contains 51 protein members (not all shown). The alignments are from the SUPERFAMILY database of structural and functional protein annotations [15].

1.3 Hierarchical master-worker models

Hierarchical master-worker models have been used in a number of scientific applications. While an exhaustive citation is not possible, some examples of their use-cases include: parallel branch and bound algorithms [1]; parallel genetic algorithms and evolutionary computing (surveyed in [7]; numerous grid computing systems (e.g., [37]); mobile network design (e.g., [42]); and building process controllers (e.g., [17]). In bioinformatics, they have been used for parallelizing BLAST [28], [32] and parameter estimation in systems biology [19]. Berthold et al. (2008) analyzes the performance of different hierarchical master-worker configurations.

While there are dynamic load distribution 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 [28]. 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 than 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, the task units (i.e., pairs to be aligned) are also determined 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.

2 Supplementary Information for the Algorithmic Section

2.1 Pair generation using suffix trees

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.

To implement exact matching using suffix trees, we use the optimal pair generation algorithm described in [24], 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 [46] 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 [29], [44], [46] can be used, or efficient distributed memory codes can be used for parallelism [13], [26]. Either way, the tree index can be generated in one preprocessing step and stored in the disk. 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.

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).

A note about the cutoff ψ in the context of amino acid data: Recall that the suffix tree-based pair generation algorithm outlined above is designed to detect any pair of sequences which have at least one maximal (exact) match of length $\geq \psi$. The idea was to set the value of ψ such that the presence of an exact match of length $\geq \psi$ can be used as a necessary-but-not-sufficient condition for homology. (The default value set in our implementation is $\psi = 5$.) However, when applied to amino acid data, it is possible (although with a low probability) that two sequences do *not* contain any exact match of length ψ or greater, and still are homologous. This is due to the fact that amino acid sequences' alignment scores are determined not necessarily by matches or mismatches, but based on substitution matrices (e.g., PAM, BLOSUM). This implies that our suffix tree-based pair generation approach could potentially miss out on a homologous pair of sequences on the basis of the lack of an exact match. While this could result in some loss of sensitivity for *pGraph*, in practice it can still be expected to be more sensitive than other approaches that use alignment heuristics (e.g., BLAST), because of its use of optimaility guaranteeing dynamic programming. In addition, the suffix tree-based algorithm serves as a more effective filter thereby resulting in significant run-time savings. We validate both these claims in the Section 3.2 of Supplementary File.

2.2 One-sided communication for sequence fetching at the consumers

During the design phase of the consumer, besides the I/O and non-blocking versions, we also considered a third option that uses MPI one-sided communications (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.

2.3 Buffer management and sequence fetch protocols at the consumers

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* collectively load the input 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 always be increased to fit the input size if necessary. The characteristic of this application in practice is that thousands of processors are needed to serve the purpose of computation, while the memory on tens of processors are typically sufficient to fit the input sequence data. The 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 potential hotspot scenarios.

When a consumer receives a batch of new pairs from its master, it first identifies the sequences which are neither in $\mathcal{S}_c^s$ nor in $\mathcal{S}_c^d$, and subsequently sends out sequence requests to those consumers in the same subgroup that contain those sequences. When a consumer receives a batch of requests from another consumer, it packs the related sequences and dispatches 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 (by about $\sim$60% as shown

Notation	Description
S	input set of n amino acid sequences
m	sum of the length of all sequences in S
p	total number of processors
q	number of subgroups
r	number of producers per subgroup
c	number of consumers per subgroup ($c = q - r - 1$)
P_{buf}	pair buffer at a producer (10^8 pairs);
M_{buf}	pair buffer at a master (9×10^4 pairs);
C_{buf}	pair buffer at a consumer (6×10^3 pairs);
S_{buf}	pair buffer at the supermaster (4×10^6 pairs);
$\mathcal{S}_c$	set of sequences cached at a consumer at any given point of time ($\mathcal{S}_c = \mathcal{S}_c^s \bigcup \mathcal{S}_c^d$)
$\mathcal{S}_c^s$	set of statically loaded sequences that total in length $O(\frac{m}{c})$ static sequence cache
$\mathcal{S}_c^d$	dynamic sequence cache which contains any sequence ($\notin \mathcal{S}_c^s$) that is temporarily fetched into local memory during alignments
b_1	batch size of a message from producer or supermaster to a master; (30K pairs)
b_2	batch size of a message from a master to a consumer (2K pairs);

TABLE S1
Key notation, buffers and buffer parameters used in *pGraph*. Each buffer is of a certain fixed size in our implementation. The unit for all buffer sizes and batch sizes is in number of pairs. Default values used in the implementation are indicated inside brackets. In our implementation each "pair" of strings stores only the integer ids for the two corresponding strings, implying a fixed size (e.g., 8 bytes on a 32-bit CPU architecture).

in the Experimental Results section). Furthermore, the worst-case dynamic sequence cache size is proportional to $2 \times |C_{buf}|$.

3 Supplementary information for the Experimental Results section

3.1 Generation of the suffix tree index

To generate the generalized suffix tree required for each input, a construction code from one of our earlier developments [26] was used. The suffix tree index for each input is generated as a forest of subtrees, using a cutting depth of 4 amino acid residues (implying, all branches of the tree will be cut at a string depth of 4 starting from the root). The tree statistics for the different input sets are shown in Table S2. 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 [13], [26].

Table S2 also shows the number of subtrees generated for each input set. The reason why the number of subtrees do not increase beyond 160,000 is because we used a cutting depth of 4 to cut the generalized suffix tree into a forest. Given the size of the alphabet (amino acid) is 20, this implies at most $20^4 = 160,000$ subtrees. This is achieved for $n = 640K$ beyond which the increase in input size has an effect only on the size of the individual subtrees and not on the number of subtrees itself. For all our runs, we assume that the tree index is already built using any method of choice and stored in the disk.

3.2 Comparison with parallel BLAST

We compared *pGraph* against ScalaBLAST [32], which is one of the popular parallel implementations of BLAST.

3.2.1 Performance evaluation

In order to enable the all-against-all pairwise homology, we used the "self-BLAST" functionality which compares a database of sequences to itself. There are other settings which impact parallel performance including the setting of subgroup sizes and assigning manager nodes. We then ran Scalablast on the Chinook supercomputer and using the same data sets as inputs. In our experiments we aimed to find how large an input Scalablast can scale to using as many processors as possible. The results are as follows: i) For n=320K, Scalablast took 40 minutes to complete on 512 processors. In comparison, pGraph takes 34 minutes on the same number of processors, despite computing Smith-Waterman alignments. ii) For n=640K and n=1,280K, Scalablast could not finish on 1,024 processors within 24 hours. In comparison, *pGraph* takes 68 minutes and 260 minutes for the 640K and 1,280K inputs, respectively, on the same number of processors. Note also that with *pGraph*, we were able to solve a problem size as large as n=2,560K on 128 processors.

These results show the inadequacy of a BLAST-based solution to address the scaling demands of the homology problem, and demonstrate the merits of using a suffix-tree based index in combination with the parallel techniques of dynamic work generation and processing proposed in the paper. The primary contributing reason for the limited scaling observed in BLAST is likely to be its indexing approach. BLAST uses look-up tables, which only captures short, fixed-length

No. input sequences	Total sequence length (in number of residues)	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 S2
Sequence and suffix tree index statistics for different input sets.

matches. Such a method would lead to an evaluation of significantly more volume of pairs than a suffix-tree based approach used in *pGraph*. However, it was not possible to confirm this reasoning given the reporting functions of the BLAST output. Look-up table-based filtering works generally well when a small set of arbitrary queries is compared against a large database, which is what BLAST is designed for, but becomes inadequate to provide runtime gains when a large database is compared to itself.

3.2.2 Qualitative comparison

We also compared the compositions of the homology graphs output by both *pGraph* and *Scalablast*. Given the general absense of biologically validated benchmarks for metagenomics community data, and owing to the fact that n =320K was the largest input size for which we had a successful run of *Scalablast* due to local job policies, we used the outputs for the 320K input for comparison. In the absense of a biologically validated benchmark, the next best benchmark that one can obtain would be by performing a brute force optimal alignment comparison of all the $\binom{n}{2}$ pairs. However its computation was not feasible due to the sheer magnitude of pairs to be evaluated ($\approx 52 \times 10^9$ pairs). Therefore, we compared the outputs directly against one another, taking the union of edges output by both methods as the reference for sensitivity calculations. To make it a fair comparison, we set identical alignment cutoffs for calling an edge in both methods. The criteria to call an edge between two sequences s_i and s_j were that they contain a local alignment: i) with a minimum 40% identity; and ii) the alignment covers at least 80% of the longer sequence. Even though the BLAST tool does not support these cutoffs directly, it was easy to write a post-processing script that would extract those hits (query, subject) that satisfy these criteria. As for *pGraph*, these are user-specified parameters. In addition, there is a *pGraph*-specific cutoff (ψ) for setting the minimum match length required of any maximal matching pair of sequences to be generated by a suffix tree. We tested the *pGraph* outputs for ψ=4 and ψ=5.

Table S3 shows the results of comparing the homology graphs output by *pGraph* and BLAST. The results can be summarized as follows:

- As can be observed, the number of edges detected by *pGraph* is significantly greater than the number of edges detected by BLAST. In fact, for ψ=5, the *pGraph* detects 24.44% more edges than BLAST while consuming roughly the same amount of time. This is a direct result of using Smith-Waterman based dynamic programming algorithm. And when ψ is decreased to 4, *pGraph* detects 36.38% more number of edges than BLAST, although at the expense of more time.
- However, the set of edges identified by *pGraph* is not a superset of the set of edges identified by BLAST. In fact, the table shows that roughly 65-70% of the BLAST edges are also captured by *pGraph*; on the other hand, BLAST captures only 50% of the *pGraph* edges. The edges missed by *pGraph* but detected by BLAST can be attributed to the differences in the underlying indexing strategies deployed — *pGraph* uses the ψ cutoff which is an exact match criteria; whereas, BLAST uses a lookup table, which for amino acid data is set to 3 residues, although a "match" could be inexact and is determined using a character substitution matrix. On the other hand, the edges missed by BLAST but detected by *pGraph* represent cases where the dynamic programming-based optimal alignment calculation is more sensitive than the BLAST heuristic.
- When compared against the total number of edges detected by both methods, *pGraph* outperforms BLAST by delivering a much improved sensitivity (74-82%) than BLAST (60.19%).

These results collectively show the extent to which sensitivity can be improved by deploying optimality-guaranteeing dynamic programming at a large-scale, and the effectiveness of the *pGraph* algorithm in achieving the same.

4 Future Work and Extensions

4.1 Extending to multicore parallelism

Our parallel design is also amenable to multicore architectures. One way of achieving multicore parallelism is to map every subgroup to a multi-core compute node, and use a new multithreaded implementation (e.g., using

		pGraph	
	BLAST	ψ=5	ψ=4
#Edges identified	5,081,706	6,323,830	6,930,182
% increase in the #edges captured by pGraph relative to BLAST		24.44%	36.38%
#Edges identified by both BLAST and *pGraph*		3,320,505	3,568,789
#Edges identified only by *pGraph*		3,003,325	3,361,393
#Edges identified only by BLAST		1,761,201	1,512,917
Sum of the number of edges identified by both methods			8,443,099
Sensitivity	**60.19%**	**74.90%**	**82.08%**

TABLE S3
Qualitative comparison of the edges output by *pGraph* and *Scalablast* for the n =320K input. Sensitivity of a method is calculated by dividing the number of edges identified by that method by the sum of the number of edges identified by both methods.

OpenMP) to distribute pairs dynamically among the consumer threads. If the set of input sequences would fit in the shared memory of a single node then only one copy of the sequence cache needs to be maintained per node and the cores on that node can share it. This will eliminate the need to communicate among consumers. Note that consumers the sequence cache is read-only and so will not require any locking. Depending on the magnitude of the memory used and the number of cores sharing it, potential memory access bottlenecks may arise which need to be evaluated. On the other hand, if the sequence cache is too big to fit in a single node's shared memory then our current approach of distributing the sequence cache across consumers can be extended into one that distributes the cache across a fixed set of compute nodes and have a dedicated MPI process within each node responsible for communicating the required sequences from a remote node. There are more variants possible which could be explored.

4.2 Hardware acceleration for computing alignments

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.

4.3 Effect of storing all sequences at every consumer

In the current approach, we store only part ($O(\frac{m}{c})$) of the sequence input at every consumer in order to emulate a low-memory-per-core scenario - i.e., it is not fair to assume that the local memory will always be able to hold all the n sequences as the input size grows. However, if there is enough memory available at every consumer to store all the input sequences in its local memory, then does it automatically imply a performance improvement, owing to the elimination of the need for inter-consumer communication? Or could storing multiple copies of the input in a memory shared by a pool of consumers cause memory contention issues that could negate the benefits of storing all sequences locally? To answer this question, we re-ran our experiments on the n =640K and n =1,280K inputs with all the sequences pre-cached by every consumer. Given that the test platform contains 8 cores per node (and 4 GB RAM per core), and assuming that all the 8 cores of a node are occupied by consumers, this means storing 8 copies of the entire input on every node's memory. For instance, in case of the n =1,240K input, this means that roughly about 1.7GB of the total available 32GB RAM is dedicated to storing sequences. The results showed that this scheme of storing the entire input at every consumer, despite eliminating the need for any inter-consumer communication, provided only a marginal 1-2% improvement in the total runtime compared to the default non-blocking implementation ($pGraph_{nb}$). This is not surprising though because the non-blocking communication implementation is anyway so effective that it leaves practically no residual communication overhead for sequence fetches, and therefore achieves comparable performance relative to the full-sequence cached runs. To see the real effect of memory contention issues, we need to test out on much larger data sets. In fact, we could expect that for very large input sizes (n >>2,560K) memory contention could potentially become a serious enough issue that resorting to non-blocking communication may actually be a better option — a claim that obviously needs to be tested out.

4.4 Extension to a generic parallel library

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 abstracted into

application-specific work generation ($producer()$)and work processing ($consumer()$) functions — conceptually similar to the task separation achieved by the $mapper()$ and $reducer()$ functions in MapReduce [9]. Incorporating this feature would enable our parallel framework to be used as a generic parallel library in a broader range of data-intensive scientific computing applications.

pGraph: Algorithmic pseudocodes

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
```

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 signals to all consumers
```

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**

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