Towards Scalable Optimal Sequence Homology Detection

Jeff Daily
Pacific Northwest National Laboratory
email: jeff.daily@pnnl.gov

Sriram Krishnamoorthy
Pacific Northwest National Laboratory
email: sriram@pnnl.gov

Ananth Kalyanaraman
Washington State University
email: ananth@eecs.wsu.edu

***Abstract*—The field of bioinformatics and computational biology is experiencing a data revolution — experimental techniques to procure data have increased in throughput, improved in accuracy and reduced in costs. This has spurred an array of high profile sequencing and data generation projects. While the data repositories represent untapped reservoirs of rich information critical for scientific breakthroughs, the analytical software tools that are needed to analyze large volumes of such sequence data have significantly lagged behind in their capacity to scale. In this paper, we address homology detection, which is a fundamental problem in large-scale sequence analysis with numerous applications. We present a scalable framework to conduct large-scale optimal homology detection on massively parallel supercomputing platforms. Our approach employs distributed memory work stealing to effectively parallelize optimal pairwise alignment computation tasks. Results on 120,000 cores of the Hopper Cray XE6 supercomputer demonstrate strong scaling and up to 2.42×10^7 optimal pairwise sequence alignments computed per second (PSAPS), the highest reported in the literature.**

I. Introduction

The field of bioinformatics and computational biology is currently experiencing a data revolution — the exciting prospect of making fundamental biological discoveries is fueling the rapid development and deployment of numerous cost-effective, high-throughput sequencing technologies that have cropped up in a span of three to four years e.g. Illumina [1]. The result is that the DNA and protein sequence repositories are being bombarded with both raw sequence information (or "reads") and processed sequence information (which could be in the form of DNA and amino acid/open reading frames types of data). Traditional databases such as the NCBI GenBank are reporting that their database sizes are following a Moore's law-like trajectory, roughly doubling every 18 months. In what seems to be a significant paradigm-shift, individual projects are now capable of generating billions of raw sequence data that need to be analyzed in the presence of already annotated sequence information.

An extended version of the well known sequence search model [2] is the all-against-all sequence comparison model, which finds direct use in a number of applications including genome assembly [3], protein family characterization [4] and transcriptomic clustering [5]. A variant can also be used for incremental annotation of new batches of sequences in the presence of already annotated sequences. Despite its broad scope in application, there are hardly any scalable software options for implementing the all-against-all comparison model. Most real world applications have resorted to running heuristic driven approaches and brute-force parallelization to tackle the large data challenge. For instance, one of the largest metagenomics survey projects known till date [6] parallelized the all-against-all homology detection phase by manually partitioning the job across 125 dual processors systems and 128 16-processor nodes each containing between 16GB-64GB of RAM. These approaches can be at best be described *ad hoc* and run the risk of being non-replicable in other settings. Even the limited space of parallel solutions have been shown to scale only up to few thousand cores [3], [7], taking a few hours to solve modest sized problems ($n \approx 10^6$ sequences). Such solutions, given the exponential growth rates in data, are incremental in nature.

Quantifying the scaling requirements: In a nutshell, a recurring (and magnifying) challenge in this area of sequence analysis has become one where the rate of processing the data "lags significantly behind" the rate at which the data is generated. Therefore, in this paper, we ask the following research question: *What would it mean to close the widening gap between data generation and processing? i.e., at what rate should the software run in order to catch up with the data generation?* Any software solution that claims to close the gap between the two rates should at least be able to complete processing of the data in time comparable to the time it took to generate it (if not quicker).

Using pairwise sequence alignments (*PSA*) as the basic unit of measuring the work in all-against-all homology detection, let us consider the following calculations: The Illumina/Solexa HiSeq 2500[1], which is one of the more popular sequencers today, can sequence $\times 10^9$ reads in ~11 days [1]. A brute-force all-against-all comparison would imply $\times 10^{18}$ pairwise sequence alignments (PSAs). However most pairwise comparisons tend to result in poor alignments. Therefore, exact-matching techniques are used in practice to filter out the search space, so that PSA needs to be performed only on the more promising pairs. While the efficacy of the filtering technique is data-dependent, for the purpose of calculation we will assume 99.9% savings (based on our experiences with some of the more effective filters [3], [5], [7]). This would still leave $\times 10^{15}$ PSAs to perform. While the time for each

[1] While there are other faster technologies, we use Illumina as a representative example.

U.S. Government work not protected by U.S. copyright

PSA is another variable and depends on the lengths of the strings being aligned, on an average each PSA tends to take a few milliseconds on state-of-the-art CPUs for reads of a few hundred characters. This implies a total of 277M CPU hours. To complete this scale of work in time comparable to that of data generation (11 days), we need the software to be running on 10^6 cores with close to 100% efficiency. This calculation yields a target of 10^9 PSAPS to achieve, where *PSAPS* is defined as the number of Pairwise Sequence Alignments Per Second.

In addition to achieving large PSAPS counts, achieving fast turn-around times (in minutes) for small- to mid-size problems also become important in practice. This is true for use-cases — in which a new batch of sequences need to be aligned against an already annotated set of sequences, or in analysis involving already processed information (e.g., using open reading frames from genome assemblies to incrementally characterize protein families) — where the number of PSAs required to be performed could be small (when compared to that generated in *de novo* assembly) but needs to be performed multiple times due to the online/incremental nature of the application.

Contributions: In this paper, we set out with the goal of evaluating the feasibility of designing a scalable parallel framework that can achieve orders of magnitude higher PSAPS performance than contemporary software. As a result, we present a work-stealing based parallel approach to perform large-scale homology detection. Multiple attributes distinguish our method from other work: i) We choose the all-against-all model not only for its broad scope of applications, but also because it occupies an upstream phase in most sequence analysis workflows; ii) To ensure high quality of the output, each PSA is evaluated using the *optimality-guaranteeing* Smith-Waterman algorithm [8] (as opposed to the traditional use of faster sub-optimal heuristics); iii) We use protein/putative open reading frame inputs from real world data sets to capture a more challenging use-case where a skewed distribution in sequence lengths can cause nonuniformity in PSA tasks; and iv) To the best of our knowledge, this effort represents the first use of work-stealing for this problem domain.

The key contributions are as follows:

1) Demonstration of homology detection at the largest scale in number of cores $\times 10^5$ cores (previous highest was 2K cores [7]);
2) Use of distributed-memory work stealing for dynamic load balancing (it has been evaluated using benchmarks but not demonstrated in full applications);
3) Highest PSAPS performance (2.42×10^7 at 120,000 cores) reported for *optimal* homology detection — roughly two orders of magnitudes higher than the top PSAPS reported previously (6.59×10^5 on 2048 cores [7]);
4) High efficiencies for small problem sizes (∼75-100% efficiencies for turnaround time of the order of minutes; our results also show that for a given scale larger problem size quickly improves efficiency)

II. Background and Related Work

Sequence homology detection is a fundamental problem in bioinformatics with a pervasive base of applications. Broadly, this is the task of comparing two or more biomolecular sequences in an effort to identify sequence-level similarities and discrepancies. Sequence homology between two biomolecular sequences can be evaluated either using optimal alignment algorithms in time proportional to product of the sequence lengths [8], [9], or using faster, approximation heuristic methods such as BLAST [2], FASTA [10], or USEARCH [11]. Several studies have shown the importance of deploying optimality-guaranteeing methods to ensure high sensitivity (e.g., [12], [13]). Our own study [7] of an arbitrary collection of 320K ocean metagenomics amino acid sequences shows that a Smith-Waterman-based optimal alignment computation could detect 36% more similar pairs than was possible using a BLAST-based run under similar parameter settings. Improving sensitivity of homology detection becomes particularly important when dealing with such environmental microbial data sets [14] due to the sparse nature of sampling in the input. Yet, to the best of our knowledge, none of the large-scale efforts attempted so far have deployed optimality-guaranteeing methods in their analysis, perhaps owing to a lack of scalable tools.

A key challenge in detecting sequence homology at a large-scale in an all-against-all setting stems from the sheer volume of pairs that needs to be aligned. A brute-force implementation on n input sequences would compute $\binom{n}{2}$ pairwise alignments. To reduce this quadratic search space, specialized string data structures such as the lookup table [2] and suffix trees/arrays [15], [16] can be used. These data structures can serve as faster exact matching filters and can be used to identify a subset of sequence pairs based on their "promise" to exhibit high similarity. Subsequently, alignment computation is performed only on the short-listed pairs. While the efficacy of filtering depends on the choice of the underlying data structure, even with the better filters the number of pairs for alignment could be very large. For example, even a modest size input of 2.56×10^6 amino acid/protein sequences from an ocean metagenomics database generated 5.258×10^9 pairs to align using a suffix-tree based filter (∼99.839% savings relative to an exhaustive search) [7].

The state-of-the-art of parallel tools for sequence homology detection can be summarized as follows: Bulk of the parallelization efforts in the past have targeted database search. Most notable massively parallel tools are ScalaBLAST [17] and mpiBLAST [18], both of which have scaled to thousands of cores. Lin *et al.* [19] report linear scaling up to 32,768 BlueGene/P cores, matching 250K DNA sequences (queries) against a set of ∼6M microbial DNA sequences (database) in 12 hours (implying ∼384K CPU hours). To the best of our knowledge, this is the largest scale of study reported so far using the BLAST heuristic. In principle, any database search implementation can be made to fit the all-against-all model (by treating each sequence as being part of both the query

set and database). However, in practice, such a solution may not be appropriate owing to the fact that the bulk of the developmental complexity in parallel BLAST tools originates from the need to gather and sort the database hits per query, and output them to I/O in a ranked fashion. In contrast, under the all-against-all model, the output that is expected is a homology graph (without the requirement to either sort by vertices or by the edges' similarity weights). In other words, the time spent in doing complex parallel I/O operations could be better spent on improving the quality of the alignment computed.

Recently, we took advantage of the above observation, and built a hierarchical master/worker parallel implementation called *pGraph* to compute optimal alignments in parallel [7]. In this implementation, a producer module uses a suffix tree index for identifying promising pairs (based on variable-length exact matches), which are then aligned by a consumer module. Experimental results showed linear scaling up to 2,048 cores (on the EMSL Chinook supercomputer) on an 2.56M input protein sequence set (derived from an ocean metagenomics project). More specifically, the framework aligned 5.258 billion pairs using the Smith-Waterman algorithm [8] in just over 2 hours on 2,048 cores ($\sim$4.5K CPU hours)). To the best of our knowledge, this is the first parallel framework to scale to such high volumes of optimal alignment computations in parallel.

Such efforts focused on long running calculations (many hours) which correspond to significant amounts of computation and potentially simplifying the load balancing challenge. In this paper, we focus on obtaining large PSAPS performance on calculations that run in minutes. The focus on scale coupled with short turnaround times makes load balancing the attendant calculation a particularly challenging problem. To this end, our implementation is the first homology detection application to employ distributed work stealing. Given the above focus on achieving large PSAPS performance, we explore the problem space of scalable dynamic load balancing runtimes in this paper. While ideally these ideas should be coupled with efficient filtering techniques to ensure that the PSAs computed are those that are likely to result in fruitful alignment results, such filtering approaches are not explored here.

Work stealing is a dynamic load balancing technique with well-studied properties in terms of space and time bounds. While traditionally studied in the context of shared memory, recent interest has been devoted to studying work stealing in a distributed memory setting. This is in part to work stealing's ability to dynamically adapt to variations in the execution environment due to load imbalance, faults, or power/energy considerations. Distributed memory work stealing [20]–[22] has been evaluated in the context of the unbalanced tree search benchmark [23], and benchmarks from multi-resolution methods [24], self-consistent field calculations [25], and tensor contraction expressions arising in Coupled Cluster methods [26]. To the best of our knowledge, distributed memory work stealing has not been employed in scaling a full application in general, and homology detection in particular.

Optimized implementations of single-node optimal alignment computation have been developed for a variety of systems including vector instruction sets [27], GPUs [28], [29], FPGAs [30], and other specialized architectures [31], [32]. Our work complements such efforts and has the capacity to leverage them toward a scalable distributed memory parallel homology detection implementation.

III. Optimal Homology Detection

Notation: Let $S = \{s_1, s_2, \ldots s_n\}$ denote a set of n sequences over a fixed input alphabet Σ. For DNA, $\Sigma = \{a, c, g, t\}$. Alternatively, for amino acid/protein sequences, the alphabet contains 20 symbols (one for each of the 20 amino acids). Let $|s|$ denote the length of a sequence s, and let $N = \Sigma_{i=1}^{n} |s_i|$ denote the sum of the length of all sequences in S. Let $s[i \ldots j]$ denote the substring starting at index i and ending at j in s. Throughout this paper, we use the terms "sequences" and "strings" interchangeably.

An *alignment* between two strings is an order-preserving way to map every character in one string to either a character in the other string (event is called a "*substitution*"), or to a blank symbol (called a "*gap*"). A substitution is a "*match*" if the two characters are identical, and a "*mismatch*" otherwise. An alignment of a character to a gap symbol is referred to as an "indel" (for insertion/deletion). In an alignment model, the user assigns a score for each of these events — rewarding matches with a positive score and penalizing mismatches and indels with a negative score. As a general practice, integer scores are used. Also, while aligning protein sequences, the scores are typically derived from a pre-defined table called the "substitution matrix", which is of size $(|\Sigma|+1) \times (|\Sigma|+1)$ [33]. The problem of computing an *optimal alignment* between two strings s_1 and s_2 becomes one of identifying an alignment that maximizes the alignment score. This formulation is called the *global alignment* problem because the alignment is defined to cover the entire strings. A generalized version of this problem is called *local alignment*, in which the problem is one of finding a best matching pair of substrings that yield the maximum score when globally aligned against one another. Local alignments are useful when only parts of the sequences are expected to match, which is generally the case while aligning two protein sequences.

Computing optimal alignments: Given strings s_1 and s_2 of lengths n_1 and n_2, respectively, both global [9] and local [8] optimal alignments can be computed using dynamic programming in $O(n_1 \times n_2)$ time and $O(n_1 + n_2)$ space [34]. For simplicity of exposition, let us consider the global alignment problem. The Needleman-Wunch ("NW") dynamic programming algorithm [9] uses a recurrence $T(i, j)$ which is the optimal score for aligning the prefixes $s_1[1 \ldots i]$ and $s_2[1 \ldots j]$. It follows that $T(n_1, n_2)$ is the final optimal score. And it can be shown that $T(i, j)$ only depends on $T(i-1, j)$, $T(i, j-1)$ and $T(i-1, j-1)$. Therefore, the algorithm initializes and computes a table $T[0 \ldots n_1][0 \ldots n_2]$ in a row-major (or column-major) fashion, spending constant time at

each cell (i, j). This phase is called the forward phase. Once this phase is completed, a retrace from the cell (n_1, n_2) is performed to yield the optimal path. Although the recurrence is slightly different, the algorithm's overall structure is identical in the Smith-Waterman algorithm (abbreviated as "SW") for local alignment. It should be evident that the forward phase of this algorithm can be implemented in $O(n_1 \times n_2)$ time and $O(n_1 + n_2)$ space (by storing only the last two rows). It turns out that the retrace procedure can also be implemented to run within the same bounds [34].

Evaluating alignments: Homology refers to an expected degree of similarity between two sequences. While there is no one definition for homology, common practice is to infer homology through the evaluation of an alignment result. Following this practice, we define homology as follows. Consider two sequences s_1 and s_2 of lengths n_1 and n_2, respectively, where $n_1 \leq n_2$ without loss of generality.

Definition 1. *Two sequences s_1 and s_2 are said to be* homologous *if they share a local alignment whose score is at least τ_1% of the ideal score (with n_1 matches), and the alignment covers at least τ_2% of n_2 characters.*

The parameters τ_1 and τ_2 are user-specified, with defaults for protein sequences set as $\tau_1 = 40\%$ and $\tau_2 = 80\%$ [7]. Note, however, that neither the choice of these parameters nor the ensuing outcome of an alignment evaluation have any bearing on the time or space consumed in computing that alignment. It is also to be noted that any alignment (not necessarily the optimal) can be used for detecting the presence of homology between two sequences. However, using a sub-optimal alignment result runs the risk of failing the homology criteria (even with the corresponding optimal result passing the criteria). Therefore, it is important to use optimal alignment computation to ensure sensitivity is not lost while detecting homology [7], [12], [13].

The Homology Detection Problem: Given S, and parameters $\{\tau_1, \tau_2\}$, determine all pairs of sequences that are homologous.

Henceforth, we use the term "pair" in this paper to refer to an arbitrary pair of sequences (s_i, s_j).

Pair generation: Recall that our goal for this paper is to evaluate whether *optimal* pairwise sequence alignments can be carried out at scale for large real world data. As noted in Section II, there are some sophisticated ways, using string indices such as suffix trees/arrays, to identify a subset of pairs to undergo optimal alignment evaluation (instead of a brute-force evaluation of all $\binom{n}{2}$ pairs). While implementing such filters is important from an application-scaling perspective, for the purpose of this paper, we decided to implement a light-weight "simulator" of such a filter, primarily to allow us to stay focused on PSAPS performance[2]. The findings of this study should still extend to cases when a real filter is applied because filters only reduce the number of pairs from the quadratic search space, and one can always find larger inputs that could generate filtered workload of comparable size.

We design our pair generation module to mimic the pair selectivity of a suffix tree-based index [7]. The use of such a filter reduces the number of alignments from the theoretical maximum by a factor of at most 0.1%. Therefore, we designed a pair enumerator that selects (at random) 0.1% of the $\binom{n}{2}$ pairs.

IV. Parallelization Using Work Stealing

In this section, we describe the execution environment and our implementation of homology detection in parallel.

We analyzed the load imbalance for an all-against-all alignment of a small set of 15,000 sequences from the CAMERA database. We observed that a significant fraction of tasks are of the order of milliseconds or lower, with a large number of sub-millisecond tasks. The large number of tasks together with the wide disparity in the task processing times exacerbates problems associated with static load balancers due to small errors in estimation of alignment times. The alignments include a few large tasks taking few tenths to over one second.

Despite their counts, the smallest alignment operations consume a negligible fraction of the total processing time. On the other hand, alignment operations that can be processed in a 1ms to 100ms consume almost 90% of the total processing time. This shows that the alignment operations critical to a load balanced execution vary by up to two orders of magnitude in their processing time.

A computation dominated by a few large operations can employ static scheduling focused on such large tasks. Alternatively, a computation consisting of a large number of homogeneous tasks can be effectively load balanced by equally distributed the load across the processors. In the case of homology detection, exemplified by the characteristics discussed above, neither approach is effective. Static load partitioning will need to first find the tasks that meaningfully contribute to the execution time, accurately estimate the alignment times and then partition the large number of tasks identified.

A. Data distribution and placement

A significant challenge in the design of parallel homology detection is the management of the sequence data. We studied the CAMERA data set. The largest protein data set within the CAMERA [35] portal is approximately 5.6GB in size, representing 43M sequences. While seeming small, matching two such data sets requires as much as 9.2×10^{14} number of alignments. Given the non-linear nature of the computation, this is both compute intensive and data intensive.

In order to maximally utilize the available memory to minimize data costs, we employ the multi-threaded MPI model with one MPI process per shared memory node and one thread per processor core. The threads share the data available in SMP node, reducing communication. When the data set fits within the total memory available in the shared memory node, it is replicated on each node. When the data does not fit into one SMP node, the data is distributed among the processes, with

[2]The complexities of implementing a scalable filter is a different problem that warrants a separate study.

one-sided communication to obtained the necessary data from a remote process. We do not evaluate this model in this paper.

The data is read from an input file and distributed using MPI-IO. Once the data sets are initialized they are read-only, allowing sharing of sequence data among the threads without conflicts.

B. Enumeration and Processing of Alignment Operations

The candidate pairs of sequences are chosen using a combinatorial number system of degree 2. The number system produces lexicographically sorted binomial coefficients of degree 2, associating each combination with a non-negative number. In addition to providing a lexicographic ordering of all $\binom{n}{2}$ pairs, this allows one to compute the place within the lexicographic ordering of a given 2-combination without the need to explicitly compute the previous 2-combinations. [36] We use this property to map a contiguous sequence from a non-negative index to a given worker.

One of the drawbacks to using the original pGraph implementation was that sequences from the input set were distributed among the nodes and periodically needed to be communicated to the pair consumers (the processes performing the pair alignment.) This strategy is also known as *database partitioning*. Our implementation takes advantage of the multi-threaded MPI model and stores the input data set on each MPI rank, sharing the input data set among the worker threads. The largest data set we tested contains 20M sequences and can be stored in memory within 3GB, however, the largest protein data set within the CAMERA [35] portal is approximately 5.6GB in size. We do not consider the technique of database partitioning in this paper.

The Smith-Waterman (SW) alignment implementation requires memory proportional to the size of the largest sequence L. Specifically, it requires $40 * L$ bytes. The largest sequence we encountered for our data set was 32,794 characters resulting in a storage size of approximately 1.44MB for each thread of execution.

C. Parallel Processing

Our focus on execution at scale and short turnaround times, requires that we maximally exploit the available parallelism. In accordance with this principle, we expose each individual alignment operation as a *task*. A task is the basic unit of parallel operation that can be independently migrated and scheduled by the load balancer. In our design, a task can be migrated to any process that holds the data required by the task. This enables efficient evaluation of the task without requiring data communication.

Each alignment task is uniquely identified by the pair of sequences to be aligned. We employ the pair generation strategy above to organize the tasks into a one-to-one correspondence with a sequence of integers that range from 0 through the total number of alignments. This enumeration is then used to equally partition the alignments to be performed among the processor cores. Note that the sequential partitioning balances the number of tasks and not their execution times.

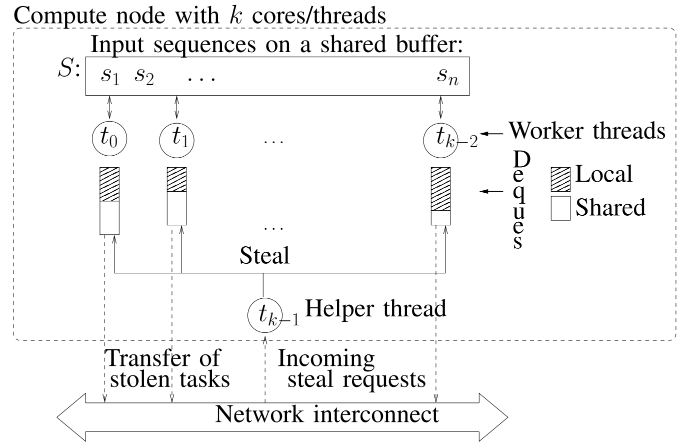

Fig. 1. Schematic of the execution on a compute node.

All processor cores collectively begin execution of the tasks that are initially seeded on them. The computation is then dynamically load balanced in response to local detection of load imbalance using distributed memory work stealing. The details of the work stealing procedure are presented in [20]. In this paper, we summarize the specifics of the design employed for the parallelization of the homology detection implementation.

The schematic illustration of the execution on each compute node is shown in Figure 1. Each processor core maintains the tasks seeded to it in a local *deque*. When local tasks finish execution, the processor core turns into a thief attempting to steal from a victim core. The stolen tasks are populated into the thief's local deque, and the thief goes back to executing tasks in its local deque. This procedure is repeated until termination is detected.

The steal operations are performed by the thief using active messages implemented on MPI. In executing a steal operation, the thief sends a message on a communicator and tag reserved for incoming active messages. An incoming message on such a communicator context and tag is interpreted as an active message, with the corresponding function invoked with the incoming data. The result of the function's execution is sent back to the source of the incoming message. Thus the active messages are implemented in terms of non-blocking MPI two-sided communication operations.

In order to ensure quick distribution of work to the thieves, we dedicated one thread to handle incoming active message requests initiated by the thieves. In the absence of such a thread, achieving good performance required regular polling for incoming requests as part of the executing application. This poses a challenge in identifying the appropriate polling granularity. Polling too often impacted application execution times due to the increased overhead. Polling too infrequently resulted in slow work redistribution and starvation of thieves. More perversely, our experience indicated that the correct polling frequency depending on the platform and the scale for a particular execution — a prohibitive challenge when one

attempts a performance portable implementation. While dedicating a thread consumes processing resources, it alleviated these challenges resulting in repeatable and consistently good execution across platforms and processor core counts.

We employ one process per SMP node, with one thread per core. One of the threads is dedicated to processing incoming active messages. A steal message targeted at a victim core is sent to the helper thread associated with the victim's process. The steal operation executed by the helper thread involves locking the victim's deque and stealing the requested number of tasks. If no tasks are available to be stolen, the thief attempts to steal from another victim.

The deque is organized as a split queue with a local and a shared portion. The local operation enables quick processing of tasks by the core owning the deque without the need for locks. The helper thread lock the *shared* portion of the victim's deque during a steal. Each core periodically releases work from the local to the shared portion, when the share portion is observed to have run out of work. When the local portion is empty, tasks are moved from the shared portion to the local portion. While releasing work to the shared portion can be performed without a lock, acquiring work requires the shared portion of the deque to be locked. The deque consisting of the local and shared portions is implemented on a bounded circular buffer. This makes acquiring and releasing work into simple arithmetic operations. This also enables steal operations to copy the stolen tasks using efficient contiguous MPI communication calls.

A thief steals half of the work in the shared portion of a victim's deque. When only a few cores have remaining work, this strategy leaves both the victim and the thief with work available for other thieves enabling quick distribution of work among idle cores. A thief selects a victim at random. Stealing half the work coupled with random stealing approximates broadcasting of work, when only a few core have work.

Termination detection is performed using the four-counter method [37]. The algorithm proceeds as waves moving up and down a binary tree of processor cores counting the number of created and processed tasks. As proven by Francez [37], termination is detected when these numbers are observed to be equal in two successive waves. Thus termination is detected two waves after all tasks have been processed. We observed in our evaluation that only a small number of waves, often less than five including termination detection, are executed in an execution lasting several minutes. This is due to the fact that a core propagates the termination detection wave only if it has completed processing all of its local tasks.

The output of the application consists of the pairs which passed the alignment criteria, as well as for each successful pair the alignment optimal score, self-score ratio, and percent identity. These values are buffered on each thread of computation until termination is detected, at which point the primary thread on each SMP node writes these values to disk, one file per SMP node. I/O was an insignificant fraction of the overall computation and is not discussed further.

Sophisticated static partitioning schemes that take alignment time estimates into account incur additional cost. We considered less intrusive approaches to deriving a good initial load balance. We considered randomization of the sequence, round-robin and block-cyclic mapping of sequence pairs to integers. These can be effected with minimal space or time overhead. Empirical evaluation showed that these optimizations did not materially impact the observed performance. We do not discuss them further.

V. RESULTS

Our parallel homology detection framework was tested using an arbitrary collection of 2560K amino acid sequences representing an ocean metagenomic data set available at the CAMERA metagenomics data archive [35]. The sum of the length of the sequences in this set is 390,345,021, and the mean$\pm\sigma$ is 152.48 ± 167.25; the smallest sequence has 1 amino acid residue and longest 32794 amino acid residues. Smaller size subsets containing 1280K and 1920K were derived from the 2560K sequence set and used for scalability tests. We use protein/putative open reading frame inputs from metagenomic data sets to capture a more challenging use-case where the skewed distribution in sequence length can cause nonuniformity in the PSA tasks.

Experiments were performed on the *Hopper* supercomputer at the National Energy Research Scientific Computing Center (NERSC). Hopper is a 1.28 petaflop/sec Cray XE6 consisting of 6,384 compute nodes made up of 2 twelve-core AMD 'MagnyCours' 2.1 GHz processors and 32GB RAM per node. Hopper's compute nodes are connected by the Cray Gemini Network which is a custom high-bandwidth (8.3GB/s), low-latency ($< 1\mu s$) network with a network topology of a 3D torus. We compiled our application using the the Intel® C++ 64 Compiler XE, version 12.1.2.273 using the flags -O3 and -pthread. The MPI library is a custom version of mpich2 for Cray XE systems, version 5.4.4.

Our solution was tested using sequence data sets ranging from 320K through 2.5M sequences. The smaller data sets were derived from the largest data set by taking the first n sequences from the largest data set. The number of alignments was reduced from the theoretical $\binom{n}{2}$ maximum by a factor of 0.1% based on the discussion in Section III. For instance, 819,199,360, 1,843,199,040, and 3,276,798,720 pairs were aligned for the 1.2M, 1.9M, and 2.5M sequence datasets, respectively.

The sizes were chosen to stress the parallel framework in the strong scaling setting, with anticipated execution time under a few minutes at the maximum scale considered. For a given degree of parallelism, larger problem sizes and the associated execution times simplify the job of the load balancer, as evidenced by the evaluation in this section.

The execution times on Hopper are shown in Figure 2 and tabulated in Table I. For each problem size evaluated, in addition to observed performance, ideal anticipated performance based on perfect scaling based on the execution on the smallest core count for that problem size is also shown. The 2560K sequence set took 134 seconds on 120000 Hopper cores. These results correspond to considerably shorter

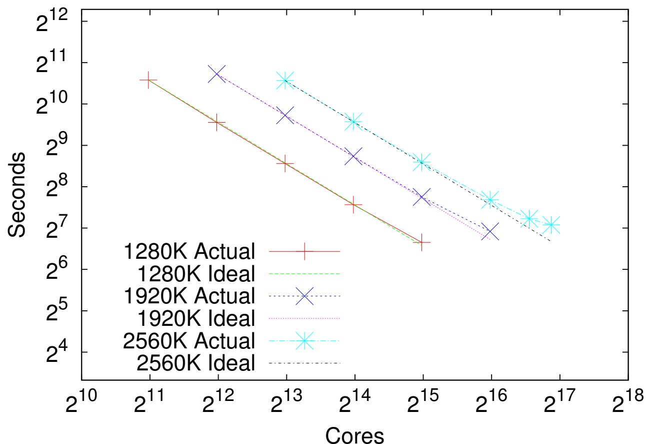

Fig. 2. Execution times on Hopper for varying input sizes ($n = 1280K, 1920K$, and $2560K$).

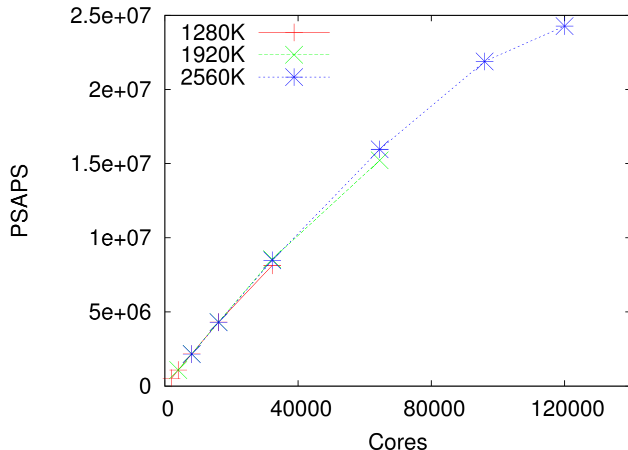

Fig. 3. PSAPS (Pairwise Sequence Alignments Per Sec) performance on Hopper systems on various input sizes ($n = 1280K, 1920K$, and $2560K$).

	Number of Hopper cores (p)						
(n)	2016	4032	8064	16128	32256	64512	120000
1280K	1529	752	377	189	100	-	-
1920K	-	1692	846	426	215	121	-
2560K	-	-	1518	760	386	205	134

TABLE I
TABLE SHOWING THE PARALLEL RUNTIMES (IN SEC) AS A FUNCTION OF THE INPUT SIZE AND CORE COUNT, ON HOPPER. AN ENTRY "-" MEANS THAT THE CORRESPONDING RUN WAS NOT PERFORMED.

turnaround times and large system scaling than considering in prior work [7]. This is mainly attributable to the work stealing approach because nothing changed with respect to the alignment implementation.

We calculated efficiencies for the experiments by assuming perfect scaling starting from the smallest core count employed for a particular sequence set. We observe a consistently high efficiency across problem sizes and core counts, when ignoring the loss of a core. For each problem size, efficiency drops the most when the execution wall clock time reduces to less than a couple of minutes. On Hopper at 120,000 cores, we observe 75% efficiency for 2560K sequences. We observe that for a given core count, efficiencies significantly improve with increased problem size. In particular, we observe efficiencies above 90% whenever the execution time is at least 500 seconds.

We observe that the dedicated core has negligible impact on Hopper. We expect the cost to be further reduced on future systems with larger number of cores. In addition, the helper thread can be multiplexed with, say, the operating system thread or other runtime threads. In particular, on BlueGene/Q, the helper thread could be multiplexed on one of 16 cores, which support four threads each, or on the seventeenth core dedicated for the operating system and other supporting activities.

The PSAPS achieved on Hopper are shown in Figure 3. Our implementation is able to achieve sustained PSAPS performance across all input sizes tested. The highest PSAPS performance obtained was 2.42×10^7 on 120000 Hopper cores for the 2560K input. This is two orders of magnitude greater than the previous reported top PSAPS performance (6.59×10^5 on 2K cores [7]) for this problem on the same input.

VI. Discussion and Conclusions

All-against-all sequence homology detection is pivotal to gaining biological understanding of the interplay among biomolecular sequences. Although essential to guarantee quality, computation of optimal alignments toward homology detection has never been used in any large-scale application till date owing to long runtimes and a lack of scalable tools. In this paper, we demonstrate that it is possible to scale billions of optimal alignment computations on more than 100,000 cores. Through the use of distributed-memory work stealing, we were able to demonstrate up to 24.2 million PSAPS and 75% efficiency on 120,000 cores[3] on the Hopper Cray XE6 supercomputer for a calculation running under 2 minutes. We demonstrate efficiencies above 90% for calculations running for at least 200 seconds.

Several research directions have been planned for the near future. To further improve PSAPS performance, a combination of techniques needs to be pursued. Fine-grain parallel implementations exist to accelerate Smith-Waterman alignment computation for vector instruction sets and GPUs. As future work, we intend to integrate these optimized kernels to further enhance the achieved PSAPS rate. This could be essential to effectively utilize next generation supercomputing platforms.

Recently, work stealing has been adapted to exploit the property of persistence – repetition of execution characteristics across calculation steps – to achieve strong scaling for calculations lasting tens of seconds. We intend to extend that idea to homology detection, by incrementally improving work stealing "schedules" by partitioning an all-against-all alignment computation into a few initial batches of alignment operations, followed by the rest of the calculation. Rather than dedicate a core, which imposes a severe performance penalty on systems with few processor cores per node, multi-threaded

[3]relative to an execution on 8064 cores

architectures allow the helper thread to handle incoming active messages to be multiplexed with operating system threads or an application worker thread.

Processing large sequence databases necessarily requires distributing the data across multiple shared memory nodes. This problem is exacerbated by multiple disjoint memory domains in accelerator-based systems. We intend to study work stealing among sub-groups of processes that duplicate a subset of the data, for load balancing homology detection with partitioned data without incurring data movement costs.

Finally, we plan to implement and integrate a scalable and yet effective filtering (using string indexes) for rapid identification of promising pairs. Dynamic load balancing is required for implementing this filtering step as well, as the underlying data structures tend to be irregular and need to work in tandem with alignment computation. A unified solution using work-stealing for both pair generation and alignment could be an interesting exploration.

Acknowledgments

This work was supported in part by the DOE Office of Science, Advanced Scientific Computing Research, under award numbers 59193 and DE-SC-0006516; U.S. Department of Energy's Pacific Northwest National Laboratory under the Extreme Scale Computing Initiative; and NSF grant IIS 0916463. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.

References

[1] "Illumina sequencing," http://www.illumina.com/systems.ilmn.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, Oct. 1990.

[3] A. Kalyanaraman, S. Emrich, P. Schnable, and S. Aluru, "Assembling genomes on large-scale parallel computers," *Journal of Parallel and Distributed Computing*, vol. 67, no. 12, pp. 1240–1255, Dec. 2007.

[4] C. Wu and A. Kalyanaraman, "An efficient parallel approach for identifying protein families in large-scale metagenomic data sets," in *SC'08*, 2008, p. 35:135:10.

[5] A. Kalyanaraman, "Efficient clustering of large EST data sets on parallel computers," *Nucleic Acids Research*, vol. 31, no. 11, pp. 2963–2974, Jun. 2003.

[6] S. Yooseph *et al.*, "The sorcerer II global ocean sampling expedition: Expanding the universe of protein families," *PLoS Biology*, vol. 5, no. 3, p. e16, 2007.

[7] C. Wu, A. Kalyanaraman, and W. R. Cannon, "pgraph: Efficient parallel construction of large-scale protein sequence homology graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, no. PrePrints, 2012.

[8] T. Smith, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, Mar. 1981.

[9] S. Needleman and C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, Mar. 1970.

[10] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison," *Proceedings of the National Academy of Sciences*, vol. 85, no. 8, p. 2444, 1988.

[11] R. C. Edgar, "Search and clustering orders of magnitude faster than BLAST," *Bioinformatics*, vol. 26, no. 19, pp. 2460–2461, Aug. 2010.

[12] 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, Nov. 1991.

[13] E. G. Shpaer, M. Robinson, D. Yee, J. D. Candlin, R. Mines, and T. Hunkapiller, "Sensitivity and selectivity in protein similarity searches: A comparison of SmithWaterman in hardware to BLAST and FASTA," *Genomics*, vol. 38, no. 2, pp. 179–191, Dec. 1996.

[14] Committee on Metagenomics: Challenges and Functional Applications, the National Research Council, *The New Science of Metagenomics: Revealing the Secrets of Our Microbial Planet*. The National Academies Press, 2007.

[15] P. Weiner, "Linear pattern matching algorithms." IEEE, Oct. 1973, pp. 1–11.

[16] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," in *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, Philadelphia, PA, USA, 1990, pp. 319–327.

[17] C. Oehmen and J. Nieplocha, "ScalaBLAST: a scalable implementation of BLAST for High-Performance Data-Intensive bioinformatics analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 8, pp. 740–749, Aug. 2006.

[18] A. Darling, L. Carey, and W. Feng, "The design, implementation, and evaluation of mpiBLAST," *Proceedings of ClusterWorld*, vol. 2003, 2003.

[19] H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma, and W. chun Feng, "Massively parallel genomic sequence search on the blue gene/p architecture," in *SC'08*, Nov. 2008, pp. 1–11.

[20] J. Lifflander, S. Krishnamoorthy, and L. Kale, "Work stealing and persistence-based load balancers for iterative overdecomposed applications," in *High-Performance Parallel and Distributed Computing (HPDC)*. ACM, 2012.

[21] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *SC'09*, 2009, pp. 53:1–53:11.

[22] V. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, 2011, pp. 201–212.

[23] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C. Tseng, "UTS: an unbalanced tree search benchmark," *Languages and Compilers for Parallel Computing*, pp. 235–250, 2007.

[24] T. Yanai, G. Fann, Z. Gan, R. Harrison, and G. Beylkin, "Multiresolution quantum chemistry in multiwavelet bases: Analytic derivatives for hartree–fock and density functional theory," *The Journal of Chemical Physics*, vol. 121, p. 2866, 2004.

[25] R. J. Harrison *et al.*, "Toward high-performance computational chemistry: Ii. a scalable self-consistent field program," *Journal of Computational Chemistry*, vol. 17, no. 1, pp. 124–132, 1996.

[26] R. J. Bartlett and M. Musiał, "Coupled-cluster theory in quantum chemistry," *Rev. Mod. Phys.*, vol. 79, pp. 291–352, Feb 2007.

[27] M. Farrar, "Striped smith–waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[28] Y. Liu, W. Huang, J. Johnson, and S. Vaidya, "GPU accelerated smith-waterman," in *Computational Science ICCS 2006*, ser. Lecture Notes in Computer Science, 2006, vol. 3994, pp. 188–195.

[29] S. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for smith-waterman sequence alignment," *BMC bioinformatics*, vol. 9, no. Suppl 2, p. S10, 2008.

[30] I. Li, W. Shum, and K. Truong, "160-fold acceleration of the smith-waterman algorithm using a field programmable gate array (FPGA)," *BMC bioinformatics*, vol. 8, no. 1, p. 185, 2007.

[31] V. Sachdeva, M. Kistler, E. Speight, and T. Tzeng, "Exploring the viability of the cell broadband engine for bioinformatics applications," *Parallel Computing*, vol. 34, no. 11, pp. 616–626, 2008.

[32] S. Sarkar, T. Majumder, A. Kalyanaraman, and P. P. Pande, "Hardware accelerators for biocomputing: a survey." IEEE, May 2010, pp. 3789–3792.

[33] S. F. Altschul, *Substitution Matrices*. John Wiley & Sons, Ltd, 2001.

[34] E. W. Myers and W. Miller, "Optimal alignments in linear space," *Computer Applications in the Biosciences : CABIOS*, vol. 4, no. 1, pp. 11–17, 1988.

[35] S. Sun *et al.*, "Community cyberinfrastructure for advanced microbial ecology research and analysis: the CAMERA resource," *Nucleic Acids Research*, vol. 39, no. Database, pp. D546–D551, Nov 2010.

[36] D. E. Knuth, "Generating all combinations and partitions," in *The Art of Computer Programming*, ser. Fascicle 3. Addison-Wesley, 2005, vol. 4, pp. 5–6.

[37] N. Francez, "Distributed termination," *ACM Trans. Program. Lang. Syst.*, vol. 2, pp. 42–55, January 1980.