Orion: Scaling Genomic Sequence Matching with Fine-Grained Parallelization

Kanak Mahadik*, Somali Chaterji†, Bowen Zhou*, Milind Kulkarni*, Saurabh Bagchi*
*School of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana
†Department of Computer Science, Purdue University, West Lafayette, Indiana
Email: {kmahadik,schaterji,bzhou,milind,sbagchi}@purdue.edu

***Abstract*—Gene sequencing instruments are producing huge volumes of data, straining the capabilities of current database searching algorithms and hindering efforts of researchers analyzing large collections of data to obtain greater insights. In the space of parallel genomic sequence search, most of the popular software packages, like mpiBLAST, use the database segmentation approach, wherein the entire database is sharded and searched on different nodes. However this approach does not scale well with the increasing length of individual query sequences as well as the rapid growth in size of sequence databases. In this paper, we propose a fine-grained parallelism technique, called Orion, that divides the input query into an adaptive number of fragments and shards the database. Our technique achieves higher parallelism (and hence speedup) and load balancing than database sharding alone, while maintaining 100% accuracy. We show that it is 12.3X faster than mpiBLAST for solving a relevant comparative genomics problem.**

I. Introduction

One of the foundational building blocks of computational biology is *sequence alignment*, looking for similarities between particular DNA, RNA or protein sequences and a database of other sequences. Finding regions of similarity between target sequences and databases helps biologists understand structural, functional and evolutionary relationships between sequences to predict biological function of genes, find evolutionary distance between sequences and do genome assembly by finding common regions and repeats within a genome. For example, finding large overlaps between the DNA sequences of a newly discovered biological specimen (the *query*) and the DNA sequences of known organisms (the *database*) can highlight evolutionary relationships between the organisms.

The classic algorithm for performing sequence alignment, identifying matches between a query and a database of sequences, is the *Basic Local Alignment Search Tool* (BLAST) [1], [2]. BLAST operates by comparing each of the sequences in the input query set against each of the sequences in a database to identify *alignments* that partially or completely overlap. The more similarity there is, the higher the alignment's score. *E-value* ia numerical value that captures the likelihood that the similarity is statistically significant. Alignments with E-value below a certain threshold are output as potential matches by the algorithm. Section II describes the algorithm in more detail.

The National Center of Biotechnology Information (NCBI) provides public databases of gene sequences that researchers can search using BLAST.[1]. Unfortunately, the explosive growth in the number of biological sequences poses a formidable challenge to the current database searching algorithms. In December 2013, the GenBank database—hosted by NCBI—had about 170 million sequences, and the number of bases has doubled approximately every 18 months [3], [19]. Given the exponential growth in the size of sequence databases, and the requirement to query longer sequences, current database searching algorithms struggle to provide the alignment and search results in a timely manner. Early parallel BLAST implementations [5], [7] exploited coarse-grained parallelism: individual queries can be processed simultaneously against the same database. However, while such parallelism improves throughput, it does not help an individual researcher with a single query: For example, a BLAST job with a query sequence of 100,000 contiguous fragments (i.e., contigs or overlapping sequenced data reads) BLASTed against the non-redundant (NR) nucleotide database could take 70 days [29]! To provide genomics researchers with reasonable latency for their searches, exploiting additional parallelism has become a necessity.

The most popular open source parallelization of BLAST is mpiBLAST, using, unsurprisingly, MPI to run BLAST in parallel on clusters [8]. mpiBLAST adopts a natural parallelization strategy. Because BLAST compares the input query against each sequence in the database separately, parallelism can be exploited by performing multiple such comparisons concurrently. mpiBLAST thus *shards* the database into multiple pieces each containing a subset of the databases's sequences and distributes the shards across the computational nodes in the cluster. These shards can then be searched independently and simultaneously for alignments with the input query.

Unfortunately, while mpiBLAST can exploit parallelism by sharding large databases, and even by processing multiple input queries in parallel, it has significant limitations for many biological use cases. In *long sequence alignment*, a long input query is matched against a database. Such use cases are becoming increasingly common. With the rapid expansion of next generation sequencing technologies, the number of organisms whose entire genomes are being sequenced has been growing at a rapid pace. Once a genome is sequenced, it is annotated, which involves (among other processes) comparing the newly-sequenced genome, or parts thereof, with that of a closely-related organism or with the expansive NT database, to establish the evolutionary relations of this newly-sequenced

[1] http://blast.ncbi.nlm.nih.gov/

978-1-4799-5500-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SC.2014.42

organism. This results in large queries, with the upper bound being the size of the entire genome, which can be millions of nucleotides.

In this scenario, mpiBLAST runs out of parallelization opportunities. There is but one input sequence, so parallelism by processing multiple queries simultaneously is impossible. And increasing the number of database shards to increase parallelism suffers from diminishing returns: even if the database contains enough sequences to profitably create additional shards, additional shards increase scheduling overhead as well as the time required to aggregate the output from each query-shard work unit.

Moreover, mpiBLAST's parallelization strategy can lead to severe load imbalance with large queries, or with queries of very different sizes. If a query sequence is long, or has many matches with a particular database sequence, it will take a long time to process, while a short query sequence, or one with little similarity to a database sequence can be completed much faster. As a result, the execution time of different query-shard work units can vary significantly, a problem that is only exacerbated as queries get longer [25], [12]. Further, it is difficult to predict what the running time for a unit of work will be from simple metrics as the length of the query [12]. Consequently, the static load balancing approach of mpiBLAST tends to create severe load imbalances among the different nodes processing different work units, as we experimentally show in our evaluation.

To address these concerns, we propose *Orion*, a new parallel BLAST implementation that exploits finer-grained parallelism than mpiBLAST, achieving both more parallelism in the face of long sequences as well as better load balance. The key insight behind Orion is that a single, long query sequence need not be matched against a database sequence serially; instead, the query can be *fragmented* into sub-queries (which we call "query fragments"), each of which can be matched against the database independently and in parallel. Figure 1 captures the various levels of parallelism inherent in sequence alignment. The early approaches to sequence alignment primarily targeted the lowest level, processing multiple queries in parallel against the entire database, while mpiBLAST exploits the two lowest levels, processing the same query against different database shards simultaneously. Orion exploits all levels of parallelism: inter-query, intra-database, and *intra-query*.

Query fragmentation itself is not a new strategy in the BLAST community. It first arose in recent work that noted that BLAST's performance is severely degraded by cache misses as query size grows, and proposed query fragmentation as a solution [6], [11]. Such strategies either require access to the entire query to compute alignments [6], or require that the query fragments overlap by a substantial amount to avoid missing alignments [11], obviating parallelism or necessitating substantial extra work.

In contrast, in Orion, we limit the size of the overlap by querying the input parameters such as the thresholds in the BLAST algorithm and the penalties due to a mismatch in BLAST, and employ a novel extension and aggregation strategy to avoid missing alignments. Our fragmenting strategy is such that practically there is no loss in accuracy, i.e., *every* sequence that will be matched successfully in BLAST will also be matched successfully in Orion. However, the overlaps are not so large as to eliminate the scope for intra-query parallelism.

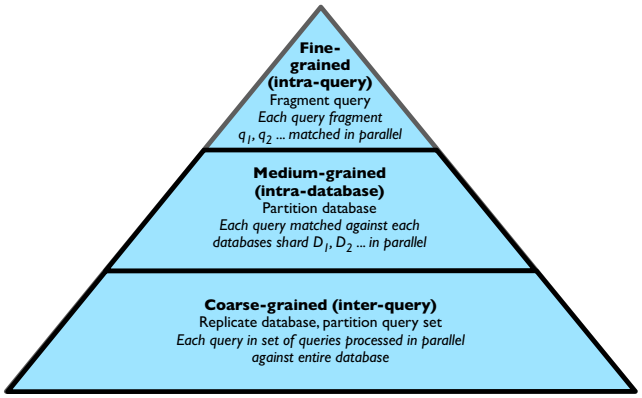

Fig. 1: Parallelism in genomic sequence search. Our solution Orion is the first to exploit opportunity for parallelism at all three levels. mpiBLAST, for example, only uses the lower two levels.

We introduce three chief novelties:

1) We develop an analytical model based on BLAST's scoring formula that identifies the optimal fragmentation strategy, avoiding redundant work.
2) We introduce a speculative extension strategy that allows alignments that may cross query fragment boundaries to be identified.
3) We build an aggregation algorithm that combines full and partial alignments from each fragment to generate a final set of alignments that matches the original sequential algorithm.

We parallelize and implement our algorithm using the Hadoop MapReduce framework, and demonstrate that our algorithm yields better parallelization, performance and load balance than mpiBLAST, while producing the same results.

Outline Section II describes the basic BLAST algorithm, as well as mpiBLAST's parallelization strategy. Section III details the design of Orion's fragmentation and aggregation algortihms. Section IV discusses the Hadoop implementation of Orion. Section V compares Orion to both sequential BLAST and mpiBLAST. Section VI surveys related work, while Section VII concludes.

II. Background

This section provides background on the general concepts of sequence alignment; BLAST, the most popular algorithm for performing sequence alignment; and mpiBLAST, the most common parallel implementation of BLAST.

A. Sequence alignment

Sequence search typically examines one or more *query sequences*, Q, against a *database* of reference sequences, D. The sequences might be nucleotide sequences (*e.g.*, genomes of organisms) or peptide sequences (the chains of amino

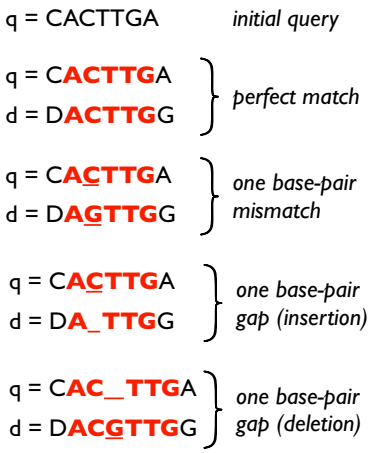

Fig. 2: Query sequence and possible matching database sequences. Matching alignments are shown in bold, red text. Mismatches and gaps (both inserted and deleted bases) are underlined. In the second alignment, a possible match is found by positing that a nucleotide was altered in the database sequence to produce the query sequence. In the third alignment, a possible match is found by positing that a nucleotide was *inserted* into the database sequence to produce the query sequence, while in the fourth alignment, a possible match is found by positing that a nucleotide was *removed* from the database sequence.

acids that make up a protein). We will focus on nucleotide sequences for the remainder of the paper. Each query sequence $q \in Q$ is compared against each database sequence $d \in D$ to determine their similarity. Similarity is determined by looking for long subsequences that are common to both d and q. High similarity between nucleotide sequences indicate that the same gene might exist in both sequences, or that both sequences have similar biological function. Similarly, regions of little similarity between sequences might indicate that such regions do not have any biological importance (they are "junk DNA").

A nucleotide sequence is represented by a string of bases drawn from $\{A, C, G, T\}$, so finding common sequences between two such strings seems like it can be solved using traditional string-matching algorithms. However, because genomes are constantly mutating, it is often useful to look not for *exact* matches, but merely *good* matches between sequences. Common alterations to genomic sequences include changes of a single base, leading to a *mismatch* between sequences, and insertion or deletion of a single base, leading to a *gap* between sequences. Hence, alignment must consider several scenarios when looking for a good match. Consider the query sequence $q = CACTTGA$ shown in Figure 2. There are several possible database sequences that could "match" q, once mismatches and insertions and deletions of bases from the query are taken into account.

Each of the database sequences in Figure 2 represent a possible alignment; the only difference is in the "score" given to the alignment: fewer mismatches or gaps produce a higher score. Nevertheless, having a mismatch or gap does not disqualify a particular match: a long alignment with one or two mismatches can produce a higher score than a short alignment with no mismatches. The classic dynamic programming algorithm for computing alignments with gaps and mismatches is Smith-Waterman [27].

B. BLAST

The basic Smith-Waterman algorithm suffices to find alignments, but it is slow ($O(mn)$ time to find alignments between sequences of length m and n) and has high space overhead ($O(mn)$ space to store the scores in the dynamic programming matrix). Altschul *et al.* designed the Basic Local Alignment Search Tool (BLAST) to perform faster alignments, at the cost of accuracy (potentially missing some alignments) [1]. While the details of BLAST are quite complex, here we provide a high level intuition of BLAST's operation. We describe BLAST in terms of a single query q and database sequence d, though the algorithm ultimately operates on sets of both.

BLAST has three phases: (i) the *k-mer match* phase; (ii) the *ungapped alignment* phase; (iii) the *gapped alignment* phase. In all three phases, BLAST relies on a scoring function that provides a numerical score for the current proposed alignment. In the first phase, BLAST considers every k-length subsequence (called *k-mers*) of q and d and looks for k-mers that appear in both.[2] This step is performed efficiently by creating a lookup table with all k-letter words in q. The algorithm then walks through d and uses the lookup table to see if a k-length subsequence of d matches any part of q. These matches are *seeds* of potential alignments.[3]

In the second phase, ungapped alignment, each seed is *extended* both to the left and right allowing both perfect matches (corresponding nucleotides in q and d) and mismatches (different nucleotides in q and d). While perfect matches increase the score of the potential alignment, mismatches decrease the score. BLAST tracks the current score of the alignment, s, and the maximum score seen so far for the current seed, s_{max}. If $s_{max} - s$ is greater than some threshold t_x (called the *X-drop* threshold), the second phase terminates, returning the alignment with the peak score for the current seed. If the returned alignment's score s is greater than some threshold t_u (which we call the *ungapped threshold*), the alignment is passed to phase three. As an optimization, if a seed is contained within a previously-found alignment, the seed can be skipped.

In phase three, gapped alignment is performed. The ungapped alignment is extended in both directions, this time allowing insertions and deletions to occur as the alignment is extended. As in the second phase, the maximum score of the alignment s_{max} is tracked, and if the current score s drops below s_{max} by more than t_x, the phase is terminated and the resulting alignment is returned.

After each seed is processed, all the alignments that score above a threshold of statistical significance (called the *E-value*) are sorted and returned to the user. Numerically, the lower the E-value is the better the match is, i.e., lesser is the chance that the alignment happened purely by chance. Therefore, if the calculated E-value is *less than* the E-value threshold, is

[2]When performing nucleotide (DNA or RNA) alignment, only exact k-mer matches are identified; when performing protein alignment, partial matches can be found, with scores based on the particular peptides matched.

[3]Note that this is the phase where inaccuracy relative to Smith-Waterman is introduced, as alignments that do not have a k-mer seed will be missed.

Parameter	Description	Default value
k	Length of initial seeds	11
t_x	X-drop value	20, 15
t_u	Ungapped alignment threshold	N/A
E	Final reporting threshold	10

TABLE I: Parameters and default values for BLAST. There are two default x-drop values, the first for ungapped alignment and the second for gapped alignment. There is no default value for t_u, as the score threshold for significance is dependent on query and database sequence length.

the alignment output to the end user. Table I summarizes the parameters used in BLAST.

C. mpiBLAST

Figure 1 shows the types of parallelism that arise in BLAST. Most early attempts to parallelize BLAST exploited the coarsest granularity of parallelism: each query q in the set of queries Q is processed independently. The database D of sequences is replicated on each compute node, and queries are then processed simultaneously on each node [7], [5]. Later approaches adopt a more aggressive, finer-grained parallelization strategy: in addition to partitioning the query set Q into individual queries $Q_1, Q_2, \ldots$, the database is partitioned into subsets $D_1, D_2, \ldots$. For clarity, we will refer to partitioning the query set as *segmenting* the query set, and partitioning the database as *sharding* the database. Each pair (Q_i, D_j) represents a work unit, applying one query segment against one database shard. The work units can be processed in parallel, with the results from each query aggregated later. Perhaps the best-known example of this parallelization strategy is mpiBLAST [8].

mpiBLAST follows the master-worker paradigm. Before alignment can start, the master shards the database into disjoint partitions of approximately equal size and places them in shared storage. The master uses a greedy algorithm to assign unprocessed database shards to its workers. Query segments are then handed to each worker. A worker executes the basic BLAST algorithm for the query segment on its database shard(s) and sends the results back to the master. The master ensures that every query segment is processed against every database shard, and also aggregates the results for each query, performing the final sorting to present the queries' alignments. mpiBLAST achieves parallelism by segmenting the queries and sharding the database, and in addition improves performance relative to non-sharing implementations by choosing shard sizes so that each shard fits in a worker node's main memory.

mpiBLAST works well when Q contains many short sequences and D is large, affording it opportunities both to create sufficient parallelism and to provide load balance (by generating far more work units than worker processes). However, in many biological settings, these assumptions do not hold true. For example, it is common to match a single, large query sequence against a small database (*e.g.*, matching a long human DNA sequence against a database containing genomes for each human chromosome). In such settings, mpiBLAST cannot generate enough work to provide parallelism and load balance. Even if the database is large enough to shard, long queries lead to more variable runtime [25], [12], creating load imbalance problems. Moreover, because mpiBLAST relies on the basic BLAST algorithm at each worker, it suffers from poor performance in the face of long queries [6].

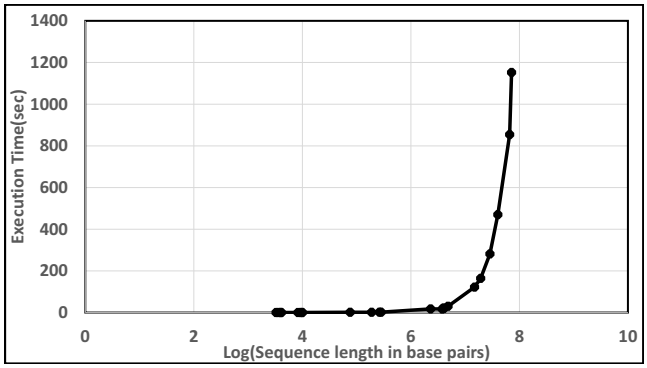

Fig. 3: mpiBLAST behaviour for long sequences

We studied the scalability of mpiBLAST, in terms of length of query sequences handled by performing experiments to search Human genes from the NCBI gene database(http://www.ncbi.nlm.nih.gov) over the *Drosophila melanogaster* database. The sequences ranged from 3000bp to 99Megabp (base pairs) in length. We used a small test cluster of 4 nodes and 64 cores to do the experiments. To enable mpiBLAST to fully exploit available parallelism, we made 64 shards of the database. Figure 3 shows that performance of mpiBLAST is good at query sequences of length less than 1 Mbp, but starts to worsen at a threshold of 1Mbp. The performance worsens rapidly beyond this threshold of 1Mbp, reaffirming the poor performance of mpiBLAST in the face of long queries as mentioned above.

In the next section, we discuss our design of a new parallel BLAST implementation that provides parallelism and load balance even for large queries.

III. Design of Orion

This section discusses the design of Orion. Implementation-specific details are discussed in Section IV. The high-level architecture of Orion is shown in Figure 4.

A. Query fragmentation

As introduced earlier, Orion uses as a fundamental strategy, the fragmentation of a query and matching the fragments in parallel. Continuing with the notation from Section II, we have a query set Q, which comprises individual queries $Q_1, Q_2, \cdots, Q_m$. The entire database is D and it is sharded into disjoint shards $D_1, D_2, \cdots, D_n$. Further, Orion fragments each query Q_i into fragments $Q_{i1}, Q_{i2}, \cdots, Q_{ik}$. Our design creates equal-sized query fragments, by determining the optimal fragment size.

A simple approach to query fragmentation is as follows: for a given query Q_i,match each query fragment against each database shard in parallel, using baseline, sequential BLAST. After all fragments of Q_i have been matched against each

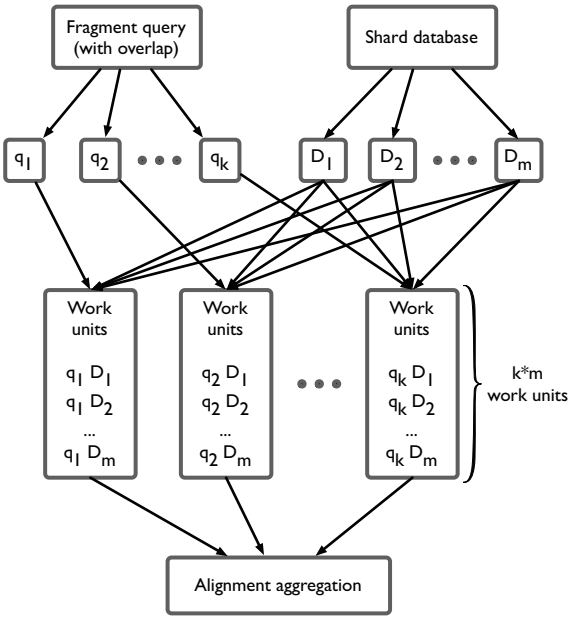

Fig. 4: High Level Architecture of Orion

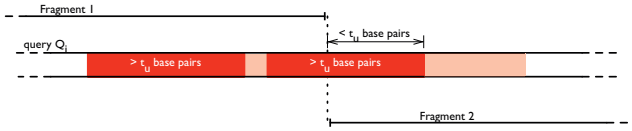

Fig. 5: Example alignment that spans two disjoint query fragments. The alignment is shaded, while the darker shaded regions represent ungapped sub-alignments that would be reported as part of Phase ii of BLAST.

database sequence, aggregate the results, combining alignments from neighboring fragments that can be concatenated to form a larger alignment, and report them. Unfortunately, this simple strategy, which assumes that query fragments are independent, is incorrect; if an alignment spans two query fragments, then the *portion* of the alignment that lies in each fragment may not have a high enough score to be reported.

Consider Figure 5. It shows an alignment that spans two query fragments with no overlap. The shaded(dark and light) portion of the query represents the alignment that should be reported, while the darker shaded portion represents ungapped sub-alignments that exceed the threshold t_u, introduced in Section II (it is the number of base pairs that produce a long enough alignment to pass the score threshold). While the search over fragment 1 will return a partial alignment, triggered by the first ungapped sub-alignment, the search over fragment 2 will not return any alignments: the portion of the final alignment that lies in fragment 2 does not have any sufficiently-long ungapped alignments to pass the threshold in phase ii of BLAST.

This situation is not a corner case, rather it is quite common in practice, with the likelihood increasing with decreasing size of each query fragment. The choice of short query fragments

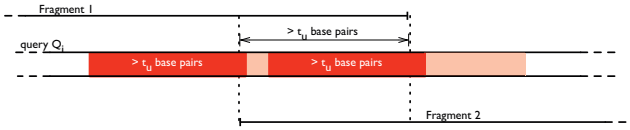

Fig. 6: Alignment with sufficient overlap

is of course appealing from the point of view of increasing the number of work units and the degree of parallelism. Note, also, that this issue applies not only to the t_u threshold, but also to the two other thresholds in BLAST: the initial k-mer threshold (if a k-mer spans two fragments, it will never be discovered) and the final E-value thresholds. In general, if the overall alignment passes a threshold, but the sub-alignments found on each fragment do not, the alignment will be missed.

Fragment overlap: To overcome the missed alignment problem described above, Orion uses a combination of *overlapping query fragments* and *alignment aggregation*. To see why overlapping fragments can be useful, consider overlapping neighboring query fragments by k nucleotides. By doing so, it is no longer possible to miss a k-mer match. Intuitively, the overlap should be large enough such that the following condition holds.

> If there is a matching sequence between the query and the database, then the partial matches within each query fragment should be able to pass each of the thresholds of the three phases.

How large is large enough will depend on various factors — the lengths of the query and of the database, the thresholds for ungapped and gapped alignments, the E-value threshold, and the word size for the initial k-mer matches. Now there is a downward pressure on the size of overlap. Too much overlap will mean the work of matching will be duplicated in nodes that are processing adjacent query fragments. Some earlier, non-parallel implementations of BLAST have suggested overlapping queries, but typically choose extremely large overlap values to avoid missing alignments [30]

Orion chooses the overlap to be t_u, and can find the whole alignment of Figure 5 Fragment 1 sees a partial alignment and Fragment 2 sees a partial alignment, and there is no longer any way to miss any sub-alignments, as shown in Figure 6

B. Alignment aggregation

In Orion, rather than adopting an *ad hoc* approach to fragment overlap, we use a more disciplined strategy. In particular, we note that we can introduce an additional *alignment aggregation* phase to the search process. As Orion processes a single query fragment, if an alignment does not hit a query boundary (i.e., the entire alignment fits in a single fragment), it is returned as normal. But if a partial alignment *does* hit a fragment boundary, it *may* be part of a larger alignment that spans two fragments. Hence, Orion returns these alignments as well.

After all of the query fragments have been processed, Orion performs alignment aggregation. Any alignments that lie entirely within a single fragment can be returned as is (note

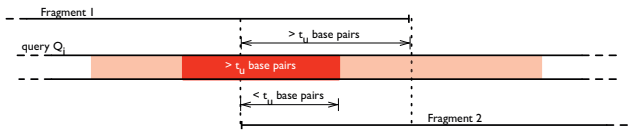

Fig. 7: Alignment with fragment overlap. Fragment 2 must perform gapped extension despite not seeing a high-scoring alignment.

that alignments that lie entirely within the overlap between two fragments will be returned by both fragments). However, any alignments that hit query boundaries must be combined with alignments from the other side of the boundary. Orion "undoes" the overlap between the alignments, merges them together and then reports the result only if the *combined* alignment passes all the score thresholds.

1) Speculative extension: For the reduction phase to work properly, if a partial alignment hits the fragment boundary, Orion must perform gapped extension even if the partial alignment doesn't meet the ungapped alignment threshold. To see why this is necessary, consider the alignment in Figure 7.

The alignment contains a single ungapped subalignment that exceeds the threshold. This subalignment falls entirely within fragment 1, so fragment 1 proceeds with gapped alignment, finding the lightly shaded portions of the alignment. However, fragment 2 *does not* see enough of the ungapped alignment to trigger gapped extension, and hence the portion of the alignment that lies only in fragment 2 would be missed.

To avoid this problem, Orion performs gapped extension *speculatively*: fragment 2 performs gapped extension for its partial alignment anyway. Because the actual score of the ungapped alignment is not known (as it lies partially in fragment 1), Orion uses a relative scoring metric. Rather than extending the alignment until the score drops to t_x below the maximum score seen so far, Orion starts the scoring at 0, and extends the alignment until the score drops to $-t_x$. This results in slightly longer gapped extensions, but the excess is cleaned up during alignment aggregation.

We note, also, that fragment overlap plays a role in speculative extension. If Orion performs an extension, speculative or otherwise, of a partial alignment that hits a fragment boundary, and the extension is terminated (due to X-dropoff) *within the overlap region*, then the partial alignment does not need to be returned, as the neighboring fragment will be able to see the entire alignment (consider if the lightly shaded portion on the right side of Figure 7 did not exist; Fragment 1 would see the entire alignment).

2) Possible missed alignments: There is one corner case where Orion will miss a query alignment that the baseline BLAST would have found. Such a miss happens due to the query fragmentation of Orion, and despite the overlaps in the query fragments. The inaccuracy arises in the case where an alignment spans two fragments, but the portion of the alignment that lies in one fragment does not contain any k-mer matches. In this case, that fragment will not even initiate the search for an alignment. We expect this case to be extremely rare in practice. Experimentally we find that such a miss never happens in our evaluation, and thus we achieve accuracy of 100%.

C. Calculating overlap length

So the question arises what should be the ideal overlap length. The overlap must be at least k: smaller overlaps may result in k-mer hits being missed. Increasing overlap length beyond k makes extensions more likely to terminate within fragment boundaries, resulting in less work during alignment aggregation. Nevertheless, making overlaps too large results in redundant work during the search phase.

We choose our overlap size with these criteria in mind. In particular, we choose our overlap size to ensure that ungapped alignments that pass the t_u threshold lie within each fragment. According to [13], the expected value (E-value) of a single distinct alignment may be calculated by the formula

$$E = Kmne^{-\lambda S}$$

where, K and λ are Karlin-Altschul parameters, m and n are the effective lengths of the query sequence and database, respectively, and S is the alignment score. The "effective" lengths are shorter than the actual lengths to account for the fact that an optimal alignment is less likely to start near the edge of a sequence than it is to start away from that edge. We want to calculate the smallest value of S that will cause the calculated E-score to be less than the threshold E-value (the notation we have used for the latter is simply E).

Putting these constraints together (detailed derivation follows that in [10]), we derive the following formula for fragment overlap (L).

$$S_{lb} = \lceil \frac{ln(Kmn/E_{th})}{\lambda} \rceil$$

$$L = max(k, S_{lb}/p) \quad (1)$$

where, k is the word size of the initial k-mer match, S_{lb} is the shortest ungapped alignment that still passes the E-value test (i.e., calculated E score is exactly equal to E-value and any shorter ungapped alignment will not pass this test). This S_{lb} is then divided by the reward for match of one single bp, p, to come up with the length of the overlap (in terms of bp). To account for the degenerate case where the calculated value of S_{lb}/p is smaller than the length of the initial k-mer match, the max is taken in the final calculation of L.

This choice of L guarantees the following property. Consider two adjacent fragments F_1 and F_2 (Figure 5 or Figure 7 may serve as a reference). If in the baseline (unfragmented) query, there is a sequence with enough of a match with the database such that $E_{calculated} \leq E$, then, there is enough overlap between F_1 and F_2 such that there will be a subsequence in either F_1 or F_2 that will give $E_{calculated}$(subsequence) $\leq E$.

D. Threshold for fragment size

Intuitively, it seems clear that Orion should not fragment a query that is smaller than a certain size. This is due to the fact that there is a certain overhead of fragmentation—divide

the query up, send each query fragment to a separate node, and after the parallel matches, aggregate the results of the individual matches to create the final output. These costs must be balanced against the additional scope for parallelization, and (to a second order effect) better load balancing, that results from fragmenting the query. Further, there is a constant cost of running the baseline sequential BLAST.

Orion takes these two factors into account to select a desired query fragment length. The desired query fragment length depends on *both the database and the exact query* simply because the amount of work that is to be done depends on these two elements. However, for the purpose of calibration, it is clearly infeasible for Orion to determine this desired query fragment length for *every* query for each database it is to be run against. Therefore, we make the practical simplification of performing this calibration once for each database that the matching is going to be performed against. We find that experimentally this simplification is justified with little performance degradation compared to the ideal design choice.

IV. Implementation

In this section we describe the implementation of Orion.

A. Sharding the database and fragmenting the query

Orion uses mpiBLAST's `mpiformatdb` tool to format and to shard the database. It divides the database into a specified number of shards, which are approximately equal in size and are then placed on shared storage.

To fragment the query, Orion uses a simple preprocessing step that takes as input the database length, the original query sequence and the desired fragment length of each query. Orion then calculates the overlap length using Equation 1, fragments the input query sequence using the fragment length and overlap length parameters, and places the fragmented query sequence on shared storage.

B. Parallel BLAST search

Orion's parallel BLAST search on each fragment/shard work unit naturally fits into the MapReduce paradigm [9], with each of the fragment/shard search tasks as a "map" task. We use Hadoop streaming to implement the map phase of the parallel blast search. The map tasks run NCBI blastall for every fragment/shard pair with the specified arguments for the program, the database shard, and the query. The outputs are the parsed BLAST results for search of the query over the respective database shard. The parsed output of BLAST search reports for each alignment the identifier for the database sequence, the offsets of the alignment in the database, the length of the database sequence, the query fragment identifier, query fragment length, offsets of alignment in the query fragment, the sense of the alignment, the E-value, and the number and location of matches, mismatches, and gaps. This information resides in files stored on HDFS. The identifier for the database sequence as the key and the alignment information as the value is fed to the reduce phase.

C. Aggregation of results

The aggregation phase is the Reduce phase of Orion's Map-Reduce job. It is required to merge overlapping alignments that cross over fragment boundaries and present the alignments as a single alignment as would have been reported by BLAST. The key is the database sequence identifier which divides the space of alignments results. In simple words, it first collects all alignments from all the query fragments that matched a particular database sequence together. It then finds overlapping or adjacent alignments from this set and aggregates them. Finally the set contains all aggregated alignments. The benefit of choosing sequence identifier from the database as the key is that multiple reducers can work in parallel over different database sequences.

D. Sorting of results to create final output

Orion outputs alignment results in decreasing order of their scores or increasing E-value. Orion samples the score data for a rough approximation of the distribution of the score values, and then different ranges of values are assigned to different reducers to sort in parallel. Finally the merge is done in parallel, since the range of score values for each reducer task is known. The result is the final set of alignments sorted according to E-values, exactly what would be returned by (serial) BLAST.

V. Evaluation

In this section we present a performance evaluation of Orion on the Gordon supercomputing system. We first compare the execution times of Orion and mpiBLAST, the most popular open-source parallel implementation of BLAST. We then compare the scalability and the effectiveness of load balancing of the two solutions. We also evaluate the overall speedup for Orion, and do a sensitivity study to determine the relationship between query fragment length and execution time for Orion. We use a biologically relevant comparative genomics problem which searches queries from the human genome over the *Drosophila melanogaster* database, to validate that Orion has performance gains in realistic scenarios, as we detail in Section V-B.

A. Experimental Setup

We use the Gordon supercomputing system to run our experiments. Gordon is a dedicated XSEDE cluster maintained by the San Diego Supercomputer Center. Each compute node contains two 8-core 2.6 GHz Intel EM64T Xeon E5 (Sandy Bridge) processors and 64 GB of DDR3-1333 memory. We used a cluster of 64 such nodes, each node having 16 cores.

In these experiments the internal BLAST implementation for both mpiBLAST and Orion used default values for E-value, match rewards, mismatches and gap penalty, and the drop off values and all other configurable parameters (see Table I). The overlap length was calculated using Equation 1. The relevant parameters for the overlap equation are given in Table II.

We used Hadoop version 1.1.1 and mpiBLAST's latest version-1.6.0 in the experiments. The Hadoop cluster was setup such that one node acted as both the master node and the slave node. All the other nodes were configured as slave

Parameter	Value
Length of Drosophilia database	122,653,977
k	0.711
λ	1.374

TABLE II: Parameters required to calculate overlap length

nodes. The master node in the Hadoop cluster assumes the role of namenode, secondary namenode and jobtracker. The slave nodes act as datanodes and tasktrackers. All the nodes in the cluster act as both storage and compute nodes. Each node was configured to run a maximum of 16 map and reduce tasks concurrently, to match the number of cores on the nodes.

B. Biological relevance of evaluation strategy

With the availability of whole-genome sequences for an increasing number of species, we are now faced with the challenge of decoding the information in these sequences. Comparative genome sequence analysis for multiple species at varying evolutionary distances, often termed phylogenetic footprinting, is a powerful approach for identifying protein coding and functional noncoding sequences. Drosophila or fruit fly has been valuable as a model organism for studying human behavior, development, and diseases, given the parallels between the genomes of humans and these tiny flies. In addition, their short life spans and prolific breeding allows for quick turnaround of large-scale biological experiments. Comparison of the Drosophila genome with the human genome, for example, revealed that approximately 75% of human disease genes have homologs in Drosophila [4]. Motivated by this, in this paper we have used Drosophila as a model reference genomic database for aligning a set of long genomic scaffolds of human chromosomes; scaffolds are assemblies of contigs and gaps reconstructed from the NGS reads. The final goal of the genomic comparisons, as done in this paper, would be to explore the evolutionarily-conserved sequences from Drosophila to humans. For example, ultra-conserved elements (UCEs) are arguably the most constrained sequences in the human genome and the majority of these are outside the protein-coding regions [15]. Thus, one exciting use case for such rapid comparisons of long human chromosomal sequences with other databases (e.g., Drosophila database), at different evolutionary distances, could be to discover new UCEs present across varying evolutionary distances. Interestingly, single nucleotide polymorphisms (SNPs) in UCEs have been linked to cancer risk, impaired transcription factor binding, and homeobox gene regulation in the central nervous system [23]. Our future efforts will be directed at aligning long or complete cancer genome sequences, from databases such as the Cancer Genome Atlas Network [18], with normal genome sequences to detect the altered sequences driving different types of cancer.

C. Comparison of Execution Times

In this section we compare the time to completion of a query set for Orion and mpiBLAST. We use human chromosome contigs as our query sequences, and the *Drosophila melanogaster* representing the fruit fly genome as our database. The Drosophila database has an unformatted size of 118MB database and contains 1170 sequences. All the databases were taken from NCBI. Contigs are contiguous sequences that form part of the organism's genome after cleanup has been performed on the raw NGS instrument reads.

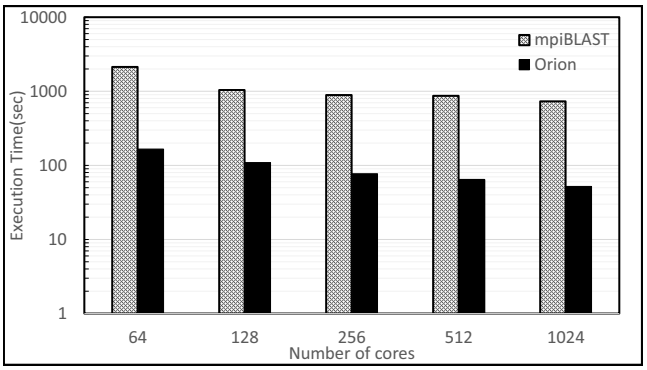

Fig. 8: Execution Time Comparison for Orion and mpiBLAST

Orion is aimed at solving the problem of delivering an efficient and low latency genomic sequence search system for long sequences. To validate this we choose a query set that consists of 16 sequences which are genomic contigs and scaffolds randomly selected from different human chromosomes. The query sizes range from 1 Mbp (Mbp=10^6 base pairs) to 71 Mbp. mpiBLAST performance is sensitive to the number of database shards used, and Orion performance too is sensitive to the number of database shards and query fragments. Hence the number of shards chosen for Orion and mpiBLAST, and the fragment size chosen for Orion were such that both Orion and mpiBLAST have optimal performance for the specific configuration of the experimental machine. We performed the experiment by varying the number of cores in the cluster to study the scalability of Orion and mpiBLAST.

Figure 8 shows the performance of of mpiBLAST and Orion for the chosen query set. Note the logarithmic scale on the Y-axis. From the figure we see that the performance of Orion is significantly better than mpiBLAST at all configurations of number of cores in the system. As expected, as the number of cores increase the execution time goes down for both Orion and mpiBLAST. However Orion performs about 12.3X better on an average for the chosen query set.

Now, looking at the performance on individual query sequences within the query set, we noted that Orion is 23X faster than mpiBLAST for the longest (71 Mbp) of the query sequences. We also noted that the gain of Orion over mpiBLAST increases with increase in query sequence length. Further, mpiBLAST could not handle sequences longer than 96 Mbp and terminated with an error message complaining that it required about 2178 Gb of memory for dynamic programming! The vast majority of the human chromosomes are longer than 96 Mbp and thus, with the current state-of-the-art, we would not be able to run a parallel sequence matching for this wide variety of genomic sequences.

It should be noted that while Orion achieved superior performance for the longer queries, it did not miss any alignments reported by mpiBlast, which is the same as alignments reported by BLAST. Thus, the accuracy of Orion remained at *100% for all the query sequences*.

D. Load Balancing

mpiBLAST's parallelization strategy of segmenting the input query set into individual queries can lead to severe load imbalance among the worker processes. Thus, some processes do the bulk of the work and a majority of the processes terminate quickly. In contrast Orion's query fragmentation strategy divides the entire work into smaller work units, each of which is handled by a Hadoop task. This reduces the variability in load distribution, and enables greater predictability in execution times of each work unit. This ultimately leads to a more efficient use of system resources.

We validate this by comparing the search times of mpiBLAST's processes and Orion's Map and reduce tasks' run times in the 256 core configuration of Experiment 1. Since the running times of the tasks in Orion and the processes in mpiBLAST are not comparable, we use *Coefficient of Variation* (CV) to measure the variability in the run times. CV is defined as Mean/Standard Deviation. Table III shows that CV for mpiBLAST processes' run times is higher than Orion's. This shows that Orion achieves better load balance than mpiBLAST.

Metric	mpiBLAST	Orion
Average (s)	315.78	2.10
Standard Deviation (s)	182.18	0.25
Coefficient of Variation	0.58	0.24

TABLE III: Average, standard deviation (in seconds) and coefficient of variation for processes in mpiBLAST and Map and Reduce Tasks in Orion

E. Scalability Tests

To evaluate the scalability of Orion, we run and profile a sequence search job with long queries over the Drosophila database. The sequences used here are even bigger than the ones used in Experiment 1, we used 32 sequences in the range of 1Mbp-99Mbp, and thus well beyond the usable range of mpiBLAST.

We increase the number of cores in the system from 4 nodes (64 cores) to 64 nodes (1024 cores) and measure the speedup achieved as illustrated in Figure 9. As can be seen, Orion scales to 1024 cores at a nearly constant parallel efficiency, *i.e.,* the slope of the speedup curve is almost constant. At 1024 cores, Orion achieves a speedup of 5 times the baseline of 64 cores. This speedup demonstrates that Orion can fully leverage the massive parallelism of today's supercomputing systems while solving important biological problems.

F. Comparison with Blast+

In this experiment we compared the performance of Orion and BLAST+. BLAST+ is a new suite of BLAST tools that runs on the NCBI servers. It is interesting to compare Orion and BLAST+ since BLAST+ also performs what they call "query splitting" to address the failure of BLAST to run long sequences [6]. BLAST+ is designed to run on standalone Linux/Windows boxes and uses multithreading for enhancing performance. We ran Homo sapiens chromosomal sequences and genomic scaffolds, shown on the X axis in Figure 10 as queries over the Drosophila database using Orion and

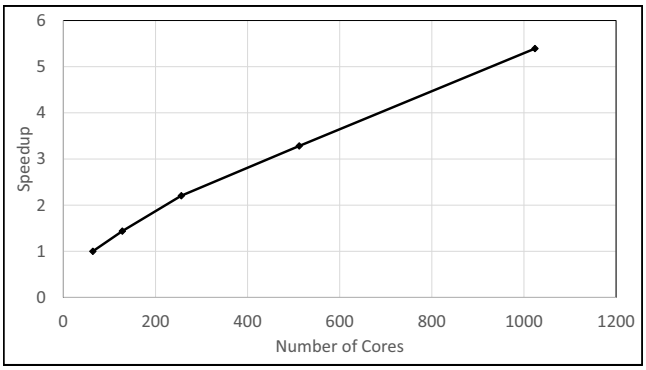

Fig. 9: Speedup for Orion of searching Homo Sapien genomic scaffolds on Drosophila database

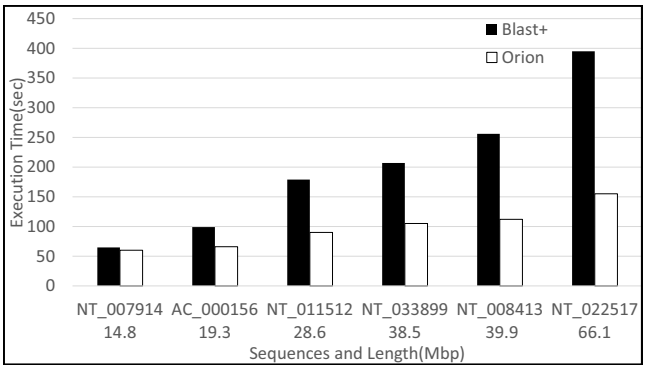

Fig. 10: Comparison of BLAST+ and Orion

BLAST+. We ran BLAST+ with 16 threads to fully utilize the available cores in the node, and ran Orion with 16 Map and Reduce tasks on a single node. Note that BLAST+ is only capable of running on a single node, which severely limits its applicability for large workloads.

As seen in the figure, Orion performs better than BLAST+ for all the sequences with length of more than 10 Mbp. For smaller sequences, BLAST+ performs better than Orion due to the constant overhead of Hadoop job setup and tear down Orion has, which is higher than the completion time of BLAST+ for the smaller queries. However it should be noted that this is a small constant overhead. Also the performance gains for Orion increase with increasing query sequence length. The performance gain of Orion over BLAST+ can be attributed to the finer level of parallelism of Orion. It exploits both intra-database and intra-query parallelism, while BLAST+ can only exploit intra-query parallelism.

G. Sensitivity study of Orion for different fragment lengths

Here, we studied the sensitivity of Orion to different fragment lengths and show the results in Figure 11. We note that there are competing concerns regarding fragment length. Larger fragments mean less opportunities for alignments to cross boundaries, and thus less work to perform during alignment aggregation. However, as fragments get longer, the scope for parallelism decreases, and if fragments get too long,

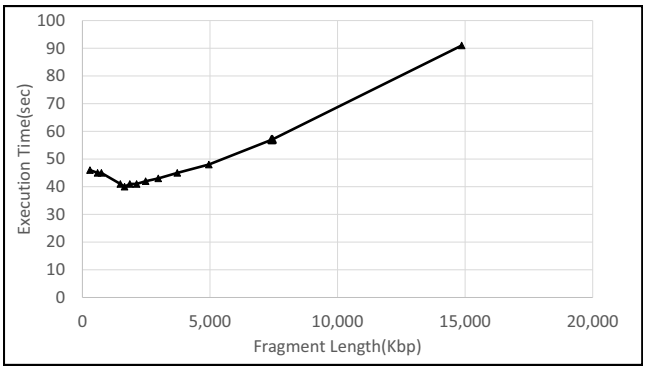

Fig. 11: Sensitivity of Orion to fragment length

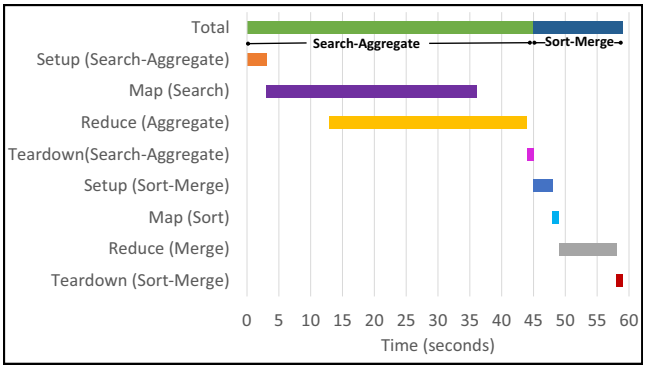

Fig. 12: Timeline of events for Orion

BLAST (which Orion uses) begins to suffer from poor cache behavior [6]. In addition the number of work units which is given by the number of query fragments times the number of database shards, should be larger than the number of available cores. Hence, we expect there to be a sweet spot in performance. We show this sweet spot for a 14.5M base-pair query against the Drosophila database. The ideal fragment length is 1.6M base pairs. This kind of calibration of Orion can be done once, for each database and then it can be used with the optimal (or near optimal) fragment size determined during the calibration. Note that for small queries, with size smaller than the optimal fragment length, the sweet spot is never hit and Orion does not benefit from fragmenting the query.

H. Time distribution for phases of Orion

Orion's running time can be decomposed into two primary MapReduce jobs: (1) the Search-Aggregate job and (2) the Sort-Merge job. We profiled Orion to determine how each job contributes to the total execution time of Orion. Because the behavior of Orion varies from fragment to fragment, we present results of aligning a representative input sequence, NT_007914, which has 14.5M base pairs, with the Drosophila database. The experiment was run on a single node, with 16 mappers and 16 reducers.

For each MapReduce job, we profiled the different phases of the job: (1) Setup, (2) Map, (3) Reduce, and (4) Teardown. For each job, and phase within the job, we recorded at what time the job/phase began and ended. The timeline of events is shown in Figure 12. The different phases of the two jobs are shown on the Y axis while the X axis shows the timeline. Note that Hadoop takes advantage of pipelining, so portions of each job's Map phase can overlap with its Reduce phase.

Based on these measurements, we see that the execution time is dominated by the Search-Aggregate job. Within this job, the Map phase performs the initial search for alignments and runs in a completely parallel manner, while the Reduce phase aggregates together the results from the query fragments. The Reduce phase overlaps with the Map phase, as the Reduce phase includes the time to copy results from the mappers to the reducers, and the copying starts while the Map phase is still running. The aggregation itself begins once all the data is available.

The Sort-Merge job also executes as a Map Reduce job, with the Map partitioning the alignments with different range of scores and assigning them to different reducers. The Reduce phase merges all these alignments into the final sorted output. Since the Map phase is completely parallel, the Reduce (Merge) phase dominates the sorting time as expected. The setup and teardown times are significant for jobs with short execution times. This detailed breakdown and timing of all the phases of execution, helps to understand Orion's behaviour and can be used for optimizing the execution times of Orion.

I. Results on larger databases

Orion consistently outperforms mpiBLAST over other databases as well. We also performed experiments over two larger databases — the Mouse genome database (unformatted size 2.77G) and the NT database (56.5 G) and found similar results. For example, with mpiBLAST, the search of a single query sequence NG_007092 having 2311 Kilo basepairs over the mouse database took 2664 seconds to complete while Orion completed the search in 201 seconds while for the even larger NT database a single query sequence NT_077570, having 263 Kilo base pairs, took almost an hour and a half (5,271.8 seconds), while Orion ran in 15 minutes using the sweet spot for the fragment length determined for query NT_077570. Thus Orion also scales to bigger databases. At these large database sizes the difference in matching times between our solution and the current state-of-art becomes even more significant and impactful.

VI. Related Work

A vast body of research [24], [14], [21], [17], [20], [16], [22], [28] has addressed the parallelization of sequence alignment algorithms based on various parallel programming paradigms in the wake of the massive data sets generated by next-generation high-throughput sequencing systems. These parallelization methods can be classified into two categories by their approaches to data decomposition. In the first category, where mpiBLAST belongs, the database that contains reference sequences is partitioned into multiple shards and hence a query sequence can be searched simultaneously against different shards by different execution units, i.e., processes or threads. Methods in the second category consider a large set of queries and by parallelizing the alignment of multiple queries, reduce the overall finish time. The set of queries

are simply split into smaller subsets and the alignment of different subsets are executed in parallel by different execution units. The rest of this section reviews several representative works from both the categories. None of the schemes described here adopt the same query fragmentation strategy as Orion (though some fragment sequences in the database). To our knowledge, Orion's fragmentation strategy is unique among parallel BLAST implementations.

CloudBurst [24] is modeled after the short read-mapping program RMAP [26], but implements the algorithm as a classic MapReduce program to parallelize execution using multiple compute nodes. Like RMAP, CloudBurst takes a seed-and-extend approach, extracting all k-mers in the reference sequence and non-overlapping k-mers in all queries in the map phase, sorting all k-mers by their sequence in the shuffling phase, and finally in the reduce phase identifying k-mers shared between the reference sequence and the queries and extending them into end-to-end alignments allowing for a fixed number of insertions, deletions or mismatches. It is optimized for the alignment of many short queries against long reference sequences. Like the database sharding in mpiBLAST, CloudBurst partitions database sequences into 65 kb chunks with 1kb overlaps to support cross-chunk alignment of queries shorter than 1 kb. The shuffle phase which is essentially an all-to-all communication among all compute nodes imposes a high throughput demand on the network and will eventually become the scalability bottleneck.

CloudBLAST [16] uses the Hadoop MapReduce framework to parallelize the alignment of a set of queries. Similar to our approach that builds on top of the established BLAST implementation, CloudBLAST runs BLAST as map tasks of Hadoop over a distributed cluster of virtual machines. The set of queries are partitioned into subsets, which is then assigned to map tasks that search the subset of queries over the entire database. Without exploiting the parallelism from database sharding, CloudBLAST suffers poor performance when dealing with large reference databases.

Yang et al. [30] also identify the scalability limitation of BLAST for long query sequences and employ the Hadoop framework to speedup the alignment of long sequences. The parallelism in their scheme comes solely from database sharding: they exploit the file segmentation in HDFS to split a large database into 64MB chunks and run a query as parallel map tasks against different chunks of the database. Long reference sequences in the database are also split into fragments of fixed size with overlaps to reduce the possibility that a map task needs to access chunks of the database stored on a remote node.

GPU-BLAST [28] achieves nearly 4X speedup on a 1.15 GHz NVIDIA Fermi GPU over the single-threaded NCBI-BLAST running on an 2.67 GHz Intel Xeon CPU. It takes the database sharding approach by assigning the reference sequences in the database to different GPU threads for parallel alignment. To mitigate the performance penalty from thread divergence, GPU-BLAST also includes a preprocessing step that sorts all reference sequences in the database by their lengths to avoid having threads of the same warp work on reference sequences with significant length differences.

VII. Conclusions

With the ever increasing importance of gene sequencing and alignment to systems biology, and the corresponding increase in the number, size and variety of queries and genomic databases, it is of paramount importance that computational sequencing algorithms be parallelized efficiently. Prior approaches to parallel BLAST search did not exploit all available parallelism, leading to unacceptably slow performance when performing matches on large query sequences. In this paper, we have presented Orion, which uses a novel parallelization strategy, fragmenting individual queries into overlapping fragments. Through a careful analysis, we determine how to fragment the queries such that the accuracy of the final alignments is not reduced. Our evaluation with real biological use cases shows that Orion significantly outperforms the most popular parallel BLAST implementation, called mpiBLAST, for large queries. For example, with a large NCBI database called the NT database and a long query corresponding to a human chromosome, Orion shows a 5X improvement in execution time over mpiBLAST. Further, the nature of genome alignment is such that static scheduling does not work well. As a result, mpiBLAST shows significant load imbalance. Orion on the other hand, thanks to using the Hadoop framework, achieves load balancing across all the computational cores.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments and feedback. This work was supported in part by NSF grant CCF-1337158. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.

References

[1] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[2] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402, 1997.

[3] Dennis A Benson, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, and Eric W Sayers. Genbank. *Nucleic acids research*, 38(suppl 1):D46–D51, 2010.

[4] Ethan Bier. Drosophila, the golden bug, emerges as a tool for human genetics. *Nature Reviews Genetics*, 6(1):9–23, 2005.

[5] RC Braun, Kevin T Pedretti, Thomas L Casavant, Todd E Scheetz, Clayton L Birkett, and Chad A Roberts. Parallelization of local blast service on workstation clusters. *Future Generation Computer Systems*, 17(6):745–754, 2001.

[6] Christiam Camacho, George Coulouris, Vahram Avagyan, Ning Ma, Jason Papadopoulos, Kevin Bealer, and Thomas Madden. Blast+: architecture and applications. *BMC bioinformatics*, 10(1):421, 2009.

[7] N. Camp, H. Cofer, and R. Gomperts. High throughput BLAST. Technical report, Silicon Graphics, Inc., 1998.

[8] Aaron Darling, Lucas Carey, and Wu-chun Feng. The design, implementation, and evaluation of mpiblast. *Proceedings of ClusterWorld*, 2003, 2003.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] E. Michael Gertz. BLAST Scoring Parameters. ftp://ftp.cbi.edu.cn/pub/software/blast/documents/developer/scoring.pdf.

[11] Clark Francis. Fragblast. http://www.clarkfrancis.com/blast/fragblast.html.

[12] Mark K Gardner, Wu-chun Feng, Jeremy Archuleta, Heshan Lin, and Xiaosong Ma. Parallel genomic sequence-searching on an ad-hoc grid: experiences, lessons learned, and implications. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 22–22. IEEE, 2006.

[13] Samuel Karlin and Stephen F Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proceedings of the National Academy of Sciences*, 87(6):2264–2268, 1990.

[14] Ben Langmead, Kasper Hansen, and Jeffrey Leek. Cloud-scale rna-sequencing differential expression analysis with myrna. *Genome Biology*, 11(8):R83, 2010.

[15] Igor V Makunin, Viktor V Shloma, Stuart J Stephen, Michael Pheasant, and Stepan N Belyakin. Comparison of ultra-conserved elements in drosophilids and vertebrates. *PloS one*, 8(12):e82362, 2013.

[16] Andréa Matsunaga, Maurício Tsugawa, and José Fortes. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 222–229. IEEE, 2008.

[17] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A. DePristo. The genome analysis toolkit: A mapreduce framework for analyzing next-generation dna sequencing data. *Genome Research*, 20(9):1297–1303, 2010.

[18] Roger McLendon, Allan Friedman, Darrell Bigner, Erwin G Van Meir, Daniel J Brat, Gena M Mastrogianakis, Jeffrey J Olson, Tom Mikkelsen, Norman Lehman, Ken Aldape, et al. Comprehensive genomic characterization defines human glioblastoma genes and core pathways. *Nature*, 455(7216):1061–1068, 2008.

[19] Ilene Mizrachi. Genbank. *National Center for Biotechnology Information (US)*, 2013.

[20] Tung Nguyen, Weisong Shi, and Douglas Ruden. Cloudaligner: A fast and full-featured mapreduce based tool for sequence mapping. *BMC Research Notes*, 4(1):171, 2011.

[21] Henrik Nordberg, Karan Bhatia, Kai Wang, and Zhong Wang. Biopig: a hadoop-based analytic toolkit for large-scale sequence data. *Bioinformatics*, 29(23):3014–3019, 2013.

[22] Luca Pireddu, Simone Leo, and Gianluigi Zanetti. Mapreducing a genomic sequencing workflow. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 67–74, New York, NY, USA, 2011. ACM.

[23] Taewoo Ryu, Loqmane Seridi, and Timothy Ravasi. The evolution of ultraconserved elements with different phylogenetic origins. *BMC evolutionary biology*, 12(1):236, 2012.

[24] Michael C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[25] Scott Schwartz, W James Kent, Arian Smit, Zheng Zhang, Robert Baertsch, Ross C Hardison, David Haussler, and Webb Miller. Human–mouse alignments with blastz. *Genome research*, 13(1):103–107, 2003.

[26] Andrew Smith, Zhenyu Xuan, and Michael Zhang. Using quality scores and longer reads improves accuracy of solexa read mapping. *BMC Bioinformatics*, 9(1):128, 2008.

[27] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[28] Panagiotis D. Vouzis and Nikolaos V. Sahinidis. Gpu-blast: Using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 2010.

[29] Pierre Wit, Melissa H Pespeni, Jason T Ladner, Daniel J Barshis, François Seneca, Hannah Jaris, Nina Overgaard Therkildsen, Megan Morikawa, and Stephen R Palumbi. The simple fool's guide to population genomics via rna-seq: an introduction to high-throughput sequencing data analysis. *Molecular ecology resources*, 12(6):1058–1067, 2012.

[30] Xiao-liang Yang, Yu-long Liu, Chun-feng Yuan, and Yi-hua Huang. Parallelization of blast with mapreduce for long sequence alignment. In *Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on*, pages 241–246. IEEE, 2011.