

Coordinating Computation and I/O in Massively Parallel Sequence Search

Heshan Lin, *Member, IEEE*, Xiaosong Ma, *Member, IEEE*, Wuchun Feng, *Senior Member, IEEE*, and Nagiza F. Samatova

Abstract—With the explosive growth of genomic information, the searching of sequence databases has emerged as one of the most computation- and data-intensive scientific applications. Our previous studies suggested that parallel genomic sequence-search possesses highly irregular computation and I/O patterns. Effectively addressing these run-time irregularities is thus the key to designing scalable sequence-search tools on massively parallel computers. While the computation scheduling for irregular scientific applications and the optimization of noncontiguous file accesses have been well studied independently, little attention has been paid to the interplay between the two. In this paper, we systematically investigate the computation and I/O scheduling for data-intensive, irregular scientific applications within the context of genomic sequence search. Our study reveals that the lack of coordination between computation scheduling and I/O optimization could result in severe performance issues. We then propose an integrated scheduling approach that effectively improves sequence-search throughput by gracefully coordinating the dynamic load-balancing of computation and high-performance noncontiguous I/O.

Index Terms—scheduling, parallel I/O, bioinformatics, parallel genomic sequence search, BLAST

1 INTRODUCTION

Genomic sequence search forms an important class of applications used widely and routinely in the life sciences. A sequence-search tool compares a set of query sequences against a database of DNA or amino-acid sequences using an alignment algorithm and then reports the statistically significant matches between the query sequences and the database sequences. The similarities that are found between a new unknown sequence and a sequence of known function can help identify the function of the new sequence and find sibling species from a common ancestor.

Today, the collective amount of genomic information is doubling every 12 months [1]–[3] while the computational horsepower of a single processor is only doubling every 18–24 months. The widening disparity between computational capability and the massive technological advances in sequencing technology necessitates highly scalable sequence-search tools that are capable of taking advantage of state-of-the-art massively parallel computers. For instance, initial studies have shown that large sequence-search jobs, such as genome-to-genome comparisons [4] and database-to-database alignments [5], take hours or even days to complete on supercomputers consisting of thousands of processors.

Our past experiences suggest that the key design challenge of massively parallel sequence-search tools lies in their highly irregular computation-and-I/O patterns:

- The execution time of a search task is hard to predict from metrics such as the size of the input data. Tasks processing the same amount of input can have execution times that differ by *orders of magnitude* [5].
- The output data distribution on different processes is fine-grained as well as irregular, varying from one query to another depending on the search results [6].

The reasons for these run-time irregularities are two-fold. First, the required compute time and the output data distribution of a search task depend on the similarities between the compared sequences, which can vary significantly even between different tasks processing the same amount of input data. Second, popular sequence-alignment algorithms, such as BLAST [7], [8], employ heuristics to improve computational efficiency, making it hard to predict the execution time of a search task.

The efficient scheduling for irregular scientific applications has been extensively investigated during the last decade, with a wealth of techniques proposed for dynamic load-balancing by leveraging applications' runtime profiles [9]–[12] or probabilistic signatures [13]–[17]. Most of these scheduling studies, however, focus on compute-intensive applications and do not address the irregular I/O issue in data-intensive applications. Existing studies on noncontiguous I/O optimizations [18]–[23], on the other hand, emphasize improving *actual* I/O performance in isolation and often ignored interactions with the scheduling of computation. Thus, the interaction between I/O optimizations and the computation scheduling for irregular scientific applications has not been well understood.

In this paper, we systematically address the runtime irregularities of parallel genomic sequence-search applications. We consider our contributions as follows:

- An empirical study that shows how the lack of coordination between the I/O optimization and the com-

• Heshan Lin and Wuchun Feng are with the Department of Computer Science, Virginia Tech.
E-mail: {hlin2,feng}@cs.vt.edu.

• Xiaosong Ma and Nagiza F. Samatova are with the Department of Computer Science, North Carolina State University and the Computer Science and Mathematics Division, Oak Ridge National Laboratory.
E-mail: {ma,samatova}@csc.ncsu.edu.

putation scheduling can result in severe performance degradation for genomic sequence search, a representative example of data-intensive applications with irregular compute kernels.

- An integrated scheduling approach that gracefully coordinates fine-grained, dynamic load-balancing and asynchronous output processing to maximize throughput and prevent the aforementioned performance degradation for large-scale sequence search.
- An asynchronous, two-phase I/O technique that optimizes concurrent noncontiguous I/O access without paying synchronization overhead imposed by traditional collective I/O techniques.
- A performance evaluation of our integrated scheduling approach and asynchronous two-phase I/O, running atop mpiBLAST [24], an open-source, parallel sequence-search tool. Hereafter, we refer to the above collectively as *mpiBLAST-PIO*, short for mpiBLAST with parallel I/O¹.

2 BACKGROUND AND OTHER RELATED WORK

2.1 Genomic Sequence-Search Algorithms

Genomic sequence-search tools employ different alignment algorithms to compare a query sequence against a database of sequences. Examples of such alignment algorithms include Smith-Waterman [25], Needleman-Wunsch [26], FASTA [27] and BLAST [7]. Of these, the BLAST family of algorithms is most widely used in biological and biomedical research. BLAST compares a query sequence against a database of sequences with a two-phase, heuristic-based alignment algorithm. The database sequences that are most similar (measured by a statistic called *E-value*) to the query sequence will be reported along with the matching results.

2.2 Genomic Sequence-Search Parallelization

Because searching a sequence database is computation- and data-intensive, many parallel approaches have been investigated to cope with the rapid growth of sequence databases. Hardware-based solutions [28]–[30] parallelize the computation of individual alignments. These solutions are highly efficient but require custom hardware such as field-programmable gate arrays (FPGAs). The optimization techniques presented in this paper focus on software-based parallel solutions. These techniques, however, can be generalized to efficiently glue many hardware processing units together to deliver even higher search throughput.

Early parallel sequence-search software adopted the *query segmentation* approach [31]–[33], where a sequence-search job is parallelized by having individual compute nodes concurrently search disjoint subsets of queries against the whole sequence database. This *embarrassingly parallel* approach scales well when the database can fit in the memory of a compute node. However, query segmentation suffers expensive paging overhead when the database is much larger than the memory of a compute node. This

issue motivated *database segmentation* [24], [34]–[36], where the sequence database is partitioned and distributed across compute nodes. By fitting large databases into the aggregate memory of multiple nodes, database segmentation eliminates the paging issue and allows timely sequence analysis to keep up with the fast growing database sizes. However, this approach introduces computational dependencies between individual nodes because the distributed results generated at different nodes need to be merged to produce the final output. The parallel overhead caused by merging results increases as the system size grows, consequently limiting the program’s scalability for large-scale deployments.

In summary, neither query segmentation nor database segmentation alone is sufficient to offer scalable sequence-search solutions on massively parallel computers. Consequently, recent efforts adopted a combination of both segmentation approaches. Rangwala *et al.* developed a parallel BLAST that extended and optimized pioBLAST [36], a research prototype developed in one of our previous studies, on Blue Gene/L [38]. Oehmen *et al.* presented ScalaBLAST [39], a highly efficient parallel BLAST built on top of the Global Array [40] toolkit. These tools organize processors into equal-sized groups and assign a subset of input queries to each group. Within a processor group, each processor searches the assigned queries on a distinct portion of the database. Both tools employ static load balancing and *implicitly assume that the execution time of a BLAST search task is predictable from the total sizes and/or the numbers of the input queries*. For instance, in ScalaBLAST, the input queries are split among processor groups such that the query batch assigned to each group contains presumably the same amount of “work units.” The work units of a query batch are calculated based on a “*trial-and-error*” model that takes into account both the total number of characters and the number of queries.

However, our recent study discovered that for certain types of BLAST search, there is *no clear correlation* between the amount of input data and the execution time of a sequence-search task [5]. In addition, tasks processing the same amount of input can have execution times *differing by orders of magnitude*. The weak correlation between the sequence size and its search time can be attributed to that 1) the amounts of alignment computing depends more on the similarities between the sequences than on the sequence sizes; 2) the BLAST algorithm is heuristic-based, and thus, its execution time is not necessarily proportional to the sequence sizes. As a consequence, the statically partitioned and load balanced approaches of [38] and [39] will result in significant performance degradation due to their implicit assumption of a predictable correlation between the amount of input data and the execution time of a sequence-search task. Observing the limitations of existing static-load balancing approaches, with mpiBLAST-PIO, we adopt a dynamic load-balancing approach and focus on effectively reducing the query scheduling overhead. Our previous study showed that our dynamic load balancing approach can deliver 30% performance improvements over the aforementioned static load balancing approaches when searching skewed DNA sequences [37].

1. The mpiBLAST-PIO package is available for download at www.mpiblast.org.

Finally, this paper extends from the preliminary studies conducted in our three previous conference publications [36], [37], [41]. The distributed result processing (Section 3.3) was first introduced in our IPDPS'05 paper [36]. A preliminary design of the hierarchical architecture (Section 3.1) was presented in our ACM CF'07 paper [41]. Part of the dynamic load balancing optimizations (Section 3.2) and I/O optimizations (Section 3.4) was discussed in our SC'08 paper [37].

2.3 mpiBLAST

mpiBLAST [24] is an open-source, sequence-search tool that parallelizes the NCBI BLAST toolkit [7]. The original design of mpiBLAST follows a database segmentation approach and a master-worker style. The master uses a greedy algorithm to assign pre-partitioned database fragments to workers. Each worker then concurrently performs a BLAST search on its assigned database fragment. The results from different workers are merged and written to the file system on the master. mpiBLAST achieves good speedups when the number of processes is small or moderate. However, as we shall see in Section 3.3, the scalability of mpiBLAST, and any approaches that employ database segmentation for that matter, can be greatly hampered by its centralized output processing design.

2.4 MapReduce

MapReduce is a programming model designed to simplify parallel data processing on commodity clusters [42]. While MapReduce has proven to be effective for many web data processing applications, it is less suitable for large-scale sequence search for several reasons. First, MapReduce only handles one-dimensional input and hence is not suitable for implementing both query segmentation and database segmentation approaches. Moretti et al. has reported a similar observation that MapReduce is not sufficient to express all-to-all style computation [43]. The existing MapReduce BLAST implementation, i.e., CloudBLAST [44], only implements query segmentation and stores the entire database on each node. As we discussed in Section 2.2, such a design is not scalable to large databases. Second, cluster MapReduce implementations are coupled with distributed file systems such as the Google File System [45] that require local storage on each compute node. However, many supercomputers today adopt a disk-less design on compute nodes. Third, MapReduce imposes a restricted data format, i.e., data is stored in key-value pairs. The output of sequence-search tools such as BLAST is not completely structured. For instance, output data such as global search statistics are hard to express in key-value pairs, making it difficult to implement database segmentation in MapReduce. The flexibility of the MPI model allows fine-grained processing of this kind of unstructured data.

2.5 Optimization of Noncontiguous I/O

With database segmentation, the search results generated at each processor will appear noncontiguously in the global output file. Thus, optimizing noncontiguous I/O is important to improve the performance of parallel sequence

search. Traditionally, there are two techniques widely adopted to optimize noncontiguous I/O performance used in popular parallel I/O libraries such as ROMIO [19].

The first one is *data sieving*, introduced in the PASSION I/O library [46], which targets noncontiguous I/O requests issued from one process. It replaces the original small, noncontiguous I/O requests with larger ones and uses additional in-memory data manipulation to pick out portions of the data specified in the original read request(s) or to update portions of the data specified in the original write request(s). Since these operations are much faster than disk seeks, data sieving can considerably improve noncontiguous I/O performance at the cost of accessing extra amounts of data. However, with many processes accessing shared files with fine-grained and interleaved write patterns, such as the output of parallel sequence-search applications, data sieving can incur too much extra data access and, hence, yield unsatisfactory I/O performance.

The second technique, *collective I/O*, was designed to improve parallel noncontiguous I/O by having multiple processes coordinate their operations to combine small, noncontiguous I/O requests into large, sequential ones [18], [19], [47]. The most popular collective I/O strategy used today is *two-phase I/O* [18]. With a two-phase write, involved processes first exchange data to form a contiguous write request, then write such buffered blocks to the file system in parallel. Collective I/O has been widely used in data-intensive numerical simulations. In this paper, we argue that collective I/O, when implemented synchronously, incurs high synchronization cost whenever computational phases of a parallel program are not balanced.

Recently, Bent et al. designed PLFS [48], a fast checkpoint file system that optimizes noncontiguous write accesses by having processes log their checkpoint data and the corresponding offsets to individual files. For read accesses, a global index needs to be constructed to offer a logical view for the logged data belonging to a file. Although PLFS can considerably improve the noncontiguous write performance, it does not fully address the I/O issue in large-scale sequence search. Unlike the checkpoint data, sequence search results need to be frequently read for further analysis. However, read accesses in PLFS are inefficient, because data is stored noncontinuously, and accessing the global index incurs extra overhead. In addition, PLFS generates one output file per process, which would significantly increase the file management overhead on peta-scale machines with hundreds of thousands of processors.

3 INTEGRATED COMPUTATION AND I/O SCHEDULING

In this section, we first present the software architecture of mpiBLAST-PIO. Then we discuss the details of our proposed computation and I/O scheduling optimizations.

3.1 Software Architecture

mpiBLAST-PIO adopts a hierarchical architecture as depicted in Fig. 1. At the top level, processors in the system are organized into equal-sized *partitions*, which are supervised by a dedicated *supermaster* process. The supermaster

is responsible for assigning tasks to different partitions and handling inter-partition load balancing. Within each partition, there is one *master* process and many *worker* processes. The master is responsible for coordinating both computation and I/O scheduling in a partition. It periodically fetches a subset of query sequences (defined as a *query segment*) from the supermaster and assigns them to workers, and it coordinates the output processing of queries that have been processed in the partition. Such a hierarchical design avoids creating scheduling bottlenecks as the system size grows by distributing the scheduling workload to multiple masters.

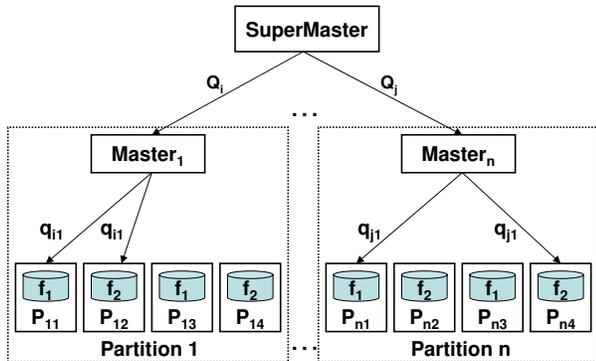


Fig. 1. mpiBLAST-PIO software architecture. Q_i and Q_j are query segments fetched from the supermaster to masters, and q_{i1} and q_{j1} are query sequences that are assigned by masters to their workers.

In this architecture, the compute processes in the system are segregated into two groups – masters and workers. In order to maximize the system throughput, it is important to keep both groups of processes equally busy so that the system idleness is minimized. The key to balancing the workload between the masters and workers is to choose an appropriate partition size S_p (defined as the number of workers in the partition). To this end, our design supports mapping an arbitrary number of workers to a master and allows users to determine the appropriate S_p value through initial profiling with sampled sequences from the original query.² As the master’s workload increases monotonically as S_p grows, an optimal S_p can be found by comparing the program performance at gradually increased S_p values. Note that the optimal S_p value is platform- and workload-dependent. While automatic tuning of this parameter is challenging and beyond of the scope of this paper, our experiences on multiple platforms and search workloads suggest that using 128 workers per partition is a reasonably good choice.

To enable database segmentation, mpiBLAST-PIO pre-partitions the sequence database into fragments and stores the fragments onto the shared file system. mpiBLAST-PIO defines two running modes, *non-sharing* and *sharing*, according to how the databases are distributed and stored on the worker nodes.

2. We found that using randomly sampled query sequences from a BLAST job to perform initial profiling is practical in finding appropriate parameter values in our system.

The *non-sharing* mode assumes that input database fragments are not shared among different parallel BLAST jobs. This mode is suitable for platforms without locally attached disks, such as IBM Blue Gene systems. In this mode, the fragments are replicated to the worker nodes’ memory in a way similar to those replication schemes used in previous parallel BLAST studies [38], [41]. During system initialization, all workers in the system are organized into temporary equal-sized replication groups, and the first group is designated as the I/O group. All database fragments are disjointly assigned to the workers in the I/O group in a round-robin fashion. Each worker in the I/O group then reads in its assigned fragments in parallel and broadcasts them to the corresponding workers in all other groups.

In institutions where BLAST is heavily used by many users and cluster nodes are equipped with locally attached disks, it is desirable to enable sharing of common sequence databases between BLAST jobs to reduce the cost of data movement. In the *sharing mode*, the database fragments used by a BLAST job will remain on the local disks of the worker nodes after the job is finished. At the beginning of mpiBLAST-PIO execution, workers report the cached fragments on their local disks to the master. This fragment distribution information is then taken into account during the scheduling decision.

3.2 Fine-Grained, Dynamically Load-Balanced Computation Scheduling

BLAST search time is highly variable and unpredictable, as found in our past research [5]. To the best of our knowledge, there is no effective way to estimate the execution time of a given BLAST search. So, without a priori knowledge of queries’ processing time, we simply use a greedy scheduling algorithm to assign fine-grained tasks to idle processes.

To effectively balance loads across multiple partitions, we assign a small query segment to each partition, especially for running on supercomputers with a large number of partitions. On the other hand, fine-grained, query-segment allocation incurs two problems. First, the scheduling overhead increases as the query segment size decreases. Second, using small query segments forces frequent synchronization between the workers within each partition, leaving faster workers to wait for their slower peers to finish before acquiring a new segment of queries to work on. In particular, with small query segments, there are not enough queries to “cancel out” the per-query imbalance of search time, therefore intra-partition load imbalance may worsen, resulting in degradation of resource utilization.

We address the above problems associated with small scheduling granularity via *dynamic worker grouping* and *proactive query prefetching*. Specifically, the search of a query is broken into a set of tasks corresponding to the set of database fragments this query has to be searched against. The masters dynamically maintain a window of outstanding tasks. Whenever a worker finishes its current task, it contacts the master to request another one. The set of workers that work on one particular query is thus dynamically formed. This can achieve better load balancing within a partition compared to statically assigning search tasks of a query to a fixed set of workers. With query prefetching,

the master requests the next query segment when the total number of outstanding tasks in the window falls under a certain threshold. By combining these two techniques, workers will not be slowed down by waiting for their peers or for the next batch of query sequences.

The scheduling process running on the master is given in Algorithm 1 below. The master maintains a list of query sequences, QL , that are being processed in the partition. A query sequence in QL corresponds to $|F|$ individual tasks, each searching the query sequence against a distinct database fragment. The master keeps track of how many tasks have been completed by workers. When observing that the number of total incomplete tasks of all query sequences in QL is less than the number of workers ($|W|$) in the partition, the master issues a query prefetching request to the supermaster. To overlap network communication with local job scheduling, the master receives the query segment in the background with a non-blocking MPI call. The new query segment received from the supermaster is then appended to the end of QL .

Algorithm 1 Master Scheduling Algorithm

```

Let  $QL = \{q_1, q_2, \dots\}$  be the list of unfinished query sequences
Let  $F = \{f_1, f_2, \dots\}$  be the set of database fragments
Let  $Unassigned_i \subseteq F$  be the set of unassigned database fragments
for query sequence  $q_i$ 
Let  $W = \{w_1, w_2, \dots\}$  be the set of workers in this partition
Let  $D_i \subseteq W$  be the set of workers that cached fragment  $f_i$ 
Let  $C_i \subseteq F$  be the database fragments cached by worker  $w_i$ 
Let  $assignment_i$  refer to the assignment to the  $i^{th}$  worker
Require:  $|W| \neq 0$ 
while not all query sequences have been finished do
  if number of all unassigned fragments in  $QL < |W|$  then
    Issue segment prefetching request to supermaster
  end if
  if Received a query segment  $QS$  from supermaster then
    for  $q_i \in QS$  do
      Append  $q_i$  to  $QL$ 
       $Unassigned_i \leftarrow F$ 
    end for
  end if
  Receive task request from worker  $w_j$ 
   $q_c \leftarrow QL.head$ 
   $assignment_j \leftarrow \langle \emptyset, 0 \rangle$ 
  while  $q_c \neq QL.tail$  and  $assignment_j = \langle \emptyset, 0 \rangle$  do
    if  $\exists f_i \in Unassigned_c$  and  $w_j \in D_i$  then
      Find  $f_k$  such that  $k = \arg \min(|D_k|)$ 
      and  $w_j \in D_k$  and  $f_k \in Unassigned_c$ 
       $assignment_j \leftarrow \langle q_c, f_k \rangle$ 
    else
      if sharing mode is used then
        Find  $f_k$  such that  $k = \arg \min(|D_k|)$ 
        and  $f_k \in Unassigned_c$ 
         $assignment_j \leftarrow \langle q_c, f_k \rangle$ 
      end if
    end if
    if  $|Unassigned_c| = 0$  then
       $QL.head \leftarrow QL.head.next$ 
    end if
     $q_c \leftarrow q_c.next$ 
  end while
end while

```

In the above design, the size of a prefetched query segment (S_q) is configurable. Using a smaller S_q can yield better load balancing results, but it increases the number of messages sent to the supermaster. In practice, S_q can be configured to an arbitrarily small value as long as the

supermaster is not overloaded by the prefetching messages. Based on our experiments on the System X cluster (configuration details will be described in Section 4.1) using 1024 processors, for typical BLAST searches the supermaster is not a performance bottleneck even when the S_q is set to 1. In our previous study that used a similar scheduling algorithm on the IBM Blue Gene/P system [37], we found that a S_q value of 5 was sufficient to scale across 32,768 processing cores.

Within each partition, workers periodically report to the master for assignments when idle. Upon receiving a task request from an idle worker (w_j), the master scans QL to determine a task for w_j as follows. For the current query sequence being examined (q_c):

- 1) If w_j has cached some database fragments that have not been searched against q_c , the cached fragment that is least distributed (i.e., cached by fewest workers) in the partition will be assigned to the worker.
- 2) If w_j has not cached any unsearched fragment of query sequence q_c and the sharing mode is used, the least-distributed fragment is assigned to w_j , who will then load the assigned fragment from the shared file system into its local cache before the computation.

Here data locality is taken into account to reduce data movements and keep partition-wide data distribution to a minimum under the sharing mode. If no tasks can be found for this worker, the scheduling algorithm moves on to the next query in QL . The same scheduling procedures are repeated until a task is decided for the idle worker or until all unfinished query sequences in QL have been examined, in which case the worker has finished its own portion of work. By allowing incomplete tasks to be independently scheduled to any worker that has cached the corresponding database fragments, our scheduling algorithm helps balance the workload among workers even when execution times of different search tasks are highly skewed.

3.3 Scalable Distributed Result Processing

While most existing studies on parallel sequence search have focused on parallelizing the computational part of sequence search, we have found that the efficient handling of output data is crucial in sustaining parallel execution efficiency on large-scale systems. In this section, we identify several performance issues in mpiBLAST's original protocol for processing results. We then present a lightweight alternative for mpiBLAST-PIO. The new protocol can significantly reduce the non-search overhead of sequence-search tools using the *database segmentation* approach and enable efficient processing of output-intensive queries.

As discussed in Section 2.3, mpiBLAST originally adopted a centralized result-processing approach to merge the results that are generated by different workers. Specifically, in mpiBLAST, a worker produces *result alignments* after searching a query sequence against a database fragment. Each result alignment is a piece of the intermediate result, which describes a hit area identified from an in-database sequence. Information regarding an alignment, such as sequence IDs, E-values, and the locations of the hit are stored in a per-alignment data structure. Fig. 2(a)

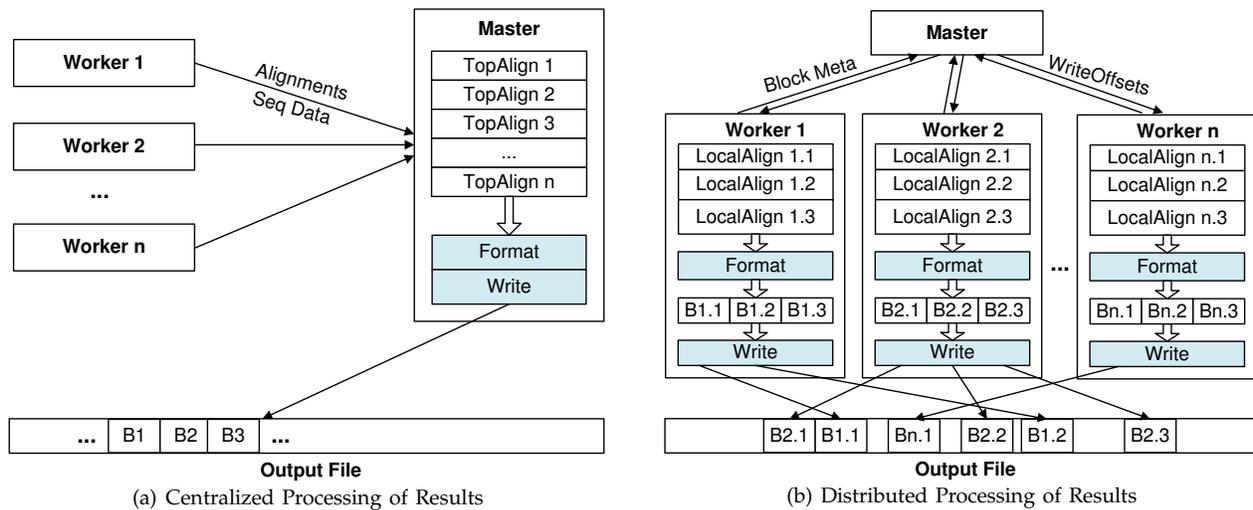


Fig. 2. A comparison of centralized versus distributed result processing. In the centralized design, the master serially formats and writes the globally top-ranked result alignments (e.g., TopAlign 1). In the distributed design, individual workers concurrently format locally top-ranked alignments (e.g., LocalAlign 1.1) into output blocks (e.g., B1.1). The workers then coordinate and write the output blocks to the file system with parallel I/O.

depicts the procedures of centralized result processing. When a search task is finished, the worker sends the result alignments together with the corresponding sequence data to the master. The result alignments belonging to the same query will be merged into a list in the order of their E-values, and the corresponding sequence data will be buffered and used later in the result formatting step. A query is ready for output when the result alignments of all database fragments have been received. The result data of multiple ready queries are processed and written in their submission order. To process a ready query, the master calls the output routine of NCBI BLAST, which in turn, formats each qualified alignment (in the top k range of the alignment list) and appends the corresponding result data block to the output file.

The above centralized design assumes that the result formatting and writing can be easily handled by a single compute node. This assumption, however, is not valid given the ever-growing scale of sequence databases, query workloads, and parallel computers. For example, researchers have found that searching individual “hard” queries against large DNA sequence databases could yield gigabytes of output data [5]. As a result, centralized result merging and formatting becomes the major scalability bottleneck in mpiBLAST. Fig. 3 shows the execution breakdown of searching 300 nr sequences against the database itself with mpiBLAST v1.4 on System X at Virginia Tech (configurations to be described in Section 4). The “search time” refers to the average time spent on the actual BLAST search algorithm by each worker. The “other time” includes all parallel overhead, which is dominated by the result processing cost at the scale of our experiments. As shown in Fig. 3, the search time decreases near-linearly as more workers are used, but the non-search overhead also increases rapidly. Consequently, the overall execution time stops decreasing even when at 32 workers.

Several reasons account for the poor scalability of centralized result processing. First, all result alignments need to be buffered at the master before output, imposing a high memory demand on this single node, causing a serious performance degradation, or simply forbidding the completion of certain output-intensive queries. Second, the result formatting and writing are performed sequentially, making the master a potential performance bottleneck in handling a bulky result volume. Finally, the result sequence data need to be sent over the network to the master for preparing output, adding high message-passing cost to the application-visible overhead.

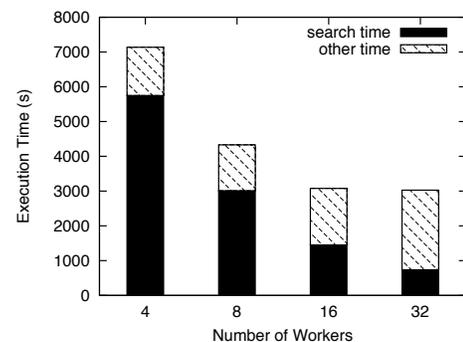


Fig. 3. Performance of mpiBLAST-v1.4 with centralized result processing design

To address these problems, mpiBLAST-PIO adopts a distributed result processing design to enhance its scalability in searching both individual and multiple queries on a large number of processors. Figure 2(b) illustrates this new output workflow. First, after generating local result alignments, workers take one more step to format the alignments into output blocks and store them in memory buffers. Each output block is stored with the E-value of the corresponding alignment. Next, workers submit *block*

metadata, which consists of the E-value and size of each local output block, to the master. The master then merges and filters output blocks of unqualified alignments according to their E-values. When the block metadata of all workers have been received for one query, the master calculates the in-file offsets of globally qualified output blocks and sends those back to the workers who buffered corresponding output blocks. Now knowing the subset of their qualified local output blocks, the workers write the output buffers that they already prepared to the file system. These write operations can be carried out in parallel using several strategies, to be discussed in Section 3.4. Note that the workers will proceed with searching new query sequences when the output offsets of a searched query sequence are not readily computed on the master. Thus, output processing on the master is overlapped with search computing on the workers without incurring visible synchronization cost.

The above distributed result processing greatly reduces the parallel overhead compared to the centralized scheme. First, it improves the level of parallelism by shifting the bulk of work in result formatting from the master to the workers and allows concurrent output preparation. Second, it alleviates the memory space bottleneck at the master node by having all workers collaboratively buffer intermediate results. Third, only a small amount of data (i.e., E-values, sizes and write offsets of output blocks) needs to be exchanged between the master and the workers, dramatically reducing the system communication volume.

It is worth noting that the E-value calculation is affected by the database size. As such, earlier versions of mpiBLAST generate inaccurate E-values because the size of a database fragment is smaller than that of the entire database. mpiBLAST-PIO solves this issue by broadcasting the global database information to all workers and having workers use those information to perform E-value calculation. By doing so, mpiBLAST-PIO can generate exactly the same results as NCBI BLAST does.

3.4 Parallel Output Scheduling

Our dynamic computation scheduling and distributed result processing leave each involved worker a set of noncontiguous output data blocks to write to disjoint ranges in the output file. Efficiently writing those output data to the file system is another challenge to sustaining high sequence-search throughput that needs to be overcome.

Optimization of noncontiguous I/O operations has been well-studied for parallel numerical simulations, which often possess predictable data access patterns and balanced computation models. However, in our situation, the unique aspects of parallel sequence-search applications complicate the I/O design in several ways:

- The output data distribution is fine-grained and irregular, varying from one query to another depending on its search results [5]. Straightforward, uncoordinated I/O can result in poor I/O performance.
- Unlike in timestep simulations, where the computation time is well-balanced across processes, here, computation time could be significantly imbalanced across workers searching the same query on different

database fragments. In addition, there is no inherent synchronization in the computation core between searching different queries. Synchronous parallel I/O techniques may incur high parallel overhead and have negative impacts on our load balancing algorithms.

The above observations suggest that traditional noncontiguous I/O optimization techniques, specifically data sieving and collective I/O (described in Section 2.5), may not be suitable for massively parallel sequence search. In this paper, we investigate an alternative I/O optimization that employs an asynchronous, two-phase writing technique. We compare it with existing parallel I/O optimizations by evaluating four output strategies: *WorkerIndividual*, *WorkerCollective*, *MasterMerge* and *WorkerMerge* (as illustrated side-by-side in Fig. 4). Among them, the first three are based on existing I/O techniques, and the last one (*WorkerMerge*) is based on our proposed I/O optimization.

3.4.1 *WorkerIndividual*

As described in Section 3.3, once the workers receive write offsets of buffered output blocks from the master, they can go ahead and issue write requests to the shared file system to write out the buffered output blocks. Fig. 4(a) depicts the procedure of the *WorkerIndividual* strategy with an example setting consisting of three workers, assuming the database is also segmented into three fragments. Whenever a worker finishes a search assignment, it checks with the master to receive offset information for previously completed queries. If such information arrives, the worker will first write local qualified output blocks to the shared file system before searching its next assignment. Note that as the result merging cannot be finalized until all workers complete searching the query sequence q_i , a worker likely will not be able to proceed with output right after it finishes searching this query. Instead of blocking this worker until the write offsets for q_i are released by the master, the scheduler lets the worker go ahead and request the next query sequence, q_{i+1} and start computation again.

The writing of noncontiguous output data can be done in two ways. The intuitive way is to perform a seek-and-write operation for every block via POSIX I/O calls. This is a slow solution as it will result in many small I/O requests, unfavored by typical file systems. An alternative way is to use the noncontiguous write method provided by MPI-IO [49]. Each worker first creates a file view that describes the locations to be written, then just calls `MPI_File_write()` to issue writes of all output data at once. MPI-IO libraries such as ROMIO [19] provide optimizations for this kind of noncontiguous write with data sieving [50]. In our experiments, MPI-IO calls will be used when data sieving is supported by the underlying file system, otherwise we fall back to seek-and-write with POSIX functions.

The major advantage of *WorkerIndividual* is that it does not introduce any synchronization overhead in the I/O phase. Because the output processing/writing of previous query sequences can be well overlapped with the computation for later query sequences, workers alternate between computation and I/O without wasting time waiting for other workers. This strategy is expected to work efficiently if the noncontiguous writing performance is well sustained

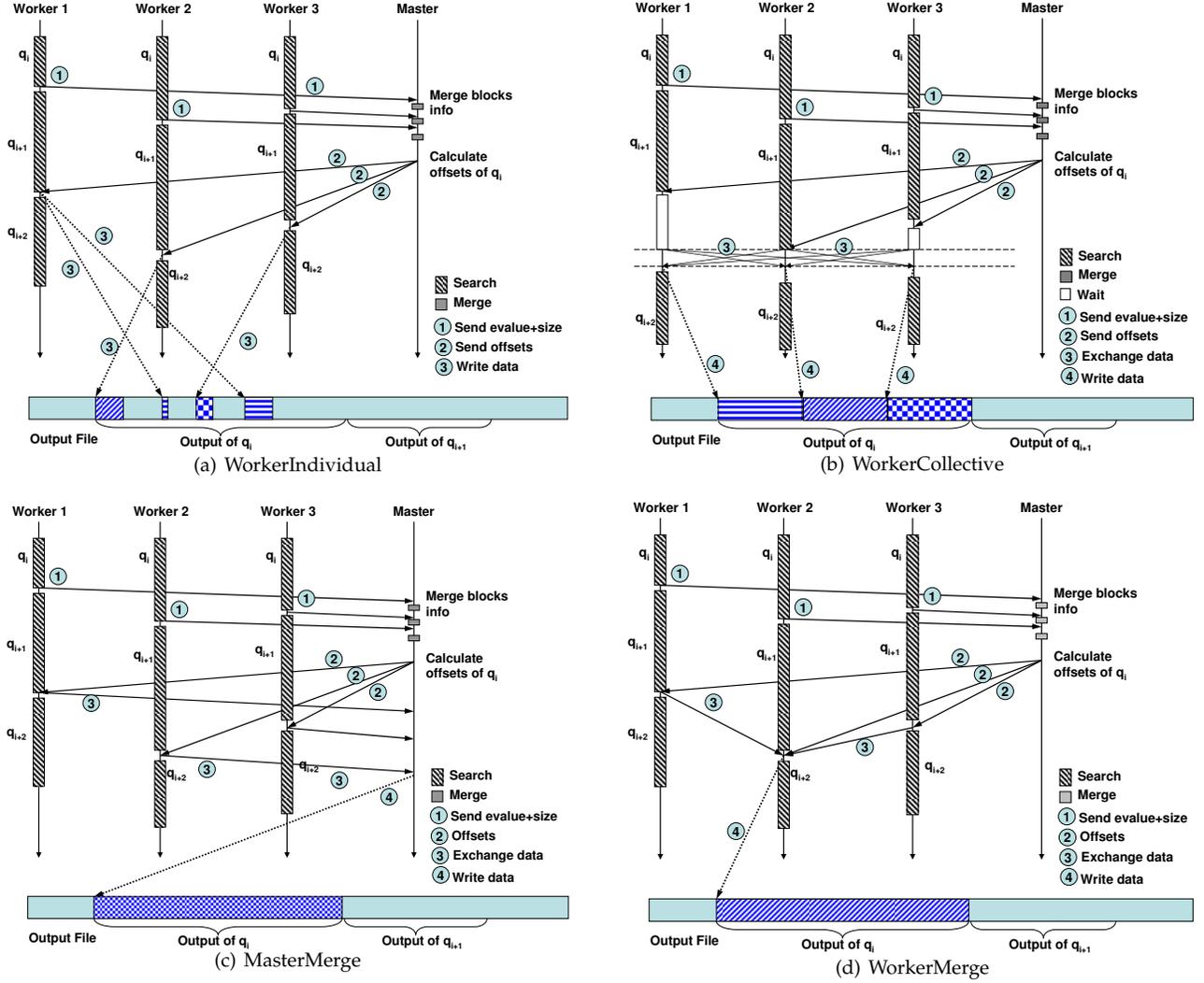


Fig. 4. Four output strategies compared in this study.

by the underlying file system. The disadvantage, however, is that the noncontiguous accesses may be inefficient. Even with data sieving, the concurrent irregular I/O requests from multiple workers generate much contention on the file system, leading to undesirable I/O performance.

3.4.2 WorkerCollective

Collective I/O appears to be a natural solution when we have a large number of small, noncontiguous I/O requests accessing a shared file with an interleaving pattern. A corresponding output strategy for parallel sequence search, which we call *WorkerCollective*, lets the workers coordinate their write efforts into larger requests. As illustrated in Fig. 4(b), after receiving write offsets from the master, rather than performing individual writes, the workers will issue an MPI-IO collective write request. Like in the case of *WorkerIndividual*, the result merging of q_i likely will not be done right after the search of this query is completed. To overlap the master's result processing with workers' searching, all the workers involved in searching q_i continue with query processing, until between assignments they find

that the file offset information regarding q_i has arrived. At this point, a worker will enter the collective output call for q_i . The advantage of this strategy lies in its better I/O performance compared to the noncontiguous write approach, by combining many small write requests into several large contiguous ones through extra data exchange over the network. However, even with the overlap discussed above, this strategy still incurs frequent synchronization, as collective I/O calls are essentially barriers that force workers to wait for each other (as shown with the white boxes in Fig. 4(b)). While very suitable for time-step simulations, this communication pattern is undesirable for parallel sequence-searches, which are known to have imbalanced computation.

3.4.3 MasterMerge

Another intuitive solution, especially considering the popular use of Network File System (NFS) servers on commodity clusters, is to let the master handle all the output. We call this the *MasterMerge*. With *MasterMerge*, the workers proceed as in the previous two schemes, until the result

merging outcome is communicated back to the workers. At this point, rather than writing qualified local output blocks to the shared file, the workers forward them to the master. The master then merges the output data in its memory and issues large, sequential write requests to the output file. Fig. 4(c) shows the output process using MasterMerge. This approach avoids concurrent I/O by many workers on a system with limited parallel I/O support and merges small I/O requests without enforcing synchronization on workers. However, the master can easily become a bottleneck on large-scale systems.

In implementing this strategy, the master’s memory constraint has to be taken into account. We defined a maximum write buffer size (*MBS*) in the master to coordinate incremental output communication, similar to the scheme used in common two-phase I/O implementations [18]. That is, only *MBS* amount of data will be collected and written at each operation.

3.4.4 WorkerMerge

Recognizing the limitations of the aforementioned approaches, we propose WorkerMerge, an output strategy that performs asynchronous, two-phase writes with merged I/O requests. With this strategy, after the master finishes result merging for query q_i , it appoints one of the workers to be the writer for this query. To minimize data communication, we select the worker with the largest volume of qualified output data to play the writer role, who will collect and write the entire output for this query. The workers involved in searching q_i are notified about the output data distribution and the writer assignment, and send their output data for q_i to the writer. In the example depicted in Fig. 4(d), worker 2 is selected as the writer for query q_i . After receiving output offsets, worker 1 and worker 3 send their output blocks to worker 2 using non-blocking MPI sends, then continue with the next search assignment. After worker 2 finishes searching query q_{i+1} , it receives output blocks sent by worker 1 and 3, then performs a contiguous write.

In our implementation, the same incremental communication strategy used in MasterMerge is adopted here to guard against buffer space shortage. Such data collection is conducted using non-blocking MPI communication to overlap with search computations. A writer checks the status of the collection between searching two assignments. Whenever the data is ready, it issues an individual write call to output a large chunk of data. One may be concerned that in the extreme case, all writes will be handled by a worker if the fragments on this worker always generate more output data than other fragments for all queries. This extreme case rarely happens in reality. In addition, even in that extreme case, the write duties will be shared by multiple workers in a partition that cache the same set of database fragments.

The WorkerMerge strategy takes advantage of collective I/O and removes the synchronization problem. Meanwhile, it resolves the bottleneck problem of MasterMerge by offloading output gathering and writing to workers. It seamlessly works with our dynamically load-balanced computation scheduling algorithm and allows a large number of workers to be efficiently supervised by a master.

One may argue that the MPI-IO standard does provide asynchronous collective I/O with *split collective read/write operations* [49]. The split collective operations allow the overlap of I/O and computation by separating a single blocking collective call into a pair of “begin” and “end” operations. However, our framework cannot benefit from them for two reasons. First, split collective I/O is not yet supported in popular MPI-IO libraries [19]. Second, in our target scenario, the data distribution (in terms of an MPI file view) is computed dynamically depending on the local result merging process, therefore a new file view needs to be constructed for each query’s output. Since the `MPI_File_set_view` call has only a blocking form, there is no way to remove inter-worker synchronization even with split collective write functions.

In our current design, the result from each query is written by one writer process. For queries that generate large amounts of output data, using multiple writers may be beneficial. Our work targets large BLAST jobs processing many queries on supercomputers. With a large number of concurrent groups working on queries and our proposed asynchronous writing, the underlying I/O parallelism in the system is expected to be well utilized. Therefore, the main issue here is whether the individual writers will have enough memory space to buffer the single-query output, which can be addressed by our incremental buffering and writing design.

4 PERFORMANCE EVALUATION

4.1 Experiment Setup

To evaluate the computation and I/O scheduling approaches presented earlier in this paper, we perform extensive experiments with mpiBLAST-PIO on three clusters with varying sizes, architectures, interconnection types, operating systems, and file systems, as described below.

Jacquard: Jacquard is a 356-node Opteron cluster located at National Energy Research Scientific Computing Center (NERSC). Each node has dual Opteron 2.2 GHz processors and 6 GB of physical memory. The nodes are interconnected with a high-speed InfiniBand network. Shared file storage is provided by the GPFS filesystem [51]. The MPI library is MVAPICH version 0.9.5-mlx1.0.1.

IA64: IA64 is a distributed/shared memory hybrid of commodity systems based on the Intel Itanium 2 processor. It is located at Ohio Supercomputer Center (OSC). The partition used in our experiments consists of 110 compute nodes, each has two 1.3 Gigahertz Intel Itanium 2 processors and 4 GB of physical memory. The nodes are interconnected with Myrinet and Gigabit Ethernet. Shared file storage is provided by a PVFS filesystem. The operating system is Linux and the MPI library is a version of MPICH optimized for the Myrinet high-speed interconnect.

System X: System X is a 1100-node Mac OS cluster located at Virginia Tech (VT). Each node consists of two 2.0-GHz IBM PowerPC 970 CPUs and 4 GB of physical memory. System X uses two interconnection fabrics, InfiniBand and Gigabit Ethernet. Shared file storage is provided by a ZFS [52] distributed filesystem. The MPI library is a customized version of MPICH 1.2.5.

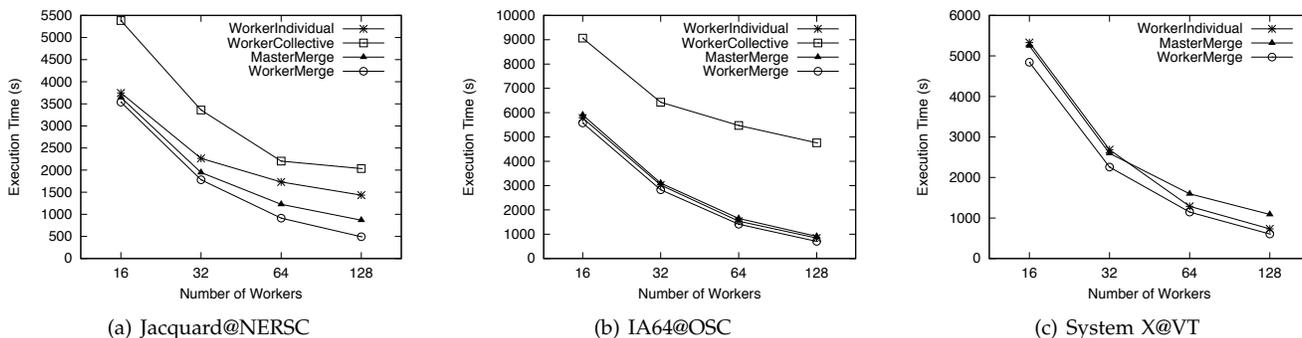


Fig. 5. Node scalability results of searching 1000 randomly sampled *nt* sequences on a different number of workers.

It is worth noticing that due to a special combination of the file system (ZFS) and the MPICH customization, MPI-IO (especially collective I/O) is not well supported on System X. Therefore for all experiments on System X, we only show results of the three output strategies other than *WorkerCollective*.

The experiment database is *nt*, a nucleotide sequence database that contains the GenBank, EMB L, D, and PDB sequences. At the time when our experiments are performed, the *nt* database contained more than 5 million sequences with a total raw size of about 20GB and a formatted size of about 6.5GB. To stress test the scalability of *mpiBLAST-PIO*, we use sequences randomly sampled from *nt* itself as queries because these queries are guaranteed to find close matches in the database.

In our experiments, *mpiBLAST-PIO* is configured to run in the sharing mode on IA64 and System X, where the database is pre-distributed to the local disks of compute nodes to save the database redistribution time in consecutive runs. The execution times reported on these systems do not include the database distribution time. On Jacquard, the program is configured to run in the non-sharing mode as this platform does not provide per node local storage for applications. The sequence database is distributed to the memory of all processors using the replication approach described in Section 3.1 at each run, and this overhead is included in the overall execution time.

4.2 Scalability Comparison of Output Strategies

In this section we evaluate the scalability of four output strategies discussed in Section 3.4 with regard to both system sizes and output sizes. The experiment query set consists of 1000 randomly sampled *nt* sequences sized 5KB or less³. The sequences within this length range account for 96% of overall sequences in the database. Our past experiences suggested that searching these sequences incurs high I/O demands. For all experiments in this section, we configured *mpiBLAST-PIO* to use only one partition. Here we are only looking at I/O scaling, in the next sections we will evaluate load balancing and the overall program scaling. The *nt* database is partitioned into 32 fragments.

3. *mpiBLAST* failed to complete this search job because the amount of data required to be buffered on the master node exceeded the node memory size.

These fragments are distributed on every 8 workers in a Round-Robin fashion. This configuration works well for the BLAST search jobs used in the experiments according to our experiences. All experiments are repeated three times and the average results are reported. The result variances are less than 5% in these experiments, hence the error bars are not included in the figures.

Fig. 5 shows the node scalability test results, where we plot the overall execution time of searching the given query set against *nt* as a function of the number of workers. We find that across all three platforms, the *WorkerMerge* approach works consistently the best. Overall, its winning margin increases as the number of workers grows. With 128 workers, it outperforms the *WorkerIndividual* strategy by an average factor of 1.8 over the three test systems, *WorkerCollective* by 5.4, and *MasterMerge* by 1.7. In addition, *WorkerMerge* achieves near-linear scaling from 16 to 128 workers on all three tested platforms. As expected, it outperforms other strategies by adopting distributed, merged I/O without enforcing additional synchronization that slows down query processing.

For the two systems that have collective I/O support, the *WorkerCollective* approach gives the worst performance. This is due to the synchronization cost associated with periodic collective I/O operations. Interestingly, the differences between the other three approaches look very different on the three platforms. Most notably, the *WorkerIndividual*, *MasterMerge*, and *WorkerMerge* strategies yield very similar performance on the IA64 system. One major reason is that the CPU frequency on this machine is relatively low (1.3GHz), while the interconnection bandwidth and the I/O bandwidth are high. Overall this results in a lower pressure on the output components, as results are generated rather slowly but consumed fast. On Jacquard, it is evident that *WorkerMerge* outperforms *MasterMerge*, and *MasterMerge* outperforms *WorkerIndividual*.

Next, we perform a set of output scalability tests. By default, NCBI BLAST reports the matches between a query sequence and the top 500 database sequences that are closed to the query based on the alignment results. This configuration is used in the previous group of tests. In the output scalability tests, we vary the output size by configuring BLAST to report the top 250, 500, 1000, and 2000 result database sequences. The corresponding total output sizes are 428MB, 768MB, 1.4GB, and 2.4GB, respectively. The

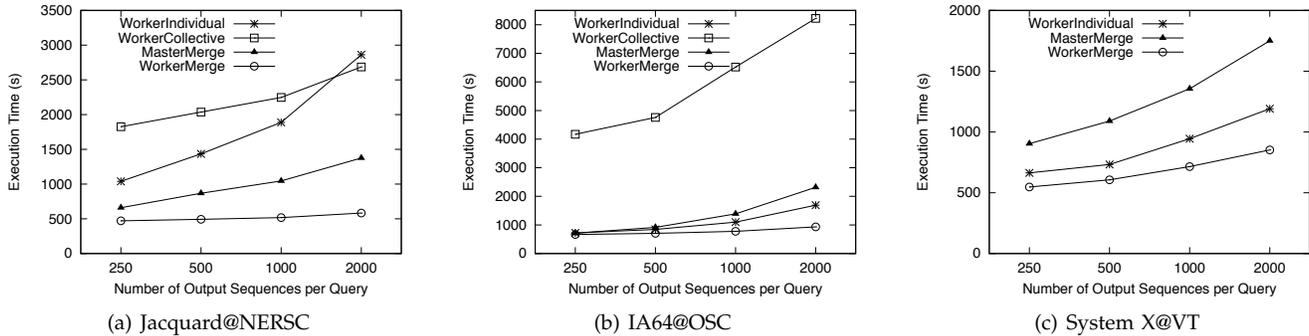


Fig. 6. Output scalability results of searching 1000 randomly sampled nt sequences with different amount of output data.

total number of workers used here is fixed at 128. The rationale behind the output scalability tests is that as parallel computers become more powerful and databases grow larger, the I/O-to-computation ratio of genomic sequence-searches is expected to increase in the near future. By varying the amount of output data, our tests arguably evaluate how well different output strategies accommodate the performance trend of future sequence-search jobs. In addition, our tests address users’ needs for gathering large amounts of results; according to the feedbacks provided in mpiBLAST users’ mailing list, it is not unusual that nowadays BLAST users choose to have several thousands of result sequences reported.

Fig. 6 shows the results of the output scalability tests, where we plot the overall execution time as a function of the number of reported result sequences. The advantage of WorkerMerge over the other strategies is more evident than in the node scalability tests. When reporting 2000 result sequences, WorkerMerge outperforms the second best strategy by a factor of 2.4, 1.8 and 1.3 on Jacquard, IA64 and System X respectively, and outperforms the worst strategy by a factor of 4.9, 8.8, and 2.1. In addition, the performance curves of WorkerMerge are much flatter than those of the other three strategies, suggesting that WorkerMerge is less sensitive to the growth of output sizes than the others.

Overall the relative performance differences between various strategies are similar to those in the node scalability tests. A new observation is that on Jacquard, the performance of WorkerIndividual degrades fast as more results are reported, causing a worse execution time than WorkerCollective at 2000 result database sequences. The scalability of WorkerIndividual is much better on IA64 and System X. One explanation is that the noncontiguous write approach used in WorkerIndividual is better supported by the file systems on the two platforms. In particular, PVFS (on IA64) provides special optimization for this write pattern with LIST I/O technique [21].

It is clear that for both types of scalability tests, WorkerMerge greatly outperforms the other strategies on all three tested platforms. Therefore, we will configure mpiBLAST-PIO to use WorkerMerge in the rest of the experiments.

4.3 Fine-grained Dynamic Load Balancing

In Section 3.2 we presented the details of our fine-grained, dynamic load-balancing algorithm. A key factor of the

design is to minimize the scheduling overhead by having masters proactively prefetch query segments from the supermaster. In this section, we evaluate the efficacy of query-segment prefetching with two synthesized workloads that have different balancing sensitivity to the task granularity.

The first workload uses a query set consisting of 1000 randomly sampled nt sequences sized 1KB or less. Our past experiences suggest that the search time distribution is relatively balanced for these sequences. For this workload, the load balance results are relatively insensitive to the task granules. For the second workload, we purposely synthesize a query set with a skewed search time distribution, where the load balancing results are highly sensitive to the task granules. We mixed expensive queries with inexpensive ones in terms of their search times as follows. According to our previous study [5], expensive query sequences in the nt database are likely to be larger than 5KB. With this hint, we first randomly sample 10,000 query sequences under 50KB from the nt database. These sequences are separated into two groups, with the first group consisting of sequences larger than 5KB and the second group consisting of the rest. Then we extract 10% of the most expensive sequences out of group one and put them together with 10% sequences randomly extracted from group two. This results in a 1000-sequence query set with expensive ones in the beginning.

Fig. 7(a) gives the Cumulative Distribution Function (CDF) plot of the processing time of each query sequence in the above two workloads when searched on 32 processors on System X. We label the first and the second workloads with “Balanced” and “Skewed,” respectively. As can be seen, 98.8% of query sequences in the first workload can be processed within 10 seconds, and the longest processing time is only 50 seconds. In the second workload, most of the query sequences (96.9%) can still be processed within 10 seconds. However, the rest of the sequences are much more expensive, with processing times ranging from 40 to 400 seconds.

We search the two workloads on System X using 166 processors configured into 5 partitions, with each consisting of 32 workers. The nt database is partitioned and pre-distributed in the same way as in the scalability tests. Each experiment is repeated three times and the average numbers are reported. Again, the result variance are quite small (less than 5%), so we do not include the error bars in

the result figures.

Fig. 7(b) shows the results of searching the balanced workload with and without query-segment prefetching on various task granules (i.e., query segment sizes). The overall execution time is reported as it measures the comprehensive performance impacts and matters the most to the end users. As expected, our prefetching design can significantly reduce the scheduling overhead when using small task granules compared to the non-prefetching design. Specifically, without prefetching of query segments, the overall execution time increases by a factor of 1.4 when the query segment size drops from 5 to 1. This is because when the task granule is small, there is not enough work in a query segment to balance loads across workers in the partition, causing worker idleness. In contrast, with prefetching, the overall execution time is about the same across different segment sizes. We further measure the worker idle time by subtracting the average time that the workers spend on doing useful work (i.e., searching, processing and writing results) from the overall execution time. As shown in Table 1, non-prefetching configurations incur significantly higher idle time than the prefetching counterparts for small segment sizes. In particular, the idle time of the non-prefetching configuration is more than 10-fold higher than that of the prefetching configuration for segment size 1.

The results of searching the skewed workload are quite different than those of the balanced workload, as shown in Fig. 7(c). For this heavy-headed query set, a large query segment size can cause significant imbalance in the processing times of individual segments. As a result, in general the overall execution time increases as the query segment size grows. However, without prefetching, using segment size 5 delivers better performance than using segment size 1. The reason is when the segment size is 1, the overhead caused by the worker idleness offsets the gain of fine-grained load balancing. The worker idle issue is greatly resolved with prefetching of query segments. Consequently, with prefetching, the system achieves the best performance at the smallest task granule (query segment size 1), where the advantage of fine-grained load balancing is fully taken. In particular, the best prefetching case (query segment size 1) outperforms the best non-prefetching case (query segment size 5) by a factor of 1.4.

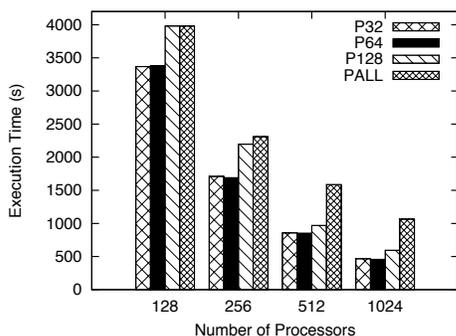


Fig. 8. Scalability of searching 5000 nt sequences with various partition sizes on System X.

4.4 Overall System Scalability

In order to evaluate the scalability of our integrated scheduling approach on massively parallel machines, in this experiment we benchmark mpiBLAST-PIO with up to 1024 processors on System X. We vary the system size to include from 128 to 1024 processors. To see how different partition sizes will affect the system throughput, for each system size, we perform multiple tests by configuring mpiBLAST-PIO to use 4 different partition sizes (in terms of number of workers): 32, 64, 128 and the system size (PALL). Note that when the partition size equals the system size (e.g., using 128 as the partition size on 128 processors), there is only one master process. The query set consists of 5000 randomly sampled nt sequences sized 5KB or less. The database is partitioned and distributed as other experiments presented previously. The prefetching query segment size is set to one.

Segment Size	1	5	10	20
Prefetch	26.27	20.37	33.79	45.95
NoPrefetch	279.15	99.58	71.64	54.54

TABLE 1

Worker idle time (measured in seconds) comparison for the balanced workload.

The performance results are shown in Fig. 8. The first observation is that the execution times decrease nicely for all configurations as the system size grows. Surprisingly, even with a single master (labeled PALL), mpiBLAST-PIO scales well to 1024 processors, which suggests that our inner-partition scheduling algorithm is highly efficient. However, the benefit of hierarchical scheduling on large system size is evident. With 512 processors and above, PALL is significantly slower than all other three configurations, simply because the master will become a performance bottleneck when managing too many workers.

The performance impacts of using different partition sizes are affected by a combination of several factors. On the one hand, using larger partition sizes can save the number of master processes and give more horse power to the actual search computation. On the other hand, larger partition sizes could overburden the master process with increasing loads of scheduling and output coordination, and consequently incur higher parallel overhead. Table 2 shows the measured processing time on the master node, including scheduling and output coordination for various partition sizes on 1024 processors. Clearly, the master load increases noticeably as the partition size grows. Interestingly, P32 and P64 deliver almost identical performance. This suggests that when partition size increases from 32 to 64, the saving in search computation time is counteracted by the parallel overhead increase. Nonetheless, mpiBLAST-

Partition Size	32	64	128	1024
Master Processing Time (s)	91.20	161.31	302.91	902.15

TABLE 2

Master processing time with different partition sizes.

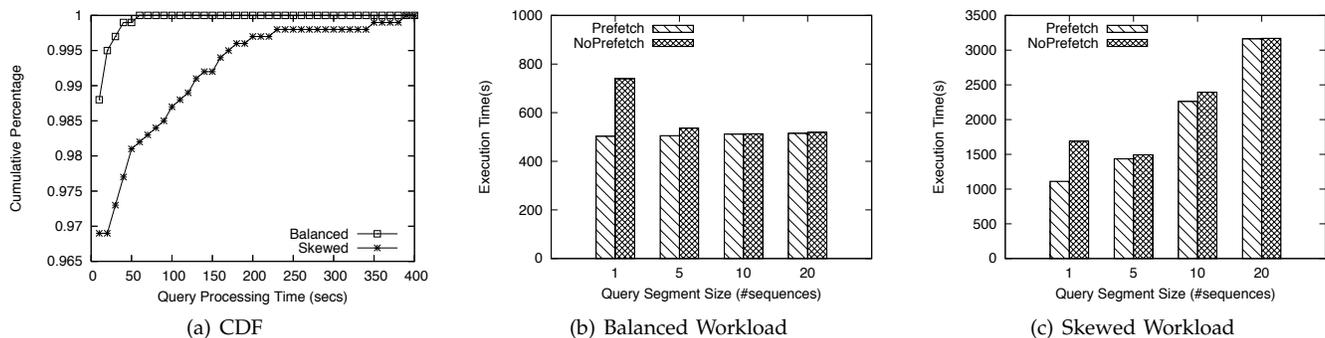


Fig. 7. Performance impacts of query segment prefetching.

PIO achieves almost linear speedup up to 1024 processors when the partition size is set to 64, with a parallel efficiency of 92% on 1024 processors as shown in Fig. 9.

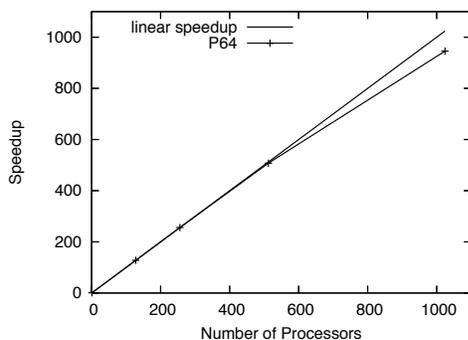


Fig. 9. Speedup of searching 5000 nt sequences on System X with hierarchical scheduling (partition size 64).

5 CONCLUSION

We consider large-scale genomic sequence-search as a class of parallel applications that possess highly irregular runtime behaviors in both computation and I/O. We identify that for this type of application, the incoordination between the I/O optimizations and the computation scheduling could result in serious performance degradations. Consequently, we propose an integrated scheduling approach that nicely coordinates dynamic computation load-balancing and high-performance non-contiguous I/O for maximizing the sequence search throughput. We implemented our optimizations on mpiBLAST and developed a research prototype named mpiBLAST-PIO. The experimental results on multiple platforms demonstrate that our integrated scheduling approach allows large-scale sequence search to efficiently scale on general parallel computers. In the future, we plan to generalize our study to other scientific applications with irregular computation and I/O patterns such as parallel HMMER [53] and large-scale protein family identification [54].

ACKNOWLEDGMENTS

This work is in part supported by the following funding sources: 1) DOE ECPI Award (DE-FG02-05ER25685); 2)

NSF CAREER Award (CNS-0546301); 3) Dr. Xiaosong Ma's joint appointment between NC State University and Oak Ridge National Laboratory; 4) Scientific Data Management Center (<https://sdm.lbl.gov/sdmcenter/>) under the DOE's Scientific Discovery through Advanced Computing Program; 5) Los Alamos National Laboratory contract W-7405-ENG-36. We are grateful to Virginia Tech Advanced Research Computing, Ohio Supercomputing Center and High Performance Computing Center at North Carolina State University for granting us access to their supercomputing resources. This research also used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. DOE under Contract No. DE-AC02-05CH11231. The authors thank Jeremy Archuleta and Tom Scogland for their constructive feedbacks on the manuscript.

REFERENCES

- [1] D. Benson, I. Karsch-Mizrachi, D. Lipman, J. Ostell, and D. Wheeler, "GenBank," *Nucleic Acids Res.*, vol. 36, January 2008.
- [2] J. Ostell, "Databases of Discovery," *ACM Queue*, vol. 3, no. 3, 2005.
- [3] *National Research Council, The New Science of Metagenomics: Revealing the Secrets of Our Microbial Planet.* National Academy of Sciences, 2007.
- [4] S. Schwartz, J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. Hardison, D. Haussler, and W. Miller, "Human-Mouse Alignments with BLASTZ," *Genome Res.*, vol. 13, 2003.
- [5] M. Gardner, W. Feng, J. Archuleta, H. Lin, and X. Ma, "Parallel Genomic Sequence-Searching on an Ad-Hoc Grid: Experiences, Lessons Learned, and Implications," in *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing*, 2006.
- [6] A. Ching, W. Feng, H. Lin, X. Ma, and A. Choudhary, "Exploring I/O Strategies for Parallel Sequence Database Search Tools with S3aSim," in *Proceedings of the International Symposium on High Performance Distributed Computing*, June 2006.
- [7] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology*, vol. 215, no. 3, 1990.
- [8] S. Altschul, T. Madden, A. Schffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman, "Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs," *Nucleic Acids Research*, vol. 25, no. 17, 1997.
- [9] M. Warren and J. Salmon, "A Parallel Hashed Oct-Tree N-body Algorithm," in *Supercomputing '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. New York, NY, USA: ACM, 1993.
- [10] J. Chen and V. Taylor, "Mesh Partitioning for Distributed Systems: Exploring Optimal Number of Partitions with Local and Remote Communication," in *PPSC*, 1999.

- [11] K. Schloegel, G. Karypis, and V. Kumar, "Dynamic Repartitioning of Adaptively Refined Meshes," in *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 1998.
- [12] A. Sohn and H. Simon, "S-HARP: A Scalable Parallel Dynamic Partitioner for Adaptive Mesh-based Computations," in *Proceedings of Supercomputing 98, Orlando, Florida, 1998*. [Online]. Available: citeseer.ist.psu.edu/article/sohn98sharp.html
- [13] S. Hummel, E. Schonberg, and L. Flynn, "Factoring: A Method for Scheduling Parallel Loops," *Commun. ACM*, vol. 35, no. 8, 1992.
- [14] S. Hummel, J. Schmidt, R. Uma, and J. Wein, "Load-Sharing in Heterogeneous Systems via Weighted Factoring," in *SPAA '96: Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*. New York, NY, USA: ACM, 1996.
- [15] I. Banicescu and S. Hummel, "Balancing Processor Loads and Exploiting Data Locality in N-body Simulations," in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. New York, NY, USA: ACM, 1995.
- [16] I. Banicescu and V. Velusamy, "Load Balancing Highly Irregular Computations with the Adaptive Factoring," in *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2002, p. 195.
- [17] I. Banicescu, V. Velusamy, and J. Devaprasad, "On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring," *Cluster Computing*, vol. 6, no. 3, 2003.
- [18] R. Thakur and A. Choudhary, "An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays," *Scientific Programming*, vol. 5, no. 4, 1996.
- [19] R. Thakur, W. Gropp, and E. Lusk, "On Implementing MPI-IO Portably and with High Performance," in *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, May 1999.
- [20] —, "Optimizing Noncontiguous Accesses in MPI-IO," *Parallel Computing*, vol. 28, no. 1, January 2002.
- [21] A. Ching, A. Choudhary, W. keng Liao, R. Ross, and W. Gropp, "Noncontiguous I/O through PVFS," in *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*. Washington, DC, USA: IEEE Computer Society, 2002.
- [22] A. Ching, A. Choudhary, K. Coloma, W. keng Liao, R. Ross, and W. Gropp, "Noncontiguous I/O Accesses Through MPI-IO," in *CCGRID '03: Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2003.
- [23] F. Isaila and W. Tichy, "View I/O: Improving the Performance of Non-Contiguous I/O," *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, Dec. 2003.
- [24] A. Darling, L. Carey, and W. Feng, "The Design, Implementation, and Evaluation of mpiBLAST," in *Proceedings of the ClusterWorld Conference and Expo, in conjunction with the 4th International Conference on Linux Clusters: The HPC Revolution*, 2003.
- [25] T. Smith and M. Waterman, "Identification of Common Molecular Subsequences," *J. Mol. Biol.*, vol. 147, pp. 195–197, 1981.
- [26] S. Needleman and C. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *J. Mol. Biol.*, vol. 48, 1970.
- [27] D. Lipman and W. Pearson, "Improved Tools for Biological Sequence Comparison," *Proc Natl Acad Sci*, vol. 85, no. 8, 1988.
- [28] R. Luthy and C. Hoover, "Hardware and Software Systems for Accelerating Common Bioinformatics Sequence Analysis Algorithms," *Biosilico*, vol. 2, no. 1, 2004.
- [29] C. White, R. Singh, P. Reintjes, J. Lampe, B. Erickson, W. Dettloff, V. Chi, and S. Altschul, "BioSCAN: A VLSI-Based System for Biosequence Analysis," in *ICCD '91: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*. Washington, DC, USA: IEEE Computer Society, 1991.
- [30] "Biocerator," <http://eta.embl-heidelberg.de:8000/>, 1994, CompuGen Ltd.
- [31] R. Braun, K. Pedretti, T. Casavant, T. Scheetz, C. Birkett, and C. Roberts, "Parallelization of local blast service on workstation clusters," *Future Gener. Comput. Syst.*, vol. 17, no. 6, 2001.
- [32] N. Camp, H. Cofer, and R. Gomperts, "High-throughput BLAST," http://www.sgi.com/industries/sciences/chembio/resources/papers/HTBlast/HT_Whitepaper.html.
- [33] E. Chi, E. Shoop, J. Carlis, E. Retzel, and J. Riedl, "Efficiency of Shared-Memory Multiprocessors for a Genetic Sequence Similarity Search Algorithm," Technical Report TR97-005, University of Minnesota, Computer Science Department, 1997.
- [34] R. Bjornson, A. Sherman, S. Weston, N. Willard, and J. Wing, "TurboBLAST(r): A Parallel Implementation of BLAST Built on the TurboHub," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.
- [35] D. Mathog, "Parallel BLAST on Split Databases," *Bioinformatics*, vol. 19, no. 14, 2003.
- [36] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova, "Efficient Data Access for Parallel BLAST," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*. Washington, DC, USA: IEEE Computer Society, 2005.
- [37] H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma, and W. Feng, "Massively Parallel Genomic Sequence Search on the Blue Gene/P Architecture," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08)*, 2008.
- [38] H. Rangwala, E. Lantz, R. Musselman, K. Pinnow, B. Smith, , and B. Wallenfelt, "Massively Parallel BLAST for the Blue Gene/L," in *High Availability and Performance Workshop*, 2005.
- [39] C. Oehmen and J. Nieplocha, "ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 8, 2006.
- [40] J. Nieplocha, R. Harrison, and R. Littlefield, "Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers," *The Journal of Supercomputing*, vol. 10, no. 2, 1996.
- [41] O. Thorsen, K. Jian, A. Peters, B. Smith, H. Lin, W. Feng, and C. Sosa, "Parallel Genomic Sequence-Search on a Massively Parallel System," in *ACM International Conference on Computing Frontiers*, 2007.
- [42] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [43] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain, "All-Pairs: An Abstraction for Data-Intensive Computing on Campus Grids," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 1, pp. 33–46, 2010.
- [44] A. Matsunaga, M. Tsugawa, and J. Fortes, "Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications," in *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 222–229.
- [45] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.
- [46] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kudipudi, "Passion: Optimized I/O for Parallel Applications," *IEEE Computer*, vol. 29, no. 6, June 1996.
- [47] J. May, *Parallel I/O for High Performance Computing*. Morgan Kaufmann Publishers, 2001.
- [48] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "Plfs: a checkpoint filesystem for parallel applications," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–12.
- [49] *MPI-2: Extensions to the Message-Passing Standard*, Message Passing Interface Forum, Jul. 1997.
- [50] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999.
- [51] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the First Conference on File and Storage Technologies*, 2002.
- [52] "ZFS at OpenSolaris.org," <http://www.opensolaris.org/os/community/zfs/>.
- [53] K. Jiang, O. Thorsen, A. Peters, B. Smith, and C. P. Sosa, "An Efficient Parallel Implementation of the Hidden Markov Methods for Genomic Sequence-Search on a Massively Parallel System," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, pp. 15–23, 2007.
- [54] C. Wu and A. Kalyanaraman, "An Efficient Parallel Approach for Identifying Protein Families in Large-scale Metagenomic Data Sets," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–10.