Massively Parallel Genomic Sequence Search on the Blue Gene/P Architecture

Heshan Lin*, Pavan Balaji†, Ruth Poole‡, Carlos Sosa§ Xiaosong Ma¶ and Wu-chun Feng‖

*Department of Computer Science, North Carolina State University
Email: hlin2@ncsu.edu

†Mathematics and Computer Science Division, Argonne National Laboratory
Email: balaji@mcs.anl.gov

‡IBM Systems & Technology Group, IBM
Email: rjpoole@us.ibm.com

§Biomedical Informatics and Computational Biology, University of Minnesota, Minneapolis, MN 55455
Blue Gene Software Development, IBM, Minneapolis, MN Email: cpsosa@us.ibm.com

¶Department of Computer Science, North Carolina State University
Computer Science and Mathematics Division, Oak Ridge National Laboratory
Email: ma@cs.ncsu.edu

‖Department of Computer Science, Virginia Tech
Email: feng@cs.vt.edu

***Abstract*—This paper presents our first experiences in mapping and optimizing genomic sequence search onto the massively parallel IBM Blue Gene/P (BG/P) platform. Specifically, we performed our work on mpiBLAST, a parallel sequence-search code that has been optimized on numerous supercomputing environments. In doing so, we identify several critical performance issues. Consequently, we propose and study different approaches for mapping sequence-search and parallel I/O tasks on such massively parallel architectures. We demonstrate that our optimizations can deliver nearly linear scaling (93% efficiency) on up to 32,768 cores of BG/P. In addition, we show that such scalability enables us to complete a large-scale bioinformatics problem — sequence searching a microbial genome database against itself to support the discovery of missing genes in genomes — in only a few hours on BG/P. Previously, this problem was viewed as computationally intractable in practice.**

I. Introduction

Genomic sequence search forms an important class of applications that are used widely and routinely in life sciences. Newly discovered sequences are commonly searched against a database of known nucleotide or amino acid sequences. Similarities between the new sequence and a sequence of known functions can help identify the functions of the new sequences and to find sibling species from a common ancestor. For example, in 2003, sequence searching helped biologists to identify the similarities between the recent SARS virus and the more well-studied coronaviruses [10], thus enhancing the biologists' ability to combat the new virus.

With the explosive growth of sequence databases, genomic sequence search has emerged as one of the most compute- and data-intensive applications in scientific computing. Parallel sequence search applications, such as mpiBLAST [7], [28], have previously proposed designs to achieve parallelism in both the required computation as well as the data I/O. While these designs work well for computer systems with hundreds to thousands of nodes and fast I/O subsystems, it is not clear whether such designs would still be effective on next-generation high-end supercomputers (e.g. BG/P) with *tens* of thousands of processors. For example, two unique problems with genomic sequence search that were not readily noticeable on small- and medium-sized clusters become apparent in massive petascale systems:

- The run times for different searches are highly irregular. Because fast sequence-alignment algorithms are heuristics-based, the amount of processing required as well as the results size of a given task are hard to predict [8]. Tasks containing the same amount of input data could have computation times that differ by orders of magnitude, thus creating huge idle periods for some processes, and consequently, resulting in under-utilization of resources. To further complicate the situation, the nodes available in the system are segregated into two types of processes—masters and workers, which possess widely different execution characteristics. Hence, keeping all processes busy, and thus, keeping all resources utilized, is a challenge.
- The I/O time for output can be large for systems with limited I/O capabilities such as Blue Gene/P. Thus, even if the computation required to search the queries is efficient, the output can easily become a bottleneck on such systems. Hence, efficiently processing this output data and writing it to a file system is critical to achieve

high performance for sequence-search applications.

In this paper, we propose two tightly coupled optimizations to mpiBLAST: (i) fine-grained and dynamically load-balanced task scheduling and (ii) scalable, asynchronous output processing. With fine-grained and dynamically load-balanced task scheduling, we allow flexible placement of master and worker processes to balance their computation loads. In addition, different worker processes can dynamically re-group and thus avoid long idle periods caused by the large difference in the computational times for different tasks. With asynchronous output processing, we obtain the performance benefit of parallel I/O without the synchronization overhead imposed by traditional collective I/O techniques.

Together with detailed descriptions of the above designs, we present extensive evaluation of these designs on large-scale BG/P systems. Our experimental results demonstrate that these two enhancements allow the application to scale almost linearly (93% efficiency) to 32768 cores of BG/P. We also show that those optimizations allow us to tackle a real computational biology problems, i.e., sequence searching a microbial genome database against itself to support the discovery of missing genes in genomes. This problem, previously viewed as computationally intractable, completed in only a few hours on BG/P.

The rest of the paper is organized as follows. Section II provides a brief overview of mpiBLAST and the BG/P architecture. Section III surveys related work. In Section IV, we present the details of our proposed task mapping and I/O optimizations. Section V discusses the performance evaluation of the proposed optimizations on the BG/P architecture. Finally, we present our conclusions in Section VI.

II. Background

In this section, we provide a brief background about the mpiBLAST sequence-search application (in Section II-A) and the BG/P architecture (in Section II-B).

A. mpiBLAST

The original software architecture of mpiBLAST [7] follows a master-worker design. Our previous work enhanced it with a hierarchical design [28], which organizes all processes into equal-sized *partitions* and has a *supermaster* process dedicated to supervise all the partitions. The supermaster is responsible for assigning tasks to different partitions and handling inter-partition load balancing. Within each partition, one process is designated as the master, and the other processes serve as workers. During the execution, first a copy of the database is replicated to each partition. After the database replication is done, the master process in a partition periodically fetches a batch of query sequences from the supermaster and assigns them to the workers within its partition. After all the queries in the current query batch have been searched and the corresponding output has been written by workers, the master starts to fetch the next query batch from the supermaster. The same procedures repeat until all input queries are completed.

mpiBLAST employs a distributed result-processing scheme, which originated from pioBLAST [9], to improve its output performance. Specifically, after a worker finishes searching a query against a database fragment, it prepares and stores the local results in memory buffers and only sends the metadata information needed for result merging (e.g., the size and the similarity score of each result element) to the master process. Once the master receives information for all results for a query, it calculates the output offsets of qualified results and sends the offsets back to the corresponding workers. The workers then call MPI-IO functions to write locally buffered, non-contiguous data to the file system in parallel.

Although the current mpiBLAST design works well on hundreds to thousands of processors with fast I/O subsystems, it has several disadvantages when the program is deployed on petascale machines with tens of thousands of processors and relatively limited I/O capability. First, after a master fetches a query batch, it has to wait until all workers finish the corresponding work before fetching the next query batch, thereby wasting compute power on workers that finish their assignments earlier. Second, each master can only manage one copy of the database. This prevents the system from mapping an arbitrary number of workers to a master, which is important to designing task scheduling algorithms that can maximize the resource utilization. Third, the existing parallel I/O approach is not scalable when processing jobs with large output on systems like BG/P, which will be explained in Section V-B.

B. Overview of the Blue Gene/P Architecture

The Blue Gene/P architecture supports a distributed memory, message-passing programming model [17]. It uses system-on-a-chip (SoC) technology to deliver four 850-MHz PowerPC 450 processors, capable of achieving a theoretical peak performance of 13.6 gigaflops/chip [22]. Each such SoC constitutes a Compute Node. A Compute Node attached to a processor card with 2 GB of memory creates the compute and I/O cards. Two rows of 16 compute cards then make up a node card. Next, a midplane consists of 16 node cards stacked in a rack. A rack holds two midplanes for a total of 32 node cards.

The PowerPC 450 core itself contains the first-level (L1) cache, which is 64-way set associative. The second level (L2R and L2W) of caches, one dedicated per core, are 2 KB in size. They are fully associative and coherent; they act as prefetch and write-back buffers for L1 data. The L2 cache line is 128 bytes in size. Each L2 cache has one connection toward the L1 instruction cache running at full processor frequency. Each L2 cache also has two connections toward the L1 data cache, one for the writes and one for the loads, each running at full processor frequency. The third-level (L3) cache is 8-way set associative and 8 MB in size with 128-byte lines. Both banks can be accessed by all processor cores. The L3 cache has three write queues and three read queues: one for each processor core and one for the 10-Gigabit network.

There can be up to two I/O cards per node card. When these nodes do not have a local file system, I/O operations need to be sent to an external device. In order to reach this external

device (outside the environment), a *compute node* sends data to an I/O node, which in turn carries out the I/O requests [22]. In the BG/P systems used in our study, the file system are configured with the Global Parallel File System (GPFS) [19], [17].

Applications on Blue Gene/P may run in three different modes: Symmetrical MultiProcessing (SMP) Node mode, Virtual Node mode (VN), and Dual Node mode (DUAL). In the first mode, each compute node executes a single task with a maximum of four threads. Node resources (primarily the memory and the torus network) are shared by all threads. In VN mode, four single-threaded tasks are run on each node, one task per core. Each task gets 1/4 of the total memory of the node. Finally, in the DUAL mode, two tasks can be run on a node. Each task gets half of the memory and cores and can consist of at most two threads.

III. Related Work

A. Parallel Genomic Sequence Search

Early parallel sequence-search software adopted the *query segmentation* approach [3], [4], [5], where a sequence-search job is parallelized by having individual compute nodes independently 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 will suffer high paging overhead when searching databases much larger than the memory of a compute node, because the database needs to be repeatedly scanned when searching multiple queries. This issue motivated *database segmentation* [2], [7], [11], [9], where the sequence database is partitioned and distributed across compute nodes. By fitting large databases into the aggregate memory of multiple compute nodes, database segmentation eliminates the paging issue and allows timely sequence analysis to keep up with fast growing database sizes. Nonetheless, this approach introduces computation dependency between individual nodes because the distributed results generated at different nodes need to be merged to produce final output. The parallel overhead caused by results merging will increase as the system size grows, consequently limiting the program's scalability on large-scale deployments.

Recent efforts in designing large-scale sequence-search applications achieved high scalability by adopting a combination of both segmentation approaches. Rangwala et. al. developed a parallel BLAST implementation optimized for Blue Gene/L [18]. Oehmen et. al. reported ScalaBLAST [16], a highly efficient parallel BLAST built on top of the Global Array [14] 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 employs static load balancing approaches, assuming the execution time of a BLAST search task is predictable from the 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 same amounts of "work units". The work units of a query batch is calculated based on a "*trial-and-error*" model that takes into account both the total number of characters and the number of queries in a batch. Although such a static load-balancing design scales well by avoiding centralized task scheduling, our recent study discovered that the execution time of a sequence search task is often hard to predict based on the simple metrics of its input data [8]. We also found that searching different jobs with same amounts of input data could yield in execution time differing by *orders of magnitude*. These findings led us to believe that in order to address a broad class of sequence-search jobs, dynamically load balancing is imperative to designing massively parallel sequence-search applications.

B. Optimization of Non-contiguous I/O

In many parallel scientific applications, processes need to access data files in a non-contiguous manner [6], [15], [20], [21], [27]. There are two techniques widely adopted to optimize non-contiguous I/O performance used in popular parallel I/O libraries such as ROMIO [26].

The first technique is *data sieving*, introduced in the PASSION I/O library [24], which targets non-contiguous I/O requests issued from one process. It replaces the original small, non-contiguous I/O requests with larger ones, with additional in-memory data manipulation to pick out portions of data specified in the original read request(s), or to update portions of data specified in the original write request(s). Since these operations are much faster than disk seeks, data sieving can considerably improve non-contiguous I/O performance at the cost of accessing extra amounts of data. However, when many processes access shared files with fine-grained and interleaved write patterns, such as the output of parallel sequence-search applications, data sieving incurs too much extra data access and yields unsatisfactory I/O performance.

The second technique, *collective I/O*, was designed to improve parallel non-contiguous I/O performance by having multiple processes coordinate their operations to combine small, non-contiguous I/O requests into large, sequential ones [12], [23], [26]. The most popular collective I/O strategy used today is *two-phase I/O* [23]. 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. While collective I/O has been widely used in data-intensive parallel numerical simulations, it could incur high synchronization cost when computational phases of a parallel program are not balanced.

IV. Scaling mpiBLAST to Massively Parallel Systems

As described in Section I, while existing designs of mpiBLAST work well for small-to-medium scaled supercomputers with fast I/O subsystems, several issues need to be addressed in order to scale to massively parallel systems. In this section, we discuss two approaches that we propose to allow mpiBLAST to scale on such systems.

A. Fine-grained and Dynamically Load-balanced Task Scheduling

As discussed in Section I, the existing design of mpiBLAST falls short in two aspects with respect to maximizing resource utilization. First, the irregular run times of different workers can result in large idle times for worker processes. Second, the amount of work performed by the masters and workers is unequal, and the current design is not flexible enough to handle arbitrary mappings of number of workers to each master without losing performance. In this section, we present various designs that allow us to efficiently handle these issues.

1) Fine-grained Data Management for Flexible Master-Worker Ratios: mpiBLAST adopts a hierarchical approach to handle the input of query data. In this approach, the number of masters to workers is not entirely flexible. For example, for a database of size *S*, fragmenting the database into *F* fragments might result in the best performance. This means that each inner-partition must contain *F* workers and one master process. Given the vastly different work roles of the masters and workers, such stringent restrictions on the number of workers assigned to each master can lead to idleness for either the master (if the work the master performs is much less than what the workers perform) or the workers (if the work the workers perform is much less than what the master performs).

To handle such issues, we present a fine-grained data management approach that allows flexibility to assign an arbitrary ratio of the number of masters to the number of workers. Specifically, in this approach, each inner-partition will still have exactly one master but an arbitrary number of workers. To avoid performance degradation, database fragmentation is still performed in the ideal fragment size, *F*, as described above. However, now each inner-partition can have multiple copies of the entire database. Thus, each master can handle multiple worker groups simultaneously, allowing for a more fine-grained and flexible ratio for the total number of masters in the system to the total number of workers. Note also that a worker can store multiple database fragments.

2) Efficient Load Balancing Using Dynamic Worker Group Management: BLAST search time has been found highly variable and unpredictable [8]. To the best of our knowledge, there is no effective way to estimate the execution time of a given BLAST search in the existing literature. Without a priori knowledge of queries' processing time, a greedy algorithm that assigns fine-grained tasks to idle processes appear to be a sufficient solution to load balancing. The challenge then becomes reducing the scheduling overhead that may be exacerbated when using small task granules, at both inter- and intra-partition levels.

At the inter-partition level, a natural load balancing design would be having the master request a query segment from the supermaster whenever a partition is idle. After all query sequences in the segment are finished, the master then requests another query segment from the supermaster. However, doing so may hurt the inner-partition load balance, as the workers that finish their assignments faster will have to wait the slower workers to finish their own assignments, after which the master will fetch another query segment. This inner-partition imbalance will worsen as the *query segment size* (defined as the number of query sequences in the segment) decreases below a certain level, where there is insufficient work to be balanced across all workers.

To handle these issues we propose a dynamic worker group management approach where the master prefetches several queries from the supermaster. The idea is to have masters maintain a window of outstanding work; whenever a worker is done with its current assignment, it requests the master for next assignment. The set of workers that work on one particular query is dynamically created as different workers become available, consequently minimizing the idle time caused by workers' waiting for new query segment to be fetched.

The scheduling process running on the master is described in Algorithm 1. At the start, the master constructs a list of query sequences, QL, sorted by their ids that are being processed in the partition. A query sequence q_i in QL is corresponding to $|F|$ individual tasks, with each task searching q_i against a distinct database fragment. A query q_i is completed once all of its tasks have been completed. The master sends a prefetching request for more query segments to the supermaster when it observes that the number of total uncompleted tasks of the query sequences in QL is less than the number of workers. To overlap network communications with local job scheduling, the master receives the query segment in the background with nonblocking MPI calls. In doing so, the inner-partition imbalance is alleviated as a idle worker does not need to wait other workers to finish their assignments before getting to tasks of a new query segment.

At the inner-partition level, our scheduling algorithm avoids synchronization between workers by enabling flexible task sharing between workers in different replication groups. As described in Algorithm 1, in mpiBLAST workers report to the master to request a new task once it finishes the current task. Upon receiving a task request from a worker (w_j), the master scans QL in FIFO order to look for an uncompleted task. For the current query q_c being examined, if the worker has cached some database fragments that needed by q_c's uncompleted tasks, the task that contains the cached fragment with smallest id will be assigned to the worker. If no tasks can be found for this worker, the scheduling algorithm will search $q_c + 1$ in QL for a task. The scheduling process repeated until all queries are completed.

B. Parallel Output Strategies

Our cross- and inner-partition dynamic task scheduling and query prefetching greatly improves the search throughput for large sequence search jobs. These techniques allow the workers to proceed without being delayed by synchronization. (e.g., waiting for new queries to be assigned or for peers processing the same query to finish). As a tradeoff, such fully dynamic and asynchronous query processing does bring a new challenge, especially on petascale machines with tens of thousands of cores: concurrent output to the shared result file.

Algorithm 1 Master Scheduling Algorithm

Let $QL = \{q_1, q_2, ...\}$ be the list of unfinished query sequences
Let $F = \{f_1, f_2, ...\}$ be the set of database fragments
Let $Unassigned_i \subseteq F$ be the set of unassigned database fragments for query sequence q_i
Let $W = \{w_1, w_2, ...\}$ 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
Receive database fragments distribution from workers
Require: $|W| \neq 0$
Initialize QL by fetching a query segment from supermaster
while QL is not empty **do**
  if number of all unassigned fragments in $QL < |W|$ **then**
    Issue segment prefetching request to supermaster
  end if
  if Receive 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
  $q_c \leftarrow QL.head$
  $assignment_j \leftarrow \ < \varnothing, 0 >$
  while $q_c \neq QL.tail$ and $assignment_j = \ < \varnothing, 0 >$ **do**
    if $\exists f_i \in Unassigned_c$ and $w_j \in D_i$ **then**
      Find f_k such that $k = min\{l | f_l \in C_j \wedge f_l \in Unassigned_c\}$
      $assignment_j \leftarrow \ < q_c, f_k >$
    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 this section, we discuss efficient parallel I/O techniques suitable for the sequence search workload on large-scale platforms.

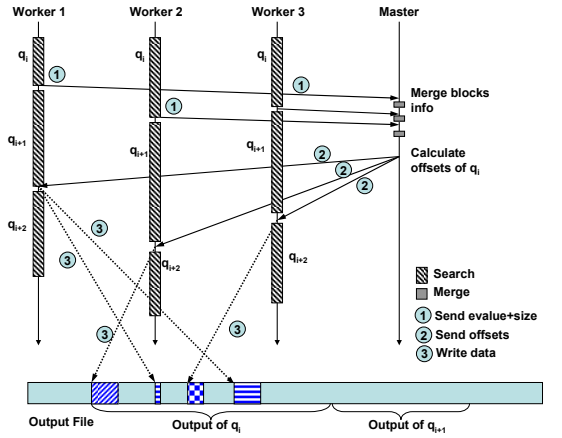
The optimizations of non-contiguous I/O operations have been well studied for parallel numerical simulations, which often possess predictable data access patterns and balanced computation models. However, the unique aspects of parallel sequence-search applications complicate the I/O design as follows:

- The output data distribution is fine-grained as well as irregular, and varies from one query to another. 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 the 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. Therefore, using synchronous parallel I/O techniques may incur high parallel overhead and have negative impacts on load balancing.

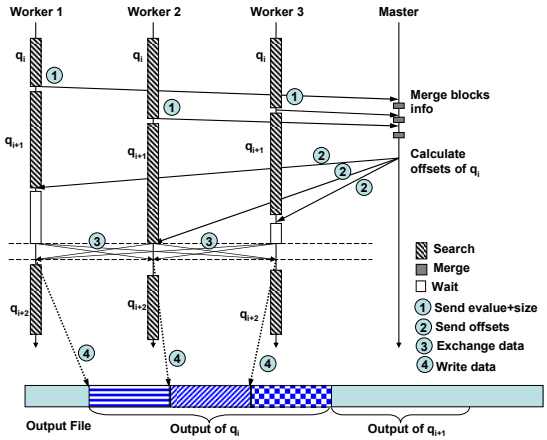
The above observations suggest that traditional non-contiguous I/O optimization techniques, specifically data sieving and collective I/O (described in Section III-B), may not be suitable for massively parallel sequence search. In this paper, we investigate an alternative I/O optimization for parallel sequence search which employs an asynchronous, two-phase writing technique. We compares it with with existing parallel I/O optimizations by evaluating three alternative output strategies on the BG/P system: *WorkerIndividual*, *WorkerCollective*, and *WorkerMerge* (as illustrated side by side in Fig. 1). Among them, the first two are based on existing I/O techniques, and the last one (WorkerMerge) is our proposed I/O optimization. Recall that we presented the results processing protocol in Section II-A. In this section, we focus our discussion on how the non-contiguous data buffered at workers are written to the file system.

1) WorkerIndividual: This I/O approach has been used in the previous mpiBLAST design. Once the workers receive write offsets of buffered output blocks from the master, they go ahead and issue write requests to the shared file system to write out the buffered output blocks. Fig. 1(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 is available, the worker will first write local qualified output blocks to the shared file system before searching its next assignment. Note that as the results 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 immediately after it finishes searching the query. Instead of blocking this worker until the write offsets for q_i are available, the scheduler let it go ahead to request the next query sequence, q_{i+1} and start computation again.

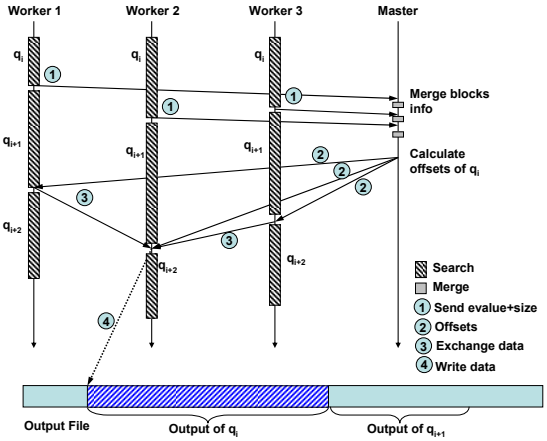
The writing of non-contiguous output data can be done in two ways. The intuitive way is to perform a seek-and-write operation for every block via POSIX I/O calls. This solution is inefficient as it will result in many small I/O requests unfavored by typical file systems. An alternative approach is to use the non-contiguous write method provided by MPI-IO [13]. Specifically, each worker first creates a file view that describes the locations to be written, then simply calls `MPI_File_write()` to issue writes of all output data at once. MPI-IO libraries such as ROMIO [26] provide optimizations for this type of non-contiguous write with data sieving [25] according to the file access pattern. Our experiments on BG/P show that using MPI-IO non-contiguous write has considerable better performance than



(a) WorkerIndividual



(b) WorkerCollective



(c) WorkerMerge

Fig. 1. Three output strategies compared in this study. WorkerIndividual adopts the data sieving I/O technique when possible. WorkerCollective is based on the collective I/O technique. WorkerMerge is based on the proposed asynchronous, two-phase I/O technique.

using individual POSIX I/O calls. Hence in our experiments we use MPI-IO calls for this output strategy.

The major advantage of WorkerIndividual is that it does not introduce any synchronization overhead in the I/O phase. Workers alternate between computation and I/O, without blocking on other workers. This strategy is expected to work efficiently if the non-contiguous writing performance is well sustained by the underlying file system. The disadvantage, however, is that the non-contiguous accesses may be inefficient.

2) WorkerCollective: Collective I/O appears to be a natural solution when we have a large number of small, non-contiguous 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. 1(b), after receiving write offsets from the master, rather than performing individual writes, the workers will issue a MPI-IO collective write request. Like in the case of WorkerIndividual, the results 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 found 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 non-contiguous 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. 1(b)). While suitable for time-step simulations, this communication pattern is undesirable for parallel sequence-searches, which have imbalanced computation intervals.

3) WorkerMerge: Recognizing the limitations of the aforementioned approaches, we propose WorkerMerge, an output strategy that performs asynchronous, decentralized 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. 1(c), worker 2 is selected as the merger 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 implementing this strategy, the memory constraint of involved processes has to be taken into account. We defined a maximum write buffer size (MBS), to coordinate incremental output communication, similar to the scheme used in common 2PIO implementations [23]. Specifically, a write leader will only gather MBS amount of data from the peer workers

before issuing a write to the file system. The MBS value is set to 4MB in our current implementation.

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 output data for a write operation has been collected, it issues an individual write call to output a large chunk of data.

The WorkerMerge strategy takes advantage of collective I/O and removes the synchronization problem. It seamlessly works with our dynamically load-balanced 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* [13]. The split collective operations do 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 [26]. 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. However, this will inevitably add synchronization overhead in the I/O process, which is what our design tries to avoid. Our work targets large BLAST jobs processing many queries on supercomputers. Here, 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 is whether the individual writers will have enough memory space to buffer the single-query output, which can be addressed by the incremental buffering and writing strategies already adopted in parallel I/O libraries such as ROMIO.

V. Performance Results

In this section we present the performance evaluation of our optimizations of mpiBLAST on BG/P systems. We first compare the scalability of different output strategies, then we evaluate the overall system scalability with a real bioinformatic genome search problem.

A. Experiment Platforms

We use several Blue Gene/P systems located at Argonne National Laboratory (ANL) and IBM Rochester with different compute-to-I/O node ratios. At ANL, we use a one-rack BG/P system named Surveyor, which has a compute-to-I/O node ratio of 64:1. At IBM Rochester, we use a 8-rack system and a one-rack system, configured with compute-to-I/O node ratios of 64:1 and 128:1 respectively. In all systems, each node is equipped with 2GB memory, and GPFS is provided as a shared file system.

B. Comparison of Output Strategies

In this section, we evaluate the scalability of three I/O strategies discussed in Section IV-B. To stress test the program, we use `nt`, one of the largest nucleotide sequence databases downloaded from NCBI, as the experimental database. The `nt` database contains in total over 6 million GenBank, EMB L, D, and PDB sequences, with a raw size of 23 GB and a formatted size of 7.4GB at the time the experiments were conducted. We use sequences randomly sampled from the `nt` database itself as the query sequences.

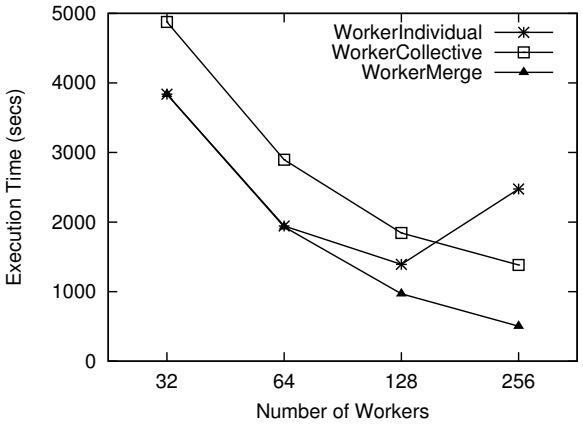
First, we evaluate the three output strategies by configuring mpiBLAST to use only one partition. In doing so, we focus solely on I/O scalability and isolate other factors such as load balancing between partitions. The `nt` database is partitioned into 128 fragments. At the beginning of the program execution, a database copy is replicated across sets of 32 workers. That is, every 32 workers in the whole system form a replication group. The first group of workers serves as the I/O group. Each worker in the I/O group reads in their assigned fragments in parallel and broadcasts those fragments to the corresponding workers in all other replication groups using MPI communication.

The query set is 512 randomly sampled `nt` sequences whose sizes are less than 5KB each. Our past experiences suggest that searching these sequences is not very compute-intensive, and consequently incurs high I/O demands. Fig. 2 shows the execution time for searching the queries with an increasing number of worker processes on two BG/P systems with different I/O-to-compute node ratios. We observe that in the two tested systems, both WorkerCollective and WorkerMerge approaches scale well as the number of workers grows, with WorkerMerge consistently outperforming the WorkerCollective by a considerable margin. This suggests that the I/O approaches used in both output strategies are scalable, and the their write throughput can be sustained by the I/O subsystems on either system. However, the performance of WorkerCollective is significantly affected by the synchronization overhead associated with periodic collective I/O operations.

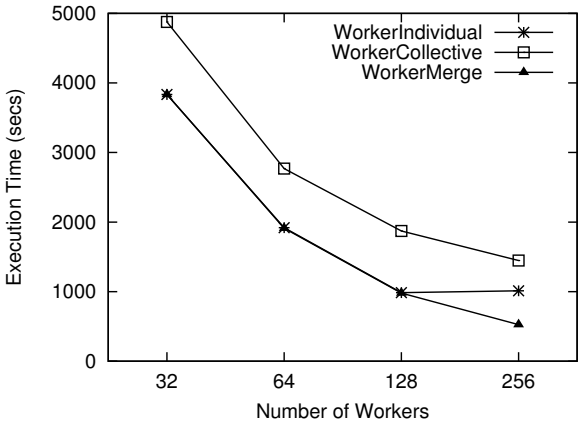
On the other hand, WorkerIndividual works well only with small numbers of processes. In fact, the performance of WorkerIndividual is almost the same as that of WorkerMerge in small scale tests. However, because of the non-scalable I/O approach, the performance of WorkerIndividual degrades rapidly as system size grows. Interestingly, the trends of the degradation are different on the two tested systems. On the IBM Rochester system, which has an compute-to-I/O node ratio of 128:1, WorkerIndividual starts performing worse than WorkerMerge at 128 workers, but still outperforms WorkerCollective. From 128 to 256 workers, the performance of WorkerIndividual degrades so much that there is a negative scaling impact on the overall performance. On the ANL system, which has better I/O capability with an compute-to-

Profile	128 Workers - IBM Rochester						128 Workers - ANL					
Strategies	WorkerIndividual		WorkerCollective		WorkerMerge		WorkerIndividual		WorkerCollective		WorkerMerge	
	Time(s)	Percent	Time(s)	Percent	Time(s)	Percent	Time(s)	Percent	Time(s)	Percent	Time(s)	Percent
Init	14.83	1.06%	14.82	0.81	16.91	1.75%	29.41	2.99%	21.87	1.16%	28.21	2.88%
BLAST	939.61	67.50%	946.93	51.37	941.33	97.22%	937.89	95.22%	943.56	50.39%	937.66	95.78%
Write	397.03	28.52%	793.09	43.03%	0.24	0.23%	4.29	0.44%	828.72	44.27%	0.13	0.01%
Other	40.65	2.92%	88.39	4.80%	9.73	1.01%	13.32	1.35%	78.21	4.18%	13.01	1.33%
Total	1392.12	100%	1843.27	100%	968.22	100%	984.93	100%	1872.36	100%	979.02	100%

TABLE I
EXECUTION BREAKDOWNS OF VARIOUS OUTPUT STRATEGIES ON TWO SYSTEMS WITH DIFFERENT I/O TO COMPUTE NODE RATIOS.



(a) Rochester 128:1



(b) ANL 64:1

Fig. 2. Node scalability results of searching 512 randomly sampled `nt` sequences against `nt` database itself with increasing number of worker processes. The compute-to-I/O node ratio is 128:1 in the Rochester system and 64:1 in the ANL system.

I/O node ratio of 64:1, WorkerIndividual works well until 128 workers, and increases slightly at 256 workers.

Nonetheless, the WorkerMerge strategy consistently outperforms other strategies on both systems. With 256 workers, WorkerMerge outperforms the WorkerCollective by a factor of 2.7, and WorkerIndividual by a factor of 4.9 on the IBM Rochester system. On the ANL system, it outperforms WorkerCollective and WorkerIndividual by a factor of 2.7 and 1.9, respectively.

To further understand the results, we present the breakdowns of the executions with 128 worker processes on both systems in Table I. For each time fraction, the average execution times of all workers is reported. The most surprising observation in these results is that the execution time spent on the BLAST functionality is almost identical across different tests. It is, in fact, the time spent on writing the output that makes the most dramatic difference between various output strategies and system configurations. Specifically, WorkerMerge, which performs the best, spends less than 1 second or 0.3% of the total execution time on the actual writing. While for WorkerCollective, about 800 seconds or over 40% of the time is spent on writing the results. It is worth noting that we benchmark the write time in WorkerCollective by measuring the execution time spent on the collective I/O operations including setting file views and performing collective writes. Hence the write time measured for WorkerCollective is dominated by the synchronization cost. As for WorkerIndividual, the write time differs by an order of magnitude on the two systems with different I/O capability. For the same BLAST job, it takes around 400 seconds to write the output on the Rochester system but only slightly more than 4 seconds on the ANL system. Note that the write time is measured as the time spent on calling MPI_File_write() function. One possible reason for this behavior is that when the I/O subsystem is not capable to process the non-contiguous I/O requests quickly enough, more and more successive non-contiguous I/O requests enter the file system resulting severe I/O contention.

The actual amount of data generated by the BLAST job in the above experiments is about 285 MB. According to our I/O benchmarks on the IBM Rochester system, the peak performance we measured with one MPI process doing sequential I/O is 105 MB/s, which implies that the complete output data can be written to the file system within a few seconds with sequential, contiguous I/O. This explains why the writing cost is trivial in WorkerMerge. The results of WorkerIndividual tests show that uncoordinated, non-contiguous writing results in poor I/O performance. Collective I/O, on the other hand, improves the actual I/O performance but hurts the overall system performance with significant synchronization costs. WorkerMerge performs the best because it can achieve high I/O performance while allowing involved processes to collaborate asynchronously.

Next, we compare the performance of output strategies by scaling the number of partitions. As can be seen in Fig. 2,

WorkerIndividual works well with a small number of workers and can deliver almost the same throughput as WorkerMerge for 128 workers or less. Since the I/O subsystem in BG/P scales with the system size, one might wonder whether WorkerIndividual can achieve a similar performance as WorkerMerge if we limit the number of worker processes in a partition and grow the number of partitions. Such a hypothesis can be verified by examining the scalability of various strategies within our hierarchical scheduling framework. In this group of tests, we fix the partition size at 128, and increase the system size from 512 to 2048 cores. The input is a set of 2048 sequences randomly sampled from the `nt` database. The experiments are performed on the ANL system. As Fig. 3 shows, both WorkerCollective and WorkerMerge scale well as the system size grows, while WorkerMerge holds a significant winning margin. It is evident that WorkerIndividual scales poorly even when the I/O system scales with the system size. As a result, the overall system performance stops decreasing from 1024 to 2048 cores.

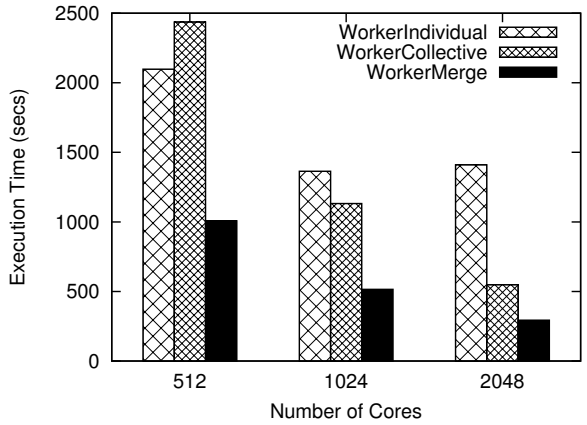


Fig. 3. Scalability of three output strategies on multiple partitions.

C. Load Balancing

In this section, we compare two existing static load-balancing approaches with our dynamic load-balancing design. The first static approach splits the input queries among equal-sized processor groups so that the queries assigned to each group have the same total lengths, as used by Rangwala et. al. in their parallel BLAST implementation for BG/L [18]. We call this approach *LengthSplit*. The second static approach, used in ScalaBLAST [16], splits queries by taking into account the number of queries in addition to the total query lengths. Specifically, each group is assigned a batch of queries that contains the same mount of "work units", which is calculated as follows for N queries:

$$\sum_{i=0}^{N} length(q_i) + N * weight\_factor \qquad (1)$$

We will refer to this approach as *WeightSplit*. In our experiments, the $weight\_factor$ is configured as 225 as used in [16]. In our experiments, we again use `nt` as the test database.

According to our previous study [8], the compute-intensive sequences in the `nt` database are likely to be longer than 5KB. So we create an input query set by collecting the first 1024 sequences in the `nt` database that are shorter than 20KB; thus some long sequences in the query set can be expected to cause search imbalances. We do not include sequences longer than 20KB as those sequences could take hours to search based on our past experiences, preventing the experiments from finishing within the queue time limit on the test platform.

To implement the static load-balancing approaches, we use 1024 nodes on the ANL BG/P system divided into groups of 128 nodes and launch an instance of mpiBLAST in each group. We split the input queries into 8 batches using LengthSplit or WeightSplit depending on the static load-balancing approach, and have each mpiBLAST instance search a distinct batch of queries[1]. In comparison, in the dynamic load-balancing approach, we configure mpiBLAST to run on all processors performing load balancing among the 8 groups. For each approach, the maximum, average, and minimum processing time spent on each group are reported in Fig. 4 with the overall program execution time being the maximum group execution time. From the disparity between the maximum and average group execution times, we can see that the synthesized input query set creates large load imbalances among all three approaches. Interestingly, the performance of both static approaches is very similar; in both cases, the maximum group execution time is about 2.3 times higher than the average group execution time. On the other hand, although the dynamic load-balancing approach is also affected by search imbalances, it brings down the ratio of maximum execution time to average execution time to 1.8, resulting in performance improvements by a factor of 1.39 and 1.42 compared to LengthSplit and WeightSplit respectively.

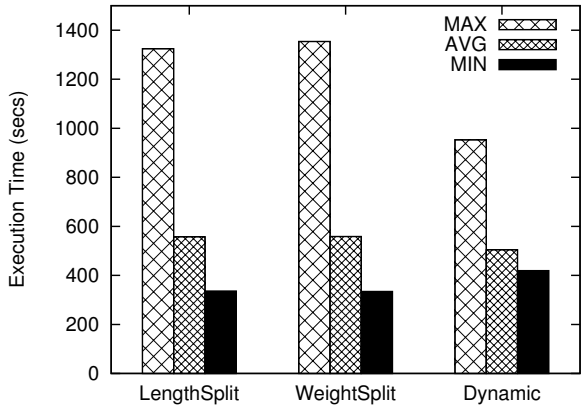


Fig. 4. Performance comparison of static and dynamic load-balancing approaches. LengthSplit and WeightSplit are two static load-balancing approaches used in other large-scale sequence-search applications. Dynamic is the dynamic load-balancing approach used in mpiBLAST.

[1]We were not able to compare our load balancing approach directly to other sequence-search tools adopting static load balancing because the source codes of those tools are not publicly available.

D. Scalability Tests

To evaluate the scalability of mpiBLAST, we run and profile a large-scale, sequence-search job — sequence searching the entire microbial genome database against itself — in order to address two vitally important problems in computational biology: (1) discovering missing genes in genomes and (2) inferring structure in genomic sequence databases.[2]

1) Problem Description: Sequence searching the entire microbial genome database against itself has several utilities in computational biology as described in [1]. A summarized description of these utilities is noted below:

Discovering Missing Genes: Genome annotation identifies the location of genes and the coding regions in a genome, determines what those genes do, and then annotates this information to the genome. Part of the above process entails accurately determining the location and structure of protein-encoding and RNA-encoding genes via computational analysis. If done improperly, we end up *predicting false genes* or *missing real genes*.

A popular method for locating genes, known as the similarity method, requires the comparison of genomic segments with a database of gene sequences found in similar organisms. If the sequence is conserved, then the segment that is being evaluated is likely to be a coding gene. Genes that do not fit a given genomic pattern and do not have similar sequences in current annotation databases may be systemically missed.

To detect missed genes, we use the similarity method and compare raw genomes against each other rather than comparing a raw genome to a database of known genes. For instance, if gene x in genome X and gene y in genome Y have been missed and x is similar to y, then the similarity method will find both. However, the *only* way of identifying this is to perform an all-to-all comparison of the entire microbial genome database against itself, which is highly compute-intensive.[3]

Adding Structure to Genetic Sequence Databases: One of the major issues with sequence searching is the structure of the sequence database itself. Currently, these databases are "structured" as a flat file in human-readable format, e.g., ASCII text, and each new sequence that is discovered is simply appended to the end of the file. Without more intelligent structuring, the query sequence needs to be compared to every sequence in the database (several millions currently) forcing the best-case to take just as long as the worst-case. By organizing and providing structure to the database, searches can be performed more efficiently by being able to discard irrelevant portions entirely. One way to provide structure to the sequence database is to create a sequence similarity tree where "similar" sequences are closer together in the tree than dissimilar sequences. The connections in the tree are created by determining how "similar" the sequences are to each other; sequence search can be used to determine the sequence similarity. To create *every* connection, however, an "all-to-all" sequence search must be performed where the input query being the same as the database, resulting in an output size of N^2 values (where N is the number of sequences in the database).

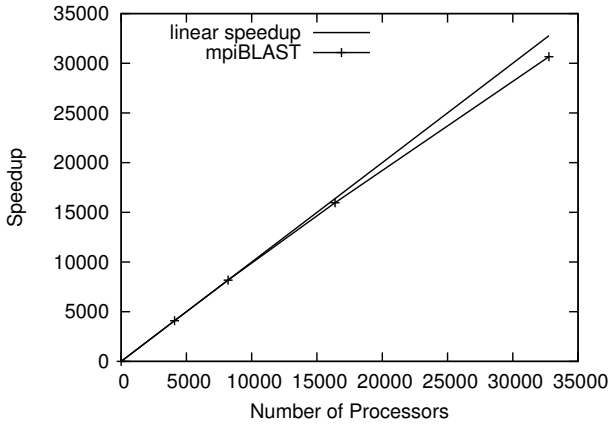


Fig. 5. Speedup of searching a quarter million of microbial genome sequences against the microbial genome database itself.

2) Performance Results: We first systematically evaluated the combination of mpiBLAST's hierarchical scheduling and its optimized parallel output strategy by searching 0.25 million query sequences randomly sampled from the microbial genome database against the database itself. We performed initial profiling by varying the partition size, the number of database replicas per partition, and the number of database fragments. This profiling allowed us to identify the ideal values for different parameters for our system — 64 database fragments, 128 as the partition size, and 16 as the replication group size.

Based on this profiled information, in this experiment, we increase the system size from 1 rack to 8 racks and measure the speedup achieved as illustrated in Fig. 5. As can be seen, our enhancements allow mpiBLAST to scale almost linearly from 4096 cores to 32768 cores, achieving a *93% parallel efficiency* at the largest test scale. This near-perfect speedup demonstrates that the synergy of mpiBLAST's scalable task and I/O scheduling design can take full advantage of the massive parallelism offered by the BG/P system.

Finally, we leverage the processing power of BG/P system, enhanced by our optimizations, to solve the problem of searching the entire microbial genome against itself. To avoid data loss of possible hardware failures during the long run, we split the database into 64 query files, each consisting of about 85MB of sequence data, and continue submitting jobs to the 8-rack system until the whole genome search is completed. This problem, previously considered to be computationally intractable in practice, was completed within *12 hours*.

[2]Note: Several of the co-authors on this paper were involved in a similar endeavor that won the SC—07 Storage Challenge Award, but by running in a significantly different environment, a distributed ad-hoc grid consisting of over 12,000 processors from six U.S. supercomputing sites and a petabyte files at the Tokyo Institute of Technology.

[3]When this computation was done as part of the SC—07 Storage Challenge, it required 12,000+ processors to compute over a two-week period and write to a petabyte file system.

VI. Concluding Remarks

In this paper, we presented several issues that arise when scaling parallel sequence search applications such as mpiBLAST to massive scale systems such as BG/P. Specifically, issues related to task-mapping and data I/O that are unnoticeable on small and medium clusters with a fast I/O subsystem can form significant bottlenecks on massively parallel systems with limited I/O capabilities. We presented several designs to alleviate these bottlenecks, performed extensive evaluations to understand their performance benefits, and used the best of the designs to enhance mpiBLAST. We also demonstrated that these designs allow the application to scale almost linearly (93% efficiency) on upto 32768 cores of BG/P.

VII. Acknowledgments

We thank IBM Rochester for providing access to Blue Gene/P development systems and acknowledge the support of the the Biomedical Informatics and Computational Biology (BICB) program at University of Minnesota, Rochester. On an individual basis, we are grateful to Carl Obert and John Thomas for their support and to Jeremy Archuleta whose feedback helped improve the presentation of the paper.

This research has been partially supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357; a DOE Early Career PI Award DE-FG02-05ER25685 also from the Office of Advanced Scientific Computing Research; NSF Awards CCF-0621470 and CPA-0702182; an IBM Faculty Award; and a joint appointment between NCSU and ORNL.

References

[1] P. Balaji, W. Feng, H. Lin, J. Archuleta, S. Matsuoka, A. Warren, J. Setubal, E. Lusk, R. Thakur, I. Foster, D. S. Katz, S. Jha, K. Shinpaugh, S. Coghlan, and D. Reed. Distributed Data I/O with ParaMEDIC: Experiences with a Worldwide Supercomputer. In *Proceedings of the IEEE International Supercomputing Conference (ISC);* **Best paper award**, Dresden, Germany, June 2008.

[2] 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.

[3] R. Braun, K. Pedretti, T. Casavant, T. Scheetz, C. Birkett, and C. Roberts. Parallelization of local BLAST service on workstation clusters. *Future Generation Computer Systems*, 17(6), 2001.

[4] N. Camp, H. Cofer, and R. Gomperts. High-throughput BLAST. http://www.sgi.com/industries/sciences/chembio/resources/papers/HTBlast/HT_Whitepaper.html.

[5] 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.

[6] P. Crandall, R. Aydt, A. Chien, and D. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, 1995.

[7] 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.

[8] 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 SC—06*, 2006.

[9] 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, 2005. IEEE Computer Society.

[10] M. Marra, S. Jones, C. Astell, R. Holt, A. Brooks-Wilson, Y. Butterfield, J. Khattra, J. Asano, S. Barber, S. Chan, A. Cloutier, S. Coughlin, D. Freeman, N. Girn, O. Griffith, S. Leach, M. Mayo, H. McDonald, S. Montgomery, P. Pandoh, A. Petrescu, G. Robertson, J. Schein, A. Siddiqui, D. Smailus, J. Stott, G. Yang, F. Plummer, A. Andonov, H. Artsob, N. Bastien, K. Bernard, T. Booth, D. Bowness, M. Drebot, L. Fernando, R. Flick, M. Garbutt, M. Gray, A. Grolla, S. Jones, H. Feldmann, A. Meyers, A. Kabani, Y. Li, S. Normand, U. Stroher, G. Tipples, S. Tyler, R. Vogrig, D. Ward, B. Watson, R. Brunham, M. Krajden, M. Petric, D. Skowronski, C. Upton, and R. Roper. The genome sequence of the sars-associated coronavirus. *Science*, 2003.

[11] D. Mathog. Parallel BLAST on split databases. *Bioinformatics*, 19(14), 2003.

[12] J. May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann Publishers, 2001.

[13] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Standard*, July 1997.

[14] J. Nieplocha, R. Harrison, and R. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2), 1996.

[15] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and Michael L. Best. File-access characteristics of parallel scientific workloads. *IEEE Trans. Parallel Distrib. Syst.*, 7(10), 1996.

[16] C. Oehmen and J. Nieplocha. Scalablast: A scalable implementation of blast for high-performance data-intensive bioinformatics analysis. *IEEE Trans. Parallel Distrib. Syst.*, 17(8), 2006.

[17] D. Quintero and M. Hennecke. GPFS Multicluster with the IBM System Blue Gene Solution and eHPS Clusters. IBM Redpaper, REDP-4168-00, October 24, 2006, http://www.redbooks.ibm.com/abstracts/redp4168.html?Open.

[18] 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.

[19] 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.

[20] E. Smirni, R. Aydt, A. Chien, and D. Reed. I/O requirements of scientific applications: An evolutionary view. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, 1996.

[21] E. Smirni and D. A. Reed. Lessons from characterizating the input/output behavior of parallel scientific applications. *Perform. Eval.*, 33(1), 1998.

[22] C. Sosa and G. Lakner. IBM System Blue Gene Solution: Blue Gene/P Application Development . IBM RedBook, SG24-7287, ISBN 0738488674, Rochester, Minnesoat, 2008. http://www.redbooks.ibm.com/abstracts/sg247287.html?Open.

[23] R. Thakur and A. Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4), 1996.

[24] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.

[25] 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.

[26] 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.

[27] R. Thakur, W. Gropp, and E. L. Lusk. An experimental evaluation of the parallel i/o systems of the ibm sp and intel paragon using a production application. In *Proceedings of the Third International ACPC Conference with Special Emphasis on Parallel Databases and Parallel I/O*, London, UK, 1996. Springer-Verlag.

[28] O. Thorsen, B. Smith, C. Sosa, K. Jiang, H. Lin, A. Peters, and W. Feng. Parallel genomic sequence-search on a massively parallel system. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, New York, NY, USA, 2007. ACM.