Exploring I/O Strategies for Parallel Sequence-Search Tools with S3aSim

Avery Ching[†], Wu-chun Feng[‡], Heshan Lin[§], Xiaosong Ma[§*], Alok Choudhary[†]

[†] Department of EECS
Northwestern University
{aching,choudhar}@ece.northwestern.edu

[§] Department of Computer Science
North Carolina State University
{hlin2@,ma@csc}.ncsu.edu

[‡] Department of Computer Science
Virginia Tech
feng@cs.vt.edu

Abstract

Parallel sequence-search tools are rising in popularity among computational biologists. With the rapid growth of sequence databases, database segmentation is the trend of the future for such search tools. While I/O currently is not a significant bottleneck for parallel sequence-search tools, future technologies including faster processors, customized computational hardware such as FPGAs, improved search algorithms, and exponentially growing databases will emphasize an increasing need for efficient parallel I/O in future parallel sequence-search tools.

Our paper focuses on examining different I/O strategies for these future tools in a modern parallel file system (PVFS2). Because implementing and comparing various I/O algorithms in every search tool is labor-intensive and time-consuming, we introduce S3aSim, a general simulation framework for sequence-search which allows us to quickly implement, test, and profile various I/O strategies. We examine a variety of I/O strategies (e.g., master-writing and various worker-writing strategies using individual and collective I/O methods) for storing result data in sequence-search tools such as mpiBLAST, pioBLAST, and parallel HMMer. Our experiments fully detail the interaction of computing and I/O within a full application simulation as opposed to typical I/O-only benchmarks.

1 Introduction

Sequence-search is one of the fundamental tasks routinely performed in computational biology. Sequence-search is typically used to find similarities between newly discovered DNA or amino-acid sequences and those in known nucleotide or protein databases. The results of sequence-search can be used to predict the structures and functions of new sequences. They also allow people to estimate the evolution distance in phylogeny reconstruction and perform gnome alignments. With the introduction of advanced sequencing technologies, sequence databases are rapidly growing. For example, GenBank [1] (a widely used DNA sequence database maintained by the National Center for Biotechnology Information) increased in size by over five orders of magnitude from 1982 to 2004 [18]. Parallel sequence-search tools are necessary for sequence analysis of modern and future sequence databases.

Query segmentation and database segmentation are the popular design choices for parallel sequence-search tools on general-purpose parallel machines. Many existing parallel sequence-search tools are based on query segmentation [4, 6, 3, 20, 10]. In this approach, the entire sequence database is replicated to all processors and a set of query sequences are segmented into fractions. Each processor searches a fraction of query sequences against the entire sequence database. When the sequence database does not fit into the processor memory, query segmentation suffers repeated I/O introduced by loading sequence data back and forth between the file system and the main memory. In database segmentation, the entire set of query sequences is replicated to all processors and the sequence database is partitioned (as shown in Figure 1. Each processor searches whole query sequences against a fraction of the sequence database. Super-linear speedup is possible when the sequence database is larger than the processor memory by fitting the large database into the aggregate memory of all processors [9]. Parallel sequence-search tools that use database segmentation include mpiBLAST [9], pioBLAST [14], TurboBlast [2] and parallel BLAST [17].

[*] Xiaosong Ma has a joint appointment in the Computer Science and Mathematics Division at Oak Ridge National Laboratory.

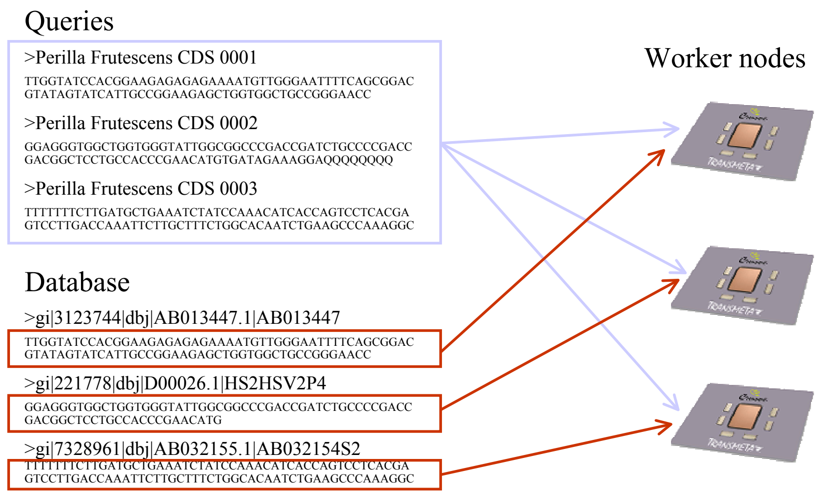
Queries
>Perilla Frutescens CDS 0001
TTGGTATCCACGGAAGAGAGAGAAAATGTTGGGAATTTTCAGCGGAC
GTATAGTATCATTGCCGGAAGAGCTGGTGGCTGCCGGGAACC
>Perilla Frutescens CDS 0002
GGAGGGTGGCTGGTGGGTATTGGCGGCCCGACCGATCTGCCCCGACC
GACGGCTCCTGCCACCCGAACATGTGATAGAAAGGAQQQQQQQ
>Perilla Frutescens CDS 0003
TTTTTTTCTTGATGCTGAAATCTATCCAAACATCACCAGTCCTCACGA
GTCCTTGACCAAATTCTTGCTTTCTGGCACAATCTGAAGCCCAAAGGC
Database
>gi|3123744|dbj|AB013447.1|AB013447
TTGGTATCCACGGAAGAGAGAGAAAATGTTGGGAATTTTCAGCGGAC
GTATAGTATCATTGCCGGAAGAGCTGGTGGCTGCCGGGAACC
>gi|221778|dbj|D00026.1|HS2HSV2P4
GGAGGGTGGCTGGTGGGTATTGGCGGCCCGACCGATCTGCCCCGACC
GACGGCTCCTGCCACCCGAACATG
>gi|7328961|dbj|AB032155.1|AB032154S2
TTTTTTTCTTGATGCTGAAATCTATCCAAACATCACCAGTCCTCACGA
GTCCTTGACCAAATTCTTGCTTTCTGGCACAATCTGAAGCCCAAAGGC
Worker nodes

Figure 1. Database segmentation.

Database segmentation is expected to be the inevitable trend of future parallel sequence-search tools for following reasons. First, the rapid growth of sequence databases prohibits a sequence database from fitting into the memory of a single processor. Second, as sequence databases increase in size, searching a query against the whole database will take substantial time and result in resource under-utilization when the number of sequences is relatively small compared to the number of processors. Database segmentation offers better resource utilization on large-scale machines regardless of number of input query sequences.

Although current I/O costs in parallel sequence-search tools (such as mpiBLAST) are relatively small in proportion to overall execution time, we believe future I/O performance will be increasingly important to sequence-search throughput because of following reasons. First, the performance gap between processor speed and I/O speed continues to widen, making I/O much more significant in overall execution time. Search times are shrinking as we use advanced computational hardware such as multicore-chip architectures. Solutions based on field programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs), such as BioScan [23], Parcel's GeneMatcher [11], Compugen's Bioccelerator [15] and TimeLogic's DeCypher [12], have proven to be very efficient and can deliver orders-of-magnitude performance improvements in comparing large sequences. Second, the development of smarter heuristic algorithms (such as SSAHA [19], PatternHunter [16], and BLAT [13]) greatly reduces the sequence-search costs.

Our past experience with parallel sequence-search tools has led us to believe that an individual worker-writing I/O strategy would significantly improve overall execution times. In this paper, we compare such an I/O strategy against other I/O strategies for parallel sequence-search tools using database segmentation. We have developed S3aSim, a sequence similarity search algorithm simulator, to get a detailed understanding of I/O strategies in parallel sequence-search tools.

In Section 2, we describe and compare the various I/O strategies for parallel sequence-search tools. In Section 3, we introduce S3aSim, our simulation tool for understanding I/O in parallel sequence-search tools. In Section 4, we present our S3aSim results with respect to increased processors and increased computational ability. In Section 5, we conclude with our contributions in this paper and discuss possibilities for future work.

2. I/O Algorithms in Parallel Sequence-Search Tools

Although parallel sequence-search tools employ different sequence-alignment algorithms, they have much in common when using the database segmentation approach, as proposed by [9]. First, each processor searches a fraction of the database. These searches on different processors are embarrassingly parallel. Second, the output results in query sequence match database similarities ordered by statistics representing the alignment qualities. Finally, local outputs in different processors need to be merged and sorted according to the search statistics (usually a similarity score) in order to produce the final output. Most parallel sequence-search tools will use the message passing interface (MPI) and its associated I/O chapter (MPI-IO) for code portability. In the rest of the paper, we will refer to some example parallel database-search tools such as mpiBLAST and pioBLAST.

Older versions of mpiBLAST would store all the results on master/worker nodes until the end of program execution. Since this can result in exceeding memory limits in large application runs, the current design path (for example mpiBLAST 1.4) has headed towards writing the results out immediately after a query is processed or after every *n* queries have been processed, where *n* is a fraction of the overall input query set. More frequently writing out the results also allows users to resume a failed application run at the appropriate input query. Workers always send their results back to the master ordered by score. As the master receives the ordered results, it can easily merge them together with its list of ordered results. Basically, the sorting costs are offloaded as much as possible to the workers so the master can focus on its primary job of distributing tasks to the workers.

We note that parallel sequence-search algorithms have a unique set of I/O characteristics when writing out results.

- **Non-uniform result size** - The size of each result is relative to the minimum searching threshold and up to three times the maximum of the input query and the matching database sequence (for BLAST output).

- **Result count** - This is completely data dependent and can range from no results to hundreds of thousands of results depending on the input queries and the database size.

- **Unstructured data** - When the workers write to the output file, the resulting data is noncontiguous and unstructured (i.e., no regularity). When the master writes to the output file, the resulting I/O calls are large and contiguous.

In the upcoming subsections, we describe possible I/O strategies for writing a results file.

2.1. Master-Writing

After each worker completes its queries, it sends its ordered results to the master. This involves sending over its scores and actual result data. The master merges the ordered results with its ordered results. If all the fragments for an input query have been processed, it writes the ordered results to a file in one contiguous I/O call. We will refer to this master-writing strategy as the MW strategy in the rest of the paper.

The MW strategy resembles the I/O strategy used in mpiBLAST 1.2. A large difference, however, is that mpiBLAST 1.2 did not write out results after each query. In mpiBLAST 1.2, the master wrote all its results at the end of the application run. This limited the size of input queries and the target database used. It also provided little opportunity for resuming an application run after failure.

The main advantage of the MW strategy is that it writes the output data contiguously. Contiguous I/O is much more efficient that noncontiguous I/O [7]. The MW strategy is also easy to implement and debug. However, one disadvantage of the MW strategy is that the master is a centralized point of contention where the full result data is sent. Only a single process is gathering all the results and doing the writing on behalf of all the workers. Also, while the master is writing, it cannot assign new tasks to the workers (causing potentially long wait times before a worker can begin a new task). While nonblocking I/O could reduce this overhead, blocking I/O is commonly used in a MW strategy to avoid overloading the memory of the master process. As we scale up the number of workers, the MW strategy will likely not scale as well as the other I/O strategies.

2.2. Worker-Writing: Collective I/O

An application designer can choose to have the workers write the results themselves in order to help the master focus on assigning work. After each worker finishes processing its input query against its database fragment, it sends over only its scores of the ordered results to the master. The master merges in the ordered scores with its own ordered scores. If all the fragments for an input query have been processed, it sends the workers the location of where to write each result in the aggregate output file. This location information consists of a list of 64-bit offsets sent to each worker with results. All of the workers then synchronize to write their results collectively to the correct locations in the output file. Since results are written to mutually exclusive locations in the file, the data is interleaved but not overlapping. We will refer to this worker-writing strategy with collective I/O as the WW-Coll strategy. We refer to the more general class of worker-writing strategies as WW strategies.

The WW-Coll strategy, proposed by pioBLAST [14], uses MPI-IO collective writes to instruct workers to simultaneously write all of their results at the end of program execution. When compared to the MW strategy, collective worker-writing allows the I/O bandwidth to scale up with the number of workers. In most cases, having more clients writing simultaneously provides better I/O throughput to a high-performance file system. However, as noted earlier, a disadvantage of the WW strategies is that the workers must use noncontiguous I/O methods. Furthermore, with the WW-Coll strategy, all the workers must synchronize with each other to write. This synchronization cost is at least the time from when the first worker receives its result list to when the last worker receives its result list (before collective I/O begins). On the other hand, since the worker only sends the scores and not the actual results to the master, the amount of data exchanged with the master is reduced from the MW strategy (even including the overhead of the location list data passed from the master to the workers).

2.3. Worker-Writing: Individual I/O

We propose to modify the WW-Coll strategy to use individual I/O in parallel sequence-search tools. Instead of using collective I/O, we let the workers write results after completing an input query (or a group of input queries) using individual noncontiguous I/O methods. Our modified WW strategy begins with the master issuing input queries to the workers. The workers are responsible for processing the queries, generating the sorted results, and sending the ordered scores to the master. The master returns the location list to the workers and each worker writes the result data to the output file on its own (not collectively) when it notices it has received the location list from the master. While workers wait for the location list from the master, they can process additional queries, unlike the WW-Coll strategy. Since collective I/O requires all involved processes to block until synchronized, the WW-Coll strategy cannot allow worker processes to begin upcoming queries until after the I/O operation.

That is, we try to eliminate the synchronization time inherent in collective I/O and relieve pressure on the file system by writing when a worker is ready instead of forcing all workers to simultaneously write. Eliminating the synchronization time should have a significant impact on overall application performance and balance out the load on the file system. We used two different noncontiguous I/O methods (POSIX I/O and list I/O). The POSIX I/O method is the MPI_Write() call without optimization. The list I/O method, described in [8], is an optimization for high-performance file systems. We call our modified WW strategies with POSIX I/O and list I/O, the WW-POSIX strategy and WW-List strategy, respectively.

3. S3aSim

Each of the aforementioned I/O strategies would be difficult to compare in a single application (such as mpiBLAST or pioBLAST). The main difficulties are implementation time and complexity. Each I/O strategy requires substantial changes to the overall parallel search algorithm. They could also require changes in network protocol and intricacies of the actual search algorithms (for example, modifying NCBI [18] BLAST code). We do not wish to compare mpiBLAST 1.2, pioBLAST, and our individual worker-writing strategy in another parallel sequence-search tool to compare I/O strategies. At this time, no benchmarks for parallel I/O in bioinformatics exist. In order to create a fair comparison of I/O strategies that provides flexibility in altering input parameters (such as computational time, input query size, I/O strategies), we created S3aSim: a <u>s</u>equence <u>s</u>imilarity <u>s</u>earch <u>a</u>lgorithm <u>sim</u>ulator.

Algorithm 1 Master Process

```
1:  Distribute input variables to the workers and setup internal data structures.
2:  while {1} do
3:      MPI_Recv() a request for work.
4:      if All queries have been scheduled then
5:          Notify worker all queries have been scheduled.
6:      else
7:          MPI_Send worker (query #, fragment #).
8:          Post MPI_Irecv() for worker scores (and results if MW).
9:      end if
10:     Check MPI_Irecv() to see if any workers have finished sending results for their (query #, fragment #).
11:     if All queries have been scheduled then
12:         Continue {Inform other workers that all queries have been scheduled before proceeding}
13:     end if
14:     if Use Parallel I/O then
15:         MPI_Isend() offset list to workers for any completed queries.
16:         Check to see which MPI_Isend() calls completed.
17:     else
18:         Write finished results to output file for completed queries.
19:     end if
20:     if All queries scheduled, processed, and results written to output file then
21:         Exit()
22:     end if
23: end while
```

S3aSim provides many benefits for our I/O strategy comparison including simple implementation, variability of many input parameters, and integration with the multi-processing environment (MPE) and Jumpshot [24] for easy debugging. It is a tool to understand how computation and I/O interact together in a typical parallel sequence-search tool. It uses both MPI and MPI-IO calls for portability on many large-scale machines. S3aSim allows the user to customize the total number of fragments of the database, total number of input queries, a box histogram of input query sizes, a box histogram of database sequence sizes, a min/max count of results per input query, a minimum result size per query, variable simulated compute speeds, MPI-IO hints, parallel I/O, write all data at the end (similar to mpiBLAST 1.2 and pioBLAST), and many others.

The primary disadvantage of S3aSim is that the modeling of the computational time is inexact. Compute time is modeled as a constant startup cost + linear time based on the size of the result (anywhere from the minimum input size to three times the maximum of the input query and the matching database sequence). We use three times the maximum of the input query and the matching database sequence as the S3aSim result size since the actual BLAST output is generally formatted with the input sequence, database sequence, and the matches between them. This formula can be modified to more accurately model various search algorithms for future work.

Algorithm 2 Worker Process

```
1: Receive input variables from the master and setup internal data structures.
2: while {1} do
3:   MPI_Send() the master a request for work.
4:   MPI_Recv() response from the master.
5:   if (query #, fragment #) then
6:     Compute search algorithm on (query #, fragment #).
7:     if Use Parallel I/O then
8:       Merge current results with previous results for this query.
9:     end if
10:    MPI_Isend() scores (and results if MW) to the master.
11:    if Use Parallel I/O then
12:      Post MPI_Irecv() offset list from the master.
13:    end if
14:  end if
15:  Check all pending MPI_Isend() for completion.
16:  if Use Parallel I/O then
17:    Check all pending MPI_Irecv() for completion.
18:    For all offset lists received, write results to output file.
19:  end if
20:  if All queries scheduled, processed, and results written to output file then
21:    Exit()
22:  end if
23: end while
```

The basic master algorithm is outlined in Algorithm 1 and the basic worker algorithm is outlined in Algorithm 2. In order to maximize the amount of time spent on distributing work, the master uses the blocking MPI_Recv() call when handling worker requests. We used MPI_Isend()/MPI_Irecv() calls for all other communication (receiving results and sending offset lists). When we used parallel I/O (either individual or collective worker writing), the workers only sent the scores of their results to be sorted by the master. If the master was writing, both the scores and the results of a search on a (query #, fragment #) were sent to the master. We note that whenever a process (master or worker) is checking on a nonblocking communication call, it will only test for completion (MPI_Test()) instead of blocking on completion (MPI_Wait()) to allow the process to continue to make progress if the test fails. It will only call MPI_Wait() if the process cannot proceed further until the completion of this particular nonblocking communication call.

S3aSim timing is broken up into different timing phases. We will describe each timing phase with respect to the overall program execution for both the master and the worker.

- **Setup Time** - For the master, this time includes sending out input variable information to the workers (step 1 in Algorithm 1). For the worker, this time includes receiving the input variable information from the master (step 1).

- **Data Distribution** - For the master, this time includes waiting for the next worker request and sending the worker a response (steps 3, 5, and 7). For the worker, this time includes sending the work request and receiving a response (steps 3 and 4).

- **Compute** - For the master, this time is always 0 since the master never does any searching. For the worker, this time includes the search algorithm time (step 6).

- **Merge Results** - For the master, this time is 0. For the worker, this time includes the time to merge results together if we are using parallel I/O (step 8).

- **Gather Results** - For the master, this time includes posting MPI_Irecv() operations for scores (and possibly results) from the worker as well as checking on the associated MPI_Irecv() operations (steps 8 and 10). For the worker, this time includes sending off the scores (and possibly results) as well as checking on the associated MPI_Isend() operations (steps 10 and 15).

- **I/O** - For the master, this time includes all write operations to the output file (step 18). For the worker, this time includes all write operations to the output file (step 18).

- **Sync** - For the master, this time includes waiting for all the processes to synchronize at the end of the application (not shown in the algorithm). For the worker, this time includes waiting for all the processes to synchronize at the end of the application (not shown in the algorithm). When the query sync mode is on, this time includes the time for all processes to synchronize after writing out the results for an input query.

- **Other** - For the master and the worker, this phase includes all remaining time.

3.1. Parallel Virtual File System 2 (PVFS2) and ROMIO

The Parallel Virtual File System 2 (PVFS2) [22] is a parallel file system for commodity Linux clusters that is a complete redesign of PVFS1 [5]. It provides both a cluster-wide consistent name space and user-defined file striping found in PVFS1 but also adds functionality to provide better scalability and performance. Many of its components are modular in design which allows flexibility in customizing PVFS2 to meet individual needs. Most relevant to this paper is the native support for high-performance noncontiguous data access (list I/O, for example).

Since S3aSim has no overlapping writes, its I/O phase should proceed in parallel in the WW strategies without any serialization. PVFS2 does not provide a method for atomicity of overlapping writes, so there is no I/O serialization due to false sharing. Other file systems which use serialization mechanisms, such as file locking, for handling atomic overlapping I/O may unnecessarily serialize writes in the I/O phase. Such serialization mechanisms should be turned off since our application has no overlapping I/O.

ROMIO is the MPI-IO implementation developed at Argonne National Laboratory [21]. It builds upon the MPI-1 message passing operations and supports many underlying file systems through the use of an abstract device interface for I/O (ADIO). ADIO allows the use of file-system specific optimizations such as the list I/O interface used in our experiments. Additionally ROMIO implements the data sieving and two phase optimizations as generic functions that can be used for all file systems supported by ADIO. It also implements a datatype flattening system that is used to support list I/O for PVFS2. We used the default two phase I/O method in ROMIO as our collective I/O implementation in the WW-Coll strategy.

3.2. Test Environment

All tests were run on the Feynman cluster at Sandia National Laboratories. Feynman, composed of Europa nodes, Ganymede nodes, and I/O nodes, has a total of 371 computers with dual processors. In order to keep our testing as homogeneous as possible, we only used the Europa nodes that were dual 2.0-GHz Pentium-4 Xeon CPUs with 1-GByte RDRAM. They were connected with a Myrinet-2000 network and used the Redhat Linux Enterprise operating system. Since each of compute nodes had dual CPUs, we ran two compute processes per node.

We also used 16 computers in our PVFS2 file system. All 16 computers ran the PVFS2 server with one computer additionally handling metadata server responsibilities. All PVFS2 files created use the default 64-KByte strip size, which totals to a 1-MByte stripe across all I/O servers. Our version of PVFS2 was tuned for better noncontiguous I/O by adjusting default system parameters.

3.3. Test Setup

With so many input variables, there are numerous tests we could run with S3aSim. In this paper, we chose to focus on testing the scalability of the I/O strategies with respect to processor count and compute speed. In order to get the characteristics of an NCBI database, we chose the NT database (nt.gz from ftp://ftp.ncbi.nih.gov/blast/db/FASTA/) as our example database. The NT database has a minimum sequence length of 6 bytes, a maximum sequence length of slightly over 43 MBytes, and mean sequence length of 4401 bytes. We used the same histogram to represent our input query set of 20 queries (roughly maps to approximately 86 KBytes of input queries). We chose 128 fragments and a result count from 1000 to 2000 per query (over the entire database). Results were written to the output file after each query. MPI_File_sync() was always called immediately after every MPI_File_write() or MPI_File_write_all(). Although we use different numbers of processors, the results are always identical since they are pseudo-randomly generated. Each data point we present generated roughly 208 MBytes of output data and was averaged over 3 test runs.

Our tests were designed for comparing the performance of the various I/O strategies with respect to an increase in processes or reduced computational time (custom search hardware and/or better algorithms). One suite of tests used 2 to 96 processors. The second suite of tests used compute speeds from 0.1 to 25.6. The breakdown of the application into phases is crucial to understanding why certain I/O strategies work better than others.

Another goal of our testing was to examine whether the inherent I/O synchronization of collective I/O would be very costly and how it might be improved. Collective I/O, in nearly all noncontiguous I/O cases, outperforms POSIX I/O and, in some noncontiguous I/O cases, outperforms list I/O in pure I/O tests. It is rare (if ever) when an I/O comparison takes into account interaction with an application when doing performance evaluations. It is very hard to directly examine the effect of inherent I/O synchronization in collective I/O. In order to expose this effect, our tests used the "query sync" option. Basically, if the query sync option is on, S3aSim will force all worker nodes to synchronize after doing any I/O. When looking at the performance changes from individual I/O methods with query sync off to with the query sync on, we can gain a better understanding of how the inherent synchronization in collective I/O hurts overall performance and whether using individual noncontiguous I/O methods in a collective I/O implementation might actually improve performance. For example, a collective I/O method could be implemented using list I/O with a forced synchronization at the end of the I/O operation (similar to our WW-List tests with query sync on). In our upcoming performance evaluation and discussion we refer to not using and using the query sync option as "no-sync" and "sync", respectively.

4. Performance Evaluation

Our first test suite examines process scalability in S3aSim. Figure 2 shows the overall S3aSim execution times on a logarithmic scale to emphasize the performance variation in the different I/O strategies. As expected, all no-sync I/O strategies perform as good as or better than their sync counterparts. The individual WW strategies outperform both the WW-Coll and MW in the no-sync cases. WW-Coll performance is about the same with or without the sync option. It is expected that WW-Coll would have roughly the same performance with or without sync since the inherent synchronization in collective I/O means that synchronizing after all workers do I/O is negligible in our test cases. With the query sync option, WW-Coll (45.54 secs) gets closer to WW-List (40.24 secs) at 96 processors. In overall execution time at 96 processors, WW-List outperforms the other I/O strategies by 364% (MW), 33% (WW-POSIX), and 75% (WW-Coll) in the no-sync cases and 182% (MW), 37% (WW-POSIX), and 13% (WW-Coll) in the sync cases. Noticeable performance gains due to adding more workers slowed considerably at about 32 processes. Generally, at this point the I/O phase time was dominant and even increased slightly with more processes due to the increased frequency of I/O requests (from shorter query processing times). A larger file system configuration with more I/O bandwidth may have provided more scalable I/O performance.

Figure 3 and Figure 4 show the overall breakdown of each run with respect to the different timing phases and different I/O strategies. We present both the no-sync and sync results from the workers to illustrate how each I/O strategy is affected by adding a forced synchronization component. The effect of forced synchronization to MW makes a negligible performance difference (a maximum of 5% in overall execution time mostly due to test variance). Since the workers all wait until the master does I/O before beginning the next query, the forced synchronization is cheap.

WW-POSIX is largely affected by synchronization (up to 69% in overall execution time). The sync phase time increases due to the forced synchronization (from 1.01 secs to 12 secs at 96 processors), which also increases the data distribution phase time (from 3.21 secs to 19.04 secs at 96 processors). The WW-POSIX I/O phase time actually decreases steadily from no-sync to sync since each worker is writing data at a slower rate and handing out less I/O ops/s (up to 17% I/O phase time decrease at 96 processors). This slower rate of I/O ops/s allows PVFS2 to improve pure I/O performance slightly.

WW-List is moderately affected by synchronization. Since its overall I/O costs are always less than WW-POSIX, the sync phase time increase from no-sync to sync is less than WW-POSIX (from 0.41 secs to 5.87 secs at 96 processors). Similar to WW-POSIX, the sync phase time increase causes the data distribution time to rise (from 4.47 secs to 18.47 secs at 96 processors). WW-List also shows an I/O phase time reduction of 34% at 96 processors due to less stress on the file system.

WW-Coll is at most affected by 6% in moving from no-sync to sync cases. Since its own inherent sync bears most of the forced synchronization costs, adding an additional synchronization after I/O is quick and negligible in overall execution time. While collective I/O methods generally outperform POSIX I/O methods [7], we note that WW-Coll loses to WW-POSIX in the no-sync case. As we described in Section 2.3, the inherent synchronization of WW-Coll is more costly than the gains in I/O performance alone over POSIX I/O. While workers are waiting to do collective I/O after processing their portion of the query, they are wasting time, which shows up in the data distribution time.

Our second suite of tests held the number of processors constant at 64 and examined how improving the compute time would affect each I/O strategy. As mentioned in Section 1, improving the compute time could be accomplished in hardware (new processors, custom search hardware) or software (improved search algorithms).

Our first suite of tests in Figure 2 used compute speed = 1, (which we refer to as the base compute speed). This test suite (Figure 5 varied the compute speed parameter from 0.1 to 25.6. First of all, we note that increasing the compute speed up to 25.6 times faster than the base compute speed made less than a 2% difference in overall execution time for both the no-sync and sync cases for MW. Clearly, the application phases besides the compute phase are the bottleneck here. The other strategies faired much better than MW. The individual WW strategies (WW-List and WW-POSIX) both surpassed WW-Coll and MW significantly in the no-sync cases, indicating that individual WW strategies will strongly benefit from hardware or software improvements on the compute phase. In overall execution time with compute speed = 25.6, WW-List outperformed the other I/O strategies by 592% (MW), 32% (WW-POSIX), and 98% (WW-Coll) in the no-sync cases and by 444% (MW), 65% (WW-POSIX), and 58% (WW-Coll) in the sync cases. Similar to the first suite of tests, I/O times generally start slightly increasing as we improve the compute speed due to more I/O ops/s.

Figure 6 and Figure 7 show the overall breakdown of each run with respect to the different timing phases and dif-

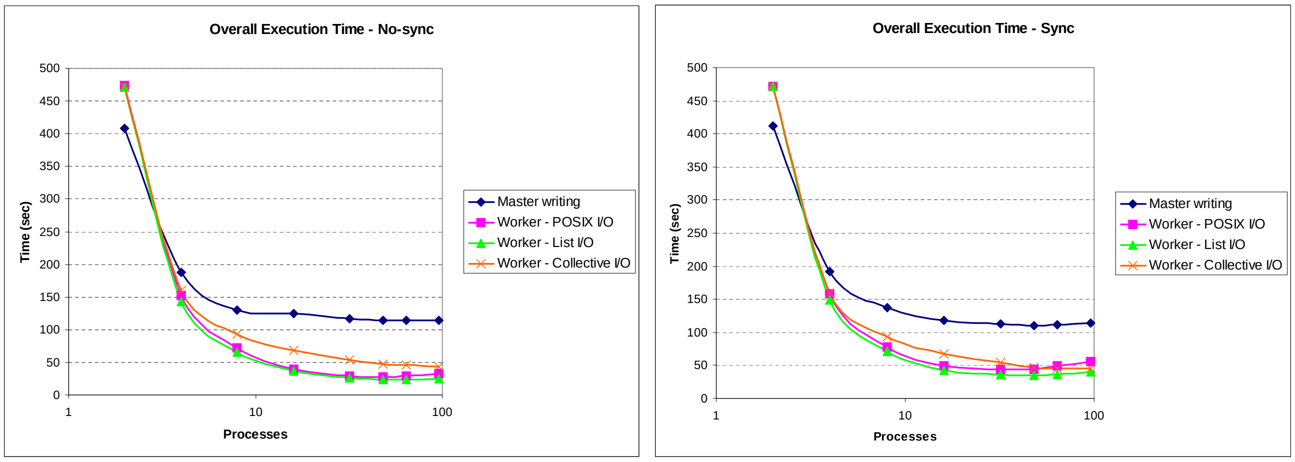

Figure 2. Results when scaling up the number of processors with no-sync/sync query options.

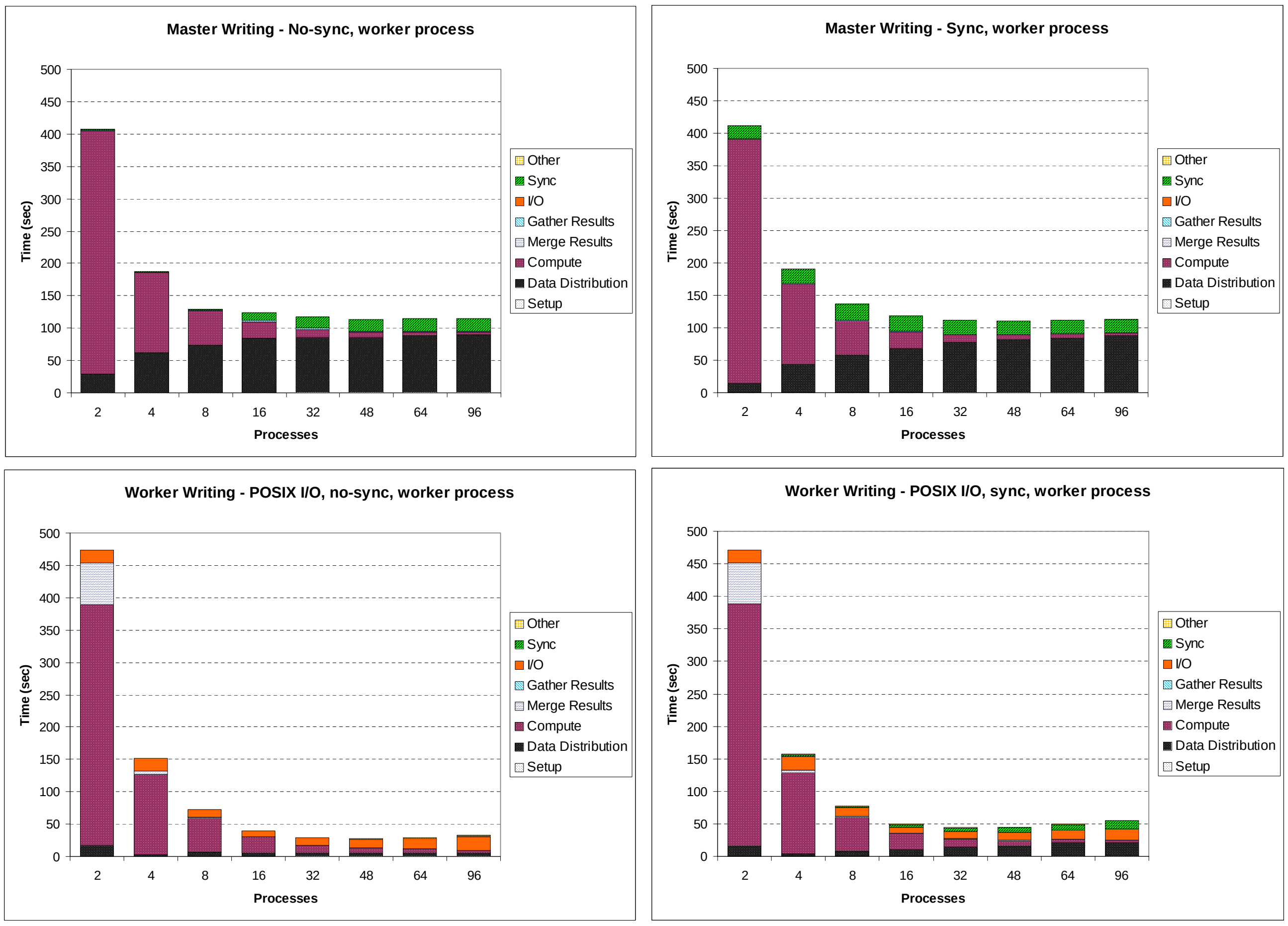

Figure 3. Individual phase timing results when scaling up the number of processors with no-sync/sync query options for MW and WW-POSIX.

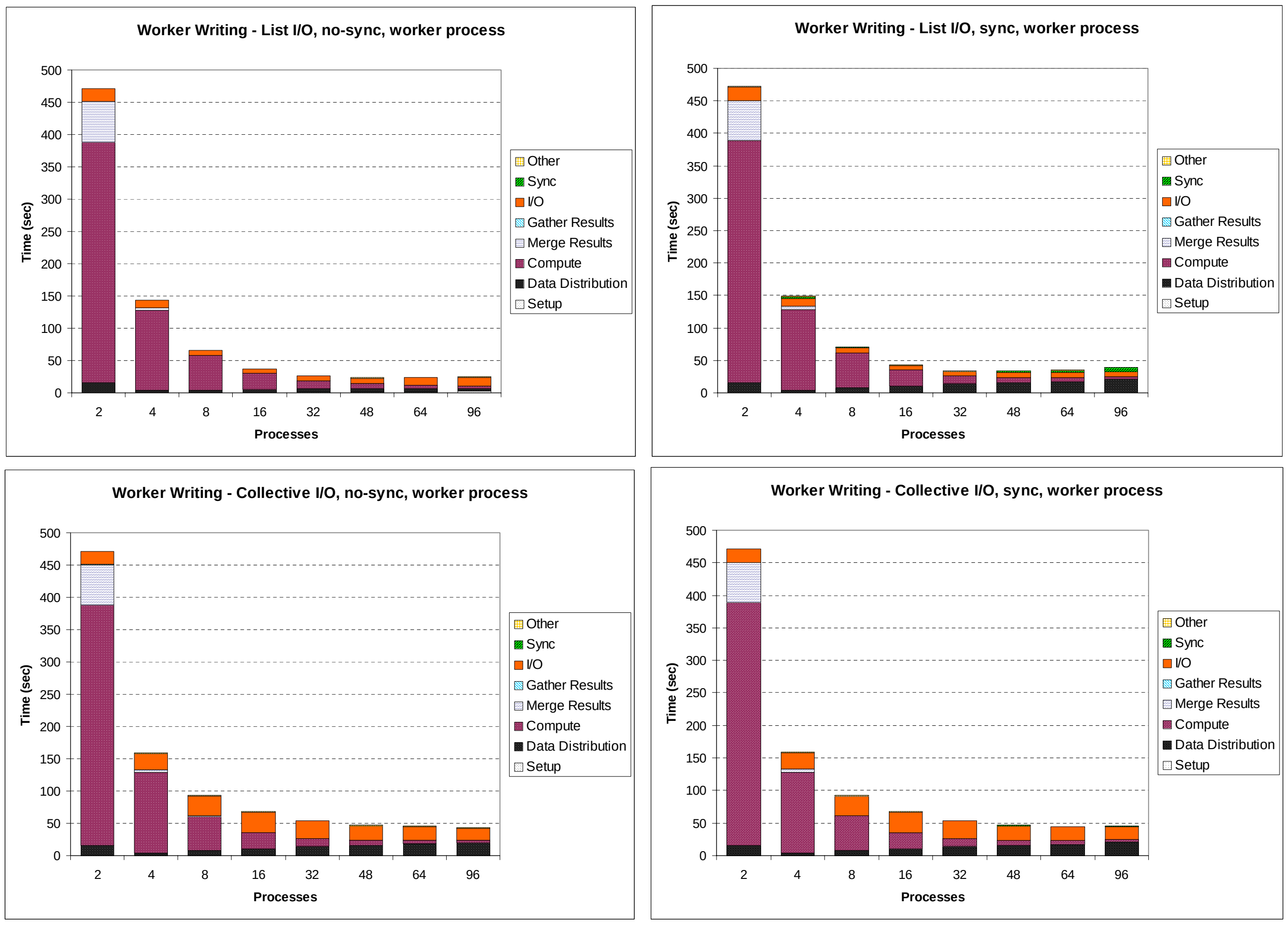

Figure 4. Individual phase timing results when scaling up the number of processors with no-sync/sync query options for WW-List and WW-Coll.

ferent I/O strategies. As the compute speed increases, we note that the effect of the compute speed on the overall execution time is reduced. At compute speed = 0.1, workers spend close to an average of 54 secs in the compute phase in both the no-sync and sync cases. At compute speed = 25.6, workers spend slightly more than 0.8 secs in the compute phase in both the no-sync and sync cases. The large variance in long compute phase times among workers leads to a large wait time when workers are synchronizing.

At slow compute speeds (0.1 to 0.4) with MW, forced synchronization adds some overhead (48% when compute speed = 0.1). The data distribution phase is mostly to blame as it causes 72.50 secs of the 75.47 secs difference at compute speed = 0.1. Since the absolute compute time variance among workers is high at compute speed = 0.1, the data distribution time is significantly affected. From compute speed = 0.8 to 25.6, synchronization makes little difference with MW (at most 2%).

WW-POSIX is substantially affected by the forced synchronization (at most 162% when compute speed = 0.1) from the high compute time variance. From compute speed = 1.6 to compute speed = 25.6, the overhead of forced synchronization is slightly above 50%. Since the compute time variances are less significant at this point (compute times are less than 4 secs when compute speed = 1.6 and 0.85 secs when compute speed = 25.6), most of the change in execution time is due to the synchronization overhead and the increased data distribution phase time. When compute speed = 25.6, sync phase time increases from 1.09 secs to 7.758 secs and data distribution time increase from 2.30 secs to 9.14 secs when going from no-sync to sync.

Similar to WW-POSIX, WW-List is strongly affected by the large compute time variance at low compute speeds. However, due to its optimized noncontiguous list I/O method, it incurs smaller overhead as sync phase time and data distribution phase time increase (when compute speed

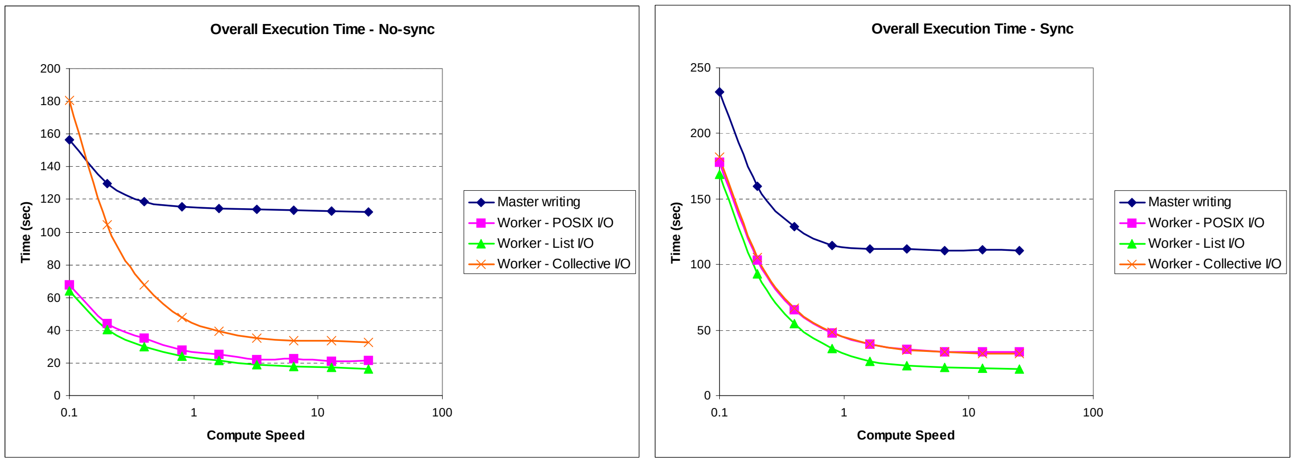

Figure 5. Results when scaling up the compute speed with no-sync/sync query options.

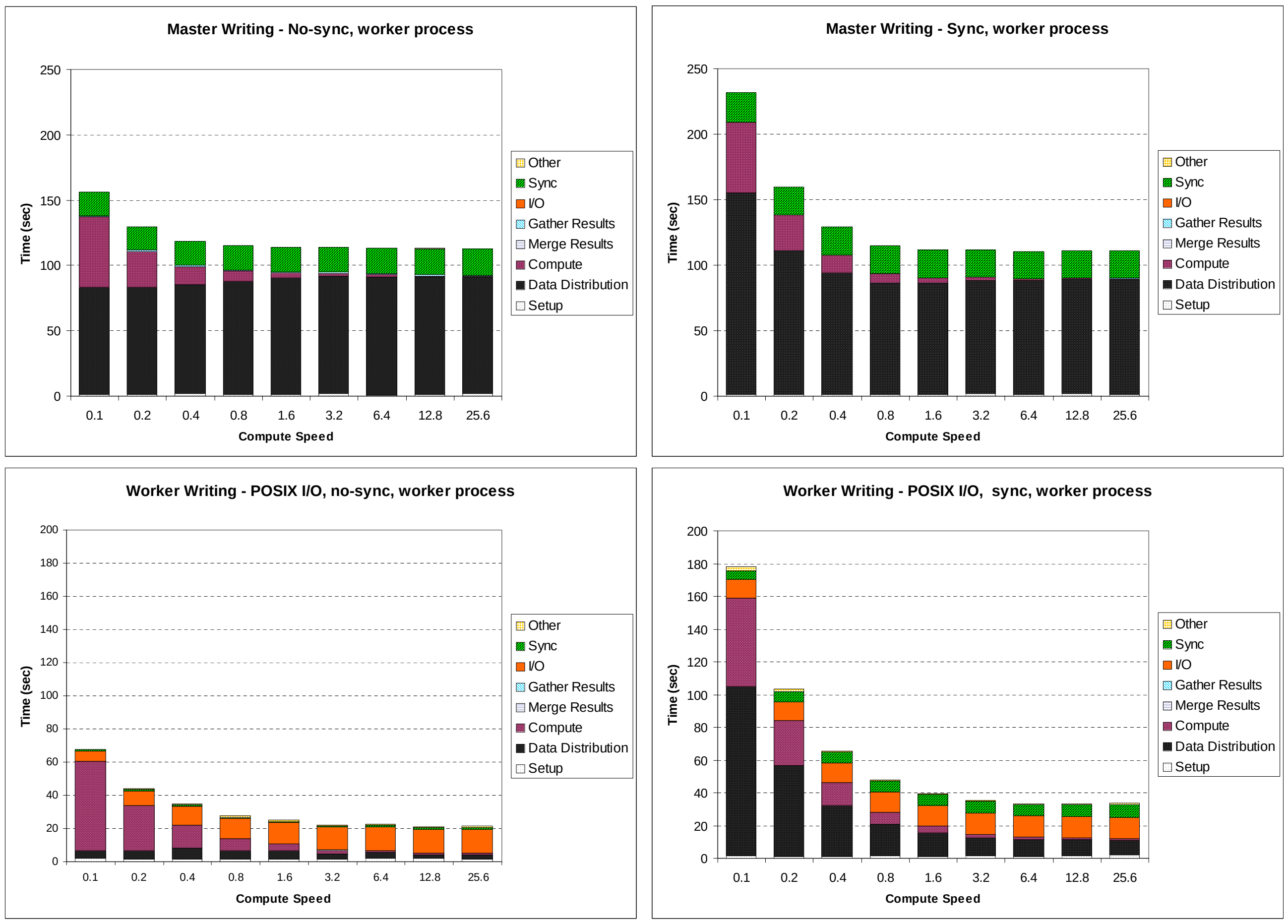

Figure 6. Individual phase timing results when scaling up the compute speed with no-sync/sync query options for MW and WW-POSIX.

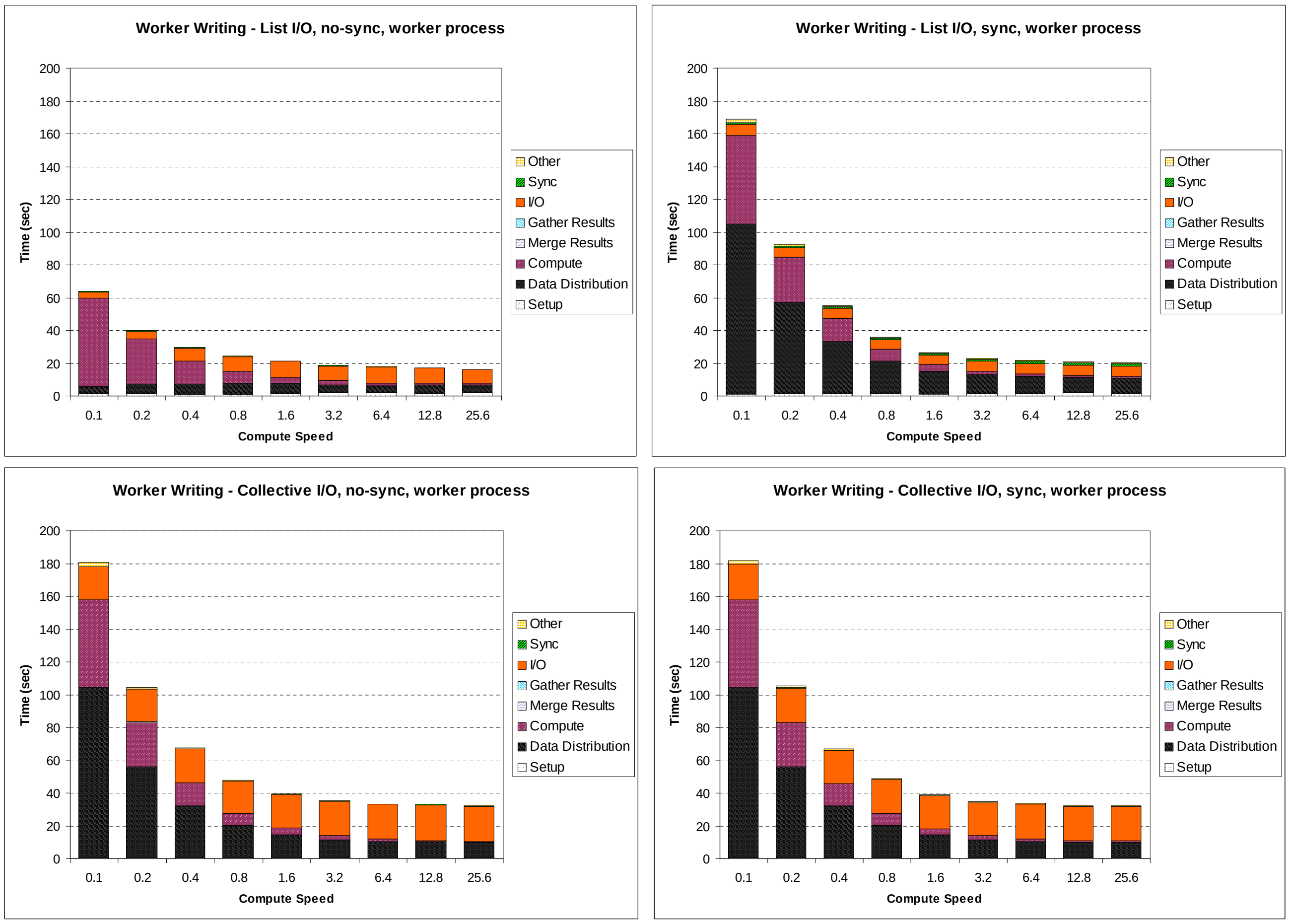

Figure 7. Individual phase timing results when scaling up the compute speed with no-sync/sync query options for WW-List and WW-Coll.

= 25.4, sync phase time increases from 0.35 secs to 1.36 secs and data distribution phase time increases from 3.96 secs to 9.36 secs) from no-sync to sync. The benefits of list I/O over POSIX I/O allow WW-List to provide improvements that show up in I/O phase time, sync phase time, and data distribution phase time.

WW-Coll is hardly affected when going from no-sync to sync (at most 4%). Again, since the inherent synchronization in collective I/O pays for the variance in compute times among workers, the trend of seeing relatively higher data distribution times as in the other I/O strategies is not present in the sync cases. In general, when compute time variance is large, WW-Coll always pays a high synchronization cost unlike individual WW strategies. Two phase I/O in ROMIO was not as efficient as list I/O with synchronization in almost all of our test cases. A collective I/O implementation based on list I/O might be appropriate for access patterns similar to parallel sequence-search.

5. Conclusion and Future Work

Our paper provides several contributions in understanding I/O strategies for parallel sequence-search tools. While these results are based on search applications similar to mpiBLAST and pioBLAST, the performance trends we observed can be extrapolated to other parallel sequence-search applications.

- **Proposed individual worker-writing I/O strategies for parallel sequence-search tools** - The individual WW strategies outperformed the other I/O algorithms in all no-sync test cases. WW-List beat all I/O methods in both no-sync and sync test cases.
- **Developed S3aSim for understanding phase interaction of parallel search algorithms** - S3aSim is a flexible tool for understanding how various I/O strategies perform when using database segmentation.

- **Detailed the possible cost of synchronization with collective I/O in a full application simulation** - To date, most I/O studies compared I/O methods in I/O-only benchmarks, which does not expose the I/O synchronization penalty in collective I/O. Our study also suggests that in some cases, a collective I/O method implemented with list I/O and forced synchronization may be a more efficient collective I/O method than the default two phase I/O method in ROMIO.

In the future, we would like to pursue further research on I/O strategies with S3aSim. There are many other input variables that can significantly affect overall application performance such as different I/O characteristics (different query sizes, databases, amount of results), hybrid query segmentation/database segmentation strategies, new I/O algorithms, as well as many others. We hope that our work will aid in the development of future parallel database search tools in conjunction with modern parallel file systems.

Acknowledgments

This work was supported in part by Los Alamos National Laboratory contract W-7405-ENG-36, DOE's SciDAC program (Scientific Data Management Center) award number DE-FC02-01ER25485, the NSF/DARPA ST-HEC program under grant CCF-0444405, and the DOE HPCSF program. We would also like to thank Sandia National Laboratories for allowing us to use their Feynman cluster.

References

[1] D. Benson, M. Boguski, D. Lipman, J. Ostell, B. Ouellette, B. Rapp, and D. Wheeler. GenBank. *Nucleic Acids Res.*, 27:12–17, 1999.

[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. C. Braun, K. T. Pedretti, T. L. Casavant, T. E. Scheetz, C. L. Birkett, and C. A. Roberts. Parallelization of local blast service on workstation clusters. *Future Gener. Comput. Syst.*, 17(6):745–754, 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] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[6] 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.

[7] A. Ching, A. Choudhary, W. keng Liao, R. Ross, and W. Gropp. Evaluating structured I/O methods for parallel file systems. In *International Journal of High Performance Computing and Networking*.

[8] A. Ching, A. Choudhary, W. keng Liao, R. Ross, and W. Gropp. Noncontiguous I/O through PVFS. In *Proceedings of the 2002 IEEE International Conference on Cluster Computing*, September 2002.

[9] A. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiBLAST. In *Proceedings of the 4th International Conference on Linux Clusters: The HPC Revolution*, 2003.

[10] M. F. Grant J.D., Dunbrack R.L. and O. M.F. Beo-Blast: distributed BLAST and PSI-BLAST on a Beowulf cluster. *Bioinformatics*, 18:765–766, 2002.

[11] P. Inc. Fast data finnder (fdf) and genematcher. http://www.paracel.com, 2000.

[12] T. Inc. Decypher. http://www.timelogic.com, 1996.

[13] W. Kent. BLAT - the BLAST-like alignment tool. *Genome Research*, 12(4), 2002.

[14] H. Lin, X. Ma, P. Chandramohan, A. Geist, , and N. Samatova. Efficient data access for parallel blast. In *Proceedings of 19th International Parallel and Distributed Processing Symposium*, 2005.

[15] C. Ltd. Bioccerator. http://eta.embl-heidelberg.de:8000/, 1994.

[16] L. M. Ma B, Tromp J. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[17] D. R. Mathog. Parallel BLAST on split databases . *Bioinformatics*, 19:1865–1866, 2003.

[18] NCBI. Growth of genbank. http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.

[19] M. J. Ning Z, Cox AJ. SSAHA: A Fast Search Method for Large DNA Databases. *Genome Res.*, 11(10):1725–1729, 2001.

[20] I. Sharapov. Computational applications for life sciences on sun platforms: Performance overview. Whitepaper, 2001.

[21] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.

[22] The parallel virtual file system 2 (PVFS2). `http://www.pvfs.org/pvfs2/`.

[23] C. T. White, R. K. Singh, P. B. Reintjes, J. Lampe, B. W. Erickson, W. D. Dettloff, V. L. Chi, and S. F. 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*, pages 504–509, Washington, DC, USA, 1991. IEEE Computer Society.

[24] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.