Efficient Data Access for Parallel BLAST

Heshan Lin * Xiaosong Ma † Praveen Chandramohan ‡ Al Geist § Nagiza Samatova ¶

Abstract

Searching biological sequence databases is one of the most routine tasks in computational biology. This task is significantly hampered by the exponential growth in sequence database sizes. Recent advances in parallelization of biological sequence search applications have enabled bioinformatics researchers to utilize high-performance computing platforms and, as a result, greatly reduce the execution time of their sequence database searches. However, existing parallel sequence search tools have been focusing mostly on parallelizing the sequence alignment engine. While the computation-intensive alignment tasks become cheaper with larger machines, data-intensive initial preparation and result merging tasks become more expensive. Inefficient handling of input and output data can easily create performance bottlenecks even on supercomputers. It also causes a considerable data management overhead. In this paper, we present a set of techniques for efficient and flexible data handling in parallel sequence search applications. We demonstrate our optimizations through improving mpiBLAST, an open-source parallel BLAST tool rapidly gaining popularity. These optimization techniques aim at enabling flexible database partitioning, reducing I/O by caching small auxiliary files and results, enabling parallel I/O on shared files, and performing scalable result processing protocols. As a result, we reduce mpiBLAST users' operational overhead by removing the requirement of prepartitioning databases. Meanwhile, our experiments show that these techniques can bring by an order of magnitude improvement to both the overall performance and scalability of mpiBLAST.

*Department of Computer Science, North Carolina State University

†Department of Computer Science, North Carolina State University; Computer Science and Mathematics Division, Oak Ridge National Laboratory. ma@csc.ncsu.edu

‡Computer Science and Mathematics Division, Oak Ridge National Laboratory

§Computer Science and Mathematics Division, Oak Ridge National Laboratory

¶Computer Science and Mathematics Division, Oak Ridge National Laboratory. samatovan@ornl.gov

1 Introduction

In the past decade, research in computational biology has been dramatically accelerated by the ever-increasing computational power, which allows computers to help people understand the composition and functional capability of biological entities and processes. A well-known outcome of the fusion between high-performance computing and high-throughput experimental biology was the assembly of the human genome by Celera Genomics using a cluster with nearly a thousand processors [11]. Computation-enabled breakthroughs like this have huge impacts on solving important problems in areas such as medicine and environmental science.

One computational biology area that has demanded and benefited from parallel computing is *sequence database search*. These tools search for similarities between given query sequences (*e.g.*, DNA, amino-acid sequences) and known sequences in a database. This helps people to identify the function of newly discovered sequences or to identify species of a common ancestor, and forms the foundation of more challenging tasks such as whole-genome alignment. As a key component of many bioinformatics research methods, popular sequence search tools such as BLAST [1] are used on daily basis by scientists.

For these scientists, the average search cost on a sequential machine is growing despite faster CPUs: with the help of new experimental technology and the Internet for data collecting/sharing, existing sequence databases are growing at a speed much faster than the average memory size equipped at a single computer node [7]. Searching databases that cannot fit into the main memory requires a large amount of I/O operations. Parallelizing database search is the obvious way to avoid exploding sequence search time. Fortunately, search tools such as BLAST perform pair-wise comparison and are embarrassingly parallel. In Section 2.1 we will give examples of parallel sequence search tools. In particular, the mpiBLAST implementation [7] is open-source, uses the portable communication library MPI [18], and runs on a number of parallel platforms. Since published in 2003, mpiBLAST has been steadily attracting new users.

However, although tools such as mpiBLAST can almost linearly shorten the computationally intensive sequence alignment process as more processors are used, unoptimized data handling can easily bring operational and management overhead, as well as creating new bottlenecks in large parallel executions. For example, mpiBLAST requires the biological database be *pre-partitioned* into many fragments, which are to be copied to individual nodes' local storage at a cluster for a parallel search. This pre-partitioning step increases operational overhead, creates a large number of small files, and may need to be re-performed if a larger number of processors are to be used. Another example is *result merging*. A parallel search that partitions the database needs to consolidate sequence segments found from individual database partitions for each query sequence, and report them by the degree of similarity to the query sequence. A non-scalable design of this communication and I/O intensive process may result in a result merging time much larger than the sequence alignment time, as we have observed in mpiBLAST runs. In addition, this bottleneck situation deteriorates as the database size, query size, result size, or machine size grows.

In this project, we investigate the end-to-end data flow in mpiBLAST and present a set of optimizations that enables efficient, as well as scalable data transfer and management. The main idea is to maximize the utilization of memory and shared file systems available on parallel computers, and explore parallelism in application components outside the BLAST sequence alignment engine. We demonstrate through experiments on multiple platforms that these optimizations can bring dramatic improvement in both overall performance and scalability. In addition, our solutions alleviate users from the potentially expensive and tedious database pre-partitioning tasks, and enable flexible, dynamic database distribution for future performance fine tuning.

The rest of this paper is organized as follows: section 2 discusses background information on parallel sequence search tools and mpiBLAST, as well as related work on parallel I/O. Section 3 presents our optimization approaches. Section 4 shows performance results and analysis. Section 5 discusses future work and Section 6 concludes the paper.

2 Background and related work

2.1 Parallel sequence search

Our research is closely related to studies in designing high-throughput biological sequence search programs, especially parallel search programs or execution methods [2, 3, 4, 6, 7, 10, 12, 13, 15, 16, 20, 24]. Among them, hardware based BLAST accelerators [15] are normally not freely accessible. In addition, they are tied to specific hardware technologies and not portable. Our work is intended to optimize parallel sequence search software by reducing their non-search overhead in a portable manner.

Earlier work in parallel sequence search mostly adopts the *query segmentation* method [3, 4, 6], which partitions the sequence query set. This is relatively easy to implement and allows the BLAST search to proceed independently on different processors. However, as databases are growing larger rapidly, this approach will incur higher I/O costs and have limited scalability. Our paper follows the more recent trend of pursuing *database segmentation* [2, 7, 16], where databases are partitioned across processors. This approach better utilizes the aggregate memory space and can easily keep up with the growing database sizes.

2.2 mpiBLAST

Our optimization techniques for parallel BLAST will be built and evaluated in one portable existing implementation, the mpiBLAST tool developed at Los Alamos National Laboratory [7]. Among the several published parallel BLAST codes, this implementation reported the highest speedup, underwent the largest scalability tests, and was directly integrated with the NCBI toolkit [19].

The parallelism of mpiBLAST search is based on database segmentation. Before the search, the raw sequence database needs to be formatted, partitioned into fragments, and stored in a shared storage space. mpiBLAST organizes parallel processes into one master and many workers. The master uses a greedy algorithm to assign un-searched fragments to workers. Then the workers copy the assigned fragments to their local disks (if available) and perform BLAST search concurrently. Upon finishing searching one fragment, a worker reports its local results to the master for centralized result merging. The above process repeats until all the fragments have been searched. Once the master receives results from all the workers for a query sequence, it calls the standard NCBI BLAST output function to format and print out results to an output file. mpiBLAST achieves good speedup, especially when the number of processes is small or moderate, by fitting the database into main memory and eliminating repeated scanning of disk-resident database files. More mpiBLAST design details will be discussed in the rest of the paper.

2.3 Other related work

Our work complements the above existing efforts of exploring parallelism in data-intensive bioinformatics applications, *with a special focus on I/O and data management*. In particular, we experimented with using MPI-IO [17, 23], which has been traditionally utilized by large-scale parallel simulations, in biological sequence search codes. To our

knowledge, our proposed approaches of using collective I/O and dynamic database partitioning have not been studied in any previous research (the only existing work on parallel I/O for bioinformatics codes that we are aware of [26] is based on executing mpiBLAST on top of PVFS [5], a file system with parallel I/O support. The database still needs to be pre-partitioned.). Meanwhile, our optimization techniques are orthogonal to improvement on sequence search algorithms (*e.g.*, [9]), and will work together with these improvements to further enhance search applications' overall performance. In fact, optimization on the search algorithms reduces computation costs and highlights the requirement for more efficient data access and memory management schemes.

In addition, we plan to incorporate methodologies from existing work on parallel data mining (*e.g.*, [21, 25]) and memory management for bioinformatics tools (*e.g.*, [8, 14]), in our future research for designing efficient result pruning and query batching.

3 Efficient parallel data handling for BLAST: design and implementation

In this section, we present the major techniques that we use to enable efficient I/O and data handling in mpiBLAST. First, to illustrate the performance problem addressed by this paper, we timed the main components of an mpiBLAST run on one of our test platforms, an SGI Altix system at Oak Ridge National Lab (more details on machine configuration will be given in Section 4.1). The mpiBLAST version we used in our experiments throughout this paper is 1.2.1, the latest release available from the mpiBLAST web site.

Our primary research goal is to *reduce non-search overheads in parallel BLAST*. mpiBLAST authors reported low overhead in 2003 using one to twenty five processes [7]. We repeated their experiments by searching the same query set against the same database on the aforementioned SGI Altix system. In Figure 1(a), we show the break down of search and non-search ("other") time in mpiBLAST's overall execution time, for tests using different number of processes. Although the database (GenBank `nt`) has approximately doubled its size since 2003, our results confirmed the mpiBLAST authors' results within the same range of process numbers. However, it is clear that the portion of non-search time grows steadily as more processes are performing the search in parallel. When the number of processes increases from 16 to 64, the portion of search time in the total execution time slips from 95.6% to 70.7%. Further, it is well recognized that the search time is very sensitive to input queries. Our experiments show that for searches against the smaller GenBank `nr` database using the default search threshold and randomly sampled query sequence sets of similar sizes as in the above test, a much more significant portion of time needs to be spent on non-search tasks (see Section 4).

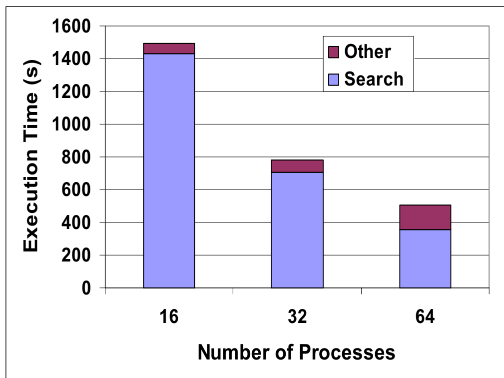
In the rest of this section, we present several techniques that reduce the costs of the non-BLAST-search components of mpiBLAST. We implemented these techniques in our enhanced version of mpiBLAST, called pioBLAST in this paper. The sequence search kernel is identical to that in mpiBLAST, which in turn builds on top of the NCBI BLAST Toolkit [19]. Given the same input query and database, pioBLAST and mpiBLAST generate the same output. As improving the utilization of available aggregate memory was one important motivation for database partitioning parallel BLAST tools such as mpiBLAST, our discussion in this paper will be focused on the scenario where the aggregate memory is large enough to accommodate the database files and intermediate search results. This is also easily satisfied with today's parallel computers.

3.1 Direct global database access and dynamic partitioning

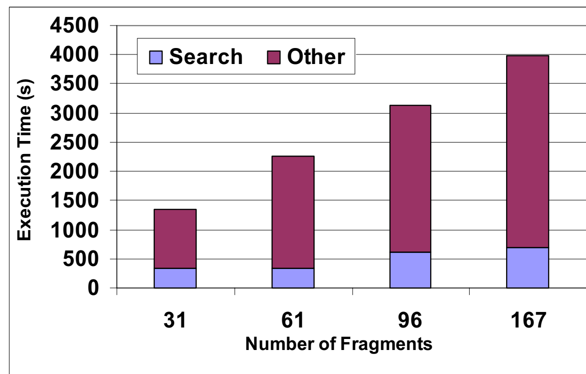
Raw sequence databases, usually stored in widely used FASTA format, need to be indexed before they can be searched by BLAST. This is normally done through the `formatdb` tool included in the NCBI BLAST toolkit. `formatdb` scans through the raw FASTA database and produces a set of sequence and index files. In a BLAST execution, these files are searched against and the raw database is not used. Note that users searching repeatedly against the same database only need to perform the `formatdb` preprocessing once.

mpiBLAST provides a tool called `mpiformatdb`, a wrapper of the standard `formatdb`, which integrates the database formatting and partitioning tasks. `mpiformatdb` is a sequential tool. With `mpiformatdb`, users can specify the number of *fragments* to partition the database into. This will generate n sets of output files, where n is the user-specified number of fragments. At run time, if there are local disks attached to the individual nodes, these database fragments will be copied by processors that they are assigned to from shared storage to the private local storage space of these processors.

This static partitioning approach has a few drawbacks. First, it creates a large number of smaller files, which are harder to manage, migrate, and share. Second, the database needs to be re-partitioned if the number of processors to be used in an mpiBLAST search exceeds the number of existing fragments. This brings inconvenience to users. Second, as `formatdb` is relatively expensive by itself (*e.g.*, it took 6 minutes and 22 minutes on a head node of the SGI Altix system that we used, to pre-process and partition the 1GB `nr` database and the 11GB `nt` database respectively), the re-partitioning cost is not trivial. Third, with



(a) Distribution of execution time



(b) Performance sensitivity to number of fragments

Figure 1. mpiBLAST Performance

a given number of processes, mpiBLAST's performance is sensitive to the number of database fragments, as shown in mpiBLAST authors' previous paper [7]. This is confirmed by our experiments. Figure 1(b) depicts the change in mpiBLAST's execution time with 32 processes when using different numbers of pre-generated database fragments. This test searched an 150KB randomly sampled query set against the `nr` database. When the number of fragments increases, both the search time and non-search time rise. The increase in search time is largely due to the I/O cost embedded in the NCBI BLAST search kernel, which grows as the total candidate result size increases when more fragments are searched. The increase in non-search time is associated with the higher result merging and retrieving overhead. As a result, the overall performance degrades significantly as the number of fragments grows. Therefore, creating a large number of fragments for running on different number of processors is not a good option.

In pioBLAST, we use parallel I/O to access shared databases and eliminate static partitioning. With the help of index files and well structured sequence files generated by `formatdb`, pioBLAST can easily perform dynamic partitioning and avoid generating any *physical* database fragments. From the global index files, it is easy to identify the file offsets of a given database fragment. The target databases are first formatted and put in a shared file system. The master process calculates the file ranges for each partition and distributes them to the worker processes, in terms of (*start_offset, end_offset*) pairs. The worker processes then use such file ranges to read, in parallel, different segments of the global sequence and index files into their local memory using MPI-IO. This way, one set of global formatted database files can be partitioned dynamically into an arbitrary number of virtual fragments at execution time. This *virtual partitioning* resembles the process of array distribution in many parallel numerical simulations. We further added limited modifications to the NCBI BLAST engine that redirect accesses to disk files, used to hold database fragments in NCBI BLAST and mpiBLAST, to in-memory buffers filled in pioBLAST's parallel input stage.

3.2 Result caching and merging

For each input query sequence, a BLAST search produces "alignment hits", where a query sequence is aligned with sequences in the database. A controllable *E-value* determines how many alignments will be reported in the final output. The search hits are sorted by their *scores*, which measure the quality of each alignment [1]. When the database is partitioned, individual workers do not have enough information to determine whether their local results will qualify to be included in the global output. Hence the local search results need to be merged and filtered.

In our experiments on multiple platforms, the result merging phase incurs the highest overhead in mpiBLAST, and can easily grow to surpass the actual BLAST search time as more workers are used. The reason is that mpiBLAST uses a merging scheme that serializes the processing between workers. First, the workers send their local result alignments (pieces of their local database sequences that were found to align well with the query sequence) to the master. The master sorts these result alignments by scores indicating their degree of similarity to the query sequence. For each result alignment to appear in the global output, the master requests sequence information from the worker that submitted that alignment. This process is repeated, in serial, for all global result alignments.

In pioBLAST, we improved the mpiBLAST result merging phase. First, the workers cache sequence information

[1] For more details, see the BLAST paper [1].

that are potentially useful in the output as local results are discovered, which eliminates future accesses to multiple memory-cached database files to retrieve such information. Second, when workers submit their local results to the worker, they only submit data items that are used in the sorting and filtering process, *i.e.*, alignment identifications, necessary scores, and alignment output sizes. The alignment output size is computed by calling a modified NCBI BLAST output routine that redirects the formatted result data from file output to memory buffers. This greatly reduces the total message volume. After the master merges the local results and identify the global results, it notifies the workers of the selected alignments. Now knowing the subset of their qualified local hits, the workers can write out corresponding result buffers that they already prepared, saving these data's return trip to the master. This shifts the bulk of work in result processing to the workers, and allow output preparation to proceed in parallel.

3.3 Parallel output

Both mpiBLAST and pioBLAST produce search output as a single file, with output data organized by query sequences. For each query sequence, the output starts with a header containing search statistics, followed by result alignments. Just like Internet search engines output pages ranked by their relevance to user queries, BLAST outputs alignments ranked by the degree of similarity between the query sequence and database sequences, which is denoted by the alignment scores.

In mpiBLAST, all output is handled by the master, who invokes the NCBI BLAST's output routines to create the header and processes individual results alignment for every query sequence. The workers' processing power and I/O bandwidth are not utilized.

In pioBLAST, we parallelize file output again using MPI-IO. As mentioned in the previous section, the workers cache eligible local results in memory buffers and send the size of each result alignment output to the master. As the sizes of the header and each alignment output record are known to the master, it can compute the offset ranges for individual alignment output record and distribute this information to the workers. Finally, the workers work together to create an MPI file view, which defines the ranges in a shared file visible to all processes. Collective I/O is then used to enable all the workers to write out buffered results in parallel. This allows data scattered in *noncontiguous* source memory locations and destination file locations to be output in a single MPI-IO operation, where the MPI-IO library efficiently performs data shuffling to combine these scattered accesses into large, sequential writes.

Figure 2 illustrates the difference between pioBLAST and mpiBLAST in result handling. Note that in mpiBLAST, the communication between the master and the workers to retrieve sequence information and the preparation of the output buffers are both serialized by the master. In pioBLAST, the workers carry out most of the output job and work in parallel.

4 Performance results and analysis

In this section, we evaluate pioBLAST and compare it with mpiBLAST, in terms of performance and scalability.

In most of our experiments, we used the GenBank `nr` database, a protein type repository frequently searched against by bioinformatics researchers. The size of the raw `nr` database is nearly 1 GB, consisting of 1,986,684 peptide sequences. With today's typical cluster configuration, a GenBank database can easily fit into the aggregate memory of a small number of nodes. To better control the query output size, we created several input query sets, each containing a different number of query sequences, by randomly sampling the `nr` database itself.

Note that the previous mpiBLAST results were benchmarked using the GenBank `nt` database, which is currently about 11 GB. With databases of such a size, `formatdb` will partition the output sequence and index files into multiple volumes. At this point, pioBLAST does not handle multiple global database files, and there are two design alternatives in addressing this problem. One is to extend pioBLAST's parallel input function to read multiple global files simultaneously. This may bring performance benefits, but complicates pioBLAST's virtual partitioning and load balancing schemes. The other is to modify `formatdb` to output single-volume output files. This does not require any modification of pioBLAST, and simplifies file management, but may not be an option on file systems that do not support large files. We plan to make the design choice between the two alternatives after inquiring initial users of pioBLAST.

All experiments were performed in shared queues. Except for one or two out-liners, which were obviously caused by concurrent jobs, the variance we observed was in general very small, with a 95% confidence interval within $\pm$ 3% of the average. This applies to both mpiBLAST and pioBLAST, therefore we omit the error bars in result charts.

4.1 Results from an SGI Altix

The SGI Altix system that we used, Ram, resides at Oak Ridge National Lab. It has 256 Intel 1.5 GHz Itanium2 processors running Linux, each with 6 MB of L3 cache, 256K of L2 cache, and 32K of L1 cache. Each processor has 8 GB of memory, which combines into 2 Terabytes of total system memory. Both mpiBLAST and pioBLAST used the SGI XFS [22] parallel file system.

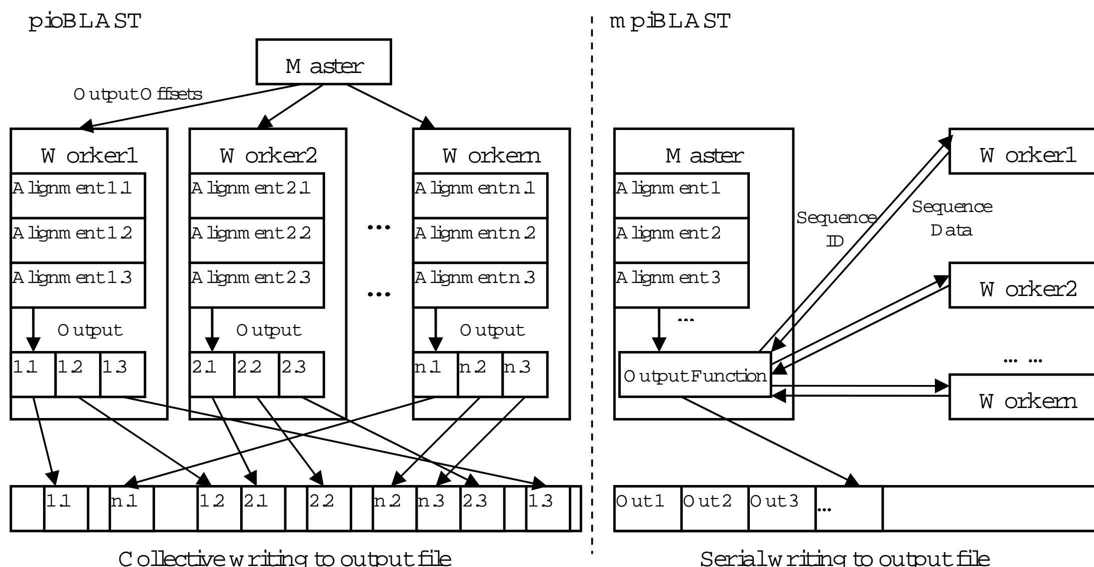


Figure 2. The output processes in pioBLAST and mpiBLAST.

	Copy/Input	Search	Output	Other	Total
mpiBLAST	17.1	318.5	1007.2	11.3	1354.1
pioBLAST	0.4	281.7	15.4	10.4	307.9

Table 1. Breakdown of execution time for mpiBLAST and pioBLAST searching a 150KB sequence query against the `nr` database.

First, we take a close look at a sample execution of mpiBLAST and pioBLAST, searching the same 150KB randomly sampled sequence query set against the `nr` database using 32 processes. For mpiBLAST, we pre-partitioned the database into 31 fragments, one for each worker process. This partitioning, which we call *natural partitioning*, is equivalent to pioBLAST's current default partitioning strategy. As discussed earlier in this paper, natural partitioning appears to result in the best overall performance for mpiBLAST, according to both the mpiBLAST authors' results [7] and our own experience. In the rest of the paper, all mpiBLAST results are benchmarked with natural partitioning.

Table 1 shows the breakdown of total execution time, for each program, between several major phases. mpiBLAST spends around 17 seconds on the *copy* stage, where it copies the database fragments into individual nodes' local storage. Note that on this Altix there is no local storage open to user jobs, so here mpiBLAST copies the data files to the shared job scratch space on XFS. pioBLAST does not have the copy stage. Instead, it has an input stage, where it performs on-the-spot partitioning and parallel input through MPI-IO. Each worker reads in one contiguous range from every shared database file. This takes pioBLAST less than half a second. At the end of this input stage, all pioBLAST worker have contiguous memory buffers holding their local portions of database files.

In the search stage, pioBLAST performs the same computation as mpiBLAST does, but saves the time of importing the local database partition into the memory. The BLAST search kernel from NCBI can use memory mapped files to input the disk-resident database. This can bring the benefits of implicitly overlapping I/O and computation, as well as avoiding subsequent I/O. mpiBLAST uses this BLAST feature, with its I/O performed implicitly through memory mapped files in the search stage. In pioBLAST, however, we slightly modified the NCBI BLAST kernel to work with the local database memory buffers explicitly filled in the input stage, instead of memory buffers mapped directly with disk-resident files.

The most significant improvement comes from the result processing stage. For both programs, with the default *e-value*, around 100MB of output data is produced. mpiBLAST spends 954 seconds on the result merging and output stage, due to its heavily serialized implementation. In particular, the "result fetching" process (not shown in the table), in which the master queries individual workers for result alignment data, accounts for more than 40% of the total output time. pioBLAST dramatically reduced the output time, to a total of 12.6 seconds, with its efficient result caching, merging, and parallel file writing.

The "other" category includes tasks not counted in the previous three columns. Examples of such tasks include query broadcasting, as well as the initialization and cleanup operations of the NCBI BLAST kernel.

In summary, pioBLAST significantly reduces the over-

Query size	26KB	77KB	159KB	289KB
Output size	11MB	47MB	96MB	153MB

Table 2. Query sizes and corresponding search output sizes

head of parallel searches. As a result, it improves the portion of time spent on the BLAST search from mpiBLAST's 24.5% to 95.5% for this particular search.

Next, we examine pioBLAST's scalability in two dimensions: parallelism and query/output data size.

To test how pioBLAST scales with an increasing number of processes and compares with mpiBLAST, we ran both programs using different numbers of processors. The query size is fixed and we used the same 150KB query as in the previous group of experiments, which will generate an output file of nearly 100MB. Natural partitioning is used for mpiBLAST and an equivalent virtual partitioning is used for pioBLAST. The total number of processes ranges from 4 to 64. However, since `mpiformatdb` cannot generate all arbitrary numbers of fragments, we could not partition the `nr` database into 63 fragments, but 61. Consequently we used 62 processes rather than 64 for both mpiBLAST and pioBLAST in the last group of tests.

The results are shown in Figure 3(a). We break down the total execution time into search and non-search (other) times. For both programs, the search time decreases nicely as the number of processes grows. However, in mpiBLAST the non-search time, most spent on result merging and output, increases steadily as more workers are used. When more than 31 workers are used, the increase in output time offsets the decrease in search time, causing the overall execution time to grow. With 61 workers, only 10.3% of mpiBLAST's total execution time is spent on the BLAST search. In contrast, with pioBLAST, a much smaller portion of time is spent on non-search tasks. In its case, as more workers are deployed the non-search time keeps *decreasing*, although at a rate slower than the search time decreases. From 32 total processes to 62, pioBLAST achieves an overall speed up of 1.86, and with 61 workers still 92.4% of its total execution time is spent on the BLAST search.

For data scalability tests, we fixed the number of processes to 62 for both programs. A series of synthetic query sequence sets of different sizes, again randomly sampled from the `nr` database, are used to generate different output sizes. Table 2 lists the corresponding query and output sizes.

Figure 3(b) shows the results, grouped by output size. As the query and output data size grows, both mpiBLAST's and pioBLAST's overall execution times scale roughly with the output size, so do their search times. The difference is that the total execution time is dominated by result output time in mpiBLAST, and by search time in pioBLAST. One notable detail not portrayed well by the small bars in Figure 3(b) is that the non-search time of pioBLAST less than doubled from the 11MB output to 153 MB output. This growing rate is much lower compared to that of mpiBLAST.

4.2 Results from an IBM Blade Cluster

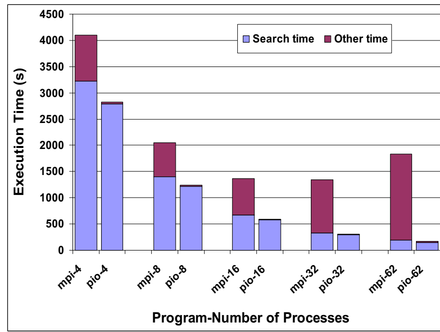
To evaluate pioBLAST's performance on a different parallel architecture, we repeated the process scalability experiments on the IBM Blade Cluster at the High Performance Computing Center, North Carolina State University. This Linux cluster has 64 nodes offering a total of 128 Intel Xeon 2.8-3.0 GHz Processors. Each blade node is equipped with 4 GB memory and 40 GB disk place. The shared file system is NFS.

Since this cluster is of a smaller scale and has significantly longer queue waiting time, we do the tests using up to 32 processes only. In general, Figure 4 shows the same trends as we observed on the ORNL Altix system. One important difference here is that the shared file system has significantly worse performance compared to XFS on the Altix. As a result, pioBLAST's portion of search time decreases from 93% with 4 processes to 64% with 32, a much worse deterioration compared to on the Altix, although still considerably milder than that of mpiBLAST (from 50% with 4 processes to 14% with 32). Here, since mpiBLAST's search time embeds a certain amount of I/O, its search time also does not scale well as more processes are used.

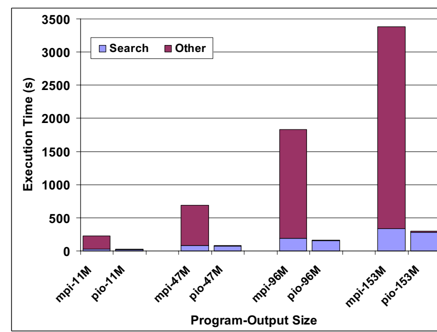
5 Discussion and future work

There are several directions along which we plan to extend pioBLAST.

First, at this point we have only implemented the simple natural partitioning strategy in pioBLAST. Since each worker accesses a single, sequential part of the global files, we use the individual I/O interfaces of MPI-IO in the input phase. pioBLAST's partitioning framework allows easy extension of natural partitioning into more sophisticated strategies through manipulating the file ranges. In particular, dynamic load balancing can be readily incorporated by assigning smaller file ranges to workers. Knowing that having too many fragments degrades the overall mpiBLAST performance, pioBLAST can adaptively find a compromise between load balancing and controlling communication overhead, by starting from coarse fragments and gradually refining the task granularity. Further, the file ranges can be decided at run time and differentiated between different workers, ideal for scenarios where we have heterogeneous nodes or skewed search.



(a) Node scalability



(b) Output scalability

Figure 3. Scalability tests on the ORNL SGI Altix

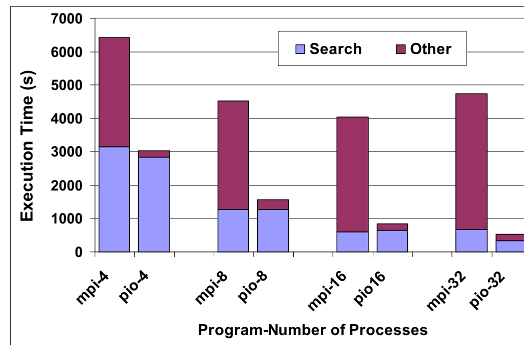


Figure 4. Process scalability results from the NCSU Blade Cluster

Second, when the number of workers grows, the size of local results remains constant and the total size of result alignments to be screened and merged by the master increases linearly. pioBLAST's result merging scheme can be further improved by early score communication. This communication may easily be added to broadcast the current global score threshold, so that workers can perform local pruning to stop processing for local results that fall under the global cut line.

In addition, even with efficient result pruning, our result caching and output schemes need to be adaptive to large output data sizes for tasks such as database-against-database BLAST and whole genome comparison. We plan to add adaptive approaches, such as query batching and pipelining that adjust to the amount of available memory, to use the aggregate memory more effectively in handling larger tasks.

6 Conclusion

In this paper, we proposed, designed, and evaluated a set of I/O and communication related optimizations for parallel sequence search applications. These optimizations show significant benefits in reducing both the search execution time and the management costs. Although we choose to implement and demonstrate our optimizations through mpiBLAST, the optimizations themselves are not specific to this particular implementation or even the BLAST algorithm. More specifically, our solutions (including dynamic input data partitioning, parallel I/O, and Dis-centralized result processing) are independent of search algorithms, file formats, or sequence data structures, making these solutions applicable to parallel search tools that adopt a database-partitioning approach in general.

We consider the major contributions of this paper to be as follows:

1. We demonstrated that input and output data handling is of great performance significance in parallel sequence search applications.

2. We enabled online, dynamic partitioning of shared databases, thereby eliminating the pre-partitioning process, which divides a database to static fragments. This offers low operational cost, reduced I/O, convenience in data storage and management, and more flex-

ibility in adjusting partitioning granularity.

3. We applied parallel I/O techniques that have been traditionally used in large-scale parallel simulations to bioinformatics applications, and such techniques proved to be effective in this new context. Especially, with the aids of collective and non-contiguous I/O provided by MPI-IO, we dramatically improved the output performance of parallel BLAST by parallelizing the originally serialized result processing procedure.

4. We designed and implemented schemes for aggressive result caching and for efficient result merging, both utilizing the available aggregate memory, CPU resources, interconnection bandwidth, and I/O bandwidth to improve the scalability of parallel sequence search.

5. We performed extensive performance evaluation and analysis on multiple architectures, using a combination of different query sizes and numbers of processors.

7 Acknowledgments

This work was funded in part or in full by the US Department of Energy's Genomes to Life program (www.doegenomestolife.org) under the ORNL-PNNL project, "Exploratory Data Intensive Computing for Complex Biological Systems", faculty startup funds from North Carolina State University, and a joint faculty appointment of Xiaosong Ma between NCSU and ORNL. The work of Heshan Lin is supported by the Scientific Data Management Center (http://sdmcenter.lbl.gov) under the Department of Energy's Scientific Discovery through Advanced Computing (DOE SciDAC) program (http://www.scidac.org). We gratefully acknowledge the Center for Computational Science of Oak Ridge National Laboratory, and the High Performance Computing Center at NCSU, for granting us accesses to the SGI Altix system and IBM Blade cluster respectively, and for providing technical support. We are also thankful to the developer teams of the following open source software: NCBI BLAST and mpiBLAST.

References

[1] S. Altschula, W. Gisha, W. Millerb, E. Meyersc, and D. Lipmana. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3), 1990.

[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] P. Carns, W. Ligon III, R. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[6] E. Chi, E. Shoop, J. Carlis, E. Retzel, and J. Riedl. Efficiency of shared-memory multiprocessors for a genetic sequence similarity search algorithm, 1997.

[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] R. de Carvalho Costa and S. Lifschitz. Database allocation strategies for parallel BLAST evaluation on clusters. *Distributed and Parallel Databases*, 13(1), 2003.

[9] S. Delaney, Gregory Butler, C. Lam, and L. Thiel. Three improvements to the BLASTP search of genome databases. In *Proceedings of the 12th International Conference on Scientific and Statistical Database Management*, 2000.

[10] J. Grant, R. Dunbrack Jr., F. Manion, and M. Ochs. BeoBLAST: distributed BLAST and PSI-BLAST on a Beowulf cluster. *Bioinformatics*, 18(5), 2002.

[11] G. Heffelfinger et al. Genomes to Life project proposal. http://www.genomes2life.org/SNL-ORNL-GTL-Proposal.doc.

[12] K. Hokamp, D. Shields, K. Wolfe, and D. Caffrey. Wrapping up BLAST and other applications for use on Unix clusters. *Bioinformatics*, 19(3), 2003.

[13] H. Kim, H. Kim, and D. Han. Hyper-BLAST: A parallelized BLAST on cluster system. In *Proceedings of the International Conference on Computational Science*, 2003.

[14] M. Lemos and S. Lifschitz. A study of a multi-ring buffer management for BLAST. In *Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, 2003.

[15] R. Luthy and C. Hoover. Hardware and software systems for accelerating common bioinformatics sequence analysis algorithms. *Biosilico*, 2(1), 2004.

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

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

[18] Message Passing Interface Forum. *MPI: Message-Passing Interface Standard*, June 1995.

[19] National Center for Biotechnology Information. NCBI BLAST. http://www.ncbi.nlm.nih.gov/BLAST/.

[20] National Institutes of Health. BLAT on Biowulf. http://biowulf.nih.gov/apps/blat.html.

[21] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. 22nd Int. Conf. Very Large Databases*, 1996.

[22] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Technical Conference*, 1996.

[23] 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.

[24] J. Wang and Q. Mu. Soap-HT-BLAST: high throughput BLAST based on web services. *Bioinformatics*, 19(14), 2003.

[25] M. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3), 2000.

[26] Y. Zhu, H. Jiang, X. Qin, and D. Swanson. A case study of parallel I/O for biological sequence analysis on Linux clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.