LA-UR-03-2862

Approved for public release;
distribution is unlimited.

c.1

Title: The Design, Implementation, and Evaluation of mpiBLAST

Author(s): Aaron Darling, Lucas Carey, and Wu-chun Feng

qq

Submitted to: ClusterWorld Conference & Expo 2003

LOS ALAMOS NATIONAL LABORATORY
3 9338 01045 2430

Los Alamos
NATIONAL LABORATORY

99

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Form 836 (8/00)

The Design, Implementation, and Evaluation of mpiBLAST

Aaron E. Darling[1], Lucas Carey[2], Wu-chun Feng[3]

[1] Dept. of Computer Science, University of Wisconsin, Madison WI 53703, USA,
darling@cs.wisc.edu,
[2] Center for Developmental Genetics, SUNY Stony Brook, USA
lcarey@odd.bio.sunysb.edu
[3] Advanced Computing Laboratory, Los Alamos National Laboratory, USA
feng@lanl.gov

Abstract. mpiBLAST is an open-source parallelization of BLAST that achieves superlinear speed-up by segmenting a BLAST database and then having each node in a computational cluster search a unique portion of the database. Database segmentation permits each node to search a smaller portion of the database, eliminating disk I/O and vastly improving BLAST performance. Because database segmentation does not create heavy communication demands, BLAST users can take advantage of low-cost and efficient Linux cluster architectures such as the bladed Beowulf [8, 16]. *In addition to presenting the software architecture of* mpiBLAST, we presenta detailed performance analysis of mpiBLAST to demonstrate its scalability.

1 Introduction

The BLAST family of sequence database-search algorithms serves as the foundation for much biological research. The BLAST algorithms search for similarities between a short query sequence and a large, infrequently changing database of DNA or amino acid sequences [1, 2]. Newly discovered sequences are commonly searched against a database of known DNA or amino-acid sequences. Similarities between the new sequence and a gene of known function can help identify the function of the new sequence. Other uses of BLAST searches include phylogenetic profiling and pairwise genome alignment. Unfortunately, traditional approaches to sequence homology searches using BLAST have proven to be too slow to keep up with the current rate of sequence acquisition [12].

Because BLAST is both computationally intensive and embarrassingly parallel, many approaches to parallelizing its algorithms have been investigated [4, 5, 7, 10, 13–15]. We present an open-source parallelization of BLAST that segments and distributes a BLAST database among cluster nodes such that each node searches a unique portion of the database.

Database segmentation in BLAST offers two primary advantages over other parallel BLAST algorithms. First, database segmentation can eliminate the high overhead of disk I/O. The sizes of bioinformatic databases are now larger than

core memory on most computers, forcing BLAST searches to page to disk. Database segmentation permits each node to search a smaller portion of the database, thus reducing (or even eliminating) extraneous disk I/O, and hence, vastly improving BLAST performance. With sequence databases doubling in size each year, the problem of extraneous disk I/O is expected to persist. The adverse effects of disk I/O are so significant that BLAST searches using database segmentation can exhibit super-linear speedup versus searches on a single node.

Second, database segmentation in mpiBLAST does not produce heavy intercommunication between nodes, allowing it to continue achieving super-linear speedup over hundreds of nodes. Consequently, scientists using BLAST with database segmentation can take advantage of low-cost and highly efficient Linux clusters such as Green Destiny [8, 16]

mpiBLAST, an open-source parallelization of BLAST, uses the Message Passing Interface [11] (version 1) to implement database segmentation, allowing it to work on diverse system architectures. mpiBLAST has been designed to run on clusters with job-scheduling software such as PBS (Portable Batch System). In such environments, it adapts to resource changes by dynamically re-distributing database fragments.

2 The BLAST Algorithm

BLAST searches a query sequence consisting of nucleotides (DNA) or peptides (amino acids) against a database of nucleotide or peptide sequences. Because peptide sequences result from ribosomal translation of nucleotides, comparisons can be made between nucleotide sequences and peptide sequences. BLAST provides functionality for comparing all possible combinations of query and database sequence types by translating the sequences on the fly. Table 1 lists the names used to refer to searches on each possible combination of query versus database type.

Table 1. BLAST search types

Search Name	Query Type	Database Type	Translation
blastn	Nucleotide	Nucleotide	None
tblastn	Peptide	Nucleotide	Database
blastx	Nucleotide	Peptide	Query
blastp	Peptide	Peptide	None
tblastx	Nucleotide	Nucleotide	Query and Database

The algorithms for each type of search operate nearly identically. The BLAST search heuristic [1] indexes both the query and target (database) sequence into words of a chosen size (11 nucleotides or 3 residues by default). It then searches for matching word pairs (hits) with a score of at least T and extends the match along the diagonal. Gapped BLAST [2] consists of several modifications to

the previous algorithm that result in both increased sensitivity and decreased runtime. Gapped BLAST (hereafter referred to simply as BLAST) moves down the sequences until it has found two hits, each with a score of at least T, within A letters of each other. An ungapped extension is performed on the second hit, generating a 'high-scoring segment pair' (HSP). If the HSP score exceeds a second cutoff, a gapped extension is triggered simultaneously forward and backward. Standard BLAST output consists of a set of local gapped alignments found within each query sequence, the alignment's score, an alignment of the query and database sequences, and a measure of the likelihood that the alignment is a random match between the query and database (*e-value*).

3 Related Work

3.1 BLAST Hardware Parallelization

Parallelization at the hardware level takes place during the sequence alignment itself. Such techniques are capable of parallelizing the comparison of a single query sequence to a single database entry, but require custom hardware with a greater degree of parallelization than is present in symmetric multi-processor (SMP) or symmetric multi-threaded (SMT) systems. The first hardware BLAST accelerator was reported by R.K. Singh [15]. More recently, TimeLogic [14] has commercialized an FPGA-based accelerator called the DeCypher BLAST hardware accelerator.

3.2 Query Segmentation

Query segmentation splits up a set of query sequences such that each node in a cluster or CPU on an SMP system searches a fraction of the query sequences. By doing so, several BLAST searches can execute in parallel on different queries. BLAST searches using query segmentation on a cluster typically replicate the entire database on each node's local storage system [4, 5]. If the database is larger than core memory, query-segmented searches suffer the same adverse effects of disk I/O as traditional BLAST. When the database fits in core memory, however, query segmentation can achieve near linear scalability for all BLAST search types, even on SMP architectures [7].

3.3 Database Segmentation

In database segmentation, independent segments of the database are searched on each processor or node, and results are collated into a single output file. Several implementations of database segmentation exist, the first of which was within NCBI's BLAST itself. NCBI-BLAST implements database segmentation by multithreading the search such that each processor in an SMP system is assigned a distinct portion of the database.

Database segmentation has also been implemented in a closed-source commercial product by TurboWorx, Inc. called TurboBLAST [3, 6]. TurboBLAST

provides ada tabase segmentation and distribution mechanism explicitly designed for use on networks of workstations. By using TurboWorx's proprietary TurboHub scheduling and load balancing software, TurboBLAST dynamically adapts to the current cluster environment. However, its proprietary implementation only results in linear speed-up (see `http://www.turboworx.com/products/turboblast_overview.html`). Furthermore, a recent survey on bioinformatics and Linux clusters (see `http://bioinformatics.org/pipermail/bioclusters/2002-October/000432.html`) shows that *none* of the sample population uses this distribution, primarily because of its exorbitant cost and its proprietary nature, which makes it difficult to integrate with other bioinformatics codes.

Recently another implementation of database segmentation was released at `ftp://saf.bio.caltech.edu/pub/software/molbio/parallelblast.tar`. `parallelblast` is composed of a set of scripts that operate in the Sun Grid Engine/PVM environment. Aside from requiring the SGE/PVM environment, it also differs from mpiBLAST in that it is not directly integrated with the NCBI toolkit and does not explicitly provide a load-balancing mechanism.

4 mpiBLAST Algorithm

The mpiBLAST algorithm consists of two primary steps. First, the database is segmented and placed on a shared storage device. Second, mpiBLAST queries are run on each node. If a node does not yet have a database fragment to search, it copies a fragment from shared storage. Fragment assignments to each node are determined by an algorithm that minimizes the number of fragment copies during each search.

4.1 Formatting and Querying the Database

Database formatting is done by a wrapper for the standard NCBI formatdb called mpiformatdb. mpiformatdb formulates the correct command line arguments to cause NCBI formatdb to format and divide the database into many small fragments of approximately equal size. Additional command line parameters to mpiformatdb allow the user to specify the number of fragments or the fragment size. Upon successful completion of formatdb, the formatted fragments are placed on shared storage.

Querying the database is accomplished by directly executing the BLAST algorithm as implemented in the NCBI development library available at `ftp://ftp.ncbi.nih.gov/toolbox/ncbi_tools/`. Upon startup, each worker process reports to the master process which database fragments it already has on local storage. Next, the master process (that with rank 0), reads the query sequences from disk and broadcasts them to all processes in the communication group. When the query broadcast has completed, each process reports to the master that it is idle. The master, upon receiving an idle message, assigns the idle worker a database fragment to either search or copy. The worker copies or

Algorithm 1 mpiBLAST master

Let *results* be the current set of BLAST results
Let $\mathbf{F} = \{f_1, f_2, ...\}$ be the set of database fragments
Let **Unsearched** $\subseteq \mathbf{F}$ be the set of unsearched database fragments
Let **Unassigned** $\subseteq \mathbf{F}$ be the set of unassigned database fragments
Let $\mathbf{W} = \{w_1, w_2, ...\}$ be the set of participating workers
Let $\mathbf{D}_i \subseteq \mathbf{W}$ be the set of workers that have fragment f_i on local storage
Let **Distributed**=$\{\mathbf{D}_1, \mathbf{D}_2, ...\}$ be the set of $\mathbf{D}$ for each fragment
Require: $|\mathbf{W}| \neq 0$
Ensure: $|\textbf{Unsearched}| = 0$
  Unsearched $\leftarrow$ **F**
  Unassigned $\leftarrow$ **F**
  results $\leftarrow \emptyset$
  Broadcast queries to workers
  while $|\textbf{Unsearched}| \neq 0$ **do**
    Receive a *message* from a worker w_j
    if *message* is a state request **then**
      if $|\textbf{Unassigned}| = 0$ **then**
        Send worker w_j the state $SEARCH\_COMPLETE$
      else
        Send worker w_j the state $SEARCH\_FRAGMENT$
      end if
    else if *message* is a fragment request **then**
      Find f_i such that $\min_{\mathbf{D}_i \in \textbf{Distributed}} |\mathbf{D}_i|$ and $f_i \in$ **Unassigned**
      if $|\mathbf{D}_i| = 0$ **then**
        Add w_j to $\mathbf{D}_i$
      end if
      Remove f_i from **Unassigned**
      Send fragment assignment f_i to worker w_j
    else if *message* is a set of search results for fragment f_i **then**
      Merge *message* with *results*
      Remove f_i from **Unsearched**
    end if
  end while
  Print *results*

searches its assigned fragment and reports to the master that it is idle when complete. This process is repeated until all database fragments have been searched.

The master process uses a greedy algorithm to determine which fragments to assign each worker. First, if the idle worker has any unsearched fragments that no other worker has on local storage, the worker is assigned to search the unique fragment. If a worker has no unique fragment, the worker is assigned the unsearched fragment which exists on the smallest number of other workers. Finally, if an idle worker has no unsearched fragments, it is told to copy the unsearched fragmentexi sting on the fewest other workers. The set of fragments currently being copied is tracked by the master to prevent duplicate copy assignments to different workers.

Algorithm 2 mpiBLAST worker

```
queries ← Receive the queries from the master
currentState ← Receive the state from the master
while currentState ≠ SEARCH_COMPLETE do
  currentFragment ← Receive a fragment assignment from the master
  if currentFragment is not on local storage then
    Copy currentFragment to local storage
  end if
  results ← BLAST(queries, currentFragment)
  Send results to master
  currentState ← Receive the state from the master
end while
```

When each worker completes a fragment search, it reports the results to the master. The master merges the results from each worker and sorts them according to their score. Once all results have been received, they are written to a user-specified output file using the BLAST output functions of the NCBI development library. This approach to generating merged results permits mpiBLAST to directly product results in any format supported by NCBI-BLAST, including XML, HTML, tab delimited text, and ASN.1.

5 mpiBLAST Performance

NCBI-BLAST and mpiBLAST have been benchmarked on several systems in an effort to characterize their performance and scalability. We first present the performance of NCBI-BLAST when the database is larger than core memory, demonstrating a significant decrease in performance caused by additional disk I/O. Next, we show that mpiBLAST (with its database-segmenting technique) achieves superlinear speed-up on multiple nodes when the database is larger than the core memory of a single node. We continue by assessing the scalability of mpiBLAST to many nodes. Then, we present the additional running time incurred by various components of the mpiBLAST algorithm as it scales.

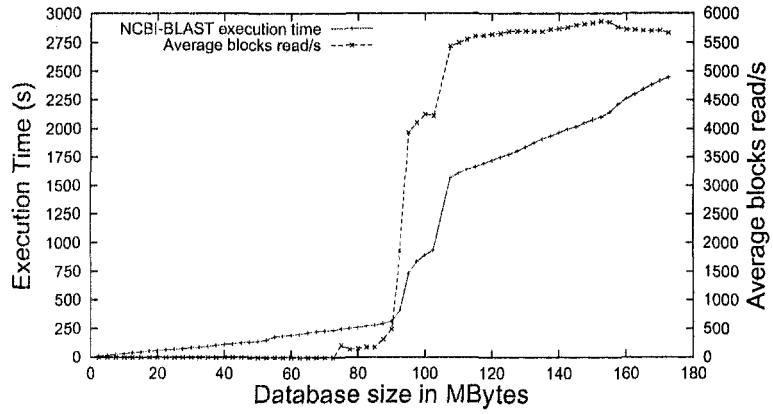

Fig. 1. (left) The performance of a blastn search using NCBI-BLAST when run on a system with 128MB RAM. As the database size grows larger than core system memory total running time increases rapidly.A sharp increase in disk I/O is also observed when the database is large because it is no longer entirely cached by the system.

Our benchmarking methods have been carefully designed to accurately reflect a typical usage pattern by molecular biologists.

5.1 Benchmarking Methods

When benchmarking BLAST search performance, decisions about the type of search to perform can significantly influence timing results. Factors such as query length, number of queries, total database size, length of database entries, and sequence similarity between the query and database entries affect the amount of time consumed by the BLAST algorithm. [7] Each factor must be carefully considered if the benchmarks are to accurately reflect typical BLAST usage patterns by molecular biologists.

We have endeavored to perform benchmarks that model the typical usage of BLAST when integrated into a high throughput genome sequencing and annotation pipeline. When used in this context, each BLAST query is a predicted gene in a newly sequenced organism. The BLAST search results are used to assist human annotators in determining the biological role of each predicted gene. [9] Because many organisms have thousands of genes, the large number of search queries generated by genome sequencing and annotation projects demand heavy computation. We have chosen to model this scenario because sequencing and annotation projects can benefit from mpiBLAST's improved BLAST performance.

The benchmarks described in the following sections utilize predicted genes from a newly sequenced bacterial genome as BLAST queries. The query gene lengths are approximately exponentially distributed with a mean $\theta = 747.2$ base pairs and standard deviation $\sigma = 684.2$. The database sequences are taken from

the GenBank nt database, a large public repository of non redundant nucleotide sequences. Ignoring a small number of outliers whose length is greater than 25,000 bp, the length of the nt database entries can also be reasonably approximated by an exponential distribution where $\theta = 1370$.

5.2 Low Memory Performance

NCBI-BLAST was benchmarked on a system with 128MB memory using increasingly large database sizes to determine the effect of databases that do not fit in core memory. Each run measured the total running time of a blastn search using the same set of query sequences against a larger database. Weuti lized Linux's BSD process-accounting facilities to collect system-activity statistics.

Figure 1 shows total BLAST run times alongside the average blocks read per second from the disk for each database size tested. Formatted BLAST databases are compressed versions of the raw sequence databases. A formatted nucleotide database consumes approximately 25% as much space as a text file containing the sequences. As the database size exceeds the total system memory size, BLAST running times and average blocks read per second increase sharply. Because the operating system can not cache the entire database BLAST must wait for it to be reread from disk when processing each query sequence.

Like NCBI-BLAST, the performance of mpiBLAST suffers when confronted with low memory conditions. However, because mpiBLAST effectively uses the aggregate memory of all worker nodes, the database can grow much larger before causing extra disk I/O.

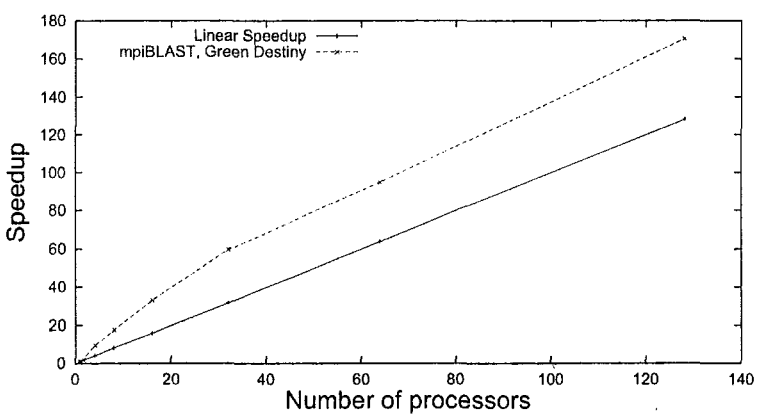

Fig. 2. Speedup of mpiBLAST on Green Destiny. 300kb of query sequences were searched against a 5.1-GB database. The size of the formatted database is approximately 1.2 GB, much larger than the 640-MB core memory per node. The search causes heavy disk I/O when a single node is used.

To get an overview of scalability when the database is larger than a single node's core memory, we benchmarked mpiBLAST on Green Destiny [8, 16]. Green Destiny is a 240-node bladed Beowulf cluster based on the Transmeta Crusoe processor. Each compute node consists of a 667-MHz TM5600, 640MB RAM, 100-Mb/s Ethernet, and a 20-GB hard drive running under Linux 2.4.

Figure 2 shows mpiBLAST performance measurements taken on Green Destiny. Fragments of a 5.1-GB uncompressed database were pre-distributed to each worker and a short query was executed to prime the buffer-cache. By priming the cache, we hope to simulate the case when the cluster is processing many BLAST queries in quick succession. Each timed run used 300Kbytes of predicted gene sequences.

The single worker search consumed 22.4 hours whereas 128 workers completed the search in under 8 minutes. Relative to this single-worker case, mpiBLAST achieved super-linear speedup in all cases tested. However, as the number of workers increases the efficiency of mpiBLAST decreases.

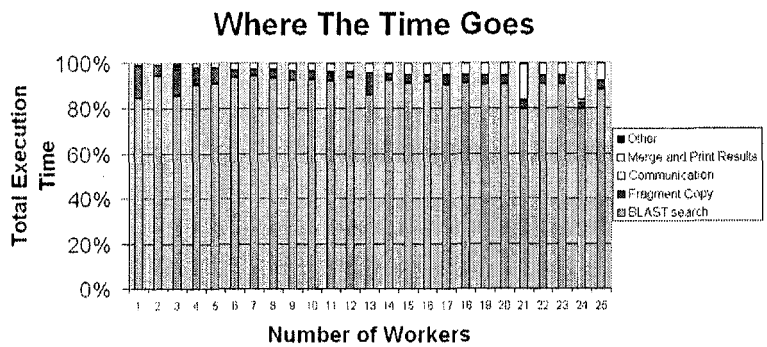

Fig. 3. How time is spent in mpiBLAST. Each bar is a composite that shows how time was spent on the longest running worker node in addition to the time spent merging results by the master node. Total execution time is largely dominated by BLAST search time.

5.3 Where does the time go?

The decrease in efficiency observed when scaling mpiBLAST to many nodes leads us to ask "What is mpiBLAST doing with the extra time?" mpiBLAST's running time can be decomposed into five primary components: (1) MPI and mpiBLAST initialization, (2) database-fragment copying time, (3) BLAST search time, (4) communication time, and (5) result merging and printing time. In order to determine how each component contributes to the total execution time, we profiled

mpiBLAST with the MPE library to collect wall-clock timing statistics and used `gprof` to measure CPU usage.

Measurements were taken on systems located in the Galaxy cluster at SUNY Stony Brook. Each node contains dual 700-MHz Pentium III processors with 1-GB PC133 SDRAM, 100-Mb/s Ethernet connected to a Foundry Networks Big Iron 8000 switch, and a 20-GB hard drive.

Two gigabytes of the nt database were formatted into 25 fragments. Each run measured the components of execution time on 1 through 25 workers using the same set of database fragments and an 10-kb query of predicted ORF sequences. Figure 3 shows the contribution of each component to the total running time of mpiBLAST. Based on these measurements, we conclude that for small numbers of workers, execution time is dominated by BLAST searches. As more workers are utilized, the time spent formatting and writing results grows relative to total execution time. Communication consistently accounts for less than 1% of the total execution time.

Although some workers may finish before others during the search phase, the master waits until all workers have completed before formatting the results. Thus, the total execution time is dependent on the longest running worker. Each bar in Figure 3 shows the run-time of components of the longest running worker in addition to the time spent formatting by the master in order to accurately reflect the components of the total execution time.

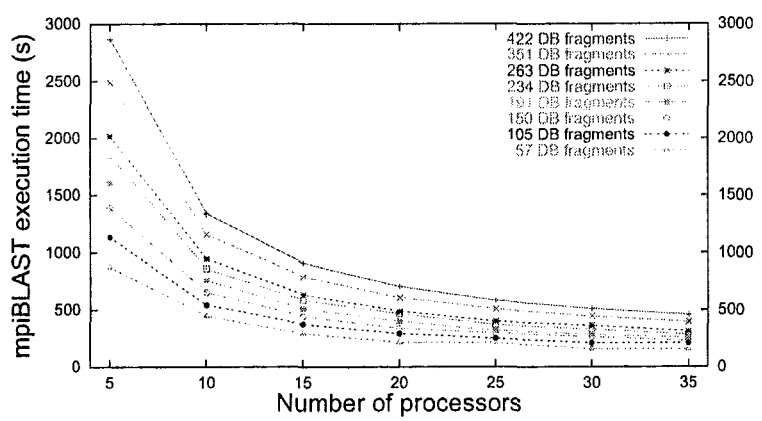

Fig. 4. The overhead of performing the same mpiBLAST search increases with the number of database fragments used. Each measurement of running time (y-axis) was taken by formatting an identical database with a varying number of fragments. The unusual numbers of database fragments arise because NCBI formatdb's segmentation method tries to guarantee a maximum fragment size, not a particular number of fragments.

The measurements discussed here were taken by searching the same 25 fragment database with a variable number of workers. In a search using a single worker, all 25 fragments would be assigned to the same worker. When searching with 25 workers, each worker searches a single fragment. However, when searching with some number of workers that is not an even divisor of the number of fragments, an imbalance in the number of fragments searched by each worker occurs. In such a scenario, some workers complete early while the other workers search the remaining fragments. Also, some database fragments may take much longer to search than others because the query sequence is very similar to that fragment. Since result formatting proceeds after all workers have completed searching, an imbalance in the ratio of workers to fragments can result in execution time beyond what would be observed in the balanced case.

One potential solution to the problems of imbalance in the worker/fragment ratio and variable fragment search times would be segmenting the database into a large number of small fragments. The expectation is that a small fragment would get searched quickly. In the case of imbalance, workers that must search an additional fragment would not delay result formatting by much. In the case of highly variable fragment search times, the large number of fragments would allow mpiBLAST to balance the load among the workers, assigning additional database fragments to workers as they complete fragment searches.

A tradeoff exists when segmenting the database into many small fragments because there is significant overhead in searching extra fragments. Figure 4 shows the total execution time of mpiBLAST when searching the same database broken into a variable number of fragments. Searching a 422 fragment versus a 105 fragment database incurs an additional 140% wall clock time. The time required to format and output results increases with the number of fragments used, but is independent of the number of processors used. Figure 5 shows measurements of the result formatting and output component times for mpiBLAST when searching a database broken into a variable number of fragments.

The measurements suggest that by varying the number of database fragments, an mpiBLAST user can trade additional CPU overhead and some wall clock execution time for less variability in the execution time over different queries. Increasing the number of processors reliably shortens the execution time but may also require increasing the number of database fragments, which increases the cost of the serial result format and output component of execution time. The optimal balance between number of processors and number of fragments will depend on the priorities of the individual user.

Finally, it is important to note that in many cases fragment copy time will be negligible or non-existent because the database will have already been distributed during a previous search.

6 Future Work

There are several directions for future work on mpiBLAST's algorithms. mpiBLAST does not provide transparent fault tolerance when a node goes down. A

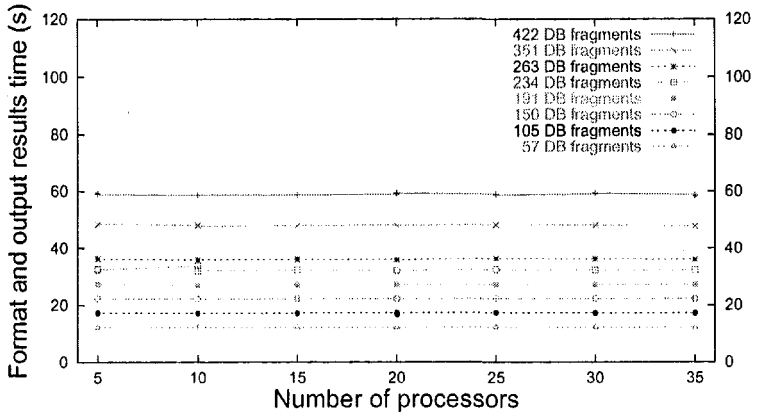

Fig. 5. The overhead of formatting and outputting results for the same mpiBLAST search increases with the number of database fragments used. The time spent formatting and outputting results is independent of the number of processors used because it is a serial component of the algorithm executed on the master node.

transparent fault tolerance mechanism could be easily integrated into the current mpiBLAST algorithm. Each node would periodically message the master that it is still alive and searching. If the master does not receive a message from a particular node before a timeout occurs, that node's work would be reassigned to another node. Fragment searching would continue as normal without the downed node.

A second potential improvement to the mpiBLAST algorithm is the integration of database updates. To implement such a scheme, each node could check a central repository of versioning information for the database fragments. If a fragment has been updated the node responsible for processing that fragment can retrieve an updated copy of the fragment. The master node would also check the database for new fragments that should be searched.

Because mpiBLAST spends the majority of its time executing NCBI Toolbox code, improvement to the Toolbox could signficantly influence performance. Our measurements indicate that there is high overhead for using additional database fragments. Further profiling to reduce the fragment overhead would allow mpiBLAST to more efficiently load-balance the search and reduce total search time.

7 Conclusion

We have described mpiBLAST, an open-source, MPI-based implementation of database segmentation for parallel BLAST searches. Database segmentation yields near linear speedup of BLAST in most cases and super-linear speedup in low memory conditions. mpiBLAST directly interfaces with the NCBI de-

velopment library to provide BLAST users with interface and output formats identical to NCBI-BLAST.

Finally, analyzing the components of mpiBLAST's running time shows that the bulk of execution time is spent performing BLAST searches. Communication consumes a relatively small portion of time. Merging and printing BLAST results also represents a relatively small amount of the total execution time. Our findings indicate that mpiBLAST scales well to at least one hundred nodes.

8 Acknowledgements

Many thanks go to Eric Weigle and Adam Englehart of the Los Alamos RADIANT group for their support and insightful comments. Wea lso thank the referees for their suggestions. Use of the Galaxy cluster in the AMS Department of SUNY Stony Brook is gratefully acknowledged. Aaron E. Darling was supported in part by NLM Training Grant 1T15LM007359-01.

References

1. S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.
2. S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI–BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.
3. R. Bjornson, A. Sherman, S. Weston, N. Willard, and J. Wing. Turboblast: A parallel implementation of blast based on the turbohub process integration architecture. In *IPDPS 2002 Workshops*, April 2002.
4. 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):745–754, April 2001.
5. N. Camp, H. Cofer, and R. Gomperts. High-throughput BLAST, September 1998.
6. R. Chen, C. Taaffe-Hedglin, N. Willard, and A. H. Sherman. Benchmark and performance analysis of TurboBLAST on IBM xSeries server cluster, 2002.
7. E. Chi, E. Shoop, J. Carlis, E. Retzel, and J. Riedl. Efficiency of shared-memory multiprocessors for a genetic sequence similarity search algorithm, 1997.
8. W. Feng, M. Warren, and E. Weigle. The bladed beowulf: A cost-effective alternative to traditional beowulfs. In *Proceedings of IEEE Cluster 2002*, 2002.
9. J. D. Glasner, G. P. III, P. Liss, A. Darling, T. Prasad, M. Rusch, A. Byrnes, M. Gilson, B. Biehl, F. R. Blattner, and N. T. Perna. ASAP, a systematic annotation package for community analysis of genomes. *Nucleic Acids Research*, 31(1):147–151, January 2003.
10. E. Glemet and J. Codani. LASSAP, a LArge Scale Sequence compArison Package. *Computer Applications In The Biosciences*, 13(2):137–143, April 1997.
11. W. Gropp, E. Lusk, and A. Skjellum. Using MPI: Portable parallel programming with the Message Passing Interface, 1999.
12. W. J. Kent. Blat – the BLAST-like alignment tool. *Genome Research*, 12:656–664, April 2002.

13. K. Pedretti, T. Casavant, R. Braun, T. Scheetz, C. Birkett, and C. Roberts. Three complementary approaches to parallelization of local BLAST service on workstation clusters. *Lecture Notes In Computer Science*, 1662:271–282, 1999.
14. A. Shpuntof and C. Hoover. Personal communication, August 2002.
15. R. K. Singh, W. D. Dettloff, V. L. Chi, D. L. Hoffman, S. G. Tell, C. T. White, S. F. Altschul, and B. W. Erickson. BioSCAN: A dynamically reconfigurable systolic array for biosequence analysis.
16. M. Warren, E. Weigle, and W. Feng. High-density computing: A 240-node beowulf in one cubic meter. In *Proceedings of SC2002*, 2002.