Massively Parallel BLAST for the Blue Gene/L

Huzefa Rangwala*‡, Eric Lantz†‡,
Roy Musselman‡, Kurt Pinnow‡, Brian Smith‡ and Brian Wallenfelt‡
*Computer Science Department,University of Minnesota-Twin Cities, MN 55455
Email:rangwala@cs.umn.edu
†Computer Sciences Department, University of Wisconsin-Madison, WI 53706
Email:lantz@cs.wisc.edu
‡ IBM Systems and Technology Group, Rochester, MN 55901
Email:mussel,kwp,smithbr,bwallen@us.ibm.com

Abstract— **The focus of this article is to explain our research involved with running a parallel implementation of the widely used BLAST algorithm on thousands of processors, available on supercomputers like the IBM Blue Gene/L. Our work involved optimally splitting up the set of queries as well as the database. We also found solutions to reduce the I/O thereby delivering a fast, high throughput BLAST. Our results show that we are capable of performing at least 2 million BLAST searches per day against a database of 2.5 million protein sequences. We also show very good performance in terms of speedup and efficiency.**

I. Introduction

There has been an ever increasing growth in the size of nucleotide and protein databases due to advances in sequencing technology. There is a high demand in the area of computational biology to extract useful information from these massive databases. This has led to the usage and development of high performance computing power to help researchers sift through the voluminous biological data. Supercomputers, clusters, and custom designed bioinformatics hardware solutions attempt to tackle the various problems stemming from analysis of biological data.

BLAST (Basic Local Alignment Search Tool) [1], [2] is a widely used bioinformatics application for rapidly querying nucleotide (DNA) and protein sequence databases. Given a query sequence, the goal is to find the most similar sequences from a large database. This has applications in the identification of functions and structures of unknown sequences or understanding the evolutionary origin of DNA or protein query sequences. Due to its high importance, BLAST also finds a place in almost all the bioinformatics solutions.

The simplest approach to find the most similar database sequences to a query is to calculate the pairwise score of the alignment between the query and every sequence in the database, using an alignment algorithm like Smith-Waterman. However, the computational cost of this method is high, and most of the alignments have very poor scores. BLAST uses some heuristic methods to reduce the running time with little sacrifice in accuracy. BLAST is available on the web or for download and the most popular implementation is from NCBI (the National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov/BLAST).

Recently, there have been several approaches to using BLAST on supercomputers and clusters. Some of these approaches [3], [4], [5] focus on the "embarrassingly parallel" solution of distributing the query set across several cluster nodes, each of which executes a serial job. Throughput is increased, but the time for a particular query to complete is unchanged. Another approach is to parition the database among cluster nodes and have each node search an assigned part of the database for the same query [6], [7], [8]. This approach of database splitting was developed in the `mpiBLAST` [6] implementation and optimized in `pioBLAST` [9]. Only one query is searched at a time, but the portion of the database each processor has to look at is reduced.

Our research focused on using a larger number of processors for running BLAST than ever before. This was done with Blue Gene/L [10], the world's fastest supercomputer[1]. Each refrigerator-sized rack of Blue Gene/L consists of 1024 nodes. Each node has two 700 MHz PowerPC 440 processors and 512 MB of memory. Compared to other large supercomputers, the Blue Gene has low power consumption per processor and a very fast interconnect network.

Our parallel BLAST solution not only focused on the database fragmentation as introduced in `mpiBLAST` but also integrated query load division. This scheme allowed us to use a large number of processors. We introuced new input and output handling schemes extending the efficient data access methods seen in the `pioBLAST` implementation.

In this report we begin Section II with a discussion of related work in terms of `mpiBLAST` and `pioBLAST`. We follow this with the discussion of our BLAST solution as well as introduced I/O techniques in Section III. Section IV shows our results on the Blue Gene/L for various datasets. We close the paper with Section V which states the conclusions and future directions for this area.

II. Previous Work

Earlier work in achieving a faster BLAST solution has led to customized BLAST hardware [11], [12], shared memory parallel BLAST and distributed memory parallel BLAST. As

Note: Huzefa Rangwala and Eric Lantz both shared equal portion of the ideas and work presented in this paper. They should both be considered as the first authors of this work.

[1] http://top500.org

discussed in [9] there have been two adopted techniques: one dealing with splitting of the database and the other dealing with the query load distribution. In this section we discuss the fairly recent work using the database fragmentation strategy.

A. mpiBLAST

`mpiBLAST` [6] was developed at Los Alamos National Laboratories. This work introduced the database fragmentation strategy in the context of BLAST. It is an open source project that uses the standard message passing protocol (MPI) [13] for its parallel BLAST implementation. It works on a wide range of clusters and supercomputers and has gained popularity amongst members of the bioinformatics community needing a high performance BLAST.

It works by initially dividing up the database into multiple fragments so that each processor has a separate smaller fragment to work on (this program is called `mpiformatdb`). The searching of a fragment is independent of any other fragment lending a very parallelizable solution.

Most of the nodes are workers: they open up a fragment file, call the NCBI serial code to conduct the search, and return the results. There is one scheduler node that reads in the query (or queries), broadcasts to the other nodes, and decides which fragments each worker should search. The last node is a writer node that receives the results from each worker, sorts them by score, and writes the output file.

`mpiBLAST` tries to substantially reduce the disk I/O read times by reducing the size of database to be read in by each cluster node and making it small enough to fit in memory. However, `mpiBLAST` suffers from non-search overheads with increasing number of processors and varying database sizes as discussed in [9].

B. pioBLAST

In a data driven application like BLAST, it is apparent that poor handling of I/O can lead to performance bottlenecks. `pioBLAST` [9] optimized `mpiBLAST` by introducing flexible database partitioning options, enabling concurrent access to shared files by use of parallel I/O, and caching of intermediate files and results.

`pioBLAST` stands for parallel I/O BLAST and uses MPI-IO [14], [15] for efficient data access. MPI-IO allows multiple processors to read or write files simultaneously. This is especially helpful in combining the results from all of the processors at the end of the run. The scheduler and writer nodes are combined into a single process called the master. The database fragments and other files are mapped into memory buffers, so they are not read more than once, even through multiple queries.

`pioBLAST` is also able to segment the database dynamically, eliminating the need to preformat the database. However, this strategy would not work well when the database size exceeds the memory on the master node. We did not test this functionality as it was efficient to have a distributed database, fragmented beforehand. This would allow us to efficiently read the entire database through multiple sources as described later in Section III.

When we ran `pioBLAST` for increasing number of fragments, we noticed that the running time no longer improved significantly. There was a limit to how many fragments the database could be broken into. This was due to the fact that as the number of fragments grew the amount of time needed to combine the results increased, resulting in a higher cost for parallelization.

III. Methods

To utilize as many nodes of the Blue Gene/L as possible, we implemented an additional parallelization of distributing the query set along with the database segmentation. We were also aware of the strengths of the Blue Gene/L, a large number of low power processors integrated with high speed interconnection networks. The work described in [16] gave a good overview of designing scalable applications for the Blue Gene/L.

We defined a *group* as a set of processors working on a fraction of the total input queries, with each processor having a fragment of the database and the group collectively having the entire database. Each group consists of a master node and a number of workers. For our tests, we use what the `pioBLAST` paper [9], calls *natural partitioning*, where the number of processors in the group is one more than the number of database fragments. Each group is assigned a subset of the overall query sequences. For a complete job, the overall number of processors used is given by the equation:

$$\#processors = (\#fragments + 1) \times \#groups \qquad (1)$$

The current implementation only supports static query set partitioning. Therefore we needed to attempt to distribute the query set so that each group would have approximately the same work load. We found that balancing the total lengths of the queries assigned to each group (as opposed to the number of queries) worked well to avoid processor idling.

Rather than having all the groups reading the database from the input source, we employed an efficient scheme where only the first group was assigned to be the reader. The processors in the reader group would be responsible for reading the needed fragment. These processors would then broadcast the fragment to corresponding processors needing the same fragment in the other groups. This scheme, which we call the *broadcast strategy*, is illustrated with Figure 1, where the red line shows the efficient data flow of this strategy. The blue line represents the alternative *all-reader strategy*, where every processor must retrieve a fragment from disk.

Instead of using a single disk as the input source we replicated our database on multiple file servers (four IBM X345 systems). The database read load was shared across the four file systems. This method also distributed the load of writing intermediate files. This technique is similar in concept to data mirroring and has been widely used before to reduce disk access bottlenecks in other applications [17].

We also employed a radical model of driving input data into the application by implementing a server agent. This agent

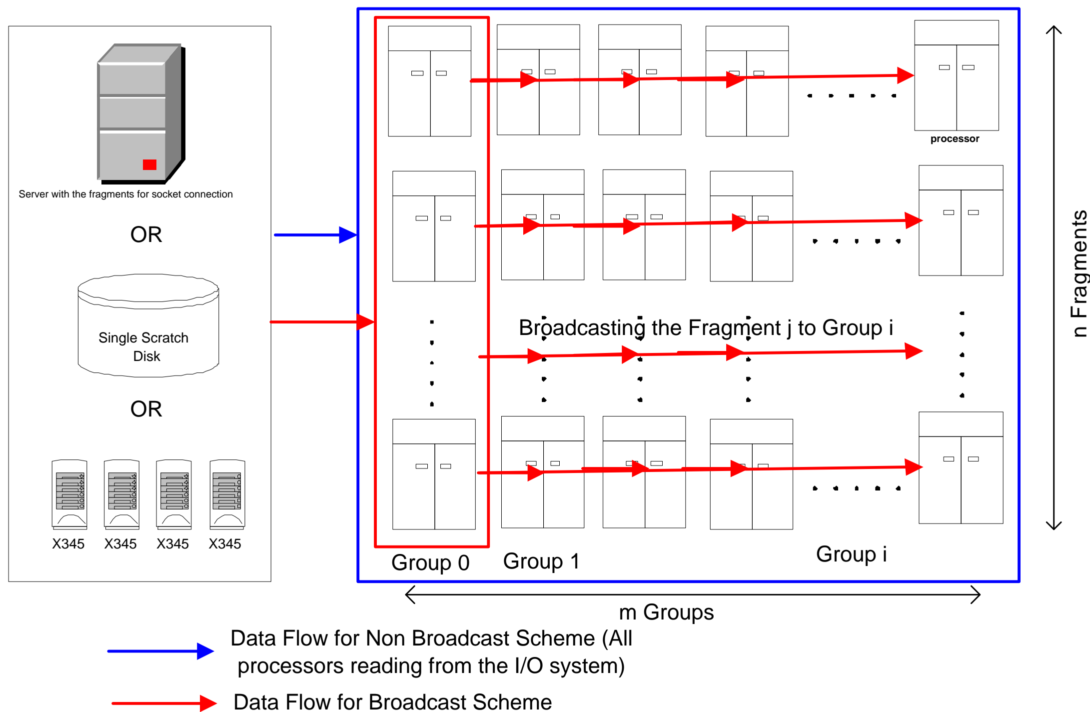

Fig. 1. Parallel Blast Model. The database is read from one of the three sources shown on the left via one of the data flow schemes.

would store the database fragments into its memory and then transfer the data to the worker nodes when requested using the TCP connection to avoid the file system overhead. Currently, our implementation used a single server and single port to handle the incoming requests for database fragments from the worker nodes. The server process would spawn child processes depending upon the fragment requested and set up a new port for the transfer. This strategy was an initial model and showed promising results.

We saw an increase in the performance when we replicated the servers such that the requests made by the worker nodes was distributed across the several servers. In the future we would like to try a dedicated server-fragment strategy where a server would be responsible for only one fragment. The worker nodes would have information regarding the server they would need to connect for getting their fragment. From our single server model results, we anticipate a substantial reduction in the input read times with the dedicated server-fragment strategy.

IV. Performance Evaluation

In this section we evaluate our parallel BLAST solution on the Blue Gene/L. We ran several experiments with the different techniques proposed in the earlier section. This helped us in understanding the optimal parameters for achieving a high throughput and highly scalable parallel BLAST implementation. Most of the tests were performed on the development Blue Gene/L system at IBM Rochester. This development system consisted of 4096 nodes, where each node consisted of two 700 Mhz PowerPC 440d processors. The system was enclosed in four racks or towers.

A. Datasets

For benchmarking purposes, we used freely available datasets and databases. There were two databases and three input files used in the data reported in this paper. One problem that we faced when benchmarking or performing comparative studies was that the database were continually updated with new sequences.

Databases:

1) *nr:* The nonredundant protein database. Contains 2.5 million protein sequences as of June 2005. 1.2 GB unformatted, 1.6 GB formatted. Source: NCBI

2) *nt:* Nucleotide database. Contains 2.5 million nucleotide sequences as of August 2005. 16 GB unformatted, 4.5 GB formatted. Source: NCBI

Query files:

1) *arabidopsis:* 1168 EST protein sequences from *Arabidopsis thalinia*. Average length: 465 bases (min 128, max 706). Source: Bioinformatics Benchmark System v. 3 from Scalable Informatics[2]

2) *arabidopsis_full:* 28014 EST protein sequences from *Arabidopsis thalinia*. Average length 419 bases (min 28,

[2]The file has "tomato" in its name, but the sequences all have arabidopsis in their descriptions

max 715). Source: Bioinformatics Benchmark System v. 3

3) *e.chrysanthemi:* 441 *Erwinia chrysanthemi* bacterial genome nucleotide sequences. Average length 658 bases (min 96, max 4367). Source: Aaron Darling (used in [6])

All of the graphs and tables reported in this paper benchmark the performance of `blastx` (querying a nucleotide sequence against a protein database), using the nucleotide *arabidopsis* query file against the protein *nr* database. The performance results remain similar when using `blastn` (querying a nucleotide sequence against a nucleotide database) or `blastp` (querying a protein sequence against a protein database). Additional results are from `blastx` query of *arabidopsis_full* versus *nr* and `blastn` query of *e.chrysanthemi* versus *nt*.

B. Results

A significant portion of the non-search fraction of the total BLAST run time is dependent on the database read time. Reading a large database can cause a huge stress on the network bandwidth as well as the I/O of any system. As mentioned in Section III we employed various strategies to reduce the database read time compared to the actual search time. We ran several experiments to compare these database reading techniques using the *arabidopsis* dataset against the *nr* database.

Figure 2 shows the database read times using 63 fragments with an increasing number of processors for various data driving strategies. Both the single disk (scratch) and the multiple X345 file systems show improvement with the broadcast strategy over the all-reader strategy. The data mirroring technique as seen with the use of four X345 systems gives the best I/O performance which finally gets reflected in the total run time. This is illustrated in Figure 3, which shows the total run times with increasing number of processors using 63 fragments. It was interesting to observe that with the broadcast strategy we were able to keep the input read times constant as we increased the number of groups and hence the number of processors.

TABLE I

Percentage improvement in the database read times using single group read and broadcast strategy. Number of fragments used = 31

	Scratch Disk			X345 Systems		
Groups	No Bcast	Bcast	Improv.	No Bcast	Bcast	Improv.
1	9.27	9.27	0.00%	8.00	8.00	0.00%
2	18.61	9.49	49.01%	5.80	8.00	-37.9%
4	21.96	9.85	55.15%	9.54	7.93	16.88%
8	49.63	11.04	77.76%	23.69	9.92	58.13%
16	115.14	11.61	89.92%	28.08	14.79	47.33%
32	220.62	15.66	92.90%	39.88	9.88	75.23%
64	311.31	26.44	92.00%	77.26	15.29	80.21%

The percentage improvement in database read times is shown in Table I. The percentage improvement of the broadcast strategy over the all-reader strategy rises considerably with an increasing number of processors. As we increase the number of processors, the all-reader strategy causes concurrent access to the input source and leads to congestion. However, in the broadcast strategy there are a constant number of processors accessing the input source irrespective of the number of groups. Hence, we see a constant read time for the same number of fragments which explains the high performance gain.

We performed a similar study on the database read times for our simple implementation of single server single port model. It was observed that the database read times were comparable to a single disk results. Preliminary testing of replicating servers, such that multiple servers would share the load as input sources showed relatively better performance compared to a single server. The next step would be to test the dedicated server-fragment strategy described earlier in section III.

The other main component of our experiments was to establish the optimal number of groups and fragments that could be run on the Blue Gene/L utilizing as many processors as possible. Figure 4 shows the total run time across different partition sizes, using increasing number of groups. Employing the broadcast strategy, along with a single input source, we

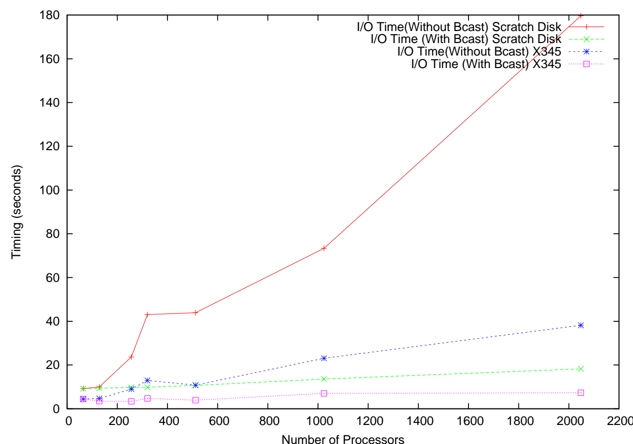

Fig. 2. Database Read Time Performance for x345 vs Scratch Disk. The database was split into 63 fragments.

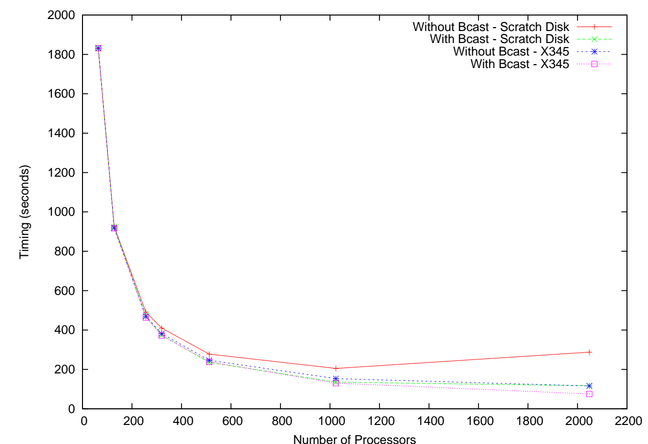

Fig. 3. Blast Run Time Performance for x345 vs Scratch Disk. The database was split into 63 fragments.

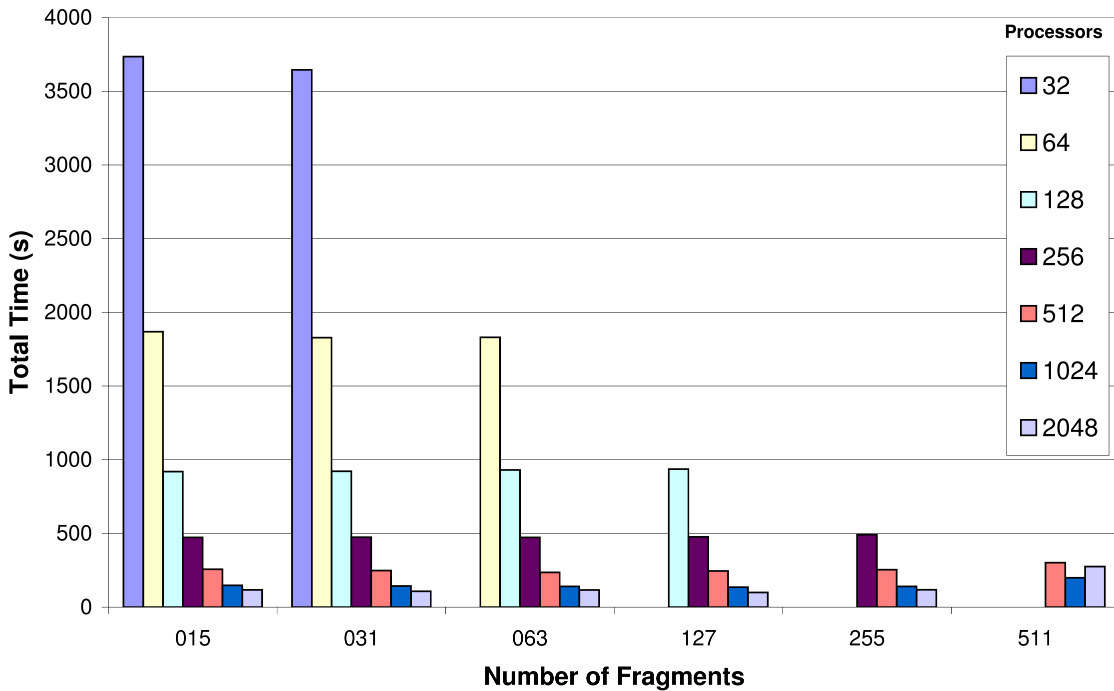

Fig. 4. Total runtimes for using different partition sizes with varying number of processors on a single scratch disk. For example, following equation 1, the tallest bars represent 2 groups of 15 fragments and 1 group of 31 fragments, respectively.

see that there is almost perfect scalability as we increase the number of groups for almost all fragment sizes. It is also interesting to note that the run time remains the same for same number of processors with different fragment size.

However, for 511 fragments increasing the number of processors leads to an increase in the run times. Looking at the timing profiles, we realized this was due to the large number of intermediate files and results that needed to be recombined. The size of the fragment to be searched per processor becomes very small, making the results combination step greater than the actual search time.

Figure 5 shows the speedup performance achieved by our scheme with increasing processors across different fragment sizes. Unlike `mpiBLAST` results [6], we assume our serial run times to be void of any paging overhead and hence compute it with reference to a parallel job using 16 processors on the Blue Gene/L.

For the same *arabidopsis* dataset, the running time on 16 processors was over 7000 seconds, which reduced almost linearly to only 73 seconds (our best run time for the benchmark) using a full rack (1024 nodes/2048 processors, 32 groups of 64 processors). A larger test with 28,014 EST sequences (*arabidopsis_full*) finished in under 20 minutes, equating to an amazing rate of 2 million sequences per day. These runs were made using the four X345 systems and the broadcast strategy.

We also compared our results against the *e.chrysanthemi* benchmark used by the `mpiBLAST` group[3] on the Green Destiny cluster [18]. We were able to complete the task in 71 seconds using 512 nodes of Blue Gene. The reported run time for `mpiBLAST` on the Green Destiny cluster was 10 minutes using 128 processors.

[3]http://mpiblast.lanl.gov/About.Performance.html

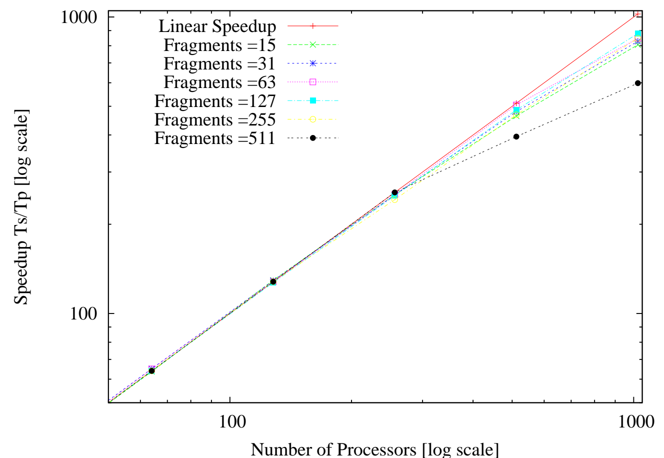

Fig. 5. Speedup curve on scratch disk with varying fragment sizes

V. CONCLUSIONS

Our work has helped achieve a massively parallel BLAST search engine, delivering high throughput while remaining efficient with the usage of a large number of processors. The result of this work is what we believe to be the fastest BLAST machine yet developed. Our tests show it is able to handle many times more queries per day than NCBI[4], though we were have not yet tested as diverse a set of queries and databases as available on NCBI's servers.

Using thousands of processors raises technical challenges not conceivable on the level of a few dozen processors,

[4]As of April 2005, NCBI handles 400,000 queries per day according to ACM Queue vol. 3, no. 3, April 2005

and issues such as I/O and problem decomposition must be resolved in even more detail. We have introduced several practical ways of reducing the disk access times in data driven applications like BLAST.

Currently, our query partitioning technique is static and depends on the number of characters to be processed by each group. Dynamic load balancing and partitioning of the query set would be a possible future study. It would also be interesting to test different algorithms for distribution of the query set as well as database fragmentation.

We also discussed the possible implementation of a dedicated-fragment server model for better handling of the requests made by the several worker nodes. Future opportunities for research include adapting this structure for non-heuristic homology searches using the Smith-Waterman algorithm. Another widely used version of BLAST called PSI-BLAST iteratively refines the searches. Parallelizing this implementation would involve challenges of trying to predict the sequences that would be used for the next round of search. The ideas in this paper could also be extended to several other data driven applications on massively parallel environments.

ACKNOWLEDGMENTS

The work was supported by the IBM Systems Technology Group at Rochester, Minnesota. Huzefa Rangwala and Eric Lantz were working as interns and would like to express deep gratitude to all the members involved with the Blue Gene/L's software and development team. Special thanks to our manager Patrick Keane, technical leaders, Jose Moreira, Jeff Parker, Tim Mullins, Carlos Sosa, Charles Archer and Joseph Ratterman (all involved with the Blue Gene/L Systems Team) for their encouragement and technical help in this effort.

We would like to express our gratitude to Heshan Lin of NCSU for helping us with the `pioBLAST` code.

IBM, PowerPC, and Blue Gene are registered trademarks of IBM Coorporation.

REFERENCES

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–4010, 1990.

[2] S. F. Altschul, L. T. Madden, A. A. Schffer, 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 Research*, vol. 25, no. 17, pp. 3389–402, 1997.

[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*, no. 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*, 1998.

[5] E. Chi, E. Shoop, J. Carlis, E. Retzel, and J. Riedl, "Efficiency of shared-memory multiprocessors for a genetic sequence similarity search algorithm," *Technical Report, University of Minnesota, CS department*, vol. TR97-05, 1997.

[6] A. E. Darling, L. Carey, and W. Feng, "The design, implementation, and evaluation of mpiblast," *In proceedings of CluterWorld and Expo*, 2003.

[7] R. Bjornson, A. Sherman, S. Weston, N. Willard, and J. Wing, "Turboblast(r): A parallel implementation of blast built on turbohub," *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.

[8] D. Mathog, "Parallel blast on split databases," *Bioinformatics*, vol. 19, no. 14, 2003.

[9] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova, "Efficient data access for parallel blast," *International Parallel and Distributed Processing Symposium*, 2005.

[10] "Blue gene," *IBM Journal of Research and Development*, vol. 49, no. 2–3, 2005.

[11] R. Luthy and C. Hoover, "Hardware and software systems for accelerating common bioinformatics sequence analysis algorithms," *Biosilico*, vol. 2, no. 1, 2004.

[12] K. Muriki, K. Underwood, and R. Sass, "Rc-blast: Towards an open source hardware implementation," *Proceedings of 4th IEEE International Workshop on High Performance Computational Biology*, 2005.

[13] "Mpi: Message-passing interface standard," *Message Passing Interface Forum*, 1995.

[14] J. May, "Parallel i/o for high performance computing," *Morgan Kaufmann Publishers*, 2001.

[15] R. Thakur, W. Gropp, and E. Lusk, "On implementing mpi-io portably and with high performance," *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, 1999.

[16] G. Almasi and et. al, "Early experience with scientific applications on the blue gene/l supercomputer," *Proceedings of the 11th International Euro-Par Conference, Euro-Par 2005*, vol. 3648, 2005.

[17] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet, "Alphasort: A cache-sensitive parallel external sort," *VLDB*, vol. 4, pp. 603–627, 1995.

[18] W. Feng, "Greendestiny + mpiblast = bioinfomagic," *Proceedings of the 10th International Conference on Parallel Computing*, 2003.