Efficient Parallel Execution of Sequence Similarity Analysis Via Dynamic Load Balancing

James D. Jackson
Philip J. Hatcher
Department of Computer Science
Kingsbury Hall
University of New Hampshire
Durham, NH 03824 USA

Abstract

We present a parallel approach to analyzing sequence similarity in a set of genomes that employs dynamic load balancing to address the variation in execution time for genes of different lengths and complexity, the variation in processor power in different nodes of the computer cluster, and the variation in other load on the nodes. Our approach executes using MPI on a cluster of computers. We provide experimental results to demonstrate the effectiveness of using our approach in conjunction with NCBI BLAST.

1 Introduction

The identification of putative orthologous genes is critical for comparative and functional genomics. The first step in analyzing orthology in a set of genomes is to identify homologs. This is typically done by computing pairwise alignments between all genes in the set of genomes. Even using a heuristic-based approach such as BLAST [1], this is a computationally intensive task, since there are $O(n^2)$ pairs to consider in a set of n genes.

At the University of New Hampshire we have built a system for analyzing genomic data that allows genomes to be added incrementally. Homologs are computed and stored for all pairs of genomes. When a new genome is added, homologs are computed individually for it paired with every other genome already in the system. Once the homologs are computed, a variety of approaches can be executed to analyze orthology for any subset of the genomes in the system.

Since the homology analysis is the most computationally intensive component of the system, we have provided a parallel implementation for it. This parallel implementation, which utilizes MPI [6], is designed to run on either large-scale clusters of identical processors, such as the IBM Cell cluster available on our campus, or on smaller-scale clusters of heterogeneous Intel servers found in individual departments.

Targeting clusters of heterogeneous machines meant that we needed to include dynamic load balancing in our approach in order to spread the work across processors with varying clock speeds. In addition, of course, the homology analysis itself presents a varying workload, since the time to analyze a gene against a set of genes varies upon both the gene length and the complexity of the gene. For example, Figure 1 shows the variation in running time when using BLAST to analyze a set of Human[1] proteins against itself. Generally, running time correlates with protein length but there is considerable variation.

We also wanted our approach to work with a variety of tools for doing the homology analysis. For instance, while most of our users are primarily focused on using BLAST, there are efficient implementations of the Smith-Waterman alignment algorithm for the IBM Cell [5] [7] that we wanted to be able to use on that cluster. We also did not want to require that the source code for the tool be available. Therefore, our implementation allows the homology analysis tool to be invoked as an executable, connecting to its inputs and outputs through the file system.

There has been considerable work on parallelizing BLAST. One of two basic strategies is followed: query segmentation or database segmentation.

In query segmentation (e.g. [2]), the sequence database is replicated on all processing nodes, the query sequences are distributed among the nodes, and different queries are worked on at the same time. This allows BLAST to be treated as a black box and works well when the sequence database fits in the memory of the processing nodes.

If the sequence database does not fit in memory,

[1] *Homo sapiens* data (67033 proteins; 32M amino acids) downloaded from the European Bioinformatics Institute in November 2010.

then database segmentation (e.g. [4]) can be employed. The sequence database is distributed among the processing nodes, the queries are replicated on all processing nodes, and queries are processed against different segments of the database at the same time. However, all E-values must be corrected before they are reported, since the E-value is dependent on the overall size of the database. This requires that the BLAST box be opened up in order to implement the necessary correction.

In our environment the sequence database is relatively small, containing the sequences from a single genome, and therefore it will fit in memory. In addition, we want to treat the homology analysis tool as a black box. Therefore, our implementation utilizes query segmentation.

Nearly all implementations of query segmentation use simple static load balancing where queries are evenly distributed in advance among the processing nodes, either based upon query count or query length. Wang and Lefkowitz [8] implemented a hybrid approach in which 90% of the workload is allocated statically and the remaining 10% is held in reserve to allocate dynamically on demand. Because of our interest in heterogeneous clusters, and clusters with dynamically varying work loads, our implementation is apparently unique in that it fully utilizes dynamic load-balancing. We demonstrate that this use of dynamic load-balancing can be performed with very low overhead.

2 Implementation

Our implementation currently utilizes NCBI BLAST [3], targets a Linux environment, and is depicted visually in Figure 2. We use a master/worker model to dynamically distribute the load. There is one master MPI process that reads the query file. The master reads a block of queries from the file, and then sends them to a worker MPI process that has requested more work. We allocate one worker process for each processor in the cluster we are utilizing. The master and the worker communicate using MPI messages.

A worker runs a BLAST search on the queries it has been sent. The search is performed against the sequence database, which is replicated within each worker process. The worker uses fork/exec to initiate the BLAST executable and utilizes pipes to send queries to and receive results from the BLAST process.

Workers send BLAST results to a single writer MPI process. The writer process writes the results into a file. After the worker sends the results to the writer, it then requests more work from the master process.

A key issue is choosing the block size for the blocks of queries to be sent from the master to the workers. In general a small block size is desirable to get the most benefit from the dynamic load balancing, otherwise one worker process might end up processing the last block long after all other workers have finished. On the other hand, too small a block size could result in too much overhead in MPI message passing and the repeated invocations of BLAST.

In addition, what units should we use to measure block size? Because of the variability in the time required for different queries, we decided to use total sequence length as the metric for block size, rather than number of queries. We pack queries into a block until the block size is reached. Note that the last query in the block will likely cause the actual block size to exceed the target block size, as we do not split queries across two blocks.

Our system uses a block size of 20K amino acids (or nucleotides). In the next section, which provides an experimental evaluation of our approach, we will include a discussion of the experiments that informed our decision on block size.

3 Experimental Evaluation

We have evaluated our approach on a 36-processor Linux cluster consisting of the following four nodes:

- c0: 12 Intel Xeon 2.66 GHz processors, 3 GB main memory;
- c1: 8 Intel Xeon 1.86 GHz processors, 1.5 GB main memory;
- c2: 8 Intel Xeon 1.86 GHz processors, 1.5 GB main memory;
- c3: 8 Intel Xeon 3.00 GHz processors, 1.5 GB main memory.

This cluster is representative of the clusters we are targeting: ones with nodes that have a large amount of memory and that have a varying number of processors with varying clock speeds.

The evaluation used the following protein datasets:

- *Homo sapiens* (aka Human): 67033 proteins; 32M amino acids; downloaded from the European Bioinformatics Institute in November 2010.
- *Mus musculus* (aka Mouse): 50033 proteins; 24M amino acids; downloaded from the European Bioinformatics Institute in November 2010.

- *Caenorhabditis brenneri* (aka Brenneri): 43238 proteins; 16M amino acids; Augustus gene predictions downloaded from Wormbase in July 2008.
- *Burkholderia cenocepacia AU 1054* (aka AU1054): 6531 proteins; 2.6M amino acids; downloaded from the Integrated Microbial Genome database in September 2006.

These genomes were chosen to reflect the variety of genomes that we expected to handle in our system: from mammals to bacteria. Note that all these datasets will fit comfortably in the memory of the nodes of our cluster.

Each execution time reported in this section is the minimum of at least three runs and is reported to the nearest second. Execution times were obtained while running on a dedicated cluster.

We first ran a series of experiments to study block size, utilizing the Human, Brenneri and AU1054 genomes. Since our system is designed to BLAST one genome against another, we studied the nine combinations of those genomes BLAST-ed against each other as query and database, while varying the block size over the values 1K, 5K, 10K, 20K, 40K and 80K amino acids. Figure 3 graphs the block size against the run time for these nine BLAST runs using our system on our 36-processor cluster. Note that there is a trough in the graphs from block size 10K up to 40K, which is why we settled on a block size of 20K for our system.

We focused on two additional runs, Human versus Mouse and Brenneri versus Brenneri, to judge the overall effectiveness of our approach. These runs were performed using a block size of 20K and all 36 processors in our cluster.

The execution time for Human versus Mouse using dynamic load balancing was 1654 seconds. We compared this to a static load balancing approach, where the total number of amino acids in the query sequences were divided as evenly as possible, while respecting query boundaries, into 36 chunks to match the 36 processors in our cluster. The execution time using this static approach was 2253 seconds. In this case, therefore, dynamic load balancing provided a 1.4 speedup over static load balancing. This indicates that the dynamic approach could automatically adjust for the varying processor clock rates in our cluster. Of course, doing experiments on a dedicated cluster with processors of varying clock rates is an effective simulation for a cluster of processors with identical (or varying) clock rates, where the load on the processors is dynamically changing due to other programs executing on the cluster.

To try to judge how well we were utilizing the cluster we also compared out dynamic load balancing execution time to a serial execution. We ran Human vs. Mouse serially on c3, the node with the fastest processors in the cluster. This serial run had an execution time of 42,756 seconds, meaning that the dynamic load balancing approach provided a speedup of 25.9 over serial execution. We need to adjust this speedup figure to account for the varying clock rates in the cluster, because we do not have 36 3.0 GHz processors. Instead we have 12 2.66 GHz processors on c0, the equivalent of 10.6 3.0 GHz processors. And we have 16 1.86 GHz processors on c1 and c2, the equivalent of 9.9 3.0 GHz processors. And, of course, we have 8 3.0 GHz processors on c3. This is a total of the equivalent of 28.5 3.0 GHz processors. Our actual speedup of 25.9 therefore provides an efficiency of 25.9/28.5, 91%.

This high efficiency indicates that the dynamic load balancing has low overhead. First, the additional MPI message-passing time required by the dynamic load balancing (as compared to static load balancing) is negligible compared to the time spent in BLAST itself. Second, the time spent repeatedly initiating BLAST on a processor for each chunk of 20K amino acids is small. Repeated reading of the database sequences is very fast due to the large amount of disk caching that Linux systems perform. Note this is true both for an individual processor and for a group of processors on one node of the cluster, which will all share the disk cache on that node. However, the initial read of the database must be performed over the network since we use NFS to mount a common filesystem onto all nodes of the cluster. One portion of the overhead is due to the time to read the database across the network into the nodes of the cluster. Another portion of the overhead is due to the presence of a master process and a writer process, in addition to the 36 worker processes. These two processes must be executed on processors that are also executing worker processes.

We repeated this analysis for Brenneri versus Brenneri, obtaining similar results. Dynamic load balancing provided a 1.3 speedup over static load balancing in this case, and a 25.0 speedup, i.e. 88% efficiency, over serial execution on c3. The lower efficiency for this smaller problem indicates that the overheads that are present are relatively constant, and are reduced when operating on larger genomes. This implies that the overhead is largely due to the time to startup the set of worker processes and load the sequence database across the network to each node of the cluster.

To confirm that the load is distributed appropriately among the various nodes of the cluster, we instrumented our code to count the number of blocks that were processed on each node for the Brenneri versus Brenneri run using dynamic load balancing. These numbers are presented in Table 1. Column 4 shows the ratios of the number of blocks processed per processor for each node of the cluster to the number of blocks

processed per processor for c3, the fastest node. Column 5 indicates the ratio of each node's clock rate to the clock rate of c3. Note that the number of blocks processed on c2 matches exactly what would be predicted by the clock rates for c2 and c3, but that the numbers of blocks processed on c0 and c1 lag a bit from what would be predicted. This is because the master process is on c0 and the writer process is on c1, causing those nodes to underperform a bit relative to c2 and c3. Taking the impact of the master process and the writer process into account, it is clear that blocks are being distributed to match the clock rates of the processors on the different nodes.

Finally, we ran our dynamic load balancing for Brenneri versus Brenneri using 8 worker processes on just c3. This completed in 1813 seconds. We compared this to running NCBI blastall using 8 threads on c3. This completed in 2475 seconds, meaning that our scheme that replicates the database for each worker process provides a 1.4 speedup over blastall using one thread for each processor, even allowing for our approach running two extra processes, the master and the writer.

4 Conclusions

We have shown that performing sequence similarity analysis in parallel using dynamic load balancing can effectively distribute the load given a cluster with processors with varying clock rates. This implies that this approach will be equally effective in the context of fluctuating loads on the processors due to other jobs running on the cluster. The overhead of the dynamic load balancing seems to be low enough that there would be no reason not to use the approach even in a situation where the cluster is dedicated and contains homogeneous processors.

Acknowledgements

We thank W. Kelley Thomas, director of the Hubbard Center for Genome Studies at the University of New Hampshire, for motivating and assisting this work. Support for James Jackson was provided by NIH COBRE ARRA supplement 3P20RR018787-07S1.

References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, Oct. 1990.

[2] 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.

[3] C. Camacho, G. Coulouris, V. Avagyan, N. Ma, J. Papadopoulos, K. Bealer, and T. Madden. Blast+: architecture and applications. *BMC Bioinformatics*, 10(1):421, 2009.

[4] A. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiBLAST. In *Proceedings of the ClusterWorld Conference and Expo*, 2003.

[5] M. Farrar. Optimizing Smith-Waterman for the Cell broadband engine. `http://farrar.michael.googlepages.com/SW-CellBE.pdf`.

[6] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2.* High Performance Computing Center Stuttgart, 2009.

[7] A. Szalkowski, C. Ledergerber, P. Krhenbhl, and C. Dessimoz. SWPS3—fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. *BMC Research Notes*, 1, 2008.

[8] C. Wang and E. Lefkowitz. SS-Wrapper: a package of wrapper applications for similarity searches on Linux clusters. *BMC Bioinformatics*, 5, 2004.

node node	blocks per node (b/n)	blocks per processor (b/p)	b/p relative to c3	GHz relative to c3 GHz
c0	267	22.25	.76	.89
c1	128	16	.55	.62
c2	144	18	.62	.62
c3	234	29.25	1.0	1.0

Table 1: Blocks processed per node and per processor for Brenneri versus Brenneri.

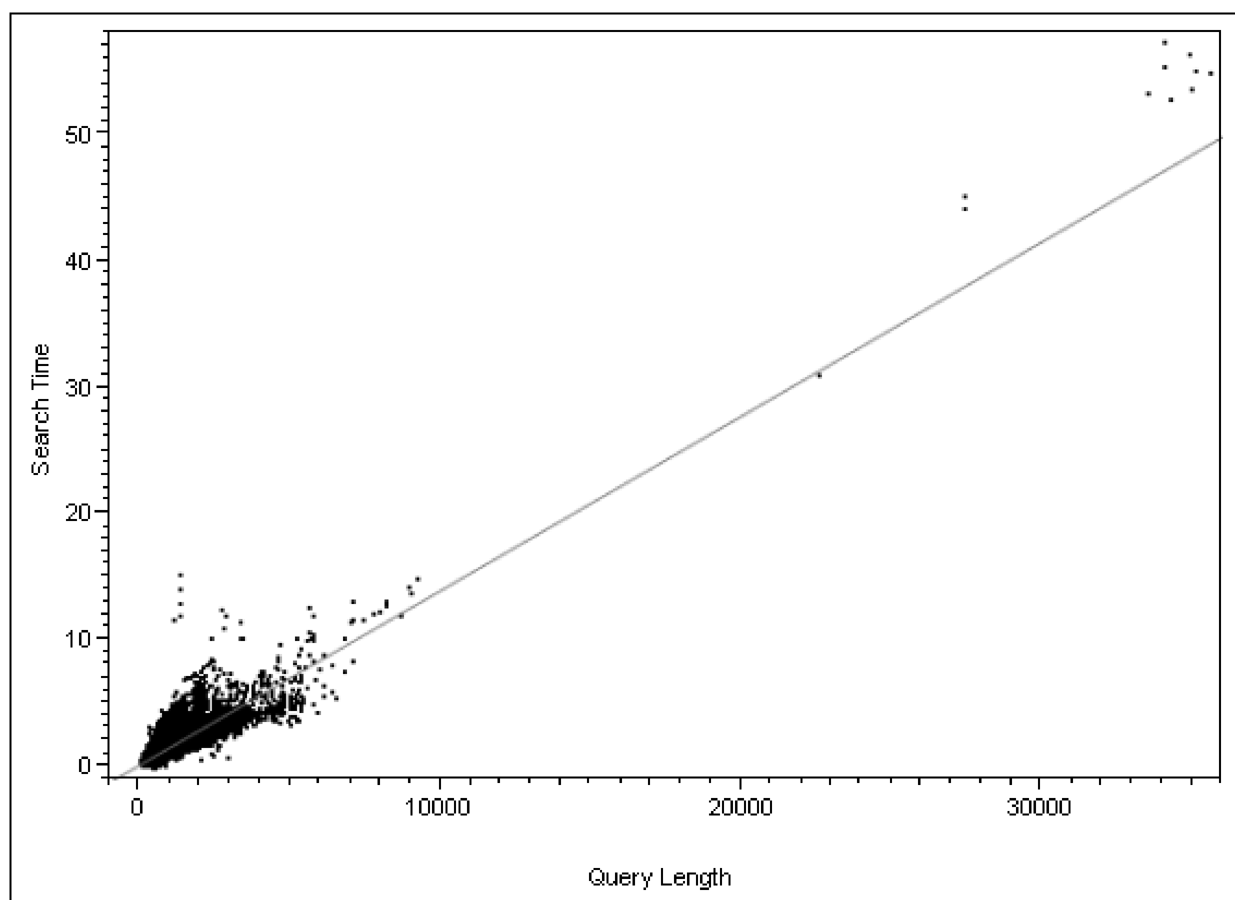

Figure 1: Time to compute BLAST alignments for Human query proteins of varying lengths against all Human proteins.

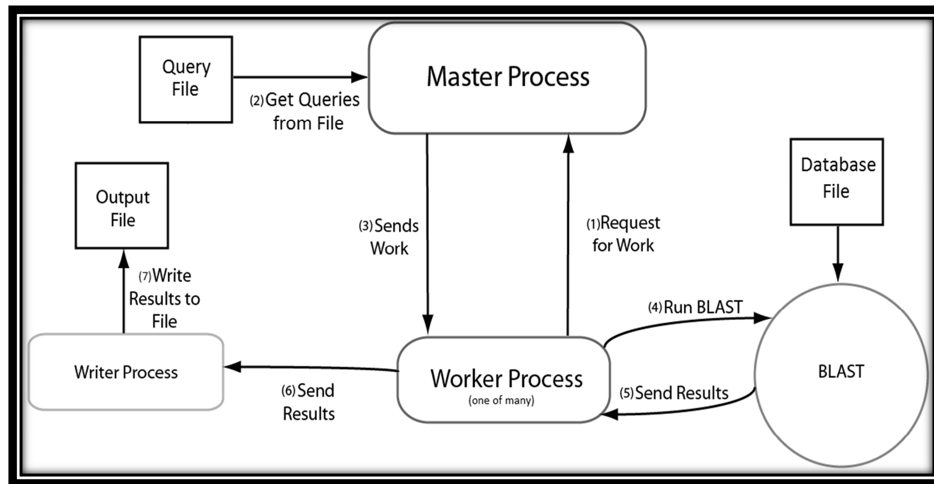

Figure 2: The processes and their interactions in our implementation.

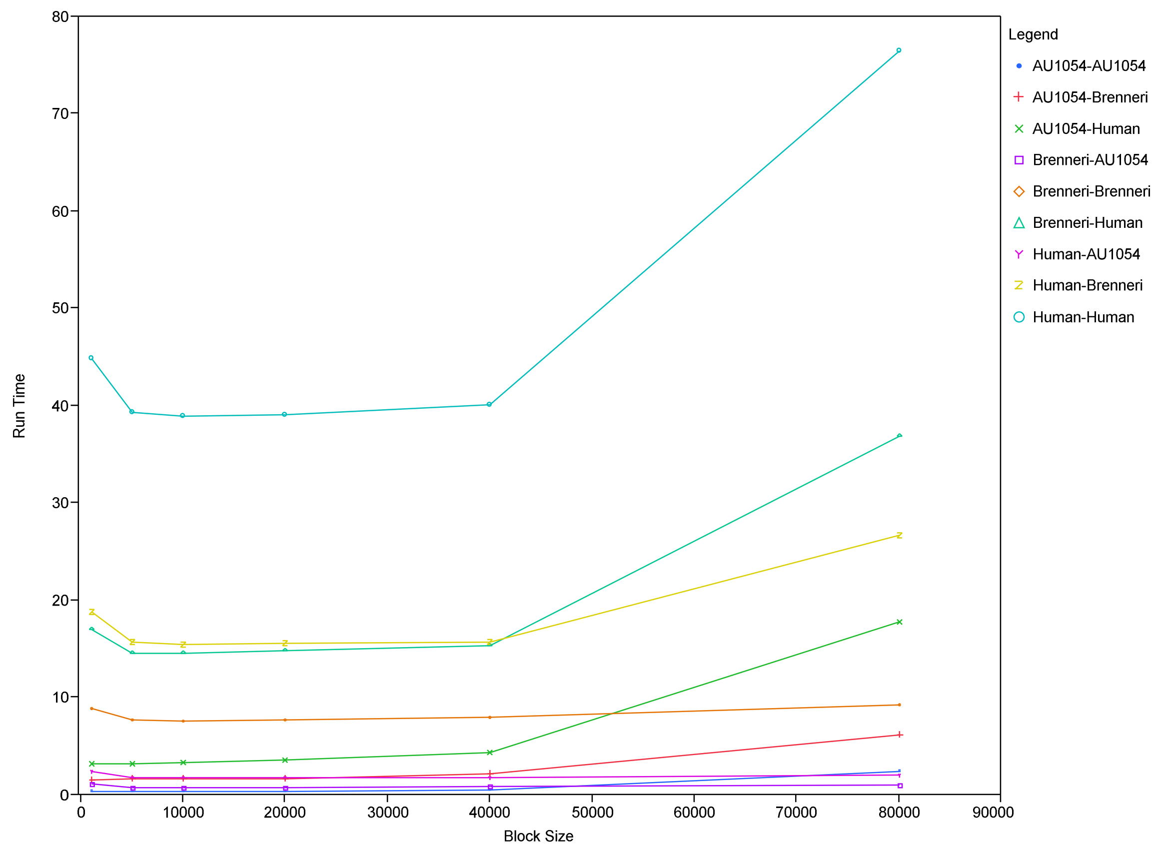

Figure 3: Run time (minutes) versus block size (total sequence length) for nine parallel BLAST runs using various sized genomes.