An Efficient Parallel Implementation of the Hidden Markov Methods for Genomic Sequence Search on a Massively Parallel System

Karl Jiang, Oystein Thorsen, Amanda Peters,
Brian Smith, *Member*, *IEEE*, and Carlos P. Sosa, *Member*, *IEEE*

Abstract—Bioinformatics databases used for sequence comparison and sequence alignment are growing exponentially. This has popularized programs that carry out database searches. Current implementations of sequence alignment methods based on hidden Markov models (HMM) have proven to be computationally intensive and, hence, amenable to architectures with multiple processors. In this paper, we describe a modified version of the original parallel implementation of HMMs on a massively parallel system. This is part of the HMMER bioinformatics code. HMMER 2.3.2 uses profile HMMs for sensitive database searching based on statistical descriptions of a sequence family's consensus [1]. Two of the nine programs were further parallelized to take advantage of the large number of processors, namely, *hmmsearch* and *hmmpfam*. For our study, we start by porting the parallel virtual machine (PVM) versions of these two programs currently available as part of the HMMER suite of programs. We report the performance of these nonoptimized versions as baselines. Our work also includes the introduction of an alternate sequence file indexing, multiple-master configuration, dynamic data collection and, finally, load balancing via the indexed sequence files. This set of optimizations constitutes our modified version for massively parallel systems. Our results show parallel performance improvements of more than one order of magnitude (16 times) for *hmmsearch* and *hmmpfam*.

Index Terms—Hidden Markov models, HMMER, massively parallel systems, multiple-master parallelization, parallel implementation, genomic sequence search, bioinformatics.

1 Introduction

1.1 Motivation

The objective of sequence alignment is to be able to select two sequences (a collection of characters representing amino acids or nucleotides) and compare them to determine the degree of similarity. The degree of similarity (based on some threshold) provides researchers with a measurement they can use to reach conclusions about the homology between the two sequences. Schuler's [2] definition of similarity and homology is "Similarity is an observable quantity that might be expressed as, say, percent identity or some other suitable measure. Homology, on the other hand, refers to a conclusion drawn from the data two genes share a common evolutionary history."

In recent years, the number of sequences and, therefore, the size of databases available for comparison have grown exponentially. This growth has prompted scientists to develop faster and more sophisticated algorithms to keep pace with the increasing size of the databases [3]. Software improvements combined with state-of-the-art hardware have allowed computational biologists to enter a new era of comparative genomics [4], [5], [6].

- *K. Jiang, O. Thorsen, A. Peters, and B. Smith are with IBM, Rochester, 3605 Highway 52 North, Rochester MN 55901. E-mail: {kjiang, othorse, apeters, smithbr}@us.ibm.com.*
- *C.P. Sosa is with the University of Minnesota Supercomputing Institute, 509 Walter Library, 117 Pleasent St SE, Minneapolis, MN 55455. E-mail: cpsosa@us.ibm.com.*

Manuscript received 2 Dec. 2006; revised 12 Apr. 2007; accepted 23 May 2007; published online 12 June 2007.
Recommended for acceptance by T. Davis.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0389-1206.
Digital Object Identifier no. 10.1109/TPDS.2007.70712.

One example of this new growth may be seen in the use of probabilistic-based methods in bioinformatics, in particular, in the database searches. Although hidden Markov models (HMMs) were initially introduced for pattern recognition in digitized acoustics of the human voice [7], they have become popular in bioinformatics. Current efforts in this area (including software) have been reviewed by Eddy [8], [9], [10]. HMMs in bioinformatics have been used in multiple-sequence alignments [8], [9], [10], [11], [12]. They are also used in sequence analysis to produce an HMM that represents a sequence profile to study sequence composition and patterns [13], to locate genes, and to predict protein structures.

The list of molecular biology databases is constantly increasing, and more scientists rely on this information [14]. The Nucleic Acids Research (NAR) Molecular Biology Database collection expanded by 139 more databases in 2006. This is just for the databases published in *Nucleic Acids Research* [14]. The demand for programs that can help analyze all this data will continue to increase. Carrying out, for example, database searches and sequence alignment tends to be not only computationally intensive but also highly I/O demanding. Recent attempts to complete a search of the NT database with the same NT database as a query have illustrated the magnitude of the computational resources required for a task of this magnitude [15]. As the databases continue to grow, so does the need for software optimized for highly parallel systems.

1.2 Other Related Work

The parallelization of three of the HMMER programs (*hmmcalibrate*, *hmmsearch*, and *hmmpfam*) has been implemented using Portable Operating System Interface for Unix (POSIX) multithreading on shared-memory machines. On distributed-memory machines, a message passing library

such as a parallel virtual machine (PVM) or message passing interface (MPI) is required [13]. HMMER uses PVM [16]. This original implementation is based on a master/worker scheme designed for a cluster of workstations. The algorithm assigns one of the nodes in the cluster as the master, and the master spawns jobs to the other nodes (workers) [13]. This approach was implemented in the three HMMER programs mentioned above [13].

In the original implementation, the program *hmmpfam* requires HMM databases on the master node, as well as on each of the worker nodes. Hence, a local file system on the master and workers is desirable for faster I/O performance. Two steps are necessary: indexing of the database and predistributing the database to each node in the cluster [13].

Database fragmentation (or predistribution) is an important consideration when parallelizing programs that carry out similarity sequence searches. Searching a database against a given query is a fundamental process in bioinformatics. The idea of query and database fragmentation for parallelizing database search algorithms has been documented in the literature [17], [18], [19], [20], [21], [22], [23], [24], [25], [26]. As we mentioned in the previous paragraph, this requires distribution of the database or fragments to all the nodes. Depending on the parallelization scheme, different fragments of the query will search the entire database on different nodes, or the entire query will search fragments of the database on different nodes [27]. This point is particularly important on distributed-memory machines where accessing the file system may be time consuming.

Similar to the implementation found in *hmmcalibrate*, *hmmsearch*, and *hmmpfam*, other authors have utilized the master/worker scheme as described in [27]. This approach has been divided into a "master-writing" implementation where the master is the focal point and coordinates not only all the information received from the workers but I/O as well [27]. Although this approach has certain advantages, it is not well suited for massively parallel machines since the master becomes the bottleneck as more workers are added to finish the task sooner [27].

The second approach is to let the workers carry out their own I/O. There are two ways to perform this task. The "worker-writing" approach is implemented via either collective or individual mechanisms [27]. The latter case has been shown to be more efficient [27]. A recent study makes use of the Global Arrays toolkit [28] to distribute the database to remote nodes on the system, combined with a "worker-writing" scheme [22].

The importance of efficient parallel implementations of the *hmmpfam* module has been documented in the literature. Compiler and runtime optimizations have been shown to improve performance on shared-memory systems [29].

1.3 Our Contributions

In this study, we attempt to improve the scalability of the three programs that are part of the HMMER 2.3.2 package. The initial port from PVM to MPI [30] without any optimization is not encouraging. The scalability observed for this version is not higher than 64 nodes. This type of performance is not satisfactory on a massively parallel machine.

Recently, Gardner et al. [15] reported their experiences in implementing mpiBLAST-PIO on GreenGene. This system is an ad hoc computing grid composed of several heterogeneous clusters [15]. Since mpiBLAST-PIO is also a database search program, the implementation on a system of this nature faces many of the challenges that we encountered optimizing *hmmsearch* and *hmmpfam*. Particularly relevant to our study is their software architecture. Since all of the nodes on GreenGene have multiple processors, they define them as *virtual nodes*. Each processor corresponds to a *group*, and groups were from 8 to 16 physical nodes. Each group was defined as an independent worker. Their hierarchical approach involves a *GroupMaster* and a *SuperMaster*. In this hierarchical scheme, the role of the SuperMaster is to assign queries to the GroupMasters. The GroupMaster runs the queries on each of the virtual nodes.

Our optimized versions of *hmmsearch* and *hmmpfam* were designed to take advantage of a massively parallel architecture. The system utilized in this study is the Blue Gene/L (BG/L) massively parallel supercomputer from IBM [31]. In contrast to clusters of workstations, the BG/L architecture consists of thousands of processors interconnected by a customized network for communication [31]. I/O is carried out via the I/O cards rather than the local file system [31].

We show how a similar scheme with additional improvements can work very well on a massively parallel machine. We also show that a simple master/worker type of approach is not sufficient to handle excessive communication between the workers and the master, in particular, since the master has to serialize all the requests coming from the workers. We show that by using our hierarchical approach, scalability is improved by more than one order of magnitude (16 times).

The rest of the paper is organized as follows: Section 2 provides an overview of the software and hardware utilized in this study. This section identifies some of the challenges faced with porting the software to the BG/L hardware as well, since the initial parallelization of these three programs was intended for clusters of workstations. Section 3 proposes the use of the *SuperMaster* approach and additional optimizations to improve scalability. Section 4 summarizes the performance obtained as a result of our improvements. Section 5 analyzes the results from Section 4 and, finally, Section 6 shows the conclusions reached from this study.

2 Software and Hardware Overview

2.1 Software

A complete discussion of hidden Markov models may be found in the work by Krogh et al.[12]. HMMER 2.3.2 consists of nine different programs: *hmmalign*, *hmmbuild*, *hmmcalibrate*, *hmmconvert*, *hmmemit*, *hmmfetch*, *hmmindex*, *hmmpfam*, and *hmmsearch* [13]. Out of these nine programs, three have been parallelized: *hmmcalibrate*, *hmmpfam*, and *hmmsearch*. *hmmcalibrate* is used to identify statistical significance parameters for a profile HMM. *hmmpfam* is used to search a profile HMM database. *hmmsearch* is used to carry out sequence database searches [13].

2.2 Hardware Architecture

The architecture of BG/L has been described elsewhere [31], but it is important to provide a brief overview of the components of BG/L to understand the results presented here. The components of BG/L can be categorized as building blocks, and each block contributes to the overall architecture. In this building block scheme, the smallest component is the *chip*. The BG/L basic block is a PowerPC 440 dual-core processor (one node). Each processor core runs at a frequency of 700 MHz, and each processor core can perform four floating-point operations per cycle, giving a theoretical peak performance of 5.6 Gflops/chip. The chip

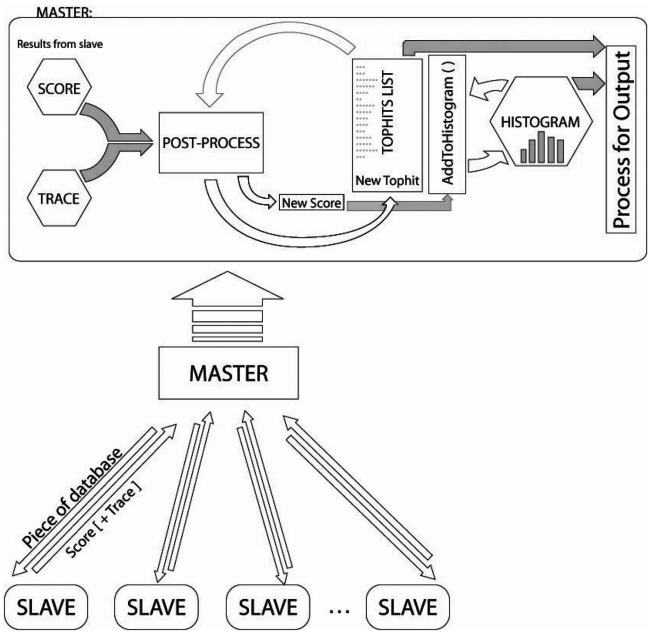


Fig. 1. The original parallel scheme for *hmmsearch* (the one for *hmmpfam* is similar) based on the PVM model. Two postprocessing steps, one processing the score and trace and the other building a histogram from the scores.

constitutes the computation node [31]. *This is what we refer to in this work as nodes*.

The next building block is the *computation card*. Two computation nodes attached to a processor card along with memory (RAM) create a computation card (two nodes). The amount of RAM per card is 1 Gbyte (512 Mbytes per compute node) [30]. Machines with 2 Gbytes per card are now available but were not used in this study. The *I/O card* is the next building block. This card is physically very similar to the computation card; however, the I/O card has the integrated Ethernet enabled for communicating with the outside world [30]. *The I/O cards and computation cards are all plugged into a node card or node board.* There are two rows of eight computation cards on the node card and 0, 1, or 2 I/O cards depending on the I/O configuration [31]. A *midplane* consists of 16 computation cards stacked in a rack. A *rack* holds two midplanes, for a total of 32 node cards or 1,024 computation nodes. The largest system currently produced is made of 64 racks for a total of 65,536 computation nodes [31].

Finally, the computation nodes may be configured at boot time in one of following ways: the first way is the virtual node mode (VN). This configuration uses both processors separately, running a different process of the user's application on each processor. The second way is the coprocessor node mode (CO). This configuration uses the secondary processor as an offload coprocessor for processing the I/O of the main processor [31].

3 Parallel Implementation

3.1 Initial Port

The PVM version of HMMER parallelizes three of these modules: *hmmcalibrate*, *hmmpfam*, and *hmmsearch*. Because PVM is not available on the BG/L system, the PVM calls needed to be converted to MPI calls. The original parallel scheme for *hmmsearch* and *hmmpfam* are depicted in Fig. 1. This figure illustrates the two postprocessing steps carried out by the master. The port was relatively straightforward. The PVM calls usually consisted of a `pvm_initsend`, any number of `pvm_pk` (pack) instructions or `pvm_upk` (unpack) statements, and a `pvm_send` or `pvm_recv`. These were replaced with either an `MPI_Send` for each `pvm_send` and an `MPI_Recv` for every `pvm_recv` or a `memcpy` for every `pvm_pk` and `pvm_upk` and `MPI_Send` and `MPI_Recv` to send and receive the entire package. Functions were constructed to pack the HMM data along with other control structures in parallel with the PVM to MPI conversions. Broadcasts, initialization functions, and other PVM functions required by HMMER were also converted to MPI equivalents. Once the conversion was done, the new MPI implementation was verified against the original single-processor code.

3.2 Optimizations

Several optimizations were introduced into the HMMER code, as detailed below, for the two modules *hmmsearch* and *hmmpfam*. These include the implementation of an alternate sequence file indexing, switching from a single-master/worker scheme to a multiple-master configuration, dynamic data collection, database caching in *hmmpfam*, and load balancing. These techniques are detailed in the next few sections.

3.2.1 Alternate Sequence File Indexing

HMM databases, if paired with a proper index file, can be positioned using the function `HMMFilePositionByIndex()`. HMMER did not employ an analogous system with sequence files since it read them sequentially. In order to more efficiently distribute sequences to the workers, it seemed like such capability would be very helpful. Therefore, a new indexing scheme for sequence files was produced. The index file is simply [sequence file name].i and contains a list of offsets into the sequence file where sequences start. This file is also made human readable, as the values are not stored as binary `ints` but printed `ints`. To access a particular sequence, `n`, the program may simply open the index file and skip to an offset proportional to the sequence number. The first number in the file is the total number of sequences. This scheme will be covered in later sections.

3.2.2 Multiple-Master Configuration

The single-master configuration for the *hmmsearch* and *hmmpfam* modules is simply not able to handle the communication traffic between the master and workers when a large number of workers are present. There are two solutions to this: either make the worker do multiple sequences at a time and send a summary of the results back in a large block or keep the existing infrastructure and simply add another management layer, a *SuperMaster* layer. That is, multiple masters will each have their own group of workers [15]. Groups of 32 were chosen for *hmmsearch* because it was at this point when parallel efficiency started to decline significantly. This number is 16 for *hmmpfam* as the master workload is more intensive. Each of these masters will operate independently from the other masters but will communicate their results to the SuperMaster, whose job is simply the collation of the data and the printing of final results. The second method has many advantages over the first. First, the workers only send back structures occasionally. These structures or traces are not

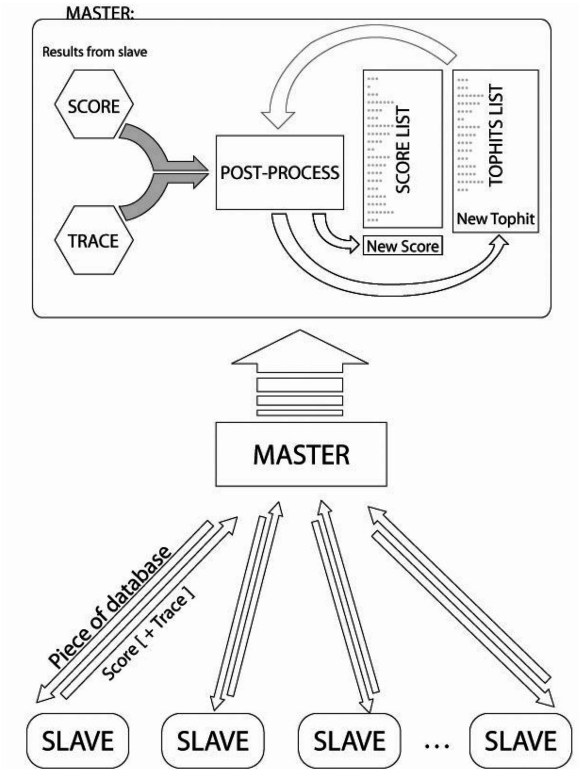


Fig. 2. New parallel master/worker relationship is basically unchanged. The master does not deal with the histogram and passes raw scores directly to the SuperMaster.

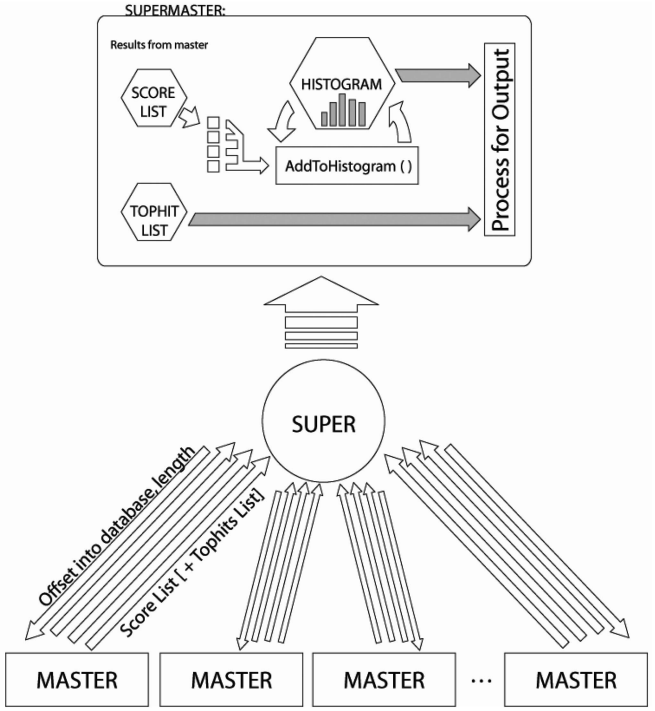


Fig. 3. SuperMaster receives a list of scores and, occasionally, the top hits lists from the master. The SuperMaster adds scores to the histogram. For load balancing, the SuperMaster sends the master offsets into the sequence database and sets of sequences when the master runs out of work.

well organized for a simple `MPI_Send` routine—that is, they have many members that are pointers to different parts of memory. In PVM, it is a routine matter to pack these members into the send buffer and send them off and, at the other end, unpack them back into a receiving structure. On BG/L, one can either perform a send for each part of the trace, eliminating the need for memory copies, or pack the information before sending by using `MPI_Pack` MPI derived data types or just manually copying the data into a contiguous buffer. Even though BG/L's communication network is incredibly fast, it was found that just copying the data manually was slightly faster than doing discrete sends.

Furthermore, using a multiple-master structure, the masters are able to do an intermediate processing step: postprocessing the score (a measure of how well a sequence matches with a particular HMM) with the trace and updating the top hit list. Now, the masters do not need to send traces back to the SuperMaster. Instead, they only send lists of scores and the list of top scores. This is much more efficient, since less MPI calls are required. Initially, these results were collected with an `MPI_Gather` at the very end. The next section describes improvements on this. To summarize, the multiple-master scheme makes use of the original code with little or no modifications but with the drawback that it removes a few nodes from the set of workers.

3.2.3 Dynamic Data Collection

As previously discussed, *hmmsearch* performs a gather operation on the scores at the end of computation and then collects the top hits from each master node. This was found to be extremely inefficient since the majority of the masters would have to wait for the other masters to finish processing their data before sending back their top hits. Therefore, a dynamic data collection scheme was implemented. In such a scheme, a value called a score list length tolerance (`sctol`) was introduced. The master nodes process one score per processed data fragment or per communication with a worker. The master allocates a buffer big enough for `sctol` floats and stores the scores in there until it is full. When this happens, the master sends back the entire buffer to the SuperMaster (see Fig. 2).

Furthermore, there is another tolerance called the top hit list length tolerance (`hitol`). Every time a new top hit is added to either top hit list, the respective counter is incremented. After the score buffer is flushed to the SuperMaster, the master checks if either of the counts has exceeded or met the top hit tolerance. If it has, the master sends back the top hit list buffer. When the SuperMaster receives the score buffer, it immediately adds each of these scores to the histogram (see Fig. 3).

What this allows is for data to be gradually fed back to the SuperMaster, so it is not overwhelmed in the final stages. This improved performance dramatically and allowed *hmmsearch* to scale well beyond 128 nodes.

There is no list of scores in *hmmpfam*, so the only difference is that a score buffer is neither sent nor received. Periodically, the master checks if the top hit buffers have been filled and flushes them back to the master. The rest of the program is the same.

3.2.4 Database Caching in hmmpfam

There is a key difference between *hmmpfam* and *hmmsearch*; *hmmsearch* searches *one* HMM against a sequence database, whereas *hmmpfam* searches *many* sequences against an HMM database such as Pfam (a large collection of multiple-sequence alignments and hidden Markov models covering many common protein domains and families). Many

challenges existed in scaling *hmmpfam*. The first was that the original PVM scheme called for individual workers to open the HMM database and read their own fragment every time they were assigned a job. When scaled to thousands of processors, this results in an excessive use of the file system that decreases scalability. Therefore, the decision was made to mimic *hmmsearch* and send the database pieces to the workers through point-to-point communication. However, if one master wants to send the fragment every time, the system suffers an overhead of almost 100 percent at 32 nodes. A new scheme was required. Because *hmmpfam* searches with many sequences, the workers use the same HMMs over and over again. However, the entire Pfam database is too large to fit in the 512 Mbytes of memory on the BG/L nodes. Our solution is that workers will cache each HMM as they receive it from the master, so it never has to be sent twice. However, an additional measure has to be put into place so that workers do not receive too many HMMs and run out of memory. To do this, masters split up the database fragment they are assigned and allocate a fraction of it for each worker such that no fragment part of the database exceeds 512 Mbytes. One block is left to load balance between the workers, this block is not assigned to any particular worker and can be distributed freely at the end. Using this method, the first sequence to be searched takes more time, but the system dramatically speeds up as more sequences are processed.

The alternative to this method is database fragment predistribution. This could be faster in some cases, especially when a worker can be set to perform a search over the entire fragment and report all their findings at once to a master. However, problem granularity is sacrificed. If database fragments are predistributed, this constrains which worker can work on a given HMM in the database. In our fluid approach, we never have to wait on any workers as long as there are database HMMs left there will always be tasks for a worker who finishes early. A database fragment predistribution scheme has been used in other implementations [26].

3.2.5 Load Balancing

One topic that has not been discussed is how the masters decide which parts of the database to process against a given query. The SuperMaster has the job of dividing the database among the masters. In the case of *hmmsearch*, this is made easy by the previously mentioned newly indexed sequence files. The SuperMaster first reads the first number in the file to determine how many sequences in total are there. In the case where no load balancing has been implemented, the SuperMaster simply divides this number by the number of masters. Then, it checks the index file for the offset of the sequence at the beginning of each master's block and sends that offset to the appropriate master along with how many subsequent sequences to do. The master simply opens the sequence file, jumps to the appropriate offset, and starts reading sequences.

Load balancing is made extremely easy by this index file. Instead of distributing all the sequences immediately, the SuperMaster distributes a certain percentage defined by a constant (say, 70 percent) in the usual manner. Then, when the masters are sending back their last set of scores, they request more sequences. The SuperMaster can either respond that there is no more work, in which case the master simply shuts down, or the SuperMaster sends another offset/length pair, and the process starts over again. The SuperMaster sends smaller and smaller chunks of the database out after the initial offering. This makes sure that one master does not take significantly longer than the others, as the closer the job is to finishing, the more granular the distribution process becomes. This optimization had modest gains but was still quite noticeable on a full rack.

After all the features are taken into account, the SuperMaster's task can be expressed with the following pseudocode:

```
for each master {
 message[0] = read_offset_from_file();
 message[1] = next_length();

 MPI_Send(message, master);< - - Tells master
 to start at offset, work on length number
      of sequences.
}

while master_count > 0 {

 MPI_Recv(scores, master);

 AddToHistogram(scores);
 total_queries− = scores_received;

 MPI_Recv(request, master);

 if(request == ghit){
   MPI_Recv(ghit, master);
   mpiUnpackHits(ghit);
}
else if(request == dhit){
 MPI_Recv(dhit, master);
 mpiUnpackHits(dhit);
}
else if(request == more_sequences){ <
- -Load balancing
 if(no_more)
   MPI_Send(no_more, master);
 else {
   message[0] = read_offset_from_file();
   message[1] = next_length();

   MPI_Send(message, master);
 }
}
else if(request == done)
 master_count-;
else
 do_nothing();
}
.......
sort_hits();
print_results();
```

The master's role is mostly unchanged, except that it no longer has to worry about printing, sorting, or adding the scores to the histogram.

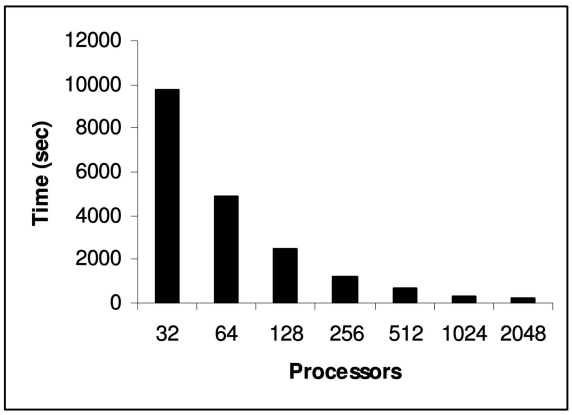


Fig. 4. *hmmcalibrate* parallel performance using the first 327 entries of the Pfam database.

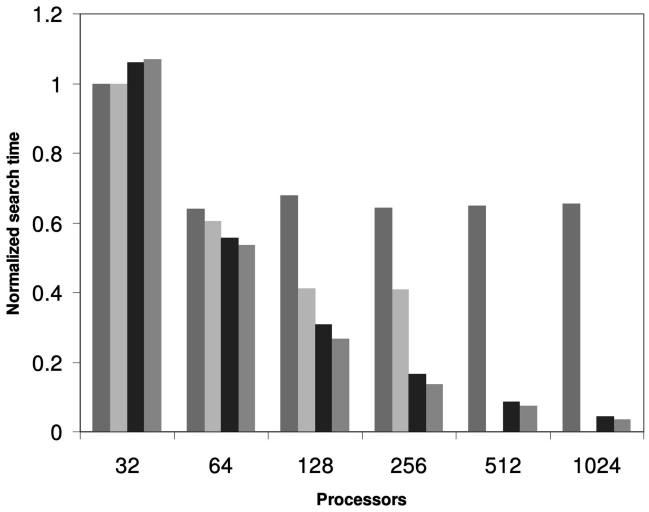


Fig. 5. *hmmsearch* parallel performance using 50 proteins of the globin family from different organisms and the UniProt release 8 database. For each processor count, the left bar shows the original PVM to MPI port. Notice scaling stops at 64 nodes. The second bar shows the multiple master implementation. The third bar shows the dynamic data collection implementation and the right bar shows the load balancing implementation.

4 Performance

4.1 Query and Databases

A multiple-sequence alignment of 50 proteins of the globin family from different organisms were used to generate an HMM profile, this HMM profile file was initially used to test *hmmcalibrate*. This provides an empirical calculation of the expectation values, and it is described in [13]. To test the scalability of *hmmcalibrate*, we created a smaller database by including only the first 327 entries of the Pfam database [33]. *hmmcalibrate* was run using two flags: the *-mpi* flag that we defined in this work and *-num*, which was empirically set to 80,000 to accommodate the large number of nodes. *-num* corresponds to a number of synthetic sequences to determine one of the internal parameters in the program [13]. The same 50 globin proteins used to initially test *hmmcalibrate* were also used with *hmmsearch* to search the UniProt Release 8 database [32]. The Pfam database [33] with the *Drosophila melanogaster* (7LESS_DROME) [13] sequence was used to test the performance of *hmmpfam* with a single sequence. In addition, we selected the first 4,000 sequences out of the Ensembl Database Release 40 [34] superset of all translations resulting from known or novel gene predictions. This last test set was selected to look at the performance of the code as a function of multiple sequences in the query.

All the tests used in this study were carried out in CO. The only exceptions are any runs that required 2,048 processors. In this case, we ran in VN since we only had a system with one rack or 1,024 nodes in CO and 2,048 in VN. All the figures show performance as a function of the different levels of optimization. All the results are cumulative.

4.2 Performance Results

The first module tested corresponds to *hmmcalibrate*. Fig. 4 summarizes the performance of this module up to 2,048 nodes. Although this module was not optimized, the parallel efficiency is still 75 percent on 2,048 nodes. Fig. 4 illustrates the performance of *hmmcalibrate* using only the first 327 entries in the Pfam database [33]. The additional HMM profiles tested showed similar performance (not included in Fig. 4).

Fig. 5 summarizes the performance of *hmmsearch* from HMMER 2.3.2 that was converted from PVM to MPI compared to our version in various stages of development.

The original MPI port scaled only to 64 nodes in CO. The first level of optimization is the use of multiple masters, which improved scaling to 128 nodes. We did not see any improvement beyond 128 nodes in CO, so only one extra point (256) beyond 128 nodes is shown in Fig. 5. Further optimization by implementing dynamic data collection improves scalability to 1,024 nodes. Finally, Fig. 5 shows that by adding load balancing, performance is improved even further.

The last module that was ported is *hmmpfam*. Fig. 6 summarizes the performance of the three versions of this code.

The first version corresponds to a direct port from PVM to MPI. The second version introduces the multiple-master scheme. The third version optimizes the reading process. Fig. 6 shows that the original version only scales to 64 nodes in CO. Implementing the multiple-master scheme increases scalability to 256 nodes in CO. Further optimization of the I/O operations combined with the multiple-master scheme

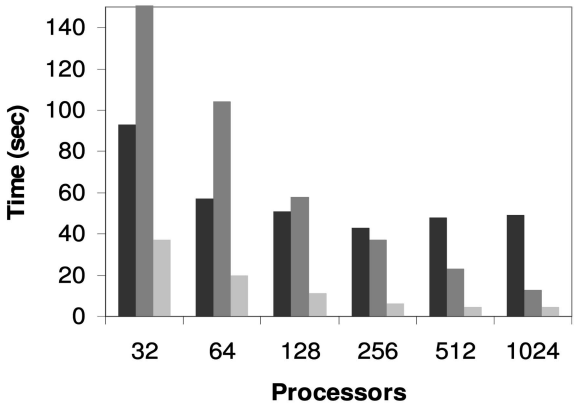


Fig. 6. *hmmpfam* parallel performance using the 7LESS_DROME single sequence and the Pfam database. For each processor count, the left bar represents the Original version of HMM 2.3.2 after the PVMto MPI port. The middle bar shows the multiple-master implementation, and the right bar shows the additional gains from I/O optimizations.

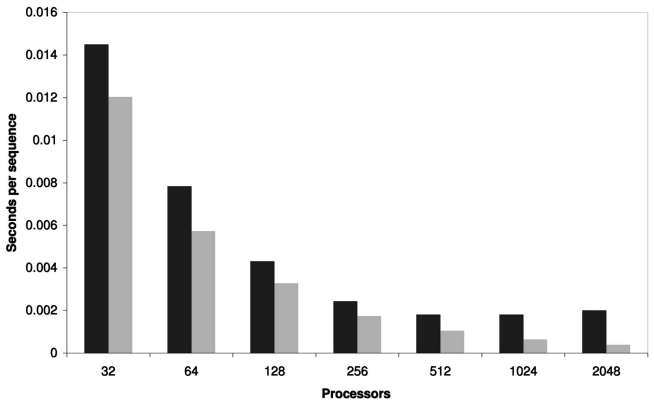


Fig. 7. *hmmpfam* parallel performance comparison between caching (black bar) and noncaching (gray bar) per character in a sequence. This test case corresponds to 4,000 sequences from the Ensembl database and Pfam database.

improves overall performance, but scalability is still only up to 256 nodes in CO.

The performance data summarized in Fig. 7 is the time per character in a sequence with and without the caching scheme.

The performance for 4,000 sequences shows a 50 percent runtime improvement from caching the database. At 1,024 nodes, the caching scheme shows a four-time speedup over the noncaching approach.

5 Discussion

The initial master/worker implementation works well on small clusters of workstations but is not well suited for a massively parallel machine. This becomes apparent when running a straight MPI port of *hmmsearch* or *hmmpfam* on greater than 32 processors. Parallel efficiency drops drastically after 32 processors, with performance gains completely stopping above 64 processors. A simple profiling of the MPI calls reveals that this parallel drop-off is caused by requests flooding the master. The master reads the query and, depending on the module, either the database or an index of the database. It then distributes work one entry at a time to the workers (see Fig. 1). When the number of workers increases, a dramatic increase in network contention between the master and workers was observed, causing reduced scalability due to increased waiting time for workers. This is observed when using 64 nodes—by the time the master sends work to worker 64, worker 1 has already completed its work and wants to send results back to the master. Thus, the master has requests in its receive buffer constantly, and its communication bandwidth is saturated.

Due to low data transfer, *hmmcalibrate* does not suffer from these parallelization issues even though it uses a similar mechanism. *hmmcalibrate* is not affected by the communication bottleneck because the workers only receive the number of sequences to generate and run against a given query. Computation time heavily outweighs the communication, especially as the number of random sequences increases. Therefore, there was no need to improve scalability for *hmmcalibrate*.

The next level of optimization corresponds to the introduction of multiple masters. As illustrated in Fig. 5, the performance is similar to the single-master version when using 32 nodes. This is not surprising since with so few nodes the single master has not become a bottleneck. The multiple-master scheme is advantageous beyond 64 nodes and scales up to 128 nodes.

Further improvements via dynamic data collection improve scalability up to 2,048 nodes. This buffering or gradual sending of the data avoids bandwidth contention and improves scalability. Since there is not a list of scores in the *hmmpfam* program, the master checks the buffer status and flushes them to the SuperMaster if necessary.

Of the three programs, *hmmpfam* performs the most I/O. This presents a problem since scaling I/O-bound programs is difficult. As previously pointed out, in the original PVM implementation, each worker is doing I/O. This is difficult when thousands of processors are trying to access the file system. In this case, we proceeded to take advantage of all the aggregate memory available on BG/L by distributing fragments of the database via point-to-point communication. However, depending on the total amount of local memory, a caching scheme was required that would include awareness of the total local memory. The idea is not to exceed local memory while still being able to distribute the database. For the cases tested here, this appeared to be a valid alternative to database predistribution.

6 Future Work

We have applied this approach to several other applications, in particular, mpiBLAST-PIO [35]. In this application, we have stored the database in memory and avoid reading files directly from a disk. With all nodes sharing the same file system, I/O contention severely limited the scaling of this application. To address this, we implemented a modified version of the "virtual file manager" (VFM). This is used to store not only the database fragments in the memory but also the query file and various temporary files. This eliminates disk I/O and allows file distribution using MPI when workers need the same file. To further increase scalability, we are implementing this scheme as part of *hmmpfam*.

Compiler and code tuning have been shown to be important to improve performance [29]. Some of the techniques that we have explored in the past include encapsulating conditionally used code, outer loop optimizations, copying data to a temporary vector to utilize prefetch streams, and global code motion of invariant control structures. These optimization techniques and others need to be explored on BG/L.

Finally, standard performance tools will be used to identify any potential bottlenecks in addition to the ones described in this section.

7 Conclusion

This paper proposes a parallel scheme to increase scalability of two of the programs in HMMER, namely, *hmmsearch* and *hmmpfam*. The third program discussed here, *hmmcalibrate*, showed good performance with the original parallel implementation. For the two other programs, we have shown that our new implementation shows good scalability on a massively parallel system. Both programs now scale to thousands of processors for the queries and databases tested here. We report several levels of optimization, starting with alternate sequence file indexing, the introduction of a multiple-master configuration, dynamic data collection, and load balancing via the indexed sequence

files. With this new implementation, we have demonstrated that *hmmsearch* and *hmmpfam* are programs well suited for a system with thousands of distributed computation nodes. The scalability is nearly linear for all the hardware configurations tested in this study.

ACKNOWLEDGMENTS

The authors would like to thank Carl Obert for his support and valuable discussions on this work. They would also like to thank the Extreme Blue and Speed Team programs at IBM Rochester, Minnesota, for sponsoring this project. Especially, they would like to thank the staff members at the OnDemand Center, Cindy Mestad, Steven M. Westerbeck, and Geoffrey S. Costigan, for providing valuable resources and assistance throughout the entirety of this project. The authors also thank the reviewers and editors for their time, effort, and constructive feedback, in particular, for bringing to their attention the work cited in [36] that was not available at the time that this work was carried out.

REFERENCES

[1] R. Durbin, S.R. Eddy, A. Krogh, and G.J. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge Univ. Press, 1998.

[2] G.D. Schuler, *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins,* A.D. Baxevanis and B.F. Francis Ouellette, eds. Wiley-Liss, Inc., 1998.

[3] D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, and D.L. Wheeler, "GenBank," *Nucleic Acids Research,* vol. 34, pp. D16-D20, 2006.

[4] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman, "Basic Local Alignment Search Tool," *J. Molecular Biology,* vol. 215, pp. 403-410, 1990.

[5] S. Altschul, T.L. Madden, A.A. Schaffer, J. Zheng, Z. Zhang, M. Miller, and D.L. Lipman, "Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs," *Nucleic Acid Research,* vol. 25, pp. 3389-3402, 1997.

[6] W.R. Pearson and D.J. Lipman, "Improved Tools for Biological Sequence Comparison," *Proc. Nat'l Academy of Science,* vol. 85, pp. 2444-2448, 1988.

[7] L.R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proc. IEEE,* vol. 77, pp. 257-286, 1989.

[8] S.R. Eddy, "Multiple Alignment Using Hidden Markov Models," *Proc. Third Int'l Conf. Intelligent Systems Molecular Biology (ISMB '95),* vol. 3, pp. 114-120, 1995.

[9] S.R. Eddy, "Hidden Markov Models," *Current Opinions in Structural Biology,* vol. 6, pp. 361-365, 1996.

[10] S.R. Eddy, "Profile Hidden Markov Models," *Bioinformatics,* vol. 14, pp. 755-763, 1998.

[11] P. Baldi, P.Y. Cauvin, T. Hunkapillar, and M.A. McClure, "Hidden Markov Models of Biological Primary Sequence Information," *Proc. Nat'l Academy of Sciences,* vol. 91, pp. 1059-1063, 1994.

[12] A. Krogh, M. Brown, I.S. Mian, K. Sjolander, and D. Haussler, "Hidden Markov Models in Computational Biology: Applications to Protein Modeling," *J. Molecular Biology,* vol. 235, pp. 1501-1531, 1994.

[13] S.R. Eddy HMMER User's Guide: Biological Sequence Analysis Using Profile Hidden Markov Models, Version 2.3.2, http://hmmer.wustl.edu/, Oct. 1998.

[14] M.Y. Galperin, "The Molecular Biology Database Collection: 2006 Update," *Nucleic Acids Research,* vol. 34, pp. D3-D5, 2006.

[15] M.K. Gardner, W.-C. Feng, J. Archuleta, H. Lin, and X. Ma, "Parallel Genomic Sequence-Searching on an Ad-Hoc grid: Experiences, Lessons Learned and Implications," *Proc. ACM/IEEE Supercomputing Conf. (SC '06),* 2006.

[16] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and Y. Sunderam, *PVM: Parallel Virtual Machine. A User"s Guide and Tutorial for Networked Parallel Computing,* J. Kowalik, ed. MIT Press, 1994.

[17] A. Darling, L. Carey, and W.-C. Feng, "The Design, Implementation and Evaluation of mpiBLAST," *Proc. ClusterWorld Conf. and Expo,* 2003.

[18] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova, "Efficient Data Access for Parallel BLAST," *Proc. 19th Int'l Parallel and Distributed Processing Symp. (IPDPS '05),* 2005.

[19] R. Bjornson, A. Sherman, S. Weston, N. Willard, and J. Wing, "TurboBLAST: A Parallel Implementation of BLAST Built on the TurboHub," *Proc. 16th Int'l Parallel and Distributed Processing Symp. (IPDPS '02),* 2002.

[20] D.R. Mathog, "Parallel BLAST on Split Databases," *Bioinformatics,* vol. 19, pp. 1865-1866, 2003.

[21] K. Hokamp, D.C. Shields, K.H. Wolfe, and D.R. Caffrey, "Wrapping Up BLAST and Other Applications for Use on Unix Clusters," *Bioinformatics,* vol. 19, pp. 441-442, 2003.

[22] C. Oehmen and J. Nieplocha, "ScalaBLAST: A Scalable Implementation of BLAST for High-Performance Data-Intensive Bioinformatics Analysis," *IEEE Trans. Parallel and Distributed Systems,* vol. 17, pp. 740-749, 2006.

[23] R.C. Braun, K.T. Pedretti, T.L. Casavart, T.E. Sheetz, C.L. Birkett, and C.A. Roberts, "Parallelization of Local BLAST Service on Workstation Clusters," *Future Generation Computer Systems,* vol. 17, pp. 745-754, 2001.

[24] J.D. Grant, R.L. Dunbrack, F.J. Manion, and M.F. Ochs, "Beo-BLAST: Distributed BLAST and PSI-BLAST on a Beowulf Cluster," *Bioinformatics,* vol. 18, pp. 765-766, 2002.

[25] H. Stockinger, M. Pagai, L. Cerutti, and L. Falquet, "Grid Approach to Embarrassingly Parallel CPU-Intensive Bioinformatics Problems," *Proc. Second IEEE Int'l Conf. One-Science and Grid Computing (e-Science '06),* 2006.

[26] W. Zhu, Y. Niu, J. Lu, and G.R. Gao, "Implementing Parallel HMM-pfam on the EARTH Multithreaded Architecture," *Computational Systems Bioinformatics,* 2003.

[27] A. Ching, W.-C. Feng, H. Lin, Y. Ma, and A. Chodhary, "Exploring I/O for Parallel Sequence-Search Tools with S3aSim," *Proc. 15th IEEE Int'l Symp. High Performance Distributed Computing,* pp. 229-240, 2006.

[28] J. Nieplocha, R. Harrison, and R. Littlefield, "Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computing," *J. Supercomputing,* vol. 10, pp. 197-220, 1996.

[29] U. Srinivasan, P.-S. Chen, Q. Diao, C.-C. Lim, E. Li, Y. Chen, R. Ju, and Y. Zhang, "Characterization and Analysis of HMMER and SVM-RFE Parallel Bioinformatics Applications," *Proc. IEEE Int'l Symp. Workload Characterization (IISWC '05),* pp. 87-98, 2005.

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

[31] O. Lascu, N. Allsopp, P. Vezolle, J. Follows, M. Hennecke, F. Ishibashi, M. Paolini, S. Prakash, H. Reddy, C.P. Sosa, A. Tabary, and D. Quintero, *Unfolding the IBM Server Blue Gene Solution.* IBM Corp., Int'l Technical Support Organization, http://www.redbooks.ibm.com/abstracts/sg246686.html?Open, 2005.

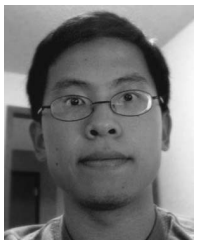
[32] C.H. Wu, R. Apweiler, A. Bairoch, D.A. Natale, W.C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M.J. Martin, R. Mazumder, C. O'Donovan, N. Redaschi, and B. Suzek, "The Universal Protein Resource (UniProt): An Expanding Universe of Protein Information," *Nucleic Acids Research,* vol. 34, pp. D187-D191, 2006.

[33] A. Bateman, E. Birney, L. Cerruti, R. Durbin, L. Etwiller, S.R. Eddy, S. Griffiths-Jones, K.L. Howe, and E.L. Sonnhammer, "The Pfam Protein Families Database," *Nucleic Acids Research,* vol. 30, pp. 276-280, 2002.

[34] E. Birney, D. Andrews, M. Caccamo, Y. Chen, L. Clarke, G. Coates, T. Cox, F. Cunningham, V. Curwen, T. Cutts, T. Down, R. Durbin, X.M. Fernandez-Suarez, P. Flicek, S. Gräf, M. Hammond, J. Herrero, K. Howe, V. Iyer, K. Jekosch, A. Kähäri, A. Kasprzyk, D. Keefe, F. Kokocinski, E. Kulesha, D. London, I. Longden, C. Melsopp, P. Meidl, B. Overduin, A. Parker, G. Proctor, A. Prlic, M. Rae, D. Rios, S. Redmond, M. Schuster, I. Sealy, S. Searle, J. Severin, G. Slater, D. Smedley, J. Smith, A. Stabenau, J. Stalker, S. Trevanion, A. Ureta-Vidal, J. Vogel, S. White, C. Woodwark, and T.J.P. Hubbard, "Ensembl 2006," *Nucleic Acids Research,* vol. 34, pp. D556-D561, 2006.

[35] O. Thorsen, B. Smith, C.P. Sosa, K. Jiang, H. Lin, A. Peters, and W.-C. Feng, "Parallel Genomic Sequence-Search on a Massively Parallel System," to be published, 2007.

[36] J.P. Walters, J. Landman, and V. Chaudhary, "Optimized Cluster-Enabled HMMER Searches," to be published in, *Grids for Bioinformatics and Computational Biology,* E.-G. Talbi and A. Zomaya, eds., John Wiley & Sons, 2007.



Karl Jiang is currently an undergraduate at the University of Miami, studying computer engineering, physics, and mathematics, and is currently working as part of a co-op program at IBM Rochester, Minnesota. He works on parallelizing life sciences applications on the Blue Gene massively parallel supercomputer. His research interests are in parallel computing, as well as sociology.

Oystein Thorsen received the BS degree in computer engineering from Agder University College, Norway, in 1999 and the master's degree in computer science from Michigan Technological University in 2006. He is a PhD student in the Department of Computer Science, Michigan Technological University. During this time, he has also worked as a research assistant, working on various topics related to high-performance computing and Unified Parallel C. He is currently part of the co-op program at IBM Rochester, Minnesota, working on life sciences applications for the Blue Gene Software Development Group.

Amanda Peters received the BS degree in both computer science and physics from Duke University in 2005. She currently works at IBM Rochester, Minnesota, on the Blue Gene supercomputers. She is involved with the porting, validating, and optimizing of life science applications. Her research area of interest is bioinformatics.

Brian Smith received BS degrees in both computer engineering and electrical engineering and the MS degree in computer engineering from Iowa State University in 2000 and January 2005, respectively. He currently works at IBM Rochester, Minnesota, on the Blue Gene supercomputers. His job responsibilities include the communications stack for Blue Gene and applications porting and optimizing. He previously worked on high-performance computing at the US Department of Energy Ames Research Laboratory. He is a member of the IEEE

Carlos P. Sosa (M'06) received the PhD degree in physical chemistry from Wayne State University. He completed his postdoctoral work at the Pacific Northwest National Laboratory. He is a senior technical staff member in the Systems and Technology Group of IBM and the technical lead of the Chemistry and Life Sciences Team in the Blue Gene/L Development Group. His work is on scientific applications with emphasis in life sciences, parallel programming, benchmarking, and performance tuning. He is the author or a coauthor of multiple peer-reviewed papers. He is also a coauthor of two IBM RedBooks. His area of interest is future POWER architectures, Blue Gene/L, and cellular molecular biology. He is a member of the IEEE, the IEEE Computer Society, the American Chemical Society, and the International Society for Computational Biology.