# Improving MPI-HMMER's Scalability With Parallel I/O*

Rohan Darole, John Paul Walters, and Vipin Chaudhary
Department of Computer Science and Engineering
University at Buffalo, The State University of New York
Buffalo, NY 14260
{rdarole, waltersj, vipin}@buffalo.edu

May 22, 2008
Technical Report # 2008-11

**Abstract**

As the size of biological sequence databases continues to grow, the time to search these databases has grown proportionally. This has led to many parallel implementations of common sequence analysis suites. However, it has become clear that many of these parallel sequence analysis tools do not scale well for medium to large-sized clusters. In this paper we describe an enhanced version of MPI-HMMER. We improve on MPI-HMMER's scalability through the use of parallel I/O and a parallel file system. Our enhancements to the core HMMER search tools, *hmmsearch* and *hmmpfam*, allows for scalability through 256 nodes where MPI-HMMER was previously limited to 64 nodes.

## 1 Introduction

As the size of biological sequence databases continue to grow exponentially, outpacing Moore's Law, the need for highly scalable database search tools increases. Because a single processor cannot effectively cope with the massive amount of data present in today's sequence databases newer MPI-enabled search tools have been created to reduce database search times. These distributed search tools have proven highly effective and have enabled researchers to investigate larger and more complex problems.

HMMER [5, 6, 4] is perhaps the second most used sequence analysis suite. MPI-HMMER is a freely available MPI implementation of the HMMER sequence analysis suite [17, 8]. MPI-HMMER is used in thousands of research labs around the world [12, 11]. In previous work it has been shown to scale nearly linearly for small to mid-sized clusters up to 64 nodes. However, as database sizes increase, the need for greater MPI-HMMER scalability has become clear.

In this paper we improve on the scalability of MPI-HMMER through the use of parallel I/O and a parallel file system. This allows us to eliminate much of the communication that previously acted

*This research was supported in part by NSF IGERT grant 9987598, MEDC/Michigan Life Science Corridor, and NYSTAR.

as a bottleneck to MPI-HMMER. By using parallel I/O, we are able to offload most communication to a cluster's dedicated I/O nodes, thereby reducing the participation of the master node in the parallel computation. Our contributions are:

- We characterize MPI-HMMER, showing its existing bottleneck
- Provide a parallel I/O implementation of MPI-HMMER to improve scalability on clusters greater than 64 nodes.

The remainder of this paper is organized as follows: in Section 2 provide a brief overview of HMMER and MPI-HMMER. In Section 3 we describe the existing HMMER acceleration work. In Sections 4 and 5 we describe our implementation and results, and in Section 6 we present our conclusions and future work.

# 2 HMMER and MPI-HMMER Background

MPI-HMMER is based on the HMMER [5, 6, 4] sequence analysis suite. HMMER allow scientists to construct profile Hidden Markov Models (HMMs) of a set of aligned protein sequences with known similar function and homology, and provides database search functionality to compare input HMMs to sequence databases (as well as input sequences to HMM databases).

HMMER includes two database search tools, *hmmsearch* and *hmmpfam*. *hmmsearch* accepts as input a profile HMM, and searches the HMM against a database of sequences (such as the NR database). The *hmmpfam* tool performs similarly, but searches one or more sequences against an HMM database (such as the Pfam database). These tools nearly perform the opposite functions from one another, with the exception that *hmmpfam* allows for searching multiple sequences against a database where *hmmsearch* restricts the input to a single HMM.

HMMER includes a PVM (parallel virtual machine) implementation of a master-worker in its source distribution. MPI-HMMER is based on this model however, its I/O improvements have led to significant speedup and scalability over the PVM implementation. In particular, MPI-HMMER uses both database fragmentation and double-buffering to reduce the overhead of message passing, and to mask (as much as possible) the communication latency.

Database fragmentation results in the master node sending a database chunk to the worker node, rather than a single database entry at each iteration. It is based on the observation that sending a small number of large messages is generally more efficient than sending a large number of short messages. We combine database fragmentation with double-buffering to hide the impact of message passing, thereby allowing a worker node to compute and return results while simultaneously receiving the next batch.

The basic structure of an *hmmsearch* is shown in Figure 1 and is described algorithmically by:

1. The master reads the HMM from disk and sends it to each worker.
2. The master reads the sequences from the sequence file and sends database fragments to each worker.
3. After receiving a database fragment, workers computes the similarity score for each sequence in the database fragment and returns the results to the master.
4. The master performs post-processing against all hits.
5. If additional sequences remain unprocessed, go to step 2.

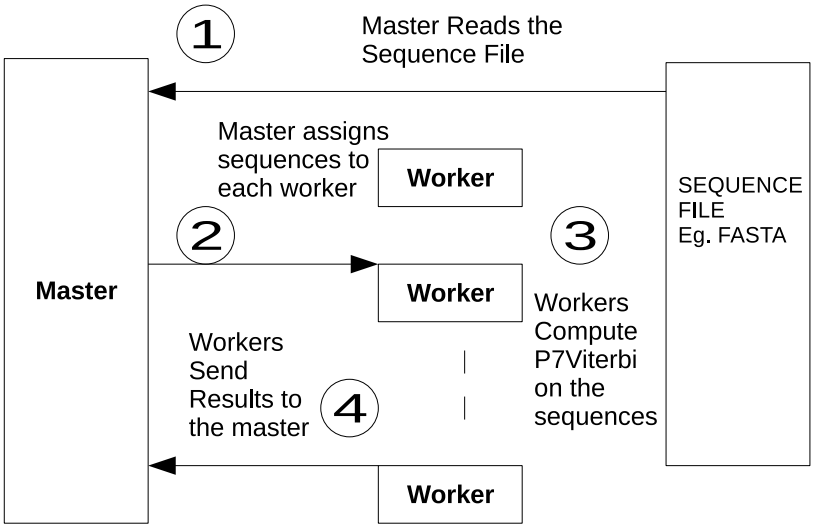


Figure 1: MPI-HMMER's *hmmsearch* design.

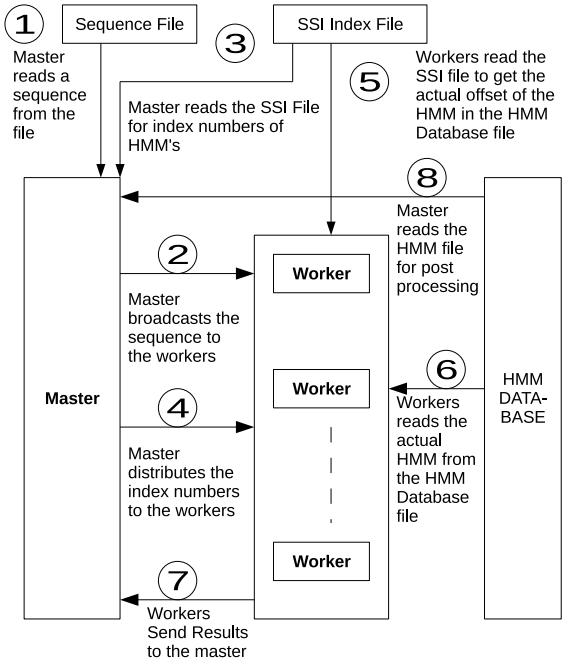


Figure 2: MPI-HMMER's *hmmpfam* design.

MPI-HMMER's *hmmpfam* MPI implementation functions similar to *hmmsearch*, except that an indexing strategy is used to help combat *hmmpfam*'s I/O-bound nature. Rather than sending distributing HMMs from the master node, the master simply distributes index chunks to the workers. The workers then directly read the HMMs from the HMM database. This means that a copy of the HMM database must be available either locally or via a network storage. This method functions similarly to our parallel I/O implementation, but relies on standard UNIX file I/O rather

than high-speed parallel I/O. The indexes are distributed using a double-buffering scheme in the same way that *hmmsearch* double-buffers the sequence database fragments. The PVM implementation of *hmmpfam* uses a similar strategy, but does not employ either double-buffering or database fragmentation.

Similarly, *hmmpfam* functions as shown in Figure 2 is described by:

1. The master node reads sequence from the input sequence file.
2. The master node broadcasts the sequence to each worker.
3. Master then reads the SSI index file to determine which HMMs the worker will process.
4. The master distributes the corresponding index numbers to the workers.
5. The worker then reads the each HMM from the HMM database based on the indexes given by the master.
6. The worker then computes the scores for a given sequence against each assigned HMM and sends the results to the master.
7. Master performs post-processing against all hits.
8. If additional HMMs remain unprocessed, go to step 3.
9. If additional sequences remain, go to step 1.

Excellent performance is achieved through 32 and 64 nodes in *hmmpfam* and *hmmsearch* respectively. Ultimately, the bottleneck in the computation is in the single master that must send and receive all results. After 32 or 64 nodes are reached the master node becomes 100% utilized between processing the results and sending a new batch of database entries. In *hmmfpam*, the problem is further exacerbated due to the I/O-bound nature of *hmmpfam*. This results in a pfam search that cannot scale beyond 32 nodes, while the compute-bound *hmmsearch* is capable of an additional doubling to 64 nodes.

# 3 Related Work

HMMER itself includes a PVM (parallel virtual machine) implementation of the core database search tools. However, its scalability is limited due to its reliance on PVM and its non-optimized message passing strategy. Because it does not handle database fragmentation, each node is given only a single sequence to process at each iteration. This results in a substantial message passing penalty for each database entry.

In addition to our existing implementation, MPI-HMMER [17, 8], there has been a variety of work in accelerating HMMER. The most closely related implementation is the IBM Bluegene/L work performed by Jiang et al. [9]. With the highly parallel Bluegene/L along with its specialized network fabric, the Jiang et al. port is capable of scaling up to 1024 nodes provided that each node is allocated its own IO coprocessor. Thus, their reported 1024 node scalability actually uses 2048 nodes. They use a hierarchical master model to help alleviate the single master bottleneck present in MPI-HMMER.

SledgeHMMER [2] is a web service designed to allow researches to perform Pfam database searches without having to install HMMER locally. SledgeHMMER includes caching of results to enable rapid look-up of precomputed searches. It also includes a parallel optimization as well as database caching of HMM databases into local memory.

ClawHMMer was the first GPU-enabled *hmmsearch* implementation and is capable of efficiently utilizing multiple GPUs in the form of a rendering cluster [7]. Other optimizations, including several FPGA implementations, have been demonstrated in the literature [15, 13, 10, 16]. FPGAs can achieve excellent performance, at the cost of exceptionally long development times. The advantage of both FPGAs and GPUs is their potential for high parallelism within a single GPU/FPGA. However, the implementations are rarely portable. Other acceleration strategies, such as the use of network processors have also been described in the literature [18].

Other sequence analysis suites have been enhanced with both multi-master and parallel IO strategies. mpiBLAST is perhaps the most used parallel sequence analysis suite [3]. Its original implementation has been further enhanced using a multi-master strategy similar to the Bluegene/L implementation described above [14].

# 4 Parallel IO Implementation

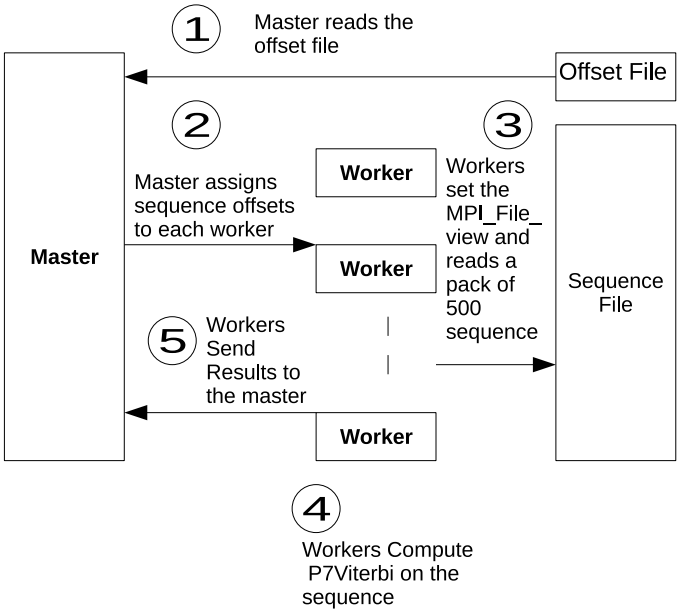


Figure 3: Parallel I/O *hmmsearch* design.

We modified the database distribution mechanism from both *hmmsearch* and *hmmpfam* to include the use of parallel MPI I/O in order to alleviate a major portion of the master node's network overhead. We use the low-level *MPI_File_read_at()* primitives to allow workers to scan to their appropriate database locations to begin computation. In Figures 3 and 4 we show the overall schematic of our parallel I/O-optimized *hmmsearch* and *hmmpfam* implementations.

We used indexing on all sequence databases to enable *hmmsearch* to perform parallel reads with distributed workers. We first pre-process the sequence database (offline) which generates an index consisting of the sequence offsets, and sequence length with one tuple per line and with the first line consisting of the total number of sequences and the total number of bytes for all sequences. This makes finding a particular sequence offset quite straightforward. It also makes

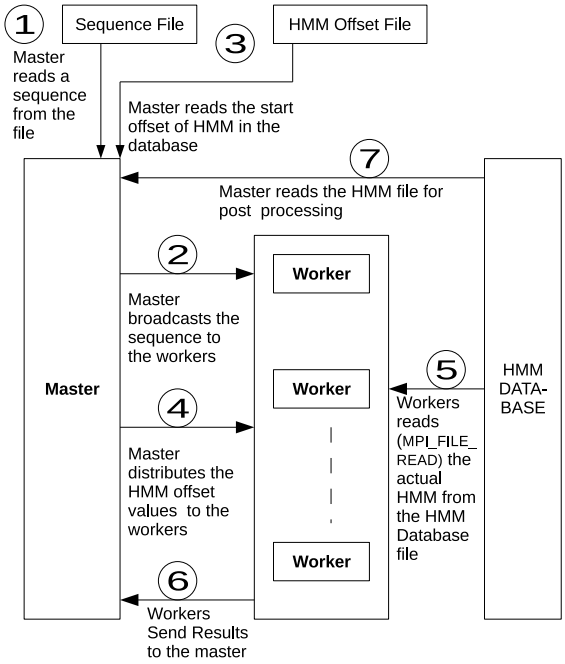


Figure 4: Parallel I/O *hmmpfam* design.

database distribution easy, allowing for distribution by either the number of sequences or the length of sequences.

Because *hmmpfam* already uses an indexing strategy, we kept a similar technique for the parallel I/O implementation. In this case, however, we modified the *hmmpfam* offset file to include a similar tuple to the *hmmsearch* implementation in order to remove the reliance on application-level indexes in *hmmpfam* searches.

## 4.1 Post-processing

The original MPI-HMMER implementation followed the PVM HMMER model in that it required all results to be returned to the master for post-processing. Making matters worse, extraneous data (for non-hits) was continually being sent back to the master node in order to facilitate the post-process. We reduce the number of messages being returned to the master node to only those that result in hits.

As a result, only database hits return extensive information to the master node. For non-hits, only a 32-bit floating point score need be returned to the master. To put things into perspective, a typical database search results in only a 2% hit rate. It is only these 2% that need to send detailed Viterbi traces to the master.

## 4.2 Database Fragmentation

MPI-HMMER was the first to introduce database fragmentation into HMMER database searching. Each fragment was a small chunk of the database, typically 12 sequences per message. This

worked well for the smaller clusters that MPI-HMMER targeted, and provided effective load balance over the duration of the computation. However, with individual worker nodes capable of reading database fragments as-needed, we found that larger database fragments were needed in order to keep all worker nodes busy.

Through experimentation, we found that approximately 500 sequences or HMMs provided a good balance of communication to computation, allowing the double-buffering to almost completely overlap with the computation. Once a node has received its overall chunk of the database, it continues to read and process 500 sequences or HMMs at every iteration.

## 4.3 Double Buffering (*hmmsearch*)

Double buffering is used to improve performance by overlapping I/O with computation. In the parallel I/O implementation of MPI-HMMER, there are two opportunities to employ double-buffering. Before a worker begins to score the first sequence in its batch, it triggers an *MPI_File_read_at()* for the next database fragment. This allows the next fragment to begin its transfer while scoring and post-processing commences.

The second opportunity to employ double-buffering is in returning the results to the master node. After a worker computes the scores for its database fragments, it returns the scores to the master node. However, because the worker need not wait for the master to acknowledge the receipt of data it may send it and proceed to compute the scores for the next database fragment.

## 4.4 Load Balancing (*hmmsearch*)

There are several strategies that could be used in order to improve the load distribution over tens or hundreds of nodes or processors. MPI's parallel I/O allows worker nodes to read data from databases without interaction with a master node. Thus, the most obvious strategy is to pre-distribute the database at the beginning of computation. That is, the master node sends each worker a start and finish offset into the database, and workers simply compute on their database portion.

However, this is only minimally effective due to the scoring algorithm's reliance on both the sequence length and the HMM lengths. In Listing 1 we provide a short code listing of the most time-consuming portion of HMMER's Viterbi scoring algorithm, *P7Viterbi()*. Here we can see that HMMER is dependent on both the length of the sequence (the outer $i$-loop), as well as the length of the HMM (the inner $k$-loop, lines 2-20. By simply allocating database chunks (either from an HMM database or sequence database) based on only the total number of sequences or HMMs, there will be a natural load imbalance due to differing sequence and HMM lengths within the database.

Our solution was to instead allocate database fragments based on the lengths of the sequences. Because the master node knows both the total number of sequences in the database, as well as the total lengths of all sequences, it is able to allocate fragments of the database based on those lengths rather than the number of entries. In this manner, a node that is allocated a database fragment with many long sequences is allocated fewer total sequences in order to maintain a reasonable balance of computation.

```
 1  for (i = 1; i <= L; i++) {
      for (k = 1; k <= M; k++) {
 3      mc[k] = mpp[k-1]   + tpmm[k-1];
        if ((sc = ip[k-1]  + tpim[k-1]) > mc[k]) mc[k] = sc;
 5      if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k]) mc[k] = sc;
        if ((sc = xmb  + bp[k])         > mc[k]) mc[k] = sc;
 7      mc[k] += ms[k];
        if (mc[k] < -INFTY) mc[k] = -INFTY;
 9
        dc[k] = dc[k-1] + tpdd[k-1];
11      if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
        if (dc[k] < -INFTY) dc[k] = -INFTY;
13
        if (k < M) {
15        ic[k] = mpp[k] + tpmi[k];
          if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
17        ic[k] += is[k];
          if (ic[k] < -INFTY) ic[k] = -INFTY;
19      }
      }
21  }
```

Listing 1: The most time consuming portion of the *P7Viterbi* algorithm.

### 4.5 Database Caching (*hmmpfam*)

Because of the design of *hmmpfam*, multiple input sequences may be searched against the HMM database. This is not true of *hmmsearch*, where a database search is limited to a single input HMM. This presents a simple opportunity for optimizing the performance of *hmmpfam* in that we may cache the HMM database entries, either in memory or on local storage, for subsequent sequence iterations. We are not the first to implement this feature [9]. However, we include the caching due to its simplicity and effectiveness.

## 5 Results

In this section we describe the actual performance of our parallel I/O enabled *hmmsearch* and *hmmpfam* implementations. All tests were carried on out at the University at Buffalo's Center for Computational Research (CCR) [1]. The CCR's hardware resources consist of 1056 nodes, each equipped with 2 3.2 Ghz Intel Xeon processors, 2048 MB RAM, gigabit ethernet and Myrinet 2G network interfaces, and an 80 GB SATA hard disk.

For our tests we used the gigabit ethernet network interface. In previous tests, no substantial improvement was found with the use of the Myrinet interfaces. For mass storage the CCR includes a 25 TB (usable) EMC CX700-based SAN as well as as 25 TB Ibrix parallel file system. The Ibrix file system includes 21 segment servers (often called I/O nodes by other parallel file systems). The Ibrix file system's physical storage exists as a pool of storage on the EMC SAN. Segment servers

are connected to the EMC SAN via fiber channel.

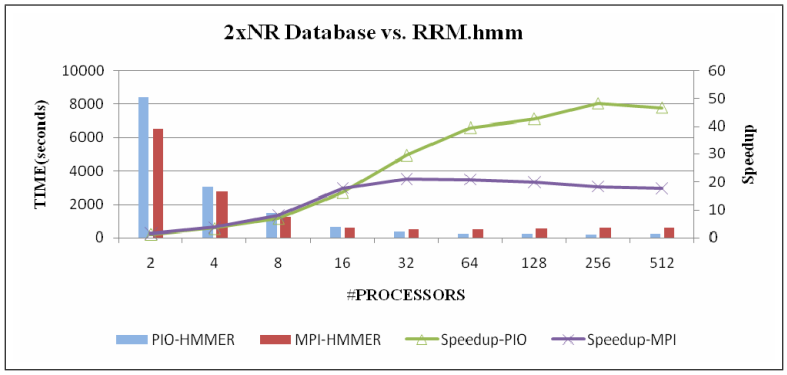


Figure 5: Comparing MPI-HMMER and PIO-HMMER for 77 state HMM and small database.

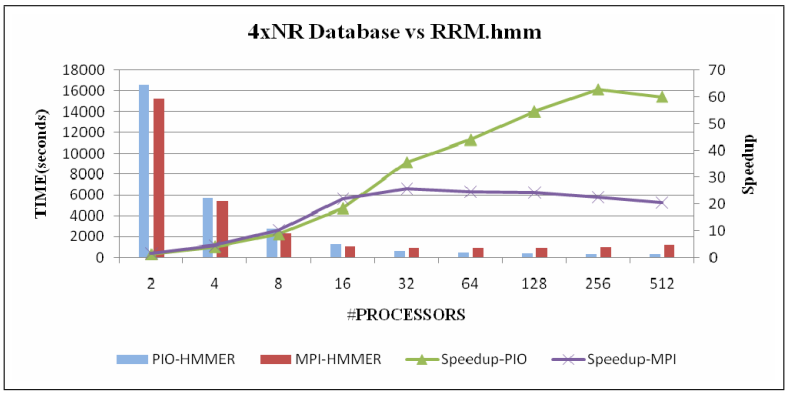


Figure 6: Comparing MPI-HMMER and PIO-HMMER for 77 state HMM and large database.

## 5.1 *hmmsearch* Performance

Our *hmmsearch* results present data for two HMMs and two database sizes. We doubled and quadrupled the NR database to provide sufficient data for large-scale analysis. Because the *P7Viterbi* algorithm is sensitive to the size (number of states) of the input HMM, we have tested our enhancements for both a 77 state HMM and a 236 state HMM.

In Figures 5 and 6 we show the performance of the smaller 77 state HMM (named rrm in the HMMER distribution) against both sized databases. This represents a worst-case performance of both HMMER implementations as the smaller HMM is both computationally light-weight and generates tens of thousands of hits against the NR database. Clearly, the parallel I/O implementation outperforms standard MPI-HMMER by a wide margin. While both show a slight improvement with the larger database, we can see that MPI-HMMER tops off and begins exhibit performance degradation at 32 processors. The parallel I/O implementation, however, continues to demonstrate speedup through 256 nodes for both database sizes.

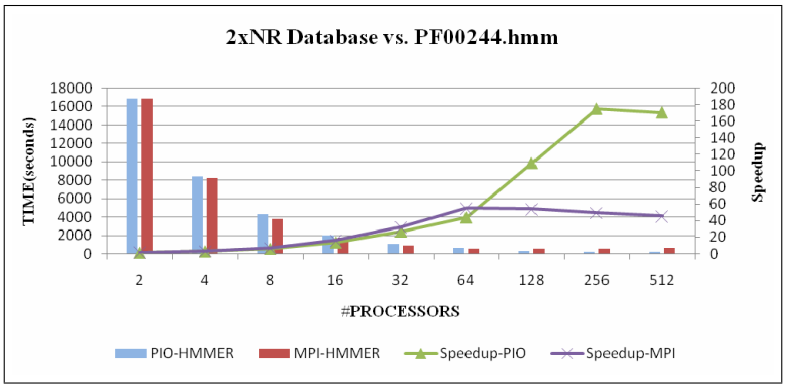


Figure 7: Comparing MPI-HMMER and PIO-HMMER for 236 state HMM and small database.

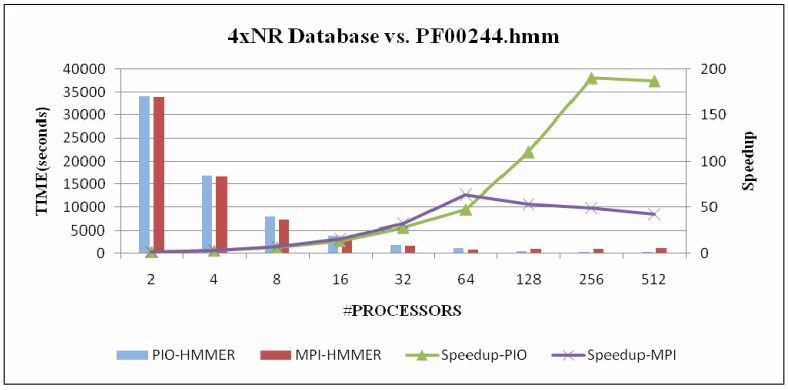


Figure 8: Comparing MPI-HMMER and PIO-HMMER for 236 state HMM and large database.

In Figures 7 and 8 we compare the 236 state HMM (a 14-3-3 protein named PF00244.hmm) against both databases. Again the parallel I/O implementation performs well through 256 processors. However, the larger HMM now contributes a greater amount of computation, demonstrating the compute-bound nature of *hmmsearch*. Indeed, for a cluster of size 256 processors we are able to achieve nearly **190x** performance, far outpacing the MPI-HMMER implementation. MPI-HMMER, however, exhibits better performance for smaller cluster sizes and is able to maintain almost perfectly linear speedup through 64 nodes, provided the database is large enough to provide adequate computation. This is due to the fine grained load balancing of MPI-HMMER where much smaller database fragments are distributed to nodes as-needed. In the parallel I/O implementation, all portions of the database are allocated at the beginning of computation, based on sequence lengths. Such static load balancing cannot account for the individual variances at run-time.

The performance impact of our I/O optimization is shown in Figure 9. As we show, reducing the I/O between the master and the worker nodes represents the single greatest performance impact of all optimizations. Indeed, our implementation improves by a factor of nearly **1**50x between the non-optimized and the optimized cases. By reducing the amount of communication both the master and workers spend less time communicating, instead spending a greater proportion of their time in computation.

In Figure 10 we compare the result of two database distribution schemes. In the non-load

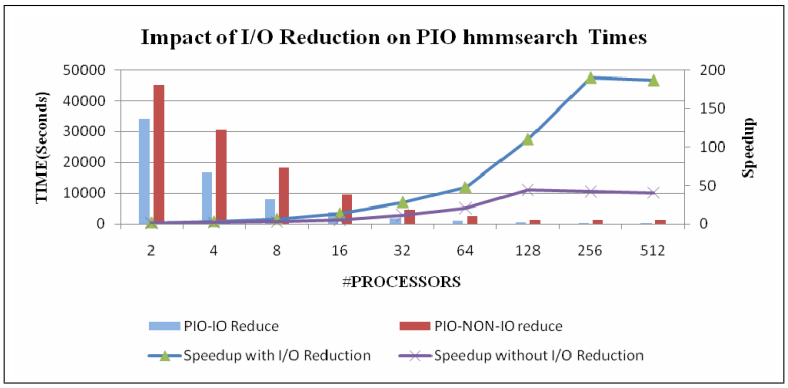


Figure 9: The impact of reducing the I/O from worker to master.

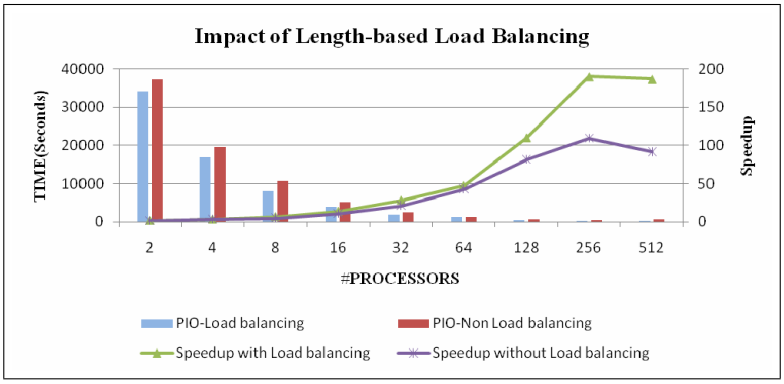


Figure 10: Length-based load balancing vs. static work division.

balanced case, all offsets are distributed equally among processors. Thus, for $n$ processors and a database consisting of $p$ entries, each processor is allocated $\frac{p}{n}$ sequences. Through experimentation we found that this results in considerable imbalance, with many processors completing their assigned work and waiting for several processors to finish. Because the *P7Viterbi* algorithm is sensitive to the length of the sequence, we changed the load distribution scheme to allocate an approximately equal length of sequences to each processor. This provided up to an **80x** performance improvement over the non-load balanced solution.

Finally, we present the results of our parallel I/O *hmmsearch* with each processor accessing a dedicated network card in Figure 11. By dedicating a network interface to each processor, we are able to achieve a speedup of **220x** on 256 nodes as opposed to the **190x** performance achieved with two processors per node. Thus for a cluster of only 256 processors we demonstrate *hmmsearch* performance that is comparable to, and possibly exceeding, the performance of the Jiang et al. BG/L implementation [9]. This performance is achieved at only a small fraction of the cost of a BG/L, using gigabit ethernet and a commercially available parallel file system using only commodity components.

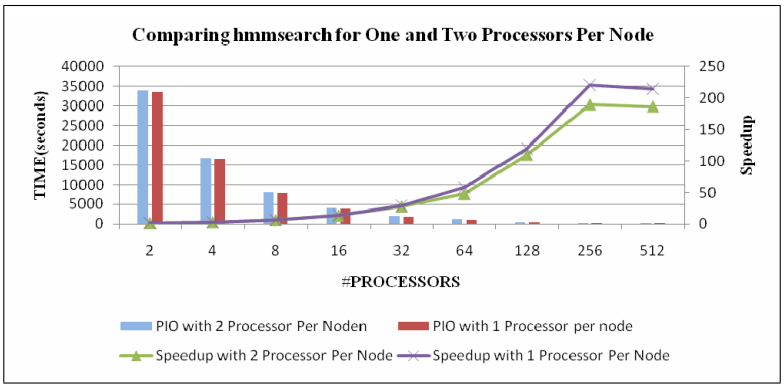


Figure 11: *hmmsearch* performance with dedicated network interfaces.

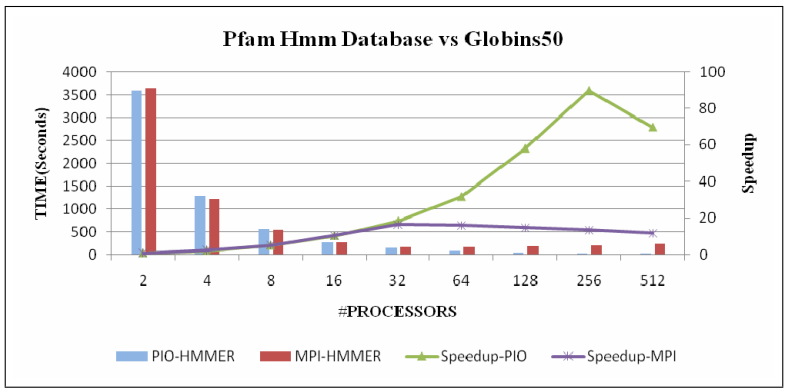


Figure 12: Basic *hmmpfam* implementation without database caching.

## 5.2 *hmmpfam* Performance

In this section we present the results of our *hmmpfam* implementation. We compare a 50 sequence globins input to the Pfam HMM database, and show results for both database caching and non-database caching. In Figure 12 we show the performance of *hmmpfam* without HMM database caching. In this case, each *hmmpfam* process is required to re-read the Pfam database for all 50 sequences. As can be seen, MPI-HMMER's performance peaks at 32 nodes. This is due to the I/O-bound nature of *hmmpfam* and the lack of communication optimization in MPI-HMMER. Thus, a great deal of communication is required for an MPI-HMMER *hmmpfam* search. The parallel I/O implementation, however, again scales to 256 nodes. This results in a per-sequence time of only 0.62 seconds compared to MPI-HMMER's best per-sequence time of 3.4 seconds.

In Figure 13 we improve on the per-sequence *hmmpfam* times by utilizing database caching for subsequent sequence iterations. As a result we are able to improve the overall scalability to **287x** at 256 nodes. This results in a per-sequence search time of only 0.19 seconds. We expect to further improve the search time for the first (non-cached) sequence by implementing double-buffering and improved load balancing (such as that used in our *hmmsearch* implementation).

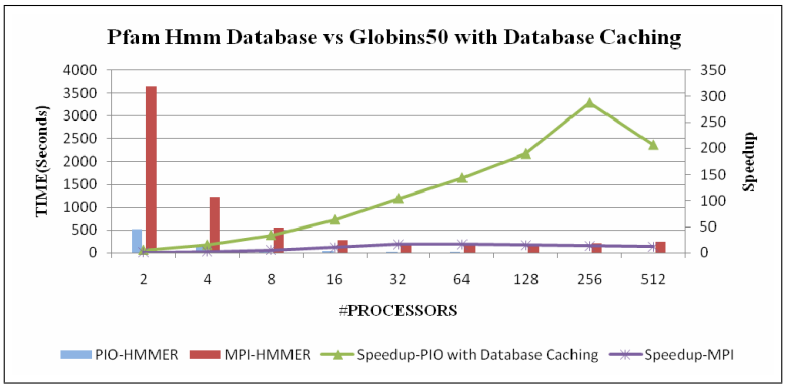


Figure 13: *hmmpfam* implementation using database caching.

# 6 Conclusions and Future Work

In this paper we have shown that MPI parallel I/O can be effectively applied to the MPI-HMMER implementation of the HMMER sequence analysis suite to achieve exceptional performance on commodity hardware. We have demonstrated BG/L performance on a commodity cluster, using an inexpensive network (gigabit ethernet) and the Ibrix parallel file system. Similar hardware is commonly accessible worldwide. Due to Ibrix's lack of file striping, we hope to further test both *hmmsearch* and *hmmpfam* on more suitable parallel file systems, such as PVFSv2. PVFSv2, in particular, provides optimized MPI I/O support as well as file striping over multiple I/O nodes. We hope to further improve the load balancing strategy to allow for a more dynamic run-time-balanced computation. Finally, we expect to extend the remaining *hmmsearch* optimizations (load balancing, double-buffering, etc.) to *hmmpfam* in order to further reduce the compute time of non-cached searches.

# Acknowledgements

We would like to gratefully acknowledge Muzammil Hussain for various comments and discussions regarding this project and its implementation.

# References

[1] University at Buffalo. The Center for Computational Research. `http://www.ccr.buffalo.edu`, 2006.

[2] G. Chukkapalli, C. Guda, and S. Subramaniam. SledgeHMMER: A Web Server for Batch Searching the Pfam Database. *Nucleic Acids Research*, 32(Web Server issue), 2004.

[3] A. Darling, L. Carey, and W. Feng. The Design, Implementation, and Evaluation of mpiBLAST. In *4th International Conference on Linux Clusters: The HPC Revolution 2003 in conjunction with the ClusterWorld Conference and Expo*, 2003.

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

[5] S. Eddy. HMMER: Profile HMMs for Protein Sequence Analysis. `http://hmmer.janelia.org`, 2006.

[6] S. R. Eddy. Profile Hidden Markov Models. *Bioinformatics*, 14(9), 1998.

[7] D. R. Horn, M. Houston, and P. Hanrahan. ClawHMMER: A Streaming HMMer-Search Implementation. In *In SC '05: The International Conference on High Performance Computing, Networking and Storage*, 2005.

[8] J. P. Walters. MPI-HMMER. `http://www.mpihmmer.org/`, 2008.

[9] K. Jiang, O. Thorsen, A. Peters, B. Smith, and C. P. Sosa. An Efficient Parallel Implementation of the Hidden Markov Methods for Genomic Sequence Search on a Massively Parallel System. *Transactions on Parallel and Distrbuted Systems*, 19(1):15–23, 2008.

[10] R. P. Maddimsetty, J. Buhler, R. Chamberlain, M. Franklin, and B. Harris. Accelerator Design for Protein Sequence HMM Search. In *Proc. of the 20th ACM International Conference on Supercomputing (ICS06)*, pages 287–296. ACM, 2006.

[11] MPI-HMMER. Mpi-hmmer at ccr. `http://www.ccr.buffalo.edu/display/WEB/Application+Software+By+Subject`, 2008.

[12] MPI-HMMER. Mpi-hmmer on bigred. `http://racinfo.uits.indiana.edu/bioinformatics/mpi-hmmer-bigred.shtml`, 2008.

[13] T. F. Oliver, B. Schmidt, J. Yanto, and D. L. Maskell. Acclerating the Viterbi Algorithm for Profile Hidden Markov Models using Reconfigurable Hardware. *Lecture Notes in Computer Science*, 3991:522–529, 2006.

[14] 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. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 59–68, New York, NY, USA, 2007. ACM.

[15] TimeLogic BioComputing Solutions. DecypherHMM. `http://www.timelogic.com/`, 2006.

[16] J. P. Walters, X. Meng, V. Chaudhary, T. F. Oliver, L. Y. Yeow, B. Schmidt, D. Nathan, and J. I. Landman. MPI-HMMER-Boost: Distributed FPGA Acceleration. *VLSI Signal Processing*, 48(3):223–238, 2007.

[17] J. P. Walters, B. Qudah, and V. Chaudhary. Accelerating the HMMER Sequence Analysis Suite Using Conventional Processors. In *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06)*, pages 289–294, Washington, DC, USA, 2006. IEEE Computer Society.

[18] B. Wun, J. Buhler, and P. Crowley. Exploiting Coarse-Grained Parallelism to Accelerate Protein Motif Finding with a Network Processor. In *PACT '05: Proceedings of the 2005 International Conference on Parallel Architectures and Compilation Techniques*, 2005.