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

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

Abstract

We present PIO-HMMER, an enhanced version of MPI-HMMER. PIO-HMMER improves on MPI-HMMER's scalability through the use of parallel I/O and a parallel file system. In addition, we describe several enhancements, including a new load balancing scheme, enhanced post-processing, improved double-buffering support, and asynchronous I/O for returning scores to the master node. Our enhancements to the core HMMER search tools, hmmsearch and hmmpfam, allow for scalability up to 256 nodes where MPI-HMMER previously did not scale beyond 64 nodes. We show that our performance enhancements allow hmmsearch to achieve between 48x and 221x speedup using 256 nodes, depending on the size of the input HMM and the database. Further, we show that by integrating database caching with PIO-HMMER's hmmpfam tool we can achieve up to 328x performance using only 256 nodes.

1 Introduction

As the size of biological sequence databases continue to grow exponentially, out pacing 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 [4–6] is perhaps the second most widely used sequence analysis suite. MPI-HMMER is a freely available MPI implementation of the HMMER sequence analysis suite [8, 17]. MPI-HMMER is used in thousands of research labs around the world. 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 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 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. We call our new implementation PIO-HMMER. 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 we 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.

2 HMMER and MPI-HMMER Background

HMMER is a sequence analysis suite that allows 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) [4–6]. It 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 [11]). The *hmmpfam* tool performs similarly, but searches one

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

or more sequences against an HMM database (such as the Pfam database [13]). 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 master-worker style PVM (parallel virtual machine) implementation 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.

We use the term "database fragmentation" to refer to the sending of multiple database elements (sequences or HMMs) rather than a single database element per message. 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.

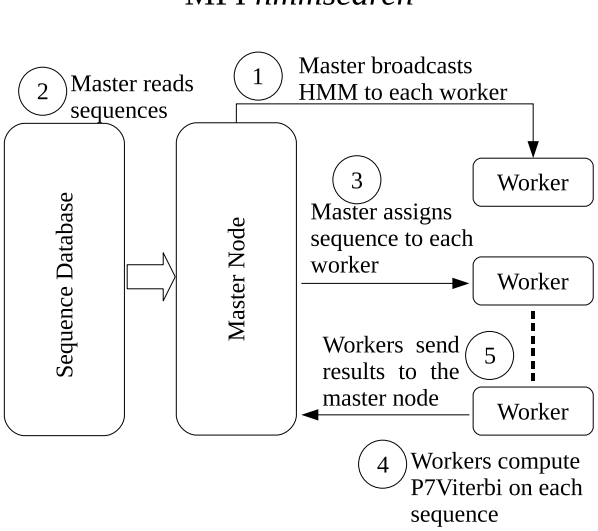

Figure 1. MPI-HMMER's *hmmsearch* design. For clarity we omit the double-buffering details. This closely mirrors the PVM implementation.

The basic structure of *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 a batch of sequences (database fragment) from the sequence database.
3. The master then sends a database fragment to each worker.
4. After receiving a database fragment, workers compute the similarity score for each sequence.
5. Workers return the results to the master node for post-processing.
6. If additional sequences remain unprocessed, go to step 2.

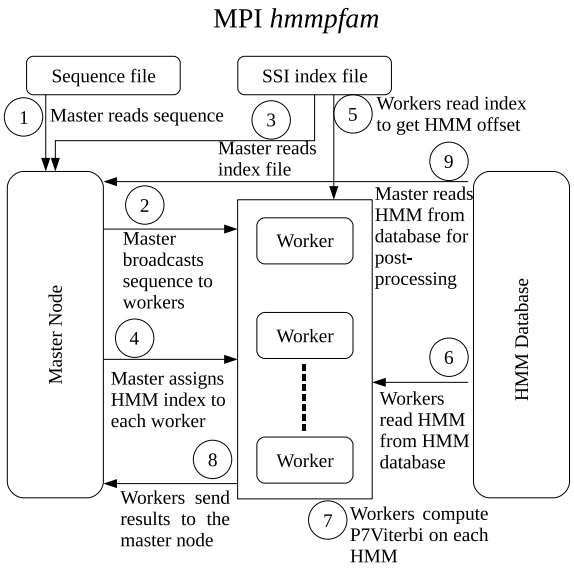

Figure 2. MPI-HMMER's *hmmpfam* design. *hmmpfam* is more I/O intensive and as a result exhibits lower scalability. Using database fragmentation helps to keep nodes busy while reducing communication.

MPI-HMMER's *hmmpfam* implementation functions similar to *hmmsearch*, except that an indexing strategy is used to help combat *hmmpfam*'s I/O-bound nature. Rather than 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 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 se-

quence database fragments. The PVM implementation of *hmmpfam* uses a similar strategy, but does not employ either double-buffering or database fragmentation.

hmmpfam functions as shown in Figure 2 and is described by:

1. The master node reads a sequence from the input sequence file.
2. The master node broadcasts the sequence to each worker.
3. The master then reads an index file to determine which HMMs each worker will process.
4. The master distributes the corresponding indexes to the workers.
5. The workers then obtain an offset into the HMM database based on the indexes.
6. The workers read each HMM from the HMM database based on the offsets obtained in step 5.
7. The worker computes the scores for a given sequence against each assigned HMM
8. The workers return the scores to the master node.
9. Master performs post-processing against all hits.
10. If additional HMMs remain unprocessed, go to step 3.
11. 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. The master eventually becomes becomes 100% utilized between processing the results and sending a new batch of database entries. In *hmmfpam*, the problem is further exacerbated due to its I/O-bound nature.

3 Related Work

HMMER itself includes a PVM implementation of the core database search tools. However, its scalability is limited as the PVM HMMER implementation does not utilize non-blocking I/O nor parallel I/O. Further, because it does not include support for 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. The database fragmentation could be easily ported to HMMER's PVM implementation; however, without non-blocking sends the full complement of the MPI-HMMER optimizations is not possible.

In addition to our existing implementation, MPI-HMMER [8, 17], 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]. Given 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 I/O coprocessor. 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 researchers 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 [10, 12, 15, 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 I/O 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 I/O Implementation

In this section we describe the major enhancements that have been added to PIO-HMMER. In addition to parallel I/O, we include enhanced post-processing, database fragmentation, double buffering, load balancing, and database caching that complement the parallel access to sequence databases. In this section we describe each of these enhancements.

We modified the database distribution mechanism for 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_iread_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 preprocess the sequence database (offline) which generates an index consisting of the sequence offsets, and sequence length with one tuple per

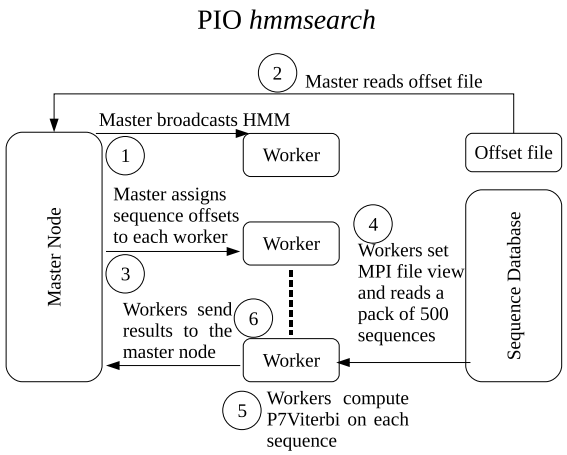

Figure 3. Parallel I/O *hmmsearch* design. The master pre-distributes indexes to the workers and the workers read from the database at their designated locations. Double buffering details are omitted for clarity.

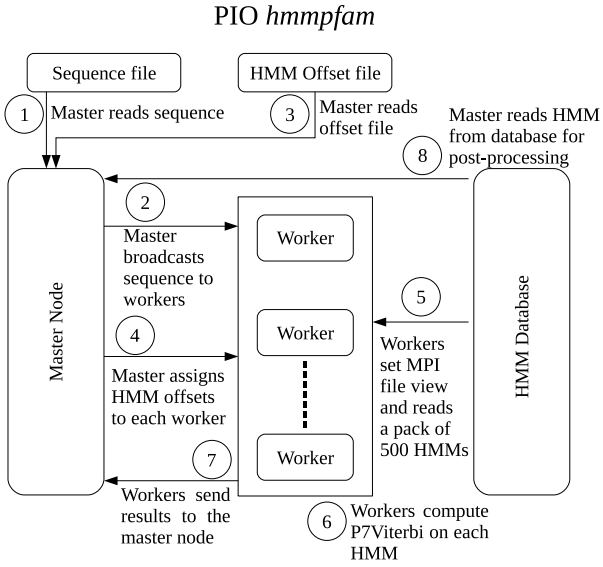

Figure 4. Parallel I/O *hmmpfam* design. Workers read directly from the parallel file system as-needed.

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 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 used a similar technique for its 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 is 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 return 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 balancing over the duration of the computation. However, by using parallel I/O, 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 chunk of the database, it continues to read and process 500 sequences or HMMs at every iteration.

4.3 Double Buffering (*hmmsearch*) and Asynchronous I/O

Double buffering is used to improve performance by overlapping I/O with computation. In the parallel I/O implementation of MPI-HMMER, we are able to

employ both double-buffering and asynchronous I/O. Double-buffering is used by the workers to keep their input sequence buffers full. Before a worker begins to score the first sequence in its batch, it triggers an *MPI_File_iread_at()* for the next database fragment. This allows the next fragment to begin its transfer while scoring and post-processing commences.

Asynchronous I/O is used when 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 immediately 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 portion of the database. However, this strategy is only minimally effective due to HMMER's scoring algorithm relying on the lengths of both the HMM and sequence. 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 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 (stored on line 1 of the index file), 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. The master can perform this job by traversing the index once, thus the serialization of this step is quite minimal and does not contribute undue overhead to the implementation.

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 [2, 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 2x3.2 GHz Intel Xeon processors, 2GB RAM, gigabit ethernet, 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 network. For mass storage the CCR includes a 25 TB (usable) EMC CX700-based SAN as well as a 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.

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, the RNA recognition motif) against both databases. This represents a worst-case performance of both HMMER implementations as the smaller HMM is both computationally lightweight 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 levels off and begins to exhibit performance degradation at 32 processors. The parallel I/O implementation, however, continues to demonstrate nonlinear speedup through 256 nodes for both database sizes.

In Figures 7 and 8 we compare the 236 state HMM (a 14-3-3 protein) against both databases. Again the parallel I/O implementation performs well through 256

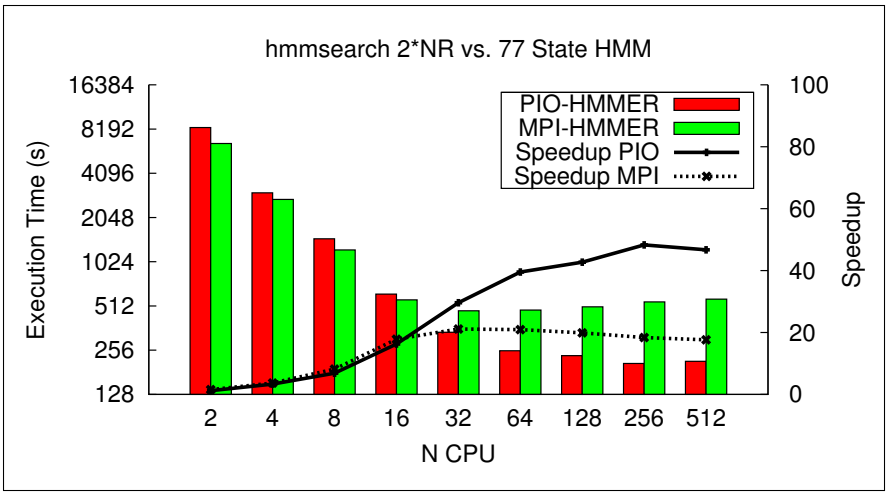

Figure 5. Comparing MPI-HMMER and PIO-HMMER for 77 state HMM and small database. The small HMM and small database size results in poor overall scalability. PIO-HMMER peaks at 48x compared to MPI-HMMER's 21x speedup.

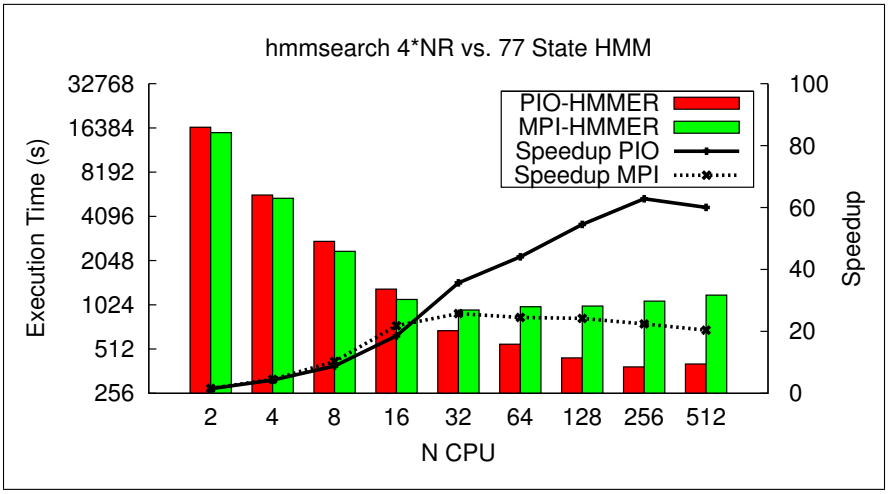

Figure 6. Comparing MPI-HMMER and PIO-HMMER for 77 state HMM and large database. The larger database provides more computation, resulting in a PIO-HMMER speedup of 63x compared to MPI-HMMER's peak of 24x.

processors. However, the larger HMM now results in a greater amount of computation, demonstrating the compute-bound nature of *hmmsearch*. Indeed, for a cluster of 256 processors we are able to achieve 190x speedup (see Figure 8), far out pacing 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 as long as 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 communication between the master and the worker nodes represents the single greatest performance impact of all op-

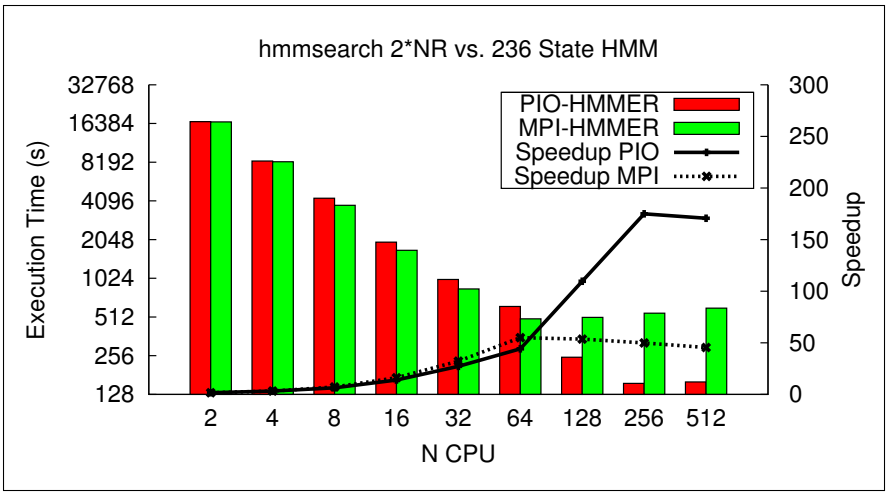

Figure 7. Comparing MPI-HMMER and PIO-HMMER for 236 state HMM and small database. The larger HMM provides a more compute-intensive workload and reduces some of the network contention on the master node. PIO-HMMER is able to achieve 175x speedup at 256 nodes while MPI-HMMER peaks at 55x speedup at 64 nodes.

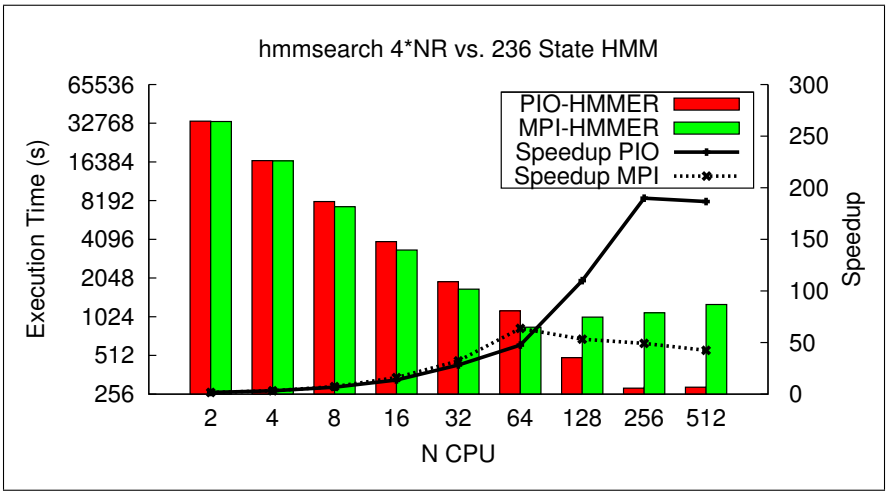

Figure 8. Comparing MPI-HMMER and PIO-HMMER for 236 state HMM and large database. Even with a large database and HMM, MPI-HMMER is unable to achieve speedup beyond 64 nodes due to the bottleneck on the master, while PIO-HMMER achieves speedup through 256 nodes.

timizations. Indeed, our implementation improves from 42x to 190x between the non-optimized and the optimized cases. By reducing the amount of communication, both the master and workers spend a greater proportion of their time performing useful computation.

In Figure 10 we compare the result of two database distribution schemes. In the non-load 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 others 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 nearly a doubling of speedup in the load-balanced case, from 109x to 190x

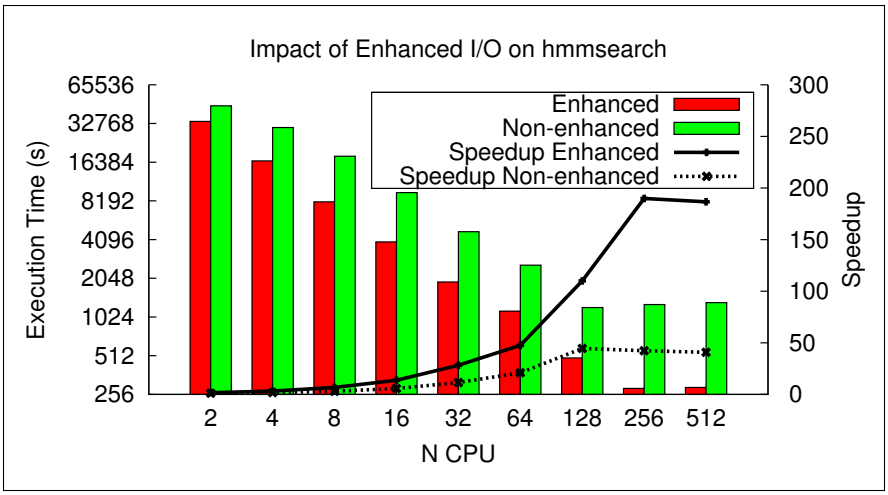

Figure 9. The impact of reducing communication from worker to master. Extraneous information to facilitate post-processing is no longer needed. As a result we improve performance from 42x to 190x.

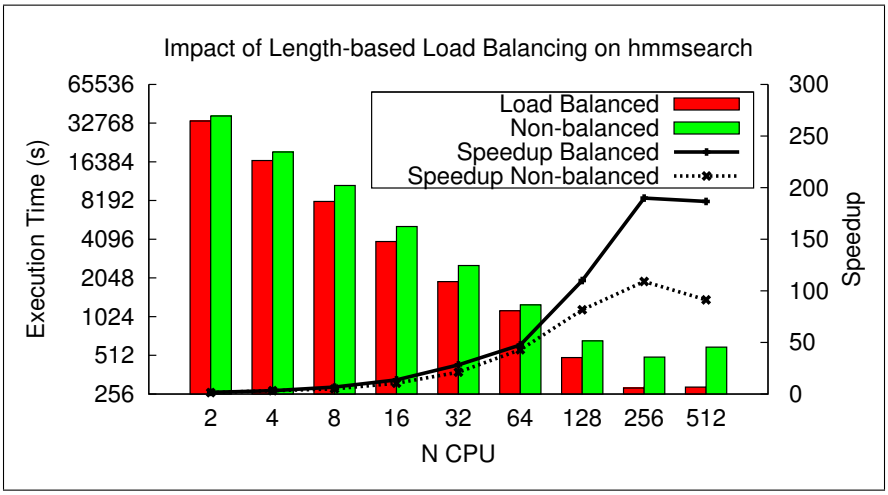

Figure 10. Length-based load balancing vs. static work division. Nodes are allocated sequences based on sequence lengths. The entire database is allocated before beginning computation. We nearly double the speedup through this approach from 109x in the non-balanced case to 190x with load balancing.

speedup.

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. While we cannot directly compare our implementation to the Jiang et al. Bluegene/L work (see [9]) due to their presentation of normalized results rather than measured timings, we believe that we are competitive, particularly in terms of raw runtime. Further, this performance is achieved at only a small fraction of the cost of a Bluegene/L, using gigabit ethernet and a commercially available parallel file system, all of which are commodity components.

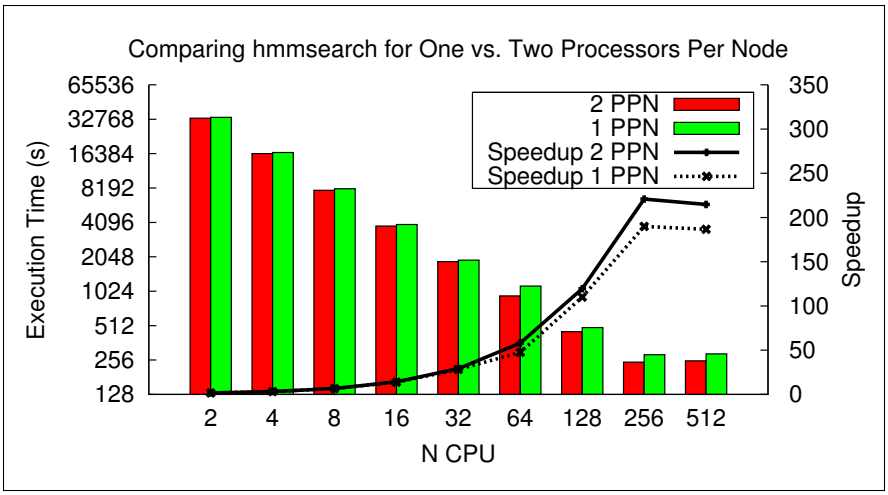

Figure 11. *hmmsearch* **performance with dedicated network interfaces. We show an improvement from 190x to 221x by allowing each node dedicated access to a network interface.**

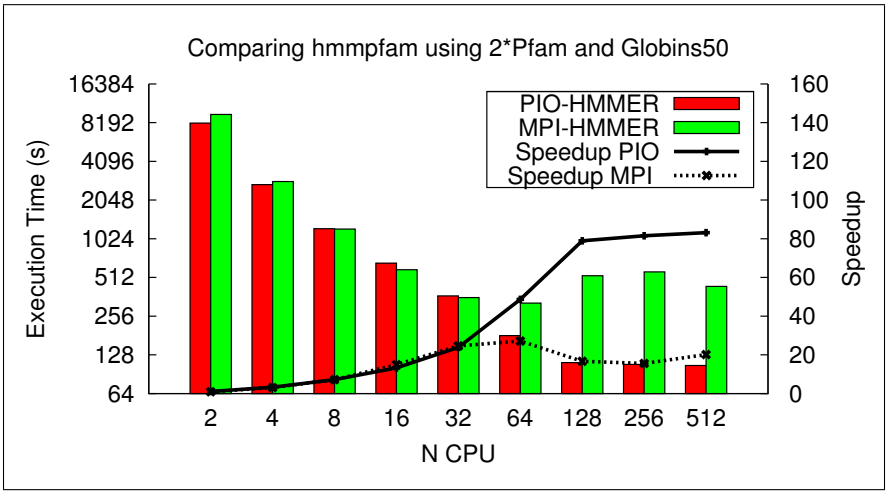

Figure 12. Basic *hmmpfam* **implementation without database caching. For each sequence query we read from the HMM database. Subsequent sequence searches have the same I/O characteristics as the first sequence. We achieve a maximum speedup of 83x.**

5.2 *hmmpfam* Performance

In this section we present the results of our *hmmpfam* implementation. We compared 50 globin sequences to the Pfam HMM database, and show results for both database caching and non-database caching. In order to provide adequate computation for large numbers of processors we doubled the size of the Pfam database from from 800 MB to 1.6 GB, resulting in 20,680 HMMs. In Figure 12 we show the performance of *hmmpfam* without HMM database caching. In this case, each *hmmpfam* process must re-read the Pfam database for all 50 sequences. From Figure 12, we can see that MPI-HMMER's performance peaks at 64 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 each MPI-HMMER *hmmpfam* search. PIO-HMMER, however, is able to achieve nonlinear speedup through 512 nodes. This results in a per-sequence time of only 2.1 seconds compared to MPI-HMMER's best per-sequence time of 6.4 seconds, and translates to a performance improve-

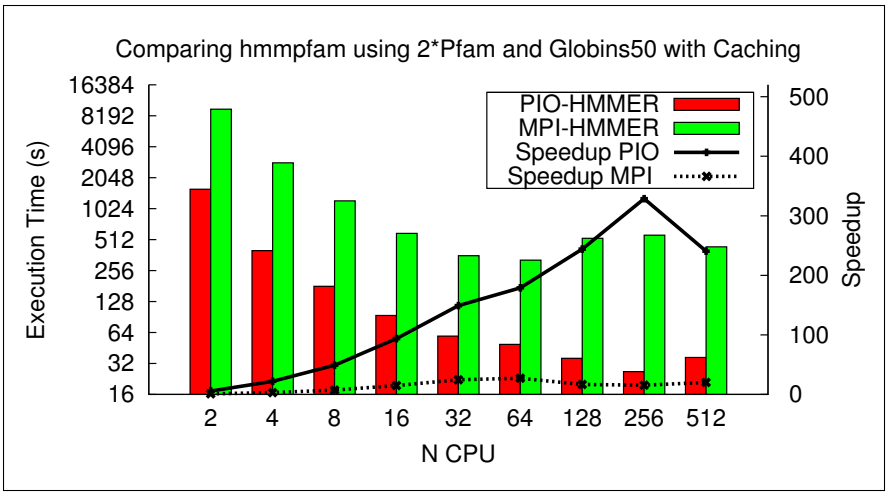

Figure 13. *hmmpfam* **implementation using database caching. Because the same HMMs are searched for each sequence query, we cache all HMM locally in order to further reduce file system I/O and communication. We achieve a maximum speedup of 328x.**

ment of 83x, where MPI-HMMER's *hmmpfam* is limited to 27x.

In Figure 13 we improve on the per-sequence *hmmpfam* times by utilizing database caching for subsequent sequence iterations. Rather than discarding the HMMs after their use, each node caches the assigned HMMs in memory for subsequent reuse. As a result we are able to improve the overall speedup to 328x at 256 nodes. This results in a per-sequence search time of only 0.53 seconds. We expect that implementing double buffering and load balancing in *hmmpfam* would further improve the search time for the first non-cached sequence.

5.3 Future Improvements

As we have shown, HMMER can benefit greatly from the use of parallel I/O. Because our cluster is a production system, we were unable to test multiple parallel file systems in order to determine the file system's impact on performance. However, we believe that better performance could be achieved by using a parallel file system that is geared more towards scientific computing. The Ibrix file system, for example, does not support striping a single file over multiple I/O nodes or segment servers. Instead the segment servers are designed to balance many files over the file system. This results in improved cluster performance overall, but is less useful for database-driven applications such as ours. Traditional parallel file systems such as PVFS/PVFSv2 would likely prove better suited to PIO-HMMER.

We also hope to further improve the load balancing strategy to allow for a more dynamic run-time-balanced computation. Currently, we divide the database proportionally among the worker nodes by sequence length. While we were able to achieve excellent improvement over the more naive (static allocation) approach discussed in Section 4 (and shown in Figure 10) we believe that further improvements are possible. Further, various system hardware (interconnects, CPU clock speeds, etc.) will impact load balancing as well as the size of database fragments that should be processed by each node. We plan to revisit the load balancing issue in the future.

We are also in the process of integrating a hierarchical approach similar to that described by Jiang et al. [9]. Not only would this help to alleviate the single-master bottleneck, it would also allow for the easy integration of heterogeneous accelerators (such as GPUs, FPGAs, and Cell processors). In our previous work we have shown that multiple FPGAs can work in conjunction with general-purpose processors to accelerate MPI-HMMER searches [16]. With a hierarchical model, accelerators could be arranged into homogeneous groups (e.g. one group of FPGAs and another group of GPUs). We hope to then extend these improvements to the *hmmpfam* tool in order to further reduce the compute time of non-cached searches.

6 Conclusions

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 hard-

ware. We have demonstrated this performance on a commodity cluster, using an inexpensive network (gigabit ethernet) and the Ibrix parallel file system. Similar hardware is commonly accessible worldwide, and as we have demonstrated it can be used to achieve 221x speedup for *hmmsearch* and 328x speedup for *hmmpfam*. This is more than 3x the peak performance of the publicly available MPI-HMMER. We hope to make the PIO-HMMER implementation available to the public shortly.

Acknowledgments

We would like to gratefully acknowledge Muzammil Hussain for various comments and discussions regarding this project and its implementation. We also acknowledge the anonymous reviewers for their helpful suggestions in improving this manuscript.

References

[1] University at Buffalo, The Center for Computational Research. `http://www.ccr.buffalo.edu`, 2009.

[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*. IEEE Computer Society, 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`, 2009.

[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*. IEEE Computer Society, 2005.

[8] J. P. Walters. MPI-HMMER. `http://www.mpihmmer.org/`, 2009.

[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] NCBI. The NR (non-redundant) database. `ftp://ftp.ncbi.nih.gov/blast/db/FASTA/nr.gz`, 2009.

[12] 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.

[13] Pfam. The PFAM HMM library: a large collection of multiple sequence alignments and hidden markov models covering many common protein families. `http://pfam.sanger.ac.uk/`, 2009.

[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. ACM, 2007.

[15] TimeLogic BioComputing Solutions. DecypherHMM. `http://www.timelogic.com/`, 2009.

[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. IEEE Computer Society, 2006.

[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*. IEEE Computer Society, 2005.