Autotuned Parallel I/O for Highly Scalable Biosequence Analysis

Haihang You
National Institute for Computational Science
P.O. Box 2008
Oak Ridge, TN 37831
hyou@utk.edu

Bhanu Rekapalli
National Institute for Computational Science
P.O. Box 2008
Oak Ridge, TN 37831
bhanu@utk.edu

Qing Liu
National Institute for Computational Science
P.O. Box 2008
Oak Ridge, TN 37831
qliu5@utk.edu

Shirley Moore
Electrical Engineering and Computer Science Department
University of Tennessee,
Knoxville, TN 37996
shirley@eecs.utk.edu

ABSTRACT

In recent years, the rate of genomics sequence generation increased dramatically due to significant advances in the sequencing technology. The genomics data is accumulating at an exponential rate in various databases all around the world and rapid analysis techniques will enhance the knowledge discovery in the fields of medicine and biotechnology. Analysis of such growing sequence databases demands tremendous computational power that can only be provided by massively parallel computers. Improving the performance and scalability of bioinformatics tools thus becomes a critical step in the quest to transform ever-growing raw genomics data into biological knowledge. In this paper we describe an efficient parallel implementation of a profile hidden Markov models (profile HMMs) code used for protein domain identification, along with auto-tuned parallel I/O optimization. Experimental results show linear speedup with increasing numbers of computing cores on a supercomputer, allowing the domain identification of millions of proteins in few minutes using hundreds of thousands computing cores.

Categories and Subject Descriptors

J.3 [**Computer Applications**]: Life and Medical Sciences; B.4.3 [**Input/Output and Data Communications**]: Interconnections—*Parallel I/O*; D.1.3 [**Programming Techniques**]: Concurrent Programming —*Distributed programming, Parallel programming*

TeraGrid '11, July 18-21, 2011, Salt Lake City, Utah, USA.

General Terms

Performance

Keywords

parallel I/O; massively parallel; auto-tuning; bioinformatic; hidden Markov models

1. INTRODUCTION

The exponential growth of genomics data in public databases [1] is not only with whole genome sequencing but also due to metagenomics [11] sequence generation. The proteins generated by metagenomics sequencing are far greater in numbers than non-redundant protein databases. Improving the performance and scalability of sequence analysis tools thus becomes a critical step in the quest to transform ever-growing raw genomics data into useful biological knowledge. One such bioinformatics tool is HMMER [7] package which is an open-source implementation of probabilistic models such as profile Hidden Markov Models (HMMs) for protein domain identification using PFam (Protein Families) models [8] along with sequence homology searches and protein family classification. It is known for its algorithmic enhancements for reducing computation time.

HMMER2 is a computationally intensive search algorithm which has been ported onto advanced architectures such as computer clusters, shared memory architectures, and also hardware accelerators such as GPUs (Graphics Processing Unit) and FPGAs (Field Programmable Gate Arrays). There are various implementations of HMMER with different optimization and parallelization techniques applied, such as JackHMMER [22], ClawHMMER [10], MPI-HMMER-Boost [19], MPI-HMMER [20] and SledgeHMMER [3]. The FPGA accelerated HMMER can achieve 100-fold speedup over software implementation running on single processor desktop [13]. However, most of the accelerated approaches are primarily focused on decreasing the computation time of single protein sequence analysis with varying percentage of sensitivity of results. HSP-HMMER [16] tried to address the

problem of identifying protein domains on very-large scale. It used HMMER2 algorithm and load balancing techniques to scale HMMER to thousands of cores on Cray XT4 architecture. The latest version of HMMER3 is 100 times faster than HMMER2. Despite performance improvement of HMMER3 algorithm, HMMER tools are still unable to keep up with the exponential growth of biological databases. There are now more than 100 million protein sequences in all public databases around the world, and this number is growing rapidly. Thus, to identify protein domains in all currently available protein sequences is a challenging computational task.

To the best of our knowledge, we have not seen any implementation of protein sequence analysis that can successfully scale up to hundreds of thousands of processors. According to the current ranking of the world's fastest supercomputers [14], Kraken, located at Oak Ridge National Laboratory, is one of the world's fastest machine. It provides tremendous computing power that could speed up the sequence analysis process and potentially solve the problem of very-large scale genomics data analysis.

In this paper we describe an efficient parallel implementation of HMMER3 along with auto-tuned parallel I/O optimization. Experimental results show linear speedup with increasing numbers of computing cores on a supercomputer. We were able to identify domain models for 100 million proteins in less than six minutes by running a capability job with 98K cores on Kraken. This paper is organized as follows. In Section 2, we briefly introduce HMMER, our modified HMMER parallel algorithm, and the scalability issues faced by our approach. Section 3 describes the optimizations that are suitable for improving I/O performance and load balancing. Experimental results are presented in Section 4. In Section 5, we describe our I/O auto-tuning framework in more detail. Finally, conclusions are provided in Section 6.

2. PARALLEL MODIFIED HMMER ALGORITHM

HMMER is used for searching sequence databases for homologs of protein sequences, and for making protein sequence alignments along with protein domain identification. It implements methods using probabilistic models called profile hidden Markov models (HMMs)[6]. The *hmmscan* tool of the HMMER package is used to search a query protein sequence against the PFam (protein family) database, which is one of the largest public collection of conserved protein domains [8].

We modified the *hmmscan* code of the HMMER package using MPI (Message Passing Interface) [18] to distribute individual serial *hmmscan* jobs to each of the many computer cores participating in the parallel run. Each *hmmscan* job gets access to its own individual input sequence and there is no communication between any of the many participating computer nodes. This scenario is often referred to as ideally parallel. The advantage is not only in simplifying programming issues, but also in avoiding the overhead associated with sending and receiving messages. The latter problem is likely a significant factor contributing to the low performance observed with the MPI-HMMER, especially when the number of computer cores is sufficiently high (more than 500). The performance of both MPI-HMMER and our approach are affected by the intense disk I/O. Input data has to be read for Pfam database and each protein sequence, and the resulting analysis has to be written to separate output files. When using 1000 or more compute cores, this can create very intense I/O traffic.

Nowadays multi-core CPUs are in the mainstream. On the Cray XT5, each compute node contains two sockets, each with a hex-core AMD Istanbul CPU. First, we chose a hybrid programming model combining MPI with Pthreads. One MPI process runs on each compute node and creates twelve threads which do the *hmmscan* job while only one thread does I/O. In this manner, we can decrease the number of readers and writers by twelve times compared to the original MPI-HMMER which would have one MPI process per core. A side benefit of this approach is to balance the workload to some extent, since each sequence has a different length and its processing time is proportional to the length. While one thread is processing a long length sequence, other threads can grab remaining sequences to work on. We have been able to mitigate the I/O problem partially by rearranging the distribution of protein sequences that are handed over to each MPI process on each compute node. The main idea consists of distributing protein sequences of different lengths so that each job finishes (and writes results) and starts a new sequence (reads) at a different time, thus randomizing the I/O events as much as possible [16]. This minimizes simultaneous reads/writes and avoids major time delays due to traffic jams. Second, the size of the Pfam database is around one Gigabyte. Instead of having each MPI process read the database, which would create intense I/O traffic on the file system, we subset the MPI processes and split the global communicator into multiple sub-communicators. Each sub-communicator has one reader that reads in the database and broadcasts it inside the group. On a parallel computer, the fact is that network speed is faster than I/O speed. Third, using a similar principle, we concatenate the output files that each MPI process generates inside each sub-communicator and write the result into one file. This approach lowers the number of files generated and consequently improves I/O performance.

3. IMPLEMENTATION AND I/O OPTIMIZATION

We implement the multi-threaded parallel modified HMMER algorithm as described in Section 2. On each multi-core compute node, a multi-threaded HMMER process is created as shown in Figure 1. For this implementation, each process has an entire node (12 cores, 16GB RAM) at its disposal. Looking at a single node, there is one process that forks off twelve threads, so we have a total of twelve workers. The master reads the database into RAM while reading the query sequences and inputting them into a work queue. Each worker takes sequences, as they need them, from the queue and performs domain identification on them. Each process has its own set of inputs and outputs in the same file hierarchy used for the ideally parallel approach. Each MPI process reads in two files, a Pfam database file and a sequences file, and writes out a file with results. This approach decreases the input contention for reading in the database by twelve times, and also decreases the number of resultant files by twelve times. This implementation worked out much better for both I/O and load balancing problems than the ideally parallel approach had. This helped the I/O

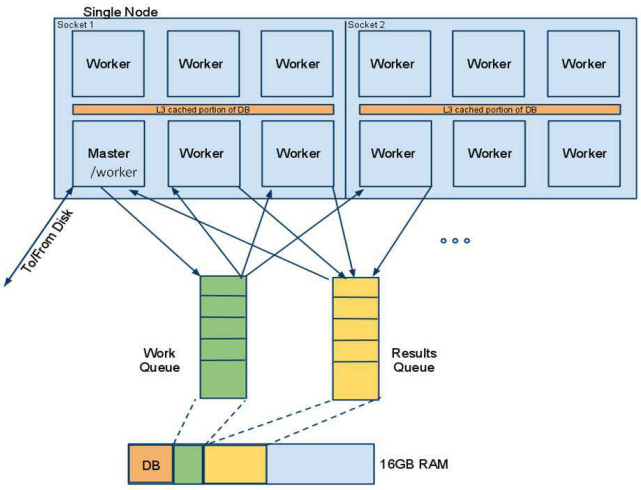


Figure 1: One HMMER process on a single compute node

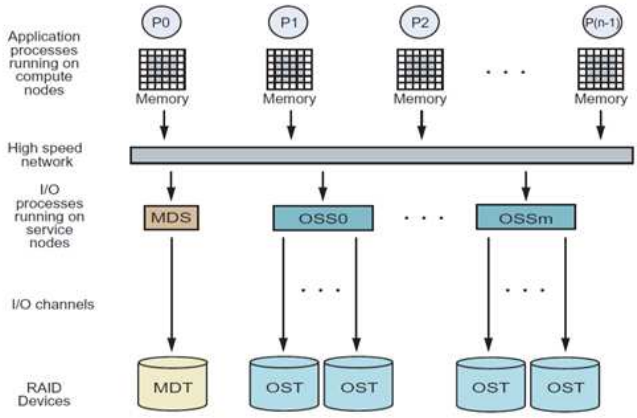


Figure 2: Lustre file system architecture

by limiting the number of concurrent reads and writes to disk by a factor of 12. It also helped with the load balancing problems by allowing cores to share the workload that each node has. For instance, if a core gets a sequence that takes exceptionally long to compute, the other cores can continue to pull sequences off the queue without having to wait on that one to complete. The cores within a node share the workload so that the workload is more evenly distributed. One extra benefit of this implementation is the caching behavior. Since the database is shared among all the cores in a node, the database can be cached and shared so that each core does not have to go to RAM for every database access. This is especially beneficial when each of the queries takes approximately the same amount of time to compute so that each core accesses the same portion of the database at the same time.

The previously described optimizations are still not enough when we try to scale the code to a full machine, Kraken, the Cray XT5 hosted at the National Institute for Computational Sciences, which has 99,072 compute cores, 8,256 compute nodes and a Lustre parallel file system. Because during a full machine run each MPI process on a compute node still generates I/O traffic which causes overwhelming contention not only on Lustre's metadata server, but also on storage servers. We will give a brief introduction of Lustre file system and our approach to optimize I/O within an auto-tuning framework in following sections.

3.1 Lustre Parallel File System

Nowadays, a parallel shared file system is a must have for a supercomputer. To utilize I/O effectively is essential for an application to scale up. Our current work is in the context of Lustre, but our ideas should be applicable to other distributed file systems. Lustre[4] is a distributed file system used for large scale cluster computing. It can support up to tens of thousands of client systems and serve petabytes of storage and hundreds of GBs per second of I/O throughput. As of June, 2010, 15 of the top 30 supercomputers in the world use the Lustre file system. A Lustre file system consists of two major units:

1. A single metadata target (MDT) per file system that stores metadata, such as filenames, directories, permissions, and file layout, on the metadata server (MDS)

2. One or more object storage servers (OSSes) that store file data on one or more object storage targets (OSTs). An OSS typically serves between two and eight targets, with each target being a local disk file system up to 8 terabytes (TBs) in size. The capacity of a Lustre file system is the sum of the capacities provided by the targets.

The architecture of a typical Lustre system is shown in Figure 2.

To access a file, a client has to complete a filename lookup on the MDS. Consequently, either a file is created if the file does not exist or information about the file is returned to the client. The information includes on which OSTs the file resides, and the offsets and sizes on each OST. The client then opens the file and does I/O operations directly to the OSTs. In this paper, we focus on the OSS and OST part and consider the MDS and MDT as the independent process. In other words, we will deal only with queuing and writing/reading.

3.2 Tuning Parameters of Lustre I/O

The I/O of the Lustre file system is very complicated. Here we list the parameters we are interested in that could affect the I/O performance:

1. Lustre stripe count

2. Lustre stripe size

3. I/O transfer size

4. Number of I/O processes

When an application does I/O on a Lustre file system, choosing different parameter values can affect the I/O performance dramatically. Sometimes users can see several magnitude difference in performance. For example, stripe size and stripe count(number of OSTs) are common parameters to tweak on a lustre file system.

3.3 HMMER I/O Optimization

To eliminate intense I/O contention, we divide the MPI global communicator into multiple sub-communicators, and the process with rank zero in each sub-communicator performs the I/O. It reads in the Pfam database and broadcasts it to other processes inside the sub-communicator. It also gathers output data from peer processes and aggregates the data into one file. The parameters mentioned in Section 3.2 will have a big impact on the overall performance. Empirical optimization techniques have been successfully applied to numerous software packages such as ATLAS and FFTW for achieving good performance. In our work, we applied similar techniques to acquire optimal values of parameters mentioned above.

3.3.1 I/O Auto-tuning

The I/O auto-tuning framework is discussed in section 5. To begin the auto-tuning process, we use an I/O benchmark IOR[12] to measure the I/O performance for I/O patterns of a single file written by a single process. For the single file and single process I/O case, we utilize the framework to search for optimal values for Lustre stripe count, and stripe size, transfer size per write. Given a set of initial set of parameters, IOR takes it as input, and it will report performance which will be used by the search engine to come up with the next set of parameters. Previous search results and sets of parameters are stored in a database, so that the search process can avoid redoing the test for an existing result. This procedure will stop when an optimal set of parameters is reached.

3.3.2 Search Space

Given the I/O pattern to be tuned, we can define the search space in different ways. See Table 1 for a summary of the search spaces. For sequential POSIX I/O case, we defined a search space with three parameters with lower and upper bounds: Lustre stripe count, stripe size, and transfer size. For parallel POSIX I/O case, we add an extra parameter: number of I/O processes.

Table 1: Summary of the Search Spaces

Code	Dimension	Bounds
IOR (sequential POSIX I/O)	Lustre stripe count	1 - 160
	Lustre stripe size	1M - 256M
	I/O transfer size	1M - 256M
IOR (parallel POSIX I/O)	Lustre stripe count	1 - 160
	Lustre stripe size	1M - 256M
	I/O transfer size	1M - 256M
	Number of I/O processes	1 - 12K

3.3.3 Search Techniques

Essentially, we are trying to solve an optimization problem of the function:

$$f(x_1, x_2, \cdots, x_n)$$

The parameters x_1 through x_n represent the tuning parameters, such as Lustre stripe count and size. Typically these are integer values, but in some cases could be real numbers. The value of the function is the performance of the I/O benchmark using that set of parameters. Performance can be evaluated in many ways, but the results presented in this paper are based on using IOR to measure I/O Megabytes per second. In [17], we have examined a variety of search heuristics such as Simplex Method, Genetic Algorithm, Simulated Annealing, Particle Swarm Optimization, Orthogonal and Random search method. Having effective search techniques will become increasingly important when empirical tuning become more sophisticated and the search spaces consequently grow.

3.3.4 Search Results

Search results are shown in Table 2. Figure 3 shows the I/O performance with searched parameters. The number of processes sharing one file decides the size of the subcommunicator in parallel HMMER. It is relatively easy to conduct sequential POSIX I/O tuning on Kraken, since the search task can be launched in interactive mode and there is no waiting time in the job queue for each benchmark test during the search process. But the parallel version can not avoid submitting benchmark test through the job queue, and it makes the search process take too much time to finish. And it generates too much I/O traffic on a production machine. So we obtained Lustre stripe count, stripe size and transfer size by running sequential POSIX I/O tuning. By conducting a few parallel HMMER experiments with different subcommunicator sizes, we chose it to be 1024. On Kraken, the default Lustre stripe count is 4 and stripe size is 1 MB. By default, transfer size the same as stripe size.

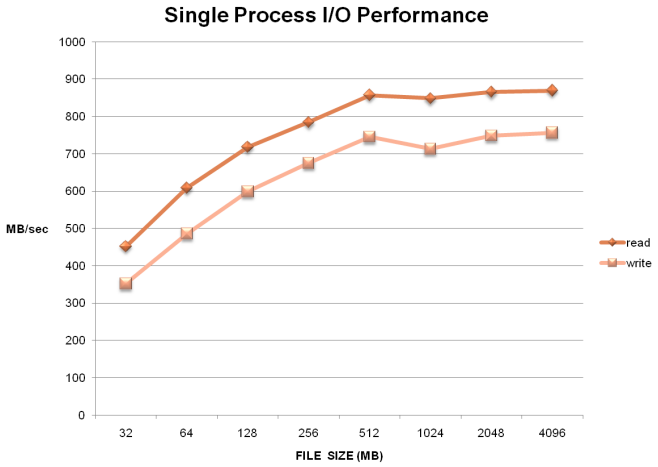


Figure 3: Single process IO performance on Kraken

3.4 Application Programming Interface

We developed an I/O API, ATLIO(Auto-Tuned Linux I/O), as an extension of the ANSI C file I/O API. It seamlessly replaces the POSIX I/O API with very few source code changes for HMMER versions that use POSIX I/O to read and write task-local files in parallel. For now, ATLIO supports the following two I/O functionalities:

1. Parallel write
2. Parallel read

3.4.1 Parallel Write

Sequential HMMER writes data into a file using multiple POSIX $fwrite()$ calls with small data sizes. Obviously this approach generates too much I/O contention when

Parameters	Optimal Value	Default Value
Lustre Stripe Size	32MB	1MB
Lustre Stripe Count	5	4
Transfer Size	32MB	1MB
Number of processes/shared file	1024	1

Table 2: Optimized and default parameters for Lustre on Kraken

tens of thousands of MPI processes write simultaneously. $ATLIO\_File\_Write()$ does buffering automatically on each MPI process and sends the data to the rank zero process of the subcommunicator when the buffer is full. Since HMMER output consists of a list of hits that do not need to be in any particular order, the data can be merged without ordering and thus, it is relatively easy for the implementation. The writer simply concatenates received data and writes to a file. Listing 1 is an example of writing to a file:

Listing 1: Parallel write

```
/* initialize the configuration data structure */
ATLIO_Init(&cfg);
/* open file */
ATLIO_File_Open(&cfg, comm, filename, "w", fh);
/* write to file */
ATLIO_File_Write(&cfg, fh, buf, count, datatype);
/* close file */
ATLIO_File_Close(&cfg, fh);
/* clean up */
ATLIO_Finalize(&cfg);
```

$ATLIO\_Init()$ initializes the structure with parameters generated from the auto-tuning process. It sets the Lustre stripe count, stripe size, transfer size, and buffer size, the size of the subcommunicator,etc. $ATLIO\_Finalize()$ cleans up the allocated data structure.

3.4.2 Parallel Read

Reading a file is identical to Listing 1, except with a call to $ATLIO\_File\_Read$ instead of to $ATLIO\_File\_Write$. The reader opens and reads data into a buffer, and broadcasts to peer processes. For parallel HMMER, it is used to read the Pfam database which is shared across every process. The parameter of number of readers is determined by the I/O throughput and network speed is set at initialization time. Synchronization has to be taken care of during the reading between peer processes.

4. EXPERIMENTAL RESULTS

In this section, we briefly describe the system used for the experiments and explain the setup of the experiments for the HMMER full machine capability run. We present results to show scalability of our parallel HMMER implementation and the effectiveness of the parallel I/O technique we provide.

4.1 Kraken

Kraken is a Cray XT5 supercomputer hosted by the National Institute for Computational Sciences (NICS) located at the Oak Ridge National Laboratory in the US[15]. Kraken has a total number of 8,256 compute nodes. Each node has two sockets, and each socket has a 2.6 Ghz hex-core AMD Opteron processor. The total number of cores is 99,072 with a peak performance of 1.03 PetaFLOPS. Each compute node has 16GB memory and thus the total compute memory 129TB. Kraken is attached to a Lustre parallel file system.

4.2 Experiment Setup

Protein sequences vary significantly in length. The computation time of HMMER's *hmmscan* is dependent both on the query sequence length and the domain database size. All the nodes in our approach use Pfam24 database. Thus the computation time is proportional to the sequence length, with longer sequences taking more time to finish than shorter sequences. We divide the query sequence file into N input files of approximately equal amino acids(AA) count, with each file about the same size, so that all the nodes finish the computation at about the same time. First the total number of amino acids(AA) count A in the query file is calculated. The AA count for each job will be A/N, denoted as a. Then all the sequences in the query file are sorted in descending order based on sequence lengths. Next we traverse through the sorted list allocating a sequence to a file, working from input file 0 to $N-1$. Once the first N sequences are allocated to these N input files, we restart the allocation, working from input file $N-1$ to 0 in reverse order. This allocation process is done in a round robin fashion until all the sequences are allocated to these N input files. The smallest sequences with length less than 50 AA are allocated at the end, ensuring almost the same number of AA count per each file [16].

4.3 Experiment Results

There are three sets of experiments:

1. Non-Optimized, ideally parallel HMMER with default Lustre settings shown in Table 2.
2. Multi-Threading, running one MPI process and multiple threads on each compute node with default Lustre settings.
3. Optimized I/O, multi-threading parallel HMMER with optimized I/O and pre-selected optimal Lustre settings shown in Table 2.

Figure 4 shows the performance comparison in terms of execution time of three versions of parallel HMMER running on Kraken. We can see that non-optimized version won't scale. Even though the ideally parallel approach is easy to implement, the performance is bad due to severe I/O contention between each process. The multi-threading version performed better than the non-optimized version because it eliminates I/O contention by twelve times. But it does not solve the I/O contention problem and we can see it takes more time to finish when the job size increases. The optimized I/O version shows the best performance. It combines multi-threading parallel HMMER with optimized I/O

and auto-tuned Lustre parameters. Table 3 shows experimental results of the optimized I/O version using from 1008 up to 96000 compute cores. We measured total execution time and total number of queries that are processed. HmmerSearch uses a profile hidden Markov model, which represents a group of aligned sequences, as a query to search a sequence database for related sequences. The alignments of the profile HMM to the best-scoring sequences are displayed in the output. We also give the average time for processing a single query. As shown in Figure 5(a) and Figure 5(b), our implementation of parallel HMMER can achieve linear speedup as we use more compute cores up to a full machine run, and our approach is perfectly scalable on a massive parallel supercomputer like Kraken.

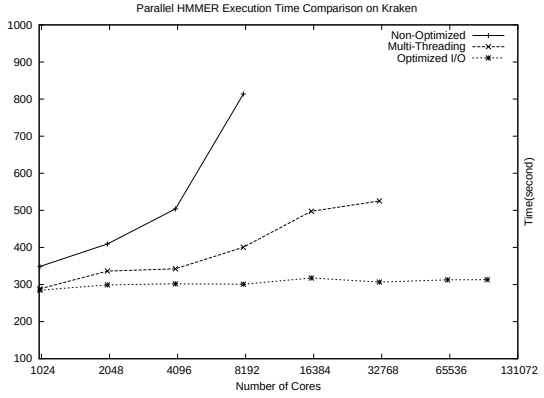
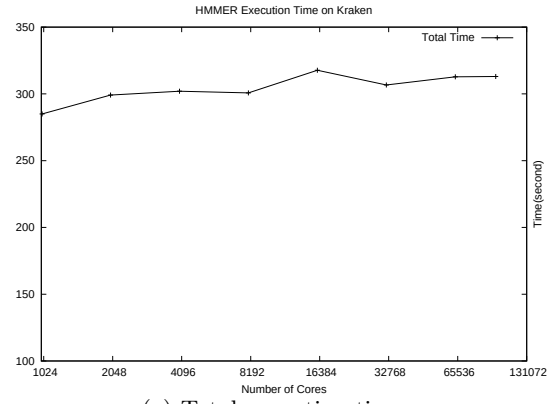
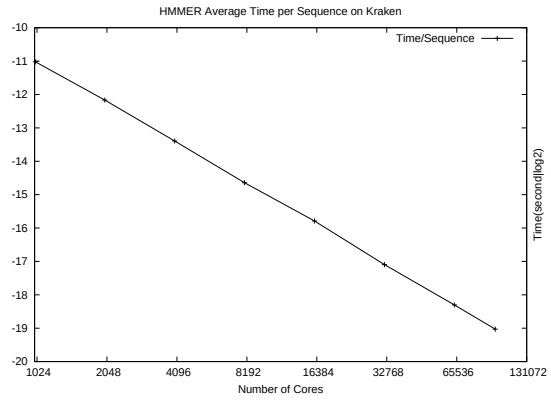


Figure 4: Parallel HMMER execution time comparison on Kraken



(a) Total execution time



(b) Average execution time per sequence

Figure 5: Parallel HMMER with I/O optimization on Kraken

5. I/O AUTO-TUNING FRAMEWORK

In this section, we describe our I/O auto-tuning framework, which we used in Section 3. More details can be found in [23]. Figure 6 shows the structure of the auto-tuning process. In the first step to calibrate the simulation on a specific system, we run a benchmark such as IOR[12] and compare with the simulation results. By adjusting the setup parameters of the simulation and the mathematical models, the simulation will behave in the same way as the real system. We call this step the training process. The goal is to figure out key parameters for the mathematical model and, based upon that auto-tuning can be done by simulation (not on real system). Then for a given I/O setup, we use the mathematical model to generate a set of parameters that serve as the starting point of the tuning process. Each set of parameters is input to the simulation, which reports performance that is used by the search engine [17] to come up with the next set of parameters. Previous search results and sets of parameters are saved into a database, so that if search process finds the parameters set has already be done, it skips to the next set of parameters This procedure stops when an optimal set of parameters is reached.

The advantages of our approach are as follows:

- Simulation will release the burden on the real system since it will not generate real I/O traffic.
- The auto-tuning framework is faster compared to running the search process on the real system, since each benchmark job has to go through the batch queuing systems, and also it takes more time to finish each job.
- The noise of the real system will be avoided, and the framework can provide parameters without much error.

To summarize, an application user who wants to use this tool to auto-tune I/O should follow the steps below

- Use IOR to do parameter training for a particular system.
- Feed the parameters that are generated in the previous step to the mathematical model and start auto-tuning process. This should be significantly faster than conventional auto-tuning, which occurs on a real system.
- Adopt the optimized parameters into the application code.

6. CONCLUSION

HMMER is one of the essential computational tools widely used by biologists to construct profile hidden Markov models for the detection of protein sequence similarity, protein family classification, and functional annotation. To the best of our knowledge, it has not been able to scale up to hundreds of thousands compute cores, limiting its capability to

Cores	Queries	Total Time (s)	avg time/query
1008	1006809	284.97	2.8304E-04
2004	2001699	299.11	1.4943E-04
4008	4003352	302.02	7.5442E-05
8004	7994733	300.77	3.7621E-05
16008	15989466	317.71	1.9870E-05
32004	31966977	306.67	9.5933E-06
64008	63933972	312.81	4.8927E-06
96000	95888993	313.04	3.2646E-06

Table 3: Parallel HMMER with I/O optimization experimental results on Kraken

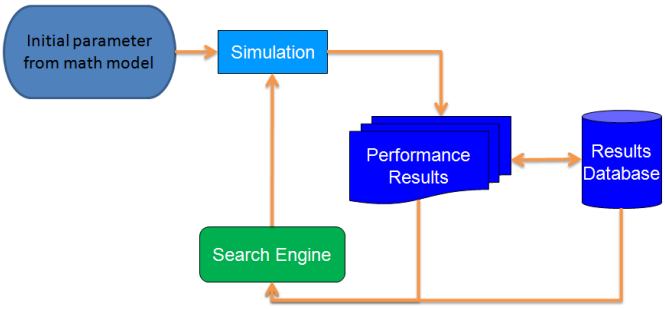


Figure 6: I/O Auto-tuning Framework

satisfy the needs of the advance of computational biology research on very large scale. In this paper we demonstrate the effectiveness of our parallel implementation of the HMMER *hmmscan* tool along with our I/O auto-tuning strategy. We also describe an auto-tuned I/O library for such applications. Empirical optimization has been shown to be an effective technique for optimizing code for a particular platform such as ATLAS [21, 5], PHiPAC [2], and FFTW [9]. Our research demonstrates that it also can be effective for I/O tuning. We plan to build an auto-tuning I/O framework based on mathematical modeling and simulation that will generate an I/O library with optimal parameters for an underlying architecture and file system.

Acknowledgment

This research used resources at the National Institute for Computational Sciences supported by the National Science Foundation. This research was also supported in part by the National Science Foundation under grant EPS-0919436.

7. REFERENCES

[1] D. A. Benson, M. S. Boguski, D. J. Lipman, J. Ostell, and B. F. Francis. Ouellette genbank. In *Nucl. Acids Res.*, volume 26, pages 1–7, 1998.

[2] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *International Conference on Supercomputing*, pages 340–347, 1997.

[3] G. Chukkapalli, C. Guda, and S. Subramaniam. Sledgehmmer: a web server for batch searching the pfam database. In *Nucl. Acids Res.*, volume 32, 2004.

[4] Cluster File Systems, Inc. Lustre: A scalable, highperformance file system. Technical report, White paper (2002).

[5] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[6] S. R. Eddy. HMMER3: a new generation of sequence homology search software . `http://hmmer.janelia.org`.

[7] S. R. Eddy. Profile hidden Markov models. *Bioinformatics*, 14:755–763, 1998.

[8] R. Finn, J. Mistry, J. Tate, P. Coggill, A. Heger, J. Pollington, O. Gavin, P. Gunesekaran, G. Ceric, K. Forslund, L. Holm, E. Sonnhammer, S. Eddy, and A. Bateman. The pfam protein families database. *Nucleic Acids Research Database Issue*, (38):D211–222, 2010.

[9] M. Frigo and S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[10] D. R. Horn, M. Houston, and P. Hanrahan. Clawhmmer: A streaming hmmer-search implementatio. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 11, Washington, DC, USA, 2005. IEEE Computer Society.

[11] P. Hugenholtz and G. W. Tyson. Microbiology: Metagenomics. In *Nature*, volume 455, pages 481–483, 2008.

[12] IOR. The ASCI I/O stress benchmark. `https://computing.llnl.gov/?set=code\&page=sio\_downloads`.

[13] R. P. Maddimsetty, J. Buhler, R. D. Chamberlain, M. A. Franklin, and B. Harris. Accelerator design for protein sequence hmm search. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 288–296, New York, NY, USA, 2006. ACM.

[14] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. TOP500 Supercomputing Sites. `http://top500.org`.

[15] National Institute for Computational Sciences. Kraken. `http://www.nics.tennessee.edu/computing-resources/kraken`.

[16] B. Rekapalli, C. Halloy, and I. B. Zhulin. Hsp-hmmer: a tool for protein domain identification on a large scale. In *Proceedings of the 2009 ACM symposium on*

Applied Computing, SAC '09, pages 766–770, New York, NY, USA, 2009. ACM.

[17] K. Seymour, H. You, and J. Dongarra. A comparison of search heuristics for empirical code optimization. In *3rd International Workshop on Automatic Performance Tuning*, 2008.

[18] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[19] J. Walters, X. Meng, V. Chaudhary, T. Oliver, L. Yeow, B. Schmidt, L. Nathan, and J. Landman. Mpi-hmmer-boost: Distributed fpga acceleration. *Journal of VLSI signal processing*, 2007.

[20] J. P. Walters, R. Darole, and V. Chaudhary. Improving mpi-hmmer's scalability with parallel i/o. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[21] R. C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, January 2001.

[22] 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 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 173–184, Washington, DC, USA, 2005. IEEE Computer Society.

[23] H. You, Q. Liu, Z. Li, and S. Moore. The design of an auto-tuning I/O framework on Cray XT5 system. In *Cray User Group meeting (CUG 2011)*, Fairbanks, Alaska, May 2011.