The Design of an Auto-Tuning I/O Framework on Cray XT5 System

Haihang You*, Qing Liu‡, Zhiqiang Li† and Shirley Moore§
*National Institute for Computational Science
Oak Ridge, TN 37831
hyou@utk.edu
‡Mathematics Department
University of Tennessee, Knoxville, TN 37996
zli@math.utk.edu
†Oak Ridge National Laboratory
Oak Ridge, TN 37831
liuq@ornl.gov
§Electrical Engineering and Computer Science Department
University of Tennessee, Knoxville, TN 37996
shirley@eecs.utk.edu

***Abstract*—As high performance computing (HPC) heads towards the exascale era, the computing power surges tremendously and applications will scale to hundreds of thousands cores. Consequently, the amount of data processed and generated will increase dramatically. 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. We have developed a mathematical model to describe parallel I/O activities that serves as the basis for an I/O auto-tuning infrastructure for HPC systems. Our current work is in the context of Lustre, but our ideas should be applicable to other distributed file systems. This paper explains our model, which is based on queuing theory, describes the auto-tuning process, and gives experimental results over Lustre on the Cray XT5 that show low relative error.**

***Keywords*-I/O; performance modeling; auto-tuning; queuing theory; simulation**

I. Introduction

As HPC heads toward the exascale era, the high performance computing power surges tremendously and this trend will continue in years to come. Applications can scale to hundreds of thousands cores; consequently, the amount of data processed and generated will increase dramatically. Nowadays, a parallel shared file system is a must have for a supercomputer. Utilizing I/O effectively is essential for an application to scale up. We would like to study the I/O characteristics of applications, design mathematical models to describe I/O activities, and build an I/O auto-tuning infrastructure for HPC systems.

Our current work is in the context of Lustre, but our ideas should be applicable to other distributed file systems. Lustre[1] 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 1.

To access a file, a client has to complete a filename lookup on the MDS. Subsequently, either a file is created if the file does not exist or information about the file is returned to

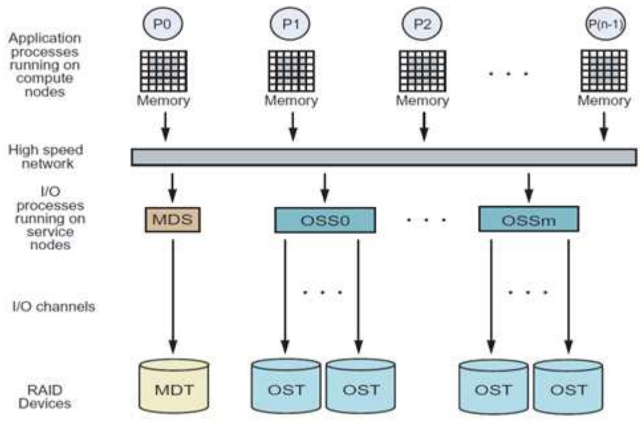


Figure 1. Lustre file system architecture

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 the paper, we focus on the OSS and OST part and consider the MDS and MDT part as the independent process. In other words, we only consider queuing and writing/reading.

The I/O of the Lustre file system is very complicated. The following parameters can affect the I/O performance:

1) Lustre stripe count
2) Lustre stripe size
3) I/O transfer size
4) Number of processes per file
5) Number of files
6) Total 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 drastically. Sometimes users can see several orders of magnitude difference in performance. For example, stripe size and stripe count (number of OSTs) are common parameters to tweak on a Lustre file system. Thus, an auto-tuning process seems desirable. But in practice, an auto-tuning process may add too much load to the file system and affect other users. In addition, other users' processes may contribute noise to the tuning process and disturb its accuracy. To avoid these effects, we model the entire system mathematically and simulate the system. If result from the model and the simulation match, we can start the auto-tuning process on the computer based on the modeling and simulation results. After the tuning process, we apply the optimal parameters we found to the real processes.

The paper is arranged as follows: First, we introduce mathematical models for two different cases. Second, we give results of running the simulations and provide comparisons between the mathematical modeling and simulation results. Then we give results from running the auto-tuning tests on three different systems: Kraken (Cray XT5, Jaguar (Cray XT4), and Jaguarpf (Cray XT5). We use the Kraken test results to tweak the parameters. Finally, we propose a more interesting model for future work.

II. Mathematical Model

We set up our model with information we gathered for Kraken, the Cray XT5 hosted at the National Institute for Computational Sciences. The specifications of its file system are shown in Table II.

The total time or the system time which is spend to write a job contains two parts: waiting time in the queue and the service time (or writing time) at the disk. In this section we build the mathematical models to find the total time in average. The goal is given by the following equation:

$$E(T) = E(W) + E(S) \tag{1}$$

Name	Value	Source
Seek time	0.004s	Kraken webpage
Write time	0.5Gbit/s	Kraken webpage
Inter-arrival time	13.47804s (in average)	June 2010 usage logs
Service time	4.453082s (in average)	June 2010 usage logs
Transfer speed	10 Gbit/s	Kraken webpage

Table I
Specification of Kraken's file system

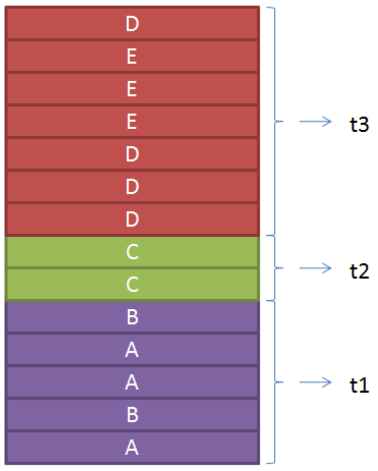


Figure 2. Inter-arrival times between I/O requests

where T is the total time, W is the waiting time and S is the service time.

We make the following assumptions for the whole section:

1) The seek time is a constant. From the Table I, the seek time is 0.004s for the Kraken file system.
2) The inter-arrival times are independently exponential distributed with parameter λ.
3) The service times are independented exponential distributed with parameter γ.
4) The service times are independent from the inter-arrival times.
5) There is no request at the moment when the system starts to work.

The inter-arrival time is defined as the difference of arrival times between the last requests of two jobs. Since the system time of a job depends on the last block, we treat the entire job as one block a rriving at the moment when the last block arrives. The service time is the time which the system spend to finish a job. If two jobs arrive at the same time, we treat them as one job arriving at that moment from the system point of view. Figure 2 explains this idea. Jobs A and B both arrive at time t_1. Job C arrives at time t_2. Then $t_2 - t_1$ is the inter-arrival time. Similarly, $t_3 - t_2$ is the inter-arrival time between job C and job (D+E).

arrival time(s)	number of experiments	error
1000	1000	0.71%
	5000	0.91%
10000	1000	0.13%
	5000	0.06%
20000	1000	0.23%
	5000	0.05%

Table II
MODELING ERRORS

A. One File and One OST

First we start with the simplest case: one OSS and one OST. Since the seek time is too small comparing with the service time and waiting time, we ignor the seek time in this case. Then this can be medeled by classical queuing theories which are discussed in many references, e.g. [2], [3]. Suppose that a file with size M is to be writen at time t. Let $W(t)$ be the waiting time of the request. In this case, the service time is fixed since the file size is known. So the waiting time in the queue is the only random factor.Since the waiting time only depends on the current position in the queue, $W(t)$ is a Markov process.

The following equation for expected waiting time is given in [4]:

$$E(W(t)) = (\frac{\lambda}{\gamma} - 1)t + \int_0^t p_0(u)du \qquad (2)$$

where $p_0(t) = P(\text{system is idle at time t})$. To understand the above formula clearly, define a process $\bar{W}(t)$ in the same way as $W(t)$ except that there is no barrier at 0, that is, $\bar{W}(t)$ can take negative value. So the process $\bar{W}(t)$ decreases deterministically at unit rate for all t. The first two terms on the right side of the above equation give the expected value of $\bar{W}(t)$. The final term in the above equation is the correction for the total time spent by $W(t)$ at 0, during which no such decrease occurs.

We have established the mathematical model. Next we use simulations to verify it. We run the simulation in Matlab and compare the results with those from the mathematical model. Table II shows the errors.

Moreover, it has been proven that the Markov process $W(t)$ is stable as time tends to infinity, if $\rho = \frac{\lambda}{\gamma} < 1$. In other words, there is an invariant probability distribution for the limit of $W(t)$. From the table, when the number of experiments are 5000, the error at time 10000 seconds is almost same as the one at time 20000 seconds. Our comparison shows the distribution tends to be stable.

B. One File and Multiple OST's

In the Lustre file system, I/O parallelism is the most important feature for parallel applications. So we need to consider a system with N OST's. We assume that these OST's are independent. Actually, the transfer times should be a part of inter-arrival time. But since the transfer time is very small compared with the exponential inter-arrivals, we simplify our model by ignoring the transfer times. For the same reason, we don't consider the seek time. Let the service time at the ith OST be $S_i, i = 1, 2, ...$ which is exponentially distributed with parameter γ. In this case, we have to consider two parts: waiting time and service time, since they both depend on the number of OST's. The waiting time for a file that is divided into $n (n \leq N)$ blocks is the largest waiting time on those n OST's, denoted by $W(t)$, where t is the moment when the file arrives to the queue. Intuitively, the more OST's that are used, the more time it takes to wait in the queue. But the service time can be reduced by using more OST's. Therefore, we have to balance these two parts and find optimal settings to minimize the system time.

To find the expected system time, we first consider the distribution of $W(t)$. Let $F(w)$ be the distribution function of the waiting time $W(t)$. The expected system can be calculated by the following equation:

$$E(W(t)) = \int_0^\infty wF(dw).$$

Let $W_i(t), i = 1, 2, ..., n$ be the waiting time of the file that arrives at time t on the i-th OST. We assume that all the $W_i(t)$ are independent and identically distributed, that is, there is no relationship between two waiting times in the two different queues and they have the same distribution. Since the distibutions of $W_i(t)$ are the same we define $F_1(w)$ as the distribution function in the first queue. By the defination of the distribution function, we have

$$\begin{aligned} P(W(t) \leq w) &= P(\max_{0 \leq i \leq n} W_i(t) \leq w) \\ &= P(W_1(t) \leq w, W_2(t) \leq w, ..., W_n(t) \leq w) \\ &= P(W_1(t) \leq w)^n \end{aligned}$$

where the last equation follows from the idependence of waiting times.

To calculate the distribution of the waiting time at the first OST, we need to find the density function. Let $f(w, t)$ be the density of waiting time in first queue. The distribution can be given by

$$F_1(w) = \int_0^w f(u, t)du.$$

From [4], we have

$$\begin{aligned} \frac{\partial f(w,t)}{\partial t} &= \frac{\partial f(w,t)}{\partial w} - \lambda f(w,t) + \lambda f(0)\gamma e^{-\gamma w} \\ &\quad + \lambda \int_0^w f(w-u,t)\gamma e^{-\gamma u} du, \\ \frac{dp_0(t)}{dt} &= -\lambda p_0(t) + f(0,t), \end{aligned}$$

where $p_0(t) = P(W_1(t) = 0)$.

If we assume that $\rho = \frac{\lambda}{\gamma} < 1$, which means that the average inter-arrival time $(\frac{1}{\lambda})$ is longer than the average service time ($\frac{1}{\gamma}$), we may find the equilibrium solution by assuming that, as $t \to \infty, f(w,t) \to f(w)$. In other word, the density of the waiting time has a formula which doesn't depend on time after a long time period. It could be the case for Kraken, since it has run for a long time. According to the usage data from June 2010, we also find $\rho = \frac{\lambda}{\gamma} < 1$ for the system of Kraken. So letting $t \to \infty$, we have

$$\begin{aligned} \lambda f(w) - f'(w) &= \lambda f(0)\gamma e^{-\gamma w} + \lambda \int_0^w f(w-u)\gamma e^{-\gamma w} du, \\ \lambda p_0 &= f(0). \end{aligned}$$

Solving for $f(w)$ and p_0,

$$\begin{aligned} f(w) &= \rho\gamma(1-\rho)\exp\left(-\gamma(1-\rho)w\right), \\ p_0 &= 1-\rho \end{aligned}$$

Now we are ready to calculate the corresponding distribution function. Let $F_1(w)$ be the distribution function of the waiting time in the first queue. For $w > 0$, we have:

$$\begin{aligned} F_1(w) &= \int_0^w f(u)du \\ &= \int_{0-}^w \rho\gamma(1-\rho)\exp\left(-\gamma(1-\rho)w\right) + p_0 \\ &= \rho[1-\exp\left(-\gamma(1-\rho)w\right)] + (1-\rho) \\ &= 1-\rho\exp\left(-\gamma(1-\rho)w\right) \end{aligned}$$

Noting that $p_0 = 1-\rho$ is the probability that the system is idle at time 0, we have the distribution function of waiting time in one queue:

$$F_1(w) = \begin{cases} (1-\rho) & \text{if } w = 0 \\ 1-\rho\exp\left(-\gamma(1-\rho)w\right) & \text{if } w > 0 \end{cases} \tag{3}$$

Substituting (3) into (3), we have

$$F(w) = F_1(w)^n = \begin{cases} (1-\rho)^n & \text{if } w = 0, \\ (1-\rho\exp\left(-\gamma(1-\rho)w\right))^n & \text{if } w > 0 \end{cases} \tag{4}$$

Let the writing speed on the OST is V, then the service time of the file whose size is M is $\frac{M}{nV}$. The expected mean of the system time is given by

$$\begin{aligned} E(T) &= E(W) + S \\ &= \int_0^\infty w dP(w) + \frac{M}{nV} \\ &= \int_0^\infty wn[1-\rho\exp\left(-\gamma(1-\rho)w\right)]^{n-1} \\ &\quad \times\rho\gamma(1-\rho)\exp\left(-\gamma(1-\rho)w\right)dw + \frac{M}{nV} \\ &= \frac{1-(1-\rho)^n}{\gamma(1-\rho)}\sum_{i=1}^{n-1}\frac{1}{i} + \frac{\rho^n}{\gamma(1-\rho)n} + \frac{M}{Vn} \end{aligned}$$

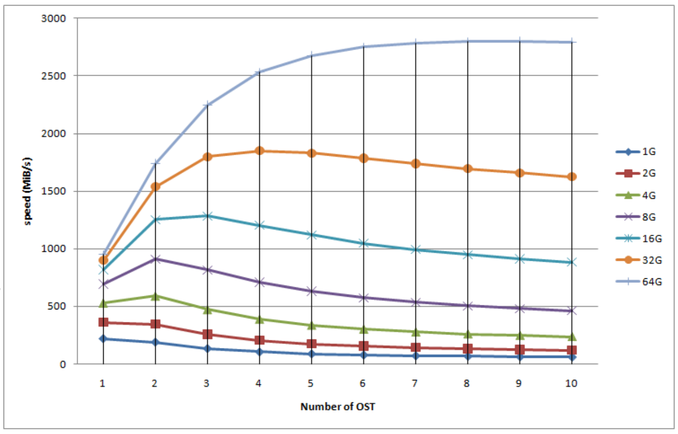


Figure 3. Speed vs number of OSTs

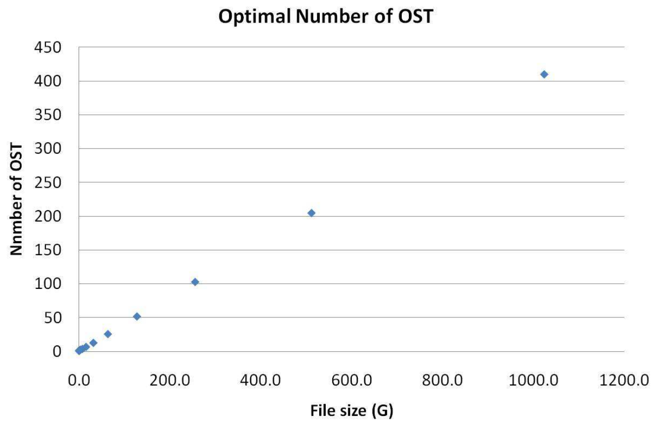


Figure 4. Effect of file size on optimal number of OSTs

From the above equation, we can see that the first term increases as n increases and the other two terms decrease as n increases. Although we don't know the explicit formula for the optimal number of OST's, we still can tell that it must be at somewhere in the middle. We use Matlab to find it for a file whose size is known.

Figure 3 shows the relationship between n and system time for writing a 50G file. The result matches experience with the real application: the more OST's that are used, the more time is spent on waiting and the less time is spent on writing. So there must be a optimal number in the middle to minimize the time that is spent to finish the job.

Figure 4 shows the relationship between the file size and the optimal number of OSTs. The optimal number of OST's increases as the file size becomes larger.

Then we simulate the system on the computer. Let T_s (resp. T_m) be the system time for the file from simulation (resp. math model), and let N_s (resp. N_m) be the optimal number of OST's from the simulation (resp. math model). The comparison between the mathematical model and the

Table III
COMPARISON FOR 1000 EXPERIMENTS

File size	T_s	T_m	N_s	N_m	relative error
1	49.1129	48	1	1	2.2%
10	114.5909	117.312	4	4	2.3%
50	176.3773	181.9326	24	20	3.1%
100	202.7398	210.1419	42	40	3.7%
500	263.1881	274.9212	179	199	4.5%
1000	292.9404	302.6972	425	399	3.3%

simulation results is shown in Table III:

From Table III, we can see that the relative errors do not change much as the file size increases. This gives us confidence for ignoring the transfer time and seeking time.

In real practice, it may not be easy to find the parameter λ. We test the system to find the distribution of the system time. Since we know the file sizes used in the tests, the service times are known. Then, we can find the distribution of the waiting time; that is, the parameter λ can be found by this approach.

III. TESTS ON LUSTRE FILE SYSTEMS

In this section, we give the results from three different Cray XT4/5 systems. We use the test data from Kraken to tweak the parameters. The reason we do this is that we found that different system's settings are very different. For example, Jaguar has cache on, but Jaguarpf has no cache. This caused a big difference when we ran the tests. Figure 5 shows the speed on the cache system and Figure 6 shows the speed on the no cache system. The x-axis represents stripe size and the y-axis represents transfer size. The legend on the write gives achieved I/O speed in MB/s.

From these figures, we can see that the test speed on the cached system is much faster. The reason is that the write or read time on the disk was not counted when the cache was turned off. So although we know the physical speed of the disk, we still cannot apply our model to the real system. We have to tweak the speed by comparing with the test results. Since we use Kraken, we tweak the speed using the test data from Kraken. Our approach is that we write one file on a single OST and compare the test results with the simulation we established in Section 2.1. The figure (7) is the average writing speed for a 1G file on Kraken. The x-axis represents stripe size and the y-axis represents transfer size. The legend on the write gives achieved I/O bandwidth in MB/s. Because we only test the results for the case in which the stripe size is larger or equal to transfer size, the data above the diagonal are all zeros. Figure 8 shows the results from the simulation.

We also use these data to tweak the parameter. After comparing the simulation and test results, we decide to use $V = 1G/s$ for Kraken. We calculate two different kinds of relative error. One is: error1 $= \frac{\text{sum of (test result}-\text{simulation)}}{\text{sum of test result}} = 0.07879$. The other one is: error2 $= \frac{\text{sum of (test result}-\text{simulation)}^2}{\text{sum of (test result)}^2} = 0.06364$.

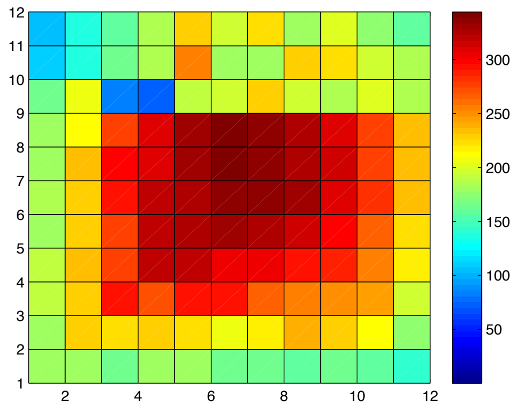


Figure 5. I/O speeds writing a file with cache

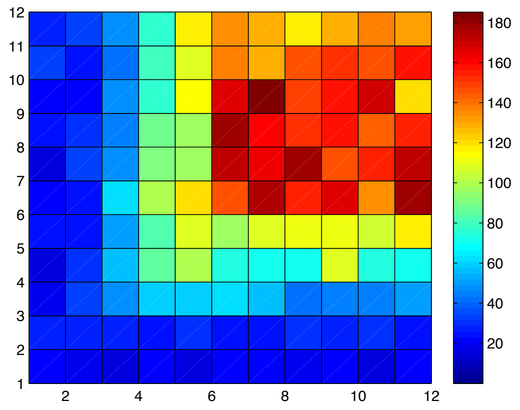


Figure 6. I/O speeds writing a file with no cache

After this preparation, we are ready to deal with writing one file on multiple OST's. To do that, we use all the parameters we found from Kraken and run the simulation on a single computer. Then we can build the framework for any one file writing. The same work can be done similarly for reading a file from multiple OST's.

IV. AUTO-TUNING FRAMEWORK

Figure 9 shows the structure of the auto-tuning process, adapted from [5]. In the first step to calibrate the simulation on a specific system, we run a benchmark such as IOR[6] and compare with the simulation results. We adjust the setup parameters of the simulation and mathematical models so that the simulation will behave in the same way as the real system. We call this step the training process. Then for a given I/O setup, we use the mathematical model to generate a set of parameters which is the starting point of tuning process. Each set of parameters is input to the simulation, which reports performance that is used by the search engine

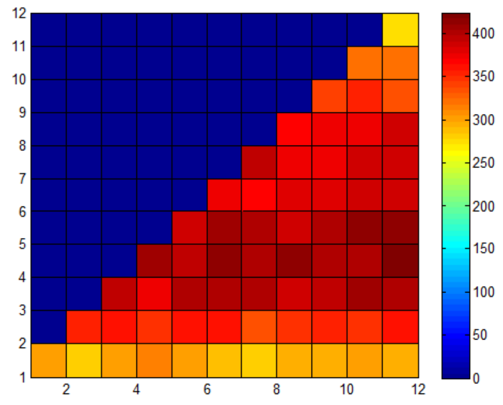


Figure 7. I/O speeds writing a file on Kraken

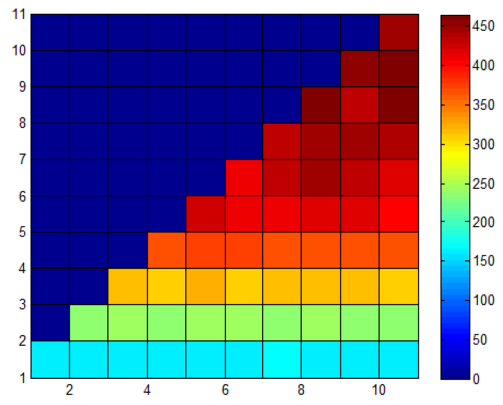


Figure 8. Simulated I/O speeds

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 stops when an optimal set of parameters is reached.

- Simulation will release the burden on the real system.
- The auto-tuning framework is faster compared to running the search process on the real system, since each benchmark job has to go through batch queue, 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.

V. Application Example

The HMMER package is an open-source implementation of profile Hidden Markov Model (HMM) methods for sensitive database searches [7]. It is one of the essential computational tools widely used by biologists to construct profile hidden Markov models for the detection of protein

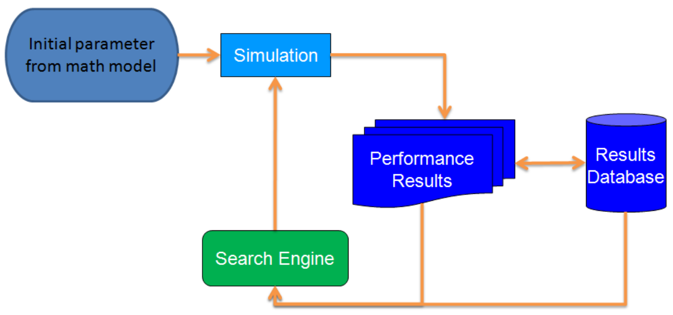


Figure 9. Auto-tuning

sequence similarity, protein family classification, and functional annotation. 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.

The latest version of HMMER3 [8] is 100 times faster than previous version of HMMER. 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 had not previously seen any implementation of protein sequence analysis that can successfully scale up to hundreds of thousands of processors. 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 [9]. We have implemented an efficient parallel version of $hmmscan$, along with an optimized I/O implementation using our parallel I/O auto-tuning framework. Experimental results show linear speedup with increasing numbers of computing cores on Kraken, a Cray XT5, allowing the analysis of 100 millions of proteins in less than six minutes by running a capability job with 98K computing cores. The details of the implementation will be published elsewhere. Here we summarize the I/O optimization part.

A. *HMMER I/O Optimization*

To eliminate intense I/O contention, we divided 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 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 Lustre parameters mentioned in section I have a big impact on the overall performance. We used the tuning process described in section IV to arrive at the Lustre parameters given in Table V. The number of processes sharing one file determines the size of the subcommunicator in parallel HMMER.

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.

1) Search Space: Given the I/O pattern to be tuned, we can define the search space in different ways. See Table IV 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 IV
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

2) 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 [5], 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) Search Results: Search results are shown in Table V. Figure 10 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.

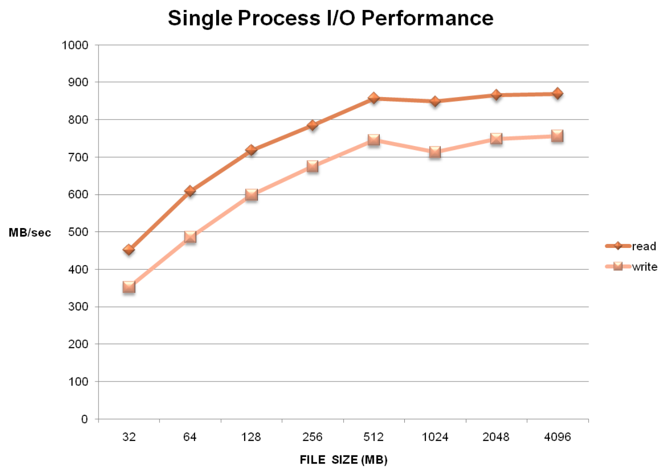


Figure 10. Single process IO performance on Kraken

B. Experimental Results

There are three sets of experiments:

1) Non-Optimized, ideally parallel HMMER with default Lustre settings shown in Table V.
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 V.

Figure 11 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 VI shows experimental results of the optimized I/O version using from 1008 up to 96000 compute cores. We measured total

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 V
OPTIMIZED AND DEFAULT PARAMETERS FOR LUSTRE ON KRAKEN

execution time and total number of sequences that are processed. And we also give the average time for processing a single sequence. As shown in Figure 12(a) and Figure 12(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.

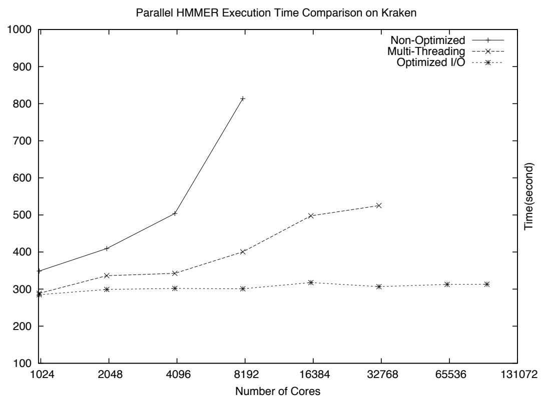


Figure 11. Parallel HMMER execution time comparison on Kraken

VI. RELATED WORK

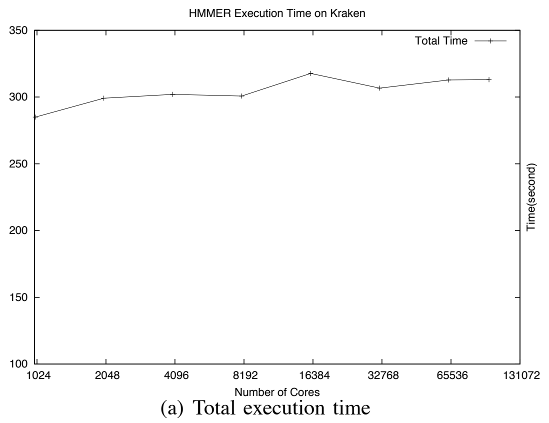
Queuing models have long been used to model disk array systems for performance analysis and prediction [10], [11].

Approximate queueing models for internal parallel processing by individual programs in a multiprogrammed system are developed in [12]. The solution technique is developed by network decomposition. The models are formulated in terms of CPU:I/O and I/O:I/O overlap and applied to the analysis of these problems. The authors include both CPUs and I/O devices in their model and include distributions for seek and transfer times which we ignore. There models can be solved exactly for a restricted class of systems but require inexact solution in general.

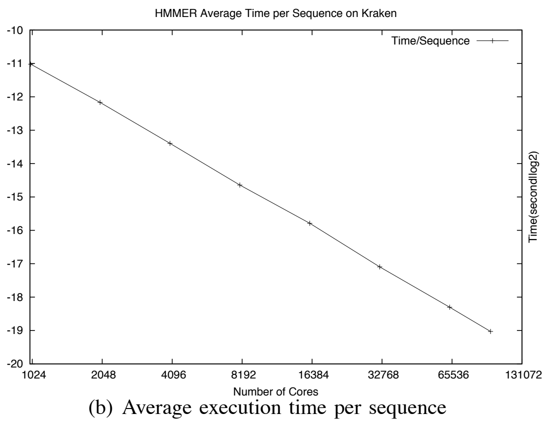
Our work differs from previous queuing models in that we focus on the specific problem of determine the optimal parameter settings for writing a file in a parallel file system. We also make simplifying assumptions so that we have a Markov process for which we can obtain an analytic expression.

Auto-tuning of parallel I/O has been explored in the Panda project [13], specifically focusing on collective I/O requests that read and write large arrays. Auto-tuning parameters include array disk layout and disk unit size (i.e., buffer size). Either a rule-based system or an empirical search (the most successful type of search was found to be adaptive simulated annealing) could be used to try to determine optimal settings of the parameters. This Panda work pre-dated MPI I/O. The Panda I/O optimization was designed to work on systems with low variance in I/O performance.

Characterization, tuning, and optimization of parallel I/O on the ORNL Cray XT4 Jaguar computer is presented in [14]. For I/O tuning, the authors experimented with different settings of tuning parameters for use of an extended two-phase collective I/O protocol with three I/O benchmarks. Tuning parameters included the buffer size of the I/O aggregation and the number of I/O aggregator processes. Although they did not use a formal auto-tuning framework, they ran



(a) Total execution time



(b) Average execution time per sequence

Figure 12. Parallel HMMER with I/O optimization on Kraken

Cores	Query	Total Time	avgtime/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 VI
PARALLEL HMMER WITH I/O OPTIMIZATION EXPERIMENTAL RESULTS ON KRAKEN

the codes several times with different settings to determine the best setting.

Our work is the first research of which we are aware that combines a queuing system modeling approach with an auto-tuning framework for a parallel shared file system. Another contribution of our work is being able to tune I/O with only a small number of runs on the real system and in the presence of noise and high variability that can affect application runs.

VII. FUTURE WORK

Our on-going research focus is to build a full simulation of multiple processes and multiple OSTs when there are many processes. We need to divide them into groups and treats each group as one file. How to group them is very important and there are many ways to do it. For instance, we can do it according to the distances of the processes. Our idea is to find an optimal number of processes for each group. In this case, the other important parameter is the number of OST's assigned to each group. The difficulty here is that there may be many overlapping. So how to balance these two numbers needs to be considered. We still use the model we introduced and the only difference is that there are multiple groups to write. We assume these groups arrive at the same time and block size is fixed. So when the blocks from different groups are assigned to the same OST, we put them in the same position in the queue. There are two parts of randomness in this model, waiting time and service time. This means that we need to minimize the largest combination of waiting time and service time. In other words, we need to find the optimal pair (m, n), where m is the number of groups and n is the number of OST's for one group. The randomness of the service is from the OST assignments of the system. We assume that the system treats each group as one request and assigns the OST's to all the groups randomly and independently. Therefore, the service time on each OST depends on the number of blocks assigned. We use a random matrix to describe this situation as follows:

Let $A = (a_{ij})_{m\times N}$ be a random matrix with elements 1 or 0, where N is the number of OST's in the system. The following condition is satisfied:

$\sum_{j=1}^{N} a_{ij} = n$ for $1 \leq i \leq m$.

Let $X_j = \sum_{i=1}^{m} a_{ij}$ be the number of blocks at the jth OST, $1 \leq j \leq N$. Then the service time on the jth OST is given by $\frac{X_j M}{mnV}$, where M is the total size of the whole job and V is the writing speed on the disk.

The randomness of the waiting time can be described by queuing theory we mentioned before. Then the mathematics problem we are facing can be stated as finding (m, n) to minimize $E(\max_{1\leq j\leq N}(\frac{X_j M}{mnV} + W_j(t)1_{(X_j=1)}))$, where $W_j(t)$ is the waiting time at the jth OST.

The other question we are interested in is how to validate the parameter λ. We plan to use a statistical approach to test in the system periodically for a long time.

ACKNOWLEDGMENT

This research was supported in part by the U.S. Department of Energy Office of Science under contract DE-FC02-06ER25761 and by the U. S. National Science Foundation. In addition, this research used resources at the National Institute for Computational Sciences supported by the National Science Foundation. This research also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] Cluster File Systems, Inc., "Lustre: A scalable, highperformance file system," Tech. Rep., White paper (2002).

[2] V. E. Benes, "On queues with poisson arrivals," in *The Annals of Mathematical Statistics*, vol. 28, no. 3, Sept. 1957, pp. 670–677.

[3] U. N. Bhat, *An Introduction to Queuing Theory*. Boston, MA, USA: Birkhäuser, 2008.

[4] D. R. Cox and V. Isham, "The virtual waiting-time and related processes," in *Advances in Applied Probability*, vol. 18, no. 2, June 1986, pp. 558–573.

[5] 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.

[6] IOR, “The ASCI I/O stress benchmark,” https://computing.llnl.gov/?set=code&page=sio_downloads.

[7] S. R. Eddy, “Profile hidden Markov models,” *Bioinformatics*, vol. 14, pp. 755–763, 1998.

[8] Sean R. Eddy, “HMMER3: a new generation of sequence homology search software ,” http://hmmer.janelia.org.

[9] 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*, no. 38, pp. D211–222, 2010.

[10] S. Chen and D. Towsley, “performance evaluation of raid architectures,” in *IEEE Transactions on Computers*, vol. 45, Oct. 1996, pp. 1116–1130.

[11] E. Varki, A. Merchant, J. Xu, and X. Qiu, “Issues and challenges in the performance analysis of real disk arrays,” in *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, June 2004, pp. 559–574.

[12] D. Towsley, K. Chandy, and J.C.Browne, “Models for parallel processing within programs: application to cpu: I/o and i/o: I/o overlap,” in *Communications of the ACM*, vol. 21, no. 10, Oct. 1978, pp. 821–831.

[13] W. Chen and M. Winslett, “Automated tuning of parallel i/o systems: an approach to portable i/o performance for scientific applications,” in *IEEE Transaction of Software Engineering*, vol. 26, no. 4, April 2000.

[14] W. Yu, J. Vetter, and S. Oral, “Performance characterization and optimization of parallel i/o on the cray xt,” in *IEEE International Parallel and Distributed Processing Symposium(IPDPS’08)*, Miami, FL, USA, 2008.