A Parallelism Oriented Three Phase Based Burst Buffering Aware Scheduler

Michael Shell
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: http://www.michaelshell.org/contact.html

Homer Simpson
Twentieth Century Fox
Springfield, USA
Email: homer@thesimpsons.com

James Kirk
and Montgomery Scott
Starfleet Academy
San Francisco, California 96678-2391
Telephone: (800) 555–1212
Fax: (888) 555–1212

***Abstract*—In an computing world full of Big Data, moderate I/O performance drastically slows down the overall execution time of applications. Burst buffer nodes comes to rescue by providing both higher bandwidth and IOPS. In this paper, we model the execution of scientific applications that generates huge amount of data on supercomputing system equipped with burst buffer nodes. Computing nodes are then able to be freed earlier due to these efficient and reliable IO broker, possibly improving the utilization of task execution pipeline. We thus characterize generic applications by 3 phases: *data stage in* phase, *running* phase and *data stage out* phase. The novel 3-phase burst buffer aware workload scheduler is proposed on the assumption that resources in different phases should be scheduled separately. In both data stage input and output phase, it is possible to maximize data transfer throughput or task parallelism when allocating burst buffer. In the running phase, we consider maximize the value of scheduling tasks with both computing node resources and burst buffer node demand. Further investigating shows that all aforementioned optimization problems are equivalent to 0-1 knapsack problem, requiring exponential time to solve. We use dynamic programming and memorization technique to give precise solutions. We simulate our 3 phase scheduler scheduling a supercomputing system in the scenarios both with and without burst buffer on a discrete event simulator to demonstrate that 1) burst buffer node can accelerate the execution pipeline of all task, thus improving the utilization of the computing nodes as well as task response time 2) our scheduling algorithm can further benefit burst buffer equipped system.**

1. Introduction

Challenges due to huge amount of scientific data used by applications. Data requirements for applications at ALCF BG/P.

Storage gap between memory and disk.

Current architecture: IO forwarding nodes/IO gateways with PFSs sit between compute nodes and SAN (storage nodes + disk array)

Burst buffer nodes fills the gap by utilizing various types of memory, for example, non-volatile random-access memory (NVRAM), solid state drive (SSD).

The volume of data read/write may affect the architecture model of burst buffer.

Burst buffer nodes on Trinity is composed of IO nodes and 2 PCIe SSD cards.

In ANL's supercomputing platform, burst buffers are potential distributed on each memory hierarchy. They may be put at computer nodes, board cabin or IO nodes. We may also need to use burst buffer as intermediate storage.

2. Modeling HPC System

2.1. System Resources

The system contains resources like compute node, main memory, burst buffer and IO node. In most system architecture, compute node are coupled with main memory. Since IO capacity is more than sufficient, We assume IO nodes are always available. Therefore, the schedule targets are modeled as compute node and burst buffer. We consider a system with CN compute nodes and BB GB of burst buffer. Burst buffer are able to be shared by applications with any fraction of total amount. Later we will see that when doing optimal scheduling, we may need to allocate by a fixed fraction of the total resources(number of compute nodes or amount of burst buffer).

2.2. User Jobs

Users request the system to run their applications as jobs. We define $J = (job_1, job_2, ..., job_n)$ to be the set of all jobs in the system during a period of time interested. Job consists of 3 different phases. In the first phases, input data are load in from IO nodes to burst buffer. We refer this phase as *stage-in* phase. It is very easy for user to provide the amount of load-in data because input data is available before application execution. Then the job can run on the compute nodes when been scheduled; except reading data from burst buffer, there may or may not be any interaction between computer node and burst buffer. These interactions are mainly due to fault tolerance reasons, like check-pointing. This phase is called *running* phase. When computation is done, output data needs to be staged out to IO nodes from memory or burst buffer, namely *stage-out* phase. Burst buffer plays as IO agent for compute node. From the view point of compute node, it uses burst buffer as IO nodes. For a job at the stage-in phase, compute nodes is not allocated to it yet but it will not effect reading in data. When scheduler allocate compute nodes to a job, these nodes are exclusively used by this job until it enters stage-out phase. Compute nodes are release as soon as computation is done, meaning they are available for jobs waiting to be run. Staging output data out to IO node is the business just between burst buffer and IO nodes.

2.3. Resource Demand

Users typically provide their resource demand for their jobs. This is the most important information scheduler could get from the users. Therefore, each job is associated with a demand vector in the form of $(c, bb\_in, bb\_run, bb\_out)$, where c is the number of needed compute node in running phase, bb_in is the volume of burst buffer user predicted for file stage in, bb_run is the volume of burst buffer user preferred for checkpointing during running, bb_out is the volume of burst buffer needed to hold the resulting output data. A lazy user may just use the $\max\{bb\_in, bb\_run, bb\_out\}$ at every stage. However, we do not make any assumption about bb_run and bb_out because it is nontrivial to predict them, both for system scheduler and application owner.

3. 3-Phase Scheduler

Traditional batch scheduler just looks at the field of c_i when making scheduling decision, which just simply ignore the burst buffer node demand from the job. To be burst-buffer aware, our scheduling mechanism is divided into 3 phases to adopt to the characteristic of jobs in burst buffer context. Scheduler schedules jobs in 3 distinct set/queue. The stage-in set Q_I contains all the jobs that needs to load input data. The running set Q_R contains all the jobs waiting to be run with loaded data. The stage-out set Q_O contains all the jobs that needs to write output data to IO nodes. At anytime, a job can only appear in one of the 3 sets, apparently. This fact motivates separated scheduling idiom to be used in different phases, or for different job sets/queues.

3.1. IO-BB Scheduling

In the stage-in phase, only burst buffer demand is considered. Scheduling are made based on the value of bb_in of jobs in Q_I. If we care about data transfer throughput, we should transfer as much data as possible by doing the following optimization:

$$\max \sum_{i \in S} bb\_in_i \text{ s.t.}$$

$$\begin{cases} S \cup NS = J \\ \sum_{i \in S} bb\_in_i \leq BB_{available} \end{cases} \tag{1}$$

If we care about task parallelism, following optimization could help:

$$\max |S| \text{ s.t.}$$

$$\begin{cases} S \cup NS = J \\ \sum_{i \in S} bb\_in_i \leq BB_{available} \end{cases} \tag{2}$$

The number of tasks doing data loading will be maximized.

3.2. CN-BB Scheduling

Running jobs require not only compute nodes, but burst buffer to ensure performance and correctness. Scheduling are accordingly made based on the value of c and bb_run of jobs in Q_R. To maximize multiple types of resource's utilization, we convert it to the knapsack problem by defining the value of the job_i as

$$v_j = \frac{c_i/CN}{rt_i} \times \frac{bb\_run_i/BB}{rt_i} \tag{3}$$

where rt_i is the running time of job_i, the time it takes up the computing nodes. By definition 3, we prefer these tasks that claims to take up node resources with short duration. Unfortunately, it is difficult to predict rt_i before actually running the job. Of course we could use the *expected running time* ert_i specified by user. However, by examining the log traces from ANL, we found that the variance between rt_i and ert_i is significantly different. For now we can just assume rt_i is constant for all jobs. In the future, we could adopt machine learning or data mining ideas to predict the running time of a job with demand vector. Notice that then the value of a task is proportional to $c_i * bb_i$. The optimizing formula can thus be

$$\max \sum_{i \in S} c_i * bb\_run_i \text{ s.t.}$$

$$\begin{cases} S \cup NS = J \\ \sum_{i \in S} c_i \leq CN_{available} \\ \sum_{i \in S} bb\_run_i \leq BB_{available} \end{cases} \tag{4}$$

3.3. BB-IO Scheduling

Scheduling are made based on the value of bb_out of jobs in Q_O. Optimization formula for different purpose are almost the same as these in IO-BB scheduling.

3.4. Solving the Optimization Problems

It is trivial to show that optimization problem 1 and 2 are equivalent to 0-1 knapsack problem. Problem 4 can be informally treat as two dimension 0-1 knapsack problem. In fact, we expect all of them are NP-hard problems. We can solve them with dynamic programming in pseudo polynomial time. Applying memorization could also help accelerate the solving process. In fact we are not interested in the optimal result of problem 1, 2 and 4 at all but in one combination of jobs that yields the optimal solution, which can also be easily tracked back down by keeping memorizations.

Since problems 1, 2 and 4 are very similar, their solution is also highly related. First, for problem 1, the recursive relationship is given by 5. In 5, the memo we keeps during solving is the optimal solution for $jobs = (job_1, job_2, \ldots, job_i)$ with w GB of available burst buffer. It turns out that the recursion for problem 2 is extremely similar to 5 The memo in 6 is the same as that in 5. The recursion for 4 is a little complicated but still straightforward Here we should keep the memo of the optimal solution for $jobs = (job_1, job_2, \ldots, job_i)$ with c computing nodes and w GB of burst buffer being available.

Scheduler can obtain an optimal combination of jobs by examining the memo. Take the problem 4 problem for example. We start from $dp(n, CN, BB)$. If $c_n \leq CN$ and $bb\_sin_n \leq BB$, job_n should be scheduled if $dp(i-1, c, w) \leq dp(i-1, c-c_i, w-bb\_sin_i) + c_i bb\_sin_i$ and recurse with $dp(n-1, CN-c_i, BB-bb\_sin_i)$; otherwise, job_n should be skipped and we recurse the process on $dp(n-1, CN, BB)$. The time complexity of solving 5 and 6 is $O(n \times BB)$. The time complexity of solving 7 is $O(n \times CN \times BB)$. Notice that CN and BB may be very large integers, making the pseudo-polynomial algorithm unsuitable to be used by scheduler. In practice, we could reduce the time complexity by allocating resource in a coarser granularity. For example, jobs usually asks for compute node in the unit of 512 nodes; its demand for burst buffer nodes usually in the unit of 100 GB. Then we could divide both CN and c_i by 512; divide both BB and $bb_i n$ by 100. It is also possible to reduce the value of n, the number of jobs in the queue. For example, whenever we more resources, we can consider only $\frac{1}{\alpha}n$ jobs in the queue. This will give us only the partial optimal solution in exchange of less computation complexity.

$$dp(i,w) = \begin{cases} 0, \text{ if } i = 0 \\ dp(i-1, w), \text{ if } bb\_in_i > w \\ \max\{dp(i-1,w), dp(i-1, w - bb\_in_i) + bb\_in_i\}, \text{ if } bb\_in_i \leq w \end{cases} \tag{5}$$

$$dp(i,w) = \begin{cases} 0, \text{ if } i = 0 \\ dp(i-1, w), \text{ if } bb\_in_i > w \\ \max\{dp(i-1,w), dp(i-1, w - bb\_in_i) + 1\}, \text{ if } bb\_in_i \leq w \end{cases} \tag{6}$$

$$dp(i,c,w) = \begin{cases} 0, \text{ if } i = 0 \\ dp(i-1, c, w), \text{ if } c_i > c \text{ or } bb\_run_i > w \\ \max\{dp(i-1,c,w), dp(i-1, c-c_i, w - bb\_run_i) + 1\}, \text{ if } c_i \leq c \text{ and } bb\_run_i \leq w \end{cases} \tag{7}$$

4. Simulation Results

We consider simulating the full Trinity super computer. The number of compute nodes on Trinity is about 18936, 9436 Intel Haswell nodes and at least 9500 Intel Xeon Phi nodes. There are 16 cores on each processor, thus totally 302976 cores. In the following experiments, we compare two identical system except that IO nodes are replaced by the same number of burst buffer nodes. Burst buffer nodes provide total capacity of 4.0 PB PCIe SSD intermediate storage. Eventually Trinity will support up to 576 burst buffer nodes of 3.7 PB. Sequential read/write between burst buffer and compute nodes is 8.0 GB/s. Bandwidth between CPU node and IO node is set to 2.5 GB/s. Job trace is from ANL's Blue Gene Intrepid system from January to September 2009. Each log entry contains information like jobs' submission time, running time, number of cores user requested and so on. It is truncated so that only the first 1000 jobs are used in simulation. We patched 3 fields to each job's log entry: the amount of input data $data\_in$, the amount of written data during checkpointing $data\_run$ and the amount of output data $data\_out$. We assume they follows uniform distribution with low boundary 1 TB and high boundary 60 TB.

Following sections discuss or answer 3 key questions. Will Cerberus improve application performance by utilizing burst buffer nodes? Can Cerberus with optimization improve application performance? Will job demand on burst buffer effect Cerberus?

4.1. Cerberus vs. 1-Phase Batch Scheduler

In this section, we demonstrate that by utilizing burst buffer nodes, job scheduler could improve the applications' performance. Figure x compares CDF of the response time of 1000 jobs. When scheduler can allocate burst buffer to jobs, all jobs finish in 360,000 seconds counting from their submission time. However, the worst case in system without burst buffer is catastrophical. There are jobs that takes 1,400,000 seconds to finish, almost as 4 times slow as the most non-responsive job in system equipped with burst buffer. In average case, more than 90% of the jobs scheduled by Cerberus response faster than 1-Phase Batch scheduler. The improvement mainly comes from the difference of IO operation efficiency between traditional IO nodes and burst buffer nodes. There are only less than 10% of the jobs response quicker without needing burst buffer.

4.2. Cerberus vs. Cerberus with Optimization

If we consider optimizing either burst buffer's data throughput or the parallelism across jobs, dynamic programming based job scheduler can further reduce jobs' wait time. We plot in Figure x the resulting response time of three different scheduler regarding how they handle jobs in their queues(input queue, run queue, and output queue). The first scheduler uses naive first come first serve (FCFS) policy. Whoever at the front of queue are considered favorably. The second and third scheduler treat jobs in run queue identically. They choose jobs according to the optimization solution given by 7 However, they treat jobs in the input queue and output queue differently. The second scheduler will select these jobs in its queue so that volume of transferred data is maximized. The third scheduler tries to optimize the number of schedulable jobs by using 6.

4.3. Cerberus vs. Demand Granularity

In this section we validate our 3-phase model. Applications are benefited when scheduler dividing jobs into 3 separated phases and scheduling are based on corresponding

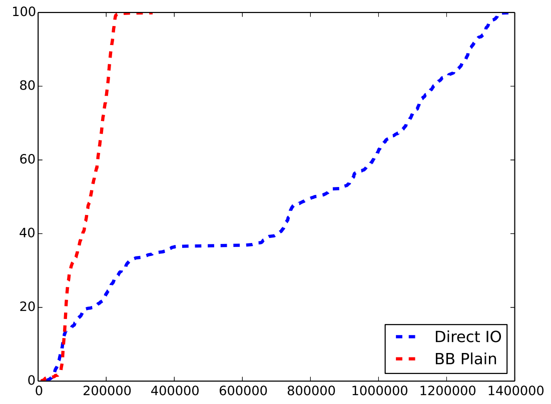

Figure 1. Application Response Time, IO Node Only vs. Burst Buffer System

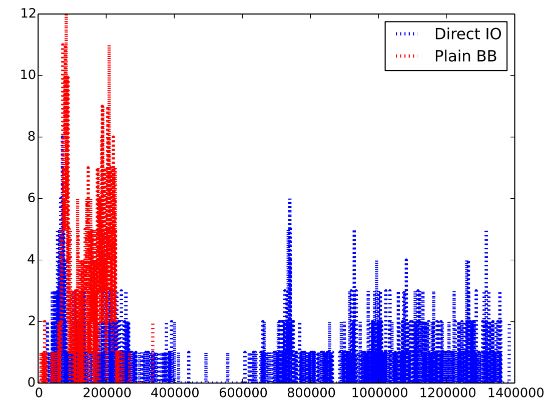

Figure 2. Application Throughput, IO Node Only vs. Burst Buffer System

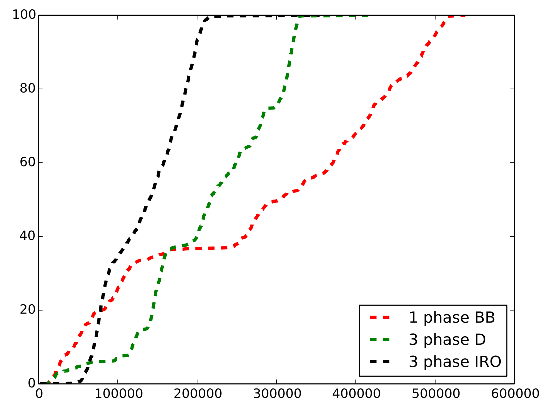

Figure 3. Application Response Time, 1 Phase Model vs. 3 Phase Model

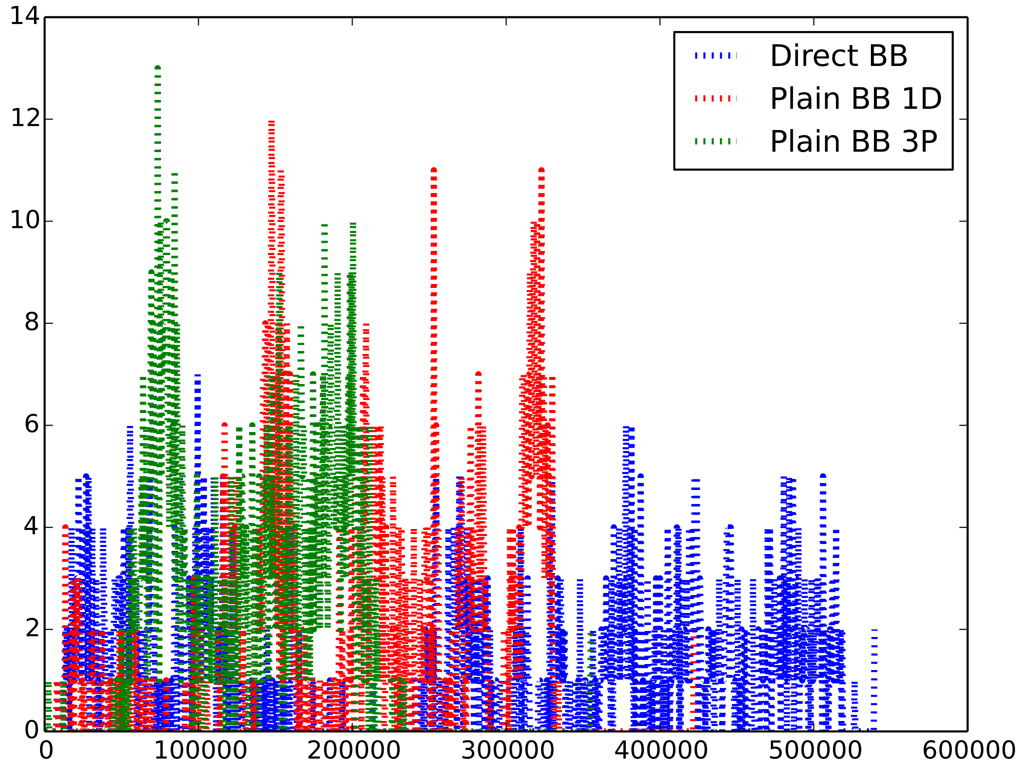

Figure 4. System Throughput, 1 Phase Model vs. 3 Phase Model

burst buffer demand in each phase. This suggests that user should provide burst buffer demand as granular as possible.

In Figure 3, we plot 3 different scheduling results by 3 FCFS scheduler. Jobs in the first case, denoted as 1 phase BB, are modeled as just 1 phase because user just provides a general burst buffer demand throughout entire application life time. We assume this demand is the $\max\{data\_in, data\_out, data\_run\}$. This is the traditional scheduling scheme except job has additional burst buffer demand and scheduler must subject to burst buffer capacity constraint. Jobs in the second and third cases have 3 phases and are scheduled by Cerberus. However, in the second case, denoted as 3-phase D, Cerberus only knows the overall burst buffer demand, same as the information in case 1. In the 3rd case, named 3-phase IRO, users kindly provided all the burst buffer demand in all 3 phases. This is the same case as in section **??** when we demonstrating burst buffer is beneficial. We simulate the 3 cases with the same generated random data volume. For 1-phase-modeled jobs, scheduler will make decision based on $\max\{data\_in, data\_out, data\_run\}$ since we assume user will only tell the upper bound of its application's demand. However, in simulation, we use the generated data amount as the same as 3-modeled jobs.

Unsurprisingly, jobs' response time is improving as long as they could utilizing burst buffer. Let's compare scheduling results of 1-phase and 3-phase, both of which only have rough data information of application. More than 60% of the 3-phase-modeled jobs finish faster than 1-phase-modeled jobs. The longest 3-phase-modeled job takes 400,000 seconds to finish while the slowest 1-phase-modeled job needs about 540,000 seconds to finish. The improvement is about 26% for the worst case. The reason of such improvement is as follows. For the 1-phase-modeled jobs, burst buffer nodes will be exclusively taken by scheduled jobs throughout their lifetime. In contrast, each time a 3-phase-modeled job finish inputing, running or outputing, Cerberus will reclaim burst buffer and CPU resources. This gives Cerberus more opportunity to schedule the system resources. At last, when comparing the case of 3-phase IRO with 3-phase D, we find another advantage of our 3-phase model. If benign users can provide finer-grain information of data/IO demand, Cerberus can programme each queue separately and get better scheduling result. In our simulation, when Cerberus knows more about application's demand in different phases, the worst absolute response time is less than 300,000 seconds. This is 25% improvement to 3-phase-modeled jobs when Cerberus only knows the upper bound of data demand, 44%

better than the slowest 1-phase-modeled job. In average case, 80% of the 3-phase-modeled jobs scheduled by Cerberus finish earlier than 1-phase-modeled jobs, on the same condition that the same amount of burst buffer nodes are available to applications. Meanwhile, more than 90% of the jobs takes less time if user specifies data usage demand at each phase to Cerberus.

Figure x describes system throughput of these three different scenarios. It helps us examine the performance of the scheduling in time sequence. For 1-phase-modeled job, we can see an obvious 'throughput gap' from 150,000 second to 250,000 second approximately. This is the result of too aggressive scheduling at the beginning. For the case of 3-phase D, throughput also starts provocatively, but not as provocatively as 1-phase scheduling case; it is then calmed due to frequent resource release, indicated by multiple crests and troughs from 0 to 350,000 seconds. The 3-phase IRO runs counter to the both previous cases. Even though beginning with throughput trough, Cerberus manages to make the system having high throughput from 50,000 to 250,000 seconds, during which system can achieve throughput around 10 to 12 per 500 seconds.

5. Conclusion

The conclusion goes here.

Acknowledgments

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.