ILLINOIS INSTITUTE OF TECHNOLOGY

PROJECT REPORT

On

Interference-aware Application Scheduler for Supercomputers

CS 550: Advanced Operating Systems

Authors:

Antoine Tehio
A20432639

Rhea Shetty
A20432742

Guided by:

Anthony Kougkas
PhD candidate

Hariharan Devarajan
PhD candidate

INDEX

1. Introduction

As technologies like the Internet of Things (IoT), artificial intelligence (AI), and 3-D imaging evolve, the size and amount of data that organizations have to work with is growing exponentially. High-performance computing (HPC) is the ability to process data and perform complex calculations at high speeds, parallely. HPC solutions have three main components:

- Compute
- Network
- Storage

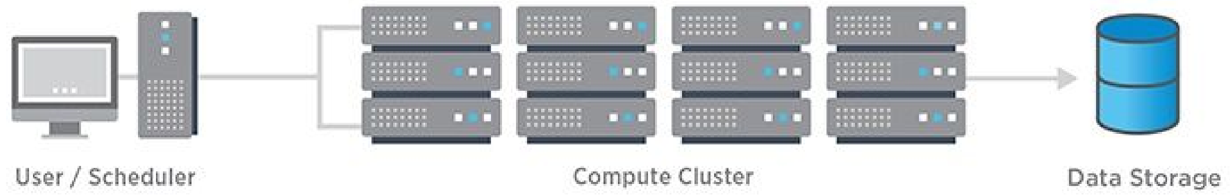


Figure 1: General Overview of HPC Systems [5]

Compute nodes in HPC have very high processing power and as a result their computing performance is phenomenal. Therefore, data is processed by compute nodes very quickly. The data nodes that store this data face the issue of being I/O bound, which causes a bottleneck. These nodes have low bandwidth, and high storage capacity, and slow I/O speeds. Accessing the large data requirements of compute nodes from data nodes throttles the performance of HPC systems.

To mitigate this issue, an intermediate layer of storage having high bandwidth and lower storage capacity was introduced. This layer is called burst buffer. The burst buffer has fewer nodes that are shared by multiple compute nodes. Burst buffer nodes are made of SSDs and NVMe (non-volatile memory express). The compute nodes transfer data to the burst buffer nodes for I/O. The burst buffer nodes then flush out the data to data nodes, hene performing the I/O while the compute nodes continue computing other tasks. The burst buffer nodes can be thought of as a temporary storage layer. It can be used for caching as well.

The I/O of applications running on compute nodes is usually in phases. An I/O phase is followed by a compute phase of the compute nodes. There is a large I/O phase involved at the beginning and end of an application, which is evident in the following graph:

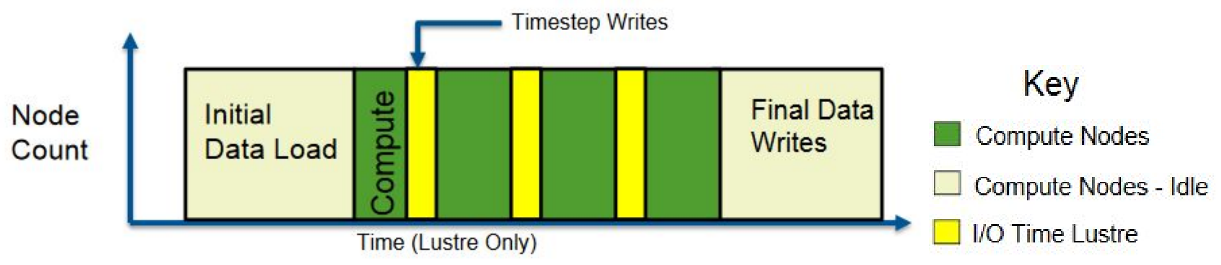


Figure 2: I/O phases over time [6]

Cray's Datawarp is a software defined storage solution for allocating storage of burst buffer nodes and it is integrated with Slurm, a workload manager. It even provides a global namespace, thus acting as a file system for burst buffer nodes. This enables the usage of burst buffer nodes as a global storage cache.

2. Problem Statement

Every application sends its I/O requests to the burst buffer nodes. As burst buffer nodes are a shared resource, many applications compete with each other for bandwidth. This contention is called interference. Interference degrades the performance of the system. Burst buffer management is a fairly new concept. Hence, the softwares available for it are still nascent. They do not address the interference issue. Therefore, there is a need for a scheduler that is aware of the interference caused by applications.

Additionally, when an application is allocated resources on the burst buffers, the allocation remains exclusive to that application until all I/O requests from the application are complete. But as we know from the nature of these applications, these I/O phases are in bursts and are followed by compute phases. The burst buffer nodes are idle during compute phase as there is no I/O to perform. This reduces efficiency as the buffer nodes are not being utilized optimally. During the compute phase of one application, the buffer nodes can be allocated to another application's I/O phase. To achieve this, the scheduler should have information about the I/O phases of each application and must be able to schedule other applications when the I/O phase of one application is finished.

The allocation of buffer nodes resources to applications is a dynamic process. I/O requests are constantly scheduled and flushed out at any given point in time. The scheduler must know the state of the buffer nodes at every instant to be able to schedule I/O requests and ensure correctness.

3. Proposed Solution

To address the problems mentioned in Section 2, an interference aware application scheduler was proposed in Harmonia [1]. Harmonia solves the I/O bottleneck problem and improves the utilization of burst buffers. This scheduler acts as an application orchestrator.

3.a) Functionalities

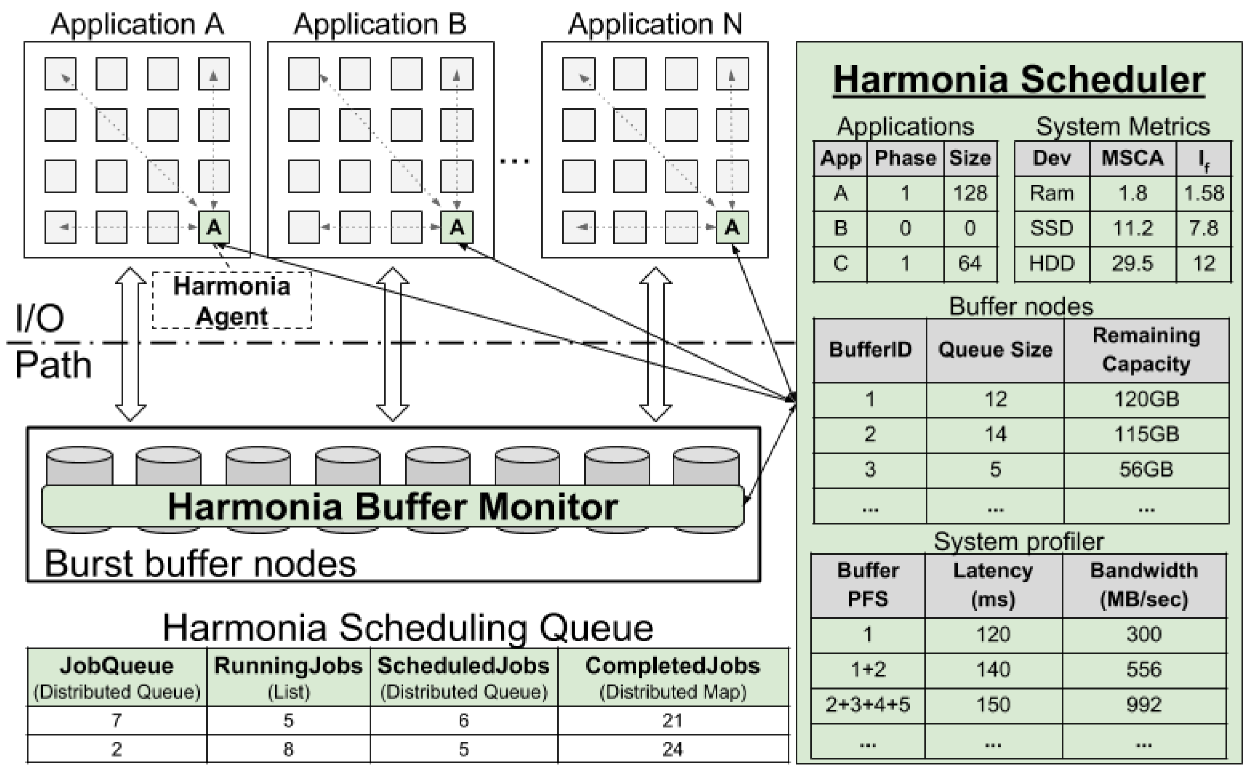
Our goal is to implement Harmonia as efficiently as possible, while addressing the issues mentioned in Section 2. The system have the following capabilities:

1. The I/O phases of applications are usually in bursts. These bursts have peak amounts of I/O requests, especially during the start and end of an application. Our system **should/would/is(?)** be able to handle these peak requests that are received and process them as quickly as possible.
2. The received requests are processed in batches of a fixed size. If the number of requests received over a fixed amount of elapsed time is less than the batch size, the requests are processed regardless of the size of the batch. This way no request will wait forever.
3. The system implements one of the five scheduling policies described in Harmonia, depending on the user's choice (the parameter can be change in the code).
4. The system is constantly be aware of burst buffers status by updating the buffer nodes before scheduling. It keeps track of the state of its resources.
5. On completion of an I/O phase, the buffer nodes should be flushed and the resources should be made available for other applications to use.

By taking care of the points above, the system aims to increase the buffering system efficiency by minimizing I/O interferences and maximizing the overall system performance through optimization of the implementation.

3.b) Design

Our design is inspired by the design of Harmonia. From the design of Harmonia, we inferred that our system would need an "agent" that interacts with the applications, a "monitor" that is aware of the buffer nodes status, and a "scheduler" to perform the scheduling of I/O phases to buffer nodes.



Harmonia Scheduling Queue

JobQueue (Distributed Queue)	RunningJobs (List)	ScheduledJobs (Distributed Queue)	CompletedJobs (Distributed Map)
7	5	6	21
2	8	5	24

Applications

App	Phase	Size
A	1	128
B	0	0
C	1	64

System Metrics

Dev	MSCA	I_f
Ram	1.8	1.58
SSD	11.2	7.8
HDD	29.5	12

Buffer nodes

BufferID	Queue Size	Remaining Capacity
1	12	120GB
2	14	115GB
3	5	56GB
...	...	...

System profiler

Buffer PFS	Latency (ms)	Bandwidth (MB/sec)
1	120	300
1+2	140	556
2+3+4+5	150	992
...	...	...

Figure 3: Harmonia An Interference-Aware Dynamic I/O Scheduler [1]

Our system has client server architecture. In our system, the application scheduler (Harmonia) is the server that processes client requests. It is a separate node in the network. Each application is a client of the server. The clients communicate with the server over the network. They send job requests in the form of I/O phases for the server to schedule. The overall design of the system is as follows:

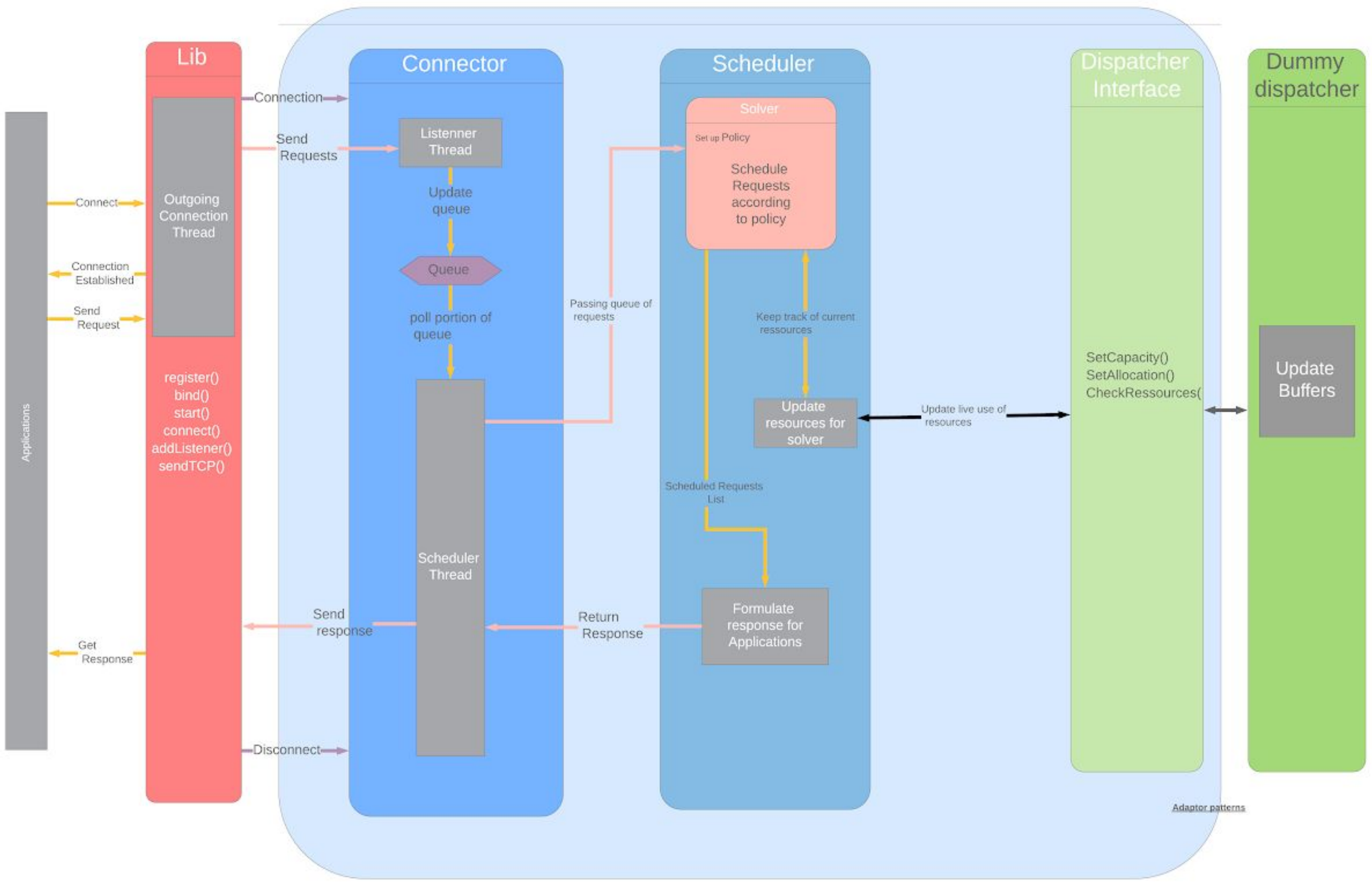


Figure 4: Overall Component Design of the System

The four major components of the system are:

1. Library: The Library interacts with applications and handles communication with the server. Requests generated by applications are sent over the network using the library. Responses to applications' requests generated by the server are then given to the applications, that created the request, by the library.

2. Connector: This is the first component that is inside the server node. It communicates with the library to receive requests and send responses to and from the applications. It listens for incoming client connections and requests. It sends batches of requests to the next component (Scheduler) for scheduling.

3. Scheduler: This component forms the crux of the entire system. It accepts batches of requests from the Connector for processing. It also receives updates about the buffer nodes status constantly. It implements one of the five scheduling policies to schedule the requests in an optimum fashion using information of buffer node states. It then generates a response based on the output of the scheduling policy and sends it to the Connector to send to the client.

4. Dispatcher: Ideally, the last component should have been Datawarp to update buffer nodes. We have simulated buffer nodes and it is called Dummy Dispatcher. The Dispatcher component handles the resources of Dummy Dispatcher. It also tells the updated status of buffer nodes mimicked in the Dummy Dispatcher to the Scheduler. The size of the resources can be changed in the code (number of buffers, size of buffers).

A sequence diagram depicting the above mentioned flow of the program is as follows:

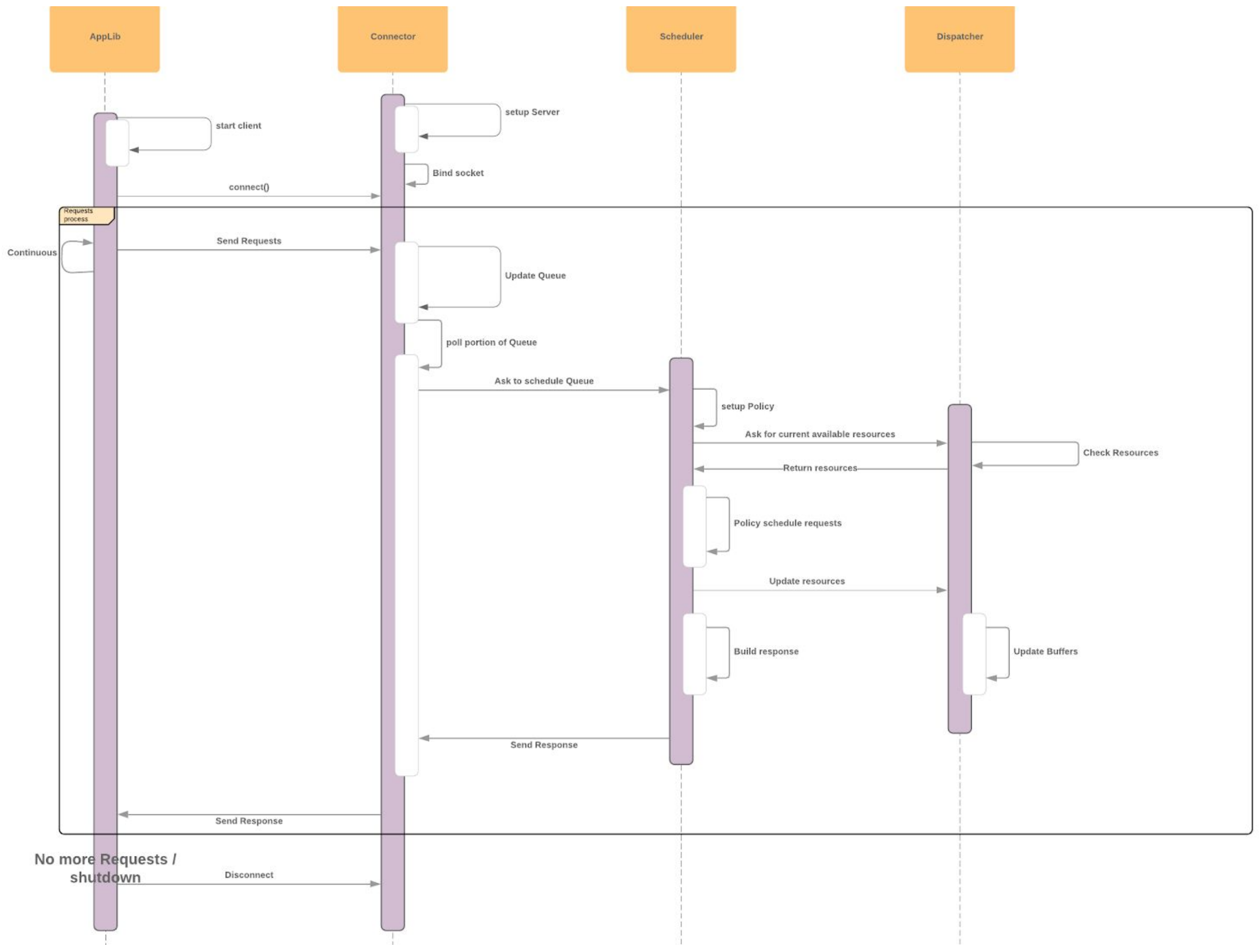


Figure 5: Sequence diagram of the system

The diagram shows the execution flow of the system, from the beginning of the client till the client disconnects.

4. Implementation

The system is implemented in Java programming language and is a multi-threaded program. The requests generated by applications contain the following information:

a. Priority: An integer signifying the priority value of the application
b. Payload: Data for which I/O needs to be performed
c. Completion Time: Time required to complete the I/O phase

The communication between the Library and Connector is asynchronous. KryoNet, a Java library that provides a clean and simple API for efficient TCP and UDP client/server network communication using NIO, is used to handle the asynchronous TCP communication. KryoNet uses the Kryo serialization library to automatically and efficiently transfer object graphs across the network.[3]

A queue is used to store incoming requests. The queue is implemented using Linked List in the Connector module. Thread pool executor is used for handling the asynchronous connection callback threads. The pool executes when a batch of requests is ready for scheduling. These scheduler threads call the solver method of the Scheduler component for executing scheduling policies. Maintaining synchronicity between scheduler threads was a key challenge of our implementation. We had to ensure that the state of buffer nodes updated by one thread was reflected to other threads before the other threads start scheduling, to ensure correctness.

The Scheduler uses factory pattern for the implementation of the scheduling policies, whereas the Dispatcher has an adaptor pattern. The five scheduling policies that we implemented are:

a. Application Priority
b. Maximum Buffer Efficiency
c. Maximum Bandwidth Utilization
d. Minimum Stall Time
e. Application Fairness

Once the Scheduler has scheduled the requests, the response is sent to the clients. The response contains the buffer node ID that has been allocated and the time at which the I/O must start. The client has a timeout period, within which if a response is not received, the client quits.

5. Results

The system was tested on Asus Zenbook having Intel Core i5-6200U x64 processor with 2.30GHz 2.40GHz speed, 8 GB RAM, 1TB HDD, and Windows 10 operating system. The system operates a little faster it does not affect the communication overflow limitations.

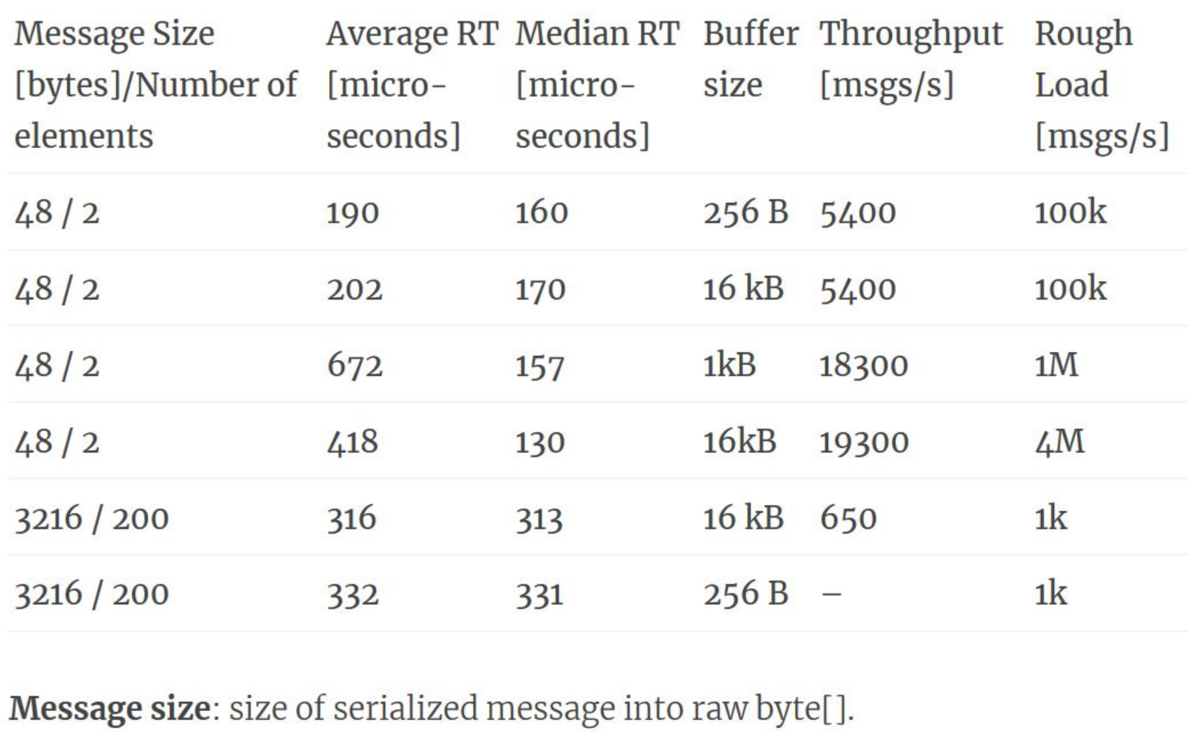
The system was evaluated to test the following:

1. Flooded with requests: The I/O phases of applications occur in bursts and each application has peak I/O operations at the start and end of the application. We flooded the system with I/O requests to test how many requests it could handle.
2. Time required to process requests: This marks the throughput of the system.
3. Time required to send response: The response time considers the throughput as well as the network latency.
4. Memory utilization of the program: The heap and stack space required by all threads and other components of the system were measured through Java programming language's runtime environment.

The following readings were taken before and after optimization of the program:

System performance	**Before**	**After**
Average Response Time	0.182 ms	0.060 ms
Throughput	500 req/s	10000 req/s
Max Request Capacity	1M	1M
Memory Utilization	1MB	2MB

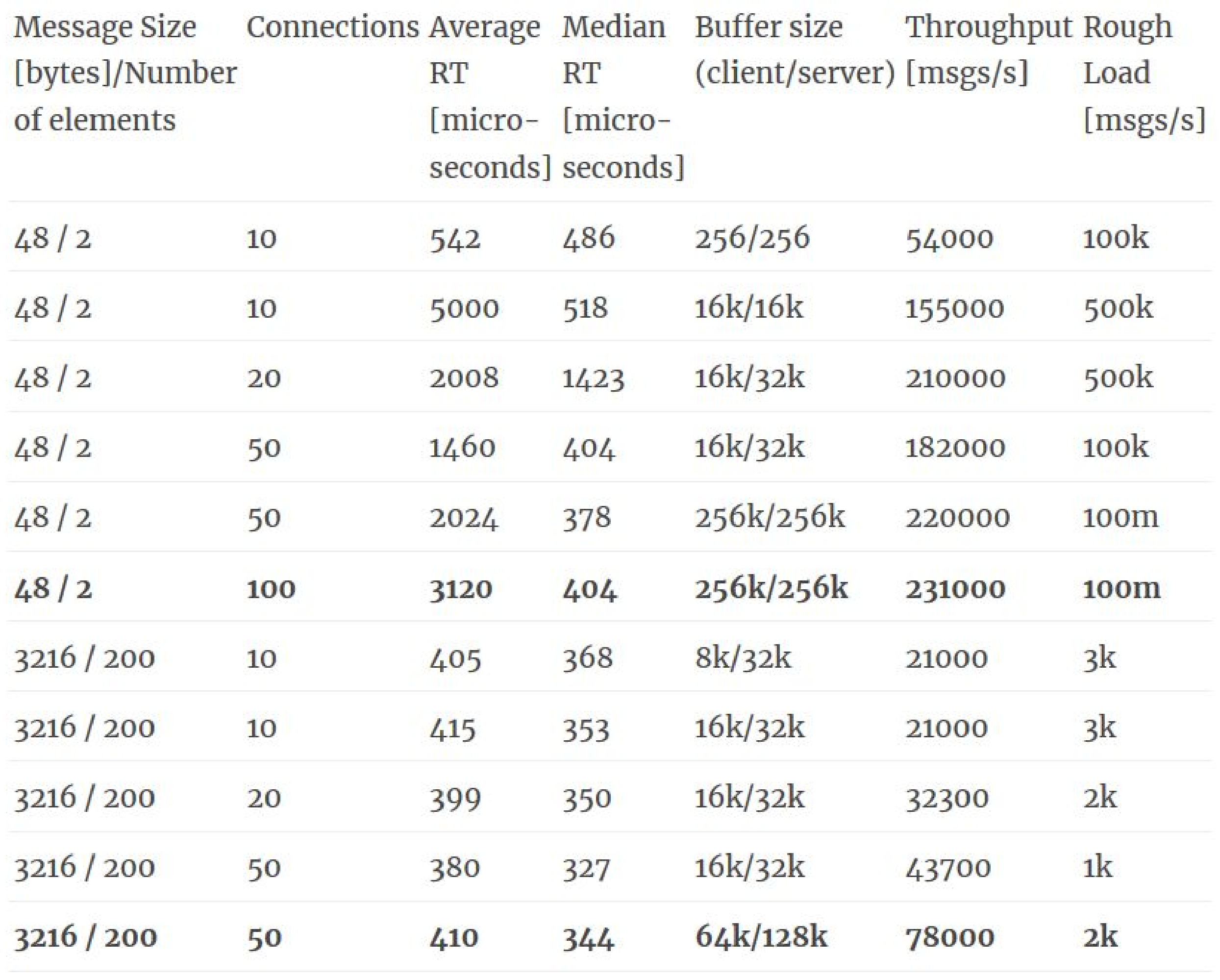
The performance of the system improved significantly post optimization. Results can be improved with further optimization. The performance of the asynchronous communication will be constrained by the performance of KryoNet. KryoNet uses two buffers, Object buffer and Write buffer, which causes a bottleneck in the communication of the system. The size of Object buffer has to be at least as big as the largest object to be transferred. In KryoNet, a single thread is used for deserialization, this results in a huge throughput with really high latency. Additionally, There is no queuing and just the rejection of a message can happen if the connection is under heavy load. The performance of KryoNet [4] is as follows:



Message Size [bytes]/Number of elements	Average RT [micro-seconds]	Median RT [micro-seconds]	Buffer size	Throughput [msgs/s]	Rough Load [msgs/s]
48 / 2	190	160	256 B	5400	100k
48 / 2	202	170	16 kB	5400	100k
48 / 2	672	157	1kB	18300	1M
48 / 2	418	130	16kB	19300	4M
3216 / 200	316	313	16 kB	650	1k
3216 / 200	332	331	256 B	–	1k

Message size: size of serialized message into raw byte[].

The performance of KryoNet when multiple connections on multiple ports are accepted is as follows:



Message Size [bytes]/Number of elements	Connections	Average RT [micro-seconds]	Median RT [micro-seconds]	Buffer size (client/server)	Throughput [msgs/s]	Rough Load [msgs/s]
48 / 2	10	542	486	256/256	54000	100k
48 / 2	10	5000	518	16k/16k	155000	500k
48 / 2	20	2008	1423	16k/32k	210000	500k
48 / 2	50	1460	404	16k/32k	182000	100k
48 / 2	50	2024	378	256k/256k	220000	100m
48 / 2	**100**	**3120**	**404**	**256k/256k**	**231000**	**100m**
3216 / 200	10	405	368	8k/32k	21000	3k
3216 / 200	10	415	353	16k/32k	21000	3k
3216 / 200	20	399	350	16k/32k	32300	2k
3216 / 200	50	380	327	16k/32k	43700	1k
3216 / 200	**50**	**410**	**344**	**64k/128k**	**78000**	**2k**

This shows that using multiple ports in the Connector for receiving client connections can improve the performance of the system. But with this increase, the Scheduler also needs to be upgraded to be able to handle the increased amount of requests that are received.

Those results are falsed by the fact that in the program we update the “Dummy” buffers and so the scheduling time is increased. New tests must be done after the implementation of DataWrap and Slurm in order to provide real insight on the performances.

6. Conclusion & Future Scope

The problem of interference and exclusive allocation of burst buffer nodes to applications can be solved by using an interference aware application scheduler like Harmonia. Our implementation is the first step towards Harmonia.

In the future, It can be implemented to integrate with existing technologies like Datawarp and Slurm in order to perform the scheduling tasks efficiently. It is the next step in the project, in order to put it in action, it must be linked to the burst buffers via DataWrap and Slurm. It would also give us more insight about the performances of this program.

The program could be able to switch from the different policies while it is running in the request of the user.

The system could further be improved by having the Connector communicate on multiple ports. KryoNet's serialization and deserialization creates a bottleneck. A Kryonet communication queue can be implemented. Kryonet gives a robustness to the program in the communication part. But is it not limiting the performances (the limited number of requests that can be handled is defined by the capacities of Kryonet).

In the scheduling of the buffers, should buffers be able to reschedule between them independently from Harmonia's scheduling. This way the stall time of the requests could be improved. Some questions still have to be figured out in order to create an optimal program.

Concurrency needs to be taken care of in systems like these that have parallel execution of threads. Concurrent execution delivers good throughput of the system. The requests received by the Connector are currently processed on first come first basis based on arrival time. A priority queue can be implemented to sort requests as per their creation time to provide fairness. With these improvements, the system can be perfected and deployed with real burst buffer nodes instead of a Dummy Dispatcher. An optimization of the synchronization can be done in the future. This would improve the speed of execution and the correctness.

Acknowledgment

This material is based upon the work of Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun, “Harmonia: An Interference-Aware Dynamic I/O Scheduler for Shared Non-Volatile Burst Buffers.” Technical Report. Illinois Institute of Technology 2017.

References

[1] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun, “Harmonia: An Interference-Aware Dynamic I/O Scheduler for Shared Non-Volatile Burst Buffers.” Technical Report. Illinois Institute of Technology 2017. https://ieeexplore.ieee.org/document/8514889

[2] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun, “Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system,” in Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing. ACM, 2018, pp. 219–230. https://www.researchgate.net/publication/325708791_Hermes_a_heterogeneous-aware_multi-tiered_distributed_IO_buffering_system

[3] KryoNet Library source: https://github.com/EsotericSoftware/kryonet

[4] KryoNet performance: https://martin.podval.eu/2014/06/13/kryonet-simple-but-super-fast-tcp-communication-in-java-performance-benchmarks/

[5] HPC information: https://www.netapp.com/us/info/what-is-high-performance-computing.aspx

[6] Datawarp overview: https://www.nersc.gov/assets/Uploads/dw-overview-overby.pdf

[7] Dave Henseler, Benjamin Landsteiner, Doug Petesch, Cornell Wright, and Nicholas J. Wright. Architecture and Design of Cray DataWarp

[8] Slurm: https://slurm.schedmd.com/overview.html

[9] Burst Buffer Architecture : https://www.nersc.gov/users/computational-systems/cori/burst-buffer/burst-buffer/