Benchmarking the state-of-the-art Task Execution Frameworks of Many- Task Computing: MATRIX Bench

Kumar Singh, Dhirendra| Bhatia, Gopal Krishna| Bhatia, Varun| Kamra, Akshay
A20331285| A20327723| A20325982| A20327452
{dkumarsi, gbhatia5, vbhatia2, akamra}@hawk.iit.edu

Abstract-- In recent years, the world of high performance computing has been developing rapidly. The goal of this project is to benchmark and compare all 8 different systems in detail about their performance under the same workloads up to 128 VMs on Amazon EC2. Many Task Computing, an emerging programming paradigm on supercomputers embraces many applications in such domains as biology, economics, and statistics, as well as data intensive computations and uncertainty quantification. Its high inter-task parallelism and intense data processing features place new challenges on the existing hardware- software stack on supercomputers. Those new challenges include resource provisioning, job scheduling, load balancing, data management and resiliency. Many-Task Computing (MTC) is a distributed paradigm of data-flow driven programming models, which bridges the gap between High Performance Computing (HPC) and High Throughput Computing (HTC). MTC applications are structured as graphs of loosely- coupled short tasks, with explicit input and output dependencies forming the graph edges. Many-task computing denotes high-performance computations comprising multiple distinct activities, coupled via file system operations. The average number of tasks and volumes of data may be extremely large. MTC needs scalable task execution framework to handle billions of jobs/tasks. The representative runtime systems for MTC are Charm++, Legion, Swift, Hadoop YARN, and Spark. Each one uses a task execution framework similar to MATRIX .

I. BACKGROUND

(Benchmarking is) the ongoing activity of comparing one's own process, product, or service against the best-known similar activity, so that challenging but attainable goals can be set and a realistic course of action implemented to efficiently become and remain best of the best in a reasonable time. As computer architecture advanced, it became more difficult to compare the performance of various computer systems simply by looking at their specifications. Therefore, tests were developed that allowed comparison of different architectures.

Benchmarks are designed to mimic a particular type of workload on a component or system. Synthetic benchmarks do this by specially created programs that impose the workload on the component. Application benchmarks run real-world programs on the system. While application benchmarks usually give a much better measure of real-world performance on a given system, synthetic benchmarks are useful for testing individual components, like a hard disk or networking device.

II. PROBLEM STATEMENT

Looking at the near future in order to enhance the computer performance the data intensive computing is very important. In this type of computing the data that is transferred or processed in the computer is taken into consideration. There are various systems such as MATRIX, Swift, Charm++,Legion,Yarn,Spark,which are contributing in Data intensive computing. All these systems focus on one thing about how the data should be more efficiently handled. As MATRIX runs on a distributed architecture, here we run nodes on different systems including MATRIX so that we can concur which system is better and gives a better efficiency and throughput with minimum latency. The earlier stages of benchmarking developments stressed a process and/or activity orientation. Recently, however, the scope of benchmarking appears to have expanded to include strategies and systems. Despite recent advancements, the field of benchmarking still suffers from the lack of theoretical developments, which are badly needed to guide its multi-faceted applications. The Many-task computing (MTC) paradigm aims to define and address the challenges of scheduling fine-grained data-intensive workloads. This fine-grained nature

of the architecture permits a more flexible arbitration of commands between behaviors and provides incrementally added behaviors with complete access to the internal state of existing behaviors.

III. MOTIVATION

The motivation for the project mainly focuses on understanding the benchmarking and the development of the given systems. The given systems of the project were compared and contrasted with MATRIX providing us the results with the latency and the throughput for each of the given system. During this particular phase of development the primary goal of the project never swayed away from analyzing the efficiency of the system under various loads and various performance criteria's. So to suffice this need of project, we underwent the project through sleep tasks, or by putting load on the system or invoking the N-queens problem on a given system.
The systems were then installed and executed to understand the execution framework and the way the system could have been set up to have dependent and reliable values. To cater to this the systems were executed on various amazon cloud service platforms. During these executions all of these data is then monitored and is then plotted for the graph, by studying which we can understood the way the system reacts when under different loads and the way it performs in the execution.
Thus by studying all these graphs which were then contrasted to that of MATRIX, we can understand the way various systems perform under the similar load and which systems performs better.

IV. RELATED WORK

Our work is related to a broad range of previous work, including programming models and languages, task distribution systems, load balancing and distributed- data systems. We are evaluating the performances of 128 VMs on six different systems and there have been previous works done on each system on the evaluation of the performances. There have been a lot of research projects that are related to our work about load balancing and data-aware scheduling. We implement our technique in MATRIX, a distributed task scheduler for many-task computing. Load balancing techniques are important to obtain the best performance for distributed task scheduling systems that have multiple schedulers making scheduling decisions. There are many examples for load balancing techniques like: Falkon, Sparrow, Mesos, Qunicy etc. Falkon aims to enable the rapid and efficient execution of many tasks on large compute clusters, and to improve application performance and scalability using novel data management techniques, it has problems to scale to petascale system and suffered from poor load balancing under unpredictable task execution times. It uses a centralized index server to store the metadata, which leads to poor scalability. Sparrow is similar to our work in that it implemented distributed load balancing and each scheduler is aware of all the compute daemons causing a lot of resource disagreement when tasks are large. Compared to MATRIX, MATRIX outperforms Sparrow nine times and also there is an implementation issues with Sparrow as it is developed in Java, which has little support in high-end computing systems. Charm++ enables users to easily expose and express much of the parallelism in their algorithms while automating many of the requirements for high performance and scalability in centralized or fully distributed fashion. But, it gives poor scalability (in case of centralized) and poor load balancing (in case of distributed) at large scales. Flexibility is key to Legion's design, allowing for continual evolution of the system to keep pace with advances in technology. Legion began as an unexpected growth of Sequoia, a locality-aware programming language capable of expressing computations in deep memory hierarchies. X10 is another parallel programming language designed to operate on distributed memory machines. X10 provides support for clusters of GPUs but requires the programmer to write all code managing data movement through both the cluster and GPU memory hierarchies. Logical regions in Legion enable the programmer to describe locality independent of memory layout.

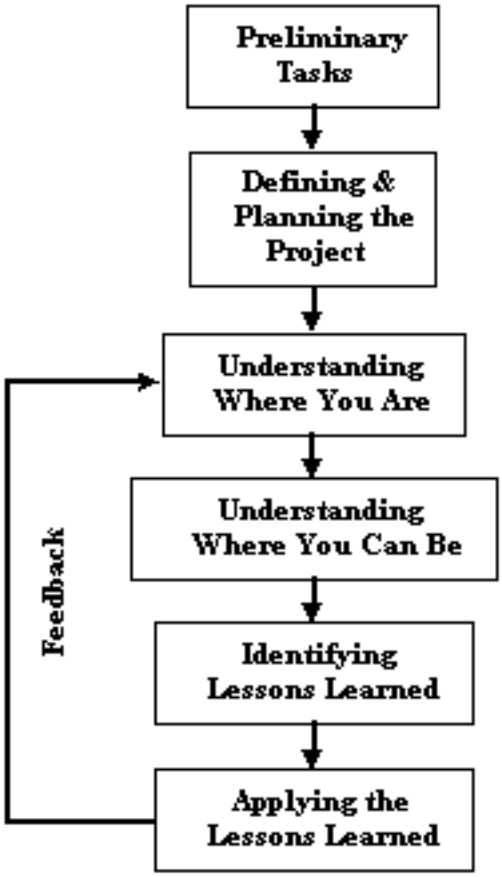
Till now, same limitation has been recognized in the Hadoop architecture. Early Hadoop clusters used some of these systems like: Condor, Torque, Moab but they did not support the Map Reduce model, neither the data locality nor the elastic scheduling needs of map and reduce phases were explainable and some of the issues were due to the distributed schedulers which were originally created to support MPI and HPC application models. These cluster schedulers do allow clients to specify the types of processing environments but not locality constraints, which is a big concern for Hadoop as of now.

There are many other examples and they all have some other issues which leads MATRIX to outperform many times, based on their poor performance.

V. PROPOSED SOLUTION

Benchmarking refers to running a set of representative programs on different computers and networks and measuring the results. Benchmark results are used to evaluate the performance of a given system on a well- defined workload. Much of the popularity of standard benchmarks comes from the fact that they have performance objectives and workloads that are measurable and repeatable In this project we compared MATRIX with the other given systems. The process was under a state of stepwise incrementing the load on the system by testing on 64 nodes with Amazon EC2 instances. These testing was done to the loads by a multiple of 2 which would provide us the result as on 1 node, then on 2 nodes then 4; 8; 16; 32; 64 and eventually on 128 nodes. But we evaluated the performances up to 64 nodes and that too only for MATRIX. We took different workloads for different systems. For example, we used Sleep tasks in Matrix and Legion but used Map-reduce application in Swift and Yarn. During this process we understood the efficiency and the performance of the system under a state of stepwise incrementing the load on the system. These values were then plotted on a graph thus providing us with the accurate information of the changing vital values of the given systems. Now that we had the basic objectives and the definitions, we need a process to achieve the objectives, which is described in the figure given aside. First we downloaded all system codes, some have been provided to us and for rest have been downloaded post which we monitored the performance on one node and then we monitored the performances and latencies up to 64 nodes. Average Time Per Task Per CPU defines the average time to execute one task from one CPU's perspective for all the tasks. Ideally, each CPU would process tasks sequentially without holding to wait tasks. Therefore, the ideal average time should be equal to the average task length of all the tasks. In reality, the average time should be larger than the ideal case, and the closer they are, the better. The throughput is the reciprocal of the average time multiplying the number of CPUs.

We implemented various systems as discussed above, each system works on different languages, we have used applications of Java, C,C++ and Python. Like Swift works on Java, we used Java in Sparrow, C++ in Matrix. We used the existing codes for each system and downloaded them and configured according to our requirements and ran the workloads to measure the performance of the system by incrementing the count of the nodes. We have used many Lines of Codes (LOC's) for every systems, for example, for matrix, we downloaded around 8000 Lines of codes, similarly, for sparrow, we downloaded around 3000 LOC, for Swift, we downloaded 2500 LOC and we wrote some shell scripts in order to help us in evaluating the system's performance more accurately and precise.



VI. ARCHITECTURE, IMPLEMENTATION DETAILS AND EVALUATION METRICES

In this section, we present the performance evaluation results of the six systems. By successfully implementing the benchmarking techniques we would be successfully be able to check and understand the system behavior by understanding the throughput and latency of the systems. During this process of variation and analysis the system would undergo variations in the processing techniques and thus by understanding the systems behavior we plot the graphs and see the variations of the systems performance by incrementing the load and can compare the results of all the systems with the performance of MATRIX. This comparative analysis would enable us to find the contrast between these systems and help us in improving the performance of MATRIX. We are going to benchmark our system based on two

parameters. Scale and Performance: Scale To test how well our system scales we are going scale our system from one node to 128 nodes performance. We are going to test our system on different workloads based on the sample file. Tasks may be small or large, uniprocessor or multiprocessor. The set of tasks may be static or dynamic, homogeneous or heterogeneous, loosely coupled or tightly coupled.

1.MATRIX:

Architecture:

MATRIX comprised of three components: client, scheduler and executor. Each compute node runs a scheduler, an executor and a ZHT server. All the schedulers are fully-connected. The client is a benchmarking tool that issues requests to generate a set of tasks, submits the tasks to any scheduler, and monitors the task execution progress. Each scheduler schedules tasks to local executor. Whenever a scheduler has no more ready tasks, it communicates with other schedulers to pull ready tasks through load balancing techniques (e.g. work stealing). ZHT is used to monitor the execution progress by MATRIX client, and to keep the system state meta-data by MATRIX scheduler in a distributed, scalable, and fault tolerant way.MATRIX aims to deliver high throughput, high efficiency, and low latency for data-intensive applications in both scientific computing and Cloud domains.

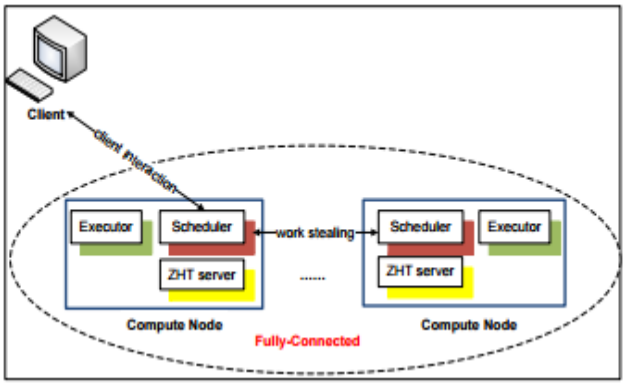


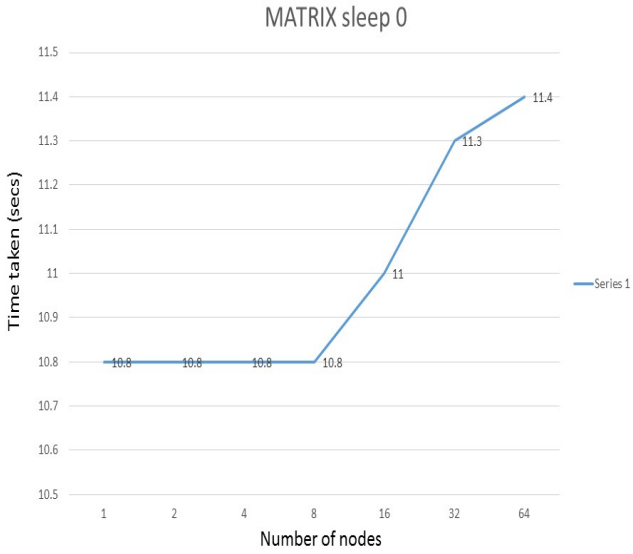
Figure 1: Matrix Architecture Overview

Implementation Details:

1. The codebase of the MATRIX version 2 is made open source on Github: https://github.com/kwangiit/matrix_v2
2. FIRSTLY install Google protocol buffers c++ binding,
3. THEN install Google protocol buffers c binding since the latter depends on the former.

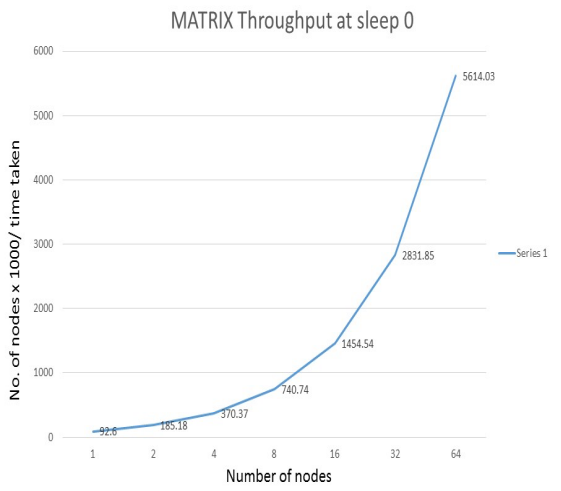
Evaluation:

Instance type used is: m3.xlarge
Location: US West (Oregon)
Number of clients used: 1 for one node, 2 for 2 nodes,4 for 4 nodes and so on.
Number of tasks used: 1000 on one node, 2000 on 2 nodes, 4000 on 4 nodes and so on.



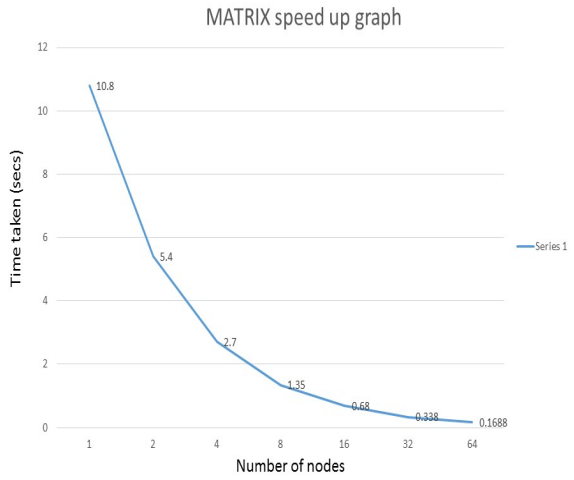
We run experiments for upto 64 nodes. For Sleep "0" task, we ran 1000 tasks for "**one single node**",it took around **10.8 seconds** ,then we kept on multiplying the number of tasks by the number of nodes. On "**2 nodes**" we ran 2000 number of tasks and it took around **10.8 seconds** ,on "**4 nodes**" we ran 4000 number of tasks and it took **10.8 seconds** and we kept on increasing the number of tasks as we increased the nodes, on "**16 nodes"**, it took **11 seconds** ,on "**32 nodes"**, it took **11.3 seconds** and on **64 nodes**, it took **11.4 seconds**. As you can see from the above graph, the graph remains constant. That means the load was properly distributed among the other nodes.

We ran the tasks on different different nodes with only **one Client**, the performance of Matrix was not very efficient. For example, when we ran the tasks on 2 nodes with only one client, it took 12 seconds but with **2 clients**, it took 10.8 seconds. Similarly, when we ran the tasks on 16 nodes with only one client, the time increased from 12 seconds to 14 seconds but when we run the tasks with more clients on 16 nodes, there was a slight change in the time taken, that is , from 10.8 seconds to 11 seconds.

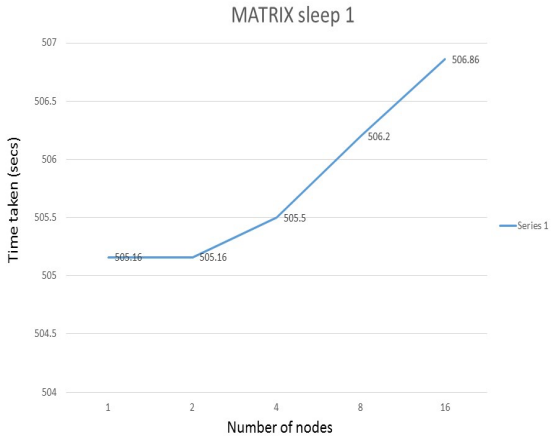


As you can see from the above graph the throughput of matrix kept on increasing as we increased the number of nodes for Sleep "0" task.

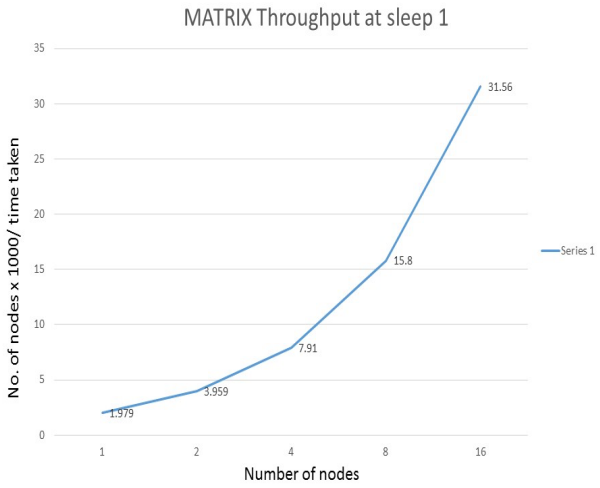
Matrix Speedup graph at Sleep 0:



We generalized the graph for all nodes,for example, to get the speed up graph, we kept same number of tasks on all nodes. On 1 node, we took 1000 tasks,on 2 nodes, we took same 1000 tasks and so on. As you can see from the above graph, on "**1 node**" ,time taken is **10.8 seconds** and on "**2 nodes"**, it took **5.4 seconds**, exactly half and on **"4 nodes"**, it took **2.7 seconds**, exactly 1/4th of the time taken on one node and so on. So, we can see from the above graph, the speed up kept on increasing as we kept on increasing the number of nodes for the same amount of task.



Similarly, For Sleep "1" task, we conducted the same experiment, we ran 1000 tasks for "**one single node**",it took around **505.16 seconds** , On "**2 nodes**" we ran 2000 number of tasks and it took around **505.16 seconds** ,on "**4 nodes**" we ran 4000 number of tasks and it took **505.5 seconds** and we kept on increasing the number of tasks as we increased the nodes, on "**8 nodes**", it took **506.2 seconds.** As you can see from the above graph, the graph remains almost constant. That means the load was properly distributed among the other nodes on Sleep "1" as well.

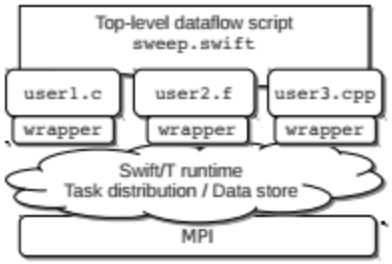


The throughput of Matrix at Sleep "1" also kept on increasing as we increased the number of nodes.

2. Swift:

Architecture:

Swift is a simple scripting language for executing many instances of ordinary application programs on distributed parallel resources. Swift scripts run many copies of ordinary programs concurrently.



Overview of Swift/T application model

Swift/T application consists of a Swift/T script and some number of user code "extensions" coded in a native programming language. These consume and produce data in native types, such as integers, floats, pointers to byte data, etc. Swift/T wrappers, which may be automatically generated, allow these extensions to be executed by the runtime. Thus, the user may produce a massively concurrent program without writing MPI directly.

```
1  main {
2    file d[];
3    int N = string2int(argv("N"));
4    // Map phase
5    foreach i in [0:N-1] {
6      file a = find_file(i);
7      d[i] = map_function(a);
8    }
9    // Reduce phase
10   file final <"final.data"> = merge(d, 0, tasks-1);
11 }
12
13 (file o) merge(file d[], int start, int stop) {
14   if (stop-start == 1) {
15     // Base case: merge pair
16     o = merge_pair(d[start], d[stop]);
17   } else {
18     // Merge pair of recursive calls
19     n = stop-start;
20     s = n % 2;
21     o = merge_pair(merge(d, start,     start+s),
22                    merge(d, start+s+1, stop));
23   }
24 }
```

Figure 2: MapReduce-like application expressed in Swift.

In this code above, find_file(), map_function(), and merge_pair() are user-written leaf functions that consume and produce ordinary files. These functions can be implemented in any language and presented to Swift.

Implementation Details:

Since Swift is written in Java, Java Runtime Environment 1.7 or greater is a required by Swift.

1. Download Java-8
2. Download the latest packaged binaries.
3. Example Swift scripts can be found in the swift-0.94.1/examples directory. Scripts have a .swift file extension. Run the following commands to launch a simple hello world application.

Evaluation:

Instance type used is: t2.micro
Location: US West (Oregon)

For Swift, we ran the simple application of Map-reduce program and we monitored the performance of the system on different nodes.We ran the application on "**2 nodes"** and it took around **17 seconds,** on "**4 nodes**", it took **19 seconds** but on "**8 nodes**" it took **18 seconds** and on "**16 nodes**" it took **15 seconds** and again at "**32 nodes**" , it came back to **14 seconds**, so if we notice from the above graph, it took average time of **16.6 seconds**.

The throughput of Swift was almost equal on 2 nodes and 4 nodes as you can see from the above graph but it kept on increasing as we increased the number of nodes.

3. Spark:

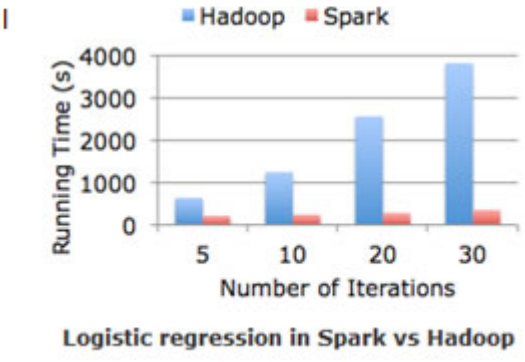
Architecture:

Spark is an open source cluster computing system developed in the UC Berkeley AMP Lab. The system aims to provide fast computations, fast writes, and highly interactive queries. Spark significantly outperforms Hadoop MapReduce for certain problem classes and provides a simple Ruby-like interpreter interface as shown in Figure 3.

```
val file = spark.textFile("hdfs://...")

file.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
```

Word Count implemented in Spark



Logistic regression in Spark vs Hadoop

Figure3: Graph showing time of Hadoop vs Spark

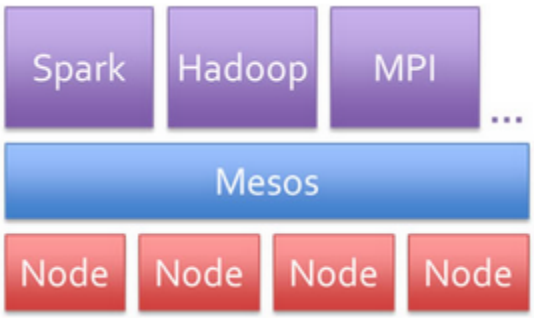


Figure4: Spark Architecture

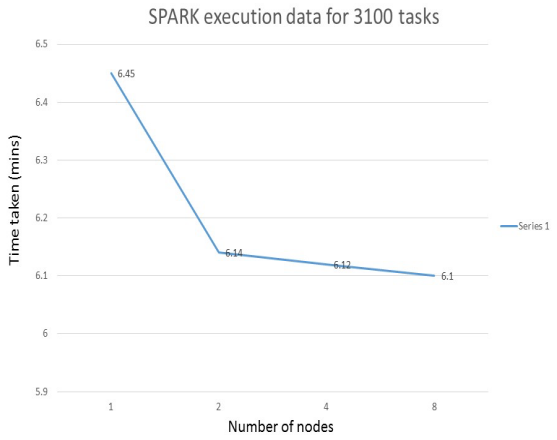
Spark is built to run on top of the Apache Mesos cluster manager. This allows Spark to operate on a cluster side-by-side with Hadoop, Message Passing Interface (MPI), HyperTable, and other applications. This allows an organization to develop hybrid workflows that can benefit from both dataflow models, with the cost, management, and interoperability concerns that would arise from using two independent clusters

Implementation details:

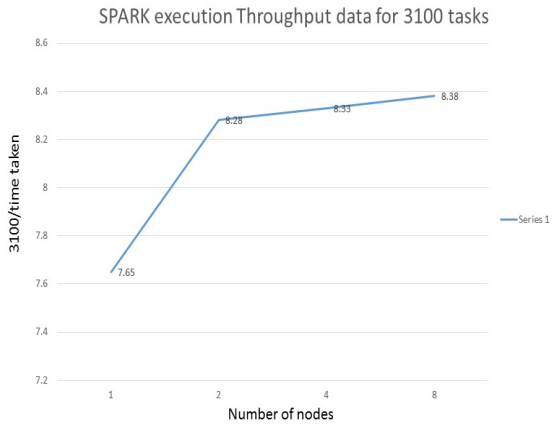
1. Download the code from github: https://github.com/radlab/sparrow
2. Build the system by changing the configuration.
3. Then run a simple frontend and backend application.

Evaluation:

Instance type used is: t2.micro
Location: US East (N. Virginia)
Number of tasks used: 3100



We ran 3100 tasks on "**1 node**" and took **6.45 seconds,** on "**2 nodes**" it took **6.14 seconds,** it gradually decreased upto **6.12 seconds** when ran 3100 task on **"4 nodes",** the execution time decreased for 3100 tasks to run as we increased the number of nodes but it was not that much it was almost equal after running on "2 nodes".



The throughput of Spark as you can see from the above graph, it gradually increased from **7.65** to **8.28** but after running it on "2 nodes", it increased but slowed down a bit.

4. Yarn:

Architecture:

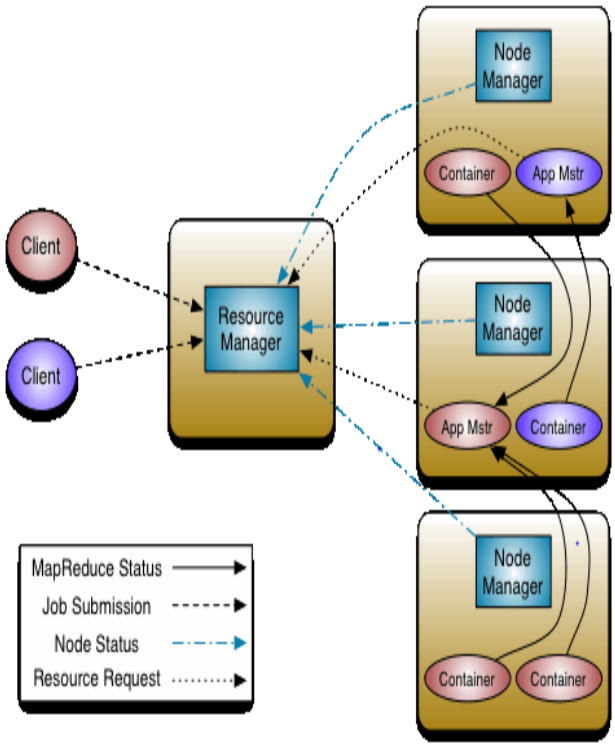


Figure5: Yarn Architecture Overview

The ResourceManager has two main components: Scheduler and ApplicationsManager. The Scheduler is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. The Scheduler is pure scheduler in the sense that it performs no monitoring or tracking of status for the application. Also, it offers no guarantees about restarting failed tasks either due to application failure or hardware failures. The Scheduler performs its scheduling function based the resource requirements of the applications; it does so based on the abstract notion of a resource *Container* which incorporates elements such as memory, cpu, disk, network etc. In the first version, only memory is supported. The Capacity Scheduler supports hierarchical queues to allow for more predictable sharing of cluster resources. The Node Manager is the per-machine framework agent who is responsible for containers, monitoring their resource and reporting the same to the Resource Manager/Scheduler.

Implementation Details:

Download the latest Hadoop version and refer this link for setup from: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html. From here download hadoop-2.5.1.tar.gz from the '/stable' folder. Using command "wget".

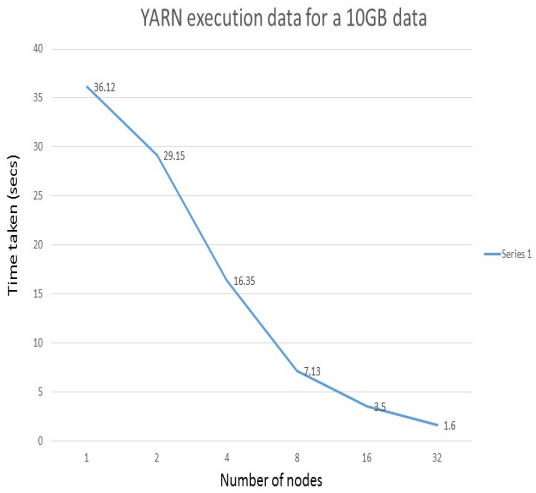
After this step we need to modify following files. yarn-site.xml; core-site.xml; mapred-site.xml; hdfs-site.xml; Yarn-env.sh; hadoop-env.sh

All the setup can be done through the link provided above.

After doing the setup we will benchmark the Yarn system by providing the workload of 10GB file, in this we have done the word count of the 10gb file and measure the time taken by the system to do word count by increasing the number of nodes.

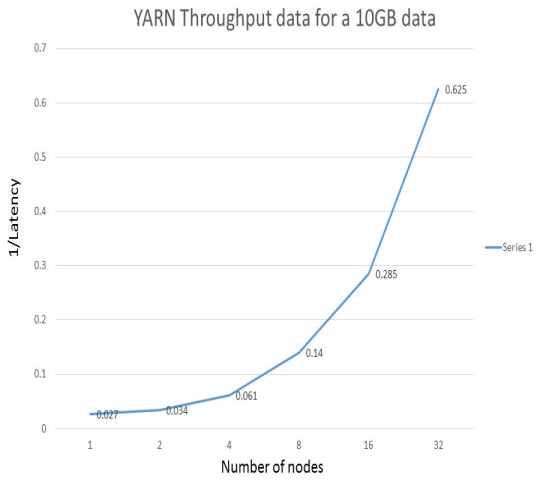
Evaluation:

Instance type used is: m3.xlarge
Location: US West (Oregon)
Workload: file



We ran 10 GB file workload and perform the operation of word count on that file. We measured the performance of Yarn on the time it took to perform the word count operation on varying number of nodes. As As you can see from the graph below as we are increasing the number nodes the time taken to perform the operation is decreasing. On "**1 node",** it took **36.12 seconds, on "2 nodes"**, it took **29.15 seconds, "4 nodes"**, it took **16.35 seconds** and so on.

Thus ,with the increase in the number of nodes time is decreasing thus performance of yarn is increasing.



The throughput of the graph shown above keeps on increasing as we run the same workload and keep on increasing the number of nodes.

5. Charm++:

Architecture:

CHARM++ is a machine independent parallel programming language that runs on most shared and distributed memory machines. It employs an object-oriented approach to parallel programming.It is a runtime system and a programming language implementing the message passing paradigm. Unlike MPI, Charm++ does not manipulate processes but independant computing objects called chares.

NQueens: NQueens is a backtracking search problem to place N queens on a N by N chess board so that they do not attack each other. We target at finding all solutions for N Queen problem.To study the evolution of the system in parallel using Charm++, we divide the NxN board into multiple pieces each of size at most kxk. The pieces together constitute a 2-dimensional array of chares. Since you need to know your neighbor's data for each cell, each chare sends its border to the 4 neighboring chares, and copies data sent by them into a ghost region around its array.

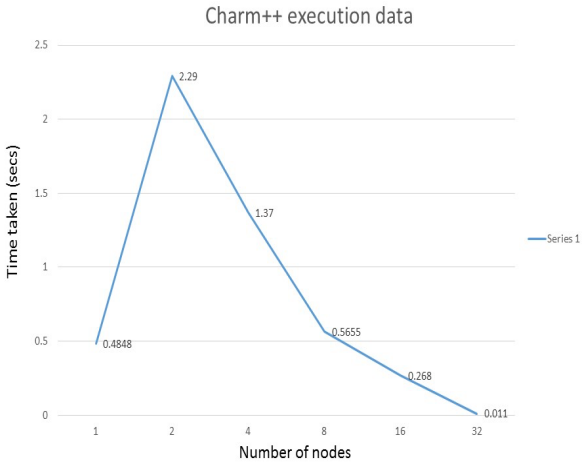
Implementation Details:

1. Download the code from : http://charm.cs.illinois.edu/software
2. Download Java and change the configuration to build the system .
3. Run the application of NQueens Problem by running the below command:

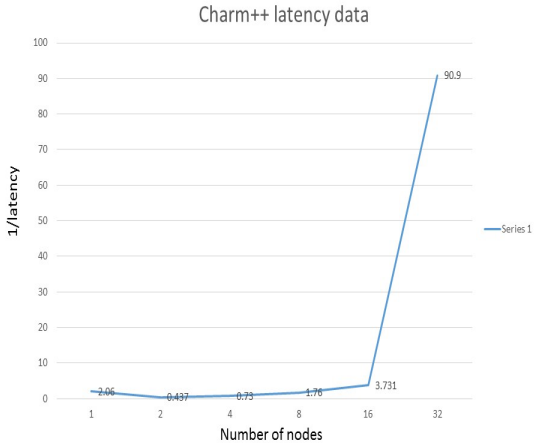
   ./charmrun ++local ./pgm 12 6 +p2

Evaluation:

Instance type used is: t2.small
Location: US West (Oregon)



We ran the "N queens problem algorithm" to find all possible solutions of this problem, we ran the application on "**1 node**", it took **0.48 seconds,** we ran the same on "**2 nodes"**, and it increased exponentially and goes to **2.29 seconds,** the graph exponentially grows on 2 nodes , it took more time when ran on 2 nodes, but as we ran the application of more nodes, the time gradually decreased as we increased the number of nodes as you can see from the graph above. For example, on 4 nodes it took **1.37 seconds**, on 8 nodes it took **0.56 seconds** and so on.



The throughput of Charm++ was little bit weird, from 1 node till 16 nodes, it was almost constant, but it grows exponentially when ran on 32 nodes.

6. Legion

Architecture:

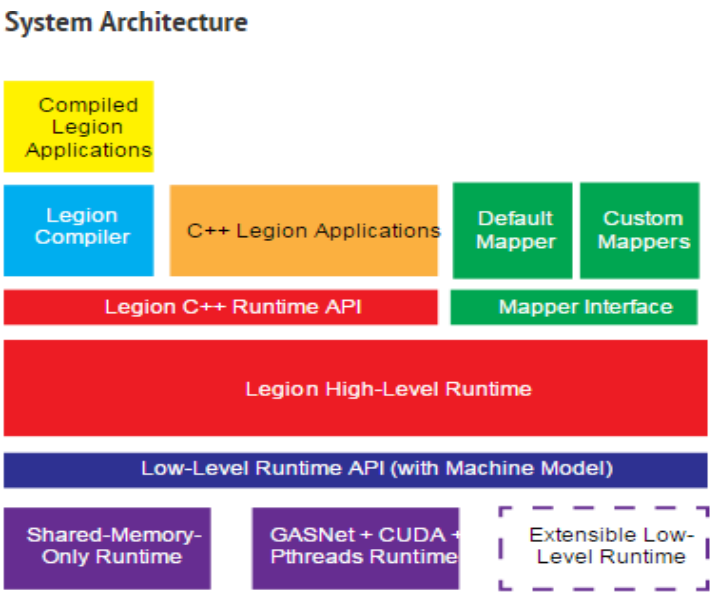


Figure6: Legion Architecture Overview

The above figure shows the architecture of the Legion programming system. Applications targeting Legion have the option of either being written in the compiled Legion language or written directly to the Legion C++ runtime interface. Applications written to the compiled Legion language are translated to the C++ runtime API by our source-to-source Legion compiler.

The Legion high-level runtime system implements the Legion programming model and supports all the necessary API calls for writing Legion applications. Mappers are special C++ objects that are built on top of the Legion-mapping interface, which is queried by the high-level runtime system to make all mapping decisions when executing a Legion program.

Implementation Details:

1. Download git clone https://github.com/StanfordLegion/legion.git
2. Set the LG_RT_DIR variable, and then build the system.
3. Once the application successfully builds, the hello_world binary should exist. The binary can be run just like any other C++ application by typing ./hello_world. The application should print “Hello World!” and exit cleanly.

Evaluation:

We ran a simple “Hello world” application and evaluated the performance based on sleep function value, for example, we ran the application with “**sleep 1 task**”, it took **1 second,** we ran the same with “**sleep 2”**, it took **2 seconds,** and when run with “**sleep 4”**, it took 4 seconds and similarly, when run with “**sleep 8”**, it took **8 seconds.**
But we encountered the below error for each CONDUIT build.

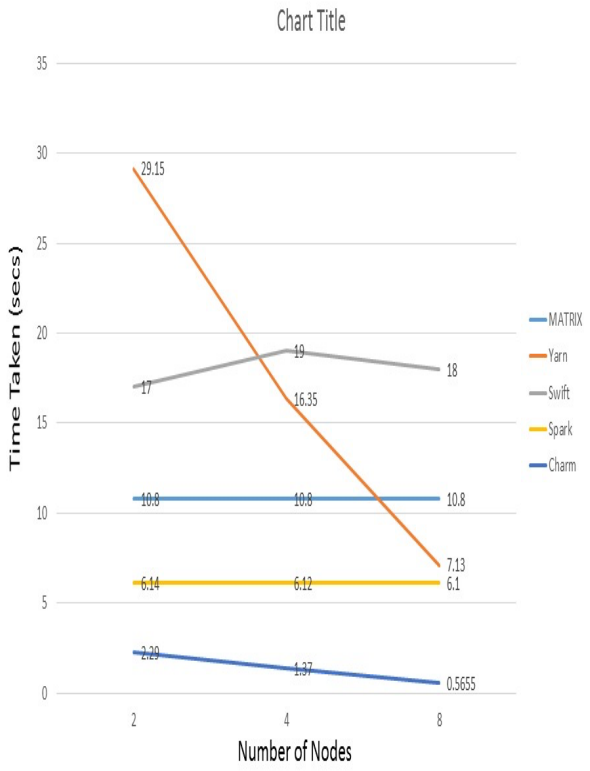
```
configure error: User requested --enable-gemini but I don't know how to build gemini programs for your system

configure: error: See /home/akshay/Desktop/DIC/GASNet-1.24.0/config.log for details.
akshay@akshay-virtual-machine:~/Desktop/DIC/GASNet-1.24.0$
```

VII. CONCLUSION

Applications for extreme-scales are becoming more data- intensive and fine-grained in both task size and duration. Task schedulers for data-intensive applications at extreme-scales need to be scalable to deliver the highest system utilization, which poses urgent demands for both load balancing and data- aware scheduling. Finally analyzed and implemented nodes on the systems and studied the throughput, latency of these systems. We came up with the results of all the systems and we had developed a concise report and arrived at values suggesting the benefit of each of the system over the other and helped us compare them to MATRIX. Thus allowing us to understand and extend the MATRIX. The most important feature of fine-grained architecture is that it permits a more flexible arbitration of commands between behaviors and provides incrementally added behaviors with complete access to the internal state of existing behaviors. The purpose of benchmarking is to improve performance or services by identifying where changes can be made in either what is done

or how things are done. So, basically we measured the performances of the systems based on the same workloads on different nodes but different workloads on different systems and compared MATRIX with the other given systems. We went step-by-step, incrementing the load on the system by testing up to 64 nodes with Amazon EC2 instances.



The graph above clearly shows the latencies of all the systems we benchmarked including MATRIX. As you can see Orange line indicates the performance for “Yarn”, Grey line indicates performance of “Swift”, blue line indicates for “Matrix”, Yellow for “Spark” and Purple for “Charm”. We have indicated only for “2nodes”, “4nodes” and “8 nodes”.

So, in conclusion, as seen from the graph above, for two systems line remains straight, that is, for Matrix and Spark. But for “Spark” time taken for 3100 tasks to run on “2 nodes” is 6.14 seconds and same number of tasks on “4 nodes”, it takes 6.12 seconds. So that means, even after increasing the number of nodes by 2(that is 4 nodes in total), there isn’t much difference in the time taken, ideally the time taken by the system to run the same task on 4 nodes should be much lesser. For Matrix, we see it took 10.8 seconds of time to run 2000 number of tasks on 2 nodes and same 10.8 seconds of time to run 4000 numbers of tasks on 4 nodes and same 10.8 seconds of time to run 8000 tasks on 8 nodes. Thus, you see time remains constant as we increase the number of tasks as we increase the number of nodes. So, in short, Matrix distributes the work properly among nodes.

Thus, we conclude that based on our analysis and the results obtained; we see that Matrix is much better system than any other systems we worked on.

VIII. FUTURE WORK

We have executed these six system on different nodes scaling up to 64 nodes performance by a multiple of 2 which provided us with the result as on 1 node, then on 2 nodes then 4; 8; 16; 32 and eventually on 64 nodes. After our success and the results obtained we have realized that more tests would be carried out over on these systems and all the systems will be scaled to 128 nodes.
Also, the future lies in real implementation of these systems by hosting large number of virtual machines as intermediate nodes and calculating the overall throughput of the system. Future work also lies in perfecting the small flaws that lies in our overall design of these systems.

In near future impetus will also be given on exploring more options on improving the performances and throughput of these systems.Our prime goal is that to work on “Legion” system properly in near future and come out with the perormance of the system on multiple nodes.And also work on all these systems with the same workload and will priortize to implement all the system with sleep task varying from 1 to 20 upto 128 nodes.

IX. REFERENCES

[1] http://www.cs.iit.edu/~iraicu/teaching/CS554-S15/proj/MATRIX-Bench.pdf
[2] https://github.com/kwangiit/matrix_v2
[3] http://charm.cs.illinois.edu
[4] http://charm.cs.illinois.edu/newPapers/14-24/sc14numa.pdf
[5] http://charm.cs.illinois.edu/newPapers/13-26/paper.pdf
[6] http://legion.stanford.edu
[7] http://legion.stanford.edu/pdfs/legion-fields.pdf
[8] http://swift-lang.org/main/
[9] http://swift- lang.org/papers/pdfs/Swift_2013.pdf
[10] http://hadoop.apache.org/docs/current/hadoop

-yarn/hadoop-yarn-site/YARN.html
[11] https://github.com/radlab/sparrow
[12] http://dl.acm.org/citation.cfm?id=1188455.1188528
[13]http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7004443
[14] *B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos*: a platform for fine-grained resource sharing in the data center. In Proceedings of the 8th USENIX conference on Networked systems design and implementation, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association
[15] L. Kale´ and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.
[16] W. W. Shu and L. V. Kale´, "A dynamic load balancing strategy for the Chare Kernel system," in *Proceedings of Supercomputing '89*, November 1989, pp. 389–398.
[17] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Pro- ceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.
[18] I. Raicu. "Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing", ISBN: 978- 3-639-15614-0, VDM Verlag Dr. Muller Publisher, 2009.
[19] im Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.
[20] M. Hategan, J. Wozniak, and K. Maheshwari, "Coasters: uniform resource provisioning and access for scientific computing on clouds and grids," in *Proc. Utility and Cloud Computing*, 2011.
[21] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Par. Comp.*, vol. 37, pp. 633–652, 2011.
[22] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, B. Clifford. "Toward Loosely Coupled Programming on Petascale Systems", IEEE/ACM Supercomputing 2008.
[23] E. L. Lusk, S. C. Pieper, and R. M. Butler, "More scalability, less pain: A simple programming model and its implementation for extreme computing," *SciDAC Review*, vol. 17, pp. 30–37, Jan. 2010.
[24] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A report on the Sisal language project," *J. Parallel and Distributed Computing*, vol. 10, no. 4, pp. 349 – 366, 1990
[25] T. G. Armstrong, Z. Zhang, D. S. Katz, M. Wilde, and I. T. Foster, "Scheduling many-task workloads on supercomputers: Dealing with trailing tasks," in *Workshop on Many-Task Computing on Grids and Supercomputers*, 2010.
[26] I. Raicu, Y. Zhao et al. "Falkon: A Fast and Light-weight tasK executiON Framework," IEEE/ACM SC 2007
[27] K. Wang, A. Rajendran, K. Brandstatter, Z. Zhang, I. Raicu. "Paving the Road to Exascale with Many-Task Computing", Doctoral Showcase, IEEE/ACM Supercomputing/SC 2012.
[28] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu. "FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems." IEEE BigData, 2014
[29] G. Zhang, E. Meneses, A. Bhatele, and L. V. Kale. "Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers", Conference on Parallel Processing Workshops, ICPPW10, 2010.
[30] Google. "Google Protocol Buffers," available at http://code.google.com/apis/protocolbuffers/, 2014.
[31] http://www.cs.duke.edu/~kmoses/cps516/dstream.html
[32] http://legion.stanford.edu/overview/
[33] http://www.cs.iit.edu/~iraicu/teaching/CS554-F13/best-reports/2013_IIT-CS554_MATRIX-HPC.pdf

X. APPENDIX

We will specify this on the basic of the whole workflow of the project.

Brainstorming sessions – Dhirendra, Varun, Gopal, and Akshay.
Research and development – Dhirendra, Varun, Gopal, Akshay
Documentation – Dhirendra, Akshay, Varun, Gopal
Deciding the workflow of the project -Gopal, Dhirendra

System Installations:
Matrix: Dhirendra, Varun
Swift: Varun, Dhirendra
Charm++: Akshay, Gopal
Legion: Akshay, Gopal
Yarn: Varun, Gopal
Spark: Dhirendra, Akshay

Installation and Evaluation:
MATRIX - Dhirendra Singh
Yarn - Varun Bhatia
Swift - Gopal Bhatia

Charm++ - Akshay Kamra
Spark - Dhirendra Singh, Gopal Bhatia
Legion - Akshay Kamra, Varun Bhatia

XI. ACKNOWLEDGEMENT

We would like to thank professor **Ioan Raicu** for giving us this opportunity to take up this project as it was a huge learning curve for us and has helped us develop our knowledge more in terms of all the systems, how differently they work and how fast they can execute the task, understanding the need of data intensive computing in the current computing world and helping us gain knowledge throughout the semester. We would also like to thank **Ke Wang** who has helped us understand all these systems in a short time and guided us perfectly.