Benchmarking Task Execution Frameworks of Many-Task Computing

Mitesh Patekar, Urvish Shah, Girija Ogale
A20325055, A20325208, A20327192
mpatekar@hawk.iit.edu, ushah7@hawk.iit.edu, gogale@hawk.iit.edu

***Abstract*—**
Soon, exascale systems will be everyday part with 1-million nodes and 1-billion threads of execution. Because of this exascale parallelism it is predicated that, in future, programming models will employ over-decomposition to generate many more tasks than available parallelism. State-of-the-art job schedulers follow a centralized paradigms. Many-Task Computing (MTC) is an example of distributed paradigms of over-decomposition which bridges gap between High Performance Computing (HPC) and High Throughput Computing (HTC). MTC applications often demand a short time to solution, so, MTC needs scalable task execution framework to handle billions of tasks. Charm++, Legion, YARN, Spark, HPX are some of the runtime systems for MTC. All these systems uses Task Execution Framework which is similar to MATRIX. The goal of this project is to benchmark and compare all these systems by calculating average task scheduling latency, throughput and efficiency of each system.

Keywords: Benchmarking, Many Task Computing, Task Execution, Framework, MATRIX, Legion, YARN, Swift and Charm++.

I. Background Information

It is predicted that by 2020, we will be dealing with exascale systems which will have millions of nodes and billions of execution threads. Exascale systems will unravel many mysteries and will present new challenges and opportunities in the field of distributed and cloud computing. It is said that future programming models will employ over-decomposition to generate many more task than available parallelism. Many Task Computing (MTC)[7] is an example of such distributed paradigms of over-decomposition.

MTC needs scalable task execution framework to handle billions of jobs/tasks. Task execution over large scale, distributed systems is important to achieve good performance and high system utilization. Within every runtime systems there remains its own Task Execution Framework (TEF) which is responsible for the execution of number of fine grained tasks of MTC applications. Most TEFs have centralized Master/Slaves architecture, where a centralized server is in charge of the task execution. Most have poor scalability at the extreme scales of with fine-granular workloads. The solution to this problem is to move to the decentralized architectures that avoid using a single component as a manager. Distributed schedulers are normally implemented in either hierarchical or fully distributed architectures to address the scalability issue.

MATRIX[1] is a distributed MTC task execution framework which uses the adaptive work stealing[1] for distributed load balancing and a distributed key-value store[10] for task metadata management. MATRIX currently supports MTC workloads.

In this project, we have evaluated and compared Task Execution Framework of MATRIX with other TEFs of such systems like Charm++[4], Legion[3], YARN[2], Swift[16] using parameters like throughput, efficiency and latency providing some workloads to each system.

II. Problem Statement

This project aims to benchmark and compare the TEFs of Charm++, Legion, YARN, Swift**,** and MATRIX runtime systems. MATRIX is Fully-distributed task execution framework for Many-Task Computing data intensive applications. Charm++ is an Object-oriented, asynchronous message passing parallel programming framework in C++ supported by an adaptive runtime system. It enhances user productivity and allows programs to run portably from small multicore computers to the largest supercomputers. It 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. YARN stands for "Yet-Another-Resource-Negotiator". It is next generation compute framework for Hadoop. YARN facilitates writing arbitrary distributed processing frameworks and applications. It provides the daemons and APIs necessary to develop generic distributed applications. Swift is a parallel scripting language that allows the writing of scripts that distribute program execution across distributed computing

resources including multicores, clusters, clouds, grid and supercomputers. Legion is data-centric parallel programming system for writing portable high performance programs targeted at distributed heterogeneous architecture. It provides a common framework for implementing applications which can achieve portable performance across a range of architectures.

The main task of this project is to figure out how to run the same workloads over all above mentioned TEFs of MTC and add code to measure the same performance metrics. We need to figure out exactly where scheduling happens i.e. we need to find where scheduling starts and where does it finish. And then add timestamps between two interfaces.

Once we figure out where exactly scheduling starts and where does it finish, we need to add source code for benchmarking the systems. We are to measure performance metrics throughput, average scheduling latency and efficiency of six different TEFs of runtime systems.

The challenging part of this project is to run the systems under same workloads on different architecture of respected TEFs to measure mentioned metrics.

III. Related Work

Task execution frameworks have been around for quite a long time and have played a major role in distributing the task. Centralized architecture where a Master is handling all the slaves is not appropriate for the demands of exascale computing, due to poor scalability and single point of failure. Also, Centralized architecture has moderate scalability. Distributed architecture, on the other hand, with the work stealing technique seems to be a better approach towards fine grained tasks.

MATRIX[1] a distributed task execution fabric which applies the adaptive work stealing techniques for distributed load balancing and a distributed key-value store for task metadata management. The work stealing from neighbors creates overhead and could lead to a bottleneck. The limitation with MATRIX is that, the dynamic mechanism of work stealing technique introduces some overhead when selecting neighbors.

Falkon[8] is a centralized task execution framework with a hierarchical task dispatching implementation. Because of its distributed architecture, MATRIX outperforms Falkon across the board for all workloads with different durations. Based on results of experiments performed, MATRIX maintains high throughputs and was 4 times more efficient than centralized system.

Sparrow[6] is a distributed task scheduler with multiple schedulers pushing tasks to workers. CloudKon[9] is a distributed task scheduler specific to the cloud environment. MATRIX also outperformed these two task schedulers and showed better performance due to its distributed nature of adaptive work stealing and near perfect load balancing when submitting tasks.

IV. Proposed Solution

A. MATRIX Bench:

MATRIX Bench is the benchmarking project. Here the main goal is to examine performance of mentioned five runtime systems and compare it with performance of MATRIX using performance metrics. So this is empirical performance evaluation project. All the five runtime systems are implemented in various programming technologies like Python, C++, different scheduling plugins, storage systems and different platforms. Major goals of proposed project are listed below:

i. To understand the architecture of all the mentioned TEFs and downloading and running each runtime system on the testbed for the sample program output will be the first phase of this project.
ii. Understanding different algorithms implemented for various TEFs to run MTC applications under same workloads and compare its throughput, latency and efficiency with MATRIX to improve its performance & extend it to the maximum system utilization will be the outcome of this project.
iii. Modification in the existing code to improve its performance after getting benchmarking results will be the core part of the solution for this project.
iv. MATRIX gives overhead on work stealing technique because of its architecture. The result of this project will help us extending Matrix by comparing it with different TEFs to execute number of fine grained tasks on runtime system.

All the systems that we benchmarked are installed on Linux Ubuntu 14.01.1 operating system. The proposed work for all the systems is mentioned below:

1. MATRIX:

After reading papers related to MATRIX and ZHT, we installed MATRIX on local machine. We referred GitHub[12] page for the same. For running MATRIX, there are some dependencies of Google protocol buffer C binding, Google protocol buffers C++ binding and ZHT distributed key-value store. It is required that Google protocol buffers C++ binding must be installed before installing Google protocol buffer C binding. All were installed successfully on local machine. To run MATRIX, first run ZHT server then run MATRIX server.

We made some changes in scheduler_stub.cpp code to assign sleep workloads.

2. Legion:

We read paper of Legion and understood the concept behind implementation of Legion. For installation, we referred Legion GitHub[13] page. Legion is installed on local machine and run successfully on single node. An implementation of the POSIX threads library is required for running all Legion applications. We need python 2.4 and C++ compiler to run the debugging tool.

3. Charm++:

We first studied the Charm++ system[14] and program structure and successfully executed sample programs of Charm++ on Local Machine. We also implemented sample queens program from Charm/examples/ charm++/queens. Also studied and explored the Nodelist file[15] structure of Charm++. Nodelist file is used to specify the addresses of nodes to connect with.

4. Swift:

Swift has the same script and code which runs on multicore computers, clusters, clouds, grids, and supercomputers. Swift is a fast, easy and flexible. To run swift Install python and 'libcloud' library. Since we are using an Ubuntu instance, first created a 'launchpad' virtual machine, to launch the 'headnode' and connected 'workernodes'. Setup the configuration file to make provide the information like number of worker images and type of image used and key. Swift script executes to create the 'headnode' and multiple worker nodes. We executed the Wordcount program and tested it across 32 worker roles. The results showed that swift has more throughput than hadoop.

5. YARN:

For running YARN scheduler of Hadoop system, openJdk is one of the required dependencies. First we installed Hadoop on single node. We then added small piece of commands in "bashrc" file to get identity of instance using eval 'ssh-agent' and ssh the .pem file. For running the yarn scheduler we changed four of the .xml files named core-site, hdfs-site, mapred-site and yarn-site to give the replication permission for multi nodes and to add the instance identity. All worked perfect on multi nodes by adding slave instance id in the slaves file of Master node and we were able to run the word count program of 10GB dataset on upto 64 nodes. Performance is well explained in evaluation part.

B. Problem Faced:

To run the legion on multi node, dependences were too high for the system we used. CUDA & different conduits for network like TCP we could not able to setup correctly. In yarn multi node system, problems we faced was sometimes data node were not starting in every slaves instances and that was stopping the workload from executing perfectly.

Running matrix system on multi node was really challenging. The .aws script we used to run system on AWS instances was after all the schedulers started, clients was unable to assign the workload to each scheduler after taking it from workloadfile.

The setup of Charm++ required build command depending on the type of operating System and processor. Followed the command given on guideline to build successfully. Also while executing program faced an error "Can't establish the connection". Another challenge faced is that we created 2 amazon instances on AWS and added the public address in nodelist file. But while executing it was asking for the public key pair. We tried inserting public key pair name in nodelist , but it showed an error -"Can't find public key".

But we solved it and now able to run particular system on multi node with defined workloads except legion.

V. EVALUATION

A. Metrics:

The metrics we will be using to evaluate the performance of the above mentioned systems are throughput, efficiency and average task scheduling latency.

i. **Throughput** is the measure by using which we can calculate how fast the system executes tasks. We will measure throughputs for all the mentioned TEFs in the unit of MB/sec.

ii. **Efficiency** is the extent to which time is well-used for the task execution. It is ratio between expected execution time to complete the task under given workload and the actual time taken by task to finish execution. The expected execution time is measured by multiplying the number of tasks per core with the average task execution time calculated.

iii. **Average scheduling latency** is the average of time that the system is unproductive because of the scheduling tasks. More the latency is less is the throughput. Hence, less latency means systems performance is better. We will observe latencies of all the systems and will decide which one gives best performance.

B. Workload:

Workload given to each system to determine throughput and latency is given below with the graphical representation:

1. MATRIX:

We cloned MATRIX code on local machine. To run MATRIX on single node, MATRIX membership list of scheduler is

specified. Also we specified the ZHT membership list of server. Number of ZHT messages were 17562. We changed config file to change number of tasks to 1000/node with length of sleep 0. Time required for running these tasks was about 11 seconds and throughput was 90.90 MB/sec. Then we scaled the system up to 64 nodes and ran the workload of 1000 tasks per node for sleep 0 tasks. Latency and Throughput dropped as the number of instances increased and reached to 23 seconds and 2782 Mb/s respectively. Performance we got is shown in graphs.

Workload:
We executed 1000 tasks/node with sleep 0 task on c3.large instance.

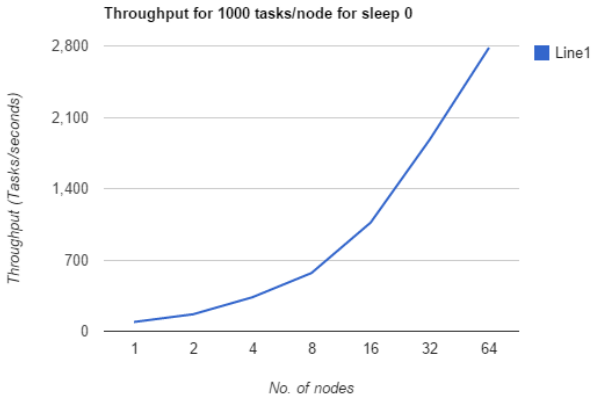

Fig 1: Throughput of MATRIX

Fig 2: Latency of MATRIX

2. Legion:
We changed .bashrc file to set LG_RT_DIR variable to the runtime directory to run program in legion. We then executed the sample "hello world" program. For this "make" is required to make sure that "legion.h" is included in the code. Sample code runs well on single node legion system.

Workload:
We ran sample program on single node with sleep 0, sleep 1, sleep 10 and sleep 100 tasks.

3. Charm++:
We executed the sample "queens" program[15] locally on a single node. The time of execution for this program on a single node is 0.65 s. In future, we added the addresses of ec2 instances in nodelist file and executed the program through ++nodegroup command to scale the system to 64 nodes. We ran the same queen program on multi node system and graphs we got are like shown in fig. time taken to run that program was decreasing and throughput was increasing as no. of instances increased.

Workload:
We executed sample Queen program with sleep 0 task on t2.small instance.

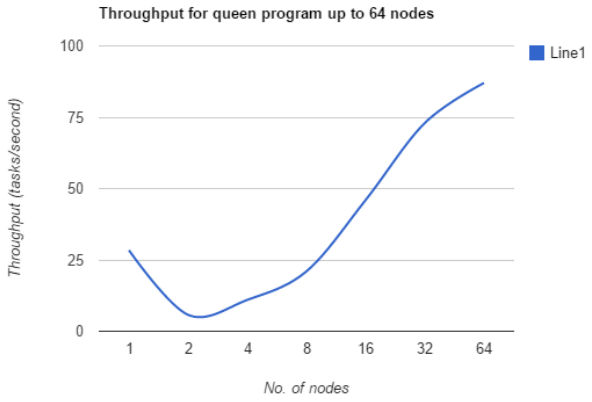

Fig 3: Throughput of Charm++

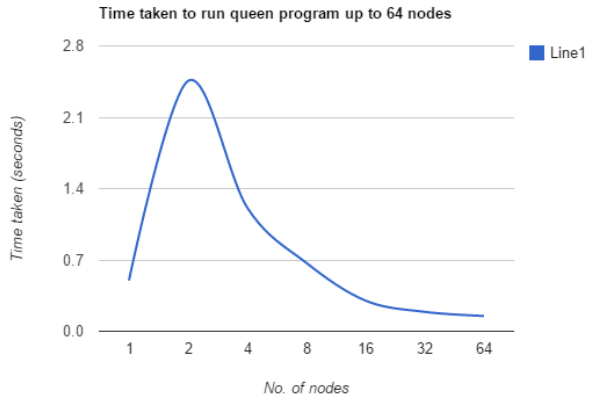

Fig 4: Latency of Charm++

4. Swift

Swift script executes to create the headnode and multiple worker nodes. We ran the same workload on swift multi node system as Hadoop and the results showed that swift has more throughput than Hadoop. To run the same workload the nature of the latency and throughput remained same as Hadoop but the numbers we got made us deciding that swift runs more efficiently.

Workload:

We executed 10GB dataset on Wordcount program on t2.micro instance.

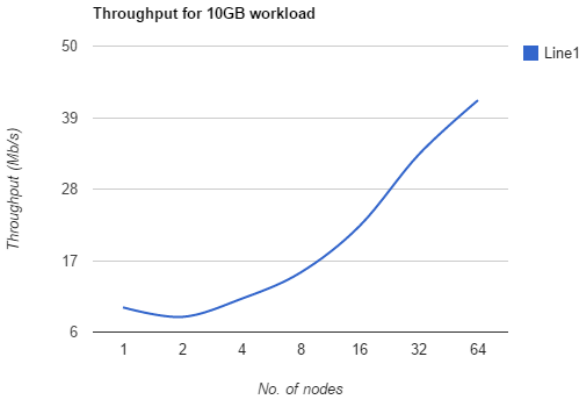

Fig 5: Throughput of Swift

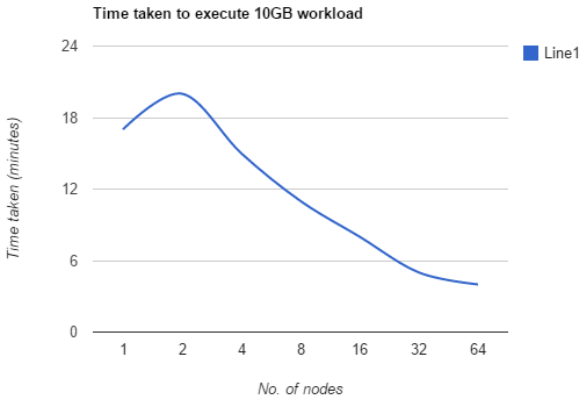

Fig 6: Latency of Swift

5. YARN

To run yarn on multi node, private IP of all the slaves are specified in slaves file of master node. We changed bashrc to remove steps to get identity of an instance every time we connect. Time taken to run the workload of 10GB dataset on word count program was decreasing continuous after a small hike for 2 nodes and throughput was increasing significantly up to 64 nodes. Multi node yarn system make it easy to handle large dataset to run efficiently.

Workload:

We executed 10GB dataset on Wordcount program on t2.micro instance.

Fig 7: Throughput of YARN

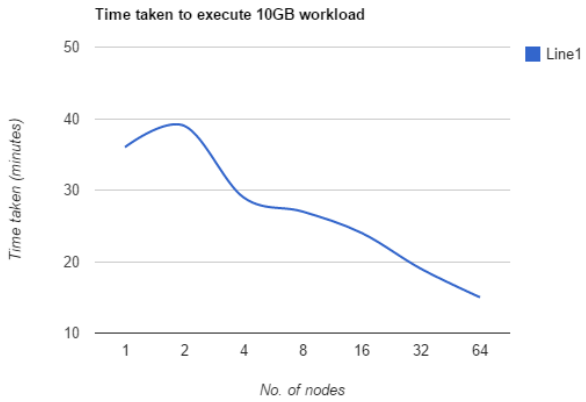

Fig 8: Latency of YARN

Thus, we have evaluated each system with different workloads and derived throughput and latency.

VI. Future Work:

We are planning to extend this work in future and will try to get proper output by giving proper and same workload to all systems. And by examining the results we can decide which system gives better performance in terms of throughput and efficiency. The results that we will get from our work, will help expand the work of MATRIX for better system utilization and overcome current limitations. As we have not executed same workload on each system, it is not possible at this time to compare each system's performance with one another.

VII. Conclusion

With the emergence of exascale system, applications are becoming more fine-grained in both task size and duration. For such exascale systems, task schedulers need to be distributed, scalable and efficient. MTC for such applications needs distributed scalable task scheduling frameworks to handle billions of jobs/tasks. Distributed architecture of TEF is required for the same. MATRIX has such distributed task scheduling framework. In our work we have compared task execution frameworks of Charm++, YARN, Legion and Swift with TEF of MATRIX to find out which one is better in sense of throughput, latency, and which one gives better efficiency. Based on partial results that we got from our work, we can say that MATRIX has better performance in terms of performance than that of other systems that we benchmarked.

VIII. References

[1] Ke Wang, Anupam Rajendran, Xiaobing Zhou, Kiran Ramamurthy, Iman Sadooghi, Michael Lang, Ioan Raicu. Distributed Load-Balancing with Adaptive Work Stealing for Many-Task Computing on Billion-Core Systems. Under preparation at Journal of ACM Transactions on Parallel Computing (TOPC), 2014

[2] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, and Jason Lowe, Hitesh Shah, Siddharth Seth, and Bikas Saha, Carlo Curino, Owen O'Malley and Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator [Industrial Paper]

[3] Michael Bauer, Sean Treichler, Elliott Slaughter, Alex Aiken. Legion: Expressing Locality and Independence with Logical Regions. International Conference on Supercomputing (SC 2012)

[4] Gengbin Zheng, Laxmikant Kale, Eric Bohm, Abhinav Bhatele, Chao Mei, Filippo Gioachin, Ramprasad Venkataraman, Yanhua Sun. Parallel Languages/Paradigms: Charm ++ - Parallel Objects. SC14

[5] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, Dietmar Fey, HPX – A Task Based Programming Model in a Global Address Space, PGAS 2014: The 8th International Conference on Partitioned Global Address Space Programming Models (2014)

[6] Kay Ousterhout, Patrick Wendell, Matei Zaharia, Ion Stoica. Sparrow: Distributed, Low Latency Scheduling.SOSP '13

[7] Ioan Raicu, Ian T. Foster, Yong Zhao. Many-Task Computing for Grids and Supercomputers. MTAGS 2008.

[8] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster and Mike Wilde.2007. Falkon: a Fast and Light-weight tasK executiON framework. SC 07 Proceedings of the 2007 ACM/IEEE conference on Supercomputing, Article No. 43. ACM
I. Sadooghi, S. Palur, et al. Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing, Proceedings of the International Symposium on Cluster Computing and the Grid (CCGRID), 2014.

[9] T. Li, X. Zhou, et. Al. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table, in Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS), 2013

[10] Amazon Elastic Compute Cloud (Amazon EC2), Amazon Web Services, [online] 2013, http://aws.amazon.com/ec2/

[11] MATRIX: a MAny-Task computing execution fabRIc at eXascale, https://github.com/kwangiit/matrix_v2

[12] Legion Language Compiler https://github.com/StanfordLegion/ legion/ blob/master/deprecated/compiler/README.md

[13] CHARM ++ installaton guide http://charm.cs.illinois.edu/manuals/html/charm++/A.html

[14] Charm++ execute programs http://charm.cs.illinois.edu/manuals/html/charm++/C.html

[15] Charm++ Download link http://charm.cs.illinois.edu/software

[16] Justin M. Wozniak, Timothy G. Armstrong, Michael Wilde, Daniel S. Katz, Ewing Lusk, Ian T. Foster. Swift/T: Large-scale Application Composition via Distributed-memory Dataflow Processing.