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

Thomas Dubucq, Tony Forlini, Virgile Landeiro Dos Reis, and Isabelle Santos

Illinois Institute of Technology, Chicago, IL, USA
{tdubucq, tforlini, vlandeir, isantos1}@hawk.iit.edu

***Abstract*— Technology trends indicate that exascale systems will have billion-way parallelism, and each node will have about three orders of magnitude more intra-node parallelism than today's peta-scale systems. The majority of current runtime systems focus a great deal of effort on optimizing the inter-node parallelism by maximizing the bandwidth and minimizing the latency of the use of interconnection networks and storage, but suffer from the lack of scalable solutions to expose the intra-node parallelism. Many-task computing (MTC) is a distributed fine-grained paradigm that aims to address the challenges of managing parallelism and locality of exascale systems. MTC applications are typically structured as direct acyclic graphs of loosely coupled short tasks with explicit input/output data dependencies. The task execution framework of MTC has a distributed design that focuses on hiding execution latency and achieving good load balancing in the face of large number of compute resources, storage, and number of tasks. Runtime systems with an asynchronous MTC task execution model include MATRIX, Charm++, Legion, HPX, STAPL, Swift/T, Hadoop's YARN and Spark/Sparrow. This project aims to benchmark these runtime systems with an emphasis on the task execution framework, using MTC workloads of different data dependency patterns. The goal is to understand the performance, scalability, and efficiency of these runtime systems in scheduling different data-intensive workloads with a variety of task granularity.**

***Index Terms*—benchmark, data intensive computing, exascale, many-task computing, MATRIX, MTC**

I. Background Information

THE eight systems that we are about to benchmark are all open-source projects. Most of them are academics research projects but YARN which is the Hadoop resource manager component is a professional project. This system allows to run distributes workloads and applications, often following the Map-Reduce paradigm. Swift is a workflow system which allows to run parallel applications with its parallel programing model. This project is developed by University of Chicago and ANL. Spark is a project from UC Berkeley encompassed by the Sparrow project which is a large scale task scheduler system with in-memory queries system. Charm++ is a distributed parallel programming paradigm from UIUC and Legion is a data-centric parallel programming system developed by Stanford University. Finally HPX is a general purpose C++ runtime system for parallel and distributed applications of any scale developed by Louisiana State University and Staple is a framework for developing parallel programs from Texas A&M.

MATRIX is a many-task computing job scheduling system [3]. There are many resource managing systems aimed towards data-intensive applications. Furthermore, distributed task scheduling in many-task computing is a problem that has been considered by many research teams. In particular, Charm++ [4], Legion [5], Swift [6], [10], Spark [1][2], HPX [12], STAPL [13] and MATRIX [11] offer solutions to this problem and have each separately been benchmarked with various metrics.

The main difficulty of the project is to grasp the complexity of each runtime system and figure out the scheduling part of the system in order to be able to measure the same metrics on every system. The main goal of this benchmark is to draw a comparison of the scaling performances of these scheduling systems while testing it on different types of workloads. As a matter of fact, it is often difficult to make sure that we run the benchmarks the exact same way on different systems. That means that we have to make sure that we only take in account the relevant metrics and measures relative to our study. Thus we need to produce similar workloads on every of these runtime systems which can imply having similar DAG decomposition for the application on which we want to measure the benchmark. In some cases runtime systems don't handle the decomposition of the application in DAGs. Thus we will have to produce this DAGs on our own. Finding the right testbeds is also one of the tremendous challenge in doing benchmark of different runtime systems since we ought to find MATRIX benefits over concurrent systems.

II. Proposed Solution

This project will compare seven scheduling systems through benchmarks to evaluate throughput and efficiency. In order to perform an accurate comparison, a detailed performance profile will be done for each system with the same workload. Different workloads of fine granularity and intensity will be used, as well as different scales up to 128 nodes. The benchmarks for each system will be performed using Amazon Web Services (AWS). For each of these runtime systems we will need to follow installation guidelines and tutorials in order to grasp the

architecture and dependencies of these programs. Once these programs are installed and running on one instance, we will need to set up and automate as far as possible the deployment of the application on several node in our Amazon cluster. Some of these systems may require the same underneath running systems such as Open MPI for message passing interface or GASNet for network high performance communication.

1) Charm++

The Charm ++ system is made of a parallel programming paradigm and a runtime system. In this section we will explain the runtime system architecture and how the charm++ programming paradigm allows us to write applications decomposed as Direct Acyclic Graphs. Charm++ programs use objects called chares to represent the entities responsible for launching the different tasks of the application. Thus the parallelism is not implicit as it is in other runtime systems such as Swift.

As a matter of fact, one needs to explicitly write the dependencies between the different chares. Furthermore, those chare objects can communicate through asynchronous message passing. Those messages don't interfere with the running state of a chare hence the asynchronous design. Each chare has an associated state and belongs to the global object space which keeps track of all the tasks to execute. Finally the programmer doesn't need to consider the number or processors on which to run the tasks or the type of interconnect for the message passing, one only needs to specify how chare objects interact with each other.

We built the Charm++ runtime system on a vanilla Ubuntu OS within a c3.large EC2 instance. All the nodes on which the application will run need to be configured to allow SSH connection between each other and to specify the list of all the nodes within the nodelist configuration file. Our first test application is a simple bag of sleep tasks which simply waits for a given amount of time. The code for this application basically consists in a parallel for loop going through a chare array which launches the sleep tasks in a distributed way on the different nodes. Then the arguments for this application are the number of seconds to wait and the number of nodes on which to run the application.

We set up a timing of each task and we write it into a log file in order to reduce the overhead of a display on the standard output. Finally a python script reads through the file to compute the total latency of the application. We also compute the total running time in order to be able to know the communication overhead we subtracting the latency from the total running time.

2) Legion

The Legion runtime system has a similar architecture. The different tasks are represented by a function which is called by a TaskLauncher object. Unlike for Charm++, the parallelism does not have to be explicit. The way the different tasks are interacting defines the Direct Acyclic Graph of the application. Finally a top level task is responsible for launching all the subsequent tasks of the application. One can specify leaf tasks in order to optimize the running time of the application since those tasks have no data dependencies with other tasks.

However the Legion runtime system requires underlying systems which are GASNet for network interconnect and message passing and CUDA because GASNet requires GPU enabled configuration to be able to run. Nevertheless we don't plan on running the benchmark on GPU instances so far. The GASnet application needs to be built with the TCP/UDP so called "conduit" to use the classic network communication. One might also set up MPI as the underlying communication conduit.

Moreover the application follows the same previous pattern with a parallel for loop of sleep tasks. The different nodes on which to run the application are defined within the shell environment with an environment variable specifying the IP addresses of each node. The same python script is responsible for retrieving the latency of each task and summing it up to get the total latency. The throughput is computed by taking the number of task executed per second.

Finally, benchmarking dynamic runtime systems such as Legion is quite tricky to do correctly. In particular, some benchmarks which are otherwise appealing, such as launching a bunch of empty or nearly empty tasks in order to measure runtime overhead are especially poor predictors of Legion performance. This is because the Legion runtime operates in a pipeline and goes to quite a bit of effort to ensure that analysis stays off the critical path.

As a result, to get real insight into the Legion system, one would need to take the runtime architecture into account when designing the benchmarks. Since Legion's feature set diverges so widely from other task based runtimes it is not clear whether it is even possible to perform an apples-to-apples comparison with even the closest comparable runtimes.

3) Swift/T

The objective of this project is to be able to benchmark several systems on the same hardware: EC2 instances. As Swift/T has only a very limited documentation to deploy on Amazon Web Services, there has been different challenges when installing and configuring Swift/T so it can run on several EC2 nodes.

Our first step has been to install and configure correctly Swift/T to run on a local machine. This has been done only with one limitation. The Swift/T documentation indicates that either OpenMPI or MPICH can be used but when configuring it with OpenMPI, it does not run as it seems that some shared libraries are not in the right folder. We did not look more into details for this issue as Swift/T was running correctly using MPICH. We also noticed that having both MPI implementations on a system lead to a more complicated configuration and most of the time it is easier to keep only MPICH to make Swift/T run smoothly.

Once Swift/T had been running on a local machine, we followed the same process to install it on an EC2 instance and created an AMI image so we can launch several instances with Swift installed. Once again, we used MPICH.

The part of the Swift/T installation that has been the most time consuming is the deployment of Swift/T on several instances (i.e. run a Swift/T program in a distributed fashion).

First of all, it is indicated in the documentation that compute nodes need to be able to SSH between them. To allow this, we have created a private key that is distributed through all the compute nodes and we modify the SSH configuration on every compute node to use this private key. Note that the private key and the SSH configuration file are not stored in the AMI image

but are distributed from a local machine to the EC2 nodes using a python script.

We also need to create a hostfile that contains all the IP addresses of the workers and that is used by MPI: this is also generated by the python script and distributed over all the workers. This python script has been developed using the boto module for EC2. Finally, to run a program on several machine, we compile it using STC and send the *.tic file to all the workers. Then we can connect to a worker and run the following command:

```
export TURBINE_LAUNCH_OPTIONS="-f=hosts.txt"
turbine program.tic
```

Justin M. Wozniak has been kind enough to answer our questions about Swift/T and how to run it on several EC2 instances once the program has been compiled using STC. The problem encountered and that took us several days to debug is that there has been a change in the environment variable name from TURBINE_LAUNCH_OPTS (December 2014) to TURBINE_LAUNCH_OPTIONS (March 2015) but the documentation has not been updated except in one place (http://swift-lang.org/Swift-T/guide.html#_concurrency).

Neither the guide related to Swift/T on EC2 (http://www.mcs.anl.gov/exm/local/guides/turbine-sites.html#_ec2) nor the program documentation (turbine –h) have been updated to this day. However, once we have noticed this problem, we have been able to run benchmarks without any issue.

In order to simplify the benchmarking process, we have written a small python API based on boto that allows a user to launch EC2 instances, create automatically the configuration files needed for Swift/T and copy them to all the running instances, deploy a given Swift script with possible arguments. Using this API and a JSON file describing the benchmarks we want to run, we are able to run every benchmark and get back the results in one command line.

4) Sparrow

The source code of Sparrow provides an example of sleeping benchmark but unfortunately it lacks several features required for our project. Indeed their benchmark consists in requesting the scheduling of bags of tasks of random length sleeps at a regular interval. No acknowledgement of task ending was made and the different timings were only prompted in the debugger log. Therefore we needed to code our own Frontend and Backend applications. The following paragraphs will detail how the implementation of these two applications answer to the specifications of our benchmark.

First of all we needed to have a total time measurement for individual tasks and for the whole benchmark. The implemented solution is a TCP server/clients. The Frontend application runs a multithreaded TCP server while each Backend application connects to it and send acknowledgement messages. Each Backend has a dedicated thread on the server, whose only job is to store the message arrival time and the message content in a thread safe fifo queue that is then processed by the main thread of the Frontend. On the Backend side, messages are batched to reduce network congestion. These messages contain the number of finished tasks, individual relative delay compared to the earliest task finished and total batching time. These recorded delays are then subtracted on the Frontend side to get the most accurate timing.

We also needed to log the local time spend during each phases of the scheduling. That feature has been added in the Frontend and Backend applications and also in the Sparrow node monitor -which is responsible for the assignment of a task to a particular backend-. The results are stored in a csv text file. Both The Frontend and the Sparrow node monitor have their own file while the backend uses a thread safe class to store its results -indeed, since the execution of requested tasks is multithreaded, a thread safe solution was required-.

Unfortunately there is no possible way to stop the processes required in the benchmark in the Frontend code -the Sparrow API launches threads that you don’t have handles on- so we needed to use external scripts to automatize the benchmarking. The following paragraphs will detail the python scripts we coded.

Two scripts are in charge of the configuration files necessary for the whole benchmark. One creates the configuration files of both the Frontend and Backend application and the other modifies the Sparrow nodemonitor configuration file -adds new nodes’ IP addresses to an existing configuration or sets a new configuration file-.

One script is responsible of launching and terminating AWS instances and another allows to launch and kill Backend applications and nodemonitor applications locally on a node.

We also coded a script that wraps all the SSH commands necessary to the benchmark -start/kill the processes on node, set the configuration files on a new node, update the configurations on an already working node and fetch the results on the nodes-

Finally a script wraps all of the previous scripts to launch a series of benchmarks with different settings.

5) HPX

HPX is a general purpose C++ runtime system for parallel and distributed applications developed by the STE||AR Group at Louisiana State University. By a runtime system, we mean that any application that uses HPX will be directly linked to its libraries. This library adheres to the C++11 standard and uses the Boost C++ libraries. HPX is an open-source implementation of the ParalleX (Kaiser, Hartmut, Maciej Brodowicz, and Thomas Sterling. "Parallex an advanced parallel execution model for scaling-impaired applications." Parallel Processing Workshops, 2009. ICPPW'09. International Conference on. IEEE, 2009) theoretical execution model.

Using the HPX runtime system, parallel applications use futures. A future can be described as a value that exists now or will be produced in the future. In other terms, a future encapsulates a delayed computation. It acts as a proxy for a result initially not known, most of the time because the computation of the result has not completed yet.

Typically a function is used to produce the value of a future. This producer function will be executed asynchronously in a new HPX thread. Once the action has finished executing, a write operation is performed on the future. When the result of a delayed computation is needed, the future is read, or the reader thread is suspended until the future is ready.

Help and documentation for HPX can be obtained on the STE||AR Group website, on the IRC #ste||ar channel on freenode, or through the HPX users mailing list.

Benchmarks of HPX have been done by the STE||AR Group (Raj, Rekha. Performance Analysis with HPX. Diss. Louisiana State University, 2014) regarding rate of execution, thread idle-rate and task length among others on up to 16 cores of one node. We have started by installing and running HPX on a local machine. The first step was to install all of the dependencies for HPX, and testing them to ensure that they were properly functioning. The dependencies are the Boost C++ libraries, the Portable Hardware Locality (HWLOC) library, CMake, the google-perftools development files, and libunwind, a dependency of google-perftools. To ensure a working Boost installation, we compiled Boost from sources ourselves. Some of the dependencies listed in the HPX documentation were ambiguous. For instance the documentation only indicates that google-perftools is needed, however, libgoogle-perftools-dev is also required.

Furthermore, regarding the dependencies, not all compatible versions are compatible amongst themselves. For instance, Boost V1.49.0 and later are recommended, but Boost V1.56.0 can't be used for HPX with gcc V4.6.x, so we had to double check the version numbers for all dependencies.

Finally, the C++ HDF5 libraries must be compiled with enabled threadsafety support. This has to be explicitly specified while configuring the HDF5 libraries as it is not the default.

The next step was to build HPX. HPX requires an out-of-tree build. In other terms, HPX must be built in a directory separate from the source directory. For this build, it was necessary to specify the location of the Boost installation previously mentioned. Once the build is complete, HPX is delivered with a testing utility. The following step was to run an elementary program on a local machine using HPX. The documentation for HPX states that compiling and linking HPX needs approximately 2GB of memory per parallel process to be available. However, to compile the hello_world example with a single parallel process, that was not enough and we had to use a 4GB swap file in order to double our amount of memory. The hello_world program prints out a hello world message on every OS-thread on every locality.

The subsequent step is to write verbose sleep tasks. In order to write an application which uses services from the HPX runtime system it is necessary to initialize the HPX library by inserting certain calls into the code of the application. This can be done by including the file hpx/hpx_main.hpp, in which case the main() function will be the HPX entry point, or by including the file hpx/hpx_init.hpp and providing an hpx_main function that will be invoked at the specified entry point for HPX.

Writing these tasks has had its own set of difficulties. For instance, to build HPX components, the documentation states that the following command may be used to compile a sample hello world program.

```
c++ --o hello_world hello_world_component.cpp
`pkg-config ---cflags ---libs -hpx_component` --
DHPX_COMPONENT_NAME=hello_world
```

In fact, to compile the functions using HPX, I had to remove the excess '-'s and add the option -lboost_program_options to the command.

6) STAPL

STAPL (Standard Adaptive Parallel Library) is a parallel C++ library developed at Texas A&M University. There are a lot of publications concerning this project from 1998 to 2015 but unfortunately, the code is nowhere to be found on the internet. We tried to contact the developing team by emailing stapl-support@tamu.edu and by directly emailing one of the creator of STAPL (Lawrence Rauchwerger) but we did not get any response from them. Therefore, we have not been able to make any progress on this part of the project.

7) MATRIX

MATRIX is a fully-distributed task execution framework for Many-Task Computing data intensive applications developed at the IIT. MATRIX delegates a scheduler on each compute node to manage local resources and schedule tasks. MATRIX is used on top of ZHT -Zero Hop distributed hashTable, also developed at the IIT- which manages task metadata. MATRIX implements a data-aware work stealing technique to optimize both load balancing and data-locality. Matrix is used to launch pre-compile sleep programs, already existing on each node.

Python scripts are used to launch instances, run benchmarks and grab the logs of every instance. The results are processed with the tools supplied in the dataproc folder.

Minor issues were encountered during the deployment and running of MATRIX. First, slight errors in the configuration file -e.g. wrong folder in a path- entails segmentation fault at launch without any other information, which took some time to fix. The launching of a scheduler must be done after all ZHT have been started on all instances, which caused some trouble for higher scales.

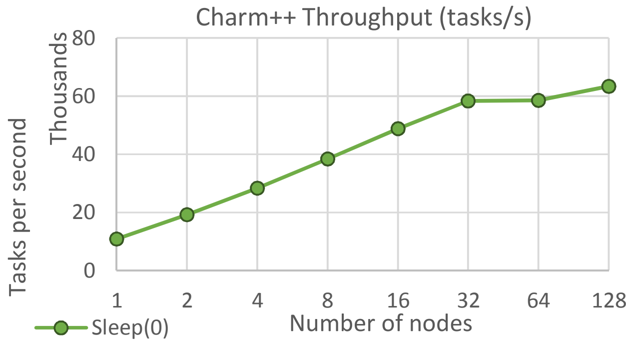
III. Evaluation

To evaluate the performance for each of these projects, we will compute the throughput, the latency and the efficiency with workloads of varying sizes. We will also scale every evaluated system from 1 node to 128 nodes in order to evaluate the scalability. The latency of a system is defined as the average time period between the issue of a request and the beginning of the system's response to this request. The lower the latency the better the performance of the system. We will measure the average latency by issuing a very large amount of small request) and by recording the latency time for each request. The throughput of a system is the number of request that it can handle per second of running time. It should be kept as high as possible. We will measure it by submitting several workloads of varying size to the evaluated systems and by computing the average number of requests executed per second. The efficiency of a system is defined as the ratio of the effective throughput over the theoretical throughput.

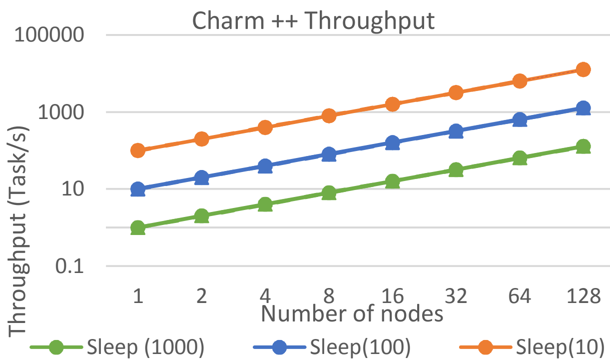
1) Charm++

As previously said, the experiments has been performed on C3.large Amazon EC2 instances with an dedicated AMI for each runtime system. Python scripts are responsible for automating the launch of instances following the scaling from 1 to 128 nodes. Other python scripts are responsible for

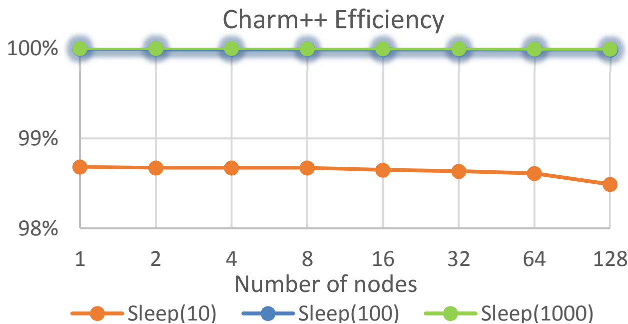
automating the variation of experiments variables such as the sleep time and the number of tasks within the bag of tasks.
We first compute the throughput which is a relevant metric for Sleep(0) tasks which do nothing on purpose. Here are the results we found for Charm++.



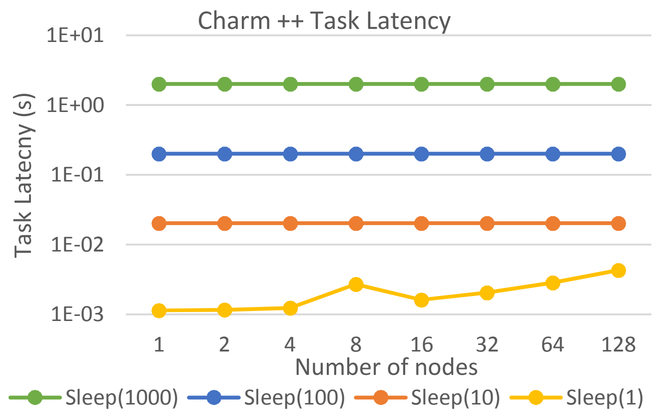
Charm++ seems to give us satisfying results with an increasing throughput as the number of nodes increases. The maximal value reached for the throughput is about 63K tasks per second. Then we computed the throughput and efficiency for tasks from 10ms to 1 second.



First, we didn't plot the throughput for 1ms tasks because we obtained wrong results for these measures since the total execution time was below the ideal execution time. Most likely some tasks have been dropped because the number of tasks to execute was too high (~12 billion tasks). For the other sleep tasks, we find an increasing throughput up to more than 10000 tasks per second on 128 nodes which is a satisfying number.



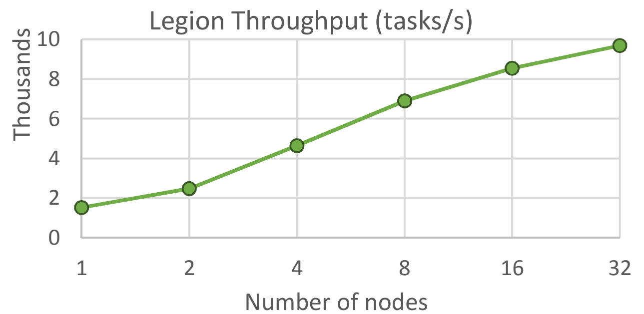
Concerning the efficiency, it is almost 100% efficient for both sleep(10) ms and sleep(100) ms. Then it falls nearly blow 99% efficiency for sleep(1) sec which shows that Charm++ gives excellent results and good scaling for this benchmark.
Finally we plotted the task latency for all kind of non-null sleep for all number of nodes:



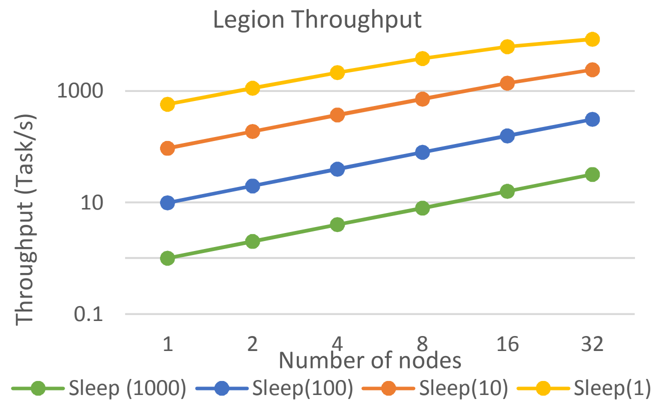
As expected, the task latency is steady as we increase the number of nodes on which we run the application, except for fine grain tasks.

2) *Legion*

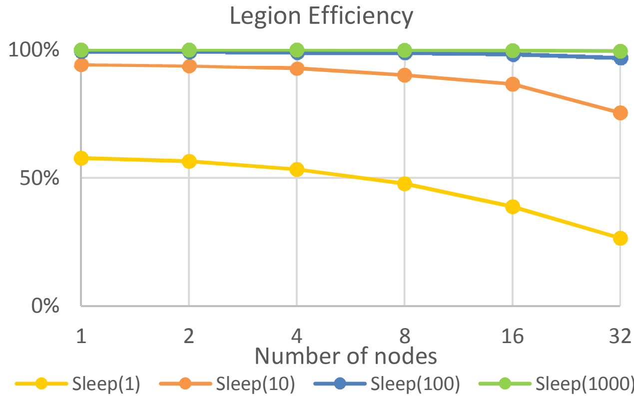
We launched the same workload with the same kind of application with Legion runtime system. Unfortunately, the legion system is not built to run on heavy clusters. As a consequence, the implementation of the runtime system limits the number of machines on which to run applications to 32. Thus we performed our benchmark up to this limit but weren't able to scale up to 128 nodes.



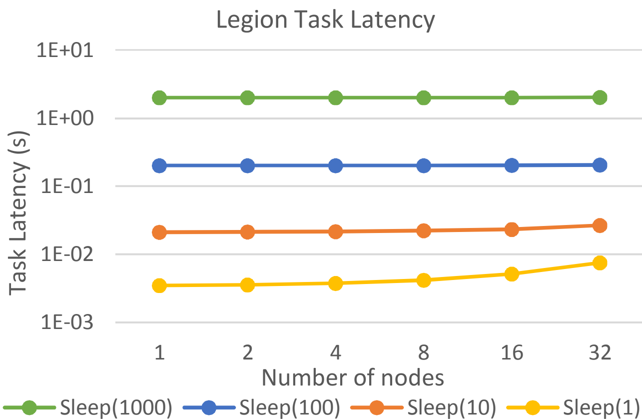
We found that the throughput for sleep(0) tasks is increasing which indicates that the system scales for 32 nodes. On the other hand the throughput for 32 nodes is about 10000 tasks par seconds which is way below the performance of charm++ but above other systems performance.



As expected the throughput is increasing with the scale. The maximum throughput is reached for sleep(1) ms for 128 nodes with about 10000 tasks per second. We also found that bigger task have a doubling throughput with the number of nodes, but sleep(1) ms tasks tend to reach an upper bound when the scale increases.



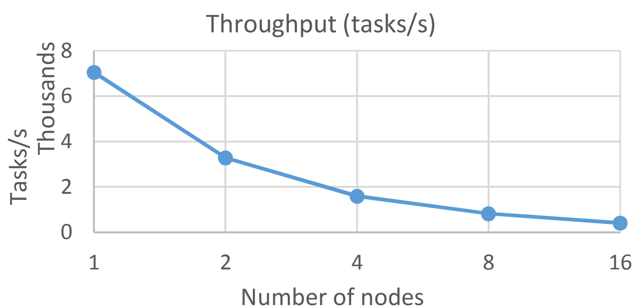
As for Charm++, we found that coarse grained tasks show very good efficiency and are also scaling. On the other hand smaller tasks such as sleep(10ms) or sleep(1ms) show poorer efficiency, with a lower bound of 28% for sleep(1ms) for 32 nodes. That means that the scheduling and communication overhead introduced for fine grained tasks is slower that the task duration itself. Thus the cost of scheduling is higher for such tasks which might not worth such deployment.

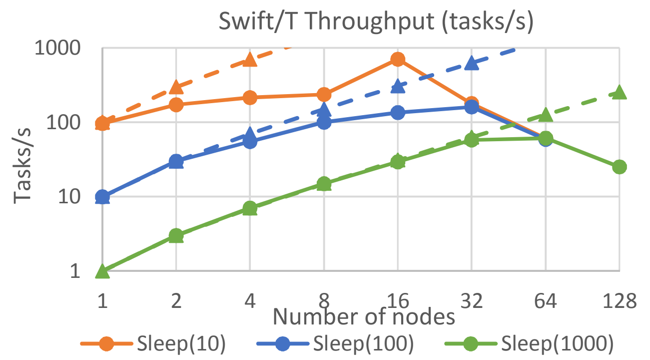


As for Charm++, the task latency is steady with the scale but not for sleep(1) ms tasks.

3) Swift/T

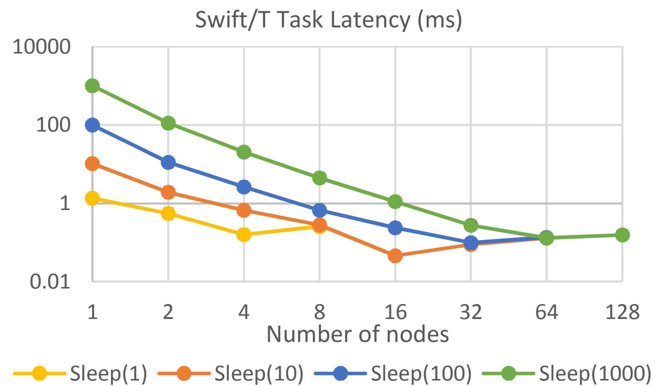
The benchmark results for Swift/T are far from being what is expected from this system. We started by running sleep(0) tasks



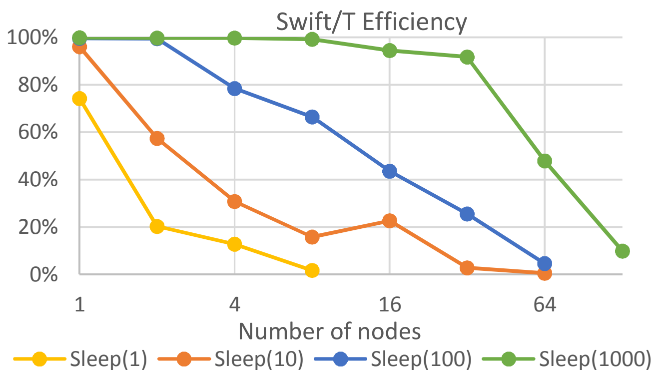


in order to benchmark the system throughput and realized that the system simply does not scale for these tasks.

Instead of being multiply by two when multiplying the number of nodes by two, the throughput is actually divided by two. This is clearly a configuration problem but discussions with two Swift developers - Justin Wozniak and Timothy Armstrong – did not change this situation. To convince ourselves that this was not due to an error from our side, we ran different types of tasks (1ms to 1000ms) and looked at the throughput, efficiency, and task latency metrics.



For the throughput, we notice that the sleep(1000) tasks are the tasks that scale the best as the computed throughput (solid line) is close to the ideal throughput (dashed line) up to 32 nodes. Once we reach the scale of 32 nodes, then the throughput stabilizes and the decreases when benchmarking on a cluster of 128 nodes. We observe the same trend for sub-second tasks except that the smaller the task length, the faster the computed throughput moves away from the ideal throughput when scaling up. One bottleneck could be the load balancing server as only one load balancing server is used in this benchmark. We have tried scaling up dynamically the number of load balancing

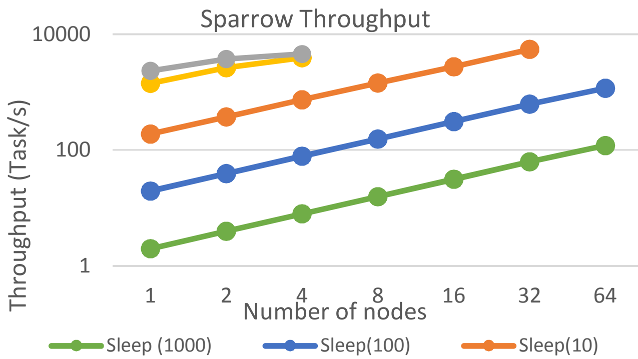


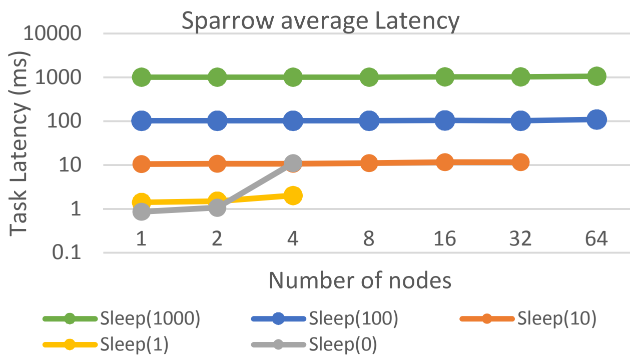
servers and it did not seem to affect the results. However, we think this possible solution should be evaluated again and this could a part of the future work on this project.

In a similar fashion, we observe that the computed efficiency and task latency get very poor very fast (starting at 2 nodes) for sub-second tasks. However, the efficiency stays correct up to 32 nodes for the one second tasks with a value of 92%.

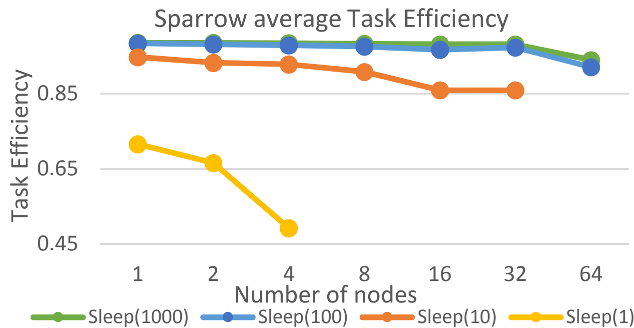
4) *Sparrow*

We ran the benchmarks with a single Frontend and up to 64 nodes. Unfortunately, high number of tasks submissions from a single point started to cause troubles when the benchmarks required more than 10,000 tasks. After this point, we noticed inconsistency in task identification and task duplications –more tasks would be registered as finished than the number of tasks submitted-. This situation would continue up to a certain point, then with the number of submission still increasing, we blocked on runtime errors. We realized that a single submission point causes a bottleneck as the rest of the system is scaling.





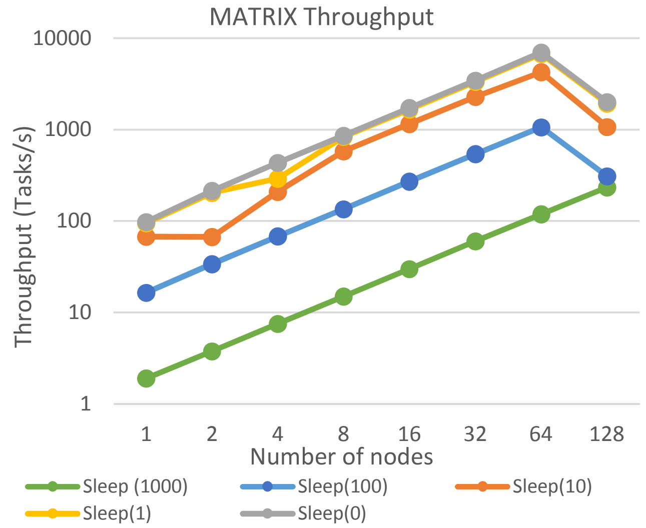
For the configurations which we were able to benchmark show good scalability. While coarser tasks scale linearly finer task seem to reach an upper bound but this bound could due to the single-Frontend bottleneck.



Sparrow latency is extremely low. A close look to the number shows that it actually increases at higher scales, but that increased delay could again be attributed to the Frontend bottleneck. This is easier to see on the Efficiency graph below.

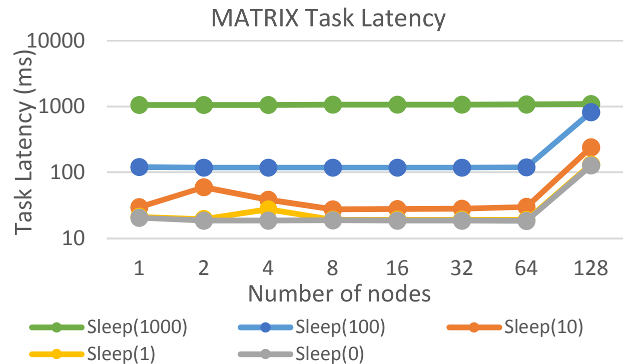
5) *MATRIX*

MATRIX was configured to create a DAG of type bag of tasks.

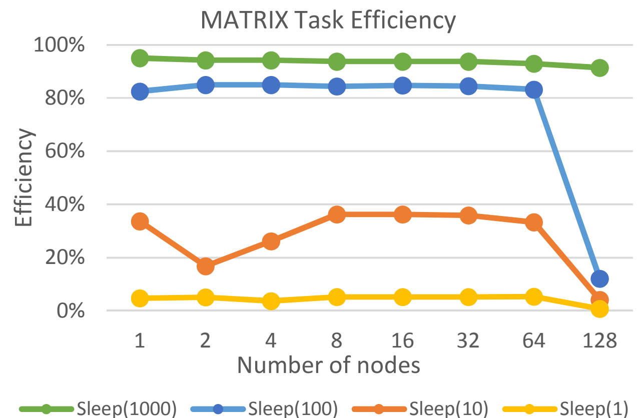


MATRIX throughput is scaling up to 64 nodes, after which throughput for all sleeps except sleep(1000) drops. As for Sparrow, this might be due to the client bottleneck.

As for the throughput, latency shows anomalies for the 128



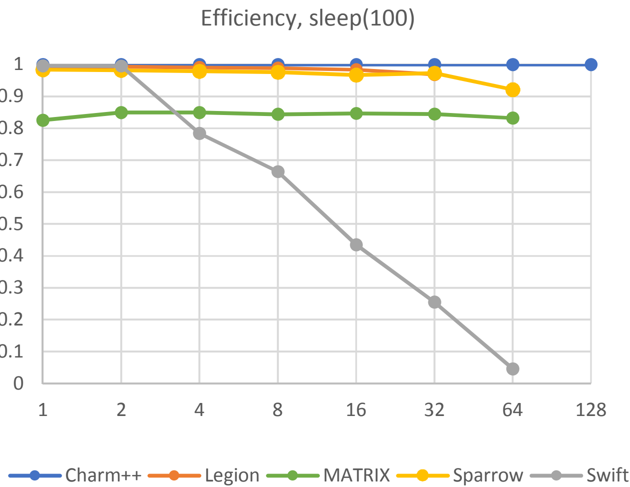
nodes scale. We can also notice that MATRIX has a low bound for its latency of 10-20ms. This can be explained by the dependency checking for each task when it is not needed for the benchmarks we ran.



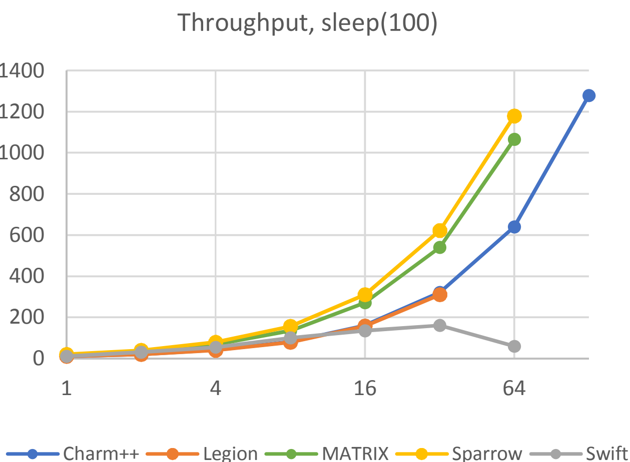
The efficiency graph shows that MATRIX has more trouble with the finer grain workloads while having good efficiency for coarser grain workloads.

6) Summary

We chose to use the sleep(100) benchmark as a comparison point as it was the one for which we had the more data.



Charm++, Legion and Sparrow have near perfect efficiency even at the higher scales of our benchmark. MATRIX has a good efficiency, and stay constant over the benchmark. Its lower results could be due to systematic task dependency checking. Swift efficiency crashes after 2 nodes.



All system benchmarked except Swift show scaling throughput. Indeed, each system throughput roughly doubles when the scale is doubled. Nevertheless two groups seem to emerge: the 10 tasks per sec per node-Charm ++ and Legion- and the 20 tasks per sec per node -MATRIX and Sparrow-.

Here is a table which sums up the workloads for every runtime system:

System	Per task duration (ms)	Expected time (s)
Swift/T	1 10 100 1000	3 3 30 100
Charm++	1/10/100/1000	100
Legion	1/10/100/1000	10
Sparrow	1/10/100/1000	25
MATRIX	1/10/100/1000	25

IV. Related Work

MATRIX is a many-task computing job scheduling system [3]. There are many resource managing systems aimed towards data-intensive applications. Furthermore, distributed task scheduling in many-task computing is a problem that has been considered by many research teams. In particular, Charm++ [4], Legion [5], Swift [6], Hadoop YARN [7], Spark [1][2], HPX [9], STAPL [10] and MATRIX [8] offer solutions to this problem and have each separately been benchmarked with various metrics.

V. Conclusion

Many research teams tackled the distributed task-scheduling subject recently. Each team has come up with their own system and compared it to the systems the industry is currently using. Unfortunately the results each of them separately presented does not give as is an overall view of the different systems performance against each other. The testing environment, workload, and worker’s specification they used were different, making a direct comparison of all systems difficult. This project aims to use a single testing environment, test these systems on the same workload, and use the same metrics each one of them to provide a fair comparison of the systems studied.

As a future work, we would like to run workloads decomposed in different kind of direct acyclic graphs in order to test a large variety of possible application performance. Those DAGs could be for instance set in fan-in, fan-out of parallel architecture which simulates different application workloads. Finally to simulate intensive applications with a lot of dependencies we could look at the traces of such applications and simulate the corresponding programs of our own.

VI. References

[1] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

[2] Ousterhout, Kay, et al. "Sparrow: distributed, low latency scheduling." *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013.
[3] Rajendran, Anupam. *MATRIX: Many-Task Computing Execution Fabric for Extreme Scales.* Diss. Illinois Institute of Technology, 2013.
[4] Zheng, Gengbin, Lixia Shi, and Laxmikant V. Kalé. "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI." *Cluster Computing, 2004 IEEE International Conference on*. IEEE, 2004.
[5] Bauer, Michael, et al. "Legion: expressing locality and independence with logical regions." *Proceedings of the international conference on high performance computing, networking, storage and analysis.* IEEE Computer Society Press, 2012.
[6] Wozniak, Justin M., et al. "Turbine: A distributed-memory dataflow engine for high performance many-task applications." *Fundamenta Informaticae* 128.3 (2013): 337-366.
[7] Wozniak, Justin M., et al. "Swift/T: Large-scale Application Composition via Distributed-memory Dataflow Processing." *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 2013.
[8] Phillips, James C., et al. "Petascale Tcl with NAMD, VMD, and Swift/T." *Proceedings of the 1st First Workshop for High Performance Technical Computing in Dynamic Languages*. IEEE Press, 2014.
[9] Wozniak, Justin M., et al. "A model for tracing and debugging large-scale task-parallel programs with MPE." *Proc. LASH-C at PPoPP* (2013).
[10] Vavilapalli, Vinod Kumar, et al. "Apache hadoop yarn: Yet another resource negotiator." *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013.
[11] Wang, Ke, Kevin Brandstatter, and Ioan Raicu. "Simmatrix: Simulator for many-task computing execution fabric at exascale." *Proceedings of the High Performance Computing Symposium*. Society for Computer Simulation International, 2013.
[12] Kaiser, Hartmut, Maciej Brodowicz, and Thomas Sterling. "Parallex an advanced parallel execution model for scaling-impaired applications." *Parallel Processing Workshops, 2009. ICPPW'09*. International Conference on. IEEE, 2009.
[13] An, Ping, et al. "STAPL: An adaptive, generic parallel C++ library." *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 2003. 193-208.

APPENDIX

As this project is a benchmarking of several systems, distributing the work between team members has been easy:

- Thomas Dubucq benchmarked the systems MATRIX and Sparrow;
- Tony Forlini benchmarked the systems Charm++ and Legion;
- Virgile Landeiro Dos Reis benchmarked Swift/T;
- Isabelle Santos benchmarked HPX.