Harvesting the computational power of heterogeneous clusters to accelerate seismic processing

Caian Benedicto* (CEPETRO/UNICAMP), Ian Liu Rodrigues (CEPETRO/UNICAMP), Martin Tygel (IMECC/UNICAMP), Mauricio Breternitz Jr. (AMD), and Edson Borin (IC/UNICAMP)

Abstract

Cluster environments are crucial to modern geophysics. Major processing companies make use of one or more computational environments, whether they be in-house clusters or third-party public clouds, to guarantee the efficient execution of their processing flows. But the diversification of such environments created a demand for software tools that are able to scale with efficiency in these ever-increasing ecosystems. Aside from efficiency requirements, these tools must also be able to handle and recover automatically from the faults that arise from these new and complex ecosystems. In this paper, we discuss how we leverage the Scalable Partially Idempotent Tasks System (SPITS) programming model and the PY-PITS runtime system to efficiently harvest the computing power of heterogeneous systems in order to solve geophysics problems. We also present an experiment in which we combine the computational resources from several clusters and workstations simultaneously to perform the regularization of seismic data and demonstrate the scalability and robustness of the system.

Introduction

Distributed and parallel computing are in the heart of modern geophysics problems, in such a way that it is not uncommon for large companies to have more than one cluster (Huang et al., 2016). That's because the demand for computing power only grows over time, as datasets become larger, and newer methods become more complex and time-consuming, such as Reverse Time Migration (RTM) and Full Wave Inversion (FWI). As specific tools become part of different strategies for processing different datasets, clusters get allocated for different sets of tasks and tailored to provide the maximum efficiency possible given the tools that are commonly used, this includes adjusting jobs policies, such as the time limit a job is allowed to run.

Along with the computational power provided by in-house clusters, the recent development of the cloud computing models (Hayes, 2008) enables companies to quickly expand and shrink their computational power on demand to meet their needs. Nonetheless, unlike private clusters, the access to the computing resources on the Cloud is performed through a network access, e.g. the Internet, which is typically slower and more susceptible to connection problems.

In order to address these challenges, several approaches have been developed in the past, however, most of them require the installation of specialized or custom software, which usually requires the intervention of the systems administrators. In this paper, we discuss how we leverage the PY-PITS runtime system (Borin et al., 2016) and the SPITS programming model (Borin et al., 2015) to efficiently execute seismic processing programs on real-world heterogeneous clusters and present an experiment that demonstrates the efficiency and robustness of the system when performing the regularization of seismic data on several clusters and workstations simultaneously.

The paper is organized as follows: first, we present the related work, then, we provide an overview of the SPITS programming model and the PY-PITS runtime system. After this, we discuss the execution of the PY-PITS runtime and user programs on heterogeneous clusters and show the experimental results. Finally, we present our conclusions.

Related Work

Grid computing (Foster and Kesselman, 2003) is a well-established programming model which leverages the computing power of several heterogeneous computers connected through the internet and/or dedicated links. To allow interoperability, many technologies were standardized by the Open Grid Forum[1], such as the GridFTP, to account for data transfer, and the Simple API for Grid Applications (SAGA).

Despite all efforts to standardize grid computing, Medeiros et al. (2003) saw a *hint of immaturity* in the area by surveying users of many grid systems. They verified many of them had trouble configuring and diagnosing faults that occurred during execution of the applications, requiring deep knowledge of the underlying system. More efforts to ease the usage of grid systems were discussed by Bal et al. (2009), where a framework was proposed to simplify the programming and deployment of grid applications. Taura et al. (2003) also developed a framework that allows dynamic join/leave of resources. Although these systems are more generic, PY-PITS strives for simplicity.

Souza et al. (2014) were able to utilize workstations for computation that otherwise would spend more than half of the week time (67%) idle. But their approach

[1] More information available at: www.ogf.org

based on Message Passing Interface (MPI) required the introduction of delays in the code to throttle down the application when a user was detected. Furthermore, the workstations comprised an entirely separate cluster, and the lack of dynamic provisioning of MPI required the job to be manually split between clusters and constantly checked for an increasing number of "dead" processes that would cause the job to halt. Our approach also takes advantage of idle workstations in order to increase the computing resources available, however, the dynamic nature of our system allows us to simply terminate the processes in the workstations and restart them, automatically aggregating the resource to the whole, also removing the need to manually split the work between clusters.

Souza et al. (2015) discuss the necessity for robust and portable software in heterogeneous systems in the industry. A modular library allows them to harness several types of hardware and then hand-tune each module for specific targets. Even though this work focus on the evaluation of Central Processing Unit (CPU) code running on heterogeneous clusters environments, the PY-PITS runtime system allows the user to build modules that use the computational power of hardware accelerators. In fact, we have already designed and executed applications that combine PY-PITS with the Open Computing Language (OpenCL) framework to harness the computational power of hardware accelerators in heterogeneous clusters.

SPITS Programming Model and the PY-PITS Runtime System

Most seismic processing problems (e.g., Common-Midpoint (CMP), Common-Reflection-Surface (CRS), migration) tend to fall into a class known as "embarrassingly parallel" problems. These problems can usually be decomposed into a series of completely independent tasks where an operator is applied to each sample of each trace in a given dataset.

The SPITS programming model (Borin et al., 2015) aids in the process of converting these types of problems into applications by providing a clear distinction of the responsibilities of each component in the system. The Job Manager is responsible for executing the `generate_task` function, which splits the input into a series of independent tasks that are sent to Task Managers. Each task is consumed by a unit called Worker, which calls the `execute_task` function to process the task and produce a single result. This result is then passed to the Committer, which calls the `commit_task` function to assemble the final output from each of the partial results. From a developer standpoint, only these 3 functions must be implemented to solve the problem and all the details about dispatching, receiving, and resending tasks and results are managed by the runtime, which implements the Task Manager, the Job Manager, and the Committer. In our tests, we use an open source reference runtime implementation called PY-PITS (Borin et al., 2016) to execute our seismic processing applications.

The PY-PITS runtime was implemented in Python to allow better compatibility with a variety of operating systems and features means for dynamically adding and removing Task Managers, as well as fault tolerance.

Running PY-PITS on Heterogeneous Clusters

Communication and Connectivity

The PY-PITS implementation follows a producer-consumer model where the producer (Job Manager) actively pushes tasks to each consumer (Task Managers) in the network using Transmission Control Protocol (TCP) sockets (Tanenbaum, 2013). Therefore, Task Managers are addressed by a pair `<ip:port>`, where `ip` is the computing node Internet Protocol (IP) address and `port` is the port assigned by the operating system to the TCP socket initiated by the Task Manager process. As a consequence, every computing node must be reachable by the computer that is running the producer (Task Manager).

In a common single-cluster environment, all of the computing nodes are connected to the same local network, hence, all computing nodes can be directly reached through their local IP addresses by other computing nodes. Consequently, reachability is not a concern when both the Job Manager and the Task Managers are executed on the cluster computing nodes.

Nonetheless, reachability becomes a problem when the user wants to execute the Task Managers on the cluster computing nodes and the Job Manager on a machine outside of the cluster local network, as illustrated in Figure 1. The problem lies on the fact that clusters are usually configured so that the computing nodes are connected to a local network (with local IP addresses) and are not directly reachable from machines outside of the cluster.

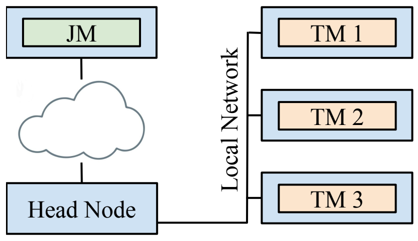

Figure 1: Executing the Job Manager outside of the cluster local network.

In order to access computing nodes on the cluster, the user must first access an entry node, which is connected to the cluster local network and is also reachable from the outer world. This node, also known as head node, is often responsible for managing the job submission system and is used by users to queue jobs and monitor their progress.

One way of enabling a Job Manager that is running outside of the cluster to reach the computing nodes is to configure the system to forward network packets from selected network ports at the head node to ports at the computing nodes. For example, the head node could be configured to forward packets sent to port 5000 of the head node to port 2500 of computing node 1, packets sent to port 5001 of the head node to port 2500 of computing node 2 and so on. In this sense, the user could configure the Task Managers to connect their TCP sockets to port 2500 and inform the Job Manager that the Task Managers are located at the addresses `<ip:5000>`, `<ip:5001>`, ..., where `ip` is the IP of the head node.

Configuring the head node to perform such forwarding in a persistent way may cause several problems, including issues with security, authentication, multi-user port sharing, etc. Moreover, it would require manual intervention from the system administrator. An alternative, and simple, way to solve this problem is to let the user perform port forwarding on the head node via secure shell (*ssh*) commands. As it turns out, the secure shell daemon and its counterpart application allow users to access the head node and perform port forwarding.

We implemented a shell script that relies on *ssh* to automate port forwarding. The script takes advantage of the Network File System (NFS) communication between the machines and the head node of each cluster, as well as the PY-PITS announcing system, to periodically check, through an SSH connection to the head node, if any new Task Managers started executing. If the list of active Task Managers does change, the script automatically builds a new *ssh* command to perform all forwarding to local ports in the node running the Job Manager and updates the local list of available Task Managers.

Performing all redirections with the least number of connections is essential when forwarding a large number of ports because it ensures that we do not reach the maximum number of allowed *ssh* connections to a head node. The tolerance to network failures implemented by PY-PITS allows us to destroy a previous *ssh* connection and associated port forwards and quickly create a new one without compromising the executing job, thus requiring only one connection to the head node. This flexibility would otherwise be impossible with frameworks and libraries that require a persistent connection to the computing nodes.

Code portability / Compilation

Maintaining portability of the software is essential when dealing with heterogeneous systems, as well as ensuring that the full potential of each system is harnessed.

In order to address the portability of the PY-PITS runtime system, we designed it using the Python language. In this way, the runtime system can be executed on any system that contains the Python virtual machine. Choosing Python as the development language for the runtime allowed us to maximize the number of Linux systems that support it without the need to install any additional packages. That is because, unlike the Java virtual machine, the Python interpreter is installed by default on most Linux server distributions and nearly all Desktop distributions. Starting on Python 2.5, all libraries required by the runtime are available in the Python distribution and no additional installation is required.

It is important to notice that virtual machines may incur extra overhead when executing guest code. Nonetheless, our experiments indicate that the PY-PITS runtime does not require much CPU time and that executing it on top of the Python virtual machine has very little effect on the overall execution time.

The portability of the user module must also be addressed. However, in this case, the extra overhead introduced by a virtual machine could affect the performance of the overall system. Most computing clusters are built with machines that employ AMD and Intel x86 processors. These processors evolved over time but maintain a common instruction set. In this sense, we could compile a single compatible binary and deploy it to all machines in the computing environment, nonetheless, this solution would fall short on performance because of the several differences between the many variations of the x86 architectures, for instance, vector instructions. To solve this problem, we decided to automate, through a set of scripts and secure shell application, the deployment and compilation of the user source code to the entire environment, thus allowing the local compilers to choose the best optimizations for each machine.

Experimental Results

In this section we discuss an experiment that illustrates the benefits of using PY-PITS when running seismic processing code on heterogeneous clusters and evaluate the efficiency of the overall system.

Figure 2 shows the network diagram for the machines considered during the test. Workstations 1 through 6 are user workstations intended for light seismic processing. Cluster 1 is an in-house cluster, and thus it was fully dedicated to the test. Clusters 2 and 3, on the other hand, are shared with other laboratories and have no specific resource manager, so competition for resources may happen if more than one user happen to use the computing nodes at the same time. Cluster 4 is a cluster designed for jobs that have a short average execution time, and it is managed by a Portable Batch System (PBS) compatible software that offers "fast" queues and a "long" queue, with the maximum execution time being 10 minutes, 40 minutes and 2 hours for the "fast" queues and 2 days for the "long" queue. Finally Cluster 5 is a high-performance computing cluster, also managed by PBS, with queues limited by 3 days of execution.

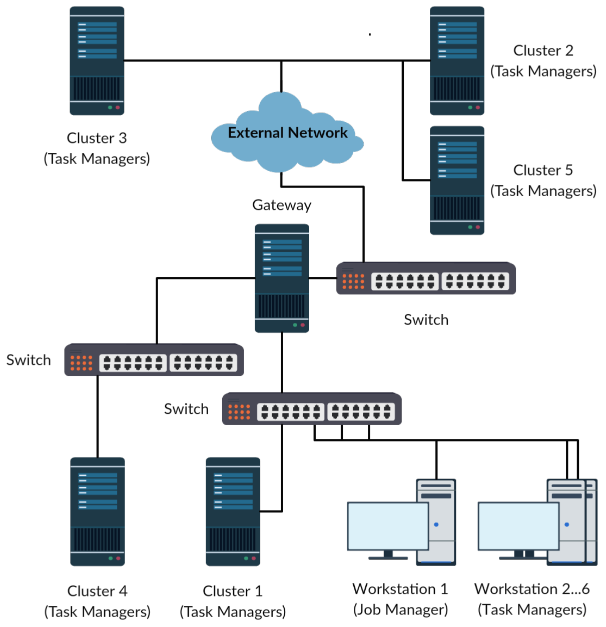

Figure 2: Network diagram of the environment considered in the test.

In total, there were 121 computing nodes available for use during the test, with their specific configuration presented

Table 1: Computational resources available for the test.

Cluster	Nodes	CPU	Virtual Cores	RAM	OS
Workstations	2	Intel(R) Core i7-4790	8	32GB	LMDE 2
	2	Intel(R) Core i7-5930K	12	64GB	CentOS 6.8
	1	Intel(R) Core i5-4460	4	16GB	Linux Mint 17.3 Rosa
Cluster 1	4	Intel(R) Xeon X5675	24	189GB	RocksCluster 6.2 (Based on Centos 6.6)
Cluster 2	2	Intel(R) Xeon E5-2630	12	32GB	Ubuntu 16.04.1 LTS
	2	Intel(R) Xeon E5-2630	24	32GB	Debian GNU/Linux 8
Cluster 3	8	Intel(R) Xeon X5673	24	189GB	Ubuntu 12.04.5 LTS
	4	Intel(R) Xeon E5-2450	32	189GB	Ubuntu 12.04.5 LTS
Cluster 4	11	AMD Opteron 6276	32	64GB	Debian 6.0.10
	3	AMD Opteron 6276	32	128GB	Debian 6.0.10
	1	AMD Opteron 6276	32	64GB	Debian 8.7
	1	AMD Opteron 6380	32	256GB	Debian 8.7
Cluster 5	80	Intel(R) Xeon E5-2670	48	64GB	SUSE Linux Enterprise Server 12 SP1

in Table 1.

The seismic processing application performs the regularization of seismic data. This process assembles seismic traces for specific coordinates based on other seismic traces. In order to compute new traces on specific coordinates, the application interpolates existing seismic traces. Each task is identified by the coordinates of the seismic trace that must be assembled. The Job Manager sends the coordinates to Task Managers, which reads the input traces from a local files (previously copied to the clusters), assembles the new trace and sends it to the PY-PITS Committer.

In order to evaluate the system, we collect the following information:

- CPU time used by the user module: our regularization module was instrumented to measure the amount of CPU time the `execute_task` function takes to finish by invoking the `getrusage` system call at the beginning and at the end of the function.
- Task Manager processes wall time and CPU usage: the Task Manager code was instrumented to record the start and end time and the amount of CPU time used by the process by invoking the `getrusage` system call. Since the Task Manager may contain working threads that use multiple processing cores, ideally, the CPU time should be equal to the wall time multiplied by the number of CPU cores allocated to the Task Manager process. This would indicate that the process used all the CPU cores during all its lifetime. However, in cases where the application is sharing CPU resources with other processes or the application is waiting for input/output (I/O) (network packets, for example), the amount of CPU time consumed by the application is smaller than the wall time multiplied by the number of cores.
- Start and end time stamps for task execution: Task Managers were instrumented record the start and end time of every call to the `execute_task` function, implemented by the user module. This information allows us to quantify the amount of time the runtime takes between invoking the `execute_task` function for consecutive tasks, which helps us identify cases in which the Task Manageris starving.

The test was performed as follows: First, the PY-PITS runtime and source code of the module were transferred to the clusters and workstations and the source code was compiled using the compiler available at each machine at the maximum optimization level available. The next step was to set up the directory where tests would run and transfer the required input files. Then we set up and ran the Job Manager on Workstation 1 as well as the port forwarding scripts connecting this workstation with all other clusters. All workstations were in the same network so no forwarding was necessary between them. Workstations 2 to 6 were set up with a Task Manager each, allocating the maximum number of CPU cores available for each Task Manager. Each cluster was set up individually, the clusters 1, 2 and 3 also executed a single Task Manager per machine that allocated the maximum number of CPU cores. For cluster 4 we queued 2 types of Task Managers, one allocating a single CPU core per Task Manager, for the fast queues, and another allocating 8 CPU cores per Task Manager, for the long queue. Finally, for cluster 5 we queued Task Managers allocating 8 CPU cores.

The computation was distributed to 183 Task Managers across the clusters and workstations and took 11 hours and 56 minutes to finish.

Resource Allocation

Figure 3 shows the relative resource allocation per machine. This metric shows the percentage of CPU cores being used by our application in a given computing node when compared to the maximum number of CPU cores concurrently allocated on the same computing node to our application during the entire execution. This chart allow us to visualize the effects of the dynamic provision of computing resources and how our application uses the resources over time, including situations caused by network slowdowns and loss of connectivity.

The first event to be observed occurs in workstations 4 and 5 at 3h:00m, and it is caused by users returning to their workstations, therefore terminating the opportunistic execution of the Task Managers. The next event is a simulated network failure with cluster 3, again at 3h:00m. Notice that the cluster 3 only ran out of tasks to process at 3h:10m, which happens because PY-PITS keeps an internal queue of tasks to hide the network latency. As soon as the connection is reestablished, cluster 3 is able

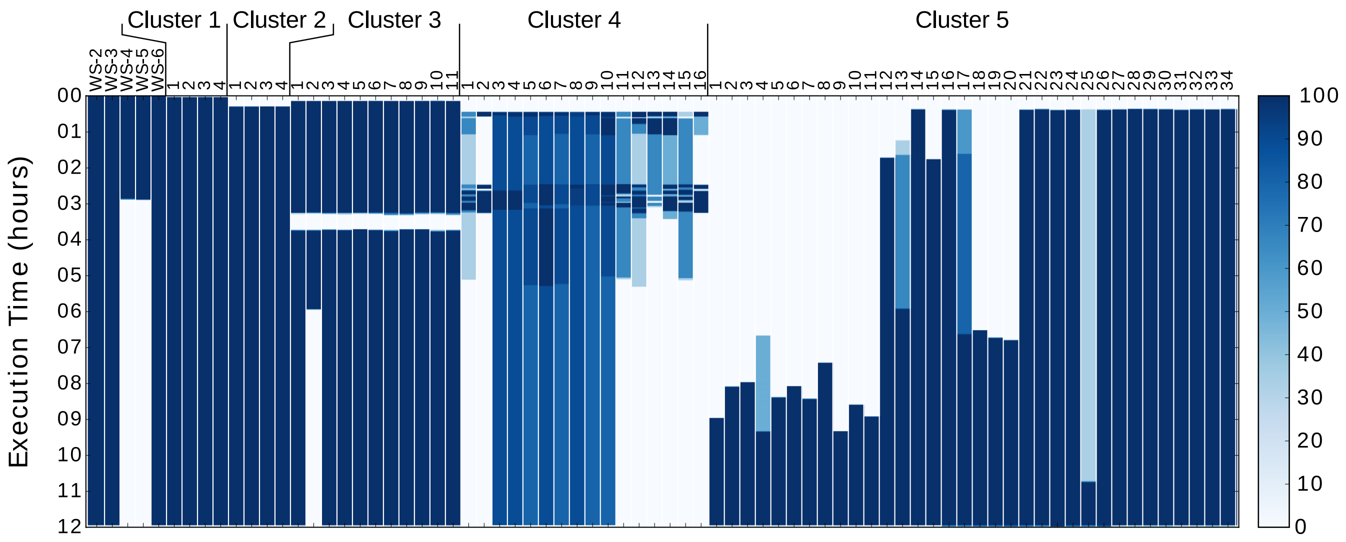

Figure 3: Relative resource allocation per machine over time.

to resume the computation. Near the time 5h:50m we can see the Task Manager at machine 2 of cluster 3 being terminated. That was caused by an user with higher privilege in that cluster requiring the full machine. Furthermore, in cluster 4 we can see several Task Managers executing and terminating due to the time limit of the fast queues. Finally, cluster 5 shows the Task Managers gaining access to compute nodes over time, enabling the system to take advantage of the computating resources as soon as they are released by other jobs.

Computation Efficiency

At the end of the execution the computation efficiency of each Task Manager was verified by comparing the amount of CPU time the Task Manager process used to execute the PY-PITS runtime and the regularization module against the Task Manager wall time multiplied by the number of CPUs allocated to the Task Manager. As a result, we used these metrics to quantify the overall computation efficiency and the overhead imposed by the PY-PITS runtime system. The efficiency allows us to see that, even if a certain number of CPU cores is allocated to the Task Managers, and they manage to pull tasks and push results without starvation, the competition with other processes running on the machine and even the overhead introduced by the runtime itself may reduce the amount of computation dedicated to solving the problem.

Figure 4 shows the computation efficiency only for the Task Managers executed until the end of the test, thus excluding the Task Managers from Workstations 4 and 5, as well as Machine 2 from Cluster 3 and the Task Managers submitted on the fast queue of cluster 4. It is possible to observe that, even though Workstation 6, Machine 1 from Cluster 2, and the entire Cluster 3 had nearly 100% relative resource allocation per machine (Figure 3), their efficiency were the lowest ($\sim 80\%$) when compared to other systems. In these cases, it happens because our application process did not have exclusive access to the resources, so processes from other users were competing for CPU time. The simulated network failure that affected Cluster 3 also contributed to the poor efficiency, though small compared to the overall execution time, because it led the Task Managers to a complete halt until the connection could be reestablished. Cluster 4 exhibited an odd efficiency drop, but at the present time there is no metric that allowed us to diagnose such effect. It may be associated to the PBS system allocating more CPU cores than what's is physically available, as part of a policy to mitigate blocking IO. Cluster 5 also presented 2 odd drops in efficiency, with significantly more runtime overhead, 1.5%, but again no metrics are available to diagnose the situation. Despite these predicted cases and inconsistencies, the average efficiency of the considered Task Managers was 96%. The overhead of the PY-PITS runtime was also small, with 99.7% of the CPU time being dedicated to the execution of the module, on average. The average used is the geometric mean of the individual efficiencies, to properly average normalized results.

Conclusions

Heterogeneous computational environments are a growing reality on the industry for many logistical reasons and cost. Thus it is fundamental that modern applications deal with these environments with efficiency.

We have used the SPITS programming model and its reference runtime PY-PITS to develop a seismic data regularization software that features scalability, fault tolerance, dynamic provisioning and that can be deployed in a wide variety of hardware and software configurations.

We were able to aggregate computational resources from five different clusters inside and outside our network, along with 6 user workstations, totaling 183 Task Managers on 75 different machines, occupying from 1 to 32 cores each, and one Job Manager / Committer.

The performance tests covered several real-world scenarios, including intermittent network connectivity, competition for resources in a shared execution environment, queuing of many jobs in a cluster with time-limited job queues, and opportunistic situations where workstations could only be used for processing while the user was away. Instrumentation of the code showed that

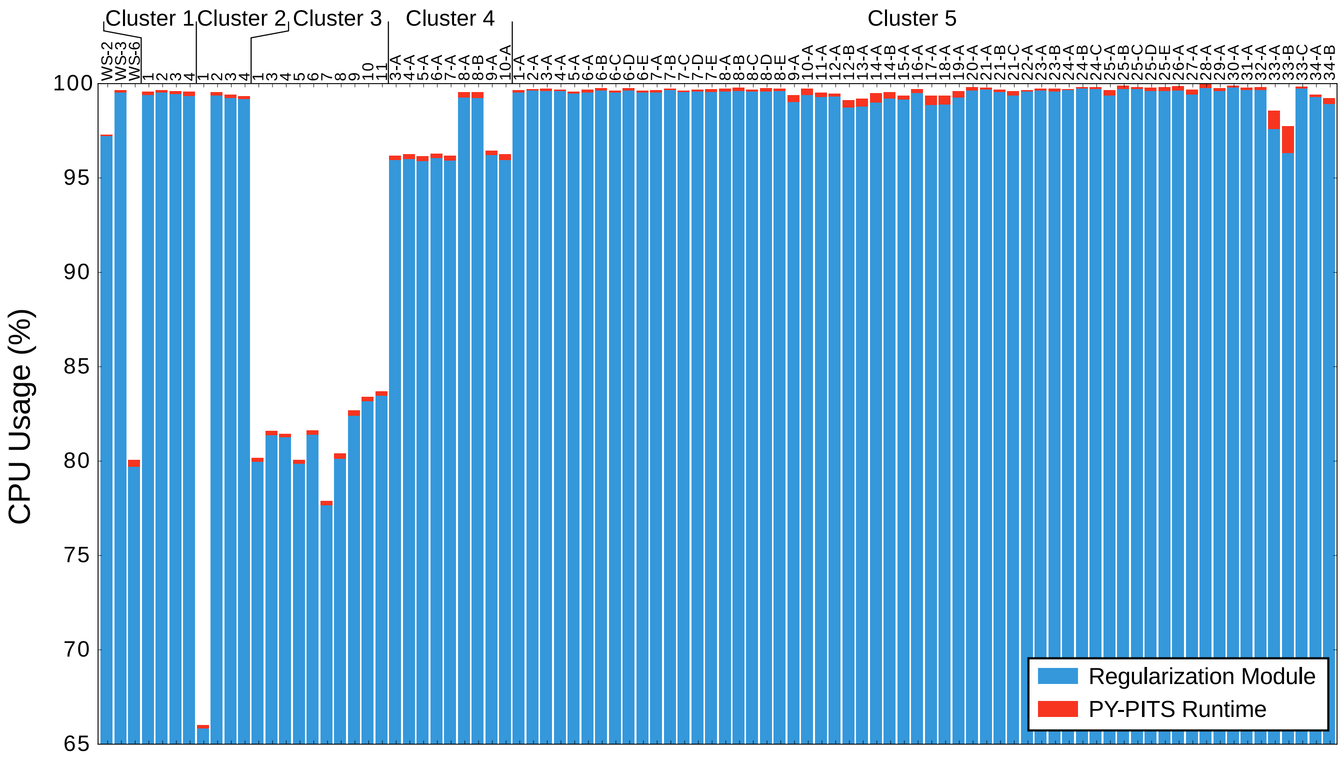

Figure 4: Computing efficiency per Task Manager.

PY-PITS was able to sustain a constant flow of tasks without causing starvation of the Workers. Additionally, PY-PITS only contributed, on average, to 0.3% of the total computation time, thus not imposing significant overhead to the computation.

Acknowledgments

We acknowledge support from the National Council for Scientific and Technological Development (CNPq-Brazil), the National Institute of Science and Technology of Petroleum Geophysics (ICTP-GP-Brazil) and the Center for Computational Engineering and Sciences (Fapesp/Cepid # 2013/08293-7-Brazil). We also acknowledge support from the sponsors of the Wave Inversion Technology (WIT) Consortium, CAPES and Petrobras, as well as the CEPETRO and its laboratories that supplied part of the computational infrastructure.

References

Bal, H. E., N. Drost, R. Kemp, J. Maassen, R. V. V. Nieuwpoort, C. V. Reeuwijk, and F. J. Seinstra, 2009, Ibis : Real-World Problem Solving using Real-World Grids.

Borin, E., C. Benedicto, I. L. Rodrigues, F. Pisani, M. Tygel, and M. Breternitz, 2016, Py-pits: A scalable python runtime system for the computation of partially idempotent tasks: 2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), 7–12.

Borin, E., I. L. Rodrigues, A. T. Novo, J. D. Sacramento, M. Breternitz, and M. Tygel, 2015, Efficient and fault tolerant computation of partially idempotent tasks: Presented at the 14th International Congress of the Brazilian Geophysical Society & EXPOGEF, Rio de Janeiro, Brazil, 3-6 August 2015, Society of Exploration Geophysicists.

Foster, I., and C. Kesselman, 2003, What is the grid: A three point checklist, **20**.

Hayes, B., 2008, Cloud computing: Commun. ACM, **51**, 9–11.

Huang, Z., N. Rustagi, N. Veeraraghavan, A. Carroll, R. A. Gibbs, E. Boerwinkle, M. G. Venkata, and F. Yu, 2016, A hybrid computational strategy to address WGS variant analysis in >5000 samples: BMC Bioinformatics, **17**, 361.

Medeiros, R., W. Cirne, and F. V. Brasileiro, 2003, Why are they so bad and What can be done about it ?: Computing.

Souza, P., R. Bonfá, J. Vasconcelos, R. Gonzales, T. Teixeira, R. Paoliello, A. Bissoli, and L. Bozi, 2014, A cluster of workstations for seismic data processing using GPU: Presented at the EAGE Workshop on High Performance Computing for Upstream, EAGE Publications.

Souza, P., C. Newburn, and L. Borges, 2015, Heterogeneous architecture library: Presented at the Second EAGE Workshop on High Performance Computing for Upstream, EAGE Publications.

Tanenbaum, 2013, Computer networks, 5th edition: Pearson India.

Taura, K., K. Kaneda, T. Endo, and a. Yonezawa, 2003, Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources: ACM SIGPLAN Notices, 216–229.