Torque Extension for Scheduling in Cluster Environments

Răzvan Ghițulete, Ana Ion, Mihai Prică
Automatic Control and Computers Faculty
Politehnica University of Bucharest
Emails: {razvan.ghitulete, ana.ion, mihai.prica}@cti.pub.ro

***Abstract*—This paper presents a solution for scheduling different types of tasks in a cluster environment running the Torque open source resource manager. Given the heterogeneity of the resources specific to clusters, it is important to provide efficient ways of dispatching jobs, maximizing throughput and performance. Currently, Torque only offers a first come, first served implementation (without using any scheduling plugins) and this is the problem we address in this paper. The proposed solution provides means of scheduling using different algorithms such as Bag-of-Tasks, aperiodic and DAG scheduling. We will present implementation details for each of the models, evaluating their performance using different workload scenarios.**

***Index Terms*—cluster scheduling, DAG, Bag-of-Tasks, Torque**

I. Introduction

Cluster technologies enable the aggregation of distributed resources for solving large scale and computationally intensive problems. The parallel and distributed computing community has put a lot of effort in developing and understanding scheduling algorithms for this kind of environments. Several of these algorithms have been implemented in commercial or open source solutions. Most of the clusters use *space-sharing* [10] algorithms that assign resources to a single task until that task completes execution. Some of the most common examples in this category are first in, first out (FIFO), round robin (RR), shortest job first (SJF), longest job first (LJB). FCFS and FIFO work in a similar manner, assigning tasks to processors in the order they arrive. RR assigns the tasks in the order of their arrival, using a cyclical approach. SJF and LJF sort, ascending and respectively descending, the tasks based on the length of their execution time, targeting the improvement of the average turnaround time or that of the system's utilization.

Based on these simple scheduling models, more complex approaches can be developed. The Bag-of-Tasks model [5] represents an application made of a collection of independent and identical tasks that are to be scheduled on a master-worker platform. Given the case of a homogeneous environment, a greedy approach will achieve an optimal throughput. Still, clusters are usually characterized by resource heterogeneity in both CPUs and communication bandwidths, so it becomes crucial to select which resources will be mapped to each component of the application, before initiating the computation phase.

The aperiodic scheduling paradigm is strictly related to the distinction between hard and soft tasks. If meeting a task's deadline is critical to the system's operation, then the task is considered hard and scheduling decisions must be based on respecting this constraint. If it is desirable to meet a task's deadline, but missing it is tolerable, then the task is considered to be a soft one. For each of the categories exist different scheduling approaches. This paper will consider the problem of scheduling soft deadline aperiodic tasks that require fast average response time from the scheduling entity. This goal is achieved by the use of heuristic algorithms, as obtaining optimal solutions is generally a NP-hard problem.

In comparison to the Bag-of-Tasks model, the directed acyclic graph (DAG) representation is used when the input jobs are characterized by execution dependencies. Each node in the graph represents an executable job and each directed edge implies a precedence constraint between two tasks. The task scheduling problem consists of allocating the tasks on existing resources so that all precedence constraints are respected and the makespan (overall execution time) is minimized. Various heuristic can be used to achieve this goal, some of the most popular ones being Dynamic Level Scheduling (DLS), Heterogeneous Earliest Finish Time (HEFT) and Fastest Critical Path (FCP).

Taking into consideration the aspects previously presented, the goal of this paper is to enhance the Torque scheduling solution with support for the described models and to compare these approaches. Torque is a distributed resource manager that was developed starting from the PBS project. It provides control over batch jobs and distributed computing resources. It is an open source tool that adds to PBS [3] important features in terms of scalability, usability, reliability and functionalities. The main objective of this project is the design and implementation of a cluster scheduling service on top of Torque. This will improving the scheduling quality by focusing as much as possible on the user demands. The solution will be validated by thorough testing in a real-world, heterogeneous and dynamic environment.

The rest of this document is organized as follows. Section 2 presents the current scheduling solutions Torque is compatible with. Section 3 offers details on the design of the solution, presenting the architecture of the meta-scheduler we propose. Section 4 describes the implementation details. In section 5 experimental results are discussed, while the last section, 6, synthesizes our conclusions and outlines future work directions.

II. Related Work

A. Open Source Resource Managers

Selecting the proper scheduler is an important decision that affects directly the performance of the cluster's utilization. Responsiveness, availability, scheduling techniques are all dependent on the used scheduling tool. The default Torque scheduler, *pbs_sched*, is a basic solution that manages tasks in a FCFS manner. It provides poor utilization of the resources and does not scale under heavy workloads. The alternative options are Maui Scheduler or Moab Workload Manager.

1) Maui Job Scheduler: Maui [1] is an advanced open source job scheduler designed to optimize the utilization of the system's resources in heterogeneous environments, such as clusters. Moreover, it is highly suitable for policy driven tasks, being focused on fast turnaround time for large parallel jobs. These aspects make it an efficient solution for HPC. It is compatible with other resource managers, being able to replace built-in schedulers and to improve the overall performance.

Maui uses a two phase scheduling algorithm. The first phase schedules high priority jobs, using advance reservation. During the second phase, a backfill algorithm is used in order to schedule low priority jobs. The fair-share technique is used when making scheduling decisions based on job's history. This approach is motivated by Maui's internal behaviour that is based on a single, unified, working queue that maximizes the utilization of resources.

Although users are guaranteed specific QoS, administrators are granted most of the control when using Maui. In this manner, access to resources can be filtered according to scheduling policies. Administrators are allowed to enable different QoS levels of access for users and jobs, which can then be preemptively identified. Maui uses QBank for allocation management, tools which allows multisite control over the use of resources. The scheduling daemon is centralized and runs on a single node.

Preemption is also supported, high priority jobs being allowed to interrupt lower priority or backfill jobs if resources are not available. Resources reserved for high priority tasks can be assigned to lower priority ones if no high priority task is queued. However, the resources will be reclaimed once a high priority job is submitted.

Although offering a elaborate model of scheduling, Maui lacks some of the essential features needed for efficient scheduling in cluster environments. It does not support multiple queues submission. In addition, it is a rather rigid platform, new scheduling algorithms being difficult to integrate. The priority driven scheduling may lead to poor performance caused by process starvation.

2) Moab Workload Scheduler: Moab [2] is a solution to manage a HPC environment with support for job scheduling and workload management, offering an adaptive switcher for both Linux and Windows workloads. At a high level it applies site policies and optimizations to orchestrate jobs and services across compute, network and storage resources. More than just scheduling facilities, it provides an entire ecosystem for cluster monitoring and performance evaluation.

Moab's functionality is based on the assignment of priorities to jobs, based on credentials, resources, usage and job attributes. The priority of a job is computed as sum of the priorities of different individual factors. After assigning a dynamic priority value (which can change during runtime), a task is submitted to a queue.

Offering a pluggable architecture, Moab can interface with external Resource Managers (such as Torque). The resource requirements are fulfilled using job templates. It is possible to define new resources, but also to add or remove existing ones from running queues. The queues can be configured to use groups of resources and the scheduler may take decisions based on resources availability.

Moab has an extensive set of scheduling algorithms. It can schedule batch jobs, parallel jobs, and service workload. Moab's support for parallel jobs relates directly to the parallel support provided by the Resource Managers. Torque supports parallel and array jobs. Time based recurrence jobs are supported in Moab through the use of Triggers and Reservations. Using Reservations guarantees availability of resources and job start time — which might be critical for recurrence jobs. Moab supports various event based scheduling in the form of triggers. Some of the supported events are: threshold limits in reservations, jobs hold/preempt scenarios, and scheduler events. Moab can schedule workflows (which can be a group of jobs). It also supports workflow creation through templates, API and custom processes.

B. Proprietary Resource Managers

Besides the open source resource manager, commercial solutions are also available. Platform Load Sharing Facility (PLSF) is part of this category. It pursues maximization of resource utilization within the constraints of the local administration policies. PLSF uses basic scheduling algorithms, as well as advance reservation or backfill. The two available versions, PLSF and PLSF HPC, offer dynamic scheduling decision mechanisms. Using these means, jobs can be migrated to computing nodes or rescheduled during execution, at the cost of a higher scheduling computation cost. PLSF can interface with external schedulers, such as Maui, enabling sophisticated scheduling. The solution proposed in this paper is only compatible with the Torque resource manager, but, in the future, support for PLSF may also be added.

III. Architecture

One of the main design goals for this project was the decoupling between the proposed Torque extension and the existing resource manager so that our solution will be as independent as possible while making use of the underlying scheduling architecture. Another major design goal was the seamless integration between the meta-scheduler and the existing manager. In this scope, the design is a simple one, both from the user's perspective as well as from the developer's point of view. An overview of the design is represented in Figure 1.

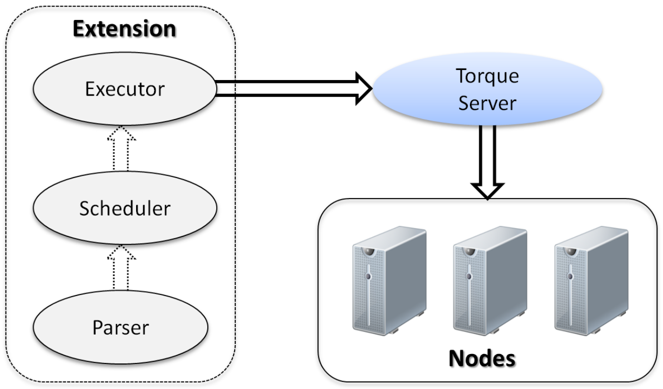

Figure 1. Extension architecture overview

Based on these goals, the extension was built using a modular design, with three independent stages that communicate in a pipeline fashion.

A. Parser

The Parser is the main entry point into the program and is responsible for interpreting the user provided input. It receives a resources description, and also a list of the tasks that have to be scheduled. It has a modular design, so that new configuration file formats can easily be integrated. Different types of workloads can be specified, as we enhanced support for dependent tasks execution as well as bag of tasks scenarios. After the parser has finished reading and validating the user provided input, it can start the second stage of the pipeline.

B. Scheduler

While the parser is the main entry point into the application, the scheduler stage represents the core. Based on the invoked scheduling algorithm and the workload provided, a run order will be outputted. This will, then, be forwarded to the next and final stage of the pipeline, the executor. Various scheduling policies are implemented at this stage, the final scheduling configuration being highly dependent on the resources that are available. As well as the parser, the scheduler is designed so that it can be easily extended by adding more scheduling algorithms, without the need to modify the core application.

C. Executor

In the final stage of the pipeline, the tasks are sent to the cluster's resource manager to be executed. Extra precautions have to be taken to ensure the tasks are not scheduled in a different order by the basic Torque scheduler. This stage could as well be integrated in the Scheduler, but this would bring unwanted overhead to it and would also restrict the flexibility associated with the future extension of the Scheduler.

IV. Implementation

In order to enhance Torque with support for the scheduling models that were mentioned in the Introduction section of this paper, we created a Python implementation of a meta-scheduler, built according to the architectural design previously described. As the response time is not critical (scheduling is statically performed, using input files), a high level programming language suits the implementation. In addition, Python also offers support for DAG representations (the pygraph packet) and is a free and open source programming language, thus the solution being portable and easy to extend. Figure 2 illustrates the classes diagram used in the implementation. The modular design allows easy integration of the current solution with other scheduling models or algorithms.

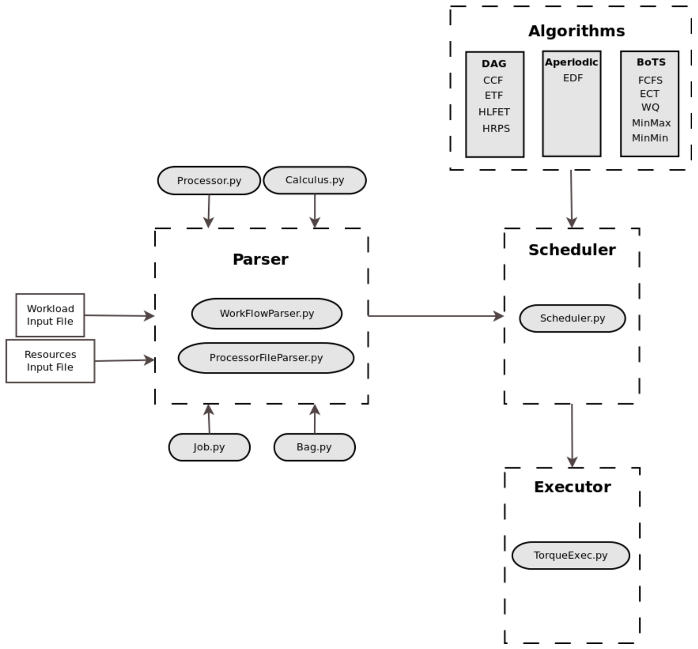

Figure 2. Classes diagram

As mentioned at the beginning of this document, the Torque extension we propose consolidates the current version of the resource manager by adding support for new types of scheduling. The DAG model is associated with tasks that are tightly coupled and present dependencies among each other. Aperiodic scheduling is specific for running tasks that have to be completed before meeting a deadline. The Bag of Tasks approach addresses independent tasks that reach the system organized in separate ”bags” (applications). The results obtained for each of these paradigms are discussed later in this paper, different heuristics for each of the scheduling model being analyzed.

A. DAG Scheduling

This type of scheduling is specific to applications where we have dependencies between the tasks and an estimated running time is known a priori for each task. The application can be modeled using a directed acyclic graph (DAG) in which the nodes represent the tasks and the edges represent inter-task dependencies.

For a solution to be valid, all the inter task dependencies have to be satisfied. The results of the algorithms are compared based on the overall completion time (makespan). Because the problem is NP-complete, various heuristics are used for minimizing the time for the scheduling itself.

Most of the heuristics are based on list-scheduling which consists of two phases: task prioritizing, where each task is assigned a priority and the actual scheduling, where the tasks are sorted by their priority and are assigned to the processor that minimizes a cost function. If the processor selection phase starts after the priorities have been assigned to all the tasks, the algorithm is called static and if the two phases are interleaved the algorithm is called dynamic. For this project we chose to implement two static and two dynamic list-scheduling algorithms [7].

- *Highest Level First with Estimated Time (HLFET):* The HLFET algorithm is one of the simplest static list-scheduling algorithms. At each step, the tasks are scheduled on the processor that allows for the earliest start-time. It uses an attribute called static level (SL) as the scheduling priority. The static level of a node is the cost of the longest path from the node to the exit node without taking into account the communication costs between the tasks. The tasks are sorted in descending order by their static level and are scheduled on the processor that allows for the earliest execution start time.
- *Modified Critical Path (MCP):* This algorithm is very similar to HLFET, except it uses the LST(Latest Start Time) attribute for the task prioritizing. The tasks are sorted in ascending order by their LST and are processed in a sequential manner. Each task is scheduled on the processor that allows for the earliest execution start time. In the case of equivalent LST value, the LST values of the children of the tasks are taken into consideration to break the tie.
- *Earliest Time First (ETF):* This main goal of this dynamic list-scheduling algorithm is achieving a good load balancing on the available processors. At each step of the algorithm, the earliest start-time is computed for each ready node (node having all parents scheduled) and the one with the smallest time to start is selected. In the case of a tie, the node with highest static level priority will be chosen.
- *Dynamic Level Scheduling (DLS):* The DLS algorithm is very similar to ETF except it uses an attribute called dynamic level(DL) as the scheduling priority. The dynamic level is the difference between the static level of a node and the earliest start-time on a processor. At each step of the algorithm, the DL is computed for each ready node(node having all parents scheduled) and the node-processor pair that gives the largest value of DL is selected for scheduling.

B. Bag of Tasks Scheduling

For the Bag of Tasks scheduling paradigm, there are two scheduling stages that were taken into consideration: bag selection and task scheduling inside the bag. The high level of parallelism which characterizes cluster environments where Torque is usually run, allows the implementation of different policies for the two steps involved by the Bag of Tasks model. The strategies further described will be compared in terms of the makespan that was obtained during testing.

There are several techniques for performing the bag selection phase [6]. Each bag is associated a queue when it enters the system and the centralized scheduler decides the way the tasks in each bag are mapped to the available resources. Moreover, we assume all bags are submitted at the beginning of the scheduling period, thus all information about tasks inside a bag or tasks duration is known a priori.

In our implementation we considered the following bag selection policies:

- *First Come First Served Exclusive (FCFS-excl):* Applications (bags) are scheduled in the order of their arrival. No other bag is given access to resources until all the tasks in a bag are completed. This approach is indicated especially when the submitted bags belong to different users and there are strong security policies enforced.
- *First Come First Served Shared (FCFS-shared):* In a similar manner to the previously described policy, applications are also scheduled in the order of their arrival, but resources are not longer assigned to a single Bag. Once the current running bag finishes executing its tasks, jobs from the following bag will be submitted in the system. Certainly, this approach ensures a better utilization of the available resources, as the processors that complete execution earlier are no longer kept idle until all tasks in a bag are finished.
- *Minimum Bag First (MinBag):* This policy uses available a priori information in order to sort the bags by their execution time (execution time of a bag is sum of the execution time of the tasks inside a bag). In this manner the bags that have the minimum completion time are submitted first, so in case of a system failure they are more probable to have been executed. Both shared and exclusive access to resources are available for this policy.

Task scheduling inside a bag has the highest impact on the resulting maskespan. Ideally, the tasks inside a bag are identical and they target the same behavioral response from the system. Nevertheless, the I/O operations and the computation cycles of a task often lead to different execution costs. The heuristics presented in [8] target the same meta-scheduling approach that we address in this paper. Taking into consideration that the scheduling technique we use statically assigns jobs to resources before starting execution, it is important to implement a method that delivers a good response time. Under these circumstances, a near optimal solution is considered to be satisfactory. The Torque extension we developed uses the following heuristics for programming the tasks inside a bag:

- *First Come First Served (FCFS):* The tasks are executed in the order of their arrival. Although this heuristic has the best response time, no information about resources state or performance is used by the scheduler. In addition, load on the machines during execution is not taken into consideration.

- *Earliest Completion Time (ECT):* Each task is assigned to the processor that leads to the earliest completion time. When comparing the time to complete the task on a processor the following function is used to evaluate the performance:

$$ratio = TaskCost * ProcLoad/ProcPerformance \quad (1)$$

  The parameters in the formula 1 represent the cost to execute the task, as mentioned in the workload input file (*TaskCost*), the time the task has to wait until execution for the chosen processor (*ProcLoad*) and the performance unit assigned to the processor (*ProcPerformance*).
- *MinMin:* Min-min heuristic uses minimum completion time (MCT) as a metric, so the task which can be completed will be firstly executed. Starting from the set of all unmapped tasks, the set of minimum completion times, *M = min(completion time(Ti , Mj)) for (1 $\leq$ i $\leq$ n, 1 $\leq$ j $\leq$ m)*, is computed. The completion time for a task is computed using the evaluation function mentioned in 1. M consists of one entry for each unmapped task. The task with the minimum completion time is selected and assigned to the corresponding machine. The mapped task is removed from the set and the process repeats until all tasks are assigned resources (U is empty).
- *MaxMin:* Like MinMin, the MaxMin heuristic also uses the MCT as metric. Starting from the same set U of unmapped tasks, the M set of minimum completion times is found. The task with the maximum completion time is chosen and assigned to the machine, removing it from the set U. The steps are repeated until all the tasks are mapped.
- *WorkQueue:* This basic heuristic randomly chooses the next task to be scheduled. The designated job is assigned the best machine (the one which leads to the minimum completion time).

C. Aperiodic Scheduling

The implemented Torque Extension also has a module that support scheduling for aperiodic tasks. This paradigm is completely different from the ones presented above due to the unique restrictions introduced by each task. Due to the fact that there is no guarantee, whatsoever, that the tasks can be scheduled respecting the user provided deadlines, our meta-scheduler introduces the concept of soft task deadlines, meaning that the outputted scheduling might miss some deadlines.

To illustrate the aperiodic scheduling feature, we implemented a modified version of the Earliest Deadline Scheduling (EDF) heuristic. As the name clearly states the main criteria for scheduling tasks is the deadline. Tasks are sorted in ascending deadline order and then are scheduled at the next suitable time interval. The EDF[11] heuristic is one of the simplest heuristics available for scheduling aperiodic tasks, but also one of the few that can be used on the Torque infrastructure, due to its non-preemptive behavior. This non-preemptiveness exposed by the Torque scheduler is also the main reason for introducing the soft deadline concept, in our implementation, and allowing tasks to miss deadlines. Another difference from the standard EDF algorithm is the ability to schedule on multiple cores. This is done by scheduling each task on the processor that offers the earliest start time possible which will in turn offer the highest probability of not missing any deadlines.

V. Experimental Results

In order to test the implementation of the Torque extension we developed, we ran a series of tests in a simulated environment. Details regarding the set up of the machines, as well as the workflow used for testing are presented in the following sections, specific to each type of scheduling.

A. Bag of Tasks Experimental Results

For testing the Bag of Tasks implementation we designed, we created a heterogeneous environment that uses eight nodes to run the tasks. As mentioned before, the scheduling phase is performed before running any of the jobs, so the tasks are submitted to Torque using the order outputted by the scheduler. Figure 3 illustrates the makespan values obtained using the FCFS bag selection, while in Figure 4 is presented the evolution of the system using the minimum bag selection policy. Both policies were tested using the same workload and tasks were scheduled according to the five heuristics.

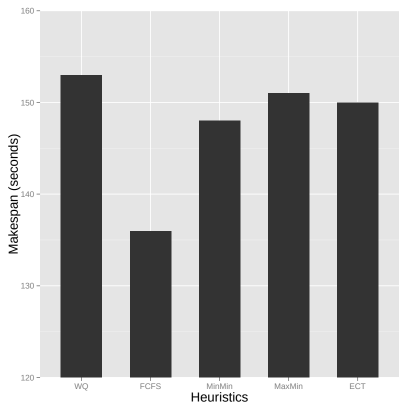

Figure 3. FCFS-shared bag selection

The good performance delivered by the MinMin heuristic is based on minimizing the makespan by choosing the fastest processor for each task. This choice guarantees that all tasks will executed their earliest completion time, while adding the minimum load on running nodes. The homogeneity of tasks specific to a bag (application) leads to their optimal scheduling when using the FCFS heuristic for both bag selection and task scheduling.

B. Aperiodic Scheduling Experimental Results

For testing the EDF implementation provided, we used the dataset from Table I which required the heuristic to provide a suitable scheduling for a 3 node cluster. The output of the algorithm can be observed in the last two columns of the same

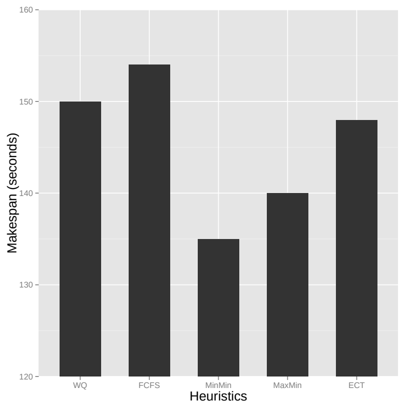

Figure 4. MinBag-shared bag selection

	Arrival	Cost	Deadline	Node	Start Time
Task_A	1	1	20	1	1
Task_B	5	4	18	2	5
Task_C	4	2	25	3	4
Task_D	3	1	21	2	3
Task_E	2	8	29	1	9
Task_F	3	1	22	3	3
Task_G	3	6	11	1	3
Task_H	3	9	26	3	6
Task_I	2	1	16	3	2
Task_J	9	15	30	2	9
Task_K	2	8	45	3	15
Task_L	1	2	15	2	1

Table I
APERIODIC TASKS

Table I which illustrates a valid planning of the tasks with no missed deadlines.

C. DAG Scheduling Experimental Results

A random graph generator [4] was used to generate the input files for the DAG module. The graphs are generated based on a number of parameters like the total number of tasks, the number of levels in the graph and the ratio of communication cost to computation cost. The algorithms were evaluated by taking into consideration the overall execution time (makespan). The results can be seen in Table II. The rows marked with B represent the percentage of cases when the algorithm on the row was better that the algorithm on the column. The rows with E represent the number of cases where the two results had the same makespan.

The results vary depending on the characteristics of the graph. The dynamic algorithms averaged a smaller makespan and, consequently, a higher speed-up than the static list-scheduling algorithms but with the cost of a higher computation time. If the scheduling time is a important factor, then the lower complexity of the static algorithms makes them the better choice. The two dynamic algorithms have almost the same performances, while HLFET is better than MCP.

The input files use the same format as DAGMan [9], a meta-scheduler within HTCondor. The format was extended with a new section for specifying the communication costs between tasks.

		HLFET	MCP	ETF	DLS
HLFET	B		51%	36%	25%
	E		15%	19%	40%
MCP	B	34%		38%	33%
	E	15%		7%	10%
ETF	B	45%	55%		18%
	E	19%	7%		43%
DLS	B	35%	57%	39%	
	E	40%	10%	43%	

Table II
COMPARISON OF DAG SCHEDULING ALGORITHMS

VI. CONCLUSIONS

Scheduling is one of the core actions when using heterogenous systems in order to run jobs. Such environments need powerful resource managers to ensure that the best results are delivered to the end user. Torque is a popular implementation for this type of application and the meta-scheduler implemented by this project extends its capabilities by adding an extension that allows it to properly schedule workloads like DAG, Bag-of-Tasks or aperiodic tasks. The experimental results we measured validate the implementation of the heuristics described in the previous sections. As we built a modular system, in the future we plan to add support for other types of scheduling too, as well as to stress test our current implementation.

The meta-scheduler proposed by this article does not provide any mechanisms for fault tolerance. The implementation of a rescheduling policy for tasks that generate errors at runtime is a possible future work.

REFERENCES

[1] Maui Scheduler. http://www.adaptivecomputing.com/resources/docs/maui/mauiadmin.

[2] Moab Workload Manager. http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/.

[3] Portable Batch System. www.openpbs.org.

[4] Synthetic DAG generation. http://www.loria.fr/ suter/dags.html.

[5] M. Adler, Y. Gong, and A. L. Rosenberg. Optimal sharing of bags of tasks in heterogeneous clusters. In *5th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'03)*, pages 1 – 10. ACM Press, 2003.

[6] C. Anglano and M. Canonico. Scheduling algorithms for multiple bag-of-task applications on desktop grids: a knowledge-free approach.

[7] T. Hagras and J. Janecek. Static vs dynamic list-scheduling performance comparison. pages 16 – 21. Acta Polytechnica, 2003.

[8] H. Izakian, A. Abraham, and V. Snasel. Performance comparison of six efficient pure heursitics for scheduling meta-tasks on heterogeneous distributed environments. pages 695 – 710. Neural Network World, 2009.

[9] J. W. Peter Couvares, Tevik Kosar and K. Wenger. Workflow in condor. In *Workflows for e-Science*. Springer Press, 2007.

[10] P. S. and K. Nithya.D. An integrated approach of task scheduling with space sharing algorithm for effective processor allocation and scheduling of parallel application. In *International Journal of Research and Reviews in Electrical and Computer Engineering*, United Kingdom, 2011. Science Academy Publisher.

[11] J. L. Sungyoung Lee, Hyungill Kim. A soft aperodic task scheduling algorithm in dynamic-priority systems.