Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Integer Programming Based Heterogeneous CPU-GPU Cluster Scheduler for SLURM Resource Manager

Seren Soner, Can Özturan

Dept. of Computer Engineering, Bogazici University, Istanbul, Turkey

Abstract

We present an integer programming based heterogeneous CPU-GPU cluster scheduler for the widely used SLURM resource manager. Our scheduler algorithm takes windows of jobs and solves an allocation problem in which free CPU cores and GPU cards are allocated collectively to jobs so as to maximize some objective function. We perform realistic SLURM emulation tests using the Effective System Performance (ESP) workloads. The test results show that our scheduler produces better resource utilization and shorter average job waiting times. The SLURM scheduler plug-in that implements our algorithm is available at http://code.google.com/p/slurm-ipsched/.

1. Introduction

SLURM [1] is an open-source resource management software distributed under the GPL license. It was designed with simplicity, portability and scalability in mind. It has a plug-in mechanism that can be used by developers to easily extend SLURM functionality by writing their own plug-ins. Older versions of SLURM had a simple first-come-first-served (FCFS) scheduler, but the recently introduced new version has advanced features such as backfilling, fair share, preemption, multi-priority, advanced reservation. Some supercomputer centers use SLURM coupled with other schedulers such as MOAB [2], Maui [3] and LSF [4]. SLURM has been receiving a lot of attention from the supercomputer centers lately. It has been reported by main SLURM developers that about 40% of TOP500 installations use SLURM [5].

Recently, heterogeneous clusters and supercomputers that employ both ordinary CPUs and GPUs have started to appear. Currently, such systems typically have 8 or 12 CPU cores and 1-4 GPU per node. Jobs that utilize only CPU cores or both CPU cores and GPUs can be submitted to the system. Job schedulers like SLURM [1] take one job at a time from the front of the job queue; go through the list of nodes and best-fit the job's resource requirements to the nodes that have available resources. In such systems, GPU resources can be wasted if the job scheduler assigns jobs that utilize only CPU cores to the nodes that have both free CPU cores and GPUs. Table 1 illustrates this problem on a system with 1024 nodes with each node having 8 cores and 2 GPU's. The example submits three sleep jobs each of which lasts for 1000 seconds. Suppose jobs J1, J2, J3 appear in the queue in this order (with J1 at the front of the queue, J2 second and J3 third). Note that SLURM will grant jobs J1 and J2 with the requested resources immediately as follows:

- J1 will be best-fit on 512 nodes, hence allowing no other jobs to use the GPUs on these nodes.
- J2 will be allocated 512 nodes using 4 cores and 2 GPUs on each node.

This allocation, however, will cause J3 to wait 1000 seconds until J1 and J2 are finished. On the other hand, if we take all the jobs, J1, J2 and J3, collectively and solve an assignment problem, it is possible to allocate the requested resources to all the jobs and finish execution of all of jobs in 1000 seconds rather than in 2000 seconds. This can be done by assigning 512 nodes with 4 cores and 2 GPUs to each of jobs J2 and J3 and 1024 nodes with 4 cores to J1.

Table 1. Example Illustrating Advantage Of Solving Collective Allocation Problem

Job	***Resources Requested***	***Slurm Command***
J1	4096 cores	`srun -n 4096 sleep 1000`
J2	2048 cores on 512 nodes with 2 GPUs per node	`srun -N 512 --gres=gpu:2 -n 2048 sleep 1000`
J3	2048 cores on 512 nodes with 2 GPUs per node	`srun -N 512 --gres=gpu:2 -n 2048 sleep 1000`

In this paper, we are motivated by the scenario exemplified by Table 1 to develop a scheduling algorithm and a plug-in for SLURM that collectively takes a number of jobs and solves an assignment problem among the jobs and the available resources. The assignment problem that is solved at each step of scheduling is formulated and solved as an integer programming problem.

In the rest of the paper, we first present related work on job schedulers in Section II. Then, we present the details our integer programming based scheduler in Section III. Implementation details of the SLURM plug-in we developed are given in Section IV. The plug-in can be downloaded at http://code.google.com/p/slurm-ipsched. To test the performance of our plug-in, we preferred to carry out realistic direct SLURM emulation tests rather than use a simulator. The details of the benchmark tests and the results obtained are given in Sections V and VI respectively. We conclude the paper with a discussion in Section VII.

2. Survey of Related Work

Several job schedulers are currently available. An excellent and in-depth assessment of job schedulers was carried out by Georgiou [6] in his PhD thesis. We briefly review the most widely used systems on clusters and supercomputers.

PBSpro [7] is a commercial scheduler which descended from the PBS system originally developed in NASA. In addition to the professional PBSpro, an unsupported original open source version called OpenPBS is also available. PBSPro has the usual scheduler features: FCFS, backfilling, fair share, preemption, multi-priority, external scheduler support, advanced reservation support and application licenses. Recently, GPU scheduling support has been introduced by providing two approaches: (i) simple approach in which only one GPU job at a time is run on any given node exclusively and (ii) advanced distributed approach which is needed when sharing of the nodes by multiple jobs at the same time is required or if individual access to GPUs by device number is required.

MOAB [2] is a commercial job scheduler that originated from the PBS system. It supports the usual FCFS, backfilling, fair share, preemption, multi-priority, advanced reservation and application licenses. MOAB is just a scheduler and hence it needs to be coupled with a resource manager system.

TORQUE [8] is a resource management that is the open-source version of PBSPro. Similarly, MAUI [3] is a scheduler that is open-source version of the MOAB. MAUI supports FCFS, backfilling, fair-share, preemption, multi-priority, advanced reservation and application licenses. TORQUE also has some GPU support. Jobs can specify their request for the GPUs but it is left up to the job's owner to make sure that the job executes properly on the GPU.

LSF [4] is a commercial scheduler which descended from the UTOPIA system. Like the other schedulers, it supports FCFS, backfilling, fair-share, preemption, multi-priority, advanced reservation and application licenses. Newer features include live cluster reconfiguration, SLA-driven scheduling, delegation of administrative rights and GPU-aware scheduling. LSF can also be fed with thermal data from inside of the servers so that it can try to balance workloads across server nodes so as to minimize hot areas in clusters.

LoadLeveler [9] is a commercial product from IBM. It was initially based on the open-source CONDOR system [10]. It supports FCFS, backfilling, fair-share, preemption, multi-priority, advanced reservation and application licenses. It has a special scheduling algorithm for the Blue Gene machine which extends LoadLeveler's reliable performance.

OAR [11] is a recently developed open source resource management system for high performance computing. It is the default resource manager for the Grid500 which is a big real-scale experimental platform where computer scientists can run large distributed computing experiments under real life conditions. OAR has been mostly implemented in Perl. In addition to the above systems, Condor [10] and Oracle Grid Engine [12] (formerly known as Sun Grid Engine) systems are also widely used, especially in grid environments.

Since heterogeneous CPU-GPU systems have appeared very recently, we do not expect that the aforementioned commercial or open source job schedulers have done much for optimizing scheduling of such systems. For example, SLURM has introduced support for GPUs but its scheduling algorithms were not optimized for GPUs as we illustrated with the example in Table 1. In fact, in SLURM, if the number of nodes (with -N option) is not specified and just the number of cores (with -n) are given together with the number of GPUs per node (with --gres=gpu:), then this can actually lead to different total number GPUs to be allocated in different runs of the same job (since the number of GPUs allocated depends on the number of nodes over which the requested cores are allocated).

In this paper, we employ integer programming (IP) techniques in order to get the assignment of jobs to the resources. IP techniques have been used for scheduling before. For example, [13] used IP techniques for devising a dynamic voltage frequency scaling (DVFS) aware scheduler. [14] solved a relaxed IP problem to implement collective match-making heuristics for scheduling jobs in grids. In this paper, we contribute an IP formulation for node based CPU-GPU systems and implement a SLURM plug-in for it. We are not aware of any IP based scheduling plug-in that has been developed for SLURM.

3. Scheduling Algorithm

Our approach for scheduling involves taking a window of jobs from the front of the queue and solving an assignment problem that maps jobs to the available resources. Figure 1 depicts our scheduling approach. This window based assignment step is basically repeated periodically. Note that the assignment problem that needs to be solved here involves co-allocation, i.e. multiple CPUs and GPUs need to be co-allocated). Therefore, we expect this problem to be NP-hard (perhaps by a transformation from the subset sum problem).

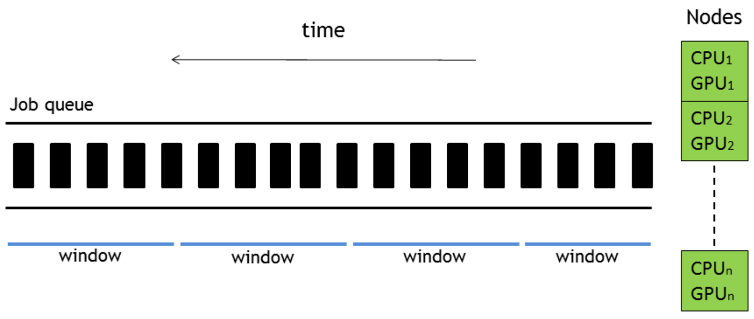

Figure 1. Scheduling windows of jobs

To formulate the assignment problem as an IP, we first make the following definitions:

- s_j: binary variable showing whether job j is selected or not,
- p_j: priority of job j,
- c_j: node packing variable for a job,
- r_j: no. of CPUs requested by job j,
- g_j: no. of GPUs requested by job j,
- x_{ij}: no. of CPUs allocated to job j at node i,
- R_i: available no. of CPUs at node i,
- G_i: available no. of GPUs at node i,
- t_{ij}: binary variable showing whether job j is allocated any resource on node i,
- $N_{min,j}$: minimum no. of nodes that job j requests,
- $N_{max,j}$: maximum no. of nodes that job j requests,
- N: no of nodes on the system,
- M: no. of of jobs in the window.

The IP formulation is then given as follows:

$$max \sum p_j(s_j - c_j)$$

$$\textstyle\sum_j^M x_{ij} \leq R_i \;\; \forall i \tag{1}$$

$$\textstyle\sum_i^N x_{ij} = r_j s_j \;\; \forall j \tag{2}$$

$$\textstyle\sum_j^M g_j t_{ij} \leq G_i \quad \forall i \tag{3}$$

$$c_j = \frac{\sum_i^N t_{ij}}{2N} \quad \forall j \tag{4}$$

$$N_{min,j} \leq 2Nc_j \leq N_{max,j} \quad \forall j \tag{5}$$

$$t_{ij} = \begin{cases} 1, & x_{ij} > 0 \\ 0, & x_{ij} = 0 \end{cases} \quad \forall i,j \tag{6}$$

We can view this problem as a variation of the knapsack problem [15], where each job has its utilization value as its priority, and its CPU and GPU requests as its sizes. The objective is to maximize the sum of selected jobs' priorities. We also want to reduce the number of nodes over which a selected job's allocated resources are spread. To specify this preference, in constraint (4) we compute node packing factor c_j for a job which is the ratio of the number of nodes over which a selected job's allocated resources are spread to twice the total number of nodes. Note that c_j is always less than or equal to half. We then subtract c_j from s_j in the objective function in order to disfavor solutions that spread allocated resource over large number of nodes. Constraint (1) sets the limits on number of processes that can be assigned to a node, (2) sets a job's total number of allocated CPUs on the nodes to either 0 if the job is *not* selected, or to the requested r_j number of CPUs if it is selected. If a job is allocated some CPUs on node *i*, it also should be allocated GPUs on that node. This is controlled by constraint (3). If a job requests that the number of nodes it gets is between some limits, these limits are enforced by (5). If these limits are not defined, $N_{min,j}$ is set to 1, and $N_{max,j}$ is set to a large value. Constraint (6) sets the binary variables t_{ij} which indicate whether job *j* is allocated any resource on node *i*.

The number of variables and the number of constraints of the IP formulation given in terms of the number of jobs (*M*) and the number of nodes (*N*) are shown in Tables 2 and 3.

Table 2. Number of Variables in the IP formulatıon

Variable name	***No. of Variables***
s_j	*M*
c_j	*M*
x_{ij}	*M*N*
t_{ij}	*M*N*
Total	*2M*(1 + N)*

Table 3. Number of Constraints in the IP formulatıon

Constraint	***No. of Constraints***
(1)	*N*
(2)	*M*
(3)	*N*
(4)	*M*
(5)	*2*N*
(6)	*2*M*N*
Total	*2*(M + 2*N + MN)*

4. Development of a SLURM Plug-in

SLURM employs a central controller daemon (slurmctld) that runs on a central or a management node and daemons (slurmd) that run on each compute node. Each slurmd daemon interacts with the controller daemon, and passes information about the job currently running on that node and the node's status [1].

SLURM has been designed as a light-weight system. In SLURM, almost everything is handled by plugins. SLURM provides several plug-ins such as scheduler, resource selection, topology and priority plug-ins. The plug-in which we focus on is the scheduler plug-in which works on the controller daemon.

SLURM's backfill scheduler creates a priority queue, and collects the resource requirements of each job in the queue. The scheduler tries to start a job by taking into consideration the reservations made by other higher priority jobs also. If it can fit a job on the nodes with available resources, the job is started; otherwise, it is scheduled to be started later [16].

The selection of the specific resources is handled by a plug-in called resource selection. SLURM also provides a resource selection plug-ins called linear that allocates whole nodes to jobs and consumable resources that allocates individual processors and/or memory to jobs [16]. In this work, a plug-in called IPCONSRES was designed, which is a variation of SLURM's own cons_res plug-in. IPCONSRES allows our IP based scheduler plug-in (which we named IPSCHED) to make decisions about the node layouts of the selected jobs.

The priorities of the jobs are calculated by SLURM. Our plug-in does not change these values, but only collects them while when an assignment problem is solved. In this work, two types of test runs were made. In the first type of runs, priority type was selected as basic in SLURM. This corresponds to a first-come-first-served scheduling. SLURM takes the first submitted job's priority to be a large number, and every arriving job's priority will be one less than that of the previously arrived job. In the second type of runs, SLURM's multifactor priority plug-in was used. This plug-in allows the job priorities to increase with increasing job size and aging. Multifactor priority is calculated using parameters such as the age, the job size, fair-share partition and QOS. The settings of priority factors used in this work are explained in Section VI, namely the Results section.

In SLURM, GPUs are handled as generic resources. Each node has a specified number of GPUs (and other generic resources if defined) dedicated to that node. When a user submits a job that uses GPUs, he has to state the number of GPUs per node that he requests. The GPU request can be 0 if the job is running only on CPU.

The IPSCHED plug-in has been designed in such a way that only one small change needs to be made to the common files used by SLURM. This change which is located in SLURM's common/job_scheduler.c file disables the FIFO scheduler of SLURM; thus allowing each job to be scheduled by our IPSCHED plug-in. Even if the queue is empty and the nodes are idle, a submitted job will wait a certain number of seconds (named as SCHED_INTERVAL) and run when it is scheduled by our IPSCHED plug-in.

IPSCHED plugin runs every SCHED_INTERVAL seconds. By default this value is 3 and can be changed in the SLURM configuration file. IPSCHED solves the IP problem we defined in Section III to decide which jobs are to be allocated resources.

The workings of our plug-in are explained in the pseudo-code given in Figure 2. The plug-in first gets a window of the priority ordered jobs. The number of jobs in the window is limited by MAX_JOB_COUNT variable (by default this is set to 200 but can be changed using SLURM configuration file). The plug-in also collects the job priority, number of CPUs and GPUs requested and store these in the pj, rj and gj arrays. Afterwards, it obtains the "empty CPU" and "empty GPU" information from all the nodes and stores these values in Ri and Gi arrays. The Ri, Gi, pj and rj values are then used to create the integer programming problem, which is solved using CPLEX [17].

```
IPSCHED Algorithm

1. Generate priority ordered job window of size up to MAX_JOB_COUNT
2. From each job in the window collect
    a. priority ($p_j$)
    b. CPU request ($r_j$)
    c. GPU request ($g_j$)
    and store these into job_list array.
3. From each node collect
    a. number of empty CPU's
    b. number of empty GPU's,
    and store these into the node_info array.
4. Form the IP problem
5. Solve the IP problem and get $s_j$ and $x_{ij}$ values.
6. For jobs with $s_j = 1$, set job's process layout matrix and start the job by:
    a. For each node i, assign processors on that node according to $x_{ij}$
    b. Start the job, no more node selection algorithm is necessary.
```

Figure 2. IPSCHED scheduling steps

The SCHED_INTERVAL is also used to set a time limit during the solution of the IP problem. If the time limit is exceeded, in that scheduling interval, none of the jobs are started, and the number of jobs in the window is halved for the next scheduling interval. In step 6, a layout matrix which has nodes as columns and selected jobs as rows is used to show mapping of jobs to the nodes. This matrix is used to assign and start the job on its allocated nodes.

Note that IPSCHED can handle a job's minimum/maximum and processor/node requests but it assumes that the jobs do not have explicit node requests. Such job definitions may break up the system, since the selection of jobs is made based only on the availability of resources on the nodes.

Finally, we also note that our IPSCHED plug-in has been developed and tested on version 2.3.3 of SLURM, which is the latest stable version.

5. Testing with ESP benchmarks

To test the performance of our IPSCHED plug-in, we use workloads derived from ESP benchmarks [18] (to be more precise, derived from version 2, i.e. ESP-2). The goal of the ESP is to test the performance of a scheduler under heavy workloads. The (normalized) job size (no. of cores requested by job), count and execution time characteristics of the ESP workloads are given in Table 4. The sum of the completion times of the original ESP jobs is 10984 seconds (3.05 hours). ESP jobs are submitted randomly with inter-arrival times that have a Gaussian distribution. In the original ESP test, when Z-jobs (full configuration jobs) are submitted, no other jobs are allowed to run. However, we have removed this rule as was done by [6] and configured Z-jobs similar to the other jobs in the workload.

Table 4. ESP benchmark job characteristics

Job	***Size***	***Count***	***Execution time (s)***
A	0.03125	75	257
B	0.06250	9	341
C	0.50000	3	536
D	0.25000	3	601
E	0.50000	3	312
F	0.06250	9	1846
G	0.12500	6	1321
H	0.15820	6	1078
I	0.03125	24	1438
J	0.06250	24	715
K	0.09570	15	495
L	0.12500	36	369
M	0.25000	15	192
Z	1.00000	2	100

In ESP, initially 50 jobs are released. Afterwards, A through M jobs are submitted with a mean inter-arrival time of 30 seconds, and the two Z jobs are submitted after all other jobs are submitted at the 80th and 240th minute.

ESP jobs request just CPU cores. They also do not specify the number of nodes request. Since our work has been designed for heterogeneous CPU-GPU systems, we have modified the ESP job set. In our tests, we emulated a system that had 1024 nodes, each of which had 2 GPU's and 8 cores. To prepare a CPU-GPU workload, we simply duplicated all the ESP jobs with the exception of Z-jobs. One copy requested CPU-cores only (as in the original ESP) and its corresponding copy requested CPU-cores as well as 2 GPUs per node. Since we have doubled all of the jobs other than the Z-jobs in the workload, we have also doubled the earliest time to submit Z-jobs [19]. In our modified ESP workload set that resulted, there were a total of 458 jobs, and the sum of completion times was 21822 seconds (6.06 hours).

Our ESP based CPU-GPU workloads were tested on a real SLURM system. SLURM was operated in emulation mode. In this mode, jobs were submitted as sleep jobs. Even though testing by actual SLURM emulation takes a long time and hence limits the number of tests we carry out, it has the advantage that, we can observe the real behavior of SLURM together with its overheads that would otherwise not be possible if simulations were carried out.

Table 5 summarizes the parameters of the various tests we have performed. The objective and priority functions were changed for both our IPSCHED plug-in and SLURM's backfill plug-in. In the multifactor priority experiments, job age factor weight and job size factor weight was taken as equal, and for each minute a job spends in the queue, its priority was increased by one. In this work, no fair share policy and QoS was used.

Each experiment has been repeated three times and mean and standard deviation of waiting times, slowdown ratios and utilization metrics have been tabulated in Table 6.

Table 5. Tests and their characteristic

Experiment	***Plugin***	***Priority Type***	***Objective Function***
1-SLURM	Backfill	Basic	-
2	IPSCHED	Basic	$max \sum p_j r_j (s_j - c_j)$
3	IPSCHED	Basic	$max \sum p_j (s_j - c_j)$
4-SLURM	Backfill	Multifactor	-
5	IPSCHED	Multifactor	$max \sum p_j (s_j - c_j)$
6	IPSCHED	Multifactor	$max \sum p_j r_j (s_j - c_j)$

Table 6. Results of the tests

Experiment	***Waiting Time (mean ± std)***	***Slowdown Ratio (mean ± std)***	***Utilization (mean)***
1-SLURM	1.60 ± 0.836	18.11 ± 25.49	0.90
2	0.91 ± 1.224	11.31 ± 18.97	0.93
3	0.77 ± 1.257	9.95 ± 18.87	0.92
4-SLURM	2.42 ± 1.758	22.75 ± 22.02	0.89
5	0.88 ± 1.223	10.75 ± 18.20	0.94
6	2.08 ± 1.577	22.06 ± 21.84	0.93

We also note that since we have emulated 1024 nodes with 8 cores, and the smallest of the ESP jobs require 256 cores (8192 * 0.03125), these tests were not indicative of the IP node packing variable c_j, which was used to enforce better packing in the IPSCHED plug-in. This is because CPU core requirements of all the jobs were a multiple of 8. As a result, no node was assigned more than one job in the tests given in Table 5. Therefore, another set of experiments were performed, where ESP job size characteristic were modified to be SYSTEM_SIZE * JOB_FRACTION + RAND(-4,4). In this way, on average, the number of resources a job requests will be roughly equal to that of the original ESP, but not be multiples of 8 so that the packing effectiveness can be tested. We also calculate a ratio called the packing factor for a job. This is the ratio of the actual number of allocated nodes to the minimum number of nodes that could be allocated. A packing factor value that is close to unity for a job, means that smaller communication cost will result.

Experiment 7 and 8 are modified versions of experiment 4 and 6 respectively. Since ESP jobs do not have the concept of node request, IP constraint (5) is disabled during these tests, because, by ESP job set's definition, the jobs request certain number of processors, but not nodes. We also note that we use modified version 6 rather than 5 in order to favor big jobs.

Table 7. Results List With Randomly Modified ESP Job Sizes

Experiment	***Waiting Time (mean ± std)***	***Slowdown Ratio (mean ± std)***	***Utilization (mean)***	***Packing factor (mean± std)***
7-SLURM	2.50 ± 1.802	23.80 ± 22.93	0.88	1.021 ± 0.029
8	1.09 ± 1.111	13.12 ± 16.78	0.94	1.016 ± 0.026

The tests results shown in Tables 6 and 7 show that our IPSCHED produces better waiting times, slowdown ratios and utilizations. Figure 3, in particular, shows that shorter waiting times obtained in

IPSCHED are because of favoring of smaller sized jobs. SLURM seems to favor large sized jobs and hence on average it produces longer waiting times. Work carried out by [20] suggested that large sized jobs should be favored. We disagree with the suggestion in [20] because to draw such a conclusion requires knowing about the nature of jobs that are submitted to a system. There can be widely used applications that may best run on small number of processors. There can also be applications that may have components that run on massive number of processors as well as components that may require small number of processors. Such jobs can be submitted as multiple jobs or workflows to the system so as to use a supercomputer in an efficient way and with the expectation to get faster responses. We believe that the size of a job by itself cannot dictate the importance of that job. We also believe that it is quite natural for a user to get faster response when he requests less resources. Therefore, providing schedulers that deliver the expected behavior is more important especially if they can deliver the same or better levels of efficiency.

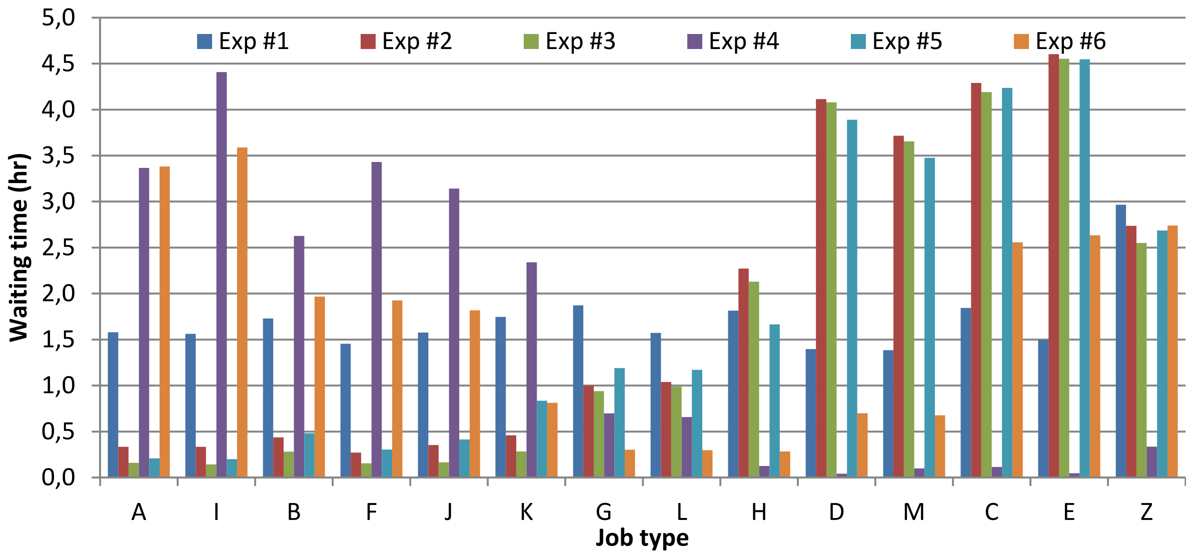

Figure 3. Comparison of waiting times in different scheduling mechanisms

7. Discussion and Conclusions

SLURM is a popular open source resource management system. In this work, our main aim has been to develop a plug-in that provides integer programming based job scheduling. Depending on the problem, integer programming packages like CPLEX can now solve IPs with variable numbers in the thousands range. We have also devised an IP formulation to schedule windows of jobs. The number of jobs in a window was chosen in such a way that CPLEX would be able to handle the IP problem with thousands of variables in a reasonable time. Our scheduling interval was 4 seconds and CPLEX was able to solve the IP on average in 3 seconds. We also note that since integer programming solvers provide a general purpose optimization platform, our IPSCHED plug-in can also be used as a template to easily implement other integer programming based plug-ins that address other issues such as energy aware scheduling or simply other scheduling formulations.

To test our IPSCHED, we have used a modified form of the popular ESP workloads. We note, however, that ESP workloads have been designed for a homogeneous system with CPU cores and we modified it in an ad-hoc manner to come up with a heterogeneous CPU-GPU workload. The results obtained from the SLURM emulation experiments showed that our IPSCHED outperformed SLURM in terms of mean waiting times, slowdown ratio and utilization. We believe that as more realistic CPU-GPU workloads with node usage specifications are produced in the future, we expect the performances of IPSCHED and SLURM's own plug-ins to have wider gaps.

As future work, we plan to do the following: Firstly, we would like to incorporate a fair share policy into our IP formulation. Secondly, we would like to incorporate a more advanced topology awareness formulation. Currently, we assumed that all the nodes are at equal distance from each other. We plan to design IP formulations that can take into consideration the topology of the network switches. Such IP formulations, however, will have excessive number of variables that will make it difficult to solve in a reasonable amount of time. To resolve this difficulty, relaxed versions of the IP can be solved with a linear programming package and integer solutions could then be attempted to obtain heuristically from the relaxed solution.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528 and FP7-261557. We thank Matthieu Hautreux from CEA, France, for discussions and his help during the implementation of our plug-in. The authors have used the Bogazici University Polymer Research Center's cluster to carry out the tests in the paper..

References

[1] B. Yoo, M. A. Jette and M. Grondona, “SLURM: Simple Linux Utility for Resource Management”, in Job Scheduling Strategies for Parallel Processing, Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, Eds., pp. 44–60. Springer Verlag, 2003, Lect. Notes Comput. Sci. vol. 2862.

[2] Moab Workload Manager Documentation, http://www.adaptivecomputing.com/resources/docs/

[3] Bode, D.M. Halstead, R. Kendall, Z. Lei, and D. Jackson, “The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters,” in Proceedings of the 4th Annual Showcase and Conference (LINUX-00), Berkeley, CA, 2000, pp. 217–224, The USENIX Association.

[4] LSF (Load Sharing Facility) Features and Documentation, http://www.platform.com/workload-management/high-performance-computing

[5] SLURM User Group Meeting 2010, http://sc10.supercomputing.org/schedule/event_detail.php?evid=bof115

[6] Y. Georgiou, “Resource and Job Management in High Performance Computing”, PhD Thesis, Joseph Fourier University, France, 2010.

[7] Nitzberg , J. M. Schopf , J. P. Jones, “PBS Pro: Grid computing and scheduling attributes, Grid resource management: state of the art and future trends”, Kluwer Academic Publishers, Norwell, MA, 2004

[8] Torque Resource Manager Documentation, http://www.adaptivecomputing.com/resources/docs/

[9] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skovira, “Workload Management with LoadLeveler”. IBM, first ed., Nov 2001. ibm.com/redbooks

[10] T. Tannenbaum, D. Wright , K. Miller , M. Livny, “Condor: a distributed job scheduler”, Beowulf cluster computing with Linux, MIT Press, Cambridge, MA, 2001

[11] N. Capit, G.D. Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard, “A batch scheduler with high level components,” in 5th Int. Symposium on Cluster Computing and the Grid, Cardiff, UK, 2005, pp. 776–783, IEEE.

[12] W. Gentzsch, “Sun Grid Engine: Towards Creating a Compute Power Grid,” in Proc. First IEEE International Symposium on Cluster Computing and the Grid (1st CCGRID’01), Brisbane, Australia, May 2001, pp. 35–39, IEEE Computer Society (Los Alamitos, CA).

[13] M. Etinski, J. Corbalan, J. Labarta, M. Valero, "Linear programming based parallel job scheduling for power constrained systems," High Performance Computing and Simulation (HPCS), 2011 International Conference on , vol., no., pp.72-80, 4-8 July 2011.

[14] O. Erbas, C. Ozturan, Collective Match-Making Heuristics for Grid Resource Scheduling, HiPerGRID - High Performance Grid Middleware , Brasov, Romania, Sep. 2007.

[15] S. Martello, T. Paolo, “Knapsack problems: Algorithms and computer interpretations”. Wiley-Interscience. Chichester (1990).

[16] SLURM Documentation, http://www.schedmd.com/slurmdocs/

[17] Cplex Optimization, Inc, “Using the CPLEX Callable Library”. Incline Village, NV 89451-9436, 1989-1994.

[18] A.T. Wong, L. Oliker,W.T.C. Kramer, T.L. Kaltz, D.H. Bailey, “ESP: A System Utilization Benchmark,” in SC2000: High Performance Networking and Computing. Dallas Convention Center, Dallas, TX, USA, November 4–10, 2000, ACM, Ed., pub-ACM:adr and pub-IEEE:adr, 2000, pp. 52–52, ACM Press and IEEE Computer Society Press.

[19] A.T. Wong, L. Oliker, W.T.C. Kramer, T.L. Kaltz, D.H. Bailey, “ESP: A system utilization benchmark.” In: IEEE/ACM Supercomputing, Dallas, TX, November 2000, pp. 52–52 (2000)

[20] P. Andrews, P. Kovatch, V. Hazlewood, T.Baer, Scheduling a 100,000 Core Supercomputer for Maximum Utilization and Capability, 39th International Conference on Parallel Processing Workshops (ICPPW), 2010.