Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

An Auction Based SLURM Scheduler for Heterogeneous Supercomputers and its Comparative Performance Study

Seren Soner[a], Can Özturan[a,*]

[a]Computer Engineering Department, Bogazici University, Istanbul, Turkey

Abstract

SLURM is a resource management system that is used on many TOP500 supercomputers. We present a heterogeneous CPU-GPU scheduler plug-in, called AUCSCHED, for SLURM that implements an auction based algorithm. In order to tune the topological mapping of jobs to resources, our plug-in determines at scheduling time, for each job, the best resource choices based on node contiguity from available ones. Each of these choices is then expressed as a bid that a job makes in an auction. Our algorithm takes a window of jobs from the front of the job queue, generates multiple bids for available resources for each job, and solves an assignment problem that maximizes an objective function involving priorities of jobs. We generate several CPU-GPU synthetic workloads and perform realistic SLURM emulation tests to compare the performance of our auction based scheduler with that of SLURM's own back-fill scheduler. In general, AUCSCHED has a few percentage points of better utilization over SLURM/BF plug-in but topologically SLURM/BF is leading to less fragmentation whereas AUCSCHED is leading to less spread. SLURM's as well as our plug-in produce high utilizations around 90% when workloads are made up of jobs requesting no more than 1 GPU per node. On the other hand, when workloads contain jobs that request 2 GPUs per node, it is observed that the system utilization drops drastically to the 65-75% range both when our AUCSCHED and SLURM's own plug-in are used. This points to the need to further study of scheduling jobs that utilize multiple GPU cards on nodes. Our plug-in which builds on our earlier plug-in called IPSCHED is available at http://code.google.com/p/slurm-ipsched/.

Project ID:

1. Introduction

SLURM [1] is a GPL licensed open resource management system that is used on many TOP500 supercomputers. It is estimated by SLURM developers that as many as 30% of the supercomputers in the November 2012 TOP500 list are using SLURM [2]. In particular, it is stated that one third of the 15 most powerful systems in this list use SLURM. These are: No. 2 Sequoia at LLNL, No. 7 Stampede at TACC, No. 8 Tianhe-1A in China, No. 11 Curie at the CEA in France and No. 15 Helios at Japan's International Fusion Energy Research Centre. Our work contributes a heterogeneous CPU-GPU scheduler plug-in, called AUCSCHED, for SLURM that implements an auction based algorithm. This plug-in, which builds on our earlier plug-in called IPSCHED [3], is available at http://code.google.com/p/slurm-ipsched/.

If a job is allocated some resources, then depending on what resources it has been assigned, the job itself may perform tuning to achieve better performance on these resources, for example, by using topologically aware communication. A complementary tuning can also be performed by the scheduler of a resource manager by helping a job to achieve better runtime performance by placing it on resources that will lead to faster execution. Such may be the case, for example, if a communication intensive job is allocated nodes that are in close vicinity to each other. Since a scheduler has access to information about available resources and is the authority that makes allocation decisions, it can enumerate and consider alternative candidate resource allocations for each job. This work considers this complementary approach that aims to tune allocations of jobs at the scheduling level. The work is carried out within the context of linear mapping of jobs to one-dimensional array of nodes. This is the default mode of resource selection in SLURM. Slurm documentation [4] states the following about

*Corresponding author.
tel. +90-212-359-7225 fax. +90-212-387-2461 e-mail. ozturaca@boun.edu.tr

this mode: *"SLURM's native mode of resource selection is to consider the nodes as a one-dimensional array. Jobs are allocated resources on a best-fit basis. For larger jobs, this minimizes the number of sets of consecutive nodes allocated to the job."*

Table 1: SLURM submission options covered in AUCSCHED

Option	Explanation
`-n`	number of cores
`-N`	number of nodes
`--ntasks-per-node`	number of cores per node
`--gres=`Xgpu	X gpu's per node
`--contiguous`	contiguous node allocation

Table 1 lists the SLURM options that our plug-in AUCSCHED supports. These are namely, the number of cores, the number of nodes, the number of cores per node and the number of GPUs per node that can be requested by a job and the option that states whether the allocation is explicitly requested to be contiguous.

Our proposed methodology can be summarized as follows: Our algorithm takes a window of jobs from the front of the job queue, generates multiple bids for available resources for each job, and solves an assignment problem that maximizes an objective function involving priorities of jobs. To achieve a topologically aware mapping of jobs to processors, the bids generated include requests for contiguous allocations. We generate various CPU-GPU synthetic workloads and perform realistic SLURM emulation tests to compare the performance of our auction scheduler with that of SLURM's own back-fill scheduler.

The whitepaper is organized as follows: In Section 2, we first survey the related work. In Section 3, we present a mathematical formulation of the allocation problem as an auction problem in which jobs' bid for resources. The details of bid generation process are given in Sections 4 and 5. In Section 6, we first present details of a synthetic CPU-GPU system workload generator we developed to generate workloads. Then, we show the results of scheduling these workloads using SLURM backfill scheduler (SLURM/BF) and our AUCSCHED. The whitepaper is concluded with a discussion of results and future work in Section 7.

2. Related work

Several job schedulers are currently available. An excellent and in-depth assessment of job schedulers was carried out by Georgiou [5] in his PhD thesis. In their recent work, Georgiou and Hautreux [6] evaluated scalability and efficiency of SLURM resource and job management system on large HPC Clusters. In our earlier work [3], we developed a SLURM plug-in called IPSCHED which utilized an integer programming formulation to do assignment of a window of jobs to the available resources. IPSCHED just aimed at packing as many cores as possible on a node but did not attempt to achieve node contiguity of allocations. In the current work, AUCSCHED employs a bid mechanism and tries to allocate contiguous blocks of nodes. The solution of the optimization problem is again obtained by using an integer programming package. The auction mechanism we implement for SLURM has some similarities to the previous work we did on multi-unit nondiscriminatory combinatorial auctions [7].

Work on contiguous node allocation of jobs within the context of first-come-first served with backfilling policy on k-ary n-tree networks have been carried out by [8]. In this work, non-contiguous, contiguous and a relaxed version of contiguous, called, quasi-contiguous allocations of jobs were studied by performing simulations. It was reported that contiguous allocations resulted in severe scheduling inefficiency due to increased system fragmentation. Their proposed quasi-contiguous approach reduced this adverse effect. Their simulations were carried out by using the INSEE simulator [9]. Their work did not address CPU-GPU scheduling and used workloads from the Parallel Workload Archive [10].

3. Formulation of the Auction Problem

Given a window of jobs from the front of the job queue, bid generation phase (explained in Sections 4 and 5) generates a number of bids for available resources for each job, and solves an allocation problem that maximizes an objective function involving priorities of jobs. The allocation problem is solved by formulating it as an integer programming (IP) problem. The IP problem is solved using the CPLEX [11] solver. Table 2 shows the list of symbols and their meanings. The objective and the constraints of the optimization problem are formulated as follows:

$$Maximize \quad \sum_{j \in J} \sum_{c \in B_j} (P_j + \alpha \cdot F_{jc}) \cdot b_{jc} \tag{1}$$

Table 2: List of main symbols, their meanings, and definitions

Symbol	Meaning
J	Set of jobs that are in the window: $J = \{j_1, \ldots, j_{\|J\|}\}$
P_j	Priority of job j
N	Set of nodes : $N = \{n_1, \ldots, n_{\|N\|}\}$
C	Set of bid classes : $C = \{c_1, \ldots, c_{\|C\|}\}$
N_c	Set of nodes making up a class c
B	Set of all bids, $B = \{b_1, \ldots, b_{\|B\|}\}$
B_j	Set of bid classes on which job j bids, i.e. $B_j \subseteq C$
C_{jn}	The set $\{c \in C \mid c \in B_j \ \ and \ \ n \in N_c\}$
A_n^{cpu}	Number of available CPU cores on node n
A_n^{gpu}	Number of available GPUs on node n
R_{jc}^{cpu}	Number of cores requested by job j in bid c
R_j^{gpu}	Number of gpus per node requested by job j
R_{jc}^{node}	Number of nodes requested by job j in bid c
R_j^{cpn}	Number of cores per node requested by job j. If not specified, this parameter gets a value of 0.
F_{jc}	Preference value of bid c of job j in the interval $(0, 1]$. This is used to favor bids with less fragmentation.
α	A factor multiplying the preference value F_{jc} so that the added preference values do not change the job priority maximizing solution (See Equation 9).
b_{jc}	Binary variable for a bid on class c of job j.
u_{jn}	Binary variable indicating whether node n is allocated to job j
r_{jn}	Non-negative integer variable giving the remaining number of cores allocated to job j on node n (i.e. at most one less than the total number allocated on a node).

subject to constraints :

$$\sum_{c \in B_j} b_{jc} \leq 1 \ for \ each \ j \in J \tag{2}$$

$$\sum_{n \in N_c} u_{jn} = b_{jc} \cdot R_{jc}^{node} \quad for \ each \ (j,c) \in J \times C \ s.t. \ c \in B_j \tag{3}$$

$$\sum_{n \in N_c} \sum_{c \in B_j} u_{jn} + r_{jn} = \sum_{c \in B_j} b_{jc} \cdot R_{jc}^{cpu} \ for \ each \ j \in J \tag{4}$$

$$\sum_{j \in J} u_{jn} + r_{jn} \leq A_n^{cpu} \ for \ each \ n \in N \tag{5}$$

$$\sum_{j \in J} u_{jn} \cdot R_j^{gpu} \leq A_n^{gpu} \ for \ each \ n \in N \tag{6}$$

$$0 \leq r_{jn} \leq u_{jn} \cdot min(A_n^{cpu} - 1, R_{j,max}^{cpu} - 1) \quad for \ each \ (j,n) \in J \times N \tag{7}$$

$$u_{jn} + r_{jn} = \sum_{c \in C_{jn}} b_{jc} \cdot R_j^{cpn} \quad for \ each \ (j,n) \in J \times N \ s.t. \ R_j^{cpn} > 0 \ and \ C_{jn} \neq \emptyset \tag{8}$$

Our objective as given by equation 1 is to maximize the summation of selected bids' priorities. In case there are multiple solutions that maximize the summation of priorities P_j, (i.e. the value of $\sum_{j \in J} \sum_{c \in B_j} P_j \cdot b_{jc}$) an additional positive contribution $\alpha \cdot F_{jc}$ is added to the priority in order to favour bids with less fragmentation. Since we do not want the added contributions to change the solution that maximizes the summation of priorities, we can choose this contribution as follows:

$$P_{min} > \alpha \cdot (|B| + 1) > \sum_{j \in J} \sum_{c \in B_j} \alpha \cdot F_{jc} \tag{9}$$

We generate a bid preference value F_{jc} in the interval $(0, 1]$ as explained in Section 5 and choose α as follows so as to satisfy the inequality 9:

$$\alpha = \frac{P_{min}}{|B| + 1} \tag{10}$$

Constraint 2 ensures that at most one of the bids of a job can be selected in a solution. Constraint 3 makes sure that the number of nodes requested by a job is allocated exactly if the corresponding bid that requests the nodes is selected. The left hand side in this constraint gives the number of nodes allocated. The right hand hand side becomes equal to the number of nodes requested by bid for class c of job j if the bid variable b_{jc} is set to 1. Note that for some jobs R_{jc}^{node}, the number of nodes requested, can be explicitly stated using the `-N` SLURM option. If this option is not given, then it can be set by the bid generator ; since each bid requests specific nodes, the number of nodes requested is known for the bid. Constraint 4 makes sure the total number of CPU cores is equal to the requested number if the bid for class c of job j is selected. Constraint 5 sums up allocated cores on a node and makes sure the number of CPU cores allocated does not exceed what is available. Constraint 6 does the same thing for the GPUs, i.e. checks whether number of CPU cores allocated does not exceed the available number of GPUs. Constraint 7 ensures that for a selected bid, we do not have the case where $r_{jn} > 0$ and $u_{jn} = 0$; in other words, if cores are allocated on a node, then we should have $u_{jn} = 1$ and the remaining number should be assigned to r_{jn}. The final constraint, i.e. 8, is generated for jobs for which SLURM's cores per node option (`--ntasks-per-node`) is specified.

4. Nodeset and Bid Class Generation

In order to generate bids for each job, we need to scan the 1D array of nodes in order to look for nodes that have enough number of available cores and/or gpus. It may be costly to do the scanning for each bid, especially, since we are going to generate multiple bids for each job. For this reason, the so called *nodesets* are first created. Nodesets are basically contiguous block of nodes that have at least certain numbers of cores and GPUs in each of the nodes and a certain total number of cores in the block. Formally it is defined as a as a 4-tuple (n_i, n_j, c, g) such that the following holds:

$$A_n^{cpu} > 0 \quad \forall n \in \{n_i, \ldots, n_j\}$$
$$A_n^{gpu} \geq g \quad \forall n \in \{n_i, \ldots, n_j\}$$
$$\sum_{n \in \{n_i, \ldots, n_j\}} A_n^{cpu} = c$$

Here, n_i and n_j are the first and the last nodes respectively in the nodeset, c is the total number of cores in the nodeset, and g is the number of GPUs per node in the nodeset. Figure 1 illustrates nodesets on a small 12 node system. Suppose that the topmost 1D array shows the numbers of available *cores/GPUs* on the system. The eight nodesets constructed are shown in the figure.

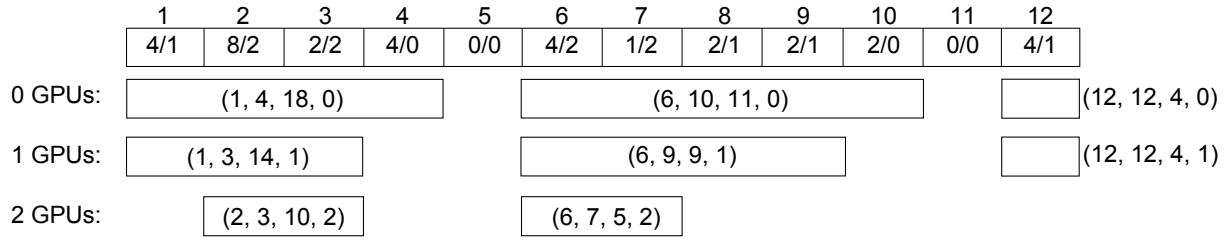

Fig. 1: Determination of nodesets for a 12-node system

A bid class is basically a set of nodes. A bid of a job is an instantiation of a bid class. If multiple jobs have bids of the same class, this means they are bidding for the same set of nodes. From nodesets, bid classes are

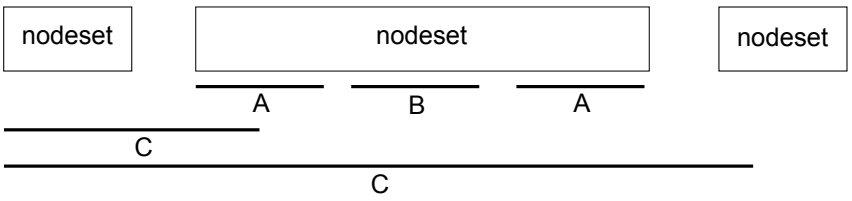

Fig. 2: Bid class generation example

generated. Using these nodesets, the bid generation is made according to the resource requests of jobs. A job's resource requests can be classified into the following types:

1. Only total number of cores (R_j^{cpu}) is specified.
2. Number of nodes (R_j^{node}) , total number of cores (R_j^{cpu}) and number of GPUs per node (R_j^{gpu}) are specified.
3. Number of nodes (R_j^{node}) , cores per node (R_j^{cpn}) and number of GPUs per node (R_j^{gpu}) are specified.

For the first type of job, each of the nodesets with $g = 0$ are tested for the total number of cores available in the nodeset. Then,

A. If the nodeset has enough cores for the job, the bid-class requires as few nodes as possible aligning the bid class to the beginning or end of the nodeset such that the requested total number of cores constraint is satisfied.

B. Same mechanism as in A above is applied, but this time without limiting the nodes to the beginning or end of the nodeset.

C. The nodeset is combined with their neighbours to create non-contiguous nodesets with higher number of cores. This type of bid class is only created if a job does not specifically request contiguous allocation.

If a job is of type 2, each of the nodesets with $g = R_j^{gpu}$ are considered. In this case,

A. If the nodeset has at least R_j^{cpu} cores on R_j^{node} nodes, than a bid class requesting this number of cores and nodes is created, aligning it to the beginning or end of that node set.

B. Same mechanism as in A above is applied, however this time the bid-class may request nodes from the middle of the nodeset.

C. As mentioned earlier, the nodesets in this case are combined with their neighbours to create non-contiguous nodesets. Again this type of bid is only created if a job does not specifically request contiguous allocation.

Jobs of type 3 are handled similarly to type 2 jobs. However, this time each node in the nodeset is checked if there are R_j^{cpn} cores available on that node. Figure 2 illustrates examples of alignments of the type A, B and C bid classes on the nodesets.

5. Bid Generation

We illustrate the bid generation process with the aid of an example shown in Figure 3. Figure 3 shows the bid generation process and the outcome of the auction that tells us what resource allocations are to be done. The AUCSCHED plugin first retrieves system state and information about the jobs in the queue. The jobs' resource requirements, their priorities and information about whether they want contiguous allocations are retrieved. For the example Figure 3, the window size is taken as four for simplicity. Because of this only the first four of the jobs participate in the auction for resources. The resource requirements of these four jobs are as follows:

- J_1: $R_j^{cpu} = 512$ cores,
- J_2: $R_j^{node} = 64$ nodes, $R_j^{cpn} = 2$ cores per node, $R_j^{gpu} = 1$ GPU per node,
- J_3: $R_j^{node} = 64$ nodes, $R_j^{cpn} = 4$ cores per node node, $R_j^{gpu} = 2$ GPUs/node,
- J_4: $R_j^{node} = 128$ nodes, $R_j^{cpn} = 1$ core per node.

The plugin looks at the system state. There are two available blocks of nodes from node 1 to node 64, and the other from node 81 to node 144 that form nodesets $(1, 64, 512, g)$ and $(1, 64, 512, g)$ for $g = 0, 1, 2$. After the nodesets are created, the AUCSCHED plugin can now generate possible bid classes. The generated bids have different preference values, F_{jc}. Subsection 5.1 explains how preference values are assigned. In general, we do not enumerate all possible bids. Since each bid appears as a binary variable in the IP solved, such an act would explode the total number of variables. Hence, it would not be possible to solve our IP problem within

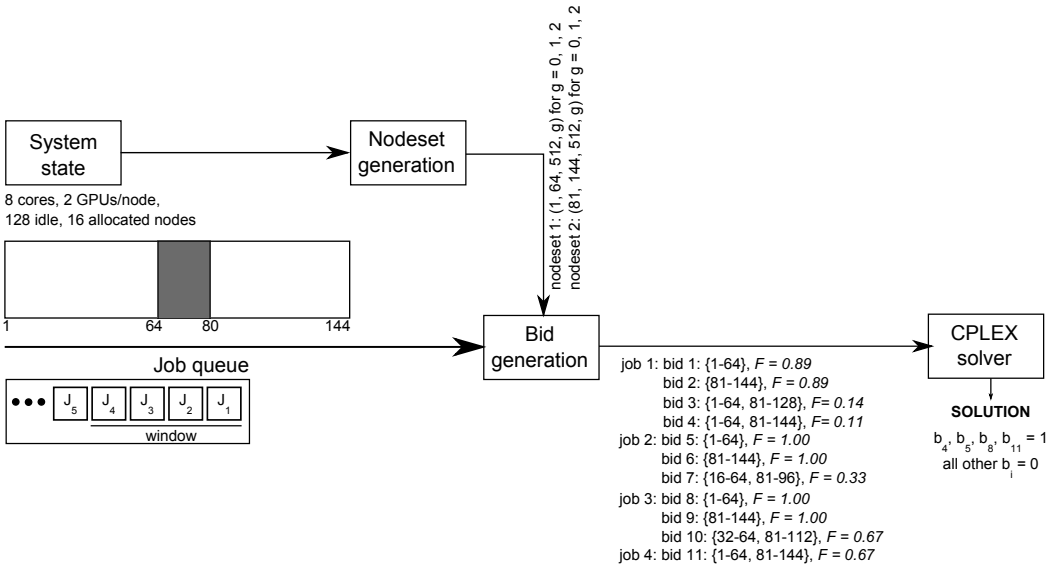

Fig. 3: Detailed bid generation figure for a 144 node system

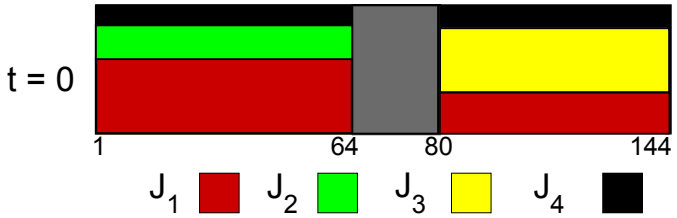

Fig. 4: Allocation by AUCSCHED for the 4 jobs in the queue. X-axis represents the nodes, Y-axis represents the cores in each node.

an acceptable time. In our plug-in there is a variable called $MAXBIDSPERJOB$ which is the limit on the number of bids generated by our system for each job. In Figure 3, we have shown at most 4 bids generated by the bid generator. The curly brackets next to the bids show the nodes requested by these bids.

CPLEX Solver [11] takes all bids as the input, and solves the IP problem. In this example, bids b_4, b_5, b_8 and b_{11} win the auction. As a result, the following resource allocations are performed:

- J_1 is allocated to the nodes $\{1-64, 81-144\}$,
- J_2 is allocated to the nodes $\{1-64\}$,
- J_3 is allocated to the nodes $\{81-144\}$,
- J_4 is allocated to the nodes $\{1-64, 81-144\}$.

Figure 4 shows the outcome of this allocation for our AUCSCHED plugin. it can be seen that this allocation is the optimal allocation. Figure 5 shows the allocation performed by SLURM/BF, and the consumable resources plugin for resource selection.

In addition to the type A,B and C bids, we also added an option that would make use of SLURM's resource selection plugin to determine what SLURM's allocation would be and added this as a bid to each job's set of bids. In order to determine the allocation for the second job in the queue, we first update our node information table as if that job had been scheduled, and let SLURM find its next allocation for the second job. We refer to these bids as SLURM bids. So our bid generation can be thought of as producing an enriched set of bids: i.e. bids corresponding to SLURM's would be allocation and the bids of type A,B and/or C that we generate.

5.1. F_{jc} Preference Value Calculation

A type A bid can be more desirable than a type B and C bid since it is a contiguous block and is aligned with another allocated block and hence, may lead to less fragmentation. A type B can be more desirable than type C, since it is a contiguous block. We are, therefore, motivated to assign preference values in such a way that

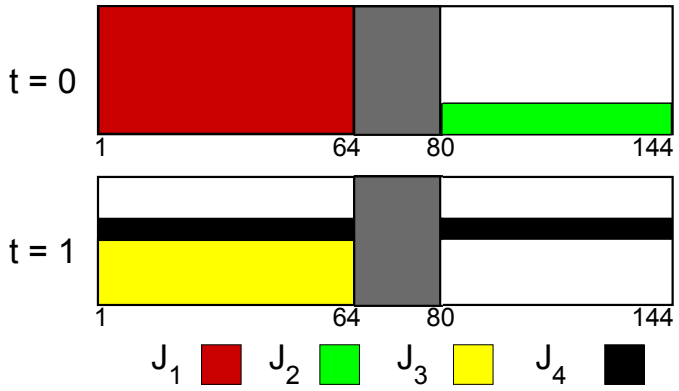

Fig. 5: Allocation by SLURM/BF for the 4 jobs in the queue. . X-axis represents the nodes, Y-axis represents the cores in each node. The pane on top shows the allocation for $t = 0$, and pane on bottom shows the allocation for $t = 1$.

$F_{jc_A} > F_{jc_B} > F_{jc_C}$ where c_A, c_B and c_C are respectively type A, B and C bids. We implement a function that has the following range of values:

$$F_{jc_C} \in (0, \frac{1}{2}), \qquad F_{jc_B} \in (\frac{1}{2}, \frac{3}{4}), \qquad F_{jc_A} \in (\frac{3}{4}, 1) \tag{11}$$

Let S_c denote the set of nodesets from which a bid's nodes comes from. For type A and B bids, $|S_c| = 1$. For type C bids $|S_c| > 1$. Let S denote the set of all nodesets. The formula used to calculate the preference value F_{jc} of a bid c is given as follows:

$$F_{jc} = 1.0 - k_1 - k_2 \cdot \frac{|N_c|}{|N| + 1} - k_3 \cdot \frac{|S_c|}{|S| + 1} \tag{12}$$

This function together with the contants k_1, k_2 and k_3 as given in Table 3 satisfies the range contraints given in Equation 11. The idea behind the function is to basically disfavour allocations that are fragmented or that leads to fragmentation.

Table 3: Constants for F_{jc} calculation for different job and bid classes

Job type 1			
	bid type A	bid type B	bid type C
k_1	0	0.25	0.5
k_2	0.25	0.25	0.25
k_3	0	0	0.25
Job types 2 and 3			
	bid type A	bid type B	bid type C
k_1	0	0.5	0.5
k_2	0	0	0
k_3	0	0	0.5

Finally we note that SLURM bids are added to the system with coefficients $k_1 = k_2 = k_3 = 0$, so that all of them have the highest F_{jc} values possible. In the next section, we present results from tests that contain SLURM bids as well tests that do not contain SLURM bids.

6. Workloads, Tests and Results

In order to compare the performance of the AUCSCHED plug-in with that of SLURM/BF, we carried out realistic SLURM emulation tests. In emulation tests, we ran an actual SLURM system and submitted jobs to it just like in a real life SLURM usage; the difference being that the jobs do not carry out any real computation or communication, but rather they just sleep. Such an approach is more advantageous than testing by simulation since it allows us to test the actual SLURM system rather than an abstraction of it. Since events are triggered

as in actual real life execution, a disadvantage of the emulation approach, however, is the longer time required to complete a test and hence the smaller number of test runs that can be run. For example, suppose that the next event will happen after 5 seconds. Then in an event based simulator, the current time variable can be added 5 with an add instruction and the time variable advances 5 seconds forward instantly whereas in our emulations, actual 5 seconds should elapse before the next event happens. This then implies a single test may take hours to complete. Note that even though there are efforts to speed up SLURM emulation tests by intercepting time calls [12], such an approach will still not be able to speed up our runs. This is because in our AUCSCHED plug-in, at every 5 seconds a scheduling step is invoked which solves an NP-hard problem using CPLEX. The solver can take as much as 5 seconds. As a result, even if simulation or [12]'s approach is used, the time taken by a test cannot be reduced. For this reason, we were able to perform a total of 826 tests over a period of one month with the computational resources available to us. We basically report and discuss these results.

Our tests have been conducted on a cluster system with 7 nodes with each node having two Intel X5670 6-core 3.20 Ghz CPU's and 48 GB's of memory. SLURM was compiled with the *enable-frontend* mode, which enabled one *slurmd* daemon to virtually configure 128, 256 or 1024 nodes.

The following cluster systems were emulated in our tests.

- *Machine-S* : a 128 node small system with 8 cores and 2 GPUs on each node.
- *Machine-M* : a 256 node medium system with 8 cores and 2 GPUs on each node.
- *Machine-L* : a 1024 node large system with 8 cores and 2 GPUs on each node.

Since heterogeneous CPU-GPU systems appeared quite recently, there is no publicly available workload for such systems. Parallel Workload Archive [10] does not contain workloads for such recent systems. In our previous work [3], a modified version of ESP [13] benchmark was used. ESP benchmark aims to test the performance of a scheduler. In ESP workloads, requests for GPU, node, contiguous allocation, cores per node are not present as it was mainly designed for a homogenous core based system. Therefore, we have decided to implement our own workload generator which is described in the next subsection.

6.1. Workload Generator

The workload generator that was developed takes the following as input:

- the maximum number of cores a job can request,
- minimum and maximum execution time of each job,
- estimated run time of a workload,
- job type as explained in Section 5,
- number of cores per node if specified by a job (a list of possible values),
- number of cores per GPU if specified by a job (a list of possible values).

Table 4: Workload types and job distributions

	Jobs with core request only	Jobs with node request (4 or 8 cores/node)	Jobs with 1 GPU/node request (1 or 2 cores/GPU)	Jobs with 2 GPU/node request (1 or 2 cores/GPU)
Workload I	100%	0%	0%	0%
Workload II	0%	100%	0%	0%
Workload III	50%	50%	0%	0%
Workload IV	40%	40%	20%	0%
Workload V	33.3%	33.3%	16.6%	16.6%

Three different versions of each of the five workloads given in Table 4 have been prepared. In the 1st version, none of the jobs requests a contiguous resource allocation. In the 2nd version, roughly half of the jobs request a contiguous resource allocation. Finally, in the third version, all jobs request a contiguous resource allocation. The execution time of each job in the workload was distributed uniformly between 60 and 600 seconds. Other parameters such as job type, number of CPUs per node and number of CPUs per GPU are selected from a list of possible values uniformly. The number of CPUs a job requests is multiples of the CPUs per node in a system. Each version of workload has been generated 7 times with the same parameters in order to reduce the effects of randomly generated jobs. Each test has been conducted in equal conditions (the jobs as well their submission orders) for AUCSCHED and SLURM/BF. In total, 413 tests have been conducted for each of the plugins.

6.2. Results

In Tables 5, 6, 7, we report the following statistics obtained from the tests for *Machine-S*, *Machine-M* and *Machine-L*.

- Runtime: The time it takes to schedule all jobs in workload.
- Teoretical runtime: is taken to be the summation of duration of each job times its requested core count, all divided by the total number of cores.
- Utilization: The ratio of the theoretical runtime to the runtime taken by the test.
- Waiting time : The waiting time of each job in the workload, from submission until scheduling.
- Fragmentation: Number of contiguous node blocks allocated to a job.
- Spread: ratio of difference between the indices of the last and first nodes plus one to the number of nodes allocated to a job
- Slowdown ratio: The ratio of the a job's turnaround time to execution time.

Note that in the tables, waiting time, fragmentation, spread and slowdown have been reported as averages over all jobs in each workload version. Also, the standard deviation of average waiting time, average fragmentation and average spread metrics have been calculated separately for each simulation, and have been averaged over simulations which have the same workload version. In the tests, *MAXBIDSPERJOB* parameter was taken as 5, i.e. for each job at most 5 bids were generated. For solving the IP problem with CPLEX, a time limit of 5 seconds was used. Also in these tests SLURM bids were generated and used in additon to the type A, B and C bids.

We have also carried out additional tests by varying some of the parameters we have used in Tables 5, 6, 7. Table 8 reports the results for the runs that just use type A,B and C bids and use no SLURM bids. In this table, *largest* keyword stands for a test with *MAXBIDSPERJOB* parameter set as 15, *longsolve* stands for using CPLEX solution time of 10 seconds, *shortsolve* stands for a CPLEX solution time of 3 seconds. Finally, *contigonly* stands for AUCSCHED solutions where only type A and B (i.e. contiguous block) bids are created. The distribution of different workload types in the results can be seen on the header row for each of the emulated system.

From the results given in Tables 5, 6, 7, we observe the following:

- Generally, AUCSCHED has a few percentage points of better utilization over SLURM/BF.
- Generally SLURM/BF is leading to less fragmentation but AUCSCHED is leading to less spread.
- In workloads I, II and III, there are no GPU jobs. The utilizations in these cases are quite high. In workload IV, there are some GPU jobs that require 1 GPU per node (there are no jobs requesting 2 GPU per node). It is observed that utilizations are lowered in IV especially when contiguously requested job fraction increases. When workloads contain jobs that request 2 GPUs per node (i.e. in V), it is observed that the system utilization drops drastically to 65-75% range both when our AUCSCHED as well as SLURM's own plug-in are used. We should, however note that definition of theoretical runtime we use is only a lower bound - it is not the optimal value. Computation of the optimal schedule and hence the optimal runtime value is an NP-hard problem. Hence, we may be faced with the following scenarios : (i) The algorithms in both plugins are working nicely but this low utilization may be close to the best that can be obtained due to more complex combinatorial nature of the problem, (ii) the algorithms in both plugins are not good enough to produce good solutions. Both cases, however, points to the need to further study of scheduling jobs that utilize multiple GPU cards on nodes.
- Slowdown ratio of AUCSCHED is generally higher than SLURM/BF. This is caused by the fact that SLURM's BF favors smaller jobs whereas AUCSCHED favors larger jobs. As a result, in AUCSCHED, we have higher waiting times for larger number of jobs, which in turn increases the overall average waiting times and slowdown ratios.

From the results given in the Table 8 for the difficult V workload, it is difficult to draw any general patterns. It is interesting to note however that, in the case of V.1-contigonlybids, and V.2-contigonlybids, AUCSCHED's scheduling of all the jobs as if they were explicitly requested as contiguous (even though they were not) did not lead to reduction in utilization rates. As a result AUCSCHED could schedule with no fragmentation at all, yet achieve more or less the same utilization as SLURM/BF.

Table 5: Results for Machine-S system

Workload version	Contiguous job fraction	Average theo. runtime (hours)	Method	Util	Average Runtime (hours)	Average Waiting time (hours)	Average Fragmentation	Average Slowdown ratio	Average Spread
I.1	0%	4.02	AUCSCHED	95%	4.21	1.74 ± 1.16	3.92 ± 2.69	27.88	3.28 ± 3.35
			SLURM/BF	95%	4.21	1.71 ± 1.17	2.20 ± 1.30	23.58	5.33 ± 5.97
I.2	50%	4.07	AUCSCHED	95%	4.29	1.72 ± 1.17	2.71 ± 2.66	26.91	2.01 ± 2.08
			SLURM/BF	93%	4.39	1.64 ± 1.20	1.63 ± 1.09	21.33	3.40 ± 5.08
I.3	100%	4.04	AUCSCHED	93%	4.30	1.77 ± 1.18	1.00 ± 0.00	27.88	1.00 ± 0.00
			SLURM/BF	91%	4.41	1.53 ± 1.26	1.00 ± 0.00	22.01	1.00 ± 0.00
II.1	0%	4.06	AUCSCHED	97%	4.20	1.69 ± 1.16	3.04 ± 2.32	26.98	5.35 ± 5.42
			SLURM/BF	95%	4.25	1.72 ± 1.19	2.28 ± 1.35	23.64	5.28 ± 5.57
II.2	50%	4.04	AUCSCHED	94%	4.30	1.70 ± 1.16	2.10 ± 1.86	27.53	3.70 ± 4.88
			SLURM/BF	93%	4.37	1.64 ± 1.20	1.58 ± 1.04	22.01	3.35 ± 5.19
II.3	100%	3.99	AUCSCHED	93%	4.28	1.74 ± 1.17	1.00 ± 0.00	28.06	1.00 ± 0.00
			SLURM/BF	91%	4.40	1.55 ± 1.25	1.00 ± 0.00	22.63	1.00 ± 0.00
III.1	0%	4.06	AUCSCHED	93%	4.30	1.79 ± 1.20	3.10 ± 2.51	28.65	3.77 ± 4.26
			SLURM/BF	96%	4.25	1.75 ± 1.19	2.30 ± 1.33	24.45	5.43 ± 5.75
III.2	50%	4.02	AUCSCHED	93%	4.34	1.75 ± 1.19	1.92 ± 1.74	27.87	2.42 ± 3.39
			SLURM/BF	93%	4.37	1.64 ± 1.20	1.62 ± 1.13	21.83	3.20 ± 4.80
III.3	100%	4.06	AUCSCHED	93%	4.36	1.79 ± 1.20	1.00 ± 0.00	28.68	1.00 ± 0.00
			SLURM/BF	91%	4.48	1.57 ± 1.27	1.00 ± 0.00	22.80	1.00 ± 0.00
IV.1	0%	4.03	AUCSCHED	93%	4.36	1.82 ± 1.20	3.43 ± 2.70	28.60	3.46 ± 3.44
			SLURM/BF	93%	4.33	1.69 ± 1.20	3.03 ± 2.17	21.71	5.63 ± 6.03
IV.2	50%	4.04	AUCSCHED	91%	4.46	1.82 ± 1.21	2.25 ± 2.20	29.41	2.55 ± 3.60
			SLURM/BF	89%	4.56	1.60 ± 1.22	1.80 ± 1.36	20.29	3.24 ± 4.78
IV.3	100%	3.96	AUCSCHED	89%	4.44	1.76 ± 1.21	1.00 ± 0.00	28.15	1.00 ± 0.00
			SLURM/BF	87%	4.53	1.53 ± 1.27	1.00 ± 0.00	21.33	1.00 ± 0.00
V.1	0%	4.04	AUCSCHED	71%	5.72	1.96 ± 1.43	2.68 ± 2.44	30.85	3.24 ± 4.73
			SLURM/BF	71%	5.71	1.72 ± 1.42	2.42 ± 1.67	20.89	4.70 ± 6.29
V.2	50%	4.19	AUCSCHED	68%	6.15	1.82 ± 1.44	2.11 ± 2.09	31.16	2.40 ± 3.28
			SLURM/BF	68%	6.15	1.71 ± 1.49	1.81 ± 1.46	20.63	2.62 ± 3.64
V.3	100%	4.03	AUCSCHED	65%	6.19	1.75 ± 1.45	1.00 ± 0.00	30.98	1.00 ± 0.00
			SLURM/BF	65%	6.21	1.66 ± 1.55	1.00 ± 0.00	21.01	1.00 ± 0.00

Table 6: Results for Machine-M system

Workload version	Contiguous job fraction	Average theo. runtime (hours)	Method	Util	Average Runtime (hours)	Average Waiting time (hours)	Average Fragmentation	Average Slowdown ratio	Average Spread
I.1	0%	4.02	AUCSCHED	97%	4.47	1.88 ± 1.27	4.70 ± 3.42	28.72	5.60 ± 6.91
			SLURM/BF	95%	4.53	1.80 ± 1.28	2.56 ± 1.65	23.47	6.52 ± 7.73
I.2	50%	4.16	AUCSCHED	93%	4.47	1.78 ± 1.15	3.03 ± 3.14	28.36	3.37 ± 5.12
			SLURM/BF	92%	4.53	1.70 ± 1.21	1.85 ± 1.46	22.89	4.01 ± 7.41
I.3	100%	4.01	AUCSCHED	93%	4.34	1.79 ± 1.20	1.00 ± 0.00	28.67	1.00 ± 0.00
			SLURM/BF	90%	4.46	1.60 ± 1.29	1.00 ± 0.00	23.76	1.00 ± 0.00
II.1	0%	4.04	AUCSCHED	97%	4.18	1.70 ± 1.16	3.62 ± 2.74	27.16	7.31 ± 7.49
			SLURM/BF	95%	4.25	1.70 ± 1.19	2.60 ± 1.56	23.31	7.32 ± 8.68
II.2	50%	3.88	AUCSCHED	93%	4.17	1.62 ± 1.12	2.40 ± 2.32	26.32	4.55 ± 6.59
			SLURM/BF	91%	4.24	1.56 ± 1.16	1.80 ± 1.33	20.86	4.43 ± 7.83
II.3	100%	3.99	AUCSCHED	92%	4.32	1.75 ± 1.18	1.00 ± 0.00	27.92	1.00 ± 0.00
			SLURM/BF	90%	4.45	1.56 ± 1.26	1.00 ± 0.00	22.71	1.00 ± 0.00
III.1	0%	4.06	AUCSCHED	96%	4.24	1.75 ± 1.16	4.14 ± 3.21	27.80	6.55 ± 7.29
			SLURM/BF	95%	4.28	1.73 ± 1.20	2.57 ± 1.53	23.76	6.75 ± 7.88
III.2	50%	3.95	AUCSCHED	93%	4.25	1.65 ± 1.13	2.53 ± 2.51	26.71	4.56 ± 7.17
			SLURM/BF	92%	4.30	1.59 ± 1.17	1.81 ± 1.38	21.60	4.12 ± 7.35
III.3	100%	3.99	AUCSCHED	92%	4.33	1.77 ± 1.18	1.00 ± 0.00	28.03	1.00 ± 0.00
			SLURM/BF	90%	4.44	1.57 ± 1.27	1.00 ± 0.00	22.45	1.00 ± 0.00
IV.1	0%	3.96	AUCSCHED	95%	4.16	1.67 ± 1.13	6.25 ± 4.66	26.41	7.17 ± 8.44
			SLURM/BF	93%	4.25	1.63 ± 1.19	3.45 ± 2.41	21.15	7.04 ± 8.86
IV.2	50%	4.08	AUCSCHED	90%	4.54	1.79 ± 1.23	3.04 ± 3.32	28.62	3.94 ± 5.55
			SLURM/BF	88%	4.62	1.61 ± 1.25	1.98 ± 1.65	20.54	3.96 ± 7.22
IV.3	100%	3.91	AUCSCHED	88%	4.44	1.71 ± 1.19	1.00 ± 0.00	28.05	1.00 ± 0.00
			SLURM/BF	86%	4.56	1.50 ± 1.27	1.00 ± 0.00	20.97	1.00 ± 0.00
V.1	0%	4.02	AUCSCHED	67%	6.07	1.93 ± 1.41	3.06 ± 2.86	30.45	3.84 ± 6.52
			SLURM/BF	67%	6.08	1.81 ± 1.49	2.70 ± 1.88	21.03	5.28 ± 8.40
V.2	50%	4.06	AUCSCHED	64%	6.36	2.02 ± 1.45	2.34 ± 2.66	31.87	2.82 ± 4.77
			SLURM/BF	63%	6.48	1.72 ± 1.56	1.82 ± 1.53	23.40	2.88 ± 4.77
V.3	100%	3.99	AUCSCHED	60%	6.68	2.01 ± 1.47	1.00 ± 0.00	33.01	1.00 ± 0.00
			SLURM/BF	60%	6.69	1.78 ± 1.66	1.00 ± 0.00	19.16	1.00 ± 0.00

Table 7: Results for Machine-L system

Workload version	Contiguous job fraction	Average theo. runtime (hours)	Method	Util	Average Runtime (hours)	Average Waiting time (hours)	Average Fragmentation	Average Slowdown ratio	Average Spread
I.1	0%	4.00	AUCSCHED	96%	4.18	1.75 ± 1.15	4.91 ± 2.96	27.98	5.05 ± 8.22
			SLURM/BF	95%	4.21	1.69 ± 1.18	3.01 ± 1.86	23.88	10.74 ± 19.85
I.2	50%	3.98	AUCSCHED	93%	4.30	1.71 ± 1.14	3.16 ± 2.98	27.73	2.90 ± 4.38
			SLURM/BF	91%	4.37	1.66 ± 1.18	1.97 ± 1.54	22.46	5.75 ± 13.38
I.3	100%	4.07	AUCSCHED	92%	4.43	1.86 ± 1.24	1.00 ± 0.00	28.83	1.00 ± 0.00
			SLURM/BF	89%	4.59	1.70 ± 1.31	1.00 ± 0.00	24.08	1.00 ± 0.00
II.1	0%	4.06	AUCSCHED	96%	4.24	1.75 ± 1.17	4.49 ± 3.32	27.99	9.45 ± 15.40
			SLURM/BF	95%	4.30	1.76 ± 1.21	3.01 ± 1.98	24.80	10.52 ± 18.54
II.2	50%	4.04	AUCSCHED	93%	4.30	1.72 ± 1.16	2.67 ± 2.63	28.06	5.36 ± 9.19
			SLURM/BF	91%	4.43	1.67 ± 1.20	1.96 ± 1.54	22.66	5.79 ± 13.79
II.3	100%	3.97	AUCSCHED	91%	4.34	1.76 ± 1.17	1.00 ± 0.00	27.77	1.00 ± 0.00
			SLURM/BF	88%	4.48	1.60 ± 1.25	1.00 ± 0.00	23.08	1.00 ± 0.00
III.1	0%	3.96	AUCSCHED	95%	4.17	1.72 ± 1.15	4.19 ± 2.94	27.38	6.30 ± 9.29
			SLURM/BF	95%	4.17	1.67 ± 1.15	3.13 ± 2.00	23.28	10.26 ± 15.78
III.2	50%	4.03	AUCSCHED	92%	4.40	1.70 ± 1.18	2.72 ± 2.68	26.96	4.25 ± 6.80
			SLURM/BF	90%	4.45	1.62 ± 1.21	1.98 ± 1.60	21.76	6.06 ± 16.05
III.3	100%	4.01	AUCSCHED	91%	4.41	1.82 ± 1.21	1.00 ± 0.00	29.16	1.00 ± 0.00
			SLURM/BF	88%	4.54	1.66 ± 1.27	1.00 ± 0.00	24.33	1.00 ± 0.00
IV.1	0%	4.01	AUCSCHED	95%	4.21	1.72 ± 1.15	9.79 ± 8.12	27.23	9.80 ± 15.46
			SLURM/BF	93%	4.33	1.69 ± 1.19	3.74 ± 2.69	22.56	9.43 ± 19.97
IV.2	50%	4.02	AUCSCHED	89%	4.50	1.78 ± 1.21	3.80 ± 4.96	27.76	4.38 ± 7.51
			SLURM/BF	87%	4.60	1.61 ± 1.25	2.11 ± 1.80	19.86	4.57 ± 10.57
IV.3	100%	4.03	AUCSCHED	87%	4.60	1.88 ± 1.26	1.00 ± 0.00	30.16	1.00 ± 0.00
			SLURM/BF	85%	4.76	1.640 ± 1.30	1.00 ± 0.00	22.53	1.00 ± 0.00
V.1	0%	3.95	AUCSCHED	65%	6.12	1.93 ± 1.40	3.74 ± 4.57	30.84	4.61 ± 11.53
			SLURM/BF	63%	6.33	1.88 ± 1.37	2.90 ± 2.15	21.41	6.78 ± 14.10
V.2	50%	4.08	AUCSCHED	72%	5.67	1.52 ± 1.22	2.94 ± 4.27	23.65	3.21 ± 6.97
			SLURM/BF	65%	6.29	1.77 ± 1.52	1.97 ± 1.73	21.44	3.74 ± 11.23
V.3	100%	4.04	AUCSCHED	61%	6.63	2.05 ± 1.51	1.00 ± 0.00	32.45	1.00 ± 0.00
			SLURM/BF	59%	6.85	1.91 ± 1.55	1.00 ± 0.00	21.94	1.00 ± 0.00

Table 8: Additional non standard run results for Machine-L system (excluding SLURM bids)

Workload version	Contiguous job fraction	Average theo. runtime (hours)	Method	Util	Average Runtime (hours)	Average Waiting time (hours)	Average Fragmentation	Average Slowdown ratio	Average Spread
V.1	0%	4.04	AUCSCHED	64%	6.29	2.02 ± 1.47	3.39 ± 3.66	32.00	4.06 ± 8.67
			SLURM/BF	60%	6.71	1.86 ± 1.54	2.83 ± 1.96	21.60	6.86 ± 14.26
V.2	50%	4.02	AUCSCHED	63%	6.35	2.01 ± 1.45	2.64 ± 3.64	3.00	3.00 ± 6.43
			SLURM/BF	58%	6.96	1.70 ± 1.47	2.00 ± 1.74	20.59	3.67 ± 8.27
V.3	100%	4.01	AUCSCHED	59%	6.80	2.07 ± 1.52	1.00 ± 0.00	33.35	1.00 ± 0.00
			SLURM/BF	55%	7.26	1.74 ± 1.59	1.00 ± 0.00	19.98	1.00 ± 0.00
V.1 - largest	0%	4.03	AUCSCHED	71%	5.74	2.46 ± 1.55	3.69 ± 4.13	39.03	2.34 ± 3.57
			SLURM/BF	66%	6.16	1.78 ± 1.50	2.76 ± 1.89	22.50	5.82 ± 9.39
V.2 - largest	50%	3.99	AUCSCHED	63%	6.35	2.50 ± 1.63	2.36 ± 3.75	39.46	1.97 ± 3.37
			SLURM/BF	56%	7.11	1.70 ± 1.49	2.01 ± 1.76	20.70	3.59 ± 7.22
V.3 - largest	100%	3.92	AUCSCHED	58%	6.79	2.61 ± 1.76	1.00 ± 0.00	41.03	1.00 ± 0.00
			SLURM/BF	57%	6.88	1.67 ± 1.50	1.00 ± 0.00	21.78	1.00 ± 0.00
V.1 - largest - longsolve	0%	3.95	AUCSCHED	67%	5.92	2.60 ± 1.61	2.85 ± 3.52	41.32	2.18 ± 4.33
			SLURM/BF	59%	6.68	1.75 ± 1.44	2.90 ± 2.14	21.72	6.81 ± 14.65
V.2 - largest - longsolve	50%	4.00	AUCSCHED	60%	6.63	2.67 ± 1.70	2.01 ± 3.08	42.18	1.51 ± 1.81
			SLURM/BF	59%	6.77	1.70 ± 1.47	1.98 ± 1.74	20.74	3.75 ± 8.23
V.3 - largest - longsolve	100%	3.98	AUCSCHED	56%	7.18	2.85 ± 1.86	1.00 ± 0.00	45.13	1.00 ± 0.00
			SLURM/BF	58%	6.87	1.71 ± 1.53	1.00 ± 0.00	22.43	1.00 ± 0.00
V.1 - largest - shortsolve	0%	3.93	AUCSCHED	68%	5.77	2.01 ± 1.39	6.41 ± 8.04	32.70	2.96 ± 5.77
			SLURM/BF	60%	6.56	1.71 ± 1.39	2.87 ± 2.03	21.65	6.69 ± 13.31
V.2 - largest - shortsolve	50%	4.02	AUCSCHED	64%	6.28	2.07 ± 1.47	2.75 ± 4.19	32.86	1.92 ± 3.40
			SLURM/BF	58%	6.94	1.73 ± 1.48	2.00 ± 1.79	21.47	3.69 ± 8.86
V.3 - largest - shortsolve	100%	3.95	AUCSCHED	59%	6.74	2.13 ± 1.54	1.00 ± 0.00	34.28	1.00 ± 0.00
			SLURM/BF	58%	6.82	1.74 ± 1.52	1.00 ± 0.00	23.13	1.00 ± 0.00
V.1 - contigonlybids	0%	4.03	AUCSCHED	59%	6.88	2.08 ± 1.54	1.00 ± 0.00	33.08	1.00 ± 0.00
			SLURM/BF	60%	6.71	1.77 ± 1.44	2.83 ± 2.11	22.29	6.15 ± 11.20
V.2 - contigonlybids	50%	3.97	AUCSCHED	60%	6.67	2.04 ± 1.50	1.00 ± 0.00	32.00	1.00 ± 0.00
			SLURM/BF	59%	6.73	1.71 ± 1.47	2.03 ± 1.81	20.88	3.86 ± 8.85

7. Discussion and Conclusions

This work formulated job scheduling process as an auction problem in which a a window of jobs from the front of the job queue submit bids for the resources they request. Such an approach can help us achieve two main objectives:

1. Development of new scheduling heuristics and software for new heterogeneous supercomputer architectures.
2. Provide scheduler support for applications that may provide alternative implementations or different resource requirements. provided by the programmer/developer.

The motivation for objective (1) comes mainly because for many years schedulers were used and optimized for core based systems. But with the recent emergence of heterogeneous systems with accelerator cards such as GPUs, the scheduling problem becomes more complex and more sophisticated. As a result new combinatorial optimization algorithms need to be developed to schedule applications which may use these accelerators. The fact that in our tests with 2 GPU cards per node, utilization values dropped drastically provides evidence for the case that scheduling of such systems need to be studied further. The motivation for objective (2) comes from the fact that mechanisms must be provided by the scheduler to the applications (that may provide alternative implementations and/or different resource requirements) to first express these alternatives and secondly to handle such alternative specifications during scheduling. Our auction mechanism which was integrated into SLURM as AUCSCHED plug-in, in fact, addresses this second issue. Internally bids are generated by our AUCSCHED for each job. Only one of these bids wins and leads to allocation of resources as requested by the winning bid. Hence if alternative implementations and/or different resource requirements are provided by a scheduled job, then each of these alternatives can be expressed as a bid and only one of them can be selected for execution (the one which best fits the available resources on the system). Such a capability allows the system resources to be used more efficiently and also let a user's job be executed sooner. For example, consider an application that can run on just cores as well as make use of GPUs if it is allocated on nodes with available GPUs. If the scheduler is made aware of this information, then only one job can be submitted with these specifications and depending on the availability the job can be allocated just cores or cores and GPUs for its execution. The current release of AUCSCHED includes the internal bid mechanism. The next release will also include SLURM command directives for enriching the expressibility of alternative implementations and resource requests.

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-283493. We thank Itir Karac for her help during the implementation of AUCSCHED plug-in. The authors have used the Bogazici University Polymer Research Center's cluster to carry out the tests in the paper.

References

1. A. Yoo, M. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management", in Job Scheduling Strategies for Parallel Processing vol. 2862 of Lecture Notes in Computer Science, pp. 44-60, Springer Berlin Heidelberg (2003).
2. "Slurm Used on the Fastest of the TOP500 Supercomputers", `http://www.hpcwire.com/hpcwire/2012-11-21/slurm_used_on_the_fastest_of_the_top500_supercomputers`, (2012).
3. S. Soner, C. Ozturan, "Integer Programming Based Heterogeneous CPU-GPU Cluster Scheduler for SLURM Resource Manager", 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), pp.418-424, (2012).
4. "Slurm Documentation - Topology Guide", `http://www.schedmd.com/slurmdocs/topology.html`, (2012).
5. Y. Georgiou, PhD Thesis, Scholes, "Contributions For Resource and Job Management in High Performance Computing", Universite de Grenoble, France (2010).
6. Y. Georgiou and M. Hautreux, "Evaluating scalability and efciency of the Resource and Job Management System on large HPC Clusters", Job Scheduling Strategies for Parallel Processing Lecture Notes in Computer Science Volume 7698, pp 134-156, (2013).
7. A. H. Ozer, C. Ozturan, "A model and heuristic algorithms for multi-unit nondiscriminatory combinatorial auction", Computers & Operations Research, Volume 36, Issue 1, pp 196-208, (2009).

8. J. A. Pascual, J. Navaridas and J. Miguel-Alonso, "Effects of Topology-Aware Allocation Policies on Scheduling Performance", Job Scheduling Strategies for Parallel Processing Lecture Notes in Computer Science Volume 5798, pp 138-156, (2009).
9. F. J. Ridruejo, and J. Miguel-Alonso, "INSEE: An Interconnection Network Simulation and Evaluation Environment", In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 10141023. Springer, Heidelberg (2005).
10. D. Feitelson, "Parallel workloads archive", `http://www.cs.huji.ac.il/labs/parallel/workload`, (2005).
11. "IBM ILOG CPLEX Optimize", `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`, (2012).
12. A. Lucero, "Simulation of batch scheduling using real production-ready software tools", Proceedings of the 5th IBERGRID, (2011).
13. A. T. Wong, L. Oliker, W. T. C. Kramer, T. L. Kaltz, and D. H. Bailey, "ESP: A System Utilization Benchmark", in Supercomputing, ACM/IEEE Conference, (2010).