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

Partnership for Advanced Computing in Europe

Topologically Aware Job Scheduling for SLURM

Seren Soner[1], Can Özturan[1,*]

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

Abstract

SLURM is a popular resource management system that is used on many supercomputers in the TOP500 list. In this work, we describe our new AUCSCHED3 SLURM scheduler plug-in that extends our earlier AUCSCHED2 plug-in with a capability to do topologically aware mappings of jobs on hierarchically interconnected systems like trees or fat trees. Our approach builds on our previous auction based scheduling algorithm of AUCSCHED2 and generates bids for topologically good mappings of jobs onto the resources. The priorities of the jobs are also adjusted slightly without changing the original priority ordering of jobs so as to favour topologically better candidate mappings. SLURM emulation results are presented for a heterogeneous 1024 node system which has 16 cores and 3 GPUs on each of its nodes. The results show that our heuristic generates better topological mappings than SLURM/Backfill. AUCSCHED3 is available at http://code.google.com/p/slurm-ipsched/.

1. Introduction

SLURM is a popular resource management system that is used on many supercomputers in the TOP500 list. SLURM provides two primary modes of operation for topology-aware job placement in order to reduce network contention: One mode for hierarchical interconnects like a tree (or a fat tree) and another mode for three-dimensional torus architectures. In this work, we contribute a new AUCSCHED3 SLURM scheduler plug-in that has a capability to do topologically aware mappings of jobs on hierarchically interconnected systems. Figure 1 shows examples of hierarchical interconnects that can be handled by SLURM. Figure 1(a) is a simple tree and (b) is a fat tree. The SLURM document [1] remarks that listing every switch connection results in a slower scheduling algorithm for SLURM to optimize job placement and as a matter of practicality suggests configuring a fat tree topology like the one in Figure 1(b) as the one in Figure 1(a).

In recent literature, the problem of topology aware mapping has been studied at the system and application levels. Bhatele [2] develops topology aware mapping algorithms for parallel applications with regular and irregular communication patterns. Bhatele remarks that message latencies are not independent of the number of hops between processors. Bhatele introduces a metric called hop-bytes which is the weighted sum of the number of hops between the source and destination for all messages, with the weights being the message sizes. Bhatele remarks the importance of hop count between communicating tasks.

The work by Pascual *et al.* [3] studies the effects of topology aware allocation policies on scheduling performance. In particular, they use simulation techniques to evaluate contiguous and quasi-contiguous (a relaxed version of contiguous) allocations on tree topologies. They report that in applications where communication locality leads to effective reduction of execution time, the gains more than compensate the scheduling inefficiency, hence, resulting in better overall performance.

The work by Georgiou and Hautreux [4] evaluates (i) the scalability of SLURM to manage large numbers of resources and jobs and (ii) network topology aware placement of jobs on systems with a tree interconnect topology. For testing workloads, our work makes use of SLURM emulation similar to what Georgiou and Hautreux [4] used in their work.

The SLURM document [1] also states the following about the technique used to optimize job performance on a hierarchical interconnect: *"The basic algorithm is to identify the lowest level switch in the hierarchy that can satisfy a job's request and then allocate resources on its underlying leaf switches using a best-fit algorithm."*

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

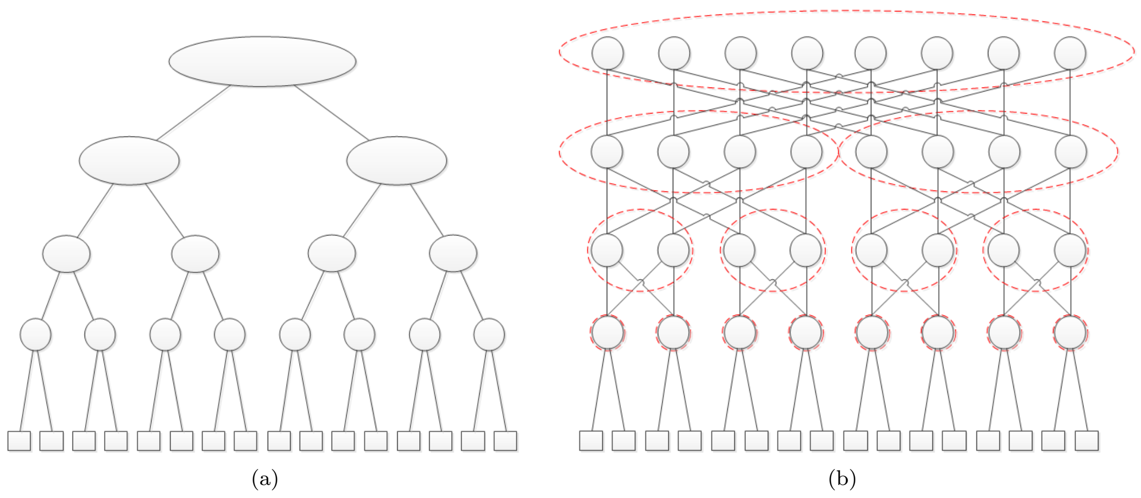


Fig. 1: Example switch configurations: a tree (a) and a fat tree (b)

Our new plug-in AUCSCHED3 that we propose is based on our previous auction based scheduling algorithm of AUCSCHED2 [5] and works by generating bids for topologically good mappings of jobs onto the resources.

This whitepaper is organized as follows: In Section 2, we present our new topologically aware auction based scheduling model. In Section 3, we present details of workloads that we used to test the new topology aware SLURM plug-in. In Section 4, SLURM emulation results are presented for a heterogeneous 1024 node CPU-GPU system. Finally, the whitepaper is concluded with a discussion of results and future work in Section 5.

2. Topology Aware Auction Based Scheduling Model

Our auction based scheduling algorithms [5, 6, 7] work by (*i*) taking a window of jobs from the front of the job queue and the available resource information from the nodes (*ii*) generating a number of bids for each job and (*iii*) solving an integer programming (IP) problem to select the bids of the jobs for deciding which resources to allocate to the jobs. We first present the IP formulation which is derived in [5] and used in AUCSCHED2 plug-in. We then present modifications that we make to this IP formulation in the current work in order to make it topologically aware for hierarchically interconnected systems.

Table 1: List of main symbols used in IP formulation

Symbol	Meaning
R_i^{gres}	Number of generic resources per node requested by bid i
A_n^{cpu}	Number of available CPU cores on node n
A_n^{gres}	Number of available generic resources on node n
P_j	Priority of job j
C_{ji}	A heuristically assigned cost value in $(0, 3]$ of bid i of job j
α	A constant multiplying C_{ji}
T_{in}	Number of cores on node n requested by bid i
U_{in}	Boolean parameter indicating whether bid i requires any resources on node n
J	Set of jobs that are in the window: $J = \{j_1, \ldots, j_{\|J\|}\}$
N_i	Number of nodes requested by bid i
L_i	Level of lowest level common switch of the nodes requested by bid i
L_{max}	Level of the highest level switch in the system,
P_{min_diff}	Minimum absolute priority difference between any pair of jobs in the window
N	Set of nodes : $N = \{n_1, \ldots, n_{\|N\|}\}$
B_j	Set of bid indices for job j : $B_j = \{i_1, \ldots, i_{\|B_j\|}\}$
b_i	Binary variable corresponding to bid i

Given the list of symbols in Table 1, our IP formulation presented below selects a number a jobs whose bids maximize an objective function that sums the priorities of jobs adjusted with a heuristically assigned cost value that makes it topologically and generic resource aware:

$$Maximize \quad \sum_{j\in J}\sum_{i\in B_j}(P_j - \alpha \cdot C_{ji}) \cdot b_i \tag{1}$$

subject to constraints :

$$\sum_{i\in B_j} b_i \leq 1 \; for \; each \; j \in J \tag{2}$$

$$\sum_{j\in J}\sum_{i\in B_j} b_i \cdot T_{in} \leq A_n^{cpu} \; for \; each \; n \in N \tag{3}$$

$$\sum_{j\in J}\sum_{i\in B_j} b_i \cdot U_{in} \cdot R_i^{gres} \leq A_n^{gres} \; for \; each \; n \in N \tag{4}$$

The objective function given by (1) is different from that used in our previous AUCSCHED2 formulation [5]. In the objective function, we subtract the cost C_{ji} (multiplied by a constant called α) of each bid from the priority of the job. The idea is to let the factor $\alpha \cdot C_{ji}$ slightly adjust the priority by a small amount in order to disfavour topologically not so good bids from being selected. The constraints (2-4) are the same as those used in AUCSCHED2.

Our heuristic basically involves defining a function C_{ji} that penalizes topologically not so good bids in a careful way so as to maintain the original ordering of the jobs in the queue. The following function is used to calculate the cost of a bid i and, as will be shown, it also helps us to maintain the ordering when multiplied by the scaling constant α:

$$C_{ji} = 1.0 + \frac{N_i}{|N|} + \frac{L_i}{L_{max}} - \frac{R_i^{gres}}{G_{max}} \tag{5}$$

Here, N_i, L_i and R_i^{gres} are respectively the number of nodes, the level of common lowest level switch of the nodes and the number of generic resources (e.g. GPU or Xeon Phi) requested by bid i. $|N|$, L_{max} and G_{max} are respectively the number of nodes, the level of the highest level switch and the maximum number of generic resources per node available in the system. Note that $C_{ji} \in (0, 3]$, since $\frac{N_i}{|N|} \in (0, 1]$, $\frac{L_i}{L_{max}} \in [0, 1]$ and $\frac{R_i^{gres}}{G_{max}} \in [0, 1]$.

Let P_{min_diff} denote minimum absolute priority difference between any pair (j_1, j_2) of jobs in the window:

$$P_{min\_diff} = \min_{j_1,j_2\in J} |P_{j_1} - P_{j_2}| \tag{6}$$

We choose the scaling constant α as follows:

$$\alpha = \frac{P_{min\_diff}}{3} \tag{7}$$

Now, we can prove that this α value always preserves the priority order. We do proof by contradiction. We want to show that if $P_{j_1} > P_{j_2}$, then $P_{j_1} - \alpha \cdot C_{j_1,i_1} > P_{j_2} - \alpha \cdot C_{j_2,i_2}$ for any given $j_1, j_2 \in J$, $j_1 \neq j_2$, $i_1 \in B_{j_1}$ and $i_2 \in B_{j_2}$. Suppose that this is not the case, i.e.:

$$P_{j_1} - \alpha \cdot C_{j_1,i_1} \leq P_{j_2} - \alpha \cdot C_{j_2,i_2} \tag{8}$$

Then,

$$\begin{gathered} P_{j_1} - P_{j_2} \leq \alpha \cdot (C_{j_1,i_1} - C_{j_2,u}) \\ P_{min\_diff} \leq P_{j_1} - P_{j_2} \leq \alpha \cdot (C_{j_1,i_1} - C_{j_2,i_2}) \\ P_{min\_diff} \leq P_{j_1} - P_{j_2} \leq \frac{P_{min\_diff}}{3} \cdot (C_{x,i_1} - C_{j_2,i_2}) \\ 3 \leq C_{j_1,i_1} - C_{j_2,i_2} \end{gathered} \tag{9}$$

But since $C_{j_1,i_1}, C_{j_2,i_2} \in (0, 3]$, we know that the following is true:

$$-3 < C_{j_1,i_1} - C_{j_2,i_2} < 3 \tag{10}$$

Inequality (9) then contradicts with inequality (10). Therefore, the value of α given in equation (7) maintains the order of priorities of jobs in the window after they are adjusted by the costs.

The bid generation phase used in AUCSCHED3 is similar to that of AUCSCHED2. We explain just the differences in this work. In AUCSCHED2, a 1-D array of nodes is scanned in order to generate *nodesets*, which are a contiguous set of nodes. Using the nodesets, bids of jobs are then generated. In AUCSCHED3, we proceed in a similar fashion with some modifications. We generate the nodesets by using the switch information along with the available resource information at each node. We scan the nodes which are covered by each switch for contiguous set of resources. After the nodesets are generated, we generate the bids. After the bids are generated, we form the auction based resource allocation problem which is then solved as the IP problem given in (1-4). As the IP solver, CPLEX [8] package is used.

3. Workload Generation

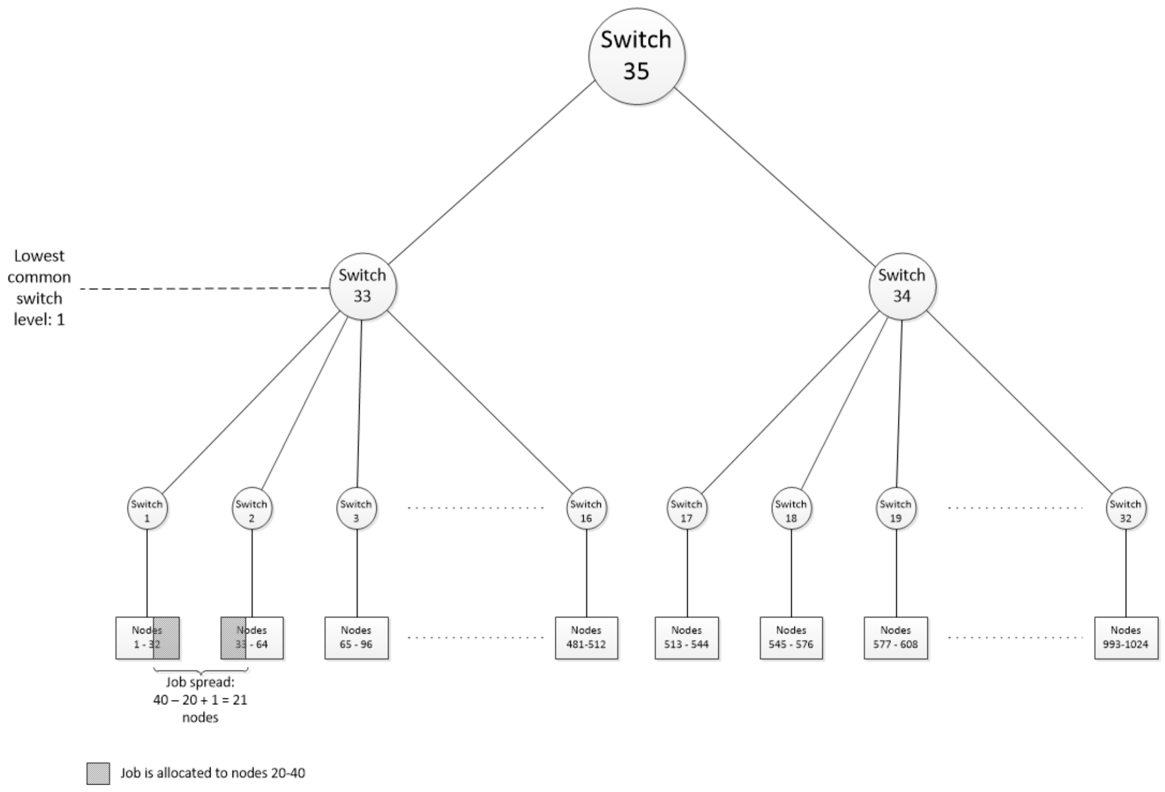


Fig. 2: 1024-node system emulated and illustration of job spread and lowest level common switch metrics

We test the effectiveness of our new AUCSCHED3 plug-in on a hierarchically interconnected 1024 node system with 16 cores and 3 GPUs on each of its nodes. Figure 2 shows the switch topology of the system.

We design six different workloads (numbered 1 through 6) that take into consideration the types of jobs

Table 2: Job types in the workload

Job Type	Description	SLURM job submission command
A	only x cores	`srun -n x`
B	x cores on y nodes	`srun -n x -N y`
C	x cores on y nodes, 1 GPU on each node	`srun -n x -N y --gres=gpu:1`
C'	x cores on y nodes, 1 to 3 GPUs on each node	`srun -n x -N y --gres=gpu:1-3`
D	x cores on y nodes, 2 GPUs on each node	`srun -n x -N y --gres=gpu:2`
D'	x cores on y nodes, 2 to 3 GPUs on each node	`srun -n x -N y --gres=gpu:2-3`
E	x cores on y nodes, 3 GPUs on each node	`srun -n x -N y --gres=gpu:3`

Table 3: Workload types and job distributions

Workload ID	Number of jobs	Percentage of job types A	B	C	D	E
1	350	100	0	0	0	0
2	2095	100	0	0	0	0
3	350	0	100	0	0	0
4	2095	0	100	0	0	0
5	350	20	20	20	20	20
6	2095	20	20	20	20	20
5'	same as 5, but uses GPU ranges					
6'	same as 6, but uses GPU ranges					

that can be submitted to SLURM. There are five types of jobs (named A, B, C, D, E) with each type making different resource requests as shown in Table 2. This table shows the SLURM job submission commands for each type of job. The workloads are made up of various percentages of these job types and are shown in Table 3. The execution time of each job is uniformly distributed between 30 and 300 seconds. The number of cores (`x` in Table 2) is uniformly distributed between 1 and the number of cores in the system. The number of nodes (`y` in Table 2) is uniformly distributed between 1 and the number of nodes in the system. Each workload is generated and tested 7 times. Average of 7 tests are reported for each workload. As shown in Table 3, workloads 1 and 2 include jobs with only core requests. Workloads 3 and 4 include jobs with both core and node requests. In workloads 5 and 6, the percentage of each job type in the workload is taken equally as 20%.

When testing SLURM's Backfill plug-in, we use type A, B, C, D and E jobs. AUCSCHED2 and AUCSCHED3 also provide support for generic resource ranges (which is not available in SLURM/Backfill). Such a feature can be useful to runtime auto-tuning applications that can make use of a variable number of generic resources such as GPUs. Job types C' and D' and workloads 5' and 6' are the same as their counterparts but use GPU ranges. Therefore, we only test these using AUCSCHED3.

4. Tests and Results

In order to test our previous AUCSCHED2 plug-in [5], we performed SLURM emulation tests. In emulation tests, jobs are submitted to an actual SLURM system just like in a real life SLURM usage; the only 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.

To test our new AUCSCHED3 plug-in, we proceed in a similar fashion by conducting emulation tests. We are able to retrieve topology related information of allocated jobs and hence we can evaluate goodness of allocations.

The emulation tests are conducted on a local 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 is compiled with the *enable-frontend* mode, which enables one *slurmd* daemon to virtually configure all of the 1024 nodes in the supercomputer system that we emulate.

The results are analyzed using the following performance measures and reported in Table 4:

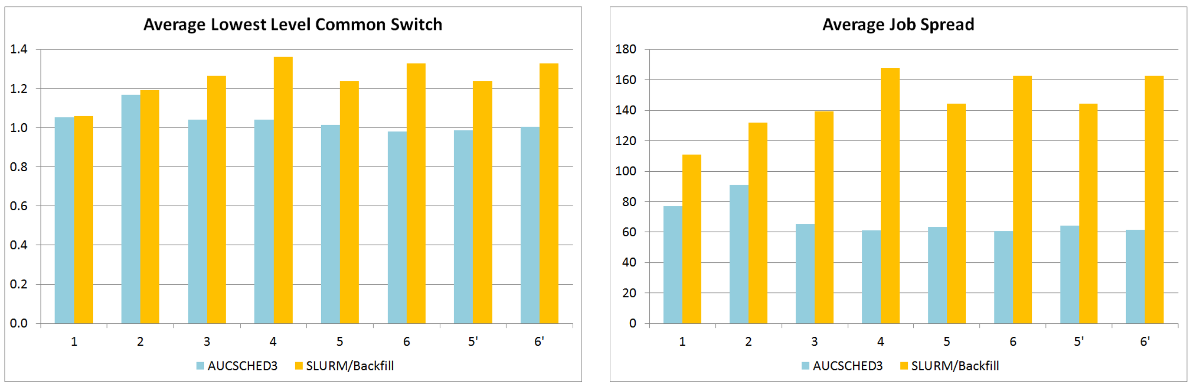
- *Lowest level common switch:* The lowest level common switch from which all the nodes allocated to a job can be reached. See Figure 2 for an example. Average values for all the jobs are reported in the Table 4.
- *Spread:* is the distance from the first node to the last node allocated to a job. See Figure 2 for an example. Average spread values for all the jobs are reported in the Table 4.
- *Utilization:* The ratio of the theoretical run-time to the observed run-time of the test (workload). Here, observed run time of the workload is the time from the submission of the first job to the completion of the last job. Theoretical run time is calculated by summing up the duration of each job multiplied by the number of requested cores, divided by the total number of cores available in the system. This definition of theoretical run time is not the exact value, but provides a lower bound for the theoretical run time. Computation of the optimal theoretical run-time is an NP-complete problem [9, p. 65].

We also plot average lowest common switch level and the average spread measures in Figure 3. To get a better insight, the distribution of jobs and the total job durations over lowest common switch levels of all jobs in all workloads are given in Figure 4(a) and (b) respectively. The plots in Figure 5(a,b) show the same information for type 5, 6, 5' and 6' jobs. We note that plots (a) show percentages of the numbers of jobs whereas plots (b) show percentages of total job durations. Even though each job has a different execution time, the plots in (a)

Table 4: Test results

Workload ID	Plugin	Avg. lowest level common switch	Avg. spread	Utilization (%)
1	AUCSCHED3	1.05 ± 0.68	77.2 ± 165.6	90
	SLURM/Backfill	1.06 ± 0.77	110.7 ± 230.7	89
2	AUCSCHED3	1.17 ± 0.62	90.9 ± 179.5	96
	SLURM/Backfill	1.19 ± 0.69	131.9 ± 250.3	95
3	AUCSCHED3	1.04 ± 0.58	65.4 ± 140.3	78
	SLURM/Backfill	1.26 ± 0.68	139.4 ± 262.6	82
4	AUCSCHED3	1.04 ± 0.58	61.1 ± 134.8	85
	SLURM/Backfill	1.36 ± 0.69	167.8 ± 298	88
5	AUCSCHED3	1.01 ± 0.67	63.5 ± 151.4	83
	SLURM/Backfill	1.24 ± 0.72	144.3 ± 272.7	81
6	AUCSCHED3	0.98 ± 0.56	60.9 ± 137.5	89
	SLURM/Backfill	1.33 ± 0.70	162.7 ± 293.7	87
5'	AUCSCHED3	0.99 ± 0.65	64.4 ± 149.4	86
	SLURM/Backfill	1.24 ± 0.72	144.3 ± 272.7	81
6'	AUCSCHED3	1.00 ± 0.59	61.4 ± 138.2	91
	SLURM/Backfill	1.33 ± 0.70	162.7 ± 293.7	87

and (b) are quite similar with minor deviations from each other. With regard to this similarity, we note that job durations are generated using uniform distribution as stated in Section 4 and nothing special is done or observed about placement, for example, of long or short running jobs on lower level switches.



(a) (b)

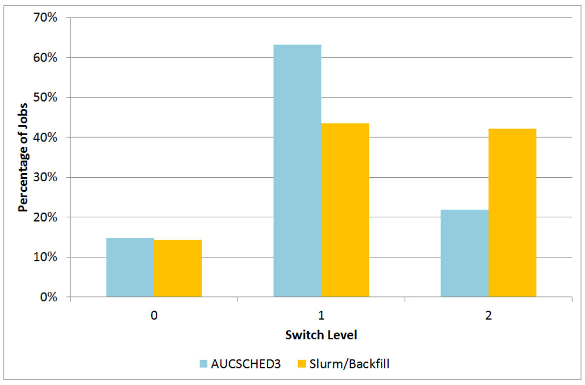
Fig. 3: Results of AUCSCHED3 compared to SLURM/Backfill plug-in, (a) average lowest level common switch (b) average job spread

5. Discussion and Conclusions

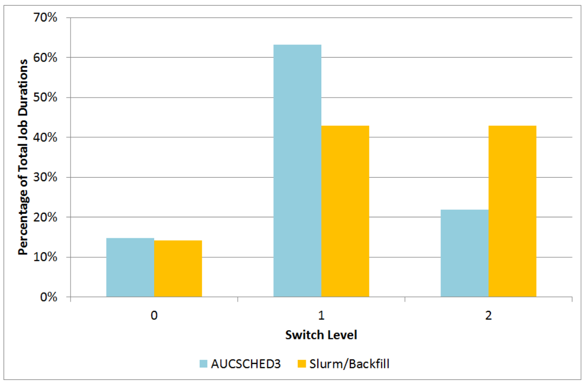
Jobs with good topology mappings are likely to run faster since communication will be faster. Hence:

- From the users' perspectives, allocation topology improvements are definitely important and are going to be welcomed by the users.
- From the system administrators' perspective (who aim to achieve high system utilization) jobs with good topology mappings will individually contribute to higher system utilization since they will run faster. However, good topology mappings of jobs should not come at the expense of delayed execution of jobs that wait for resources with good topology to be available which in turn may lead to overall reduced system utilization.

Since we are performing emulation with jobs doing no computation and communication, improvements in execution times and utilizations are not going to be reflected in the results. Hence, when comparing performances of AUCSCHED3 and SLURM/Backfill from system administrators' perspective, we should expect AUCSCHED3

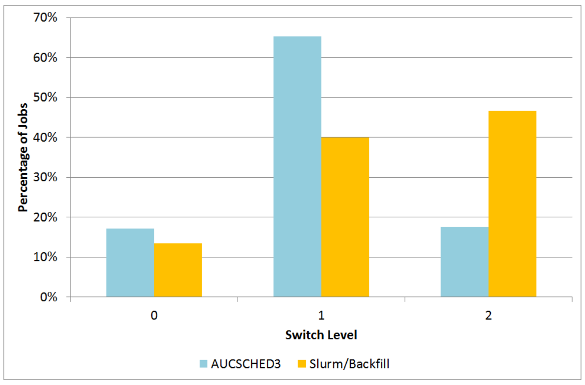


(a)

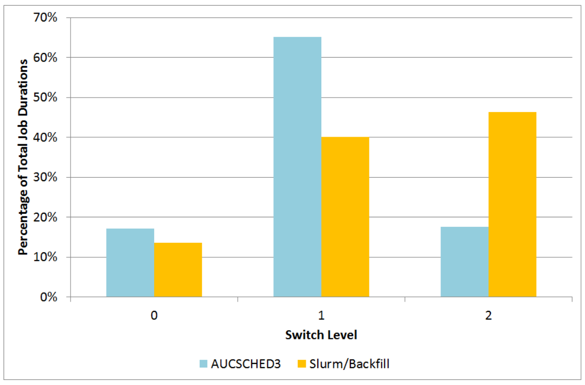


(b)

Fig. 4: Distribution of percentage of jobs over switch levels (a) and percentage of total job durations over switch levels (b) for all workloads



(a)



(b)

Fig. 5: Distribution of percentage of jobs over switch levels (a) and percentage of total job durations over switch levels (b) for workloads 5, 6, 5' and 6'

results to provide not only better topology mappings but also be accompanied by more or less the same utilizations than SLURM/Backfill.

The results we have obtained on the emulated 1024 node system show that our AUSCHED3 plug-in is able to generate better topological mappings than SLURM/Backfill, as shown in Figure 3 in all types of workloads. It can do this while keeping system utilization levels even higher than that of SLURM/Backfill in the case of workloads 1,2,5,6,5',6'. In the case of workloads 3 and 4, utilizations drop slightly by 4% and 5% respectively. However, considering the fact that the switch levels of AUCSCHED3 mappings are lower, the execution times are likely to be shorter due to fast communication and hence the differences in utilizations in these cases are likely to be smaller.

Overall, we observe that especially in our workloads involving jobs that request both CPU and GPU resources, AUCSCHED3 is able to generate topologically better mappings of jobs with improved system utilization.

In the future, we plan to carry out more exhaustive performance study of AUCSCHED3 using workloads with different distributions and systems with different architectures. We also plan to add capabilities to do topologically aware mappings for the torus architecture.

Acknowledgements

This work was financially supported by the PRACE-2IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493. The authors have used the Bogazici University Polymer Research Center's cluster to carry out the tests in the paper.

References

1. Slurm topology guide. Online, 2014. `http://slurm.schedmd.com/topology.html`.

2. Abhinav Bhatele. *Automating topology aware mapping for supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 2010.
3. J. P. Pascual, J. Navaridas, and J. M. Alonso. Effects of topology-aware allocation policies on scheduling performance. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 5798 of *Lecture Notes in Computer Science*, pages 138–156. Springer Berlin Heidelberg, 2009.
4. Yiannis Georgiou and Matthieu Hautreux. Evaluating scalability and efficiency of the resource and job management system on large hpc clusters. In *Job Scheduling Strategies for Parallel Processing*, pages 134–156. Springer, 2013.
5. C. Ozturan S. Soner and I. Karac. Extending slurm with support for gpu ranges. Technical report, PRACE, 2013. `http://www.prace-project.eu/IMG/pdf/wp80-2.pdf`.
6. S. Soner and C. Ozturan. Integer Programming Based Heterogeneous CPU-GPU Cluster Schedulers for Slurm Resource Manager. *Journal of Computer and System Sciences*, 2014, doi:10.1016/j.jcss.2014.06.011.
7. S. Soner and C. Ozturan. An auction based slurm scheduler for heterogeneous supercomputers and its comparative performance study. Technical report, PRACE, 2013. `http://www.prace-project.eu/IMG/pdf/wp59_an_auction_based_slurm_scheduler_heterogeneous_supercomputers_and_its_comparative_study.pdf`.
8. IBM ILOG CPLEX. Online, 2013. `http://www-01.ibm.com/software/integration/optimization/cplex/`.
9. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.