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

Partnership for Advanced Computing in Europe

Extending SLURM with Support for GPU Ranges

Seren Soner[a], Can Özturan[a,*], Itir Karac[a]

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

Abstract

SLURM resource management system is used on many TOP500 supercomputers. In this work, we present enhancements that we added to our AUCSCHED heterogeneous CPU-GPU scheduler plug-in whose first version was released in December 2012. In this new version, called AUCSCHED2, two enhancements are contributed: The first is the extension of SLURM to support GPU ranges. The current version of SLURM supports specification of node range but not of GPU ranges. Such a feature can be very useful to runtime auto-tuning applications and systems that can make use of variable number of GPUs. The second enhancement involves the implementation of a new integer programming formulation in AUCSCHED2 that drastically reduces the number of variables. This allows faster solution and larger number of bids to be generated. SLURM emulation results are presented for the heterogeneous 1408 node Tsubame supercomputer which has 12 cores and 3 GPU's on each of its nodes. AUCSCHED2 is available at http://code.google.com/p/slurm-ipsched/.

1. Introduction

SLURM [1] is a popular resource management system is used on many TOP500 supercomputers. As many as 30% of the supercomputers in the November 2012 TOP500 list were reported to be using SLURM [2]. An auction based heterogeneous CPU-GPU SLURM scheduler plug-in, called AUCSCHED, was developed and released in December 2012 [3]. In this work, we present enhancements that we added to our earlier AUCSCHED plug-in which has recently been released as AUCSCHED2. In the rest of the paper, we use AUCSCHED1 to refer to the earlier version. Two main enhancements are contributed in AUCSCHED2:

1. *Extension of SLURM to support GPU ranges*: The current version of SLURM supports specification of node range but not of GPU ranges. Such a feature can be very useful to runtime auto-tuning applications and systems that can make use of variable number of GPUs (or other accelerators).
2. *A new integer programming (IP) formulation:* The new AUCSCHED2 IP greatly reduces the number of variables and hence allows faster solution and larger number of bids to be generated.

Currently, if a user wants to run his job on GPUs, he can submit the job with the following command to SLURM:

`srun -n` x `-N` y `--gres=gpu:`z

and the system will allocate x cores on y nodes with z GPUs on each node to the job. However, the job may have the capability to tune itself at runtime to make use of any number of GPUs on the nodes assigned to it. In such a case, specification of an exact number of GPUs may delay the starting of the job if the nodes with the specified exact number of GPUs are not available. Another negative consequence may also be possible: if less than z GPUs per node are available, and yet, if there are no other jobs in the queue requesting less than z GPUs per node, this implies these GPUs will not be allocated and hence not utilized during the time interval in question. In our earlier work [3], our SLURM emulation tests showed that when a good fraction of jobs in the workloads request multiple GPUs per node, the system utilization dropped drastically to 65-75% range both when our AUCSCHED1 as well as SLURM's own plug-ins were used. When heterogeneous resources are present in a system and when job requests are also heterogeneous (i.e. varied), it may be more difficult to find a matching allocation especially if the resource requests are specified as exact numbers. This in turn may lead

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

to a lower utilization of the system. Motivated by these, we want to let SLURM users be able to request a a range of GPUs on each node. In AUCSCHED2, users may submit their job as follows:

srun -n x -N y --gres=gpu:z_{min}-z_{max}

where z_{min} is the minimum number of GPUs, and z_{max} is the maximum number of GPUs requested on each node.

This whitepaper is organized as follows: In Section 2, we present the new IP formulation incorporating GPU ranges using areduced number of variables. In Section 3, we present details of how GPU ranges support is integrated into SLURM. In Section4, SLURM emulation results are presented for the heterogeneous 1408 node Tsubame supercomputer which has 12 cores and 3 GPUs on each of its nodes. Finally, the whitepaper is concluded with a discussion of results and future work in Section 5.

2. New AUCSCHED2 Integer Programming Formulation

Given a window of jobs from the front of the job queue, a number of bids are generated for each job in the bid generation phase as explained in detail in AUCSCHED1 whitepaper [3]. An allocation problem that maximizes an objective function involving priorities of jobs is then solved as an IP problem. The IP problem is solved using the CPLEX [4] solver. Table 1 shows the list of symbols and their meanings in AUCSCHED2 formulation. The new objective and the constraints of the optimization problem are as follows:

Table 1: List of main symbols, their meanings, and definitions

	Symbol	Meaning
inputs	R_j^{gpu}	number of GPUs per node requested by job j
	A_n^{cpu}	number of available CPU cores on node n
	A_n^{gpu}	number of available GPUs on node n
	P_j	priority of job j
	F_i	preference value of bid i ranging between 0 and 1.
	α	a constant multiplying F_i
	T_{in}	no. 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	set of nodes : $N = \{n_1, \ldots, n_{\|N\|}\}$
	B_j	set of bid indices for job j : $B_j = \{i_1, \ldots, i_{\|B_j\|}\}$
variables	b_i	binary variable corresponding to bid i

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

subject to constraints :

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

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

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

The objective function given by equation 1 is the same as the one in the previous AUCSCHED1 formulation [3]. It maximizes the summation of selected bids' priorities with positive contribution $\alpha \cdot F_{jc}$ added to the priority in order to favour bids with less fragmentation (this is explained in detail in [3]). In AUCSCHED2, in order to favour a job's bids with a higher number of GPUs over this job's other bids, the preference value calculation of F_i has also been modified. Constraint 2 ensures that at most one bid is selected for each job. Constraint 3 makes sure that the number of allocated cores to jobs do not exceed available (free) number of cores on each node. T_{in} is the number of cores on node n that is requested by bid i. This information is determined during bid generation phase, and it is an input to the IP solver and not a variable. Constraint 4 ensures that the number of GPUs allocated to jobs are available on each node. U_{in} is a boolean parameter, and is set to 1 in the bid generation phase if T_{in} is positive, 0 otherwise.

In this new formulation, the only type of variable is b_i, and hence, the total number of variables is $|B|$. The number of constraints is also reduced to $|J| + 2|N|$. When compared with those in AUCSCHED1, these numbers are greatly reduced enabling us to increase the number of bids in the bid generation phase. This in turn helps us to find better solutions.

3. SLURM Implementation

AUCSCHED2 is implemented in a very similar fashion as AUCSCHED1. Hence, the whitepaper [3] can be consulted for implementation details regarding preference value calculation, nodeset generation, bid generation and general working logic of AUCSCHED1. AUCSCHED2 is implemented as a scheduler plug-in along with resource selection plug-in ipconsres (which is a variation of SLURM's own cons_res plug-in). Here, we just present AUCSCHED2 specific details.

In AUCSCHED2, we have modified the options processing files for *sbatch*, *srun*, and *salloc*; which are various job submission commands used in SLURM. We have also modified the *gres.c* file in order to parse the requests for a range of GPUs. In the original SLURM implementation, the *gres.c* file parses the string `gpu:z` in the submission `srun -n` x `--gres=gpu:`z. We extend this capability by allowing the user to submit the job by `srun -n` x `--gres=gpu:`$z_{min} - z_{max}$, which allows the job to request at least z_{min}, at most z_{max} GPUs per node. This submission is still backwards compatible; submitting a job using `srun -n` x `--gres=gpu:`z will still work as in the original SLURM implementation.

When a job is submitted with the `srun -n` x `-N` y `--gres=gpu:`$z_{min} - z_{max}$ option, the bid generator will try to bid on nodesets which have

- at least x cores on y nodes
- at least z_{min} GPUs on each node.

Several bids will be generated for the same job, but the bids may vary in the number of GPUs requested. Therefore, to distinguish between different bids, we have modified our preference value calculation slightly to increase the preference value of bids which request more GPUs. It should be noted that preference value only affects the selection between bids of the same job.

4. Workloads and Tests

In order to test our AUCSCHED2 plug-in, we conduct 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. The tests are conducted on a local cluster system with 9 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 1408 nodes in a supercomputer system that we emulate. The 1408 node system that we emulate is Japan's Tsubame [5] system which has 12 cores and 3 NVIDIA M2090 GPU's on each of its nodes.

We design a workload that takes into consideration the types of jobs that can be submitted to SLURM. There are seven types of jobs with each type making a different resource request as described below. When testing SLURM/Backfill, we use type A,B,C,D,E jobs. When testing AUCSCHED2 without GPU ranges, we use A,B,C,D,E type jobs. AUCSCHED2 that offers GPU ranges support is tested by using A,B,C',D',E type jobs. The probability of each job type in the workload is taken as equal.

While generating the workloads, for job types C' and D', we assume that the execution time of each job is inversely related to the number of GPUs allocated to that job. Our motivation in using this assumption is that, such jobs with GPU ranges linearly scale their performance with the number of GPUs allocated on a node. Whereas this assumption can be questioned, we believe it is reasonable. Since on a node, there are a few GPU cards and if there is too much overhead with the use of variable number of GPUs on a node, the user might as well use a specific (exact) number of GPUs on a node that gives the best performance and not submit the job with GPU ranges. Given the execution time t when the minimum number of GPUs is used, the execution time in AUCSCHED2 is calculated as follows:

$$t' = t \cdot \frac{minimum\ no.\ of\ GPUs\ requested}{no.\ of\ allocated\ GPUs} \tag{5}$$

For example, suppose a job of type C' takes 150 seconds when running with 1 GPU on each node and it requests at least 1 and at most 3 GPUs on each node. If the job is allocated 2 GPUs, the execution time of that job will be taken as 75 seconds.

Table 2: Job types in the workload

Job Type	Description	SLURM job submission
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`

We also have a parameter that sets the ratio of jobs that request contiguous allocation in the workload. For *NC* workloads, none of the jobs requests contiguous resource allocation. For *HC* workloads, approximately half of the jobs request contiguous resource allocation. For *FC* workloads, all of the jobs request contiguous resource allocation. We have generated two workloads of different lengths in order to test the scheduler's performance more extensively. Workloads 1 and 2 have a mean theoretical runtime of 5 and 10 hours, respectively. The number of jobs in the workloads are 350 and 700, for workloads 1 and 2, respectively.

In Table 3, we report the theoretical run time, average run time, CPU utilization, GPU utilization, average waiting time, average fragmentation of each job in the workload, for SLURM's Backfill and AUCSCHED2 without jobs having GPU ranges job and AUCSCHED2 with jobs having gpu ranges. We note that every test has been performed 27 times, and the resulting values are the average over all workloads of the same type. The definitions of reported results are as follows:

- *CPU Utilization:* The ratio of the theoretical run-time to the observed run-time taken by the test (workload). Run-time is the time it takes to complete all the jobs in a workload. Theoretical run-time is taken to be the summation of duration of each job times its requested core count, all divided by the total number of cores. Note that this definition of theoretical run-time we use is only a lower bound ; it is not the optimal value. Computation of the optimal schedule and hence the optimal run-time value is an NP-hard problem.
- *GPU Utilization:* The ratio of the total allocated GPU time to the total available GPU time. The available GPU time is calculated by multiplying the run-time taken by the test, the number of GPUs at each node and the number of nodes. The allocated GPU time is calculated by summing up the total number of GPUs allocated at each time at any node.
- *Waiting time:* Time from job submission until scheduling.
- *Fragmentation:* Number of contiguous node blocks allocated to a job.

5. Discussion and Conclusions

When comparing SLURM's implementation with our the GPU ranges implementation, the tests performed on the synthetically generated workloads show that GPU ranges feature can result in a 2-6% higher CPU utilization when multiple GPUs are used. Average fragmentation and runtimes are also slightly lower or equal in all the test cases. We also observe a 1-5% increase in the GPU utilizations in our GPU ranges implementation. The majority of average waiting times are also slightly lower. Hence, our tests provide convincing evidence that the GPU ranges feature can be useful to users and to supercomputer centers.

As identified in [3], low utilizations are obtained when there exists a sizeable fraction of jobs that use multiple GPU cards in the workload. As we observed in this work, GPU ranges can lessen the effects of this, albeit, only slightly. Currently, our GPU ranges implementation allocates the same number of GPUs on all the nodes. For example, if a range of 1-3 GPUs is given, the number of GPUs on all the nodes in a single bid is the same. (i.e. they can be 1 GPU per node or they can be 2 GPUs on *all* nodes requested, etc). The R_j^{gpu} term in constraint 4 in the IP formulation implements this. This is the most intuitive way to implement this, since, if allocated nodes can have different numbers of GPUs (while still being in the same range), load balancing issues arise. On the other hand, if the application can tune load balancing of its processes when different numbers of GPUs are allocated on the nodes then we can possibly expect such flexible GPU range implementations to lead to higher utilizations. In the future, we may implement this additional flexibility, perhaps, by making use of our PMAXFLOW maximum flow solver[6].

Our new IP formulation, besides having the advantage of reduced number of variables, can also help us to implement other heuristics: For example, linear programming (LP) relaxed heuristics. As we head towards

Table 3: Results for the emulated Tsubame system

Workload Type	Average Theo. Runtime (hours)	Method	CPU Util.	GPU Util.	Average Runtime (hours)	Average Waiting Time (hours)	Average Fragmentation
1.NC	3.12	SLURM/Backfill	64%	65%	4.87	1.30 ± 1.36	1.47 ± 1.01
		AUCSCHED2	65%	66%	4.79	1.37 ± 1.26	1.45 ± 1.09
		AUCSCHED2 (gpu ranges)	67%	69%	4.65	1.25 ± 1.30	1.47 ± 1.15
1.HC	3.12	Backfill	60%	69%	5.20	1.22 ± 1.38	1.29 ± 0.69
		AUCSCHED2	64%	71%	4.87	1.34 ± 1.21	1.23 ± 0.65
		AUCSCHED2 (gpu ranges)	66%	72%	4.72	1.21 ± 1.30	1.22 ± 0.63
1.FC	3.24	SLURM/Backfill	62%	64%	5.05	1.46 ± 1.47	1.00 ± 0.00
		AUCSCHED2	63%	64%	5.13	1.38 ± 1.52	1.00 ± 0.00
		AUCSCHED2 (gpu ranges)	66%	65%	4.98	1.34 ± 1.50	1.00 ± 0.00
2.NC	6.21	SLURM/Backfill	67%	67%	9.28	2.47 ± 2.52	1.36 ± 0.72
		AUCSCHED2	68%	68%	9.15	2.78 ± 2.36	1.22 ± 0.73
		AUCSCHED2 (gpu ranges)	71%	70%	8.76	2.98 ± 2.40	1.19 ± 0.64
2.HC	6.17	SLURM/Backfill	63%	64%	9.79	3.12 ± 2.88	1.19 ± 0.54
		AUCSCHED2	64%	65%	9.64	2.94 ± 2.37	1.13 ± 0.59
		AUCSCHED2 (gpu ranges)	66%	68%	9.35	2.89 ± 2.35	1.08 ± 0.58
2.FC	6.34	SLURM/Backfill	61%	62%	10.39	3.06 ± 2.77	1.00 ± 0.00
		AUCSCHED2	62%	64%	10.22	3.28 ± 2.42	1.00 ± 0.00
		AUCSCHED2 (gpu ranges)	63%	67%	10.05	3.14 ± 2.50	1.00 ± 0.00

exascale computing with larger number of resources on the supercomputers, resource allocation algorithms that are sophisticated and yet, at the same time, scalable will be needed.

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. AndyB. Yoo, MorrisA. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*. 2003.
2. Slurm Used on the Fastest of the TOP500 Supercomputers, 2012. `http://www.hpcwire.com/hpcwire/2012-11-21/slurm_used_on_the_fastest_of_the_top500_supercomputers`.
3. 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`.
4. IBM ILOG CPLEX. Online. `http://www-01.ibm.com/software/integration/optimization/cplex/`.
5. TSUBAME 2.0 Hardware and Software Specifications, 2011. `http://www.gsic.titech.ac.jp/sites/default/files/TSUBAME_SPECIFICATIONS_en_0.pdf`.
6. S. Soner and C. Ozturan. Experiences with parallel multi-threaded network maximum flow algorithm. Technical report, PRACE, 2013.