JOB SCHEDULING FOR HETEROGENEOUS SUPERCOMPUTERS

by

Seren Soner

B.S, in Chemical Engineering, Boğaziçi University, 2007

M.S, in Chemical Engineering, Boğaziçi University, 2009

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering
Boğaziçi University
2016

JOB SCHEDULING FOR HETEROGENEOUS SUPERCOMPUTERS

APPROVED BY:

Prof. Can Özturan (Thesis Supervisor)	....
Assoc. Prof. D. Turgay Altılar	....
Assoc. Prof. Alper Şen	....
Prof. Haluk Rahmi Topçuoğlu	....
Prof. Oğuz Tosun	....

DATE OF APPROVAL: 13.01.2016

ACKNOWLEDGEMENTS

First and foremost, I would like to offer my most sincere gratitude to my advisor Prof Can Özturan. He has supported me both mentally and scientifically during my PhD thesis. He has always pushed me forward and been understanding and friendly at the same time. I could not ask for a thesis advisor that was friendlier, more helpful, or a better guide.

I would like to thank Prof. Oğuz Tosun and Assoc. Prof. D. Turgay Altılar for their valuable inputs over the years. I would also like to thank Prof. Haluk Rahmi Topçuoğlu and Assoc. Prof. Alper Şen for reading and commenting on the thesis.

I would like to thank my friends at the Polymer Research Center for their help, friendliness. They were always the best type of distractions one can hope for in an office. I was really lucky to have you as colleagues.

I would like to acknowledge the financial support from TÜBİTAK BİDEB 2011. This work was also financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528, FP7-261557 and RI-283493. Bogazici University Polymer Research Center's cluster has been used for the tests funded by DPT project 2009K120520.

Last but not least, I would like to offer my thanks to my parents, my sister, and my wife, Dr. Zeynep Kurkcuoglu Soner. I am forever indebted to for their endless care and love. This thesis (along with everything else I have in life) was made possible thanks to your support and being with me whenever I need someone.

ABSTRACT

JOB SCHEDULING FOR HETEROGENEOUS SUPERCOMPUTERS

This thesis addresses the job scheduling problem for heterogeneous supercomputers where accelerators such as GPGPUs or co-processors are employed. On homogeneous supercomputers, the problem of scheduling user jobs to the available resources is NP-hard. Heterogeneous systems make the scheduling problem combinatorially more difficult. In this thesis, we aim to (i) design a new class of scheduling algorithms for state-of-the-art heterogeneous supercomputers, (ii) implement these scheduling algorithms as ready to use open source plugin software (iii) demonstrate the effectiveness of these algorithms by emulating real life usages. We propose four different models to solve the scheduling problem on heterogeneous supercomputers. In the first model, we formulate a simple co-allocation problem that does not take topology into consideration. In the second model, we implement the problem as an auction problem and automatically generate multiple bids for each job by assuming a one dimensional system topology. In the third model, we support moldable jobs that may request a range of resources. In our fourth model, we also consider topologically aware scheduling for hierarchical fat tree interconnection architectures. All of these models are formulated as integer programming problems and are solved periodically at each scheduling step. We use existing workloads to test the performance of our scheduling algorithms and also develop our own workload generator that generates realistic workloads for heterogeneous systems. The tests carried out show that our algorithms perform better than the traditional backfilling algorithm in terms of system utilization, average job waiting time and/or job fragmentation.

ÖZET

HETEROJEN SÜPERBİLGİSAYARLAR İÇİN İŞ ÇİZELGELEME

Bu tez, GPGPU veya ekişlemci gibi hızlandırıcıların kullanıldığı heterojen süperbilgisayarlardaki iş çizelgeleme problemini ele almaktadır. Homojen süperbilgisayarlarda, mevcut kaynaklara kullanıcı işlerinin çizelgelenmesi problemi NP-zor sınıfındadır. Heterojen sistemler iş çizelgeleme problemini birleşimsel olarak daha zor yapmaktadırlar. Bu tezde amaçlarımız (i) son teknoloji heterojen süperbilgisayarlar için yeni tür iş çizelgeleme algoritmaları tasarlamak, (ii) bu algoritmaları kullanılmaya hazır açık kaynak kodlu yazılım ekleri olarak gerçekleştirmek ve (iii) bu algoritmaların etkinliğini gerçek hayat kullanımını öykünerek göstermektir. Heterojen süperbilgisayarlardaki iş çizelgeleme problemini çözmek için dört farklı model önerilmiştir. İlk modelde, topolojiyi dikkate almayan basit bir beraber tahsis etme problemi formülleştirilmiştir. İkinci modelde, problemi bir müzayede problemi olarak ele alıp ve bir boyutlu bir sistem topolojisi varsayarak her iş için otomatik olarak birden fazla teklif yaratılmıştır. Üçüncü modelde, sayı aralıkları kullanarak kaynak istekleri yapabilen şekillendirilebilir işler desteklenmiştir. Dördüncü ve son modelimizde, hiyerarşik şekilde bağlanmış şişman-ağaç topolojisi de dikkate alınmıştır. Tüm bu modeller tam sayı programlama problemi olarak formüle edilip, her çizelgeleme adımında periyodik olarak çözülmektedir. Çizelgeleme algoritmalarının başarımlarını test etmek için daha önceden tanımlanmış olan iş yüklerine ek olarak, heterojen sistemler için daha gerçekçi iş yükleri yaratacak olan kendi iş yükü üreticimiz de geliştirilmiştir. Yapılan testler algoritmalarımızın geleneksel geri dolgulama algoritmalarından sistem kullanımı, ortalama iş bekleme süresi ve/veya iş parçalanması açılarından bakınca daha iyi performans ortaya koyduğunu göstermektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES viii
LIST OF TABLES xii
LIST OF SYMBOLS xv
1. INTRODUCTION 1
  1.1. Architectures of the State-of-the-Art Supercomputers 1
  1.2. A Scheduling Example 7
  1.3. Contributions of this Thesis 9
  1.4. Outline of this Thesis 10
2. A SURVEY of JOB SCHEDULING in SUPERCOMPUTERS 12
  2.1. Scheduling Algorithms 14
    2.1.1. Job Scheduling in Heterogeneous Supercomputers 17
    2.1.2. Topology Awareness 19
      2.1.2.1. Fat-Tree Topology 21
    2.1.3. Energy Awareness 22
  2.2. Complexity of Scheduling 23
    2.2.1. Integer Programming 23
      2.2.1.1. Knapsack Problem 25
      2.2.1.2. Exact and Heuristic Solutions 27
    2.2.2. Integer Programming in Scheduling and Auction Mechanisms . 28
3. SIMPLE INTEGER PROGRAMMING BASED SCHEDULER 30
  3.1. Formulation 30
  3.2. IPSCHED Slurm Implementation 35
4. AUCTION BASED SCHEDULER WITH LINEAR TOPOLOGY SUPPORT 37
  4.1. Bid generation phase 45
  4.2. Preference value calculation 47
  4.3. AUCSCHED1 Slurm Implementation 49

5. AUCTION BASED SCHEDULER WITH MOLDABILITY SUPPORT . . . 51
5.1. Improved IP Formulation 51
5.2. aucsched2 Slurm Implementation 55
6. AUCTION BASED SCHEDULER WITH HIERARCHICAL INTERCONNECT SUPPORT 58
6.1. aucsched3 Slurm Implementation 62
7. EXPERIMENTAL SETUP 65
7.1. Slurm Implementation 65
7.2. Simulation vs Emulation 66
7.3. Workloads and Test Conditions 67
7.3.1. ESP Workloads 67
7.3.2. Our workload generator 68
7.3.3. Workloads generated with GPU ranges 70
7.4. Performance Metrics 72
8. RESULTS and DISCUSSION 75
8.1. General comparison of the models introduced 75
8.2. Comparison of IPSCHED and AUCSCHED1 76
8.3. Comparison of AUCSCHED2 with AUCSCHED1 80
8.4. AUCSCHED3 Tests 89
8.4.1. Tests on a hypothetical 1024 node supercomputer 89
8.4.2. Tests on a simulated 1408 node TSUBAME 2.0 supercomputer . 93
9. CONCLUSIONS 97
9.1. Future Work 98
APPENDIX A: RESULT TABLES 100
REFERENCES 111

LIST OF FIGURES

Figure 1.1. K-computer system configuration overview. 2

Figure 1.2. TSUBAME 2.0 system architecture. 4

Figure 1.3. Example assignment for *i)* backfilling algorithm and *ii)* co-allocation. 8

Figure 2.1. Slurm components. 15

Figure 2.2. Scheduling for an example of four jobs using *(a)* FCFS and *(b)* backfilling. 17

Figure 2.3. An example fat-tree with 16 compute nodes, 4 intermediate level and 2 root level switches. 21

Figure 3.1. Scheduling of windows of jobs. 30

Figure 3.2. IPSCHED scheduling steps. 34

Figure 3.3. An example assignment of jobs to 16 nodes by IPSCHED. 34

Figure 4.1. Auction Based Scheduler with Linear Topology Support (AUCSCHED1). 42

Figure 4.2. A sample assignment of jobs to 16 nodes by AUCSCHED1. 43

Figure 4.3. Detailed nodeset generation, bid generation and solution figure for a 144 node system. 44

Figure 4.4. Determination of nodesets for a 12-node system. 46

Figure 4.5. Bid generation algorithm. 48

Figure 5.1. A sample assignment of jobs to 16 nodes by AUCSCHED2. 55

Figure 6.1. Example switch configurations: (a) a tree and (b) a fat-tree. . . . 58

Figure 6.2. An example assignment of jobs to 16 nodes on a two level tree by AUCSCHED3. 63

Figure 7.1. Fragmentation and spread values for two jobs, J_1 and J_2. 73

Figure 7.2. An example showing job spread and lowest level common switch metrics. 74

Figure 8.1. Comparison of system utilization in different experiments for ESP workloads. 78

Figure 8.2. Comparison of system utilization in different experiments for our workloads. 79

Figure 8.3. Comparison of job waiting times (in hours) in different experiments for ESP workload. 80

Figure 8.4. Comparison of job waiting times (in hours) in different experiments for our workloads. 81

Figure 8.5. Comparison of job packing factors in different experiments for ESP workload. 82

Figure 8.6. Comparison of job fragmentation in different experiments for ESP workload. 83

Figure 8.7. Comparison of job fragmentation in different experiments for our workloads. 84

Figure 8.8. Comparison of job spread in different experiments for ESP workload. 85

Figure 8.9. Comparison of job spread in different experiments for our workloads. 86

Figure 8.10. Utilization comparison in the tests. 87

Figure 8.11. Fragmentation comparison in the tests. 87

Figure 8.12. Spread comparison in the tests. 88

Figure 8.13. Average lowest level common switch of AUCSCHED3 compared to Slurm/Backfill plug-in. 91

Figure 8.14. Average job spread of AUCSCHED3 compared to Slurm/Backfill plug-in. 91

Figure 8.15. Distribution of percentage of jobs over switch levels (a) and percentage of total job durations over switch levels (b) for all workloads. 92

Figure 8.16. 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’. 92

Figure 8.17. Utilization comparison between AUCSCHED3 and Slurm/Backfill for simulated TSUBAME 2.0 system with different generated workloads as described in Table 8.6. 95

Figure 8.18. Lowest level common switch comparison between AUCSCHED3 and Slurm/Backfill for simulated TSUBAME 2.0 system with different generated workloads as described in Table 8.6. 96

Figure 8.19. Job spread comparison between AUCSCHED3 and Slurm/Backfill for simulated TSUBAME 2.0 system with different generated workloads as described in Table 8.6. 96

LIST OF TABLES

Table 1.1. Top ranked supercomputers in November 2015 Top500 List. 5

Table 1.2. Example illustrating advantage of solving collective allocation problem. 7

Table 2.1. Slurm plug-ins and their tasks. 15

Table 2.2. Different scheduling problems and their complexities and heterogeneity levels. 24

Table 3.1. List of main symbols, their meanings, and definitions for IPSCHED. 31

Table 3.2. Number of variables in IPSCHED IP formulation. 33

Table 3.3. Number of constraints in IPSCHED IP formulation. 33

Table 4.1. List of symbols, their meanings, and definitions for AUCSCHED1. 38

Table 4.2. Number of Variables in AUCSCHED1 model IP formulation. . . . 41

Table 4.3. Number of Constraints in AUCSCHED1 IP formulation. 41

Table 4.4. Constants for F_{jc} calculation for different job and bid classes. . . . 49

Table 5.1. List of symbols, their meanings, and definitions for Auction Based Scheduler with Linear Topology Support. 52

Table 5.2. Constants for F_i calculation for different bid types in AUCSCHED2. 54

Table 6.1. List of symbols, their meanings, and definitions for AUCSCHED3. 59

Table 7.1. ESP Benchmark Job Characteristics. 69

Table 7.2. Workload types and job distributions. 70

Table 7.3. Job types in the workload. 71

Table 8.1. Number of variables in the IP problems for all models developed in this thesis. 75

Table 8.2. Features for all models developed in this thesis. 76

Table 8.3. Experiments and their characteristics. 77

Table 8.4. Workload types and job distributions for AUCSCHED2 test. . . . 86

Table 8.5. Workload types and job distributions. 89

Table 8.6. Workload types and job distributions. 94

Table A.1. IPSCHED, AUCSCHED1 and Backfill comparison results for ESP workload. 100

Table A.2. IPSCHED, AUCSCHED1 and Backfill comparison results for our workload generator. 101

Table A.3. Results for AUCSCHED2, AUCSCHED1 and Backfill for the emulated Tsubame 2.5 system. 106

Table A.4. AUCSCHED3 and Backfill comparison for an emulated 1024 node hypothetical supercomputer. 109

Table A.5. AUCSCHED3 and Backfill comparison for an emulated TSUBAME 2.0 supercomputer using workloads in Table 8.6. 110

LIST OF SYMBOLS

A_n^{cpu}	Number of available CPU cores on node n
A_n^{gpu}	Number of available GPUs on node n
b_i	binary variable corresponding to bid i
b_{jc}	Binary variable for a bid on class c of job j
B	Set of all bids, $B = \{b_1, \ldots, b_{\lvert B\rvert}\}$
B_j	set of bid indices for job j : $B_j = \{i_1, \ldots, i_{\lvert B_j\rvert}\}$ variables
c_j	node packing variable for a job j
C	Set of bid classes : $C = \{c_1, \ldots, c_{\lvert C\rvert}\}$
C_{jn}	The set $\{c \in C \mid c \in B_j \ \ and \ \ n \in N_c\}$
F_{jc}	Preference value of bid c of job j in the interval $(0, 1]$.
J	Set of jobs that are in the window: $J = \{j_1, \ldots, j_{\lvert J\rvert}\}$
K_{ji}	A heuristically assigned cost value in $(0, 3]$ of bid i of job j
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
N	Set of nodes : $N = \{n_1, \ldots, n_{\lvert N\rvert}\}$
N_c	Set of nodes making up a class c
N_i	Number of nodes requested by bid i
P_j	Priority of job j
P_{min_diff}	Minimum absolute priority difference between any pair of jobs in the window
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)
R_j^{cpn}	Number of cores per node requested by job j. If not specified, this parameter gets a value of 0
R_j^{cpu}	Number of cores requested by job j
R_j^{gpu}	Number of GPUs per node requested by job j
R_j^{gres}	Number of generic resources per node requested by job j

$R_{j,min}^{node}$	Minimum number of nodes requested by job j
$R_{j,max}^{node}$	Maximum number of nodes requested by job j
R_{jc}^{cpu}	Number of cores requested by job j in bid c
R_{jc}^{node}	Number of nodes requested by job j in bid c
s_j	Binary variable indicating whether job j is allocated or not
t_{nj}	binary variable showing whether job j is allocated any resource on node n
T_{in}	no. of cores on node n requested by bid i.
u_{jn}	Binary variable indicating whether node n is allocated to job j
U_{in}	boolean parameter indicating whether bid i requires any resources on node n.
x_{nj}	no. of cores allocated to job j at node n
α	A factor multiplying the preference value F_{jc} so that the added preference values do not change the job priority maximizing solution
β	a constant multiplying K_{ji}

1. INTRODUCTION

The term supercomputer stands for the most predominant computer in terms of computational rate, memory, cost, or size [1]. The use of supercomputers has increased drastically in the last few decades. During 1970s, supercomputers were mainly used for weather forecasting and aircraft design [2]. During 1980s, galaxy formation was simulated [3]. During 1990s, supercomputers replaced wind tunnels in aeronautics research [3]. By 2000s, supercomputers were also used for investigating protein folding mechanisms [4]. Molecular dynamics simulations, cancer research, advanced seismic analysis are just a few of the many application areas in this decade that utilize supercomputers [5–7].

Technological progress has enabled more economical supercomputers to be developed. The approximate cost per giga-floating point operations per second (GFLOPS) in 1960s was 8.3 trillion USD [8], which fell down to 42,000 USD in 1990s [9]. Currently, this cost is around 0.08 USD [10]. Besides this cost, the computational capacity is also increasing. The first teraflop rate (1,068 GFLOPS) was reached in 1997 by Intel ASCI Red [9, 11] which had one thousandth flop rate of the IBM Roadrunner [12] in 2008 [13]. Currently, the fastest supercomputer, Tianhe-2 [14], has a maximal LINPACK benchmark [15] performance of 33,863 TFLOPS. Although the computational capacity has increased by a factor of 33 in the last 7 years, the power consumption only increased by a factor of 5, which is mainly due to the important emphasis put on green supercomputing [13, 16]. The current fastest supercomputer is in peta-FLOPS scale. In the next decade, the aim is to reach exa-FLOPS scale (also called exascale) [17].

1.1. Architectures of the State-of-the-Art Supercomputers

A distributed system is a collection of several computational entities, each having local memory. The communication between these entities are handled by message passing mechanisms. Supercomputers are distributed systems having massive computing power. The current supercomputer architectures include both homogeneous computa-

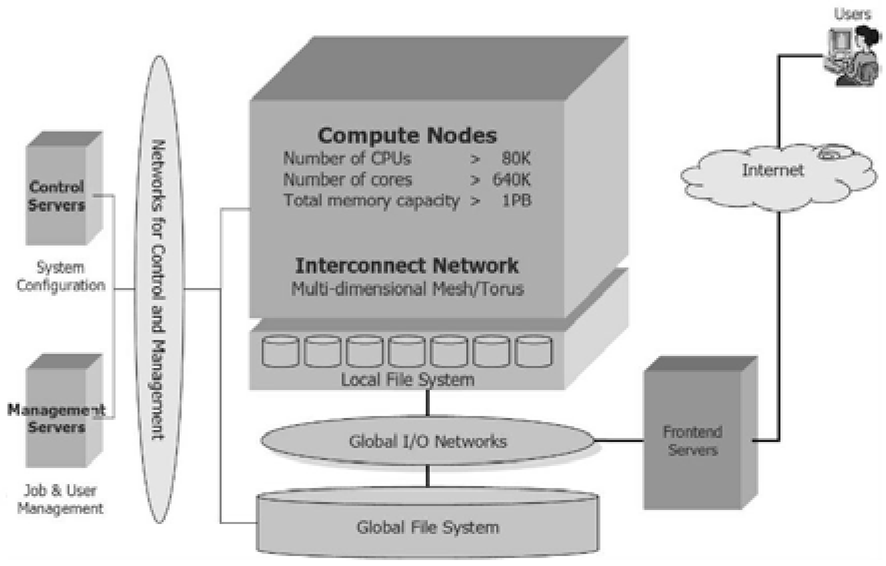


Figure 1.1. K-computer system configuration overview [18].

tional engines that are uniform and heterogeneous systems with multiple and different computational engines such as co-processors and/or GPUs.

The components of a supercomputer are generally the following:

(i) Compute nodes
(ii) Parallel file systems
(iii) Networking infrastructure (i.e. switches and cables)
(iv) Cooling equipment

The components of a compute node are typically the following:

(i) Processor(s)
(ii) Local memory
(iii) Local disk drive
(iv) Network connection
(v) Accelerators or co-processors (for heterogeneous supercomputers)

For example, a homogeneous supercomputer, K-computer consists of 76,800 computational nodes. In each of these computational nodes, there are 16 GB of memory, 8 cores, a torus network connection, local disks. Additionally, there is a global file system, accessible from each compute node. The global I/O network enables access of users to reach the file systems and the compute nodes. There is also a central resource job management system that allows job user management [18]. The architecture is shown in Figure 1.1.

A heterogeneous supercomputer has a hybrid architecture, and may include accelerators like Graphics Processing Units (GPU), Xeon Phis or Field Programmable Gate Array (FPGA) boards to carry out computational tasks. A notable example is TSUBAME 2.0 [19], which has 1408 computational nodes. Each of these nodes employs 16 cores and 3 NVIDIA M2070 Graphics Processing Units (GPUs) and local solid state disks (SSDs). The computation may be carried out on the cores and the GPUs. Ideally, hybrid jobs will make use of all of these resources. TSUBAME's compute nodes are connected via Infiniband [20] using a widely used fat-tree architecture (see Section 2.1.2.1 for detailed information on fat-tree architecture). These compute nodes are also connected to an external parallel file system with this Infiniband network. The description of the TSUBAME architecture is given in Figure 1.2.

Message Passing Interface [21] (MPI) is a communications protocol which aims high performance, scalability and portability on distributed systems. It is the most dominant communications protocol used in high performance computing. MPI can also run on shared memory computers.

All supercomputers in the most recent Top 500 list use Linux/Unix based operating systems [22]. In a traditional computer, the job scheduler is responsible for assigning processors and other resources such as disks etc. to a job. On a supercomputer, however, the job scheduler has to be responsible for not only thousands of cores but also for communication resources such as switch bandwidths and licenses etc. Therefore, job resource management systems of supercomputers have evolved into a complex piece of software. A resource job management system of a distributed system efficiently assigns the jobs submitted by the users to the system resources. Some of the

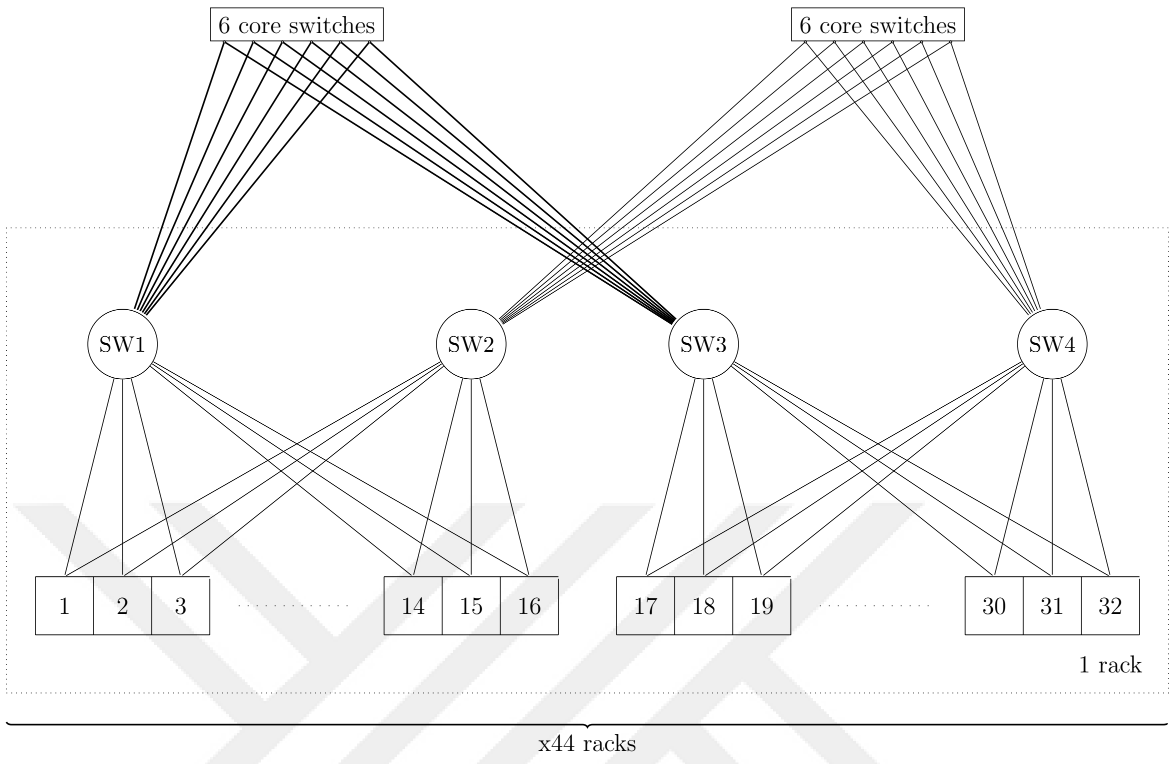


Figure 1.2. TSUBAME 2.0 system architecture.

available resource job management systems have been introduced in Chapter 2.

With the introduction of heterogeneity on the latest platforms, resource management of different hardware components, and scheduling of jobs onto these components have now become more important [23]. Some of the fastest supercomputers according to the Top 500 list published in November 2015 [22] are listed in Table 1.1. In the topmost 15 supercomputers in this list, there are seven heterogeneous supercomputers. The speeds reported in Table 1.1 are the maximum computational rate a computer can achieve when running the LINPACK benchmark [24].

The recent supercomputer designs use GPUs or Xeon Phis on heterogeneous compute nodes such as those on Tianhe-2 [14], Titan [25], Piz Daint [28], Stampede [29], TSUBAME 2.0 [19]. The two largest GPU manufacturers NVIDIA and AMD have a variety of GPUs that have thousands of smaller cores optimized for handling massively parallel arithmetic operations compared with multi-core CPUs with smaller number of cores [30]. Intel is also in the race for heterogeneous supercomputing with its coprocessor, Xeon Phi or Intel Many Integrated Core Architecture [31]. All of the latest models

Table 1.1. Top ranked supercomputers in November 2015 Top500 List [16].

Rank	**Name (Country)**	**Speed (PFLOPS)**	**Technical Properties (on each node)**
1	Tianhe-2 (China) [14]	33.86	16K nodes, 16 cores and Xeon Phi on each node
2	Titan (USA) [25]	17.59	18K nodes, 16 cores and Nvidia Tesla K20X GPU on each node
3	Sequoia (USA) [26]	17.17	98K nodes, 16 cores on each node
4	K-Computer (Japan) [18]	10.51	88K nodes, 8 cores on each node
5	Mira (USA) [27]	8.59	49K nodes, 16 cores on each node
7	Piz Daint (Switzerland) [28]	6.27	5K nodes, 8 cores and Nvidia Tesla K20X GPU on each node
10	Stampede (USA) [29]	6.27	6.4K nodes, 16 cores and Xeon-Phi co-processor on each node
25	TSUBAME 2.0 (Japan) [19]	2.78	1408 nodes, 16 cores and 3 Nvidia M2070 GPUs on each node

of the three manufacturers have a similar price, capacity, and power consumption; the price ranges between 3000 to 5000 USD, capacity ranges between 3 to 5 TFLOPS, and the power consumption ranges between 250 to 400 W [32–34].

Green 500 [35] is another Top500 list that aims to list the most energy-efficient supercomputers in the world. As a motivation for this list, it is stated that even though the theoretical flop rate, and the linpack performance metrics in terms of FLOPS are of interest, it is necessary to develop a new performance metric based on energy consumption since more and more energy (both in terms of electrical power and cooling) is needed to run the state-of-the-art supercomputers. This new metric considers the

FLOPS per the total energy consumed by the servers and the cooling facilities.

It should be noted that, of the topmost 40 supercomputers in the most recent Green 500 list in November 2015, 38 employ GPUs or co-processors to achieve higher computational rates with less energy consumption [36]. Two employ only CPU cores as computation resources. Job schedulers developed for these supercomputers have only focused on the management of mainly cores. Heterogeneous systems make scheduling problem combinatorially more difficult. With the advances in the recent years, GPUs and Intel's coprocessor Xeon Phi have been playing increasing role in scientific computing applications. The recently built supercomputers are employing Xeon Phi co-processors or GPUs in order to exploit higher parallelism levels. TianHe-2 system, which was built in 2013 and which ranks as first in the November 2015 Top500 List [22], employs a total of 16K compute nodes, each having 2 Intel Ivy Bridge Xeon CPUs (16 cores) and Xeon Phi co-processors [14]. Titan supercomputer [25], which ranks as second, has 18,688 nodes, each of which contains 16-core AMD Opteron 6274 CPUs and Nvidia Tesla K20X GPU's. TSUBAME 2.0 [19], built in 2011, has 1408 compute nodes with 2 Intel X5670 CPU's (12 cores) and 3 NVIDIA M2050 GPU's per node. TSUBAME 2.0 supercomputer ranks 26th in the Green500 list [35], and ranks as 25th in the November 2015 list with a capacity per power consumption of 2,952 GFLOPS/KW.

The emergence of such heterogeneous supercomputers also necessitates schedulers that can handle generic resources such as GPUs or coprocessors available on the nodes. The users of these supercomputers may submit jobs which utilize only CPU cores, or combination of CPU cores and one or more generic resource on each node. A scheduler of such heterogeneous systems needs to efficiently assign the available resources to the nodes, so that the jobs in the queue will not wait unnecessarily. Most of the schedulers, such as Slurm [37], employ a priority ordered queue and schedule the jobs by taking them one by one from the front of the queue, using a First-Come First-Served (FCFS) or backfilling algorithm. In FCFS, the scheduler starts the jobs from the priority ordered queue one by one. In backfilling algorithm, if the system cannot start the first job due to resource unavailability, it can start another job in the queue, as long as that job does not delay the first job's starting time (see Section 2.1 for more information on

FCFS and backfilling). The following example shows how such a scheduling mechanism based on taking one job at a time from the front of the queue may cause unnecessary waiting for some jobs in the queue.

1.2. A Scheduling Example

Consider a small heterogeneous cluster with 4 nodes. On this cluster, each node has 12 cores and 3 GPU's, similar to configuration of the nodes of the Tsubame supercomputer [19]. In this example, there are three jobs. The jobs J_1, J_2 and J_3, whose resource requests are given in Table 1.2 arrive in the given order. *i)* The backfilling

Table 1.2. Example illustrating advantage of solving collective allocation problem.

Job	**Resources Requested**
J_1	24 cores
J_2	6 cores and 2 GPUs per node
J_3	6 cores and 3 GPUs per node

algorithm would take the first job at the front of the queue, go through the list of nodes and best-fit the job's resource requirements to the nodes that have available resources. Therefore, the resulting allocation would be as follows:

- J_1 will be allocated the first 2 nodes, leaving the GPUs of that nodes unutilized.
- J_2 will be allocated the last 2 nodes, using 6 cores and 2 GPUs on each node, leaving half of the cores unutilized.
- After one of the jobs finishes, J_3 will be allocated resources, therefore, increasing the total turn around time for J_3.

ii) If we take all the jobs in the queue, and solve an assignment problem, we can come up with an efficient solution where all the jobs in the queue are allocated at the same time, and all the resources in the system are utilized, in the following manner.

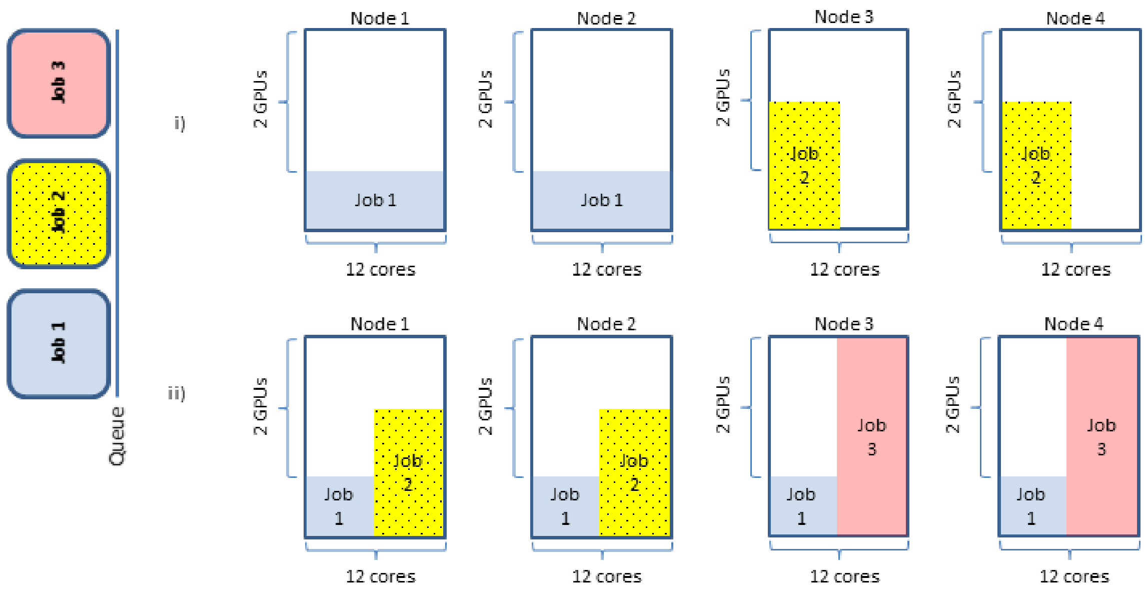


Figure 1.3. Example assignment for *i)* backfilling algorithm and *ii)* co-allocation.

- J_1 will be allocated 6 cores on all of the 4 nodes.
- J_2 will be allocated 6 cores and 2 GPUs on the first two nodes.
- J_3 will be allocated 6 cores and 3 GPUs on the last two nodes.

Figure 1.3 illustrates the system and the assignment outcomes for *i)* and *ii)*.

In order to overcome such problems we propose algorithms which instead of taking one job, take a window of jobs from the front of the queue and solve an integer programming problem at every scheduling step. We formulate an integer programming (IP) problem which is similar to an m-dimensional multiple knapsack problem formulation, in which the jobs have multiple dimensions (number of cores and number of resources per node), and there are multiple knapsacks (nodes). The difference from the m-dimensional multiple knapsack problem is that, the jobs can be allocated more than one node. A more detailed information on m-dimensional multiple knapsack problem has been given in Subsubsection 2.2.1.1.

1.3. Contributions of this Thesis

As we move towards the exascale computing, the emergence of heterogeneous supercomputers with massive numbers of cores and accelerators necessitate design and development of new generation of schedulers that can handle heterogeneity of the resources. The traditional scheduling problems on homogeneous systems require the solution of NP-hard problems. With the introduction of heterogeneity, even more combinatorially complex problems have to be solved. The good news is that new generation of fast integer programming solvers help us to solve combinatorially complex problems with large numbers of variables in a fast manner. The main objectives of this thesis are (i) the design of new class of scheduling algorithms for state-of-the-art heterogeneous supercomputers (ii) development of scheduling software that can easily and readily be used on the current supercomputers and (iii) demonstration of the effectiveness of newly developed algorithms not by simulation but rather in a setting that is close to the real life usage. To achieve these objectives, the thesis makes the following contributions:

(i) The jobs and the vast number of heterogeneous supercomputer resources are viewed as a market where jobs need to acquire resources. This view enables us tackle the complex heterogeneity scenario and develop a new class of job scheduling models where the jobs bid for resources that are most appropriate for themselves just like people bid for goods that have the most utility for themselves in auction markets.

(ii) Four new combinatorial optimization problems are developed for models ranging from a simple one involving no topology to an advanced one involving topology aware scheduling. These optimizations are formulated as integer programs and solved using state-of-the-art IP packages. Furthermore, based on the performance of the IP solvers for our IP formulations, automatic bid generation processes adjusts the number of bids generated so as to solve the resulting IP problems in a few seconds which makes the use of IP solvers in supercomputer jobs scheduling practical in real life settings.

(iii) Open source and free General Public License (GPL) licensed software in the form of a Slurm plug-in is developed that incorporates our various scheduling models and algorithms. The software can easily and readily be installed and used on a supercomputer system. It is distributed at: `https://github.com/aucsched/`.

(iv) The models and algorithms developed are tested in settings that are close to a realistic supercomputer usage scenario. Instead of simulation, more realistic emulation techniques are used to demonstrate effectiveness of our models and algorithms.

1.4. Outline of this Thesis

Chapter 2 briefly reviews the recent work in supercomputing, integer programming and knapsack problems in particular.

Chapter 3 covers our first model named Simple Integer Programming Based Scheduler. In this model, we formulate and solve a simple co-allocation problem that does not use any topology information. In our tests, we show that this model performs better in terms of system utilization and resource fragmentation than the Slurm's backfill model under different system conditions and settings.

Chapter 4 covers our second model named Auction Based Scheduler with Linear Topology Support. In this model, we view the job scheduling problem as an auction problem and multiple bids are generated for each job automatically. In this model, the topology of the system is assumed as linear one dimensional array. We compare our implementation of Auction Based Scheduler with Linear Topology Support with Simple Integer Programming Based Scheduler and Slurm's backfill method and show that in our tests this model performs better in terms of system utilization and resource fragmentation.

Moldable jobs are jobs that may run on different resource sizes and configurations, but these resources are fixed at the start of execution. Chapter 5 covers our third model named Auction Based Scheduler with Moldability Support. In this model, we introduce moldability support by allowing a job to request a range of generic resources. Such jobs can run with the defined minimal generic resource request, but may exhibit better performance with higher number of generic resources. This model is similar to the model defined in Chapter 4. However, the main difference is that, this model enables the moldability of generic resources at the start of the execution. With this feature, the users can submit a range for the generic resources they request so that they will get at least some number and at most some number of generic resources. If possible, the scheduler will allocate them higher number requested resources for better performance. This feature can be very useful for jobs with moldability capability. Additionally, in this model, the integer programming problem formulation is redefined. The number of variables in the new formulation is reduced drastically which in turn allows the system to generate higher number of bids and obtain better solutions.

Chapter 6 covers our fourth model named Auction Based Scheduler with Hierarchical Interconnect Support. In this model, a fat-tree system topology is considered. Instead of reducing the distances between the nodes on a one dimensional linear arrangement, this method aims to do topologically aware scheduling so as to reduce the number of links (hops) that need to be traversed while the job processes communicate.

In Chapter 7, available workloads in the literature, experimental settings and performance metrics are presented. In Chapter 8, comparisons of the models proposed in this thesis, results obtained for the four different scheduler models, and their comparisons with the current Backfill method are presented. Finally, the thesis is concluded in Chapter 9, and future work is discussed.

2. A SURVEY of JOB SCHEDULING in SUPERCOMPUTERS

High Performance Computing (HPC) clusters generally have a uniform hardware and configuration. This hardware is connected with high speed switches. In such clusters, job schedulers, together with resource management systems are used in order to schedule the jobs and watch the status of the available resources and the job queue.

In a cluster, when jobs are submitted, there are a few parameters that are taken into consideration by the scheduler. The priority of a job increases when the *waiting time* of a job in the queue increases. Some supercomputers assign higher priorities to the jobs with a larger *job size*, in order to encourage running larger parallel jobs in systems. In some cases, a certain user or a group may use a larger portion of the system for a certain amount of time. In order to allow fair usage of the system resources, a *fair-share* policy may be used, which assigns a lower priority to those that have already been consuming the system resources.

Most batch schedulers used in these HPC clusters prioritize the queue of pending jobs, and attempt to run the highest priority job. If no resources are available for this job, the scheduler tries to run the next job in the queue. In order to tune these schedulers, the administrator can either change the priority calculation or the resource selection.

In priority calculation, several factors can be used:

- Waiting time
- Job size
- Fair-share

In resource selection, several factors should be considered:

- Core and node requests of the jobs
- Additional constraints such as memory, GPU, disk requests
- Node-to-node communication delays
- Node contiguity
- Packing of jobs into a small set of nodes

A recent PhD thesis by Georgiou [38] makes an in-depth assessment of widely used job schedulers. Resource and job management systems aim to *(i)* satisfy users demands for carrying out their computations and *(ii)* achieve good utilization by efficient allocation of resources to jobs. Examples to the widely used resource job management systems include PBS Pro [39], Moab [40], TORQUE [41], Maui [42], and SGE [43]. Basically, all of the schedulers above support FCFS scheduling, backfilling, fair share, preemption (releasing a job from resources for higher priority jobs that are in the queue), multi-factor priority (allowing different factors to calculate the job priority), advanced reservation (reservation of resources in a future time), application licenses. Below, we briefly explain each resource job management system's features.

PBS Pro [39] (and its unsupported open source version TORQUE [41]) is a commercial scheduler. It supports GPU scheduling in two approaches: (i) the basic approach where only one GPU job can be run on any given node at a time and (ii) advanced approach where the job can request each GPU on any node explicity. The second approach is applicable when a single node is shared among multiple jobs. TSUBAME 2.0 [19] uses PBSPro and its advanced GPU scheduling property. PBS Pro also extends accounting by keeping GPU usage data by the users.

Moab [40] (and its open source version Maui) is also a commercial job scheduler, but it needs to be coupled with a job/resource manager system such as Torque, Slurm, LSF etc. Moab only handles the scheduling of the jobs in the queue. Moab has GPU support, and automatically selects the coldest GPU by tracking their temperatures.

SGE [43] is a commercial job resource management system from Oracle. Open Grid Scheduler/Grid Engine [44] is the open source version based on SGE. Both SGE and OGS have simple GPU scheduling support. In SGE, queues are defined to manage priority policies and job execution on the hosts. SGE also supports advanced reservations and application licenses.

Slurm [37] is an open source job resource management system, distributed under GPL. The main design goals behind Slurm were portability and scalability. The plug-in mechanism allows the developers to extend Slurm functionality to different systems or purposes. It is designed as a light-weight system. There is a central controller daemon, *Slurmctld* on the central management node. This controller daemon is responsible for reading the configuration file, monitoring and managing the nodes, grouping nodes into partitions, accept job submission/cancellation requests from the users, and assign resources to the jobs in the pending queue. There are also separate daemons named, *Slurmd*, at each compute node. *Slurmd* communicates with *Slurmctld* and passes information about the node's status, the resources, and the job currently running on that node [37]. The user commands can run anywhere in the cluster but interact with the controller daemon. Some of the Slurm commands are *scontrol*; for administration purposes by root user, *sinfo*; for reporting Slurm partition and node status, *squeue*; for reporting Slurm jobs with their properties such as job names, user names, *scancel*; for cancelling previously submitted Slurm jobs, *sacct*; for displaying Slurm job accounting data and *srun*; for running jobs accessing resources managed by Slurm.

Figure 2.1 depicts some of the Slurm commands and their interactions with the daemons. Slurm separates the cluster into partitions, which are basically job queues with different constraints. The jobs in a partition are allocated nodes within that partition. In Slurm, there are different plug-ins for each task. Some of the widely used plug-ins and their definitions are given in Table 2.1. All of these plug-ins run at the central daemon *Slurmctld*.

2.1. Scheduling Algorithms

Given a finite set A of tasks, a length $l(a) \in \mathbb{Z}^+$ for each $a \in A$, a number $m \in \mathbb{Z}^+$ of processors, and a deadline $D \in \mathbb{Z}^+$, the problem of finding a partition $A =$

Table 2.1. Slurm plug-ins and their tasks.

Plug-in Type	Plug-in Name	Task
scheduling	basic	FCFS schedulling
	backfill	Backilling
Resource Selection	linear	Treat each node as one resource
	cons_res	Treat each consumable resource (core or memory) separately
Priority Calculation	basic	decremental counter, first job in the queue has the highest priority
	multifactor	Different factors are used to calculate each job's priority such as the job's age, fair-share usage, size
Generic resources	gpu	Handle GPUs as generic resources on each node

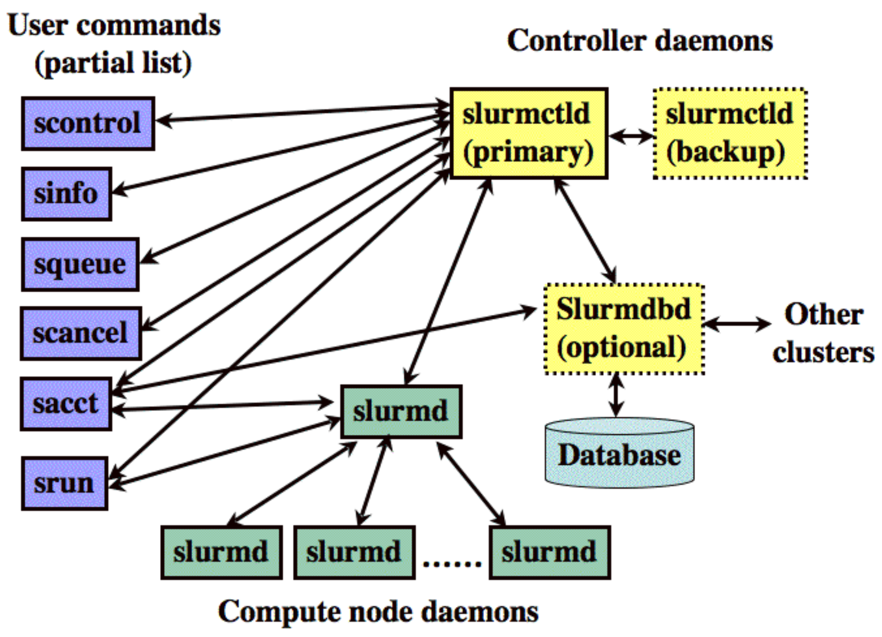


Figure 2.1. Slurm components [45].

$A_1 \cup A_2 \cup \cdots \cup A_m$ of A into m disjoint sets such that $max\{\sum_{a \in A_i} l(a)\colon 1 \leq i \leq m\} \leq D$ is proven to be NP-complete by restriction to Partition Problem, by choosing $m = 2$ and $D = \frac{1}{2} \sum_{a \in A_i} l(a)$ [46]. Other, harder NP-complete scheduling problems are shown in Table 2.2. Therefore heuristics algorithms are used at the supercomputer centers to find a good scheduling solution.

The first, and the most basic scheduling algorithm is FCFS. The first job in the queue to arrive is selected, and run to completion. However, in the current schedulers, the queue is ordered by the priorities of the jobs, which is not only dependent on the job arrival time, but also other features such as job size and fair-share policies as described above.

The most commonly used algorithm is Argonne Extensible Scheduler System (EASY) Backfilling [47]. In the EASY backfilling algorithm, if the first job in the queue has to wait due to the job's requirements, some of the resources may stay idle, even though there are enough resources to accommodate requirements of other jobs in the queue. The EASY backfiling algorithm calculates the expected ending time of each job running in the system, and calculates the earliest future-time to start for the jobs currently waiting in the queue. The scheduler than scans the queue for smaller jobs,that could currently run, and end before the earliest future-time calculated above. This method was introduced by the Argonne lab in [48]. In this method, the users to supply an estimate for their job's runtimes.

The number of reservations (the number of jobs that the earliest future-time is calculated for), the lookahead (the number of jobs to scan in the queue in order to backfill) are some of the parameters for the EASY backfilling method [49].

An example comparing the FCFS and the backfilling method has been given in Figure 2.2. Assume four jobs are submitted in the order of J_1, J_2, J_3 and J_4 with the CPU requests as $4, 4, 8, 4$ and priori known execution times as $1, 2, 1, 1$. In first-come first-served, the scheduler would start J_1 and J_2 at $t = 0$, wait until J_3 can be started, and start J_3 at $t = 2$ and J_4 at $t = 3$. However, when backfilling is employed, the scheduler would know that J_2 would end at $t = 2$, so the earliest future-time of J_3 is at $t = 3$, and since there is availablew resources and enough time for J_4 to execute, it could be backfilled into the system.

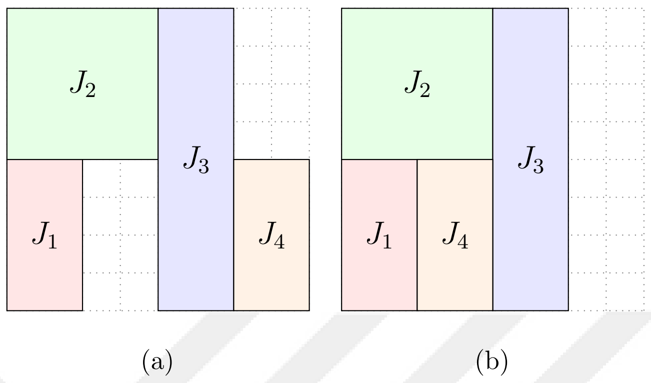


Figure 2.2. Scheduling for an example of four jobs using *(a)* FCFS and *(b)* backfilling. X-axis gives the time and y-axis gives the processors.

Another concept widely used in supercomputers is preemption. Preemption is the interruption of a preemptee job by a preemptor job in the queue which has higher priority. Preemption can be used to optimize FCFS or backfilling techniques by defining a priority gap between the preemptee and preemptor jobs. In this thesis, we have not focused on preemption.

2.1.1. Job Scheduling in Heterogeneous Supercomputers

Heterogeneous supercomputers are systems that utilize multiple processor types such as CPUs, GPUs, or co-processors such as Xeon Phi. For heterogeneous supercomputers, like homogeneous ones, the optimal scheduling problem is an NP-complete problem [50].

In practice, the schedulers of many resource managers deal with heterogeneity problem by defining different *partitions* in the same cluster. Some resource managers even supply a hierarchical view of resources, which allows a better view of the cluster topology to the users [23]. There are also task placement approaches like *cgroups* and Non-Uniform Memory Access (NUMA) which allow hierarchical views inside the nodes, in CPU level [51, 52]. However, these approaches require the users to be more aware of the system properties and their jobs characteristics.

Many heuristics have been proposed to solve this optimal scheduling problem in heterogeneous clusters. In Min-Min scheduling algorithm, tasks are scheduled to different machines based on their expected completion times. The expected completion table is updated after each job is scheduled. In [53], the authors present an energy-aware task scheduling algorithm based on the Min-Min scheduling algorithm named EAMM. Their proposed algorithm performs better than the original Min-Min scheduling algorithm under a simulated heterogeneous cluster.

Another study considers a job as a directed acyclic graph which schedules parallel tasks, which in turn assigns the job to a certain processor or a set of processors [54]. They have implemented two scheduling algorithms, and tested their scheduling algorithms on a simulation for a 10-processor homogeneous machine and a 20-processor heterogeneous machine. Both of their algorithms performed better than the common method which fix the number of processors per job.

In order to make better use of heterogeneity, Iserte et al. [55] introduces a new type of device called *rgpu* which allows the users to use GPUs from any node in the system, unlike the current Slurm implementation where the job can only use the GPUs on the nodes allocated to it. Iserte et al. supports remote GPU virtualization in Slurm by viewing all GPUs as being in a global pool.

The concept of moldability is also important for heterogeneous systems. Moldable jobs are jobs that have resource requirements fixed at submittal time, instead of static jobs which have resource requirements are fixed at development. Moldability of a job helps increasing the resource utilization of the system [56]. Demand for moldability features of schedulers are starting to appear in recent works. For example, OmpSs, which is a new parallel programming model built on OpenMP standard, aims to achieve asynchronous parallelism and heterogeneity [57]. The system can choose at run-time any of different specialized versions of tasks to achieve higher performance by making use of the flexibility of the model [58]. Qasem delivers another framework that automatically tunes the applications to different architectures to gain high performance [59].

2.1.2. Topology Awareness

Computation performance is related to the the number of switches a job's tasks has to go through to communicate with other tasks. Finding the right set of resources for a job reduces the number of switches required for a job's different tasks to communicate, and increases the system performance.

Topology aware mapping problem is an NP-hard problem [60, 61]. Many metaheuristics have been developed to solve this problem, including simulated annealing [62] and genetic algorithms [63]. Bokhari [60] has developed a method which uses pairwise exchanges between nodes. However, none of these techniques are used in real machines, due to their computational cost and the fact that they are generally developed for topologies those are not currently used.

Bhatele [64], surveys effects of topology aware placement of jobs in his PhD thesis. He describes a metric named *hop-bytes*, which is calculated by summing up the message size multiplied by number of hops between source and destination, for all messages. He proposes that hop-bytes is a better metric to evaluate the performance than the previously used maximum dilation, and proposes various algorithms for three dimensional networks to automatically place the jobs on a network so as to minimize the hop-bytes. He develops application specific topology-aware job placement techniques for widely used softwares such as NAMD and OpenAtom.

Pascual et al. [65], carries out a simulation based study in order to test whether the system performance can be increased by assigning resources to jobs in a more topologically aware way. They introduce a speed-up factor, suggesting that a job would perform better due to the topologically aware resource assignment, which would lead to a communication locality [65]. In order to improve communication performance, a contiguous network partition is assigned to each job in the work by [65]. Contiguous network partition, although increases the communication performance, results in a scheduling inefficiency due to system fragmentation. However, quasi-contiguous allocation, which is a relaxed version of contiguous allocation reduces this affect and therefore improve the overall system performance.

cpuset is a Linux mechanism to assign set of CPUs and memory to a set of tasks. In a recent study, cpusets were used to allow a topologically aware assignment of cores within a node to the jobs, which was also integrated into IBM Platform Load Sharing Facility [66]. Their implementation allows allocation of cpusets to jobs in a first-fit algorithm of resources into jobs, by also taking into consideration the memory constraints of the jobs.

In a similar study, a new topologically aware scheduling algorithm has been developed for Slurm, which considers the InfiniBand topology [67]. The proposed relaxed topology-aware scheduling algorithm allows the users to choose a process distribution scheme. The algorithm discovers the topology, makes a topology-aware mapping for the compute nodes and find the best possible set of nodes that is available in the system by considering the available bandwidth in the switches. Their implementation increases the performance of a system of 64 nodes by up to 8%, and can decrease the network latency by up to 40% [67].

The topologies of the supercomputers in the Top500 list are also listed. The most commonly used are the fat-tree, 3-D torus and mesh topologies. Many of the supercomputers in Table 1.1, including Tianhe-2 (which is ranked first), Stampede and TSUBAME use fat-tree topologies [14, 19, 29]. Supercomputers employing InfiniBand network [20] most commonly use a fat-tree topology. More detailed information on fat-tree topology has been given in Subsubsection 2.1.2.1 and in Figure 2.3. The fourth model proposed in this thesis includes topology aware scheduling for hierarchically interconnected supercomputers.

In meshes (and in tori), the compute nodes are connected directly to one another instead of going through a switch. In n-dimensional meshes, compute nodes are connected to their neighbours in n dimensions. The most common case in mesh and torus topologies is 3D mesh and torus. A torus, is essentially a mesh, with the only difference being the nodes at the end of each dimension are connected to each other. Titan [25] employs 3D torus topology as interconnect. Mira [27] and Sequoia [26] employ 5D and K-computer [18] employs a 6-D torus topology.

Besides the fat-tree and 3D torus topologies, another popular topology is dragonfly topology. In this topology, nodes are grouped together. Within each group, there is an intra-group connection network and groups are linked together using inter-group connection network. The Piz Daint supercomputer is connected using the dragonfly topology [28].

2.1.2.1. Fat-Tree Topology. Fat-tree topology has been presented as a network topology in 1985 by Leiserson [68]. This type of network defines the compute nodes as the leaves, and the switches as the intermediate nodes of a tree. From the root to the leaves, the bandwidth on the links decreases. When building a fat-tree network, many switches are redundantly used to allow reliability. A three-level fat-tree network, with 16 nodes and 4 intermediate level switches is given in Figure 2.3. It should be noted that the edges on the higher levels are bolder, which shows that the bandwidth of these links are higher.

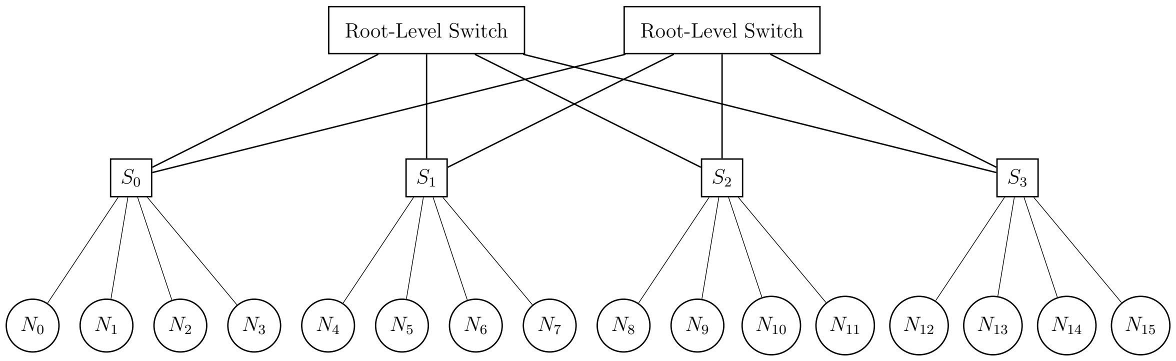


Figure 2.3. An example fat-tree with 16 compute nodes, 4 intermediate level and 2 root level switches. N_n stands for the n^{th} node and S_m stands for the m^{th} switch.

2.1.3. Energy Awareness

In this subsection, we present some of the energy awareness techniques that are being studied in the recent years. In this thesis, we do not directly deal with energy awareness. However, we are interested in obtaining high CPU utilization and accelerator utilization values for the supercomputers, which would in turn lead to more energy efficient systems.

The petascale supercomputers in the Top500 list [69] have a power consumption of ranging from 1.5 MW to 18MW. Piz Daint, which is ranked 7 in the latest Top 500 list (see Table 1.1) has a total power consumption of 1.75 MW and a computational capacity per power of 3,186 GFLOPS/KW. TSUBAME 2.0, which is ranked 15 has a a power consumption of 927 KW and a computational capacity per power of 2,951 GFLOPS/KW [35]. Most other top supercomputers in the Top500 list have relatively lower computational capacity per power consumed, and this value goes down to 1,901 GFLOPS/KW for Tianhe-2, which is on the top of that list [69].

Since the aim is to reach exascale rate in the next decade. If the power consumption per computational capacity stays the same, an exascale supercomputer will require a power plant which needs to produce 530 MWH energy per hour. This high power cost also means that there should be a dedicated power plant next to each supercomputer to cover this supercomputer's power requirement. A study in 2005 shows that, 1.2% of total energy in the United States is consumed by the servers and their peripheral units, and this percentage is doubled every 5 years [70]. Therefore, new techniques are studied to make the schedulers of these new supercomputers more energy aware [71]. Some of these techniques are presented here.

One approach is to enable Dynamic Voltage and Frequency Scaling (DVFS)-enabled scheduling [71]. The aim is to minimize the power consumption by minimizing the processor supply voltage, by scaling down processor frequency. DVFS-enabled scheduling is studied in [72] by considering the problem as a directed acyclic graph. Since the model is an NP-complete problem, approximation algorithms are suggested [72] for its solution.

Etinski et al. [73] suggests policy named *MaxJobPerf*, which aims to maximize the job performance under a power limit. Their method uses an integer programming problem to decide which jobs to run from the queue and their CPU frequencies.

2.2. Complexity of Scheduling

Table 2.2 shows a list of scheduling problems and their complexities. Distributed computing on a two-processor computer problem can be solved by reducing the problem to a minimum cut problem, where the source and the sink are the two processors, and there is a node for every module of the program [74]. Multiprocessor scheduling with a set deadline problem can be proved to be NP-complete by transformation from partition problem [46]. Other scheduling problems are even harder, and the heterogeneity level increases as we go down in Table 2.2.

2.2.1. Integer Programming

A linear programming (LP) problem is defined as an optimization problem for which the goal is to maximize or minimize a linear objective function of the decision variables. The values of the decision variables must satisfy some linear equality or inequality constraints.

An integer programming problem is a linear programming problem in which the variables are integers. If all of the variables are integers, the problem is called a pure integer programming problem [75]. Traveling salesman problem [76], vertex cover problem [46] and knapsack problems are examples that can be expressed as integer programming problems.

There are many algorithms for solving linear programming problems. The most famous is Simplex Algorithm [77]. In Simplex Algorithm, linear problems are converted

Table 2.2. Different scheduling problems and their complexities and heterogeneity levels.

Problem	**Complexity**	**Resources**	**Heterogenity Level**
Distributed computing on a 2-processor computer [74]	P	2 processors	Homogeneous
Multiprocessor scheduling with deadline [46]	NP-complete	m processors	
Scheduling with multiple nodes	NP-hard	N nodes, m processors	$\updownarrow$
Scheduling with multiple nodes and generic resources	NP-hard	N nodes, m processors, G generic resources	
Topology aware scheduling with multiple nodes and generic resources	NP-hard	N nodes, m processors, G generic resources, s switches	Heterogeneous

into the following standard form:

$$\text{maximize } cx \tag{2.1}$$

subject to

$$Ax \leq b \tag{2.2}$$

$$x \geq 0 \tag{2.3}$$

A basic feasible solution is obtained from the standard form. If the basic feasible solution is not optimal, successive pivot operations are performed to improve this feasible solution at each step.

2.2.1.1. Knapsack Problem. In our models, we will be using modified versions of knapsack problems. A **knapsack problem** is an NP-hard combinatorial optimization problem, where the objective is to find the total value of a set of items possible, given a maximum total weight W, and each item i's weight and value denoted as v_i and w_i, respectively. The decision variable x_i represents the number of copies from item i is taken into the knapsack and there are total of N items. Then the knapsack problem is defined as follows:

$$\text{maximize} \sum_{i=1}^{N} v_i \cdot x_i \tag{2.4}$$

subject to

$$\sum_{i=1}^{N} w_i \cdot x_i \leq W \tag{2.5}$$

$$x_i \in \mathbb{Z}^+ \quad \forall i \quad 1 \leq i \leq N \tag{2.6}$$

Knapsack problem has many variants. The most notable is the **0-1 knapsack**

problem, in which case an item is either selected, or not; the decision variables x_i are binary.

$$\text{maximize} \sum_{i=1}^{N} v_i \cdot x_i \tag{2.7}$$

subject to

$$\sum_{i=1}^{N} w_i \cdot x_i \leq W \tag{2.8}$$

$$x_i \in \{0,1\} \tag{2.9}$$

In another variant of the knapsack problem, ***m*-dimensional knapsack problem**, each item i has a size in m dimensions $(w_{i,1}, w_{i,2}, ..., w_{i,m})$ and the knapsack has a capacity of m-dimensions $(W_1, W_2, ..., W_m)$ [78].

$$\text{maximize} \sum_{i=1}^{N} v_i \cdot x_i \tag{2.10}$$

subject to

$$\sum_{i=1}^{N} w_{i,j} \cdot x_i \leq W_j \quad \forall j \quad 1 \leq j \leq M \tag{2.11}$$

$$x_i \in \mathbb{Z}^+ \quad \forall i \quad 1 \leq i \leq N \tag{2.12}$$

In **multiple knapsack problem**, which is similar to bin packing problem, there are multiple (K) knapsacks with different capacities $(W_1, W_2, ..., W_K)$ and the aim is to maximize the selected set of items in all knapsacks. In this problem, the decision variable $x_{i,j}$ indicates the number of copies of item i is placed in knapsack j.

$$\text{maximize} \sum_{i=1}^{N} \sum_{j=1}^{K} v_i \cdot x_{i,j} \tag{2.13}$$

subject to

$$\sum_{i=1}^{N} w_i \cdot x_{i,j} \leq W_j \quad \forall j \quad 1 \leq j \leq K \tag{2.14}$$

$$x_{i,j} \in \mathbb{Z}^+ \quad \forall i \quad 1 \leq i \leq N \; \forall j \quad 1 \leq j \leq K \tag{2.15}$$

In the following chapters, we show that our integer programming models for solving the scheduling problem for a heterogeneous system, is a combination of multiple knapsack problem and the m-dimensional knapsack problem.

2.2.1.2. Exact and Heuristic Solutions. Branch-and-bound method for solving integer programming problems relies on the following observation: If a solution to a linear programming problem relaxation of an integer programming problem has all variables as integers, the optimal solution to the former is also the optimal solution to the latter [79]. The algorithm creates a space search tree, by exploring the branches of this tree. The branches are checked against estimated upper and lower bounds on the optimal solution. These branches are excluded if it cannot produce better solution than the previously known solutions.

LP-solvers try to find the optimal solution for a given problem using various algorithms. CPLEX, lp_solve, SCIP, SoPlex, Gurobi are some of the popular LP-solvers. Some of them are commercial and some of them are open-source [80].

lp_solve is the most widely used open-source linear programming solver. Its solution is based on branch and bound method and a modified version of the simplex method. It can also solve pure linear, mixed integer and binary problems. lp_solve offers a direct interface for writing and solving the LP problems, or it can be used externally via a programming interface.

The most widely known commercial LP solver is IBM ILOG CPLEX Optimization Studio, or shortly CPLEX. CPLEX uses primal and dual simplex algorithms, sifting algorithms and logarithmic barrier algorithms [81]. CPLEX can be used from its direct interface, or CPLEX calls can be made externally via its Application Programming Interface (API).

In a recent survey [82], it has been shown that cumulative speed-up in CPLEX in the last two decades is over 30,000 fold. The speed up from version 10 to 11 is around 8 [83] on the same set of hardware. This is due to both a better usage of the current hardware and better algorithms. With the usage of multiple threads, it is now possible to solve integer programming problems with around 100,000 variables in a matter of seconds. This has allowed us to model the scheduling problem as an integer programming problem, solve it using CPLEX, and repeat this mechanism in every few seconds.

Additionally, heuristic methods have been developed to search for solutions of integer programming problems. Some of these include tabu search, simulated annealing, or ant colony optimization. These meta-heuristic methods can be applied to various integer programming problems.

2.2.2. Integer Programming in Scheduling and Auction Mechanisms

Integer and linear programming solvers have been made use of in proposed schedulers before. Erbas et al. [84] proposed the use of linear programming solvers for job scheduling in grids. Shen et al. [85] solves an integer programming problem for resource provisioning and job allocation policies. Blanco et al. [86] proposes the solution of a mixed integer problem that aims to minimize the overall makespan.

In [87], a *reservation based scheduling* approach is proposed for big-data clusters. In this approach, they use a mixed-integer-linear-programming formulation to solve the scheduling problem. This work is tested on a 256-node system running Hadoop, using real and synthetic workloads. Their results show that they can improve latency for almost 40% of the jobs. In [88], authors propose a method composed of graph algorithms and linear programming to produce optimal mapping of tasks to a heterogeneous architecture. They show that using their method they can reduce the total system cost (defined as the execution cost of tasks and the communication cost between nodes) by 13% on a simulated system.

Dynamic pricing for resources in cloud computing has been investigated several times [89–91]. It has been shown that maximum profit scheduling for a cloud can be formulated as a multiple knapsack problem [89]. This model uses the demand for the resources to calculate a price for the resource at the cloud. If the price a user is willing to pay is higher than that calculated price, the resource is allocated. This has also been shown for infinite horizon case, where the optimal price is not time-dependent but varies with utilization [90]. In [91], it is suggested that the acceptance of a request would depend on the bid price and the system utilization in a cloud. G-commerce [92] defines the resource producers as *sellers* and resource consumers as *buyers* in a grid environment. It uses auction mechanism and market economy to come up with a market equilibrium at a certain price. In [93] a tool for the grid users has been designed to come up with a necessary budget to calculate the cost of completing a certain project at a given deadline.

[94] defines a model and heuristic algorithms for multi-unit nondiscriminatory combinatorial auctions. The model finds the most profitable set of bids by designing the auction problem as an integer programming problem. It has been shown that bartering models can be used on grids to trade multiple resources using directed hypergraphs [95]. These models can enable resource owners to automatically get something in return from job owners. In [96], a model has been proposed to use combinatorial auctions to handle resource scheduling in supercomputing grids. Double auction market for grid resource allocation has also been used in [97], where the price is flexible in a grid. The environment is thought of as a two-sided market; both the resource demanders and the suppliers are consumers and providers of the market. They propose a method named Stable Continuous Double Action (SCDA) method which reduces unneccessary price volatility and is efficient in both economic and scheduling points of view. We are not aware of any method that uses integer programming solvers to schedule jobs to resources for heterogeneous clusters.

3. SIMPLE INTEGER PROGRAMMING BASED SCHEDULER

Our first model, Simple Integer Programming Based Scheduler, abbreviated as IPSCHED [98], is based on a knapsack-like formulation, which is NP hard. Figure 3.1 depicts our window based scheduling approach. At each scheduling step, we pick a window of jobs from the front of the priority ordered job queue and the available resource information (cores and GPUs or other generic resources) from the nodes. We form and solve an integer programming problem. The problem aims to maximize the sum of selected jobs' priorities, while reducing the number of nodes over which a selected job's allocated resources are packed. This is an assignment problem and its solution tells us how to co-allocate multiple cores and generic resources on the nodes. This process is repeated periodically.

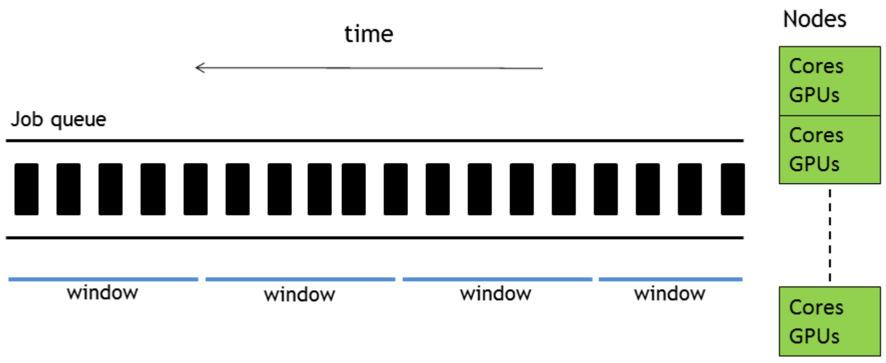


Figure 3.1. Scheduling of windows of jobs.

3.1. Formulation

To formulate the assignment problem as an IP, we first make the definitions in Table 3.1. The IP formulation of the IPSCHED model is then given as follows:

$$max \sum_{j \in J} P_j \left(s_j - c_j\right) \tag{3.1}$$

Table 3.1. List of main symbols, their meanings, and definitions for IPSCHED.

Symbol	**Meaning**
J	Set of jobs that are in the window: $J = \{j_1, \ldots, j_{\lvert J\rvert}\}$
P_j	Priority of job j
N	Set of nodes : $N = \{n_1, \ldots, n_{\lvert N\rvert}\}$
A_n^{cpu}	Number of available CPU cores on node n
A_n^{gres}	Number of available generic resources on node n
R_j^{cpu}	Number of cores requested by job j
R_j^{gres}	Number of generic resources per node requested by job j
$R_{j,min}^{node}$	Minimum number of nodes requested by job j
$R_{j,max}^{node}$	Maximum number of nodes requested by job j
s_j	Binary variable indicating whether job j is allocated or not,
c_j	node packing variable for job j. This variable is set as the ratio of the number of nodes allocated to a job to the total number of nodes in the system.
x_{nj}	no. of cores allocated to job j at node n
t_{nj}	binary variable showing whether job j is allocated any resource on node n

$$\sum_{j \in J} x_{nj} \leq A_n^{cpu} \ \forall n \in N \tag{3.2}$$

$$\sum_{n \in N} x_{nj} = R_j^{cpu} s_j \ \forall j \in J \tag{3.3}$$

$$\sum_{j \in J} R_j^{gres} t_{nj} \leq A_n^{gres} \ \forall n \in N \tag{3.4}$$

$$c_j = \frac{\sum_{n \in N} t_{nj}}{2|N|} \ \forall j \in J \tag{3.5}$$

$$R^{node}_{j,min} \leq 2|N|c_j \leq R^{node}_{j,max} \ \forall j \in J \tag{3.6}$$

$$t_{nj} = \begin{cases} 1, & x_{nj} > 0 \\ 0, & x_{nj} = 0 \end{cases} \ \forall n \in N, j \in J \tag{3.7}$$

Constraint 3.2 limits the total number of processes that can be assigned to any node n by the number of available CPUs on that. Constraint 3.3 sets the total number of CPUs allocated for a job to either 0 (if the job is not selected), or to R^{cpu}_j (if the job is selected) for each node n and each job j. Constraint 3.4 limits the total number of generic resources that can be assigned to any node n by the number of available generic resources on that node. It also ensures that if a job is allocated some CPUs on any given node, it should also be allocated R^{gres}_j generic resources on that node. In constraint 3.5 packing factor, c_j, of job j is calculated by considering the number of nodes on which a job is allocated resources. The total number of nodes allocated to a job is divided by $2|N|$ so that the value of c_j is in the interval $[0, \frac{1}{2}]$, and hence the value of the objective function given in 3.1 is always positive.

The number of variables and constraints of the IP formulation are given in Tables 3.2 and 3.3 in terms of number of jobs in the window ($|J|$) and the number of nodes in the system ($|N|$). The total number of variables is $|J| \cdot |N|$, which is quite large. This in turn necessitates putting of a limit on the number of jobs in the window (defined as *MAX_JOB_COUNT*). In this model, only three types of job submission requests are available:

(i) total number of cores

(ii) total number of nodes

(iii) certain number of generic resources per node

Table 8.2 lists a comparison of available job submission requests in different models.

Table 3.2. Number of variables in IPSCHED IP formulation.

Variable name	**Number of variables**
s_j	$\|J\|$
c_j	$\|J\|$
x_{ij}	$\|J\| \cdot \|N\|$
t_{ij}	$\|J\| \cdot \|N\|$
Total	$2\|J\| \cdot (1 + \|N\|)$

Table 3.3. Number of constraints in IPSCHED IP formulation.

Constraint	**Number of constraints**
3.2	$\|N\|$
3.3	$\|J\|$
3.4	$\|N\|$
3.5	$\|J\|$
3.6	$2\|N\|$
3.7	$2\|J\|\|N\|$
Total	$2 \cdot (\|J\| + 2\|N\| + \|J\|\|N\|)$

The algorithm of IPSCHED is given in Figure 3.2. In IPSCHED, we first define a job window size named *MAX_JOB_COUNT*. The priorities of each job are set by the job resource management system. For each job in the window, we retrieve the priority, CPU and generic resource request of that job from the central controller. Also, for each node, we retrieve the number of available CPUs and generic resources. This information, i.e. (P_j, R_j^{cpu}, R_j^{gres}, A_n^{cpu}, A_n^{gres}), gives us all the parameters necessary to form our integer programming problem. We use CPLEX to solve the integer programming problem and the job selection. Thus, the resource selection problems are solved simultaneously.

```
1  Generate priority ordered job window of size up to $MAX\_JOB\_COUNT$
2  From each job $j$ in the window, collect the following into $job\_list$ array
3      a. priority $P_j$
4      b. CPU request $R_j^{cpu}$
5      c. Gres request $R_j^{gres}$
6  From each node $n$ in the system, collect the following into $node\_info$ array
7      a. number of available CPUs $A_n^{cpu}$
8      b. number of available gres $A_n^{gres}$
9  Form the IP problem
10 Solve the IP problem and get $s_j$, $t_{nj}$ and $x_{nj}$ values
11 For jobs with $s_j = 1$, set job's process layout matrix and start the job by
12     a. For each node $n$, assign processors on that node according to $x_{nj}$
13     b. Start the job, no more node selection algorithm is necessary.
```

Figure 3.2. IPSCHED scheduling steps.

Each job having $s_j = 1$ are started by the scheduler on the nodes where $t_{nj} = 1$. Additionally, the number of cores assigned to any job j on node n is given by x_{nj}.

In this model, the topology of the system is not considered. In the integer programming formulation, when resources are assigned to jobs, it may be the case that the resources assigned to a job are not consecutive. This is due to the fact that every t_{nj} is an independent variable, and there is no constraint limiting those t_{nj} variables to be consecutive. Figure 3.3 shows an example assignment of 5 jobs to a 16 node system using IPSCHED scheduler.

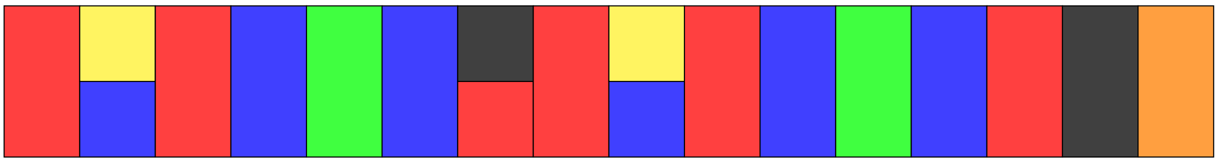


Figure 3.3. An example assignment of jobs to 16 nodes by IPSCHED. Jobs are designated in different colors (red, green, blue, orange, yellow, black).

3.2. IPSCHED Slurm Implementation

Since scheduling mechanism is designed as a plug-in for Slurm, we have designed our own plug-in, which is named as *ipsched*. This plug-in works on the controller daemon and is similar to the backfill plug-in already included in Slurm. As described in Figure 3.2, this *ipsched* plug-in responsible for gathering available resource information and job requirement information, forming and solving the IP problem. As a result of this IP problem, we know which jobs to be scheduled, and the resources that are going to be assigned to these jobs.

In Slurm, 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 [99]. 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 when an assignment problem is solved.

In Slurm, generic resources 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* plug-in runs every *SCHED_INTERVAL* seconds. By default this value is 3 and can be changed in the Slurm configuration file.

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 cannot make 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 was the latest stable version during the time of development. *ipsched* is publicly available at `https://github.com/aucsched/ipsched`.

4. AUCTION BASED SCHEDULER WITH LINEAR TOPOLOGY SUPPORT

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 run-time 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. As it will be shown in Chapter 8, the allocation process does not consider topology. Therefore, the allocation of jobs to nodes are spread across nodes randomly. This would, in turn, decrease the communication performance of a job. Since a scheduler has access to information about available resources and is the authority that makes the allocation decisions, it can enumerate and consider alternative candidate resource allocations to each job. This model considers this complementary approach that aims to tune mappings of jobs at the scheduling level. Auction Based Scheduler with Linear Topology Support, abbreviated AUCSCHED1, attempts to allocate contiguous blocks on one-dimensional array of nodes [100].

Our proposed methodology is similar to that of the IPSCHED discussed earlier. 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.

Given the list of symbols and their meanings in Table 4.1, the IP formulation of AUCSCHED1 model is as follows:

Table 4.1. List of symbols, their meanings, and definitions for AUCSCHED1.

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 $\{c \in C \mid c \in B_j \ \ and \ \ n \in N_c\}$
R_{jc}^{cpu}	Number of cores requested by job j in bid c
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 4.9).
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).

$$Maximize \quad \sum_{j \in J} \sum_{c \in B_j} (P_j + \alpha \cdot F_{jc}) \cdot b_{jc} \tag{4.1}$$

subject to constraints :

$$\sum_{c \in B_j} b_{jc} \leq 1 \; for \; each \; j \in J \tag{4.2}$$

$$\sum_{n \in N_c} u_{jn} = b_{jc} \cdot R^{node}_{jc}$$
$$for \; each \; (j,c) \in J \cdot C \; s.t. \; c \in B_j \tag{4.3}$$

$$\sum_{n \in N_c} \sum_{c \in B_j} u_{jn} + r_{jn} = \sum_{c \in B_j} b_{jc} \cdot R^{cpu}_{jc} \; for \; each \; j \in J \tag{4.4}$$

$$\sum_{j \in J} u_{jn} + r_{jn} \leq A^{cpu}_{n} \; for \; each \; n \in N \tag{4.5}$$

$$\sum_{j \in J} u_{jn} \cdot R^{gres}_{j} \leq A^{gres}_{n} \; for \; each \; n \in N \tag{4.6}$$

$$0 \leq r_{jn} \leq u_{jn} \cdot min(A^{cpu}_{n} - 1, R^{cpu}_{j,max} - 1)$$
$$for \; each \; (j,n) \in J \cdot N \tag{4.7}$$

$$u_{jn} + r_{jn} = \sum_{c \in C_{jn}} b_{jc} \cdot R^{cpn}_{j}$$
$$for \; each \; (j,n) \in J \cdot N \; s.t.$$
$$R^{cpn}_{j} > 0 \; and \; C_{jn} \neq \emptyset \tag{4.8}$$

Our objective as given by equation 4.1 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:

$$P_{min} > \alpha \cdot (|B| + 1) > \sum_{j \in J} \sum_{c \in B_j} \alpha \cdot F_{jc} \tag{4.9}$$

The bid preference values, F_{jc}, are selected as explained in the Section 4.1 and are in the interval $(0, 1]$. α is chosen as follows, so that inequality 4.9 holds:

$$\alpha = \frac{P_{min}}{|B| + 1} \tag{4.10}$$

At most one bid per job can be selected in a solution set, this is ensured by Constraint 4.2. Constraint 4.3 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 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, is fixed during job submission time. For other jobs, this parameter is set during bid generation phase (see Section 4.1). Constraint 4.4 ensures that if a bid is selected, than the total number of CPU cores allocated to a job is exactly as many as requested by that bid. Constraint 4.5 and 4.6 make sure that total number of allocated cores and generic resources on any given nodes does not exceed that are available on that node, respectively. Constraint 4.7 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. 4.8, is generated for jobs for which cores per node option (R_j^{cpn}) is specified, and ensures that the number of cores allocated on each node is equal for a bid.

The variable and constraint types in AUCSCHED1 model are given in Tables 4.2

Table 4.2. Number of Variables in AUCSCHED1 model IP formulation.

Variable name	**Number of variables**
b_{jc}	$\|B\|$
u_{jn}	$\|N_c\|$
r_{jn}	$\|N_c\|$
Total	$2 \cdot \|N_c\| + \|B\|$

Table 4.3. Number of Constraints in AUCSCHED1 IP formulation.

Constraint	**Number of constraints**
4.2	$\|J\|$
4.3	$\|B\|$
4.4	$\|J\|$
4.5	$\|N\|$
4.6	$\|N\|$
4.7	none
4.8	$\|N_c\|$
Total	$2 \cdot \|N\| + 2 \cdot \|J\| + \|B\| + \|N_c\|$

and 4.3, respectively. The number of variables in AUCSCHED1 is $2 \cdot |N_c| + |B|$. It should be noted that the number of variables are in order of number of bids and number of nodes included in the bids, which is in the order of $O(|N| \cdot |B|)$. Therefore, we have set a limit on maximum number of bids created in the bid generation phase, which we refer to as *MAXBIDS*. In our implementation, we have allowed each job to generate 15 bids and we have set *MAXBIDS* as 150.

The algorithm of AUCSCHED1 is given in Figure 4.1. Similar to the algorithm given in Figure 3.2, the information regarding the job resource requests, job priority, and available resource information of each node is gathered. Afterwards, *nodesets* are

```
1  Generate priority ordered job window.
2  From each job $j$ in the window, collect the following into $job\_list$ array
3      a. priority $P_j$
4      b. CPU request $R_j^{cpu}$
5      c. Gres request $R_j^{gres}$
6  From each node $n$ in the system, collect the following into $node\_info$ array
7      a. number of available CPUs $A_n^{cpu}$
8      b. number of available gres $A_n^{gres}$
9  Generate nodesets
10 Generate bids (up to $MAX\_BIDS$), and create the IP problem
11 Solve the IP problem and get $b_{jc}$, $u_{jn}$ and $r_{jn}$ values
12 For bids with $b_{jc} = 1$, set job's process layout matrix and start the job by
13     a. For each node $n \in N_c$, assign $u_{jn}$ processors on that node.
14     b. Start the job, no more node selection algorithm is necessary.
```

Figure 4.1. Auction Based Scheduler with Linear Topology Support (AUCSCHED1).

generated, which are contiguous blocks of nodes, defined in Section 4.1. Using the job resource requests and the nodeset information, multiple bids (up to *MAX_BIDS*) are generated for each job in the window. Finally, CPLEX is used to solve an assignment problem that maximizes the objective function involving the priorities and preference value of each job.

In this model, there are four types of job submission requests available:

(i) total number of cores
(ii) total number of nodes
(iii) certain number of generic resources per node
(iv) certain number of cores per node
(v) contiguous allocation

Table 8.2 lists a comparison of available job submission requests in different models.

Figure 4.2 shows an example assignment of 5 jobs to a 16 node system using AUCSCHED1.

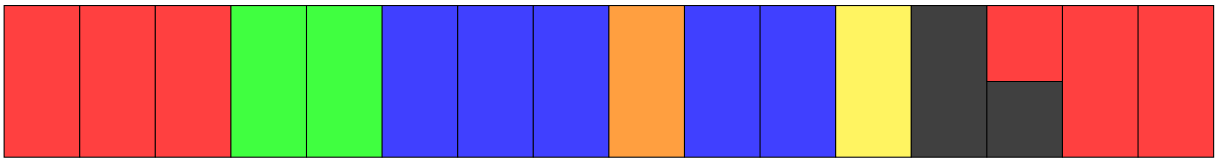


Figure 4.2. A sample assignment of jobs to 16 nodes by AUCSCHED1. The job requirements are same as in Figure 3.3.

We illustrate the bid generation and solution process of AUCSCHED1 with the aid of an example shown in Figure 4.3. Figure 4.3 shows the bid generation process and the outcome of the auction that tells us what resource allocations are to be done. AUCSCHED1 first retrieves system state and information about the jobs in the queue. Jobs' resource requirements, their priorities and information about whether they want contiguous allocations are retrieved. For the example Figure 4.3, 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 scheduler 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 $(81, 144, 512, g)$ for $g = 0, 1, 2$. After the nodesets are created, now possible bid classes are generated. The generated bids have different preference values, F_{jc}. Section 4.2 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 an an acceptable time. In our plug-in there

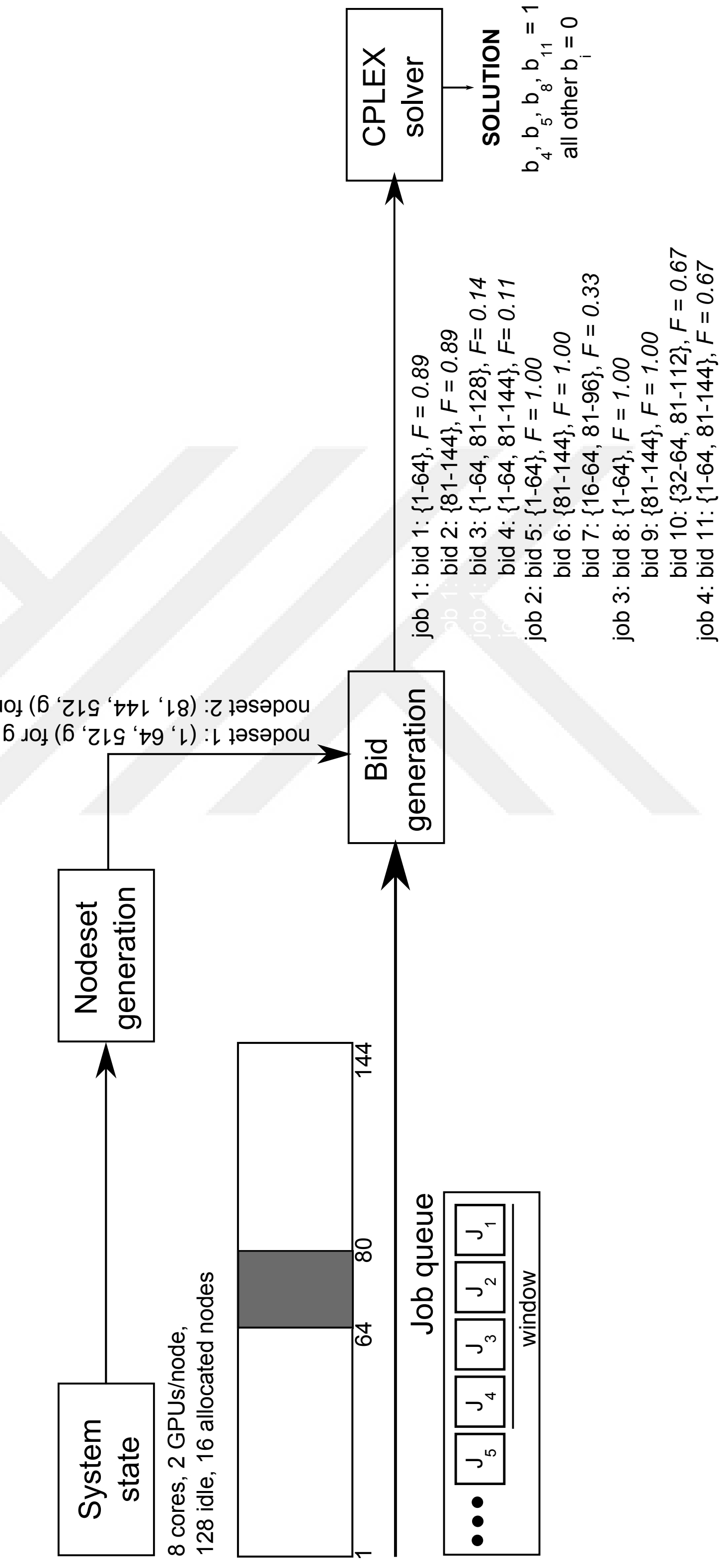

System state
8 cores, 2 GPUs/node,
128 idle, 16 allocated nodes
1
64
80
144
Nodeset generation
nodeset 1: (1, 64, 512, g) for g = 0, 1, 2
nodeset 2: (81, 144, 512, g) for g = 0, 1, 2
Job queue
J_5
J_4
J_3
J_2
J_1
window
Bid generation
job 1: bid 1: {1-64}, F = 0.89
bid 2: {81-144}, F = 0.89
bid 3: {1-64, 81-128}, F= 0.14
bid 4: {1-64, 81-144}, F= 0.11
job 2: bid 5: {1-64}, F = 1.00
bid 6: {81-144}, F = 1.00
bid 7: {16-64, 81-96}, F = 0.33
job 3: bid 8: {1-64}, F = 1.00
bid 9: {81-144}, F = 1.00
bid 10: {32-64, 81-112}, F = 0.67
job 4: bid 11: {1-64, 81-144}, F = 0.67
CPLEX solver
SOLUTION
$b_4, b_5, b_8, b_{11} = 1$
all other $b_i = 0$

Figure 4.3. Detailed nodeset generation, bid generation and solution figure for a 144 node system.

is a variable called $MAXBIDSPERJOB$ which is the limit on the number of bids generated by our system for each job. In Figure 4.3, 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 [81] 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 $\{1-64, 81-144\}$.

4.1. Bid generation phase

Before generating the bids, we generate *nodesets*. Nodesets are contiguous block of nodes, which have some total number of cores and some generic resources available per node in these nodes. It can be defined as a 4-tuple (n_i, n_j, c, g) such that the following statements hold:

$$A_n^{cpu} > 0 \ \ \forall n \in \{n_i, \ldots, n_j\}$$

$$A_n^{gres} \geq g \ \ \forall n \in \{n_i, \ldots, n_j\}$$

$$\sum_{n \in \{n_i, \ldots, n_j\}} A_n^{cpu} = c$$

Here, n_i and n_j are the first and the last nodes respectively, c is the total number of cores, and g is the number of generic resources per node in the nodeset. Figure

4.4 illustrates nodesets on a small 12 node system. Suppose that the topmost 1D array shows the numbers of available *cores/generic resources* on the system. The eight nodesets constructed are shown in the figure.

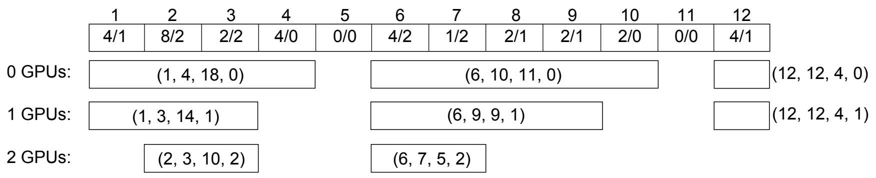


Figure 4.4. Determination of nodesets for a 12-node system.

From these nodesets, bid classes are generated according type of each job. Bid classes are sets of nodes, and bids are instantiations of bid classes. Multiple jobs can have bids of the same class, which means they are bidding for the same set of nodes. We have classified jobs with respect to its resource requests in the following manner:

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 generic resources per node (R_j^{gres}) are specified.
3. Number of nodes (R_j^{node}), cores per node (R_j^{cpn}) and number of generic resources per node (R_j^{gres}) are specified.

For job type 1, nodesets which have $g = 0$ are tested for the total number of cores available in the nodeset.

A. $c \geq R_j^{cpu}$, i.e. 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 that node set, satisfying the number of cores constraint.
B. Same mechanism as in (A) is applied, but the bid class is not aligned to the beginning or end of the nodeset.
C. The nodeset may be combined with the neighbouring nodesets to create non-contiguous nodesets, with higher number of cores, so that $c_k + c_l + c_m \geq R_j^{cpu}$, where k, l and m are nodeset indices. This type of bid class is only created if a job does not explicitly request contiguous allocations.

For job type 2, nodesets which have $g = R_j^{gres}$ are tested for the total number of cores available in the nodeset.

A. $c \geq R_j^{cpu}$ and $n_j - n_i + 1 = R_j^{node}$ i.e. the nodeset has enough cores for the job on the exact number of nodes that job requests. The bid class is created aligning to the beginning or end of that node set.
B. Same mechanism as in (A) is applied, but the bid class is not aligned to the beginning or end of the nodeset.
C. The nodeset may be combined with the neighbouring nodesets to create non-contiguous nodesets, with higher number of cores, so that $c_k + ... + c_m \geq R_j^{cpu}$, where $m - k + 1 = R_j^{node}$. This type of bid class is only created if a job does not explicitly request contiguous allocations.

Job type 3 is very similar to job type 2. There is one additional constraint for creating bid classes for jobs with type 3, and that is each node is checked if there is at least R_j^{cpn} cores.

After the bid classes are generated, AUCSCHED1 generates the bids. While generating the bids, the algorithm in Figure 4.5 is used. In total, there is at most *MAXBIDS* bids are created. For each job, at most *MAXBIDSPERJOB* are created.

4.2. Preference value calculation

The bids should have different preference values due to their contiguity ratios and allocated locations. $F_{jc_A} > F_{jc_B} > F_{jc_C}$ inequality should hold, since type A bid has a contiguous allocation, and is aligned to the nodeset, type B bid has a contiguous allocation, and type C bid has a non-contiguous allocation. We have implemented a function which results in the following preference value ranges:

$$F_{jc_C} \in (0, \frac{1}{2}), \quad F_{jc_B} \in (\frac{1}{2}, \frac{3}{4}), \quad F_{jc_A} \in (\frac{3}{4}, 1) \tag{4.11}$$

```
Generate priority ordered job window $(J)$ of size up to $MAX\_JOB\_COUNT$
Generate nodesets
$b_{total} \leftarrow 0$ ;                                  // the total number of bids
while $j = 0$ to $|J| - 1$ do
    $b_j \leftarrow 0$ ;                                    // number of bids for job $j$
    get $type$ of job $j$
    generate bid classes (A) of type $type$, increment $b_j, b_{total}$
    generate bid classes (B) of type $type$, increment $b_j, b_{total}$
    if $b_j < MAXBIDSPERJOB$ then
        generate bid classes (C) of type $type$, increment $b_j, b_{total}$
    end
    if $b_{total} \geq MAXBIDS$ then
        break
    end
end
```

Figure 4.5. Bid generation algorithm.

Let S denote the set of all nodesets and S_c denote the set of nodesets from which a bid c's nodes comes from. Since they have contiguous allocations, there is only one nodeset involved, and $|S_c| = 1$, for type A and B bids. For type C bids, on the other hand, $|S_c| > 1$. The formula used to calculate the preference value F_{jc} of a bid c is given as follows:

$$F_{jc} = 1.0 - k_1 - k_2 \cdot \frac{|N_c|}{|N| + 1} - k_3 \cdot \frac{|S_c|}{|S| + 1} \tag{4.12}$$

This function together with the constants k_1, k_2 and k_3 that are given in Table 4.4 satisfies the range constraints given in Equation 4.11. The idea behind the function is to basically disfavour allocations that are fragmented or that leads to fragmentation.

Table 4.4. Constants for F_{jc} calculation for different job and bid classes.

Job type 1			
	bid type A	**bid type B**	**bid type C**
$\mathbf{k_1}$	0	0.25	0.5
$\mathbf{k_2}$	0.25	0.25	0.25
$\mathbf{k_3}$	0	0	0.25
Job types 2 and 3			
	bid type A	**bid type B**	**bid type C**
$\mathbf{k_1}$	0	0.5	0.5
$\mathbf{k_2}$	0	0	0
$\mathbf{k_3}$	0	0	0.5

4.3. AUCSCHED1 Slurm Implementation

The implementation of this model, named *aucsched1*, is similar to the implementation of IPSCHED. Similarly, we have disabled the FIFO scheduling of Slurm, and use the same resource selection mechanism (*ipconsres*).

In addition to what is available in IPSCHED, in AUCSCHED1, the users may now use Slurm options `--ntasks-per-node` and `--contiguous`. The former allows defining R_j^{cpn}, the number of cores per node requested by job j. If this is not specified explicitly, this parameter gets a value of 0. The latter allows requesting a contiguous allocation of resources (i.e. type B and type C bids defined in Section 4.1). If a contiguous allocation is requested, non-contiguous bids which are combination of non-consecutive nodesets will not be generated.

AUCSCHED1 was developed and tested on version 2.5.4 of SLURM, which was the latest stable version during the time of development. AUCSCHED1 is publicly available at `https://github.com/aucsched/aucsched`.

5. AUCTION BASED SCHEDULER WITH MOLDABILITY SUPPORT

Auction Based Scheduler with Moldability Support (AUCSCHED2) is another scheduler model that enhances our AUCSCHED1 Auction Based Scheduler with Linear Topology Support given in Chapter 4. The two main enhancements are:

(i) *A new integer programming (IP) formulation:* The IP formulation used in AUCSCHED2 greatly reduces the number of variables and hence allows faster solution and larger number of bids to be generated. This is explained in detail in Section 5.1.

(ii) *Extension of Slurm to support generic resource ranges*: The current schedulers support specification of node ranges but not of generic resource ranges. Such a feature can be very useful to runtime auto-tuning applications and systems that can make use of variable number of generic resources (or other accelerators). This is explained in detail in Section 5.2.

5.1. Improved IP 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 Section 4.1. An allocation problem that maximizes an objective function involving priorities of jobs is then solved as an IP problem.

Given the symbols and their meanings in Table 5.1 , the new objective function and the constraints of the optimization problem are as follows:

$$Maximize \ \sum_{j \in J} \sum_{i \in B_j} (P_j + \alpha \cdot F_i) \cdot b_i \tag{5.1}$$

Table 5.1. List of symbols, their meanings, and definitions for Auction Based Scheduler with Linear Topology Support.

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 : $N = \{n_1, \ldots, n_{\|N\|}\}$
N	Set of nodes requested by bid i: $N = \{n_1, \ldots, n_{\|N\|}\}$
B	Set of all bids, $B = \{b_1, \ldots, b_{\|B\|}\}$
B_j	Set of bids for job j : $B_j = \{i_1, \ldots, i_{\|B_j\|}\}$ variables
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.
R_i^{gres}	number of generic resources per node requested by bid i.
R_j^{gres}	minimum number of generic resources per node requested by job j.
$R_j^{gres,max}$	maximum number of generic resources per node requested by job j. If not set, this is taken as equal to R_j^{gres}.
R_j^{cpu}	Number of cores requested by job j
A_n^{cpu}	number of available CPU cores on node n
A_n^{gres}	number of available generic resources on node n
R_j^{cpn}	Number of cores per node requested by job j. If not specified, this parameter gets a value of 0.
F_i	preference value of bid i ranging between 0 and 1.
α	A factor multiplying the preference value F_i so that the added preference values do not change the job priority maximizing solution (See Equation 5.6).
b_i	binary variable corresponding to bid i

subject to constraints :

$$\sum_{i \in B_j} b_i \leq 1 \; for \; each \; j \in J \tag{5.2}$$

$$\sum_{j \in J} \sum_{i \in B_j} b_i \cdot T_{in} \leq A_n^{cpu} \ for\ each\ n \in N \tag{5.3}$$

$$\sum_{j \in J} \sum_{i \in B_j} b_i \cdot U_{in} \cdot R_i^{gres} \leq A_n^{gres} \ for\ each\ n \in N \tag{5.4}$$

The objective function given by equation 5.1 is the same as the one in the previous Auction Based Scheduler with Linear Topology Support formulation (see Chapter 4). It maximizes the summation of selected bids' priorities with positive contribution $\alpha \cdot F_i$ added to the priority in order to favour bids with less fragmentation (see Section 4.1). Constraint 5.2 ensures that at most one bid is selected for each job. Constraint 5.3 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 5.4 ensures that the number of generic resources 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. Similar to the previous Auction Based Scheduler with Linear Topology Support formulation, we do not want the added positive contribution $\alpha \cdot F_{jc}$ to change the solution that maximizes the summation of priorities. When we choose α as in Equation 5.6, the inequality in Equation 5.5 holds, thus conserving the original priority order.

$$P_{min} > \alpha \cdot (|B| + 1) > \sum_{j \in J} \sum_{c \in B_j} \alpha \cdot F_{jc} \tag{5.5}$$

$$\alpha = \frac{P_{min}}{|B| + 1} \tag{5.6}$$

In the bid generation phase, bids are generated so that the number of generic resources requested by a bid is within the limits set during submission for that job, or formally $R_j^{gres} \leq R_i^{gres} \leq R_j^{gres,max}$. Also, if the job requests a certain number of cores per node (R_j^{cpn}), this is also satisfied during the bid generation phase by setting $T_{in} = R_j^{cpn}$ for all nodes $n \in N_i$.

In AUCSCHED2, in order to favour a job's bids with a higher number of generic resources over this job's other bids, the preference value calculation of F_i has also been modified and is now calculated as in Equation 5.7.

$$F_{jc} = 1.0 - k_1 - k_2 \cdot \frac{|N_c|}{|N|+1} - k_3 \cdot \frac{|S_c|}{|S|+1} + k_4 \cdot \frac{R_i^{gres}+1}{R_j^{gres,max}+1} \tag{5.7}$$

The last term in Equation 5.7, $k_4 \cdot \frac{R_i^{gres}+1}{R_j^{gres,max}+1}$, is added so that bids requesting higher number of gress obtain higher preference values. It should be noted that preference value only affects the selection between bids of the same job. The constants for the calculation for F_i in AUCSCHED2 are given in Table 5.2.

Table 5.2. Constants for F_i calculation for different bid types in AUCSCHED2.

Job type 1			
	bid type A	**bid type B**	**bid type C**
$\mathbf{k_1}$	0	0.25	0.4
$\mathbf{k_2}$	0.25	0.25	0.25
$\mathbf{k_3}$	0	0	0.25
$\mathbf{k_4}$	0.1	0.1	0.1
Job types 2 and 3			
	bid type A	**bid type B**	**bid type C**
$\mathbf{k_1}$	0	0.5	0.4
$\mathbf{k_2}$	0	0	0
$\mathbf{k_3}$	0	0	0.1
$\mathbf{k_4}$	0.1	0.1	0.1

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 find better solutions.

Figure 5.1 shows a sample assignment of 5 jobs to a 16 node system using *AUCSCHED2*, where all jobs are allocated a contiguous set of nodes.

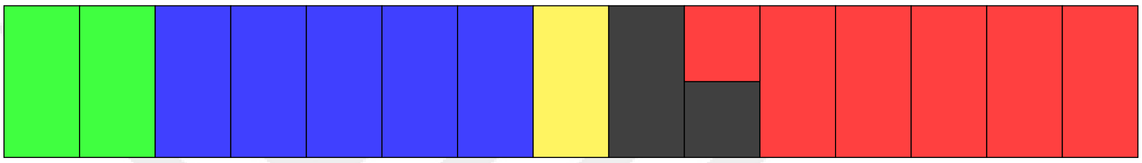


Figure 5.1. A sample assignment of jobs to 16 nodes by AUCSCHED2. The job requirements are same as in Figure 3.3, except all jobs are contiguously allocated.

5.2. aucsched2 Slurm Implementation

This model is also implemented in Slurm, and it is named *aucsched2*. *aucsched2* is implemented in a very similar fashion as *aucsched*. Hence, the implementation details regarding preference value calculation, nodeset generation, bid generation and general working logic explained above still hold for *aucsched2*. *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.

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. 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 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.

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 generic resources. 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.

The bids will be generated in a way that, $z_{min} \leq R_i^{gpu} \leq z_{max}$ will hold. *aucsched2* is publicly available at `https://github.com/aucsched/aucsched2`.

6. AUCTION BASED SCHEDULER WITH HIERARCHICAL INTERCONNECT SUPPORT

The previous versions of auction based scheduler consider the topology of a system as a 1-D array of nodes. However, when the supercomputers in the Top 500 list [22] are considered, we can see many supercomputers with tree and fat-tree interconnect, such as the Tianhe-2 [14] that ranked first in the Top 500 list in November 2015 [22] or the Tsubame 2.5 [19] that ranked thirteenth in the same list. In this section, we present a new model, Auction Based Scheduler with Hierarchical Interconnect Support, abbreviated AUCSCHED3, which considers tree and fat-tree topologies similar to those shown in Figure 6.1.

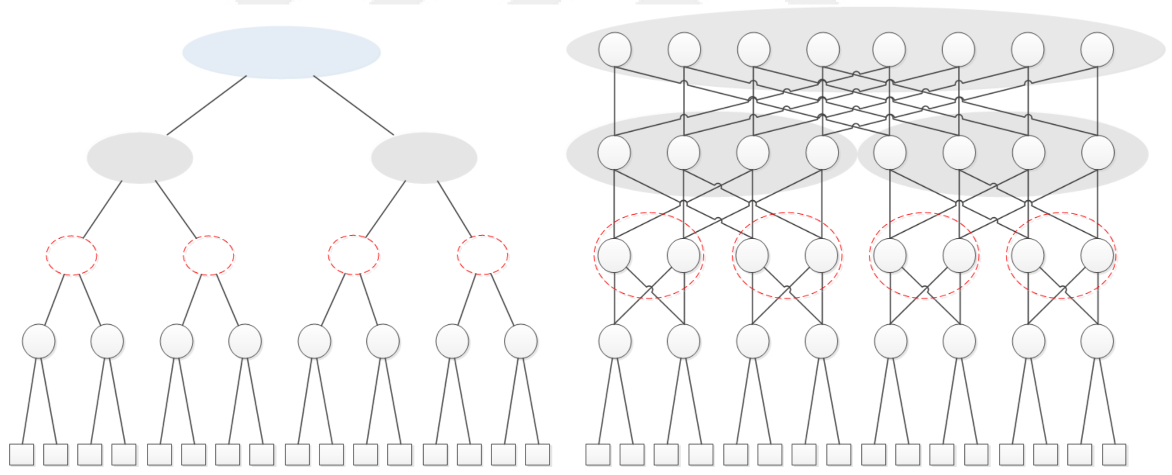


Figure 6.1. Example switch configurations: (a) a tree and (b) a fat-tree.

Our auction based scheduling algorithms described in Chapters 3, 4, 5 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 Chapters 4 and used in our AUCSCHED1 model. 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 6.1. List of symbols, their meanings, and definitions for AUCSCHED3.

Symbol	**Meaning**
J	Set of jobs that are in the window: $J = \{j_1, \ldots, j_{\|J\|}\}$
P_j	Priority of job j
P_{min_diff}	Minimum absolute priority difference between any pair of jobs in the window
N	Set of nodes : $N = \{n_1, \ldots, n_{\|N\|}\}$
N	Set of nodes requested by bid i: $N = \{n_1, \ldots, n_{\|N\|}\}$
B	Set of all bids, $B = \{b_1, \ldots, b_{\|B\|}\}$
B_j	Set of bids for job j : $B_j = \{i_1, \ldots, i_{\|B_j\|}\}$ variables
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
K_{ji}	A heuristically assigned cost value in $(0, 3]$ of bid i of job j
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.
R_i^{gres}	number of generic resources per node requested by bid i.
R_j^{gres}	minimum number of generic resources per node requested by job j.
$R_j^{gres,max}$	maximum number of generic resources per node requested by job j. If not set, this is taken as equal to R_j^{gres}.
R_j^{cpu}	Number of cores requested by job j
A_n^{cpu}	number of available CPU cores on node n
A_n^{gres}	number of available generic resources on node n
R_j^{cpn}	Number of cores per node requested by job j. If not specified, this parameter gets a value of 0.
β	a constant multiplying K_{ji}
b_i	binary variable corresponding to bid i

Given the definitions and names of the symbols in Table 6.1, 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:

$$\textbf{maximize} \quad \sum_{j \in J} \sum_{i \in B_j} (P_j - \beta \cdot K_{ji}) \cdot b_i \tag{6.1}$$

subject to constraints :

$$\sum_{i \in B_j} b_i \leq 1 \; for \; each \; j \in J \tag{6.2}$$

$$\sum_{j \in J} \sum_{i \in B_j} b_i \cdot T_{in} \leq A_n^{cpu} \; for \; each \; n \in N \tag{6.3}$$

$$\sum_{j \in J} \sum_{i \in B_j} b_i \cdot U_{in} \cdot R_i^{gres} \leq A_n^{gres} \; for \; each \; n \in N \tag{6.4}$$

The objective function given by (6.1) is different from that used in our previous AUCSCHED1 model formulation. In the objective function, we subtract the cost K_{ji} (multiplied by a constant called β) of each bid from the priority of the job. The idea is to let the factor $\beta \cdot K_{ji}$ slightly adjust the priority by a small amount in order to disfavour topologically not so good bids from being selected. The constraints (eqs. (6.2) to (6.4)) are the same as those used in AUCSCHED1 model and are explained in Section 5.1.

Our heuristic basically involves defining a function K_{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 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 β:

$$K_{ji} = 1.0 + \frac{N_i}{|N|} + \frac{L_i}{L_{max}} - \frac{R_i^{gres}}{G_{max}} \tag{6.5}$$

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 $K_{ji} \in (0, 3]$, since $\frac{N_i}{|N|} \in (0, 1]$, $\frac{L_i}{L_{max}} \in [0, 1]$ and $\frac{R_j^{gres}}{G_{max}} \in [0, 1]$.

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

$$P_{min\_diff} = \min_{j_1, j_2 \in J} |P_{j_1} - P_{j_2}| \tag{6.6}$$

We choose the scaling constant β as follows:

$$\beta = \frac{P_{min\_diff}}{3} \tag{6.7}$$

Now, we can prove that this β value always preserves the priority order. We do proof by contradiction. We want to show that if $P_{j_1} > P_{j_2}$, then $P_{j_1} - \beta \cdot C_{j_1,i_1} > P_{j_2} - \beta \cdot C_{j_2,i_2}$ 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.:

$$P_{j_1} - \beta \cdot C_{j_1,i_1} \leq P_{j_2} - \beta \cdot C_{j_2,i_2} \tag{6.8}$$

Then,

$$P_{j_1} - P_{j_2} \leq \beta \cdot (C_{j_1,i_1} - C_{j_2,i_2}) \tag{6.9}$$

$$P_{min\_diff} \leq P_{j_1} - P_{j_2} \leq \beta \cdot (C_{j_1,i_1} - C_{j_2,i_2}) \tag{6.10}$$

$$P_{min\_diff} \leq P_{j_1} - P_{j_2} \leq \frac{P_{min\_diff}}{3} \cdot (C_{x,i_1} - C_{j_2,i_2}) \tag{6.11}$$

$$3 \leq C_{j_1,i_1} - C_{j_2,i_2} \tag{6.12}$$

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

$$-3 < C_{j_1,i_1} - C_{j_2,i_2} < 3 \tag{6.13}$$

Inequality (6.12) then contradicts with inequality (6.13). Therefore, the value of β given in equation (6.7) maintains the order of priorities of jobs in the window after they are adjusted by the costs.

Figure 6.2 shows an example assignment of 5 jobs to a 16 node system with tree topology using AUCSCHED3.

6.1. aucsched3 Slurm Implementation

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. Slurm documentation [101] 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."*

Slurm document [101] 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 6.1(b) as the

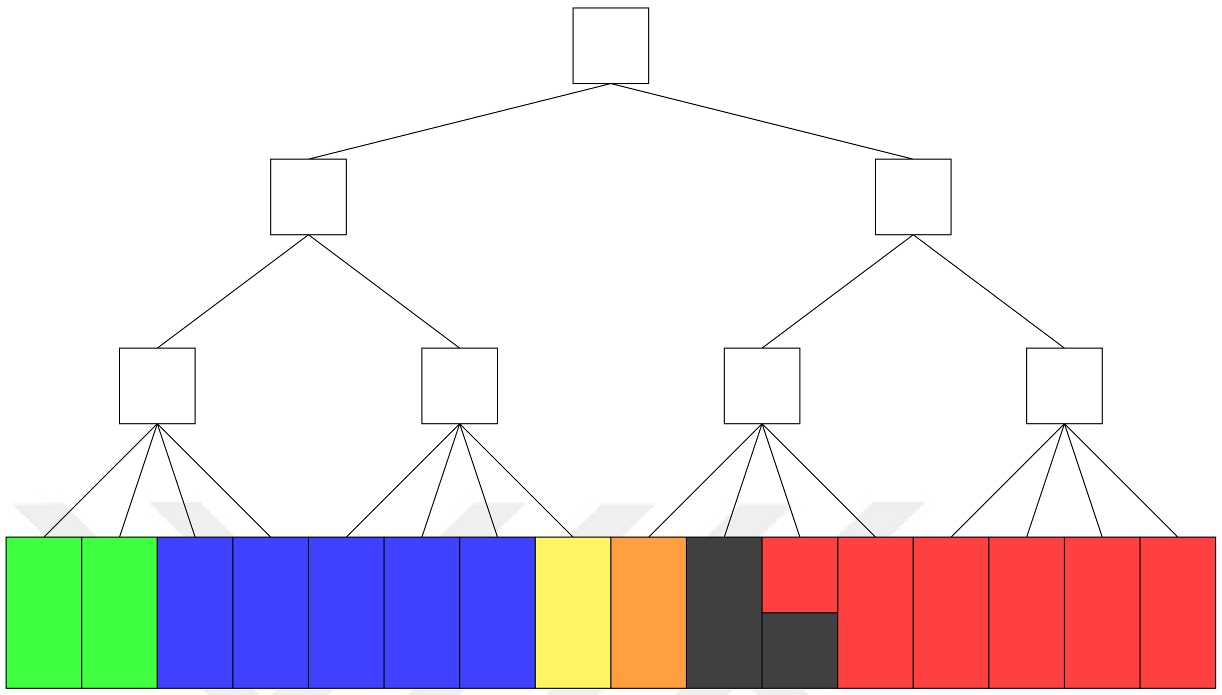


Figure 6.2. An example assignment of jobs to 16 nodes on a two level tree by AUCSCHED3. The job requirements are same as in Figure 3.3.

one in Figure 6.1(a). Therefore we consider the fat-tree topology as a tree topology in our implementation.

This model is also implemented in Slurm, and it is named *aucsched3*. *aucsched3* is publicly available at `https://github.com/aucsched/aucsched3`. The implementation of *aucsched2* is very similar to the implementation of *aucsched*. Hence, the implementation details regarding preference value calculation, nodeset generation, bid generation and general working logic explained above still hold for *aucsched2* except the nodeset generation scheme.

The bid generation phase used in AUCSCHED3 model is similar to that of AUCSCHED1 model. In AUCSCHED1 model, a 1-D array of nodes is scanned in order to generate *nodesets* which are contiguous sets of nodes (see Section 4.1 for detailed description of nodesets). Using the nodesets, bids of jobs are then generated. In AUCSCHED3 model, 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.

7. EXPERIMENTAL SETUP

In this chapter, we present the details of our setup for experiments that we carried out to asses the performance of our algorithms. We first discuss scheduling mechanisms and plugins of Slurm. We then explain our testing methodology which is based on emulation rather than simulation. Workloads that are used and the test environment are discussed next. Finally, performance metrics used are presented.

7.1. Slurm Implementation

To test the performance of the models, we have implemented and integrated the models proposed in the previous chapters into Slurm as scheduling plug-ins. Due to the reasons described in Section 7.2, instead of coding a simulator, we wanted to integrate our scheduler models into a resource-job management system, so that it would give us the chance to see its performance on real systems, and that the scientific community would be able to use it on real systems. We chose Slurm for this purpose, since Slurm is one of the most widely used scheduler [102, 103] in the Top500 supercomputers list. It is also open-source, portable and scalable [104].

In Slurm, the backfilling mechanism works as follows: The first job in the queue is considered for execution on available resources. If it can be started (i.e. requested resources can be satisfied), the job goes to the resource selection phase. If it cannot be started, the earliest possible future time to start that job is calculated, and a reservation is created for that job in the future. In the resource selection phase, the best-fit nodes are selected that satisfies the constraints of the job, such as required nodes and generic resources etc.

The scheduling mechanism is implemented as plugins in Slurm. A scheduling plugin chooses which job to run at a time, and a resource selection plug-in chooses the appropriate resources to run this job on. Our models work in such a way that, the scheduling plugin also decides on the resources required for this job. *ipsched*, *aucsched*,

aucsched2 and *aucsched3* are publicly available at `https://github.com/aucsched/ipsched`, `https://github.com/aucsched/aucsched`, `https://github.com/aucsched/aucsched2` and `https://github.com/aucsched/aucsched3`, respectively. We have also developed a resource selection plugin which works as a counterpart of the scheduling plugin, named *lpconsres*.

7.2. Simulation vs Emulation

In a simulation, a specific aspect of the system is focused, and the rest of the system is not considered. The main advantage is that the experiment is reproducible and controllable. Slurm simulator [105] can intercept time calls in Slurm, and speed up tests in Slurm for different scheduling and priority settings.

An emulation on the other hand, allows us to use a model of the system and the actual software of the system. Slurm has an emulation mode, in which, 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. In emulation tests, jobs are submitted to an actual Slurm system just like in real life Slurm usage. 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 in our approach the most time consuming operation is the solution of the NP-hard problem using CPLEX, the simulation approach would not be attractive to us. Instead, we have decided to carry out realistic Slurm emulation tests, so the Slurm system we run is actually active, and the jobs in the workload are submitted just like on a real supercomputer. The only difference is that the jobs do not include any real computation or communication, but they are only sleep jobs.

7.3. Workloads and Test Conditions

A workload of a supercomputer consists of sequence of jobs, submitted at arbitrary times by different users [49]. The arrival times of these jobs are unknown to the scheduler, hence, this scenario is also called on-line. The workload selection is one of the most important part for experimentation.

The parallel workload archive [106] is a repository which includes information about the supercomputers and their workloads. It is widely used in order to test different schedulers, priority and QOS calculations. In the archive, there are 31 different workloads from 1993 to 2010 ranging from machines with 64 cores to 163,840 cores. However, none of these workloads include any information about GPUs. Also, logs maintained in the PWA [106] range from 4 months to years. Since we are performing simulation instead of emulation, we have to make sure that the tests are can be completed in a reasonable amount of time. Besides, it has been shown that if systems are underloaded, system performances are typically the same. We should, therefore, consider higher load conditions in order to expose the differences of systems [107].

To test the performance of our scheduler models and the implemented plugins, we use three kinds of workloads:

- Workloads derived from ESP (Effective System Performance) benchmarks [108] (to be more precise, derived from version 2, i.e. ESP-2).
- Workloads generated by our own workload generator.
- Workloads generated by our own workload generator with GPU ranges capability.

The workloads used in our tests are described in detail in the next subsections.

7.3.1. ESP Workloads

ESP workload [108] is designed to test the performance of a scheduler under a heavy workload. Table 7.1 lists the normalized job sizes (no. of cores requested by job), count and execution time characteristics for the ESP workload. The sum of the completion times of the original ESP jobs is 10984 seconds (3.05 hours). All ESP jobs, except Z-jobs are submitted randomly with inter-arrival times that have a Gaussian distribution. The Z-jobs are submitted at 40th and 120th minutes.

In ESP workloads, the job's resource requests are made in such a way that a job spans across multiple nodes, but since the number of cores a job requests is the multiple of the number of cores on any node, no two jobs can run on any given node. We have modified the ESP workload by duplicating each job, and letting the duplicate job request 2 GPUs per node along with the CPU-cores it already requests. 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 [108]. In our modified ESP workload set that resulted, there are a total of 458 jobs, and the sum of completion times is 21822 seconds (6.06 hours).

7.3.2. Our workload generator

Since original ESP workloads do not have node, core per nodes and GPUs requests, we have decided to implement our own workload generator that generate workloads with such resource requests. The workload generator that was developed takes the following as input :

- the maximum number of cores a job can request,
- minimum and maximum execution times of each job,
- estimated run time of a workload,
- job type as explained in Section 4.1,
- number of CPU cores per node if specified by a job (a list of possible values),
- number of CPU cores per GPU if specified by a job (a list of possible values).

Table 8.5 shows the workloads generated and their different kinds of job compositions. The execution time of each job in the workload is distributed uniformly between

Table 7.1. 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

60 and 600 seconds. Other parameters such as the job type, the number of CPU cores per node and the number of cores per GPU are selected from a list of possible values uniformly. The number of CPU cores, R_j^{cpu}, a job requests is taken as multiples of the CPU cores per node in the system. Then, given this R_j^{cpu} number of cores:

- If a job requests cores only, then Slurm option `-n` R_j^{cpu} is submitted.
- If a job requests nodes and cores, `--cores-per-node` option is taken to be either 4 or 8. The number of nodes specified with `-N` option is then taken to be $R_j^{cpu}/4$ or $R_j^{cpu}/8$ respectively.
- If a job requests R_j^{gpu} GPUs per node, then the number of nodes is given by $R_j^{cpu}/(x \cdot R_j^{gpu})$, where x is the number of cores per GPU, given in Table 8.5.

Table 7.2. Workload types and job distributions.

	core request only	**node request (4 or 8 cores/node)**	**1 GPU/node request (1 or 2 cores/GPU)**	**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%

7.3.3. Workloads generated with GPU ranges

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.

Table 7.3. 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`

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 generic resource ranges scale their performance with the number of generic resources allocated on a node. We assume that not all jobs can scale in a linear fashion with the number of generic resources allocated, therefore we use normal distribution to scale the execution time. On a node, there are a few GPU cards. If there is 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, hence, 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:

$$t' = t \cdot N(1, 0.5) \cdot \left(\frac{minimum\ no.\ of\ gres\ requested}{no.\ of\ allocated\ gres} \right) \tag{7.1}$$

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 $150 \cdot N\left(1, 0.5\right) \cdot \left(\frac{1}{2}\right)$, which is normally distributed with a mean of 75 seconds.

It should be noted that this new execution time is decided by the job itself and not by the scheduler. Therefore, the scheduler is unaware of the jobs actual execution time, t', but is only informed of the wallclock time by the user, t. The user defines a *time limit*, but the job itself may take less time than planned, when using multiple GPUs. Since we design online schedulers, the execution time is unknown, and the jobs are scheduled without prior knowledge.

7.4. Performance Metrics

The results obtained in the tests are analyzed using the following performance measures:

- *Theoretical Runtime:* Theoretical run-time (T_{theo} in Equation 7.3) is taken to be the summation of duration of each job times its requested core count, all divided by the total number of cores in the system. This theoretical run time assumes a homogeneous, CPU-only system. 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 [46].

$$T_{theo} = \frac{\sum_{j \in J} t_j \cdot R_j^{cpu}}{A_n^{cpu}} \tag{7.2}$$

- *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.

$$U = \frac{T_{theo}}{T_{actual}} \tag{7.3}$$

- *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:* The waiting time of each job in the workload, from submission until scheduling.
- *Packing Factor:* The ratio of the number of nodes a job is allocated to the minimum number of nodes that job could be allocated onto (i.e. packed into).
- *Fragmentation:* Number of contiguous node blocks allocated to a job. See Figure 7.1 for an example on a system with linear topology.
- *Lowest level common switch:* The lowest level common switch from which all the nodes allocated to a job can be reached. See Figure 7.2 for an example. Average value for all the jobs is reported in the table.
- *Spread:* Ratio of the difference between the indices of the last and first nodes plus one to the number of nodes allocated to a job (assuming node indices start with 1). See Figure 7.1 for an example for a system with linear topology and Figure 7.2 for a system with tree topology.

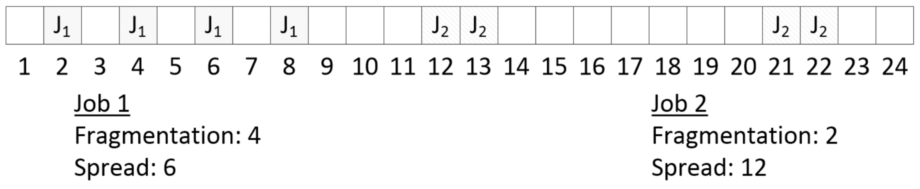


Figure 7.1. Fragmentation and spread values for two jobs, J_1 and J_2.

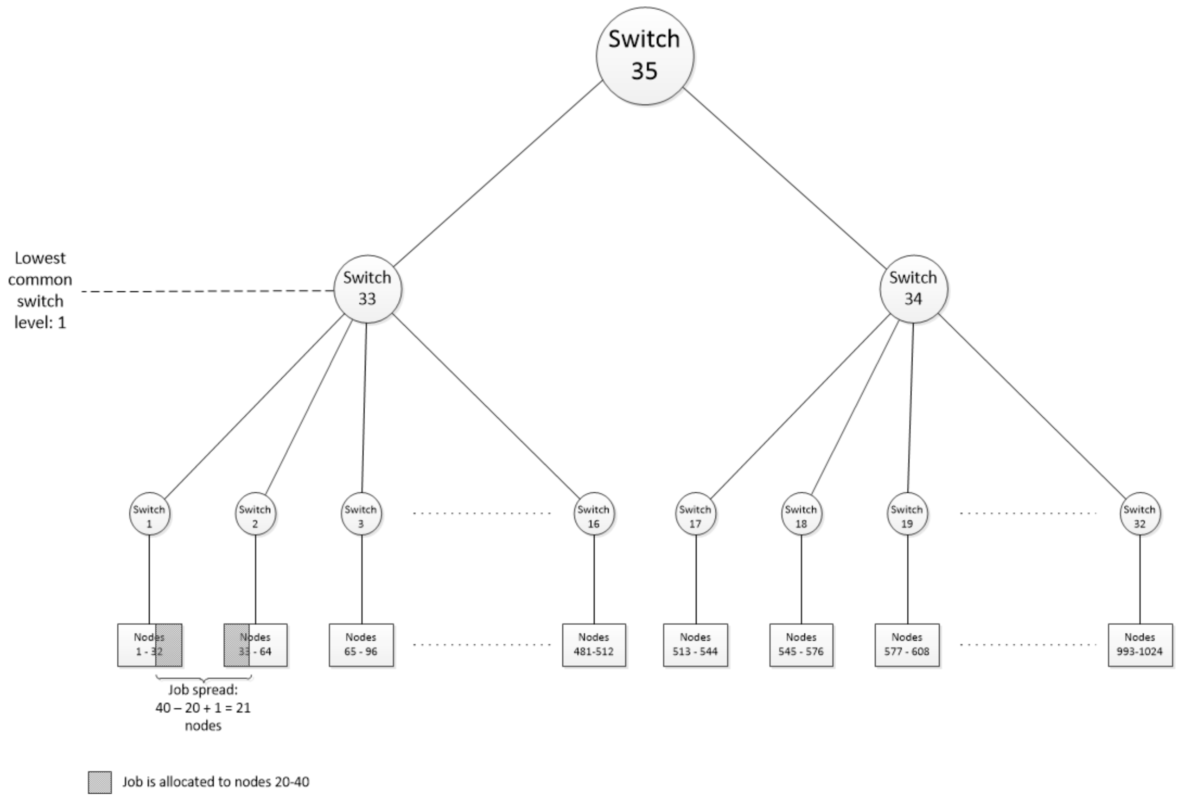


Figure 7.2. An example showing job spread and lowest level common switch metrics.

8. RESULTS and DISCUSSION

8.1. General comparison of the models introduced

The first model, IPSCHED presented in Chapter 3, does not pay attention to the location of nodes allocated to jobs. The nodes assigned to a job may be close or far apart in the network topology. If multiple processes of a job have high communication when placed on different nodes, the time spent for communication may exceed the time allocated for computation. Therefore, in the second model, AUCSCHED1 given in Chapter 4, we also pay attention to the topology of the system. The jobs are allocated to the nodes in close vicinity to each other as much as possible. The system is considered as a 1-D array of nodes. In this problem, jobs may also explicitly request a contiguous set of nodes. Our third model, AUCSCHED2 given in Chapter 5, considers another integer programming problem. It is similar to the second one, but in this model, we reduce the number of variables in the IP problem and allow users to submit jobs with GPU ranges, therefore, allowing the user jobs to be malleable from system perspective. In the fourth model, AUCSCHED3 given in Chapter 6, we allow the system to have a hierarchical interconnect topology, and use the switch information in order to obtain topology aware allocation of resources to jobs.

Table 8.1. Number of variables in the IP problems for all models developed in this thesis.

	IPSCHED	**AUCSCHED1**	**AUCSCHED2**	**AUCSCHED3**
Number of variables	$2\|J\| \cdot (1+\|N\|)$	$2 \cdot \|N_c\| + \|B\|$	$\|B\|$	$\|B\|$
Number of constraints	$2 \cdot (\|J\| + 2\|N\| + \|J\|\|N\|)$	$2 \cdot \|N_c\| + \|B\|$	$\|J\| + 2\|N\|$	$\|J\| + 2\|N\|$

Table 8.1 summarizes the number of variables in the integer programming formulation for each model. Table 8.2 summarizes the features supported in each model. In

these tables, the oldest model developed is IPSCHED and the newest is AUCSCHED3. As can be observed, the number of variables in the newer models are lower than those in the earlier model. Also, the features that are supported by each new model are a superset of those in the earlier models, e.g. any feature that exists in AUCSCHED2 for example also exists in AUCSCHED3.

Table 8.2. Features for all models developed in this thesis.

Option	**Explanation**	**IPSCHED**	**AUCSCHED1**		
			1	**2**	**3**
-n	number of cores	✓	✓	✓	✓
-N	number of nodes	✓	✓	✓	✓
–ntasks-per-node	number of cores per node		✓	✓	✓
–gres=gpu:X	X gpu's per node	✓	✓	✓	✓
–contiguous	contiguous node allocation		✓	✓	✓
–gres=gpu:X-Y	range of GPUs [X-Y]			✓	✓
topology	system topology	none	1-D	1-D	Tree

8.2. Comparison of IPSCHED and AUCSCHED1

In this section, we compare our first two models: IPSCHED and AUCSCHED1 with each other and the Slurm Backfill plugin. Table 8.3 summarizes the characteristics of the emulation experiments that we have carried out. Two sets of priority options were selected for conducting the tests. These are basic [109] and multifactor [110]. Our models do not change these values, but only collects them. In this work, two types of priorities are used. Basic priority setting corresponds to a FCFS scheduling. Slurm takes the priority of the first submitted job to be a large number, and every arriving job's priority will be one less than that of the previously arrived job. Multifactor priority setting 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.

Table 8.3. Experiments and their characteristics.

Experiment	**Plug-in**	**Priority Type**	**Objective Function**
1	Slurm Backfill	Basic	-
2	Slurm Backfill	Multifactor	-
3	*IPSCHED*	Basic	$max \sum P_J \left(s_j - c_j\right)$
4	*IPSCHED*	Multifactor	$max \sum P_J \left(s_j - c_j\right)$
5	*IPSCHED*	Basic	$max \sum P_J R_j^{cpu} \left(s_j - c_j\right)$
6	*IPSCHED*	Multifactor	$max \sum P_J R_j^{cpu} \left(s_j - c_j\right)$
7	*AUCSCHED1*	Basic	$max \sum_{j \in J} \sum_{c \in B_j} \left(P_j + \alpha \cdot F_{jc}\right) b_{jc}$
8	*AUCSCHED1*	Multifactor	$max \sum_{j \in J} \sum_{c \in B_j} \left(P_j + \alpha \cdot F_{jc}\right) b_{jc}$

Every test is repeated under equal conditions for each workload and for each experiment given in Table 8.3. For ESP workloads, 56 tests are conducted (7 repetitions $\times$ 8 experiments). For our workloads, 280 tests are conducted (7 repetitions $\times$ 5 workloads $\times$ 8 experiments).

Results obtained from each experiment are given in Table A.1 for ESP workloads and Table A.2 for our workloads, and are plotted in Figures 8.1 through 8.9. The test results for each experiment have been averaged in ESP workloads. Similarly, the test results for each experiment-workload pair are averaged for our workloads.

In Figures 8.1 and 8.2, we plot the utilizations obtained from ESP and our workloads respectively. When ESP workloads are used, AUCSCHED1 clearly gives the best utilization, followed by IPSCHED, and Slurm Backfill performing the worst. In the case of our workloads, IPSCHED is taking the lead by a few percentage points when I, II and III type workloads are used. In these first three workloads, there are no jobs that make requests for GPUs. In the case of workload IV which contain some jobs requesting 1 GPU per node, AUCSCHED1 is again performing the best and IPSCHED and Slurm coming behind with more or less same utilization values. When workloads contain jobs

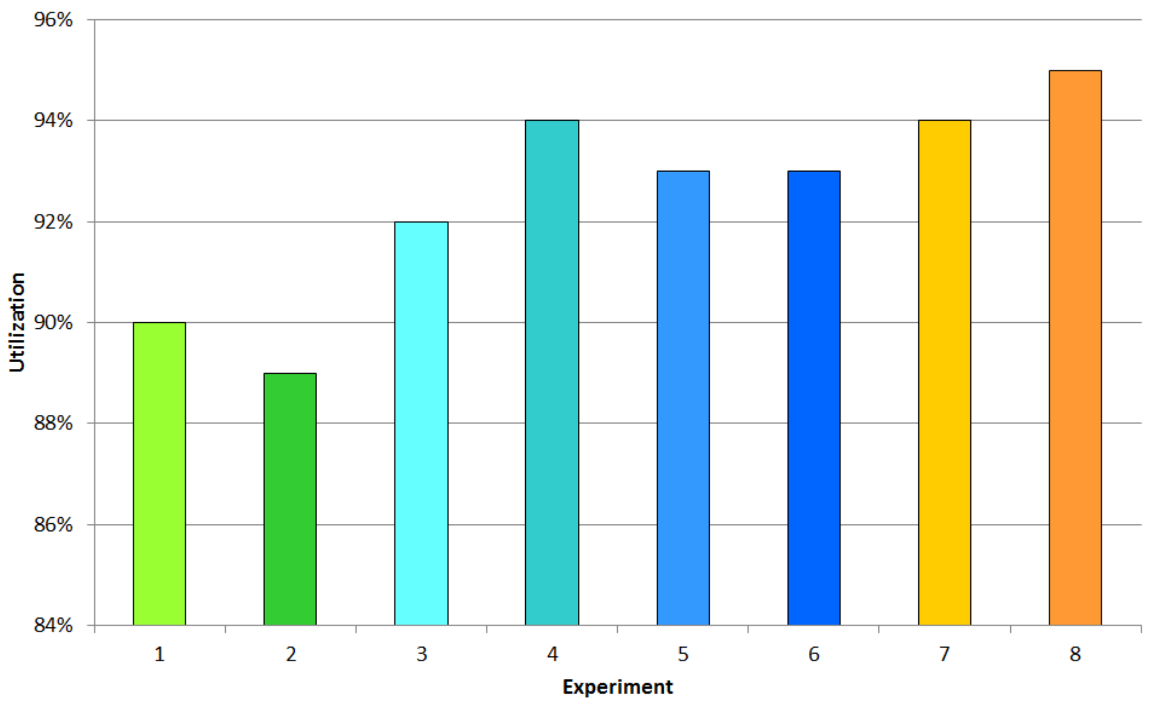


Figure 8.1. Comparison of system utilization in different experiments for ESP workloads.

that request 2 GPUs per node (i.e. in the case of workload V), it is observed that the system utilizations drops drastically to 57-70% range. As stated earlier, since the definition of theoretical run-time that we used in the calculation of utilization is only a lower bound and not the optimal value, we may be faced with the following scenarios: (*i*) The algorithms in all plug-ins are working properly but this low utilization may be close to the best that can be obtained due to more complex combinatorial nature of the problem (i.e. allocation of multiple unit heterogeneous CPU-GPU resources), (*ii*) the algorithms employed in all plug-ins need further improvements to produce higher utilization solutions. Both cases, however, point to the need to further study of scheduling jobs that utilize multiple GPU cards on nodes. Also, it is notable that with 70% utilization performance, AUCSCHED1 has outperformed Slurm's Backfill in this case.

When we look at the waiting times in Figures 8.3 and 8.4, we see that they are the lowest for IPSCHED, followed by Slurm Backfill, followed by AUSCHED. This is due to the fact that, IPSCHED generally favours smaller sized jobs.

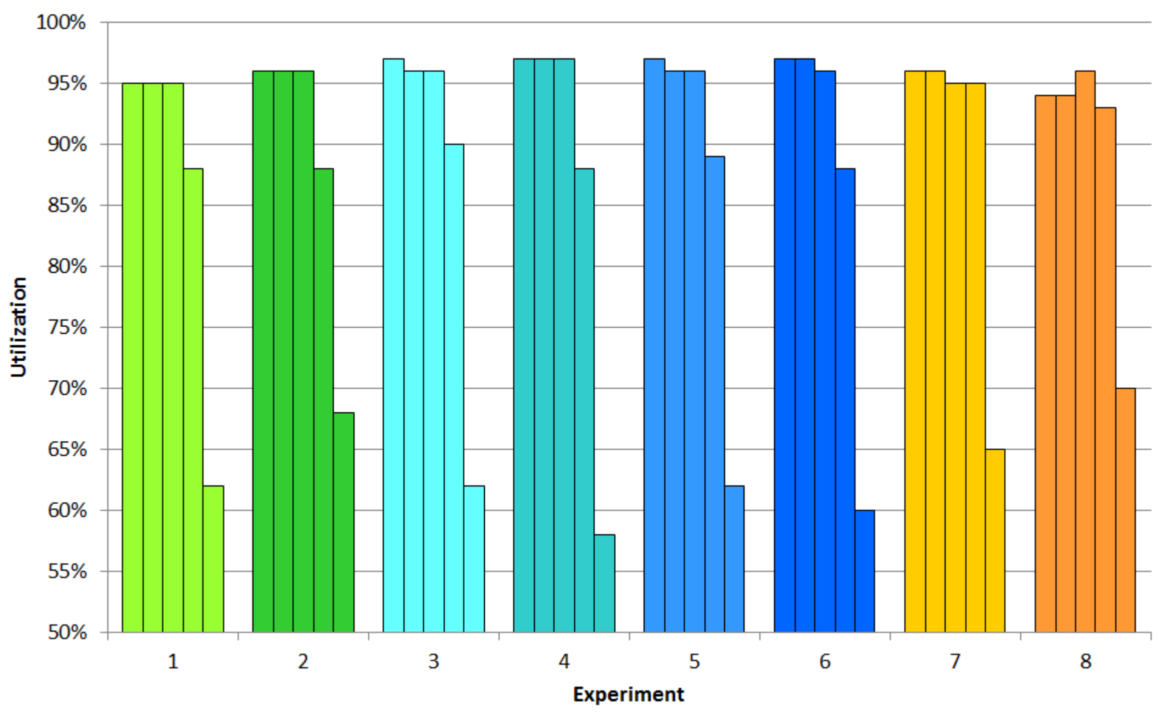


Figure 8.2. Comparison of system utilization in different experiments for our workloads.

Figure 8.5 shows the packing factor performance of the three plug-ins for ESP experiments. The jobs are best packed using AUCSCHED1, followed by IPSCHED and finally Slurm backfill. Note that packing factor has not been calculated for our workloads, since we may have explicit cores per node requests by jobs these workloads.

Figures 8.6 and 8.7 show the fragmentation performance of ESP and our workloads respectively. We do not plot fragmentation performance of IPSCHED since no optimization is done by IPSCHED to allocate contiguous blocks and hence reduce fragmentation. As the two plots show, Slurm Backfill does better job of reducing fragmentation than AUCSCHED1 in both ESP and our workloads.

Note there may be cases, where, even though an allocation can be fragmented, (i.e. placed on multiple blocks), the gaps between the blocks can still be small ; perhaps by one node. In such cases, these multiple blocks may in fact be close to each other in terms of topological placement (for example by being on the same communication switch). Hence, a large fragmentation value may not necessarily imply a bad placement

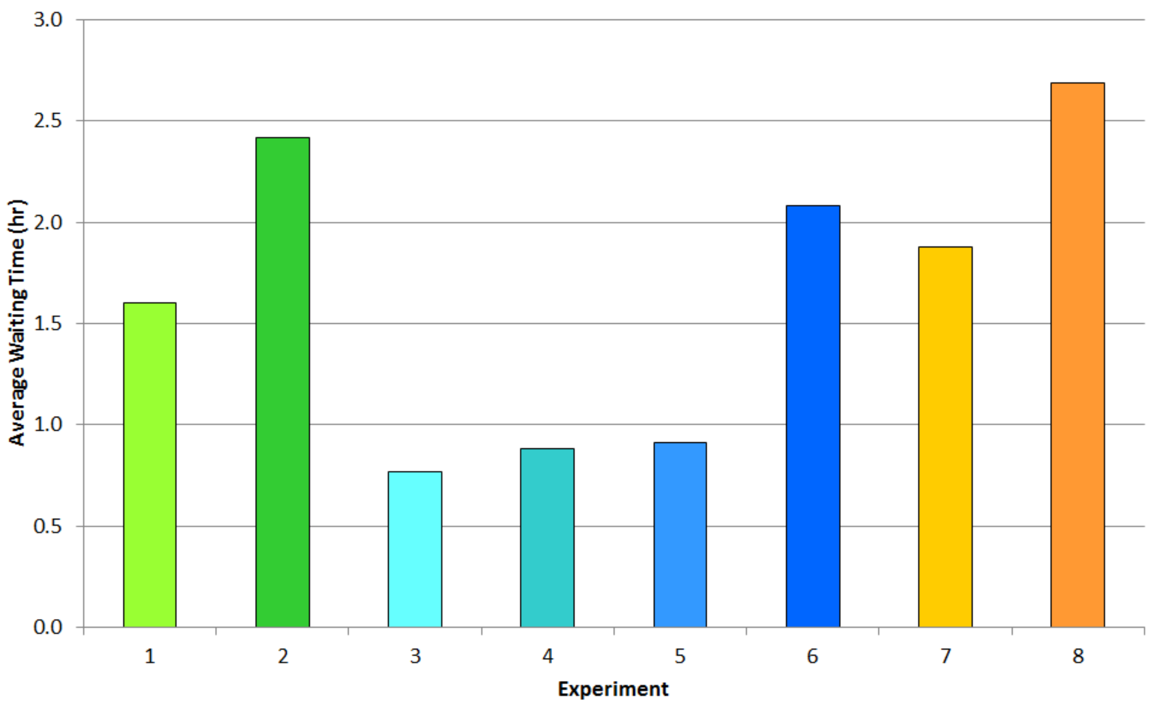


Figure 8.3. Comparison of job waiting times (in hours) in different experiments for ESP workload.

for the purpose of communication. Spread measure allows us to look into this kind of separation on allocations. As seen in plots in Figures 8.8 and 8.9, AUCSCHED1 is doing a better job of reducing the spread in allocations. We do not again report spread performance of IPSCHED since no special optimization is done by it to reduce spread.

8.3. Comparison of AUCSCHED2 with AUCSCHED1

In this section, we present our tests comparing AUCSCHED2 with AUCSCHED1 and Slurm Backfill plugins. In order to test our AUCSCHED2 model, we conduct Slurm emulation tests. The 1408 node system that we emulate is Japan's Tsubame [19] system which has 12 cores and 3 NVIDIA M2090 GPU's on each of its nodes. In this section, we compare AUCSCHED2 with AUCSCHED1 and the Slurm Backfill method.

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's Backfill plug-in, we use

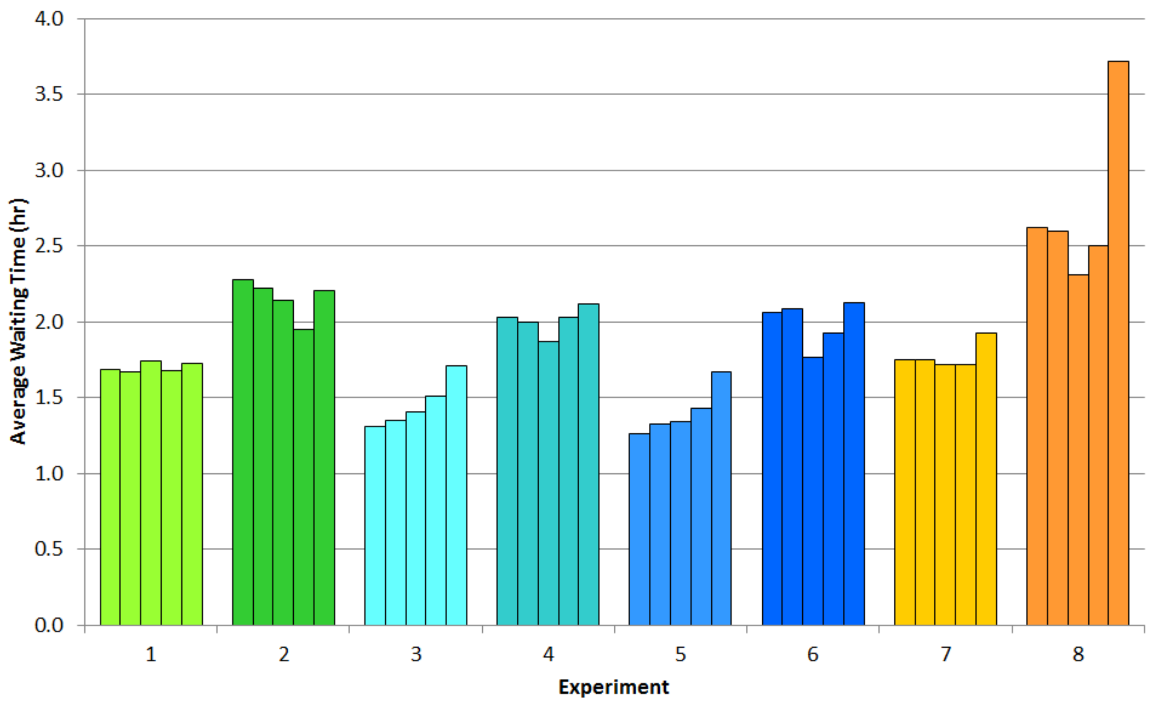


Figure 8.4. Comparison of job waiting times (in hours) in different experiments for our workloads.

type {A, B, C, D, E} jobs. AUCSCHED2 that offers generic resource 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 generic resources allocated to that job as described in Equation 7.1.

We also have a parameter that sets the ratio of jobs that request contiguous allocation in the workload. For workloads with a contiguity ratio of 0.00, 0.50 and 1.00, none, half, and all of the jobs in the workload request contiguous resource allocation, respectively. The system may still allocate a set of contiguous nodes for a job which does not explicitly request a set of contiguous nodes, since the preference value of such bids is set higher (see Equation 4.12). We have generated several workloads of different lengths in order to test the scheduler's performance more extensively. Workloads 1 to 3 and 7 to 9 have a mean theoretical runtime of 5 hours, where workloads 4 to 6 and 10 to 12 have an average theoretical runtime of 10 hours. Workloads 1 to 6 only include type A jobs, 7 to 12 include all types of jobs. Table 8.4 give a detailed explanation for

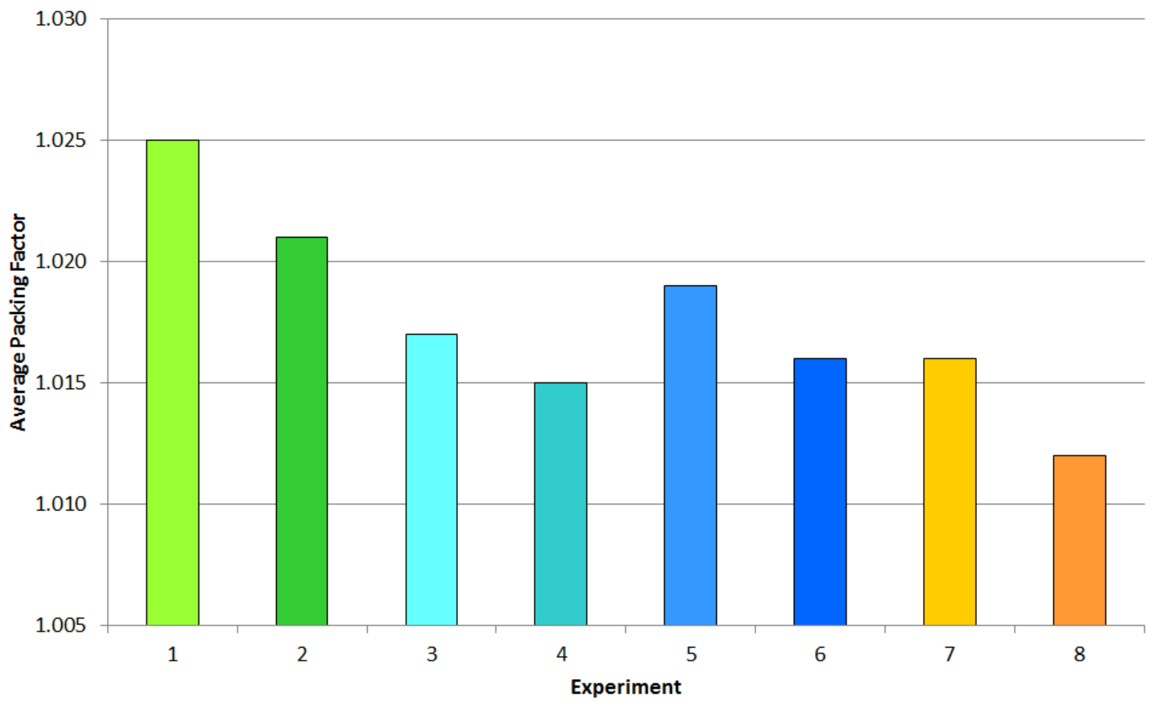


Figure 8.5. Comparison of job packing factors in different experiments for ESP workload.

the number of jobs in the workloads, the job types etc.

Note that fragmentation and spread metrics give us information about the topological goodness of allocations. Smaller fragmentation and spread values indicate a better allocation since communication costs will be lower due to allocated nodes being closer to each other. Example Figure 7.1 given in Section 7.4 illustrates fragmentation and spread values for two jobs J_1 and J_2 on a system with 24 nodes. J_1 has a spread of 6 and a fragmentation of 4. J_2 has a spread of 10 and a fragmentation of 2. Note that if nodes 2-8 are covered by one level of switch whereas nodes 12-22 are covered by two levels of switches, then this example shows that, although fragmentation is an important metric for communication, if spread is lower, job J_1 allocation can result in better communication.

When we look at the results in Table A.3 and the plots in Figures 8.10, 8.11 and 8.12, we observe the following:

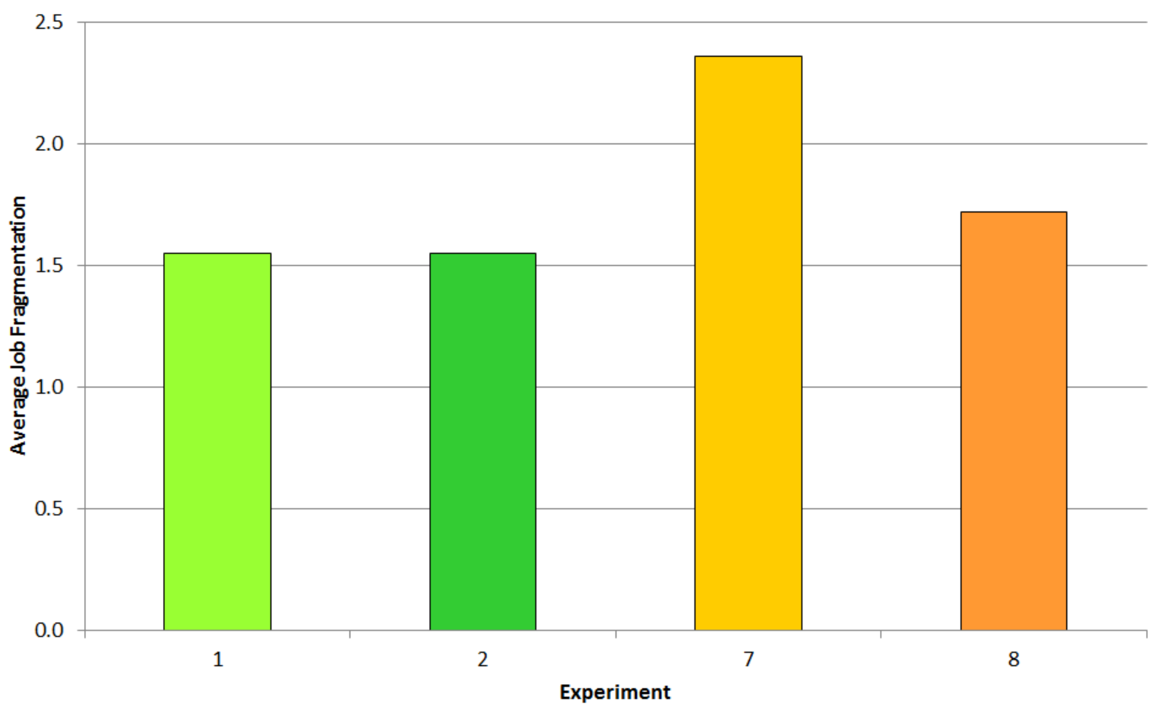


Figure 8.6. Comparison of job fragmentation in different experiments for ESP workload.

- In all the tests, AUCSCHED2 achieves better utilization than both AUCSCHED1 and Slurm's Backfill. Utilization increases between 4 to 10 percentage points are realized by AUCSCHED2 compared to Slurm's Backfill. In tests 7 through 12 that involves GPU ranges (moldability), AUCSCHED2 again surpasses Slurm's Backfill in utilization performance by 4 to 6 percentage points. As expected, the utilizations are lower in the cases of 3, 6, 9 and 12 where all jobs request contiguous allocation (ranging between 82% to 92% for AUCSCHED2 and 79% to 83% for Slurm's Backfill) since the additional contiguity parameter may make it more difficult to find a contiguous allocation for a jobs (even though non-contiguous allocations may exist).
- When we compare waiting time metric for three plugins, we observe that the the waiting time is smaller in AUCSCHED2 in both cases. The ratio of waiting time ranges from 1.03 to 1.07 between AUCSCHED1 and AUCSCHED2 and from 1.05 to 1.12 between Slurm Backfill and AUCSCHED2.
- In all the tests (except for 3, 6, 9 and 12), AUCSCHED2 achieves better spread measures than both AUCSCHED1 and Slurm's Backfill. In tests 3, 6, 9 and

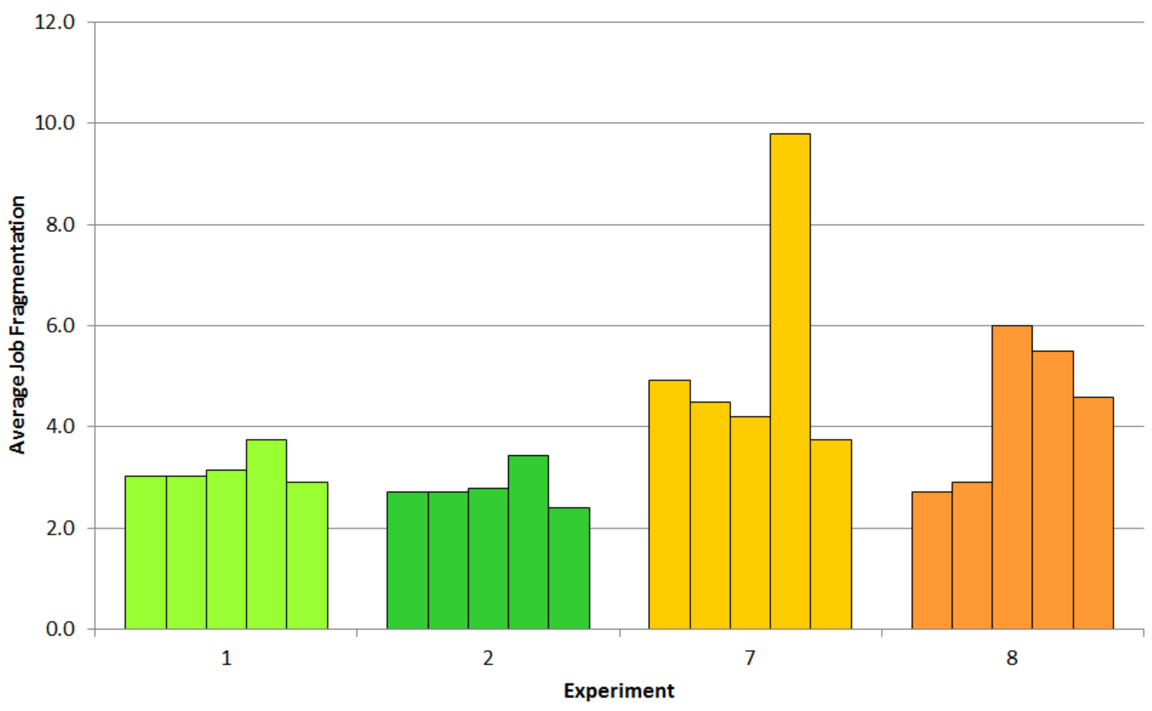


Figure 8.7. Comparison of job fragmentation in different experiments for our workloads.

12, all jobs request contiguous allocation and hence spread values should be 1 for both AUCSCHED2 and Slurm Backfill. In other cases, the ratio between the spread values range from 1.54 to 2.61 for AUCSCHED2 and Slurm's Backfill and from 1.07 to 1.51 for AUCSCHED2 and AUCSCHED1. It is interesting to note here that AUCSCHED2 achieves higher utilizations than Slurm Backfill by 4 to 10 percentage points when all jobs are explicitly requested to be laid out in contiguous manner. We can then consider using AUCSCHED2 to do contiguous allocation for all jobs (irrespective of whether the jobs explicitly requested contiguous allocation or not) and expect to achieve better utilization than Slurm Backfill. With better spread values, we can assume that for a real system, due to lower number of hops in the network, the job execution times would be smaller for AUCSCHED2, hence the utilization would be higher than reported above.

- In workloads 1 to 6, which include only type A jobs (see Table 8.5), the fragmentation is high in AUCSCHED2 compared to AUCSCHED1 and Slurm's Backfill. The fragmentation value is equal to 1 in tests 3, 6, 9 and 12. In tests $1, 2, 4, 5$, the ratio between the spread values range from 7.75 to 2.62 for AUCSCHED2

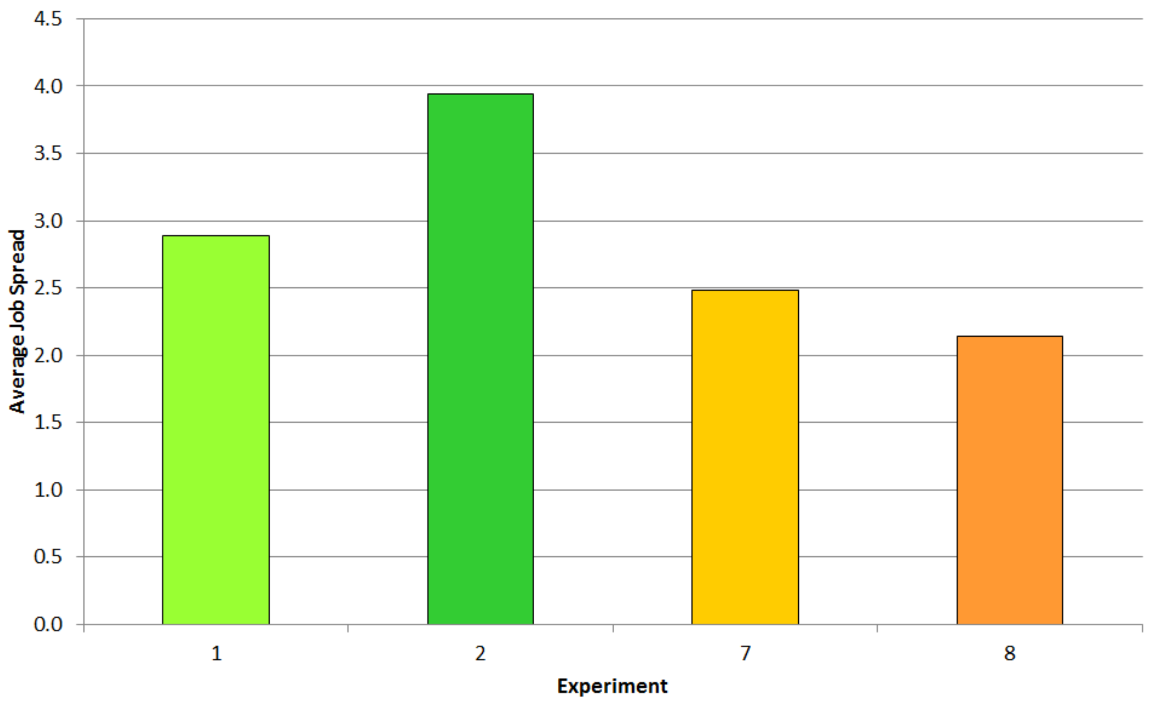


Figure 8.8. Comparison of job spread in different experiments for ESP workload.

and Slurm's Backfill and from 8.82 to 2.73 for AUCSCHED2 and AUCSCHED1. In tests $7, 8, 10, 11$ the ratio ranges between 0.93 and 1.03 for AUCSCHED2 and Slurm's Backfill and between 0.98 and 1.17 for AUCSCHED2 and AUCSCHED1. This is a design choice we made in AUCSCHED2. In order to increase the system utilization, AUCSCHED2 creates bids for type A jobs spanning a large number of nodes. Therefore, these jobs may have large fragmentation. The preference value of bids which span a large number of nodes is low.

- In tests 1 and 7, very high utilizations are achieved by AUCSCHED2 but fragmentation values are noticeably higher than that of Slurm Backfill (97% and 86% for AUCSCHED2, compared to 87% and 81% for Slurm Backfill). Even though this may look inferior when compared with Slurm Backfill, it is not so, especially, when we consider the spread values. In these two cases, spread values of AUCSCHED2 are still better than those of Slurm Backfill. Figure 7.1 illustrates that we can in fact have allocations with high fragmentation and low spread values which are superior from communication cost perspective since nodes can be closer together on the interconnection network.

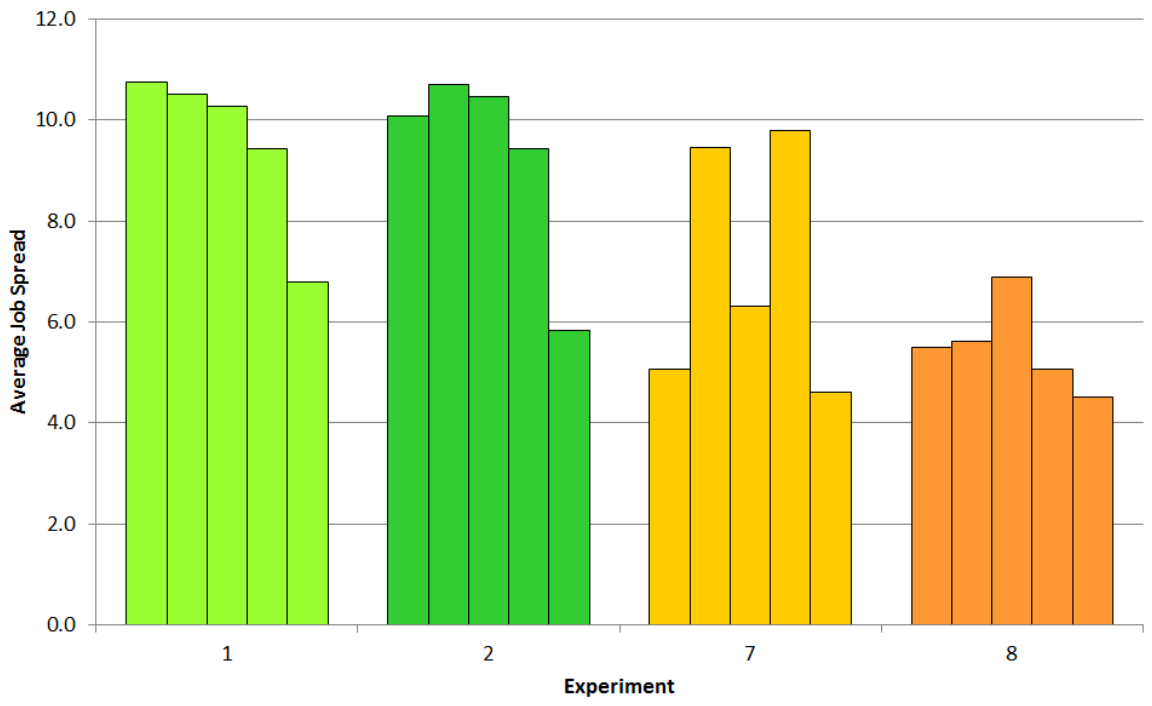


Figure 8.9. Comparison of job spread in different experiments for our workloads.

Table 8.4. Workload types and job distributions for AUCSCHED2 test.

Workload ID	Average length (hours)	Contiguity ratio	Percentage of job types				
			A	B	C	D	E
1	6.28	0.00	100	0	0	0	0
2	6.27	0.50	100	0	0	0	0
3	6.26	1.00	100	0	0	0	0
4	12.53	0.00	100	0	0	0	0
5	12.54	0.50	100	0	0	0	0
6	12.48	1.00	100	0	0	0	0
7	5.43	0.00	20	20	20	20	20
8	5.42	0.50	20	20	20	20	20
9	5.50	1.00	20	20	20	20	20
10	10.90	0.00	20	20	20	20	20
11	10.63	0.50	20	20	20	20	20
12	10.89	1.00	20	20	20	20	20

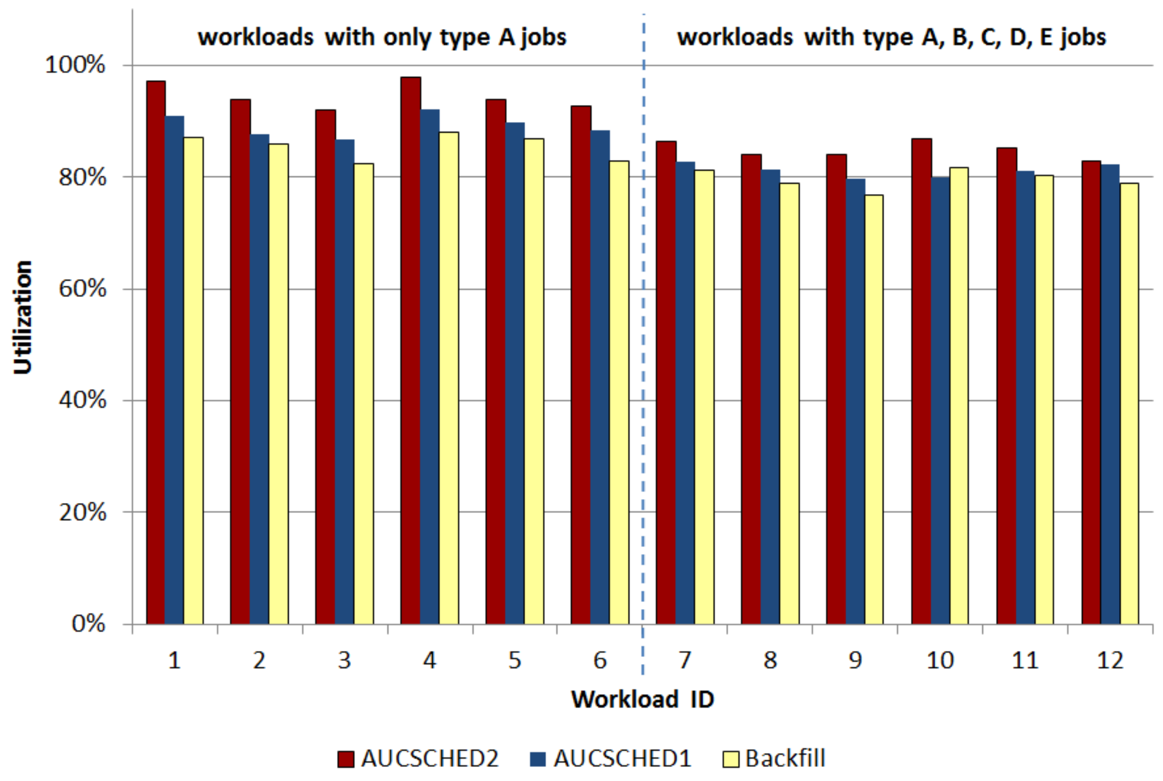


Figure 8.10. Utilization comparison in the tests.

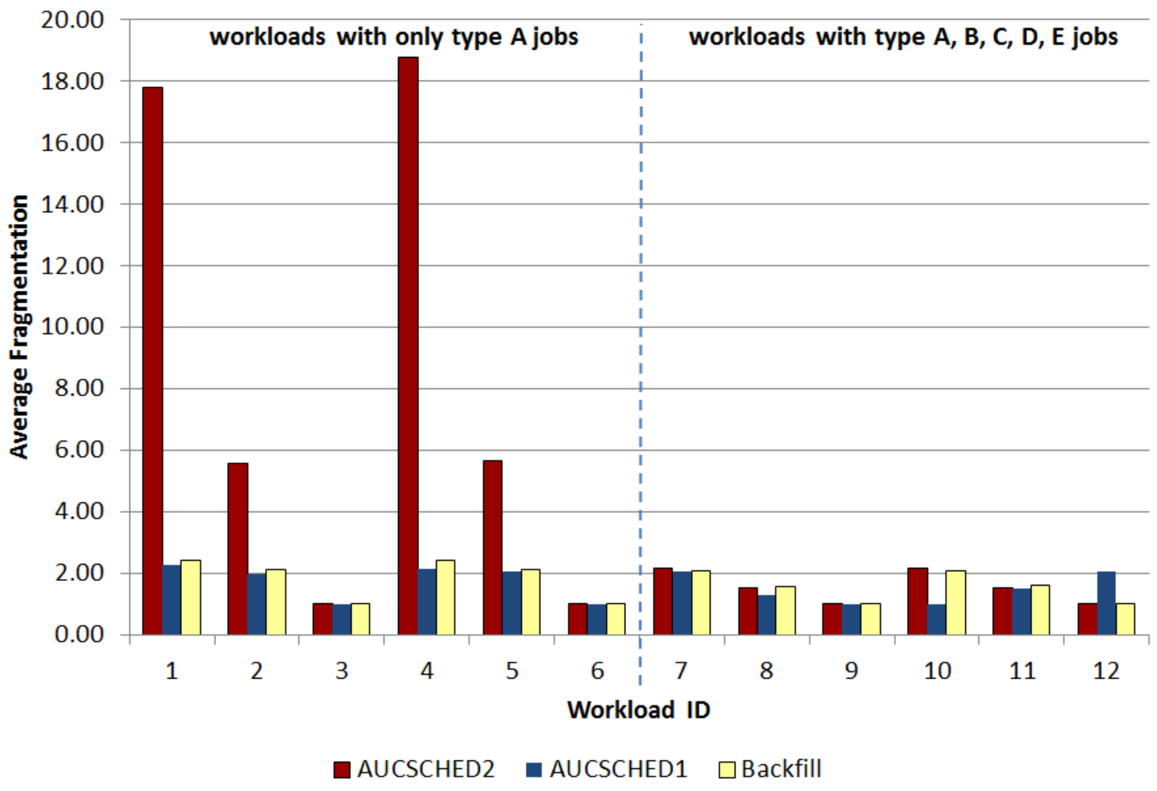


Figure 8.11. Fragmentation comparison in the tests.

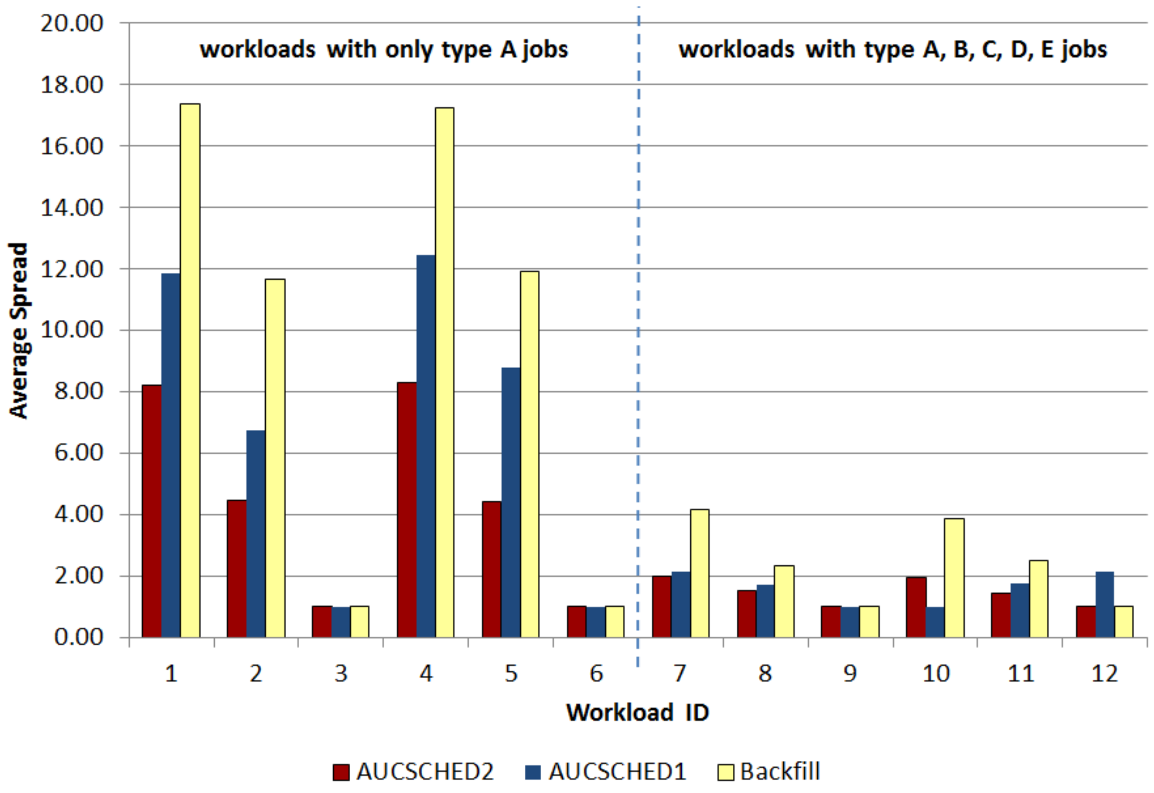


Figure 8.12. Spread comparison in the tests.

8.4. AUCSCHED3 Tests

8.4.1. Tests on a hypothetical 1024 node supercomputer

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 7.2 shows the switch topology of the system.

Table 8.5. Workload types and job distributions.

Workload	**Number**	**Percentage of job types**				
ID	**of jobs**	**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					

We design six different workloads (numbered 1 through 6) that take into consideration the types of jobs 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 7.3. This table shows the Slurm job submission commands for each type of job. The workloads are made of as various percentages of these job types and are shown in Table 8.5. As pointed out in Table 8.5, workloads 1 and 2 include jobs with only core requests. Workloads 3 and 4 includes 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. AUC-

SCHED2 and AUCSCHED3 also provides generic resource ranges support (which is not available in Slurm/Backfill). Such a feature can be useful to runtime auto-tuning applications and systems that can make use of variable number of generic resource 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.

To test our new AUCSCHED3 plug-in, we proceed in a similar fashion as before by conducting emulation tests. We are able to retrieve topology related information of allocated jobs and hence we can evaluate goodness of allocations. On the other hand, we should be careful about interpretation of results. Jobs with good topology mappings are likely to run faster since communication will be faster. Hence:

- From the users' perspectives, our allocation topology improvements are definitely important and are going to be welcomed by the users.
- From the overall system utilization perspective (i.e. the administrators' perspective), topology improvements will also increase the utilization of the system. However, this should not come at the expense of reduced overall 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 results to provide not only better topology mappings but also be accompanied by more or less the same utilizations than Slurm/Backfill.

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 *lowest level common switch*, *spread*, *utilization* performance measures explained in Section 7.4.

The results are given in Table A.4. We also plot average lowest common switch level and the average spread measures in Figures 8.13 and 8.14. 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 (119,805 jobs in total) are given in Figure 8.15(a,b) respectively. The plots in Figure 8.16(a,b) show the same information for all type 5, 6, 5' and 6' type jobs (51,345 jobs in total). We note that (a) plots show percentages of the numbers of jobs whereas (b) plots show percentages of total job durations but these distributions are almost the same.

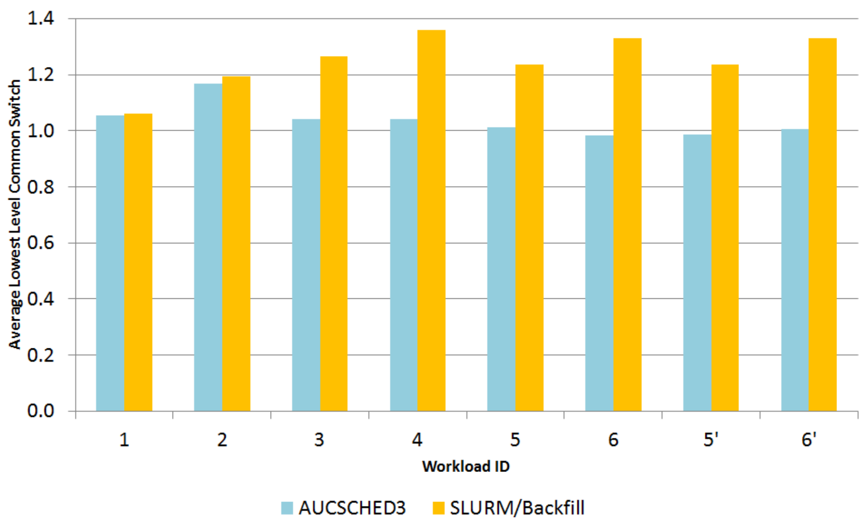


Figure 8.13. Average lowest level common switch of AUCSCHED3 compared to Slurm/Backfill plug-in.

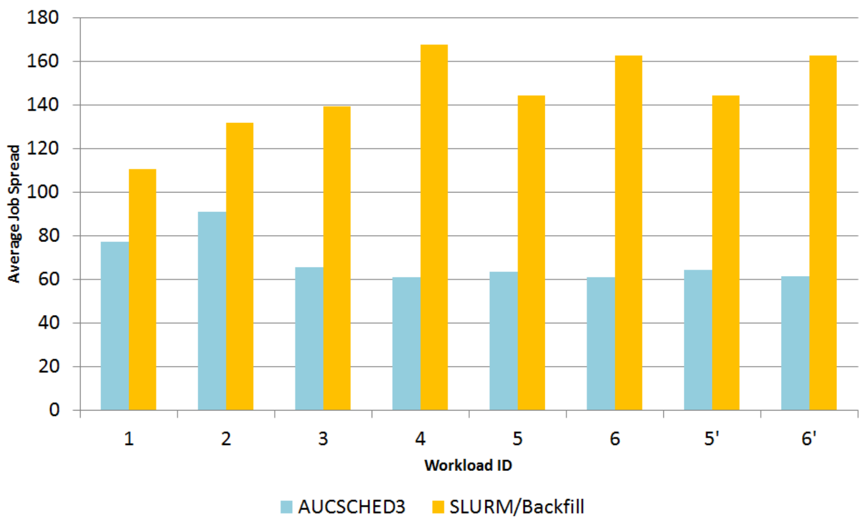
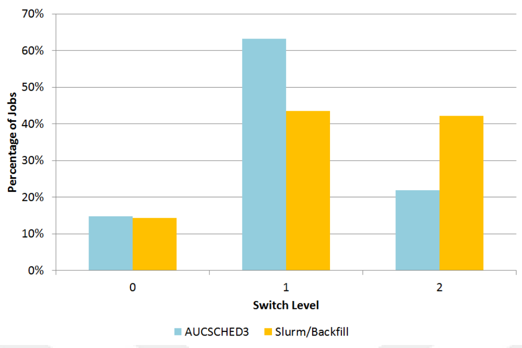
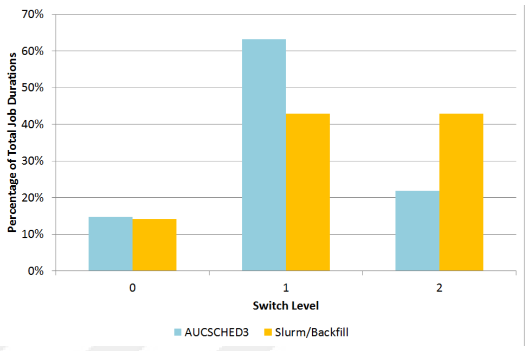


Figure 8.14. Average job spread of AUCSCHED3 compared to Slurm/Backfill plug-in.

The results we have obtained on the emulated 1024 node system shows that our

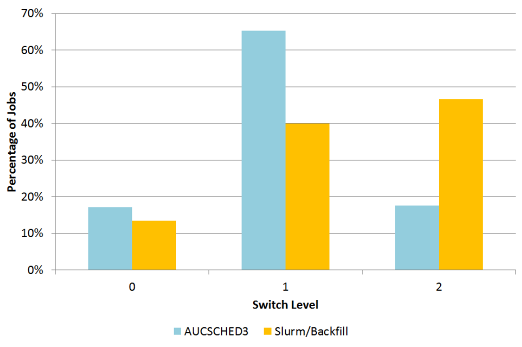


(a)

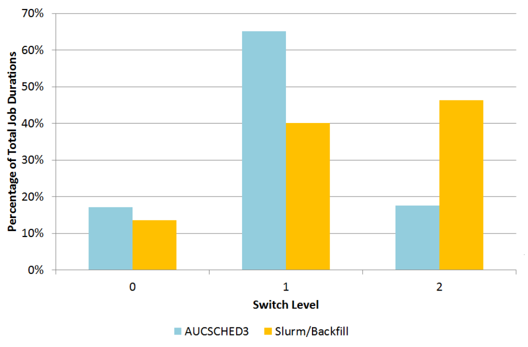


(b)

Figure 8.15. Distribution of percentage of jobs over switch levels (a) and percentage of total job durations over switch levels (b) for all workloads.



(a)



(b)

Figure 8.16. 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’.

AUSCHED3 plug-in is able to generate better topological mappings than Slurm/Backfill. 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. But, 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.

8.4.2. Tests on a simulated 1408 node TSUBAME 2.0 supercomputer

We have decided to test our plugin on a realistic system, TSUBAME 2.0 [19], which is one of the Top500 supercomputers in the world, as we did with our AUC-SCHED2 plugin. TSUBAME 2.0 built in 2011, has 1408 compute nodes with 2 Intel X5670 CPU's (12 cores) and 3 NVIDIA M2050 GPU's per node [19].

As in Figure 6.1, we have simplified the fat-tree topology to make it easier for Slurm to parse and handle. In our implementation, every 16 node is connected to a first level switch. Every 8 first level switch is connected to a second level switch. And all of the second level switches are connected to a third level switch, which is the highest level.

In order to better test our plugin, we have increased the number of jobs, and average time a workload takes to execute. We have preserved the job fractions described in Section 8.3. The details of each workload can be found in Table 8.6. Each test has been repeated 7 times on a node with two Intel X5670 6-core 3.20 Ghz CPU's and 48 GB's of memory. The average of these results are given in Table A.5 and Figures 8.17, 8.18 and 8.19.

From Figure 8.17, we can observe that *AUCSCHED3* always performs better than Slurm/Backfill in terms of utilization. The workloads in Table 8.6 are much longer than the workloads used in the previous cases. Therefore, the initial filling time becomes more negligible in the overall execution period. Therefore, utilization rates are much

Table 8.6. Workload types and job distributions.

Workload ID	**Number of jobs**	**Average Workload Time (hours)**	**Fraction of Contiguous Jobs**	**Percentage of job types**				
				A	**B**	**C**	**D**	**E**
1	3491	5.02	0.0	100	0	0	0	0
2	2095	2.98	0.0	0	100	0	0	0
3	3491	4.94	0.0	20	20	20	20	20
4	3491	5.02	0.5	100	0	0	0	0
5	2095	3.02	0.5	0	100	0	0	0
6	3491	4.95	0.5	20	20	20	20	20
7	3491	4.98	1.0	100	0	0	0	0
8	2095	2.96	1.0	0	100	0	0	0
9	3491	5.00	1.0	20	20	20	20	20

higher than the previous cases. The difference in utilization can increase up to 9%, in the workloads 3, 6 and 9, which are the cases that involve GPUs. The co-allocation feature of *AUCSCHED3* avoids the inefficiencies caused by the traditional sequential scheduling mechanisms for heterogeneous systems.

In Figure 8.18, the lowest level common switch has been compared for *AUCSCHED3* and Slurm/Backfill (see Section 7.4 for lowest level common switch description). In all workloads except 1, 4 and 7, the lowest level common switch is lower for *AUCSCHED3* by a small margin. However, in the workloads, the lowest level common switch for *AUCSCHED3* is slightly higher than Slurm/Backfill (around 4%). This is caused by the bid generation algorithm in *AUCSCHED3*, which tries to create as many bids as possible for jobs requesting only cores (type A jobs). Therefore, jobs may be too distributed in the system.

As can be observed from Figure 8.19, the job spread is always lower for *AUC-*

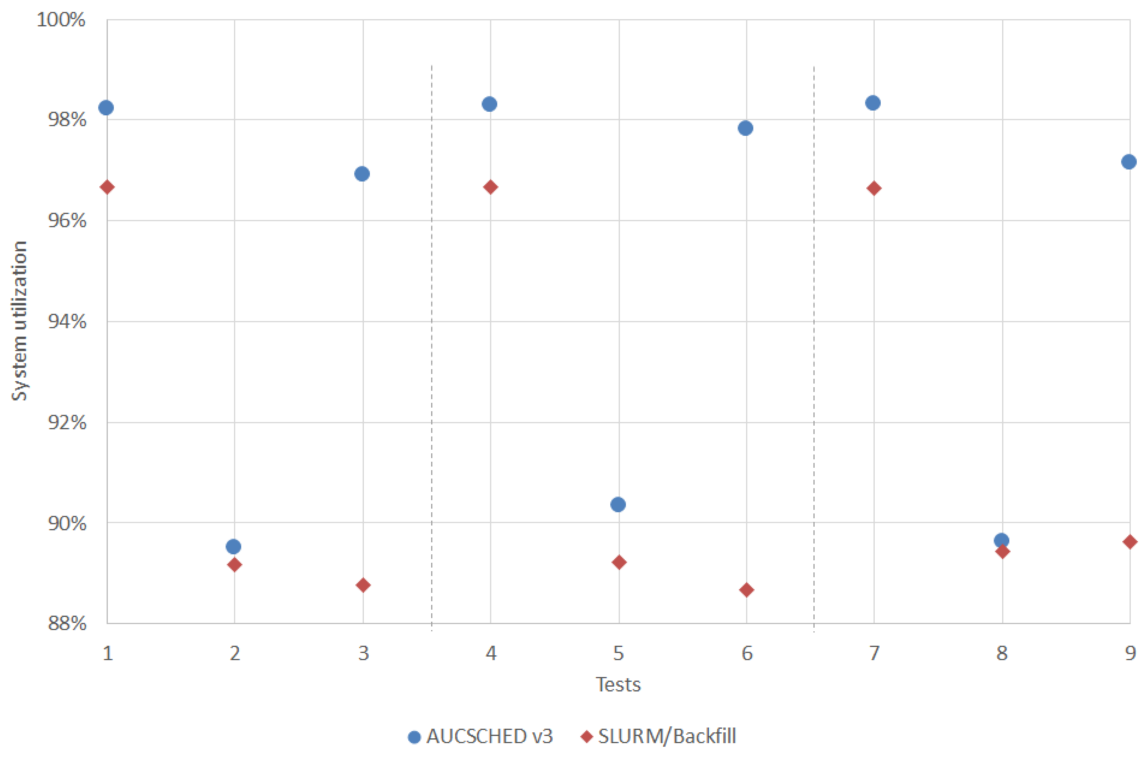


Figure 8.17. Utilization comparison between AUCSCHED3 and Slurm/Backfill for simulated TSUBAME 2.0 system with different generated workloads as described in Table 8.6.

SCHED3 compared to Slurm/Backfill. The difference in spread between different mechanisms ranges from 6% to 21%. The difference is lowest in the workloads 1, 4 and 7, this is caused by the bid generation algorithm in *AUCSCHED3*, as explained in the lowest level common switch case.

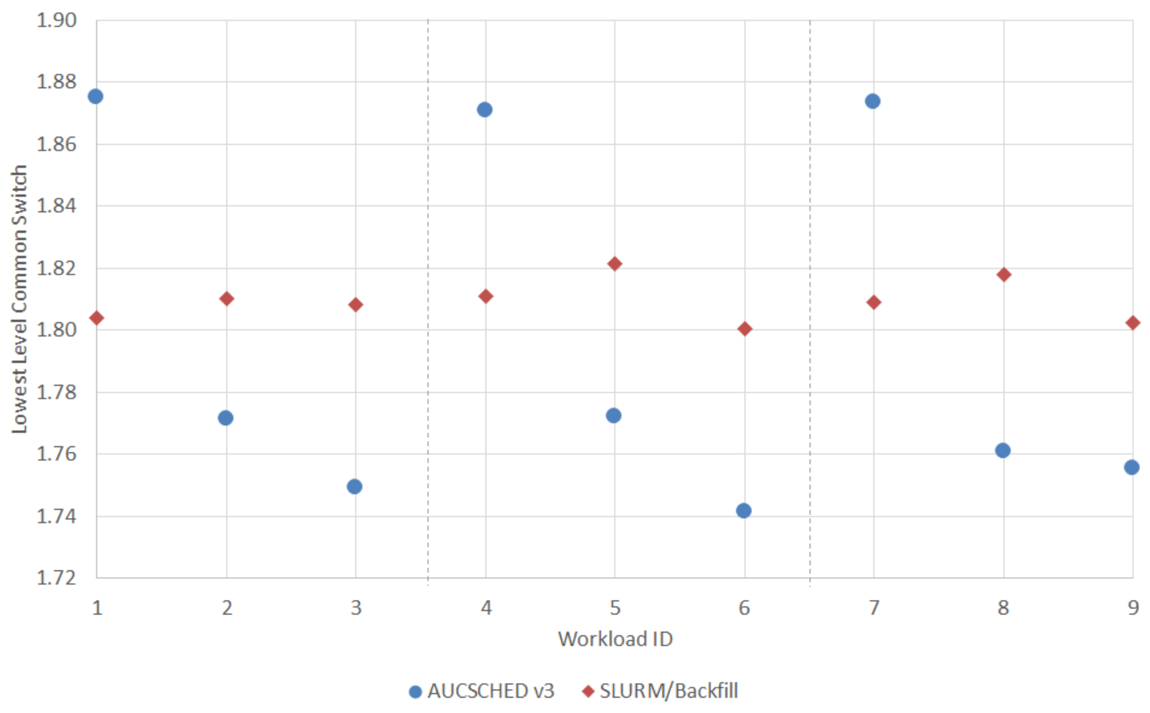


Figure 8.18. Lowest level common switch comparison between AUCSCHED3 and Slurm/Backfill for simulated TSUBAME 2.0 system with different generated workloads as described in Table 8.6.

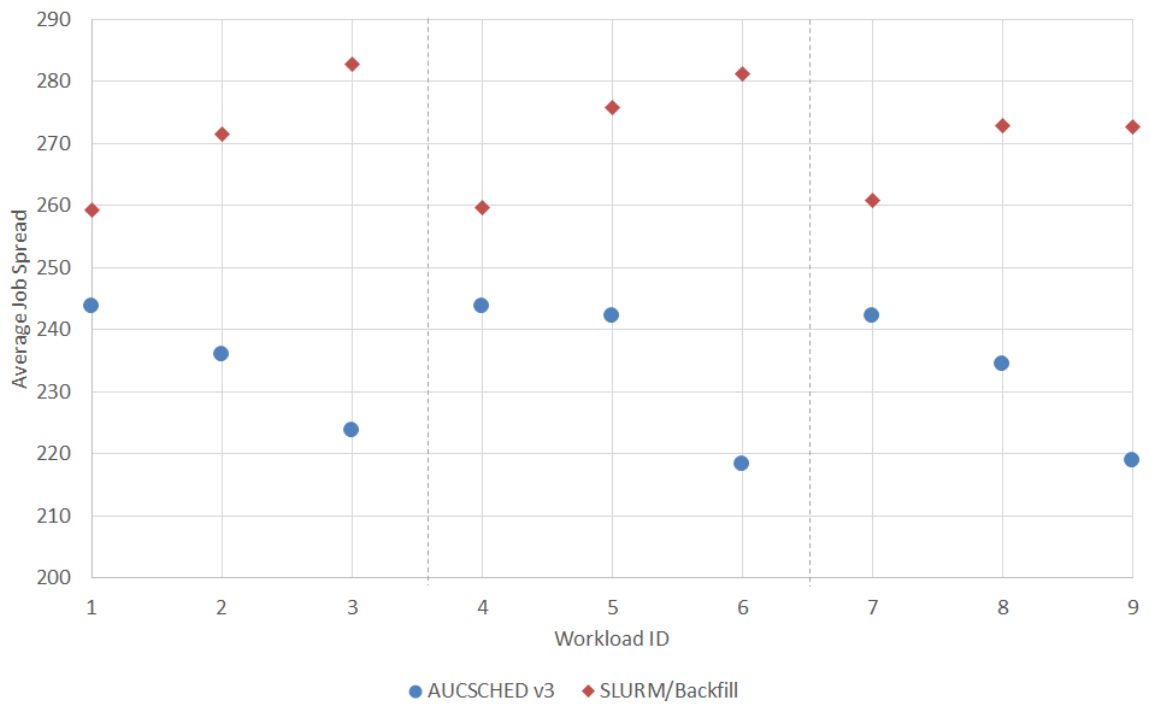


Figure 8.19. Job spread comparison between AUCSCHED3 and Slurm/Backfill for simulated TSUBAME 2.0 system with different generated workloads as described in Table 8.6.

9. CONCLUSIONS

In this thesis we present four different scheduling algorithms. Our IPSCHED, AUCSCHED1, AUCSCHED2 and AUCSCHED3 algorithms utilize IP solvers and are implemented as ready to use Slurm plug-ins. We are not aware of any integer programming based scheduler algorithms being implemented in real schedulers. Our plugins are available at `https://code.google.com/p/Slurm-ipsched/`. Implementing our algorithms as Slurm plug-ins allows us to make realistic tests by emulating a real heterogeneous supercomputer.

The auction based AUCSCHED1 job scheduler employs bids that compete in an auction for acquiring resources. We consider tightly coupled supercomputer clusters where multiple cores and GPU resources will be acquired at the same time and the auction will be repeated frequently (roughly every 5 seconds). Hence, we are required to solve a far more complex combinatorial optimization problem in a very short period of time. To the best of our knowledge, our AUCSCHED1 is probably the first auction based scheduler that has been developed for clusters. Our AUCSCHED1 scheduler performs better in terms of utilization than the currently used Backfill algorithm.

Our AUCSCHED2 scheduler makes two major contributions: (*i*) It extends Slurm to support generic resources (e.g. GPU) moldability by specification of resource ranges and (*ii*) it provides a new integer programming formulation with drastically reduced number of variables for our existing auction based scheduler. Generic resource moldability feature can be very useful to run-time auto-tuning applications that can make use of variable number of generic resources such as GPUs. It can also increase utilization of expensive supercomputer resources by providing flexibity in resource allocation. The results we obtain indeed provide evidence of increased utilizations. The new integer formulation enables us to generate higher number of bids for each job and obtain solution in a shorter period of time. When compared with the results of our previous AUCSCHED1 implementation, we see improvements in utilization, mainly as a result of being able to generate higher number of bids for each job.

In our AUCSCHED3 scheduler, we have also considered a fat-tree based system interconnection topology. AUCSCHED3 is able to generate both topologically aware better mappings of jobs and achieve higher system utilizations especially in workloads involving jobs that request both CPU and GPU resources.

It should be noted that the schedulers that have been proposed in this thesis are designed for supercomputers with high utilization rates. If the utilization rates are low, the advantages of using this scheduler cannot not be observed. It should also be noted that, the schedulers proposed will not be suitable for grids. In grids, using a distributed scheduler will be more advantageous.

9.1. Future Work

In the future, the importance of the schedulers for the heterogeneous supercomputers will keep increasing since supercomputer usage is becoming mainstream not only in academic and research environments but also in industry. Some of the possible research topics that can be conducted in the future are as follows:

- A future research topic may be on how to do topologically aware mappings for the torus architecture. One approach may be linearization of the topology of the system by using Hilbert space filling curves and making allocation decisions based on this linearized topology.
- Another possible research topic may be the addition of an accounting system which also make use of preemption. A model can be developed where each user has some *credits* in their account. If the user needs the job to be completed quickly, they submit the job with a higher credit/core-hour. If some job, currently running in the system, has lower credit/core-hours assigned to the job, this job may be preempted by the job in the queue. This model maximizes the profit for the supercomputer. Users who are struggling to meet deadlines can give more credits to have their jobs completed jobs quickly.
- Another future work area could be the development of heuristic algorithms for the models that were proposed in this thesis. One possible heuristic could be based

on relaxation of the variables in the IP, and testing the solution for feasibility.

- If one also aims to take into account reservations, the proposed models and the algorithms may be slightly altered to achieve that. One possible option is to add reservation support for the nodes at future times, and then check for them during the bid generation phase.

APPENDIX A: RESULT TABLES

Table A.1. IPSCHED, AUCSCHED1 and Backfill comparison results for ESP workload.

Experiment	**Waiting time mean (std)**	**Slowdown Ratio mean (std)**	**Utilization mean**
1	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	1.88 (1.612)	20.64 (26.52)	0.94
5	2.42 (1.758)	22.75 (22.02)	0.89
6	0.88 (1.223)	10.75 (18.20)	0.94
7	2.08 (1.577)	22.06 (21.84)	0.93
8	2.69 (1.497)	29.14 (26.93)	0.95

Table A.2. IPSCHED, AUCSCHED1 and Backfill comparison results for our workload generator.

Workload Version	**Average Theoretical Run-time**	**Experiment**	**Average Utilization**	**Average Runtime (hours)**	**Average Waiting Time mean (std)**	**Slowdown Ratio mean (std)**
I	4.00	1	95%	4.21	1.69 (1.18)	23.68 (22.33)
		2	97%	4.12	1.31 (1.20)	21.45 (27.31)
		3	97%	4.12	1.26 (1.15)	21.53 (28.05)
		4	96%	4.17	1.75 (1.15)	27.98 (29.06)
		5	96%	4.15	2.28 (1.23)	31.90 (24.23)
		6	97%	4.14	2.03 (1.23)	32.87 (33.13)
		7	97%	4.12	2.06 (1.25)	32.63 (33.00)
		8	94%	4.24	2.62 (1.22)	41.87 (32.46)

Table A.2. IPSCHED, AUCSCHED1 and Backfill comparison results for our workload generator (continued).

Workload Version	**Average Theoretical Run-time**	**Experiment**	**Average Utilization**	**Average Runtime (hours)**	**Average Waiting Time mean (std)**	**Slowdown Ratio mean (std)**
II	4.06	1	95%	4.27	1.67 (1.16	23.60 (22.51)
		2	96%	4.24	1.35 (1.22)	22.15 (28.11)
		3	96%	4.23	1.33 (1.18)	22.16 (28.15)
		4	96%	4.22	1.75 (1.17)	27.99 (32.46)
		5	96%	4.23	2.22 (1.23)	30.91 (24.01)
		6	97%	4.19	2.00 (1.22)	32.28 (32.47)
		7	97%	4.19	2.09 (1.25)	33.41 (34.00)
		8	94%	4.31	2.60 (1.25)	35.84 (32.46)

Table A.2. IPSCHED, AUCSCHED1 and Backfill comparison results for our workload generator (continued).

Workload Version	**Average Theoretical Run-time**	**Experiment**	**Average Utilization**	**Average Runtime (hours)**	**Average Waiting Time mean (std)**	**Slowdown Ratio mean (std)**
III	3.96	1	95%	4.16	1.74 (1.19)	24.33 (23.65)
		2	96%	4.12	1.41 (1.24)	22.95 (29.20)
		3	96%	4.11	1.34 (1.22)	22.01 (28.56)
		4	95%	4.17	1.72 (1.15)	27.38 (32.46)
		5	96%	4.13	2.14 (1.26)	28.44 (20.42)
		6	97%	4.09	1.87 (1.14)	28.27 (28.94)
		7	96%	4.08	1.77 (1.15)	27.65 (28.19)
		8	96%	4.12	2.31 (1.16)	36.07 (32.46)

Table A.2. IPSCHED, AUCSCHED1 and Backfill comparison results for our workload generator (continued).

Workload Version	**Average Theoretical Run-time**	**Experiment**	**Average Utilization**	**Average Runtime (hours)**	**Average Waiting Time mean (std)**	**Slowdown Ratio mean (std)**
IV	4.01	1	88%	4.56	1.68 (1.18)	22.42 (18.89)
		2	90%	4.46	1.51 (1.23)	21.87 (17.96)
		3	89%	4.48	1.43 (1.17)	21.54 (17.41)
		4	95%	4.22	1.72 (1.15)	27.23 (32.46)
		5	88%	4.57	1.95 (1.26)	22.86 (12.41)
		6	88%	4.54	2.03 (1.23)	25.17 (19.44)
		7	88%	4.54	1.93 (1.31)	24.81 (22.27)
		8	93%	4.31	2.50 (1.27)	39.53 (32.46)

Table A.2. IPSCHED, AUCSCHED1 and Backfill comparison results for our workload generator (continued).

Workload Version	**Average Theoretical Run-time**	**Experiment**	**Average Utilization**	**Average Runtime (hours)**	**Average Waiting Time mean (std)**	**Slowdown Ratio mean (std)**
V	3.95	1	62%	6.37	1.73 (1.39)	21.36 (18.20)
		2	62%	6.36	1.71 (1.31)	21.21 (17.65)
		3	62%	6.41	1.67 (1.28)	20.98 (16.98)
		4	65%	6.11	1.93 (1.40)	30.84 (32.46)
		5	68%	5.81	2.21 (1.54)	26.30 (18.72)
		6	58%	6.81	2.12 (1.36)	44.71 (34.21)
		7	60%	6.58	2.13 (1.38)	40.19 (30.17)
		8	70%	5.65	3.72 (1.64)	58.57 (32.46)

Table A.3. Results for AUCSCHED2, AUCSCHED1 and Backfill for the emulated Tsubame 2.5 system.

WLID	**Method**	**Utilization (%)**	**Average Run-Time (hours)**	**Waiting Time mean (std)**	**Fragmentation mean (std)**	**Spread mean (std)**
1	AUCSCHED2	97.17	6.45	2.42 (1.37)	17.80 (13.86)	8.2 (11.45)
	AUCSCHED1	91.01	6.90	2.45 (1.39)	2.25 (1.98)	11.87 (18.02)
	Backfill	87.17	7.18	3.23 (1.32)	2.42 (2.33)	17.36 (35.97)
2	AUCSCHED2	93.83	6.70	2.51 (1.55)	5.58 (7.35)	4.47 (7.66)
	AUCSCHED1	87.81	7.14	3.09 (1.52)	1.98 (1.79)	6.75 (9.16)
	Backfill	86.00	7.27	2.96 (1.43)	2.12 (2.18)	11.68 (26.00)
3	AUCSCHED2	92.17	6.77	2.71 (1.55)	1.00 (0.00)	1.00 (0.00)
	AUCSCHED1	86.70	7.22	2.78 (1.60)	1.00 (0.00)	1.00 (0.00)
	Backfill	82.50	7.58	2.98 (1.68)	1.00 (0.00)	1.00 (0.00)
4	AUCSCHED2	98.00	12.83	4.52 (2.54)	18.79 (14.17)	8.31 (11.61)
	AUCSCHED1	92.23	13.59	4.73 (2.60)	2.13 (2.09)	12.45 (23.54)
	Backfill	88.00	14.29	4.91 (2.71)	2.42 (2.34)	17.24 (35.32)

Table A.3. Results for AUCSCHED2, AUCSCHED1 and Backfill for the emulated Tsubame 2.5 system.

WLID	**Method**	**Utilization (%)**	**Average Run-Time (hours)**	**Waiting Time mean (std)**	**Fragmentation mean (std)**	**Spread mean (std)**
5	AUCSCHED2	94.00	13.33	4.27 (2.19)	5.66 (7.41)	4.42 (7.70)
	AUCSCHED1	89.87	13.95	4.63 (2.43)	2.07 (2.10)	8.79 (15.45)
	Backfill	86.83	14.48	4.94 (2.68)	2.14 (2.23)	11.9 (27.07)
6	AUCSCHED2	92.67	13.51	4.79 (2.72)	1.00 (0.00)	1.00 (0.00)
	AUCSCHED1	88.47	14.11	5.09 (2.79)	1.00 (0.00)	1.00 (0.00)
	Backfill	83.00	15.09	5.43 (2.82)	1.00 (0.00)	1.00 (0.00)
7	AUCSCHED2	86.33	6.28	2.60 (1.73)	2.16 (1.63)	1.98 (3.19)
	AUCSCHED1	82.73	6.56	2.27 (1.76)	2.07 (1.13)	2.13 (1.52)
	Backfill	81.17	6.69	1.94 (1.83)	2.10 (1.39)	4.16 (10.07)
8	AUCSCHED2	84.00	6.46	2.60 (1.75)	1.52 (1.11)	1.52 (2.10)
	AUCSCHED1	81.35	6.67	2.34 (1.70)	1.3 (0.85)	1.70 (1.08)
	Backfill	79.00	6.87	1.95 (1.86)	1.55 (1.10)	2.35 (7.24)

Table A.3. Results for AUCSCHED2, AUCSCHED1 and Backfill for the emulated Tsubame 2.5 system.

WLID	**Method**	**Utilization (%)**	**Average Run-Time (hours)**	**Waiting Time mean (std)**	**Fragmentation mean (std)**	**Spread mean (std)**
9	AUCSCHED2	84.00	6.55	2.72 (1.82)	1.00 (0.00)	1.00 (0.00)
	AUCSCHED1	79.85	6.89	2.50 (1.89)	1.00 (0.00)	1.00 (0.00)
	Backfill	76.83	7.16	1.98 (1.97	1.00 (0.00)	1.00 (0.00)
10	AUCSCHED2	87.00	12.50	5.63 (3.52)	2.16 (1.65)	1.93 (3.00)
	AUCSCHED1	82.45	13.20	5.13 (3.58)	2.05 (1.29)	2.13 (2.99)
	Backfill	81.67	13.32	4.05 (3.60)	2.09 (1.34)	3.88 (10.76)
11	AUCSCHED2	85.33	12.44	5.52 (3.48)	1.5 (1.09)	1.45 (1.79)
	AUCSCHED1	81.24	13.08	4.29 (3.48)	1.52 (1.11)	1.74 (1.89)
	Backfill	80.33	13.25	3.92 (3.58)	1.61 (1.14)	2.50 (7.06)
12	AUCSCHED2	82.83	13.16	5.88 (3.63)	1.00 (0.00)	1.00 (0.00)
	AUCSCHED1	80.01	13.62	5.27 (3.67)	1.00 (0.00)	1.00 (0.00)
	Backfill	78.83	13.85	4.08 (3.76)	1.00 (0.00)	1.00 (0.00)

Table A.4. AUCSCHED3 and Backfill comparison for an emulated 1024 node hypothetical supercomputer.

Workload ID	**Plugin**	**Lowest Level Common Switch mean (std)**	**Average Spread**	**Utilization (%)**
1	AUCSCHED3	1.05 (0.68)	77.2	90
	Slurm/Backfill	1.06 (0.77)	110.7	89
2	AUCSCHED3	1.17 (0.62)	90.9	96
	Slurm/Backfill	1.19 (0.69)	131.9	95
3	AUCSCHED3	1.04 (0.58)	65.4	78
	Slurm/Backfill	1.26 (0.68)	139.4	82
4	AUCSCHED3	1.04 (0.58)	61.1	85
	Slurm/Backfill	1.36 (0.69)	167.8	88
5	AUCSCHED3	1.01 (0.67)	63.5	83
	Slurm/Backfill	1.24 (0.72)	144.3	81
6	AUCSCHED3	0.98 (0.56)	60.9	89
	Slurm/Backfill	1.33 (0.70)	162.7	87
5’	AUCSCHED3	0.99 (0.65)	64.4	86
	Slurm/Backfill	1.24 (0.72)	144.3	81
6’	AUCSCHED3	1.00 (0.59)	61.4	91
	Slurm/Backfill	1.33 (0.70)	162.7	87

Table A.5. AUCSCHED3 and Backfill comparison for an emulated TSUBAME 2.0 supercomputer using workloads in Table 8.6.

Workload ID	**Plugin**	**Lowest level common switch mean (std)**	**Average spread**	**Utilization (%)**
1	AUCSCHED3	1.88 (0.41)	243.8	98
	Slurm/Backfill	1.8 (0.53)	259.3	97
2	AUCSCHED3	1.77 (0.53)	236	90
	Slurm/Backfill	1.81 (0.52)	271.5	89
3	AUCSCHED3	1.75 (0.55)	223.7	97
	Slurm/Backfill	1.81 (0.53)	282.7	89
4	AUCSCHED3	1.87 (0.42)	243.7	98
	Slurm/Backfill	1.81 (0.52)	259.7	97
5	AUCSCHED3	1.77 (0.52)	242.2	90
	Slurm/Backfill	1.82 (0.50)	275.8	89
6	AUCSCHED3	1.74 (0.56)	218.4	98
	Slurm/Backfill	1.8 (0.54)	281.3	89
7	AUCSCHED3	1.87 (0.42)	242.1	98
	Slurm/Backfill	1.81 (0.52)	260.8	97
8	AUCSCHED3	1.76 (0.54)	234.4	90
	Slurm/Backfill	1.82 (0.51)	272.8	89
9	AUCSCHED3	1.76 (0.54)	218.9	97
	Slurm/Backfill	1.8 (0.53)	272.7	90

REFERENCES

1. Segall, R., *Research and Applications in Global Supercomputing*, Advances in Systems Analysis, Software Engineering, and High Performance Computing.

2. Cray Research Inc., *The CRAY-1 Computer System*, 1976, `http://web.archive.org/web/20150801213906/http://archive.computerhistory.org/resources/text/Cray/Cray.Cray1.1977.102638650.pdf`, August 2015.

3. Hickey, H., "Bringing Supercomputers to Life (Sciences)", *Biomedical Computation Review*, Vol. 6, pp. 7–15.

4. Anderson, D. P., J. Cobb, E. Korpela, M. Lebofsky and D. Werthimer, "SETI@Home: An Experiment in Public-resource Computing", *Communications of the ACM*, Vol. 45, No. 11, pp. 56–61, Nov. 2002.

5. "Short Inverted Repeats Are Hotspots for Genetic Instability: Relevance to Cancer Genomes", *Cell Reports*, Vol. 10, No. 10, pp. 1674 – 1680, 2015.

6. Salazar, J., *Supercomputers Surprisingly Link DNA Crosses to Cancer*, 2015, `http://web.archive.org/web/20150801213658/https://www.tacc.utexas.edu/-/supercomputers-surprisingly-link-dna-crosses-to-cancer`, August 2015.

7. Gupta, S., *China's Investment in GPU Supercomputing Begins To Pay Off Big Time!*, 2011, `http://web.archive.org/web/20150801213732/http://blogs.nvidia.com/blog/2011/06/09/chinas-investment-in-gpu-supercomputing-begins-to-pay-off-big-time/`, August 2015.

8. IBM Corporation, *IBM 1401 Data Processing System*, 1961, `http://web.archive.org/web/20150801213801/http://ed-thelen.org/`

comp-hist/BRL61-ibm1401.html, August 2015.

9. Warren, M., J. Salmon, D. Becker, M. Goda, T. Sterling and W. Winckelmans, “Pentium Pro Inside: I. A Treecode at 430 Gigaflops on ASCI Red, II. Price/Performance of $50/Mflop on Loki and Hyglac”, *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pp. 61–61, Nov 1997.

10. Angelini, C. and I. Wallossek, *Radeon R9 295X2 8 GB Review: Project Hydra Gets Liquid Cooling*, 2014, http://www.tomshardware.com/reviews/radeon-r9-295x2-review-benchmark-performance,3799.html, August 2015.

11. *Top 500 Supercomputer Sites - June 1997*, 1997, http://web.archive.org/web/20150801213521/http://www.top500.org/lists/1997/06/, August 2015.

12. Barker, K. J., K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin and J. C. Sancho, “Entering The PetaFLOP Era: The Architecture and Performance of Roadrunner”, *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, p. 1, IEEE Press, 2008.

13. *Top 500 Supercomputer Sites - June 2008*, 1997, http://web.archive.org/web/20150801213640/http://www.top500.org/lists/2008/06/, August 2015.

14. Alba, D., *China’s Tianhe-2 Caps Top 10 Supercomputers*, 2013, http://web.archive.org/web/20150801214058/http://spectrum.ieee.org/tech-talk/computing/hardware/tianhe2-caps-top-10-supercomputers, August 2015.

15. Dongarra, J. J., P. Luszczek and A. Petitet, “The LINPACK benchmark: Past, present, and future”, *Concurrency and Computation: Practice and Experience*, Vol. 15, pp. 803–820, 2003.

16. *Top 500 Supercomputer Sites - November 2014*, 2014, http://web.archive.org/web/20150801214749/http://www.top500.org/lists/2014/11/, August 2015.

17. Ashby, S., P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina *et al.*, "The Opportunities and Challenges of Exascale Computing", *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee*, pp. 1–77, 2010.

18. Yokokawa, M., F. Shoji, A. Uno, M. Kurokawa and T. Watanabe, "The K computer: Japanese Next-generation Supercomputer Development Project", *Proceedings of the 17th IEEE/ACM International Symposium on Low-Power Electronics and Design*, pp. 371–372, 2011.

19. *TSUBAME 2.0 Hardware and Software Specifications*, 2011, `http://www.gsic.titech.ac.jp/sites/default/files/TSUBAME_SPECIFICATIONS_en_0.pdf`, August 2015.

20. InfiniBand Trade Association, *InfiniBand Architecture Specification: Release 1.0*, 2000.

21. Gropp, W., E. Lusk, N. Doss and A. Skjellum, "A High-performance, Portable Implementation of the MPI Message Passing Interface Standard", *Parallel computing*, Vol. 22, No. 6, pp. 789–828, 1996.

22. *Top 500 Supercomputer Sites - November 2015*, 2015, `http://web.archive.org/web/20151215233700/http://www.top500.org/lists/2015/11/`, [December 2015.

23. Georgiou, Y., *Contributions For Resource and Job Management in High Performance Computing*, Ph.D. Thesis, Universite de Grenoble, France, 2010.

24. Dongarra, J. and P. Luszczek, "LINPACK Benchmark", *Encyclopedia of Parallel Computing*, pp. 1033–1036, 2011.

25. *ORNL Debuts Titan Supercomputer*, 2012, `http://web.archive.org/web/20150801214857/http://www.ornl.gov/ornl/news/news-releases?`

ReleaseNumber=mr20121029-00, August 2015.

26. Wait, P., *IBM's Sequoia is World's Fastest Supercomputer*, 2012, http://www.informationweek.com/software/information-management/ibms-sequoia-is-worlds-fastest-supercomputer/d/d-id/1104906?, January 2016.

27. Chen, D., N. Eisley, P. Heidelberger, S. Kumar, A. Mamidala, F. Petrini, R. Senger, Y. Sugawara, R. Walkup, B. Steinmacher-Burow *et al.*, "Looking Under the Hood of the IBM Blue Gene/Q Network", *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 69, 2012.

28. CSCS, *Piz Daint*, 2013, http://web.archive.org/web/20150801212737/http://www.cscs.ch/computers/piz_daint/index.html, August 2015.

29. *Texas Advanced Computing Center Stampede*, 2013, http://web.archive.org/web/20150801212737/https://www.tacc.utexas.edu/stampede/, August 2015.

30. NVIDIA, *What is GPU computing? GPGPU, CUDA and Kepler explained*, 2015, http://web.archive.org/web/20150801212315/http://www.nvidia.com/object/what-is-gpu-computing.html, August 2015.

31. Jeffers, J. and J. Reinders, *Intel Xeon Phi Coprocessor High-Performance Programming*, Elsevier Science, 2013.

32. The Register, *High Five From AMD: New Supercomputer GPU Maxes Out at 5.07 TFLOPS*, 2014, http://web.archive.org/web/20150801212235/http://www.theregister.co.uk/2014/08/06/amd_s9150_s9050_gpus/, August 2015.

33. NVIDIA, *NVIDIA Tesla K40*, 2014, http://web.archive.org/web/20150801212301/http://www.nvidia.com/content/tesla/pdf/

nvidia-tesla-k40-2014mar-lr.pdf, August 2015.

34. Intel, *Intel Xeon Phi Coprocessor 7120A (16GB, 1.238 GHz, 61 core)*, 2014, http://web.archive.org/web/20150801212210/http://ark.intel.com/products/80555/Intel-Xeon-Phi-Coprocessor-7120A-16GB-1_238-GHz-61-core, August 2015.

35. Sharma, S., C.-H. Hsu and W. chun Feng, “Making a Case For a Green500 List”, *20th International Parallel and Distributed Processing Symposium*, April 2006.

36. *The Green500 List - November 2015*, 2015, http://web.archive.org/web/20151215233551/http://www.green500.org/lists/green201511&green500from=1&green500to=100, December 2015.

37. Yoo, A., M. Jette and M. Grondona, “SLURM: Simple Linux Utility for Resource Management”, *Job Scheduling Strategies for Parallel Processing*, Vol. 2862 of *Lecture Notes in Computer Science*, pp. 44–60, Springer Berlin Heidelberg, 2003.

38. Georgiou, Y., *Contributions For Resource and Job Management in High Performance Computing*, Ph.D. Thesis, Universite de Grenoble, France, 2010.

39. Nitzberg, B., J. Schopf and J. Jones, “PBS Pro: Grid Computing and Scheduling Attributes”, *Grid Resource Management*, Vol. 64 of *International Series in Operations Research & Management Science*, pp. 183–190, Springer US, 2003.

40. Moab, *Workload Manager Documentation*, 2012, http://docs.adaptivecomputing.com/, December 2015.

41. Adaptive Computing, *Torque Resource Manager Documentation*, 2012, http://web.archive.org/web/20160112185238/http://docs.adaptivecomputing.com/, January 2016.

42. Bode, B., D. M. Halstead, R. Kendall, Z. Lei and D. Jackson, “The Portable

Batch Scheduler and The Maui Scheduler on Linux Clusters", *Proceedings of the 4th Annual Linux Showcase & Conference, ALS '00*, 2000.

43. Gentzsch, W., "Sun Grid Engine: Towards Creating a Compute Power Grid", *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, pp. 35 –36, 2001.

44. *Open Grid Scheduler/Grid Engine*, 2012, `http://web.archive.org/web/20150801215102/http://gridscheduler.sourceforge.net/`, August 2015.

45. Slurm, *Slurm Quick Start User Guide*, 2014, `http://web.archive.org/web/20150801211715/http://Slurm.schedmd.com/quickstart.html`, August 2015.

46. Garey, M. R. and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA, 1979.

47. Schwiegelshohn, U., "How to Design a Job Scheduling Algorithm", W. Cirne and N. Desai (Editors), *Job Scheduling Strategies for Parallel Processing*, Vol. 8828 of *Lecture Notes in Computer Science*, pp. 147–167, Springer International Publishing, 2015.

48. Skovira, J., W. Chan, H. Zhou and D. Lifka, "The EASY — LoadLeveler API project", D. Feitelson and L. Rudolph (Editors), *Job Scheduling Strategies for Parallel Processing*, Vol. 1162 of *Lecture Notes in Computer Science*, pp. 41–47, Springer Berlin Heidelberg, 1996.

49. Tsafrir, D., *Modeling, Evaluating, and Improving the Performance of Supercomputer Scheduling*, Ph.D. Thesis, School of Computer Science and Engineering, the Hebrew University, Jerusalem, Israel, September 2006.

50. Lee, G., *Resource Allocation and Scheduling in Heterogeneous Cloud Environments*, Ph.D. Thesis, Electrical Engineering and Computer Sciences, University of California at Berkley, California, USA, 2012.

51. Corbet, J., *Notes From a Container*, 2007, `http://web.archive.org/web/20150801212113/https://lwn.net/Articles/256389/`, August 2015.

52. Lameter, C., "NUMA (Non-Uniform Memory Access): An Overview", *Queue*, Vol. 11, No. 7, pp. 40–51, Jul. 2013.

53. Li, Y., Y. Liu and D. Qian, "A Heuristic Energy-aware Scheduling Algorithm for Heterogeneous Clusters", *15th International Conference on Parallel and Distributed Systems (ICPADS 2009)*, pp. 407–413, IEEE, 2009.

54. Barbosa, J. and B. Moreira, "Dynamic Job Scheduling on Heterogeneous Clusters", *8th International Symposium on Parallel and Distributed Computing*, pp. 3–10, 2009.

55. Iserte, S., A. Castello, R. Mayo, E. Quintana-Orti, F. Silla, J. Duato, C. Reano and J. Prades, "SLURM Support for Remote GPU Virtualization: Implementation and Performance Study", *IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 318–325, Oct 2014.

56. Klein-Halmaghi, C., *Cooperative Resource Management for Parallel and Distributed Systems*, Ph.D. Thesis, Université de Lyon, Lyon, France, 2012.

57. Ayguadé, E., R. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. Gonzàlez, F. Igual, D. Jiménez-González, J. Labarta, L. Martinell, X. Martorell, R. Mayo, J. Pérez, J. Planas and E. Quintana-Ortí, "Extending OpenMP to Survive the Heterogeneous Multi-Core Era", *International Journal of Parallel Programming*, Vol. 38, pp. 440–459, 2010.

58. Planas, J., R. Badia, E. Ayguade and J. Labarta, "Self-Adaptive OmpSs Tasks in Heterogeneous Environments", *IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS 2013)*, pp. 138–149, May 2013.

59. Qasem, A., *Automatic Tuning of Scientific Applications*, Ph.D. Thesis, Rice University, Houston, TX, USA, 2008.

60. Bokhari, S. H., “On the Mapping Problem”, *IEEE Transactions on Computers*, Vol. 100, No. 3, pp. 207–214, 1981.

61. Lee, S.-Y. and J. Aggarwal, “A mapping strategy for parallel processing”, *IEEE Transactions on Computers*, Vol. 100, No. 4, pp. 433–442, 1987.

62. Kirkpatrick, S., C. D. Gelatt, M. P. Vecchi *et al.*, “Optimization by Simulated Annealing”, *Science*, Vol. 220, No. 4598, pp. 671–680, 1983.

63. Bäck, T., *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*, Oxford University Press, 1996.

64. Bhatele, A., *Automating Topology Aware Mapping for Supercomputers*, Ph.D. Thesis, Dept. of Computer Science, University of Illinois, USA, 2010.

65. Pascual, J. A., J. Navaridas and J. Miguel-Alonso, “Job Scheduling Strategies for Parallel Processing”, chap. Effects of Topology-Aware Allocation Policies on Scheduling Performance, pp. 138–156, Springer-Verlag, Berlin, Heidelberg, 2009.

66. Smith, C., B. McMillan and I. Lumb, *Topology Aware Scheduling in the LSF Distributed Resource Manager*, 2001, `http://web.archive.org/web/20160121225141/https://cug.org/5-publications/proceedings_attendee_lists/2001CD/S01_Proceedings/Pages/Authors/Smith_C/Smith.htm`, January 2016.

67. Subramoni, H., D. Bureddy, K. Kandalla, K. Schulz, B. Barth, J. Perkins, M. Arnold and D. K. Panda, “Design of Network Topology Aware Scheduling Services for Large InfiniBand Clusters”, *IEEE International Conference on Cluster Computing*, pp. 1–8, 2013.

68. Leiserson, C. E., “Fat-trees: Universal Networks for Hardware-Efficient Supercomputing”, *IEEE Transactions on Computers*, Vol. 100, No. 10, pp. 892–901, 1985.

69. *TOP 500 Supercomputer Sites for November 2012*, 2012, `http://web.archive.org/web/20150801215018/http://www.top500.org/lists/2012/11/`, August 2015.

70. Koomey, J. G., *Estimating Total Power Consumption by Servers in the US and the World*, Vol. 15, February, 2007, `http://energy.lbl.gov/EA/mills/HT/documents/data_centers/svrpwrusecompletefinal.pdf`, January 2016.

71. Hagimont, D., L. Broto, A. Gadafi and N. D. Palma, “Experience with Autonomic Energy Management Policies for JavaEE Clusters.”, *Handbook of Energy-Aware and Green Computing*, pp. 855–872, 2012.

72. Aupy, G., A. Benoit, F. Dufossé and Y. Robert, “Reclaiming the Energy of a Schedule: Models and Algorithms”, *Concurrency and Computation: Practice and Experience*, Vol. 25, No. 11, pp. 1505–1523, 2013.

73. Etinski, M., J. Corbalan, J. Labarta and M. Valero, “Linear Programming Based Parallel Job Scheduling for Power Constrained Systems”, *International Conference on High Performance Computing and Simulation (HPCS)*, pp. 72 –80, July 2011.

74. Ahuja, R. K., T. L. Magnanti and J. B. Orlin, *Network Flows*, Prentice Hall, 1988.

75. Winston, W. L. and J. B. Goldberg, *Operations Research: Applications and Algorithms*, Vol. 3, Duxbury Press, Boston, MA, USA, 2004.

76. Lawler, E., *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, New York, NY, USA, 1985.

77. Murty, K. G., *Linear Programming*, Wiley, New York, NY, USA, 1983.

78. Shih, W., "A Branch and Bound Method for the Multiconstraint Zero-One Knapsack Problem", *Journal of the Operational Research Society*, Vol. 30, pp. 369–378, 1979.

79. Land, A. H. and A. G. Doig, "An Automatic Method of Solving Discrete Programming Problems", *Econometrica*, Vol. 28, No. 3, pp. 497–520, 1960.

80. Meindl, B. and M. Templ, "Analysis of Commercial and Free and Open Source Solvers for the Cell Suppression Problem", *Transactions on Data Privacy*, Vol. 6, No. 2, pp. 147–159, 2013.

81. "IBM ILOG CPLEX Optimizer", `http://web.archive.org/web/20150801215704/http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html`, 2012.

82. Bixby, R. E., "A Brief History of Linear and Mixed-Integer Programming Computation", *Documenta Mathematica*, pp. 107–121, 2012.

83. IBM, *IBM ILOG CPLEX Optimizer Performance Benchmarks*, 2014, `http://web.archive.org/web/20150801211733/http://www-01.ibm.com/software/commerce/optimization/cplex-performance/`, August 2015.

84. Erbas, O. and C. Ozturan, "Collective Match-Making Heuristics for Grid Resource Scheduling", *High Performance Grid Middleware, Brasov, Romania*, September 2007.

85. Shen, S., K. Deng, A. Iosup and D. Epema, "Scheduling Jobs in the Cloud Using On-Demand and Reserved Instances", F. Wolf, B. Mohr and D. an Mey (Editors), *Euro-Par 2013 Parallel Processing*, Vol. 8097 of *Lecture Notes in Computer Science*, pp. 242–254, Springer Berlin Heidelberg, 2013.

86. Blanco, H., F. Guirado, J. Lérida and V. Albornoz, "MIP Model Scheduling for Multi-Clusters", *Euro-Par 2012: Parallel Processing Workshops*, Vol. 7640 of *Lecture Notes in Computer Science*, pp. 196–206, 2013.

87. Curino, C., D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan and S. Rao, "Reservation-based Scheduling: If You're Late Don't Blame Us!", *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–14, 2014.

88. Sun, Q., Q. Zhuge, J. Hu, J. Yi and E. H.-M. Sha, "Efficient Grouping-Based Mapping and Scheduling on Heterogeneous Cluster Architectures", *Computers & Electrical Engineering*, 2014.

89. Zhang, Q., E. Gürses, R. Boutaba and J. Xiao, "Dynamic Resource Allocation For Spot Markets in Clouds", *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE'11)*, 2011.

90. Xu, H. and B. Li, "Maximizing Revenue with Dynamic Cloud Pricing: The Infinite Horizon Case", *IEEE International Conference on Communications (ICC 2012)*, pp. 2929–2933, June 2012.

91. Anandasivam, A. and M. Premm, "Bid Price Control and Dynamic Pricing In Clouds", *Proceedings of the European Conference on Information Systems (ECIS 2009)*, pp. 1077–1089, 2009.

92. Wolski, R., J. Plank, T. Bryan and J. Brevik, "G-commerce: Market Formulations Controlling Resource Allocation on the Computational Grid", *15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, 2001.

93. Sandholm, T., K. Lai, J. Ortiz and J. Odeberg, "Market-Based Resource Allocation using Price Prediction in a High Performance Computing Grid for Scientific Applications", *15th IEEE International Symposium on High Performance Distributed Computing (HPDC 2006)*, pp. 132–143, 2006.

94. Özer, A. H. and C. Özturan, “A Model and Heuristic Algorithms for Multi-Unit Nondiscriminatory Combinatorial Auction”, *Computers & Operations Research*, Vol. 36, No. 1, pp. 196–208, 2009.

95. Özturan, C., “Resource Bartering in Data Grids”, *Scientific Programming*, Vol. 12, No. 3, pp. 155–168, 2004.

96. Harchol-Balter, M., “Auction-Based Scheduling for the TeraGrid”, , 2007, `http://web.archive.org/web/20150801211555/http://www.cs.cmu.edu/~harchol/teragrid.pdf`, August 2015.

97. Tan, Z., *Market-Based Grid Resource Allocation Using a Stable Continuous Double Auction*, Ph.D. Thesis, Faculty of Engineering and Physical Sciences, University of Manchester, Manchester, United Kingdom, 2007.

98. Soner, S. and C. Özturan, “Integer Programming Based Heterogeneous CPU-GPU Cluster Schedulers for SLURM Resource Manager”, *Journal of Computer and System Sciences*, Vol. 81, No. 1, pp. 38–56, 2014.

99. Slurm, *Slurm Documentation*, 2012, `http://web.archive.org/web/20150801214353/http://slurm.schedmd.com/documentation.html`, August 2015.

100. Soner, S. and C. Ozturan, “A New Auction-Based Scheduler for Heterogeneous Systems with Moldable Generic Resources Support”, *Concurrency and Computation: Practice and Experience*, 2015.

101. Slurm, *Slurm Topology Guide*, 2014, `http://web.archive.org/web/20150801211652/http://slurm.schedmd.com/topology.html`, August 2015.

102. *Slurm Used on the Fastest of the TOP500 Supercomputers*, 2012, `http://web.archive.org/web/20150801213418/http://slurm.net/2012/11/21/slurm-used-on-the-fastest-of-the-top500-supercomputers/`,

August 2015.

103. "Slurm User Group Meeting in Supercomputing 2013", http://web.archive.org/web/20150801213438/http://slurm.schedmd.com/SC13_BOF/SC13_BOF_SchedMD.pdf.

104. Hussain, H., S. U. R. Malik, A. Hameed, S. U. Khan, G. Bickler, N. Min-Allah, M. B. Qureshi, L. Zhang, W. Yongji, N. Ghani, J. Kolodziej, A. Y. Zomaya, C.-Z. Xu, P. Balaji, A. Vishnu, F. Pinel, J. E. Pecero, D. Kliazovich, P. Bouvry, H. Li, L. Wang, D. Chen and A. Rayes, "A Survey on Resource Allocation in High Performance Distributed Computing Systems", *Parallel Computing*, Vol. 39, No. 11, pp. 709 – 736, 2013.

105. Lucero, A., "Simulation of Batch Scheduling Using Real Production-Ready Software Tools", *Proceedings of the 5th IBERGRID*, 2011, https://www.bsc.es/media/4856.pdf.

106. Feitelson, D., *Parallel Workloads Archive*, 2005, http://web.archive.org/web/20150801212505/http://www.cs.huji.ac.il/labs/parallel/workload/, August 2015.

107. Feitelson, D. G., "Metric and Workload Effects on Computer Systems Evaluation", *Computer*, Vol. 36, No. 9, pp. 18–25, 2003.

108. Wong, A., L. Oliker, W. Kramer, T. Kaltz and D. Bailey, "ESP: A System Utilization Benchmark", *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, p. 15, nov. 2000.

109. *SLURM Priority Plugin API, Slurm Documentation*, 2012, http://web.archive.org/web/20150801220158/http://slurm.schedmd.com/priority_plugins.html, August 2015.

110. *Multifactor Priority Plugin, Slurm Documentation*, 2012, http://web.

archive.org/web/20150801220033/http://slurm.schedmd.com/priority_multifactor.html, August 2015.