Communication-aware Job Scheduling using SLURM

Priya Mishra, Tushar Agrawal, Preeti Malakar
Indian Institute of Technology Kanpur
{priyamis,tusharag,pmalakar}@iitk.ac.in

ABSTRACT

Job schedulers play an important role in selecting optimal resources for the submitted jobs. However, most of the current job schedulers do not consider job-specific characteristics such as communication patterns during resource allocation. This often leads to sub-optimal node allocations. We propose three node allocation algorithms that consider the job's communication behavior to improve the performance of communication-intensive jobs. We develop our algorithms for tree-based network topologies. The proposed algorithms aim at minimizing network contention by allocating nodes on the least contended switches. We also show that allocating nodes in powers of two leads to a decrease in inter-switch communication for MPI communications, which further improves performance. We implement and evaluate our algorithms using SLURM, a widely-used and well-known job scheduler. We show that the proposed algorithms can reduce the execution times of communication-intensive jobs by 9% (326 hours) on average. The average wait time of jobs is reduced by 31% across three supercomputer job logs.

KEYWORDS

job scheduling, communication-aware, job-aware, SLURM

ACM Reference Format:
Priya Mishra, Tushar Agrawal, Preeti Malakar. 2020. Communication-aware Job Scheduling using SLURM. In *49th International Conference on Parallel Processing - ICPP : Workshops (ICPP Workshops '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3409390.3409410

1 INTRODUCTION

Job scheduling is an important area of research that deals with cluster management and resource allocation as per job requirements. Job requirements are usually specified in terms of number of nodes and the wall-clock time required for the execution of the job. Most job schedulers allocate nodes without considering the job characteristics, such as whether a job is communication-intensive or compute-intensive or I/O-intensive. Further, most current resource managers do not consider the communication patterns of a communication-intensive job while allocating nodes. This may lead to interference from other communication-intensive jobs, which may share the same network links and or switches for communications at the same time. Additionally, placing frequently communicating node-pairs several hops away, often leads to high communication times. This may be due to network congestion and long-hop communications [7, 19]. We conducted an experiment (spanned over several days) on our department cluster (50-node Intel Core i7 nodes connected via tree topology and 1G Ethernet) to study the effect of contention caused due to sharing of switches or links on the execution time of communication-intensive jobs. We execute two parallel MPI [4] jobs, J1 and J2, spread across two switches that are connected in tree-like topology. These jobs execute MPI_Allgather collective communication using large message size (1 MB). The first job J1 is executed repeatedly on 8 nodes (4 nodes on 2 switches) during this experiment. The second job J2, is executed every 30 minutes on 12 nodes spread across the same two switches. Hence, the two jobs share switches. Their effect on each other's execution time is shown in Figure 1, where the blue curve shows J1, and the orange curve shows J2. The execution time (in seconds) is shown along the y-axis and the wall-clock time (in seconds) along the x-axis (10-hour data). We observed a sharp increase in the execution time of J1 whenever J2 was executed (seen as spikes in the blue curve). This occurred when the two jobs ran simultaneously and thus communicated simultaneously over shared switches/links. This shows that it is important to carefully select nodes to avoid performance degradation.

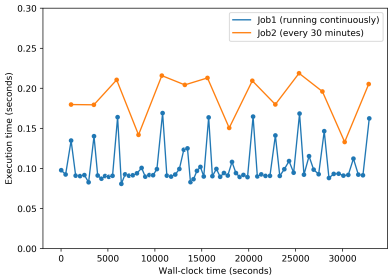


Figure 1: Execution of two communication-intensive jobs, J1 and J2.

In this work, we propose node allocation algorithms for tree-based network topologies. These resource allocation algorithms consider the job's behavior (compute or communication-intensive) as an additional input job parameter to improve the performance of communication-intensive jobs. We consider the underlying algorithms of MPI collectives instead of a global communication matrix. MPI collectives occur more prominently than point-to-point in production supercomputing applications [9]. MPI collectives often constitute the most time-consuming part of application runtimes (e.g. FFTW (MPI_Alltoall), OpenFOAM (MPI_Allreduce)) [9]. We considered three standard communication patterns – recursive doubling (RD), recursive halving with vector doubling (RHVD), and binomial tree algorithm. They occur in most MPI collectives.

ICPP Workshops '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8868-9/20/08…$15.00
https://doi.org/10.1145/3409390.3409410

We propose three node allocation algorithms – greedy, balanced, and adaptive. The greedy algorithm aims at improving the performance of communication-intensive jobs by allocating resources on the least contended switches. The balanced algorithm aims at allocating nodes in powers of two on the leaf switches, assuming that the communication pattern is governed by one of the popular parallel algorithms, such as recursive doubling and binomial tree [26]. Unbalanced allocations usually have more inter-switch communications in the intermediate steps due to the steps involved in these parallel algorithms. Therefore allocation in powers of two minimizes inter-switch communication. Since most of the communicating node-pairs may be present on the same switch in balanced allocation, the distance or hops between them also reduces, and there is a lesser possibility of sharing switches or links with other jobs. We also propose an adaptive algorithm that selects the best – the one with lower communication cost between the above two. We implemented our algorithms using SLURM, Simple Linux Utility for Resource Management [27], which is an open-source cluster management and job-scheduling system. We evaluated the performance of our algorithms using three job logs from Intrepid, Theta, and Mira supercomputers. Our algorithms resulted in a maximum reduction in execution times by ~77% for recursive doubling communication pattern. We got similar (~76%) maximum improvements for recursive halving vector doubling and binomial tree communication patterns as well. Our main contributions are:

- We propose three node allocation algorithms to improve the performance of communication-intensive jobs
- We implement our algorithms in SLURM and demonstrate improvement over the default allocation strategy
- We propose a novel way of optimizing based on the communication patterns/underlying algorithms of MPI collectives

The rest of the paper is organized as follows: §2 surveys prior work. §3 discusses factors affecting job scheduling. In §4, we describe our node allocation algorithms. §5 describes the experimental setup used for evaluations. §6 presents and discusses the results. Finally, §7 concludes this paper.

2 RELATED WORK

The resources allocated for a communication-intensive job play a crucial role in determining its performance. Several works [12, 14, 15, 20, 23, 24] have focused on optimizations for providing effective allocation on different topology such as fat-tree clusters. Many schedulers [13, 27] use first-in-first-out scheduling with backfilling without considering the impact of communication-intensive jobs scheduled on the same switches. This does not avoid contention due to multiple jobs sharing the network switches/links. This may lead to severe performance degradation for communication-intensive jobs due to resource fragmentation and network contention. Pollard et al. propose a scheduling strategy to avoid inter-job interference on fat-tree clusters [20]. The jobs are differentiated based on their size and are allocated resources such that there is no sharing of network links at any level in the fat-tree. However, these restrictions negatively impact the wait time, which has to be compensated by possible speedups in execution times. Soner et al. [23] developed an auction-based and ILP-based scheduler plugin for SLURM to favor topologically better allocations with lower levels of common switch and minimum node-spread. However, they do not consider the currently running communication-intensive jobs during node allocations. Also, solving the integer program at each scheduling attempt may introduce large overhead.

Subramoni et al. introduce topology-aware plugins for node allocation and task distribution in SLURM to mitigate performance degradation due to over-subscription and map processes based on the application P2P communications [24]. Georgiou et al. use the TREEMATCH algorithm in SLURM to obtain the most optimal allocation given the communication matrix for the application [12]. However, the above approaches, do not consider the impact of neighboring jobs and network contention on the application performance. Jokanovic et al. [15] propose the creation of virtual partitions to separate small and large jobs. They propose a set of constraints for a network topology with 18 nodes/switch to control job fragmentation. However, larger and irregular topologies (we consider a tree topology with 330-380 nodes/switch) will require more complex constraints to fit jobs of varying sizes. We consider the communication patterns of jobs and the switch contention due to communication-intensive jobs. Our scheduling approach aims at selecting nodes on the least contended switches.

Many of the above studies consider a global communication matrix to select nodes while minimizing the number of hops or hop-bytes [7, 12, 24]. While it is easier to fetch this data using profilers, they give incomplete information. This is because profilers give data about P2P communications in an application, but not necessarily the underlying P2P communications in MPI collectives. Further, a change in the number of nodes/processes requires re-profiling the application in the above approaches before mapping to the allocated nodes. However, we use the parallel algorithm communication structure of collectives to select the nodes. This gives a more definitive communication pattern without incurring profiling cost. This is important for applications where the collective communication costs dominate the execution times [16, 21]. Our strategies consider all stages of algorithms (RD, RHVD, Binomial) and allocate based on the costliest communication step/stage. This is difficult to achieve using a communication matrix.

3 SCHEDULING OF PARALLEL JOBS

Next, we discuss some important factors that affect job scheduling, specifically those considered in this study – the job scheduler, the network topology and the communication patterns in the job.

3.1 SLURM job scheduler

Job schedulers are used to allocate and manage resources in a cluster. In this study, we have used SLURM (Simple Linux Utility for Resource Management). It is a widely-used, open source resource management and job scheduling system [27]. It has a centralized control daemon `slurmctld` that maintains information about all the jobs running or waiting in the cluster. Each compute node has a `slurmd` daemon running on it. The pending jobs are placed in a queue and considered for scheduling in the order of their priority. The SLURM architecture uses multiple plugins for different tasks. The scheduler plugin selects the type of scheduler to use. SLURM uses a First-In-First-Out scheduling policy with backfilling. The select/linear plugin is used for allocating entire nodes to the jobs.

In our study, we propose new node allocation algorithms using this plugin. SLURM supports resource allocations for clusters with tree and fat-tree network topologies via the topology/tree plugin. The default node allocation algorithm identifies the lowest level switch that has the requested number of nodes available. For example., assume that nodes $n0$ and $n1$ are already allocated in the fat-tree network shown in Figure 2. Now if a job requires 4 nodes, the lowest level switch would be $s1$, whereas if a job requires 6 nodes, the lowest level switch would be $s2$. The algorithm first allocates nodes on those leaf switches that have minimum number of nodes available. This is known as best-fit allocation, typically used to reduce resource fragmentation. We use this plugin and assume tree/fat-tree topologies for developing our algorithms.

3.2 Network Topology

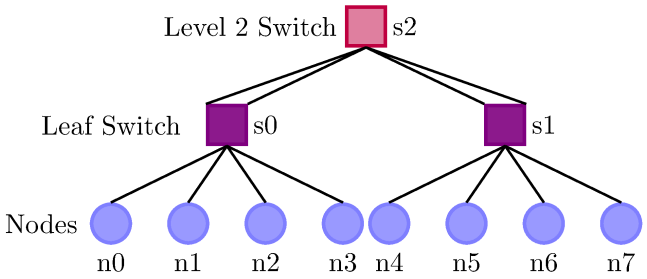


Figure 2: Fat-tree Network Topology

The network topology plays an important role in communication. It has considerable impact especially on the performance of communication-intensive jobs. Network contention due to sharing of links/switches between multiple jobs often leads to performance degradation. This is evident from the experiment described in Figure 1. One of the popular network topologies is fat-tree [17], and it is used in many of the top 500 supercomputers. We consider fat-tree based network topology for the implementation and evaluation of our algorithms. An example of a two-level fat-tree topology is shown in Figure 2. The level 1 switches, also known as leaf switches ($s0$ and $s1$), are connected to the compute nodes ($n0 - n7$). Each compute node is connected to only one leaf switch. The level 2 switches ($s2$) may be connected to one or more level 1 switches.

3.3 Communication Patterns

In this paper, we consider Message Passing Interface (MPI) as the main parallel library for communication. We assume that the parallel jobs submitted to the job scheduler use MPI for communication. This is indeed the case at many supercomputing centers. Prior studies retrieve a global communication matrix using profiling [8, 14, 24], which does not always reflect the most crucial communications. Such matrices convey the overall communications between two processes during the entire course of program execution. Typically this information is treated as a static input for deciding the best mapping and/or job allocation. Optimizing overall communications may not always lead to minimizing communication times because the temporal communication information between two processes is not considered. Moreover, most of the profilers [6, 22] may not be able to extract the point-to-point (P2P) communications that are internal to the MPI collectives. In contrast, we leverage the knowledge of application's most time-consuming communication function algorithm to recommend a better node allocation. It is challenging to profile and get the P2P communications due to MPI collective functions, which are used in majority of scientific applications. Therefore, we take into account the collective algorithms [5, 25, 26] for our work. We explain this next.

MPI applications use different MPI functions for communications depending on their needs. For example, MPI_Reduce is a commonly used reduction function to compute basic aggregate statistics such as summation, average. MPI_Bcast is a function to broadcast data, MPI_Allgather function is used when every process requires data to be gathered from every other process. Many application profiles reveal the existence of the most time-consuming MPI function, such as MPI_Allreduce in the case of OpenFOAM and AMBER, MPI_Alltoall in the case of CPMD [21]. Many of these functions are implemented using variants of recursive doubling/halving and ring algorithms in most MPI implementations [25, 26]. Detailed analysis of the implementation also reveals that the current MPI collectives use either ring or binomial tree or recursive halving/doubling based algorithms [5]. Recursive doubling is used in MPI_Allreduce (depicted in Figure 3), recursive doubling with vector doubling is used in MPI_Allgather. Some forms of these communications are also used in MPI_Reduce, MPI_Gather and MPI_Bcast, such as recursive halving and binomial tree. We propose parallel algorithm-aware job

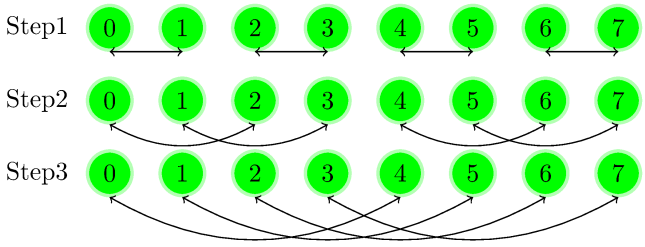


Figure 3: Recursive Doubling

scheduling. Therefore, we consider the communication patterns of these algorithms to determine which nodes may be more suitable for a given communication-intensive job. A better node allocation for a communication-intensive job is proposed, considering the current allocation of the running communication-intensive jobs and the communication pattern of the most-time consuming MPI collective function of an application. In the next section, we explain our node selection algorithms.

4 COMMUNICATION-AWARE SCHEDULER

In this section, we describe our node allocation algorithms. We propose two main node allocation algorithms – greedy and balanced allocation. Further, we assume that every job has been categorized as either communication-intensive or compute-intensive. This is usually true and can be deduced from the MPI profiles of an MPI application [16]. It can also be done through user input. The algorithms first identify the lowest-level switch in the tree topology that has the requisite number of nodes available (line 2 in Algorithms 1 and 2). If this lowest-level switch is a leaf switch, the requested nodes are allocated from this switch (lines 3-5 in Algorithms 1 and 2). The greedy algorithm aims to select nodes such that there is less

contention for network links among the communication-intensive jobs. While achieving this using the knowledge of jobs that are running, the balanced algorithm also considers the communication pattern and tries to allocate most communicating node-pairs on the same switch. Table 1 shows some of the common notations used.

Table 1: Notations

Notation	Description
i	Node index
L	Leaf switch
L_i	Leaf switch connected to node i
L_nodes	Total number of nodes on the leaf switch
L_comm	Number of nodes running communication-intensive jobs on the leaf switch
L_busy	Number of nodes allocated on the leaf switch
$C(i, j)$	Contention factor when nodes i and j communicate
$d(i, j)$	Distance between nodes i and j
$msize$	Message Size

4.1 Greedy Allocation

Algorithm 1: Greedy Allocation

```
   Input  : Job J, Number of requested nodes N
   Output : Set of allocated nodes A
1  A = { }
2  Find lowest level switch P with at least N nodes free
3  if P is a leaf switch then
4  |  Add N nodes to A
5  |  return A
6  else
7  |  if J is communication-intensive then
8  |  |  Sort the leaf switches connected to P in increasing order of
   |  |    communication ratio
9  |  else
10 |  |  Sort the leaf switches connected to P in decreasing order
   |  |    of communication ratio
11 |  R ⟵ N   // R is remaining nodes
12 |  Let L[1], L[2], .... L[k] be k leaf switches in sorted order
13 |  for each leaf switch L[i] in sorted order do
14 |  |  Add min(free[L[i]], R) nodes to A
15 |  |  R ⟵ R−min(free[L[i]], R)
16 |  |  if R == 0 then
17 |  |  |  return A
18 |  end
```

The greedy algorithm (Algorithm 1) aims at selecting the best underlying leaf switch at each step until the requested number of nodes have been allocated. For a communication-intensive job, the nodes are allocated on leaf switches that have maximum number of free nodes and minimum communication-intensive jobs already running. This is done to minimize fragmentation and contention. We characterize the leaf switches using their communication ratio:

$$Communication\ Ratio\ (L) = \frac{L\_comm}{L\_busy} + \frac{L\_busy}{L\_nodes} \quad (1)$$

The numerators and denominators are defined in Table 1. The first term of the communication ratio depends on the number of nodes running communication-intensive jobs relative to the number of busy nodes on the leaf switch. This is related to the contention on the leaf switch. The second term gives a measure of the number of available nodes on the switch, which helps control node-spread. Minimum communication ratio of a switch implies minimum contention and maximum nodes available for allocation. Therefore, the leaf switches are selected in increasing order of their communication ratio for a communication-intensive job (lines 7-8). A compute-intensive job is not affected by network-contention or fragmentation. Hence the leaf switches are selected in decreasing order of their communication ratio (lines 9-10). This ensures that the leaf switches with lower communication ratios are available for other communication-intensive jobs that may be queued or submitted in the future without affecting the performance of a compute-intensive job. The requested number of nodes is then allocated from these leaf switches considering them in sorted order (lines 11-18), where $free[L[i]]$ is the number of nodes available on the leaf switch $L[i]$.

4.2 Balanced Allocation

The balanced allocation (Algorithm 2) aims at allocating nodes in powers of two on the underlying leaf switches to minimize inter-switch communication. For example, consider a job with the communication pattern of recursive doubling shown in Figure 3. The best allocation for this job would be to allocate all the eight nodes (assuming the maximum number of nodes required for this job is 8) on the same leaf switch. However, if at least two leaf switches have to be used for allocating eight nodes, it may be better to allocate four nodes on each leaf switch rather than allocating the nodes in any other unbalanced way (e.g., 3 nodes on one switch and 5 nodes on another switch). This is because there would be inter-switch communications in Step2 and Step3 in the case of unequal/unbalanced allocation, whereas in a balanced allocation, all the communications in these two steps will be intra-switch. Bounding most communications to within a leaf-level switch (intra-switch) is usually better because there are fewer inter-job communications within a level 1 switch, whereas several jobs will use level 2 switches (inter-switch) to communicate among their processes though there may be multiple paths between level 1 switches (refer Figure 2).

The leaf switches are considered in decreasing order of their free node count (lines 9-10). This is to minimize the fragmentation of resources for a communication-intensive job, under the assumption that the communication algorithms are based on some of the popular algorithms in the literature (see Section 3.3). However, instead of allocating all the nodes available on the leaf switch, the number of nodes allocated is the largest power of two that can be accommodated on that leaf switch (lines 13-14). This number (alloc_size S, line 8) is deduced by sub-dividing number of nodes to a lower power of two nodes such that it can be satisfied by the available nodes in a leaf switch. If a leaf switch does not contain the maximum number of requested nodes, say 2^l, it is divided into two 2^{l-1} nodes (see Figure 4), and these are tried to be allocated on the same switch. This is continued for all the leaf switches in sorted order until the requested number of nodes is allocated or all the leaf switches have been utilized (lines 12-20). The algorithm selectively allocates only in powers of two initially, thereby leaving

Algorithm 2: Balanced Allocation

```
   Input  : Job J, Number of requested nodes N
   Output : Set of allocated nodes A
1  A = { }
2  Find lowest level switch P with atleast N nodes free
3  if P is a leaf switch then
4  |  Add N nodes to A
5  |  return A
6  else
7  |  R ⟵ N // R is remaining nodes
8  |  alloc_size S ⟵ N
9  |  if J is communication-intensive then
10 |  |  Sort the leaf switches connected to P in decreasing order
   |  |    of free nodes
11 |  |  Let L[1], L[2], .... L[k] be k leaf switches in sorted order
12 |  |  for each leaf switch L[i] in sorted order do
13 |  |  |  while S > free[L[i]] do
14 |  |  |  |  S /= 2
15 |  |  |  end
16 |  |  |  Add min(S, R) nodes to A
17 |  |  |  R ⟵ R−min(S, R)
18 |  |  |  free[L[i]] ⟵ free[L[i]]−min(S, R)
19 |  |  |  if R == 0 then
20 |  |  |  |  return A
21 |  |  end
22 |  |  if R > 0 then
23 |  |  |  for each leaf switch L[i] in reverse sorted order do
24 |  |  |  |  Add min(free[L[i]], R) nodes to A
25 |  |  |  |  R ⟵ R− min(free[L[i]], R)
26 |  |  |  |  if R == 0 then
27 |  |  |  |  |  return A
28 |  |  |  end
29 |  else
30 |  |  Sort the leaf switches connected to P in increasing order of
   |  |    free nodes
31 |  |  for each leaf switch L[i] in sorted order do
32 |  |  |  Add min(free[L[i]], R) nodes to A
33 |  |  |  R ⟵ R−min(free[L[i]], R)
34 |  |  |  if R == 0 then
35 |  |  |  |  return A
36 |  |  end
```

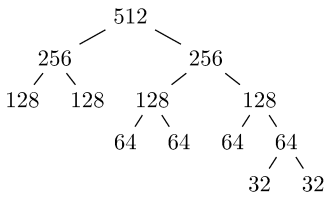


Figure 4: Balanced Allocation. Split nodes at each level l 2^l to two 2^{l-1} if sufficient nodes are not available at a leaf switch. The leaf nodes of this tree represent the actual number of nodes allocated in the leaf switches.

some unallocated free nodes in the leaf switches that were decided by SLURM to contain the requisite number of free nodes for the current job under consideration. Therefore, it may not be possible to satisfy the node requirement entirely while strictly allocating in powers of two. In such a case, when there are still remaining nodes to be allocated, and all the leaf switches have been used, the algorithm traverses the leaf switches again but in reverse sorted order. The remaining free nodes on each leaf switch are allocated (lines 22-27) until the node request is satisfied. The leaf switches are selected in increasing order of free nodes for a compute-intensive job (line 30). All the free nodes on each leaf switch are allocated without any constraint of allocating in powers of two (lines 31-35). This preserves leaf switches with larger numbers of free nodes for communication-intensive jobs. Since a compute-intensive job is not affected by fragmentation or contention, this does not affect the performance of compute-intensive jobs. Consider a communication-intensive job requesting 512 nodes. Figure 4 shows an example of node sub-division in powers of 2 for balanced allocation. Table 2 shows the corresponding free and allocated nodes on leaf switches. The algorithm begins with 512 node count and recursively halves it until the allocation size can be accommodated in that leaf switch. Therefore, the algorithm allocates 128 nodes on L[1] and L[2]. The allocation size is further halved for L[3]. This continues until the required number of nodes have been allocated.

Table 2: Balanced allocation for job requiring 512 nodes

Leaf Switch	L[1]	L[2]	L[3]	L[4]	L[5]	L[6]	L[7]
Free Nodes	160	150	100	80	70	50	40
Allocated Nodes	128	128	64	64	64	32	32

4.3 Adaptive Allocation

In the previous sections, we discussed greedy and balanced node allocation algorithms. However, based on the distribution of free nodes on the leaf switches, one algorithm may be more suitable than the other. The greedy allocation strategy minimizes contention and fragmentation, however, this allocation is unbalanced and may have more inter-switch communication than the optimal case. The balanced allocation strategy minimizes inter-switch communication, but the constraint of allocating in powers of two may lead to fragmentation of resources that can outweigh the benefit of decreased inter-switch communication. Therefore, we propose an adaptive allocation strategy that compares both allocations and selects the more optimal node allocation for a job. This is based on estimates of the communication cost for both greedy and balanced allocation. The cost of communication is calculated using effective hops for each communicating node-pair, which is based on the distance between them and contention on the leaf switches connected to them. The adaptive algorithm selects the one with the least cost. For compute-intensive jobs, it selects the one with higher cost. We discuss in detail this communication cost in the next section.

5 EXPERIMENTAL SETUP

Next, we describe the job logs used, implementation details, and evaluation metrics. We performed our experiments using SLURM-19.05.0 on an Intel Xeon (Ivybridge) processor.

5.1 Job Logs

We use the job logs of Intrepid, Theta, and Mira [3] supercomputers. We obtained Intrepid logs (2009) from the Parallel Workload Archive [1, 11]. We obtained Theta (2018) and Mira (2019) logs from Argonne

Leadership Computing Facility. [2]. Intrepid is a Blue Gene/P system of 40K 4-core nodes. Mira is a Blue Gene/Q system consisting of 48K 16-core nodes. The Theta supercomputer consists of 4,392 64-core nodes. These logs contain information about each job, such as job name, nodes requested, submission times, start times, etc. We used 1000 jobs from each of these logs. The maximum node requirements in Theta, Mira, and Intrepid logs were 512, 16384 and 40960, respectively. The job logs do not have information about the nature of the job. We assume that some of these are communication-intensive, while some are compute-intensive. This results in a mix of different types of jobs, typical to supercomputers. We varied this percentage of communication-intensive jobs from 30% – 90%. Further, we consider jobs with power-of-two node requirements, as typically used by application developers [10, 15, 18], and also found in the logs that we used. Theta logs have 90% power-of-two jobs. Intrepid and Mira logs have more than 99% power-of-two jobs. We emulate these logs by configuring SLURM with the enable-front-end option. The emulated jobs run for the same duration as their execution times (which are taken from the job logs). Each job log takes 2 – 5 days to complete per configuration/algorithm.

5.2 Implementation Details

We modify the data structures and functions related to the select/linear plugin of SLURM for implementing our algorithms. We use the environment variable `JOBAWARE` to execute our proposed algorithms. When `JOBAWARE` is defined, SLURM executes our proposed algorithm, else it executes the default allocation. The proposed algorithms have negligible overhead (less than 0.1 second). The network topology information is provided in a topology.conf file. For each leaf switch, the topology.conf file lists the nodes connected to it, and for the higher level switches, it contains a list of the underlying (child) switches. We obtained fat-tree topology files from IIT Kanpur and Lawrence Berkeley National Laboratory. The former has 16 nodes/leaf switch, whereas the latter has $\geq$ 300 nodes/leaf switch. An example topology.conf file for the tree topology of Figure 2 is shown below.

```
SwitchName=s0 Nodes=n[0-3]
SwitchName=s1 Nodes=n[4-7]
SwitchName=s2 Switches=s[0-1]
```

5.3 Runtime Estimates

Our algorithms only affect communication times, and we assume that the compute times are unaffected. Thus, we mainly focus on communication time performance. We estimated the communication times based on the maximum #hops incurred in a job [7, 12]. The hops are calculated using an estimation of contention in a shared switch. We took into account the number of hops in a communication, based on the particular communication algorithm. Communication-intensive job performance depends on whether the job shares switches (shares communication paths) with other communication-intensive jobs, as was illustrated in Figure 1. Next, we describe two terms that are used to calculate effective hops for a communication-intensive job based on their allocation.

1. Contention Factor $C(i, j)$: We define this for two cases: (1) Nodes that communicate are present on the same leaf switch (e.g., n0 and n1 in Figure 5) and (2) they are present on different switches (e.g., n0 and n4 in Figure 5). If nodes i and j are present on same leaf switch, i.e. $L_i = L_j$ (see Table 1):

$$C(i,j) = \frac{L_i\_comm}{L_i\_nodes} \tag{2}$$

If nodes i and j are present on different leaf switches i.e. $L_i \neq L_j$:

$$C(i,j) = \frac{L_i\_comm}{L_i\_nodes} + \frac{L_j\_comm}{L_j\_nodes} + \frac{1}{2}\frac{L_i\_comm + L_j\_comm}{L_i\_nodes + L_j\_nodes} \tag{3}$$

The first two terms represent the individual contention on the two leaf switches, L_i and L_j. The third term accounts for the contention on the lowest level common switch connecting L_i and L_j. A factor of half is used because the number of links double as we move up in a fat-tree. Contention is incurred at the three switches sequentially, therefore we sum the contention at each switch. Consider the example shown in Figure 5. Job1 is running on nodes $n0$, $n1$, $n4$, $n5$ and Job2 is running on $n2$, $n3$. Both jobs are communication-intensive. Nodes $n6$ and $n7$ are unallocated. Using Eq.2, $C(n0, n1) = \frac{4}{4} = 1$. We require switch-level contention at $s0$, $s1$ and $s2$ to calculate $C(n0, n4)$. Using Eq.3, $C(n0, n4) = \frac{4}{4} + \frac{2}{4} + \frac{1}{2}\frac{6}{8} = 1.875$. Thus this captures the fact that contention is higher for jobs spread across multiple switches. We obtained a high correlation of 0.83 between the execution time of jobs and contention values as calculated using Equations 2 and 3 for the study we conducted (Figure 1).

2. Distance $d(i, j)$: The distance between any two nodes i and j in a tree/fat-tree topology is given by:

$$d(i,j) = 2 * Lowest\ level\ of\ common\ switch \tag{4}$$

Therefore, in a two-level tree topology if i and j are on the same leaf switch, $d(i, j) = 2$, else $d(i, j) = 4$. Thus, $d(n0, n1) = 2$ and $d(n0, n4) = 4$ in Figure 5. Similarly, $d(i, j)$ may be calculated for any higher level tree topology. Using the above definitions, the effective hops between any two nodes i and j that communicate can be estimated as below:

$$Hops(i,j) = d(i,j) * (1 + C(i,j)) \tag{5}$$

This takes into account the contention on the shared links/switches and thus represents effective distance of communication. Hops(i,j) multiplied by the message size, *msize*, gives an estimate of effective hop-bytes(i,j). This gives an indication of communication time [7]. *msize* depends on the algorithm. For example, *msize* doubles in the case of vector doubling algorithms. In figure 5, we can calculate $Hops(n0, n1)$ and $Hops(n0, n4)$ using Eq. 5. Using the values of distance and contention as derived above, $Hops(n0, n1) = 2*(1+1) = 4$ and $Hops(n0, n4) = 4 * (1 + 1.875) = 11.5$.

The total cost of communication for a job is calculated by considering the hops for all communicating node-pairs.

$$Cost = \sum_{n=1}^{N} \max_{i,j \in S_n} Hops(i,j) \tag{6}$$

where S_n represents the set of all node-pairs communicating at nth step and N is the total number of steps in the communication algorithm. To estimate the modified runtime of a communication-intensive job under our allocation strategy, we calculate the cost of communication (Eq. 6) for both the default algorithm and our algorithm. The total runtime T of a job can be modeled as: $T = T_{compute} + T_{comm}$ where $T_{compute}$ and T_{comm} are the compute and communication times of the job respectively. We assume that

the communication time increases or decreases in the ratio of the cost of communication, whereas the compute time remains constant. The modified runtime is:

$$T' = T_{compute} + T_{comm} * \frac{Cost\_Job\text{-}aware}{Cost\_Default} \quad (7)$$

The runtime of a compute-intensive job is not affected.

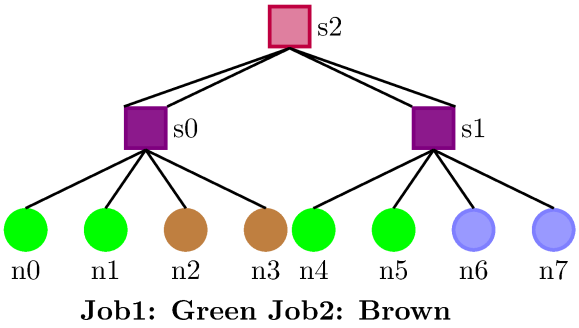


Figure 5: Cost of communication for a job spread across two leaf switch

5.4 Evaluation

We use five metrics to evaluate our algorithms – (1) Execution time (time between start and completion of a job), (2) Wait Time (time between submission and start of a job) (3) Turnaround Time (time between submission and completion of a job) (4) Node Hours (Number of nodes used by a job multiplied by its execution time) and (5) Cost of communication (calculated using Eq. 6). We conduct two types of experiments. In the first type of experiment, we ran 1000 jobs using the submission times derived from the original job logs (refer Section 5.1). These are referred to as *continuous* runs. For a fair comparison, we conducted another experiment. In this, we first ran a few jobs to partially occupy the cluster and then submitted one job at a time, i.e., the next job was submitted after the completion of the previous job. This provided a common starting point (the partially occupied cluster state) to compare the allocation (and hence performance) of every job under different node allocation algorithms. These are referred to as *individual* runs. This type of comparison is not possible with continuous runs because every job has a different starting point (state of the cluster) under different algorithms. This happens due to different node selection by different algorithms for every job. For example, the default SLURM algorithm selects nodes based on switch fragmentation, whereas our greedy algorithm considers communication-intensive jobs on the leaf switch. This leads to a difference in free/busy nodes in the cluster and hence results in different cluster states for a given job under different scheduling algorithms. Moreover, the difference in communication times as a result of different node allocations also results in different execution times for the same job. Therefore, for fair comparison and a better idea about the algorithm's performance, we evaluate both *continuous* and *individual* runs for default and our algorithms.

6 RESULTS

In this section, we compare the proposed algorithms with the default SLURM algorithm. We evaluate for recursive doubling/halving (RD), recursive halving vector doubling (RHVD), and binomial communication patterns.

6.1 Impact on execution time and wait time

Table 3 summarizes the results obtained for different job logs when 90% of the jobs are considered to be communication-intensive. The last eight columns of the table show the overall execution time and wait time (in hours) for *continuous* runs using default, greedy, balanced, and adaptive algorithm. Rows 1, 2, and 3 show the results for Intrepid, Theta, and Mira job logs, respectively. For each job log, we show the results for recursive halving vector doubling (RHVD) (top sub-row) and recursive doubling/halving (RD) (bottom sub-row) algorithms. The job characteristics (resource requirements) for each job log are different, and hence the range of values is different for each job log (see §5).

We obtained maximum improvements of 78%, 77%, and 82% with greedy, balanced, and adaptive allocation for *continuous runs*. From Table 3, we observe that the balanced and adaptive algorithms always perform better than default and greedy. Balanced and adaptive lead to improvement of about 7 – 22% in execution time over the default for all the three job logs. There was an improvement of 21 – 79% in wait times for balanced and adaptive algorithms over the default. This shows that the balanced and adaptive algorithms are able to improve performance by allocating nodes in a switch according to the communication pattern of the algorithm. For example, in the recursive halving communication pattern, the first half of the nodes do not communicate with the second half after the first step. Thus, allocating in powers of 2 leads to lesser inter-switch communication. The decrease in execution time of communication-intensive jobs means that resources become available faster for queued jobs. This decreases the wait times of both communication-intensive and compute-intensive jobs. The average wait time reductions were 35%, 26%, and 32% for Intrepid, Theta, and Mira logs. Hence, although the proposed algorithms do not directly impact the execution times of compute-intensive jobs, they may still benefit from the reduced execution times of communication-intensive jobs.

We note that greedy performs better for Intrepid and Theta jobs. This is because the greedy algorithm attempts to minimize contention. Hence the communication cost is lowered by selecting the least contended switches first. However, it may still result in unbalanced allocation, which may lead to higher inter-switch communication. This is avoided in balanced allocation; thus, it gives lower execution times. We obtained almost similar execution time reductions for both greedy and balanced for Theta logs. This is because greedy and balanced both allocated powers of 2 nodes per leaf switch due to fewer nodes/switch in the topology. Note that our algorithms have a higher impact on RHVD than RD, because the total number of parallel communications is higher for RHVD. Thus, the effect of node selection for a job in a less contended switch is more pronounced in the case of RHVD in almost all the logs. We note that there is little or negative improvement for the greedy algorithm in the case of Mira. This is because there were many communicating node-pairs on the same switches in the case of default, which led to lower communication cost. This did not happen for greedy, which considers only the node contention while allocating. Thus the more communication-heavy RHVD pattern had reduced performance. The balanced algorithm improves this by balancing the allocations per switch, and therefore we observe positive improvement. Another factor responsible for this reduced performance is the difference in available link/switch in the case

Table 3: Execution and wait times (in hours) in all three job logs.

Algorithm		Execution Time (hours)				Wait Time (hours)			
		Default	Greedy	Balanced	Adaptive	Default	Greedy	Balanced	Adaptive
1.	RHVD	1382	1351	1256	1251	57	49	27	27
	RD		1345	1264	1257		52	32	33
2.	RHVD	2189	1740	1700	1663	45303	31190	34539	33092
	RD		1810	1731	1706		34901	35874	31809
3.	RHVD	3289	3956	2342	2435	17387	34966	3685	4751
	RD		3285	2559	2637		15845	6336	5631

of default and greedy due to their different execution times. Therefore, we also compare by ensuring the same cluster state (busy/free nodes) for all algorithms and perform individual runs (refer §6.3).

6.2 Variation in Communication Patterns

Here, we consider various combinations of compute and communication times in a job, as well as different MPI collective communication patterns (RHVD, RD, Binomial) in a job, similar to mini/proxy exascale applications such as CrystalRouter and CMC2D [16]. We assume that 90% jobs in the job log spent significant amount of time in communication. We ran five sets of experiments with different compute and communication ratios – A: 67% compute, 33% RHVD; B: 50% compute, 50% RHVD; C: 30% compute, 70%RHVD; D: 50% compute, 15% RD, 35% Binomial; E: 30% compute, 21% RD, 49% Binomial. The communication ratios used here are similar to previous studies [14]. The communication patterns of D and E are similar to CMC2D [16]. Figure 6 compares the percentage

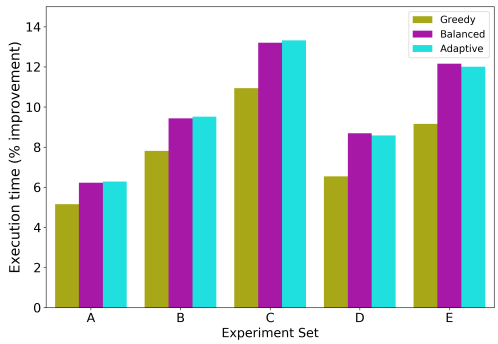


Figure 6: % Reduction in execution time using various communication patterns for Theta.

improvements in execution time for the different job log mix of Theta job logs. The percentage improvement in execution time is shown along the y-axis. The x-axis represents the different experiment sets A, B, C, D, and E, as mentioned above. We noted up to 49% improvement in execution times in all three cases using the proposed algorithms. The average improvement was 5.89%, 8.92%, 12.49%, 7.94%, and 11.11% for Theta log in the experiment sets A, B, C, D, and E, respectively. We note that as the communication ratio increases from 33% in A to 70% in C, the gains are higher for the same communication pattern. This is because a larger fraction of execution time reduces as a result of better node allocation achieved using the proposed algorithms. The same trend is seen between experiment sets D and E, which have the same communication pattern (similar to CMC2D) but different communication ratios (E is more communication-intensive). For Intrepid log, we noted a maximum average improvement of 2.59%, 3.92%, 5.49%, 3.71%, and 5.19% in experiment sets A, B, C, D, and E, respectively across all algorithms. In case of Mira, we obtained a maximum average improvement of 7.20%, 10.90%, 15.27%, 6.68%, and 9.36% in experiment sets A, B, C, D, and E, respectively. The Intrepid and Mira logs also show that the gains improve with increase in communication ratio. Also, for the same communication ratio of 50% in experiment sets B and D, we obtain higher gains for B in all logs. This shows that the communication-heavy RHVD (used in B) gains more than RD and Binomial (used in D). The same is true for experiment sets C and E, which have the same communication ratio of 70% but different communication patterns. This demonstrates that our algorithms are able to reduce the communication cost of different combinations of parallel algorithms due to system-aware and balanced allocations.

6.3 Continuous vs. individual runs

Table 4 summarizes the average percentage improvements in the execution times for *individual* runs. We did this comparison for 200 randomly selected jobs under different algorithms for all job logs. Rows 1, 2, and 3 show the results for Intrepid, Theta, and Mira job logs, respectively. For each job log, we show the results for recursive halving vector doubling (RHVD) (top sub-row) and recursive doubling/halving (RD) (bottom sub-row) algorithms. The last three columns show the percentage improvements in execution time using greedy, balanced, and adaptive over the default algorithm. We note that for all job logs, the proposed algorithms perform better than the default algorithm. We obtained an improvement of about 2 – 11 % in execution time using greedy allocation. We also observe that similar to *continuous* runs, balanced and adaptive always perform better than greedy and default algorithm in *individual* runs. There was an improvement of about 7 – 25 % in execution times using balanced and adaptive allocation. The obtained results further show that for a given state of cluster (similar busy/free nodes), the proposed algorithms always provide a similar or better allocation than the default algorithm in all the job logs.

Table 4: Improvements in execution times.

Algorithm		Execution Time (% improvement)		
		Greedy	Balanced	Adaptive
1.	RHVD	3.65	7.23	7.81
	RD	1.70	8.12	8.29
2.	RHVD	9.65	9.65	9.65
	RD	13.56	13.56	13.56
3.	RHVD	10.84	19.69	21.71
	RD	9.45	24.32	24.91

Figure 7 compares the execution times of continuous runs vs. individual runs for 200 jobs in Theta job log. Note that the sequence of job execution may be different due to the backfilling policy in the case of continuous runs. Here, the jobs are assumed to be using recursive doubling/halving communication pattern as the most

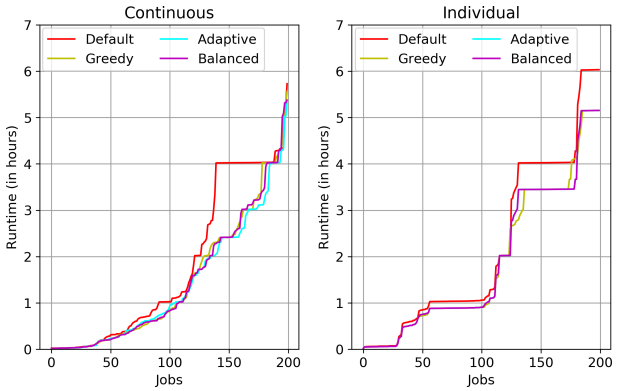


Figure 7: Comparison of execution times for Theta job log using recursive doubling/halving communication pattern in the continuous and individual runs.

time-consuming communication component. The y-axis shows execution times, and x-axis shows the job ids. The left sub-graph shows the execution time under different allocations for continuous runs, and the right sub-graph shows the execution times for individual runs. We achieved either similar or higher performance gains over default for all the algorithms during the individual runs, as can be noted in the figure. There is a maximum reduction of 70% and 15% in execution times for continuous and individual runs (see between job id 125 – 175). We obtained a maximum improvement of 76%, and 82% in execution time for continuous runs in the Intrepid, and Mira logs respectively. For the individual runs, we observed a maximum improvement of 57%, and 61% in the Intrepid and Mira logs.

6.4 Impact on Communication Cost

Figure 8 compares the communication cost of different allocations for all job logs. Here, 90% jobs are considered as communication-intensive and are assumed to be using binomial communication pattern. The communication cost (Equation 6) is shown along the y-axis and the corresponding node ranges are shown along the x-axis. The proposed algorithms have lower cost of communication than the default in all cases. We obtained an average reduction of about 3.4% using greedy allocation, whereas the balanced and adaptive result in ~11% improvement. Greedy algorithm only tries to reduce switch contention. Balanced algorithm aims at allocating nodes in powers of two to reduce inter-switch communication. Reducing inter-switch communication reduces the cost of communication since more communicating node-pairs are within the same leaf switch, hence the distance between them (see Equation 4) is reduced. Further, contention is usually lower when the communicating node-pairs are present on the same leaf switch since it is not affected by contention at higher level switches (see Equation 2). This is the case when communicating node-pairs are present on different leaf switches (see Equation 3).

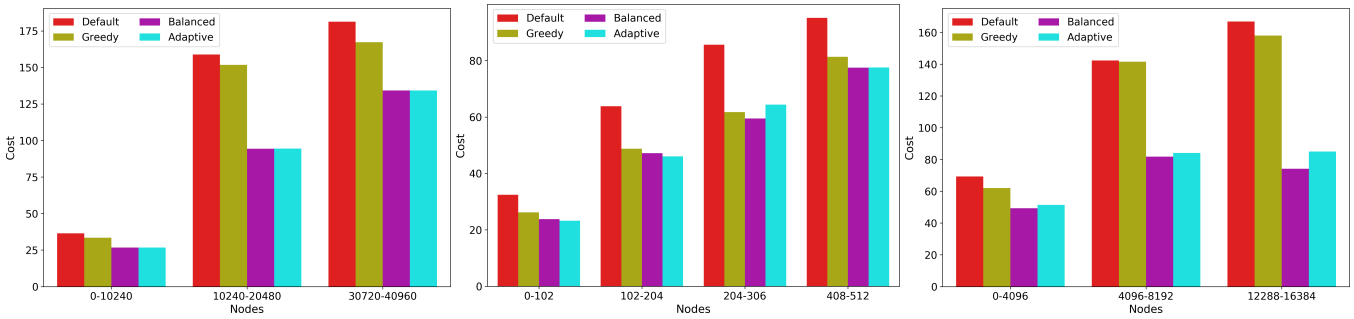
We also noticed a slight increase in the cost of communication for adaptive allocation over balanced allocation in some cases. This may have been due to errors in estimating the relative cost of communication. We plan to further investigate this in future. We also compare the cost of communication for all job logs under different allocations using other communication patterns such as recursive doubling/halving (RD) and recursive halving vector doubling (RHVD). We noticed a reduction of 5.56%, 5.72% and 5.72% in the communication costs of RD, RHVD and binomial respectively for Intrepid log across all algorithms. There was an improvement of 15.88%, 17.84% and 15.87% for Theta logs in the communication costs of RD, RHVD and binomial respectively. Similarly, for Mira logs, we obtained an improvement of 5.48%, 6.09% and 5.40% in the communication costs for RD, RHVD and binomial respectively.

6.5 Variation with percentage of communication jobs

Figure 9 compares the average turnaround times (in hours) and node hours for Intrepid job log for different algorithms. The y-axis shows the average turnaround time (in hours) (left figure) and average node-hours (right figure). The x-axis shows the percentage of communication-intensive jobs (varied as 30%, 60% and 90%) present in the job logs. The jobs are assumed to use recursive halving vector doubling (RHVD) communication pattern as the most time-consuming communication. We observe that for all percentages of communications, the proposed algorithms perform better than the default algorithm (lower is better). We obtained an average improvement of 0.6 – 2.8% in turnaround times and 0.5 – 1.9% in node hours using greedy algorithm. Balanced and adaptive had average improvements of 2.2 – 11.1% in turnaround times and 2.3 – 7.8% in node-hours. The adaptive allocation gave an average improvement of 2.55% and 2.61% in turnaround time and node-hours, respectively, when the percentage of communication-intensive jobs was 30%. This increased to 11.10% and 7.85% improvement in turnaround time and node-hours, respectively, as the percentage of communication-intensive jobs was increased to 90%. We got similar improvements for RD communication pattern as well. This is because the number of communication-intensive jobs increases with higher percentages. Hence, more jobs are allocated nodes in a job-aware way. For Theta and Mira job logs, we conducted experiments for 90% case. The average reduction in turnaround times across all algorithms for Theta and Mira was 25.7% and 28.7% respectively. Lower job turnaround times and lower node-hours imply jobs finished faster and thus improves system throughput (up to 31% for Theta and 12.5% for Mira). This shows the utility of our algorithms.

7 CONCLUSIONS AND FUTURE WORK

Performance of a communication-intensive job is affected by node-spread, job interference and network contention. Considering job characteristics such as communication patterns can lead to more optimal allocations and lower communication times. We propose three node allocation algorithms – greedy, balanced and adaptive, that consider the nature of the job (compute-intensive and communication-intensive) while allocating resources. We implement and evaluate our algorithms using SLURM. We show that the proposed algorithms can lead to reduced execution times and wait times using job logs from Intrepid, Theta and Mira supercomputers. We obtained an average improvement of 6% in the execution time and 35% in wait time for Intrepid across all the algorithms. There was an average improvement of 17% in execution time and 26% in wait time for Theta logs across all the algorithms. Balanced and adaptive allocations mostly perform better than greedy. We obtained an average improvement of 16% in execution time and 71% in wait time for Mira logs using balanced and adaptive allocations. Our work shows the utility of job-aware node allocations in terms of reduced execution times, wait times and improved system



(a) Intrepid (b) Theta (c) Mira

Figure 8: Comparison of cost of communication for different logs using Binomial algorithm under different allocations

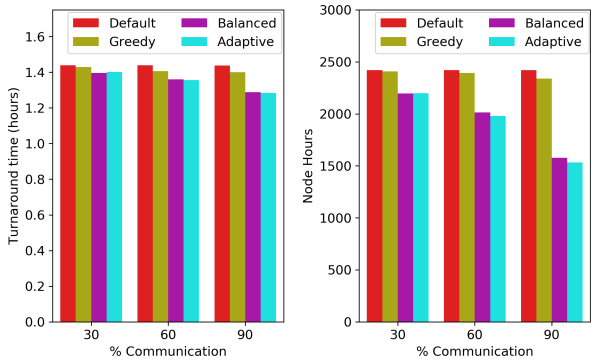


Figure 9: Comparison of turnaround times (in hours) and node hours for Intrepid for recursive halving vector doubling (RHVD)

throughput. As a part of our future work, we plan on including other communication patterns such as ring, stencil, etc. in our analysis. Process mapping after node allocation can provide further improvements and we would like to explore this. We would also like to extend our optimizations to other topologies using appropriate contention factor. Further, we plan on developing I/O-aware scheduling algorithms that consider I/O patterns in addition to communication patterns while allocating resources.

ACKNOWLEDGMENTS

We are thankful to the Computer Center, IIT Kanpur for help with the HPC2010 topology. The data for Theta and Mira was generated from resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We thank the National Energy Research Scientific Computing Center for help with the Cori topology.

REFERENCES

[1] 2005. *Parallel Workload Archive.* www.cse.huji.ac.il/labs/parallel/workload/
[2] 2019. *ALCF, ANL.* https://reports.alcf.anl.gov/data/index.html
[3] 2020. *Mira and Theta.* https://www.alcf.anl.gov/alcf-resources
[4] 2020. MPICH. https://www.mpich.org.
[5] 2020. MPICH Source Code. https://github.com/pmodels/mpich.
[6] Laksono Adhianto and et al. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.
[7] T. Agarwal, A. Sharma, A. Laxmikant, and L. V. Kale. 2006. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium.*
[8] B. Brandfass, T. Alrutz, and T. Gerhold. 2013. Rank reordering for MPI communication optimization. *Computers & Fluids* 80 (2013), 372 – 380.
[9] Sudheer Chunduri and et al. 2018. Characterization of MPI Usage on a Production Supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18).*
[10] W. Cirne and F. Berman. 2001. A comprehensive model of the supercomputer workload. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538).* 140–148.
[11] Dror G. Feitelson, Dan Tsafrir, and David Krakov. 2014. Experience with using the Parallel Workloads Archive. *J. Parallel and Distrib. Comput.* 74, 10 (2014).
[12] Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, and Adèle Villiermet. 2018. Topology-Aware Job Mapping. *Int. J. High Perform. Comput. Appl.* 32, 1 (2018), 14–27.
[13] Robert L. Henderson. 1995. Job Scheduling Under the Portable Batch System. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS '95).* Springer-Verlag, Berlin, Heidelberg, 279–294.
[14] Emmanuel Jeannot and Guillaume Mercier. 2010. Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures. In *Euro-Par 2010 - Parallel Processing.* 199–210.
[15] Ana Jokanovic, Jose Carlos Sancho, German Rodriguez, Alejandro Lucero, Cyriel Minkenberg, and Jesus Labarta. 2015. Quiet Neighborhoods: Key to Protect Job Performance Predictability. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium.* 449–459.
[16] Benjamin Klenk and Holger Fröning. 2017. An Overview of MPI Characteristics of Exascale Proxy Applications. In *High Performance Computing.* Springer International Publishing, 217–236.
[17] C. E. Leiserson. 1985. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.* 10 (1985), 892–901.
[18] Hui Li, David Groep, and Lex Wolters. 2004. Workload Characteristics of a Multi-Cluster Supercomputer. In *Proceedings of the 10th International Conference on Job Scheduling Strategies for Parallel Processing* (New York, NY) *(JSSPP'04).* Springer-Verlag, Berlin, Heidelberg, 176–193. https://doi.org/10.1007/11407522_10
[19] Jose A Morinigo and et al. 2020. Performance drop at executing communication-intensive parallel algorithms. *Journal of Supercomputing* (2020).
[20] Samuel D. Pollard, Nikhil Jain, Stephen Herbein, and Abhinav Bhatele. 2018. Evaluation of an Interference-Free Node Allocation Policy on Fat-Tree Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18).* IEEE Press.
[21] Gilad Shainer, Tong Liu, Pak Lui, and Richard Graham. 2011. *Accelerating High Performance Computing Applications Through MPI Offloading.* Technical Report. HPC Advisory Council.
[22] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.* 20, 2 (2006), 287–311.
[23] Seren Soner and Can Özturan. 2014. *Topologically Aware Job Scheduling for SLURM.* Technical Report. Partnership for Advanced Computing in Europe.
[24] Hari Subramoni, Devendar Bureddy, Krishna Chaitanya Kandalla, Karl W. Schulz, Bill Barth, Jonathan L. Perkins, Mark Daniel Arnold, and Dhabaleswar K. Panda. 2013. Design of network topology aware scheduling services for large InfiniBand clusters. *IEEE International Conference on Cluster Computing (CLUSTER)* (2013).
[25] Rajeev Thakur and William D. Gropp. 2003. Improving the Performance of Collective Operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Jack Dongarra, Domenico Laforenza, and Salvatore Orlando (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–267.
[26] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of Collective Communication Operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.
[27] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing.* 44–60.