Network and Load-Aware Resource Manager for MPI Programs

Ashish Kumar*
Indian Institute of Technology
Kanpur, India
akashish@iitk.ac.in

Naman Jain*
Indian Institute of Technology
Kanpur, India
namanj@iitk.ac.in

Preeti Malakar
Indian Institute of Technology
Kanpur, India
pmalakar@iitk.ac.in

ABSTRACT

We present a resource broker for MPI jobs in a shared cluster, considering the current compute load and available network bandwidths. MPI programs are generally communication-intensive. Thus the current network availability between the compute nodes impacts performance. Many existing resource allocation techniques mostly consider static node attributes and some dynamic resource attributes. This does not lead to a good allocation in case of shared clusters because the network usage and system load vary. We developed a load and network-aware heuristic for resource allocation. We incorporated the current network state in our heuristic. It is able to reduce execution times by more than 38% on average as compared to the default allocation.

CCS CONCEPTS

• **Theory of computation** → **Scheduling algorithms**; • **Computing methodologies** → *Parallel computing methodologies*; • **Networks** → *Network performance evaluation.*

KEYWORDS

MPI, resource discovery, resource allocation, static and dynamic attributes, network-aware allocator, resource monitoring

ACM Reference Format:
Ashish Kumar, Naman Jain, and Preeti Malakar. 2020. Network and Load-Aware Resource Manager for MPI Programs. 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/10.1145/3409390.3409406

1 INTRODUCTION

Message Passing Interface (MPI) is primarily used to write distributed memory parallel programs for clusters or supercomputers. The MPI library (e.g. MPICH [5]) takes care of the communication setup and messaging required to run parallel codes on multiple nodes. Nodes are allocated by a job scheduler when an MPI job runs on a supercomputer. Whereas, the user is expected to specify a list of hostnames while executing an MPI job on an unmanaged cluster. Typically, users randomly select a few nodes without much knowledge about the current network connectivity of these nodes. They may also specify all the nodes of the cluster in a file, in which case the process managers [2] select the nodes based on the order specified in the file. However, job schedulers or resource/process managers such as Hydra, Slurm [19], PBS do not adapt to the *current state* of the nodes before allocating *a good set* of nodes to the jobs. Current state indicates the current CPU usage, memory usage, available network bandwidth etc. We define a good set of nodes as those that have low CPU load, low CPU utilization, high network bandwidth and low memory usage.

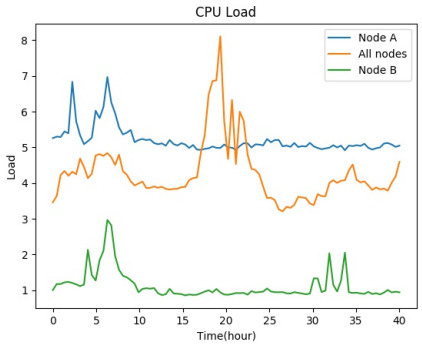
It is generally assumed that a cluster exclusively runs jobs using a process manager only. The above assumption may not hold true in case of non-dedicated/shared compute clusters such as compute lab clusters in academic institutions. Further, in case of non-dedicated clusters, the resource usage has a lot of variation, as shown in Figures 1 and 2. Figure 1 shows the resource usage variation across 20 nodes of a cluster in Indian Institute of Technology Kanpur over a period of two days. Each node is equipped with a 6-core hyper-threaded Intel Core i7 processor. Figure 1(a) shows the CPU load (i.e. number of processes waiting to be executed) of two randomly selected nodes (node A and B) as well as the average across 20 nodes of our cluster (orange curve). It can be observed that there are occasional spikes in the CPU load in all three curves. However, average CPU load is mostly low, and node B typically has quite low CPU load. Figure 1(b) shows the system-wide network I/O usage of two distinct nodes (randomly chosen) as well as the average (orange curve) across all nodes in the cluster over a period of 2 days. In particular, we measured the number of bytes sent and received at the network interface of the nodes using an external library [6]. This shows the network activity at the nodes, and the figure shows a lot of variation. A spike can be observed during the initial few hours in all the curves. After the 12^{th} hour, the network activity of both nodes A and B seem to be relatively low. This suggests these two nodes may be good candidates for running communication-intensive MPI programs. Figure 1(c) shows the average CPU utilization and memory usage across all nodes. The CPU utilization varies between 20% – 35%.

All metrics shown in Figure 1 vary considerably across nodes because the cluster is shared among many users. Different users use these systems for different purposes, such as to conduct practical lab sessions, perform research experiments, complete assignments, view video lectures, etc. In this scenario, where users can log into any cluster node and the cluster has multiple usages, a traditional job scheduler will fail to serve the purpose of running one job at a time on these nodes exclusively. This sometimes leads to increased CPU load in unmanaged clusters (e.g. see the spike at the 20^{th} hour in Figure 1(a)). However, many times the overall cluster utilization is low. Note that the CPU utilization is always below 35% (Figure 1(c)). Thus there is a scope of utilizing the under-utilized CPU capacity

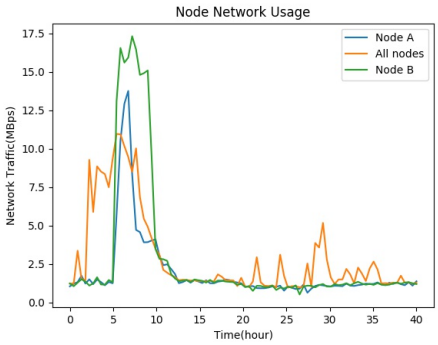
*Both authors contributed equally to the paper

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

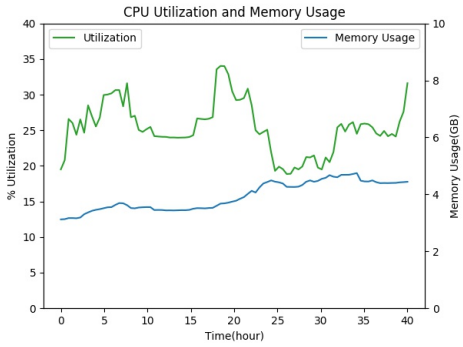
ACM ISBN 978-1-4503-8868-9/20/08…$15.00
https://doi.org/10.1145/3409390.3409406



(a) Variation in CPU load



(b) Variation in network I/O usage.



(c) Variation in CPU utilization and memory usage.

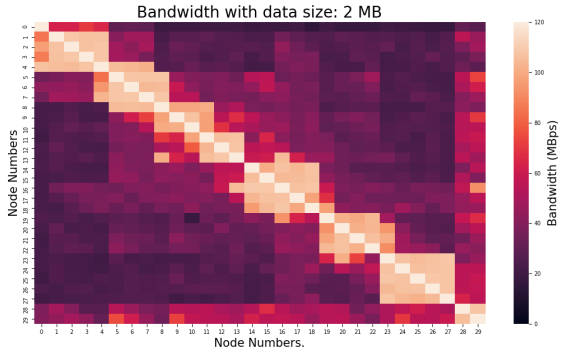
Figure 1: Variation in node resource usage in a shared cluster in IIT Kanpur.

for other jobs. For example, it can serve as a testbed cluster to run MPI parallel jobs that require multiple nodes. Further, it can also serve as an on-demand platform for running urgent MPI jobs, such as epidemic modeling, wildfire modeling, for which cloud has often been used [14, 15]. This is especially useful because the job wait times on supercomputers may be in the order of days.

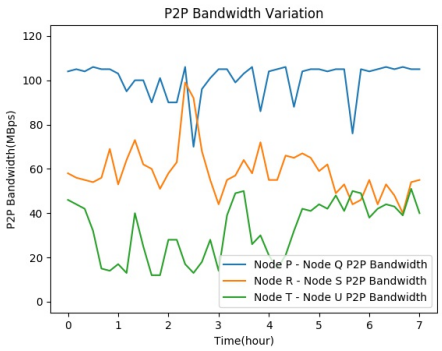
MPI jobs are most often communication-intensive [13], therefore the cluster topology, the network bandwidth and latency between the allocated nodes play an important role. Figure 2(a) shows a heatmap of peer-to-peer (P2P) measured bandwidth between thirty nodes of our cluster averaged over ten runs. The bandwidth varies a lot across pairs which can be observed by light (high available bandwidth) and dark (low available bandwidth) patches. Node numbering is based on physical proximity (1 – 4 hops). We can observe that nodes with closer proximity have somewhat higher bandwidth (as expected) than those that are distant. However, note that there are variations in the graph. Though the sequentially numbered nodes (with closer proximity) have better connectivity, the performance of a program is subject to other factors such as the system load on those nodes as well as fluctuations in network connectivity. These fluctuations are due to other network-intensive jobs. Thus, selecting sequentially numbered nodes may not result in the best set of nodes. Furthermore, detailed variation in P2P bandwidth with time can be observed in Figure 2(b). It depicts the network bandwidth (averaged across ten runs) of three node pairs (randomly-selected) in our cluster. P2P bandwidth of node pairs vary considerably across time and across node pairs. Note that the bandwidth of a given pair tends to fluctuate significantly around a base value, which is determined by how good the node pair is physically connected, i.e., topology. These fluctuations are mainly due to shared network switches and links with various network-intensive jobs running on these and other nodes of the cluster. Knowledge of these variations across nodes and time can help us to allocate resources in a better way. For example, statistical methods can be used to model variations in system parameters. However, the user may not be aware of resource usage variation (for e.g., note the peaks in CPU load and network usage; and light and dark patches in P2P bandwidth).

In this work, we address the problem of allocating a good set of nodes to run MPI jobs in a shared cluster that has variability in resource usages such as CPU utilization, network load etc. We do not assume that MPI jobs are exclusively run on the cluster machines. This is in contrast to Condor [16] that assumes dedicated systems for MPI jobs. Condor-like distributed computing systems [15, 16] are built to utilize unused compute cycles, and not for MPI jobs in non-dedicated clusters. We also consider clusters that vary in software and hardware configurations. For example, our cluster has a few 8-core and a few 12-core machines. In such cases, the MPI libraries do not distinguish between the compute capacities of these nodes, but instead rely on the user's knowledge of the cluster configuration. This can hurt MPI job performance. Moreover, some jobs may be communication-intensive, some may be compute-intensive. Therefore, it is necessary to allocate the right set of nodes based on the job characteristics. In this scenario of varying hardware configurations and varying resource utilization of the nodes, it is challenging to match jobs to appropriate resources. In fact, optimal task allocation is an NP-hard problem [10].

We propose a resource matchmaker that collects information about the resources and provides the most suitable resources to MPI jobs. Many existing matchmaking techniques consider static node attributes [19] and few dynamic node attributes [1, 16] of the resources. Static attributes such as RAM, network bandwidth, CPU speed do not provide enough information about the suitability of a resource. We also need to consider dynamic attributes of the resources such as CPU load, available memory, bandwidth and latency. MPI programs are usually communication-intensive, therefore it is crucial to consider the effective bandwidth between allocated nodes.



(a) Variation in P2P bandwidth across pair of nodes.



(b) Variation in P2P Bandwidth across time.

Figure 2: P2P Bandwidth variation across node pairs

Many existing matchmakers [16] do not consider the current network state while various others [11] consider the network topology but not the current state of network between the compute nodes. We also consider the data in/out flow rate at a node, that represents the current network usage of the node. There may be contention and congestion in the network due to existing jobs, which may decrease the current network availability between nodes.

Our technique takes into account both static and dynamic attributes of resources, including network availability of the allocated nodes. We consider both current and historical data of node attributes and network availability variations across time and nodes. First, we find a few candidate groups of resources that satisfy the job requests using a greedy algorithm. Then, we calculate the compute and network cost of each group. The group with the minimum weighted sum of compute and network cost is allocated to the job. We evaluated our algorithm on a maximum of 64 cores of IIT Kanpur's Intel cluster using two mini applications, miniMD and miniFE. We observed an average of 32 – 49% improvement over random, sequential and load-aware allocation algorithms.

The rest of the paper is organized as follows. Section 2 surveys some important prior studies. Section 3 explains the system workflow and the node allocation policy. Section 4 describes the monitoring tools that collect data for the allocation algorithm. Section 5 gives a performance evaluation of our algorithm. Finally, we conclude and discuss future work in Section 6.

2 RELATED WORK

There has been a lot of study on efficient resource allocation. Kaur et al. [12] proposed a two-phase ranking approach that considers static and dynamic resource attributes for resource selection. They consider multiple resource attributes like CPU load, total cores and memory for resource selection. They generate a ranking based on pairwise comparison of resources, using PROMETHEE-II [9], AHP (Analytic Hierarchical Process) and SAW (Simple Additive Weight) [7] methods. However, they only consider computation power but not the network availability among the resources, which is crucial for MPI.

Yang et al. [18] present a model for grid resource allocation where the grids may span several domain administrations via the internet. They compute the total power of a cluster site based on the compute load, available processors, memory usage of the site nodes, and intra-network usage. They use a greedy algorithm considering the cluster's current total power and network availability between those clusters. It selects the group of clusters with minimum compute load and network traffic simultaneously minimizing the number of clusters to minimize inter-cluster traffic. However, they do not specify any allocation policy inside the selected cluster.

HTCondor [16] is a workload management system for compute-intensive jobs. Users submit their serial or parallel jobs to HTCondor. It places them into a queue, chooses when and where to run the jobs, carefully monitors their progress, and informs the user upon completion. Users may specify requirements and ranking criterion of resources. The matchmaker selects the top nodes based on their ranks. Several research extensions to the match-making have been built for co-allocation of more than one resource. The ranking criterion is limited to local node attributes. In contrast, we balance the load across nodes considering the network state of the cluster in addition and we do not assume dedicated systems for MPI jobs. Georgiou et al. [11] present a method for mapping jobs based on network topology and application characteristics. It gathers affinity between the processes to be run for the job and then uses Tree-match algorithm for mapping it to the nodes.

There are many workload managers for computing clusters and grid. Slurm [19] and Moab [1] with Torque are two such load managers that are extensively used in academia and industry for the scheduling and allocating node resources to jobs. While Slurm allows static attributes based filter for node allocation, Moab, in addition to that, allows giving priority functions based on the dynamic attributes of nodes. However, both of them do not consider the current state of network availability between the nodes. These two managers provide support for developing custom plugins where a more sophisticated node allocation algorithm can be incorporated. We plan to integrate our system in future.

Network Weather Service [17] is a distributed performance forecasting framework that monitors and forecasts CPU and network performance continuously. It then applies various time series methods and uses the method that exhibits smallest prediction error for next forecast. These forecasts are utilized extensively for guiding

scheduling decisions. We use a composite metric for CPU and network load, similar to Network Weather Service (NWS) [17]. We have developed our own light-weight resource monitoring system for more flexibility with respect to the metrics.

Alicherry et al. [8] presented a selection algorithm for VM placement in distributed cloud to minimize the maximum distance or latency between the VMs. They aim to reduce the possibility of tasks running on distant pairs of virtual machines, which will lead to large communication latencies and hence delay overall completion times. They modeled the problem in the form of a sub-graph selection problem, and they proposed the Min-Star algorithm for data center selection. We used a similar approach for node selection. We take into account the state of the node during allocation and create a composite metric using the state of a node and network links to define edge weights in the network graph because we target shared clusters in contrast to dedicated VMs.

3 NETWORK AND LOAD-AWARE ALLOCATION

We now describe our node allocation system. We first give a brief overview of the system and then describe the rest in detail.

3.1 System Overview

Our system consists of two core components, namely the Resource Monitor and Node Allocator. Figure 3 gives an overall workflow of these components. Resource Monitor is a distributed monitoring system for cluster. Resource Monitor uses light-weight daemons for periodically updating list of active nodes (livehosts) and node statistics such as CPU load, CPU utilization, memory usage and network status. Node Allocator allocates nodes based on user request. It considers node attributes and network dynamics. The Node Allocator uses data collected by the Resource Monitor. Next, we describe the system parameters such as compute load and network load that affect the overall performance of programs, and hence are used in our system. These parameters are calculated using the data provided by Resource Monitor and later used in Node Allocator.

3.2 System Parameters

System parameters such as local node attributes and network statistics are required to be considered for jobs that run on multiple nodes. The local node attributes can be modeled to represent the state of a node while the peer-to-peer data traffic i.e bandwidth and latency can be modeled to represent network state between two nodes. Next, we define compute load and network load accordingly.

3.2.1 Compute Load. We define the compute load as the overall load of a specific node. We determine compute load based on various dynamic node attributes, such as the CPU load, CPU utilization, memory usage and node data flow rate (described below) along with static attributes of a node such as the CPU clock speed and total memory (Table 1 column 1). CPU load refers to the number of processes waiting to be executed by the operating system. CPU utilization refers to the amount of current usage of the CPU cores. These are aggregate values across all logical cores. High CPU load and high CPU utilization imply that processors are busy executing other jobs, and hence are less favorable to run our jobs. Node data flow rate refers to the amount of data (packets) transmitted and received by a node over a period of time. This indicates congestion at the node. We also use the running mean of the last 1, 5, and 15 minutes of historical data of dynamic attributes which allows our allocator to make a more informed selection.

Table 1: Node Attributes

Attribute Name	Optimization Criterion
CPU/Core Count	maximize
CPU Frequency	maximize
Total Memory (RAM)	maximize
Node Current Users	minimize
Average CPU Load (1,5 and 15 min)	minimize
CPU Utilization (1,5 and 15 min)	minimize
Node data flow rate (1,5 and 15 min)	minimize
Available Memory (1,5 and 15 min)	maximize

Different node attribute have different optimization criterion for optimal node selection (see column 2 of Table 1). Some node attributes are desired to be as low as possible while others need to be as high as possible. For example, CPU utilization represents a minimization criterion, as it is desired that the nodes to be allocated should have low CPU utilization. Node data flow rate represents a minimization criterion, as we prefer minimum traffic going in and out of the node. Available memory represents a maximization criterion, as it is desirable to have high available memory in the nodes. Similarly, it is desirable to maximize CPU/core count, CPU frequency and total memory while CPU load and Node current users have to be minimized. First, the attribute values of each node are normalized by dividing the value by the sum of attribute values of all nodes. Then, we convert all the attributes in unidirectional units (same sign). This is done by complementing (with respect to the maximum value) for attributes having maximization criterion.

After this, we follow Simple Additive Weights (SAW) method [7] for computing the overall cost/load on a particular node. For this, we need to assign relative weights to system attributes. For compute-intensive jobs, we assign higher weights to attributes like CPU load and CPU utilization. For memory and network-intensive jobs, higher weights are given to available memory and node data flow rate. We define compute load (CL_v) for node v in Equation 1, where *attributes* refer to the set of all attributes as shown in Table 1. w_a represents the relative weight and val_{va} represents the normalised attribute value for node v.

$$\mathrm{CL}_v = \sum_{a \in \text{attributes}} w_a * val_{va} \tag{1}$$

3.2.2 Network Load. We define network load for a pair of connected nodes as the load/traffic on the network that affects the available bandwidth and latency between those nodes. The dynamic attributes considered in this work are effective network bandwidth and latency. It is desirable to select nodes that have low latency and high bandwidth between them. Bandwidth represents maximization criterion while latency represents minimization criterion. We make them both unidirectional (same sign) by complementing bandwidth (i.e. peak bandwidth − available bandwidth). We denote this term as *complement of available bandwidth*. Normalization is done similar to compute load. We define P2P network load $NL_{(u,v)}$ as

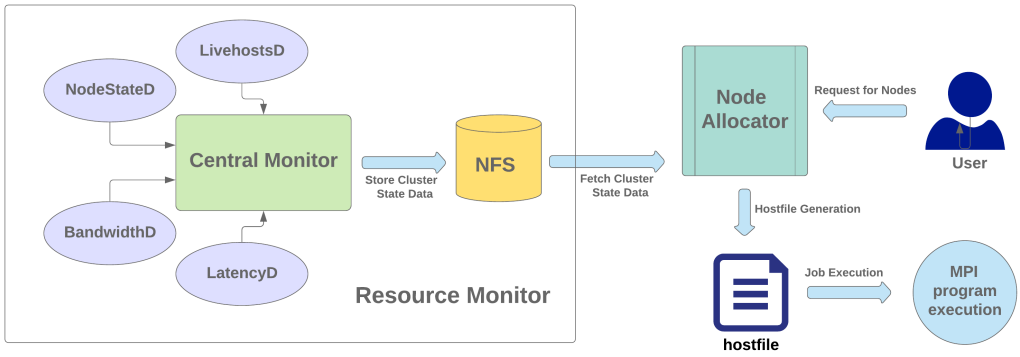


Figure 3: Node Allocation System Workflow

the weighted sum of P2P latency and complement of available P2P bandwidth, as shown in Equation 2. $\mathrm{LT}_{(u,v)}$ and $\overline{\mathrm{BW}}_{(u,v)}$ represent latency and complement of available P2P bandwidth respectively between the nodes u and v. w_{lt}, w_{bw} are the associated weights.

$$\mathrm{NL}_{(u,v)} = w_{lt}\mathrm{LT}_{(u,v)} + w_{bw}\overline{\mathrm{BW}}_{(u,v)} \tag{2}$$

Segregating the two attributes provides flexibility to the user to tune individual weights according to their applications/jobs. For example, consider a program that requires extensive communications, but the communication volume is low. The user would like to execute such a job on a group of nodes with minimum latency. Hence we can set the latency weight (w_{lt}) to be high. If a program requires bulky communications, then the bandwidth term should dominate, and hence w_{bw} would be given higher weightage. We take the average of network load between all pairs of nodes to compute the network load of a group of nodes.

3.3 Node Allocation Policy

Parallel jobs execute on multiple nodes that communicate over the network. User specifies the total number of processes and process count per node (optionally) for execution. An optimal resource allocator should select a group of nodes that meets user's request, minimize communication time based on network load, and maximize parallel program performance. Here we describe our node allocation policy to select a good set of nodes for a particular job.

3.3.1 System Modeling. Node selection problem can be modeled as sub-graph selection problem for a fully connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Each vertex $v \in \mathcal{V}$ represents a compute node in the set $\mathcal{V}$ of all compute nodes. Edges $\mathcal{E}$ represent the set of network links between compute nodes. n represents the number of processes to be allocated and pc_v represents the effective processor count on each node v, i.e. the number of processes that can be allocated on that node based on the node usage by other users. We calculate pc_v using Equation 3. Here, $coreCount_v$ is the logical core count of a node v while $Load_v$ refers to the average CPU load of node v. Note that this value can be overridden by providing process per node (ppn).

$$pc_v = coreCount_v - \lceil Load_v \rceil \,\%coreCount_v \tag{3}$$

We define a vector of effective processor count of nodes, $\mathcal{PC}$ as $\lfloor\, pc_v \mid v \in \mathcal{V} \,\rfloor$. Our node selection problem is to find a sub-graph in $\mathcal{G}$ such that the weighted sum of total compute load and total network load of nodes of the sub-graph is minimized. To find this best sub-graph, one may inspect all the sub-graphs, and take the minimum. However this brute-force search would not scale well. We present a greedy heuristic for sub-graph selection.

3.3.2 Set of Candidate Sub-graphs. We find $|\mathcal{V}|$ candidate sub-graphs. Candidate sub-graph corresponding to a node v is constructed by taking v as the starting node. Then we follow a greedy approach to find other nodes of the sub-graph such that the overall cost/load of the sub-graph is minimized. We add a node to the sub-graph until user request is satisfied and such that the addition of that node incurs minimal network and compute load. We outline the pseudo-code in Algorithm 1.

Let the starting node be v. For each node u in $\mathcal{V}$, other than starting node v, we calculate the cost to add u, represented by $A_v(u) = \alpha \times CL(u) + \beta \times NL(v,u)$ (line 5). Note that, we set the addition cost of starting node $A_v(v)$ to 0 (line 4). Next, we sort the nodes on the basis of $A_v(u)$ (line 6). At each step, we add u with minimum $A_v(u)$ (line 9) until user request is satisfied i.e. when $\sum_{v \in sub-graph} pc_v$ (allocated processes) of selected nodes equals requested process count n (line 8). If the user request is not satisfied after adding all the nodes of $\mathcal{G}$ to the sub-graph (this condition may arise when $\sum_{v \in \mathcal{G}} pc_v < n$), we proceed by allocating rest of the processes in round robin fashion (lines 12–13). Hence, we get a candidate sub-graph that minimizes the overall cost and satisfies user's required number of processes corresponding to node v. We call this sub-graph $\mathcal{G}_v$. We find candidate sub-graph corresponding to each node in the graph making a set of $|\mathcal{V}|$ candidate sub-graphs $C = \{\mathcal{G}_v | v \in \mathcal{V}\}$. The total time complexity of candidate generation algorithm(Algorithm 1) is $O(V \log V)$ where V is the total number of nodes in the cluster. Thus time complexity of generating all candidate sub-graphs is $O(V^2 \log V)$.

Selection of the best candidate: Once we have all candidate sub-graphs, we select the sub-graph with minimal compute and network cost/load. For a given $\mathcal{G}_v = (\mathcal{V}_v, \mathcal{E}_v)$, we define total compute cost/load as $C_{G_v} = \sum_{u \in \mathcal{V}_v} CL_u$. Similarly, we define total network load as $N_{G_v} = \sum_{(x,y) \in \mathcal{E}_v} NL_{(x,y)}$. We normalized the above costs by dividing them by $\sum_{G_v \in C} C_{G_v}$ and $\sum_{G_v \in C} N_{G_v}$ respectively. We define total load/cost of the sub-graph $\mathcal{G}_v$ in Equation 4.

$$T_{G_v} = \alpha \times C_{G_v \text{ Normalized}} + \beta \times N_{G_v \text{ Normalized}} \tag{4}$$

Algorithm 1: Candidate Generation Algorithm

Input : Node v, Graph $\mathcal{G}$, List of effective processor count $\mathcal{PC}$, Requested number of processes n
Output: $\mathcal{G}_v$ sub-graph with v included

$\mathcal{V}_v \longleftarrow \phi$
allocated process count $\longleftarrow 0$
k $\longleftarrow$ Total number of nodes in $\mathcal{G}$
$A_v(v) \longleftarrow 0$
Calculate $A_v(u)$ for each node other than v
Let $u_1, u_2, u_3, \ldots, u_k$ be the vertices sorted in increasing order of addition load $A_v(u)$
i $\longleftarrow 1$
while *allocated process count* $< n$ **do**
  $\mathcal{V}_v \longleftarrow \mathcal{V}_v \cup \{u_i\}$
  allocated process count$\longleftarrow$ allocated process count + pc_{u_i}
  i $\longleftarrow$ i+1
  if $i == k + 1$ **then**
    i $\longleftarrow 1$
end
return $\mathcal{G}_v = (\mathcal{V}_v, \mathcal{E}_v)$

Algorithm 2: Best Candidate Selection Algorithm

Input : Graph $\mathcal{G}$, Set of Candidate Sub-graphs C
Output: Best Candidate Sub-graph $\mathcal{G}_{\text{bestCandidate}}$

1 bestCandidate $\longleftarrow \phi$
2 minTotalLoad $\longrightarrow \infty$
3 **foreach** $\mathcal{G}_v \in C$ **do**
4 Calculate C_{G_v}, N_{G_v} and T_{G_v}
5 **if** $T_{G_v} <$ *minTotalLoad* **then**
6 bestCandidate $\longleftarrow v$
7 minTotalLoad $\longleftarrow T_{G_v}$
8 **end**
9 **return** $\mathcal{G}_{\text{bestCandidate}}$

Here, α is set high for jobs that are compute-intensive and β is set high for jobs that are communication-intensive. These weights can be provided by user while requesting for node allocation. Note that $\alpha + \beta = 1$. We outline the pseudo-code in Algorithm 2. First we compute T_{G_v} for each candidate sub-graph (line 4). We select $\mathcal{G}_v$ that has the minimum T_{G_v} and is allocated for the job (lines 5–7). The time complexity for selecting the best candidate is $O(V(\frac{n}{ppn})^2)$ where V is the total number of nodes in the cluster, n is the requested number of process and ppn is process per node. The total run-time of whole algorithm (Algorithm 1 and Algorithm 2) is ~1-2 ms. Thus our algorithm leads to practically nil overhead. However, our solution may need to be adapted for larger scale by grouping the nodes based on cluster topology and calculating inter-group bandwidth/latency so that P2P bandwidth/latency calculation requires less amount of communication. Next, we describe how we monitor resources and collect current node states such as CPU load and network usage.

4 RESOURCE MONITORING

We now describe our monitoring tool that collects static and dynamic data of cluster that is required by node allocation algorithm. We have implemented our own efficient light-weight monitoring system as it provides more flexibility and does not require any admin access. We can also use other cluster monitoring tools. Resource Monitor component of Figure 3 gives an overview of our resource monitoring tool. We use daemons that run on the cluster nodes in a distributed manner. These daemons periodically collect the node and network status, such as which nodes are reachable (connected), the network bandwidth, etc. Each node daemon writes its data to the shared file system, which in our case was the Network File System (as shown in Figure 3). Central Monitor Process keeps track of all other daemons and enables them to continue operating correctly in case of failures.

Livehosts: We run a daemon LivehostsD (see Figure 3) that periodically pings every node in the cluster to check whether the node is up or not. We maintain a list of active nodes, called livehosts. This ensures that only the nodes that are up are allocated for user's job. We run this daemon on a few selected nodes at different frequencies in the cluster to ensure fault tolerance and to get real-time data regarding all active nodes.

Node state: We require the values of various static and dynamic attributes of the livehosts in the cluster to deduce the compute load of each node. For this, we run a node state daemon called NodeStateD (see Figure 3) on each livehost. The daemon extracts the real-time system state using Unix system utilities such as `lscpu` and `uptime` and the `psutil` library [6]. We query the node once and maintain the static attributes of a node such as the core count, CPU speed, and the total physical memory. We measure dynamic node attributes such as CPU load, CPU utilization, bandwidth usage, memory usage, and the count of connected users on a node periodically. These daemons are lightweight and extracts data every 3–10 seconds. Further, the daemons keep track of the running mean of the last 1, 5, and 15 minutes of historical data of dynamic attributes. This allows our allocator to make a more informed selection. Node data flow rate is obtained from the total data received and transmitted data through the node network interface using the `psutil` library network statistics function.

P2P latency and bandwidth: We calculate the P2P latency and the effective bandwidth to estimate network performance of the cluster using LatencyD and BandwidthD (see Figure 3). We run an MPI program at regular intervals of 1 minute for latency and 5 minutes for bandwidth to measure pairwise latencies and bandwidths. We distribute these P2P calculations on all nodes such that each node only calculates its own latency/bandwidth with all other nodes. Furthermore we schedule these P2P calculations in a few rounds such that one node communicates with only one other node in each round (n/2 distinct pairs of nodes communicate at a time). There are n-1 such rounds of communications. We maintain average of last 1 and 5 minutes of P2P latency and use this in our algorithm. We use the instantaneous effective bandwidths to allocate the best set of nodes (i.e. which have a high bandwidth among them).

Central Monitor: Central Monitor launches, supervises and removes LivehostsD, NodestateD, BandwidthD and LatencyD daemons (see Figure 3). The monitor ensures that all daemons are up and running.

If any daemon crashes, it is relaunched on appropriate nodes. We keep one master and one slave instance of Central Monitor to avoid single point of failure. If the master process dies, the slave will detect that the process is dead. The slave will become new master and launches a new slave on another node. If slave dies, master launches a new slave on another node. If Central Monitor (both master and slave) stops (due to network failure or simultaneous failure of both master and slave), all other daemons will still continue to perform their job. However, the other daemons won't be restarted in case of failure.

5 EXPERIMENTS AND RESULTS

Now we present performance evaluations of network and load-aware allocation algorithm. We compare network and load-aware allocation algorithm with *random*, *sequential* and *load-aware* allocation. *Random* allocation randomly selects the required number of nodes from active nodes. *Sequential* allocation first selects a random node and adds neighboring nodes (topologically) as required. This is because users often tend to select consecutive nodes. *Load-aware* allocation selects the group of nodes with minimal load.

We experimented on the Intel cluster of the Department of Computer Science and Engineering, Indian Institute of Technology Kanpur. The cluster has a tree-like hierarchical topology with 4 switches. Each switch connects 10–15 nodes using Gigabit Ethernet. Our experimental setup consists of 40 12-core Intel Core nodes (4.6 GHz) and 20 8-core Intel Core nodes (2.8 GHz). We used two Mantevo benchmarks, miniMD [4] and miniFE [3] for performance evaluation of network and load-aware allocation algorithm. MiniMD is a simple, parallel molecular dynamics (MD) mini-application that uses spatial decomposition, where each process owns subsets of the simulation box whose size (s) can be specified by the user. MiniFE is an proxy application for unstructured implicit finite element codes which sets up a brick-shaped problem domain of hexahedral elements. MiniFE parameters nx, ny, nz specify global number of elements in each dimension.

We used weights of 0.3, 0.2, 0.2, 0.1, 0.1, 0.05 and 0.05 for CPU load, CPU utilization, node bandwidth, used memory, logical core count, CPU clock speed and total physical memory respectively to calculate compute load (Equation 1). We gave higher weights to CPU load and utilization because higher values of these reduces performance. Most systems have 16 GB memory, and only 25% memory was used on average. Hence, we gave lower weight to memory. Systems attributes like total physical memory, clock speed and logical core count do not vary much across nodes, hence were also given lower weights. The values of w_{lt} and w_{bw} were set to 0.25 and 0.75 respectively (Equation 2). We set α and β to 0.3 and 0.7 respectively (Equation 4) for MiniMD, and 0.4 and 0.6 respectively (Equation 4) for MiniFE. These values were determined empirically. One may set these weights by profiling an application and decide the relative weights on the basis on the computation and communication times. Some knowledge about the application may also help to set these relative weights. Through a few profiling runs, we noted that the percentage of communication time was higher for miniMD (40-80%) than for miniFE (25-60%). Thus, more weight was given to network load in case of miniMD.

5.1 Strong Scaling: miniMD

Figure 4 shows miniMD execution times (y-axis) on 8 – 64 processes with 4 processes/node. We varied the problem size (s) from 8 to 48 uniformly distributed at an interval of 8 (2K – 442K atoms), as shown in the x-axis. We ran all four approaches in sequence for fair evaluation, and repeated this for 5 times to account for network variability. Each data point in the figure is the average of 5 runs. We can observe that random allocation performs worst on almost all configurations. This is because the probability of running on highly loaded nodes is high with random allocation. Load-aware performed better than sequential for less number of nodes whereas worse for a large number of nodes. This is because when the node count is high, network dynamics impact the communication times more and hence the performance. In network and load-aware allocation algorithm, we incorporated these network dynamics and thus achieve the best performance. We observed an increase in execution time for a given problem size on increasing the number of nodes. Also, there is saturation at high atom count in $\#procs = 64$. These can be attributed to increased communication overhead as number of nodes increases. In this case communication time dominates computation time and the execution time is mainly determined by communication time.

Our experiments show significant performance gains in terms of execution times compared to other allocation algorithms for a given parameter. We summarize the improvements over random, sequential and load-aware allocation in Table 2. Our algorithm performs better than random by 49.9%, sequential by 43.1%, and load-aware by 32.4% on an average. The maximum performance improvements (last column) are more than 84% in all cases. The

Table 2: Percentage gain in performance of network and load-aware allocation algorithm for miniMD executions.

Allocation Policy	Average Gain	Median Gain	Maximum Gain
Random	49.9%	50.7%	87.8%
Sequential	43.1%	42.1%	84.5%
Load-Aware	32.4%	29.8%	87.7%

reason for this significant performance gain is that the network and load-aware algorithm selects those nodes that have low CPU load and utilization and good network connectivity (high bandwidth and low latency) among them. We also observed that the average CPU load per logical core for network and load-aware allocation algorithm, load-aware allocation, sequential and random are 0.43, 0.31, 0.68 and 0.72 respectively (see Figure 5). Note that network and load-aware allocation strategy performs better than load-aware (see Figure 4) despite more average CPU load per logical core. This clearly shows the benefit of good network connectivity between the nodes. We also noted that the coefficient of variation (i.e ratio of standard deviation to mean) for runs of network and load-aware allocation algorithm was 0.07 as compared to 0.13 for load-aware and 0.27 for sequential allocation. This shows that network and load-aware allocation algorithm was indeed able to select a stable set of nodes.

5.2 Strong Scaling: miniFE

Figure 6 shows results for various configurations of miniFE. We varied the problem dimension (nx) from 48 to 384 (48, 96, 144, 256, 384),

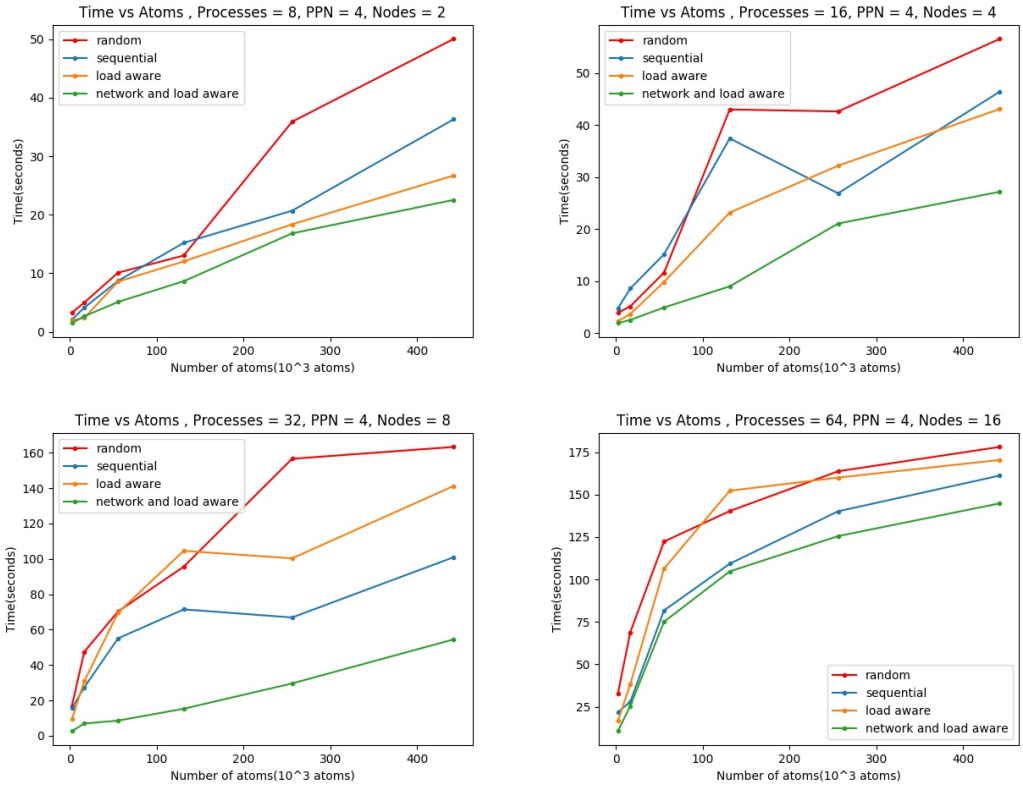


Figure 4: Comparison of network and load-aware allocation with random, sequential and load-aware node allocations for miniMD executions on 8, 16, 32, 64 processes.

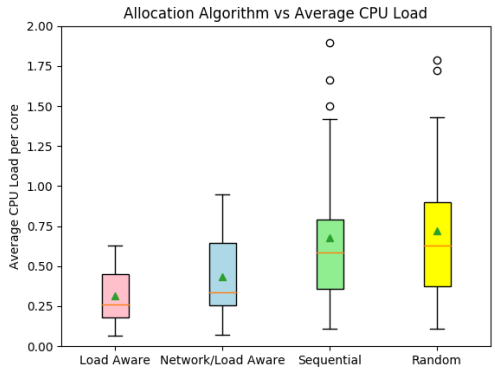


Figure 5: Average CPU load per logical core for the allocation algorithms across several runs of miniMD (corresponding to Figure 4)

and fixed nz and ny equal to nx. For each of these problem configurations, we experimented on 8 – 48 process counts with 4 processes per node (corresponding to each sub-plot). We ran all the four approaches in sequence, and repeated this for 5 times to account for network variability. Each data point in the figure is the average of 5 runs. The plots changes from concave upwards (#*procs* = 8) to linear (#*procs* = 32) to concave downwards (#*procs* = 48). This is due to slight decrease in execution times for larger problem sizes (*nx*) and longer execution times for smaller problem sizes with increasing number of processes. This may be due to more overall communication for large number of nodes. If we increase nodes for small *nx*, the communication overhead is comparatively more than that for large problem sizes. Our experiments show lesser communication times for miniFE as compared to miniMD. We noted about 40% time was spent in miniFE communications when using 48 processes, while the same was more than 50% in case of miniMD. Since our resource broker accounts for both compute and network costs, and we set the weights appropriately for these applications, our network and load-aware algorithm selects the best set of nodes. Thus we obtain significant performance gains in terms of execution time. We note that network and load-aware allocation algorithm performs better than random by 48.1%, load-aware by 34.4%, and sequential by 31.1% on an average. The maximum performance improvements with respect to random, sequential and load-aware allocation are 92.1%, 89.5% and 80.4% respectively (see Table 3). We also noted that the coefficient of variation (i.e ratio of standard deviation to mean) for runs of network and load-aware allocation algorithm was 0.05 as compared to 0.08 for load-aware and 0.11 for sequential allocation.

Table 3: Percentage gain in performance of network and load-aware allocation algorithm for miniFE executions

Allocation Policy	Average Gain	Median Gain	Maximum Gain
Random	47.9%	50.4%	92.1%
Sequential	31.1%	28.0%	80.4%
Load-Aware	34.8%	38.7%	91.0%

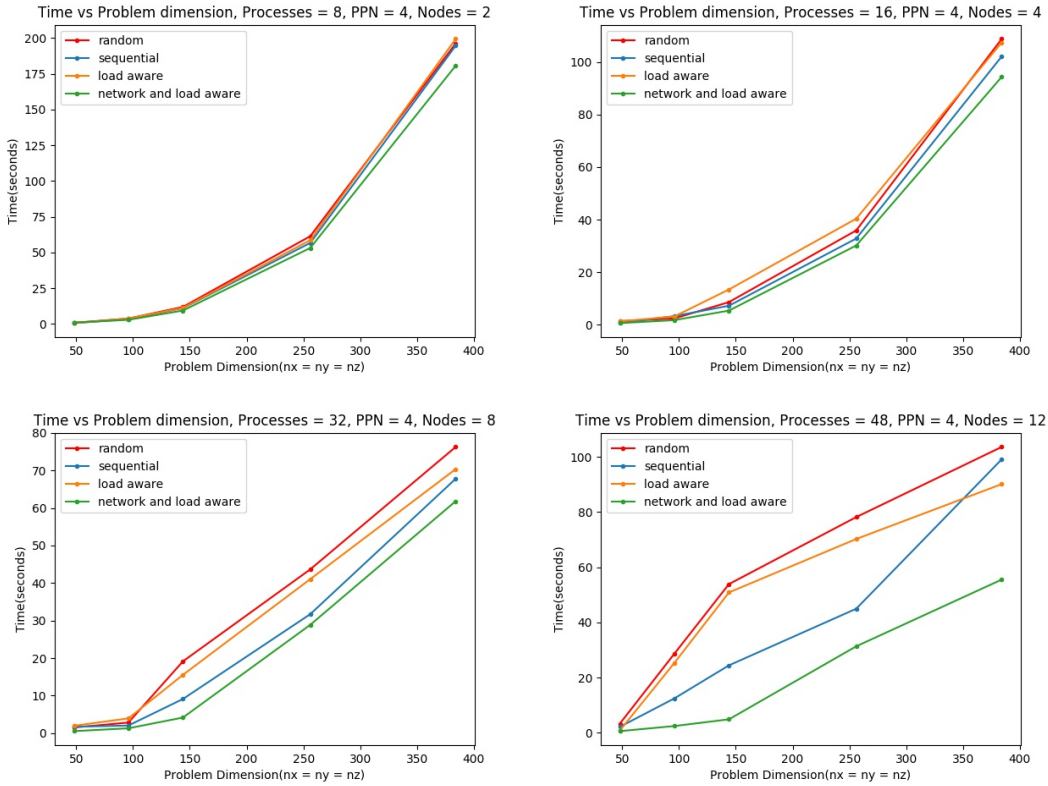


Figure 6: Comparison of network and load-aware allocation with random, sequential and load-aware resource allocations for miniFE executions on 8, 16, 32, 48 processes.

5.3 Resource Allocation Analysis

We analyze the utility of our network and load-aware algorithm in detail using the results from miniMD run on 32 processes (4 processes/node) and problem size (s)=16 (16K atoms). miniMD was run on four sets of 8 nodes using all the above mentioned four allocation methods. The experiment took 4.43s to complete when run on nodes allocated by network and load-aware allocation. The execution time using the nodes allocated by load-aware, sequential and random were 12.31s, 24.91s and 27.61s respectively. This performance gain can be explained using the current state of allocated nodes using different algorithms.

Table 4: Usage of allocated resource group during allocation

Algorithm	Avg. CPU load	Avg. bandwidth	Avg. latency
Random	1.242	17.07	546.46
Sequential	1.262	10.72	304.25
Load Aware	0.453	18.64	354.51
Network and load-aware	0.633	5.36	82.90

Table 4 columns 2, 3 and 4 show respectively the CPU load averaged over 8 nodes, complement of available bandwidth (see Section 3.2.2) and latency averaged over all P2P links in 8 node sub-network allocated by the algorithms (column 1). We observe that the nodes allocated according to network and load-aware allocation algorithm have low average complement of available bandwidth which denotes high available bandwidth for the job. The nodes also have low average latency and average CPU load compared to the nodes allocated by other allocation algorithms. Network and load-aware allocation algorithm performed better than load aware despite of slightly higher CPU load. This signifies the importance of network connectivity between allocated nodes.

Figure 7 provides more insight about state of cluster at the time of allocation. There are three parts to the figure. At the top, it has a heatmap of available P2P bandwidth. The first row indicates the node number (csews1, csews4, etc.). Darker shade implies lower P2P available bandwidth (as indicated by a larger number representing complement of available bandwidth) and lighter shade implies higher P2P available bandwidth for a job. These nodes are spread over three switches, separated by a vertical bold blue line. The middle part of the figure shows the nodes selected by different algorithms. The nodes that were selected are colored. The bottom row of the figure shows CPU load of each node during allocation.

Observe that nodes numbered *csews9* and *csews10* were not selected by load aware algorithm because they are slightly loaded. These nodes despite being loaded, were selected by network and load-aware allocation algorithm because they have good network connectivity with the other nodes selected by the algorithm. Note that network and load-aware allocation algorithm did not select *csews4* as it has higher CPU load and would have severely affected the application performance. We can observe that network and load-aware algorithm automatically captures topology as it has selected nodes which are topologically close to each other thus reducing number of hops while considering the load on the sub-network of

NODES	csews1	csews4	csews5	csews6	csews8	csews9	csews10	csews12	csews13	csews15	csews16	csews19	csews20	csews23	csews28	csews32	csews50	csews51	csews54
csews1	0	50	17	15	10	22	10	23	68	11	16	69	56	44	30	70	41	36	38
csews4	50	0	29	24	14	12	25	21	32	10	9	80	71	45	29	73	61	67	52
csews5	17	29	0	29	11	18	14	13	51	7	7	70	61	53	35	55	37	52	39
csews6	15	24	29	0	15	15	9	9	55	21	31	75	65	48	38	59	41	55	51
csews8	10	14	11	15	0	23	18	6	56	9	12	81	64	57	27	61	33	60	53
csews9	22	12	18	15	23	0	11	5	50	9	6	75	37	40	55	57	22	55	56
csews10	10	25	14	9	18	11	0	5	44	22	17	66	68	53	43	61	37	55	52
csews12	23	21	13	9	6	5	5	0	32	6	9	71	64	52	43	71	41	60	53
csews13	68	32	51	55	56	50	44	32	0	43	50	11	10	8	43	10	52	69	54
csews15	11	10	7	21	9	9	22	6	43	0	10	65	66	54	39	45	50	65	54
csews16	16	9	7	31	12	6	17	9	50	10	0	44	78	57	61	60	59	59	57
csews19	69	80	70	75	81	75	66	71	11	65	44	0	13	29	11	10	48	72	38
csews20	56	71	61	65	64	37	68	64	10	66	78	13	0	14	8	10	49	69	47
csews23	44	45	53	48	57	40	53	52	8	54	57	29	14	0	10	24	43	77	40
csews28	30	29	35	38	27	55	43	43	43	39	61	11	8	10	0	15	44	58	29
csews32	70	73	55	59	61	57	61	71	10	45	60	10	10	24	15	0	47	72	38
csews50	41	61	37	41	33	22	37	41	52	50	59	48	49	43	44	47	0	5	13
csews51	36	67	52	55	60	55	55	60	69	65	59	72	69	77	58	72	5	0	9
csews54	38	52	39	51	53	56	52	53	54	54	57	38	47	40	29	38	13	9	0

Our	csews1	csews4	csews5	csews6	csews8	csews9	csews10	csews12	csews13	csews15	csews16	csews19	csews20	csews23	csews28	csews32	csews50	csews51	csews54
Load	csews1	csews4	csews5	csews6	csews8	csews9	csews10	csews12	csews13	csews15	csews16	csews19	csews20	csews23	csews28	csews32	csews50	csews51	csews54
Seq	csews1	csews4	csews5	csews6	csews8	csews9	csews10	csews12	csews13	csews15	csews16	csews19	csews20	csews23	csews28	csews32	csews50	csews51	csews54
Random	csews1	csews4	csews5	csews6	csews8	csews9	csews10	csews12	csews13	csews15	csews16	csews19	csews20	csews23	csews28	csews32	csews50	csews51	csews54
Load	0.68	5.19	0.58	0.68	0.66	1.08	0.71	0.24	0.78	0.56	0.55	0.7	0.38	1.1	0.62	0.54	0.43	0.35	0.42

Figure 7: Peer-to-peer bandwidth and CPU Load.

selected nodes also. The load-aware algorithm selected nodes across various switches (csews5, csews8, csews12 from switch 1, csews20, csews32 from switch 2 and csews50, csews51, csews54 from switch 3) that results in poor network connectivity among the selected nodes. Sequential algorithm selected nodes (csews4, csews9) with high load. Random algorithm selected nodes that are highly loaded (csews5) as well as nodes (csews19, csews32) that have very poor network connectivity with other selected nodes (such as csews1, csews4). The network and load aware allocation algorithm is aware of the compute and network load of the available nodes. Thus it outperformed random, sequential, and load-aware allocation algorithms and provided a stable set of nodes for execution.

6 CONCLUSIONS AND FUTURE WORK

We presented an approach for network and load-aware node allocation for MPI jobs. We considered dynamic as well as static attributes of the system simultaneously such as the CPU and network usage. Our algorithm reduces runtimes by more than 38% over random, sequential, and load-aware allocations due to less interference from external factors. If the overall load on the cluster is extremely high, the performance gain will not be significant because there are not enough lightly loaded processors; in that case, our tool should recommend waiting rather than allocating it right away. The current algorithm considers allocation in only a single cluster. For a large department/institute that may span over multiple clusters, we need to consider the large overheads between nodes from different clusters. We plan to extend the algorithm to scale to large-scale distributed systems. Also, it is challenging to determine the relative weights for resource attributes and computation-communication characteristics for large applications. We plan to enhance profiling tools for this purpose. We also intend to explore integrating our tool as a plugin for SLURM job scheduler.

REFERENCES

[1] 2018. *Moab Workload Manager.* http://docs.adaptivecomputing.com/9-1-3/MWM/moab.htm

[2] 2019. *Hydra Process Manager.* https://wiki.mpich.org/mpich/index.php/Hydra_Process_Management_Framework

[3] 2020. *miniFE.* https://github.com/Mantevo/miniFE

[4] 2020. *miniMD.* https://github.com/Mantevo/miniMD

[5] 2020. *MPICH.* https://www.mpich.org

[6] 2020. *psutil library.* https://psutil.readthedocs.io

[7] Lazim Abdullah and C. W. Rabiatul Adawiyah. 2014. Simple Additive Weighting Methods of Multi criteria Decision Making and Applications: A Decade Review.

[8] M. Alicherry and T. V. Lakshman. 2012. Network aware resource allocation in distributed clouds. In *2012 Proceedings IEEE INFOCOM.* 963–971.

[9] Jean-Pierre Brans and Bertrand Mareschal. 2005. *Promethee Methods.* Springer New York, 163–186.

[10] A. Burns. 1991. Scheduling hard real-time systems: a review. *Software Engineering Journal* (1991), 116–128.

[11] Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, and Adèle Villiermet. 2018. Topology-aware job mapping. *The International Journal of High Performance Computing Applications* 32, 1 (2018), 14–27.

[12] Mandeep Kaur and Sanjay S. Kadam. 2017. Discovery of resources using MADM approaches for parallel and distributed computing. *Engineering Science and Technology, an International Journal* 20, 3 (2017), 1013 – 1024.

[13] Benjamin Klenk and Holger Fröning. 2017. An Overview of MPI Characteristics of Exascale Proxy Applications. In *High Performance Computing.* Springer International Publishing, 217–236.

[14] B. Posey, A. Deer, W. Gorman, V. July, N. Kanhere, D. Speck, B. Wilson, and A. Apon. 2019. On-Demand Urgent High Performance Computing Utilizing the Google Cloud Platform. In *IEEE/ACM HPC for Urgent Decision Making (UrgentHPC).*

[15] Igor Sfiligoi, Frank Wuerthwein, Benedikt Riedel, and David Schultz. 2020. Running a Pre-Exascale, Geographically Distributed, Multi-Cloud Scientific Simulation. In *International Supercomputing Conference.*

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2005. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience* 17, 2-4 (2005), 323–356.

[17] Rich Wolski, Neil T Spring, and Jim Hayes. 1999. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems* 15, 5 (1999), 757 – 768.

[18] C-T Yang and et al. 2007. A Grid Resource Broker with Network Bandwidth-Aware Job Scheduling for Computational Grids. In *Advances in Grid and Pervasive Computing.*

[19] 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.