Network and Load-aware Node Allocator for MPI Programs

Ashish Kumar*†‡, Naman Jain*†§, Preeti Malakar†¶
†Department of Computer Science and Engineering, IIT Kanpur
‡akashish@iitk.ac.in, §namanj@iitk.ac.in, ¶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 40% on average as compared to the default.**

I. Introduction

Message Passing Interface (MPI) is primarily used to write distributed-memory parallel programs. The MPI library (e.g. MPICH [1]) takes care of communication setup and messaging required to run parallel codes on multiple nodes. The user is expected to specify a list of hostnames while executing an MPI job in 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 [3], 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 the cluster exclusively runs MPI jobs or runs jobs using a process manager only.

The above assumption may not hold true in case of compute lab clusters at academic institutions, as shown in Figure 1. It shows the CPU load (i.e. number of processes waiting to be executed) and network usage of 2 distinct nodes (randomly chosen) as well as the average across all nodes in a cluster in the Indian Institute of Technology (IIT Kanpur) over a period of 2 days. The rightmost subfigure shows the average CPU utilization and memory usage. Note that the metrics vary a lot across nodes. This is because the cluster is shared across many users (mainly students of different departments). Many times, the overall cluster utilization is low, and hence serves as a testbed cluster to run parallel jobs that require multiple nodes. However, the programmer may not be aware of resource usage variation (for e.g., note the peaks in CPU load and network usage). We address the problem of allocating a good set of nodes to run MPI jobs in a shared cluster with variable resource usages. We do not assume that MPI jobs are exclusively running on the cluster.

Cluster resources may also vary in their software and hardware configurations. For example, a cluster may have a few 4-core and a few 8-core machines. In this case, the MPI library does not distinguish between the compute capacities of these nodes, but instead relies on the user's knowledge of the cluster configuration. This can hurt the 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. In this scenario of varying hardware configurations and varying resource usages of the nodes, it is challenging to match jobs to the appropriate resources. In fact, the optimal task allocation is an NP-hard problem [4]. We propose a resource matchmaker, whose task is to collect information about the resources and provide suitable resources to the job.

Static attributes like RAM, network bandwidth, CPU speed do not provide enough information about the suitability of a resource. We also need to consider the dynamic attributes of the resource like CPU load, available memory, available bandwidth. MPI programs are usually communication-intensive, therefore it is crucial to consider the effective bandwidth between allocated nodes. Many existing matchmakers [5] do not consider network state while various others [6] consider the network topology but not current state of network between the compute nodes. There may be contention and congestion in the network due to existing jobs, which may decrease the current network availability between nodes. We propose a technique that takes into account both static and dynamic attributes of resources, simultaneously considering network availability of the allocated nodes. First, we find a few candidate groups of resources that satisfy the job requests using a greedy algorithm. Then, we calculate the computation and network cost of each group and the group with the minimum weighted sum of computation and network load/cost is

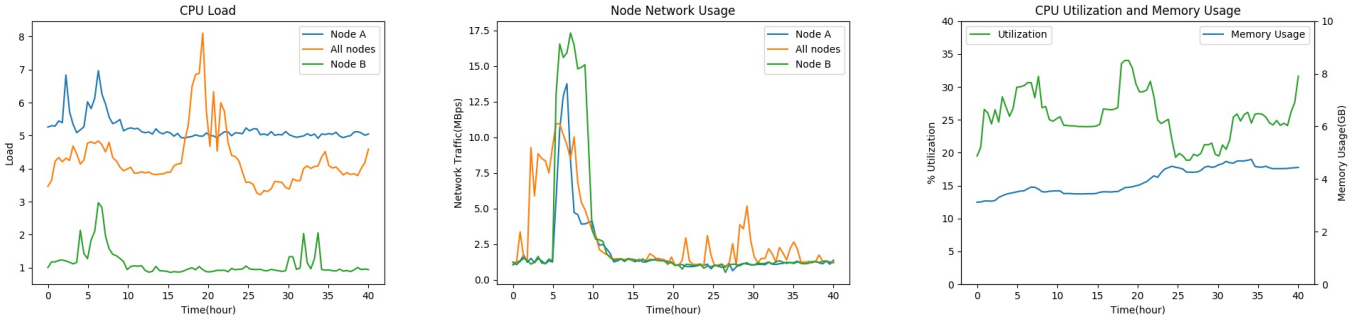

Fig. 1: Variation in resource usage (CPU Load, Node Network Usage, and CPU Utilization and Memory Usage)

allocated to the job. We observed an average of 32 – 49% improvement over various resource allocation strategies.

II. Related Work

There has been a lot of study on efficient resource allocation. Kaur et al. [7] proposed a two-phase ranking approach that considers static and dynamic resource attributes for resource selection. They generate a ranking based on pairwise comparison of resources, using PROMETHEE-II [8], AHP (Analytic Hierarchical Process) and SAW [9] methods. However, they only consider computation power but not the network availability among the resources, which is crucial for MPI. Yang et al. [10] 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.

Condor [5] provides parallel universe for execution of MPI jobs. Users may specify requirements that has to be met and ranking criterion of resources. The matchmaker will select the top nodes based on their ranks. The ranking criterion is limited to local node attributes. Network Weather Service [11] 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. In contrast, we use a composite metric for CPU and network load. Alicherry et. al. [12] presented a selection algorithm for VM placement in distributed cloud, to minimize the maximum distance between the VMs. They modeled the problem in the form of sub-graph selection problem and they proposed the Min-Star algorithm for data center selection. We used a similar approach for node selection.

III. Node Allocation Algorithm

We describe our node allocation algorithm in this section. We first describe the system parameters such as computation load and network load that affect the overall performance of codes, and hence are used in our algorithm. *Compute Load*: We define the compute load as the overall load of a specific node. We determine the compute load based on various dynamic node attributes, such as the CPU load, CPU utilization, memory usage and the available network bandwidth at a node along with static attributes of a node such as the CPU clock speed and the total memory. 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. High CPU load and high CPU utilization imply that processing resources are busy executing other jobs, and hence are less favorable to run our jobs. In our work, the CPU utilization and CPU load represents the aggregate value across all logical cores. Node bandwidth refers to the amount of data transmitted and received by a node over a period of time.

First, the attribute values of each node are normalized. CPU utilization represents a minimization criterion, while available memory represents a maximization criterion. We convert all the attributes in unidirectional units. This is done by complementing (with respect to the maximum value). For computing the overall cost/load on a particular node, we follow Simple Additive Weights (SAW) method [9]. For compute-intensive jobs, we provide more weight to attributes like CPU load and CPU utilization. For memory and network-intensive jobs, more weight is given to available memory and node bandwidth respectively. We define compute load (CL_v) for node v in Equation 1, where $attributes$ refer to the set of all attributes. w_a represents the relative weight and val_{va} represents the normalised attribute value for node v.

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

Network Load: We define network load for peer-to-peer (P2P) nodes as the load/traffic on the network that affects the network connectivity between those nodes. We use

SAW to compute P2P network load. Effective network bandwidth and latency are the dynamic attributes that we consider. Since it is desirable to achieve low latency and high bandwidth, we make them unidirectional to make it minimization criterion. The bandwidth attribute gives the estimate of used bandwidth between two nodes. Normalization is done similar to compute load. We define P2P network load $NL_{(u,v)}$ as the weighted sum of P2P latency and P2P used bandwidth, as shown in Equation 2. $\text{LT}_{(u,v)}$ and $\text{BW}_{(u,v)}$ represent latency and used bandwidth respectively between the nodes u and v. w_{lt}, w_{bw} are the associated weights.

$$\text{NL}_{(u,v)} = w_{lt}\text{LT}_{(u,v)} + w_{bw}\text{BW}_{(u,v)} \quad (2)$$

Segregating the attributes provides flexibility to the user to tune individual weights according to their applications/jobs. For e.g., 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. To compute the network load of a group of nodes, we take the average of all P2P network load.

Node Allocation Policy: Parallel jobs execute on multiple compute nodes that communicate over the network. User specifies the total number of processes for execution and process count per node. An optimal resource allocator should select a group of nodes that meets user's request, optimize network usage, and maximize parallel program performance. Here, we present our node allocation policy to select a good set of nodes for a particular job. Next, we present a greedy algorithm for the same.

This problem can be modeled as a 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). We define $\mathcal{PC}$ as $\left[\, pc_v \mid v \in \mathcal{V} \,\right]$.

$$pc_v = coreCount_v - Load_v \,\%\, coreCount_v \quad (3)$$

Our node selection problem is to find a sub-graph in $\mathcal{G}$ such that the weighted sum of total computation load and total network load of nodes of the sub-graph is minimized. We outline the pseudo-code in Algorithm 1. To find this best sub-graph, one may inspect all the sub-graphs, and take the minimum but this brute-force search would not scale well and would be very time-consuming. We present a greedy heuristic for sub-graph selection. We find $|\mathcal{V}|$ candidate sub-graphs. Candidate sub-graph corresponding to a node v is constructed by taking v as the starting node. Once we have a starting node v, 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 computation load.

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). In case 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. Now, we have a set of candidate sub-graphs $\mathcal{C} = \{\mathcal{G}_v | v \in \mathcal{V}\}$. We have $|\mathcal{V}|$ such candidate sub-graphs. Once we have all the candidate sub-graphs, we select the sub-graph with minimal computation and network

Algorithm 1: Candidate Selection Algorithm

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

1 $\mathcal{V}_v \longleftarrow \phi$
2 allocated process $\longleftarrow$ 0
3 k $\longleftarrow$ Total number of nodes in $\mathcal{G}$
4 $A_v(v) \longleftarrow 0$
5 Calculate $A_v(u)$ for each node other than v
6 Let u_1, u_2, u_3, ... , u_k be the vertices sorted in increasing order of addition load $A_v(u)$
7 i $\longleftarrow$ 1
8 **while** *allocated processes* $< n$ **do**
9 $\quad \mathcal{V}_v \longleftarrow \mathcal{V}_v \cup \{u_i\}$
10 $\quad$ allocated process $\longleftarrow$ allocated process + pc_{u_i}
11 $\quad$ i $\longleftarrow$ i+1
12 $\quad$ **if** $i == k+1$ **then**
13 $\quad\quad$ i $\longleftarrow$ 1
14 **end**

cost/load. For a given $\mathcal{G}_v = (\mathcal{V}_v, \mathcal{E}_v)$, we define total computation 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.

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

Here, α is set high for jobs that are compute-intensive and β is set high for jobs that are communication-intensive. Note that $\alpha + \beta = 1$. We select $\mathcal{G}_v$ that has the minimum T_{G_v} and is allocated for the job. The total runtime of this algorithm is $\sim$1-2 ms and its time complexity is $\mathcal{O}(V^2 \log V)$ where V is the total number of nodes in the cluster. Next, we describe how we monitor the resources and collect current node states such as CPU load.

IV. Resource Monitoring

We have implemented our own monitoring system as it provides more flexibility. We can also use other cluster monitoring tools. We use daemons that run on the cluster nodes in a distributed manner. These daemons periodically collect the node and network status such as whether the nodes are up or not, what is the network bandwidth, etc. Each node daemon writes its data to the shared file system, which in our case was the Network File System.

Live hosts: We run a daemon 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 live hosts. 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 live hosts.

Node state: We need the values of various static and dynamic attributes of the live hosts in the cluster to deduce the compute load of each node. For this purpose, we run a node state daemon on each live host. The daemon extracts the real-time system state using Unix system utilities and the `psutil` library [13]. 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 typically very lightweight and extracts data around every 3–10 seconds. Furthermore, the daemons keep track of the running mean of the last 1, 5, and 10 minutes of historical data of dynamic attributes which allows our allocator to make a more informed selection. For node bandwidth usage, we consider the total data received and transmitted data through the node network interface. We used the `psutil` library network statistics function for this.

P2P latency and bandwidth: We calculate the P2P latency and the effective bandwidth to estimate network performance of the cluster. The daemon on each node runs an MPI program at regular intervals to measure pairwise latencies and bandwidths. We distribute these P2P calculations on all node daemons such that each node only calculates its latency/bandwidth with all other nodes. We maintain average of last 1 and 5 minutes P2P latency and use this in our algorithm. We group the nodes using the network topology (tree-like in our case), instead of calculating all P2P bandwidths. We then calculate inter-group and intra-group bandwidth to estimate all P2P bandwidths. We use the instantaneous effective bandwidths to allocate the best set of nodes (i.e. which have a high bandwidth among them).

V. Experiments and Results

We compare our 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 used weights of 0.3, 0.2, 0.2, 0.1, 0.1, 0.05, 0.05 for CPU load, CPU utilisation, node bandwidth, used memory, logical core count, CPU clock speed, total physical memory respectively to calculate compute load (Equation 1). We gave higher weights to CPU load and utilization because higher the load and utilization, worse is the performance. Most systems have 16 GB memory, and only 25% memory was used on average. Hence, we gave low weight to memory. Systems attributes like total physical memory, clock speed and logical core count do not vary much across nodes, hence were given less weights. The values of w_{lt} and w_{bw} were set to 0.25 and 0.75 (Equation 2). We set α and β to 0.3 and 0.7 respectively (Equation 4). These values were calculated empirically.

We used 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 one of the Mantevo benchmarks (miniMD [14]). It 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. We varied the problem size (s) from 8 to 48 (2K, 16K, 55K, 131K, 256K, 442K atoms) uniformly distributed at an interval of 8. For each problem size, we experimented with different process counts (8, 32, 64) with 4 processes per node, as shown in

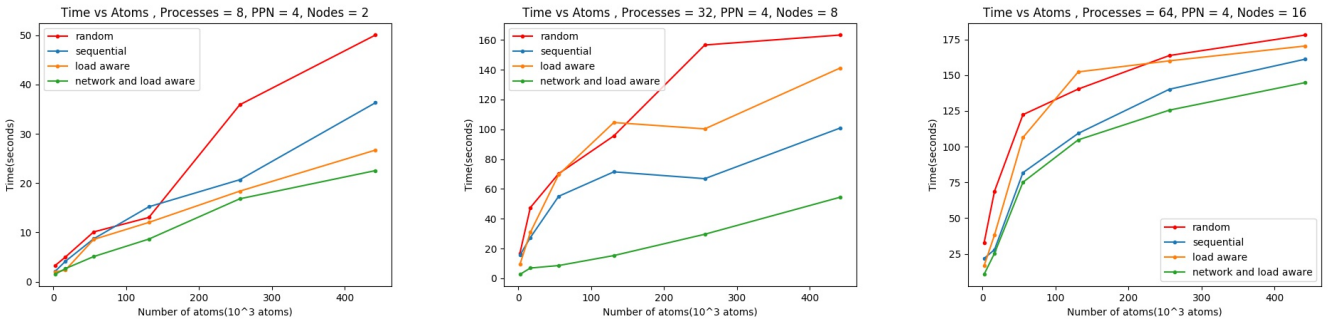

Fig. 2: Comparison of our algorithm (network and load aware) with random, sequential and load-aware resource allocations on 8, 32, 64 processes.

Figure 2. 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.

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 with respect to random, sequential and load-aware allocation were 87.8%, 84.5%, 87.7% respectively. The 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. Random allocation performs the worst because the probability of running on highly loaded nodes is high. 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 our network and load-aware algorithm, we incorporated these network dynamics and thus achieve the best performance. We observed that the average CPU load per logical core for our algorithm, load-aware allocation, sequential and random are is 0.43, 0.31, 0.68 and 0.72 respectively. Our allocation strategy performed better than load-aware (see Figure 2) despite more average CPU load per logical core. This clearly shows the benefit of good network connectivity between the nodes allocated by our algorithm. We also noted that the coefficient of variation (i.e ratio of standard deviation to mean) for runs of our algorithm was 0.074 as compared to 0.13 for load-aware and 0.27 for sequential allocation. This shows that our algorithm was indeed able to select a stable set of nodes.

VI. 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 40% over random, sequential and load-aware allocations due to less interference from external factors. 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 systems. Also, it is challenging to determine the relative weights for resource attributes and computation-communication characteristics for large applications. We plan to use profiling for this. We also plan to test on AMR applications.

References

[1] MPICH. [Online]. Available: https://www.mpich.org

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

[3] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, 2003, pp. 44–60.

[4] A. Burns, "Scheduling hard real-time systems: a review," *Software Engineering Journal*, pp. 116–128, 1991.

[5] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience." *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[6] Y. Georgiou, E. Jeannot, G. Mercier, and A. Villiermet, "Topology-aware job mapping," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 14–27, 2018.

[7] M. Kaur and S. S. Kadam, "Discovery of resources using MADM approaches for parallel and distributed computing," *Engineering Science and Technology, an International Journal*, vol. 20, no. 3, pp. 1013 – 1024, 2017.

[8] J.-P. Brans and B. Mareschal, *Promethee Methods*. Springer New York, 2005, pp. 163–186.

[9] L. Abdullah and C. W. R. Adawiyah, "Simple additive weighting methods of multi criteria decision making and applications: A decade review," 2014.

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

[11] R. Wolski, N. T. Spring, and J. Hayes, "The network weather service: a distributed resource performance forecasting service for metacomputing," *Future Generation Computer Systems*, vol. 15, no. 5, pp. 757 – 768, 1999.

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

[13] psutil library. [Online]. Available: https://psutil.readthedocs.io

[14] miniMD. [Online]. Available: https://github.com/Mantevo/miniMD