This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

SAND2021-10802C

Using Monitoring Data to Improve HPC Performance via Network-Data-Driven Allocation

Yijia Zhang*, Burak Aksar*, Omar Aaziz†, Benjamin Schwaller†,
Jim Brandt†, Vitus Leung†, Manuel Egele* and Ayse K. Coskun*
* Boston University, Boston, MA, USA; E-mail: {zhangyj, baksar, megele, acoskun}@bu.edu
† Sandia National Laboratories, Albuquerque, NM, USA; E-mail: {oaaziz, bschwal, brandt, vjleung}@sandia.gov

***Abstract*—On high-performance computing (HPC) systems, job allocation strategies control the placement of a job among available nodes. As the placement changes a job's communication performance, allocation can significantly affects execution times of many HPC applications. Existing allocation strategies typically make decisions based on resource limit, network topology, communication patterns, etc. However, system network performance at runtime is seldom consulted in allocation, even though it significantly affects job execution times.**

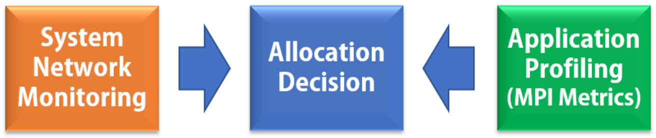
In this work, we demonstrate using monitoring data to improve HPC systems' performance by proposing a Network-Data-Driven (NeDD) job allocation framework, which monitors the network performance of an HPC system at runtime and allocates resources based on both network performance and job characteristics. NeDD characterizes system network performance by collecting the network traffic statistics on each router link, and it characterizes a job's sensitivity to network congestion by collecting Message Passing Interface (MPI) statistics. During allocation, NeDD pairs network-sensitive (network-insensitive) jobs with nodes whose parent routers have low (high) network traffic. Through experiments on a large HPC system, we demonstrate that NeDD reduces the execution time of parallel applications by 11% on average and up to 34%.

***Index Terms*—HPC system, job allocation, network congestion**

I. Introduction

High-performance computing (HPC) systems are playing an irreplaceable role in our society by providing compute resources to support many scientific research and engineering projects. On HPC systems, the communication traffic generated by many simultaneous parallel applications travels on the shared interconnections, often creating network contention that leads to performance degradation. Prior works have observed that network contention on HPC systems is causing significant performance variation as high as 2x [1], 3x [2], 7x [3], or even 8x [4] in terms of delays in job execution times. Therefore, it is important to find new approaches to minimize the impact of network contention and improve job performance.

Job allocation on HPC systems significantly affects the execution times of jobs because placements affect the path and latency of communication. Existing allocation strategies typically make decisions based on resource limit, network topology, communication pattern, etc. [5]–[12]. However, as congestion hot spots in an HPC network changes from time to time, allocation strategies based on those static properties cannot maintain the best performance. Therefore, it is important for an HPC job allocator to be network-data-driven, i.e., to use monitoring data to make decisions. In spite of that, most HPC job allocation strategies does not take network data into consideration, except for a few works that monitor networks or profile job characteristics with a large overhead [13], [14].


System Network Monitoring
Allocation Decision
Application Profiling (MPI Metrics)
Fig. 1: The Network-Data-Driven job allocation framework.

Many HPC systems are equipped with monitoring systems that collect metrics from a variety of hardware performance counters. For example, Cray XC systems offers the ability to monitor hundreds of network counters per router [15]. Multiple HPC systems, including the 12k-node Cori system (LBNL, USA), the 19k-node Trinity system (LANL, USA), and the 28k-node Blue Waters (NCSA, USA), have been running the Lightweight Distributed Metric Service (LDMS) [16] to collect performance metrics from CPU, memory, I/O, network, etc. Some prior works have conducted offline analysis based on the collected performance metrics [4], [17]–[19].

Motivated by the opportunity to use monitoring data to help mitigate congestion, we propose a Network-Data-Driven (NeDD) job allocation framework for HPC systems. NeDD makes allocation decisions based on monitored network performance at runtime. The framework comprises three components, as shown in Fig. 1: (1) a low-cost network monitoring component that quantifies the traffic intensity in each router or link of an HPC system; (2) an application profiling component that determines whether an application is sensitive to network congestion or not; (3) a congestion-aware allocation component that allocates nodes to jobs based on the network and job characteristics with the goal of minimizing congestion. Through experiments on a production HPC system, we evaluate our proposed allocation framework by comparing with other state-of-the-art job allocation strategies. The contributions of this work are listed as follows:

This work has been partially funded by Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under Contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

- We propose a Network-Data-Driven (NeDD) job allocation framework for HPC systems. This framework monitors the network performance of an HPC system at runtime at a fine granularity and places an application based on both the system's network performance and a job's communication characteristics.
- We implement our proposed framework by characterizing network performance using low-cost hardware performance counters and by characterizing a job's sensitivity to network congestion using its MPI operation statistics.
- Through experiments on a large production HPC system, we demonstrate that our proposed framework reduces the execution times of parallel applications by 11% on average and up to 34%.

II. Related Work

We briefly review some latest work by grouping HPC job allocation strategies into five categories as follows:

Strategies based on static properties allocate nodes according to the system's and the job's static properties, including network topology, memory capacity, job size (node count of the job), etc. For example, some methods utilize the row, column, and group structures of dragonfly networks [5], [6]. Some approaches differentiate large-size or small-size jobs to reduce system fragmentation [7].

Strategies based on profiled job characteristics use job characteristics such as communication graphs and message sizes to make decisions. For example, Soryani's work places computational tasks with larger message sizes closer to each other [8]. Michelogiannakis' work and Yan's work construct weighted communication graphs of applications and place the tasks with larger weights closer [9], [10].

Strategies based on thermal and energy constraints focus on meeting a thermal or power constraint. For example, Cheng's work schedules and migrates workload among distributed data centers based on green energy supply [20]. Cao's work optimizes system throughput under power and cooling constraints [21].

Strategies based on other jobs' placement consult the placement of other jobs so as to reduce network interference. For example, Pollard's work places a job to the smallest topological level in a fat-tree network and aims to avoid sharing a switch among different jobs [11]. Zhang's work selects the lowest topological level in a dragonfly network that a job fits in and places the job in a spread manner within that level to reduce congestion [12].

Strategies based on system performance status collect system performance status at runtime and make allocation decisions to balance the CPU, memory, storage, or network resource usage. For example, Werstein's work compares the CPU load on each node when allocating nodes to jobs [22]. LaCurts' work measures the network throughput between every pair of nodes in a system and prioritizes mapping communication-heavy tasks of an application to high-throughput node pairs [13].

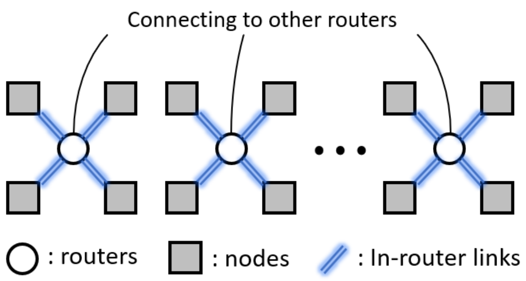


Fig. 2: NeDD quantifies the network traffic intensity of a router according to the flits per second metric collected over links.

In summary, existing allocation strategies seldom consult network monitoring data at runtime, and as a result, most strategies cannot adapt to the change of network hot spots. Part of the reason is that quantifying the entire network's performance of a large HPC system is not easy. For example, LaCurts' work quantify network performance by continuously querying the network throughput of all pairs of nodes through ping-pong tests [13], which generates an unacceptable overhead for large systems. In comparison, NeDD uses monitoring data to infer network hot spots in the system, which has a much lower overhead than active query methods.

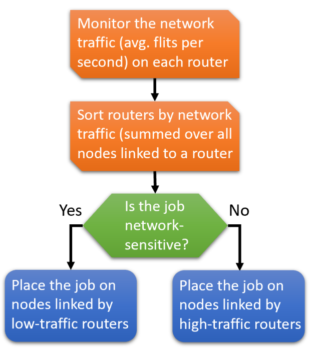
III. The NeDD Job Allocation Framework

To incorporate network monitoring data into job allocation decisions, we propose the NeDD job allocation framework.

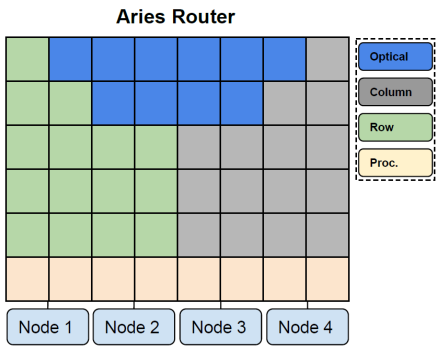
A. The Principle and Implementation of NeDD

As shown in Fig. 1, NeDD comprises three components: a network monitoring component, an application profiling component, and an allocation decision component. The *network monitoring component* characterizes the network performance based on network metrics. Its purpose is to obtain an up-to-date knowledge of which nodes, routers, or network links suffer from network congestion. The *application profiling component* determines an application's sensitivity to network congestion. The purpose of this component is to obtain the knowledge of which application's performance degrades more significantly than the others under network congestion conditions. Finally, the *allocation decision component* determines where to place a certain job based on the network and application characteristics provided by the previous two components. In principle, this allocation decision component places more network-sensitive jobs on nodes that are suffering from less network congestion. Detailed implementation choices are discussed in the following.

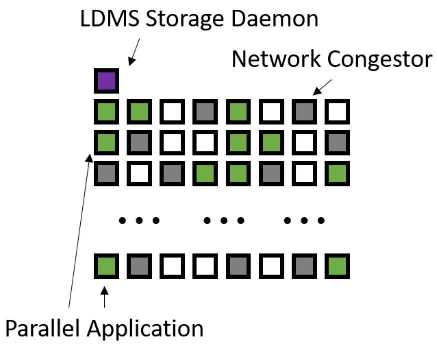
The network monitoring component collects the flits per second metric between each node and its parent router to quantify the network performance at runtime. Figure 2 shows the design of the system we experiment with where four nodes are linked to each router. In this figure, the blue links connecting each node with its parent router are the places where we collect the flits per second network statistics. Using these network metrics, we quantify the *network traffic intensity* of each router defined as the total flits per second


Monitor the network traffic (avg. flits per second) on each router
Sort routers by network traffic (summed over all nodes linked to a router
Is the job network-sensitive?
Yes
No
Place the job on nodes linked by low-traffic routers
Place the job on nodes linked by high-traffic routers

(a) Flowchart of NeDD.



(b) Aries router architecture.



(c) One-job experiments

Fig. 3: The NeDD flowchart, router architecture, and design of experiments.

summed over all nodes directly connected to that router. To mitigate the noise in these network metrics generated by bursty communication traffic, we take the average of these metrics' values over a certain time window.

The application profiling component determines whether an application is sensitive to network congestion or not, and one direct approach to quantify that sensitivity is to compare the application's performance in situations with or without network congestion. However, this direct approach requires running a set of experiments that execute the application multiple times, which poses a heavy burden on the valuable compute resources. Fortunately, our previous work has demonstrated that the MPI usage statistics of parallel applications, such as the ratio of time spent on MPI operations, are good indicators of applications' sensitivity to network congestion [4]. Therefore, in the implementation of NeDD, we characterize an application's sensitivity to network congestion based on its MPI statistics, which can be obtained by running the application only once. From the collected MPI statistics, we classify an application as either *network-sensitive* or *network-insensitive*, based on its ratio of time spent on MPI operations.

The allocation decision component controls where to place a job among the available nodes in an HPC system. In our implementation, we apply a straightforward strategy that pairs a network-sensitive job with the nodes whose parent routers have low network traffic intensity. On the other hand, we pair a network-insensitive job with the nodes whose parent routers have high network traffic intensity. We allocate nodes to network-insensitive applications in this way because this allocation decision can leave the other low-traffic routers for other network-sensitive applications. To be specific, as shown in Fig. 3 (a), the allocation decision component first obtains the collected network traffic intensity of all routers in the system, and then, it sorts all nodes from low-traffic to high-traffic according to their parent routers' network traffic intensity. Finally, For a network-sensitive job, it prioritizes selecting nodes whose parent routers have lower network traffic intensity. Conversely, for network-insensitive jobs, it prioritizes selecting nodes whose parent routers have higher network traffic intensity.

B. *Discussion on Real-world Deployment*

As our allocation strategy requires the knowledge of whether a job is network-sensitive or not, the real-world deployment of our framework will benefit from a mechanism that can identify the jobs in the queues. One way to identify jobs in HPC systems is to maintain a database that records the mapping between a job's executable binary file's name and its corresponding application name. In this way, although the names of different submission scripts may vary, we can identify them as running the same application as long as they call the same binary file. Sometimes, the same application can have multiple different binary names, and in that case, this information could be gathered manually through interaction with users. As an example, the Cori system we experiment with maintains such a database for application identification.

As NeDD relies on applications' MPI statistics to determine its sensitivity to congestion, in the real-world deployment of NeDD, we can maintain a database of applications' MPI statistics. Any unprofiled application will be launched with a default allocation strategy, and its MPI statistics collected in its first run will be used for future runs. Since the same application run with different node counts may have different network-sensitivity, ideally, we need to profile an application's MPI statistics for each node count that it is run with. However, to reduce the overhead generated by the profiling procedure, we can assume that the network-sensitivity of an application maintains the same for a range of node counts.

IV. Experimental Methodology

In this section, we describe the system setup, the applications we use, the experimental design, the baseline allocation policies, the GPCNeT tool we use to create network congestion, and the CrayPat tool we use to profile applications.

A. *System and Application Setup*

To evaluate NeDD, we experiment on Cori, a large HPC system in production. It is a 12k-node Cray XC40 system located at the Lawrence Berkeley National Laboratory, USA. The system features a dragonfly network topology [23], and an adaptive routing policy is applied to the network.

To monitor the system's hardware performance, the Cori system runs the LDMS as a background service on every node of the system [16]. Except for a couple of nodes dedicated to recording the entire system's performance counter values to files, the overhead of running LDMS is low, as the typical CPU usage of LDMS on a compute node is less than 0.1%. The LDMS has been continously running on the entire Cori system for more than two years, and on every node, it is collecting hundreds of metrics from hardware performance counters at the granularity of once per second [15]. In our experiments, we build and run our own instance of LDMS, and we configure our LDMS to collect metrics once per second.

Figure 3 (b) shows the architecture of an Aries router in the Cori system [24]. Each router connects four computing nodes and contains 48 tiles. The 8 tiles at the bottom, called Ptiles (Processor tiles), contain network traffic directed to/from the four nodes. The other 40 tiles, called Ntiles (Network tiles), contain traffic directed to/from either the Ptiles of this router or the Ntiles of some other routers.

For our evaluation, we experiment with six different real-world or benchmark applications, including HACC [25] (cosmological simulation), HPCG [26] (conjugate gradient calculation), LAMMPS [27] (molecular dynamics), MILC [28] (quantum chromodynamics), miniMD [29] (molecular dynamics), and QMCPACK [30] (electronic structure calculation).

We run experiments on nodes with the Knights Landing micro-architecture. Each node contains 68 cores. We run the application on all 68 cores per node when running miniMD, LAMMPS, QMCPACK, and HPCG. We use 64 cores per node when running MILC and HACC as our inputs for MILC and HACC do not support using all 68 cores. We configure application inputs to make their typical execution time to be within several minutes, and the same inputs are used throughout our experiments. For these applications with our inputs, their execution times spent on the initialization phase are typically within 6 seconds, so the execution times we report are mainly for their processing phases.

B. Experimental Design

We evaluate NeDD through controlled experiments on Cori. In these experiments, we take some idle nodes from the system, run a network congestor to create network congestion, and then allocate nodes to one or two applications.

Figure 3 (c) shows an example of running a network congestor and one application. To conduct that experiment, we first get N idle nodes (squares in the figure) from the system by the Slurm scheduler. Then, we run a network congestor using the GPCNeT on C randomly selected nodes (grey squares) [31]. In each repetition of our experiments, the congestor nodes are re-selected. After running the congestor for two minutes, we collect network traffic data over the last two minutes from all nodes, and we sort them according to the network traffic intensity. Then, we run an application on M nodes (green) using different allocation policies. During the experiments, we run our experiment control script and the LDMS storage daemon in one node (purple) which is prevented from running either the congestor or the application to avoid potential interference.

In our one-job experiments, we set $N = 201$, $C = 64$, and $M = 32$, so our allocator selects 32 nodes out of $136 = N - 1 - C$ available nodes to place the job. In addition, we also have two-job experiments, where we allocate nodes to two different jobs (each with $M = 32$ nodes), start the two jobs simultaneously, and measure their execution times.

We compare the following allocation strategies:

- **Low-Traffic-Router** places a job on nodes whose parent routers have low network traffic intensity. This is NeDD's strategy for network-sensitive applications.
- **High-Traffic-Router** prioritizes routers with high network traffic intensity. This is NeDD's strategy for network-insensitive applications.
- **Random** strategy places a job randomly.
- **Low-Stall-Router** strategy prioritizes routers with low Ntile network stalls (i.e., network stall count summed over all Ntiles in a router).
- **Fewer-Router** places a job into fewer routers by prioritizing routers connected to more idle nodes.

In our experiments, we also record the execution time of the applications when we do not run the network congestor (and we allocate nodes following the Fewer-Router strategy). This case is denoted as "No-Congestor".

C. Network Congestor and MPI Tracing Tool

In our experiments, we use the Global Performance and Congestion Network Tests (GPCNeT) tool to create network congestion in a controlled way [31]. GPCNeT is a tool that injects network congestion and benchmarks the communication performance of HPC systems. When launched on a set of nodes, GPCNeT runs one or multiple congestor kernels on 80% of nodes, and the other 20% of nodes run a canary test to evaluate the impact of the congestor kernels. In our experiments, we configure GPCNeT to run the RMA (Remote Memory Access) Broadcast congestor kernel.

We use the CrayPat tool to profile a job's MPI statistics [32] following the observation from our previous work that a job's sensitivity to network congestion can be estimated by its ratio of time spent on MPI operations [4]. CrayPat is an easy-to-use performance analysis tool that can be installed on Cray XC platforms to instrument an executable to trace calls to functions. CrayPat is supported on the Cori system [33].

V. Results

In this section, we first profile the MPI operation statistics of all applications that we use. Then, we show the results for the one-job allocation experiments. Finally, we show the results for the two-job allocation experiments.

A. Application Profiling

To estimate the network-sensitivity of applications, we use the CrayPat tool to profile the MPI statistics of applications when running them without the GPCNeT network congestor. In Table I, the "MPI Operation" column shows the ratio of time

TABLE I: The ratio of execution time spent on MPI operations.

Application	MPI Operation	MPI_Allreduce	MPI_Sendrecv/Send/Isend	MPI_Wait/Waitall	MPI_(other)
miniMD	68.9%	1.1%	65.8%	0	2.0%
LAMMPS	51.5%	22.4%	10.5%	12.7%	5.9%
MILC	48.2%	1.9%	7.7%	34.0%	4.6%
HACC	49.7%	1.4%	0	41.2%	7.1%
QMCPACK	19.3%	14.2%	0	<0.1%	5.1%
HPCG	11.5%	<0.1%	4.6%	6.4%	0.5%

that an application spends on MPI operations. The breakdown into specific MPI operations is shown in other columns where Some typical MPI operations are listed. The "MPI_(other)" column shows the other MPI operations not specified.

Table I shows that the six applications have different ratios of time spent on MPI operations, and we use that metric to classify an application as either network-sensitive or network-insensitive. In our system, we find QMCPACK is not affected by the congestor much, so we set the threshold to be above 19.3% and classify QMCPACK and HPCG as network-insensitive applications. We classify the other applications, including HACC, LAMMPS, MILC, miniMD, as network-sensitive applications since their ratio of time spent on MPI operations are much larger and are in the range of 48.2%∼68.9%.

B. One-job Experiments

The results for running the network congestor with a single job are shown in Fig. 4. These experiments use $N = 201$ nodes in total (including one node running the LDMS storage daemon), run network congestor on $C = 64$ nodes, and run a parallel application on $M = 32$ nodes. From Fig. 4 (a) to Fig. 4 (f), we conduct experiments with application miniMD, LAMMPS, MILC, HACC, QMCPACK, and HPCG, respectively. In each figure, we compare the execution times of the application when placed by different allocation strategies. Colored bars are cases with the network congestor, and the white bar ("No-Congestor") represents the case without the network congestor. Each bar shows the distribution of results from multiple ($\geq$10) runs using a certain allocation strategy, and the red point shows the mean execution time.

As shown in Figs. 4 (a)-(d), for network-sensitive applications, network congestion leads to a large increase in job execution time. For example, the average execution time for miniMD when running with congestor and placed by the Random strategy is 5x as large as the No-Congestor case. For these network-sensitive applications in Figs. 4 (a)-(d), the average execution time using NeDD is among the lowest values compared with other allocation strategies. Especially for miniMD, NeDD performs 34% (comparing the mean value) better than Random, 32% better than Low-Router-Stall, and 2% better than Fewer-Router. When averaged over the four applications, NeDD is performing 19% better than Random, 12% better than Low-Router-Stall, and 2% better than Fewer-Router. When averaged over the three allocation strategies, Random, Low-Stall-Router, and Fewer-Router, the improvement of NeDD is 11%. This proves that NeDD improves network-sensitive jobs' performance.

On the other hand, in Figs. 4 (e)(f), for network-insensitive applications like HPCG and QMCPACK, the impact of congestion on performance is negligible, and different allocation strategies perform similarly. This can be seen from the fact that the variance of execution times for each allocation strategy is similar to that in the No-Congestor case. Especially, the mean execution times of all strategies are within the error bars of the No-Congestor case. This proves that the network-insensitive jobs can be safely placed on high-traffic routers without causing much performance degradation while reserving the low-traffic routers for network-sensitive jobs.

C. Two-job Experiments

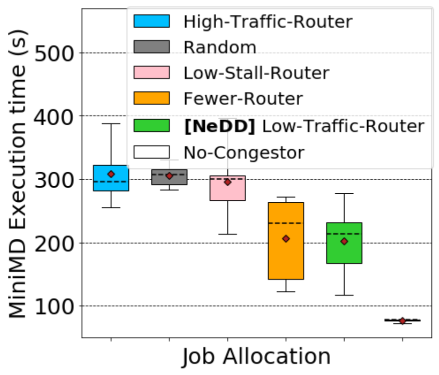
The results for two-job experiments are in Fig. 5. Similarly, we use $N = 201$ nodes in total, run network congestor on $C = 64$ nodes, and run each of the two jobs on $M = 32$ nodes. The figure caption, such as in Fig. 5 (a), "miniMD + MILC" means that we allocate nodes first to a miniMD job and then to a MILC job. After that, the two jobs start simultaneously. We use the X-axis and the Y-axis to show the execution time of the two jobs, respectively. Each point represents the mean value from multiple (≥ 5) runs.

In Fig. 5 (a), because both miniMD and MILC are network-sensitive applications, NeDD prioritizes pairing each job with nodes whose parent routers have lower network traffic, and the result of NeDD (green) is located at the left-bottom part of the figure, demonstrating that NeDD outperforms other allocation strategies for both jobs.

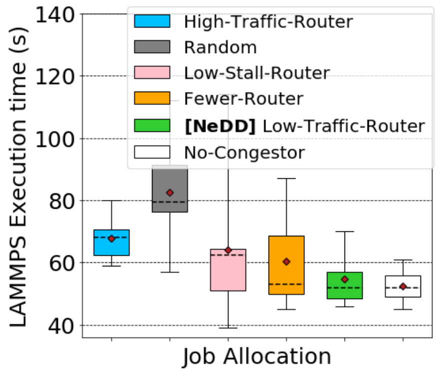
Figure 5 (b) shows the combination of a network-sensitive job with a network-insensitive job. For this combination, NeDD pairs miniMD with low-traffic routers while pairing QMCPACK with high-traffic routers. The result shows that NeDD is on the left side of the figure, demonstrating that NeDD outperforms other allocation strategies for the network-sensitive application, miniMD. Although the execution time of QMCPACK when applying NeDD is not the lowest, the performance difference between NeDD and Low-Stall-Router (the best strategy for QMCPACK in this case) is less than 1%.

Figure 5 (c) shows the same combination of miniMD and QMCPACK but with the scheduling order changed. Here, QMCPACK is placed before the allocation of miniMD. Similarly, NeDD improves the network-sensitive job by 40% (compared to Fewer-Router strategy) while the network-insensitive job's performance degrades by less than 1%.

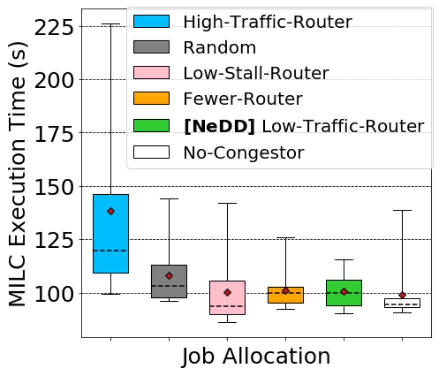
In Figs. 4,5, the Low-Stall-Router strategy performs worse than NeDD for some applications, which seems counter-intuitive at first thought. In fact, it is caused by noise in Ntile metrics. As Low-Stall-Router makes allocation decisions



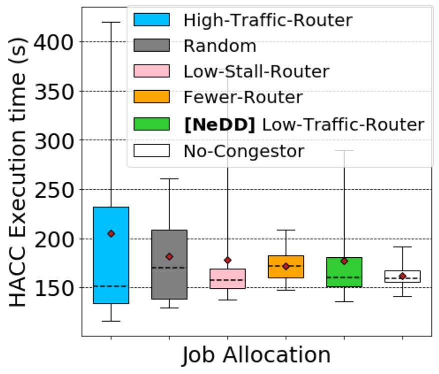
(a) miniMD



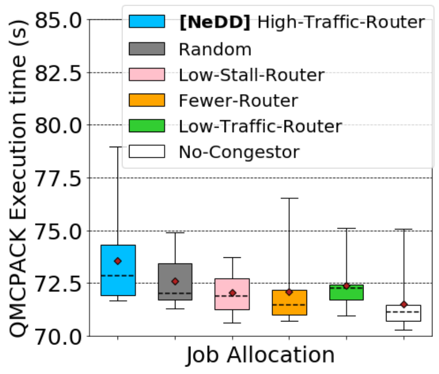
(b) LAMMPS



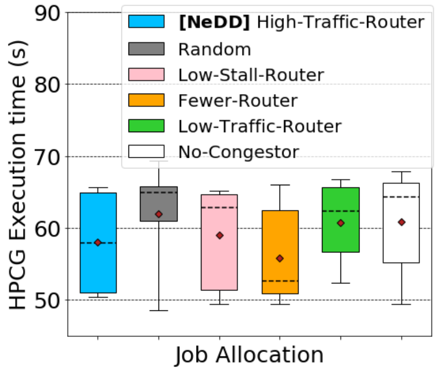
(c) MILC



(d) HACC

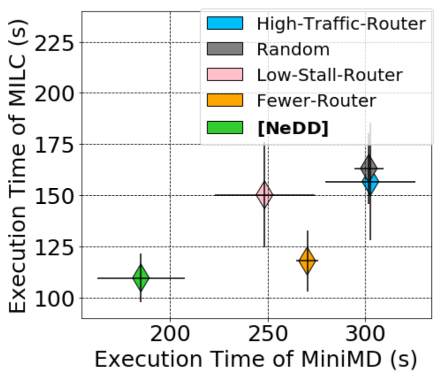


(e) QMCPACK

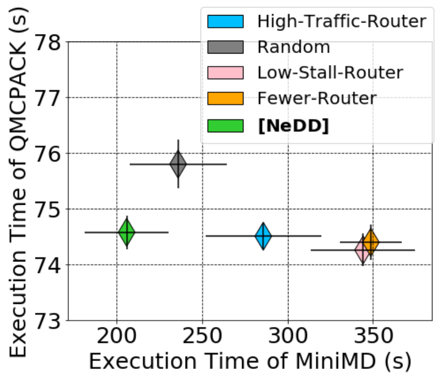


(f) HPCG

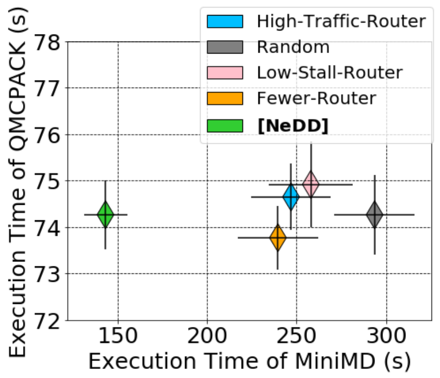
Fig. 4: One-job experiments comparing the performance of different job allocation strategies. Error bars show the minimum and maximum execution times in multiple (≥ 10) runs for each application. Colored area shows the first and third quartiles. The dashed black line shows the median, and the red point shows the mean.



(a) miniMD + MILC



(b) miniMD + QMCPACK



(c) QMCPACK + miniMD

Fig. 5: Results comparing different allocation strategies for two jobs. The diamond-shaped points show the mean execution time from multiple (≥ 5) runs. Error bars show the standard error of the mean.

based on the stall counts from Ntiles, whose traffics always include communications with some Ntiles in other routers, so the corresponding stall metrics do not well reflect the network congestion happening on the current router.

VI. Conclusion

In this work, we demonstrate using monitoring data to improve HPC performance via a Network-Data-Driven job allocation framework. Through controlled experiments, we demonstrate that NeDD reduces the execution time of parallel applications by 11% on average and up to 34%. In future works, it is possible to take other approaches to extract congestion information from HPC systems' network traffic data such as by the region-growth clustering algorithm [34]. Machine-learning algorithms can also be applied to profile applications and predict their performance under network congestion [2], [35]. In addition, it is possible to replace the current application classifier by a "smoothed" version that assigns each application a network-sensitivity value instead of using a bipartite classification strategy.

Acknowledgment

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

References

[1] A. Bhatele *et al.*, "There goes the neighborhood: Performance degradation due to nearby jobs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, 2013, pp. 41:1–41:12.

[2] A. Bhatele, J. J. Thiagarajan, T. Groves *et al.*, "The case of performance variability on dragonfly-based systems," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020.

[3] S. Chunduri, K. Harms *et al.*, "Run-to-run variability on xeon phi based cray xc systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17, 2017, pp. 52:1–52:13.

[4] Y. Zhang, T. Groves, B. Cook *et al.*, "Quantifying the impact of network congestion on application performance and network metrics," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 162–168.

[5] N. Jain, A. Bhatele, X. Ni *et al.*, "Maximizing throughput on a dragonfly network," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 336–347.

[6] B. Prisacari, G. Rodriguez, P. Heidelberger *et al.*, "Efficient task placement and routing in dragonfly networks," *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pp. 129–140, 2014.

[7] A. Jokanovic, J. C. Sancho, G. Rodriguez *et al.*, "Quiet neighborhoods: key to protect job performance predictability," *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 449–459, 2015.

[8] M. Soryani *et al.*, "Improving inter-node communications in multi-core clusters using a contention-free process mapping algorithm," *J. Supercomput.*, vol. 66, no. 1, p. 488–513, 2013.

[9] G. Michelogiannakis *et al.*, "Aphid: Hierarchical task placement to enable a tapered fat tree topology for lower power and cost in hpc networks," in *International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 228–237.

[10] Y. Baicheng, Y. Zhang *et al.*, "Lpms: A low-cost topology-aware process mapping method for large-scale parallel applications on shared hpc systems," in *48th International Conference on Parallel Processing: Workshops*, ser. ICPP 2019, 2019.

[11] S. D. Pollard, N. Jain, S. Herbein *et al.*, "Evaluation of an interference-free node allocation policy on fat-tree clusters," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 333–345.

[12] Y. Zhang, O. Tuncer, F. Kaplan, K. Olcoz, V. J. Leung, and A. K. Coskun, "Level-spread: A new job allocation policy for dragonfly networks," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 1123–1132.

[13] K. LaCurts *et al.*, "Choreo: Network-aware task placement for cloud applications," in *Conference on Internet Measurement Conference*, 2013, p. 191–204.

[14] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, p. 407–420, Aug. 2015.

[15] Cray Inc., "Aries hardware counters (4.0)," https://pubs.cray.com/bundle/Aries_Hardware_Counters_S-0045-40/page/Aries_Hardware_Counters.html, 2018.

[16] A. Agelastos *et al.*, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014, pp. 154–165.

[17] S. Jha, J. Brandt, A. Gentile, Z. Kalbarczyk, G. Bauer, J. Enos, M. Showerman, L. Kaplan, B. Bode, A. Greiner, A. Bonnie, M. Mason, R. K. Iyer, and W. Kramer, "Holistic measurement-driven system assessment," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 797–800.

[18] R. Izadpanah, N. Naksinehaboon, J. Brandt, A. Gentile, and D. Dechev, "Integrating low-latency analysis into hpc system monitoring," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: Association for Computing Machinery, 2018.

[19] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Diagnosing performance variations in hpc applications using machine learning," in *High Performance Computing*, J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, Eds. Cham: Springer International Publishing, 2017, pp. 355–373.

[20] D. Cheng, C. Jiang, and X. Zhou, "Heterogeneity-aware workload placement and migration in distributed sustainable datacenters," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 307–316.

[21] T. Cao, W. Huang, Y. He, and M. Kondo, "Cooling-aware job scheduling and node allocation for overprovisioned hpc systems," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 728–737.

[22] P. Werstein, H. Situ, and Z. Huang, "Load balancing in a cluster computer," in *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2006, pp. 569–577.

[23] J. Kim, W. J. Dally *et al.*, "Technology-driven, highly-scalable dragonfly topology," *International Symposium on Computer Architecture (ISCA)*, pp. 77–88, 2008.

[24] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray xc series network," https://www.alcf.anl.gov/files/CrayXCNetwork.pdf, 2012.

[25] K. Heitmann *et al.*, "The outer rim simulation: A path to many-core supercomputers," *The Astrophysical Journal Supplement Series*, vol. 245, no. 1, p. 16, nov 2019.

[26] J. Dongarra, M. A. Heroux, and P. Luszczek, "A new metric for ranking high-performance computing systems," *National Science Review*, vol. 3, no. 1, pp. 30–35, 01 2016.

[27] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of computational physics*, vol. 117, no. 1, pp. 1–19, 1995.

[28] G. Bauer, S. Gottlieb, and T. Hoefler, "Performance modeling and comparative analysis of the milc lattice qcd application su3_rmd," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012, pp. 652–659.

[29] M. A. Heroux *et al.*, "Improving performance via mini-applications," https://www.osti.gov/biblio/993908, 2009.

[30] J. Kim, A. D. Baczewski, T. D. Beaudet *et al.*, "QMCPACK: an open sourceab initioquantum monte carlo package for the electronic structure of atoms, molecules and solids," *Journal of Physics: Condensed Matter*, vol. 30, no. 19, p. 195901, apr 2018.

[31] S. Chunduri *et al.*, "GPCNeT: Designing a benchmark suite for inducing and measuring contention in hpc networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, 2019.

[32] Cray Inc., "Cray performance measurement and analysis tools user guide (7.0.0)," https://pubs.cray.com/content/S-2376/7.0.0/cray-performance-measurement-and-analysis-tools-user-guide/craypat, 2020.

[33] Lawrence Berkeley National Laboratory, "CrayPat Documentation," https://docs.nersc.gov/development/performance-debugging-tools/craypat/, 2020.

[34] S. Jha, A. Patke, J. Brandt *et al.*, "Measuring congestion in high-performance datacenter interconnects," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Feb. 2020, pp. 37–57.

[35] N. Jain, A. Bhatele, M. P. Robson, T. Gamblin, and L. V. Kale, "Predicting application performance using supervised learning on communication features," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 95:1–95:12.