Tangram: Colocating HPC Applications with Oversubscription

Qingqing Xiong, Emre Ates, Martin C. Herbordt, Ayse K. Coskun
Department of Electrical and Computer Engineering
Boston University, Boston, Massachusetts, 02215
Email: {qx,ates,herbordt,acoskun}@bu.edu

***Abstract*—In a cluster that is shared by many users, jobs often need to wait in the queue for a significant amount of time. Much research has been done to reduce this time with scheduling, including aggressive back-filling strategies and sharing nodes among different jobs. Although most resources are shared to some extent in HPC clusters, it is somewhat surprising that a well-known technique used on commercial clouds, i.e., oversubscribing nodes so that CPU cores are shared among jobs, is rather rare. This is partially due to concerns about interference.**

This paper presents *Tangram*, a framework for colocating applications in HPC clusters. Tangram uses prior knowledge of applications, such as whether they are I/O or CPU intensive, to predict whether potential colocations improve overall performance. To predict with sufficient accuracy, Tangram uses a combination of performance counter measurements, knowledge of past colocation performance, and machine learning. We show that Tangram can choose colocations to reduce makespan by 19% on average and by 55% in the best case, while limiting the performance degradation caused by colocation from 1598% to 26% in the worst case.

***Index Terms*—HPC, supercomputing, scheduling, interference, oversubscription, colocation**

I. Introduction

Since mid-1960s—when the first production system was designed as a cluster by Burroughs [1]—clusters have been serving as significant computing platforms. They are widely used by enterprises and academic organizations where hundreds or thousands of users might be sharing the compute resources at the same time. In a cluster that is shared by many users, typically a large number jobs are submitted by different users. These jobs usually need to wait in the queue for a substantial amount of time before starting. The nodes allocated to a job, however, are sometimes not fully used. For example, some applications require the number of ranks to be a power of two or three, but the total number of cores does not necessarily match this number. In another scenario, a job may fail within a user's reservation and the resources may stay idle and get wasted. All of these phenomena indicate that there is substantial room for improving the utilization and efficiency of clusters.

Much research has targeted achieving higher utilization of clusters through scheduling algorithms, including via back-filling (i.e. scheduling short jobs before long ones [2]); or by colocating multiple jobs on the same node where each job has its own cores [3], [4]. This kind of colocation might still result in degradation of performance for individual jobs if both jobs use the same resources intensively. Furthermore, such colocation might miss out on possible advantages in the case where a job has one thread on each core, but does not utilize the cores completely (e.g., when the job is not CPU intensive).

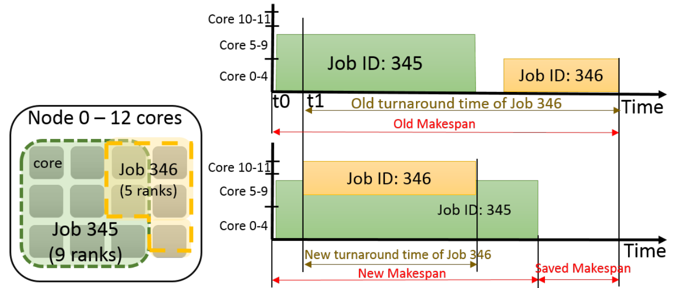

Fig. 1. The job with ID 345 uses 9 cores and starts running at $t0$, and Job 346, which requires 5 cores, arrives at $t1$. The makespan without colocating is shown as the old makespan in the figure; with colocation, the makespan reduces to the "new makespan". The turnaround time for Job 346 also changes from the old one to the new one.

We propose a scheduling mechanism that improves cluster performance by colocating jobs on the same set of nodes with overlaps on the resources such as CPU cores, memory, and I/O; this mechanism is also known as *oversubscription*. Oversubscription is a common strategy/phenomenon in cloud computing for achieving higher resource utilization. A cloud is oversubscribed when the sum of customers' requests for a resource exceeds the actual physical available capacity [5]. However, to the best of our knowledge, the concept has not been widely used in HPC domain, possibly due to interference concerns.

Colocating with overlaps might indeed encounter a significant amount of interference (*overloading*). The first question then, is whether it is possible to colocate applications on the same set of nodes with overlaps under an *endurable interference*. We consider endurable interference based on the combination of makespan and turnaround time. From the perspective of cluster providers, endurable interference indicates higher efficiency, thus, a shorter total time to execute all jobs in the queue, i.e., *makespan*. For users, the total time from job submission to job completion, which is called the *turnaround time*, is of primary interest. Note that there are other aspects (e.g., power and real-time deadlines) that the providers and users might also care about; in this paper, we focus on these two straightforward metrics (makespan and

turnaround time) as they represent commonly sought after goals. We show an example of our idea in Fig. 1, where the total makespan of running two jobs can be reduced with colocating.

The second question is how to achieve endurable interference. In modern HPC systems, information on applications can be collected through several methods such as offline profiling or online monitoring. We argue that by using prior knowledge of the applications, such as whether they are memory-intensive, I/O-intensive, or CPU-intensive, it is possible to overlap multiple applications on the same set of nodes with the minimal interference.

In this paper, we demonstrate *Tangram*, a framework to colocate HPC applications in clusters through a workflow that combines offline profiling, machine learning, and scheduling. We construct a model to represent the relationship between application characteristics and colocation performance. We evaluate the performance of Tangram by using application profiling results during scheduling on a real cluster. Specifically, our contributions in this work are as follows:

1) We design Tangram, a scheduling framework that selectively oversubscribes nodes to improve the overall makespan, while taking into account the interference characteristics of jobs. The interference characteristics are predicted using performance counters from past runs, as well as the colocation performance of other jobs.
2) We evaluate our framework on real hardware, with comparisons against different schedulers that do not consider interference or do not oversubscribe.

The remainder of this paper starts with an overview of related work in Sec. II. Then, we discuss our Tangram framework in Sec. III. We describe our testbed, the benchmarks used, and the baselines in Sec. IV. We follow up in Sec. V with results and analysis. We discuss our conclusions in Sec. VI.

II. Related Work

There has been extensive work on improving HPC system utilization and performance using different scheduling algorithms, e.g., aggressive back-filling [2] or node allocation with certain assumptions on communication patterns [6], [7].

Colocating jobs is another typical strategy for improving HPC system utilization; however, with sharing resources such as network, L3-cache and main memory rather than oversubscribing cores. For example, Simakov et al. [4] study the interference effects of implementing colocation (referred as *node-sharing* in the paper) in a production cluster without oversubscription. There is much research on studying network interference between different nodes [6], [8], [9], and also some research on studying the cache interference within the same node [10].

To the best of our knowledge, there is not much work on colocating jobs with oversubscribing cores in HPC systems. The most relevant is perhaps that of Wende et al. [3] who investigate the impact of unfavorable process placement and oversubscription of compute resources on the performance and scalability of applications on two HPC systems (including the Cray XC40). Their goal is to investigate interference rather than providing an approach to improve cluster utilization. There are some approaches in the multicore space about oversubscribing cores within a single CPU [11], [12]. However, these approaches focus on sharing a single node so do not consider network interference. Also, they provide do not provide any method for limiting negative interference. Furthermore, they assume the thread count of the applications is modifiable by system operators and software, which is often not the case for modern HPC clusters. Utrera et al. [13] also study a similar system where the thread count of jobs is malleable.

Oversubscription exists widely in cloud systems and data centers, where characterizing interference has also been studied. Baset et al. [14] provide an overview of oversubscription, along with methods for mitigating overload. There are several papers on characterizing interference between applications in the cloud. CPI^2 [15] looks at cycles per instruction as an indicator of performance, and continuously measures CPI and caps CPU usage if there is negative interference to latency sensitive workloads. Paragon [16] measures interference using micro-benchmarks, and runs the applications with these micro-benchmarks before scheduling to analyze the interference. DeepDive [17] identifies contention by analyzing performance counter data and finds the source of interference by looking at the performance counters using machine learning, and makes placement and VM migration decisions accordingly.

In contrast to the related work, we schedule HPC applications while enabling oversubscription of CPU resources. We do not make any assumptions on the malleability of the MPI framework, and use the traditional MPI job submission behavior where users choose the thread and node count. Furthermore, we provide a method to limit the negative interference caused by oversubscription.

III. Tangram

In the Tangram game, blocks are placed based on their characteristics (e.g., whether there is a right or acute angle). Similarly, our Tangram is a framework for colocating HPC applications on clusters based on their resource usage characteristics. Tangram looks at *makespan*, i.e., the total time taken to finish all of the jobs in the queue, in order to decide whether to colocate jobs or not. If colocation would improve makespan for a job pair, we call the interference between the jobs *endurable*. We predict whether the interference would be endurable or not by collecting performance metrics.

Overall, Tangram consists of three stages: (i) offline profiling of isolated applications and collection of metrics; (ii) profiling colocation performance; (iii) creating a statistical model for predicting colocation performance; (iv) using colocation performance predictions during scheduling.

A. Selection of Metrics

Representing application characteristics in a concise and accurate way [18], [19] is essential for predicting colocation performance accurately. DeepDive [17], which works in cloud

environments, chooses cores, memory, disk, and the network interface to represent the physical machine's major resources. In our HPC cluster environment, the disk is accessed through the network file system and there is no separate local disk; so we choose CPU, memory, and I/O as the three major components to profile. From one job's perspective, when sharing resources (e.g., main memory) with other jobs, what matters at each point-in-time is how intensively the resources are used by others rather than the total amount of resources used by others. Therefore, we choose the following three intensity metrics, where each metric targets one component:

- *CPU intensity*: Instructions Per Cycle (IPC) per node
- *Memory intensity*: L2 or last level cache misses per kilo-instructions per node (depending on the system[1])
- *I/O intensity*: Ratio of wait time to wallclock time as follows:

$$I_{I/O} = \frac{t_{real} - \frac{t_{sys}+t_{user}}{n_{rankspernode}}}{t_{real}}$$

The I/O intensity is derived using the system, the user time, and the real time. that is, when the application is not using the CPU or the memory, we consider it is handling I/O operations. CPU and memory intensity metrics are derived from combinations of hardware counters: CPU intensity is calculated using the total retired instruction count and the real cycle number; the memory intensity is calculated using the L2 or last-level cache miss number and the retired instruction count. The definitions of metrics might vary depending on a given platform, nevertheless, Tangram can be easily used with another set of metrics.

B. Profiling Colocation

The second stage of the Tangram framework is to profile each application's performance when it is colocated with another application.

Colocating applications on the same node implies sharing resources between applications. When colocating without overlaps, where each application has its own cores, interference may exist in the last-level cache (LLC) (if the architecture is shared-LLC), the main memory, the network interface card/controller (NIC), etc. We colocate applications with overlaps on cores, so potential interference occurs in CPU resources, all levels of the cache, and in the context switch overhead: the operating system might need to switch between different applications' processes very often. Note that it is possible to restrict the context switch frequency of the OS when doing colocation to achieve better performance, however, this is not covered by this paper.

We measure the real execution time of each application when colocating. The aggregated real time of applications is the baseline makespan, i.e., the case where the jobs are running one after another. In a real cluster, the total makespan might be larger because of scheduler delays, which we neglect in this paper.

[1]We collect LLC or L2 cache misses depending on the counters available in PAPI

Although, we perform offline profiling to obtain the performance metrics, other methods can be used for production systems. It is possible to use online methods such as using historical data, since the same application is usually submitted multiple times; another method is to run jobs for a short profiling run using back-filling, as suggested by Thebe et al. [20].

C. Predicting Interference

After collecting the data from the first two stages, we use machine learning to predict whether there is endurable interference when two given applications are colocated. We train a support vector machine (SVM) classifier using the total CPU, memory and I/O interference metrics of the application pairs, and we use a Boolean value indicating whether there is makespan improvement or not as the label.

The SVM is a supervised learning model, which aims at finding a hyperplane that divides the different classes of observations. In our case, the observations belong to two classes: colocations that improve makespan or not. Each observation is represented by the total CPU, I/O, and memory interference. We use the SVM with the radial basis function kernel, which maps our three metrics to a higher dimensional feature space, enabling the SVM to find possible non-linear relationships in the data.

D. Operation of Tangram

At runtime, before colocating two jobs, Tangram calculates the total interference characteristics, and uses the SVM to predict whether colocation will have positive effects (reduce makespan). During scheduling, Tangram may survey available nodes with empty resources (mainly cores) in the cluster, and use the intensity metrics to decide whether to colocate the next job in the queue, or wait for nodes to free up.

Even though the makespan is reduced, the turnaround time and performance of individual jobs may be adversely affected by colocation. We assume that reducing makespan and the queue wait time is more important than the performance of individual jobs. In practice, the scheduling policy may treat the performance-critical jobs differently by not colocating them or by partitioning the machine.

IV. Experimental Setup

To test the benefits and impact of Tangram, we perform a series of experiments with our method in a real cluster with a variety of applications.

A. Computing Cluster

We run our experiments on Boston University's Shared Computing Cluster (SCC) [21] located in the Massachusetts Green High Performance Computing Center (MGHPCC). The SCC system currently includes over 14500 CPU cores, and over 4.2 petabytes of storage for research data. We test colocation performance on two problem sizes: a two-node test where each application is mapped to two nodes, and an eight-node test where each application is mapped to eight nodes.

TABLE I
DETAILED CONFIGURATIONS OF TWO SYSTEMS

Configurations	Two-node system	Eight-node system
CPU/node	Two six-core 3.07GHz Intel Xeon X5675	Two eight-core 2.6GHz Intel Xeon E5-2670
Main memory/node	48GB	128GB
Network	QDR InfiniBand	FDR InfiniBand
Applications	6 applications	6 applications
Rank Count	13, 14, 15	72, 80, 88, 96

The two tests are performed on two subsets of nodes with the configuration shown in Table I.

The MPI applications are compiled with OpenMPI 1.6.4 with GCC 4.4.7 and InfiniBand enabled. The operating system is CentOS 6, with the Linux kernel v. 2.6.32. We install our applications using the Spack package manager [22].

B. Performance Counter Collection

To collect performance counters, we use PapiEX and PAPI [23]. PAPI is a widely used tool for collecting performance counters from machines with different CPUs. PapiEx is a PAPI-based program for measuring hardware performance events of an application without re-compiling. For MPI programs, PapiEx can gather statistics across ranks. We collect system time, real time and user time using GNU bash builtin function `time`, version 4.1.2.

C. Applications

The majority of HPC applications are using the Message Passing Interface (MPI). The MPI [24], [25] has been considered as the dominant scalable parallel programming model, and it is widely used in HPC clusters. There are also other parallel programming models such as OpenMP and Pthread which focus on intra-node operations, whereas MPI supports both inter- and intra-node programming. In this paper, we choose MPI applications to experiment and verify with.

We select a wide range of applications from three different proxy application suites. From Mantevo benchmark suite [26], we use miniAMR (adaptive mesh refinement), miniGhost (stencil computation), and miniMD (molecular dynamics simulation). From Lawrence Livermore National Laboratory proxy applications [27], we use MACSio (I/O benchmarking with CPU computation as well) and kripke (S_N transport). From National Energy Research Scientific Computing Center proxy applications [28] we use IOR (I/O benchmark). We can estimate the characteristics of these six applications based on their purposes: MACSio and IOR are I/O-bound, and the other four are used for scientific computation, which are possibly CPU-bound or memory-bound, e.g., miniAMR performs multiple times of read and write for each cell in each timestep, which can be considered as memory-bound.

Based on our assumptions, applications with different resource usage characteristics should colocate with low interference. We will analyze the colocation performance results with the knowledge of these applications.

D. Colocating Applications

We run our tests from a batch script submitted to the scheduler of SCC (Sun Grid Engine). For oversubscription, we run two applications in one script, where one application is running in the background using `&`, and we use `wait` command to wait for the termination of both applications.

In the two-node test, we run each application with 13, 14 and 15 ranks; in the eight-node test, we run each application with 72, 80, 88 and 96 ranks. All applications are run with the ranks equally split to all the nodes (using `-npernode`), so each node is guaranteed to be partially empty and oversubscription in each node is guaranteed. We run all combinations of applications, for a total of 84 colocated runs (including colocating each application with itself) for the eight-node and 63 colocations for the two-node case.

We collect the running time for each of these combinations, and compare with the two applications' isolated performance. We collect PAPI data separately for each application with each rank count, and sum up the per rank metrics into each application's metric results.

E. Baselines

We have two baselines for our method: 1) colocate every combination regardless of interference (referred to as *All colocated*); 2) the default scheduling method in current clusters, where nothing is colocated if there are not enough dedicated cores for each of the applications (referred to as *No colocation*).

V. EXPERIMENTAL RESULTS

In this section, we show the performance of our method by evaluating its decisions under different contexts. In both the two-node and the eight-node cases, we use three tests to demonstrate the performance of Tangram:

- **CV** We test our model first by performing 10-fold cross-validation (CV). We split all combinations of runs into 10 sets randomly, and train our model with 9 sets and test with the 10th one.
- **(I)** We train the SVM with all runs except for the runs with one specific rank count (iterated over all rank counts), and use the SVM to predict the colocation performance of the remaining runs.
- **(II)** We train the SVM with five of the applications, and use the SVM to predict the colocation performance of the sixth application (iterated over all applications). To prevent negative interference, we do not colocate the unknown application with itself.

The CV test represents the operation of Tangram when the training set accurately represents the applications and inputs seen at runtime. The last two tests are performed to evaluate the robustness of Tangram to new conditions, such as rank counts and applications that are new to the system.

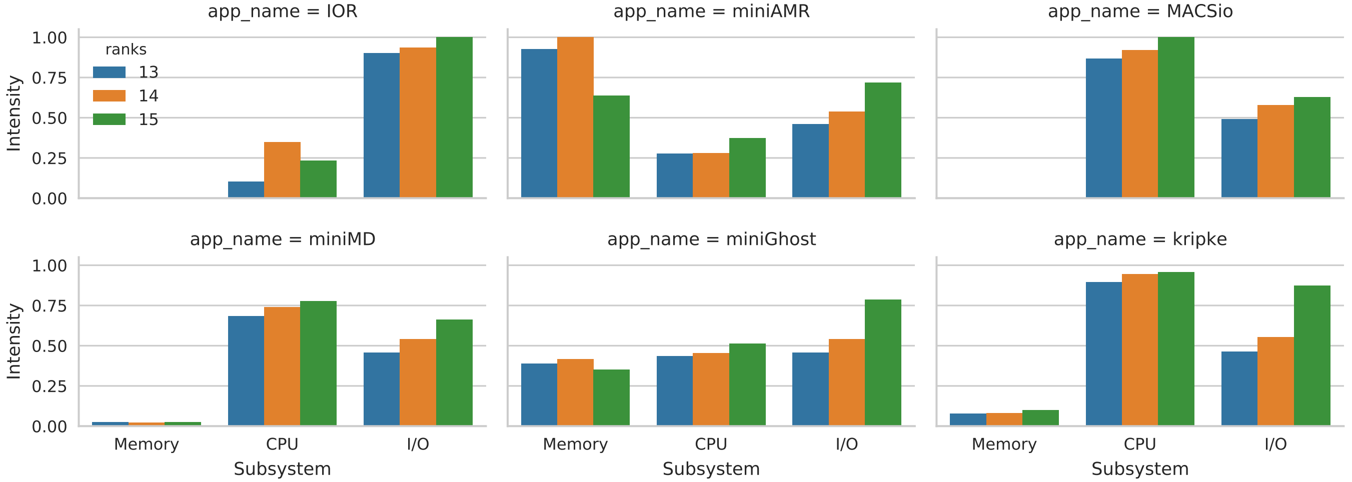

Fig. 2. The calculated intensities of each applications, with different total rank count. The results are normalized to the highest intensity of each metric.

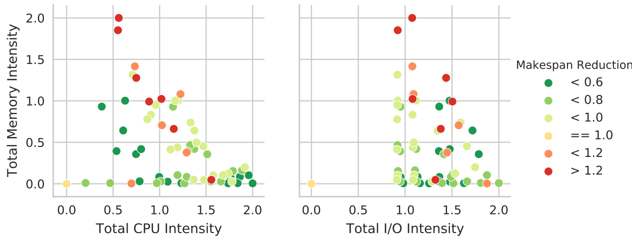

Fig. 3. The total intensity for each subsystem, and the resulting makespan reduction, for all combinations of applications in the two-node case. A smaller makespan reduction indicates a smaller interference between the corresponding applications.

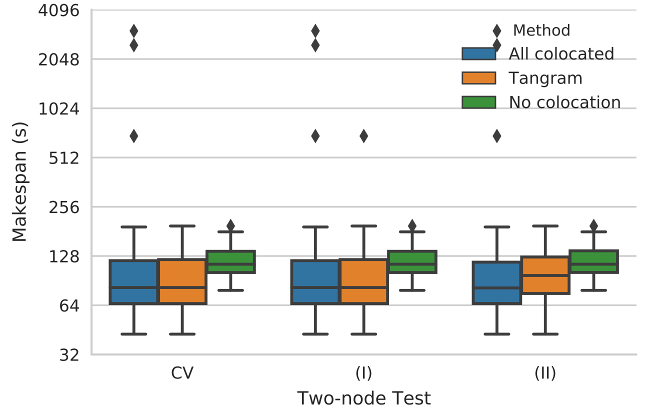

Fig. 4. The absolute makespan for each method. Tangram improves overall makespan while also limiting the worst case performance.

A. Two-node Results

1) Interference Characteristics: We report the three intensity metrics for each of our applications in Fig. 2. From the results, we make the following observations:

- IOR has the highest I/O intensity, Kripke and MACSio have the highest CPU intensity, and miniAMR has the highest memory intensity among the six applications.
- For each application, the CPU and I/O intensity metrics scale up with the rank number, since the results are per-node – more ranks result in a higher intensity. However the memory intensity sometimes does not follow the trend, especially for miniAMR and miniGhost, which are weak scaling applications, so a larger rank count simulates a different environment, which results in different metrics.

Overall, each application has different characteristics, which enable us to colocate them with endurable interference.

In Fig. 3, we show the relation among the three metrics and the makespan reduction with colocation. The makespan reduction is the colocated makespan divided by the makespan without colocation. From the result, we can see that the worst makespan degradations are clustered together in high-interference regions. The SVM uses this knowledge to make predictions on colocation performance.

2) Test Results: Figure 4 shows the distribution of makespan for the two-node tests. Colocating applications can improve performance. However, makespan degradation can also be as high as $17\times$ of the default case. This happens when colocating two identical miniAMR applications, where each of them has both high memory and I/O intensities. In the cross-validation results, our method outperforms the default scheduling method (No Colocation) in makespan improvement by 25%, while having no loss in turnaround time (Table. II). Tangram is also robust against unknown applications and rank counts, shown in tests (I) and (II).

B. Eight-node Results

When scaling up all six applications from running with less than 20 ranks to more than 90 ranks, the normalized intensity metric results and trends from Fig. 2 mostly hold.

Figures 6 and 7 show the distribution of makespan and turnaround time, respectively, for the eight-node test. We summary the test results of both the two-node and the eight-node cases in Table II. Table II summarizes the geometric mean, the maximum, and the minimum values of makespan and turnaround time of the colocating methods. The values

TABLE II
RESULTS FOR COLOCATING TESTS: CROSS VALIDATION (CV), TESTING WITH UNKNOWN RANK COUNT (I), AND TESTING WITH AN UNKNOWN APPLICATION (II). AC REPRESENTS THE ALL COLOCATED CASES. THE RESULTS ARE NORMALIZED WITH RESPECT TO NO COLOCATION.

Node-Test	Method	Makespan Gmean	Makespan Max	Makespan Min	Turnaround Time Gmean	Turnaround Time Max	Turnaround Time Min
2-CV	Tangram	0.75	1.22	0.41	1.00	1.74	0.62
2-CV	AC	0.80	16.98	0.41	1.13	24.02	0.62
2-(I)	Tangram	0.81	1.26	0.47	1.08	2.08	0.65
2-(I)	AC	0.83	16.98	0.47	1.19	24.02	0.65
2-(II)	Tangram	0.82	1.30	0.41	1.04	2.19	0.55
2-(II)	AC	0.77	16.98	0.41	1.10	24.02	0.55
8-CV	Tangram	0.90	2.30	0.50	1.06	3.25	0.73
8-CV	AC	1.20	6.57	0.50	1.69	9.29	0.73
8-(I)	Tangram	0.93	2.82	0.50	1.08	3.93	0.68
8-(I)	AC	1.20	6.57	0.50	1.67	9.29	0.68
8-(II)	Tangram	0.96	2.12	0.50	1.05	3.07	0.68
8-(II)	AC	1.14	6.57	0.50	1.60	9.29	0.68

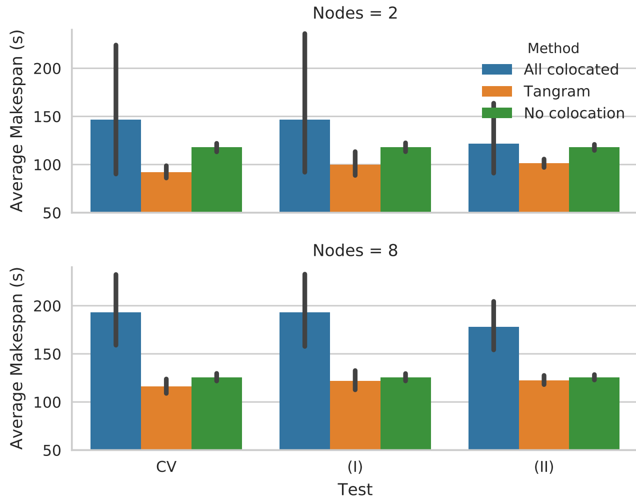

Fig. 5. The average makespan for each test.

are normalized to the performance of no colocation for each combination. Figure 5 also presents an overview of the results. Tangram clearly improves makespan, while limiting the negative interference.

From the results, we make the following observations:

- Tangram outperforms the default scheduling method (No Colocation) in makespan improvement by 10% when doing cross validation, while only having a loss in turnaround time of 6%.
- For the eight-node test, All colocated (AC) performs 14-20% worse than the default case. This is because there are more overlapped cores on each node on average compared to the two-node case.
- The turnaround time is calculated by assuming one of the applications start immediately, while the other one waits for the first one to finish. Therefore, turnaround time is usually improved for the second application while it's degraded for the first application. This can be seen in the bi-modal distributions for no colocation in Fig. 7. Note that when running job queues where applications comes one after another, all the applications except the very first one may achieve a shorter turnaround time.
- Overall, the geometric mean of the turnaround time increases between 6% and 8% compared to the default, and the maximum degradation is 393%. Note that this happens when sharing 6 cores per node, and in normal operation this much oversubscription might be blocked altogether if the variation is deemed too high.

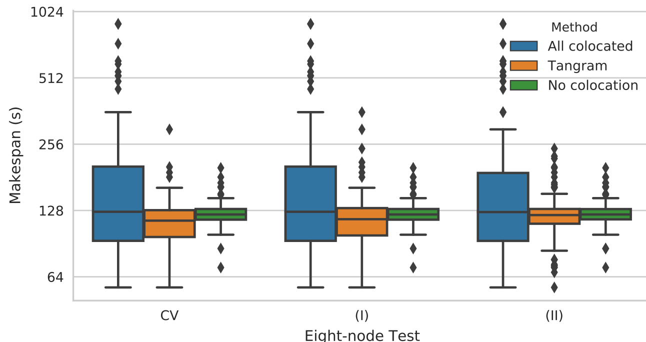

Fig. 6. Absolute makespan for each method in the eight-node test. Tangram improves overall makespan while also limiting the worst case performance.

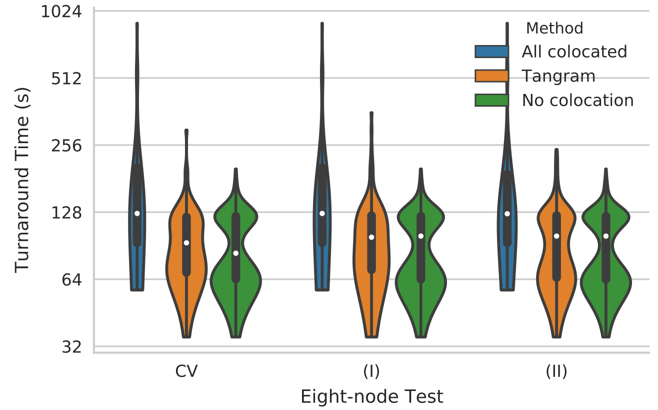

Fig. 7. Absolute turnaround time for each method. Tangram improves turnaround time for some applications. Bi-modal appearance of the no-colocation method is because of the wait time applied to half of the applications.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have introduced Tangram, a framework for colocating HPC applications with oversubscription. Tangram uses machine learning and performance counters to predict interference, and colocates applications in a way that provides improvements in makespan without severe degradation of performance for any job.

For future work, we can evaluate Tangram using job queues to get a better estimate of makespan improvement. A scheduling policy that involves Tangram would require more consideration on the collection of profiling data. Adding FPGAs in clusters is becoming a trend [29], [30]. Traditional CPU-only systems require the colocated applications to take turns to share one resource, whereas FPGAs can provide "unlimited" copies of a single resource that multiple applications can do the same task at the same time. Colocation would be more beneficial in such heterogeneous systems.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation through Awards #CNS-1405695 and #CCF-1618303/ 7960; by grants from Microsoft and Red Hat; and by Altera through donated FPGAs, tools, and IP.

REFERENCES

[1] G. F. Pfister, *In Search of Clusters (2nd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[2] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Characterization of backfilling strategies for parallel job scheduling," in *Proceedings of the International Conference on Parallel Processing Workshop*, 2002, pp. 514–519.

[3] F. Wende, T. Steinke, and A. Reinefeld, "The Impact of Process Placement and Oversubscription on Application Performance: A Case Study for Exascale Computing," in *Proceedings of the 3rd International Conference on Exascale Applications and Software (EASC'15)*. Edinburgh, Scotland, UK: University of Edinburgh, 2015, pp. 13–18.

[4] N. A. Simakov, R. L. DeLeon, J. P. White, T. R. Furlani, M. Innus, S. M. Gallo, M. D. Jones, A. Patra, B. D. Plessinger, J. Sperhac, T. Yearke, R. Rathsam, and J. T. Palmer, "A Quantitative Analysis of Node Sharing on HPC Clusters Using XDMoD Application Kernels," in *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale (XSEDE'16)*. New York, NY, USA: ACM, 2016, pp. 32:1–32:8.

[5] R. Householder, S. Arnold, and R. Green, "On Cloud-based Oversubscription," *International Journal of Engineering Trends and Technology*, vol. 8, no. 8, pp. 425–431, 2014.

[6] A. Jokanovic, J. C. Sancho, G. Rodriguez, A. Lucero, C. Minkenberg, and J. Labarta, "Quiet neighborhoods: Key to protect job performance predictability," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 449–459.

[7] W. Liu, V. Lo, K. Windisch, and B. Nitzberg, "Non-contiguous processor allocation algorithms for distributed memory multicomputers," in *Proceedings of Supercomputing '94*, Nov 1994, pp. 227–236.

[8] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: Performance degradation due to nearby jobs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13)*. New York, NY, USA: ACM, 2013, pp. 41:1–41:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503247

[9] R. E. Grant, K. T. Pedretti, and A. Gentile, "Overtime: A Tool for Analyzing Performance Variation Due to Network Interference," in *Proceedings of the 3rd Workshop on Exascale MPI (ExaMPI'15)*. New York, NY, USA: ACM, 2015, pp. 4:1–4:10. [Online]. Available: http://doi.acm.org/10.1145/2831129.2831133

[10] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei, "A Practical Method for Estimating Performance Degradation on Multicore Processors, and Its Application to HPC Workloads," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 83:1–83:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389109

[11] H. Kamal and A. Wagner, "Added Concurrency to Improve MPI Performance on Multicore," in *2012 41st International Conference on Parallel Processing*, Sept 2012, pp. 229–238.

[12] C. Iancu, S. Hofmeyr, F. Blagojevi, and Y. Zheng, "Oversubscription on multicore processors," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010, pp. 1–11.

[13] G. Utrera, J. Corbalan, and J. Labarta, "Scheduling parallel jobs on multicore clusters using CPU oversubscription," *The Journal of Supercomputing*, vol. 68, no. 3, pp. 1113–1140, Jun 2014. [Online]. Available: https://doi.org/10.1007/s11227-014-1142-9

[14] S. A. Baset, L. Wang, and C. Tang, "Towards an understanding of oversubscription in cloud," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE'12)*. Berkeley, CA, USA: USENIX Association, 2012, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228283.2228293

[15] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI2: CPU Performance Isolation for Shared Compute Clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. New York, NY, USA: ACM, 2013, pp. 379–391. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465388

[16] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. New York, NY, USA: ACM, 2013, pp. 77–88. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451125

[17] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013, pp. 219–230. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/novakovic

[18] E. Ates, O. Tuncer, A. Turk, V. J. Leung, J. Brandt, M. Egele, and A. K. Coskun, "Taxonomist: Application Detection through Rich Monitoring Data," in *Euro-Par 2017: Parallel Processing*, 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] O. Thebe, D. P. Bunde, and V. J. Leung, "Job scheduling strategies for parallel processing," E. Frachtenberg and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Scheduling Restartable Jobs with Short Test Runs, pp. 116–137. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04633-9_7

[21] Boston University, "Shared Computing Cluster (SCC)," 2018. [Online]. Available: https://www.bu.edu/tech/support/research/computing-resources/scc

[22] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The Spack Package Manager: Bringing Order to HPC Software Chaos," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. New York, NY, USA: ACM, 2015, pp. 40:1–40:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807623

[23] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting Performance Data with PAPI-C," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[24] MPI Forum, "MPI: A Message-Passing Interface Standard," Knoxville, TN, USA, Tech. Rep., 1994.

[25] ——, "MPI: A Message-Passing Interface Standard: Version 3.1," Tech. Rep., June 4, 2015.

[26] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[27] Lawrence Livermore National Laboratory, "Co-design at Lawrence Livermore National Lab," 2018. [Online]. Available: https://codesign.llnl.gov/proxy-apps.php

[28] National Energy Research Scientific Computing Center, "NERSC-8 / Trinity Benchmarks," 2018. [Online]. Available: http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/

[29] Q. Xiong, A. Skjellum, and M. Herbordt, "Accelerating MPI Message Matching Through FPGA Offload," in *Proc. IEEE Conf. on Field Programmable Logic and Applications*, 2018.

[30] Q. Xiong, P. Bangalore, A. Skjellum, and M. Herbordt, "MPI Derived Datatypes: Performance and Portability Issues," in *Proceedings of the EuroMPI Conference*, 2018.