Spread-n-Share: Improving Application Performance and Cluster Throughput with Resource-aware Job Placement

Xiongchao Tang*
Tsinghua University, and
Sangfor Technologies Inc.
txc13@mails.tsinghua.edu.cn

Haojie Wang*
Tsinghua University
wang-hj18@mails.tsinghua.edu.cn

Xiaosong Ma
Qatar Computing Research Institute
xma@hbku.edu.qa

Nosayba El-Sayed†
Emory University
nosayba.ae@emory.edu

Jidong Zhai
Tsinghua University
zhaijidong@tsinghua.edu.cn

Wenguang Chen
Tsinghua University
cwg@tsinghua.edu.cn

Ashraf Aboulnaga
Qatar Computing Research Institute
aaboulnaga@qf.org.qa

ABSTRACT

Traditional batch job schedulers adopt the Compact-n-Exclusive (CE) strategy, packing processes of a parallel job into as few compute nodes as possible. While CE minimizes inter-node network communication, it often brings self-contention among tasks of a resource-intensive application. Recent studies have used virtual containers to balance CPU utilization and memory capacity across physical nodes, but the imbalance in cache and memory bandwidth usage is still under-investigated.

In this work, we propose Spread-n-Share (SNS): a new batch scheduling strategy that automatically scales resource-bound applications out onto more nodes to alleviate their performance bottleneck, and co-locate jobs in a resource compatible manner. We implement Uberun, a prototype scheduler to validate SNS, considering shared-cache capacity and memory bandwidth as two types of performance-critical shared resources. Experimental results using 12 diverse cluster workloads show that SNS improves the overall system throughput by 19.8% on average over CE, while achieving an average individual job speedup of 1.8%.

CCS CONCEPTS

• **General and reference → Performance**; • **Software and its engineering → Scheduling**; **Multiprocessing / multiprogramming / multitasking**.

*Majority of work performed at Qatar Computing Research Institute during research internship
†Part of work performed at Qatar Computing Research Institute

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356152

ACM Reference Format:
Xiongchao Tang, Haojie Wang, Xiaosong Ma, Nosayba El-Sayed, Jidong Zhai, Wenguang Chen, and Ashraf Aboulnaga. 2019. Spread-n-Share: Improving Application Performance and Cluster Throughput with Resource-aware Job Placement. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3295500.3356152

1 INTRODUCTION

Parallel programs are widely used today, on platforms including High-performance Computing (HPC) clusters, supercomputers, datacenters, and cloud systems. Traditionally, scientific computing programs use HPC clusters and supercomputers, and "commercial" parallel/distributed applications such as big data and machine learning favor datacenters and clouds. In recent years however, there is a trend of convergence between these two application areas: emerging big data and machine learning applications are increasingly ported to and optimized for large-scale HPC platforms [43, 44, 61], while conventional HPC applications are exploring public cloud offerings [62, 63].

On both types of platforms, batch jobs are managed and dispatched with schedulers. These include SLURM [60], LSF [37], and PBS [13] on HPC platforms, as well as YARN [55] and Mesos [35] on datacenters and clouds. Parallel jobs, scientific and commercial alike, are scheduled in a *Compact-n-Exclusive* manner (*CE* for short): processes of a parallel job are packed into as few compute nodes as possible, usually fully occupying all available CPU cores or memory capacity within each node; allocated compute nodes are dedicated to a single job at a given time. This practice is mainly based on the notion that a smaller "node footprint" allows a parallel job with a fixed number of processes to better utilize the much faster intra-node communication, minimizing traffic via the slower interconnection network. Trimming inter-node communication also reduces performance inconsistency caused by interference among concurrent jobs.

Besides efficiency considerations, such CE scheduling fits well with common charging models on parallel computing platforms. With either the "resource units" accounting on supercomputers, or

the monetary on-demand rental charges on cloud-based clusters, a job's cost is accounted as *node-hours* consumed, by multiplying the total number of nodes allocated and a job's total execution time.

As a result, CE scheduling is a dominant policy adopted on HPC platforms. In particular, all the top 10 supercomputers (according to the November 2018 Top500 ranking [54]) use CE for parallel jobs.[1] While CE does reduce per-job node usage and better utilizes intra-node communication, when running resource-intensive applications it often brings severe *self-contention*, by placing homogeneous tasks within a node. With growing core counts and fast improving interconnection speed in modern systems, the assumption that CE delivers better single-application efficiency is questionable.

There have been studies on breaking the CE pattern, especially on commercial platforms. E.g., schedulers like YARN, Mesos, and Kubernetes adopt virtual containers, with processes of a single job not necessarily compacted on a minimum node footprint [19, 33, 35]. Their workload balancing policies, however, focus on metrics such as CPU utilization and memory capacity.

In this work, we propose a new batch job scheduling strategy, *Spread-n-Share (SNS)*, that targets better overall resource utilization on parallel platforms *and* better application performance. Instead of contracting parallel batch jobs to their minimum node footprints, SNS *scales them out* to use more nodes when a resource bottleneck due to self-contention is detected, and co-locates them in a resource compatible manner. We demonstrate that modern resource-intensive parallel applications often have performance constrained by per-node memory bandwidth or cache capacity, and explore the integration of resource-aware workload scaling/placement and intra-node resource allocation, using hardware features such as the Intel Cache-Allocation Technology (CAT) [39] for last level cache (LLC) partitioning.

Below we illustrate SNS with a sample multi-workload experiment on a small testbed, with compute nodes (dual Xeon E5-2680 v4 CPU) connected via 50 Gbps Infiniband. We run a simple workload mix with three resource-intensive parallel programs: `MG`, `HC`, and `TS`, each using 16 cores. `MG` (MultiGrid) is an HPC program from the NPB benchmark [11]. `HC` (H264) is a video coding program from the SPEC CPU 2006 benchmark [52]; though it is a sequential program, we simultaneously submit 16 instances. `TS` (TeraSort) is a Spark [8] sorting program from Hibench [36].

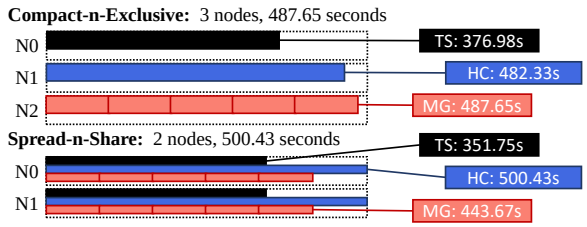

Figure 1: Spread-n-Share (SNS) reduces the overall node-hours by 34.58%, and improves the performance of `MG` and `TS` by 9.02% and 7.17%, respectively.

[1]One of them, namely ABCI, supports node sharing, but only for jobs using one node, or sub-tasks of a single job.

The top of Figure 1 shows the schedule layout under the common practice of CE. As each node has 28 cores, only one program can run on each node at any time, thus CE has the 3 programs run on separate nodes. We repeat `MG` five times so that the three programs finish in close time. The bottom of the figure shows the schedule using SNS, which spreads the programs onto two nodes. Though sharing resources with other programs, the performance of `MG` and `TS` improves by 9.02% and 7.17% respectively, thanks to higher memory bandwidth and more cache available, while `HC` sees a less significant performance loss (3.75%). In addition to individual programs' performance improvement, node sharing reduces idle cores and thus improves the system throughput. Even with fewer nodes, the start-to-end time of all jobs with SNS is only 2.62% longer (487.65s vs. 500.43s). The overall resource usage (in node-seconds) is reduced by 34.58%.

The novel contributions of our proposed SNS strategy lie in that it (1) joins job auto-scaling and co-location to simultaneously improve system throughput and application performance, (2) focuses on two less-studied performance factors: memory bandwidth and shared-cache capacity, (3) employs a flexible and extensible algorithm to accommodate multiple shared resources for QoS-aware scheduling, and (4) leverages recent/upcoming hardware features (such as CAT) for inter-workload resource/interference management. To the best of our knowledge, this work is the first to spread and co-locate jobs according to their LLC or memory bandwidth requirements.

We implement SNS in Uberun, a prototype scheduler, with a profiler purely based on PMUs (Performance Monitoring Units) and requiring no application modification (source or binary). We evaluate Uberun using a variety of parallel programs for machine learning, data analytics, and scientific computing. Most tests are on a modern cluster with CAT-enabled nodes, plus several studying SNS's behavior on much larger clusters using trace-driven simulation. Results show that SNS is able to both improve the overall system throughput and speed up most resource-constrained jobs, significantly outperforming the CE strategy, as well as a node-sharing intermediate solution.

2 APPLICATION SCALING BEHAVIOR

We begin by studying the behavior of individual parallel applications when scaled out to use more nodes. To this end, we characterize four representative programs with varying resource-access patterns: `MG`, `CG`, `EP`, and `BFS`. The first three programs are from the NPB parallel benchmark suite [11] (all running with CLASS-D input size): `MG` is a 3-D Poisson equation solver using a multi-grid method (used in Figure 1), with high memory bandwidth demand; `CG` finds the smallest eigenvalue of a symmetric positive definite matrix, with random memory accesses; and `EP` is an embarrassingly parallel Monte-Carlo algorithm, which is not memory-intensive. `BFS` is the breadth-first search algorithm from the Graph500 benchmark [34], running at a graph scale of 24, featuring random memory accesses.

We first assess the overall performance behavior of the four programs, with number of processes (job scale) fixed at 16, when spread onto a growing number of cluster nodes (same platform as in Section 1). The processes are evenly divided across nodes for load balance. Figure 2 shows varied scaling behavior: `MG` benefits the most from spreading, followed by `CG` and `EP`, while `BFS` performs

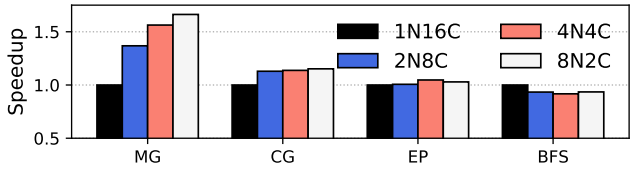

Figure 2: Sample scaling behavior of 16-process runs (N: Nodes, C: Cores per node)

best when running on a single node. Below we discuss follow-up experiments to understand these scaling behaviors.

Memory bandwidth As a reference, we run the popular STREAM memory bandwidth benchmark [48] using increasing number of cores within a single node, as plotted in Figure 3 (*y axis in log scale*).

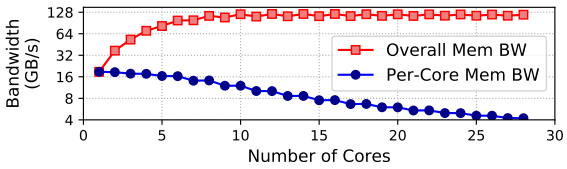

Figure 3: STREAM bandwidth w. growing number of cores

With a single core performing sequential memory access, STREAM reaches its peak bandwidth (red curve) at 18.80GB/s. With 2 cores, the peak doubles to 37.17GB/s. As more cores are used, the linear growth in aggregate bandwidth quickly stops and levels off around 8 cores, reaching 118.26GB/s at full capacity (with all 28 cores used).

The blue curve plots *per-core* bandwidth, which exhibits a consistent declining trend. At 28 cores, it dips to 4.22GB/s, only 22.45% of observed single-core peak. This demonstrates the rather early onset of the memory bandwidth bottleneck on modern multi-core processors when a parallel program has homogeneously bandwidth-hungry processes.

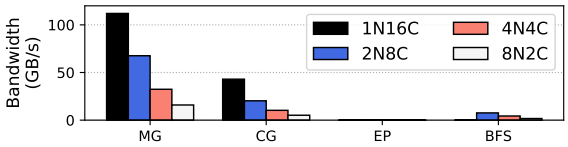

Figure 4: Per-node memory bandwidth consumption

We now turn to our programs, using the Linux `perf` tool [24] to measure bandwidth consumption. Figure 4 shows that when running on a single node, the average memory bandwidth consumption of `MG`, `CG`, `EP`, and `BFS` is 112.0GB/s, 42.9GB/s, 0.09GB/s, and 0.12GB/s respectively. Therefore, while the other three are rather bandwidth-light, `MG` is bound by this resource type, consuming nearly the STREAM-measured aggregate peak. In fact, when it runs on two nodes, each occupies an aggregate bandwidth of 67.6GB/s, bringing the program overall to 135.2GB/s, around 20GB/s above the per-node peak. Such bandwidth-hungry jobs are clear beneficiaries of spreading.

Shared LLC Besides memory bandwidth, the shared last level cache forms another resource contention point, especially among homogeneous processes. Here we measure our programs' LLC miss rate, again using hardware counters.

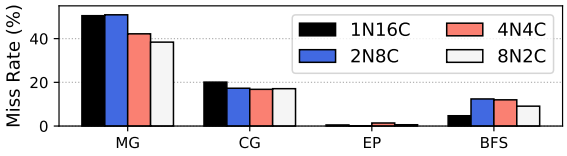

Figure 5: Impact of scaling on LLC miss rate

Figure 5 shows the cache miss rate dropping for `MG` and `CG` when scaling out, thanks to increased LLC capacity. For `EP`, whose cache miss rate is very low to begin with, the small changes in cache miss rate contribute little to performance. For `BFS`, scaling out actually incurs much *higher* miss rates, caused by communication-related code/data access.

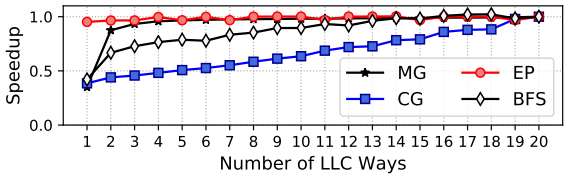

Figure 6: Performance normalized to full LLC ways

To further understand these programs' cache behavior, Figure 6 depicts each program's cache sensitivity when running on a single node, while varying LLC way allocation via Intel's CAT technology (which also affects cache associativity). Naturally, such sensitivity is highly application-dependent. E.g., `MG` needs only 3 LLC ways to achieve 90% of the performance under full cache capacity. The saturation points for `CG` and `BFS`, on the other hand, are 10 and 18 ways, respectively. `EP` is insensitive to cache capacity. Such diverse behavior again motivates mixed job placement.

Network communication An obvious counter factor for scaling out is network communication: a larger node footprint would incur higher inter-node traffic. Even with fast IB, network bandwidth (6.8 GB/s on our test platform) is still far below that of intra-node communication via shared memory, plus latency is higher. Fortunately, although the gap between memory and network bandwidth looks dramatic, scaling out a program may not hurt performance proportionally, since network communication happens with lower intensity compared to memory accesses.

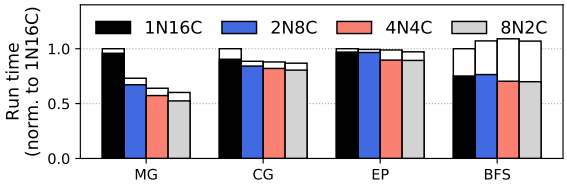

Figure 7: Time breakdown between computation (colored) and inter-process communication (white)

Figure 7 demonstrates this with mpiP [57] profiling results, breaking down a program's execution into computation and communication time, normalized to the single-node total execution time. For

the three NPB programs, communication accounts for under 10% of their total run time, and their gain from computation efficiency when scaled out more than offsets any loss in communication performance. Interestingly, CG actually benefits from spreading *communication wise* as well. Under CE, CG incurs wait time for late senders/receivers. Running on more nodes, processes have less resource contention and smaller progress gaps, significantly mitigating such waits. When spread, BFS executes different instruction flows for inter-node communication, generating pressure on the memory hierarchy (see higher bandwidth and LLC miss rate in Figure 4 and Figure 5). In addition, BFS's computation time on two nodes is longer than on one, making it the only program in this group *not* benefiting overall from scaling out.

3 SYSTEM OVERVIEW

3.1 Problem Definition

SNS adds resource-aware job scaling and co-location to a conventional batch scheduler's duties. Besides common input such as cluster capacity and incoming job sequence, it takes as input:

- the scaling profile of each program, accumulated from prior production runs, which summarizes the programs' performance behavior when spread to different node footprints, In particular, each profile captures a program's sensitivity to LLC capacity (details in Section 4.1).
- for each job, a user-specified *slowdown threshold* α that indicates its tolerance to performance degradation from sharing nodes with other jobs (e.g., an α value of 0.9 indicates that a job needs to retain 90% of its peak performance when running solo), and
- the total available manageable resources per node, including main memory bandwidth and LLC ways.

SNS can be plugged into common baseline scheduling policies, such as FIFO or priority-based algorithms. At each scheduling point (usually, when the system starts or when a job finishes), SNS works on top of the baseline scheduling algorithm, and estimates the resource requirements of jobs to be scheduled. An SNS-enabled scheduler (1) selects the job to be dispatched, (2) decides its desirable execution scale (how many nodes to spread it onto), (3) selects the node(s) to run it on, and (4) adjusts per-node resource allocation for it as well as other jobs sharing nodes with it.

3.2 SNS Terms

Next we define key terms to be used in our discussion. We hereafter use the term *cluster* to refer to a parallel platform in general, covering in-house clusters, datacenters, supercomputers, and cloud-based virtual clusters. A *node* refers to a physical compute server in a cluster. A *job* refers to an application instance to run on a cluster, and a *program* to the application itself (i.e., an executable binary).

We introduce two concepts for parallel job scheduling decisions. First, *scale factor* indicates the degree of "spreading out" in running a job. A job with a scale factor of 1 runs on as few nodes as possible (using the minimum node footprint as occurring under CE scheduling). A job with a scale factor of k uses $k \times N$ nodes, where N is its minimum footprint. Second, the *node mode* indicates how a node is utilized: E (exclusive, running at most one job at any time) or S (shared).

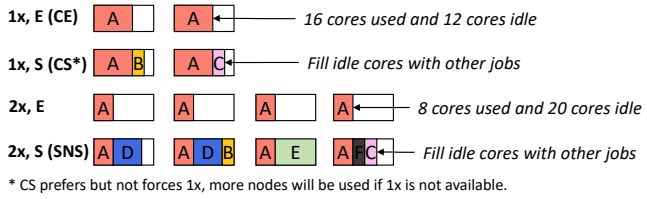

Figure 8: Alternative scheduling policies

Figure 8 illustrates the above using a 32-process job A and several other jobs, running on 28-core server nodes. With the **CE** policy (scale factor 1, node mode E), job A occupies 2 nodes and uses 16 cores of each for load balance, leaving 24 idle cores in total. By relaxing the exclusive constraint, one obtains an intermediate policy, **CS** (Compact-n-Share: scale factor 1, node mode S), which reduces core wastage. In addition to allowing node sharing, CS does not force a scale factor of 1 when available resources do not permit such compact placement, but schedules the job with the lowest scale factor currently possible. In this example, with CS, job A also occupies 2 nodes and uses 16 cores of each, while two other jobs, B and C, utilize the remaining cores of these nodes. Our proposed **SNS** policy adds automatic scaling to CS's node sharing, plus resource-aware job co-location. In this case, it adopts a scale factor of 2 for job A, spreading it to 4 nodes, pairing it with other jobs based on their resource demands, and performs workload-aware per-job resource allocation on each of the 4 nodes.

3.3 Overall Approach and Architecture

SNS performs resource-aware parallel job scheduling by focusing on *shared* resources within each node. The overall goal is to improve cluster throughput while protecting the performance of individual jobs. The latter can be specified by users (the acceptable slowdown factor α). Based on each program's resource consumption profiles, it performs (1) scaling out of individual jobs to reduce self-contention ("spreading"), (2) resource-aware co-location of jobs to improve utilization of server cores and other resources ("sharing"), and (3) workload-aware cache way allocation to enhance performance and inter-job isolation.

Note that the above tasks integrate into a tightly-coupled decision making process, instead of a list of sequential steps. First, the "spreading" decisions are made within the constraint of current node availability, rather than based solely on a certain program's scaling behavior. Secondly, a program's usage patterns of different resources are not independent, the most obvious case being a process's memory bandwidth consumption significantly influenced by the amount of LLC capacity it is allocated. Finally, when different applications have conflicting demands constrained by overall resources, the decision needs to search for an SNS plan that optimizes global utilization/throughput.

In this proof-of-concept study, we focus on a specific scope in resource management: memory bandwidth and LLC capacity. Unlike CPU cores, these lower layers of memory hierarchy are shared by processes, with much lower degrees of isolation. Meanwhile, the increasing core density of modern servers makes them more

bottleneck-prone than ever [64]. In addition, the availability of hardware per-core LLC way allocation in newer Intel processors [39] enables workload-aware cache capacity management, which is crucial to SNS's functioning. In this paper we do not explicitly quantify the impact of SNS on disk locality, as supercomputers and many clusters today do not have node-local storage but use shared parallel/distributed file systems, and disk locality has been shown as not performance critical in datacenter computing [7]. In addition, our profiling (see Subsection 4.1) measures the limit of spreading a program, which captures negative impact from I/O as well if there is any (I/O intensive applications typically benefit from spreading out due to enlarged aggregate bandwidth). Nevertheless, the above SNS decision making can be easily extended to consider other resource types, such as inter-node network and I/O bandwidths, whose management will add orthogonal dimensions to the current memory-oriented decision making, and can be accommodated by the SNS scheduling algorithm.

Our current approach is based on continuous workload profiling, using our own lightweight monitoring tool named Kunafa[2] to gauge their resource use patterns. In particular, we use hardware performance monitoring units on modern processors to profile the instruction-per-cycle (IPC) value and memory bandwidth consumption of a program, with different numbers of cache ways. The generated profiles can then be used with any future job submissions (details in Section 4.1). Nowadays, both HPC and commercial platforms often run recurring jobs whose code and data properties remain stable across multiple runs [3, 18, 45, 46], thus enabling the recurring usage of profiling data.

We design Uberun as a scheduler prototype supporting SNS. It works on top of existing parallel frameworks, nowadays indispensable and often co-existing on shared platforms [35, 55]. Our implementation and evaluation center around three popular frameworks: TensorFlow [2] (machine learning), Spark [8] (data analytics), and MPI [31] (HPC).

In current common practice, such frameworks are used in the CE manner: users typically request a certain number of nodes, where they initiate a parallel program's execution on top of a framework. A certain server node is running one framework at a given time. While existing meta-schedulers like Mesos [35] target sharing a physical cluster among multiple parallel frameworks (such as Hadoop and MPI), they divide this physical cluster into partitions that are assigned to each individual framework, which are in turn responsible for scheduling parallel jobs of the corresponding type *within the given partition*. Though the partitions themselves are not fixed, adjusting their boundaries is a rather heavy-weight operation and may force termination of running jobs.

Uberun, on the other hand, schedules at the job-level and co-locates jobs across all parallel frameworks to enhance the utilization of each server node. By doing so, it generates more resource-complementary job combinations to benefit from SNS. It further adds additional management mechanisms to scale out individual jobs and perform inter-workload cache partitioning, which are not supported by existing meta-schedulers. To our best knowledge, this is the first work targeting detailed resource management for inter-workload parallel job scheduling. Additionally, meta-schedulers like Mesos become trivial when there is only one framework on a cluster, while Uberun still uses SNS to perform resource-aware job scaling, co-location, and intra-node resource allocation, all beyond the functioning of the original, single-framework parallel scheduler.

[2]https://github.com/qcri/Kunafa

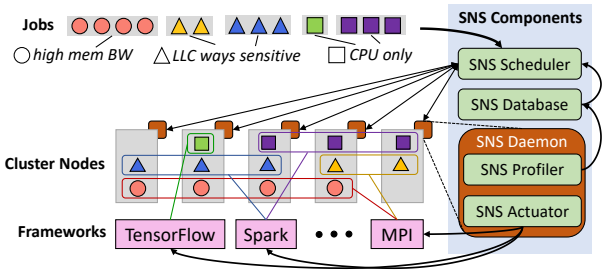

Figure 9: Architecture of Uberun

Figure 9 illustrates the architecture of our proposed SNS scheme, which takes an input job sequence and dispatches programs to co-run on cluster nodes with appropriate resource allocation. It comprises the following modules:

- **Daemon** This is a per-node component that carries out execution profiling and decision actuating, on each compute node. Its *profiler* (Section 4.1) collects lightweight PMU data to characterize the current job's resource usage, passing them to the central SNS database. Its *actuator* (Section 4.4), on the other hand, implements decisions made by the SNS scheduler, launching programs at the specified scale on top of appropriate frameworks, binding job processes to specific cores, and allocating LLC capacity as directed.
- **Database** This central component receives monitoring data from the per-node daemons and maintains historical information on parallel programs, storing per-program resource usage statistics (Section 4.1).
- **Scheduler** Residing on a dedicated server together with the database, this central component integrates the attributes of pending jobs, the per-program profiles from the database, and the current resource status information to make scheduling decisions (Section 4.2, 4.3, and 4.4).

The node zoom-in in Figure 9 gives a hypothetical example of five nodes shared by three programs. Job 0 is a CPU-only job and uses little bandwidth or cache. Job 1 is sensitive to LLC ways, and occupies over half of LLC. However, it consumes a small portion of memory bandwidth. Job 2, on the contrary, occupies only 4 LLC ways but consumes more than half memory bandwidth. By co-locating diverse programs on shared nodes, we are able to utilize the system resource in a more balanced manner.

4 DESIGN

4.1 Program Profiling

The SNS profiler monitors the following performance metrics for a program and generates its profile at different scales.

Execution time The profiler captures the execution time of a program's exclusive run at a certain execution scale. Typically, several such data points would reveal the scale that delivers the best application performance, which is related to the node size

(number of cores per node) but independent of the cluster size. Profiling exclusive runs removes inter-application interference. It also provides more accurate readings as with current processor monitoring hardware, the majority of performance metrics are only available at the node rather than core level.

Resource demand The resource demand information reflects the relationship between a program's resource consumption and its performance, in the form of the IPC-LLC and BW-LLC curves. The IPC-LLC curve captures the trend of the overall application performance (in instructions-per-cycle) relative to the number of LLC ways allocated, while the BW-LLC curve captures that of the application's measured memory bandwidth. Steeper curves indicate higher sensitivity to LLC allocation. Note that in general, a larger LLC allocation does not bring negative impact to a program's IPC, but it could affect its memory bandwidth both ways: it may lower bandwidth by reducing cache misses, or raise it by speeding up memory accesses with fewer misses.

The above metrics form the basis for our current scheduling, using a framework capable of accommodating new resource metrics collected in the future. Also, SNS currently targets scheduling at the job granularity, therefore taking all metrics as average values (of all processes). More detailed profiling is a rather straight-forward extension to enable job/process migration or dynamic scaling.

For each application, SNS performs the above profiling work at different scales. The search space of *scale* is relatively small in practice and only a few profiling runs are required for a new application: the range of scale factor corresponds only to the number of cores per node, regardless of the number of nodes in a cluster. On typical processors used in today's supercomputers and datacenters, 1-4 unique power-of-two scale factors need to be tested. Also, profiling is terminated when an application's scaling out "saturates", where further spreading it does not help.

Rather than performing dedicated profiling runs, we piggyback per-program scaling-out profiling on *normal* runs, by trying out different spreading plans. Given that many programs are repeatedly run on the same platform [3, 18, 45, 46] and scaling factors have quite limited ranges, we can converge to an optimized plan within several trials. This also implies that a new application can start to benefit from SNS scheduling quickly, after a few initial runs. Section 5.1 provides more details on profiling.

4.2 Single-Program Scaling

Our scaling-out profiling adopts a simple trial-and-error approach, always starting with a scale factor of 1× and node mode of E, i.e., the execution model under the conventional CE policy. During the run, the SNS profiler collects resource usage data and saves them in its database. The program's next execution will then be scaled out to 2× while remaining in the E mode (with processes divided as evenly as possible across nodes). This exploration continues until reaching a configurable scaling limit, such as with under x cores per node utilized, or seeing performance degradation above y%. At the end of these trials, the profiling data in the SNS database is used to classify each program into three categories:

- *Scaling*: performance benefiting from scaling out to more nodes. For such programs, the SNS database has identified its constraining resource type (such as memory bandwidth, LLC ways, or both), as well as its empirically identified ideal scale factor (the node footprint that delivers the shortest execution time for its run with specified number of processes).
- *Compact*: performance suffering from scaling out. Such a program should be run under the CE mode if possible. Programs with heavy network communication or high sensitivity to data locality are potential compact programs.
- *Neutral*: performance scale-agnostic, with execution time varying within 5% across the entire range of eligible scale factors. This type of programs become ideal "fillers" that, though not benefiting from scaling out themselves, are flexible enough to run at any scale without significant performance degradation.

4.3 Resource Allocation at Fixed Scale

This subsection describes the resource allocation for a given scale factor k, whose selection will be discussed in Section 4.4. For a given k, SNS performs customized resource allocation for a target job based on two factors: (1) its program's performance profiles (Section 4.1), and (2) a user-specified slowdown threshold α, which indicates how much performance degradation (from CE execution) can be tolerated with SNS's aggressive co-scheduling.

The slowdown threshold α is an optional parameter valued between 0 and 1, to be specified by users when submitting a job. It indicates how much slowdown the job owner can tolerate. Slowed jobs may be compensated by a datacenter or supercomputer, such as by receiving lower price or higher scheduling priority [16]. When unspecified, SNS adopts a default α value of 0.9; i.e., it attempts to keep the performance loss within 10%.

Suppose each node has T cores, given the total number of processes P requested by the target job, and the selected scale factor k, the job will be spread to $n = k \times \lceil P/T \rceil$ nodes, using $c = \lceil P/n \rceil$ cores per node.

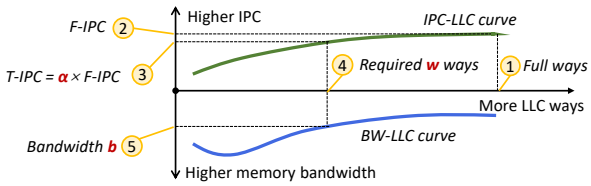

Figure 10: Estimating bandwidth and LLC demand

SNS relies on the IPC-LLC curve, as part of the per-program profile, to make LLC allocation, using IPC as a proxy metric for performance. Figure 10 illustrates its decision making, showing both the IPC-LLC (green) and the BW-LLC (blue) curves. Starting from the IPC at full cache way allocation (F-IPC), SNS estimates the *tolerable IPC (T-IPC)* as $\alpha \cdot$ F-IPC (steps 1-3 in Figure 10). Using the IPC-LLC curve, SNS finds w, the mininum LLC ways needed to achieve T-IPC, while the BW-LLC curve gives b, the estimated average memory bandwidth consumption at such LLC allocation (steps 4-5).

4.4 Job Scheduling and Resource Allocation

Finally, we put all things together and describe the integrated SNS scheduling and resource allocation process. To *simultaneously*

enhance resource-intensive applications' performance and improve the overall cluster throughput, SNS performs scheduling and co-placement using an age-based priority algorithm, considering the per-job demands in cores, LLC ways, and memory bandwidth. The overall scheduling process is shown in Figure 11.

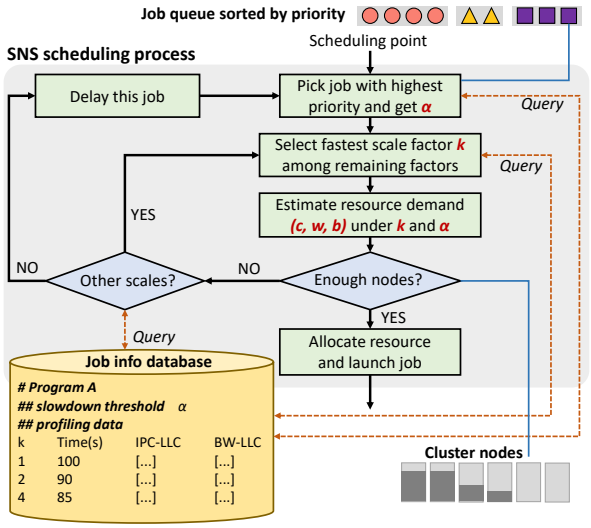

Figure 11: SNS scheduling process

For the job with the highest priority to be scheduled, SNS estimates its per-node resource requirement (by looking up the profiled curves as described in Section 4.3), with the best-performing scale factor. Then SNS tries to find enough nodes with sufficient resource currently available, *i.e.*, n nodes each with c cores, w LLC ways, and b memory bandwidth. If the search fails, SNS will evaluate sub-optimal scale factors, in descending order of profiled exclusive run overall performance.

When resources can be located for the scale factor currently assessed while meeting the degradation constraint α, the search terminates and the job gets dispatched, with the amount of residual resources deducted accordingly from the nodes assigned. To reduce node fragmentation, SNS first clusters nodes into groups by their idle cores count, then attempts to schedule a job onto nodes within a single group, to maintain approximately equal consumption of resources within such groups and facilitate future parallel jobs' placement. If the job cannot fit in any single group, SNS will expand its search to the whole cluster. In either case, SNS selects the k *idlest* nodes, for better load balance and less inter-job interference. In particular, SNS calculates a metric for each node by its occupied percentage of cores C_o, LLC ways W_o, and memory bandwidth B_o. It then selects nodes with the lowest values of $C_o + B_o + \beta \times W_o$. We assign a higher weight β to W_o (2 in our prototype) as we found LLC a more significant factor regarding performance interference within a node.

Once the n nodes are assigned, the SNS actuator launches the target job and implements the actual resource allocation via LLC way partitioning and additional optimizations such as core affinity binding. Memory bandwidth allocation can be done by Intel's recent MBA technique [39]. However, our test cluster (purchased in 2018) does not have MBA-equipped processors, therefore bandwidth allocation is done by estimating the total usage by jobs based on their individual profiles, assuming each job consuming the average bandwidth as observed in their solo runs profiled. To not waste cache resources, though SNS keeps record of available LLC ways beyond the total "allocated" amount to all active jobs running on a node, it gives away such unused resources to these jobs in equal shares, reclaiming them whenever a new job is dispatched to the same node. Finally, if the program happens to run in exclusive mode, the SNS monitor will take advantage of this opportunity to perform profiling (see Section 4.1) and update the profile database upon the job's termination.

If none of the scale factors enables the job to be immediately scheduled, the scheduler will delay its execution (with aging policies to promote its ranking) and evaluate the next job ordered by priority. A configurable "age limit" prevents starvation, so that resource-demanding jobs do not get delayed once reaching this limit.

5 IMPLEMENTATION

5.1 Prototype Implementation Issues

Coordinating with underlying frameworks We implemented SNS scheduling in Uberun, a prototype running on Linux. It has daemons on each cluster node, and a scheduler and database on a master node (recall Figure 9). Uberun stores profiling data in a JSON-format file and caches them as key-value pairs in memory at runtime. It performs scheduling atop existing parallel frameworks (i.e., MPI [31], Spark [8], and TensorFlow [2]) for multi-node job execution, and relies on the Linux `cpuset` process-core binding for fine-grained resource management within each node. MPI provides explicit interfaces for binding. For Spark, we use standalone mode and modify the available cores of spark worker daemons according to a job's resource allocation. For TensorFlow jobs, we explicitly set the number of cores per node in TensorFlow applications' code.

Profiling method To profile cache sensitivity, the Uberun monitor module periodically modifies LLC allocation at run time, sampling and recording the IPC and memory bandwidth during each of such fixed-allocation episodes. In particular, the PMU events *Instructions Retired*, *UnHalted Core Cycles* on *Core* [1], and *REQUESTS* on *Home Agent (a component controlling memory access)* [38] are used to obtain the IPC and memory bandwidth, respectively. The adjustment is repeated over time to capture different program phases. To reduce the overhead incurred by such adjustment (including cache flushing when allocation is decreased), our prototype sets the allocation adjustment frequency at once every 5 seconds. It further samples the LLC allocation range (at 2, 4, 8, and 20 ways only) and performs linear interpolation for missing data points in the cache sensitivity curves.

Though PMU sampling brings negligible cost, lowering LLC allocation does bring visible slowdown (average at 19% in our tests). Therefore, at each profiled scale, Uberun captures the total run time in a separate run without manipulating LLC allocation. For a new application, profiling needs to be performed for each scale. Fortunately, the number of possible scales is small. A typical modern processor, e.g., Xeon E5-2680 v4, has less than 30 cores, and 4 cores are required to saturate memory bandwidth (see Figure 3), limiting the scale factor to $30/4 < 8$ even with the most severe bandwidth

demand. In Uberun, we use four candidate scales: 1, 2, 4, and 8. In other words, at most 8 profiling runs are required for each application.

Cache allocation actuation With the processors on our test platform, the 20 LLC ways can be grouped into at most 16 partitions, supporting at most 16 jobs running on a node with disjoint LLC allocation. However, we found such high concurrency not practical for parallel jobs, as most programs need at least 2 ways, due to the dramatic loss of cache associativity when limited to a single way.

5.2 Road-map to Production Platforms

Integration into industrial schedulers We implemented Uberun in scripting languages (Python and Bash scripts) for concept validation, as this is needed to have any modified scheduler deployed on a production cluster. We perceive no technical challenges in rewriting in C/C++ and integrating it into SLURM [60] or other industrial schedulers. However, in our validation we work with fixed programs, while on production platforms programs may be modified between submissions and it would not be practical to perform a full set of profiling (piggybacked on production runs) between two adjacent code changes. On one hand, whether a program is friendly to scaling out or sensitive to cache/bandwidth allocation is often a stable feature that is expected to withstand most program modifications. On the other hand, there do exist significant program re-designs or accumulated gradual changes that eventually alter an application's relevant performance behavior. To this end, SNS-enabled production scheduler could perform sustained, light-weight monitoring on programs' key performance metrics, such as the distribution of IPC, cache miss rate, and memory bandwidth readings, to trigger re-profiling when deemed necessary. In addition, users could be provided with optional hint interfaces in job submission, to alert the cluster manager of major changes that warrant re-profiling.

Availability of core-level resource control Recent technology enabling fine-grained resource allocation among co-running applications sharing the same physical node are increasingly available. For instance, public clouds such as Amazon EC2 and Aliyun SCC have already deployed hardware supporting CAT, the LLC way partitioning mechanism employed by SNS [4, 5]. In addition, upcoming architecture features will provide more advanced resource allocation mechanisms, such as L2 cache partitioning, cache partitioning for processes instead of cores, and memory bandwidth partitioning [39]. Our proposed SNS and Uberun design is general enough to accommodate additional resource usage monitoring and resource-aware application co-scheduling.

Meanwhile, though hardware performance counters (PMUs) have been widely available for over a decade, they are not often accessible to cloud or datacenter application users. However, as SNS is intended to work from the cluster management side, PMU access should not be an issue.

Underlying support for job placement This work also confirms that compared to HPC parallel frameworks like MPI, popular commercial frameworks often lack support for fine-grained affinity setting [32], to use partial nodes and pin processes/threads to specific cores. E.g., we had to modify TensorFlow application code to control the number of cores to use per node. It is desirable for popular parallel frameworks to add such interfaces to facilitate resource-aware job scaling and placement.

6 EVALUATION

6.1 Setup and Methodology

Platforms We test on a local cluster with 8 compute nodes connected by EDR Infiniband (up to 50 Gbps). Each node has dual Intel Xeon E5-2680 v4 processors and 128 GB DDR4 memory, where each processor has 14 cores sharing 35MB LLC, with 20 ways to be assigned to subset of cores via CAT. We deploy frameworks and libraries, including HDFS, Hadoop, Spark, TensorFlow, and MPI, all from their official repository without modification.

Test Programs Since there is no real-world workload trace with LLC sensitivity or memory bandwidth information, we have to use benchmarks for experiments and simulations. We use a variety of common cluster workloads, performing tasks such as machine learning, data analytics, graph computing, and scientific computing. We select 12 programs, focusing on sampling different memory access behaviors, from widely used benchmark suites: 3 from HiBench [36], 4 from NPB [11], 1 from Graph500 [34], 2 from TensorFlow-Examples [10], and 2 from SPEC CPU 2006 [52]. We generate random job sequences with these programs to evaluate the efficiency of SNS, in system throughput and individual job execution time. Following common practice [21, 47, 52, 53], we use *arithmetic mean* for time (in seconds) and *geometric mean* for speedup or normalized time (dimensionless).

The test programs usually come with varied input sizes and we size them to make their execution time in close magnitude (50s to 1200s). For our 3 Spark programs, we use the `bigdata` test size for `WC` (Word Count), `large` for `NW` (NWeight, iterative computation of associations between graph vertices n-hop away), and `huge` for `TS` (Terasort). Our 2 machine learning programs are multi-threaded but unable to run on multiple nodes. Both run `10,000` iterations, and we select batch size of 32 for `GAN` (Deep Convolutional Generative Adversarial Networks) and 128 for `RNN` (Dynamic Recurrent Neural Network). For MPI programs, we adopt problem size `D` for `MG` (MultiGrid), `CG` (Conjugate Gradient), `EP` (Embarrassingly Parallel), as well as `LU` (Lower-Upper symmetric Gauss-Seidel), and graph scale 24 for `BFS`. Finally, as clusters typically also encounter many sequential executions, we use 2 SPEC CPU programs, `HC` (H.264 video Coding) and `BW` (Blast Waves fluid dynamics computation), with the `ref` problem size.

Test Program Scaling To recreate the common scenario on clusters utilizing multi-core processors, we submit multiple replicas of a sequential program as a parallel job (collection of independent but highly similar tasks). This is routinely done in applications such as image/text processing, visualization, bioinformatics, and computer architecture design simulations. MPI jobs, Spark jobs, and these replicated sequential jobs are all able to scale out and run on all eight nodes of our cluster.

Figure 12 provides cache profiling results for our test programs, showing the least number of LLC ways (out of 20) needed to obtain 90% of the performance at full LLC allocation, and the corresponding average memory bandwidth measured at that allocation level. I.e., the blue and pink bars (using left and right y axes, respectively) are the w and b values in Figure 10. Each program

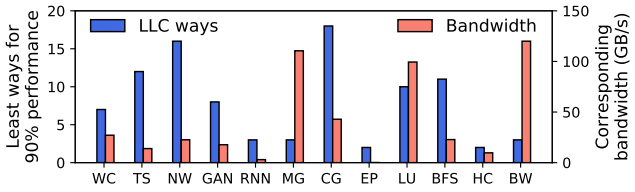

Figure 12: Cache sensitivity, tested with 16 cores on 1 node

.

runs with 16 processes/instances (8 cores per socket), to have a consistent level of concurrency across all programs, as the MPI programs require a power of 2 rather than arbitrary scale. The same number of LLC ways are allocated simultaneously across the two sockets. The results display diverse sensitivity to LLC way allocation, showing cache-insensitive applications (such as EP and HC) happy with only 2 LLC ways while cache-hungry ones (such as NW and CG) demanding almost all cache ways. Also, at such near-saturation LLC allocation level, the programs incur very different memory bandwidth consumption. With the traditional way of placing processes/tasks of the same application together on each node, such homogeneous resource demands often create either significant resource shortage or wastage.

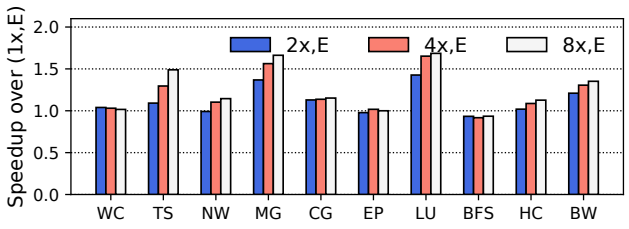

Figure 13: Speedup of scaling out, 16 processes used

Figure 13 summarizes the test programs' scaling behavior, showing the speedup of each 16-process/task run when spread onto 2, 4, and 8 nodes vs. their CE mode run on a single node, executing exclusively in both cases. Five programs (MG, CG, LU, TS, and BW) belong to the *scaling* class, with visible speedup when spread out. They also have different ideal scales: CG peaks at scale 2 (13% faster) and the other four scale all the way to 8 nodes, receiving over 30% speedup with respect to CE. The latter four scale well for different reasons: MG, LU, and BW are bandwidth-intensive, while TS enjoys larger caches for its sorting. SNS recognizes these scaling programs and unlike conventional schedulers, automatically spreads them out whenever possible.

Among the others, only one program (BFS) belongs to the *compact* class, due to the cost of inter-node communication. SNS carefully preserves such jobs' compact execution. Finally, four (EP, WC, NW, HC) are classifed *neutral*. Though they may gain slightly from higher CPU frequency, larger cache, or higher memory bandwidth, they are not significantly bound by these factors. Moreover, those advantages only appear with exclusive mode, and likely disappear when co-located with other jobs. SNS scales such programs *passively*, not for improving their performance but to utilize residual cores distributed on multiple nodes.

6.2 Overall Performance

Test case generation We evaluate Uberun with 36 randomly generated job sequences, each with 20 jobs sampled from our test program set and lasts for about 30 minutes on the 8-node cluster. We focus on the impact of SNS on both cluster utilization and individual programs' performance, therefore submitting all jobs the same time (to study a "time segment" of continuous batch job scheduling). A job uses 16 or 28 processes, to cover programs with rigid power-of-2 scale requirement and more flexible parallel/distributed applications (whose users often configure the number of processes/threads/tasks to match the core count per node). SNS adopts the default slowdown threshold, with α at 0.9.

For each job sequence, we define a metric, *scaling ratio*, to describe the fraction of jobs benefiting from scaling-out. It is calculated as the percentage of core-hours consumed by *scaling jobs*, based on their performance under CE. Our randomly generated job sequences are found to possess scaling ratios from 0.4 to 0.8 (see study on the impact of varying this ratio in Section 6.3).

System throughput To evaluate system throughput, we compare SNS strategy with CE and CS strategies. We do not compare against the state-of-the-art schedulers like SLURM, YARN, MESOS, or Quasar, because SNS is a resource allocation strategy that none of the other schedulers has enabled. The experiments are designed to validate its advantages over CE and CS. Since current schedulers have little support for MPI/Spark/TensorFlow mixed jobs, all three strategies are implemented in a prototype scheduler (Uberun), using a basic scheduling algorithm. However, this work is orthogonal to existing schedulers and scheduling algorithms; they can also use SNS strategy and consider LLC and memory bandwidth as manageable resources.

First, Figure 14 shows the overall cluster throughput improvement of SNS and CS, both over CE. Each data point represents a job sequence, with x and y values giving scaling ratio and normalized throughput. The overall throughput is defined as the reciprocal of the average submit-to-finish time of all jobs in a sequence.

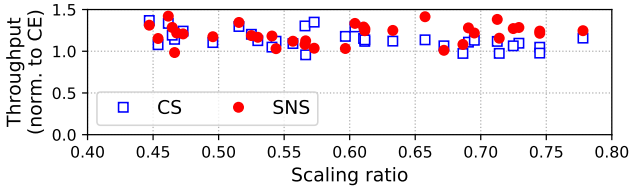

Figure 14: Throughput improvement of 36 random job sequences, ordered by scaling ratio along the x axis

Both CS and SNS improve the throughput, with average gain over CE at 13.7% and 19.8%, respectively. For CS, the improvement mostly comes from shorter wait time, as unlike CE, it does not waste idle cores. SNS, in contrast, could considerably speed up scaling jobs (which in turn benefits throughput), while schedule

and co-locate less aggressively than CS, due to its resource profiling guided scheduling (see the discussion of Figure 19).

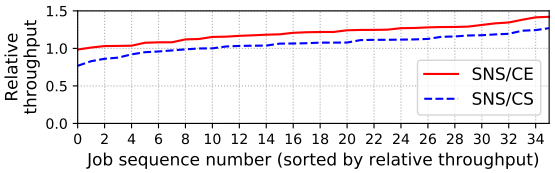

Figure 15: SNS throughput improvement over CE and CS

Figure 15 further shows the improvement brought by SNS, by plotting its relative throughput to CE (solid red) and CS (dashed blue) respectively. Job sequences are sorted in ascending order of y value, therefore the same x-axis index does not imply the same sequence. SNS improves CE except for one sequence (2% loss), as scaling out may in rare cases cause longer average wait time. However, as mentioned above, its improvement over CE is near 20% on average and up to 42.1%. Vs. CS, SNS wins for 26 out of 36 sequences. The 10 other cases it underperforms (by 9.1% on average, up to 23.2%), due to node fragmentation and more conservative co-scheduling. Still, it beats CS in 72% of the cases, with an average throughput improvement of 11.5% (up to 27.0%).

Individual job run time Figure 16 examines individual jobs' run time distribution. The blue and red solid lines trace average job run time of CS and SNS for each sequence, normalized to CE. The average is calculated as the geometric mean of the normalized run time of jobs within this sequence. The blue/red dashed lines mark the per-sequence maximum/minimum normalized job run time (the one speeded up or slowed down most from CE).

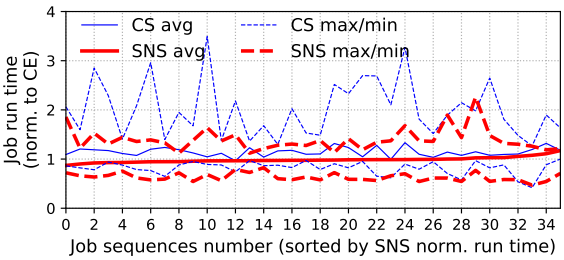

Figure 16: Run time of individual jobs

For all sequences, SNS-produced average run time is below that of CS (by spreading out scaling jobs), and is within 17.2% over CE (by performing resource-aware co-scheduling). While CS aggressively utilize cores available, it fails to harvest the potential of spreading out jobs for application performance enhancement (minimum run time significantly closer to the CE baseline), and fails to avoid co-placing resource-incompatible applications together (maximum run time much higher than the SNS curve, with up to 3.5× slowdown from baseline).

However, with our prototype implementation of SNS, a small subset of jobs (136 out of 720 executions) do experience violation of the slowdown threshold, exceeding the factor of 1.1 ($\alpha = 0.9$) by an average of 28.3% and up to 125.9%. One reason is inaccurate profiling: a program could be in different phases under different LLC allocations, producing phase-biased results that lead Uberun to under- or over-estimate the LLC ways needed. A second reason is the use of average bandwidth to guide scheduling, coupled with the lack of hard memory bandwidth control in our testbed, where programs may temporarily exceed their "bandwidth allocation". Future investigation on more detailed/advanced resource monitoring and upcoming hardware bandwidth allocation features will help alleviate the problem.

Comparing Figure 15 and Figure 16, SNS has 9 job sequences (out of 34) with longer average runtime than CE, while only one with lower throughput. This is because throughput is not only affected by runtime, but also by wait-time (submit-to-start). By allowing jobs to run at different scales, SNS not only exploits more resources, but also possesses less rigidness in scheduling, resulting in shorter wait time. Section 6.3 and Section 6.4 provide detailed results on SNS's impact on runtime and wait-time.

Load balance By spreading out resource-bound programs, SNS balances resource usage across nodes. Figure 17 shows the per-node memory bandwidth usage of a random job sequence, under CE and SNS respectively. The rows of the matrix represent cluster nodes 0-7, while the columns represent 60 30-second monitoring episodes. The deeper the color of a cell, the higher average memory bandwidth measured within the episode. Also, Figure 18 gives a histogram showing the memory bandwidth distribution for the episodes in Figure 17. Compared to CE, SNS has an obvious "smoothing" effect, reducing both the occurrences of near-peak and near-idle episodes. Consequently, the variance (standard derivation divided by peak) of average bandwidth across cells is 0.25 with SNS vs. 0.40 with CE. In other words, SNS has better load balance in terms of memory bandwidth usage across cluster nodes.

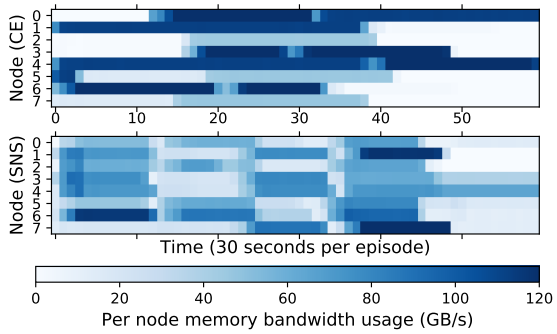

Figure 17: Load balance in memory bandwidth usage

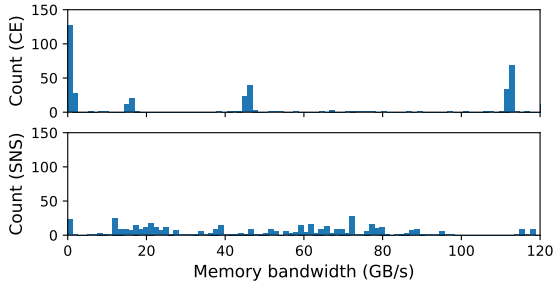

Figure 18: Episode count by bandwidth intervals

6.3 Impact of Workload Scaling Ratio

We now investigate the impact of the scaling ratio of a workload mix on the efficiency of SNS. To help control this ratio, we adopt simplified job mixes, with instances of BW as scaling jobs and HC as neutral jobs, and create 11 sequences with different scaling ratios, each containing 30 28-core jobs. Since all jobs occupy a full node, CS and CE behave the same and we omit CS from the comparison.

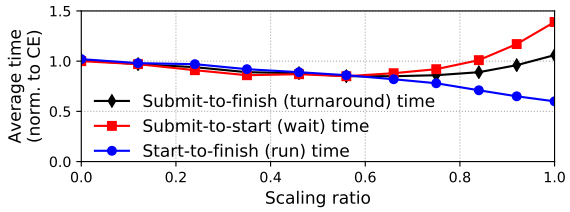

Figure 19: Impact of the ratio of scaling jobs

Figure 19 shows three metrics, all normalized to CE: the start-to-finish (run) time, the submit-to-start (wait) time, the submit-to-finish (turnaround) time. For each sequence, we plot the arithmetic average value of all its jobs.

For the job sequence without any job benefiting from scaling (scaling ratio at 0), SNS schedules all jobs with scale factor 1, therefore converging with the CE performance. As expected, when the scaling ratio increases, the SNS average job run-time (relative to CE) decreases, as more jobs gain from being spread out. This in turn has a positive impact on the average job wait time (submit-to-start), as faster executions make resources available sooner, till the scaling ratio reaches 0.75. Beyond that point, the SNS relative wait time to CE begins to grow, as Uberun makes early decisions to spread out most jobs considering their individual performance, leaving most nodes in partially-utilized states, further forcing later jobs to be scaled out. Compared to CE, such scheduling is less flexible in node selection, and results in higher ratio of idle "bubbles" in the schedule. We suspect that such fragmentation problem is highlighted by our small testbed cluster size: larger clusters, with node count typically much higher than most job scales, would provide large enough playgrounds with sufficient leeway to accommodate spread out jobs. Section 6.4 uses trace-driven simulation to verify such conjecture for larger platforms.

Still, SNS outperforms CE in *all three metrics*, for most of the scaling ratio range, improving both user experience/cost-effectiveness and overall cluster throughput. Between scaling ratios of 35% and 85%, it trims job turnaround time by over 10% on average from CE.

6.4 Simulation for Larger Clusters

We evaluate SNS on much larger clusters via simulation, driven by job traces from the LANL Trinity cluster [6]. From the traces, we only select parallel jobs and re-size them to fit in nodes with the same configuration as our local test cluster, to reuse job profile data. We also filter out the jobs using over 4,096 nodes. This results in 7,044 jobs in total, replayed according to the original submission timestamp on simulated clusters with 4,096, 8,192, and 32,768 nodes, lasting for 1900 simulated hours.

Since there are no public traces containing jobs' cache sensitivity profiles, we again map jobs in a trace to our 12-program test set given in Section 6.1, with 5 *scaling*, 4 *neutral*, and 3 *compact* programs. We generate two job sequences, with scaling ratios 0.9 and 0.5, to verify our aforementioned conjecture (Section 6.3). Again the scaling ratio here specifies the sampling bias between scaling and non-scaling programs (for jobs in a trace), with each group sampled uniformly. Once a simulated job has been mapped to a sampled program, we use the run time from the job trace as its CE run time. Additionally, we apply program-specific profile data to the simulated job, taken from real-system measurements: the program's speedup from scaling out, the LLC-BW curve, and the LLC-IPC curve. Figure 20 shows the average wait and run time, normalized to the CE average turnaround time.

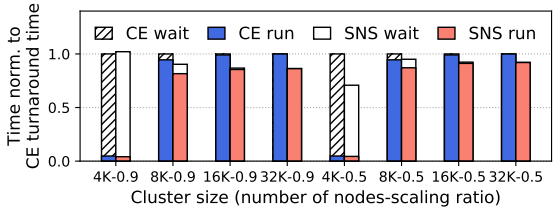

Figure 20: Run/wait time breakdown of CE and SNS

We first look at the results with scaling ratio 0.9. The 4K-node cluster is stampeded, with wait time dominating job turnaround, where SNS suffers slightly more due to its node fragmentation. As suspected earlier, job wait time drops with larger clusters, and the advantage of SNS over CE become more evident. On the 8K-node cluster, run time reduction far outweighs wait time increase. With 32K-node, SNS improves the system throughput by 15.7% over CE.

With scaling ratio 0.5, SNS has a significant improvement on the 4K-node cluster. The 4K-node cluster sees significant wait time reduction by SNS, as here the modest run time improvement (around 8%) brings large impact on wait time when the system is congested. On larger clusters, wait time is insignificant and SNS brings higher improvement on turnaround time at a scaling ratio of 0.9 than at 0.5, confirming our earlier hypothesis that larger clusters are more friendly to SNS's scaling out.

7 RELATED WORK

Batch job co-location Co-locating jobs on shared nodes can improve cluster throughput and energy efficiency [15, 42, 58]. However, the performance of *individual jobs* may be degraded due to resource contention (e.g., the LLC [51]). An array of approaches have been proposed in past work to make decisions on which applications are good for node sharing.

The Dominant Resource Fairness (DRF) approach [33] considers both CPU and memory usage while scheduling, but requires users to report their demands accurately. Breslow et al. propose a fair pricing model [16] for node sharing, giving users discounts based on the amount of interference from other users' jobs. Ginseng [9] applies a market driven approach for LLC allocation among co-placed programs, to maximize the overall social welfare. While also exploiting LLC allocation, our approach differs from Ginseng's as it requires no user-defined performance valuation functions, but instead uses an automatic algorithm to maximize throughput.

The most related work to ours are ClavisMO [12] and Poncos [14]. Both divide a physical node into two slots, classify jobs into shared-resource intensive and non-intensive groups, then assign resource-conflicting jobs to different slots to reduce contention. ClavisMO, along with other studies (e.g., [65]), examined shared network resources. Our work is different in several ways. First, our approach is designed for multiple types of shared resources, with memory bandwidth and cache ways in the current prototype but could easily extend to more resources such as I/O bandwidth or IOPS. Also, it not only decides which jobs can run together, but also actuates resources using hardware features such as CAT. Second, instead of statically having two slots per node, our algorithm performs flexible co-scheduling with more detailed profiling data. Third, our approach simplifies deployment as it does not rely on runtime process migration. In addition, one of this work's novelties, in contrast to all the above approaches, lies in that it targets multi-node programs and adds the dimension of automatic job scaling.

QoS-aware co-location Datacenter workloads include both latency-sensitive and batch jobs. Recent schedulers that explore co-locating often target mixed types, such as Mesos [35], Borg [56], and Bubble Flux [59]. They aim to improve the overall utilization while maintaining the quality of latency-sensitive services, mainly by prioritizing the latter and carefully offering idle resources to batch jobs. In contrast, our work focuses on the resource compatibility between multiple batch jobs sharing nodes. In this regard, it is similar to Quasar [26], a cluster manager where users specify their jobs' performance constraints for QoS-aware resource allocation; our approach considers more resource types than Quasar (LLC and memory bandwidth).

PerfIso [40] uses several spare cores as buffers to handle service bursts in SLO jobs. Since our approach focuses on batch jobs, it uses *all* cores for running jobs. Quartet [27] schedules together tasks that share data, and significantly increases the cache hit rate of Hadoop and Spark jobs. 3Sigma [49] leverages the job runtime historic distribution, rather than historic average, to improve the accuracy of prediction. Other characterization studies focused more on the virtual machine (VM) as a basic workload unit. Cortez et al. [22] propose a system that predicts CPU utilization using VM features such as size and type.

As far as we know, state-of-the-art QoS-aware schedulers use CPU utilization and memory capacity as two critical factors, while this work highlights the potential of taking LLC capacity and memory bandwidth into consideration.

Obtaining program resource demand Most aforementioned solutions, including our own, require knowledge of programs' resource demand, and could leverage exising profiling work [12, 25, 28]. Profiling is affordable, given the low overhead of current tools and the abundance of recurring jobs on common platforms [3, 18, 45, 46]. We consider center-administered profiling a more practical and portable solution than the alternative of requesting users to (accurately) specify resource usage of their jobs [9, 20, 33].

Exploiting workload diversity for co-location Workload diversity, observed in real-world clusters [6, 29, 30, 50], provides opportunity for researchers to explore how job co-location can improve system performance. E.g., Blanche et al. investigate the co-location of industrial software applications [23] and observe that node-sharing can improve overall throughput and reduce total monetary charge. Brown et al. [17] show that graph analytic programs may benefit from sharing nodes with HPC programs. The KPart LLC sharing machenism [28] allows latency-critical jobs to co-run effectively with batch jobs with QoS-aware LLC partitioning. Our proposal is in line with the above recent studies that exploit workload diversity, with particular focus on diverse *scaling behavior* and *multiple shared resources*. In terms of filling shared nodes by diverse workloads, more advanced packing algorithms [41] may help SNS further reduce fragmentation and improve overall throughput, and are orthogonal to SNS.

8 CONCLUSION

We propose a batch job scheduling strategy, Spread-n-Share (SNS), that improves *simultaneously* application performance and cluster utilization by resource-aware job co-execution. SNS scales out resource-bound applications to alleviate their performance bottleneck, and co-locates resource-compatible jobs on shared nodes. Our work also reveals LLC capacity and memory bandwidth as crucial shared resources, beyond CPU utilization and memory capacity studied in most prior systems. Experimental results confirm that SNS successfully exploits workload diversity and better utilizes modern multi-core cluster nodes. In particular, SNS extends and optimizes the common multi-tenant execution model on cloud platforms for HPC.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments and suggestions. Our work has been partially motivated by workload characteristics on clusters or supercomputers we have worked with, such as QCRI's in-house and cloud-based clusters, Raad at Texas A&M University Qatar, and Sunway TaihuLight at NSCC-Wuxi. We also thank Bowen Yu and Xu Ji for their valuable feedback. This project is partially supported by the National Key R&D Program of China (Grant No. 2016YFB0200100), National Natural Science Foundation of China (Grant No. 61722208). Jidong Zhai is the corresponding author of this paper.

REFERENCES

[1] Intel 64 and IA-32 Architectures Software Developer Manuals | Intel Software https://software.intel.com/en-us/articles/intel-sdm.

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: a system for large-scale machine learning. In *OSDI* (2016), vol. 16, pp. 265–283.

[3] Agarwal, S., Kandula, S., Bruno, N., Wu, M.-C., Stoica, I., and Zhou, J. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, pp. 21–21.

[4] Alibaba. Alibaba cloud instance type families https://www.alibabacloud.com/help/doc-detail/25378.htm.

[5] Amazon. Amazon ec2 instance types https://aws.amazon.com/ec2/instance-types/.

[6] Amvrosiadis, G., Park, J. W., Ganger, G. R., Gibson, G. A., Baseman, E., and DeBardeleben, N. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), USENIX Association.

[7] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. Disk-locality in datacenter computing considered irrelevant. In *HotOS* (2011), vol. 13, pp. 12–12.

[8] Apache. Spark Standalone Mode - Spark 2.3.1 Documentation https://spark.apache.org/docs/latest/spark-standalone.html, 2018.

[9] AssafSchuster, L. O.-Y. Ginseng: Market-driven llc allocation. In *2016 USENIX Annual Technical Conference* (2016), p. 295.

[10] Aymericdamien. Tensorflow examples https://github.com/aymericdamien/tensorflow-examples, 2018.

[11] Bailey, D., Harris, T., Saphir, W., Wijngaart, R. V. D., Woo, A., and Yarrow, M. *The NAS Parallel Benchmarks 2.0.* NAS Systems Division, NASA Ames Research Center, Moffett Field, CA, 1995.

[12] Blagodurov, S., Fedorova, A., Vinnik, E., Dwyer, T., and Hermenier, F. Multi-objective job placement in clusters. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for* (2015), IEEE, pp. 1–12.

[13] Bode, B., Halstead, D. M., Kendall, R., Lei, Z., and Jackson, D. The portable batch scheduler and the maui scheduler on linux clusters. In *Annual Linux Showcase & Conference* (2000).

[14] Breitbart, J., Pickartz, S., Lankes, S., Weidendorfer, J., and Monti, A. Dynamic co-scheduling driven by main memory bandwidth utilization. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)* (2017), IEEE, pp. 400–409.

[15] Breslow, A. D., Porter, L., Tiwari, A., Laurenzano, M., Carrington, L., Tullsen, D. M., Snavely, A. E., Snavely, A. E., and Snavely, A. E. The case for colocation of high performance computing workloads. *Concurrency and Computation: Practice and Experience 28*, 2 (2016), 232–251.

[16] Breslow, A. D., Tiwari, A., Schulz, M., Carrington, L., Tang, L., and Mars, J. Enabling fair pricing on high performance computer systems with node sharing. *Scientific Programming 22*, 2 (2014), 59–74.

[17] Brown, K., and Matsuoka, S. Co-locating graph analytics and hpc applications. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on* (2017), IEEE, pp. 659–660.

[18] Bruno, N., Agarwal, S., Kandula, S., Shi, B., Wu, M.-C., and Zhou, J. Recurring job optimization in scope. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD '12, pp. 805–806.

[19] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, omega, and kubernetes. *Queue 14*, 1 (2016), 10.

[20] Chandrasekar, K., Seshasayee, B., Gavrilovska, A., and Schwan, K. Task characterization-driven scheduling of multiple applications in a task-based runtime. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware* (2015), ACM, pp. 52–55.

[21] Citron, D., Hurani, A., and Gnadrey, A. The harmonic or geometric mean: does it really matter? *ACM SIGARCH Computer Architecture News 34*, 4 (2006), 18–25.

[22] Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 153–167.

[23] De Blanche, A., and Lundqvist, T. Node sharing for increased throughput and shorter runtimes: an industrial co-scheduling case study. In *HiPEAC Workshop on Co-Scheduling of HPC Applications* (2018), pp. 15–20.

[24] De Melo, A. C. The new linux perf tools. In *Slides from Linux Kongress* (2010), vol. 18.

[25] Delimitrou, C., and Kozyrakis, C. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 77–88.

[26] Delimitrou, C., and Kozyrakis, C. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices 49*, 4 (2014), 127–144.

[27] Deslauriers, F., McCormick, P., Amvrosiadis, G., Goel, A., and Brown, A. D. Quartet: Harmonizing task scheduling and caching for cluster computing. In *HotStorage* (2016).

[28] El-Sayed, N., Mukkara, A., Tsai, P.-A., Kasture, H., Ma, X., and Sanchez, D. Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on* (2018), IEEE, pp. 104–117.

[29] El-Sayed, N., and Schroeder, B. Reading between the lines of failure logs: Understanding how hpc systems fail. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on* (2013), IEEE, pp. 1–12.

[30] El-Sayed, N., Zhu, H., and Schroeder, B. Learning from failure across multiple clusters: A trace-driven approach to understanding, predicting, and mitigating job terminations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (2017), IEEE, pp. 1333–1344.

[31] Forum, M. Mpi specification http://mpi-forum.org/docs/, 2016.

[32] Garefalakis, P., Karanasos, K., Pietzuch, P. R., Suresh, A., and Rao, S. Medea: scheduling of long running applications in shared production clusters. In *EuroSys* (2018), pp. 4–1.

[33] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi* (2011), vol. 11, pp. 24–24.

[34] Graph500. Graph 500 reference code http://graph500.org/, 2018.

[35] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI* (2011), vol. 11, pp. 22–22.

[36] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on* (2010), IEEE, pp. 41–51.

[37] IBM. Lsf documentations https://www.ibm.com/support/knowledgecenter/en/sswrjv_10.1.0/lsf_welcome/lsf_welcome.html, 2018.

[38] Intel. Intel xeon processor e5 and e7 v4 families uncore performance https://www.intel.com/content/www/us/en/products/docs/processors/xeon/xeon-e5-e7-v4-uncore-performance-monitoring.html.

[39] Intel. Intel resource director technology https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html, 2018.

[40] Iorgulescu, C., Azimi, R., Kwon, Y., Elnikety, S., Syamala, M., Narasayya, V., Herodotou, H., Tomita, P., Chen, A., Zhang, J., et al. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 519–532.

[41] Johnson, D. S. Fast algorithms for bin packing. *Journal of Computer and System Sciences 8*, 3 (1974), 272–314.

[42] Koop, M. J., Luo, M., and Panda, D. K. Reducing network contention with mixed workloads on modern multicore, clusters. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on* (2009), IEEE, pp. 1–10.

[43] Kurth, T., Zhang, J., Satish, N., Racah, E., Mitliagkas, I., Patwary, M. M. A., Malas, T., Sundaram, N., Bhimji, W., Smorkalov, M., et al. Deep learning at 15pf: supervised and semi-supervised classification for scientific data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), ACM, p. 7.

[44] Lin, H., Zhu, X., Yu, B., Tang, X., Xue, W., Chen, W., Zhang, L., Hoefler, T., Ma, X., Liu, X., Zheng, W., and Xu, J. Shentu: Processing multi-trillion edge graphs on millions of cores in seconds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC18)* (2018), ACM.

[45] Liu, Y., Gunasekaran, R., Ma, X., and Vazhkudai, S. S. Server-side log data analytics for i/o workload characterization and coordination on large shared storage systems. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for* (2016), IEEE, pp. 819–829.

[46] Luu, H., Winslett, M., Gropp, W., Ross, R., Carns, P., Harms, K., Prabhat, M., Byna, S., and Yao, Y. A multiplatform study of i/o behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (2015), ACM, pp. 33–44.

[47] Mashey, J. R. War of the benchmark means: Time for a truce. *SIGARCH Comput. Archit. News 32*, 4 (Sept. 2004), 1–14.

[48] McCalpin, J. Memory bandwidth: Stream benchmark performance results https://www.cs.virginia.edu/stream/.

[49] Park, J. W., Tumanov, A., Jiang, A., Kozuch, M. A., and Ganger, G. R. 3sigma: distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference* (2018), ACM, p. 2.

[50] Rodrigo Álvarez, G. P., Östberg, P.-O., Elmroth, E., Antypas, K., Gerber, R., and Ramakrishnan, L. Hpc system lifetime story: Workload characterization and evolutionary analyses on nersc systems. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2015), HPDC '15, ACM, pp. 57–60.

[51] Simakov, N. A., DeLeon, R. L., White, J. P., Furlani, T. R., Innus, M., Gallo, S. M., Jones, M. D., Patra, A., Plessinger, B. D., Sperhac, J., et al. 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* (2016), ACM, p. 32.

[52] SPEC. Spec cpu 2006 benchmark set https://www.spec.org/cpu2006/, 2006.

[53] Tang, X., Zhai, J., Qian, X., and Chen, W. plock: A fast lock for architectures with explicit inter-core message passing. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019* (2019), pp. 765–778.

[54] Top500. Top 500 list on 2018 november https://www.top500.org/lists/2018/11/, 2018.

[55] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), ACM, p. 5.

[56] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 18.

[57] Vetter, J., and Chambreau, C. mpip: Lightweight, scalable mpi profiling.

[58] Wu, X., and Taylor, V. Using processor partitioning to evaluate the performance of mpi, openmp and hybrid parallel applications on dual-and quad-core cray xt4 systems. In *the 51st Cray User Group Conference (CUG2009)* (2009), pp. 4–7.

[59] Yang, H., Breslow, A., Mars, J., and Tang, L. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 607–618.

[60] Yoo, A. B., Jette, M. A., and Grondona, M. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing* (2003), Springer, pp. 44–60.

[61] You, Y., Buluç, A., and Demmel, J. Scaling deep learning on gpu and knights landing clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), ACM, p. 9.

[62] Zaspel, P., and Griebel, M. Massively parallel fluid simulations on amazon's hpc cloud. In *Network Cloud Computing and Applications (NCCA), 2011 First International Symposium on* (2011), IEEE, pp. 73–78.

[63] Zhou, A. C., Gong, Y., He, B., and Zhai, J. Efficient process mapping in geo-distributed cloud data centers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), ACM, p. 16.

[64] Zhuravlev, S., Saez, J. C., Blagodurov, S., Fedorova, A., and Prieto, M. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR) 45*, 1 (2012), 4.

[65] Zimmer, C., Gupta, S., Atchley, S., Vazhkudai, S. S., and Albing, C. A multi-faceted approach to job placement for improved performance on extreme-scale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), IEEE Press, p. 87.