A Quantitative Analysis of Node Sharing on HPC Clusters Using XDMoD Application Kernels

Nikolay A. Simakov, Robert L. DeLeon, Joseph P. White, Thomas R. Furlani, Martins Innus, Steven M. Gallo, Matthew D. Jones, Abani Patra, Benjamin D. Plessinger, Jeanette Sperhac, Thomas Yearke, Ryan Rathsam, Jeffrey T. Palmer

Center for Computational Research, University at Buffalo, SUNY, Buffalo, NY

ABSTRACT

In this investigation, we study how application performance is affected when jobs are permitted to share compute nodes. A series of application kernels consisting of a diverse set of benchmark calculations were run in both exclusive and node-sharing modes on the Center for Computational Research's high-performance computing (HPC) cluster. Very little increase in runtime was observed due to job contention among application kernel jobs run on shared nodes. The small differences in runtime were quantitatively modeled in order to characterize the resource contention and attempt to determine the circumstances under which it would or would not be important. A machine learning regression model applied to the runtime data successfully fitted the small differences between the exclusive and shared node runtime data; it also provided insight into the contention for node resources that occurs when jobs are allowed to share nodes. Analysis of a representative job mix shows that runtime of shared jobs is affected primarily by the memory subsystem, in particular by the reduction in the effective cache size due to sharing; this leads to higher utilization of DRAM. Insights such as these are crucial when formulating policies proposing node sharing as a mechanism for improving HPC utilization.

Categories and Subject Descriptors

C.4 [Performance of Systems]: *Design studies, Fault tolerance, Measurement techniques, Modeling techniques, Performance attributes, Reliability, availability, and serviceability.*

General Terms

Management, Measurement, Documentation, Performance, Design, Reliability, Verification.

Keywords

XDMoD, TACC_Stats, node sharing, performance co-pilot, SUPReMM, HPC

XSEDE16, July 17-21, 2016, ,

DOI: http://dx.doi.org/10.1145/2949550.2949553

1. INTRODUCTION

Large supercomputers typically execute jobs in exclusive mode, in which entire nodes are assigned exclusively to a given job, and no other jobs can access those nodes while that job is running. However, there are numerous cases in which a job cannot efficiently use all of the cores available on a node. Such situations are typical of serial applications, poorly scalable parallel software, or small problem sizes. This problem is exacerbated as core counts increase. Additionally, many HPC jobs are embarrassingly parallel tasks such as parameter sweeps, in which many **small** jobs run concurrently with varying input data. Accordingly, when such jobs are run in exclusive mode, cores can go unused, and the supercomputer may be underutilized.

Large supercomputing facilities often favor large parallel jobs to advance applied computational sciences and perform large-scale, previously unreachable simulations. Consider that out of all

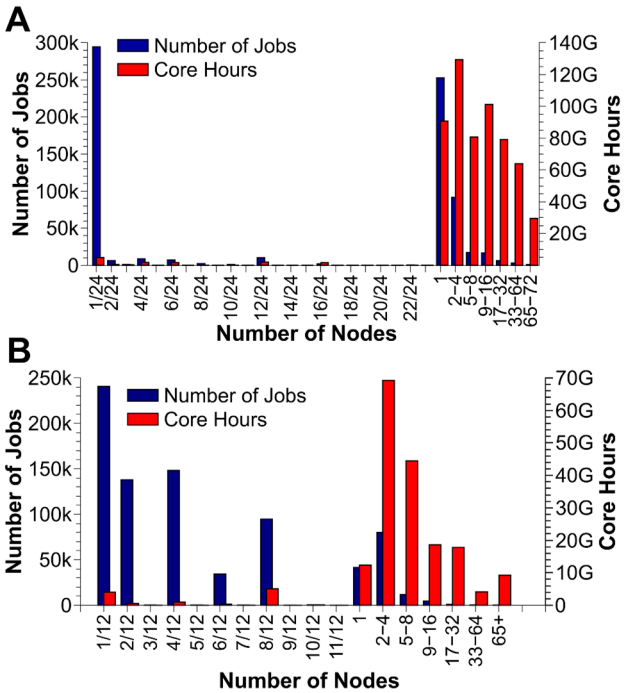

Figure 1. Number of core hours and number of jobs analyzed by node count on (A) Comet supercomputer at the San Diego Supercomputer Center (SDSC) and (B) the 12 cores per node sub-cluster at The Center for Computational Research, University at Buffalo (CCR). Jobs that use less than the entire node are analyzed by the number of cores used on that node. In 2015, node shared jobs comprised 80% of all jobs or 5.7% of all used core hours at CCR.

XSEDE resources, only one, Comet at SDSC, allowed node sharing as of early 2016; see Figure 1 A. However, academic HPC centers typically serve a diverse mix of disciplines with greatly varying computational needs that include serial and small parallel jobs. Node sharing is already a common practice at such centers, as shown in Figure 1 B. With the current trend for nodes to comprise increasingly larger numbers of cores, node sharing may ultimately be routinely implemented on the majority of HPC systems.

The increase in overall job throughput attained from node sharing can be significant, since it packs together jobs which do not fully occupy all cores on the node. Even when some nodes are shared by multiple jobs, and other nodes are fully occupied by single jobs, node sharing has been shown to increase throughput and energy efficiency in HPC clusters by 10-20% [1–4]. From a practical point of view, node sharing can increase the overall system throughput, especially when the job mix includes a substantial percentage of jobs that use fewer than the total number of cores per node. However, many users and resource providers may avoid node sharing, since it is unclear how much node sharing effects individual job performance.

Modern computers are designed to handle multi-processor loads and typically consist of multiple CPUs, each of which comprises multiple cores. For the present paper, we will consider a CPU to be the integrated unit encompassing a set of cores with their associated memory and caches. Each core has its own compute and cache units; while main memory (and for some CPU models, the last level cache memory) is shared between all the cores, current multi-channel memory is also designed to handle multi-processed loads. In fact, single-core processes cannot saturate the whole memory bandwidth. For example, on Intel Xeon processors, a single core is only able to obtain 15 to 25 percent of peak CPU bandwidth measured by the STREAM benchmark [5]. Other system resources, such as network and file systems, are already shared among all processes. However, the network is not critical for single-node jobs, and file systems are often used only sparingly since a large portion of HPC applications are compute or memory bound. There is an obvious advantage to sharing nodes, from both a scheduling and a resource efficiency perspective, as long as it can be demonstrated that the consequences of increased job contention are acceptable.

In a previous paper [6] we presented a statistical analysis based on TACC_Stats metrics, in which we compared the Center for Computational Research's (CCR) production HPC cluster (Rush), where node sharing is permitted, with TACC's Stampede and Lonestar4, which do not share nodes. Very little detectable job contention for system resources was seen, based on the similar distributions of various metrics (memory usage, cache usage, cache read miss rates, file write rates, IB network usage and core usage) between shared-node and exclusive-node jobs on Rush. We concluded that the adverse consequences of node sharing were few. However, the paper did not make a detailed study of the effect of node sharing on the execution time of shared and exclusively run jobs. This is an important consideration, since user allocations are impacted by execution time.

In the current paper, we take the node sharing analysis further, by actively running application kernels [7] in both shared and exclusive modes, then quantitatively comparing the results. In addition, we apply a machine learning model to the data in order to identify possible causes of runtime discrepancies and predict node-sharing related performance. The rest of the paper briefly discusses related work and concludes with a summary and a discussion of the benefits and consequences of node sharing.

2. ANALYSIS METHODS

This section provides an overview of the HPC cluster on which the study was performed, and describes the experimental setup; the problem sizes tested; the XDMoD Application Kernel and SUPREMM modules; the data collection methods; and defines the metrics used in the study.

The study was performed on CCR's x86_64 Linux cluster, Rush, which is a heterogeneous system containing 8, 12, 16 and 32 core nodes. For consistency, we performed calculations using only 12-core nodes. The 12-core sub-cluster consists of 368 Dell C6100 servers, each with two Intel "Westmere" Xeon six-core 2.40GHz (E5645) processors and 48GB of memory. Intel Turbo Boost Technology was off. All nodes in the sub-cluster are interconnected with QDR QLogic InfiniBand and gigabit Ethernet. The cluster has two shared file systems, a 3PB IBM GPFS high-performance parallel file system for the global shared parallel scratch space (the target of all I/O based application kernels), and Isilon network-attached storage arrays for general network file system access. Slurm was used as the cluster resource manager; it allows execution of jobs in both exclusive and shared modes. In shared mode, no manual control of node sharing is possible; also, Slurm constrains jobs to their assigned cores (cgroups) and limits memory usage. The default NUMA policy allows allocation on both NUMA nodes. The operating system was not rebooted or otherwise reset between jobs.

To study the effect of node sharing on application performance, several applications and benchmarks with the same input parameters were executed repeatedly in shared mode, in which multiple jobs can run on a single node, and in exclusive mode, in which a single job is constrained to run on a given node. Two job sizes were tested. The first job size uses one core out of a total of 12 cores on the node. This is the smallest possible job size, and the most common job size among node-sharing jobs at our facility; see Figure 1B. Single-core jobs have the greater node resource sharing, including L3 cache, memory controller, file systems and network. The second job size uses a single CPU (six cores) out of the two CPUs on the node. This test enables us to study the sharing effect that occurs when one job is encapsulated on a single CPU and the number of shared node resources is thus reduced. In exclusive mode, all jobs were executed on a single six-core CPU. In shared mode, the job's cores were assigned by the scheduler, and the distribution among CPUs was not known *a priori.*

Application execution and output parsing was performed using the XDMoD Application Kernel performance monitoring module [7]. This module performs repetitive automatic execution of customized "application kernels" on an HPC system to continuously monitor its performance and reliability (quality of service). An application kernel is a combination of an application with a selected set of input parameters. These parameters are chosen so that the application will consume modest resources (execution time) on the HPC system, while providing quality of service information. A number of different application kernels are routinely run on XSEDE's resources, and at our facility, to track quality of service (QoS) [7]. In this article we use the following set of application kernels: NWChem [8], GAMESS [9], NAMD [10], Enzo [11], Graph500 [12], HPCC [13] and IOR [14]. Detailed descriptions of these application kernels are given in reference [7]. Briefly, HPCC is a general benchmark application for several computational schemes; NWChem is a general molecular calculation application; GAMESS is a molecular electronic structure calculation application; NAMD is a molecular dynamics application; IOR is a parallel files benchmark; and Enzo is an adaptive mesh astrophysics application.

Table 1. List of the top 10 applications executed simultaneously with application kernels by other users

Application	Frequency	Description
ORCA	625	Quantum Chemistry
perl	146	n/a
java	123	n/a
CHARMM	115	Molecular Dynamics
python	56	n/a
Titan2D	32	Geophysics
VASP	25	Quantum Chemistry
LAMMPS	18	Molecular Dynamics
GROMACS	13	Molecular Dynamics
GraSPI	11	Bioinformatics

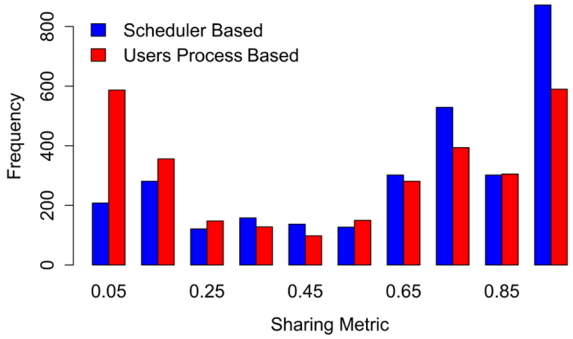

Figure 2. Node sharing metric distribution. Exclusive jobs are not included in this distribution.

The performance measurements were obtained from the Application Kernel performance monitoring module and the XDMoD SUPReMM performance monitoring module. The Application Kernel module records information from the applications themselves, such as the application execution time (wall time) and the FLOP/s rate for subroutines within the application (e.g. FLOP/s of matrix multiplication in HPCC). The SUPReMM module records information from the operating system and hardware on the compute nodes, and does not require additional process instrumentation. The data from the SUPReMM module includes application identification for all jobs, process performance measurements, operating system resource utilization, and hardware counters. Individual performance metrics were collected by running Performance Co-Pilot (PCP) [15] on the compute nodes. The wall time figures employed in this study measure only the application execution time, and do not include the time taken by preparatory steps such as copying input files or the scheduler's prolog and epilog processing. Both the Application Kernel and SUPReMM modules are currently available for installation within Open XDMoD (http://xdmod.sourceforge.net/) [16].

Not all cores in a node are necessarily assigned to a job while a test job is running there. Therefore, we define two metrics to quantify the amount of sharing that occurs during test job execution (a test job is a job we submit). The first, called the **scheduler-based sharing metric,** takes into account whether jobs were assigned cores on the shared node, but does not consider their activity. For example, if all of the cores on the node were assigned to jobs for the entire duration of the test job, then the scheduler-based metric is one. However, if the node ran only the test job then the metric is zero. Note that we are submitting the jobs as a user and have no control over how Slurm schedules them. The scheduler-based sharing metric only looks at whether or not jobs were assigned to the node during the execution of the test job, and does not consider their actual activity or resource usage.

We define a second metric, the **user-process-based sharing metric,** to account for the actual core activity of the jobs that are running alongside the test job. This metric is defined as the total core time used by all jobs *excluding the test job,* divided by the total core time possible for the non-test jobs. For example, if every other job scheduled to run with the test job had no core activity, the user-process-based sharing metric would be zero, while the scheduler-based metric could lie anywhere between zero and one. In both metrics, zero corresponds to execution of the test job without any node sharing, and one corresponds to complete sharing when all cores of the node are occupied the entire time the test job is running.

To summarize the metrics: The scheduler-based sharing metric provides a simple measure of core occupancy by jobs, while ignoring the actual resource usage of the jobs. Such a metric is convenient for estimation of shared-node effects under typical loads. The user-process-based sharing metric includes the actual core usage for the other jobs on the node, and can be useful for regression analysis of node sharing effects. In the ideal case, all jobs on the resource have 100% core usage then the scheduler-based and user-process-based metrics are identical.

3. DATA AND OUTLIERS

Data from more than five thousand Application Kernel test jobs were analyzed. 56% of the test jobs were executed in exclusive mode and 44% in shared mode. 13% of the test jobs were shared entirely with other jobs of the same test user, 18% were shared exclusively with other users' jobs and 12% were shared simultaneously with other users' jobs and test users' jobs. The various test jobs shared the node with a wide range of applications from different scientific fields executed by other users (Table 1). Some of these applications were identified as a general purpose programming language, since application detection is based simply on the executed binary name. Sharing metrics show good coverage, ranging from low levels of node sharing to high levels of sharing, in which the whole node was occupied during test job execution (Figure 2).

A number of outliers were removed from the analysis. These outliers show substantially longer runtimes, regardless of whether they are shared or exclusive jobs, and therefore have no utility in the present analysis. They fall into three classes: A) In 2.0% of all jobs, longer runtimes were caused by a known issue with the IPMI driver in the Linux kernel. Namely, a kernel thread trespassing on the cores where test jobs were running resulted in excessive core usage. This issue was subsequently resolved by a kernel update. B) In 0.7% of all jobs, processes belonging to other jobs ran on cores assigned to test jobs. In this case, the rogue processes escape from the cpuset and memory cgroups assigned by Slurm, which can happen with applications that use task startup methods that do not interoperate well with Slurm. Even if Slurm failed to constrain users' processes, the operating system can distribute process among available cores. Therefore, it is not surprising that most of these jobs' wall times remain within a few percent of the mean wall time for exclusive jobs. Only a few cases exceeded 10% of this figure; these jobs were removed from further analysis. C) Another 1.0% of jobs had wall times exceeding three standard deviations from the mean, and were removed from this analysis. The causes of the excessive wall times for these jobs have not been determined; however, it is assumed that these jobs have the same issues described above, but the causes were not identified by PCP.

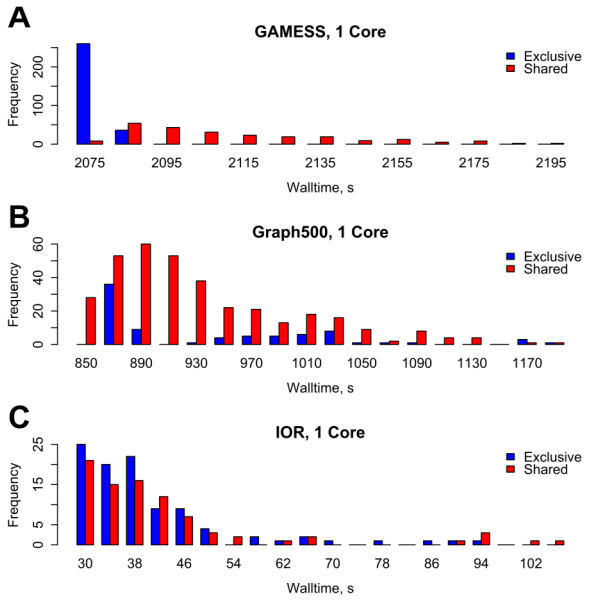

Figure 3. Wall time distribution of GAMESS (A), Graph500 (B) and IOR (C) single-core jobs. The distributions of NWChem, NAMD and HPCC are qualitatively similar to that of GAMESS.

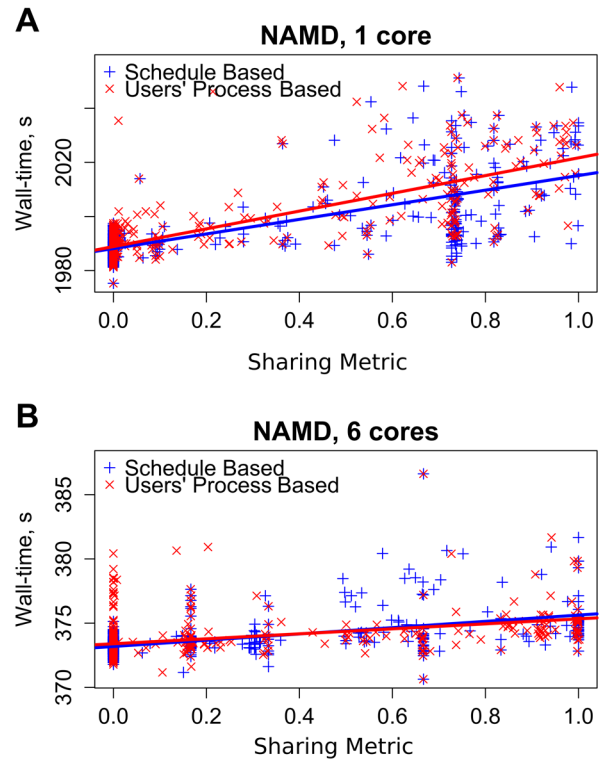

Figure 4. Regression of wall time against the node sharing metrics of the NAMD application kernel executed on a single core (A) and six cores (B).

4. SINGLE-CORE JOBS

We consider the difference between running the application kernels in exclusive mode versus on a single core within a node. As expected, all computationally intensive application kernels executed in exclusive mode ran faster than those executed in shared mode. On average, the wall time difference for GAMESS, NWChem, NAMD and HPCC is within several percent, reaching a maximum of 2.6% for NWChem (Table 2). In the case of jobs with a user-process-based sharing metric higher than 75% the highest average wall time difference for computationally intensive application kernels was 4.6% for NWChem. The wall time distributions of these application kernels are very similar (see Figure 3.A). They are positively skewed (long tail on the right side, that is, to longer runtimes); the distribution width is wider for shared jobs; the peak of exclusive jobs is located to the left of the shared jobs peak; and the minimal wall time is nearly identical between exclusive and shared modes. These distributions are positively skewed due to the application kernels having a minimal time to run and a long tail that is caused by various interferences that increases the runtime of the node-sharing jobs.

The dependence of wall time on the sharing metric for NAMD is shown in Figure 4. GAMESS, NWChem and HPCC exhibit a similar dependency. A regression analysis shows that these applications have very low p-value (<< 0.001) and a low R-squared value (Pearson regression correlation coefficient squared, ranging from 0.3 to 0.6). This implies that there is a statistically significant dependence on the shared metric (both scheduler and user-process variants). However, the large variance shows that though the regression captures the general trend, there are other unexplained sources contributing to the wall time variation. Graph500 also exhibits a strong dependence on the sharing metric (p-value << 0.001). However, it has a higher variance, due to differences in memory distribution between NUMA nodes.

Graph500 wall time distributions differ from the previously mentioned distributions in having two distinct peaks in exclusive mode (Figure 3.B). Graph500 has moderate memory usage (~5.5 GB), and further analysis showed that the first peak is associated with most of the memory being allocated on the nearest NUMA node, while the second peak is comprised of jobs that have a significant portion of the memory being allocated on the furthest NUMA node. The mean wall time of the second peak is 16% larger than that of the first peak. It is interesting to note that the used nodes have 24 GB on each NUMA node; theoretically Graph500 kernels should always allocate their memory on the closest NUMA device. However, it does not happen (by default at least, since like most end users we make no special attempt to force near memory allocation using, say, numactl), probably due to a previously allocated I/O cache. In shared mode, each of these two peaks widen, and they merge together. Graph500 also exhibits a strong dependency on the sharing metric (p-value << 0.001). However, it has a higher variance, due to differences in memory distribution between the NUMA nodes. The Graph500 results demonstrate that it is important to pay attention to NUMA even for single-core jobs with modest memory consumption, in order to achieve the highest possible performance.

The file-system benchmarking application kernel, IOR, has a nearly identical distribution for the exclusive and shared modes; see Figure 3 C. IOR mainly performs file system read and write operations. Since the file system is the slowest component of the system, IOR should be not affected much by sharing of other resources. It is likely due to this sparse I/O pattern and very small

Table 2. Wall time average and standard deviation of application kernels executed on single core.

App Kernel	Mean Wall Time, s			Wall Time Standard Deviation, s		
	ex*	sh+	sh>0.75#	ex*	sh+	sh>0.75#
GAMESS	2078	2112 (+1.7%)	2130 (+2.5%)	2.5	27.7 (x11.1)	23.3 (x9.3)
NWChem	533.0	546.8 (+2.6%)	559.5 (+5.0%)	3.2	18.4 (x5.8)	23.0 (x7.3)
NAMD	1988	2005 (+0.8%)	2019 (+1.6%)	3.7	15.9 (x4.3)	13.7 (x3.7)
Graph500	941	933 (-0.9%)	959 (+1.8%)	84.4	67.4 (x0.8)	67.0 (x0.8)
HPCC	1705	1726 (+1.2%)	1740 (+2.0%)	7.9	26.3 (x3.3)	25.6 (x3.2)
IOR	41	43 (+5.6%)	45 (+10.7%)	12.8	16.9 (x1.3)	18.4 (x1.4)

***- executed exclusively (ex); + – executed in shared mode (sh); # - processes based sharing metric is greater than or equal to 75% (sh > 0.75).**

Table 3. Wall time average and standard deviation of application kernels executed on six cores.

App Kernel	Mean Wall Time, s					Wall Time Standard Deviation, s				
	Same CPU			Any CPU		Same CPU			Any CPU	
	ex*	sh+	sh>0.75#	sh+	sh>0.75#	ex*	sh+	sh>0.75#	sh+	sh>0.75#
GAMESS	361	363 (+0.5%)	363 (+0.7%)	362 (+0.3%)	363 (+0.6%)	0.7	1.8 (x2.5)	1.5 (x2.1)	2.6 (x3.6)	2.1 (x3.0)
NWChem	97.8	98.9 (+1.1%)	98.9 (+1.1%)	98.6 (+0.8%)	98.7 (+0.9%)	0.5	1.0 (x2.2)	0.9 (x1.8)	1.6 (x3.3)	1.3 (x2.8)
NAMD	373	375 (+0.4%)	375 (+0.5%)	375 (+0.4%)	375 (+0.6%)	0.5	1.9 (x3.9)	1.3 (x2.6)	2.0 (x4.1)	1.6 (x3.2)
Graph500	481	485 (+0.9%)	487 (+1.3%)	482 (+0.3%)	487 (+1.3%)	11.4	9.9 (x0.9)	6.8 (x0.6)	13.0 (x1.1)	9.5 (x0.8)
HPCC	364	366 (+0.7%)	366 (+0.8%)	364 (+0.1%)	365 (+0.5%)	4.0	4.7 (x1.2)	4.8 (x1.2)	7.4 (x1.8)	6.3 (x1.6)
IOR	257	257 (+0.3%)	256 (-0.4%)	258 (+0.5%)	258 (+0.5%)	17.0	20.2 (x1.2)	15.7 (x0.9)	20.1 (x1.2)	18.5 (x1.1)
ENZO	6059	6060 (+0.0%)	6090 (+0.5%)	6050 (-0.2%)	6076 (+0.3%)	405.6	408.0 (x1.0)	411.9 (x1.0)	406.5 (x1.0)	420.1 (x1.0)
ENZO, first peak	5699	5726 (+0.5%)	5753 (+0.9%)	5722 (+0.4%)	5739 (+0.7%)	135.0	136.9 (x1.0)	152.2 (x1.1)	137.7 (x1.0)	157.4 (x1.2)
ENZO Second peak	6453	6497 (+0.7%)	6523 (+1.1%)	6490 (+0.6%)	6528 (+1.2%)	162.8	149.4 (x0.9)	143.0 (x0.9)	148.4 (x0.9)	142.9 (x0.9)

***- executed exclusively (ex); + – executed in shared mode (sh); # - processes based sharing metric is greater than or equal to 75% (sh > 0.75).**

wall time of the IOR (~40 s) kernel that we were not able to detect significant wall time differences.

5. SIX-CORE JOBS

We consider the difference between running the application kernels in exclusive mode versus running them on six cores within a node. First, we will consider only jobs for which processes were assigned to the same CPU. In this case, computationally intensive application kernels behave very similarly to single-core ones, but with even smaller differences between wall times for exclusive and shared jobs. Even for jobs with a user-process-sharing metric greater than 75%, the average difference between shared and exclusive jobs only reaches 1.3% for Graph500. This is not surprising, since for jobs run this way, the application kernels have fewer shared resources, and all CPU caches and the memory controllers are dedicated to these jobs. However, the memory is not completely isolated because we do not enforce a strict NUMA allocation policy; the processes of other jobs may still access memory attached to the CPU of the test job.

If we include jobs where the cores are distributed between the two CPUs, the average numbers remain similar. We note the surprising result that some of the shared jobs are faster than the fastest exclusive jobs. When the jobs' processes are divided between the two CPUs, one can get concurrent utilization of both memory controllers. We did not perform simulations with equal processes distributed between CPUs in exclusive mode, but the estimates show that the difference between such jobs and shared jobs is within one percent for NAMD and ENZO, and up to 6.5% for NWChem. This means that novice users, unaware of core binding, can occasionally enjoy the benefits of a more optimal process distribution by running jobs in shared mode.

Similar to the single-core results (with the exception of IOR) the wall time regression against the sharing metric shows a statistically significant slope but with a large variance. However, as expected, the slope is even smaller than in the case of single- core jobs due to the reduced sharing of memory resources.

Enzo, a cosmological simulation code, differs from the other application kernels in having two distinct peaks in the wall time distribution. The performed simulation utilizes an adaptive mesh refinement (AMR) method for solving partial differential equations. AMR combined with dynamic load balancing leads to small differences in the calculated time series. This manifests itself in two separate peaks. If treated separately, each peak behaves very

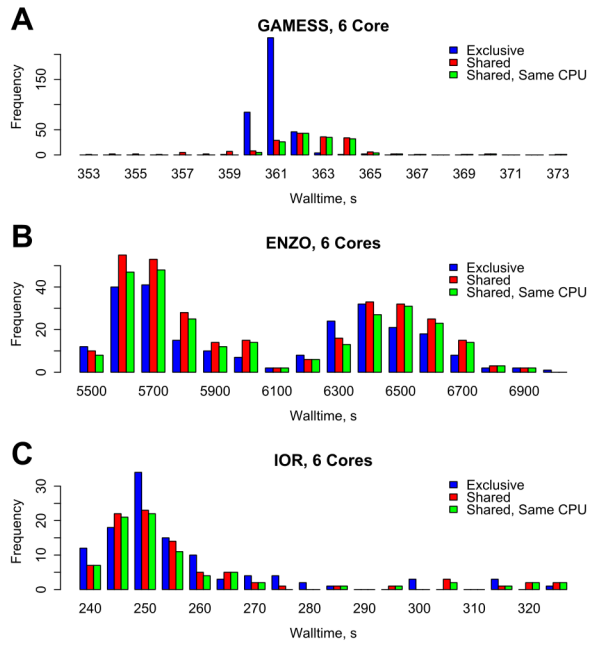

Figure 5. Wall time distribution of GAMESS (A), ENZO (B) and IOR (C) single-core jobs. The distributions of NWChem, NAMD, Graph500 and HPCC are qualitatively similar to that of GAMESS.

similar to NAMD and other computationally intensive application kernels.

For IOR, the file system benchmark, six-core jobs behave in a similar fashion to the single-core jobs in the sense that there is no statistically significant difference between exclusive and shared jobs.

6. MACHINE LEARNING MODEL

For further analysis of node sharing effects, we developed a machine learning model that employed data from the single-core jobs of the application kernels for NWChem, GAMESS, NAMD and HPCC. Single-core jobs were selected because they suffer the most from node sharing. IOR was excluded due to its file system specialization, and Graph500 was excluded due to strong dependence on NUMA. We only considered hardware performance metrics that characterize performance of shared resources, particularly last level caches (LLCs), main memory controllers, network interfaces and file systems. The data were all collected by PCP running on the compute nodes and processed using the SUPReMM software. The metrics included in the model are listed in

Many metrics are strongly correlated; to simplify the analysis we excluded these correlated metrics. For example, LLC misses correlate with LOCAL_DRAM and REMOTE_DRAM metrics, since reading from the dynamic random-access memory (DRAM) occurs if the data sought was not found in any cache. The LLC miss metric is available on a per-CPU basis, while LOCAL_DRAM and REMOTE_DRAM metrics are reported per core. The latter allows a more clear separation between our test job and other jobs executed on the same host. Because of this, we used LOCAL_DRAM and REMOTE_DRAM metrics in our analysis, and omitted other LLC metrics. By convention, the cores, CPUs and NUMA hosts were relabeled in such a way that index 0 always corresponds to the device on which the application kernel was executed. Therefore, core0 of CPU0 always corresponds to the core where the application kernel was running, cores 1-5 correspond to other cores on the same CPU (i.e. CPU0) and cores 6-11 correspond to CPU1.

Correlation matrixes for NAMD and the other application kernels for both exclusive and shared modes of execution were constructed. For the shared mode there is not much correlation observed between the metrics. The highest correlations for the shared mode are seen when various cores read from local DRAM. The local LLC and DRAM sharing of the same CPU cores results in a decrease in the LLC size available to each core. This leads to an increase in local DRAM usage, and thus a longer execution time. Interestingly, local DRAM utilization by cores on the other CPU also strongly correlates with the wall time of the test jobs. Some of the shared jobs run by other users were multi-core, so it is possible that several cores on our test job CPU and the other CPU on the same node are from the same job and performance of these cores are strongly correlated. Another possibility is that during a read from local DRAM, the CPU needs to check the possible presence of read data on the other CPU, which might affect the performance of the other CPU. Metrics for network and parallel file system show a very weak correlation with the other metrics, which is explained since the application kernels only use the parallel file system and the network briefly, for reading input and for standard output. The other application kernels in this analysis exhibit very similar behavior.

In order to gain further insight into the factors that influence the runtime difference between the shared and exclusive application kernels, we constructed a random forest (rF) machine learning regression model [17] using the attributes described in Table 4. The size of training and testing jobs were 738 and 187 jobs respectively. The root mean square deviation (RMSD) between observed and model calculated data were 0.005 and 0.011 for training and testing data respectively. This is a statistically significant improvement for the testing data over the null model, which assumes that the shared mode jobs run as fast as exclusive mode jobs. The RMSDs for the null model were 0.031 for both training and testing data. The paired t-test of the observed and model calculations has p-value of 0.84 indicating that the observed and model data cannot be statistically distinguished. Finally, Figure 6 shows the relative importance of the attributes in the rF model. First in importance is the application kernel type. This is not surprising, since each application kernel is characterized by different resource utilization patterns, and we have previously shown that such patterns can be used to identify applications [18]. For successful modeling, it is crucial to have different application kernel resource usage patterns. The three next most important variables are the local DRAM utilization by test job core and adjacent cores, followed by other CPU local DRAM utilization. As mentioned in the correlation analysis, this reflects the fact that the available CPU0 LLCs are reduced per job, which leads to higher DRAM utilization. The next most important variable is remote DRAM utilization by the test job core. With default non-strict NUMA policies, the system sometimes fails to allocate space on the local DRAM and needs to use remote DRAM. Operations with a remote DRAM are slower than with local DRAM, resulting in a longer run-time. MEM_NUMA0 measures the maximal memory utilization of test job local DRAM, it is very similar to LOCAL_DRAM _core* but less direct as it does not specify how often the data was used. In summary, contentions on LLC and DRAM are the most important contributors affecting shared job performance. The importance of variables in the model can help to determine the job mix leading to the most contention in a shared node system, as will be explored in the discussion below.

Table 4. Metrics utilized for machine learning model.

Metric Name	Description
App kernel	Application kernel name
wall time	Execution duration
LOCAL_DRAM _core0	Average rate of local DRAM load instruction retiring on core0. Local DRAM is the memory directly attached to the CPU.
LOCAL_DRAM _cores***i-j***	Average rate of local DRAM load instruction retiring on ***i*** to ***j*** cores
REMOTE_DRAM _core0	Average rate of remote DRAM load instruction retiring on core0. Remote DRAM is the memory attached to the other CPU on the compute node.
REMOTE_DRAM _cores***i-j***	Average rate of remote DRAM load instruction retiring on ***i*** to ***j*** cores
MEM_NUMA***i***	Maximal memory utilization on NUMA host ***i***
IB	Average combined receiving and transmitting bandwidth through the InfiniBand interface during application execution
ETH	Average combined receiving and transmitting bandwidth through Ethernet interface during application execution
GPFS	Average GPFS combined reads and writes bandwidth
HDD	Average local hard drive combined reads and writes bandwidth

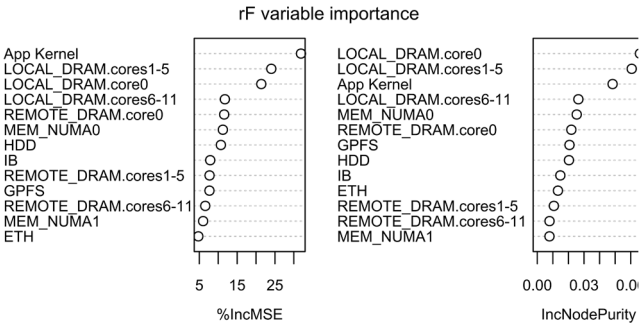

Figure 6. Variable importance in the rF model. The attributes are listed on the y-axis in descending order of model importance. The x-axis indicates quantitatively how crucial each attribute is in the model, that is how the model performance would be deteriorated if the attribute were omitted from the model. The left plot indicates how much the mean square error of the model would increase and the right plot is proportional to how much the residual sum of squares would change if the attribute was omitted from the model.

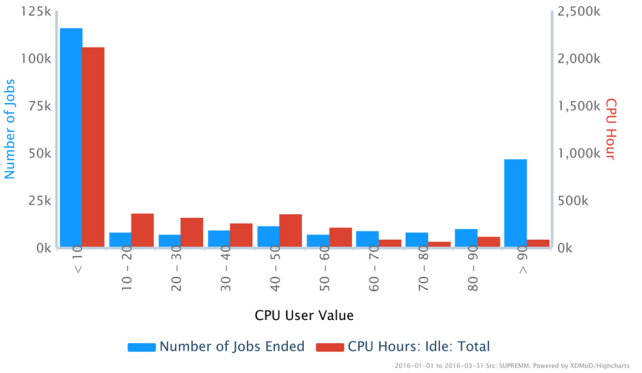

Figure 7. CPU utilization (CPU User Percent) of single-node jobs executed in exclusive mode for first three months of 2016 for all XSEDE resources with performance metrics support. Most of the single-node jobs run on XSEDE have a CPU User fraction of < 10 percent.

7. RELATED WORK

The advantages of node sharing were discussed in references [1–4] where quantitative advantages were given dependent on the job mix in the context of developing a fair pricing algorithm for shared node jobs. Some issues on node sharing that apply to disk cache and memory bandwidth have also been discussed [19, 20]. Our XSEDE14 paper [6] described a general statistical analysis of node sharing.

8. DISCUSSION/SUMMARY and FUTURE WORK

The present analysis has examined the consequences of jobs sharing nodes in an HPC environment. Overall, the data show little difference between shared and exclusive test jobs run on CCR's Rush production cluster, suggesting that job interference is not a significant issue. This is similar to the conclusion reached in our previous paper [6]. While this analysis was limited to CCR's production cluster, we expect that many academic HPC centers see a similar mix of jobs, and therefore that the job throughput benefits of node sharing will greatly outweigh the minor impact on execution times shown here.

The current study goes considerably further than our prior study by providing a quantitative analysis of the runtime effects of node sharing due to job contention. Correlation analysis and machine learning regression models enumerate the critical attributes that contribute to this contention. The main factor affecting the runtime of shared jobs is the reduced effective cache size available to jobs, which leads to their higher utilization of DRAM. Concurrencies on other shared resources, such as network and file systems, have a smaller influence. An analysis of the job mix studied indicates that this is likely to be representative of the general case. We also found that the proper usage of NUMA control should be beneficial for serial jobs with moderate memory requirements (that is, jobs that fit entirely on local DRAM with sufficient margin), because the system occasionally allocates memory on remote DRAM for such jobs.

We introduced a scheduler-based sharing metric that quantifies the amount of sharing that occurs for a given job. We demonstrated that this metric correlates with job runtime and that the correlation can be improved by using a metric that includes information about resource usage, in this case, core usage. Incorporating core usage improved the model, even though the individual job processes are constrained to different cores by cgroups. This raises the question as to how this model could be improved further. Since the machine learning model established that the memory subsystem contention was a major factor affecting the wall time, future models will likely include memory bandwidth.

Node sharing offers clear benefits for HPC centers in increased throughput and more efficient resource usage. There are benefits to the center user as well, since a slight increase in job runtime can be

compensated by a reduction in queue waiting time. However, the slight increase in runtime due to node sharing causes a very slight negative impact on user allocation. Our analysis and modeling is the first step to developing quantitative factors by which user allocations may be protected from sharing- related costs.

What is the potential impact of node sharing for XSEDE? The only XSEDE resource currently allowing node-sharing is Comet, but we propose that an expansion of node-sharing would be beneficial. To illustrate, Figure 7 7 shows CPU utilization as measured by CPU User fraction for single-node jobs on XSEDE. While the plotted data represents all XSEDE resources, it is not surprisingly comprised primarily of Stampede jobs. The figure suggests that most single-node jobs run on XSEDE have a CPU User fraction of less than ten percent, indicating that only a fraction of the available cores are being used. The practical implication for XSEDE is that enabling node sharing in the form of a serial queue would improve the throughput and resource utilization, shorten the wait time for single-node jobs, and potentially improve the overall throughput.

We are now working to implement an automated underperforming job discovery tool as a part of the XDMoD project [16]. Inefficient jobs such as those discussed with high system process activities, and processes escaped from cgroups, can be flagged by this tool, so that user support can be alerted about performance concerns. Furthermore, using this tool, user support personnel can study the previous performance of users' jobs and where appropriate recommend running in shared mode. Such approaches would improve HPC resource utilization, shortening the queues for some users, and making more efficient use of the available resources. In our future work, we will attempt to quantify this effect using our data and queue prediction tools.

ACKNOWLEDGMENT

This work was supported by the National Science Foundation under awards OCI 1025159, 1203560, and is currently supported by award ACI 1445806 for the XD metrics service for high performance computing systems.

REFERENCES

1. Iancu C, Hofmeyr S, Blagojevic F (2010) Oversubscription on multicore processors. 2010 IEEE Int. Symp. Parallel Distrib. Process. IEEE, pp 1–11

2. Breslow AD, Tiwari A, Schulz M, Carrington L, Tang L, Mars J (2013) Enabling fair pricing on HPC systems with node sharing. Proc. Int. Conf. High Perform. Comput. Networking, Storage Anal. - SC '13. ACM Press, New York, New York, USA, pp 1–12

3. Koop MJ, Luo M, Panda DK (2009) Reducing network contention with mixed workloads on modern multicore, clusters. 2009 IEEE Int. Conf. Clust. Comput. Work. IEEE, pp 1–10

4. Breslow AD, Porter L, Tiwari A, Laurenzano M, Carrington L, Tullsen DM, Snavely AE (2016) The case for colocation of high performance computing workloads. Concurr Comput Pract Exp 28:232–251. doi: 10.1002/cpe.3187

5. STREAM Benchmark Results on Intel Xeon and Xeon Phi | Karl Rupp. https://www.karlrupp.net/2015/02/stream-benchmark-results-on-intel-xeon-and-xeon-phi/. Accessed 22 Apr 2016

6. White JP, Barth WL, Hammond J, DeLeon RL, Furlani TR, Gallo SM, Jones MD, Ghadersohi A, Cornelius CD, Patra AK, Browne JC (2014) An Analysis of Node Sharing on HPC Clusters using XDMoD/TACC_Stats. Proc. 2014 Annu. Conf. Extrem. Sci. Eng. Discov. Environ. - XSEDE '14. ACM Press, New York, New York, USA, pp 1–8

7. Simakov NA, White JP, DeLeon RL, Ghadersohi A, Furlani TR, Jones MD, Gallo SM, Patra AK (2015) Application kernels: HPC resources performance monitoring and variance analysis. Concurr Comput Pract Exp 27:5238–5260. doi: 10.1002/cpe.3564

8. Valiev M, Bylaska EJ, Govind N, Kowalski K, Straatsma TP, Van Dam HJJ, Wang D, Nieplocha J, Apra E, Windus TL, de Jong WA (2010) NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. Comput Phys Commun 181:1477–1489. doi: 10.1016/j.cpc.2010.04.018

9. Schmidt MW, Baldridge KK, Boatz JA, Elbert ST, Gordon MS, Jensen JH, Koseki S, Matsunaga N, Nguyen KA, Su S, Windus TL, Dupuis M, Montgomery JA (1993) General atomic and molecular electronic structure system. J Comput Chem 14:1347–1363. doi: 10.1002/jcc.540141112

10. Phillips JC, Braun R, Wang W, Gumbart J, Tajkhorshid E, Villa E, Chipot C, Skeel RD, Kalé L, Schulten K (2005) Scalable molecular dynamics with NAMD. J Comput Chem 26:1781–802. doi: 10.1002/jcc.20289

11. Norman ML, Bryan GL, Harkness R, Bordner J, Reynolds D, O'Shea B, Wagner R (2007) Simulating Cosmological Evolution with Enzo. eprint arXiv:0705.1556

12. Graph 500. http://www.graph500.org/. Accessed 11 Aug 2014

13. Luszczek PR, Bailey DH, Dongarra JJ, Kepner J, Lucas RF, Rabenseifner R, Takahashi D (2006) The HPC Challenge (HPCC) benchmark suite. Proc. 2006 ACM/IEEE Conf. Supercomput. p 213

14. IOR HPC Benchmark | Free System Administration software downloads at SourceForge.net. http://sourceforge.net/projects/ior-sio/. Accessed 11 Aug 2014

15. Performance Co-Pilot, System Performance and Analysis Framework. http://pcp.io/. Accessed 4 Apr 2016

16. Palmer JT, Gallo SM, Furlani TR, Jones MD, DeLeon RL, White JP, Simakov N, Patra AK, Sperhac J, Yearke T, Rathsam R, Innus M, Cornelius CD, Browne JC, Barth WL, Evans RT (2015) Open XDMoD: A Tool for the Comprehensive Management of High-Performance Computing Resources. Comput Sci Eng 17:52–62. doi: 10.1109/MCSE.2015.68

17. Liaw A, Wiener M (2002) Classification and Regression by randomForest. R News 2:18–22.

18. Gallo SM, White JP, DeLeon RL, Furlani TR, Ngo H, Patra AK, Jones MD, Palmer JT, Simakov N, Sperhac JM, Innus M, Yearke T, Rathsam R (2015) Analysis of XDMoD/SUPReMM Data Using Machine Learning Techniques. 2015 IEEE Int. Conf. Clust. Comput. IEEE, pp 642–649

19. Eklov D, Nikoleris N, Black-Schaffer D, Hagersten E (2011) Cache Pirating: Measuring the Curse of the Shared Cache. 2011 Int. Conf. Parallel Process. IEEE, pp 165–175

20. Eklov D, Nikoleris N, Black-Schaffer D, Hagersten E (2012) Bandwidth bandit: Understanding memory contention. 2012 IEEE Int. Symp. Perform. Anal. Syst. Softw. IEEE, pp 116–117