



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

System Noise Revisited: Enabling Application Scalability and Reproducibility with Simultaneous Multithreading

E. A. Leon, I. Karlin, A. T. Moody

April 17, 2015

International Parallel and Distributed Processing Symposium
Chicago, IL, United States
May 23, 2016 through May 27, 2016

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

System Noise Revisited: Enabling Application Scalability and Reproducibility with SMT

Edgar A. León, Ian Karlin, and Adam T. Moody

Livermore Computing

Lawrence Livermore National Laboratory

Livermore, CA, United States of America

Email: {leon,karlin1,moody20}@llnl.gov

Abstract—Despite significant advances in reducing system noise, the scalability and performance of scientific applications running on production commodity clusters today continue to suffer from the effects of noise. Unlike custom and expensive leadership systems, the Linux ecosystem provides a rich set of services that application developers utilize to increase productivity and to ease porting. The cost is the overhead that these services impose on a running application, negatively impacting its scalability and performance reproducibility. In this work, we propose and evaluate a simple yet effective way to isolate an application from system processes by leveraging Simultaneous Multi-Threading (SMT), a pervasive architectural feature on current systems. Our method requires no changes to the operating system or to the application. We quantify its effectiveness on a diverse set of scientific applications of interest to the U.S. Department of Energy showing performance improvements of up to 2.4 times at 16,384 tasks for a high-order finite elements shock hydrodynamics application. Finally, we provide guidance to system and application developers on how to best leverage SMT under different application characteristics and scales.

Keywords-system noise; jitter; simultaneous multithreading; SMT; scalability; reproducibility; parallel performance;

I. INTRODUCTION

A salient cause of performance degradation of many parallel scientific applications at scale is system processes (e.g. I/O daemons) interfering with application threads on compute nodes. Because the application has no control over when and where these system processes run, the interference appears as *noise* to the application. We refer to system noise as any process (hardware or software) that delays an application’s execution and is not directly controlled by the application. Although noise has been studied for over two decades [1], more recent work [2], [3] identified noise as a key limiter to the scalability of high-performance computing (HPC) applications.

Significant research and development since then have virtually eliminated noise on leadership capability systems such as *Sequoia*¹ at Lawrence Livermore National Laboratory (LLNL). These gains, however, come with the high cost associated with the research, development, and maintenance

of a radically simplified operating system (OS) and specialized hardware. Commodity systems, on the other hand, are composed at a fraction of the cost by using hardware and software designed for mass consumer markets. Furthermore, an open-source, community-developed solution like Linux is used, which provides a rich set of services that application developers utilize. Many day-to-day science applications run on commodity systems. These systems, however, suffer from system noise affecting the scalability and performance reproducibility of applications.

Noise originates from the OS, parallel file system, resource manager, cache and network contention, etc. It already consumes a significant amount of execution time on today’s commodity systems, and it can reach critical proportions in next-generation clusters if left unchecked.

Rather than trying to eliminate noise, our work moves it off the critical path. Our approach leverages simultaneous multithreading (SMT), an architectural feature in the processor that provides multiple hardware threads of execution per core, to *absorb* noise and *isolate* an application from system interference. Unlike core specialization, where a core (or set of cores) is dedicated for system processing, our approach allows an application to use all of the cores of a node. As we show in this paper, this approach dramatically improves synchronous operations and, ultimately, application performance. Furthermore, these improvements are *free* from an application’s perspective since, in many instances, using the additional hardware threads for application work would result in lower performance. In addition, our approach does not require changes to the OS or the application. In previous work [4] we observed that SMT can reduce noise and provide further improvements over core specialization using a well-known micro-benchmark for noise (FWQ).

The contributions of this work are as follows. First, we identify the processes that are major contributors to noise in a current large-scale commodity cluster. As the Linux ecosystem changes over time, this characterization should inform other HPC centers of the impact of these processes in recent system software. Second, we demonstrate the effectiveness of our approach in reducing noise quantifying its impact on synchronization operations, which are extremely

¹Sequoia ranks third on the Top500 list as of November 2015.

sensitive to noise. Third, we show that for a suite of parallel scientific applications, representative of U.S Department of Energy workloads, using SMT provides significant performance benefits at no cost to the application. Furthermore, our approach does not require changes to the OS or the application. And, fourth, we analyze the tradeoff of using SMT threads for application work versus system processing and relate these results to application characteristics. The goal of this analysis is to provide guidance to application developers on how to best leverage SMT resources.

The paper is organized into the following areas. We start by identifying and characterizing sources of noise on a modern, large-scale Linux cluster (Sections II and III). Then, we describe our SMT approach (Sections IV and V) and apply it to synchronous operations (Section VI), known to be sensitive to noise. Next, we analyze the impact of our approach on a suite of HPC applications (Sections VII and VIII), which, ultimately, determine the end-benefits of this technique. Related work and conclusions complete the paper (Sections IX and X).

II. EXECUTION ENVIRONMENT

All experiments in this paper were run on the *cab* machine at LLNL. The *cab* system is a 1,296 node Linux cluster with two Intel SandyBridge (Xeon E5-2670) processors and 32 GB of memory per node. Each processor has eight cores with two hardware threads per core (Hyper-Threading). The nodes are connected via an InfiniBand QDR (QLogic), single-rail network. It runs the Tri-Lab Operating System Software (TOSS). At the time the experiments were executed, *cab* was running TOSS version 2.2, which is based on Red Hat Enterprise Linux Server release 6.5. Each processor has a theoretical peak memory bandwidth of 51.2 GB/s and uses DDR3 memory at 1.6 GHz. The resource manager in use is SLURM [5], [6] version 2.3.3.

III. SYSTEM NOISE

In this section, we identify and quantify the sources of system noise. Our goal is to assess the amount of noise that different system processes contribute at scale, where noise is most significant. The most direct experiment would be to execute a large-scale application that is susceptible to noise and then disable processes one-by-one to measure the effect each has on application performance. However, there are *many* processes on a typical compute node—we counted 735 different system processes—which makes this approach intractable.

Instead, we decided to filter the set of system processes using single node tests. Picking a compute node that had been running for several days, we sorted the system processes by the amount of CPU time each had accumulated—our assumption being that the noisiest processes are those that use the most CPU time. Then, using a single-node noise benchmark, we killed processes until we reached a state

where the noise signal was substantially quieter. We then re-enabled each process in isolation to assess its individual contribution to single-node noise. This way, we reduced the set of 735 processes to a handful of likely candidates worthy of large-scale testing. The experiments in this section were executed using *cab*’s default SMT configuration: one hardware thread per core.

A. Single-node Noise

For our single-node noise assessment, we used the Fixed Work Quantum (FWQ) benchmark². FWQ records a series of samples, where each sample records the time required to execute a fixed amount of work. In order to run one task per core we modified FWQ to run as an MPI job where each task is bound to a core. The tasks only communicate to synchronize their start time and to aggregate sample data at the end. On a noiseless system, each sample time should be identical. However, on a noisy system, some samples take longer than normal to complete, because the application process is interrupted to allow the system process to execute.

Figure 1 shows the FWQ results for several configurations. The horizontal axis is the sample number, while the vertical axis is the time taken for each sample in milliseconds. All cores recorded samples in parallel, and all are displayed individually on the graph. For this test, FWQ was configured to record 30,000 samples each with a nominal execution time of 6.8 milliseconds.

The far left graph in Figure 1 shows the noise signal obtained on the system before we disabled any system processes. On a noiseless system, we would expect each core to plot a solid horizontal line at 6.8 milliseconds. All sample points above this line indicate interference from system processes. The next subfigure to the right, shows the results obtained after we unmounted and unloaded Lustre; unmounted NFS; and disabled *slurmd*, *snmpd*, *cerebrod*, *crond*, and *irqbalance*. Although these processes account for the majority of the observed noise, there is at least one other process that we could not identify contributing noise. Regardless, we consider this state to be our “quiet” system. The last two plots, show the results obtained when re-enabling just *snmpd* or just Lustre on the quiet system. One can see that Lustre and *snmpd* each produce distinct patterns in the data.

B. Noise at Scale

After identifying the processes that produce notable noise on a single node, we investigated the impact of these processes at scale. Even if a process appears to be noisy on a single node, its effect on large-scale application performance may not be significant if the process executes synchronously across compute nodes [7]. Conversely, noise that is not synchronous amplifies with scale, so that even

²<https://asc.llnl.gov/sequoia/benchmarks/>

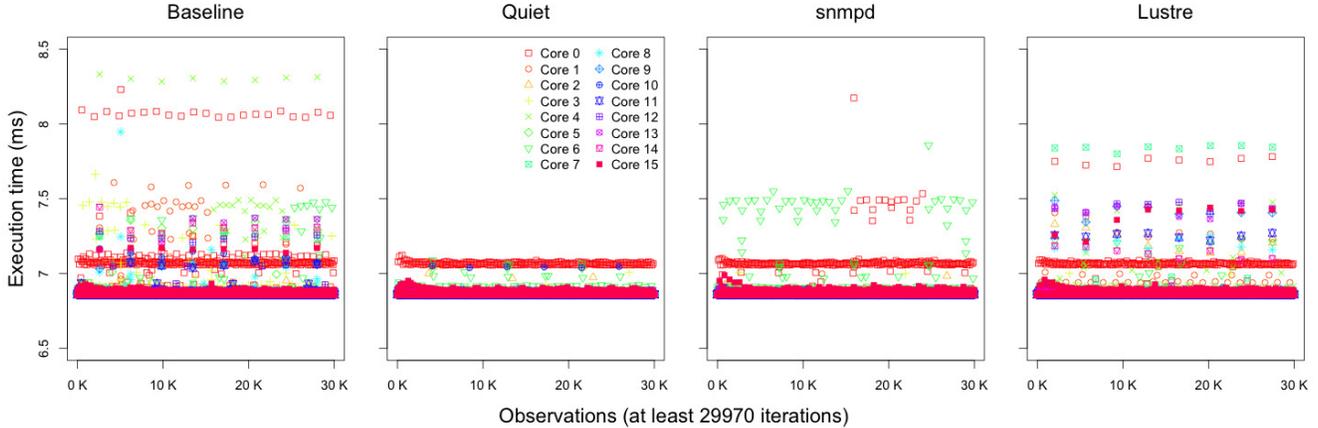


Figure 1. Measuring noise on a single-node using FWQ with different system configurations.

small perturbations on a single node can result in significant performance loss at scale.

For this test, we ran a benchmark that executes a series of back-to-back calls to `MPI_Barrier` measuring the number of processor cycles taken for each call. After completing all iterations, the benchmark outputs the cycle counts recorded by MPI rank zero.

We ran this benchmark using 16 processes per node (PPN) for one million iterations at various node counts. We used four different system configurations and computed the average time per operation and the standard deviation of all samples. The results are shown in Table I.

Table I
BARRIER STATISTICS FOR 1M OBSERVATIONS AND 16 PPN. LARGEST RUN CONSISTS OF 1024X16 MPI TASKS. TIMES IN μs .

Nodes		64	128	256	512	1024
Baseline	Avg	16.27	16.82	20.74	35.34	52.40
	Std	170.68	45.28	112.91	351.99	462.73
Quiet	Avg	13.28	16.09	18.43	22.57	28.27
	Std	15.78	19.68	26.58	37.57	61.13
Lustre	Avg	13.31	16.26	18.38	23.20	29.12
	Std	15.79	21.78	25.92	44.32	63.34
snmpd	Avg	13.44	16.39	21.73	25.17	38.67
	Std	18.10	24.24	223.53	145.76	246.93

Our results show that the quiet system performs and scales significantly better than the baseline system. At 1024 nodes, the average barrier cost is reduced by almost 50% while the standard deviation is reduced by nearly an order of magnitude. Also of note is the difference in scalability when enabling Lustre and when enabling `snmpd` on the quiet system. Lustre has a minimal impact on large-scale performance, while `snmpd` has a significant effect. This type of analysis can be used to determine which processes are most detrimental to large-scale performance.

IV. USING SMT TO REDUCE SYSTEM INTERFERENCE

Simultaneous multithreading (SMT) is a processor hardware feature that allows multiple threads to execute concurrently within a multiple-issue, dynamically-scheduled processor (core). This feature combines both thread-level parallelism (TLP) and instruction-level parallelism (ILP) by allowing multiple threads to issue independent instruction streams that can be executed concurrently on the processor's issue slots. Each thread has its own set of registers, but it *shares* most of the processor resources including execution units and caches.

In this work, we leverage SMT to reduce system interference by utilizing hardware thread contexts for system processing as opposed to application work. Since SMT commodity processors are not designed for HPC, using *all* hardware threads for a given application does not necessarily render the best performance. With increasing concurrency, an application can overwhelm a particular resource within the processor, at which point, adding additional threads does not help but instead may decrease performance. Similarly, for many memory-bound applications, adding additional threads puts more constraints on the already saturated memory system, which, again, can be detrimental to performance.

Note SMT can result in better application performance by, for example, hiding long latency loads or other pipeline stalls, but sometimes performance does not improve further when a shared resource is over-utilized. At the same time one resource is oversubscribed, other resources within the same core may be idle. In this case a more diverse instruction mix can still leverage the available resources.

Our approach strives to use additional thread contexts, enabled by SMT, for system processing. As we show in Section VIII, most of our HPC applications cannot leverage the additional hardware threads effectively, but those applications show great improvements by enabling the threads and leaving them idle to absorb system interference.

For all experiments in this work, we use Intel Xeon processors, which use Intel’s Hyper-Threading technology [8] to implement SMT. Hyper-Threading is an SMT-2 technology providing two hardware threads per core. We describe our SMT policies or configurations to mitigate noise in the next section.

V. SMT CONFIGURATIONS

On the cab machine, Hyper-Threading is enabled in the BIOS, but the secondary hardware threads are disabled at boot time. SLURM is configured to allow re-enabling of these threads if a user requests it for the duration of her job. If the user does not request the additional threads, her job is executed using a single hardware thread per core (16 CPUs per node). We refer to this default configuration as *Single-Thread (ST)*. When Hyper-Threading is explicitly enabled by a user (32 CPUs per node), we consider three additional configurations. First, the job consists of at most one *worker* (software thread or process) per core, i.e., it does not fully utilize the additional hardware threads. The idea is to leave these resources for the OS and other system processes. This configuration is referred to as *Hyper-Thread (HT)*. Second, the job utilizes as many workers as hardware threads. This configuration is called *HTcomp*, since the hyper-threads are used for application compute work. Last, we have *HTbind* that is similar to HT except that HTbind explicitly binds each application process or thread to a hardware thread. Table II summarizes all four configurations.

Table II
SMT CONFIGURATIONS.

<i>ST</i>	SMT-1	Don’t use more workers than cores
<i>HT</i>	SMT-2	Don’t use more workers than cores
<i>HTcomp</i>	SMT-2	Use as many workers as HW threads
<i>HTbind</i>	SMT-2	Like HT but bind workers to HW threads

The difference between HT and HTbind is simply in how user processes and threads are bound to the underlying hardware resources. HT uses the default process affinity provided by SLURM, which divides the number of cores by the number of processes and binds each process to the core subset. In this way, a process is bound to a core or a set of cores but threads within a process are not bound and may migrate among the resources assigned to the process. HTbind uses more strict affinity by binding each process to a single CPU for MPI-only applications and by binding each thread to a single CPU for MPI+OpenMP applications. The goal of HTbind is to avoid possible migrations by the OS. We should also mention that SLURM’s default affinity policy is used for the other two configurations: ST and HTcomp.

We employ the SMT configurations above to assess the impact on application performance when using the additional hardware threads provided by Hyper-Threading for system

processing. We start by quantifying the benefits of HT over ST on collective, synchronous operations. Later, in the context of the application analysis, we address an important question: should we use the hyper-threads for application work instead of system processing? As we show in Section VIII, this is application dependent, with some unable to take advantage of them, others always benefiting, and the rest showing a small performance increase from HTcomp that quickly diminishes as scale increases, at which point HT and HTbind provide substantial performance improvements.

VI. FASTER AND REPEATABLE COLLECTIVE OPERATIONS

In this section we demonstrate that the performance of globally synchronous collective operations can be dramatically improved at scale using SMT. We use Allreduce and Barrier operations although we focus more on the former since the impact is similar for both. As both Allreduce and Barrier are globally synchronous, if any process is slow to start an operation, all processes are delayed in completing that operation.

Our micro-benchmark is outlined below and consists of back-to-back Allreduce operations (sum of two doubles) for a large number of iterations (at least 500K). The cost of each operation in processor cycles is measured by MPI rank zero. On a noiseless system, each operation should take the same amount of time. On the other hand, if any process is delayed by noise before starting an operation, the time of that operation as measured by MPI rank zero is increased by the cost of the delay. We note that noise may delay a process during the middle of an operation, in which case, the cost of that delay will register in either the current or the next operation as measured by rank zero, but it will not impact both.

```

for(i=0; i<iters; i++)
  start = get_cycles()
  MPI_Allreduce(..., MPI_COMM_WORLD)
  stop = get_cycles()
  sample[i] = stop - start

```

Figure 2 plots the cost in cycles of each Allreduce operation. We employ 16 PPN over multiple node counts resulting in 256 to 16,384 MPI tasks. The ST configurations on the top clearly show that the execution time of each operation varies significantly from run to run due to noise. In fact, some operations were subject to extreme noise events and measured many orders of magnitude higher than normal, so for clarity, we do not show those that take more than 20 million cycles. Furthermore, as demonstrated in the related literature, as we increase the number of MPI processes from 64(nodes)x16(PPN) to 1024x16 the effect of noise increases dramatically.

Unlike ST, the secondary thread of execution in the HT configurations (bottom graphs) *absorbs* most of the noise resulting in repeatable and faster collectives. Although a few

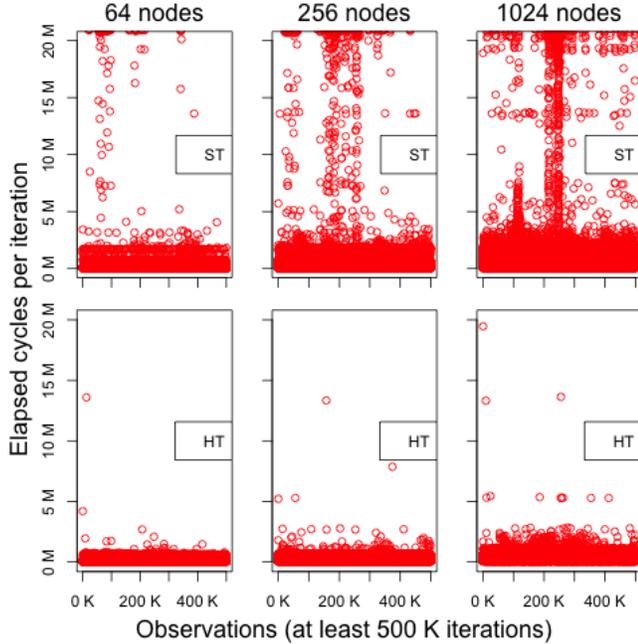


Figure 2. Improvements of HT (bottom) over ST (top) for Allreduce operations. In a noiseless system, a single horizontal band would be shown toward the bottom of the graph—and nothing else. For ease of visualization, we capped the Y-axis much lower than the max cycles incurred by ST runs.

data points still show some variability, there is a dramatic improvement over the default configuration.

Since Figure 2 does not capture all data points (y-axis capped), we turn to the histograms in Figure 3. We classify every Allreduce operation into bins according to their (logarithmic) elapsed cycles and for each bin we show the cost of its Allreduce operations relative to the total cost across all observations (500K), which are binned according to their elapsed processor cycles.

In an ideal system without noise, there would only be one bar occupying 100% of the cycles (y-axis) pegged at the first bin from the left (x-axis). As the top histograms show, as scale increases, the cycles spent on Allreduce operations without noise (shortest cycles) diminishes rapidly. In contrast, HT, even at a high number of processes (1024x16), spends most of the Allreduce cycles on operations that are close to the minimum time.

We also include the Barrier statistics in Table III collected using 16 PPN. The standard deviation (Std) shows the improvements with HT. For reference, we transferred the results for the “quiet” system from Section III-B, in which we disabled numerous system processes known to cause significant noise. Note that the average (Avg) HT case performs as well as the quiet system, but more importantly in the HT case, *all* of the noisy system processes are still

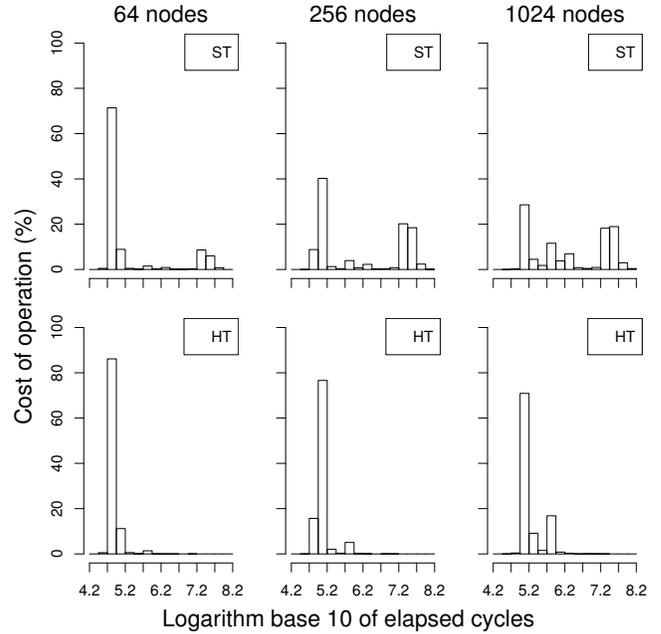


Figure 3. Improvements of HT (bottom) over ST (top) for Allreduce. Y-axis shows the cost of Allreduce operations relative to the total cost across all observations (500K), which are binned according to their elapsed processor cycles.

running. In fact, HT achieves a lower standard deviation than even the quiet system, presumably because it absorbs noise from additional processes that were still active on the quiet system.

Table III
BARRIER STATISTICS FOR 500K OBSERVATIONS AND 16 PPN. LARGEST RUN CONSISTS OF 1024x16 MPI TASKS. TIMES IN μs .

Nodes	16	64	256	1024	
ST	Min	4.80	5.66	6.78	5.78
	Avg	10.41	32.29	25.05	71.20
	Max	16,007.10	29,956.87	24,070.32	30,428.81
	Std	66.92	474.65	233.16	333.30
HT	Min	4.80	5.11	7.03	7.97
	Avg	9.89	13.38	18.82	28.28
	Max	921.92	5,220.44	2,458.86	7,871.85
	Std	3.09	10.23	15.76	35.22
Quiet	Avg	N/A	13.28	18.43	28.27
	Std	N/A	15.78	26.58	61.13

This analysis shows that synchronization operations clearly benefit from SMT by leveraging the additional hardware threads provided by HT to absorb system noise. An important remaining question is whether it is better for the application to use the additional SMT hardware threads for application work or whether it is better to leave those threads available for system processes. We address this question next.

VII. APPLICATION SUITE

We provide an empirical study of mitigating system noise with SMT on a representative suite of HPC applications. Our codes include four MPI production applications and four MPI+OpenMP mini-applications from the CORAL benchmark suite³, which represent U.S. Department of Energy (DOE) workloads and technical requirements used to procure three 100+ Petaflop/s computers. The applications represent diverse physics and application areas including hydrodynamics simulations, nuclear reactor criticality, and laser plasma interaction.

A. *miniFE*

miniFE is an approximation to an unstructured implicit finite element or finite volume application in a few thousand lines. miniFE’s main kernel assembles a sparse linear system from the steady-state conduction equation. The resulting system is solved using an un-preconditioned conjugate-gradient solver resulting in two main communication patterns of a 27-point halo exchange and MPI AllReduce operations [9].

B. *AMG2013*

AMG2013 is an algebraic multigrid benchmark application derived directly from the BoomerAMG solver in the HyPre linear solvers library [10]. The default Laplace-type problem is built from an unstructured grid with various jumps and anisotropy in one part. AMG2013’s dominant message patterns include Allreduce operations and small and medium point-to-point messages.

C. *LULESH*

LULESH is a Lagrangian explicit hydrodynamics application that solves the Sedov problem on a staggered grid mesh. It has numerical algorithms, data motion requirements, and a programming style that are similar to complex, production applications. LULESH is used for DOE co-design activities and as a machine procurement benchmark. On node LULESH is a mix of memory-bound and compute-bound kernels [11], while between nodes it requires three halo exchanges per timestep that are overlapped with computation. LULESH performs one optional AllReduce per timestep. Without this AllReduce, the code runs correctly, but requires more timesteps to complete a given amount of simulated time.

D. *BLAST*

BLAST is an explicit, arbitrary-order, finite-element-based hydrodynamics application. For our study we use a higher order problem and a partially assembled CG solve that is more compute intense than LULESH and miniFE and results in the entire code being compute bound [12]. Similar to LULESH, BLAST uses both halo exchanges and AllReduce operations as its primary communication patterns.

³<https://asc.llnl.gov/CORAL-benchmarks/>

E. *Ardra*

Ardra is a discrete ordinates (Sn) neutron transport code used for various engineering problems. For this paper, we used a reactor criticality eigenvalue problem. The main communication pattern in Ardra is small-message wavefront sweeps that occur concurrently from all corners (four in 2D and eight in 3D) of the mesh. A smaller portion of its messaging occurs in a multi-grid solver with similar properties to AMG [13].

F. *Mercury*

Mercury is a Monte Carlo particle (neutrons, gamma rays and charged particle) transport code. In our experiments we used a Godiva and Water problem, which looks at the criticality of a uranium mass in water. Mercury uses small and medium point-to-point messages to communicate particles to neighboring parts of the mesh. It also uses frequent AllReduce operations to test for completion of all particles [14].

G. *UMT*

UMT is a deterministic transport (Sn) mini-application. It solves 3D non-linear radiation transport calculations on an unstructured grid. UMT uses both threads and MPI to increase available parallelism and scalability. Its main communication patterns are large point-to-point messages to its nearest neighbors and medium size Allreduce operations [15].

H. *pF3D*

pF3D simulates laser-plasma interactions in National Ignition Facility (NIF) experiments. Our test problem is representative of production laser-plasma simulations, but with I/O turned off to increase problem turnaround. pF3D has three messaging patterns: 6-point halo, AllReduce, and 2D FFT. However, the large messages sent in the 2D FFT dominates message passing time [16].

VIII. APPLICATION SCALABILITY AND REPRODUCIBILITY

In this section we investigate how the SMT configurations described in Section V impact the performance and reproducibility of our applications. All of the results include at least five runs for each configuration. The experiment configurations are shown in Table IV.

Because of space constraints, we only focus on those experiments that demonstrate the most important trade-offs of using SMT and those that highlight the properties that make each application different from the others. In addition, because of the large experimental space (including eight codes, one or two inputs per code, four to six node counts, one or two PPN, four policies, and multiple runs per configuration) and the limited time on the cab production machine, we ran only a few data points for the HTbind

Table IV

EACH EXPERIMENT CONFIGURATION WAS EXECUTED ON MULTIPLE NODES RANGING FROM 8 TO 1024. TPP STANDS FOR OPENMP THREADS PER MPI PROCESS. ARDRA, BLAST, MERCURY, AND PF3D ARE MPI APPLICATIONS, ALL OTHERS ARE MPI+OPENMP. BECAUSE OF THE SIMILARITIES BETWEEN HT AND HTBIND FOR ARDRA, MERCURY, AND PF3D, WE RAN THE HT CONFIGURATION ONLY.

App	Size	PPN	TPP	SMT
miniFE	264x256x256 per node	2	8 16	ST,HT,HTbind HTcomp
		16	1 2	ST,HT,HTbind HTcomp
AMG2013	12x24x12 per process	2	8 16	ST,HT,HTbind HTcomp
		16	1 2	ST,HT,HTbind HTcomp
Ardra	200 per task	16 32	NA	ST,HT HTcomp
LULESH	108,000 per node	4	4 8	ST,HT,HTbind HTcomp
	864,000 per node		4 8	ST,HT,HTbind HTcomp
LULESH Fixed	108,000 per node	4	4 8	ST,HT,HTbind HTcomp
	864,000 per node		4 8	ST,HT,HTbind HTcomp
BLAST	147,456 per node	16 32	NA	ST,HT,HTbind HTcomp
	589,824 per node	16 32	NA	ST,HT,HTbind HTcomp
Mercury	15,000 per process	16 32	NA	ST,HT HTcomp
UMT	12x12x12 per process	16	1 2	ST,HT,HTbind HTcomp
pF3D	128x192x16 per process	16 32	NA	ST,HT HTcomp

policy for Ardra, Mercury, and pF3D. The reason being that HT and HTbind are similar for these applications as explained in Section VIII-B.

Based on our analysis as explained later in this section, we group applications into three categories that represent the correlations between application characteristics and their response to our SMT strategy. These groupings are: *memory bandwidth bound* applications, *compute-intense small message* applications with small messages and/or frequent synchronization operations, and *compute-intense large message* applications with few or insignificant synchronization operations.

We employ two types of plots. In the *scaling* plots we vary the number of nodes and MPI processes, and each data point represents the average of all the runs made for that experimental configuration. We use *box-and-whisker* plots (box plots) to represent the variation of performance between runs. In a box plot the main box represents the first

(bottom) and third (top) quartiles with the median drawn as a horizontal line inside the box. The vertical dashed lines are the whiskers and represent the minimum and maximum values excluding outliers, which are represented by single data points.

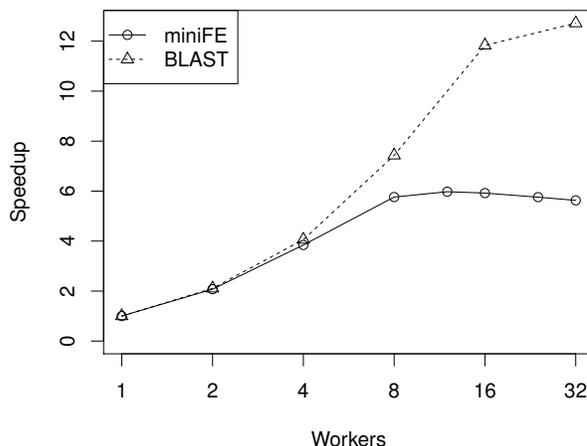


Figure 4. Single-node strong scaling of miniFE and BLAST.

A. Memory Bandwidth Bound Applications

AMG, miniFE, and Ardra are all memory bandwidth bound applications. Figure 4 shows the performance of miniFE when strong scaled on node. Typical of memory bandwidth bound applications, miniFE scales well for small core counts and then its performance is flat. The flattening occurs due to on-node memory bandwidth being saturated. Therefore, these applications never benefit from using hyper-threads for compute (HTcomp) and sometimes their performance degrades.

Table IV shows the problem sizes and configurations run for each application. For miniFE and AMG we used two configurations, 2 MPI processes per node (1 process per socket, 8 OpenMP threads) and 16 processes per node (1 process per core). For both codes we used the suggested problem sizes from the applications websites and ran weak scaling tests. For AMG, we used the default Laplace type problem. Ardra ran an eigenvalue reactor criticality problem using 16 and 32 MPI tasks per node.

As shown in Figure 5, the scaling of both configurations of miniFE and the 16 PPN version of AMG is similar. Ardra scales somewhat worse than the other two applications from 16 to 128 nodes. Our first observation is that using the hyper-threads for compute (HTcomp) decreases performance for all three applications. Also, when comparing HT and HTbind to ST, enabling the hyper-threads for system processing never hurts and sometimes helps. For all applications, we make this comparison to determine the costs and benefits of our SMT approach.

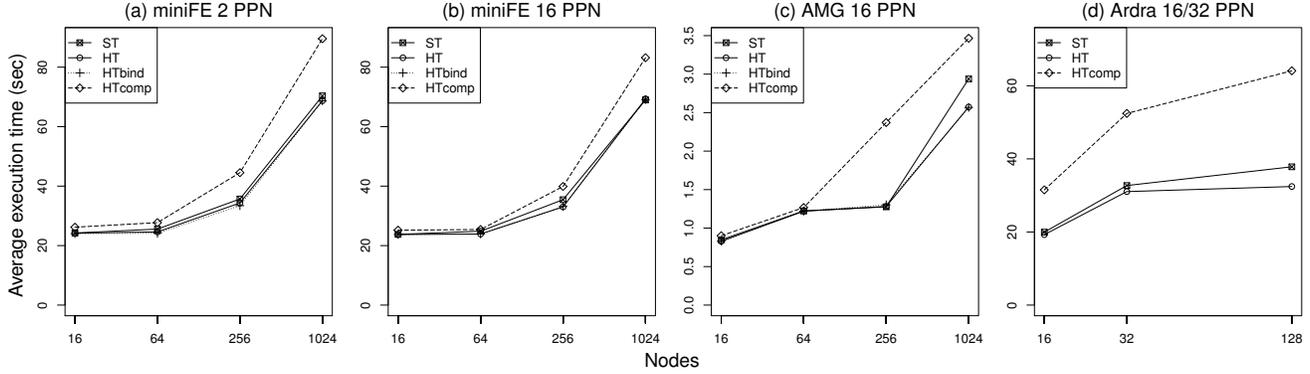


Figure 5. miniFE, AMG2013, and Ardra performance scaling. Additional hardware threads for compute decrease performance.

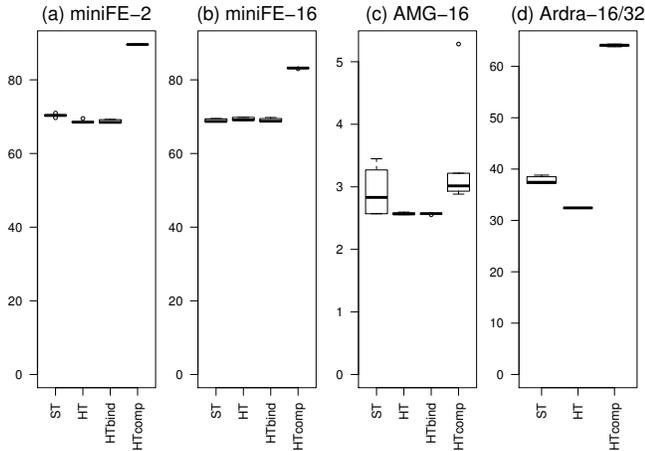


Figure 6. miniFE (2 and 16 PPN), AMG2013 (16 PPN), and Ardra (16/32 PPN) execution time (secs) variability across multiple runs. miniFE and AMG experiments executed on 1024 nodes and Ardra experiments executed on 128 nodes.

While for all three applications HT and HTbind improve performance at scale, the benefit is larger for AMG and Ardra. The timed sections of AMG and miniFE are iterative solvers that perform Allreduces each iteration. miniFE with its large problem size spends most of its time computing on node. In contrast, AMG, with a smaller problem size and an algorithm that results in multiple smaller grids, performs relatively more frequent Allreduces that consume more of its runtime. Therefore, AMG sees a larger gain from HT and HTbind than miniFE. Ardra, which sends the smallest messages of the three applications, shows an even larger performance gain from our HT policy. Its 15% runtime reduction at 128 nodes is the largest of all applications at that scale and larger than AMG and miniFE at 1024 nodes.

We also measured the performance variation between runs as shown in the box plots of Figure 6. Even at the largest scale of 1024 nodes, miniFE exhibits reproducible performance and does not seem to be affected by noise.

It is important to note, however, that miniFE scales poorly (see Figure 5). In weak-scaling mode, perfect scaling with no overhead for additional nodes would result in a flat horizontal line.

In contrast to miniFE, AMG and Ardra are affected by system noise as demonstrated by the taller box plots in Figure 6. The impact is different for the two applications. For AMG the fastest ST runs are as fast as HT, but the ST runs have significant run-to-run variation. In contrast, all the Ardra HT runs are faster than ST, but there is less run-to-run variation of the ST runs compared to AMG.

B. Compute-intense Small Message Applications

LULESH, BLAST, and Mercury, all send smaller messages of 10KB or less and when strong scaled in single-node experiments can take advantage of all computational resources including hyper-threads. Figure 4 shows single-node strong scaling for BLAST, which is typical of these applications. Performance improves almost linearly up to at least half the cores, and continues to improve, though slower, with additional resources and hyper-threads.

When scaled up these applications all have a cross-over point of how to achieve the best performance with SMT. Figure 7 shows application scalability. While the exact cross-over point varies from less than 16 nodes for LULESH and Mercury to between 16 and 64 nodes for BLAST the trend is the same. At small scale using hyper-threads for compute (HTcomp) results in the best runtime. Then, at larger scale using hyper-threads to mitigate noise (HT or HTbind) is best. In addition, the gains from HT or HTbind increase with scale.

Performance gains at scale with HT/HTbind vary from 20% for Mercury at 256 nodes to 2.4x for the smaller BLAST problem at 1024 nodes. For LULESH and BLAST, which we ran with two problem sizes, the smaller problem sizes showed a greater performance improvement. An example of this is shown in Figure 7, where BLAST's larger problem size had a 1.5x speedup. When using HT or HTbind the smaller LULESH problem size was 1.44x

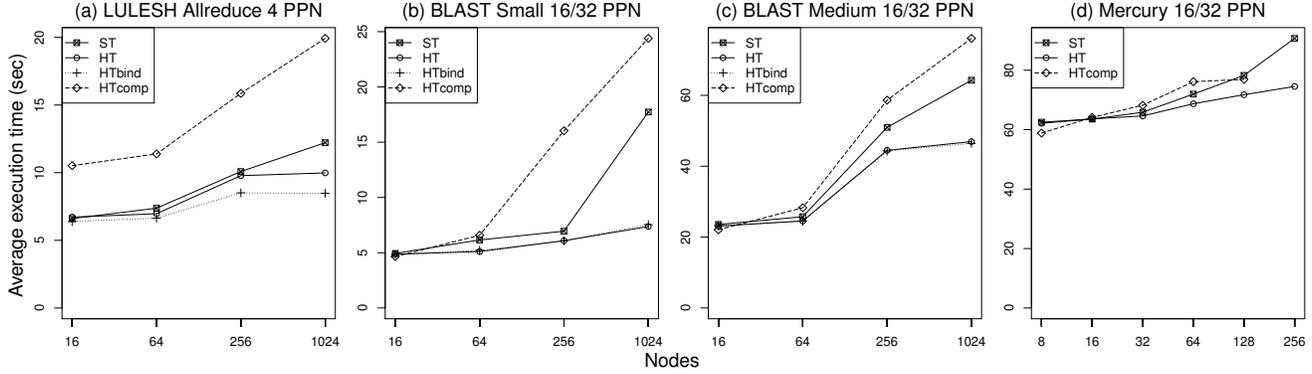


Figure 7. LULESH, BLAST, and Mercury performance scaling. HT and HTbind achieve the best performance at medium and large scale; HTcomp is best for a small number of nodes.

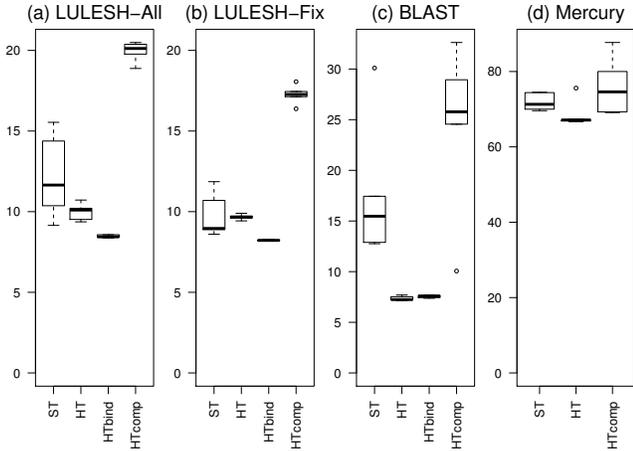


Figure 8. LULESH-Allreduce, LULESH-Fixed, BLAST (small problem size), and Mercury execution time (secs) variability across multiple runs. LULESH and BLAST experiments executed on 1024 nodes and Mercury experiments executed on 64 nodes.

faster compared to 1.07x with the large size. The larger speedups for the smaller problem sizes show that mitigating system noise, while important across problem sizes, is even more important when strong scaling. It is worth mentioning that our problem sizes are representative of complex multi-physics simulations where one component uses a fraction of the memory to leave space for other types of physics.

Figure 8 shows the run-to-run performance variability. For all applications HT improves runtime and performance variability. However, only for LULESH (2 versions, described later in this section), the only MPI+OpenMP code in this group, is HTBind better than HT. This is because with HT, SLURM binds each process (4 PPN) to 4 cores, allowing the 4 threads per process to migrate within these cores. HTbind, on the other hand, binds each thread to 1 hardware thread per core. For applications with 16 PPN there is little difference between HT and HTbind because the former allows a process to migrate only within two hardware

threads of a core.

In addition to our other tests, we ran LULESH using two code variants. The first was the default LULESH version. For the second, we modified the code to run using a fixed time step, which we refer to as *LULESH Fixed*. In the fixed variant, we remove the single Allreduce from the code without impacting correctness. Since Allreduce is sensitive to noise, we employ LULESH Fixed to analyze the impact that Allreduce has within an application. It is important to note that LULESH posts sends and receives as soon as possible to allow overlapping of communication and computation.

In Figures 8a-b, we show LULESH results for the largest runs, where noise is most pronounced. For the ST configuration, the fixed time step improves performance and reduces the impact of noise (smaller performance variability) relative to the Allreduce variant. These results show the performance degradation system noise can cause when an Allreduce operation is executed every two hundredths of a second. When hyper-threads (HT or HTBind) are used to mitigate noise LULESH with the Allreduce has similar performance as LULESH Fixed. Thus, by using our SMT strategy, algorithmic changes are not as important for scalability and performance. This detail is important, since modifying existing algorithms is complex and the Allreduce in LULESH allows larger timesteps to be taken resulting in faster time to solution. Finally, it is worth noting that even codes without globally synchronous communication, like LULESH Fixed, can benefit from our SMT approach to mitigate noise.

C. Compute-intense Large Message Applications

Similar to the applications in the previous section, UMT and pF3D have single-node profiles where performance increases with thread count and HTcomp. These gains were larger than the previous applications—pF3D with HTcomp increased performance by 20% on an 8-node job. In addition, UMT and pF3D differ from the previous two groups in their

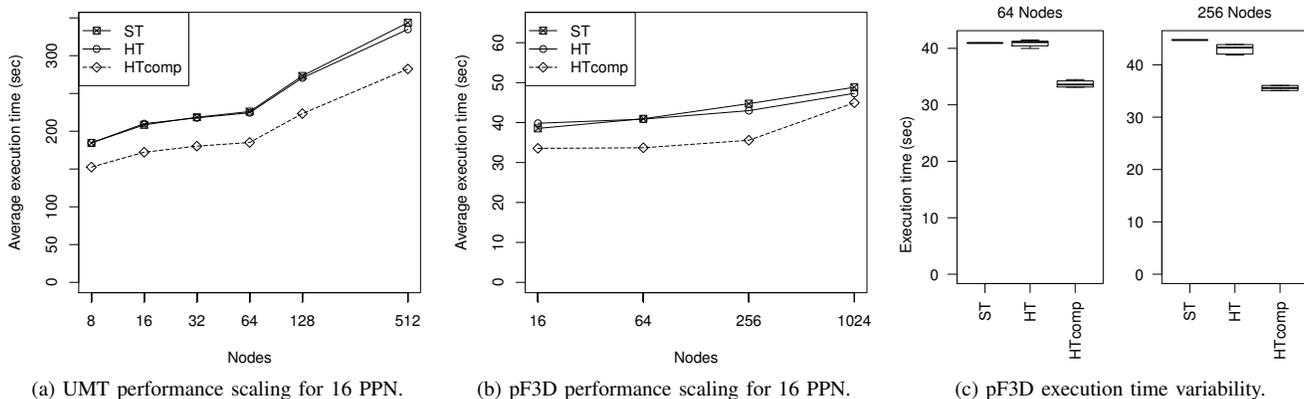


Figure 9. UMT and pF3D performance scaling and execution time variability.

messaging patterns. Both applications send large messages with the average point-to-point message in UMT larger than 150KB and its most frequent Allreduce operations being about 1KB or 5KB in size. Most of pF3D’s messages were all-to-all messages of sizes 12KB and 48KB on 64-task sub-communicators.

Figures 9a and 9b show the scaling performance of both applications. They show the same trend: HTcomp is best at all scales. For pF3D, however, the performance gap between HTcomp and HT starts to close at larger scale. For UMT, HT is slightly faster than ST, but pF3D shows no improvement from HT likely due to having only one collective operation per timestep. We expect at large enough scale there would be a cross-over point for UMT (similar to the previous group of applications), but we only had 1024 nodes to test on.

Figure 9c shows that pF3D is still impacted by noise at large scale and that HT does not reduce it. For this application, with minimal globally synchronous communication and large messages, HT does help performance relative to ST though it does not reduce noise. The source of noise has been documented in previous work [16]. UMT and pF3D, two compute intense applications with large messages, demonstrate that utilizing hyper-threads for system processing is beneficial relative to ST, but for these applications it is best to use hyper-threads as extra computation engines.

D. General Findings and Recommendations

Overall, enabling hyper-threads for system processing results in better performance and less noise across a wide range of scientific applications. This approach never reduced performance and we often observed significant gains, especially at scale. However, the impact of hyper-threads varies with application characteristics and scale.

For the memory-bound applications in this study (AMG, miniFE, and Ardra) turning on hyper-threads and not using them—from an application’s perspective—was always best. System processes can use the hyper-threads to avoid application interference. For these applications using hyper-

threads for extra compute resulted in worse performance than the single-threaded configuration. For the compute-intensive large message applications (UMT and pF3D) using hyper-threads for extra compute was best regardless of scale. For these applications turning on hyper-threads and not using them had a small, but noticeable positive effect over the single-threaded configuration. For the compute-intensive small message applications (LULESH, BLAST and Mercury), it was best to use hyper-threads for extra compute only at small scale, while applications at medium and large scales benefited the most by leaving the additional threads for mitigating noise.

Based on our results, we recommend that computer centers enable hyper-threads and bind application processes and threads, especially for large-scale jobs that are most susceptible to noise. However, care should be taken to educate users about the positives and negatives of the additional resources and how to use them effectively. For example, OpenMP using all available CPUs can result in worse performance with Hyper-Threading enabled than Hyper-Threading disabled. In this case, a user may need to specify an appropriate number of threads when Hyper-Threading is enabled.

IX. RELATED LITERATURE

Leadership supercomputing systems, such as LLNL’s *Sequoia*, are designed from the ground-up to impose minimal noise and to provide high scalability using techniques including a light-weight OS [17], [18] and hardware resource isolation between the operating system and applications. Isolation approaches include Cray’s core specialization [19] and the 17th core in IBM Blue Gene/Q [20]. While effective, these specially-built systems are quite expensive, and a majority of clusters are built from noisy but less costly commodity components. Commodity clusters are attractive because they are inexpensive, both in hardware and software, and they provide a full-featured OS (Linux) that is widely used and maintained by the technical community. Since the components of a commodity cluster were not designed for

high-performance computing, however, system noise limits the performance of applications at scale.

Many researchers have identified and characterized sources of noise and the associated impact on applications. These studies cover a wide range including analyzing noise through benchmarks, modeling, and simulation, and studying different sources of noise based on their frequency and duration characteristics [2], [21]–[27]. Fewer research studies, however, focus on mitigating noise [3], [7], [19], [28], [29]. Co-scheduling of system services has demonstrated significant improvements reducing noise although it requires a global clock and changes to the operating system. Other solutions are tailored to specific sources of noise resulting in one-off solutions that become obsolete when operating systems change.

The work by De et al. [28] is the closest to ours. The authors investigate the use of SMT to mitigate noise by changing Linux scheduling policies and priorities and isolating CPUs from kernel threads and interrupts. In their study, a micro-benchmark is used to evaluate the impact of their policies using simulation and a real cluster of eight nodes. The authors show substantial reductions of noise under the assumption that applications would run on a single thread per core. Our work is different in several ways: our methodology does not require changes to the OS or the application, we focus on real workloads from the U.S. Department of Energy, our experiments are done on real hardware at scale, and we analyze the tradeoff of using SMT for system processing versus application work. As we demonstrated, there is a class of applications that can leverage the additional hardware threads.

The work we present in this paper focuses on mitigating system noise on commodity clusters, which are systems that leverage the investments of a ubiquitous and continuously maintained OS. It relies on simultaneous multithreading, a pervasive architectural feature in modern processors, by leveraging SMT hardware threads for system processing. A key differentiator of this work with other noise mitigation studies is *simplicity*: our approach does not require changes to the OS or to applications. Moreover, we analyze the tradeoff of using hardware threads for system processing versus application work on real applications, on a real system at scale. Unlike core specialization where a core or a subset of cores is dedicated to the OS, our approach allows an application to use all the cores on a node.

X. SUMMARY AND CONCLUSIONS

Despite significant advances in reducing system noise, scientific applications running on production, commodity clusters today continue to suffer the effects of noise on scalability and performance. In this work, we leverage SMT to move system processing off the critical path and demonstrate prominent improvements on the performance of a diverse set of scientific applications. We also show substantially higher

scalability and performance reproducibility of synchronous operations. This study addresses the tradeoff of using the additional hardware resources provided by SMT for application work as opposed to system processing. In many cases, especially at scale, an application cannot take advantage of the additional hardware threads and, furthermore, trying to do so results in performance loss. The operations resulting from system processing, on the other hand, can co-exist with application work on the same core. In addition, the benefits of our approach require no changes to either the OS or the application. Finally, we correlate the impact of SMT to the characteristics of applications providing guidance to system and application developers on how to best leverage this feature.

Future work includes analyzing the influence of synchronization frequency, compute-to-communication ratio, and global versus neighborhood collectives on system noise.

ACKNOWLEDGMENTS

We would like to thank our LLNL colleagues Trent D’Hooge and Mark Grondona for their support with the TOSS system software and Steve Langer and Louis Howell for their help with pF3D and Mercury. We also thank the anonymous reviewers for their encouraging and detailed feedback. Prepared by LLNL under Contract DE-AC52-07NA27344. LLNL-CONF-669795.

REFERENCES

- [1] T. B. Tabe, J. P. Hardwick, and Q. F. Stout, “Statistical analysis of communication time on the IBM SP2,” *Computing Science and Statistics*, vol. 27, pp. 347–351, 1995.
- [2] F. Petrini, D. J. Kerbyson, and S. Pakin, “The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q,” in *Conference on Supercomputing*, ser. SC’03. Phoenix, AZ: ACM/IEEE, Nov. 2003.
- [3] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts, “Improving the scalability of parallel jobs by adding parallel awareness to the operating system,” in *Conference on Supercomputing*, ser. SC’03. Phoenix, AZ: ACM/IEEE, Nov. 2003.
- [4] E. Rosenthal, E. A. León, and A. T. Moody, “Mitigating system noise with simultaneous multi-threading,” in *International Conference for High Performance Computing, Networking, Storage and Analysis; Research Poster*, ser. SC’13. Denver, CO: ACM/IEEE, Nov. 2013.
- [5] A. B. Yoo, M. A. Jette, and M. Grondona, “SLURM: Simple Linux utility for resource management,” in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, vol. 2862.

- [6] Y. Georgiou and M. Hautreux, "Evaluating scalability and efficiency of the resource and job management system on large HPC clusters," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7698.
- [7] T. Jones, "Linux kernel co-scheduling for bulk synchronous parallel applications," in *International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS'11, Tucson, AZ, May 2011.
- [8] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, Feb. 2002.
- [9] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, Sep. 2009.
- [10] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Multigrid smoothers for ultraparallel computing," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, Oct. 2011.
- [11] I. Karlin, J. McGraw, J. Keasler, and B. Still, "Tuning the LULESH mini-app for current and future hardware," in *Nuclear Explosive Code Development Conference*, ser. NECDC'12, Livermore, CA, Oct. 2012.
- [12] S. Langer, I. Karlin, V. Dobrev, M. Stowell, and M. Kumbera, "Performance analysis and optimization for BLAST, a high order finite element hydro code," in *Nuclear Explosive Code Development Conference*, ser. NECDC'14, Los Alamos, NM, Oct. 2014.
- [13] U. Hannebutte and P. Brown, "Aradra: Scalable parallel code system to perform neutron-and radiation-transport calculations," Lawrence Livermore National Laboratory, Tech. Rep. UCRL-TB-132078, 1999.
- [14] P. S. Brantley, S. A. Dawson, M. S. McKinley, M. J. O'Brien, D. E. Stevens, B. R. Beck, E. D. Jurgenson, C. A. Ebberts, and J. M. Hall, "Recent advances in the Mercury Monte Carlo particle transport code," in *International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering*, ser. M&C'13, Sun Valley, ID, May 2013.
- [15] P. F. Nowak and M. K. Nemanic, "Radiation transport calculations on unstructured grids using a spatially decomposed and threaded algorithm," in *International Conference on Mathematics and Computation, Reactor Physics and Environmental Analysis in Nuclear Applications*, Madrid, Spain, Sep. 1999.
- [16] S. H. Langer, A. Bhatele, and C. H. Still, "pF3D simulations of laser-plasma interactions in National Ignition Facility experiments," *Computing in Science & Engineering*, vol. 16, no. 6, pp. 42–50, Nov 2014.
- [17] R. Riesen, R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira, "Designing and implementing lightweight kernels for capability computing," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 6, pp. 793–817, Apr. 2009.
- [18] M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski, "Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'10. New Orleans, LA: ACM/IEEE, Nov. 2010.
- [19] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella, "Leveraging the Cray Linux Environment core specialization feature to realize MPI asynchronous progress on Cray XE systems," in *Cray User Group*, ser. CUG'12, Stuttgart, Germany, Apr. 2012.
- [20] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, 2012.
- [21] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The influence of operating systems on the performance of collective operations at extreme scale," in *International Conference on Cluster Computing*, ser. Cluster'06. IEEE, Sep. 2006.
- [22] P. De, R. Kothari, and V. Mann, "Identifying sources of operating system jitter through fine-grained kernel instrumentation," in *International Conference on Cluster Computing*, ser. Cluster'07. Austin, TX: IEEE, Sep. 2007.
- [23] K. B. Ferreira, R. Brightwell, and P. G. Bridges, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'08. Austin, TX: IEEE/ACM, Nov. 2008.
- [24] K. B. Ferreira, P. G. Bridges, R. Brightwell, and K. T. Pedretti, "The impact of system design parameters on application noise sensitivity," in *International Conference on Cluster Computing*, ser. Cluster'10. Heraklion, Greece: IEEE, Sep. 2010.
- [25] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'10. New Orleans, LA: ACM/IEEE, Nov. 2010.
- [26] D. Doerfler, M. Epperson, J. Ogden, C. T. Vaughan, and M. Rajan, "Application performance on the Tri-Lab Linux capacity cluster-TLCC," *International Journal of Distributed Systems and Technologies*, vol. 1, no. 2, Apr. 2010.
- [27] S. Seelam, L. Fong, A. Tantawi, J. Lewars, J. Divirgilio, and K. Gildea, "Extreme scale computing: Modeling the impact of system noise in multicore clustered systems," in *International Symposium on Parallel & Distributed Processing*, ser. IPDPS'10. Atlanta, GA: IEEE, Apr. 2010.
- [28] P. De, V. Mann, and U. Mittal, "Handling OS jitter on multi-core multithreaded systems," in *International Symposium on Parallel & Distributed Processing*, ser. IPDPS'09. Rome, Italy: IEEE, May 2009.
- [29] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, "A quantitative analysis of OS noise," in *International Parallel & Distributed Processing Symposium*, ser. IPDPS'11. Anchorage, AK: IEEE, May 2011.