SAND2016-4873C

How I Learned to Stop Worrying and Love In Situ Analytics:

Leveraging latent synchronization in MPI collective algorithms

Scott Levy
Center for Computing Research
Sandia National Laboratories
sllevy@sandia.gov

Kurt B. Ferreira
Center for Computing Research
Sandia National Laboratories
kbferre@sandia.gov

Patrick Widener
Center for Computing Research
Sandia National Laboratories
pwidene@sandia.gov

Patrick G. Bridges
Department of Computer Science
University of New Mexico
bridges@cs.unm.edu

Oscar H. Mondragon
Department of Computer Science
University of New Mexico
omondrag@cs.unm.edu

ABSTRACT

Scientific workloads running on current extreme-scale systems routinely generate tremendous volumes of data for post-processing. This data movement has become a serious issue due to its energy cost and the fact that I/O bandwidths have not kept pace with data generation rates. *In situ* analytics is an increasingly popular alternative in which post-simulation processing is embedded into an application, running as part of the same MPI job. This can reduce data movement costs but introduces a new potential source of interference for the application. Using a validated simulation-based approach, we investigate how best to mitigate the interference from time-shared *in situ* tasks for a number of key extreme-scale workloads. This paper makes a number of contributions. First, we show that the independent scheduling of *in situ* analytics tasks can significantly degradation application performance, with slowdowns exceeding 1000%. Second, we demonstrate that the degree of synchronization found in many modern collective algorithms is sufficient to significantly reduce the overheads of this interference to less than 10% in most cases. Finally, we show that many applications already frequently invoke collective operations that use these synchronizing MPI algorithms. Therefore, the syncronization introduced by these MPI collective algorithms can be leveraged to efficiently schedule analytics tasks with minimal changes to existing applications. This paper provides critical analysis and guidance for MPI users and developers on the importance of scheduling *in situ* analytics tasks. It shows the degree of synchronization needed to mitigate the performance impacts of these time-shared coupled codes and demonstrates how that synchronization can be realized in an extreme-scale environment using modern collective algorithms.

EuroMPI 2016 Edinburgh, UK

© 2016 ACM. ISBN 123-4567-24-567/08/06...$15.00

DOI: 10.475/123_4

CCS Concepts

•**Software and its engineering → Scheduling; Input / output; Message passing;** *Massively parallel systems;*

Keywords

in situ analytics, message passing interface, collective algorithm synchronization, high-performance computing

1. INTRODUCTION

Next-generation extreme-scale computing systems are projected to be dramatically larger than current state-of-the-art capability-class machines. MPI-based simulation workloads running on these systems routinely write tremendous amounts of simulation output data per process to a parallel filesystem for later use. This has become a serious issue in large bulk-synchronous parallel applications. Because typical I/O bandwidths on modern platforms have not kept pace with data generation rates, significant contention forces longer waits for I/O to complete and in turn lengthens simulation run times. Additionally, tight energy budgets are raising concerns over the energy consumed by transporting large quantities of data over the shared interconnect. A system's energy budget and I/O bandwidth are typically fixed design quantities, which means that scaling data-intensive MPI applications requires a strategy for reducing data volumes.

As data produced for certain purposes (such as checkpoints) cannot be abbreviated without jeopardizing simulation correctness, developers of many leadership-class applications have begun to explore what can be done with the large amounts of output data intended for processing by post-simulation analysis tasks. *In situ* analytics is an increasingly popular technique in which such post-simulation processing is embedded into an application, running as part of the same MPI job and processing data as it is generated. This technique can have significant benefits, including the elimination of the movement of large raw data files both between nodes and between the application and the filesystem.

There are potential drawbacks, however. The addition of analytics tasks, which are periodically executed by the application, introduces a new potential source of interference between processes. This interference may combine with other scheduling factors to significantly impede performance at scale. Complicating this is the manner in which *in situ* analytics are incorporated into the application. A *space-sharing* approach isolates analytics processing from application processing by dedicating cores to it, while *time-sharing* causes each core to alternate between the main computation and analytics. Space-sharing attempts to fully isolates application performance from analytics at the cost of increased data movement between cores. It also can result in resource allocation imbalances where either the main computation or the analytics are over-provisioned in terms of CPU cycles. Time-sharing attempts to reduce data movement requirements by placing both tasks on the same CPU core, but does so at the cost of possibly interrupting the forward progress of the simulation. Time-sharing has the better potential for making efficient use of available resources, and for this reason it is the focus of this work.

Coordinating the execution of the computation and analytics is a straightforward approach to mitigating interference by globally isolating the two tasks' computation and communication from each other. Coordination can come in many forms, from synchronized hardware clocks to message-based collective operations. Each of these synchronization methods comes with costs and an associated synchronization resolution. However, the necessary degree of synchronization required from the application to keep impacts low has not yet been studied in detail. This guidance is necessary to calibrate developer approaches to the design of applications using *in situ* analytic processes, and can also inform feature selection for next-generation hardware.

In this paper, we show that *not* synchronizing analytics at all can have, as might be expected, a significant negative effect of performance. We also show that, perhaps counter to intuition, tight synchronization on the order of that typically provided by hardware facilities does not provide performance benefits commensurate with its potential cost. More specifically, we discuss how the performance of applications in which both simulation and *in situ* analytics tasks share on-node resources is affected by the degree of synchronization among their processes. Therefore, this paper makes the following contributions:

- We describe a simulation-based approach to the analysis of time-shared MPI applications using *in situ* analytics.
- We show that independent execution of analytics tasks in such applications can result in severe performance degradation, with slowdowns exceeding 1000% in some cases.
- We demonstrate that specific implementations of common MPI collective operations can induce non-trivial synchronziation between application processes.
- We demonstrate that relatively loose inter-process synchronization, on the order of that may be introduced by MPI collective algorithms for collectives that *are already being executed*, can provide significant performance benefits. In some cases reducing slowdowns to less than 10%.

The remainder of this paper is structured as follows: Section 2 provides general background on *in situ* analytics and Section 3 describes our approach to simulating the impact of these tasks on MPI applications. Section 4 then evaluates the impact of analytics on MPI applications and the viability of using the latent synchronization provided by collective communication algorithms to mitigate this impact. Finally, Section 5 describes related work and Section 6 concludes.

2. IN SITU ANALYTICS

Output from computational simulations is used for several purposes, including fault-tolerance, input to analysis and steering, and to build visualizations for a more complete understanding of results. A prevalent approach to analysis is for simulations to write their result data to networked parallel filesystems where they are then available for consumer processes and analytics to read directly. This arrangement has provided a separation of concerns between simulations and their associated analysis tasks. Also, new analysis tasks can be developed independently of the simulation, relying solely upon the semantics and structure of the simulation output data found on disk.

Parallel filesystem bandwidth, however, has not kept pace with the rates at which extreme-scale applications can generate information. Consequently, applications must either refrain from scaling to hardware limits, idle at large scales while complete result data is being written (an understandably unpopular choice), or find some way to write less data in order to maintain computation rates (which disrupts "downstream" analytics processing). The overwhelming trend has been for applications to preserve their scalability headroom while keeping average CPU utilization as high as possible. Attempts at addressing this problem have therefore emphasized reductions in the volume of data which must be stored.

Many analysis processes involve reductions from raw data, either to produce summary or sampled information or to convert into output formats more suitable for analytics running on desktop-scale systems instead of leadership-scale supercomputers. At the same time, CPU cycles have become a relatively cheap commodity on modern multi-core processors. Conversion to *in situ* analytic processes exploits both of these factors, providing reduced data output volume (in turn reducing I/O bandwidth contention) by allocating application-"owned" CPU cycles to analysis tasks.

Other approaches to locating data reduction or analysis processing have also been studied. Among these are so-called *in-transit* arrangements [4], where raw data is not handled by the same cores that produce it. Instead, data is relocated using an RDMA transfer to another node in the system which performs the analysis task and writes data to the filesystem. This avoids the bandwidth constraints and contention imposed by a parallel filesystem (typically an "analysis node" is shared as an RDMA target by a tunable number of compute nodes) at the cost of sacrificing nodes that would otherwise participate in the application.

2.1 Representative Workloads

A variety of analysis tasks, from different scientific disciplines and having differing computational characteristics, have been implemented using *in situ* processing within large-scale MPI-based application codes. We describe representative examples here.

Visualization and Steering.

For large-scale scientific simulations, *in situ* analytics codes have been used to prepare visualizations of results [38]. Without an *in situ* component, data must be transferred either to storage or to another machine so that it can be processed and rendered for display. The quality of visualized information available to scientists is then dependent on the volume of raw data that can be moved over the network. Because the volume of data that can be reasonably moved is a fraction of the total, scientists are forced to select a subset of timesteps or to eliminate features. Using *in situ* processing addresses this issue by extracting and storing only the visualization-relevant features for full-fidelity processing later, or by producing rendered images directly without incurring data transport costs. Steering the simulation (i.e. selecting between available code paths) can be accomplished by executing conditional statements based on the output of *in situ* processing [38].

Cosmology Analytics.

A common task in cosmology is to manipulate particle datasets produced by N-body simulations so that they may be treated as a continuous field for the purposes of analytics. Due to the wide variance in particle densities in these datasets, Voronoi tesselation [3] is used to produce unstructured meshes where mesh cell sizes correspond to particle spacing. These meshes can then be used to estimate particle density and other quantities for the purposes of isolating clusters and voids. Performing simulation-scale, parallel *in situ* tesselation allows longitudinal observation of statistical changes without incurring the I/O overhead of storing and retrieving particle and tesselation data [29].

PreDatA - Preparatory Data Analytics.

PreDatA [39] is a component-based middleware that allows user-defined custom processing to be inserted between applications and storage. Examples of this processing include data sorting, filtering, and histogram generation. Several large-scale applications have used PreDatAto implement *in situ* analytics. These include the Gyrokinetic Toroidal Code (GTC) [24], a computational-science application used for 3D particle-in-cell simulations of plasma micro-turbulence; and Pixie3D [6], a 3D MHD (Magneto Hydro-Dynamics) solver.

SmartPointer.

SmartPointer [37] is a program for preparation and visualization of data generated by molecular dynamics applications. It provides a flexible framework with configurable components. One of these components, *Bonds*, enhances the LAMMPS simulation code [31] with *in situ* crack detection and tracking capabilities. Specifically, Bonds uses atom bonding information from LAMMPS to conduct a compute-intensive analysis that determines the location of adjacent molecules in a simulated material which are no longer bonded (i.e. a crack has formed). This information can then be analyzed or visualized independently of LAMMPS.

2.2 Studying in-situ workload interference

The evaluation we describe in the remainder of this paper uses a workload derived from the execution of Bonds within SmartPointer. Several factors influenced this choice:

- The Bonds analysis component is freely available. Additionally, the LAMMPS application codebase with which Bonds works is also freely available and its behavior is well-understood thanks to its use in many research efforts. Results using this combination should be straightforwardly reproducible.
- The computational intensity of Bonds has been shown to be comparable to other analytics workloads (such as that generated in Goldrush [40] using PreDatA). We therefore believe Bonds to be a reasonable proxy for studying analytics workloads [27].
- Like the other representative workloads we have described in this section, Bonds does no communication of its own; it relies on communication performed by LAMMPS to obtain ghost cell information from other nodes in the simulation. A number of emerging analytics workloads do use a non-trivial amount of communication orthogonal to their respective main MPI application computation tasks, including in many cases MPI collectives. In this paper, we do not discuss such communicating analysis workloads, although our results do provide general guidance on how to mitigate their performance impacts as well and this remains a topic of interest for future work.

3. EXPERIMENTAL APPROACH

In this section we outline the experimental approach used to investigate the influence of synchronization in time-shared analytics tasks. First, we describe our validated simulation framework for examining analytics impact. This simulation framework allows a level of detail and functionality not possible with actual implementations on most current systems. For example, the simulator provides us with a global clock that can be used to enforce precise synchronization limits. In addition, we can determine precisely where slowdowns are accumulating, at per-message and per-collective levels. Second, we provide important guidance on interpreting our results in the context of MPI collective operation semantics. Finally, we conclude the section by detailing the important scientific workloads and production analytics tasks used in this evaluation.

3.1 Simulating the Performance Impact of In Situ Analytics

Our simulation-based approach models the impact of analytics on application performance by modeling the effect of the CPU cycles that are taken from the application to run the analytics code. This approach allows a level of fidelity and control not always possible in implementation-based approaches. It also allows us to examine application performance on systems that are much larger than those that are generally available for systems research.

Our simulation framework is based on `LogGOPSim` [20]. `LogGOPSim` uses the LogGOPS model, an extension of the well-known LogP model [7], to model the temporal cost of communication events. An application's communication events are generated from traces of application execution. These traces contain the sequence of MPI operations invoked by each application process. `LogGOPSim` uses these traces to reproduce all communication dependencies, including indirect dependencies between processes which do not communicate directly. For collective operations, the communication dependencies depend on the collective algorithm that is used. The collective algorithm that `LogGOPSim` uses is configurable. For example, the supported collective algorithms for simulating `MPI_Allreduce()` include: binomial tree, dissemination, linear, and `MPI_Reduce_scatter()`/`MPI_Allgather()`.

`LogGOPSim` can also extrapolate traces from small application runs; a trace collected by running the application with p processes can be extrapolated to simulate performance of the application running with $k \cdot p$ processes. The extrapolation produces exact communication patterns for MPI collective operations and approximates point-to-point communications [20]. The validation of `LogGOPSim` and its trace extrapolation features have been documented elsewhere [19, 20]. To simulate the impact of depriving the application of CPU cycles in order to run *in situ* analytics codes, `LogGOPSim` accepts an *execution trace*: an ordered list of an analytics code's execution, expressed as the start time and duration of each of its computation periods.

We use the Linux `ftrace` utility [33] to collect an execution trace of the Bonds function of SmartPointer analysis. `ftrace` allows us to measure the computational requirements of analytics running in-line with the application. Specifically, the `sched_switch` events provide a trace of CPU time slices used by the analytics code. A portion of Bonds' execution pattern is shown in Figure 1. These data show periodicity in the execution of Bonds; every few seconds it executes for tens of milliseconds, but is otherwise largely idle.

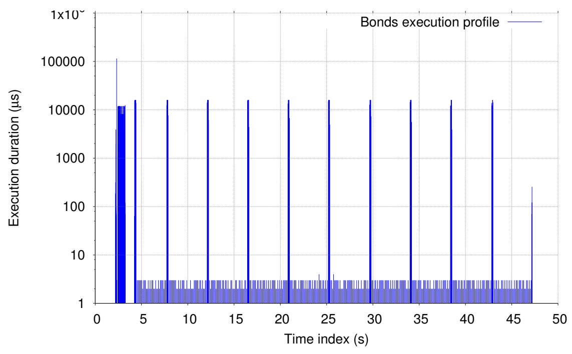


Figure 1: Execution pattern of the Bonds function of SmartPointer analysis. Each impulse represents the duration of a single execution interval.

`LogGOPSim` can simulate the degree of synchronization between analytics codes running in different application processes by adding an initial offset to the replay of the execution trace. Using an initial offset of zero for application processes will simulate analytics execution that is tightly synchronized across application processes. At the other extreme, choosing the initial offset for each simulated process uniformly at random will simulate uncoordinated execution of analytics. Choosing the starting point from a normal distribution will simulate different degrees of synchronization of analytics execution. The standard deviation of the distribution controls the degree of synchronization.

3.2 Collective Synchronization in MPI

In this paper, we consider how collective algorithms may impact the degree to which application processes will synchronize. However, we observe that these results need to be carefully interpreted. Guidance provided in the MPI Standard, version 3.0, cautions against the perils of relying on the synchronizing effect of collective operations [16, §5.1]. In accordance with the MPI standard, we do not rely on synchronization for correctness. Rather, we examine how particular *implementations* of common MPI collective operations may influence the synchronization of application processes. Therefore, while we refer in the text to the degree of synchronization of MPI collective operations, we are actually referring to the degree of synchronization of the modern point-to-point collective algorithms [17, 32, 35] used by current MPI libraries to perform these operations. The MPI specification itself makes no such synchronization guarantees.

3.3 Application Workload Details

In the remainder of the paper, we present results from simulation experiments based on the behavior of a set of five workloads. These workloads were chosen to be representative of scientific applications that are currently in use and computational kernels thought to be important for future extreme-scale computational science. They include:

- LAMMPS: A scientific application developed in Sandia National Laboratories to perform molecular dynamics simulations. We used the LAMMPS *2D crack* and *Lennard-Jones* potentials [31].
- CTH: A code developed at Sandia National Laboratories for modeling complex problems that are characterized by large deformations or strong shocks [9].
- HPCCG: A simple conjugate gradient solver from the Mantevo suite of mini-applications [18, 34].
- LULESH: An application that represents the behavior of a typical hydrocode [23].

CTH and LAMMPS are important U.S. DOE applications which run for long periods of time on production machines and exhibit a range of different communication structures. HPCCG represents an important computational pattern in key HPC applications. LULESH is an exascale application proxy from the DOE ExMatEx co-design center [11].

4. EXPERIMENTAL RESULTS

In this section, we discuss our experimental evaluation of the extent to which latent synchronization introduced by MPI communication algorithms can be used to ameliorate interference from computational *in situ* analytics tasks. To measure *inter-process synchronization*, we record the global simulated time when the first and last processes enter and exit the collective operation.[1] From these data, we determine how synchronized the application processes by computing the temporal distance between the fastest and slowest processes.

Subsection 4.1 quantifies the potential impact of *in situ* analytics tasks when they are scheduled at the two extremes of synchronization: totally unsynchronized and perfectly synchronized. These results show that uncoordinated execution has potentially catastrophic performance results, while globally synchronized scheduling of analytics, which is expensive to implement in practice, can eliminate virtually

[1]The exit order may differ from the entry order. For example, the first process to enter the collective may not be the first process to exit the collective.

Collective operation microbenchmark

```
interval_duration ∈ $\{50ms, 500ms, 5s, 50s\}$
collective_operation ∈ {27pt stencil, MPI_Allreduce(), MPI_Bcast(), MPI_Reduce() }

procedure COLLECTIVE_MICRO(interval_duration, collective_operation)
    for all intervals do
        execute collective_operation
        sleep interval_duration
    end for
end procedure
```

Algorithm 1: Pseudocode of collective operation microbenchmark.

all analytics-based interference. The subsections that follow describe our use of microbenchmarks and full-fledged MPI applications to quantify the degree to which different MPI collective algorithms induce inter-process synchronization. Finally, Subsection 4.5 evaluates the degree to which this latent synchronization may ameliorate the impact of analytics interference on MPI applications. These results show that leveraging even approximate synchronization can significantly reduce the performance interference of *in situ* analytics.

4.1 Examining the extremes in analytics scheduling

We begin by considering the performance impact of the two extremes of analytics scheduling: allowing analytics tasks to be independently scheduled within each application process vs. perfectly synchronizing the execution of analytics tasks across processes. The former is both simple to implement and minimizes added communication, while the latter relies on expensive hardware capabilities, for example, a global hardware clock. Each simulated process begins at an offset from the beginning of the Bonds execution trace (see Figure 1). To simulate independently-scheduled analytics tasks, the offset for each process is randomly generated; for perfectly-synchronized execution, the offsets of all processes are identical.

The results of these experiments are shown in Figure 2. For some applications (e.g., CTH-st, HPCCG, and LAMMPS-crack) totally unsynchronized scheduling of Bonds can have a disastrous effect on application performance. Perfectly synchronizing Bonds execution does eliminate this performance overhead, but in practice, achieving sufficiently tight synchronization on large-scale distributed systems is prohibitively expensive.

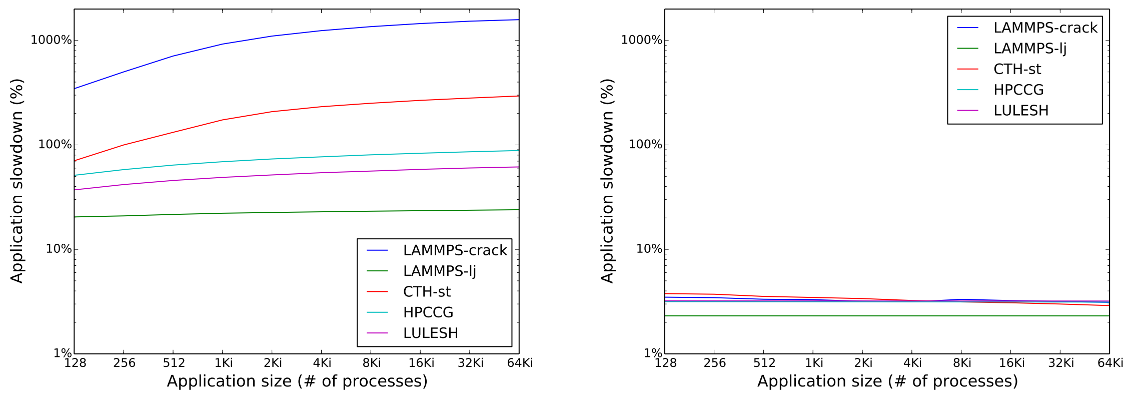
4.2 Communication Algorithm-induced Synchronization

We next evaluate the degree to which MPI collective communication algorithms effectively synchronize process execution. Our hypothesis is that this latent synchronization is sufficient to provide most of the benefits of perfectly-synchronized scheduling of analytics. To do so, we developed a set of microbenchmarks to determine how much inter-process synchronization might be introduced by the communication dependencies generated by MPI collective algorithms. The MPI standard describes the high-level semantics of each supported collective operation; it does not specify inter-process communication dependencies. As a result, MPI collective operations are not guaranteed to synchronize application execution (*see* Section 3.2). However, the structure of the communication pattern created by a particular implementation may influence inter-process synchronization. Our microbenchmarks allow us to examine the degree to which specific implementations of MPI collective operations cause application processes to synchronize.

We use these microbenchmarks to study the synchronization impact of four common collective communication motifs in MPI applications: (i) dissemination (e.g., a common motif for `MPI_Allreduce()`); (ii) binomial tree dispersal (e.g., a common motif for `MPI_Bcast()`); (iii) binomial tree aggregation (e.g., a common motif for `MPI_Reduce()`); and (iv) stencil communication. For the stencil communication experiments, the microbenchmarks use a three-dimensional 27-point stencil. In MPI, stencil communication can be implemented using a communicator with topology information attached, like `MPI_Cart_create()`, and a neighborhood collective like `MPI_Neighbor_alltoall()`. For the binomial tree experiments the microbenchmark uses the process with rank 0 as the root of the collective operation. To evaluate the impact of collective frequency, we conducted a set of experiments in which we considered the impact of four inter-collective periods: every 50 milliseconds, 500 milliseconds, 5 seconds and 50 seconds. Pseudocode for our microbenchmark is shown in Algorithm 1.

We collected a trace of the execution of each benchmark. We then ran each trace in `LogGOPSim` in conjunction with the execution trace collected from SmartPointer analysis running the Bonds function. We then measured the temporal distance between each simulated process, leveraging the global clock provided by the simulator.

The results of these experiments are shown in Figure 3. Although each microbenchmark runs for more than six minutes, we show only the first two minutes of execution here to highlight the fine-grained behavior of each. This figure shows that each of these communication motifs results in significantly different amounts of inter-process synchronization. `MPI_Bcast()` is only able to synchronize the processes to within a few seconds. The stencil communication introduces more inter-process dependencies and is able to keep the processes within a few hundred milliseconds of each other. `MPI_Allreduce()` creates more rigid dependencies and is able to tightly synchronize the processes. At the completion of an `MPI_Allreduce()`, the processes are synchronized to within a few tens of milliseconds, depending on the interval between



(a) Totally unsynchronized (b) Perfectly synchronized

Figure 2: Impact of the extremes (totally unsynchronized and perfectly synchronized) of the synchronization of analytics execution on application performance.

collectives. At the initiation of the next `MPI_Allreduce()`, the processes are not as tightly synchronized but are still within a few hundred milliseconds of each other.

4.3 Application-level Synchronization

We next quantify the degree to which MPI collective algorithms synchronize the execution of real applications. The results of these experiments are shown in Figures 4 and 5. Figure 4 shows the synchronization effect of the dissemination algorithm. As a whole, this figure confirms the data in Section 4.2. Specifically, it shows that MPI collectives that use the dissemination algorithm (e.g., `MPI_Allreduce()`) have a tendency to tightly synchronize application processes. Although there are outliers, the application's processes are within a few tens of milliseconds of each other following most calls to `MPI_Allreduce()`. This is true even when the application's processes are not well synchronized before the collective (*see e.g.,* Figure 4b (HPCCG) and Figure 4e (LULESH)). Figure 5 shows the same data for the binomial dispersal algorithm (e.g., an implementation of `MPI_Bcast()`). In contrast to the dissemination algorithm, the binomial dispersal algorithm has very little impact on process synchronization. The distributions of the temporal distances between the fastest and slowest process in all five applications are essentially the same before and after the execution of collectives that use this algorithm.

4.4 Application Inter-collective Times

We have shown that modern collective communication algorithms have the side effect of introducing a non-trivial amount of synchronization into the application's execution. To better understand the viability of using this synchronization to coordinate the execution of analytics tasks, we next investigate how frequently MPI applications invoke collective operations that use these algorithms.

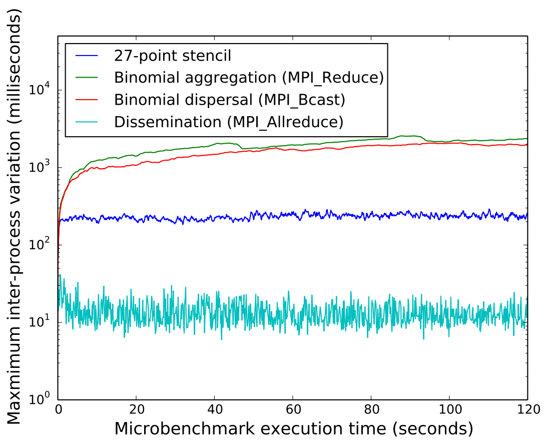
Figure 6 shows the discrete cumulative distribution functions (CDF) of the inter-collective times of MPI collective operations for each of our workloads. This CDF shows the distribution of the inter-collective times for each of five workloads. In this figure, a point at (x, y) indicates that, for a given application, at least $(x * 100)\%$ of the inter-collective times are smaller than y seconds. For example, Figure 6e demonstrates that 100% of the inter-collective times for `MPI_Allreduce()` in LULESH are less than 150 milliseconds.

The first thing we observe in these data is that `MPI_Allreduce()` is the most common collective operation for all five workloads. In fact, for LULESH and HPCCG, `MPI_Allreduce()` is the only collective operation used.[2] The next observation is that the rate of `MPI_Allreduce()` varies significantly between applications:

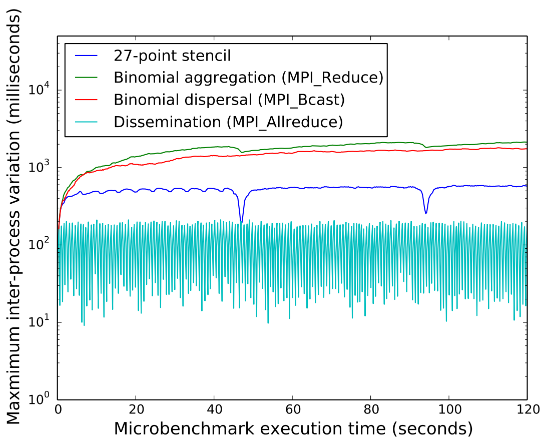
- in CTH-st, the inter-collective time for 80% of `MPI_Allreduce()` operations is less than 125 milliseconds;
- in HPCCG, the inter-collective times for `MPI_Allreduce()` are bimodal: approximately half are between 40 and 50 milliseconds, and approximately half are between 300 and 500 milliseconds;
- in LAMMPS-crack, 80% of the `MPI_Allreduce()` inter-collective times are between 9 and 10 milliseconds, but there is also a small number that are in excess of 150 milliseconds; and
- in LAMMPS-lj, half of the `MPI_Allreduce()` inter-collective times are between 10 and 100 microseconds, but more than 10% are in excess of 5 seconds.

Of the remaining collective operations, only `MPI_Bcast()` is called more than a handful of times during the execution of these five applications. CTH-st, LAMMPS-crack, and LAMMPS-lj call `MPI_Bcast()` frequently (i.e., the inter-collective times tend to be small). Collectively, these data demonstrate that collective operations that use the dissemination (e.g., `MPI_Allreduce()`) and binomial tree dispersal (e.g., `MPI_Bcast()`) are common in important MPI workloads and are done frequently enough in many applications to be utilized to schedule analytics tasks.

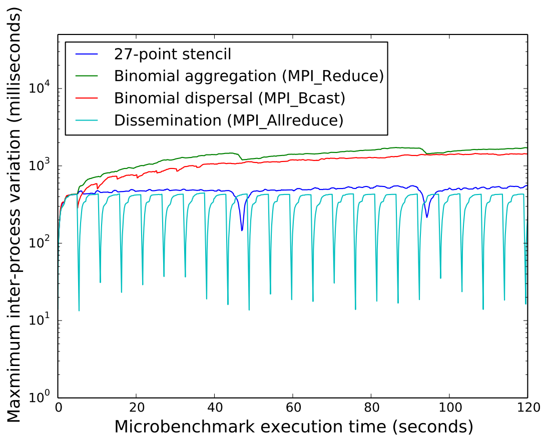
[2]While `MPI_Allreduce()` is the only collective operation that we observed for these applications in our experiments, the occurrence of MPI collective operations may depend on the inputs provided to the application.



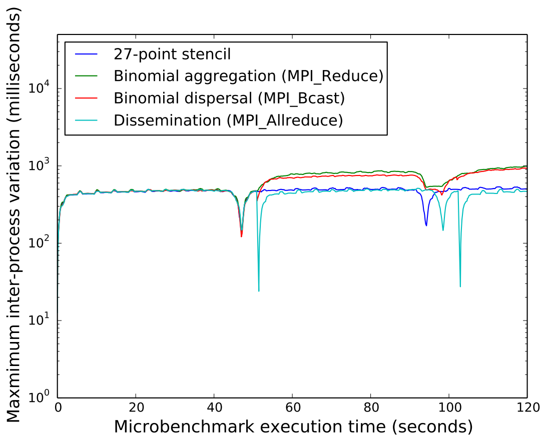
(a) 50ms inter-collective interval



(b) 500ms inter-collective interval



(c) 5s inter-collective interval



(d) 50s inter-collective interval

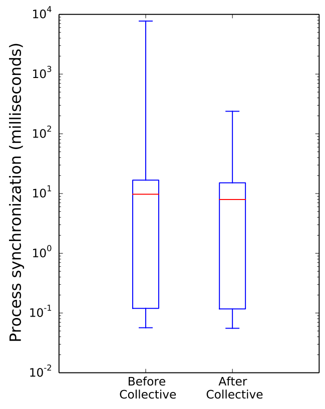
Figure 3: Microbenchmark results with Bonds interference. Maximum inter-process synchronization during the execution of three simple microbenchmarks.

4.5 Performance Impact of Synchronizing Analytics Execution

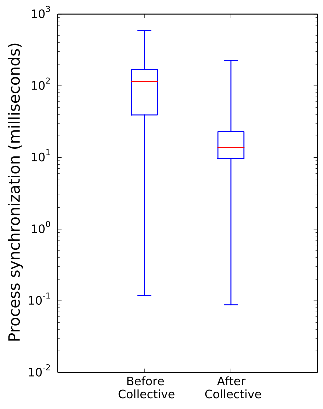
Based on the preceding experiments that characterize the extent to which MPI collective algorithms may synchronize application execution, we turn to the potential performance impact of leveraging existing synchronization to schedule analytics. We use our simulation framework to examine the impact of synchronizing the execution of the analytics code across application processes. For each of our five workloads, we simulate the degree of analytics execution synchronization by staggering the starting point of each simulated process in the analytics execution trace. We randomly select each offset from a normal distribution with mean 0. The standard deviation of the distribution corresponds to how tightly synchronized the set of processes in each trial are; based on the results presented above we use the values 0 seconds (perfectly synchronized), 10 milliseconds, 100 milliseconds, 1 second, and ∞ (totally unsynchronized).

The results of these experiments are shown in Figure 7. These data show that perfect synchronization of analytics execution is not necessary. Approximate synchronization is sufficient to dramatically reduce the performance impact of analytics execution. Specifically, if the scheduling of each execution period of the analytics execution is normally distributed with a standard deviation of 100 milliseconds (i.e. on average, 95% of the processes will be within 200 milliseconds of ideal), then the performance impact will be less than 20% for all but LAMMPS-crack, even for 64Ki processes. The data in Figure 3 suggest that this degree of synchronization can be obtained by leveraging latent synchronization in applications that regularly invoke the MPI collectives that use the dissemination algorithm (e.g. `MPI_Allreduce()`). Moreover, these data show that if the analytics can be scheduled within 20 milliseconds of ideal (i.e., if the standard deviation of the distribution is less than 10 milliseconds), which might be possible by leveraging the existing frequent calls to `MPI_Allreduce()` many applications already do (*see* Figure 3a), then the application slowdown drops below 10%.

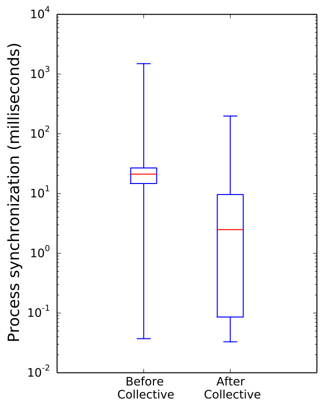
Finally, we compare the performance of existing synchronization methods with the approximate synchronization that is induced by MPI collectives that are implemented with the



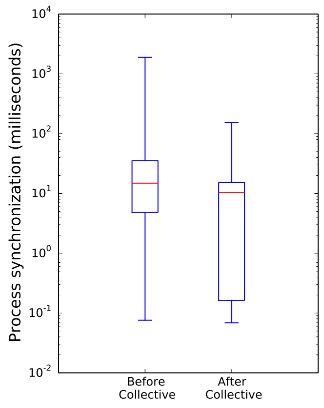
(a) CTH-st



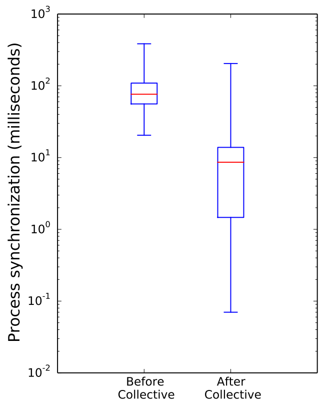
(b) HPCCG



(c) LAMMPS-crack

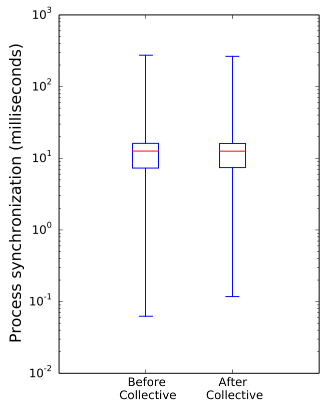


(d) LAMMPS-lj

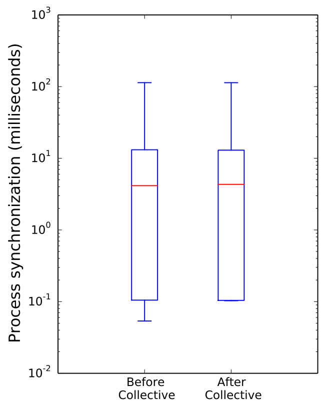


(e) LULESH

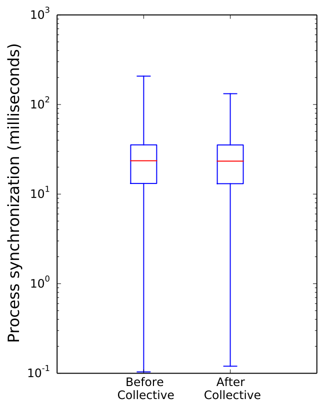
Figure 4: Impact of the dissemination algorithm (e.g., one possible implementation of `MPI_Allreduce()`) on process synchronization. The top and bottom of the boxes are the third and first quartiles, respectively. The red line is the median and the whiskers represent the minimum and maximum values.



(a) CTH-st

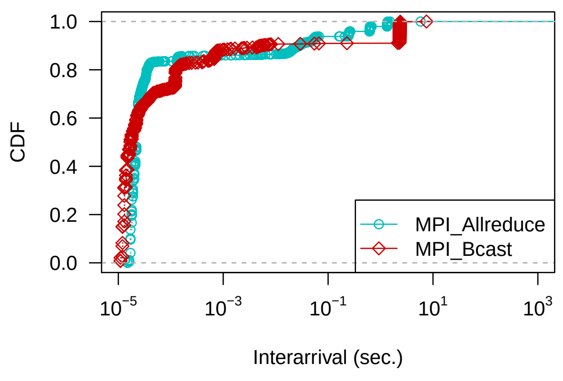


(b) LAMMPS-crack

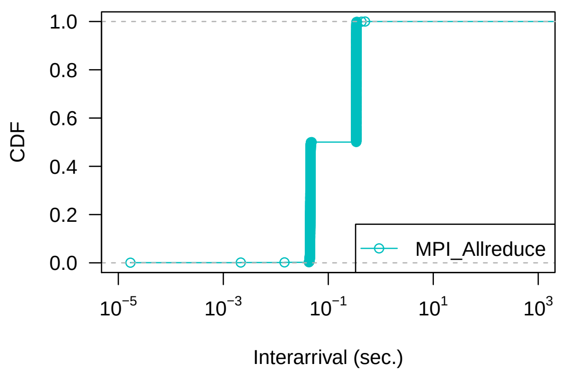


(c) LAMMPS-lj

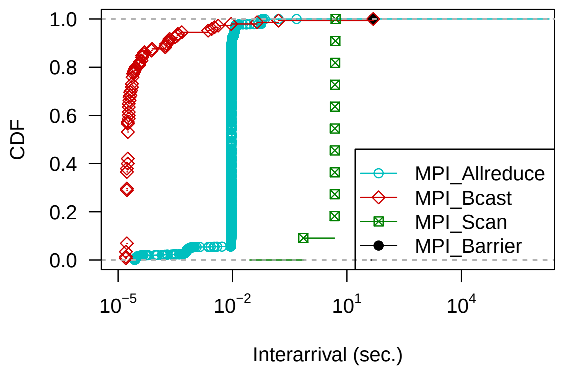
Figure 5: Impact of binomial tree dispersal (e.g., one possible implementation of `MPI_Bcast()`) on process synchronization. LULESH and HPCCG are omitted because neither uses collectives that are commonly implemented with this algorithm. The top and bottom of the boxes are the third and first quartiles, respectively. The red line is the median and the whiskers represent the minimum and maximum values.



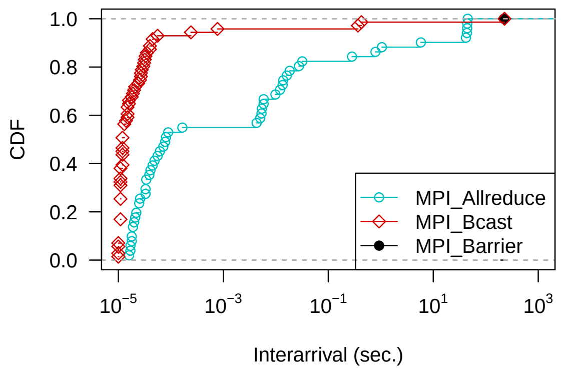
(a) CTH-st



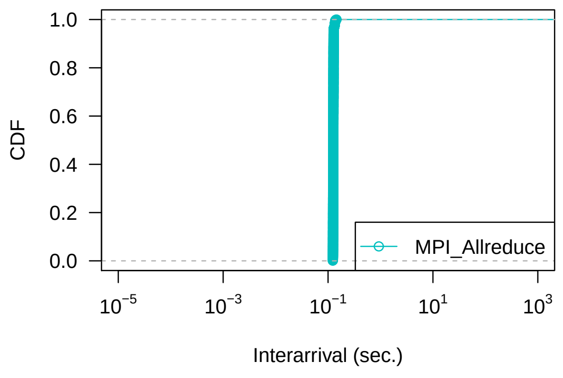
(b) HPCCG



(c) LAMMPS-crack

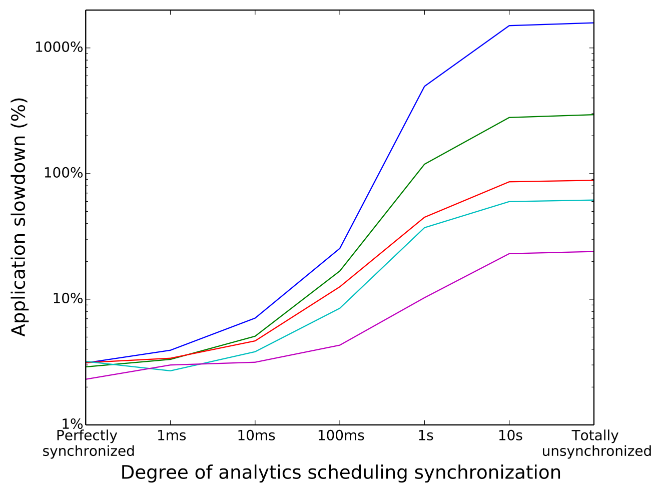


(d) LAMMPS-lj

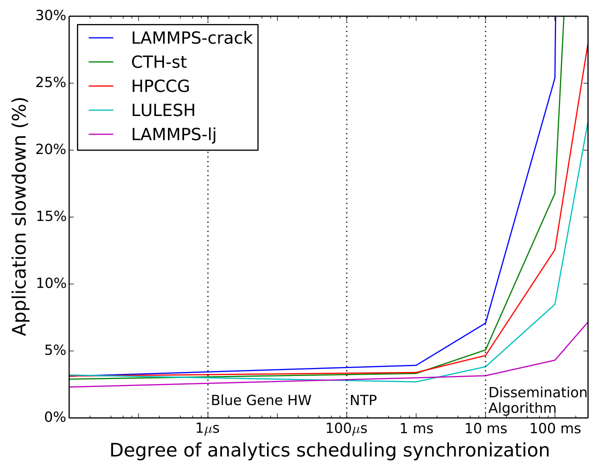


(e) LULESH

Figure 6: Discrete cumulative distribution function (CDF) of the MPI collective inter-arrival time for each application. NOTE: the CDF for `MPI_Barrier()` is represented by a single point for both LAMMPS-crack and LAMMPS-lj because only two such operations occur during their execution.



(a) Complete results



(b) Magnified view of tight synchronization results

Figure 7: Impact of analytics execution on application performance as a function of the degree of inter-process synchronization for 64 Ki processes. The magnified view is annotated to show the approximate degree of synchronization that is obtainable using three different approaches.

dissemination algorithm. We examine two common hardware-based synchronization mechanisms. First, systems like the IBM BlueGene series have a separate global interconnect for high performance time synchronization [1]. For these systems, the variance in global time has been shown to be less than 10 microseconds. For some machine configurations it may be possible to synchronize within 1 microsecond by using the global interrupt network [2,21]. Another hardware-based approach is to use highly accurate and self-adjusting clocks on each node, for example the clock provided by a Global Positioning System (GPS) card [26]. This approach has been shown to be able to synchronize per-node clocks to within 1 microsecond. Software-based techniques are also common. For example, running the network time protocol (NTP) on a leadership-class system has been shown to be able synchronize per-node clocks to within 100 microseconds [21]. In this paper, we have shown that leveraging the synchronization induced by the dissemenation algorithm in applications that rely on frequent calls to `MPI_Allreduce()` may be able to synchronize application processes to within 10 milliseconds (*see* Figure 3a).

Figure 7b is a magnified view of the results in Figure 7a for tight synchronization. This figure is annotated to show the approximate degree of synchronization that is achievable using: hardware mechanisms (e.g., the BlueGene global interrupt network), software mechanisms (e.g., NTP), and the dissemination algorithm that is commonly used to implement collectives such as `MPI_Allreduce()`. Although the synchronization induced by the dissemination algorithm is much looser than the other mechanisms, this figure shows that the impact of each of these techniques on application performance is comparable. For all three types of synchronization, the application slowdown is less than 10%. The advantage of leveraging the dissemination algorithm is that it allows the scheduling of analytics to be integrated into the natural execution of the application (e.g., at the completion of a `MPI_Allreduce()` operation) rather than by the arbitrary expiration of a timer. Moreover, leveraging collective operations does not require the addition of expensive hardware features.

5. RELATED WORK

Characterizing time-sharing impacts.

The effects of time-sharing CPU cores between applications and background system activities have been extensively studied [13,14,28,30]. Additionally, the time-sharing of resources between application and potentially more harmful workloads have been studied, including asynchronous checkpointing [15,25], and *in situ* analytics systems [40]. In contrast, our work focuses on understanding the effects of time-shared in-situ analytics applications and the requirements of mitigation techniques based on global synchronization, and without the assumption that significant serial portions exist that can be exploited as in [40]. Most closely related, Mondragon et al. [27] outline the challenges involved in scheduling independent in situ analytic tasks in emergent HPC applications. Using an optimistic analytic model, this work compares the overheads in time- verses space-shared analytics tasks. Overall, this brief work shows that the performance impact of space-shared tasks is greatly reduced in comparison to that of time-shared ones, at the cost of dedicating additional hardware to analytics. In comparison, our work demonstrates how the synchronization inherent in many modern collective algorithms, applications, and hardware can be exploited to reduce the performance impact of time-shared analytics to a level comparable to that of space-shared without requiring dedicated hardware.

Simulating time-sharing impacts.

`LogGOPSim` [20] has been used previously to study the impact of time-sharing on application performance. Hoefler et al. [19] used this simulator to evaluate the impact of background system activities on application performance. Levy et al. [25] extended `LogGOPSim` and demonstrated that this extension could be used to accurately predict the impact of resilience mechanisms on applications performance. Ferreira et al. [15] subsequently used `LogGOPSim` to demonstrate the impact of asynchronous checkpoint/restart on application performance at scale. Similarly, Widener et al. [36] used `LogGOPSim` to study how non-blocking collectives might be used to reduce the performance penalties associated with asynchronous checkpoint/restart. In this work, we leverage `LogGOPSim` to study the impacts of time-sharing analytics codes and the ability of the global synchronization latent in many current MPI collective operations to mitigate overheads.

Other works have used simulation- and emulation-based approaches to study time-sharing. Engelmann et. al. [10] use an over-subscription method in their extreme scale simulator xSim [5] to investigate the effects of time-sharing CPU cores among short-lived system tasks on HPC systems with up to 2M nodes and 1 billion cores. They study the effects of interference amplification and absorption on MPI collectives and propose this framework as a tool to be used in hardware/software co-design of future HPC systems. Pradipta De et. al. [8] propose an emulation-based approach to study similar performance impacts. These over-subscription and emulation approaches, however, have significant limitations which do not apply to our work. First, while these approaches can potentially provide a tremendous amount of detail about the performance of applications, significant costs exist. Due to limits on the degree of over-subscription and detail in emulation, the physical system size required for their execution increases proportionally to the size of the studied application. Moreover, as the size of the simulated system grows and the degree of over-subscription therefore increases, the required simulation time increases dramatically. Lastly, over-subscription and system emulation can place significant limits on the size of the problem that can be studied as the memory for each simulated node must exist in the memory of one physical node. In contrast, our approach allows us to simulate interference overheads for systems comprised of tens or hundreds of thousands of nodes on very modest hardware (e.g., a single node). In many cases, this simulation completes in less time than it would take to run the application itself, but at the cost of producing possibly less detail about the computation [25].

Synchronization Mechanisms.

Feitelson et. al. [12], in one of the first works that studied gang-scheduling, compared two synchronization techniques: a busy-waiting-based gang-scheduling technique implemented by the runtime, which allowed applications processes to be scheduled at the same time; and a blocking, with uncoordinated scheduling, synchronization mechanism. They show that fine-grained parallel applications benefit from using the gang-scheduling policy and otherwise incur context switching overheads while blocking (i.e., the processor switches to another thread during each block) to perform synchronization.

More recently, Jones et al. [22] proposed a mitigation approach based on co-scheduling system background tasks. They show the performance benefits of this approach on BSP applications. By decreasing interference, this approach facilitates the communication between application processes and reduces the amount of time that the fastest processes spend waiting for group operations (e.g. MPI collectives). In this work, we focus on the performance impact of coupled application and analytics workloads and leveraging the side-effect global coordination existing in current production workloads to mitigate impacts.

6. CONCLUSION AND FUTURE WORK

As data-intensive MPI applications become more prevalent, understanding the performance implications of techniques such as *in situ* analytics will also become critical for accurate performance prediction on next-generation systems. Using a simulation-based approach, we have in this paper examined how the synchronization of time-shared *in situ* analysis tasks can affect overall application performance. We have demonstrated that some degree of synchronization is desirable in order to prevent severe slowdowns. We have shown that, perhaps counter to intuition, that strong performance benefits are possible without necessarily resorting to expensive tight synchronization solutions. In particular, our results show that even relatively loose synchronization (e.g. within a few hundred milliseconds of ideal) of the execution of *in situ* analysis tasks is sufficient to significantly reduce slowdowns due to interference. For many important MPI applications, these benefits may be realizable simply by relying on the latent synchronization induced by existing invocations of MPI collectives implemented with the dissemination algorithm, like `MPI_Allreduce()`.

More work is needed along several axes for a complete understanding of the types of performance interference we have studied here. While introducing fast "nearby" storage such as NVRAM into supercomputer designs may reduce contention for I/O bandwidth, issues surrounding data volume and how to most efficiently reduce it are unlikely to disappear completely for large-scale MPI applications and their associated analytic codes. *In situ* analysis is part of a spectrum of techniques being researched to address these issues, and the performance impact of interference between, for instance, processes involved in an "in-transit" arrangement have not been explored in any detail. Also, increasing core counts mean that some applications may choose to use a space-sharing approach for their *in situ* data analysis, with effects that remain to be explored. Finally, analysis codes that perform non-trivial communication of their own (independent of the main simulation) may in turn become more commonplace; characterizing the degree to which such communication may provide even tighter inter-process synchronization is part of our ongoing and future work.

7. REFERENCES

[1] N. R. Adiga, G. Almási, Y. Aridor, R. Barik, D. K. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. A. Blumrich, A. A. Bright, J. R. Brunheroto, C. Cascaval, J. G. Castaños, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. L. Chiu, T. M. Cipolla, P. Crumley, K. M. Desai, A. Deutsch, T. Domany, M. B. Dombrowa, W. E. Donath, M. Eleftheriou, C. C. Erway, J. Esch, B. G. Fitch, J. Gagliano, A. Gara, R. Garg, R. S. Germain, M. Giampapa, B. Gopalsamy, J. A. Gunnels, M. Gupta, F. G. Gustavson, S. Hall, R. A. Haring, D. F. Heidel, P. Heidelberger, L. Herger, D. Hoenicke,

R. D. Jackson, T. Jamal-Eddine, G. V. Kopcsay, E. Krevat, M. P. Kurhekar, A. P. Lanzetta, D. Lieber, L. K. Liu, M. Lu, M. P. Mendell, A. Misra, Y. Moatti, L. S. Mok, J. E. Moreira, B. J. Nathanson, M. Newton, M. Ohmacht, A. J. Oliner, V. Pandit, R. B. Pudota, R. A. Rand, R. D. Regan, B. Rubin, A. E. Ruehli, S. Rus, R. K. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. D. Steinmacher-Burow, K. Strauss, C. W. Surovic, R. A. Swetz, T. Takken, R. B. Tremaine, M. Tsao, A. R. Umamaheshwaran, P. Verma, P. Vranas, T. J. C. Ward, M. E. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. J. Krolak, C. Li, T. A. Liebsch, J. A. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. D. Wait, J. Wittrup, M. Bae, K. A. Dockser, L. Kissel, M. K. Seager, J. S. Vetter, and K. Yates. An overview of the BlueGene/L supercomputer. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Baltimore, Maryland, USA, November 16-22, 2002, CD-ROM*, pages 7:1–7:22, 2002.

[2] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 253–262, New York, NY, USA, 2005. ACM.

[3] F. Aurenhammer. Voronoi diagrams — a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1992.

[4] J. C. Bennett, H. Abbasi, P. T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–9, Nov 2012.

[5] S. Böhm and C. Engelmann. xSim: The extreme-scale simulator. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 280–286. IEEE, 2011.

[6] L. Chacón. A non-staggered, conservative, finite-volume scheme for 3D implicit extended magnetohydrodynamics in curvilinear geometries. *Computer Physics Communications*, 163(3):143–171, 2004.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.

[8] P. De, R. Kothari, and V. Mann. A trace-driven emulation framework to predict scalability of large clusters in presence of OS jitter. In *Cluster Computing, 2008 IEEE International Conference on*, pages 232–241. IEEE, 2008.

[9] J. E. S. Hertel, R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. PetneY, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A software family for multi-dimensional shock physics analysis. In *Proceedings of the 19th Intl. Symp. on Shock Waves*, pages 377–382, July 1993.

[10] C. Engelmann. Investigating operating system noise in extreme-scale high-performance computing systems using simulation. In *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2013*, pages 11–13, 2013.

[11] Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). http://exmatex.lanl.gov/. Retrieved 16 Jan 2014.

[12] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[13] K. B. Ferreira, P. G. Bridges, R. Brightwell, and K. Pedretti. Impact of system design parameters on application noise sensitivity. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing (Cluster 2010)*, September 2010.

[14] K. B. Ferreira, R. Brightwell, and P. G. Bridges. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*, November 2008.

[15] K. B. Ferreira, P. Widener, S. Levy, D. Arnold, and T. Hoefler. Understanding the effects of communication and coordination on checkpointing at scale. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, 2014.

[16] M. P. I. Forum. MPI: A Message-Passing Interface Standard Version 3.0, 09 2012.

[17] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17(1):1–17, Feb. 1988.

[18] 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*, 2009.

[19] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[20] T. Hoefler, T. Schneider, and A. Lumsdaine. LogGOPSim: simulating large-scale applications in the LogGOPS model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 597–604. ACM, 2010.

[21] T. Jones. Personal communication, May 2016.

[22] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, et al. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 10–10. IEEE, 2003.

[23] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen,

Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong. LULESH programming model and performance ports overview. Technical Report LLNL-TR-608824, Lawrence Livermore National Laboratory, December 2012.

[24] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney. Grid-based parallel data streaming implemented for the gyrokinetic toroidal code. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 24. ACM, 2003.

[25] S. Levy, B. Topp, K. B. Ferreira, D. Arnold, T. Hoefler, and P. Widener. Using simulation to evaluate the performance of resilience strategies at scale. In *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, pages 91–114. Springer, 2014.

[26] A. Mink, R. J. Carpenter, and M. Courson. Time synchronized measurements in cluster computing systems. In *Proceedings of the 13th International Conference on Parallel and Distributed Computing*, PDSC2000, pages 1–7, New York, NY, USA, 2000. IEEE.

[27] O. H. Mondragon, P. G. Bridges, S. Levy, K. B. Ferreira, and P. Widener. Scheduling in-situ anaytics in next-generation applications. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, page 4. ACM, 2016.

[28] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, and P. Beckman. The ghost in the machine: Observing the effects of kernel operation on parallel application performance. In *Proceedings of SC'07*, 2007.

[29] T. Peterka, J. Kwan, A. Pope, H. Finkel, K. Heitmann, S. Habib, J. Wang, and G. Zagaris. Meshing the universe: integrating analysis in cosmological simulations. In *Proc. SC12 Ultrascale Visualization Workshop*, Salt Lake City, UT, November 2012.

[30] F. Petrini, D. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of SC'03*, Phoenix, AZ, 2003.

[31] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

[32] B. Prisacari, G. Rodriguez, C. Minkenberg, and T. Hoefler. Bandwidth-optimal all-to-all exchanges in fat tree networks. In *Proceedings of the 27th International ACM International Conference on Supercomputing*, ICS '13, pages 139–148, New York, NY, USA, 2013. ACM.

[33] S. Rostedt. Debugging the kernel using ftrace. http://lwn.net/Articles/365835/, 2009.

[34] Sandia National Laboratory. Mantevo project home page. http://mantevo.org, Jan. 2014.

[35] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[36] P. Widener, K. B. Ferreira, S. Levy, and T. Hoefler. Exploring the effect of noise on the performance benefit of nonblocking allreduce. In *Proceedings of the 21st European MPI Users' Group Meeting*, page 77. ACM, 2014.

[37] M. Wolf, Z. Cai, W. Huang, and K. Schwan. SmartPointers: personalized scientific data portals in your hand. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 20–20. IEEE, 2002.

[38] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K.-L. Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, May/June 2010.

[39] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA–preparatory data analytics on peta-scale machines. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[40] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky. GoldRush: resource efficient in situ scientific data analytics using fine-grained interference aware execution. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 78. ACM, 2013.