On noise and the performance benefit of nonblocking collectives

Patrick M. Widener[1], Scott Levy[2], Kurt B. Ferreira[1], and Torsten Hoefler[3]

[1]Center for Computing Research, Sandia National Laboratories *, Albuquerque NM
[2]Department of Computer Science, University of New Mexico, Albuquerque NM
[3]Computer Science Department, ETH Zürich, Switzerland

Abstract

Relaxed synchronization offers the potential of maintaining application scalability by allowing many processes to make independent progress when some processes suffer delays. Yet, the benefits of this approach in important parallel workloads have not been investigated in detail. In this paper, we use a validated simulation approach to explore the noise mitigation effects of idealized nonblocking collectives in workloads where these collectives are a major contributor to total execution time. Although nonblocking collectives are unlikely to provide significant noise mitigation to applications in the low-OS-noise environments expected in next-generation HPC systems, we show that they can potentially improve application runtime with respect to other noise types.

1 Introduction

Nonblocking collective operations [15,19], newly introduced in MPI-3.0 [12,24], allow application programmers to overlap collective communication with the application's computation. At scale, blocking collective operations can significantly degrade application performance because, in most cases, each application process must participate in the collective before any process can make further progress. As a result, a laggard process can slow the progress of all of its peers and consequently the entire application. Nonblocking collectives have the potential to alleviate the impact of process variability by allowing a process to make progress even if its peers are late entering a collective.

Delayed participation in a collective operation can be caused by several phenomena, including load imbalance, fault tolerance activities, and operating system *noise*. While noise exists on most computing platforms, it has a disproportionate effect in HPC systems because the applications that run on these systems tend to be highly synchronized. For many important HPC applications, operating system noise has been shown to have serious consequences for overall application performance [8].

**Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly-owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.*

The effects of operating system (OS) noise on high-performance computing (HPC) applications have been examined for insights into designing scalable hardware, system software, and applications [1, 25]. Recently, system designers have invested time and effort to reduce or eliminate sources of noise in the OS. These efforts have significantly reduced the noise introduced by system software on HPC systems; there now exist operating systems that are essentially noiseless (e.g., IBM's CNK) [16, 30]. As a result, the need for nonblocking collectives to address the performance impact of OS noise appears to be waning[1].

Although the impact of OS noise may no longer be a significant issue, there are many potential sources of noise in future extreme-scale systems. For example, fault tolerance is projected to be a significant challenge on future systems. The dominant approach to fault tolerance is checkpoint/restart. Because checkpointing activities deprive the application of CPU cycles, they can be modeled as OS noise [23]. Other trends in extreme-scale system design (e.g., adaptive runtimes, in situ analytics) also have the potential to introduce noise-like events that impact application performance. As a result, even if OS noise is itself no longer an issue, emerging sources of noise mean that nonblocking collectives may still yield a performance benefit on next-generation extreme-scale systems.

In this paper, we investigate the performance benefit of nonblocking collective operations on application performance at scale, in different noise environments. We begin with a simple analytical model of application performance to identify upper bounds on application speedup. We use simulation to further examine the impact of nonblocking collectives. We focus on a simulated `MPI_Iallreduce()` (i.e., the nonblocking variant of `MPI_Allreduce()`) because, as we will show, it is the dominant communication operation in many important parallel workloads. To examine the potential upside of `MPI_Iallreduce()`, we use a validated simulator to compare the standard `MPI_Allreduce()` with an idealized cost-free `MPI_Iallreduce()`. We also use a set of microbenchmarks to examine the different potentials for application speedup of dissemination vs. binomial-tree collective algorithms. Specifically, we show that:

- in low-noise environments that are typical of current and expected future systems, `MPI_Iallreduce()` is unlikely to provide significant noise mitigation benefits without algorithmic changes;
- for other sources of noise such as resilience protocol overheads, `MPI_Iallreduce()` may have a significant beneficial effect on application performance; and
- the expected noise mitigation effect from nonblocking collectives can vary significantly according to their underlying implementation algorithms and the interprocess dependencies those algorithms create.

[1]Other uses for nonblocking collectives include overlapping computation and communication as well as complex synchronization protocols [18].

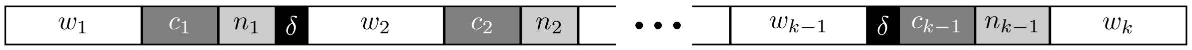


Figure 1: A simple, hypothetical example of a trace of the execution of a single application process with blocking collectives. We model the execution trace as a sequence of interleaved periods of communication (c_i) plus noise propagation (n_i), work (w_i) and checkpoints (δ). Communication intervals consist of blocking communication operations, including delays introduced by remote checkpointing noise. Work intervals consist of non-blocking point-to-point communication and periods of computation. A checkpoint may occur at any point in a work interval, but we model checkpoints as an extension of the duration of the checkpoint-free work interval. As a result, identifying the precise moment at which a checkpoint occurs is unnecessary.

2 Experimental Approach

In this paper, we use an analytic model and a validated simulator to explore the potential benefits of nonblocking collectives in the presence of different noise profiles. Our analytic model allows us to identify upper bounds on the speedup of an application if nonblocking collective operations are allowed only to overlap with computation (i.e., a nonblocking collective cannot overlap with any other communication operation). For bulk synchronous parallel (BSP) applications, this is roughly equivalent to allowing collectives to overlap with a single application iteration. Simulation allows us to explore models of communication/computation overlap that are difficult to capture with an analytic model. In particular, simulation allows us to examine the performance impact of allowing nonblocking collectives to overlap with each other and with blocking point-to-point collective operations. Moreover, simulation allows us to more accurately account for noise effects and their impact on the performance of collective operations.

2.1 Modeling nonblocking collectives

One of the principal benefits of nonblocking collectives is that they allow applications to overlap computation and collective communication. As a result, they can help mitigate the impact of noise by allowing application processes to continue to advance, even if one or more communication partners are delayed by noise. For the purposes of this section, we focus on noise introduced by checkpointing activities. However, our results can be easily extended to other sources of noise. To determine the potential benefits of allowing computation and collective communication to overlap, we begin with a simple model of application execution. As shown in Figure 1, we model the execution trace of each application process as an interleaved sequence of blocking communication operations and computation; the application's time-to-solution is the maximum of the execution times of its constituent processes.

We refer to periods of computation (including checkpointing) and nonblocking point-to-point operations (e.g., `MPI_Isend()`, `MPI_Irecv()`) as *work intervals* (w_i). We model periods when the application is computing checkpoints as the expansion of individual work intervals. For this simple model, we need not distinguish between the checkpoints that are taken collectively (i.e., coordinated checkpointing) or independently (i.e., uncoordinated checkpointing). Collective operations and blocking point-to-point operations comprise *communication intervals* (c_i). In the presence of noise, communication events may require additional time to complete because one or more of the participants has been delayed by noise events. We capture these additional communication costs as *noise effects* (n_i). This simple execution model allows us to express an application's time-to-solution as:

$$T_s = W_k + \sum_{i=1}^{k-1}(c_i + n_i + W_i) \tag{1}$$

where:

$$\begin{aligned}
k &= \text{number of communication and work intervals that comprise the application's execution} \\
c_i &= \text{duration of the } i^{th} \text{ communication interval} \\
n_i &= \text{duration of the noise effects associated with the } i^{th} \text{ communication interval} \\
W_i &= w_i + d(i) \text{ (duration of the } i^{th} \text{ work interval)} \\
w_i &= \text{duration of computation in the } i^{th} \text{ work interval} \\
d(i) &= \begin{cases} \delta & \text{if a checkpoint occurs during } w_i \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

We note that by changing the definition of $d(i)$, this model can be applied to other sources of noise. In this instance, the benefit of nonblocking collectives depends on the extent to which collective communication (including noise effects) can overlap work intervals. Conceptually, the simplest way to model this overlap is to allow each collective operation only to overlap with the work interval that immediately precedes it. The idea is that nonblocking collectives will allow the application to initiate the collective at an earlier point in the execution than when blocking collectives are used.[2]

In this model, we do not allow nonblocking collectives to overlap with each other or with blocking point-to-point operations. We assume that: (i) allowing work and communication to overlap does not alter the time required to complete any communication operation (i.e., the duration of noise effects is dependent only on the associated communication interval); (ii) with nonblocking collectives, checkpoints occur in the same work interval as for blocking collectives (i.e., the checkpoint interval is determined by the amount of work completed); and (iii) the target application can tolerate complete overlap of work and communication

[2]Alternatively, we could model the communication/computation overlap by allowing nonblocking collectives to overlap only with the succeeding computation interval. The result obtained in this section would be the same.

intervals. Given this model of communication/computation overlap, we can then express the application's time-to-solution as:

$$\begin{aligned}
\hat{T}_s &= W_k + \sum_{\substack{i=1 \\ c_i=collective}}^{k-1} \max(c_i + n_i, W_i) + \sum_{\substack{i=1 \\ c_i=p2p}}^{k-1} (c_i + n_i + W_i) \\
&= W_k + \sum_{\substack{i=1 \\ c_i=collective \\ (c_i+n_i)\geq W_i}}^{k-1} (c_i + n_i) + \sum_{\substack{i=1 \\ c_i=collective \\ (c_i+n_i)<W_i}}^{k-1} W_i + \sum_{\substack{i=1 \\ c_i=p2p}}^{k-1} (c_i + n_i + W_i) \qquad (2)
\end{aligned}$$

2.2 Simulation environment

Our simulation approach is based on capturing and examining the communication structure of applications. This structure reflects synchronization of applications and exposes dependencies through the establishment of *happens-before* relations [22]. This is especially useful for the examination of asynchronous operations because of the possible formation of transitive dependencies between processes which do not communicate directly. For MPI programs, static analysis of communication structure is complicated by the difficulty of both offline message matching [3] and modeling interactions analytically. We use instead a discrete-event simulator and model application communication as events.

Our simulator framework comprises `LogGOPSim` [17] and the tool chain developed by Levy et al. [23]. `LogGOPSim` uses the LogGOPS model, an extension of the well-known LogP model [4], to simulate application traces that contain all exchanged messages and group operations. In this way, `LogGOPSim` reproduces all happens-before dependencies and the transitive closures of all delay chains of the application execution. It can also extrapolate traces from small application runs with p processes to application runs with $k \cdot p$ processes. The extrapolation produces exact communication patterns for all collective communications and approximates point-to-point communications. Noise injection into simulations is done by constructing a time-indexed list of detours and their durations; this list is given as input to the simulator which introduces the described delays into the execution of the simulated application.

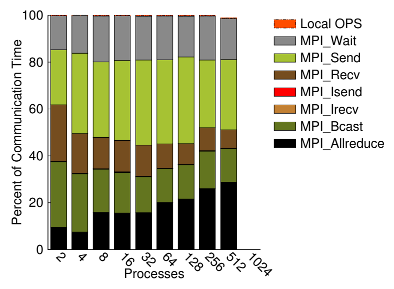
`LogGOPSim` and its trace extrapolation features have been validated [16,17]. The complete tool chain has been validated against experiments and established models [23].

2.3 HPC Workload Descriptions

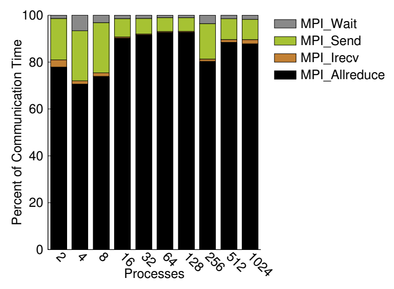
We present results from the simulation and analysis of a set of workloads. These workloads represent scientific applications that are currently in use and computational kernels thought to be important for future extreme-scale computational science. They include:

- CTH, a multi-material, large deformation, strong shock wave, solid mechanics code [6],
- HPCCG, a conjugate gradient benchmark code that is part of the Mantevo [14] suite of mini-apps,
- LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [26], a classical molecular dynamics code developed at Sandia National Laboratories,
- LULESH, the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics proxy application [7, 21], used by the Extreme Materials at Extreme Scale co-design center at Los Alamos National Laboratory [7],
- miniFE, the Mantevo [14] finite element mini-application which implements kernels representative of the implicit finite-element suite of mini-apps,
- AMG, an algebraic multigrid solver for linear systems arising from problems on unstructured grids [13].

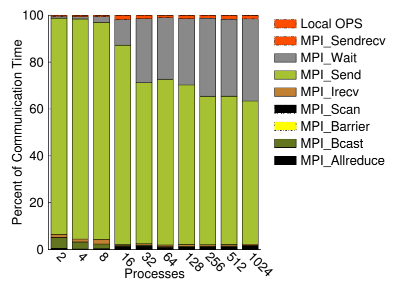
Taken as a whole, these applications: are typically run (or are projected to be run) at extreme-scale, run for long periods of time, and represent a diverse mix of computational techniques and methods.



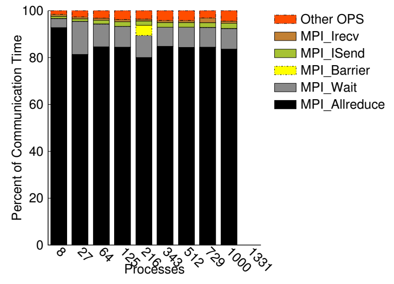
(a) CTH



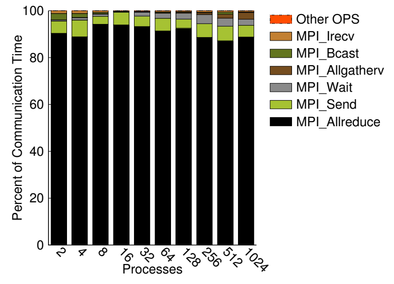
(b) HPCCG



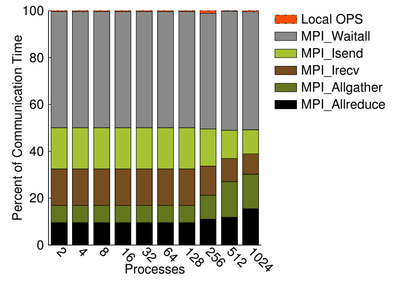
(c) LAMMPS



(d) LULESH



(e) MiniFE



(f) AMG2013

Figure 2: Percentage of communication time spent in each MPI function for CTH, HPCCG, LAMMPS, LULESH, MiniFE, and AMG.

2.4 Choosing collectives to examine

Previous investigations [8] have identified `MPI_Allreduce()` and other collectives as operations that are particularly sensitive to OS noise. Exploratory implementations of nonblocking collectives [15] have been developed to address these and other issues. To support our inquiry into the effect of noise on nonblocking collective operations, we break down the composition of MPI operations for our set of applications.

Figure 2 shows the percentage of communication time spent in various MPI functions for each of the applications we consider. These results were collected using the `mpiP` profiling library [29]. `MPI_Allreduce()` is by far the dominant collective operation across this application set. The communication time of LULESH and MiniFE is dominated by `MPI_Allreduce()` for process counts up to 1024. Similarly, HPCCG spends more than 80% of its communication time in `MPI_Allreduce()` at almost all the tested process counts. In contrast, CTH spends less than 30% of its time in `MPI_Allreduce()` (although the fraction increases significantly as the size of the application increases), and LAMMPS spends comparatively little time there. `MPI_Allgather()` is the only other collective to contribute a non-trivial proportion of communication time, and that only in AMG2013. These results suggest that a nonblocking `MPI_Allreduce()` implementation might prove beneficial for at least HPCCG, LULESH, and MiniFE, and that `MPI_Allgather()` may also be of interest.

The communication patterns of our set of workloads are largely dominated by `MPI_Allreduce()`. While this usage is representative of many current scientific applications, some future representative set of applications may use other collectives most often. We constructed a set of microbenchmarks to exercise our simulated nonblocking approach for other collective operations. To obtain different perspectives, we chose two collectives typically implemented using a dissemination algorithm (`MPI_Allreduce()` and `MPI_Allgather()`) and two typically implemented using a binomial tree algorithm (`MPI_Scatter()` and `MPI_Gather()`) [28] (assuming that these implementation algorithms will not change when MPI libraries with nonblocking collectives are available). We present evaluations of simulations of these microbenchmarks along with simulations of our workload set in Sections 3 and 4.

2.5 Simulating nonblocking collectives

By default, our simulator generates patterns of blocking communications to simulate collective operations. For example, when an `MPI_Allreduce()` is encountered in an application trace, the simulator generates SEND and RECV events using a dissemination algorithm similar to that used by MPICH and Open MPI. We added to the simulator the ability to select different algorithms for implementing the `MPI_Allreduce()` operation. For this work, we emulate an optimal nonblocking implementation, but do so in a manner that does not require modification of the application. This operation works as follows: the nonblocking allreduce implementation assumes the the application was able to start the operation early enough and has sufficient overlapping progress such that when the `MPI_Allreduce()` operation is reached by a node, it completes immediately and incurs no further overhead. Simulated nonblocking versions of other collectives are similarly generated.

This approach assumes it is actually possible to re-factor the application in a manner that the non-blocking collectives can be fully utilized (cf. [10]). For some application workloads, this re-factoring may not be possible due to algorithmic details. Therefore, our strategy for nonblocking allreduce provides an optimistic upper bound on the performance speedups that can be realized, but this upper bound is not necessarily tight. In addition to nonblocking collectives, this work may also be useful in studying systems where collectives are supported by targeted optimizations or specialized networking and hardware, for example.

3 Low-noise environments

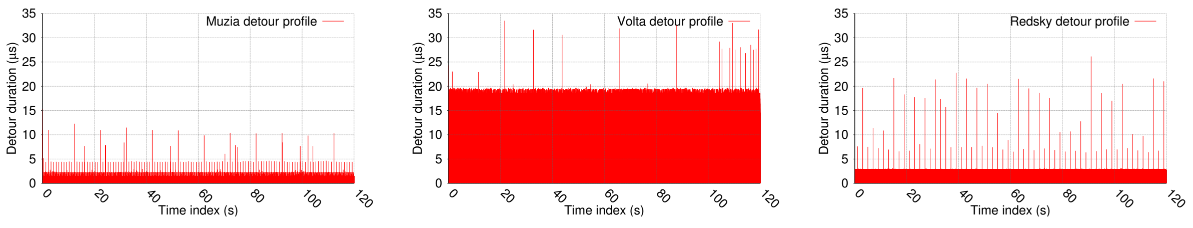
Nonblocking collective operations allow the application to overlap collective communication with computation. As a result, they have the potential to mitigate the performance impact of system noise [8] by allowing the application to absorb noise events. However, it is not clear how much noise absorption nonblocking collectives enable or how much their noise absorption impacts application performance. Therefore, we investigate the performance impact of nonblocking collectives using system noise patterns collected on current systems and a synthetic noise pattern we believe to be representative of future extreme-scale systems.

3.1 Characterizing system noise

We measure the noise environment on current HPC systems and consider how it might differ on future systems. Figure 3 shows the idle system noise patterns collected on three systems using the `selfish` [16] system noise measurement microbenchmark: *Volta*, a Cray XC30; *Muzia*, a Cray XE6; and *RedSky*, a Sun-Blade x6275 capacity system with Infiniband. Both Cray systems run the Cray Linux Environment (CLE), Cray's optimized Linux kernel, and *RedSky* uses CentOS-based software on the compute nodes.

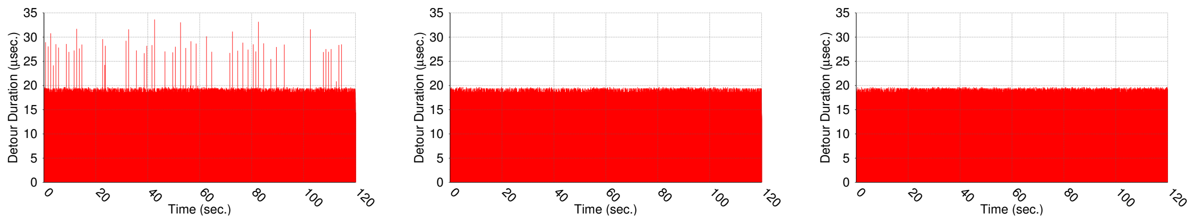
From the figure, we see that *Muzia* has the lowest system noise signature, *RedSky* has a slightly larger degree of noise, and *Volta* has the largest volume of the three. Most importantly, we see from this figure that all of these systems have comparatively low noise signatures, with all noise event durations less than 35 microseconds. Note that, while CLE has been tuned to reduce its noise impact on applications, neither CentOS or CLE approaches the extremely low level of noise interference present in lightweight kernels such as those in the Catamount family [27] or IBM's CNK [11]. We focus on more heavyweight kernels as we expect that the majority of future applications will execute in those environments[3].

[3] [27] and [11] discuss tradeoffs between development effort and performance benefit posed by the use of lightweight kernels



(a) *Muzia* (b) *Volta* (c) *RedSky*

Figure 3: Noise profiles for three Sandia cluster systems collected using the **selfish** detour collection tool.



(a) No CPU Cores Bound to OS (b) OS bound to One Core (c) OS bound to Four Cores

Figure 4: Impact of Cray's OS core specialization on the *Volta* cluster.

As previous work has shown that system noise events can significantly impact HPC performance [8,16], great effort has been made to lower the duration of noise events. For example, Cray introduced OS core specialization in the CLE which allows the user to bind CPU cores to OS and system software tasks, thereby reducing the system noise on the application cores. Figure 4 shows the impact of this core specialization functionality on `selfish` noise traces collected on *Volta*, with 0, 1, and four cores dedicated to system software tasks, respectively. Core specialization can greatly "smooth" unpredictable spikes in noise duration, which is of great benefit to applications even if system noise is not notably reduced in general.

3.2 Modeling nonblocking collectives in a low-noise environment

In low-noise environments, we expect noise effects (n_i) to be small. If we assume that $c_i \gg n_i$ for all i, we can rewrite the time-to-solution with blocking collectives (Equation 1) as:

$$T_s \approx W_k + \sum_{i=1}^{k-1}(c_i + W_i)$$

Similarly, we can rewrite the time-to-solution with non-blocking collectives (Equation 2) as:

$$\hat{T}_s \approx W_k + \sum_{\substack{i=1 \\ c_i=collective}}^{k-1} \max(c_i, W_i) + \sum_{\substack{i=1 \\ c_i=p2p}}^{k-1} (c_i + W_i)$$

which inform this expectation.

$$= W_k + \sum_{\substack{i=1 \\ c_i=collective \\ c_i \geq W_i}}^{k-1} c_i + \sum_{\substack{i=1 \\ c_i=collective \\ c_i < W_i}}^{k-1} W_i + \sum_{\substack{i=1 \\ c_i=p2p}}^{k-1} (c_i + W_i)$$

We can then express the potential speedup in application execution as:

$$\begin{aligned}
\text{speedup} &= \frac{T_s}{\hat{T}_s} \\
&= \frac{W_k + \sum_{i=1}^{k-1}(c_i + W_i)}{W_k + \sum\limits_{\substack{i=1 \\ c_i=collective \\ c_i \geq W_i}}^{k-1} c_i + \sum\limits_{\substack{i=1 \\ c_i=collective \\ c_i < W_i}}^{k-1} W_i + \sum\limits_{\substack{i=1 \\ c_i=p2p}}^{k-1} (c_i + W_i)} \\
&= \frac{W_k + \sum_{i=1}^{k-1}(c_i + W_i)}{W_k + \sum\limits_{\substack{i=1 \\ c_i=collective}}^{k-1} W_i + \sum\limits_{\substack{i=1 \\ c_i=collective \\ c_i \geq W_i}}^{k-1} (c_i - W_i) + \sum\limits_{\substack{i=1 \\ c_i=p2p}}^{k-1} (c_i + W_i)} \\
&\leq \frac{W_k + \sum_{i=1}^{k-1}(c_i + W_i)}{W_k + \sum_{i=1}^{k-1} W_i}
\end{aligned} \tag{3}$$

If the application's computation-to-communication ratio is β, then we can rewrite Equation 3 as:

$$\text{speedup} \leq \frac{1 + \beta}{\beta} \tag{4}$$

A similar derivation yields:

$$\begin{aligned}
\text{speedup} &\leq \frac{W_k + \sum_{i=1}^{k-1}(c_i + W_i)}{W_k + \sum_{i=1}^{k-1} c_i} \\
&\leq 1 + \beta
\end{aligned} \tag{5}$$

Both Equations 4 and 5 place an upper bound on the achievable speedup. If $\beta \geq 1$ (i.e., the application spends more time performing computation than communication) then Equation 4 is a tighter bound. Conversely, if $\beta < 1$ then Equation 5 is a tighter bound. We observe that for applications that spend a

Application	Computation-to-Communication Ratio	Maximum Speedup
AMG2006	1.59	1.63
CTH	1.25	1.80
HPCCG	95.15	1.01
LAMMPS	37.46	1.03
LULESH	4.37	1.23
miniFE	10.31	1.10

Table 1: Computation-to-communication ratios and the corresponding upper bounds on application speedup for each of our workloads running on 1024 nodes

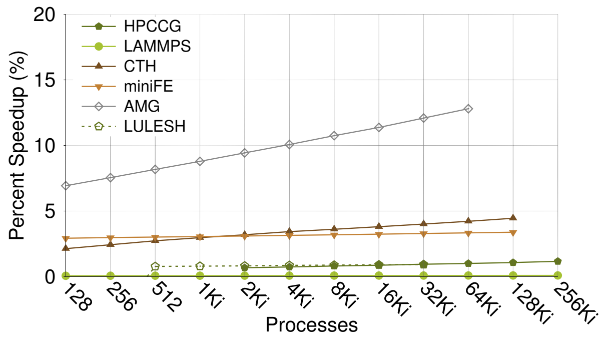
considerable fraction of their communication time in point-to-point operations (*see* Figure 2) these are not tight upper bounds.

In Table 1, we consider the implications of these upper bounds given the computation-to-communication ratios for our set of workloads. These ratios are empirically determined by running each workload on 1024 nodes. The measurements in this table show that all of these applications spend more time in computation than communication. However, there is considerable variation across applications in the total fraction of execution time that is used for computation. For some of the applications (e.g., AMG2006 and CTH) the time spent in communication is relatively large. As a result, the benefit of using nonblocking collectives to overlap communication and computation is potentially significant. At the other end of the spectrum, the execution time of HPCCG, LAMMPS and miniFE are dominated by computation. The benefits of using nonblocking collectives for these applications are much more modest.

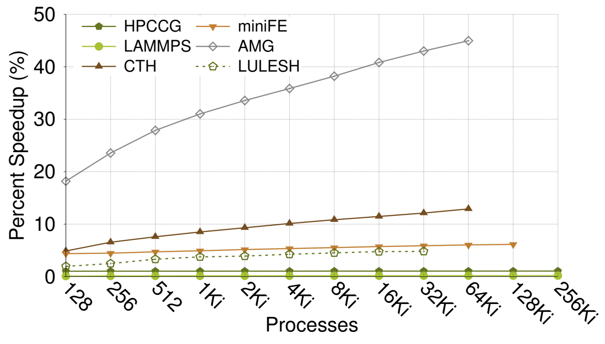
3.3 Application simulation results

We conducted simulations of our target applications to examine how they would respond in the presence of different types of noise. We used three different noise traces: *system noise*, the worst-case idle system noise signature from *Volta* as found in Figure 4(a); *daemon noise*, a synthetic noise trace with detour durations of 2.5ms at a frequency of 10 Hz, designed to represent the types of detours that would be introduced by a typical system-level daemon process and similar to one used in previous work [8]; and *asynchronous checkpointing noise*, a 1-second duration, 120-second period application interference pattern similar to processes taking checkpoints in an uncoordinated checkpoint/restart resilience scheme at extreme scale [23]. In each case, we measured the application speedup of our idealized `MPI_Iallreduce()` implementation when compared to the use of a normal blocking `MPI_Allreduce()`, both in the presence of noise.

The speedup results are presented in Figure 5. Subfigure 5(a) shows that in the presence of system noise, `MPI_Iallreduce()` produces almost no application speedup for any of the studied applications. Subfigure 5(b) shows roughly similar results, with only AMG and CTH showing over 10% speedup as process count

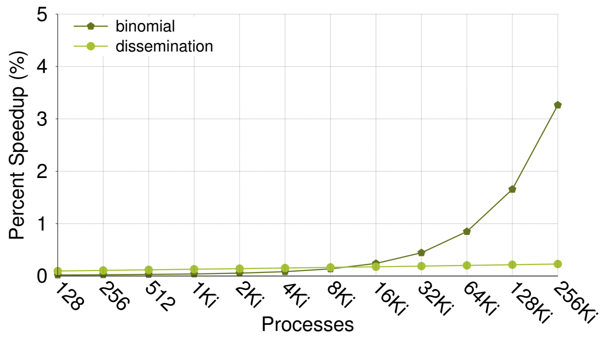


(a) System noise

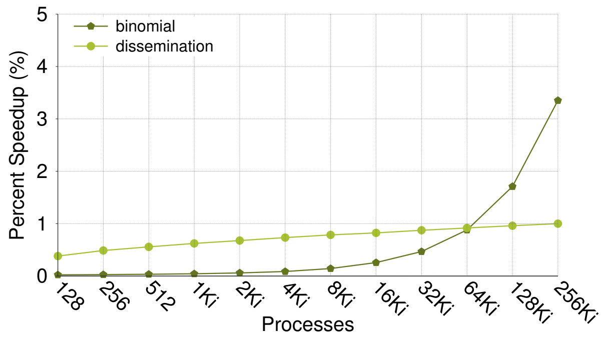


(b) Synthetic noise (10 Hz, 2.5 ms)

Figure 5: Observed application speedup achieved by switching from standard allreduce to an idealized nonblocking version in two low-noise environments.



(a) System noise



(b) Synthetic noise (10 Hz, 2.5 ms)

Figure 6: Observed microbenchmark speedup achieved by using idealized nonblocking versions of different collectives in low-noise environments.

increases. The behavior of AMG in these two noise regimes is largely explained by its unique (among this set of applications) pattern of collective inter-arrival times [9]. These results indicate that `MPI_Iallreduce()` will not provide much mitigation effect of system noise to applications, even for heavy allreduce users (as are HPCCG and MiniFE, for example). While it is dangerous to extrapolate this conclusion to other potential nonblocking collective implementations, these results suggest that whatever other benefits they may have for applications, mitigation of system noise is not among them.

3.4 Microbenchmark analysis

The results of our microbenchmark simulations in the low-noise environments appear in Figure 6. In these experiments, nonblocking versions of dissemination-based collectives improve completion time only slightly, even as scale increases. In contrast, nonblocking versions of binomial algorithm collectives demonstrate strongly increasing speedup as the size of the application grows past 16Ki processes. This behavior can be explained by considering the effect of messaging in each algorithm at the larger scales. In the binomial-tree algorithms, a root node is responsible for sending all the data required by its entire subtree; at larger process counts, even with small messages (8 bytes in these experiments), the time required to send these data amounts becomes noticeable. Our idealized simulation of nonblocking collectives assumes that this message send time will be completely overlapped by application computation, and therefore will not contribute to the total runtime of the application. This result also roughly agrees with our application simulation results, where `MPI_Iallreduce()` does not significantly mitigate either system or daemon noise.

4 Asynchronous checkpointing noise

4.1 Modeling asynchronous checkpointing and nonblocking collectives

Asynchronous checkpointing has the potential to introduce noise events that dramatically increase the time required to complete a collective operation. By increasing the amount of time that application processes spend in communication, noise propagation effectively decreases the application's native ratio of computation-to-communication. When the impact of noise propagation is (or may be) significant, we can express the potential speedup that may be obtained by using non-blocking collectives to overlap communication and computation as:

$$\begin{aligned}
\text{speedup} &= \frac{T_s}{\hat{T}_s} \\
&= \frac{W_k + \sum_{i=1}^{k-1}(c_i + n_i + W_i)}{W_k + \sum\limits_{\substack{i=1 \\ c_i=collective \\ (c_i+n_i)\geq W_i}}^{k-1}(c_i+n_i) + \sum\limits_{\substack{i=1 \\ c_i=collective \\ (c_i+n_i)<W_i}}^{k-1} W_i + \sum\limits_{\substack{i=1 \\ c_i=p2p}}^{k-1}(c_i+n_i+W_i)} \\
&\leq \frac{\sum_{i=1}^{k} W_i + \sum_{i=1}^{k-1}(c_i+n_i)}{W_k + \sum\limits_{\substack{i=1 \\ c_i=collective \\ (c_i+n_i)\geq W_i}}^{k-1}(c_i+n_i) + \sum\limits_{\substack{i=1 \\ c_i=collective \\ (c_i+n_i)<W_i}}^{k-1}(c_i+n_i) + \sum\limits_{\substack{i=1 \\ c_i=p2p}}^{k-1}(c_i+n_i+W_i)} \\
&\leq \frac{\sum_{i=1}^{k} W_i + \sum_{i=1}^{k-1}(c_i+n_i)}{\sum_{i=1}^{k-1}(c_i+n_i)}
\end{aligned}$$

$$= 1 + \frac{\beta \sum_{i=1}^{k-1} c_i}{\sum_{i=1}^{k-1}(c_i + n_i)} \tag{6}$$

A similar derivation yields:

$$\text{speedup} = \frac{T_s}{\hat{T_s}}$$
$$\leq 1 + \frac{\sum_{i=1}^{k-1}(c_i + n_i)}{\sum_{i=1}^{k} W_i}$$
$$= 1 + \frac{\sum_{i=1}^{k-1}(c_i + n_i)}{\beta \sum_{i=1}^{k-1} c_i} \tag{7}$$

Equations 6 and 7 establish upper bounds on the achievable speedup. The effective upper bound on speedup will be given by the smaller of the two. Equation 6 will be a tighter upper bound when the application's computation-to-communication ratio is smaller than the degree to which time spent in communication is inflated by noise propagation. More precisely, Equation 6 is a tighter upper bound on application speedup when:

$$\beta \leq \frac{\sum_{i=1}^{k-1}(c_i + n_i)}{\sum_{i=1}^{k-1} c_i}$$

Conversely, Equation 7 will be the tighter bound when:

$$\beta > \frac{\sum_{i=1}^{k-1}(c_i + n_i)}{\sum_{i=1}^{k-1} c_i}$$

The maximum value of the upper bound on speedup is two. It reaches this value when the computation-to-communication ratio is exactly equal to the factor by which noise propagation inflates the application's

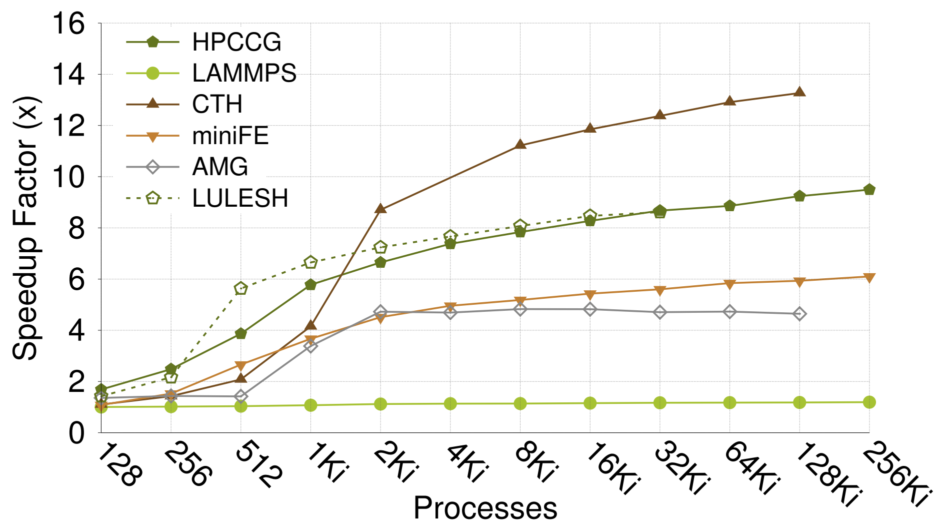


Figure 7: Speedup results for asynchronous checkpointing noise (0.5 Hz, 1s)

communication time. In other words, nonblocking collectives can reduce the application's time-to-solution by as much as a factor of two when the application's computation and communication (including noise propagation) exactly overlap.

These upper bounds show that when noise propagation is modest, the benefits of nonblocking collectives will be limited by the application's computation-to-communication ratio (β). This is consisent with what we observed in Section 3. However, as the application's execution begins to be dominated by noise propagation, the opportunities to overlap communication (and noise-induced delays) with computation are eventually exhausted and the application is unable to absorb additional noise.

As our model shows, the performance benefit of nonblocking collectives is potentially significant. However, for applications that spend a large fraction of their execution on point-to-point operations and their associated work intervals, this bound is not particularly tight and the benefits will be more modest. In practice, the benefit of non-blocking collectives will depend on the target application's computation/communication pattern and the extent to which it can allow collective communication operations and computation to overlap.

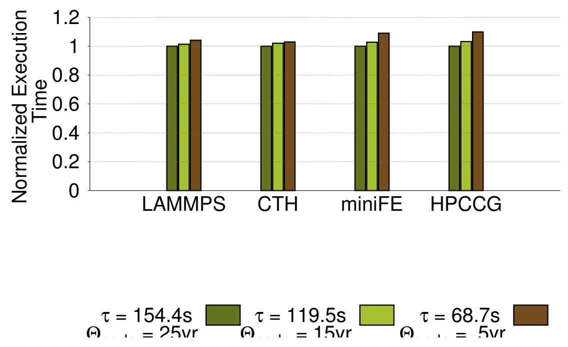
4.2 Application simulation results

Figure 7 shows application speedup when we simulate applications using an idealized `MPI_Iallreduce()` in the presence of asynchronous checkpointing noise. The noise characteristics we use here reflect an application using asynchronous checkpointing where each process takes a checkpoint every 120 seconds (locally timed as the checkpoint operation is not coordinated across nodes), writes 2 GiB of data per checkpoint, and has a data path to stable storage with 2 GiB/second write bandwidth. This configuration results in a checkpoint frequency of 0.5 Hz and a checkpoint duration of 1 second. These checkpoint interval and checkpoint size figures are typical of many production workloads, and the write bandwidth reflects the planned incorporation of solid-state storage on a per-node basis in exascale machines currently being designed by DOE and other organizations.

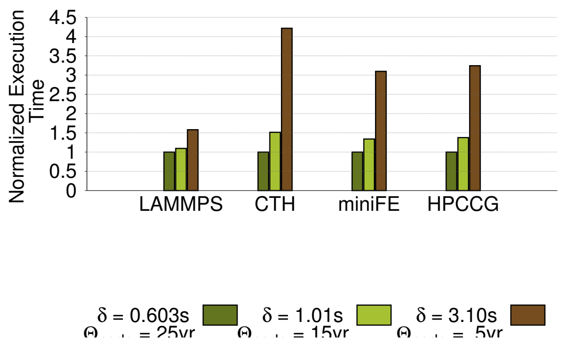
Using an idealized `MPI_Iallreduce()` allows us to consider the case where collectives are allowed to overlap with other communication operations. The results shown in Figure 7 imply that allowing more liberal overlap of computation and communication can lead to significant speedup for most of the applications we considered. However, exploiting this difference in behavior will likely require radical changes to the computational structure of the applications.

Figure 7 also shows a markedly different result than we observed for the low noise environment. This difference reflects an important point about the nature of noise, on which we digress briefly. Asynchronous checkpointing noise introduces detours with very long durations compared to OS or daemon noise — on the order of seconds, as opposed to the millisecond- and microsecond-scale durations of OS/daemon noise. Prior research on OS noise [8] has shown that the duration of noise events has a larger impact on application performance than the frequency of those events. In the context of checkpointing, this suggests that checkpoint commit time, rather than checkpoint interval, has the larger impact; this is confirmed by experimental results [9] shown in Figure 8. We vary each of checkpoint interval (Figure 8(a)) and checkpoint commit time (Figure 8(b)), holding the other constant in each case to satisfy Daly's optimality equation [5]. In both cases, we measured simulated application completion time for a subset of the applications we consider in this paper. These figures show that changes to the checkpoint interval have very little impact on application performance, while increasing checkpoint commit time can significantly increase time-to-solution.

Returning to our discussion of the potential impact of `MPI_Iallreduce()`, Figure 7 shows that many of our applications could possibly benefit significantly from its ability to mitigate the effect of asynchronous checkpoint noise. HPCCG and miniFE both show large speedup factors, as does CTH. The outlier here is LAMMPS, which benefits little from this improvement because of its infrequent use of `MPI_Allreduce()`. This result suggests that other nonblocking collectives may show similar benefits in applications using asynchronous checkpointing.



(a) Frequency varies, duration fixed



(b) Duration varies, frequency fixed

Figure 8: Effects on execution time of LAMMPS, CTH, miniFE, and HPCCG observed when varying checkpoint frequency (τ) vs. checkpoint duration (δ), for different node MBTF (Θ_{node}) values.

4.3 Microbenchmark analysis

The results of our microbenchmark simulations in a high-noise environment appear in Figure 9. In contrast to the low-noise microbenchmark result, here the completion times of our simulated dissemination-based collectives (e.g., `MPI_Iallreduce()` and `MPI_Iallgather()`) are sped up by a noticeable factor. The completion times of simulated binomial-tree collectives (e.g., `MPI_Igather()` and `MPI_Iscatter()`) are barely affected, only beginning to register a performance benefit as scale increases past 128Ki processes.

One of the principal reasons for the stark performance differences in Figures 7 and 9 is the difference in the dependencies that are created for collectives based on the dissemination algorithm and the dependencies that are that are created for collectives based on a binomial tree. Figure 10 shows an example of the dependencies that are created for each algorithm. The dependencies shown for the binomial tree correspond to collectives that use the tree for distribution (e.g., `MPI_Bcast()`, `MPI_Scatter()`). The most obvious difference is in the sheer number of dependencies that are created. The binomial tree algorithm creates $(n - 1)$ direct dependencies, whereas the dissemination algorithm creates $(n \log n)$ direct dependencies. For collectives running on extremely large systems, the difference between the number of dependencies created by these two algorithms will be stark. For example, for a collective with 32Ki participants, the dissemination algorithm will generate more than 15 times as many direct dependencies. The nature of the dependencies that are created is also important. As shown in Figure 10(a), the dissemination algorithm yields bidirectional dependencies between every pair of nodes. For example, in round 1, n_1 is dependent on n_0. In round 3, n_0 is (indirectly) dependent on n_1. As a result, the total number of (direct and indirect) dependencies created by the dissemination algorithm is $O(n^2)$. These dependencies allow noise events to propagate in the system; if any of the participants in a collective operation is delayed by a noise event, it has the potential to delay all of the participants. In contrast, the total number of dependencies are much

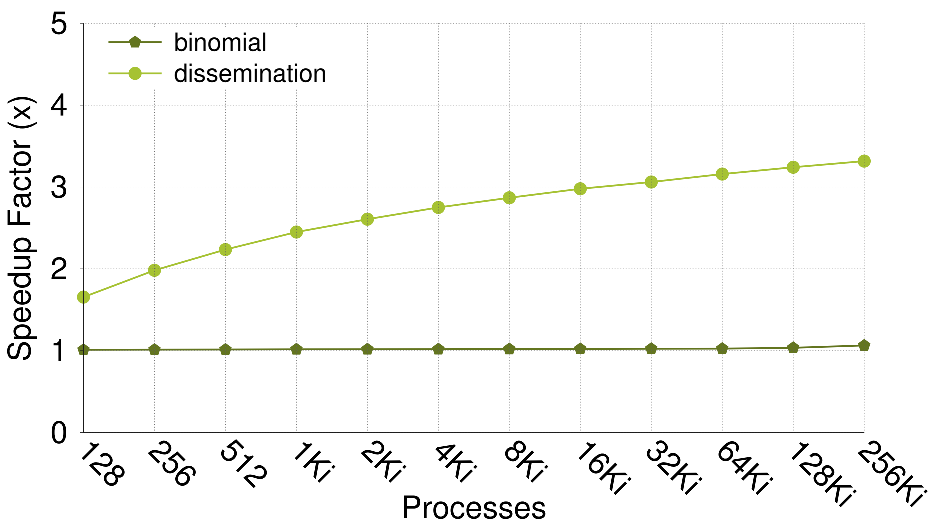


Figure 9: Observed microbenchmark speedup achieved by using idealized nonblocking versions of different collectives in a high-noise regime.

smaller in the binomial tree algorithm $O(n \log n)$; when the binomial tree is used for distribution, dependencies only extend to ancestors in the tree. For example, all of the nodes are dependent on n_0 but n_0 has no dependencies.

The significance of these dependencies emerges when we consider nonblocking collectives. The idealized nonblocking collectives that we use in the simulator allow the application to make progress even when the collective operation is delayed by the propagation of noise events. As a result, we see that nonblocking collectives yield significant speedups in Figures 7 and 9. In contrast, the limited dependencies in

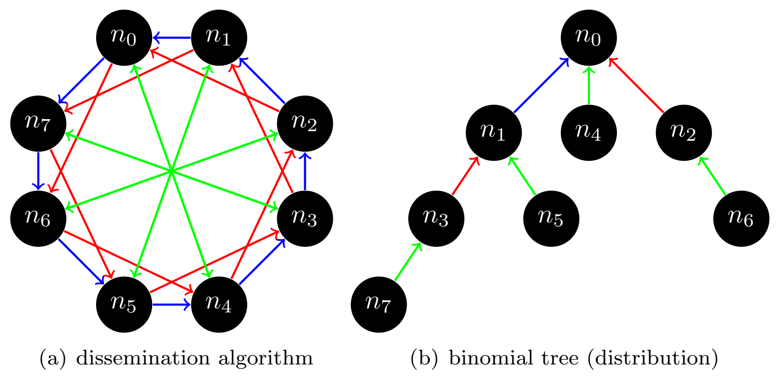
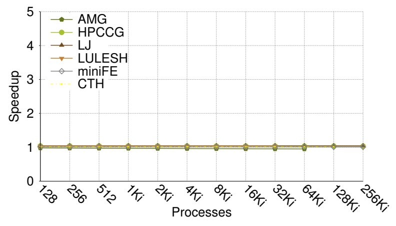
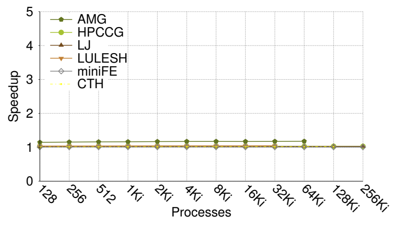


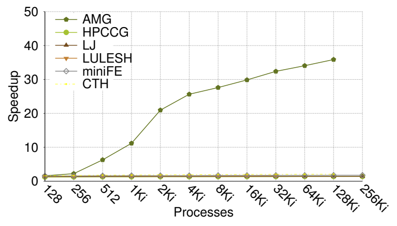
Figure 10: The round-by-round communication dependencies between eight nodes participating in collectives based on: (a) the dissemination algorithm; and (b) the binomial-tree algorithm for distribution (e.g., `MPI_Bcast()`, `MPI_Scatter()`). Dependencies that arise in round 1 are shown in blue; dependencies that arise in rounds 2 and 3 are shown in red and green, respectively. The arrows point in the direction of the dependencies. For example, in round 1, n_1 is dependent on n_0 for both algorithms. This figure illustrates the stark difference in the nature of the dependencies that are generated by the two algorithms.



(a) OS noise



(b) OS Daemon



(c) Checkpoint

Figure 11: Comparison of our idealized non-blocking collectives to a scenario with no injected noise. For each noise type, a speedup of 10 means that an application encountering no noise events completes 10 times faster than it does using our idealized non-blocking collectives in the presence of noise.

binomial-tree-based collectives are less sensitive to participant delay and therefore benefit much less from nonblocking collectives.

Comparison of non-blocking `MPI_Allreduce()` to no injected noise. Lastly, we compare the performance of our idealized non-blocking allreduce to a scenario with no injected noise. We use the speedup metric described previously. In this case, speedup tells us how effective a non-blocking `MPI_Allreduce()` is at mitigating all noise impact on our applications. Figure 11 shows the speedup results for the three injected noise types. From this figure we see that for nearly all the applications except for AMG, `MPI_Iallreduce()` mitigates all noise impact. AMG's behavior can be explained in light of Figure 2; a significant portion of AMG's communication time is spent in another dissemination-based collective, `MPI_Allgather()`, which as we have discussed is subject to significant noise impact. This result underscores the potential performance benefit of an `MPI_Iallreduce()` in noisy environments.

5 Related work

Techniques for improving the overlap of communication and computation in MPI applications have been thoroughly explored over the past decade. These techniques include offload [2] and nonblocking collectives. Based on results showing the potential performance benefit of nonblocking collectives [20], Hoefler et al. have argued for the inclusion of nonblocking collectives in the MPI standard [19]. They have also developed a library that implements nonblocking versions of all of the MPI collectives and characterized its impact on application performance [15]. Although the early work only briefly discusses system noise, Hoefler et al. subsequently made the connection between nonblocking collective operations and OS noise explicit [16].

Despite this body of research, our work is the first to examine the relationship between OS noise characteristics and nonblocking collectives. We are also the first to explore how nonblocking collectives could be used to improve the performance of emerging resilience techniques (e.g., uncoordinated checkpointing).

6 Conclusion

Nonblocking collective operations are of increasing interest, not only as enablers of continued application scalability but also as potential mitigators of OS noise effects. We have in this paper used a simulation approach to investigate how applications using an idealized nonblocking allreduce operation might respond in the presence of different noise environments, at scales likely to be encountered in future extreme scale systems. Our results indicate that, for noise caused by operating system activity, `MPI_Iallreduce()` is unlikely to provide much benefit. We suggest that, by themselves, nonblocking collective operations of other types should not be automatically assumed to be able to mitigate such noise effects. However, we have also shown that the effect of other noise types, such as that caused by checkpoint/restart activity, might be usefully reduced by the introduction of nonblocking collectives. We have also demonstrated that the algorithm used to implement a nonblocking collective may have a non-trivial effect on its performance benefit. Application designers should consider carefully the characteristics of all the noise sources on their target systems to decide whether refactoring their code to take advantage of nonblocking collectives will be worthwhile.

References

[1] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, Susan Coghlan, and Aroon Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16, 2008.

[2] Ron Brightwell, Rolf Riesen, and Keith D Underwood. Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. *Intl. Journal of High Performance Computing Applications*, 19(2):103–117, 2005.

[3] Greg Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *Proceedings of the 7th annual IEEE/ACM Intl. Symposium on Code Generation and Optimization*, pages 1–12. IEEE Computer Society, 2009.

[4] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.

[5] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.

[6] Jr. 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.

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

[8] Kurt B Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, page 19. IEEE Press, 2008.

[9] Kurt B. Ferreira, Scott Levy, Patrick M. Widener, Dorian Arnold, and Torsten Hoefler. Understanding the effects of communication and coordination on checkpointing at scale. In *Proc. Inernational Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, New Orleans, Louisiana, November 2014. IEEE/ACM.

[10] P Ghysels and W Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 2013.

[11] M. Giampapa, T. Gooding, T. Inglett, and R.W. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *High Performance Computing, Networking,*

Storage and Analysis (SC), 2010 International Conference for, pages 1–10, Nov 2010.

[12] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. Scientific and Engineering Computation. MIT Press, 2014.

[13] Van Emden Henson and Ulrike Meier Yang. Boomeramg: A parallel algebraic multigrid solver and preconditioner. *Appl. Num. Math.*, 41:155–177, 2002.

[14] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratory, 2009.

[15] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proc. of the 2007 Intl. Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[16] 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 (SC'10)*, Nov. 2010.

[17] T. Hoefler, T. Schneider, and A. Lumsdaine. LogGOPSim - Simulating large-scale applications in the loggops model. In *Proc. of the 19th ACM International Symp. on High Performance Distributed Computing*, pages 597–604. ACM, Jun. 2010.

[18] T. Hoefler, C. Siebert, and A. Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. In *Proc. of the 2010 ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*, pages 159–168. ACM, Jan. 2010.

[19] T. Hoefler, J. Squyres, G. Bosilca, G. Fagg, A. Lumsdaine, and W. Rehm. Non-blocking collective operations for MPI-2. Technical report, Open Systems Lab, Indiana University, Aug. 2006.

[20] Torsten Hoefler, Prabhanjan Kambadur, Richard L Graham, Galen Shipman, and Andrew Lumsdaine. A case for standard non-blocking collective operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 125–134. Springer, 2007.

[21] Ian Karlin, Abhinav Bhatele, Bradford L. Chamberlain, Jonathan Cohen, Zachary Devito, Maya Gokhale, Riyaz Haque, Rich Hornung, Jeff Keasler, Dan Laney, Edward Luke, Scott Lloyd, Jim McGraw, Rob Neely, David Richards, Martin Schulz, Charle H. Still, Felix Wang, and Daniel Wong. LULESH programming model and performance ports overview. Technical Report LLNL-TR-608824, December 2012.

[22] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[23] Scott Levy, Bryan Topp, Kurt B. Ferreira, Dorian Arnold, Torsten Hoefler, and Patrick Widener. Using simulation to evaluate the performance of resilience strategies at scale. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2013 SC Companion:*. IEEE, 2013.

[24] Message Passing Interface Forum. MPI: A message-passing interface standard, version 3.0. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, September 2012.

[25] Fabrizio Petrini, DK Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 55–55. IEEE, 2003.

[26] Steven J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal Computation Physics*, 117:1–19, 1995.

[27] Rolf Riesen, Ron Brightwell, Patrick Bridges, Trammell Hudson, Arthur Maccabe, Patrick Widener, and Kurt Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience*, 21(6):793–817, April 2009.

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

[29] Jeffrey Vetter and Chris Chambreau. mpiP: Lightweight, scalable MPI profiling. *URL: http://www. llnl. gov/CASC/mpiP*, 2005.

[30] Kazutomo Yoshii, Kamil Iskra, Harish Naik, Pete Beckman, and P. Chris Broekema. Characterizing the performance of "big memory" on Blue Gene Linux. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 65–72. IEEE, 2009.