Characterizing the Influence of System Noise on Large-Scale Applications by Simulation

Torsten Hoefler
University of Illinois at Urbana-Champaign
Urbana IL 61801, USA
htor@illinois.edu

Timo Schneider and Andrew Lumsdaine
Indiana University
Bloomington IN 47405, USA
{timoschn,lums}@cs.indiana.edu

***Abstract*—This paper presents an in-depth analysis of the impact of system noise on large-scale parallel application performance in realistic settings. Our analytical model shows that not only collective operations but also point-to-point communications influence the application's sensitivity to noise. We present a simulation toolchain that injects noise delays from traces gathered on common large-scale architectures into a LogGPS simulation and allows new insights into the scaling of applications in noisy environments. We investigate collective operations with up to 1 million processes and three applications (Sweep3D, AMG, and POP) with up to 32,000 processes. We show that the scale at which noise becomes a bottleneck is system-specific and depends on the structure of the noise. Simulations with different network speeds show that a 10x faster network does not improve application scalability. We quantify noise and conclude that our tools can be utilized to tune the noise signatures of a specific system.**

I. Motivation and Background

The performance impact of operating system and architectural overheads (*system noise*) at massive scale is increasingly of concern. Even small local delays on compute nodes, which can be caused by interrupts, operating system daemons, or even cache or page misses, can affect global application performance significantly [1]. Such local delays often cause less than 1% overhead per process but severe performance losses can occur if noise is propagated (*amplified*) through communication or global synchronization. Previous analyses generally assume that the performance impact of system noise grows at scale and Tsafrir et al. [2] even suggest that the impact of very low frequency noise scales linearly with the system size.

A. Related Work

Petrini, Kerbyson, and Pakin [1] report that the parallel performance of SAGE on a fixed number of ASCI Q nodes was highest when SAGE used only three of the four CPUs per node. It turned out that "resonance" between the application's collective communication and the misconfigured system caused delays during each iteration. Jones, Brenner, and Fier [3] observed similar effects with collective communication and also report that, under certain circumstances, it is beneficial to leave one CPU idle. A theoretical analysis of the influence of noise on collective communication [4] suggests that the impact of noise depends on the type of distribution and their parameters and can, in the worst case (exponential distribution), scale linearly with the number of processes. Ferreira, Bridges, and Brightwell use noise-injection techniques to assess the impact of noise on several applications [5]. Beckman et al. [6] analyzed the performance on BlueGene/L, concluding that most sources of noise can be avoided in very specialized systems.

Previous work was either limited to experimental analysis on specific architectures with injection of artificially generated noise (fixed frequency), or to purely theoretical analyses that assume a particular collective pattern [4]. These previous results show the severity of the problem but allow little generalization and provide limited insight into application behavior on real machines. Effects, such as absorption of noise, are described but not further investigated [5]. One common theme in all previous works is to look at collective communications as the main problem and often model such operations as strictly synchronizing opaque entities. However, the Message Passing Interface (MPI) standard [7] states that *"[...] a collective communication call may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier function."* This invalidates all simple models in use today. The synchronization properties of an application depend on the collective algorithm, point-to-point messaging, and the system's network parameters.

We chose a simulation approach similar to Sottile et al.'s [8] and improve it by using noise traces from existing systems combined with detailed simulation and extrapolation of collective operations and parallel application traces. Our simulator enables us to simulate applications on HPC systems that cannot be accessed easily at full scale or that do not exist yet and it also allows us to investigate the effect of changing network speeds and other system parameters.

B. Contributions

In this work, we introduce an open-source measurement and simulation framework that measures OS noise and assesses its impact on large-scale applications by simulation. We build upon a detailed model for dependencies in applications and synchronization (cf. Lamport's happens-before relation) that considers collective as well as point-to-point patterns of real applications. We perform simulations for a set of applications and systems, explain phenomena observed by

SC10 November 2010, New Orleans, Louisiana, USA 978-1- 4244-7558-2/10/$26.00

other researchers, and show how the performance of collective operations and applications is decreased by noise.

Such techniques are very helpful in evaluating system software and parallel applications targeting to run on upcoming Peta- and Exascale computers that are not yet available. For example, asynchronous (threaded) progression or active messages might be needed for programming such extreme-scale systems but could increase noise on those systems. The new approaches to implement parallel applications or new concepts for the system software that will be needed to achieve reasonable performance at this scale can be analyzed with our toolchain.

Our approach combines theory and practice in that we use a detailed network model to describe all synchronization at the message level and analyze the global impact of node-local system noise. The key contributions of this work are several:

- A detailed dependency and synchronization model for noise propagation and absorption in parallel applications.
- A discrete-event simulation strategy to investigate the impact of real-world and artificial OS noise on real applications. The strategy accurately reproduces previous experimental results and provides significant insight into previous observations.
- Simulation results of up to 1 million processes using real-world noise traces. The results show that the influence of real-world noise on collective communication can be very different from that of artificially generated noise.
- Simulation results showing that point-to-point messaging influences the noise sensitivity of applications.
- Simulation results including the effects of co-scheduling and network speed in the context of noise.
- A quantification of an effect that we call *noise bottleneck* where increasing the network speeds does not improve application performance due to noise.

In the next section, we discuss established measurement techniques for system noise and present an enhanced noise benchmark. In Section III, we model the synchronization properties of point-to-point messages and collective operations. Then, we introduce the established LogGPS model to simulate the behavior of collective operations under the influence of noise in Section IV. In Section V, we simulate complete applications with our collected noise traces from real systems.

Our methodology provides important insight into the effects of noise on parallel applications and enables us to explain various phenomena found in previous studies. For example, our model shows that the impact of noise depends on the type of collective operation (we are able to explain Ferreira, Bridges, and Brightwell's finding that broadcast is significantly less sensitive to noise than allreduce [5]). Our model also explains why small-message collectives are more affected by noise than larger ones and why low-frequency noise with higher amplitude degrades the performance significantly while high-frequency noise has nearly no impact.

II. Measuring System Noise

A straightforward noise measurement technique, called fixed work quantum (FWQ), measures the time t_i to compute several fixed workloads. FWQ assumes that the minimum time t_{min} represents the noiseless execution and all other times t_i are perturbed by $t_i - t_{min}$. The main problem with this approach is that the sampling frequency is not constant because each perturbation influences the start of the next sample. Sottile and Minnich [9] propose an inverse measurement technique, fixed time quantum (FTQ), which counts the number of fixed-work computations that can be performed in a specific time and thus enables the application of techniques from signal analysis.

But we argue that both methods fail to record noise with high frequency because the fixed workload needs t_{min} to compute. This effectively means that the sampling frequency is limited to $1/t_{min}$ and all noise that has a higher frequency simply elevates t_{min} and underestimates noise. Additionally, if the Nyquist-Shannon sampling theorem is not satisfied , then aliasing could lead to wrong (Moiré) observations. Thus, we conclude that to capture all noise frequencies accurately, the workload has to be chosen as small as possible ($t_{min} \rightarrow 0$). For our experiments, we choose a FWQ benchmark with a workload close to zero.[1] To manage the huge number of measurements, we only store the time of the perturbed measurements similar to Beckman's "selfish detour" benchmark [6] which also uses a tight loop to measure perturbations.

The first difference from "selfish detour" is that we define the threshold relative to t_{min} (instead of a fixed threshold) and thus allow the benchmark to run on a wide variety of systems. We used $9 \cdot t_{min}$ as threshold to filter cache misses that are caused by recording the data. Such cache misses occurred on all systems and caused a detour between $6 \cdot t_{min}$ (Opteron) and $8 \cdot t_{min}$ (BlueGene/P). Another difference is that we assess t_{min} in a separate step such that we do not need to update t_{min} in the benchmark loop. This removes one of the three branches in the critical loop and increased the sampling frequency (benchmark resolution) by approximately 30% in our tests. We measured all times with architecture-dependent high-resolution timers (RDTSC on x86, MFTB on PowerPC, AR.ITC on IA64). All benchmarks are implemented in the publicly available tool Netgauge [10][2].

A. Analyzing Real-World Architectures

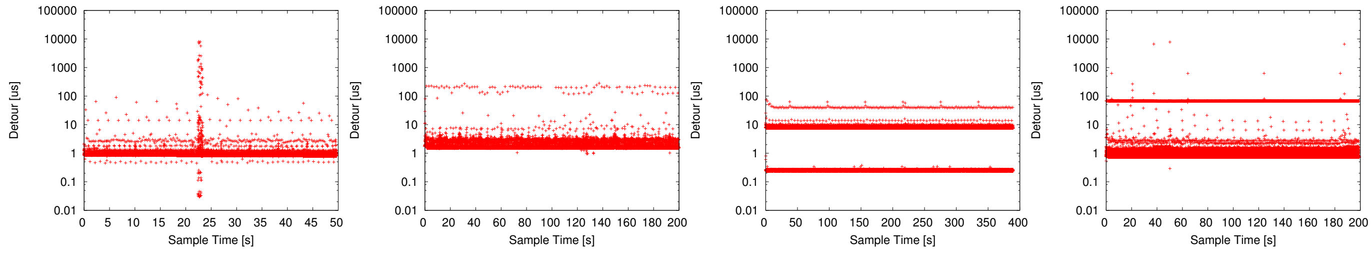
We analyze four different systems that represent today's common large-scale system architectures, often scaling to tens or even hundreds of thousands of processing cores. The first system represents Linux clusters with InfiniBand such as the Ranger system at TACC that run a default Linux kernel. The other three systems represent specialized machines with custom operating system kernels: SGI Altix 4700, Cray XT-4, and BlueGene/P. For our benchmarks, we used the standard batch mechanisms without special tuning to run our jobs (as

[1] We use a tight loop and t_{min} denotes loop overhead.

[2] `http://www.unixer.de/research/netgauge/osnoise`

System/Architecture	R_{peak}	**OS**	t_{min}	1 ppn	c ppn
CHiC Cluster, diskless, 2152 Opteron 2.6 GHz cores	11.2 TFlop/s	Linux 2.6.18	3.74 ns	0.26%	0.21%
SGI Altix 4700, 2048 Itanium II 1.6 GHz cores	13.1 TFlop/s	Linux 2.6.16	25.1 ns	0.05%	n/a
Jugene, BlueGene/P, 295k PPC 450 850 MHz cores	825.5 TFlop/s	CNK 2.6.19.2	29.4 ns	0%	0%
Intrepid, BlueGene/P, 164k PPC 450 cores	458.6 TFlop/s	ZeptoOS 2.6.19.2	29.12 ns	0.02%	0.08%
Jaguar, Cray XT-4, 150k Opteron 2.1 GHz cores	1.38 PFlop/s	Linux 2.6.16 CNL	32.9 ns	0.02%	0.02%

TABLE I
SYSTEM PARAMETERS AND SERIAL NOISE OVERHEAD FOR ALL INVESTIGATED MACHINES WITH EITHER ONE CORE OR ALL CORES PER NODE USED. t_{min} REPRESENTS THE MINIMUM LOOP TIME (MEASUREMENT ACCURACY).



(a) CHiC (Linux) (b) SGI Altix (c) ZeptoOS (d) Jaguar

Fig. 1. Scatter plot of 10^5 detours on the different machines using all cores per node.

a typical user would do). We ran the noise benchmark three times on each system, recording 10^5 events, and chose the result with the lowest total detour (all runs differed by less than 0.1%). On systems with c processing cores per node, we executed our benchmark with 1 process per node (ppn) as well as with c ppn.

Table I shows the configuration of all test systems. Overheads of the sampling loop and timer access limit the sampling frequency on real systems, even if the workload is zero. Such loop overheads are system dependent and vary between a few CPU cycles (3.74 ns) and 32.9 ns, as listed in Table I. As discussed before, we cannot reliably measure noise frequencies higher than $\frac{1}{2t_{min}}$ Hz (134 MHz on our most accurate system). However, we assume that this limit is only of theoretical interest because most noise has a much lower frequency. In the following, we use the configuration where all cores on a node are used by the application (as most parallel codes are executed). Figure 1 shows scatter plots of the noise patterns for some of our investigated systems.

Figure 1(a) shows the diskless CHiC system with low regular noise but reproducible longer interruptions (as seen around 23 seconds in the plot).[3] Figure 1(b) shows that most detours on the SGI Altix lie in the 1–8 μs range while 220 μs interruptions occur approximately every 2 seconds. We measured absolutely no system noise on the BlueGene/P system running CNK (the benchmark ran for several hours and did not collect a single detour). This is consistent with the results by Yoshii et al. [11] who also report CNK as absolutely noiseless. ZeptoOS on BlueGene/P, however, causes low noise in a regular pattern as shown in Figure 1(c). The XT-4 part of the Jaguar system, number one in the current top-500 list (06/10), also shows high and infrequent random detours in addition to two baselines.

[3]Other investigated large Linux systems (e.g., Ranger and Juropa) show show structurally similar noise patterns and are omitted for brevity.

III. AN ANALYTICAL MODEL FOR NOISE PROPAGATION

Now, we present a suitable model to analyze noise effects on applications. Noise (or "detours" as discussed in the previous section) can either be absorbed or propagated by synchronization. Processes are often synchronized implicitly by remote data dependencies (cf. happens-before relation). For example, a receive cannot finish before the corresponding (matching) send has been posted and the network transmission cost has been paid (recv/send dependency). We analyze those effects in detail by utilizing the LogGOPS network model to characterize all situations where noise is transported or absorbed.

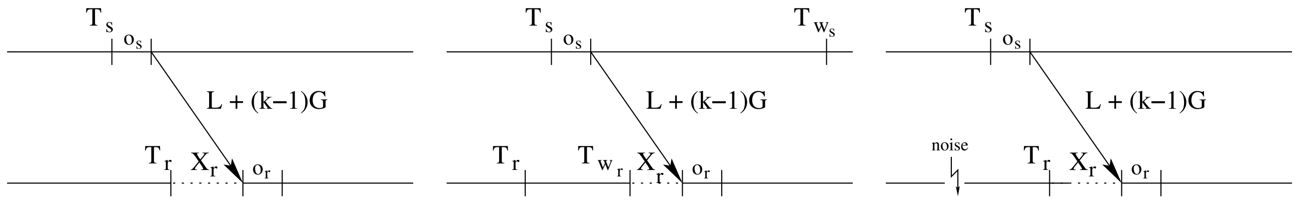
A. *The LogGOPS Model*

The LogGPS model [12] is a member of the LogP model family. LogP models are often used to model parallel applications and network transmissions.

Multiple researchers have shown that the LogP model family is able to model many parallel algorithms and architectures accurately (e.g., [13]). LogGPS additionally offers support for modeling the synchronization effects of rendezvous messages. We use the extended LogGOPS model in our simulation which includes an additional parameter O [14] that models the overhead per byte. Table II describes all parameters of the LogGOPS model briefly (see [12], [14] for details).

L	maximum latency between any two endpoints
o	CPU overhead, o_s for send and o_r for receive
g	inter-message gap, the minimum delay between two messages ($1/g \equiv$ message-rate)
G	gap per byte (1/G $\equiv$ bandwidth)
O	overhead per byte
P	number of communicating processes
S	threshold for *eager* messages that are buffered on the receiver. Messages larger than S block the sender in the *rendezvous protocol* until the receive has been posted, while messages smaller than S are sent immediately

TABLE II
LOGGOPS PARAMETERS



(a) Blocking transfer (b) Nonblocking transfer (c) Absorption

Fig. 2. Examples for blocking (a) and nonblocking (b) point-to-point synchronization and noise absorption (c).

The LogGOPS model ignores contention in the network and might thus underestimate communication costs. Our simulations are mostly targeted towards investigating noise propagation and one could see a congestion-free execution as showing the worst effect of noise. If necessary, average network contention can be modeled by increasing G in the LogGOPS model. We chose this approach to limit simulation resources and allow for larger process counts, cf. [14]. We discuss the influence of the network parameters to noise propagation and absorption in Section V-B.

In the following, we describe synchronization effects in parallel applications and derive an analytical model for noise propagation (sometimes also called amplification) and absorption. This model allows reasoning about the effects of noise and forms a base for our simulation. We discuss blocking and nonblocking point-to-point messages in detail before we proceed to more complex collective communication patterns.

B. Blocking Point-to-Point Communication

If the receive of a message is started too early, then the receiver must wait until the message is sent. Likewise, if the send of k bytes (and $k > S$, i.e., a rendezvous-send) is started too early then the sender must wait until the receiver is ready. Now let us discuss what "too early" means.

Figure 2(a) shows the scenario of a late sender and the associated synchronization overhead X_r. Both, the sender and receiver are assumed to start at time $t = 0$. T_s denotes the time when the send is started and T_r the time when the receive is posted. We assume that all other time is spent with computation that advances the algorithm (that is, no polling or testing). Let $N = o_s + L + (k-1)G + o_r$ denote the network overhead in the LogGP model. We define the synchronization overhead on the receiver as $X_r = \max\{T_s + N - T_r, 0\}$.

For $k > S$ (rendezvous protocol), the synchronization overhead X_s can also occur on the sender $X_s = \max\{T_r - N - T_s, 0\}$. In this case, the only scenario where neither sender nor the receiver are delayed is $T_r = T_s + N$, that is, the receive is posted exactly at the right moment. Such timing is very unlikely and blocking communication often propagates noise.

C. Nonblocking Point-to-Point Communication

A common method to avoid synchronization overheads and to reduce communication costs in general is nonblocking communication. Figure 2(b) shows the communication diagram for a nonblocking send/receive pair. Nonblocking transfers are split into two phases, the posting of the operation and the waiting for completion. Synchronization and data transfer can now happen *in the background* and the associated overheads can be hidden. However, nonblocking transfers underlie several restrictions and synchronization overheads can still occur if operations are *waited for* too early. Figure 2(b) shows an example where the receiver waits for an operation before the message arrives. The rendezvous-send and receive synchronization overheads are $X_r = \max\{T_s + N - T_{w_r}, 0\}$ and $X_s = \max\{T_r - N - T_{w_s}, 0\}$ respectively. The main difference from the blocking case is that, if the time between a send or receive and the respective wait is large enough, then synchronization can be avoided. Informally, no synchronization overhead occurs on the receiver, when the received data is needed *late enough*, that is, $T_{w_r} \geq T_s + N$. Synchronization overhead on the sender (rendezvous protocol) can be avoided if the send has *enough time to complete*, that is, $T_{w_s} \geq T_r - N$.

D. Noise Propagation and Absorption

As discussed before, system noise occurs locally at each process and usually has little impact on the process itself ($<$ 0.25%, cf. Table I). However, the synchronization described before can lead to noise propagation. But noise can also be consumed in existing synchronization delays and disappear completely (cf. [15]). Figure 2(c) shows an example where noise on the receiver is completely absorbed in X_r. However, if the system noise had happened at the same time on the sender, then noise would have been propagated and X_r would have been increased. Generally, only a limited amount of noise can be subsumed in a synchronization phase. We use σ_α to denote the noise that happens before time α.

If **blocking communication** is used, then σ_{T_s} propagates to the receiver but might be absorbed if the receive is posted *late enough*, that is, $T_r \geq T_s + \sigma_{T_s} + N$. The synchronization overheads (including noise propagation) on receiver and rendezvous-sender are $X_r = \max\{T_s + \sigma_{T_s} + N - T_r, 0\}$, and $X_s = \max\{T_r + \sigma_{T_r} - N - T_s, 0\}$, respectively. The condition for $X_r = X_s = 0$ (execution without synchronization overhead) is now $T_r = T_s + N + \frac{\sigma_{T_s} - \sigma_{T_r}}{2}$. If the detours on sender and receiver are identical, then $X_s = X_r = 0$ iff $T_r = T_s + N$ as in the noiseless case.

We expect applications that use **nonblocking communication** to be relatively resistant to system noise due to the possibility to hide some synchronization overheads. The synchronization overhead on receiver and rendezvous-sender can be modeled as $X_r = \max\{T_s + \sigma_{T_s} + N - T_{w_r}, 0\}\, and\, X_s = \max\{T_r + \sigma_{T_r} - N - T_{w_s}, 0\}$, respectively. We conclude that no synchronization overhead occurs on the receiver and all noise on the sender is absorbed if $T_{w_r} \geq T_s + \sigma_{T_s} + N$. The noise at the receiver can be absorbed on the sender (rendezvous protocol) if $T_{w_s} \geq T_r + \sigma_{T_r} - N$. Thus, nonblocking point-to-point communication has a higher potential to absorb noise than blocking communication.

E. Collective Operations

Collective operations often have more complex dataflow dependencies than point-to-point messages.We can, however, identify the following dependence classes in MPI:

1) **broadcast, scatter:** all non-root processes depend on the root process
2) **reduce, gather:** the root process depends on all non-root processes
3) **scan, exscan:** each process depends on all processes with a lower rank
4) **alltoall, allgather, allreduce, barrier, reduce_scatter:** each process depends on all other processes

Those semantic dependencies are lower bounds for synchronization and noise propagation, which means for example that an eager broadcast (at least) propagates all noise that happened on the root before the call (σ_{T_s}) to all other processes. This model assumes a linear implementation of the algorithm and would perform asymptotically worse than a binomial-tree implementation [runtime of $\Omega(P)$ vs. $\Omega(\log P)$]. Thus, at large scale, optimized algorithms must be used to implement collective operations. Such algorithms usually add recv/send (data) dependencies to the (minimal) semantic dependencies, which can cause additional noise propagation from intermediate processes. For example, the binomial tree shown in Figure 3 has multiple *paths* from the root node to the destinations and additional recv/send dependencies are

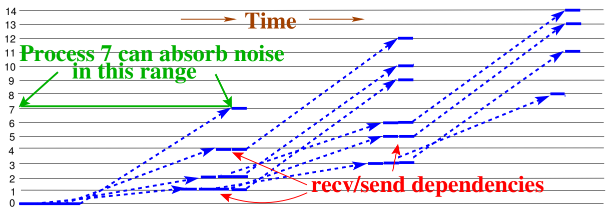


Fig. 3. LogGOP diagram for a binomial broadcast tree ($P = 15$).

introduced along each path. The longest paths in the 15-process example are $(0, 1, 3, 7)$, $(0, 2, 6, 14)$, $(0, 1, 5, 13)$, and $(0, 1, 3, 11)$ with four recv/send dependencies along each path. Each detour σ_{T_s} that precedes any send along these paths might delay all following processes. On the other hand, if all processes post the broadcast operation at the same global time, all but the root (process 0 in our example) can absorb some detour. Some processes (e.g., process 13) could even absorb three times as much as others. Also, if we take a detailed look at the longest paths, on all but the root node (e.g., processes 1, 2, or 3), noise that happens before the message is received is likely to be absorbed, and only detours during the short period between the receive and the send will delay the operation. Thus the binomial broadcast is relatively insensitive to noise.

The binomial-tree argument shows that the influence of noise and its propagation can, even for simple algorithms, not easily be assessed analytically. Even the globally dependent algorithms in the fourth category depend on the details of the underlying point-to-point algorithm. Figure 4 shows the LogGP diagram of two barrier operations with a compute

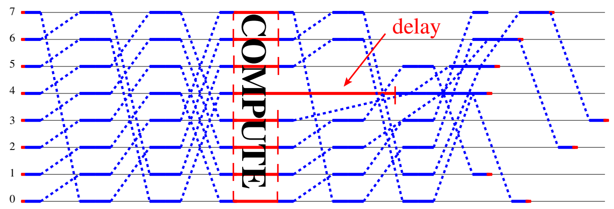


Fig. 4. LogGP diagram of two barriers with process 4 delayed ($P = 8$).

phase between them. We assume that the barrier is implemented with the dissemination algorithm and process 4 is delayed during the compute phase. All processes leave the second barrier at different times due to recv/send dependencies and process 3 is delayed most. This example shows clearly that current models, which model the collective operation as a black box (and assume that all processes are delayed in the same way, e.g., [2]) cannot be used to assess the effects of noise propagation accurately. An accurate analytical model has to account for the whole communication and synchronization of each send/receive pair and all recv/send dependencies to account for each noise propagation and absorption correctly. Finding such models for complex communication patterns seems infeasible. Thus, we propose a full LogGOPS simulator that enables accurate simulation of large-scale systems.

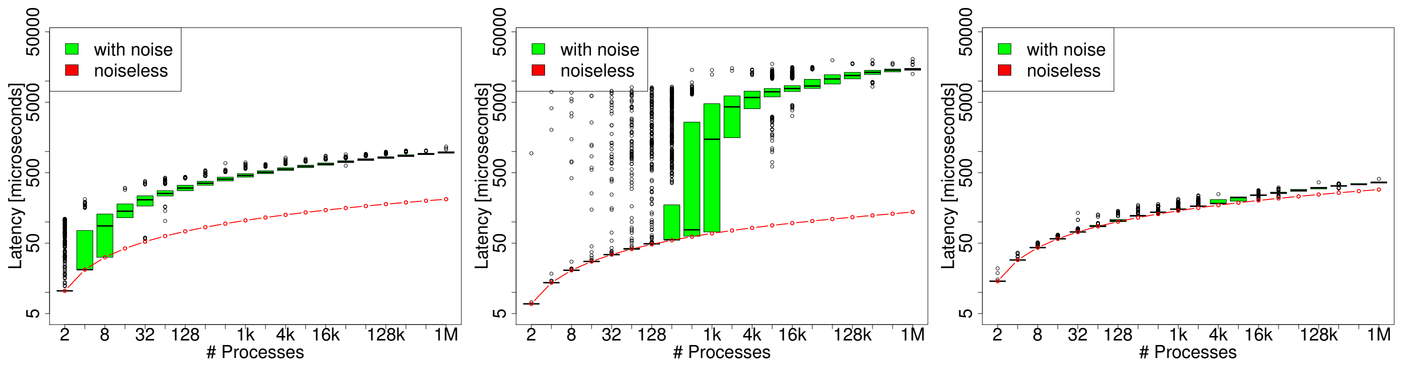
IV. LogGOPS Simulation Framework

The LogGOPS simulation toolchain consists of a trace collector, a schedule generator, an optimized LogGOPS discrete-event simulator similar to [16], and a visualizer.

The trace collector is a library that uses the MPI profiling interface [7, §14] in order to record all MPI calls of an application with minimal overhead.

The schedule generator reads the MPI traces and represents the control- and dataflow in our happens-before application. Collective operations are replaced with suitable point-to-point algorithms. The generator supports state-of-the-art collective algorithms, such as n-ary (binomial) trees, dissemination, recursive doubling, and pipelined trees. A mapping from collective operation to algorithm (e.g., allreduce $\mapsto$ binary tree reduce + binary tree broadcast, or barrier $\mapsto$ dissemination) can be specified in the schedule generation phase. In this work, we used the dissemination algorithm for small allreduce, allgather, alltoall, and barrier calls and the binomial tree algorithm for small scatter, gather, and broadcast calls.

The simulator reads the schedule, performs the full LogGOPS simulation (cf. Section III) and reports the end times for each process. The simulator was shown to predict collective operations up to 128 processes with an average error of less than 1% and full MPI applications with an error below 2%. A complete description of the simulator and a detailed performance and accuracy study is available in [14] and the



(a) 1,000 Hz, 100 μs (b) CHiC (c) ZeptoOS

Fig. 5. Boxplots for 1-byte dissemination-based collectives (e.g., allreduce) on different architectures.

whole open-source toolchain is available online[4].

In the following simulations, we used LogGP parameters that reflect each system accurately. The parameters are shown in Table III and were measured with the method described in [17] using the default MPI library and settings on each system and we assume $S = 65$ kiB and $O = 0$ (high overlap, see [14]).

System	L	o	g	G
CHiC	5.33 μs	0.77 μs	1.56 μs	0.00125 μs
Altix	1.64 μs	0.99 μs	0.90 μs	0.00123 μs
CNL	4.77 μs	2.87 μs	2.04 μs	0.00267 μs
ZeptoOS	8.67 μs	3.42 μs	2.81 μs	0.00269 μs
XT-4	9.9 μs	1.75 μs	3.40 μs	0.00058 μs

TABLE III
LogGP PARAMETERS FOR ALL MACHINES.

A. System Noise Input

The simulator supports the injection of system noise into all computations that are performed on the CPU: the application computation, the LogGOPS overheads o_s, o_r, and O, and the reductions in collective operations. We use the selfish detour traces that we gathered with Netgauge (cf. Section II) as input. Each trace contains 10^5 detour events and spans several minutes of benchmarking time. We also produced artificial (fixed frequency and detour) noise traces to reproduce previously published experiments. At startup, the simulator assigns a random time offset in the trace to each process. This process-specific time offset is increased during each local computation and all detours that occurred during the event are added to the simulation time.

B. Simulating Collective Operations

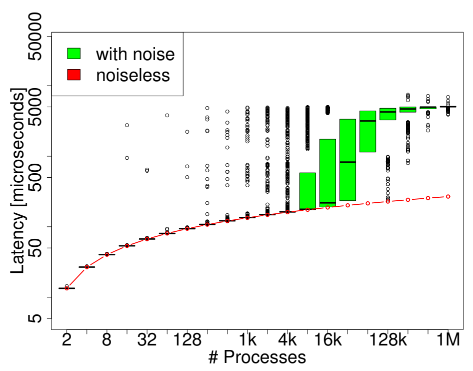
Our first experiment strives to reproduce the benchmark results presented by Beckman et al. [6] who used noise injection on BlueGene/L (BG/L) and investigated the behavior of barrier, allreduce and alltoall at large scale. While barrier is supported directly by the BG/L hardware, allreduce used a pattern similar to the dissemination pattern. We use LogGP parameters from BlueGene/P running CNL because we do not have access to a BlueGene/L. Thus, we expect the impact to be slightly lower, but asymptotically similar. Like Beckman et al., we used unsynchronized noise with a fixed frequency of $1{,}000, 100$, and 10 Hz causing detours of $16, 50, 100$, and 200 μs. We reproduced the two key observations: (1) The maximum slowdown of collective operations involving many processes scales linearly with the injected noise and (2) the maximum slowdown of the collective operation (in our case a factor of 13 on 32,768 processes where [6] reported a factor of 18) scales logarithmically with the number of processes. We also certified that a detour of 16 μs does not affect the operation significantly. In addition to that, we were able to analyze the statistical distribution of the latency for every single collective operation. Figure 5(a) shows a boxplot of the results of 1,000 simulations per process count for 100 μs noise with a frequency of 1,000 Hz (we remark that this is an extreme case of 10% noise overhead).

We chose boxplots [18] to present our results because, contrary to simpler line plots, they allow us to display the statistical effects that are crucial to understand the influence of noise. The boxes in the plot show the upper and lower quartile and small circles represent statistical outliers. The noise-less performance is plotted as a lowest (red) line in the diagrams. The first observation is alarming: The outliers at small process counts quickly become the median at large process counts and converge at a high level. This means that noise occasionally increases the latency for small runs but slows down large runs deterministically as one can see in Figure 5(a). This deterministic slowdown of single collective operations becomes a *bottleneck* that is relatively independent of the network parameters at large scale.

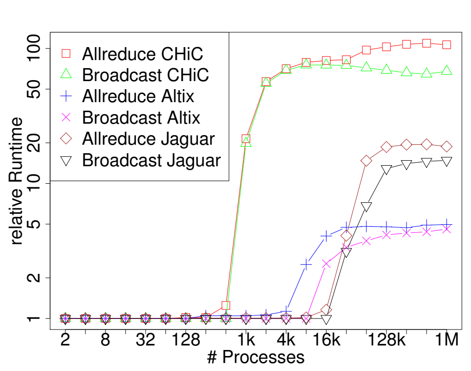
Our experiments with fixed noise also show that the (rooted) tree-based operations (e.g., broadcast and reduce) are less affected than (all-to-all) dissemination-based operations (e.g., barrier, allreduce, alltoall) as conjectured in [5]. However, the maximum detour for both types of operations scales logarithmically with the number of processes.

1) Simulation of Real-World Systems: Having shown that our simulation reproduces previous benchmark results with artificial noise well, we proceed to simulate the systems discussed in Section II-A. We remark that all previous studies only considered noise with fixed frequency and amplitude. In this experiment, we use our measured noise traces as input for the simulation. Our traces reflect detours on the real machines over several seconds. The results of 1,000 simulations per process of a dissemination-based algorithm (e.g., barrier or

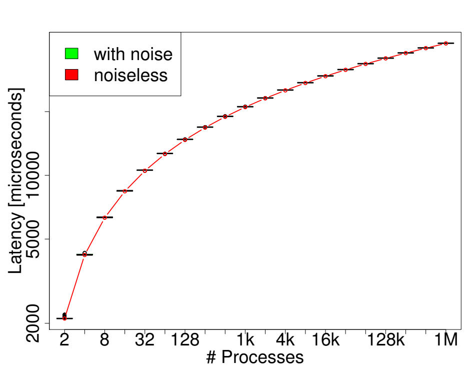
[4]http://www.unixer.de/LogGOPSim (2010)



(a) Jaguar



(b) Tree vs. dissemination.



(c) 1 MiB messages 1 kHz.

Fig. 6. Different experiments: (a) dissemination pattern on Jaguar (b) broadcast vs. allreduce, (c) large messages with fixed 1,000 Hz, $100\mu s$ noise

allreduce) are shown in Boxplots 5(b), 5(c), 6(a). Results of the Altix system show a similar shape and are omitted due to space constraints.

Despite a small number of outliers, noise has a very low influence on small-scale jobs on the CHiC and Jaguar systems. The most interesting observation is that each system seems to have a characteristic number of processes where the influence of noise rapidly raises to a very high level and stabilizes at a high latency. We predict that, due to this system-specific noise-bottleneck, applications will not scale well beyond 512, 4,096 and 8,192 processes on CHiC, Altix and Jaguar, respectively. ZeptoOS however, affects collective communications only slightly and does not exhibit the noise-bottleneck. The maximum slowdown at large scale (1 million processes) is 148.05, 6.45, 0.41 and 24.73 for CHiC, Altix, ZeptoOS and Jaguar, respectively.

2) Different Collective Operations: We simulated allreduce and broadcast with 2.5 ms detours at 10 Hz as benchmarked by Ferreira, Bridges, and Brightwell [5]. Ferreira et al. report a slowdown of 32 for allreduce and our simulation predicts a factor of 30. Our simulation also certifies that broadcast is significantly less influenced by noise than allreduce. However, we argue that artificial noise with a fixed frequency does not reflect real-world systems accurately. Figure 6(b) shows the median latency increase due to noise for 1,000 allreduce and broadcast simulations for our real-world noise traces. Again, we see the noise bottleneck, a clear inflection point at certain node counts. We also note that all systems seem to operate very smoothly with jobs smaller than 512 processes. We omitted the ZeptoOS system in the plot because collective are nearly not affected by its noise, even though it has, with 0.08%, a much higher serial noise than Jaguar (0.02%). This is because the noise on the ZeptoOS is very well balanced and regular. The maximum slowdown of the binomial-tree pattern at large scale (1 million processes) is 59.96, 1.93, 0.02 and 9.01 for CHiC, Altix, ZeptoOS and Jaguar, respectively. We clearly see that the specific noise pattern of a system (cf. Figure 1) is more important than the serial noise overhead.

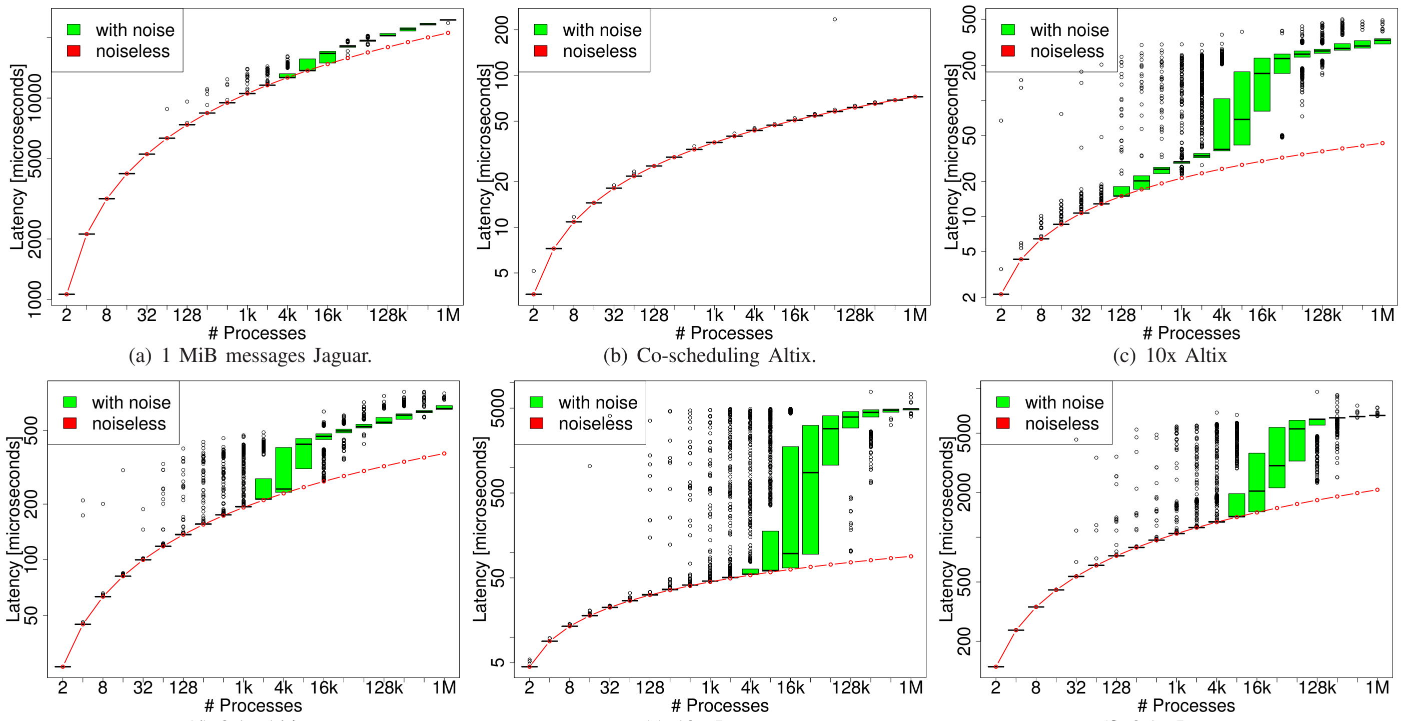
Our identified noise bottlenecks explain Petrini's previous observations (Figure 2 in [1]) where the application scales as expected up to 256 processes and then the execution time suddenly jumped (if all processors on each node are used). It seems interesting that each noise pattern (system) has such a precise and specific inflection point. We recommend using our simulation to tune the operating system such that the noise bottleneck lies beyond the maximum job size.

3) Noise and Large Message Transfers: Our simulations show that large messages are much less affected. This is because noise can be absorbed in the message transmission time (while paying $(k-1)G$ in the LogGOPS model). We repeated the experiments with the artificial noise (Figure 5(a)) with 1 MiB messages and we saw no significant slowdown for this configuration as shown in Figure 6(c). Our real-world traces showed similar behavior: Figure 7(a) shows the influence of noise on 1 MiB messages in Jaguar (cf. Figure 6(a)). Similar behavior has also been shown with the injection of artificial noise in [5]. The simulator can be used to analyze the exact relation between message size and noise propagation but this is outside of the scope of this work.

4) Experiments with Co-Scheduling: The simulator can be used to analyze the influence of co-scheduling all noise on all processes. Co-scheduling has been discussed as a possible solution to the noise problem [1], [19]. In our model, we assume that all noise in the systems is perfectly synchronized (that is, all simulated processes start at the same random position in the noise-trace). Figure 7(b) shows a dissemination-based collective operation in a co-scheduled Altix system. We see that the median collapses into the minimum and the noise bottleneck vanishes. However, the system is not completely immune to noise (we see two outliers), but the probability of noise propagation is much less than in asynchronously scheduled systems. All other systems show a similar behavior and plots are omitted due to space restrictions.

5) Influence of the Network Parameters: Several previous studies suggest that the influence of noise is tightly coupled to the network parameters. Our model in Section III also indicates that slower networks can absorb more noise in g, G and L than faster networks. Our simulation framework provides an excellent tool to study the influence of the network parameters on the noise-sensitivity of parallel programs. Here, we investigate collective operations with 10 times faster and 10 times slower networks. We multiply or divide L, g, and G with/by the factors and leave the host overhead o constant.

We observe in Figures 7(d)-7(f) that the increase and



(a) 1 MiB messages Jaguar.
(b) Co-scheduling Altix.
(c) 10x Altix
(d) 0.1x Altix
(e) 10x Jaguar
(f) 0.1x Jaguar

Fig. 7. Different boxplots for dissemination-based collectives on Altix and Jaguar systems (Figs. 7(d)-7(f) show 10x faster and 10x slower network).

reduction in network speed translates directly into reduced or increased latency at small process counts. However, the difference vanishes at large process counts due to the same characteristic noise bottleneck that we observed in the previous sections. We also see that a slower network is able to absorb more noise than a faster network. It is thus very important to analyze OS noise together with the network parameters in large-scale networks. We will discuss the influence of the network on the noise sensitivity of applications in detail in the next section.

Collective microbenchmarks (or microsimulations) can give a rough estimation of the scaling of parallel applications that heavily use collective communication. However, the timing of collective calls and point-to-point operations is characteristic to each application. Successive collective and point-to-point operations might interfere with each other (e.g., the times when processes leave one operation might influence the following collective operations leading to more amplification). Thus, we analyze real-world applications in the following and use communication/computation traces as input for our simulation.

V. Parallel Applications

We choose three scalable applications with different communication characteristics. All investigated applications are bulk synchronous and consist of multiple computation phases separated by global collective communications. The computation phases often include localized (nearest-neighbor) point-to-point communication. The main characteristics in bulk synchronous parallel applications are the types and sizes of communication operations and the time between the calls. We analyzed our example applications and found characteristic patterns of computation/communication phases.

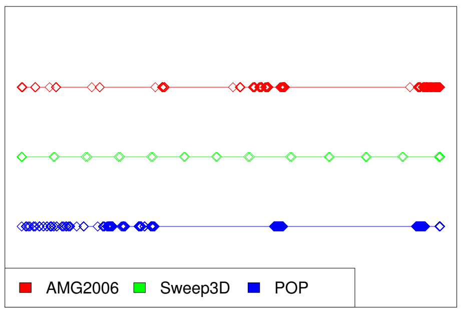
Figure 8(a) shows those patterns for the three analyzed applications. The horizontal lines represent normalized application run time and the points indicate calls to collective operations. Only Sweep3D has a regular structure (fixed frequency) of collective calls. We conclude that, in order to model applications accurately, we have to consider the particular (characteristic) collective and point-to-point call pattern for each application.

We ran the applications on 100 (Sweep3D), 125 (AMG2006), or 128 (POP) nodes of a quad-core low-noise system[5] with 1 process per node. The overhead caused by the trace collection was less than 1% in our experiments and simulations with the system's LogGOPS parameters were within $\pm 1.5\%$ of the real application runtime.

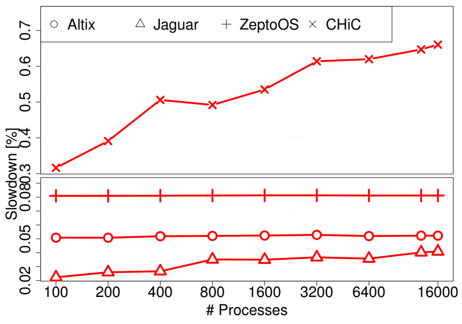
We use our framework as described in Section IV. The schedule generator is able to extrapolate the application traces in order to simulate larger runs. Therefore, it replicates the original trace multiple times, renames the processes accordingly, and recomputes the collective operation patterns. We assume sparse communication in the point-to-point patterns and keep (renamed) partners and timings identical in all replicas of the trace. This replication technique retains many characteristic properties of the application (collective and point-to-point operation types, sizes and duration of the computation) and models a weak-scaling execution [14].

a) Sweep3D: Sweep3D [20] solves a neutron transport problem on a 3D Cartesian geometry. The problem is solved in two nested iterative loops until convergence. The inner loop

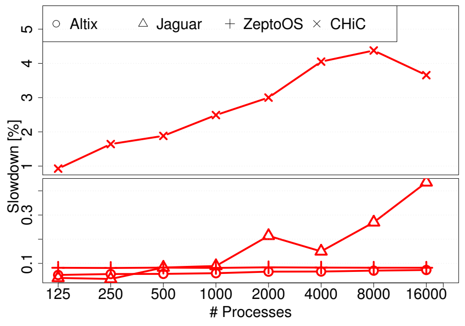
[5]The system had 0.02% highly regular serial noise.



(a) Collective Patterns



(b) Sweep3D



(c) AMG2006

Fig. 8. Collective communication call pattern (a) and full application simulation results for different noise patterns (b,c).

employs a wavefront algorithm where each grid cell cannot be computed before all previous cells in the dimension have been computed. The outer loop uses an allreduction to check for convergence.

Sweep3D had 17.2% communication overhead in our run, and spent 7.6% of it in various collective operations (4 broadcast, 3 barrier, 32 allreduce). All point-to-point communication is blocking. Our largest simulation with 16,000 processes processed more than 289 million messages. Figure 8(b) shows the slowdown due to noise on all systems. The CHiC system causes the highest influence with a slowdown of up to 0.7%. Altix and ZeptoOS show nearly no noise propagation while the slowdown of Jaguar starts at its serial noise level and increases due to propagation. However, we see that Sweep3D is practically immune to noise, even at large runs.

b) AMG2006: The Algebraic Multigrid [21] (AMG) 2006 benchmark represents the core computation of several AMG solvers. We used a three-dimensional Laplace type problem on an unstructured domain with 512 unknowns on 125 processors, arranged as a cube, with a refinement factor of 6 in each dimension. Nearest neighbor communication and the global convergence checks dominate the communication load. Similar Algebraic Multigrid Solvers have been reported to scale up to 125,000 processors on BlueGene/L.[6]

AMG2006 spends 9.72% of the runtime in collective communication functions (24 allgather(v), 356 allreduce, 1 barrier, 23 broadcast, 8 scan). 45% of the communication is mostly spent with nonblocking point-to-point communication. Our largest simulation with 16,000 processes simulated more than 222 million messages. Figure 8(c) shows the influence of the different noise patterns to AMG2006. The slowdown for 16,000 processes nearly reaches 5% on CHiC. Jaguar starts very low at the serial overhead (0.02%) but passes the constant ZeptoOS and Altix quickly. However the slowdown of the three systems remains less than 1%.

c) POP: The Parallel Ocean Program [22] (POP) models general ocean circulation and is used in several climate modeling applications. The logically (mostly) rectangular problem domain is decomposed into two-dimensional blocks with halo-zones for parallel execution. It uses nearest neighbor communication together with global data exchanges. POP was shown to scale up 10,000 processes by Ferreira et al. in [5].

[6]See the "ASC Sequoia Benchmark Codes".

POP called 608 allreductions, 575 barriers and 703 broadcasts in our modified X1 benchmark and spent, on 128 CPUs, 77.2% of the time in communication. 77% of this time is spend in collective calls and 0.2% of the time in nonblocking point-to-point communication. POP is commonly run with such high communication overheads [5]. Our largest simulation with 32,000 processes simulated more than 625 million messages. POP is well known for its noise sensitivity and shows slowdowns of more than 100% on CHiC. ZeptoOS shows no noise propagation while Altix raises above 1%. Jaguar again dominates the lower league with up to 5% noise overhead as shown in Figure 9(a).

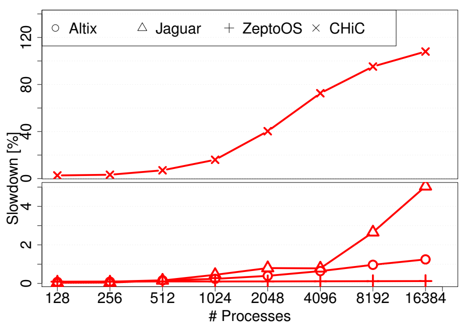
We remark that BlueGene/P shows no noise and applications running on it are not affected at all. We also simulated all applications and systems with co-scheduling as described in Section IV-B4. Co-scheduling eliminated nearly all noise propagation and showed excellent scaling behavior so that results were omitted from the graphs (less than 0.5% slowdown).

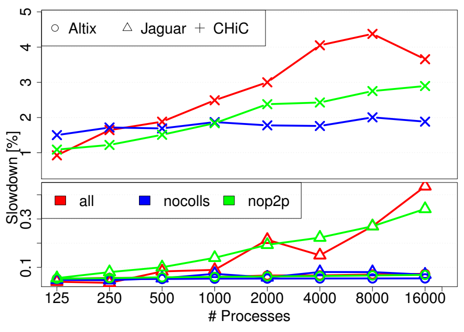
A. Influence of Point-to-Point Communications

Point-to-point messages are commonly ignored in noise analysis. However, as seen in Section III, they can propagate or eliminate noise. To assess the influence of point-to-point messages in isolation, we repeated all simulations (which simulated all communications) and either ignored point-to-point ("nop2p") or collective ("nocolls") calls during the schedule generation. Figures 9(b) and 9(c) show those detailed simulations. Simulations with point-to-point and collective calls are marked as "all". We see that point-to-point messages increase the noise sensitivity at large scale. The results also show that the influence on point-to-point messages grows with a significantly smaller slope than for collective operations. The influence is also clearly depending on the application communication characteristics such that POP is less affected by its (rare) point-to-point communication than AMG. However, our results show that point-to-point messages cannot be ignored in careful noise analyses.

B. Influence of Network Parameters

We also investigated the effect of faster and slower networks by manipulating the LogGP parameters as described in Section IV-B5. Figure 10 shows the influence of increased and decreased network speeds to POP running on the CHiC


(a) POP


(b) AMG

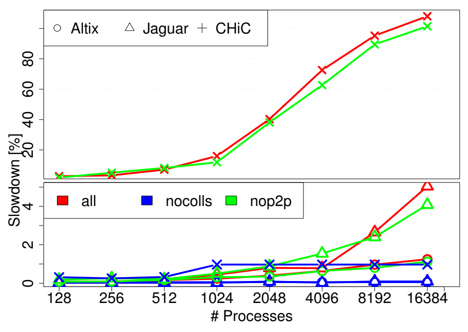

(c) POP

Fig. 9. Simulation results for POP and comparison of the influence of point-to-point vs. collective operations on AMG (b) and POP (c).

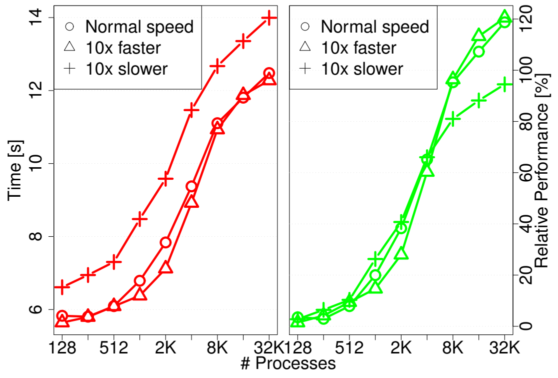


Fig. 10. Network influence on POP

system. The left side shows the total running time with noise and the right side shows the relative slowdown. We observe a similar effect as in Section IV-B5 that slower networks tend to be less affected by noise. Also, at large-scale, faster networks are not able to improve the application speed significantly because noise propagation is becoming a bottleneck.

VI. Summary and Conclusions

In this work we discussed how to analyze the influence of OS noise on applications with analytical modeling and simulation. We show that synchronization of point-to-point and collective communication and OS noise are tightly entangled and can not be discussed in isolation.

In order to assess the more complex interactions in collective communication, we present a LogGOPS-based simulation scheme. Our simulator is able to simulate single collective operations with one million processes in the order of minutes, and full applications with up to 32,768 processes and more than 625 million messages in the order of a day on a single CPU. By comparing the simulation results to previous works, we found that the method accurately reflects effects on large-scale benchmarks. One of our key results is that the influence of noise to collective operations depends on the noise pattern (structure) on the particular machine and is not represented well by static noise models. Based on this, we found that each collective algorithm has a specific inflection point where the slowdown caused by noise suddenly increases and noise becomes a significant bottleneck. Earlier publications about the effects of noise [1] noticed this but did not quantify it or explore it in detail.

We show that the discussions of effects on applications also need to consider the history (arrival patterns) from previous communication operations because the invocation pattern of collective operations and point-to-point transfers are specific to each application and can not be captured by simple (fixed frequency) models. We show detailed simulation results for three applications in different settings and are able to assess the noise-sensitivity of those applications. We found that application scalability is mostly determined by the noise pattern and not the serial noise intensity. Interestingly, we see that Jaguar which has only 0.02% serial noise performs significantly worse then ZeptoOS with four times as much (0.08%) noise. This is due to the fact that ZeptoOS' noise pattern is very balanced while Jaguar shows spurious high detours.

We also study how noise influences applications with different network speeds. We show that noise impacts slower networks less than faster ones. At scale, when it becomes a bottleneck, it eliminates all advantages of a faster network in collective operations as well as full applications. This finding is crucial for the design of large-scale systems because the noise bottleneck must be considered in system design. In addition, we show that co-scheduling would eliminate most noise propagation.

Our simulation toolchain offers multiple new perspectives: (1) the application developer can investigate her application at different scales and systems and find "noise bottlenecks", (2) the system designer can adjust the OS noise level and pattern such that the inflection point is above the typical job size, and (3) novel techniques, such as non-blocking collectives or co-scheduling can be investigated with simulations.

Our proposed model (Section III) suggests that nonblocking operations can be used to relax the synchronization and mitigate noise propagation. A possible direction for future research is the use of nonblocking collective operations that separate starting the operation and waiting for the data.

Acknowledgments: The authors want to thank (alphabetically) Franck Capello, Steven Gottlieb, William Gropp, William Kramer, and Marc Snir for helpful comments that improved the quality of the manuscript. This work was supported by the Department of Energy project FastOS II (LAB 07-23) and the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois.

References

[1] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proc. of the ACM/IEEE Conf. on High Performance Networking and Computing*, page 55. IEEE/ACM, November 2003.

[2] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *Proc. of the 19th Intl. Conf. on Supercomputing*, pages 303–312, 2005.

[3] T.R. Jones, L.B. Brenner, and J.M. Fier. Impacts of Operating Systems on the Scalability of Parallel Applications. Technical report, Lawrence Livermore National Laboratory, 03 2003.

[4] Saurabh Agarwal, Rahul Garg, and Nisheeth Vishnoi. The Impact of Noise on the Scaling of Collectives: A Theoretical Approach. In *12th Annual IEEE Intl. Conf. on High Performance Computing*, 2005.

[5] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proc. of the 2008 ACM/IEEE Conf. on Supercomputing*, pages 1–12, 2008.

[6] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, and Susan Coghlan. The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale. In *Proceedings of IEEE Cluster2006*, 2007.

[7] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009.

[8] Matthew J. Sottile, Vaddadi P. Chandu, and David A. Bader. Performance analysis of parallel programs via message-passing graph traversal. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006.

[9] M. Sottile and R. Minnich. Analysis of microbenchmarks for performance tuning of clusters. In *Proceedings of IEEE Cluster2004 International Conference on Cluster Computing*, pages 371–377, 2004.

[10] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm. Netgauge: A Network Performance Measurement Framework. In *High Performance Computing and Communications, Third International Conference, HPCC 2007, Houston, USA, September 26-28, 2007, Proceedings*, volume 4782, pages 659–671. Springer, 9 2007.

[11] K. Yoshii, K. Iskra, P. C. Broekema, H. Naik, and P. Beckman. Characterizing the Performance of Big Memory on Blue Gene Linux. Technical report, Argonne National Lab, March 2009. ANL/MCS-P1589-0309.

[12] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. LogGPS: A Parallel Computational Model for Synchronization Analysis. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 133–142, 2001.

[13] Gihan R. Mudalige, Mary K. Vernon, and Stephen A. Jarvis. A plug-and-play model for evaluating wavefront computations on parallel architectures. In *IEEE Intl. Symp. on Par. and Distr. Proc.*, pages 1–14, 2008.

[14] T. Hoefler, T. Schneider, and A. Lumsdaine. LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model. Jun. 2010. ACM Workshop on Large-Scale System and Application Performance (LSAP 2010).

[15] Aroon Nataraj, Alan Morris, Allen D. Malony, Matthew Sottile, and Pete Beckman. The ghost in the machine: observing the effects of kernel operation on parallel application performance. In *Proc. of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[16] Radu Rugina and Klaus Erik Schauser. Predicting the Running Times of Parallel Programs by Simulation. In *12th International Parallel Processing Symp.*, page 654, 1998.

[17] T. Hoefler, A. Lichei, and W. Rehm. Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks. In *21st IEEE International Parallel & Distributed Processing Symposium*. IEEE Computer Society, March 2007.

[18] J. D. Emerson and H. Strenio. Box-plots and batch comparison. Understanding Robust and Exploratory Data Analysis, 1983.

[19] Terry Jones et al. Improving the Scalability of Parallel Jobs by adding Parallel Awareness to the Operating System. In *Proc. of the 2003 ACM/IEEE Conf. on Supercomputing*, page 10, 2003.

[20] K. R. Koch, R.S. Baker, and R.E. Alcouffe. Solution of the first order form of three-dimensional discrete ordinates equations on a massively parallel machine. In *Trans. of Am. Nucl. Soc.*, volume 65, pages 198–199, 1992.

[21] Van Emden Henson and Ulrike Meier Yang. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41(1):155–177, 2002.

[22] P. W. Jones, P. H. Worley, Y. Yoshida, J. B. White, III, and J. Levesque. Practical performance portability in the Parallel Ocean Program (POP). *Concurr. Comput.: Pract. Exper.*, 17(10):1317–1327, 2005.