

Analysis of Application Sensitivity to System Performance Variability in a Dynamic Task Based Runtime

Galen Shipman[†]
Patrick McCormick
Los Alamos National Laboratory
Los Alamos, NM, USA

Kevin Pedretti
Stephen Olivier
Kurt B. Ferreira
Sandia National Laboratories
Albuquerque, NM, USA

Ramanan Sankaran
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Sean Treichler
Alex Aiken
Stanford University
Stanford, CA, USA

Michael Bauer
NVIDIA
Santa Clara, CA

ABSTRACT

Application scalability can be significantly impacted by node level performance variability in HPC. While previous studies have demonstrated the impact of one source of variability, OS noise, in message passing runtimes, none have explored the impact on dynamically scheduled runtimes. In this paper we examine the impact that OS noise has on the Legion runtime. Our work shows that 2.5% net performance variability at the node level can result in 25% application slowdown for MPI+OpenACC based runtimes compared to 2% slowdown for Legion. We then identify the mechanisms that contribute to better noise absorption in Legion, quantifying their impact. Furthermore, we assess the impact of OS noise at the granularity of communication, task scheduling, and application level tasks within Legion demonstrating that *where* noise is injected can significantly effect scalability. The implications of this study on OS and runtime architecture is then discussed.

1. INTRODUCTION

Modern high performance computing systems and applications are composed of a variety of system and user level services, all of which are subject to varying degrees of performance variability. In particular, system level services such as parallel file systems, I/O forwarding layers, and system monitoring daemons can compete with application threads for node level and system wide resources. Similarly, at the application level, multi-physics packages and coupled analytics can create contention for resources. Emerging power management services such as power capping at the node and CPU/GPU level can introduce similar performance variability in the form of application and system level threads competing for execution under a fixed power budget. All of these sources of variability can significantly harm the performance and scalability of production applications.

Traditionally the management of resource contention has been an operating system (OS) level function and in HPC the general approach has been to minimize OS level service interrupts also known as *OS noise*. In practice, this is done by containing OS level services through core specialization or by function shipping OS services to a remote sys-

tem thereby providing a level of isolation of system services from the application. Management of resource contention at the application level is handled by the application developer. Many application developers opt for a static partitioning of resources mirroring domain decomposition due to its simplicity. However, this approach leaves many HPC applications vulnerable to the effects of OS noise, especially at scale.

The growing effects of OS noise has been one of the contributing factors in the development of dynamic runtime systems such as Charm++ [1], the Open Community Runtime [2], Uintah [3], StarSs [4] and Legion [5]. Dynamic runtime systems have the potential to react to the performance variability introduced by resource contention and mitigate its effects. While prior work has examined the impact of OS noise within static runtime environments, no prior work has examined the effects of OS noise on dynamic runtime systems. Our work explores these effects and demonstrates the ways that a dynamic runtime system such as Legion can mitigate the effects of OS noise in comparison to traditional static systems such as MPI and OpenACC. We investigate these effects within the context of S3D, a production-grade turbulent combustion simulation. This paper makes the following novel contributions:

1. A comparison of the impact of OS noise on both static and dynamic runtime systems using a real-world application
2. An analysis of the mechanisms that help absorb OS noise in a dynamic runtime
3. Identification of dynamic runtime tasks that are more susceptible to OS noise

Section 2 gives a brief overview of the Legion runtime system. Section 3 describes each version of the S3D application used in this study. Our evaluation and results are then presented in Section 4 and we conclude with implications of our work for future systems in Section 5.

2. THE LEGION RUNTIME SYSTEM

Legion [5, 6, 7] is a dynamic task-based runtime system which relies on a dynamic program analysis to both discover

parallelism and map applications on target machines. Legion programs are structured as a hierarchical tree of tasks, with tasks permitted to launch arbitrary numbers of sub-tasks. All data in Legion is stored within *logical regions*, a relational abstraction that decouples data specification from both its layout and placement in the memory hierarchy. Importantly, all tasks are required to name the set of logical regions that they will access during their execution. The Legion runtime performs a dynamic dependence analysis based on the logical region usage of different tasks to implicitly extract parallelism from Legion programs. From this analysis, the Legion runtime constructs an explicit asynchronous task graph to be executed by the lower-level Realm runtime (see Section 2.1). To hide the latency of this analysis, Legion relies on a *deferred execution model* [8], in which the runtime is performing its dynamic analysis well in advance of the actual application execution. This allows Legion to discover task parallelism, execute tasks out-of-order, and hide operation latencies while still preserving sequential program semantics.

The dynamic analysis performed by Legion is important for two reasons. First, automatic extraction of parallelism eases the burden on programmers. Second, it makes it possible to dynamically adapt how a Legion program is *mapped* onto a target architecture. To map a Legion application onto a particular machine, every task is assigned a processor on which to run, and a *physical instance* must be created (or reused) in some memory for each logical region requested by a task. These decisions are referred to as the *mapping* of a Legion program. In Legion, all of these mapping decisions are exposed directly to the application programmer through a *mapping interface*. A crucial aspect of the Legion mapping interface is that it allows the construction of *mapper objects* (also called mappers) to handle dynamic queries from the runtime about how to map tasks and logical regions. By making this interface dynamic, mappers can introspect the state of the application and the underlying hardware when making mapping decisions. For example, mappers can observe which processors are executing tasks faster or slower (possibly depending on OS noise) and schedule tasks onto more lightly loaded processors. A mapper can continually monitor the changing state of the machine, and decide the best course of action as anomalous behavior occurs throughout a run.

2.1 Realm

Realm[8] is a low-level runtime that provides a portable interface for execution of asynchronous task graphs on systems with heterogeneous computational resources and complicated memory hierarchies. Realm uses a deferred execution model in which all operations (tasks, copies, even critical sections in the form of *reservations*) are included in the task graph with dependencies captured explicitly by *events*.

Realm handles the *scheduling* of operations (i.e., determining when dependencies have been satisfied and when the assigned execution resources are available), but leaves the *mapping* decisions (i.e. where tasks and data should be placed) to the Realm client - a combination of the Legion runtime and the application’s mapper objects in this case.

To allow the client to make intelligent mapping decisions, Realm provides a model of the underlying machine in the form of a graph of *processors* (e.g. x86 CPU cores, CUDA-capable GPUs) and *memories* (e.g. a node’s system memory,

a GPU’s framebuffer, or remote-DMA-accessible memory on another node), with edges indicating memories accessible to a given processor or pairs of memories between which efficient copies can be performed.

The client’s control over the system is maximized by having the performance of operations mapped to the machine model be as predictable as possible. Thus, Realm focuses on providing an efficient mechanism for execution and not interfering with the mapping decisions made by the client. It also tries to reduce, or at least isolate, performance variability in the the underlying operating system and hardware.

The most obvious way in which Realm does this is in the way it sets the CPU affinity masks of threads. Realm uses a large number of threads internally:

- A persistent thread is created for each “CPU” processor exposed in the machine graph. (The number to expose is configurable at runtime startup.) All tasks mapped onto that processor by the client will be executed in this thread. With good mapping decisions, these threads should be constantly busy.
- A persistent thread is created for each “GPU” processor that runs application tasks mapped to that processor. The typical activity of a task mapped to a GPU processor is to perform CUDA kernel launches, and these threads spend most of their time waiting for the CUDA operations to complete.
- Persistent threads are also created for “utility” processors, which are used by the Legion runtime for its dynamic analysis. This analysis work is bursty, so while it is common to expose several utility processors, they are often idle.
- Background DMA threads are created for each NUMA domain and each GPU to perform copies that have been requested by the Realm client. These threads are often very busy, but entirely memory-bound.
- Finally, several background progress threads are created to deal with sending and receiving inter-node messages via the GASNet API[9].

Realm uses `sched_setaffinity` on Linux-based systems to control which CPU cores may be used by each thread. For each “CPU” processor thread, an available CPU core is chosen and the thread’s affinity mask is set to include only that core. The core is also removed from every other thread’s affinity mask, eliminating processor interference due to any other Realm threads. On machines that support Hyper-Threading[10], the second core is removed from the affinity mask of all threads. Once all “CPU” processor threads have been assigned their cores, all other Realm threads share the remaining cores, including any enabled by Hyper-Threading, allowing resource sharing for the bursty workloads these other threads perform. The affinity mask of the original application thread is also adjusted, in the hopes that any other background threads that are created (e.g. for asynchronous file I/O) are similarly kept from interfering with the main computation threads that are being used by the application.

3. APPLICATION

Our goal in this work is to understand the effects of OS noise on production applications. We therefore focused our work on a real production application: S3D, a turbulent combustion simulation. We first provide a brief overview of

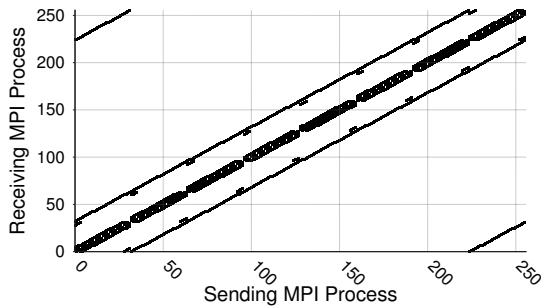


Figure 1: *Spyplot of S3D communication pattern (9 point stencil) at 256 processes*

S3D and then cover the several different implementations of S3D used in our study.

S3D [11] is a reacting flow solver for the direct numerical simulation of turbulent combustion in canonical geometries. It solves the fully coupled unsteady conservation equations for species mass, momentum and energy. The equations are solved on a conventional structured Cartesian mesh using higher order finite difference methods and advanced in time with an explicit multi-stage Runge-Kutta integration scheme with built-in error estimators. The formulation used in S3D is specifically tailored for simulating multi-species chemically reacting flow using detailed models for the thermophysical, chemical and molecular transport properties. The computational kernels that compute these properties are a significant fraction of the computational cost in comparison to other flow solvers that are used for non-reacting flows.

Three different implementations of S3D are examined in this paper. The same benchmark problem is used in each case, and measures the time to solution, presented as wall-clock time per timestep, for a grid size of 48^3 gridpoints per node using a representative n-heptane/air chemical model.

3.1 S3D - MPI only

The original implementation of S3D used a pure MPI-based single program multiple data (SPMD) parallelization. The three-dimensional structured cartesian mesh was equally partitioned among the multiple MPI tasks to achieve perfect load balancing. On multi-core systems, multiple MPI tasks are placed on each network node to utilize all cores available on that node. The finite difference operators used for computing the derivatives and filtering the solution require ghost zones that are 4 and 5 gridpoints wide, respectively, to be exchanged. The ghost zone exchange occurs between the x , y and z neighbors of the three-dimensional cartesian communication topology, also called a box topology, using asynchronous point-to-point message passing. Global communications are required for periodic synchronization and monitoring, but are usually not performance critical. The communication pattern of S3D is illustrated in Fig. 1 and is seen to be very regular due to the nearest neighbor point-to-point (p2p) communication.

Since the communication is dominated by point-to-point message passing among the neighbors in the cartesian topology, the application benefits tremendously when the task placement is chosen such as to minimize the distance that such messages have to travel. In the case of the pure-MPI implementation described here, task placement is achieved at two levels. We use GAMPI, a parallel genetic-algorithm based optimization tool that takes the set of nodes allocated

by the batch scheduler and computes a task ordering by minimizing the mean internode distance [12].

3.2 S3D - MPI+OpenACC

The MPI-only S3D was refactored and augmented with OpenMP and OpenACC directives by Levesque *et al.* [13] thereby porting it to GPU accelerated architectures while also improving its scalability on massively parallel multicore systems. Since all of S3D’s computations occur in loops that traverse the three-dimensional Cartesian grid, they could be easily converted to OpenMP either manually or through automatic parallelization. However, such an approach would have suffered from having too fine granularity and therefore not having enough computation to offset the fixed cost in spawning the threads for each OpenMP region. Therefore, S3D was refactored by combining the various grid loops to form large computational regions that could be hybridized with OpenMP without loss of efficiency. Once S3D was hybridized, the OpenACC directives were added in addition to OpenMP to allow the kernels to be executed on accelerators as described in Ref. [13]. Furthermore, recent advances in the OpenACC directives and compiler support has allowed the parallel regions in S3D-OpenACC to be executed asynchronously on the accelerator using multiple parallel streams of execution on the device. S3D-OpenACC only uses 1 MPI rank per network node and so does not require on-node task placement as described above for S3D-MPI. However, it uses the multi-node task placement computed using GAMPI and MPI collective I/O similar to S3D-MPI.

3.3 MPI+Legion Version of S3D

The Legion version of S3D takes an alternate approach. Starting from the original MPI-only implementation, it replaces an entire call to the `integrate` function (i.e. six stages of right-hand-side calculation followed by explicit Runge-Kutta integration) with a handoff of the input data to a Legion task that performs the equivalent computation. The main Fortran application thread waits for the Legion computation to finish and receives the updated state of the simulation. After possibly performing some occasional filtering, analysis, or checkpointing, the data is then handed back to the Legion side to calculate the next time step. This approach yields the performance benefits of moving the main computation into Legion while allowing the bulk of the code that deals with user interface, monitoring, and file I/O unchanged.

The Legion code is implemented using a tree of tasks. The top-level task creates logical regions that will be used throughout the execution of the simulation. For each time step, sub-tasks perform each of the various physical and chemical computations. Many of these tasks work on different subsets of the simulation state, and as we will show, Legion’s ability to discover additional parallelism and execute tasks out-of-order are key contributors to its improved OS noise mitigation.

The Legion version of S3D uses a customized mapper which supports two different mapping strategies: a *mixed* strategy divides work between the CPUs and GPUs and *all-GPU* strategy places as much work as possible on the GPU. (The relative performance of the two mappings depends on processor speeds, memory sizes, and system bus bandwidth - the correct strategy for a given machine is usually determined empirically.) When possible, the mapper will load

balance work by assigning tasks to sets of processors where the task can execute. In Section 4 we will show how this load balancing feature improves tolerance to OS noise.

4. EVALUATION

In this section we present our experimental evaluation of the impact of system performance variability on three versions of S3D each optimized for different runtimes. First we describe the experimental environment 4.1 used to conduct our evaluation. Next we describe our testing methodology 4.2 describing how performance variability is introduced in each of the three versions of S3D and the amount of performance variability “injected” into the application. We then present the impact of this induced performance variability (noise injection) 4.3 on the performance of the three versions of S3D. Finally, we conclude this section with an analysis of the mechanisms 4.4 within Legion that help mitigate performance variability.

4.1 Experimental Environment

All experiments were performed on two Cray XK-7 systems, the Titan supercomputer and a smaller scale testbed of the same configuration at the Oak Ridge National Laboratory. Each compute node in these systems is composed of one 16-core AMD Opteron running at 2200MHz and one Nvidia K20X GPU with 32 Gigabytes of DRAM memory and 6 Gigabytes of GDDR memory. Nodes are interconnected via the Cray Gemini interconnect in a 3-D torus topology. The system software environment was Cray CLE 5.2UP02.

To verify the experimental results as well as get a better understanding of how noise gets amplified and absorbed in S3D, a validated OS noise simulator was also used. This simulator framework comprises `LogGOPSim` [14] and the tool chain developed by Levy et al. [15]. `LogGOPSim` uses the `LogGOP` model, an extension of the well known `LogP` model [16], 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 [14]. Levy et al.’s tool chain adds the capability to simulate resilience activities such as checkpointing and other scenarios such as in-situ analytics and coupled codes. This tool chain extension has been validated against experiments and established models in [15, 17].

OS Noise was injected into the application using the Selfish Detour [18] utility developed as part of the ZeptoOS project at Argonne National Laboratory. This utility utilizes the `SIGALRM` signal available in POSIX environments to trigger noise injection within one or more application level threads at a configurable interval. A signal handler then samples a random distribution to determine if a configurable amount of work will be conducted within the signal handler thereby creating additional work for the CPU and blocking the application thread during execution of the handler. For all of our experiments we set the probability to unity, guaranteeing that work will be conducted at each interval.

4.2 Testing Methodology

For our experiments, we used the three versions of the S3D application described in Section 3: a CPU only MPI version, a mixed CPU/GPU MPI+OpenACC version, and a Legion version. The Legion version was tested in two configurations: 1) All CPU, where S3D tasks were mapped to only CPUs, and 2) CPU+GPU, where S3D tasks were mapped to both CPUs and GPUs. Using the selfish detour utility, noise is injected into each of the S3D versions. The same set of noise signatures is tested across all S3D versions and this noise is selectively targeted at one or more bound CPU threads. The hybrid MPI+OpenACC and Legion versions differ significantly from the MPI only version of S3D in that the former run a single process per node while the MPI only version is optimally configured with a single process per core on this system. This impacts our noise injection strategy as a single signal handler for noise is generally configured for each process. To accommodate this difference, for the MPI only version, the noise interval is increased by the number of processes per node and the interval timer used for each process is offset by $local\ process\ number \times noise\ duration$ resulting in a wave front of noise across the local processes. In all versions noise injection is not coordinated between nodes.

The noise signatures used for our experiments were based on previous work [19, 20]. For the MPI version of S3D, we explore the impact of injecting noise to a single process per node vs. distributing noise across all MPI processes on the node. For the MPI+OpenACC version, noise is injected into the single master process running on the host, affecting all CPU threads within the OpenACC runtime. For both Legion versions we explore the effects of noise injection on different thread types:

- S3D Fortran Thread: All noise is injected on the thread running the Fortran component of the application. This thread is mainly busy at start-up and tear-down, and idle for most of a run.
- App Thread 1: All noise is injected on the first CPU processor thread allocated to application level tasks. This thread is bound to a dedicated physical core.
- Realm Threads: All noise is injected on the low-level Realm threads responsible for handling active messages, performing data movement, and offloading work onto GPU processors. Realm threads are bound to the set of physical cores not used by CPU processor threads.
- Legion Threads: All noise is injected on the Legion runtime threads responsible for performing the dynamic program analysis necessary to map and execute a Legion program on a target machine. Legion threads are bound to the same physical cores as Realm threads.
- Any Thread: Noise is injected on all threads.

The ability to control where noise is injected within the Legion runtime provides the opportunity to assess not only the ability of dynamic task scheduling to potentially absorb noise but also the impact of system noise on distinct task types in the runtime.

Each noise profile was injected using two different methods, *unscaled* and *scaled*. For the unscaled method, the noise profile being injected on each node was distributed evenly among the threads being targeted on that node, resulting in the node as a whole experiencing one copy of the noise profile. For the scaled noise injection method, the noise profile

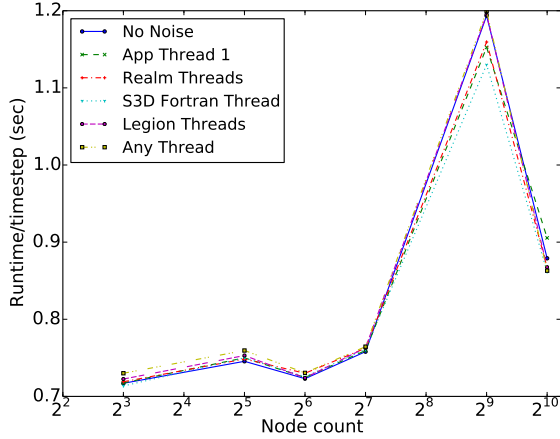


Figure 2: Legion CPU+GPU version with 10Hz-2500us-unscaled noise signature (runtime per step)

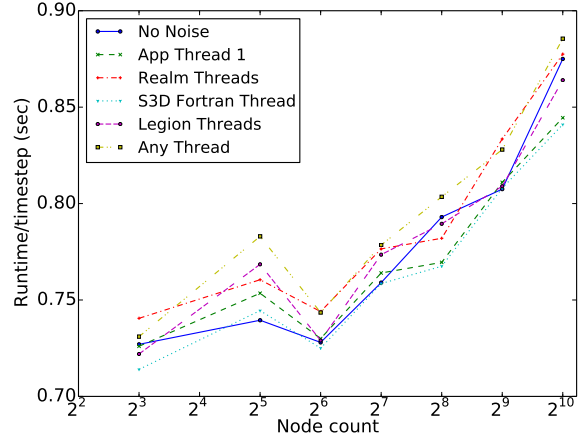


Figure 4: Legion CPU + GPU version with 10Hz-2500us-scaled noise signature (runtime per step)

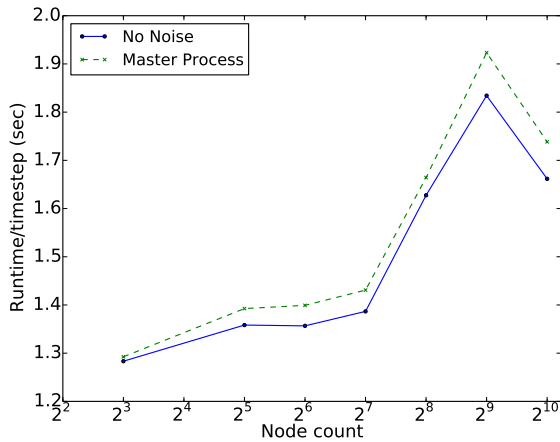


Figure 3: OpenACC version with 10Hz-2500us-unscaled noise signature (runtime per step)

was first scaled up by the number of CPU cores per node, then distributed evenly among the threads being targeted on each node. At a node level, the scaled method results in a net noise equal to one copy of the noise profile per core.

4.3 Noise Injection: Impact on S3D Versions

We first examine the impact of a noise duration of approximately 2500usec occurring at a frequency of 10Hz which corresponds empirically to a loaded preemption of a process by the operating system.

Figures 2,3 illustrate the performance in terms of *walltime/timestep* for the Legion CPU+GPU and MPI+OpenACC S3D versions for unscaled noise injection. The Legion CPU+GPU version shows very little performance impact, 1 – 2%, to this noise pattern irrespective of the number of nodes. The MPI+OpenACC version incurs a minimum of .75% performance degradation at 16 nodes and maximum of 4.5% at 1024 nodes. Overall S3D absorbs this noise signature well for both of these configurations and does not appear to amplify noise as a function of scale due to its use of a nearest neighbor stencil communication pattern. The Legion All CPU and MPI only versions were similarly insensitive.

Next we examine the impact of this same noise signature, 10Hz – 2500usec, scaled by the number of cores per node.

In the Legion and MPI+OpenACC versions this is accomplished by injecting the noise signature with frequency scaled by $10Hz \times \text{cores per node}$ to the single application process per node. In the MPI only version this is accomplished by injecting the noise signature without frequency scaling to every process in the “Distribute to All Processes” configuration and the frequency scaled noise signature to the first MPI process on each node in the “Concentrate to Process 0” configuration. As illustrated in Figure 4 the Legion CPU+GPU version is marginally impacted by this noise signature with the most significant impact (up to 6% degradation) occurring when noise is injected to any thread or restricted to just the low-level Realm threads. This is a result of noise impacting communication and work scheduling threads in the runtime limiting the efficient scheduling of tasks and communication of data dependencies in the S3D stencil. The Legion GPU+CPU version realizes little performance impact, < 1%, when noise is injected into an application CPU processor thread (“App Thread 1”) or the Fortran thread. Figure 5 illustrates a similar performance impact in the Legion CPU only version when noise is injected into the Legion, Realm, or any thread with a performance degradation of up to 16% compared to 5% or less when noise is injected into the application threads. These results indicate that Legion is better able to adapt to noise on the application threads than noise on the Legion runtime’s communication and utility threads. The MPI+OpenACC version slowdown is more pronounced, resulting in 16% – 26% degradation as the noise impacts any communication or CPU level calculations within the process as illustrated in Figure 6. The MPI only version exhibits up to a 4% slowdown when noise is distributed across all processes on a node. When noise is concentrated on node local process 0, performance degradation from 4 – 16% was observed as illustrated in Figure 7. This is a result of a single process per node significantly degrading local stencil computations. When contrasted with the Legion CPU only version with noise injection directed at a single computation task we begin to see the impact of static mapping of tasks to cores when 2.5% net noise is induced. Dynamic task scheduling as in the Legion CPU only version alleviates some of the impact of this type of noise.

Next we consider a shorter duration noise signature (250usec) with higher frequency (100Hz). S3D is in general better able to contend with this noise signature when

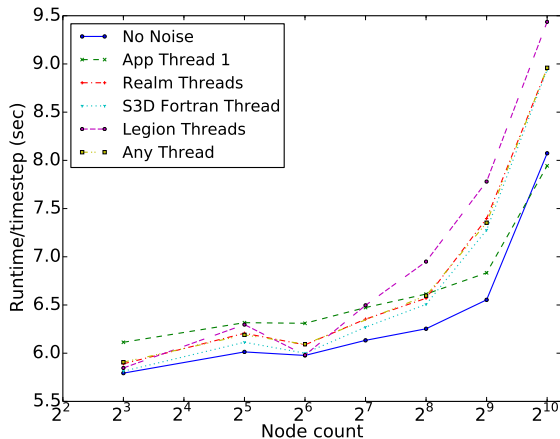


Figure 5: Legion all CPU version with 10Hz-2500us-scaled noise signature (runtime per step)

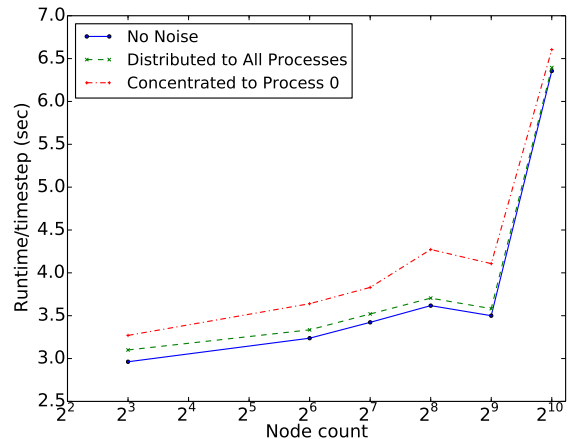


Figure 7: MPI only version with 10Hz-2500us-scaled noise signature (runtime per step)

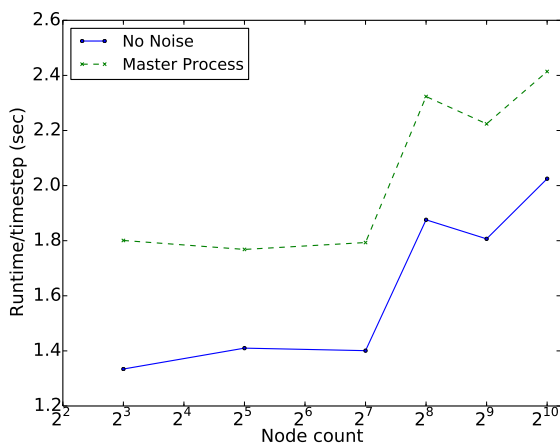


Figure 6: OpenACC version with 10Hz-2500us-scaled noise signature (runtime per step)

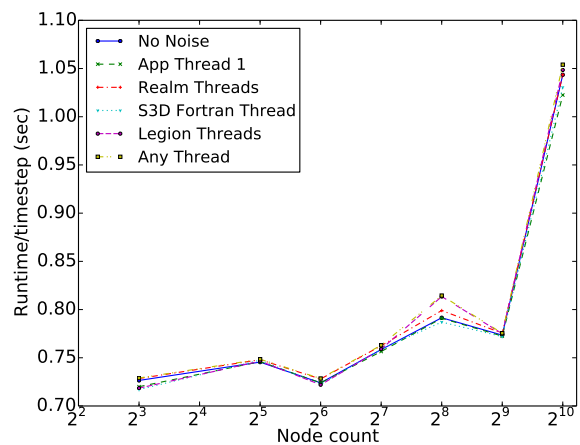


Figure 8: Legion all GPU version with 100Hz-250us-unscaled noise signature (runtime per step)

compared to the 10Hz – 2500usec noise signature. In the unscaled noise signature case we see very little performance impact, from 1 – 3% as illustrated in Figures 8 and 9. Legion CPU only and MPI only exhibit similar results.

When noise is scaled by core count the performance impact is more pronounced. The Legion CPU+GPU version exhibits up to an 8% slowdown with this noise signature with the largest slowdown at 256 nodes and noise injected to either Realm or Legion level threads again demonstrating that the performance degradation of communication and scheduling threads is the most impacting to application performance for S3D. Application performance is minimally affected across all scales when noise is only injected in application level tasks. More pronounced is the performance impact of OS noise on the Legion CPU only version of S3D. Figure 10 illustrates the impact this noise signature has on the Legion CPU only version. Of note is the significant difference in performance degradation when noise is injected in communication and scheduling threads, up to 12.8%, compared to a maximum of 6% when noise is isolated to application computation tasks. Perhaps more significant is the inverse relationship between these scenarios in terms of performance degradation as a function of scale. Figure 11 provides a more detailed view of this phenomena. S3D performance is degraded by 6% from 16 to 256 nodes and

then drops to 0% as scale increases to 512 and 1024 nodes. Conversely, S3D exhibits very little performance degradation when noise is isolated to communication and scheduling threads at smaller node counts but then increases significantly as scale increases, indicating that at larger node counts Legion’s dynamic scheduling of application level tasks can effectively mitigate noise even as scale increases but that communication and task scheduling threads are more susceptible to OS noise in this application.

In all of these cases, the performance impact of directing noise to the “S3D Fortran Core” is significantly lower than the impact of noise on the Realm or Legion threads. Recall that all of these threads are sharing the same set of physical cores (i.e. the ones not assigned to run application tasks), so noise events handled by any of these threads have identical impacts on the availability of execution resources, potentially displacing communication and scheduling work being performed by the runtime. The difference is in which runtime operations are interrupted (because they are assigned to the thread that is handling the noise event). The Fortran thread has no work assigned to it during the bulk of the S3D execution, while the Legion and Realm threads have scheduling and communication operations whose delay impacts overall performance more, especially at scale.

The MPI+OpenACC versions exhibits moderate perfor-

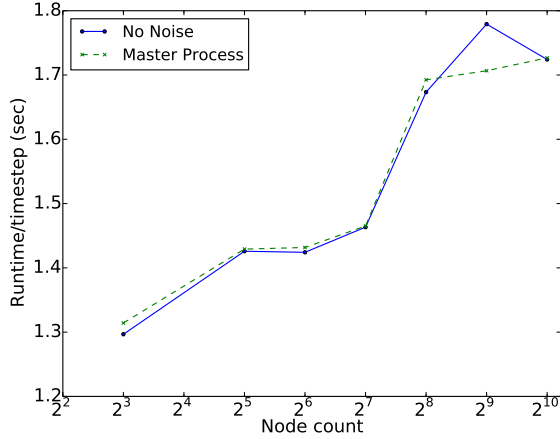


Figure 9: *OpenACC version with 100Hz-250us-unscaled noise signature (runtime per step)*

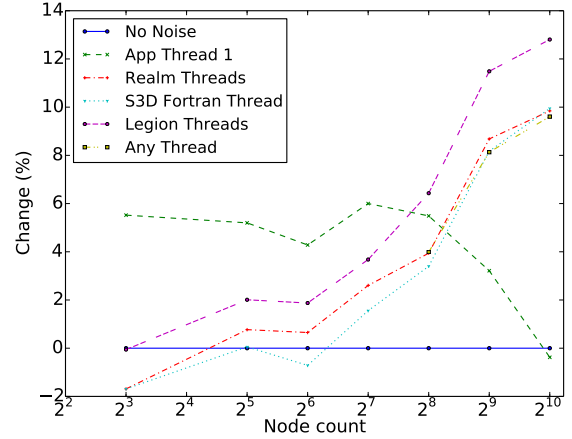


Figure 11: *Legion all cpu version with 100Hz-250us-scaled noise signature (Percentage degradation)*

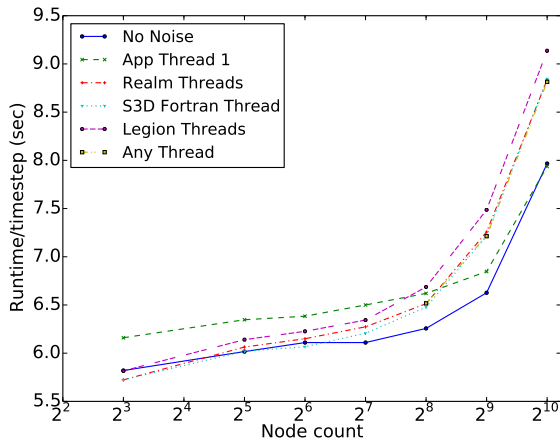


Figure 10: *Legion all cpu version with 100Hz-250us-scaled noise signature (runtime per step)*

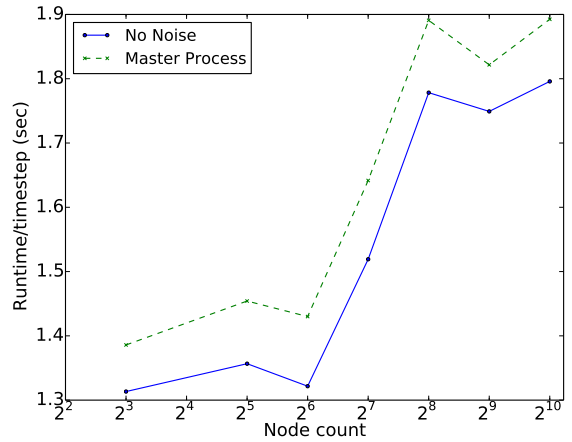


Figure 12: *OpenACC version with 100Hz-250us-scaled noise signature (runtime per step)*

mance degradation under this noise signature, between 4 – 7.5% as illustrated in Figure 12. The MPI only version exhibits a 4 – 7% degradation when noise is injected in node local process 0 and a minimal degradation of 1% when noise is spread across all processes on the node. Compared with the longer duration, lower frequency noise signature (10Hz – 2500usec) the statically mapped version of S3D are significantly less impacted by this noise signature. This is in contrast to the Legion version of S3D whose performance degradation is similar across both noise signatures and with respect to which threads are targeted with noise.

4.4 Mechanisms in Legion that Mitigate Noise

While our performance studies suggest that dynamic task scheduling can alleviate some forms of OS noise a more thorough analysis of the scheduling mechanisms that contribute to this is in order. Legion provides two primary mechanisms for dynamic task scheduling, task scheduling windows similar to out-of-order instruction windows [21] and load balancing of tasks across processors. To evaluate the impact of each of these mechanisms we begin with an experiment at a fixed node count (64) with the same task window size (1024) used in our previous experiments and comparing the no-noise case with the scaled noise signatures with and without load balancing. Table 1 summarizes these results. In this

experiment load balancing reduces the impact of OS noise directed at an application level task from 20% to 5%. These results indicates that load balancing of computational tasks plays a significant role in reducing the impact of OS noise.

Table 1: *Legion all cpu version runtime/step with and without load balancing*

Noise signature (scaled - App Thread 1)	With load balancing	Without load balancing
No noise	6.007	6.01
100Hz-250us	6.373	7.3
10Hz-2500us	6.337	7.27

In our next experiment we demonstrate the impact of task scheduling windows by reducing the window size from 1024 to 2. As in the load balancing experiment we fix the number of nodes at 64 and examine the impact of our two noise signatures with load balancing enabled. Figures 13 and 14 illustrate the impact of reducing the window size with noise signatures of 100Hz – 250us and 10Hz – 2500us respectively. As the window size decreases from 1024 to 256 the impact of noise remains fairly constant indicating that Legion is still able to identify sufficient work within the task stream such that the induced noise continues to have minimal impact on performance. As the task window is reduced

to 128 Legion is no longer able to find sufficient work resulting in performance degradation that begins to dominate any induced noise effects. Performance continues to degrade significantly as the window size is decreased further. These

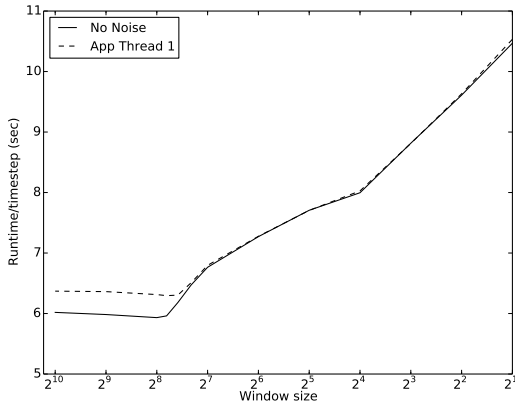


Figure 13: *Legion all cpu version runtime/step with varying window size and 100Hz-250us scaled noise signature*

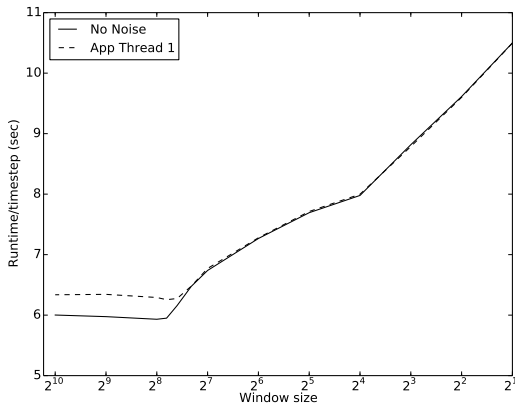


Figure 14: *Legion all cpu version runtime/step with varying window size and 10Hz-2500us scaled noise signature* results indicate that dynamic load balancing coupled with the ability to find sufficient parallelism through out-of-order task windowing to schedule both play a significant role in mitigating node level performance variability.

5. CONCLUSIONS

This paper has presented a study of the impact of OS noise in dynamic runtime environments. Previous works have focused on the impact of static runtime environments (MPI) and have not explored the impact of OS noise on different application level tasks such as computation, communication, and task scheduling. Our results show that dynamic runtime environments such as Legion can absorb OS noise when load balancing and out-of-order task scheduling mechanisms are employed. We show that when noise is injected into a single core with a $10\text{Hz} \times 2500\text{us}$ signature the MPI+OpenACC version of S3D realizes up to a 25% slowdown while the Legion CPU+GPU version realizes at most a 2% slowdown. We further demonstrate that adaptive load balancing can reduce overall impact of noise in the Legion CPU only version from 20% slowdown to 6% when sufficient work is scheduled in

advance by the runtime. This result indicates that on node load balancing and out-of-order execution strategies are important mechanisms in dealing with performance variability.

Our results also demonstrate that application performance is significantly impacted by *where* noise is injected and what threads of execution are interrupted or displaced by OS noise. Previous work has demonstrated that how noise is generated (noise signature) is often more important than the net noise on a system. Our work demonstrates that OS noise that impacts communication and scheduling threads in a dynamic runtime environment reduces application performance up to 12% and grows as a function of node scale. Conversely, under the same noise signature targeted instead to application tasks results in at most a 6% performance degradation at 16 nodes and decreases to 0% at 1024 nodes. This is more pronounced in our experimentation as the number of cores dedicated to the runtime is $1/8\text{th}$ of those dedicated to application level tasks. This places additional constraints on the runtime system when attempting to mitigate this form of noise. Future systems such as Intel’s Knights Landing will contain significantly more processing cores than today’s systems. In these systems, runtimes such as Legion will likely be able to consume more processing cores with similar overheads presenting new opportunities to mitigate the impact of OS noise within communication and scheduling threads.

The observed differences in the noise sensitivity of the different Legion threads show that, in dynamic runtime environments, having the OS run a background operation on a separate thread can be significantly better than “hijacking” a user-level thread. The former consumes execution resources, but still allows the runtime to load-balance across the remaining resources, while the latter also delays specific application or runtime operations that have been assigned to that user-level thread.

Acknowledgments

The majority of the work presented in this work is supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the United States Department of Energy, under the guidance of Dr. Sonia Sachs. Additional support was provided by the Department of Energy’s National Security Administration, Advanced Simulation and Computing Program.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Los Alamos National Laboratory is operated by Los Alamos National Security LLC for the U.S. Department of Energy under contract DE-AC52-06NA25396. 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.

LA-UR-15-22880 Approved for public release; distribution is unlimited.

6. REFERENCES

- [1] L. V. Kale and S. Krishnan, *CHARM++: a portable*

- concurrent object oriented system based on C++.
- ACM, 1993, vol. 28, no. 10.
- [2] "Open Community Runtime," <https://xstackwiki.modelado.org/images/1/13/Ocr-v0.9-spec.pdf>, September 2014. [Online]. Available: <https://xstackwiki.modelado.org/images/1/13/Ocr-v0.9-spec.pdf>
 - [3] Q. Meng, A. Humphrey, and M. Berzins, "The uintah framework: a unified heterogeneous task scheduling and runtime system," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion.*, Nov 2012, pp. 2441–2448.
 - [4] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with starss," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.
 - [5] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 66:1–66:11. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2388996.2389086>
 - [6] S. Treichler, M. Bauer, and A. Aiken, "Language support for dynamic, hierarchical data partitioning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 495–514. [Online]. Available: <http://doi.acm.org/10.1145/2509136.2509545>
 - [7] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Structure slicing: Extending logical regions with fields," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014 – to appear.
 - [8] S. Treichler, M. Bauer, and A. Aiken, "Realm: An event-based low-level runtime for distributed memory architectures," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 263–276. [Online]. Available: <http://doi.acm.org/10.1145/2628071.2628084>
 - [9] K. Yelick, D. Bonachea, W. yu Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, "Productivity and performance using partitioned global address space languages," 2007.
 - [10] D. Koufaty and D. T. Marr, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 56–65, Mar. 2003. [Online]. Available: <http://dx.doi.org/10.1109/MM.2003.1196115>
 - [11] J. H. Chen, A. Choudhary, B. d. Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo, "Terascale direct numerical simulations of turbulent combustion using S3D," *Computational Science & Discovery*, vol. 2, p. 015001, 2009.
 - [12] R. Sankaran, J. Angel, and W. M. Brown, "Genetic algorithm based task reordering to improve the performance of batch scheduled massively parallel scientific applications," *Concurrency and Computation: Practice and Experience*, Mar. 2015. [Online]. Available: <http://onlinelibrary.wiley.com/doi/10.1002/cpe.3457/abstract>
 - [13] J. Levesque, R. Sankaran, and R. Grout, "Hybridizing S3D into an Exascale application using OpenACC: An approach for moving to multi-petaflops and beyond," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC12)*, Nov. 2012, pp. 1–11.
 - [14] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, Jun. 2010, pp. 597–604.
 - [15] S. Levy, B. Topp, K. B. Ferreira, D. Arnold, T. Hoefler, and P. Widener, "Using simulation to evaluate the performance of resilience strategies at scale," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2013 SC Companion.* IEEE, 2013.
 - [16] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "Logp: towards a realistic model of parallel computation," *SIGPLAN Not.*, vol. 28, no. 7, pp. 1–12, Jul. 1993.
 - [17] S. Levy, B. Topp, K. B. Ferreira, D. Arnold, P. Widener, and T. Hoefler, "Using simulation to evaluate the performance of resilience strategies and process failures," Sandia National Laboratories, Technical Report SAND2014-0688, 2014.
 - [18] "Selfish Detour Noise Injection Utility," <http://www.mcs.anl.gov/research/projects/zeptoos/downloads/>. [Online]. Available: <http://www.mcs.anl.gov/research/projects/zeptoos/downloads/>
 - [19] K. B. Ferreira, R. Brightwell, and P. G. Bridges, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Supercomputing'08)*, November 2008.
 - [20] K. B. Ferreira, P. Widener, S. Levy, D. Arnold, and T. Hoefler, "Understanding the effects of communication and coordination on checkpointing at scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 883–894. [Online]. Available: <http://dx.doi.org/10.1109/SC.2014.77>
 - [21] P. K. Dubey, G. B. Adams, and M. Flynn, "Instruction window size trade-offs and characterization of program parallelism," *Computers, IEEE Transactions on*, vol. 43, no. 4, pp. 431–442, 1994.