Performance Characterization of Parallel Discrete Event Simulation on Knights Landing Processor

Barry Williams
Binghamton University
barry@cs.binghamton.edu

Dmitry Ponomarev
Binghamton University
dima@cs.binghamton.edu

Nael Abu-Ghazaleh
UC Riverside
naelag@ucr.edu

Philip Wilsey
University of Cincinnati
philip.wilsey@uc.edu

ABSTRACT

Performance and scalability of Parallel Discrete Event Simulation (PDES) is often limited by fine-grain communication, especially in execution environments with high communication cost. However, the low cost of on-chip communication in emerging many-core processors offers a promise to substantially alleviate conventional PDES bottlenecks. In this paper, we present a detailed evaluation and characterization of multi-threaded ROSS simulator on Intel's Knights Landing (KNL) processor. KNL is the second generation of the Intel Xeon Phi family of processors offering significant architecture improvements including 64 out-of-order multi-threaded cores, sharing of some levels of the cache hierarchy among the cores, fast 2D mesh interconnect network and the ability to reconfigure the processor to support various clustering modes.

We analyze the performance and scalability of ROSS simulator on KNL processor under different thread counts, communication patterns, event processing granularities, synchronization periods, thread placement policies, and workload partitioning schemes. We conclude that within a single KNL processor, up to 2X performance improvement can be achieved compared to commodity Xeon multicore processors. We show that in most cases the performance of ROSS scales well with the best results achieved when thread affinity is assigned, CPU cores are evenly loaded, cache sharing is exploited and communication is limited to small clusters of cores.

Keywords

Parallel Discrete Event Simulation, Intel Xeon Phi, Knights Landing, Manycore Architectures, Performance

1. INTRODUCTION

Parallel Discrete Event Simulation (PDES) is a fine-grain communication-dominated application that has been difficult to scale beyond a modest number of computing nodes, despite the presence of abundant parallelism in many models. Earlier efforts to scale PDES were limited by high communication latencies in traditional cluster computing environments [1, 2, 3]. The advent of multi-core and many-core processors and systems that use these processors as building blocks led to several more recent research efforts to accelerate PDES on these emerging platforms. For example, Bauer et al. demonstrated scalable PDES on IBM BlueGene's supercomputer [4]. Recent work also investigated PDES performance and scalability on multi-core systems such as Intel's Core i7, AMD Magny-cours [5, 6], and the Tilera architecture [7].

A recent trend in computer architecture is the emergence of many-core processors that feature several tens of simple cores integrated on the same chip. A prominent example of such architecture is the Intel Xeon Phi family of processors [8]. Unlike other specialized accelerators, such as the Tilera processor [9] or General Purpose Graphical Processing Units (GPGPUs), the Xeon Phi is similar in the core architecture to standard x86 processors, allowing the use of the rich tool chains and programming environments developed for x86 ecosystem, thus facilitating faster design and deployment of applications.

The first generation of Xeon Phi, called the Knights Corner (KC) microarchitecture, was implemented as an accelerator card interfaced to the rest of the system using PCIe interconnect. To run applications on KC, they have to be explicitly moved to the accelerator and the amount of available memory is limited to what is provisioned on the card - typically 8 or 16GB. The study of [10] investigated performance and scalability of PDES (using multi-threaded ROSS simulator as a vehicle) on KC-based platform. Given the limitations of the KC architecture, the results reported in [10] showed comparable performance to commodity systems. This is true for most simulation scenarios including the ones with a small percentage of events generated remotely. The performance advantage of KC was demonstrated only when the vector units were fully utilized. The reasons for the lack of scalability are the memory limitations, slow in-order individual cores, and the absence of shared caches, requiring memory access for every cross-core communication.

At the end of 2016, Intel released the second generation of the Xeon Phi family, called Knights Landing (KNL). The

SIGSIM-PADS '17, May 24–26, 2017, Singapore.

DOI: http://dx.doi.org/10.1145/3064911.3064929

KNL microarchitecture and organization are significantly different and more advanced than the KC microarchitecture. First, the KNL processor is implemented as the main CPU in the system and programs executing on it have access to the entire system memory, including the disk. Second, the out-of-order execution capability with branch prediction was added to individual cores, thus alleviating the computational bottleneck of slow in-order cores of KC. Third, neighboring cores in KNL share their L2 cache (which acts as the last-level cache in this system), thus allowing threads executing on these "buddy cores" to access their shared data through the L2 cache and avoid expensive memory accesses.

These important new features of KNL make it imperative to investigate, characterize and understand PDES performance and behavior on this architecture and contrast it with the earlier results obtained for KC processors. Our goal in this paper is to understand how the new features of KNL impact PDES performance under different conditions, execution scenarios and parameter settings.

We pursue these investigations via experiments running ROSS parallel discrete event simulation kernel [11] on KNL processor. For our experiments, we use a multithreaded version of ROSS [5], where the simulation processes have been replaced with threads communicating directly through shared memory without relying on MPI-based communication. Previous studies demonstrated that this variant of ROSS is more efficient compared to the MPI-based version as it reduced the amount of data copying in the course of communication and takes more direct advantages of available shared cache hierarchy (the L2 caches shared by the buddy cores in the case of KNL). Our goal in this paper is to determine conditions under which PDES performance can scale up with the number of cores on a single KNL node, which is capable of executing up to 256 threads in total - 4 threads on each of the 64 cores. We also aim to understand performance bottlenecks and consider possible approaches for future work that can address these bottlenecks.

The main contributions and the key conclusions of this paper are:

- **KNL offers significant performance gains for PDES**: Our study represents only the first look at using KNL architecture for executing PDES applications, yet we demonstrated promising results. If the issues of core loading, model partitioning and thread placement are carefully considered in concert with the architecture details, significant performance advantages can be realized compared to an unoptimized simulation, and also compared to simulation running on commodity Xeon multicore systems. We note that it is not just the sheer power of the cores, but also the new architecture design features that make simulations on KNL substantially faster and more scalable compared to commodity Xeon processors. We expect that the results of our work will motivate future research into more architecture-specific optimizations to further boost PDES performance on KNL systems. Future research needs to also consider expanding this work across multiple KNL nodes.

- **Affinity should be controlled**: We show that in most scenarios, the best results in terms of committed event rate of simulation are achieved when affinity is used to pin threads to CPU cores.

- **Balanced core loading matters**: We show that in most scenarios, the best results in terms of committed event rate of simulation are achieved when each CPU core is evenly loaded with modest number of threads - typically 1 or 2 threads per core provides the best performance. Filling the chip to capacity by executing 4 threads per core often results in core oversaturation, worse cache performance and overall slowdown. Furthermore, balanced loading results in better performance over round-robin core assignment scheme even if some cores remain completely unused - this is because synchronous progress of simulation threads promoted by even loading is critical for PDES.

- **Communication patterns matter**: We demonstrate that specific thread-to-core placement for the same number of threads per core and the same amount of remote communication significantly impact performance in many cases. In addition, we demonstrate that the *message flow* needs to be considered as it also has a noticeable impact on simulation performance.

- **Communication-aware partitioning is critical**: We demonstrate that it is critical to properly partition the simulation model to minimize the number of different cores with which a thread communicates. Several-fold performance improvement can be realized if most communications are limited to only a cluster of cores.

- **Detailed performance characterization**: We perform a detailed evaluation of multithreaded ROSS on an Intel KNL processor. We investigate the performance sensitivity to a number of simulator parameters, including percentage of remote events, synchronization period, event processing granularity, thread placement and workload distribution policies. We also compare simulation on KNL against KC and commodity Xeon processors and show that KNL significantly outperforms them.

The rest of the paper is organized as follows. Section 2 provides the background on Intel's Knights Landing architecture and describes our evaluation and experimentation methodology. The results of our experiments are presented and discussed in Section 3. Section 4 reviews the related work and we offer our concluding remarks in Section 5.

2. BACKGROUND AND EXPERIMENTAL SETUP

In this section, we overview ROSS simulator, review the Intel Knights Landing architecture and describe our experimental setup.

2.1 Overview of ROSS Simulator

PDES is a parallel implementation of DES [12], extending the performance and capacity advantages of parallel processing to this important application. The key idea behind PDES is to break the simulation model into multiple Logical Processes (LPs) and allow these LPs to execute in parallel on multiple processing cores. The LPs communicate with each other by exchanging time-stamped event messages [12,

13]. The LPs have their own local event queues and process the events from these queues in time-stamped order. Some events are generated locally within the LP, while other events are generated remotely and the timing of the event arrival to a destination LP depends on the physical delays that the message encounters while traversing the on-chip interconnects (for core-to-core communication within a chip) or network links (for cluster-level communication).

Due to the event dependencies between the LPs and physical delays in the system, a PDES simulation engine needs to use synchronization mechanism to ensure that events are executed at different LPs in correct time-stamped order. Two types of synchronization algorithms are used in PDES systems: conservative and optimistic. To support recovery to a safe state upon a rollback in an optimistic simulator, state checkpoints during simulation run have to be created. These event histories can grow large over time and the ones that are no longer needed must be garbage collected. To achieve this, the Global Virtual Time (GVT) is periodically computed to compute the global progress of the simulation.

For the experiments in this paper, we use ROSS [11] optimistic PDES simulator. To effectively exploit the shared memory available on Knights Landing processors, we use multi-threaded implementation of ROSS [5], where the simulation processes are implemented as threads, as opposed to processes, requiring no expensive MPI-based communications and directly exploiting shared memory hierarchy.

PDES simulations have to be driven by benchmarks. The most popular and versatile benchmark for evaluating PDES is the classical *Phold* model. *Phold* is a synthetic, but versatile benchmark that allows characterization of the performance of applications under different scenarios. For example, it allows control of the percentage of events generated locally to the same core and the percentage of events generated for the other cores (thus, requiring inter-core communication and delays). One can also alter the event processing granularity (EPC) to control how much CPU processing is required for each event. As a result, this allows us to evaluate systems with different computation/communication balance (by varying the EPC) and with different execution locality patterns (by varying the percentage of remote events).

2.2 Overview of the Intel Knights Landing Architecture

The Intel Knights Landing [8, 14] is the second generation of the Many Integrated Core (MIC) architecture designed to be used as both a standalone processor and a co-processor for High Performance Computing (HPC) applications.

Knights Landing processors [15] feature up to 72 cores, each capable of executing 4 simultaneous threads. The cores run at a maximum frequency of 1.3 GHz and can achieve better than 6000 Gflops/s single precision and 3000 Gflops/s double precision when the vector processing units are utilized fully.

A major upgrade to the Knights Corner architecture, Knights Landing adds branch prediction and out-of-order execution logic to each core. Vector Processing Units (VPU) have been increased to 2 per core. Also, a 1 Mbyte L2 cache is now shared between every core pair, forming a tile. Finally, KNL systems are augmented with 16 GB on-package MCDRAM memory module. In its default mode, it acts as the L3 cache to the DDR4 memory. Our version of the Knights Landing processor has 64 cores and 96 GB of DDR4 memory. The high-level diagram of the Knights Landing architecture used for this study (not including the DDR4 memory) is shown in Figure 1.

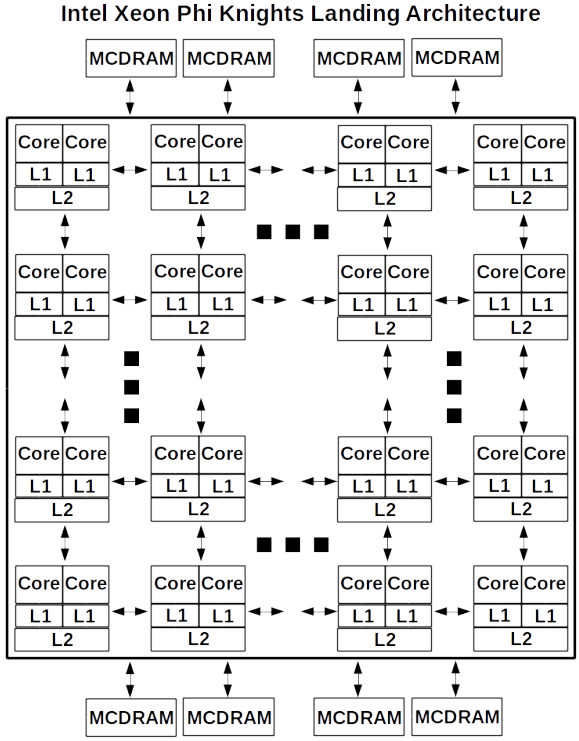

Figure 1: Intel Knights Landing Architecture

KNL processor is commonly socketed and utilized as a standalone CPU as is the case in our experimental system. KNL runs standard Linux distributions as a full host computer thus eliminating the idiosyncrasies of accelerator interfacing.

2.3 Experimental Setup and Metrics

Table 1 summarizes the configurations and hardware details of the host Xeon processor and the Xeon Phi Knights Corner (KC) and Knights Landing (KL) processors. Note that Xeon and Knights Corner systems are only used for comparison purposes in the results presented in Section 3.8.

For all presented experiments, we execute the multi-threaded version of ROSS simulator driven by the Phold benchmark. In the Phold benchmark, we vary the thread count, the percentage of remotely generated events, the GVT interval, the event processing granularity (EPC), and the thread placement and assignment policies. Our goal is to understand the behavior and scaling trends of the ROSS simulator while executing on a single Knights Landing node. We also directly compare performance against a 12-core Xeon processor and a Knights Corner system (the previous generation of Xeon Phi).

Platform	Xeon	KC	KL
Model	E5-2620	5110p	7230
Frequency	2.40GHz	1.053GHz	1.3GHz
# of Cores	12	60	64
Memory Type	DDR4 2133	GDDR5	DDR4 2400
Memory Size	60G	8G	96GB + 16GB
OS	CentOS 6.6	uOS	CentOS 7.2
Compiler	Intel Parallel Studio Cluster V 2017	Intel Parallel Studio Cluster V 2017	Intel Parallel Studio Cluster V 2017

Table 1: Details of Experimental Platforms

We report the performance results in terms of committed events per second. As we increase the number of processing nodes, we maintain the number of starting events per node, thus proportionately increasing the total number of events generated by the simulator. If the underlying system is capable of efficiently keeping up with this load without incurring additional delays, we can expect the committed event rate to also show improvements commensurate with the increase in the number of nodes. This is known as *weak scaling* [4]. Our results and scaling trends clearly indicate the potential for using Knights Landing processors as efficient engines for running parallel simulations. A summary of the simulation parameters is shown in Table 2.

Variable	Value	Description
Remote %	0 - 100	Proportion of Events Local vs Remote
EPC	0, 100, 500	Event Processing Time
Initial Events	128 * # Threads	Number of events to start simulation
GVT Period	32, 128, 512	Synchronization frequency

Table 2: Simulation Parameters

3. EXPERIMENTAL RESULTS AND DISCUSSION

We present our analysis along multiple dimensions. First, we analyze in detail the scaling trends and the impact of multiple simulation parameters (such as the percentage of remote events, the event processing granularity, and the GVT period) on the performance of ROSS simulator on KNL. We first perform these studies assuming the round-robin assignment of threads to cores, and then study different thread assignment and placement policies. We first consider simple scaling where at most one thread is placed on a core, and then we investigate the impact of multithreading support by executing multiple threads per core, up to 256 threads total.

3.1 Performance under Round-Robin Scheduling

We first analyze ROSS performance under the Round-robin assignment of threads to cores. While this assignment scheme is the most natural, it can create uneven loading of the individual cores at large thread counts in situations where multiple threads are placed on some cores. We quantify the performance problems arising from that effect in this section. The issue of uneven core loading on the same chip becomes especially important in many-core architectures where both the likelihood of such scenario and its impact on performance increase. We also analyze the impact of thread affinity on the performance.

3.1.1 Single Thread per Core: Impact of Thread Affinity

In our first experiment, we analyze the committed event rate of ROSS executed on KNL for different percentages of events that are generated remotely. For this experiment, we assume a fairly large GVT interval of 512 (a parameter in ROSS) and the event processing granularity (EPC) of zero. A large GVT interval reduces the overhead of GVT computation (which is significant on many-core architecture such as KNL, as we demonstrate later), but increases the rollback cost. The low EPC value implies that event processing does not consume any CPU time, other than generating a new event and sending it to a random destination LP. This represents an extreme case of a communication-dominated model. In our experiments, we used a *weak scaling* model, where the number of simulation events increases proportionately with the number of threads. The metric of interest in this case is the total number of committed events per second. In a scalable execution scenario, the addition of extra cores/threads will translate into commensurate increases in this metric.

We present the results for two cases. In the first case, we scale the simulation up to the available number of cores by placing at most one simulation thread on each core. In the second case, we exploit the presence of simultaneous multithreading support within each core and scale simulation upto the maximum number of threads supported by the core (up to 4 threads per core). Since the trends and observations from these experiments are quite different, we present these results using separate figures and analyze them separately.

The results for simulations upto 64 threads are presented in Figure 2. The total committed event rate is depicted as a function of the number of threads for different remote percentages. In this experiment, we used the model with zero percent remote events (that is, all events are generated locally and the cross-core communication is only needed for computing GVT), and another model with 5% remote events to gauge the impact of remote communication. The results are presented for each case with and without thread affinity setting. When thread affinity is set, each simulation thread is pinned to a particular core throughout its execution, thus promoting the exploitation of caches and locality of references. When thread affinity is not set, threads can bounce across different cores following context switches and CPU yielding events.

For executions up to 64 cores, the simulation performance scales well for both 0% and even 5% remote events, although better scaling and substantially higher performance is observed for 0% remote events, as expected. Specifically, for models with affinity set, we observed a 34X speedup compared to sequential simulation for 0% remote event case, and only 20X speedup for 5% case. In terms of direct comparison, the case with 0% remote is roughly twice as fast as the case with 5% remote for 64-way simulations. It is

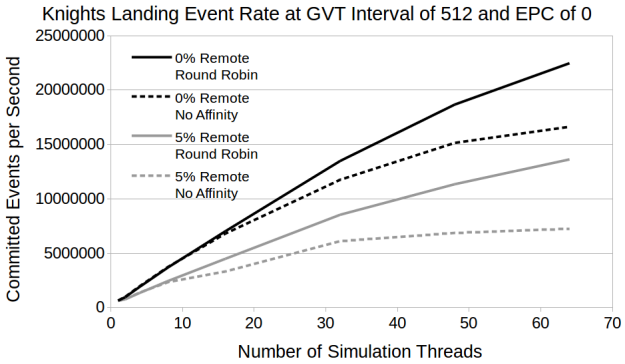

Figure 2: Committed Event Rate with and without affinity, 0% & 5% Remote

still impressive to observe that even simulations with 5% remote events enjoy solid scaling on KNL architecture up to 64 cores, which is in sharp contrast to the results obtained for similar scale on multi-chip systems [1].

Another important observation from the results presented in Figure 2 is that thread affinity matters and it significantly improves performance for simulation models with and without remote events at larger thread counts. For the case with no remote events, there is virtually no performance difference between the runs with and without affinity for thread count below 16. As the simulation thread count increases above 16, the system with thread affinity starts to outperform the one without it, and the performance gap widens as the number of simulation threads increases. For 64-way simulation, we observed 35% performance benefits of affinity. The reason behind such behavior is that the increase in the number of simulation threads and utilized cores leaves fewer cores idle and increases the likelihood that a particular thread will be reassigned to a different core upon a context switch or a CPU yielding event. We observe similar trends for the simulation model with 5% remote events, but this time the divergence starts at a lower point (about 8 cores), performance with no affinity flattens much earlier, and the relative advantage of affinity is even higher at large core counts (for example, the difference is 88% for 64-way simulations). Therefore, in the rest of the paper we assume fixed thread affinity, unless indicated otherwise.

3.1.2 *Single Thread per Core: Impact of Remote Communication*

Figure 3 compares the scaling of fixed-affinity models for different values of remote percentages when the number of threads is varied from 2 to 64, with at most one simulation thread per physical core. The simulation performance scales linearly all the way to 64 cores for the fraction of remote communications up to 10% - this is quite an impressive result showing better scalability than on multi-chip systems of similar scale. Of course, as seen from the figure, the slope of the performance graphs decreases with higher remote percentages, because cross-core communication starts to become a more dominant factor. At 25% remote events, the simulation scales to about 32 cores, at which point the performance curve flattens. For the models with 50% remote communications and above, there is no scalability and adding more cores/threads does not result in higher performance. In summary, results presented in Figures 2 and 3 lead to the following key observations:

Observation 1: If a simulation model can be properly partitioned so that the percentage of remotely generated events is kept at 10% or below (for the parameters used in this study), almost linear performance increases of as much as 34X can be realized on a KNL system as the simulation scales to 64 threads when at most one thread is placed per core. As the simulation scales to 128 threads, performance continues to increase to as much as 44X as shown in the following section. If partitioning can not achieve this level of locality, then performance tapers off earlier or the model does not scale at all.

Observation 2: Thread affinity is an important feature for achieving higher performance of ROSS on a KNL system. Specifically, when threads are pinned to the individual cores, performance improvements in the range of 35% to 88% are realized for 64-way simulation. Furthermore, the advantages of set affinity increase with larger thread counts and larger fraction of remote communication.

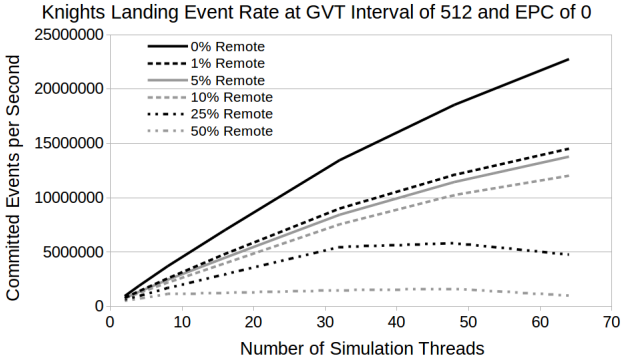

Figure 3: Committed Event Rate For Different Remote Percentages

3.1.3 *Scaling with Multiple Threads per Core*

The results presented above demonstrate scaling trends for up to the number of available cores. Next, we investigate the opportunities to extract further performance from a KNL chip by increasing the number of threads placed on each core, thus exploiting SMT support available within each core. Figure 4 extends the results of Figure 2 by increasing the number of simulation threads to 256 (four threads for each core) in a round-robin assignment of threads to cores.

As seen from the results, even a small remote percentage has a dramatic impact on the simulation event rate. With zero percent remote events (i.e. all generated events are targeting the same LP), the simulation event rate first grows linearly with the number of cores up to the point where the number of simulation threads exceeds the number of cores (which happens at 64 cores). For simulations with no remote events, as the number of threads exceeds the number of physical cores, the simulation commit event rate drops because of the imbalanced core loading, and the simulation progress is limited by slow threads. Eventually, performance

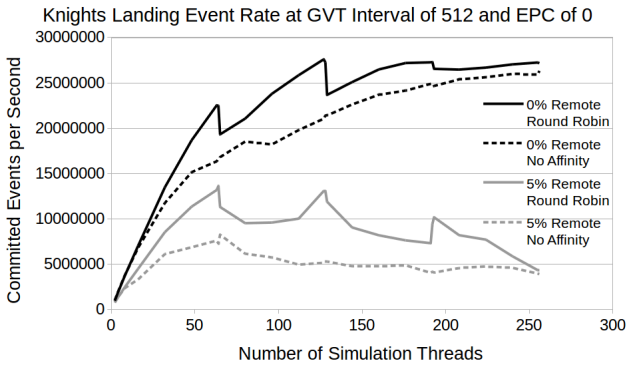

Figure 4: Committed Event Rate with and without affinity, 0% & 5% Remote

recovers as more threads are added, but the nature of this recovery depends on weather or not thread affinity is used.

With thread affinity control, the simulation threads that are pinned in a way that they share physical cores become slower. As a result, they consistently lag in performance and impede the simulation progress, because all threads have to be periodically synchronized to compute GVT. As can be observed from the graph with thread affinity, as we move from 64-way to 65-way simulation, performance drops by 14%, which roughly matches the performance drop that a single thread experiences when it simultaneously executes on a core with another thread. As we add more simulation threads, less cores are being wasted waiting at GVT interval for slower threads and the overall simulation performance gradually improves. It takes 24 more cores to match performance of a 64-thread simulation, and the performance peak is reached at 128 threads (two threads per physical core). After that, performance again drops sharply due to further core imbalance and slowly recovers. Performance never reaches the level recorded at 128 threads due to over utilization of cores beyond that point.

It is also interesting to compare this case with simulation results where thread affinity is not set and threads can migrate among the physical cores across context switches. In this case, performance grows more gradually and there is only a very small drop from 64 to 65 cores that is quickly recovered. This is because no single simulation thread becomes a constant bottleneck and threads are not tied up by slower cores. However, due to poorer exploitation of caches in simulation with no thread affinity, the absolute performance is always below the case where affinity is used. Similar trends can be observed for the case with 5% remote events, although unbalanced execution creates additional rollbacks in this case, which degrades performance further and makes recovery from performance dips more challenging. In this case, the best performance is observed with 64 threads, the case with 128 threads almost matches it.

For simulation models with non-zero remote percentage, the additional problem due to uneven progress of the simulation threads is the increased probability of straggler events and rollbacks. Despite much lower values of committed event rates at non-zero remote percentages, those simulations also show slower performance increases with the increase in thread count.

3.2 Impact of Balanced Loading

To address performance problems of round-robin thread assignment, we studied alternative thread placement schemes that preserve balanced core loading when the number of threads exceeds the number of available physical cores. Figure 5 and Figure 6 show the impact of these schemes on performance. Specifically, we implemented two such schemes, called *linear* and *balanced*. *Linear* assignment assigns threads to cores in a way that consecutively saturates individual cores. For example, the first four threads are placed on core 0, the next four threads are placed on core 1 and so on. *Balanced* assignment scheme evenly distributes all simulation threads across all cores, such that the number of threads per core depends on the total number of threads in simulation. For example, for a 96-way simulation, 48 cores (cores numbered 0 through 47) will execute two threads each, and the other 16 cores will remain idle.

We present the results for the scenario with no remote events (Figure 5) and 10% remote events (Figure 6). In the case of no remote events, the performance dip due to imbalanced core loading is encountered only because of GVT computation cycle. The impact is modest, only about 10% of performance is lost with round-robin assignment when we start exceeding the number of cores. Note that balanced and round-robin assignments perform about the same, but linear assignment exhibits smoother growth with lower performance at smaller thread counts. This is because at smaller thread counts, a few CPU cores remain over utilized (executing 4 threads each) while others remain idle. For the execution scenario with 10% remote events (Figure 6), the performance drop after 64 cores is much more significant - about 50% of performance is lost with round-robin after we reach 64 and then 128 simulation threads. This is because a non-trivial amount of remote communications causes straggler events, increases the number of rollbacks due to asynchrony in simulation progress, and decreases the simulation efficiency. Balanced execution matches the round-robin performance peaks at multiples of 64 threads, but provides better performance at all intermediate points. As in the case with no remote communication, performance of linear scheme lags behind the other two, but the graph is more smooth without rigid peaks and valleys. At very large thread counts, all three schemes converge to the same values. From these results, we make the following observation:

Observation 3: To avoid potential performance problems, it is important to balance the core usage and assign each core the same number of threads. While the performance peaks at discrete points match the performance of a round-robin scheme, balanced loading provides significantly better performance for the intermediate points, and the performance difference increases with higher percentage of remote events.

From all performance results presented up to this point, the following observation can be made in terms of optimal core loading for the models considered so far:

Observation 4: When the percentage of remote events is significant (e.g. above 10%), it is counter-productive to place more than one thread to a physical core. With lower

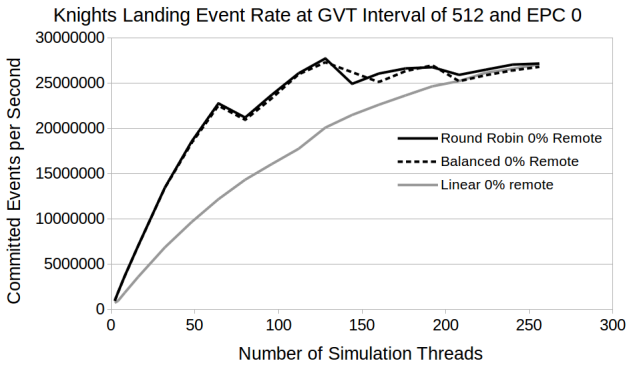

Figure 5: Committed Event Rate for KNL for Different Placement Schemes, 0% Remote

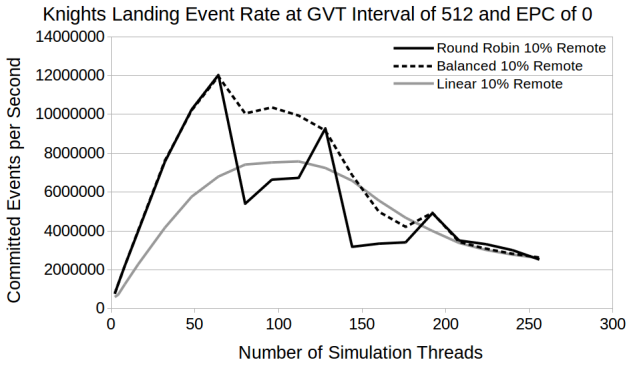

Figure 6: Committed Event Rate for KNL for Different Placement Schemes, 10% Remote

remote percentage, a higher performance can be achieved with two threads per core, but the thread-to-core assignment has to be balanced. No model with small event processing granularity (EPC of 0) features performance improvements beyond two threads per core.

3.3 Impact of Event Processing Granularity

Figure 7 shows the impact of the event processing granularity (EPC) on the simulation commit event rate for different number of threads with no remote events. The results are shown for the large GVT value of 512. Three different EPC values are compared - 0, 100 and 500. Larger EPC values add longer processing loops inside each event. In all three cases, we used balanced assignment of threads to cores, as described previously.

As expected, the event processing rate decreases as the EPC increases and the simulation becomes more computationally-bound. However, the performance scalability curves exhibit quite different behavior depending on the EPC value. For the EPC value of zero, the maximum performance is achieved with 2 threads per core and further increase in the number of threads does not increase the event commit rate. In contrast, as the EPC value increases and simulation becomes more computationally bound, placing additional threads on the cores results in further performance gains - the highest performance is observed for 256-way simulation for both 100 and 500 EPC values.

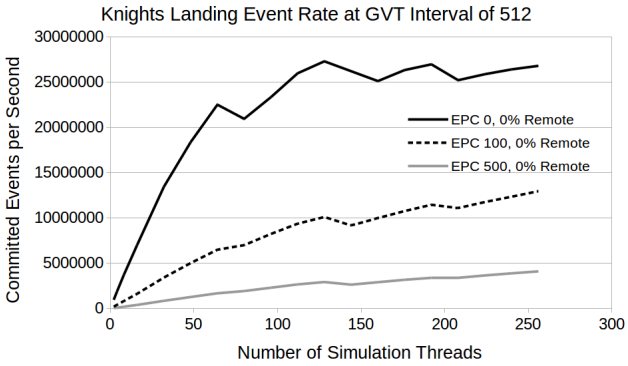

Figure 7: Committed Event Rate for EPC values of 0, 100 & 500 with no Remote Communication

Figure 8 shows the event commit rate for different EPC values (0, 100 and 500) for the situation where 10% of events are generated remotely. In this case of 10% remote events, the balance between computation and communication is tilted towards communication for the EPC of 0. Consequently, as seen from the results of Figure 8, the scenario with EPC of 0 and 10% remote events produces more rollbacks, lower efficiency and lower event commit rate at larger thread counts; the performance is highest at 64 threads in this case and drops continuously after that. In contrast, even with 10% remote events, simulations with 100 and 500 EPC continue to scale up to the maximum number of threads, although the performance increases at larger thread counts are lower than in the case with no remote communication. A somewhat counterintuitive result demonstrating the complex nature and the inter-dependencies of simulation parameters is that after about 150 threads the absolute performance of simulation with EPC of 100 exceeds that of simulation with EPC of 0, despite the former requiring 100 times longer processing time for each event compared to the latter. The reason is that longer processing delays provide more opportunities for the remote events to be generated on time, thus decreasing rollbacks and increasing simulation efficiency — the execution progresses more predictably and steady in this case.

In summary, our experiments with various EPC values and remote communication frequencies lead to the following observation:

Observation 5: As event processing time increases and simulation becomes more computation-bound, the commit event rate scales with the number of threads all the way to the maximum number of threads that can be executed on a KNL chip (256). More surprisingly, the absolute performance of simulations with higher EPC can be higher than simulation with lower EPC at large thread counts for models with substantial remote communication.

3.4 Impact of GVT Period

Next, we analyze the impact of the GVT period on the simulation performance. Figures 9 and 10 depict the simulation event rate for three different GVT intervals: 512, 128 and 32. Results are presented for the remote percentages

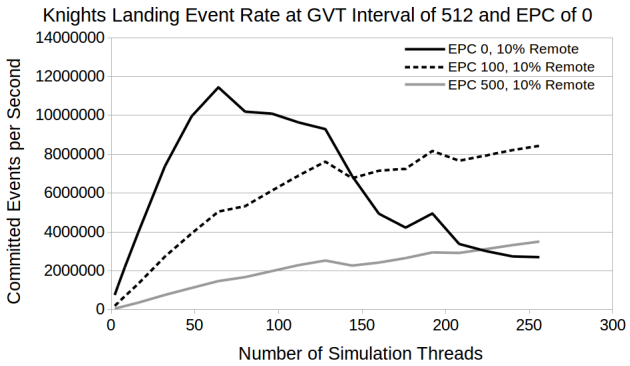

Figure 8: Committed Event Rate for KNL for EPC values of 0, 100 & 500 for the Remote Percentage of 10%

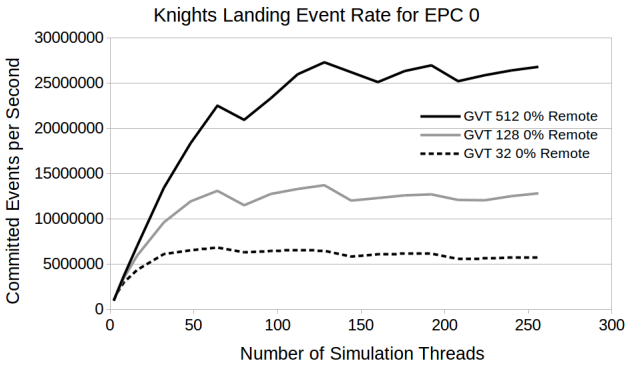

Figure 9: Committed Event Rate for Different GVT Periods, no Remote Events

of 0% and 10% and the EPC value of 0. Larger GVT values reduce the frequency of checkpoint creation and GVT update cycles, but increase the memory pressure (because larger amount of state needs to be maintained between consecutive checkpoints) and make individual rollbacks more expensive. On the other hand, smaller GVT intervals require less state information to be maintained for rollback recovery, but increase the overhead of GVT maintenance because GVT iterations are needed more frequently.

As seen from the presented results, in all cases that we evaluated, larger GVT periods resulted in significantly better performance, indicating that GVT maintenance is an expensive operation. For example, according to Figure 9, there is almost 5X performance improvement between the GVT periods of 32 and 512 for 256-way simulation. There is also about 2X performance difference between the GVT periods of 128 and 256. The reason for such drastic impact is that GVT updates in ROSS rely on expensive all-reduce operations that involves full group communication and synchronization across simulation threads to agree on the minimal value of their local virtual times. At the low level, this involves chip-wide cache coherence traffic and OS intervention, leading to significant slowdowns. While more efficient asynchronous implementations of the GVT algorithm are possible [16, 17], their exploration and adaptation for ROSS framework running on KNL is left for future work. The results shown in Figures 9 and 10 lead to the following observation.

Observation 6: Synchronous GVT updates on the KNL architecture are relatively expensive. At the same time, memory capacity is not a limitation since the KNL processor has access to the entire system memory. Therefore, larger GVT intervals provide better performance.

3.5 Impact of Cache Sharing

Next, we investigate the impact on cache sharing between threads on performance. The Knights Landing processor has a two-level cache hierarchy. Each core has its own private Level 1 (L1) cache but shares its Level 2 (L2) cache with a neighboring core as part of a tile. To investigate the performance impact of this cache architecture, we evaluated three communication schemes that we call *same node*, *same tile* and *tile pairs*. In *same node* mode, each thread communicates with another thread running on the same core — this

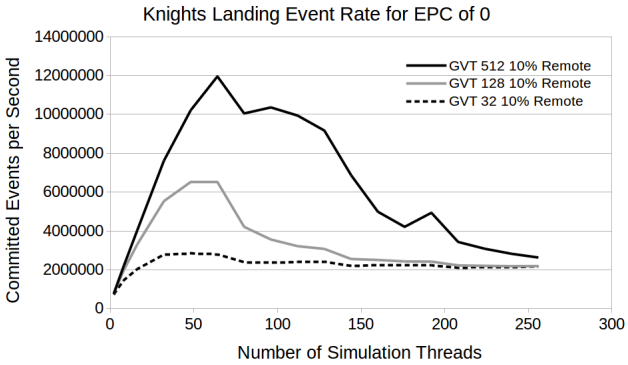

Figure 10: Committed Event Rate for Different GVT Periods, 10% Remote Events

model maximizes the exploitation of the L1 cache. In *same tile* pattern, each thread communicates with a thread running on an adjacent (buddy) core — in this case the data can come from the shared L2 cache. With *tile Pairs*, each thread communicates with a thread running on a core in the next tile, requiring communication beyond the local L1 and L2 caches. We note that to realize *same node* pattern, multiple threads have to be placed on the same core. For example, for a 64-way simulation, 32 cores will have two threads each and the other 32 cores will be unused. Therefore, the positive effect of L1 cache sharing is offset by overutilization of cores and reduced processing throughout in general.

Figure 11 shows the performance of our three communication schemes scaling with at most one thread per core up to 64 cores. All events are generated remotely in this model. As one would expect, communication in *tile pairs* mode is slower than communication that exploits the L1 or L2 cache. However, there is negligible difference between communication utilizing the L1 cache (*same node*) and the L2 cache (*same tile*) — in this case exploitation of L1 caches for faster communication is offset by the extra processing load on the cores. At the same time, the *same tile* model allows for the low-latency communication through L2 caches without core overloading.

Figure 12 extends the data to 256 simulation threads introducing additional core loading. The trends seen above are

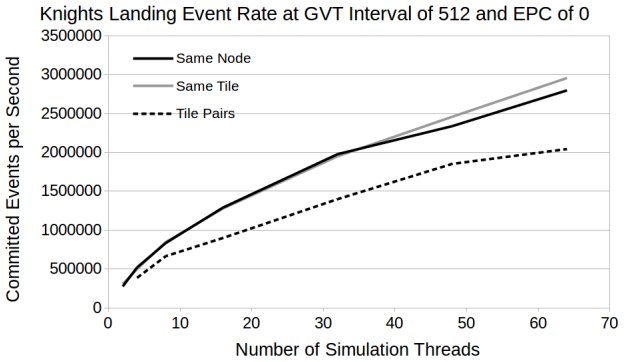

Figure 11: Committed Event Rate for Communication Patterns Using *Same Node, Same Tile and Tile Pairs* Policies at 100% Remote Percentage and EPC of 0 up to 64 Threads

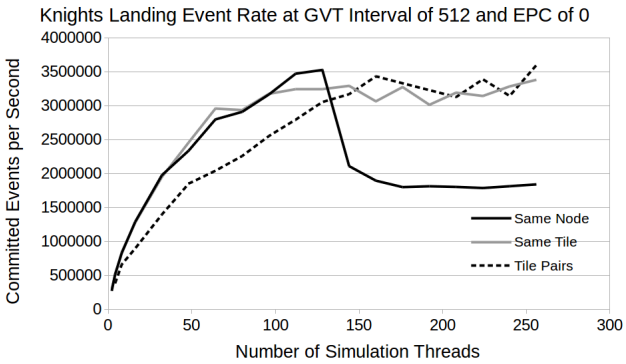

Figure 12: Committed Event Rate for Communication using *Same Node, Same Tile, Tile Pairs and Next Tile* Policies upto 256 Threads

preserved until thread count reaches 128, after that point a different behavior is observed. First, there is a precipitous drop in performance of *same node* pattern (Figure 12). The reason for the drop is that the pressure on a small L1 cache further increases with more co-located threads, resulting in more cache misses and less advantages due to cache sharing. Furthermore, we observe that *same tile* and *tile pairs* converge in performance as the number of threads increases. The following observation summarizes our cache-related experiments.

Observation 7: For simulations up to 64 threads, attempting to exploit L1 cache for low-latency communication by collocating communicating threads on the same core does not result in performance gains due to the extra core loading. However, limiting communication to the same tile without increasing core loading (thus exploiting the L2 cache) improves the performance by almost 50% compared to the case when communications cross the tile boundary. As the thread count further increases and L2 cache capacity is exceeded, the advantages of tile-level communication are lost.

3.6 Impact of Partitioning, Clustering and Message Flow

As demonstrated in previous sections, remote communication can have a significant impact on performance. One of the approaches to minimize amount of remote communications is model partitioning that localizes communication to small cluster of nodes [18]. In this section, we evaluate the impact of such partitioning on performance. We varied our simulated configurations from 128 2-node clusters to a single 256-node cluster, the latter being equivalent to non-clustered operation considered in previous sections. We evaluated three message flow algorithms within these clusters - a directional flow pattern, a random message flow pattern, and a bidirectional flow pattern.

The directional pattern sends all message traffic to the next node in the cluster with the final node in the cluster wrapping messages back to the first. In the phold model, after a message is sent at initialization, messages are only sent after being received. Thus, this represents a very balanced and uniform message flow pattern.

Conversely, the random message flow pattern has an equal chance of the message destination being any node within the cluster. Should the destination end up being the source node, a different node in the cluster is picked using the direction pattern outlined above. This is done to ensure message traffic is to a different core and is thus exercising the same communication path.

The bidirectional pattern alternates message traffic between previous and subsequent nodes in the cluster with the destinations decrementing and incrementing respectively. The destinations are wrapped around to remain within the cluster. Similar to directional flow, this represents a very balanced and uniform message flow pattern, but also utilizes more communication channels.

As seen from the results of Figure 13, clustering offers as much as 3.5X speedup when compared with random non-clustered communication. We also observe that message flow has an impact on communication performance, although less than the effects of cluster size. At the point of 2 nodes per cluster, the destination is the same regardless of the algorithm as there is only one valid destination. Interestingly, directional flow is the slowest of the three at this cluster size due again to increased GVT all reduce processing. However, as the node size increases beyond 4 nodes per cluster, directional communication provides a performance advantage over random and bidirectional communication, resulting in a 25% gain even at 256 node cluster size (this point represents non-clustered communication). In summary, these results lead to the following observation:

Observation 8: Simulation model partitioning and limiting communication to a small number of adjacent cores can lead to significant improvement in simulation performance, especially for small cluster sizes, upto 3.5X in some scenarios. This is achieved by only limiting the size of communicating clusters, and not reducing the remote percentage (which still remains at 100% for these results).

3.7 Impact of Communication Distance with at Most One Thread per Core

Additionally, for the scenarios with at most one thread per core, we implemented two policies for placing threads

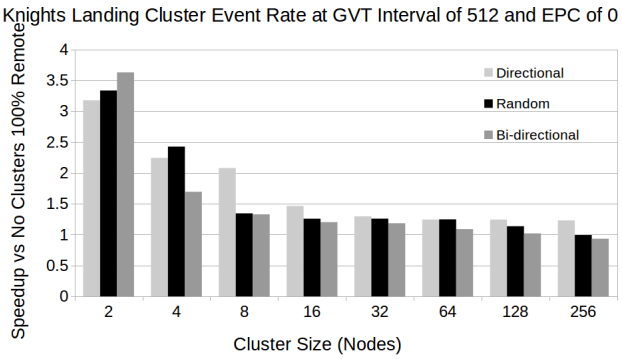

Figure 13: Committed Event Rate for KNL for Different Message Traffic Patterns Within Clusters of Varying Sizes

among the cores: *distant* placement and *nearby* placement. In the distant policy, the communicating threads are placed as far from each other as possible thus requiring the longest number of links to be traversed on the mesh interconnect for communication. Conversely, in the nearby policy, the threads are placed as close as possible to each other to minimize the number of link traversals in the mesh. The intent of these experiments is to gauge the impact of thread placement on the interconnect performance.

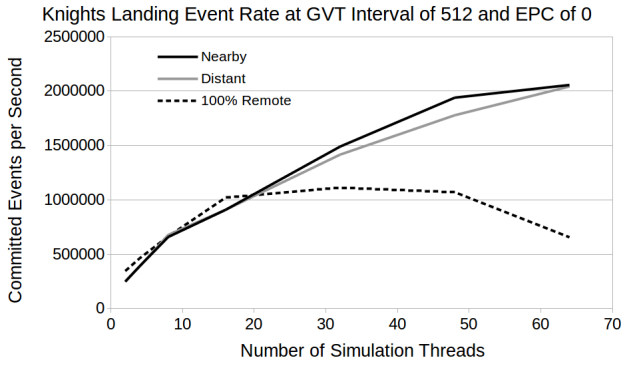

Figure 14: Committed Event Rate for KNL for *Nearby and Distant* Placement at 100% Remote Communication and EPC of 0

We compared the performance of two placement policies at 100% remote events, as this provides the highest communication intensity between threads and stress-tests our scenarios. As seen from the results of Figure 14, thread placement has little effect on the committed event rate. We observe that even when communication between threads has to traverse the entire length of the interconnect, there is no change in performance. This alludes to the high performance nature of the Knights Landing interconnect design. Similar conclusions were made in the study of [7] for the Tilera processor.

Observation 9: The mesh interconnect is not the limiting factor in performance and is not saturated during PDES execution even under stress-testing experiments with distant placement described above. This also corroborates previously reported conclusions for Tilera many-core chip [7].

3.8 Performance Comparison with Knights Corner and 12-core Xeon

Intel Xeon Phi architectures provide interesting opportunities to partition, map, and execute PDES simulations on many cores within the same chip. However, while demonstrating impressive speedups against own sequential execution, some previous manycore architectures failed to match the raw performance achieved on a commodity Xeon processor. In this section, we directly compare the performance of PDES on a Xeon dual socket 12-core system with Knights Corner and Knights Landing Xeon Phi architectures.

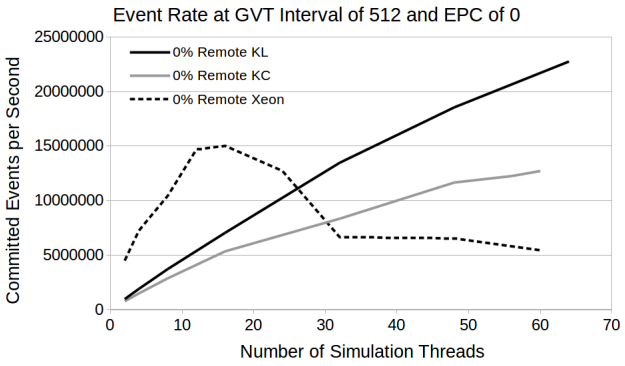

Figure 15: Performance Comparison of KNL, KC and 12-core Xeon for 0% Remote Events

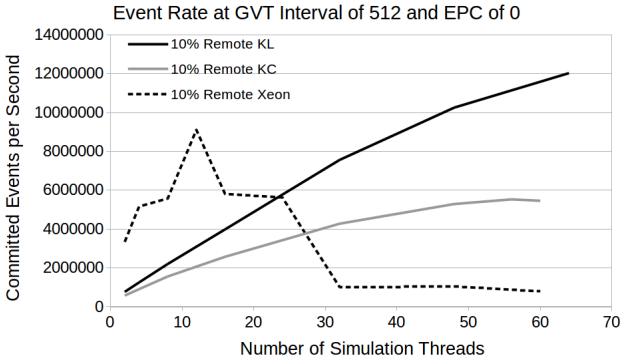

Figure 16: Performance Comparison of KNL, KC and 12-core Xeon for 10% Remote Events

Figures 15 and 16 show the performance of PDES on the three processors for 0% and 10% remote communication respectively. These results are shown for up to 64 threads, we expand to more threads later. As expected, the Xeon peaks at 12 nodes and begins a steeper decline at 24 threads when its hardware thread count is exceeded (Xeon supports 2-way hyperthreading). KNL, as reported earlier in this paper, scales almost linearly to 64 threads. KC, though linear, fails to achieve the peak event rate of Xeon resulting in 40% lower performance for the 10% remote case. Conversely, KNL achieves a 32% performance increase over Xeon for the 10% remote communication case and 51% increase for simulations with no remote events.

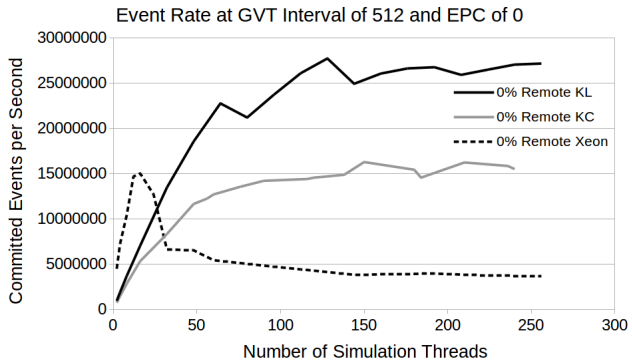

Figure 17: Performance Comparison of KNL, KC and 12-core Xeon

As we scale past 64 nodes, shown in Figure 17, KC and KNL continue to provide performance increases up to 128 threads. It is only at this point that KC matches the performance of the 12-core Xeon. At 128 threads, KNL expands its performance gain over the Xeon to 84%, and the performance of KNL stays about the same as the number of threads grows beyond 128. These results are summarized by the following observation.

Observation 10: Xeon Phi architectures are well suited to PDES simulation. Though Knights Corner processor exhibits at best equal performance to commodity Xeon processors, Knights Landing provides almost a 2X performance increase and retains scalability to high number of threads.

4. RELATED WORK

The works of [5, 6] investigated several optimizations within multithreaded PDES simulator to achieve scalability on relatively smaller-scale platforms such as Intel's Core-i7 and AMD's Magny-cours. The work of [7] investigated PDES performance on the Tilera processor, whose architecture is close in spirit to the KNL design studied in this paper. While the results of [7] demonstrate excellent scalability of the simulation and the capability of the interconnection network to sustain high throughput even under heavy pressure, no comparison against the simulation results on a traditional high-frequency multicore processor were provided. Therefore, it is difficult to gauge the practicality of using the Tilera system for running PDES based on these results and relatively low performance of Tilera cores. Our study corroborated some of the results achieved on Tilera, specifically that longer communication distance on the interconnect do not impact ROSS performance.

In the supercomputing domain, Bauer et al. designed scalable PDES for IBM's Blue Gene supercomputer [4]. To achieve almost linear speed-up, they rely on reverse computation - a mechanism that replaces state saving that is built into ROSS simulator. Reverse computation essentially undoes the computations that need to be rolled back by performing reverse operations (for example, to roll back addition, subtraction is performed). If the rollbacks are infrequent (as in the case of the models considered in [4]), reverse computation is more efficient than state saving, because the overhead of computing GVTs and creating checkpoints is eliminated. In this paper we evaluate a range of models, including the ones with high remote percentage rate, high rollbacks and low efficiency. Consequently, we do not use reverse computation, but rely on traditional state saving mechanism which is implemented in ROSS.

The work of [19] is the follow-up to [4], reporting impressive event processing rates on Sequoia BlueGene/Q supercomputer. The recent effort of [10] evaluated PDES performance on Knights Corner processor. The main conclusion of [10] is that Knights Corner does not outperform the host Xeon processor in terms of event rate unless vector units are fully utilized, and increasing the number of threads does not alter that trend - this is corroborated by our comparison results shown in Section 3.8. The reasons behind such subpar performance are slower in-order cores and limited amount of physical memory on the accelerator card.

Several other studies investigated the performance of various parallel applications on Xeon Phi (Knights Corner) platforms [20, 21, 22, 23, 24, 25]. However, all of these applications are very different from PDES and in general offer more parallelization opportunities. Evaluating PDES on KNL provides an insight of how similar fine-grain communication-dominated applications will be expected to perform on these platforms.

5. CONCLUDING REMARKS

In this paper, we performed a comprehensive characterization of PDES performance on the Intel Knights Landing processor in an effort to understand whether this architecture is an attractive hardware platform for running distributed simulations. Specifically, we studied performance sensitivity of PDES to many key simulation parameters, including percentage of remote events, event processing granularity, GVT period, thread placement strategies and communication patterns. In general, the results of our evaluations are very promising demonstrating significant speedup compared to sequential execution (up to 44X) and also performance improvements compared to both Knights Corner and commodity Xeon processors (up to 2X).

Our results indicate that the best performance on KNL processors is achieved when thread affinity control is used, at most two threads are placed on each core and physical cores are evenly balanced. We showed that simulation scales well even in the presence of significant fraction of events that are generated remotely and that scaling continues for larger thread counts with increased event processing granularity. We quantified the benefits of localizing communication to smaller clusters of cores and showed that a performance improvement of up to 3.5X can be achieved compared to all-to-all communication models. We also demonstrated that GVT computation on a KNL system is expensive thus favoring simulation setups with larger GVT periods.

Our future work will consider extension of this study to clusters of KNL processors and exploitation of the specifics of memory organization and clustering modes.

6. ACKNOWLEDGMENTS

This material is based upon work supported by the AFOSR under Award No. FA9550-15-1-0384 and DURIP award FA9550-15-1-0376.

7. REFERENCES

[1] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, "GTW: a Time Warp system for shared memory multiprocessors," in *Proceedings of the 1994 Winter Simulation Conference* (J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, eds.), pp. 1332–1339, Dec. 1994.

[2] R. M. Fujimoto and M. Hybinette, "Computing global virtual time in shared-memory multiprocessors," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 4, pp. 425–446, 1997.

[3] R. Fujimoto and K. Panesar, "Buffer management in shared-memory Time Warp system," in *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS 95)*, pp. 149–156, June 1995.

[4] D. Bauer, C. Carothers, and A. Holder, "Scalable time warp on bluegene supercomputer," in *Proc. of the ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS)*, 2009.

[5] D. Jagtap, N.Abu-Ghazaleh, and D.Ponomarev, "Optimization of parallel discrete event simulator for multi-core systems," in *International Parallel and Distributed Processing Symposium*, May 2012.

[6] J. Wang, D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev, "Parallel discrete event simulation for multi-core systems: Analysis and optimization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1574–1584, 2014.

[7] D. Jagtap, K. Bahulkar, D. Ponomarev, and N. Abu-Ghazaleh, "Characterizing and understanding pdes behavior on tilera architecture," in *Workshop on Principles of Advanced and Distributed Simulation (PADS 12)*, July 2012.

[8] G.Chrysos, "Intel xeon phi x100 family coprocessor - the architecture," in *Intel white paper*, 2012.

[9] "Tilera TILE64 processor," 2008. Documentation from Tilera Website http://www.tilera.com.

[10] H. Chen, Y.Yao, and W. Tang, "Can mic find its place in the world of pdes?," in *Proceedings of International Symposium on Distributed Simulation and Real Time Systems (DS-RT)*, 2015.

[11] C. Carothers, D. Bauer, and S. Pearce, "ROSS: A high-performance, low memory, modular time warp system," in *Proc of the 11th Workshop on Parallel and Distributed Simulation (PADS)*, 2000.

[12] R. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, pp. 30–53, Oct. 1990.

[13] D. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 405–425, July 1985.

[14] A. S. amd R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. HUtsell, R. Agarwal, and Y. Liu, "Knights landing: Second-generation intel xeon phi product," in *IEEE Micro*, 2016.

[15] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[16] S. Srinivasan and P. F. Reynolds Jr, "Non-interfering gvt computation via asynchronous global reductions," in *Proceedings of the 25th conference on Winter simulation*, pp. 740–749, ACM, 1993.

[17] G. Chen and B. K. Szymanski, "Dsim: scaling time warp to 1,033 processors," in *Proceedings of the 37th conference on Winter simulation*, pp. 346–355, Winter Simulation Conference, 2005.

[18] K. Bahulkar, J. Wang, N. Abu-Ghazaleh, and D. Ponomarev, "Partitioning on dynamic bahavior for parallel discrete event simulation," in *26th IEEE/ACM/SCS Workshop on Principles of Advanced and Distributed Simulations (PADS)*, July 2012.

[19] P. D. Barnes Jr, C. D. Carothers, D. R. Jefferson, and J. M. LaPre, "Warp speed: executing time warp on 1,966,080 cores," in *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pp. 327–336, ACM, 2013.

[20] A. Ramachandran, J. Vienne, R. Wijmgaart, L. Koesterke, and I. Sharapov, "Performance evaluation of nas parallel benchmarks on intel xeon phi," in *Proceedings of International Conference on Parallel Processing (ICPP)*, 2013.

[21] G. Misra, N. Kurkure, A. Das, M.Valmiki, S. Das, and A. Gupta, "Evaluation of rodinia codes on intel xeon phi," in *Proceedings of the 4th International Conference on Intelligent Systems, Modelling and Simulation*, 2013.

[22] A. Heinecke, K. Vaidanathan, M. Smelianskiy, A. Kobutov, R. Dubtsov, G. Henri, A. Shet, G. Chrysos, and P. Dubey, "Design and implementation of the linpack benchmark for single and multi-node systems based on intel xeon phi coprocessor," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[23] S. Pennycook, C. Hughes, M. Smelianskiy, and S. Jarvis, "Exploring simd for molecular dynamics using intel xeon processor and intel xeon phi coprocessors," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[24] M. Lu, L. Zhang, H. Hyunh, Z. Ong, Y. Liang, B. He, R. Goh, and R. Huynh, "Optimizing the mapreduce framework on intel xeon phi coprocessor," in *Proceedings of International Conference on Big Data*, 2013.

[25] B. Xie, X. Liu, J. Zhan, Z. Jia, Y. Zhu, L. Wang, and L. Zhang, "Characterizing data analytics workloads on intel xeon phi," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pp. 114–115, IEEE, 2015.