Interference Resilient PDES on Multi-core Systems: Towards Proportional Slowdown

Jingjing Wang, Nael Abu-Ghazaleh and Dmitry Ponomarev
Computer Science Department
State University of New York at Binghamton
{jwang36, nael, dima}@cs.binghamton.edu

ABSTRACT

Parallel Discrete Event Simulation (PDES) harnesses the power of parallel processing to improve the performance and capacity of simulation, supporting bigger models, in more details and for more scenarios. PDES engines are typically designed and evaluated assuming a homogeneous parallel computing system that is dedicated to the simulation application. In this paper, we first show that the presence of interference from other users, even a single process in an arbitrarily large parallel environment, can lead to dramatic slowdown in the performance of the simulation. We define a new metric, which we call proportional slowdown, that represents the idealized target for graceful slowdown in the presence of interference. We identify some of the reasons why simulators fall far short of proportional slowdown. Based on these observations, we design alternative simulation scheduling and mapping algorithms that are better able to tolerate interference. More precisely, the most resilient simulators will allow dynamic mapping of simulation event execution to processing resources (a work pool model). However, this model has significant overhead and can substantially impact locality. Thus, we propose a locality-aware adaptive dynamic-mapping (LADM) algorithm for PDES on multi-core systems. LADM reduces the number of active threads in the presence of interference, avoiding having threads disabled due to context switching. We show that LADM can substantially reduce the impact of interference while maintaining memory locality reducing the gap with proportional slowdown. LADM and similar techniques can also help in situations where there is load imbalance or processor heterogeneity.

Categories and Subject Descriptors

I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete event, Parallel*

SIGSIM-PADS'13, May 19–22, 2013, MontrÃľal, QuÃľbec, Canada.

Keywords

PDES, Multi-cores, Proportional Slowdown, Interference

1. INTRODUCTION

Discrete Event Simulation (DES) is an efficient simulation methodology used in domains where changes of state occur at discrete times. It is widely used in a range of application domains such as the simulation of computer and telecommunication systems, war-gaming, transport systems, operational planning and biological simulations. Parallel Discrete Event Simulation (PDES) [12] harnesses the computational power and resources of parallel computing to improve the performance and capacity of DES, allowing the simulation of larger models, in more details, and for more scenarios.

PDES simulators have traditionally been designed under the assumption of a homogeneous environment with no interference from other co-located applications. Interference from other applications as well as other noise in the system creates competition for the available resources leading to potential slow downs in parallel applications [20, 31, 27, 22]. In the presence of interference, we expect an application to slow down proportionately to the reduction in its share of the resources: a metric introduced in this paper which we call *proportional slowdown*. Surprisingly, we found the impact of interference on PDES to far exceed proportional slowdown, even when the amount of interference is small. For example, when evaluating a multi-threaded fixed-mapping PDES engine, we discover that even 1 external load can result in a performance slowdown of a factor of up to 3.9 for an 8-way simulation on the core i7 platform, and up to 2.8 for a 48-way simulation on an AMD Magny-Cours platform.

A primary reason for the disproportionately high cost of interference is related to the granularity of the operating system (OS) scheduler. In particular, when the OS schedules an interfering process on a core, it has to context switch one of the simulation threads out, making it inactive. As a result, this thread is stalled, while, assuming optimistic simulation, other simulation processes surge forward. Eventually, when the OS schedules the process again, its late events cause rollbacks throughout the simulation: thus, most of the time on all processes is lost, and additional inefficiency results from the overhead of rollbacks. The problem continues to occur whenever the noise process is scheduled. Describing and characterizing this behavior in the presence of noise is one of the contributions of this paper.

It is important to note that this problem differs significantly from load-balancing problem that can be solved with dynamic object migration [23, 15]: unless all objects are mi-

grated away from a context switched thread, the problem persists. Moreover, the problem happens at the granularity of the OS scheduler: a thread stops execution completely while it is context switched out, only to return to normal execution later. The victim thread typically changes based on affinity settings and OS scheduler decisions.

Carothers et al. noted that interference from external loads can pose significant performance slowdown [7], but did not elaborate on the reasons, or propose solutions to this problem. More recently, Malik et al. [19] studied the problem of executing PDES in a cloud environment where many applications are scheduled on the same infrastructure and co-interfere with each other. The impact of the problem was extensive, and they proposed the use of a master-worker model to provide flexibility of assigning available computational threads to execute the available simulation events. Flexible mapping of workers to work can provide resilience to interference as well as load imbalance, but could result in poor locality as simulation state moves frequently between hardware threads, rather than staying with one thread, and reaping the benefits of caching. To the best of our knowledge, no prior work in PDES has analyzed the impact of interference on PDES, explained the slowdown, or proposed solutions to it.

The goal of this paper is to develop alternative organizations to PDES simulation that are more resilient to the impact of external interference. In conventional PDES implementations a simulation model is partitioned across multiple processing elements (PEs). Each PE (a group of LPs) is executed by a process (or thread). In many existing PDES simulators such as ROSS [3] and WarpIV [30], the mapping between PEs and processes (or threads) is established at the initialization of simulation, and does not change during the simulation (the so called fixed-mapping, or FM, scheme). FM can achieve optimal performance in a load balanced simulation in the absence of external interference, primarily because it promotes locality of memory references, and because it incurs little overhead for scheduling [14]. However, FM suffers in the presence of interference because a stalled thread remains responsible for simulating its objects leading to poor performance.

In this paper, we explore interference-resilient execution of PDES on two multi-core platforms: a quad-core Intel core i7 system, and a 48-core AMD Opteron Magny-Cours platform with Non-Uniform Memory Access (NUMA) latencies. We first propose a dynamic-mapping (DM) scheme that is capable of dynamically changing the mapping between PEs and threads during the simulation. In particular, each thread attempts to work on a PE in a round-robin fashion. For correctness, each PE can only be mapped to one thread at a time. As a result, DM has limited opportunities to solve the problem: a thread is often switched out while in the middle of processing events on a PE. Other threads thus cannot assist and execute the PE until the thread gets scheduled again and releases the lock, causing the PE to lag far behind of others. Thus, although some performance benefit can be obtained, we discovered that the baseline DM cannot effectively solve the interference problem.

To address this problem, we propose an adaptive DM scheme that reduces the number of active threads when interference is detected. As a result, the number of threads is again matched to the available hardware contexts, and the simulation does not have to suffer extended periods when one of its threads is switched out. In this context, the remaining threads have to service a number of PEs that is larger than them. Having the threads switch in round robin fashion among the PEs, promotes load balanced operation but leads to poor locality as PEs move among threads causing cache interrogation. More precisely, the Locality Aware Adaptive DM (LADM) scheme creates a schedule where each thread is primarily associated with one PE, but spends a portion of its time helping one other PE. The proportion of time is chosen so that the total active time each PE receives remains balanced, avoiding straggler objects. Since each thread works on a limited number of PEs (two under reasonable interference conditions), locality is kept high.

LADM has the following key characteristics:

1. In the absence of external loads, LADM incurs small performance loss (less than 5%) compared with the optimal FM implementation on both the Intel core i7 and AMD Magny-Cours machines. The loss includes the overhead of detecting interference, but also the cost of rate misprediction of interference; a problem we hope to address with more careful design of the detector.
2. LADM can substantially reduce the impact of interference, thus reducing the gap with proportional slowdown.
3. LADM can successfully detect performance anomalies caused by external loads during the simulation without user interaction.
4. LADM exhibits good locality of memory references on both multi-core platforms.

As a result, LADM is able to achieve 2-4X improvement in performance in the presence of interference on both a 4-core (8 hardware threads) Intel core i7 and a 48-core AMD Magny-cours machine.

The remainder of the paper is organized as follows. Section 2 provides background information regarding both PDES simulator and two multi-core platforms we used in our experiments. We then define *proportional slowdown* to quantify the PDES performance in the presence of external loads in Section 3. In Section 4, we show the actual impact of external loads on the performance of fixed-mapping PDES simulators. We then explain why the performance of PDES simulator with FM implementation suffers considerably when the simulation is interfered by external loads. In Section 5, we provide a design overview of the baseline DM mechanism. In Section 6, we provide the details of our LADM scheme that can address the limitations in the baseline DM implementation. Section 7 presents an experimental evaluation. In Section 8, we overview some related work. Finally, in Section 9 we present some concluding remarks.

2. BACKGROUND

In this section, we first overview the multi-threaded simulator used in this paper. We follow by providing an overview of the two multi-core platforms used in the experiments: a quad-core Intel core i7 system, and a 48-core AMD Opteron Magny-Cours.

2.1 ROSS-MT: A state-of-the-art Multi-thread PDES Simulator

We use a recently developed multi-threaded version [17] of the Rensselaer's Optimistic Simulation System (ROSS) [3].

ROSS is a state-of-the-art PDES simulation engine which supports both conservative and optimistic simulations. The multi-threaded ROSS (ROSS-MT) encapsulates each group of objects as a PE and assigns each PE to a thread (i.e., it uses Fixed Mapping). The thread-based implementation allows optimizing communication using fast shared memory operations.

In ROSS-MT, a simulation model is partitioned across multiple PEs. Each PE processes the events in time-stamp order, and communicates with others via time-stamped events. In the optimistic simulation, ROSS-MT leverages efficient reverse computation [4], instead of the more conventional state saving [21], to undo incorrectly processed events in case of rollbacks. The events for each PE are executed repeatedly within a simulation loop until the simulation time of the PE reaches the simulation completion time. During each iteration, a designated number of events (batch size) can be processed before moving to the next iteration. Global virtual time (GVT) is computed every GVT interval, which is a configurable variable.

2.2 Multi-core Platforms

We use two multi-core platforms with significantly different CPU and memory organizations. The first is a quad-core Intel core i7 system. In this platform, each core has private 32 KB L1 and 256 KB L2 caches, and shares 8MB L3 cache with other cores. With Hyper-Threading enabled, each core can simultaneously execute two hardware threads which share both L1 and L2 caches. The second architecture we use is an AMD 48-core machine. It consists of four AMD Opteron 12-core chips, connected with Hyper-Transport links. Each chip has two dies, with each die holding six cores. Each core has private 64 KB L1 and 512 KB L2 caches, and shares 6MB L3 cache with other cores on the same die. In addition, the memory accesses to different memory regions on this platform have non-uniform memory access (NUMA) latencies [9].

3. IDEAL SLOWDOWN UNDER INTERFERENCE

Consider a PDES simulation running with N_p threads on a multi-core platform. Let N_c be the total count of hardware threads such that all these threads can execute at the same time; hardware threads refers to cores, or hardware contexts in the case of Simultaneous Multi-Threaded (SMT) processors. Suppose that an external interfering load can start and terminate at any time during the simulation. Thus, to measure performance more accurately, we divide the simulation into n small intervals $[X_{j-1}, X_j]$ indexed by j. In addition, let N_{total} be the total number of software threads executing on the machine (i.e., the number of PDES threads, as well as the number of external loads running concurrently) during the interval j. We assume that the operating system scheduler fairly allocates its CPU resources to each thread. In other words, each load obtains $(\frac{N_c}{N_{total}})$ of the available CPU time on average during the interval j, assuming that N_{total} loads compete for N_c CPUs. Therefore, the expected PDES slowdown under such conditions during the interval j is approximated by:

$$S_j = \frac{N_{total}}{N_c} = \frac{N_p + N_e}{N_c} \quad (1)$$

where N_e is the number of external loads running concurrently with PDES during the interval j. Note that the above reasoning assumes that threads are computation bound and are therefore available to run whenever the scheduler schedules them. We call S_j the *proportional slowdown* during the interval j, since S_j increases proportionately to the number of interfering load processes. We assume that N_{total} is always greater than or equal to N_c, and the interference from external loads on PDES performance occur if $N_{total} > N_c$.

The run time of the entire PDES simulation in the presence of external loads can be approximated by adding up the expected run time across all intervals. Let T_j be the execution time required for the interval j of a FM simulation without interference. By multiplying T_j by the corresponding S_j, we obtain $T_j^{'}$, defined as the execution time required for the interval j of the simulation in the presence of external loads. Therefore,

$$T_{ideal} = \sum_{j=1}^{n} T_j^{'} = \sum_{j=1}^{n} T_j \times S_j \quad (2)$$

denotes the ideal runtime of the entire simulation in the presence of external loads. T_{ideal} represents a best case scenario where the presence of interference merely reduces the amount of available resources and results in a slowdown proportional to this reduction. We will show that in practice, the impact is significantly worse than T_{ideal} because of the dependencies between the threads belonging to one application.

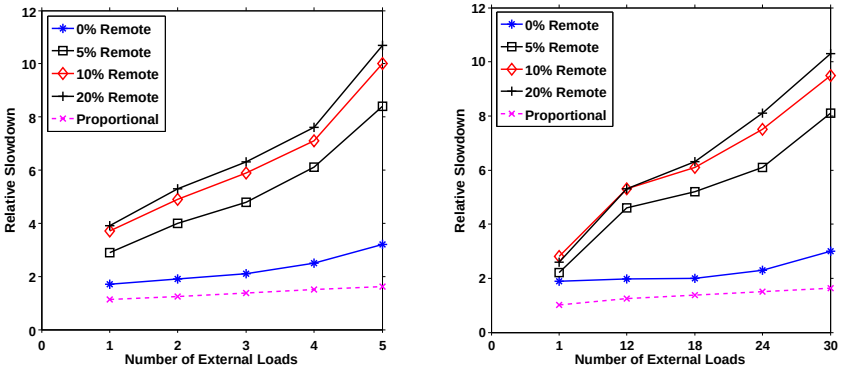
4. MEASURED IMPACT OF INTERFERENCE

In the previous section, we defined *proportional slowdown* as a metric that expresses the ideal slowdown of an application in the presence of interference from external processes. In this section, we evaluate the slowdown experienced by both the ROSS-MT and WarpIV PDES simulators, showing that both far exceed proportional slowdown. In the next section, we start exploring approaches to improve the performance of simulation in the presence of interference.

4.1 PDES Slowdown Under Interference

For most of the experiments, we use the Phold simulation model [13], which equally distributes a number of simulation objects among PEs. We use a controllable version of Phold that allows specifying the communication percentage between different objects on different cores. The simulation consists of 8 PEs running on the Intel core i7 platform, and 48 PEs running on the AMD 48-core machine, with 1000 objects per PE. Each PE was also mapped to a different thread, thus all CPU resources were used by ROSS-MT threads in the absence of external loads. In addition, we selected a GVT interval of 128 on both platforms, with a batch size of 24 events. Although the results are somewhat sensitive to the GVT interval (as a small GVT interval acts as a throttle to the simulation [26]), these values are in the range where ROSS-MT is most efficient across a range of models.

We use a CPU-intensive process as the external load; the process repeatedly performs computation within a tight loop. Thus, the process when active competes continuously for CPU cycles with the ROSS-MT threads. In the experiments, the external load is started with ROSS-MT, and executes for the duration of the simulation. Thus, *proportional slowdown* from 1 external load can be calculated by Equation 1 to be $\frac{9}{8}$ for the core i7 and $\frac{49}{48}$ for the AMD



(a) Intel Core i7 System (b) AMD Magny-Cours System

Figure 1: The Relative Slowdown of ROSS-MT caused by External Loads

Magny-cours. In these experiments, we do not set the CPU affinity for either ROSS-MT or the noise process, providing the OS scheduler complete freedom in scheduling the processing threads to the hardware resources.

Figure 1(a) and Figure 1(b) show the relative slowdown experienced by ROSS-MT as the number of external loads increases, on the Intel core i7 system and AMD Magny-Cours machine respectively. The relative slowdown is calculated by dividing the execution time of simulation in the presence of interference by the one without interference. We show these results as the percentage of remote communication is increased, which increases the dependencies among the different PEs. ROSS-MT with 0% remote communication performs close to proportional slowdown: since there are no dependencies between PEs, if a PE is delayed it does not affect the progress at other PEs. The OS scheduler does not always context switch out the same thread; thus, all threads make progress with their computation. In contrast, the interference from external loads dramatically degrades the performance of ROSS-MT even when a small amount of remote communication exists, far beyond proportional slowdown. For example, even 1 external load can result in a performance slowdown in ROSS-MT of a factor of up to 3.9 on the Intel core i7 machine, and up to 2.8 on the AMD Magny-Cours machine; these values far exceed the ideal slowdown of 1.125 and 1.02 for the core i7 and the Magny-cours respectively.

The problem is not specific to ROSS: we were able to demonstrate similar trends, and even worse slowdown, on the WarpIV PDES simulator [30]. Table 1 show 4-way optimistic and conservative simulations interfered by 1 external load on a quad-core processor; due to export control restrictions on WarpIV, we had to run this experiment on a quad-core Xeon machine. Somewhat surprisingly, the simulation almost stops when the external load takes 100% of the time on one core (a situation which occurred some times, a decision that the linux scheduler makes). We believe this situation is due to the fuzzy barrier used in GVT computation in WarpIV [16]. At any given time, one thread is not executing and the fuzzy barrier condition is not met. However, even when the external load gets a lower scheduling priority and shares one of the CPU cores with a PDES process, the WarpIV simulation still experiences a performance slowdown of a factor of about 2. The situation was the same for both conservative and optimistic simulation.

Table 1: Execution Time of a 4-way Simulation on a Quad-core Processor using WarpIV Simulator

	Optimistic	Conservative
No External Load	6 sec	10 sec
1 External Load takes 50% CPU of a core	12 sec	19 sec
1 External Load takes 100% CPU of a core	$>$ 4.7 hours	$>$ 40 hours

4.2 Explaining the Impact of Interference

Table 2 and Table 3 show the efficiency of ROSS-MT achieved on the two multi-core platforms. The interferences from even one external load substantially reduces the efficiency of the simulation (from around 95% to around 61% on the Intel core i7 platform). Additional interfering processes further degrade efficiency.

To understand the drop in efficiency and the resulting slowdown, we first explain the event processing mechanism within ROSS-MT. As is typical with most PDES simulators, each thread is assigned a unit of work comprising of a group of objects (PE). The groups of objects assigned to each thread are selected, often via a partitioning algorithm (e.g., [2]), to minimize costly communication and to load balance computation. Each thread is responsible for processing all events whose destination is an object in its PE group. Thus, the mapping of work to threads is fixed.

Consider a 2-way simulation of ROSS-MT, with 1 LP per PE, as seen in Figure 2. PE 1 and PE 2 are executed by thread 1 and thread 2 respectively. Suppose an external load starts and interferes with thread 2 at wall clock time t_1, after a GVT computation phase (which requires barrier synchronization in ROSS). Once the interfering noise process is scheduled, thread 2 is context switched out and stops execution, while thread 1 continues. Thread 2 does not get scheduled again until the noise process exhausts its OS quantum (or otherwise, some other hardware context becomes available); the OS quantum is typically in the 10s of milliseconds, sufficient for Thread 1 to execute several million CPU cycles. At a wall clock time t_2 ($t_2 > t_1$), thread 2 resumes execution, and PE 2 sends an event e_1 to PE 1. Due to the large pause in execution, this event is most likely a straggler as PE 1 has executed far ahead of PE 2 limited

Table 2: Efficiency of 8-way Simulation on the Intel Core i7 machine

Remote Communication (%)	Number of External Loads					
	0	1	2	3	4	5
0	100%	100%	100%	100%	100%	100%
5	94.6%	61.0%	51.4%	48.0%	45.3%	42.3%
10	96.3%	51.6%	46.0%	42.9%	40.8%	38.3%
20	96.8%	49.8%	43.6%	40.9%	38.8%	36.8%

Table 3: Efficiency of 48-way Simulation on the AMD Magny-Cours machine

Remote Communication (%)	Number of External Loads					
	0	1	12	18	24	30
0	100%	100%	100%	100%	100%	100%
5	95.9%	78.5%	47.0%	44.5%	45.4%	42.1%
10	96.4%	65.7%	41.7%	39.5%	39.0%	38.0%
20	96.9%	66.5%	41.5%	39.0%	36.6%	35.9%

only in the ROSS case by the GVT computation interval; in other simulators, the degree of optimism can be unbounded.

Upon receiving e_1, PE 1 is rolled back to a simulation time before that of e_1, and then is re-executed. Thus, not only is processing time lost at PE 2 while it is context switched out, but most of the time available to PE 1 is also wasted, which explains why the slowdown exceeds proportional slowdown. The overhead of large rollbacks in terms of state restoration (or reverse computation), sending anti-messages, and other data structure restoration exacerbates the inefficiency. This effect exists whenever any of the simulation threads is context switched out, leading to the type of slowdown that we observe.

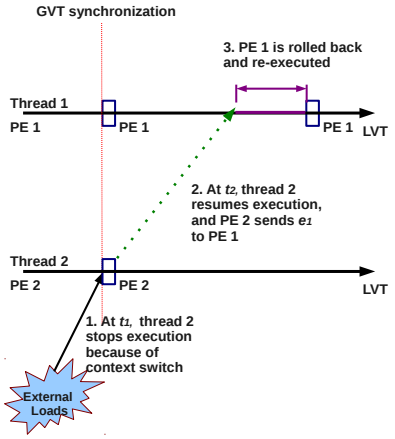


Figure 2: A Rollback caused by Interferences from External Loads

Note that the effect also holds if we use conservative simulation. A thread that is context switched out is not able to update the lookahead at other LPs, preventing them from proceeding. In fact, this problem generalizes to any parallel application with dependencies.

5. CAN DYNAMIC MAPPING HELP?

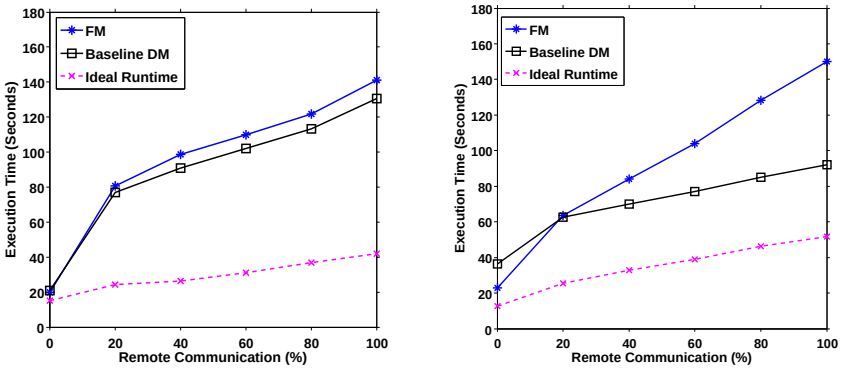
To address the destructive behavior that occurs in the presence of interference, we first attempt *dynamic mapping* (DM) of threads to PEs. More specifically, in this scheme, we periodically remap the threads to different PEs (recall that each PE encapsulates a group of objects in the simulation). The intuition behind DM is that it allows active threads to rotate across the different PEs, avoiding having a PE lag far behind the others.

Recall that each thread in ROSS-MT executes a loop that repeatedly performs the simulation tasks such as sending and receiving events and event processing. To implement DM, we add a new step at the beginning of the loop where a thread determines which PE to associate itself with; the base implementation simply rotates threads in a round-robin fashion across the PEs. Consider the example as shown in Figure 2. After thread 2 finishes the execution of PE 2 for an iteration, it then switches to PE 1. Thus, in principle, the active thread alternates working on PE 1 and PE 2, reducing the LVT difference between them. Alternative basis for scheduling PEs to threads are possible (for example, attempting to work on the PE with the lowest LVT).

Note that a side-effect of remapping threads to PEs is a loss of data locality: FM permanently maps a hardware thread to a unit of work, and the caches for the core are populated with the data relevant to it. As DM remaps work across cores, the PE data must be brought to each new core (from shared lower level caches or main memory).

A second, more serious, limitation of DM is its limited opportunity for assisting performance. More precisely, for correctness, two threads cannot be attached to the same PE concurrently, which prevents remapping from being able to assist if the context switched thread happens to hold the lock on the PE. We implemented efficient synchronization using a condition variable and a spin lock for each PE. More precisely, a PE status is checked (without locking); if the status is busy, the thread moves on to the next PE. If the status is free, the thread acquires the spin lock for the PE, and checks if it is still free. If it is, the thread sets the PE to busy, and is admitted to work on the PE. Once the iteration is over, it sets the PE status to free and moves on again to the next PE.

Thus, DM is limited if the first thread is switched out while in the middle of processing a batch since the PE will be marked as busy until the thread is scheduled again. Since this is the common case, DM cannot effectively solve the problem. Figure 3 shows the performance of original FM ROSS-MT in comparison to the baseline DM version for both the Intel core i7 (Figure 3(a)) and the AMD Magny-



(a) Intel Core i7 System (b) AMD Magny-Cours System

Figure 3: Performance of FM vs. Baseline DM (Interfered by 1 External Load)

Cours (Figure 3(b)) platforms. In particular, the entire simulation is interfered by 1 external load. We find that DM achieves up to 10% performance improvement over FM on the Intel core i7 platform. Moreover, DM can achieve better performance than FM on the Magny-cours only under high remote communication ($> 20\%$). The gap remains substantial with respect to ideal runtime (Equation 2).

Table 4: Efficiency of FM vs. Baseline DM on the Intel Core i7 System (Interfered by 1 External Load)

Remote Comm (%)	20	40	60	80	100
FM	49.9%	47.1%	47.8%	48.6%	46.7%
Baseline DM	50.7%	49.5%	49.8%	51.2%	49.7%

Table 5: Efficiency of FM vs. Baseline DM on the AMD Magny-Cours System (Interfered by 1 External Load)

Remote Comm (%)	20	40	60	80	100
FM	67.0%	62.9%	61.1%	58.5%	56.8%
Baseline DM	87.3%	87.9%	88.5%	88.3%	88.3%

The efficiency of a simulation interfered by 1 external load is shown for both the Intel core i7 (Table 4) and the AMD Magny-Cours (Table 5) platforms. While the results of AMD Magny-Cours system show improvements in efficiency of the baseline DM in comparison to the FM version are observed, the efficiency remains low especially for the core i7. In most cases, DM was not able to help because the context switched thread held the PE lock.

6. LOCALITY AWARE ADAPTIVE DM

DM offers only limited relief from the slowdown experienced in the presence of interference. In addition, DM experiences poor cache locality, because of transient short term association between threads and PEs. In this section, we propose a locality-aware adaptive DM (LADM) scheduler that is capable of addressing limitations of DM.

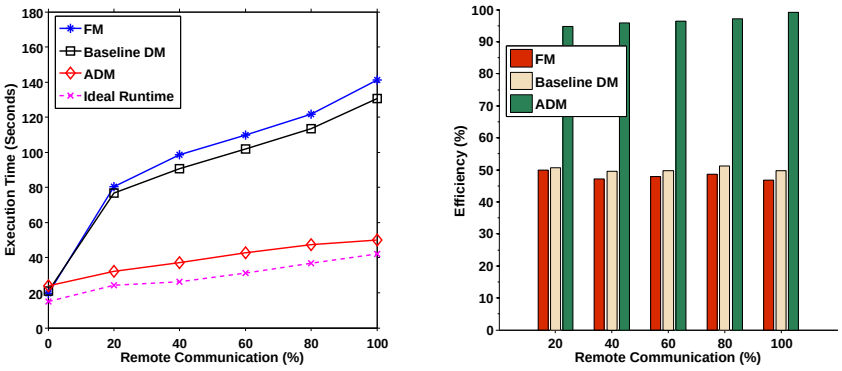
6.1 Adapting the Number of Threads

The first improvement to DM, which we call adaptive DM (ADM), adjusts the number of work threads to the available hardware contexts: when a noise process is detected, the number of active threads is reduced to avoid experiencing expensive context switches. Thus, only active threads are allowed to execute PEs. Supporting ADM requires two main mechanisms: one to detect the presence of interference, and another to adjust the number of active threads. Finally, a third mechanism is required to check if the interference is no longer there and to reactivate idle threads. We discuss these mechanisms in the remainder of this subsection.

The presence of noise is detected as follows. During execution, each active thread periodically monitors its total event processing time (the period is set to $\frac{T_{gvt}}{4}$ simulation loop iterations in our implementation, where T_{gvt} is the GVT interval). The *average processing time per event* (APTE) of each active thread is calculated by dividing the total event processing time by the corresponding number of processed events. A performance anomaly is decided if the rate of maximum APTE to minimum APTE is beyond a user-defined threshold. After the mechanism decides a performance anomaly, the status of the thread with maximum APTE will be configured as "inactive". Each thread checks its status at the beginning of the simulation loop, and inactive threads idle. We discover that the threshold plays an important role in the performance of simulation. If the value of the threshold is too large, then the performance anomaly is not reliably detected when interference from external loads exist. On the other hand, too small a value can cause ADM to incorrectly inactivate a PDES thread in an interference-free environment. We use a threshold of 1.8, which we empirically found to work effectively on both platforms.

The final mechanism checks if the noise has disappeared, and hardware contexts are again available. One inactive thread is re-activated periodically. If noise remains present, then the deactivation logic detects that and deactivates the thread. Thus, the reactivation period must be significantly larger than the detection period to avoid too frequent testing: (we use $10 \times T_{gvt}$, 40 times larger than the detection period).

ADM can reduce the effect of interferences from external loads, and thus significantly improves the performance of the simulation in the presence of external loads. Consider an 48-way simulation interfered by 1 external load on the 48-core AMD Magny-Cours machine, for example. Once a performance anomaly is successfully detected, the simulation is then executed by 47 active threads. The OS scheduler will



(a) Execution Time (b) Efficiency

Figure 4: Performance of ADM on the Intel Core i7 System (Interfered by 1 External Load)

later assign each thread to a different core, thus reducing interferences between PDES threads and the external load.

6.2 Improving the Data Locality

To improve the data locality, we modified the ADM scheduler to increase locality: we call this implementation locality-aware adaptive dynamic-mapping (LADM). Similar to FM, at the initialization of simulation, each thread is assigned to a primary PE, and maintains this assignment in the absence of interference to maximize locality. Once interference is detected and a thread (or more) is deactivated, the PE assigned to the inactive thread is marked as an orphan until such a time where its thread is reactivated. The remaining active threads divide their time between their primary PEs and orphan PEs.

In particular, after each event processing iteration on its primary PE, each active thread checks PEs on the orphan list in a round-robin fashion; it selects an orphan that is currently behind its primary PE in the number of processing iterations (alternatively, LVT may be used). The status of the selected PE is then checked, and the spin lock for it is acquired if its status is free. Once the thread is admitted to work on the PE, it executes N_{batch} iterations before switching back to its primary PE. We set N_{batch} to 10 in our simulations. The thread returns to its primary PE if all the orphan PEs have caught up with it. Unlike ADM, the PEs whose primary thread is active remain exclusively processed by that thread, and only orphan PEs experience a loss of locality.

6.3 The Expected Runtime of LADM

Suppose that the simulator is configured with N_p threads at the initialization of simulation, where N_p equals with the total count of hardware threads on the multi-core platform. In addition, we divide the simulation into n small intervals $[X_{j-1}, X_j]$ indexed by j. Let N_j be the number of external loads running concurrently with PDES during the interval j of the simulation. Once LADM detects N_j ($N_j < N_p$) external loads, the simulation is then executed by (N_p - N_j) active threads during the interval j. The expected runtime of the entire simulation is thus approximated by:

$$T_{expected} = \sum_{j=1}^{n} \frac{N_p}{N_p - N_j} \times T_j \quad (3)$$

where T_j is the execution time required for the interval j of a FM simulation without interference. Moreover, LADM allows at least 1 active thread to execute the simulation if $N_j \geq N_p$.

It is important to note that LADM does not achieve proportional slowdown. ADM schedulers simply give up hardware contexts that are in contention to avoid a situation where they are context switched. Because of this conservative behavior, it is possible for interference loads to crowd-out the simulation threads resulting in significant slowdown under high interference. However, the OS scheduling policy will cause inefficient operation if more threads are running than there are available hardware contexts. To approach proportional slowdown, alternative OS scheduling policies are needed.

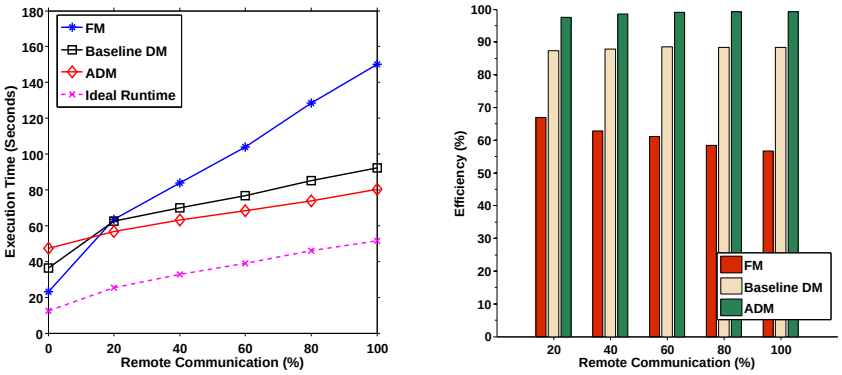
7. PERFORMANCE EVALUATION

This section presents a performance evaluation of ADM and LADM in comparison to FM. Most of the experiments use the Phold benchmark [13], with 1000 objects per PE. For some experiments, we use Personal Communication Services (PCS) system to show that the technique transitions to real models [6].

7.1 Evaluation of ADM

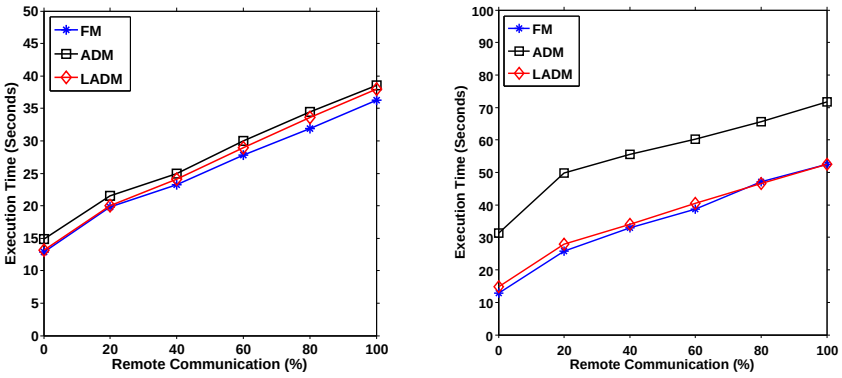
In the first experiment, we evaluate the performance of ADM without the data locality optimization. Figure 4 and Figure 5 show the performance of ADM compared to FM and baseline DM on the core i7 and Magny-Cours platforms respectively. In this experiment, we introduce the interfering load at the start of the simulation, and it runs for the duration. In Figure 4(a) and Figure 5(a), we see the execution time as a function of the percentage of remote communication (the communication between PEs). ADM achieves better performance than FM on the core i7, but only outperforms FM at high remote communication ($\geq$ 20%) percentages.

The behavior can be partially explained by the high cost of lower level cache accesses on the Magny-cours relative to the core i7. At 100% remote communication, for example, ADM can achieve a speedup of 2.8X against FM on the core i7 machine, and 1.8X on the Magny-Cours platform. ADM performs closer to the ideal runtime (predicted by proportional slowdown) on the core i7 platform than on



(a) Execution Time (b) Efficiency

Figure 5: Performance of ADM on the AMD Magny-Cours System (Interfered by 1 External Load)



(a) Intel Core i7 System (b) AMD Magny-Cours System

Figure 6: Performance of Locality-aware Adaptive Dynamic-Mapping Scheme (No External Load)

the Magny-Cours machine. Thus, the benefit from ADM is offset somewhat by the increased cache misses and the poorer locality that results from continuous re-mapping of the work. The locality aware version of ADM attempts to address this issue.

As described in Section 6, the baseline DM will experience poor efficiency when contention becomes heavy, while ADM can reduce such contention by inactivating some threads. To demonstrate this, we present efficiency of corresponding simulations (as shown in Figure 4(b) and Figure 5(b)). Clearly, the baseline DM exhibits a poor efficiency similar with FM on the core i7 machine, but can achieve relatively better efficiency on the Magny-Cours machine. In contrast, ADM can achieve efficiency of over 90% on both platforms.

7.2 The Impact of Data Locality

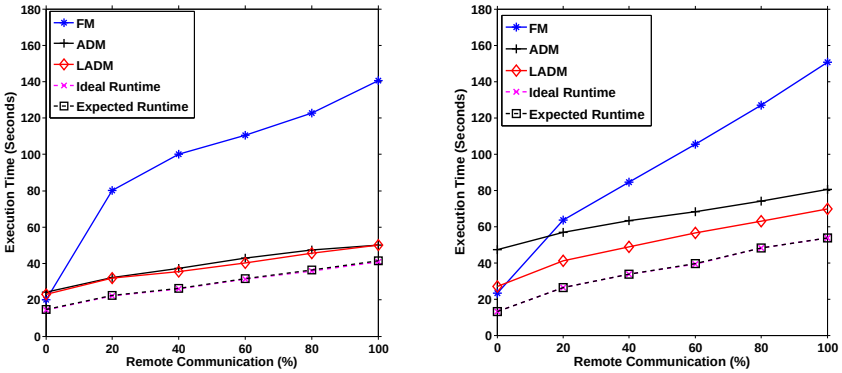
In ADM, each active thread moves to the next free PE in a round-robin fashion at the beginning of the simulation loop, even when there is no interference. Thus, ADM can lead to poor cache locality, as each thread accesses different PEs causing their state to be interrogated between caches. LADM improves data locality by associating threads with primary PEs. Only orphan PEs (those whose primary thread is inactive) experience a loss of locality as their events are processed by the other active threads.

The next experiment evaluates the performance of FM, ADM and LADM in the absence of external loads to measure the overhead of the mechanisms when they are not needed. As seen in Figure 6, LADM performs up to 11% better than ADM on the core i7 machine, and up to 53% on the Magny-Cours machine. In addition, LADM incurs small performance loss (less than 5%) relative to the FM version. The overhead is partially due to the extra checking that LADM does; however, we also noticed that rarely, LADM incorrectly detects the presence of interference. We believe that there is room for improving the interference detection algorithm in future work.

In the next experiment, we consider a scenario with 1 external interfering process (Figure 7). At high remote communication ($\geq$ 20%), LADM outperforms the original FM by a factor of up to 2.8X on the core i7 machine, and up to 2.2X on the Magny-Cours machine. In addition, LADM performs up to 43% better than ADM on the Magny-Cours machine, due to the fact that LADM can achieve better data locality. Figure 8 shows the cache miss rates, demonstrating how LADM has substantially lower cache miss rates than ADM.

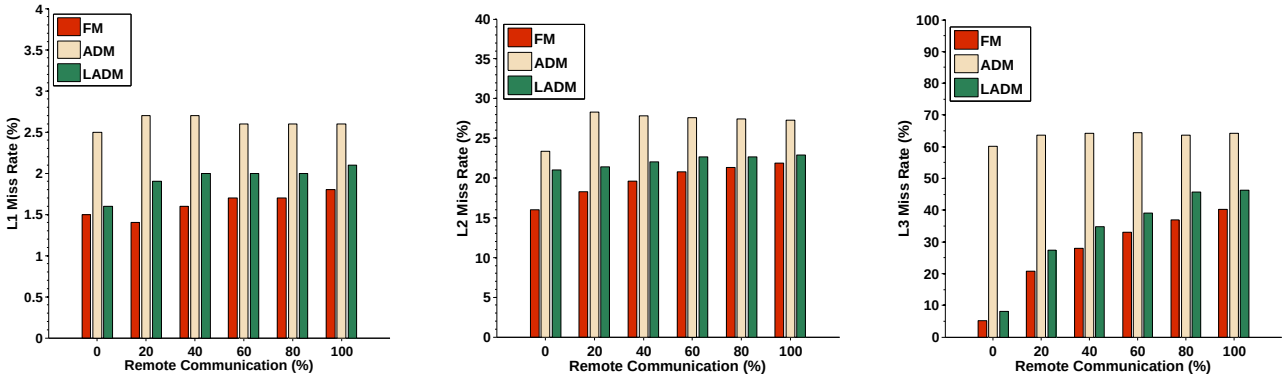
7.3 Impact of Event Processing Granularity

In the next experiment we modify the Phold model to increase the granularity of event processing time. In particular, a new parameter, called *EPC*, is defined to control the amount of computation for each event processing in Phold.



(a) Intel Core i7 System (b) AMD Magny-Cours System

Figure 7: Performance of Locality-aware Adaptive Dynamic-Mapping Scheme (1 External Load)



(a) L1 Cache Miss Rate (b) L2 Cache Miss Rate (c) L3 Cache Miss Rate

Figure 8: Cache Performance of 48-way Simulation on the AMD Magny-Cours System

A higher value of EPC gives more computation load increasing the ratio of computation to communication.

We evaluate the performance of FM and LADM as the number of external loads is increased, for both the Intel (Figure 9) and AMD Magny-Cours (Figure 10) platforms at remote communication percentage of 40%. As seen in Figure 9 and Figure 10, LADM performs better than FM on both platforms when the simulation is interfered by external loads. In addition, the gap with the ideal performance is decreased as EPC increases. Moreover, the gap between LADM and the ideal performance on the Magny-Cours machine is larger than that on the core i7 machine. We believe that the penalty of a cache miss on the AMD Magny-Cours machine is high, due to its NUMA characteristics.

Another interesting observation is that FM performs closer to LADM as EPC increases. We discover that FM is capable of achieving relatively better efficiency at higher EPCs. As each event requires more time for processing in the case of higher EPC, the advance rate of each PE in FM is more balanced than that in the case of lower EPC.

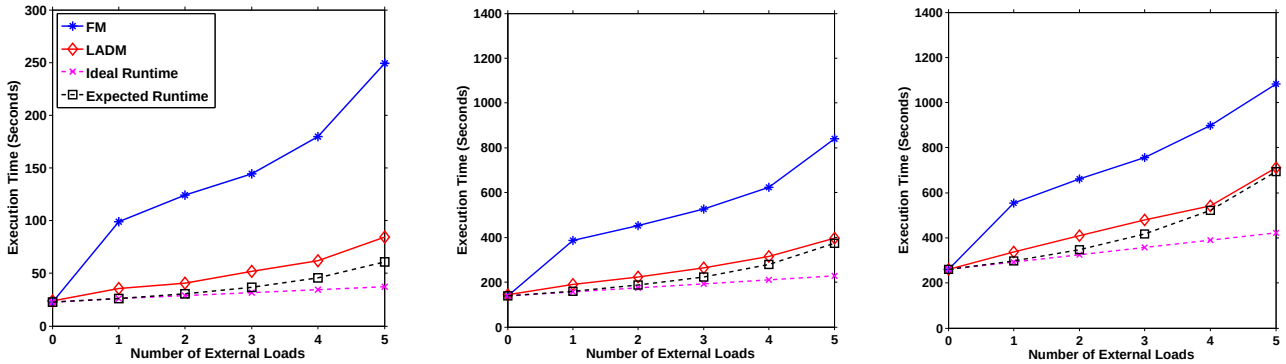
7.4 Performance Evaluation of PCS Model

In this experiment, we study a model of a Personal Communication Services (PCS) system [6]. The PCS model simulates how a cellular provider infrastructure handles a number of mobile phone calls. In this model, an event represents a mobile phone call, sent from one cell phone tower to another. Each cell phone tower has a fixed number of channels. Upon receiving a call, the cell phone tower assigns an available channel to the call, and later releases the allocated channel when the call completes. If all channels are busy, the call is blocked. In addition, the call is handed-off to the destination cell phone tower if the call's connected portable is leaving the area of the cell phone tower [6, 5].

The PCS simulation consists of 36864 cells (LPs) distributed among 8 PEs on the Intel core i7 machine, and 48 PEs on the AMD Magny-Cours machine. Moreover, we fixed the number of channels per cell phone tower at 10. Figure 11(a) and Figure 11(b) show the performance of PCS model in the presence of external loads on the core i7 machine and the AMD Magny-Cours machine respectively. Clearly, LADM performs better than FM on both platforms. At the case of 5 external loads, for example, the performance of LADM exceeds that of FM by a factor of 3.7X on the core i7 machine. In addition, LADM outperforms FM by a factor of about 2.5X at the case of 30 external loads on the Magny-Cours machine.

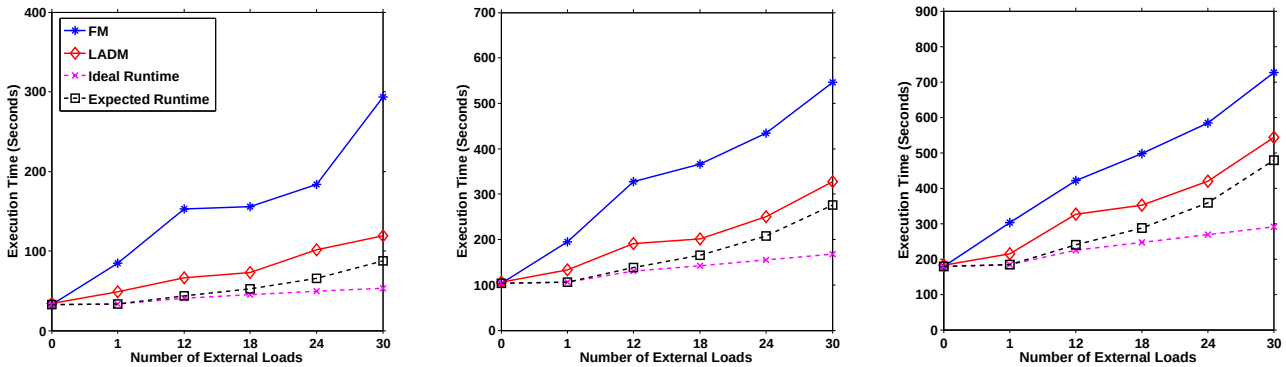
8. RELATED WORK

In this section, we first overview some prior works in the context of PDES. We follow this by describing the interference problem in the general parallel processing community.



(a) EPC=0 (b) EPC=500 (c) EPC=1000

Figure 9: Impact of Event Processing Granularity on the Intel Core i7 Machine



(a) EPC=0 (b) EPC=500 (c) EPC=1000

Figure 10: Impact of Event Processing Granularity on the AMD Magny-Cours System

8.1 Dynamic load-balancing Approaches for PDES

Dynamic load-balancing approaches rely on a monitoring scheme to detect load imbalance, and make dynamic adjustment to improve the performance of simulation. These approaches differ in metrics of detecting load imbalance, and balancing schemes.

Vitali et al. [29, 28] present a load-sharing scheme developed for a symmetric multi-threaded optimistic PDES simulator. Each PE is executed by multiple worker threads, in order to improve parallelism of the simulation. The approach works by allowing a PE that is lagging behind to acquire additional threads to assist with its computation. Thus, the approach is on the face of it similar to our approach in that threads can be redirected to work on lagging PEs. The approach can effectively foster load balanced simulation, but cannot effectively solve the interference problem, as other threads cannot assist when threads keep getting context switched in the middle of event processing.

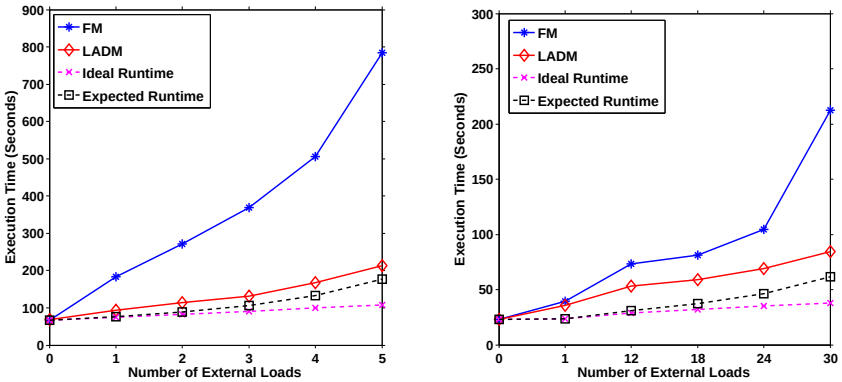
Wilsey et al. [8] proposed a different approach to support run-time core frequency adjustment on many-core systems, with the goal of accelerating the critical path of execution of the Time Warp simulation. To balance workloads of LPs, the cores containing LPs with larger rollbacks are underclocked, while the cores having LPs with smaller rollbacks are overclocked. Though this approach may reduce rollbacks caused by external loads, the performance issue caused by the interference still exists as LPs can't advance if their executing thread is switched out.

Carothers et al. [5, 7] designed a scheme to support background execution of Time Warp. A background central process periodically monitors the workload of each processor, and dynamically determines the set of processors to be used for the Time Warp Simulation. LPs are then distributed across these processors, by using object migration which is widely used in many existing dynamic load-balancing approaches [24, 10, 18]. Dynamic object migration cannot solve the interference problem as well unless all objects are migrated away from a context switched thread.

8.2 Other Approaches to Reduce the Effect of Interference on PDES

In this paper, we demonstrate that the optimistic fixed-mapping PDES simulation kernel can suffer considerably in the presence of interference on the multi-core platforms, due to excessive rollbacks being generated. Malik et al. [19] observed the same behavior present in the cloud environment. To reduce excessive rollbacks caused by interference, they developed a protocol, called *TW-SMIP*, with the goal of identifying straggler messages early and thus avoiding frequent rollbacks.

Replication is another approach that is capable of reducing the effect of interference. As presented in [25], multiple copies of PDES simulation are executed simultaneously on heterogeneous workstation cluster. It allows the runtime re-



(a) Intel Core i7 System (b) AMD Magny-Cours System

Figure 11: Performance of PCS Model

configuration in terms of runtime resource availability, and thus this approach can adapt to interferences from external loads.

8.3 Interference in General Parallel Processing

Similar to PDES, most parallel applications have dependencies between executing threads. Thus, when the interference occurs, active threads have to wait for context switched ones before continuing to execute and the pace of the execution is determined by the slowest thread. As a result, the performance of these applications can be substantially harmed [20, 27, 22]. Two approaches are widely used to balance workloads of threads at run-time: *work-sharing* and *work-stealing*. In work-sharing, when a thread completes its task, it grabs a new one from a central work pool shared across all threads [1]. In contrast, in work-stealing scheme, once a thread finishes its tasks, it steals other threads' tasks [11]. However, neither approach can solve the interference problem unless a context switched thread does not hold any task.

9. CONCLUSIONS

In this paper, we demonstrated the sometimes dramatic slowdown that can result in the presence of external interference. We presented a new metric, called *proportional slowdown*, to measure the idealized slowdown of PDES in the presence of interference and showed that in practice the observed slowdowns far exceed it. We proposed to use dynamic mapping to allow active threads to work on the PEs in a fair way, allowing the simulation to continue to proceed even if one or more threads are context switched. We then proposed a locality-aware dynamic-mapping (LADM) scheme that improves the locality of the proposed adaptive scheme by attempting to keep PEs assigned to their primary threads. Our experimental results showed that LADM is significantly better able to tolerate interference than fixed-mapping implementation, thus reducing the gap with proportional slowdown.

Our future work targets effective approaches for improving the algorithm of scheduling between PEs and threads. In the current study, each active thread looks up a PE on the orphan list in a round-robin fashion. We believe that a better locality can be achieved if the workloads of PEs on the orphan list are equally divided, and each workload is assigned to a specific active thread. We also plan to improve the accuracy of the interference detection algorithm.

10. ACKNOWLEDGEMENTS

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-11-2-0004 and by National Science Foundation grants CNS-0916323 and CNS-0958501. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies and endorsements, either expressed or implied, of Air Force Research Laboratory, National Science Foundation, or the U.S. Government.

11. REFERENCES

[1] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, Nov. 1999.

[2] K. Bahulkar, J. Wang, N. Abu-Ghazaleh, and D. Ponomarev. Partitioning on dynamic behavior for parallel discrete event simulation. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 221–230. IEEE, 2012.

[3] C. Carothers, D. Bauer, and S. Pearce. ROSS: A high-performance, low memory, modular time warp system. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 53–60. IEEE, 2000.

[4] C. Carothers, K. Perumalla, and R. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM TOMACS*, 1999.

[5] C. D. Carothers and R. M. Fujimoto. Background execution of time warp programs. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 12–19. IEEE, 1996.

[6] C. D. Carothers, R. M. Fujimoto, and Y.-B. Lin. A case study in simulating pcs networks using time warp. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 87–94. IEEE, 1995.

[7] C.D.Carothers and R. M. Fujimoto. Efficient execution of time warp programs on heterogeneous,

now platforms. *IEEE Transactions on Parallel and Distributed Systems*, 11:299–317, 2000.

[8] R. Child and P. Wilsey. Dynamically adjusting core frequencies to accelerate time warp simulations in many-core processors. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 35–43. IEEE, 2012.

[9] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30(2):16–29, 2010.

[10] R. Eduardo, D. Grande, and A. Boukerche. Dynamic load redistribution based on migration latency analysis for distributed virtual simulations. In *Haptic Audio Visual Environments and Games (HAVE)*. IEEE, 2011.

[11] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.

[12] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, oct 1990.

[13] R. Fujimoto. Performance of time warp under synthetic workloads. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):23–28, 1990.

[14] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley Interscience, Jan. 2000.

[15] D. Glazer and C. Tropper. On process migration and load balancing in time warp. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):318–327, 1993.

[16] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *Proc. ASPLOS*, pages 54–63, 1989.

[17] D. Jagtap, N.Abu-Ghazaleh, and D.Ponomarev. Optimization of parallel discrete event simulator for multi-core systems. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 520–531. IEEE, 2012.

[18] M. Y. H. Low. Managing external workload with bsp time warp. In *Proceedings of the 2002 Winter Simulation Conference*. IEEE, 2002.

[19] A. W. Malik, A.J.Park, and R. Fujimoto. Optimistic synchronization of parallel simulations in cloud computing environments. In *Proceedings of the International Conference on Cloud Computing*, pages 49–56. IEEE, 2009.

[20] A. Nataraj, A. Morris, A. Malony, M. Sottile, and P. Beckman. The ghost in the machine: observing the effects of kernel operation on parallel application performance. In *Proc. of ACM/IEEE Confernece on Supercomputing*, pages 1–12. IEEE, 2007.

[21] A. Palaniswamy and P. A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *Proc. of the 7th Workshop on Parallel and Distributed Simulation (PADS 93)*, pages 127–134. Society for Computer Simulation, July 1993.

[22] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Proc. of ACM/IEEE Confernece on Supercomputing*, page 55. ACM, 2003.

[23] P. Reiher and D. Jefferson. Virtual time based dynamic load management in the time warp operating system. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 103–111, 1990.

[24] V. Sachdev, M. Hybinette, and E. Kraemer. Controlling over-optimism in time-warp via cpu-based flow control. In *Proceedings of the 2004 Winter Simulation Conference*. IEEE, 2004.

[25] K. H. Shum. Replicating parallel simulation on heterogeneous clusters. *Journal of Systems Architecture*, 44:273–292, 1998.

[26] S. C. Tay, Y. M. Teo, and S. T. Kong. Speculative parallel simulation with an adaptive throttle scheme. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 116–123. IEEE, 1997.

[27] D. Tsafrir, Y. Etsion, D. Feitelson, and S. Kirkpatrick. System noise, os clock ticks, and fine-grained parallel applications. In *Proc. of ACM/IEEE Confernece on Supercomputing*, pages 303–312. ACM, 2005.

[28] R. Vitali, A. Pellegrini, and F. Quaglia. Assessing load-sharing within optimistic simulation platforms. In *Proceedings of the 2012 Winter Simulation Conference*. IEEE, 2012.

[29] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 211–220. IEEE, 2012.

[30] WarpIV Technologies (J. Steinman et al). The warpiv parallel simulation kernel version 1.5.2, 2008. Software available from `http://www.warpiv.com/`.

[31] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proc. of ASPLOS*, pages 129–142. ACM, 2010.