The Performance Implication of Task Size for Applications on the HPX Runtime System

Patricia Grubel*§, Hartmut Kaiser†§, Jeanine Cook*‡§, Adrian Serio†§
*Klipsch School of Electrical and Computer Engineering, New Mexico State University,
†Center for Computation and Technology, Louisiana State University
‡Sandia National Laboratories
§STE||AR Group (stellar-group.org)

***Abstract*—As High Performance Computing moves toward Exascale, where parallel applications will be expected to run on millions of cores concurrently, every component of the computational model must perform optimally. One such component, the task scheduler, can potentially be optimized to runtime application requirements. We focus our study using a task-based runtime system, one possible solution towards Exascale computation. Based on task size and scheduler, the overheads associated with task scheduling vary. Therefore, to minimize overheads and optimize performance, either the task size or the scheduler must adapt. In this paper, we focus on adapting the task size, which can be easily done statically and potentially done dynamically. To this end, we first show how scheduling overheads change with task size or granularity. We then propose and execute a methodology to characterize these overheads and dynamically measure the effects of task granularity. The HPX runtime system [1] employs asynchronous fine-grained task scheduling and incorporates a dynamic performance modeling capability, providing an ideal experimental platform. Using the performance counter capabilities in HPX, we characterize task scheduling overheads and show metrics to determine optimal task size. This is the first step toward the goal of dynamically adapting task size to optimize parallel performance.**

Keywords*—*Task Granularity, Task Scheduling, Task Parallelism, HPX, ParalleX

I. Introduction

High performance computer architectures on the path to Exascale are expected to support increased parallelism through higher core and thread/core counts and tighter integration of GPU and GPU-like accelerators, in addition to significant changes in processor core and memory architecture. As these architectures evolve to support massive amounts of concurrency, runtimes and programming models are changing to improve Scalability, Programmability, Performance portability, Resilience, and Energy Efficiency.

There is an ongoing debate in the community on how to achieve all of the above. An underlying hypothesis of the presented work is that achieving the goal of dramatic scalability improvements for contemporary strong scaling impaired applications and future Exascale applications will require a new execution model to replace the conventional communicating sequential processes (CSP) model best represented by the MPI application programming interface. Although this position is controversial and a focus of community-wide debate, the Exascale computing study [2] concluded that a new execution model and programming methodology is required for dramatic scalability improvements in such problems.

Execution models (and runtime systems that implement them) that support scalability in massively parallel systems are beginning to appear in the HPC community [3], [4], [5], [6], [7]. These models and their respective runtime systems aim to support parallelism through massive multithreading where an application is split into numerous tasks or threads of varying size that execute concurrently. Runtime adaptive resource management and decision making is a centerpiece of the scalability strategy. Runtime adaptivity strongly relies on the ability to identify possible decision criteria usable to steer the runtime system parameters in the desired direction, ensuring best possible performance, energy efficiency, or resource utilization for the application. Despite being critical to success, many of these decision criteria have not yet been determined.

An important component of these runtime systems that can be dynamically adapted to optimize performance is the thread scheduler. Many of the runtime systems implement different thread schedulers that result in widely varying overheads for different task granularities [8]. This provides an opportunity to optimize performance by dynamically adapting the thread scheduling algorithm and/or task granularity. The goal of our research is to develop adaptive thread scheduling to improve performance of parallel applications. Achieving this goal requires: 1. Identification of metrics that can be used to monitor performance and signal the need for runtime changes to task size, 2. Design an auto-tuning infrastructure to make dynamic changes to task size, and 3. Perform evaluations of the technique across several platforms and applications. In this paper we report our early results for the first step.

A. Task Granularity

An important influence on thread scheduling overheads is the granularity of the tasks distributed among processors. The granularity of a task is the amount of time the task executes continuously without synchronization or communication. Fine-grained tasks have small amounts of computation between events for communication or synchronization, while coarse-grained tasks perform computations continuously for long periods of time. Fine-grained tasks can help in optimizing load balancing amongst the parallel processors, but if the application is characterized by a very large number of fine-grained tasks, this can cause higher overall overheads for task creation, management, communication and synchronization, as well as contention on resources such as cache hierarchy or the interconnection network. Coarse-grained tasks make it difficult to perform efficient load balancing amongst the processors causing idle time. The solution to these problems appears to use an efficient grain size for the application. For some application classes, this is the solution. However, there are classes of scaling impaired applications, such as graph applications, that inherently employ fine-grained tasks. These types of applications can benefit significantly from thread scheduling mechanisms that adapt at runtime by detecting granularity size (specific to the underlying hardware) and subsequently tuning

either the scheduling mechanisms and/or the task granularity (if possible) to perform more efficiently. Even applications that are not characterized by a large percentage of fine-grained tasks can benefit from automatic task-size detection and tuning at runtime.

The initial steps toward runtime adaptivity include describing the relationship between overheads and task granularity, understanding how schedulers affect performance for different task sizes, and ascertaining metrics that dynamically determine task granularity. Although our initial results indicate that different schedulers optimize performance for different task size, that performance study is outside the scope of this work and will be done thoroughly in the future.

B. HPX Runtime System

HPX is a general purpose C++ runtime system for distributed parallel applications of any scale. We briefly describe its implementation, thread scheduling system, and performance monitoring system.

In order to tackle the Exascale challenges, a new execution model is required that exposes the full parallelization capabilities of contemporary and emerging heterogeneous hardware to the application programmer in a simple and homogeneous way. HPX is designed to implement such an execution model. It represents an innovative mixture of a global system-wide address space, fine-grain parallelism, and lightweight synchronization combined with implicit, work queue based, message driven computation, full semantic equivalence of local and remote execution, and explicit support for hardware accelerators through percolation. HPX has been designed to replace conventional CSP with fine-grained threading and asynchronous communication, thereby eliminating explicit and implicit global barriers and improving performance of parallel applications.

The design of the API exposed by HPX is aligned as much as possible with the latest C++11 Standard [9], the C++14 Standard [10], and related proposals to the standardization committee [11], [12], [13]. HPX implements all interfaces defined by the C++ Standard related to multi-threading (such as *future*, *thread*, *mutex*, or *async*) in a fully conforming way. These interfaces were accepted for ISO standardization after a wide community-based discussion and since then have proven to be effective tools for managing asynchrony. HPX seeks to extend these concepts, interfaces, and ideas embodied in the C++11 threading system to distributed and data-flow programming use cases. Every possible effort was made to keep all of the implementation of HPX fully conforming to C++, ensuring a high degree of code portability and performance portability of HPX applications. For a more detailed description of HPX and its API see [7].

Thread Scheduling System: HPX-threads, also referred to as tasks, are first class objects in the sense that they have an immutable name in HPX's global address space. They maintain a thread state, an execution frame, and a (operation specific) set of registers. HPX-threads are implemented as user level threads utilizing the $M : N$ model (also known as hybrid-threading). Hybrid threading implementations use a pool of N kernel threads to execute M library threads. HPX-threads are cooperatively (non-preemptively) scheduled in user mode by the thread scheduler on top of one operating system (OS) thread (e.g., pthread) per core, also referred to as worker thread. This way, HPX-threads can be scheduled without a kernel transition, ensuring high performance. Additionally the full use of the OS time quantum per OS thread can be achieved even if an HPX-thread is suspended for any reason as other HPX-threads may be executed. The scheduler is cooperative in the sense that it will not preempt a running HPX-thread until it finishes execution or that thread cooperatively yields its execution (ends a thread-phase). This is particularly important since it avoids costly context switches of operating system threads.

The thread manager currently implements different scheduling policies. However all measurements presented here are done using a priority based FIFO (first-in-first-out) scheduling scheme, where each OS thread works from a separate priority queue of tasks. This is very similar to Intel's Thread Building Blocks [14], or Microsoft's PPL [15]. At creation time the thread manager captures the machine topology and is parameterized with the number of resources it can use, the number of OS threads mapped to its allocated resources, and its resource allocation policy (NUMA awareness). By default it will use all available cores and will create one static OS thread per core.

All HPX-thread scheduling policies use a dual-queue scheme to manage threads. The five HPX-thread states are staged, pending, active, suspended, and terminated. An HPX-thread is first created by the thread scheduler as a thread description, and placed in a staged queue. Since staged threads have not yet been allocated a context, they are easily created and can be moved to queues associated with other memory domains with only very small associated memory costs. The thread scheduler will eventually remove the staged HPX-thread, transform it into an object with a context, and place it in a pending queue where it is ready to run. Once an HPX-thread is running, it is in the active state, and can suspend itself for synchronization or communication. HPX-threads that have suspended execution are in the suspended state, waiting for resources or data to become ready to execute; at that time they will be placed back in the pending queue. A thread that has completed execution will enter the terminated state.

For this study we use the Priority Local-FIFO scheduler, a composition of the Priority Local scheduling policy and the lock free FIFO queuing policy. The Priority Local scheduler uses one pending and one staged queue per worker thread with normal priority, has a specified number of high priority dual queues, and has one low priority queue for threads that will be scheduled only when all other work has been done. When looking for work under the Priority Local policy, the thread manager looks in the worker thread's own pending queue first, then in its staged queue. When the worker thread runs out of work in its local queue system, the thread manager searches the local NUMA domain first through other staged queues, then pending queues. If it does not find work, it will search other NUMA domains, starting with staged queues then pending queues Figure 1.

HPX Performance Monitoring System: The performance monitoring system of HPX provides the mechanism to monitor hardware and software behavior through measured performance counters. HPX performance counters are first class objects, each with a global address mapped to a unique symbolic name, useful for introspection at runtime by the application or the runtime system. Through HPX's predefined interface new performance counters can be easily added. HPX uses the PAPI [16], [17] interface to implement HPX hardware counters. HPX counters are easily accessible through an API at runtime or through a command-line interface for post processing performance analysis. In this study, hardware and software performance counters are used to determine task granularity for the experiments and to obtain measurements of system performance. Since the performance counters are available at runtime, the metrics obtained from them can be used for adaptation of the scheduling policies or to vary the grain size for certain classes of applications.

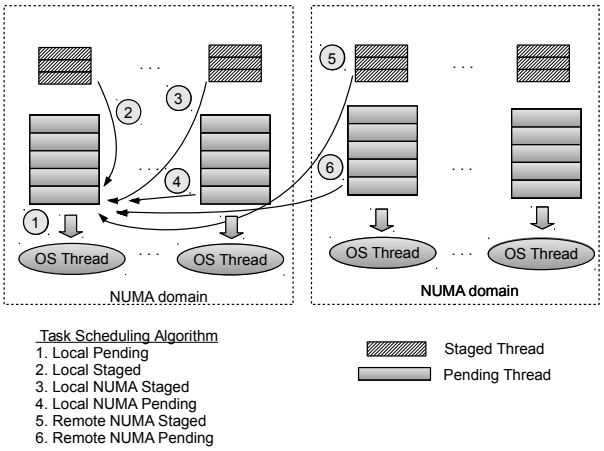


Fig. 1: Schematics of working principle of the Priority Local Scheduler

C. Benchmark

This study uses the one dimensional heat distribution benchmark, HPX-Stencil, (*1d_stencil_4*, available in the HPX distribution package), a representation of the class of scientific applications using iterative kernels. The regular updates in the stencil code provides us with the means to control grain size for our experiments. HPX-Stencil is one of the benchmarks in a series of benchmarks developed to demonstrate the steps in futurization of parallel applications using HPX [7]. The calculation simulates the diffusion of heat across a ring by breaking the ring up into discrete points and using the temperature of the point and the temperatures of the neighboring points to calculate the temperature of the next time step. This dependency is captured in Fig. 2, and explicitly describes the data dependencies captured by the original algorithm. We use the asynchronous threading API of HPX to execute all tasks in proper sequence as defined by the dependency graph. Each of the tasks is launched as a separate (lightweight) HPX-thread using `hpx::async` generating an `hpx::future` that represents the expected result of each of the tasks. The `hpx::future` instances are combined into the dependency tree using the additional HPX facilities to compose Futures sequentially and in parallel. These compositional facilities allow creating task dependencies that mirror the data dependencies described by the original algorithm. Here, the Future objects represent the terminal nodes and their combination represents the edges and the intermediate nodes of the dependency graph.

HPX's lightweight threading system imposes relatively low overhead and allows one to create and schedule a large number of tasks (up to several million concurrent tasks). This efficiency combined with the semantics of Futures allow the direct expression of the generated dependency graph as an execution tree generated at runtime, providing a solid base for a highly efficient auto-parallelization.

In HPX-stencil, the HPX code has been futurized. This means that the dependencies of the calculation have been expressed using futures. The process of futurizing parallel applications using HPX is documented in the HPX manual [18]. In addition, the data points have been split into partitions, and each "partition" is represented with a future. By changing the number of data points in each partition (varying the available input parameters for number of grid points per partition and number of partitions) we can change the number of calculations contained in each future. In this way, we are able to control the grain size of the problem.

This benchmark was chosen because task granularity can be easily controlled, allowing us to use task size as the basis of our experiments and enable us to construct a simple test case in runtime adaptivity. We obtained similar results from micro benchmarks but for brevity they are not included. We are in the process of studying a variety of applications with different workloads.

II. Experimental Methodology

Our goal is to explore how to dynamically adapt task granularity in a programming model that uses fine-grained asynchronous task scheduling mechanisms. In parallel applications, with regular parallel loops, we can easily modify grain size statically to improve performance. We need to be able to determine granularity and adjust it at runtime. To this end, we use HPX-Stencil with its controllable partition size and asynchronous data-flow constructs. Varying grain size from fine-grain to coarse-grain will cause different overheads. Executing applications with millions of fine-grained tasks can cause overheads for thread management and due to contention for queuing and memory resources. While executing coarse-grained tasks can cause overheads caused by poor load balancing and starvation of worker threads. We determine metrics that measure performance behavior, determine granularity and associated overheads then use the facilities in HPX to measure required event counts. This characterization is a first step towards adapting grain size for performance improvement.

The experiments for this study comprise executing the HPX parallel benchmark, HPX-Stencil, described in Sec. I-C, over a large range of partition sizes, to vary granularity, and for an increasing number of cores for strong scaling performance. We compute the heat equation of 100 million grid points for 50 time steps for each data set. When collecting performance and counter data, we make multiple runs and calculate means and standard deviation of these counts. We compute the metrics using the average of the required event counts.

A. Performance Metrics

We compute and analyze numerous metrics. We present only those metrics that are useful to our goal of determining grain size and associated overheads that can be used dynamically to adapt granularity. The metrics and their associated performance event counts are as follows:

Execution Time: We measure the execution time of the heat diffusion for the benchmark to assess performance. To vary grain size, the size of the partition (grid points per partition) is increased and the number of partitions is decreased, so that, for each experiment, heat diffusion is calculated for the same number of grid points.

Thread Idle-rate: The idle-rate event count, */threads/idle-rate*, is the ratio of thread management overhead to execution time. HPX measures $\sum t_{\text{exec}}$, the running sum of times spent on the computation portion of each HPX-thread, and $\sum t_{\text{func}}$, the running sum of total times to complete each HPX-thread. HPX computes idle-rate (I_{r}) as shown in Eq. 1.

$$I_{\text{r}} = \frac{\sum t_{\text{func}} - \sum t_{\text{exec}}}{\sum t_{\text{func}}} \tag{1}$$

In Sec. IV we show idle-rate can be used to make decisions to adjust scheduling mechanisms and grain size to optimize performance by monitoring it at runtime and setting a level of tolerance.

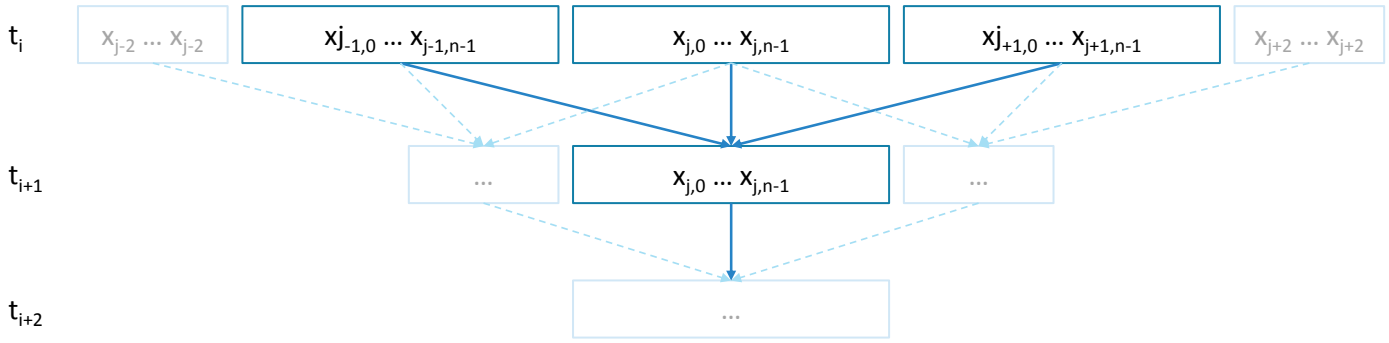


Fig. 2: Dependencies of Heat Distribution Benchmark, HPX-Stencil. Inside each partition, the temperature of a point in the next time step is calculated using the current point's temperature and the temperatures of its neighbors. In order for a partition to be ready to calculate the next time step t_{I+1}, the three closest partitions from the previous time step t_I must have calculated their temperatures.

Task Duration: The average execution time of the computation of an HPX-thread, task duration (t_d) is obtained from the */threads/time/average* HPX performance counter, and computed as shown in Eq. 2. The number of HPX-threads executed, n_t, is also available as counter */threads/count/cumulative*.

$$t_d = \frac{\sum t_{exec}}{n_t} \tag{2}$$

Task Overhead: The average time spent on thread management for each HPX-thread, task overhead (t_o), is obtained from the */threads/time/average-overhead* performance counter, and is computed by HPX's performance monitoring system as shown in Eq. 3.

$$t_o = \frac{\sum t_{func} - \sum t_{exec}}{n_t} \tag{3}$$

Task duration and overhead performance counters were added to HPX as a part of this study and are now available for dynamic measurements at runtime. Additional counters were added to measure average duration and overheads of HPX-thread phases. Each time a thread is activated, either as a new thread or one that has been suspended and reactivated, a thread phase begins. The number of phases, phase duration, and phase overhead can be useful to monitor the affects of suspension and are available as the counters, */count/cumulative-phases*, */threads/time/average-phase*, and */threads/time/average-phase-overhead*.

HPX-thread Management Overhead: We compute the HPX-thread management overhead of the benchmark as shown in Eq. 4. This metric is computed per core (divided by the number of cores, n_c) to be compared with the execution time of the benchmark. Although we calculate this metric for the entire run, for dynamic measurements it can be calculated over any interval of interest.

$$T_o = \frac{t_o * n_t}{n_c} \tag{4}$$

Wait Time: When running on multiple cores the duration of a task can increase due to waiting on resources. We compute the average *wait time* per HPX-thread, as the difference between the measured average task duration, t_d, of each experiment and, t_{d1}, task duration of the same experiment on one core (Eq. 5). *Wait time* is an additional time spent in parallel applications and does not include the overheads caused by task management. We also see in our results that *wait time* can be negative since behaviors such as caching effects can cause the time for one core to be larger than that for multiple cores [8].

$$t_w = t_d - t_{d1} \tag{5}$$

We use Eq. 6 to compute the *wait time* per core for comparison to execution time. For dynamic measurements this metric can be calculated for any interval of the application. This metric requires measurements from running on one core that can be taken at a one time cost prior to data runs or by running a small number of iterations upon initialization of the application.

$$T_w = \frac{(t_d - t_{d1}) * n_t}{n_c} \tag{6}$$

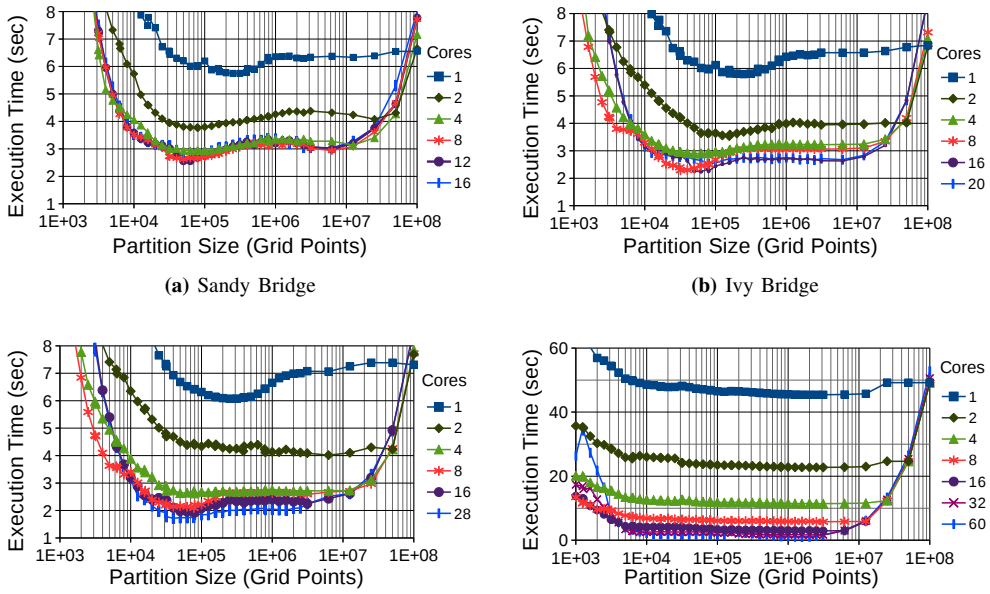
Note: We performed extensive experiments assessing overheads caused by invoking the timers used for the idle-rate, task duration and overhead counters. There were no significant overheads except for the cases where the experiments were run on only one core and the task durations were less than four microseconds.

Thread Pending Queue Metrics: The HPX counters, pending queue accesses and misses (*/threads/count/pending-accesses* and *-misses*), count the number of times the thread scheduler looks for work in the associated pending queue of each worker thread and the number of times it fails to find work there. The counters register the activity by the HPX-thread scheduler on the queues. Counters (idle-rate, task duration, and task overhead) required for the other metrics use timestamps and may not be available on all platforms, so we present the pending queue metrics as viable alternatives. Although individual counts are available for each pending queue, in this paper, the total count for all the pending queues are reported for each experiment. Counts are also available for the staged queues and were measured for this study but are not presented because the values were insignificant.

III. Experimental Platforms

HPX is a unified computational runtime system designed for parallel computation on either a single server or in distributed mode on homogeneous or heterogeneous clusters. This study is only concerned with performance useful for determining task granularity and optimizing task scheduling, therefore the experiments measure performance on a single node.

Experiments are performed on an Intel Xeon Phi coprocessor and three Intel nodes of the Hermione cluster, Center for Computation and Technology, Louisiana State University, running the Debian GNU/Linux Unstable, kernel version 3.8.13 (on the Xeon



(a) Sandy Bridge

(b) Ivy Bridge

(c) Haswell

(d) Xeon Phi (1 thread per core)

Fig. 3: Execution Time vs. Task Granularity (partition size) for various Intel microarchitectures

Phi, 2.6.38.8 k1om), using HPX V0.9.10. The specifications of the platforms are shown in Table III. We ran the experiments on the Xeon Phi for 1 to 4 threads per core. There was no improvement in results with multiple threads per core. So, we only present results from running with one thread per core.

TABLE I: Platform Specifications.

Node	**Haswell (HW)**	**Xeon Phi**
Processors	Intel Xeon E5-2695 v3	Intel Xeon Phi
Clock Frequency	2.3 GHz (3.3 turbo)	1.2 GHz
Microarchitecture	Haswell (HW)	Xeon Phi
Hardware Threading	2-way (deactivated)	4-way
Cores	28	61
Cache/Core	32 KB L1(D,I)	32 KB L1(D,I)
	256 KB L2	512 KB L2
Shared Cache	35 MB	
RAM	128 GB	8 GB
Node	**Ivy Bridge (IB)**	**Sandy Bridge (SB)**
Processors	Intel Xeon E5-2679 v3	Intel Xeon E5 2690
Clock Frequency	2.3 GHz (3.3 turbo)	2.9 GHz (3.8 turbo)
Microarchitecture	Ivy Bridge (IB)	Sandy Bridge (SB)
Hardware Threading	2-way (deactivated)	2-way (deactivated)
Cores	20	16
Cache/Core	32 KB L1(D,I)	32 KB L1(D,I)
	256 KB L2	256 KB L2
Shared Cache	35 MB	20 MB
RAM	128 GB	64 GB

IV. Experimental Results

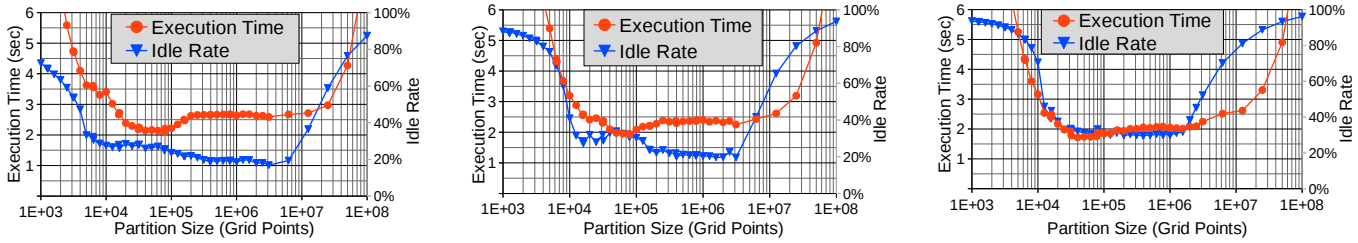
Using the benchmark HPX-stencil, we ran experiments over a large range of task grain sizes by varying the partition size from 160 points to 100 million points, adjusting the number of partitions to keep the number of grid points at 100 million. For each experiment, we compute the heat equation of 100 million grid points for 50 time steps (for experiments on the Xeon Phi we compute five time steps). We increase the number of cores used for each experiment but keep the total number of grid points the same for strong scaling results. We use the mean of ten samples for each of the experiments. We compute the mean, standard deviation, and coefficient of variation (COV) (the ratio of the standard deviation to the mean) of the execution times and event counts. COVs for execution times and event counts are less than 10%, (most are less than 3%) for experiments using less than 16 cores. For a few sample sets using more than 16 cores and when the partition size is less than 32,000, COVs range up to 21% on the Haswell node. For the Xeon Phi, COVs for partition sizes less than 5000 were greater than 10% for some of the sample sets. COV for event counts have the same behavior for the same sample sets. We will investigate this thouroughly in the future.

We examine the performance of the benchmark, HPX-stencil, by granularity and number of cores in Figure 3. On all platforms, execution time is large for very fine-grained tasks due to overheads caused by task management and for coarse-grained tasks where overheads are caused by poor load balance, not enough work to spread among the cores. In between these areas, we expect to see the execution time flatten out since task management overheads should be minimum. However, *wait time* as explained in II-A also influences the execution time and is dependent on task granularity, the number of cores used, and the underlying architecture.

Using the metrics in Sec. II-A we model the effects of varying task granularity and the number of parallel processors on overheads and performance. We present experimental measurements and resulting metrics for the Haswell and Xeon Phi platforms. Results of the metrics from the other two platforms are similar to the results from the Haswell node so are not presented here.

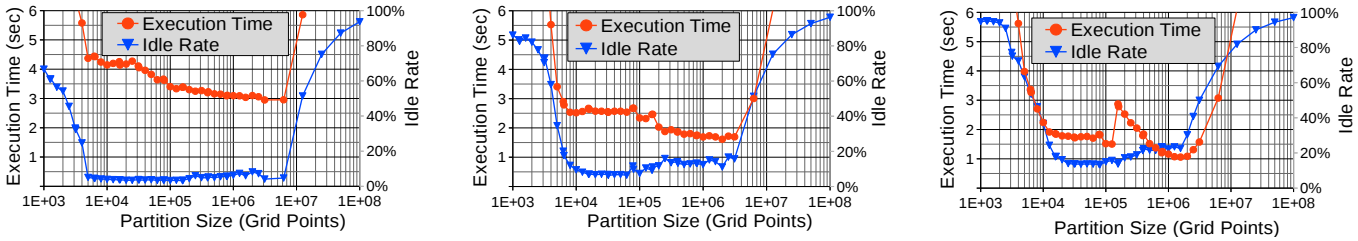
A. Idle-rate

Idle-rate is the ratio of time spent on HPX-thread management to that of execution, Figures 4 and 5. For very fine-grained tasks (small partition sizes) there are a large number of tasks to manage, and the task management is a large percentage, up to 90%, of the execution time. The regions on the left sides of the graphs show the increased execution times for very fine-grained tasks with partition sizes less than 12,500 grid points. The average task duration for computing 12,500 grid points using one core is 21 microseconds on Haswell and 1.1 milliseconds on the Xeon Phi. On the other extreme for very coarse-grained tasks idle-rate increases due to starvation. The tasks are so large that, at times, cores have no work to do while waiting for results, but the thread scheduler continues to look for work.



(a) Haswell 8 cores **(b)** Haswell 16 cores **(c)** Haswell 28 cores

Fig. 4: Idle-rate: Intel Haswell



(a) Xeon Phi 16 cores **(b)** Xeon Phi 32 cores **(c)** Xeon Phi 60 cores

Fig. 5: Idle-rate: Intel Xeon Phi

Idle-rate can be used to determine a range of adequate grain sizes, but since it does not mimic the effects of the parallelization time and overheads of the benchmark, it cannot be used to determine an optimal grain size. In figures 4a and 4b, for partition sizes from 20,000 to 100,000 even though idle-rate increases, the execution time decreases. This also appears in Figures 5b and 5c for partition sizes from 100,000 to 800,000 for 32 and 60 cores on the Xeon Phi coprocessor. In Sec. IV-C we show that this behavior is caused by *wait time*.

The experimental results indicate that, depending on requirements, an acceptable grain size can be determined by setting a threshold for the idle-rate. For example, on the Haswell node for 28 cores with a maximum threshold for idle-rate at 30%, the smallest partition size is 78,125 with an average task duration of 99 microseconds (average task duration measured while running with one core). At this partition size, the average execution time is 1.75 seconds, which is within the standard deviation (0.03) for the minimum time of 1.71 seconds using a partition size of 40,000.

B. HPX-thread Management Overhead

We compute the overhead, Eq. 4, caused by management of HPX-threads to determine the effect on the execution time. Figures 7 and 8 show that for both platforms the overhead is high for very fine- and coarse-grained tasks, and the behavior of the execution time in those regions is the same as the overhead. However, in the center region the behavior of the HPX-thread management overhead is relatively flat and execution time does not follow that behavior. For fine- to medium-grained tasks, where task management is not the predominant overhead, waiting on resources has a similar behavior to performance as shown in the following section.

C. Wait Time

To assess the effect of waiting on resources, we first compute the average *wait time* per HPX-thread, Eq. 5. Results from our experiments show that the *wait time* per HPX-thread increases with the number of cores and with the partition size as shown for the Haswell node in Figure 6.

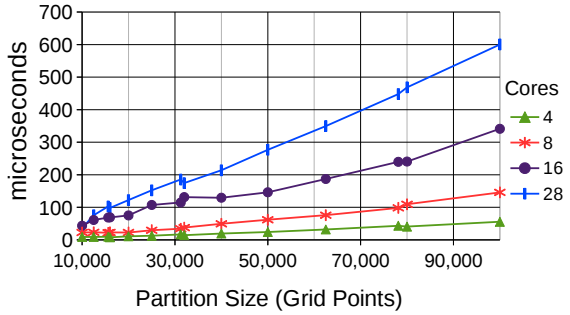
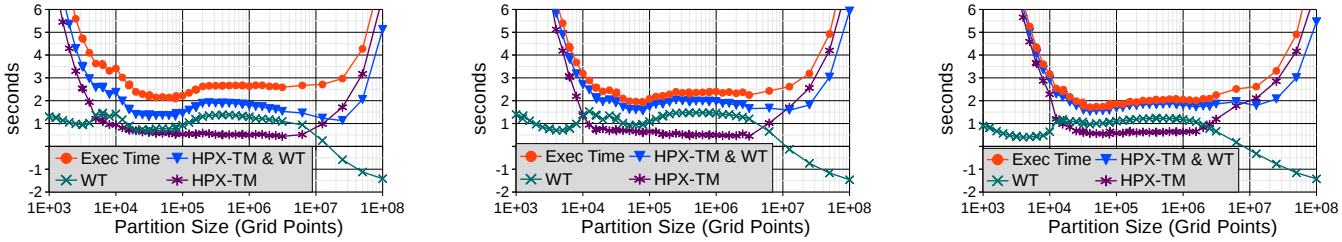


Fig. 6: Wait Time per HPX-Thread (Haswell)

To compute the *wait time* for the entire experiment we use Eq. 6. The results of the *wait time* metric for the Haswell node in Figure 7 and for the Xeon Phi in Figure 8 show that the behavior for fine- to medium-grained task size mimics the execution time. This region is for partition sizes ranging from 20,000 to 1,000,000 grid points with task durations of 32 microseconds to 1.3 milliseconds for the Haswell node and 1.8 to 50 milliseconds on the Xeon Phi. We note that *wait time* is negative (i.e. the average task duration run on multiple cores is larger than when run on one core) for the experiments with very coarse-grained tasks. This occurs when the number of tasks per time step are smaller than the number of cores used. These task sizes are well beyond the range of fine-grained to medium-grained tasks of interest for task parallelism but are presented here for completeness.

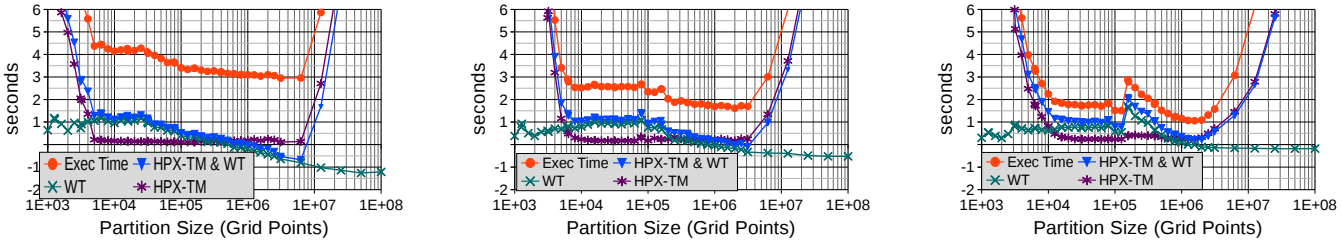
D. Combined Costs: HPX-thread Management and Wait Time

The combination of time for managing HPX-threads and waiting on resources show that these are the driving effects on execution time (Figures 7 and 8). Not only does the behavior of the combined costs mimic that of execution time, but we can also see that the costs increase with parallelism causing the execution time to stay relatively the same after eight cores. The gap between execution time and the cost of thread management and *wait time* depicts the actual computation time. As the number of cores used increases (i.e.



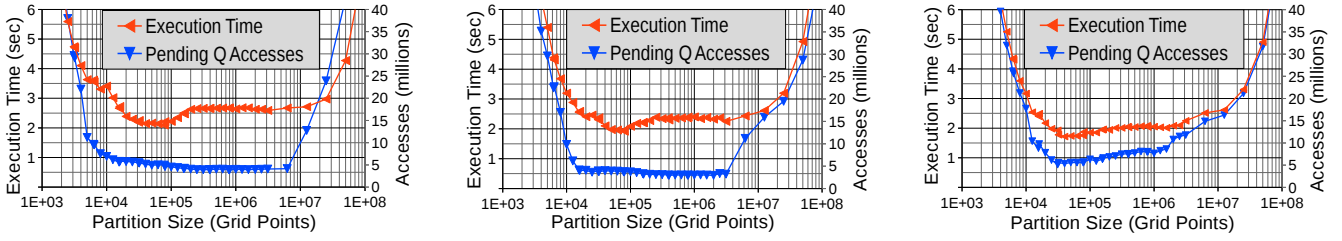
(a) Haswell 8 cores **(b)** Haswell 16 cores **(c)** Haswell 28 cores

Fig. 7: HPX-Thread Management (TM) and Wait Time (WT): Intel Haswell



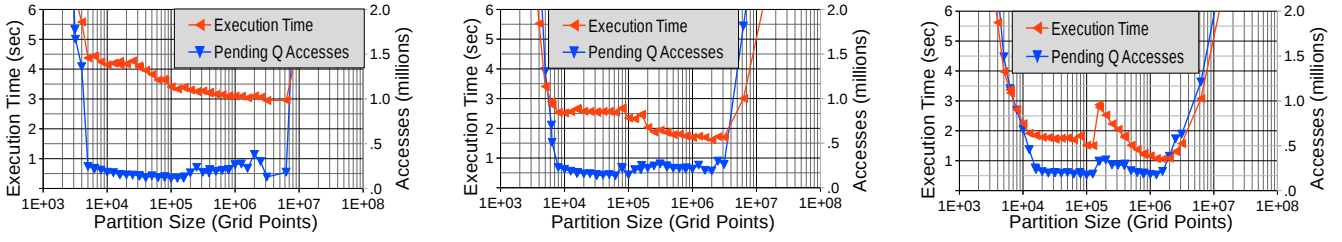
(a) Xeon Phi 16 cores **(b)** Xeon Phi 32 cores **(c)** Xeon Phi 60 cores

Fig. 8: HPX-Thread Management (TM) and Wait Time (WT): Intel Xeon Phi



(a) Haswell 8 cores **(b)** Haswell 16 cores **(c)** Haswell 28 cores

Fig. 9: Pending Queue Accesses: Intel Haswell



(a) Xeon Phi 16 cores **(b)** Xeon Phi 32 cores **(c)** Xeon Phi 60 cores

Fig. 10: Pending Queue Accesses: Intel Xeon Phi

increased parallelism) the computation time decreases, but overheads and *wait time* increases impairing scaling.

E. Thread Pending Queue Accesses

Measuring the number of accesses to the pending queues gives an indication of the amount of activity involving the thread scheduler. Figures 9 and 10 show that this metric can be used to determine adequate task grain size. For example, the minimum pending queue accesses for Haswell when running on 28 cores occurs when the partition size is 31,250 and the execution time is 1.925 seconds, within 13% of the minimum time. This metric gives similar results to the idle-rate metric but does not require timestamps.

V. Related Work

Work related to adaptive thread scheduling includes the work based on introspection of hardware behavior by A. Porterfield et al. in [19]. They use RCRdaemon to collect hardware memory and power measurements and based on those measurements throttle the number of running OS threads. Their results indicate that this type of adaptive scheduling can improve performance and save energy. This requires the ability to perform throttling actions at the hardware level to be effective in restarting threads. An implementation of the thread scheduling mechanism has been interfaced with HPX and we plan to apply our findings to use our metrics with their scheduler.

S. Olivier et al. in [8] characterizes one of the components, work time inflation, that causes poor performance in task parallel applications. They use a locality framework to mitigate work time inflation for task parallel OpenMP programs. They extended that work by augmenting the Qthreads [5] library with locality-aware scheduling. Our wait time metric is a measure of work time inflation, and we characterize wait time as related to varying task granularity.

In [20] Y. Sun et al. employ grain size adaptation using an execution tree cut-off strategy to expand the task grain size in the implementation of the framework, Parallel State Space Search Engine (ParSSSE) over the Charm++ runtime system [15]. They extended ParSSSE to incorporate adaptive task granularity by sampling the time it takes to expand individual nodes of the graph and estimating the average time for the entire application. Their work shows the benefit of adaptive granularity for dynamic graph problems in a runtime system that utilizes task parallelism and message driven execution for state space search problems.

VI. Conclusions and Future Work

Our goal is to dynamically adapt task grain size to optimize parallel performance using HPX, an open source, general purpose, C++ runtime system. In this paper, we present a methodology and the resulting characterization study that characterizes and evaluates *scheduling overheads* and *time waiting on resources* affected by task granularity for parallel programs. We show that by collecting pertinent event counts, we can determine an optimal grain size to minimize scheduling overheads and wait time for best performance.

For future work, we will apply the methodology to dynamically adapt grain size to minimize scheduling overheads and improve performance of parallel applications. We plan to conduct a performance study with a variety of thread scheduling policies to determine methods to dynamically adapt the thread scheduler. A. Porterfield's throttling scheduler [19] and an initial implementation of the policy engine from the APEX prototype in [21] have been integrated with HPX. We plan to apply our findings to drive the policy engine with our metrics for adapting thread granularity and scheduling policies.

: **Acknowledgements.**
This work is supported by NSF grant number CCF-111798. We thank Bryce Adelstein-Lelbach for his work on the HPX thread scheduler and the addition of counters that made this study possible. We also thank the anonymous reviewers for their insightful recommendations.

References

[1] H. Kaiser, T. Heller, A. Berge, and B. Adelstein-Lelbach, "HPX V0.9.10: A general purpose C++ runtime system for parallel and distributed applications of any scale," 2015. [Online]. Available: http://dx.doi.org/10.5281/zenodo.16302

[2] P. Kogge *et al.*, "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," University of Notre Dame, Notre Dame, IN, Tech. Rep. TR-2008-13, 2008.

[3] C. E. Leiserson, "The Cilk++ concurrency platform," in *DAC '09: Proceedings of the 46th Annual Design Automation Conference*. New York, NY, USA: ACM, 2009, pp. 522–527. [Online]. Available: http://dx.doi.org/10.1145/1629911.1630048

[4] G. Contreras and M. Martonosi, "Characterizing and improving the performance of intel threading building blocks," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, Sept 2008, pp. 57–66.

[5] "The Qthread Library," 2014, http://www.cs.sandia.gov/qthreads/.

[6] K. Wheeler, R. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–8.

[7] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: ACM, 2014, pp. 6:1–6:11. [Online]. Available: http://doi.acm.org/10.1145/2676870.2676883

[8] S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins, "Characterizing and mitigating work time inflation in task parallel programs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 65:1–65:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389085

[9] The C++ Standards Committee, "ISO International Standard ISO/IEC 14882:2011, Programming Language C++," Geneva, Switzerland: International Organization for Standardization (ISO)., Tech. Rep., 2011, http://www.open-std.org/jtc1/sc22/wg21.

[10] ——, "ISO International Standard ISO/IEC 14882:2014, Programming Language C++," Geneva, Switzerland: International Organization for Standardization (ISO)., Tech. Rep., 2014, http://www.open-std.org/jtc1/sc22/wg21.

[11] Niklas Gustafsson and Artur Laksberg and Herb Sutter and Sana Mithani, "N3857: Improvements to std::future¡T¿ and Related APIs," The C++ Standards Committee, Tech. Rep., 2014, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3857.pdf.

[12] Vicente J. Botet Escriba, "N3865: More Improvements to std::future¡T¿," The C++ Standards Committee, Tech. Rep., 2014, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3865.pdf.

[13] Chris Mysen and Niklas Gustafsson and Matt Austern and Jeffrey Yasskin, "N3785: Executors and schedulers, revision 3," , Tech. Rep., 2013, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3785.pdf.

[14] Intel, "Intel Thread Building Blocks 3.0," 2010, http://www.threadingbuildingblocks.org.

[15] PPL, "PPL - Parallel Programming Laboratory," 2011, http://charm.cs.uiuc.edu/.

[16] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra, "Using PAPI for hardware performance monitoring on linux systems," in *International Conference on Linux Clusters: The HPC Revolution*, jun 2001.

[17] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing*. Springer Verlag, 2009, pp. 157–173, 3rd Parallel Tools Workshop.

[18] The STEllAR Group, Louisiana State University, "HPX Users Manual," 2007-2014, available under the Boost Software License (a BSD-style open source license). [Online]. Available: http://stellar-group.github.io/hpx/docs/html/

[19] A. Porterfield, R. Fowler, A. Mandal, D. O'Brien, S. Olivier, and M. Spiegel, "Adaptive Scheduling Using Performance Introspection, RENCI Technical Report TR-12-02, Renaissance Computing Institute," 2012.

[20] Y. Sun, G. Zheng, P. Jetley, and L. V. Kale, "An Adaptive Framework for Large-scale State Space Search," in *Proceedings of Workshop on Large-Scale Parallel Processing (LSPP) in IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2011*, Anchorage, Alaska, May 2011.

[21] K. Huck, S. Shende, A. Malony, H. Kaiser, A. jh, R. Fowler, and R. Brightwell, "An early prototype of an autonomic performance environment for exascale," in *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '13. New York, NY, USA: ACM, 2013, pp. 8:1–8:8. [Online]. Available: http://doi.acm.org/10.1145/2491661.2481434