Performance Analysis with HPX

Rekha Raj

Project submitted to the
Graduate Faculty of the
Louisiana State University
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Systems Science

Dr. Hartmut Kaiser, Chair
Dr. Jianhua Chen
Dr. Jian Zhang

November 12, 2014
Baton Rouge, Louisiana

Performance Analysis with HPX

Rekha Raj

Abstract

With High Performance Computing moving towards Exascale, where parallel applications will be required to run concurrently on millions of cores, every part of the computational model must perform ideally to achieve optimal performance. The task scheduler is one of such entities that could be enhanced to runtime application prerequisites. Not only the overheads associated with task scheduling vary depending on the task size but also are affected by the different schedulers used. One of the best strategies used to ameliorate performance and to decrease the overheads would be to adapt the scheduler to the application's needs or to vary the task size or a combination of both. In this work, we show how scheduling overheads vary with the different types of schedulers used in the High Performance ParalleX (HPX) runtime system. In addition to this, we show how the different schedulers perform with varying task size. This performance analysis focuses on understanding how the different schedulers perform for a specific HPX benchmark in turn helping us to settle on a scheduler that could give optimal performance. HPX is an ideal experimental platform as it makes use of asynchronous fine-grained task scheduling and incorporates a dynamic performance modeling capability. In addition to this, HPX offers performance counter capabilities with which we can characterize scheduling overheads.

Acknowledgments

I would like to express my heartfelt gratitude towards Dr. Hartmut Kaiser who has the substance of a genius and without whose guidance this project work would not have been possible. Right from when I joined the Ste||ar group, Dr. Hartmut Kaiser has offered priceless support, backing and guidance.

My sincere thanks to the members of my committee, Dr. Jianhua Chen and Dr. Jian Zhang, for their continuous support and encouragement. Also, I appreciate the assistance provided by the staff members of Division of Computer Science and Engineering.

In addition, a special thank you to all my colleagues from Ste||ar lab for the support provided over the course of time. Patricia Grubel for the insightful discussions and always willing to help attitude, Vinay Amatya who introduced me to SLURM jobs, Thomas Heller for looking into issues when things went wrong; and everyone else for their timely support.

I also would like to thank the staff at Center for Computation and Technology and IT Support for their continuous assistance.

Last but not the least, I whole-heartedly thank my family for their unconditional support and guidance throughout my journey. Finally, I would like to express my sincere thanks to my friends for making my stay at LSU enjoyable and memorable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High Performance Computing (HPC) is utilized for undertakings that are exceedingly computationally comprehensive, information escalated or a mix of both. These assignments specifically cannot be comprehended by a smart phone or a common workstation. HPC utilizes supercomputers and computer clusters to take care of the progressed computational issues. It permits researchers and designers to comprehend complex scientific, engineering, and business issues. However, current advances in high-performance computing suffers from the issues tormenting parallel reckoning. These issues incorporate productive usage of system resources, scalability and inability to handle rapidly evolving workload. The most scathing bottlenecks to the compelling utilization of the new generation HPC systems include overheads, latency, starvation and contention. To moderate the previously stated issues, it is necessary to re-evaluate the methodology of parallelization models.

Execution models that support scalability in massively parallel systems are beginning to appear in the HPC community. ParalleX is one such execution model which overcomes confinements of current models of parallelism by assuaging contention, overhead, latency and starvation. It improves the performance of the HPC systems by facilitating a new computing

dynamic through the application of message-driven computation in a global address space context with lightweight synchronization [4]. ParalleX implements components, for example, multi-threading, parcels, percolation and local control objects (LCO). The Parallex execution model replaces communicating sequential processes (CSP) to give another computing standard exemplifying the principles for arranging and carrying out scalable computations of high efficiency incredibly surpassing the capacities of today's issues. The Parallex execution models and their particular run-times are backing parallelism through enormous multi-threading where an application is split into various tasks or threads of varying size that execute simultaneously. Scalability, the capability to keep up system effectiveness with changing workloads, is affected by the runtime adaptive resource management and also decision making. These run-times have components that can be dynamically adapted to optimize performance. One such component is the thread scheduler. These run-times make use of different thread schedulers which with different task size and granularity produce varying overheads [12]. It gives us an opportunity to optimize the performance by adjusting the thread scheduling algorithms.

This work focuses on measuring the overheads associated with the different thread schedulers for a specific HPX application.

Chapter 2

Related Work

Adaptive thread scheduling has been a topic of research for a long time. Many methodologies have been implemented to achieve greater performance by making use of adaptive thread scheduling. Chan introduced a new design of adaptive thread scheduler [1] which makes use of two concurrency control techniques Throttle and Probe that bring performance gain by eliminating excessive threading. Yi Guo's [6] work makes use of a Scalable Locality aware Adaptive Work-Stealing scheduler. SLAW makes use of locality hints provided by the programmer to perform locality-aware scheduling. Ibrahim Hur used history-based memory scheduler that takes into consideration the command history to schedule commands that could match some expected command patterns [8]. As an efficient scheduler that could provide dynamic load balancing among CPU cores is imperative, a work-stealing scheduler was implemented by J. Nakashima [11]. An API was proposed that could alter the scheduling strategies by taking into consideration hardware and application specific knowledge. Cooperative polling [10] introduced by Charles Krasic made use of an application-level event scheduler and a kernel thread which work together to transmit time-constrained events with minimum kernel preemption.

Chapter 3

HPX Runtime System

High Performance ParalleX (HPX) is one of the first attempts towards implementing a runtime framework usage of the ParalleX execution model. HPX is a general purpose C++ runtime system for parallel and distributed applications. By a runtime system, we mean that any application that uses HPX will be directly linked to its libraries. The main advantage of using a runtime system is that the runtime system will be started only when the application is launched and will automatically shut down when the application stops [9]. HPX, with its increased scalability, high efficiency and ease of programming, has been successful in replacing Communicating Sequential Processes execution model paving way to a new prototype exemplifying the governing principles for organizing and conducting exceedingly scalable computations significantly surpassing the ability of today's issues. A brief description on HPX implementation, its performance monitoring system and also the different schedulers supported by HPX is mentioned below.

The design of the API exposed by HPX conforms to the existing C++11 Standard [3], the (draft) C++14 Standard, and related proposals to the C++ standardization committee [5]. All the interfaces defined by the C++ Standard related to multi-threading like future,

thread, mutex or async have been implemented by HPX on top of its own user-level threading system in a fully conforming way. After a wide community-based discussion, these interfaces were accepted for ISO standardization and since then have proven to be effective tools for managing asynchrony. HPX looks to expand these concepts, interfaces, and ideas embodied in the C++11 threading system to distributed and data-flow programming use cases. Each conceivable exertion was made to keep the greater part of the usage of HPX completely fitting in with C++ to ensure a high degree of code portability, and a high degree of performance portability of HPX applications as well. A more detailed description of HPX and its API can be obtained from [5].

3.1 HPX Subsystems

Active Global Address Space (AGAS). HPX makes use of AGAS instead of Partitioned Global Address Space (PGAS). The essential part of AGAS is the implementation of a global address space that compasses all localities an application is currently running on which allows objects to be moved to other localities and this does not change the address of the component.

Parcel Subsystem. It acts as a communication channel in the HPX runtime system. Communication is done in the form of active messages called parcels. The parcels encase remote method calls. Data communication is done by attaching parameters to the parcel. These parcels are sent and received by localities. The parcels received from the localities are converted into HPX threads. The threading subsystem then schedules, executes and recycles these threads.

Threading Subsystem. The HPX threading subsystem is in charge of creating, scheduling, executing and destructing HPX threads. HPX makes use of a hybrid threading model, otherwise known as the M:N model. In this model, N HPX threads are mapped onto M

kernel/OS threads [7]. This model supports fine-grained parallelization.

Local Control Objects (LCOs). Any object that is capable of creating a new HPX thread or which could re-activate a suspended thread comes under this category. LCOs implement constraint-based synchronization which controls parallelization and synchronization in HPX applications.

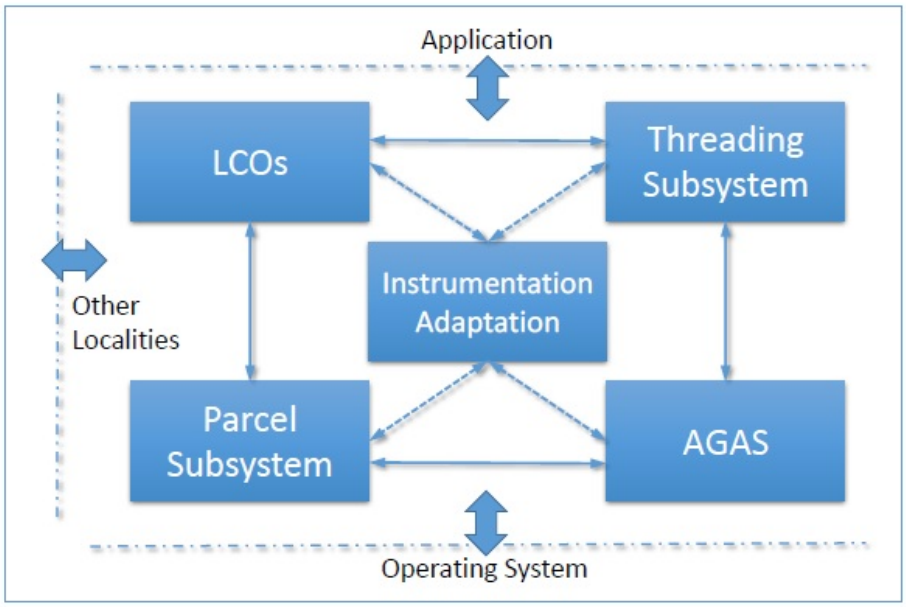

Figure 3.1: The modular structure of HPX implementation [9]

3.2 Task Granularity

A problem is often described as a collection of tasks. The granularity of the task is the amount of time the task executed continuously without synchronization or communication. Task granularity distributed among processors is one of the important aspects of thread scheduling overheads. Fine-grained tasks perform small amounts of computations where as coarse-grained tasks perform long periods of continuous computations. A greater number of smaller tasks gives more opportunities for load balancing, but they incur more overheads

due to the creation and scheduling of large quantities of threads and also due to the frequent transfer of data among processors. There is an increase in overheads due to synchronization, communication and also the contention on resources. A smaller number of larger tasks, on the other hand, reduce the management overhead, but they provide fewer opportunities for load balancing. Since both the fine-grained and coarse-grained tasks have disadvantages of their own, a better solution would be to use an efficient grain size for the application. For many of the applications, this solution strategy can be employed. However, there are applications such as the graph applications which naturally employ the fine-grained tasks. These applications are said to be scaling-impaired. The scaling-impaired applications can benefit greatly from adaptive thread scheduling mechanisms. By adaptive thread scheduling mechanism, we mean mechanisms that could essentially detect the granularity size and could modify the scheduling strategy or the task granularity to perform more efficiently.

The initial steps to the runtime adaptivity incorporate understanding how the different schedulers affect the performance for different task sizes and also understanding the relationship between the overheads and task granularity. Performance analysis of different schedulers for different task sizes is undertaken in this work.

3.3 HPX Thread Scheduling Policies

The HPX threading subsystem is responsible for scheduling and executing the HPX threads. HPX threads could be scheduled easily, and a kernel call is not required when there are context switches between the HPX threads. This reduces the overhead associated with the thread execution and suspension. A work-queue based execution strategy is used and the work stealing strategy of the HPX thread manager is similar to systems like Parallel Patterns Library (PPL) [2].

There are different stages through which HPX threads have to traverse. These stages in the life cycle of HPX threads are staged, pending, active, suspended and terminated. Whenever an HPX thread is created by a thread manager, it is placed in a staged queue. Staged threads do not have any context associated with them. This allows them to be moved to queues within other memory domains without incurring memory costs. The HPX thread scheduler then gives the staged threads a context and then they are placed in the pending queue. A thread is ready to run when it is in the pending queue. An HPX thread goes to the active state when it starts running. The active threads can suspend themselves while synchronization or communication is required. The threads enter the suspend state when synchronization or communication is being done. Once resources or data become available, these threads will be moved to the pending queue again. Once the execution is complete, the thread enters the terminated state.

The HPX runtime system currently supports six different thread scheduling policies. These scheduling policies are local, local-priority, abp-priority, hierarchy, static-priority and periodic-priority. In order to use any of these thread scheduling policies, HPX should be built with the appropriate scheduler flag turned on. For example, in order to turn on the static-priority scheduling policy, we would use the flag, cmake $-DHPX\_THREAD\_SCHEDULERS = static - priority$. In addition to this, we can invoke these policies from the command line using the option $--hpx : queuing$. To invoke the static priority scheduling policy from the command line, we use the option $--hpx : queuing = static$.

Local-Priority Scheduling Policy. This scheduling policy is the default policy used when running HPX applications. In this policy, each OS thread has its own queue from which it can pull its work. The number of high priority queues in this case will always be equal to the number of OS threads. High priority threads, as the name suggests are executed by the OS threads before executing any other work. In addition to the high priority queues, a low priority queue is also maintained. Threads from this queue will be scheduled only when

there is no other work left to be done.

NUMA sensitivity can be turned on for this scheduling policy using the command line option $--hpx:numa-sensitive$. NUMA sensitivity encourages work stealing from same NUMA domains first and then from other NUMA domains. The thread manager looks at the worker threads own pending queue first and then its staged queue. At the point when there is no work left for the worker thread in the local queue system, the thread manager searches the local NUMA domain to check the staged queues first and then the pending queues to find work. If it fails to find work in the local domain, it search for work in other NUMA domains. Staged queues will be searched first followed by the pending queues.

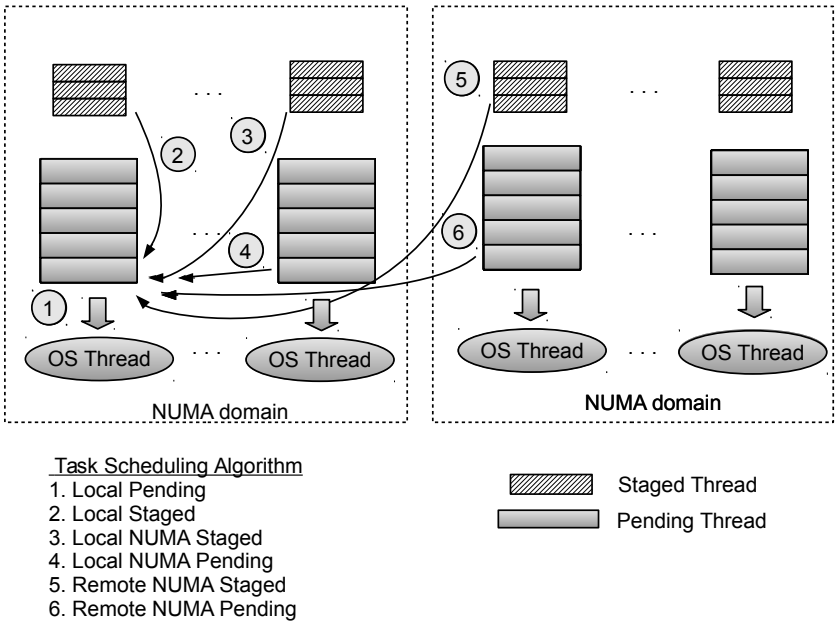

Figure 3.2: Local-Priority Scheduler

Local Scheduling Policy. The local scheduling policy is similar to the local-priority scheduling policy except for the fact that it does not have any priority queues. For the local scheduling policy, each OS thread has its own queue from which it pulls its work.

ABP-Priority Scheduling Policy. ABP scheduling policy, unlike the local and local-priority scheduling policies, makes use of a lock-free double ended queue for each thread. Work is added from the top of the queue and work stealing is done by taking threads from the bottom of the queue. By default, the number of high priority queues in the ABP scheduling policy is equal to the number of OS threads. Work is taken from the high priority queues first. When no work left, then the low priority queue comes into the picture. ABP scheduling mechanism also supports NUMA sensitivity like local-priority scheduler.

Hierarchy Scheduling Policy. In this policy, a tree of work items is maintained. New work is obtained by the OS thread by traversing through the tree. Work stealing is done from the parent queue in the tree.

Static-Priority Scheduling Policy. One queue per OS thread is maintained and it is through this queue the OS pulls its work. In this policy, threads are distributed in a round robin fashion handling all processes without priority. Each HPX-thread will be executed only for a particular time slot. Once the time slot is over, the next process will get executed. The HPX-thread is resumed only when it is allocated a time slot later on. The main difference between static-priority scheduling and other scheduling policies is that no work stealing happens in the case of a static-priority scheduler.

Chapter 4

Benchmark

The performance analysis of the various scheduling policies was done on the HPX Stencil application. This application is a one-dimensional heat distribution benchmark which is available in the HPX distribution package. This code mimics the dispersion of heat across a ring by breaking the ring into discrete points. The temperature of the next time step is computed using the temperature of the point and the neighboring points.

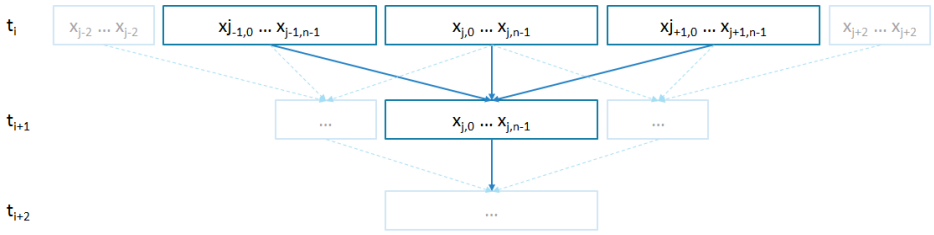

Figure 4.1: Heat Distribution Benchmark, HPX Stencil

This benchmark was developed primarily to exhibit the steps involved in the futurization of parallel applications using the HPX runtime system. In short, in this application we use futures. A future can be described as a value that exists now or will be produced in the future. Typically a function is used to produce the value of a future. This producer function will be executed asynchronously in a new HPX thread. Additionally, data points are split

into partitions, and each partition is represented with a future. The number of calculations in each future can be changed by varying the number of data points in each partition, which in turn helps us to control the grain size of the problem.

The dependency graph shows the sequence in which the tasks get executed. The asynchronous threading API of HPX is used to execute all the tasks in the desired sequence. We use hpx:async to launch each of these tasks as a lightweight HPX thread. This in turn generates an hpx:future that represents the expected result for each of the tasks. These future instances are combined into the dependency tree using the additional HPX facilities to compose futures sequentially and in parallel. The future objects represent the terminal nodes, and their combination represents the edges and the intermediate nodes of the dependency graph.

4.1 Platform

HPX is a unified computational runtime system. It can run parallel computation on either a single server or in distributed mode on homogeneous or heterogeneous clusters. This performance measure was done only on a single node. Experiments were performed on one node of the Hermione cluster, a heterogeneous system with a variety of nodes, Center for Computation and Technology, Louisiana State University, running the Debian GNU/Linux Unstable, kernel version 3.8.13, using version HPX V0.9.9. The specifications of the platform are shown below,

Table 4.1: Platform Specifications

Node Name	***Ariel***
Processors	Intel Xeon E5-2690 (2 processors)
Frequency	2.9GHz (3.8 turbo)
Microarchitecture	Sandy Bridge
Cores	16
Cache/Core	32KB L1 D and I, 256KB L2
Shared Cache	29MB L3
Memory	32GB DDR3

Chapter 5

Test System

Our main aim is to explore the performance of different HPX schedulers on HPX-Stencil application. HPX-Stencil offers us the ability to control the partition size. When we vary grain size from fine-grain to coarse-grain, different overheads can be observed. Depending on the different scheduling mechanisms, the overheads may vary again. Executing applications with a larger number of fine-grained tasks can help in achieving load balancing, parallelism and speed-up but can cause overheads. These overheads include the overheads for thread creation and deletion and due to contention for queuing resources and allocation of memory for thread stacks. On the other hand, executing smaller numbers of coarse-grained tasks can cause overheads caused by poor load balancing and starvation of worker threads. We determine metrics that measure performance behavior, and associated overheads and then use the facilities in HPX to measure required event counts. For this study, we executed the benchmark HPX-Stencil over a large range of partition sizes, to vary the granularity, and for an increasing number of cores for scaling performance. The computations are done using the heat equation for 50 time steps and 100 million grid points for each data set. When collecting performance and counter data, we make multiple runs and calculate means and variances of these counts. We calculate the metrics using the average of the required event

counts.

5.1 Performance Metrics

The metrics and their corresponding event counts that were used for our study are as mentioned below

Rate of Execution. The execution rate is a measure of the number of updates per second as the grain size increases. It is the ratio of the total number of grid points calculated over the execution time, also referred to as lattice updates per second (LUPS).

Thread Idle-rate. Thread idle-rate is the ratio of time spent on thread management tasks to the execution time. HPX can measure the time spent on just the computation portion of each task and total time to complete each task. These times are then accumulated for the application. The below mentioned equation is used by HPX to compute idle-rate.

$$i_r = 1 - (t_{comp}/t_{tot})$$

Here t_{comp} is the accumulated computation time of the tasks and t_{tot} is the accumulated total time. To adjust scheduling mechanisms and grain size, idle rate is a key. This is obtained from the */threads/idle-rate* performance counter.

Phase Duration. It represents the average time it takes to execute each phase of the task. A thread phase begins whenever a thread is activated, or it has been reactivated from its suspended state. Whenever the thread manager activates an HPX-thread, it changes the thread's state from pending to active, and each occurrence is recorded by the HPX performance counter, */threads/count/cumulative-phases*. Phase duration is estimated as

below,

$$p_d = (t_e * c)/p$$

Here, t_e is the overall execution time, c is the number of cores and p is the number of executed thread phases, from the counter, */threads/count/cumulative-phases*. Since here we use the overall execution time, phase duration includes overheads too.

Task Length. Task Length is the average computation time of the task in each uninterrupted phase. We estimate it using idle-rate, i_r, overall execution time, t_e, and the number of cores c. The difference between the average task length and phase duration is that phase duration which includes the overheads caused by task management.

$$t_l = (1 - i_r) * (t_e * c)/p$$

Pending Accesses and Pending Misses. HPX maintains counters that can compute the number of times the thread scheduler looks for work in the associated pending queues of each worker thread. This count is computed by the */threads/count/pending-accesses* counter. In addition to this, HPX includes a counter that can measure the number of times the thread scheduler fails to find work in the pending queues. The pending-misses count is assessed by the */threads/count/pending-misses counter*.

Last Level Cache Misses. HPX provides a provision of accessing hardware counters as HPX performance counters using PAPI interface. Last Level Cache Misses is one such performance counter maintained by HPX, */papi/PAPI_L3_TCM*. While performing the performance study it was observed that the last level cache misses affected the performance greatly with increase in grain size.

Chapter 6

Results

For our performance study, the HPX-Stencil application was run on 1, 4, 8 and 16 cores of Ariel node on Hermione cluster. The partition size was varied from 160 points to 100 million points and for each experiment, the heat equation was computed for 50 time steps. The application was run using Local-Priority, Static-Priority, ABP-Priority and Hierarchy scheduling policies. In Figure 6.1, we show how the scheduling overheads vary with the different types of schedulers and also how these overheads vary with partition size. For each sample, we calculate the scheduling overheads by multiplying the idle-rate with the execution time of the sample where idle-rate is the ratio of time spent on thread management to execution time.

When the grain size is small, the overheads are high. The percentage of overheads in fine grain is a large percentage of the execution time. There are fixed overheads due to thread creation and deletion and in addition to this fixed overhead, there are overheads caused by the larger number of threads. The increasing number of threads in fine grain can cause contention on memory caches and queues which leads to additional overheads. Upon executing the benchmark on 16 cores for the different schedulers, we could observe that for a partition

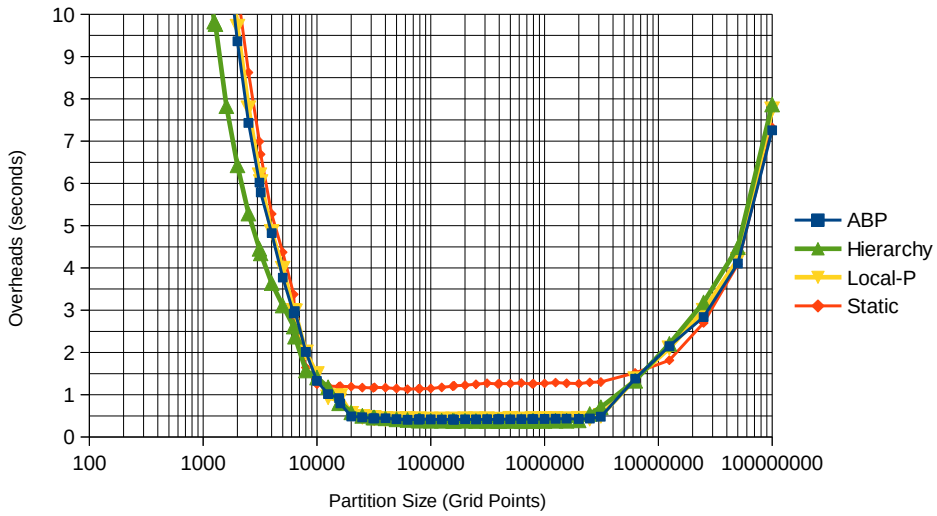

Figure 6.1: HPX-Stencil : Task Management Overheads (16 Cores)

size less than 8000, the overheads were more than 50 percent of the execution time. With an increase in grain size, there is a fluctuation in the idle-rate counts. It can be observed that the overheads associated with the static scheduler is high when compared to the other schedulers when the partition size is between 10000 and 3125000. The overheads associated with static scheduler are high because when the other schedulers have the ability to steal work, static scheduler can not perform any work stealing and due to this reason some threads are sitting idle waiting for work.

For partition sizes ranging from 20,000 to 3,125,000, the overheads are below 0.5 seconds. When the grain size reaches 6,250,000, the overheads start increasing again. At this grain size, the number of partitions is equal to the number of cores, i.e., 16. With the increase in grain size, there is an increase in the overheads due to poor load balancing.

The execution time for the different schedulers for different partition sizes on 16 cores is given in Figure 6.2. The execution time of the schedulers varies with increasing grain size. For the ABP and the Local-Priority schedulers, the best execution time is 2.26 seconds that is obtained when the grain size is 32,000. The best execution time of the Static and Hierarchy

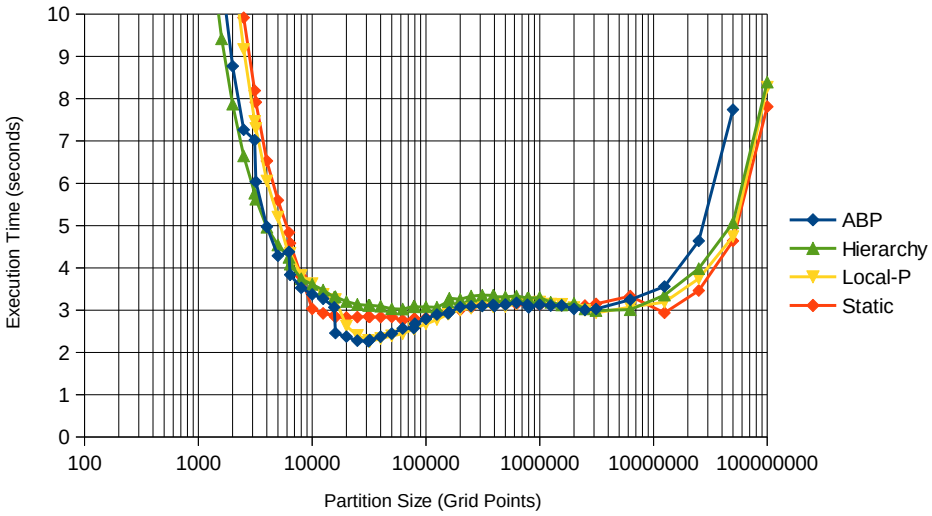

Figure 6.2: HPX-Stencil : Execution Time (16 Cores)

schedulers is obtained when the partition size is 62,500. The best execution time of Static scheduler is 2.7, seconds and that for the Hierarchy scheduler is 3.02 seconds.

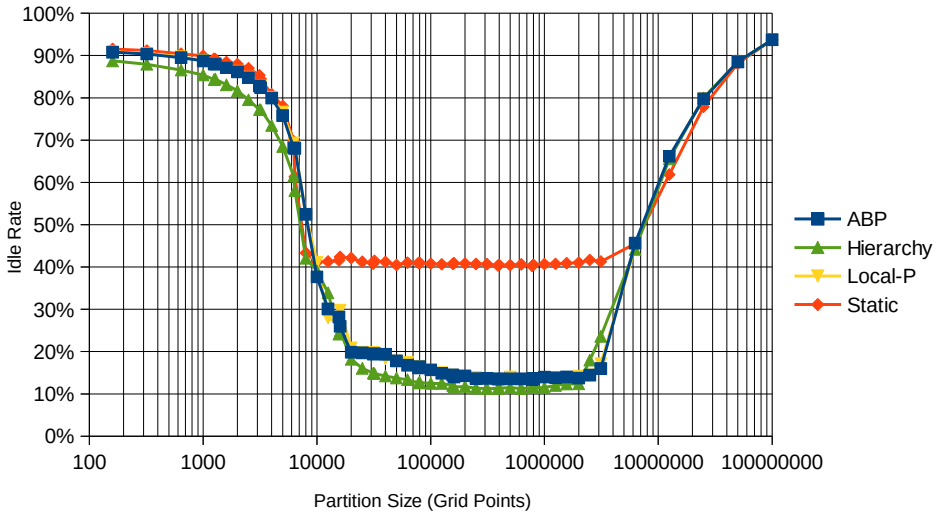

Figure 6.3: Idle-Rate (16 Cores)

Idle-rate for the different schedulers is given in Figure 6.3. We can measure the idle-rate at runtime as HPX has a performance counter to account for it. It is a ratio of thread management to the overall time. Figure 6.2 shows the results obtained by running the idle-

rate performance counter for HPX-Stencil. For very fine grain size, the overheads are high compared to the amount of actual work being performed. The increase in overheads occurs when the partition size is very small. This is because the HPX schedulers have to create and manage a large number of threads. For the coarse grain, on the other hand, the scheduler continuously looks for work as all the worker threads are not kept busy, hence an increase in overheads. For 16 cores, the idle rate for the ABP, Local-Priority and Hierarchy scheduler falls below 21 percent for a partition size of 20,000 points. The idle rate for these schedulers further decreases with an increase in grain size. The idle-rates for the static scheduler, on the other hand, are very high. At a partition size of 20,000 while the other schedulers have an idle rate less than 21 percent, the static scheduler has and idle rate of 42 percent. The idle rates are high for the static scheduler because it makes use of the round-robin thread distribution and hence the threads are waiting for work to be assigned to them.

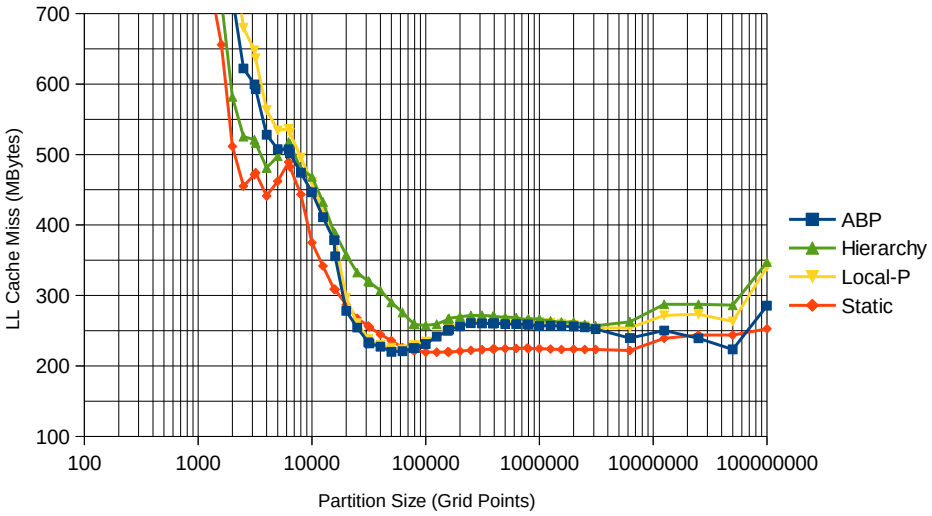

Figure 6.4: HPX-Stencil - Last Level Cache Misses

From Figures 6.2 and 6.3, we can see that though the hierarchy scheduler has the lowest idle rate, its performance is not as good as the other schedulers. The Last Level Cache(LLC) Misses shown in Figure 6.4 help us to explain the additional overheads that are incurred. LLC misses plays an important role, since it measures the number of expensive memory

accesses. However, we can observe that the LLC misses for hierarchy scheduler is high when compared to the other schedulers. This reduces the execution time as the data needs to be transferred from the memory which is slower when compared to transferring data from the cache.

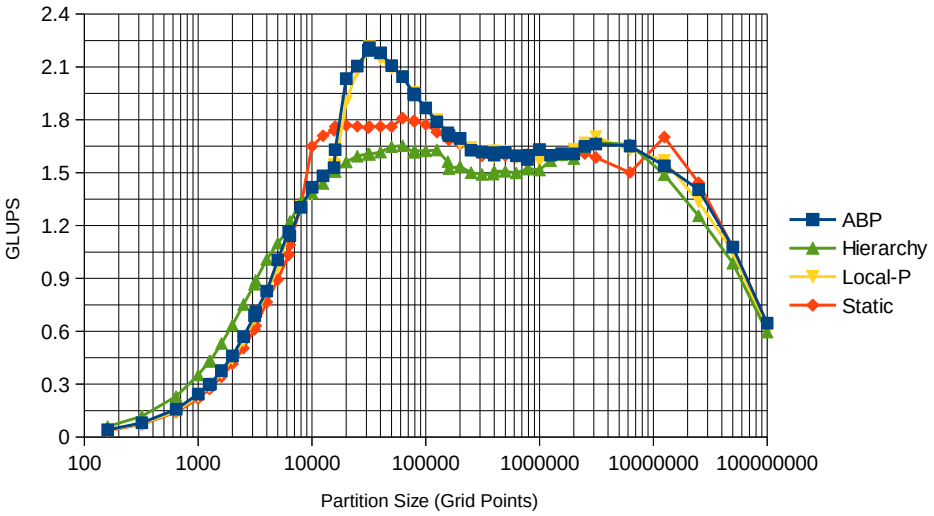

Figure 6.5: HPX-Stencil-Performance on Ariel node

Lattice Updates Per Seconds can be used as a measure of performance. Figure 6.5 shows the performance of HPX-Stencil with different schedulers on Ariel node using 16 cores. LUPS is the number of updates per second. For the very fine-grained tasks, the performance is very low. The same is the case with the coarse-grained tasks. However, this is due to starvation as there is not enough work for the worker threads. For 160 grid points, the LUPS is very low and then it increases with the grain size. The LUPS for ABP and Local-Priority schedulers reaches its peak with 2.2 GLUPS at partition size 32,000. For the hierarchy scheduler, the maximum value is 1.62 GLUPS at 125,000 grid points and for static scheduler it is 1.60 GLUPS at 1,000,000 partition size. Performance declines from 6,250,000 partition size due to poor load balancing.

Results from the performance metrics phase duration and task length are shown in Figures 6.6 and 6.7 respectively. Phase duration specifies the computation time with task manage-

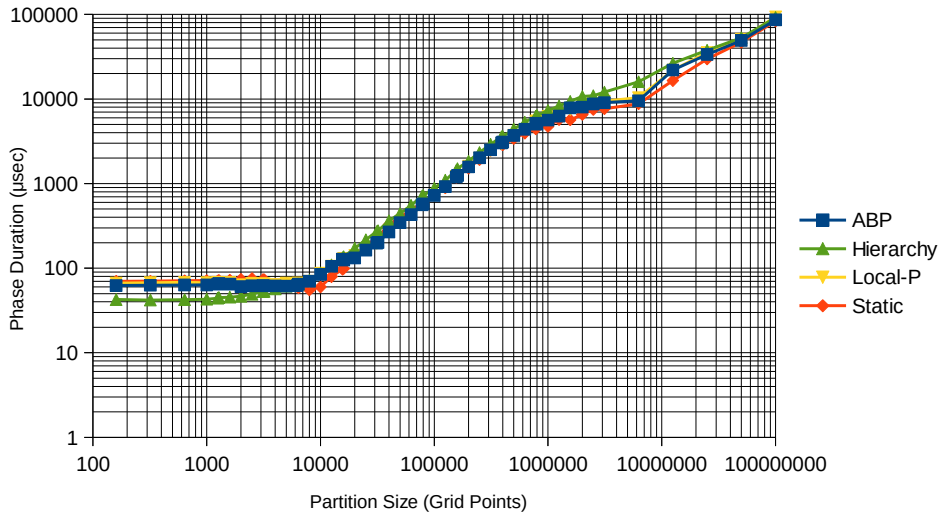

Figure 6.6: Granularity - Phase Duration

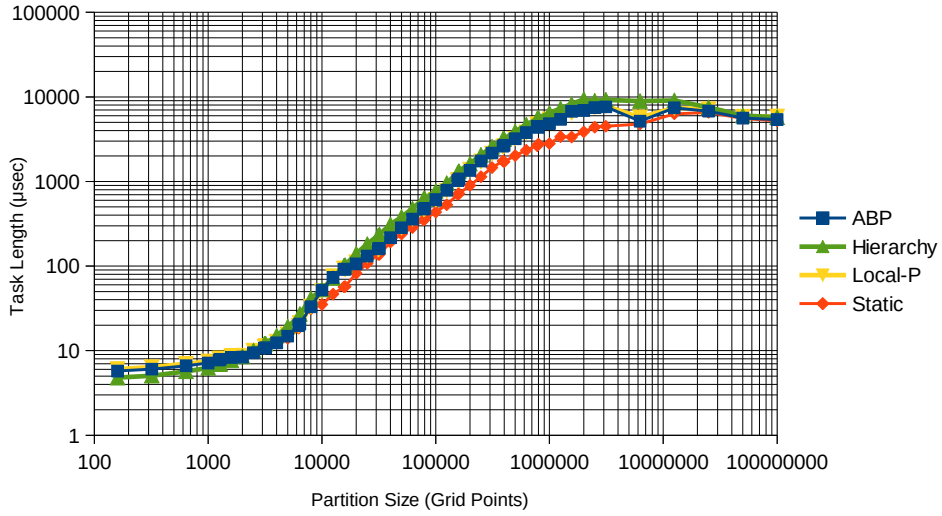

Figure 6.7: Granularity - Task Length

ment overheads. Task length, on the other hand, is the computation time of tasks. When the partition size is less than 20,000 grid points, phase duration is large when compared to the task length which in turn indicates large overheads. For grid points above that, both phase duration and task length lie very close to each other indicating minimal overheads. But for coarse-grained tasks again the overheads rise.

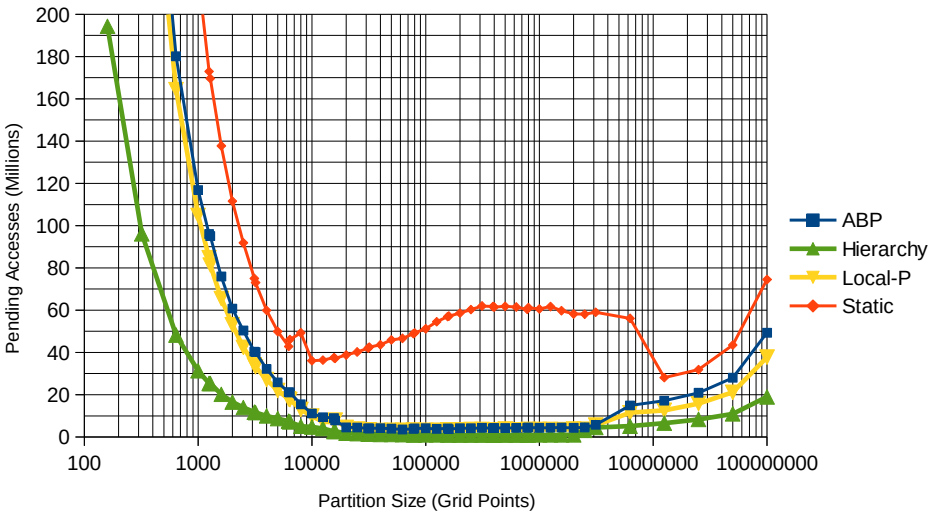

Figure 6.8: Pending Accesses

Figure 6.8 shows the results from the pending accesses performance counter. It is a measure of the number of times a worker thread on the referenced locality tried to find pending HPX threads in its associated queue. This performance counter can be of great help when using a time-stamp is not practical for some multi-threaded architectures like the Intel Xeon Phi.

Execution times of the different schedulers for HPX Stencil on 1, 4, 8 and 16 cores on ariel node is shown in Figures 6.9, 6.10, 6.11, and 6.12.

With the increase in the number of cores, we can see an improvement in the performance of all the four schedulers. The execution time is very high in general when only a single core is used. As the number of cores increases, the execution time decreases and the best execution time of each of the schedulers is when 16 cores is used. Table 6.1 shows the best execution times of each of these schedulers for the different number of cores and the partition size at which this performance is obtained. The best execution times of both the ABP and Local-Priority schedulers is 2.26 seconds for 16 cores at partition size 32,000. The task length and the phase duration for these two schedulers also lie very close to each other at this partition size. The phase duration is around 200 microseconds and the task length is about

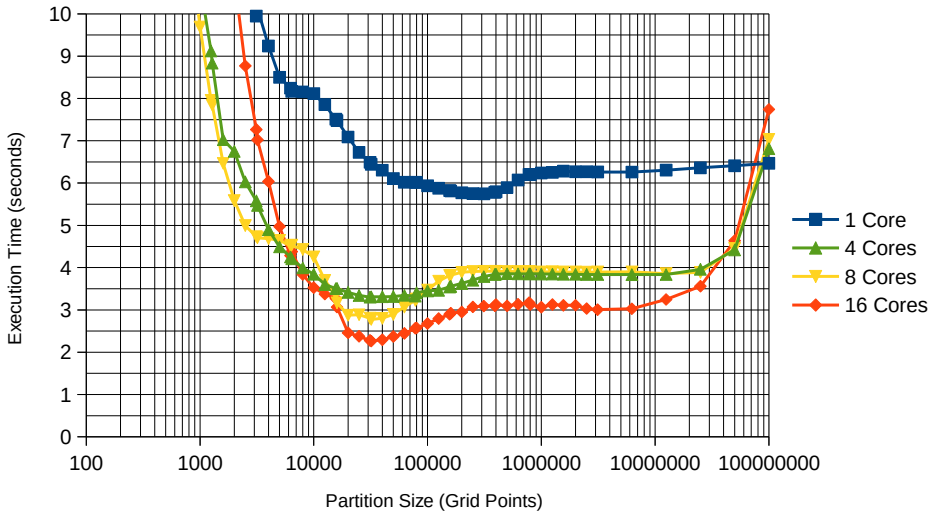

Figure 6.9: ABP - Execution time over 1, 4, 8 and 16 cores

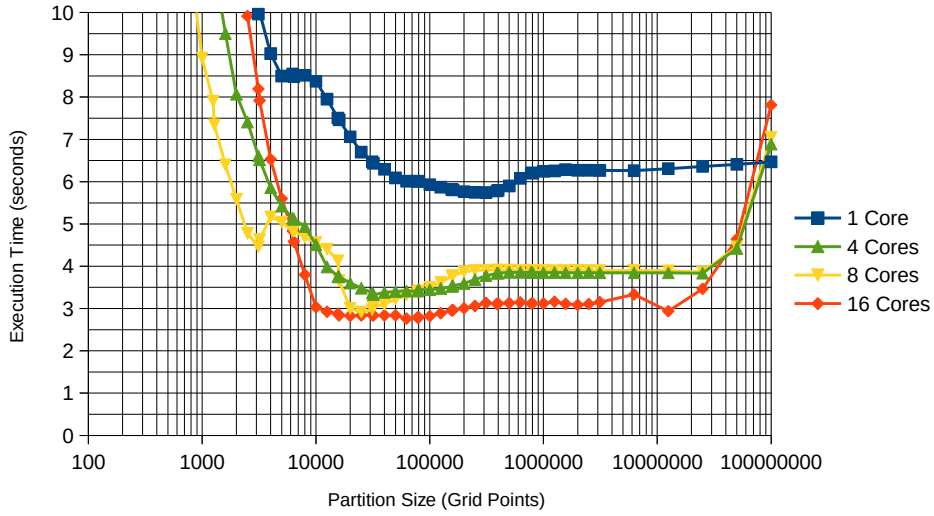

Figure 6.10: Static - Execution time over 1, 4, 8 and 16 cores

160 microseconds. The minimum execution time of the Static scheduler is 2.76 seconds for 16 cores at partition size 62,500, and that of Hierarchy scheduler is 2.99 seconds at 3,125,000 partition size. Here, we are assuming strong scaling as the amount of workload remains the same but the number of processors is increasing. With an increase in the number of cores, the overall performance improves as the workload is parallelized over the other cores. When considering the standard deviation of the samples, some of the partition sizes close to those

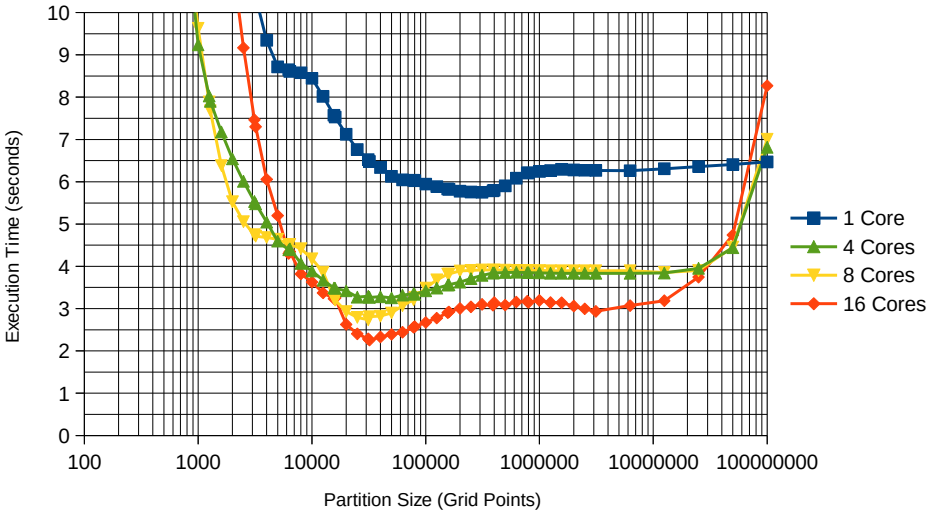

Figure 6.11: Local-P - Execution time over 1, 4, 8 and 16 cores

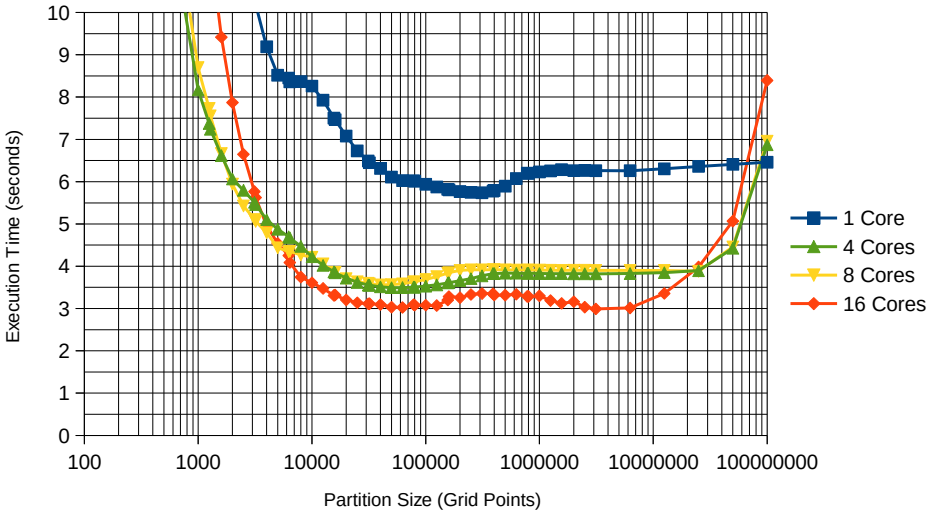

Figure 6.12: Hierarchy - Execution time over 1, 4, 8 and 16 cores

given in the table have execution times that are very similar.

Table 6.1: Minimum Execution time

Scheduler	No. of Cores	Min. Exec. Time (seconds)	Partition Size Range	Phase Duration	Task Length
ABP	1	5.75	312,500	353.46	352.86
	4	3.30	31,250	81.47	72.12
	8	2.77	32,000	135.75	108.85
	16	2.26	32,000	201.48	162.23
Static	1	5.74	312,500	353.24	352.70
	4	3.33	32,000	83.71	66.96
	8	2.91	25,000	112.81	79.10
	16	2.76	62,500	483.88	285.39
Local-P	1	5.75	312,500	353.75	353.12
	4	3.27	32,000	82.68	73.35
	8	2.73	31,250	130.77	104.41
	16	2.26	32,000	200.15	160.96
Hierarchy	1	5.74	312,500	353.11	352.45
	4	3.49	50,000	138.36	125.90
	8	3.56	40,000	224.36	200.24
	16	2.99	3,125,000	12082.97	9239.35

Chapter 7

Future Work

As task length and phase duration provide good insight on how the application is performing with varying grain size, these entities are now added as new performance counters in HPX. Testing these performance counters will be taken up in the future. In addition to this, we plan to conduct performance analysis tests on other HPX benchmarks. Though our initial analysis suggests that LL cache misses incur additional overheads, which cause the execution time to be high even when the idle rate is less for the hierarchy scheduler, we need to further investigate on the other possible causes of overheads.

Our performance study was done by taking into consideration the total overheads and counts for all the OS threads. We plan to perform a study where we look at the individual OS thread information. In addition to this, while performing this performance analysis, we took into consideration the total execution time. A possible approach would be to investigate the performance at certain intervals by using performance counters like queue lengths.

Chapter 8

Conclusions

We present a methodology to evaluate scheduling overheads associated with grain size for parallel applications. We introduced metrics that can determine overheads affected by granularity. We made use of HPX performance counters to measure specific events available at runtime. These counters in turn help us to calculate the thread scheduling overheads with varying grain size. The performance analysis of different HPX schedulers was done for an HPX benchmark. From our performance analysis, we conclude that for the heat distribution benchmark, HPX-Stencil, we can use ABP, Local-Priority or the Static scheduler as there is not much variation in the performance provided by these three schedulers. However, the hierarchy scheduler does not appear to be a right choice for this benchmark as the overheads are high, which leads to a deterioration in performance when using this scheduling policy.

Bibliography

[1] Adaptive thread scheduling techniques for improving scalability of software transactional memory. http://www.open-std.org/jtc1/sc22/wg21/.

[2] Microsoft parallel pattern library. http://msdn.microsoft.com/en-us/library/dd492418.aspx, 2010.

[3] The c++ standards committee. http://www.open-std.org/jtc1/sc22/wg21/, 2011.

[4] Hpx 0.9.9. http://stellar-group.github.io/hpx/docs/hpx.pdf, 2011–2014.

[5] D. Etiemble H. Kaiser A. Tan, J. Falcou. Automatic task-based code generation for high performance domain specific embedded language, July 2014.

[6] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. Slaw: a scalable locality-aware adaptive work-stealing scheduler. In *Parallel and Distributed Processing*, pages 1–12. IEEE, 2010.

[7] T. Heller, H. Kaiser, A. Schafer, and D. Fey. Using hpx and libgeodecomp for scaling hpc applications on heterogeneous supercomputers. Denver, USA, 2013. ACM.

[8] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *Proceedings of the 37th International Symposium on Microarchitecture*. IEEE, 2004.

[9] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. Hpx - a task based programming model in a global address space. Technical report.

[10] C. Krasic, M. Saubhasik, A. Sinha, and A. Goel. Fair and timely scheduling via cooperative polling. Nuremberg, Germany, 2009. ACM.

[11] J. Nakashima, S. Nakatani, and K. Taura. Design and implementation of a customizable work stealing scheduler. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Super Computers*, pages 1–8, New York, NY, USA, 2013. ACM.

[12] 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 28th international conference on Human factors in computing systems*, pages 1–12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.