DYNAMIC ADAPTATION IN HPX - A TASK-BASED PARALLEL RUNTIME SYSTEM

BY

PATRICIA A. GRUBEL, B.S., M.S.

A dissertation submitted to the Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy, Engineering

Specialization in Electrical Engineering

New Mexico State University

Las Cruces New Mexico

August 2016

DEDICATION

Laus Deo

In loving memory of my parents, Peter and Mary Zagone.

ACKNOWLEDGMENTS

Through this journey there has been so much support from people in all areas of my life including exceptional mentors and colleagues, loving family members and wonderful caring friends. Some I will mention in particular here, but there are many more that helped and encouraged me during my graduate studies and made this dissertation possible. So it is not just my dissertation but a very wide group effort.

I thank my advisor, Jeanine Cook, whose knowledge, guidance, and inspiration were instrumental in the success of my studies, research, and completion of this dissertation. Little did I really know what I was getting into when I met you, but from the very start I knew you would be an inspiration. I thank Hartmut Kaiser, my mentor at Louisiana State University, who taught me about the capabilities of HPX and C++. Thank you for sharing your expertise, all your support and encouragement. Your response to any need, including computer resources and HPX issues, was phenomenal. I thank the other committee members, Dr. David Voelz and Dr. Yuho Jin, for their helpful technical discussions, comments, and reviews.

I appreciate Sandia National Laboratories for funding my studies and early research with a graduate research fellowship and the NSF for funding this research through grant 1111798.

I want to also acknowlege the helpful technical disscussions with the guys in the NMSU ACAPS lab, Soumik Banergee, Parsa Amini, Alireza Nazari, Po Chou Su, Samer Haddad, Mohammad Qayum, Nafiul Sadique, and Waleed Kohlani, who did a great job teaching the Computer Performance Analysis class.

I thank my collaborators of the Stellar group at Louisiana State University and the international collaborators who supported me in many ways. Bryce Adelstein-Lelbach supported me early on in the areas of thread scheduling, assessing overheads, and all things C++. Parsa Amini, left ACAPS but still supported me from LSU, with technical discussions, helping me when systems caused me headaches and when I caused my own, and for his careful review of this dissertation. Adrian Serio keeps the team motivated and makes sure resources are available. Thank you for your encouragement and making sure I finished in a timely manner. I got those pokes to keep going even when it seemed I was ignoring them. Also thank you for all of your reviews and suggestions on the papers. I also enjoyed working with Zahra Khatami and Vinay Amayta. I thank Thomas Heller, University of Erlangen-Nuremberg, for all his help with HPX configurations and solving issues and for many technical discussions about HPX.

Also I appreciate the knowledge and help Kevin Huck and Nicholas Chaimov, University of Oregon, gave me with APEX, performance tools, and the Standard C++ INNCABS suite.

I thank my extended family and many friends who encouraged me and prayed

for me. To those who lent their ears to me during all trials, thank you for listening and still encouraging me. You know who you are. I want to especially thank Linn Morrison and her late husband Ted, whom I miss very much. They welcomed me into their home and hearts and made those commutes enjoyable. Linn always asks me when "we" will get "our" Ph.D. and its true she owns this right along with me because she helped me a great deal by being there for me. I am so grateful for all you did for me and am glad to share it with you.

Last but by no means least, I owe a great debt of thanks to my wonderful family especially my husband, Paul. Thank you for encouraging me to start this journey and all through it, listening to me and helping me in every step. And to my children, especially those who had to live with me while I was working on this for different phases of your own education and pursuits, your faith in me and encouragement to continue through the daily grind meant so much to me.

VITA

1973-1976	B.S. Electrical Engineering, New Mexico State University, Las Cruces, NM
1976-1981	M.S. Electrical Engineering, New Mexico State University, Las Cruces, NM
1976-1977	Electronics Engineer, Instrumentation Directorate, White Sands Missile Range, NM
1977-1979	Operations Research Analyst, 6585 Test Group Holloman Air Force Base, NM
1979-1981	Electronics Engineer, 6585 Test Group Holloman Air Force Base, NM
1981-1982	Software Engineer, OAO Corporation Vandenberg Air Force Base, CA
1982-1984	Senior Electronics Engineer, High Energy Laser Program White Sands Missile Range, NM
1984-1985	Senior Systems Engineer, Dynalectron Corp. Alamogordo, NM
1985-1990	Consulting Systems Engineer, Denmar Technical Services Alamogordo, NM
1990-2010	Educator and Tutor, K-12 and College Alamogordo, NM
2010-2010	Graduate Teaching Assistant, New Mexico State University Las Cruces, NM
2011-2013	Graduate Research Fellowship, Sandia National Laboratories, New Mexico State University, Las Cruces, NM
2013-2016	Graduate Research Assistant, New Mexico State University Las Cruces, NM

PUBLICATIONS

Patricia Grubel, Hartmut Kaiser, Kevin Huck and Jeanine Cook, "Using Intrinsic Performance Counters to Assess Efficiency in Task-based Parallel Applications", accepted in IPDPS Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications (HPCMASPA), Chicago, May 27, 2016.

Patricia Grubel, Hartmut Kaiser, Jeanine Cook, and Adrian Serio, "The Performance Implication of Task Size for Applications on the HPX Runtime System", in *2015 IEEE International Conference on Cluster Computing*, Chicago, IL, 2015, pp. 682-689.

PROFESSIONAL AND HONORARY SOCIETIES

Institute of Electrical and Electronics Engineers

Association for Computing Machinery

FIELD OF STUDY

Major Field: Electrical and Computer Engineering

ABSTRACT

DYNAMIC ADAPTATION IN

HPX - A TASK-BASED PARALLEL RUNTIME SYSTEM

BY

PATRICIA A. GRUBEL, B.S., M.S.

Doctor of Philosophy

New Mexico State University

Las Cruces, New Mexico, 2016

Dr. Jeanine Cook, Chair

As parallel computation enters the *exascale* era where applications may run on millions to billions of processors concurrently, all aspects of the computational model need to undergo a transformation to meet the challenges of scaling impaired applications. One class of models aimed towards exascale computation is the task-based parallel computational model. Task-based execution models and their implementations aim to support parallelism through massive multi-threading where an application is split into numerous tasks of varying size that execute concurrently. Thread scheduling mechanisms used to manage application level tasks are a fundamental part of the task-based parallel computational model.

In task-based systems, scheduling threads onto resources can incur large overheads that vary with the underlying hardware. In this work, our goal is to dynamically control task grain size to minimize these overheads. We use performance studies to determine measurable events and metrics derived from them that indicate how tuning task granularity will improve performance. We aim to build a closed loop system that measures pertinent events and dynamically tunes task grain size to improve performance of parallel applications. High Performance ParalleX (HPX), the first implementation of the ParalleX execution model, is a runtime system that employs asynchronous fine-grained tasks and asynchronous communication for improved scaling of parallel applications. HPX is a modular system that has a dynamic performance modeling capability and a variety of thread scheduling policies and queuing models for work stealing and load balancing. It provides the ideal framework for studying parallel applications with the ability to make dynamic performance measurements and implement adaptive mechanisms. Therefore, dynamic tuning of task granularity is developed within the HPX framework.

CONTENTS

LIST OF TABLES xiii

LIST OF FIGURES xx

1 INTRODUCTION 1

1.1 Runtime Adaptivity in Task Based Parallelism 2

1.2 Dissertation Organization 3

2 BACKGROUND 4

2.1 ParalleX Model 4

2.2 Futures and Dataflow Constructs Eliminate Global Barriers . . . 6

2.2.1 Futures 7

2.2.2 Dataflow 8

2.3 HPX Runtime System 9

2.3.1 HPX Thread Scheduling 10

2.3.2 Parcel Transport Layer 14

2.3.3 Local Control Objects 15

2.3.4 Active Global Address Space 15

2.3.5 Performance Monitoring System 16

2.4 Task Granularity 17

3 RELATED WORK 20

3.1 Adaptive Task Grain Size 20
3.2 Adaptive Schedulers 26
4 DISSERTATION CONTRIBUTION 32
5 PERFORMANCE IMPLICATION OF TASK GRANULARITY . 36
5.1 Task Granularity Experimental Methodology 36
5.1.1 Stencil Benchmark 37
5.1.2 Performance Metrics 40
5.1.3 Experimental Platforms 44
5.2 Task Granularity Experimental Results 45
5.2.1 Idle-rate 50
5.2.2 HPX Thread Management Overhead 58
5.2.3 Wait Time 63
5.2.4 Combined Costs: HPX Thread Management and Wait Time 69
5.2.5 Thread Pending Queue Accesses 70
5.3 Summary of Task Granularity Experiments 79
6 USING INTRINSIC PERFORMANCE COUNTERS TO ASSESS OVERHEADS DURING EXECUTION 81
6.1 Challenges Using Performance Monitoring Tools 84
6.2 Performance Counter Experimental Methodology 87
6.2.1 Benchmarks 87
6.2.2 Configurations 89

6.2.3 Performance Counter Metrics 91
6.3 Performance Counter Experiments 95
6.4 Performance Counter Experimental Results 97
6.5 Summary of Performance Counter Experiments 114
7 ADAPTIVE METHODOLOGIES 116
7.1 Tuning Task Granularity Example 118
7.2 Tuning Task Granularity Results 120
8 CONCLUSIONS AND FUTURE WORK 123
APPENDICES 128
A TASK GRANULARITY SUPPLEMENTARY RESULTS 128
A.1 Task Granularity Results Sandy Bridge 128
A.2 Task Granularity Results Ivy Bridge 136
B PERFORMANCE ASSESMENT (INNCABS) SUPPLEMENTARY 144
B.1 HPX vs. C++11 Standard 144
B.2 Overheads Using HPX Performance Counters 152
B.3 Offcore Bandwidth Utilization 157
REFERENCES 162

LIST OF TABLES

1 Platform Specifications for Task Granularity Experiments 45

2 Correlation of Metrics to Execution Time - Haswell 51

3 C++11 Standard INNCABS Executed with TAU and HPCTookKit 86

4 Translation of Syntax: C++11 Standard to HPX 89

5 Platform Specifications for Performance Counter Experiments . . 90

6 Software Build and Run Specifications for INNCABS 95

7 Benchmark Classification and Granularity 97

8 Platform Specifications for Performance Counter Experiments . . 119

9 Tuning Task Granularity Results 121

LIST OF FIGURES

1 Global Barriers and Thread Idle Time 7

2 Constraint Based Synchronization Using Futures 8

3 HPX Runtime System 10

4 HPX Thread State Diagram 12

5 Priority Local-FIFO scheduler 14

6 Parallel State Space Search Engine (ParSSSE) Abstraction Layers 21

7 ParSSSE Adaptive Grain Size Control (from [51]) 22

8 Dependencies of Heat Distribution Benchmark 38

9 Sandy Bridge: Execution Time vs. Task Granularity (partition size) 48

10 Ivy Bridge: Execution Time vs. Task Granularity (partition size) 48

11 Haswell: Execution Time vs. Task Granularity (partition size) . . 49

12 Xeon Phi: Execution Time vs. Task Granularity (partition size) . 49

13 Haswell (4 cores): Idle-Rate 52

14 Haswell (8 cores): Idle-Rate 52

15 Haswell (16 cores): Idle-Rate 53

16 Haswell (28 cores): Idle-Rate 53

17 Xeon Phi (8 cores): Idle-Rate 54

18 Xeon Phi (16 cores): Idle-Rate 54

19 Xeon Phi (32 cores): Idle-Rate 55
20 Xeon Phi (60 cores): Idle-Rate 55
21 Haswell (4 cores): Thread Management per Core 59
22 Haswell (8 cores): Thread Management per Core 59
23 Haswell (16 cores): Thread Management per Core 60
24 Haswell (28 cores): Thread Management per Core 60
25 Xeon Phi (8 cores): Thread Management per Core 61
26 Xeon Phi (16 cores): Thread Management per Core 61
27 Xeon Phi (32 cores): Thread Management per Core 62
28 Xeon Phi (60 cores): Thread Management per Core 62
29 Wait Time per HPX-Thread (Haswell) 63
30 Haswell (4 cores): Wait Time per Core 65
31 Haswell (8 cores): Wait Time per Core 65
32 Haswell (16 cores): Wait Time per Core 66
33 Haswell (28 cores): Wait Time per Core 66
34 Xeon Phi (8 cores): Wait Time per Core 67
35 Xeon Phi (16 cores): Wait Time per Core 67
36 Xeon Phi (32 cores): Wait Time per Core 68
37 Xeon Phi (60 cores): Wait Time per Core 68
38 Haswell (4 cores): Thread Management and Wait Time per Core . 71
39 Haswell (8 cores): Thread Management and Wait Time per Core . 71

40 Haswell (16 cores): Thread Management and Wait Time per Core 72

41 Haswell (28 cores): Thread Management and Wait Time per Core 72

42 Xeon Phi (8 cores): Thread Management and Wait Time per Core 73

43 Xeon Phi (16 cores): Thread Management and Wait Time per Core 73

44 Xeon Phi (32 cores): Thread Management and Wait Time per Core 74

45 Xeon Phi (60 cores): Thread Management and Wait Time per Core 74

46 Haswell (4 cores): Pending Queue Accesses 75

47 Haswell (8 cores): Pending Queue Accesses 75

48 Haswell (16 cores): Pending Queue Accesses 76

49 Haswell (28 cores): Pending Queue Accesses 76

50 Xeon Phi (8 cores): Pending Queue Accesses 77

51 Xeon Phi (16 cores): Pending Queue Accesses 77

52 Xeon Phi (32 cores): Pending Queue Accesses 78

53 Xeon Phi (60 cores): Pending Queue Accesses 78

54 Alignment: HPX vs. C++11 Standard 100

55 Pyramids: HPX vs. C++11 Standard 101

56 Strassen: HPX vs. C++11 Standard 102

57 Sort: HPX vs. C++11 Standard 103

58 FFT: HPX vs. C++11 Standard 105

59 UTS: HPX (C++11 Standard fails) 106

60 Alignment: Overheads 108

61 Pyramids: Overheads 109
62 Strassen: Overheads 109
63 FFT: Overheads 110
64 UTS: Overheads 110
65 Alignment: Offcore Bandwidth Utilization 111
66 Pyramids: Offcore Bandwidth Utilization 112
67 Strassen: Offcore Bandwidth Utilization 112
68 FFT: Offcore Bandwidth Utilization 113
69 APEX Integration with HPX 117
70 Sandy Bridge (4 cores): Idle-Rate 128
71 Sandy Bridge (8 cores): Idle-Rate 128
72 Sandy Bridge (12 cores): Idle-Rate 129
73 Sandy Bridge (16 cores): Idle-Rate 129
74 Sandy Bridge (4 cores): HPX Thread Management per Core . . . 130
75 Sandy Bridge (8 cores): HPX Thread Management per Core . . . 130
76 Sandy Bridge (12 cores): HPX Thread Management per Core . . 131
77 Sandy Bridge (16 cores): HPX Thread Management per Core . . 131
78 Sandy Bridge (4 cores): Wait Time per Core 132
79 Sandy Bridge (8 cores): Wait Time per Core 132
80 Sandy Bridge (12 cores): Wait Time per Core 133
81 Sandy Bridge (16 cores): Wait Time per Core 133

82 Sandy Bridge (4 cores): Thread Management and Wait Time . . . 134

83 Sandy Bridge (8 cores): Thread Management and Wait Time . . . 134

84 Sandy Bridge (12 cores): Thread Management and Wait Time . . 135

85 Sandy Bridge (16 cores): Thread Management and Wait Time . . 135

86 Ivy Bridge (4 cores): Idle-Rate 136

87 Ivy Bridge (8 cores): Idle-Rate 136

88 Ivy Bridge (16 cores): Idle-Rate 137

89 Ivy Bridge (20 cores): Idle-Rate 137

90 Ivy Bridge (4 cores): HPX Thread Management per Core 138

91 Ivy Bridge (8 cores): HPX Thread Management per Core 138

92 Ivy Bridge (16 cores): HPX Thread Management per Core 139

93 Ivy Bridge (20 cores): HPX Thread Management per Core 139

94 Ivy Bridge (4 cores): Wait Time per Core 140

95 Ivy Bridge (8 cores): Wait Time per Core 140

96 Ivy Bridge (16 cores): Wait Time per Core 141

97 Ivy Bridge (20 cores): Wait Time per Core 141

98 Ivy Bridge (4 cores): Thread Management and Wait Time 142

99 Ivy Bridge (8 cores): Thread Management and Wait Time 142

100 Ivy Bridge (16 cores): Thread Management and Wait Time 143

101 Ivy Bridge (20 cores): Thread Management and Wait Time 143

102 Sparselu: HPX vs. C++11 Standard 144

103 Round: HPX vs. C++11 Standard 145
104 NQueens: HPX (C++11 Standard fails) 146
105 Health: HPX (C++11 Standard fails) 147
106 FIB: HPX (C++11 Standard fails) 148
107 Floorplan: HPX (C++11 Standard fails) 149
108 QAP: HPX (C++11 Standard fails) 150
109 Intersim: HPX (C++11 Standard fails) 151
110 Sparselu: Overheads 152
111 Round: Overheads 152
112 Sort: Overheads 153
113 NQueens: Overheads 153
114 Health: Overheads 154
115 FIB: Overheads 154
116 Floorplan: Overheads 155
117 QAP: Overheads 155
118 Intersim: Overheads 156
119 Sparselu: Offcore Bandwidth Utilization 157
120 Round: Offcore Bandwidth Utilization 157
121 NQueens: Offcore Bandwidth Utilization 158
122 Health: Offcore Bandwidth Utilization 158
123 FIB: Offcore Bandwidth Utilization 159

124 Floorplan: Offcore Bandwidth Utilization 159

125 QAP: Offcore Bandwidth Utilization 160

126 UTS: Offcore Bandwidth Utilization 160

127 Intersim: Offcore Bandwidth Utilization 161

1 INTRODUCTION

High Performance Computing (HPC) is expected to reach *exascale* by the end of the decade. All aspects of the computational model will evolve to support massive amounts of concurrency. Processor core and memory architectures will undergo design changes and architectures will continue to increase core and thread per core counts with tighter integration of many core accelerators. Runtimes and programming models will evolve to improve *scalability*, *programmability*, *performance*, *portability*, *resilience*, and *energy efficiency*. The Exascale Computing Study [40] concluded that a new execution model and programming methodology is required to achieve these goals for contemporary scaling impaired applications and future *exascale* applications. The study specifically recommends the possible solution of utilizing "modern, more expressive and asynchronous programming paradigms and languages" [40] to address scaling impaired applications with inherently fine-grained tasks. An underlying hypothesis of this 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 exemplified by the MPI application programming interface.

1.1 Runtime Adaptivity in Task Based Parallelism

Execution models (and runtime systems that implement them) that support scalability in massively parallel systems are beginning to appear in the HPC community. Runtime libraries that support massive task-based parallelism include Intel®TBB [18], Charm++ [48], Qthreads [6], and HPX [37]. Also, Intel®Cilk™Plus [41] and OpenMP 3.0 [4] have added language extensions for task-based parallelism. Chapel [14] is an experimental programming language with task-based parallelism. These models and their implementations aim to support parallelism through massive multi-threading where an application is split into numerous tasks or threads of varying size that execute concurrently. Runtime adaptive resource management and decision making are necessary components 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.

An important component of these runtimes that can be dynamically adapted to optimize performance is the thread scheduler, the mechanism that schedules the application level tasks. Many of these runtimes implement different thread schedulers that result in widely varying overheads for different task sizes or granularities [46]. This provides an opportunity to optimize performance by dynamically

adapting the thread scheduling algorithm and/or the task granularity. Our goal is to determine criteria and subsequent methods for tuning grain size.

1.2 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 presents background information including discussions on the ParalleX execution model, the effectiveness of *future* and *data-flow* objects, the HPX runtime system, and the effects of task granularity on parallel applications. Chapter 3 explores research related to adaptive techniques as applied to parallel applications with special emphasis on thread scheduling and task grain size. The hypothesis and resulting contributions of this research are presented in Chapter 4. The methodology used to measure the effects of task grain size and the experimental results are observed in Chapter 5. The use of the performance monitoring capabilities in HPX to measure events and provide metrics for runtime adaptivity are demonstrated in the experimental methodology in Chapter 6. The results from benchmarks with a variety of task granularity and synchronization requirements are also presented. In Chapter 7, we explore dynamic adaptation with the integration of HPX and the Autonomic Performance Environment for eXascale (APEX) library [33]. An example and results for tuning grain size based on information gathered from the HPX performance monitoring framework are presented. Finally, the conclusion and future work is presented in Chapter 8.

2 BACKGROUND

Using a task-based parallel computational model for scaling impaired parallel applications is one possible solution toward solving the challenges faced to achieve *exascale* computation. This section presents background information about the ParalleX execution model and its first implementation, HPX, the task-based runtime system used for this research.

2.1 ParalleX Model

The ParalleX [36] execution model departs from traditional communicating sequential processes (CSP) with a new paradigm that aims to implement new forms of fine- and medium-grain task parallelism to increase the total amount of concurrent operations. It makes efficient use of task parallelism by the elimination of explicit and implicit global barriers for more efficient use of processors and reduction of overheads. ParalleX supports a new form of global address space, called the Active Global Address Space (AGAS), that allows dynamic parallel processes to span multiple and shared nodes, providing context for local executing threads. Migration of work across nodes is allowed to follow the distributed state of a task that is made up of a set of threads executing on different nodes. To accomplish this, ParalleX uses parcels (a form of active messages [59, 27]) that are essentially explicit message-driven computations that reduce and hide latency

by moving work to data as opposed to the traditional method of moving data to work, enabling the runtime to execute work close to where the data is located. ParalleX provides *dataflow* [21, 22, 10] and *future* [11, 29, 32] synchronization semantics to replace global barriers and provide constraint-based scheduling.

To achieve massive concurrency required by advances in experimental sciences and informatics, the HPC community is moving toward execution models that target asynchronous task parallelism. HPX [37], the first implementation of ParalleX [36], implements asynchronous task parallelism in a homogeneous programming interface for both intra-node (local) and inter-node (distributed) parallelism. This research deals with the intra-node parallelism, specifically thread scheduling. Through HPX's homogeneous programming model for local and remote operations, the results of this research will extend to distributed execution of applications on multiple nodes.

Execution models that target fine-grained asynchronous task parallelism allow more efficient system utilization of parallel architectures. However, fine-grained task parallelization can generate increased overheads associated with creation, deletion, and management of massive quantities of tasks. This work characterizes overheads and employs dynamic adaptive mechanisms in the HPX framework to minimize overheads and improve performance and scalability.

To facilitate this work, we characterize task grain size in the HPX framework and measure the effects of task granularity on scalability. We extend our ini-

tial work on characterizing task granularity to include dynamic measurements of task grain size and associated overheads for parallel benchmarks with a variety of task granularities, parallel constructs, and synchronization requirements. We incorporate the findings of the above studies with experimental methodologies to implement tuning of grain size in parallel benchmarks.

The following sections give background information on *future* and *dataflow* objects, the HPX runtime system, including its thread scheduling mechanisms and performance monitoring capability, and information about task granularity in task-based runtime systems.

2.2 Futures and Dataflow Constructs Eliminate Global Barriers

Most current parallel applications are permeated with implicit or explicit global barriers impeding progress of computation. Global barriers prevent computational progression when a thread reaches the barrier until all threads arrive at the barrier. Once all threads complete computations and reach the barrier, usually only one thread performs the reduction, further impeding computation as illustrated in Figure 1. Current solutions for task parallelism use continuation models that implement *futures* or *dataflow* constructs to replace implicit or explicit global barriers. HPX provides the means of eliminating global barriers with the implementation of both *futures* and *dataflow* objects as Local Control Objects (LCOs)(Section 2.3.3).

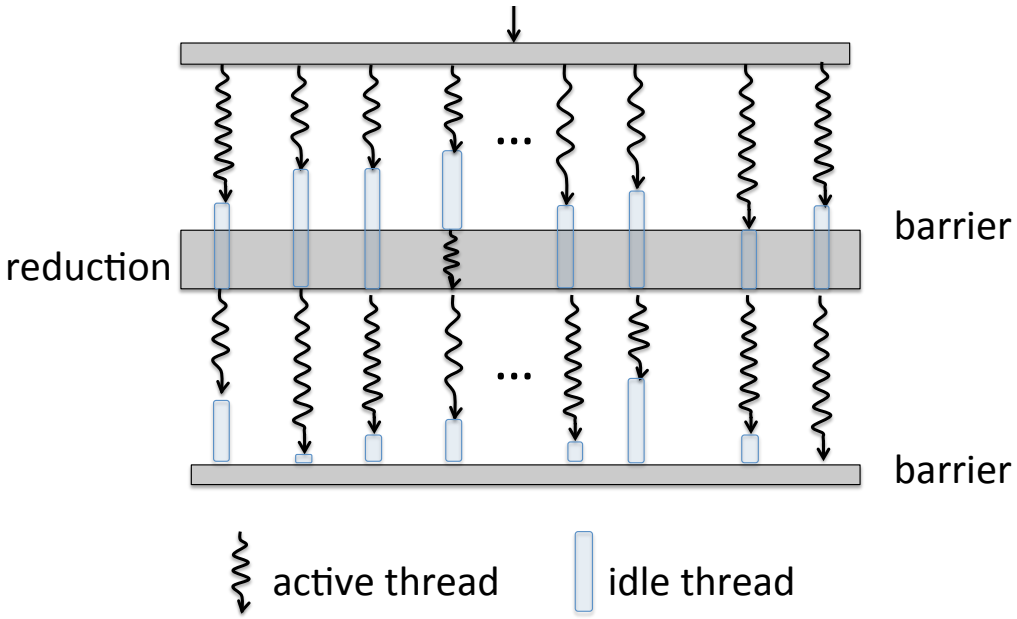

Figure 1: Global Barriers and Thread Idle Time

2.2.1 Futures

A *future* [11, 29, 32] is a proxy for a result that is not known because it may not have been computed yet. The thread that requests a value from a *future* will suspend until the value is available, enabling the thread manager to schedule other work. If the value is already available, the requesting thread will continue computation. Once the result of a *future* becomes available, any HPX thread waiting for it will be reactivated by the *future*. HPX threads are lightweight threads that are scheduled onto coarser grained operating system (OS) threads, also referred to as worker threads. The scheduling of HPX threads by worker threads is explained in more detail in section 2.3.1. Figure 2 illustrates the execution of a *future* and synchronization of tasks dependent on its result.

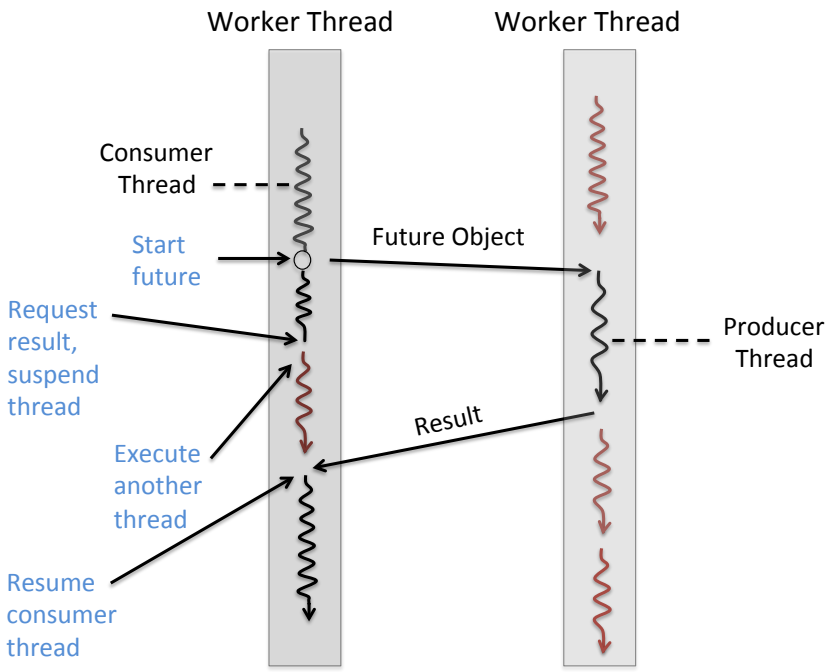

Figure 2: Constraint Based Synchronization Using Futures

2.2.2 Dataflow

Dataflow objects [21, 22, 10] are mechanisms that facilitate parallel computations without global barriers. A *dataflow* LCO manages data dependencies of asynchronous operations by waiting for the results from a set of *futures*. Once the specified *futures* are ready, the *dataflow* object activates a continuation function as a new thread passing encapsulated results from the *futures*.The dataflow itself is exposed as a future representing the result computed by the continuation. Parallel programming models utilizing *dataflow* objects build dynamic execution trees.

2.3 HPX Runtime System

In order to tackle *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.

HPX [37] is a general purpose C++ runtime system for parallel and distributed applications. The HPX Application Programming Interface (API) provides a homogeneous programming model for parallelization of applications on conventional and distributed architectures. The API closely adheres to the C++11/14 Standards [53] - [54], and related proposals to the standardization committee. HPX implements interfaces related to multi-threading (such as *future*, *thread*, *mutex*, and *async*) as defined by the C++ Standard and extends the interfaces in the C++11 threading system for *dataflow* and distributed programming. Conform-

ing to C++ facilitates code portability of HPX applications.

The five main modules of HPX (Figure 3) are: Thread Scheduling System, Parcel Transport Layer, Local Control Objects, Active Global Address Space (AGAS), and Performance Monitoring System and are described in the following sections.

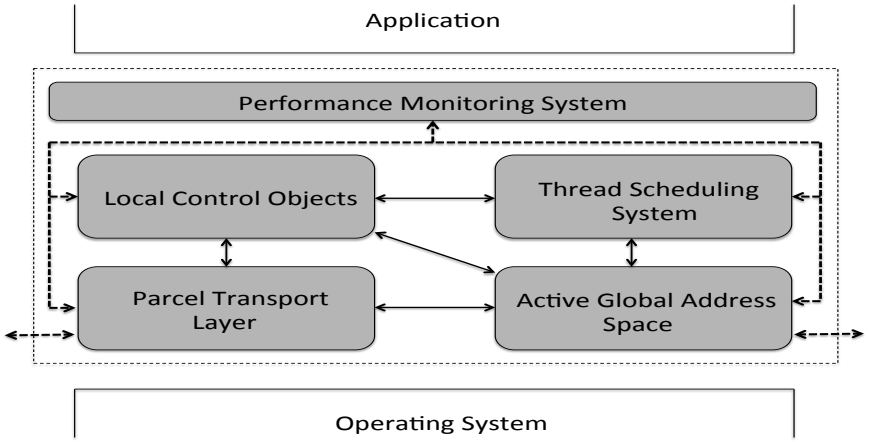

Figure 3: HPX Runtime System

2.3.1 HPX Thread Scheduling

HPX threads are instruction streams that are first class objects. Each HPX thread maintains a thread state, an execution frame, and an operation specific set of registers. HPX threads are implemented as user-level threads utilizing the hybrid-threading (M:N) model. Hybrid-threading implementations use a pool of N kernel threads (usually one kernel thread per processing unit or core) to execute M library threads. HPX threads are cooperatively (non-preemptively) scheduled by the thread scheduler on top of one kernel thread (OS thread, also referred to

as worker thread) per core. This way, HPX threads can be scheduled without a kernel transition, ensuring high performance. Also, work can continue to be executed by the OS thread even if an HPX thread is suspended. The scheduler will not preempt a running HPX thread until it finishes execution or cooperatively yields its execution, minimizing the expensive cost of context switches. This implementation of the scheduling model enables each core to process millions of application threads per second for fine-grained threading.

The thread manager currently implements several scheduling policies. Measurements presented by this research are obtained from execution of applications using a priority-based first-in-first-out (FIFO) scheduling scheme, where each OS thread works from a separate queue of tasks. This is similar to Thread Building Blocks(TBB) [35] and Cilk++ [41]. When execution begins, the thread manager captures the machine topology and maps the number of OS threads to its allocated resources according to the binding specified by the user. By default, it will use all available cores and will create one static OS thread per core (or per hardware thread in the case of systems with hyper-threading or multiple threads per core). With command line options the user can bind OS threads to the cores either manually or by several built in policies. Also, scheduling policies are modular and new ones can be added to HPX.

All HPX thread scheduling policies use a dual-queue (staged and pending) scheme to manage threads. The HPX thread state diagram, Figure 4, illustrates

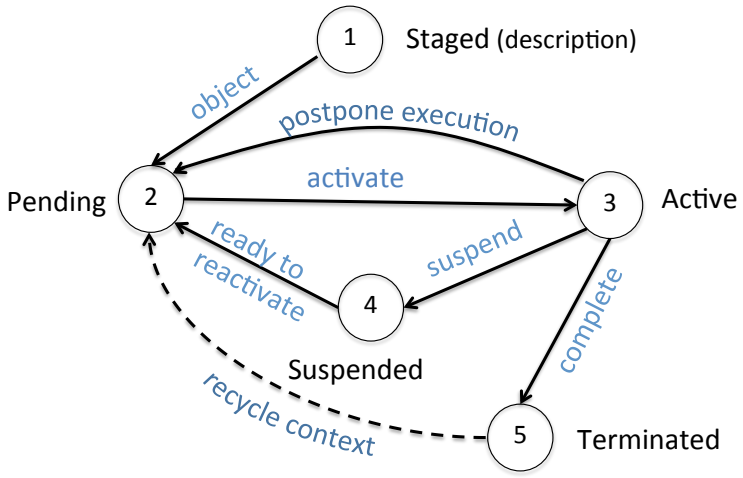

Figure 4: HPX Thread State Diagram

the transistion of threads among the following five states:

1. *Staged*: An HPX thread is first created by the thread scheduler as a thread description, and placed in a staged queue. Staged threads have not yet been allocated a context, they are easily created and can be moved to queues associated with other memory domains without associated memory costs.

2. *Pending*: 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 be executed.

3. *Active*: Once an HPX thread is executing, it can suspend itself for synchronization or communication. An active thread can also transition from active to pending for short periods of execution postponement to use primitive synchronizations, such as spin-locks.

4. *Suspended*: An HPX thread that has suspended execution is waiting for data or resources to resume. Once the dependencies are available the thread will be placed back in the pending queue.

5. *Terminated*: When a thread has completed execution, it is placed in a terminated list, and the context of terminated threads will be recycled for new threads.

The Priority Local-FIFO scheduler, a composition of the Priority Local scheduling policy and the lock free FIFO queuing policy, is used to obtain the results presented in this dissertation. The Priority Local Scheduler uses one pending and one staged queue per worker thread with normal priority, has a specified number of high priority 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, as illustrated in Figure 5. This process is also referred to as 'work-stealing'.

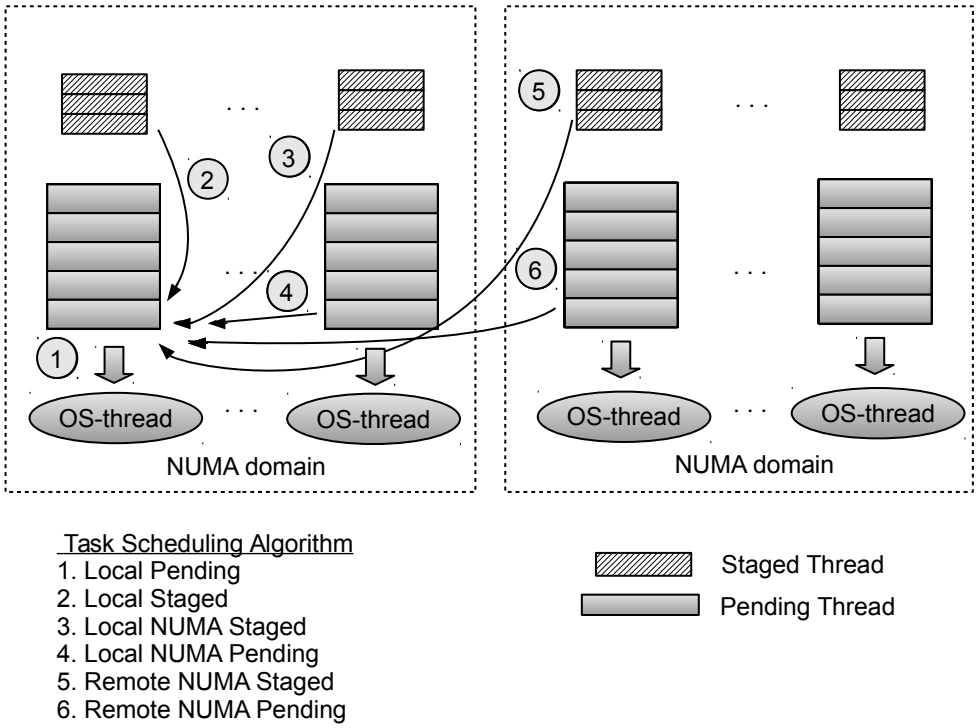

Figure 5: Priority Local-FIFO scheduler

2.3.2 Parcel Transport Layer

The Parcel Transport Layer manages parcels (HPX active messages [59, 27]) providing HPX applications the ability to send messages that invoke methods on remote localities. Parcels are used to send work to data, move data, and return results. When an action is required on a remote locality, it is converted to a parcel encompassing the global identifier (GID) of the action on the remote location, the arguments for the action, and a possible continuation. Upon receipt of a parcel on the requested locality, the parcel handler activates the action as an HPX thread.

2.3.3 Local Control Objects

Local Control Objects (LCOs) control parallelization and synchronization of HPX applications. An LCO is any object that may create, activate, or reactivate an HPX thread. LCOs support managing execution flow, event driven threads, thread suspension and reactivation, and protection of shared resources. LCOs replace global barriers with constraint-based synchronization through the principal LCOs in HPX, *futures* (Section 2.2.1) and *dataflow* objects (Section 2.2.2). Other LCOs include suspended threads (can reactivate themselves) and traditional synchronization primitives including mutexes, semaphores, spin-locks, and barriers. Through the use of LCOs in a global address system, HPX provides a means of controlling synchronization on any locality. A locality is usually a node in a compute cluster but can be specified by the user or application. For example, a user can specify each socket on a node to be a locality.

2.3.4 Active Global Address Space

Active Global Address Space (AGAS) implements global addresses spanning all localities. AGAS assigns a GID for every object on the system, that has to be accessed remotely. Systems like X10 [16], Chapel [14], and UPC [58, 28] use Partitioned Global Address Space (PGAS) where data is statically partitioned to locations, as opposed to AGAS. AGAS adapts the address space throughout the run of an application and supports migration of objects. When an object is

migrated in HPX the associated GID remains the same and AGAS updates the address mapping system.

2.3.5 Performance Monitoring System

The Performance Monitoring System of HPX provides mechanisms to monitor behavior of hardware, the application, the HPX runtime, or the operating system 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 execution time by the application or the runtime system. The performance counters are used to provide information about how well the runtime system or the application is performing. Counter data can help determine system bottlenecks and fine-tune system and application performance. The HPX runtime system, its networking, global addressing, thread scheduling and other systems provide counter data that an application can consume to provide information to the user as to how well the application is performing. HPX also implements hardware counters through the PAPI [23, 52] interface, giving the ability to monitor the underlying hardware system.

Applications can also use counter data to determine how much system resources to consume. For example, an application that transfers data over the network can consume counter data from a network switch to determine how much data to transfer without competing for network bandwidth with other network

traffic. The application can use the counter information to adjust its transfer rate as the bandwidth usage from other network traffic increases or decreases.

Performance counters are HPX components that expose a predefined interface. HPX exposes special API functions to create, manage, read the counter data, and release instances of performance counters. Performance counter instances are accessed by name, and these names have a predefined structure. The advantage of this is that any performance counter can be accessed locally or remotely (from a different locality).

Counter data may be accessed in real time during the execution of an application. In HPX this capability is the basis for building higher-level runtime-adaptive mechanisms that may change system or application parameters with the goal of improving performance, energy consumption, or parallel efficiency.

Through HPX's predefined interface and special API functions, new performance counters are easily incorporated. The counters are easily accessible during execution through an API or through the command-line interface for post processing performance analysis. Runtime introspection of performance counters facilitates the development of adaptive mechanisms such as adaptive thread schedulers.

2.4 Task Granularity

An important factor for thread scheduling management is the granularity of the tasks distributed among processors. The grain size of a task is the amount

of time the task executes continuously without suspension due to synchronization or communication. Fine-grained tasks have small amounts of computation between suspension events for communication or synchronization, while coarse-grained tasks perform computations continuously for long periods of time. The use of fine-grained tasks can optimize load balancing amongst the parallel processors, but if the application is characterized by a massive number of fine-grained tasks, significant overheads may be produced by task creation, management, communication and synchronization, as well as contention on resources such as memory allocation for stacks. Coarse-grained tasks make it difficult to perform efficient load balancing amongst the processors causing idle-time, but are associated with overheads that account for a smaller percentage of total execution time.

The solution to these problems appears to be simply to use an efficient grain size for the application. 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 by detecting granularity (specific to the underlying hardware) and subsequently tuning either the scheduling mechanisms and/or 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 execution time. Steps toward runtime adaptivity include describing the relationship between overheads and task granularity, understand-

ing how schedulers affect performance for different task sizes, ascertaining metrics that can be used to dynamically determine task granularity and sources of overheads, and finally build a feedback loop in the runtime system to make adaptations during execution for improved performance or use of resources.

3 RELATED WORK

This section focuses on research related to adaptive techniques as applied to parallel applications with special emphasis on thread scheduling and task grain size. There have been numerous studies of dynamic scheduling mechanisms. The problems have become increasingly complex with increases in the number of cores and the complexity of the memory hierarchy. Developers of task parallel implementations are expanding capabilities by incorporating functionalities such as multiple data dependencies like the utilization of *dataflow* LCOs in HPX, and increased functionality of task parallelism in OpenMP 4.0 [4]. Most of the research relevant to adaptive scheduling has been accomplished using parallel applications that employ loop parallelism, although work in the area of task parallelism has increased significantly in recent years. We explore research methods that are related to scheduling techniques within the HPX platform.

3.1 Adaptive Task Grain Size

One way of tuning grain size is to employ a technique, commonly referred to as cut-off where an execution tree stops parallelizing work at some depth of the tree and continues by serializing subsequent computation. The cut-off technique is useful for applications with recursive kernels and graph applications.

Y. Sun et al. [51], employ grain size adaptation using an execution tree cut-off

strategy in the implementation of a framework over the Charm++ [48] runtime system. Charm++ supports message driven task parallelism with limited migration for parallel applications on distributed computer systems. The framework, Parallel State Space Search Engine (ParSSSE) [51]) is an abstraction above the runtime system that enables users to solve state space search problems without directly programming the parallelization details (Figure 6).

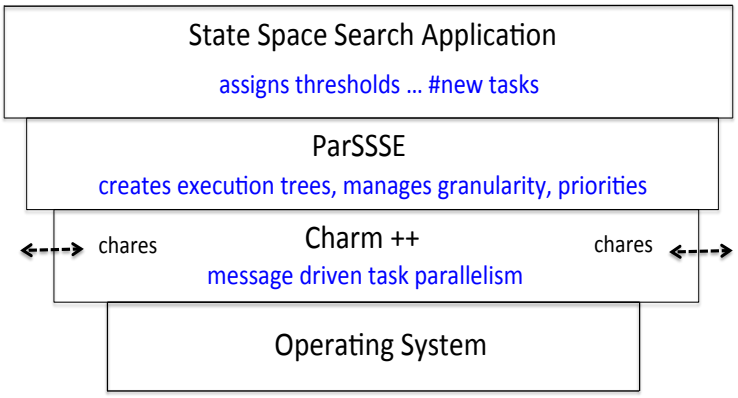

Figure 6: Parallel State Space Search Engine (ParSSSE) Abstraction Layers

State space search applications perform searches of massive size graphs such that the graph is developed as the search progresses, thereby constructing and searching a dynamic graph. ParSSSE begins execution by generating a task for each node of the execution tree, generating fine-grained tasks to facilitate spreading work amongst the processors. This quickly establishes saturation of work for all the processors. Each of the initial nodes then generates an execution tree for subsequent tasks that are parallelized until a depth threshold (specified by the

application) is met. After that depth, the execution of tasks is serialized producing medium-grained tasks to minimize overheads of the search. This generates static task granularity dependent upon user specifications, requiring the user to determine, perhaps through trial and error, proper depth for optimal performance on a particular platform.

ParSSSE is extended 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. If the estimated expansion time exceeds ten times the creation and scheduling overheads, tasks are split into a user specified number of new parallel tasks (Figure 7).

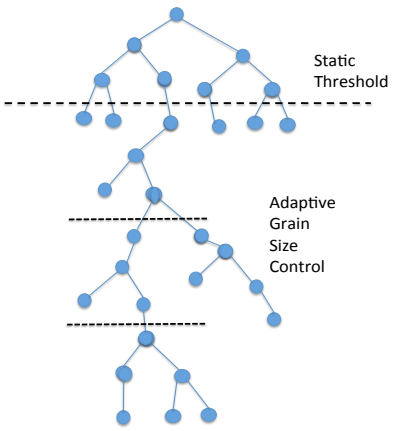

Figure 7: ParSSSE Adaptive Grain Size Control (from [51])

Results were obtained solving an NQueens [12] problem and executing the Unbalanced Tree Search (UTS) [45] benchmark. Comparisons were made using ParSSSE with and without the adaptive grain size algorithm. The results using

adaptive grain size performed within 10% of the best manually chosen depth threshold. It is mentioned that the variances of the samples were smaller when using the adaptive algorithm, although the amount of variance is not reported. This research illustrates 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 algorithms. Some questions that this research invokes are:

1. Were the overheads assessed during execution?
2. Would it be beneficial to adjust the percentage of overheads used to determine when to enforce adaptive granularity?
3. Why use adaptive cutoff when execution is "ten times" the amount of the overhead?
4. Should the tuning be reassessed periodically, especially for long running applications?

Dynamic scheduling has been employed for a long time, for OpenMP loops, as a technique where idle processors steal a fixed number of iterations from busy processors as in [61] and [50]. The number of iterations stolen are 1/p of the iterations (where p is the number of processors). Although this form of scheduling results in dynamic load balancing (specific to the hardware), the chunk size of stolen iterations is statically specified for the execution of the application.

OpenMP was designed for execution of parallel loops of dense numerical applications, and has been upgraded to support other requirements for parallel programming. One improvement is the implementation of task parallelism in

OpenMP 3.0. During the development of task parallelism for OpenMP 3.0, A. Duran, J. Corbalan and E. Ayguadé [25] evaluated the proposed task strategies using the Nanos experimental OpenMP research runtime. Two scheduling algorithms, breadth-first and work-first, are evaluated with and without cut-off strategies. They use two cut-off policies, one based on a maximum number of tasks allowed in the pool and one based on the maximum recursion depth. They also evaluate using *tied* and *untied* tasks; *tied* tasks are default in OpenMP. When a task is *tied*, if it suspends and is restarted, it will always be executed by the thread that initially executed it. Tied tasks should mitigate performance degradation due to memory sharing, but in the experiments that employ *tied* tasks, performance is impeded. The work-first scheduler performs best. The cut-off techniques improve performance, however it is not clear which policy is better. They conclude that it is beneficial to estimate the granularity of the tasks at execution time and utilize that knowledge to determine a beneficial cut-off depth; they accomplish this in subsequent research and develop Adaptive Tasks Cutoff (ATC).

A. Duran et al. continue their previous work by developing ATC [24], a mechanism that uses the estimation of granularity to tune cut-off at execution time. They implement a dynamic profiler that, in full mode, times the workloads of OpenMP tasks and nano-tasks. A nano-task is an aggregation of OpenMP tasks executed as one user-level task. The profiler causes overheads, so they run it just long enough to collect sufficient statistics to estimate a cut-off. At each level the

subtrees are timed and averaged to compute the grain size. If the grain size is less than a specified time (1 msec in their experiments) the level is closed and all subsequent checks will not require timing. If a level is closed sub tasks are not created and continuation on the same task occurs ensuring a larger grain size. Thus they then tune granularity by implementing the estimated cut-off. Overheads from minimum profiling of just the nano-tasks are small enough that it can be turned on to determine if full profiling should be exercised again to re-tune the cut-off depth. The results are comparable to the best cut-off levels determined by tedious manual methods. Although the technique is implemented for OpenMP task parallelism, it can be adapted to other task parallel frameworks, and illustrates the importance of measuring task granularity during execution. Although this method is adaptive, it uses a static specified time to tune task grain size.

More recent work implementing adaptive grain size for OpenMP loops in parallel applications with irregular workloads is accomplished by M. Durand et al. in [26]. They incorporate an adaptive loop scheduler in the libGOMP runtime library. In applications where the iterations are irregular (i.e. the execution time of iterations are not uniform) eventually some processors complete computations and become idle. The scheduler initially distributes iterations of the loop evenly to the worker threads. When a thread completes its iterations, it steals one half of the remaining iterations from a victim thread.

The number of iterations run on the victim thread decreases and the number

of stolen iterations varies with each steal in an attempt to balance granularity of the workloads. The algorithm is implemented with measures to ensure NUMA locality by attempting steals from the victim threads in the same NUMA domain until a threshold of unsatisfied steals has occurred, then it steals from random victims.

The adaptive loop scheduler performs better than the available OpenMP schedulers (static, dynamic, and guided) for KMeans, a benchmark in the Rodinia suite [17], and a fluid simulation benchmark. Both benchmarks exhibit irregular workloads per iteration of the OpenMP loops. The scheduler mitigates idle times often caused by implicit barriers in OpenMP by modifying the grain size of tasks in the loop. This adaptive scheduler is designed for applications that use OpenMP loops with irregular computations for each iteration, and is portable since it is hardware agnostic.

3.2 Adaptive Schedulers

We also consider adaptive scheduling techniques for parallel applications other than those that tune task granularity.

J. Nakashima, S. Nakatani, and K. Taura [44] implement an API that gives the application programmer the ability to customize the behavior of the scheduler in the Massive Threads library [3]. The Massive Threads library implements light weight user level threads (tasks) for parallel applications. The API allows

the application to add hints to tasks and to guide the scheduler to adapt stealing based on those hints. The hints are added to the bottom of the task stack. Each scheduling queue includes a hint cache. When a task is ready for execution it is placed on a scheduling queue and the hint is added to the hint cache for that queue (hint cache is updated). The hint cache is only locked when it is updated, allowing workers to read the cache without incurring overheads associated with locks. Threads searching queues for available tasks read the hint cache to determine which available tasks are best to steal.

Two types of hints (depth-aware and affinity-aware) are employed to illustrate proof of concept. The depth-aware hint is used by a recursive divide-and- conquer multiplication problem. The hint for each task gives an indication of the levels of computation that will be executed by the task. The hint is an indication of grain size. This scheduler steals tasks with the largest potential granularity. The affinity-aware hint contains information about the relationship of the affinity of the task to the worker so that when a thread looks for work it will steal tasks that have the closest affinity.

Employment of task hints gives flexibility for programmers to use parameters specific to the application, but requires the programmer to be able to understand the scheduler and determine parameters that will help optimize scheduling of parallel tasks. The hints may lead to varying performance on different platforms, possibly resulting in poor performance portability.

In [57], A. Tzannes et al. implement lazy scheduling, a runtime adaptive scheduler designed for applications that employ loop parallelism. Lazy scheduling uses load conditions to effectively coarsen task granularity dynamically. The scheduler features private deques (double-ended queues) for each worker that store task descriptions (TDs). A TD contains the description of a range of tasks scheduled by parallel constructs such as loops or reducers in parallel applications. The worker threads each have a shared deque to make tasks available for other workers to steal. The scheduler checks the size of the shared deque to assess if other workers are busy. If the number of tasks in the deque falls below a threshold, it is an indication that other workers are stealing work and are not busy. The scheduler then splits a TD and pushes tasks onto the shared deque. If the number of tasks in the shared queue is above the threshold, the worker continues to execute local work. Lazy scheduling relies on frequent poling of the shared deque. Although poling incurs some overheads, continuation of local work when other workers are busy effectively coarsens grain size and limits expensive deque transactions.

Three lazy scheduling policies implemented using Intel's Thread Building Blocks (TBB) are:

1. DF-LS, depth-first lazy scheduling with a threshold of one
2. DF2-LS, depth-first lazy scheduling with a threshold of two
3. BF-LS, breadth-first lazy scheduling

For the DF-LS policy, if the shared queue is empty the scheduler splits the

current TD and pushes tasks onto the shared queue. For BF-LS it pushes the oldest postponed task (with the shallowest nesting depth) onto the shared deque first. BF-LS performed best for larger number of cores for a variety of benchmarks. Lazy scheduling is a method that adapts scheduling at runtime, does not require tedious manual tuning by the programmer, and has the potential for portable performance for various hardware platforms.

The authors also demonstrate the feasibility of using lazy scheduling techniques for non-loop parallel applications. They implement a custom scheduler, Lazy Splitting, for the Unbalanced Search Tree (UTS) benchmark by modifying the number of tasks to move from the private to shared queue, when the shared queue is empty and a private queue has at least two tasks. In this manner when another queue has no tasks in its private queue and there are none remaining in the shared queue the number of tasks remaining in another private queue are recursively split to share. This gives a means of approximating lazy scheduling as described previously for parallel loop applications. The three lazy scheduling policies are then incorporated into a TBB implementation of UTS. Speedup results from the BF-LS scheduler are very good and from the Lazy Splitting scheduler, near perfect.

U. Acar, A. Charguéraud, and M. Rainey [8] implement work stealing using private deques for storage of local work and explicit communication to determine availability of work and to send work to idle processors. In addition, busy workers

have to poll their deques regularly, update the system with information about the availability of work in their deques, and push work to requesting idle workers. The results for applications that use fork-join algorithms are comparable to the more typical eager work stealing methods, where work is split and not dependent on polling the load. An experiment using the technique on a benchmark employing a pseudo-depth-first-search algorithm showed promising scaling results for irregular graph algorithms. One disadvantage of this method is that if busy workers are executing tasks with long durations, idle workers will continue to search for work without success, even though possible work is on the busy workers deque.

Other work of interest includes adaptive thread scheduling based on introspection of hardware behavior as in the work done by A. Porterfield et al. in [47]. They use RCRdaemon to collect hardware memory and power counters, then based on the hardware performance, throttle the number of threads used for memory bound applications. Their results indicate that this type of adaptive scheduling can improve performance and save energy, but requires the ability to perform decisions and throttling actions at the hardware level to be effective in restarting threads.

In other relevant work S. Olivier et al. [46] characterize overheads for task-based parallel programs and augmented the Qthreads [6, 60] library with locality-aware scheduling. The locality aware scheduler uses a shared queue with a LIFO policy for each NUMA domain. Performance of the locality-aware scheduler is a great improvement over other tested schedulers. The characterization of the work

time inflation illustrates benefits from the use of the locality aware scheduler. Our *wait time*, 5.1.2 metric is a measure of work time inflation, and we characterize *wait time* as it relates to varying task granularity in Sections 5.2.3 and 5.2.4.

4 DISSERTATION CONTRIBUTION

HPX [37] implements asynchronous task scheduling using *future* and *dataflow* constructs (Section 2.2) eliminating implicit barriers associated with execution models such as OpenMP [20, 15] and Cilk Plus [41]. This facilitates fine-grained parallelism for increased performance by allowing worker threads to perform other useful work when tasks self-suspend while waiting for resources or data. Although implementations of asynchronous task parallelism can improve performance by keeping the processors busy with useful work, additional overheads for management of fine-grained tasks can degrade the gain in performance. In addition, parallelization of massive quantities of fine-grained tasks can cause contention on resources and inflate task execution time (grain size) as shown in results in Chapter 5. Dynamic measurement of task granularity, scheduling overheads, and performance metrics, with subsequent adaptive thread scheduling can minimize overheads and improve performance for parallel applications. Currently, the user usually needs to manually experiment and adjust parameters that affect task grain size of the application to improve performance. Such an effort can be time consuming, inefficient, and inaccurate. The procedure has to be repeated if the computer system undergoes modifications or if the application is ported to another system. This work proposes the incorporation of adaptive measures to eliminate the manual process of task granularity optimization by providing a feedback system

where measurements are taken of intrinsic events and dynamic changes are made thus improving programmability and portability while minimizing overheads and improving performance and scalability. The formal hypothesis of this research is:

> *H: Adaptive task-based parallel runtime systems can measure intrinsic events to assess scheduling overheads, resource usage, and performance efficiency of the parallel application on the underlying hardware then through a feedback loop dynamically tune task grain size or adapt thread schedulers for improved performance.*

In order to accomplish dynamic adaptation of the thread scheduler or tune task size, monitoring the system during execution is essential. While current performance monitoring tools are designed for monitoring synchronous execution, they are not always useful for improving codes that implement asynchronous tasks. Performance tools such as HPCToolkit [9] and TAU [49] provide profiling or tracing of application codes through instrumentation or periodic sampling and are useful for post run analysis. They do not support task-based parallel runtime systems because they are not designed to deal with millions of short-lived threads. Results of experiments attempting to use these tools are documented in Chapter 6 and show that they are inadequate for the purpose of monitoring task-based parallel applications during execution time. Also in Chapter 6 the results of the experiments performed with a variety of benchmarks illustrate the usefulness of performance monitoring by the runtime in order to measure intrinsic performance counters necessary to steer dynamic adaptation. An example of dynamically tuning grain

size based on information gathered from the HPX performance monitoring system is presented in Chapter 7.

This dissertation specifically makes the following contributions:

1. Aided in the development and evolution of HPX through dicsovering issues and bugs in the thread scheduling and performance monitoring systems of HPX.

2. Characterization of the influence of task grain size on performance of task-based parallel applications in the HPX runtime system; determination of the metrics and their correlation to the performance and overheads of applications.

3. The implementation of new performance counters to measure average task and phase duration and task scheduling overheads, giving the application a means of determining the effective granularity during execution. Implementation of counters to measure cumulative scheduling overheads and task execution time so that measurements of factors contributing to degradation of performance can be made over any interval of execution.

4. Application of the methodologies to measure grain size and overheads to dynamically tune a parallel benchmark for optimal performance. We illustrate dynamic adaptation using the integration of HPX with APEX to use performance information from the runtime system by the execution policies in

APEX through a dynamic feedback system to tune grain size for improved performance.

The following chapters present the experiments used to achieve the contributions.

5 PERFORMANCE IMPLICATION OF TASK GRANULARITY

Steps toward implementing adaptivity in task-based runtime systems include describing the relationship between scheduling overheads and task granularity, understanding how schedulers affect performance for different task sizes, and ascertaining metrics that dynamically determine effects of parallelization on task granularity and scheduling overheads. In order to fully understand the effects of task granularity on parallel applications we run experiments to assess overheads and performance while varying task size. In this chapter, we expand our findings presented in [30].

5.1 Task Granularity Experimental Methodology

Our goal is to explore how to dynamically tune task granularity in a programming model that uses fine-grained asynchronous task scheduling mechanisms. In parallel applications, with regular parallel loops, task grain size can be modified statically to improve performance. We need to be able to determine granularity and adjust it at execution. To this end, we use HPX-Stencil, described in detail in Section 5.1.1, with its controllable partition size and asynchronous data-flow constructs. Varying task grain size from fine-grain to coarse-grain will result in different overheads. Executing applications with millions of fine-grained tasks can cause overheads for thread management and overheads associated with con-

tention for queuing and memory resources. We determine metrics that measure performance behavior, determine granularity and associated overheads then use the facilities in HPX to read required event counts and derive the metrics. This characterization is a first step toward tuning grain size for performance improvement.

The experiments for this study comprise executing the HPX parallel benchmark, HPX-Stencil, described in Section 5.1.1, over a large range of partition sizes, to vary granularity, and for an increasing number of cores for strong scaling performance. When collecting performance and counter data, we make ten runs and calculate the mean, standard deviation and 90% confidence intervals of the counts. We compute the metrics using the average of the required event counts.

5.1.1 Stencil 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. This benchmark was chosen because task granularity can be easily controlled, allowing us to use task size as the basis of our experiments and enabling us to construct a simple test case in runtime adaptivity. In HPX-stencil, the grid points are divided into partitions; each partition contains a fixed number of grid points specified by the user. The regular updates for the grid points in each partition is computed as one task.

Specifying the size of the partition provides us with the means to control grain size for our experiments. The results from this study with the stencil benchmark allow us to implement new counters that we are able to then use for experiments with a variety of benchmarks (see Chapter 6). The calculation simulates the

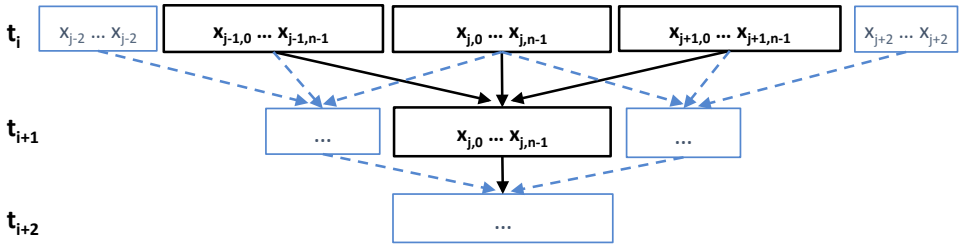

Figure 8: Dependencies of Heat Distribution Benchmark

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.

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 for the next time step. This dependency is captured in Fig. 8, 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 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 additional HPX facilities. These compositional facilities give the ability to create 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) [37]. This efficiency combined with the semantics of *futures* allow the direct expression of the generated dependency graph as an execution tree generated during execution, 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 [55]. 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 control the grain size of the problem and vary it from ∼3 µs to ∼30 ms for the Xeon E5 nodes and from ∼100 µs to ∼400 ms for the Xeon Phi coprocessor.

5.1.2 Performance Metrics

We compute and analyze numerous metrics, and 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 time spent on the computation portion of each HPX thread (task), and $\sum t_{\text{func}}$, the running sum of total times to complete each task. The time

to complete execution of tasks includes task management overhead and we compute the overhead by subtracting the computation time of tasks. HPX computes the idle-rate (I_{r}) as shown in Eq. 1.

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

In Section 5.2.1 we show the affect of task granularity on idle-rate and the correlation with execution time.

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 is computed as shown in Eq. 2. The number of HPX threads executed, n_{t}, is also available as counter */threads/count/cumulative*.

$$t_{\mathrm{d}} = \frac{\sum t_{\mathrm{exec}}}{n_{\mathrm{t}}} \tag{2}$$

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.

$$t_{\mathrm{o}} = \frac{\sum t_{\mathrm{func}} - \sum t_{\mathrm{exec}}}{n_{\mathrm{t}}} \tag{3}$$

Task duration and overhead performance counters were added to HPX as a part of this study and are now available for dynamic measurement. 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 as 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 effects 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 for the entire run 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.

$$T_{\mathrm{o}} = \frac{\sum t_{\mathrm{func}} - \sum t_{\mathrm{exec}}}{n_{\mathrm{c}}} \tag{4}$$

Wait Time

When running on multiple cores the duration of a task can increase due to overheads caused by parallelization on the underlying hardware. We compute the average *wait time* per thread (t_{w}), Eq. 5, as the difference between the average task duration (t_{d}) and task duration for the same experiment run on one core (t_{d1}). Since *wait time* is a direct function of the task duration, a measurement only of the computation time of the task, it does not include task management overhead. This additional time is due to overheads caused by cache misses, non-uniform memory latencies, memory interconnect, cache coherency and/or memory bandwidth saturation.

$$t_{\mathrm{w}} = t_{\mathrm{d}} - t_{\mathrm{d1}} \tag{5}$$

We use Eq. 6 to compute *wait time* per core (T_{w}). This is effectively the total *wait time* of all tasks divided by the number of cores for comparison to execution time. 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.

$$T_{\mathrm{w}} = \frac{(t_{\mathrm{d}} - t_{\mathrm{d1}}) * n_{\mathrm{t}}}{n_{\mathrm{c}}} \tag{6}$$

Pending Queue Accesses

The HPX counter, pending queue accesses, (*/threads/count/pending-accesses*) counts the number of times the thread scheduler looks for available HPX threads in the worker thread's associated pending queue. This count is available for each worker thread or as the total from all worker threads. For this study we use the total count. The counter registers the activity by the HPX thread scheduler on the pending queues. The pending queue misses, the number of times the scheduler fails to find available HPX threads is also available as */threads/count/pending-misses*.

5.1.3 Experimental Platforms

HPX is a runtime system designed for parallel computation on either a single node or in distributed mode on homogeneous or heterogeneous clusters. This study is only concerned with performance useful for determining task granularity and tuning task granularity. Therefore, the experiments measure performance on a single node. HPX employs a unified API for both parallel and distributed applications, thus the findings of this study can be applied to the distributed case.

Experiments are run on an Intel®Xeon®Phi™coprocessor and three Intel®E5 nodes of the Hermione cluster, Center for Computation and Technology, Louisiana State University, running Debian GNU/Linux Unstable, kernel version 3.8.13 (on the Xeon Phi, 2.6.38.8 k1om), using HPX V0.9.10. The specifications of the

platforms are shown in Table 1. The three Xeon E5 nodes have hyper-threading disabled, so the computations are done using one thread per core. For the Xeon Phi we run experiments for 1 to 4 threads per core. The insights from the results using multiple threads per core are not different than for those when executing with one thread per core, so only results from running with one thread per core are presented for the Xeon Phi.

Table 1: Platform Specifications for Task Granularity Experiments

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) 256 KB L2	32 KB L1(D,I) 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) 256 KB L2	32 KB L1(D,I) 256 KB L2
Shared Cache	35 MB	20 MB
RAM	128 GB	64 GB

5.2 Task Granularity Experimental Results

Using the benchmark HPX-stencil, we run experiments over a large range of task grain sizes by varying the partition size from 160 to 100,000,000 grid points,

resulting in task grain sizes ranging from (∼3 µs to ∼30 ms) for the Xeon E5 nodes and from 1000 to 100,000,000 grid points resulting in grain sizes from (∼100 µs to ∼400 ms) for the Xeon Phi. The number of partitions is adjusted to keep the total number of grid points constant at 100,000,000. For each experiment, the heat diffusion is computed for 50 time steps on the Xeon E5 nodes. For experiments on the Xeon Phi five time steps are computed, since a full sweep of experiments (ie. varying partition size and core counts for 10 samples) took 20 hours, compared to 12 hours for 50 time steps on the Haswell node. Although, the clock speed of the Haswell processors are only twice the speed of the Xeon Phi processors, task computations take 50 times longer on the Xeon Phi. This can be explained by the differences in the micro-architecture of the two. The Xeon Phi coprocessor has in-order execution cores, each with two execution pipelines [2]. The Haswell architecture has out-of-order cores, a more complex design than the Xeon Phi, that can dispatch eight instructions per clock cycle [1], resulting in execution times an order of magnitude faster on the Haswell.

As the number of cores used for each experiment is increased the total number of grid points is kept 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 90% confidence intervals for execution times and counts. For idle-rate we compute the harmonic means of the samples. We also compute correlation coefficients between the metrics and execution time over six ranges of task granularity

using each sample.

Performance monitoring can cause perturbation of execution time. To assess any additional overheads, we run the application with and without building the timing counters in the runtime system. We perform extensive experiments assessing overheads caused by invoking the timers used for the idle-rate, task duration, and overhead timing counters. The overheads are less than the standard error except for some cases where the experiments are run on only one core and task durations are less than 4 µs. Tasks that have a duration of 4 µs or less have greater than 19% overhead since the per task overheads are 760 ns [37]. Tasks with such short durations incur overheads that are relatively large compared to the function and should be coarsened if possible.

We examine the performance of the HPX-stencil benchmark, in Figures 9 - 12, showing how execution time is affected by task granularity. On all platforms, execution time is large for very fine-grained tasks due to overheads caused by task management and for very coarse-grained tasks where overheads are caused by poor load balance. In between these two task-size ranges, we expect execution time to remain constant since task management overheads are small. However, *wait time* as explained in Section 5.1.2 also influences the execution time and is dependent on task granularity, the number of cores used, and the underlying architecture.

Using the metrics defined in Section 5.1.2 we measure the effects of varying task granularity and the number of cores used on overheads and performance.

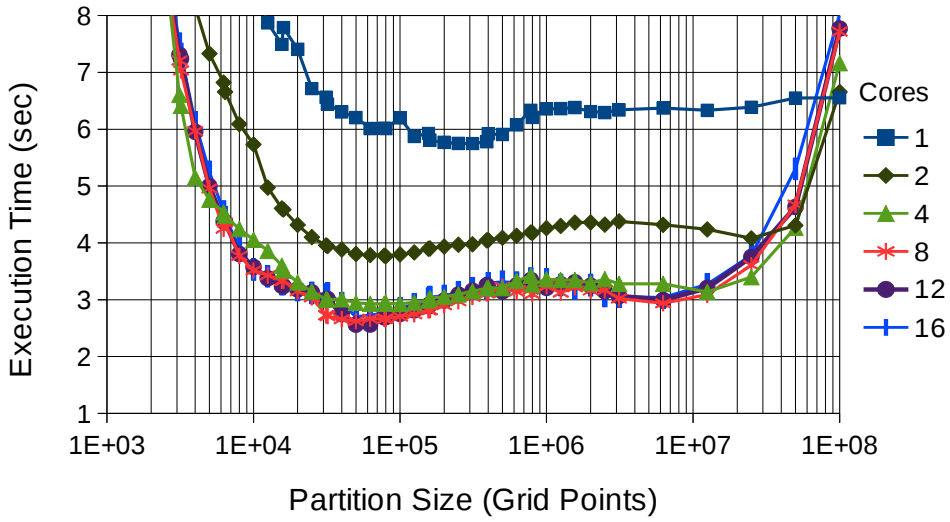

Figure 9: Sandy Bridge: Execution Time vs. Task Granularity (partition size)

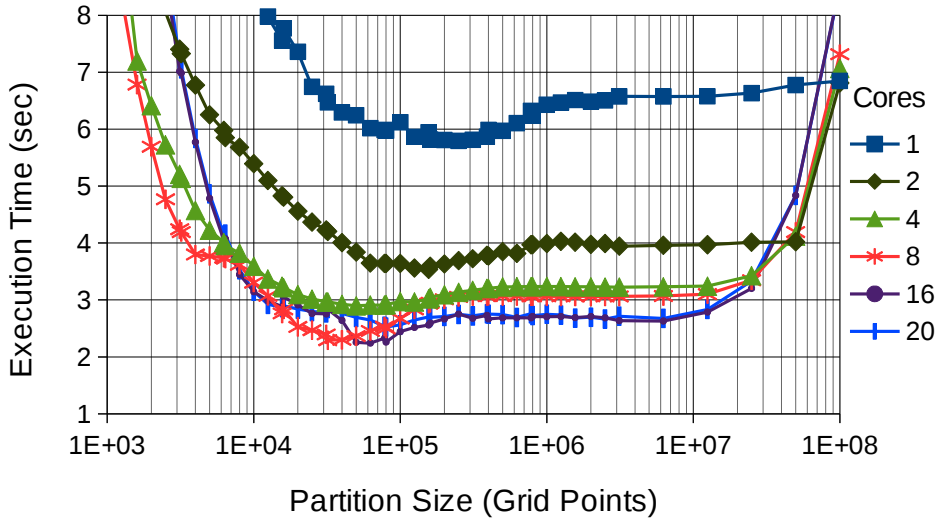

Figure 10: Ivy Bridge: Execution Time vs. Task Granularity (partition size)

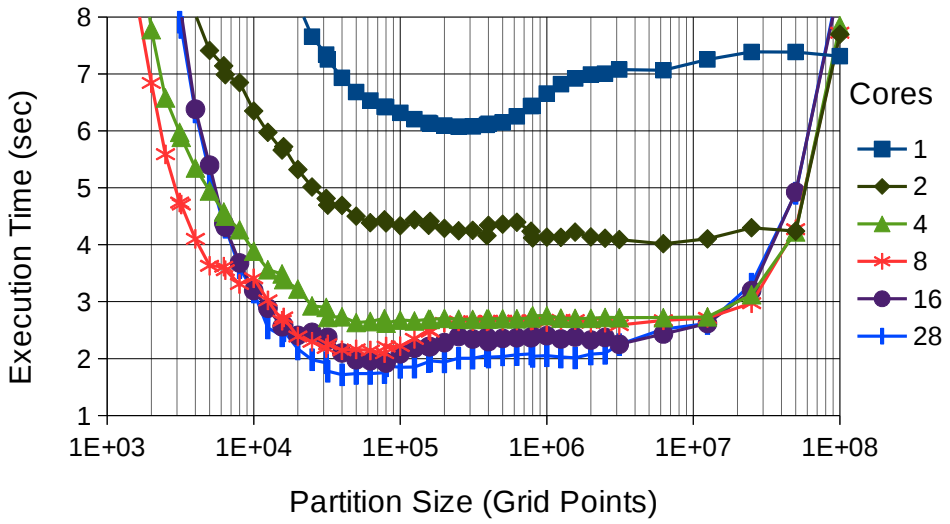

Figure 11: Haswell: Execution Time vs. Task Granularity (partition size)

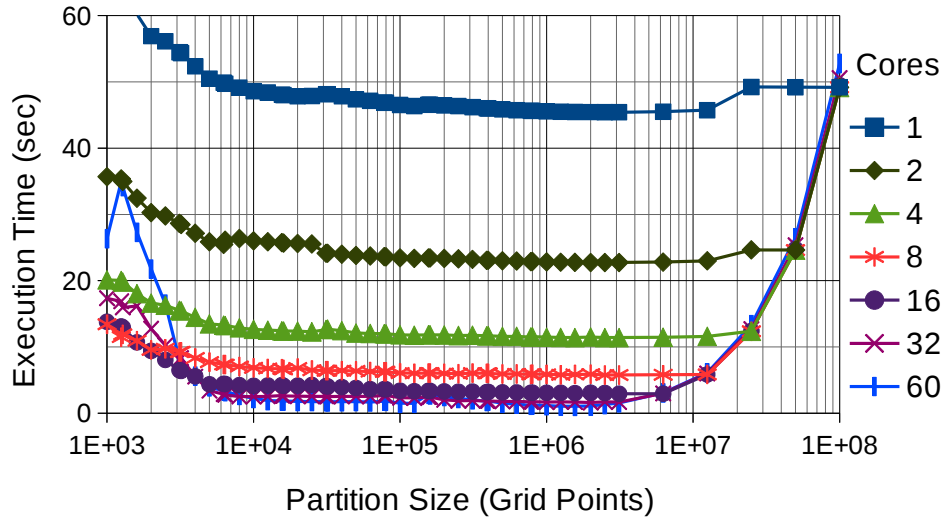

Figure 12: Xeon Phi: Execution Time vs. Task Granularity (partition size)

We present experimental measurements and resulting data for the Haswell and Xeon Phi platforms. Results of the metrics for the other platforms are similar to the results from the Haswell node and are in Appendix A.1 (Sandy Bridge) and Appendix A.2 (Ivy Bridge).

To determine the correlation between the metrics and execution time for the Haswell node, we divide the data into six ranges of task granularity and calculate the correlation between each metric and execution time using all samples for each range. The correlations are listed in Table 2. In the following sections, we examine the results for the metrics and their associated correlations.

5.2.1 Idle-rate

Idle-rate is the ratio of time spent on HPX thread management to that of execution, shown in Figures 13 - 16 for the Haswell node and Figures 17 - 20 for the Xeon Phi. We only present the results from these two platforms in this section since the results from the Haswell nodes are very similar to the Sandy Bridge and Ivy Bridge nodes. The results from the other nodes are in Appendix A.1 (Sandy Bridge), Figures 70 - 73 and Appendix A.2 (Ivy Bridge) Figures 86 - 89.

For small partition sizes, there are a large number of very fine-grained tasks to manage, and task management is a large percentage, up to 90%, of the execution time. The region on the left side shows the large idle-rate and execution times for very fine-grained tasks with partition sizes less than 12,500 grid points. The

Table 2: Correlation of Metrics to Execution Time - Haswell

For a variety of task granularity ranges.

Granularity	**TM**	**WT**	**WT+TM**	**Idle-rate**	**P-Access**
28 Cores:					
Very Fine	0.999476	0.954147	0.999515	0.348061	0.999421
Fine	0.911898	0.657842	0.995333	0.746256	0.812874
Lower Medium	0.886485	0.644466	0.919982	0.836826	0.259002
Upper Medium	0.809178	0.692299	0.925372	0.609874	0.491171
Coarse	0.838559	0.845881	0.984155	0.476614	0.860650
Very Coarse	0.974325	-0.815015	0.977232	0.718515	0.992850
16 Cores:					
Very Fine	0.999901	0.967321	0.999984	0.408203	0.999771
Fine	0.456427	0.938973	0.991474	-0.465968	-0.270003
Lower Medium	0.340618	0.594374	0.995548	0.070707	0.033850
Upper Medium	0.882440	0.497341	0.994201	0.807701	0.301322
Coarse	0.710203	0.524331	0.978720	0.554692	0.289311
Very Coarse	0.977919	-0.792151	0.980274	0.759265	0.953443
8 Cores:					
Very Fine	0.999785	0.912314	0.999883	0.610751	0.999101
Fine	0.708940	0.636628	0.890115	0.122512	0.249017
Lower Medium	0.422941	0.882626	0.968495	-0.081097	-0.542913
Upper Medium	0.541566	0.478580	0.845531	0.402554	-0.433614
Coarse	0.982384	0.065747	0.691708	0.973899	0.142434
Very Coarse	0.984972	-0.785002	0.956071	0.831534	0.969126
4 Cores:					
Very Fine	0.999728	0.869302	0.999146	0.766364	0.997811
Fine	0.830655	0.519580	0.735368	0.455680	0.577314
Lower Medium	0.841350	0.830339	0.976786	0.742679	-0.072411
Upper Medium	0.911439	0.957041	0.988756	0.852246	-0.563450
Coarse	0.929920	0.659185	0.839686	0.890149	-0.453998
Very Coarse	0.997503	-0.811933	0.942513	0.934292	0.989301

Granularity	**Range of Points per Partition**	**Grain Size (µs)**
Very Fine	160 to 15625	3 to 26
Fine	16000 to 80000	29 to 101
Lower Medium	100000 to 160000	125 to 194
Upper Medium	200000 to 400000	241 to 481
Coarse	500000 to 800000	601 to 998
Very Coarse	> 800000	1283 to 33072

TM - Thread Management Overhead per Core, WT - Wait Time per Core
P-Access - Pending Queue Accesses

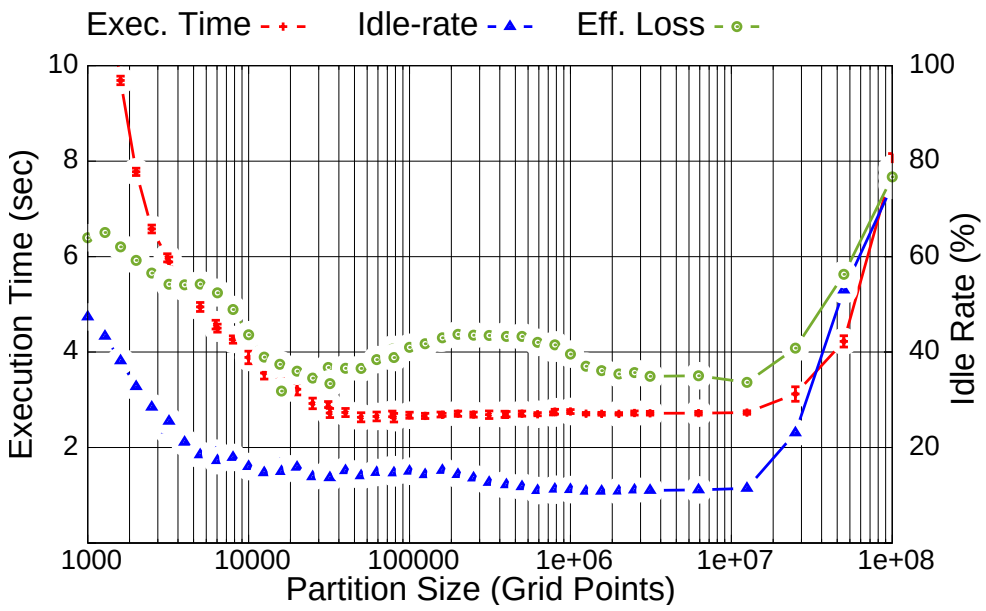

Figure 13: Haswell (4 cores): Idle-Rate

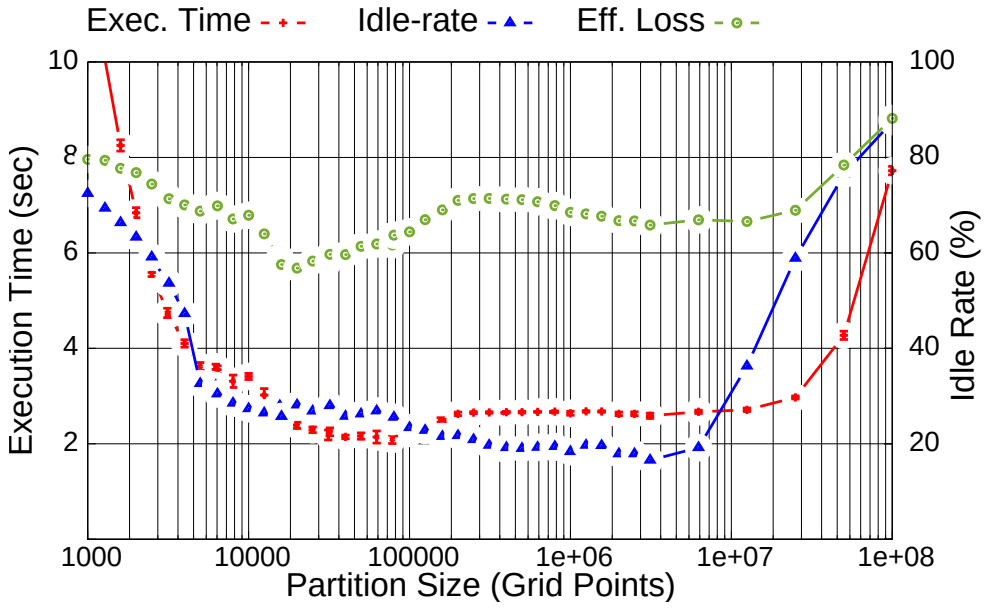

Figure 14: Haswell (8 cores): Idle-Rate

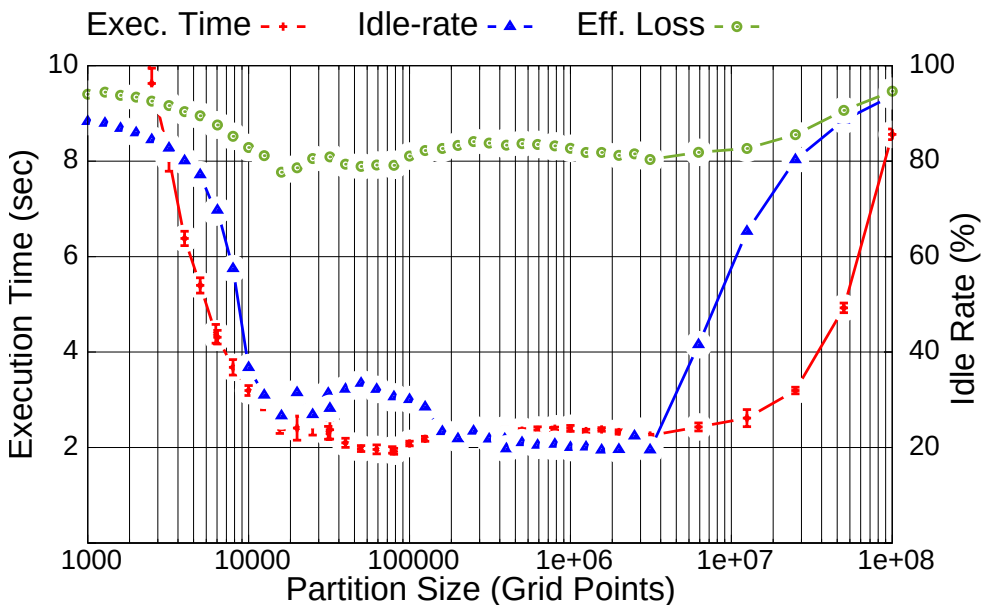

Figure 15: Haswell (16 cores): Idle-Rate

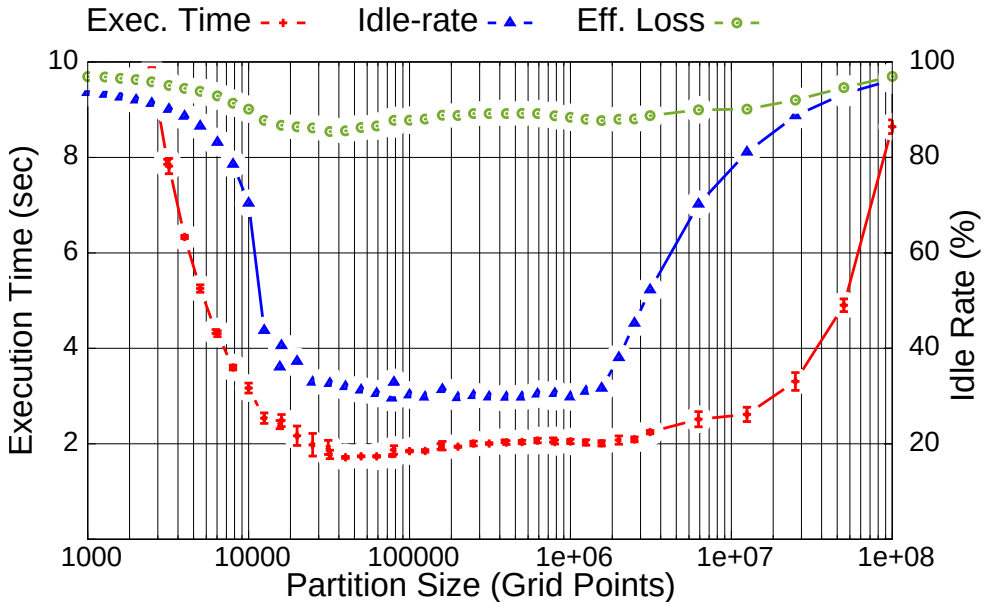

Figure 16: Haswell (28 cores): Idle-Rate

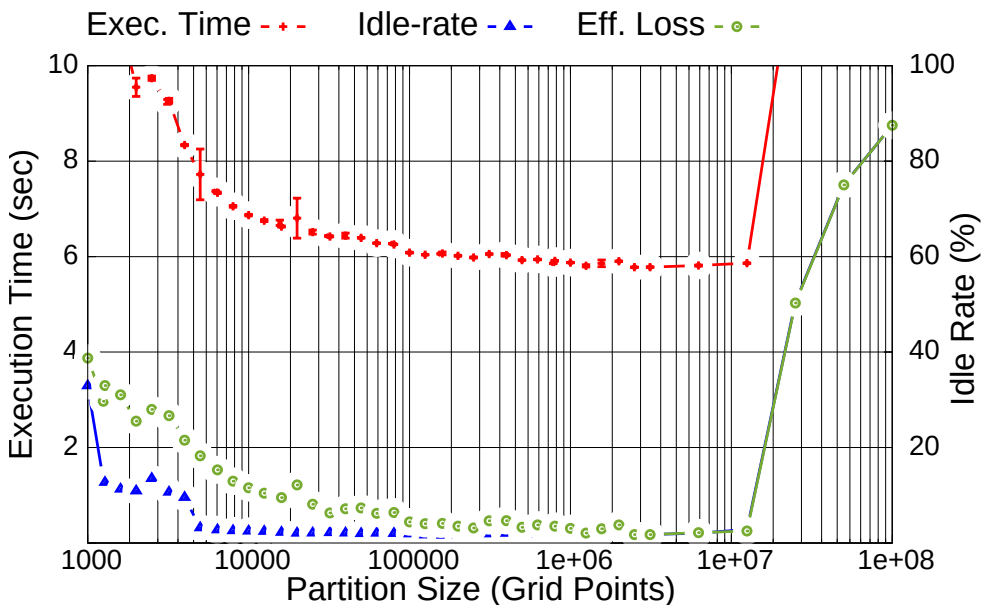

Figure 17: Xeon Phi (8 cores): Idle-Rate

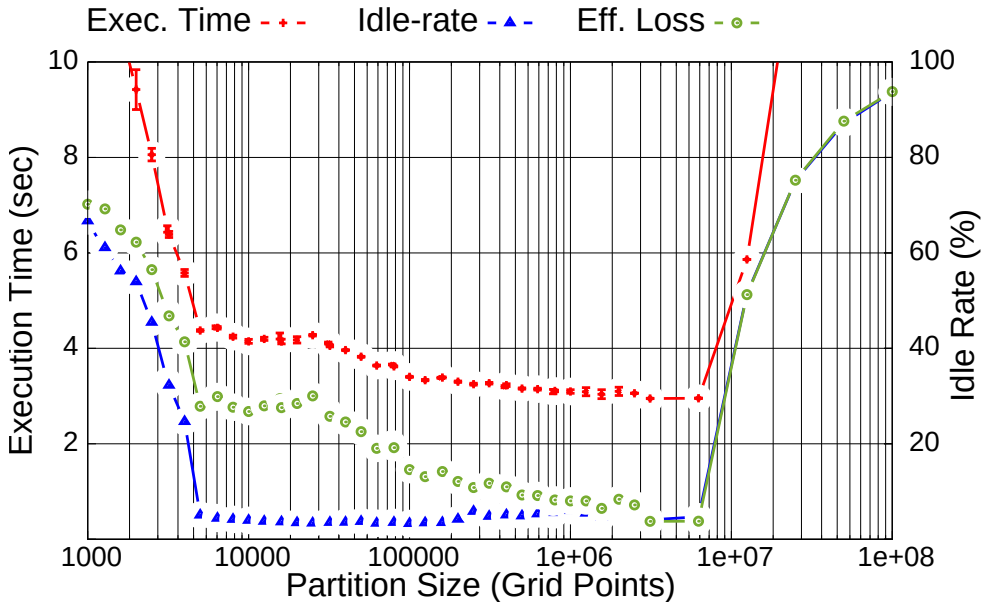

Figure 18: Xeon Phi (16 cores): Idle-Rate

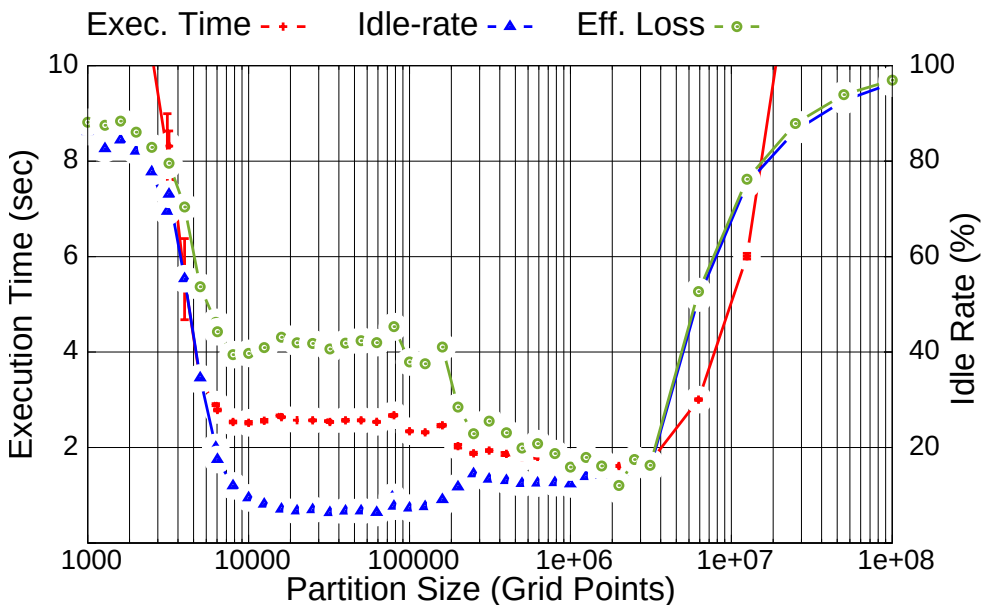

Figure 19: Xeon Phi (32 cores): Idle-Rate

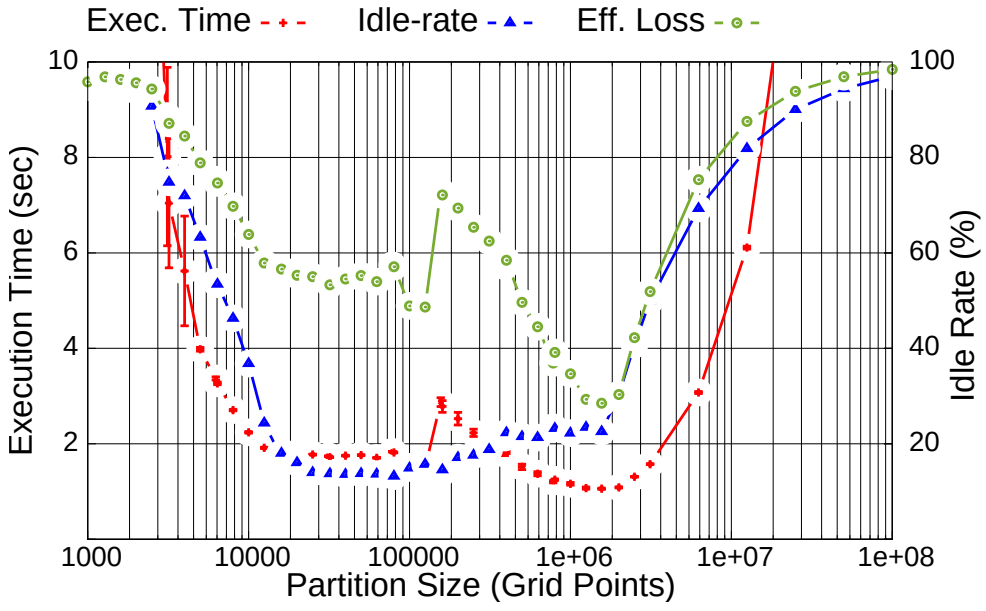

Figure 20: Xeon Phi (60 cores): Idle-Rate

average task duration for computing 12,500 grid points using one core is 21 µs on Haswell and 1.1 ms on the Xeon Phi. Although, the clock speed of the Haswell processors are only twice the speed of the Xeon Phi processors, task computations take 50 times longer on the latter. This can be explained by the differences in the micro-architecture of the two. The Xeon Phi coprocessor has in-order execution cores, each with two execution pipelines [2]. The Haswell architecture has out-of-order cores, a more complex design than the Xeon Phi, that can dispatch eight instructions per clock cycle [1].

For large partition sizes, idle-rate increases due to starvation. The tasks are large and concurrency is difficult to maintain, resulting in poor load balance where cores are left with no meaningful work, but the thread scheduler continues to look for work.

For strong scaling the workload is kept constant as parallel resources (i.e. cores) are increased. Ideal strong scaling is achieved when speedup, Eq. 7, is the same factor as the increase in the number of cores, resulting in 100% efficiency, Eq. 8. However, overheads caused by parallelization limit scaling, resulting in loss of efficiency, Eq. 9. Both loss of efficiency and idle-rate increase as the number of cores are increased, limiting scaling. However, idle-rate does not account for all of loss of efficiency.

$$
\begin{aligned}
SU &= \frac{T_1}{T_n} \text{ where} \\
T_1 &= \text{ execution time on 1 core} \\
T_n &= \text{ execution time on n cores}
\end{aligned}
\tag{7}
$$

$$
\textit{Efficiency} = \frac{SU}{n} * 100\% \tag{8}
$$

$$
\textit{Efficiency Loss} = 100\% - \textit{Efficiency} \tag{9}
$$

Idle-rate can be used to determine a range of task granularity that will not impose large scheduling overheads, but since the only overhead it considers is task management, it cannot be used alone to determine optimal grain size. Pronounced examples of this can be seen in Figures 14 and 15 for the Haswell node for 8 and 16 cores, for partition sizes from 20,000 to 100,000 (32 to 125µs), even though idle-rate increases, execution time decreases. Another clear example is shown in Figure 20 for 60 cores on the Xeon Phi. There is a jump in execution time when the partition size is greater than 125,000 grid points (∼11 ms). The next partition size, 156,250 grid points, uses more than 1 MB of memory for each partition, stressing the memory system for each task. Idle-rate does not reflect this drastic change since there is no additional task management overhead. By visual inspection, we can see that idle-rate does not behave the same as execution time. Correlations of idle-rate with execution time for each task granularity range on the Haswell node

are listed in Table 2 and vary widely across both task granularity and core counts. The values are less than 0.5 in most cases, confirming our visual inspection that this metric alone does not correlate well to execution time for fine- to coarse-grained task ranges and is not sufficient to determine optimal grain size.

5.2.2 HPX Thread Management Overhead

Thread management overhead per core for HPX is computed in Eq. 10 and compared to execution time in Figures 21 through 28.

$$T_{\mathrm{o}} = \frac{\sum t_{\mathrm{func}} - \sum t_{\mathrm{exec}}}{n_{\mathrm{c}}} \tag{10}$$

The overhead is high for very fine- and very coarse-grained tasks, and the behavior is similar to execution time in those regions. The correlation of task management overhead per core with execution time for the Haswell node is nearly perfect (~ 1.0) for the very fine-grained and very coarse-grained ranges, Table 2. However, in the center region the behavior of the HPX thread management overhead is relatively constant, but execution time is not. Although as the number of cores are increased, task management also increases but does not appear to have much effect on execution time, for fine- to coarse-grained tasks. Correlations of task management overhead with execution time vary from $\sim$.3 to $\sim$.9 for fine- to coarse-grained tasks.

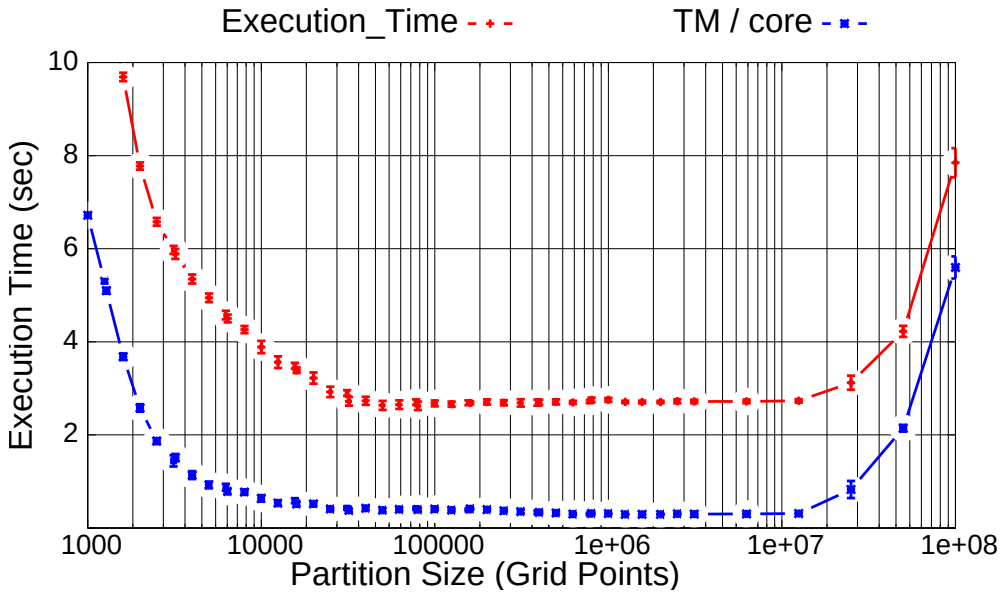

Figure 21: Haswell (4 cores): Thread Management per Core (TM/core)

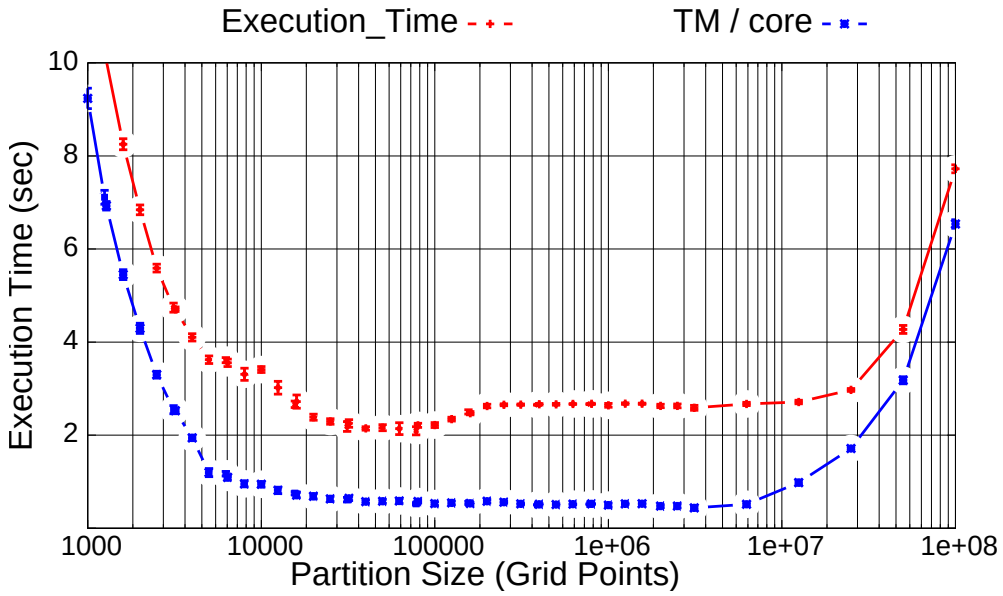

Figure 22: Haswell (8 cores): Thread Management per Core (TM/core)

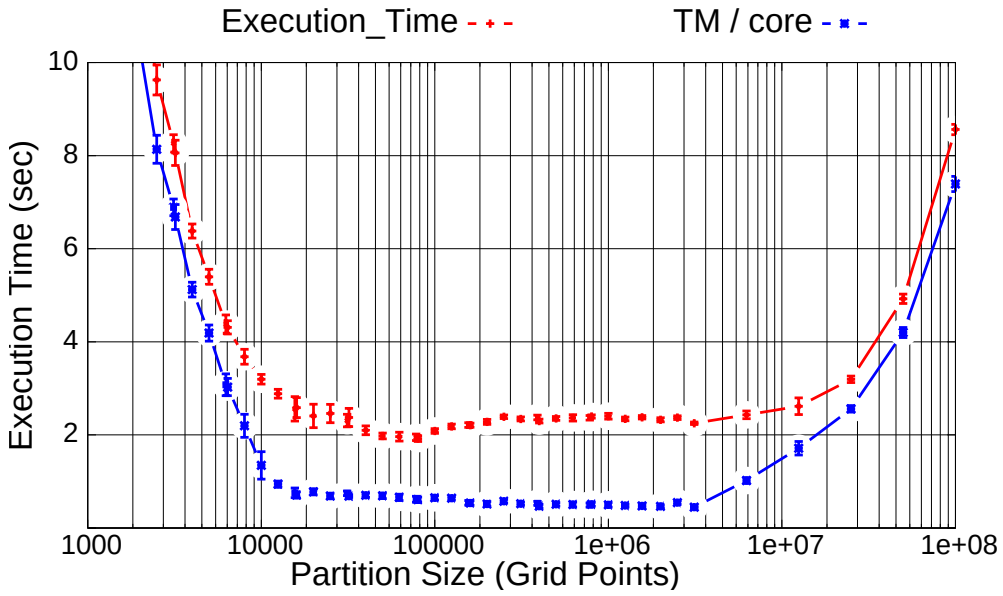

Figure 23: Haswell (16 cores): Thread Management per Core (TM/core)

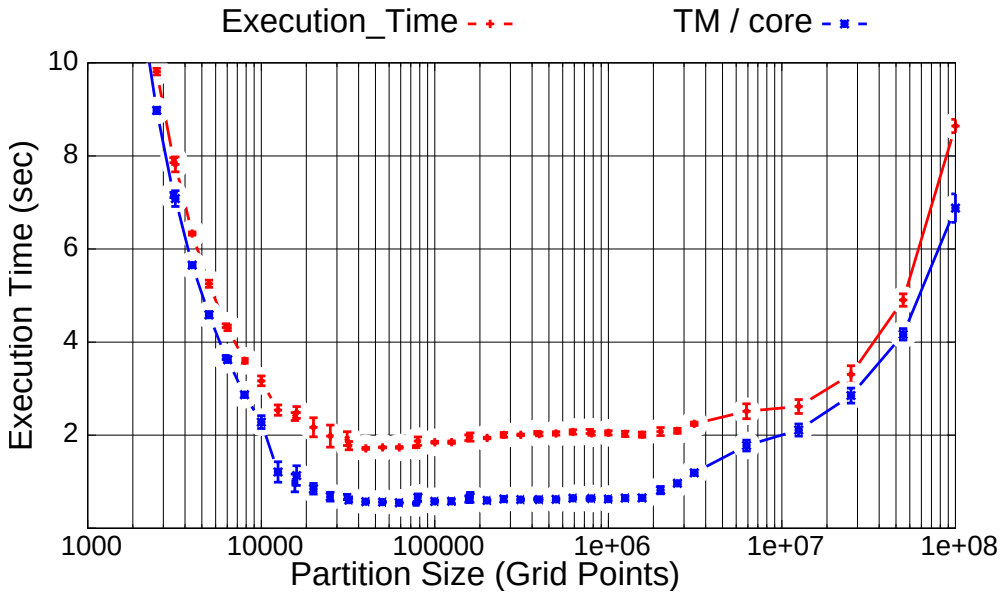

Figure 24: Haswell (28 cores): Thread Management per Core (TM/core)

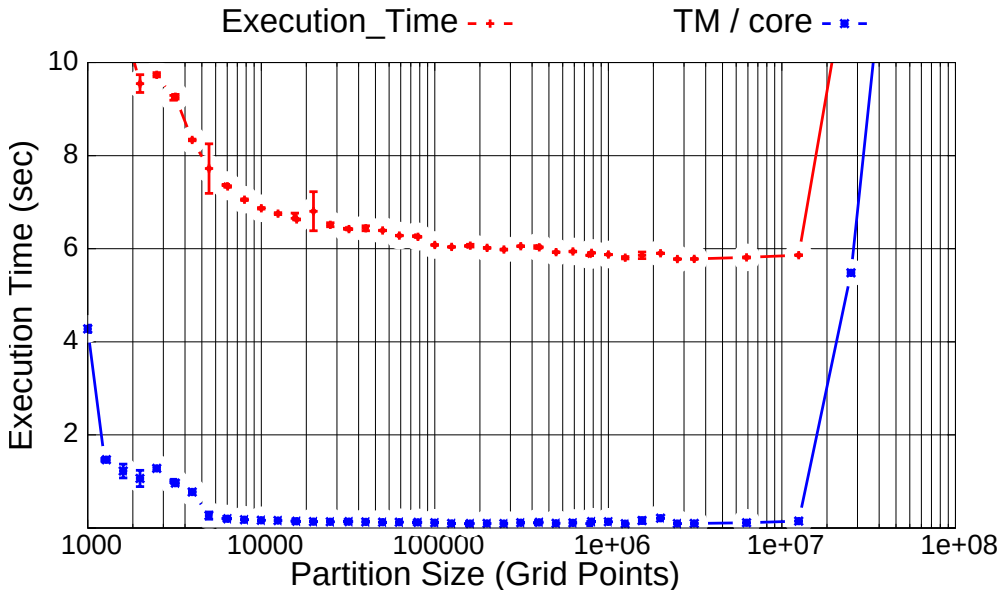

Figure 25: Xeon Phi (8 cores): Thread Management per Core (TM/core)

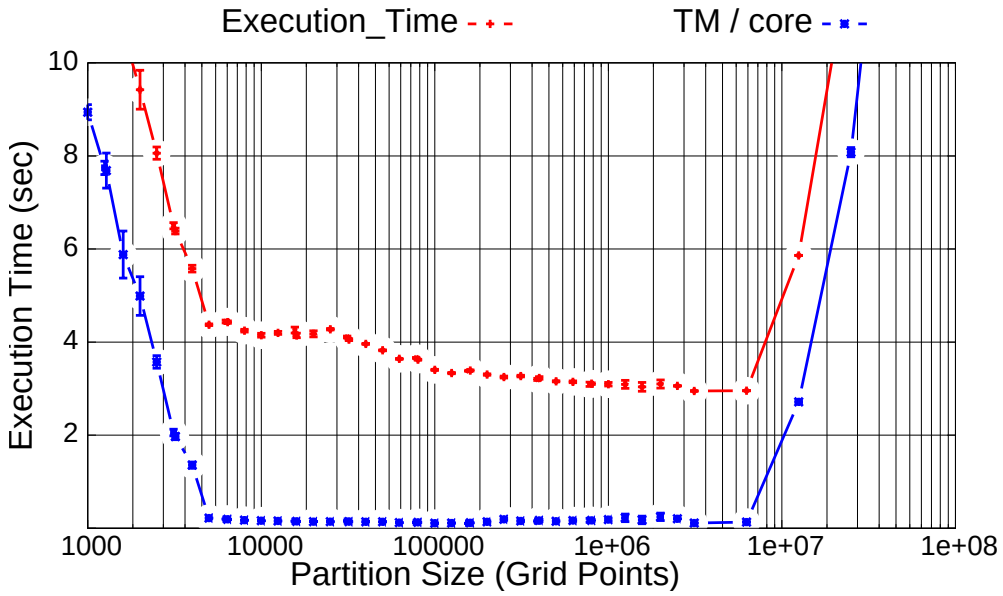

Figure 26: Xeon Phi (16 cores): Thread Management per Core (TM/core)

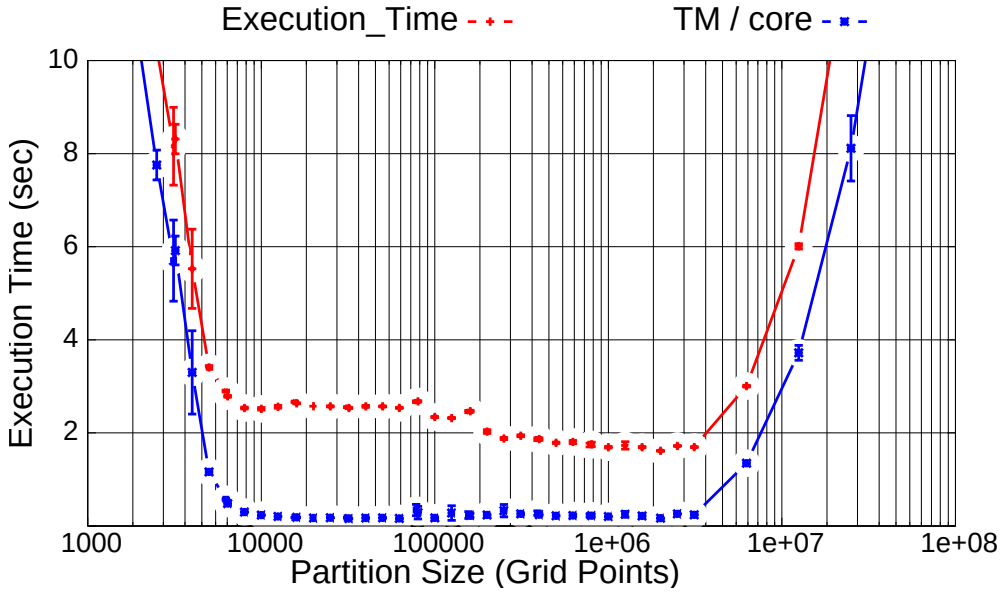

Figure 27: Xeon Phi (32 cores): Thread Management per Core (TM/core)

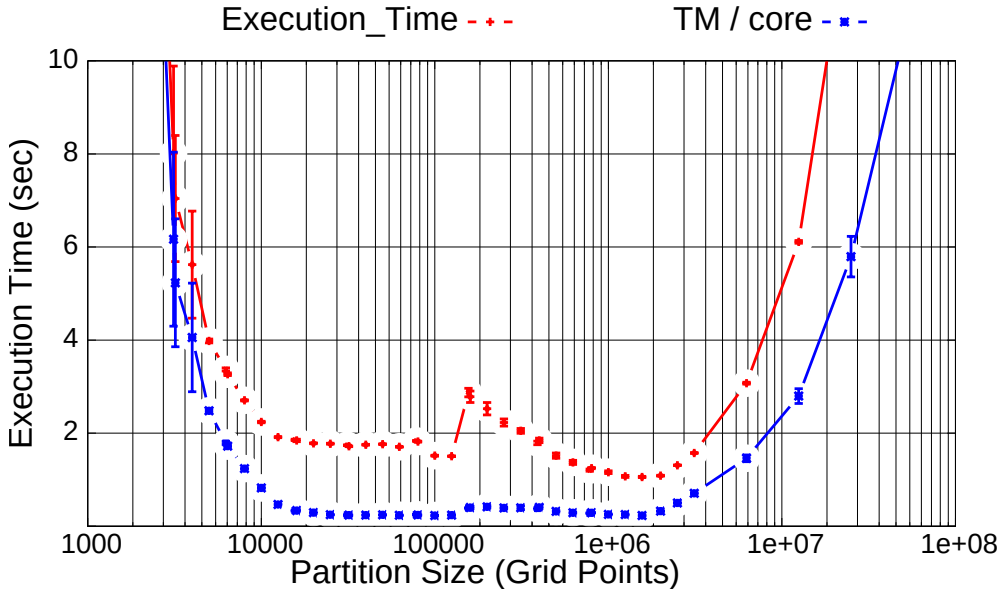

Figure 28: Xeon Phi (60 cores): Thread Management per Core (TM/core)

5.2.3 Wait Time

When the benchmark is executed on multiple cores the duration of a task increases due to overheads caused by parallelization on the underlying hardware. The additional time is due to overheads caused by cache misses, non-uniform memory latencies, memory interconnect, cache coherency and/or memory bandwidth saturation. The average *wait time* per HPX thread, is computed as the

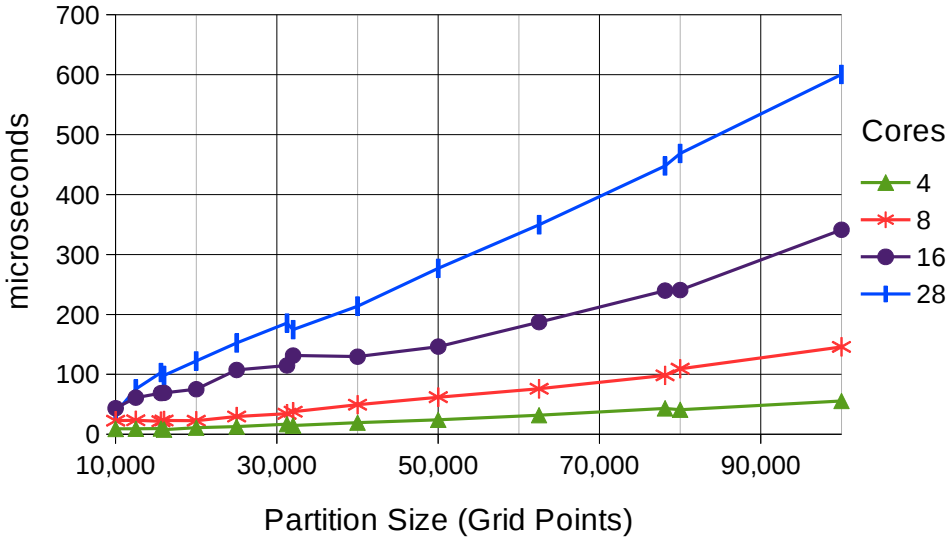

Figure 29: Wait Time per HPX-Thread (Haswell)

difference between the average task duration (t_{d}) and task duration for the same experiment run on one core (t_{d1}) in Eq. 11.

$$t_{\mathrm{w}} = t_{\mathrm{d}} - t_{\mathrm{d1}} \tag{11}$$

Wait time per HPX thread increases with the number of cores and with the partition size as shown for the Haswell node in Figure 29.

To compute *wait time* per core for the entire experiment we use Eq. 12.

$$\begin{aligned} T_{\mathrm{w}} &= \frac{(t_{\mathrm{d}} - t_{\mathrm{d1}}) * n_{\mathrm{t}}}{n_{\mathrm{c}}} \\ n_{\mathrm{t}} &= \text{number of tasks} \\ n_{\mathrm{c}} &= \text{number of cores} \end{aligned} \tag{12}$$

The results of *wait time* for the Haswell node in Figures 30 - 33 and for the Xeon Phi in Figures 34 - 37 show that for fine- to medium-grained tasks *wait time* and execution time have the same behavior. This region is for partition sizes ranging from 20,000 to 1,000,000 grid points with task durations of 32 µs to 1.3 ms for the Haswell node and 1.8 to 50 ms on the Xeon Phi.

Correlation coefficients for *wait time* with execution time indicate that correlations are strong for very fine-grained tasks with values ranging from 0.87 to 0.97 and are above 0.5 for medium- and fine-grained task ranges. However, the correlations are $\sim$ -.8 for the very coarse-grained range for all core counts. This indicates a strong inverse relationship.

Wait time measures the increase in execution time of tasks when run on multiple cores compared to executing on a single core. However, *wait time* is negative for very coarse-grained tasks, indicating that the task duration is shorter when run on multiple cores than on one core. The metric uses a measure of task duration

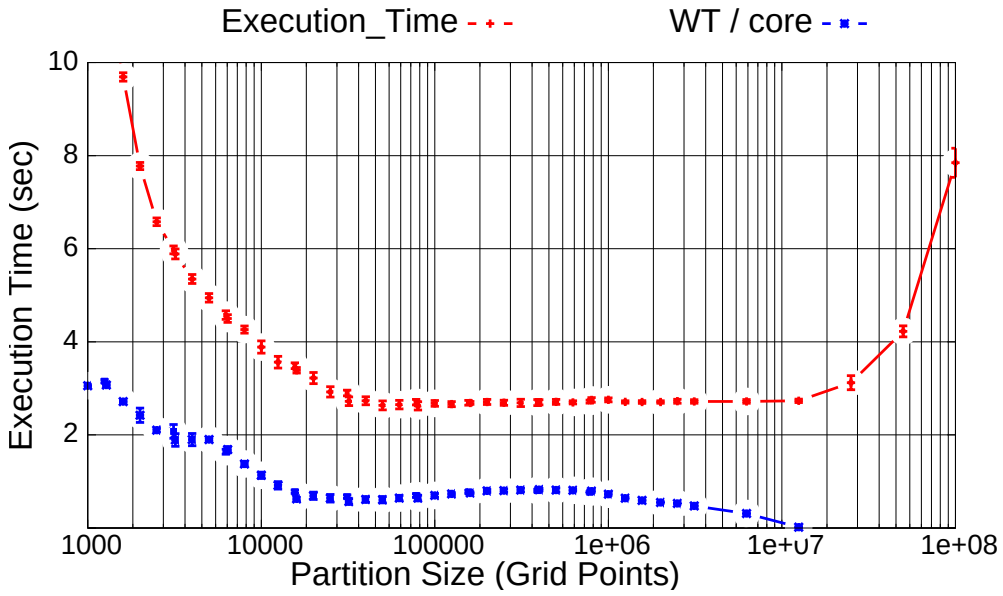

Figure 30: Haswell (4 cores): Wait Time per Core (WT/core)

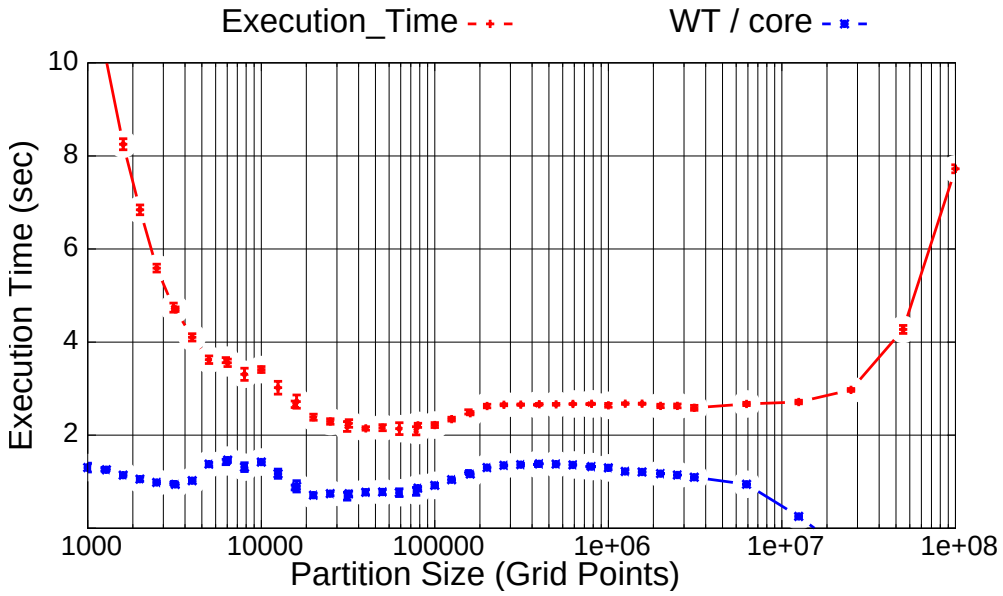

Figure 31: Haswell (8 cores): Wait Time per Core (WT/core)

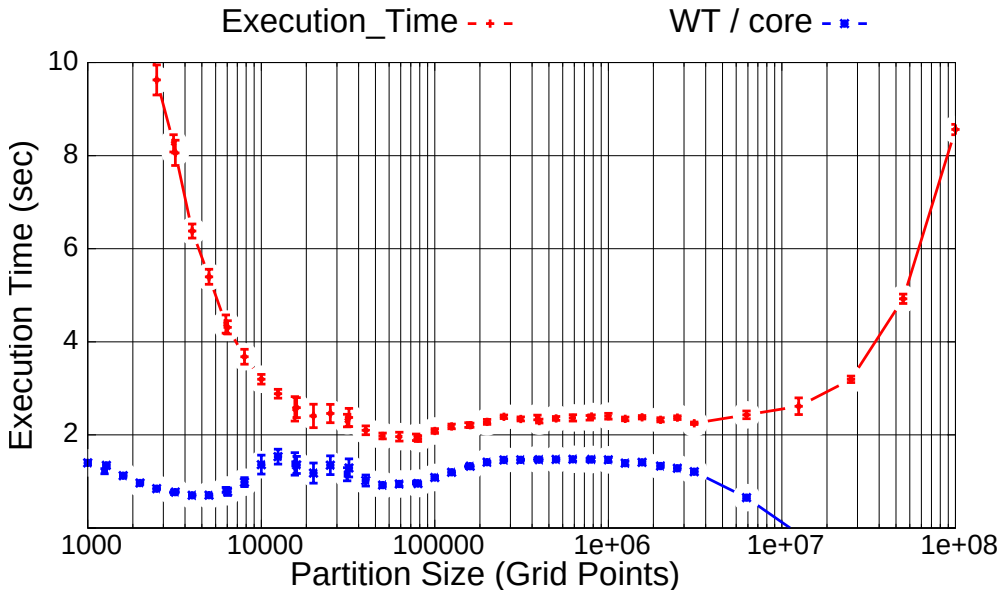

Figure 32: Haswell (16 cores): Wait Time per Core (WT/core)

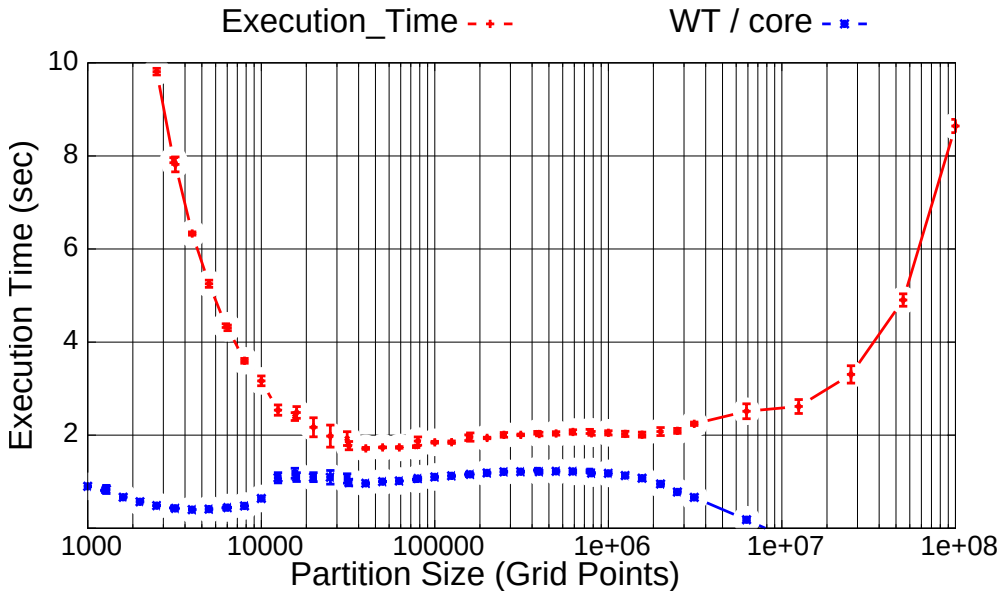

Figure 33: Haswell (28 cores): Wait Time per Core (WT/core)

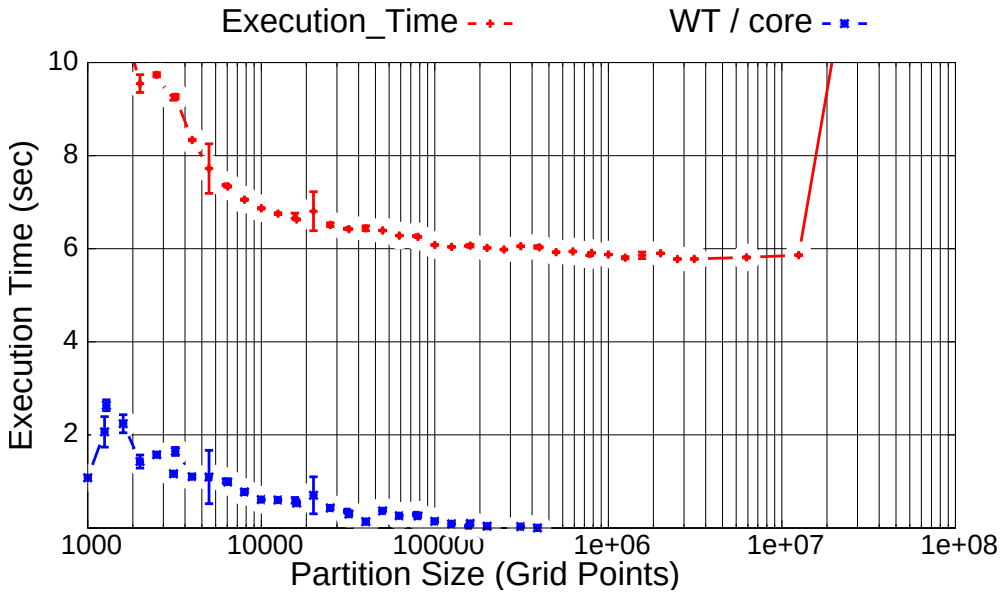

Figure 34: Xeon Phi (8 cores): Wait Time per Core (WT/core)

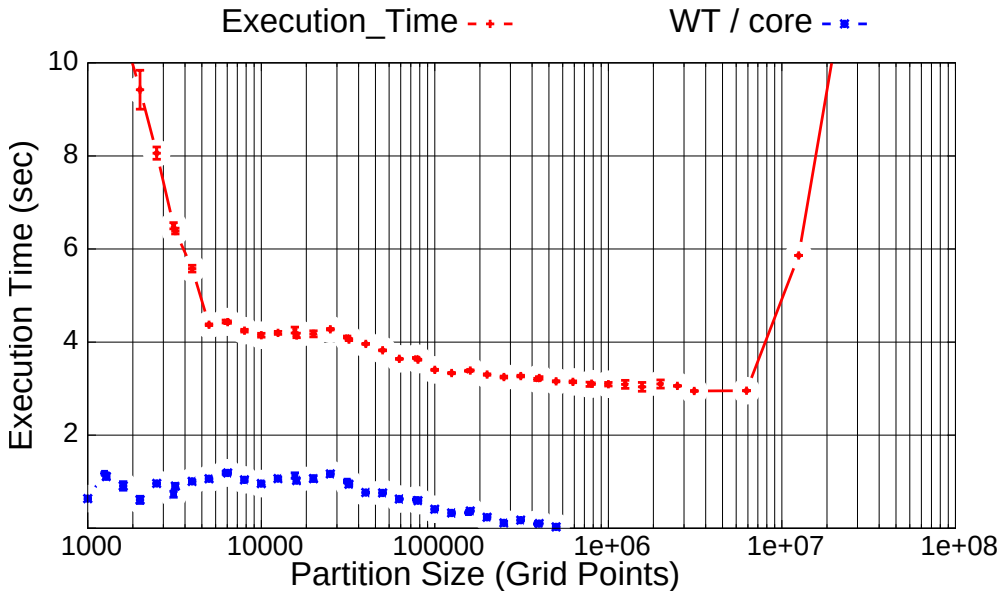

Figure 35: Xeon Phi (16 cores): Wait Time per Core (WT/core)

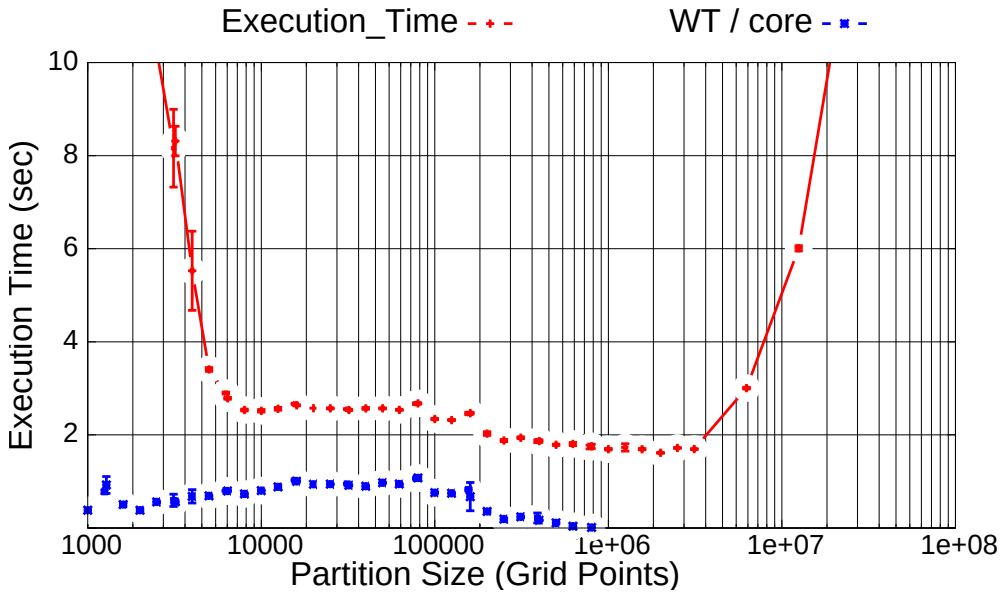

Figure 36: Xeon Phi (32 cores): Wait Time per Core (WT/core)

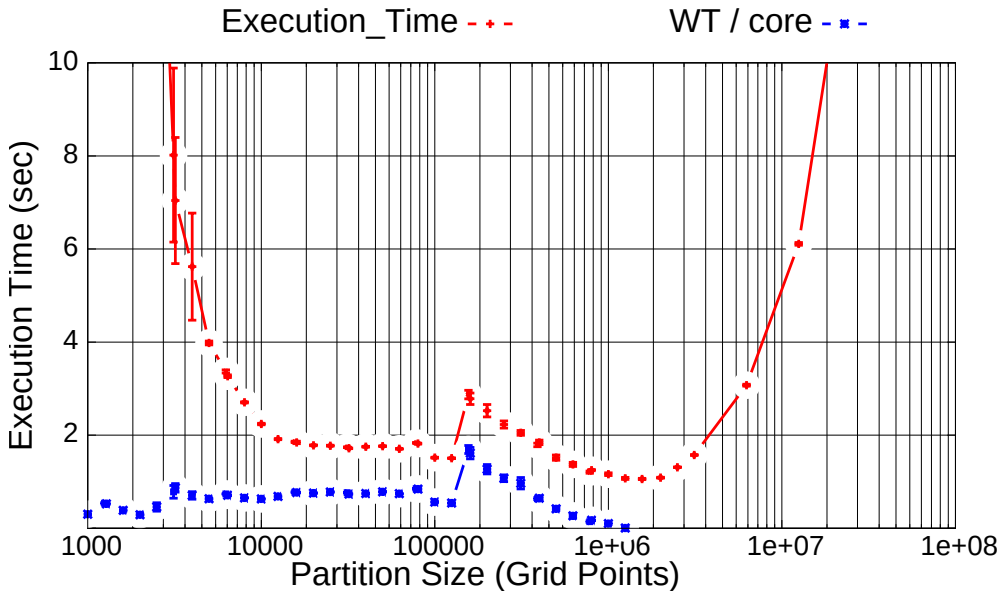

Figure 37: Xeon Phi (60 cores): Wait Time per Core (WT/core)

of all HPX threads and includes helper tasks that are invoked to support timers and synchronization. The number of these types of helper tasks increases with the number of cores used. When tasks are split into large partitions (i.e. very coarse-grained tasks), the number of tasks that perform the computations decreases and the proportion of helper tasks increases. For fine- to medium-grained tasks the proportion of helper tasks is much less than 1% of the total number of tasks, on the Haswell node, and does not perturb the calculation for task duration. For very coarse-grained tasks the proportion ranges from 11% to 130%. This is even more apparent in the results for the Xeon Phi since the number of time steps computed is 1/10 that of Haswell. *Wait time* is negative at smaller partition sizes than on the Haswell because there are fewer computations. The portion of tasks that perturb this metric is amortized when there is a large number of tasks for smaller task granularity and with longer running applications. Since task-based systems are designed for scheduling massive numbers of fine-grained tasks, the very coarse-grained tasks are beyond the range of interest but are presented for completeness.

5.2.4 Combined Costs: HPX Thread Management and Wait Time

When thread management overhead and wait time are combined, the correlations with execution time are very strong, greater than 0.9 for all task granularity ranges when executing with 16 and 28 cores and near or greater than 0.7 for all

other cases on the Haswell node, Table 2. Figures 38 through 45 shows the that the costs increase with parallelism causing the execution time to stay relatively constant after eight cores. The gap between execution time and the combined costs of thread management and *wait time* depicts the actual computation time. As the number of cores used increases (i.e. increased parallelism) the computation time decreases, but overheads and *wait time* increase, impairing performance and scaling behavior.

5.2.5 Thread Pending Queue Accesses

The number of accesses to the pending queues is a measure of the amount of activity involving the thread scheduler and does not require support from system timers. Figures 46 - 53 show that this metric for very fine-grained tasks follows the behavior of execution time. Coarsening tasks to minimize pending queue access can decrease task management overheads and reduce execution time. Correlations of the thread pending queue accesses with execution time show strong correlation for very fine-grained tasks. Correlations of pending queue accesses with execution time are greater than 0.9 for the very fine-grained and very coarse-grained task ranges, but vary considerably for the other ranges.

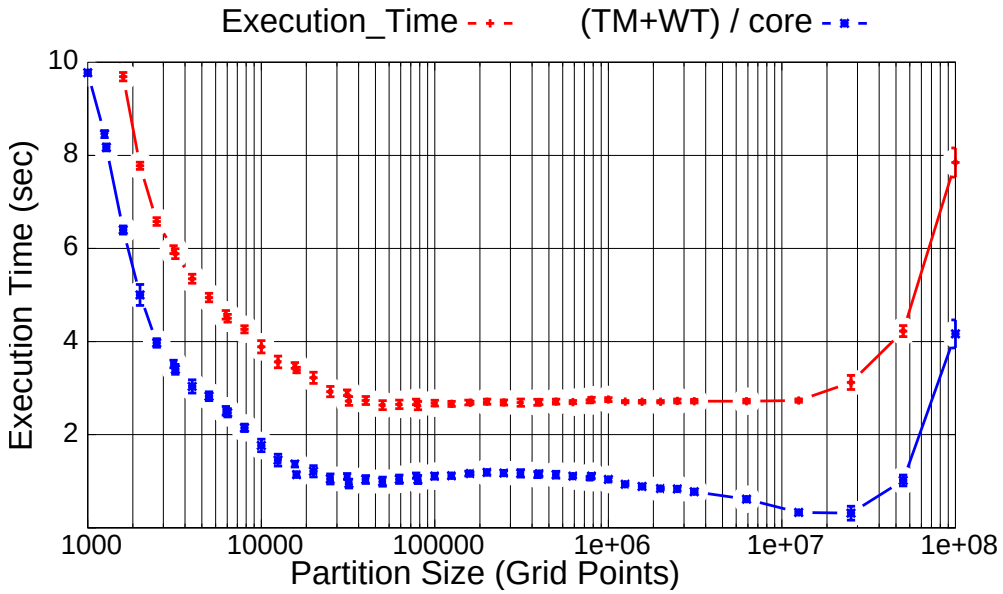

Figure 38: Haswell (4 cores): Thread Management and Wait Time per Core ((TM+WT)/core)

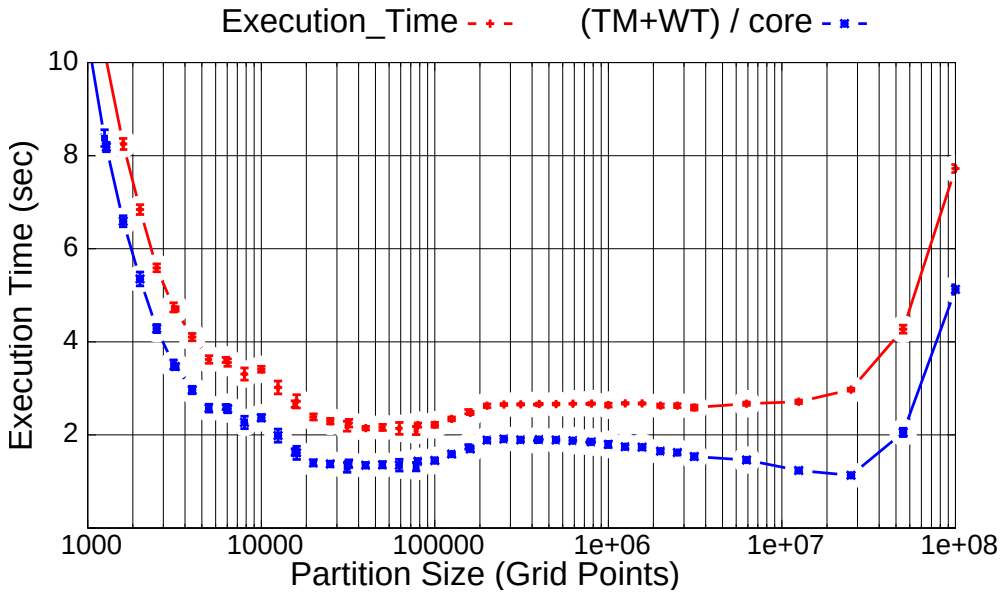

Figure 39: Haswell (8 cores): Thread Management and Wait Time per Core ((TM+WT)/core)

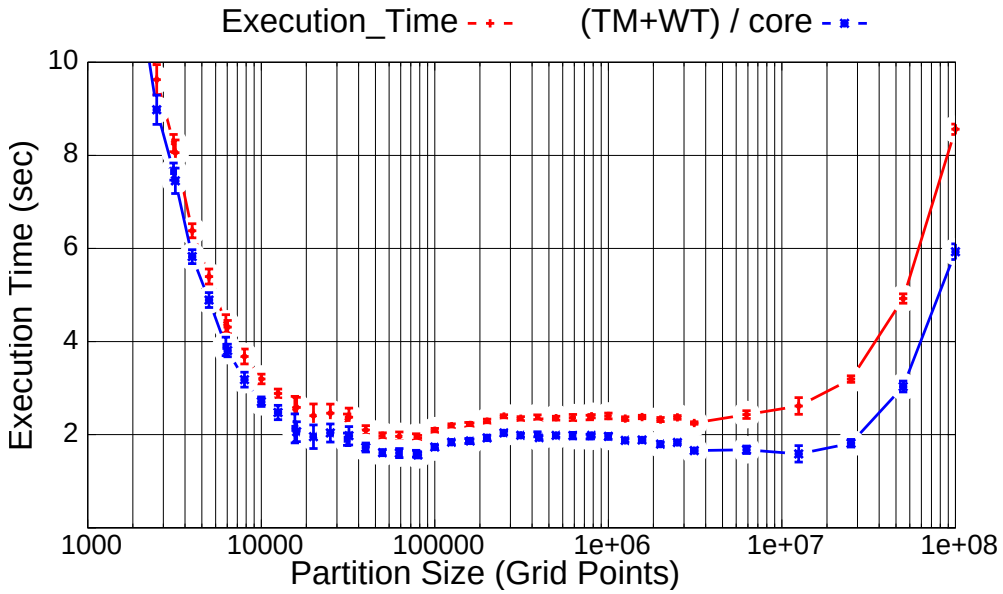

Figure 40: Haswell (16 cores): Thread Management and Wait Time per Core ((TM+WT)/core)

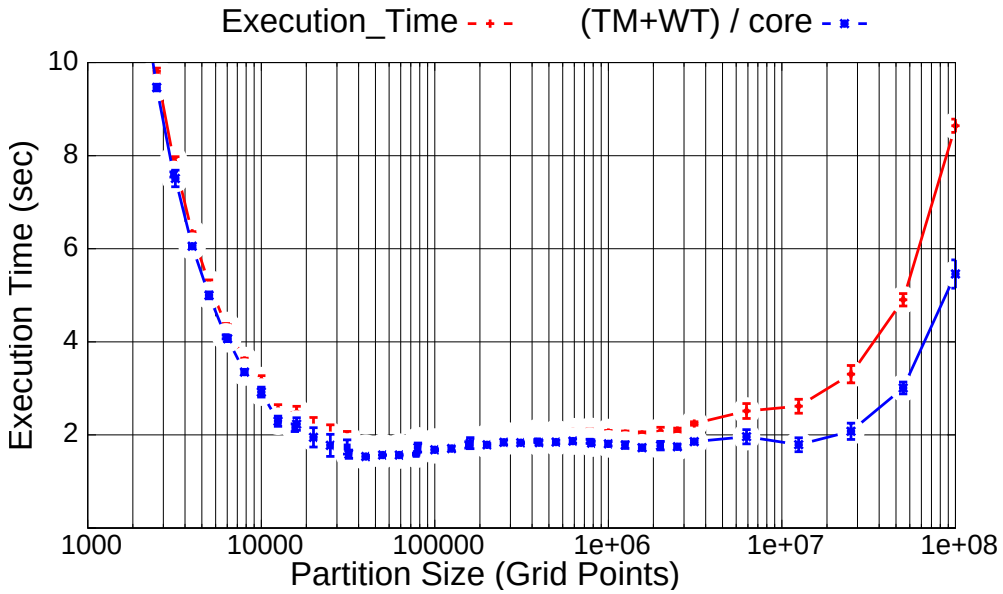

Figure 41: Haswell (28 cores): Thread Management and Wait Time per Core ((TM+WT)/core)

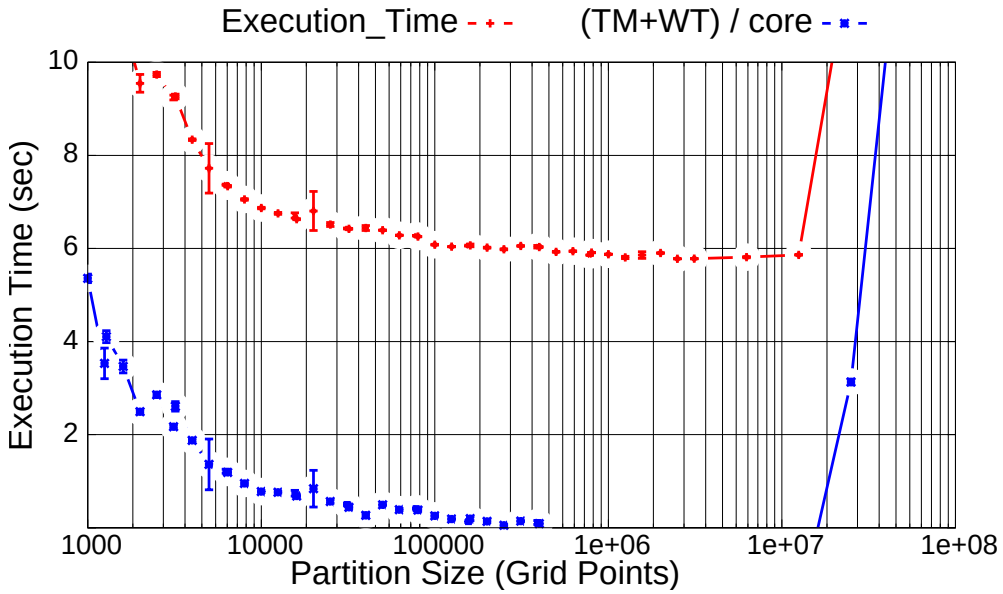

Figure 42: Xeon Phi (8 cores): Thread Management and Wait Time per Core ((TM+WT)/core)

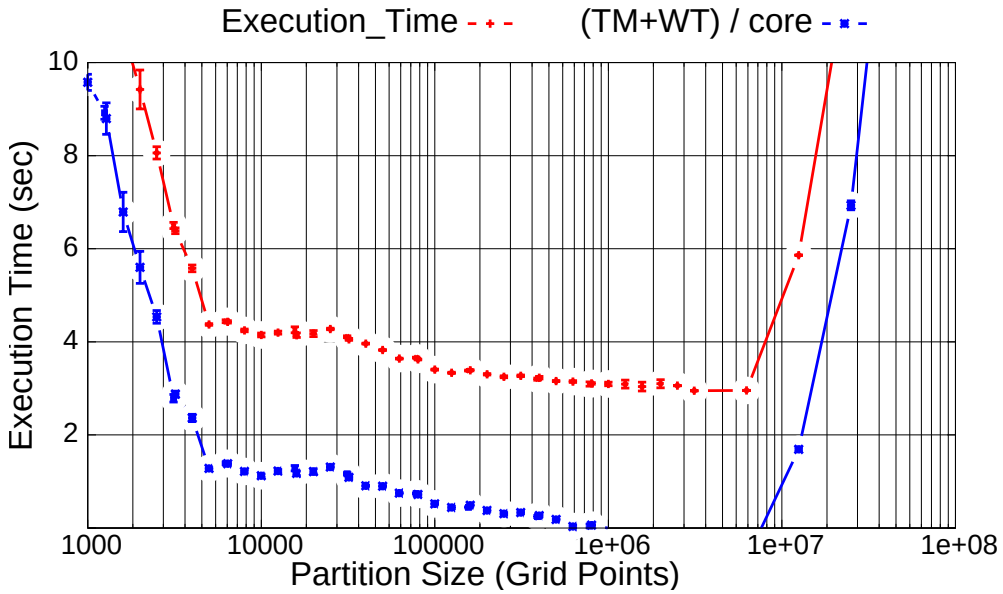

Figure 43: Xeon Phi (16 cores): Thread Management and Wait Time per Core ((TM+WT)/core)

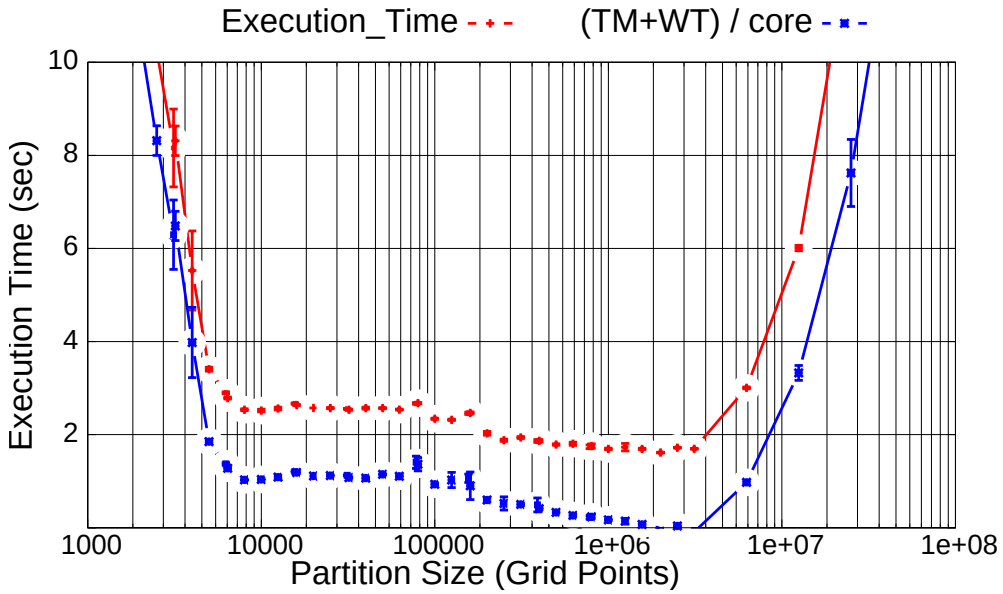

Figure 44: Xeon Phi (32 cores): Thread Management and Wait Time per Core ((TM+WT)/core)

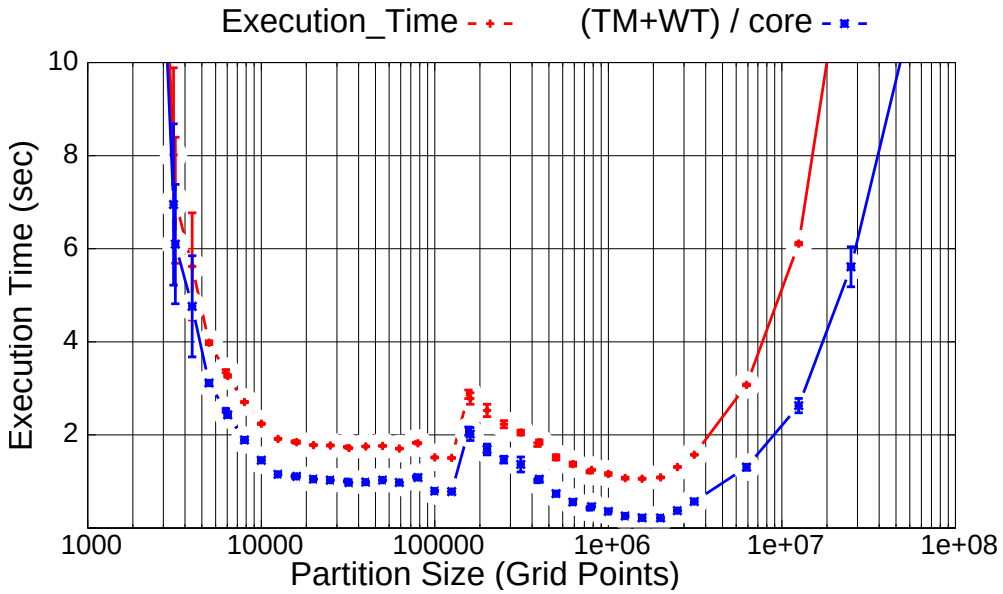

Figure 45: Xeon Phi (60 cores): Thread Management and Wait Time per Core ((TM+WT)/core)

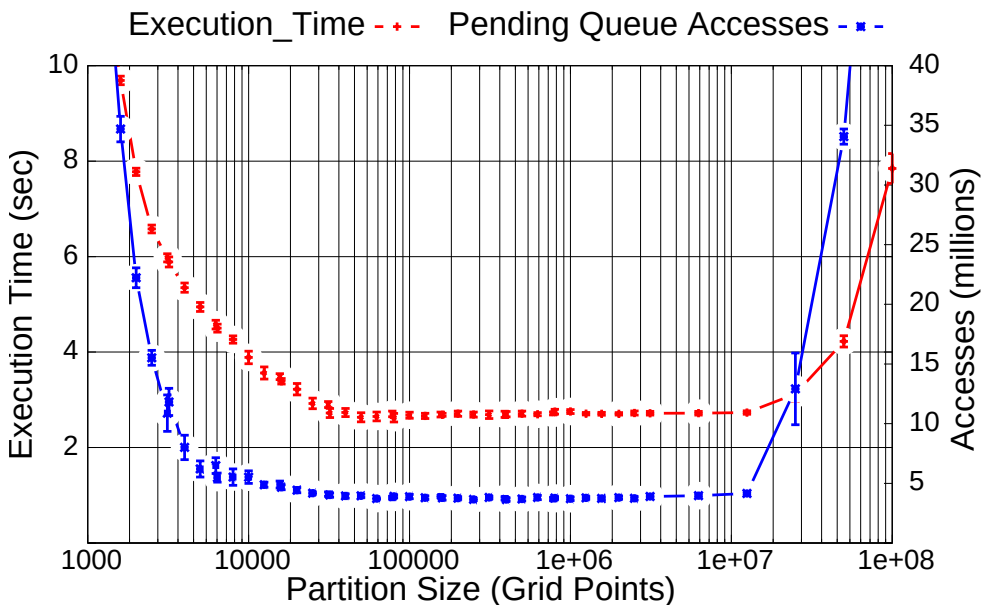

Figure 46: Haswell (4 cores): Pending Queue Accesses

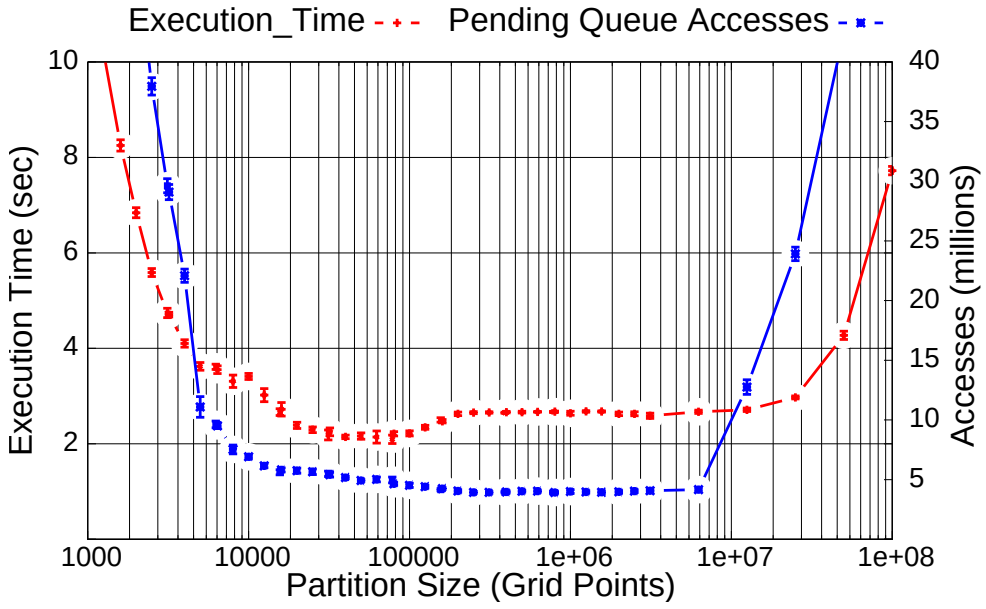

Figure 47: Haswell (8 cores): Pending Queue Accesses

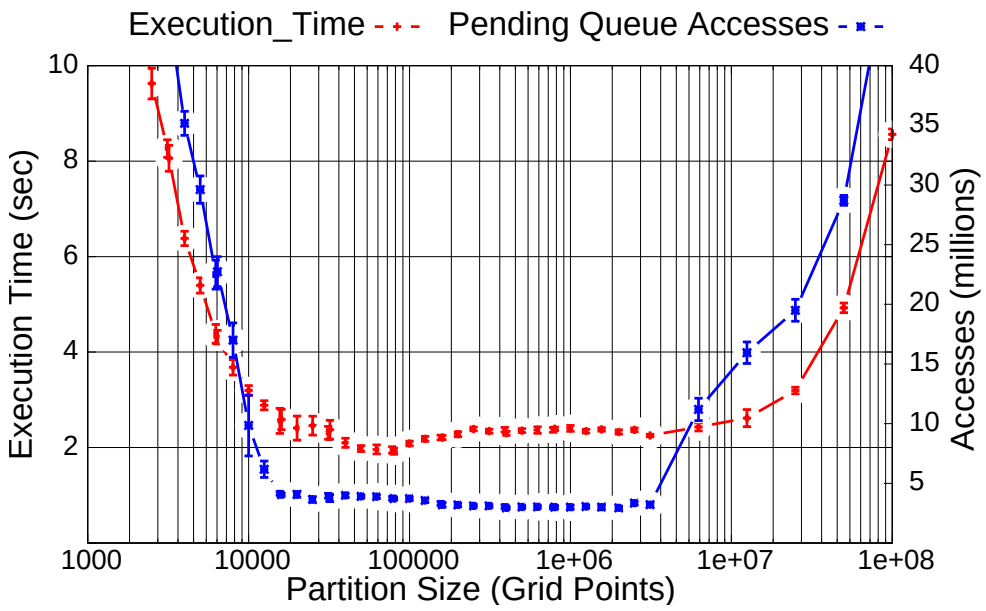

Figure 48: Haswell (16 cores): Pending Queue Accesses

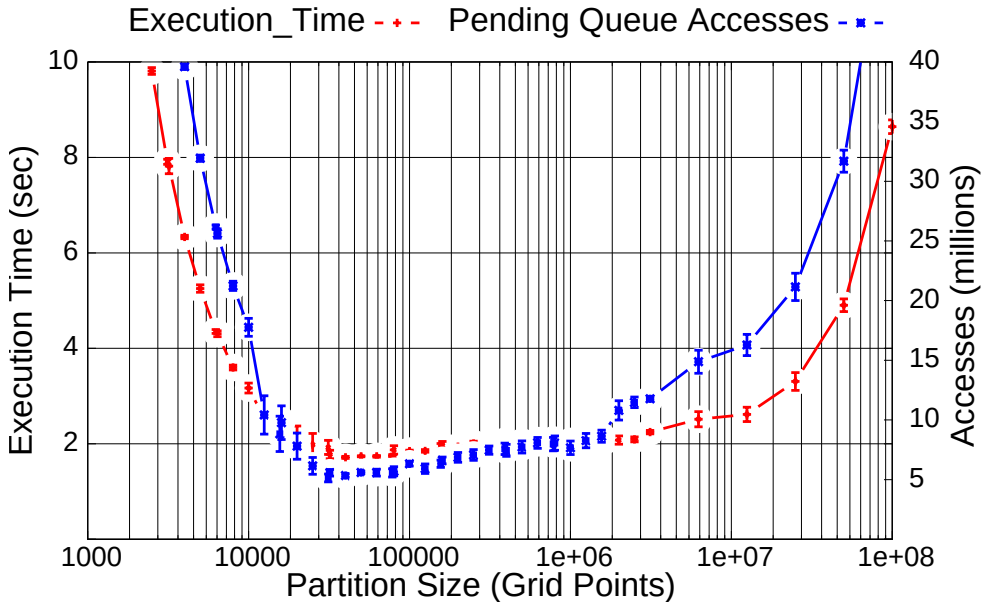

Figure 49: Haswell (28 cores): Pending Queue Accesses

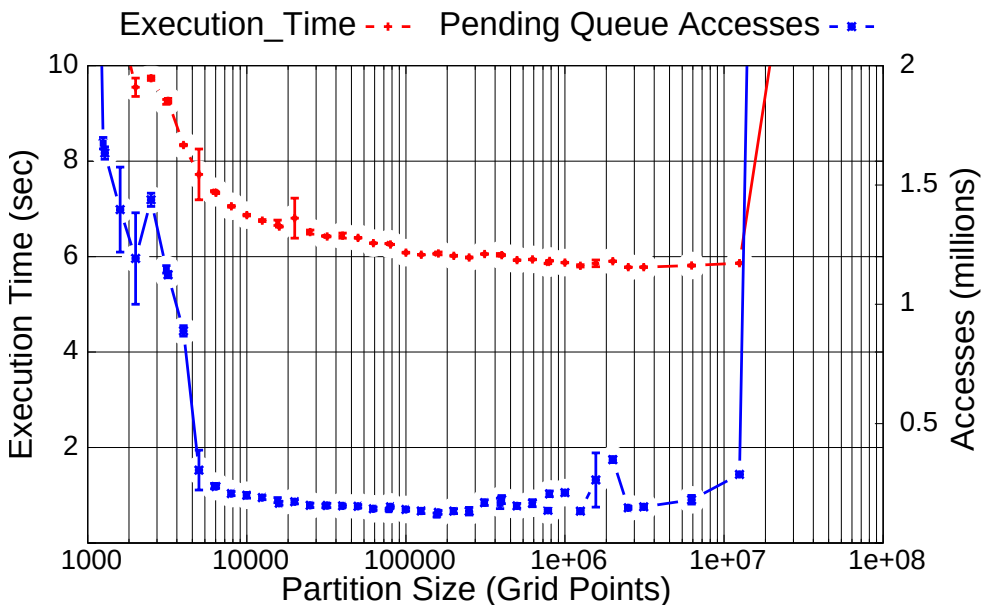

Figure 50: Xeon Phi (8 cores): Pending Queue Accesses

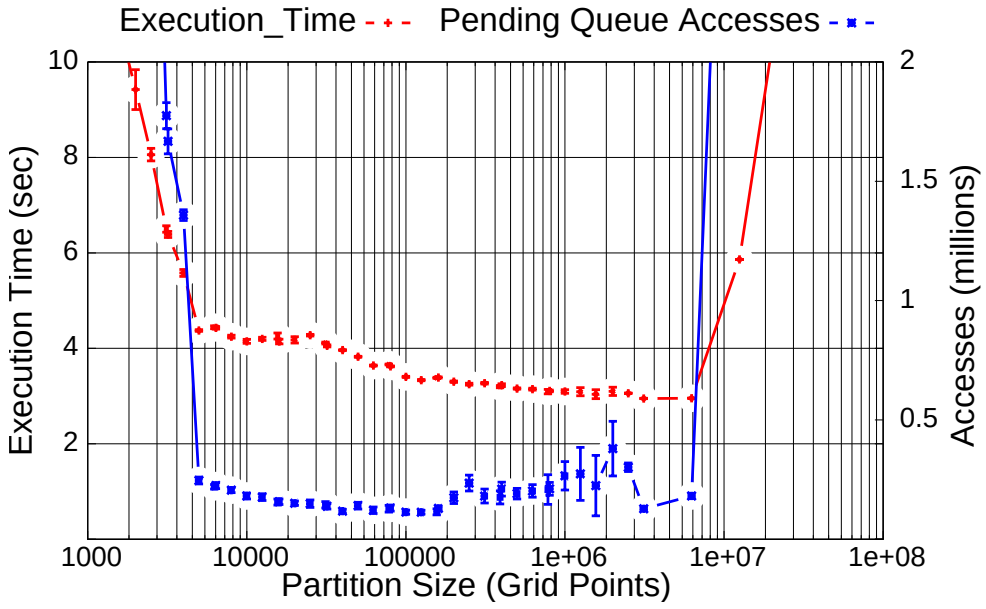

Figure 51: Xeon Phi (16 cores): Pending Queue Accesses

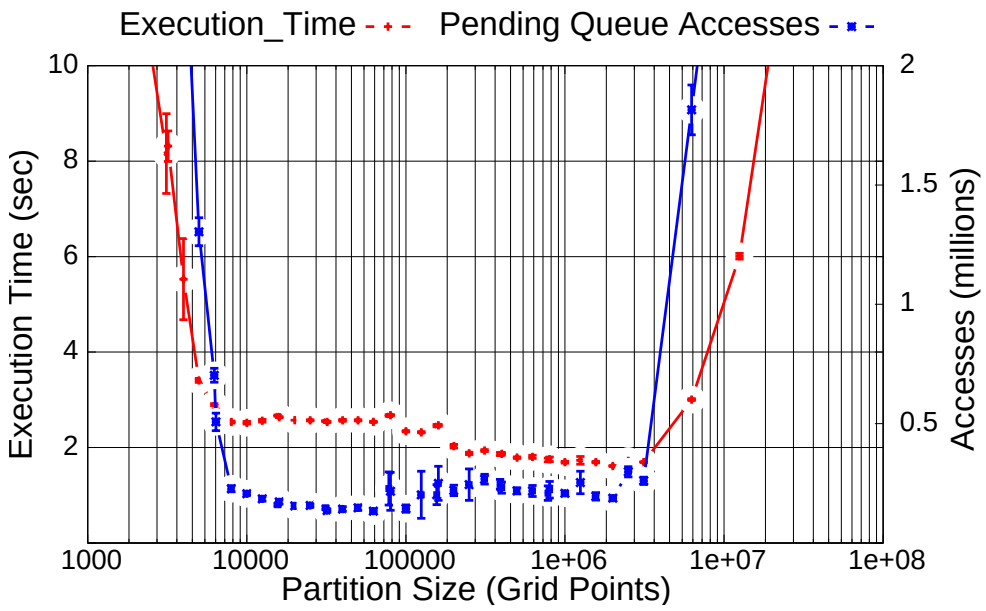

Figure 52: Xeon Phi (32 cores): Pending Queue Accesses

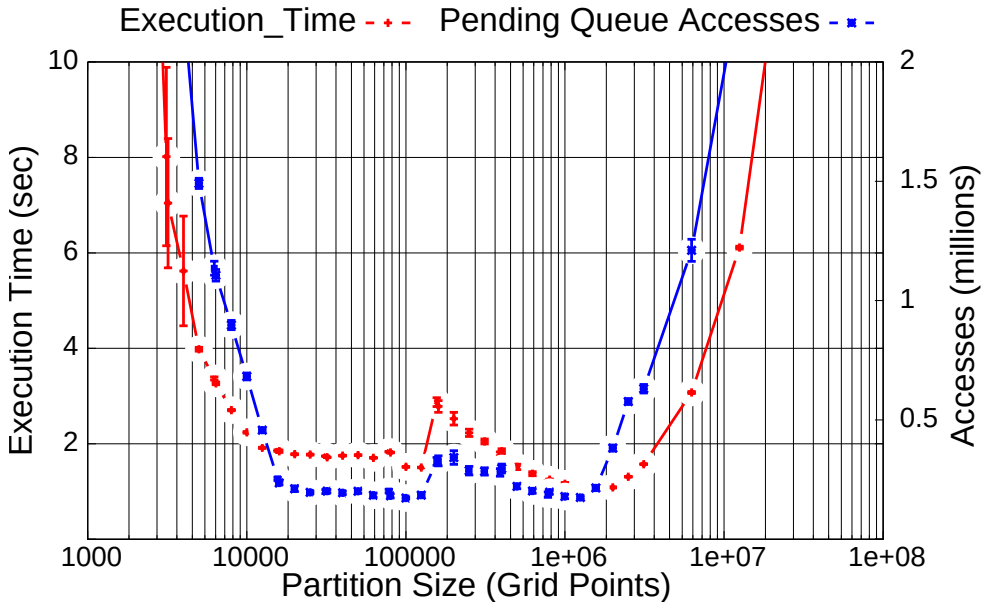

Figure 53: Xeon Phi (60 cores): Pending Queue Accesses

5.3 Summary of Task Granularity Experiments

Prior to this research, the HPX runtime system had few counters that assessed thread scheduling behavior and overheads. Developers depended on the idle-rate counter (ratio of time spent on HPX thread management to that of execution) to assess overheads and modified their code to minimize idle-rate for improved performance. This usually helps improve performance but does not consider costs other than thread management. Several counters were added to assess activity by the scheduler, such as the number of accesses and misses on queues, the time tasks spend waiting in the queues, number of tasks stolen, and average task durations. We study the behavior of these counters over a wide range of task granularity.

In this chapter, we present the characterization study of pertinent metrics that evaluate performance behavior and overheads and how they are affected by task granularity. Idle-rate and pending queue accesses are HPX counters and can be used to tune grain size to amortize task management costs. However, using them exclusively does not take into consideration other overheads caused by parallelization on the underlying architecture for fine- to medium-grained tasks.

For the one dimensional stencil benchmark on the Haswell node, correlations of task management overheads to execution time are greater than 0.9 for tasks with durations less than 27 µs and greater than 1 ms, but for tasks from 27 µs to 1 ms vary from 0.3 to 0.9. While correlations of wait time are only consistently

greater than 0.9 for tasks less than 27 µs. However, when the two metrics, task management overheads and wait time are combined, correlations to execution time are above 0.9 for all task granularities when run on more than 8 cores.

Metrics that are derived from available counts include thread management overhead and *wait time* and show that combining these two components of overheads give us the best correlation with execution time. We show that by measuring intrinsic events and calculating pertinent metrics, we can determine optimal grain size to amortize task management overheads and *wait time* for best performance. However, *wait time* is dependent on metrics from running with only one core so will have to be collected prior to runtime or as a portion of the execution causing additional overheads.

Results from this study led to the implementation of performance counters that time task execution and task management overheads. The subsequent study adapted from [31] and presented in Chapter 6, explains the challenges of performance monitoring of asynchronous task-based systems and demonstrates the capabilities of measuring intrinsic events by the runtime system for a variety of benchmarks using the new performance counters to assess application efficiency and resource usage of the underlying hardware during execution time, paving the path toward runtime adaptation.

6 USING INTRINSIC PERFORMANCE COUNTERS TO ASSESS OVERHEADS DURING EXECUTION

As the High Performance Computing community continues to explore solutions for future computational needs, computing systems are evolving quickly. Computer systems continue to grow more complex with increasing core counts per node, deeper memory hierarchies, an increasing variety of heterogeneous nodes, and networks are becoming more intricate. Additionally, applications are growing larger, more complex, and more integrated with other applications in workflows. One solution, task-based parallelism with runtime adaptivity, as proposed by this dissertation, depends on readily available, low overhead, performance metrics that are able to give an introspective view of any part of the system on demand.

Task-based programming systems, such as Charm++ [39] and OpenMP 3.0 Tasks [13] and many other research and commercial runtimes, have slowly emerged over the last three decades. Unfortunately, each model requires a specific solution to the problem of performance measurement. Current performance monitoring tools are designed for the much more common case of synchronous execution and are not able to monitor intrinsic events of asynchronous task-based parallel applications during execution. Most of the widely used open-source parallel performance measurement tools (such as HPCToolkit [9] or TAU [49]) are based on the profiling or tracing of application codes through either instrumentation or

periodic sampling. These types of tools are quite useful for post-mortem analysis and optimization of large scale parallel application codes. However, they currently fail to support massive quantities of asynchronous tasks. In addition, because they are designed for post-mortem analysis they are not easily extended to implement runtime adaptive mechanisms. The challenges of using performance monitoring tools and measurements that demonstrate the inability of currently available tools to provide the type of performance monitoring required for dynamic adaptation are presented in Section 6.1.

In addition to previously mentioned task-based programming systems, task-based parallelism is implemented in the C++11 Standard [53]. The implementation of task parallel constructs in the standard is designed to increase parallel programming productivity and portability of parallel applications with the potential of increased performance through compiler support. To assess the performance of the standard implementation, Thoman, Gschwadtner, and Fahringer introduced the Innsbruck C++11 Async Benchmark Suite (INNCABS [56]), consisting of parallel benchmarks with varying task granularities and synchronization requirements. The performance study using INNCABS illustrates the use of the C++11 Standard parallel constructs across readily available compiler platforms. However, their results demonstrate that the standard implementation of the parallel constructs do not perform well and are not adequate to replace current third party implementations of task-based parallelism.

HPX, a general purpose C++ task-based runtime system for parallel and distributed applications (see description in Section 2.3), is one solution that employs improved programming productivity and portability since its API adheres to the C++11/14 Standards [53] - [54] and is designed to run parallel tasks on current platforms and increasingly complex platforms of future designs. HPX employs a unified API for both parallel and distributed applications thus the ease of programming extends to the distributed use case. In addition, HPX implements a performance monitoring framework that enables both the runtime system and the application to monitor intrinsic events during execution.

We show the ease of porting the INNCABS benchmark suite to the HPX runtime system, the improved performance of benchmarks that employ fine-grained task parallelism and the capabilities and advantages of using the performance monitoring system in HPX to give detailed insight of the performance and behavior of benchmarks with varying task granularities and synchronization requirements.

This chapter, adopted from [31], illustrates the capabilities of the HPX runtime system to schedule massive numbers of small tasks efficiently for parallel applications with the ability to monitor intrinsic software and hardware counters at execution time. The C++11 Standard solution, based on kernel threads, is not sufficient to provide adequate scalability of parallel applications, and third party runtime libraries, such as HPX, are required for support. Measuring intrinsic events through the runtime can give the ability to assess scheduling and paral-

lelization costs and their impact on performance and provide that information to guide adaptive schedulers.

The remainder of this chapter discusses the challenges of using existing tools for performance monitoring at execution time, the ease of porting benchmarks from the C++11 Standard to HPX, the improvement in performance, and the advantages of using HPX with the ability to monitor intrinsic events. Finally, the results of experiments measuring performance counters to assess overheads and performance of the application are presented.

6.1 Challenges Using Performance Monitoring Tools

It is important to understand how C++ parallel constructs are distinct from other thread-parallel models, and why it is so difficult to measure and understand the performance behavior. The Standard C++ implementation constructs, executes, and destroys an Operating System (OS) thread for every task created with `std::async`, resulting in thousands or even millions of OS threads being created. While this implementation is certainly within the capabilities of the OS kernel, it is somewhat naive and inefficient and presents a significant challenge to performance tools that are not explicitly designed to support this type of implementation.

Widely used open-source parallel performance measurement tools like HPC-Toolkit [9] and TAU [49] provide profiling and/or tracing of many different types of parallel application models. These tools use several methods to observe ap-

plication behavior, including source instrumentation, binary instrumentation, library wrappers, performance tool interfaces, callbacks, and/or periodic sampling based on interrupts combined with call stack unwinding. These tools are capable of context-aware measurement with variable levels of resolution and subsequent overhead. Large scale parallel codes with concurrency greater than hundreds of thousands of processes and/or threads have been successfully measured.

However, these tools fail to adequately support the current implementation of C++ task parallel constructs. Both TAU and HPCToolkit make design assumptions about the overall number of OS threads they expect to see in a given process space. In the case of TAU, the data structures used to store performance measurements are constructed at program launch to minimize perturbation of the application during execution. While the maximum number of threads per process is a configurable option (default=128), it is fixed at compilation time. When set to a much larger number (i.e. 64k) TAU causes the benchmark programs to crash. While HPCToolkit doesn't set a limit on the number of threads per process, the introduced overhead becomes unacceptable as each thread is launched and the file system is accessed, and in most benchmark cases the program crashes due to system resource constraints. Table 3 shows the results of running the C++11 Standard INNCABS benchmarks at full concurrency with either TAU or HPCToolkit using the test system and protocol described in Section 6.2.2.

In addition, because they are designed for post-mortem analysis these tools

are not easily extended to implement runtime adaptive mechanisms. In both cases, post-processing of the performance data (usually done at program exit) is required before an accurate performance profile containing the full system state (across nodes and/or threads) is possible.

Table 3: C++11 Standard INNCABS Executed with TAU and HPCTookKit. When the benchmarks are executed with these tools resulting overheads and failures confirm that the tools fail to support the current implementation of C++ task parallel constructs.

	Baseline		**TAU**	**HPCToolkit**	
Benchmark	time	tasks	time	time	overhead
Alignment	971	4950	SegV	112,795	11516%
FFT	48,423	2.04E+06	SegV	timeout	
Fib	Abort	N/A	SegV	N/A	
Floorplan	5,788	169708	SegV	SegV	
Health	589,415	1.75E+07	Abort	Abort	
Intersim	827	1.70E+06	Abort	SegV	
NQueens	Abort	N/A	N/A	N/A	
Pyramids	2,148	112,344	SegV	275,088	12707%
Qap	SegV	N/A	N/A	N/A	
Round	155	512	SegV	5,588	3505%
Sort	7,240	328,000	SegV	Abort	
Sparselu	786	11,099	SegV	99,123	12511%
Strassen	4,782	137,256	SegV	Abort	
UTS	Abort	N/A	N/A	N/A	

In contrast, Autonomic Performance Environment for eXascale (APEX) [34, 33] has been designed for the HPX runtime to provide performance introspection and runtime adaptation using the available HPX performance monitoring framework. A discussion of the current state and future potential of APEX is provided in Chapter 7.

6.2 Performance Counter Experimental Methodology

To demonstrate the capabilities of the HPX runtime system and its performance monitoring system, we run strong scaling experiments with the C++11 Standard and HPX versions of the INNCABS benchmarks by increasing the number of cores while keeping the total workload constant for each benchmark. This section describes the benchmarks, system configuration, performance counter measurements, and the methods used to run the experiments.

6.2.1 Benchmarks

In order to assess efficiency of task-based parallel programs using intrinsic performance counters, we measure events that determine the costs of parallelization and task scheduling, then model the correlation of the costs to performance. We measure these events for benchmarks with a variety of task granularity, parallel structures, and synchronization requirements. To accomplish this, we ported to HPX the INNCABS benchmark suite [56], introduced by Thoman, Gschwadtner, and Fahringer as a suite of benchmarks using the C++11 Standard constructs for thread parallelism. The benchmarks are written and/or ported to C++11 to assess the performance achieved by using the C++11 Standard thread mechanisms for parallel applications without the support of third party libraries. Table 7 in Section 6.4 classifies the benchmarks by programming structure and synchronization and includes task granularity and scaling results of both versions. The

benchmarks are classified by the parallel structure of the tasks. Recursive parallelism is formed by scheduling trees of asynchronous tasks. Balanced recursive structures have the same number of tasks per subtree, and unbalanced recursive structures have variable number of tasks per subtree. Loop-like parallel structures schedule asynchronous tasks in a for or while loop. Co-dependent parallel structures schedule tasks that depend on mutexes shared between the tasks for synchronization.

Since HPX's API is modeled after the C++ Standards, replacing the standard task parallel structures with HPX equivalents for the INNCABS parallel benchmarks is fairly simple. This involved setting defitions to use the HPX library when compiling for HPX, creating cmake files for compilation, and in most cases for each benchmark changing the function's namespace (see Table 4). As defined in the C++ Standard, the template function `std::async`:

> ... runs the function `f` asynchronously 'as if on a new thread' (potentially in a separate thread that may be part of a thread pool) and returns a `std::future` that will eventually hold the result of that function call. [1]

The `std::thread` class is a convenience wrapper around an OS thread of execution, and the `std::mutex` class is:

> ... a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. [2]

[1] http://en.cppreference.com/w/cpp/thread/async
[2] http://en.cppreference.com/w/cpp/thread/mutex

Sources and detailed descriptions of the INNCABS [5] benchmarks are available online as are the HPX ported versions [7].

HPX significantly improves the performance of the benchmarks with sufficient concurrency due to the smaller overheads of the fine-grained lightweight user level HPX threads when compared to the use of pthreads by the C++11 Standard (see Section 6.4). For the applications where the tasks are coarse-grained, the overheads are not as significant, so the HPX versions either only slightly outperform or perform close to the C++11 Standard versions.

Table 4: Translation of Syntax: C++11 Standard to HPX

C++11 STD		**HPX**
std::async	$- - - >$	hpx::async
std::future	$- - - >$	hpx::future
std::thread	$- - - >$	hpx::thread
std::mutex	$- - - >$	hpx::lcos::local::mutex

6.2.2 Configurations

Our experiments are performed on an Intel® node on the Hermione cluster at the Center for Computation and Technology, Louisiana State University, running Debian GNU/Linux kernel version 3.8.13. The node is an Ivy Bridge dual socket system with specifications shown in Table 5. We run experiments with hyper-threading activated and compare results for running one thread per core to running two threads per core resulting in small change in performance. We deactivate hyper-threading and present only results with hyper-threading disabled. The

software is built using GNU C++ version 4.9.1, GNU libstdc++ version 20140908, and HPX version 0.9.11 (8417f14) [38].

Table 5: Platform Specifications for Performance Counter Experiments

Node	**Ivy Bridge (IB)**
Processors	2 Intel®Xeon®E5-2670 v2
Clock Frequency	2.5 GHz (3.3 turbo)
Microarchitecture	Ivy Bridge (IB)
Hardware Threading	2-way (deactivated)
Cores	20
Cache/Core	32 KB L1(D,I) 256 KB L2
Shared Cache	35 MB
RAM	128 GB

For best performance, the HPX benchmarks are configured using tcmalloc [3] for memory allocation. Comparisons were made for the standard benchmarks using system malloc and tcmalloc. The C++11 Standard versions perform best using the system memory allocator except for the Alignment benchmark. The original Alignment benchmark allocates large arrays on the stack, and execution fails for the default HPX stack size (8 KB), so the benchmark was modified to allocate the arrays on the heap for both versions. We build the standard benchmarks with the system allocator with the exception of Alignment since it performs best using tcmalloc.

The original INNCABS benchmarks can be run with any of three launch policies (*async*, *deferred*, or *optional*) as specified by the user. HPX options include these launch policies and a new policy, *fork*, added in version 0.9.11. The *fork*

[3]http://goog-perftools.sourceforge.net/doc/tcmalloc.html

launch policy is equivalent to *async* except that it enforces continuation stealing instead of child stealing, the default. Continuation stealing allows the parent thread to be stolen, and the child thread can be executed on the current processing unit, while child stealing is the opposite. This can result in performance improvements for strict fork/join use cases, where the future returned from *async* is guaranteed to be queried from the current OS-thread. We compared performance of all launch policies for both C++11 Standard and HPX versions of the benchmarks and found the *async* policy provides the best performance, so we only present the results using the *async* policy.

6.2.3 Performance Counter Metrics

To demonstrate the capabilities of the performance monitoring system, we select both runtime and hardware performance counters to observe. The software counters used in this research measure the task execution times, overheads, and efficiency of the thread scheduler in order to monitor performance of the runtime's thread scheduling system and execution performance of the tasks on the underlying hardware. The hardware counters we use demonstrate the ability to measure hardware events that are available to ascertain information that can be used for decision making such as throttling the number of cores used to save energy. Although for this research we run benchmarks that are designed for parallelism on one node, HPX performance counters can also be utilized for distributed applica-

tions to make decisions such as when to migrate data.

The HPX performance monitoring framework provides the flexibility to inspect events over any interval of the application. The INNCABS applications have support for multiple execution samples in one run. This enables us to select the number of samples and run them with one execution of the program without the overhead of starting HPX. The counters are read for each sample by using the HPX *evaluate* and *reset* API calls. HPX also includes the ability to monitor events for predefined timed intervals and for individual OS threads. For these experiments we use the cumulative total of the OS thread counts.

There are more than 50 types of performance counters available in HPX, many of those have more than 25 subtypes [4]. The counters are grouped into four groups representing the main subsystems of HPX: AGAS counters, Parcel counters, Thread Manager counters, and general counters. There are also mechanisms for aggregating the counters or deriving ratios from combinations of counters. From this large group of counters, we only use a few that demonstrate their general functionality and provide the metrics required. The metrics and their associated performance counters for these experiments are:

Task Duration

The value of the */threads/time/average* counter that measures the average time spent in the execution of an HPX thread, also referred to as an HPX

[4]http://stellar-group.github.io/hpx/docs/html/hpx/manual.html

task. *Task durations* for runs using one core give us a measure of task granularity and are reported in Table 7. When the number of cores is increased we observe an increase in *task duration* that indicates the execution is delayed due to overheads caused by parellization on underlying hardware. This is illustrated in Figure 29, Section 5.2.3.

Task Overhead

The value of the */threads/time/average-overhead* counter measures the average scheduling cost to execute an HPX thread. We observed task overheads on the order of 50-100% of the task grain size for benchmarks comprising primarily very fine-grained tasks (< 10 µs).

Task Time

Task time measures the cumulative execution time of HPX tasks during the selected interval using the */threads/time/cumulative* counter. We divide *task time* by the number of cores to show the relation to the execution time of the application.

Scheduling Overhead

The measurement of time spent in scheduling all HPX tasks using the */threads/time/cumulative-overhead* counter is *scheduling overhead*. To distinguish this from *task overhead*, this is the cumulative of task overheads for all tasks. For fine-grained applications, *scheduling overheads* can be a

major cost because the cost of scheduling the tasks is large in comparison to the task execution time.

Bandwidth Utilization

Bandwidth Utilization is estimated for the Ivy Bridge node by summing the counts of the off-core memory requests for all data reads, demand code reads, and demand reads for ownership. The count is then multiplied by the cache line size of 64 bytes and divided by the execution time. The counters are accessed in HPX as:

```
papi/OFFCORE_REQUESTS:ALL_DATA_RD
papi/OFFCORE_REQUESTS:DEMAND_CODE_RD
papi/OFFCORE_REQUESTS:DEMAND_RFO
```

Measuring hardware events (such as the off-core requests) through the HPX interface to PAPI gives the application information about the behavior on the particular system. Using native PAPI counters for the Ivy Bridge node, we compute an estimated offcore memory bandwidth utilization.

The overhead caused by collecting these counters is very small (within variability noise), but sometimes are up to 10% for the benchmarks with very fine-grained tasks when run on one or two cores.

6.3 Performance Counter Experiments

Based on the wide range of available possibilities, we conduct a large number of experiments to determine the best configuration for the build and run policies that provide the best comparisons of the benchmarks. Table 6 lists the configurations we use for the experiments. We present experiments that give a fair comparison between C++11 Standard and HPX and illustrate the capabilities of the HPX runtime system for scheduling asynchronous tasks on parallel systems with the benefit of measuring intrinsic performance events.

Table 6: Software Build and Run Specifications for INNCABS

Specification	**C++11 STD**	**HPX**
Compiler	gcc	gcc
Memory Allocation	system*	tcmalloc
Launch Policy	async	async

*tcmalloc – memory allocator used for C++11 STD Alignment

To assess performance of the benchmarks, we use strong scaling by increasing the number of cores while keeping a fixed workload. The one exception to this is the Floorplan benchmark. The `std::async` implementation uses a single task queue from which all threads are scheduled. In comparison, the HPX implementation provides a local task queue for each OS thread in the thread pool. Because of this subtle difference, the two implementations execute the tasks in a different logical ordering, causing the `hpx::async` implementation to evaluate many more possible solutions prior to pruning. In fact, the HPX implementation evaluates

over two orders of magnitude additional solutions, although the time per task is much smaller. Further study is needed to understand whether the HPX implementation can prune the search space more effectively. For the purposes of this dissertation, a fixed limit on the number of total tasks executed is added to ensure a fair comparison between the two runtimes. The input sets used in the original INNCABS paper [56] are used for each benchmark, with the exception of QAP. QAP only runs successfully using the smallest input set included with the original sources.

To maximize locality, we pin threads to cores such that the sockets are filled first. For the C++11 Standard version we use the command `taskset` and specify the proper CPU order to ensure proper affinity; this is tricky since logical core designations vary from system to system. HPX utilizes HWLOC and provides flexible thread affinity support through the `--hpx:bind` command line option. We verify that both versions properly use thread affinity by monitoring test runs using the `htop` utility.

For each experiment 20 samples are collected, however, the first sample is excluded since for these benchmarks initialization is measured in the first loop. We present the mean of the remaining 19 samples for the execution times and the counters. We compute and plot the 95% confidence interval for the execution times. The confidence intervals are small and most are not visible on the graphs. To measure the counter data, we evaluate and reset the counters for each sample

using `hpx::evaluate_active_counters` and `hpx::reset_active_counters` API functions.

6.4 Performance Counter Experimental Results

To illustrate the capability of performance monitoring on a variety of bench-

Table 7: Benchmark Classification and Granularity

Class	**Sync.**	**Task Dur.**	**Gran.**	**Scaling**		**Speedup**	
Benchmark		(µs)		STD	HPX	STD	HPX
Loop Like							
Alignment	N	2748	C	20	20	15.4	17.3
Health	N	1	VF	AF	10	-	2.8
Sparselu	N	980	C	20	20	12.2	15.6
Recursive Balanced							
FFT	N	1	VF(var)	6	6	1.3	1.5
Fib	N	1	VF	AF	10	-	3.8
Pyramids	N	246	M	20	20	8.0	12.9
Sort	N	52	F(var)	10	20	3.0	11.9
Strassen	N	107	F	SF 8	20	3.1	11.1
Recursive Unbalanced							
Floorplan	AP	5	VF	8	10	1.2	2.1
NQueens	N	28	F	AF	20	-	11.2
QAP	AP	1	VF	6	20	1.1	7.8
UTS	N	1	VF	AF	10	-	3.2
Co-dependent							
Intersim	MM	3	VF	AF	12	-	2.7
Round	2M	9671	C	20	20	20	18.7

Notes: 1. Sync. (synchronization): N-none, AP-atomic pruning
MM-multiple mutex per task, 2M-two mutex per task
2. Task Dur. (average task duration)
3. Gran. (granularity): var.-variable
VF - very fine (< 30 µs) F - fine (30 - 100 µs)
M - medium (100 - 500 µs) C - coarse (> 500 µs)
4. Scaling Behavior: Scales up to number of cores
AF-all fail, SF-some fail

marks, we run experiments using the 14 benchmarks from the INNCABS benchmark suite. Table 7 is an expansion of Table 1 in [56] that shows the structure of the benchmarks. We include measurements of *Task Duration* (task grain size) and classify the granularity according to our measurements of the HPX performance counter when the benchmark is run on one core. Included are the scaling behaviors of both the C++11 Standard and HPX versions measured in our experiments. Even though the benchmarks have a variety of structures and synchronization, the most prominent factor that affects scaling behavior and overall performance for these task parallel benchmarks is task granularity. In every category the HPX version of the benchmarks with very fine-grained tasks scale only to 10 cores and the speedup is only 3 to 4, 30-40% of the maximum expected for 10 cores. When run on more than 10 cores, or beyond the socket boundary, speedup decreases. The benchmarks with fine-grained tasks are all in the recursive category and all scale to 20 cores with speedups around 11 to 12, 55 to 60% of maximum. There is only one benchmark, Pyramids, in the medium-grained category and it also scales to 20 cores with slightly higher speedup of 13, 65% of maximum. Two of the coarse-grained benchmarks, Alignment and Sparselu have loop like parallel structures and one has a co-dependent structure, but all three have speedups above 15, 76% of the maximum speedup.

In several cases, the performance of the benchmarks are similar to others with the same task granularity, so we present a cross section of results that represents

each category of task granularity. Appendix B contains graphs of the performance and overheads of the benchmarks not shown in this chapter. First, we present execution time and speedup for comparison of the two runtime libraries and then the respective performance metrics. The coarse-grained benchmarks are Alignment, SparseLU, and Round. Figure 54 shows execution time and speedup for Alignment, a good representation of the scaling behavior for all three benchmarks. These benchmarks are all coarse-grained with task grain size ranging from ∼1 ms to ∼10 ms. Scheduling overheads for coarse-grained tasks are a small percentage of task duration. The benchmarks scale best for HPX with linear speedup that exceeds 76% of maximum for 20 cores.

The Pyramids benchmark (Figure 55) has a medium grain size of ∼250 µs and is the only application that executes faster for the C++11 Standard version than the HPX implementation. Although the C++11 Standard version executes faster up to 14 cores, it has a speedup factor of 8 for 20 cores, while for HPX there is a speedup of 13, and the minimum execution times are equivalent.

Strassen, Sort, and NQueens classify as fine-grained benchmarks with task grain sizes ∼100 µs, ∼50 µs and ∼25 µs respectively. For each of these benchmarks, HPX shows the ability to scale to 20 cores, while the standard version either does not run (NQueens and some Strassen experiments) or only scales up to 10 cores like Sort. The behavior of the execution time of the fine-grained benchmarks are typified by that for Strassen, (Figure 56) and Sort (Figure 57).

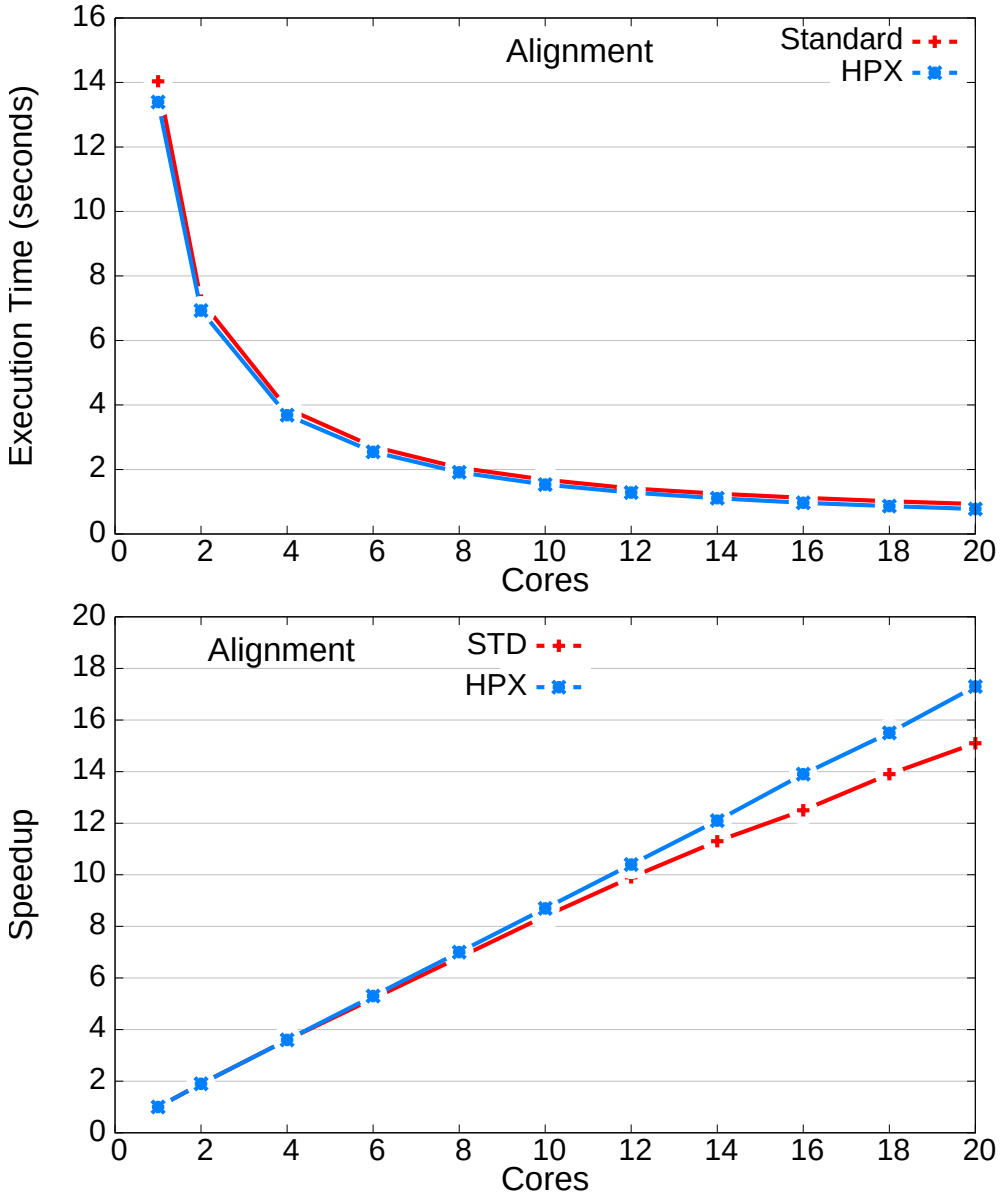

Figure 54: Alignment: HPX vs. C++11 Standard, grain size ∼3 ms, typifies scaling behavior of the coarse-grained benchmarks.

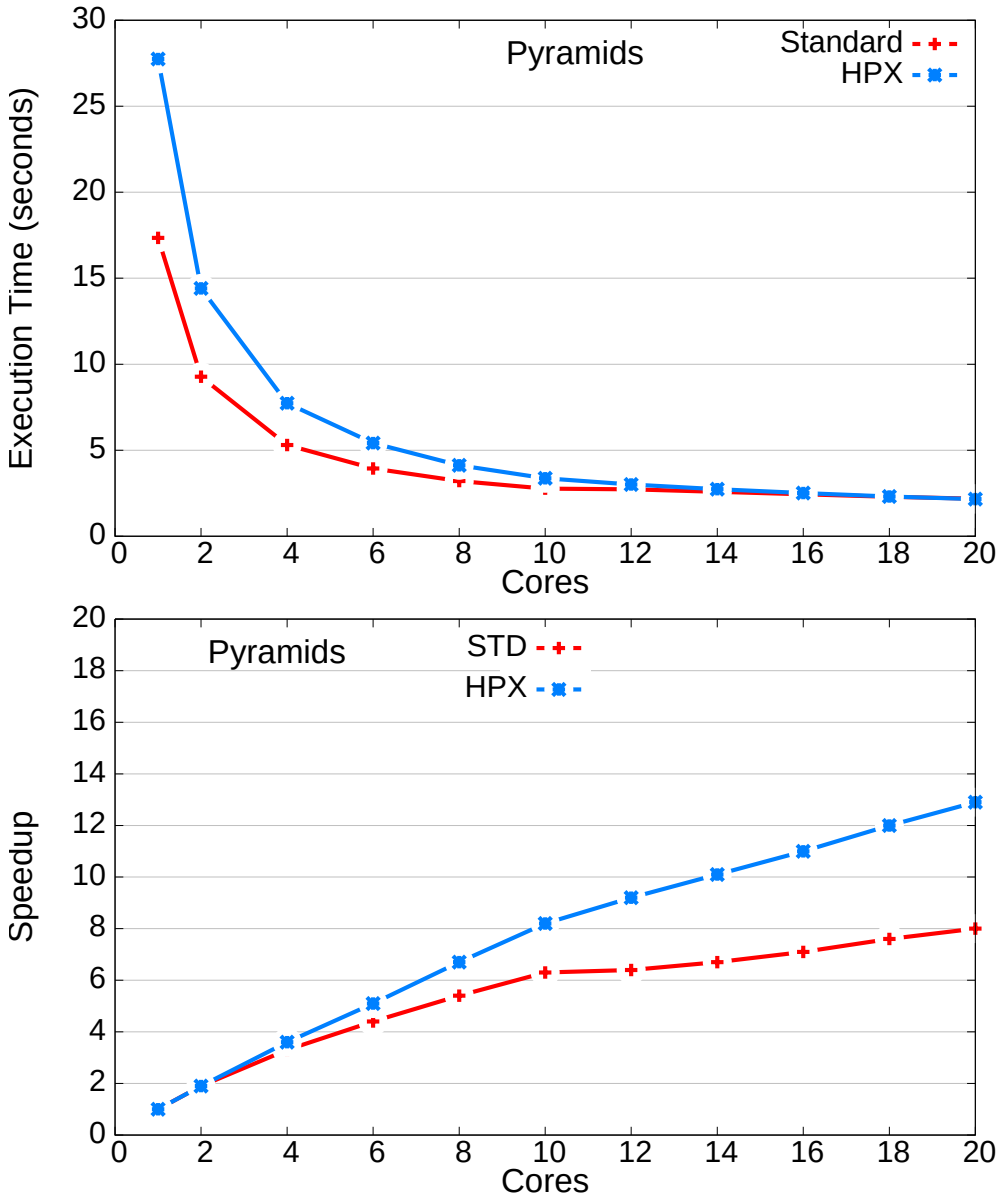

Figure 55: Pyramids: HPX vs. C++11 Standard, ∼250 µs medium-grained

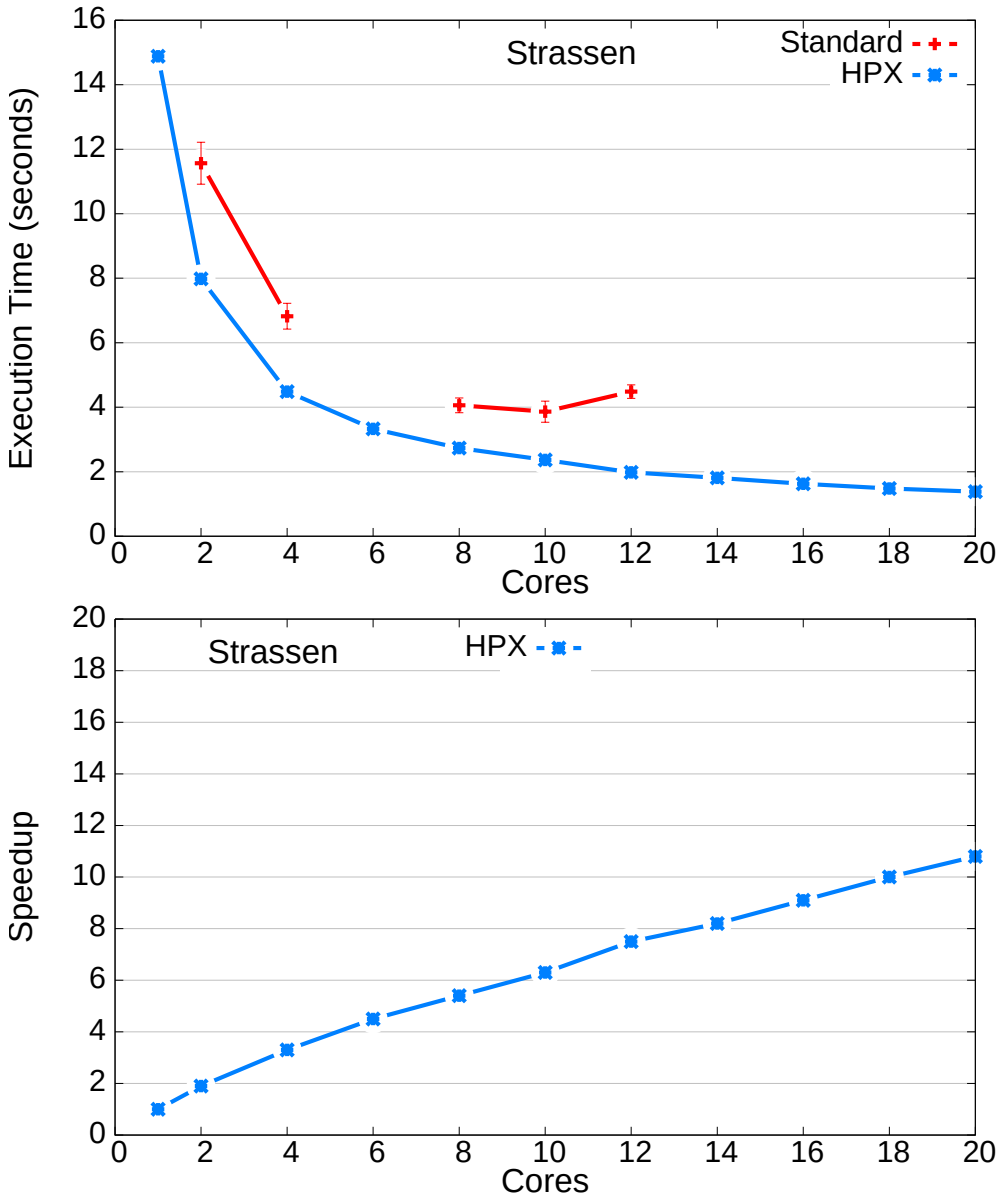

Figure 56: Strassen: HPX vs. C++11 Standard, ∼100 µs fine-grained

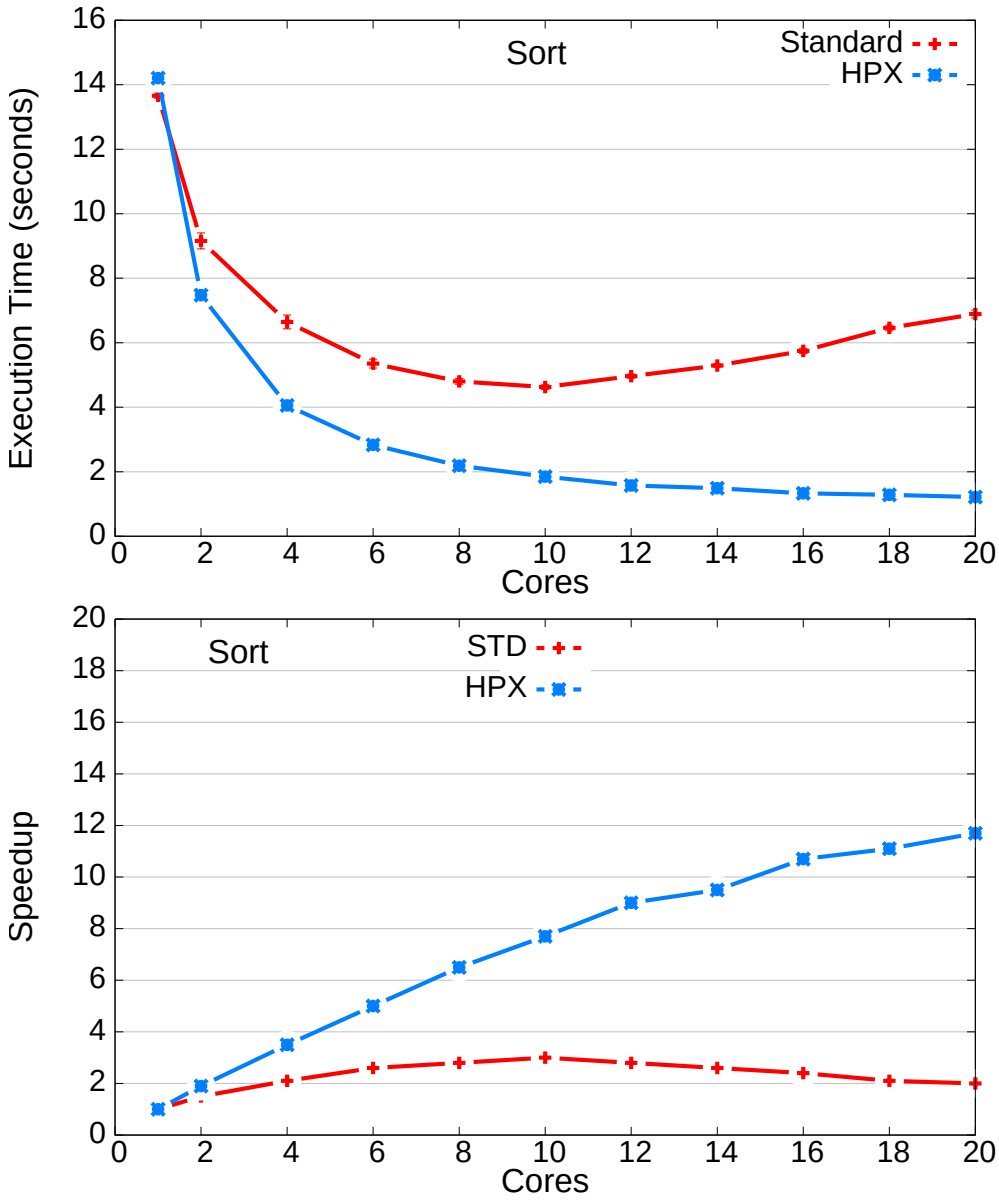

Figure 57: Sort: HPX vs. C++11 Standard, ∼50 µs fine-grained

The remainder of the benchmarks are all classified as very fine-grained since they have task sizes less than ∼5 µs. For HPX, we observe measurements of the task overhead performance counter from 0.5 µs to 1 µs for these benchmarks, so scheduling overheads are a significant portion of the execution time. The standard versions of NQueens, Health, Intersim, Fib and UTS all fail. For these we observe 80,000 to 97,000 pthreads launched by `std::async` just before failure. The stack size required per pthread is 16 KB (minimum) resulting in at least a total of 1.2 GB. The system is not able to manage these quantities of pthreads. The C++11 Standard versions of very fine-grained benchmarks that do manage to complete, FFT and Floorplan, scale poorly or not at all and have execution times much greater than the HPX versions. FFT and UTS, Figures 58 and 59, illustrate these behaviors. Context switching for the C++11 Standard version takes a kernel transition and costs on the order of micro seconds [42], while we measure the cost of HPX context switches in the tens of nano seconds.

Figures 60 - 64 illustrate the capability the performance monitoring system provides to determine factors affecting the performance of the application. The metrics and associated counters are described in Section 6.2.3.

To illustrate ideal strong scaling, we plot T, execution time, and IT, ideal execution time from Eq. 13,

$$IT_{\mathrm{N}} = \frac{T_1}{N} \tag{13}$$

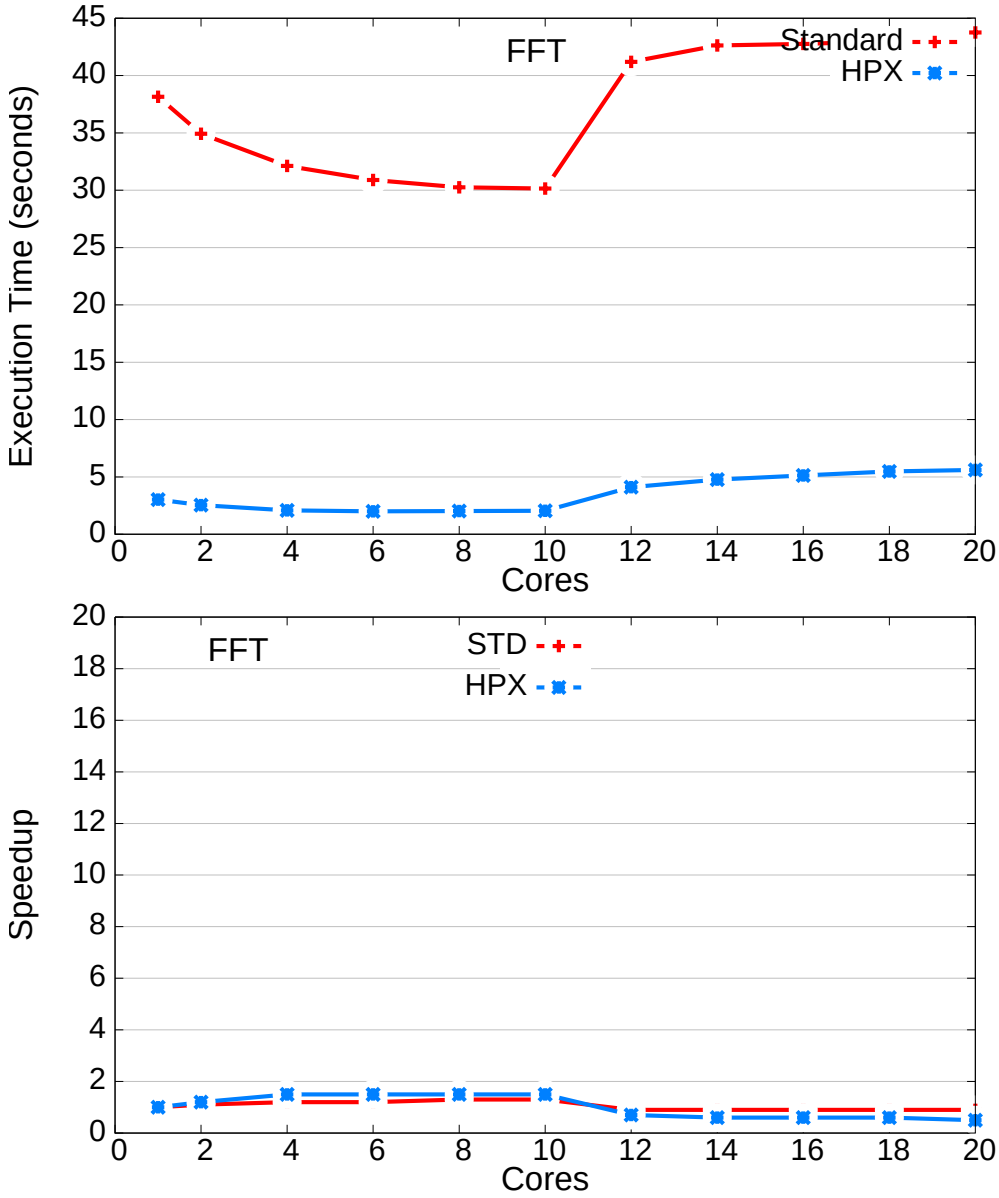

Figure 58: FFT: HPX vs. C++11 Standard, ∼1 µs very fine-grained

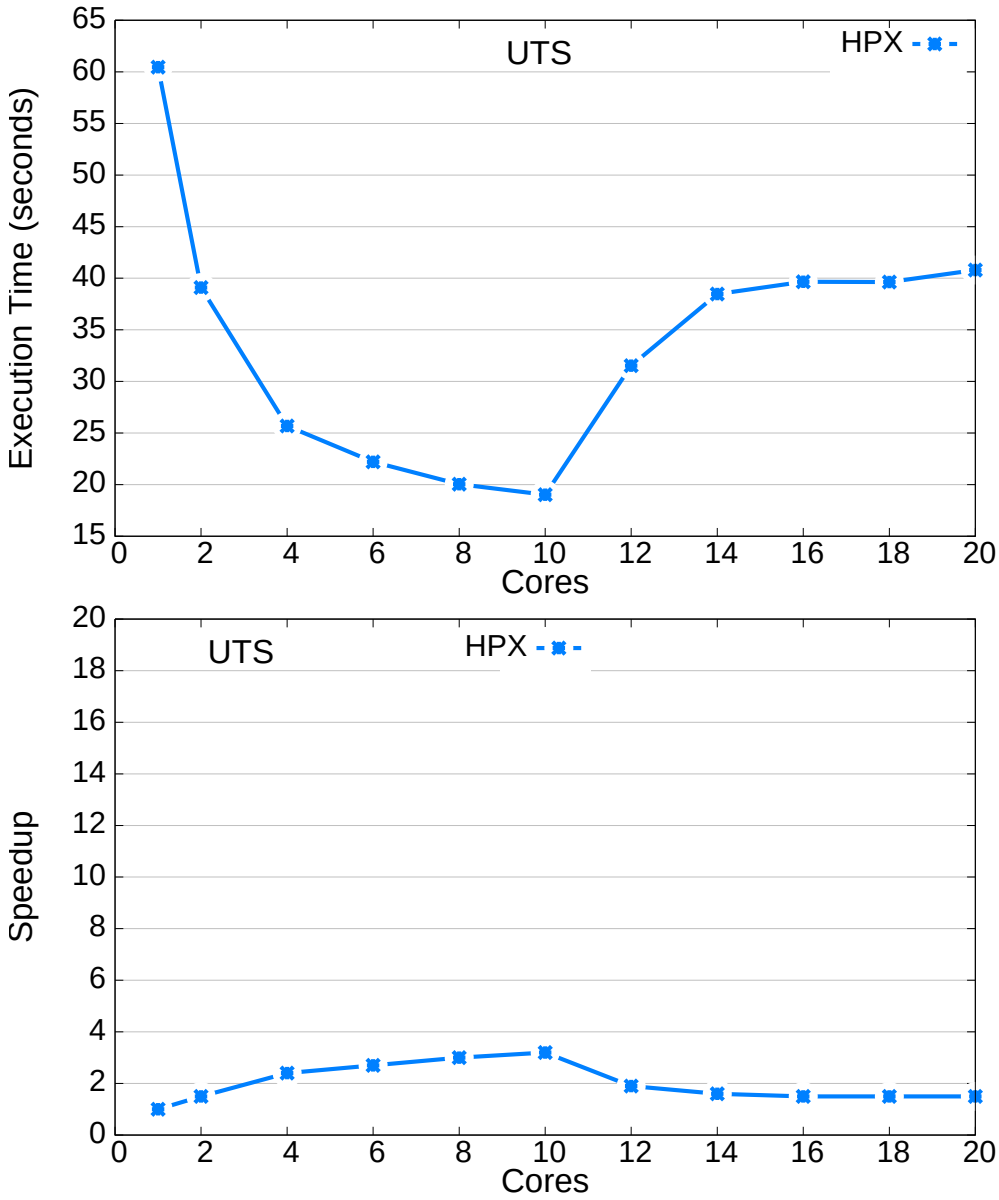

Figure 59: UTS: HPX (C++11 Standard fails), ∼1 µs very fine-grained

where N = the number of cores, IT_N is ideal time for N cores, and T_1 is measured execution time for 1 core. On the same graphs are the measured components: scheduling overheads and task time per core; also included is ideal task time. Ideal task time is the task time we expect if it scaled perfectly with added concurrency. Ideal times for execution time and task time per core are included to visualize the difference between ideal and measured as concurrency (number of cores) is increased.

When the scheduling overheads are low, Figures 60, 61, and 62, as is the case for the fine- to coarse-grained benchmarks, the overall execution time of the benchmark is composed almost totally of the time spent in actual execution of the tasks. For Strassen the overheads are slightly larger than Alignment and this shows that the execution time does not scale as close to the ideal time.

The scheduling overheads have a larger effect on the overall execution time for applications that have smaller granularity. This is further demonstrated with the measurements from the very fine-grained benchmarks, Figures 63 and 64. The combination of smaller task size and larger number of tasks executed per second also puts pressure on system resources causing the task time to increase. The effects are unique to each benchmark and underlying architecture. For UTS, Fig. 64, the scheduling overhead is not as large as the increase in execution time caused by overheads due to contention, cache misses, non-uniform memory latencies, memory interconnect, cache coherency and/or memory bandwidth saturation. The

opposite is true for FFT, Fig. 63. Increasing the number of cores that the benchmark is executed on, also increases resource overheads as seen by the growth of the gap between task execution time and its ideal. For both FFT and UTS, the jump in execution time from 10 cores to 12 cores is caused by crossing the socket boundary and thus the NUMA domain of the system.

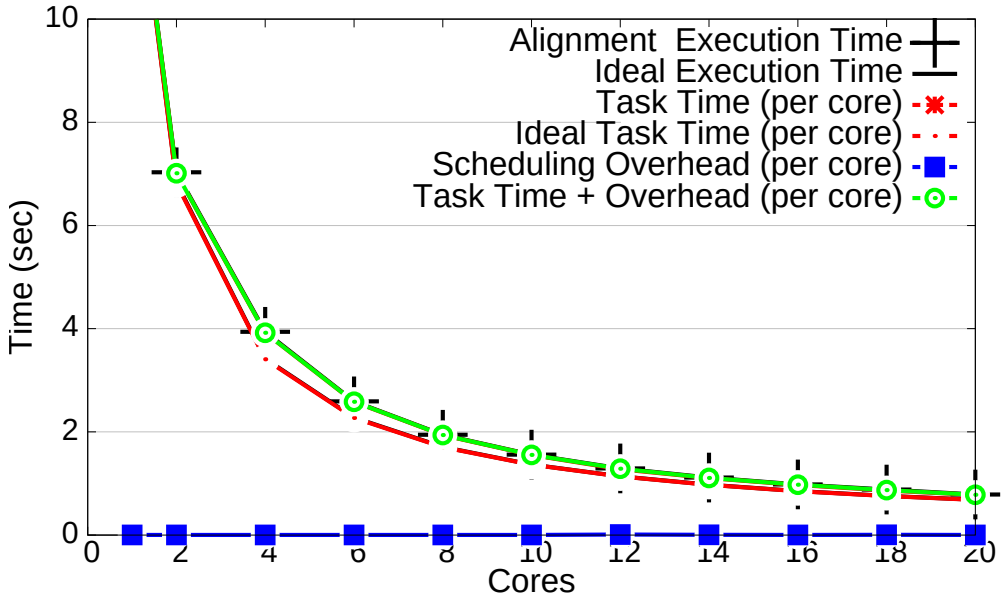

Figure 60: Alignment: Overheads, coarse-grained task size, very small scheduling overhead and the task time is close to ideal so has good scaling behavior with speedup of 17 for 20 cores.

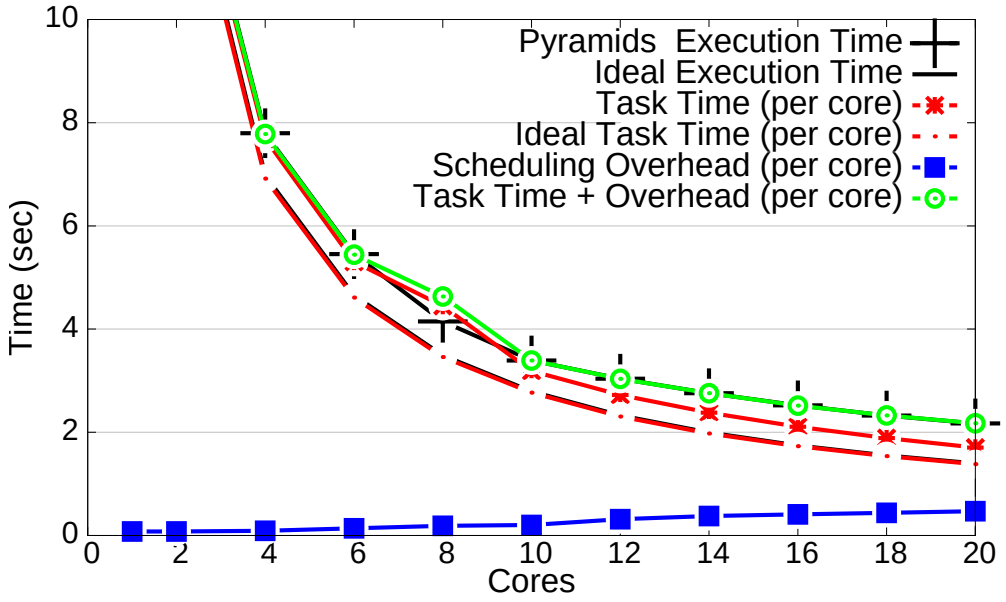

Figure 61: Pyramids: Overheads , medium-grained task size, has slightly larger scheduling overheads than Alignment and the task time is larger than ideal. Speedup for 20 cores is 13.

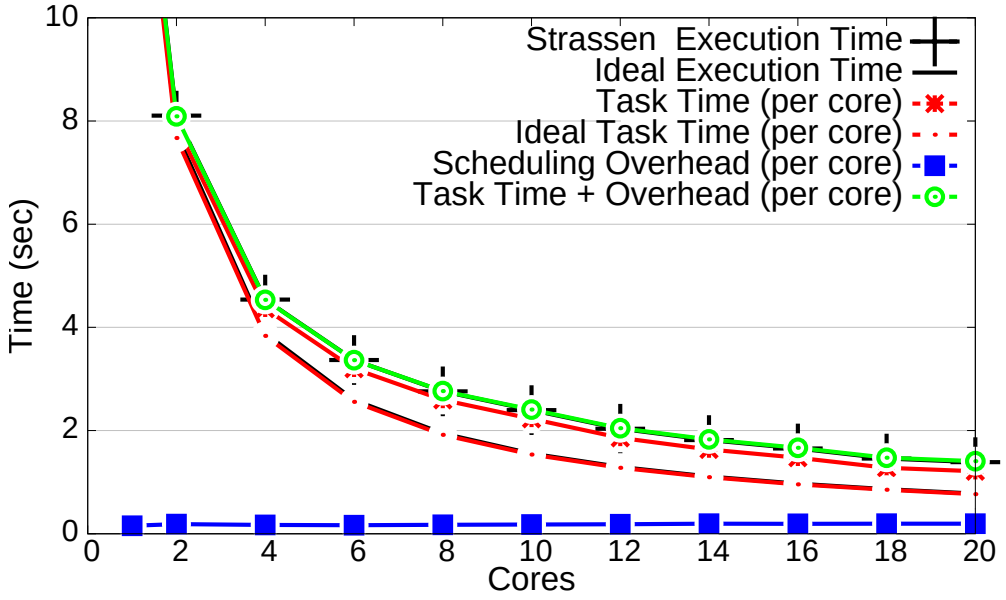

Figure 62: Strassen: Overheads, with fine-grained task size, small scheduling overheads, but the gap between the ideal and actual task time is larger than for Pyramids, and the resulting speedup is 11 for 20 cores.

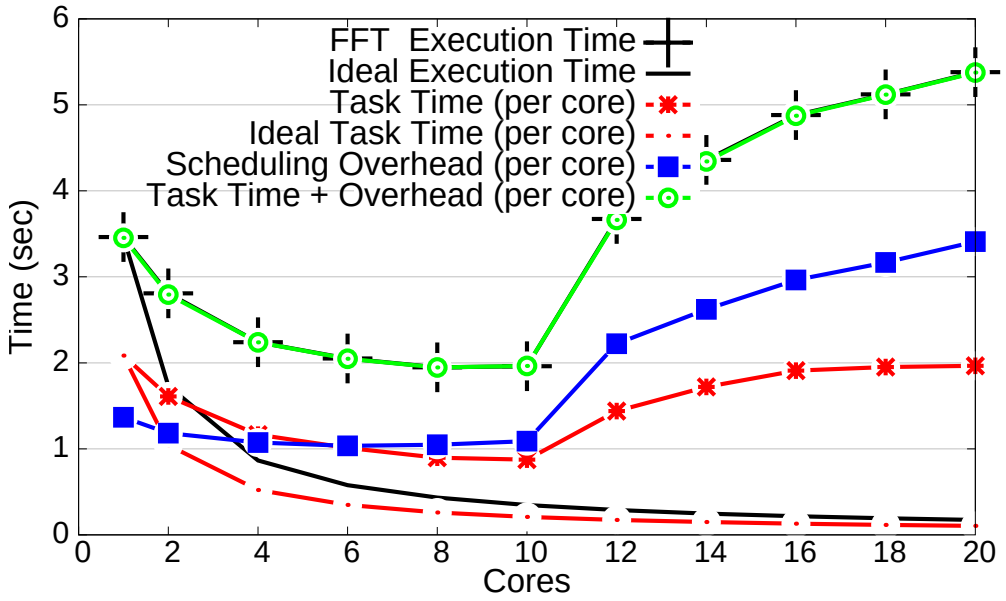

Figure 63: FFT: Overheads, very fine-grained and has scheduling overheads equivalent to the task time and both increase significantly beyond the socket boundary. This results in poor scaling and limits scaling to one socket.

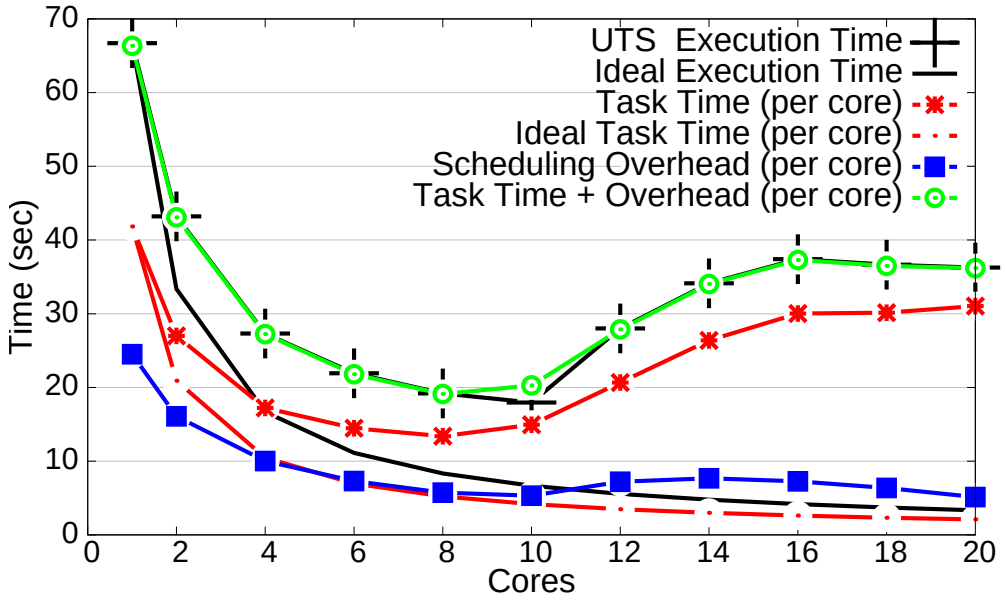

Figure 64: UTS: Overheads, very fine-grained, scheduling overheads are approximately 50% of the task time, after 4 cores task time is larger than ideal and increases after the socket boundary, resulting in poor scaling and increased execution time past the socket boundary.

Hardware counters can be utilized to monitor performance bottlenecks of the underlying system. One example is offcore bandwidth utilization, a metric derived from hardware counters (Section 6.2.3). Offcore bandwidth utilization is compared to speedup in Figures 65 - 68. For very fine-grained benchmarks, like FFT (Figure 68), bandwidth utilization increases with the number of cores used only to the socket boundary then it drops dramatically and speedup decreases significantly beyond the socket boundary. Figure 68 illustrates this for FFT a representation of the benchmarks with very fine-grained tasks. However, bandwidth utilization continues to increase for the benchmarks that scale to 20 cores.

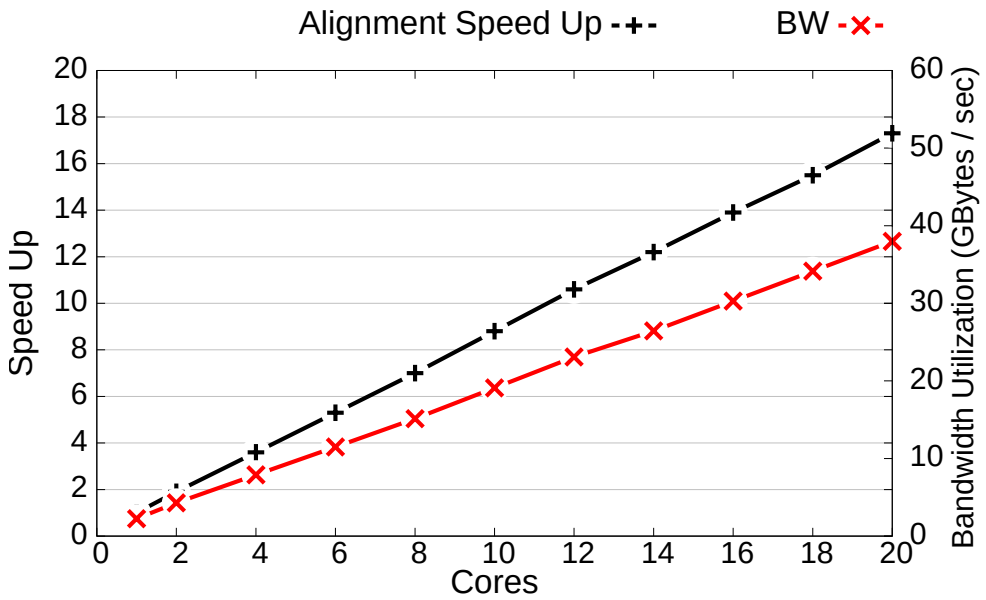

Figure 65: Alignment: Offcore Bandwidth Utilization, coarse-grained tasks

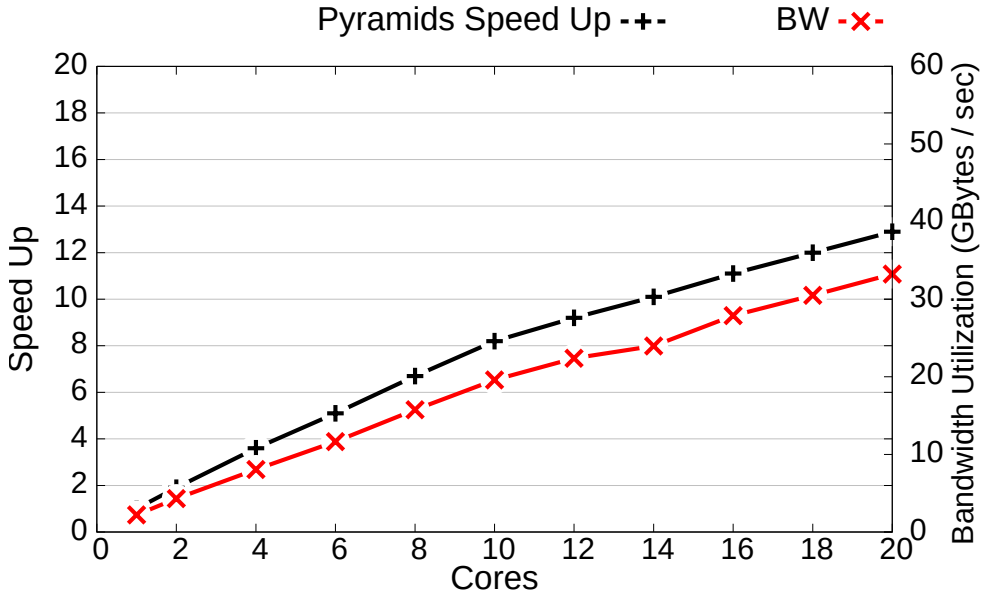

Figure 66: Pyramids: Offcore Bandwidth Utilization, medium-grained tasks

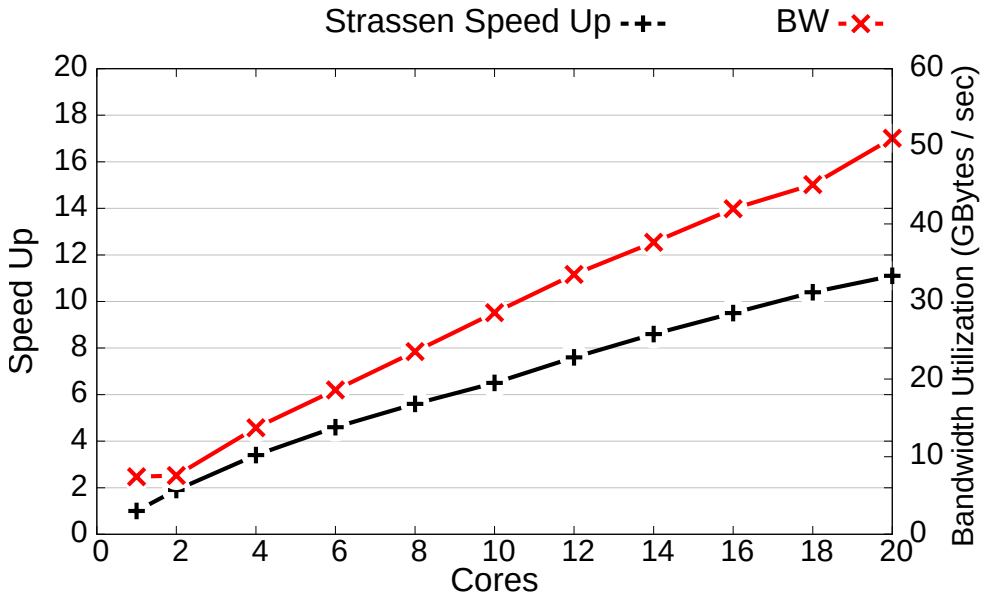

Figure 67: Strassen: Offcore Bandwidth Utilization, fine-grained tasks

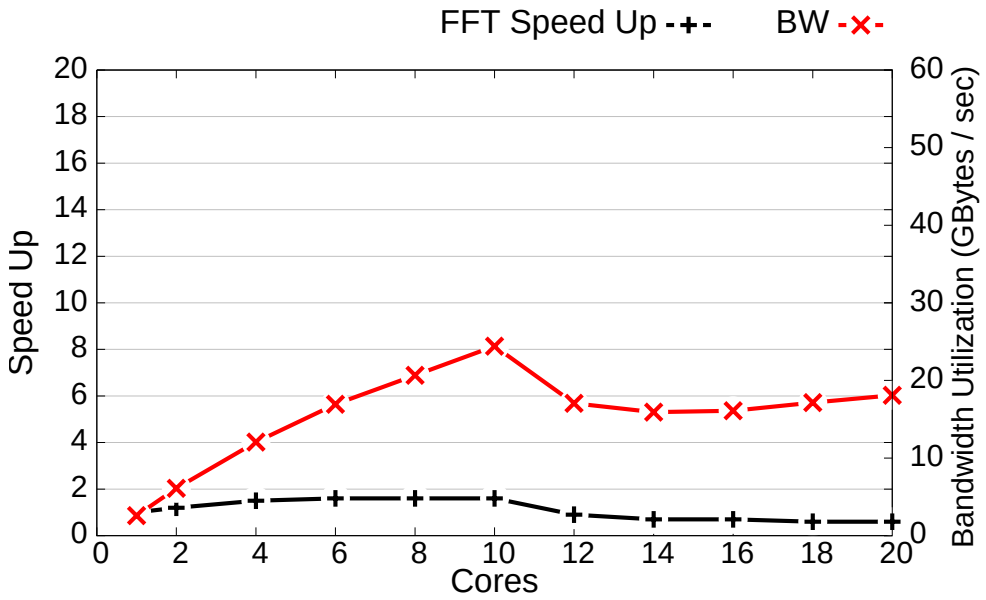

Figure 68: FFT: Offcore Bandwidth Utilization, very fine-grained tasks

6.5 Summary of Performance Counter Experiments

We demonstrate the capabilities of the performance monitoring framework in the HPX runtime system to measure intrinsic events that give detailed insight into the behavior of the application and the runtime system. Because the performance monitoring capability is implemented within the HPX runtime, the reduced measurement overheads enable us to collect performance data that provides an understanding of application efficiency and resource usage during execution.

We show the ease of porting the INNCABS parallel benchmarks to HPX and the resultant performance improvement of the benchmarks. All HPX benchmarks with task durations greater than 25 µs scale to 20 cores, although for the C++11 Standard versions only the benchmarks with task durations greater than 240 µs scale to 20 cores. The strong scaling speedup is from 9.5 to 24.5% higher for HPX than the C++11 Standard versions, except for Round with a very coarse grain size of 10 ms. The C++11 Standard version of Round scales 6.5% higher than the HPX version, but the execution times are equivalent on 20 cores. Thus HPX performs better than C++11 Standard for fine- to medium-grained benchmarks and as well as the C++11 Standard versions for coarse-grained benchmarks.

We demonstrate that current external tools are not capable of supporting C++ task parallel constructs and do not support the runtime performance monitoring that is necessary for adaptive runtime decisions. Experiments performed to assess

performance of the C++11 Standard INNCABS benchmarks using two of the most common available tools, HPCToolkit and TAU, either abort or have overheads that are 35 to 125 times the execution time without the tool. In contrast, the HPX performance monitoring framework gives the ability to monitor intrinsic events during execution and incurs overheads less than one percent when running on more than two cores. Measurements of task time and task management overheads give insight to the performance of the benchmark and can be queried for any interval of an application during execution to use for runtime adaption. Resource usage is also available by introspection of hardware counters as is demonstrated through querying and using offcore counters to estimate bandwidth utilization.

The capabilities of HPX presented pave a path toward runtime adaptivity. The results in this chapter indicate that measuring *task time* and *scheduling overheads* during execution and tuning task grain to minimize these metrics can improve performance of the benchmarks.

As previously mentioned, the APEX library extends HPX functionality [33]. APEX includes a *Policy Engine* that executes performance analysis functions to enforce *policy rules*. By including guided search and auto-tuning libraries in the analysis functions, APEX has already demonstrated an emerging capability for runtime adaptation in HPX applications using its performance monitoring framework. In Chapter 7 we explore the capabilities of the policy engine in the APEX library to demonstrate using performance counters for runtime adaptivity.

7 ADAPTIVE METHODOLOGIES

In this chapter we explore adaptive techniques using the performance monitoring framework of the HPX runtime system with the integration of the Autonomic Performance Environment for Exascale (APEX) library, as illustrated in Figure 69. The integration of APEX enables HPX to pass performance events to APEX for analysis that in turn provides an interface to external performance monitoring as well as feedback for control of the application.

The two main components of APEX are the performance introspection component and the policy engine. APEX implements runtime performance observation facilities with event listeners for both post-mortem analysis and real time adaptation. The introspection component collects information, such as HPX counters, from the application or runtime system through the use of event APIs. The events are handled either synchronously for immediate action or asynchronously and stored in a queue for background processing while execution is returned to the calling process. Introspection of events from the operating system or hardware can also be collected through periodic sampling run on additional OS threads. One method of procuring system level information such as energy consumption is using Resource Centric Reflection(RCR) [43] through the RCRdameon. APEX has four event listeners: Profiling, Concurrency, Tau and the Policy Engine. The first three are used to collect information for post-mortem processing, and are not

used for the work in this dissertation.

The Policy Engine is the event listener that provides controls to an application or runtime system by executing analysis functions on collected measurements and enforcing specified policy rules. For this study the policy engine applies the Nelder-Mead algorithm, a simplex method for minimization, as the policy rule to minimize a user specified performance counter from HPX. APEX implements the Nelder-Mead algorithm from the Active Harmony library [19].

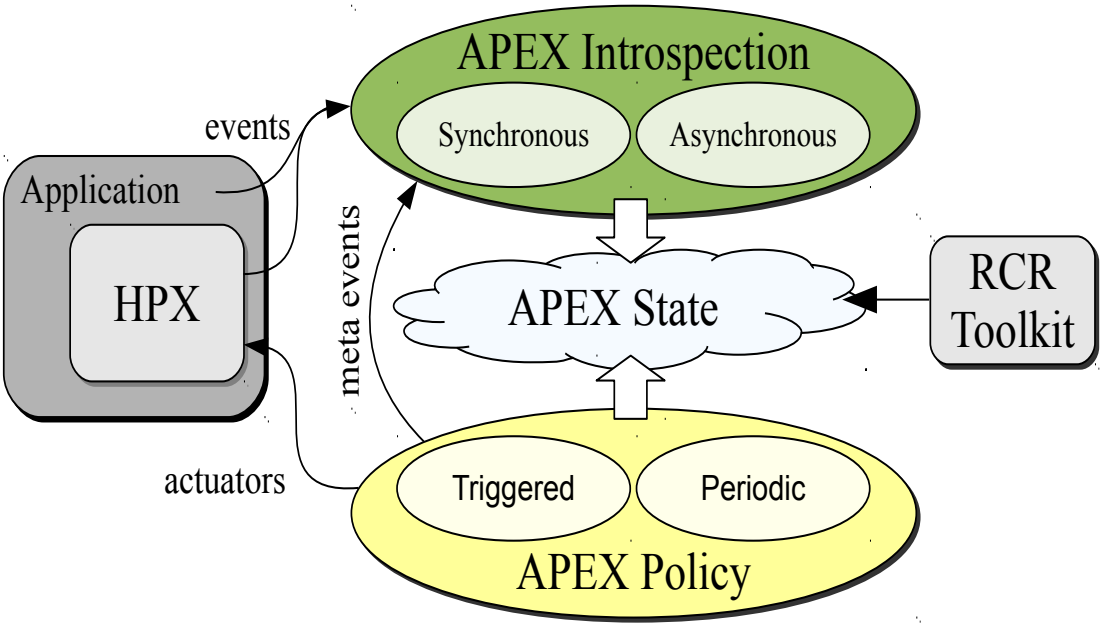

Figure 69: APEX Integration with HPX, from [33]

Runtime adaptation in HPX applications is demonstrated using the APEX prototype for both performance and power optimization by K. Huck, et al. [33]. A scheduler that throttles the number of OS threads is used to optimize either power consumption or performance. In the following section, we demonstrate

tuning task granularity for improved performance through the integration of HPX with APEX.

7.1 Tuning Task Granularity Example

To illustrate tuning task granularity using the integration of APEX with HPX, we use a modified version of the HPX-stencil benchmark (referred to as HPX-repart-stencil and is available in the HPX repository as 1d_stencil_4_repart). Each trial of the experiment calculates the heat problem for 100,000,000 grid points for 50 time steps. The policy engine uses the Nelder-Mead algorithm, a simplex method for minimization, provided by the Active Harmony library [19]. The policy is initialized with a set of partition sizes that range from 1000 points per partition to the largest size that will result in enough partitions to run at least one partition per core (5,000,000 for 20 cores). The user specifies an HPX counter to be used by the benchmark with the goal to minimize the counts. For each trial (50 time steps per trial), APEX reads the counter and reports the count to the policy engine. Based on the new count, Active Harmony refines the search to a smaller set of partition sizes until the search space converges to 5% (default) of the initial search space. Once the search space has converged, the heat calculations are repeated using the partition size chosen by the policy engine for the remaining specified number of trials. For our experiments we specify 100 trials in total.

The experiments are performed on an Intel® node on the Hermione cluster at

the Center for Computation and Technology, Louisiana State University, running Debian GNU/Linux kernel version 3.8.13, using HPX version 0.9.12 and APEX version v0.5-rc0. The node is an Ivy Bridge dual socket system with specifications shown in Table 8. The first set of experiments uses all the cores on one socket

Table 8: Platform Specifications for Performance Counter Experiments

Node	**Ivy Bridge (IB)**
Processors	2 Intel®Xeon®E5-2670 v2
Clock Frequency	2.5 GHz (3.3 turbo)
Microarchitecture	Ivy Bridge (IB)
Hardware Threading	2-way (deactivated)
Cores	20
Cache/Core	32 KB L1(D,I) 256 KB L2
Shared Cache	35 MB
RAM	128 GB

(10 cores), then the experiments are run again using 12 cores by filling the first socket and running two cores on the second socket, and the final set uses all cores on both sockets (20 cores).

Experiments are performed using each of following four HPX counters to be passed to APEX for use by the policy engine:

1. Idle-rate: */threads/idle-rate* - the ratio of time spent on thread management to that of overall execution time.
2. Cumulative: */threads/time/cumulative* - cumulative solve time of HPX tasks.
3. Overhead: */threads/time/cumulative-overhead* - cumulative time spent on task management.
4. Overall: */threads/time/cumulative-overall* - overall execution time spent on all HPX tasks, includes task solve and task management overhead.

7.2 Tuning Task Granularity Results

The results from the experiments using the HPX-repart-stencil benchmark are listed in Table 9. Since the *overall* counter provides the execution time spent on all HPX tasks, it is a measure of total execution time. Using APEX to tune grain size by selecting the partition size of the benchmark based on this count results in minimum execution time. The value of the overall task time converges for partition sizes ranging from 40,000 to 62,500 grid points that have fine-grained task durations from 50 to 74 µs.

When minimizing *idle-rate* the partition sizes selected by APEX run from 250,000 to 500,000 grid points, 286 to 586 µs task durations, and achieve execution times for the benchmark from 12 to 16% higher than the minimum execution time. Minimizing *idle-rate* biases the tuning policy towards choosing larger task grain size resulting in scheduling fewer tasks. Since idle-rate is a ratio of task management overhead to overall time, using it for adaptation does not take into consideration other overheads caused by parallelization, and the adaptive process does not minimize those overheads.

Using the *overhead* counter, a measurement of task management overheads, also results in selection of partition sizes that have medium to coarse-grained task durations, 145 to 745 µs, and execution times from 7 to 16% higher than the minimum.

Table 9: Tuning Task Granularity Results: An example using the integration of HPX and APEX to tune task granularity by automatically repartitioning the 1d stencil benchmark based on a policy to minimize the value from an HPX counter.

Counter	**Partition Size at Convergence** (grid points)	**Execution Time** (secs)	**Trials to Convergence**	**Task Duration** (µs)
10 Cores				
Idle-rate	250,000	2.731	63	286
Overhead	625,000	2.743	49	745
Cumulative	3,125	4.512	29	6
Overall	40,000	2.351	33	50
12 Cores				
Idle-rate	500,000	2.226	43	586
Overhead	160,000	2.246	35	187
Cumulative	6,400	3.560	69	10
Overall	62,500	1.992	33	74
20 Cores				
Idle-rate	500,000	1.754	63	586
Overhead	125,000	1.686	57	145
Cumulative	8,000	3.448	89	13
Overall	50,000	1.571	85	62

Notes: 1. Execution Time - average time of all trials after convergence.
2. Task Duration - average duration of HPX threads from HPX counter */threads/time/average* when running the original HPX stencil benchmark on 1 core for the same partition size.

The *cumulative* counter measures only solve time of the tasks and does not include task management overheads. When using this counter the partition sizes that are selected through the adaptive process are very fine-grained with task durations ranging from 6 to 13 µs. Minimizing the cumulative counter is biased towards minimizing task time, resulting in a bias towards choosing very small partitions. The resulting execution times are approximately twice the minimum

execution time. In Chapter 5 we show that the very fine-grained tasks can produce large task management overheads, that are essentially ignored when selecting the cumulative counter to be minimized.

The results of this study show that to minimize execution time it is best to use the *overall* counter as the input to the policy in the HPX-repart-stencil benchmark. Tuning grain size can improve performance, but in the case of this example the entire benchmark is run from 29 to 89 times in order to select a partition size that gives the best performance, incurring additional overhead from approximately 130 to 300 seconds. Policies can be changed to improve the number of trials it takes to reach convergence. For example, the search space can be shortened to include only partitions for fine to medium task granularity, or a history of previous searches can be used to limit the search space. Also, if the application were to use this type of policy to tune grain size for a limited number of time steps or a small portion of the program and then run with that task granularity for longer portions of the program the overhead will be amortized.

Future plans for using the integration of APEX and HPX to implement adaptive techniques include tuning grain size for use by HPX parallel algorithms, using multi-objective policies such as power consumption, performance and/or networking overheads to control the application. Future plans are discussed in more detail in Chapter 8.

8 CONCLUSIONS AND FUTURE WORK

Results of the characterization studies include:

1. The combination of two metrics, thread management overhead and wait time, has correlations to execution time greater than 0.9 for all task granularity ranges for the one dimensional stencil benchmark on more than 8 cores.

2. Current external tools are not capable of supporting C++ task parallel constructs and do not provide the runtime performance monitoring that is necessary for adaptive decisions during execution. Experiments that assess performance of the C++11 Standard versions of the INNCABS benchmarks using two of the most common available tools, HPCToolkit and TAU, either abort or have overheads that are 35 to 125 times the execution time without the tool.

3. The HPX performance monitoring framework gives the ability to monitor intrinsic events during execution. Additional overheads incurred by monitoring is less than one percent when running on more than two cores.

4. HPX performs better than the C++11 Standard versions of the INNCABS benchmarks for fine- to medium-grained tasks and atleast as well as C++11 Standard versions for benchmarks with coarse-grained tasks. HPX scales

to 20 cores for benchmarks with task granularity greater than 25 µs, while C++11 Standard only scales to 20 cores when task granularities are greater than 240 µs. Strong scaling speedup is from 9.5 to 24.5% higher for HPX than for C++11 Standard for benchmarks with fine- to coarse-grained tasks. For the benchmark with 10 ms tasks (the largest grain size) the C++11 Standard version scaled 6.5% higher than HPX, however the execution times were equivalent on 20 cores.

5. The example of HPX integrated with APEX for tuning task grain size to improve performance uses a policy in APEX to minimize an HPX counter. Four counters were specified as the count to minimize in order to tune task granularity: *cumulative*, *idle-rate*, *overhead*, and *overall* task time. Of these, *overall* resulted in the minimum execution time while minimizing *idle-rate* and *overhead* resulted in approximately 10% - 11% slower times, and *cumulative* resulted in twice the minimum execution time.

6. For the adaptive example, it takes from 29 to 89 trials to converge on the task grain size for best performance, incurring an overhead from approximately 130 to 300 seconds. If the application were to use this type of policy to tune grain size for a limited number of time steps or a small portion of the program and then run with that task granularity for longer portions of the program the overhead will be amortized.

Important contributions of this dissertation are:

1. The illustration of the important role task granularity plays in task-based parallel applications. Application developers should develop applications with a means to adjust task granularity.

2. Aided in guiding the maturation of the thread scheduling and performance monitoring systems of HPX through discovering issues and bugs and steering implementation of improvements.

3. Understanding the overheads produced by parallel programs using asynchronous task-based runtime systems, such as HPX, and that the amortization of these overheads is dependent on task grain size. Tuning grain size can amortize task management and wait time for optimal performance.

4. The implementation of new performance counters and further characterization of overheads for benchmarks with a variety of task granularity, parallel structures, and synchronization.

5. The example, that illustrates dynamic adaptation, tunes grain size to optimize performance by using a policy that is based on minimizing an HPX counter. The example uses the integration of HPX with APEX providing an interface to the HPX performance monitoring framework as well as feedback for control of the benchmark. The conclusion from this example is that the

policy needs to minimize the overall task time counter in order to choose optimal task granularity.

Future plans include:

1. The implementation of tuning grain size in HPX parallel algorithms. The HPX parallel algorithms are the implementation of C++ standards and proposals for parallel programming. The plan is to implement auto chunking in parallel algorithms that apply functions to a given range or set of elements. Implementing auto chunking for parallel algorithms will aim at automatically tuning task grain size based on policies specified in APEX. The example in Chapter 7 tunes task grain size directly in the benchmark through the interface of APEX with HPX. For future applications, we plan to implement tuning task grain size in the HPX parallel algorithms so that a higher level API will be available to the programmer.

2. To utilize adaptive mechanisms for distributed HPX applications. One optimization planned is to automatically determine the number of parcels to coalesce before sending them over the network to optimize networking overheads.

3. Implementing multi-objective optimization policies. Multi-objective policies will allow the application to apply adaptive measures towards more than one

objective, such as optimizing execution time, minimizing power consumption, and mitigating networking overheads.

4. The broad view of future plans includes using the performance framework of HPX with the capabilities of APEX for dynamic adaptation to improve performance, resilience, and energy efficiency for scientific applications on large scale distributed heterogeneous systems.

APPENDICES

A TASK GRANULARITY SUPPLEMENTARY RESULTS

A.1 Task Granularity Results Sandy Bridge

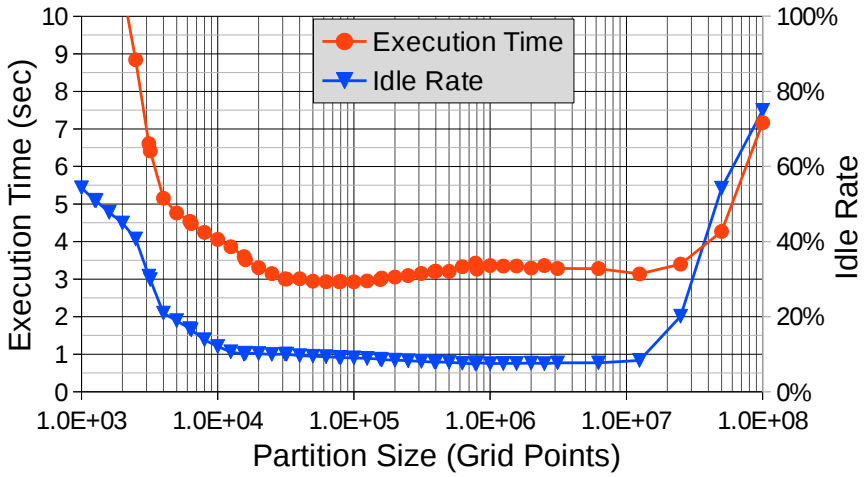

Figure 70: Sandy Bridge (4 cores): Idle-Rate

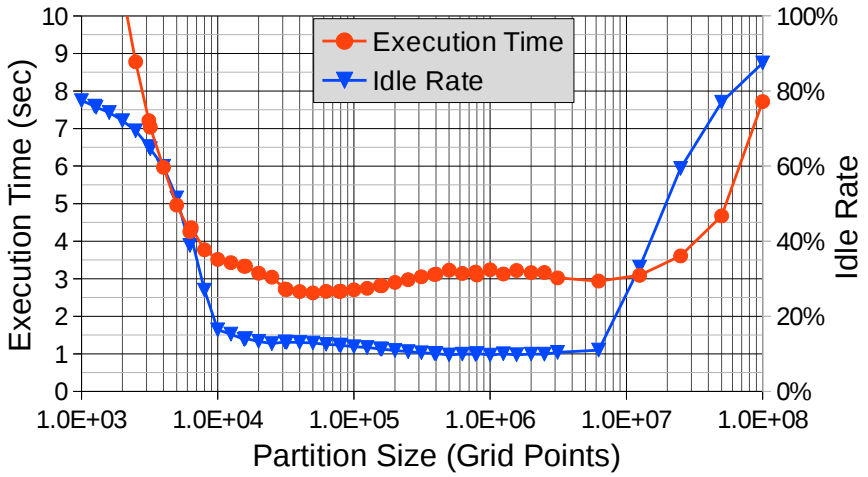

Figure 71: Sandy Bridge (8 cores): Idle-Rate

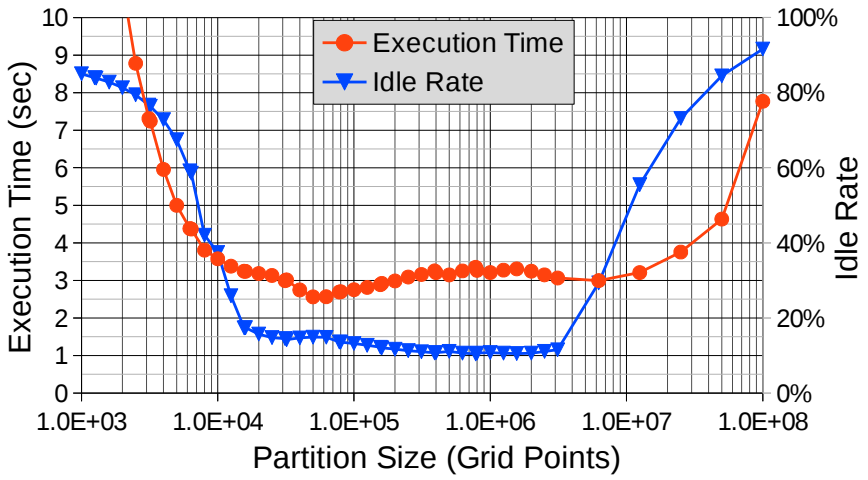

Figure 72: Sandy Bridge (12 cores): Idle-Rate

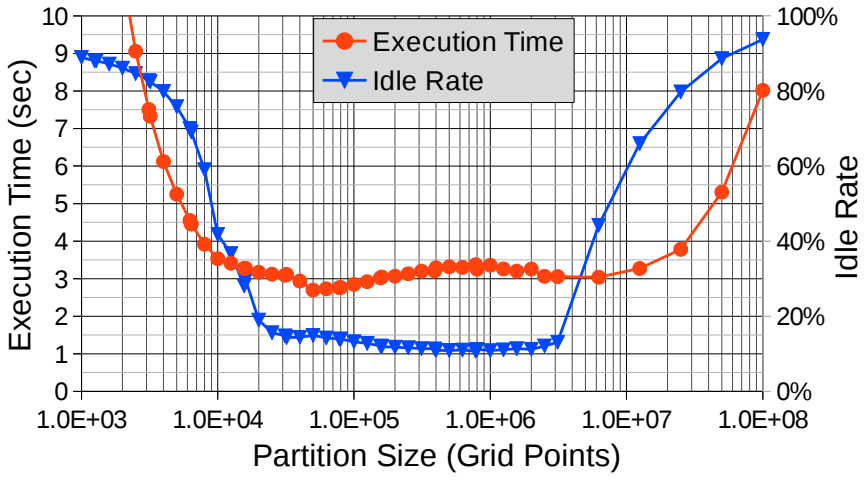

Figure 73: Sandy Bridge (16 cores): Idle-Rate

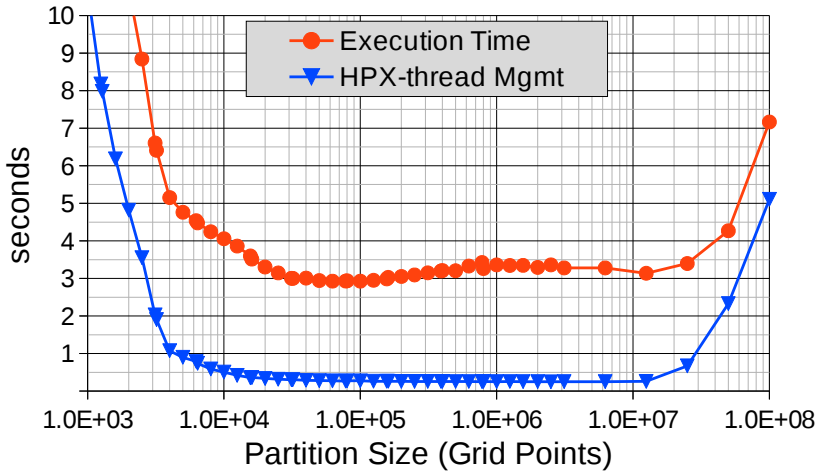

Figure 74: Sandy Bridge (4 cores): HPX Thread Management per Core

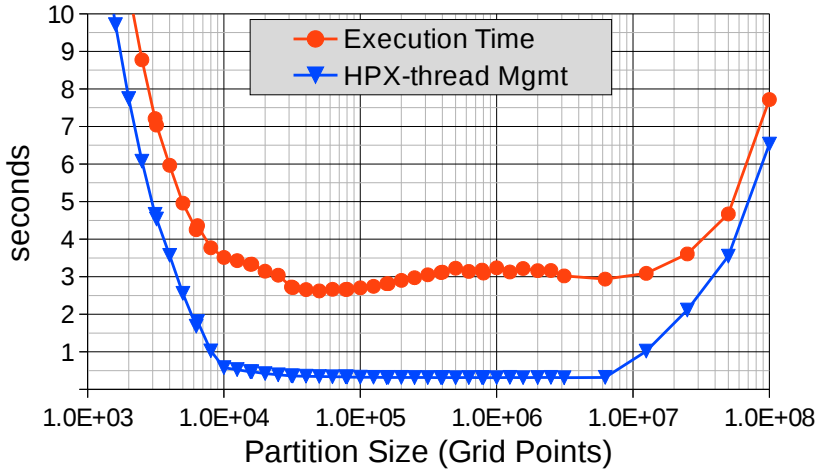

Figure 75: Sandy Bridge (8 cores): HPX Thread Management per Core

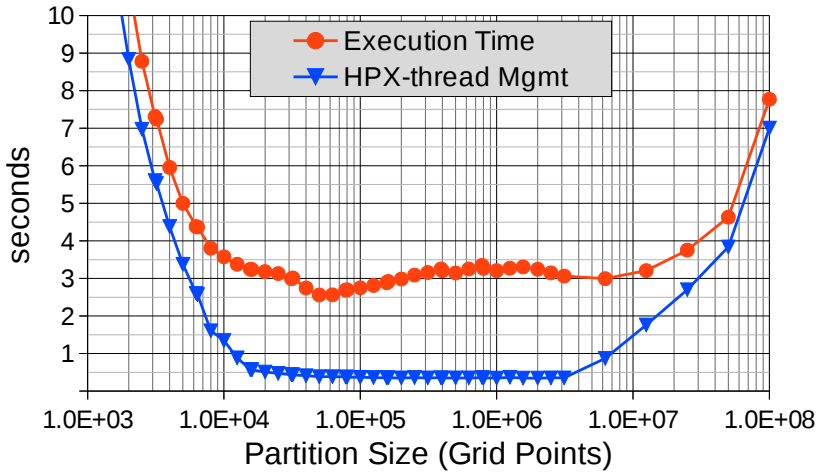

Figure 76: Sandy Bridge (12 cores): HPX Thread Management per Core

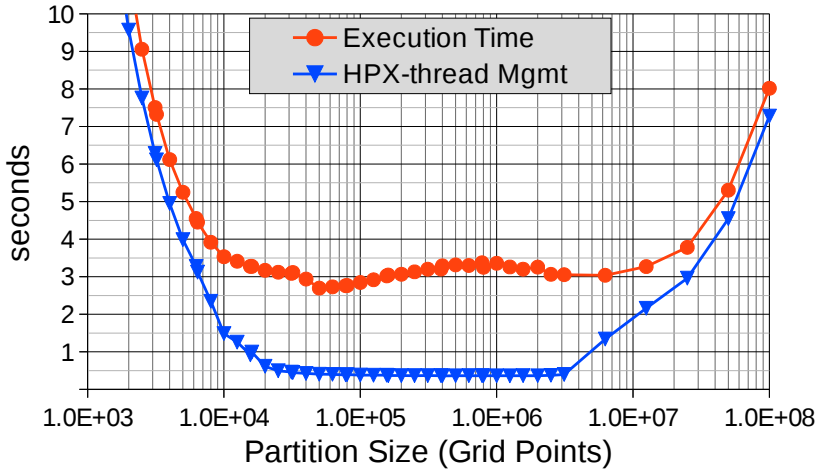

Figure 77: Sandy Bridge (16 cores): HPX Thread Management per Core

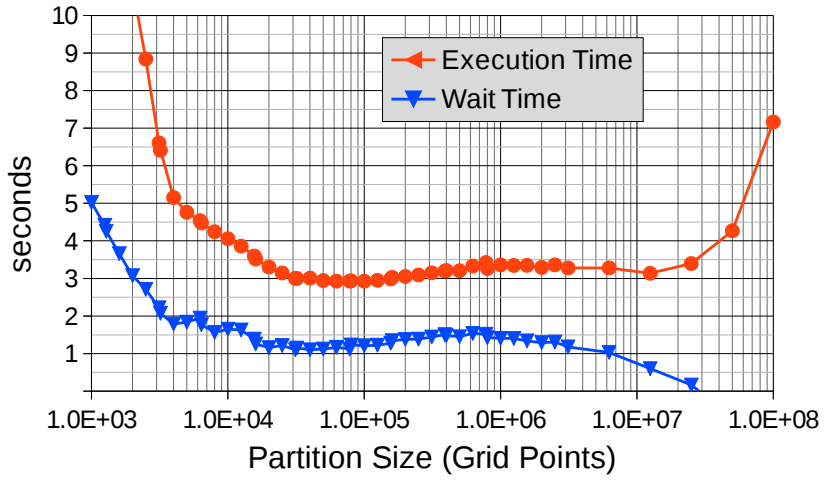

Figure 78: Sandy Bridge (4 cores): Wait Time per Core

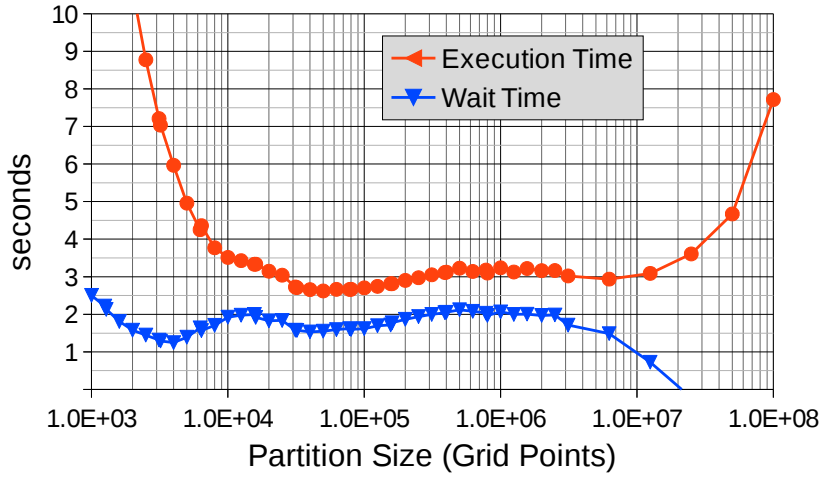

Figure 79: Sandy Bridge (8 cores): Wait Time per Core

Figure 80: Sandy Bridge (12 cores): Wait Time per Core

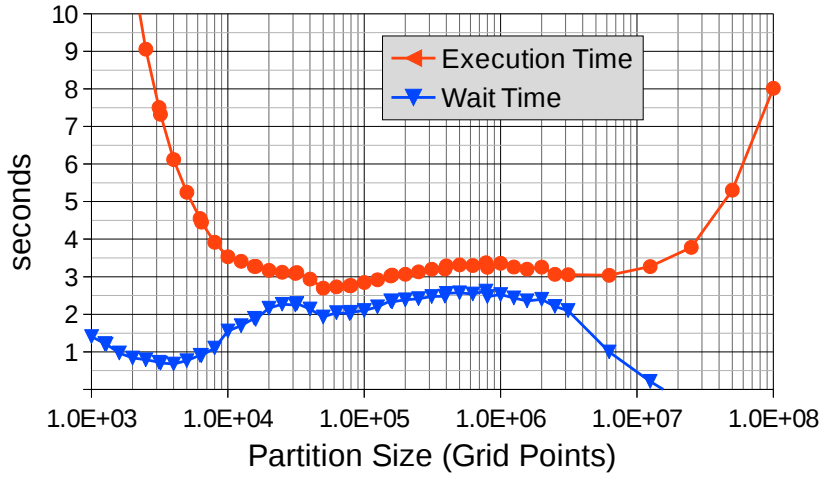

Figure 81: Sandy Bridge (16 cores): Wait Time per Core

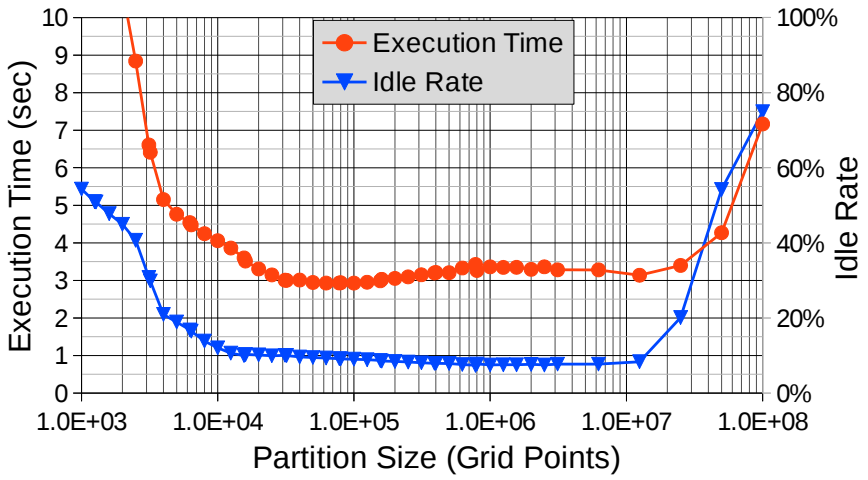

Figure 82: Sandy Bridge (4 cores): Thread Management and Wait Time

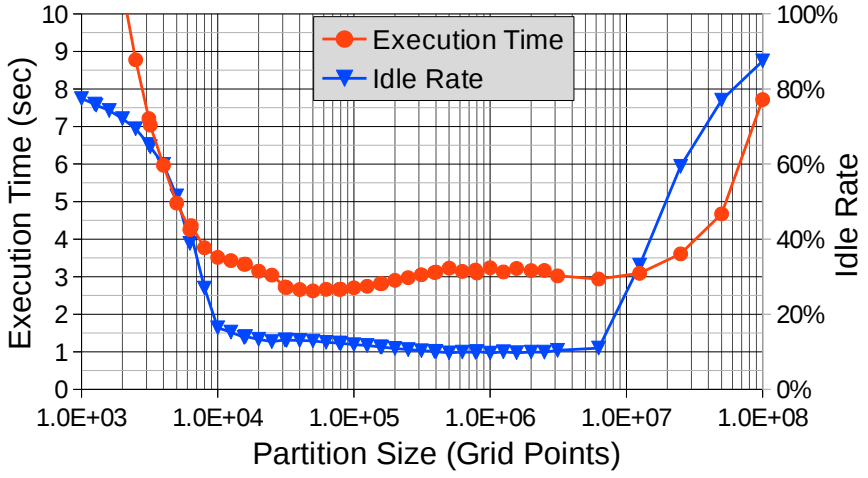

Figure 83: Sandy Bridge (8 cores): Thread Management and Wait Time

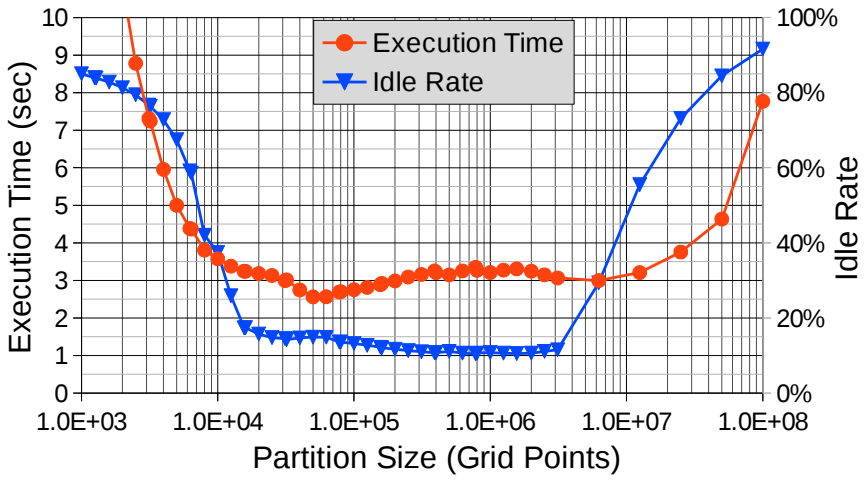

Figure 84: Sandy Bridge (12 cores): Thread Management and Wait Time

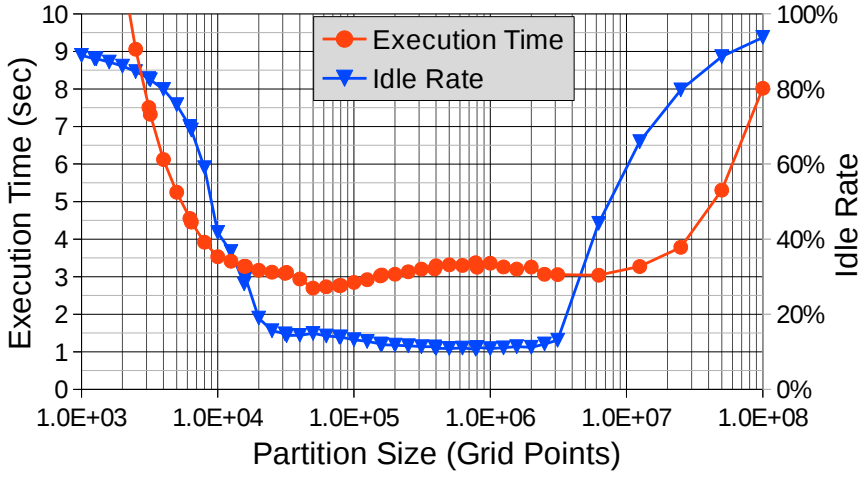

Figure 85: Sandy Bridge (16 cores): Thread Management and Wait Time

A.2 Task Granularity Results Ivy Bridge

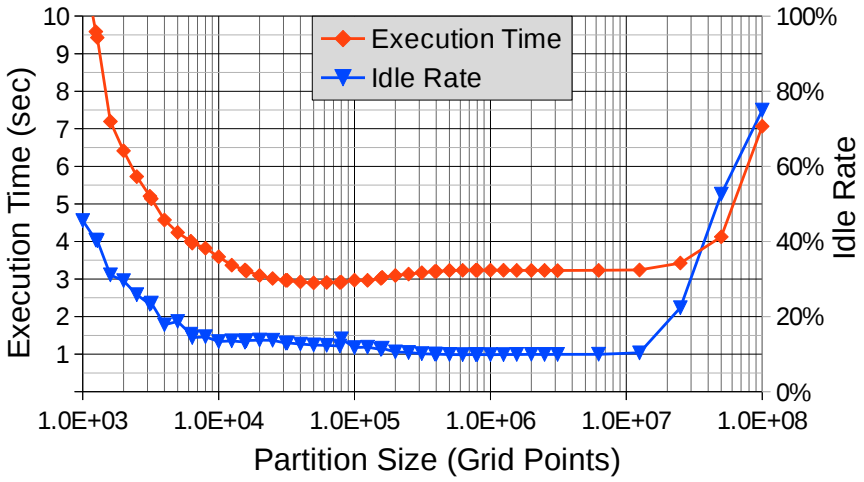

Figure 86: Ivy Bridge (4 cores): Idle-Rate

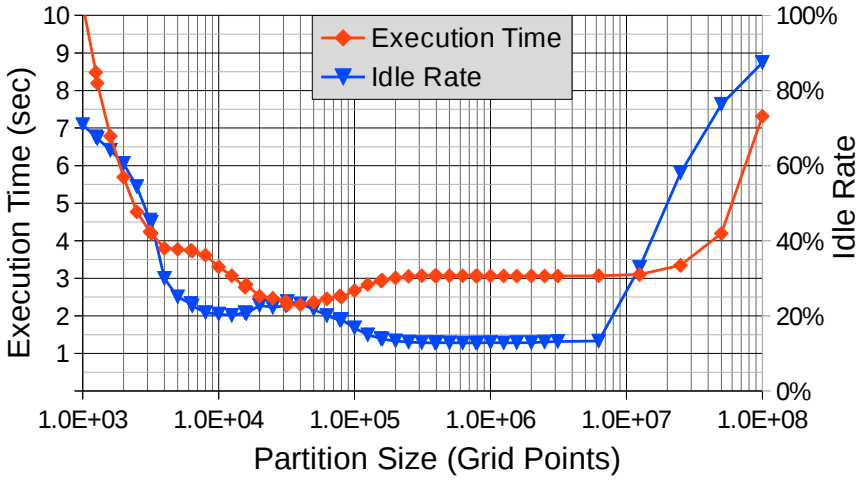

Figure 87: Ivy Bridge (8 cores): Idle-Rate

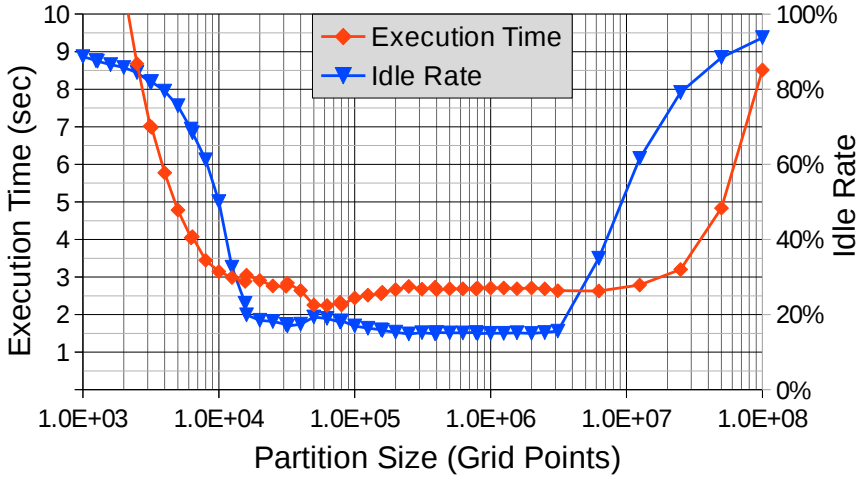

Figure 88: Ivy Bridge (16 cores): Idle-Rate

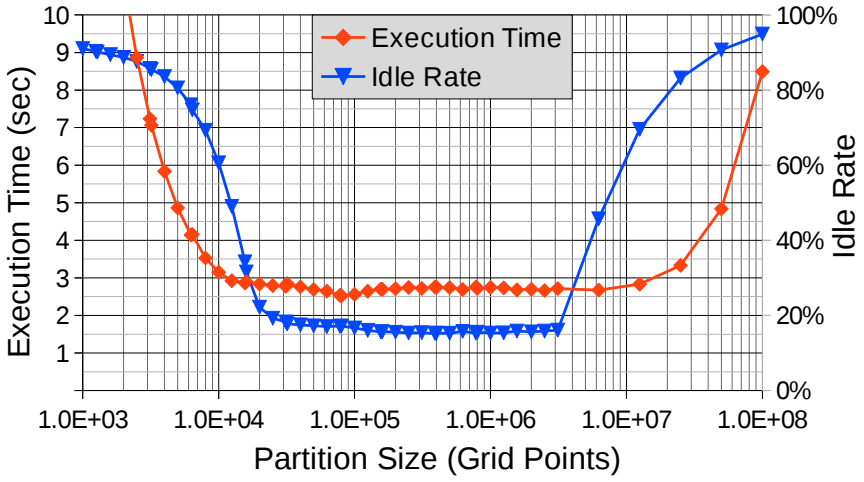

Figure 89: Ivy Bridge (20 cores): Idle-Rate

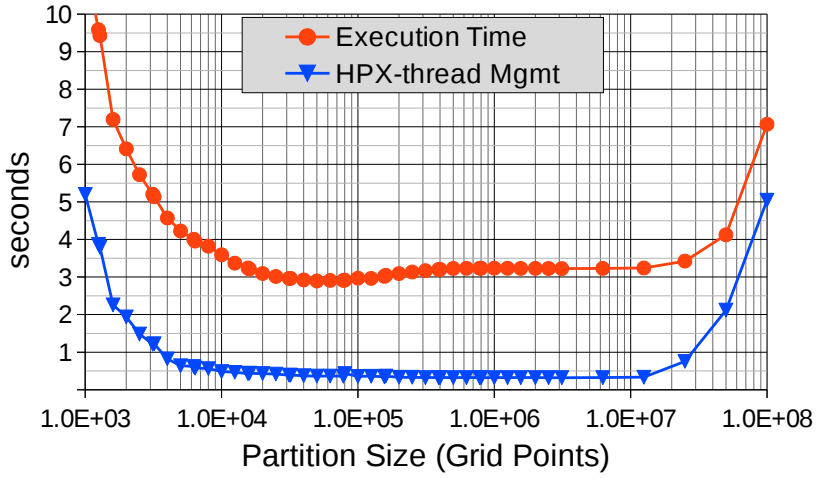

Figure 90: Ivy Bridge (4 cores): HPX Thread Management per Core

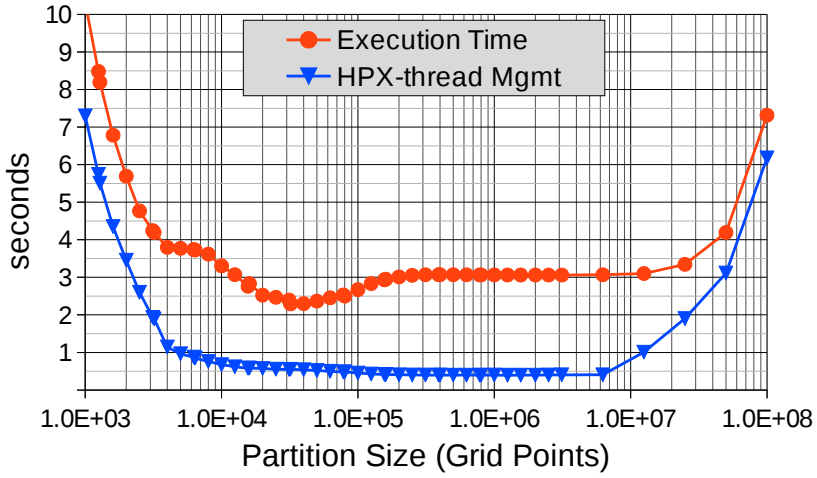

Figure 91: Ivy Bridge (8 cores): HPX Thread Management per Core

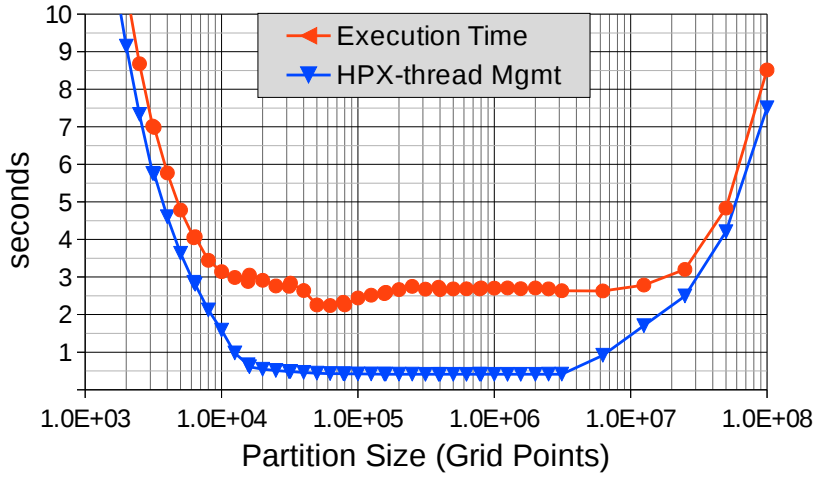

Figure 92: Ivy Bridge (16 cores): HPX Thread Management per Core

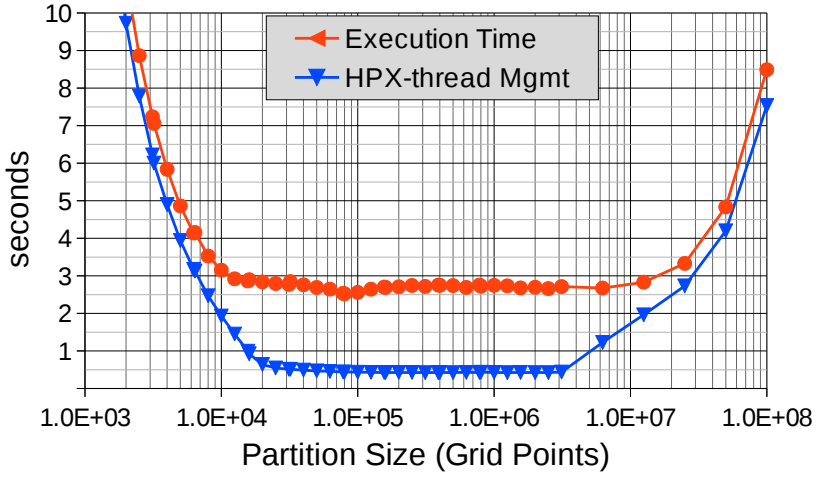

Figure 93: Ivy Bridge (20 cores): HPX Thread Management per Core

Figure 94: Ivy Bridge (4 cores): Wait Time per Core

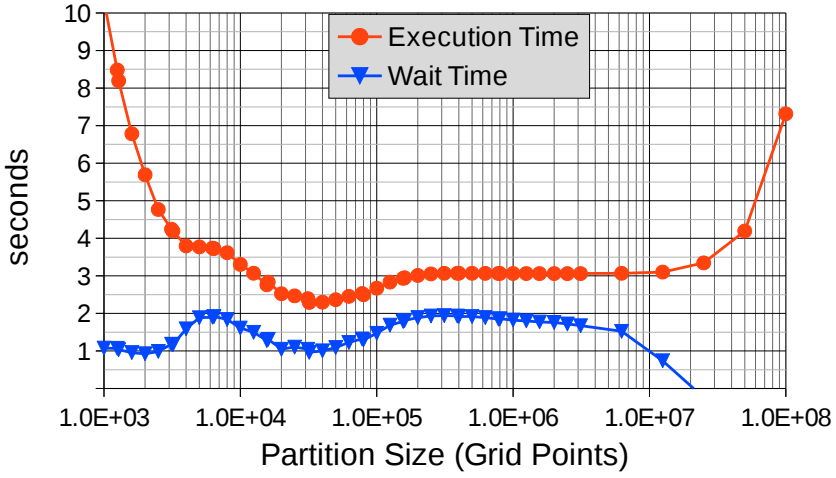

Figure 95: Ivy Bridge (8 cores): Wait Time per Core

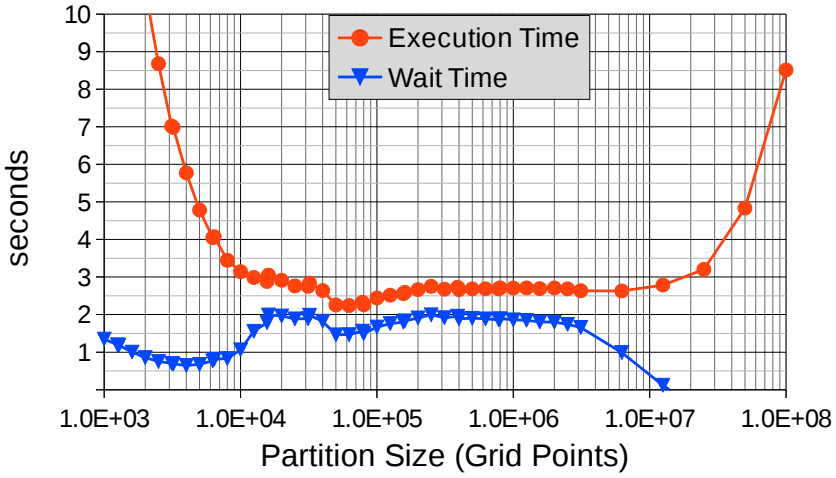

Figure 96: Ivy Bridge (16 cores): Wait Time per Core

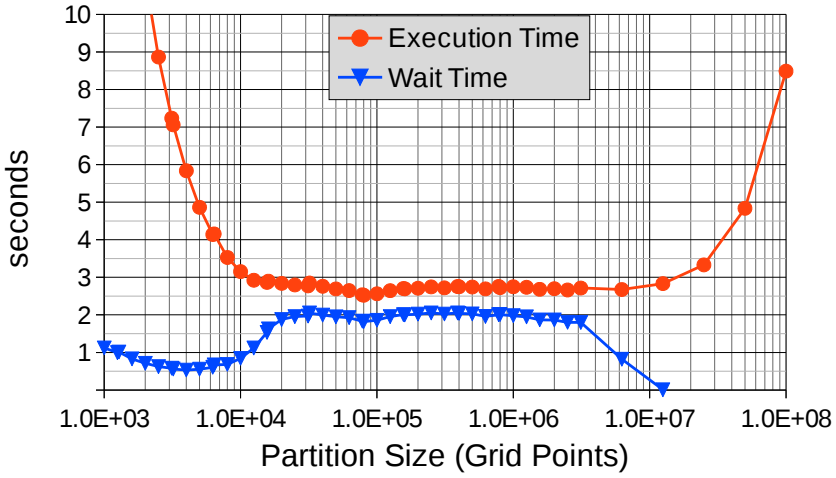

Figure 97: Ivy Bridge (20 cores): Wait Time per Core

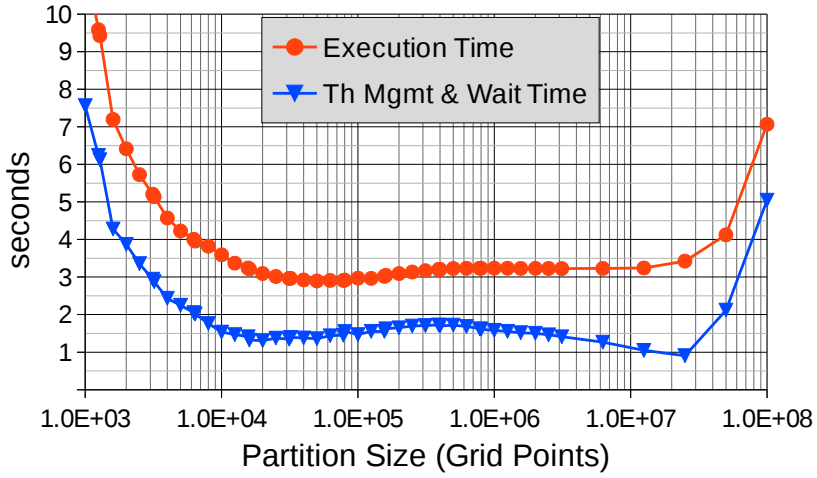

Figure 98: Ivy Bridge (4 cores): Thread Management and Wait Time

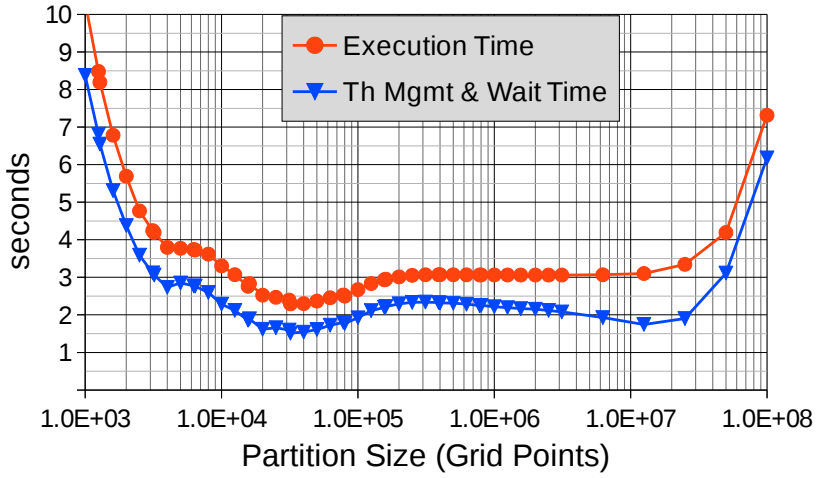

Figure 99: Ivy Bridge (8 cores): Thread Management and Wait Time

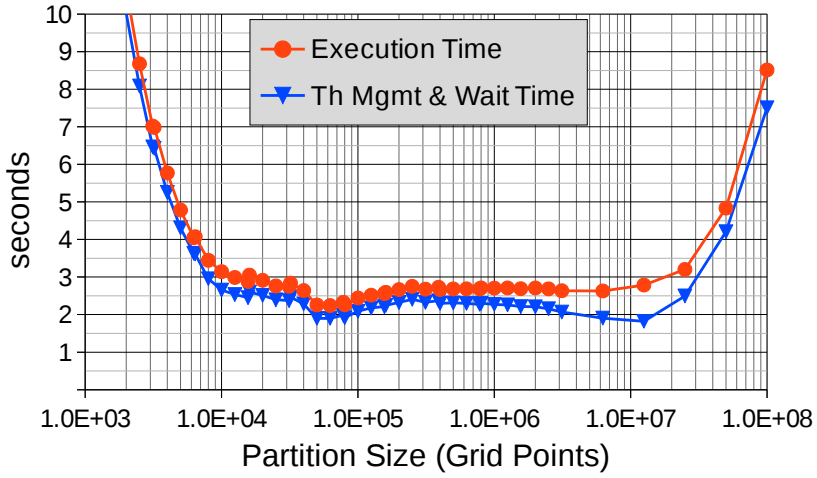

Figure 100: Ivy Bridge (16 cores): Thread Management and Wait Time

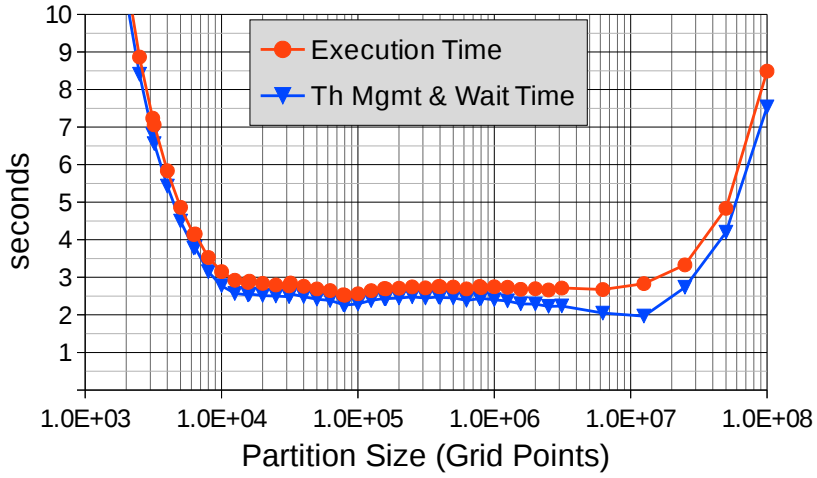

Figure 101: Ivy Bridge (20 cores): Thread Management and Wait Time

B PERFORMANCE ASSESMENT (INNCABS) SUPPLEMENTARY

B.1 HPX vs. C++11 Standard

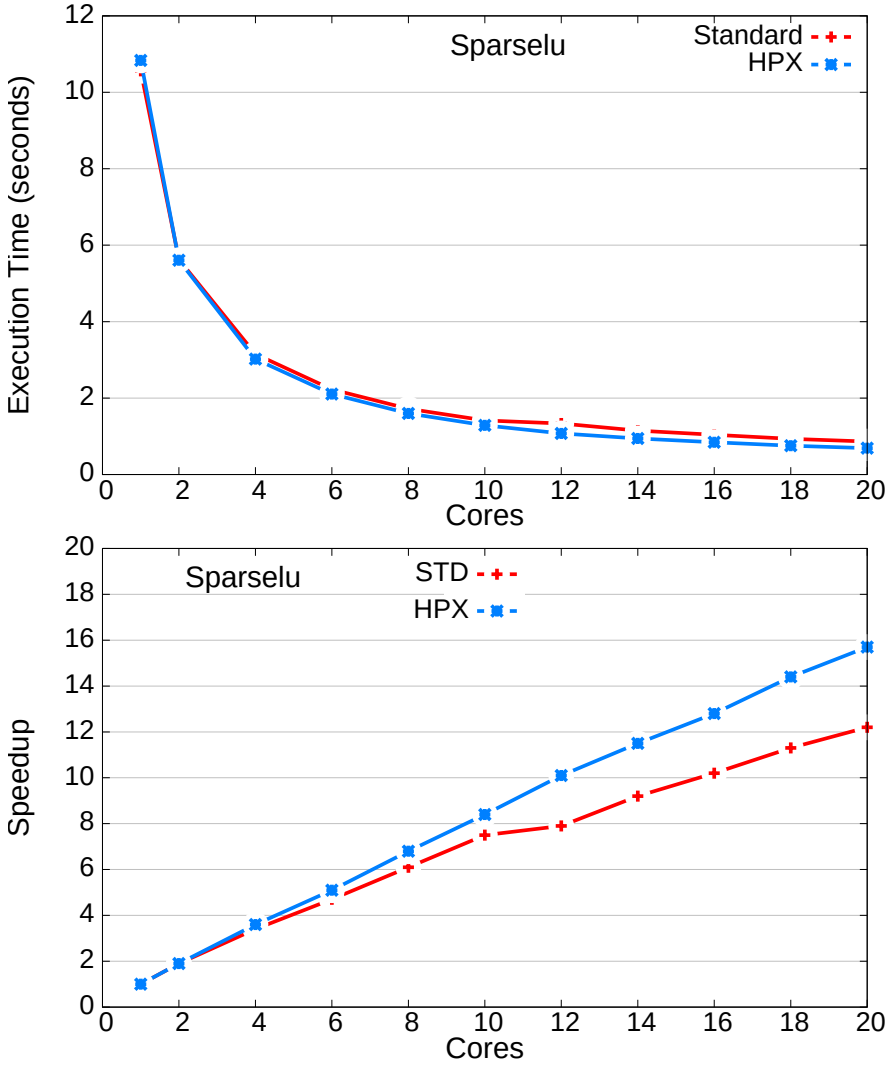

Figure 102: Sparselu: HPX vs. C++11 Standard, grain size ∼1 ms, coarse-grained

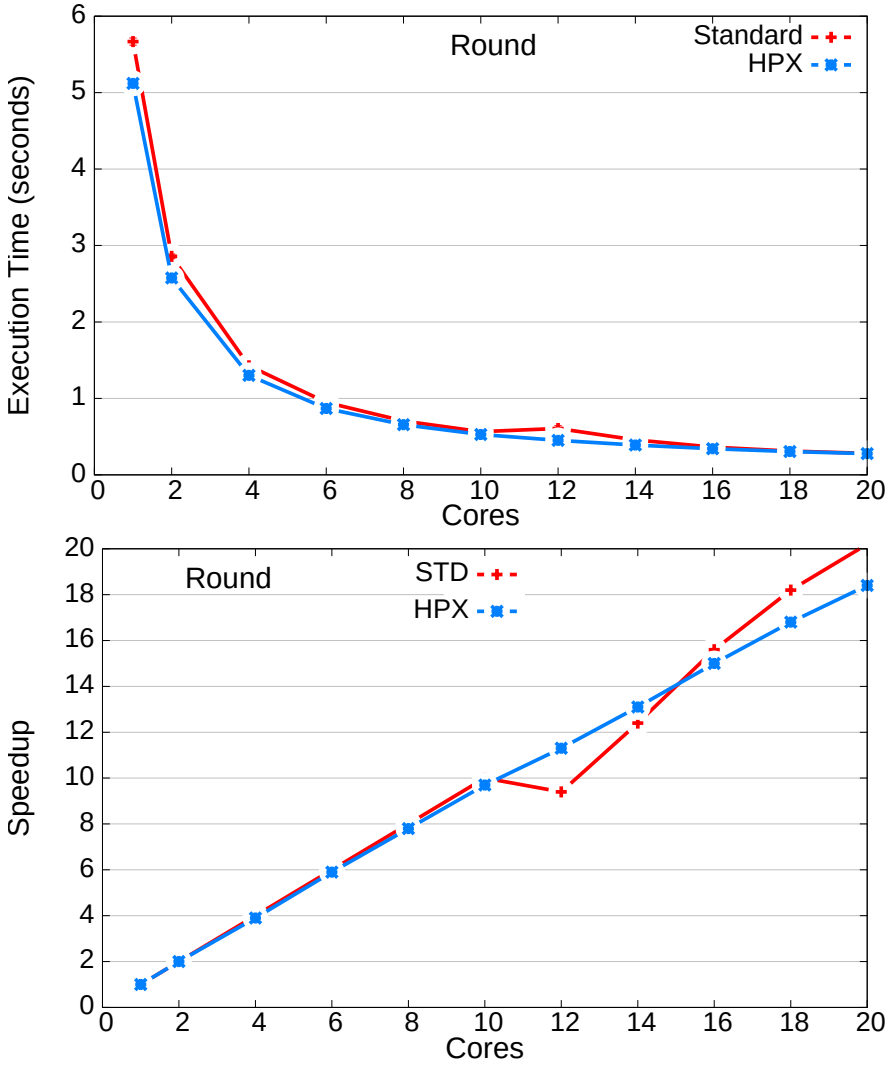

Figure 103: Round: HPX vs. C++11 Standard, grain size ∼10 ms, coarse-grained

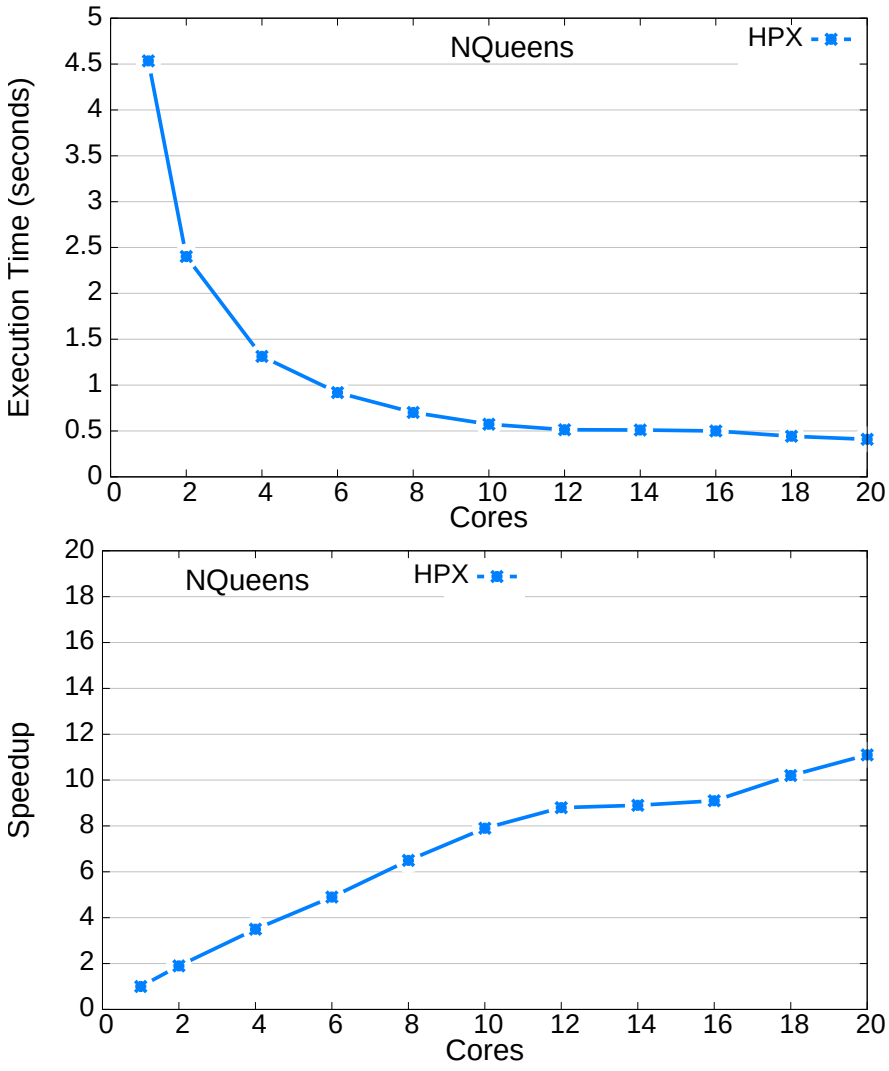

Figure 104: NQueens: HPX (C++11 Standard fails), grain size ∼28 µs, fine-grained

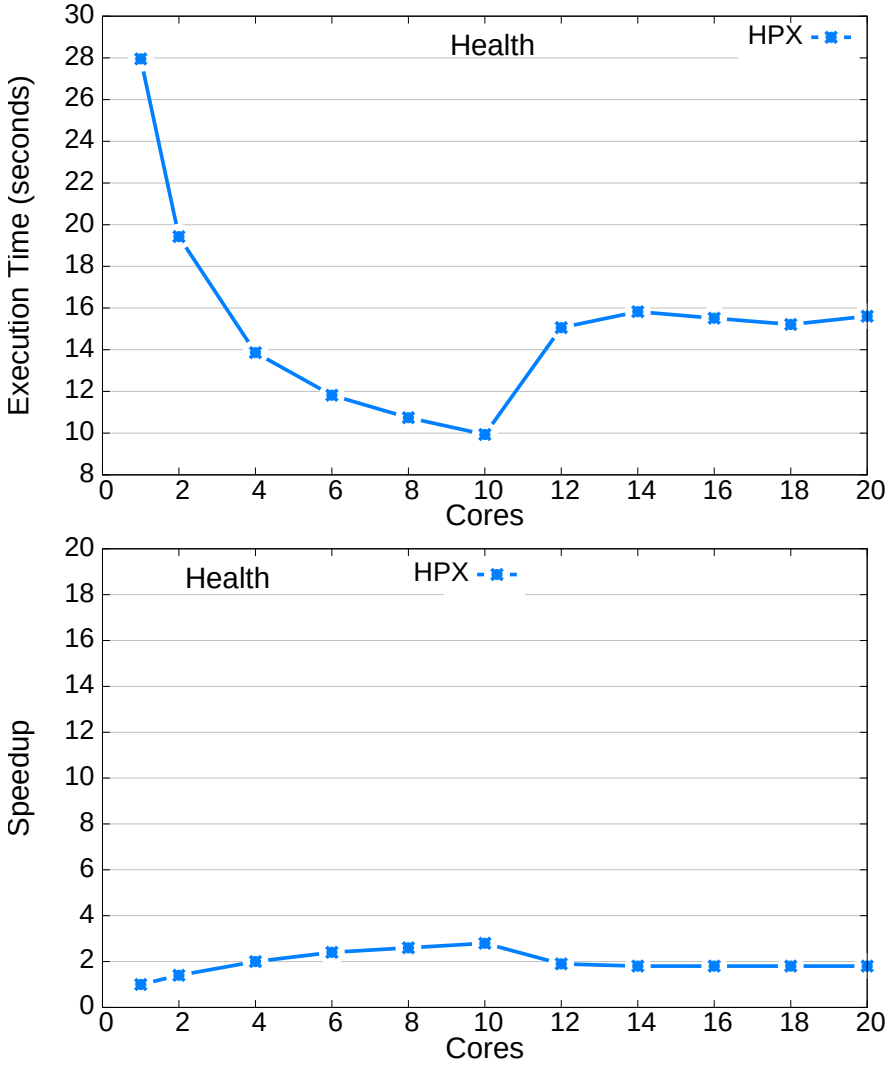

Figure 105: Health: HPX (C++11 Standard fails), grain size ∼1 µs, very fine-grained

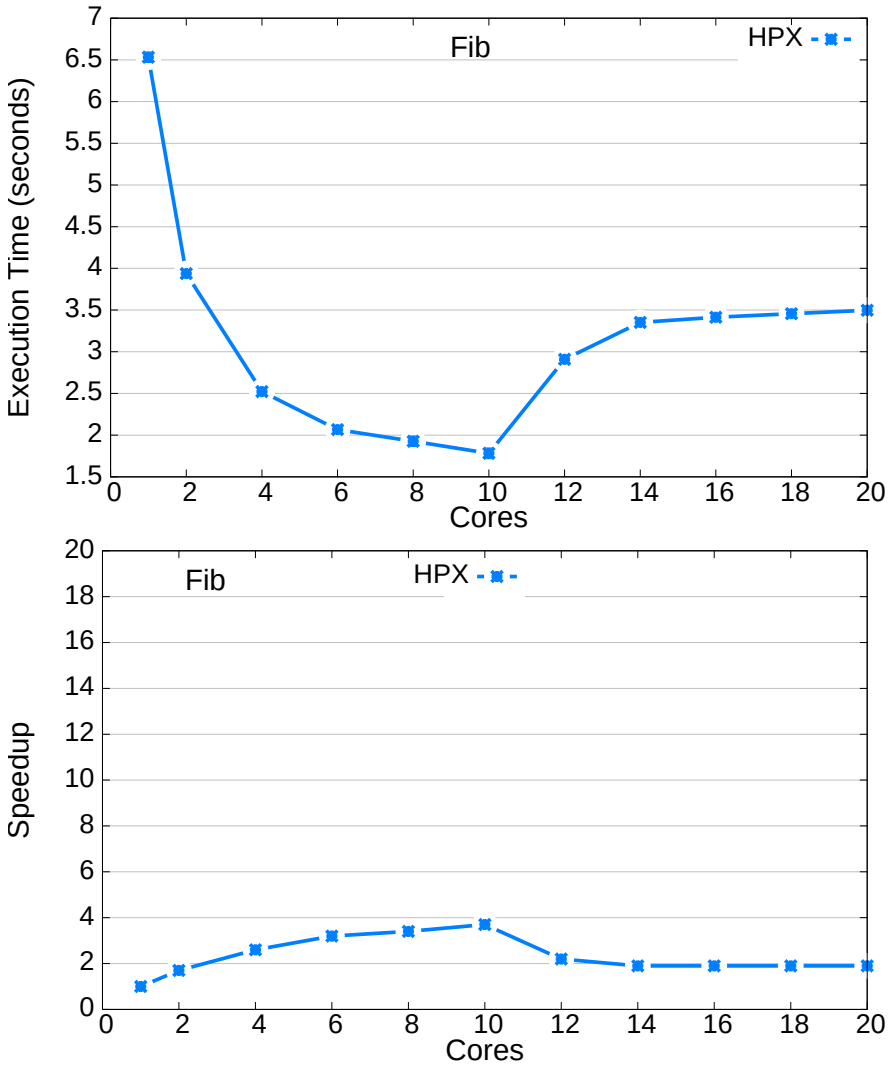

Figure 106: FIB: HPX (C++11 Standard fails), grain size ∼1 µs, very fine-grained

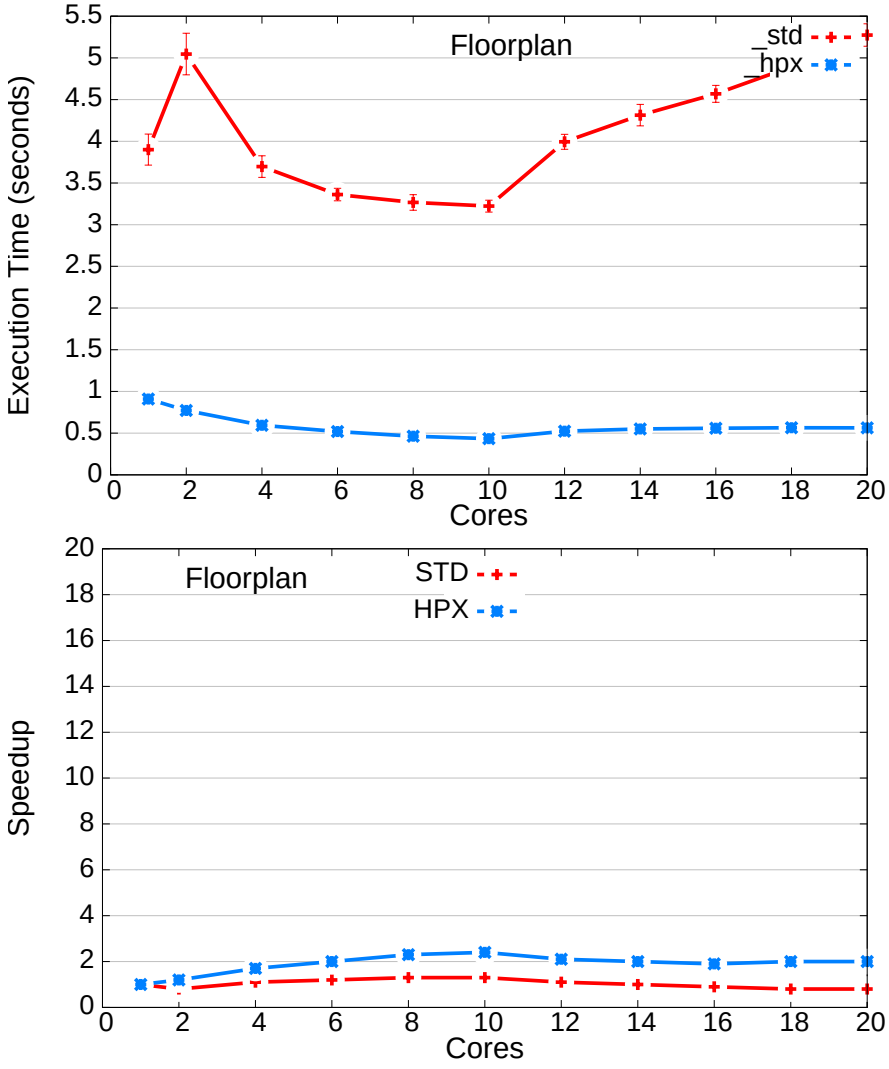

Figure 107: Floorplan: HPX (C++11 Standard fails), grain size ∼5 µs, very fine-grained

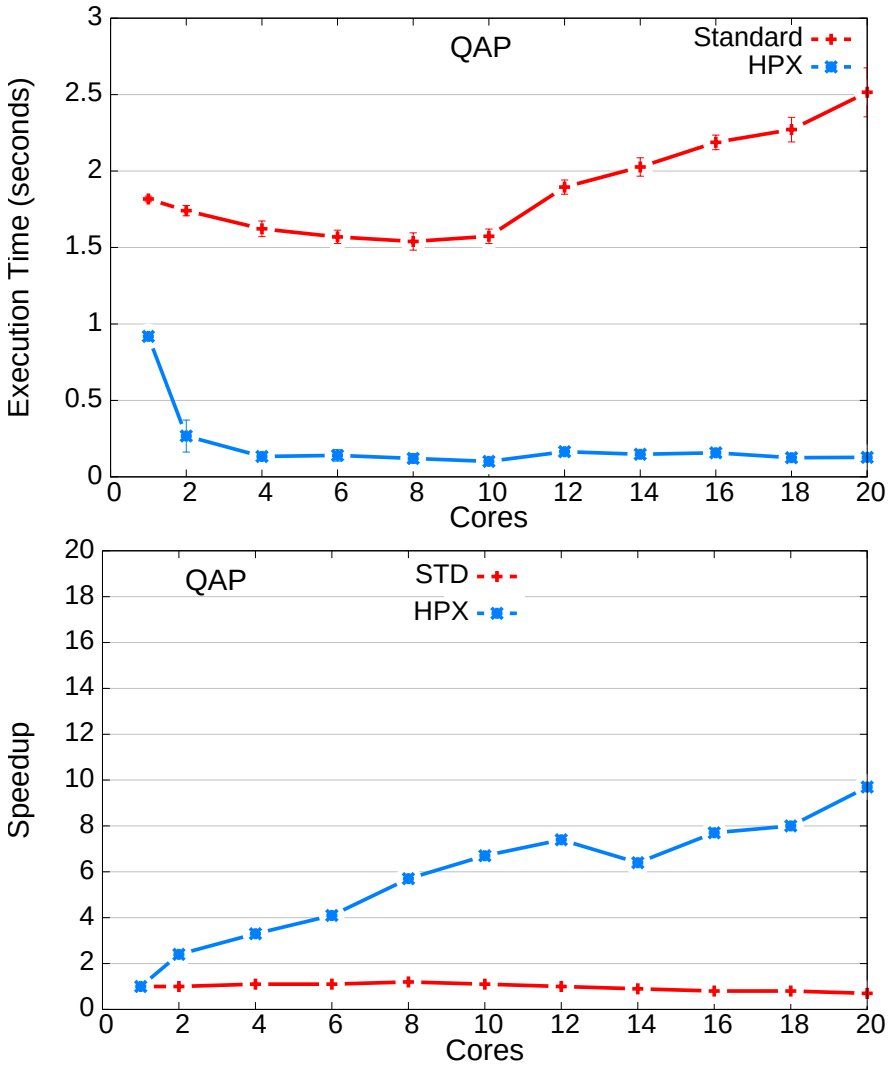

Figure 108: QAP: HPX (C++11 Standard fails), grain size ∼1 µs, very fine-grained

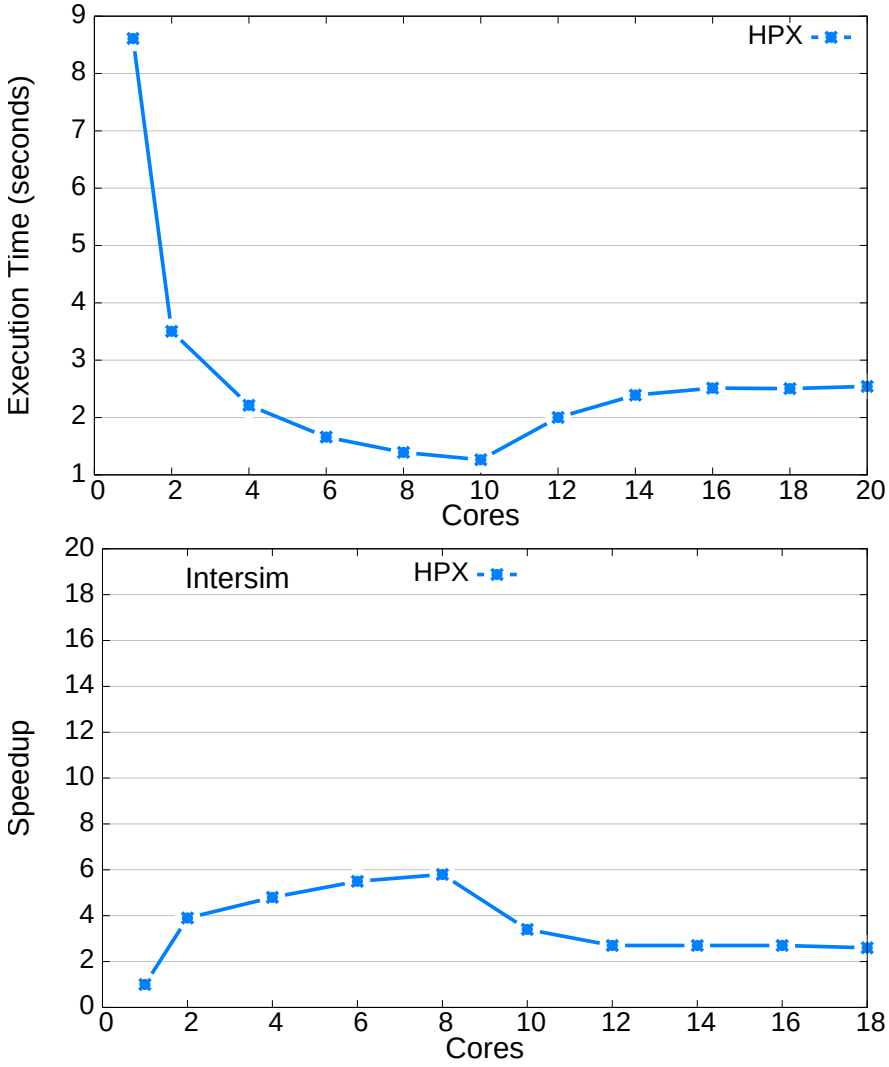

Figure 109: Intersim: HPX (C++11 Standard fails, although it displays execution time), grain size ∼3 µs, very fine-grained

B.2 Overheads Using HPX Performance Counters

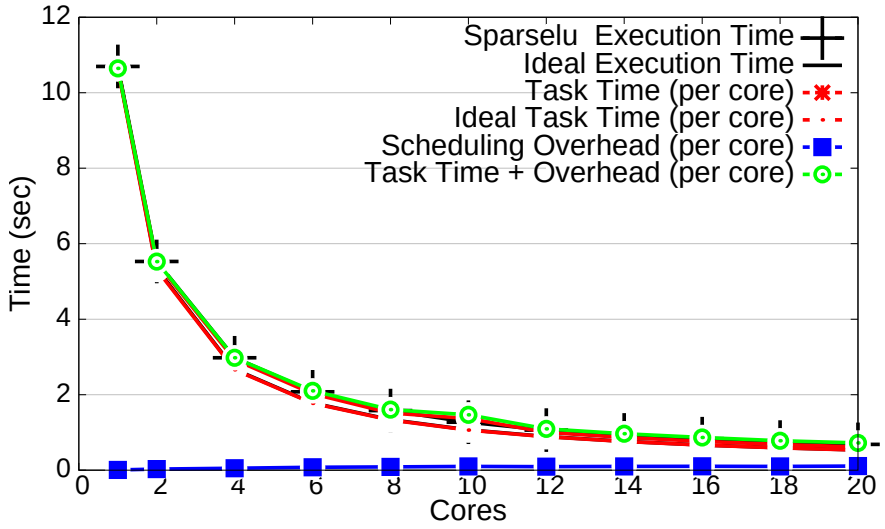

Figure 110: Sparselu: Overheads, coarse-grained ∼1 ms

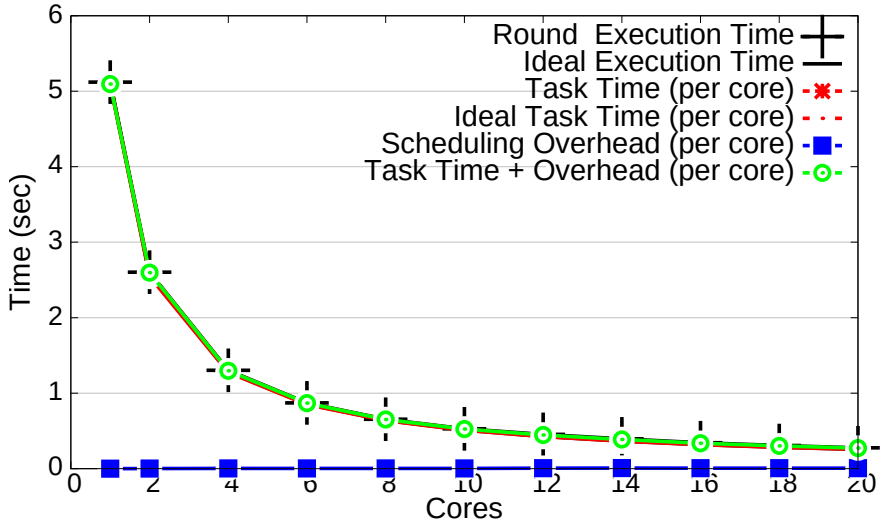

Figure 111: Round: Overheads, coarse-grained ∼10 ms

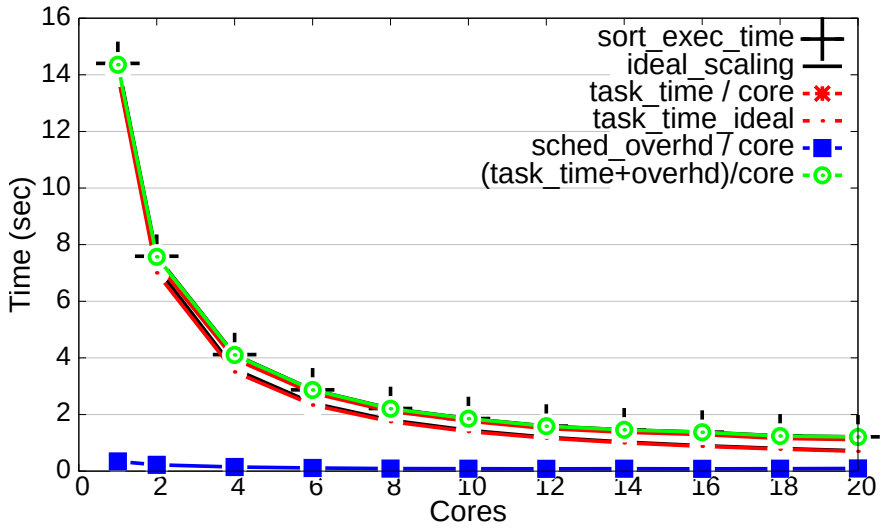

Figure 112: Sort: Overheads, fine-grained ∼50 µs

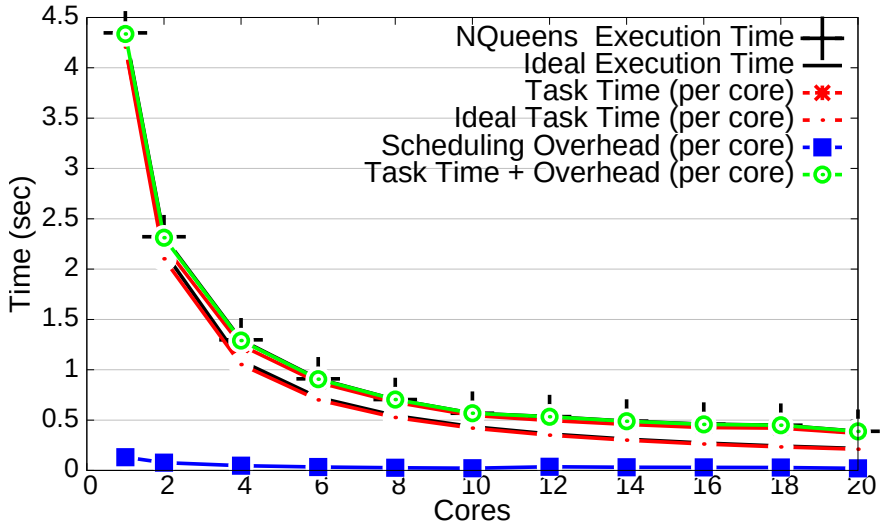

Figure 113: NQueens: Overheads, fine-grained ∼28 µs

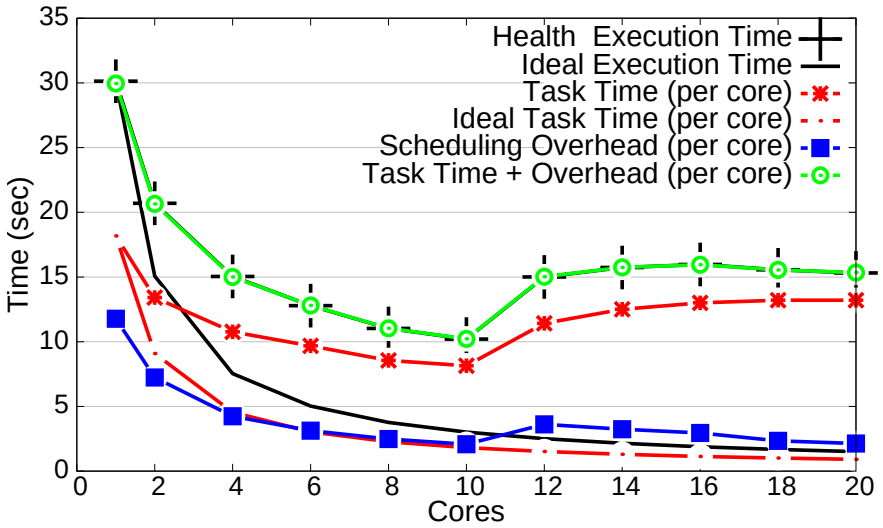

Figure 114: Health: Overheads, very fine-grained ∼1 µs

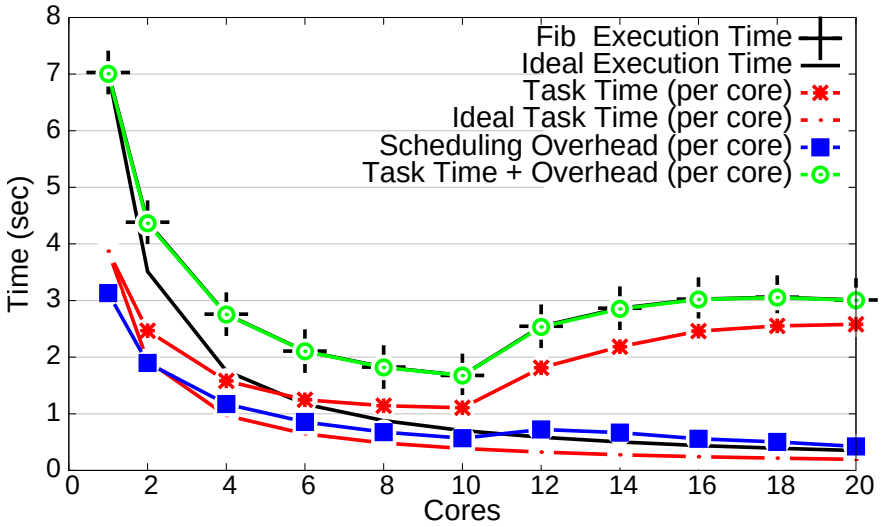

Figure 115: FIB: Overheads, very fine-grained ∼1 µs

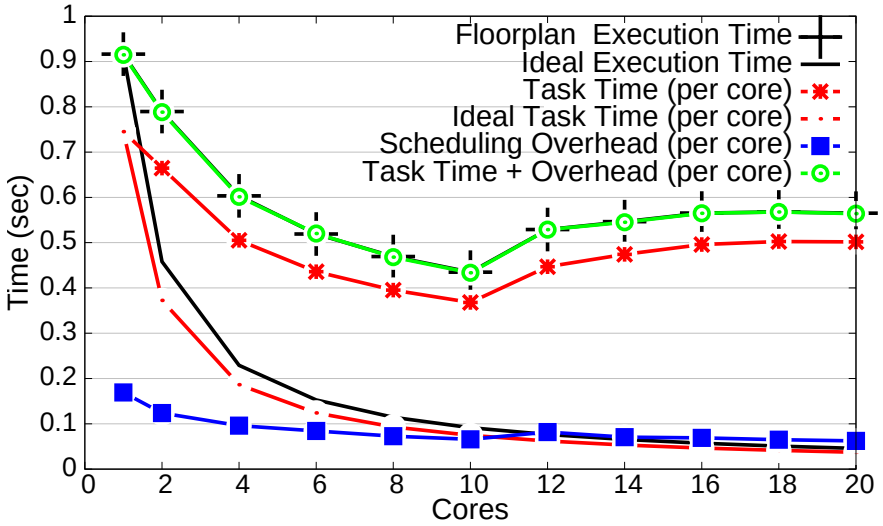

Figure 116: Floorplan: Overheads, very fine-grained ∼5 µs

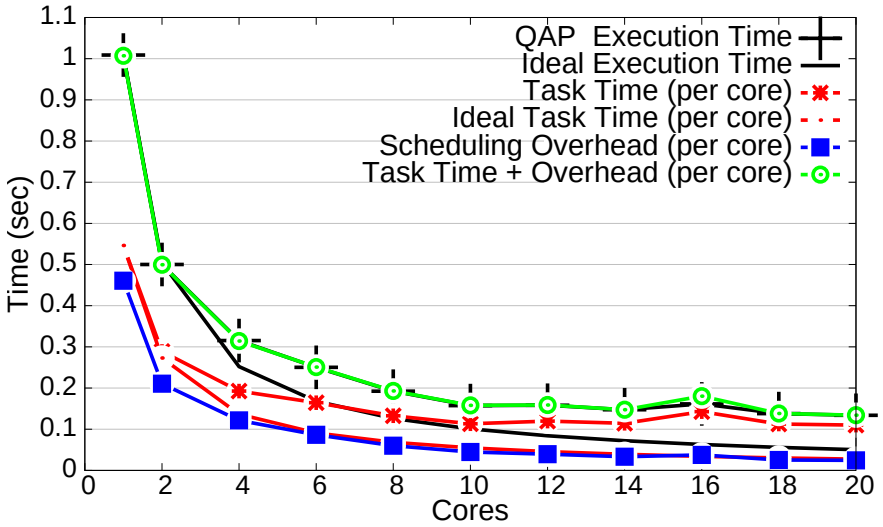

Figure 117: QAP: Overheads, very fine-grained ∼1 µs

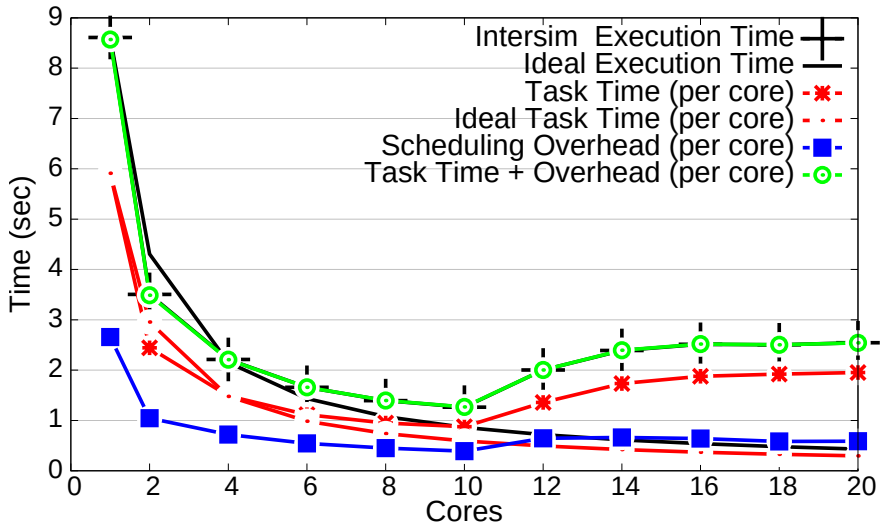

Figure 118: Intersim: Overheads, very fine-grained ∼3 µs

B.3 Offcore Bandwidth Utilization

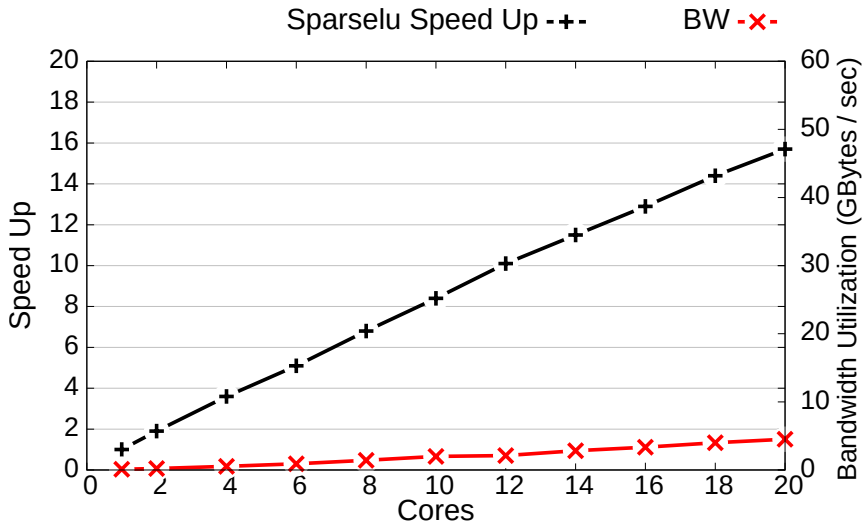

Figure 119: Sparselu: Offcore Bandwidth Utilization, coarse-grained $\sim$ 1 ms

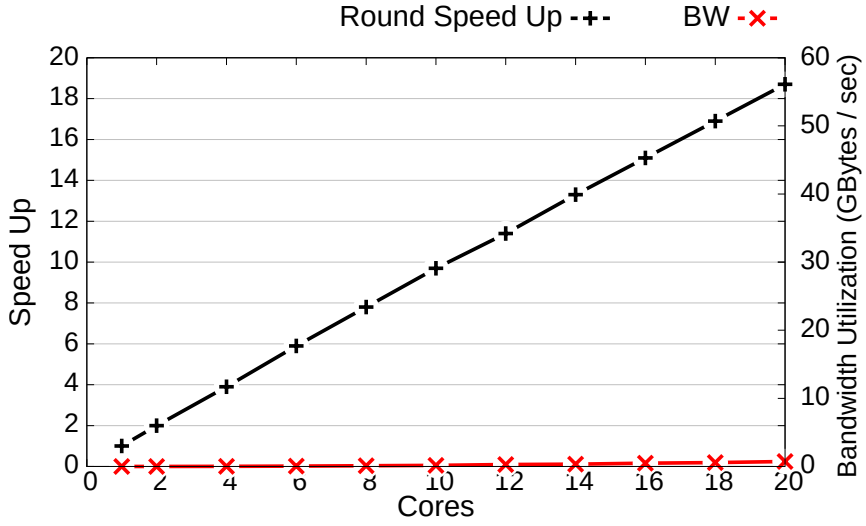

Figure 120: Round: Offcore Bandwidth Utilization, coarse-grained $\sim$ 10 ms

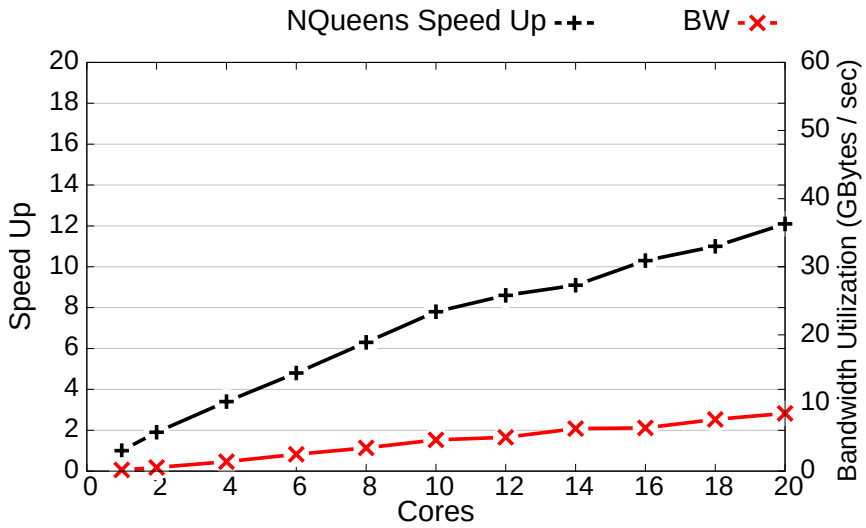

Figure 121: NQueens: Offcore Bandwidth Utilization, fine-grained ∼28 µs

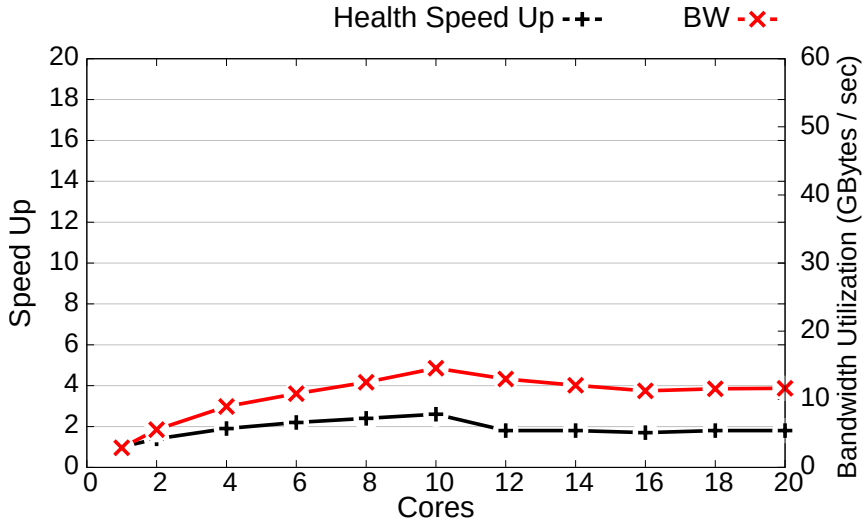

Figure 122: Health: Offcore Bandwidth Utilization, very fine-grained ∼1 µs

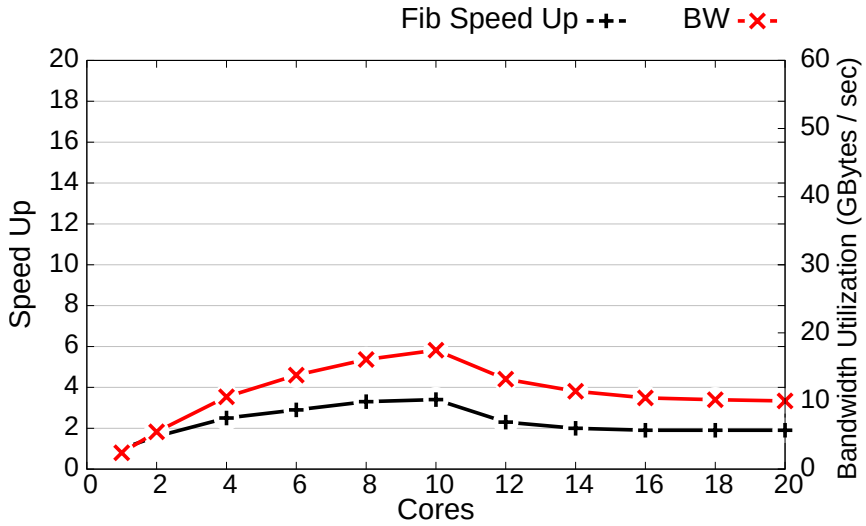

Figure 123: FIB: Offcore Bandwidth Utilization, very fine-grained ∼1 µs

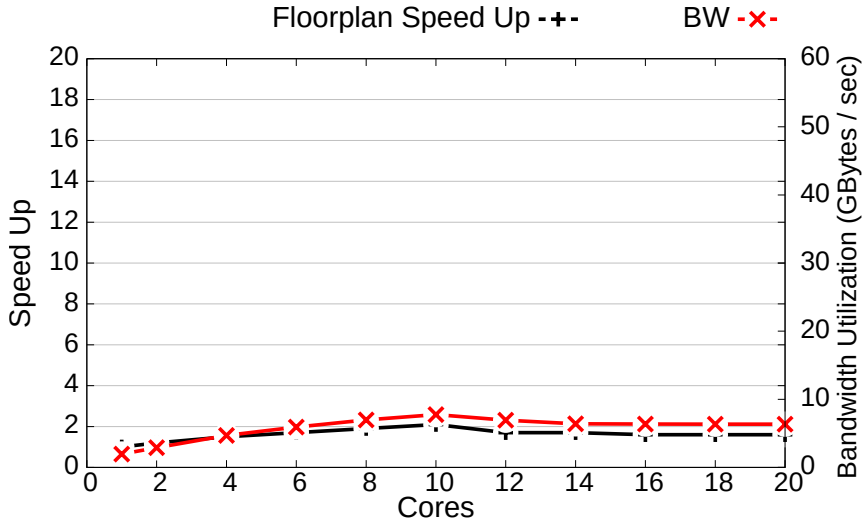

Figure 124: Floorplan: Offcore Bandwidth Utilization, very fine-grained ∼5 µs

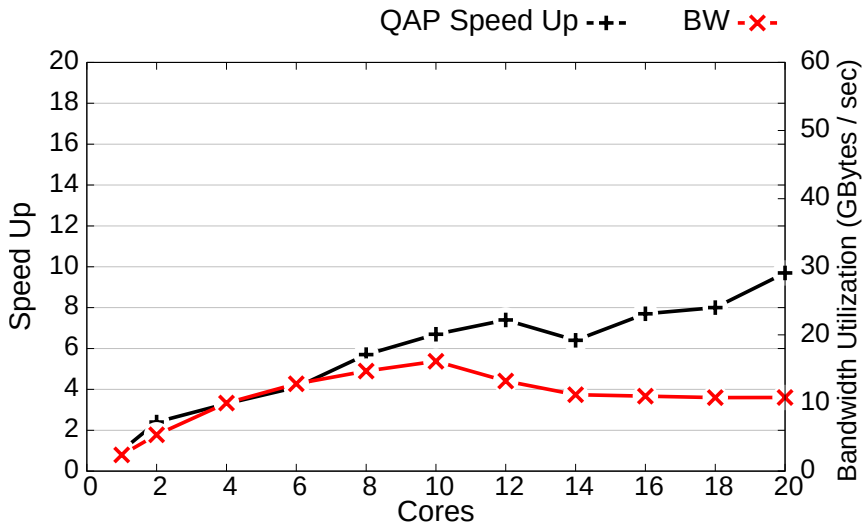

Figure 125: QAP: Offcore Bandwidth Utilization, very fine-grained ∼1 µs

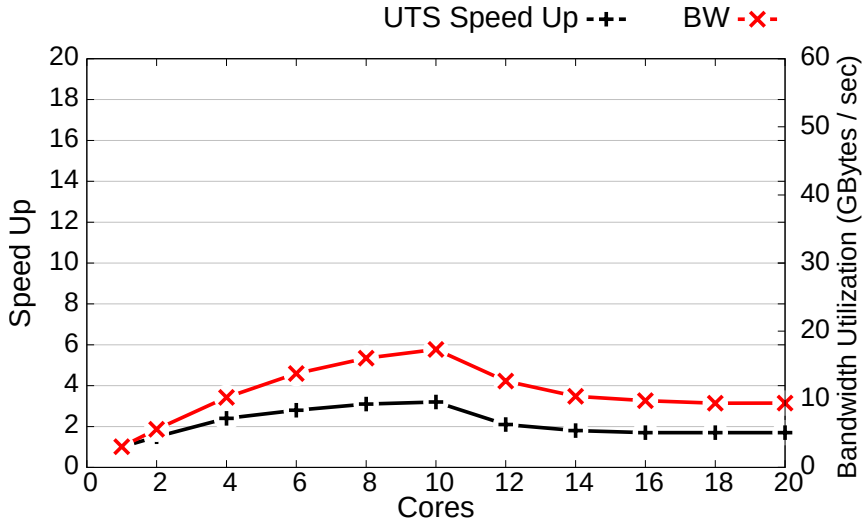

Figure 126: UTS: Offcore Bandwidth Utilization, very fine-grained ∼1 µs

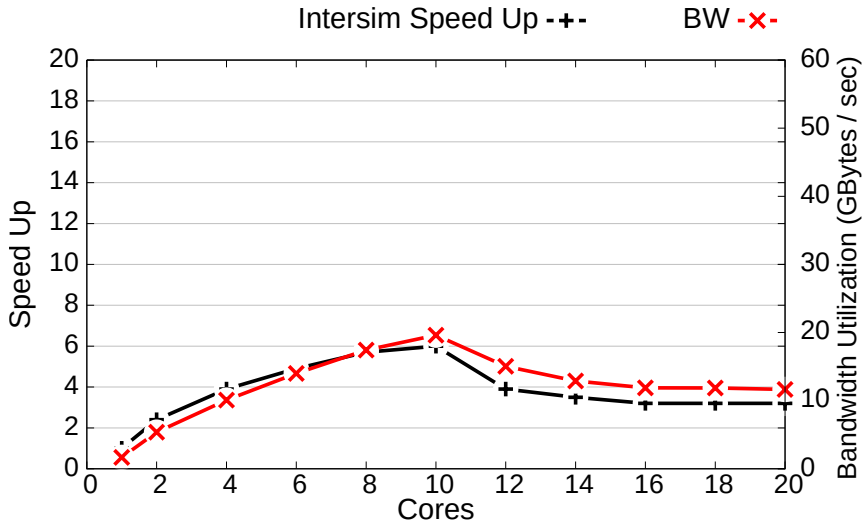

Figure 127: Intersim: Offcore Bandwidth Utilization, very fine-grained ∼3 µs

REFERENCES

[1] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, 2013.

[2] *Intel® Xeon® Phi™ System Software Developer's Guide*. Intel Corporation, 2014.

[3] MassiveThreads: A Lightweight Thread Library for High Productivity Languages, 2014. http://code.google.com/p/massivethreads/.

[4] OpenMP Specifications, 2014. http://openmp.org/wp/openmp-specifications/.

[5] The Innsbruck C++11 Async Benchmark Suite, 2014. https://github.com/PeterTh/inncabs.

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

[7] Stellar-Group/INNCABS, 2015. https://github.com/STEllAR-GROUP/inncabs.

[8] Umut A. Acar, Arthur Chargueraud, and Mike Rainey. Scheduling Parallel Programs by Work Stealing with Private Deques. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 219–228, New York, NY, USA, 2013. ACM.

[9] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs Http://Hpctoolkit.Org. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, April 2010.

[10] Arvind and R. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture". In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE '87, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*. Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 259.

[11] Henry C. Baker and Carl Hewitt. The Incremental Garbage Collection of Processes. In *SIGART Bull.*, pages 55–59, New York, NY, USA, August 1977. ACM.

[12] Jordan Bell and Brett Stevens. A survey of known results and research areas for -queens. *Discrete Mathematics*, 309(1):1 – 31, 2009.

[13] The OpenMP Architecture Review Board. OpenMP Application Program Interface V3.0, 2008.

[14] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21:291–312, 2007.

[15] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[16] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[17] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[18] 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*, pages 57–66, Sept 2008.

[19] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[20] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry- Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[21] Jack B. Dennis. First Version of a Data Flow Procedure Language. In *Symposium on Programming*, pages 362–376, 1974.

[22] Jack B. Dennis and David Misunas. A Preliminary Architecture for a Basic Data-Flow Processor. In *25 Years ISCA: Retrospectives and Reprints*, pages 125–131, 1998.

[23] 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.

[24] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An Adaptive Cut-off for Task Parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 36:1–36:11, Piscataway, NJ, USA, 2008. IEEE Press.

[25] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of OpenMP Task Scheduling Strategies. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 100–110, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] Marie Durand, Franois Broquedis, Thierry Gautier, and Bruno Raffin. An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines. In *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 141–155. Springer Berlin Heidelberg, 2013.

[27] T.v. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pages 256–266, 1992.

[28] Tarek El-Ghazawi and Lauren Smith. UPC: Unified Parallel C. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 27, New York, NY, USA, 2006. ACM.

[29] Daniel P. Friedman and David S. Wise. CONS Should Not Evaluate its Arguments. In *ICALP*, pages 257–284, 1976.

[30] P. Grubel, H. Kaiser, J. Cook, and A. Serio. The Performance Implication of Task Size for Applications on the HPX Runtime System ©2015 IEEE. Reprinted, with permission, from Grubel, P. and Kaiser, H. and Cook, J. and Serio, A. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 682–689, Sept 2015.

[31] P. Grubel, H. Kaiser, K. Huck, and J. Cook. Using Intrinsic Performance Counters to Assess Efficiency in Task-based Parallel Applications. Reprinted with permissions, from Grubel, P. and Kaiser, H. and Huck, K. and Cook, J. 2016. (accepted) IPDPSWS HPCMASPA, Chicago, May 2016.

[32] Robert H. Halstead, Jr. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, October 1985.

[33] Kevin Huck, Allan Porterfield, Nick Chaimov, Hartmut Kaiser, Allen Malony, Thomas Sterling, and Rob Fowler. An Autonomic Performance Environment for Exascale. *Supercomputing frontiers and innovations*, 2(3), 2015.

[34] Kevin Huck, Sameer Shende, Allen Malony, Hartmut Kaiser, Allan jh, Rob Fowler, and Ron 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*, ROSS '13, pages 8:1–8:8, New York, NY, USA, 2013. ACM.

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

[36] Hartmut Kaiser, Maciej Brodowicz, and Thomas Sterling. ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications. In *Parallel Processing Workshops*, pages 394–401, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[37] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar 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*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[38] Hartmut Kaiser, Thomas Heller, Agustin Berge, and Bryce Adelstein-Lelbach. HPX V0.9.11: A General Purpose C++ Runtime System for Parallel and Distributed Applications of Any Scale, 2015. http://github.com/STEllAR-GROUP/hpx.

[39] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[40] Peter Kogge et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical Report TR-2008-13, University of Notre Dame, Notre Dame, IN, 2008.

[41] Charles E. Leiserson. The Cilk++ Concurrency Platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA, 2009. ACM.

[42] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[43] Anirban Mandal, Rob Fowler, and Allan Porterfield. System-wide introspection for accurate attribution of performance bottlenecks.

[44] Jun Nakashima, Sho Nakatani, and Kenjiro Taura. Design and Implementation of a Customizable Work Stealing Scheduler. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '13, pages 9:1–9:8, New York, NY, USA, 2013. ACM.

[45] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P Sadayappan, and Chau-Wen Tseng. UTS: An Unbalanced Tree Search Benchmark. In *Languages and Compilers for Parallel Computing*, pages 235–250. Springer, 2007.

[46] Stephen L. Olivier, Bronis R. de Supinski, Martin Schulz, and Jan 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*, SC '12, pages 65:1–65:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[47] Allan Porterfield, Rob Fowler, Anirban Mandal, David O'Brien, Stephen Olivier, and Michael Spiegel. Adaptive Scheduling Using Performance Introspection, RENCI Technical Report TR-12-02, Renaissance Computing Institute, 2012.

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

[49] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[50] S. Subramaniam and D.L. Eager. Affinity Scheduling of Unbalanced Workloads. In *Supercomputing '94., Proceedings*, pages 214–226, Nov 1994.

[51] Yanhua Sun, Gengbin Zheng, Pritish Jetley, and Laxmikant 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.

[52] D. Terpstra, H. Jagode, H. You, and J Dongarra. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing*, pages 157–173. Springer Verlag, 2009. 3rd Parallel Tools Workshop.

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

[54] The C++ Standards Committee. ISO International Standard ISO/IEC 14882:2014, Programming Language C++. Technical report, Geneva, Switzerland: International Organization for Standardization (ISO)., 2014. http://www.open-std.org/jtc1/sc22/wg21.

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

[56] P. Thoman, P. Gschwandtner, and T. Fahringer. On the Quality of Implementation of the C++11 Thread Support Library. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 94–98, March 2015.

[57] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy Scheduling: A Runtime Adaptive Scheduler for Declarative Parallelism. *ACM Trans. Program. Lang. Syst.*, 36(3):10:1–10:51, September 2014.

[58] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005. http://www.gwu.edu/~upc/publications/LBNL-59208.pdf.

[59] David W. Wall. Messages as Active Agents. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 34–39, New York, NY, USA, 1982. ACM.

[60] K.B. Wheeler, R.C. 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*, pages 1–8, April 2008.

[61] Yong Yan, Canming Jin, and Xiaodong Zhang. Adaptively Scheduling Parallel Loops in Distributed Shared-memory Systems. *Parallel and Distributed Systems, IEEE Transactions on*, 8(1):70–81, Jan 1997.