2014

Parallel Processes in HPX: Designing an Infrastructure for Adaptive Resource Management

Vinay Chandra Amatya
Louisiana State University and Agricultural and Mechanical College

PARALLEL PROCESSES IN HPX:
DESIGNING AN INFRASTRUCTURE FOR ADAPTIVE RESOURCE MANAGEMENT

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science and Electrical Engineering

by
Vinay C Amatya
B.E., Tribhuvan University, 2005
M.S., Louisiana State University, 2012
December 2014

Acknowledgments

I would like to take this opportunity to thank individuals and experts in the field who helped me bring this research into fruition. With immense pleasure, I would like to thank members of my committee Dr. Hartmut Kaiser, Dr. Bijay B. Karki, Dr. Konstantin Busch, Dr. J. Ramanujam and Dr. Neal Stoltzfus for their guidance and support. I would also like to thank Dr. Thomas Sterling and Dr. Maciej Brodowicz at Indiana University for broadening my conceptual understanding of the field. Further, I cannot remain without thanking my previous colleagues in ParalleX group at LSU, Dr. Chirag Dekate, Dr. Dylan Stark, Dr. Chirs Michaels Alex Tabbal Hari Sundarajan; current colleagues in "the Stellar Group" at LSU, Dr. Zach Byerly and Bryce Lelbach, and at FAU,Thomas Heller, for constructive discussions ,exploring new ideas and invaluable suggestions.

I would also like to thank my collaborators in New Mexico State University Patricia Grubel and Parsa Amini for working with me; most importantly Patricia for her valuable comments, thought provoking discussions as well as for her flawless grammatical editing and proofreading of this thesis. Further I would like to thank Center for Computation and Technology at LSU for facilitating the resources and infrastructures required for this research.

Last but not the least, I would like to thank my parents and my family members for their patience and unwavering support in realizing this work.

This work was partially supported by The Department of Energy (DoE) through the award DE-SC0008714 (XPRESS).

Table of Contents

List of Figures

Abstract

Advancements in cutting edge technologies have enabled better energy efficiency as well as scaling computational power for the latest High Performance Computing (HPC) systems. However, complexities due to hybrid architectures as well as emerging classes of applications, have shown poor computational scalability using conventional execution models. Thus, alternative means of computation, that addresses the bottlenecks in computation, is warranted. More precisely, dynamic adaptive resource management feature, both from systems' as well as applications' perspective, is essential for better computational scalability and efficiency. This dissertation presents and expands the notion of Parallel Processes as a placeholder for procedure definitions, targeted at one or more synchronous domains, meta data for computation and resource management as well as infrastructure for dynamic policy deployment. Additionally, it also presents use cases of Parallel Processes for system resource management, task management, locality management, for load balancing, access control, namespace management and performance measurement frameworks in HPX runtime system. Further, this work lists design principles for scalability of Active Global Address Space (AGAS), a necessary feature for Parallel Processes. Also, to verify the usefulness of Parallel Processes, a preliminary performance evaluation of different task scheduling policies has been carried out using two different applications. The first application is Unbalanced Tree Search, a dynamic graph application based on a reference implementation and implemented by this research in HPX. The other application is MiniGhost, a stencil based application based on reference implementation, using bulk synchronous parallel model. The results show that local and local-priority scheduling policies in HPX provide better performance for different classes of applications. However under certain application parameters, hierarchy scheduling policy performs better than both the schedulers in UTS; meanwhile giving overall poor performance in Minighost. Similarly, static scheduling policy show no performance gain in UTS, but shows scalable performance for Minighost. Further, choice of task granularity for a given problem size also plays a

role on the performance of applications, as witnessed in UTS. These observations support the hypothesis of the need of a dynamic adaptive resource management infrastructure, for deploying different policies and task granularities, for scalable distributed computing.

1 Introduction

The High Performance Computing (HPC) community is geared towards an era of extreme scale systems, where use of unconventional architectures would be a norm [EE14], partly to get better computational efficiency and energy profile, and partly to mitigate memory limitations [McK04] and communication latencies [Mar+97]. Marking this trend, we can witness proliferation of hybrid architectures that exhibit better power efficiency as well as high degree of parallelism in computations being deployed in the latest HPC systems. Traditional execution models such as OpenMP and MPI do not guarantee to maximally harness the extreme parallelism offered by such new architectures or future generation architectures for HPC systems [Ste11]. Further, due to sheer scale of concurrency exposed by these new systems, management of computations, with minimal synchronization overheads and latencies is a big challenge [Bor10]. Meanwhile there are emerging classes of applications and application frameworks that show poor scalability in current generation systems and computation models, due to lack of adaptive task management and granularity control [KBS09].

For an efficient mapping of applications of different types, onto current and future generation systems, that provide extreme degree of parallelism, while minimizing computational overheads, latencies and inefficiencies, we need an execution model that takes all these factors into account. The ParalleX execution model [Gao+07; KBS09] is one such alternative that attempts to address the application scalability challenges in current and future generation systems. The ParalleX execution model integrates well established and tested concepts, making a holistic approach to the scalability challenges posed by the new architectures as well as difficult to scale applications.

One of the major challenges posed by extreme scale computing is management of resources, both from hardware and software perspectives. The execution model slated for systems with unprecedented levels of concurrency has to address the issue of efficiently managing each entity in context of the astronomical size of computations. Resource management entails an efficient orchestration of complexities imposed by the billion way parallelism induced by unconventional hybrid architectures. Resource management also involves efficient management of applications tasks, so that they are seamlessly mapped to the available hardware, with the least penalty, as well as dynamic adaptive remapping of tasks, depending upon how loaded the originally scheduled system resource is.

1.1 Research Objective

1.1.1 Goal

The goal of this research it to provide a new paradigm for better computation management in context of extreme scale parallelism and unconventional hybrid system resources. The goal of this is achieved by elucidating the notion of Parallel Processes, as a platform for resource management and presenting some example use cases with some experimental results. With the formalization of the notion of Parallel Processes, the goal of this dissertation is to further explore the use case scenario where Parallel Processes would play a crucial role in achieving better computational efficiency. With Parallel Processes, runtime systems would have a better platform for seamless and scalable distributed computing using distributed data structures, deploy better load balancing framework, that is not only based on user defined algorithms but also system fed real time performance data. Additionally Parallel Processes would be a platform for deploying access control framework for interacting components (users or applications) in an era of multi-modal, multi-physics application frameworks.

1.1.2 Hypotheses

The hypotheses of this dissertation are:

Hypothesis A:

For an efficient and scalable computation for extreme scale systems with complex applications, a resource management infrastructure is required, that is based not only on user defined algorithms but also on real time performance data of the application.

Hypothesis B:

The scalability of the resource management infrastructure is itself dependent on the scalability of Active Global Address Space (AGAS) service, as each component of the resource management infrastructure is a named object.

Hypothesis C:

Withing a given resource management framework, different applications behave differently to disparate execution policies.

Hypothesis D:

Dynamic adaptive resource management would provide better performance result, rather than sticking just one form of execution policy or granularity control.

1.1.3 Objectives

The dissertation discusses why resource management will be an important feature that will determine the overall computational efficiency as well as scalability of a runtime system, based on an execution model that is designed for extreme scale computing, with both system level as well as application level intricacies that make scaling difficult. In order to minimize the effect of such application and system complexities on performance scalability, we discuss ParalleX execution model, along with its runtime system implementation HPX to achieve the following objectives:

a. Formilize the notion of Parallel Processes as an infrastructure for distributed computation, with customizable policy deployment platform and optimal work management framework.

b. Identify different use case scenarios of Parallel Processes as resource management framework as well as policy deployment framework.

c. Identify design decisions for scalability of AGAS service, which impacts the overall performance of resource management framework.

d. Utilize different scheduling policies and task granularity for task management in different types of applications as a foundational example for parallel processes implementation.

1.2 Technical Strategy

The technical strategy to establish a foundation for performance scalability of applications through new runtime system based on the guidelines illustrated by an execution model for extreme scale computing comprises of:

1. Elucidation of advanced semantics of Parallel Processes.

2. Using semantics of Parallel Processes as foundation for further define framework for resource and computation management.

3. Provide design guidelines for scalability of such resource and computation management framework.

4. Development and use of reference implementation of different classes of applications for testing the hypothesis proposed.

5. Experimentation and analysis of preliminary performance results of benchmark application in context of hypotheses proposed.

With a clear semantic representation of Parallel Processes, as building blocks, development of resource management and access control framework, for advanced computation and resource management of billion way parallel entities becomes much simpler and organized. Again, identifying parameters for scalability of services or frameworks that the resource management framework depends upon would provide a holistic approach for scalable performance improvement in extreme scale computing. Finally, testing different performance parameters of benchmark applications' scalability, with the context of resource management, that shows expected performance behaviors under different settings and conditions, would verify the value of such a resource management infrastructure.

1.3 Dissertation Outline

The remainder of this dissertation is organized as follows: Chapter 2 provides an overview of the evolving paradigms of computations the HPC community is witnessing and establishes a need for a more sophisticated means of managing computations both from applications' as well as hardware resources' perspective. Chapter 3 discusses the ParalleX execution model with it principal concepts, discusses the software runtime system HPX, based on ParalleX, and discusses different scheduling policies currently implemented in HPX. Chapter 4 formalizes the notion of Parallel Processes and discusses many of its important use cases. Chapter 5 outlines a list of design requirements for scalable AGAS service. Chapter 6 presents experimental results of two different benchmark applications: Unbalanced Tree Search and Minighost for different problem sizes as well as different scheduling policies. Chapter 7 presents some related work. Chapter 8 draws the conclusion, discusses future research direction beyond the scope of this dissertation and lists key contributions of this dissertation.

2 Background

Since the introduction of first high performance computing (HPC) systems, the demand for scalable computational power has quickly proliferated for improvement of science, technology and knowledge [Lav78]. Consequently, computation models for achieving scaling concurrent computations for such systems also appeared and evolved, beyond the classical von Neuman model [Dij02; Dij68; Hoa02; RH02]. Together with major technological milestones in HPC systems as well as computation models that optimized utility of such systems, a host of complex scientific problems have become a solution target. In science and engineering disciplines such as computational fluid dynamics (CFD), applied physics, biology, environmental disasters, weather forecasts, energy physics, etc. computational simulations using HPC systems, have not only provided a platform with a cutting edge in advancing technologies, but has also resulted in break through scientific discoveries, in both academia as well as industry [UC91][Key12].

The catalytic role of computational simulations in advancement of science and technology has ushered the HPC community into an era of massive parallelism, with the introduction of massively parallel processing (MPP) systems, and commodity clusters [Pot85; BB99]. With increased computational power, larger and more complex problems could now be addressed. With the clear benefits, the demand and growth of more powerful HPC systems is ever increasing. Today, we have many HPC systems that can achieve several hundreds of tera flops to tens of peta flops [Top]. While building these massive systems is getting cheaper per flop, increasing size and scale (hence increasing performance rate) also increases the power profile for such systems in comparative ratio. Current HPC systems that have passed the peta scale barrier have energy footprints in the order of tens of

MegaWatts with efficiency of hundreds of MegaFlops per watt to couple of GigaFlops per watt [Top]. The same energy efficiency per unit computation is, however, environmentally unsustainable if we are to move to Exascale era computing, where energy footprint of a system is estimated between 60 - 130 megawatts [SA+09; SZS+07].

For scalability of the future HPC systems, they need to exhibit performance efficiency in the order of 10s of gigaflops per watt. This energy efficiency has to be obtained through new hardware architecture designs with better performance, power management as well as through new execution models, which minimizes overheads and takes evolving hardware architectures into consideration. From execution model perspective, sticking with older methods of computation may lead to inefficiencies that creep into the overall productivity of the systems. For example, as problem sizes have increased, so have the associated data [Jos]; hence movement of huge data increases inefficiency in the computation model, as it is not doing any productive work, thus exacerbating energy usage as well as introducing latency overheads. Execution models that are slated for future generation HPC systems (Exascale and beyond), should address the overheads that exist in the present computation paradigms and practices.

Typical causes in degradation of efficiency in computation with conventional systems and execution models are starvation, latency, overheads and waiting for contention resolution(SLOW) [Kai]. If there is no significant redesign of the conventional models of computation or design and development of entirely new computation models that take the performance degradation factors into account, such overheads would add up to significant amount, which cannot be ignored, when the computation model is subjected to computation with billion way parallelism and extreme scale. In Exascale systems, the sheer number of devices exposed will introduce significantly high level of concurrency issues [Ama+11b]. With the higher tendency of adopting lower clock frequency coupled with memory bandwidth limitations [JSLO], the aforementioned overheads would only get worse. Therefore such an unprecedented degree of parallelism demands radically different methods of ex-

ploring parallelism, with fine grain work flow management schemes or dynamic grain size control mechanisms for heterogeneous systems.

While the hardware achitecture of HPC systems is taking radical phase changes in multiple areas in order to achieve the next level of performance capabilities, there is an urgent need for the HPC applications that need to be prepared for tranfer from current generation HPC systems to the next generation HPC systems. The challenges are both in terms of understanding parallelism bottlenecks in applications as well as in understanding how the characteristic features of applications could be exploited on the massive systems to achieve maximum resource utilization. Some of the Exascale software challenges are discussed in the Sept, 2009 Department of Energy report [SA+09]. Important classes of applications such as those based on Partial Derivative Equations (PDE) solvers using Adaptive Mesh Refinement (AMR) and graph problems (bio-informatics, social networks, natural language processing) are difficult to scale even with current massively parallel computing environments [Str+09][Lum+06]. The first category of applications require computations to be done on a section of data for a longer time than other sections. These characteristics not only make the execution flow unbalanced but also need more synchronizations so that data dependencies are appropriately satisfied. As such, the applications cannot utilize more than a fraction of compute power of the system where these applications are executed. For the same category of problems, conventional parallel programming methods using MPI expose only coarse grain size for the problem [KW10], thereby severely limiting scalability of such applications. Meanwhile graph based applications quickly escalate in both volume and complexity, that systems that do not expose adequate parallelism, with scalable resources along with resource management features, cannot cater to the applications scalability requirements. Such dynamic applications necessitate dynamic load redistribution of tasks, so that work from heavily loaded system unit can be migrated to less loaded ones. However, this solution approach has a limitation that data movement needs to be minimized, as it is an additional overhead. A message driven fine grained parallelism mechanism would

be an attractive solution for such scenarios. Apart from the two application classes mentioned, several other classes of applications exhibit similar execution and resource usage traits. To achieve improved computational scalability for such problems and be able to exploit the billion way parallelism exposed by Exascale level of computation, to minimize overheads in the conventional model of execution, to expose dynamic grain sizes based on the need of the applications a new execution model and programming methodology is required [Ama+11b]. Limited computational scalability is a problem the HPC application developers are continually striving to mitigate even in today's Terascale systems, which still is a big representation in the top500 list[Top]. To be able to exploit the vast resources available in latest Petascale systems; and in future Exascale systems, the runtime systems on such systems need to exhibit intelligent concurrent resource management and utilization. Again, in order to achieve such high resource utilization, either the application programmers need to be aware of the concurrency and locality attributes of the application, which might necessitate rewrite of some parts of the application for every new programming model introduced. Alternatively some sort of system agnostic virtualization mechanism or runtime system/language feature that provides parallelism constructs to the applications could be used. As a convention, application classes can be broadly categorized into two groups from the perspective of scalability, a) Strong Scaling and b) Weak Scaling. Strong scaling class requires applications to be very cohesive and minutely optimized with minimal communication latency and data distribution and dependency resolution overheads. Very few applications exhibit strong scaling behavior in todays HPC systems, as the amount of work per compute unit decreases as we scale the applications with added resources. This is either due to magnification of overheads, SLOW or due to serial portions of the application limiting scalability, as depicted by Amdhal's Law [HM08]. On the other hand, weak scaling applications don't require as much cohesive organization of computations as there is lesser data dependence between units of computation of the applications. Per compute unit workload stays almost level, as the application size increases are commensurate with the

increased system resources utilized. Meanwhile, from the perspective of utilizing Exascale systems, new trends in application development are taking precedence as 'new era' weak scaling applications [SHS09]. In the 'new era' weak scaling applications, besides spatial scaling providing much finer resolution of results, we observe additional work would be done per unit data, such as in multi-phase, multi-scale, multi-modal, multi-physics, informatics, data-mining applications. These approaches either use multiple different solution methods or couple multiple applications to solve complex problems.

Witnessing the current trend in application development and system technology advancements, applications that will run in future generation HPC systems can be briefly categorized as:

a. Legacy HPC applications at Exascale.

b. Coupled Models.

c. Data Intensive Data Mining Applications with huge data sets.

d. Real-time Departmental Extreme Scale Applications.

e. Framework Technology (e.g. Cactus, NAMD).

In recent years, more and more complex scientific applications are being developed as framework technologies; as many of the the problems are multi disciplinary, where the developed framework caters not only the requirements of a particular problem set but is reusable in solving different other problems within the same domain (eg. CACTUS, NAMD) [Löf+13; Phi+05]. These complex application frameworks have many complex parts that need to be tracked and be able to provide interfaces to different specialized modules. This has larger impact in multi-modal applications as mentioned in 'new era' weak scaling applications. Providing such specialized interfaces would allow the domain scientists to put more focus on their scientific work. At the Exascale level, it may be estimated that the size and complexity of such multi-modal, multi-scale applications would

be petabytes in size. A high-level organizing feature would be desired for tackling the complexity and size, either provided at the language level or application framework level or runtime framework level.

The white paper on frameworks from the HPC Application Software Consortium (ASC) march 17, 2008 [All+08] categorizes frameworks based on levels of of interoperability:

- Minimal Component Interoperability: A majority of today's legacy HPC applications or solvers fall into this category, where there is minimal interoperability between applications of different domains, with applications being independent from other applications within the same domain. Across the domain analysis has to be done statically, after each independent application produces final results.

- Shallow Component Interoperability: Applications or solvers in this category, within a domain, are loosely coupled at some time step or discrete event. While internal state of each component solver is maintained, common data is exchanged using wrappers to data interchange interfaces over a network service.

- Deep Component Interoperability: Here domain solvers share a common service infrastructure for communication and data management. Different solver models of a particular domain are tightly connected at the level of interoperability. This allows action on common data almost without involvement of movement of data.

Further, each application frameworks has its own unique data and computation requirements. As such a different solution approach may be needed for each application category. Therefore, future application framework designs would need to be highly modular using various dynamic, efficient and parallel libraries. These parallelism techniques could be expressed in separate components, thus abstracting the implementation of parallelism. Such abstracted parallelism would minimize programming errors, meanwhile modular design would help targeted performance optimizations and code portability.

For better computational efficiency, future application frameworks would utilize some form of load balancing scheduler service/s as either external or internal component; together with dynamic performance measurement framework, which could be used as external or integrated component module with regard to the applications framework. Load balancing infrastructure made available outside the application framework as either external module or runtime service makes application programming simpler and more manageable.

Another important characteristic some leading classes of applications require is data structures/abstractions with efficient communication mechanism. While data arrays and vectors provide an effective means of expressing parallelism via data-parallel operations [Cve; Sub+93], data structures for graphs using pointers, pose a challenge in expressing such concepts efficiently [Lum+06]. Hence, an additional level of data abstraction using arrays and datafow streams [Xin+13], could be used help pointer based data structures to express better parallelism. Transforming graph data, with irregular shape and sizes, into an abstract and parallelizable construct is a desirable goal in Exascale applications.

For Exascale systems, locality is important both for containing energy footprint as well as minimizing latency. Although modern/future hardware architecture would provide better locality support, better software techniques for improved locality is warranted. Such improved software locality management schemes for Exascale systems would be a fundamental requirement for better performance efficiency and scalability for Exascale applications. While opportunities of reuse of data should be heeded by the programmers (with awareness of hardware architecture as well as programming model principles); and programming for the programming model might expose expose appropriate API for hand written locality management, the runtime system should also provide customizable locality management policies that are geared to (dynamically) support the data structure requirements given by the application that is running. The key to expressing locality in a portable manner is a parameterized decomposition of a dataset. For graphs, the structure of the problem may be exploited to generate good partitions inexpensively.

3 ParalleX Execution Model and HPX Runtime System

3.1 ParalleX Execution Model

ParalleX is an experimental execution model which makes an attempt to addresses many challenging aspects of Exascale computation systems that will be the norm in the near future [KBS09; Tab+11]. ParalleX execution model adopts message driven [Eic+92] computation model in a global address space context. ParalleX provides semantics for better efficiency through minimization of synchronization and scheduling overheads, with improved utilization of resources through asynchrony of work flow; provides language features (e.g. migration) for employing dynamic adaptive load balancing. ParalleX execution model advocates the semantics of variable granularity (fine grained to coarse grained) for worker threads and light weight synchronization objects for scalability of not only generic applications but also for scaling impaired applications. While sustained work is guaranteed with fine grained parallelism when work is available, coarser grained parallelism allows the computation to avoid costly overheads associated with creating fine grained threads, when there is less work available.

ParalleX strives to expose several forms of parallelism, in order to attain Exascale performance and beyond, improve strong scaling for certain classes of applications that are scaling impaired with existing execution models; provides a foundation for accomplishing improved scalability for classes of scientific applications, improve time and energy efficiency, reliability and programmability. As such the ParalleX execution model has been designed to directly address the following four primary factors in impending performance and scalability in applications:

- Starvation: Insufficient concurrent work to maintain high utilization of resources.
- Latencies: Time distances of remote resource access and services (such as IO or network).
- Overheads: The time spent on resources on the critical path required to implement parallelism and task synchronization which are not necessarily present in the sequential variant.
- Waiting for Contention Resolution: Delays due to conflicts on oversubscribed shared resources.

The design guidelines as detailed in [Kai+14] for ParalleX as an Exascale execution model are:

Latency Hiding:

Latencies in computation model cannot be completely avoided. There is always a certain degree of latency involved in the general computation. We can, however, take measures to minimize the overall effects of such latencies in computation through techniques both software (such as asynchronous task and communication methods) as well as hardware (latest state of art technologies) that hide latencies.

Better Granularity Control Mechanism:

It is well understood premise that if there is less available parallelism, deploying excess resources would not bring scalability. Amdhal's law [Amd67] just limits any such possibility. Again, not properly utilizing available resources, when there is enough available parallelism available also limits scalability. Hence, a balanced approach, with feedback information through performance measurement, should be deployed for dynamic adjustment of granularity control for mapping work to the resources available. Granularity control should be universally available in the programming

semantics with a choice between fine grained and coarse grained tasks, where the size of fineness and coarseness is dictated by the runtime feedback information, resources available as well as information fed by users.

Constraint Based Synchronization:

In many current standard programming models, global barriers are frequently used as a means of task synchronization. Explicit barriers, during synchronization of application's tasks minimizes exploitable parallelism, with resources idling while waiting for other parts of the computation to complete [Axe86]. The bulk of an application does not need to be barrier synchronized. Only a certain part of the application may need synchronization, which can be handled on case by case basis using constraint based synchronization, governed by a predicate. In addition, advanced data dependency determination would enable a better means of synchronizing task and resources. Such data dependency identification would allow parts of the overall task to proceed where it detects no data dependency, while the task segment whose data dependencies have not been met can wait asynchronously until all its dependencies are met. Such facility of identification of conditions for a need for synchronization should be provided by an execution model targeting future Exascale and beyond systems.

Adaptive Locality Control for Load Balancing:

As the future generation of HPC systems become more complex and heterogeneous and the applications size for such systems increases with uneven work loads, there is a need to support a way of providing dynamic work distribution across localities. Current programming models do not provide semantics for adaptive dynamic work distribution [Mes09; Yel+07]. The ability to dynamically distribute work and data when there is imbalance in the resource usage would significantly improve overall runtime performance of an application.

Moving work to data:

Maturity in our understanding of scientific problems have allowed us to integrate more complex models into even more intricate computation systems, it has become a well understood observation that these computation models have to deal with huge datasets [Jos]. Moving such huge datasets is inefficient, due to the simple fact that data movement is itself not a productive activity. Therefore, there should be a mechanism to be able to move action to data. It, however, needs some ascertaining beforehand, through empirical or analytical methods, when it is effective to move action to data and vice-versa. The new execution model should take this important factor into consideration.

Message Driven Computation:

Message driven computation has been gaining ground as a means of doing distributed computing, where the sender controls how a task is invoked at a remote locality [Cha; Aga+07]. This mechanism allows an efficient method for expressing work flow semantics, than conventional message passing or data parallel computation models [Mes09; CVG11]. The added flexibility of controlling work flow from the sender side allows efficient and fine grained synchronization and management of tasks; as well as allow asynchronous computation and communication to take place with better performance.

3.2 ParalleX Execution Model Components

The ParalleX execution model comprises of a collection of fundamental concepts; each of which exposes a particular set of properties, that have been be actualized as a software entity or hardware entity or combination of both. This section discusses the principal components of ParalleX execution model.

3.2.1 Light Weight Threads

Light Weight Threads (LWT) are the smallest possible grain size for representation of executable tasks. They are first class objects(i.e. they have a name), exist within the

context of a synchronous domain and have short lifetime. If a LWT represents more than one task, the work flow can be simple sequential function invocations, which may/may not return value, or static dataflow objects [JHM04], or combination of both. The result of LWT actions may update a local variable or a global variable, which is synchronized using local synchronizing objects, discussed below.

3.2.2 Parallel Processes

Parallel Processes are semantic constructs that are used for coordination and control of resources in a distributed environment. Minimally a Parallel Process consists of a meta data that would contain the pertinent information, such as address of a resource, resource usage, owner of a resource, etc., that the Parallel Process is responsible for. The semantic context of a Parallel Process may span multiple localities(synchronous domains), i.e. a Parallel Process may exist across multiple localities. For active Parallel Processes, a LWT would be minimal representation of a "context of execution" for the Parallel Process. Parallel processes may possess child process(es). A detailed discussion on this topic is presented in a later chapter.

3.2.3 Local Control Objects

Local Control Objects (LCO) are lightweight synchronization constructs that eliminate the need for global barriers for distributed synchronization. The goal of an LCO is to assist in dynamic global synchronization, in migration of objects and facilitate continuations as well as dynamic adaptive resource management, thereby enabling parallelism with various granularity. LCOs particularly exist in context of instantiation and synchronization of threads, hence hold complementary information on threads being synchronized, in a global perspective. An LCO is a finite state machine (FSM) [Sga10] whose state and data changes with accompanying program events. LCOs are first class objects and exist in a synchronous domain (eg. SMP systems). An LCO is always event driven and hence is instantiated directly or indirectly when a thread needs synchronization and is instantiated both in local or global perspective. LCOs however may continue to exist even after the thread has

finished execution. LCOs hold a predicate, depending upon which, it changes its state and hence allows synchronizing state change of thread(s) in context of the LCO; and an LCO may get activated (i.e. re-associated with new threads) when a task event (action) is incident on it.

3.2.4 Parcels

Parcels in ParalleX execution model bear the responsibility of communication paradigm, that allows asynchronous transfer of information (data or metadata), allows remote invocation of actions (function calls) and allows distribution of control flow through continuations. Parcels are the only means of migration of tasks objects, for dynamic adaptive load balancing in ParalleX in a distributed environment. Parcels are a type of active messages [Eic+92] that enable message driven computation. Parcels carrying actions instantiate a thread at the remote locality or instantiate a LCO that in turn will instantiate a thread if synchronization on the remote action is warranted. With mobility of tasks, work can be moved to data, which brings efficiency in situations where the data involved is too huge to move. The conventional computation method of moving data to work would still be possible by making a "get()", a simple fetch the result operation, through a parcel. Parcels can be targeted at any first class objects that exists within a given global namespace , can specify what actions to be performed at destination, and upon completion (absorption of parcel at remote locality) may update remote locality state as a side-effect, which is different from conventional send/receive method of communication. Parcels are composed of four fundamental fields: destination, action, data and continuation. Depending upon the requirements for minimizing latency and throughput, parcels' size may be small or large.

3.2.5 Percolation

Percolation is special use case scenario for parcels. Through percolation, messages carrying work, are directly targeted at the hardware resources rather than logical data objects [Jac+03]. This allows an efficient utilization of expensive resources, crosscutting the overheads involved in added abstraction layers when using logical means. Percolation, thus,

provides a means of heterogeneous computation on specialized hardware devices such as FPGA [SW10] and GPGPUs [FM04] as a means of doing generic computing. Since, within a synchronous domain such specialized hardware, also referred to as accelerators, have a different address space than the generic processing units; accelerators can be targeted by parcels, with parcel layer bridging the two architecturally disparate hardware units. Parcels can be used to make, work and data available to the specialized units as well as transfer the result/s from the accelerators to the originating computation unit. Since the percolation mechanism is aware of characteristics of specialized hardware resources, relevant tasks, when available, can always be made available to the resources with high efficiency.

Percolation allows high cost resources to be freed up, by offloading repetitive tasks to highly specialized resources. Coupled by the asynchrony facilitated by the execution model, task distribution to the accelerators and getting results back, using parcels, minimizes cost of the overall computation. Through percolation, we can get coarse grained parallelism for bulked tasks, while the accelerators themselves my expose yet another level of fine grained parallelism for the work-data combination made available to them.

3.2.6 Active Global Address Space

The Active Global Address Space (AGAS) is a fundamental component of the ParalleX execution model [Gao+07; KBS09; Tab+11]. AGAS provides a global virtual address for objects in a distributed execution environment context, thus providing a means of object visibility beyond a synchronous domain. This feature places AGAS into the class of "Global Address Space" programming models like Partitioned Global Address Space (PGAS) [Yel+07] models, with a fundamental difference. AGAS provides the means to move objects, after instantiation, across localities (conventional nodes). This aspect of AGAS is crucial for runtime adaptive dynamic load balancing across localities in a distributed computing context.

3.3 HPX: An Exascale Runtime System

The HPX runtime system represents a first attempt to develop a runtime system with comprehensive APIs with semantics based on the ParalleX execution model [Kai+14]. It is modular, feature-complete, and performance oriented representation of ParalleX model, currently targeted at conventional architectures, such as commodity clusters [BB99] and Symmetric Processing (SMP) systems [Gib66]. Its modular architecture allows for easy compile time customization and minimizes the runtime memory footprint. It enables dynamically loaded application-specific modules to extend the available functionality at runtime. Static pre-binding at link time is also supported. Its strict adherence to the Standard-C++ [The08] and the utilization of Boost [Var10] combines powerful compile time optimization techniques and optimal code generation with excellent portability. HPX is designed as an alternative to conventional computation models, such as MPI, while attempting to overcome their limitations such as: global barriers, insufficient and too coarse-grained parallelism, and poor latency hiding capabilities (difficulties in orchestrating the overlap of computation and communication).

General Design:

HPX is a state-of-the-art parallel runtime system providing a solid foundation for ParalleX applications while remaining as efficient, as portable, and as modular as possible. This efficiency and modularity of the implementation is central to the design, and dominates the overall architecture of the library (see Fig. 3.1). It exposes the necessary modules and an API to create, manage, connect, and delete any ParalleX components assigned to an application. The current implementation of HPX provides the infrastructure for the following ParalleX concepts: the active global address space, threads and their management, parcel transport and parcel management, and local control objects.

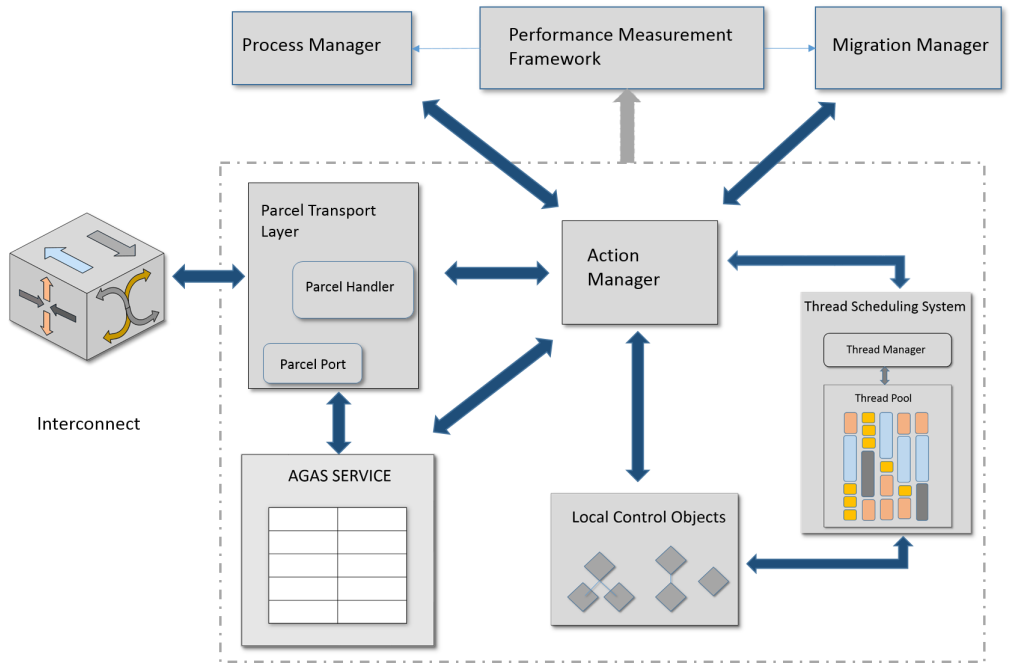

Figure 3.1: Modular structure of HPX implementation. HPX implements the supporting functionality for all of the elements needed for the ParalleX model: AGAS, parcel port and parcel handlers, HPX-threads and thread manager, ParalleX processes, LCOs, performance counters, and the means of integrating application specific components.

Threads and their Management:

The HPX-thread manager implements a work queue based execution model very similar to prior systems (Cilk++ [Lei09], TBB [Int10], PPL [Mic10]). In addition, HPX-threads are first class objects with immutable global names, enabling even remote management. Moving threads across localities (expensive operation) is avoided; instead, work migrates via "continuations" [AJ89] by sending a parcel that might cause the instantiation of a thread at the remote locality. HPX-threads are cooperatively (non-preemptively) scheduled in user mode by a thread manager on top of an OS-thread per core. The HPX-threads can be scheduled without a kernel transition, which provides a performance boost. Additionally the full use of the OS's time quantum per OS-thread can be achieved even if an HPX-thread blocks for any reason.

Parcel Transport and Parcel Management:

In HPX, parcels are an extended form of active messages [Wal82] for inter-locality communication. Parcels are the remote semantic equivalent to creating a local HPX-thread. If a function is to be applied locally, an HPX-thread is created; if it has to be applied remotely, a parcel is generated and sent which will create an HPX-thread at the remote site. Parcels are either used to move the work to the data (by applying an operation on a remote entity) or to gather small pieces of data back to the caller. Parcels enable the message driven paradigm (as developed in TAM [Cul+93], Split-C [Kri+93]) for distributed control flow and for dynamic resource management, featuring a split-phase transaction based execution model. The current implementation features TCP/IP and MPI communication layer as network protocol for the parcel transport system.

Local Control Objects (LCOs):

An LCO is an abstraction of different functionalities for event-driven HPX-thread creation, protection of data structures from race conditions and automatic event driven on-the-fly scheduling of work with the goal of letting every single function proceed as far as possible. Every object which may create (or reactivate) an HPX-thread as a result of some action exposes the necessary functionality of an LCO. LCOs are used to organize flow control. A well known and prominent example of an LCO is a 'future' [FW76; Hal85; BH77]. It refers to an object that acts as a proxy for a result that is initially not known, usually because the computation of its value has not yet completed. The future synchronizes the access to this value by optionally suspending the requesting thread until the value is available. This allows the computation to proceed unblocked until the actual value is required to produce a result rather than, say, incorporating it into a more complex data structure. Futures also permit anonymous producer-consumer computation when neither the producer

of a value, nor its consumer are known at compile time. In addition, the future construct allows a trade-off between eager and lazy evaluations by postponing the calculation of a value until it is actually required.

HPX provides specialized implementations of a full set of synchronization primitives (futures, mutexes, conditions, semaphores, full-empty bits, etc.) usable to cooperatively block an HPX-thread, while informing the thread manager that other work can be run on the OS-thread (core). The thread manager can then make a scheduling decision to execute other work.

Active Global Address Space:

The AGAS service enables the characteristics feature of "Global Address Space", (i.e. distributed global virtual address) for the instantiated first class objects of an application running on multiple localities in HPX runtime system. The objective of AGAS service is to:

- Store the mapping of a globally unique id (GID) to a tuple of local virtual address, component id, and locality information.
- Allocate and assign GIDs to newly instantiated first class objects.
- Resolve the address of an object referred to by a GID in a global address space context.

The global virtual address (GVA), identified by GID, of an object fully and unambiguously describes the location of the object referred by its GID. The GIDs assigned to an object are immutable, hence if objects are moved across localities, they retain their original identity.

Performance Counter Framework:

HPX performance counters provide an intrusive method of instrumenting the environment in which an HPX application is running, exposing metrics from hardware,

the OS, HPX runtime services, and applications. The data provided by performance counters facilitate the development of heuristic algorithms that use introspection to make smarter runtime decisions. A performance counter is first class object associated with a symbolic name which exposes a uniform interface for collecting arbitrary performance data, on demand. Instrumentation utilities can connect to a running HPX application through the parcel transport layer, query the performance counters, and then disconnect.

3.4 Thread Scheduling Policies in HPX

In this section we discuss the different task(thread) scheduling policies that are made available in the HPX runtime system.

3.4.1 Local Priority Scheduling Policy

In the local priority scheduling scheme, the runtime system maintains one thread scheduling queue per operating system (OS) thread (see fig. 3.2). The runtime system channels work to the OS threads via these individual HPX runtime system queues. Additionally, under this policy, the thread management system in HPX maintains separate queues: several for high priority queues and one for low priority queues. When there is work in the high priority queues, it is executed by the first N OS threads, before any other work are executed from the other queues. Work on the low priority queue is executed by the last OS thread, whenever there is no more other work available. When there is an imbalance of work load on threads, work is stolen from other work heavy queues. Further, the scheduling policy user, also has an option to turn on NUMA sensitivity at the command line option. When NUMA sensitivity is turned on, work stealing is done from queues associated with the same NUMA domain first, and then the queues associated with the adjacent NUMA domain/s. The Local Priority Scheduling Policy follows "First In First Out" (FIFO) policy for each user threads assigned by the runtime system to the thread management system.

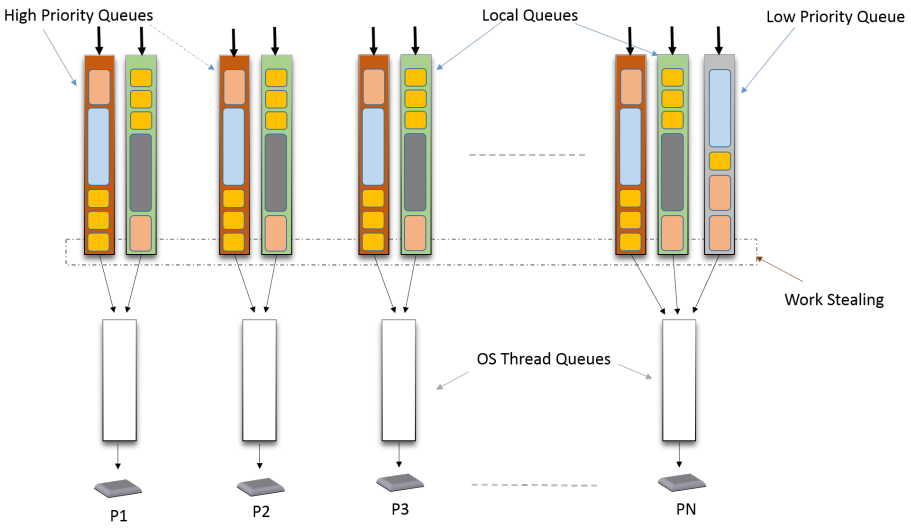

Figure 3.2: Local Priority Scheduling using FIFO Scheme.

3.4.2 Static Priority Scheduling Policy

Under the static priority scheduling policy, the thread manager maintains a work queue per OS thread (see fig. 3.3). Threads (work) are distributed to the queues in a round robin fashion. Like local priority scheduling policy, static priority scheduling policy also maintains several separate queues for high priority threads and one separate queue for low priority threads. Under this policy, there is no work stealing from other HPX thread queues.

3.4.3 ABP Priority Scheduling Policy

ABP priority scheduling policy maintains a double ended lock free queue for each OS thread. Like local priority scheduling policy, ABP priority policy also maintains a high priority queue for each corresponding OS threads, and one low priority thread queue. When a queue is empty, it tries to steal work from one of the non empty high priority queues. In ABP scheduling policy, threads are assigned to the OS threads using "Last In First Out" (LIFO) assignment policy. ABP scheduling policy also allows work stealing from the adjacent user thread queues (see fig. 3.4). Thread stealing happens at the other end of the thread queue, opposite of where thread objects are entered into the queue. Additionally, this thread policy also allows enabling NUMA sensitivity with work stealing first happening within the NUMA domain.

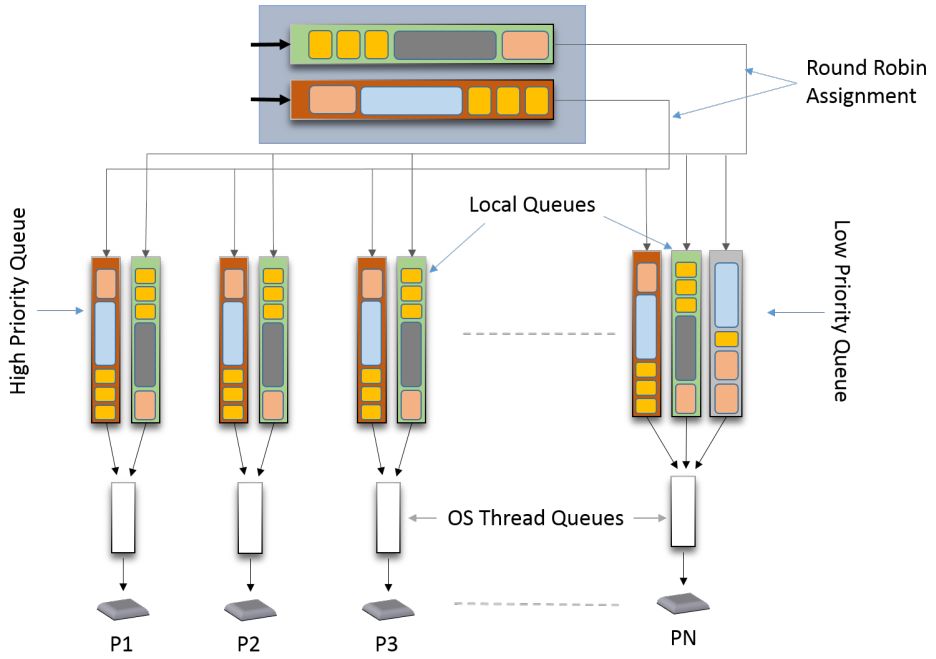

Figure 3.3: Static Scheduling Policy using Round Robin Thread Assignment.

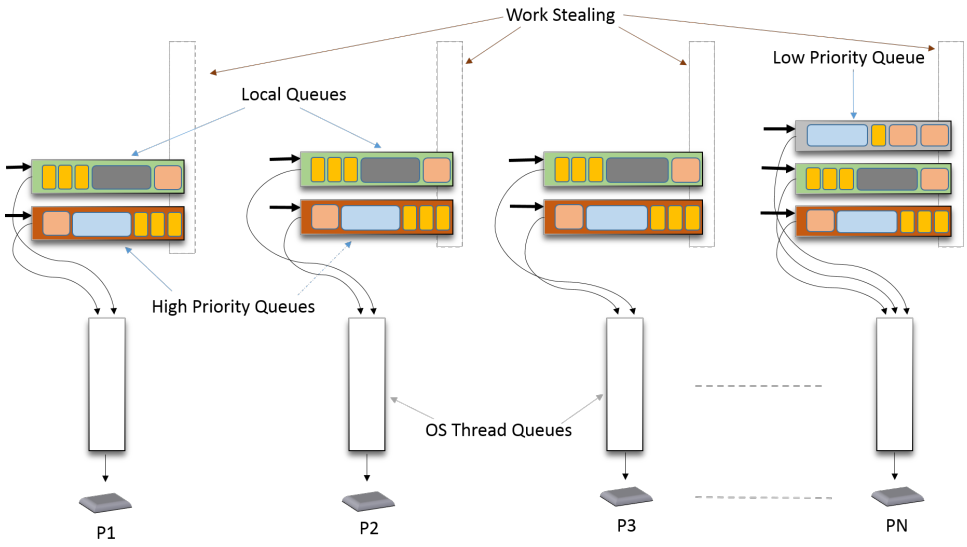

Figure 3.4: ABP Priority Scheduling Using LIFO Assignment.

3.4.4 Hierarchical Scheduling Policy

Hierarchical scheduling policy maintains a tree of ready work (fig. 3.5). The thread scheduler makes the OS threads walk through such hierarchical tree of ready work for the next task to be executed. The arity of such hierarchical thread queue can be varied, with the default being two. Hierarchical scheduling policy also maintains a pair of high priority queues and normal queues of ready work along with a single low priority queue. For work stealing, the scheduling policy targets the parent queue of the hierarchical queues. Extraction of ready work from each queue follows FIFO scheme.

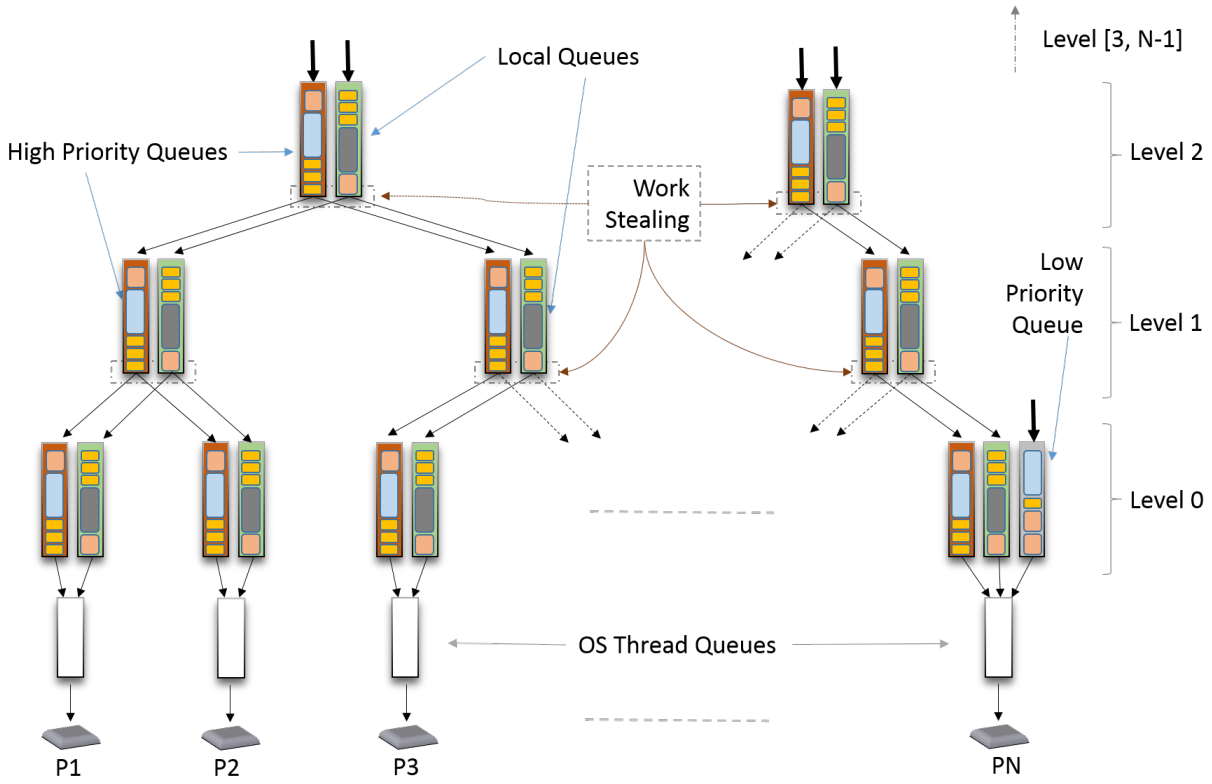

Figure 3.5: Hierarchical Scheduling Policy using Hierarchical Queues.

3.4.5 Local Scheduling Policy

Like local priority scheduling policy, under local scheduling policy, one work (thread) queue is maintained per OS thread, from where the runtime system makes work available to each ready OS threads (see fig. 3.6). This scheduling policy does not maintain additional

category of queues. Adaptive work stealing happens from one of the adjacent thread queues which has enough work in the queue. This scheduling scheme also follows a FIFO thread execution policy.

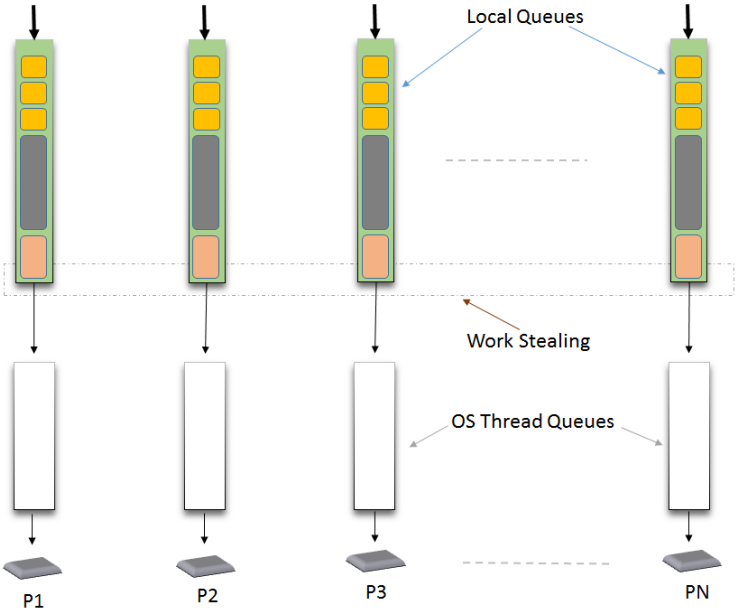

Figure 3.6: Local Queue Scheduling using FIFO Scheme.

4 Parallel Processes and Their Use Cases in HPX

4.1 Background

A computational problem by definition is not sequential. However, how we approach problems for understanding is sequential. So in order to gain maximum flexibility and efficiency in how we represent and solve problems, the problems can be represented or divided into chunks with minimal dependency. After which solving each of these chunks on its own, without following a specific order is fairly possible. For example lets take an unbalanced directed graph problem, as depicted in figure 4.1(A). Sequential execution of the problem would give poor performance. Equally dividing tasks among available resources is also difficult, as we would want to minimize dependency between such segregated tasks.

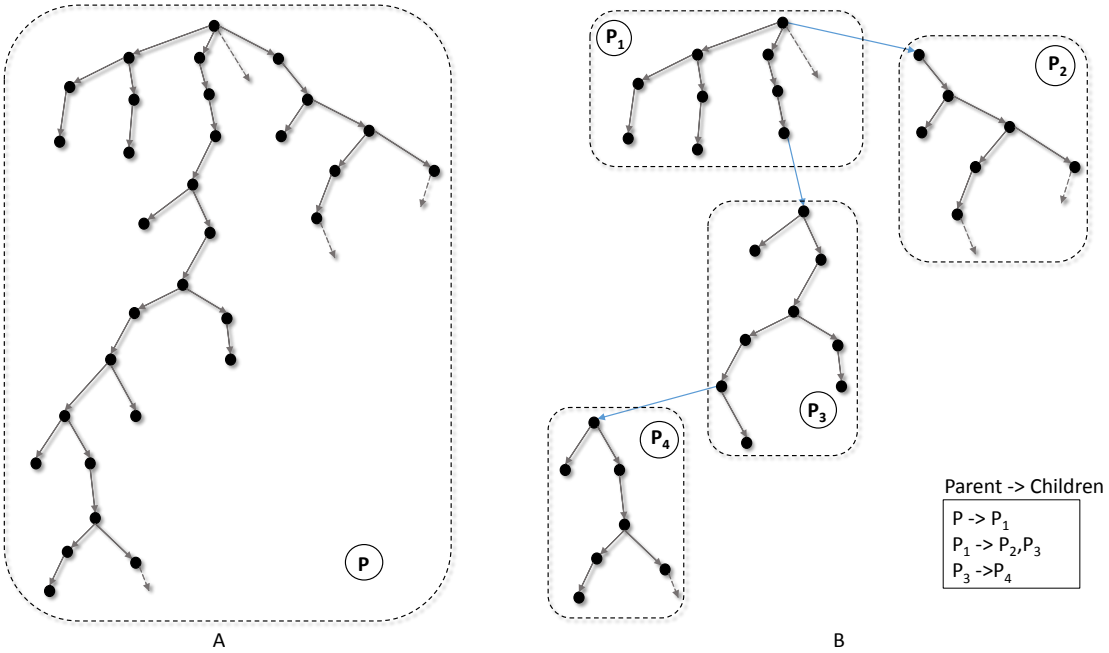

Figure 4.1: Task decomposition of an unbalanced directed graph (A) into sub process domains (B).

One possible solution is to compose the problem into manageable chunks with minimal interdependency. For example in the figure 4.1(B), problem P is decomposed into P1, P2, P3 and P4; where P2 and P3 can be executed in parallel. In addition each process group knows who its parent is as well as knows about its children, hence there is no of information loss between the task groups. These manageable chunks of tasks, which can be executed in parallel, self contained for the problem chunk, and can span multiple domains (for process P), are what we call Parallel Processes. The conceptual elaboration of the Parallel Process and identification of its use cases for computation and resource management in HPX is one of the main contributions of this thesis.

4.2 Parallel Processes

The notion of a Parallel Process in ParalleX/HPX goes beyond earlier definitions of sequential processes [Dij02; Han73] and communicating sequential processes [BHR84]; of being encapsulation of computations (a set of related operations or actions) and the associated result or value if any. Processes in HPX additionally hold static and dynamic information about the logical states of a computation, enabling context aware computing [SAW94]; hence allowing to safely encapsulate concepts and actions that many not be known during static analysis. Such Processes are termed "Parallel Processes" due to the inherent parallelizability of operations as independent fine grained tasks (or threads), or coarse grained tasks (or processes); either of both belonging to a single application's computation structure.

Thus defined Parallel Processes may exist within the context of one physical locality or multiple physical localities, exhibiting different levels of resource oriented parallelism as well. With multiple localities, sub processes of the distributed Parallel Processes would exploit locality aware parallelism (where a sub-process is defined as part of a distributed process, consisting of only a partial segment of the computation data and all the relevant meta-data). Meanwhile, within a locality, although sub-processes can be used to exploit the available resource's multi threaded parallelism, one or more fine grained user level

threads are rather executed on top of each hardware level threads to exploit the task based parallelism [PB10; Wil09]. This is true in the context of NUMA domains in multi-core systems [Bla+10] or Symmetric Processing systems (SMP) [Le+07] representing a synchronous domain.

A Parallel Process may generate children processes, whose existence may be limited to just one locality, or may be distributed across multiple localities, irrespective of where the parent process exists; thus Parallel Processes exhibit a hierarchical structure. This allows ordering of flow control, fast information dissemination as well as access rights control to relevant process objects. Parallel Processes also own the context of threads that are instantiated by them. Through this ownership, special thread instantiation and execution techniques may be applied to minimize concurrency related errors (such as deadlocks, race conditions, atomicity violations, sequential consistency) [Ber+09]. Figure 4.2 depicts the composition of working of Parallel Processes spanning multiple localities, with feedback control from the hardware as well as software resources, with Parallel Processes for measurement framework and Parallel Processes for application working together. Components and attributes that belong to a Parallel Process domain are discussed in further detail in the following sections.

4.2.1 First Class Entity

Parallel Processes in the conventional sense are instances of programs in execution (with predefined order), that have unique names and are individually assigned a fixed contiguous memory space [Han02]. Parallel Processes, during its lifetime, may additionally store logical states of the program at different stages, besides maintaining the meta data about its environment and other interacting Processes. As such, Parallel Processes may require necessary information passed to it, for change in the state of a computation or environment as well as it may return some resulting value when called upon. Therefore, being a named object, an analogy to application variables can be made regarding processes, where a set of function operations similarly applicable to variables also apply for Parallel Processes.

Meanwhile, the state information stored about the "procedure state" of an application segment may change, for example after a "get()" operation. The information thus produced could be used by itself or other interacting processes for resource management purposes. Hence typical operations on processes include being assigned to another named object of same type, being passed as an argument, being returned as value of a function or being queried about program's state information. All these features are only possible with a globally unique name assigned to each process object. While the name assigned cannot be modified, the process state captured by the state variable within the process, may be modified by the process itself or by other processes that have privileged access.

4.2.2 Process Members and Attributes

A Parallel Process by itself is just a named data object. Additionally, Parallel Processes could have other processes as members, a collection of process threads (user level threads), data structures that hold:

- application related information,
- process components related information, and
- process state related information.

Access to any of these data structures requires fulfillment of a set of predetermined requirements. The Parallel Process also encapsulates a procedure description for a program segment [Han02], as a collection of functions, which can be called in a particular order from another process that it is tasked within a given control flow. Further processes are also capable of maintaining a list of resources that they are in control of, such as synchronization objects (e.g. LCOs), communication objects(buffers and queues), policy interface objects, access rights list, etc. The collection of all such different components may be termed as process attributes; a set or combination of which a process can use to fulfill its procedure description (the mandate of a Parallel Process). Some of the process components and attributes are discussed in more detail in the following sections.

4.2.3 Process Threads

Computation Abstracts, or conventionally threads, are part of a procedure definition of a Parallel Process that contain all the necessary information for a meaningful execution of the given subroutine (full or part of the procedure definition) [DDH72]. Instantiated computation abstracts, with the data, subroutine definition and execution context, termed "process thread"' is a unit object that gets scheduled by the runtime system for execution, which is further handed down to the operating system scheduler to be executed as machine instructions. Process threads may also be treated as first class objects, in that, they also are assigned a unique address and a name. So, semantically they can be called light weight processes (slightly different than Linux light weight processes LWP) [BC05], in that they only hold part or whole of the procedure definition and access rights to relevant data structures within a given process or in other processes, whose value is to be delivered via messages. Threads exist as an instance of execution context of the procedure definition of the encapsulating process, making a Parallel Process owner of the threads created by it. The main difference between the process and the thread it owns is, the process may span multiple localities, while threads are strictly local, for obvious reasons.

Process threads can be of varying granularity; fine grained when available concurrency is higher and correspondingly coarse grained when software overhead in managing concurrency is higher than simple bulk computation. Again, the dynamic adaptive granularity size control of threads may be based on user fed information or through dynamic performance measurements of tasks. Which mode to choose between can be decided using a thread execution policy, defined beforehand. Such adaptive granularity change of threads boosts performance when parallelism is available, and minimizes overheads when less parallelism is available [TJF13]. Further, different thread scheduling schemes may be applied, which again is defined by a thread scheduling policies. Further, Process threads may be responsible for instantiation of other process threads as well as processes. Process threads have different execution states: ready, running, suspended (waiting, blocked) or finished.

4.2.4 Local Control Objects

The semantics of an LCO has been discussed earlier as one of the fundamental concepts of ParalleX Execution model. Here we discuss its relationship with processes. An LCO object is a synchronizing construct that exists within the context of a Parallel Process and not the other way. Hence it is local to the process. LCOs like Parallel Processes are first class objects, hence they are movable, mutable, copyable, directly referable, etc. within the context of the process that instantiated them. LCOs, in different instantiation scenarios, would enable Parallel Processes to synchronize operations on different types of resources the processes own [Hoa74].

As mentioned in section 4.2, a Parallel Process may exist across multiple localities. LCOs enable dynamic global synchronization and migration of objects across such distributed localities. Like nested semaphores or mutexes [CHP71], LCOs can also be organized in a nested order for multilevel task parallelism synchronization [Ayg+06]. Hence, if there are resources that are shared globally with other processes or computation objects that exist across multiple localities, a global LCO object, existing within the parent process or one of the sibling processes, could coordinate the shared usage of the resource. Generally, availability of resources depend upon the lifetime of a Parallel Process, however, it might be permitted to transfer the ownership of a resource to another process, under certain circumstances, like migration of objects.

4.2.5 Resource Ownership and Lifetime

A Parallel Process abstraction by definition allows a state change of some resources. It contains all the relevant information and features for safely controlling and managing a resource. Thus a process abstraction can have exclusive ownership of various resources, that instantiates a given resource, where a resource can be a physical hardware entity or just another software entity(eg. buffer, software cache, etc) [CYH10]. The lifetime of resources used by an application is dictated by the process that instantiates the resource. That process may belong to the application itself which uses the resource or may belong

to some other application. The resource owning process may give access rights to other processes, and synchronization amongst many interfacing processes is achieved through an LCO or a collection of LCOs. Access rights to the resources is defined by various policies for resource usage. The resources become inaccessible once the process owning it goes out of context. Under special situations, like migration of processes, ownership of resources may be transferred to other processes, ensuing the transfer of any existing LCOs pertinent with the transferred resource(s) as well.

4.2.6 Process Communication

Inter Process Communication (IPC) is one of the methods of exchanging information between processes for synchronization of tasks [Hoa78]. A Parallel Process may communicate with any of the children processes in a given process hierarchy relationship or with other independent processes in a process graph. In context of HPX, with message driven execution model semantics, IPC between Parallel Processes take place through parcels, an implementation of active message [Eic+92] in HPX. Meanwhile IPC within a given locality could be through any predetermined input/output mechanisms, such as method invocation, pipes, data copy, semaphores, message queues etc., [SR08].

4.2.7 Access Rights Protocol

Without an access rights policy for a given set of resources, software or hardware, safe usage of resources cannot be guaranteed [JT01]; particularly when the entities(processes) owning the resources are capable of migration to a different locality . Object oriented, message driven programming model allows us to categorize different processes with different roles, establishing relationships such as parent-child, master-slave, producer-consumer, etc., for flow control as well as guidance on resource usage [Chr88]. Associating access policy based on the role assumed provides a better manageability and security amongst the participating entities [BBU99]. In Exascale scenario, where there could be hundreds of thousands of processes, each with a different role, a fine grained access policy is warranted, where common objects take similar roles [Fis+09].

In context of Parallel Processes in HPX, processes may give full access rights to the threads, on the data stored by a given process. Children processes in a hierarchical structure of processes may be given access rights to the data held by parent, as inheritance feature [PH00], however access rights to data from sibling processes, and hence their encapsulated threads is not implicitly guaranteed. Also, since the parent process controls the lifetime of children processes, access rights to the children data may be implicit, through ownership semantics. However the reverse is not implicit. These criteria are with the notion of direct access of data into perspective. Access to data-structures maintained by resident threads of the processes, children processes and sibling threads may be provided indirectly. Having a well defined access right protocol on different components of Parallel Processes provides us with at least a degree of protection against data corruption due to unintended data access privileges.

4.2.8 Distributed Data Structure Abstractions

Data abstraction of the information pertinent to the context of a procedure definition of a process is one of the principle features of Parallel Processes. Data abstraction through Parallel Processes allows data of various types, even user defined types, to be encapsulated and protected, allowing little or no side effects [Lis87; Sco09]. With data abstraction in place, processes can make data available to other entities, such as local threads or other processes, via parameter passing, IPC or parcels. Data abstraction is, thus, one of the distinguishing difference between threads and processes, as threads by definition lack sophisticated data abstraction mechanisms [AT88]. Further encapsulated data by Parallel Processes may not only be related to the context of a given computation, but could comprise of important information that help the application or runtime system manage the computation.

There are several different approaches for facilitating a scalable distributed data structures such as distributed hash table based, range partitioning, relaxed balanced search tree, distributed B tree [PN04], skip graphs [Sha03], global trees [Lar+08]; with different spatial

and temporal complexities. As such using a particular method for scientific applications as a universal solution has been elusive. Meanwhile majority of scientific applications use array, graph and tree based data structure [WL98]. The distributed data structure in our case would be analogous to a segmented array that can span multiple localities (nodes), represented by process objects. The interface made available for the distributed data structure is similar to accessing and manipulation an array. Further, the hierarchical properties, that are natural to processes, allow straightforward creation and management of trees and graph like data structures.

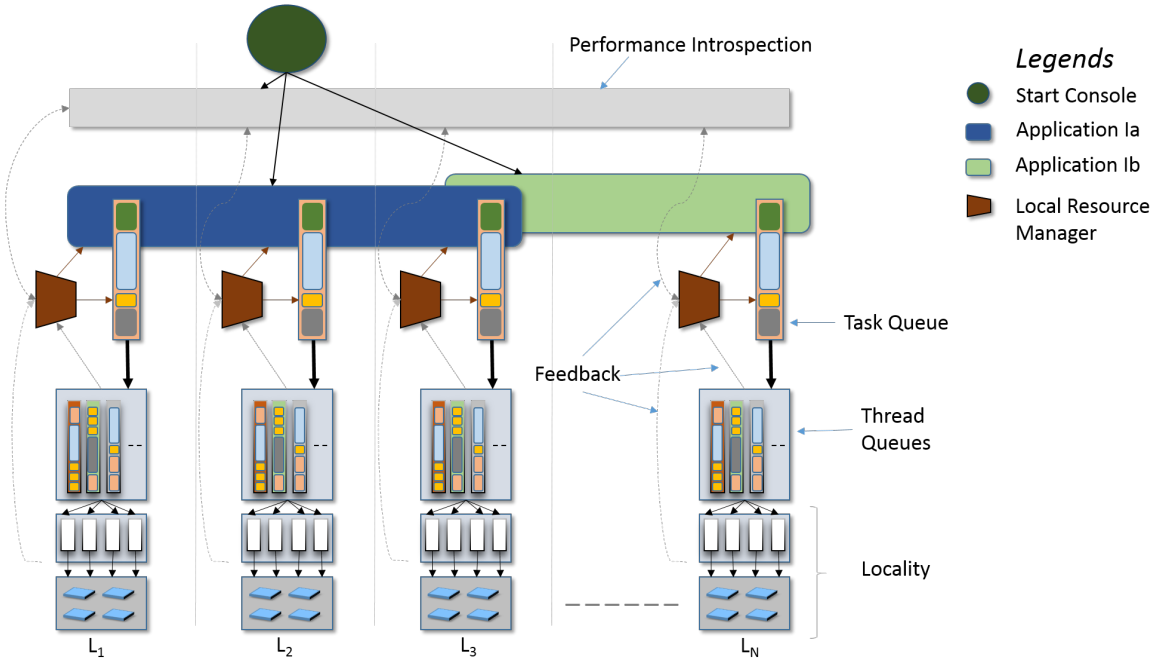

Figure 4.2: Parallel Processes, through distributed data structures, are able to span multiple localities(physical nodes), and this set of localities could be dynamic. Each Parallel Processes supports certain key attributes such as task scheduling policies, access rights, interface to kernel handlers.

4.3 Resource Management Framework

Resources can be categorized as a) hardware entities (such as physical processing elements, peripheral devices, etc.) and b) software entities (such as task queues, message buffer, data, data files, etc.), that need to be shared with other entities. While operating systems represent and manage physical or software entities as logical entities [Hoa74], a

runtime system provides additional features and proper API for using such logically represented resources according to an execution model [App90]. A runtime system targeting Exascale systems has to manage and provide a seamless interface for using a billion way concurrent resources. While efficiently making the computational resources available to the consumer applications is one aspect of resource management problem, the other aspect being effective distribution of load on the resources for efficient use of the resources.

Several aspects of an execution environment have to be considered for a resource management scheme to be an effective solution. Factors playing significant role in successful deployment of a resource management solution require runtime dynamic data about hardware performance, static and dynamic information of application characteristics as well as user requirements. Resource allocation not only depends upon availability of resources but also on the access policies defined for such resources . Further, a fine grained resource scheduling policy and control is desired for scenarios where there are two execution frameworks working interdependently or cooperatively taking the data locality concerns into account [Hin+11]. HPX processes can act as a foundation for a resource management framework that encapsulates data, provides a performance measurement framework, preserves user fed application specific meta-data, provides modular and fine grained object oriented scheduling platform that uses locality and application load information for task granularity control as well as task migration [Fis+09; WW96].

4.3.1 Resource Management

Resource management activity can be viewed from two perspectives: a) from the dynamic state of the system and b) from the dynamic state of the application (task). Performance and scalability of application through efficient utilization of resources depends not only on how applications are intelligently mapped to the resources, but also on how resources are intelligently managed based on runtime performance data as well as application's dynamic characteristics [Cao+02]. Based on this notion of resource management, the act of managing resources can be sub-categorized into the following types:

- **System Resource Management**

  Current architecture trends of many-core architecture [Int; Til] hint the future scalable parallel computing system will have many-core architecture with thousands of cores per chip under economically viable energy footprint [Bor07]. However future systems will comprise of not only many-core architecture but also a host of other different architectures, providing a heterogeneous computing environment [Dan+10]. Although both architectural paradigms allow more parallelism, they are also prone to introducing overhead in the form of hardware resource coordinations, if there is no optimal mapping of application tasks to available resources. While mapping available compute cores to application's computation entities(tasks) is a NP hard problem, with increasing compute cores; efficient resource allocation, with minimal communication overhead is nonetheless an activity that determines ultimate throughput of a system [Mar+07; Kob+11; Jah+13].

  The purpose of a system resource management service would be to monitor available resources' performance and availability, and use that information in allocation of the logical representation of system resources to the pending tasks, encapsulated in Parallel Processes objects. Such a resource coordination framework that allows an object oriented approach in coordinating resources not only allows a decentralized resource management scheme, but also allows deployment of modular resource management and usage policies [WW96]. Fine grained resource sharing between frameworks would allow tight integration of interacting applications (or application frameworks) that are concurrently utilizing the compute resources [Hin+11].

- **Task Management**

  Task Management can be categorized as software resource management activity, where management of tasks(threads) associated with relevant queues is done, using information such as task priorities, task dependency fulfillment, and preparing

them to be assigned to the available compute resources. One of the principal actions for task management service is task granularity control. Granularity control of tasks can be achieved both statically as well as dynamically. While static granularity control allows dedicated usage of resource, when the work load is balanced; dynamic granularity control [Car+09] allows adaptive work load balancing of tasks where the application's tasks are highly uneven, such as in dynamic graph applications.

However, all the procedure definitions of an application are not equally parallelizable. Some sections are more parallelizable than the others due to different dependency issues. Adjustable assignment of task granularity depends on malleability of an application. Such adjustment could be primarily based on application's parallelizability characteristics and secondarily on hardware performance characteristics, coordinated by the system resource management service. Having such feature for task size recomposition allows efficient allocation of compute resources to more scalable tasks [SLS07], rather than just using a backfilling technique or priority based scheduling [Bar+08]. The fundamental task for this service layer is to figure out the appropriate task granularity for optimal performance.

- **Locality Management**

  Running huge applications with huge datasets is becoming more possible with the advancement in system hardware technologies and scalable execution models, with decreasing overhead as well as lesser energy footprint [Hu+14]. However, not every aspect of the system hardware show linear scalability, which is particularly true for memory. The current trend of memory design paradigms in the HPC systems architecture will to a greater degree continue into the Exascale era, meaning some form of memory hierarchy will be preserved in future systems. Thus the way we write applications will imbibe principles of temporal and spatial locality principles. Although, communication speed has also been improved with each new generation

of communication infrastructures tailored for HPC systems [Pan14], the volume of data explosion in applications is too big for advanced communication infrastructure to compensate. Hence data locality optimization is fundamentally important topic for such applications.

Significance of data locality is, however, not only limited to big data applications. Task scheduling without analysis of data sharing pattern could lead to serious performance degradation [CE94]. Better locality behavior minimizes communication and data management overheads. There are several compiler based solutions as well as optimization algorithms that result in better locality behavior such as hierarchical tiling [RK+13; KRC97; Bik+06]. The newer programming models also enable better locality preservation for applications by allowing placement of relevant tasks as close as possible, along with the feature of task movement, rather than data movement.

Additionally having the capacity to deploy user fed policies for task placement adds another dimension in terms of data locality improvement, as users knowledge on application heuristics can be directly exploited. Such feature allows an intelligent task scheduling mechanism for placing relevant task on resources as close as possible, rather than just relying on scheduling algorithms locality heuristics [Rin08]. Parallel Processes in HPX provides such means of providing intelligent information about data locality for task management service, for scheduling tasks on system resources. Parallel Processes implicitly also allows hierarchical grouping of tasks, that could represent different levels of data locality for the underlying hardware device [Fat+06; Bik+06]. Further, the workflow semantics that the Parallel Processes can facilitate, would naturally co-locate new tasks, whenever created.

4.4 Load Balancing Framework

Getting better scalability of any parallel application requires careful load distribution across all the available resources. Applications that exhibit good scalability characteristics are often applications that can be partitioned into even chunks and whose load character-

istics is somewhat static [KGV94]. On the other hand, applications that cannot be evenly partitioned into regular chunks and which has dynamic load characteristics are often difficult to scale. While scalability of applications depend primarily upon characteristics of the applications, the architecture of the resources also play a significant role in determining a desired result. A load balancing framework thus play an important role in realizing the desired goal of achieving better resource utilization.

In a billion way parallelism offered by future multi-threaded heterogeneous systems, applications would also be targeted with hardware based scalability solutions, with usage of dedicated hardware resources for certain tasks. While management of availability of hardware devices is coordinated by system resource management framework, load balancing framework targets application subtasks to hardware resources based on information provided by performance measurement module about application performance and resource availability as well as based on application attributes.

Among many load balancing schemes, randomized work stealing mechanism for threaded systems introduced in Cilk Runtime System [Blu+95] has been thus far the most popular mechanism for load balancing. However future Exascale systems either using a centralized scheme or decentralized scheme cannot always guarantee optimal performance. Thus, a measurement based dynamic load balancing framework in decentralized systems, with some form of global and local control mechanism is preferred [Las+13]. Some popular schemes for load balance are sender/receiver initiated where the roles may interchange, hierarchical balancing scheme, gradient based, dimension exchange methods, global round robin, asynchronous round robin, graph partitioning, space filling, prefix sum and many more [SK93; WLR93; KGV94; AB12; Zhe+10]. Each of these strategies require different evaluation criteria to be fulfilled to be an optimal candidate policy for load balancing.

Load balancing may be achieved through thread migration across NUMA domain within a synchronous domain and computation(action) migration across synchronous domains. Efficient usage of either mechanism depends upon understanding the application

locality properties along with the physical system and its interconnects as well as task priority attributes [HWW93; SK93; KC98]. Such knowledge allows finding an appropriate task with right grain size that may be migrated. Dynamic granularity control of tasks for load balancing is important for scalability of important class of applications that require adaptive dynamic task reconfiguration.

With so many different parameters that need to be taken into consideration for different applications, it is difficult to come up with a one technique fits all solution. Hence a load balancing scheme with modular policy deployment scheme [BC03], based on the dynamic heuristics of different techniques for relevant applications gives a more versatile infrastructure for load balancing of application tasks. Parallel processes in HPX, provides this modularity option through its object oriented design. In addition, Parallel Processes' composition of tasks into process objects and its thread objects give reconfigurable task granularity on the fly as discussed in the above section. As task dependency ordering information can be stored in Parallel Processes objects, load balancing with efficient locality control may be achieved, through task co-location mechanism.

4.5 Access Control Framework

With sophistication of application management mechanism, through plethora of control semantics and intelligent dynamic information analysis capabilities, applications are becoming more autonomous with their own increasing decision making capabilities. Such sophistication in application management frameworks are opening new opportunity of harnessing Exascale resources, where applications of different domains may execute simultaneously, interacting with each other for their respective specialty (e.g. multiphysics applications). Multi-domain application simulations provide an automatic and holistic approach in solving a problem, rather than just using a single domain specialized applications, as solution approaches of different scientific domains are usually different from others [Ama+11a]. Such possibility necessitates a mechanism to control guaranteeing a controlled and safe exchange of information.

Intelligent adaptive autonomous agents of the interacting applications could use different mechanism for access control such as role based access control (RBAC), attribute based access control (ABAC) and more [Fin+08; Sto11]. These mechanisms can appropriately be represented by the process objects, with their intrinsic properties, adapting to the requirements of such mechanism. The Parallel Processes' granularity control feature would permit a granularity oriented access control of resources [Fis+09], where relevant objects may be chunked, with a collective access rights defined for the whole group of objects.

4.6 Namespace Management

Parallel systems of Exascale and beyond would enable a billion way parallelism which can be exploited with equal number of computation entities [SDM11; Agh86]. An object oriented approach for representation of such computational entities for an efficient management of parallelism has been a popular approach [Bal+97; KK93; KBS09]. Naming the objects provide a transparent and homogeneous identification of resources, allowing a better flexibility for controlling such resources, while hiding resource diversity and intricacies [Les03]. Thus having an efficient and scalable naming system would result in an efficient and scalable control of the named objects, which translates to superior application scalability.

Based on the relation of objects with other entities that they interact, namespace management may take different form. To name a few, a fast lookup service with homogeneous distribution of objects across localities may be represented using a flat namespace or using hash tables for quick access. Namespace management based on topographical view of objects can provide fast location information about objects. Hierarchical namespace management scheme is a popular method, that additionally allows locality conscious object management [Les03]. Each of these naming schemes have their own advantages and disadvantages. Based on application requirements, an appropriate scheme can be used. Name management services practiced in CORBA, DCE, DNS [Hen06; Lei99; Ter+84]systems follow a static structure for the namespace management, with the premise that named objects

do not often move; which serves a location based naming requirements. Meanwhile, other naming schemes, such as those followed in global address space systems need to follow a naming system that provides a unique name to an object, for the lifetime of an application. Such unique naming allows uniform accessibility of objects in global address space, with minimal overheads, despite the complexity and heterogeneity of resources involved.

Parallel Processes allows a flexible way of object management through its inherent properties. While a separate bookkeeping system of object names corresponding to different attributes of the object may allow fast resolution of object for a particular computational requirement, it is not always optimal. Such bookkeeping necessitates maintenance of different namespace database based on different attributes of objects. This method not only requires more space but also cannot guarantee scalability of the naming scheme. Parallel Processes on the other hand maintain a structural ordering of relevant objects, based on predetermined requirements and clues when the objects were created, with all the object attributes in present Process's meta-data. Name resolution based on object attributes is just a hierarchical traversal of relevant intermediate objects. Such attribute based name resolution has benefit not only with context oriented namespace management but also with security of objects [Les03].

4.7 Performance Measurement Framework

An application that is exposed to a billion way parallelism requires several different parallelization techniques for scalability and to minimize performance bottlenecks due to SLOW. However efficacy of such parallelization techniques can only be determined through extensive performance measurements. Through performance measurements, we can analyze if the performance bottleneck is due to misplacement of objects of an application that is sensitive to locality [WKN], or due to system's architectural limitations in providing scalable memory or network bandwidth. Insights on the cause of bottleneck provided by such measurements can then be used to figure out overhead minimization techniques.

True evaluation of performance characteristics in a highly parallel distributed environment requires a scalable performance measurement framework that can grow on par with the scaling of the application as well as the resources [BNM03; LMK08]. Not only does such performance measurement framework need to be light weight and efficient, causing minimal perturbation to actual application's performance, but also has to manage massive volume of resulting measurement data [Hel+03]. An application profiler can fetch hardware performance counter information as well as software parameters that would help post analysis of application's performance in figuring out acute bottlenecks in software execution [SM06]. Additionally, the information fetched through such analysis can also be used to predict optimal performance configuration of different systems as well as software parameters using techniques such as universal scalability law [Gun08].

Above and beyond post run analysis of applications, performance measurement framework is also crucial during the application run itself. The runtime system employs different scheduling and management techniques on tasks, messages or data items for effectuating an optimal performance of an application. These techniques are based on performance criteria, such as task priority, deadlines that components of a running application must meet [Kim+00]. A well-integrated dynamic performance measurement infrastructure efficiently measures active runtime dynamics of application elements. Information gathered from such dynamic run can be used by the scheduling and management infrastructure to continually optimize application performance; effectively creating a dynamic feedback control system [AKN04]. Such feedback system with fine grained performance tuning mechanism would require less user intervention that a post run analysis method would otherwise necessitate.

Parallel Processes provide an effective platform for achieving both the roles, as a placeholder for collecting data and center for feedback control system, of the performance measurement framework. The performance measurement related entity that a Parallel Processes would represent can be integrated as a cooperating part to the computational com-

ponent of the application's Parallel Processes. Such tight coupling of computational and measurement framework allows a fine grained analysis of performance of an application. Scalability of the ensuing performance measurement framework would be on par with the scalability of the application, yet maintaining a minimal footprint on system resources. To avoid communication overhead through transfer of large volume of performance related data, only minimal relevant information can be transferred to the final aggregating component that analyses a particular aspect of performance bottlenecks. The same infrastructure would again be useful for effective dynamic fine tuning of the application at runtime, since the performance data is recent and fine grained.

5 AGAS Design Guidelines

The characteristic features of AGAS service and its purpose has been discussed in chapter 3, sections 3.1 and 3.3. To enable scalable dynamic relocation of objects in a global address space context, AGAS needs to be a distributed service. In addition, distributed scalability of Parallel Processes and its functionality is directly dependent upon an scalable AGAS service, as each Parallel Processes entity is a first class object. As such, to realize a scalable and robust AGAS service in a distributed heterogeneous system, certain design requirements need to be fulfilled. Such design requirements include: address space scalability concerns, latency concerns, guaranteeing correctness of name resolution, fault tolerance, and associated attributes for object migration. In the following sections, we discuss some of the requirements that need to be addressed by an implementation of AGAS service. While these guidelines are generic, and apply for any distributed systems, an attempt is made to be as specific as possible, for the AGAS service.

5.1 Distributed AGAS service

If the current hardware architecture design trend is an indicator of what will be a paradigm for distributed computing, the global address space for the runtime system needs to span across hundreds of thousands of cores and accelerators on thousands of conventional nodes. Future generation HPC applications would also be equally nontrivial in complexity and size, where multi-domain interacting applications would be the norm [All+08]. From an object oriented programming context, this translates into billions of compute unit objects. An effective and efficient address resolution of large volumes of named objects require AGAS services to have multiple distributed servers, coordinating address resolution across the entire address space. During the implementation of such distributed servers, some design

questions that may need answering can be: a) Should all the data, the AGAS servers maintain, be always loaded into memory or can some part be offloaded into persistent storage, for retrieving it later?; b) What policy should the AGAS service take to remove the reference to an object that no longer exists, across the AGAS table instances?; c) What impact, integrating access policy into the AGAS service, will have on performance of AGAS service, and what is the best possible alternative to enable access policy for namespace management?; d) For unified application framework, where two interacting applications of different domains run at the same time, will having a single AGAS server for both applications give better performance or will having two separate AGAS servers a better approach?; and e) What will an efficient distributed service layout be, that is, whether the AGAS tables will have single unique tables or is replication of tables allowed to enable software caching of the table entries?

5.2 Asynchronous Coherent Table Management

The primary goal of AGAS service is to maintain a distributed table that stores the mapping of a unique global id (GID) to corresponding tuple of local virtual address (LVA), component id, and locality information. Additionally AGAS service may maintain a table of GIDs with corresponding symbolic names. For sanity of data maintained by AGAS tables, the load/store operations need to be atomic, while performing the whole address resolution task as an asynchronous operation resulting in an overlap of synchronization overheads with communication overheads. However, depending upon performance requirements of fast address translation of neighboring objects versus always correct address resolution; and using design decisions mentioned in section 5.1, an appropriate coherence model can be identified and employed for the distributed AGAS servers [Tan94].

5.3 Correctness

The AGAS service must make certain guarantees about the accuracy of its address resolution service, based on locality of an object. In this regard, AGAS service must answer two fundamental questions: a) Is the object local? and b) If the object is not local,

where can it be found? The degree of correctness to which these questions are answered may be relaxed depending upon performance requirements in a distributed environment; as the information in one end of the distributed system may take considerable time to be relayed to the other end. This also depends upon the coherency protocol decision taken, as discussed in section 5.2. Despite of whatever relaxation scheme is followed for correctness, a minimal requirement for the AGAS service that needs to be esured is about the query for local objects. If a query on an object originates from a locality where the object itself resides (i.e. a local query), then the resolution should always be correct. Meanwhile it would be permissible for the first query resolution be negative, for an object, even if the object is indeed locally present. Correctness does not mean that AGAS must always give the correct answer, it merely means that the circumstances in which AGAS may answer incorrectly are well defined.

5.4 Fault Tolerance

Being a namespace management service for distributed resources, system or application related, that span multiple localities, the AGAS service has to take account for certain failures such as response failure, halting crash,byzantine faults, etc., [Cri93; Sch90]. Depending on the design decisions taken, as mentioned in section 5.1, the layout of the distributed servers could be distributed client-server [BNM03] model, P2P model [Sch01] or other distributed service architecture. To address different kinds of faults, AGAS can follow different protocols, through redundancy (data replication) [GS97] and checkpointing [ZHK06; ZSK04] measures. Based on different fault tolerance mechanism followed, a response protocol should be followed in the event of a failure. Using real-time constraints such as response deadlines and expected success probabilities [KMR08], AGAS service can restart the concerned server in a different locality or migrate the contents of the previous AGAS server to several other live servers. Such mechanism augments the fault tolerance mechanisms already existing for the runtime system, hence improving scalability in future Exascale systems.

5.5 Features for Object Migration

Migration of first class objects, i.e. objects with names, is a distinguishing feature of the AGAS programming model when compared to other popular global address space programming models. Migration is the process of moving arbitrary objects from a constrained execution resource to a less constrained resource, when possible. For an effective object migration feature, the migration mechanism relies on three entities:

1. Serialization Service, which provides a means of flattening any data structure into a sequence of bytes and to recreate this data structure from the byte sequence at the receiving end.

2. The Parcel Transport Layer, which provides a fast means of transporting data across localities.

3. Migration Decision Service, supplies automatic dynamic resource management based on continuous performance metrics on the constrained resources, for better application scalability and efficiency.

For the object migration feature to work properly, an efficient mechanism that ensures immutability and uniqueness of object names (GIDs, and symbolic names, when in context) should be in place. This feature eliminates additional AGAS service overheads by avoiding tracking name aliases of objects to be migrated.

6 Experiments and Results

6.1 Applications

The experimental applications used in this thesis have been implemented in HPX with multi-threaded and distributed version of two different types of applications, a dynamic graph type application and stencil based application following a Bulk Synchronous Processing (BSP) model. However the experiments for the applications have been conducted on single nodes to meet the objectives outlined by this thesis.

6.1.1 UTS Benchmark

One of the benchmark application chosen for verifying the theory proposed on utility of Parallel Processes in the above sections is Unbalanced Tree Search (UTS). UTS is a well studied benchmark for dynamic graph problems; and has been used to study the performance characteristics of a variety of parallel systems as well as programming models [Oli+07]. It is a notoriously difficult to scale problem. This synthetic benchmark however mimics characteristics of several scientific research fields, such as informatics, social sciences, etc. Figure 6.1 shows and example of how a UTS graph may grow.

UTS is a tree(graph) traversal problem, where the nodes in the trees of the artificial benchmark are implicitly constructed, using a process named Galton-Watson Process [Har63]. In this process, except for the root node, every other node is generated using the information obtained from the parent, applying a cryptographic hash function on a combination of parent node's attributes and child index (from a fixed children count), for deterministic results [Oli+07]. The constructed nodes need only be retained while on the depth first search stack. Although the distribution of subtree sizes is same for each node, the generated children number varies widely; with frequent small subtrees and occasionally

very huge subtrees. For fast processing of the graphs, that is, traversing each nodes in the graph, there is no benefit in choosing one node first from the other as the expected size of the search space is same. This feature makes UTS similar to any real dynamic graph type problem. So the performance characteristics observed with different settings can be directly related to real world settings.

Since each node is self-contained, i.e. has all the information to create its children, parallel tree traversal is possible. For parallel tree traversal, a set of empty stack or queue is generated and assigned to HPX threads, and an initial root node is assigned to any of the instantiated stack. As the initial stack gets heavier, other HPX threads with empty stack, initiates stealing work from the heavy stack. Traversing the nodes in the stack by the thread is continued until its stack empties, after which the thread becomes idle. Such idle thread initiates, load balancing by moving one or more node(s) from non-empty stack of another working thread to its empty stack. Hence, work stealing is initiated by idle threads, without active participation of victim thread.

Work stealing is done in chunk sizes, granularity of work stolen, with larger chunk sizes amortizing synchronization overhead of work stealing in the exploration of the 'j' nodes. However, the expected number of children to be generated for each of the j-nodes cannot be guaranteed, although the probability of children count for each node is same. So, there is no advantage in selecting one node over other for stealing. Hence the choice of chunk size has to be a balance between load imbalance and communication cost.

The implementation of UTS in HPX closely follows reference implementation [Uts], implementing two work(node) queues, local queue for extracting work and storing work, and shared queue for for work stealing for load balancing.

6.1.2 MiniGhost

Minighost [BVH] is one of many scientific application proxies presented in Mantevo Project [Man], that is self contained and feature fundamental performance characteristics of the class of applications it belongs to. Minighost uses finite differencing method to solve

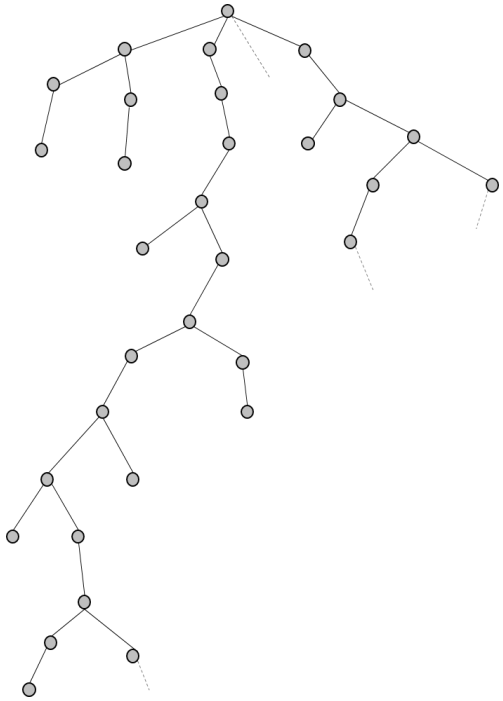

Figure 6.1: Example representation of Unbalanced Graph generated by UTS application.

equations, using difference stencils to update the grid as a function of each point and its neighbors, within a given discrete time step. The evaluation method of finite difference method naturally super imposes onto parallel processing architecture and single program multiple data (SPMD) programming model.

The computation of finite difference equations can be represented as stencils of grid points, where each evaluation in a given time-step depends upon the corresponding values of neighboring grid points, and/or corresponding values at the same grid point in previous time-steps, as depicted in figure 6.2. On parallel systems, such stencil based problems can be distributed as subproblems on available processes, with each process working on a section of the total grid points. Each subproblem computation requires data exchange from its neighboring processes, called boundary exchange. Transfer of relevant data to the neighboring processes happen via inter-process communication.

The conventional approach for such boundary value exchange has been bulk-synchronous parallel programming model [Che+96], where barrier synchronization happens at the end of each time step evaluation, so that each grid point is at the same phase of the computation,

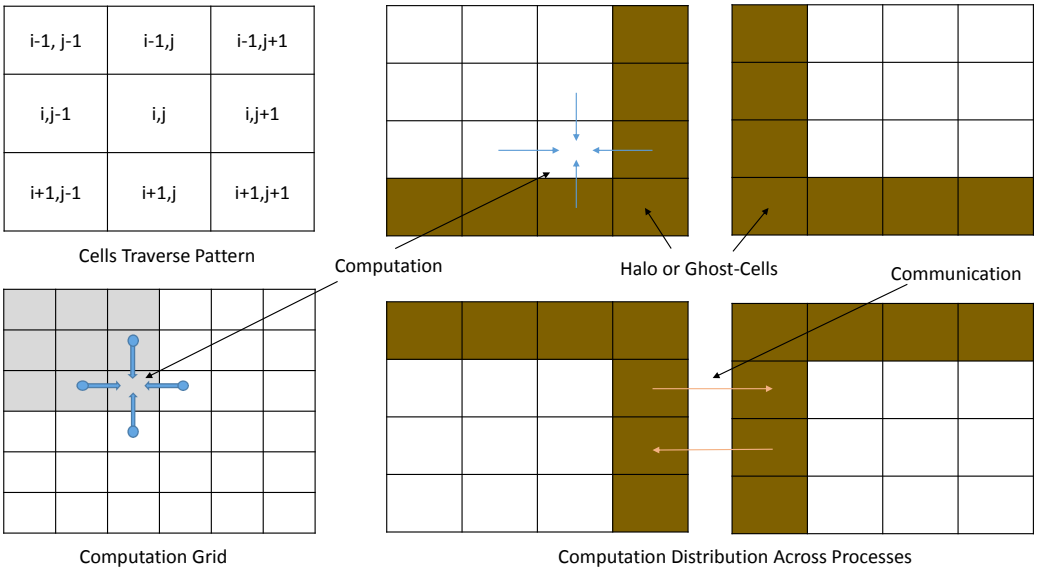

Figure 6.2: Representation of Stencil Based computation with halo exchange regions.

as depicted in figure 6.3. Since, every point does not necessarily take the same amount of time, such barrier synchronization imposes significant overhead due to processor(/thread) idling. Although, different data-aggregation techniques have minimized inter-process communication, allowing scalable performance uptill now; at Exascale problem size, both from application as well as systems perspective, scalability seems to be a big challenge [BVH].

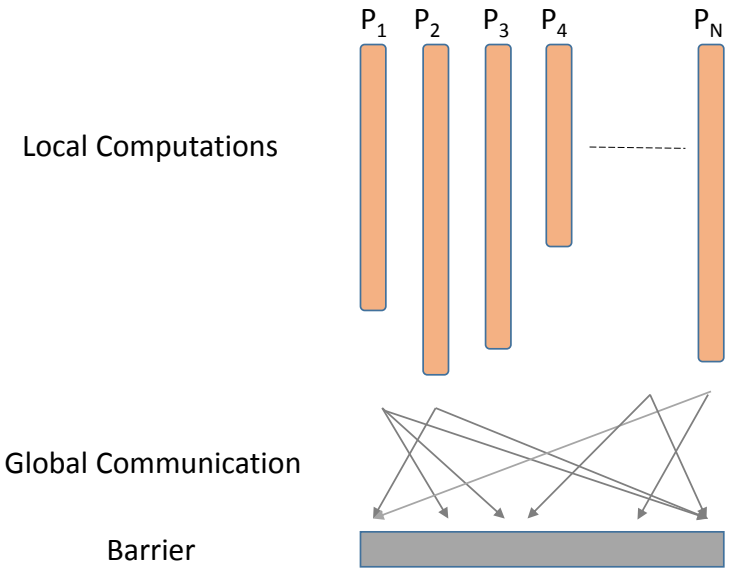

Figure 6.3: Bulk Synchronous Parallel Programming Model.

In this study, the reference implementation of MiniGhost ported into HPX is used. As mentioned in previous sections, with many of the established computational concepts adopted by ParalleX computation model, as well as a flexible and simple AGAS programming model, HPX provides a unique opportunity to move away from conventional programming techniques. The port of MiniGhost into HPX provides the opportunity to study the performance characteristics of the application class using finite difference method following an asynchronous computation as well even-driven point synchronization of (computation) tasks as well as resources.

6.2 Experimental Setup

Systems Used for Experimentation: The experiments in this thesis have been done on Marvin and Trillian nodes of Hermione test cluster at Center for Computation and Technology, LSU. Each Marvin node has 16 cores, with two sockets each along with dedicated memory for a NUMA domain. Each socket has 8 CPU cores with clock frequency of 2.7 GHz (3.5 GHz turbo). Symmetric multi-threading is two way dedicated for the NUMA domains. The micro-architecture class of the processors is Intel Sandy Bridge. Each NUMA domain has 24 GB dual channel DDR3 RAM, with system aggregate memory of 48 GB. The OS used by the SMP system is 64 bit Linux with kernel version of 3.8.13.

Each Trillian node in the Hermione cluster has 64 AMD Opteron processors, with 4 sockets. Each socket has 16 cores, with 8 cores for the two NUMA domains within the socket. Each CPU core has clock frequency of 2.1 GHz (3.0 GHz turbo for 8 or less cores; 2.4 GHz turbo for more than 8 cores). The micro-architecture class of the processors is Bulldozer. Each NUMA Node has 16GB dual channel DDR3 RAM, with system aggregate memory of 128GB. The OS used by each Trillian node is 64 bit Linux with kernel version of 3.8.13.

6.3 UTS Experiments

Reference implementation of UTS benchmark ported into HPX, were executed on single SMP Marvin and Trillian nodes, with following parameter variations:

(a). A variation in assignment of CPU cores, hence OS threads to the HPX application. This setting incrementally made more resource available to the application, of given fixed size.

(b). A variation in the problem size from approximately 110 million nodes and 3 billion nodes.

(c). A variation in the task granularity stolen, of 8 nodes, 20 nodes and 32 nodes.

(d). A variation of scheduling policy for each run configuration, using local-priority, static, local, hierarchy and abp-priority scheduling policies, described in section 3.4.

6.3.1 UTS Experiment Results

In this section, results of various experiments done on UTS benchmark is discussed. The results presented in figures 6.4, 6.5, 6.6, 6.7, 6.8, 6.9 show performance of UTS application in HPX, using five different scheduling policies local-priority, abp-priority, local, hierarchy and static. The graph type variations for this set of experiment are Geometric(1) and Binomial(3), using chunk-size of 20 for work stealing. Each figure is further discussed below.

The figure 6.4 shows the performance of UTS benchmark run on Marvin node, with OS threads varying between 1 and 16. The graph type is Geometric(1) with problem size of approximately 4 million graph nodes(or work nodes). The figure shows the performance of different scheduling policies, where the static policy has the has the least desired performance. Meanwhile, the performance of other scheduling policies, though better than static policy, do not show steading scaling performance, both compared to its previous performance trend as well as compared with the performance of other scheduling policies, for the given problem type and size.

The figure 6.5 shows the performance of UTS benchmark run on Marvin node, with upto 16 OS thread runs for Geometric(1) graph type, with problem size of approximately 100 million graph nodes. Of the different scheduling policies selected the static policy still

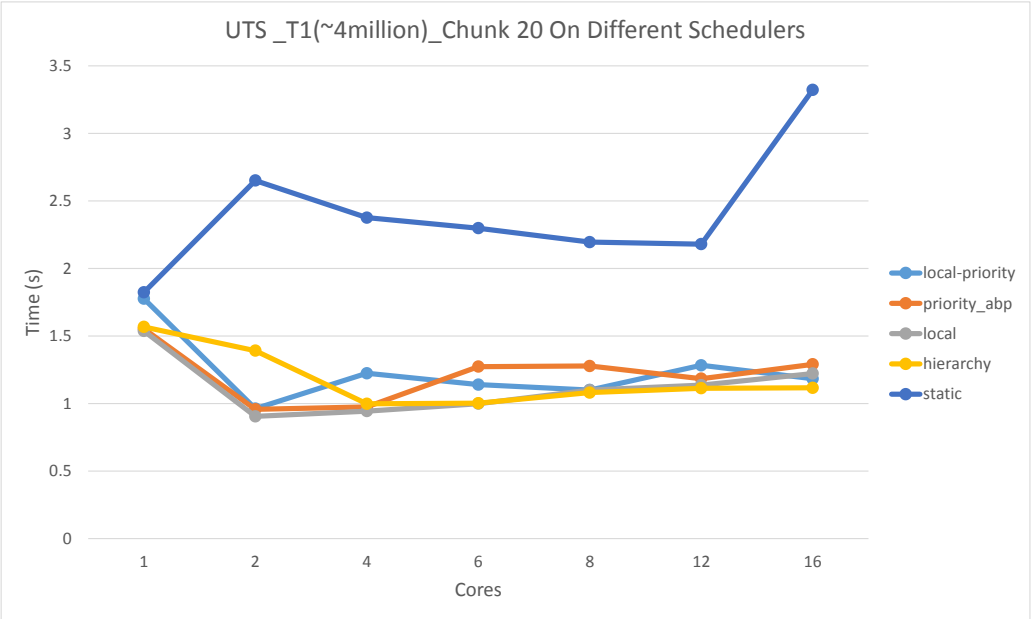

Figure 6.4: Performance Measurement of UTS benchmark in HPX with chunk size 20 and problem size of approx. 4 million nodes for Geometric(1) graph type, using different scheduling policies.

show no scaling behavior; with performance deteriorating further as the level of hardware concurrency increases. Meanwhile, the other scheduling policies show scaling behavior, with similar patterns, and eventually flattening out after 8 and more cores for this problem type and size.

The figure 6.6 shows the performance of UTS benchmark run on Marvin node, with upto 16 OS threads runs for Geometric(1) graph type, with problem size of approximately 4 billion graph noes. Out of the five scheduling policies selected, static policy still does not scale in this setting. Another thread scheduling policy, hierarchy, shows poor performance at lesser OS thread count, and gradually catches up with other scheduling policies. In this setting, all the scheduling policies show slight performance perturbations as we increase more OS threads.

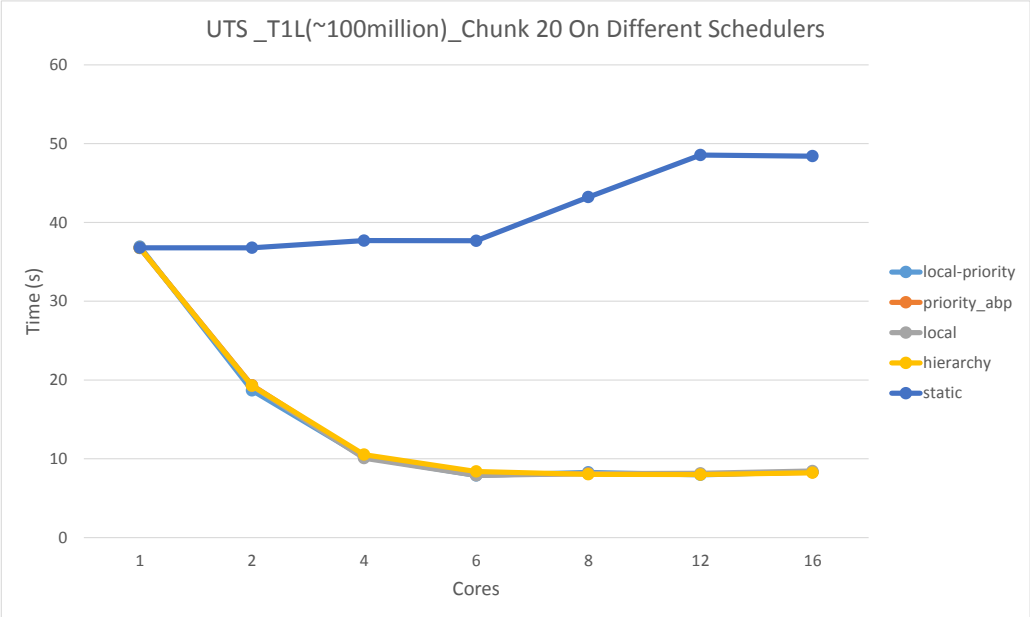

Figure 6.5: Performance Measurement of UTS benchmark in HPX with chunk size 20 and problem size of approx. 100 million nodes Geometric(1) graph type, using different scheduling policies.

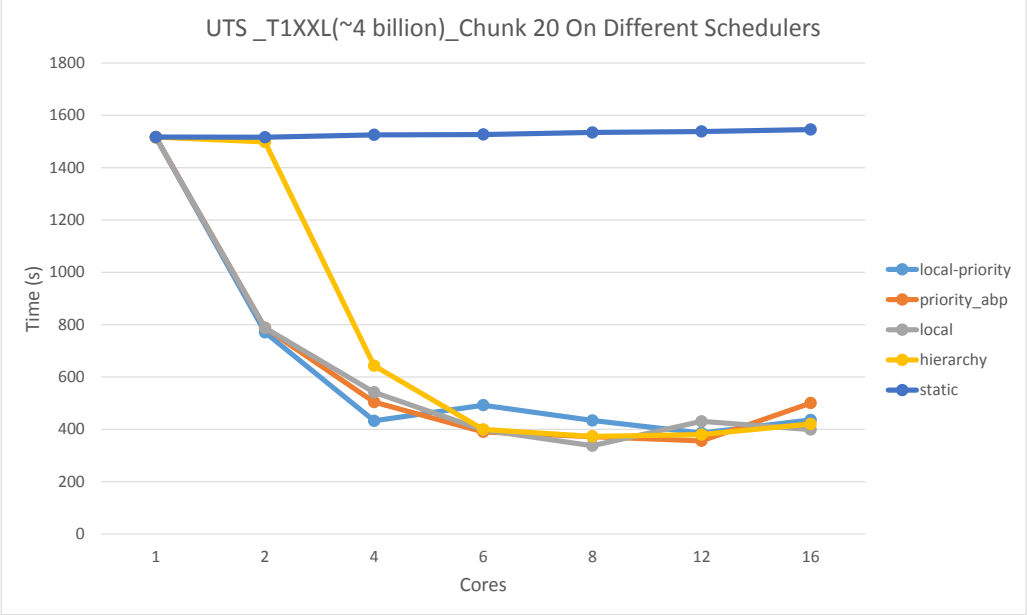

Figure 6.6: Performance Measurement of UTS benchmark in HPX with chunk size 20 and problem size of approximately 4 billion nodes, for Geometric(1) graph type, using different scheduling policies.

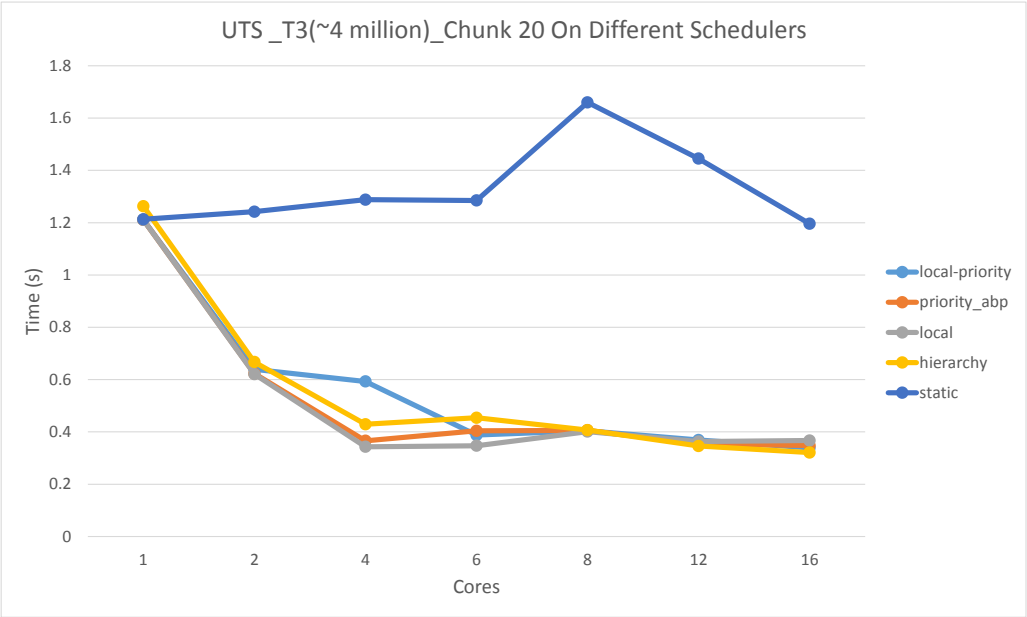

Figure 6.7: Performance Measurement of UTS benchmark in HPX with chunk size 20 and problem size of approx. 4 million nodes for Binomial(3) graph type, using different scheduling policies.

The figures 6.7, 6.8, 6.9 show performance of the UTS benchmark of Binomial(3) type on Marvin node, again with upto 16 thread variations. The problem sizes again vary from approximately 4 million, to 110 million to 3 billion. Again, in all the test configurations, the static scheduler, which does not do any work stealing, even though the UTS application does load balancing on its own, show no performance gain as the number of OS threads are increased. Other schedulers however show better better scaling compared to type 1 graph, with lesser perturbations. For T3L graph type, the abp-priority scheduler shows degraded performance.

The figure 6.10 shows the performance of UTS on Marvin node, with binomial(1) graph type, of 3 billion size, using chunk size of 32. Compared to the previous runs of same type but with chunk size 20, the different schedulers seem to show performance fluctuations. The figure 6.11 shows performance of binomial(1) type on Trillian node, with chunk size of 32. In the latter scenario, the scaling behavior is better, upto 64 OS threads for the given problem size. This tells that the architectural difference between the two hardware

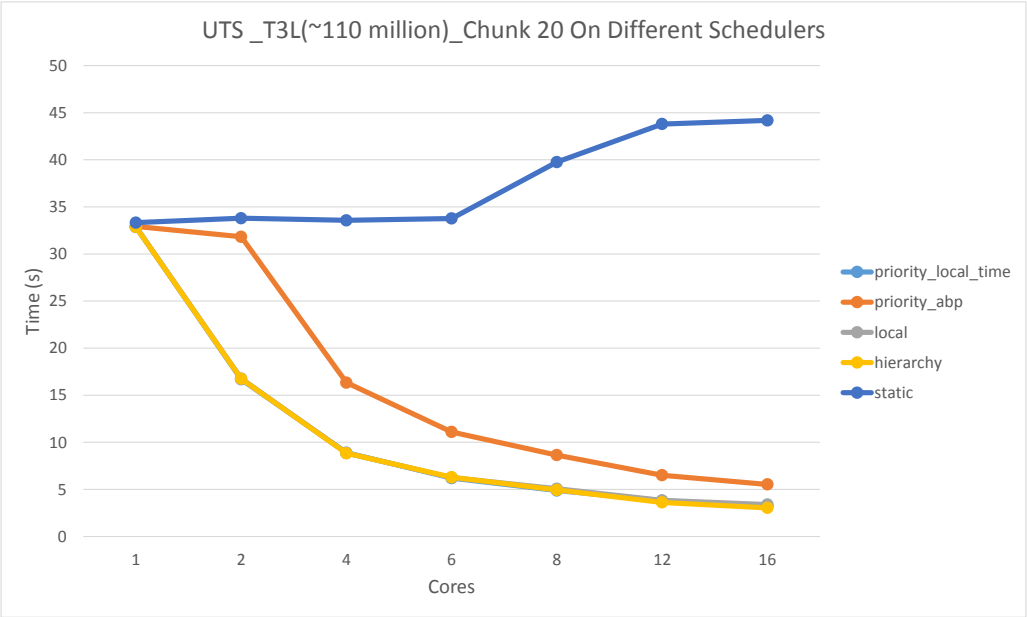

Figure 6.8: Performance Measurement of UTS benchmark in HPX with chunk size 20 and problem size of approx. 110 million nodes for Binomial(3) graph type, using different scheduling policies.

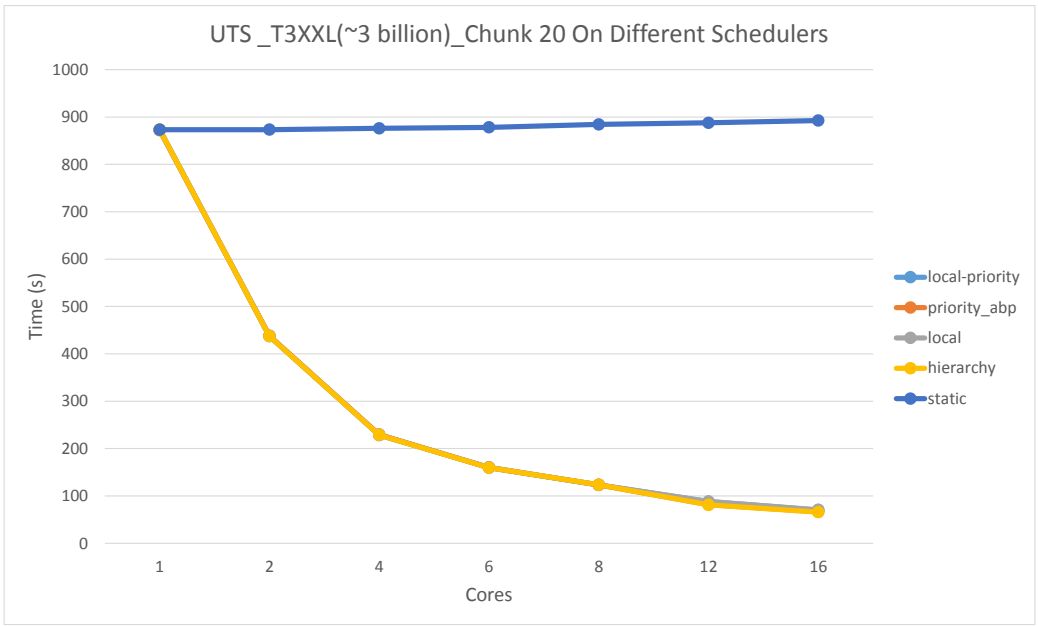

Figure 6.9: Performance Measurement of UTS benchmark in HPX with chunk size 20 and problem size of approx. 3 billion nodes for Binomial(3) graph type, using different scheduling policies.

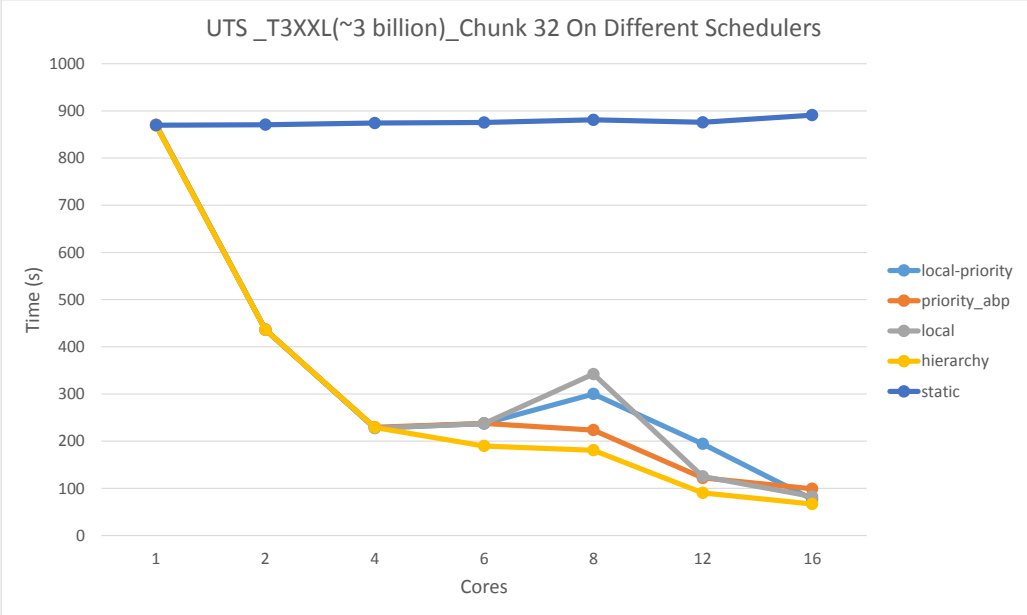

Figure 6.10: Performance Measurement of UTS benchmark in HPX with chunk size 32 and problem size of approx. 3 billion nodes for Binomial(3) graph type, using different scheduling policies.

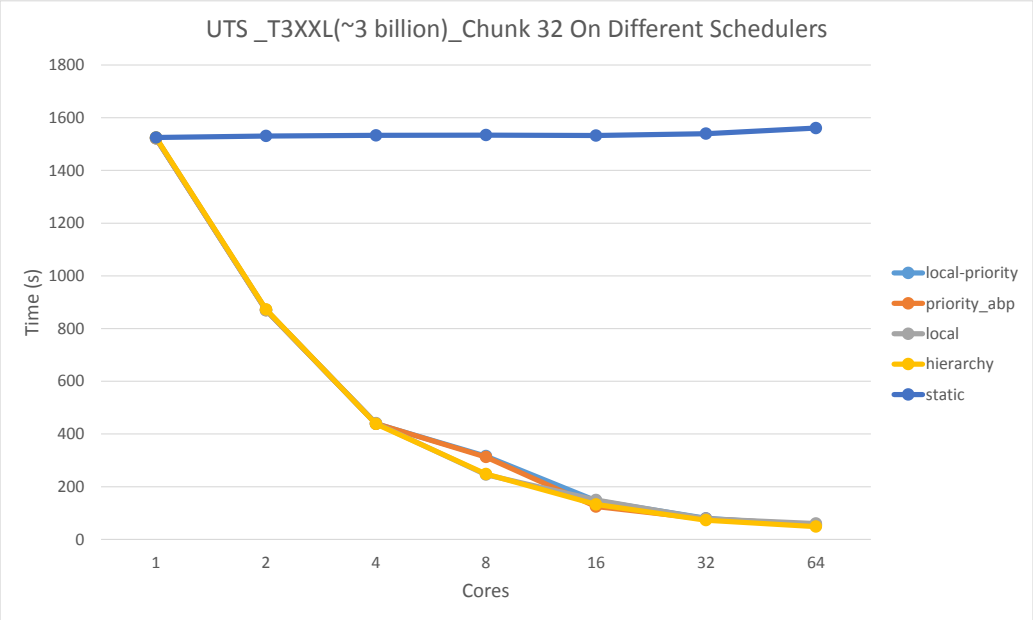

Figure 6.11: Performance Measurement of UTS benchmark in HPX with chunk size 32 and problem size of approx. 3 billion nodes, for Binomial(3) graph type, using different scheduling policies on Trillian node.

platforms, Marvin and Trillian, with different NUMA settings, do not put much impact on the performance of UTS application.

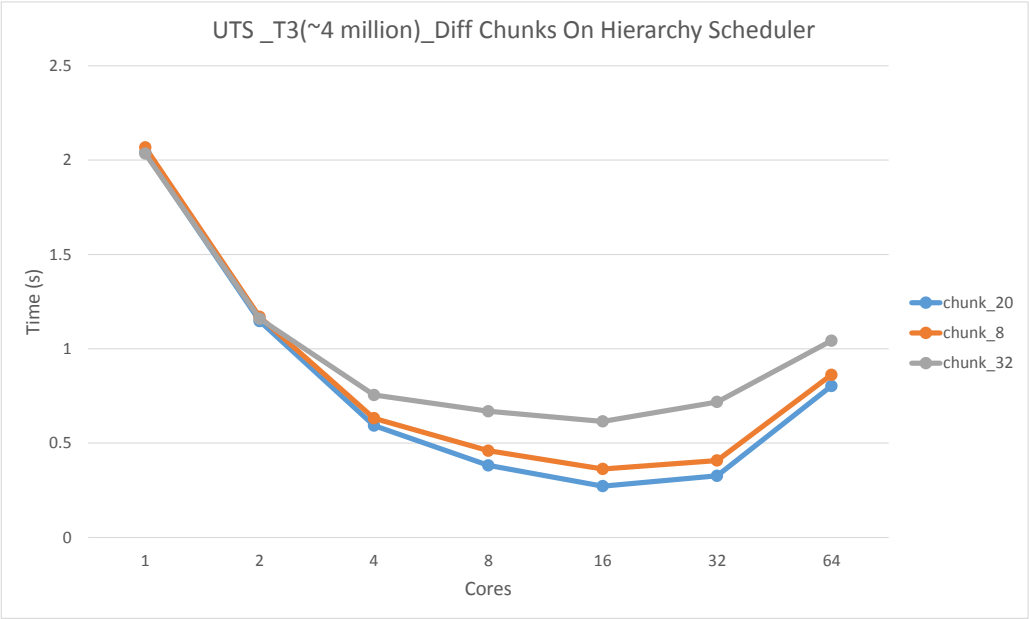

Figure 6.12: Comparison of Performance Measurement of UTS benchmark in HPX with different chunk sizes for Binomial(3) graph type of size approx. 4 million, using hierarchical scheduling policy on Trillian node.

The figures 6.12, 6.13 and 6.14 show comparison of performance of binomial(3) graph type , for problem sizes of approximate 4 million, 110 million and 3 billion with three chunks sizes 8, 20 and 32 for each comparison, using hierarchy scheduling policy. The comparisons show that certain chunk size selection gives higher performance, where the combined effect of overhead due to synchronization for load balancing and the limit of parallelism due to an unbalanced queue are both minimal. Also for all the chunk sizes, for smaller problem sizes, see figure 6.12 the performance degrades after 16 OS threads. This is due to synchronization overheads due to less available work. As the problem size is increased, as we see in figures 6.13 and 6.14 the scaling of the application also improves. In addition to that, we can also observe that higher chunk size gives better performance as the problem size increases, due to amortization of synchronization overheads. Additional experiments on the UTS benchmark application, for different graph types, chunk sizes, architectures and scheduling policy variations is listed in Appendix B.

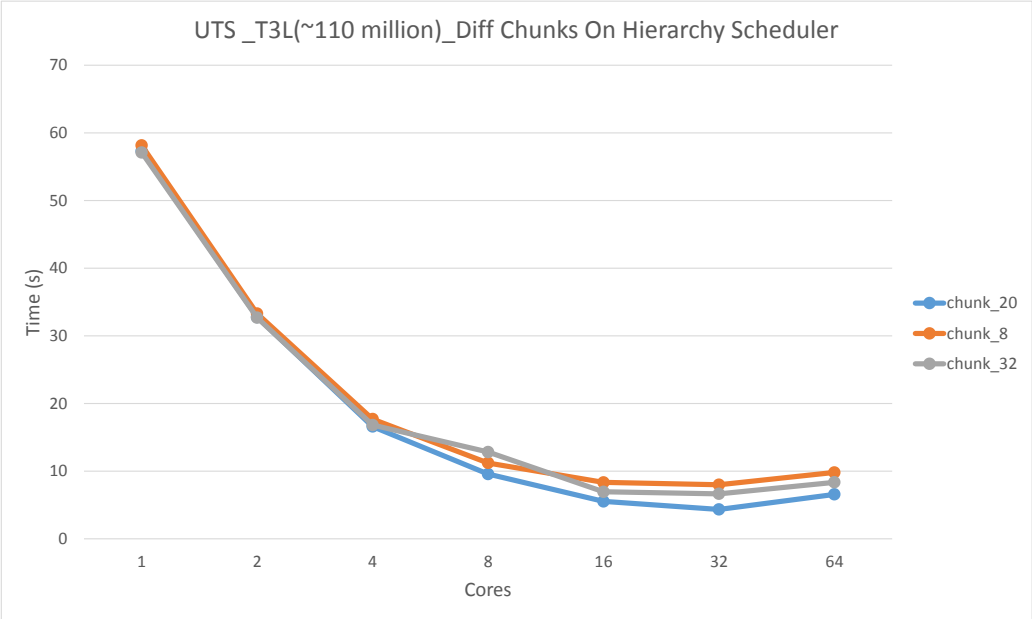

Figure 6.13: Comparison of Performance Measurement of UTS benchmark in HPX with different chunk sizes for Binomial(3) graph type of size approx. 110 million, using hierarchical scheduling policy on Trillian node.

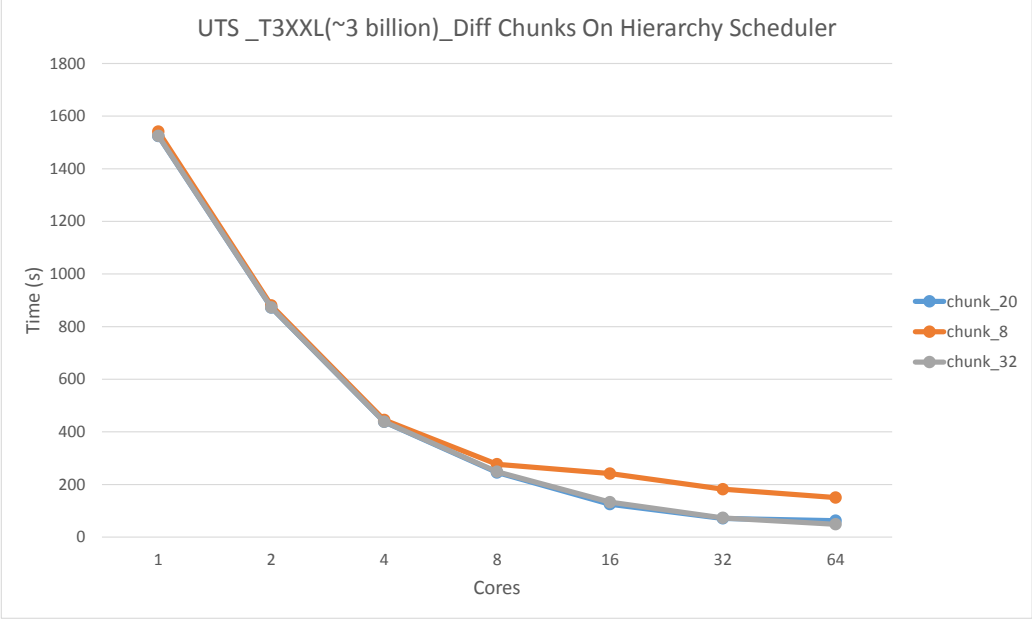

Figure 6.14: Comparison of Performance Measurement of UTS benchmark in HPX with different chunk sizes for Binomial(3) graph type of size approx. 3 billion, using hierarchical scheduling policy on Trillian node.

6.3.2 Summary

From the results discussed above, it can be inferred that:

(a). The application's behavior with different schedulers differ with different problem sizes.

(b). For dynamic unbalanced graph types, active load balancing by the runtime system makes significant impact on performance

(c). Even for the same type of application, under different application parameters, the scheduler may perform differently and

(d). Different problems within same application class may need extra attention in task management for better performance characteristics.

6.4 MiniGhost Experiments

Reference implementation of Minighost in HPX, was experimented on single SMPs, Marvin and Trillian nodes, with parameter variations of:

(a). An incremental assignment of CPU cores, hence OS threads to the HPX application. This setting incrementally made more resource available to the application, of given size. Again, the OS thread-core allocation were chosen with NUMA effect into perspective.

(b). A variation of scheduling policy for each run configuration, using local-priority, static, local, hierarchy and abp-priority scheduling policies.

(c). Problem size variation of 3D grid points of 100x100x100, 200x200x200 and 400x400x400 dimensions.

(d). Stencil evaluation methods: 2D5PT(21), 2D9PT(23) and 3D27PT(24).

For the experiment, the additional parameter applied to the applications were: number of variables, 40; number of steps was set to 20 with reduction value of 10% of the variables in each time step.

6.4.1 MiniGhost Experiment Results

In this section, the results of various experiments done on Minighost benchmark is is discussed. The results presented in figures 6.15, 6.16, 6.17 and 6.18 show the performance of Minighost application in HPX, using five different scheduling policies. The scheduling policies are same ones used in UTS benchmark experiment, which are local-priority, abp-priority, local, hierarchy and static. The following figures discuss the findings in more detail.

The figure 6.15 shows the performance of Minighost application of problem size 100x100x100, using stencil-23. The performance graph shows that while performance of some scheduling policies are better than the others, most of them follow general trend of better scaling besides the hierarchy scheduling policy. The most important distinction from the UTS results and this result is that the static scheduling policy is also showing better scaling performance, in similar trend with other scheduling policies. This distinguishes the difference in requirements of scheduling policies for the two applications.

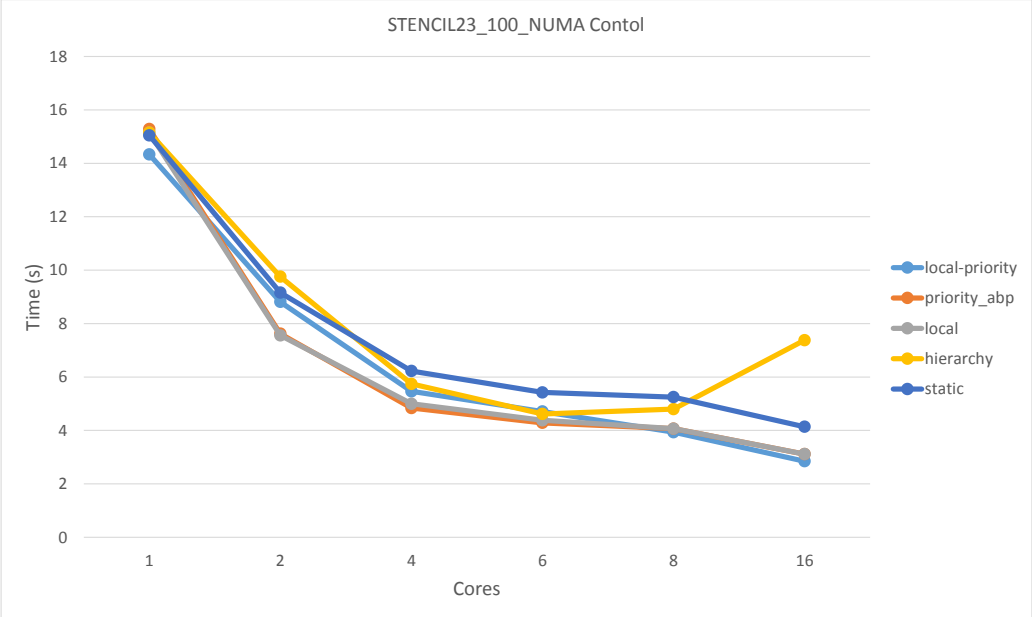

Figure 6.15: Performance Measurement of MiniGhost benchmark application, using stencil 23 and gridsize 100 in HPX for different scheduling policies.

The figure 6.16 shows the performance of Minighost application of problem size 200x200x200, using stencil-23. The performance graph shows that the performance of different scheduling policies are almost same except for the hierarchy scheduler. For hierarchy scheduler, the performance is poorer for smaller OS thread assigned, and gradually improves as the number of OS thread count increased.

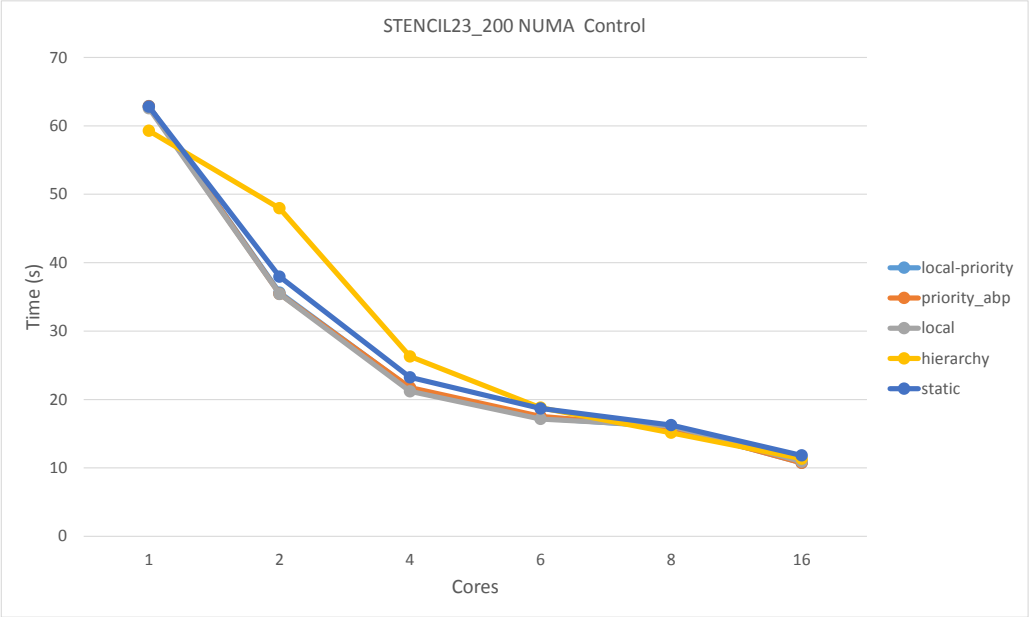

Figure 6.16: Performance Measurement of MiniGhost benchmark application, using stencil 23 and gridsize 200 in HPX for different scheduling policies.

The figure 6.17 shows the performance of Minighost application of problem size 400x400x400, using stencil-23. The performance graph shows that the performance of different scheduling policies are similar to what we witnessed in the previous setting shown in figure 6.16 except for the time taken by the application at each OS thread configuration, which is higher than the previous run, as the problem size is also bigger.

The figure 6.18 shows the performance comparison of Minighost application of problem size 100x100x100, using stencil-21 on Marvin and Trillian nodes. The performance graph shows that, besides the role played by the speed of CPU for each architecture, there is only slight difference in performance due to the scheduling policies between the two hardware architectures used in this experimentation. Further experimentation results with different

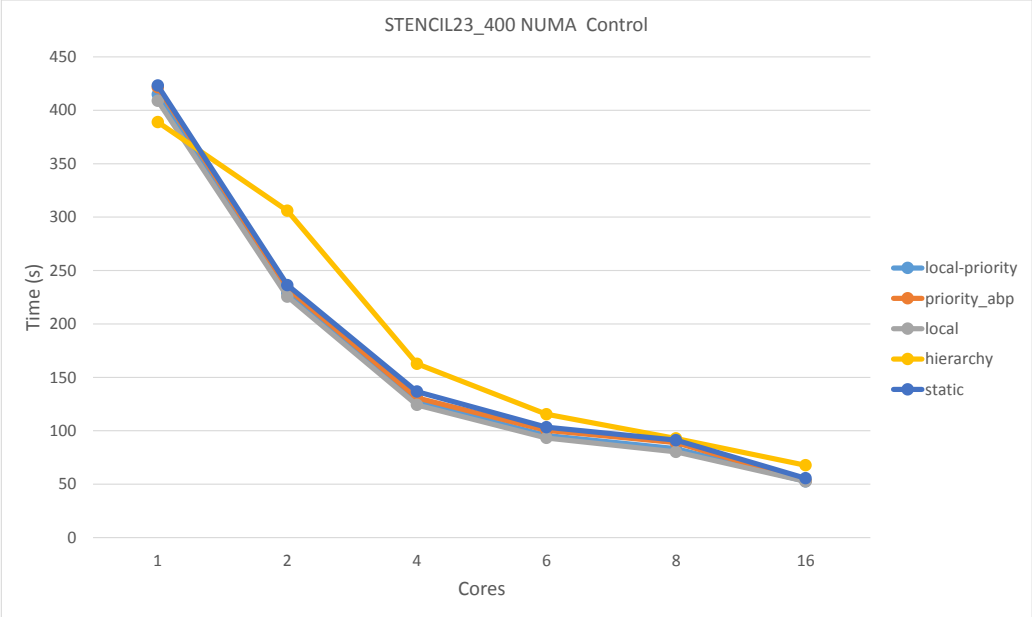

Figure 6.17: Performance Measurement of MiniGhost benchmark application, using stencil 23 and gridsize 400 in HPX for different scheduling policies.

stencil methods, problem sizes and on different architecture is presented in Appendix B.

6.4.2 Summary

From the results discussed above, it can be inferred that:

(a). For stencil based applications, that follow bulk-synchronous protocol, active load balancing by the runtime system does not necessarily translate in big performance gain.

(b). Performance under certain scheduling policies are not always same with respect to other scheduling policies; under certain situation one is better than other, and vice-versa.

6.5 Final Summary

Performance of scheduling policies primarily depend upon the applications' characteristic. It also depends upon the parameters that control or define the granularity of the task size for that application. Within the same application class, a particular execution policy may not always translate into better performance, and a closer task coordination

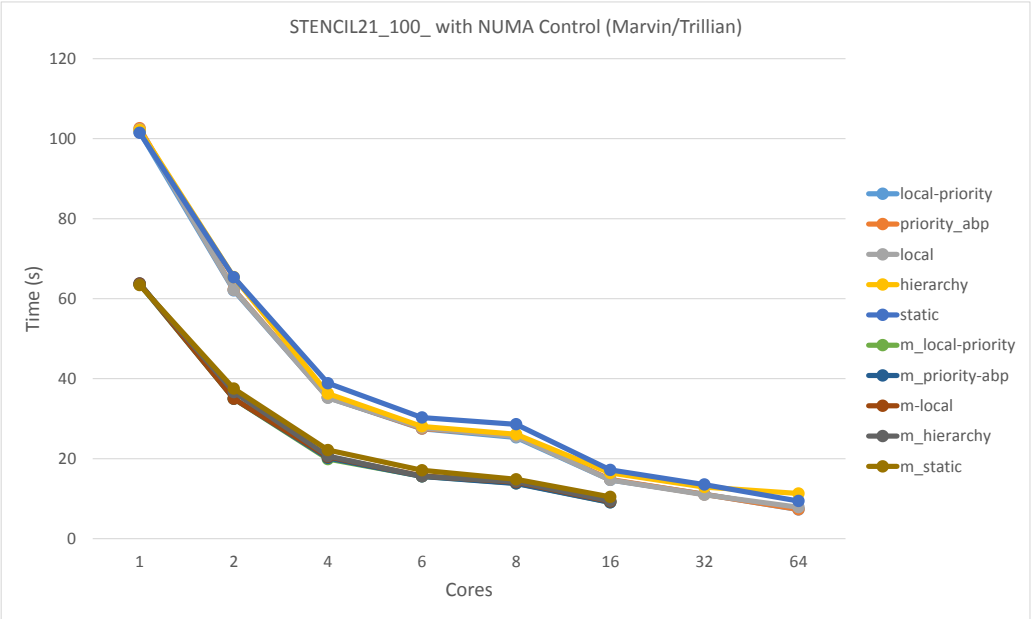

Figure 6.18: Performance Measurement of MiniGhost benchmark application, using stencil 21 and gridsize 100 in HPX for different scheduling policies and two different system architectures.

may be needed to show any advantageous effect of an execution policy. Also some execution policy may not be suitable for optimal performance. Now, across the application domains, a scheduling policy showing poor performance for the application might show better performance results with another application. Thus, from our observations from the experiments on the two different types of applications, we can infer that for applications with dynamic characteristics and non-uniform work load, a framework for adaptive task control is needed, where the task granularity as well as scheduling policies can be chosen at runtime.

7 Related Work

Chapel [CCZ04] is a parallel programming language that supports multi-threaded execution model, enables locality awareness for data distribution in a Partitioned Global Address Space (PGAS) [Yel+07] context. Chapel presents the concept of Domains [Dia+] as a named indexed set object that may represent data structures of different types and which can distributed across multiple localities (nodes in conventional sense). Chapel also supports concurrent task deployment via threads and synchronization of concurrent tasks is achieved via synchronizing variables. While individual concepts driving the Chapel programming model are powerful, Chapel does not provide a comprehensive approach of unifying these concepts with computational entities as well as the available resources, and represent it as a single entity.

Co-Array Fortran (CAF) [Coa+03] is an extension to Fortran Programming Language (Fortran 90), integrating communication and synchronization primitives as part of the language, following a Global Address Space programming model. New extension allowed direct addressing of parts of an array/data structure that is distributed across localities, via just indexing. Further extension of the language as CAF 2.0 allowing dynamic allocation of arrays, better synchronization primitives, with split-phase barriers, asynchronous communication, locality sensitive data allocation, etc. [MC+09]. While these features do take care of requirements for application scalability, they do not take into account the requirements for applications where adaptive work management needs to be taken into account. Selective mapping of the co-arrays to designated localities is not supported, which limits the flexibility in task and resource management. CAF is also a PGAS language.

X10 is another object oriented programming language introducing notion of places, regions and distributions to support parallelism of computation across a multiple heterogeneous system nodes with non-uniform data access using partitioned global address space as address resolution mechanism [ESS04] [ESS05]. X10 puts forth the notion of light weight threads that allows execution of finegrained tasks, with places representing the collection of such threads and relevant data. X10 has several synchronization semantics, in the form of clocks, unconditional atomic sections, conditional atomic sections, etc. X10 further supports heterogeneous and asynchronous computation through asynchronous partitioned global address space [Sar+10]. For communication X10 uses a form of active messages and a heuristic flow control mechanism to avoid deadlocking of resources [Aga+07]. These excellent features makes X10 on of the leading PGAS languages that takes resources heterogeneity as well as asynchrony in computations into account. However, X10 still lacks the concept of composing these ideas into a unison, for effective resource management with dynamic adaptive policy deployment features.

Titanium is yet another object oriented programming language that follows PGAS model, with explicit parallel SPMD control model [DBY05; KSY05]. It provides a distinction between local and global memory, for exploiting locality properties of application programs. Titanium details a process as a thread of control, with collection of processes terms as a *team*. Sharing of information between processes is done via shared variables, data-structures and function calls that allows access to these variables/data-structures [PNHY].

UPC is a message driven, distributed programming language that follows SPMD model, in PGAS context [HIY03]. UPC allows data affinity with the threads where they are created, with better locality behavior. In UPC, load distribution is static, with lacking features for dynamic load balancing. Again, without much granularity size variation of threads, fine grained adaptive tuning of thread sizes is not possible, limiting scalability of certain class of applications. Further, UPC does not consider namespace and access policy management schemes for scalable and secure operation of multi-domain applications.

Charm++ [KK93]is an object oriented, message driven parallel programming language written in C++. Charm++ is influenced by actors model [Agh86], where communication for concurrent objects happen only through messages. Charm++ features distributed objects, called chares, for distributed computation management. Charm++ also employs an extensive load balancing schemes based on performance measurements, with migration of objects as one of the mechanisms. While Charm++ has many of the advanced features that is required for load balancing in a hybrid environment, it does not provide the semantics for task management along with customizable fine grained policy deployment. For complex applications to be able to show better performance on complex system architectures, an adaptive dynamic computation management framework with the ability to deploy different access and execution policies is becoming more necessary.

8 Conclusions and Future Direction

This dissertations contributes in the conceptual outlining of Parallel Processes as a platform for deployment of a of scalable resource management framework. Parallel Processes can provide a universal and scalable platform for orchestration of computation, even with intricate dependencies, for a wide variety of applications, with a promise of scalability. This is partly also due to the flexibility of structuring of relevant sub-processes as well as means of decoupling non-relevant sub-processes of the overall Parallel Processes structure. Such targeted execution facility also allows deployment of selective execution, management and access policies for interacting objects within the Parallel Process hierarchy. Also, for scalable Parallel Process deployment, a scalable naming service is essential.

The contributions of this thesis is briefly discussed below:

- A holistic approach to address the scalability challenges in Exascale and beyond computing.
- Defined a conceptual framework for designing infrastructure for resource and computation management in an era of extreme scale computations.
- Provided a comprehensive list of guidelines and use cases for Parallel Processes for control, coordination and access policy of applications in emerging classes of interdependent, multi-domain application frameworks as well as scalability impaired applications.
- Discussed and identified the design requirements for AGAS service to be a scalable namespace management service in a global address space context. This in turn ensures scalability of Parallel Processes, as AGAS is a required service for it.

- Demonstrated the need of adaptive dynamic runtime resource management for applications within the same class as well as across different application classes.
- Showed the significance of dynamic load balancing, not only from an application's perspective but also from the runtime system's perspective.
- Implemented UTS benchmark in HPX programming model, based on reference implementation.

Further exploring means of storing meta data about all the relevant distributed sub-processes, either using distributed container, or having a distributed indexing scheme for distributed processes or sub-processes would be next incremental step for realizing an scalable resource management framework. In addition to the scalable referencing scheme or distributed data storage mechanism, close integration of performance measurement framework would enable placement of a feedback system for dynamic adaptive resource management.

Again, as discussed in the earlier sections, scalability of the resource management framework using Parallel Processes is directly dependent upon the scalability of AGAS system, with an efficient mechanism for facilitation of namespace management for migration of objects for distributed load balancing. So an implementation and experimentation of this topic could be a possible avenue of exploration.

In addition to this, Parallel Processes could be a platform for access control mechanism for interacting applications in a multi-modal, multi-physics scenario; as well as for defining access rules, on data objects, for third party users . Placement of such mechanism would limit unintended behavior in computation. Implementation and experiments on efficiency of an access policy framework for multi-application, multi-modal framework systems could be a next achievable target.

Different applications with different computation paradigms show different performance characteristics on different networking of system hardware. A universal network design for systems hardware has been an illusive target. With Parallel Processes, a most generic

network architecture could be used to organize network flow at the software level,when needed, according to the requirements of the applications in context, hence promising better performance.

Bibliography

[EE14] Giovanni Erbacci and Andrew Emerson. "Trends in HPC Architectures and Parallel Programmming". In: Supercomputing, Applications And Innovation Department - CINECA, 2014.

[McK04] Sally A. McKee. "Reflections on the Memory Wall". In: *Proceedings of the 1st Conference on Computing Frontiers*. CF '04. Ischia, Italy: ACM, 2004, pp. 162–. ISBN: 1-58113-741-9. DOI: 10.1145/977091.977115. URL: http://doi.acm.org/10.1145/977091.977115.

[Mar+97] Richard P. Martin et al. "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture". In: *SIGARCH Comput. Archit. News* 25.2 (May 1997), pp. 85–97. ISSN: 0163-5964. DOI: 10.1145/384286.264146. URL: http://doi.acm.org/10.1145/384286.264146.

[Ste11] Thomas Sterling. "HPC in Phase Change: Towards a New Execution Model". In: *High Performance Computing for Computational Science–VECPAR 2010* (2011), pp. 31–31.

[Bor10] Shekhar Borkar. "The Exascale challenge". In: *VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on*. Hsin Chu, Taiwan: IEEE, 2010, pp. 2 –3. DOI: 10.1109/VDAT.2010.5496640.

[KBS09] Hartmut Kaiser, Maciej Brodowicz, and Thomas Sterling. "ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications". In: *Parallel Processing Workshops*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 394–401. DOI: http://doi.ieeecomputersociety.org/10.1109/ICPPW.2009.14.

[Gao+07] G. Gao et al. "Parallex: A study of a new parallel computation model". In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. 2007, pp. 1–6. DOI: 10.1109/IPDPS.2007.370484.

[Lav78] S. H. Lavington. "The Manchester Mark I and Atlas: A Historical Perspective". In: *Commun. ACM* 21.1 (Jan. 1978), pp. 4–12. ISSN: 0001-0782. DOI: 10.1145/359327.359331. URL: http://doi.acm.org/10.1145/359327.359331.

[Dij02] Edsger W. Dijkstra. "The Origin of Concurrent Programming". In: ed. by Per Brinch Hansen. New York, NY, USA: Springer-Verlag New York, Inc., 2002.

Chap. Cooperating Sequential Processes, pp. 65–138. ISBN: 0-387-95401-5. URL: http://dl.acm.org/citation.cfm?id=762971.762974.

[Dij68] Edsger W. Dijkstra. "The Structure of the &Ldquo;THE&Rdquo;-multiprogramming System". In: *Commun. ACM* 11.5 (May 1968), pp. 341–346. ISSN: 0001-0782. DOI: 10.1145/363095.363143. URL: http://doi.acm.org/10.1145/363095.363143.

[Hoa02] C. A. R. Hoare. "The Origin of Concurrent Programming". In: ed. by Per Brinch Hansen. New York, NY, USA: Springer-Verlag New York, Inc., 2002. Chap. Towards a Theory of Parallel Programming, pp. 231–244. ISBN: 0-387-95401-5. URL: http://dl.acm.org/citation.cfm?id=762971.762978.

[RH02] Raul Rojas and Ulf Hashagen, eds. *The First Computers: History and Architectures*. Cambridge, MA, USA: MIT Press, 2002. ISBN: 0262681374.

[UC91] Office of Technology Assesment U.S. Congress. *Seeking Solutions: High Performance Computing for Science-Background Paper*. OTA-BP-TCT-77. Washington,DC: U.S. Government Printing Office, 1991.

[Key12] David Keyes. "High Performance Computing in Science and Engineering: the Tree and the Fruit". In: NITRD Symposium, 2012.

[Pot85] Jerry L. Potter. *The Massively Parallel Processor*. Cambridge, MA, USA: MIT Press, 1985. ISBN: 0262161001.

[BB99] Mark Baker and Rajkumar Buyya. "Cluster computing: the commodity supercomputer". In: *Software-Practice and Experience* 29 (1999), pp. 551–576.

[Top] *www.top500.org*. 2014. URL: {http://www.top500.org/lists/2014/06/}.

[SA+09] William Carlson Saman Amarasinghe Dan Campbell et al. "ExaScale Software Study: Software Challenges in Extreme Scale Systems". In: (2009).

[SZS+07] Horst Simon, Thomas Zacharia, Rick Stevens, et al. "Modeling and simulation at the exascale for energy and the environment". In: (2007).

[Jos] Earl Joseph. "HPC Trends, Big Data And The Emerging Market For High Performance Data Analysis". In: Salishan: IDC.

[Kai] Hartmut Kaiser. *ParalleX: A Cure for Scaling Impaired Parallel Applications*. URL: http://stellar.cct.lsu.edu/pubs/Cetraro.pdf.

[Ama+11b] Saman Amarasinghe et al. *Programming Challenges for Exascale Computing*. Tech. rep. ACSR Workshop On Exsascale Programming Challenges. Department of Energy, 2011.

[JSLO] Michael Wehner John Shalf and John Hules Leonid Oliker. "The Challenge of Energy-Efficient HPC". In: www.scidacreview.org: SCIDAC Review.

[Str+09] B. van Straalen et al. “Scalability challenges for Massively Parallel AMR Applications”. In: *Proceedings of the 23th IEEE International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12. ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5161014. URL: http://portal.acm.org/citation.cfm?id=1586640.1587714.

[Lum+06] Andrew Lumsdaine et al. *CHALLENGES IN PARALLEL GRAPH PROCESSING*. 2006.

[KW10] Humaira Kamal and Alan Wagner. “FG-MPI: Fine-grain MPI for multicore and clusters”. In: *Parallel and Distributed Processing Workshops and PhD Forum, 2011 IEEE International Symposium on* 0 (2010), pp. 1–8. DOI: http://doi.ieeecomputersociety.org/10.1109/IPDPSW.2010.5470773.

[HM08] Mark D. Hill and Michael R. Marty. “Amdahls law in the multicore era”. In: *IEEE COMPUTER* (2008).

[SHS09] Vivek Sarkar, William Harrod, and Allan E Snavely. “Software challenges in extreme scale systems”. In: *Journal of Physics: Conference Series*. 2009, p. 012045.

[Löf+13] Frank Löffler et al. “Cactus: Issues for Sustainable Simulation Software”. In: *CoRR* abs/1309.1812 (2013). URL: http://arxiv.org/abs/1309.1812.

[Phi+05] James C Phillips et al. “Scalable molecular dynamics with NAMD”. In: *Journal of computational chemistry* 26.16 (2005), pp. 1781–1802.

[All+08] Gabrielle Allen et al. *HPC Application Software Consortium Summit Concept Paper*. Tech. rep. HPC-ACS Planning Workshop, 2008.

[Cve] *std::vector*.

[Sub+93] Jaspal Subhlok et al. “Exploiting Task and Data Parallelism on a Multicomputer”. In: *In ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1993, pp. 13–22.

[Xin+13] Reynold S. Xin et al. “GraphX: A Resilient Distributed Graph System on Spark”. In: *First International Workshop on Graph Data Management Experiences and Systems*. GRADES ’13. New York, New York: ACM, 2013, 2:1–2:6. ISBN: 978-1-4503-2188-4. DOI: 10.1145/2484425.2484427. URL: http://doi.acm.org/10.1145/2484425.2484427.

[Tab+11] A. Tabbal et al. “Preliminary Design Examination of the ParalleX System from a Software and Hardware Perspective”. In: *SIGMETRICS Performance Evaluation Review* 38 (2011), p. 4.

[Eic+92] T.v. Eicken et al. “Active Messages: A Mechanism for Integrated Communication and Computation”. In: *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on* (1992), pp. 256–266.

[Kai+14] Hartmut Kaiser et al. “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*. Eugene, Oregon, USA: ACM, 2014.

[Amd67] Gene M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560.

[Axe86] Tim S. Axelrod. “Effects of synchronization barriers on multiprocessor performance”. In: *Parallel Computing* 3.2 (1986), pp. 129 –140. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/0167-8191(86)90030-X. URL: http://www.sciencedirect.com/science/article/pii/016781918690030X.

[Mes09] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. Stuttgart, Germany: High Performance Computing Center Stuttgart (HLRS), 2009.

[Yel+07] Katherine Yelick et al. “Productivity and Performance Using Partitioned Global Address Space Languages”. In: *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*. PASCO ’07. London, Ontario, Canada: ACM, 2007, pp. 24–32. ISBN: 978-1-59593-741-4. DOI: 10.1145/1278177.1278183. URL: http://doi.acm.org/10.1145/1278177.1278183.

[Cha] *The Charm++ Parallel Programming System Manual. Vesion 6.5.0*. Parallel Programming Laboratory, University of Illinois at Urbana-Champaign.

[Aga+07] Shivali Agarwal et al. “Deadlock-free Scheduling of X10 Computations with Bounded Resources”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’07. San Diego, California, USA: ACM, 2007, pp. 229–240. ISBN: 978-1-59593-667-7. DOI: 10.1145/1248377.1248416. URL: http://doi.acm.org/10.1145/1248377.1248416.

[CVG11] Long Chen, O. Villa, and G.R. Gao. “Exploring Fine-Grained Task-Based Execution on Multi-GPU Systems”. In: *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. 2011, pp. 386–394. DOI: 10.1109/CLUSTER.2011.50.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. “Advances in Dataflow Programming Languages”. In: *ACM Comput. Surv.* 36.1 (Mar.

2004), pp. 1–34. ISSN: 0360-0300. DOI: 10.1145/1013208.1013209. URL: http://doi.acm.org/10.1145/1013208.1013209.

[Sga10] Kyriakos N. Sgarbas. "Automata Theory with Modern Applications by James A. Anderson, Cambridge University Press, 2006, Viii+256 Pages". In: *SIGACT News* 41.2 (June 2010), pp. 43–46. ISSN: 0163-5700. DOI: 10.1145/1814370.1814384. URL: http://doi.acm.org/10.1145/1814370.1814384.

[Jac+03] A. Jacquet et al. "An Executable Analytical Perfromance Evaluation Approach for Early Performance Prediction". In: *In Proc. of the Workshop on Massively Parallel Processing held in conjunction with Intl. Parallel and Distributed Processing Symposium (IPDPS-03)*. Nice, France, 2003.

[SW10] Hartmut F.-W. Sadrozinski and Jinyuan Wu. *Applications of Field-Programmable Gate Arrays in Scientific Research*. 1st. Bristol, PA, USA: Taylor & Francis, Inc., 2010. ISBN: 1439841330, 9781439841334.

[FM04] James Fung and Steve Mann. "Using Multiple Graphics Cards As a General Purpose Parallel Computer: Applications to Computer Vision". In: *Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 1 - Volume 01*. ICPR '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 805–808. ISBN: 0-7695-2128-2. DOI: 10.1109/ICPR.2004.968. URL: http://dx.doi.org/10.1109/ICPR.2004.968.

[Gib66] Charles T. Gibson. "Time-sharing in the IBM System/360: Model 67". In: *Proceedings of the April 26-28, 1966, Spring Joint Computer Conference*. AFIPS '66 (Spring). Boston, Massachusetts: ACM, 1966, pp. 61–78. DOI: 10.1145/1464182.1464190. URL: http://doi.acm.org/10.1145/1464182.1464190.

[The08] The C++ Standards Committee. *Working Draft, Standard for Programming Language C++*. 2008. URL: {http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2798.pdf}.

[Var10] Various. *Boost C++ Libraries*. 2010. URL: http://www.boost.org.

[Lei09] Charles E. Leiserson. "The Cilk++ concurrency platform". In: *DAC '09: Proceedings of the 46th Annual Design Automation Conference*. San Francisco, California: ACM, 2009, pp. 522–527. ISBN: 978-1-60558-497-3. DOI: 10.1145/1629911.1630048. URL: http://dx.doi.org/10.1145/1629911.1630048.

[Int10] Intel. *Intel Thread Building Blocks 3.0*. http://www.threadingbuildingblocks.org. 2010. URL: http://www.threadingbuildingblocks.org/.

[Mic10] Microsoft. *Microsoft Parallel Pattern Library*. http://msdn.microsoft.com/en-us/library/dd492418.aspx. 2010. URL: http://msdn.microsoft.com/en-us/library/dd492418.aspx.

[AJ89] A. W. Appel and T. Jim. “Continuation-passing, closure-passing style”. In: *POPL ’89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Austin, Texas, United States: ACM, 1989, pp. 293–302. ISBN: 0-89791-294-2.

[Wal82] David W. Wall. “Messages as active agents”. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’82. Albuquerque, New Mexico: ACM, 1982, pp. 34–39. ISBN: 0-89791-065-6. DOI: http://doi.acm.org/10.1145/582153.582157. URL: http://doi.acm.org/10.1145/582153.582157.

[Cul+93] David E. Culler et al. “A compiler controlled Threaded Abstract Machine”. In: *Journal of Parallel and Distributed Computing* 18 (1993), pp. 347–370.

[Kri+93] A. Krishnamurthy et al. “Parallel programming in Split-C”. In: *Supercomputing ’93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. Portland, Oregon, United States: ACM, 1993, pp. 262–273. ISBN: 0-8186-4340-4. DOI: 10.1145/169627.169724. URL: http://dx.doi.org/10.1145/169627.169724.

[FW76] Daniel P. Friedman and David S. Wise. “CONS Should Not Evaluate its Arguments”. In: *ICALP*. 1976, pp. 257–284.

[Hal85] Robert H. Halstead Jr. “MULTILISP: a language for concurrent symbolic computation”. In: *ACM Trans. Program. Lang. Syst.* 7 (4 1985), pp. 501–538. ISSN: 0164-0925. DOI: http://doi.acm.org/10.1145/4472.4478. URL: http://doi.acm.org/10.1145/4472.4478.

[BH77] Henry C. Baker Jr. and Carl Hewitt. “The incremental garbage collection of processes”. In: *SIGART Bull.* New York, NY, USA: ACM, 1977, pp. 55–59. DOI: http://doi.acm.org/10.1145/872736.806932. URL: http://doi.acm.org/10.1145/872736.806932.

[Han73] Per Brinch Hansen. *Operating System Principles*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1973. ISBN: 0-13-637843-9.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. “A Theory of Communicating Sequential Processes”. In: *J. ACM* 31.3 (1984), pp. 560–599. ISSN: 0004-5411.

[SAW94] B. Schilit, N. Adams, and R. Want. “Context-Aware Computing Applications”. In: *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*. WMCSA ’94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 85–90. ISBN: 978-0-7695-3451-0. DOI: 10.1109/WMCSA.1994.16. URL: http://dx.doi.org/10.1109/WMCSA.1994.16.

[PB10] Artur Podobas and Mats Brorsson. “A Comparison of some recent Task-based Parallel Programming Models”. In: *Proceedings of the 3rd Workshop*

on Programmability Issues for Multi-Core Computers, (MULTIPROG'2010), Jan 2010, Pisa. Qc 20120214. 2010.

[Wil09] Anthony Williams. *C++ concurrency in action: practical multithreading*. Manning, 2009.

[Bla+10] Sergey Blagodurov et al. "A Case for NUMA-aware Contention Management on Multicore Systems". In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. Vienna, Austria: ACM, 2010, pp. 557–558. ISBN: 978-1-4503-0178-7. DOI: 10.1145/1854273.1854350. URL: http://doi.acm.org/10.1145/1854273.1854350.

[Le+07] H. Q. Le et al. "IBM POWER6 Microarchitecture". In: *IBM J. Res. Dev.* 51.6 (Nov. 2007), pp. 639–662. ISSN: 0018-8646. DOI: 10.1147/rd.516.0639. URL: http://dx.doi.org/10.1147/rd.516.0639.

[Ber+09] Emery D. Berger et al. "Grace: Safe Multithreaded Programming for C/C++". In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. Orlando, Florida, USA: ACM, 2009, pp. 81–96. ISBN: 978-1-60558-766-0. DOI: 10.1145/1640089.1640096. URL: http://doi.acm.org/10.1145/1640089.1640096.

[Han02] Per Brinch Hansen. "The Origin of Concurrent Programming". In: ed. by Per Brinch Hansen. New York, NY, USA: Springer-Verlag New York, Inc., 2002. Chap. RC 4000 Software: Multiprogramming System, pp. 153–197. ISBN: 0-387-95401-5. URL: http://dl.acm.org/citation.cfm?id=762971.762976.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds. *Structured Programming*. London, UK, UK: Academic Press Ltd., 1972. ISBN: 0-12-200550-3.

[BC05] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. ISBN: 0596005652.

[TJF13] Peter Thoman, Herbert Jordan, and Thomas Fahringer. "Adaptive Granularity Control in Task Parallel Programs Using Multiversioning". In: *Proceedings of the 19th International Conference on Parallel Processing*. Euro-Par'13. Aachen, Germany: Springer-Verlag, 2013, pp. 164–177. ISBN: 978-3-642-40046-9. DOI: 10.1007/978-3-642-40047-6_19. URL: http://dx.doi.org/10.1007/978-3-642-40047-6_19.

[Hoa74] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept". In: *Commun. ACM* 17.10 (Oct. 1974), pp. 549–557. ISSN: 0001-0782. DOI: 10.1145/355620.361161. URL: http://doi.acm.org/10.1145/355620.361161.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. "Concurrent control with "readers" and "writers"". In: *Commun. ACM* 14.10 (1971), pp. 667–668. ISSN: 0001-0782.

[Ayg+06] Eduard Ayguade et al. "Employing Nested OpenMP for the Parallelization of Multi-zone Computational Fluid Dynamics Applications". In: *J. Parallel Distrib. Comput.* 66.5 (May 2006), pp. 686–697. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2005.06.019. URL: http://dx.doi.org/10.1016/j.jpdc.2005.06.019.

[CYH10] Wen Ren Chen, Wuu Yang, and Wei Chung Hsu. "A lock-free cache-friendly software queue buffer for decoupled software pipelining". In: *Computer Symposium (ICS), 2010 International*. 2010, pp. 997–1006. DOI: 10.1109/COMPSYM.2010.5685364.

[Hoa78] C. A. R. Hoare. "Communicating Sequential Processes". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: http://doi.acm.org/10.1145/359576.359585.

[SR08] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment, Second Edition*. 2nd. Addison-Wesley Professional, 2008. ISBN: 0321525949, 9780321525949.

[JT01] Trent Jaeger and Jonathon E. Tidswell. "Practical Safety in Flexible Access Control Models". In: *ACM Trans. Inf. Syst. Secur.* 4.2 (May 2001), pp. 158–190. ISSN: 1094-9224. DOI: 10.1145/501963.501966. URL: http://doi.acm.org/10.1145/501963.501966.

[Chr88] T. W. Christopher. "Message Driven Computing and Its Relationship to Actors". In: *SIGPLAN Not.* 24.4 (Sept. 1988), pp. 76–78. ISSN: 0362-1340. DOI: 10.1145/67387.67405. URL: http://doi.acm.org/10.1145/67387.67405.

[BBU99] John Barkley, Konstantin Beznosov, and Jinny Uppal. "Supporting Relationships in Access Control Using Role Based Access Control". In: *Proceedings of the Fourth ACM Workshop on Role-based Access Control*. RBAC '99. Fairfax, Virginia, USA: ACM, 1999, pp. 55–65. ISBN: 1-58113-180-1. DOI: 10.1145/319171.319177. URL: http://doi.acm.org/10.1145/319171.319177.

[Fis+09] Jeffrey Fischer et al. "Fine-Grained Access Control with Object-Sensitive Roles". In: *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*. Genoa. Italy: Springer-Verlag, 2009, pp. 173–194. ISBN: 978-3-642-03012-3. DOI: 10.1007/978-3-642-03013-0_9. URL: http://dx.doi.org/10.1007/978-3-642-03013-0_9.

[PH00] Raju Pandey and Brant Hashii. "Providing fine-grained access control for Java programs via binary editing." In: *Concurrency - Practice and Experience* 12.14 (2000), pp. 1405–1430. URL: http://dblp.uni-trier.de/db/journals/concurrency/concurrency12.html#PandeyH00.

[Lis87] Barbara Liskov. "Keynote Address - Data Abstraction and Hierarchy". In: *Addendum to the Proceedings on Object-oriented Programming Systems, Lan-*

guages and Applications (Addendum). OOPSLA '87. Orlando, Florida, USA: ACM, 1987, pp. 17–34. ISBN: 0-89791-266-7. DOI: 10.1145/62138.62141. URL: http://doi.acm.org/10.1145/62138.62141.

[Sco09] Michael L. Scott. *Programming Language Pragmatics, Third Edition*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009. ISBN: 0123745144, 9780123745149.

[AT88] Mehmet Aksit and Anand Tripathi. "Data Abstraction Mechanisms in SINA/ST". In: *SIGPLAN Not.* 23.11 (Jan. 1988), pp. 267–275. ISSN: 0362-1340. DOI: 10.1145/62084.62107. URL: http://doi.acm.org/10.1145/62084.62107.

[PN04] Adriano Di Pasquale and Enrico Nardelli. "Scalable Distributed Data Structures: A Survey." In: *WDAS*. Ed. by Adriano Di Pasquale, Fabrizio Luccio, and Enrico Nardelli. Vol. 9. Proceedings in Informatics. Carleton Scientific, Jan. 5, 2004, pp. 87–111. ISBN: 1-894145-08-9. URL: http://dblp.uni-trier.de/db/conf/wdas/wdas2000.html#PasqualeN00a.

[Sha03] Gauri Shah. "Distributed Data Structures for Peer-to-peer Systems". PhD thesis. New Haven, CT, USA: Yale University, May 2003.

[Lar+08] D. Brian Larkins et al. "Global Trees: A Framework for Linked Data Structures on Distributed Memory Parallel Systems". In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. Austin, Texas: IEEE Press, 2008, 57:1–57:13. ISBN: 978-1-4244-2835-9. URL: http://dl.acm.org/citation.cfm?id=1413370.1413428.

[WL98] Jan-Jan Wu and Pangfeng Liu. "Distributed Data Structure Design for Scientific Computation". In: *Proceedings of the 12th International Conference on Supercomputing*. ICS '98. Melbourne, Australia: ACM, 1998, pp. 227–234. ISBN: 0-89791-998-X. DOI: 10.1145/277830.277879. URL: http://doi.acm.org/10.1145/277830.277879.

[App90] Andrew W. Appel. *A Runtime System*. 1990.

[Hin+11] Benjamin Hindman et al. "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 22–22. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488.

[WW96] Carl A. Waldspurger and William E. Weihl. "An object-oriented framework for modular resource management". In: *In IWOOOS. IEEE Computer Society*. 1996.

[Cao+02] Junwei Cao et al. "ARMS: An Agent-based Resource Managsement System for Grid Computing". In: *Sci. Program.* 10.2 (Apr. 2002), pp. 135–148. ISSN: 1058-9244. URL: http://dl.acm.org/citation.cfm?id=1239955.1239959.

[Int] *Intel Xeon Phi Coprocessor - the Architecture, howpublished = http://https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner, note = Accessed: 04-20-2014.*

[Til] *TILE-Gx Processor Family, howpublished = http://http://www.tilera.com/products/processors/TILE-Gx_Family, note = Accessed: 04-20-2014.*

[Bor07] Shekhar Borkar. "Thousand Core Chips: A Technology Perspective". In: *Proceedings of the 44th Annual Design Automation Conference*. DAC '07. San Diego, California: ACM, 2007, pp. 746–749. ISBN: 978-1-59593-627-1. DOI: 10.1145/1278480.1278667. URL: http://doi.acm.org/10.1145/1278480.1278667.

[Dan+10] Anthony Danalis et al. "The Scalable HeterOgeneous Computing (SHOC) benchmark suite". In: *in Proc. 3-rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU3*. 2010, pp. 63–74.

[Mar+07] C.A.M. Marcon et al. "Evaluation of Algorithms for Low Energy Mapping onto NoCs". In: *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*. 2007, pp. 389–392. DOI: 10.1109/ISCAS.2007.378471.

[Kob+11] Sebastian Kobbe et al. "DistRM: Distributed Resource Management for On-chip Many-core Systems". In: *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '11. Taipei, Taiwan: ACM, 2011, pp. 119–128. ISBN: 978-1-4503-0715-4. DOI: 10.1145/2039370.2039392. URL: http://doi.acm.org/10.1145/2039370.2039392.

[Jah+13] Janmartin Jahn et al. "Runtime Resource Allocation for Software Pipelines". In: *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*. M-SCOPES '13. St. Goar, Germany: ACM, 2013, pp. 96–99. ISBN: 978-1-4503-2142-6. DOI: 10.1145/2463596.2486156. URL: http://doi.acm.org/10.1145/2463596.2486156.

[Car+09] Ewerson Carvalho et al. "Evaluation of Static and Dynamic Task Mapping Algorithms in NoC-based MPSoCs". In: *Proceedings of the 11th International Conference on System-on-chip*. SOC'09. Tampere, Finland: IEEE Press, 2009, pp. 87–90. ISBN: 978-1-4244-4466-3. URL: http://dl.acm.org/citation.cfm?id=1736530.1736547.

[SLS07] Gerald Sabin, Matthew Lang, and P. Sadayappan. "Moldable Parallel Job Scheduling Using Job Efficiency: An Iterative Approach". In: *Proceedings of the 12th International Conference on Job Scheduling Strategies for Parallel Processing*. JSSPP'06. Saint-Malo, France: Springer-Verlag, 2007, pp. 94–114. ISBN: 978-3-540-71034-9. URL: http://dl.acm.org/citation.cfm?id=1757044.1757049.

[Bar+08] R. Baraglia et al. “Backfilling Strategies for Scheduling Streams of Jobs On Computational Farms”. In: *Making Grids Work*. Springer US, 2008, pp. 103–115. ISBN: 978-0-387-78447-2. DOI: 10.1007/978-0-387-78448-9_8. URL: http://dx.doi.org/10.1007/978-0-387-78448-9_8.

[Hu+14] Han Hu et al. “Toward Scalable Systems for Big Data Analytics: A Technology Tutorial”. In: *Access, IEEE* 2 (2014), pp. 652–687. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2014.2332453.

[Pan14] Robert Pan. “Ethernet versus Infiniband”. In: *Canadian Young Scientist Journal* 2014.2 (2014), pp. 11–11. DOI: 10.13034/cysj-2014-010. eprint: http://dx.doi.org/10.13034/cysj-2014-010. URL: http://dx.doi.org/10.13034/cysj-2014-010.

[CE94] Christopher Connelly and Carla Schlatter Ellis. *Workload Characterization and Locality Management for Coarse-Grain Multiprocessors*. Tech. rep. 1994.

[RK+13] Jonathan Ragan-Kelley et al. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’13. Seattle, Washington, USA: ACM, 2013, pp. 519–530. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462176. URL: http://doi.acm.org/10.1145/2491956.2462176.

[KRC97] M. Kandemir, J. Ramanujam, and A. Choudhary. *Compiler Algorithms for Optimizing Locality and Parallelism on Shared and Distributed Memory Machines*. 1997.

[Bik+06] Ganesh Bikshandi et al. “Programming for Parallelism and Locality with Hierarchically Tiled Arrays”. In: *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’06. New York, New York, USA: ACM, 2006, pp. 48–57. ISBN: 1-59593-189-9. DOI: 10.1145/1122971.1122981. URL: http://doi.acm.org/10.1145/1122971.1122981.

[Rin08] Martin C. Rinard. “Locality Optimizations for Parallel Computing Using Data Access Information.” In: *International Journal of High Speed Computing* 9.2 (July 8, 2008), pp. 161–179. URL: http://dblp.uni-trier.de/db/journals/ijhsc/ijhsc9.html#Rinard97.

[Fat+06] Kayvon Fatahalian et al. “Sequoia: Programming the Memory Hierarchy”. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC ’06. Tampa, Florida: ACM, 2006. ISBN: 0-7695-2700-0. DOI: 10.1145/1188455.1188543. URL: http://doi.acm.org/10.1145/1188455.1188543.

[KGV94] Vipin Kumar, Ananth Y. Grama, and Nageshwara Rao Vempaty. “Scalable Load Balancing Techniques for Parallel Computers”. In: *J. Parallel Distrib.*

Comput. 22.1 (July 1994), pp. 60–79. ISSN: 0743-7315. DOI: 10.1006/jpdc.1994.1070. URL: http://dx.doi.org/10.1006/jpdc.1994.1070.

[Blu+95] Robert D. Blumofe et al. "Cilk: An Efficient Multithreaded Runtime System". In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '95. Santa Barbara, California, USA: ACM, 1995, pp. 207–216. ISBN: 0-89791-700-6. DOI: 10.1145/209936.209958. URL: http://doi.acm.org/10.1145/209936.209958.

[Las+13] E. Laskowski et al. "Dynamic Load Balancing Based on Applications Global States Monitoring". In: *Parallel and Distributed Computing (ISPDC), 2013 IEEE 12th International Symposium on*. 2013, pp. 11–18. DOI: 10.1109/ISPDC.2013.11.

[SK93] A. Sinha and L.V. Kalé. "A Load Balancing Strategy for Prioritized Execution of Tasks". In: *Seventh International Parallel Processing Symposium*. Newport Beach, CA., 1993, pp. 230–237.

[WLR93] M.H. Willebeek-LeMair and A.P. Reeves. "Strategies for dynamic load balancing on highly parallel computers". In: *Parallel and Distributed Systems, IEEE Transactions on* 4.9 (1993), pp. 979–993. ISSN: 1045-9219. DOI: 10.1109/71.243526.

[AB12] H. Menon L. V. Kale F. Pellegrini A. Bhatele S. Fourestier. *Applying graph partitioning methods in measurement-based dynamic load balancing*. Tech. rep. University of Illinois,Dept. of Computer Science, 2012.

[Zhe+10] Gengbin Zheng et al. "Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers". In: *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*. ICPPW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 436–444. ISBN: 978-0-7695-4157-0. DOI: 10.1109/ICPPW.2010.65. URL: http://dx.doi.org/10.1109/ICPPW.2010.65.

[HWW93] Wilson C. Hsieh, Paul Wang, and William E. Weihl. "Computation Migration: Enhancing Locality for Distributed-memory Parallel Systems". In: *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '93. San Diego, California, USA: ACM, 1993, pp. 239–248. ISBN: 0-89791-589-5. DOI: 10.1145/155332.155357. URL: http://doi.acm.org/10.1145/155332.155357.

[KC98] V. Karamcheti and A. Chien. "A Hierarchical Load-Balancing Framework for Dynamic Multithreaded Computations". In: *Supercomputing, 1998.SC98. IEEE/ACM Conference on*. 1998, pp. 6–6. DOI: 10.1109/SC.1998.10047.

[BC03] Kevin J. Barker and Nikos P. Chrisochoides. "An Evaluation of a Framework for the Dynamic Load Balancing of Highly Adaptive and Irregular Parallel

Applications". In: *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. SC '03. Phoenix, AZ, USA: ACM, 2003, pp. 45–. ISBN: 1-58113-695-1. DOI: 10.1145/1048935.1050195. URL: http://doi.acm.org/10.1145/1048935.1050195.

[Ama+11a] Saman Amarasinghe et al. "Exascale Programming Challenges". In: *Proceedings of the Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA*. U.S Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), 2011.

[Fin+08] T. Finin et al. "ROWLBAC: Representing Role Based Access Control in OWL". In: *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*. SACMAT '08. Estes Park, CO, USA: ACM, 2008, pp. 73–82. ISBN: 978-1-60558-129-3. DOI: 10.1145/1377836.1377849. URL: http://doi.acm.org/10.1145/1377836.1377849.

[Sto11] Ivan Stojmenovic. "Access Control in Distributed Systems: Merging Theory with Practice". In: *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications* 0 (2011), pp. 1–2. DOI: http://doi.ieeecomputersociety.org/10.1109/TrustCom.2011.1.

[SDM11] John Shalf, Sudip Dosanjh, and John Morrison. "Exascale Computing Technology Challenges". In: *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*. VECPAR'10. Berkeley, CA: Springer-Verlag, 2011, pp. 1–25. ISBN: 978-3-642-19327-9. URL: http://dl.acm.org/citation.cfm?id=1964238.1964240.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986. ISBN: 0-262-01092-5.

[Bal+97] Satish Balay et al. "Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries". In: *Modern Software Tools in Scientific Computing*. Birkhauser Press, 1997, pp. 163–202.

[KK93] L.V. Kalé and S. Krishnan. "CHARM++: A Portable Concurrent Object Oriented System Based on C++". In: *Proceedings of OOPSLA'93*. Ed. by A. Paepcke. ACM Press, 1993, pp. 91–108.

[Les03] Norbert Leser. "The distributed computing environment naming architecture." In: *Distributed Systems Engineering* 1.1 (Mar. 24, 2003), pp. 19–28. URL: http://dblp.uni-trier.de/db/journals/dse/dse1.html#Leser93.

[Hen06] Michi Henning. "The Rise and Fall of CORBA". In: *Queue* 4.5 (June 2006), pp. 28–34. ISSN: 1542-7730. DOI: 10.1145/1142031.1142044. URL: http://doi.acm.org/10.1145/1142031.1142044.

[Lei99] Luke Kenneth Casson Leighton. *DCE/RPC over SMB: Samba and Windows NT: Domain Internals: An Expert Guide on Improving and Effeciency and*

Security on Windows NT. Thousand Oaks, CA, USA: New Riders Publishing, 1999. ISBN: 1578701503.

[Ter+84] Douglas B. Terry et al. *The Berkeley Internet Name Domain Server*. Tech. rep. UCB/CSD-84-182. EECS Department, University of California, Berkeley, 1984. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/1984/5957.html.

[WKN] Niklas Widell, Maria Kihl, and Christian Nyberg. *Measuring real-time performance in distributed-object-oriented systems*.

[BNM03] H. Brunst, W.E. Nagel, and A.D. Malony. "A distributed performance analysis architecture for clusters". In: *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*. 2003, pp. 73–81. DOI: 10.1109/CLUSTR.2003.1253301.

[LMK08] Chee Wai Lee, C. Mendes, and L.V. Kale. "Towards scalable performance analysis and visualization through data reduction". In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. 2008, pp. 1–8. DOI: 10.1109/IPDPS.2008.4536187.

[Hel+03] Aaron Helsinger et al. "Tools and Techniques for Performance Measurement of Large Distributed Multiagent Systems". In: *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS '03. Melbourne, Australia: ACM, 2003, pp. 843–850. ISBN: 1-58113-683-8. DOI: 10.1145/860575.860711. URL: http://doi.acm.org/10.1145/860575.860711.

[SM06] Sameer S. Shende and Allen D. Malony. "The Tau Parallel Performance System". In: *Int. J. High Perform. Comput. Appl.* 20.2 (May 2006), pp. 287–311. ISSN: 1094-3420. DOI: 10.1177/1094342006064482. URL: http://dx.doi.org/10.1177/1094342006064482.

[Gun08] Neil J. Gunther. *A general theory of computational scalability based on rational functions. arxiv.org/abs/0808.1431*. 2008.

[Kim+00] Jong-Kook Kim et al. "A QoS performance measure framework for distributed heterogeneous networks". In: *Parallel and Distributed Processing, 2000. Proceedings. 8th Euromicro Workshop on*. 2000, pp. 18–27. DOI: 10.1109/EMPDP.2000.823388.

[AKN04] Sherif Abdelwahed, Nagarajan Kandasamy, and Sandeep Neema. "A Control-based Framework for Self-managing Distributed Computing Systems". In: *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems*. WOSS '04. Newport Beach, California: ACM, 2004, pp. 3–7. ISBN: 1-58113-989-6. DOI: 10.1145/1075405.1075406. URL: http://doi.acm.org/10.1145/1075405.1075406.

[Tan94] Andrew Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1994. ISBN: 0132199084.

[Cri93] Flaviu Cristian. "Understanding Fault-Tolerant Distributed Systems". In: *Communications of the ACM* 34 (1993), pp. 56–78.

[Sch90] Fred B. Schneider. "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 299–319. ISSN: 0360-0300. DOI: 10.1145/98163.98167. URL: http://doi.acm.org/10.1145/98163.98167.

[Sch01] R. Schollmeier. "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications". In: *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*. 2001, pp. 101–102. DOI: 10.1109/P2P.2001.990434.

[GS97] Rachid Guerraoui and André Schiper. "Software-Based Replication for Fault Tolerance". In: *Computer* 30.4 (Apr. 1997), pp. 68–74. ISSN: 0018-9162. DOI: 10.1109/2.585156. URL: http://dx.doi.org/10.1109/2.585156.

[ZHK06] Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. "Performance Evaluation of Automatic Checkpoint-based Fault Tolerance for AMPI and Charm++". In: *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems* 40.2 (2006).

[ZSK04] Gengbin Zheng, Lixia Shi, and L.V. Kale. "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI". In: *Cluster Computing, 2004 IEEE International Conference on*. 2004, pp. 93–103. DOI: 10.1109/CLUSTR.2004.1392606.

[KMR08] G. Kandaswamy, A Mandal, and D.A Reed. "Fault Tolerance and Recovery of Scientific Workflows on Computational Grids". In: *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*. 2008, pp. 777–782. DOI: 10.1109/CCGRID.2008.79.

[Oli+07] Stephen Olivier et al. "UTS: An Unbalanced Tree Search Benchmark". In: *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*. LCPC'06. New Orleans, LA, USA: Springer-Verlag, 2007, pp. 235–250. ISBN: 978-3-540-72520-6. URL: http://dl.acm.org/citation.cfm?id=1757112.1757137.

[Har63] Theodore Edward Harris. *The theory of branching processes*. Berlin, Gttingen, Heidelberg, 1963. URL: http://opac.inria.fr/record=b1081408.

[Uts] *UTS Reference Implementation*. URL: {http://sourceforge.net/p/uts-benchmark/wiki/Home/}.

[BVH] Richard F Barrett, Courtenay T Vaughan, and Michael A Heroux. "MiniGhost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing". In: ().

[Man] *Mantevo*. URL: {https://mantevo.org}.

[Che+96] Thomas Cheatham et al. "Bulk synchronous parallel computinga paradigm for transportable software". In: *Tools and Environments for Parallel and Distributed Systems*. Springer, 1996, pp. 61–76.

[CCZ04] D. Callahan, B.L. Chamberlain, and H.P. Zima. "The cascade high productivity language". In: *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on*. 2004, pp. 52–60. DOI: 10.1109/HIPS.2004.1299190.

[Dia+] Roxana E. Diaconescu et al. *Reusable and Extensible High Level Data Distributions*.

[Coa+03] Cristian Coarfa et al. "Co-array Fortran performance and potential: An NPB experimental study". In: *In Proc. of the 16th Intl. Workshop on Languages and Compilers for Parallel Computing, number 2958 in LNCS*. Springer-Verlag, 2003, pp. 2–4.

[MC+09] John Mellor-Crummey et al. "A New Vision for Coarray Fortran". In: *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*. PGAS '09. Ashburn, Virginia: ACM, 2009, 5:1–5:9. ISBN: 978-1-60558-836-0. DOI: 10.1145/1809961.1809969. URL: http://doi.acm.org/10.1145/1809961.1809969.

[ESS04] Kemal Ebcioglu, Vijay Saraswat, and Vivek Sarkar. "X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access". In: *Proceedings of the International Workshop on Language Runtimes, OOPSLA*. 2004.

[ESS05] K Ebcioglu, V Saraswat, and V Sarkar. "X10: an experimental language for high productivity programming of scalable systems". In: *Proceedings of Workshop on Productivity and Performance in ...* (Jan. 2005). URL: http://dist.codehaus.org/x10/documentation/papers/P-PHEC05-paper.pdf.

[Sar+10] Vijay Saraswat et al. *The Asynchronous Partitioned Global Address Space Model*. Tech. rep. Toronto, Canada, 2010. URL: http://www.cs.rochester.edu/u/cding/amp/papers/full/The%20Asynchronous%20Partitioned%20Global%20Address%20Space%20Model.pdf.

[DBY05] Kaushik Datta, Dan Bonachea, and Katherine Yelick. "Titanium performance and potential: an NPB experimental study". In: *In proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC*. 2005.

[KSY05] Amir Kamil, Jimmy Su, and Katherine Yelick. "Making Sequential Consistency Practical in Titanium". In: *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 15–. ISBN: 1-59593-061-2. DOI: 10.1109/SC.2005.43. URL: http://dx.doi.org/10.1109/SC.2005.43.

[PNHY] K. Datta D. Gay S. L. Graham B. R. Liblit G. Pike J. Z. Su P. N. Hilfinger D. O. Bonachea and K. A. Yelick. *Titanium language reference manual. Technical Report UCB/EECS-2005-15.1*. Tech. rep.

[HIY03] Parry Husbands, Costin Iancu, and Katherine Yelick. "A Performance Analysis of the Berkeley UPC Compiler". In: *Proceedings of the 17th Annual International Conference on Supercomputing*. ICS '03. San Francisco, CA, USA: ACM, 2003, pp. 63–73. ISBN: 1-58113-733-8. DOI: 10.1145/782814.782825. URL: http://doi.acm.org/10.1145/782814.782825.

Appendices

A Terminologies

BSP: Bulk Synchronous Processing.

Exaflops: 10E18 floating points operations per second.

FIFO: First In First Out.

First Class Entity: Objects with a name.

FLOPS: Floating Point Operations Per Second.

HPC: High Performance Computing.

IPC: Inter Process Communication.

LCO: Local Control Objects (synchronizing constructs).

LIFO: Last In First Out.

Locality: 1) A synchronous domain in conventional sense; 2) Temporal/Spatial Locality.

Node: A Synchronous domain; in context of application, a vertex of a graph.

NUMA: Non Uniform Memory Access.

OS: Operating System

Processes: Named objects encapsulating a procedure definition.

SLOW: Starvation, latency, overheads, waiting for contention resolution.

SMP: Symmetric Multi-Processing.

SPMD: Single Program Multiple Data.

B Experiment Results

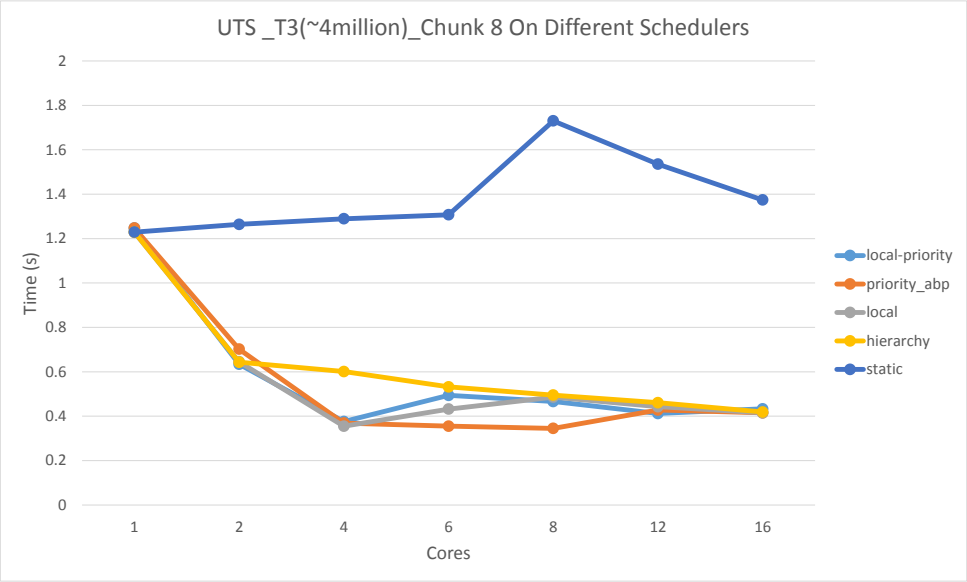

Figure B.1: Performance Measurement of UTS benchmark in HPX with chunk size 8 and problem size of approx. 4 million nodes for Binomial(3) graph type, using different scheduling policies on Marvin node.

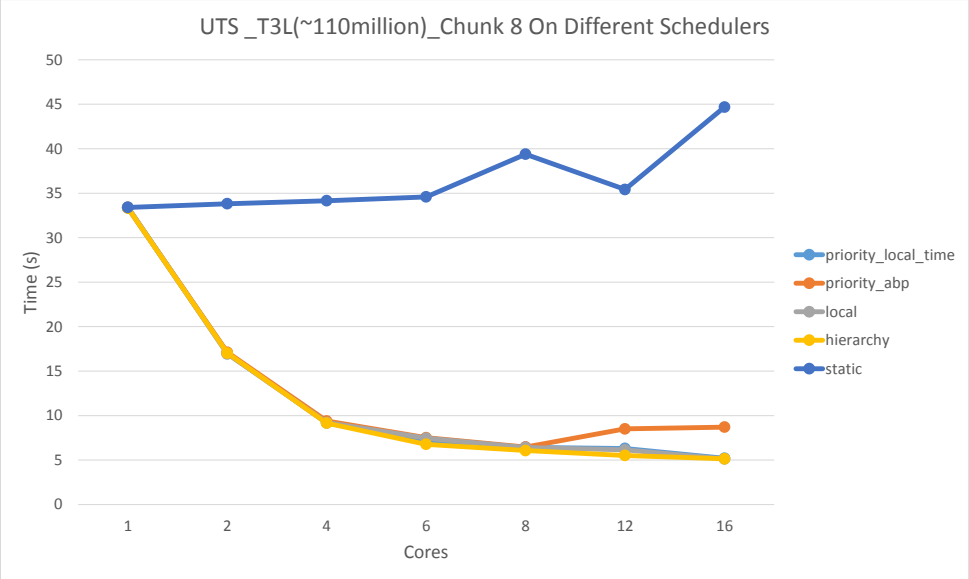

Figure B.2: Performance Measurement of UTS benchmark in HPX with chunk size 8 and problem size of approx. 110 million nodes for Binomial(3) graph type, using different scheduling policies on Marvin node.

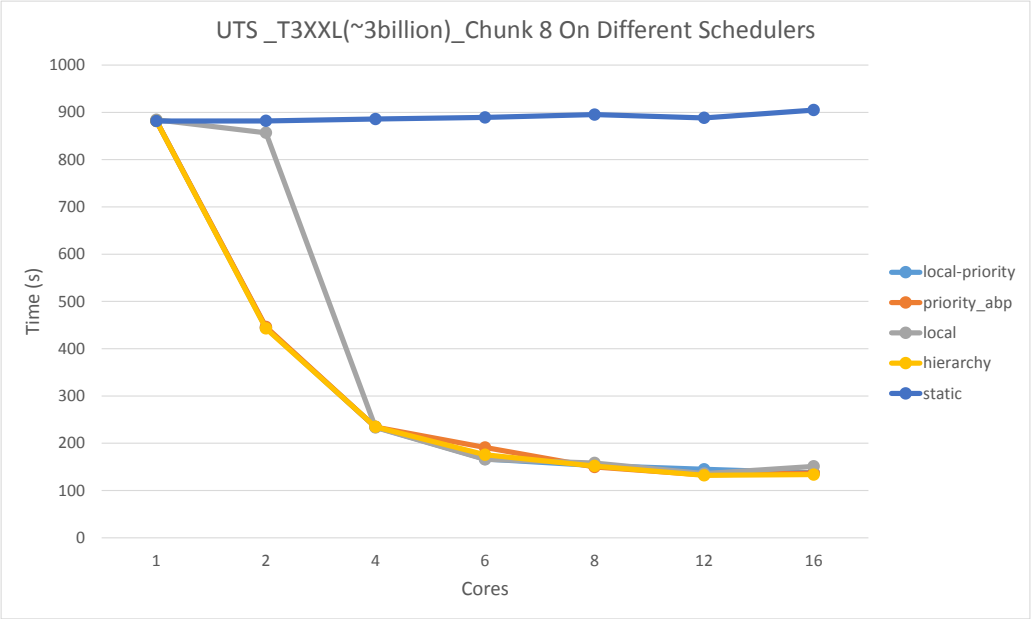

Figure B.3: Performance Measurement of UTS benchmark in HPX with chunk size 8 and problem size of approx. 3 billion nodes for Binomial(3) graph type, using different scheduling policies on Marvin node.

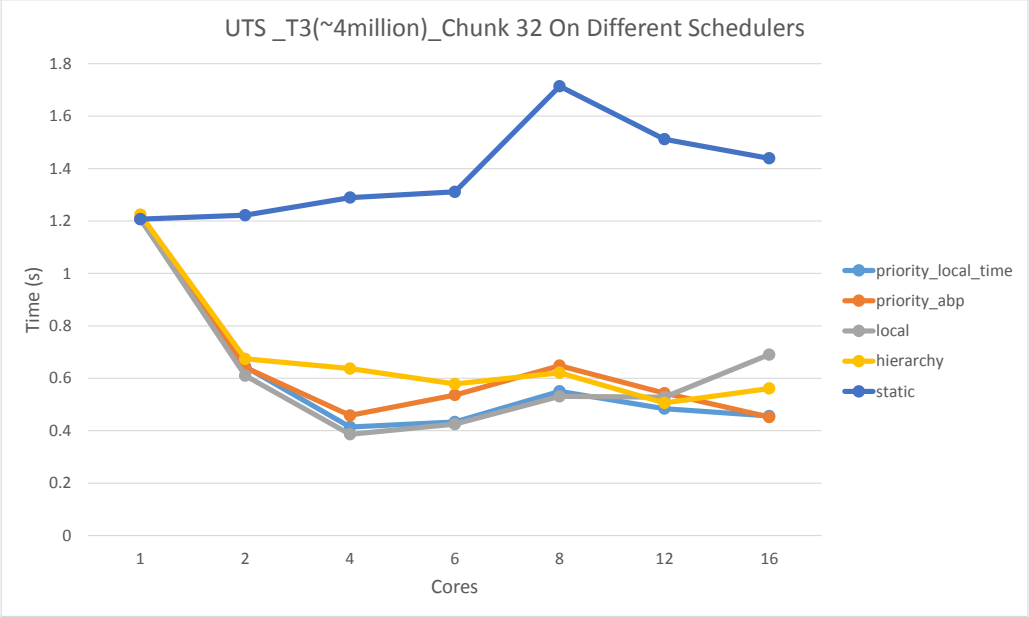

Figure B.4: Performance Measurement of UTS benchmark in HPX with chunk size 32 and problem size of approx. 4 million nodes for Binomial(3) graph type, using different scheduling policies on Marvin node.

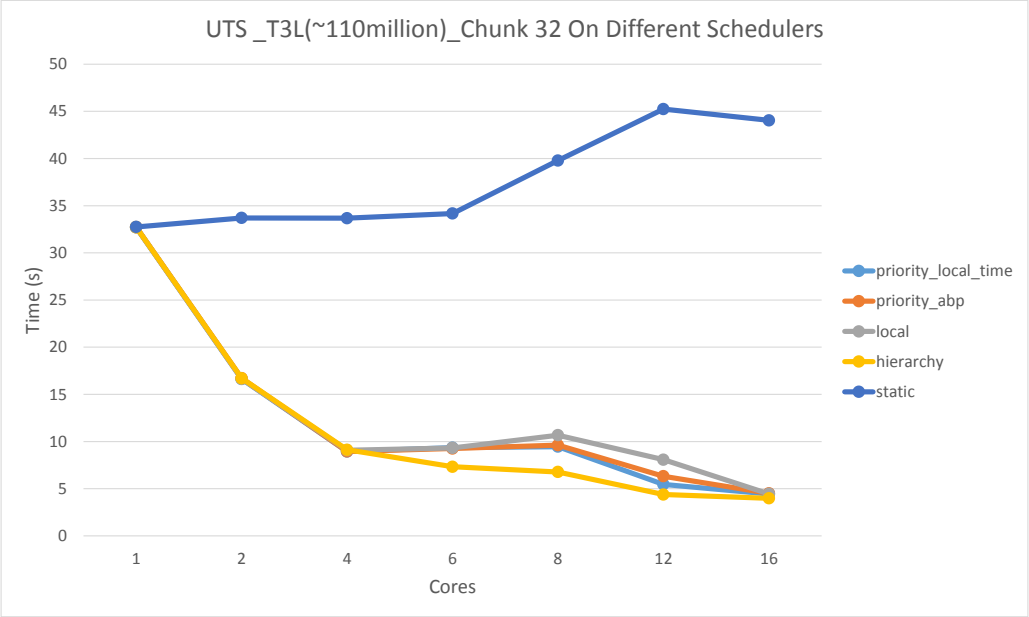

Figure B.5: Performance Measurement of UTS benchmark in HPX with chunk size 32 and problem size of approx. 110 million nodes for Binomial(3) graph type, using different scheduling policies on Marvin node.

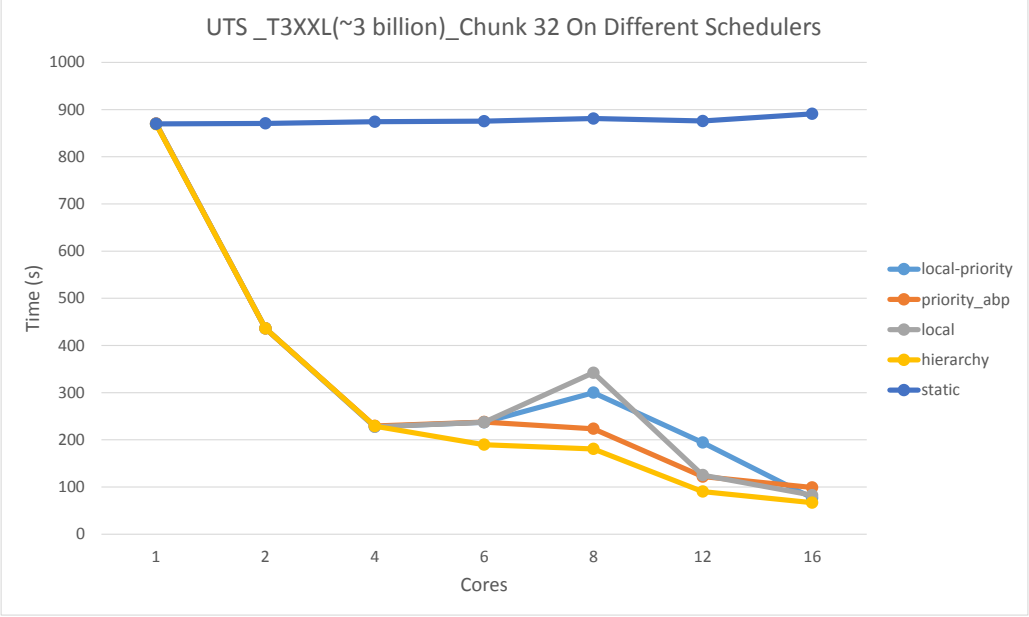

Figure B.6: Performance Measurement of UTS benchmark in HPX with chunk size 32 and problem size of approx. 3 billion nodes for Binomial(3) graph type, using different scheduling policies on Marvin node.

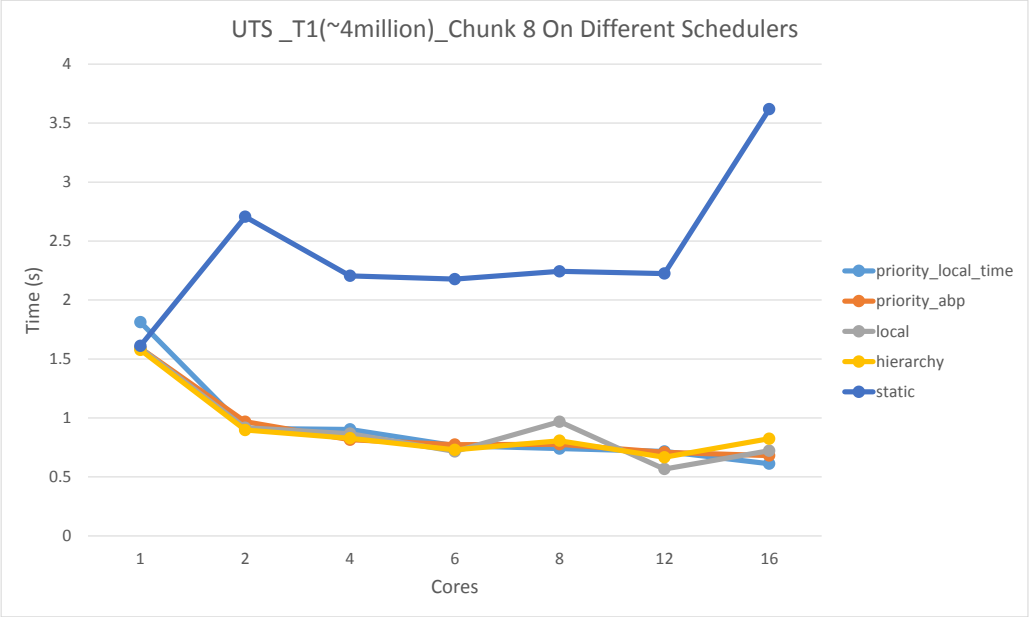

Figure B.7: Performance Measurement of UTS benchmark in HPX with chunk size 8 and problem size of approx. 4 million nodes for Geometric(1) graph type, using different scheduling policies on Marvin node.

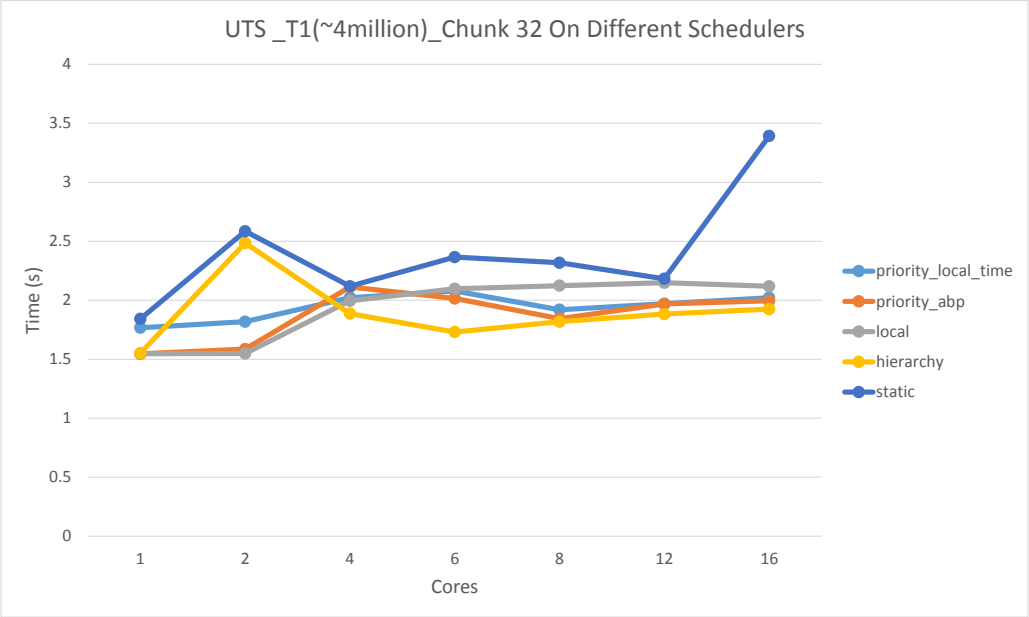

Figure B.8: Performance Measurement of UTS benchmark in HPX with chunk size 32 and problem size of approx. 4 million nodes for Geometric(1) graph type, using different scheduling policies on Marvin node.

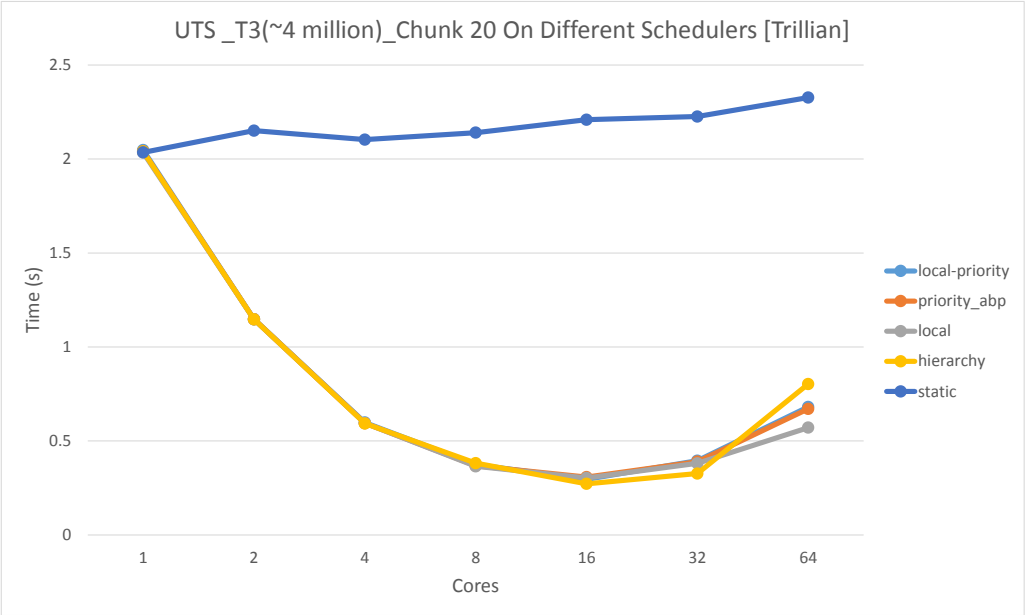

Figure B.9: Performance Measurement of UTS benchmark in HPX with chunk size 20 and problem size of approx. 4 million nodes for Binomial(3) graph type, using different scheduling policies on Trillian node.

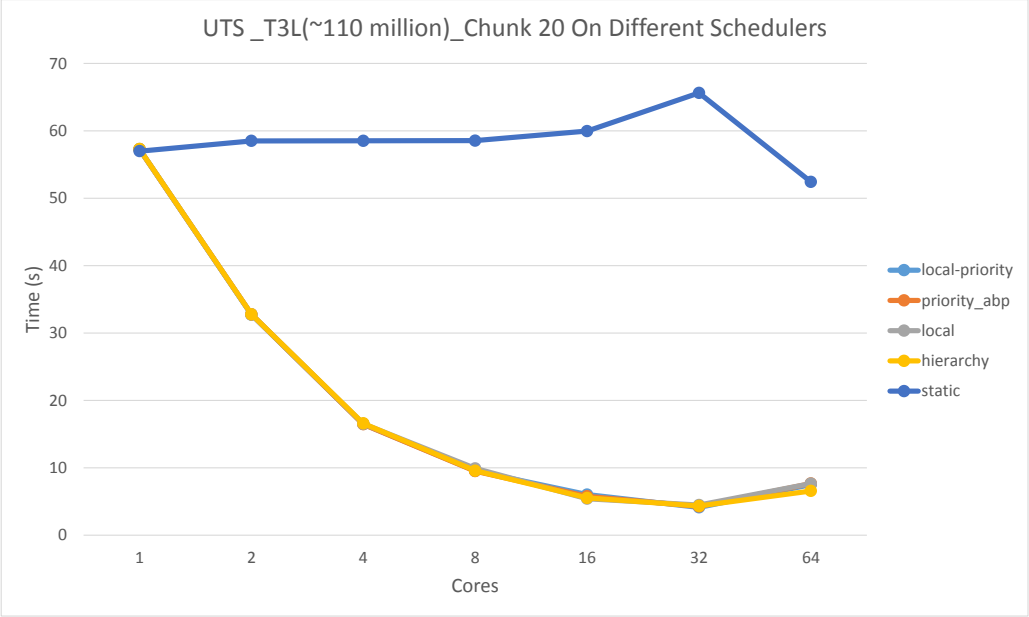

Figure B.10: Performance Measurement of UTS benchmark in HPX with chunk size 20 and problem size of approx. 110 million nodes for Binomial(3) graph type, using different scheduling policies on Trillian node.

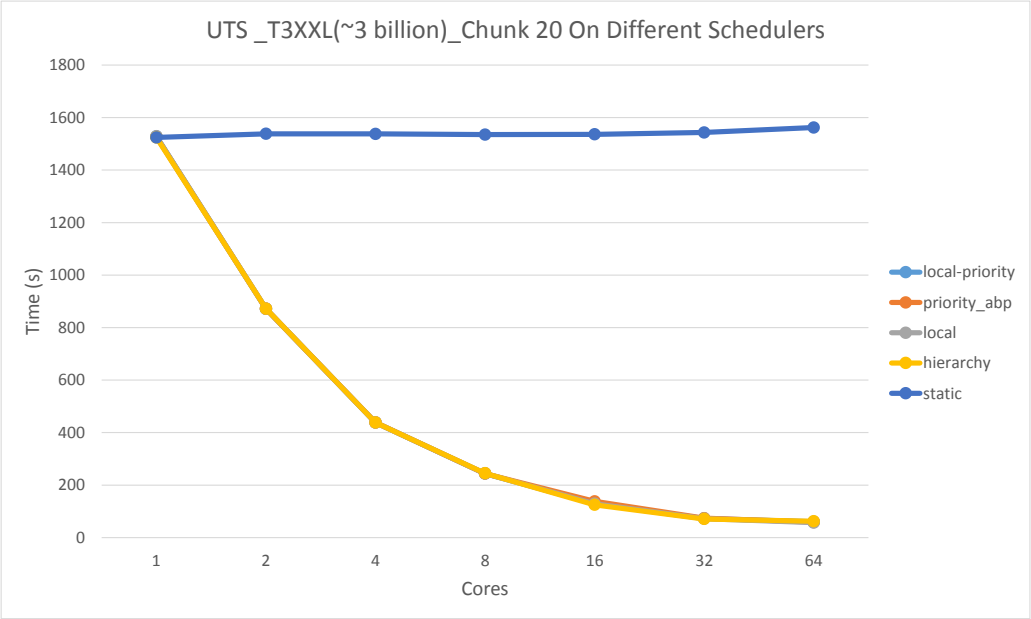

Figure B.11: Performance Measurement of UTS benchmark in HPX with chunk size 20 and problem size of approx. 3 billion nodes for Binomial(3) graph type, using different scheduling policies on Trillian node.

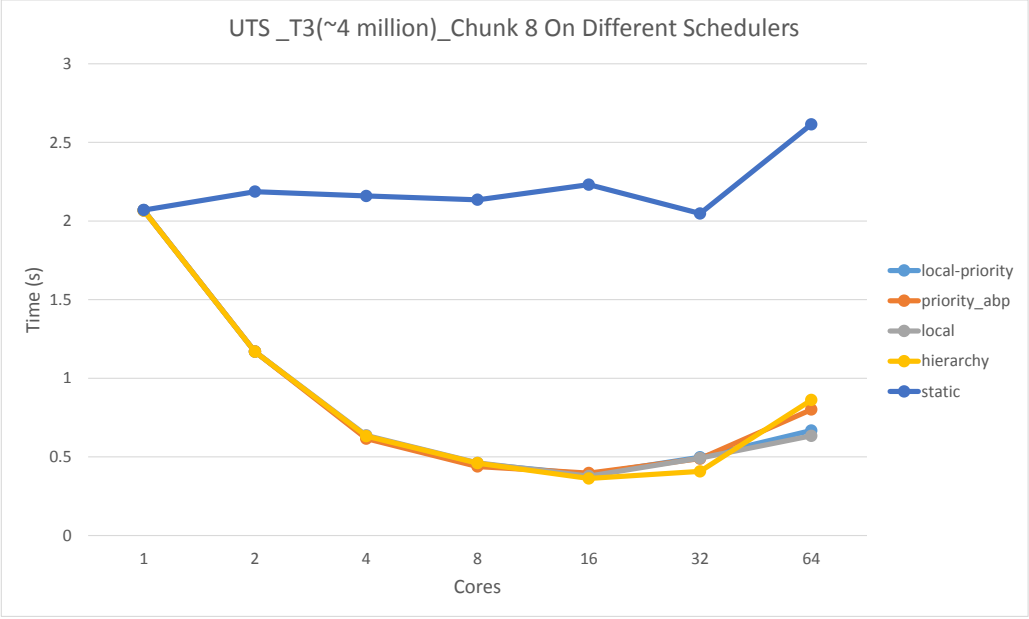

Figure B.12: Performance Measurement of UTS benchmark in HPX with chunk size 8 and problem size of approx. 4 million nodes for Binomial(3) graph type, using different scheduling policies on Trillian node.

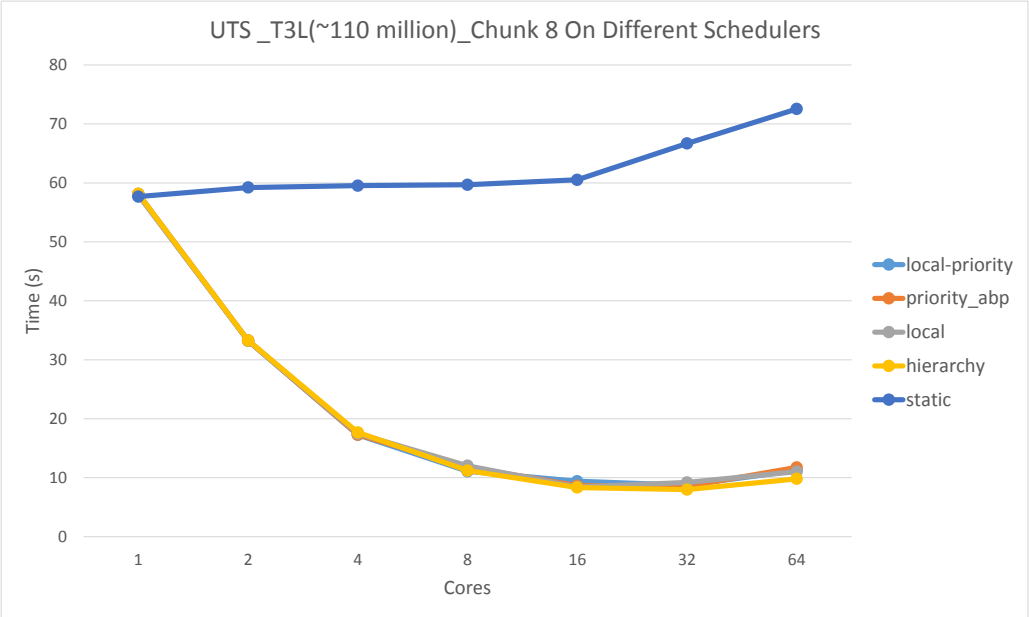

Figure B.13: Performance Measurement of UTS benchmark in HPX with chunk size 8 and problem size of approx. 110 million nodes for Binomial(3) graph type, using different scheduling policies on Trillian node.

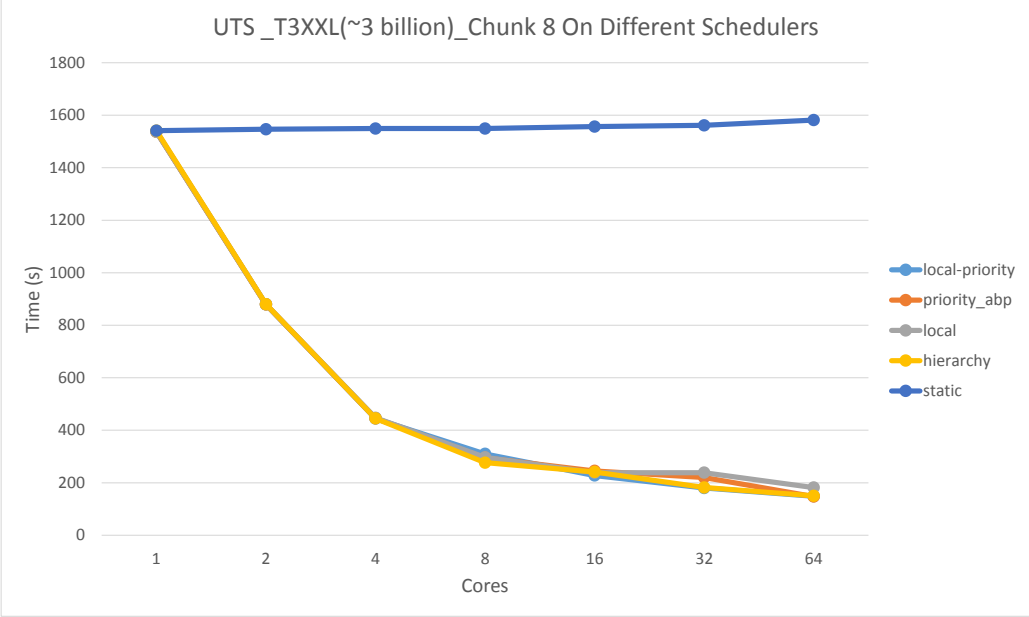

Figure B.14: Performance Measurement of UTS benchmark in HPX with chunk size 8 and problem size of approx. 3 billion nodes for Binomial(3) graph type, using different scheduling policies on Trillian node.

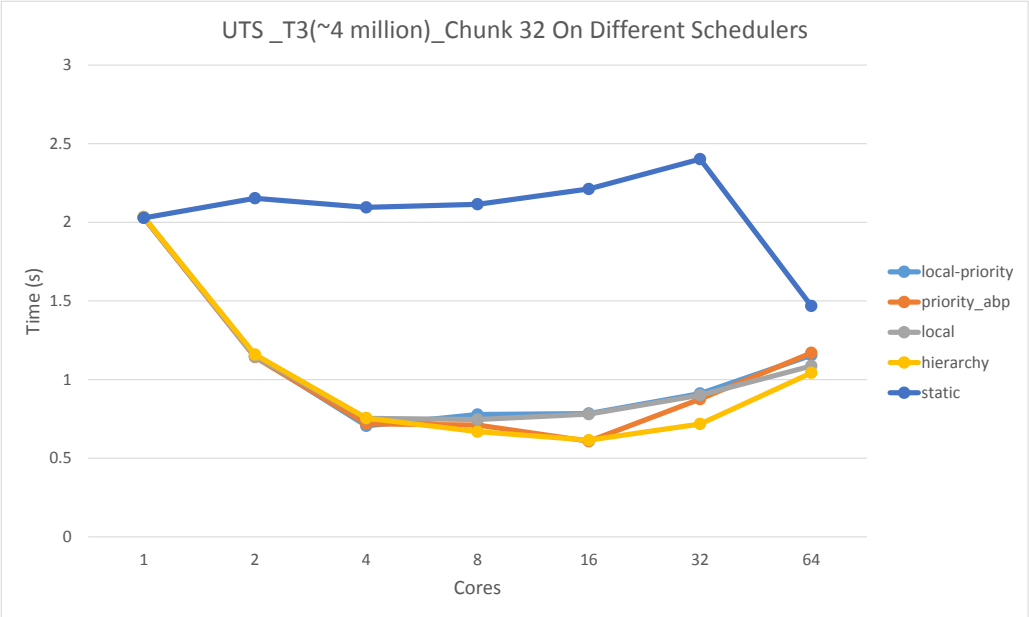

Figure B.15: Performance Measurement of UTS benchmark in HPX with chunk size 32 and problem size of approx. 4 million nodes for Binomial(3) graph type, using different scheduling policies on Trillian node.

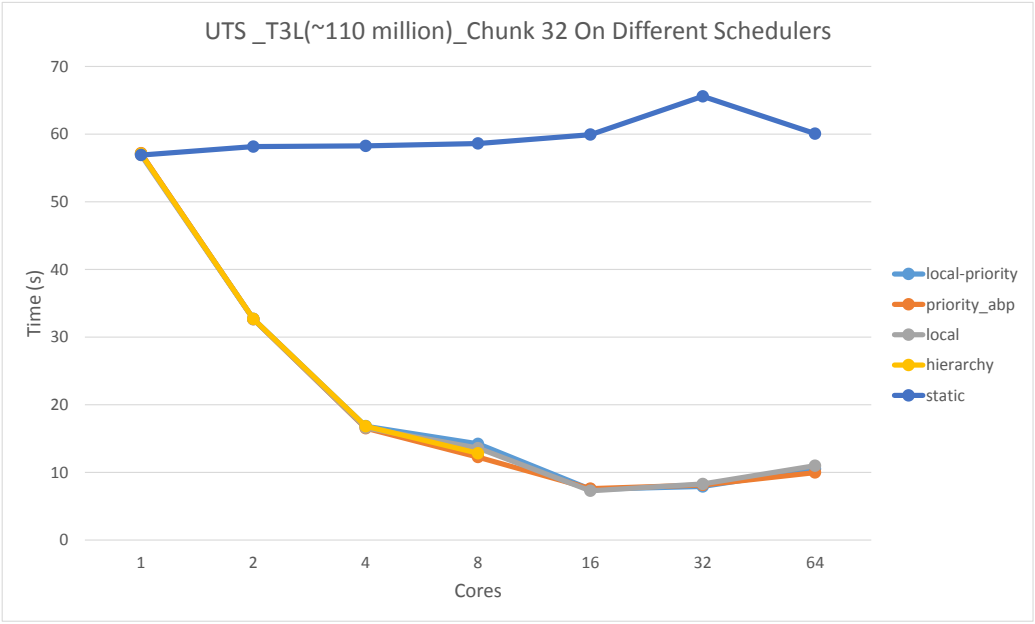

Figure B.16: Performance Measurement of UTS benchmark in HPX with chunk size 32 and problem size of approx. 110 million nodes for Binomial(3) graph type, using different scheduling policies on Trillian node.

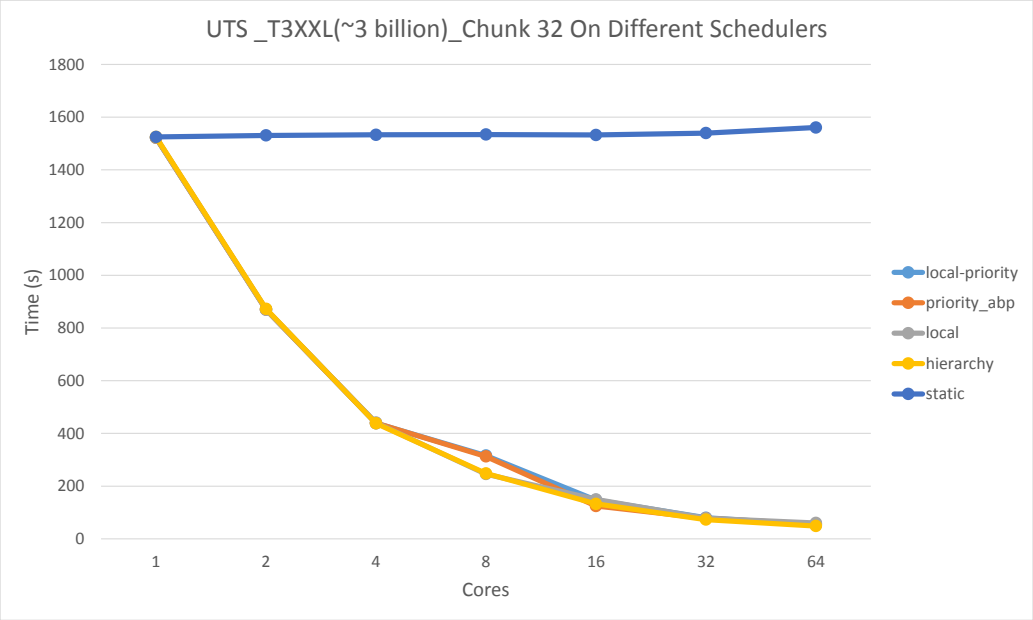

Figure B.17: Performance Measurement of UTS benchmark in HPX with chunk size 32 and problem size of approx. 3 billion nodes for Binomial(3) graph type, using different scheduling policies on Trillian node.

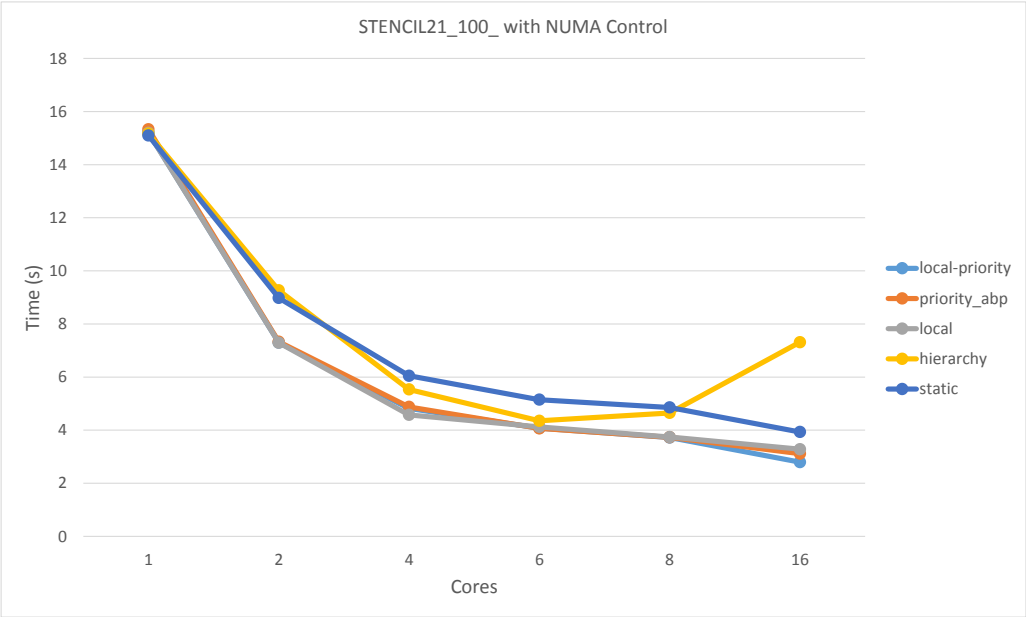

Figure B.18: Performance Measurement of MiniGhost benchmark application, using stencil 21 and gridsize 100 in HPX for different scheduling policies.

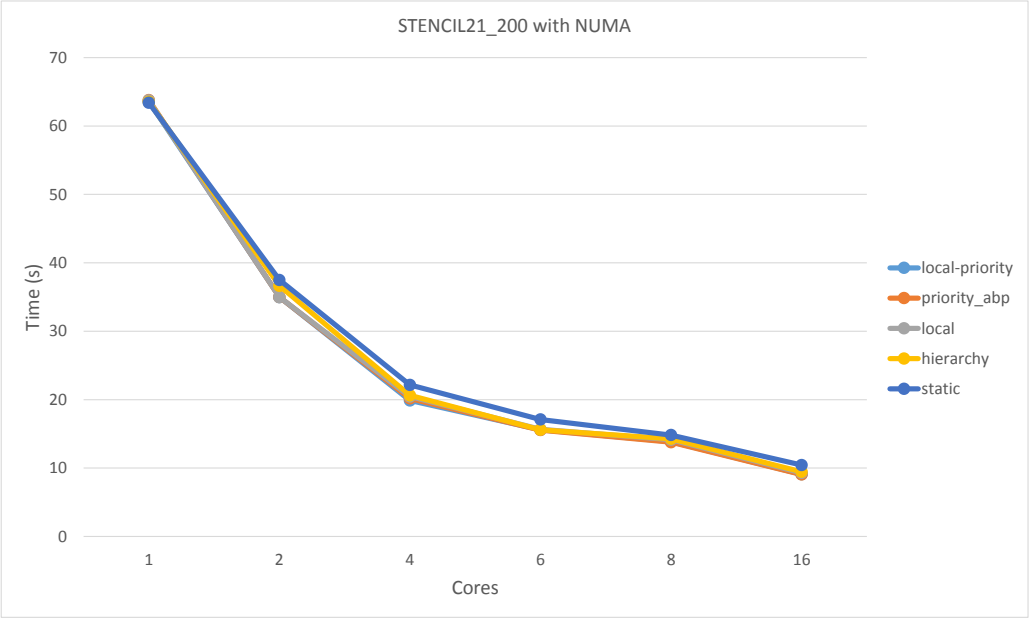

Figure B.19: Performance Measurement of MiniGhost benchmark application, using stencil 21 and gridsize 200 in HPX for different scheduling policies.

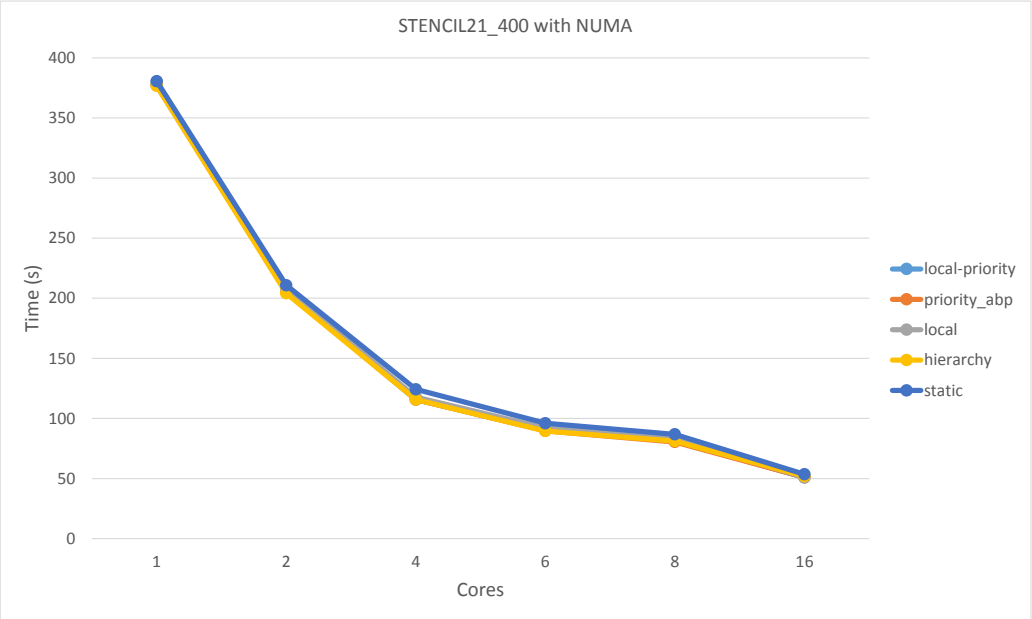

Figure B.20: Performance Measurement of MiniGhost benchmark application, using stencil 21 and gridsize 400 in HPX for different scheduling policies.

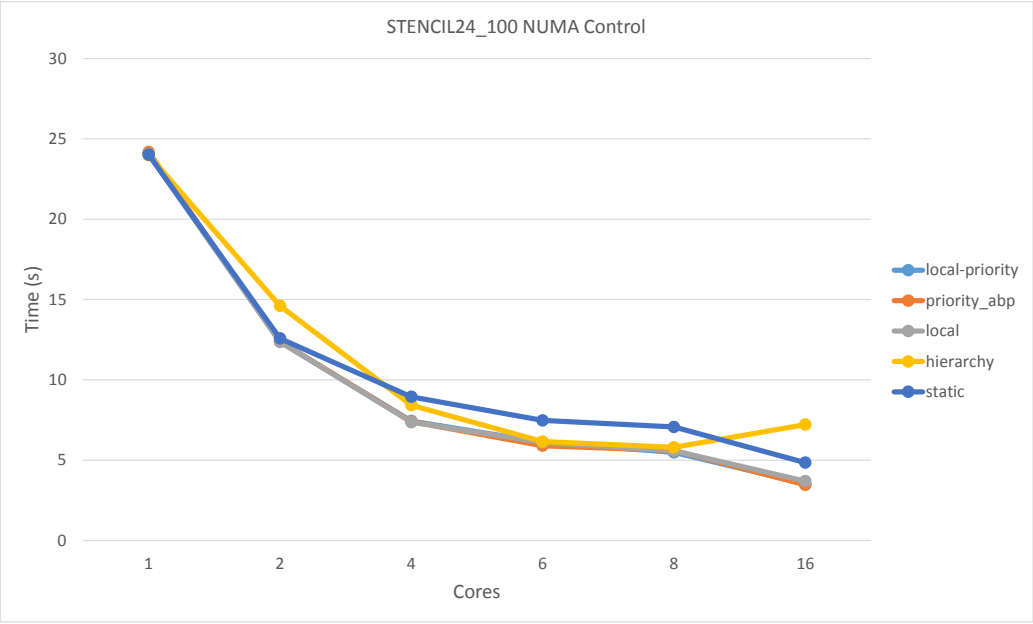

Figure B.21: Performance Measurement of MiniGhost benchmark application, using stencil 24 and gridsize 100 in HPX for different scheduling policies.

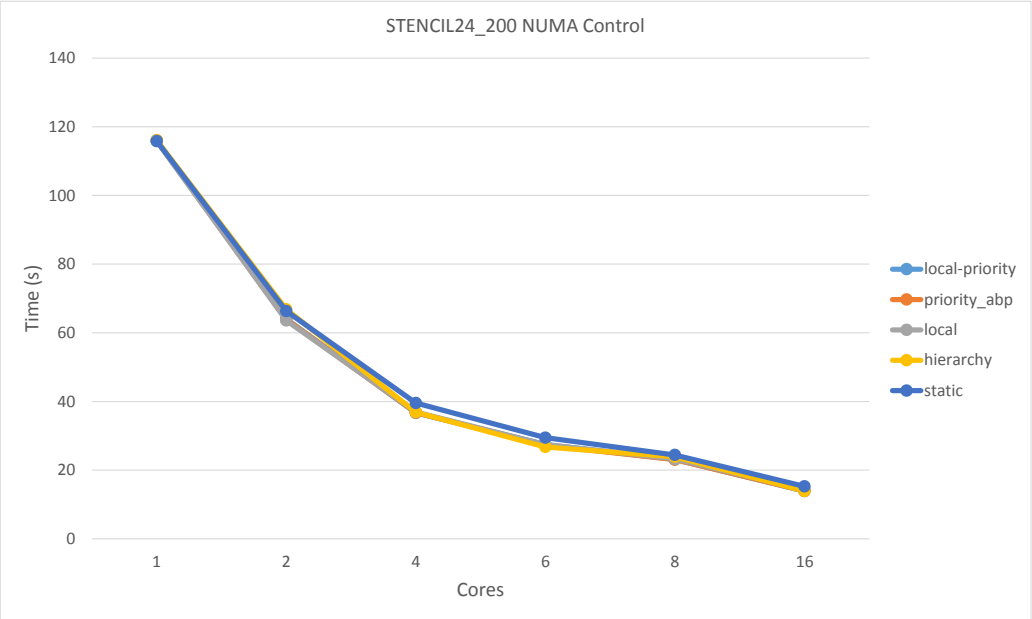

Figure B.22: Performance Measurement of MiniGhost benchmark application, using stencil 24 and gridsize 200 in HPX for different scheduling policies.

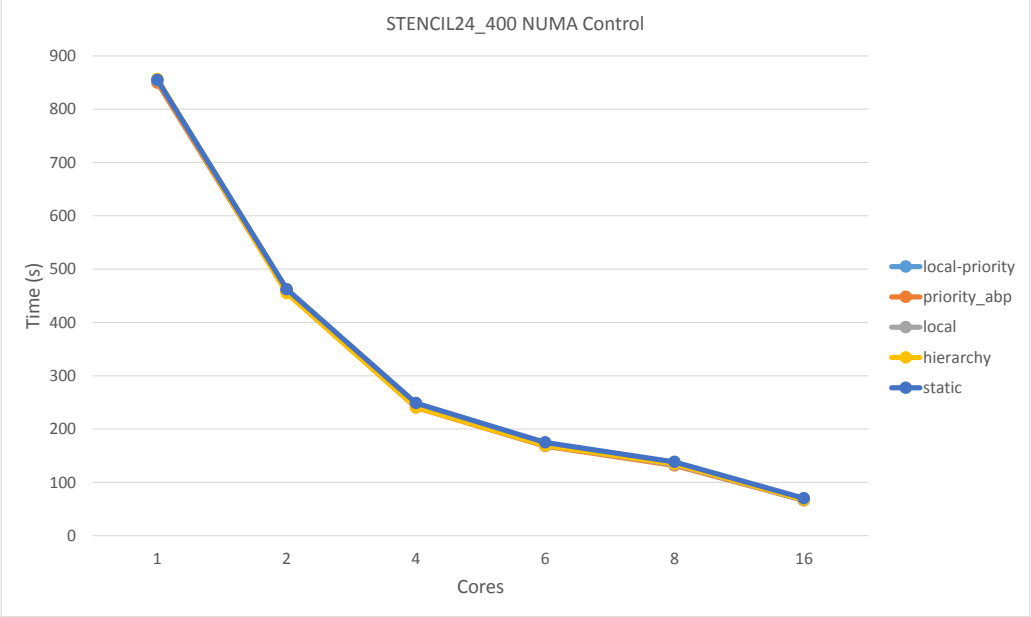

Figure B.23: Performance Measurement of MiniGhost benchmark application, using stencil 24 and gridsize 400 in HPX for different scheduling policies.

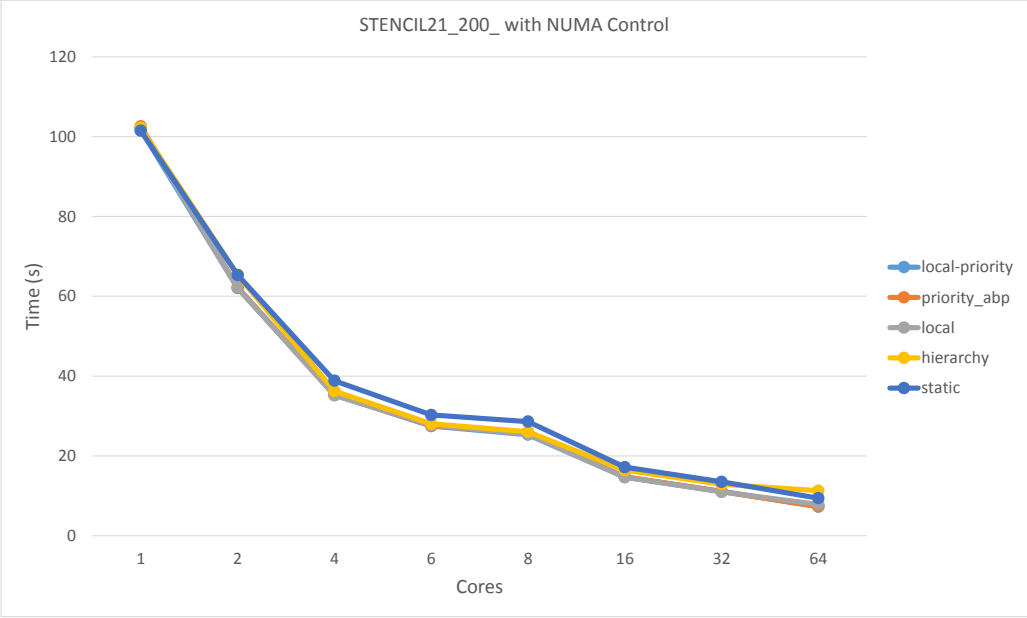

Figure B.24: Performance Measurement of MiniGhost benchmark application, using stencil 21 and gridsize 200 in HPX for different scheduling policies.

Vita

Vinay Chandra Amatya was born in Lalitpur, Nepal, in 1982. He completed his B.E. in 2005 from Institute of Engineering, Tribhuvan University, Nepal, majoring in Electrical Engineering. He completed his M.S. in System Science in 2012 at Louisiana State University. While at Louisiana State University, he had the opportunity to work as graduate researcher in several projects and groups namely Cybertools Workpackage 1 and Workpakage 3; ParalleX group and Stellar group. He is currently a candidate for degree of Doctor of Philosophy in Computer Science, which will be awarded in December 2014.