HPX – A Task Based Programming Model in a Global Address Space

Hartmut Kaiser[1]
hkaiser@cct.lsu.edu

Thomas Heller[2]
thomas.heller@cs.fau.de

Bryce Adelstein-Lelbach[1]
blelbach@cct.lsu.edu

Adrian Serio[1]
aserio@cct.lsu.edu

Dietmar Fey[2]
dietmar.fey@cs.fau.de

[1]Center for Computation and Technology,
Louisiana State University,
Louisiana, U.S.A.

[2]Computer Science 3,
Computer Architectures,
Friedrich-Alexander-University,
Erlangen, Germany

ABSTRACT

The significant increase in complexity of Exascale platforms due to energy-constrained, billion-way parallelism, with major changes to processor and memory architecture, requires new energy-efficient and resilient programming techniques that are portable across multiple future generations of machines. We believe that guaranteeing adequate scalability, programmability, performance portability, resilience, and energy efficiency requires a fundamentally new approach, combined with a transition path for existing scientific applications, to fully explore the rewards of todays and tomorrows systems. We present HPX – a parallel runtime system which extends the C++11/14 standard to facilitate distributed operations, enable fine-grained constraint based parallelism, and support runtime adaptive resource management. This provides a widely accepted API enabling programmability, composability and performance portability of user applications. By employing a global address space, we seamlessly augment the standard to apply to a distributed case. We present HPX's architecture, design decisions, and results selected from a diverse set of application runs showing superior performance, scalability, and efficiency over conventional practice.

Keywords

High Performance Computing, Parallel Runtime Systems, Exascale, Programming Models, Global Address Space

1. INTRODUCTION

Todays programming models, languages, and related technologies that have sustained High Performance Computing (HPC) application software development for the past decade are facing major problems when it comes to programmability and performance portability of future systems. The significant increase in complexity of new platforms due to energy constrains, increasing parallelism and major changes to processor and memory architecture, requires advanced programming techniques that are portable across multiple future generations of machines [1].

A fundamentally new approach is required to address these challenges. This approach must emphasize the following attributes: *Scalability* – enable applications to strongly scale to Exascale levels of parallelism; *Programmability* – clearly reduce the burden we are placing on high performance programmers; *Performance Portability* – eliminate or significantly minimize requirements for porting to future platforms; *Resilience* – properly manage fault detection and recovery at all components of the software stack; *Energy Efficiency* – maximally exploit dynamic energy saving opportunities, leveraging the tradeoffs between energy efficiency, resilience, and performance.

The biggest disruption in the path to Exascale will occur at the intra-node level, due to severe memory and power constraints per core, many-fold increase in the degree of intra-node parallelism, and to the vast degrees of performance and functional heterogeneity across cores. These challenges clearly point to radically new approaches to intra-node runtime systems. Functions expected from novel, self-aware, resilient runtime systems are the autonomic, runtime-dynamic management of resources, dynamic load balancing, intrinsic latency hiding mechanisms, management of data movement and locality, active power management, and detection and recovery from faults [1]. At the same time, the demand for homogeneous programming interfaces which simplify code development and abstract hardware specifics additionally requires computer scientist to rethink the way inter-node parallelism is exposed to the programmers.

Advanced runtime systems that support new programming models and languages which make billion-way parallelism manageable are needed. We believe that a combination a global address space together with task based parallelism let's us rethink parallelization and focus on programmability instead of message passing. We extend the notion of a Partitioned Global Address Space (PGAS) by an active component which results in a Active Global Address Space. This extensions allows us to transparently migrate Objects in the Global Address between different nodes in our Supercomputer. This allows us to support efficient and dynamic communication, synchronization, scheduling, task placement and data migration, as well as the autonomic management of resources, identifying and reacting to load imbalances and the intermittent loss of resources – all of which are major research challenges to be addressed.

This paper outlines HPX as a potential solution to efficiently utlize billion-way parallelism using a global address space. We will begin our discussion in Sec. 2 where we will demonstrate that the concepts of our execution model are aligned with the current trends in state-of-the-art programming models without succumbing to each technology's shortcomings. We will discuss the governing design principles of HPX in Sec. 3. In Sec. 4, we will present results from several experiments that we have conducted and thereby substantiate our claim of viability. Finally we will conclude, in Sec. 5, with brief summary and the focus of our future work.

2. THE STATE OF THE ART

The latest research in the field of parallel programming models points to the efficient utilization of intra-node parallelism with the ability to exploit the underlying communication infrastructure using fine grain task based approaches to deal with concurrency [1, 2, 3, 4, 5].

In recent years, programming models targeting lightweight tasks are more and more commonplace. HPX is no exception to this trend. Task based parallel programming models can be placed into one of several different categories: *Library solutions-* examples of those are Intel TBB [6], Microsoft PPL [7], Qthreads [2], StarPU [8], and many others; *Language extensions-* examples here are Intel Cilk Plus [9], or OpenMP 3.0 [10]; and *Experimental programming languages-* most notable examples are Chapel [11], Intel ISPC [12], or X10 [13].

While all of the current solutions expose parallelism to the user in different ways, they all seem to converge on a continuation style programming model. Some of them use a futures-based programming model, while others use data-flow processing of depending tasks with an implicit or explicit DAG representation of the control flow. While the majority of the task based programming models focus on dealing with node level parallelism, HPX presents a solution for homogeneous execution of remote and local operations.

When looking at library solutions for task-based programming, we are mainly looking at C/C++. While other languages such as Java and Haskell provide similar libraries, they play a secondary role in the field of High Performance Computing. Fortran, for example, has only one widely available solution, OpenMP 3.0. However, it is trivial to call Fortran kernels from C/C++ or to create Fortran language bindings for the use library. Examples of pure C solutions are StarPU and Qthreads. They both provide interfaces for starting and waiting on lightweight tasks as well as creating task dependencies. While Qthreads provides suspendable user level threads, StarPU is built upon Codelets which, by definition, run to completion without suspension. Each of these strategies, Codelets and User-Level-Threads, have their advantages. HPX is a C++ library and tries to stay fully inside the C++ memory model and its execution scheme. For this reason, it follows the same route as Qthreads to allow for more flexibility and for easier support for synchronization mechanisms. For C++, one of the existing solutions is Intel TBB, which like StarPU, works in a codelet-style execution of tasks. What these libraries have in common is that they provide a high performance solution to task-based parallelism with all requirements one could think about for dealing with intra-node level parallelism. What they clearly lack is a uniform API and a solution for dealing with distributed parallel computing. This is one of the key advantages HPX has over these solutions: A programming API which conforms to the C++11 and the upcoming C++14 standards [14, 15, 16] but is augmented and extended to support remote operations.

Hand in hand with the library-based solutions are the pragma-based language extensions for C, C++ and Fortran: They provide an efficient and effective way for programmers to express intra-node parallelism. While OpenMP has supported tasks since V3.0, it lacks support for continuation-based programming and task dependencies, focusing instead on fork-join parallelism. Task dependencies were introduced in OpenMP 4.0. This hole is filled by OmPSs [17], which serves as an experiment in integrating inter-task dependencies using a pragma-based approach. One advantage of pragma-based solutions over libraries is their excellent support for accelerators. This effort was spearheaded by OpenACC [18] and is now part of the OpenMP 4.0 specification. Libraries for accelerators, as of now, have to fall back to language extensions like C++AMP [19], CUDA [20] or program directly in low level OpenCL [21].

In addition to language extensions, an ongoing effort to develop new programming languages is emerging, aiming at better support for parallel programming. Some parallel programming languages like OpenCL or Intel Cilk Plus [9] are focusing on node level parallelism. While OpenCL focuses on abstracting hardware differences for all kinds of parallelism, Intel Cilk Plus supports a fork-join style of parallelism. In addition, there are programming languages which explicitly support distributed computing, like UPC [22] or Fortress [23] but lack of support for intra-node level parallelism. Current research, however, is developing support for both, inter and intra-node level parallelism based on a global partitioned address space (PGAS [24]). The most prominent examples are Chapel [11] and X10 [13] which represent the PGAS languages. HCMPI [25] shows similarities with the HPX programming model by offering interfaces for asynchronous distributed computing, either based on distributed data driven futures or explicit message passing in an MPI [26] compatible manner. The main difference between HPX and the above solutions is that HPX invented no new syntax or semantics. Instead, HPX implemented the syntax and semantics as defined by C++11, providing it with a homogeneous API that relies on a widely accepted programming interface.

Many applications must overcome the scaling limitations imposed by current programming practices by embracing an entirely new way of coordinating parallel execution. Fortunately, this does not mean that we must abandon all of our legacy code. HPX can use MPI as a highly efficient portable communication platform and at the same time serve as a back-end for OpenMP, OpenCL, or even Chapel while maintaining or even improving execution times. This opens a migration path for legacy codes to a new programming model which will allow old and new code to coexist in the same application.

3. HPX – A GENERAL PURPOSE PARALLEL RUNTIME SYSTEM

HPX is a general purpose C++ runtime system for parallel and distributed applications of any scale. We will describe it in this section in more detail.

With the general availability of Petascale clusters and the advent of heterogeneous machines equipped with special accelerator cards such as the Intel Xeon Phi or GPGPUs, computer scientists face the difficult task of improving application scalability beyond what is possible with conventional techniques and programming models today. In addition, the need for highly adaptive runtime algorithms and for applications handling highly inhomogeneous data further impedes our ability to efficiently write code which performs and scales well. In this paper, we refer to the main factors that prevent scaling as the **SLOW** factors: a) ***S****tarvation*, i.e. current concurrent work is insufficient to maintain high utilization of all resources, b) ***L****atencies*, i.e. the delays intrinsic to accessing remote resources and services deferring their responses, c) ***O****verheads*, i.e. the work required for the management of parallel actions and resources on the critical execution path which is not necessary in a sequential variant, and d) **W**aiting for *Contention* resolution, which is caused by the delays due to oversubscribed shared resources.

We posit that in order to tackle the challenges of **SLOW**, a completely new execution model is required. This model must overcome the limitations of how applications are written today and make the full parallelization capabilities of contemporary and emerging heterogeneous hardware available to the application programmer in a simple and homogeneous way. We have designed HPX to implement such an execution model.

HPX represents an innovative mixture of a global system-wide address space, fine grain parallelism, and lightweight synchronization combined with implicit, work queue based, message driven computation, full semantic equivalence of local and remote execution, and explicit support for hardware accelerators through percolation.

Our initial results with implementing different types of applications using HPX have been exceedingly promising. Sec. 4 will contain a discussion of results of selected benchmarks.

3.1 HPX Design Principles

HPX is a novel combination of well-known ideas with new unique overarching concepts [27, 28, 29]. It aims to resolve the problems related to scalability, resiliency, power efficiency, and runtime adaptive resource management that will be of growing importance as HPC architectures evolve from Peta- to Exascale. It departs from today's prevalent programming models with the goal of mitigating their respective limitations, such as implicit and explicit global barriers, coarse grain parallelism, and lack of overlap between computation and communication (or the high complexity of its implementation). HPX exposes a coherent programming model, unifying all different types of parallelism in HPC systems. HPX is the first open source software runtime system implementing the concepts of the ParalleX execution model [27, 30, 31] on conventional systems including Linux clusters, Windows, Macintosh, Android, XeonPhi, and the Bluegene/Q. HPX is built using existing ideas and concepts, each known for decades as outlined below, however we believe that the combination of those ideas and their strict application forming overarching design principles is what makes HPX unique.

Focus on Latency Hiding instead of Latency Avoidance
It is impossible to design a system exposing zero latencies. In an effort to come as close as possible to this goal many optimizations are targeted towards minimizing latencies. Examples for this can be seen everywhere, for instance low latency network technologies like InfiniBand [32], complex memory hierarchies in all modern processors, the constant optimization of existing MPI implementations to reduce network latencies, or the data transfer latencies intrinsic to the way we use GPGPUs today. It is important to note, that existing latencies are often tightly related to some resource having to wait for the operation to be completed. This idle time could be used instead to do useful work, which would allow the application to hide the latencies from the user. Modern systems already employ similar techniques (pipelined instruction execution in the processor cores, asynchronous input/output operations, and many more). We propose to go further than what has been accomplished today and make latency hiding an intrinsic concept of the operation of the whole system stack.

Embrace Fine-grained Parallelism instead of Heavyweight Threads
If we plan to hide latencies even for very short operations, such as fetching the contents of a memory cell from main memory (if it is not already cached), we need to have very light-weight threads with extremely short context switching times, optimally executable within one cycle. Granted, for mainstream architectures this is not possible today (even if there are special machines supporting this mode of operation, such as the Cray XMT [33]). For conventional systems however, the smaller the overhead of a context switch and the finer the granularity of the threading system, the better will be the overall system utilization and its efficiency. For today's architectures we already see a flurry of libraries providing exactly this type of functionality (even if not directly targetting HPC): non-preemptive, work queue based parallelization solutions, such as Intel Threading Building Blocks (TBB, [6]), Microsoft Parallel Patterns Library (PPL, [7]), Cilk [34], and many others. The possibility to suspend a task if some preconditions for its execution are not met (such as waiting for I/O or the result of a different task) – while seamlessly switching to any other task which can continue in the meantime – and to reschedule the initial work after the required result has been calculated, makes the implementation of latency hiding almost trivial.

Rediscover Constraint Based Synchronization to replace Global Barriers
Most code written today is riddled with implicit (and explicit) global barriers. i.e. the synchronization of the control flow between several (very often all) threads (when using OpenMP [35, 36]) or processes (MPI [26]). For instance, an implicit global barrier is inserted after each loop parallelized using OpenMP as the system synchronizes the threads used to execute the different parallel iterations. In MPI, almost each of the communication steps imposes an explicit barrier onto the execution flow as (often all) nodes have to be synchronized. Each of those barriers acts as an eye of the needle the overall execution is forced to be squeezed through. Even minimal fluctuations in the execution times of the parallel threads (processes) causes them to wait. Additionally, often only one of the executing threads is doing the actual reduce operation, which further impedes parallelism. A closer analysis of a couple of key algorithms used in scientific applications reveals that these global

barriers are not always necessary. In many cases, it is sufficient to synchronize a small subset of the executing tasks. In general, any operation could proceed as soon as the preconditions for its execution are met. For example, there is no need to wait for all iterations of a (parallel) loop to finish before continuing the calculation of other things. The computation can continue after only those iterations are done which were producing the required results for a particular next operation. This type of computing was described in the 1970s, based on the theory of static and dynamic data-flow [37, 38]. There are certain attempts today to get back to those ideas and to incorporate them with modern architectures. For instance, a lot of work is being done in the area of constructing data-flow oriented execution trees. Our results show that employing data-flow techniques in combination with the other ideas, considerably improves scalability for many problems.

Adaptive Locality Control instead of Static Data Distribution
While this principle seems to be a given for single desktop or laptop computers, it is everything but ubiquitous on modern supercomputers, which are usually built from a large number of separate nodes (i.e. Beowulf clusters), tightly interconnected by a high bandwidth, low latency network. Today's prevalent programming model for those is MPI which does not directly help with proper data distribution and data placement, leaving it to the programmer to decompose the data to all of the nodes the application is running on. There are a couple of specialized languages and programming environments based on PGAS designed to overcome this limitation, such as Chapel, X10, UPC, or Fortress. However all systems based on PGAS rely on static data distribution. This works fine as long as such a static data distribution does not result in inhomogeneous workload distributions or other resource utilization imbalances. In a distributed system these imbalances can be mitigated by migrating part of the application data to different localities (nodes). The only framework supporting (limited) migration today is Charm++ [39]. The first attempts towards solving related problems go back decades as well, a good example is the Linda coordination language [40]. Nevertheless, none of the other mentioned systems support fully dynamic migration of arbitrary data today, which forces the users to either rely on static data distribution and live with the related performance hits or to implement everything themselves, which is very tedious and difficult. We believe that the only viable way to flexibly support dynamic and adaptive locality control is to provide a global, uniform address space to the applications, even on distributed systems. This should be combined with a flexible introspection system allowing the uniform gathering of a variety of performance information which feeds into policy based, decision engines which dynamically control data placement at runtime.

Prefer Moving Work to the Data over Moving Data to the Work
For best performance, it seems obvious to minimize the amount of bytes transferred from one part of the system to another. This is true on all levels. At the lowest level we try to take advantage of processor memory caches, thus minimizing memory latencies. Similarly, we try to amortize the data transfer time to and from GPGPUs as much as possible. At high levels we try to minimize data transfer between different nodes of a cluster or between different virtual machines on the cloud. Our experience (well, it's almost common wisdom) shows that the amount of bytes necessary to encode a certain operation is very often much smaller than the amount of bytes encoding the data the operation is performed upon. Nevertheless we still often transfer the data to a particular place where we execute the operation just to bring the data back to where it came from afterwards. As an example, let us look at the way we usually write our applications for clusters using MPI. This programming model is all about data transfer between nodes. MPI is the prevalent programming model for clusters, it is fairly straightforward to understand and to use. Therefore, we often write the applications in a way accommodating this model, centered around data transfer. These applications usually work well for smaller problem sizes and for regular data structures. The larger the amount of data we have to churn and the more irregular the problem domain becomes, the worse are the overall machine utilization and the (strong) scaling characteristics. While it is not impossible to implement more dynamic, data driven, and asynchronous applications using MPI, it is overly difficult to do so. At the same time, if we look at applications preferring to execute the code close to the locality where the data was placed, i.e. utilizing active messages (for instance based on Charm++), we see better asynchrony, simpler application codes, and improved scaling.

Favor Message Driven Computation over Message Passing
Today's prevalently used programming model on parallel (multinode) systems is MPI. It is based on message passing (as the name implies), which means that the receiver has to be aware of a message about to come in. Both codes, the sender and the receiver, have to synchronize in order to perform the communication step. Even the newer, asynchronous interfaces require the explicit coding of the algorithms around the required communication scheme. As a result, any nontrivial MPI application spends a considerable amount of time waiting for incoming messages, thus causing starvation and latencies to impede full resource utilization. The more complex and the more dynamic the data structures and algorithms become, the larger are the adverse effects. The community has discovered message-driven and (data-driven) methods of implementing algorithms a long time ago, and systems such as Charm++ already have integrated active messages, demonstrating the validity of the concept. Message driven computation allows one to send messages without the need for the receiver to actively wait for them. Any incoming message is handled asynchronously and triggers the encoded action by passing along arguments and – possibly – continuations. HPX combines this scheme with work queue based scheduling as described above, which allows almost complete overlap of communication with useful work, reducing effective latencies to a minimum.

3.2 The Architecture of HPX

HPX is a runtime system. This means that any application using it will be directly linked with its libraries. While an operating system is active for the whole time a machine is running, a runtime system is started whenever the application is launched and it will be shut down whenever the application is terminated. A runtime system's task is to improve certain overall runtime characteristics of the application, like performance, energy efficiency, or scalability, etc. At the same time, the runtime system relies on and expands services provided by the operating system. The resulting architecture of HPX is shown in Fig. 1. It consists of five subsystems, all of which encapsulate a particular subset of the exposed functionality.

3.2.1 The Parcel Subsystem

All network communication in HPX is built on top of the parcel subsystem. It implements a form of active messages [41] called *parcels*. Parcels encapsulate remote method calls. A parcel contains the global address of an object to act on (the destination), a reference to one of the object's methods, the arguments to call the

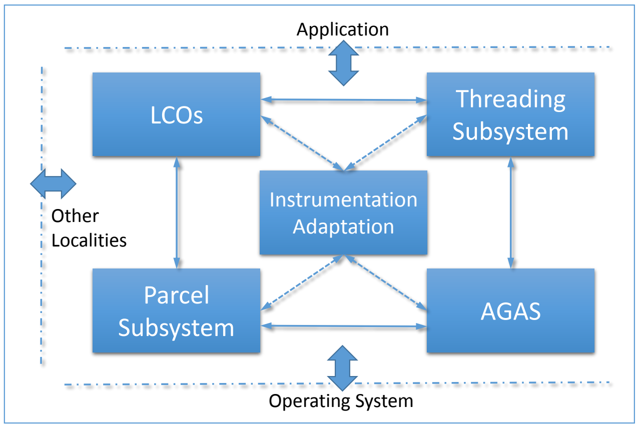

Figure 1: The modular structure of HPX implementation. HPX consists of the following subsystems: AGAS (Active Global Address Space), Parcel port and Parcel handlers, HPX-threads and thread manager, LCOs (Local Control Objects), and Performance Counters for the instrumentation of system and user code.

method with, and an optional continuation. Parcels are transmitted asynchronously. Data is communicated in the form of arguments bound to the parcel. A parcel can be conceptually viewed as a bound function. In HPX, the entities that communicate parcels are called *localities*. A locality represents a set of hardware resources with bounded, finite latencies (for example, a single socket in a large SMP machine, or a compute node). Each locality asynchronously receives inbound parcels and transmits outbound parcels.

3.2.2 The Active Global Address Space (AGAS)

The primary role of AGAS is to implement a global address space that spans all localities an application is currently running on. The target of a parcel is a global address assigned by AGAS, called a Global Identifier (GID). GIDs are mapped by AGAS to a tuple of meta-data that can be used to dynamically locate an object in the system. Unlike systems such as X10, Chapel, or UPC, which are based on PGAS, AGAS is a dynamic and adaptive address space which evolves over the lifetime of an application, similar to systems such as DEGAS [42] and Charm++. AGAS is the foundation for dynamic locality control as objects can be moved to other localities without changing their global address.

3.2.3 The Threading Subsystem

When a parcel is received by the parcel-port, it is converted into an HPX-thread, which is then scheduled, executed and recycled by the threading subsystem.In the HPX threading subsystem, M HPX-threads are mapped onto N kernel threads (OS-threads), typically one OS-thread per available processing unit (core). This threading model enables fine-grained parallelization, with the goal of using millions of threads per second on each core. Context switches between HPX-threads do not require a kernel call, reducing the overhead of HPX-thread execution and suspension. The threading subsystem uses a work-queue based execution strategy, utilizing work stealing to facilitate intra-locality load balancing. HPX-threads are scheduled cooperatively; they are never preempted by the scheduler. However they may voluntarily suspend themselves when they must wait for data required to continue execution, I/O operations, or synchronization.

3.2.4 Local Control Objects (LCOs)

LCOs provide a means of controlling parallelization and synchronization in HPX applications, and implement latency hiding. Any object that may create a new HPX-thread or reactivate a suspended HPX-thread exposes the functionality of an LCO. Support for event-driven HPX-threading, protection of shared resources, and organization of execution flow are provided by LCOs. LCOs are designed to replace global barriers with constraint-based synchronization, allowing for each thread to proceed as far in its execution as possible without active blocking or waiting. Some of the more prominent LCOs provided by HPX are described below.

- *Futures* [43, 44, 45] are proxies for results that are not yet known, possibly because they have not yet been computed. A future synchronizes access to the result value associated with it by suspending HPX-threads requesting the value if the data is not available at the time of the request. When the result becomes available, the future reactivates all suspended HPX-threads waiting for it.
- *Dataflow objects* [38, 46, 47] provide a powerful mechanism for managing data dependencies without the use of global barriers. A dataflow LCO waits for a set of futures to become ready and triggers a predefined function passing along all values encapsulated by the input futures.
- *Traditional concurrency control mechanisms* such as various types of mutexes [48], counting semaphores, spinlocks, condition variables and barriers are also exposed as LCOs in HPX.
- *Suspended threads* fulfill the criteria of being an LCO as well: once triggered they cause a thread (themselves) to be resumed

3.2.5 Instrumentation and Adaptivity

HPX implements a performance counter framework which provides an intrusive way 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 runtime introspection to make smarter decisions facilitating runtime adaptive resource management. Additionally, performance counters are a powerful debugging and optimization tool. A performance counter is a first class object (has a global address) associated with a symbolic name which exposes an uniform interface for collecting arbitrary performance data on demand throughout the system. Also, external instrumentation utilities can connect to a running HPX application through the parcel transport layer, query performance counters, and then disconnect.

3.3 The HPX API – Strictly Conforming to the C++ Standard

The design of the API exposed by HPX is aligned as much as possible with the latest C++11 Standard [14], the (draft) C++14 Standard [15], and related proposals to the standardization committee [16, 49, 50]. HPX implements all interfaces defined by the C++ Standard related to multi-threading (such as `future`, `thread`, `mutex`, or `async`) in a fully conforming way. These interfaces were accepted for ISO standardization after a wide community based discussion and since then have proven to be effective tools for managing asynchrony. HPX seeks to extend these concepts, interfaces, and ideas embodied in the C++11 threading system to distributed and data-flow programming use cases. Nevertheless, we made every possible effort to keep all of the implementation of HPX fully conforming to C++, which ensures a high degree of code portability, and – as will be shown below – enables a high degree of performance portability of HPX applications as well.

HPX's API aligns very well with the asynchronous nature of the implemented execution model. Any operation which possibly involves network access is exposed as an asynchronous function invocation returning an instance of a `future`. This `future` represents the result of the operation and can be used to either synchronize with it or to attach a continuation which will be automatically executed once the `future` becomes ready. Any exceptions thrown during the (possibly remote) function execution are dispatched back to the `future` which enables proper error handling.

The full function invocation API of HPX is presented in Table 1. This API exposes three conceptually different ways of executing a function, locally on the same physical locality as the invocation site or remotely on a different locality. As shown in this table, the HPX function invocation capabilities are based upon – and extend beyond – what is possible in C++ and the C++ Standards library today.

- *Synchronous function execution:* this is the most natural way of invoking a C++ function. The caller 'waits' for the function to return, possibly providing the result of the function execution. In HPX, synchronously executing an action suspends the current thread relinquishing the processing unit for other available work. Once the function is executed, the current thread is rescheduled.
- *Asynchronous function execution:* asynchronous invocation of a function means that it will be scheduled as a new HPX-thread (either locally or on another locality). The call to `async` will return almost immediately providing a new `future` instance which represents the result of the function execution. Asynchronous function execution is the fundamental way of orchestrating asynchronous parallelism in HPX.
- *Fire&Forget function execution:* this is similar to asynchronous execution except that the caller has no means of synchronizing with the result of the operation. The call to `apply` simply schedules a local (or remote) HPX-thread which runs to completion at its own pace. Any result returned from that function (or any exception thrown) is being ignored. This leads to less communication by not having to notify the caller.

All three forms of action execution have a homogeneous syntax regardless whether the target of the operation is local or remote (see light gray region in Table 1). The GID used for the operation determines on which locality the encapsulated function is executed. HPX will directly schedule a local HPX-thread if the GID refers to a local object. Otherwise a parcel will be created and dispatched to the locality hosting the referenced object. HPX uses its implementation of AGAS to determine the locality of the destination. The received parcel will schedule an HPX-thread on the remote locality. In HPX, the operations of creating a local thread and sending a parcel which causes a remote thread to be created and scheduled are semantically fully equivalent operations.

3.4 Achieving Programmability

The exposure of an API based on the C++-Standard ultimately positions HPX at the level of a programming interface for higher-level application frameworks. We anticipate that this API will not be used directly by domain scientists, rather it will be utilized to create higher-level domain specific application frameworks which simplify the creation of the actual applications. Examples for such frameworks are LibGeoDecomp [51], MetaScale NT^2 [52], or Boost.Odeint [53]. All these libraries support using HPX as a back end.

The C++ Standards committee puts great effort into creating APIs which are perfect for designing and writing libraries. By utilizing this API, HPX is able to leverage the powerful asynchronous mechanisms of the standard which allow the development of composable interfaces that efficiently expose parallelism. In addition, HPX's governing principles, design, and high performance implementation perfectly allow for nested parallelization by shifting the burden imposed by writing parallel codes from "Joe the Scientist" to our expert "Hero Programmer".

4. HIGH PERFORMANCE IMPLEMENTATION FOR SYSTEMS OF ANY SCALE

The availability of a standardized API (see Sec. 3.3) based on a solid theoretic foundation (see Sec. 3.1) forms the baseline of having a competitive solution solving scalability problems at Exascale. This section will demonstrate the ability of HPX to deliver adequate application performance. We further outline the techniques used to enable improved scalability beyond conventional programming techniques (such as based on MPI and OpenMP). We show performance portability by providing numbers obtained from scaling experiments on a single node, a single Intel Xeon Phi co-processor, and large scale distributed runs. All results were obtained from the same source code without special treatments.

We selected a small set of available HPX applications which were implemented with our constraint based parallelization technique (futurization) in mind. While they don't exploit every feature of HPX (they don't use adaptive locality control through migration), they highly benefit from the ability to use fine grained parallelism which is made possible by the HPX API. The presented applications are an Nbody-Code implemented with LibGeoDecomp [28] and the miniGhost [54] benchmark from the Mantevo Benchmark Suite [55]. Additionally, we compare the scheduling efficiencies of other task based library implementations (Qthreads [2] and Intel TBB [6]), using the Homogeneous-Timed-Task-Spawn benchmark.

	Intel Xeon E5	Intel Xeon Phi
Clock Frequency	2.7 (3.5 Turbo) GHz	1.1 GHz
Number of Cores	16 (2x8)	61
SMT	2-way (deactivated)	4-way
NUMA Domains	2	1
RAM	32 GB	8 GB
SIMD	AVX (256 bit)	MIC (512 bit)
GFLOPS	691.2 (896.0 Turbo)	2147.2
Microarchitecture	Sandy Bridge	Knights Corner

Table 2: Overview of the processors built into one compute node of the Stampede supercomputer. GLFOPS are presented in single precision.

The results presented here were obtained on TACC Stampede Supercomputer [56]. It consist of a total of 6400 nodes, each with two Intel Xeon E5 processors and one Intel Xeon Phi coprocessor (see Table 2). The compute nodes are interconnected with Mellanox FDR InfiniBand technology (56 Gb/s) in a 2-level fat-tree

`R f(p...)`	Synchronous Execution (returns `R`)	Asynchronous Execution (returns `future<R>`)	Fire & Forget Execution (returns **`void`**)
Functions (direct invocation)	`f(p...)` C++	`async(f, p...)`	`apply(f, p...)`
Functions (lazy invocation)	`bind(f, p...)(...)`	`async(bind(f, p...), ...)` C++ Standard Library	`apply(bind(f, p...), ...)`
Actions (direct invocation)	`HPX_ACTION(f, action)` `a(id, p...)`	`HPX_ACTION(f, action)` `async(a, id, p...)`	`HPX_ACTION(f, action)` `apply(a, id, p...)`
Actions (lazy invocation)	`HPX_ACTION(f, action)` `bind(a, id, p...)` `(...)`	`HPX_ACTION(f, action)` `async(bind(a, id, p...),` `...)`	`HPX_ACTION(f, action)` `apply(bind(a, id, p...),` `...)` HPX

Table 1: Overview of the main API exposed by HPX. This table shows the function invocation syntax as defined by the C++ language (dark gray), the additional invocation syntax as provided through C++ Standard Library features (medium gray), and the extensions added by HPX (light gray). Where: `f`: function to invoke; `p...`: (optional) arguments; `R`: return type of `f`; `action`: action type defined by `HPX_ACTION()` encapsulating `f`; `a`: an instance of the type `action`; `id`: the global address the action is applied to.

topology. The complete system is a 10 PFLOPS cluster.

The Homogeneous-Timed-Task-Spawn results were collected on a node from LSU's Hermione cluster. The test machine had two Intel Xeon E5 v2 Ivybridge processors (each with 10 cores clocked at 2.5 GHz), and 128GB of DDR3 memory.

4.1 Thread Scheduling capabilities

The HPX thread scheduling subsystem is at the heart of HPX and is designed to efficiently handle hundreds of millions of tasks of any duration. We present results from the Homogeneous-Timed-Task-Spawn (HTTS) benchmark executing no-op tasks. We compare the scheduler throughput and the thread overheads of the HPX threading subsystem with two of the most widely used cross-platform, open source task libraries - Intel TBB [6] and Qthreads [2]. The used benchmark is embarrassingly parallel, so there is no communication between workers. All scheduler queues have sufficient work available, so work stealing does not occur. Note, however, that we still pay a synchronization cost for hardware atomics used inside the scheduler on cache coherent systems. Because of these properties, we claim that the per-thread overheads measured provides a reasonable lower-bound estimate for HPX applications with similar queue lengths and task payloads.

Fig. 2 shows the scheduling throughput of HPX, Qthreads, and TBB for the HTTS benchmark on our test system. We note that TBB and Qthreads both experience a spike in per-task overheads at the socket boundary. HPX, however, maintains strongly linear behavior. In Fig. 3 we present estimated per-task overheads using HTTS results. For comparison, we also present results of the same benchmark from Qthreads (version 9f15ec9) and Intel TBB (version 4.2). Both HPX and Qthreads are able to maintain sub-microsecond overheads on all 20 cores of the test system. TBB shows the greatest growth in overheads as the number of cores is increased.

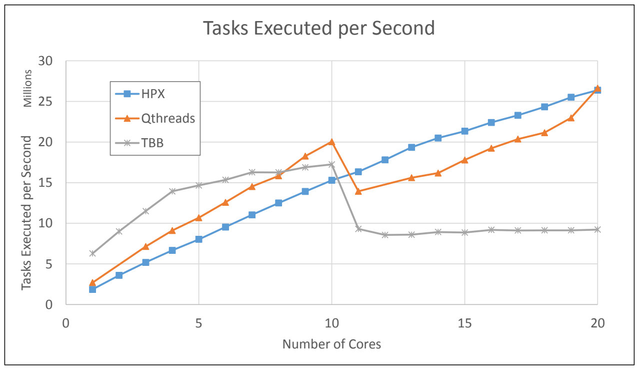

Figure 2: Scheduling throughput of HPX, Qthreads and TBB on a 2-socket Intel Ivybridge system, measured with the Homogeneous-Timed-Task-Spawn benchmark. No artificial payload was used. Each task was a no-op and returned immediately to the scheduler, simulating extremely fine grain sizes. 2.5 million tasks per core were used; on 20 cores, all three frameworks executed 50 million tasks in total.

Our results show that HPX per-task overheads are very small. We performed trials of HTTS with 800 million tasks on 20 cores, also with no-op task payloads. Those measurements show an estimated overhead per task of 760 nanoseconds. This result agrees with the estimated per-task overhead for the 50 million task trials: 758 nanoseconds (as shown in Fig. 3).

4.2 LibGeoDecomp based Nbody-Code

LibGeoDecomp [51] is an auto-parallelization Library for Geometric Decomposition codes. The user only needs to supply a simulation model together with various options specifying how the simulation domain is structured. The library then handles the spatial and temporal loops as well as the data storage. In our experiment we show how using HPX's unified programming model for implement-

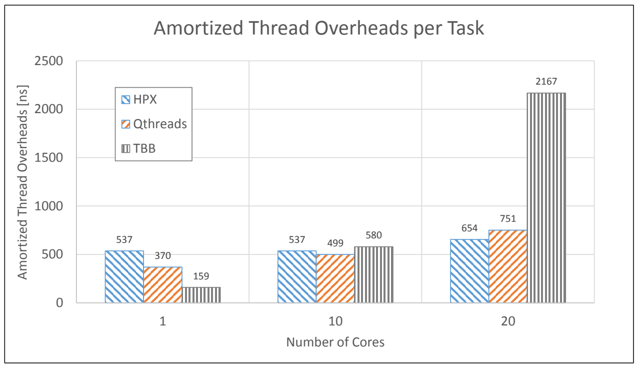

Figure 3: Estimated lower-bound for per-task overheads in HPX, Qthreads and TBB on a 2-socket Intel Ivybridge system. These estimates are based upon results from the embarrassingly parallel Homogeneous-Timed-Task-Spawn, with 2.5 million tasks per core and a no-op payload. The presented data is based on average execution times from 8 trials of the entire parameter space. Our observations have found that these overheads do not vary greatly as task duration is changed. These lower-bound approximations can be used to determine the smallest feasible magnitude of granularity on a given platform.

ing a three dimensional N-Body simulation exhibits perfect scaling at a single node level (16 cores) by reaching 98% peak performance of a single node of the Stampede supercomputer [28]. We reevaluated the performance presented in one of our previous papers and reran the benchmarks which show significant improvements in various layers of HPX.

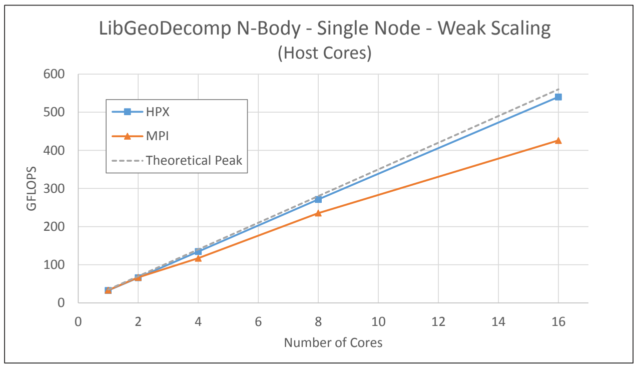

Figure 4: Weak scaling results for the HPX and MPI N-Body codes collected on a single SMP node while varying the number of cores used from 1 to 16. The figure shows the performance in GFLOPS. The results show that the HPX backend is able to achieve an almost perfect weak scaling. The HPX backend is able to sustain a performance of ∼540 GLFOPS.

The results on a single compute node haven't changed from our previously achieved performance (see Fig. 4). We are able to reach ∼98% peak performance with HPX. The performance on a single Xeon Phi (see Fig. 5), as on the host processor, remains at 89% peak performance demonstrating that our runtime system is able to efficiently utilize many-core architectures like the Xeon Phi co-processor.

The main improvements in our implementation has been gone into

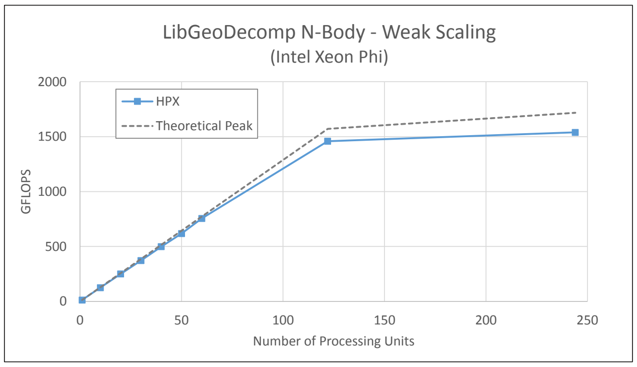

Figure 5: Weak scaling results for the HPX N-Body code collected on a single Intel Xeon Phi co-processor while varying both the number of processing units used. The overall performance does not increase much more after two threads per core are used. HPX reached a sustained peak performance of 1504.7 GFLOPS using all 244 processing units which is equivalent to a parallel efficiency of ∼89%.

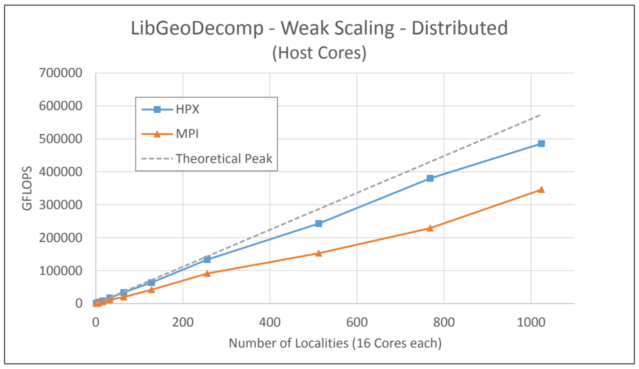

Figure 6: Weak scaling performance results for the HPX and MPI N-Body codes collected for runs on the host's cores only (16 cores per node) while increasing the number of nodes from 1 to 1024 (16 to 16384 cores). On 1024 nodes, the HPX code outperforms the equivalent MPI code by a factor of 1.4 and reaches a performance of ∼0.48 PFLOPS.

optimizing the communication layer. Fig. 6 shows the results of those improvements. We were able to improve our previous results by a factor of ∼1.35 which has the effect that the HPX backend is able to outperform the MPI implementation by a factor of 1.4 while performing at a parallel efficiency of ∼87% at 1024 nodes running on a total of 16384 CPU Cores.

LibGeoDecomp is one example of an application framework which hides the complexity of parallel programming from the end users. By exploiting parallelism with the mechanisms provided by HPX, scientific simulations written within LibGeoDecomp can be used to fully unleash the power of parallel computers by showing performance not possible with conventional practice.

4.3 HPX port of the miniGhost Benchmark

Our second application benchmark is a port of the miniGhost proxy application delivered in the Mantevo benchmark suites. This miniapp represents algorithms for solving partial differential equations with the help of a finite difference stencil. The reference implementation is written in the Bulk Synchronous Parallel Programming Model (BSP).

We present early results of our take to convert the miniGhost applications BSP formulation to the constraint-based dataflow style programming technique made available by the HPX programming model (see Sec. 3.1). It combines the techniques developed in [57] and [28] to provide an implementation which removes the bulk synchronous nature of algorithms needing halo-exchanges. It also demonstrates a way to remove the global barrier imposed by the global reduction operation needed for some of the variables by fully overlapping computation with communication. Any particular computation is scheduled whenever all preconditions for its execution are met (all data dependencies are fulfilled). HPX uses `future` objects to implicitly express those data dependencies and the composition of those `future` objects (through dataflow objects or other LCOs) ensures that only the required minimal set of data dependencies is considered.

For benchmarking the miniGhost application we chose a number of 40 variables with each variable being of the dimension $200 \times 200 \times 200$. The number of time steps was set to 20 with reducing 10% of the variables in each time step. We used the 2D5PT stencil throughout our runs, meaning that only a plane in the three dimensional volume is considered while updating a element. The characterestics of the benchmark do not change when choosing a different stencil. The number of elements in each variable was kept constant per process for the weak scaling experiment.

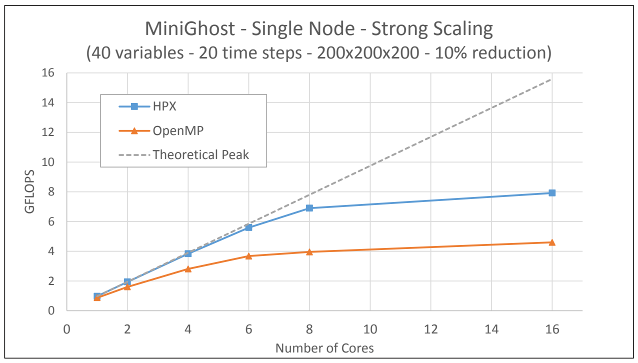

Figure 7: Strong scaling results for the HPX and OpenMP implementation of the miniGhost application. These were obtained from a single compute node of the Stampede Supercomputer. The results show that the HPX implementation shows near perfect strong scaling using up to 8 cores.

Fig. 7 shows the scaling behavior of both, the reference OpenMP implementation as well as the HPX port of the Mantevo miniGhost application. HPX easily outscales and outperforms the reference implementation. This is the result of the effective futurization of the algorithm control flow made possible by the HPX API. At 8 cores, the resulting speedup is 7.1 with a parallel efficiency of 88%. At 16 cores, the speedup is decreased due to increasing NUMA related effects. For this reason, we chose 2 processes per node for the distributed runs.

The results of running miniGhost in distributed is shown in Fig. 8. We see that the constraint based parallelization is not only advantageous for a single node but also, due to the unified semantics as described in Sec. 3.3, for larger scale distributed experiments. The

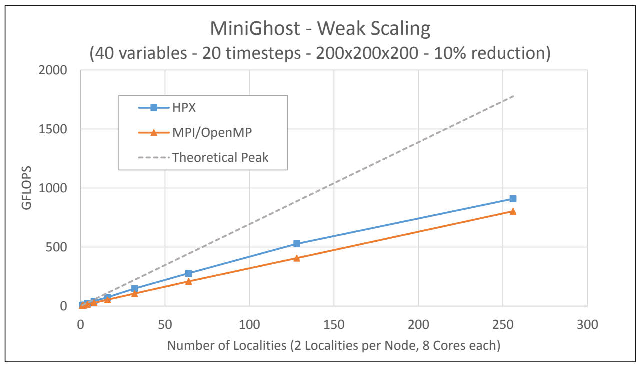

Figure 8: Weak scaling performance results for the HPX and MPI+OpenMP implementation of the miniGhost application. The experiment was conducted on up to 128 compute nodes of the Stampede Supercomputer. We are able to show that the HPX application outperforms the reference implementation by a factor of 1.13.

main reasons for the speedup can be contributed to the efficient overlapping of communication and computation due to fine grained constraints. Those fine grained constraints also benefit from the ability to be able to write a completely asynchronous reduction needed for the global sum of 10% of the used variables. It is worthy to note that the 128 node run used a total of 45.5 million HPX threads which translates to around 22 thousand threads executed per core. At a runtime of 9 seconds, this results in 5 million threads executed per seconds.

5. CONCLUSION

Exascale machines will require a paradigm shift in the way parallel programs are written. The currently dominant MPI+X effort proposes that these programming challenges can be overcome by taking a two-step approach: Use MPI for communication and synchronization of distinct processes and"X" for intra-process parallelism. However, due to the rise of many-core machines and faster internode networks, we postulate that this way of thinking about parallelism is not enough. While conventional techniques work well on current scale and most workloads, they exhibit a critical portion of serial code through explicit and implicit global barriers. With the help of unified semantics of remote and local operations in combination of a capable lightweight threading system, we are able to remove almost all serial portions of code and to perfectly overlap computation with communication without complicating the API.

HPX has been designed as a possible answer to at least part of the Exascale challenges referenced in Sec. 1, and so far the results we have seen are very promising. We have shown that HPX supports *application scalability* beyond what is possible using conventional programming models today. This has been achieved based on an API which enables high *programmability* by exposing asynchronous parallelism in a homogeneous way. The results we present demonstrate a high degree of *performance portability* based on a very portable implementation which is conforming to the latest C++ Standards. The achieved high parallel efficiency numbers support our claim of HPX being *energy efficient* through very high resource utilization. Overall, we show that the HPX runtime system is unique as:

- It is based on a solid theoretical foundation, the ParalleX execution model;

- It implements a programming model which is aligned with the theoretical model and which exposes an API compatible and conforming to the widely accepted C++11 Standard;
- It represents an existing implementation which has shown to outperform equivalent applications which are based on conventional programming APIs

Our work with HPX, however is far from complete. We hope to develop new dynamic and runtime aware load balancing mechanisms so that we will be able to handle changes in the application execution in real time. We intend to continue work on integrating with existing technologies providing a smooth migration path for application developers. In this way, we can combine the need for new programming models with a way to use existing code transparently within HPX. Finally, we plan to work on ways to tackle problems related to system resilience.

Despite this work yet to be done, our results reveal that, today, HPX can be efficiently used for homogeneous large scale applications as well as in heterogeneous environments. We believe that these finding give credence to the design of our execution model and show that HPX is competitive player in the world of HPC.

6. ACKNOWLEDGMENTS

We would like to acknowledge the NSF, DoE, the Center for Computation and Technology at Louisiana State University, and the Department of Computer Science 3 at the University of Erlangen Nuremberg who fund our work. We would also like to thank LSU HPC, LONI, XSEDE, ALCF, and the Gauss Center for Supercomputing. for granting us allocations for their compute resources.

7. REFERENCES

[1] "X-Stack: Programming Challenges, Runtime Systems, and Tools, DoE-FOA-0000619," 2012, http://science.energy.gov/ /media/grants/pdf/foas/2012/SC_FOA_0000619.pdf.

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

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

[4] M. Anderson, M. Brodowicz, H. Kaiser, B. Adelstein-Lelbach, and T. L. Sterling, "Neutron star evolutions using tabulated equations of state with a new execution model," *CoRR*, vol. abs/1205.5055, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/ corr1205.html#abs-1205-5055

[5] C. Dekate, H. Kaiser, M. Anderson, B. Adelstein-Lelbach, and T. Sterling, "N-Body SVN repository," 2011, available under a BSD-style open source license. Contact hpx-users@stellar.cct.lsu.edu for repository access. [Online]. Available: https: //svn.cct.lsu.edu/repos/projects/parallex/trunk/history/nbody

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

[7] Microsoft, "Microsoft Parallel Pattern Library," 2010, http://msdn.microsoft.com/en-us/library/dd492418.aspx.

[8] "StarPU - A Unified Runtime System for Heterogeneous Multicore Architectures," 2013, http://runtime.bordeaux.inria.fr/StarPU/.

[9] "Intel(R) Cilk(tm) Plus," 2014, http://software.intel.com/en-us/intel-cilk-plus.

[10] "OpenMP Specifications," 2013, http://openmp.org/wp/openmp-specifications/.

[11] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," *International Journal of High Performance Computing Applications*, vol. 21, pp. 291–312, 2007.

[12] "Intel SPMD Program Compiler," 2011-2012, http://ispc.github.io/.

[13] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non- uniform cluster computing," *SIGPLAN Not.*, vol. 40, pp. 519–538, October 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094852

[14] The C++ Standards Committee, "ISO/IEC 14882:2011, Standard for Programming Language C++," , Tech. Rep., 2011, http://www.open-std.org/jtc1/sc22/wg21.

[15] The C++ Standards Committee , "N3797: Working Draft, Standard for Programming Language C++," Tech. Rep., 2013, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf.

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

[17] "The OmpSs Programming Model," 2013, https://pm.bsc.es/ompss.

[18] "OpenACC - Directives for Accelerators," 2013, http://www.openacc-standard.org/.

[19] "C++ AMP (C++ Accelerated Massive Parallelism)," 2013, http://msdn.microsoft.com/en-us/library/hh265137.aspx.

[20] "CUDA," 2013, http://www.nvidia.com/object/cuda_home_new.html.

[21] "OpenCL - The open standard for parallel programming of heterogeneous systems," 2013, https://www.khronos.org/opencl/.

[22] UPC Consortium, "UPC Language Specifications, v1.2," Lawrence Berkeley National Lab, Tech Report LBNL-59208, 2005. [Online]. Available: http://www.gwu.edu/\~{}upc/publications/LBNL-59208.pdf

[23] Oracle, "Project Frotress," 2011, https://projectfortress.java.net/.

[24] PGAS, "PGAS - Partitioned Global Address Space," 2011, http://www.pgas.org.

[25] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with mpi." in *IPDPS*. IEEE Computer Society, 2013, pp. 712–725. [Online]. Available: http://dblp.uni-trier.de/db/conf/ipps/ipdps2013.html# ChatterjeeTBCCGSY13

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

[27] H. Kaiser, M. Brodowicz, and T. 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.

[28] T. Heller, H. Kaiser, A. Schäfer, and D. Fey, "Using HPX and LibGeoDecomp for Scaling HPC Applications on Heterogeneous Supercomputers," in *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '13. New York, NY, USA: ACM, 2013, pp. 1:1–1:8. [Online]. Available: http://doi.acm.org/10.1145/2530268.2530269

[29] C. Dekate, M. Anderson, M. Brodowicz, H. Kaiser, B. Adelstein-Lelbach, and T. L. Sterling, "Improving the scalability of parallel N-body applications with an event driven constraint based execution model," *The International Journal of High Performance Computing Applications*, vol. abs/1109.5190, 2012, http://arxiv.org/abs/1109.5190.

[30] A. Tabbal, M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling, "Preliminary design examination of the ParalleX system from a software and hardware perspective," *SIGMETRICS Performance Evaluation Review*, vol. 38, p. 4, Mar 2011.

[31] M. Anderson, M. Brodowicz, H. Kaiser, and T. L. Sterling, "An application driven analysis of the ParalleX execution model," *CoRR*, vol. abs/1109.5201, 2011, http://arxiv.org/abs/1109.5201.

[32] "InifiniBand Trade Association," 2014, http://www.infinibandta.org/.

[33] A. Kopser and D. Vollrath, "Overview of the Next Generation Cray XMT," in *Cray User Group Proceedings*, 2011, pp. 1–10.

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

[35] L. Dagum and R. Menon, "OpenMP: An Industry- Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[36] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[37] G. Papadopoulos and D. Culler, "Monsoon: An Explicit Token-Store Architecture," in *17th International Symposium on Computer Architecture*, ser. ACM SIGARCH Computer Architecture News, no. 18(2). Seattle, Washington, May 28–31: ACM Digital Library, June 1990, pp. 82–91.

[38] J. B. Dennis, "First version of a data flow procedure language," in *Symposium on Programming*, 1974, pp. 362–376.

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

[40] "CppLINDA: C++ LINDA implementation," 2013, http://sourceforge.net/projects/cpplinda/.

[41] D. W. Wall, "Messages as active agents," in *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '82. New York, NY, USA: ACM, 1982, pp. 34–39. [Online]. Available: http://doi.acm.org/10.1145/582153.582157

[42] K. Yelick, V. Sarkar, J. Demmel, M. Erez, and D. Quinlan, "DEGAS: Dynamic Exascale Global Address Space," 2013, http://crd.lbl.gov/assets/Uploads/FTG/Projects/DEGAS/RetreatSummer13/DEGAS-Overview-Yelick-Retreat13.pdf.

[43] H. C. Baker and C. Hewitt, "The incremental garbage collection of processes," in *SIGART Bull.* New York, NY, USA: ACM, August 1977, pp. 55–59. [Online]. Available: http://doi.acm.org/10.1145/872736.806932

[44] D. P. Friedman and D. S. Wise, "CONS Should Not Evaluate its Arguments," in *ICALP*, 1976, pp. 257–284.

[45] R. H. Halstead, Jr., "MULTILISP: A language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 501–538, October 1985. [Online]. Available: http://doi.acm.org/10.1145/4472.4478

[46] J. B. Dennis and D. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," in *25 Years ISCA: Retrospectives and Reprints*, 1998, pp. 125–131.

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

[48] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with "readers" and "writers"," *Commun. ACM*, vol. 14, no. 10, pp. 667–668, 1971.

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

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

[51] A. Schï£¡fer and D. Fey, "LibGeoDecomp: A Grid-Enabled Library for Geometric Decomposition Codes," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer, 2008, pp. 285–294.

[52] MetaScale, "NT^2 – High-performance MATLAB-inspired C++ framework," 2014, http://www.metascale.org/products/nt2.

[53] Odeint, "Boost.Odeint – a C++ Library for Solving ODEs," 2014, http://www.odeint.com.

[54] R. F. Barrett, C. T. Vaughan, and M. A. Heroux, "Minighost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing," *Sandia National Laboratories, Tech. Rep. SAND*, vol. 5294832, 2011.

[55] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 2009.

[56] Texas Advanced Computing Center - Stampede. Http://www.tacc.utexas.edu/resources/hpc/stampede. [Online]. Available: http://www.tacc.utexas.edu/resources/hpc/stampede

[57] T. Heller, H. Kaiser, and K. Iglberger, "Application of the ParalleX Execution Model to Stencil-based Problems," in *Proceedings of the International Supercomputing Conference ISC'12, Hamburg, Germany*, 2012. [Online]. Available: http://stellar.cct.lsu.edu/pubs/isc2012.pdf