Thread Management in the HPX Runtime: A Performance and Overhead Characterization

Bryce Adelstein-Lelbach*, Patricia Grubel†, Thomas Heller‡, Hartmut Kaiser*, Jeanine Cook§
*Center for Computation and Technology, Louisiana State University
†Klipsch School of Electrical and Computer Engineering, New Mexico State University
‡Chair of Computer Science 3, Computer Architectures, Friedrich Alexander University
§Sandia National Laboratories
blelbach@cct.lsu.edu, pagrubel@nmsu.edu, thomas.heller@cs.fau.de, hkaiser@cct.lsu.edu, jeacook@sandia.gov

I. Introduction

High Performance ParalleX (HPX) is a C++ parallel and distributed runtime system for conventional architectures that implements the ParalleX [1], [2], execution model and aims to improve the performance of scaling impaired applications by employing fine-grained threading and asynchronous communication, replacing the traditional Communicating Sequential Processes (CSP). Fine-grained threading provides parallel applications the flexibility to generate large numbers of short lived user-level tasks on modern multi-core systems. The HPX runtime system gives applications the ability to use fine-grained threading to increase the total amount of concurrent operations, while making efficient use of parallelism by eliminating explicit and implicit global barriers. While fine-grained threading can improve parallelism, it causes overheads due to creation, contention due to context switching, and increased memory footprints of suspended threads waiting on synchronization and pending threads waiting on resources. Measurements and thorough analysis of overheads and sources of contention help us determine bounds of granularity for improved scaling of parallel applications. This knowledge combined with the capabilities of the HPX runtime system pave the way to measure overheads at runtime to tune performance and scalability. This paper explains the thread scheduling and queuing mechanisms of HPX, presents detailed analysis of the measured overheads, and illustrates resulting granularity bounds for good scaling performance. The quantification of overheads gives substantial information resulting in determination of granularity and metrics, which can be used in future work to create dynamic adaptive scheduling.

II. The HPX Runtime System

HPX is a general purpose parallel runtime system for applications of any scale. It exposes a homogeneous programming model which unifies the execution of remote and local operations. The runtime system has been developed for conventional architectures. Currently supported are SMP nodes, large Non Uniform Memory Access (NUMA) machines and heterogeneous systems such as clusters equipped with Xeon Phi accelarators. Strict adherence to Standard C++ [3] and the utilization of the Boost C++ Libraries [4] makes HPX both portable and highly optimized. The source code is published under the Boost Software License making it accessible to everyone as Open Source Software. It is modular, feature-complete and designed for best possible performance. HPX's design focuses on overcoming conventional limitations such as (implicit and explicit) global barriers, poor latency hiding, static-only resource allocation, and lack of support for medium- to fine-grain parallelism. The framework consists of four primary modules: The HPX Threading System, Local Control Objects (LCOs), the Active Global Address Space (AGAS) and the Parcel Transport Layer.

The HPX Threading System: The HPX Threading System's core is formed by the thread-manager which is responsible for creation, scheduling, execution and deletion of HPX-Threads. HPX-Threads are very lightweight user-level threads which are scheduled cooperatively and non-preemptively. This implements a *M:N* or hybrid threading model, which is essential to enable fine-grained parallelism. The focus of this paper will be on the Threading System, as such, a detailed description can be found in Section III.

Local Control Objects (LCOs): Local Control Objects (LCOs) are used to organize control flow through event-driven HPX-thread creation, supension or reactivation. Every object that creates or re-activates an HPX-thread exposes the functionality of an LCO. As such they provide an efficient abstraction to manage concurrency.

The most prominent examples are:
1. *Futures* [5], [6], [7] are objects representing a results of which is initially not known yet because the computation of the value has not yet completed. A **future** synchronizes access to this value by either suspending the requesting thread until the value is available or by allowing the requesting thread to continue computation unencumbered if the value is already available.
2. *Dataflows* [8], [9], [10] provide a mechanism that manages asynchronous operation and enables the elimination of global barriers in most cases. The dataflow LCO construct is event-driven and acquires result values and updates internal state until one or more precedent constraints are satisfied. It subsequently initiates further program action dependent on these condition(s).

The Active Global Address Space (AGAS): AGAS is currently implemented as a set of distributed services providing a 128-bit global address space spanning all localities. Each locality serves as a partition in the global address space. AGAS

provides two naming layers: *1.* The primary namespace maps 128-bit globally unique identifiers (GIDs) to a tuple of meta-data which is used to locate an object on the current locality. *2.* A higher-level mechanism which maps hierarchical symbolic names to GIDs. Unlike PGAS [11] systems like X10 [12], Chapel [13], or UPC [14], AGAS exposes a dynamic, adaptive address space which evolves over the lifetime of an HPX application. In addition, objects in AGAS can be migrated, which leaves the GID the same and merely updates the internal AGAS mapping. This allows decoupling of objects with locality information.

The Parcel Transport Layer: Parcels are an extended form of active messages [15] that are used for inter-locality communication. Parcels form the mechanisms to implement remote procedure calls (actions). They contain a GID, the action to be invoked on the object represented by the GID and the arguments needed to call that action. A parcel port is an implementation of a specific network communication layer. Whenever a parcel is received it will be passed to the parcel handler which will eventually turn it into an HPX-Thread, which in turn will execute the specified action.

III. Threading in HPX

A. Major Design Principles

HPX's threading system utilizes the M:N model (also known as hybrid-threading). Hybrid threading implementations use a pool of kernel threads (N) to execute library threads (M). Context switching between library threads is typically much quicker than kernel-level context switches due to not requiring system calls which usually lead to expensive context switches. This allows library threads to synchronize and communicate with lower overheads than kernel threads can achieve. In many implementations, including HPX, the kernel threads are associated directly with specific processing units and are live throughout the entire execution of a program. On the other hand, library threads are ephemeral. They can be quickly created and destroyed, as they do not have to manage as much state as kernel threads. They may outnumber the kernel threads by many orders of magnitude without significant performance penalties. In HPX, kernel threads are referred to as **worker-threads**. HPX's library threads are called **HPX-threads**.

One of the core tenants of HPX's threading model is to be greedy. Wherever possible and sensible, the HPX threading system makes locally optimal choices. Localizing control and decision making reduces the need for synchronization and communication between different worker-threads. This localization is essential for multi-threaded cache-coherent and NUMA systems.

HPX operates under the assumption that it is probable that HPX applications either have ample work to perform, or will have ample work at some point in the near future. Since HPX is designed to facilitate the overlap of communication with computation, this assumption tends to hold true. So, the HPX worker-threads prefer to avoid yielding control of their processing unit whenever feasible.

The class of highly dynamic applications that HPX targets may have unpredictable load imbalances which need to be rapidly corrected at runtime. The HPX threading system utilizes a work-queue model which enables the use of work-stealing for resolving these load imbalances. When load imbalances occur, worker-threads, depleted of sufficient work, immediately begin looking for work to steal from their neighbors. The level of "aggression" of work-stealing algorithms may be constrained by the runtime to limit the contention overheads associated with work-stealing.

HPX-threads are cooperatively scheduled. The HPX scheduler will never time-slice or involuntarily interrupt an HPX-thread. However, HPX-threads may voluntarily choose to return control to the scheduler. HPX applications are often the most qualified decision-makers, because they have access to application-specific information that may influence decision-making.

B. HPX-Threads

HPX-threads are instruction streams that have an **execution context** and a **thread state** [1]. An HPX-thread context contains the information necessary to execute the HPX-thread. Each HPX-thread also has an associated state, which defines the invariants of the HPX-thread. Additionally, HPX-threads are first-class objects, meaning that they have a unique global name (a GID). So, HPX-threads are **executable** entities that are **globally addressable**.

In any HPX application, there are two types of HPX-threads. The first, **application HPX-threads**, are the most apparent. Application (or user) HPX-threads are HPX-threads that execute application code at some point during their lifetime. The second type of HPX-threads are **system HPX-threads**. System HPX-threads may be created directly by the runtime, or indirectly by the application.

An HPX-thread that sets the value of a future LCO and then executes a continuation attached to the future by the application is an application HPX-thread. An HPX-thread that only changes the state of another HPX-thread is a system HPX-thread. The distinction between application HPX-threads and system HPX-threads is significant from a performance analysis viewpoint. System HPX-threads can be treated as pure overhead (in the vast majority of cases). The overheads of application HPX-threads must be specifically identified.

1) HPX-Thread Attributes: In addition to context and state, HPX-threads have other characteristics. Collectively, we call these HPX-thread attributes. In this section, we will describe the most important of these attributes.

Each HPX-thread has an associated **payload**. A payload is the starting point of execution for an HPX-thread. In the simplest form, a payload is simply a reference to a function to execute. A payload may also include arguments passed to the function. If the payload is a method on a globally named HPX object, then the HPX-thread is said to target that object. The **payload duration** of an HPX-thread begins when payload execution begins, and ends when the payload function is completely finished executing. Amortized payload duration

is an important measure of application granularity that we will utilize in this paper.

The HPX-thread context is the data required to start or resume execution of the payload. HPX-thread contexts may be hardware, operating system or operation specific. Currently, stack-based contexts are used in almost all circumstances in HPX.

Stack-based contexts use a call stack and a stack pointer to represent the execution state. When the HPX-thread is created, the runtime places a function call into the stack (a trampoline) which will invoke the payload. When the scheduler needs to execute the HPX-thread (either to start execution or to resume it), it performs a **context switch**. The scheduler first pushes its register set to the scheduler stack and pops the register set from the HPX-thread stack. Next, the scheduler stores its own stack pointer, and loads the HPX-thread stack pointer. Finally, the scheduler performs an unconditional jump to the HPX-thread's program counter. If the HPX-thread needs to return control to the scheduler, a context switch in the opposite direction is performed.

A priority may be associated with an HPX-thread. In HPX, priorities may only be set by the entity creating an HPX-thread, and the priority of an HPX-thread is immutable. The HPX scheduler has no recourse for upgrading or downgrading the priority of HPX-threads.

2) HPX-Thread States: HPX-threads are finite state machines. Each state imposes different conditions on an HPX-thread which neither the HPX scheduler nor application code is allowed to violate. The HPX scheduler's primary role is to transition HPX-threads from one state to another, while enforcing the invariants of each HPX-thread state. There are four HPX-thread states:

- Pending - An HPX-thread that is ready for execution.
- Active - An HPX-thread that is executing.
- Suspended - An HPX-thread that is not ready for execution yet.
- Terminated - An HPX-thread that has finished execution.

An HPX-thread may voluntarily choose to return control to the HPX-thread scheduler. When this occurs, the HPX-thread specifies a requested new state, which is called the **exiting state**. HPX-threads are allowed to place themselves into any state (except, of course, active).

When an HPX-thread needs to wait for synchronization or communication to continue execution, it can suspend itself and wait to be woken up by the party that will provide the resources or data necessary for the HPX-thread to continue execution. Recall that any first-class object which, when triggered, resumes or creates an HPX-thread is an LCO. When an HPX-thread is suspended, it acts as an LCO, since setting its state to pending (i.e. triggering it) will resume its execution.

HPX-threads can also postpone execution by transitioning from active to pending. Such a state transition does not require a second actor to resume execution, which allows for very short suspensions. An HPX-thread that is using a live-waiting synchronization primitive such as a spinlock may make use of active to pending transitions to reduce pressure on a highly contended resource.

Finally, when an HPX-thread has finished executing its payload function, it returns to the scheduler with an exiting state of terminated. While execution of an HPX-thread may never be resumed, a terminated HPX-thread still has a global name and therefore is still a first class object.

In addition to voluntary state changes initiated by an HPX-thread, there are three mechanisms by which external entities can cause HPX-threads to undergo state transitions. The first we have already discussed. Any entity that knows the global name of a suspended HPX-thread can trigger it by setting its state to pending, which will resume the HPX-thread. Additionally, the HPX scheduler is the only construct that can transition pending HPX-threads to active. Last but not least, HPX-Threads have defined interruption points. An entity holding a reference to a thread, may request the thread to be interrupted.

3) Ephemerality of HPX-threads: HPX-threads are designed to be short-lived relative to overall execution time and numerous relative to the number of available processing units. The use of many, short-lived HPX-threads can be essential for parallel applications that may suffer from frequent unavoidable communication and/or load imbalances. Finer HPX-thread granularity localizes synchronization and execution control. This has the potential to allow computation to proceed further before waiting for required synchronization or communication. Additionally, dynamic load imbalances can be better redistributed if HPX-thread payloads are small. Finer granularity, of course, is not without overheads because the scheduler must manage a greater number of HPX-threads.

Parallel applications may not have complete control over the number or duration of HPX-threads that they use. However, every application has some mechanism of controlling how work is divided and therefore has some degree of control over the number of HPX-threads that it uses and the average payload duration - the grain size - of those HPX-threads. We call this **granularity control**.

C. HPX-Thread Schedulers

In the previous section, we introduced the important aspects of HPX-threads. Now we will discuss the constructs that are responsible for implementing HPX-threads - **HPX-thread schedulers**. An HPX-thread scheduler is composed of three components:

- Core Scheduling - Generic code shared by all HPX-thread schedulers.
- Scheduling Policy - A set of routines and data structure that implement the creation, distribution, execution and destruction of HPX-threads. These routines and data structures plug into the core scheduler.
- Queuing Policy - A set of routines and data structures that implements a queuing mechanism which is used by the scheduling policy.

HPX-thread schedulers are modular and easily extendable by implementing a defined interface. Any scheduling policy

can be combined with any queuing policy. Application developers can either select from a collection of scheduling and queuing policies, or create policies suited to their specific needs.

All HPX-thread schedulers make use of the **dual-queuing model**. This model requires that every HPX-thread move through two queues before becoming active. When an HPX-thread scheduler is instructed to create an HPX-thread, it constructs an HPX-thread description and places it into a queue. An HPX-thread description is an object that contains all of the attributes of an HPX-thread except for a context. An HPX-thread description represents a staged HPX-thread, and so we call this first queue a staged queue. Eventually, the HPX-thread scheduler will remove the HPX-thread description from the **staged queue**. At this point, the HPX-thread description is converted into a bona fida HPX-thread object, which has a context. This conversion is the state transition from staged to pending. The pending HPX-thread is now placed into a **pending queue**.

The dual-queuing model allows for fast, asynchronous creation of HPX-thread. HPX-thread descriptions are cheaper to create than HPX-thread objects. HPX-threads require the acquisition of an HPX-thread context, which requires a large chunk of contiguous memory for the HPX-thread's call stack. Furthermore, HPX-thread contexts are hardware dependent, which makes it more expensive and difficult to migrate them between nodes of different micro-architectures. By using HPX-thread descriptions, the HPX-thread scheduler is able to defer the acquisition of an HPX-thread context. This allows the creation operation to rapidly return to the caller. Additionally, stealing HPX-thread descriptions to resolve load imbalances is far cheaper than stealing HPX-thread objects. Stealing an HPX-thread object may move HPX-thread contexts across NUMA boundaries. Even in the case of steals that are performed within the same socket, cache and TLB penalties may be experienced.

A **terminated list** is also prescribed by the dual queuing model, for recording HPX-threads that need to be destroyed. The terminated list allows the destruction of HPX-threads to be deferred, similar to the deferral of creation operations. When HPX-thread objects are removed from the terminated list by the HPX-thread scheduler, they are partially reset. Every attribute of the HPX-thread object is restored to a pristine state, except for the HPX-thread context, which is not touched. We say that an HPX-thread object that has been reset in this manner is sanitized.

The final component of the dual-queuing model a **recycling manager**. A recycling manager is responsible for providing HPX-thread objects when HPX-thread descriptions are transitioned from staged to pending. The recycling manager receives sanitized HPX-thread objects that are removed from the terminated list. When the recycling manager needs to acquire an HPX-thread object, it either uses a sanitized HPX-thread object, or it creates a new one. If a sanitized HPX-thread object is used, then its context must be reset and loaded with the new payload function.

A collection of these four data structures - a staged queue, a pending queue, a terminated list and a recycling system - is referred to as a **queue package**. Every HPX scheduling policy uses queue packages. The quantity and hierarchy of queue packages may change from policy to policy. Additionally, two scheduling polices that use the same topology of queue packages may distinguish themselves by using different algorithms for load balancing between the queue packages. On the other hand, queue packages use HPX queuing policies. An HPX queuing policy specifies the implementation of the underlying container used by queue packages.

D. The Priority Local-FIFO Scheduler

In this paper, we present results using the Priority Local-FIFO scheduler, which is the name given to the scheduler formed by the composition of the default scheduling policy (Priority Local) and the default queuing policy (Lockfree FIFO).

The Priority Local scheduling policy uses one queue package per worker-thread for normal-priority HPX-threads. Each queue package "owns" the queue package that is associated with it, and will try to find work from that queue package before searching for work in other queues packages (e.g. work stealing). The queue package that is associated with a worker-thread is called the worker-thread's **local queue package**. Additionally, this scheduling policy uses a number of queue packages for high-priority HPX-threads and a single queue package per locality for low-priority threads.

Under the Priority Local algorithm, each worker-thread uses the following algorithm to find work to for it to do:

1) Check the local pending queue.
2) Check the local staged queue.
3) Steal from other staged queues.
4) Steal from other pending queues.
5) Perform maintenance (cleanup terminated HPX-threads, etc).

The Priority Local algorithm prefers to steal HPX-thread descriptions over pending HPX-thread objects because moving HPX-thread descriptions between processing units and across socket boundaries imposes fewer cache and NUMA penalties.

The Lockfree FIFO queuing policy uses a first-in, first-out queue implemented with compare-and-swap (CAS) atomics for pending and staged queues. Since LIFO queuing exhibits better cache behavior, and applications will not care about the order in which terminated lists are processed, we use a lockfree LIFO to implement the terminated list.

Lockfree algorithms generally have lower latencies than lock-based algorithms, because the lockfree code will never yield control back to the kernel. Keeping enqueuing/dequeuing latencies low is critical for HPX, as the runtime must be able to handle large quantities of small, short-lived HPX-threads.

The LIFO and FIFO used by this queuing policy are provided by the Boost.Lockfree library [4]. The FIFO implementation is based on the well known algorithm by Michael and Scott [16], [17]. The algorithm uses compare-and-swap atomic operations to implement a linked list. One downside

to using a linked-listed based FIFO is that it causes work-stealing to have $\mathcal{O}(n)$ time complexity. Currently, the Priority Local-FIFO algorithm does not implement bulk work stealing, instead searching for a single HPX-thread when it runs out of work. In practice, this limitation does not significantly reduce performance, except in artificial pathological situations. The lockfree LIFO is implemented as a CAS-based linked list, and has a less complex implementation as only one end of the list needs to be recorded [18], [17].

The Priority Local-FIFO scheduler features optional support for NUMA-sensitive scheduling, which may be enabled at runtime by users. These features include throttling of work-stealing across NUMA-boundaries and "smart" placement of HPX-threads so that they will execute on the socket where their data lives. Additionally, this scheduler's recycling system keeps reusable HPX-thread contexts from being moved between cores to reduce NUMA effects and increase cache performance.

Now, we turn our attention to the main results of this paper; an analysis of the fundamental overheads of HPX threading.

E. Experimental Setup

TABLE I: **Test Platforms**

System Name	**Ariel**	**Stampede**	**Lyra**
Processors	Intel Xeon E5-2690 Sandy Bridge	Intel Xeon Phi Knight's Corner	AMD Opteron 8431 Istanbul
Clock Freq.	2.9 GHz	1.1 GHz	2.4 GHz
Hardware Threading	2-way (deactivated)	4-way	No
Cores	16 (2 X 8)	61	48 (8 X 6)
NUMA Domains	2	1	8
Memory/Core	32KB L1 D & I 256KB L2	32KB L1 D & I 512KB L2	64KB L1 D & I 512KB L2
Shared Memory	29MB L3, 32GB DDR3	8 GB	98 GB

All of our experiments were performed with HPX V0.9.8, using version 1.55.0 of the Boost libraries. All x86-64 runs were performed on a system running Debian GNU/Linux Unstable and version 3.8.13 of the Linux kernel (non-preemptive). Xeon Phi runs were performed using the MPSS software stack (Linux 2.6.38). The jemalloc memory allocator was used for the x86-64 platforms, and tbbmalloc was used on the Xeon Phi. The default HPX build configuration was used, with guard pages turned off and RDTSCP-based timestamps enabled where available.

IV. HPX Performance Counters

To conduct a thorough analysis of the benchmarks studied in this paper, we used HPX's introspective performance monitoring mechanism, Performance Counters, to investigate both hardware and software performance behavior. Performance Counters are first-class representations of a singular-valued dependent variable which describes some aspect of software or hardware performance. They can be used for runtime decision making as well as post-run performance analysis and debugging. Counters fall into four categories:

- Hardware Counters, are obtained from hardware performance monitoring mechanisms. Examples: total clock cycles, number of L1 DTLB misses.
- Kernel Counters, values relating to aspects of the execution that are managed by the kernel. Examples: peak resident memory, number of virtual memory mappings.
- Runtime Counters, are exposed by the HPX runtime and describe the internal state of the runtime system. Examples: HPX-thread queue length, cumulative HPX-threads executed.
- Application Counters, which are implemented by user code and describe metrics specific to a particular application.
- Derived Counters, which take one or more Performance Counters as an input and produce a singular output value.

In addition to having a global identifier (GID), each Performance Counter has a unique, hierarchal string which is bound to it's GID. This symbolic binding is akin to the association of domain names to numeric IP addresses in the Domain Name System. The advantage of this binding is that any performance counter can be accessed remotely at any point of time during application execution, even if the running program never distributed the GIDs of its performance counters to the interested parties. This enables performance monitoring programs on hand-held devices to be able to query performance data from HPX applications in real time.

In the research described in this paper, we used performance counters (primarily hardware and runtime counters) to evaluate and eliminate invalid hypothesis about runtime performance, and to quantify the effects of our valid hypothesis. Perhaps most importantly, we were able to identify many complex bugs that had been overlooked by traditional debugging methods. Some of these bugs not only had substantial performance impacts but also caused incorrect semantics.

Most of the performance counters that are presented in this paper report memory and address resolution behavior on the Intel systems. The impacts of the memory subsystems were determined using hardware counters (provided via PAPI) for data cache and translation lookaside buffer (TLB) misses, number of cycles used for TLB walks, the total number of clock cycles, and for the Xeon Phi system the number of cycles used for computational instructions. Hardware counters are shown in Table II.

We compute cache and TLB impacts based on the average cycle latency reported by Intel [19] [20] multiplied by the measured counts, with the exception of DTLB walk duration (which measures cycles directly). Table III shows the formulae for computing the percentage of execution time wasted. The cache overheads only consider data loads and stores. Other potential memory overheads, such as instruction cache misses and store forwarding blocks were measured on the Sandy Bridge system and found to be insignificant.

The Xeon Phi L2 cache miss counters include prefetch misses, which makes it difficult to measure the impact of L2 cache misses. We compute an estimated latency impact (ELI) [20] by dividing the number of cycles not used for computation minus loads and stores by demand L1 data misses. An ELI close to the L2 data access latency of 21 cycles indicates the majority of L1 data misses are hitting in

TABLE II: **Intel Hardware Performance Counters**

Sandy Bridge		
Symbol	**PAPI Counter**	**Impact Cause**
cycles	CPU_CLK_UNHALTED	Reference
L2hit	MEM_LOAD_UOPS_RETIRED:L2_HIT	Missed L1, hit L2
L3hit	MEM_LOAD_UOPS_RETIRED:L3_HIT	Hit L3, no snoop
L3hitxs	MEM_LOAD_UOPS_LLC_HIT_RETIRED:XSNP_HIT	Hit on-package core cache w/ clean snoop hit
L3hitxsm	MEM_LOAD_UOPS_LLC_HIT_RETIRED:XSNP_HITM	Hit on-package core cache w/ snoop hit & conflict
L3miss	MEM_LOAD_UOPS_MISC_RETIRED:LLC_MISS	Missed LLC, accessed main memory
STLBhit	DTLB_LOAD_MISSES:STLB_HIT	Miss 1st level, hit 2nd level
DTLBwalk	DTLB_LOAD_MISSES:WALK_DURATION	Cycles spent in page walks

Xeon Phi		
Symbol	**PAPI Counter**	**Impact Cause**
cycles	CPU_CLK_UNHALTED	Reference
L1miss	DATA_READ_MISS_OR_WRITE_MISS	L1 data cache misses
L1misshpf	DATA_HIT_INFLIGHT_PF1	L1 miss, hit prefetch from L2
EXECcycles	EXEC_STAGE_CYCLES	Cycles executed computational instructions
LDST	DATA_READ_OR_WRITE	Loads and Stores seen by L1 data cache
L2ldmiss	PAPI_L2_LDM	L2 load misses
DTLB1	DATA_PAGE_WALK	Level 1 TLB data misses
DTLB2	LONG_DATA_PAGE_WALK	Level 2 TLB data misses

TABLE III: **Intel TLB and Cache Impact**

Sandy Bridge	
Impact	**Formula**
L1 Miss Penalty	$(12 * L2_{hit})/cycles$
L2 Miss Penalty	$(26 * L3_{hit} + 43 * L3_{hitxs} + 60 * L3_{hitxsm})/cycles$
L3 Miss Penalty	$(200 * L3_{miss})/cycles$
TLB Miss Penalty	$(7 * STLB_{hit} + DTLB_{walk})/cycles$

Xeon Phi	
Impact	**Formula**
L1 Miss Penalty	$(25 * (L1_{miss} + L1_{misshpf}))/cycles$
ELI	$(cycles - exec_{cyc} - LDST)/L1_{miss}$
L2 Miss Penalty	$250 * L2_{ldmiss}/cycles$
TLB Miss Penalty	$(25 * DTLB1_{walk} + 100 * DTLB2_{walk})/cycles$

L2. Although L2 cache miss counters may include prefetch data misses, we measured the L2 load misses to determine how they relate to the ELI.

V. Results

To study the overheads of the HPX threading subsystem, we require a controlled environment. Our primary goal is to quantify ”baseline” overheads that can be used as a launching point for future studies into the behavior of overheads in HPX. Additionally, we wanted to explore the costs associated with different degrees of application granularity.

Since every HPX subsystem makes use of HPX-threads, it is necessary to understand the costs of HPX-threads before we can hope to understand the overheads associated with other parts of the HPX runtime. This led us to restrict our study to shared-memory systems, as we wanted to isolate HPX threading overheads from network overheads originating in the parcel subsystem.

There are also overheads within the HPX threading subsystem that we want to eliminate, in order to make the scope of our ”baseline” study feasible. We wanted to restrict our set of independent variables to:

- **Amount of Processing Units**, which controls the degree of parallelism.
- **Payload Duration**, which controls the grain size.
- **Quantity of HPX-threads**, which controls the length of the queues and the number of HPX-threads that the HPX scheduler must manage.
- **Quantity of Suspensions**, which affects the efficiency of the HPX-thread context recycling system.

Notably, none of these independent variables has a non-trivial effect on the amount of work-stealing that occurs. In fact, we actively seek to ensure that negligible work-stealing occurs in our results. While work-stealing is an important and distinctive feature of HPX, we decided to exclude it from this study, both to limit the scope of our experiment and because we believe that this work is a prerequisite for any study of work-stealing in HPX. Work-stealing algorithms are usually the primary distinguishing factor between HPX schedulers, so it would further be necessary for us to present a detailed analysis and comparison of different HPX schedulers for us to discuss work-stealing.

We developed a synthetic application, the Homogeneous Timed Task Spawn (HTTS) benchmark, to implement our desired testing environment. HTTS spawns a controllable number of HPX-threads, each with an identical payload function which has a controllable duration. This controllable payload function performs no computation and makes negligible memory accesses; it performs live-waiting until it has reached its target duration. HTTS has the capability to simulate different conditions of load balance between queue packages. Additionally, artificial synchronization can be modelled by indicating that a certain ratio of HPX-threads should suspend.

As mentioned, in this study we will not examine the overheads of work-stealing. To ensure that no work-stealing occurs in our runs, we configure HTTS to statically partition all of the HPX-threads that it spawns. We are using the Priority Local-FIFO scheduler, so there is one queue package associated with each worker-thread. In our runs HTTS places an approximately equal number of HPX-threads, each with an identical payload duration, into each worker-thread's queue package.

We began by investigating HTTS with no suspensions. In

Fig. 1: **HTTS EP Efficiency: Platform Survey**

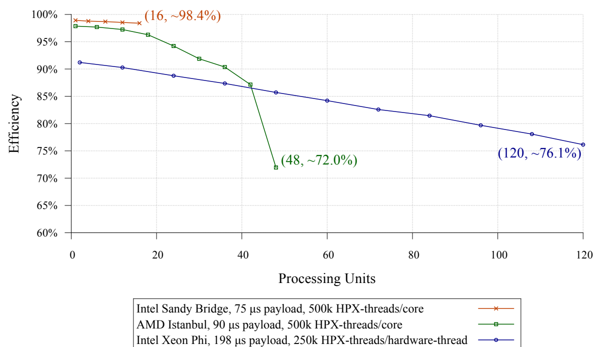
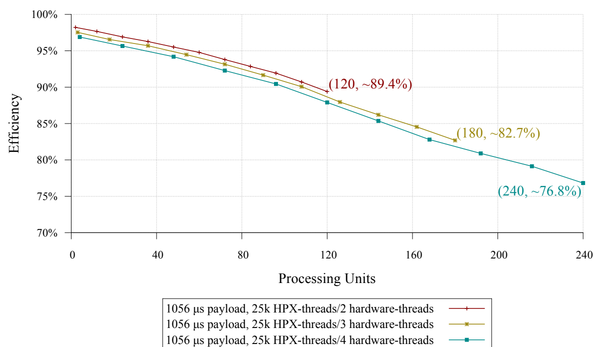


Fig. 2: **HTTS EP Efficiency: Xeon Phi**



this case, there is little to no communication or synchronization (artificial or otherwise) because no work-stealing occurs and all of the HPX-threads spawned by HTTS execute completely independent of each other.

Under these conditions of no work-stealing and no suspensions, HTTS is an embarrassingly parallel (EP) problem. We will call HTTS configured in this fashion **HTTS under EP conditions or HTTS EP**.

We can define a concise theoretical performance model under these conditions. With m HPX-threads per worker-thread, n worker-threads, a payload duration of p, the theoretical walltime w_T is simply:

$$w_T = \left(\frac{\text{HPX-threads}}{\text{Worker-thread}}\right)(\text{Payload Duration})$$
$$w_T = mp$$

If the measured walltime for a run is w_M, then the **efficiency** E of that run is:

$$E = \frac{\text{Theoretical Walltime}}{\text{Measured Walltime}}$$
$$E = \frac{w_T}{w_M} = \frac{mp}{w_M}$$

Figure 1 presents HTTS efficiency results cycles on our test systems: a 2-socket Intel Sandy Bridge system and a Xeon Phi co-processor. We used 500k HPX-threads per core for all platforms (e.g. 250k HPX-threads per hardware-thread on the Xeon Phi). Figure 2 shows the HTTS efficiency results on the Xeon Phi for differing levels of hyperthreading.

To account for different clock speeds, we defined the payload in terms of cycles, and computed payload durations for each platform. The payload selected was 217.5k cycles, corresponding to 75 µs on the Sandy Bridge system, and 198 µs on the Xeon Phi. Despite the embarrassingly parallel nature of our problem, we can clearly see that efficiency retains some dependence on the degree of parallelism.

On the Intel Xeon system, we observed largely linear behavior across a wide range of parameters, and multi-socket penalties appear to be negligible. The Xeon Phi is a single-socket system, so we expected to see linear behavior.

We performed a simple linear regression on the Sandy Bridge and Xeon Phi efficiency data, with efficiency on the y-axis, and the number of processing units minus 1 (e.g. $n-1$) on the x axis. The y-intercept of such a linear regression represents the **serial efficiency** of the application, or what percentage of theoretical performance is achieved when the application is run on one processing unit. If the behavior of efficiency with respect to the number of processing units is monotonically decreasing - that is, as long as Amdahl's law is not violated - then the 1-core case is the most efficient case.

The slope in these linear fits depicts the change in efficiency divided by the change in amount of parallelism (e.g. number of processing units). For the scope of the work we present in this paper, we assume that this value is negative. We say the magnitude of the slope is the **inefficiency gradient** of an application that exhibits linear efficiency behavior on a particular system. We can view this metric as a representation of **scalability**, or how well running the application in parallel (e.g. $n > 1$) performs compared to the 1-core case.

It is critical to consider both serial efficiency and scalability when studying a parallel application, because comparing applications using just one of these metrics will hide differences in performance. Two applications may exhibit the same speedup characteristics but have very different serial efficiencies, and vice versa. These concepts need not be limited to linear efficiency behavior.

In Table IV we list the 1-core efficiency and inefficiency gradients for various runs on our Xeon Phi and Sandy Bridge system. We also list the coefficients of determination ($0 < R^2 < 1$) obtained from the runs - the closer R^2 is to 1, the better the fit is. The high R^2 for the Sandy Bridge node indicates there is no significant change of the behavior when we cross the socket boundary (from 8 cores to 9 cores). Our analysis indicates that we can be confident that HTTS EP scales linearly on both platforms.

In the 75 µs payload runs on the Sandy Bridge platform, we achieve 98.8% efficiency in the 1-core case, so we know that the serial overheads are very low relative to the payload duration. For these runs, we lose only 0.03% efficiency with each additional core that we add, indicating very strong scalability.

On the Xeon Phi, we were interested in examining how well HTTS EP would perform when running with different hardware-thread to core ratios. The Knight's Corner generation of Xeon Phi co-processors has four hardware-threads per core, which helps to hide instruction latency. The Phi is not capable of executing from the hardware context for two cycles in a row [20] . For optimal performance, it is necessary to use at least two of these hardware-threads per core. We note that our selection of grain size for these runs was made both to account for limitations of the hardware clock and because the 1056 µs grain size is of the same magnitude of the granularity used in some of the existing HPX applications we wish to run on the Phi.

When we compare the results shown in Figures 1 and 1 with the models from Table IV, we see that our linear models fall within the range of confidence of our measured data across most of our data set. In Table V, we show the predicted and measured efficiencies for the runs with maximal parallelism (e.g. the largest number of cores).

VI. The HPX-Thread Life Cycle

We will now discuss our technique for quantifying the overheads of threading in HPX applications, and apply it to HTTS under EP conditions.

As discussed in Section III, HPX-threads are finite-state machines. To quantify the overheads of HPX-threads, we identify the measurable events that happen to HPX-threads throughout their lifetime. These events form the HPX-thread life cycle. Each of these steps involves at least one expensive operation; a large memory allocation/deallocation/access, locking, or atomics. The steps are followed sequentially by every HPX-thread, with the exception of a single three-way branch. Deferred execution (e.g. fire and forget semantics) separates some of the steps. We now outline each step, and explain its expected performance effects.

- **Describe:** Users of the HPX-threading subsystem create HPX-threads by passing an HPX-thread *description* to the HPX scheduler. This operation is inexpensive, as it involves no locking, atomics or large allocations (since the HPX-thread description has no HPX-thread context).
- **Push Staged:** The HPX-scheduler *pushes* the HPX-thread description to a *staged* queue in one of the queue packages that it manages. This operation has O(1) complexity, and involves compare-and-swap (CAS) operations.
- **Pop Staged:** Eventually, the HPX-scheduler *pops* the HPX-thread description from the *staged* queue (uses atomics).
- **Create:** The staged HPX-thread description is used to *create* a pending HPX-thread. This step involves a large memory allocation, if a new HPX-thread context must be acquired from the kernel. If the HPX-thread context is provided by the recycling manager, the HPX-thread context will still need to be reset and rebound to the HPX-thread's payload function. In both cases, expensive calls must be made into the OS virtual memory manager, and pressure will be placed on the TLB and data caches. The Create step may also require locking or atomic operations.
- **Push Pending:** The HPX-thread, which is now pending, is *pushed* to a *pending* queue. This operation uses compare-and-swap atomics.
- **Pop Pending:** A worker-thread selects the HPX-thread for execution by *popping* it from the *pending* queue.
- **Scheduler to HPX-thread:** The HPX *scheduler* context-switches to the *HPX-thread*. The HPX-thread is now active, and will execute the payload function. The overhead here is simply the cost of a context switch, which we can measure via a microbenchmark.
- **Scheduler to HPX-thread:** Once the payload function has been executed, the *HPX-thread* will context-switch back to the *scheduler*. Depending on the supplied exiting state, one of three paths is taken:
  - If the exiting state is suspended, the HPX-thread waits until an external entity set its state to pending. Then, return to **Push Pending**.
  - If the exiting state is pending, return to **Push Pending**.
  - If the exiting state is terminated, proceed to the next step.
- **Append Terminated:** The HPX-thread is *appended* to the *terminated* list. The terminated list is implemented in this scheduler using a lockfree LIFO, so this operation involves atomics.
- **Remove Terminated:** When the HPX scheduler decides to perform maintenance, the HPX-thread will be *removed* from the *terminated* list.
- **Recycle:** Finally, the HPX-thread is sanitized and *recycled*. This is a potentially expensive operation. It requires the acquisition of locks inside of the HPX scheduler and may destroy (possibly remote) objects that have references bound to the terminated HPX-thread.

We start by addressing steps that consist solely of operations on the lockfree queues. As we mentioned in Section III, the staged and pending queues are FIFOs, and the terminated list is a LIFO. Both are lockfree CAS-based implementations.

These steps of the life cycle are the simplest to address because we can easily microbenchmark them. We determined the cost of the push and pop operations of both the LIFO and the FIFO by amortized analysis. We did not consider the performance of either data structure under concurrent access, as we know that this will not occur in HTTS EP (due to a lack of work stealing). Our objective was to determine a lower bound cost. In the microbenchmark, each OS-thread pushes a specified number of objects into a locally created queue, and then removes the same number of objects from the queue. The test performs and times a few hundred to a few thousand iterations at a time (a chunk of the total requested iterations), to ensure that the test is not skewed by memory effects. We summarize our results in Table VI. The cost per lockfree operation under these embarrassingly parallel conditions grows

TABLE IV: **Linear Models of HTTS Efficiency on Intel Platforms**: Simple linear regressions of efficiency vs processing units on our Intel systems. A fixed theoretical walltime of 37.5 seconds was used for the Sandy Bridge runs, and 25k HPX-threads per hardware-thread were used for the Xeon Phi. We used a payload of 1056 µs (1.1 million cycles)for the Xeon Phi runs.

Sandy Bridge Models			
Parameters	**Inefficiency Gradient (Magnitude of Slope)**	**Serial Efficiency (y-intercept)**	R^2
500k HPX-threads/core 75 µs (217.5k cyc)	0.0339%	98.8%	0.972

Xeon Phi Models			
Parameters	**Inefficiency Gradient (Magnitude of Slope)**	**Serial Efficiency (y-intercept)**	R^2
2 HW-threads/core	0.0728%	98.7%	0.986
3 HW-threads/core	0.0849%	98.5%	0.987
4 HW-threads/core	0.0879%	98.0%	0.995

TABLE V: **Comparison of Linear Models to Measured Data**: The linear models presented in Table IV line up well with our measurements. The linear models remain within the range of confidence of the measured data.

Sandy Bridge Models		
Parameters	**Predicted Efficiency**	**Measured Efficiency**
500k HPX-threads/core 75 µs (217.5k cyc)	98.3% on 16 cores	98.4% ± 0.1% on 16 cores

Xeon Phi Models		
Parameters	**Predicted Efficiency**	**Measured Efficiency**
2 HW-threads/core	90.0% on 120 cores	89.4% ± 0.6% on 120 cores
3 HW-threads/core	83.2% on 180 cores	82.7% ± 0.5% on 180 cores
4 HW-threads/core	76.9% on 240 cores	76.8% ± 0.3% on 240 cores

very slowly with respect to the number of cores.

Every HPX-thread will incur at least 2 FIFO pushes (Push Pending and Push Staged), 2 FIFO pops (Pop Pending and Pop Staged), 1 LIFO push (Append Terminated) and 1 LIFO pop (Remove Terminated) during it's lifetime. Each suspension will invoke at least two more, as a suspended HPX-thread must re-enter a pending queue.

Using these regressions, an approximate lower bound for cost of queuing per HPX-thread on the Sandy Bridge is $4.412n + 130$ nanoseconds (where n is the number of worker-threads). In the 1-core case, this is 12% of total inefficiency per HPX-thread and in the 16-core case the cost is 10% of total inefficiency per HPX-thread. On the Xeon Phi, the cost of queuing per HPX-thread is $0.4934n + 1775$ nanoseconds. This is 7% of the total inefficiency per HPX-thread in the 2-PU case, and 5% of the total inefficiency per HPX-thread in the 120-PU case. The decrease in percentage of total inefficiency indicates that the queuing costs are not the factor driving the inefficiency of HTTS EP.

Using a microbenchmark, we were able to measure the amortized overheads of the context-switching routine used by Scheduler to HPX-thread and HPX-thread to Scheduler steps. The results of the microbenchmark indicated that this overhead was not dependent on the number of cores used by the application. Thus, we were able to come up with simple estimates for the context-switching overhead per HPX-thread. On the Sandy Bridge platform, the overhead per context switch is 45 ns ± 3 ns (6% of the total inefficiency). On the Xeon Phi, the overhead per context switch is 1300 ns ± 40 ns (10% of the total inefficiency). Because this cost does not depend on the amount of parallelism, it cannot be a driving factor of the inefficiency of HTTS EP.

TABLE VI: **Lockfree Push/Pop Costs in HTTS EP**: We modeled the amortized costs of the FIFO/LIFO operations using simple linear regressions of the data, with the number of cores (n) as the independent variable.

Sandy Bridge Models		
Operation	**Amortized Cost (n cores)**	R^2
FIFO Push	$0.715n + 20.0$	0.934
FIFO Pop	$1.11n + 13.7$	0.950
LIFO Push	$0.176n + 13.8$	0.922
LIFO Pop	$0.205n + 16.6$	0.919

Xeon Phi Models		
Operation	**Amortized Cost (n cores)**	R^2
FIFO Push	$0.0883 * x + 306.6$	0.914
FIFO Pop	$0.0617 * x + 248.6$	0.899
LIFO Push	$0.0489 * x + 164.1$	0.885
LIFO Pop	$0.0478 * x + 168.2$	0.882

To determine the cost of the Describe step, we look at the three computational phases of the timed portion of HTTS EP's kernel. These phases are:

- **Warmup Phase:** The period of time in which HPX-threads are still being created. We can measure the duration of this phase by using an HPX performance counter that measures the total length of all staged and pending queues. We evaluate this performance counter at regular intervals (10 µs) during the execution of the timed portion of HTTS EP. The warmup phase ends when this performance counter achieves a maximum value.
- **Hot Phase:** The period of time in which no HPX-threads are created and no work stealing happens. This is the phase that we are primarily interested in measuring. To

determine the length of this interval, we look at the queue length counter as well as a counter that records the cumulative number of HPX-threads stolen throughout the system. If we are in the hot phase, queue lengths are decreasing and the number of HPX-threads stolen remains constant. These are necessary but not sufficient conditions, although we have found them to be suitable except in the case of payloads smaller than 1 µs.

- **Cooldown Phase:** The period of time in which no tasks are being created and work stealing is happening. We say cooldown is the period in which no new HPX-threads are being created and work stealing is occuring. During the cooldown phase, queue lengths are decreasing and the number of HPX-threads stolen is increasing.

In HTTS EP, the Describe step for every application HPX-thread occurs during the warmup phase. During the warmup period, all worker-threads are enqueuing HPX-threads - each worker-thread enqueues an equal number of HPX-threads. A synchronization primitive is used to ensure that the none of the worker-threads begin executing HPX-threads before all HPX-threads have been enqueued. We have found the warmup phase to have a negligble duration relative to the hot phase, except in the case of very small payloads and very large HPX-thread quantities.

The Creation and Recycle steps are difficult to measure in a microbenchmark, because they are tightly integrated with the HPX scheduler. Since we were unable to extract a microbenchmark to measure these overheads, we instead added new HPX performance counters which would record the amount of walltime spent performing operations associated with these steps.

We observed that on both the Xeon Phi and the Sandy Bridge, the majority of scheduler overhead (40-60%) could be associated with the Creation step. This result is not surprising, since the Creation step (and to a lesser degree the Recycle step) is the only place in the HPX-thread life cycle where:

- **Non-Trivial Memory Allocation Occurs** HPX-thread contexts are stack-based data structures which typically occupy multiple pages of virtual memory. Storage space for these thread contexts usually dominants the memory profile of fine-grained HPX applications. HPX's thread context recycling mechanism is capable of mitigating the overheads of these allocations, but the initial allocation of virtual memory from the operating system is a non-trivial operation that blocks at the OS-level. Thus, in comparison to the rest of the life cycle steps, the allocations which may occur in the Creation step have significant overheads and scaling limitations.
- **Thread Contexts are Accessed Directly** While recycled thread contexts reduce overheads by removing unnecessary allocations at the OS level, the Creation and Recycle steps must ensure that encapsulation and system security are maintained by clearing and reinitializing terminated thread contexts. The order in which HPX-thread contexts are recycled is inherently irregular. New thread contexts are allocated from sequential regions of virtual memory, but thread contexts are recycled with FIFO semantics. FIFO recycling reduces the algorithmic overhead of recycling and should promote good cache behavior, however it also leads to memory fragmentation and TLB thrashing.

Fig. 3: **HTTS EP Overhead Breakdown: Sandy Bridge**

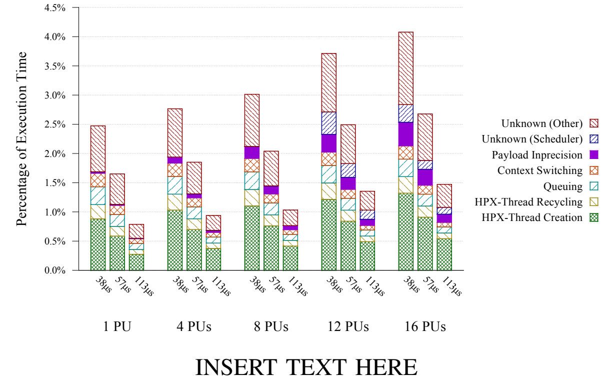
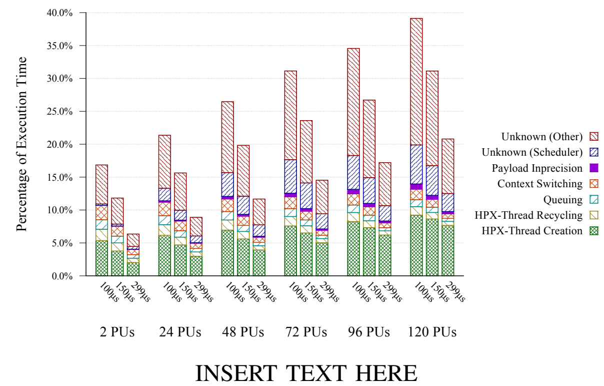


Fig. 4: **HTTS EP Overhead Breakdown: Xeon Phi**



Therefore, we conclude that the Creation and Recycle steps are the limiting factor behind HPX-thread overheads. Figures 3 and 4 show the breakdown of the measured per-HPX-thread overhead in HTTS EP. As we have discussed, the context switching overheads are fixed and represent a small portion of the total overheads. The queuing overheads grow linearly as we add more parallelism, but the rate of growth is comparatively minor. The overheads associated with the Creation and Recycle steps are significant contributors to the overall scheduler overhead, and these overheads increase when more execution units are utilized.

Our results indicate that the efficiency of the HPX scheduler will be affected by the behavior of the HPX-thread recycling system and the quantity of live HPX-thread contexts in an application. Since the recycling system is unable to reuse suspended HPX-thread contexts, this leads us to conclude that applications which frequently suspend HPX-threads may experience degraded performance due to excessive thread context allocations.

VII. Related Work

Comparable multi-threading runtime systems with work stealing threading solutions include Qthreads library [21], [22], Intel Thread Building Blocks (TBB) [23], and Cilk++ [24]. The Qthreads library supports lightweight threads with fast user level context switching and thread stealing mechanisms. Qthreads uses full empty bit (FEB) synchronization of each memory location for fast context switching and employs futures and mutex synchronizations. Using the Rose Compiler, OpenMP source code can be transformed into Qthreads applications, breaking the larger OpenMP loops into smaller lightweight qthreads. TBB, a C++ parallel library, distributes tasks among parallel processors, supports partition affinity for loops and implements random task stealing. Charms++, a C++ parallel library, uses communicating objects called chares which exchange data through remote method invocations. The runtime system captures statistics for load balancing. Cilk++ is a compiler based lightweight threading solution which implements local spawn methods with local barriers and parallel for loops. Cilk is an extension that gives users an easy way to implement parallelism to conventional codes, but employs conventional CSP processing for limited scaling of applications.

VIII. Future Work

These experiments quantify the overheads of the threading system of HPX without the complication of work stealing and synchronization. It gives us the groundwork and measurements necessary to compare the various thread scheduling policies of HPX and to further investigate work stealing and synchronization overheads. The results from these experiments also indicate some additions to our work stealing algorithms such as skipping the staged state when pending work is depleted. Further work needs to be done by making similiar measurements and comparison of irregular and regular applications or benchmarks using the HPX runtime system. Future research is planned in applying the knowledge gained from metrics at runtime to tune thread scheduling dynamically.

Acknowledgment

References

[1] *ParallelX: An Advanced Parallel Execution Model for Scaling-Impaired Applications*, ser. ICPP '09, 2009.

[2] A. Tabbal, M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling, "Preliminary Design Examination of the ParalleX System from a Software and Hardware Perspective," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 81–87, 2011.

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

[4] "Boost: a collection of free peer-reviewed portable C++ source libraries," 2013, http://www.boost.org/. [Online]. Available: http://www.boost.org/

[5] 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

[6] D. P. Friedman and D. S. Wise, "Cons should not evaluate its arguments," in *ICALP*, 1976, pp. 257–284.

[7] 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

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

[9] 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.

[10] 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.

[11] PGAS, "PGAS - Partitioned Global Address Space," 2013, http://www.pgas.org. [Online]. Available: http://www.pgas.org

[12] 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

[13] 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.

[14] 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

[15] 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

[16] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 1996, pp. 267–275. [Online]. Available: http://www.research.ibm.com/people/m/michael/podc-1996.pdf

[17] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1734069

[18] I. B. M. C. R. Division and R. Treiber, *Systems Programming: Coping with Parallelism*, ser. Research Report RJ. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986. [Online]. Available: http://domino.research.ibm.com/library/cyberdig.nsf/0/58319a2ed2b1078985257003004617ef?OpenDocument

[19] "Intel® 64 and IA-32 Architectures Optimization Reference Manuals," http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[20] http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding.

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

[22] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins, "Scheduling task parallelism on multi-socket multicore systems," in *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '11. New York, NY, USA: ACM, 2011, pp. 49–56. [Online]. Available: http://doi.acm.org/10.1145/1988796.1988804

[23] "Intel® Thread Building Blocks 4.2," http://www.threadingbuildingblocks.org/.

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