Using Intrinsic Performance Counters to Assess Efficiency in Task-based Parallel Applications

Patricia Grubel*
pagrubel@nmsu.edu
New Mexico State University
Las Cruces, New Mexico, U.S.A.

Hartmut Kaiser *
hkaiser@cct.lsu.edu
Center for Computation and Technology
Louisiana State University
Baton Rouge, Louisiana, U.S.A.

Kevin Huck
khuck@cs.uoregon.edu
University of Oregon
Eugene Oregon, U.S.A.

Jeanine Cook*
jeacook@sandia.gov
Sandia National Laboratories
Albuquerque, New Mexico, U.S.A.

* STE||AR Group (stellar-group.org)

***Abstract*—The ability to measure performance characteristics of an application at runtime is essential for monitoring the behavior of the application and the runtime system on the underlying architecture. Traditional performance measurement tools do not adequately provide measurements of asynchronous task-based parallel applications, either in real-time or for postmortem analysis. We propose that this capability is best performed directly by the runtime system for ease in use and to minimize conflicts and overheads potentially caused by traditional measurement tools.**

In this paper, we describe and illustrate the use of the performance monitoring capabilities in the HPX [13] runtime system. We describe and detail existing performance counters made available through HPX's performance counter framework and demonstrate how they are useful to understanding application efficiency and resource usage at runtime. This extensive framework provides the ability to asynchronously query software and hardware counters and could potentially be used as the basis for runtime adaptive resource decisions.

We demonstrate the ease of porting the Inncabs benchmark suite to the HPX runtime system, the improved performance of benchmarks that employ fine-grained task parallelism when ported to HPX, and the capabilities and advantages of using the *in-situ* performance monitoring system in HPX to give detailed insight to the performance and behavior of the benchmarks and the runtime system.

***Index Terms*—runtime instrumentation, performance counters; execution monitoring; HPX; task-based parallelism; many asynchronous tasks**

I. Introduction

In today's world the landscape of computing is quickly changing. Clusters continue to grow more complex with an increasing variety of heterogeneous nodes, networks are becoming more intricate, and memory hierarchies deeper. Additionally, applications are growing larger, more complex, and more integrated with other applications in workflows. It has beome clear that new techniques and paradigms such as task-based parallelism and runtime adaptivity will need to be applied in order for scientists to continue to exploit the parallelism that the machines of the future will offer. These techniques, however, heavily depend on readily available, low overhead, performance metrics that are able to take a view of the entire system, not only to understand the performance of our applications, but also to use the information to dynamically tune codes in their run environment.

Task-based programming models have slowly emerged over the last three decades from examples such as Charm++ [16] and OpenMP 3.0 tasks [5] to include many other research and commercial runtimes. Unfortunately, each model requires a specific solution to the problem of performance measurement. As we will demonstrate in this paper, current tools are designed for the much more common case of synchronous execution and are sometimes unable to provide application developers the information they need to debug and improve the codes that are using asynchronous task models. One such example community that is affected is the committee that is working on the implementation of task parallel constructs in the C++11 Standard.

The implementation of task parallel constructs in the C++11 Standard is designed to increase parallel programming productivity and portability of parallel applications with the potential of increased performance due to compiler support. To assess the performance of the Standard implementation, Thoman, Gschwadtner, and Fahringer introduced the Innsbruck C++11 Async Benchmark Suite (Inncabs [18]), consisting of parallel benchmarks with varying task granularities and synchronization requirements. The performance study using Inncabs illustrates the use of the C++11 Standard parallel constructs across readily available compiler platforms. However, their results demonstrate that the Standard implementation of the parallel constructs do not perform well and are not adequate to replace current third party implementations of task based parallelism.

Most of the widely used open-source parallel performance measurement tools (such as HPCToolkit [4] or TAU [17])

are based on profiling and/or tracing of application codes through either instrumentation and/or periodic sampling. These types of tools are quite useful for post-mortem analysis and optimization of large scale parallel application codes. However, these tools currently fail to support the current implementation of C++ task parallel constructs. For example, the GCC[1] task parallel runtime constructs, executes, and destroys an Operating System (OS) thread for every task launched with `std::async`, resulting in thousands or even millions of OS threads being created. Neither HPCToolkit nor TAU are designed to deal with millions of short-lived threads within a single process space and either introduce unacceptable amounts of overhead or crash altogether. In addition, because they are designed for post-mortem analysis they are not easily extended to implement runtime adaptive mechanisms.

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

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

This paper illustrates the capabilities of the HPX runtime system to schedule massive numbers of small tasks efficiently for parallel applications with the ability to monitor intrinsic software and hardware counters at runtime. The Standard C++ solution based on kernel threads is not sufficient to provide adequate scalability of parallel applications and third party runtime libraries are required for such support. Contributions of the research illustrate:

1) HPX provides improved scalability and performance over the C++11 thread support library for parallel applications that schedule massive quantities of fine-grained asynchronous tasks with the same programmability of the C++ Standard. We present the ease of porting the Inncabs parallel benchmark suite to use the HPX runtime system with significant improvement in scalability and performance for the benchmarks that employ fine grain parallelism.
2) The capability and advantage of employing introspective measurement techniques by the runtime to monitor events of task-based parallel applications. This enables the runtime or application to assess performance information at runtime and paves a path for possible implementation of real time adaptive techniques in addition to post-processing capabilities.
3) The ability of the HPX runtime system to provide the same information that external tools can extract from the application without the overheads, resources, and expertise required to run an external tool. Section II presents in further detail the challenges of monitoring inherent events of task-based parallel applications using available external performance tools to assess performance and system utilization.

[1]https://gcc.gnu.org

II. C++ Parallel Algorithm Challenges

It is important to understand how C++ parallel constructs are distinct from other thread-parallel models, and why it is so difficult to measure and understand the performance behavior. Implementation decisions in the runtime library have a significant effect on the observability of the resulting application, as we have briefly mentioned in Section I. Take, for example, the GCC task parallel runtime. This runtime implementation constructs, executes, and destroys an Operating System (OS) thread for every task created with `std::async` task, resulting in thousands or even millions of OS threads being created. While this implementation is certainly within the capabilities of the OS kernel, it is somewhat naive and inefficient and presents a significant challenge to performance tools that are not explicitly designed to support this type of execution model implementation.

As described in Section I, widely used open-source parallel performance measurement tools like HPCToolkit [4] and TAU [17] provide profiling and/or tracing of many different types of parallel application models. These tools use several methods to observe application behavior, including source instrumentation, binary instrumentation, library wrappers, performance tool interfaces, callbacks, and/or periodic sampling based on interrupts combined with call stack unwinding. These tools are capable of context-aware measurement with variable levels of resolution and subsequent overhead. Large scale parallel codes with concurrency greater than hundreds of thousands of processes and/or threads have been successfully measured. However, these tools fail to adequately support the current implementation of C++ task parallel constructs. Both TAU and HPCToolkit make design assumptions about the overall number of OS threads they expect to see in a given process space. In the case of TAU, the data structures used to store performance measurements are constructed at program launch to minimize perturbation of the application during execution. While the maximum number of threads per process is a configurable option (default=128), it is fixed at compilation time. Even when set to a much larger number (i.e. 64k) TAU causes the benchmark programs to crash. While HPCToolkit doesn't set a limit on the number of threads per process, the introduced overhead becomes unacceptable as each thread is launched and the file system is accessed, and in most benchmark cases the program crashes due to system resource constraints. Table I shows the results of running the

Inncabs benchmarks at full concurrency with either TAU or HPCToolkit using the test system and protocol described in Section V.

TABLE I
OVERHEAD AND OBSERVED FAILURES FOR THE INNCABS BENCHMARKS EXECUTED WITH TAU OR HPCTOOLKIT.

	Baseline		TAU	HPCToolkit	
Benchmark	time	tasks	time	time	overhead
alignment	971	4950	SegV	112,795	11516.37%
fft	48,423	2.04E+06	SegV	timeout	
fib	Abort	n/a	SegV	n/a	
floorplan	5,788	169708	SegV	SegV	
health	589,415	1.75E+07	Abort	Abort	
intersim	827	1.70E+06	Abort	SegV	
nqueens	Abort	n/a	n/a	n/a	
pyramids	2,148	112,344	SegV	275,088	12706.70%
qap	SegV	n/a	n/a	n/a	
round	155	512	SegV	5,588	3505.16%
sort	7,240	328,000	SegV	Abort	
sparselu	786	11,099	SegV	99,123	12511.07%
strassen	4,782	137,256	SegV	Abort	
uts	Abort	n/a	n/a	n/a	

In addition, because they are designed for post-mortem analysis these tools are not easily extended to implement runtime adaptive mechanisms. In both cases, post-processing of the performance data (usually done at program exit) is required before an accurate performance profile containing the full system state (across nodes and/or threads) is possible. In contrast, the APEX [11], [10] library has been designed for the HPX runtime to provide performance introspection and runtime adaptation using the available HPX performance counter framework. A longer discussion of APEX is beyond the scope of this paper, but a discussion of its current state and future potential is briefly provided in Section VII.

III. HPX – A GENERAL PURPOSE PARALLEL C++ RUNTIME SYSTEM

HPX is a general purpose C++ runtime system for parallel and distributed applications of any scale. It has been described in detail in other publications [8], [9], [13], [14]. We will highlight its main characteristics in this section [15].

HPX represents an innovative mixture of a global system-wide address space (AGAS - Active Global 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. 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 petascale to exascale. HPX departs from today's prevalent bulk-synchronous parallel programming models with the goal of mitigating traditional limitations, such as implicit and explicit (global and local) barriers, coarse grain parallelism, and lack of easily achievable overlap between computation and communication.

By modelling the API after the interfaces defined by the C++ Standards, programmers are enabled to write fully asynchronous code using hundreds of millions of threads. This ease of programming extends to both parallel and distributed applications. In addition, the implementation adheres to the programming guidelines and code quality requirements defined by the Boost collection of C++ libraries [1]. HPX is the first open source runtime system implementing the concepts of the ParalleX execution model [12], [19] on conventional systems.

Although HPX is designed to efficiently provide low level system services, various higher-level frameworks have been developed to assist programmers. Within HPX, a comprehensive set of parallel algorithms, executors, and distributed data structures have been developed–all of which are fully conforming to current C++ standardization documents or ongoing standardization work. These constructs allow developers to simply express and direct parallelism in a scalable way. HPX also provides a uniform, flexible, and extensible performance counter framework, described in Section IV.

External to HPX, libraries have been developed which provide additional functionality and extend the HPX paradigm beyond CPU computing, notably APEX [11], [10]. APEX, an introspection and adaptivity library, takes advantage of the HPX performance counter framework to gather arbitrary knowledge about the system and uses the information to make runtime-adaptive decisions based on user defined policies.

IV. THE HPX PERFORMANCE COUNTER FRAMEWORK

Performance Counters in HPX are used to provide information as to how well the runtime system or an application is performing. The counter data can help determine system bottlenecks and fine-tune system and application performance. The HPX runtime system, its networking, and other layers provide counter data that an application can consume to provide users with information of how well the application is performing. The exposed performance counters are not limited to hardware counters (as for instance provided by PAPI [6]), but can cover any numeric information interesting to the user of the application.

Applications can also use counter data to determine how much system resources to consume. For example, an application that transfers data over the network could consume counter data from a network switch to determine how much data to transfer without competing for network bandwidth with other network traffic. The application could use the counter data to adjust its transfer rate as the bandwidth usage from other network traffic increases or decreases.

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

all counters expose their data using the same API, any code consuming counter data can be utilized to access arbitrary system information with minimal effort.

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

As a convenience, all HPX applications provide command line options related to performance counters, such as the ability to list available counter types, or periodically query specific counters to be printed to the screen or save them in a file. While the command line interface is easier to use, it is less flexible and does not allow an application to adjust its behavior at runtime. The command line interface provides a convenience abstraction for querying and logging performance counter data for a user specified set of performance counters.

V. Experimental Methodology

To demonstrate the capabilities of both the HPX runtime and the HPX performance counters, we ran strong scaling experiments for the Standard C++11 and HPX versions of the Inncabs benchmarks by increasing the number of cores used while keeping the total workload constant for each benchmark. This section describes the benchmarks, system configuration, performance counter measurements, and the methods used in running the experiments.

A. Benchmarks

For this work, we ported to HPX the Inncabs benchmark suite, introduced in [18] as a suite of benchmarks using the C++11 Standard constructs for thread parallelism. The benchmarks were previously ported to C++11 by Thoman, Gschwadtner, and Fahringer to assess the performance achieved by using the Standard C++ thread mechanisms for parallel applications without the support of third party libraries. The benchmarks have a variety of task granularity workloads and synchronization demands. Table V in the Results section shows the structures and synchronization of the benchmarks and includes the average task duration as measured using an HPX performance counter.

Since HPX's API is modeled after the C++ Standard, replacing the Standard task parallel structures with HPX equivalents for the Inncabs parallel benchmarks is very simple. In most cases this just involves changing the function's namespace (see Table II). As defined in the C++ Standard, the template function `std::async`:

> ...runs the function `f` asynchronously 'as if on a new thread' (potentially in a separate thread which may be part of a thread pool) and returns a `std::future` that will eventually hold the result of that function call. [2]

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

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

> ...a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. [3]

Sources and detailed descriptions of the Inncabs benchmarks are available in [2] as are the HPX ported versions [3].

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

TABLE II
Translation of Syntax to HPX

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

B. Configurations

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

TABLE III
Platform Specifications

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

We built the software using GNU C++ version 4.9.1, GNU libstdc++ version 20140908, and HPX version 0.9.11 (8417f14) [14]. We also ran tests using the clang compiler with results that were not significantly different from those compiled with GNU so are not presented in this paper.

[3]http://en.cppreference.com/w/cpp/thread/mutex

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

The original Inncabs benchmarks can be run with any of three launch policies (`async`, `deferred`, or `optional`) as specified by the user at runtime. HPX options includes these launch policies and a new policy, `fork`, added in version 0.9.11. The `fork` launch policy is equivalent to `async` except that it enforces continuation stealing instead of the (default) child stealing. This can result in performance improvements for strict fork/join use cases, where the future returned from `async` is guaranteed to be queried from the current thread. We compared performance of all launch policies for both Standard and HPX versions of the benchmarks and found the `async` policy provides the best performance, so we only present the results using the `async` policy.

C. Performance Counter Measurements

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

The HPX performance monitoring framework provides the flexibility to inspect events over any interval of the application. The Inncabs applications are written such that the computation samples are taken inside the application. We measure the performance counters just for the computation of each sample by using the HPX `evaluate` and `reset` API calls. HPX also includes the ability to monitor for predefined timed intervals and to measure events for each individual OS thread. For this paper we use the total of the OS thread counts.

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

Task Duration: is the value of the **/thread/time/average** counter which measures the average time spent in the execution of an HPX thread, also referred to as an HPX task. Task duration for runs using one core give us a measure of task granularity and are reported in Table V. When the number of cores is increased we observe an increase in task duration that indicates the execution is delayed due to contention for shared resources as illustrated in prior work [7].

Task Overhead: is the value of the **/thread/time/average-overhead** counter which measures the average scheduling cost to execute an HPX thread. We observed task overheads on the order of 50-100% of the task grain size for the very fine-grained applications.

Task Time: measures the cumulative execution time of HPX tasks during the selected interval using the **/thread/time/cumulative** counter. We divide Task Time by the number of cores and present task time per core to show the relation to the execution time of the application.

Scheduling Overhead: is the measurement of time spent in scheduling all HPX tasks using the **/thread/time/cumulative-overhead** counter. Scheduling overheads can be a major cost for the fine-grained applications because the cost of scheduling the tasks is large in comparison to the task execution time.

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

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

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

The overhead caused by collecting these counters is usually very small (within variability noise), but sometimes are up to 10% with very fine granularity tasks when run on one or two cores. When PAPI counters are queried this overhead can go up to 16% with very fine granularity.

D. Experiments

Based on the wide range of available possibilities, we performed a large number of experiments to determine the best configuration for the build and run policies that provide the best comparisons of the benchmarks. A synopsis is provided in Table IV.

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

TABLE IV
SOFTWARE BUILD AND RUN SPECIFICATIONS

Specification	STD C++11	HPX
Compiler	gcc	gcc
Memory Allocation	system (Alignment-tcmalloc)	tcmalloc
Launch Policy	async	async

We present the experiments that give a fair comparison between Standard C++ and HPX and are most relevant to the goals of this research, to illustrate the capabilities of the HPX runtime system for scheduling asynchronous tasks on parallel systems with the benefit of measuring intrinsic performance events.

To assess performance of the benchmarks, we use strong scaling by increasing the number of cores while keeping a fixed workload. The one exception to this is the Floorplan benchmark. The `std::async` implementation provides a single task queue from which all threads are scheduled. In comparison, the HPX implementation provides a local task queue for each OS thread in the thread pool. Because of this subtle difference, the two implementations executed the tasks in a different logical ordering, causing the `std::async` implementation to prune the search much earlier and converge on the solution much sooner. In fact, the HPX implementation ordering executed over two orders of magnitude greater results to arrive at the same solution. In that case, a fixed limit on the number of total tasks executed was enforced to ensure a fair comparison of the runtimes. The input sets used in the original Inncabs paper are used, with the exception of QAP, which exceeded memory limits. QAP only runs successfully using the smallest input set included with the original sources.

Execution time for each sample of each benchmark is measured. To maximize locality, we pin threads to cores such that the sockets are filled first. For the Standard version we use the command `taskset` and specify the proper cpu order to ensure proper affinity – which is tricky since logical core designations vary from system to system. HPX utilizes HWLOC and provides flexible thread affinity support through the `--hpx:bind` command line option. We verified that both versions were properly using thread affinity by monitoring test runs using the `htop` utility.

We took 20 samples for each experiment and present data from the medians of the samples for the execution times and the counters. To measure the counter data, we evaluate and reset the counters for each sample using the HPX `hpx::evaluate_active_counters` and `hpx::reset_active_counters` API functions.

VI. RESULTS

We ran experiments using the fourteen benchmarks from the Inncabs benchmark suite to illustrate the capability of performance monitoring measurements on a variety of benchmarks. Table V is an expansion of the table in [18] which shows the structure of the benchmarks. We add the measurement of Task Duration (task grain size) and classify the granularity according to our measurements of the HPX Task Duration performance counter when the benchmark is run on one core. Included are the scaling behaviors of both the Standard C++11 and HPX versions measured in our experiments. Even though the benchmarks have a variety of structures and synchronization, the most prominent factor on scaling behavior and overall performance for these task parallel benchmarks is task granularity. In several cases, the performance of the benchmarks are similar to each other so we present a cross section of results that represent each kind. First we present the execution times to compare the two runtime libraries and then the performance metrics.

The coarse-grained benchmarks, Alignment, SparseLU, and Round, exhibit good scaling behavior for both libraries. Fig. 1 shows the execution times for Alignment, and is a good representation of all three benchmarks. These benchmarks are all coarse-grained with task grain size ranging from ∼1ms to ∼10ms. Scheduling overheads for coarse-grained tasks are a small percentage of the task, and contention of resources are small compared to execution time which gives the application the ability to scale well for both libraries.

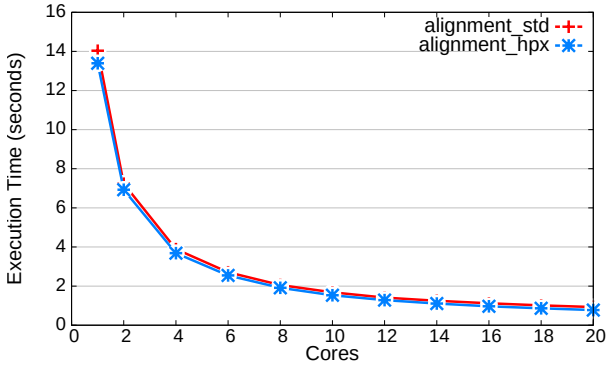

Fig. 1. Execution time of Alignment (HPX vs C++11 Standard) typifies the good scaling behavior of the coarse-grained benchmarks. The HPX results are slightly better than the results from running the Standard C++ benchmarks. This can be explained by HPX's reduced scheduling and context switching overhead.

Pyramids, Fig. 2, has a moderate grain size of ∼250µs and is the only application that executes faster for the Standard C++ version than HPX when run on more than one core. Although the Standard version runs faster until 14 cores, it has a speedup factor of 8 for 20 cores, while for HPX there is a speedup of 13, and the minimum execution times are equivalent.

Strassen, Sort, and NQueens classify as fine-grained benchmarks with task grain sizes of ∼100µs, ∼50µs and ∼25µs respectively. For each of these benchmarks, HPX shows the ability to scale well, while the Standard version either does not run (NQueens and some Strassen experiments) or only scales up to 10 cores like Sort. The behavior of the execution time of the fine-grained benchmarks are typified by that in Figures 3 and 4, although the baseline version of NQueens fails due to contention on resources for the rate pthreads are being scheduled.

TABLE V
BENCHMARK CLASSIFICATION AND GRANULARITY

Benchmark	Synchronization	Task Duration (average µs)	Granularity	Scaling Behavior Standard C++	HPX
Loop Like					
Alignment	none	2748	coarse	to 20	to 20 cores
Health	none	1.02	very fine	fail	to 10
Sparselu	none	980	coarse	to 20	to 20
Recursive Balanced					
FFT	none	1.03	variable/very fine	to 6	to 6
Fib	none	1.37	very fine	fail	to 10
Pyramids	none	246	moderate	to 20	to 20
Sort	none	52.1	variable/fine	to 10	to 16
Strassen	none	107	fine	(some fail) to 8	to 20
Recursive Unbalanced					
Floorplan	atomic pruning	4.60	very fine	to 10	to 10
NQueens	none	28.1	fine	fail	to 20
QAP	atomic pruning	1.00	very fine	to 6	to 4
UTS	none	1.37	very fine	fail	to 10
Co-dependent					
intersim	mult. mutex/task	3.46	very fine	no scaling	to 10
round	2 mutex/task	9671	coarse	to 20	to 20

Note: 'to x' means scaling (execution time improves) only occurs only up to x cores

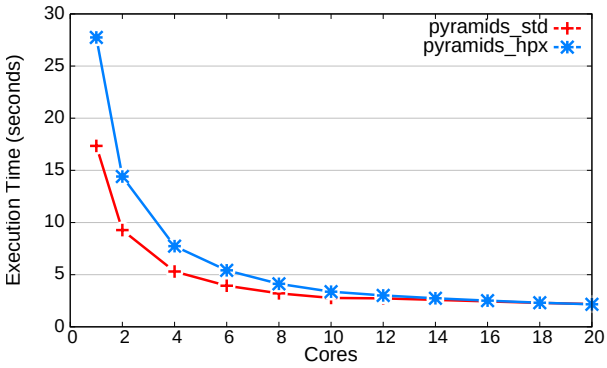

Fig. 2. Execution time of Pyramids (HPX vs C++11 Standard), Pyramids has a moderate grain size of ∼250µs and is the only benchmark that executes faster than HPX, in the low core counts. Execution times for 20 cores are equivalent.

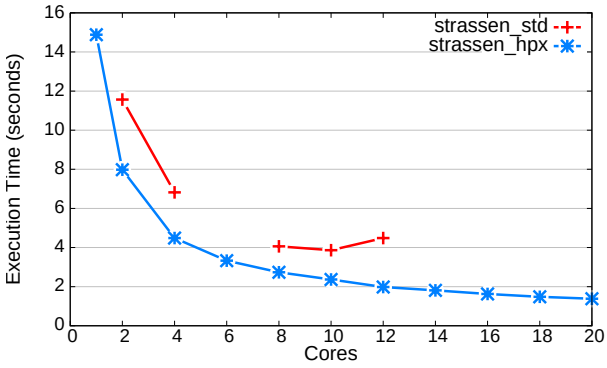

Fig. 3. Execution time of Strassen (HPX vs C++11 Standard), Strassen has ∼100µs grain size, scales well for HPX but does not run for some experiments for the Standard version. The efficiency of HPX's scheduling and context switching is even more pronounced with the fine-grain applications.

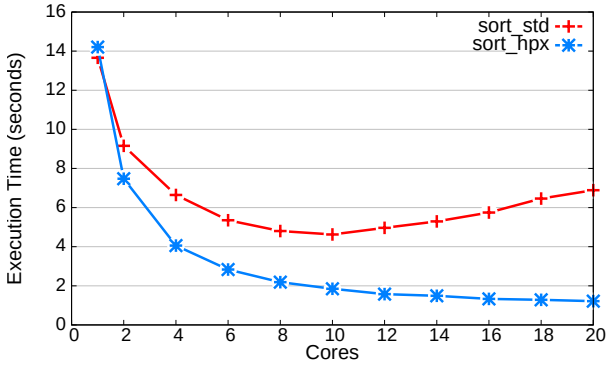

Fig. 4. Execution time of Sort (HPX vs C++11 Standard), has ∼50µs grain size, illustrates typical scaling behavior of fine-grained benchmarks. The efficiency of HPX's scheduling and context switching is even more pronounced with the fine-grain applications.

The remainder of the benchmarks are all classified as very fine-grained since they have task sizes less than ∼5µs. For HPX, we observe measurements of the task overhead performance counter from 0.5µs to 1µs for these benchmarks which means that scheduling overheads are a significant portion of the execution time. The Standard versions of NQueens, Health, Fib and UTS all fail due to contention on resources. For these we observe 80,000 to 97,000 pthreads launched by `std::async` just before failure. The available memory in the system is not sufficient to manage these quantities of pthreads. The very-fine grained benchmarks that do manage to complete, FFT, Intersim and Floorplan, scale poorly or not at all and have execution times much greater than the HPX versions. Figures 5, 6, and 7 illustrate these behaviors.

Figures 8 -12 illustrate the capability to use the performance

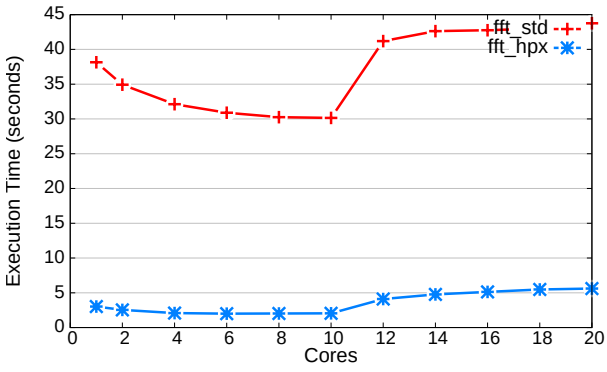

Fig. 5. Execution time of FFT (HPX vs C++11 Standard), with grain size ∼1µs is very fine-grained and shows limited scaling for HPX and much greater execution times for C++11 Standard. Scheduling and context switching costs of HPX are a larger percentage of the task size.

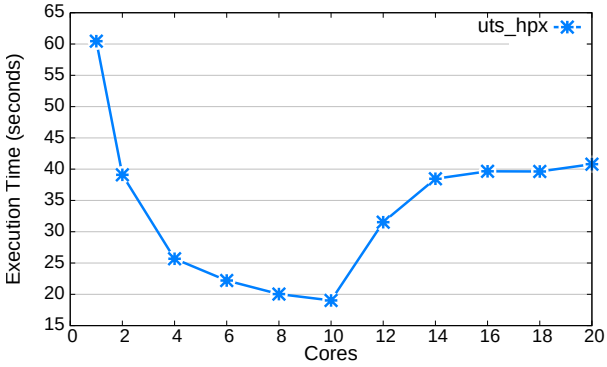

Fig. 6. Execution time of UTS (HPX vs C++11 Standard), with grain size ∼1µs is very fine-grained and exihibits some scaling for HPX until it reaches the socket boundary at 10 cores. The Standard version runs out of resources and fails due to the large number of pthreads scheduled.

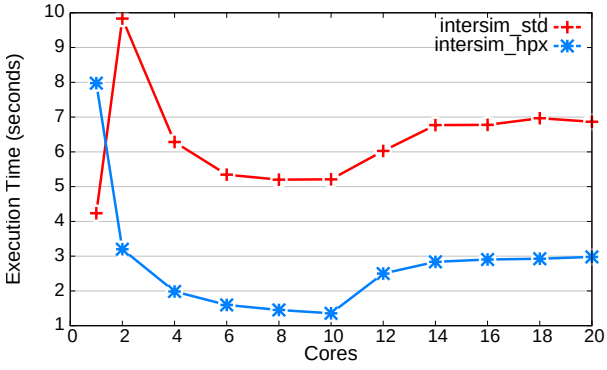

Fig. 7. Execution time of Intersim (HPX vs C++11 Standard), with grain size ∼3µs is very fine-grained and shows limited scaling for HPX and degredation for C++11 Standard. Scheduling and context switching costs of HPX are a larger percentage of the task size.

monitoring system to measure overheads and determine factors affecting the performance of the application. The metrics and associated counters are described in section V-C. We show the execution time of the benchmark (exec_time) and what the execution time with ideal scaling (ideal_scaling) would be for comparison, the task time per core with its associated ideal time, and the scheduling overheads (sched_overhd) per core. When the scheduling overheads are low, Figures 8, 10, and 9, as is the case for the fine- to coarse-grained benchmarks, the overall execution time of the benchmark is composed almost totally of the time spent in actual execution of the tasks. For Strassen the overheads are slightly larger than Alignment and this shows that the execution time does not scale as close to the ideal time. When running on 20 cores the speedup reaches a factor of 11 for Strassen, 13 for Pyramids and 17 for Alignment (compared to 1 core).

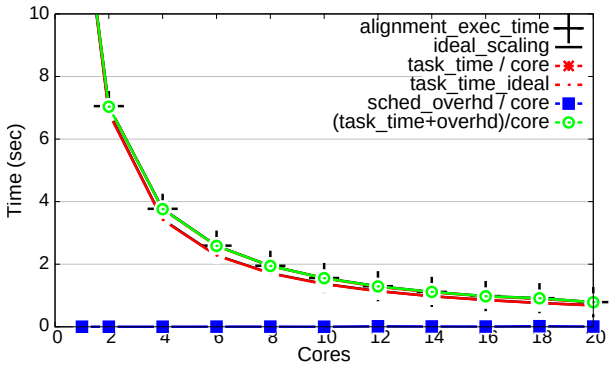

Fig. 8. Alignment Overheads, coarse-grained task size, has very small scheduling overhead and the task time is close to ideal so has good scaling behavior, speed up of 17 for 20 cores.

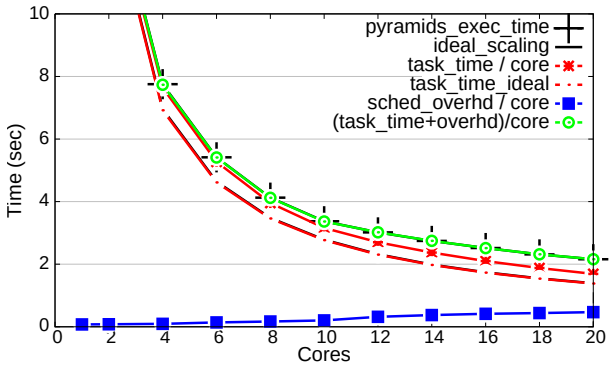

Fig. 9. Pyramids Overheads, moderate-grained task size, has slightly larger scheduling overheads than Alignment and the task time is larger than ideal. Speedup for 20 cores is 13.

The scheduling overheads have a larger effect on the overall execution time for applications that have smaller granularity. This is further demonstrated with the measurements from the very fine-grained benchmarks, Figures 11 and 12. The combination of smaller task size and larger number of tasks executed per second also puts pressure on system resources which causes a larger time in the actual execution of the

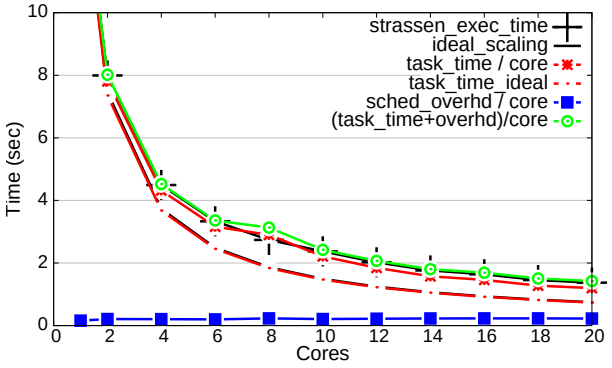

Fig. 10. Strassen Overheads, fine-grained task size, has small scheduling overheads but the gap between the ideal and actual task time is larger than for Pyramids, and the resulting speedup is 11 for 20 cores.

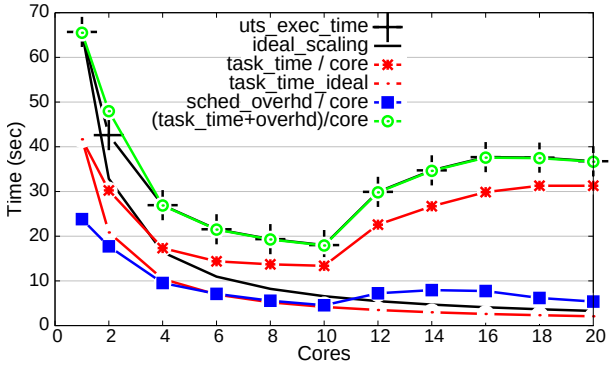

Fig. 12. UTS Overheads, very fine-grained, scheduling overheads are approximately 50% of the task time, after 4 cores task time is larger than ideal and increases after the socket boundary, resulting in poor scaling and increased execution time past the socket boundary.

task. The effects are unique to each benchmark and underlying architecture. For UTS, Fig. 12, the scheduling overhead is not as large as the increase in execution time caused by contention, while the opposite is true for FFT, Fig. 11. Increasing the number of cores that the benchmark is executed on also increases resource contention as seen by the growth of the gap between task execution time and its ideal. The jump in execution time from 10 cores to 12 cores is caused by crossing the socket boundary and thus the NUMA domains of the system. These metrics could be used to tune the grain size in parallel applications where that is feasible.

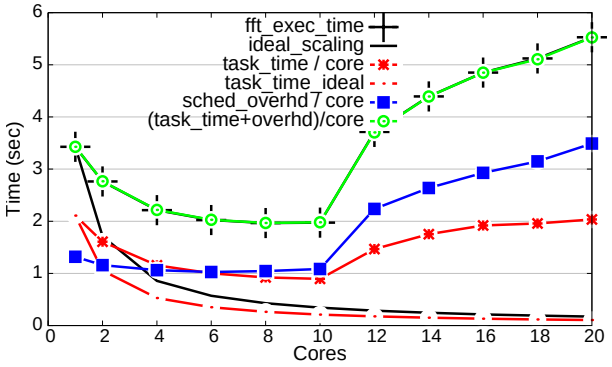

Fig. 11. FFT Overheads, very fine-grained, has scheduling overheads equivalent to the task time and both increase significantly beyond the socket boundary. This results in poor scaling and limits scaling to one socket.

Hardware counters can also be useful to monitor performance bottlenecks of the underlying system. One example would be the bandwidth measurement shown in Figures 13 - 16 as described in Sec. V-C. Bandwidth increases with the number of cores used until the socket boundary is passed for the very-fine grained benchmarks and the execution time makes a corresponding leap, while it continues to grow linearly for the benchmarks that scale well.

VII. Conclusion

We demonstrate the capabilities of the performance monitoring framework in the HPX runtime system to measure

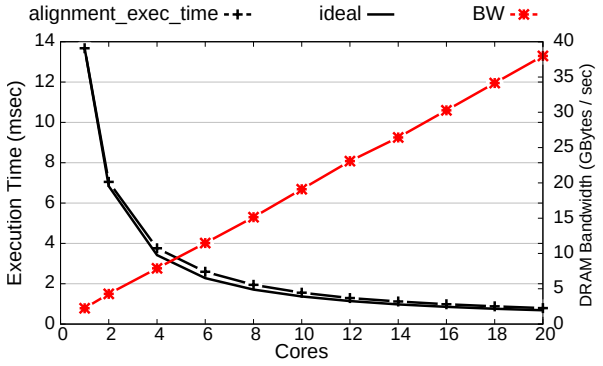

Fig. 13. Alignment OFFCORE Bandwidth, coarse-grained tasks.

intrinsic events that give detailed insight to the behavior of the application and the runtime system. The use of performance measurements give the ability to understand application efficiency and resource usage at runtime. We show the ease of porting the Inncabs parallel benchmark suite to HPX and the ensuing performance improvement of the benchmarks with fine-grained parallel tasks. We demonstrate that current external tools are not capable of supporting C++ task parallel constructs and do not support the runtime performance moni-

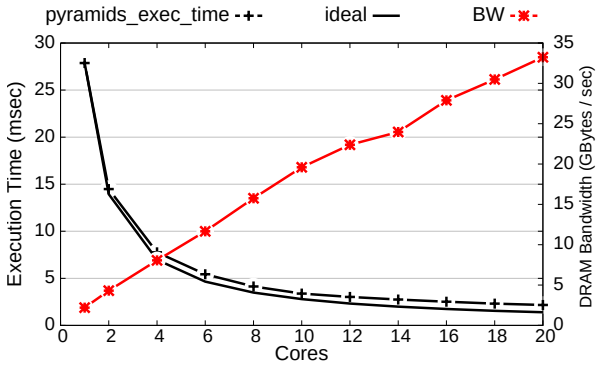

Fig. 14. Pyramids OFFCORE Bandwidth, moderate-grained tasks.

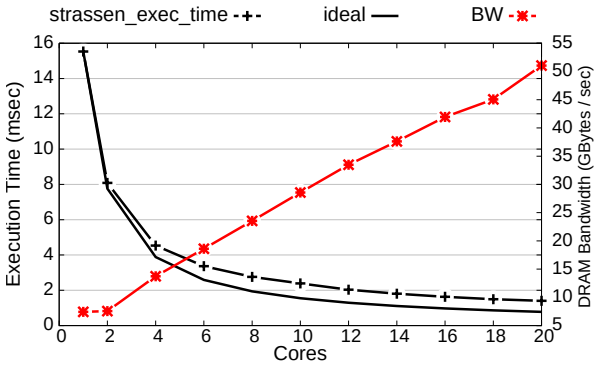

Fig. 15. Strassen OFFCORE Bandwidth, fine-grained tasks.

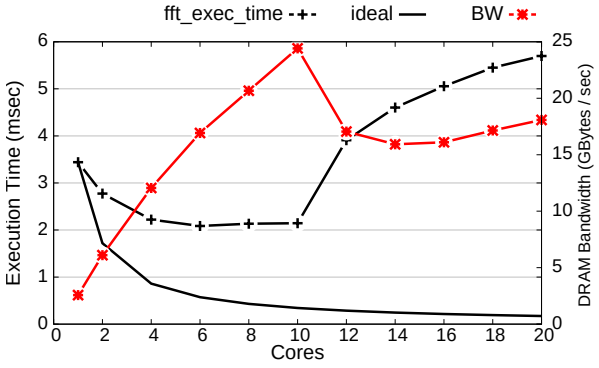

Fig. 16. FFT OFFCORE Bandwidth, very-fine grained tasks

toring that would be necessary for adaptive runtime decisions. The capabilities of the HPX runtime system presented it this paper pave a path towards runtime adaptivity.

As mentioned, the APEX library extends HPX functionality [10]. APEX includes a *Policy Engine* that executes performance analysis functions to enforce *policy rules*. By including guided search and auto-tuning libraries in the analysis functions, APEX has already demonstrated an emerging capability for runtime adaptation in HPX applications using the performance counter framework presented in this paper. In the future, we plan to use the information gained by this research with the capabilities of the APEX policy engine to provide additional runtime tuning for better performance, energy efficiency and application stability.

ACKNOWLEDGMENTS

This work is supported by NSF grant number CCF-111798.

REFERENCES

[1] Boost: a collection of free peer-reviewed portable C++ source libraries, 1998-2015. http://www.boost.org/.

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

[3] STEllAR-GROUP/inncabs, 2015. https://github.com/STEllAR-GROUP/inncabs.

[4] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, Apr. 2010.

[5] T. O. A. R. Board. Openmp application program interface v3.0, 2008.

[6] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for hardware performance monitoring on linux systems. In *International Conference on Linux Clusters: The HPC Revolution*, June 2001.

[7] P. Grubel, H. Kaiser, J. Cook, and A. Serio. The performance implication of task size for applications on the hpx runtime system. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 682–689, Sept 2015.

[8] 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.

[9] 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*, ScalA '13, pages 1:1–1:8, New York, NY, USA, 2013. ACM.

[10] K. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. Malony, T. Sterling, and R. Fowler. An autonomic performance environment for exascale. *Supercomputing frontiers and innovations*, 2(3), 2015.

[11] 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*, ROSS '13, pages 8:1–8:8, New York, NY, USA, 2013. ACM.

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

[13] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[14] H. Kaiser, T. Heller, A. Berge, and B. Adelstein-Lelbach. HPX V0.9.11: A general purpose C++ runtime system for parallel and distributed applications of any scale, 2015. http://github.com/STEllAR-GROUP/hpx.

[15] H. Kaiser, T. Heller, D. Bourgeois, and D. Fey. Higher-level parallelization for local and distributed asynchronous task-based programming. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*, ESPM '15, pages 29–37, New York, NY, USA, 2015. ACM.

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

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

[18] P. Thoman, P. Gschwandtner, and T. Fahringer. On the quality of implementation of the c++11 thread support library. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 94–98, March 2015.

[19] Thomas Sterling. ParalleX Execution Model V3.1, 2013. https://www.crest.iu.edu/projects/xpress/_media/public/parallex_v3-1_03182013.doc.