ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications

Hartmut Kaiser
Center for Computation and Technology
Louisiana State University
hkaiser@cct.lsu.edu

Maciej Brodowicz
Center for Computation and Technology
Louisiana State University
maciek@cct.lsu.edu

Thomas Sterling
Center for Computation and Technology
Department of Computer Science
Louisiana State University
tron@cct.lsu.edu

1. ABSTRACT

High performance computing (HPC) is experiencing a phase change with the challenges of programming and management of heterogeneous multicore systems architectures and large scale system configurations. It is estimated that by the end of the next decade Exaflops computing systems requiring hundreds of millions of cores demanding multi-billion-way parallelism with a power budget of 50 Gflops/watt may emerge. At the same time, there are many scaling-challenged applications that although taking many weeks to complete, cannot scale even to a thousand cores using conventional distributed programming models. This paper describes an experimental methodology, ParalleX, that addresses these challenges through a change in the fundamental model of parallel computation from that of the communicating sequential processes (e.g., MPI) to an innovative synthesis of concepts involving message-driven work-queue execution in the context of a global address space. The focus of this work is a new runtime system required to test, validate, and evaluate the use of ParalleX concepts for extreme scalability. This paper describes the ParalleX model and the HPX runtime system and discusses how both strategies contribute to the goal of extreme computing through dynamic asynchronous execution. The paper presents the first early experimental results of tests using a proof-of-concept runtime-system implementation. These results are very promising and are guiding future work towards a full scale parallel programming and runtime environment.

2. INTRODUCTION

An important class of parallel applications is emerging as scaling impaired. These are problems that require substantial execution time, sometimes exceeding a month, but which are unable to make effective use of more than a few hundred processors. One such example is numerical relativity used to model colliding neutron stars to simulate gamma ray bursts (GRB) and simultaneously identify the gravitational wave signature for detection with such massive instruments as LIGO (Laser Interferometer Gravitational Observatory). These codes exploit the efficiencies of Adaptive Mesh Refinement (AMR) algorithms to concentrate processing effort at the most active parts of the computation space at any one time. However, conventional parallel programming methods using MPI [1] and systems such as distributed memory MPPs and Linux clusters exhibit poor efficiency and constrained scalability, severely limiting scientific advancement. Many other applications exhibit similar properties. To achieve dramatic improvements for such problems and prepare them for exploitation of Petaflops systems comprising millions of cores, a new execution model and programming methodology is required [2]. This paper briefly presents such a model, ParalleX, and provides early results from an experimental implementation of the HPX runtime system that suggests the future promise of such a computing strategy.

It is recognized that technology trends have forced high performance system architectures into the new regime of heterogeneous multicore structures. With multicore becoming the new Moore's Law, performance advances even for general commercial applications are requiring parallelism in what was once the domain of purely sequential computing. In addition, accelerators including, but not limited, to GPUs are being applied for significant performance gains, at least for certain applications where inner loops exhibit numeric intensive operation on relatively small data. Future high end systems will integrate thousands of "nodes", each comprising many hundreds of cores by means of system area networks. Future applications like AMR algorithms will involve the processing of large time-varying graphs with embedded meta-data. Also of this general class are informatics problems that are of increasing importance for knowledge management and national security.

Critical bottlenecks to the effective use of new generation HPC systems include:

- *Starvation* – due to lack of usable application parallelism and means of managing it,
- *Overhead* – reduction to permit strong scalability, improve efficiency, and enable dynamic resource management,
- *Latency* – from remote access across system or to local memories,
- *Contention* – due to multicore chip I/O pins, memory banks, and system interconnects.

The ParalleX model has been devised to address these challenges by enabling a new computing dynamic through the application of message-driven computation in a global address space context with lightweight synchronization. This paper describes the ParalleX model through the syntax of the PXI API and presents a runtime system architecture that delivers the middleware mechanisms required to support the parallel execution, synchronization, resource allocation, and name space management. Section 3 describes the ParalleX model and the performance drivers motivating it. Section 4 describes the HPX runtime system architecture and particular implementation details important to the results. Section 5 describes early experiments that demonstrate key functional attributes of the runtime system. Section 6 presents and discusses the results, with concluding comments and future work presented in Section 7.

3. THE PARALLEX MODEL OF PARALLEL EXECUTION

A phase change in high performance computing is driven by significant advances in enabling technologies requiring new computer architectures to exploit the performance benefits of the technologies, while compensating for the challenges they present. Fundamental to such effective change, the architectures that reflect them, and the programming models that enable access and user exploitation of them is the transformative execution model that establishes the semantic framework tying all levels together. Historically, HPC has experienced at least five such phase changes in models of computation over the last six decades each enabling a successive technology generation. Currently significant advances in technology are forcing dramatic changes in their usage as reflected by early multicore heterogeneous architectures suggesting the need for new approaches to application programming.

The goal of the ParalleX model of computation is to address the key challenges of efficiency, scalability, sustained performance, and power consumption with respect to the limitations of conventional programming practices (e.g., MPI). ParalleX will improve efficiency by reducing average synchronization and scheduling overhead, improve utilization through asynchrony of work flow, and employ adaptive scheduling and routing to mitigate contention (e.g., memory bank conflicts). Scalability will be dramatically increased, at least for certain classes of problems, through data directed computing using message-driven computation and lightweight synchronization mechanisms that will exploit the parallelism intrinsic to dynamic directed graphs through their meta-data. As a consequence, sustained performance will be dramatically improved both in absolute terms through extended scalability for those applications currently constrained, and in relative terms due to enhanced efficiency achieved. Finally, power reductions will be achieved by reducing extraneous calculations and data movements. Speculative execution and speculative prefetching are largely eliminated while dynamic adaptive methods and multithreading in combination serve many of the purposes these conventionally provide.

ParalleX replaces the conventional communicating sequential processes model to address the challenges imposed by post-Petascale computing, multicore arrays, heterogeneous accelerators, and the specific class of dynamic graph problems while exploiting the opportunities of multithreaded multicore technologies and architectures. Instead of statically allocated processes using message-passing for communication and synchronization, dynamically scheduled multiple threads using message-driven mechanisms for moving the work to the data and local control objects for synchronization are employed. The result is a new dynamic based on local synchrony and global asynchronous operation. Key to the efficiency and latency hiding of ParalleX is the message-driven work-queue methodology of applying user tasks to physical processing resources. This separates the work from the resources and given sufficient parallelism allows the processor cores to continue to do useful work even in the presence of remote service requests and data accesses. This results in system-wide latency hiding intrinsic to the paradigm.

Global barriers are essentially eliminated as the principal means of synchronization, and instead replaced by lightweight Local Control Objects (LCOs) that can be used for a plethora of purposes from simple mutex constructs to sophisticated *futures* semantics for anonymous producer-consumer operation. LCOs enable event-driven thread creation and can support in-place data structure protection and on-the-fly scheduling. One consequence of this is the elimination of conventional critical sections with equivalent functionality achieved local to the data structure contended for by embedded local control objects and dynamic futures. The parallelism implicit in the topology of the directed graphs is exposed and exploited to address starvation by enabling more concurrency, dynamic scheduling, and adaptive

control. Local Control Objects may be embedded within the graph structures themselves guiding contention of concurrent actions on the same data helping to expose fine grained parallelism.

Unlike conventional distributed memory practices, ParalleX exhibits an *active global address space* (AGAS) that is an extension of experimental partitioned global address space (PGAS [3]). The key value of AGAS is that it allows virtual objects to be moved in physical space (across localities) without having to change the virtual name. In some applications, this is of no value as static distribution of data is sufficient. But where dynamic load balancing or dynamic directed graph problems are important to extreme scale application execution, then the ability to employ a more flexible shared global name management scheme can greatly simplify and make more efficient resource allocation over time.

ParalleX establishes a new relationship between virtual processes and the physical processing resources. Conventional practices assign a given process to a specified processor (or core). "Parallel processes" means multiple processes operating concurrently. ParalleX parallel processes incorporate substantial parallelism within a given process, map to multiple cores, and provide shared name space across multiple localities or nodes. A ParalleX parallel process can define a name space shared across several localities supporting many concurrent threads and child processes. It allows application modules to be defined with a shared name space and exploit many layers of parallelism within the same context.

ParalleX will support the phase change in HPC that is required to extend the value of future technology trends as embodied in the increase in the number of cores and variability of core structures and ISAs. Multicore is the new Moore's Law. Post-Petascale computing towards Exascale will demand millions of cores and hundreds of million-way parallelism. New classes of applications based on dynamic graph structures from Science, Technology, Engineering and Mathematics (STEM) to informatics problems will require effective execution in form and functionality very different from conventional vector-like problems (for typical problems). ParalleX provides semantics and mechanisms that will support multicore systems performing graph-based problems for extremes in scalability

4. HPX

The HPX (High Performance ParalleX) runtime system presented in this paper leverages the experience obtained from developing earlier versions of runtime systems for parallel execution models. This implementation in C++ represents a first attempt to develop a comprehensive API for a parallel runtime system supporting the ParalleX model. It provides a target for the development and implementation of the PXI specification, which is meant to define a low level API for ParalleX applications, very much as MPI [1] defines an API for communicating sequential processes. The implementation has a number of key features, described later in detail:

- It is a modular, feature-complete, and performance oriented representation of the ParalleX model targeted at conventional architectures and, currently, Linux based systems.
- Its modular architecture allows for easy compile time customization and minimizes the runtime memory footprint.
- It enables dynamically loaded application-specific modules to extend the available functionality, at runtime. Static pre-binding at link time is also supported.
- Its strict adherence to Standard-C++ [4] and the utilization of Boost [5] enables it to combine powerful compile time optimization techniques and optimal code generation with excellent portability and a modular library structure.
- It has been designed for distributed applications handling very large dynamic graphs.

We designed HPX as a runtime system providing an alternative to conventional computation models, such as MPI, while attempting to overcome limitations: such as global barriers, insufficient and too coarse grained parallelism, and poor latency hiding capabilities. The ParalleX model is intrinsically latency hiding, delivering an abundance of parallelism within a hierarchical distributed global shared name-space environment. This allows HPX to provide a multi-threaded, message-driven, split-phase transaction, non-cache coherent distributed shared memory programming model using futures based synchronization on large distributed system architectures.

4.1 General Design

The implementation level requirements of the HPX library as described in the previous section directly motivate a number of design objectives. Our most important objective was to design a state-of-the-art parallel runtime system providing a solid foundation for ParalleX applications while remaining as efficient, as portable, and as modular as possible. This efficiency and modularity of the implementation, is then central to the design, and dominates the overall architecture of the library (see Figure 1).

Figure 1 shows a block diagram of our HPX implementation. It exposes the necessary modules and an API to create, manage, connect, and delete any ParalleX parts from an application; it is generally responsible for re-

source management. The current implementation of HPX provides the infrastructure for the following ParalleX concepts:

- AGAS (active global address space)
- Parcel transport and parcel management
- Threads and thread management
- LCO's (local control objects), and
- Parallel Processes.

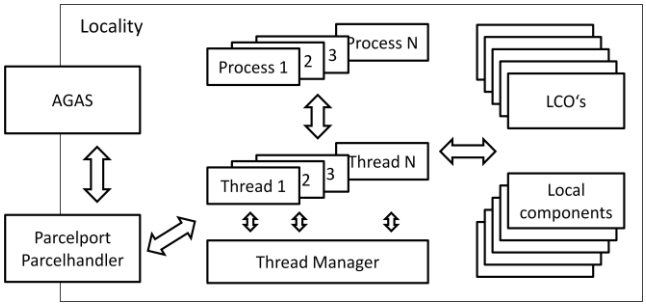

Figure 1: Modular structure of the current HPX implementation. HPX implements the supporting functionality for all of the elements needed for the ParalleX model: AGAS (active global address space), parcel port and parcel handlers, thread manager, ParalleX threads, ParalleX processes, LCO's (local control objects), and the means of integrating application specific components.

The following sections will describe design considerations and implementation specific details for those elements.

4.2 AGAS – The Active Global Address Space

The requirements for dynamic load balancing and the support for dynamic graph related problems define the necessity for a single global address space across the system. The abstraction of localities is introduced as a means of defining a border between controlled synchronous (intra-locality) and fully asynchronous (inter-locality) operations. A locality is a contiguous physical domain, managing intra-locality latencies, while guaranteeing compound atomic operations on local state. Different localities may expose diverse temporal locality attributes. Our implementation interprets a locality to be equivalent to a node in a conventional system.

Everything in ParalleX needs to be movable to a different locality without ever changing its name. At the same time, conventional SMP systems often don't have a uniform address space. The Partitioned Global Address Space [6] as a current attempt to solve this problem lacks the global immutability of names. The Active Global Address Space (AGAS) assigns global names (ids, unstructured 128 bit integers) to all entities managed by HPX. It provides a means of resolving these global ids into the corresponding local virtual addresses (LVA's), while assuming no coherence between localities. LVA's consist of the locality id, the type of the entity, and its local memory address. Moving an entity to a different locality updates this mapping, keeping all references to the moved item valid.

Our current implementation is based on centralized server/client architecture. As measurements showed this to be a bottleneck for larger numbers of localities we implemented a local caching policy minimizing the number of required network roundtrips as much as possible. The current implementation allows the creation of the globally unique id's autonomously in the locality where the entity is created, avoiding additional overhead. The modular system architecture will make possible the future replacement of the server/client architecture with a more scalable solution without touching any of the other parts of the runtime system.

4.3 Parcel Transport and Management

Any inter-locality messaging is based on Parcels. In ParalleX, parcels are an extended form of active messages. They consist of 4 parts: the global address of their destination, the action to perform, the arguments to pass on to the invoked action, and a continuation, which is a list of local control objects to be triggered after the action is executed. The continuation implements distributed flow control as it specifies the sequence of operations across localities.

Parcels are generated and sent whenever an operation has to be applied on a remote locality and are either used to move the work to the data or to gather small pieces of data back to the caller. Parcels enable message passing for distributed control flow and dynamic resource management, implementing a split phase transaction based execution model.

In the HPX implementation, parcels are represented as polymorphic C++ objects. They are serialized and deserialized using a sophisticated serialization scheme, binding the actions to execute at compile time. This highly optimal implementation removes the need to look up the function to invoke based on the action description stored in the parcel.

Parcels are sent to their destination using the implemented parcel transport layer. Currently, it establishes asynchronous P2P connections between the source and destination localities. The parcel transport layer not only buffers incoming and outgoing parcels, but it automatically resolves the global destination address of a parcel to send to the id of the destination locality. Additionally it either dispatches the incoming parcels to the local thread manager or forwards them in case the destination entity has been moved to a different locality.

While HPX is currently based on TCP/IP, we will improve the performance of the parcel transport layer by incorporating existing high performance messaging libraries, such as GASNet [7] and Converse [8].

4.4 PX-Threads and their Management

PX-threads are first class HPX objects in the sense that they have an immutable name enabling their management, even from remote localities. Although theoretically possible, we avoid moving threads to different localities. The preferred method is to send a parcel which creates a new thread continuing to work on the task at hand. We believe this scheme to be more efficient, especially on heterogeneous systems where moving threads is a very expensive operation. PX-threads maintain a thread state, an execution frame, and a (operation specific) set of registers. Allowed thread states are: *pending*, *running*, *suspended,* and *terminated*.

PX-threads are implemented as user level threads. These are cooperatively (non-preemptively) scheduled in user mode by the thread manager on top of one operating systems thread (e.g., Pthread) per core. The threads can be scheduled without a kernel transition, which provides a performance boost. Additionally the full use of the OS's time quantum per OS-thread can be achieved even if a PX-thread blocks for any reason. The scheduler is cooperative in the sense that it will not preempt a running PX-thread until it finishes execution or that thread cooperatively yields its execution on behalf of other tasks. This is particularly important since it avoids context switches and cache thrashing due to randomization introduced by preemption.

The thread manager is currently implemented as a 'First Come First Served' scheduler, where all OS threads work from a single (global) queue of tasks. Measurements showed this to be sufficient for a relatively small amount of concurrent OS threads, but the contention arising during push/pop operations on this queue quickly start to limit scalability. We are working on a more scalable work stealing scheduler using one queue per OS thread (core) combined with a single global queue (very similar to Intel's Thread Building Blocks [9], CILK++ [10], or Microsoft's Concurrency Runtime [11]). At creation time the thread manager captures the machine topology and is parameterized with the number of resources it should use, the number of OS threads mapped to its allocated resources, its resource allocation policy, and whether it should give priority to execute threads to increase cache hits or improve fairness across threads. By default it will use all available cores and will create one static OS-thread per core.

The thread manager additionally keeps a map of all existing PX-threads of a locality. It exposes a set of functions to query and manipulate any of the attributes of the PX-threads.

4.5 LCO's – Local Control Objects

A local control object is an abstraction of a multitude of different functionalities for event driven PX-thread creation, protection of data structures from inconsistent concurrent access (race conditions) and automatic on-the-fly scheduling of work with the goal of letting every single computation strand proceed as far as possible. Every object which may create (or reactivate) a PX-thread as a result of receiving a parcel from a remote locality or of being invoked locally exposes the necessary functionality of a LCO.

As mentioned in section 4.3, LCO's are being used to organize flow control. Parcel continuations are implemented as lists of LCO's enabling remote thread creation/reactivation combined with the delivery of a result or an error condition after the remote operation is finished. Figure 2 depicts a simple usage of continuations implementing the remote creation/reactivation and synchronization of two PX-threads using an eager future (a future refers to an object that acts as a proxy for a result that is initially not known, usually because the computation of its value has not yet completed) [12], [13].

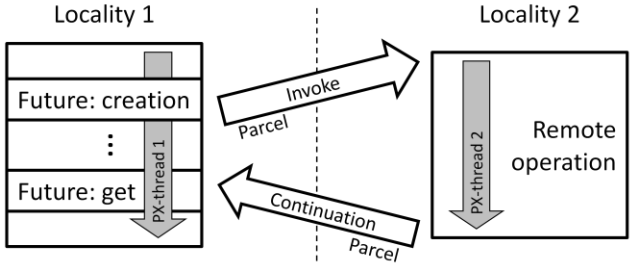

Figure 2: Schematic Flow Diagram of a Future. At the point of its creation the future sends a parcel to initiate the remote operation. This parcel carries the id of the future itself as its continuation. As the result of the remote operation is automatically sent to the specified continuations the future simply waits for the result in its get() function.

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

The thread manager can then make a scheduling decision to run other work. PX-threads require these special, cooperative synchronization primitives as it is not possible to use existing synchronization primitives as provided by the operating system because these are designed for OS-threads. While PX-threads are user lev-

el threads running on top of OS-threads, blocking on such a primitive stalls the whole OS-thread, inhibiting other PX-threads to be executed while waiting. We will explain the usage of other LCO's in section 5: dataflow templates. It is interesting to note that suspended PX-threads are LCO's as well, as they "produce" an active thread when triggered (reactivated).

5. EXPERIMENTS

This section describes the setup of two experiments conducted for this paper. First, we compared results of Fibonacci number calculations using different programming environments. Second, we compared the runtime behavior of a synthetic MPI program doing single-level evolution on a one-dimensional periodic grid with an equivalent program written using HPX.

The implementation of the Fibonacci calculation uses a recursive algorithm expecting positive numbers. All written test programs (using Java native threads, Pthreads, and PX-threads) instantiate one thread per invocation of the function `fib()` (see Figure 4), while joining the threads before returning the result value.

```
int fib(int x)
{
    if (n == 1 || n == 2)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Figure 4: Algorithm of used Fibonacci calculation. Despite its poor performance the recursive Fibonacci algorithm has been chosen because the code is trivial and allows performing good analysis of the respective overheads introduced by the different programming models.

To evaluate the efficiency of HPX in dealing with asymmetric parallel workloads (such as those characterizing GRB code), a synthetic MPI program was developed to perform a single-level evolution on a one-dimensional periodic grid, mirroring typical programming practices adopted by the scientific computing community. This application equally distributes the grid points across the available MPI processes and assigns a predefined amount of work to each fixed-sized zone grouping adjacent points. The amount of work in each zone is distributed randomly; with mean and standard deviation specified by the user (a uniform probability distribution function was used). It needs to be emphasized, that such a model is only a vastly simplified representation of real-world astrophysics simulations, in which 10-level AMR in three dimensions could produce as much as 16^{10} times the computational effort per data point of that performed at the base grid level. At the end of each time step, the MPI program performs a boundary update involving explicit message passing between the processes, acting effectively as a barrier. Special care has been taken to assure that the cumulative workloads in both MPI and HPX implementation discussed below are identical despite the randomization.

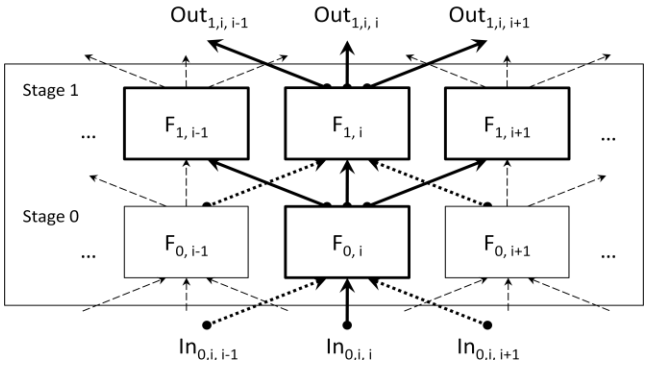

Figure 3: Schematic structure of a two stage static dataflow graph used for one dimensional adaptive mesh refinement (AMR) calculations. Each of the functional elements F corresponds to a single data point at a certain time step. Each functional element takes the 3 outputs generated at the previous time step by the adjacent elements, calculates the current time step, and provides the value to the adjacent functional elements responsible for the next time step. The outputs $Out_{1,i,i-1}$, $Out_{1,i,i}$, and $Out_{1,i,i+1}$, are connected to the inputs $I_{0,i-1,i-1}$, $I_{0,i,i}$, and $I_{0,i+1,i+1}$, forming a closed execution loop. Adaptive mesh refinement is achieved by instantiating a new static dataflow graph replacing several functional elements with a finer resolution graph.

The HPX test program uses a completely different approach of representing an equivalent data grid. Each data point is implemented as a separate dataflow template having 3 inputs and 3 outputs, where the inputs are connected to outputs of adjacent data points in a way mirroring the required data transfer between data points after each time step. Figure 3 shows a schematic overview of the layout. The implemented two stage pipeline is configured such that the outputs of stage 1 are connected back to the inputs of stage 0, minimizing the required amount of dataflow templates to 2N (with N being the number of data points). This approach completely avoids global barriers as each data point can proceed as soon as the adjacent previous data points calculated their result values.

6. RESULTS

For the sake of consistency, all experiments were performed on a dedicated dual-Opteron workstation containing 4 execution cores and 8 GB of memory. Note that this configuration sensibly limits the number of the underlying OS-threads executing the concurrent work-

loads in HPX to the maximum of four. The performance data were collected using multithreaded programs calculating the Fibonacci sequence written in HPX, Java, and Linux Pthreads (NPTL). While both HPX and Java applications used their respective implementations of futures to handle the asynchrony, the Pthreads code relied on explicit child thread joins to extract the result of lower level calculations, thus preserving the tree-like structure of computation.

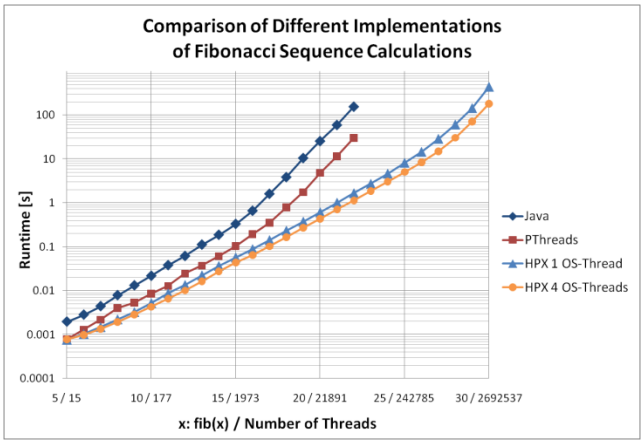

Figure 6: Runtimes for different implementations of the Fibonacci sequence. This figure depicts the dependency of the wallclock time (logarithmic scale) needed for calculation of a Fibonacci number using equivalent implementations based on different programming environments: HPX, Java, and Pthreads.

The graph of execution time shown in Figure 6 was obtained starting with the argument value of 5 and continuing until the system exhausted its physical memory (HPX), or application crashes started to occur due to violation of preset physical resource limits (Java, Pthreads). As can be seen, HPX user-level threads consistently outperform the other two implementations because of much lower overhead of thread creation and context switches, even when executing only on top of one OS-thread. Moreover, the per-thread memory requirements of HPX are substantially lower, shifting the "knee" of the performance curve towards greater argument values. This means that the point at which a substantial fraction of the working set is relegated to the main memory (as opposed to running mostly from caches) occurs for much higher number of concurrent threads in HPX, indicating greater scalability. However, HPX still has a room for improvement, as the speedup obtained with four versus one worker OS thread was smaller than 4, which suggests some scheduling inefficiencies.

The graph of the execution times comparing the MPI and HPX AMR programs collected for 50 iterations on 100 grid points and normalized to unit mean workload

Figure 5: Runtimes for equivalent MPI and HPX test model runs. This figure shows the dependency between the overall runtime of a test model run and the standard deviation of a non-uniform workload executed at each of the data points of the data grid.

per point, is shown in Figure 5. Normalization was done to eliminate the skewing of results by the communication and scheduling overheads, as the amount of work per point in a single time step was substantial (few tens of milliseconds). As expected, the performance of both applications is identical when the standard deviation is zero (MPI processes don't have to wait for each other in the update phase, as they execute the same amount of work). The situation changes radically when workload variation is introduced; since the increase in execution time of the busiest MPI process in each of the iterations is proportional to the standard deviation of the workload, the performance of the MPI application drops linearly with the standard deviation. The HPX implementation performed evenly in all cases (the run times remained constant to two significant digits), which proves the validity of ParalleX asynchronous execution model for this class of applications.

7. FUTURE WORK

This paper presents results drawn from using an early version of HPX. Our future work will concentrate on implementing ParalleX processes. These will be first class HPX objects defining a distributed, not necessarily contiguous execution environment. They are used to specify a broad task while transcending multiple localities and providing a namespace for the execution of threads.

As a parallel effort, we currently work on a specification of a ParalleX API (called PXI), which will define language bindings for C, Fortran and C++ for ParalleX ap-

plications very much like MPI specifies an API for communicating sequential processes. Implementation wise, this environment will comprise a preprocessor and a set of support libraries referred to in combination as PXlib that will permit the integration of PXI/C codes with the existing HPX runtime system, native C compilers, and Unix (including Linux) operating system calls.

8. ACKNOWLEDGMENTS

We would like to acknowledge the sponsorship in the context of different programs from DoE, NSF, and DoD. Additionally we would like to thank Steve Brandt for the constructive discussions and for helping with the performance measurements for the Fibonacci example.

9. REFERENCES

1 Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, V2.1. [Internet]. 2008 Available from: http://www.mpi-forum.org/docs/mpi21-report.pdf.

2 Kogge P, Bergman K, Borkar S, Campbell D, Carlson W, Dally W, Denneau M, Franzon P, Harrod W, Hil K, et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. DARPA IPTO; 2008.

3 Yelick K, Bonachea D, Chen WY, Colella P, Datta K, Duell J, Graham SL, Hargrove P, Hilfinger P, Husbands P, et al. Productivity and performance using partitioned global address space languages. In: International Conference on Symbolic and Algebraic Computation, Proceedings of the 2007 international workshop on Parallel symbolic computation; 2007; London, Ontario, Canada.

4 The C++ Standards Committee. Working Draft, Standard for Programming Language C++. [Internet]. 2008 Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2798.pdf.

5 Boost: a collection of free peer-reviewed portable C++ source libraries. [Internet]. 2009 Available from: http://www.boost.org/.

6 UPCConsortium. UPC Language Specifications, v1.2. Lawrence Berkeley National Lab; 2005 October. Tech Report LBNL-59208. Available from: http://upc.gwu.edu.

7 Bonachea D. GASNet Specification. 2002. U.C. Berkeley Tech Report (UCB/CSD-02-1207). Available from: http://gasnet.cs.berkeley.edu/.

8 Department of Computer Science, University of Illinois at Urbana Champaign. CONVERSE Programming Manual. [Internet]. 2009 Available from: http://charm.cs.uiuc.edu/manuals/html/converse/manual-1p.html.

9 TBB: Intel® Thread Building Blocks. [Internet]. 2009 Available from: http://www.threadingbuildingblocks.org/.

10 CILK++: Parallelism for the Masses. [Internet]. 2009 Available from: http://www.cilk.com/.

11 Microsoft Parallel Computing Platform Team. The Concurrency Runtime: Fine Grained Parallelism for C++. [Internet]. 2009 Available from: http://channel9.msdn.com/posts/Charles/The-Concurrency-Runtime-Fine-Grained-Parallelism-for-C/.

12 Friedman D. CONS should not evaluate its arguments. Automata, Languages and Programming. 1976 257-284.

13 Baker H. The Incremental Garbage Collection of Processes. In: Proceedings of the Symposium on Artificial Intelligence Programming Languages, SIGPLAN Notices 12; 1977.