Position paper: Towards a codelet-based runtime for exascale computing

Christopher Lauderdale
Rishi Khan
ET International, Inc.
{clauder,rishi}@etinternational.com

ABSTRACT

Computing systems have reached the performance limits attainable by increasing clock rates and complexity, and are now using increased thread-level parallelism and heterogeneity instead. Existing software typically deals poorly with large-scale or heterogeneous computer systems, relying on multiple poorly interacting or special-purpose software interfaces to approach scaling/heterogeneity; attempting to use these approaches on future computers (especially at exascale) will only make matters worse.

To solve this problem, software components may be broken up into pieces, called "codelets", that can be dynamically scheduled without reliance on binding to any one thread or hardware component in relation to other codelets. By placing a supporting runtime layer between the system interface and codelet-based application, a very high degree of parallelism can be exposed, scaled, and scheduled to use the available hardware efficiently and intelligently. A codelet-based runtime can be used to create a clear path forward to exascale, as well as to deal with computing challenges in the interim.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming

Keywords

exascale, many-core, parallel, program execution model, dynamic, adaptive, runtime, codelet

1. INTRODUCTION

1.1 Motivation

Present-day computer systems have reached limits in the performance attainable using coarse-grained parallelism. Due to fundamental physical limitations, increased CPU clock rate is no longer the primary means by which to boost performance. Processors can extract instruction-level parallelism to help overcome this limitation [13], but this requires increased chip complexity, which increases power usage and reduces available chip space for other components. Contemporary computers are thus coming to rely more heavily on thread-level parallelism, using a larger number of simpler, lower-power cores to replace fewer complex, higher-power ones. As core counts increase, small on-chip local memories, whether explicitly addressable or hardware-managed, have also become necessary to reduce the latency and power usage otherwise incurred by large numbers of cores accessing shared system memory. To reach exascale, these local memories will need to be less coherent or under stricter software control than most caches in current use [16].

Heterogeneity is also increasingly common in computers, as architectures with differing capabilities and requirements (*e.g.*, CPUs, GPUs/GPGPUs, and FPGAs) are mixed together more freely. Effectively utilizing and coordinating disparate components is a difficult matter, and typically involves use of special APIs for non-CPU components. Dealing with dynamic loads on these components that arise during parallel computation is also difficult, since work must typically have been statically partitioned in order to use the components in the first place.

Unfortunately, contemporary software is generally ill-equipped to deal well with the hardware features of today's computers and even more poorly equipped to deal with exascale parallelism and memory structure, due in large part to reliance on traditional sequential processing and coherent memory models. Interfaces like OpenMP [7] can ease a transition to multithreading, but software threads require enough memory, management overhead, and centralization that they may not be practical in their present form as larger-scale systems become prevalent. Interfaces like MPI [11] and SHMEM [4] make it possible to coordinate explicit data transfer across a cluster, but make it difficult to deal well with the dynamic cross-cluster load presented by many programs, leading to under- or mis-utilization of computing resources. Furthermore, inter-process/-node APIs like MPI interact poorly (or not at all) with multithreading interfaces, forcing threads, like nodes, to statically partition their workloads.

1.2 Proposed solution

One of the problems underlying the above is that it can be difficult to expose enough parallelism in load-imbalanced programs to keep computing resources busy. This stems partly from the overhead inherent in creating and switch-

EXADAPT '12 March 3, 2012, London, UK.

ing between threads; typically the operating system (OS) and/or several layers of system library must be involved, limiting the potential benefits of thread use in handling operations of unknown/unpredictable duration. In addition, neither the OS nor threading software can accurately observe or react to the small-scale behavior of a long-running software thread without either imposing unacceptable overhead or undue hardship on software programmers. Interposition of a software runtime layer between the OS and application allows the application to quickly and easily expose parallelism and while the runtime manages scheduling and placement of application components [9].

An execution model that allows an application to describe finer-grained units of work will also be necessary, so that as much of an application as is ready to run can do so as soon as possible. ***Codelets***, the fundamental unit of work considered in the model described by this paper, are small pieces of an application that can run to completion without blocking, and that explicitly suspend and resume execution if necessary to avoid running indefinitely.

One of the goals of a software runtime for exascale must be to minimize and hide long-latency memory accesses, which cause delays both from the accesses themselves and contention for bandwidth, and which increase power usage by engaging additional communications components. Codelets make it easier to coordinate access to distant and contentious memory so that application components can trigger as required data become available, rather than stalling threads during a long-latency accesses. (This allows the runtime system to step in where hardware caches would be used on few-core systems.) Using the runtime layer for nonlocal memory access also makes it possible to transparently route object accesses into other address spaces without (*e.g.*) hardware- and latency-intensive virtual memory tricks, and makes it possible for the runtime to intelligently place codelet execution based on the locations of input/output data.

An additional component of the solution to the challenges arising from exascale computing is the use of ***locales***, which give a high-level description of available hardware components and provide an interface whereby an application may schedule codelets to run, allocate memory, and exchange objects. Such an interface allows an application to specify precisely or generally where a codelet should run or where an object should reside in memory, when necessary.

SWARM (SWift Adaptive Runtime Machine) is an experimental codelet runtime that implements the ideas presented in this paper, with the aim of allowing applications to run well on single-, few-, or many-core computers, as well as allowing the application to transparently migrate across compute clusters or wider-area networks and between different kinds of computing hardware. This will allow applications to scale much more easily and widely, and makes a straightforward software path to exascale possible.

2. THE CODELET EXECUTION MODEL

2.1 Abstract machine model

In this paper, an application is assumed to be running on one or more compute nodes in a compute cluster, all of which have separate memory address spaces and communicate solely through a bus or network interconnect. Each node has one or more CPUs with some amount of DRAM shared amongst them and may have accelerators, coprocessors, or other peripherals attached, all connected via a node-wide bus. Each CPU chip or peripheral may have some amount of local addressable memory attached. A CPU chip contains one or more cores on it, each of which may time-multiplex its execution units amongst one or more hardware threads. Cores may be grouped into various levels of a communications hierarchy within the chip, and each core or group may have local memory associated with it. Cores within a group or CPU chip may be heterogeneous, as may a node's CPU chips. Figure 1 illustrates the relationships between hardware components in the machine model.

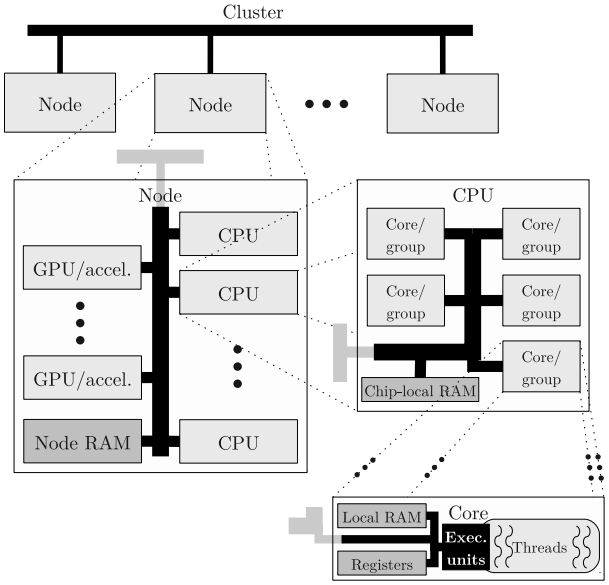

Figure 1: An abstract machine model for a codelet runtime.

For the purposes of this paper, more local memories are assumed to be smaller and have lower access latency than more remote memories. Caches and other non-addressable memories will not be considered, although they tend to follow the same pattern.

2.2 Abstract program model

2.2.1 Codelets

A codelet acts as the fundamental unit of scheduling and execution for a codelet runtime, and comprises the following:

- A ***run fork***, which describes work to be performed in order to advance the state of the program.
- An optional ***cancel fork***, which describes how to back out program state in case of an error.
- A description of the expected type of context frame (if any) used to store the codelet's state information.
- A description of the type of input data (if any) expected by the codelet.

Codelets are represented within a codelet runtime as small descriptor objects referencing run/cancel fork functions and context/input types.

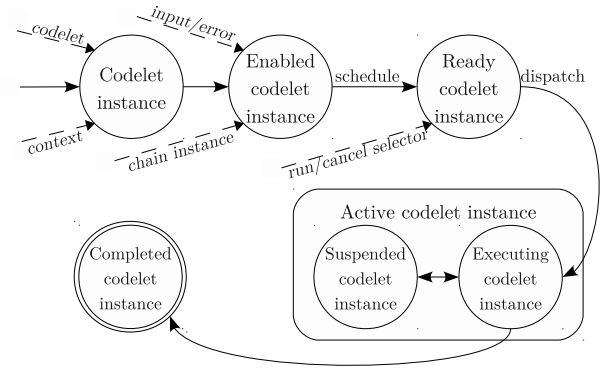

Figure 2: State relationships and transitions of a codelet instance.

By itself, a codelet is not ready to execute. A ***codelet instance*** may be created by associating a codelet with a context frame, and an ***enabled codelet instance*** additionally has associated input data for the run fork or error data for the cancel fork, as well as a ***chain instance*** which describes further work to be started once the codelet's work completes. The chain instance may be used to provide a communications path back to whatever started the codelet, and will be discussed in more detail in Sec. 2.2.2.

A ***ready codelet instance*** is an enabled instance that has been registered with a scheduler so that it may be dispatched and selected for execution, at which time the codelet instance will be considered ***active***. An active codelet executes until completion, barring special runtime or operating system measures taken to preempt or block it, although the runtime may allow it to explicitly place itself in a suspended state and let other codelets run in its place. Figure 2 shows the states a codelet instance may attain and the additional data associated with each state.

When executing, a codelet should not block, engage in long-latency operations, or run indefinitely. A codelet has exclusive use of its hardware thread during execution, and without runtime-/OS-layer preemption, any other codelets are unable to use that thread until the codelet completes. To perform a long-latency or blocking operation, a codelet should start it asynchronously and register a codelet instance to be executed upon completion; the operation may then be carried out in the background by other software/hardware layers while the codelet's thread is ceded to the runtime in the interim.

2.2.2 Codelet complexes and chaining

A ***codelet complex*** is an ad-hoc group of one or more codelets that work together to complete some task. These codelets will typically share a context, and may run sequentially or in parallel. To start a codelet complex, an ***initiator*** causes an ***entry codelet*** for the complex to be readied, passing the entry codelet appropriate context and input data, and optionally passing a chain instance referring back to the initiator's complex. If the complex involves multiple codelets, the entry codelet typically spills this information into a private context frame for later use; if the complex involves only one codelet, no private context is needed.

The chain instance passed into a codelet complex is typically used to effect communications with its initiator. A complex may ***chain*** by running or canceling the chain instance it was given at initiation. Running it (*i.e.,* causing execution of its run fork) indicates successful completion of the complex's operation; canceling it (*i.e.,* causing execution of its cancel fork) indicates that the operation resulted in an error. By convention, any chain instance passed into a complex must be run or canceled exactly once before the complex's execution completes.

Chaining has several important uses:

- Destruction of input/error parameters. Chaining provides a means by which a complex's initiator can release the memory associated with input/error data. (Chaining thus invalidates a complex's input/error parameter.)
- Return value passing. If a complex behaves like a subroutine, it can use chaining to indicate completion of the subroutine and pass a return value to its initiator via the input parameter. If the initiator needs to pass such a return value directly out to its own initiator, it can effect tail-call-like behavior by passing a higher-level chain instance into the complex it initiates.
- Bidirectional communication. Two complexes can enter into an arbitrarily long "conversation" by passing secondary chain instances during chaining. This is merely an extension of subroutine-like behavior, and allows producer-consumer and coroutine-like semantics to be implemented within the same framework. This also enables unification of inter-node and inter-complex communications.
- Extended error handling. If a faulting operation can be resumed, a secondary chain instance can be passed from a complex to its chain codelet's cancel fork; the secondary instance can be run or canceled to resume or abort the operation.

2.2.3 Interoperability between codelets and functions

Most present-day architectures use a program stack for storage of activation records, and although use of codelets does not necessitate use of a stack, neither does it conflict with stack use. For example, if run and cancel forks are implemented as functions, they can be called on the program stack by the runtime, and may in turn call other functions normally. The fork functions or their callees may also quickly suspend their execution and call back into the runtime scheduler, allowing other codelets' fork functions to run on the thread and use the remainder of the available stack space. Platform-native exception handling may also be used, although a codelet runtime should catch exceptions escaping from run/cancel forks so that they don't propagate into the rest of the runtime.

Section 2.2.2 noted that codelet complexes can be "called" much like subroutines; in addition, complexes can act as wrappers to integrate non-codelet-based functions into the codelet runtime, forwarding input data into a function as parameters and its return value out as chain input. Conversely, functions can be used to wrap access to codelet complexes by starting an entry codelet and suspending until the initiated complex runs a chain instance. If the chain instance is run normally, its input can be passed back to the suspended

codelet as a return value, and if canceled, the error can be passed back and thrown as a native exception.

The need to wrap codelets in functions and vice versa can be avoided entirely if a compiler is able to generate codelets as its normal output. In this case, functions in program code can be transparently formulated as codelet complexes in the generated binary, enabling existing code to be recompiled in codelet form and adapted more easily to a codelet runtime. Because codelet dispatch typically has higher overhead than native stack-based function calls, a compiler may also reformulate codelet dispatches as native function calls.

2.3 Locality awareness and management

All codelet execution occurs on a system component (typically a hardware thread) managed by the codelet runtime. These components can be grouped together based on the degree of information which can readily be shared between them; for example, two threads that share a memory might be grouped together, or all cores on a node. These groupings can be nested into a tree-like structure, or ***locale tree***, that describes the communication characteristics for software executing on a distributed platform.

Each locale in the tree has an associated allocator and scheduler, which respectively manage allocation of memory space and processing time within the locale. A ***leaf locale*** has no descendants, and its allocators and schedulers (or ***leaf allocators/schedulers***) manage space/time allocation within their described hardware or software structures only. Higher-level locales, which do have descendants, may manage their own allocation or their descendants'.

Because locales essentially represent bounded regions of a hardware/software platform, they provide an ideal interface for describing the transfer of data and control between those regions, including across boundaries for which ingress/egress communication normally requires special support (*e.g.*, network interfaces). Locales thus enable abstraction of disparate data transfer methods, including intra- and inter-node transfers.

The locale hierarchy facilitates inter-locale control transfer as well as data transfer, allowing a group of codelets executing within one locale to easily widen, narrow, or entirely transplant their execution scope, simply by specifiyng a different target locale for scheduling. This allows a program component's execution to be explicitly or implicitly forwarded to data upon which it needs to act, and allows the system to dynamically react to data availability and placement. This differs from most existing runtimes' capabilities, which focus primarily on data movement and make it difficult to cleanly migrate execution.

To enable codelets to easily control and inspect their position within the locale hierarchy, every active codelet instance has an ***environment buffer*** associated with it that describes the codelet's route and situation. The environment buffer contains a reference to the first, most recent, current, next, and final locale for a codelet's execution and communication, and these can be changed during execution by a codelet to modify its own routing; for example, when effecting a return via a chain instance, a codelet can reverse the route before scheduling the chain instance. A codelet complex can also use the "next locale" field of the route in a for-all loop to distribute execution across a group of locales, or can use the entire route to "walk" around the locale hierarchy collecting or updating local data, a practice especially useful for distributed graph problems (*e.g.*, Graph500, semantic web queries).

2.4 Dealing with heterogeneity

The above sections implicitly assumed that codelets have single forms of run or cancel forks that are executed when a codelet instance becomes active. However, if it is desired that a codelet be executable on disparate architectures or hardware components, multiple binary forms of the forks may be attached to the codelet's in-memory descriptor. In this case, the codelet runtime must select the appropriate binary form for the current hardware, or, if such a form can't be found, the codelet must be relocated and executed on a compatible component.

Each leaf locale has an associated hardware component description associated with it, which describes the precise kind of device represented by the locale. Among other features, the component description includes information about instruction set architecture and available extensions. When different fork forms are present for a scheduled codelet, a scheduler can use its locale's component description to find an appropriate fork form to run. When a codelet is passed to a non-leaf scheduler, any descendant schedulers with compatible architectures can potentially execute that codelet. Although codelets will likely have preferred architectures (a codelet that performs a matrix operation may run far faster on a GPU than on a CPU, for example) codelet-specified preferences will have to be weighed against resource availability (the matrix operation can only run faster on a GPU if it can actually get time on that GPU). Different binary forms of the forks may also need to perform different actions, as appropriate for the architecture in question; for example, a fork compiled for a CPU may perform a smaller portion of a matrix operation than a GPU fork, may perform it in a different way, or may even schedule components of the operation on other nearby CPUs.

2.5 Scheduling and allocation

Each locale in the hierarchy has an attached scheduler and allocator, which are used by an application to control runtime management of local time and memory. Schedulers accept enabled codelet instances and ready them for dispatch, generally buffering readied instances in a queue or deque until a thread in the scheduler's locale is ready to run them; allocators accept requests for memory blocks and, when sufficient contiguous memory becomes available within their attached locale, readies codelets to run with the block address as their input.

The locale hierarchy can be used to establish delegating schedulers and allocators whose sole purpose is to select a descendant locale with available resources and forward requests into it. Schedulers and allocators for non-leaf locales may also buffer requests and allow descendants to service them when the latter become idle or have sufficient free resources. Only leaf schedulers actually cause the execution of any codelets; non-leaf schedulers may forward or buffer scheduling requests, but do not handle them directly.

A codelet runtime's leaf schedulers will likely need to support a deque-based scheduling interface whereby a codelet can either be scheduled at the head of the deque, causing it to be dispatched immediately after the current codelet (a roughly LIFO ordering), or at the tail of the deque, causing it to be dispatched after any other readied codelet in-

stances have been run (a roughly FIFO ordering). Schedulers might not cleave to the precise deque ordering, if other measures are taken (*e.g.*) to predictively optimize codelet ordering, but in general the deque ordering should be respected. FIFO ordering is typically sufficient and is often fairer to other application components, but some recursive problems encounter exponential blowup if strict FIFO ordering is used (*e.g.,* a recursive Fibonacci complex, which maxes out at about 1.6^n scheduled codelets for input n), so the LIFO form of scheduling is often also useful.

A simple leaf scheduler operates in a tight loop, repeatedly dequeueing codelets and dispatching them until it reaches an idle state or the runtime is shut down. Codelet scheduling may also be circumvented in some cases by calling the scheduled codelet's run/cancel fork immediately within the current stack, as long as there is enough remaining stack space; this avoids the overhead of enqueueing/pushing the codelet into the scheduler's deque structure and later dequeueing it, but implicitly suspends the original codelet to do so.

When a scheduler becomes idle, it may steal work from other schedulers around it in the hierarchy by consulting its parent locale's scheduler. If that scheduler has work to do, it can pass the work to the idle scheduler; otherwise, it can look for stealable work on the idle scheduler's siblings or consult its own parent scheduler. If no suitable work can be found, an idle scheduler can place its thread in a low-power state until work arrives. When an operating system layer exists beneath the runtime, an idle software thread may block to cede processor hardware to the OS; on bare-bones hardware, the runtime may be able to power- or clock-gate a core to save power.

Locales' allocators may act in a similar fashion to schedulers with respect to allocation requests. For example, a leaf allocator, if unable to satisfy an allocation request, may push that request up to a higher level in the locale hierarchy; a non-leaf allocator may push the request upwards or downwards. However, the cost of migrating allocation requests may outweigh the benefits; migration of a request generally necessitates migration of the codelet execution resulting from request satisfaction, and that in turn may necessitate migration of a codelet's context and associated data.

3. APPLICABILITY TO PARALLEL ALGORITHM CLASSES

A codelet execution model and runtime are broadly applicable to high-performance computing algorithms, which tend to fall into one of two rough classes: fork-join and distributed dataflow. Fork-join algorithms include those based on recursion (*e.g.,* game theory and decision analysis) and data-parallel or SIMD operations (*e.g.,* partial differential equation solvers and Fast Fourier Transform). Recursive algorithms can be parallelized well within a codelet runtime by judicious use of LIFO/FIFO codelet scheduling; data-parallel algorithms typically rely on parallel-**for**-like constructs, and can use locale hierarchy traversal to issue codelets to distinct schedulers.

Distributed dataflow algorithms include those that are primarily data-dependent, such as graph traversals (*e.g.,* Graph500) or semantic web queries, and those that are primarily control-dependent, such as tiled linear algebra (*e.g.,* PLASMA [2]) and adaptive mesh refinement. Codelet instances can be easily encapsulated and associated with data or control dependencies, then dispatched when those dependencies are satisfied, allowing program components to run as soon as possible instead of relying on global synchronization to break execution into distinct stages.

4. RELATED WORK

The basis for the concept of codelets comes from dataflow-related work by Gao et al., who implemented a prototype codelet-based execution model called EARTH [18], which ran on commodity hardware but suffered from scaling limitations. This codelet execution model has been extended to address the needs of exascale systems [20], although no implementing runtime exists. The execution model described in this paper discards some of the theoretical limitations imposed on codelets in Gao et al.'s model (*e.g.,* prohibition of codelet side-effects), adds cancellation semantics to simplify integration with programming language constructs such as exceptions, and adds chaining semantics to simplify establishment of dynamic dataflow interactions and memory cleanup.

Other more commonly used frameworks for parallelizing and distributing programs have been in existence for some time, and include the following:

- MPI [11], which aids distributed coordination/communication within programs, but requires explicit transfers between address spaces and is primarily single-threaded.
- SHMEM [4], an API that establishes a distributed coordination layer and store, but deals primarily with data exchange and remote synchronization and disregards threading.
- OpenMP [7], a framework for integrating basic multithreading control structures into existing program code, but which does not deal with multiple-address-space interactions and which primarily provides for fork/join-like programming patterns (esp. parallel **for** loops).
- Cilk [5], an extension to C for parallelizing recursion by using a fork-join paradigm on top of the existing threading model. Cilk deals well with recursive programming patterns in a single address space, but does not address other problems.
- TBB [15], an API that relies heavily on C++ templates to parallelize programs and deals solely with thread interactions in a single process.
- OpenCL [17], CUDA [14], and DirectCompute, which allow programmers to create program components that run on GPUs. These frameworks are specifically tailored to GPU-like accelerators, and deal only with the CPU-GPU interface.

Unfortunately, integrating more than one of these frameworks into a single application can be difficult, and due to disparate software interfaces for each component in the overall system (nodes in a cluster, threads on a node, and accelerators on a node), programming and load-balancing in a way that effectively coordinates use of a cluster's resources can be difficult as well.

Other prior work includes ParalleX [12], which is a large-scale parallel runtime specification for which HPX [3] exists

as an implementation; the codelet runtime described in this paper is entirely compatible with ParalleX, and may be used as an underlying framework to implement its constructs.

The locale hierarchy described in this paper is a close relative of Hierarchical Place Trees (HPTs) used by the Habanero runtime [19], "places" in the X10 language [8], and locales in the Chapel language [6], as well as the Sequoia language's Parallel Memory Hierarchy (PMH) [10]. However, Sequoia requires static partitioning within the PMH, X10 allows interaction only at leaf nodes of its place tree, and Chapel locales are not arranged in any hierarchy. Habanero's HPTs deal more with data locality and transfer than scheduling or processing, although they are otherwise largely the same as a codelet runtime's locales.

5. ONGOING/FUTURE WORK

Although a full implementation of a codelet runtime as described in this paper is under active development, an earlier prototype version with a reduced scheduling interface and more limited codelet behavior already exists and is available for download from the ET International (ETI) web site [1]. SWARM is intended to present a complete runtime interface and toolkit that programmers or compilers can use to transparently distribute their applications across a compute node or cluster.

Because generating codelets by hand and ensuring that they interact correctly can be a daunting task for programmers accustomed to using traditional imperative languages, ETI is also developing a C-based language, SCALE (SWARM Codelet Association Language Extensions), to simplify asynchronous parallel programming. A prototype SCALE translator compatible with the aforementioned SWARM prototype is available for download and has been used for in-house development on SWARM.

6. CONCLUSION

The codelet execution model, when combined with a capable runtime, can allow software to be scalably parallelized much more easily and transparently than the thread-based and bulk-synchronous models in common use today. Such a runtime allows a clear path forward to exascale computing in the future, and can enable better utilization of hardware resources on present-day computers as well.

7. REFERENCES

[1] SWARM beta download. Online at `http://etinternational.com/swarm`.

[2] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. PLASMA users' guide. Technical report, ICL, University of Tennessee, Knoxville, TN, 2010.

[3] M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling. An application-driven analysis of the ParalleX execution model. Technical report, Louisiana State University, Baton Rouge, LA, USA, Sep. 2011.

[4] R. Barriuso and A. Knies. *SHMEM user's guide for C*. Cray Research, Inc., Eagan, MN, USA, June 1994.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.

[6] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *IJHPCA*, 21(3):291–312, 2007.

[7] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable shared-memory parallel programming*. The MIT Press, 2007.

[8] 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. In *OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM.

[9] J. Dongarra, P. Beckman, T. Moore, et al. The International Exascale Software Project roadmap. *IJHPCA*, 25(1):3–60, 2011.

[10] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[11] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 2nd edition, Nov. 1999.

[12] H. Kaiser, M. Brodowicz, and T. Sterling. ParalleX: An advanced parallel execution model for scaling-impaired applications. *ICPPW*, pages 394–401, Sep. 2009.

[13] P. Machanick. Approaches to addressing the memory wall. Technical report, University of Brisbane, Brisbane, QLD, Australia, 2002.

[14] NVIDIA Corporation, Santa Clara, CA. *NVIDIA CUDA programming guide*, June 2007.

[15] J. Reinders. *Intel Threading Building Blocks*. O'Reilly Media, Sebastopol, CA, July 2007.

[16] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin / Heidelberg, 2011.

[17] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–72, May 2010.

[18] K. B. Theobald. *EARTH: An efficient architecture for running threads*. PhD thesis, McGill University, Montreal, Que., Canada, 1999.

[19] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical Place Trees: A portable abstraction for task parallelism and data movement. In *Proceedings of the 22nd Workshop on Languages and Compilers for Parallel Computing*, Oct. 2009.

[20] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Position paper: Using a "codelet" program execution model for exascale machines. In *EXADAPT '11*, New York, NY, USA, June 2011. CAPSL, ACM.