Designing a Unified Programming Model for Heterogeneous Machines

Michael Garland
NVIDIA
Email: mgarland@nvidia.com

Manjunath Kudlur
NVIDIA
Email: mkudlur@nvidia.com

Yili Zheng
Lawrence Berkeley National Lab
Email: yzheng@lbl.gov

***Abstract*—While high-efficiency machines are increasingly embracing heterogeneous architectures and massive multithreading, contemporary mainstream programming languages reflect a mental model in which processing elements are homogeneous, concurrency is limited, and memory is a flat undifferentiated pool of storage. Moreover, the current state of the art in programming heterogeneous machines tends towards using separate programming models, such as OpenMP and CUDA, for different portions of the machine. Both of these factors make programming emerging heterogeneous machines unnecessarily difficult.**

We describe the design of the Phalanx programming model, which seeks to provide a unified programming model for heterogeneous machines. It provides constructs for bulk parallelism, synchronization, and data placement which operate across the entire machine. Our prototype implementation is able to launch and coordinate work on both CPU and GPU processors within a single node, and by leveraging the GASNet runtime, is able to run across all the nodes of a distributed-memory machine.

I. Introduction

Over the last 5 years, mainstream machines have quickly evolved to embrace heterogeneous architectures and intra-processor parallelism. This evolution has been driven by the search for improved computational throughput, but even more importantly, it has been driven by the need for improved energy efficiency. Computers containing both multicore CPUs, which are moderately multithreaded, and manycore GPUs, which are massively mulithreaded, are now commonplace. And this trend is apparent across a broad range of computing devices, from mobile phones and tablets up to the largest supercomputers.

Programming systems have not kept pace with this rapid architectural evolution. Contemporary mainstream programming languages largely still reflect a mental model in which processing elements are homogeneous, concurrency is limited, and memory is a flat undifferentiated pool of storage. While broad-based languages such as Java and now C++11 do provide standard constructs for concurrent threads, they are focused on managing small pools of independent threads and do not really address the needs of massive multithreaded processors and provide no model for managing heterogeneous processors. NVIDIA's CUDA platform pioneered a programming system for massive parallelism and heterogeneous processors, but is generally combined with other layers such as OpenMP for programming multicore CPUs. Today's large-scale machines consist of potentially many nodes each containing a heterogeneous collection of processors. It would not be uncommon to program such machines in a mixture of three programming models: CUDA for GPUs, OpenMP for CPUs, and MPI for inter-node communication.

In this paper, we describe the design of the Phalanx programming model. Our goal is to develop a unified programming model for heterogeneous machines, in both single and multi-node configurations. It provides a simple asynchronous task model providing constructs for launching large, structured collections of cooperating threads, thus supporting both coarse-grained task parallelism and fine-grained thread parallelism. This generalizes the kernel model developed by CUDA and, unlike constructs in many other systems which focus on launching single threads, is well-suited to controlling modern GPUs. Alongside this control model, Phalanx provides a PGAS-like memory model that presents a global address interface even on distributed machines. This is coupled with a hierarchical model of "places" that permits programs to target data & task placement at particular parts of the machine while also querying and reasoning about the location of data. The implementation of these constructs is built upon a convention for managing platform heterogeneity via information embedded in the type signature of functions.

We have built a prototype implementation of this programming model as a C++ template library. It requires no compiler support other than the language extensions for GPU computing provided by the standard CUDA C++ compiler. It is built on top of the CUDA, OpenMP, and GASNet runtimes for running code on GPUs, CPUs, and distributed nodes, respectively. Since it is built as a library, it interoperates cleanly with other existing code already built upon these underlying platforms. Thus, while it is meant to provide a convenient unified programming model, it can also leverage a substantial body of existing code.

II. Architectural Trends

Traditional processors have for many years focused on transparently scaling sequential performance through a combination of clock frequency scaling, sophisticated caches, superscalar execution, and speculation. Modern architectures have reached

This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

the point of diminishing returns where additional exploitation of instruction-level parallelism imposes an unsustainably high energy cost and have moved to expose parallelism explicitly. Parallel processor architectures have displaced traditional sequential machines across the whole gamut of machines, from mobile phones to supercomputers.

Modern GPUs have been at the leading edge of this architectural movement towards explicit parallelism. Traditional architectures, exemplified by modern CPUs, are *latency-oriented* processors whose design is focused on making single sequential threads execute as quickly as possible. In contrast, GPUs have evolved as *throughput-oriented* processors [1] whose design is geared towards maximizing the aggregate amount of work they can complete in a given time frame.

Heterogeneous machines, which provide both latency-oriented and throughput-oriented processors, are essential for energy-efficient computing [2]. Throughput-oriented processors can deliver high performance on parallel workloads using comparatively little energy. Latency-oriented processors, on the other hand, can execute sequential code much more quickly by leveraging instruction-level parallelism aggressively, but at the cost of higher energy consumption. As any real application consists of a mix of parallel and sequential tasks, it is advantageous to provide both kinds of cores in search of energy efficiency.

The trajectory of energy efficient architecture raises three important concerns with which programmers must be equipped to cope. First, machines are becoming heterogeneous which represents a departure from the fundamentally homogeneous design of current programming systems. Second, throughput-oriented processors employ massive multithreading to hide latency, which creates a level of concurrency for which most single-node programming systems are unprepared. Finally, on-chip memory hierarchies are becoming deeper while off-chip memory bandwidth is becoming more precious.

III. Phalanx Programming Model

Phalanx is a programming model whose goal is to make programming machines such as those we have just described both simpler and more uniform. It is an efficiency-oriented model, providing relatively low-level constructs that allow a programmer to map an explicitly parallel program efficiently to a heterogeneous machine. Phalanx aims to provide a suitable substrate for higher level abstractions for performance portability and productivity, in the style of Thrust [3] for example, but does not address those goals itself. The Phalanx model itself is simply concerned with providing a homogeneous notation for programming heterogeneous machines with rich memory hierarchies.

We embed the Phalanx model in C++ as a library comprised of a collection of templated types and generic functions. While we believe that the key ideas of Phalanx could be realized in other languages, this choice avoids the need for any special compiler support and allows Phalanx programs to easily interoperate with a great deal of existing code.

In the sections that follow, we describe the main components of the Phalanx programming model. Section IV provides further information on how our prototype library is implemented. In reading the following description, it is important to know that all Phalanx interfaces are contained within a Phalanx-specific name space (e.g., `phalanx::place` in C++ syntax). In describing the model, we will omit the `phalanx::` qualifier where there is no ambiguity; this corresponds to incorporating a C++ `using namespace phalanx;` declaration in the program.

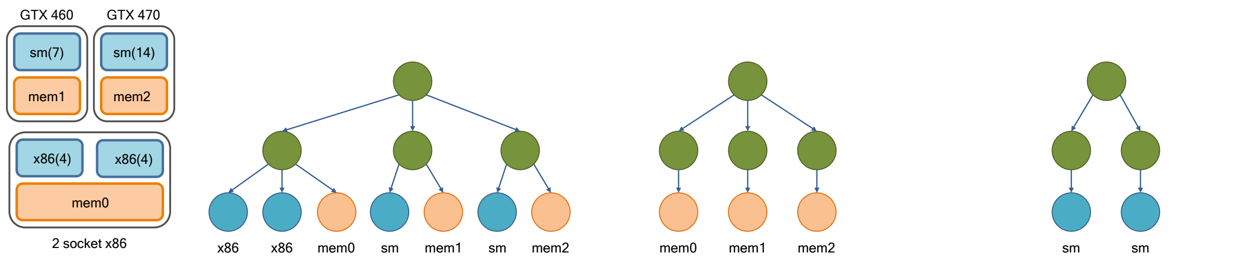
A. Heterogeneous Machine Model

The machines that this work targets feature both heterogeneous collections of processors and rich memory hierarchies. Using such machines efficiently often requires programmers to reason about the structure of the machine on which their programs are running. For instance, knowing the details about proximity of a memory in which data lives and the processors on which tasks are running allows the programmer to place computation close to the data.

Phalanx exposes the structure of the machine to the programmer through objects of type `phalanx::place`. A `place` is a hierarchical data structure that organizes the processors and memories within the machine into a tree. Individual processors are placed at the leaves of this tree. Internal nodes represent logical groups of processors. Physical memory structures co-located with one or more processors are attached to the nodes representing those processors. This hierarchical structure provides a simple means of reasoning about locality. A single processor leaf sees a chain of memories above it along the path to the root. Each successive level represents a memory that is progressively "further away", as measured by a system-defined metric such as latency or access energy.

The `place` describing the entire machine is assembled by the Phalanx runtime during initialization and is provided to the program. Subsequent calls to Phalanx interfaces for creating tasks and allocating memory require `place`-valued arguments indicating where the new task/data is to be located. The hierarchical nature of places permits the program to be more or less specific about the placement of tasks and data, at the discretion of the programmer. Launching a task on a place naming the entire machine leaves the precise placement of the new task at the discretion of the runtime, whereas specifying a place containing a single processor mandates that it must be placed there. Thus, the expert programmer can specify the precise placement of tasks and data for best performance while others are free to leave these decisions in the hands of the system.

Phalanx provides a set of interfaces to initialize, query, traverse, and filter `place` trees. We summarize three salient functions here, but this is not meant to be a complete description of the interface. Figure 1 shows a simple corresponding example. The physical machine consists of a dual socket, 4-core x86 system coupled with 2 GPUs. The GPUs themselves consist of 7 and 14 streaming multiprocessors (SMs). Each of the main CPUs and the two GPUs have an associated memory.



(a) A machine (b) `get_global_machine` (c) `slice_by_type<memory>` (d) `slice_by_isa<sm>`

Fig. 1: An example machine (a) containing two 4-core CPUs and two GPUs with 7 and 14 SMs, respectively. It is represented in Phalanx as a `place` (b) and can be subsequently filtered in different ways (c–d).

A NUMA-aware runtime might choose to present the CPU's storage as one memory per NUMA domain rather than the single memory shown here.

When a Phalanx program is initialized, it obtains an object representing the root of the hierarchical machine.

```
int main(int argc, char **argv)
{
    main_task self = initialize(argc, argv);
    place M = get_global_machine(self);
    ...
}
```

The program is then able to sub-select various pieces of this tree according to different criteria. It may, for instance, construct a tree containing only nodes of a given category:

```
    place rams = slice_by_type<memory>(M);
```

where the category is one of `memory` or `processor`. It may also construct a tree containing only processors implementing a specific ISA:

```
    place gpus = slice_by_isa<sm>(M);
```

where the ISAs used in our examples are `x86` (for the CPU) or `sm` (for the GPU).

B. *Task Model*

As in many parallel programming systems, Phalanx programs are expected to create and execute a potentially large number of asynchronous tasks. However, unlike other models with the exception of those influenced by CUDA, an individual Phalanx task can be composed of *many* threads. The collection of threads forming a task can also be structured hierarchically.

The hierarchical structure of a Phalanx task is described by values of type `phalanx::domain<Policy,SubDom>`. The `Policy` argument in this templated type specifies the scheduling policy used at this level, of which there are currently two options:

- `streaming_group`: A group consisting of independent sub-groups, which may be executed in any order, in parallel or sequentially.
- `parallel_group`: Subgroups are permitted to synchronize and must be scheduled so that unblocked subgroups can make progress.

The `SubDom` type defines the organization of the subgroups of this domain.

The base domain type is `phalanx::thread<P>`, which signifies a single thread compiled to run on processors of type `P` (e.g., `x86` or `sm`). We can then, for instance, describe a domain of completely independent GPU threads:

```
domain<streaming_group<thread<sm>>>
```

or a set of parallel thread blocks in the style of CUDA's kernel model:

```
domain<streaming_group,
       domain<parallel_group, thread<sm>>>
```

Note that domains defined in this manner are homogeneous: every level of the task hierarchy is uniform in structure. Since domains are normal C++ types, the program can use normal C++ `typedef` declarations to create more convenient names for common domain organizations.

Phalanx adopts a convention whereby functions that are task entry points always take a `domain`-valued object as their first parameter. Thus, both the structure of the task hierarchy and the target processor platform are directly encoded in the type signature of the function. By convention, we refer to these values as `self`, since they always name the executing task. For hierarchical tasks, these objects represent the root of the task structure and they export a set of interfaces for traversing and partitioning the threads of the task. The purpose of task objects is twofold. First, the hierarchical structure of tasks provides the mechanism for expressing the scope for shared resources, primarily shared storage and synchronizers. Second, it provides a handle for explicitly naming the set of threads participating in a collective operation.

Phalanx executes tasks in a SPMD fashion. Every thread in a task begins executing at the same entry point and receives the same parameters. The `domain` object passed as the first parameter provides a collection of methods with which threads can determine their position in the task and cooperate with each other. This interface includes the methods:

- `size()` and `index()` to compute the number of subgroups and the calling thread's position, respectively;
- `subdomain()` returns the subgroup containing the calling thread;

- scratchpad() to obtain a pointer to shared scratchpad memory; and
- barrier() to synchronize all threads belonging to the group.

Only domains of type parallel_group provide scratchpad and barrier methods since only they permit communication amongst their constituent threads.

Tasks are created by the phalanx::async construct, which has the following form:

```
event e = async(who, machine, shape)
               (task_entry, ...);
```

This launches a task whose entry point is a function named task_entry. The shape parameter is an object of type domain, which specifies both the structure of the task, and the resources required at different level of the task hierarchy. Scratchpad memory that will be shared by a group of threads is an example of a resource that can be requested as a part of the shape parameter. Phalanx provide utility functions streaming() and parallel() to simplify the construction of the shape parameter. For instance, the following code constructs N streaming groups of threads, each containing M threads each. Further, each thread group will share a scratchpad memory of P bytes.

```
streaming(N, scratchpad(P),
          parallel(M, thread<>()))
```

The shape along with any values passed in place of the ellipsis (...) will become its parameters. The parameter who is a domain naming the thread(s) performing the task launch, and is used to distinguish between tasks launched by a single thread and collective launches by a set of threads. The machine parameter is a place indicating where the task is to be mapped. For aggregate places, the threads of the task may be distributed at the discretion of the runtime. The domain parameter specifies the shape of the parallel task to be launched. An event object is returned by async, and this serves as a handle to the parallel task that has been launched. The handle can be passed to the phalanx::wait function that blocks until the corresponding task completes or as a precondition to other async-like functions. For example, the async_after interface:

```
event f = async_after(who, e, m, s)
                     (task_entry, ...);
```

makes the launch of the new task conditional on the completion of the event e. Declaring dependencies explicitly in this fashion, rather than waiting directly on an even in the calling thread, is essential for scalability. It allows the runtime to decouple waiting for task completion from the sequential execution of the calling thread and implement waiting efficiently. Our CUDA backend, for instance, takes advantage of this information to avoid involving the host CPU in waiting for a chain of dependent GPU tasks to complete.

C. Memory Model

Phalanx provides a PGAS-like memory model to the program. Memory locations are represented by a phalanx::ptr template whose definition is along these lines:

```
template<typename T,
         typename MemorySpace>
struct ptr {
    T& get();
    void put(const T&);
    place place();
};
```

The type T is the type of the data stored in the storage location named by the pointer. Pointers are categorized into separate memory spaces by the MemorySpace parameter. Only pointers within the same memory space may be compared or subtracted to compute offets. Pointers in different memory spaces may provide different, incompatible, implementations of the put and get interfaces. Associating memory spaces with pointers allows Phalanx to handle machines with semantically distinct address spaces. Encoding the memory space in the type of the pointer allows for efficient access to the different memory types. Typical memory spaces would include global_memory, representing global addresses spanning multiple nodes, local_memory representing the system memory within a single node, and device_memory representing the GPU address space within the local node.

The ptr class also provides a place() interface function for querying the location of the storage. This returns to the caller a value of type phalanx::place naming the location of the memory in the machine. This permits programs to allocate additional storage near this data, launch tasks near this data, or determine whether the memory is near the processor running the current task, among others. Program logic of this sort can thereby use information about the location of data to manage both task and data locality. The Phalanx memory model has been designed at this low-level so that it can interoperate easily with the host language. Phalanx pointers, with the exception of the place interface, behave exactly like normal pointers. The put and get methods are generic interfaces to dereference the pointer. Specializations of phalanx::ptr for specific memory spaces can provide implementations that can be mapped to single instructions. Thus for example, these pointers can, where appropriate, support dereferencing via the standard *ptr syntax.

Memory management in Phalanx is performed through fairly standard interface routines shown here :

```
template<typename Domain,
         typename T,
         typename MemorySpace>
ptr<T, MemorySpace>
allocate(const Domain& who,
         place M, size_t count);

template<typename Domain,
         typename T,
         typename MemorySpace>
void deallocate(const Domain& who,
                ptr<T, MemorySpace>& Pointer);
```

The `who` parameter informs the runtime about collective allocations performed by a group of thread(s) versus allocations performed by a single thread. The routines also takes the *name* of the location where memory allocation/deallocation is requested. The location can be a specific memory place, or a slice of the machine tree, in which case the runtime allocates storage in one or more of the memories present in the tree. Phalanx presents an asynchronous interface for data movement between memories. The `phalanx::copy` is the primary data movement interface:

```
template<typename Domain,
         typename T,
         typename MemorySpace1,
         typename MemorySpace2>
event copy(const Domain& who,
           ptr<T, MemorySpace1>& Pointer1,
           ptr<T, MemorySpace2>& Pointer2,
           size_t count);
```

`copy` returns a `phalanx::event` object, which can be passed to `async_after` or `wait` functions. With this functionality, programmers can build complex software pipelines which overlap data movement and computation.

IV. Prototype Implementation

The programming model we have just described is designed to be realized as a C++ template library. Moreover, it is designed so that the mapping to underlying parallel runtimes is both simple and efficient. We have chosen to build a prototype implementation using CUDA, OpenMP, and GASNet for programming the GPU, CPU, and distributed memory portions of the machine, respectively. The programming model defines its interface in terms of a set of generic functions and templated types. A backend supporting a specific platform, say CUDA, is simply a set of appropriate instantiations of these generic interfaces.

A. CUDA Backend

The CUDA backend provides instantiations restricted in such a way that the interface can be directly mapped to the primitives provided by CUDA. For instance, the `index()` and `size()` methods of `thread<sm>` and `domain<parallel_group, thread<sm>>` will expand directly to CUDA primitives as shown in Figure 2.

Launching a function using `async` automatically dispatches to the CUDA backend if the task entry point's first parameter (the task type) is one of the above types. The implementation of `async` in the CUDA backend simply maps thread blocks and grids of blocks to the first available GPU on the machine tree that is passed as the target place. Similarly, multi-grid CTAs are mapped to two or more available GPUs on the machine tree.

Events returned by `async` are implemented using the CUDA stream and event infrastructure. For each call to `async`, the backend will:

1) Create a new CUDA stream s.
2) Launch the specified kernel in stream s.
3) Record a CUDA event e in stream s.
4) Destroy stream s and return e in a `phalanx::event`.

This mode of stream use, namely creating and destroying a stream on each kernel launch, deviates from the normal CUDA idiom. However, this usage mode is supported by the CUDA runtime and allows each kernel launch to potentially run asynchronously from the others. The implementation of `async_after` is similar, but inserts a CUDA event wait prior to the kernel launch. The CUDA runtime will defer the launched kernel until the provided event has occurred, and the burden of waiting will be placed on the target GPU rather than the host CPU.

B. OpenMP Backend

We chose OpenMP to provide the underlying runtime for execution on multicore CPUs. Our choice is largely driven by pragmatic concerns. OpenMP enjoys wide support in both commercial and free compilers, and by building on OpenMP we allow Phalanx programs to interoperate with the existing base of OpenMP code.

The OpenMP backend uses compile-time template expansion to map a Phalanx `async` call onto a potentially nested sequence of `omp parallel` and `omp parallel for` blocks. A Phalanx `parallel_group` is mapped to the former while a `streaming_group` is mapped to the latter. The `thread<x86>` type implements the leaf level of the thread hierarchy corresponding to a single OpenMP thread.

Calling Phalanx `async` with a domain whose leaf level threads are of type `thread<x86>` automatically dispatches to the OpenMP backend. The OpenMP launcher maps the entire thread hierarchy on the first multicore processor found in the tree of places passed to the `async` function. In order to manage nesting correctly, the backend's task launcher stores the thread array size and indices for each of the threads explicitly in the `domain` data structure and `size()` and `index()` functions simply return these values. The `barrier()` interface uses `omp barrier` to synchronize threads that are grouped by the `parallel_group` scheduling policy.

In our current prototype implementation, we have chosen not to use the task constructs introduced in OpenMP 3.0 so that our implementation can be compiled with Microsoft's Visual C++ compiler. While this significantly improves the portability of Phalanx programs, it has the unfortunate consequence that every Phalanx task is actually synchronous since there is an implicit barrier at the end of OpenMP's parallel sections. This limitation could be eliminated by using a different runtime, such as Intel's Thread Building Blocks (TBB), although we would subsequently lose the ability to interoperate cleanly with existing OpenMP-based code.

C. GASNet Backend

The Phalanx backend for distributed-memory systems is built on top of GASNet, a widely used global address space runtime library [4]. GASNet supports communication primitives including active messages, one-sided communication and collective operations on diverse computer architectures

```
template<>
struct thread<sm> {
  __device__
  unsigned index() const {
    return threadIdx.x;
  }

  __device__
  unsigned size() const {
    return 0;
  }
};
```

(a) Thread

```
template<>
struct domain<parallel_group,
              thread<sm> > {
  __device__
  unsigned index() const {
    return blockIdx.x;
  }

  __device__
  unsigned size() const {
    return blockDim.x;
  }

  __device__
  void barrier() const {
    __syncthreads();
  }
};
```

(b) Thread array

```
// Grid of CTAs
template<>
struct domain<streaming_group,
       domain<parallel_group,
              thread<sm> > >;
// Multiple grids for multi-GPU
template<>
struct domain<streaming_group,
       domain<streaming_group,
       domain<parallel_group,
              thread<sm> > > >;
```

(c) Grids of thread arrays

Fig. 2: Examples of how Phalanx domains are mapped down to CUDA primitives.

and interconnects. Our implementation of constructs such as `phalanx::async` and `phalanx::allocate` is based on active messages, which essentially provide a mechanism to execute functions in a separate address space equivalent to remote procedural calls (RPCs).

The GASNet backend is designed to work with the CUDA and OpenMP backends hierarchically; it provides inter-node runtime services while the CUDA and OpenMP backends provide the intra-node runtime. A Phalanx application can use all three backends together to express hierarchical parallelism across the entire machine. For example, a Phalanx application may first spawn a coarse-grained task on a remote node, which then spawns fine-grained local CPU tasks with the OpenMP backend and GPU tasks with the CUDA backend.

Each OS process is mapped to a Phalanx place in the machine model, and each place is assigned a unique integer rank from 0 to $p-1$, where p is the number of processes. A `parallel_group` is implemented with a `team` object in GASNet, which is similar to a `MPI_Communicator` that supports collective communication such as barriers. Because threads in a `streaming_group` are independent by definition, they can be simply scheduled to available processors in a round-robin fashion or in a random order without special treatment.

Launching tasks on remote nodes is implemented by GASNet active messages. Each active message has a header containing the function pointer of a user space handler to be executed upon message arrival and the arguments to the handler function. As in other backends, we use C++ templates to match function prototypes for `phalanx::async` and generate stub functions for data marshaling. We first pack all arguments into an active message and then send the active message request through the network. After the remote node receives the active message, it unpacks the packet to retrieve the task information. Because there are restrictions on what functions can be executed inside active message handlers to prevent deadlocks, the receiver inserts the task into a queue for later execution rather than executing it immediately. All local and remote tasks are managed by a unified *async task queue*. Each process regularly polls the task queue to execute pending tasks. In addition, our runtime supports dynamic task dependencies between tasks. A task is fired only if all of its prerequisite tasks are completed.

A global address space is convenient for data sharing and communication but is challenging to implement on distributed-memory systems. Unlike software distributed memory systems (DSMs) such as TreadMarks [5], which use the OS virtual memory paging mechanism to provide a shared address space abstraction, we take a partitioned global address space (PGAS) approach and encode the data location in global address space pointers and name symbols.

A global address pointer encapsulates within its `phalanx::ptr` structure both a local virtual memory address and the process ID where the local pointer is valid. When accessing data through a global address pointer, the overloaded dereference operator calls an underlying GASNet function to fetch the data to a local buffer and then returns the data in the local buffer. When allocating memory in the global address space, the runtime first checks if the place for the requested memory is local. If it is local, the runtime allocates memory using regular memory management mechanism such as `malloc` and `new`. If the requested memory is for GPU, the runtime uses `cudaMalloc` to allocate the memory. If the requested memory is on a remote node, the runtime uses an active message to allocate memory, and the remote node replies to the initiating node with the resulting memory pointer.

Reading and writing data in the global address space are provided through one-sided communication functions `phalanx::put` and `phalanx::get`, respectively. These operations are mapped directly onto GASNet one-sided communication functions. In addition, we support non-blocking data communication, which allows the data transfer functions to return an event without waiting for the operation to complete. The caller may then perform other work, checking for the completion of the communication through the previously returned event.

The global address space in Phalanx covers both CPU memory and GPU memory in a cluster. We extended GASNet to support moving data between two GPUs. Due to limitations of current systems, memory transfer operations can only be initiated from CPUs in the current prototype; however, NVIDIA's GPUDirect support for direct RDMA from GPU memory may help make this limitation unnecessary in the future. We uses active messages for moving data when the source or the destination is on a GPU. To improve the performance for long messages, the GASNet GPU extension automatically pipelines the communication by overlapping network communication and CPU-to-GPU communication.

V. Evaluation

In order to evaluate the utility of the Phalanx programming model and our prototype instantiation, we have implemented a collection of common programs in our model for both single-node and multi-node machines. We focus on relatively simple computations with well known parallelizations. Our goal is to assess whether our programming model permits straightforward implementations that are reasonably efficient and scalable. We have not attempt to produce highly optimized implementations. While such implementations can be written in Phalanx, the effort spent on writing them is largely a reflection of the complexity of implementing a specific algorithm on a specific architecture rather than on the complexity of using the programming model.

A. Single Node

Figure 3 contains a simple, but complete, example of a Phalanx program for performing a basic SAXPY-style parallel computation. Note that in the current prototype, the `saxpy` function template must be manually expanded and annotated with the `__global__` declaration modifier to make this program compile with the `nvcc` compiler for the CUDA backend. Phalanx programs begin execution in a single thread, and should always first initialize the runtime and obtain the handle for the initial thread (line 13). Using this handle, the startup thread can also obtain a `place` instance describing the machine allocated to the program by the runtime (line 17). These main task and machine objects are then provided to subsequent Phalanx interfaces. Lines 20–21 allocate the x and y arrays, which are then initialized with `std::fill`.

After initial setup is complete, the code in line 26 launches a task on an x86 CPU in the current machine. By providing the entire machine `M` as the target for the launch, the program indicates that the runtime is free to chose *any* x86 core in the machine. If desired, the program can provide a more specific target that would constrain the placement of the task.

```
1  template <typename Platform>
2  void saxpy (domain<streaming_group<>,
3                     thread<Platform> > group,
4              float a, int n,
5              ptr<float> x, ptr<float> y)
6  {
7    int i = group.subdomain().index();
8    if (i < n) y[i] = a * x[i] + y[i];
9  }
10
11 int main (int argc , char** argv)
12 {
13   // Obtain handle for thread running main()
14   main_task self = initialize(argc, argv);
15   //
16   // Get object describing hierarchical machine
17   place M = global_machine (self);
18
19   // Allocate and initialize memory
20   ptr<float> x = allocate<float>(self , M, n);
21   ptr<float> y = allocate<float>(self , M, n);
22   std::fill(x, x+n, 1.0);
23   std::fill(y, y+n, 10.0);
24
25   // Launch on an X86 - type processor
26   event e1 = async(self, M, n)
27                   (saxpy<x86>, 2.0f, n, x, y);
28   wait (self , e1);
29
30   // Launch on an SM - type processor
31   event e2 = async(self, M, n)
32                   (saxpy<sm>, 2.0f, n, x, y);
33   wait(self, e2);
34
35   // Deallocate memory and end the program
36   deallocate(self, x);
37   deallocate(self, y);
38   return 0;
39 }
```

Fig. 3: A simple example of a Phalanx program for computing $y \leftarrow ax + y$ on two n-vectors x and y.

After waiting on the completion of this task (line 28), the program then launches a task on a GPU. Note that the `async` calls are identical except for the instantiation of the `saxpy` function template. In the prototype, `sm` typed functions and `x86` typed functions are dispatched to the CUDA backend and OpenMP backend respectively. The `saxpy` function itself is also identical for both the CUDA and OpenMP backend. This demonstrates how the Phalanx programming model allows the programmer to focus on mapping the parallelism using a uniform set of interfaces rather than disparate low level interfaces of a heterogeneous system.

As mentioned previously, our single-node prototype represents an extremely thin interface on top of the CUDA and OpenMP runtimes. Phalanx is designed to act as a thin veneer over the underlying platforms, and mapping of Phalanx interfaces to specific platforms are all achieved through compile time template expansions. Consequently, the runtime overhead of using Phalanx rather than code written natively in these platforms should be effectively zero. To verify this assertion, we implemented three versions of the `saxpy` program: Phalanx program in Figure 3, a native CUDA program, and a native OpenMP program. We benchmarked their performance on a

16 million element problem using a machine comprised of an Intel Core i7 950 CPU and two NVIDIA Tesla C2050 GPUs. As expected, the two portions of the Phalanx program ran at the same rate as the native CUDA and OpenMP programs.

In addition to this simple SAXPY program, we have also implemented programs for sparse matrix-vector multiplication (SpMV) and breadth-first search (BFS) on graphs. These are more complex programs that solve irregular problems. Both programs target multiple GPUs using the CUDA backend of Phalanx. BFS, for instance, launches multiple tasks to perform different stages of breadth first search. However, no additional complexity was imposed by the Phalanx model, compared to writing these programs directly in CUDA. In fact, the hierarchical machine model and event model of Phalanx provides a simpler abstraction compared to CUDA's device and stream management. However, these abstractions incurred no runtime overhead, and the Phalanx and direct CUDA versions achieved identical performance for both.

B. Distributed Memory

To demonstrate the scalability of Phalanx, we implemented three important and commonly used benchmarks: (1) dense matrix-matrix multiplication, (2) sparse matrix-vector multiplication, and (3) 2-D FFT. We benchmark the performance of our Phalanx implementation on four representative distributed memory machines described in Table I. We have chosen these parallel algorithms because they are well known to be scalable, thus focusing on the design and implementation of the Phalanx programming system rather than the scalability of the algorithms themselves.

Dense matrix-matrix multiplication computes the product of two square n-by-n matrices

$$C[i,j] = \sum_k A[i,k] \cdot B[k,j].$$

Our implementation uses the popular SUMMA algorithm [6]. The matrices are partitioned and stored in the global address space with a 2-D block-cyclic distribution. Each Phalanx place only stores a subset of matrix blocks but keeps a global matrix view through which the programmer can conveniently access any part of the physically distributed matrix. In addition, we use C++ operator overloading to make indexing a sub-block in the global matrix as convenient as writing subscripts in mathematical equations. Local matrix computations are performed with vendor-optimized BLAS libraries [7], [8], [9], which can be easily accomplished since Phalanx is simply a C++ library. Communication is performed using bulk data transfers performed via `phalanx::copy` in sub-matrix block granularity.

Sparse matrix-vector multiplication (SpMV) computes $y = Ax$ where A is a sparse matrix and x, y are dense vectors. It has low computational intensity and should be purely bandwidth bound. We store sparse matrices using the compressed sparse row (CSR) storage format. To distribute the SpMV computation, we simply partition the matrix by rows and

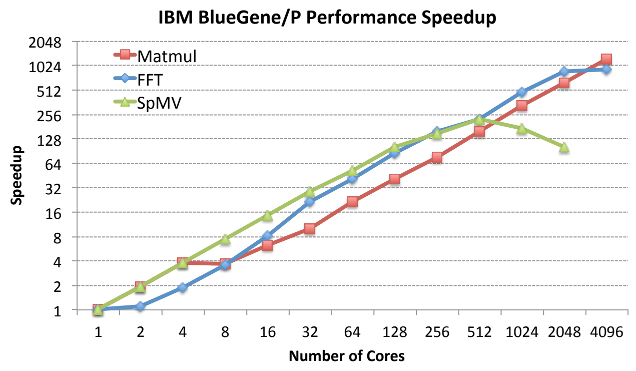


Fig. 4: Phalanx benchmark performance on the IBM BlueGene/P

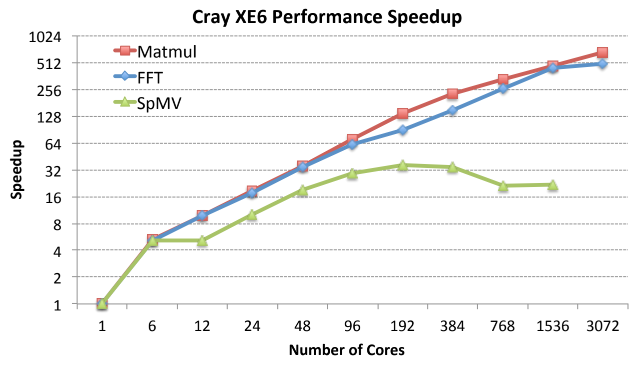


Fig. 5: Phalanx benchmark performance on the Cray XE6

distribute them evenly across the nodes. To minimize internode communication, we replicate the vector x on every place over the global address space.

The 2-D Fast Fourier Transform (FFT) is another widely used numerical method for many applications. We use the classical row-column multi-dimensional FFT algorithm, which performs 1-D FFT along each of the dimensions in succession. Because the array is partitioned and distributed on different processors, we must transpose the array globally after each local FFT phase to make the data for the next FFT phase local and contiguous. Array transpose operations require all-to-all communication patterns that are implemented by non-blocking data transfer functions. Local FFTs are performed by the FFTW library [10] for CPU and by the CUDA FFT library for GPU.

Figures 4, 5, 6 and 7 summarize the weak-scaling performance of our benchmarks on the representative distributed-memory machines, where we increase data size in proportion to the core count. The graphs show speed-up over sequential performance. Table II shows the corresponding absolute performance in GFLOPS for the smallest and largest machine configurations for each platform.

Matrix multiplication performance scales well on all machines. It also demonstrates the best scaling across all benchmarks, since it is an inherently very scalable operation, per-

System	Intel Infiniband Cluster	Cray XE6	IBM Blue Gene/P	Cray XK6
Processor	Intel Xeon X5530	AMD Opteron 6172	IBM PowerPC 450	NVIDIA Tesla X2090
Clock rate	2.4 GHz	2.1 GHz	0.85 GHz	0.65 GHz
Execution units per NUMA domain	4 cores	6 cores	4 cores	16 multiprocessors
NUMA domains per node	2 (2 sockets)	4 (2 sockets)	1 (1 socket)	1 (1 socket)
Execution units per node	8 cores	24 cores	4 cores	16 multiprocessors
Peak double precision throughput per node	84.8 GFLOPS	201.6 GFLOPS	13.6 GFLOPS	665 GFLOPS
Memory per node	24 GB DDR3-1066	32 GB DDR3-1066	2 GB DDR2	6 GB GDDR5
Peak memory bandwidth per node	25.6 GB/s	25.6 GB/s	13.6 GB/s	177 GB/s
Compiler	Intel C/C++ 11.1	PGI 11.3	IBM XLC for BlueGene	CUDA 4.1
Math library	Intel MKL 10.2	Cray Scientific Library 10.5	IBM ESSL	CUDA BLAS/FFT
Interconnect type	Infiniband 4X QDR	Gemini 3-D Torus	3-D Torus and Collective	Gemini 3-D Torus
Peak interconnect bandwidth per direction	32 Gb/s	66.4 Gb/s	3.4 Gb/s	66.4 Gb/s

TABLE I: The distributed-memory machines used in our experiments. Our experiments on the Cray XK6 perform their computations on the GPU, and use the CPU—a 16-core AMD Opteron 6274—for runtime services such as data communication.

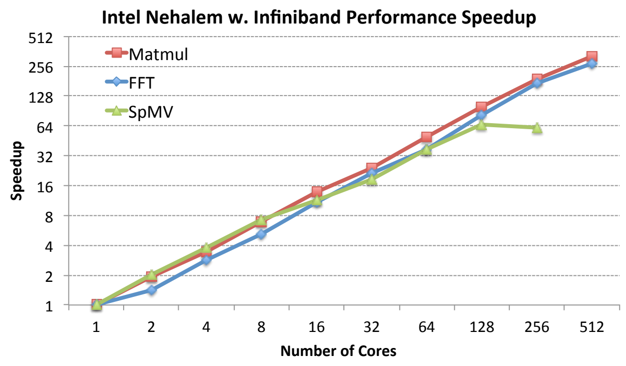


Fig. 6: Phalanx benchmark performance on the Intel Xeon (Nehalem) cluster with Infiniband

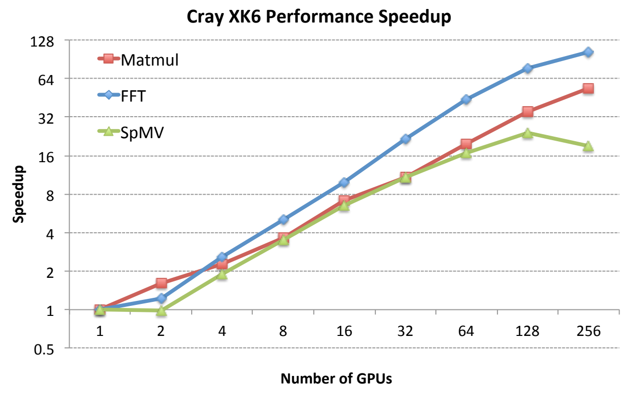


Fig. 7: Phalanx benchmark performance on the Cray XK6

forming $O(n^3)$ operations with only $O(n^2)$ data accesses. This experiment shows that our prototype Phalanx implementation has relatively low runtime overheads, thus enabling intrinsically parallel algorithms to scale well.

For FFT, each Phalanx place is mapped to a CPU core and communicates to each other through the `phalanx::copy` function. The underlying runtime automatically uses shared memory for communication when two places are on the same physical node. The all-to-all communication in FFT is bound by network bisection bandwidth. Nevertheless, FFT performance also scales reasonably well on all machines, showing that the communication primitives in Phalanx are performing well.

	Performance in GFLOPS		
System	MatMult	2-D FFT	SpMV
IBM BlueGene/P	2.45 – 3030.61	0.168 – 155.82	0.05 – 5.52
Cray XE6	6.48 – 4228.65	0.37 – 187.22	0.29 – 6.35
Infiniband Cluster	9.39 – 3008.13	0.51 – 139.0	0.22 – 14.10
Cray XK6	170.47 – 9140.50	0.31 – 32.37	0.25 – 4.86

TABLE II: Benchmark performance across platforms tabulated as a range from smallest to largest machine configurations.

SpMV scales well for small core counts, but scales less well at larger core counts. This is because our current implementation copies the vector data from the master thread to other threads sequentially. This sequential data distribution quickly becomes the scaling bottleneck due to the low computational intensity of SpMV. This limitation can be solved by incorporating broader support for efficient collective communication primitives, such as broadcast and reduction, into a future implementation of Phalanx.

VI. Related Work

The primary goal of Phalanx is to provide a single-source model for programming heterogeneous machines where all portions of the program are written using the same common notation. A number of other systems have adopted similar goals and can be broadly grouped into three categories. First are the directive-based systems [11], [12], [13], [14] which augment an existing base language (usually C or Fortran) with extra-linguistic directives to be interpreted by a custom parallelizing compiler. The second category, exemplified by

Thrust [3], provides library interfaces that abstract the details of the underlying machine. The systems in both of these two categories emphasize performance portability, a goal which they achieve by providing a restricted interface to the programmer that hides machine-specific details. Phalanx falls into a third category which aims to provide low-level constructs that have a uniform interpretation on different machines while not hiding machine-specific details.

Phalanx employs a memory model closely related to the PGAS (Partitioned Global Address Space) model used by languages such as UPC [15], Titanium [16], Chapel [17], X10 [18]. The Phalanx memory model generalizes the PGAS approach as found in these languages by providing a hierarchical rather than flat model of places and by adopting a model of memory spaces that extends beyond the usual global/local dichotomy. The hierarchical place trees developed in Habanero-Java [19], [20] are the most closely related memory model to our own. HPTs include limited support for GPUs but focus instead on higher level constructs such as array distributions and automatic data movement. Sequoia [21], [22] also provides a strong hierarchical model of memory, but imposes a relatively restrictive execution model in order to enable compile-time scheduling of task placement and data movement.

Many programming systems provide related constructs for creating new asynchronous tasks variously named `spawn` [23], [24], `begin` [17], and `async` [18], [19]. Runtime models such as ParalleX [25] reflect the kind of asynchronous task model that Phalanx expects to be provided by a runtime system. Phalanx departs from this existing practice in emphasizing the bulk nature of task creation. Rather than constructs oriented towards creating a single thread at a time, it provides constructs oriented towards launching large, hierchically structured collections of threads.

Mechanisms for managing heterogeneity in the language are relatively less common. Phalanx embeds information about the intended processor platform in the type of its function (e.g., via `thread<sm>` or `thread<x86>`). Thrust [3] uses a broadly similar approach, using special "tag" parameters to denote target platforms in the parameter list of its internal interfaces. CUDA [9] decorates functions with `__host__`, `__global__`, and `__device__` modifiers. These serve a similar purpose, but do not become part of the type signature of the function. OpenCL needs no such mechanism because it strictly separates the host and kernel languages and hence the host and kernel programs.

VII. Conclusion

We have presented the design of the Phalanx programming model, and briefly described the techniques we have used in building a prototype implementation. Phalanx provides a task model built around hierarchically structured collections of threads and hierarchical representations of the machine. Its hierarchical organization of threads provides a natural scoping mechanism for shared resources, such as memory and barriers, and its hierarchical machine model allows programmers to control the placement of tasks and data at a granularity of their choosing. Phalanx also introduces a scheme for managing processor heterogeneity by statically embedding information about the target platform in the type signature of the program's functions.

Phalanx is designed so that it can be embedded in C++. The only compiler support it requires is provided by NVIDIA's widely available CUDA compiler. It requires only a very thin interface on top of standard runtime components, which in the case of our prototype are CUDA, OpenMP, and GASNet. Consequently, the runtime overhead of using Phalanx is negligible compared to programs using the underlying runtimes directly, and Phalanx programs can freely interoperate with such code as desired. Finally, we have shown that this model can be used to program both single-processor, shared memory and multi-node, distributed memory machines.

While the current incarnation of Phalanx is already a useful programming system, there are a number of directions in which future work could improve both the programming model and its implementation. Our prototype design is fundamentally limited by the limitations of the runtimes on which it is built. As observed in Section IV-B, our use of OpenMP results in unnecessarily synchronous execution of asynchronous tasks. Exploring alternatives runtimes, such as TBB, more closely aligned with the Phalanx constructs might offer worthwhile performance improvements. Our distributed memory experiments indicate that a lack of collectives for broadcast limit the scaling of the SpMV code. Incorporating suitable collective operations will be an important extension of the programming model. We have also, by design, limited the opportunities for inter-task communication. This provides the underlying system with greater flexibility in scheduling tasks, but a more flexible mechanism for inter-task communication might prove valuable.

Acknowledgment

This research was funded in part by the DARPA UHPC program under contract HR0011-10-9-0008. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, and the National Energy Research Scientific Computing Facility (NERSC) at Lawrence Berkeley National Laboratory, which are supported by the Office of Science of the U.S. Department of Energy under contracts DE-AC02-06CH11357, DE-AC05-00OR22725, and DE-AC02-05CH11231, respectively.

References

[1] M. Garland and D. B. Kirk, "Understanding throughput-oriented architectures," *Commun. ACM*, vol. 53, pp. 58–66, Nov. 2010.

[2] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, pp. 7–17, Sep. 2011.

[3] N. Bell and J. Hoberock, "Thrust: A Productivity-Oriented Library for CUDA," in *GPU Computing Gems, Jade Edition*, W.-M. W. Hwu, Ed. Morgan-Kaufmann, 2011.

[4] D. Bonachea, "GASNet specification," University of California, Berkeley, Tech. Rep. CSD-02-1207, October 2002.

[5] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "Treadmarks: Distributed shared memory on standard workstations and operating systems," in *Proc. 1994 Winter USENIX Conference*, 1994, pp. 115–131.

[6] R. van de Geijn and J. Watts, "SUMMA: scalable universal matrix multiplication algorithm," *Concurrency and Computation: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[7] "BLAS Home Page," http://www.netlib.org/blas/.

[8] "Intel Math Kernel Library Reference Manual," http://www.intel.com/software/products/mkl/.

[9] *NVIDIA CUDA C Programming Guide*, NVIDIA Corporation, Apr. 2012, version 4.2. [Online]. Available: http://www.nvidia.com/CUDA

[10] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[11] F. Bodin and S. Bihan, "Heterogeneous multicore parallel programming for graphics processing units," *Sci. Program.*, vol. 17, no. 4, pp. 325–336, Dec. 2009.

[12] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," in *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2009, pp. 101–110.

[13] M. Wolfe, "Implementing the PGI accelerator model," in *Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10)*. New York, NY, USA: ACM, 2010, pp. 43–50.

[14] T. D. Han and T. S. Abdelrahman, "hiCUDA: High-level GPGPU programming," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 78–90, Jan. 2011.

[15] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," IDA, Tech. Rep. CCS-TR-99-157, May 1999. [Online]. Available: http://www.gwu.edu/ upc/publications/upctr.pdf

[16] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, and T. Wen, "Parallel languages and compilers: Perspective from the Titanium experience," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 266–290, Aug. 2007.

[17] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[18] 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," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 519–538, Oct. 2005.

[19] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: The new adventures of old X10," in *Proc. 9th Int'l Conference on Principles and Practice of Programming in Java*. New York, NY, USA: ACM, 2011, pp. 51–61.

[20] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: a portable abstraction for task parallelism and data movement," in *Proc. 22nd Int'l Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 172–187.

[21] 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 *Proc. 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006.

[22] M. Bauer, J. Clark, E. Schkufza, and A. Aiken, "Programming the memory hierarchy revisited: supporting irregular parallelism in Sequoia," in *Proc. 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 13–24.

[23] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *ACM SIGPLAN Notices*, vol. 30, no. 8, pp. 207–216, Aug. 1995.

[24] C. E. Leiserson, "The Cilk++ concurrency platform," in *Proc. 46th Annual Design Automation Conference (DAC '09)*. New York, NY, USA: ACM, 2009, pp. 522–527.

[25] H. Kaiser, M. Brodowicz, and T. Sterling, "ParalleX: An advanced parallel execution model for scaling-impaired applications," in *Proc. 2009 Int'l Conference on Parallel Processing Workshops*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 394–401.