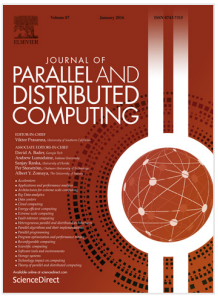
Accepted Manuscript



Automatic translation of MPI source into a latency-tolerant, data-driven form

Tan Nguyen, Pietro Cicotti, Eric Bylaska, Dan Quinlan, Scott Baden

PII: S0743-7315(17)30077-1
DOI: http://dx.doi.org/10.1016/j.jpdc.2017.02.009
Reference: YJPDC 3639

To appear in: *J. Parallel Distrib. Comput.*

Received date: 5 July 2016
Revised date: 31 January 2017
Accepted date: 21 February 2017

Please cite this article as: T. Nguyen, P. Cicotti, E. Bylaska, D. Quinlan, S. Baden, Automatic translation of MPI source into a latency-tolerant, data-driven form, *J. Parallel Distrib. Comput.* (2017), http://dx.doi.org/10.1016/j.jpdc.2017.02.009

Automatic Translation of MPI Source into a Latency-tolerant, Data-driven Form

Tan Nguyen[1], Pietro Cicotti[1], Eric Bylaska[2],
Dan Quinlan[3] and Scott Baden[1]
Email: nnguyenthanh@eng.ucsd.edu

Abstract

Hiding communication behind useful computation is an important performance programming technique but remains an inscrutable programming exercise even for the expert. We present Bamboo, a code transformation framework that can realize communication overlap in applications written in MPI without the need to intrusively modify the source code. We reformulate MPI source into a task dependency graph representation, which partially orders the tasks, enabling the program to execute in a data-driven fashion under the control of an external runtime system. Experimental results demonstrate that Bamboo significantly reduces communication delays while requiring only modest amounts of programmer annotation for a variety of applications and platforms, including those employing co-processors and accelerators. Moreover, Bamboo's performance meets or exceeds that of labor-intensive hand coding. The translator is more than a means of hiding communication costs automatically; it demonstrates the utility of semantic level optimization against a well-known library.

1. Introduction

At present, distributed-memory systems have evolved to a sophisticated level that requires applications to be heavily optimized to harness all resources provided by the hardware. An important consideration is how to minimize communication overheads in tandem with improvements in computational rates. There are two approaches to reducing communication overheads: tolerate them [56, 30, 58, 5, 61, 60, 33, 23, 21, 34, 59, 41, 22, 39, 47] or avoid them [57, 6, 43]. It is also possible to use both approaches in the same application. In this paper, we discuss an automated translation strategy that implements the first

[1]Department of Computer Science and Engineering, University of California, San Diego
[2]Environmental Molecular Sciences Laboratory, Pacific Northwest National Laboratory, Richland, WA 99354 USA
[3]Center for Advanced Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94550 USA

approach. We describe a domain-specific solution that applies to MPI applications. Since MPI is the *de facto* standard for distributed-memory programming, our approach has a broad application space.

Although MPI enables one to write communication tolerant code, it does not support the activity. For example, MPI provides immediate mode communication to express split phase algorithms [65, 38, 17], a common technique for masking communication overheads under computations. However, it doesn't assist the programmer in pipelining and scheduling computation and communication, nor how to manage a sufficiently large pool for work needed to realize overlap (e.g. processor *virtualization* or *overdecomposition* in Charm++ [35]). As a result, policy decisions affecting performance become entangled with the application, greatly affecting code development time and performance robustness. Such requirements impose a significant burden on the domain-science focused programmer who will usually defer to the expert.

To address obstacles to realizing communication overlap on high-end systems, we have developed a source-to-source translation framework, called *Bamboo* [47, 45, 46]. Bamboo transforms applications written in subset of MPI into a data-driven form that overlaps communication with computation automatically. Unlike other approaches [41, 16] that offer an explicit data-driven model, we use information about communication operations embedded in an MPI program to reason about the data dependencies among processes in order to improve performance. Armed with such knowledge, Bamboo can reformulate MPI source that doesn't overlap communication with computation into a task dependency graph representation that realizes overlap. The graph maintains a partial ordering over the execution of tasks of the graph, and the program executes in a dataflow-like fashion under the control of an external scheduler, which can overlap communication with computation automatically.

The Bamboo translation framework includes a programming model and a source-to-source translator. The programmer annotates the application with program directives, which inform Bamboo's transformations. Compared to the conventional split phase technique, these transformations not only realize overlap but also prevent policy decisions from becoming intertwined with the application. The effect is to insulate application logic from technological change, allowing the original code to continue to run correctly and to retain its familiar code structure.

The Bamboo software stack comprises 2 layers: *core message passing* and *utility layers*. The core layer transforms a minimal subset of MPI point-to-point routines (Bamboo does not support MPI's one sided communication), whereas the second translates a subset of higher level MPI functionality into equivalent point-to-point encodings, which will be then translated by the core layer. This multi-layer design allows one to customize MPI collectives, which may benefit from a specialized interconnect topology or be tailored to the application [14].

Though Bamboo supports only a subset of MPI, we have found that it can improve the performance of a wide range of applications taken from well-known application motifs on diverse platforms. We ran at scale on Edison (Cray XC30) and Hopper (Cray XE6) systems at the National Energy Research Sci-

entific Computing Center (NERSC) and on the Stampede system at the Texas Advanced Computing Center (TACC), which has advanced node architectures based on NVIDIA's Kepler and the Intel's Phi. We evaluated Bamboo against basic MPI and hand optimized code variants written by an expert to overlap communication with computation. Bamboo consistently realized a significant reduction in communication delays of the basic MPI variant. We observed that performance of applications translated with Bamboo met or exceeded that of the hand optimized code variants requiring only modest amounts of user annotation.

The remainder of the paper is organized as follows. Sec. 2 presents the Bamboo source-to-source translation framework. Sec. 3 discusses the design and implementation of the Bamboo source-to-source translator. Next, Sec. 4 presents experimental validation for various applications. Sec. 5 presents Bamboo support on advanced node technologies. Sec. 6 reviews related work. Sec. 7 concludes the paper and presents future work.

2. Bamboo

2.1. Motivation

Scalable applications are generally written under the SPMD (Same Program Multiple Data) model, and message passing has been the preferred vehicle for over two decades. The Message Passing Interface (MPI) [44] accounts for the lion's share of scalable application software, which may employ the two tier *MPI+X* programming model to unfold node level parallelism via OpenMP, CUDA or OpenCL.

MPI enables the application programmer to cater optimizations that benefit performance using heuristics, in particular, involving data motion and locality. Such domain specific knowledge is difficult to capture via general-purpose language constructs and associated compilation strategies that are unaware of application and library semantics and this helps explain the persistence of MPI. The MPI software community has been prolific, building a large body of knowledge and experience for writing high quality application software and tools. This knowledge and experience holds important clues for optimizing high performance applications. This observation motivates the design of Bamboo: a custom translator tailored to the MPI interface that effectively treats the API's members as primitives in an embedded domain specific language.

Bamboo extracts data and control dependencies from the pattern of MPI call sites and constructs a task precedence graph corresponding to the partial ordering of tasks. These tasks execute according to dataflow semantics [4, 24]. A dataflow model has two appealing attributes. First, it can automatically mask data motion costs and hence improve performance without programmer intervention [7, 21, 34, 59, 20, 22]. Second, it simplifies code development and maintenance by separating concerns surrounding policy decisions (e.g., scheduling) from program correctness. Since static analysis is not sufficient to infer matching sends and receives in a running program [48], Bamboo requires some modest amounts of programmer annotation of the original MPI program.

2.2. The Bamboo Programming Model

To illuminate our discussions about translation under Bamboo, we will use a simple example: an iterative finite difference solver for Laplace's equation in two dimensions (Fig. 1). The MPI implementation partitions the solution meshes across processors, introducing data dependencies among adjacent mesh elements that straddle the boundaries between subproblems assigned to different processors. To treat these data dependencies, the solver stores copies of off-processor boundary values in *ghost cells.* Since a conventional compiler will ignore the annotations, the code in Fig. 1 is also a legal MPI program. We next describe Bamboo's underlying programming model and its annotations.

```
1 Compute processID of left/right/up/down processors
2 for it = 1 to num_iterations {
3   #pragma bamboo olap
4   {
5     #pragma bamboo receive
6     { MPI_Irecv(RecvGhostcells) from left/right/up/down
7     }
8     #pragma bamboo send
9     { Pack boundary values to SendGhostcells
10      MPI_Isend(SendGhostcells) to left/right/up/down
11    }
12    MPI_Waitall();
13    Unpack RecvGhostcells
14    for j = 1 to N/Nprocs_Y - 2
15      for i = 1 to N/Nprocs_X - 2
16        V[j,i]=(U[j-1,i]+U[j+1,i]+U[j,i-1]+U[j,i+1])/4
17    swap(U,V)
18  }
19 }
20 free U, V, SendGhostcells, RecvGhostcells
21 MPI_Finalize();
```

Figure 1: Annotated MPI program for 2DJacobi. For purposes of clarify, some code has been omitted.

A Bamboo program is a legal MPI program, augmented with one or more code regions called *olap-regions* as shown in Fig. 1. An *olap-region* is a section of code containing communication to be overlapped with computation. The entry into an *olap-region* is called an *evaluation point*, where a task either continues or it yields control to another task because the required input data is not yet available. Receive operations residing within an *olap-region* will be included in the *input window* corresponding to the *evaluation point* of the olap-region. Bamboo preserves the execution order of *olap-regions*, which a task runs sequentially, one after the other. However, there is no implicit barrier at the exit of an *olap-region.* This allows a task to exit an *olap-region* and continue executing until it reaches the next *olap-region*, which it can enter if all the inputs defined by the corresponding *evaluation point* and *input window* are ready.

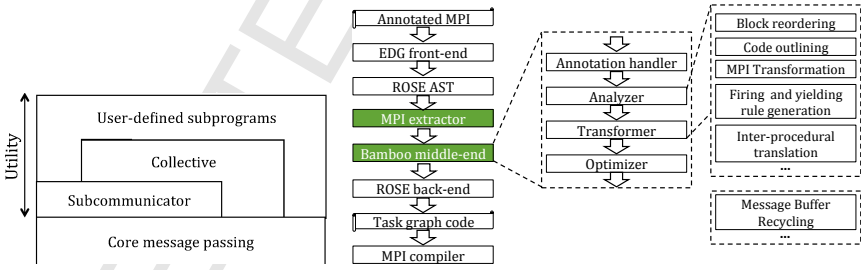
Within an *olap-region*, send and receive calls are grouped within enclosing communication blocks. All code appearing within an overlap region must be properly enclosed in a communication block, of which there are two kinds: *send* and *receive.* A *send* block contains *Sends* (MPI_Send and MPI_Isend) only.

In most cases, a *receive* block contains *Recvs* (MPI_Recv and MPI_Irecv) only, except for the following situation. If a *Send* consumes data obtained from a prior *Recv* (read after write dependence), then it has to reside within an appropriate *receive* block, either the same block as the Recv, or a later one.

Communication blocks specify a partial ordering of communication operations at the granularity of a block, including associated statements that set up arguments for the communication routines, e.g. establish a destination process. While the statements within each block are executed in order, the totality of the statements contained within all the send blocks are independent of the totality of statements contained within all the receive blocks. This partial ordering enables Bamboo to reorder send and receive blocks. For example, Bamboo can move all send blocks up front and outside of the *olap-region*, enabling all outputs to be sent out to fulfill inputs from the current olap-region onwards. Bamboo does not reorder blocks of the same type. However, because a Bamboo program executes asynchronously, inputs can arrive in any order, as they can be buffered upon arrival and then injected into tasks in the order specified by the programmer.

3. Implementation

For the sake of portability, we split the Bamboo translator into 2 software layers as shown in Fig. 2(a). The lower level layer consists of a minimal set of MPI point-to-point primitives, hence the name *core message passing*. An implementation of this layer highly depends on a runtime system that executes the generated task graph program. We will present the execution model and the implementation of the runtime system in Sec. 3.2. On top of the *core message passing* layer, we implement a *utility* layer, which supports a substantially richer set of MPI routines, including communicator splitting and collective operations.



(a) The Bamboo software stack (b) Bamboo uses the ROSE compiler framework

Figure 2: The Bamboo design and implementation

3.1. MPI Subset

Bamboo supports an important subset of MPI used in a wide range of applications: point-to-point operations (Send/Isend/Recv/Irecv/Wait/WaitAll);

a variety of collectives (see Table. 2); communicator splitting, MPI status, derived datatypes (*struct*, *contiguous* and *vector*). Bamboo does not support one-sided communication currently. Since Bamboo's runtime requires that task graphs be run time static structures (Sec.3.2). Bamboo does not support MPI dynamic process creation. It can, however, support dynamic adaptive meshes, which are treated successfully with dynamic process creation. However, Bamboo would not be applicable to graph algorithms, for example, that employed find grained communication, dynamic process creation, or both.

3.2. Runtime System

A Bamboo program runs as a set of *tasks* coupled by data dependencies. The program can over-decompose the problem, creating more tasks than the number of processing cores. The task scheduling and communication handling jobs are handled by Bamboo's runtime system called Tarragon [21, 20].

3.2.1. Task scheduling

Tasks have state, and this state is used to manage task execution. A typical Bamboo task spends most of its time circulating among the following 3 states: eXecuting (X), Waiting (W) or Runnable (R) as shown in Fig. 3. A waiting task will become runnable when all inputs are ready. The collection of all inputs of a task is represented by task's *firing rule*, which is visible to the runtime. Tasks are executed by *workers*. A runnable task will start executing when the runtime identifies an available *worker*. Since a task cannot execute unless it has first become runnable, there is no explicit message waiting within a task; this activity is factored out of task execution and handled via a callback made by the runtime. Additional task state variables can be defined by Bamboo. For example, we use task state to control iterative methods, folding the iteration inside the tasks. Tasks and Task Graphs are run time static entities. Thus, once instantiated, their structure, including tasks dependencies, can't change at run time.

Currently Tarragon uses Pthreads to implement *workers*. The runtime can be configured to have either a shared task queue among all *workers* or multiple private queues. The former configuration allows the workload to be easily balanced but also incur some overhead for memory protection.

3.2.2. Communication handler

As a task is running, it produces data. Tasks can register certain data with the runtime as outputs, which enable other tasks to become runnable (R) and ultimately execute (X). Tasks will not block when producing outputs. Instead, outputs will be buffered and processed by the communication handler. Data sent out from a task will not become visible at the destination until the recipient has entered the W state. When a task is in the R or X state, incoming messages are hidden by the runtime system, in the order they were received, and only be made visible when the task enters the W state again.

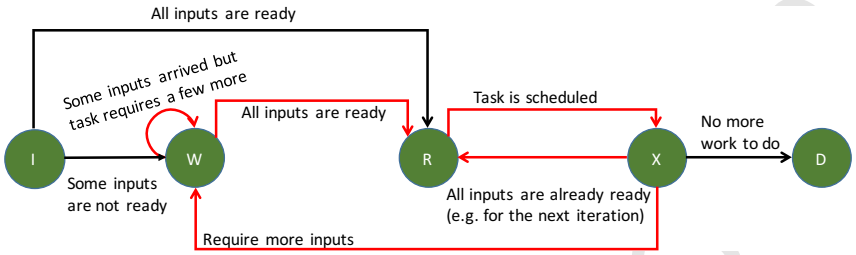


Figure 3: Except for the initial state (I) and the final state (D), a task circulates among 3 states W, R, and X. Tasks never wait for inputs and hold computing resource at the same time. In addition, tasks only receive new inputs at the W state.

Currently, Tarragon uses MPI to implement the communication handler. It uses non-blocking routines (MPI_Isend, MPI_Irecv, and MPI_Test) to handle inter-process communication requests. For intra-process communication requests, outputs of a task are injected directly to the recipient. To keep the communication handler responsive to requests, the runtime can dedicate a processor core to run the handler.

3.3. Translation: core message passing layer

3.3.1. Block reordering

To use Bamboo, the MPI source is annotated with *olap-regions*, each consisting of communication blocks, *send* and *receive* blocks, which further contain MPI function calls. Due to the way in which the generated code executes, Bamboo performs an intermediate code transformation called *block reordering*. Bamboo will reorder certain *communication blocks* in certain situations. For example, the left side of Tab. 1 shows a common communication pattern used in MPI applications that will be restructured by Bamboo. Specifically, Sends (in the *send block*) issued by a process match up with Recvs (in the *receive block*) of the other process encoded in the same iteration. Bamboo has to reorder the *send block* due to the following reason. A task is *runnable* only when all necessary data is available. If we place the corresponding send within the same iteration as the corresponding receive, data sent in one iteration will not be received until the next. But, the algorithm needs to receive data within the same iteration. To cope with this timing problem, Bamboo reorders the send block, advancing it in time so that the sending and receiving activities reside in different iterations. Bamboo will set up a pipeline, replicating the send block to the front of, and outside, the iteration loop. It also migrates the existing call to the end of the loop body, adding an appropriate guard derived from the loop iteration control logic. After reordering, the transformed code appears as shown in the right side of Tab. 1. The matching send and receive blocks now reside in different iterations. We next present how Bamboo reinterprets all MPI calls produced in this phase as task's inputs and outputs.

Before reordering	After reordering

1 #pragma bamboo olap
2 **for** (i=1;i<=nIters;i++){
3 #pragma bamboo receive
4 {MPI_Irecv(rbuf0, ...);
5 MPI_Irecv(rbuf1, ...); }
6 #pragma bamboo send
7 {MPI_Send(sbuf0,...);
8 MPI_Send(sbuf1,...); }
9 MPI_Waitall(...);
10 Compute block
11 }
	1 i=1
	2 **if**(i<=niters){
	3 MPI_Send(sbuf0,...);
	4 MPI_Send(sbuf1,...);
	5 }
	6 #pragma bamboo olap
	7 **for** (;i<=nIters;){
	8 #pragma bamboo receive
	9 {MPI_Irecv(rbuf0, ...);
	10 MPI_Irecv(rbuf1, ...); }
	11 MPI_Waitall(...);
	12 Compute block
	13 i++
	14 **if**(i<=niters){
	15 MPI_Send(sbuf0,...);
	16 MPI_Send(sbuf1,...);
	17 }
	18 }

Table 1: An intermediate code transformation that reorders blocks of code. Left: a typical MPI input program that requires code reordering. Sends within the send block of a process match with receives within the receive block of another process in the same iteration. Right: The same code with send reordered. Note that the replicated MPI_Send calls will not pose a deadlock issue. Bamboo will reinterpret MPI calls later, allowing the generated code to work correctly.

3.3.2. MPI reinterpretation

Bamboo translates MPI calls into task methods. For MPI_Comm_rank and MPI_Comm_size, Bamboo simply rewrites these routines to corresponding method invocations that return the task ID and number of tasks in the graph, since over-decomposition does not change the communication pattern. For MPI_Send and MPI_Isend, Bamboo creates a message and copies communicated data from the outgoing buffer into the data buffer of the message. Bamboo then generates a signal code notifying the communication handler of the runtime system that the output data is ready to be sent out. MPI_Recv and MPI_Irecv, however, are handled in a different way since tasks do not explicitly invoke any method to receive data from other tasks. Instead, the runtime receives and buffers incoming messages. When a task is scheduled to execute, it can pull these messages from the runtime. Bamboo uses all *Recv* calls within an *olap-region*, together with any conditional statements connected with them, to generate firing rule which decides when a task becomes executable. Details of this implementation will be discussed in Sec. 3.3.3. Beside using MPI_Recv and MPI_Irecv to generate firing rules, Bamboo also uses the information encoded in these routines to generate code that pulls messages from the runtime system. Specifically, messages are sorted by source, destination, and tag. Bamboo transforms this information into queries to the runtime system. Finally, for MPI_Wait and MPI_Waitall Bamboo simply removes these calls since each *olap-region* requires all inputs to be available before it can execute.

3.3.3. Firing rule and yielding rule

As previously mentioned, the task state often changes from X into W and from W to R. The condition that determines when the runtime can enable a transition from W to R is called the *firing rule*. Upon scheduled to execute, the task state changes from R to X. The formula that the runtime uses to reverse the state transition from X into W is called the *yielding rule*. Bamboo extracts information from MPI receive calls and associated conditional statements to generate *firing* and *yielding rules*.

Let m and C be, respectively, a message possibly received by an MPI process in an *olap-region* and the associated conditional statements. Whether a particular process should wait for message m or not is subject to the evaluation of the condition C. Thus, the *firing rule* for an *olap-region* can be written in the conjunctive normal form.

$$\bigwedge(\neg C_i \bigvee m_i) \tag{1}$$

On the contrary, we express the *yielding rule* in disjunctive normal form, where i ranges from 1 to the number of messages possibly received by a process in an *olap-region* and m_i is true means that message i has arrived.

$$\bigvee(C_i \bigwedge \neg m_i) \tag{2}$$

3.3.4. Inter-procedural translation

The code transformation and analysis modules of Bamboo may need to span procedure boundaries. For instance, Fig. 4 gives an example where the source codes of an *olap-region* and its *communication blocks* (i.e. *send block* and *receive block*) reside in different procedures. To generate *firing* and *yielding rules* for the *olap-region*, the translator needs information in the *receive block*. Inlining is a technique that exposes the calling context to the procedure's body and the procedure's side effect on the caller. Bamboo performs inlining, and the process is as follows. If a procedure directly or indirectly invokes MPI calls, Bamboo registers it as an *MPI-invoking procedure*. Bamboo will subsequently inline all *MPI-invoking procedures* from the lowest to the highest calling levels. The inlining process is transparent to the programmer and does not require any annotation. However, due to the static nature of the strategy Bamboo currently does not support recursive procedures. This requires a redesign of the graph library and run time system.

3.4. Translation: utility layer

To handle MPI functionality outside the core message passing substrate, we take a library-based approach that allows system providers and MPI programmers to easily translate a custom implementation of these routines into a task graph form. Bamboo includes default implementations of commonly used primitives.

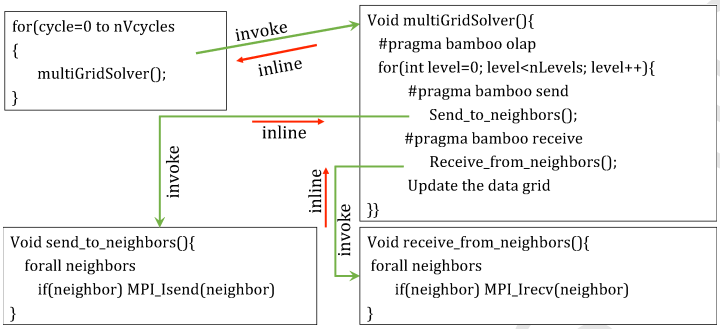


Figure 4: A multigrid solver with call chains containing MPI invocations. Bamboo registers procedures that directly or indirectly invoke MPI calls as an *MPI-invoking procedure*. It then inlines all *MPI-invoking procedures* from the lowest to the highest calling levels.

3.4.1. Collectives

Bamboo maintains a library implementing widely used collectives, by breaking them down into their point-to-point components. The source-to-source translator will automatically detect non-point-to-point MPI calls in the MPI input source and inline corresponding implementations into the program's source code before translating these codes together into a task graph form. Bamboo employs the AST merge mechanism provided by the ROSE compiler framework. This mechanism allows the ASTs generated from source codes in different files to be merged into a single AST. Tab. 2 shows algorithms that Bamboo uses to implement common collectives and the corresponding latency and bandwidth costs.

Collective API	Algorithm	Complexity
MPI_Barrier	Bruck's algorithm	$\lceil lgP \rceil \alpha$
MPI_Bcast	Binomial Tree	$\lceil lgP \rceil (\alpha + s\beta)$
MPI_Reduce	Binomial Tree	$\lceil lgP \rceil (\alpha + s\beta +$ size*opCost)
MPI_Allreduce	Recursive doubling	$\lceil lgP \rceil (\alpha + s\beta +$ size*opCost)
MPI_Scatter	Binomial Tree	$\lceil lgP \rceil \alpha +$ totalSize*β
MPI_Gather	Binomial Tree	$\lceil lgP \rceil \alpha +$ totalSize*β
MPI_Allgather	Bruck's algorithm	$\lceil lgP \rceil \alpha +$ totalSize*β
MPI_Alltoall	Bruck's algorithm	$\lceil lgP \rceil (\alpha + \frac{s}{2}\beta)$

Table 2: Default Bamboo implementation of collective operations. We use the $\alpha\beta$ model to estimate the cost of collective operations, where α is latency and β is inverse bandwidth [62].

3.4.2. Communicators

An MPI Communicator is a namespace describing the set of MPI processes that each process can communicate with for a particular MPI routine. *MPI_COMM_WORLD* is the only predefined communicator in the MPI environment, defining the order of all processes of an MPI program. Bamboo currently supports *MPI_Comm_split*, which partitions an existing communicator into multiple disjoint groups, reorders MPI ranks, or both. New communicators can be

further split in the same way. We support *MPI_Comm_split*'s color-key filtering mechanism which relies on a many-to-many mapping from the task ID set into the color set, where the color set is normally smaller than the task ID set. We also use *key*, a one-to-one mapping to sort tasks within a common color set. Bamboo implements the *MPI_Comm_split* routine using MPI point-to-point primitives as follows. All MPI processes in the existing communicator exchange information of *color*, *key*, and the corresponding rank in *MPI_COMM_WORLD*. Eventually each process holds information of the other processes. Based on the information retrieved from others, each process filters out processes with the same color. Such processes will be sorted on *key* before being assigned a new rank in the new communicator. Once the new communicator has been created, a communicator name and a process rank within the communicator will be sufficient to locate the corresponding rank in *MPI_COMM_WORLD*.

4. Results

In this section, we describe computational results with 4 applications on various platforms: NERSC's Hopper and Edison platforms, and TACC's Stampede platform, using nodes that include NVIDIA Kepler K20 GPUs. In order to assess the performance benefits of Bamboo, we built a set of variants for each application. The first variant, *MPI-basic,* is the simplest implementation that does not overlap communication with computation. All remaining variants are obtained from *MPI-basic.* The *Bamboo* variant was obtained by translating MPI-Basic with Bamboo. *MPI-Olap* was obtained by restructuring the application to overlap communication with computation via split phase coding. The third variant, *MPI-Olap,* has been manually restructured to employ split phase coding to overlap communication with computation. The fourth variant, *MPI-nocomm* was obtained by suppressing communication in the code, and is a loose upper bound on the potential performance benefit of overlapping communication with computation.

In some applications running on CPUs, we found it advantageous to use a mixed mode model MPI+OpenMP rather than "flat" MPI. To express variants based on this approach, we use an intuitive notation e.g. *MPI+OMP-ncomm* is an MPI+OMP code variant in which communication has been suppressed.

4.1. Dense linear algebra

Dense linear algebra is a class of computations on matrices where all elements are stored explicitly. Typically, this class of applications delivers a high fraction of peak processor performance. Thus, the overall performance will become much more sensitive to communication overheads as computing capability is expected to be substantially increasing in years to come. In this section, we evaluate Bamboo using matrix multiplication and matrix factorization, two operations commonly used as building blocks in dense linear algebra problems.

4.1.1. 2.5D Cannon's algorithm

2.5D Cannon (AKA *communication avoiding,* or CA) matrix multiplication algorithm [57] targets small matrices. Small matrix products arise, for example, in electronic structure calculations (e.g. *ab-initio molecular dynamics* using planewave bases [42, 14]). At a high level, the 2.5D algorithm generalizes the traditional 2D Cannon algorithm [50] by employing an additional process dimension to replicate the 2D process grid. The degree of replication is controlled by a replication factor called c. When c=1, we regress to 2D Cannon. When $c = c_{max} = nprocs^{1/3}$, we elide the shifting communication pattern and employ only broadcast and reduction. This algorithm is referred to as the 3D algorithm. The sweet spot for c falls somewhere between 1 and c_{max}, hence the name 2.5D algorithm. As in the 2D algorithm, the 2.5D algorithm shifts data in the X and Y directions. In addition, the 2.5D algorithm performs a broadcast and a reduction along the Z dimension.

Through experimentation, we observed that, for the small matrices targeted by the 2.5D algorithm, the hybrid execution model MPI+OMP yields higher performance than a pure MPI implementation, which spawns only one MPI process per core. Therefore, we used the following 3 variants: *MPI+OMP*, *MPI+OMP-olap*, and *Bamboo+OMP*. All variants perform communication at the node level, using the OpenMP interface of the ACML math library to multiply the submatrices (dgemm). *MPI+OMP* is the basic MPI implementation without any overlap. *MPI+OMP-olap* is the optimized variant of *MPI+OMP* that pipelines computations of a step of the algorithm with communication for the next step. *Bamboo+OMP* is the result of passing the annotated *MPI+OMP* variant through Bamboo. As with the previous two applications, we also present results with communication shut off in the basic variant, i.e. *MPI+OMP-nocomm*, which uses the same code as *MPI+OMP*. We conducted a weak scaling study on 4K, 8K, 16K and 32K cores on Hopper. We chose problem sizes that enabled us to demonstrate the algorithmic benefit of *data replication*.

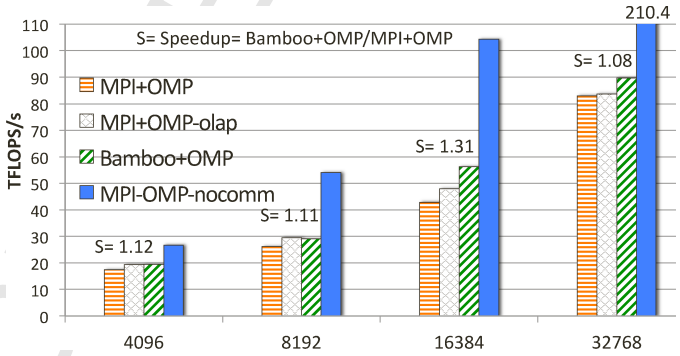


Figure 5: A weak scaling study on the 2.5D Cannon algorithm. We ran codes on up to 32K cores on Hopper. We used small matrices (N=20668 on 4K cores).

Fig. 5 shows the results with the different variants. Both *Bamboo+OMP* and

#Cores	4K	8K	16K	32K
MPI+OMP	c= {**1**, 4}	c= {**2**, 8}	c= {1, **4**, 16}	c={**2**, 8}
MPI+OMP-olap	c= {**1**, 4}	c= {**2**, 8}	c= {1, **4**, 16}	c={**2**, 8}
Bamboo+OMP	c=2, VF =8	c=2, VF=4	c=2, VF=2	c=4, VF=2

Table 3: The effects of replication and virtualization. The *MPI+OMP* and *MPI+OMP-olap* code variants have limited options for c. The boldface values within the curly braces yield the highest performance.

MPI+OMP-olap deliver the same speedup over the *MPI+OMP* variant on up to 8K cores. With 16K cores or more, *Bamboo+OMP* overtakes *MPI+OMP-olap*. Although *Bamboo+OMP* is still faster than the other variants on 32K cores, the speedup provided by *Bamboo+OMP* has dropped. We believe this behavior is the result of an interaction between the allowable replication factor c, and the degree of virtualization v.

To understand the interaction, we first look at Tab. 3, which shows the values of c that maximize performance for the different variants. Note that the 2.5D algorithm requires that the first two dimensions of the processor core geometry must be equal. For the two MPI variants, the available values for the replication factor c are limited while *Bamboo+OMP* has more options due to the flexibility offered by virtualization. For example, on 8K cores *MPI+OMP* and *MPI+OMP-olap* can set $c = 2$ or $c = 8$, i.e. other values are illegal. On 16K cores, c can be 1, 4 or 16 while on 32K cores c can take on values of 2 or 8. For *Bamboo+OMP*, performance depends not only on our choice of c but also on the degree of virtualization v. Thus, we choose a combination of replication and virtualization that is optimal and cannot choose these parameters independently. As a result, performance is not stable as we grow the number of cores. The benefit of the 2.5D algorithm is that the communication volume shrinks with c and p. The cost is $O(n^2/\sqrt{cp})$, where p is the number of cores. However, the effect of increasing v doesn't benefit from this cost function, since communication among virtualized tasks must be performed serially (hence p doesn't effectively change in that formula.) The effect is to improve pipelining as in the other applications. However, the number of messages is $O(\sqrt{p}/c^{2/3} + log(c))$. It will grow as we increase v, because the message starts are serialized. The effect is to damp c as v increases, and this is evident from the data in Tab. 3.

4.1.2. High Performance Linpack

The High Performance Linpack benchmark (HPL or Linpack for short) [27, 26, 25] is a well-known benchmark that solves a dense system of linear equations using LU factorization, and is often used to measure the performance of supercomputers. HPL uses a blocked cyclic data decomposition scheme. The HPL benchmark comprises 2 code variants. *Pdgesv0* does not make any attempt to overlap communication with computation, whereas *pdgesvK2* applies an overlapping technique called *lookahead*. We applied Bamboo annotations to *pdgesv0*. Details of the 3 code variants are as follows.

The *pdgesv0* code consists of 3 key operations: panel factorization *pFact*, panel broadcast *pBcast*, and the trailing submatrix update *pUpdate*. *pFact*

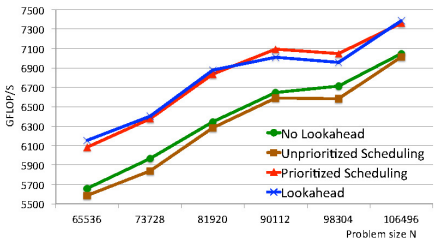
finds the *pivots* in column panel c. This step is costly since we have to factorize a skinny matrix over a subset of the processes that own the panel, including a sequence of row swap-broadcasts, one for each *pivot* within a single columns of the panel. HPL provides various panel factorization implementations, classified into recursive and non-recursive variants.

We evaluated both variants and observed no difference in performance. Thus, we used the non-recursive variants. Once the panel has been factorized it must be broadcast to column processes within the same row *(pBcast)*. This is an efficient implementation that uses a ring broadcast algorithm, shifting data to the right along column processes. The *pUpdate* operation swap-broadcasts U among row processes and then performs a *rank-1 update*. It accounts for the lion's share of LU's computational work, performing $O(N^3)$ multiply-adds. The *pdgesvK2* variant applies *lookahead* [27], a technique for overlapping communication with computation that fills idle gaps in the execution of LU. *Lookahead* utilizes the dependence structure of the blocked algorithm to orchestrate computation and data motion. It uses split-phase coding [65], and may compute multiple iterations in advance. The underlying communication structure for this synchronization is complicated and difficult to implement and follow because the application must poll for arriving data in several places in the program. These complications have prevented lookahead from being used in practice. For example, lookahead is not employed in the widely-used ScaLAPACK [8] library. This predicament has motivated new algorithmic reformulations [15] or data-driven implementations [34, 9, 15, 40] to realize overlap.

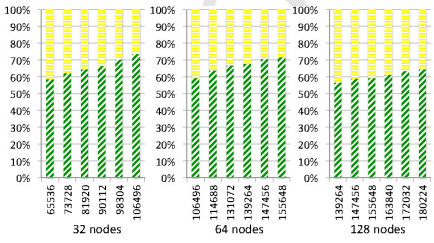
We annotated the *pdgesv0* module and translated it with Bamboo. We also added scheduling policies via task prioritization using a bamboo priority pragma [4] so that communication could be overlapped with communication more efficiently. The common wisdom in scheduling a non-preemptive task graph is that tasks should hold the core as long as they are still executable and only yield control when they need input from other tasks. This greedy strategy is intended to maintain the high hit rates of caches and TLB. However, LU factorization is an exception. Specifically, many tasks are waiting for data from the root task so that they can begin executing. Moreover, if for some reason the task that will become the next root is not scheduled soon, the next panel broadcast will be delayed. If this happens, performance could be significantly penalized since no overlap can be realized. Bamboo's *olap-regions* generally reside within an outer iteration, and HPL is no exception. Bamboo handles overlap regions as follows. When control reaches the end of an overlap region, if the priority is negative, the task yields processor/core, even if inputs are ready for the next iteration. To this end, we used 3 different values (0, -1, and 1) to represent for the priority of scheduling a task to run next. Among runnable tasks, those with higher priorities will be inserted at the top of the priority scheduling queue. Tasks with priority of 0 or 1 will execute until they cannot continue, since they

[4] We didn't present this pragma earlier since Bamboo simply translates the pragma into a method that sets task priority

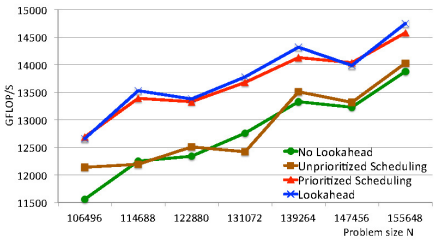
await data from other tasks that haven't yet completed. However, tasks with priority -1, and must yield the core at the end of the olap region, even if they have the data needed to continue executing. Note that this scheme is not preemptive. Neither the runtime system nor task can force another task to yield control. Depending on the availability of the input and the current priority, a task decides whether it should continue or yield processor/core to another task. For more information about how we prioritize the LU task graph, see [46].



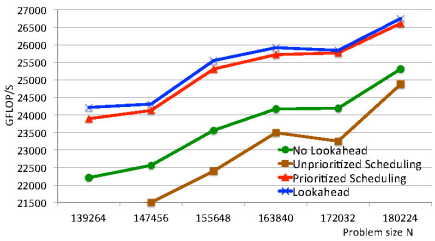
(a) 32 Stampede nodes



(b) Time distribution. In all experiments, time for dgemm (green, lower bars) varies from 57% to73%. Panel broadcast accounts for most of the rest of the time.



(c) 64 Stampede nodes



(d) 128 Stampede nodes

Figure 6: Results comparing our scheduling strategies for the transformed code without lookahead and with lookahead. Prioritization significantly improves performance, enabling our transformed code to meet the performance of lookahead for many problem sizes

We performed experiments on Stampede [1], located at the Texas Advanced Computing Center (TACC), using the Sandy Bridge processors only. We ran on up to 128 nodes (4096 cores). The results appear in Fig. 6. We chose small problems sizes to ensure that communication overhead is significant and thus we can see the benefit of overlapping communication with computation. Fig. 6 shows that Bamboo was able to meet, and sometimes slightly exceed, the performance of the painstakingly coded *lookahead* variant, so long as prioritization was employed.

The vital role of task prioritization is inevitable. Theoretically, if we use a random scheduling algorithm and we run the unprioritized Task Graph variant for a large number of times, there is possibility that we observe the performance of the prioritized Task Graph variant. However, the required number of experiments could grow exponentially in $k * N$, where N is the number of panel columns of the input matrix and k is the number of communication events oc-

curring for a particular N. We repeated each experiment more than 10 times and took the best performance, but results without task prioritization were always far below the performance of *lookahead*. Compared to the *no-lookahead* variant, the performance of the unprioritized task graph was at best comparable and in some cases it was even lower.

4.2. Structured Grid-Multigrid solver

Multigrid [11, 66, 13, 28] is a family of methods to accelerate the convergence rate of conventional iterative methods such as Gauss-Seidel Red-Black and SOR. A multigrid solver consists of multiple cycles which solve an equation via a hierarchy of meshes. At each cycle, multigrid recursively solves an error equation on a coarser grid, which it uses to correct the solution. The recursion ends at some specified bottom-most level, where a *bottom solver* solves the error equation on the coarsest grid. The solution from this grid is then projected (via interpolation) up through the hierarchy of finer meshes until reaching the finest level. At this point the cycle completes. The cycle can have a V or W shape, or may be truncated at a certain level where the bottom solver can perform more efficiently.

We translated *MiniGMG,* a multigrid solver developed at Lawrence Berkeley National Laboratory [67]. This is an MPI+OpenMP code consisting of 4000 lines, 1000 of which are MPI code that need to be translated. It does not overlap communication with computation. Owing to the complexity of restructuring this third-party code by hand, we do not provide an *MPI-Olap* variant.

This solver employs *truncated V-cycles*. On the way down of each cycle, *smooths* are applied to reduce the error before *restrictions* are used to determine the right-hand side of the coarser grids. Each *smooth* is a Gauss-Seidel Red-Black relaxation (GSRB). The V-cycle is truncated when the mesh reaches the minimal size threshold of 4^3, and the bottom solver consists of a significant number of GSRB sweeps. Finally, the solution is interpolated and smoothed upward to the next finer mesh. The GSRB kernel is optimized further with a DRAM avoiding technique, which changes the communication pattern significantly. In particular, in addition to nearest neighbor communication, adjacent processes along the diagonals also communicate. The effect of this optimization is to increase the number of neighbors that a process communicates from 6 to 26.

We conducted a weak scaling study on Edison, fixing the problem size at 8^3 boxes per core. The left part of Tab. 4 shows the execution time of modules of the MPI variant. Communication (*comm*) accounts for about 20% of the total execution time, and thus we have enough available computation to hide communication. While the communication cost grows slightly as the number of cores increases, the execution time for the other activities is stable, i.e. time to update data elements (*compute*), serialize and deserialize messages (*pack/unpack*), and copy ghost cells among boxes of the same MPI process (*box copy*). The right part of Tab. 4 shows the relative overhead of communication at each grid level. It can be seen that communication overhead increases by a factor of

2 from the finest grid L0 to L1, slowly increases (L1 to L2 and L2 to L3), or saturates from L3 to the coarsest level grids, L4.

Cores	Comm	Compute	pack/unpack	box copy	Comm/total time per level				
					L0	L1	L2	L3	L4
2048	0.448	1.725	0.384	0.191	12%	21%	36%	48%	48%
4096	0.476	1.722	0.353	0.191	12%	24%	37%	56%	50%
8192	0.570	1.722	0.384	0.191	13%	27%	45%	69%	63%
16384	0.535	1.726	0.386	0.192	12%	30%	48%	53%	49%
32768	0.646	1.714	0.376	0.189	17%	28%	44%	63%	58%

Table 4: Left: execution time in seconds of different modules in the multigrid solver. Right: the relative cost of communication at each grid level (the smaller level, the finer the grid).

Fig. 7 compares the performance between MPI and Bamboo code variants in a weak scaling study. We can see that both MPI and Bamboo are highly scalable (in a weak sense) and that Bamboo improves the performance by up to 14%. These results are promising, given that overlap strategies for multigrid present three challenges. First, communication is effective at finest grids only as the message size on these grid levels is still significant. At coarser levels, the message size gets smaller and smaller, increasing the overhead of virtualization. In addition, when moving from a fine to a coarser grid, computation shrinks by a factor of 8 whereas communication reduces by only a factor of 4, reducing the efficiency of the overlapping technique. Furthermore, the number of messages that each processor has to communicate messages with its 26 neighbors is significant. This increases the processing overhead of the runtime system that manages overlap.

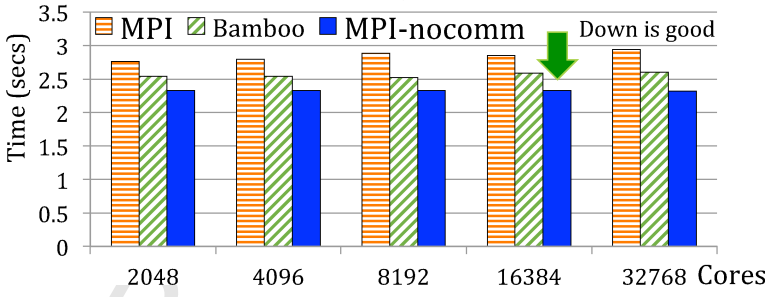


Figure 7: Weak scaling study of algebraic multigrid on up to 32,768 processor cores of Edison. At the finest level, each processor accounts for 8×128^3 boxes. Thus, in each V-cycle the finest grid size is 128^3 and the coarsest grid size is 4^3.

5. Advanced node technologies

At present, it appears that further improvements to HPC systems will mainly come from enhancements at the node level [53, 10, 52]. Node architectures are changing rapidly, and a heterogeneous design that uses *devices* (i.e. coprocessors or accelerators) to amplify node performance is gaining traction. Bamboo

supports state-of-the-art computing platforms employing advanced technologies such as Graphical Processing Units (GPUs) and Many Integrated Core (MIC). In this paper we present the results on the former. Results on MIC can be found in our previous work [45].

5.1. Graphical Processing Units

GPUs are a powerful means of accelerating compute-intensive and bandwidth-intensive applications and for lowering the power/performance ratio. CUDA (Compute Unified Device Architecture) is a well-known parallel programming model for GPUs developed by NVIDIA. Under this model, each GPU works as a *device* attached to a CPU called host. The host offloads compute kernels and dependent data to its device(s) each running thousands of CUDA threads to parallelize the workloads. The results are then collected back to the host. The host-device communication is routed over a PCIe bus, which can easily become a performance bottleneck due to its limited bandwidth.

As the demand for compute and memory increases, applications require a cluster of many GPUs. MPI+CUDA is a hybrid programming model commonly used to parallelize the application workloads across multiple GPUs. This model spawns an MPI process per GPU to work as the host. The communication between MPI processes is called host-host communication. We extend Bamboo to hide the host-host and host-device communication overheads in MPI+CUDA applications.

5.2. A GPU-aware Interface

Because MPI is not aware of device memory, Bamboo can realize overlap among hosts only. It cannot overlap data motion between host and device. We defined and implemented a GPU-aware MPI interface, which allows MPI communication routines to specify device memory as the buffer for sending and receiving message data. Since distinguishing device and host buffers is challenging at static time and costly at dynamic time, we also have the programmer specify a different MPI_COMM_WORLD communicator called CUDA_COMM_WORLD.

With a GPU-aware MPI, the programmer can manage the communication between *devices* without the need to explicitly route data via the *hosts*. Instead, the compiler and runtime system are responsible for handling the data transfer between *host* and *device*. Our proposal is similar to those proposed by MPI-ACC [2, 3] and MVAPICH2-GPU [64]. However, we integrated GPU-aware MPI with Bamboo. The result is that we can rewrite an MPI+CUDA program to a task dependency graph form, where host-device transfers are factored out of the task and are represented as edges of the graph. The runtime system can mask both host-device communication automatically using the same mechanism it uses for host-host communication [5].

[5] We note that MPI-ACC and MVAPICH2-GPU cannot realize this optimization

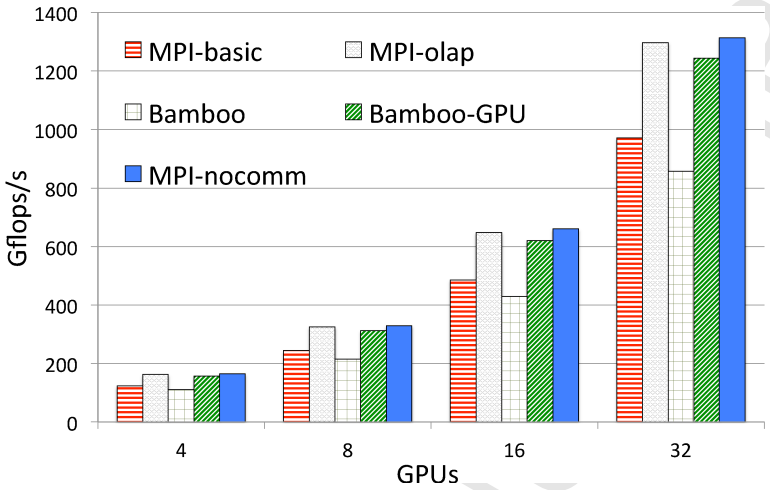


Figure 8: Weak scaling results of 3D Jacobi on up to 32 GPUs on Stampede. Bamboo-GPU outperforms MPI-basic, though it runs slightly slower than the hand optimized code.

5.3. Performance evaluation

We evaluated our GPU-aware programming model on the *3D Jacobi* solver running on a portion of Stampede containing hybrid CPU/GPU nodes. Only 32 such nodes were available at a time, so experimentation was limited to this configuration. A GPU node has a single K20 "Kepler" GPU with 5GB of fast device memory. Our applications ran out of this memory rather than on the more generous 32GB host memory, which is connected two 8-core Intel Xeon E5 "Sandy Bridge" processors. Nodes communicate via a Mellanox FDR InfiniBand interconnect. We use the Intel compiler to compile code running on the host and CUDA 5.5 to compile GPU kernel code. Mvapich handled communication among GPU nodes.

We compared 5 code variants. The first and second variants, *MPI-basic* and *MPI-olap*, employ the traditional MPI+CUDA programming model. The third variant, *Bamboo*, is the task graph program obtained by translating *MPI-basic*. The fourth variant, *Bamboo-GPU*, is generated by the Bamboo translator from a basic MPI+CUDA code written under the GPU-aware programming model, i.e. MPI-basic except with MPI data motion calls replaced by the equivalent CUDA-aware MPI and CUDA calls that transfer data between the host and device disabled. The fifth variant, *MPI-nocomm*, was obtained by removing all host-host and host-device communication calls in MPI-basic. We conducted a weak scaling study on Stampede. During normal production time, this platform supports jobs with at most 32 K20 GPU nodes, so we limited ourselves to 32 nodes. Due to this small scale, 1D decomposition scheme was sufficient to meet the needs of the application (though not scalable for larger configurations). We evaluated all code variants with a base problem size of $510 \times 512 \times 128$ per GPU, which consumes 0.765 GB of device memory per node. This problem size

is intended to mimic a more realistic application scenario, in which Jacobi would comprise one step of a multiphase algorithm. Though Jacobi uses 3 variables per mesh zone, a more realistic application would use many variables – a factor of 5 or more. Thus, a production application would consume 3/4 or more of the node's available memory and in some cases the mesh size per node would have to be reduced to avoid exceeding available memory capacity. The problem scaling size we use thus stresses communication at a level appropriate for production applications.

Fig. 8 shows the performance in GFLOP/s of all code variants. It can be seen that *Bamboo-GPU* and *MPI-olap* significantly outperform *Bamboo* and *MPI-basic*. We attribute the performance improvements of *Bamboo-GPU* compared to *Bamboo* and *MPI-basic* to the following optimizations. First, the knowledge of host-device transfer enables *Bamboo-GPU* to take advantage of locality, such as tasks computing on the same GPU only exchange the header information of messages. This optimization can save significant bandwidth of the PCI Express bus connecting *host* and *device*. We found that this optimization is very significant at small scales, where the bandwidth between host and device is more critical than between hosts. Second, we modified the runtime to use pinned memory to buffer messages. Using pinned memory can significantly increase the bandwidth between *host* and *device* [63, 18]. Third, we used asynchronous memory copies to avoid implicit synchronization on the GPU.

5.4. Future Implementation for Performance Portability

With the current implementation of our runtime system, messages among GPUs are always routed through their hosts. This policy is not optimal when all or some pairs of GPUs can communicate on a direct path. NVIDIA refers to this capability as *GPUdirect*, which can be enabled when either 1) GPUs of the same compute node share a common PCIe bus or 2) the interconection network allows the communication among GPUs on different compute nodes to bypass their hosts. Although Stampede provides neither of these, Bamboo's users may have access to GPUs clusters that have *GPUdirect* (e.g. the Comet system at San Diego Supercomputer Center). As a results, we plan to modify our implementation to support *GPUdirect* as follows.

Our runtime system employs a single MPI process per compute node to handle the communication. Thus, for inter-node comminication, we plan to use MPI implementations that support GPUdirect as the communication backend (e.g. MPI-ACC and MVAPICH2-GPU). For intra-node communication, we will need to provide our own implementation of GPUdirect. Specifically, once the runtime detects that the sender and receiver tasks locate on the same compute node, it sends the message descriptor instead of the raw data. The receiver opens the descriptor and pulls data directly from the sender using CUDA memory copy. In order to hide the communication cost, data dependency is only considered satisfied when this memory copy operation completes. We plan to use the asynchronous memory copy version so that we will not block the communication handler at the receiver side. It is worth noting that these two extensions will not require any modification on the Bamboo's programming API.

6. Related work

Danalis et al. [23] implemented transformations of MPI that realize communication overlap in collective operations. Strout et al. presented a framework for inter-procedural analysis of message-passing SPMD programs; generating MPI inter-procedural control-flow graphs that help reduce storage requirements [32]. Shires et. al [54] presented a program flow representation of an MPI program, which is useful in code optimization. β-MPI [55] generates the runtime dataflow graph of an MPI program, in order to assess communication volume. It overloads the MPI calls using macros, but does not perform source code analysis or code restructuring.

Latency tolerant applications and infrastructure for expressing them have been previously reported in the literature including Charm++ [36], KeLP2 [5, 29], Adaptive MPI [33] (built on top of Charm++), Tarragon [21, 20], Thyme [59], and others [56, 58, 61, 60, 51, 19, 30].

Charm++ supports virtualization and latency tolerance. KeLP2 is a C++ framework that supports an explicit hierarchical execution model, and masks latency. Adaptive MPI virtualizes MPI processes to support communication overlap and task scheduling. When a thread blocks on an MPI call, it yields to another thread. There is no explicit dataflow graph and the MPI source is not manipulated. Bamboo transforms MPI source into an explicit graph, which can be used to guide scheduling. Thyme is a C++ library with goals similar to Tarragon. Husbands and Yelick [34] have implemented thread-scheduling techniques for tolerating latency in dense LU factorization and use a dataflow interpretation of the algorithm that exposes the latent parallelism.

PLASMA [39] is a library for dense linear algebra and it represents applications with a dataflow graph. To conserve memory, it allocates only a portion of the graph at a time, inhibiting global optimizations. Bamboo can avoid graph expansion by controlling the outer iteration within task state.

We used the Rose source-to-source translator [49] to develop Bamboo. Rose is a member of the family of language processors that support semantic-level optimizations including Telescoping languages [37, 12] and Broadway [31]. Such language processors are able to treat a library like MPI as a domain specific language, in which the MPI entries may be optimized as language intrinsics, embedded within an ordinary language like C, C++, or Fortran. Embedded domain specific languages are expected to play an important role in Exascale computing.

7. Conclusions

This paper presented a novel interpretation of Message Passing Interface to execute MPI applications under a data-driven model that can overlap communication with computation automatically. This interpretation factors scheduling issues and communication decisions out of program execution. Specifically, by reformulating MPI source into the form of a task dependency graph, which

maintains the data dependency among tasks of the graph, we can rely on a runtime system to schedule tasks based on the availability of data and computing resources.

To implement our approach we developed Bamboo, a custom source-to-source translator that transforms MPI code into the task dependency graph representation. Bamboo treats the MPI API as an embedded domain specific language, and it requires only a modest amount of programmer annotation. The implementation of Bamboo comprises 2 software layers: *core message passing* and *utility* layers. The *core message passing* layer transforms a minimal subset of MPI point-to-point primitives, whereas the *utility* layer implements high-level routines by breaking them into their point-to-point components, which will be then translated by the *core message passing* layer. Such a multi-layer design allows one to customize the implementation of MPI high-level routines such as collectives, which may take advantage of special purpose hardware provided on some platforms. In addition, this design can reduce the amount of programming effort needed to port the *core message passing* layer to a different runtime system.

We demonstrated that Bamboo improved performance by hiding communication. We showed that by using Bamboo, we can avoid the complications of the lookahead algorithm implemented in the High Performance Linpack (HPL) benchmark, while realizing the benefits. For structured grid, we translated an iterative solver for Poisson's equation and a geometric multigrid solver for Helmholtz's equation. For all applications, we have validated our claim that, by interpreting an MPI program in terms of data flow execution, we can overlap communication with computation and thereby improving the performance significantly. Moreover, Bamboo performance meets or exceeds that of labor-intensive hand coding, at scale. Bamboo also improves performance of *communication avoiding* matrix multiplication (2.5D Cannon's algorithm). The result on this application demonstrates that the translated code not only avoids communication, but tolerates what it cannot avoid. We believe that this dual strategy will become more widespread as data motion costs continue to grow. We also validated Bamboo on advanced node architectures, which accelerate node performance by offloading compute-intensive kernels to devices such as GPUs. Bamboo not only improves performance of a program written under the MPI+CUDA programming model, but also offers a simpler interface that allows communication between GPUs to be transparent to the programmer.

Lastly, Bamboo enables the programmer to specify scheduling hints as task priorities in order to optimize the scheduler. A task with higher priority will have a higher chance to be scheduled quickly. Such task prioritization support is important in applications that consist of irregular workloads. While Bamboo's scheduler employs a non-preemptive task scheduling [20, 21, 22], it allows tasks to voluntarily yield the processor, enabling tasks of the graph to work in a more cooperative manner. This dual scheduling scheme allows hardware resources to be efficiently shared among tasks. We evaluated the task prioritization support using the High Performance Linpack benchmark. Experimental results demonstrated that we gained significant performance benefits by employing simple

prioritization schemes.

In the future we can extend Bamboo to support complicated, heterogeneous computing. A compute node may contain multiple types of multicore and many-core processors. Thus, processor cores may run at different speeds with the result that data partitioning and mapping are non-trivial programming tasks. Bamboo alleviates these challenges by supporting process virtualization. However, in the future Bamboo needs to provide an analytical model and/or auto-tuning support for finding optimal or near-optimal virtualization factors and task mapping schemes. For irregular applications, hints from the programmer may be useful to effective task migration.

Acknowledgments

This research was supported by the Advanced Scientific Computing Research (ASCR), the U.S. Department of Energy, Office of Science, contracts No. DE-ER08-191010356-46564-95715, DE-FC02-12ER26118, and DE-AC05-76RL01830. A portion of the research was supported by EMSL (Environmental Molecular Sciences Laboratory) at Pacific Northwest National Laboratory. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work also used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575. We would like to thank Samuel Williams for providing us with the MPI source code of the multigrid application. Tan Nguyen was a fellow of the Vietnam Education Foundation (VEF) while conducting this research, and was supported in part by the VEF. Scott Baden dedicates his portion of this work to Hans Petter Langtangen (1962-2016).

[1] *Stampede user guide.* http://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide.

[2] A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-c. Feng, K. R. Bisset, and R. Thakur, *Mpi-acc: An integrated and extensible approach to data movement in accelerator-based systems*, in Proc. 2012 IEEE 14th Int'l Conf. High Perf. Computing and Communication & 2012 IEEE 9th Intl Conf. Embedded Software and Sys., HPCC '12, Washington, DC, USA, 2012, IEEE Computer Society, pp. 647–654.

[3] A. M. Aji, L. S. Panwar, F. Ji, M. Chabbi, K. Murthy, P. Balaji, K. R. Bisset, J. Dinan, W.-c. Feng, J. Mellor-Crummey, X. Ma, and R. Thakur, *On the efficacy of gpu-integrated mpi for scientific applications*, in Proceedings of the 22nd Int'l Symp on High-performance Parallel and Distributed Computing, HPDC '13, NY, NY, 2013, ACM, pp. 191–202.

[4] Arvind, *Executing a program on the mit tagged-token dataflow architecture*, IEEE Transactions on Computers, 39 (1990), pp. 300–318.

[5] S. B. Baden and S. J. Fink, *Communication overlap in multi-tier parallel algorithms*, in Proc. of SC '98, Orlando, Florida, November 1998.

[6] R. E. Bank and M. Holst, *A new paradigm for parallel adaptive meshing algorithms*, SIAM Review, 45 (2003), pp. 292–323.

[7] M. Beynon, T. M. Kurc, U. V. Catalyurek, C. Chang, A. Sussman, and J. H. Saltz, *Distributed processing of very large datasets with datacutter*, Par. Comput., (2001), pp. 1457–1478.

[8] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, 1997.

[9] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, *Dague: A generic distributed dag engine for high performance computing*, Parallel Comput., 38 (2012), pp. 37–51.

[10] K. D. Bosschere, E. H. D'Hollander, G. R. Joubert, D. Padua, and F. Peters, *Applications, Tools and Techniques on the Road to Exascale Computing*, IOS Press, Amsterdam, The Netherlands, The Netherlands, 2012.

[11] W. L. Briggs, *A Multigrid Tutorial*, SIAM, 1987.

[12] B. Broom, R. Fowler, and K. Kennedy, *Kelpio: A telescope-ready domain-specific i/o library for irregular block-structured applications*, in Proc. First IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE, 2001, pp. 148–155.

[13] A. M. Bruaset and A. Tveito, *Numerical solution of partial differential equations on parallel computers*, vol. 51, Springer, 2006.

[14] E. Bylaska, K. Tsemekhman, N. Govind, and M. Valiev, *Large-scale plane-wave-based density functional theory: Formalism, parallelization, and applications*, in Computational Methods for Large Systems: Electronic Structure Approaches for Biotechnology and Nanotechnology, J. R. Reimers, ed., John Wiley and Sons, Inc., 2011.

[15] E. Chan, R. van de Geijn, and A. Chapman, *Managing the complexity of lookahead for lu factorization with pivoting*, in Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '10, NY, NY, 2010, ACM, pp. 200–208.

[16] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, *Integrating asynchronous task parallelism with mpi*, in Proc. of the 2013 IEEE 27th Int'l Symp. Parallel Distrib. Processing, IPDPS '13, 2013, pp. 712–725.

[17] W.-Y. Chen, C. Iancu, and K. Yelick, *Communication optimizations for fine-grained upc applications*, in Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, PACT '05, Washington, DC, USA, 2005, IEEE Computer Society, pp. 267–278.

[18] Y. Chen, X. Cui, and H. Mei, *Large-scale fft on gpu clusters*, in Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10, NY, NY, 2010, ACM, pp. 315–324.

[19] N. Chrisochoides, K. Barker, J. Dobbelaere, D. Nave, and K. Pingali, *Data movement and control substrate for parallel adaptive applications*, Concurrency Practice and Experience, (2002), pp. 77–101.

[20] P. Cicotti, *Tarragon: a Programming Model for Latency-Hiding Scientific Computations*, PhD thesis, Department of Computer Science and Engineering, University of California, San Diego, 2011.

[21] P. Cicotti and S. B. Baden, *Asynchronous programming with tarragon*, in Proc. 15th IEEE International Symposium on High Performance Distributed Computing, Paris, France, Jun. 2006, pp. 375–376.

[22] ———, *Latency hiding and performance tuning with graph-based execution*, in The Seventh IEEE eScience Conference, Data-Flow Execution Models for Extreme Scale Computing (DFM 2011), Galveston Island, Texas, 2011.

[23] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany, *Transformations to parallel codes for communication-computation overlap*, in Proceedings of the ACM/IEEE SC 2005 Conference, November 2005, pp. 58–68.

[24] J. Dennis, *Data flow supercomputers*, IEEE Computer, 13 (1980), pp. 48–56.

[25] J. J. Dongarra, *The linpack benchmark: An explanation*, in Supercomputing, Springer, 1988, pp. 456–474.

[26] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK users' guide*, vol. 8, Siam, 1979.

[27] J. J. Dongarra, P. Luszczek, and A. Petitet, *The linpack benchmark: Past, present, and future. concurrency and computation: Practice and experience*, Concurrency and Computation: Practice and Experience, 15 (2003), p. 2003.

[28] S. Feigh, M. Clemens, and T. Weiland, *Geometric multigrid method for electro-and magnetostatic field simulations using the conformal finite integration technique*, in 2003 Copper Mountain Conference on Multigrid Methods, Copper Mountain, Colorado, Citeseer, 2003.

[29] S. J. Fink, *Hierarchical Programming for Block–Structured Scientific Calculations*, PhD thesis, Dept. of Comput. Sci. and Eng., Univ. of Calif., San Diego, 1998.

[30] A. Gupta, G. Karypis, and V. Kumar, *Highly scalable parallel algorithms for sparse matrix factorization*, IEEE Trans. Parallel Distrib. Syst., 8 (1997), pp. 502–520.

[31] S. Z. Guyer and C. Lin, *An annotation language for optimizing software libraries*, ACM SIGPLAN Notices, 35 (2000), pp. 39–52.

[32] M. M. S. B. K. P. D. Hovland, *Data-flow analysis for mpi programs*, in Intl. Conf. on Parallel Processing, ICPP 2006, Aug. 2006, pp. 175–184.

[33] C. Huang, O. Lawlor, and L. Kalé, *Adaptive mpi*, in Proc. 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03), 2003.

[34] P. Husbands and K. Yelick, *Multithreading and one-sided communication in parallel lu factorization*, in Proc 2007 ACM/IEEE Conf. on Supercomputing (SC '07), Reno, NV, Nov. 2007, ACM.

[35] L. V. Kale and S. Krishnan, *Charm++: a portable concurrent object oriented system based on c++*, in Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, OOPSLA '93, NY, NY, 1993, ACM, pp. 91–108.

[36] ———, *Charm++: a portable concurrent object oriented system based on c++*, in Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, OOPSLA '93, NY, NY, 1993, ACM, pp. 91–108.

[37] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey, *Telescoping languages: A system for automatic generation of domain languages*, Proc. IEEE, 93 (2005), pp. 387–408.

[38] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick, *Parallel programming in split-c*, in Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, Supercomputing '93, NY, NY, 1993, ACM, pp. 262–273.

[39] J. Kurzak and J. Dongarra, *Fully dynamic scheduler for numerical computing on multicore processors, lapack working note 220*.

[40] J. Lifflander, P. Miller, R. Venkataraman, A. Arya, L. Kale, and T. Jones, *Mapping dense lu factorization on multicore supercomputer nodes*, in Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, May 2012, pp. 596–606.

[41] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, *Overlapping communication and computation by using a hybrid mpi/smpss approach*, in Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10, 2010, pp. 5–16.

[42] D. Marx and J. Hutter, *Ab-initio molecular dynamics: Theory and implementation*, in Modern Methods and Algorithms of Quantum Chemistry, J. Grotendorst, ed., NIC, Forschungszentrum Jlich, i ed., 2000, ch. 13, pp. 301–449. Publicly available at the URL: http://www2.fz-juelich.de/nic-series/Volume3/marx.pdf.

[43] P. McCorquodale, P. Colella, G. T. Balls, and S. B. Baden, *Local corrections algorithm for solving poisson's equation in three dimensions*, Comm. App. Math. and Comp. Sci., 2 (2007), pp. 57–81.

[44] Message-Passing Interface Standard, *MPI: A message-passing interface standard*, University of Tennessee, Knoxville, TN, June 1995.

[45] T. Nguyen and S. B. Baden, *Preliminary scaling results on multiple hybrid nodes of knights corner and sandy bridge processors*, in Accepted to Third International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, 2013.

[46] T. Nguyen and S. B. Baden, *Lu factorization: Towards hiding communication overheads with a lookahead-free algorithm*, in 2015 IEEE International Conference on Cluster Computing, Sept 2015, pp. 394–397.

[47] T. Nguyen, P. Cicotti, E. Bylaska, D. Quinlan, and S. B. Baden, *Bamboo: translating mpi applications to a latency-tolerant, data-driven form*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, 2012, pp. 39:1–39:11.

[48] R. Preissl, M. Schulz, D. Kranzlmuller, B. de Supinski, and D. Quinlan, *Using mpi communication patterns to guide source code transformations*, in Computational Science ICCS 2008, vol. 5103 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008, pp. 253–260.

[49] D. Quinlan, D. Miller, B. Philip, and M. Schordan, *Treating a user-defined parallel library as a domain-specific language*, in Proceedings of the 16th international Parallel and Distributed Processing Symposium, IPDPS 2002, Los Alamitos, CA, USA, April 2002, IEEE.

[50] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill Education Group, 2003.

[51] P. Raghavan, K. Teranishi, and E. Ng, *A latency tolerant hybrid sparse solver using incomplete cholesky factorization*, Numer. Linear Algebra Appl., 10 (2003), pp. 541–560.

[52] V. Sarkar, W. Harrod, and A. E. Snavely, *Software challenges in extreme scale systems*, in Journal of Physics: Conference Series, vol. 180, IOP Publishing, 2009, p. 012045.

[53] J. Shalf, S. Dosanjh, and J. Morrison, *Exascale computing technology challenges*, in Proceedings of the 9th international conference on High performance computing for computational science, VECPAR'10, Berlin, Heidelberg, 2011, Springer-Verlag, pp. 1–25.

[54] D. R. Shires, L. L. Pollock, and S. Sprenkle, *Program flow graph construction for static analysis of mpi programs*, in PDPTA, 1999, pp. 1847–1853.

[55] R. Silva, G. Pezzi, N. Maillard, and T. Diverio, *Automatic data-flow graph generation of mpi programs*, Computer Architecture and High Performance Computing, 2005. SBAC-PAD 2005. 17th International Symposium on, (24-27 Oct. 2005), pp. 93–100.

[56] A. Sohn and R. Biswas, *Communication studies of DMP and SMP machines*, Tech. Rep. NAS-97-004, NAS, 1997.

[57] E. Solomonik and J. Demmel, *Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms*, Tech. Rep. UCB/EECS-2011-72, EECS Department, University of California, Berkeley, Jun 2011.

[58] A. K. Somani and A. M. Sansano, *Minimizing overhead in parallel algorithms through overlapping communication/computation*, Tech. Rep. 97-8, ICASE, February 1997.

[59] J. Sorensen and S. B. Baden, *Hiding communication latency with non-spmd, graph-based execution*, in Proc. 9th Intl Conf. Computational Sci. (ICCS '09), Berlin, Heidelberg, 2009, Springer-Verlag, pp. 155–164.

[60] K. Teranishi, P. Raghavan, and E. Ng, *A new data-mapping scheme for latency-tolerant distributed sparse triangular solution*, in Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Los Alamitos, CA, USA, 2002, IEEE Computer Society Press, pp. 1–11.

[61] J. D. Teresco, M. W. Beall, J. E. Flaherty, and M. S. Shephard, *A hierarchical partition model for adaptive finite element computation*, Comput. Methods. Appl. Mech. Engrg., 184 (2000), pp. 269–285.

[62] R. Thakur and R. Rabenseifner, *Optimization of collective communication operations in mpich*, International J. of High Performance Computing Applications, 19 (2005), pp. 49–66.

[63] V. Volkov and J. W. Demmel, *Benchmarking gpus to tune dense linear algebra*, in Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08, Piscataway, NJ, USA, 2008, IEEE Press, pp. 31:1–31:11.

[64] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, *Mvapich2-gpu: Optimized gpu to gpu communication for infiniband clusters*, Comput. Sci., 26 (2011), pp. 257–266.

[65] C. Wen and K. Yelick, *Portable runtime support for asynchronous simulation*, in in International Conference on Parallel Processing, 1995.

[66] P. Wesseling and C. Oosterlee, *Geometric multigrid with applications to computational fluid dynamics*, Journal of Computational and Applied Mathematics, 128 (2001), pp. 311 – 334. Numerical Analysis 2000. Vol. VII: Partial Differential Equations.

[67] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker, *Optimization of geometric multigrid for emerging multi- and manycore processors*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, 2012, pp. 96:1–96:11.

Nguyen received his Ph.D. degree in Computer Science from University of California, San Diego in 2014. Nguyen's research interests include programming abstractions, communication hiding, and compiler-based performance modeling and optimization. Nguyen is currently a postdoctoral researcher at Lawrence Berkeley National Laboratory.

Dr. Cicotti is a senior computational scientist at SDSC. His research deals with aspects of emerging technology and novel system architecture. His work includes the development of runtime systems to hide communication, improve locality, and increase energy efficiency. He developed software tools for capturing and analyzing data movement, and is currently investigating the use of this information for managing data in NUMA, heterogeneous, and non-volatile memory hierarchies. Current optimization

work also includes IO and hierarchical storage systems. Finally, he collaborates on scientific data analysis projects utilizing map-reduce and emerging programming models.

Dr. Bylaska has conducted extensive research in geochemistry and heavy element chemistry for the last 15 years and is an expert in developing first principles simulations to accurately model the thermodynamics, kinetics and dynamics of molecular and condensed phase systems. He also has formal training in electronic structure methods and detailed knowledge of models and model development. He is also primary developer of the first principles plane-wave module in NWChem software. NWChem is an award-winning computational chemistry package for parallel computers developed at EMSL. Recently, he has also developed EMSL Arrows, which combines molecular modeling, SQL and NOSQL databases, email, and social networks to make molecular and materials modeling accessible to all scientists and engineers.

Dan Quinlan is the leader of the **ROSE** project in the Center for Advanced Scientific Computing. His research is in numerous areas that intersect computer science and numerical analysis. Research interests include object-oriented numerical frameworks, parallel adaptive mesh refinement, parallel multigrid algorithms, semantics-based source code transformations, C++ compiler tools/infrastructure/design, cache-based optimizations, parallel array classes, parallel data distribution mechanisms, and parallel load balancing algorithms. Dr. Quinlan earned his Ph.D. in Computational Mathematics from the University of Colorado.

Scott B. Baden is Professor in the Computer Science and Engineering at the University of California, San Diego and is currently on leave at Lawrence Berkeley National Laboratory, where he leads the Computer Languages and Systems Software Group. His research focuses on domain-specific translation, language and run time support for low cost communication, adaptive and irregular applications. Dr. Baden has a PhD in computer science from the University of California, Berkeley. He is a senior member of IEEE, a member of SIAM, and a senior fellow at the San Diego Supercomputer Center.

- We present *Bamboo*, a source-to-source translator that transforms MPI source code into a task graph formulation that executes in a data-driven fashion.
- Bamboo supports both point-to-point and collective communication.
- Bamboo supports GPUs, hiding communication among GPUs and between GPUs and their hosts.
- We present a thorough evaluation using applications with elaborate program structures and communication patterns.