Bamboo - Translating MPI applications to a latency-tolerant, data-driven form

Tan Nguyen[1], Pietro Cicotti[1], Eric Bylaska[2], Dan Quinlan[3] and Scott B. Baden[1]
[1]Department of Computer Science and Engineering
University of California, San Diego, La Jolla, CA 92093 USA
[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
Email: nnguyent@ucsd.edu, pcicotti@sdsc.edu,
Eric.Bylaska@pnnl.gov, quinlan1@llnl.gov, baden@ucsd.edu

***Abstract*— We present Bamboo, a custom source-to-source translator that transforms MPI C source into a data-driven form that automatically overlaps communication with available computation. Running on up to 98304 processors of NERSC's *Hopper* system, we observe that Bamboo's overlap capability speeds up MPI implementations of a 3D Jacobi iterative solver and Cannon's matrix multiplication. Bamboo's generated code meets or exceeds the performance of hand optimized MPI, which includes split-phase coding, the method classically employed to hide communication. We achieved our results with only modest amounts of programmer annotation and no intrusive reprogramming of the original application source.**

I. Introduction

Applications running on exascale computers will invest heavily in optimizations that reduce data motion costs, including techniques to overlap communication with computation. Since present day compiler technology cannot perform the required optimizations, the task of masking communication delays entails significant, intrusive performance programming, challenging even the expert programmer. Moreover, with no assurance that current software techniques will continue to be effective in the face of rapid technological evolution, performance robustness will be troublesome.

In order to achieve expected performance levels at the Exascale, existing code bases, rooted in Bulk Synchronous Parallelism (BSP) [34], and authored with the Message Passing Interface (MPI) [26], would require extensive modification.

In this paper, we present *Bamboo,* a source-to-source translator that transforms an MPI program into a semantically equivalent task precedence graph formulation, which automatically masks communication with available computation. Bamboo re-engineers MPI code to run as a data-driven program (like coarse-grain dataflow [2]) running under the control of runtime services which schedule tasks according to the flow of data and the availability of processing resources. The run time services rely on virtualization [20] to pipeline communication and computation.

We validated Bamboo against two important application motifs [12]: Jacobi's method for solving Poisson's equation in 3 dimensions (structured grid) and Cannon's Matrix Multiplication Algorithm (dense linear algebra). We ran on up to 98304 processor cores of NERSC's Hopper system and demonstrated that Bamboo-generated code improved performance by masking communication delays. Performance was competitive with that of carefully optimized MPI source that was manually restructured with classic split-phase coding. Indeed, Bamboo avoids the need for classic split-phase code that complicates communication tolerant applications. Whereas classic split-phase code embeds the communication tolerance strategy into the application, Bamboo factors the communication overlap strategy out of the user's code, improving code maintainability and performance robustness.

Bamboo is a custom source-to-source translator that recognizes MPI calls, in effect treating the MPI entries as primitive language objects. We implemented the translator with the ROSE compiler framework [29]. Though Bamboo relies on programmer annotations to guide the transformation process, our experience is that these annotations are intuitive and modest in number.

We target MPI because it is the dominant means of building parallel scalable applications. However, our approach is not limited to MPI and also applies to other data motion libraries such as GasNet [5]. More generally, our technique for treating MPI illustrates an approach for delivering an embedded domain specific language, in which the methods of a library API in effect become primitive language constructs.

The remainder of this paper is as follows. Sec. II introduces the Bamboo programming model. Sec. III describes the data-driven execution of the Bamboo's generated code, including the underlying runtime support. Sec. IV discusses the implementation of Bamboo. Sec. V presents performance results. Sec. VI describes related work. Finally, Sec. VII concludes the paper and discusses future work.

II. Bamboo

Scalable applications are generally written under the SPMD (Same Program Multiple Data) model, and implemented with MPI [26]. MPI enables the application programmer to cater optimizations that benefit performance to application heuristics, in particular, involving data motion and locality. Such

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

domain specific knowledge is difficult to capture via general-purpose language constructs and associated compilation strategies that are unaware of application and library semantics. This observation motivates the design of Bamboo, which is a custom translator tailored to the MPI interface.

Bamboo treats the API's members as primitives in an embedded domain specific language, reformulating the program to mask communication delays while maintaining the heuristic knowledge encoded via MPI. Bamboo uses knowledge about the MPI API to interpret an MPI program as an encoding of data and control dependencies among a partial ordering of tasks comprising the application. Bamboo extracts data and control dependencies from the pattern of MPI call sites and, with the help of runtime support, constructs a task precedence graph corresponding to a partial ordering of tasks.

Tasks run under the control of the Tarragon runtime library [10], [9], [11], which executes a task graph using data-flow like semantics [1], [15]. The dataflow model is appealing because it automatically masks data motion without programmer intervention [4], [10], [19], [32], [9], [11].

Since static analysis is not sufficient to infer matching sends and receives in a running program [28], Bamboo requires some programmer annotation of the original MPI program. In our opinion, these annotations impose a modest burden on the programmer, compared to the traditional approach of restructuring an application to hide communication via split-phase coding.

A. A first Bamboo program

To illuminate our discussions about translation under Bamboo, we will use a simple motivating example: an iterative finite difference method that uses Jacobi's method to solve Laplace's equation in two dimensions. The left side of Fig. 1, shows how the numerical kernel updates each point of a 2D mesh with the average of four nearest neighbors along the Manhattan directions. It uses an "old" and a "new" mesh, swapping the meshes after each iteration.

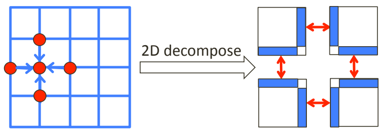


Fig. 1: Left: the 2D Jacobi stencil; right: 2D partitioning, showing data dependencies involving ghost cells.

The MPI implementation partitions the 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 we store copies of off-processor boundary values in "ghost cells" (right side of Fig. 1), which we refresh after each mesh sweep. This is shown in Fig. 2, an MPI program for the Jacobi iteration, which includes Bamboo annotations. Since a conventional compiler will ignore the annotations, the code in Fig. 2 is also a legal MPI program. We next describe Bamboo's underlying programming model and its annotations.

```
 1 MPI_Init(&argc , &argv);
 2 MPI_Comm_rank(&my_rank , MPI_COMM_WORLD);
 3 MPI_Comm_size(&numprocs , MPI_COMM_WORLD);
 4 Compute processID of left / right /up/down
 5 Allocate U, V, SendGhostcells , RecvGhostcells
 6 #pragma bamboo olap
 7   for (it=0; it<num_iterations; it++){
 8    #pragma bamboo send{
 9        pack boundary values to message Buffer
10        MPI_Isend(SendGhostcells) to left/right/up/down
11    }
12    #pragma bamboo receive{
13        MPI_Recv(RecvGhostcells) from left/right/up/down
14        unpack incoming data to ghost cells
15    }
16    MPI_Waitall();
17    for(j=1; j < N/numprocs_Y -2; j++)
18      for(i=1; i < N/numprocs_X -2; i++)
19         V(j,i)= c*(U(j,i+1)+U(j,i-1)+U(j+1,i)+U(j-1,i))
20    swap(U, V);
21  }
22 free U, V, SendGhostcells , RecvGhostcells
23 MPI_Finalize()
```

Olap-region

Send Blk

Recv Blk

Compute Blk

Fig. 2: Annotated MPI program for 2DJacobi. Some code has been omitted for the purposes of clarity. To save space, we employ non-standard C syntax for send and receive regions: an opening curly brace appears on the same line as the corresponding pragma.

B. The Bamboo Programming Model

Starting from an MPI program, Bamboo generates an equivalent, meaning-preserving task precedence graph, which we will refer to as a *task graph* from now on. This program runs under the control of runtime services, which process the graph according to dataflow semantics. In order to correctly interpret MPI program execution in terms of a task graph, Bamboo requires some additional knowledge about the input source, that comes in the form of programmer annotations.

A Bamboo program is a legal MPI program, augmented with one or more *olap-regions*. An *olap-region* is a section of code containing communication to be overlapped with computation, both located within the same *olap-region*. Lines 6-21 in Fig. 2 show an example of an *olap-region*. Bamboo can recognize opportunities to overlap communication with computation within *olap-regions* only. It preserves the execution order of *olap-regions*, which run sequentially, one after the other.

Each olap-region contains at least 2 *communication blocks*, plus a single computational block that depends upon the completion of communication. The computational block is optional and may contain other *olap-regions*. The common case is for computational blocks to be present, as in the applications presented in this paper. There are two kinds of communication blocks: *send* and *receive*. 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, pack and unpack message buffers. 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. However, Bamboo will not reorder blocks of the same type.

Bamboo currently handles 6 fundamental message passing primitives inside communication regions: blocking send and receive (*Send* and *Recv*), *asynchronous* variants (*iSend* and *iRecv*) and synchronization (*Wait* and *Waitall*). A *send* block may contain *Sends* only. In most cases, a *receive* block will contain *Recvs* only, except for the following situation. If a *Send* has a read after write dependence on a prior *Recv*, then it must reside within an appropriate *receive* block, either the same block as the Recv, or a later one. This ensures that the two calls will not be incorrectly reordered by Bamboo. Conversely, if a *Recv* has an anti-dependence on a prior *Send*, it appears that Bamboo's reordering could give rise to incorrect code. However, because Bamboo treats each *Send* with a temporary buffer, it effectively renames the Send's buffer, and we can safely place the *Recv* and *Send* within respective receive and send blocks. *Wait* and *Waitall* specify synchronization points; their semantics are preserved by Bamboo. We note that the above restrictions do not rule out the expression of certain communication patterns. Rather, they provide a methodology, that is, guidelines for inserting Bamboo annotations that ensures code correctness.

Revisiting the code in Fig. 2, a single *send* pragma at line (8) groups four *MPI_Isend* invocations together. The *iSend()* at (10) consumes the data produced by code at (9) that linearizes data into a buffer. Similarly, the *receive* pragma (12) groups four *MPI_Recv* calls. The unpacking code inside the receive block at (14) consumes data delivered by the *Recv()* at (13). By grouping the four *iSends* and four *Recvs* into a *send* and a *receive* block, respectively, the user informs Bamboo that the sending and the receiving of ghost cells are two independent activities. Once the two communication blocks complete, the computations to update the local mesh at lines (17)-(19) may execute.

C. Basic directives

Programmer annotations are a powerful means of guiding translation and effect a tradeoff between translator complexity and the application programmer's ease-of-use. Bamboo provides two kinds of directives. The first kind is required in all applications, and treats overlap regions and communication blocks as described previously. We call these "basic directives." All Bamboo directives begin with *#pragma bamboo*, followed by a directive name and any clauses associated with the directive. The syntax for the *olap-region* and the *communication block* directives are as follows, where we use regular expression syntax to specify alternatives.

```
#pragma bamboo olap
#pragma bamboo [send|receive]
```

Directives of the second kind, called "optimizations," specify optimizations that may or may not be needed in all programs, though highly scalable programs will generally require them. Since the optimization directives require some knowledge of Bamboo's underlying data-driven execution model currently implemented by Tarragon, we will defer their discussion to §III-B, after we have discussed that model.

III. Data-driven execution

A. Execution model

Bamboo generates source code to produce a new program represented as a task graph that runs under a data-flow like execution model. This program relies on a task graph library, *Tarragon,* to define, manage, and execute task precedence graphs, which are objects of type *TaskGraph*. Bamboo expects the task graph library to export this abstract base class, and generates concrete instances as needed. The nodes of a TaskGraph correspond to tasks to be executed, which are objects of type *Task*. The edges correspond to data dependencies among tasks. The taskgraph library's runtime system interprets these dependencies as communication channels and tasks communicate via active messages.

Each MPI process will be effectively transformed into a set of tasks that are executed by a group of *worker threads*. The library supports task execution via one or more *service threads*, including scheduling. In order to improve the success of hiding latency, we create more tasks than processing cores; as noted by others, virtualization is important in hiding latency [21], [32], [33]. Worker threads run to completion but the tasks do not. The combination of task virtualization and run to completion behavior requires that we take measures to avoid deadlock. To this end, a task does not explicitly wait on communication. The effect is to control when data becomes visible to the task.

The underlying execution model of Bamboo is like dataflow, but with the provision for task *state.* Tasks will generally alternate between the running and suspended states, for example, in an iterative method. When a task finishes computing, it moves data along output edges and then it suspends, at which point it is waiting on arriving data.[1] The runtime services recognize task state and will swap in a runnable task, which has met the conditions of its *firing rule.* A task's firing rule is simple: the task becomes runnable once it receives all required messages.

When a task runs-or becomes runnable-all its input data are guaranteed to be available. Any newly-arriving data will not be visible to the task until after it has suspended; the task has received all the data it needs to run, and any newly arriving data will be visible the next time the task runs. The resultant delays in data visibility are fundamentally different in a task graph program and an MPI program. To handle these semantic differences Bamboo employs code motion transformations that will be described in §IV-B.

B. Communication Layout Optimization

Although the directives discussed in §II-C are sufficient to begin using Bamboo, one more is needed to conserve storage consumed by data structures that are internal to the task graph's runtime system.

In order to transmit messages between tasks, Bamboo's runtime system relies on the *task mapping table*, a table of descriptors describing the edges that connect the tasks in a

[1] At some point the task completes. When all tasks in a task graph complete, the task graph invocation completes and a new invocation may begin.

taskgraph. This information is determined from the communication structure encoded in the pattern of message passing calls appearing in the application. However, static analysis is insufficient to solve the general case [16]. Bronevetsky reported a solution for certain kinds of geometries and when there is only blocking communication [6].

By default, Bamboo will conservatively generate an all-to-all task mapping table, such that each task has a channel to every other task. This expensive solution will not scale, since the aggregate size of the mapping table grows quadratically in the number of tasks. However, in practice, communication structures are sparse. For example, in collectives that employ spanning tree like algorithms, the number of mapping table entries per task grows as the log of the number of tasks. To this end, Bamboo provides a pragma to specify a task graph's communication structure, which conserves space consumed by the mapping table. Like Bronevetsky's solution, Bamboo frames the kinds of geometries it will accept, but it accepts non-blocking communication.

To enable Bamboo to generate a sparse task mapping table, we include support for *communication layouts*. These layouts avoid unacceptable growth in the mapping table, which is filled with only those entries required to carry out a specified communication pattern. Currently, Bamboo defines a small set of pre-defined layouts including various nearest neighbor layouts. It also provides base classes for users to quickly generate their own layouts.

In order to process the layout information, Bamboo needs to know the number of spatial dimensions in which to embed the layout, the rank of the underlying virtualized task geometry. We refer to this geometry as the *embedding geometry.* It is not necessary for Bamboo to know the bounds of the embedding geometry, as this information is specified when we run the generated program, and hence determined at runtime.

Since Bamboo cannot currently infer the the number of dimensions in the embedding geometry we provide the *dimension* directive to specify the value. The directive must appear exactly once in the program, prior to the first `olap` region pragma. The syntax for specifying the number of dimensions of the embedding geometry and a communication layout are as follows, where we use regular expression syntax to specify repeated or optional parts.

```
#pragma bamboo dimension NumProcGeomDims
#pragma bamboo olap (layout name (args)* )*
```

For example, in the 3D Jacobi method, the annotations for a 2D data decomposition in the Y and Z dimensions are as follows, where NN is a nearest neighbor layout, pre-defined by Bamboo.

```
#pragma bamboo dimension 2
#pragma bamboo olap layout NN Y layout NN Z
```

IV. Implementation

A. The Bamboo translation framework

Fig. 3 shows the block diagram of Bamboo, which is divided into a front-end (parser), a middle-end and a back-end. We used the ROSE compiler infrastructure to implement Bamboo [29], which includes the EDG front-end used to parse standard C source. This front-end generates an Intermediate Representation (IR), which is an in-memory Abstract Syntax Tree (AST). Since EDG considers the MPI calls as ordinary C function calls, we built a custom module sitting between the front and middle ends to extract information about the parameters passed to MPI functions. The four main modules of Bamboo built on top of ROSE's middle-end modify the IR to create a new form that conforms to the Tarragon API. The *annotation handler* extracts information from each Bamboo directive along with the corresponding location within the IR; the *analyzer* and *transformer* modify the IR to conform to the Tarragon model and the *optimizer* applies various transformations to improve the quality of the generated source code. Finally, the back-end completes the translation process by converting the IR back to source code.

Though the Bamboo translator encompasses all 3 sections, our effort was concentrated on the custom middle-end which is the focus of this section.

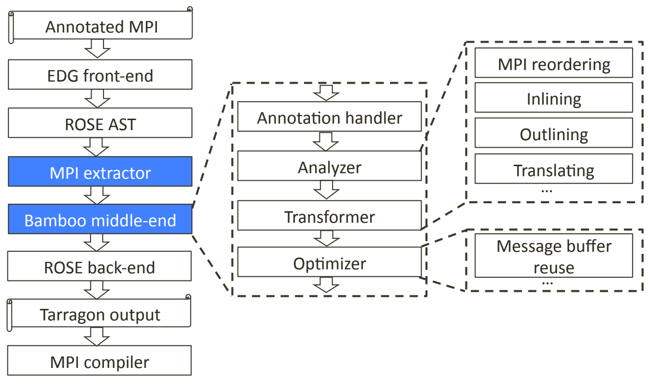


Fig. 3: The Bamboo translation framework.

B. MPI call reordering

To generate correct Tarragon code, Bamboo may need to reorder certain MPI function calls. This is a side effect of Tarragon semantics discussed previously. If a task is currently not suspended, it will be unable to receive messages via the message handler (*vinject()*) until the *next* time it is suspended. The left side of Tab. I shows a common pattern used in MPI applications that must be restructured by Bamboo. The *send* issued by a process matches up with the *receive* of the another process in the same iteration. However, 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*, advancing it in time so that the sending and receiving activities reside in different iterations. Bamboo will set up a pipeline, replicating the call to *send* to the front of, and outside, the iteration loop. It also migrates the existing call to the end

Before reordering	After reordering

`for (i=1;i<=nIters;i++){`
`  Receive`
`  Send`
`  Compute`
`}`
	`Send`
	`for (i=1;i<=nIters;){`
	`  Receive`
	`  Compute`
	`  i++`
	`  if(i<=niters)`
	`    Send`
	`}`

TABLE I: Left: a typical MPI program that requires code reordering. Right: the same code with send reordered.

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. I. The *send* and *receive* now reside in different iterations, preserving the meaning of the original code.

C. Code inlining and outlining

If a procedure other than main() 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 user and does not require any annotation. Though the ROSE infrastructure supports inter-procedural analysis, we prefer inlining, since it is more accurate. Inlining exposes the calling context to the procedure's body and the procedure's side effect on the caller. Moreover, since Bamboo inlines MPI-invoking procedures only, the amount of code requiring inlining is small. Our inlining strategy currently does not support recursive procedures.

The Tarragon API–which is the target of Bamboo–is a contract between the runtime system and the translator. Tarragon defines an abstract class called *Task.* Bamboo will derive concrete Task class(es) from this abstract class implementing (overriding) the three methods defined by Task: a firing rule, *vinject()*, the task's computations, *vexecute()* and the task's initialization, *vinit().* These methods are generated by Bamboo automatically, and are derived from the input source code. All are invoked via callbacks made by the task graph Runtime System (RTS) and are never invoked directly from application code. *Vinit()* generally executes once per task construction, but *vexecute()* may execute multiple times.

After inlining, Bamboo outlines code to the *vinit()*, *vexecute()* and *vinject()* methods, which are required to implement a concrete instance of *Task*. Bamboo splits the code into regions, with the analysis guided by Bamboo pragmas. Once it has outlined the code, Bamboo generates additional code to create and execute the TaskGraph.

D. MPI call translation

Bamboo translates MPI function calls to their Tarragon equivalents. The calls to *MPI_Init* and *MPI_Finalize* are handled trivially, and will not be discussed.

1) MPI_Comm_rank and MPI_Comm_size: These routines return the rank of a process and the total number of MPI processes. Since we virtualize the MPI processes into Tarragon tasks, Bamboo will take care of the mapping from MPI ranks to Tarragon task IDs by rewriting the calls *MPI_Comm_rank()* and *MPI_Comm_size()* to corresponding Tarragon calls that return the task ID and number of tasks, respectively.

2) MPI point-to-point communication primitives: Bamboo currently supports the following 6 point-to-point communication functions: *MPI_Send, MPI_Isend, MPI_Recv, MPI_Irecv, MPI_Wait*, and *MPI_Waitall.*

MPI_Send and MPI_Isend are translated to an invocation of the *put()* method, which is defined by *Task. Put()* is the sole data motion primitive defined by the task graph library and is like an active message.[2] Since *put()* is asynchronous, both forms of send differ only in terms of the time that the sending buffer can be safely overwritten. Bamboo copies the outgoing send buffer when issuing a *put()*, so both routines are effectively equivalent.

MPI_Recv and MPI_Irecv. When Bamboo encounters a blocking Recv, it incorporates the effect of that call into the task firing rule (defined in *vinject()*), together with any conditional statements connected with the receive, including induction variable tests of enclosing loop headers. However, a non-blocking, asynchronous *MPI_Irecv* does not require program control flow to block until either *MPI_Wait* or *MPI_Waitall* are invoked. Thus, Bamboo produces the firing rule when it encounters a corresponding *MPI_Wait* or *MPI_Waitall.*

MPI_Wait and MPI_Waitall. When Bamboo locates a non-blocking, asynchronous receive, i.e. *MPI_Irecv*, it will then look for a corresponding *MPI_Wait* or *MPI_Waitall*, and incorporate any conditional statement associated with the synchronization call when generating the Task firing rule. If *MPI_Wait* and *MPI_Waitall* correspond to an asynchronous *MPI_ISend*, no code needs to be generated; due to data buffering performed on the behalf of *put(),* there is no need to wait for a send to complete and we can safely remove any synchronization on that send.

V. Experimental evaluation

A. Applications and Testbed

We validated Bamboo against applications from two well known HPC computational motifs: a 3D Jacobi iterative solver (which we will refer to as 3D Jacobi) and dense matrix multiplication. For the latter, we implemented not only the classic Cannon algorithm [35], but also the communication avoiding variant, which targets small matrices [31].

Our computational testbed was *Hopper*, a Cray XE6 system located at the National Energy Research Scientific Computing Center (NERSC). Hopper's 153,216 cores are packaged as dual socket 12-core AMD MagnyCours 2.1GHz processors, which are further organized into two hex-core NUMA nodes. The 24-core Cray nodes are interconnected via Gemini interconnect (a 3D toroidal topology) and we used nodes with 32GB of memory. All source code was compiled using the *CC* wrapper, with the following optimization options: *-O3 -ffast-math.* This wrapper is a front end to MPI and we set it up to

[2]The *vinject()* method is the handler; a call to *put()* will trigger a callback to *vinject()* at the destination. The callback is initiated by the runtime system, which watches for incoming messages.

use the GNU compiler suite (GCC 4.6.1). High performance matrix multiply (*dgemm*) was supplied by ACML version 4.4.0.

B. 3D Jacobi iterative solver

1) Application and variant descriptions: 3D Jacobi solves Poisson's equation in three dimensions $\nabla^2 u = f$, subject to Dirichlet boundary conditions. The solver uses Jacobi's method with a 7-point central difference scheme that updates each point of the grid with the average of the six nearest neighbor values in the Manhattan directions. In order to make fair performance comparisons, we compared several variants of 3D Jacobi. All variants share the same numerical kernels and the variants using OpenMP all used the same OpenMP annotations to parallelize the loops. We blocked the 3D Jacobi kernel for cache along the Y and Z-axes, the non-contiguous axes that correspond to the two outer most loops of mesh sweep loop nest. We determined experimentally that a 4x8 block size was optimal.

The first variant, *MPI-basic*, is the simplest. It does not overlap communication with computation and is the starting point for the remaining variants.

The second variant, *MPI-olap*, has been manually restructured to employ split phase coding to overlap communication with computation. It employs a hierarchical data decomposition, subdividing the mesh assigned to each core into 8 equal parts using a 3D $2 \times 2 \times 2$ geometry. *MPI-olap* sets up a pipeline; within the outer iteration it sweeps one-half of the 8 sub-problems while communicating ghost cells for the others.

The third variant, *MPI+OMP*, employs a hybrid execution model running 1 MPI process per NUMA node, each unfolding a team of OpenMP threads to perform the mesh sweep over the work assigned to the 6 cores of the node. This hybrid variant uses just a fraction, 1/T, of the MPI processes used in the pure MPI variant, where T is the number of OpenMP threads per node. Under these conditions communication occurs at the NUMA node level: the single master thread exchanges ghost cells between nodes leaving all but one core idle during communication. NERSC recommends this variant over MPI-basic, as it saves memory.

The fourth variant, *MPI+OMP-olap*, combines the overlapping technique used in the second variant with the hybrid model used in the third. This variant takes advantage of both techniques, but hierarchical control flow reduces the effectiveness of overlap which occurs at the NUMA node level.

The fifth and the sixth variants, *Bamboo-basic and Bamboo+OMP,* were obtained by passing the *MPI-basic and MPI-OMP* variants, respectively, through the Bamboo translator. These codes run in data-driven fashion under the control of the task graph runtime system (Tarragon). The Bamboo annotations used in 3D Jacobi are somewhat different from the 2D Jacobi code previously shown in Fig. 2. We specify *dimension 3* for the embedding geometry instead of *dimension 2*. We also specify a communication layout: *layout NearestNeighbor* to match the 3D decomposition and the nearest neighbor task connection.

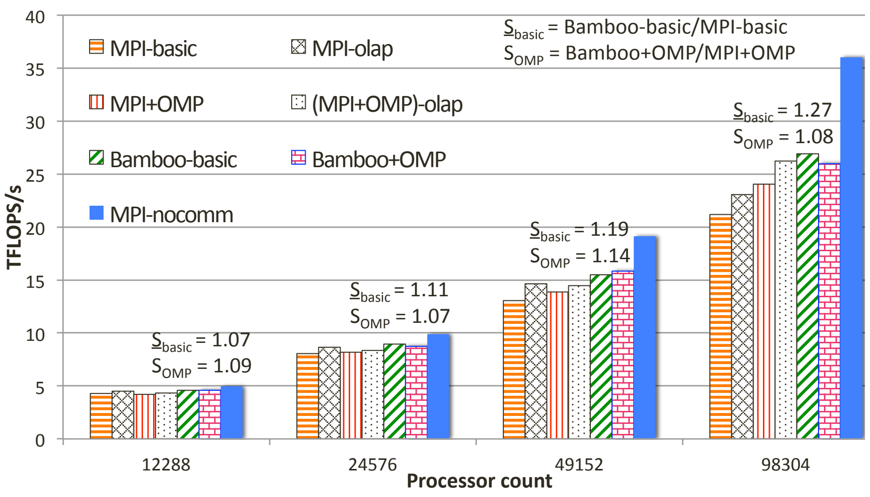
We also report performance for MPI-nocomm. This is not a true variant and is the result of turning off all message passing activity in MPI-basic. We use MPI-nocomm to establish an upper bound on performance, which we may or may not be able to realize in practice. Since 3D Jacobi is a memory bandwidth-bound application, the performance of MPI-nocomm is far below the peak performance of the hardware.

2) Aprun configuration: All jobs were launched using the *aprun* command. The pure MPI variants (MPI-basic and MPI-olap) ran with 1 process per core, while the others ran with 1 process per NUMA node, each spawning an identical number of OpenMP threads. Thus, the MPI variants were run with the following *aprun* command line arguments *-n P -N 24 -S 6,* where *P* is the total number of cores and we run with 24 MPI processes per Hopper node (*-N 24*) further organized into four groups of 6 processes per NUMA node (*-S 6*). The other variants ran with one MPI process per NUMA node using these *aprun* command line arguments: *-n p6 -N 4 -S 1*, where *p6 = P/6*. For the hybrid variants using OpenMP (MPI+OMP, MPI+OMP-olap and Bamboo+OMP), we specified *-d 6* to spawn 6 worker threads per NUMA node. For the Bamboo variants of the pure MPI codes, the translator manages thread spawning via Tarragon. It configured Tarragon to spawn 5 worker threads, each running on its own core, dedicating the remaining core to a service thread. Lastly, we specified the -ss option of *aprun*, which restricts each thread to use memory nearest to its NUMA node, improving performance.

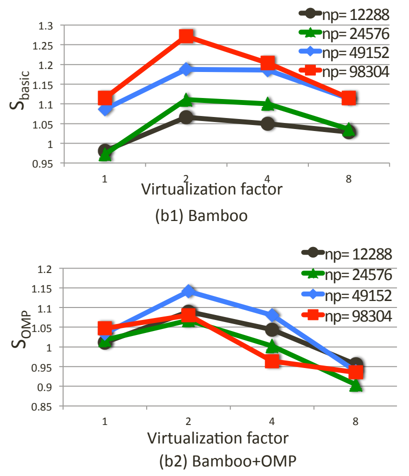
3) Results: We conducted a strong scaling study, maintaining a fixed problem size as we increase the number of processors. Strong scaling stresses communication overhead, though we still have sufficient computation to overlap with data motion (Unlike Cannon's algorithm, discussed next, where we were forced to employ weak scaling).

Fig. 4a compares the results with different variants of 3D Jacobi. Notably, *Bamboo* uniformly improves performance of both variants (*MPI-basic* and *MPI+OMP*) at all levels of parallelism. For example, on 96K (98034) cores, *Bamboo-basic* realizes a $\times 1.27$ speedup, hiding 52% of the communication delay in *MPI-basic.* More generally, the speedups ranged from 1.07 to 1.27. With strong scaling, communication overhead increases with the number of cores (from 13% to 41% over the range of 12K to 96K cores), and this explains why the performance increase delivered by Bamboo grows with the number of cores. Since the kernel is blocked for cache in all variants, we believe that most of the benefits come from latency hiding. To gain insight into the performance benefits of Bamboo, we next analyze the remaining two MPI variants.

The hybrid MPI+OMP variant demonstrates the benefits of multithreading which is also enjoyed by the Bamboo variants. Though this hybrid variant provides only a modest improvement over MPI-basic on smaller numbers of cores, it provides a large boost at 96K cores. We believe this is due to reduced communication delays achieved by hybrid MPI-thread execution at scale, which is also exhibited by Bamboo. In our strong scaling study, messages are shrinking from 192KB to 24KB as the number of cores increases from



(a) Strong scaling study.



(b) The effect of virtualization on performance.

Fig. 4: 3D Jacobi. Grid size: 3072x3072x3072 double precision.

12228 to 98304. Since only 1 MPI process per NUMA-node is communicating, MPI+OMP and Bamboo variants aggregate the shorter messages into a smaller number of longer messages, compared to 1 MPI process per core with the pure MPI implementations. Since the network interface serializes messages longer than the eager limit, it makes sense that aggregation should benefit performance. However, why this effect appears suddenly at only at 96K cores is as yet unclear, and is currently under investigation. To eliminate effects due to TLB misses, we experimented with Cray's *hugepages*, but without effect. We monitored L1 and L2 cache miss rates using *CrayPat*, but could find no correlation. Unfortunately, L3 miss rates were not available.

The Bamboo and hand optimized MPI-olap variants deliver similar performance on up to 24K cores, but Bamboo's advantage rises sharply on 48K and 96K cores. We attribute the sudden change to how Bamboo handles decomposition. MPI-olap uses a hardwired scheme of 8 tasks per core, splitting the mesh assigned a core along all 3 dimensions. Bamboo has more flexibility than MPI-olap in selecting task geometry as it can virtualize along any of the dimensions. We experimented with many geometries and found that a virtualization factor of 2 tasks per core was optimal (Fig. 4(b).)

Bamboo-basic generally outperforms Bamboo+OMP since the runtime services run independently on one core while they have to share a core with an OpenMP thread in Bamboo+OMP. This is revealed in Fig. 4(b), which also shows that the benefits of communication-computation overlap in Bamboo+OMP drop off more quickly than Bamboo-basic as we increase the virtualization factor. Both variants benefit from modest amounts of virtualization (2 tasks per core), which improves the pipelining of communication and computation, but higher levels of virtualization introduce increased scheduling costs, overwhelming any improvements due to overlap.

C. Matrix Multiplication: Cannon's algorithm

1) Application and variant descriptions: Cannon's algorithm computes the matrix product of two matrices C=A*B, employing a square processor grid to partition the three matrices. Each processor owns a portion of C. It systematically rotates sub-blocks of A and B along processor rows and columns in a sequence of $\sqrt{P}-1$ steps, where P is the number of processors. At each step, after a processor receives the next submatrix of A and B, it computes a partial matrix product to update the C partition that it owns (C += $A$*B).

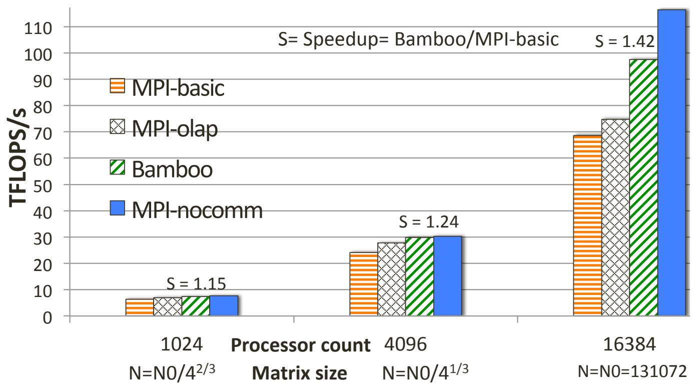
Fig. 5 shows the MPI-basic Cannon code annotated with Bamboo pragmas. We introduce a new layout called *OneNeighborBackward* (line 1) which establishes backward nearest neighbor connections between tasks in the right-to-left and down-to-up directions.

We also implemented an MPI-olap variant using a pipeline strategy that employed split-phase coding, advancing the message passing calls to enable overlap between the *dgemm* computation in one step and the matrix rotation of the next, which are independent.

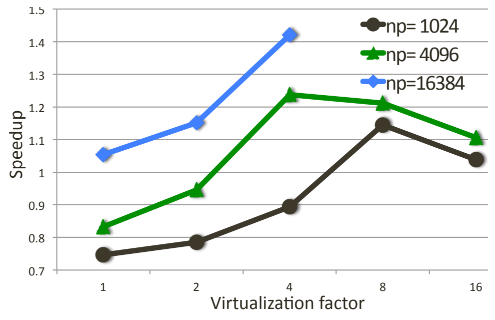
Unlike Jacobi's method, we did not use the MPI+OMP variant. It provided little advantage on the large matrices targeted by Cannon's algorithm. On the other hand, MPI+OMP was useful for the small matrices targeted by the 2.5D variant of Cannon (Communication Avoiding), which we present in the next section.

```
 1 #pragma bamboo olap layout OneNeighborBackward X Y
 2 dgemm(A, B, C);
 3 for(int step =1; step < sqrt(nprocs); step++){
 4   #pragma bamboo receive{
 5     MPI_Irecv(rA from right);  MPI_Irecv(rB from down); }
 6   #pragma bamboo send {
 7     MPI_Send(A to left);  MPI_Send(B to  up); }
 8   MPI_Waitall();
 9   swap(A, rA); swap(B,rB);
10   dgemm(A, B, C);
11 }
```

Fig. 5: Annotated code for submatrix rotation in Cannon's algorithm.



(a) Weak scaling study



(b) The effect of virtualization on performance.

Fig. 6: 2D Cannon with large matrices.

2) Aprun configuration: Since Cannon's algorithm requires a square processor grid, we ran with a square number of NUMA nodes, 4 processor cores on each node.[3] To this end, the MPI variants (MPI-basic and MPI-olap) were run with the following command line arguments, where *P* is the total number of cores: *-n P -N 16 -S 4*. For Jacobi 3D we ran Bamboo in *multi-threaded* mode with one MPI process per node and multiple worker threads per MPI process. However, *single threaded* mode was significantly faster for a CPU-bound application such as dense matrix multiplication, which runs one MPI process per core, and one worker thread per MPI process. Thus, we used same command line arguments as for the MPI variants.

3) Results: For this application, we conducted a weak scaling study. Since the application delivers a high fraction of peak performance on a single core (88%), strong scaling is of limited value. We used square matrices of size up to 128K $\times$ 128K.

Fig. 6 shows the performance of Bamboo and MPI variants on up to 16K processors. It can be seen that the communication cost increases steadily as the processor count increases. In particular, the cost increases from 17% on 1K processors to 41% on 16K processors. The reason is as follows. The number of communication steps grows as $\sqrt{P}$. Since the wallclock time spent in *dgemm* remains constant, and the size of the local sub-matrices A and B grows as $P^{2/3}$, the communication to computation ratio grows as $P^{1/6}$. In fact, the observed growth in communication is a bit higher as we've ignored the increase in message starts, which also grow as $\sqrt{P}$.

Under these conditions of growing communication costs, Bamboo speeds up the MPI-basic variant from $\times 1.15$ to $\times 1.42$, bringing performance closer to the upper bound of MPI-nocomm. To achieve these results, higher degrees of virtualization (from 4 to 8) were required compared to 3D Jacobi (2 in all cases). Fig. 6(b) shows the effect of virtualization on performance. As the degree of virtualization increases, there is a tradeoff between the benefit of overlap and the cost of scheduling the more numerous tasks, which are transmitting shorter and shorter messages. In 2D Cannon, the messages are long, e.g. 21MB on 1K processors and 8MB on 16K. Thus, we can continue to increase the virtualization factor, improving overlap without penalizing message bandwidth. By comparison, in the 3D Jacobi application the messages are much shorter (by over an order of magnitude), and there is less room to increase the virtualization factor. The cost of the transmitting the shorter, more numerous messages eventually overcomes the benefit of overlap.

The MPI-olap variant works well on up to 4096 processors, but at 16K cores it cannot compete with Bamboo. We believe that this is due to the ability of Bamboo to use virtualization to better pipeline the communication.

D. Matrix Multiplication: Communication-optimal algorithm

1) Application and variant descriptions: We have just demonstrated that Bamboo is able to mask communication in multiplying large matrices using Cannon's algorithm. We next look at the *2.5D* "communication avoiding" matrix multiplication algorithm [31], which targets small matrices. Small matrix products arise, for example, in electronic structure calculations (e.g. *ab-initio molecular dynamics* using planewave bases [25], [8]), a planned target of Bamboo. The 2.5D algorithm is interesting for two reasons. First, small matrix products incur high communication costs relative to computation, especially at large scales, which stress Bamboo's ability to mask communication delays. Second, the 2.5D algorithm introduces two new communication patterns: broadcast and reduction. Supporting these new patterns broadens the scope of Bamboo.

At a high level, the 2.5D algorithm generalizes the traditional 2D Cannon algorithm by employing an additional processor dimension to replicate the 2D processor grid. The degree of replication is controlled by a replication factor called c. When c=1, we regress to 2D Cannon. When $c = c_{max} = P^{1/3}$, 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 along the X and Y axes. In addition, the 2.5D algorithm performs a broadcast and a reduction along the Z dimension. Since

[3]We used 4 instead of 6 cores as the result of an early decision in conducting our scaling studies. The decision does not reflect any limitation of Bamboo as we ran on 6 cores per node with 3D Jacobi.

broadcast and reduction are closely related, we show only the annotated code of the broadcast routine, in Fig. 7. The *dimension* pragma (line 1) appears only once in the whole program. Whereas the 2D algorithm uses a 2D processor geometry, the 2.5D algorithm uses a 3D processor geometry. Broadcast is based on a *min heap* structure [13] constructed from the processors along the Z dimension. A min heap is a complete binary tree in which the parent's key (processor ID in our case) is strictly smaller than those of its children. The MinHeap layout annotation takes just one argument: the dimension along which we construct the MinHeap (Z). The broadcast algorithm has 2 communication blocks: one *receive* block and one *send* block. The *receive* block contains 1 *Recv* followed by 2 *Sends*. Since Bamboo will not reorder sending and receiving activities within a communication block it knows that the two *Sends* are dependent upon the completion of the *Recv*. However, following previous discussions about the independence of *send* and *receive* blocks, we infer from inspection that our annotations specify that all three point-to-point calls in the receive block are independent of all the point-to-point calls in the send block.

```
 1 #pragma bamboo dimension 3
 2 #pragma bamboo olap layout MinHeap Z
 3 {
 4    #pragma bamboo send
 5    {
 6       if(root & hasLeftChild) MPI_Send(A/B, leftChild);
 7       if(root & hasRightChild) MPI_Send(A/B, rightChild);
 8    }
 9    #pragma bamboo receive
10    {
11       if(!root & hasparent)  MPI_Recv(A/B, parent);
12       if(!root & hasLeftChild) MPI_Send(A/B, leftChild);
13       if(!root & hasRightChild) MPI_Send(A/B, rightChild);
14    }
15 }
```

Fig. 7: Annotated code for the broadcast routine in the 2.5D Cannon's algorithm.

Through experimentation, we observed that, with the small matrices targeted by the 2.5D algorithm, the hybrid execution model MPI+OMP yields higher performance than a "flat 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 uses the pipeline strategy discussed previously for the 2D algorithm. *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*.

2) Aprun configuration: In the 2.5D algorithm the number of processors $P = 2^c q^2$ for integers c and q. Thus, the number of cores is a power of 2, and we used 4 cores per NUMA node. All variants spawned MPI processes at the NUMA-node level to take the advantage of node-level parallelism using OpenMP. We ran all variants with the following *aprun* command line arguments: *-n p4 -N 4 -S 1 -d 4 -ss*, where *p4 = P/4*. The environment variable OMP_NUM_THREADS=4 in all runs.

3) Results: We conducted a weak scaling study on 4K, 8K, 16K and 32K processors. We chose problem sizes that enabled us to demonstrate the algorithmic benefit of *data replication*.

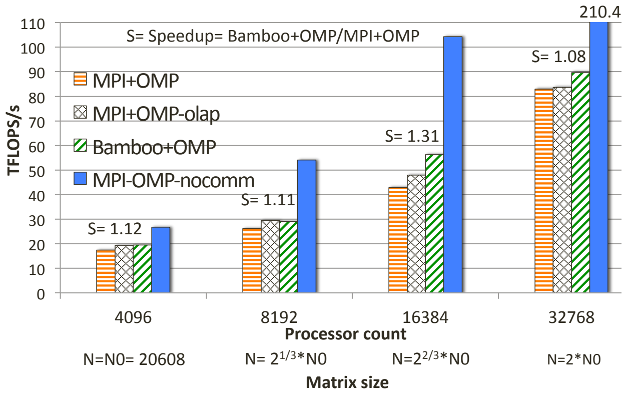


Fig. 8: 2.5D Cannon- Weak scaling study with small matrices.

#Cores	4096	8192	16384	32768
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 II: The effects of replication and virtualization. MPI+OMP and MPI+OMP-olap have limited options for c. The boldface values within the curly braces yield the highest performance. Bamboo's virtualization mechanism enables c to take on any power-of-two so long as we adjust the virtualization factor appropriately. Thus, for Bamboo+OMP we report performance for the combinations yielding the highest performance only.

Fig. 8 shows the results with the different variants. We measured the communication cost, which ranged from 35% to 61% (wallclock time). Both Bamboo+OMP and MPI+OMP-olap deliver the same speedup over the MPI+OMP variant on up to 8K processors. With 16K processors and 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.

To understand the interaction, we consult Tab. II, which shows the values of c that maximize performance for the different variants. Over the range of 8K, 16K and 32K cores, the efficiency of the MPI+OMP variant suddenly drops at 16K cores but then increases again at 32K. This variation is likely due to the effect of replication. Note that the 2.5D algorithm requires that the first two dimensions of the processor geometry be equal. For the two MPI variants (Tab. II), 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 8192 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. Thus, we choose a combination of replication and virtualization that is optimal and cannot choose these parameters independently.

We have demonstrated that Bamboo can improve the performance of communication avoiding matrix multiplication by overlapping communication with computation. Both techniques play an important role in highly scalable computing, where we must take into account a variety of performance tradeoffs, some of them algorithmic.

VI. Related work

MPI/SMPSs [24] employs a translator and programmer annotations to realize task parallelism at the function call level using SMPSs [27]. Under this approach, the programmer "taskifies" MPI calls, which may then run in parallel with computation, hence hiding communication delays. The translator generates a dataflow graph from information generated at run time. The task parallelism interface works for any routine and is not customized to MPI, though the scheduler is tuned to handle MPI programs. By comparison, Bamboo uses static analysis combined with domain-specific knowledge of the MPI API, enabling it to realize optimizations specific to MPI. For example, we can rewrite MPI collectives to reduce their granularity in the FFT [3], improving overlap. By comparison, MPI/SMPSs will taskify the monolithic collectives but is not able to restructure the MPI calls.

Danalis et al. [14] implemented transformations that realize communication-computation overlap in MPI collectives. Shires et. al [30] presented a program flow representation of an MPI program, which is useful in code optimization. Charm++ [21] supports virtualization and latency tolerance and includes a compiler. Adaptive MPI [18], built on top of Charm++, virtualizes MPI processes and supports communication overlap. 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.

PLASMA [23] 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's task graph library provides for task state, which curbs task graph growth.

We used the ROSE source-to-source translator [29] to develop Bamboo. ROSE is a member of the family of language processors that support semantic-level optimizations including Telescoping languages [22], [7] and Broadway [17]. 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.

VII. Conclusion and Future work

We have presented Bamboo, a translator to transform MPI code into a form that tolerates latency automatically. We demonstrated that Bamboo improved performance at scale on two important application motifs: structured grid and dense matrix linear algebra. 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 improve performance. Moreover, performance meets or exceeds that of labor-intensive hand coding, at scale. Bamboo also improved performance of Communication Avoiding matrix multiplication. 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.

Bamboo has some limitations, which we are currently addressing. First, it handles only the fundamental communication primitives which cover a small subset of MPI. We are adding support, for example, for communicators and for collectives. We plan to translate collectives into their components, i.e. point-to-point primitives, to take advantage of improved finer grained pipelining. Second, we have validated Bamboo against just two application motifs. In the future we plan to look new motifs such as the Fast Fourier Transform that exhibit vastly different communication patterns.

We implemented our approach with a custom source-to-source translator, which in effect treats the MPI API as an embedded domain specific language. Our translator, Bamboo, is more than a means of hiding latency that avoids costly code restructuring. It also serves as an example of the utility of semantic level optimization against a well known library.

Acknowledgments

The authors dedicate this paper to the memory of Allan Snavely. This research was supported in part by the Advanced Scientific Computing Research Petascale Tools Program of the U.S. Department of Energy, Office of Science, contract No. DE-ER08-191010356-46564-95715 and in part by NSF contract ACI0326013. Tan Nguyen is a fellow of the Vietnam Education Foundation (VEF), cohort 2009, and was supported in part by the VEF. Pietro Cicotti was also supported by NSF contract CCF1017864. 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, and Teragrid resources provided by NCSA, ORNL and SDSC. Eric Bylaska and Dan Quinlan acknowledge support by the ASCR Petascale Tools Program Contract No. DE-AC05-76RL01830. Support for computational resources housed in the Computer Science and Engineering Department at UCSD were supported by the National Science Foundation DMS/MRI Award 0821816.

References

[1] Arvind. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[2] R. G. Babb, II. Parallel processing with large-grain data flow technique. *Computer*, 17(7):55–61, July 1984.

[3] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 84–84, Washington, DC, USA, 2006. IEEE Computer Society.

[4] 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.*, pages 1457–1478, Dec 2001.

[5] D. Bonachea. Gasnet specification, v1.1. Technical Report CSD-02-1207, University of California, Lawrence Berkeley Laboratory, October 2002.

[6] G. Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[7] 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*, pages 148–155. IEEE, 2001.

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

[9] 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.

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

[11] P. Cicotti and S. B. Baden. 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.

[12] P. Colella. Defining software requirements for scientific computing, 2004.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2nd edition, 2001.

[14] 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*, pages 58–68, November 2005.

[15] J. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11):48–56, 1980.

[16] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. De Supinski, M. Schulz, and G. Bronevetsky. Formal analysis of mpi-based parallel programs. *Commun. ACM*, 54(12):82–91, Dec. 2011.

[17] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. *ACM SIGPLAN Notices*, 35(1):39–52, January 2000.

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

[19] P. Husbands and K. Yelick. Multithreading and one-sided communication in parallel lu factorization. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Reno, Nevada, Nov. 2007. ACM.

[20] L. V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[21] 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, pages 91–108, New York, NY, USA, 1993. ACM.

[22] 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*, 93:387–408, 2005.

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

[24] 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*, pages 5–16, 2010.

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

[26] Message Passing Interface Forum. MPI:A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[27] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142 –151, 2008.

[28] 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*, volume 5103 of *Lecture Notes in Computer Science*, pages 253–260. Springer Berlin / Heidelberg, 2008.

[29] 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.

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

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

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

[33] 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.*, 184:269–285, 2000.

[34] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[35] A. Zekri and S. Sedukhin. The general matrix multiply-add operation on 2d torus. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8 pp., april 2006.