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

Permalink
https://escholarship.org/uc/item/8fh786n4

Journal
Journal of Parallel and Distributed Computing, 106

ISSN
0743-7315

Authors
Nguyen, T
Cicotti, P
Bylaska, E
et al.

Publication Date
2017-08-01

DOI
10.1016/j.jpdc.2017.02.009

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

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

Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093, USA
Environmental Molecular Sciences Laboratory, Pacific Northwest National Laboratory, Richland, WA 99354, USA
Center for Advanced Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94550, USA

HIGHLIGHTS

- Bamboo is a translator that can reformulate MPI source into a task graph form.
- Bamboo supports both point-to-point and collective communication.
- Bamboo supports GPUs, hiding communication among GPUs and between hosts and GPUs.
- Bamboo speeds up applications containing elaborate data and control structures.

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 [55,29,57,4,61,60,32,22,20,33,58,40,21,38,46] or avoid them [56,5,42]. 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 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 [65,37,17], a common technique for masking communication overheads under computations. However, it does not 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++ [34]). 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 [46,44,45]. Bamboo transforms applications written in subset of MPI into a data-driven form that overlaps communication with computation automatically. Unlike other approaches [40,15] 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 does not 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 [13].

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 Scientific 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. Section 2 presents the Bamboo source-to-source translation framework. Section 3 discusses the design and implementation of the Bamboo source-to-source translator. Next, Section 4 presents experimental validation for various applications. Section 5 presents Bamboo support on advanced node technologies. Section 6 reviews related work. Section 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) [43] 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 [3,23]. A dataflow model has two appealing attributes. First, it can automatically mask data motion costs and hence improve performance without programmer intervention [6,20,33,58,19,21]. 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 [47], 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.

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 order enables Bamboo to reorder send and receive blocks. For example, Bamboo can move all send blocks up front and outside of the \textit{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 \textit{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 Section 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.

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 (\textit{struct, contiguous} and \textit{vector}). Bamboo does not support one-sided communication currently. Since Bamboo’s runtime requires that task graphs be run time static structures (Section 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 fine 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 [20,19].

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: eXecution (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, cannot 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 incurs 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.

Currently, Tarragon uses MPI to implement the communication handler. It uses non-blocking routines (MPI_Isend, MPI_Irecv,
Fig. 2. The Bamboo design and implementation.

Fig. 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.

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.

<table>
<thead>
<tr>
<th>Before reordering</th>
<th>After reordering</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1 #pragma bamboo olap</td>
<td></td>
</tr>
<tr>
<td>2 for (i=1;i&lt;niters;i++)</td>
<td></td>
</tr>
<tr>
<td>3 #pragma bamboo receive</td>
<td></td>
</tr>
<tr>
<td>4 (MPI_Irecv(rbuf0 ...));</td>
<td></td>
</tr>
<tr>
<td>5 MPI_Irecv(rbuf1 ...);</td>
<td></td>
</tr>
<tr>
<td>6 #pragma bamboo send</td>
<td></td>
</tr>
<tr>
<td>7 (MPI_Send(rbuf0 ...);</td>
<td></td>
</tr>
<tr>
<td>8 MPI_Send(rbuf1 ...);</td>
<td></td>
</tr>
<tr>
<td>9 MPI_Waitall(...);</td>
<td></td>
</tr>
<tr>
<td>10 Compute block</td>
<td></td>
</tr>
<tr>
<td>11 }</td>
<td></td>
</tr>
<tr>
<td>1 1=1</td>
<td></td>
</tr>
<tr>
<td>2 if(i&lt;niters){</td>
<td></td>
</tr>
<tr>
<td>3 MPI_Send(rbuf0 ...);</td>
<td></td>
</tr>
<tr>
<td>4 MPI_Send(rbuf1 ...);</td>
<td></td>
</tr>
<tr>
<td>5 }</td>
<td></td>
</tr>
<tr>
<td>6 #pragma bamboo olap</td>
<td></td>
</tr>
<tr>
<td>7 for (i&lt;niters){</td>
<td></td>
</tr>
<tr>
<td>8 #pragma bamboo receive</td>
<td></td>
</tr>
<tr>
<td>9 (MPI_Irecv(rbuf0 ...));</td>
<td></td>
</tr>
<tr>
<td>10 MPI_Irecv(rbuf1 ...);</td>
<td></td>
</tr>
<tr>
<td>11 MPI_Waitall(...);</td>
<td></td>
</tr>
<tr>
<td>12 Compute block</td>
<td></td>
</tr>
<tr>
<td>13 i=1</td>
<td></td>
</tr>
<tr>
<td>14 if(i&lt;niters){</td>
<td></td>
</tr>
<tr>
<td>15 MPI_Send(rbuf0 ...);</td>
<td></td>
</tr>
<tr>
<td>16 MPI_Send(rbuf1 ...);</td>
<td></td>
</tr>
<tr>
<td>17 }</td>
<td></td>
</tr>
</tbody>
</table>

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 Table 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 modifies the 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 Table 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.

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 Section 3.3.3. Besides 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.

$$\bigwedge (\neg C_i \lor m_i).$$

(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.

$$\bigvee (C_i \land \neg m_i).$$

(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.

![Fig. 4. A multigrid solver with call chains containing MPI invocations.](image)

**Fig. 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.

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.

$$\bigwedge (\neg C_i \lor m_i).$$

(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.

$$\bigvee (C_i \land \neg m_i).$$

(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.
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-nocomm 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 [56] targets small matrices. Small matrix products arise, for example, in electronic structure calculations (e.g. ab-initio molecular dynamics using planewave bases [41,13]). At a high level, the 2.5D algorithm generalizes the traditional 2D Cannon algorithm [49] 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 = cmx = nprocs/2, 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 cmx, 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.

Fig. 5 shows the results with the different variants. Both Bamboo+OMP and 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 Table 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 $O(n^2/\sqrt{p})$, where n is the number of cores. However, the effect of increasing v does not benefit from this cost function, since communication among virtualized tasks must be performed serially (hence p is not effectively change in that formula.) The effect is to improve pipelining as in the other applications. However, the number of messages is $O((2/(c^2) + \log(c)))$. 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 Table 3.
4.1.2. High Performance Linpack

The High Performance Linpack benchmark (HPL or Linpack for short) [26,25,24] 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. \textit{pdgesv0} does not make any attempt to overlap communication with computation, whereas \textit{pdgesvk2} applies an overlapping technique called \textit{lookahead}. We applied Bamboo annotations to \textit{pdgesv0}. Details of the 3 code variants are as follows.

The \textit{pdgesv0} code consists of 3 key operations: panel factorization \textit{pFact}, panel broadcast \textit{pBcast}, and the trailing submatrix update \textit{pUpdate}. \textit{pFact} finds the pivots in column panel \textit{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 \textit{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 (\textit{pBcast}). This is an efficient implementation that uses a ring broadcast algorithm, shifting data to the right along column processes. The \textit{pUpdate} operation swap-broadcasts \textit{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 \textit{pdgesvk2} variant applies \textit{lookahead} [26], a technique for overlapping communication with computation that fills idle gaps in the execution of LU. \textit{Lookahead} utilizes the dependence structure of the blocked algorithm to orchestrate computation and data motion. It uses split-phase coding [85], 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 [7] library. This predicament has motivated new algorithmic reformulations [14] or data-driven implementations [33,8,14,39] to realize overlap.

We annotated the \textit{pdgesv0} module and translated it with Bamboo. We also added scheduling policies via task prioritization using a bamboo priority pragma\(^1\) 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 \textit{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 await data from other tasks that have not yet completed. However, tasks with priority −1, 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 [45].

We performed experiments on Stampede [59], 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 \textit{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 \(k \times N\), where \(N\) is the number of panel columns of the input matrix and \(k\) is the number of communication events occurring 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 \textit{lookahead}. Compared to the no-\textit{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 [10,66,12,27] 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.

\(^1\) We did not present this pragma earlier since Bamboo simply translates the pragma into a method that sets task priority.

<table>
<thead>
<tr>
<th>#Cores</th>
<th>4K</th>
<th>8K</th>
<th>16K</th>
<th>32K</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI+OMP</td>
<td>(c = {1, 4})</td>
<td>(c = {2, 8})</td>
<td>(c = {1, 4, 16})</td>
<td>(c = {2, 8})</td>
</tr>
<tr>
<td>MPI+OMP-olap</td>
<td>(c = {1, 4})</td>
<td>(c = {2, 8})</td>
<td>(c = {1, 4, 16})</td>
<td>(c = {2, 8})</td>
</tr>
<tr>
<td>Bamboo+OMP</td>
<td>(c = 2, VF = 8)</td>
<td>(c = 2, VF = 4)</td>
<td>(c = 2, VF = 2)</td>
<td>(c = 4, VF = 2)</td>
</tr>
</tbody>
</table>
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 $8 \times 128^3$ boxes per core. The left part of Table 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 Table 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.

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.
5. Advanced node technologies

At present, it appears that further improvements to HPC systems will mainly come from enhancements at the node level [52,9,51]. Node architectures are changing rapidly, and systems will mainly come from enhancements at the node (i.e. coprocessors or accelerators) to amplify node performance is gaining traction. Bamboos 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 [44].

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 Bamboos 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, Bamboos 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 processes is called host–host communication. We extend Bamboos to hide the host–host and host–device communication overheads in MPI+CUDA applications.

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 × 512 × 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 [63,16]. 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 interconnection 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 result, 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 communication, 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. [22] 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 [31]. Shires et al. [53] presented a program flow representation of an MPI program, which is useful in code optimization. β-MPI [54] 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++ [35], KeLP2 [4,28], Adaptive MPI [32] (built on top of Charm++, Tarragon [20,19], Thyme [58], and others [55,57,61,60, 50,18,29].

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 [33] 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 [38] 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 [48] to develop Bamboo. Rose is a member of the family of language processors that support semantic-level optimizations including Teleco

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 rule, 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 manycore 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 conducted at ESMIL (Environmental Molecular Sciences Laboratory) at Pacific Northwest National Laboratory, operated for the U.S. Department of Energy by Battelle under Contract Number DE-AC05-76RL01830. 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).

References


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 Ph.D. 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.