OBJECT-ORIENTED ABSTRACTIONS FOR COMMUNICATION IN PARALLEL PROGRAMS

A Thesis

by

STEVEN MACK SAUNDERS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2003

Major Subject: Computer Science

OBJECT-ORIENTED ABSTRACTIONS FOR COMMUNICATION IN PARALLEL PROGRAMS

A Thesis

by

STEVEN MACK SAUNDERS

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

Lawrence Rauchwerger (Co-Chair of Committee)	Nancy Amato (Co-Chair of Committee)
Marvin Adams (Member)	Valerie Taylor (Member)
(Member)	(Head of Department)

May 2003

Major Subject: Computer Science

ABSTRACT

Object-Oriented Abstractions for Communication in
Parallel Programs. (May 2003)
Steven Mack Saunders, B.S., Texas A&M University
Chair of Advisory Committee: Dr. Lawrence Rauchwerger

This thesis details ARMI, a parallel communication library that provides an advanced implementation of the remote method invocation protocol (RMI), which is well suited to object-oriented programs. ARMI is a framework for expressing fine-grain parallelism in C++, and mapping it to a particular machine using shared-memory and message passing library calls. It handles low-level details such as scheduling incoming communication and aggregating outgoing communication to coarsen parallelism when necessary. These details can be adapted for different platforms to allow user codes to achieve the highest possible performance without manual modification. ARMI is used by STAPL, a generic, object-oriented, parallel C++ library, to provide a portable communication infrastructure. The basic design decisions are described, as well as a detailed analysis of the mechanisms used in the current OpenMP/Pthreads, MPI, and mixed-mode implementations. ARMI's performance is compared with both hand-coded Pthreads and MPI on a variety of machines, including an HP-V2200, Origin 3800, IBM Regatta and IBM RS6000 SP.

ACKNOWLEDGMENTS

This thesis would not have been possible without the aid of many other people. First, I would like to thank Lawrence Rauchwerger for his critical guidance and direction throughout this work. I would also like to thank my other committee members, Nancy Amato and Marvin Adams, for the opportunity to work on large projects under their expertise.

I greatly appreciate the support of my fellow graduate students in the PARASOL research group. Countless times I encountered implementation difficulties that they eagerly provided possible solutions to. I specifically want to thank my officemate, Gabriel Tanase, for the many hours of brainstorming on the dry-erase board, and Tim Smith, for selflessly giving his time to help me learn and use many new machines.

Finally, and most importantly, I would like to thank my wife, Melissa, for supporting and loving me throughout the many late nights and hours of hard work.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Communication is one of the most fundamental aspects of parallel programming. Not even the most embarrassingly parallel application can produce a useful result without some amount of communication to synchronize results. Unfortunately, expressing efficient communication is also one of the most difficult aspects of parallel programming.

Compounding the issues related to communication are the wide variety of parallel architectures. Modern architectures can be roughly split into two groups, those that provide a global address space and those that do not. Generally, the global address space is kept consistent between processors using a hardware cache coherence mechanism (e.g., Hewlett Packard V2200, Origin 3800, IBM Regatta), but may also be supported through software (e.g., Cray T3E). The alternative to a global address space is a set of private address spaces distributed among processors. Each address space may support only a single processor (e,g., cluster of workstations), or a larger number of processors (e.g., cluster of SMP's, IBM RS6000 SP). Although both architectures connect processors and memory using a network, the specific subsystem varies. The global address space is supported through the memory subsystem, whereas the distributed memory architectures generally use the I/O subsystem, which usually involves higher latencies due to more layers of abstraction in the operating system and network protocols. Correspondingly, the communication model used to program these architectures also varies.

The journal model is *IEEE Transactions on Automatic Control*.

A. Common Communication Models

The two most common models of communication in parallel programming are shared-memory and message passing. In shared-memory, a group of threads share a global address space. A thread communicates by *storing* to a location in the address space, which another thread can subsequently *load*. To ensure correct execution, synchronization operations are introduced (e.g., barriers, locks, and semaphores). The shared-memory model is considered easier to program, and is portable due to standards such as Pthreads [1] and OpenMP [2]. Furthermore, machines that implement this communication model do so with specific hardware that helps minimize overhead. Sometimes, however, its lack of explicit data distribution mechanisms can hinder scalability [3]. The biggest disadvantage of shared-memory has been its inapplicability to the largest machines. None of the massively parallel machines produced today support a single global address space. Worth noting is the solution provided by software distributed shared-memory (software DSM), which provides a software implementation of a global address space (e.g., [4]), albeit with performance penalties.

In the message passing model, a group of processes operate using private address spaces. A process communicates by explicitly *sending* a message to another process, which must use a matching *receive*. Synchronization is implied through the blocking semantics of sends and receives (e.g., a blocking receive does not return until the message has arrived). Portability has been assured by the adoption of the Message Passing Interface, MPI-1.1 [5]. MPI is currently supported by nearly all machines and scales up to massively parallel systems.

The message passing model is considered harder to program, because all sends and receives must be explicitly programmed in matched pairs. Furthermore, the user must keep track of the data distribution across the system, making dynamic or

irregular applications difficult to code. From a practical, programming experience point of view, one would prefer shared-memory if it were implemented on all systems.

There is, however, a more subtle distinction between these models. Programmers are aware of the higher latency of MPI communication and so tend to minimize its impact by using a coarse grain parallelization/communication model (e.g., using the Bulk Synchronous Processing model (BSP), such that a series of asynchronous computation supersteps are separated by global barriers [6]). This style tends to increase the critical path of programs by the time it takes to synchronize and communicate between each of these supersteps. One could argue that MPI programs scale well if the ratio of data size/processor stays above a certain value. However, if time to completion of an application of a fixed data size is the objective, then it is imperative to uncover and exploit the maximum amount of parallelism possible.

Most modern machines consist of a network of nodes, where every node is in fact another small parallel machine (i.e., a super-node). This implies that we need to exploit, concurrently, both coarse grain and fine grain parallelism. However, the widely adopted solution to writing portable code across platforms has been to only use MPI, and to simply implement MPI on shared-memory machines. Indeed, almost all shared-memory machines have very efficient MPI implementations. However, this approach still has the disadvantages of being slower than explicit shared-memory communication and being harder to code. Another serious, but possibly overlooked shortcoming of this approach is that MPI programs are, by design of the programmers, coarse grain and thus unable to exploit the fine grain parallelism available on each super-node of a large machine.

One-sided communication represents an improvement, by combining some of the strengths of shared-memory and message passing [7]. A set of processes operate using private address spaces as well as sections of logically shared-memory. A process com-

municates by *putting* information into the shared-memory, which another process can subsequently *get*. Because puts and gets operate asynchronously, and hence memory consistency is relaxed, synchronization operations are introduced (e.g., a fence blocks processes until all communication is complete). One-sided communication preserves some of the ease of shared-memory programming while maintaining the data distribution of message passing. Although it is still not widely used, several common implementations include SHMEM, ARMCI [8], LAPI [9] and the updated Message Passing Interface, MPI-2 [10]. Still, even with one-sided communication, the tendency is to write in the coarse grain model (e.g., to copy-in/copy-out large chunks of data for computation).

B. Remote Method Invocation

Remote method invocation (RMI) is an object-oriented communication model that is currently most often associated with Java [11]. It traces its origins to its function-oriented counterpart, remote procedure call (RPC) [12], which allows a process to invoke a function in a remote address space. RMI works with object-oriented programs, where a process communicates by requesting a method from an object in a remote address space. Synchronization is implied through the blocking semantics of RMI requests (e.g., Java RMI does not return until it completes [13]). Although RMI raises the level of communication abstraction by dealing with methods instead of directly accessing data by exposing the underlying shared-memory or message passing operations, it is generally associated with distributed applications, not high performance parallel applications [14, 15]. High performance run-time systems that do support RMI- or RPC-related protocols include Active Message [16], Charm++ [17, 18], Tulip [19], and Nexus [14]. Whereas Java RMI always blocks until completion to

Table I. Comparison of common communication models

Model	Address Space	Communication	Synchronization	Examples
Message Passing	private	matching send/receive	semantics of send/receive	MPI-1
One-Sided	private with shared sections	put/get	fences, locks	MPI-2, SHMEM, ARMCI
Shared-Memory	shared	load/store	barriers, locks, semaphores,	Pthreads OpenMP
Remote Method Invocation	local and remote objects	RMI request	semantics of RMI request	Java RMI, Charm++, Nexus

obtain the return value, many of the high performance implementations never block and never produce return values. Here, the only way to obtain the return value is through split-phase execution, where for example, object A invokes a method on object B and passes it a callback. When object B completes the RMI, it invokes object A again via the callback. Split-phase execution helps tolerate latency, since object A can do something else while it waits, but complicates programming.

Table I summarizes the discussion of common communication models. Each entry raises the level of abstraction, ending with RMI. We believe that RMI has several other advantages over the other models. It gives the immediate flexibility to either move data or work between processors, and thus can be more easily adapted to the needs of the application. In addition, using an RMI based communication package distances the programmer from the details of communication and its associated cost, and allows for a finer grain programming style.

C. Contribution

This thesis and the corresponding communication library, ARMI, make a number of contributions.

RMI-Based Communication - ARMI provides a style of communication, RMI, that takes advantage of the natural communication involved in object-oriented programs, the method invocation. It raises the level of abstraction of low-level message passing or shared-memory communication, and hence allows for easier parallelization. RMI also maintains data-hiding techniques, such as encapsulation, whereas other models must interact directly with data, bypassing the objects' interfaces.

Increased Functionality - ARMI supports both blocking RMI, to alleviate the need for difficult split-phase execution, and non-blocking RMI, for high performance. Since RMI's do not require matching operations as in message passing, incoming requests are scheduled internally and advanced synchronization mechanisms, similar to one-sided communication models, are provided.

Portable Framework for Expressing Fine-Grain Parallelism - The ARMI communication library also provides the definition and implementation of a framework for expressing "shared-memory style" fine-grain parallelism, and mapping it to a particular machine using shared-memory and message passing library calls. ARMI handles low-level details such as scheduling incoming communication and aggregating outgoing communication. These details can be tuned for different platforms to allow user codes to achieve the highest possible performance without manual modification. For instance, when shared-memory is available, aggregation settings may be set low. However, when only message passing is possible, aggregation may be set higher, which allows ARMI to issue requests in groups to better use the available bandwidth. Coarsening parallelism in this way allows ARMI to take maximum advantage of shared-memory while still scaling to larger message passing systems. In contrast, it is much more difficult for a library to attempt to break apart coarse grain parallelism for mapping on a shared-memory system.

Communication Infrastructure for STAPL - ARMI has been developed as part of

the run-time system for the Standard Template Adaptive Parallel Library (STAPL), a parallel superset to the C++ Standard Template Library, which provides generic parallel distributed containers and algorithms [20, 21].

CHAPTER II

THE STAPL PARALLEL PROGRAMMING ENVIRONMENT

ARMI was originally designed as a communication infrastructure for STAPL. However, it can also be used in any parallel C++ programming environment. We will now briefly present the STAPL programming environment and illustrate its capabilities through a simple example.

A. Overview

The C++ Standard Template Library (STL) is a collection of generic data structures with methods, called *containers* (e.g., vector, list, set, map), and *algorithms* (e.g., copy, find, merge, sort) [22]. Containers and algorithms are bound in terms of *iterators*. An iterator provides an abstract interface to a sequence of data, providing operations such as 'dereference current element', 'advance to next element' and 'test for equality'. Each container provides a specialized iterator (e.g., a vector provides a random access iterator, whereas a list provides just a bi-directional iterator). Since each algorithm is expressed in terms of iterators, instead of specific container methods, the same algorithm codebase is able to work with many different containers.

The Standard Template Adaptive Parallel Library (STAPL) is a sequentially consistent, parallel superset of STL [20, 21]. STAPL provides a set of parallel containers, *pContainers*, and parallel algorithms, *pAlgorithms*, that are bound in terms of *pRanges*. The pContainers provide a shared-memory view of physically distributed data. The pRange presents an abstract view of a partitioned data space. It provides a view of the distribution, random access to elements in the distribution, which is crucial for SPMD parallelism, and stores data dependencies between the elements. The pAlgorithms use pRanges to efficiently operate on data in parallel.

B. Programming Style

A STAPL user composes an application by specifying pContainers, initializing them as necessary, and then applying the appropriate pAlgorithms. The provided pContainers and pAlgorithms abstract any underlying communication. For example, dereferencing an element of a parallel vector may cause a remote miss, invoking an RMI to return the element. Similarly, a parallel sort will perform the necessary communication to permute the input to sorted order. If the necessary container or algorithm is not implemented, a more advanced STAPL user can implement their own.

The STAPL communication infrastructure, ARMI, provides a shared-object view of parallelism. Objects are distributed among the threads, where local communication occurs via regular C++ method invocation, and remote communication occurs via RMI. As defined in more detail in Chapter III, ARMI provides four fundamental communication operations: `async_rmi` to invoke a method, `sync_rmi` to invoke a method and wait for its return value, `broadcast_rmi` to invoke a method on all threads, and `reduce_rmi` to invoke a method on all threads and collect the results.

Because objects are conceptually shared, a fine-grain parallelization style is naturally expressible. For example, each element in the parallel sort can be transferred individually, allowing ARMI to aggregate as necessary, instead of hard-coding aggregation at the user level. A shared-memory system may be able to tolerate this high level of fine-grain communication, whereas a message passing system will likely perform poorly. Applying aggregation can reduce and even eliminate this performance degradation. As such, carefully tuning ARMI allows a fine-grain parallel program to efficiently exploit all possible parallelism on a shared-memory system, and automatically coarsen the parallelism via aggregation for a message passing system. In contrast, it is much more difficult for a library to attempt to break apart a coarse-grain

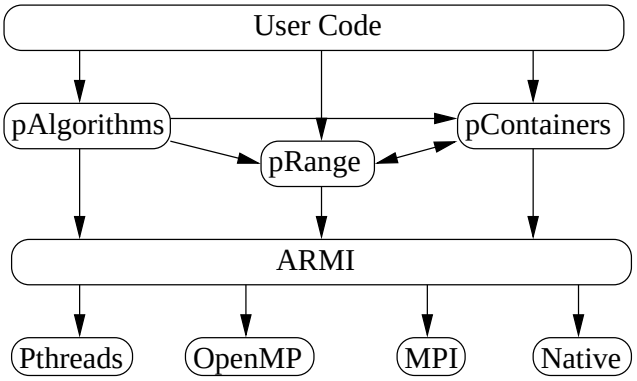

Fig. 1. STAPL component interactions

program into smaller chunks for mapping on a shared-memory system.

Using STAPL, a programmer is able to program in the easier shared-memory style, and expose as much parallelism as possible by using a fine-grain parallelization style. ARMI can be tuned to fully exploit the available parallelism in a shared-memory system, and perform the appropriate amount of aggregation in the message passing system. In addition, in environments such as clusters of SMP's, ARMI can employ mixed-mode communication by using shared-memory within nodes and message passing between nodes. Such systems may support lower aggregation settings within the node, and higher settings between nodes. Since many parallel algorithms utilize neighbor based communication, such a style can improve performance.

C. Component Interactions

Figure 1 shows the layout of STAPL's basic components, with arrows representing usage relationships. ARMI serves as the bottom layer, and abstracts the actual parallel communication model utilized via its RMI interface.

A pContainer is a distributed data structure. Although the user sees a single object, at run-time the pContainer creates one sub-pContainer object per thread

in which to actually store data. The pContainer's main job then is to maintain the consistency of the data it stores as the user invokes its various methods. Three remote communication patterns result:

1. *access* - a thread needs access to data owned by another thread (e.g., the dereference operation for a vector). The `sync_rmi` handles this pattern.

2. *update* - a thread needs to update another thread's data (e.g., the insert operation). The `async_rmi` handles this pattern.

3. *group update* - a thread needs to update the overall structure of the container (e.g., the resize operation). The `broadcast_rmi` handles this pattern.

Since pContainers' methods use RMI to implement these communication patterns, they effectively abstract the underlying communication seen by the user. An efficient library supporting both shared-memory and message passing might need to provide two versions of each container, one for shared-memory and the other for message passing. STAPL needs just one version of each pContainer by pushing the details and decision between shared-memory and message passing into the communication library. RMI also helps facilitate an easier implementation by relaxing the constraint of matching sends and receives, as in message passing.

A pAlgorithm expresses a parallel computation in terms of `parallel_task` objects. These objects generally do not use RMI directly. The specific input data per `parallel_task` are defined by the pRange, just as iterators define the input to an STL algorithm. Intermediate or temporary results that are used across threads can be maintained using pContainers within the `parallel_task`. As their methods are used to modify and store the results, the pContainers will internally generate the necessary RMI communication.

In the event that pContainers do not offer the necessary methods, RMI communication between `parallel_task`s is necessary. Three common communication patterns result:

1. *data parallel* - the same operation needs to be applied in parallel, possibly with a parallel reduction at the end.[1] A large percentage of STAPL algorithms utilize this pattern. For instance, in a `find`, each thread searches its local data for an element. Since multiple threads may find a match, a reduction is used to combine the results (i.e., thread 0's result has precedence over thread 1's, etc.). The `reduce_rmi` handles this pattern.

2. *event ordering* - computation dependencies must be satisfied. A small percentage of algorithms utilize this pattern, with master-slave computations being a common example. For instance, during a sequential depth-first search on a distributed graph, one thread begins the search on its local vertices while the other threads wait at an `rmi_fence`. As the search progresses to remote vertices, RMI can be used to tell the owning threads to continue the search on their local data. Parallelism may be exploited by performing several searches from different starting vertices, which is equally easily handled by ARMI.

3. *bulk communication* - a large number of small messages are needed. A small percentage of algorithms utilize this pattern, with sorting being a common example. The `async_rmi` operations handles this pattern, via its automatic aggregation settings.

[1] A reduction is an operation of the form $x = x \oplus exp$, where x is the current value of the tabulation, $\oplus$ is an associative operation for performing the tabulation, and exp is a process's local contribution.

Each level of STAPL serves to further remove the user from the underlying communication. ARMI provides the fundamental abstraction between shared-memory and message passing. The pContainers build upon this to create distributed data structures with a shared-memory interface. The pAlgorithms use pRanges, pContainers, and RMI when necessary, to create useful parallel algorithms. The user combines pContainers and pAlgorithms to write a program, without worrying about the underlying communication.

D. Case Study: Parallel Sorting

To illustrate how different parallel programming models affect communication, consider a common parallel algorithm for sorting: sample sort [23]. Sample sort consists of three phases:

1. Sample a set of $p-1$ splitters from the input elements.
2. Given one bucket per processor, send elements to the appropriate bucket based on the splitters (e.g., elements less than splitter 0 are sent to bucket 0). Because they are distributed based on sampled data, buckets will have varying sizes, and hence sample sort is highly dynamic.[2]
3. Sort each bucket.

Consider the following code fragments, which present implementations using fine-grain shared-memory and coarse-grain message passing. These fragments are for illustration only and do not necessarily represent the best possible implementations. Assume the input has already been generated and, for message passing, distributed.

[2]Most implementations oversample the input to increase the chance of balanced buckets. This sub-step has been removed for simplicity.

```
1  // shared-memory sample sort
2  void sort(int* input, int size) {
3    int p = //...number of threads, 0-p...
4    std::vector<int> splitters( p-1 );
5    std::vector< vector<int> > buckets( p );
6    std::vector<lock> locks( p );
7    for( int i=0; i<p-1; i++ )
8      splitters[p-1] = //...sample input...
9
10   //...fork p threads...
11   int id = //...thread id...
12   for(i=size/p*id; i<size/p*(id+1); i++) {
13     int dest = //...appropriate bucket...
14     locks[dest].lock();
15     buckets[dest].push_back( input[i] );
16     locks[dest].unlock();
17   }
18   barrier();
19
20   sort( bucket[id].begin(), bucket[id].end() );
21 }
```

```
1  // message passing sample sort
2  void sort(int* input, int localSize) {
3    int p = //...number of processes, 0-p...
4    std::vector<int> splitters( p-1 );
5    std::vector<int> bucket( p );
6    int sample = //...sample input...
7    Gather( &sample, ..., splitters, ... );
8
9    std::vector< std::vector<int> > outBucket( p );
10   for( i=0; i<localSize; i++ ) {
11     int dest = //...appropriate bucket...
12     outBucket[dest].push_back( input[i] );
13   }
14   for( int i=0; i<p-1; ++i )
15     Send( outBucket[i] ... );
16   for( int i=0; i<p-1; ++i ) {
17     Recv( tmp ... );
18     bucket.insert( tmp.begin(), tmp.end() );
19   }
20
21   sort( bucket.begin(), bucket.end() );
22 }
```

In general, shared-memory algorithms are sequential until a fork (line 10), whereas message passing algorithms are always in parallel. The shared-memory code uses a shared STL vector to communicate splitters before forking (lines 7–8), as opposed to the message passing library calls (lines 6–7). Shared-memory must calculate each thread's local portion after the fork (lines 10-12), whereas in message passing, data must be manually distributed *a priori*. Shared-memory shares the buckets by locking each insertion to ensure mutual exclusion (lines 14-16), and uses a barrier (line 18) to ensure proper event ordering of the distribution and sorting phase. Message passing buffers all the communication to each destination locally (line 12), and performs a single large communication phase (lines 14-19), which implicitly ensures event ordering.

Neither implementation is optimal. Shared-memory makes extensive use of locking (one lock per element), causing contention on the buckets. Message passing performs computation and communication in separate phases, which eliminates communication/computation overlap and increases the critical path of the code. These issues are not intrinsic to the sample sort algorithm, only to the underlying communication model and subsequent implementation. Improvements can be made at the expense of additional lines of code, which are even further removed from the basic algorithm.

We now contrast these implementations with STAPL. Since STAPL provides a parallel superset to STL, it contains a pAlgorithm for sorting, `p_sort`, which a user can use directly. The following code fragment illustrates a possible implementation of `p_sort`. Note that additional code is used to wrap the algorithm in a class. The heart of the algorithm (contained in the `execute` method) is actually shorter than either of the previous implementations.

```
1  // STAPL sample sort
2  struct p_sort : public stapl::parallel_task {
3    int *input, size;
4    p_sort(int* i, int s) : input(i), size(s) {}
5
6    void execute() {
7      int p = stapl::get_num_threads();
8      int id = stapl::get_thread_id();
9      stapl::pvector<int> splitters( p-1 );
10     stapl::pvector< vector<int> > buckets( p );
11     splitters[id] = //...sample input...
12     stapl::rmi_fence();
13
14     for( i=0; i<size; i++ ) {
15       int dest = //...appropriate bucket...
16       stapl::async_rmi( dest, ...,
17          &stapl::pvector::push_back, input[i] );
18     }
19     stapl::rmi_fence();
20
21     sort( buckets[id].begin(), buckets[id].end() );
22   }
23 }
```

The STAPL code maintains the shared-memory implementation's fine-grain approach by sending each element as it becomes available (lines 16–17). However, ARMI abstracts the mutual exclusion, and so explicit locking operations are removed. This allows the underlying implementation to aggregate requests as necessary, making the code much closer to optimal. For example, a tightly-coupled shared-memory machine may use a low aggregation factor, whereas a large distributed memory machine may use a larger setting, possibly even aggregating all messages, thus becoming the coarse grained message passing implementation at run-time.

The `rmi_fence` ensures event ordering by waiting for all `async_rmi`'s to complete before proceeding (lines 12 and 19), regardless of aggregation settings. Any remaining communication will be scheduled for execution within the `rmi_fence`.

CHAPTER III

DESIGN

The main goals of ARMI are to provide an easy to use, clean means of expressing parallelism in STL-oriented C++ code, while facilitating efficient implementation on many different parallel machines. ARMI is composed of three main interface subsystems: object registration, communication and synchronization. This section describes fundamental requirements for ARMI, the three main subsystems, as well as additional details related to data transfer, global variables and support for mixed-mode parallelism.

A. Requirements for Parallelism

We recognize two fundamental types of communication in a parallel program, regardless of programming model:

1. *statement* - a process needs to tell another process something (e.g., a result or to perform some action, as in a producer-consumer relationship). A statement is asynchronous, meaning the sending process does not generally wait for the receiving process to receive or process the information.

2. *question* - a process needs to ask another process for something (e.g., a result, which may or may not be calculated *a priori*). A question is synchronous, meaning the sending process must wait for the receiving process to process the information and reply.

In either case, the receiver may not necessarily expect the communication, as in a dynamic or irregular program. Each communication type can also be abstracted

to handle multiple processes at once, making, for example, a statement a broadcast (i.e., tell many processes something) and a question a reduction (i.e., ask a question and tabulate the answers).

Closely related to communication is synchronization, which also has two fundamental forms [24]:

1. *mutual exclusion* - operations to ensure modification to an object are performed by one process at a time. This is explicit in shared-memory (e.g., locks), and implicit in message passing, where all memory is private to a process.

2. *event ordering* - operations to inform a process or processes that computation dependencies are satisfied. This is explicit in shared-memory (e.g., semaphore signal and wait operations), and implicit in message passing, via the semantics of message sending and receiving.

Cleanly expressing these types of communication and synchronization are requirements for a parallel programming model's success. Shared-memory and message passing both fulfill them all, albeit in somewhat different ways. One goal of ARMI is to raise the level of abstraction of these issues, yielding a clean interface that lends itself to efficient implementation with either underlying model. As such, ARMI is based on the higher level RMI abstraction. RMI deals directly with an object's methods, and hence maps cleanly to object-oriented C++.

B. Object Registration

ARMI provides SPMD, shared-object parallelism. Each C++ object is associated with a single thread. Threads may share an address space (e.g., OpenMP), have a separate address spaces (e.g., MPI), or a mixture of the two (e.g., mixed-mode MPI

with OpenMP). Regardless of the granularity of the address space, ARMI names the unit of execution a thread. Upon startup, all threads begin executing the same code in parallel, as in SPMD MPI.

The shared-objects used in communication are distributed among threads, with each thread owning a local representative object. Shared-objects are identified by an `rmiHandle`, and their local objects are identified by thread id and `rmiHandle`. As such, each object that is a communication target must be registered with ARMI to obtain an `rmiHandle`, which allows for proper address translation to the local objects. Since each thread owns its local objects, it is not necessary to use RMI to access them, even for mutual exclusion. The following functions illustrate the operations provided for object registration.

1. `rmiHandle register_rmi_object(objectPtr)` - register the given local object pointer.

2. `void update_rmi_object(rmiHandle, objectPtr)` - update the location of a previously registered object. Supports objects that may move during the course of the computation (e.g., using realloc).

3. `void unregister_rmi_object(rmiHandle)` - free resources associated with a previously registered object.

Figure 2 shows a simple example with two separate shared-objects, Circle and Square, distributed among four threads. As described in more detail in Chapter IV, Section A, each thread independently registers its local representatives of the global Circle and Square objects during the course of its execution. The `rmiHandle` assigned for Circle will be the same for all threads, as will the handle for Square. These handles allow threads to independently communicate with other threads' local objects using RMI.

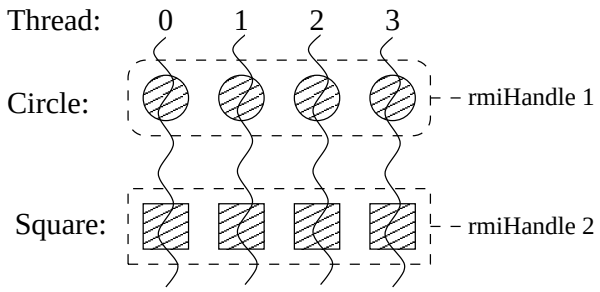

Fig. 2. Registration of two shared-objects by four threads

Although most communication is between shared-objects, ARMI also provides a collective function, `execute_parallel_task`, such that each thread automatically registers the specified `parallel_task` function object and begins execution using its `execute` method. This facilitates computations with extensive event ordering, such as master-slave or producer-consumer codes.

C. Communication

ARMI defines two basic forms of RMI, which map directly to the two fundamental types of communication.

1. `void async_rmi(destThread, rmiHandle, methodPtr, arg1...)` - makes a statement. The call issues the RMI request and returns immediately. Subsequent synchronization calls, such as `rmi_fence`, may be used to wait for completion of requests. Because it returns immediately, it is possible to aggregate multiple requests together, coarsening outgoing communication as necessary for a given machine.

2. `rtnType sync_rmi(destThread, rmiHandle, methodPtr, arg1...)` - asks a question. The call issues the RMI request and waits for the answer. Since it

waits for a return value, it is not possible to aggregate multiple `sync_rmi`'s, although a single `sync_rmi` may be transferred with an aggregated group of `async_rmi`'s.

The additional information required compared to a regular C++ method invocation is minimal. Only `destThread` is completely new, which specifies the destination thread. Whereas regular C++ method invocation uses object references or pointers, ARMI uses `rmiHandle`'s to facilitate proper translation between threads. The `methodPtr` is a pointer to a method defined by the object registered with the given `rmiHandle`. Since ARMI currently only supports an SPMD style of execution, this call-by-name model is possible, because all threads have a copy of all the code. Finally, the necessary arguments are specified.

ARMI makes several communication guarantees. First, RMI requests always maintain order, i.e., a newer request may not overtake and execute before an older request. However, there is no guarantee of fairness between threads. For example, although threads 1 and 2 may simultaneously issue multiple requests to thread 3, thread 3 may receive all of thread 1's requests before receiving any of thread 2's requests. These guarantees are consistent with most other communication libraries, including MPI.

ARMI also incorporates many-to-one and one-to-many communication to support common collective communication patterns.

1. `void broadcast_rmi(rmiHandle, methodPtr, arg1...)` - makes a statement to all threads. The call issues an RMI request from one thread, to be executed by all other threads.

2. `void reduce_rmi(rmiHandle, methodPtr, input, output)` - asks a question on all threads and collects the results (i.e., a reduction). The call issues the RMI

request and waits for the answer.

As currently defined, these operations are globally collective, in that all threads must perform the operation before any thread will release and continue. However, there is no fundamental reason a subset of threads could not optionally be specified and used. The subset could be defined in a style similar to MPI communicators, where threads explicitly register themselves with a given subset group, allowing for straightforward implementation via MPI. Alternatively, subsets could be expressed by mathematical functions (e.g., every odd numbered thread).

There is also no fundamental reason these operations need to be two-sided. For instance, a `broadcast_rmi` variant could be devised that issues a request to all other threads, without requiring them to also perform the broadcast. This is easily implemented as a series of `async_rmi`'s, but could be optimized further for certain systems.

D. Synchronization

ARMI also addresses both forms of synchronization. To ensure mutual exclusion, methods invoked by RMI execute atomically. Hence, a method invoked by multiple threads in parallel preserves thread-safety by acting as a monitor. In cases where additional RMI operations are used within the remotely invoked method, everything before the operation is atomic, as well as everything after the operation.

Event ordering is supported in two ways. The `rmi_wait` operation is provided to allow a thread to wait for the next incoming RMI before proceeding. The `rmi_fence` operation is provided to allow threads to wait until all other threads have arrived and completed all pending RMI communication. Like the collective communication operations of the previous section, `rmi_fence` is currently globally collective, in that all threads must participate. However, there is no fundamental reason a subset of

threads cannot be used instead.

The `rmi_fence` is a significant advancement compared to a typical barrier. Specifically, whereas a barrier simply blocks until all threads have arrived, `rmi_fence` continues to poll for incoming RMI requests. This allows for straightforward implementation of master-slave computations, where the slaves wait at the fence while the master dictates work via RMI. There is no limit to the amount of work performed within `rmi_fence`, and received requests are able to issue additional requests to other threads, allowing for extremely complicated communication patterns to occur. In the master-slave example, originally only one thread may be the master. As the computation progresses however, the master thread may change, multiple threads may act as masters, or threads may act as masters and slaves, both delegating and receiving work. Since `rmi_fence` guarantees not to return until all communication is complete, it handles the termination detection scheme that would often be overlayed for such algorithms. Chapter V, Section D, details the performance of a parallel algorithm for detecting strongly connected components in a graph, which makes use of these advanced facilities of `rmi_fence`.

E. Data Transfer

In ARMI, only one instance of an object exists at once, and it can only be modified through its methods. The granularity of data transfer is the smallest possible, the method arguments, and arguments are always passed-by-value to eliminate sharing. As such, ARMI avoids data coherence issues common to some DSM systems, which rely on data replication and merging. In essence, RMI transfers the computation to the data, allowing the owner to perform the actual work, instead of transferring the data to the computation.

To support message passing as an implementation model, ARMI requires that each user-defined class implement a single method, `define_type`. This method defines the class's data members in a style similar to Charm++'s PUP interface [17, 18]. Each data member is defined as local (i.e., automatically allocated on the stack), dynamic (i.e., explicitly allocated on the heap using `malloc` or `new`), or offset (i.e., a pointer that aliases a previously defined variable, for example STL vectors often maintain a dynamic begin pointer, and an offset end pointer, aliasing the end of the currently used space). This method may then be used as necessary to adaptively pack, unpack, or determine the type and size of the class based on ARMI's underlying implementation.

Figure 3 demonstrates a simple example class. The objectA class contains two locally defined variables, an array of doubles and an objectB. The objectB class stores a local integer size and a dynamically allocated integer array of that size. The `typer` is used during packing. If objectA is being transferred as an argument, ARMI will internally create a `typer` and the `define_type` method will be called (line 4). On line 6, `typer` will recursively call the `define_type` for objectB (line 13), to ensure the entire object is correctly packed.

F. Global Variables

Because the ARMI interface is designed to allow implementation using a variety of different models, non-constant global variables are not allowed, including static class members. This is because of an inconsistency in run-time behavior: as implemented in shared-memory, there will be a single shared copy of the global, whereas in MPI there will be one copy per thread. If globals are updated by multiple threads, their run-time values will vary based on the underlying implementation. For example, in shared-memory, race conditions will occur, and in message passing, threads will not

```
1  class objectA {
2    double a[10];
3    objectB b;
4    void define_type(stapl::typer& t) {
5      t.local( a, 10 );
6      t.local( b );
7    }
8  }
9
10 class objectB {
11   int size;
12   int* array;
13   void define_type(stapl::typer& t) {
14     t.local( size );
15     t.dynamic( array, size );
16   }
17 }
```

Fig. 3. Example of the `define_type` interface

see each others' local updates.

Since global variables are not uncommon in user programs, it would be useful to provide some support. Simple extensions to the current ARMI could allow a modifier to be added to each global, allowing shared-memory implementations to make the global per-thread [25]. The drawback is that, conceptually, a global variable often makes the most sense when shared among all threads. In this case, a special global variable pContainer could be created that performs the necessary mutual exclusion and data coherence between threads. The drawback here is that it introduces the data coherence issues ARMI was specifically designed to avoid, and can make access to global variables extremely expensive.

G. Mixed-Mode

Mixed-mode parallelism combines shared-memory and message passing into a single model to increase performance. Clusters of SMP's are especially well suited to such a model, because each SMP super-node provides shared-memory communication internally, while only message passing is possible between super-nodes. The goal is that leveraging the shared-memory will increase performance versus using only using message passing. There are two common approaches to providing mixed-mode parallelism: multi-protocol and nested parallelism.

In multi-protocol parallelism, shared-memory communication is used between threads within the local address space, and message passing between all other threads. However, the view of parallelism is flat, or horizontal, in that all threads are aware of, and able to communicate directly with, all other threads.

Nested parallelism is an extension to existing parallel models that allows currently executing parallel regions to create new parallel regions. These nested parallel regions could in turn create additional parallel regions. In contrast to multi-protocol parallelism, nested parallelism provides a vertical view of parallelism, such that threads may only directly communicate with their siblings and parents. A nested thread can pass communication requests to its parent, which can in turn pass the request to the proper sibling. The sibling can then delegate the request to the proper nested child.

Multi-protocol parallelism has the advantage that only one level of parallelism need be expressed to obtain a performance boost on clusters of SMP's. Nested parallelism requires that multiple parallel regions be expressed, which may not always be applicable, with the benefit that the resulting parallelism will better map to the underlying system. For example, a multi-protocol code may incur a larger amount of message passing communication than a nested parallel code, which can map each

nested region, which only uses sibling communication, to an SMP, and only use message passing between nested regions.

As currently defined, ARMI does not directly support nested parallelism, meaning that new threads cannot be created at run-time. However, multi-protocol parallelism is supported, and provides a consistent interface with the shared-memory or message passing only implementations. There is no fundamental reason that extensions to ARMI could not support nested parallelism in the future. The biggest issue requires specifying where the nested threads will be created and destroyed. A simple modification to `execute_parallel_task` could accept the number of nested threads desired when executing the task, and perform the creation internally. Other issues include how to number the nested thread ids in relation to the parent thread, and how to handle visibility of nested threads to other threads.

CHAPTER IV

IMPLEMENTATION

Figure 4 shows the layers of an ARMI implementation. As detailed in Chapter III, ARMI presents three main interface subsystems to the user: object registration, communication and synchronization. These subsystems abstract the interactions of the ARMI internals, which in turn abstract the underlying communication model(s).

ARMI has been implemented using two different underlying communication models: Threads (shared-memory) and MPI-1.1 (message passing). The Threads implementation determines at compile-time whether to use Pthreads or OpenMP. In addition, ARMI provides a mixed-mode implementation, which combines the two models for systems such as clusters of SMP's. Because of the number of subtle differences between shared-memory and message passing, the only layer shared without modification between the implementations is the RMI registry. As such, the RMI request and transfer/scheduling layers are reimplemented and tuned for each implementation. Although details could be further abstracted to reduce the number of layers requiring modification when porting, such abstractions would hinder performance, especially for shared-memory implementations. For example, as described in Chapter IV, Section D, only MPI uses the `define_type` interface, whereas Threads use less costly alternatives.

In the rest of this section, we describe the mechanisms used to implement each layer of the ARMI implementations.

A. Object Registration and RMI Registry

All objects that may serve as communication targets must first be registered. Chapter III, Section B, details the object registration interface. The main requirement

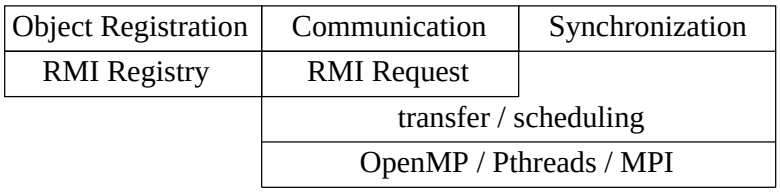

Fig. 4. Layers of an ARMI implementation

for an implementation is that each local object of a shared-object be assigned the same `rmiHandle`. Currently, an SPMD style of programming is assumed, such that each thread will register the same local objects in the same order. In the case of the example given in Figure 2, this means that all threads will always register Circles before Squares.

Although this object symmetry constraint may be restrictive in some cases, it facilitates a simple and efficient implementation in the RMI registry layer. Each thread maintains a private RMI registry, which is comprised of a vector of pointers to objects. When an object is registered by a thread, the next available entry in the vector is set to point to the given object, and the corresponding index is returned as the `rmiHandle`.

To facilitate finding the next available entry in constant time for each registration, unused entries form a free list, such that each unused entry points to the next unused entry. This free list functions as a stack, to improve locality and reduce the number of holes that occur after a large number of insertions and deletions. Registration simply deletes the head of the free list, and uses its index as the next `rmiHandle`. Unregistration reinserts the given `rmiHandle` as the head of the free list.

Since all threads register the same objects in the same order, the resulting `rmiHandle`s will always be the same on all threads, without requiring additional negotiation communication. This assumption also coincides with the current requirements

and implementation of pContainers. Unfortunately, it allows users to get undefined behavior if they do not satisfy it, and is not general to all possible situations.

It is not difficult to relax constraints, at the expense of more expensive registration and lookup procedures. To support subsets of threads registering objects, a negotiation protocol could be added that determines the next `rmiHandle` available to all participating threads. It is likely that the resulting handles would also require hashing during lookup, as opposed to the simple indexing scheme, to prevent large numbers of empty entries in the threads' vectors. To support out-of-order registration, a global handle generator could be defined, which assigns the same handle to each given name (assuming names are always distinct).

B. Communication

The communication subsystem is the most complicated interface to implement, and relies on the most underlying layers. Chapter III, Section C, details the interface as seen by the user. Similar to STL, at the top level, each communication function makes extensive use of C++ templates to specialize compile-time functionality to the exact types required for arguments and return types. To prevent code-bloat, the top-level interface is kept thin by pushing most of the code into lower layers. Each function simply creates the necessary RMI request object, discussed in Chapter IV, Section A, which handles all other details.

The most complicating factor of the communication interface layer is providing sufficient functions to handle as many arguments as required by the user. Unfortunately, the only robust solution we have found is to simply create a version of each function for 0 arguments, a version for 1 arguments, a version for 2 arguments, etc. Although this works and provides excellent performance, it makes maintenance painful

because of the large amount of replicated code. C-style solutions such as va_args, which allow a function to accept a variable number of arguments (e.g., `printf`), do not work because C++ objects aren't supported, and type deduction is not possible without additional user intervention. Preprocessor solutions can potentially reduce replicated code at the expense of additional layers of confusing C-style macros. Note that the RMI request classes, discussed in Chapter IV, Section D, encounter this same problem, and also replicate code as the solution.

C. Synchronization

The synchronization subsystem is significantly smaller than the communication subsystem, and only interacts with the transfer/scheduling layer. Chapter III, Section D details the interface as seen by the user.

The `rmi_wait` function directly interacts with the transfer/scheduling layer to wait for a single incoming request and schedule it for execution immediately. Because of the asynchronous nature of ARMI, `rmi_wait` actually waits for the next incoming request since the previous `rmi_wait` or `rmi_fence` call. If that request has already arrived and executed since the previous call, `rmi_wait` returns immediately. This prevents deadlocks in cases where multiple incoming requests are scheduled for execution at once. To implement this behavior, the scheduling layer is instrumented to increment a variable each time a request is executed, such that `rmi_wait` can see if this variable has changed since its last invocation, and execute accordingly.

The `rmi_fence` is a collective operation similar to a barrier. However, it does not release until all threads arrive and complete all outstanding communication requests. There are two complicating issues for a fence versus a barrier. First, to ensure correct execution, threads waiting at the fence must continue to poll for RMI requests.

Second, the fence protocol must correctly determine when all RMI request transfers have completed. This issue is further complicated by the fact that one RMI request could invoke a second request, which in turn invokes a third request, etc.

Most vendors provide blocking barriers, which are unsuitable for incorporating polling [19]. As such, we were forced to implement our own fence. To address the second issue, we overlay a distributed termination detection algorithm, which guarantees not to terminate until all messages have settled, with minimal overhead [26]. The algorithm tracks the number of sends minus receives performed by each thread, performs a global summation of those sends minus receives, and declares termination as soon as the global sum equals zero. To track sends and receives, the transfer layer is instrumented to increment/decrement a variable for each send/receive. The global summation is a parallel reduction inserted within the fence protocol, which iterates until the global sum equals zero.

We implemented a tree-based fence for Threads, proposed as Algorithm 11 in [27]. It was modified to poll while busy-waiting for arrival/release messages from parents and children. In addition, each parent sums its children's local sends minus receives. Upon release, all threads read the root's global value for sends minus receives to determine if termination has been detected.

We also implemented a generic tree-based fence for MPI, where children send an arrival message to their parents upon arrival, wait for a release message while polling for other incoming RMI's, and then propagate the release to their children. The communication pattern was modified to include the send minus receive sum within the arrival/release messages. We implemented three different tree patterns: a flat tree with a root and all other threads as leaves, a standard binary tree (0's children are 1 and 2, 1's children are 3 and 4, etc.), and a binary tree optimized for a hypercube. In general, the hypercube tree performed best, although the flat tree worked well when

message latency is extremely high.

D. RMI Request

RMI requests are encapsulated by *functors* [28]. A functor stores an RMI request's `rmiHandle`, method, and argument information, and allows the request to be stored, transferred, and subsequently executed. The base functor class only provides a virtual method to execute itself, whereas derived functors are specialized based on number of arguments and whether the return value is needed (e.g., `sync_rmi`). To preserve C++'s strong typing system, all functors make extensive use of templates.

As described in Chapter IV, Section A, the `rmiHandle` is really just an integer index. As stated in Chapter III, Section C, ARMI currently only supports an SPMD style of execution, and hence method pointers are sufficient for storing method information, since all threads have a copy of all the code. Should ARMI support the more relaxed MPMD style of execution in the future, where all threads may not have a copy of all the code, this approach will not be adequate. Specifically, the RMI request may need to package the necessary code for transfer, along with the argument information, since the destination thread might not recognize the given method pointer.

Storing argument information is more involved. Shared-memory RMI requests use each arguments' copy constructor to make a copy before transfer. However, copy constructors do not imply creating the object into contiguous memory (e.g., use of `malloc` or `new` for dynamic members), which is a requirement for message passing. As such, when message passing is used, the `define_type` interface is used to serialize dynamic arguments (see Chapter III, Section E).

Figure 5 shows the packed representation of objectA from Figure 3 as an argument to a single argument RMI. It is not drawn to scale. The grey regions are optional

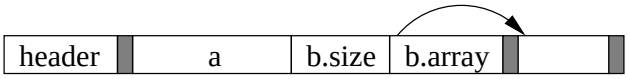

Fig. 5. Packed representation of Figure 3

padding that may be applied to ensure each section of the request is properly aligned in memory (e.g., doubles usually need to be aligned on word boundaries).

As a first pass, `define_type` is used to determine the exact space requirements of the RMI request's arguments. The necessary space is then allocated by the transfer layer (see Chapter IV, Section E for details). First, the header is copied, which contains the total size, `rmiHandle`, and method pointer, and is generally 16–24 bytes, including padding. Next, arguments are copied. ObjectA is copied into contiguous memory next to the header, and a second pass of `define_type` is used to pack its dynamic members. The dynamic member's, `b.array`, contents will be copied into contiguous memory during this pass. The absolute address of `b.array` is converted to a relative offset from the beginning of the header. After transfer to the destination communication buffer, and before execution, a third pass of `define_type` is used to unpack the object by simply adding the relative offsets to the request's new location in memory. Adjusting offsets eliminates the more costly unpack stage common to many other libraries, whereby the object is copied and reconstructed from the communication buffer to a new location in memory before execution. The only caveats to this optimization are that the invoked method is not allowed to store references to the argument after the method completes, and heap operations such as `realloc` and `free` are not allowed. Should the method require such functionality, it is trivial to use the copy constructor to get a usable copy without such constraints.

To avoid always using `define_type` when a given type does not require packing,

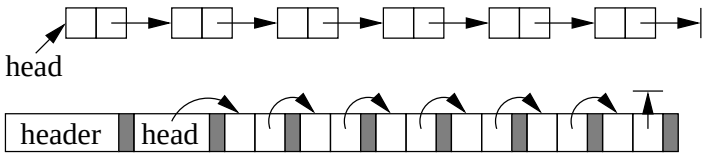

Fig. 6. Example of the standard and packed representations of a linked-list

we employ the *traits* technique. Traits allow C++ types to be associated with characteristics by using templates, and are used in STL as `iterator_traits` [22]. In ARMI's case, each type is associated with several variables storing its space requirements and whether it requires packing. The first time an object of a given type is transferred as an argument, `define_type` is used to initialize its associated traits. Unless the type requires packing, subsequent transfers will obtain all necessary information from the type's traits, bypassing `define_type`.

The `define_type` interface does encounter several surprises. For instance, when packing a dynamic member, its entire contents are copied. If that member contains other dynamic members, they also will be copied, assuming they are not null. Although this protocol is correct, it could lead to poor performance in certain situations. Consider a linked list, such as in Figure 6, where each node is composed of a local data element, and a dynamic pointer to the next node. When transferring such a linked list node as an argument, `head` for example, `define_type` will pack the rest of the list following the node, even if only the single node was really necessary. For large lists, the extra data transferred will be detrimental to performance. The user-directed solution is to require users to remove the desired nodes from the list, such that their next pointer is null before transfer, then reset the next pointer after transfer. Other solutions require extensions to the interface, such that the number of dynamic links to follow can be specified.

Pointer-based objects that contain multiple pointers to the same sub-object are also problematic. When such an object is packed, multiple copies of the aliased sub-object will be made, which is also the approach taken by Charm++'s PUP interface. To prevent making multiple copies, as required by some applications, each dynamic member's address could be entered into a hash table during the packing phase. If the given address already exists, its previous offset can be used, instead of making a copy and assigning a new offset. As such, this solution will correctly handle pointer-based objects, at the expense of hashing each dynamic member.

Virtual inheritance is difficult to support because compilers often implement it using internal relative offsets, which the user is not able to pass along to `define_type`. Unless identified and handled like other offset types, the resulting transferred object will be invalid. If compiler support is available, the internal offsets are easily identified. Since such support is not currently available, the more complicated alternative is to augment `define_type`. Since `define_type` identifies all user-defined data within the object, a more complicated protocol is able to look for "holes" in the defined data, and check to see if they contain a relative pointer. This pointer may then be handled as a regular offset pointer.

E. Transfer

The transfer layer is fundamental to ARMI, and is the layer that interacts directly with the underlying communication libraries. This section gives an overview of the mechanisms used by the Threads and MPI implementations, then discusses them specifically as applied to each of the communication functions.

The Threads implementation actually utilizes a form of message passing within shared-memory, since the semantics of RMI imply one thread telling another thread

to do something. This message passing is simpler than standard MPI however, and has several opportunities for higher performance. Each thread owns a request queue, which holds RMI requests from other threads. Instead of copying the entire request from the origin to the destination, as in MPI, only the request pointer is enqueued. Since the queue can become a bottleneck during heavy periods of communication, requests are pipelined by using multiple aggregation buffers per possible destination thread. While one or more buffers are enqueued at the destination, a different buffer can be filled. Each buffer has a separate flag to check for completion, so they can be reclaimed and reused as soon as possible. To prevent blocking, new buffers are allocated and used should the pre-allocated buffers be exhausted. The alternative to allocating additional buffers is to attempt to schedule other incoming requests while waiting for the existing buffers to become available again, which could cause starvation or deadlock, as described in the next section.

The MPI implementation is similar to Threads, although MPI implements request queuing internally. Each thread applies the same pipelined sending scheme, and uses non-blocking sends (`MPI_Isend`) to facilitate filling one buffer while the other is sending. Since the implementation is not multi-threaded, only one incoming request can be processed at a time. A single non-blocking receive (`MPI_Irecv`) is posted and then the computation is started, which allows good implementations of MPI to overlap the communication with computation. Polls simply check to see if the receive completed (`MPI_Test`), in which case the request is processed. If the request blocks for any reason (e.g., by invoking a `sync_rmi`), a new receive can be started to process other incoming requests until the blocking is complete (e.g., the return value is received). Because MPI requires copying incoming messages into a user-defined buffer, a single buffer large enough to hold all possible communications is allocated *a priori*. This static allocation scheme prevents costly dynamic allocation, at the expense of

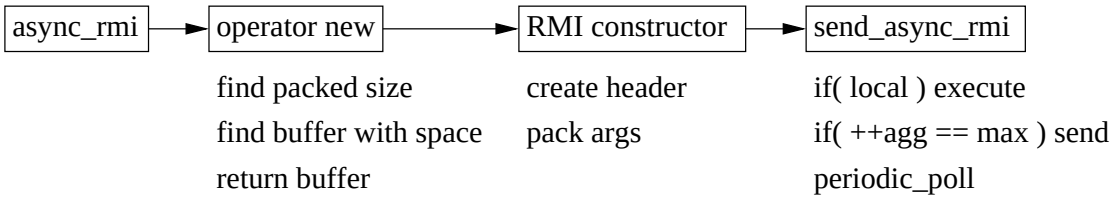

Fig. 7. Implementation of `async_rmi`

greater memory overhead.

Figure 7 shows the basics of `async_rmi` within this framework. Sending many small messages via message passing causes network traffic and consumes bandwidth. As such, `async_rmi` requests are automatically aggregated by the sending thread, and issued in groups based on a default setting that is tuned for each machine at installation, or a user-defined setting. Although aggregation does not change the amount of user data transferred, it does reduce the overhead of library calls and message header transfer associated with many small messages. Requests are allocated directly into an aggregation buffer until the aggregation factor is reached, by overloading `operator new`. As described in Chapter IV, Section D, `define_type` is used to determine the required size. The buffer for the specified destination is then obtained from the pool of outgoing buffers. If the first buffer obtained does not contain enough remaining space, it is immediately transferred, and a new buffer is obtained. In either case, the next available entry with space is returned.

The RMI request constructor then creates the RMI header in the obtained buffer space, and packs the arguments as necessary using each arguments' `define_type`. Next, `send_async_rmi` considers the request. If the destination is the local thread, the request is executed immediately. Otherwise the aggregation settings are considered, and if satisfied, the buffer is transferred. Finally, a `periodic_poll` is performed as part of the scheduling layer, which only performs a poll every n invocations.

The `sync_rmi` operation builds upon the foundation of `async_rmi`. Requests are sent in the same fashion, except the sender blocks after transfer until the return value is returned. The block operation polls continuously, enabling quick response to incoming return values. The Threads implementation uses a separate response stack for the computing thread to push the return value. A stack is used because `sync_rmi`'s could invoke methods that invoke additional `sync_rmi`'s, possibly forming cycles. The MPI implementation uses the exact same transfer facilities as `async_rmi`, and differentiates between incoming requests and return values by the message tag, contained in the message's header.

The collective operations, `broadcast_rmi` and `reduce_rmi`, take a somewhat different approach. Since MPI provides specific functions implementing the broadcast (`MPI_Bcast`) and reduction (`MPI_Reduce`) communication patterns, the MPI implementation invokes these functions directly, bypassing the previously discussed transfer layer functionality. The Threads implementation is somewhat more involved, since it must implement these patterns explicitly. The broadcast is implemented using a shared global variable storing the input argument, which all threads read before performing their local execution of the RMI. The reduction is implemented as a binary tree, with each parent combining the contribution of its two children.

F. Scheduling

RMI requests do not require matching operations on the destination thread. As such, ARMI must introduce mechanisms to schedule the processing of incoming requests. The two issues that must be balanced are ensuring a timely response to incoming requests, which may be blocking the caller (e.g., `sync_rmi`), and allowing the local computation to proceed. This is not a new problem, and we are aware of four

solutions:

1. *explicit polling* - the code explicitly polls for incoming requests [16, 19, 14, 9]. This approach is successful if polls do not dominate the local computation, but are frequent enough to yield a timely response.

2. *interrupt-driven* - the caller issues an interrupt to notify a thread of incoming requests [16, 19, 14, 9, 8]. Although this solution is often avoided due to the high cost of interrupts, it does guarantee a timely response with minimal interruption of local computation (i.e., no extraneous polls).

3. *blocking communication thread* - a separate communication thread posts a blocking receive for incoming requests [14]. Upon arrival, a request is immediately processed. This solution is successful if other threads can execute while the receive is blocking, and control returns to the communication thread soon after the receive completes.

4. *non-blocking communication thread* - a separate communication thread performs a poll for incoming requests, processes any available requests, then yields [14]. This solution is successful if the thread scheduler is effective. For example, the communication thread is scheduled at times when no computation is available, or the time slice is a good balance between computation and polling. Since typical time slices are 1/10 of a second, this is often a problem.

Our current solution is explicit polling. Polls are inserted within communication and synchronization layer calls. This has the advantage of being transparent to the user, and the drawback of poor response if no communication occurs for a long period of time. In cases where the user is aware of this, an explicit `rmi_poll` operation is available to force a poll to occur. To handle the alternative case of extremely frequent

communication, such that excessive polling would slow the computation, every nth communication call will internally perform a poll, where n may be set by the user. Low values for n will yield more timely responses, but slow the progress of the computation by imposing more unsuccessful polls.

An additional advantage of polling within communication calls is a well defined model of when incoming requests may be scheduled. Although communication threads have the potential to improve the response time of incoming requests, they must be carefully synchronized to ensure the mutual exclusion requirements of ARMI. Specifically, local operations and remotely invoked requests must both execute atomically. This is straightforward to enforce with one thread that only schedules within ARMI calls. However, with multiple threads, one thread may be executing a section of user code while another thread wants to schedule an incoming request that happens to use that same code, thus violating mutual exclusion. In addition, the exact semantics of thread support and its interaction with the operating system are often platform dependent (e.g., thread scheduling policies).

In general, when a group of RMI requests are received, as in an aggregated group of `async_rmi's`, requests are processed in FIFO order. There is one notable complication to request scheduling and execution, however, which is the possibility of nested RMI requests. This can occur when a remotely invoked method blocks inside a communication call. If another RMI is scheduled to execute while the original RMI blocks, the new RMI is known as a nested RMI request. For example, a lookup operation may be implemented using a `sync_rmi` to the thread that last owned an object. If the object moved, that thread may invoke another `sync_rmi` to a different thread. Since `sync_rmi` blocks until it obtains a return value, it is highly likely that nested RMI's will occur in this scenario.

Nested requests present a number of problems. First, the original RMI's resources must be saved while the nested RMI consumes new resources. A naive implementation could allow enough resources to be consumed that deadlock is possible, because a request necessary for progress does not have enough resources to complete. Second, nested execution can starve the original RMI request, potentially causing imbalance in the system. ARMI addresses the first issue by allocating a set of initial resources, then dynamically allocating additional resources as necessary. ARMI address the second issue by allowing the user to specify the maximum nesting level.

G. Mixed-Mode

Our mixed-mode implementation implements multi-protocol communication. It starts by creating MPI processes, which each spawn a number of local threads. As described in Chapter III, Section G, the aggregate threads provide a flat view of the parallelism, instead of exposing the nested or hierarchical parallelism. This decision allows the mixed-mode implementation to increase performance by using shared-memory communication to local threads (i.e., within the MPI process), and MPI to remote threads, without requiring mixed-mode specific hooks in the ARMI interface. However, sufficient information is available to each thread to determine the number of address spaces (i.e., MPI processes) and local threads in each should hierarchical hooks be desired in the future.

Each thread maintains a table, which maps destination thread id's to MPI processes and local thread id's. A single index into this table allows the thread to determine whether the communication is local or remote to its MPI process. In the simple case, mixed-mode RMI is implemented by adding a check in each RMI communication operation, and then calling the appropriate underlying implementation,

Threads or MPI.

Since the view of parallelism is flat, threads can individually send and receive RMI requests to and from remote MPI processes. The MPI message tag is used to identify which thread should actually receive a given request. Collective operations such as `rmi_fence` are modified to a two level hierarchical scheme, synchronizing the local shared-memory threads first, with only the root thread continuing to perform the MPI synchronization, while the other threads poll. Polling operations are modified to check both MPI and shared-memory for incoming requests.

Many MPI implementations are not thread-safe, and hence require running in serialized mode when multiple threads within the same MPI process make MPI calls. In this case, each MPI operation must be protected by a lock, potentially limiting the available concurrency. Some vendor-supplied MPI implementations are thread-safe, and do support parallel MPI operations. However, we have found that in many cases, the resulting performance is not significantly better than running in serialized mode.

This limited support for threads with MPI requires several subtle modifications to the basic ARMI implementations to improve performance. For instance, `rmi_wait` must be able to wait for either an incoming shared-memory or message passing request. Whereas blocking operations, such as `MPI_Wait`, were originally used to implement `rmi_wait`, they must be replaced with a loop that balances a non-blocking `MPI_Test` with a shared-memory poll. Such loops can cause great contention by repeatedly polling the MPI library. To alleviate the contention, ARMI only tests the MPI library every nth iteration of the loop, where n can be adjusted to a given platform based on the cost of `MPI_Test`. ARMI also tries to adapt n to a given algorithm at run-time by decreasing or increasing n's value each time `MPI_Test` is successful or unsuccessful.

Regardless, without careful tuning of implementation parameters such as n, the

resulting performance is often not an improvement versus pure MPI. However, we believe that off-loading all MPI communication to a separate communication thread will allow for increased performance. Such an implementation can be viewed hierarchically, with the communication thread at the top-level handling all MPI communication, and the lower-level worker threads using shared-memory between themselves and the communication thread. Since only the communication thread will be using the MPI library, non-thread-safe implementations can be used, and locks are unnecessary. This will eliminate contention on the MPI library, as well as complicated tuning parameters such as n. However, other issues do arise, such as whether to dedicate a processor to the communication thread, or to multi-task it with the work threads.

CHAPTER V

PERFORMANCE

We tested the various ARMI implementations on a number of different machines, including a Hewlett Packard V2200, an SGI Origin 3800, an IBM Regatta-HPC, and an IBM RS6000 SP. The *V2200* is a shared-memory, crossbar-based interconnect, consisting of 16 200MHz PA-8200 processors with 2MB L2 caches. The *O3800* is a hardware DSM, hypercube-based CC-NUMA (cache coherent non-uniform memory access) consisting of 48 500MHz MIPS R14000 processors with 8MB L2 caches. The *Regatta* is a shared-memory bus-based interconnect, consisting of 16 1.3GHz Power4 processors with 1.5MB L2 and 32MB L3 caches. The *RS6000* is a cluster of SMP's, consisting of 4 332MHz PowerPC 604e processors with 256kB L2 caches per node, with nodes connected by a dedicated high-speed switch.

A. RMI Overhead

The ARMI abstraction includes a number of overheads compared to regular method invocation. Section B details transfer latency, while this section measures the cost of creating and executing an RMI locally (i.e., everything but transfer).

The major abstraction involved in building an RMI request is using the member function pointer (method pointer), instead of invoking the method directly on a given object. Storing the method pointer allows for execution at a later time, at the cost of increased overhead due to additional memory dereferences at runtime.

Table II measures the cost of invoking an empty method directly, via a method pointer, and via a local ARMI `async_rmi`. The inliner was disabled since inlining an empty method allows the optimizer to deadcode and remove the entire invocation. In general, the method pointer generally requires twice as long as direct method

Table II. Overhead of method invocation (ns)

	V2200	O3800	Regatta	RS6000
Direct	65	12	8	60
Method Pointer	132	26	14	121
ARMI	933	325	197	842

invocation. ARMI requires a substantial amount more than this however.

Before execution is possible, the RMI request must be created, at the cost of several internal method invocations that allocate the request directly in the aggregation buffer. This helps reduce latency in the general case, but increases the cost of local execution versus creation directly on the stack. To preserve copy-by-value semantics, and possibly serialize data for transfer, all arguments must also be copied. During execution, the RMI request execution method is virtual, which incurs an additional dereference, and must also access the RMI registry to determine the location of the object specified by the given `rmiHandle`, another dereference. Although these overheads are costly compared to direct method invocation, the next section will show that they are a small percentage of the actual communication latency.

B. RMI Latency

We compared the latency of ARMI versus explicit Pthreads or MPI code using a ping-pong benchmark. One thread sends a message, and upon receipt, the receiver immediately sends a reply. ARMI uses two benchmarks. The first uses `async_rmi` to invoke a reply `async_rmi`. The second uses a single `sync_rmi`. The Pthreads benchmark uses an atomic shared variable update as the message, with ordering preserved by busy-waiting on volatile variables. The MPI benchmark explicitly matches `MPI_Send`s and `MPI_Recv`'s.

Table III. Latency of communication (us)

	V2200		O3800		Regatta		RS6000	
	Explicit	ARMI	Explicit	ARMI	Explicit	ARMI	Explicit	ARMI
Threads	15	21/18	4	6/5	2	3/3	6	16/11
MPI	16	45/49	13	15/16	6	10/11	29	66/71

ARMI results reported as `async_rmi`/`sync_rmi`

The resulting wall clock times are shown in Table III. Since ARMI is an abstraction implemented using Threads or MPI, its latency is always greater. However, the abstraction buys the user the ability to interact with an object's methods, instead of directly with data, and handles most of the low level details. The major contributor to this overhead is that ARMI attempts to make the common case fast, whereas the hand-coded benchmarks make the best possible use of their communication libraries for this specific benchmark. For example, ARMI uses non-blocking `MPI_Isend` and `MPI_Irecv` to overlap communication and computation as much as possible, which yields great benefits in larger programs. However, given the minimal amount of overlap in the ping-pong benchmark, the hand-coded MPI uses `MPI_Send` and `MPI_Recv`. We have identified that the HP implementation of MPI on the V2200 incurs a 131% penalty when using `MPI_Isend`/`MPI_Irecv` versus `MPI_Send`/`MPI_Recv` on this benchmark, increasing the latency from 16 to 37us for hand-coded MPI. Another source of overhead is that ARMI transfers RMI header information during the ping-pong, and hand-coded MPI is able to use empty messages. On the V2200, augmenting the MPI benchmark to transfer 24 bytes, the size of an RMI header, increases the latency from 37 to 45us. As such, 72% of the ARMI overhead on the V2200 can be attributed to implementation via non-blocking MPI, 27% to header information, with remaining differences due to the overhead of creating and executing RMI requests.

We also tested the impact of aggregation on message latency when issuing many

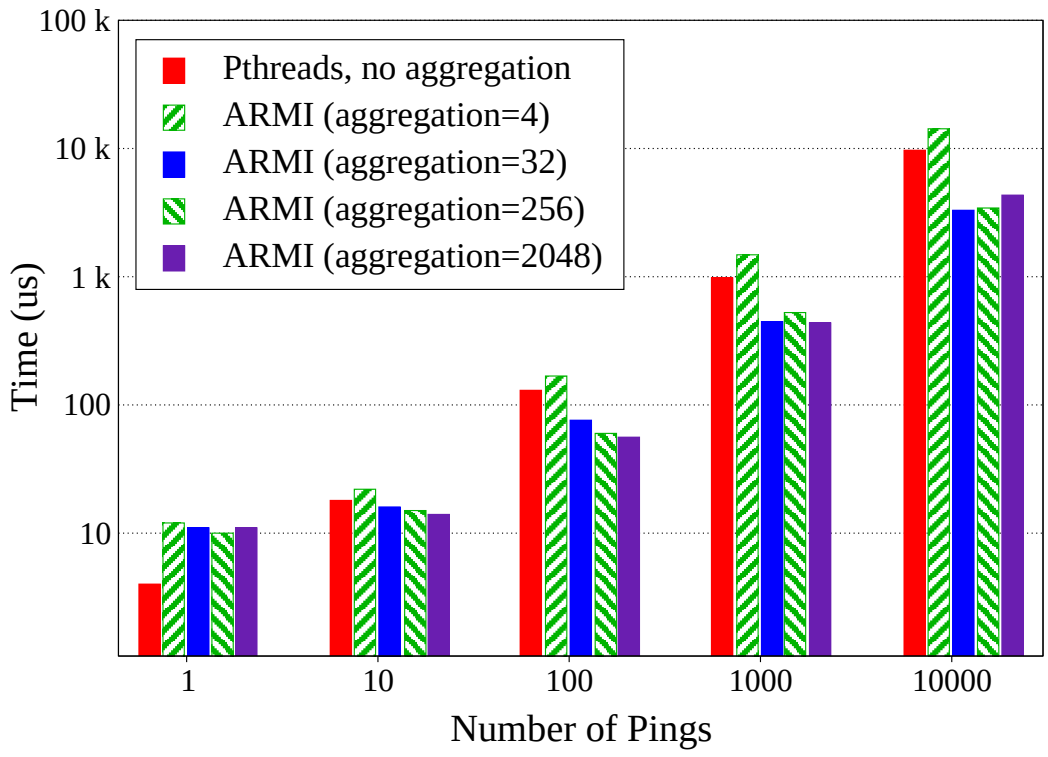

Fig. 8. Shared-memory latency (O3800)

communication requests, by re-timing the ping-pong benchmark using multiple consecutive pings before a single pong. ARMI's aggregation factor was varied from 4 to 2048 messages.

Figures 8 and 9 show the results for Threads and MPI on the O3800. For Threads, ARMI is faster after 100 pings, yielding a 3-fold improvement with an optimal aggregation buffer of 256 messages (8KB). For MPI, ARMI is significantly faster after just 10 messages, yielding a 15-fold improvement for 10,000 pings, also with an optimal aggregation buffer of 256 messages (8KB). The general trend for aggregation factor is a parabolic curve. Initially, aggregation alleviates much of the network traffic, therefore improving bandwidth. If used too liberally however, aggregation increases the amount of work that needs to be performed at the end of the computation phase,

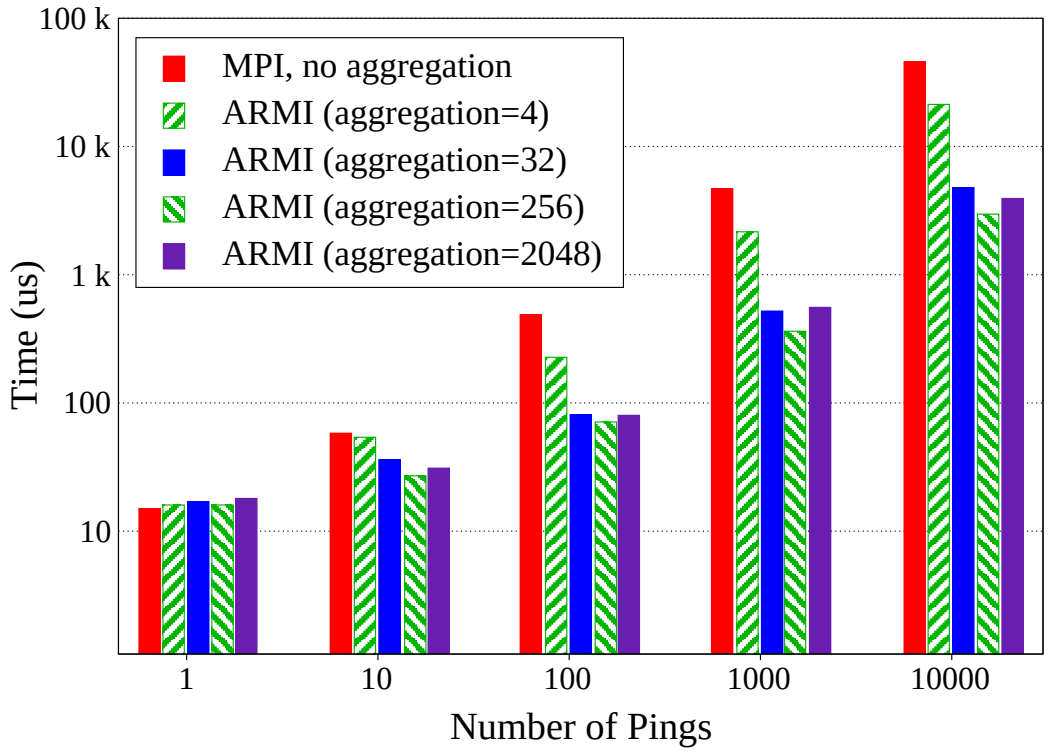

Fig. 9. Message passing latency (O3800)

thus increasing the critical path.

C. Fence Latency

We compared the overhead of `rmi_fence` versus vendor-optimized barriers. For shared-memory, we compared against the OpenMP barrier directive when available (O3800). Unfortunately, several of the test systems did not support OpenMP at the time of this writing, and thus only the ARMI results are reported. For message passing, we always compared against `MPI_Barrier`.

Figure 10 shows the results for shared-memory. Since ARMI uses a platform independent fence, which continues to poll for RMI requests while waiting for termination, we expect it to incur some overhead. Although typically much less, we

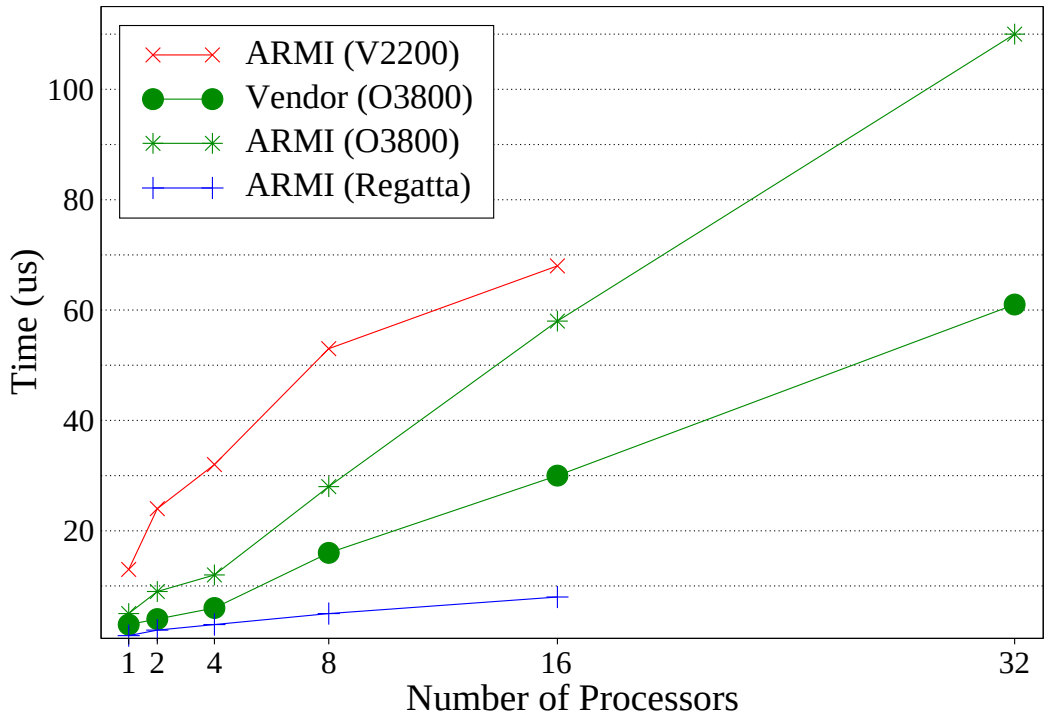

Fig. 10. Shared-memory barrier versus `rmi_fence`

have found the overhead of polling and termination detection to be as high as 23% for ARMI's Threads fence. We attribute the rest of the overhead to our platform independent implementation not taking advantage of specialized vendor features.

Figure 11 shows the results for message passing. ARMI scales competitively on the O3800 and Regatta, with only a slight increase in overhead, due to polling and termination detection. The nearly flat V2200 vendor curve implies HP's `MPI_Barrier` implementation uses shared-memory optimizations that are not available to ARMI. This result confirms the findings of [19], which demonstrate the utility of pollable, instead of just blocking, barriers for use in libraries such as ARMI to maximize performance.

Figure 12 shows the message passing and mixed-mode results for the RS6000 clus-

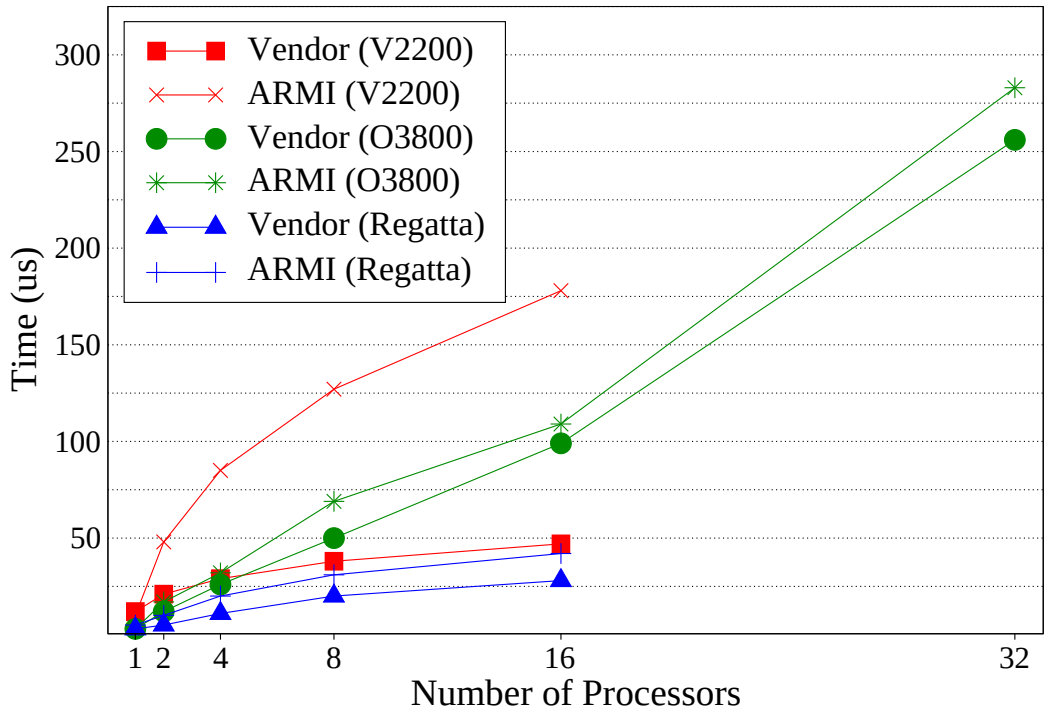

Fig. 11. Message passing barrier versus `rmi_fence`

ter. ARMI's MPI implementation is not optimized for a 4-way cluster, and hence pays penalties since more cross-node messages are being used than the vendor-optimized `MPI_Barrier`. The results on 1-4 processors also imply that the vendor barrier uses shared-memory internally, similar to the V2200. ARMI's mixed-mode implementation, which uses shared-memory synchronization within nodes, and MPI synchronization between nodes, scales much more competitively. Note that the mixed-mode implementation was tuned to poll the MPI layer very infrequently, and should be considered a best case scenario. More general applications would likely require higher settings to improve scheduling timeliness of incoming requests.

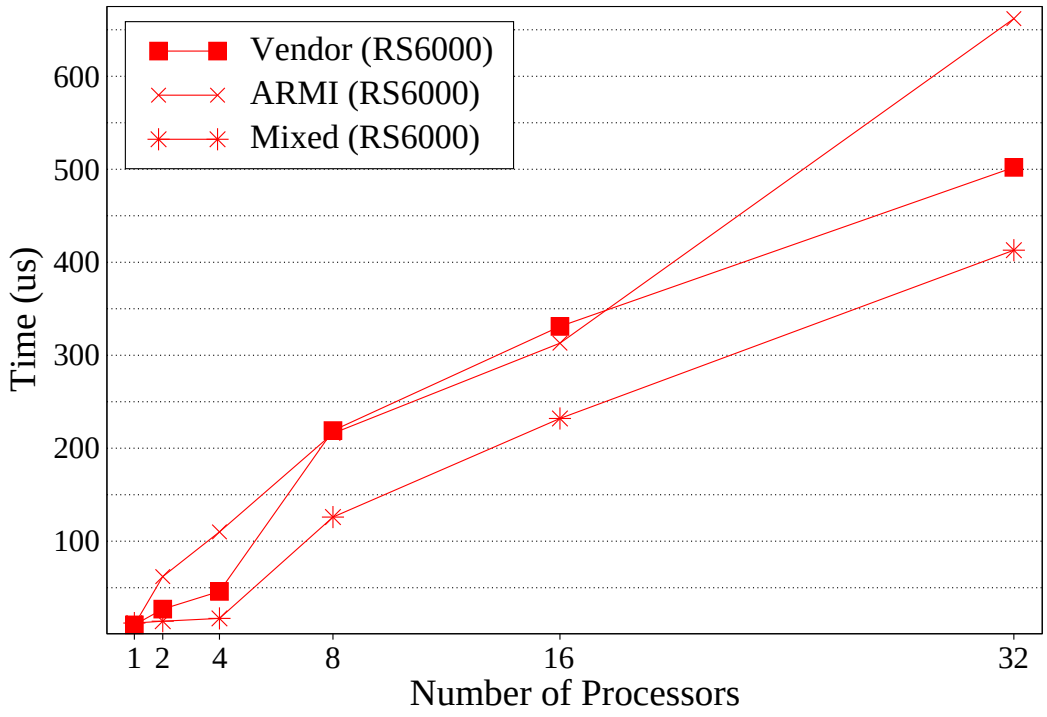

Fig. 12. Message passing barrier versus `rmi_fence` (RS6000)

D. Algorithm Performance

A variety of parallel algorithms have been implemented using ARMI. One example is the case study, sample sort. The ARMI implementation uses fine-grain RMI directly, issuing an RMI for each element in the distribution phase, to add it to the correct destination bucket. We compared our RMI-based implementation to a hand-tuned MPI program that required twice as many lines of code. The MPI program manually buffers all elements locally, then performs an all-to-all merge before the final sorting phase.

Figure 13 shows the scalability versus running on one processor on the Regatta. It compares ARMI's Threads and MPI implementations for one million integers, distributed uniformly among threads. Both are using an aggregation factor of 256

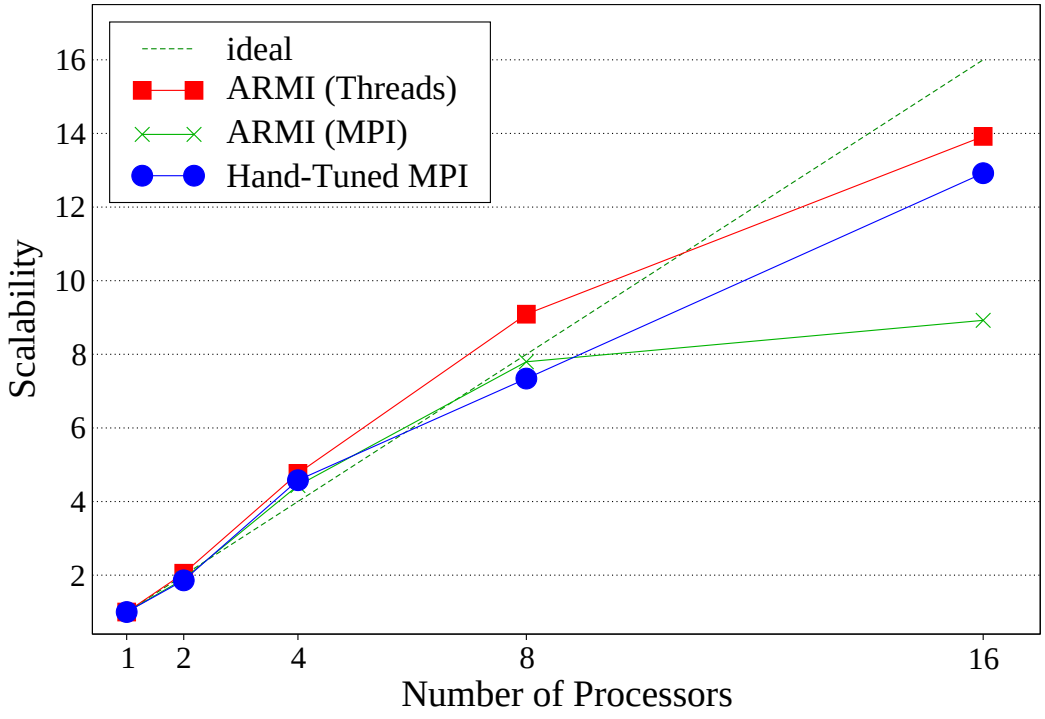

Fig. 13. Scalability of sorting 1M integers (Regatta)

requests. As shown, Threads are able to sustain more parallelism at this level of work than MPI, as well as outperform the hand-tuned MPI. The superlinear speedups starting at 4 processors occur when the input data first fit into the 1.5MB L2 cache. The drop-off at 16 processors is due to the overhead of communication, which sets a lower bound on the running-time. Since Threads have lower latency than MPI, they are able to sustain a faster running time. In addition, the Threads implementation overlaps communication and computation by communicating groups of RMI requests, whose sizes are determined by the aggregation factor, instead of using a single large merge, as in the hand-code MPI, which is unable to hide any communication latency.

Figure 14 shows the scalability as the dataset is increased from 1M to 50M integers. Even given the much larger dataset, the Threads implementation is still

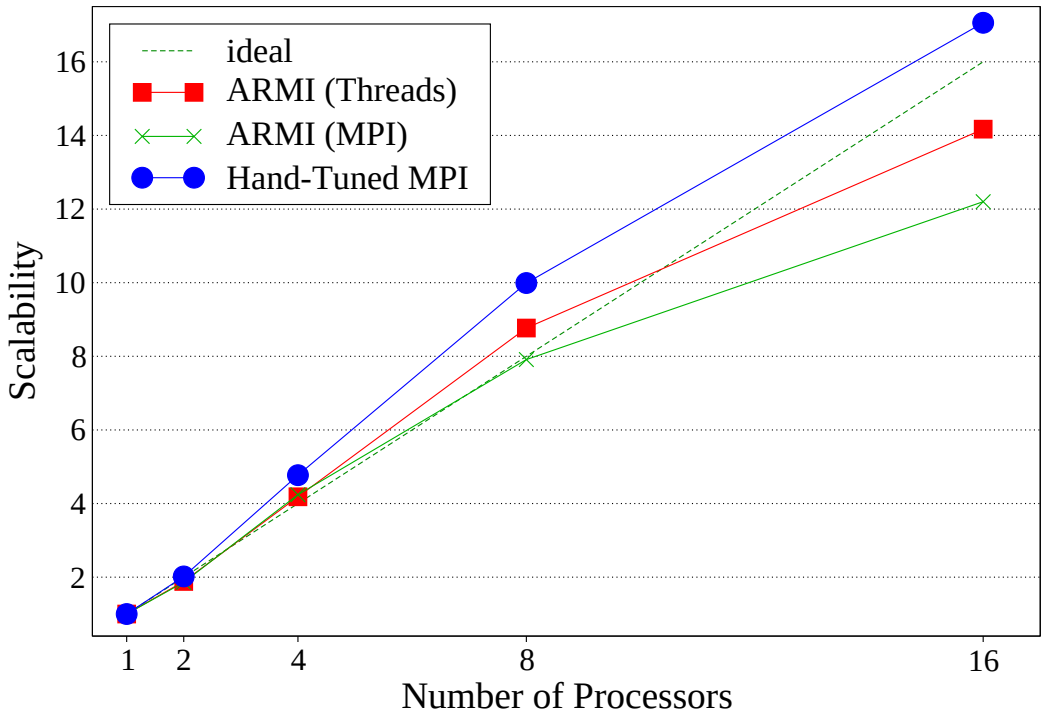

Fig. 14. Scalability of sorting 50M integers (Regatta)

able to outperform MPI. However, the increased work-per-communication ratio allows the hand-tuned MPI to able to outperform ARMI. The superlinear speedups at 8 processors occur when the data first fits into the 32MB L3 cache.

This benchmark demonstrates the value of having multiple implementations based on shared-memory and MPI. MPI can always be used for machines that only support message passing, and thus provide scalability for the largest systems. However, if a machine provides shared-memory, then the Threads implementation is able to leverage it for increased performance. As shown, smaller problems may still be run on a larger number of processors efficiently, compared to using MPI directly.

To demonstrate the effect of mixed-mode communication, we implemented a Jacobi iterative solver for Poisson's equation. Each thread owns a portion of the

matrix, where data is distributed using 1D block partitioning. The solver iterates a fixed number of times instead of testing for convergence. As opposed to sample sort's highly irregular all-to-all communication pattern, Jacobi performs neighbor communication. As such, when run on a cluster, only the outer two processors will need to use MPI, while all other processors can use shared-memory exclusively. Since each thread only invokes a single RMI on each neighbor per iteration, varying aggregation has no effect.

Figure 15 shows the scalability for the MPI implementation compared to the mixed-mode implementation on the RS6000, where the work per thread is always a 500x500 matrix over 200 iterations. Mixed-mode uses one MPI process per node, with four shared-memory threads inside. As shown, mixed-mode is able to gain a slight advantage by using the shared-memory. We note that the MPI implementation is running on top of IBM's MPI library, which is also optimized to take advantage of shared-memory within nodes. The mixed-mode implementation is still able to outperform the MPI by simply trading pointers, instead of copying data from the send buffer to the receive buffer as in MPI (not to mention any additional intermediate copies used).

Larger codes have also been implemented using ARMI. Detecting strongly connected components is an important component for many scientific codes, such as particle transport. One parallel approach is detailed in [29]. In short, it is an iterative algorithm that is composed of three main steps:

1. Sweep the graph, trimming edges that are not part of a strongly connected component.

2. Each processor chooses a pivot, and marks edges that are part of a strongly connected component.

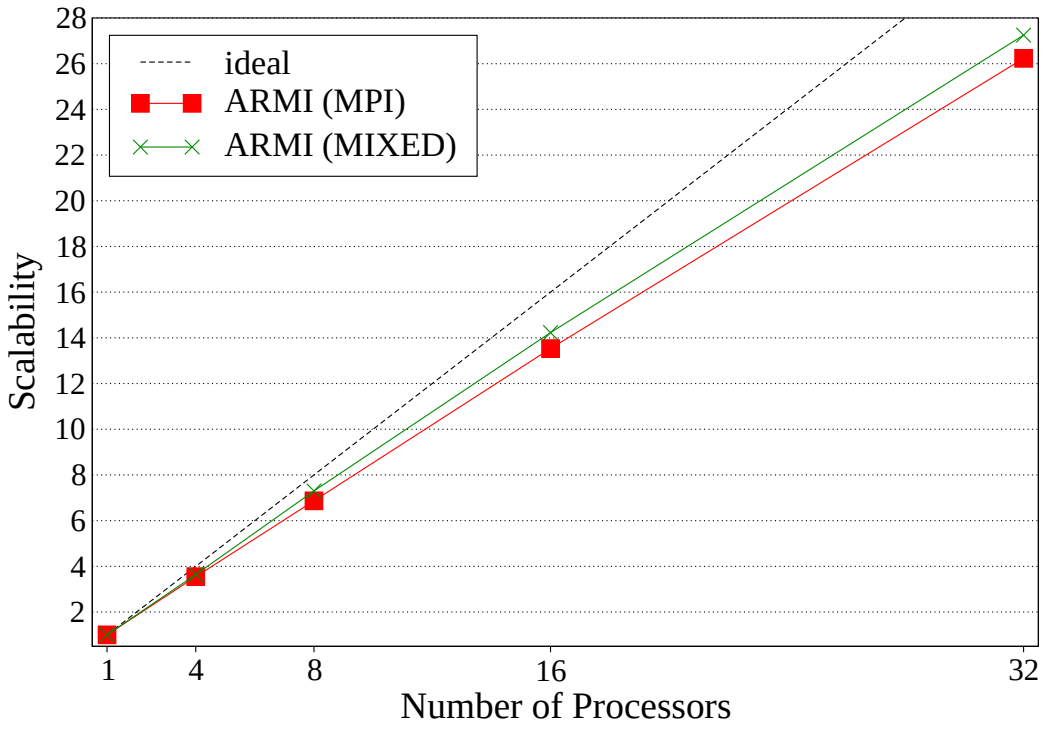

Fig. 15. Scalability of Jacobi iteration (RS6000)

3. Remove each strongly connected component, and iterate if there are remaining edges.

The communication is fine-grain, because RMI is used every time an edge is marked or trimmed. As such, the communication library is used to aggregate communication sufficiently for performance. The termination detection of `rmi_fence` is used in the trim and mark steps to substantially reduce the complexity of the code.

Figure 16 shows the scalability on the V2200 using an aggregation factor of 100. The input to the algorithm is a mesh consisting of 10k vertices, and 29114 edges, with a total of 338 strongly connected components. The superlinear speedups are possible because as processors are added, more pivots are considered, which greatly decrease the number of iterations necessary for convergence. For example, the number

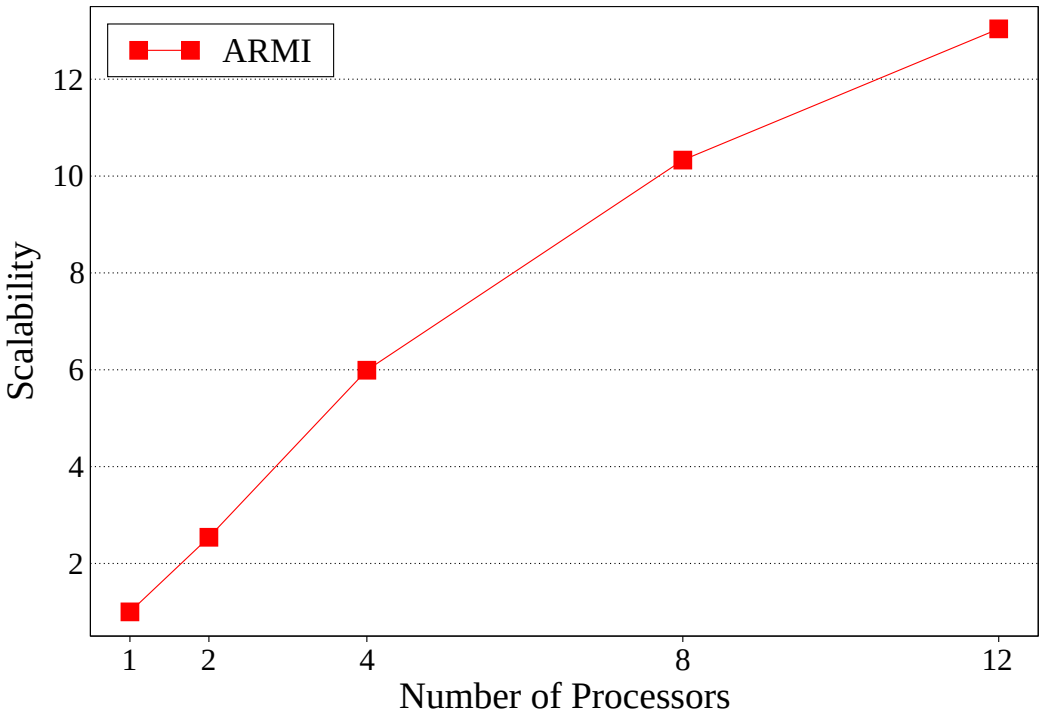

Fig. 16. Scalability of detecting strongly connected components (V2200)

of iterations required are cut in half when moving from one to two processors for this graph.

CHAPTER VI

RELATED WORK

Several other systems provide RMI-based high performance parallel communication protocols with similar goals to ARMI.

Active messages is an extension to one-sided communication that includes specifying a message handler on the receiving process [16]. The handler is intended to quickly integrate the message into the ongoing computation, as opposed to general purpose computation, as in RPC. We note that their approach of polling the less expensive shared-memory layer more often than the message passing network is similar to our mixed-mode implementation [30].

Tulip is a wrapper around existing models, and serves as a compiler target for the pC++ programming language [19]. It provides a consistent interface across a variety of platforms, and provides functionality common to message passing, one-sided communication, and remote procedure call.

ARMCI is a one-sided communication library that focuses on optimizing strided data communication [8]. Specifically, if several separate sections of an array need to be communicated, this information is passed into a single ARMCI call, whereas traditional libraries would require multiple calls. This additional information allows for increased performance by internally buffering and issuing fewer messages.

Nexus provides remote service requests (RSR), which are similar to non-blocking RPC, and can optionally spawn a new thread on the destination to perform the work [14]. Such threads allow for the dynamic creation and balancing of work.

Charm++ is an object-oriented parallel programming language that utilizes non-blocking RMI for communication [17, 18]. It emphasizes split-phase execution and the creation of a large number of parallel tasks, which it dynamically schedules and

load balances, to increase latency tolerance.

In contrast to all of these systems, ARMI includes both blocking and non-blocking communication. Similar to ARMCI, automatic aggregation buffering is available for the non-blocking requests, although ARMI is more expressive in that it will aggregate multiple discrete calls, instead of just within a single call. Although ARMI is implemented on top of existing models, such as MPI and OpenMP, it is not simply a wrapper. It attempts to raise the level of abstraction by handling low-level issues internally, and providing an object-oriented RMI interface. Unlike Charm++, ARMI is currently focused on supporting global objects with a single registered representative per thread.

Many other researchers have considered combining shared-memory and message passing into mixed-mode, often also called hybrid and multi-protocol, parallel programs. Most of these studies focus on a two-level hierarchical scheme, by explicitly using MPI for coarse-grain parallelism, and then instrumenting a smaller number of inner loops for fine-grain parallelism with OpenMP [31, 32, 33]. This programming style is most often referred to as hybrid parallelism, and is a subset of the more general nested parallelism style. It has the advantage of mapping naturally to large clusters of SMP's. In addition, improved load-balancing is often cited as an improvement, since it is easier and less costly to dynamically schedule work in shared-memory. The drawback is that two different models of parallelism must be learned and explicitly programmed to achieve high performance.

Other systems, including some MPI implementations [34], provide multi-protocol communication, which provides a flat, or one-level, view of the available processors, and internally uses shared-memory when possible and remote memory operations otherwise [30, 35]. This programming style has the advantage of increased performance while explicitly using only one model of parallelism.

One novel example is Nexus, which implements a multi-protocol layer that adaptively selects the best transport protocol [36]. Available transport methods are maintained in a table at each point in the system, and may be dynamically selected based on a variety of schemes. Although the obvious approach is to always use the fastest protocol, other criteria, such as quality of service and security, are also considered.

ARMI focuses on multi-protocol communication, and specifically to tuning such communication over a wide variety of machines. We note, however, that it is a relatively straightforward extension to implement a hybrid model on top of a multi-protocol model. Such an extension provides increased flexibility, since most hybrid models assume no message passing calls occur within the fine-grain regions. In addition, unlike other existing systems, the current implementation of ARMI translates directly into MPI, OpenMP/Pthreads, or a combination, leveraging their already highly tuned, vendor provided facilities.

CHAPTER VII

CONCLUSIONS

This thesis presented ARMI, a high level parallel communication library based on remote method invocation, which takes advantage of the natural style of communication involved in object-oriented programs. ARMI was developed in support of the STAPL parallel C++ library, but may also be used to parallelize any C++ code. It provides a simple interface of RMI-oriented functions selected for performance and ease of use.

ARMI allows the user to exploit fine-grain parallelism, while handling low-level details such as scheduling incoming communication and aggregating outgoing communication. These details can be tuned for specific machines to provide the maximum performance possible without modification to user code. A fine-grain parallelization allows ARMI to fully exploit resources on a shared-memory system, while coarsening communication via aggregation for a large message passing system.

ARMI is currently implemented using shared-memory Threads and message passing MPI. In addition, it provides a mixed-mode implementation that uses shared-memory communication when possible, and MPI otherwise, to exploit clusters of SMP's via a flat view of parallelism. Performance experiments show that ARMI is competitive with hand-coded parallel programs. A variety of ARMI codes obtain scalable performance, including sorting, Jacobi iteration, and detecting strongly connected components, on four different systems, including an HP-V2200, Origin 3800, IBM Regatta and IBM RS6000 SP.

CHAPTER VIII

RECOMMENDATIONS FOR FUTURE WORK

The current ARMI implementations provide a stable infrastructure for future research. This section outlines the specific issues encountered during its implementation where additional work could improve functionality or performance. References are given to the section that describes each issue in more depth.

As discussed in Chapter III, Section C, ARMI currently only implements two collective communication patterns, a broadcast and a reduction. Since both operations are globally collective and require matching operations by all threads, a number of alternatives are described, including expressing subsets of threads. Section C discussed `rmi_fence`, which is also globally collective and could benefit from thread subsets as well.

As outlined in Chapter III, Section F, ARMI currently offers no support for global variables, although several alternatives or extensions for support are possible.

Chapter III, Section G, describes simple extensions to ARMI to support nested parallelism, in addition to the already supported multi-protocol parallelism.

ARMI currently assumes an SPMD style of execution when registering objects. However, Chapter IV, Section A, provides a number of different alternatives that relax this constraint.

As described in Chapter IV, Section D, the `define_type` interface provides substantial support for a variety of different class types. Additional work will allow for classes that use virtual inheritance, and provide for more flexible support for linked-lists and other pointer-based structures.

ARMI currently schedules incoming RMI requests using explicit polling within ARMI calls (Chapter IV, Section F). Three other alternatives are also possible, with

multi-threading, and especially the blocking communication thread, being the most promising.

As outlined in Chapter IV, Sections E and F, incoming RMI requests are generally processed in FIFO order. In addition, the implementations fully receive and process an incoming group of requests before considering the next group. Although this is correct, it tends to move buffering to the sender, as in a rendezvous protocol, instead of to the receiver, as in an eager protocol. If multiple requests are able to be queued by the receiver before execution, it is possible to implement more interesting methods for scheduling (e.g., based on request priority). The Threads implementation provides support for this immediately through its queuing-system. The MPI implementation could use a separate communication thread that continuously receives and buffers incoming requests. The work thread could then consider which order to actually execute the buffered requests.

RMI header information is currently packaged with every single RMI request, potentially consuming a large portion of the aggregation buffer. An alternative is to group the incoming requests based on method (e.g., using hashing), then to send a single header for the entire homogeneous group.

As described in Chapter IV, Section G, current MPI implementations do not provide adequate support for threaded applications. Specifically, many implementations are not thread-safe, and hence MPI calls must be placed in a critical section. As such, substantial performance improvements are possible by implementing a separate communication thread to handle all MPI communication.

REFERENCES

[1] IEEE, *Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application: Program Interface [C Language]*, 9945-1:1996 (ISO/IEC) [IEEE/ANSI Std 1003.1 1996 Edition], Piscataway, NJ: IEEE Standard Press, 1996.

[2] OpenMP Architecture Review Board, *OpenMP - C and C++ Application Program Interface*, October 1998, Document DN 004-2229-001, www.openmp.org.

[3] Dimitrios Nikolopoulos, Eduard Ayguad, Jess Labarta, Theodore Papatheodorou, and Constantine Polychronopoulos, "The tradeoff between implicit and explicit data distribution in shared-memory programming paradigms," in *Int. Conf. on Supercomputing*, 2001, pp. 23–37.

[4] Cristiana Amza, Alan Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *IEEE Computer*, vol. 29, no. 2, pp. 18–28, 1996.

[5] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, June 1995, www.mpi-forum.org.

[6] L.G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[7] Hongzhang Shan and Jaswinder Pal Singh, "A comparison of MPI, SHMEM and cache-coherent shared address space programming models on the SGI origin2000," in *Int. Conf. on Supercomputing*, 1999, pp. 241–266.

[8] Jarek Nieplocha and Bryan Carpenter, "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems," in *Workshop on Runtime Systems for Parallel Programming of the Int. Parallel Processing Symp.*, 1999, [CDROM].

[9] Gautam Shah, Jarek Nieplocha, Jamshed Mirza, Chulho Kim, Robert Harrison, Rama Govindaraju, Kevin Gildea, Paul DiNicola, and Carl Bender, "Performance and experience with LAPI: A new high-performance communication library for the IBM RS/6000 SP," in *Int. Parallel Processing Symp.*, 1998, pp. 260–266.

[10] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, May 1998, www.mpi-forum.org.

[11] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha, *Java(TM) Language Specification (2nd Edition)*, Reading, MA: Addison-Wesley Pub Co, 2000.

[12] Jim Waldo, "Remote procedure calls and java remote method invocation," *IEEE Concurrency*, vol. 6, no. 3, pp. 5–7, 1998.

[13] Sun Microsystems, "Java remote method invocation (RMI)," http://java.sun.com/products/jdk/rmi/, 1995–2002.

[14] Ian Foster, Carl Kesselman, and Steven Tuecke, "The Nexus approach to integrating multithreading and communication," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 70–82, 1996.

[15] Madhusudhan Govindaraju, Aleksander Slominski, Venkatesh Choppella, Randall Bramley, and Dennis Gannon, "Requirements for and evaluation of RMI

protocols for scientific computing," in *High Performance Networking and Computing Conf. (Supercomputing)*, 2000, pp. 76–102.

[16] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser, "Active messages: A mechanism for integrated communication and computation," in *Int. Symp. on Computer Architecture*, 1992, pp. 256–266.

[17] Laxmikant Kale and Sanjeev Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Conf. on Object-Oriented Programming Systems, Languages and Applications*, 1993, pp. 91–108.

[18] Laxmikant Kale and Sanjeev Krishnan, "Charm++: Parallel programming with message-driven objects," in *Parallel Programming using C++*, Gregory Wilson and Paul Lu, Eds., pp. 175–213. Cambridge, MA: MIT Press, 1996.

[19] Peter Beckman and Dennis Gannon, "Tulip: A portable run-time system for object-parallel systems," in *Int. Parallel Processing Symp.*, 1996, pp. 532–536.

[20] Lawrence Rauchwerger, Francisco Arzu, and Koji Ouchi, "Standard templates adaptive parallel library," in *Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1998, pp. 402–409.

[21] An Ping, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger, "STAPL: An adaptive, generic parallel C++ library," in *Int. Workshop on Languages and Compilers for Parallel Computing*, 2001, [CDROM].

[22] Bjarne Stroustrup, *The C++ Programming Language*, Reading, MA: Addison-Wesley Pub Co, 2000.

[23] Guy Blelloch, Charles Leiserson, Bruce Maggs, Greg Plaxton, Stephen Smith, and Marco Zagha, "A comparison of sorting algorithms for the connection machine CM-2," in *Symp. on Parallel Algorithms and Architectures*, 1991, pp. 3–16.

[24] David Culler and Jaswinder Pal Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, San Francisco, CA: Morgan Freeman Publishers, Inc., 1999.

[25] Hong Tang, Kai Shen, and Tao Yang, "Compile/run-time support for threaded mpi execution on multiprogrammed shared memory machines," in *Symp. on Principles and Practice of Parallel Programming*, 1999, pp. 107–118.

[26] Devendra Kumar, "Development of a class of distributed termination detection algorithms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 2, pp. 145–155, 1992.

[27] John Mellor-Crummey and Michael Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, 1991.

[28] Rich Hickey, "Callbacks in C++ using template functors," http://www.tutok.sk/fastgl/callback.html, 1994.

[29] William McLendon III, Bruce Hendrickson, Steve Plimpton, and Lawrence Rauchwerger, "Finding strongly connected components in parallel in particle transport sweeps," in *Symp. on Parallel Algorithms and Architectures*, 2001, pp. 328–329.

[30] Steven Lumetta, Alan Mainwaring, and David Culler, "Multi-protocol active

messages on a cluster of SMPs," in *High Performance Networking and Computing Conf. (Supercomputing)*, 1997, [CDROM].

[31] Steve Bova, Rudolf Eigenmann, Henry Gabb, Greg Gaertner, Bob Kuhn, Bill Magro, Stefano Salvini, and Veer Vatsa, "Combining message-passing and directives in parallel applications," *SIAM News*, vol. 32, no. 9, pp. 10–14, 1999.

[32] L.A. Smith, "Mixed mode MPI/OpenMP programming," UK High-End Computing Technology Report, http://www.ukhec.ac.uk/publications/, 2000.

[33] Franck Cappello and Daniel Etiemble, "MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks," in *High Performance Networking and Computing Conf. (Supercomputing)*, 2000, pp. 51–63.

[34] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum, "High-performance, portable implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[35] Jarek Nieplocha, Jialin Ju, and Tjerk P. Straatsma, "A multiprotocol communication support for the global address space programming model on the IBM SP," *Lecture Notes in Computer Science*, vol. 1900, pp. 718–726, 2001.

[36] Ian Foster, Jonathan Geisler, Carl Kesselman, and Steven Tuecke, "Managing multiple communication methods in high-performance networked computing systems," *Journal of Parallel and Distributed Computing*, vol. 40, no. 1, pp. 35–48, 1997.

VITA

Steven Mack Saunders was born August 26, 1977, in Bremerton, Washington to Al and Sherry Saunders. He graduated with honors from Granbury High School, in Granbury, Texas in May, 1996. He graduated Magna Cum Laude with his B.S. in computer engineering at Texas A&M University in December, 2000. He will receive his M.S. in computer science from Texas A&M University in May, 2003. He can be reached via the Department of Computer Science at Texas A&M University / College Station, Texas 77843-3112 and via electronic mail at sms5644@cs.tamu.edu