ARMI: An Adaptive, Platform Independent Communication Library [*]

Steven Saunders
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
sms5644@cs.tamu.edu

Lawrence Rauchwerger
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
rwerger@cs.tamu.edu

ABSTRACT

ARMI is a communication library that provides a framework for expressing fine-grain parallelism and mapping it to a particular machine using shared-memory and message passing library calls. The library is an advanced implementation of the RMI protocol and handles low-level details such as scheduling incoming communication and aggregating outgoing communication to coarsen parallelism when necessary. These details can be tuned for different platforms to allow user codes to achieve the highest performance possible without manual modification. ARMI is used by STAPL, our generic parallel library, to provide a portable, user transparent communication layer. We present the basic design as well as the mechanisms used in the current Pthreads/OpenMP, MPI implementations and/or a combination thereof. Performance comparisons between ARMI and explicit use of Pthreads or MPI are given on a variety of machines, including an HP V2200, SGI Origin 3800, IBM Regatta-HPC and IBM RS6000 SP cluster.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming LanguagesProcessors; D.1.3 [**Programming Techniques**]: Parallel Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.2 [**Programming Languages**]: Language Classifications

General Terms

Languages

[*]This material is based upon work supported by the National Science Foundation under Grant No. 9872126, 9734471,9975018, 0113971, 0103742, DOE grant B347886 and Hewlett Packard

Keywords

RMI, MPI, Pthreads, OpenMP, RPC, run-time system, communication library, parallel programming

1. COMMUNICATION MODELS

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.

1.1 Shared Memory vs. Message Passing

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., locks and semaphores). The shared-memory model is considered easier to program, and is portable due to standards such as Pthreads [10] and OpenMP [22]. Furthermore, machines that implement this communication model do so by supporting it in hardware, thus generating little overhead. Sometimes, however, its lack of explicit data distribution mechanisms can hinder scalability [21]. The biggest disadvantage of shared memory communication models has been its inapplicability to very large machines. None of the largest massively parallel machines produced today support a single address space. Worth noting is the solution provided by the software distributed shared-memory (software DSM) model, which provides a software implementation of a global address space, albeit with performance penalties.

In the message passing paradigm, 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 early adoption of the Message Passing Interface, MPI-1.1 [16]. 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 favor shared-memory if it were available on all systems.

There is, however, another more subtle distinction between these two paradigms. 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. This style tends to increase the critical path of programs by the time it takes to communicate. Even with non-blocking messages the tendency is to lower the frequency of messages in favor of their length. 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. This means that we need to exploit finer grain parallelism (and communication) than the MPI programming style is suitable for.

Most modern machines today, including the massively parallel ones, consist of a network of nodes, where every node is in fact a small parallel machine. 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 use only MPI, and to implement MPI on shared memory machines as well. Indeed, almost all shared memory machines have very efficient MPI implementations that can take advantage of the shared memory communication medium. This approach has the disadvantages of being still slower than native shared memory communication and being harder to code. Another serious, but maybe 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 needed on each super-node of a large machine.

One-sided communication represents an improvement over the MPI-1 standard in terms of programming productivity because it combines some of the strengths of shared-memory and message passing [25]. A set of processes operate using private address spaces as well as sections of logically shared-memory. A process communicates 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 [19], LAPI [24] and the updated Message Passing Interface, MPI-2 [17]. Still, when using one-sided communication, the tendency is to write in coarse grain model (e.g., to copy-in/copy-out large chunks of data for computation).

1.2 Remote Method Invocation

Remote method invocation (RMI) is a communication model that works with object-oriented programs, where a process communicates by requesting a method from an object in a remote address space. It is currently most often associated with Java [12]. Synchronization is implied through the blocking semantics of RMI requests (e.g., Java RMI does not return until it completes [18]). RMI is related to its function-oriented counterpart, remote procedure call (RPC), which allows a process to request a function in a remote address space.

RMI raises the level of communication abstraction by dealing with methods, instead of directly accessing data and exposing the underlying shared-memory or message passing operations. However, it is generally associated with distributed applications, not high performance parallel applications [7, 8]. High performance run-time systems that do support RMI- or RPC-related protocols include Active Message [28], Charm++ [13], Tulip [1], and Nexus [7]. Whereas Java RMI always blocks until completion to 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.

RMI has several advantages over the previously presented protocols. It gives the flexibility to either move data (as in MPI) or work (methods) between processors, and thus can be more adapted to the needs of the application. Furthermore it works well in an object-oriented environment and places its user at a higher level of abstraction. Using an RMI based communication package distances the programmer from the details of the communication implementation and its associated cost, and allows for a finer grain programming style. The flexibility and relative simplicity of RMI pay off for any additional overhead associated with its use.

1.3 Contribution

The communication library presented in this paper, adaptive RMI (ARMI), makes a number of contributions. First, 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 styles, and hence allows for an easier parallelization. RMI also maintains data-hiding techniques, such as encapsulation, whereas other models must interface directly with data, bypassing the objects' interfaces. 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.

Second, ARMI provides the definition and implementation of a framework for expressing 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 performance possible without manual modification. ARMI adapts its behavior to the underlying architecture by using the native or lower level communication primitives and employs aggregation and scheduling to coarsen parallelism when necessary.

Third, ARMI serves as the run-time for the Standard Template Adaptive Parallel Library. STAPL is a parallel superset to the C++ Standard Template Library, and provides generic parallel containers and algorithms [23].

2. STAPL PROGRAMMING ENVIRONMENT

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

2.1 Overview

STL, the C++ Standard Template Library, 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) [27]. 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.

STAPL, the Standard Template Adaptive Parallel Library, is a parallel superset of STL that provides consistent results with its sequential counterpart [23]. 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.

2.2 Programming Style

A STAPL user composes a program 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. Because objects are conceptually shared, fine-grain parallelization is naturally expressible. For example, each element in the parallel sort can be transferred individually, instead of hard-coding aggregation at the user level with manual buffering.

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 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.

2.3 Case Study: Parallel Sorting

To illustrate how different parallel programming models affect communication, we consider a common parallel algorithm for sorting: sample sort [2]. 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.[1]
3. Sort each bucket.

Consider Figs. 1 and 2, 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.

In general, shared-memory algorithms are sequential until a fork (line 9), whereas message passing algorithms are always in parallel. The shared-memory code uses a shared STL vector to communicate splitters before forking (lines 6–7), as opposed to the message passing library calls (lines 5–6). Shared-memory must calculate each thread's local portion after the fork (lines 9-11), 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 13-15), and uses a barrier (line 17) to ensure proper event ordering of the distribution and sorting phase. Message passing buffers all the communication to each destination locally (line 11), and performs a single large communication phase (lines 13-18), 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 any communication/computation overlap and thus increases the critical path of the code. These issues are not

[1]Most implementations oversample the input to increase the chance of balanced buckets. We have removed this sub-step for simplicity.

```
void sort(int* input, int size) {
  int p = //...number of threads, 0-p...
  std::vector<int> splitters(p-1);
  std::vector< vector<int> > buckets(p);
  std::vector<lock> locks(p);
  for(int i=0; i<p-1; i++)
    splitters[p-1] = //...sample input...

  //...fork p threads...
  int id = //...thread id...
  for(i=size/p*id; i<size/p*(id+1); i++) {
    int dest = //...appropriate bucket...
    locks[dest].lock();
    buckets[dest].push_back(input[i]);
    locks[dest].unlock();
  }
  barrier();

  sort(bucket[id].begin(),bucket[id].end());
}
```

Figure 1: Example of a shared-memory sample sort

```
void sort(int* input, int localSize) {
  int p = //...number of processes, 0-p...
  std::vector<int> splitters(p-1);
  std::vector<int> bucket(p);
  int sample = //...sample input...
  Gather(&sample, ..., splitters, ...);

  std::vector< std::vector<int> > outBucket(p);
  for(i=0; i<localSize; i++) {
    int dest = //...appropriate bucket...
    outBucket[dest].push_back(input[i]);
  }
  for(int i=0; i<p-1; ++i)
    Send(outBucket[i] ...);
  for(int i=0; i<p-1; ++i) {
    Recv(tmp ...);
    bucket.insert(tmp.begin(), tmp.end());
  }

  sort(bucket.begin(), bucket.end());
}
```

Figure 2: Example of a message passing sample sort

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 algorithm.

We now contrast these implementations with STAPL. Because STAPL provides a parallel superset to STL, it contains a pAlgorithm for sorting, `p_sort`, which a user can use directly. We illustrate a possible implementation of `p_sort` in Fig. 3. 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.

The STAPL code maintains the shared-memory implementation's fine-grain approach by sending each element as it becomes available (lines 15–16). 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.

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

Figure 3: Example of a STAPL sample sort

3. DESIGN

ARMI is composed of three main subsystems: object registration, communication and synchronization. This section describes the fundamental requirements, the three subsystems, details concerning data transfer and ARMI's integration with STAPL.

3.1 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 produce-consumer relationship). A statement is asynchronous, meaning the sending process does not necessarily need to 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 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 [5]:

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, because all memory is private to a process.

2. *event ordering* - operations to inform a process 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 all of them, albeit in somewhat different ways. One of ARMI's goals is to raise the level of abstraction of these issues, yielding a clean interface that lends itself to efficient implementation with either underlying model.

3.2 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 separate address spaces (e.g., MPI), or a mixture of the two (e.g., mixed-mode MPI with OpenMP). Regardless of the granularity and sharing, we name the unit of execution a thread. Upon startup, all threads begin executing the same code in parallel, similar to SPMD MPI.

The shared-objects used in communication are distributed among threads, with each thread owning a local representative. Shared-objects are globally identified by an `rmiHandle`, while their local objects are identified by a thread id and `rmiHandle`. As such, all objects that are communication targets must be registered with ARMI to obtain an `rmiHandle`, which allows for proper translation to the local objects. Since each thread owns its local object, it is not necessary for a thread to use RMI to access it.

3.3 Communication

Threads independently access other threads' local objects via RMI. ARMI defines two basic forms of RMI, which map directly to the two fundamental types of communication.

1. `void async_rmi(dest, handle, method, 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. `rtn sync_rmi(dest, handle, method, 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 within an aggregated group of `async_rmi`'s.

The additional information required compared to a regular C++ method invocation is minimal. Only `dest` 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 `method` is a C++ 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. Because ARMI makes extensive use of C++ templates, the full type safety of regular method invocation is preserved.

Because `sync_rmi` blocks until it obtains the return value, it does not fully leverage the case when the return value is desired, but there is additional work to perform. As such, we have augmented the interface with a modified `sync_rmi` that immediately returns an opaque handle. This handle can be checked periodically to see if the return value has arrived yet, and return the data to the user. Such a prefetch-style interface provides further opportunities to overlap communication and computation.

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

1. `void broadcast_rmi(handle, method, 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(handle, method, 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). We are currently incorporating such advanced facilities into future versions of ARMI.

3.4 Synchronization

ARMI also addresses both fundamental 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. We recognize that full atomicity may not be necessary for correct execution in all situations. As such, we are actively pursuing multi-threaded implementations (e.g., using a communication thread to support a set of computation threads) that can relax constraints at run-time.

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. 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 while it waits. 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. Section 5.3 details the performance of a parallel algorithm for detecting strongly connected components in a graph that makes use of these advanced facilities of `rmi_fence`.

3.5 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 effect, 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 each class that may be transfered as an argument to implement a single method, `define_type`. This method defines the class's data members in a style similar to Charm++'s PUP interface [13]. 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.

Fig. 4 demonstrates a simple example. 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 call the `define_type` method (line 4). On line 6, `typer` will recursively call the `define_type` for objectB (line 13), to ensure the entire object is correctly packed.

```
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  }
```

Figure 4: Example of the `define_type` interface

3.6 Integration with STAPL

Fig. 5 shows the layout of STAPL's basic components, with an arrow representing a usage relationship. ARMI serves as the bottom layer, and abstracts the actual parallel communication model utilized via its RMI interface.

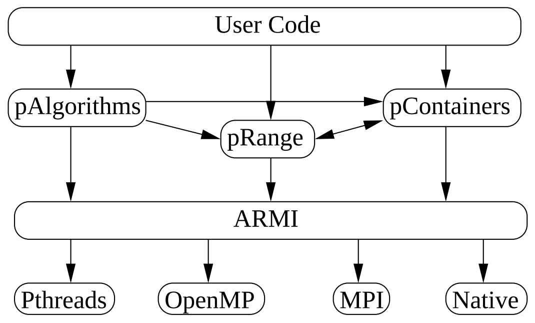

Figure 5: Basic STAPL Components

A pContainer is a distributed data structure. Although the user/programmer 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 ARMI internally 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. ARMI also helps facilitate an easier implementation by relaxing the constraint of matching sends and receives in message passing.

A pAlgorithm expresses a parallel computation in terms of `parallel_task` objects. These objects generally do not use ARMI 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 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. 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` operation handles this pattern via its automatic aggregation settings.

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.

4. IMPLEMENTATION

We have currently implemented ARMI using two different underlying programming models: Threads (shared-memory) and MPI (message passing). The Threads implementation determines at compile-time whether to utilize Pthreads or OpenMP. In addition, we have implemented a mixed-mode implementation, which combines the two models for systems such as clusters of SMP's. Since the interface remains the same, all that is required to use a different implementation is to re-compile.

In the rest of this section, we describe the basic mechanisms used to implement the three subsystems outlined in Section 3, object registration, communication, and synchronization.

4.1 Object Registration

All objects that may serve as communication targets must first be registered. Currently, we assume an SPMD style of programming, such that each thread will register the same number of objects in the same order. Although this object symmetry constraint may be restrictive in some cases, it facilitates a simple implementation where the assigned handle is simply an index into a table. It also coincides with the current implementation of pContainers. However, the constraint may be relaxed, at the expense of hashing each handle, and a more complicated handle negotiation protocol, such that both shared-objects and thread-specific objects are possible.

4.2 Communication

The communication subsystem is the most complicated to implement. Both the Threads and MPI implementations internally use a form of message passing, since the semantics of RMI imply one thread telling another thread to do something. However, the Threads implementation is able to simply trade points to RMI request buffers, whereas MPI physically transfers these buffers. To support periods of high communication, message transfer is pipelined by using multiple internal communication buffers per possible destination thread. While one buffer is being transfered to its destination, a different buffer can be used to aggregate new requests.

All `async_rmi`'s are automatically aggregated by ARMI, and issued in groups based on a default or user-defined aggregation factor. An appropriate default aggregation buffer size can be configured for each machine during installation. Because `sync_rmi`'s block, they are not aggregated. However, a single `sync_rmi` can be included at the end of a group of `async_rmi` requests to reduce message traffic.

4.2.1 Request 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 [28, 1, 7, 24]. 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 [28, 1, 7, 24, 19]. Although this solution is often avoided due to the high cost of interrupts, it does guarantee a timely response with minimal user interruption (i.e., no extraneous polls).

3. *blocking communication thread* - a separate communication thread posts a blocking receive for incoming requests [7]. Upon arrival, the 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 [7]. 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 performed within ARMI library 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 handle the alternative case of extremely frequent communication, 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.

In general, when a group of RMI requests are received, as in an aggregated group of `async_rmi's`, they are processed in FIFO order. However, the one notable complication is the possibility of nested RMI requests, which occurs 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 occurs, because a necessary request 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.

4.3 Synchronization

The synchronization subsystem involves two major functions. The `rmi_wait` function waits for a single incoming request. 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.

The `rmi_fence` function waits for all threads to arrive and complete all outstanding 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 [1]. As such, we were forced to implement our own tree-based barrier implementations for use in the fence protocol. To address the second issue, we overlay a distributed termination detection algorithm [14]. In short, the algorithm tracks the number of sends and receives performed by each thread, performs a distributed summation, and declares termination as soon as the sum equals zero. The summation is essentially a reduction that is easily incorporated into the barrier's arrival and release messages.

4.4 Mixed-Mode

The mixed-mode implementation of ARMI implements multi-protocol communication. It starts by creating MPI processes, which each in turn spawn a number of internal, local threads. The aggregate threads provide a flat view of the parallelism, but use shared-memory communication to local threads (i.e., within the MPI process), and MPI to remote threads.

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 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 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 scheme, synchronizing the local shared-memory threads first, with the root thread continuing to perform the MPI synchronization.

Some vendor supplied MPI implementations are thread-safe, and hence facilitate threads making MPI calls in parallel. However, many other implementations are not thread-safe, and require running in serialized mode, meaning each MPI operation must be protected by a lock. Blocking operations such `MPI_Wait`, which waits until a given communication completes, must be replaced with a loop that balances a non-blocking `MPI_Test` with the shared-memory poll operation. These loops can cause great contention by repeatedly polling the MPI library. During such a loop, the MPI library is only tested every nth iteration, where n is adjustable to a given platform. This solution does not always provide optimal results, so we are pursuing a multi-threaded implementation, such that a single communication thread can handle all MPI communication, reducing the need for locks.

5. PERFORMANCE

We tested the implementations of ARMI 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 symmetric multiprocessor consisting of 16 200MHz PA-8200 processors with 2MB L2 caches. The *O3800* is a hardware DSM, hypercube-based CC-NUMA 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.

5.1 RMI Overhead

The ARMI abstraction includes a number of overheads compared to regular method invocation. Section 5.2 details transfer latency, while this section measures the cost of creating and executing an RMI locally (i.e., everything but transfer latency). The major abstraction involved in building an RMI request is using the method pointer (member function 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 1 measures the cost of invoking an empty method directly, via a method pointer, and via ARMI's `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 invocation. ARMI requires a substantial amount more than this however.

Table 1: 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

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.

5.2 Latency

We tested 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. The MPI benchmark explicitly matches sends and receives.

The resulting wall clock times are shown in Table 2. Since ARMI is implemented on top of 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-tuned 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 internally 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-tuned 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-tuned MPI. Another source of overhead is that ARMI transfers RMI header information during the ping-pong, and hand-tuned 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 communication requests, by retiming the ping-pong benchmark using multiple consecutive pings before a single pong. ARMI's aggregation factor was varied from 4 to 2048 messages.

Fig. 6 show the results for MPI on the O3800. ARMI is faster after issuing just 10 pings, and yields a 15-fold improvement after 10,000 pings. In this case, the optimal aggregation factor is 256 messages (an 8KB buffer), which means ARMI will automatically translate the 10,000 pings into 40 large `MPI_Sends`, as opposed to the MPI benchmark using 10,000 small `MPI_Sends`. The general trend for the aggregation factor is a parabolic curve. Initially, aggregation alleviates much of the network traffic and makes better use of the available bandwidth. If used too liberally however, aggregation increases the amount of work that needs to be performed at the end of the computation phase, thus increasing the critical path, as seen for an aggregation of 2048. These same results hold for shared-memory systems, although the benefit is less given the already low overheads of shared-memory. For Pthreads on the O3800, an aggregation buffer of 256 messages provides a 3-fold improvement versus non-aggregated Pthreads. Although Pthreads and MPI can both make use of explicit aggregation hand-coded by the user to gain similar benefits, we note that ARMI performs aggregation automatically, and its settings can be tuned to which platform to provide maximum performance on a variety of different systems.

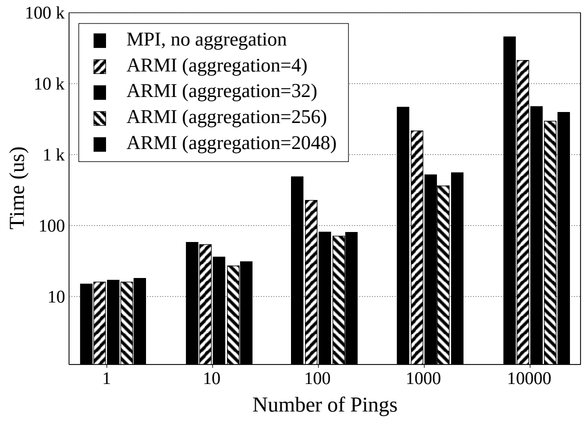

Figure 6: MPI Latency (O3800)

5.3 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 implementation that required twice as many lines of code. The implementation buffers all elements locally, then performs an all-to-all merge before the final sorting phase.

Fig. 7 shows the scalability versus running the parallel algorithm with one processor on the Regatta. It compares

Table 2: Latency (us) of explicit communication and ARMI (`async_rmi`/`sync_rmi`).

	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's Threads and MPI implementations for one million integers, distributed uniformly among threads. Both are using an aggregation factor of 256 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 scalability starting at 4 processors occurs 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-tuned MPI, which is unable to hide any communication latency.

Fig. 8 shows the scalability as the dataset is increased from 1M to 50M integers. Even given the much larger dataset, the Threads implementation is still 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.

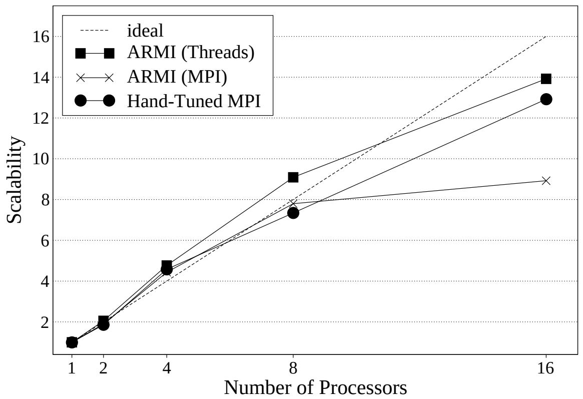

Figure 7: Scalability of sorting 1M integers (Regatta)

More complex algorithms 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 [11]. 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.
3. Remove each strongly-connected component, and iterate if there are remaining edges.

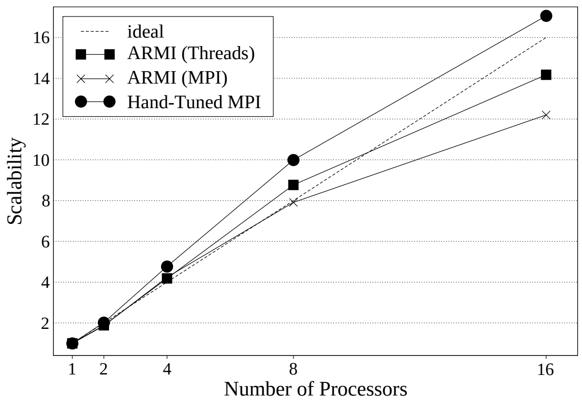

Figure 8: Scalability of sorting 50M integers (Regatta)

The communication is fine-grain, because an RMI is generated every time an edge is marked or trimmed. As such, the communication library is used to aggregate communication sufficiently for performance. The input to the algorithm is a set of 10 meshes, each consisting of approximately 10k vertices and 30k edges, for a total of 338 strongly-connected components.

Fig. 9 shows the scalability versus running the parallel SCC on one processor on the V2200, using a variety of aggregation factors. Super-linear performance is possible because as processors are added, more pivots are considered, which greatly decrease the number of iterations necessary for convergence. For example, the number of iterations required are cut in half when moving from one to two processors for this graph. As shown, aggregation has a significant effect. As expected, no aggregation yields poor performance, whereas too much aggregation degrades performance, as shown in the difference in aggregation from 75 and 100. The later effect occurs because the algorithm is irregular, and some threads will be forced to wait until other threads sufficiently fill their aggregation buffers.

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

Fig. 10 shows scalability for hand-tuned MPI, as well as the MPI and mixed-mode implementations of ARMI on

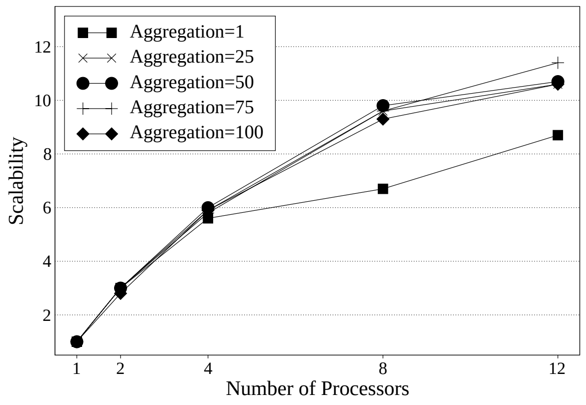

Figure 9: Scalability of detecting strongly-connected components (V2200)

the RS6000 for a 500x500 matrix/thread 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 shared-memory RMI. 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. However, the mixed-mode implementation is still able to outperform 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). Both ARMI implementations remain comparable with hand-tuned MPI.

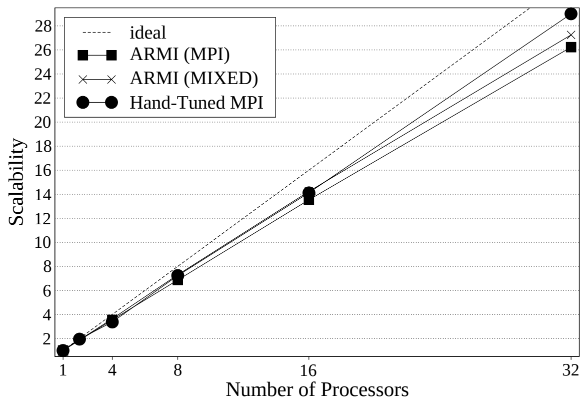

Figure 10: Scalability of Jacobi iteration (RS600)

6. RELATED WORK

A number of other libraries provide RMI-based high performance communication with similar goals to ARMI.

Active messages is an extension to one-sided communication that includes specifying a message handler on the receiving process [28]. The handler is intended to quickly integrate the message into the ongoing computation, as opposed to general purpose computation as in RPC.

Tulip is a wrapper around existing models, and serves as a compiler target for the pC++ programming language [1]. 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, by internally buffering and issuing fewer messages [19].

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 [7].

Charm++ is an object-oriented parallel programming language that utilizes non-blocking RMI communication [13]. 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 libraries, ARMI includes both blocking and non-blocking communication functions. 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.

Many other researchers have considered combining shared-memory and message passing into mixed-mode, often called hybrid and multi-protocol, parallel programs. Most of these studies focus on a two-level scheme, by explicitly using MPI for coarse-grain parallelism, and then instrumenting inner loops with OpenMP [3, 26, 4]. This programming style is most often referred to as hybrid parallelism. Other libraries, including some MPI implementations [9], 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 [15, 20]. Nexus implements a unique multi-protocol layer that adaptively selects the best transport protocol [6]. Although the obvious approach is to always use the fastest protocol, other options, such as quality of service and security, may also be considered.

ARMI also focuses on multi-protocol communication. In general, two-level schemes require learning two parallel programming models, and explicitly specifying where to use them. In contrast, only one model is necessary with multi-protocol communication, and the library can be tuned to appropriately map the available parallelism to each new machine, instead of requiring additional changes to user code. In addition, multiple levels of parallelism can be specified in a code, and the library can adaptively serialize or exploit the extra parallelism, based on the underlying machine and overhead of communication. Unlike other existing systems, the current implementation of the ARMI translates directly into MPI, OpenMP/Pthreads, or a combination, leveraging their already highly tuned, vendor provided facilities.

7. CONCLUSIONS

In this paper we present ARMI, a high level communication library that we have developed for our generic parallel library, STAPL, but which can be used independently, in any parallel C++ code. ARMI is based on an adaptive implementation of RMI, which 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.

Our first implementation shows good results, which we will continue to improve upon. We are currently working on finding a better way to alternate between communication activity and computation by using multi-threading. This approach promises to be very useful on systems that may have a dedicated communication processor or that supports multi-threading in hardware. Furthermore, such an implementation can help decouple the design of computation activity (the real work) from communication (an overhead due to the distributed nature of large machines).

8. REFERENCES

[1] P. Beckman and D. Gannon. Tulip: A portable run-time system for object-parallel systems. In *Int. Parallel Processing Symp.*, pages 532–536, 1996.

[2] G. Blelloch, C. Leiserson, B. Maggs, G. Plaxton, S. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Symp. on Parallel Algorithms and Architectures*, pages 3–16, 1991.

[3] S. Bova, R. Eigenmann, H. Gabb, G. Gaertner, B. Kuhn, B. Magro, S. Salvini, and V. Vatsa. Combining message-passing and directives in parallel applications. *SIAM News*, 32(9):10–14, 1999.

[4] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. In *Supercomputing*, pages 51–63, 2000.

[5] D. Culler and J. Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA: Morgan Freeman Publishers, 1999.

[6] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40(1):35–48, 1997.

[7] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.

[8] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon. Requirements for and evaluation of RMI protocols for scientific computing. In *Supercomputing*, pages 76–102, 2000.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[10] 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.

[11] W. M. III, B. Hendrickson, S. Plimpton, and L. Rauchwerger. Finding strongly connected components in parallel in particle transport sweeps. In *Symp. on Parallel Algorithms and Architectures*, pages 328–329, 2001.

[12] B. Joy, G. Steele, J. Gosling, and G. Bracha. *Java(TM) Language Specification (2nd Edition)*. Reading, MA: Addison-Wesley Pub Co, 2000.

[13] L. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 91–108, 1993.

[14] D. Kumar. Development of a class of distributed termination detection algorithms. *IEEE Trans. on Knowledge and Data Engineering*, 4(2):145–155, 1992.

[15] S. Lumetta, A. Mainwaring, and D. Culler. Multi-protocol active messages on a cluster of SMPs. In *Supercomputing*, 1997.

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

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

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

[19] J. Nieplocha and B. 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.

[20] J. Nieplocha, J. Ju, and T. P. Straatsma. A multiprotocol communication support for the global address space programming model on the IBM SP. *Lecture Notes in Comp. Science*, 1900:718–726, 2001.

[21] D. Nikolopoulos, E. Ayguad, J. Labarta, T. Papatheodorou, and C. Polychronopoulos. The tradeoff between implicit and explicit data distribution in shared-memory programming paradigms. In *Int. Conf. on Supercomputing*, pages 23–37, 2001.

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

[23] A. Ping, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In *Int. Workshop on Languages and Compilers for Parallel Computing*, 2001.

[24] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and experience with LAPI: A new high-performance communication library for the IBM RS/6000 SP. In *Int. Parallel Processing Symp.*, pages 260–266, 1998.

[25] H. Shan and J. P. Singh. A comparison of MPI, SHMEM and cache-coherent shared address space programming models on the SGI origin2000. In *Int. Conf. on Supercomputing*, pages 241–266, 1999.

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

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

[28] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Int. Symp. on Computer Architecture*, pages 256–266, 1992.