Parallel Processing Letters

ARMI: A High Level Communication Library for STAPL*

Nathan Thomas, Steven Saunders, Tim Smith, Gabriel Tanase, and Lawrence Rauchwerger
Parasol Lab, Dept. of Computer Science, Texas A&M University
College Station, TX 77843, USA

Received April 2004
Revised May 2006
Communicated by Kemal Ebcioglu

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. 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, Origin 3800, IBM Regatta and IBM RS/6000 SP cluster.

Keywords: RMI, MPI, Pthreads, OpenMP, Run-time system, Communication library

1. A Comparison of Communication Models

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

1.1. Shared Memory vs. Message Passing

The most common communication models in parallel programming are shared-memory and message passing. In shared-memory, threads share a global address space. A thread communicates by *storing* to a location in the address space, which another thread can then *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 like Pthreads [10]

*This research supported in part by NSF Grants EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, ACI-0326350, and by the DOE.

and OpenMP [20]. Furthermore, machines that implement this model do so by supporting it in hardware, thus generating little overhead. The biggest disadvantage of shared memory models has been its inapplicability to very large machines.

In the message passing paradigm, a group of processes cooperates using private address spaces. A process typically communicates by explicitly *sending* a message to another process, which must use a matching *receive*. However, newer libraries do employ some form of one-side communication [18,24,16], Synchronization is implied through the blocking semantics of sends and receives. Users must manually distribute data across the system, making dynamic or irregular applications difficult to code. The model is considered more difficult to program than shared memory but flourishes because of its ability to scale to massively parallel systems and its standardization as the Message Passing Interface (MPI).

There is a subtle distinction between these two paradigms. Programmers are aware of the higher latency of MPI communications and so tend to minimize its impact by using a coarse grain parallelization/communication model (in the BSP style [28]). 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. We could argue that MPI programs scale well if the ratio *local work/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.

Modern large parallel machines usually consist of a network of nodes, where every node is in fact a small parallel machine itself. This implies that we need to exploit, concurrently, both coarse grain and fine grain parallelism. The widely adopted approach to writing portable code across such platforms has been to use only MPI, making writing in such an adaptive manner extremely tenuous.

1.2. Remote Method Invocation

Remote method invocation (RMI) is a communication model that has its origins in the RPC (remote procedure call) protocol. It is most often associated with Java [11]. 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 [17]). RMI is related to its function-oriented counterpart, remote procedure call (RPC) [30], which allows a process to request a function in a remote address space. 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 [8,9]. High performance run-time systems that do support RMI- or RPC-related protocols include

Active Message [29], Charm++ [12,13], Tulip [1], and Nexus [8]. 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.

We believe that RMI has several advantages over the previously presented protocols. It gives the flexibility to either move data (as in MPI) or methods (work) between processors, and thus can be more easily 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 simplicity of RMI more than pays off for any additional overhead associated with its use.

1.3. Contribution

ARMI [23] makes several contributions. It provides a communication style, RMI, that takes advantage of the natural communication involved in object-oriented programs, method invocations. 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 RMIs do not require matching operations, incoming requests are scheduled internally and advanced synchronization mechanisms, similar to one-sided models, are provided.

ARMI also defines and implements 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 as necessary.

Finally, ARMI serves as the run-time system for STAPL. STAPL, the Standard Template Adaptive Parallel Library, is a parallel superset to the C++ Standard Template Library, which provides generic parallel containers and algorithms [22,21]. In this paper, we show performance results for ARMI and its use with several STAPL components: parallel sorting of a pVector, LU decomposition of a pMatrix, strongly

connected component detection on a pGraph, and pArray element updates.

2. Our Programming Environment: STAPL

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

2.1. STAPL 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) [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 algorithm is expressed in terms of iterators instead of container methods, allowing the algorithm to be used with a variety of containers.

The Standard Template Adaptive Parallel Library [22,21] (STAPL) is a parallel superset of STL that provides parallel containers, *pContainers*, and parallel algorithms, *pAlgorithms*, that are bound together by *pRanges*. The pContainers provide a shared-object view of physically distributed data. A pRange is a view of a work space, which is the set of tasks to be performed in a parallel computation. A pRange can be recursively partitioned into subranges defined on disjoint portions of the work space. A leaf subrange in the pRange hierarchy is the smallest schedulable entity and represents a subset of the data space and the set of tasks that will operate on it. The pAlgorithms use pRanges to efficiently operate on data in parallel.

2.2. STAPL's Programming Style

A STAPL user composes an application by specifying pContainers, initializing them, 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 provide their own implementation.

ARMI provides a shared-object view to STAPL. 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 SMPs, 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.

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 steps, the first of which is to sample the elements of the input data to select $p-1$ of them to be used as splitters.[†] These splitters are sorted in non-decreasing order and used to partition the input data into p buckets such that every element in bucket_i is less than every element in bucket_{i+1} (and the splitter between them). Second, each processor scans its elements sending them to the processor handling the appropriate bucket. Finally, each processor sorts its bucket, and copies the sorted range back to the original data structure.

Consider Figures 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 10), 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 9-12), 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 13-16), and uses a barrier (line 18) to ensure event ordering of the distribution and sorting phase. Message passing buffers all the communication to each destination (line 11-14), and

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

```
1  void sort(int* input, int size) {
2    int p = //...number of threads
3    std::vector<int> splitters(p-1);
4    std::vector< vector<int> > buckets(p);
5    std::vector<lock> locks(p); //sychronize proc. bucket access
6    for(int i = 0; i < p-1; ++i)
7      splitters[p-1] = //...sample input...
8    std::sort(splitters.begin(), splitters.end());
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_bucket = //...pick bucket using splitters...
14     locks[dest].lock();
15     buckets[dest].push_back(input[i]);
16     locks[dest].unlock();
17   }
18   barrier();  //wait for all threads to finish bucket insertion
19
20   sort(bucket[id].begin(), bucket[id].end());
21   //...copy buckets back to input source
22 }
```

Figure 1: Example of a shared-memory sample sort

performs a single large communication phase (lines 15-20), implicitly ensuring event ordering.

Neither implementation is optimal. Shared-memory's extensive use of locking causes contention on the buckets. Message passing performs computation and communication in separate phases, which eliminates their overlap, increasing the critical path of the code. These issues are not intrinsic to the algorithm, only to the underlying communication model and implementation. Improvements can be made that require additional code, which further strays from the base algorithm.

We now contrast these implementations with one using ARMI and the STAPL pVector (see Figure 3). The code fragment shown in the figure contains additional code to wrap the algorithm in a object-oriented class. The core of the algorithm (contained in the `execute` method) is actually shorter than either of the previous implementations.

The ARMI 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, or even aggregating all messages to become the coarse grained message passing implementation at run-time.

```
1  void sort(int* input, int localSize) {
2    int p = //...number of processes, 0-p...
3    std::vector<int> splitters(p-1);
4    std::vector<int> bucket(p);
5    int sample = //...sample my local input...
6    AllGather(&sample, splitters); //all procs. get all splitters
7    sort(splitters.begin(), splitters.end());
8
9    //outBucket aggregates bucket insertions
10   std::vector< std::vector<int> > outBucket(p);
11   for(i = 0; i < localSize; i++) {
12     int dest = //...pick bucket(processor) using splitters...
13     outBucket[dest].push_back(input[i]);
14   }
15   for(int i = 0; i < p-1; ++i)
16     Send(outBucket[i] ...);
17   for(int i = 0; i < p-1; ++i) {
18     Recv(tmp ...);
19     bucket.insert(tmp.begin(), tmp.end());
20   }
21
22   sort(bucket.begin(), bucket.end());
23   //...copy bucket contents back to input source
24 }
```

Figure 2: Example of a message passing sample sort

3. Design and Implementation

In this section we discuss mechanisms for object registration, data packing, and method invocation along with ARMI's consistency model and integration in STAPL. Currently, ARMI is implemented for Threads (shared-memory) and MPI-1.1 (message passing). The threads model determines at compile-time whether to use Pthreads or OpenMP. In addition, there is a mixed-mode version, which suitable for environments such as clusters of SMPs. Since the interface remains the same, all that is required to use a different implementation is to recompile.

3.1. Object Registration

The shared objects used in communication are distributed among threads, with each thread owning a local representative. Shared objects are identified by an `rmiHandle`, and 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. Currently, we assume an SPMD style of programming, such that each thread will register the same number of objects in the same order. This design coincides with that of STAPL's pContainers. However, users can bypass this symmetry constraint by using ARMI calls that enable them to manually assign rmiHandles.

```
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 = //...pick bucket using splitters...
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     //...copy buckets back to input source
22   }
23 }
```

Figure 3: Example of a sample sort using ARMI

3.2. Communication

Threads independently access other threads' local objects via RMI. There are three basic primitives in ARMI, one for requests with no return value and two for those returning a value. For both return and no return, non blocking versions exist allowing greater overlap of communication and computation. Nonblocking calls are automatically aggregated by ARMI, and issued in groups based on a default or user-defined aggregation factor.

1. `void async_rmi(dest, handle, method, arg1...)` - Nonblocking. Issues the RMI request and returns immediately. Subsequent synchronization calls, such as `rmi_fence`, may be used to wait for completion of requests.

2. `rtn sync_rmi(dest, handle, method, arg1...)` - Blocking. 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.

3. `rtn sync_rmi(opaque, dest, handle, method, arg1...)` - Nonblocking. issues the RMI request and and returns immediately. The opaque data object has a `ready()` function that can be polled for return arrival and a `value()` function to subsequently read that value. Aggregation and explicit synchronization is possible as with an `async_rmi` call.

The additional information required compared to a regular C++ method invocation is minimal. Only `dest` is completely new, which specifies the destination thread. As discussed earlier, `rmiHandle`s facilitate proper translation between threads. The `method` is a C++ pointer to a method defined by the object. Important to note is that the type safety mechanisms of C++ are maintained, through ARMI's extensive use of C++ templates.

ARMI also incorporates many-to-one and one-to-many communication to support common collective communication patterns. By default, these operations are globally collective. However, subgroups can be defined and used, with semantics that are similar to that employed with MPI communicators.

1. `void broadcast_rmi(group, 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(group, 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.

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. Request scheduling is not a new problem, and several approaches exist. Explicit polling [29,1,8,24] offers timely responses, put polling may effect the performance of local computation. Interrupt based approaches [29,1,8,24,18], offer relatively timely responses as well but with substantial overhead. Finally, blocking and nonblocking communication threads have been proposed [8] and can be useful when appropriate thread scheduling can be guaranteed.

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. We have also prototyped a non blocking communication thread version and are studying the performance and benefits of this approach.

3.3. Consistency and Synchronization

A clear discussion of memory consistency [6] issues is often absent when describing a parallel library or communication system. This omission is unfortunate, as it is left to the programmer to learn by trial and error the software's real behavior.

We describe the consistency model of ARMI in terms of the processing of requests. We do this as all communication between objects on different processors are serviced at this level. The consistency is discussed in terms of both the atomicity

and ordering of RMIs. These should be taken as minimum guarantees for use when programming directly to the ARMI API. In reality, STAPL components that use ARMI often provide interfaces to the end user that offer stricter guarantees.

Servicing of incoming RMIs on a thread are guaranteed to be atomic. By atomic, we mean that once the processing of a request has begun, it will exclusively continue to completion without interruption. This guarantee is removed when `sync_rmi` calls are invoked by a request being serviced. In this case, polling and processing of subsequent requests occurs while waiting for the return value, in an attempt to prevent deadlock. However deadlock is still a possibility, and it is the responsibility of the user to avoid creating programs where it can occur.

ARMI enforces the atomicity of accesses to an object's data by requiring that all accesses be performed through invocations of the object's methods. ARMI is informed about entry and exits from these methods by the object and determines whether a conflict exists. If the processing of the next RMI request causes conflict, this operation is blocked until the local computation has finished its access of the object.

A strict ordering on the servicing of RMIs sent from one thread to other threads is in general not guaranteed. However, there are some cases where some partial ordering is ensured. Multiple RMIs sent from a local thread to a single destination are processed in the order sent. Blocking `sync_rmi` calls from a local thread to different remote threads are serviced in the local thread's program order.

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. Hence, it provides stronger guarantees than that of a typical barrier (e.g., not only the relative program position of threads) and can be used as an RMI ordering mechanism.

3.4. 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.

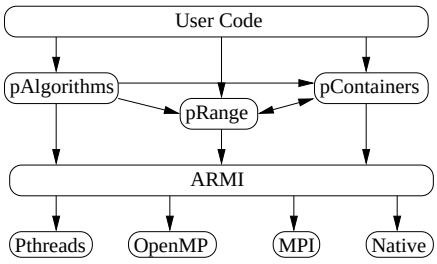
To support message passing as an implementation model, ARMI requires each class that may be transfered as an argument to implement a `define_type` method. This method defines the class's data members in a style similar to Charm++'s PUP interface [12,13]. Each data member is defined as local (i.e., automatically 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 exam-

ple 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. Interfaces also exist for user defined pack/unpack functions for objects of a given class. These functions are then inter-operable with the automated framework and can be used where programmer knowledge enables some packing optimization (i.e., only some fields of class needed).

Fig. 4(a) 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(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(typer& t) {
14     t.local(size);
15     t.dynamic(array, size);
16   }
17 }
```

(a) Example of the `define_type` interface



(b) Basic STAPL Components

Figure 4:

3.5. Integration with STAPL

Fig. 4(b) 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.

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 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. ARMI 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 end, the user can combine pContainers and pAlgorithms to write a program without concern for the underlying communication.

4. 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 (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 SMPs, consisting of 4 332MHz PowerPC 604e processors with 256kB L2 caches per node, with nodes connected by a dedicated high-speed switch.

4.1. RMI Overhead

The ARMI abstraction includes a number of overheads compared to regular method invocation. 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 1 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 invocation. ARMI requires a substantial amount more than this however.

	V2200	O3800	Regatta	RS6000
Direct	65	12	8	60
Mthd. Ptr.	132	26	14	121
ARMI	933	325	197	842

Table 1: Overhead of method invocation (ns)

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.

4.2. Latency

We tested ARMI's latency 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 a `sync_rmi` (blocking). The Pthreads version 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.

	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

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

We also tested the impact of aggregation on message latency when issuing many communication requests, by re-timing the ping-pong benchmark using multiple consecutive pings before a single pong. Aggregation was varied from 4 to 2048 messages.

Fig. 5 shows 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 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.

5. Algorithm Performance

A variety of parallel algorithms have been implemented using ARMI. One example is the case study, sample sort. 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. 6(a) shows the scalability results comparing ARMI's Thread and MPI im-

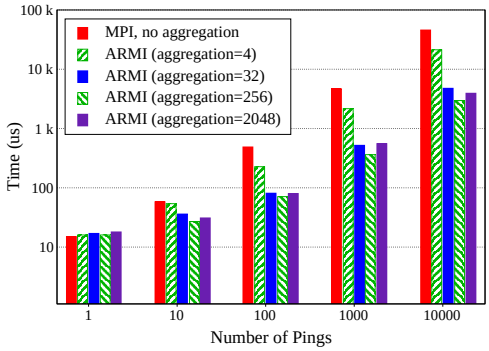


Figure 5: MPI Latency (O3800)

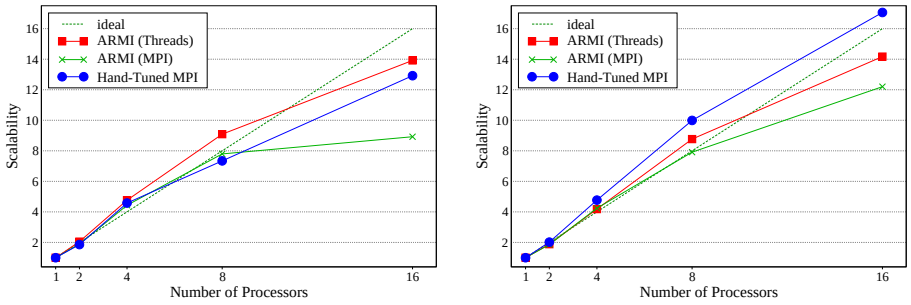
plementations versus a solely MPI-based implementation for sorting one million integers on the Regatta. In this scalability study, speedups are measured using the parallel algorithm (with P=1) as a baseline. This has been done for comparing the relative performance of implementations based on the three different communication paradigms. Both ARMI implementations use an aggregation factor of 256 requests.

As shown, Threads is able to sustain more parallelism at this level of work than MPI, as well as outperform the hand-tuned MPI. The super-linear 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 has lower latency than MPI, it is 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. 6(b) 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 super-linear speedups at 8 processors occur when the data first fits into the 32MB L3 cache.

5.1. Aggregation

We investigated the effects of aggregation by looking at three parallel applications that use different pContainers found in STAPL. The first code is a kernel often used for bandwidth testing that simulates parallel, random updates of a large distributed structure, specifically a STAPL pArray. Next, we look at the LU decomposition [5] of a matrix. In this case a STAPL pMatrix is used, which is built on the foundation of the Matrix Template Library [25]. The final code, pSCC, de-



(a) Sorting 1M ints (Regatta) (b) Sorting 50M ints (Regatta)

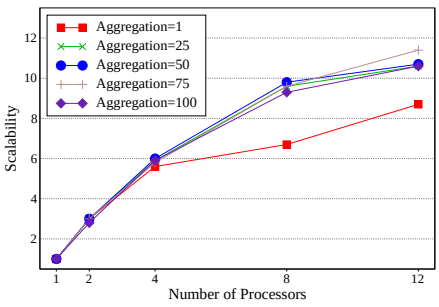
Figure 6:

tects strongly-connected components in an arbitrary pGraph using the algorithm described in [15]. Performance results for the benchmark applications with various aggregation factors are shown in Figure 7.

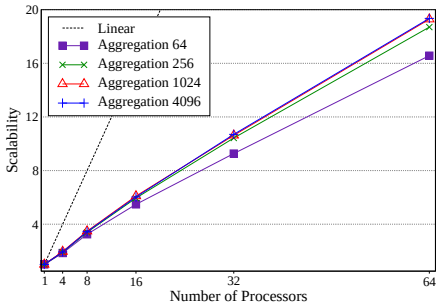
The results for pSCC, shown in Figure 7(a), demonstrate the importance of aggregation in obtaining optimal and scalable performance. Without aggregation, performance begin to drops substantially above four processors. Of note is the sweet spot occurring with an aggregation factor of 75. Above and below this value, aggregated performance is good but suboptimal. From investigation, it appears that this value is not solely dependent on the underlying communication infrastructure (MPI) or system architecture. Instead, the sensitivity is also program specific and the optimal value of aggregation is dependent greatly on the input data (e.g., structure of the graph). We are working on adaptive mechanisms in ARMI to customize the aggregation at run time for applications such as this. However, much of the gains from aggregation can come with a fixed factor, even in this dynamic case.

The need for aggregation is even more pronounced with the LU decomposition of a pMatrix (Figure 7(c)). Without aggregation, the algorithm does not scale well at all. There's only a speedup of 3 on 64 processors, versus 35 with aggregation. Further experiments backup what the graph suggests, additional aggregation offers no benefit but does not negatively impact performance. We are still investigating precisely why this is, but we believe that the additional aggregation is not greatly utilized due to synchronizations enforced by the algorithm.

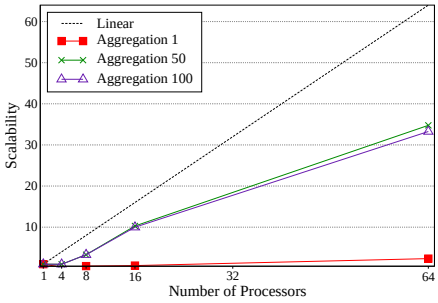
As stated earlier, the pArray update kernel, a bandwidth test that is communication bound. Indeed, communication was so overwhelming at the fine grain level, that MPI fails to properly service the program with an aggregation value of one. The program is highly regular, as the plots in Figure 7(b) suggest, and it benefits a measurable amount from additional aggregation; there is a 20% improvement in performance with an aggregation of 4,096 instead of 64 when running on 64 processors. Furthermore, the trend is for this gap to increase as the difference is only 5% at four processors.



(a) pGraph pSCC (V2200)



(b) pArray random update (RS6000)



(c) pMatrix LU decomposition (RS6000)

Figure 7: Scalability Results

These experimental results allow us to draw several conclusions. Aggregation of RMI requests is imperative for performance and the necessity of it grows with the addition of more processors. Additionally, in some applications there is an optimal selection of aggregation factor that properly balances the need to reduce communication overhead while still providing sufficient work to minimize processor idling.

6. 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 [29]. The handler is intended to quickly integrate the message into the ongoing computation, as opposed to general purpose computation as in RPC. 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 [8]. Charm++ is an object-oriented parallel programming language that uti-

lizes non-blocking RMI for communication [12,13]. 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. ARMCI is a one-sided communication library that focuses on optimizing strided data communication, by internally buffering and issuing fewer messages [18]. 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.

Others have considered combining shared-memory and message passing into mixed-mode programs, dubbed hybrid [3,26,4] or multi-protocol [14,19,7] approaches. Hybrid approaches use multiple levels of parallelism, employing a different protocol at each level, all of which are visible to the end user. Multi-protocol can be used on a single level, and employ different protocols for different destinations. ARMI focuses on multi-protocol communication and thus requires the end user to only master one set of communication primitives.

Finally, 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 will be improved 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).

References

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

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

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

[4] Cappello, F., and Etiemble, D. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. In *High Performance Networking and Computing Conf. (Supercomputing)* (2000), pp. 51–63.

[5] Choi, J., Dongarra, J. J., Ostrouchov, S., Petitet, A. P., Walker, D. W., and Whaley, R. C. The design and implementation of the ScaLAPACK LU, QR and Cholesky factorization routines. Tech. Rep. ORNL/TM-12470, Oak Ridge, TN, USA, 1994.

[6] Culler, D. E., Singh, J. P., and Gupta, A. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, inc., San Francisco, CA, 1999.

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

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

[9] Govindaraju, M., Slominski, A., Choppella, V., Bramley, R., and Gannon, D. Requirements for and evaluation of RMI protocols for scientific computing. In *High Performance Networking and Computing Conf. (Supercomputing)* (2000), pp. 76–102.

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

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

[13] Kale, L., and Krishnan, S. Charm++: Parallel programming with message-driven objects. In *Parallel Programming using C++*, G. Wilson and P. Lu, Eds. Cambridge, MA: MIT Press, 1996, pp. 175–213.

[14] Lumetta, S., Mainwaring, A., and Culler, D. Multi-protocol active messages on a cluster of SMPs. In *High Performance Networking and Computing Conf. (Supercomputing)* (1997), p. [CDROM].

[15] McLendon, W., Hendrickson, B., Plimpton, S., and Rauchwerger, L. Finding strongly connected components in parallel in particle transport sweeps. In *Symp. on Parallel Algorithms and Architectures* (2001), pp. 328–329.

[16] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, May 1998.

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

[18] Nieplocha, J., and Carpenter, B. 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), p. [CDROM].

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

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

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

[22] Rauchwerger, L., Arzu, F., and Ouchi, K. Standard templates adaptive parallel library. In *Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers* (1998), pp. 402–409.

[23] Saunders, S., and Rauchwerger, L. ARMI: an adaptive, platform independent communication library. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP)* (2003), ACM Press, pp. 230–241.

[24] Shah, G., Nieplocha, J., Mirza, J., Kim, C., Harrison, R., Govindaraju, R., Gildea, K., DiNicola, P., and Bender, C. 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.

[25] Siek, J. G., and Lumsdaine, A. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE* (1998), pp. 59–70.

[26] Smith, L. Mixed mode MPI/OpenMP programming. *UK High-End Computing Technology Report* (2000).

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

[28] Valiant, L. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 103–111.

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

[30] Waldo, J. Remote procedure calls and java remote method invocation. *IEEE Concurrency 6*, 3 (1998), 5–7.