A Parallel Communication Infrastructure for STAPL

Steven Saunders
ssaunders@tamu.edu

Lawrence Rauchwerger *
rwerger@cs.tamu.edu

Abstract: *Communication is an important but difficult aspect of parallel programming. This paper describes a parallel communication infrastructure, based on remote method invocation, to simplify parallel programming by abstracting low-level shared-memory or message passing details while maintaining high performance and portability. STAPL, the Standard Template Adaptive Parallel Library, builds upon this infrastructure to make communication transparent to the user. The basic design is discussed, as well as the mechanisms used in the current Pthreads and MPI implementations. Performance comparisons between STAPL and explicit Pthreads or MPI are given on a variety of machines, including an HP-V2200, Origin 3800 and a Linux Cluster.*

1 Motivation

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. However, expressing efficient communication is also one of the most difficult aspects of parallel programming.

There are currently two common models of communication in parallel programming: 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 by standards such as Pthreads [10] and OpenMP [22]. However, its lack of manual data distribution mechanisms can hinder scalability [21]. In addition, many large machines do not implement shared-memory. One solution has been the introduction of software distributed shared-memory (software DSM), which provides a software implementation of a global address space (e.g., [1]).

In message passing, 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). The message passing model is considered harder to program, although it is more portable thanks to the Message Passing Interface standard, MPI-1.1 [16]. Since processes use private address spaces, data distribution must be manually coded, potentially improving scalability. However, because all sends and receives must appear in matched pairs, dynamic or irregular applications can be difficult to express.

One-sided communication is another model that combines some of the strengths of shared-memory and message passing [26]. A set of processes operate using private address spaces as well as sections of logically shared-memory. A process communicates by explicitly *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 [25] and the updated Message Passing Interface, MPI-2 [17].

Remote method invocation (RMI) is another communication model, often associated with Java [11]. RMI works with object-oriented programs, where a process communicates by requesting a method from another 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 [18]). 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 is easy to program, 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 [29], Charm++ [12, 13], Tulip [2], 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.

Recently, we have developed STAPL, a parallel superset to the C++ Standard Template Library, which provides parallel containers and algorithms [24, 23]. The goal of the underlying STAPL communication infrastructure is to simplify parallel programming while maintaining high performance and portability. It utilizes both blocking RMI (to alleviate the need for difficult split-phase execution) and non-blocking RMI (for

* Department of Computer Science, Texas A&M University

high performance) to provide a clean interface for an object-oriented C++ program. Since RMI's are asynchronous (i.e., don't require matching operations as in message passing), synchronization similar to one-sided communication is used.

In this paper we introduce the communication abstractions in STAPL. The communication infrastructure provides an RMI-based interface to abstract the underlying programming model(s) used to implement it. Parallel containers use the infrastructure to provide distributed data structures. Parallel algorithms use the containers to abstract some of the necessary communication. Finally, the user can combine containers and algorithms to express a program, completely unaware of the underlying communication.

2 STAPL Overview

The C++ Standard Template Library (STL) is a collection of generic data structures called *containers* (e.g., vector, list, set, map) and *algorithms* (e.g., copy, find, merge, sort) [28]. To abstract the differences in containers, algorithms are written in terms of *iterators*. An iterator is a generalized pointer that provides operations such as 'advance to next element' or 'dereference current element'. Each container provides a specialized iterator (e.g., a vector provides a random access iterator, whereas a list provides a bi-directional iterator).

The Standard Template Adaptive Parallel Library (STAPL) is a sequentially consistent, parallel superset to STL [24, 23]. STAPL provides a set of parallel containers and parallel algorithms that are abstracted from each other via parallel iterators, named *pContainers*, *pAlgorithms* and *pRanges* respectively. The pContainers provide a shared-memory view of data by internally handling data distribution. The pRange presents an abstract view of a scoped data space, which allows random access to a partition, or subrange, of the data space (e.g., to data in a pContainer). The pAlgorithms use pRanges to operate on data in parallel.

We are designing STAPL to support many different parallel architectures, from small SMP's to massively parallel supercomputers. One goal of STAPL is to allow a single codebase to operate efficiently on these different systems. Some architectures provide efficient shared-memory support, while others are optimized for message passing. Additionally, some systems may most efficiently be programmed by combining shared-memory and message passing into mixed-mode parallelism [6, 4, 27, 20]. For instance, clusters of SMP's can use message passing between nodes in the cluster, and shared-memory within nodes. In support of this, we have created a parallel communication infrastructure that abstracts the issues of shared-memory and message passing programming, allowing for a single interface that may be optimized for specific machines.

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

In both cases, the receiver does not necessarily expect the communication, as in a dynamic 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 collection (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, where all memory is private to a process.
2. *event ordering* - operations to inform a process or processes that computation dependencies are satisfied. This is explicit in shared-memory (e.g., semaphore signal and wait operations), and implicit in message passing, via the semantics of message sending and receiving.

Cleanly expressing these types of communication and synchronization are requirements for a parallel programming model's success. Shared-memory and message passing both fulfill all of these requirements, although in slightly different ways. One goal of the communication infrastructure is to abstract these issues, yielding a clean interface that lends itself to efficient implementation with either model.

4 Case Study: Parallel Sorting

To illustrate how different parallel programming models affect communication, we consider a common parallel algorithm for sorting: sample sort [3]. 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 the following code fragments, which present implementations using explicit shared-memory or message passing.[2] We assume the input has already been generated, and, in the case of message passing, distributed.

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

[2] In general, shared-memory algorithms are sequential until a fork (line 11), whereas message passing algorithms are always in parallel.

```
1  // shared-memory sample sort
2  void sort(int* input, int size) {
3    int p = //...number of threads, 0-p...
4    std::vector<int> splitters( p-1 );
5    std::vector< vector<int> > buckets( p );
6    std::vector<lock> locks( p );
7
8    for( int i=0; i<p-1; i++ )
9      splitters[p-1] = //...sample input...
10
11   //...fork p threads...
12   int id = //...thread id...
13   for(i=size/p*id; i<size/p*(id+1); i++) {
14     int dest = //...appropriate bucket...
15     locks[dest].lock();
16     buckets[dest].push_back( input[i] );
17     locks[dest].unlock();
18   }
19   barrier();
20
21   sort(bucket[id].begin(), bucket[id].end());
22 }
```

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

The shared-memory code uses a shared STL vector to communicate splitters (lines 8-9), as opposed to the message passing library call (lines 8-9). Shared-memory must fork and calculate each thread's local portion (lines 11-13), whereas in message passing data is manually distributed a priori. Shared-memory shares the buckets by locking each insertion to ensure mutual exclusion (lines 15-17), and uses a barrier (line 19) to ensure proper event ordering of distribution and sorting. Because message passing does not know the amount and order of communication (lines 11-14), it must probe for all incoming messages (lines 15-19), which also ensures proper ordering of distribution and sorting.

Neither implementation is optimal. Shared-memory makes extensive use of locking (one lock per element), potentially causing bucket contention. Message passing sends many small messages, potentially causing network congestion. These issues are not intrinsic to the sample sort algorithm, only to the underlying communication model and subsequent implementation. Improvements can be made at the expense of additional lines of code, which are even further removed from the algorithm. For instance, insertions could be buffered into groups before locking or sending, reducing the overall number of locks or sends necessary.

We now contrast shared-memory and message passing code with STAPL. Since 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 the following code fragment. 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 the previous implementations.

```
1  // STAPL sample sort
2  struct p_sort : public stapl::parallel_task {
3    int *input, size;
4    p_sort(int* i, int s) : input(i), size(s) {}
5
6    void execute() {
7      int p = stapl::get_num_nodes();
8      int id = stapl::get_node_id();
9      stapl::pvector<int> globalSplitters( p-1 );
10     stapl::pvector< vector<int> > buckets( p );
11
12     globalSplitters[id] = //...sample input...
13     std::vector<int> splitters( globalSplitters );
14
15     for( i=0; i<size; i++ ) {
16       int dest = //...appropriate bucket...
17       stapl::async_rmi( dest, ...,
18            &stapl::pvector::push_back, input[i] );
19     }
20     stapl::rmi_fence();
21
22     sort(bucket[id].begin(), bucket[id].end());
23   }
24 }
```

As seen by the user, this pAlgorithm hides all underlying communication by appearing as a simple library call. In its implementation, the pContainers abstract some of the underlying communication. For example, the splitter privatization (line 13) makes a copy of the distributed pVector. This copy invokes the pVector copy constructor, which in turn uses the dereference operator to obtain and copy each individual element. As shown in the following fragment for the dereference operator, if the desired element is not local, RMI is used to obtain it (i.e., a question).

```
1  template<class T>
2  T& pvector<T>::operator[](const int index) {
3    if( /*...index is local...*/ )
4      return //...element...
5    else
6      return stapl::sync_rmi( /*owning node*/, ...,
7             &stapl::pvector<T>::operator[], index );
8  }
```

The explicit RMI on line 17 tells the destination bucket to add an element (i.e., a statement).[3] Because the communication infrastructure ensures remotely invoked methods execute atomically, additional code for mutual exclusion is eliminated. However, a fence is used to ensure proper event ordering (line 19).

[3]This communication could also be abstracted from the user using the pVector's `push_back` method. However, since this is a 2D pVector, a dereference is necessary to obtain the proper sub-vector. As defined above, the dereference will first try to return the sub-vector, then apply the `push_back`. We are investigating alternatives to allow the desired behavior.

5 Design

The main goals of the STAPL communication infrastructure are to provide an easy to use, clean means of expressing parallelism in STL-oriented C++ code, while facilitating efficient implementation on many different parallel machines. To be successful, the requirements of Section 3 should also be addressed. Although shared-memory and message passing can both fulfill the requirements, shared-memory is not yet implemented for large machines, and message passing can make writing C++ STL code difficult.[4] As such, we base our communication infrastructure on a higher level RMI abstraction. RMI deals directly with an object's methods, and hence maps cleanly to object-oriented C++. STL code is composed of many container-based objects, and algorithms can easily be written it terms of objects, as seen in our case study. In addition, RMI can be efficiently implemented using either model, as we will show in Section 7.

The communication infrastructure provides task-level parallelism. Each task is a C++ object and is associated with a single process or thread. In message passing, an object resides exclusively in a process's private memory. However, in shared-memory only a conceptual association can be made between threads. To help facilitate this association across possible implementations, we name each unit of execution a *node*. Upon startup, all nodes are initialized and begin executing the same code in parallel, similar to MPI. Nodes independently create local objects and access other nodes' local objects via RMI. We have currently defined two base forms of RMI, which map directly to the two fundamental types of communication.

1. `void async_rmi(destNode, localObjPtr, method, arg1...)` - makes a statement. The call issues the RMI request and returns immediately. Subsequent calls, such as a `rmi_fence`, are used to ensure completion of all requests.
2. `rtn sync_rmi(destNode, localObjPtr, method, arg1...)` - asks a question. The call issues the RMI request and waits for the answer.

The additional information required compared to a regular C++ method invocation is minimal (only `destNode` is extra). Note that `localObjPtr` must be local to the destination node. STAPL is providing a layer on top of the communication infrastructure to perform address translation to facilitate the gathering and usage of such pointers. However, the infrastructure provides an additional collective function, `execute_parallel_task`. Each calling node automatically registers the specified `parallel_task` object and begins execution using its `execute` method. This facilitates parallel algorithms, where RMI's between `parallel_tasks` can simply use the `this` pointer as the `localObjPtr`.

We are also incorporating many-to-one and one-to-many communication into the infrastructure to support common communication patterns. We have currently defined two, based on the fundamental communication types. Currently, both patterns interact with all nodes, although we are refining this interaction to node subsets via a scheme similar to MPI communicators [16]. As development proceeds, we anticipate adding additional calls for increased performance.

1. `void broadcast_rmi(localObjPtr, method, arg1...)` - makes a statement to all nodes. The call issues the RMI requests and returns immediately. Subsequent calls, such as `rmi_fence`, are used to ensure completion of the requests on remote nodes.
2. `void collect_rmi(localObjPtr, method, input, output)` - asks a question on all nodes and collects the results (i.e., a reduction). The call issues the RMI request and waits for the answer. It is a collective operation in that all nodes must execute it before any node proceeds.

5.1 Synchronization

The communication infrastructure also addresses both forms of synchronization. To ensure mutual exclusion, methods invoked by RMI execute atomically. Hence, a method invoked by multiple nodes in parallel preserves thread-safety by acting as a monitor. We recognize that some applications may not require mutual exclusion for all remotely invoked methods, and hence are considering non-atomic RMI for the future.

Event ordering is supported in two ways. An `rmi_wait` operation is provided to allow a node to wait for any RMI to be invoked before proceeding. Additionally, the `rmi_fence` allows nodes to wait until all nodes have arrived and completed all pending communication. This can be used to satisfy flow dependencies in a computation. It also allows for straightforward implementations of master-slave computations, where the slaves wait at the fence while the master signals the work to be performed via RMI.

5.2 Data Transfer

In our scheme, 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. As such, the communication infrastructure 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, the infrastructure 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++ [12, 13]. Specifically, each data member is defined as either local (i.e., automatically allocated on the stack) or dynamic (i.e., explicitly allocated on the heap using `malloc` or `new`). This method can be used as needed to adaptively pack, unpack, or determine the type and size of the class based on the infrastructure's underlying implementation.

[4] For example, MPI requires a type identifier for all messages, which can be difficult to obtain in type-independent C++ templates.

5.3 Integration with STAPL

Figure 1 shows the layout of STAPL's basic components. The communication infrastructure serves as the bottom layer, and abstracts the actual parallel communication model utilized via the RMI interface.

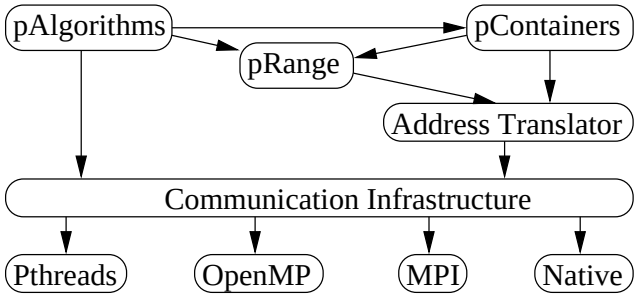

Figure 1: STAPL Layout

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 node in which to actually store data. The contents and location of each of these sub-containers are maintained using the pRange and address translator respectively[5]. The pContainer's main job then is to maintain the consistency of the data it stores as the user invokes various methods on it. Three communication patterns result:

1. *access* - a node needs access to data on another node (e.g., the dereference operation for a vector). The `sync_rmi` handles this pattern.
2. *update* - a node needs to update another node's data (e.g., the insert operation). The `async_rmi` handles this pattern.
3. *group update* - a node 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. A 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 infrastructure. In addition, message passing is a difficult model to implement data structures with, since it requires matching sends and receives. RMI abstracts the underlying message passing to an asynchronous interface, suitable for arbitrary, un-matched insertions and deletions. As such, the implementor can stay focused on the fundamental issues of a distributed data structure.

[5]We are currently defining the address translation operations for the various pContainers. One example is the pVector, which can maintain the range of indices contained in each sub-pVector, along with the corresponding node ID and local pointer.

A pAlgorithm expresses a parallel computation in terms of `parallel_task` objects. These objects generally do not use the communication infrastructure 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 nodes can be maintained using pContainers within the `parallel_task`. As their methods are used to modify and store the results, the pContainers will internally generate the necessary RMI communication.

In the event that pContainers do not offer the necessary methods, RMI communication between `parallel_task`s is facilitated by the automatic address translation described in section 5. Some algorithms require such facilities, such as a depth-first search on a distributed graph. One node begins the search on its local vertexes while the other nodes wait. As the search progresses to remote vertexes, RMI can be used to tell the owning nodes to continue the search on their local data. In addition to this event ordering based communication, two other common 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 node searches its local data for an element. Since multiple nodes may find a match, a reduction is used to combine the results (i.e., node 0's result has precedence over node 1's, etc.). The `collect_rmi` handles this pattern.
2. *bulk communication* - a large number of small messages are needed. A smaller percentage utilize this pattern, with sorting being a common example. The `async_rmi` operations handles this pattern.

As described, each level of STAPL serves to further remove the user from the underlying communication issues. The communication infrastructure provides the fundamental abstraction between shared-memory and message passing. The pContainers build upon this to create distributed data structures. The pAlgorithms generally 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.

6 Implementation

We have currently implemented the communication infrastructure using two different underlying programming models: Pthreads (shared-memory) and MPI-1.1 (message passing). 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 mechanisms used to implement the basic communication infrastructure operations.

6.1 Request Format

Regardless of implementation, all RMI requests are encapsulated internally by a *functor* [9]. A functor stores an RMI's

class, method and argument information, and allows the request to be stored, communicated, and subsequently executed. The base functor class only provides a method to execute itself, whereas derived functors are specialized based on number of arguments and whether the return value is needed. To preserve C++'s strong typing system, all functors make extensive use of C++ templates.

6.2 Request Scheduling

RMI requests do not require matching operations on the destination node. As such, we must introduce mechanisms to schedule the processing of incoming requests. The two issues that must be balanced are ensuring a timely response of incoming requests, which may be blocking the caller, 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 [29, 2, 7, 25]. This approach is successful if polls do not dominate the local computation, but are frequent enough to yield a timely response.
2. *interrupt-driven* - a hardware interrupt is used to notify a node of incoming requests [29, 2, 7, 25, 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, a request is immediately processed. This solution is successful if other threads can execute while the receive is blocking, and control returns to the communication thread soon after the receive completes.
4. *non-blocking communication thread* - a separate communication thread performs a poll for incoming requests, processes any available requests, then yields [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 timeslice is a good balance between computation and polling. Since typical timeslices are 1/10 of a second, this is often a problem.

Our current solution for both implementations is explicit polling. Although communication threads have the potential to yield better performance, creating an efficient implementation is much more involved, and often requires platform dependent knowledge (e.g., thread scheduling policies). Our current focus is on general implementations that can be further optimized for specific platforms. As such, we will consider communication threads in the future.

Both implementations perform polls within communication infrastructure 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 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.

6.3 async_rmi

The `async_rmi` is the most complicated operation to implement because it is completely asynchronous (i.e., it neither waits for a return value nor requires the destination node to expect the operation). In addition, multiple `async_rmi`'s will often be issued at once (e.g., the distribution phase of sample sort). Sending many small messages individually causes traffic and limits bandwidth. A well-known alternative is to buffer messages and issue them in groups, with the extreme case being a single communication at the end of the computation phase of an algorithm. `async_rmi`'s are automatically buffered internally, and issued in groups based on a default or user-defined aggregation factor. Requests are copied into an internal aggregation buffer until the aggregation factor is reached. The buffer is then transfered to the destination node, and each stored request is executed in FIFO order. The appropriate aggregation buffer size can be configured for each machine during installation.

The Pthreads implementation actually utilizes a form of message passing with shared-memory, since the semantics of RMI imply one node telling another node to do something. This message passing is much simpler than standard MPI however, and hence has several opportunities for higher performance. Each node has a request queue, which holds RMI requests from other nodes. Instead of copying the entire request from the origin to the destination, only the request pointer is enqueued. As such, the only buffer copies performed are by the underlying cache system when the destination actually starts traversing the buffer. The request queue has a specific entry for each node, and each entry can hold only one request pointer. This design alleviates the need for expensive lock operations during access. Instead, the owner can check for non-null entries during a poll, and the requester can busy-wait until its specific entry returns to null if it needs to send a second request. We found busy-waiting to perform faster than Pthreads conditionals for event ordering. The request queue can be a bottleneck since it can only hold one request per-node at a time. To alleviate this, we pipeline requests by using two aggregation buffers per possible destination node. While one buffer is enqueued at the destination, the other can be filled.

The MPI implementation is similar to Pthreads, although MPI internally handles the request queue. Each node applies the same pipelined sending scheme as in Pthreads, and uses non-blocking sends to facilitate filling one buffer while the other is sending. Since the implementation is not multithreaded, only one incoming request can be processed at a time. We use a single non-blocking receive from any node, which allows good implementations of MPI to overlap the communication with computation. The receive is posted and the computation started. Polls simply check to see if the receive completed, in which case the request is processed. MPI must copy incoming messages into a user-defined buffer. We allocate this buffer to be large enough for all possible communications a priori. We chose this static allocation scheme

versus an on-demand, dynamic allocation scheme for performance.

6.4 sync_rmi

The `sync_rmi` operation builds upon the foundation of `async_rmi`. Requests are sent in the same fashion, and the sender blocks until the return value is returned. The block performs continuous polling, enabling quick response of return messages. However, request scheduling on the destination node becomes even more of a concern for `sync_rmi`, since it does block computation until the return value is obtained. In some instances, the algorithm requires this. For others, we are planning a non-blocking version that returns an opaque handle. When the return value is actually needed, the handle can be queried.

The Pthreads implementation uses a separate response queue, similar to its request queue, for the computing node to enqueue the return value. The MPI implementation uses the exact same facilities as `async_rmi`, and differentiates between requests and responses by the message tag, contained in the message's header.

6.5 rmi_fence

The `rmi_fence` is a collective operation similar to a barrier. It does not release until all nodes arrive and complete all outstanding communication requests. There are two complicating issues for a fence versus a barrier. First, to ensure correct execution, nodes 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 [2]. As such, we were forced to implement our own fence. 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 node, performs a distributed summation, and declares termination as soon as the sum equals zero.

We implemented two different fences for Pthreads, a centralized barrier with sense reversal [5] and the tree-based barrier proposed as Algorithm 11 in [15]. Both fences busy-wait, where each iteration of the busy-wait performs a poll. It was straightforward to incorporate the termination detection sum within both fence's arrival protocol.

We implemented a tree-based barrier for MPI, where nodes notify their parents upon arrival, wait for a release message while polling, then propagate the release to their children. This communication pattern can easily include the termination detection sum within the arrival/release messages. We currently have implemented three different tree patterns: a flat tree with a root and all other nodes as leaves, a standard binary tree (0's children are 1 and 2, 1's children are 3 and 4, etc.), and a binary tree optimized for a hypercube.

7 Performance

We tested our two communication infrastructure implementations (Pthreads and MPI) on a number of different machines, including a Hewlett Packard V2200, an SGI Origin 3800, and a Linux cluster. 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, arranged with 4 processors per node of the hypercube. The *cluster* consists of 5 distributed memory nodes connected with a private 1Gb/s Ethernet switch. Each node contains 2 1GHz or 1.1GHz Pentium III processors with 256KB or 512KB L2 caches.

7.1 async_rmi and sync_rmi

We tested the latency of STAPL versus explicit shared-memory or message passing code using a ping-pong benchmark. One node sends a message, and upon receipt, the receiver immediately sends a reply. We measured the time between issuing the ping and receiving the pong. STAPL 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 1. Since STAPL is implemented on top of Pthreads or MPI, it necessarily adds overhead. For Pthreads, the increase is noticeable, due to the overhead of an RMI request being buffered, enqueued at the destination, then subsequently processed, whereas the explicit Pthreads code simply performs an update. Both codes preserve ordering through busy-waiting.

For MPI, the overhead is typically negligible, with the one exception being the V2200. Upon investigation, we identified that HP's MPI incurs extra startup overhead for using nonblocking sends and receives versus their blocking counterparts. However, as the number of processors increase, the nonblocking versions tend to perform better. The HP receive also appears to take time proportional to the expected size. Since STAPL uses a size large enough to hold the largest possible message, it takes more time to receive than the explicitly written MPI code, which never overestimates.

Although some codes require only a few isolated communications, others require many. As described in Section 6.3, the communication infrastructure buffers `async_rmi` requests internally based on an aggregation factor to optimize message transfer size and minimize network traffic. To measure tolerance to this traffic, we re-timed the ping-pong benchmark using multiple consecutive pings before a single pong. STAPL's aggregation factor was varied from 8 to 512 messages.

Figures 2 and 3 show the results for MPI and Pthreads on the O3800. For Pthreads, STAPL is faster after 1000 pings, yielding a 70% improvement with an optimal aggregation buffer of 128 messages (4KB). Similarly, the V2200 yields a 16% improvement with a 256 message buffer (8KB). The variance between aggregation factors is smaller than with MPI. This is

	V2200		O3800		Cluster	
	Explicit	STAPL	Explicit	STAPL	Explicit	STAPL
Pthreads	15	22/22	4	16/15	N/A	N/A
MPI	16	35/37	17	17/21	197	197/201

Table 1: Latency (us) of explicit communication and STAPL (async_rmi/sync_rmi).

because the cost of message transfer in the Pthreads implementation is low compared to the cost of sending an MPI message. As such, less overhead can be amortized with increased buffering.

For MPI, STAPL is significantly faster after just 10 messages, yielding a 6.4-fold improvement for 10,000 pings, with an optimal aggregation buffer of just 128 messages (4KB). Similarly, the V2200 yields a 2.5-fold improvement with a 32 message buffer (1KB). On the cluster, where communication is especially expensive, a 2100-fold improvement with a 512 message buffer (16KB) is possible. Our findings confirm that multiple small messages should be avoided when using message passing. Since the communication infrastructure performs such optimizations internally, the user is able to focus on simply expressing their algorithm.

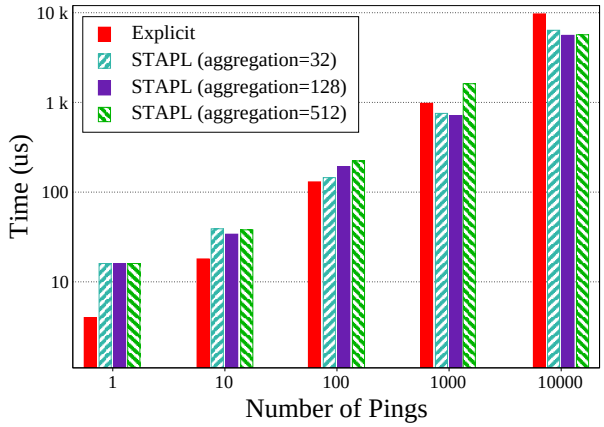

Figure 2: Pthreads (log-log scale)

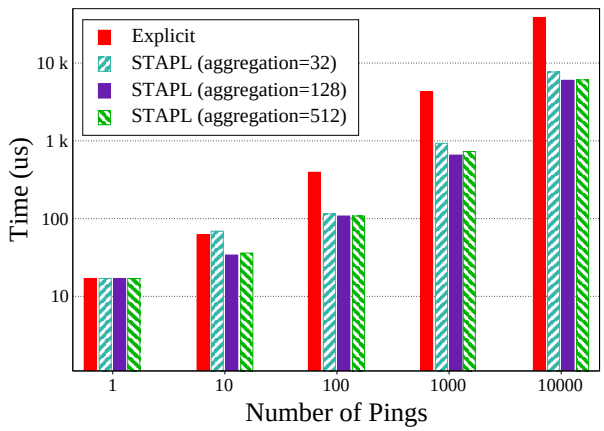

Figure 3: MPI (log-log scale)

7.2 rmi_fence

We measured the overhead of `rmi_fence` against vendor-optimized barriers. For message passing, we compared against `MPI_Barrier`. For shared-memory, we compared against the OpenMP barrier directive when available (O3800), and the MPI barrier when not (V2200). For Pthreads, we found STAPL's tree-based fence to perform best. For MPI, we found STAPL's hypercube-tree barrier to perform best on the V2200 and the O3800. The flat-tree performed best on the cluster, since message transfer latency is much higher than message startup costs.

Figure 4 show the results for shared-memory. Since STAPL uses a platform independent fence, which continues to poll for RMI requests while waiting for termination, we expect it to incur some overhead. Although typically much less, we have found the overhead of polling and termination detection to be as high as 23% for STAPL's Pthreads fence. Since this does not account for all the time difference shown in Figure 4, we attribute the rest of the overhead to our platform independent implementation.

Figure 5 shows the results for message passing. STAPL scales competitively on the O3800 and cluster, with only a slight increase in overhead, due to polling and termination detection. The V2200 is the exception, where the nearly flat curve implies HP's `MPI_Barrier` implementation uses shared-memory optimizations that are not available to STAPL. These results confirm the findings of [2], which demonstrate the utility of pollable, instead of just blocking, barriers for use in libraries such as STAPL.

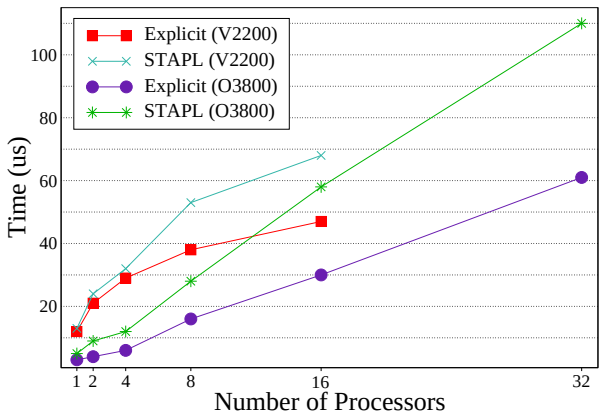

Figure 4: Shared-Memory Fence (log-log scale)

7.3 Algorithm Performance

We have implemented several parallel algorithms using the communication infrastructure, such as our case study, sample sort. We compared our 30-line RMI-based code against a

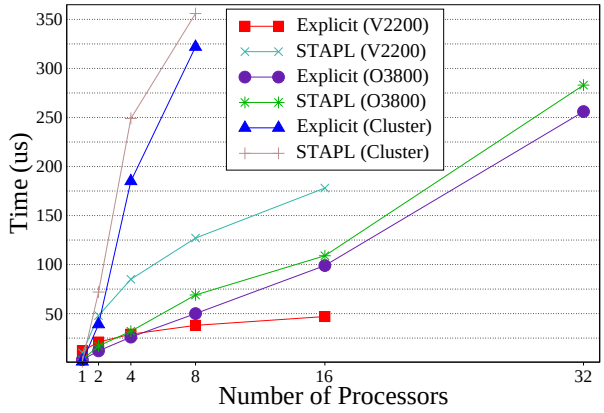

Figure 5: Message Passing Fence (log-log scale)

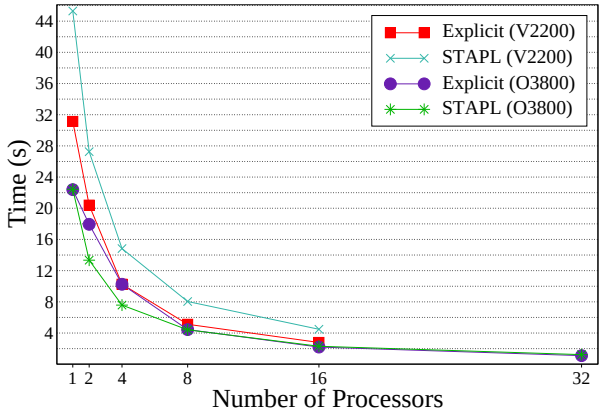

Figure 6: Time to sort 1M integers

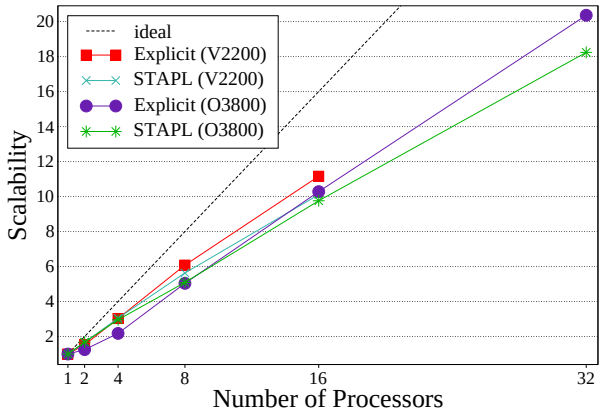

Figure 7: Scalability of sorting 1M integers

100-line hand-tuned shared-memory implementation that uses Pthreads for communication on the V2200 and OpenMP on the O3800. This hand-tuned version is significantly optimized compared to the shared-memory case study described in Section 4. Specifically, it buffers elements locally, then performs a merge before the final sorting phase. The merge is receiver-driven, such that each processor copies its portion contained in the other processors' buffer to a local array. This method has the benefit of filling each processors' cache just before beginning the sorting phase.

We tested sample sort using both STAPL communication infrastructure implementations. We found the Pthreads implementation performed best on the V2200, as expected, since it is a shared-memory machine. On the O3800, we found the MPI implementation performed best. Our experience on the O3800 shows that OpenMP outperforms Pthreads for shared-memory communication, and hence are pursuing an OpenMP implementation as well.

Figure 6 shows the wall clock times for both machines. STAPL requires more time to perform the sort, largely due to `async_rmi`'s need to construct request objects with each call. This effect is not nearly so pronounced on the O3800 however, and we attribute this to the Origin's faster, superscalar processors, which may be utilized more effectively with the extra work.

Figure 7 shows the scalability versus running on one processor. STAPL scales very competitively, and at times even out-scales the hand-tuned codes. This is because communication is occurring incrementally throughout the computation, instead of one large merge at the end. However, as the number of processors increase, the per-processor data sets get nearer to the cache size. Since the hand-tuned codes' merge brings data into cache, their scalability improves. Note the especially pronounced improvement from 4-8 processors on the O3800, during which the data does completely fit into its 8MB cache for the first time.

We have also implemented a parallel version of the STL inner product algorithm. Each processor uses the sequential algorithm to compute its local contribution, then applies a global reduction to combine the results. STAPL uses `collect_rmi` to perform the reduction. Explicit MPI uses `MPI_Allreduce`. On the V2200, explicit shared-memory uses Pthreads, and a sequential summation. On the O3800, explicit shared-memory uses OpenMP and the parallel reduction directive. All experiments calculate the inner product of a 40 million element vector with itself.

As shown in Tables 2 and 3, there is negligible difference between explicit shared-memory, message passing and STAPL. This is expected, since the majority of STAPL's overhead is the time to package the reduction into an RMI request, which occurs only once per reduction. We attribute the minor variances in time, such that STAPL is occasionally faster than the explicit versions, to machine fluctuations because the differences are not statistically significant.

Number of Processors	V2200		O3800	
	Explicit	STAPL	Explicit	STAPL
1	10.658	10.218	2.123	2.290
2	5.329	5.287	1.086	1.159
4	2.665	2.736	0.544	0.578
8	1.309	1.355	0.287	0.294
16	0.667	0.705	0.146	0.159
32			0.076	0.075

Table 2: Time (s) to compute the inner product of 40M element vectors using shared memory.

Number of Processors	V2200		O3800		Cluster	
	Explicit	STAPL	Explicit	STAPL	Explicit	STAPL
1	10.008	10.558	2.301	2.300	.964	.964
2	4.996	5.149	1.165	1.164	.577	.574
4	2.678	2.567	0.583	0.581	.285	.284
8	1.301	1.326	0.343	0.330	.142	.142
16	0.752	0.702	0.171	0.170		
32			0.088	0.084		

Table 3: Time (s) to compute the inner product of 40M element vectors using message passing.

8 Conclusions and Future Work

We have developed a parallel communication infrastructure that abstracts the details of the underlying communication models, allowing the parallel programmer to focus on cleanly expressing their algorithm. STAPL builds upon this abstraction using pContainers and pAlgorithms to make communication transparent to the user. Preliminary results show low overhead and scalable performance on a wide variety of machines.

We are actively pursuing additional implementations of the infrastructure, including OpenMP and a mixed-mode version using MPI and OpenMP. Since STAPL is making the transition to exclusively using the primitives for communication, we are currently in the integration phase. This will produce nearly one hundred pAlgorithms and pContainer methods using the infrastructure, enabling us to continue to tune its performance.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[2] P. Beckman and D. Gannon. Tulip: A portable run-time system for object-parallel systems. In *International Parallel Processing Symposium (IPPS)*, 1996.

[3] 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 *Symposium on Parallel Algorithms and Architectures (SPAA)*, 1991.

[4] 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), 1999.

[5] D. Culler and J. Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Freeman Publishers, Inc., 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 *High Performance Networking and Computing Conference (Supercomputing)*, 2000.

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

[10] IEEE. *Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application: Program Interface (API) [C Langauge]*, 1996. 9945-1:1996 (ISO/IEC) [IEEE/ANSI Std 1003.1 1996 Edition], IEEE Standard Press, ISBN 1-55937-573-6.

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

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

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

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

[15] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

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

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

[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 (RTSPP) of the International Parallel Processing Symposium (IPPS)*, 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 Computer 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 *International Conference on Supercomputing (ICS)*, 2001.

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

[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 *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2001.

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

[25] 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 *International Parallel Processing Symposium (IPPS)*, 1998.

[26] H. Shan and J. P. Singh. A comparison of MPI, SHMEM and cache-coherent shared address space programming models on the SGI origin2000. In *International Conference on Supercomputing (ICS)*, 1999.

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

[28] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Pub Co, 2000.

[29] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *International Symposium on Computer Architecture (ISCA)*, 1992.

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