MRPC: A High Performance RPC System for MPMD Parallel Computing

Chi-Chao Chang, Grzegorz Czajkowski, and Thorsten von Eicken

Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

MRPC is an RPC system that is designed and optimized for MPMD parallel computing. Existing systems based on standard RPC incur an unnecessarily high cost when used on high-performance multi-computers, limiting the appeal of RPC-based languages in the parallel computing community. MRPC combines the efficient control and data transfer provided by Active Messages (AM) with a minimal multithreaded runtime system that extends AM with the features required to support MPMD. This approach introduces only the necessary runtime RPC overheads for an MPMD environment. MRPC has been integrated into Compositional C++ (CC++), a parallel extension of C++ that offers an MPMD programming model. Basic RPC performance in MRPC is within a factor of two from those of Split-C, a highly tuned SPMD language, and other messaging layers. CC++ applications perform within a factor of two to six from comparable Split-C versions, which represent an order of magnitude improvement over previous CC++ implementations.

Keywords: remote procedure call, active messages, multithreading, MPMD, parallel computing

1 Introduction

Remote procedure call (RPC) [5] is widely used in distributed systems as the primary communication abstraction. In its most general form, an RPC specifies the data that is to be transferred and the remote operation that is to be performed with the data. Using a simple procedure call abstraction, the RPC initiator calls into a local stub, which marshals and transfers the data to the remote address space through a standard communication channel (e.g. pipes, streams, or TCP). A remote stub unmarshals the data and transfers control to a new thread that will execute the specified operation to assimilate the data. The result of the operation is sent back to the caller's address space through stubs, which then resumes computation. A RPC system typically consists of an IDL compiler for stub generation and a runtime system that interfaces with the operating system to perform data and control transfer.

Over the last decade, RPC has been extensively studied and optimized in operating systems. The focus gradually moved from the original inter-machine RPC [30] to local RPC [3,4,12,22] in which the role of the kernel is minimized during cross-domain calls on uniprocessor and shared-memory multiprocessor machines. The performance of RPC on such systems is well understood, and RPC is now fully incorporated into commercial operating systems such as Solaris or Windows NT.

In parallel-computing environments, RPC is not nearly as predominant. Application writers and language designers tend to choose communication mechanisms closely matched to the programming and execution model provided by the underlying machine. Typically, message passing and distributed shared memory abstractions are used in a Single-Program-Multiple-Data (SPMD) model, where a fixed number of identical programs operate on their local data and communicate with one another at well defined points in time.

The research on Active Messages (AM) [34] has shown that a simplified RPC can be implemented effectively in SPMD environments. AM takes advantage of the SPMD assumptions as well as of specialized hardware, allowing it, for example, to refer to remote procedures by their entry point addresses and to pass pointers to message buffers directly to user-level network interfaces. As a result, AM has simpler semantics than traditional RPC, and achieves performance close to that of raw hardware. It has been ported to a variety of multi-computers [9,26,29] and used in the implementation of SPMD parallel languages such as Split-C [11] and CRL [18].

Applications that benefit from the generality of a Multiple-Program-Multiple-Data (MPMD) model cannot use AM directly and instead require a full-blown RPC system. MPMD allows for multiple programs to dynamically create concurrently executing tasks that communicate with one another at any point in time. The advantages of the MPMD style are:

- it is well suited for applications that exhibit irregular or unknown communication patterns, or that can benefit from a "client-server" type of setting;
- it is more appropriate for meta-computing in a heterogeneous, large-scale environments because MPMD programs are a lot more loosely-coupled than those written in the SPMD style;
- and it encourages programmers to treat parallel computing software as components, promoting code re-use and the ability to compose programs [8].

As an alternative to RPC, lower-level messaging layers such as MPI [35] and PVM [32] can be used for MPMD programming and often achieve good performance. However, these systems generally sacrifice the elegance (and IDL support) of an RPC system, and require that the receiver know when to expect incoming communication, limiting the range of MPMD applications that can be expressed conveniently.

To date, implementations of RPC for MPMD parallel programming have used runtime systems (e.g. Nexus [14], Legion [17]) that implement RPC on top of the above messaging layers and add some combination of dynamic task creation, load balancing, global name space, concurrency, and heterogeneity. These implementations suffer from overheads that are prohibitively high for multi-computers, mainly for two reasons:

1. the semantics of RPC requires crossing program domains, asynchronously detecting incoming communication, spawning of new threads, and marshalling of data, all of which introduce significant overheads; and
2. current RPC implementations are designed for portability and support for a variety messaging layers and communication protocols, which are important in a distributed computing setting, but less important within a parallel machine.

As a result, the communication overheads of MPMD RPC are usually an order of magnitude higher than those found in highly tuned SPMD systems on multi-computers. This problem has naturally limited the appeal of languages based on the MPMD model in the parallel programming community.

In this paper, we exploit a different approach in providing RPC for parallel computing. We start from AM, a simple SPMD implementation of RPC, and design a minimal runtime system that extends this implementation with the features required for MPMD. The goal is to introduce only the RPC runtime overheads necessary for an MPMD environment in an attempt to isolate the basic costs of RPC-based communication on multi-computers. While a design starting with the raw hardware has the potential for better performance, our approach allows us to show that the potential gains would be minimal by comparing the SPMD and MPMD communication costs.

The viability of this approach is demonstrated by designing and implementing MRPC, a high-performance RPC system for parallel computing on distributed memory multi-computers. MRPC is fully integrated into Compositional C++ (CC++), a parallel extension of C++, from which it borrows the RPC stub generation and a global name space. The compiler front-end translates all global pointer accesses into sender stubs and wraps global method entry points with receiver stubs. MRPC provides a minimal runtime system that performs method name lookup, communication, marshalling, and buffer management, and includes a customized threads package.

MRPC is evaluated using a series of micro-benchmarks and three applications written in CC++ on the IBM SP. The benchmark performance is compared to that obtained by standard messaging layers and by Split-C, an SPMD language that is layered directly on top of AM. MRPC achieves a round-trip latency of 87 µs for a null RPC, which is 32 µs more than the base latency of AM but equal to those of IBM MPL and various MPI implementations. At the language level, the basic remote data access operations in CC++ can be optimized to within a factor of 2 from Split-C's. As a result, three CC++ applications perform within a factor of two to six from comparable Split-C versions. This represents an order of magnitude improvement over previous CC++ implementations.

This paper makes the following contributions:

- It discusses the major issues in designing an RPC system for MPMD parallel programming on a homogeneous, distributed memory multi-computer.

- It describes an efficient, lean implementation of an RPC system that is based on the CC++ language, uses AM, and achieves performance comparable to native messaging layers and a highly tuned SPMD language (Split-C), with an order of magnitude improvement over previous CC++ implementations.
- It quantifies the fundamental overheads of MRPC over SPMD RPC (namely AM) using micro-benchmarks and three application benchmarks that are written in both CC++ and Split-C.

The rest of this paper is organized as follows. Section 2 motivates the need for a high-performance RPC system for parallel computing by contrasting the requirements of this setting to the goals of RPC research over the last decade. It also states the design goals of MRPC and the role of Active Messages towards achieving our goals. Section 3 discusses the issues in RPC-based communication on a distributed memory multi-computer. Section 4 summarizes the basic send/receive mechanisms in Active Messages on the IBM SP2, describes the implementation of MRPC and shows how some of the issues presented in Section 3 are resolved. Section 5 describes the experimental setup used to evaluate the MRPC in the context of CC++. Section 6 presents the results of our experiments. Section 7 discusses related work and Section 8 concludes the paper.

2 Background

2.1 A Case for RPC Specialization

Although RPC systems have been studied extensively over the last decade, the use of RPC on MPMD multi-computers requires new techniques to be developed to achieve satisfactory performance. Just as it became necessary to optimize RPC for the local case in conventional operating systems, RPC should be specialized for high performance within a parallel machine.

Initial research on RPC focused on inter-machine calls on conventional workstations connected by a regular network (e.g. Ethernet). Key features were reliability, security, and the ability to handle a variety of argument types and to support for multiple transport layers over local or wide-area networks. The overhead introduced by these requirements is well understood and thoroughly reported by Schroeder and Burrows in the context of the DEC Firefly OS [30]. In the late 80's, the main research was on tuning the RPC implementation for best performance across the network. In the early 90's, focus shifted from cross-machine RPC to cross-domain, or local RPC. Bershad *et. al.* [3] argued that in micro-kernel operating systems RPC calls occur predominantly between different protection domains (i.e. processes) within the same machine. Lightweight RPC (LRPC) [3] was motivated by this observation and specializes RPC for the local case by reducing the role of the kernel without compromising safety.

Parallel computing on a network of workstations as well as on shared and distributed memory multi-computers spurred the development of many runtime systems and communication libraries based on RPC. For example, Nexus and Legion are both highly portable runtime systems that support a variety of user-level communication protocols and have been used as a compiler target for parallel languages such as CC++ [8], Fortran-M [13] and Mentat [16]. Both of these systems are aimed for wide-area, large-scale parallel computing.

These languages and systems implement RPC on top of user-level messaging layers such as MPI, PVM, or even TCP sockets. While the implementation is tuned for performance, maintaining portability, flexibility, and heterogeneity is not questioned. Although these are important features to strive for in a distributed setting, they often introduce unnecessary overhead in parallel applications running on homogeneous multi-computers, which is undoubtedly the common case. These overheads introduce a substantial performance gap between such systems and highly tuned SPMD systems on those same machines, limiting the appeal of the RPC-based languages in the parallel computing community.

In a sense, our goal is analogous to that of LRPC: to develop an RPC system for parallel computing that is simple and specialized for the common case.

2.2 Design Goals

We strive to maintain two characteristics of traditional RPC systems that are relevant in a homogeneous, high-performance setting:

- Transparency: From the programmer's point of view, a local RPC is indistinguishable from a remote one, which is facilitated by proxy stubs generated by an IDL compiler. Likewise, RPC in parallel computing should be transparent. The goal is not only to provide a stand-alone runtime system but to integrate it into a language that

provides automatic stub generation. MRPC has been integrated into Compositional C++ (CC++), a parallel extension of C++. We chose CC++ because it offers a clean abstraction for multiple programs sharing a single global name space.

- Simple Data and Control Transfer: LRPC uses a single kernel thread during control transfer, and a shared kernel stack between the caller and the callee for data transfer. On shared-memory multiprocessor machines, the User-Level RPC (URPC) [4] improves LRPC further by transferring the control between caller and callee through user-level threads and the data through shared memory, relying on the kernel only for processor allocation. On a multi-computer, data is transferred through communication channels across the interconnection network, and control is transferred by calling a function (possibly on a separate thread) upon the arrival of the message.

2.3 Why Active Messages?

The design proposed here builds on top of AM for three reasons. First, and most importantly, AM provides an efficient mechanism for transferring data and control on SPMD multi-computers. The caller specifies a handler (in the form of a pointer to a function) that is to be invoked upon arrival of the message, and data that is to be passed as argument. The handler is invoked by AM on the remote node and typically integrates the data into the remote computation or performs a simple remote service and sends a response back.

Second, starting from AM we can study and quantify the minimal runtime overhead necessary to extend a simple SPMD RPC to an MPMD RPC. In the context of a single language (e.g. CC++), it would be possible to perform whole-program analysis and specialized code generation in order to eliminate some of these overheads. We do not pursue this possibility, however, in order to retain language independence and in order to first identify and quantify the potential gains of such compiler optimizations.

Finally, the proposed system can be easily ported to a variety of MPPs on which implementations of AM based on a standard API exist. These implementations usually achieve a performance close to that of the raw hardware while hiding sophisticated architectural features of different kinds of network interfaces in state-of-the-art MPPs. AM have been ported to the CM-5 [34], Intel Paragon [21], Cray T3D [21], Meiko CS-2 [29], and IBM SP-2 [9].

3 RPC Design Issues

Traditional RPC introduces a number of issues into the communication layer that are absent from AM: method names must be resolved to entry point addresses, potentially complex arguments must be marshaled, and multiple threads of control must be supported. In addition, the modularity and higher levels of abstraction offered by an MPMD system that uses RPC require a more local view of the interactions of communication and computation than in SPMD ones, which affects how the arrival of messages is detected and how atomic actions are implemented.

The issues discussed in the section arise due to the semantics of RPC, and as a result, communication in MPMD systems is inherently more expensive than in SPMD ones.

3.1 Method Name Resolution

An MPMD application can be composed of multiple, separately compiled program images. This means that the compiler cannot generally determine the existence or location of a remote procedure statically. Instead the mapping from the procedure name to its entry point address must be made at runtime, requiring either extra round-trip inquiry messages or the transmission of the name instead of its address in messages. In contrast, an SPMD system handles only a single program image and can assume that remote code is located at the same addresses as on the local node.

3.2 Argument Marshalling

The arguments of an RPC can be arbitrary objects, requiring the compiler to generate stubs that serialize each argument into the outgoing message buffer and that extract the arguments or the return value on message reception. This flexibility makes RPC more powerful than the global memory access primitives in SPMD systems, which only support a “shallow copy” of user-defined data types. On the other hand, this flexibility adds one extra copying of the data on each end as well as the overhead of the serialization code. The compiler can inline this code in simple cases, but in other cases (especially in the presence inheritance) a full dynamic method invocation is required. Some systems [6,25] attempt to reduce the cost of RPC by restricting the types that can be marshaled or by only supporting shallow copies of RPC arguments. This issue can be particularly harmful to a heterogeneous system as different machines have different data representations.

3.3 Data Transfer

AM achieves efficient data transfer by requiring the sender to specify a remote buffer address where the data will be placed, which avoids dynamic memory allocation on message reception. For RPC, the compiler-generated stub allocates a receive buffer from which data will be unmarshaled, but the sender does not find out about this buffer until runtime. So, at first, data must be shipped to a generic buffer and then copied into the receive buffer. This incurs two additional copies (one for the arguments and another for the return value) into RPC besides the two required for data marshalling.

3.4 Control Transfer

AM imposes some restrictions on the handlers that are invoked upon message arrival. They are expected to integrate the data into the local computation as fast as possible and are not allowed to allocate memory on the heap, initiate a new AM request, or perform a blocking operation. RPC requires that control be transferred to a new thread at the receiving end because no restrictions are placed on the operations performed in remotely invoked procedures. In particular, a procedure may block on a lock held by the interrupted computation, requiring the latter to proceed before the former can complete. This requires that each program have multiple threads and that the reception of a message, at least logically, create a new thread.

The need for multiple threads adds a significant overhead in RPC. Besides the cost of context switching, it requires judicious introduction of locks into the runtime and communication layer to maintain thread-safety. The overhead of thread scheduling can become significant when handling a large number of RPC messages. One benefit of multithreading is the ability to hide the communication latency, but its effectiveness depends on the relative costs of the thread operations (creation, context switching, and synchronization) with respect to the latency.

Most threaded SPMD systems [7,10] minimize the threading costs by making threads run to completion to eliminate context switches, by performing custom stack management (e.g. using stacklets [15], spaghetti stacks), or by reducing synchronization costs through custom code generation [15]. Others like Split-C take an even more radical approach — offering only a single computation thread — and rely on split-phase remote accesses to tolerate latencies.

Optimistic Active Messages (OAM) [36] augments AM with threads, removing some of the restrictions in AM handlers. To implement a fast RPC, OAM optimistically executes the handler code on the stack; the handler is aborted and restarted on a separate thread if it blocks (i.e. similar ideas are used in the Concert runtime system [28] and ABCL/f [33]). OAM still assumes an SPMD model and does not specifically address the communication bottlenecks when that assumption is no longer valid.

3.5 Message Reception and Dispatch

A critical component of the communication latency is the queuing delay incurred by messages at the receiving end before they are serviced. In an SPMD system, where communication phases can be planned globally, it is feasible to require the programmer (or the compiler) to introduce explicit `poll` operations to check for message arrival. Polling is generally very cheap and can yield low latencies if executed often enough.

The more modular programming style promoted by MPMD generally favor an interrupt-driven message reception. The software interrupt generated on message arrival is propagated to the application's runtime system, which creates a new thread to handle the message. However, the overheads of the interrupt and of the kernel layers propagating it to the application are often significant, increasing the overall communication cost.

4 The Implementation of MRPC

The MRPC system consists of a runtime library that performs communication, marshalling of basic data types, method caching, and remote program execution. The current implementation runs on the IBM SP multi-computer and consists of 4028 lines of C++ code and 1881 lines of C code. The communication module is layered on top of Active Messages and uses a custom, non-preemptive threads package.

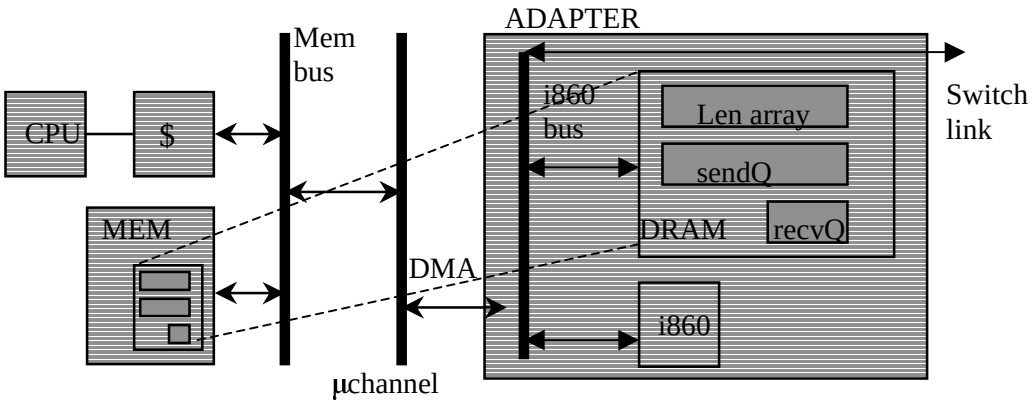

Figure 1. Schematic of the SP2 network adapter. Send/receive data structures are memory-mapped.

4.1 Active Messages on the SP2

The IBM SP2 is a distributed memory MPP consisting of Power2 RS6000 nodes[1] interconnected by a custom network fabric via network adapters that contain an Intel i860 microprocessor with 8 MB of DRAM. The implementation of AM on the SP2 (SP-AM) relies on the network adapter firmware but does not use any IBM software on the Power2 processors.

The adapter firmware allows one user process per node to access a set of memory-mapped *send* and *receive* queues directly (Figure 1). These queues allow user-level communication layers to access the network directly without any operating system intervention: the adapter monitors the send queue and delivers messages into the receive queue, all using DMA for the actual data transfer. The send queue has 128 entries while the receive queue has 64 entries per active processing node. Each entry consists of 256 bytes and corresponds to a *packet*. A memory-mapped *packet length array* is associated with the send queue. Its slots correspond to entries in the send queue and indicate the number of bytes to be transferred for each packet. The adapter monitors this array (located in the adapter memory) and transmits a packet from the send queue when the corresponding slot in the array becomes non-zero.

A packet is sent by placing the data into the next entry of the *send* queue along with some header information. Since the RS6000 memory bus does not support cache coherency, the relevant cache lines must be flushed to the main memory explicitly. The transfer size is stored in the packet length array. To receive a packet, the data in the top entry of the receive queue is copied to the user buffers.

SP-AM provides reliable, ordered delivery of messages. Because packets can still be lost due to overflows in the receive queue, SP-AM uses custom flow control with fast re-transmission of lost packets based on packet sequent numbers.

4.2 MRPC

MRPC has been integrated into Compositional C++ (CC++), a parallel extension of C++ designed for the development of task-parallel object-oriented programs. CC++ uses processor objects to abstract the different address spaces in an MPMD application. It provides a global name space across processor objects through global pointers, and parallel control structures that allow blocks of code to be executed concurrently. A regular C++ class can be elevated to a processor object, making all its public methods and data accessible by other processor objects using global pointers. The CC++ system consists of a front-end translator and a back-end compiler. The front-end translates the CC++ extensions into pure C++ code and the back-end compiler is an off-the-shelf C++ compiler. The release of CC++ (version 0.4) uses the Nexus runtime system (version 3.0)

[1] The SP2 used in this paper has “thin” processing nodes: 66 MHz, peak performance of 266 MFLOPS, 64 KB data cache with 64-byte lines, 64 MB of RAM, SPEC ratings of 144.3 SPECint92 and 205.3 SPECfp92.

```
1  void main () {
2   // Declare Worker processor object
3   proc_t wkr("worker", "node#1");
4    // Create Worker and get a global pointer
5   Worker *global gp = new (wkr) Worker(1);
6   // RPC
7   gp->DoWork(123);
8  }
```

Figure 2. Master source code in CC++.

```
1  global class Worker {
2  public:
3    Worker(int id);  // Constructor
4    DoWork(int arg); // Exported call
5  }
6
7  Worker::Worker(int id) { . . . }
8  Worker::DoWork(int arg) { . . . }
```

Figure 3. Worker processor object in CC++.

After compilation, a CC++ program consists of an executable (which contains the main entry point) and possibly a number of processor object executables. At first, the master node (typically processing node 0) starts the main executable and the other nodes remain idle waiting for an incoming request to dynamically load processor objects.

The following subsections discuss RPC binding and stub generation in the context of a simple example consisting of a master program that spawns a worker program to which it performs RPC calls. The CC++ source code for both programs are shown in Figure 2 and Figure 3. The corresponding C++ code produced by the CC++ compiler is shown in Figure 4 and Figure 5. Subsection 0 describes the steps during a basic, unoptimized RPC call before presenting the optimizations.

Binding

MRPC binding relies on the global name space provided by CC++ through global pointers that consist of a processing node number and a local address on that node. As illustrated in Figure 2 and Figure 3, the master program running on processing node 0 creates an instance of the `Worker` processor object on node 1. The compiler translates the remote instantiation into an RPC to the Worker constructor stub (Figure 4, lines 8-12) that returns a global pointer to the new `Worker`. The master code can thereafter invoke `Worker` public methods remotely by referencing them using the global pointer.

Stub Generation

The compiler translates all global pointer accesses into *caller* stubs (Figure 4, lines 8-12 and 14-17) and wraps global method entry points with *callee* stubs (Figure 5, lines 11-19). It generates an instance of a bi-directional RPC endpoint (`mrpc_endpt`) for every RPC occurrence, as seen in lines 2 and 3 of Figure 4 and line 13 of Figure 5.

```
1   void cCpP_main() {
2     mrpc_EndPt endpt1;
3     mrpc_EndPt endpt2;
4     cCpP_GlobalPtr cCpP_gptmp;
5
6     proc_t wrkr("worker", 1);
7     // Inlined stub for processor object binding
8     cCpP_GlobalPtr gp = (cCpP_gptmp = mrpc_new_pobj(wrkr, "Worker"), // alloc gp
9       endpt1.InitRPC(cCpP_gptmp, "WorkerCons"), // sets up RPC
10      endpt1.SendRPC(),                         // initiates RPC and wait for return value
11      endpt1 >> cCpP_gptmp,                     // unmarshals return value
12      cCpP_gptmp);                              // and assigns return value to gp
13    // Inlined stub for calling DoWork
14    (endpt2.InitRPC(gp, "DoWork"),
15     endpt2 << 123,
16     endpt2.SendRPC(),
17     endpt2.Reset()
18    );
19 }
```

Figure 4. Master code in C++ generated by CC++ front-end translator. The remote instantiation of the Worker processor object is translated into a RPC. Note the generated caller stub code for both RPC occurrences.

```
1   class Worker {
2   public:
3     Worker(int id);
4     DoWork(int arg);
5     cCpP_entry_Worker(void *cCpP_obj, mrpc_buffer_t *cCpP_inbuf);
6     cCpP_entry_DoWork(void *cCpP_obj, mrpc_buffer_t *cCpP_inbuf);
7   }
8   Worker::Worker(int id) { . . . }
9   Worker::DoWork (int arg) { . . . }
10
11  Worker::cCpP_entry_Worker(void *cCpP_obj, mrpc_buffer_t *inbuf) {
12    int cCpP_rflag;
13    mrpc_EndPt endpt;
14    endpt.RecvRPC(inbuf, cCpP_rflag); // grab RPC info
15    {
16      int cCpP_arg1;
17      endpt >> cCpP_arg1; // unmarshal argument
18      cCpP_GlobalPtr cCpP_retval = new class Worker(cCpP_arg1); // call procedure
19      if (cCpP_rflag) { endpt << cCpP_retval; } // marshals return value
20    }
21    if (cCpP_rflag) { endpt.ReplyRPC(); } // send reply back
22  }
23  static cCpP_RegisterHandler cCpP_rh_Worker("WorkerCons",
24                                             &Worker::cCpP_entry_Worker);
```

Figure 5. Worker code in C++ generated by CC++ front-end translator. The compiler generates a callee stub for every public method in Worker, and registers them at runtime initialization (through the construction of static objects of type `cCpP_RegisterHandler`). Only the callee stub for the `Worker` constructor is shown here -- the stub for `DoWork` is very similar.

Each `mrpc_endpt` holds send (S-) and receive buffers (R-buffers) and implements the RPC runtime interface, namely `InitRPC`, `SendRPC`, `RecvRPC`, `ReplyRPC`, and `Reset`.

Calling

The following are the steps of a basic RPC in MRPC:

1. The *caller* stub (Figure 4) initiates the RPC by calling into `InitRPC` (lines 9 and 11) which
   2. allocates and initializes the S- and R-buffers,
   3. copies the method name into the S-buffer,
   4. copies the local address part of the destination global pointer into the S-buffer, and
   5. copies the address of the caller endpoint into the S-buffer, to be used for reply.
6. It marshals the arguments into the S-buffer (line 15).
7. It calls `SendRPC` (lines 10 and 16) which issues an AM bulk transfer of the S-buffer to a static, per-node buffer, and blocks the thread.
8. Upon arrival of the message, AM invokes a *dispatch handler*, which
   9. extracts the name of the method from the static buffer and looks up its entry point address,
   10. extracts the local address of the global pointer from the static buffer,
   11. creates an R-buffer and copies the rest of the data from the static buffer into the R-buffer, and
   12. creates and schedules a new thread to execute the *callee* stub, passing the local processor object address and the R-buffer as arguments.

13. The *callee* stub (Figure 5) calls `RecvRPC` (line 14) which attaches the R-buffer to the callee endpoint and moves the address of the caller's endpoint to the S-buffer, in preparation for a reply.
14. It unmarshals the arguments from the R-buffer (line 17).
15. It calls the remote procedure (line 18).
16. It marshals the return value (if any) into the S-buffer (line 19).
17. It calls `ReplyRPC` (line 20), which issues an AM reply message that transfers the contents of the S-buffer into a static, per-node, caller buffer.
18. Upon arrival of the reply message, AM invokes a *return handler*, which
    19. extracts the address of the caller endpoint,
    20. copies the rest of the data from the static buffer into the caller endpoint's R-buffer, and
    21. unblocks the caller's thread.
22. When the thread is resumed (back to Figure 4), the caller stub unmarshals the return value out of the R-buffer if necessary (line 11).
23. It calls `Reset` (line 17) only if the compiler will reuse the endpoint later.

It is important to note that the dispatch and return handlers run atomically with respect to incoming messages and safely copy the data from the static buffer area to R-buffers. This eliminates the need for the caller to issue an extra round-trip message to determine the receive buffer, but still requires two data copies on each side.

In the MRPC implementation, the basic RPC described above is optimized as follows.

Method Stub Caching

The entry point addresses for remote method stubs are resolved using a look-up into a local hash table (add step 2.1). Each processing node maintains a table of stub addresses which is indexed by processor number and method name hash value. During runtime initialization, local method stubs are registered into the table (as seen in Figure 5), and remote entries are marked as invalid.

The caller uses the processor number (taken from the global pointer) and the method hash value to index into the table. If the entry is valid, the stub address is fetched and passed to the remote node in the message, eliminating the need for step 9. If the entry is invalid, the entire method name is passed in the message[2] (as done in step 3) and the resolution occurs at the remote end (step 9). The stub address is piggybacked with the RPC reply (add to step 17), updating the caller's local table entry. The main benefit of resolving the name on the caller side whenever possible is to reduce the size of the messages.

Method stub caching effectively solves the name-mapping performance problem between different address spaces. This technique can be extended to a scenario where multiple programs execute on the same processing node by introducing the program ID as another index to the hash table.

Fast Copying and Persistent Buffers

To reduce the marshalling overheads, MRPC pre-allocates the S- and R-buffers[3] and provides fast data copying routines. R-buffers for recently invoked methods are kept allocated so they can be managed by the sender. Initially, for a "cold" RPC, arguments are marshaled into the S-buffer and transferred to a per-node static buffer area at the receiver's end. The dispatch handler allocates a new R-buffer, copies the data from the static buffer area into the R-buffer (as in step 11), and marks it as attached to the method being called (add step 11.1). The address of the R-buffer is returned with the stub table update message. Subsequent cached invocations will copy data directly into the per-node R-buffer associated with the remote method, eliminating one extra data copy on the callee side.

---

[2] It is tempting to simply send the hash value to the callee instead of the entire method name. This doesn't work because of possible collisions in the hash table.

[3] The size of the S- and R-buffers is a compile-time constant (currently 16 Kbytes). It is not difficult to implement a data transmission protocol (as the one documented in [9]) to handle data transfers of arbitrary sizes.

To eliminate a data copy on the caller side, the caller can allocate an R-buffer and pass its address into the RPC message (add step 5.1). The callee can ship the return value directly into this buffer. This has not been incorporated into the current implementation as it requires a slight modification in the compiler.

Message Reception and Scheduling

Due to the high cost of software interrupts on message arrival on the IBM SP, message reception is based on polling that occurs on a node every time a message is sent [9]. In order to avoid deadlocks when there is no runnable thread, MRPC forks a polling thread one each processing node at initialization. The scheduling policy of runnable threads is FIFO.

5 Experiments

The experimental setup consists of a series of CC++ and Split-C communication micro-benchmarks and four applications: EM3D, Water, FFT, and Blocked LU Decomposition[4]. The AM layer and the threads package have been carefully instrumented to account for the number, types, and sizes of message transfers as well as the number of threads, context switches, and synchronization operations. All experiments run on an IBM SP with an AIX 3.2.5 operating system. Although the languages use different back-end compilers (CC++ uses IBM C++ and Split-C uses gcc), the performance of the FP kernel in all three applications is virtually the same for both compilers. We chose relatively small data sets for these applications to minimize the effects of data cache in our results.

5.1 Micro-benchmarks

Figure 6 and Figure 7 show the micro-benchmarks used:

- Several variations on Ping-Pong measure the r/t time of a null RPC (calling a null method of a remote object referenced by a global pointer and waiting for its completion): *0-Word Simple* (no thread switches at the sender

```
// Split-C definitions
double lx;
double *global gpY;
double lA[20];
void *global gpA;



// 0-Word N/A

// 1-Word N/A

// 2-Word N/A

// 0-Word Atomic RPC
atomic(foo, 0);
// GP 2-Word Read
lx = *gpY;
// GP 2-Word Write
*gpY = lx;
// Bulk Read
bulk_read(&lA,gpA,20*sizeof(double));
// Bulk Write
bulk_write(gpA,&lA,20*sizeof(double));
```

Figure 6. Split-C Micro-benchmarks Pseudo-Code

```
// CC++ definitions
double lx;
double *global gpY;
ARRAYOFDOUBLE lA(20);
ARRAYOFDOUBLE *global gpA;
OBJ *global gpObj;
int ly, lz;

// 0-Word RMI
gpObj->foo();
// 1-Word RMI
gpObj->foo(ly);
// 2-Word RMI
gpObj->foo(ly, lz);
// 0-Word Atomic RMI
gpObj->atomic_foo();
// GP 2-Word Read
lx = *gpY;
// GP 2-Word Write
*gpY = lx;
// Bulk Read
lA = gpObj->get(gpA);
// Bulk Write
gpObj->put(lA, gpA);
```

Figure 7. CC++ Micro-benchmarks Pseudo-Code

[4] The CC++ version of these applications is heavily based on the original Split-C implementations [23] to allow for a fair comparison.

nor receiver), *0-Word*, *1-Word*, *2-Word* (each with a thread switch at the caller side only), *0-Word Threaded* [5] (thread switches at both caller and callee sides), and *0-Word Atomic* (*0-Word Threaded* with the method executed atomically).

- Remote access to a 64-bit `double` through a global pointer (*GP Read/Write*), which consists of an RPC with a return value (`double`).
- Bulk transfer of an array of 20 `double`s using a CC++ RPC and Split-C bulk reads and writes (*Bulk-Read/Write*).

5.2 EM3D

EM3D is a parallel application that simulates electromagnetic wave propagation [11,24]. The main data structure is a distributed graph. Half of its nodes represent values of an electric field (E) at selected points in space, and the other corresponds to values of the magnetic field (H). The graph is bipartite: no two nodes of the same type (e.g. E or H) are adjacent. Each of the processors has the same number of nodes, and each node has the same number of neighbors. Computation consists of a sequence of identical steps: each processor updates values of its local H- and E-nodes as a weighed sum of their neighbors.

Three versions of EM3D in CC++ and Split-C are compared here by varying the percentage of adjacent nodes that are located on remote processors. Each version uses a different method to transfer data. The first version (*em3d-base*) dereferences a global pointer to a remote node each time the value is needed. Since some co-located graph nodes may share remote neighbors, introducing local ghost nodes, which represent remote graph elements can eliminate redundant global accesses. This simple form of caching is used in *em3d-ghost*, where first the values of all ghost nodes are fetched and the main computation loop is purely local. This version can be further optimized by aggregating all ghost nodes being transferred from one processor to another. *Em3d-bulk* uses this optimization to issue bulk transfers instead of many individual fetches.

The benchmark runs shown in this paper uses a synthetic graph of 800 nodes distributed across 4 and 8 processors where each node has degree 20 for a total of 4000 edges. The fraction of edges that cross processor boundaries is varied from 10% to 100% in order to change the computation to communication ratio.

5.3 Water

Water is an N-body molecular dynamics application taken from the SPLASH benchmark suite [31] that computes the forces and energies of a system of water molecules. The computation iterates over a number of steps, and each of which involves computing the intra- and inter-molecular forces for molecules contained in a "cubical" box, which runs in $O(N^2)$ time. A predictor-corrector method is used to integrate motion of water molecules over time. The total potential energy is calculated as the sum of intra- and inter-molecular potentials. The main data structure is an array of molecules distributed statically across all processors. The intra-molecule interactions are computed locally, whereas the inter-molecule ones require reads and writes of remote data.

Two versions of Water written in CC++ and Split-C are compared. The base version (*water-atomic*) issues atomic reads and writes to access and update the remote molecules. The optimized version (*water-prefetch*) replaces the atomic read requests with selective prefetching, where selected data of remote molecules are bundled and fetched from their respective processors prior to local computing. Both versions are run with a data size of 512 molecules.

5.4 FFT

This benchmark computes the *n*-input butterfly algorithm for the discrete one-dimensional FFT problem using *P* processors. The algorithm is divided into three phases: (i) $log(n) - log(P)$ local FFT computation steps using a cyclic layout where the first row of the butterfly is assigned to processor 1, the second to processor 2, and so on; (ii) a data re-mapping phase towards a blocked layout where the n/P rows are placed on the first processor, the next n/P rows on the second processor, and so on; and (iii) $log(P)$ local FFT computation steps using the blocked layout. In the first and third phases, each processor is responsible for transforming n/P elements.

[5] *0-Word Threaded* is the default RPC variation generated by the current CC++ compiler and is used in all the applications. We present the other variations to provide the reader with more insight into the costs of RPC.

In both the Split-C and CC++ versions, each processor allocates a single *n/P*-element vector to represent its portion of the butterfly. Communication occurs only in the data re-mapping phase where each processor uses bulk communication to send a n/P^2-element chunk of data to each remote processor. The communication is staggered to avoid hot spots at the destination. Both versions are run with a data size of 1 million points.

5.5 Blocked LU Decomposition

This application implements LU factorization of a dense matrix as described in the SPLASH benchmark suite [31]. The matrix is divided into blocks distributed among processors. Every step comprises three sub-steps: first, the pivot block *(i,i)* is factored by its owner; second, all processors which have blocks in the *i*-th row or *i*-th column obtain the updated pivot block; third, all internal blocks are updated. All remote blocks requested in a given sub-step need to be fetched since they were modified in preceding sub-steps.

The base Split-C version (*sc-lu*) uses one-way stores for explicitly transferring pivot blocks and prefetches all blocks before beginning the third sub-step. In the CC++ version (*cc-lu*), the one-way stores and prefetches are replaced by RMIs. The input is a 512x512 matrix of `double`s with a block size of 16x16.

6 Results

6.1 Micro-benchmarks

Table 8 shows the results of the micro-benchmarks. The r/t time of a *0-Word Simple* is only 12 μs slower than that of an AM request (one word). The cost of a *0-Word Threaded* raises to 87 μs due to a couple of thread switches. Other variations of the null RPC scale according to the number of thread operations involved. Because of method stub caching, the method lookup cost is about 1 μs and is accounted for in MRPC's *Runtime* overhead. Although the marshalling of basic types introduces a negligible cost in the *CC++ Runtime*, the data is sent using AM bulk transfer primitives, which incurs an additional 15 μs overhead (as seen in *1-Word*, *2-Word*). This overhead is avoided in *GP Read/Write* since accesses to simple data types through global pointers are optimized using small request/reply AM. The cost of marshalling becomes significant for the array of 20 doubles. Bulk reads cost more than bulk writes in CC++ because the return data has to be copied twice: once from the static buffer to the receive buffer, and again from the receive buffer to the CC++ object. This cost would be eliminated if the initiator of a bulk read passed an R-buffer address where the return value would be stored, as described earlier.

Table 9 compares the MRPC null r/t latency to AM and other messaging layers. The performance of the MRPC primitives used through CC++ is competitive with other messaging layers. This result is encouraging since it

Benchmarks	*CC++/MRPC*							*Split-C*			*Nexus*			
	Total (us)	*AM (us)*	*Threads*				*Runtime (us)*	***Total (us)***	*AM (us)*	*Runtime (us)*	***Total (us)***	*MPL (us)*	*DCE Threads (us)*	*Runtime (us)*
			Time (us)	*Yield*	*Create*	*Sync*								
0-Word Simple	**67**	55	4	0	0	10	8	-	-	-	**158**	88	0	70
0-Word	**77**	55	12	1	0	15	10	-	-	-	-	-	-	-
1-Word	**94**	70	12	1	0	15	12	-	-	-	-	-	-	-
2-Word	**95**	70	12	1	0	15	13	-	-	-	-	-	-	-
0-Word Threaded	**87**	55	21	2	1	10	11	-	-	-	**240**	88	~50	~102
0-Word Atomic	**88**	55	21	2	1	14	12	**56**	53	3	-	-	-	-
GP 2-Word R/W	**92**	55	21	2	1	10	16	**57**	53	4	-	-	-	-
BulkWrite 40-Word	**154**	70	21	2	1	10	63	**74**	70	4	-	-	-	-
BulkRead 40-Word	**177**	70	21	2	1	10	86	**75**	70	5	-	-	-	-

Table 8. Micro-benchmark results. The *Total* time reported for each test was obtained by averaging over 10000 iterations. The standard deviation is about 1 μs. In CC++, *Total* is the sum of the messaging layer (*AM*), the *Threads Time*, and the *Runtime*. *Threads Time* is estimated by multiplying the number of thread calls per iteration with their respective costs (4.6 μs for thread creation, 5.8 μs for a context switch, and 0.4 μs for a lock, unlock, or condition variable signal call). In Split-C, *Total* is just *AM* plus *Runtime*. The DCE threads time in Nexus is just an estimate since Foster *et. al.* [14] only report the total overhead on top of MPL.

Communication Layer	*R/T Latency (us)*
AM 1-word req/rep	51
MRPC null-RPC	87
IBM MPL (AIX 3.2.5) send/recv	88
IBM MPI-F (AIX 3.2.5) send/recv	88
MPICH/AM send/recv	88
Nexus 3.0 RPC	240

Table 9. Basic r/t latencies of various messaging layers, in μs.

demonstrates that a traditional RPC system with minimal compiler support (besides the generation of stubs) can be implemented on top of AM as efficiently as a message passing system.

6.2 Applications

Table 10 compares the absolute execution times of a subset of the benchmark runs on 4 and 8 processors using Split-C, CC++/MRPC and CC++/Nexus[6]. For most of the benchmarks (except for *em3d-bulk*), the performance difference between applications running with Split-C and with CC++/Nexus is about an order of magnitude. Applications that use CC++/MRPC performs within the order of magnitude from those of Split-C, closing the performance gap substantially. CC++/MRPC numbers seem to scale reasonably from 4 to 8 processors. Nevertheless, for applications with intensive bulk-data transfers and low computation-to-communication rations such as FFT and LU, the numbers suggest that scalability could be an issue for a larger number of processing nodes.

Applications	*Absolute Execution Time (seconds)*					
	SC4	*NX4*	*CC4*	*SC8*	*NX8*	*CC8*
em3d-gp	68	4743 (70x)	136 (2x)	69.4	4610 (66x)	114 (1.6x)
em3d-bulk	0.26	2.26 (9x)	0.29 (1.1x)	0.16	2.89 (18x)	0.18 (1.1x)
water-atomic 512	1.79	59.2 (33x)	10.0 (5.6x)	1.14	39.0 (34x)	5.54 (4.9x)
water-prefetch 512	1.40	21.1 (15x)	4.89 (3.5x)	0.75	12.3 (16x)	2.58 (3.4x)
fft 1M	1.62	39.5 (24x)	3.36 (2.1x)	0.78	38.4 (49x)	1.80 (2.3x)
lu	0.81	-	2.91 (3.6x)	0.51	-	2.55 (5x)

Table 10. Absolute execution times of a subset of the benchmark runs using Split-C (*SC*), CC++/Nexus (*NX*) and CC++/MRPC (*CC*) on 4 and 8 processors, in seconds. The numbers in parenthesis are the performance ratios between the *NX* and *CC* times to *SC*.

To better identify the costs that contribute to the remaining gap, the execution times for each of the applications running on CC++/MRPC are broken down into a *CPU* component for (mostly) integer or FP computations, a *net* component for communication latencies and overheads, a *thread mgmt* component for thread creations and context switches, a *thread sync* component for locks and signals (but excluding the waiting time, which is accounted for in *net*), and a *marsh+copy* component for stub invocations and data marshalling. For Split-C applications, the communication overheads and latencies are accounted for in *net*, and the remaining goes in *CPU*.

Figure 11 shows the per-edge EM3D performance. CC++ is competitive with Split-C for each of the versions. In *em3d-base*, the big difference between CC++ and Split-C for low remote edge percentages is due to the overhead of accesses to local data through global pointers. As the percentage of remote edges increases, the relative performance of CC++ converges to about a factor of 2 of Split-C. In *em3d-ghost*, the number of global accesses is much smaller than in *em3d-base* and the relative performance converges quickly to 2.5 as the number of remote edge increases.

[6] The CC++ compiler v0.4 with Nexus v3.0 is configured with the TCP/IP communication protocol running over the SP2 high-performance switch. Due to technical problems, we have been unable to configure the compiler to use IBM MPL.

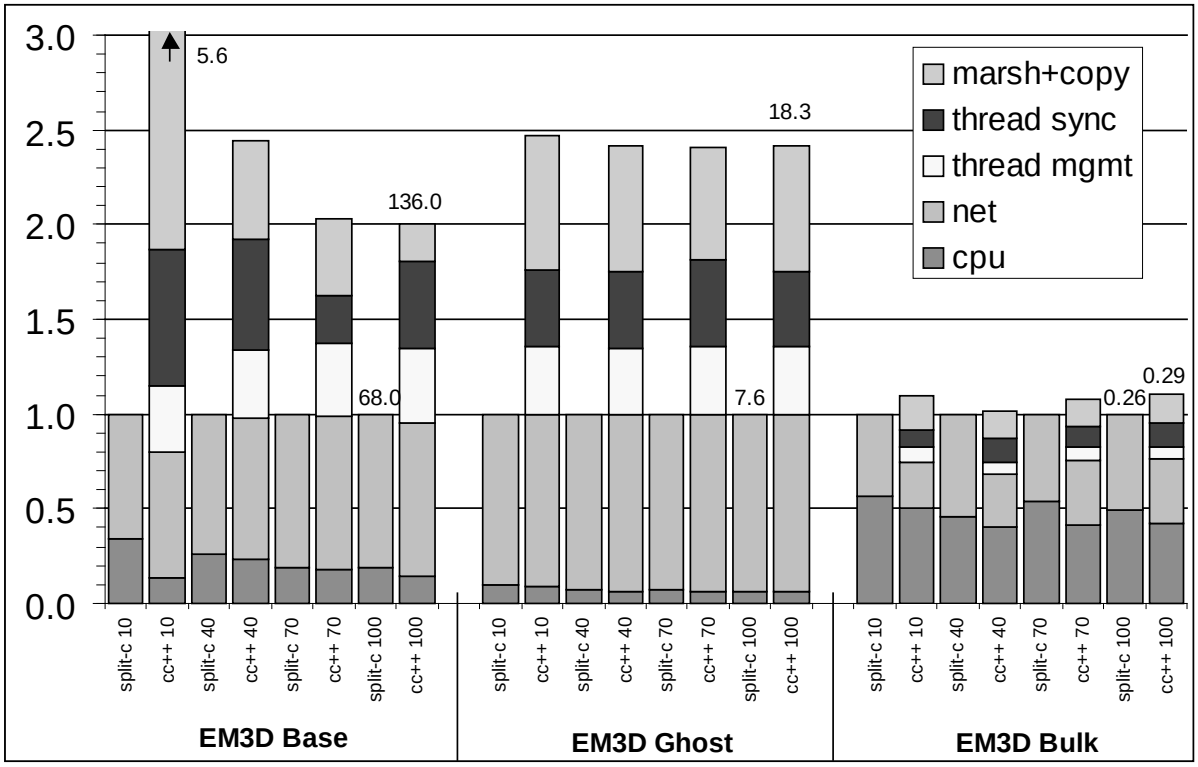

Figure 11. Breakdown of EM3D per-edge execution times for 10%, 40%, 70%, and 100% of remote edges on 4 processors, normalized against Split-C. The absolute execution time (in seconds) for 100% of remote edges is indicated above the respective bars.

Even though bulk transfers require an additional copy in CC++, no significant performance difference is observed in *em3d-bulk*. This is because the total number of bytes transferred per edge is small (about 5 bytes). To really observe a significant hit, the problem size has to be increased by a factor of about 200. It is important to notice that the optimizations used in all three versions of EM3D benefit Split-C and CC++ equally. For 100% remote edges, *em3d-ghost* reduces the execution time of *em3d-base* by 87-89%, and *em3d-bulk* reduces that of *em3d-ghost* by more than 95% for both languages.

Figure 12 shows the performance of the main computation loop in Water with an input of 512 molecules. *Water-atomic* uses small messages to read from and write to the remote molecules. The performance gap between CC++ and Split-C is about 5.6 on 4 processors and 4.8 on 8 processors. *Water-atomic* can be further optimized by replacing the atomic read requests with selective prefetching (*water-prefetch*). This technique causes a 10-fold reduction in remote accesses, thus yielding a 21% improvement to the Split-C version and 51% to the CC++/MRPC version on 4 processors. On 8 processors, the performance improvement is 53% for CC++/MRPC, closing the gap to about 3.4, and 68% for Split-C, indicating that the Split-C version scales better. In Water *marsh+copy* component dominates the other components by far.

In *fft*, the gap between the Split-C and CC++/MRPC versions is about 2.1 on 4 processors and slightly worse on 8. The local computation time is the same in both – the difference is mostly due to data marshalling. In lu, the gap of 3.6 is also due to the additional data copying during matrix block transfers (33% of the gap on 4 processors but 48% on 8) and to synchronization (48% on 4 but only 28% on 8). The *net* time in the CC++/MRPC version of *lu* is about 2x higher than in the Split-C one, mostly due to busy waiting (polling).

6.3 Discussion

CC++ applications perform within a factor of two to six of Split-C. The thread costs show that multithreading in RPC accounts for 25-50% of the performance gap towards AM. Synchronization incurs a significant amount of overhead: from 14% (of the performance gap) in *water-atomic* and 19% in *em3d-ghost* to as high as 48% in *lu*. 98-

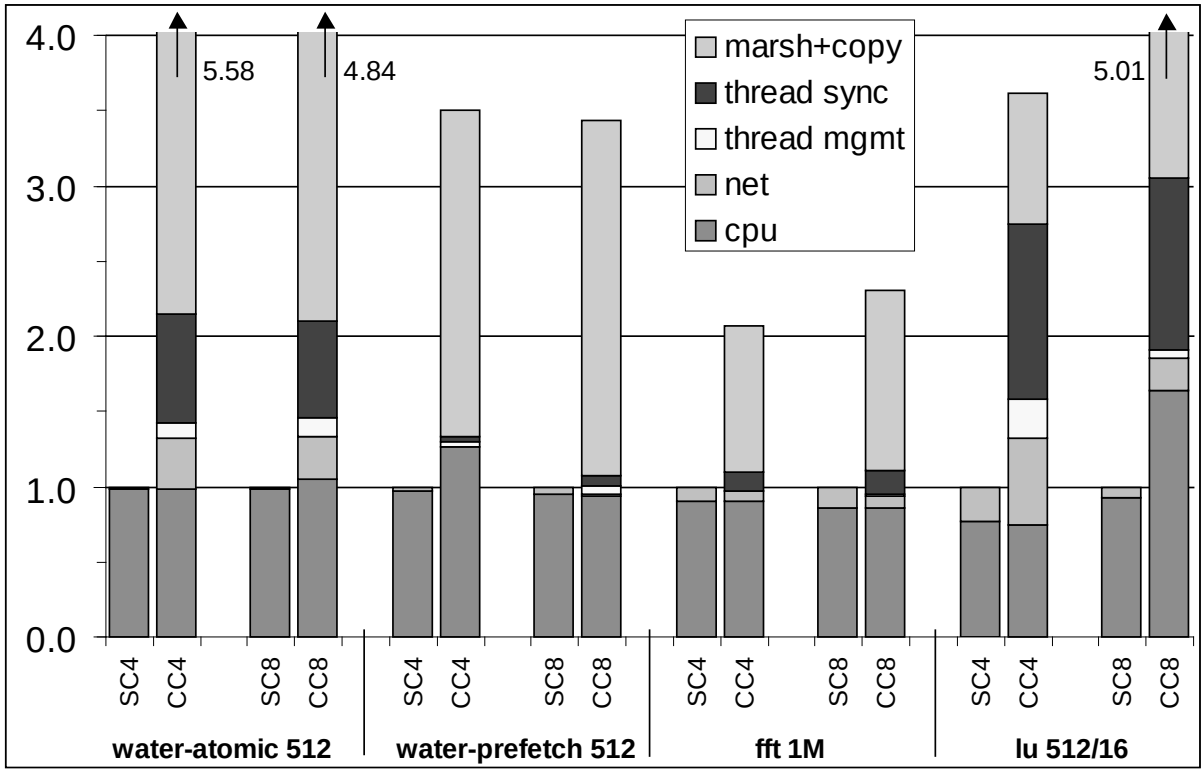

Figure 12. Breakdown of two versions of Water, one version of FFT and LU absolute execution times on 4 and 8 processors, normalized against Split-C.

99% of this overhead is to ensure consistency of shared data and thread-safety in the runtime and communication layers. This is exacerbated by the observation that about 95% of lock acquisitions are contention-less. The cost of thread management is acceptable (between 10-15% in CC++ applications), but not insignificant. This underscores the need for a highly tuned threads package as the costs would be prohibitively high if a more heavyweight or preemptive threads package were used. 75-85% of this cost is due to context switches, a large fraction of which can be attributed to the polling thread. This overhead may be alleviated in the future by reducing the cost of software interrupts, which eliminates the need for the polling thread. The overhead of method name translation is negligible due to the stub caching. Data copying overheads are only substantial when large amounts of data are transferred and when the communication-to-computation ration is high, especially as in FFT and LU.

We believe that these results can be generalized to different MPP architectures. The network times will depend on the overhead and round-trip latency of the underlying AM implementations. As all the other overheads depend on the CPU speed, we anticipate that the trend towards faster processing nodes relative to the network performance will most likely reduce the performance gap between SPMD and MPMD.

7 Related Work

MRPC is closely related to a variety of systems ranging from runtime systems for parallel, object-oriented languages to messaging and RPC systems. A direct comparison of these systems is difficult because differences in hardware and software platforms yield different design trade-offs. However, implementers of RPC-based systems have come across similar issues discussed in this paper. We summarize here three RPC-based systems for parallel computing that achieve performance comparable to MRPC.

7.1 Concert

The Illinois Concert system consists of compiler and runtime support for concurrent object-oriented languages. The compiler performs a number of optimizations based on static, global information for good sequential performance.

For superior parallel speedups, it relies on optimizations that enhance data locality using dynamic pointer alignment and object caching, that support efficient multi-threading through a stack-heap execution model and dynamic load balancing, and that provide efficient communication.

The performance of RPC using Concert is documented in the context of two languages: Concurrent Aggregates (CA) [19] and ICC++ [20]. The round-trip time of a 2-word RPC in CA running on the CM-5 with 33 MHz Sparc1 nodes is 28.6 μs or 941 instruction cycles, about 32% higher than the cost of 2-byte array transfer using the CM-5 Active Messages. Most of the overhead is due to activation frame creation, frame scheduling, and method name translation. In the case of ICC++ running on the Cray T3D (150 MHz DEC Alpha 21064 nodes) over Fast Messages, the round-trip latency of a 4-word RPC is reported at 17.9 μs, of which 29% (5.2 μs) is due to the send and receive overheads in Fast Messages, and 31% (5.5 μs) is due to the Concert runtime system.

7.2 ABCL

ABCL is a parallel language that adopts a programming model based on concurrent objects that have their own thread of control and are message driven [37]. It has been implemented on an EM-4, a multi-computer with special hardware support for packet-driven data-flow architectures, and on an AP1000, a multi-computer with 25 MHz Sparc processors. Without specialized hardware support, remote method invocation on AP1000 requires some additional software overhead for message-buffer management, object scheduling and method lookup. Because of the execution model of ABCL, the semantics of a remote method invocation is much simpler than MRPC's: (i) method invocations are one-way, asynchronous operations, (ii) there is no data marshalling since data is always located in registers or in the object's stack (data still needs to be copied to/from the system message buffers), and (iii) because each object has its own thread of control, message dispatch amounts to scheduling the callee object and looking up the method in the object's virtual method table.

The authors report a round-trip RPC latency of about 18 μs, or 160 instruction counts. The send overhead (request and reply) accounts for 40 instructions, and the receiver consumes a total of 60 instructions for polling, extracting the message and managing data buffers, and dispatching the method.

7.3 Orca and Fast Messages

Orca is a parallel language that supports coherent caching of shared-objects [2]. It is implemented on top of Panda, a communication library that provides RPC and totally ordered multicast at user-level. Panda has been recently ported to run on a network of eight 50 MHz Sparc workstations using a modified version of Fast Messages (in order to support software interrupt and multicast) over Myrinet [1].

The performance optimizations used in Panda include user-level threads that are tightly integrated with message reception (a full context-switch is only required if the handler blocks) and the elimination of buffer fragmentation since it is already provided by FM. The authors report an RPC round-trip latency in Panda of 171 μs, 79 μs (or 85%) higher than a round-trip latency of FM. In addition, the overheads introduced by the Orca language add another 157 μs, increasing the total cost of an Orca remote method invocation to 328 μs (about 3.6x the basic round-trip latency of FM).

8 Conclusion

MRPC is an RPC system that is designed and optimized for MPMD parallel computing. Existing systems based on RPC incur an unnecessarily high cost when used on high performance multi-computers, limiting the appeal of RPC-based languages in the parallel computing community. MRPC combines the efficient data and control and data transfer provided by AM with a minimal multithreaded runtime system that complements AM when the SPMD assumptions are dropped.

Results show that basic remote data access operations in CC++ can be optimized to within a factor of two from Split-C and native messaging layers. This is encouraging since it demonstrates that a traditional RPC system with minimal compiler support (besides the generation of stubs) can be implemented on top of AM as efficiently as a highly tuned message passing system. As a result, our CC++ applications perform within a factor of two to six from Split-C, and with an order of magnitude improvement over previous CC++ implementations. In general, a large fraction of the remaining gap is due to thread synchronization (15-30%), which is necessary for maintaining thread-safe communication, and to RPC overheads in MRPC (over 50%) such as stub invocations, data marshalling, and

message dispatch. Thread management operations such as creation and context switching account for less than 15% of the gap.

The overheads of MRPC over AM indicate two major opportunities for further optimizations at the compiler level. First, the data transfer could be streamlined if the compiler can pre-allocate storage required for data marshalling. Second, the thread management cost could be reduced through code generation that uses special calling conventions to implement cactus stacks, stacklets [18], or other innovative call stacks. However, our application-benchmark results suggest that future efforts should be aimed at eliminating the data marshalling overhead whenever possible, as in some applications it accounted for more than 50% (and as high as 90%) of the performance difference between CC++ and Split-C.

In summary, the performance achieved with MRPC suggests that the MPMD model based on RPC is reasonable for high performance applications running on multi-computers. In many cases, the simplicity of the RPC abstraction along with the software-engineering benefits such as modularity and program composition outweigh the performance gap between MPMD and SPMD. We also believe that the inevitable trend towards faster processing nodes relative to the network performance on MPPs will further reduce that gap.

9 Acknowledgements

We would like to thank Carl Kesselman (ISI/USC) for his assistance with the CC++ compiler, Vijay Karamcheti (NYU) for providing us the null method invocation numbers of ICC++, and the anonymous reviewers for their valuable comments. This work has been sponsored by IBM under the joint project agreement 11-2691-A and University Agreement MHVU5851, and by NSF under contracts CDA-9024600 and ASC-8902827. Chi-Chao Chang is supported in part by a doctoral fellowship (200812/94-7) from CNPq/Brazil.

10 References

1. H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, and K. Verstoep. Performance of a High-Level Parallel Language on a High-Speed Network. In *Technical Report IR-400*, Vrije Universiteit, Amsterdam, April 1996.

2. H. Bal, M. F. Kaashoek, and A. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. In *IEEE Transactions on Software Engineering*, 18(3):190-205, March 1992.

3. B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight Remote Procedure Call. In *ACM Transactions on Computer Systems (TOCS)*, 8(1):37-55, February 1990.

4. B. Bershad, T. Anderson, E. Lazowska, and H. Levy. User-Level Interprocess Communication for Shared Memory Multiprocessors. In *ACM Transactions on Computer Systems (TOCS)*, 9(2):175-198, May 1991.

5. A. Birrel and G. Nelson. Implementing Remote Procedure Calls. In *ACM Transactions on Computer Systems (TOCS)*, 2(1):39-59, February 1984.

6. A. Birrel, G. Nelson, S. Owicki, and E. Wobber. Network Objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, Asheville, NC, December 1993.

7. R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of 5th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Santa Barbara, CA, July 1995.

8. K. Chandy and C. Kesselman. Compositional C++: Compositional Parallel Programming. In *Proceedings of 6th International Workshop in Languages and Compilers for Parallel Computing*, pages 124-144, 1993.

9. C-C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken. Low-Latency Communication on the IBM RISC System/6000 SP. In *Proceedings of ACM/IEEE Supercomputing*, Pittsburgh, PA, November 1996.

10. D. Culler, S. Goldstein, K. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing (JPDC)*, June 1993.

11. D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumeta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of ACM/IEEE Supercomputing*, Portland, OR, November 1993.

12. B. Ford and J. Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. In *Proceedings of Winter 1994 USENIX Conference*, San Francisco, CA, January 1994.

13. I. Foster and K. Chandy. Fortran-M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing (JPDC)*, 25(1), 1994.

14. I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach for Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing (JPDC)*, 37, 1996.

15. S. Goldstein, K. Schauser, and D. Culler. Lazy Threads, Stacklets, and Synchronizers: Enabling Primitives for Compiling Parallel Languages. In *3rd Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, 1995.

16. A. Grimshaw. An Introduction to Parallel Object-Oriented Programming with Mentat, Technical Report 91-07, University of Virginia, July 1991.

17. A. Grimshaw, and W. Wulf. Legion -- A View From 50,000 Feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, California, August 1996.

18. K. Johnson, M. F. Kaashoek, and D. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Cooper Mountain, CO, December 1995.

19. V. Karamcheti, and A. Chien. Concert – Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Proceedings of ACM/IEEE Supercomputing*, Portland, OR, November 1993.

20. V. Karamcheti, J. Plevyak, and A. Chien. Runtime Mechanisms for Efficient Dynamic Multi-threading. In *Journal of Parallel and Distributed Computing (JPDC)*, 37(1):21-40, 1996.

21. A. Krishnamurthy, K. Schauser, C. Scheiman, R. Wang, D. Culler, and K. Yelick. Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, October 1996.

22. J. Liedtke. Improving IPC by Kernel Design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, Asheville, NC, December 1993.

23. B-H. Lim, C-C. Chang, G. Czajkowski, and T. von Eicken. Performance Implications of Communication Mechanisms in All-Software Global Address Space Systems. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Las Vegas, NV, June 1997.

24. N. Madsen. Divergence Preserving Discrete Surface Integral Methods for Maxwell's Curl Equations Using Non-Orthogonal Unstructured Grids. Technical Report 92-04, RIACS, February 1992.

25. S. O'Malley, T. Proebsting, and A. Montz. USC: A Universal Stub Compiler. In *Proceedings of the Symposium on Communication Architectures and Protocols (SIGCOMM)*, London, UK, August 1994.

26. R. Martin. HPAM: An Active Message Layer for a Network of Workstations. In *Proceedings of Hot Interconnects II*, Palo Alto CA, August 1994.

27. S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of ACM/IEEE Supercomputing*, San Diego, CA, November 1995.

28. J. Plevyak, V. Karamcheti, X. Zhang, and A. Chien. A Hybrid Execution Model for Fine-Grained Languages on Distributed Memory Multicomputers. In *Proceedings of ACM/IEEE Supercomputing*, San Diego, CA, December 1995.

29. K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *Proceedings of the 9th International Parallel Processing Symposium (IPPS)*, Santa Barbara, CA, April 1995.

30. M. Schroeder and M. Burrows. Performance of Firefly RPC. In *ACM Transactions on Computer Systems (TOCS)*, 8(1):1-17, February 1990.

31. J. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, pages 5-44, March 1992.

32. V. Sunderam, G. Geist, J. Dongarra, and R.Manchek. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, 20(4), April 1994.

33. K. Taura, A. Yonezawa. Fine-Grain Multithreading with Minimal Compiler Support – A Cost Effective Approach to Implementing Efficient Multithreading Languages. In *Proceedings of ACM Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, June 1997.

34. T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium in Computer Architecture (ISCA)*, Gold Coast, Australia, May 1992.

35. D. Walker and J. Dongarra. MPI: A Standard Message Passing Interface. *Supercomputing*, 12(1), 1996.

36. D. Wallach, W. Hsieh, K. Johnson, M. Kaashoek, and W. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. In *Proceedings of 5th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Santa Barbara, CA, July 1995.

37. A. Yonezawa, editor. ABCL: An Object-Oriented Concurrent System – Theory, Language, Programming, Implementation and Application. The MIT Press, 1990.