Evaluating the Performance Limitations of MPMD Communication

Chi-Chao Chang[†], Grzegorz Czajkowski[†], Thorsten von Eicken[†], and Carl Kesselman[‡]

[†] Department of Computer Science
Cornell University
Ithaca, NY 14853

[‡] Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292

Abstract

The MPMD approach for parallel computing is attractive for programmers who seek fast development cycles, high code re-use, and modular programming, or whose applications exhibit irregular computation loads and communication patterns. RPC is widely adopted as the communication abstraction for crossing address space boundaries. However, the communication overheads of existing RPC-based systems are usually an order of magnitude higher than those found in highly tuned SPMD systems. This problem has thus far limited the appeal of high-level programming languages based on MPMD models in the parallel computing community.

This paper investigates the fundamental limitations of MPMD communication using a case study of two parallel programming languages, Compositional C++ (CC++) and Split-C, that provide support for a global name space. To establish a common comparison basis, our implementation of CC++ was developed to use MRPC, a RPC system optimized for MPMD parallel computing and based on Active Messages. Basic RPC performance in CC++ is within a factor of two from those of Split-C 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. The results suggest that RPC-based communication can be used effectively in many high-performance MPMD parallel applications.

1 Introduction

The Multiple-Program-Multiple-Data (MPMD) model is attractive yet rarely exploited in parallel applications running on distributed memory multi-computers. In contrast to the Single-Program-Multiple-Data (SPMD) model, in which a fixed number of identical programs operate on their local data and communicate with one another at well defined points in time, the MPMD model is more general. It allows multiple programs to dynamically create concurrently executing tasks that communicate with one another at any point in time. This approach is well suited for applications that exhibit irregular or unknown communication patterns, or that can benefit from a "client-server" type of setting. Software engineering is made easier due to the modularity of the code, which promotes code re-use and the ability to compose programs [4].

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 sockets). 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. From a software-engineering point of view, RPC is a widely accepted communication abstraction for an MPMD environment [10].

In multi-computers, lower-level messaging systems such as MPI [27] and PVM [23] can be used for MPMD programming and typically 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.

High-level MPMD languages (e.g. Mentat [13], CC++ [4], and Fortran-M [9]) and runtime systems (e.g. Nexus [11]) support some combination of dynamic task creation, load balancing, global name space, concurrency, and heterogeneity. Due to the need for crossing program domains, for asynchronously detecting incoming communication, and for potentially spawning new threads, the communication overheads in these systems are often prohibitively high for a multi-computer. As a result, systems based on an SPMD model (e.g. Split-C [8] and CRL [14]) have a significant performance advantage over

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.

MPMD-based ones and are preferred for parallel application development.

This paper investigates the feasibility of high-performance parallel computing using an MPMD model by analyzing the fundamental performance limitations in communication. This analysis is based on the implementations of two representative languages, CC++ and Split-C, running on the IBM RS/6000 SP (a.k.a. SP2). CC++ offers an MPMD programming model with RPC as the primary communication abstraction and has been used in meta-computing applications [16]. Split-C is built on Active Messages (AM) [26] and has been widely used in high-performance parallel computing research [6,17]. This work focuses on a homogeneous environment in order to isolate the inherent costs of MPMD over SPMD communication. Other factors that affect the performance of MPMD applications such as load balancing, data distribution, and heterogeneity are beyond the scope of this paper.

The original implementation of CC++ is layered on top of Nexus, a modular, highly portable runtime system that supports heterogeneous machines and networks. Because of the difficulty in distinguishing the fundamental MPMD communication overheads from those introduced by software engineering constraints, a new implementation of CC++ was developed to use MRPC [5], an RPC system designed and optimized for MPMD parallel computing. MRPC combines the efficient control and data transfer provided by AM, a simplified SPMD RPC mechanism, with a minimal multithreaded runtime system that extends AM with the features required to support MPMD. The design of MRPC allows us to study and quantify the minimal runtime overhead necessary to extend a simple SPMD RPC to an MPMD RPC. MRPC has been integrated into CC++ without any modification to CC++'s front-end translator or the back-end compiler. In a sense, this paper identifies and quantifies the potential gains of compiler optimizations such as whole-program analysis and specialized code generation that further reduce the RPC communication costs.

Basic RPC performance in CC++ using MRPC is within a factor of two from those of Split-C 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. Our observations suggest that the MPMD model is reasonable for many applications running on multi-computers especially when the software-engineering benefits outweigh the small performance gap.

This paper makes the following contributions:

- It discusses the major limitations of RPC-based communication for MPMD parallel computing on homogeneous, distributed memory multi-computers.
- It quantifies these limitations using a direct comparison between an MPMD language (CC++) and an SPMD one (Split-C) through a series of micro-benchmarks and three representative applications.

The rest of this paper is organized as follows. Section 2 gives a brief introduction to Split-C and CC++. Section 3 discusses the issues in RPC-based communication. Section 4 briefly describes the CC++ implementation on top of MRPC. Section 5 describes the experimental setup used to evaluate communication in CC++. Section 6 presents the results of our experiments. Section 7 discusses related work and Section 8 concludes.

2 Split-C and CC++: SPMD versus MPMD

Split-C is a parallel extension of C that supports efficient access to global address space using global pointers. It provides a small set of global access primitives and simple parallel-storage layout declarations. The compiler performs simple source-to-source transformations, converting the language extensions into runtime library calls. The global name space assumes an SPMD model: all processors execute the same program. Split-C has been ported to several distributed memory multiprocessors and is generally very efficient.

CC++ is 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 through language extensions, making all its public methods and data accessible by other processor objects using global pointers.

The key differences between these languages are:

Control and Synchronization: In Split-C, the program executing on one node is single-threaded and synchronizes with the other nodes through `barrier` calls. In CC++, new threads of control can be created using `spawn`, and control blocks can execute concurrently if annotated with the `par` and `parfor` keywords. Synchronization is achieved using write-once `sync` variables.

Global Name Space: The structure of Split-C's global name space is made visible to the programmer in that a global pointer consists of a processing node number and a local address on that node. In particular, arithmetic on the node part of the global pointer is used to access static variables on arbitrary nodes and to spread arrays across all nodes. The Split-C type system distinguishes global pointers from ordinary local ones and communication takes place automatically when a remote pointer is dereferenced. Unlike Split-C, global pointers in CC++ are opaque. The

compiler front-end translates all global pointer dereferences into RPCs. Member methods[1] of remote objects referenced by a global pointer can be invoked directly. Method invocation stubs with argument marshalling and unmarshalling code and communication calls into the runtime system are generated automatically.

Communication: Split-C uses a number of variants of the C assignment statement to access remote locations synchronously, to issue split-phase gets, puts, and one-way stores. A number of bulk-transfer primitives support the efficient transfer of contiguous memory blocks. In CC++, all communication takes place in the form of RPCs. Although there is no restriction on the number and types of arguments and of the result value that a remote method can have, CC++ programmers have to provide their own data marshalling operations for complex data structures. Bulk data transfer is attained by passing all the data as arguments to an RPC.

3 Performance Limitations of RPC

RPC introduces a number of issues into the communication layer that are absent from Split-C: method names must be resolved to entry point addresses, arguments must be marshaled, and multiple threads of control must be supported. In addition, the modularity and higher levels of abstraction offered by CC++ require a more local view of the interactions of communication and computation than in Split-C, 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.

Method Name Resolution: A CC++ 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 method statically. The mapping from the method name to its entry point address must be made at runtime, which requires either extra round-trip inquiry messages or the transmission of the name instead of its address in messages. In contrast, the Split-C runtime system handles only a single program image and assumes that remote code and data are located at the same addresses as on the local node.

Argument Marshalling: In CC++ 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 CC++'s RPC strictly more powerful than the global memory access primitives in Split-C, which only supports a "shallow copy" of user-defined data types. On the other hand, this flexibility incurs at least one extra copying of the data as well as the overhead of calling the serialization methods. The CC++ compiler can only inline these calls in simple cases, but in other cases (especially in the presence of inheritance) a full dynamic method invocation is required. Some systems [1,19] attempt to reduce the cost of RPC by restricting the types that can be marshaled or by only supporting shallow copies of RPC arguments.

Data Transfer: Split-C supports efficient data transfer by requiring the sender to specify a remote buffer address where the data will be placed, avoiding dynamic memory allocation on message reception. In CC++, 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 arguments and another for the return value) into RPC besides those required for data marshalling.

Control Transfer: An 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 methods. In particular, a method 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 CC++. 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 incoming 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 [3,7] minimize the threading costs by making threads run to completion to eliminate context switches, by performing custom stack management (e.g. using stacklets [12] or spaghetti stacks), or by reducing synchronization costs through custom code generation [12]. Split-C takes an even more radical approach — offering only a single computation thread — and relies on split-phase remote accesses to tolerate latencies.

Message Reception: 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

[1] In this paper, we use the terms *method* and *procedure* interchangeably.

(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. This approach is used in Split-C.

The more modular programming style promoted by CC++ 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 latency.

4 CC++ over MRPC: An Overview

MRPC is an RPC system designed and optimized for MPMD parallel computing on a homogeneous, distributed memory multi-computer. The system consists of a runtime library that performs communication, marshalling of basic data types and remote program execution. The current implementation runs on the IBM SP multi-computer. The total source code size is 4028 lines of C++ and 1881 lines of C. The communication module is layered on top of AM and uses a custom, non-preemptive threads package. We present a brief overview of the MRPC implementation in this section – a thorough description can be found in [5].

CC++ has been ported to use MRPC. It provides MRPC with stub generation and a global name space. The system is composed of a front-end (CC++ to C++) translator and a back-end C++ compiler. The front-end translates all global pointer accesses into caller stubs and wraps global procedure entry points with callee stubs.

After compilation, a CC++ program consists of an executable (which contains the main entry point) and possibly a number of processor objects. 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 caller stub instantiates a bi-directional endpoint that holds a send (S-) and a receive buffer (R-buffer). It transmits the contents of the S-buffer (control information, callee stub entry point and call arguments) using AM to a static, per-node buffer zone, and blocks the caller thread. Upon arrival of the message in the callee side, a *dispatch* handler is responsible for extracting the control information from the static buffer, copying the arguments from the static buffer to a newly allocated R-buffer, looking up and invoking the callee stub with the R-buffer. The callee stub unmarshals the arguments, calls the method, and issues a reply AM to the caller with the return value. A *return* handler[2] is responsible for copying the return value from the static buffer into the caller R-buffer and for unblocking the caller thread. The use of the static buffer zone eliminates the need for the caller to issue an extra round-trip message to determine an R-buffer, but still requires two data copies on the caller (during RPC reply) and callee sides.

Stub caching is used to eliminate the overhead of name mapping. The entry point addresses for callee stubs are resolved using a look-up into a local hash table. Each processing node maintains a table of stub addresses which is indexed by processor number and method name hash value. During runtime initialization, local callee stubs are registered into the table, and remote entries are marked as invalid. The initiator of an RPC 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 entry point address is fetched and passed to the remote node in the message. If the entry is invalid, the entire method name is passed in the message and the resolution occurs at the remote end with a message being sent back to update the local entry. The main benefits of resolving the name on the caller side are to reduce the size of the messages and calculate the hash value at runtime initialization.

To reduce the marshalling overheads, S- and R-buffers are pre-allocated, and R-buffers are made *persistent*, i.e. for recently invoked procedures are kept allocated so they can be managed by the caller. Initially, for a "cold" RPC, arguments are marshaled into the S-buffer and transferred to a per-node static buffer area at the callee side. The dispatch handler allocates a new R-buffer, copies the data from the static buffer area into the R-buffer, and marks it as attached to the method being called. The address of the R-buffer is returned with the stub table update message. Subsequent cached invocations will copy data directly into the persistent, per-node R-buffer associated with the remote procedure, thus eliminating one extra data copy on the callee side.

To eliminate a data copy on the caller side during the RPC reply, the caller can allocate an R-buffer and pass its address into the RPC message. The callee can ship the return value directly into this buffer. This optimization has not been incorporated into the current implementation.

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 [6]. In order to avoid deadlocks when there is no runnable thread, MRPC forks a polling thread on each processing node at initialization.

[2] Both the dispatch and the return handlers run atomically with respect to incoming messages.

```
// 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));
// Prefetch
for (i = 0; i < 20; i++)
  lx := *gpY; // split-phase
sync();
```

Figure 2. 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 RPC
gpObj->foo();
// 1-Word RPC
gpObj->foo(ly);
// 2-Word RPC
gpObj->foo(ly, lz);
// 0-Word Atomic RPC
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);
// Prefetch
parfor (i = 0; i < 20; i++)
  lx = *gpY;
```

Figure 1. CC++ Micro-benchmarks Pseudo-Code

5 Experimental Setup

The experimental setup consists of a series of CC++ and Split-C communication micro-benchmarks and three applications: EM3D, Water, and Blocked LU Decomposition[3]. 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.

Micro-benchmarks: Figure 2 and Figure 1 show the micro-benchmarks used:

- Several variations on Ping-Pong to measure the round-trip time of the 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 caller or callee), *0-Word*, *1-Word*, *2-Word* (each with a thread switch at the caller side only), *0-Word Threaded* [4] (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 plus a reply message with the 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*).
- Prefetching of 20 remote `double`s accessed through global pointers using `parfor` blocks in CC++ and split-phase gets in Split-C (*Prefetch*).

EM3D: is a parallel application that simulates electromagnetic wave propagation [8,18]. 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

[3] The CC++ version of these applications is heavily based on the original Split-C implementations [17] to allow for a fair comparison.

[4] *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.

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 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.

Water: is an N-body molecular dynamics application taken from the SPLASH benchmark suite [22] 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 inputs of 64 and 512 molecules distributed over 4 processors.

Blocked LU Decomposition: implements LU factorization of a dense matrix as described in the SPLASH benchmark suite [22]. 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 RPCs. The input is a 512x512 matrix of `double`s with a block size of 16x16.

6 Results

Micro-benchmarks: As seen in Table 3, the round-trip time of a *0-Word Simple* is only 12 μs slower than the base round-trip time of the AM layer, and 21 μs faster than IBM MPL. Other variations of the null RPC scale according to the number of thread operations involved. Due to method stub caching, the method lookup cost is about 3 μs and is accounted for in *CC++ 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 μsecs 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 active messages.

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.

The prefetching benchmark shows that the overhead of thread management reduces the effectiveness of latency hiding substantially.

Instrumentation: The execution times of the main computation kernels for each of the applications are broken down into a *CPU* component for (mostly) integer and FP computations, a *net* component for communication latencies and overheads, a *thread mgmt* component for thread creations, scheduling, and context switches, a *thread sync* component for locks and signals (but excluding the waiting time, which is accounted for in net), and a *runtime* component for stub invocations, argument marshalling, method name lookup, and other runtime overheads in CC++. Split-C's runtime is accounted for in the *CPU* component.

Benchmarks	CC++							Split-C		
			Threads							
	Total (us)	AM (us)	Time (us)	Yield	Create	Sync	Runtime (us)	Time (us)	AM (us)	Runtime (us)
0-Word Simple	67	55	4	0	0	10	8	-	-	-
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	-	-	-
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
Prefetch 20-Word	35.4	5.3	21	2	1	10	9.1	12.1	6.2	5.9

Table 3. Micro-benchmark results. The *Total* time reported for each test was obtained by averaging over 10000 iterations. In CC++, *Total* is the sum of the messaging layer (*AM*), the *Threads Time*, and the *Runtime*. In Split-C, it is just *AM* plus *Runtime*. *Threads Time* is estimated by multiplying the number of thread calls per iteration with their respective costs (4.6 µsecs for thread creation, 5.8 µsecs for a context switch, and 0.4 µsecs for a lock, unlock, or condition variable signal call). The round-trip latency of IBM's native MPL under AIX 3.2.5 is 88 µs.

EM3D: Figure 4 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

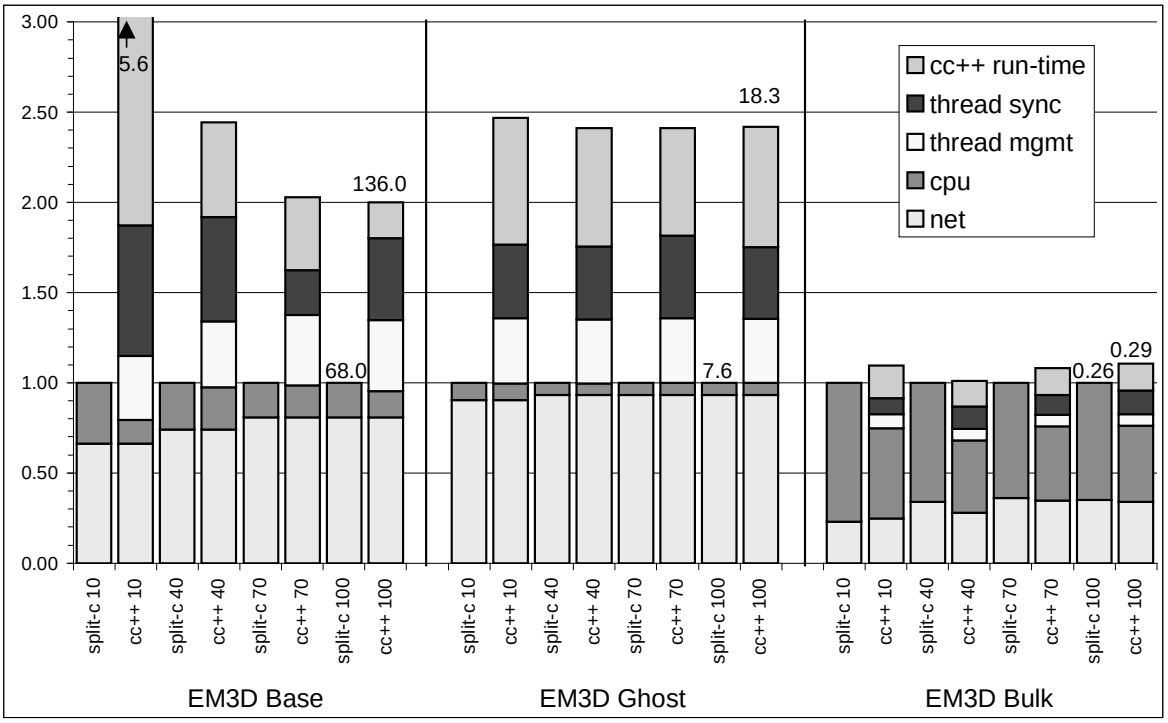

Figure 4. Breakdown of EM3D per-edge execution times for 10%, 40%, 70%, and 100% of remote edges, normalized against Split-C. The absolute execution time (in seconds) for 100% of remote edges is indicated above the respective bars.

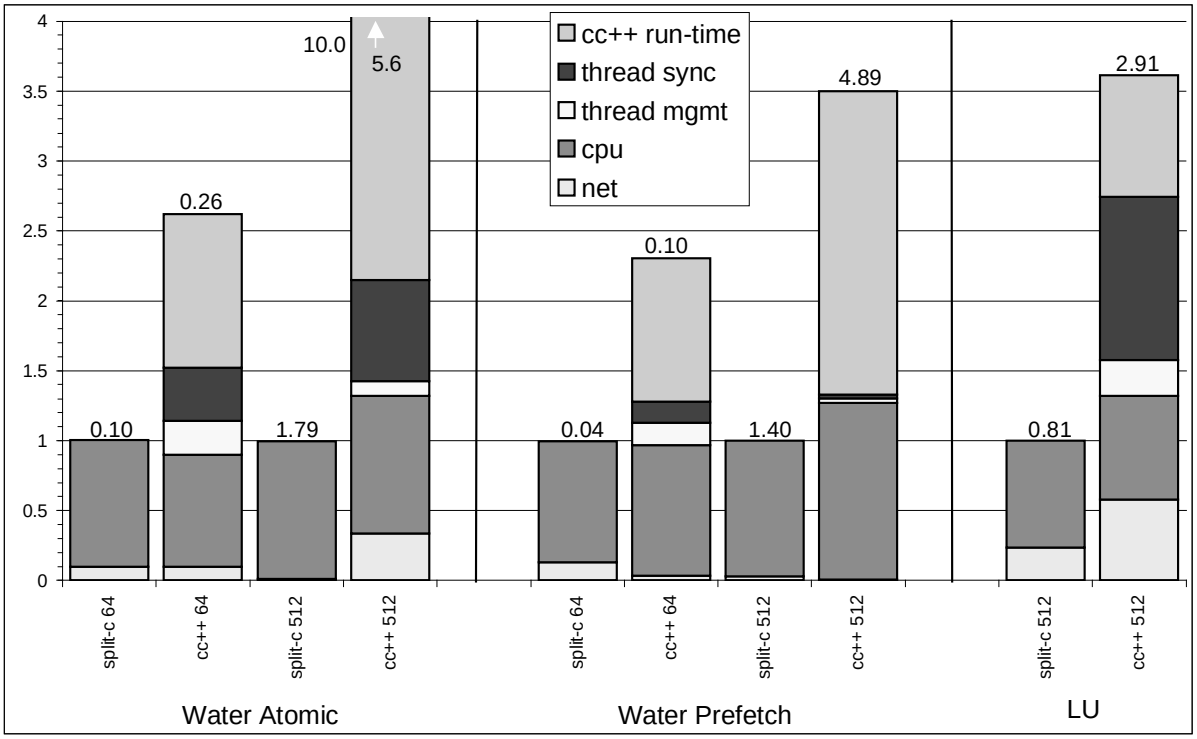

Figure 5. Breakdown of Water and LU absolute execution times (printed on top of each bar, in seconds), normalized against Split-C.

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. 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 very 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.

Water: Figure 5 shows the performance of the main computation loop in Water. *Water-atomic* uses small messages to read from and write to the remote molecules. The performance gap between CC++ and Split-C is 2.6 for 64 molecules and 5.6 for 512 molecules. The number of remote accesses in CC++ increases quadratically with the input size, increasing its runtime overhead and degrading its relative performance. *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 and thus yields in both Split-C and CC++ a 60% improvement for 64 molecules. However, for 512 molecules, CC++ improves by 51% compared to Split-C's 22%, closing the performance gap to 3.5. The impact of this optimization is larger in CC++ because of higher communication latencies. CC++ *runtime* accounts for about 50-60% of the gap, a great deal of which is due to data marshalling.

LU: The performance gap of 3.6 between CC++ and Split-C is mainly due to the extra data copying during matrix block transfers (about 20% of the gap) and to synchronization (about 32%). The *net* time in *cc-lu* is about 2 times higher than in *sc-lu*, mostly due to busy waiting (polling).

Comparison with CC++/Nexus: Additional measurements with the same set of applications compiled with CC++/Nexus[5] show that the CC++/MRPC yields improvements of 5 to 35-fold over CC++/Nexus. In compute-bound applications (*water-atomic/prefetch* with 512 molecules and *cc-lu*), the performance gap between CC++/MRPC and CC++/Nexus is about 5x to 6x. In applications with higher communication to computation

[5] 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.

ratios, the gap is about 16x to 22x in *water-atomic/prefetch* (with 64 molecules), 10x in *em3d-bulk*, 29x in *em3d-ghost*, and 35x in *em3d-base* (all with 100% remote edges).

Discussion: The micro-benchmark results demonstrate that the basic MPMD communication in CC++ is competitive with Split-C as well as other messaging layers. CC++ applications perform within a factor of two to six from Split-C.

The thread costs show that multithreading in RPC accounts for 25-50% of the performance gap between CC++ and Split-C. 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 32% in *cc-lu*. 98-99% 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 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.

Stub invocations, data marshalling, and message dispatch account for over 50% of the performance gap. The overhead of method name translation is negligible due to the stub caching. Data copying overheads are especially harmful when large amounts of data are transferred, as in LU, and when the communication to computation ratio is high, as in Water.

7 Related Work

As far as we know, this is the first study that compares and evaluates the performance of MPMD communication with respect to SPMD on a multi-computer. Previous research on MPMD systems [11,13] usually emphasizes other aspects such as portability, flexibility, and heterogeneity, making it difficult to identify the fundamental overheads. Moreover, such systems are usually evaluated in isolation.

A key performance aspect in the MPMD model is the efficient integration of communication and threads. The simplest form of single-threaded remote method invocation was introduced by Active Messages [26]. Optimistic Active Messages (OAM) [28] 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 re-started on a separate thread if it blocks. But OAM assumes an SPMD model and does not specifically address the communication bottlenecks when that assumption is no longer valid. Nexus also provides a framework for integrating threads with communication [11], but does not investigate the performance impact in applications.

A large amount of literature [7,12,15,20,21] describes sophisticated compiler and runtime schemes that reduce the cost of thread management in languages that support fine-grain concurrency. All of them provide specialized frame management and procedure-calling convention that requires the compiler to generate native or assembly-like C code. Our results show that these optimizations could potentially reduce the performance gap between SPMD and MPMD parallel applications by about 10 to 15%. Taura and Yonezawa provide a comprehensive summary of these schemes in [25], and propose a cost-effective, library-based implementation of threads that maps onto the single stack execution model of C.

8 Conclusion

This paper investigates the feasibility of high-performance MPMD parallel computing by analyzing the performance limitations of the MPMD communication paradigm. The analysis is based on a direct comparison of between an MPMD language (CC++) and an SPMD one (Split-C) running on an IBM SP multi-computer.

Results show that basic remote data access in CC++ can be optimized to within a factor of two from Split-C and native messaging layers. 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 fraction of the remaining gap is due to thread synchronization (15-30%), which is necessary for maintaining thread-safe communication. Thread management operations such as creation and context switching account for less than 15% of the gap. RPC overheads such as stub invocations, argument serialization and message dispatch contribute to the majority of the gap (over 50%). As a result, we are currently investigating compiler-based techniques to streamline stub invocations, to eliminate data copying altogether during argument serialization, and to dispatch messages more efficiently using caller-managed thread scheduling.

This work suggests that the MPMD model based on RPC is reasonable for high-performance applications running on multi-computers. In many cases, the simplicity of RPC abstraction along with the software-engineering benefits such as modularity and program composition outweigh the performance gap between SPMD and MPMD.

9 References

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

2. 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.

3. 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.

4. 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.

5. C-C. Chang, G. Czajkowski, and T. von Eicken. MRPC: A High Performance RPC System for MPMD Parallel Computing. Submitted to *Software: Practice and Experience*, special issue on Parallel and Distributed Operating Systems, June 1997.

6. 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.

7. 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.

8. 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.

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

10. I. Foster, J. Geisler, S. Tuecke, and C. Kesselman. Multimethod Communication for High-Performance Metacomputing. In *Proceedings of ACM/IEEE Supercomputing*, Pittsburgh, PA, November 1996.

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

12. 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.

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

14. K. Johnson, M. 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.

15. 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.

16. C. Lee, C. Kesselman, and S. Schwab. Near-Real-Time Satellite Image Processing: Meta-Computing in CC++. *IEEE Computer Graphics and Applications*, 16(4), July 1996.

17. 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 6th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Las Vegas, NV, June 1997 (to appear).

18. 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.

19. 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.

20. E. Mohr, D. Kranz, and R. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264-280, July 1991.

21. 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.

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

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

24. K. Taura, S. Matsuoka, and A. Yonezawa. An Efficient Implementation Scheme of Concurrent Object-Oriented Languages on Stock Multi-computers. In *Proceedings of the 4th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, San Diego, CA, May 1993.

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

26. 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.

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

28. 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.