Appears in PPoPP '95, July 1995.

Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation

Deborah A. Wallach, Wilson C. Hsieh, Kirk L. Johnson,
M. Frans Kaashoek, and William E. Weihl
M.I.T. Laboratory for Computer Science
Cambridge MA 02139, USA

{kerr,wchsieh,tuna,kaashoek,weihl}@lcs.mit.edu

Abstract

Low-overhead message passing is critical to the performance of many applications. Active Messages [27] reduce the software overhead for message handling: messages are run as handlers instead of as threads, which avoids the overhead of thread management and the unnecessary data copying of other communication models. Scheduling the execution of Active Messages is typically done by disabling and enabling interrupts, or by polling the network. This primitive scheduling control, combined with the fact that handlers are not schedulable entities, puts severe restrictions on the code that can be run in a message handler. This paper describes a new software mechanism, *Optimistic Active Messages* (OAM), that eliminates these restrictions; OAMs allow arbitrary user code to execute in handlers, and also allow handlers to block. Despite this gain in expressiveness, OAMs perform as well as Active Messages.

We used OAM as the base for an RPC system, *Optimistic RPC* (ORPC), for the Thinking Machines CM-5 multiprocessor; it consists of an optimized thread package and a stub compiler that hides communication details from the programmer. ORPC is 1.5 to 5 times faster than traditional RPC (TRPC) for small messages and performs as well as Active Messages (AM). Applications that primarily communicate using large data transfers or are fairly coarse-grained perform equally well, independent of whether AMs, ORPCs, or TRPCs are used. For applications that send many short messages, however, the ORPC and AM implementations are up to three times faster than the TRPC implementations. Using ORPC, programmers obtain the benefits of well-proven programming abstractions such as threads, mutexes, and condition variables, do not have to be concerned with communication details, and yet obtain nearly the performance of hand-coded Active Message programs.

This work was supported in part by the Advanced Research Projects Agency under contract N00014-94-1-0985, by a NSF National Young Investigator Award, by Project Scout under ARPA contract MDA972-92-J-1032 and by a fellowship from the Computer Measurement Group. Weihl was also supported by the Digital Equipment Corporation while on sabbatical at Digital's Systems Research Center.

1 Introduction

Low-overhead message passing is critical to the performance of many applications. Active Messages [27] reduce the software overhead for message handling: messages are run as *handlers*, which are fragments of user code without thread descriptors that are executed on the stack of a running computation. This scheme avoids the overhead of thread management and the unnecessary data copying of other communication models. Scheduling the execution of Active Messages is typically done by disabling and enabling interrupts, or by polling the network. This primitive scheduling control, combined with the fact that handlers are not schedulable entities, puts severe restrictions on the code that can be run in a message handler. For example, a message handler is not allowed to block, and can only run for a short period of time. Blocked Active Messages create deadlocks; long-running Active Messages can congest the network. This paper describes a new software mechanism: *Optimistic Active Messages* (OAM), which eliminates these restrictions. OAMs conceptually allow arbitrary user code to be executed in response to a message, and allow arbitrary synchronization between messages and computation; despite this gain in expressiveness, OAMs perform as well as Active Messages.

Optimistic Active Messages achieve the performance of Active Messages (AMs) by allowing user code to execute in a message handler instead of a thread, thus avoiding thread management overhead and data copying time. Because handlers are not schedulable entities, executing arbitrary user code in message handlers is difficult. The most serious problem is deadlock. If a message handler blocks (*e.g.,* waits to acquire a lock), deadlock can easily result. Our solution is to be *optimistic* and compile handlers with the assumptions that they run without blocking and complete quickly enough to avoid causing network congestion. At runtime, these assumptions must be verified; if they are false, we revert to a slower, more general technique for processing the handler (*e.g.,* we create a separate thread). This approach eliminates thread management overhead for handlers that run to completion, and will achieve performance equal to Active Messages when most handlers neither block nor run for too long. Furthermore, it frees the programmer from the burden of dealing with the restrictions of Active Messages, which greatly simplifies writing parallel applications.

To validate our approach we have built an efficient thread library and a stub compiler for remote procedures written in C; we call this system *Optimistic RPC* (ORPC). The system runs on the CM-5 [18], a high-performance multicomputer. Programmers specify remote procedures; from them, the stub compiler generates code for handling messages, marshaling, and implementing Optimistic Active Messages. For comparison, we also wrote a version of our system that creates a thread for each message, such as a traditional RPC system would do; we call this version *Traditional RPC* (TRPC). Using our RPC systems we have written or ported several applications, including Water from the SPLASH benchmark suite [22]. We compare these RPC implementations to hand-coded AM implementations.

We have chosen to implement an RPC system because it is a simple and programmer-friendly communication model that is widely used. Our approach can easily be applied to other programming models that create threads for message arrivals: for example, object-oriented parallel systems such as ABCL/f [23], Concurrent Smalltalk [11], Concurrent Aggregates [6], Orca [3], and Prelude [28]. In fact, we have ported the Orca system to the CM-5 and modified the compiler to run simple method calls in handlers using OAMs. For a number of microbenchmarks and Orca applications we measured performance improvements that ranged from 2 to 30 times faster than the original code [13, 15].

This work makes four contributions. First, we have built an RPC system that hides communication details from the programmer and still provides performance equal to that of hand-coded AM implementations on microbenchmarks.

Second, we show that by implementing an efficient thread package, an RPC system that generates a thread for each message can be quite competitive with Active Messages. In the best case, AM is only 33% faster than TRPC for short messages on the CM-5. For long messages the cost of starting a thread per message is amortized over the amount of data sent, since it does not vary with data size.

Third, even though TRPC is fast, it is not fast enough; we show that ORPC can bridge the performance gap between TRPC and AM. In our applications, most remote procedures can indeed be executed optimistically in handlers without having to create a thread. Therefore, our system delivers the programming convenience of RPC with nearly the performance of Active Messages.

Finally, we show that applications that primarily communicate using large data transfers or are fairly coarse-grained perform equally well, independent of whether AMs, ORPCs, or TRPCs are used. For applications that send many short messages, however, the ORPC and AM implementations are up to three times faster than the TRPC implementations.

The remainder of the paper is structured as follows. Section 2 describes the ideas underlying Optimistic Active Messages in more detail. Section 3 describes our prototype implementation of Optimistic RPC on the CM-5. Section 4 describes the experiments that we performed to measure our system. Finally, Section 5 discusses related work, and Section 6 summarizes our results.

2 Optimistic Active Messages

In this section, we discuss the Active Message programming model in more detail. We then describe Optimistic Active Messages, which generalizes Active Messages by eliminating its disadvantages.

Active Messages are messages that initiate computation in handlers. A handler executes on the stack of a running computation. Because it has no thread descriptor, it is not under the control of a scheduler. Using Active Messages has two major performance advantages: (1) thread management time (creation, scheduling, and termination) is avoided, and (2) data from the message can be immediately integrated into the computation by copying it directly from the network interface, which is especially important for large messages. Other communication models, such as RPC or send/receive, create threads and/or copy data through buffers; as a result, Active Messages outperforms these other models, and is even competitive with communication models that rely on hardware support [27].

Unfortunately, because handlers are not under the control of a scheduler, severe restrictions must be put on the code that is executed in them. Handlers must execute without blocking; otherwise, deadlock may result. For example, a handler that attempts to acquire a held lock may cause deadlock. Another form of deadlock can arise when a handler attempts to send a reply message while the network is full. In addition, if a handler runs too long (or is not immediately executed), it blocks messages from other processors in the network. In general, Active Messages put many burdens on the programmer, who must worry about deadlock in the network, synchronization between messages and computation, and other timing-dependent problems.

Programmers are forced to carefully craft their applications in order to benefit from Active Messages. Instead of using mutexes, they typically synthesize critical regions by disabling interrupts and polling for messages. In addition, programmers avoid sending messages in the middle of handlers to guarantee atomicity (the send routine on the CM-5 first polls the network before sending to avoid distributed deadlocks). Instead of using condition variables, conditional synchronization is often implemented in an ad-hoc manner by testing a condition in the handler: if the condition holds, the data is integrated with the computation; otherwise, some form of continuation is created. Finally, programmers attempt to avoid blocking the network by polling frequently enough outside of critical regions (but not too frequently, since otherwise the overhead of polling might slow down the application). On the CM-5, programmers are helped by a substantial amount of buffering in the network, which allows applications to poll infrequently; on architectures like Alewife [1] there is limited buffering in the network, so infrequent polling causes other processors to be blocked very quickly. In short, unless applications are fairly static, writing programs with Active Messages requires carefully orchestrating communication with computation, and sometimes turns into a black art.

For more dynamic applications, programmers should be able to use well-proven programming constructs such as threads, mutexes, and condition variables, so that they can properly schedule communication with computation. Threads do not have the restrictions of handlers, and can be rescheduled. Creating a thread in our best implementation on a SPARC node of the CM-5 takes only 7 microseconds in the best case (when the processor is idle). In the cases where thread creation also includes an inter-thread context switch, however, the cost is 60 microseconds. Threads can be made less expensive by restricting their functionality, such as in TAM [8], Filaments [10], and Cilk [4]. However, such threads are not allowed to block, which makes them unusable as a solution in our context. Since the cost of the cheapest round-trip communication on the CM-5 is approximately 10 microseconds, even the relatively small overhead added by the fastest thread package will be too much for some applications, as we will see in Section 4. A different solution is needed to bridge the gap between running a message in a handler and as a thread.

Our solution is to *optimistically* allow arbitrary user code to execute in a message handler, and to promote the handler to a full thread whenever it needs to be rescheduled (*e.g.*, when it cannot complete or when it runs for too long). This method allows arbitrary synchronization between communication and computation. Applications can decide to execute the message immediately as a handler (interrupting the current thread), postpone executing the message until the computation thread leaves a critical region, or postpone its execution by synchronizing it on some event in the application.

To free the programmer from the burden of choosing when to execute a message as a handler and when to upgrade to a thread, we propose to let the compiler and runtime system make these decisions. Remote procedures are compiled with two assumptions: that they usually run without blocking, and that they complete quickly enough to avoid causing network congestion. When a remote procedure might block or run for a long time, we include checks in the compiled code that detect these situations and revert to a slower, more general technique for processing the remote procedure call (*e.g.*, creating a separate thread). (The definition of "too long" is both architecture- and application-dependent; accounting for the former is straightforward, but accounting for the latter could be more difficult.) This strategy eliminates thread management overhead for remote procedures that run to completion within message handlers.

When a remote procedure call cannot complete within a message handler, we say that the optimistic execution as an Active Message must *abort*. We identify three reasons why an OAM may have to abort: it needs to wait for a lock or on a condition variable, it needs to send a message when the network is busy (*i.e.*, the network interface buffers are full), or it needs to run for a long time. Each of these conditions is straightforward to detect.

There are three ways to abort an OAM. First, an OAM can create a continuation for itself: the remainder of the OAM executes in a separate thread. This strategy can be viewed as lazy thread creation [20]. Second, an OAM can undo its execution and start a thread to recompute the whole remote procedure call: for example, this scheme is necessary when the call must execute atomically. Finally, an OAM can undo its execution and send a negative acknowledgment to the sender, which would then be responsible for backing off and resending the OAM.

3 Implementation

To evaluate our approach, we have implemented Optimistic RPC on the CM-5. The implementation consists of three parts: (1) an efficient thread package, (2) a stub compiler, and (3) Optimistic Active Messages.

3.1 Thread package

We built a simple, optimized, non-preemptive, user-level thread package for the CM-5 SPARC nodes. It provides thread creation, termination, and scheduling as well as locks and condition variables. A thread consists of a thread descriptor and a stack. Only one thread may be *running* at a time. A thread may be *runnable* but not running, which means that it is waiting to be run. A thread may also be neither running nor runnable; for example, it may be waiting for a lock or condition variable. After a lock or condition variable is released or signaled, respectively, the waiting thread will be marked as runnable.

The thread scheduler decides which threads to run. Threads run to completion except under two conditions: if they suspend while trying to access a lock or condition variable, or if they voluntarily yield. The thread scheduler runs in the context of the thread that called it. When the thread scheduler is asked to schedule a new thread to run, it picks a runnable thread and performs a full context switch, if needed. If no such thread exists, it polls the network for messages.

The thread package implements an optimization to avoid the overhead of context switching, which is high on the CM-5 SPARC nodes (52 microseconds). Whenever the scheduler is running on the stack of a terminated thread, a newly created thread can be called directly from the scheduler; *i.e.*, no context switch needs to occur. This optimization works only for newly created threads because they have no saved registers to load in off their stack; we will refer to this optimization as the *live-stack* optimization [10]. We attempted to perform the related optimization of not saving the registers of a terminated thread while context switching to a runnable thread, but were unable to code this optimization because of the SPARC register windows.

3.2 Stub compiler

The stub compiler takes a user specification of a remote procedure and generates C code for handlers, stubs, marshaling, and data transfer. For each remote procedure it generates the following: an initialization routine, which must be called before the procedure is ever invoked; a termination routine, which may only be called after the last time the procedure is called, and prints statistics about the application behavior as well as cleaning up after the RPC; and handler code. The generated code communicates using Active Messages or the CM-5 block transfer primitive (scopy), depending on the size of the data being sent. Programmers need not be concerned with communication, in that programmers can call remote procedures like regular procedures.

The stub compiler allows a remote procedure to have an arbitrary number of parameters. The first parameter must be the processor ID of the "server" to send it to. The rest of the parameters may be of any primitive type, or may be arrays representing blocks of memory (buffers). The stub compiler currently does not handle the marshaling of structs or unions, but doing so would be straightforward. Parameters may be "in" arguments, in which case their value is passed into the remote procedure at runtime, or "out" arguments, in which case they are set when the RPC returns. Buffer arguments must be followed by an additional argument, specified at run time, which indicates the size of the buffer. The stub compiler handles both synchronous and asynchronous RPCs.

The stub compiler can create stubs for both Traditional RPC (TRPC), which always creates a new thread to run a remote procedure call, and Optimistic RPC (ORPC), which runs a remote procedure as an Optimistic Active Message.

3.3 Optimistic Active Messages

The stub compiler uses Optimistic Active Messages to conceptually execute arbitrary remote procedures in handlers. Thread management overhead is only incurred if a handler is forced to abort. A handler must abort if it requires a lock that is held, or needs to wait for a condition that is false. The stub compiler automatically generates code that checks these locks and conditions; if an abort occurs, the generated code creates a low-cost thread. The stub compiler does not need to generate code to check whether the network is full before sending messages, because the CM-5 software automatically drains the network when sending messages from a handler.

There are several restrictions in our prototype, none of which are fundamental. First, we only abort an optimistic RPC if a lock is unobtainable, or a condition being waited on is false. As a result, programmers must avoid writing long-running remote procedures, as the stub compiler does not yet insert the code in handlers that checks for long-running computations. Second, abort is currently implemented by rerunning the entire function as a thread. This implies that a remote procedure can only modify global state when the following conditions are true: (1) it has acquired all the locks that it needs and (2) it has tested all the conditions that must be true for it to execute.

4 Experiments

In this section we evaluate the performance of our prototype. We compare Traditional RPC and Optimistic RPC using microbenchmarks and four applications. This enables us to show which factors are important in determining end-to-end remote procedure call time, as well as overall application performance.

All of our experiments were run on a 32 MHz 128-node CM-5, using Version 3.2 of the CMMD library [25] running under Patch 2 of CMOST Version 7.3. The CM-5 provides efficient user-level communication: user processes have direct access to the network interface. Because taking interrupts is fairly expensive on the CM-5, all of our applications use carefully tuned polling to check for messages.

4.1 Microbenchmarks

We use microbenchmarks to pinpoint the difference in overheads among TRPC, ORPC, and AM. We also briefly discuss how the difference between the systems varies with the amount of data transferred in the RPC call.

Our thread package allows us to investigate two scheduling strategies for remote procedure calls that execute as threads: placing them at the front of the queue and placing them at the back of the queue. We performed all of our experiments with both strategies: placing threads at the back of the queue always performed worse than placing them at the front of the queue. Therefore, we report all of our results with incoming threads scheduled at the front of the queue. This decision also allows us to factor out scheduling order in our comparison, since both ORPC and TRPC run remote procedure calls first.

System	No thread running	Some thread running
TRPC	21	74
ORPC	14	14
AM	13	-

Table 1: Time (in microseconds) for a round-trip "null" remote procedure call.

4.1.1 Baseline performance

Our first experiment establishes the baseline performance of the system. The client performs a remote procedure call of no arguments that increments a variable on the server. This call can never abort in ORPC. This experiment was performed under two conditions: where there was no thread currently running on the server (the server node has a thread waiting for the experiment to finish, but the thread is not runnable because it is doing a condition wait) and where there was a thread running on the server (the server node's thread is in a tight poll and yield loop waiting for the experiment to finish). Table 1 shows the results.

The Active Message times are provided as the baseline. It measures the best AM performance using the provided libraries from Thinking Machines. ORPC is almost as fast as AM whether or not a thread is currently running on the server.

When a thread is already running, ORPC is more than five times faster than TRPC, which always starts a thread. The overhead of TRPC is due to finding and initializing a new thread structure, and then context switching to the new thread. The context switch alone contributes about 50 microseconds. When no thread is currently running, on the other hand, the live-stack optimization can be applied for TRPC, leaving ORPC only 40% faster than TRPC. For some SPMD applications this optimization can always be applied, while in other cases it can never be applied. In multithreaded applications, this optimization is unlikely to be of much use.

In our implementation, the cost to abort an ORPC is only slightly more expensive than starting a thread. An abort is either 7 or 60 microseconds, depending on whether the live-stack optimization can be applied.

4.1.2 Bulk data transfer performance

We also measured a "null" RPC call with varying amounts of data. As the amount of data sent in the call increases, the absolute performance difference stays constant, and the relative difference becomes smaller. When the amount of data being sent in the RPC is greater than 16 bytes, the bulk data transfer mechanism is needed, which increases the total time for an RPC by about 40 microseconds.

4.2 Application performance

In this section we describe the performance differences among the three systems for four applications: the Triangle puzzle, the Traveling salesman problem, Successive overrelaxation, and Water. For each application, we picked a problem size with a challenging granularity.

All speedup curves are shown normalized to the running time of a purely sequential program that was measured on a single processor of the CM-5. 95% confidence intervals are given; they are generally indistinguishable from the data points.

Our applications tend to have only one computation thread per processor (but potentially many communication-related threads). As a result, the live-stack optimization can be applied frequently, since messages are often processed when the computation thread is suspended waiting on a condition variable. This fact improves the relative performance of the TRPC implementations.

4.2.1 Triangle puzzle

The program solving the Triangle puzzle is an example of a fine-grained application that sends many small messages. The Triangle puzzle is an exhaustive search problem. It consists of a triangular board containing n holes on each side. Each hole except for the center one is filled with a peg at the start. Pegs are removed by jumping, as in checkers. A solution is a sequence of moves that leaves only one peg on the board. The goal is to count the number of solutions. This implementation uses breadth-first search to solve the puzzle: at every step all processors extend each position by one move, and place the (non-redundant) extensions in a transposition table [19]. Each extension is sent using an asynchronous RPC to the processor which holds the correct part of the transposition table for that position. The RPC blocks when the lock on the transposition table is taken.

We ran with a triangle with 6 holes on each side. A sequential C version of the code solved this problem in 13.7 seconds. For this problem size, 688,348 remote procedure calls of 16 bytes are performed, of which none block. As can be seen in Figure 1, the ORPC and AM implementations are almost three times faster (2.9 and 3.2 times, respectively) than the TRPC implementation. The difference is entirely due to the additional overhead for thread management; most of the RPCs in the TRPC implementation require a context switch.

4.2.2 Traveling salesman problem

A Traveling salesman problem (TSP) algorithm finds the shortest Hamiltonian tour of a set of cities. Our implementation uses a master/slave style program based on a branch-and-bound algorithm. The master (server) generates partial routes and stores them in a job queue. Each slave (client) repeatedly performs the following steps: it sends an RPC that requests one partial route from the job queue; it then generates all full routes from the partial route by using the "closest-city-next" heuristic. Each remote procedure call locks the work queue and checks if jobs are available. If the queue is locked or if no jobs are available, the remote procedure blocks. All slaves keep track of the shortest route that has been found, which is used to prune the search tree.

We show the results for a 12-city problem, in which the master creates 7920 partial routes of length five. A sequential C version of the code solved this problem in 12.4 seconds. As can be seen in Figure 2, the performance of the four systems is nearly identical up to 16 slaves. As the number of slaves increases, the master node (the server) becomes more loaded with requests. At 64 slaves the TRPC system becomes overloaded, and the performance drops dramatically. Because the ORPC and hand-coded system can handle requests faster, they both perform well at 64 slaves. At 127 slaves, the ORPC system is overloaded; as shown in Table 2, at 127 slaves the queue is almost always locked when the slaves request jobs, and all jobs end up aborting. The reason the hand-coded version does not suffer from this problem is that it's master generates all jobs before accepting any job requests; the master can therefore assume the queue will never be locked, and therefore never suspends. Indeed, if it did not generate all the jobs first, it would need to suspend occasionally, and then the code would be much more difficult to write, since there is no thread model associated with active messages.

# Slaves	# OAMs	Successes	% Successes
1	7355	7354	100.0%
2	7339	7337	100.0%
4	7335	7331	99.9%
8	7332	7324	99.9%
16	7186	7170	99.8%
32	7108	7076	99.5%
64	7069	7005	99.1%
127	7039	1	0.0%

Table 2: Percentage of Optimistic Active Messages that succeeded (executed without aborting) in the TSP application.

We conclude from this data that ORPC scales better than TRPC because of its reduced communication overhead at the master. At one slave, the live-stack optimization can be performed nearly 100% of the time; as the number of slaves increases, this percentage drops to 75% at 32 slaves, 20% at 64 slaves, and 0% for 127 slaves, which further increases the overhead of TRPC as compared to ORPC. We also conclude that ORPC scales almost as well as the hand-crafted Active Messages version of this program. Finally, we validate our assumption of optimism: up until 127 slaves the percentage of RPCs that abort is less than 1%.

4.2.3 Successive overrelaxation

Successive overrelaxation (SOR) is an iterative method for solving discrete Laplace equations on a grid. The grid is partitioned into subgrids. During each iteration the algorithm updates each non-boundary point of the subgrid with the average value of its neighbors. These updates are coded as remote procedure calls that store the boundary points of a subgrid into a buffer at the neighbor; if the buffer is not empty, the remote procedure blocks. After each iteration, all workers determine whether they have converged; when they have, the program terminates. To factor out the barrier cost from our experiments, we use two special hardware primitives of the CM-5 to implement the convergence test in both implementations: the global OR set and get pair, which provides a split-phase barrier, and a global reduction. Even though the RPC version of SOR uses threads, mutexes, and condition variables, it requires 15% fewer lines to express than the hand-coded AM version, in which the programmer synthesizes critical regions using polling and explicitly codes all the handlers, block transfers, and port allocations.

We present results for a problem size of 482 rows by 80 columns in Figure 3. The grid is partitioned along the row axis; neighboring threads exchange 80 double-precision floats every iteration for 100 iterations. The sequential C version of this program runs in 15.3 seconds. The difference between the two RPC systems is small

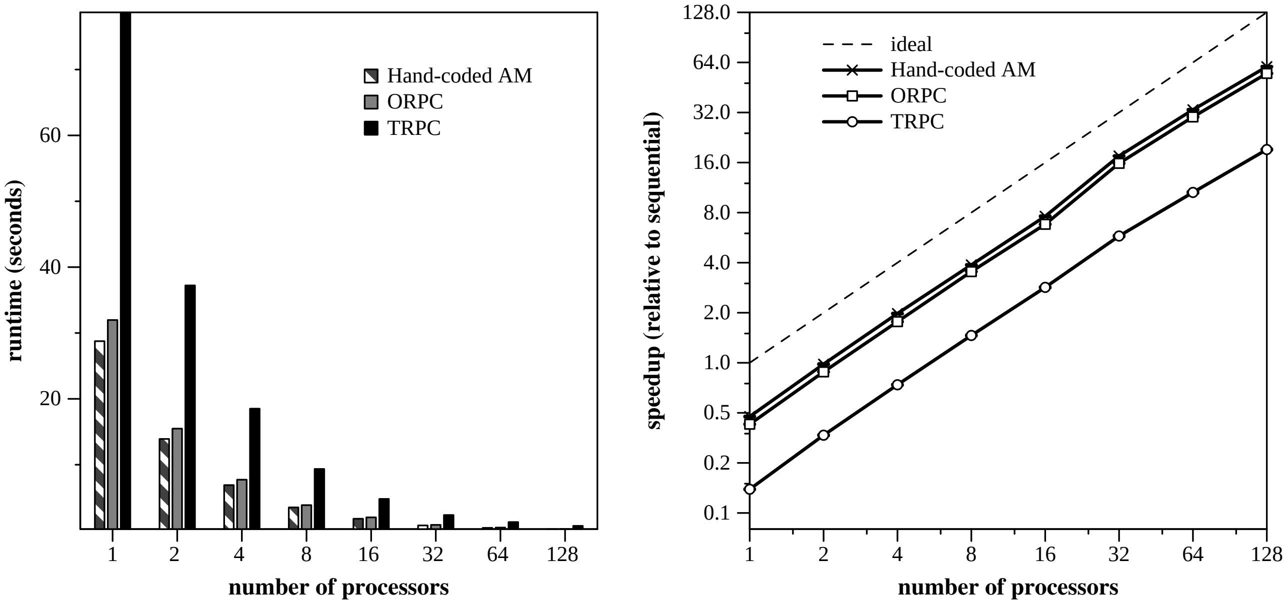


Figure 1: These graphs illustrate the performance of TRPC, ORPC, and hand-coded AM versions of the Triangle puzzle on a size 6 triangle.

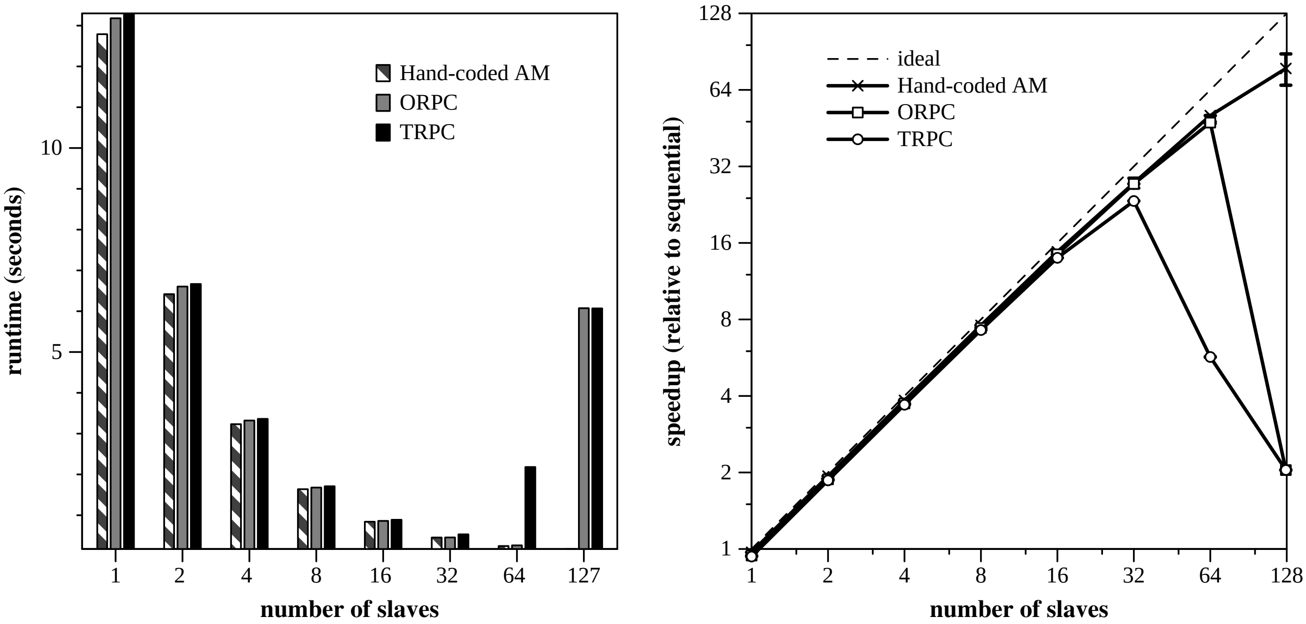


Figure 2: These graphs illustrates the performance of TRPC, ORPC, and hand-coded AM versions of the Traveling salesman problem on 12 cities, where the master process generates 7920 partial routes of length five.

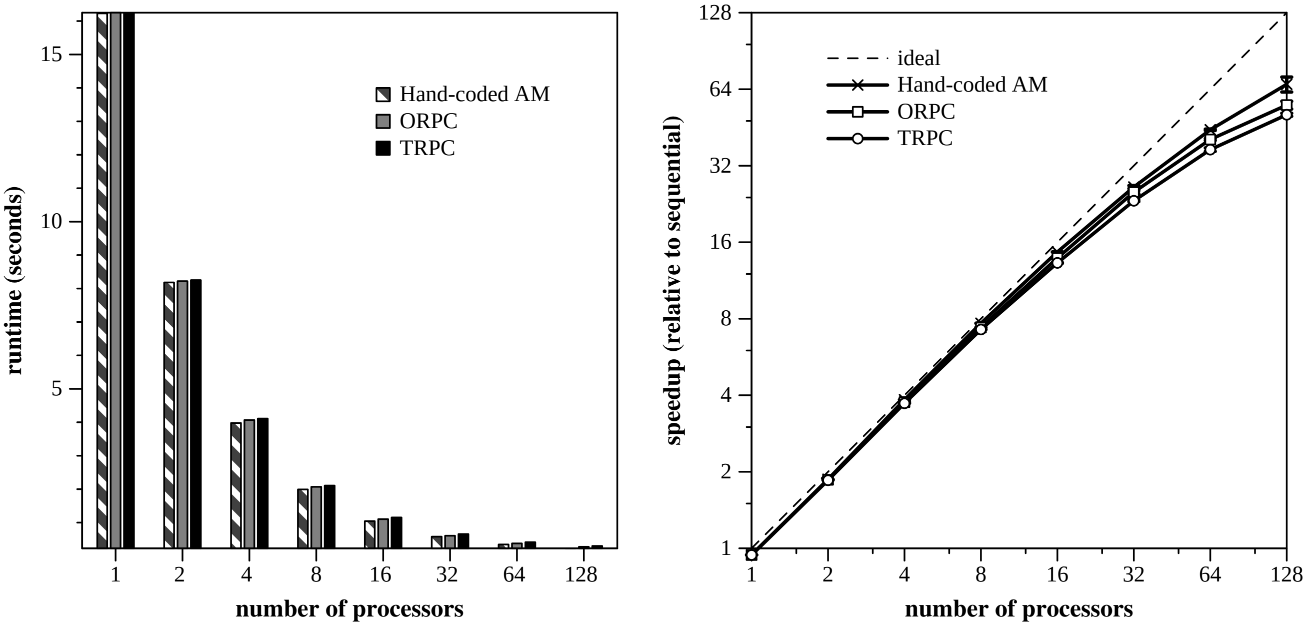


Figure 3: These graphs illustrate the performance of TRPC, ORPC, and hand-coded AM versions of Successive overrelaxation on a 482×80 grid.

(ORPC is 8% faster than TRPC at 128 processors), but consistent across different problem sizes. It happens that no Optimistic RPC aborts for any problem size for this application. The hand-coded Active Message version is faster because it avoids an extra data copy. In the Active Message version, the data is moved directly into the application's data structures; in the RPC versions, the data is first copied into a buffer to preserve RPC (call-by-value) semantics, and is then copied by the application into its own data structures. We could add an RPC with sender-specified destinations for data; a hand-generated version of such an RPC performs identically to the Active Message version. Note that both the Active Messages and the hand-generated RPC version require that the buffers be empty at the time of message arrival.

The performance difference between the ORPC and TRPC versions of the applications is small because data transfers dominate communication costs; at each iteration two 640-byte messages are exchanged between adjacent processors. This performance difference is exactly what we expect given the results from the microbenchmarks. More than 90% of all TRPCs cannot benefit from the live-stack optimization, and therefore cost 60 microseconds each more than the corresponding ORPCs; nevertheless, because data transfer time dominates communication costs, this effect is small. At smaller problem sizes, the absolute performance differences remain the same, but form a higher portion of the total runtime.

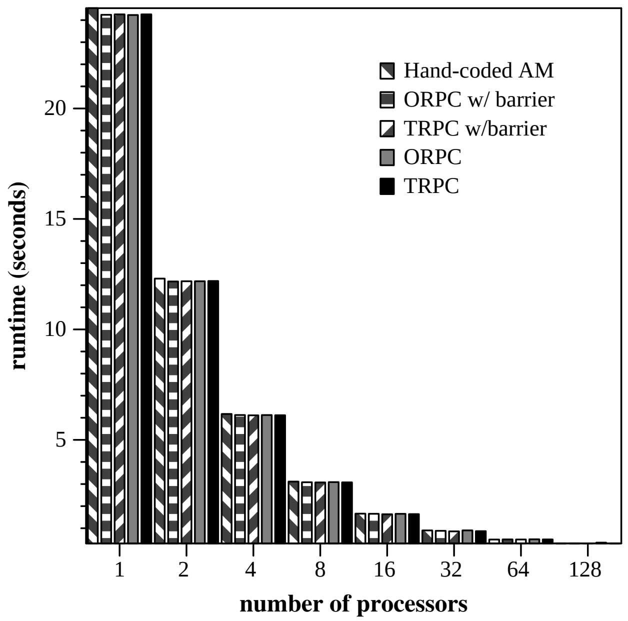
4.2.4 Water

Water is an n-body molecular dynamics application that "evaluates forces and potentials in a system of water molecules in the liquid state," as reported in the SPLASH parallel application suite description [22]. We use a message-passing version of the application that was written by John Romein of Vrije Universiteit for the Amoeba system [21]. This version is computationally equivalent to the SPLASH code; it computes the same result as the SPLASH version, and achieves nearly identical single-node performance. During each iteration two major communication phases occur (separated by local computation): in the first one, each processor sends the positions of every molecule it owns to every other processor; in the second one, each processor queues all of the updates to the accelerations of non-local molecules and sends out a single message containing those updates to the relevant processors (approximately half of them, depending on how many molecules were allocated per processor). The remote procedures that perform these operations can potentially block.

Our Active Messages implementation of this application assumes that when a processor receives a message it is ready to receive the incoming data, and thus will never need to block. If this assumption is incorrect, (*i.e.*, at a place where an OAM would abort), the program dies. In order to avoid blocking, the Active Messages version executes a hardware barrier between each iteration. This solution happens to avoid aborts in the experiments, but is not bulletproof. The correct solution, however, was too labor-intensive to implement, since we would have to duplicate much of the code in our thread package; this merely reemphasizes the utility of our thread package.

We ran a problem size of 512 molecules for five iterations. We discarded the time of the first iteration to discount cold-start cache effects, and averaged the other four iteration times. We follow the SPLASH report in ignoring the potential energy calculations in our measurements. The sequential C version of this program takes 24 seconds per iteration.

We present measurements of five implementations of this application: AM with barriers, ORPC, TRPC, ORPC with barriers, and TRPC with barriers. Because the size of messages sent in this application is large, the performance difference between these implementations is small. At 128 processors, all implementations are within 1% of each other, except that the ORPC implementa-



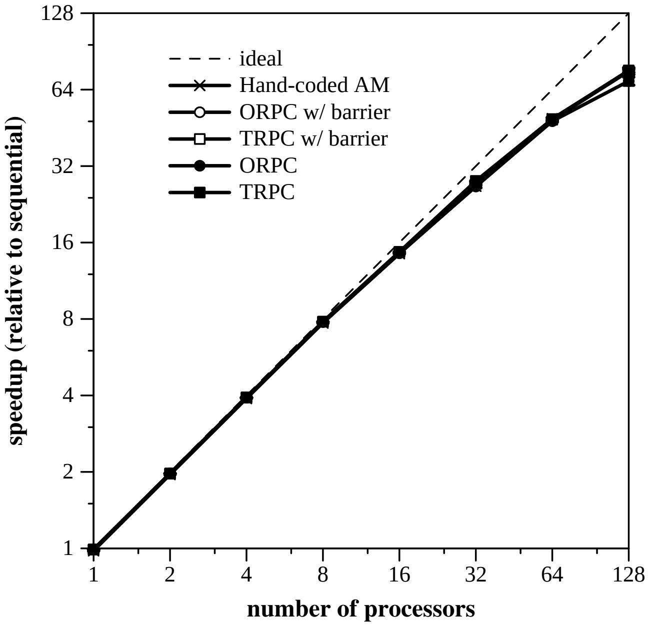


Figure 4: These graphs illustrate the performance of TRPC, ORPC, and hand-coded AM versions of Water running on 512 molecules. Data is presented for the RPC versions both with and without barriers between each time step.

# Processors	# OAMs	Successes	% Successes
2	24	24	100.0%
4	120	120	100.0%
8	528	528	100.0%
16	2208	2207	100.0%
32	9024	9006	99.8%
64	36480	36364	99.7%
128	146688	146084	99.6%

Table 3: Percentage of Optimistic Active Messages that succeeded (executed without aborting) in the Water application (no barriers).

tion without barriers is 10% slower than the rest (for reasons that we cannot yet explain). One problem may be that the ORPCs are running too long; aborting them and running them as TRPCs may improve performance. Another possibility is that of timer inaccuracy: we have determined that purely sequential computation is timed at up to 5% slower when there are messages in the network, even when the machine is running in dedicated mode. Since the ORPC implementation is probably leaving message in the network longer, even if these messages are not slowing down communication, the timers would unfairly penalize it. Adding barriers speeds up the ORPC version. This phenomenon on the CM-5 has been previously observed [5]. The ORPC implementation with barriers never aborts.

Our conclusions from this application are twofold. First, programming with TRPC and ORPC is easier than programming with Active Messages. Second, as can be see in Table 3, our assumption of optimism holds.

4.3 Summary of results

Using microbenchmarks we have shown that RPC, in the best case, can be competitive with AM. ORPC can bridge the performance gap between RPC and AM: ORPC achieves the performance of AM with the programming convenience of RPC.

We have also shown that for our applications the assumption of optimism holds: few Optimistic RPCs abort. Fine-grained applications that send many messages run up to three times faster with ORPC than with RPC. The ORPC implementations perform nearly as well as hand-crafted AM implementations, in which the programmer must be concerned with synthesizing critical regions and communication. Coarse-grained applications, or applications that send large messages, perform equally well whether they use ORPC, AM, or RPC.

5 Related work

This work generalizes the original Active Messages work of von Eicken et al. [27], which showed that software approaches to fast communication can be effective.

Alewife [1], the J-Machine [9], and Tera [2] are examples of hardware-intensive approaches to fast communication and multithreading. Our approach is a software alternative that does not depend on the existence of hardware thread contexts; Active Messages research indicates that little hardware is needed to integrate computation and communication. Mul-T [17] and CST [12], two high-level languages that were implemented on Alewife and the J-Machine, respectively, hide communication details and provide good performance by exploiting the special-purpose hardware on those machines.

In the Split-C compiler [7, 27], remote memory operations are directly compiled into Active Messages; the compiler synthesizes a primitive version of shared memory that provides fast remote references through Active Messages. Because Split-C programs are SPMD, and all communication can consist of only non-blocking remote references, the restrictions on handlers are not an issue.

In the Concert system [16], methods that are known to be non-blocking at compile-time are compiled to Active Messages. In addition, the Concert system uses compile-time and runtime techniques to select the cheapest alternatives for doing method dispatch. For blocking operations, Concert would benefit from using Optimistic Active Messages.

The ABCL/f language [23] compiles method invocations to a variant of Optimistic Active Messages. When a message handler is invoked, it executes on the stack; if it blocks, its frame is then copied to the heap. This frame copying is more expensive than abort in our system, and it requires more runtime support than our system does; our prototype compiler generates the necessary code for the continuation. ABCL/f uses this facility for local as well as remote invocation, which allows efficient fine-grained multithreading.

In the Prelude language [28], certain methods are always known to be short: these are methods that get and set fields within objects. Gets and sets on non-atomic objects (where methods do not implicitly acquire a lock) are compiled directly to Active-Message-like code; those on atomic objects are compiled to a form of Optimistic Active Messages.

Thekkath, Levy, and Lazowska propose separating control and data transfer in the structure of distributed systems by removing the remote procedure call abstraction and instead using only remote memory-reads and writes [24]. Our work, in contrast, shows that control transfer can be cheap; as a result, programmers do not need to sacrifice programmability for efficiency.

The Peregrine system [14] provides a highly optimized RPC package for distributed systems. It always uses a thread to implement remote procedure calls, but avoids some data copying by using the received packet buffer as the server's thread stack. Their RPC is still expensive: the software overhead for a null RPC in their system is 300 microseconds on SUN-3/60 workstations. However, their system also supports communication between address spaces, and relies on the kernel to perform communication; because the CM-5 provides user-level access to the network and uses strict gang scheduling, these overheads are not incurred in our system. van Doorn and Tanenbaum are exploring the use of Active Messages for objects and group communication in distributed systems [26].

6 Conclusion

We have presented the design and implementation of a system, Optimistic RPC (ORPC), that gives the programmer the convenience of RPC with nearly the performance of Active Messages. ORPC provides a fast thread library and a stub compiler that generates communication code from a high-level specification. The stub compiler employs a new runtime mechanism, Optimistic Active Messages, to conceptually allow arbitrary user code to run in Active Message handlers. Programmers do not have to be concerned with writing handlers for sending and receiving requests and replies, allocating ports for DMA, marshaling arguments, or restricting code executed in remote procedures. Implementations of some of our applications are 15% shorter than the corresponding Active Messages implementations, and still achieve similar performance.

We show the surprising result that in the best case AM is only 33% faster than RPC for short messages. For long messages, the cost causing this difference is amortized over the amount of data sent, since it does not vary with data size. Although our RPC is fast, it is not fast enough; Optimistic RPC can bridge the performance gap between RPC and AM.

We have measured the performance impact of RPC, ORPC, and AM for a number of applications. Optimistic remote procedure calls in our applications seldom abort. Applications that primarily communicate using large data transfers or are fairly coarse-grained perform equally well, independent of whether AMs, ORPCs, or TRPCs are used. For applications that send many short messages, however, the ORPC and AM implementations are up to three times faster than the TRPC implementations. Using Optimistic RPC, programmers obtain the benefits of well-proven programming abstractions such as threads, mutexes, and condition variables, do not have to be concerned with communication details, and yet can obtain the performance close to that of hand-coded Active Message programs.

Acknowledgments

We thank John Romein of Vrije Universiteit for providing us with a documented, debugged, message-passing version of Water that could be easily ported to our RPC systems. We thank Kevin Lew for providing us with the AM version of the Triangle Puzzle Problem. Our stub compiler is based on a C front end originally written by Michael Noakes and Eric Brewer. We also thank Robert Bedichek, Kevin Lew, Dawson Engler, Sandeep Gupta, and the anonymous referees for their useful comments.

References

[1] AGARWAL, A., BIANCHINI, R., CHAIKEN, D., JOHNSON, K. L., KRANZ, D., KUBIATOWICZ, J., LIM, B.-H., MACKENZIE, K., AND YEUNG, D. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (June 1995). To appear.

[2] ALVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. "The Tera Computer System". In *Proceedings of the International Conference on Supercomputing* (Amsterdam, The Netherlands, June 1990), pp. 272–277.

[3] BAL, H. E., KAASHOEK, M. F., AND TANENBAUM, A. S. "Orca: A Language for Parallel Programming of Distributed Systems". *IEEE Transactions on Software Engineering 18*, 3 (March 1992), 190–205.

[4] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming* (Santa Barbara, California, July 1995). To appear.

[5] BREWER, E. A., AND KUSZMAUL, B. C. How to get good performance from the CM-5 data network. In *Proceedings of the 8th International Parallel Processing Symposium (IPPS '94)* (Apr. 1994), pp. 858–867.

Also available as ftp://ftp.lcs.mit.edu/pub/supertech/papers/IPPS94-bandwidth.ps.Z.

[6] CHIEN, A. "Concurrent Aggregates (CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines". Tech. Rep. 1248, MIT Artificial Intelligence Laboratory, July 1990.

[7] CULLER, D., DUSSEAU, A., GOLDSTEIN, S., KRISHNAMURTHY, A., LUMETTA, S., VON EICKEN, T., AND YELICK, K. "Parallel Programming in Split-C". In *Proceedings of Supercomputing '93* (Portland, OR, Nov. 1993), pp. 262–273.

[8] CULLER, D., SAH, A., SCHAUSER, K., VON EICKEN, T., AND WAWRZYNEK, J. "Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine". In *Proceedings of the 4th Conference on Architectural Support for Programming Languages and Systems* (Apr. 1991), pp. 164–175.

[9] DALLY, W., FISKE, J., KEEN, J., LETHIN, R., NOAKES, M., NUTH, P. R., DAVISON, R., AND FYLER, G. "The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms". *IEEE Micro 12*, 2 (Apr. 1992), 23–39.

[10] ENGLER, D. R., ANDREWS, G. R., AND LOWENTHAL, D. K. Filaments: Efficient support for fine-grain parallelism. Tech. Rep. 93-13, University of Arizona, Apr. 1993.

[11] HORWAT, W. "A Concurrent Smalltalk Compiler for the Message-Driven Processor". Tech. Rep. 1080, MIT AI Lab, May 1988.

[12] HORWAT, W., CHIEN, A., AND DALLY, W. "Experience with CST: Programming and Implementation". In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation* (Portland, OR, June 21–23 1989), pp. 101–109.

[13] HSIEH, W. C., JOHNSON, K. L., KAASHOEK, M. F., WALLACH, D. A., AND WEIHL, W. E. Efficient implementation of high-level languages on user-level communication architectures. Tech. Rep. MIT/LCS/TR-616, Massachusetts Institute of Technology Laboratory for Computer Science, 1994.

[14] JOHNSON, D. B., AND ZWAENEPOEL, W. The Peregrine high-performance RPC system. Tech. Rep. TR91-151, Rice University, March 1991.

[15] KAASHOEK, M. F., WEIHL, W. E., WALLACH, D. A., HSIEH, W. C., AND JOHNSON, K. L. Optimistic active messages: Structuring systems for high-performance communication. In *Sixth SIGOPS European Workshop: Matching Operating Systems to Application Needs* (Wadern, Germany, Sept. 1994), pp. 23–28.

[16] KARAMCHETI, V., AND CHIEN, A. "Concert — Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware". In *Proceedings of Supercomputing '93* (Portland, OR, Nov. 15–19, 1993), pp. 598–607.

[17] KRANZ, D., JOHNSON, K., AGARWAL, A., KUBIATOWICZ, J., AND LIM, B. "Integrating Message-Passing and Shared-Memory: Early Experience". In *Proceedings of the 4th Symposium on Principles and Practice of Parallel Programming* (San Diego, CA, May 1993), pp. 54–63.

[18] LEISERSON, C., ABUHAMDEH, Z., DOUGLAS, D., FEYNMAN, C., GANMUKHI, M., HILL, J., HILLIS, W., KUSZMAUL, B., PIERRE, M. S., WELLS, D., WONG, M., YANG, S., AND ZAK, R. "The Network Architecture of the Connection Machine CM-5". Early version appeared in Proceedings of SPAA '92, Nov. 9, 1992.

[19] LEW, K., JOHNSON, K. L., AND KAASHOEK, M. F. A case study of shared-memory and message-passing implementations of parallel breadth-first search: The triangle puzzle. In *Proceedings of the Third DIMACS Algorithm Implementation Challenge* (Oct. 1994). Also available on the World Wide Web, http://www.pdos.lcs.mit.edu/ lew/papers/dimacs94.ps.

[20] MOHR, E., KRANZ, D. A., AND HALSTEAD JR., R. H. Lazy task creation: A technique for increasing the granularity of parallel pr. *IEEE Transactions on Parallel and Distributed Systems* (July 1991), 264–280.

[21] ROMEIN, J. W. Water — an n-body simulation program on a distributed architecture. Master's thesis, Vrije Universiteit, Amsterdam, August 1994. Also available as ftp://ftp.cs.vu.nl/pub/john/water/report.ps.gz.

[22] SINGH, J., WEBER, W., AND GUPTA, A. "SPLASH: Stanford Parallel Applications for Shared Memory". *Computer Architecture News 20*, 1 (Mar. 1992), 5–44.

[23] TAURA, K. "Design and Implementation of Concurrent Object-Oriented Programming Languages on Stock Multicomputers". Master's thesis, University of Tokyo, Feb. 1994.

[24] THEKKATH, C. A., LEVY, H. M., AND LAZOWSKA, E. D. Separating data and control transfer in distributed operating systems. In *Proceedings of the 6th Conference on Architectural Support for Programming Languages and Systems* (Oct. 1994), pp. 2–11.

[25] THINKING MACHINES CORPORATION. *"CMMD Reference Manual"*, version 3.0 ed. Cambridge, MA, May 1993.

[26] VAN DOORN, L., AND TANENBAUM, A. S. Using active messages to support shared objects. In *Sixth SIGOPS European Workshop: Matching Operating Systems to Application Needs* (Wadern, Germany, Sept. 1994), pp. 112–116.

[27] VON EICKEN, T., CULLER, D., GOLDSTEIN, S., AND SCHAUSER, K. "Active Messages: a Mechanism for Integrated Communication and Computation". In *Proceedings of the 19th International Symposium on Computer Architecture* (Gold Coast, Australia, May 1992), pp. 256–266.

[28] WEIHL, W., BREWER, E., COLBROOK, A., DELLAROCAS, C., HSIEH, W., JOSEPH, A., WALDSPURGER, C., AND WANG, P. "PRELUDE: A System for Portable Parallel Software". Tech. Rep. MIT/LCS/TR-519, MIT Laboratory for Computer Science, October 1991. Shorter version appears in *Proceedings of PARLE '92*.