Improving the Performance of Message-Passing Applications by Multithreading*

Edward W. Felten and Dylan McNamee
Department of Computer Science and Engineering
University of Washington
Technical Report 92-09-07

Abstract

Achieving maximum performance in message-passing programs requires that calculation and communication be overlapped. However, the program transformations required to achieve this overlap are error-prone and add significant complexity to the application program. We argue that calculation/communication overlap can be achieved easily and consistently by executing multiple threads of control on each processor, and that this approach is practical on message-passing architectures without any special hardware support. We present timing data for a typical message-passing application, to demonstrate the advantages of our scheme.

1 Introduction

The most natural way to program many scientific algorithms on message-passing machines is in the loosely synchronous, or coarse-grain data parallel, style. Such programs alternate between phases of pure calculation and pure communication. This approach is easy for the programmer to understand, so the resulting programs are relatively simple to debug and maintain [4]. Unfortunately, the performance of these programs suffers because they are unable to overlap communication with calculation.

Programmers have responded to this performance problem by restructuring their programs to allow greater communication/calculation overlap. This may involve rearranging computations so that communicated data are available earlier or are used later; it may also involve the use of nonblocking communication. These transformations tend to increase the complexity of programs, which makes understanding, debugging, and maintaining them more difficult.

Essentially, there is a choice between two programming models: the "blocking" model, which leads to simple programs, and the "nonblocking" model, which leads to complex programs that perform well.

In this paper we argue that the benefits of both programming models can be achieved by using multiple threads of control on each processor. Each thread is programmed in the blocking style, and a runtime scheduler switches between threads in order to overlap the calculation of one thread with the communication of others. Furthermore, we show this strategy is practical on ordinary message-passing hardware.

Multiple threads of control, managed in software at the user level, are a well-known technique for expressing concurrency on uniprocessors and shared-memory multiprocessors [1]. Thread systems also have been implemented on message-passing machines. However, we believe we are the first to propose the routine use of threads for scientific applications on multicomputers.

Processors like the transputer [8] provide hardware support for fast switching between threads of control; machines based on such processors are meant to be programmed in a multithreaded style. Our results provide evidence that hardware support for multithreading may not be necessary.

2 How to use threads

For loosely synchronous algorithms, threads can be used to emulate virtual processors. For example, a program running on p nodes may start two threads on each node, and have each thread emulate a single node on a $2p$-node machine. Since programs are typically structured to run on a variable number of nodes anyway, this should require only minimal changes to the program. Normal scheduling of threads by the runtime system will provide calculation/communication overlap.

We emphasize that these threads are not necessarily emulating the virtual processors found in some parallel programming systems. Such systems offer virtual processors

*This paper appeared in *Proceedings of the Scalable High Performance Computing Conference,* pages 84-89, April 1992.

for the programmer's convenience in expressing fine-grain parallelism; the virtual processors are then aggregated into larger units by a compiler. In contrast, our virtual processor emulation is intended to improve performance; we advocate using only a few virtual processors per physical processor, so the overhead of thread management is low.

For work-heap algorithms like parallel tree-search, each node can run several worker threads. While one worker is waiting for the central work-heap to give it more work, the others workers can run. This effectively hides the latency of requests to the work-heap.

3 Structure of the runtime system

Our runtime system manages multiple threads of control on each node of the message-passing machine; our implementation of these functions is similar to previous thread systems for uniprocessors and shared-memory multiprocessors [1]. The system provides primitives to create and start threads, and to synchronize threads on the same node using standard synchronization primitives like locks and barriers. Our runtime system also allows threads to communicate via message-passing, either between nodes or on the same node. We provide primitives to send and receive messages; these operations are blocking, but they block only the thread executing the communication. This allows us to overlap one thread's calculation with other threads' communication.

Our runtime system is called NewThreads. NewThreads runs on the iPSC/2, on networks of workstations, and on shared-memory multiprocessors. See section 6 for further description of NewThreads and its implementation.

4 Performance of multithreaded applications

We have measured the performance of an example multithreaded application on our 32-node iPSC/2. The application is a two-dimensional partial differential equation solver using relaxation on a five-point stencil. We measured the performance of three implementations of this algorithm: a standard message-passing program using Intel's NX primitives, a NewThreads program using one worker thread per processor, and a NewThreads program using two threads per processor.

Figure 1 shows the speedup of the PDE solver as a function of problem size. The NX and single-threaded NewThreads times are approximately equal, and the two-thread times are somewhat worse for small problems and significantly better for big problems. The NX and single-thread programs suffer a sharp drop in performance when

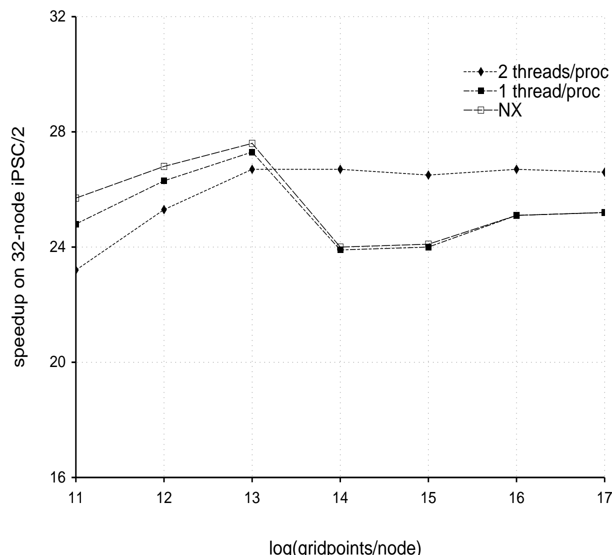

Figure 1: Performance of the PDE solver on a 32-node iPSC/2, as a function of problem size.

the entire dataset no longer fits in cache. The two-thread program seems immune to this cache effect. We hypothesize that the two-thread program has poorer locality of reference for small problem sizes, so in effect it has already suffered a drop in cache hit rates at problem sizes smaller than those on the graph. In any case, all three programs have low cache hit rates for the largest problem sizes; these data points show the true benefit of multithreading.

Cache effects complicate the evaluation of multithreading. We believe that the large cache effects observed in our experiments are due to the iPSC/2 architecture, and will not be as large on future machines. High-performance RISC processors [5, 6] tend to have small first-level caches, and the most important cache statistic is the hit rate in this first-level cache. Since these fast processors will be working on larger datasets, it seems unlikely that one processor's dataset will fit entirely in the first-level cache. Efficient use of small first-level caches will require support in the compiler for blocking of array operations [7]. With only a few threads per processor, the compiler should be able to block the operations of each thread, and achieve comparable hit rates for single-threaded and multithreaded programs.

It would be natural to expect the performance of the one-thread program to be identical to that of the NX program. The difference is due to extra data copying done in the current NewThreads implementation of remote message-passing: incoming NewThreads messages are buffered in kernel memory rather than being received directly into user-level buffers. This extra copy is unnecessary and could

be eliminated, which would further improve the relative performance of NewThreads benchmarks.

5 A performance model for multithreaded programs

We can construct a performance model in order to understand the performance implications of multithreading. For simplicity, we will restrict our discussion to loosely synchronous programs that have perfect load balance. We will also consider only the case with two threads per processor.

Each thread in the program executes an alternating sequence of calculation and communication phases. Each calculation phase requires time $calc(n)$, where n is the number of datapoints per thread. Each communication phase consists of a thread sending some constant number k of messages and then receiving k messages; each message is of length $comm(n)$. This model applies to our example program.

We model communication latency for a message of length λ as $\alpha_R + \beta_R \lambda$ for a remote message and $\alpha_L + \beta_L \lambda$ for a local message. We assume that remote message transmission can be overlapped with computation (if computation is available), but local communication cannot be overlapped since it uses the CPU. A scheduling operation that switches between threads costs time c. The total number of datapoints is N, and the program runs on p processors, requiring a total of $\phi(p)$ phases. For convenience, we will define $n = \frac{N}{p}$ to be the number of data points per processor.

For the PDE solver, $calc(n) = n$, $comm(n) = \sqrt{n}$, $k = 4$, and $\phi(p)$ is some large constant. The standard blocking program requires time

$$T_b(n,p) = n + 4\alpha_R + 4\beta_R\sqrt{n} \qquad (1)$$

per update.

With two threads per node, each thread has $\frac{n}{2}$ gridpoints and transmits and receives three remote messages and one local message. If the threads are scheduled correctly, the calculation of one thread overlaps perfectly with the remote communication latency of the other thread. Each phase is divided into two subphases in which one thread uses the CPU while the other is communicating. Thus, the time per subphase is the maximum of the CPU time of the first thread and the communication latency of the second thread. The time per update is

$$T_2(n,p) = 2\max\left\{\begin{array}{l} c + n/2 + \alpha_L + \beta_L\sqrt{n/2}, \\ 3\alpha_R + 3\beta_R\sqrt{n/2} \end{array}\right. \qquad (2)$$

The model makes three qualitative predictions. For small values of n the constant terms dominate, so the blocking program is better. For very large values of n, both methods tend toward n operations per update; in this limit computation time dominates everything else. For intermediate values of n, though, the multithreaded program is faster since the multithreading overhead can be amortized over a large number of operations, and the advantage of calculation/communication overlap is realized.

The observed data match these conclusions. The blocking program dominates for small problem sizes, but multithreading performs significantly better for larger problems. Our machine does not have enough memory to operate in the large-n region.

5.1 Modeling the optimal handcoded program

We also have a model for the "optimal handcoded" program that overlaps communication and calculation by rearranging the single-threaded program and using nonblocking messages. For clarity, we will present a simplified nonblocking algorithm that is not quite optimal, but performs as well as the optimal nonblocking algorithm for large problem sizes. We note that finding the truly optimal nonblocking program is quite difficult.

The handcoded program divides each update into two parts. In the first part, nodes simultaneously exchange boundary data (using nonblocking communication) and update their non-boundary points; in the second part boundary points are updated. The time per update is

$$T_{opt}(n,p) = \max\left\{\begin{array}{l} n - 4\sqrt{n}, \\ 4\alpha_R + 4\beta_R\sqrt{n} \end{array}\right\} + 4\sqrt{n}. \qquad (3)$$

The first term represents the update of the node's non-boundary points, overlapped with communication of edge data, and the second terms are the update of the boundary data.

It is interesting to compare the multithreaded program with the optimal handcoded program. Their performance is similar, but there are some extra overheads in the multithreaded program. First, the multithreaded program must switch between threads twice per update. Second, the multithreaded program initiates six remote messages per update, while the handcoded program sends four. (The total volume of data sent is the same.) Finally, the multithreaded program uses local communication operations, which are not strictly necessary. Context switches and local messages have been made inexpensive by careful implementation of the runtime system, and the effect of sending extra messages is small (most of the cost of these remote messages has been overlapped with calculation). Thus, while the handcoded program is significantly more complex than the multithreaded program, this added complexity does not lead to much difference in performance.

6 NewThreads

NewThreads, our runtime system, is implemented in C++, and consists of a collection of object classes that implement the basic abstractions of the programming system. NewThreads is not an operating system, since it is not designed to protect users from each other or to arbitrate access to shared resources; NewThreads merely provides runtime support to a single application program running within some operating system. It is entirely user-level code, and doesn't require kernel modification to run on any of its platforms[1]. A consequence of this implementation is that the system is easy to modify or extend to support specific needs of advanced users.

6.1 Threads

The Thread class defines a thread of control. A thread is started by giving it a procedure to execute and a set of arguments for that procedure. Some time later, the thread will acquire the CPU and start running. From time to time, the thread may block waiting for some event, and another thread will be scheduled. When the thread's start procedure returns, the thread is removed from service. A program initially has a single thread, running the `Main` procedure.

NewThreads normally uses nonpreemptive scheduling; this means that a thread runs until it finishes, blocks waiting for some event, or deliberately gives up the CPU to another thread.

6.2 Communication

NewThreads provides primitives to send and receive messages between threads on the same node, or on remote nodes. These operations are blocking, but they block only the thread that calls them; other threads on the same node can still run. Message-passing operations are designed to be very fast in the local case so the programmer doesn't have to worry about eliminating local message-passing operations.

6.2.1 Ports

A Port is a receptacle for messages. Messages are sent to ports, rather than to nodes or threads. Each port is named by a globally unique portID; a thread on any node can send a message to any port, regardless of location, given just that port's ID.

Since Ports are encapsulated as a C++ class, threads need a pointer to a Port object to invoke the member functions. Each node has a port table, which takes a portID and returns the corresponding Port, which may then be used to send messages to the port, or retrieve messages from the port.

From a single node's point of view, Ports are similar to message types in Intel's NX system. Each node in an NX program can expect to receive messages with different meanings; some message types may be synchronization events, and other types may be transmitting various subsets of an application's data. A NewThreads programmer would create a distinct port for each of these types of messages, and might optionally decide to use separate threads to handle the messages arriving on these ports.

6.2.2 Remote procedure call

Remote procedure call (RPC) [2] has been a popular and effective method of programming on distributed systems. For many purposes, RPC has advantages over direct message passing. RPC's semantics are already familiar to sequential programmers; an RPC invocation acts like a procedure call. RPC has been carefully studied, and very efficient implementation techniques are known. Finally, the machinery to make messages appear to the programmer as procedure calls can be automated through the use of stub generators.

RPC provides the programmer with the illusion of invoking a procedure on a remote node. In reality, the programmer is calling a local "stub" procedure that copies its arguments into a message buffer and sends them to the remote node. The message is delivered to a server thread on the remote node which unpacks the arguments and calls the procedure. When the remote procedure finishes, its return values are packed into a message and sent back to the originating thread, which then returns from the stub procedure.

An important advantage of RPC is that all the details of the implementation are hidden from the programmer. The programmer simply provides a list of procedures which can be remotely invoked, along with the types of their arguments and return values. A "stub compiler" then generates all the code to manage the message-passing, packing and unpacking of message buffers, creation of the server thread, and so on.

We have implemented RPC in NewThreads. Programmers may use RPC for communication that requires a request/response pattern, or they may use RPC more directly to implement the abstraction of a shared object providing a service to a set of distributed threads. We do not advocate using RPC for all communication, but we do believe that RPC is a useful addition to the standard send/receive facilities.

[1] NewThreads currently runs on the iPSC/2, Sequent Symmetry, and networks of DECstation 3000 series workstations.

6.3 Naming Service

We have implemented a naming service built on top of RPC. Using the naming service, a programmer can associate a name string with a portID. Other threads can look up the portID associated with that name and commence communication with that port. Using ports and the naming service makes it easy to dynamically determine the location of a thread (or a set of data) rather than hard-wiring the location into the program.

For example, a thread that manages the data for an object called LocalSum1 can advertise this to the other threads in the job by calling `rpc_nameRegister("LocalSum1",portID)`. A thread on any node can find the portID of the thread managing this data by calling `id = rpc_nameLookup("LocalSum1")`. A result of the location transparency of Ports is that the programmer can balance the load of the computation by placing any port on any node in the system.

6.4 Synchronization

NewThreads provides basic synchronization primitives between threads on the same node, including Locks, Conditions, Monitors and Barriers. In addition, the programmer can use messages to synchronize local threads.

6.5 Performance of thread primitives

In NewThreads, thread creation and destruction, thread context switch, thread synchronization, and message passing between threads on the same node, all are extremely efficient. As an illustration, the cost of a create/destroy pair (creating a thread which immediately destroys itself when it runs) in the iPSC/2 implementation of NewThreads is 0.135 milliseconds, whereas the cost of the analogous operation involving an NX process on the same hardware is 250 milliseconds — more than three orders of magnitude slower. Message passing between nodes in NewThreads on the iPSC/2 uses the underlying NX primitives so it is no faster (and in fact is marginally slower). The key advantage of NewThreads, though, is that the dramatically improved performance of most operations makes it possible to use multiple threads per node to efficiently overlap the calculation of one thread with the communication of others, yielding maximum performance in the context of a simple programming model.

7 Implications for kernel structure

As Anderson [1] has argued, thread operations are inherently cheaper when implemented at the user-level rather than in the kernel. Intel's NX kernel, like most multicomputer operating systems, provides no particular support for user-level threads. Although NX is flexible enough to allow implementation of our runtime system, the kernel structure could be changed to better support the needs of systems like ours. The most important change would be to allow threads to block in the kernel without blocking the entire process. Kernel support for blocking message-passing operations by threads would lower the overhead of our message-passing operations substantially.

Supporting user-level scheduling in the presence of unpredictable events like message arrival or page faults presents a set of problems that have been described by Anderson [1]. Anderson's solution, called scheduler activations, solves these problems and would provide a suitable mechanism for a multicomputer kernel. Scheduler activations also provide a fast and flexible mechanism for delivering messages to the application. A project to integrate scheduler activations into the Mach 3.0 microkernel is currently underway in our department. Since Intel's Paragon system [3] will run an operating system based on Mach 3.0, we are optimistic that this mechanism can be made available.

8 Conclusions

Overlapping calculation and communication is essential if message-passing applications are to achieve maximum performance. This overlap can be achieved by application-specific program restructuring, but this is error-prone and leads to poorly structured programs. Multithreading is a general and natural way to express programs with calculation/communication overlap. Writing the natural "blocking" program with more than one thread per node provides both good structure and the performance benefits of overlap.

Supporting multithreading in software imposes some overhead, but this overhead can be made very small. We have demonstrated that for a typical scientific algorithm, a multithreaded program performs significantly better than an equivalent single-threaded program.

Multithreading has other advantages, including improved load balance. By putting several threads with uncorrelated loads on the same physical processor, we can even out fluctuations in load, and achieve better performance.

9 Acknowledgments

The authors are grateful to Tom Anderson, Jeff Chase, Alex Klaiber, Ed Lazowska, Hank Levy, Rik Littlefield,

Steve Otto, Burton Smith, David Walker, and John Zahorjan for useful conversations and comments on earlier drafts of this paper. We also thank the early users of NewThreads for their feedback.

This material is based upon work supported by the National Science Foundation (Grants DCR-8352098, CCR-8619663, and CCR-8703049), the Washington Technology Center, Digital Equipment Corporation (the External Research Program and the Systems Research Center), and Apple Computer Company. Felten was supported in part by a Mercury Seven Fellowship and an AT&T Ph.D. Scholarship.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 95–109, 1991. Also in *ACM Transactions on Computer Systems* (February 1992).

[2] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comp. Sys.*, 2(1):39–59, Nov. 1984.

[3] Intel Corporation. Paragon XP/S product overview, 1991.

[4] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors*. Prentice Hall, 1988.

[5] Intel Corp. *i860 64-bit Microprocessor Programmer's Reference Manual*, 1990.

[6] Gerry Kane. *MIPS R2000 RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1987.

[7] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[8] Colin Whitby-Strevens. The transputer. In *Proceedings of 12th International Symposium on Computer Architecture*, pages 292–300, 1985.