MIMD-Style Parallel Programming Based on Continuation-Passing Threads

Michael Halbherr* Yuli Zhou* Chris Joerg*

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 01239

Appears in *Proceedings of 2nd International Workshop on Massive Parallelism*, October 1994, Capri Italy.

ABSTRACT

In this paper we present Threaded-C, a multi-threaded parallel programming package based on threads in explicit continuation-passing style, designed for current message-passing massive parallel architectures. It incorporates two effective software mechanisms targeted towards lengthening sequential threads in order to offset the costs of dynamic scheduling, and towards preserving the locality of computations to reduce the network traffic. It is currently implemented as a C language extension along with a runtime system implemented on the CM-5 that embodies a work stealing scheduler. Real world applications written in this package, such as ray-tracing and protein folding, have shown impressive speedup results.

1. Introduction

Multi-threaded computation models have typically been proposed as a general solution to exploit dynamic, unstructured parallelism. Due to the relative high cost of communication, however, it is critical to preserve the computation locality on current parallel architectures. In addition, thread scheduling and synchronization adds extra overhead at runtime, which must be offset by using reasonably long threads.

Several research machines, such as the HEP[7], the Monsoon dataflow system[14], or the forthcoming Tera machine[1], are designed to overcome these problems in hardware. They provide highly integrated, low overhead, message interfaces as well as hardware support for scheduling and synchronization. Disregarding the debate of whether such

*Research performed at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310.

machines are commercially or technically viable, however, the problem of programming most of the current parallel machines, which have no special hardware support for multi-threading, still remains.

The alternative is to seek solutions in software. The implicit parallel languages Id[11] and Sisal[9] simplifies the programming task by providing a higher level abstraction to the programmer, while relying on the compiler to partition the program into sequential threads. The compiler technology, however, is yet unable to generate long enough threads, and the question of scheduling to preserve locality still needs to be solved. TAM[5] was designed to address these difficulties, although the solution it proposes is insufficient, it was one of the main motivations for our work.

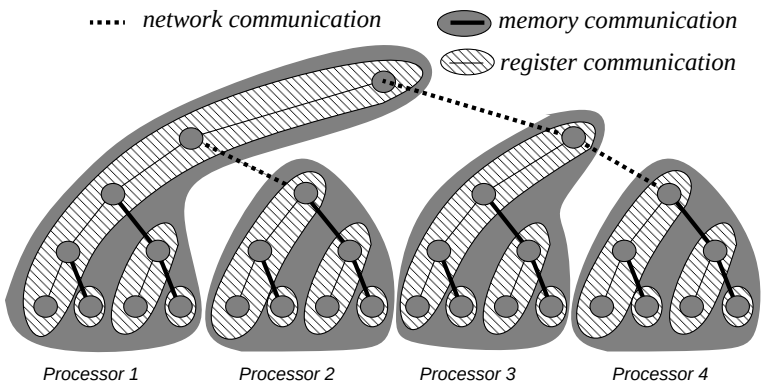

Figure 1: maximizing communication locality

Threaded-C provides an abstraction of threads in continuation-passing style (CPS), making it easy to implement automatic and programmer controlled optimizations to solve the afore-mentioned problems. The current implementation is built solely on top of active messages[17] on CM-5, using a new implementation in the Strata package[4].

Two software mechanisms are build into Threaded-C: First, the scheduler uses work stealing as its default policy, creating patterns of locality in which threads can pass arguments through local memory rather than across the network most of the time; Second, a thread can transfer control to another thread, thus completely avoiding scheduling overhead.

We show the effect of these optimizations in Figure 1. As this diagram clearly shows, there are three communication levels, namely register communication, memory communication and network communication. Whenever possible, a Threaded-C program arranges for most of data transfer to occur within the two faster levels, providing an optimal compromise between increasing the sequentiality of the application for better locality and exposing enough parallelism to enable good load balancing.

2. The Parallel Continuation Machine

PCM is the runtime system for Threaded-C. A Threaded-C program consists

of a collection of *threads*. Statically, a thread is just a sequence of instructions. At runtime, arbitrary numbers of dynamic instances of a thread can be created, each with its own set of arguments. PCM implements the necessary mechanisms for dynamic scheduling of threads and their communication/synchronization. The key elements of PCM's execution environment are illustrated in Figure 2 and explained in the following.

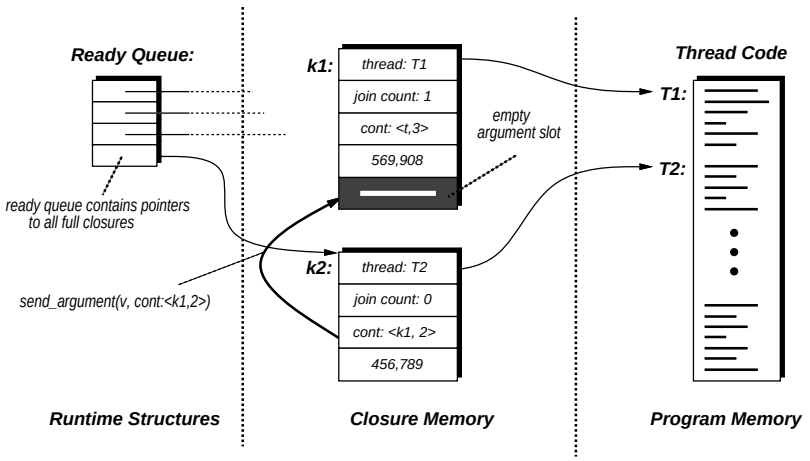

Figure 2: Elements of the PCM model

New instances of threads are created by making closures, which act as the intermediary between the application code and the runtime system. A closure is a data structure containing a pointer to the thread code and all the arguments needed to execute the code. A closure is ready to execute if it has obtained all the arguments, such full closures are passed to the scheduler, which keeps them in a *ready queue* and will later schedule them for the encapsulated threads to run.

In order to enable points of synchronization, a closure can be created with some arguments missing. These arguments will be supplied by other threads in the future. A thread sending an argument to a non-ready closure must first obtain a reference to where the argument is to be sent; Such references are called *continuations*, which is just a pointer to a closure plus an integer offset into the closure. A closure have an additional slot called the *join counter*, whose value indicates how many arguments are missing. No closure is given to the scheduler until the join counter reaches zero.

At runtime, PCM enqueues a closure for the first thread on a specific processor. All processors then enter the scheduling loop, executing threads off its local ready queue. Full closures may migrate across the processors either by explicit remote posts or automatically by the scheduler. In this respect PCM is extremely flexible in accommodate various scheduling policies. In practice, we have found that using the ready-queue locally as a stack, combined with demand based migration of the oldest closures form a reasonably general heuristic that is both time and space efficient (the scheduling issue will be discussed further in section 4). The program is terminated

by the last thread, which sends signals to all processors causing them to exit the scheduling loop.

3. Threaded-C

Threaded-C is implemented as an extension to C. A Threaded-C program consists of threads, marked by the specifier **thread**, and normal C functions. A preprocessor expands the threads into C functions, and copies C code literally. The resulting C program is then compiled and linked with the PCM runtime library. Figure 3 shows the Threaded-C program that computes the Fibonacci function.

```
thread fib_fork (cont k, int n) {
    if (n<2) send_argument (n, k);
    else {
        cont s1, s2;
        new_thread (fib_sum, k, ?s1, ?s2);
        new_thread (fib_fork, s1, n−1);
        new_thread (fib_fork, s2, n−2); }
}

thread fib_sum (cont k, int x, int y) {
    send_argument (x+y, k);
}
```

Figure 3: PCM program to compute Fibonacci

We note the continuation-passing style of threads, instead of returning to a caller, each thread in the example is explicitly passed a continuation, to which it sends an argument. To enable global communication, the closure pointer inside continuations contains the ID of the processor that owns the closure.

The preprocessor expands a thread, for example `fib_fork`, into a C function that takes a closure, and fetches from it all specified arguments before starting actual execution. Within a thread, predefined PCM primitives can be used in addition to normal C statements. The primitive new_thread allocates a closure initialized with all available arguments. An outstanding argument is indicated by `?x`, with `x` set to the continuation pointing to the empty argument slot. The preprocessor calculates the number of outstanding arguments and initializes the join counter. It will automatically post the closure if the initial join count is 0. The primitive send_argument sends an argument along the continuation. On receiving the argument the closure's join counter is decremented by 1, and the closure is posted if it reaches 0.

4. Scheduling Threads on a Multiprocessor

We first introduce a cost model for PCM, then describe briefly the work stealing scheduler implemented on CM-5 aimed at minimizing the scheduling and communication overheads. In the following discussion we will ignore the tail-call mechanism, whose effect is relatively obvious.

Assuming that at any time each processor is either executing a thread or one of the overhead scheduling task, as described below, we can reduce the analysis to that of the time spent on an average thread. The parallel efficiency ϵ will be defined as the fraction of the overall execution time actually performing useful computation, as shown in Equation (1).

$$\epsilon = \frac{R_t}{M_c + k \cdot (S_l + \pi_1 \cdot T_a) + (P_c + \pi_2 \cdot T_c) + S_c + R_t} \tag{1}$$

In the equation above, M_c ($\approx$ 10 cycles) is the time of allocating a closure; S_l ($\approx$ 10 cycles) is the time of sending an argument locally, and T_a ($\approx$ 100 cycles) is the additional transport time of sending an argument globally; P_c ($\approx$ 10 cycles) is the time of posting a closure locally, if migrated, an additional transport time of T_c ($\approx$ 500 cycles) is incurred; S_c ($\approx$ 15 cycles) is the control transfer time from scheduler to a thread and back; R_t is the average run length of a thread; k is the average thread arity.

There are two important ratios in the equation, reflecting the overall frequency of global send arguments (π_1) and closure migration (π_2). Specifically, π_1 is the fraction of arguments that have to be sent across the network, π_2 is the fraction of closures that migrate from one processor to another. Assuming a typical value of $2\pi_1 = \pi_2 = \pi$ and $k = 2$, we transform Equation (1) into

$$\epsilon = \frac{R_t}{55 + 600\pi + R_t} \tag{2}$$

As evident from Equation (2), it is imperative to reduce global communication frequency indicated by π. To achieve minimal value for π we implemented a lazy scheduling policy known as *work stealing*, which migrates closures to idle processors on demand (the parallel language Mul-T[10] implements work stealing in a different runtime environment). In addition, processors use their ready-queues in LIFO order, this heuristic results in lower resource requirements for most computations. Steal requests, on the other hand, are served in FIFO order, which greatly reduces migration frequencies for computations structured with recursion. Both examples considered in the remainder of this paper typically migrated less than 1% of all dynamically created closures, Recent theoretical results[3] also show that breadth-first stealing achieves linear speedup in the presence of modest parallel slackness.

5. Two Case Studies

5.1. Parallel Ray Tracing

Several characteristics of parallel ray-tracing makes it an ideal application for Threaded-C: First, the task of tracing a complicated picture with a reasonable size, like the one shown in Figure 4, contains enough parallelism to justify the use of a parallel processor. Second, the considerable variance in the amount of processor cycles required to trace individual rays, due to the acceleration method employed, necessitates the use of a dynamic load balancing schema to guarantee acceptable utilizations and to ensure scalability. Third, the ray tracer can be packaged into threads with grain-sizes coarse enough to offset the overhead introduced by the dynamic scheduler.

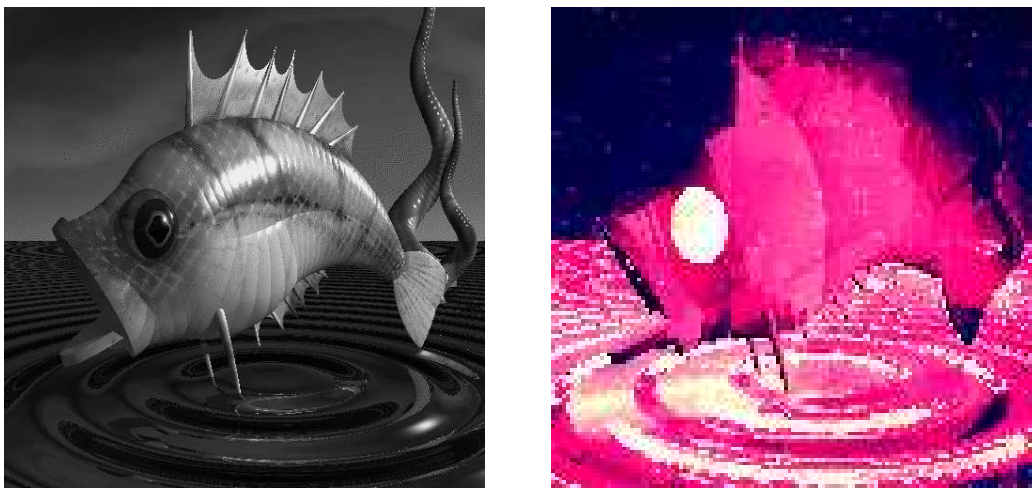

Figure 4: Traced picture and work histogram

We transformed the POV-Ray program, which has at its core a simple doubly nested loop iterating over each pixel in a two dimensional display window, into a recursion that split the window into smaller subwindows until the size reaches a single pixel. It then simply calls the original sequential trace function with the pixel coordinates of the unit sized subwindow. Although the original package is fairly large (~26,000 lines of C), only the control loop needs to be parallelized, resulting in very little modification.

The right side of Table 1 shows the statistics collected for runs of the Threaded-C program on different machine sizes. There is no measurable difference between the runtime of the original sequential program and that of the multi-threaded program running on one processor. This is a result of the rather coarse thread granularity of the parallel ray tracer.

For comparison, the left side of the table shows the speedup numbers obtained from an implementation which uses a static load balancing scheme. The static algorithm employs a simple work distribution that assigns exactly the same number of rays to each node. Two timings are listed: the execution time of the fastest processor and the execution time of the slowest processor. In this case, even in two processor configuration the slower processor already requires 64% more compute cycles than

Table 1: Ray tracing results, times are in seconds

	static load distribution			dynamic load distribution			
procs	min time	max time	pixels	time	max pixels	min pixels	impr.
1		1590	262144	1590	262144		
2	602	988	131072	795	135312	126832	24%
4	236	523	65536	396	79073	48867	32%
8	101	257	32768	189	45175	21464	35%
16	46	128	16384	90	22616	10711	43%
32	23	70	8192	46	12543	5859	52%
64	10	36	4096	23	7210	2595	56%

the faster processor, and the gap widens as we increase the number of processors. The right side picture of Figure 4 show the uneven work distribution per pixcel of the ray-tracing program. In this histogram brighter points represent higher workloads.

Two additional columns under the dynamic load distributions shows the variance of number of pixels traced per processor, which reflect the effect of the dynamic load balancer. The linear speed up over all machine configurations is the result of both the low overhead introduced by the parallelization and the sufficient distribution of the ample parallelism across the available processors.

5.2. Protein Folding

The protein folding code[13] was developed together with Vijay Pande of the Center for Material Sciences and Engineering at MIT. A protein is described as a chain of monomers, where each monomer may represent a number of amino acids. In their work[12], the lattice model[15] was used, which assumes each monomer to sit on a lattice point. Each possible folding of the polymer can then be described as some path through the set of all lattice points, where each point is traversed exactly once. A polymer of length 26 is shown in Figure 5, with each shade representing a different type of monomer. Depending on their type, two neighboring monomers will exert some attractive or repulsive force on each other. The energy of a folded polymer is the sum of the forces between all of the neighboring monomers. The problem is then to find a folding with the minimum energy. It is necessary to check all possible foldings of a given polymer and compute a histogram of the energy values.

The heart of the Threaded-C program is a search algorithm that finds all possible Hamiltonian paths in a cube. Each time a path is found the energy value for that folding is calculated and a result histogram is updated with the value. The parallel implementation recursively creates a new thread for each possible extension of the current path, passing the set of already visited lattice points along as a arguments. Each of the newly created threads can then be freely scheduled by the PCM scheduler.

Many variations of this program have been run on a range of problem and machine

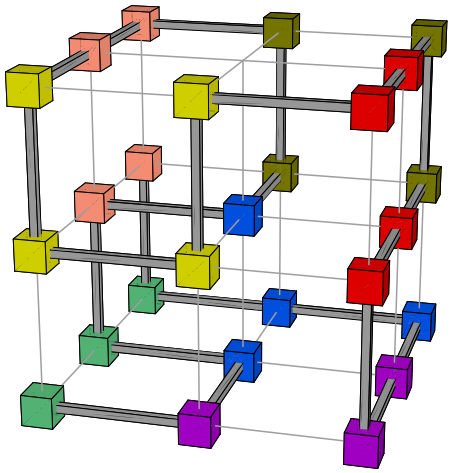

Figure 5: A folded polymer

sizes. Results for two problem sizes are shown in Figure 6. The figure on the left shows a smaller problem instance, namely a $3 \times 3 \times 3$ cube which has 103,346 paths. The figure on the right shows a larger problem instance, namely a $3 \times 3 \times 4$ cube which has over 48 million paths. In both figures the horizontal line shows the execution time for the optimized sequential code. The sloped line shows the execution time of the multi-threaded code on various machine sizes.

Again, we are interested in the overhead introduced by the parallel execution model and in the speedup behavior of the protein folding code. When running on one processor the PCM program was 14 and 22 percent slower than the sequential version for the two examples; Otherwise the speedup curves are almost perfectly linear for both problem instances studied.

6. Conclusions and Future Work

We have presented Threaded-C and its supporting runtime system PCM, and demonstrated how it is used to parallelize existing sequential applications via two real life examples. The continuation-passing style of the threads in Threaded-C has enabled a simple yet very flexible execution model, allowing both automatic and programmer controlled optimizations of scheduling policies to maximize the communication locality within the program, which is the key to obtain good parallel speedup on a machine like CM-5.

In the meanwhile, we have excluded global data structures in order to study dynamic scheduling in a simple execution model. With the addition of global data structures, dynamic scheduling is not always applicable, instead two key issues need to be addressed on a more application specific level: First, data structures need to be evenly distributed across all processors; Second, they need to be aligned with the computation. Data structures and computational threads can be aligned either

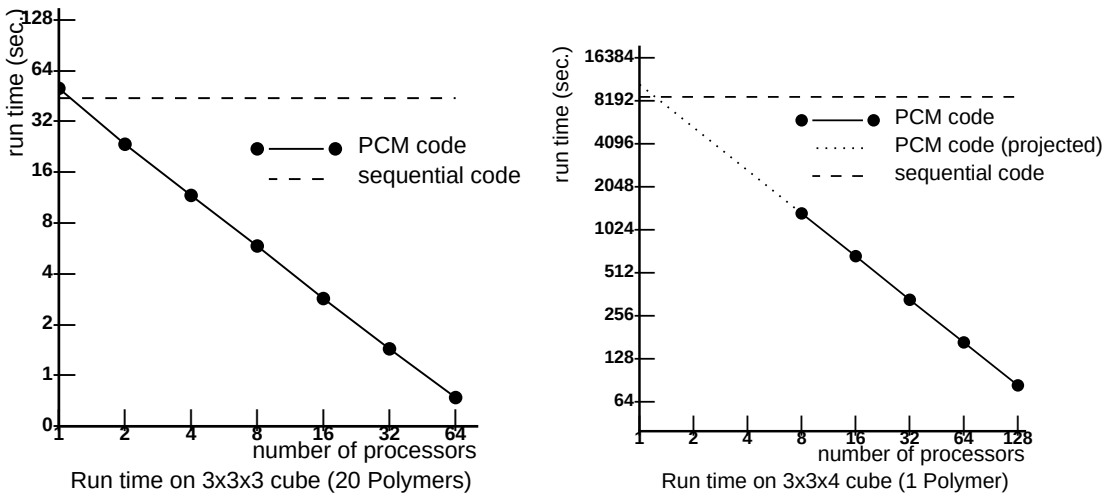

Figure 6: Run times for protein folding

by moving the data structure, as done by architectures based on globally coherent caches[8] [6], or by migrating the computation threads. Our preliminary results show that both techniques are needed. However, no general solution has yet surfaced.

7. Acknowledgement

The authors wish to thank Bobby Blumofe, Charles Leiserson and especially Ken Jones for helpful comments on earlier drafts of the paper.

8. References

1. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proc. of International Conference on Supercomputing*, pages 1–6, 1990.
2. Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. on Computers*, 39(3):300–318, March 1987.
3. Robert Blumofe and Charles E. Leiserson. Work-Stealing Algorithms for Scheduling Multithreaded Computations. Manuscript in progress.
4. Eric Brewer and Robert Blumofe. *Strata Reference Manual.* Lab for Computer Science, MIT, version 2.0 edition, January 1994.
5. D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. of the 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa-Clara, CA, April 1991.

6. Kendall Square Research. *KSR1 Technical Summary*, 1992.
7. J. Kuehn and B. Smith. The Horizen Supercomputer System: Architecture and Software. In *Proc. of Supercomputing '88*, Orlando, Florida, November 1988.
8. D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *The 17th Annual International Symposium on Computer Architecture Conference Proceedings*, Seattle, Washington, May 1990.
9. J.R. McGraw, S.K. Skedzielewski, S.J. Allan, R.R. Odledhoeft, , J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. Sisal: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2. Technical report, Lawrence Livermore National Laboratories, Livermore CA, March 1985.
10. E. Mohr, D. Kranz, and R. Halstead. Lazy Task Creation. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):264–280, July 1991.
11. R.S. Nikhil. Id Language Reference Manual. Computation Structure Group Memo 284-2, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139, July 1991.
12. Vijay Pande, Alexander Yu, Grosberg, and Toyoichi Tanaka. Thermodynamic Procedure to Construct HeteroPolymers that can be Renatured to Recognize a Given Target Molecule. *submitted to Nature*.
13. Vijay Pande and Christopher Joerg. Enumeration of the Hamiltonian Walks on a Cubic Sublattice. To appear in *Jounal of Physics: A*, 1994.
14. G.M. Papadopoulos and D.E. Culler. Monsoon: an Explicit Token-Store Architecture. In *The 17th Annual International Symposium on Computer Architecture Conference Proceedings*, Seattle, Washington, May 1990.
15. E. Shaklmovich and A. Gutin. *J. Chem. Phys.*, 93:5967, 1990.
16. G. L. Steele and G. J. Sussman. LAMBDA, the Ultimate Imperative. AI Memo 353, MIT AI Lab., March 1976.
17. T. von Eicken, D.E. Culler, Seth C. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *In The 19th Annual International Symposium on Computer Architecture Conference Proceedings, Australia*, May 1992.