LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIMD-Style Parallel Programming Based on Continuation-Passing Threads

Computation Structures Group Memo 355
March 7, 1994

Michael Halbherr, Yuli Zhou and Chris F. Joerg

This report describes research done 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.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

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

Abstract

Today's message passing architectures are characterized by high communication costs and they typically lack hardware support for synchronization and scheduling. These deficiencies present a severe obstacle to obtaining efficient implementations of parallel applications whose communication patterns are either highly irregular or dependent on dynamic information.

In this paper we present a model based on continuation-passing threads in which we try to overcome these difficulties. The model 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. The model 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

This paper presents the parallel continuation machine (PCM), a dynamic execution model creating patterns of locality which are necessary to harvest the computing power of today's message passing architectures. We will first concentrate on explaining the key ideas underlying the implementation, and then demonstrate how they give rise to extremely efficient parallel programs via two real-world examples.

Parallel programs can be classified along several dimensions, such as grain-size, communication regularity, and whether the execution depends on runtime data. We believe

*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.

that existing programming models, such as data parallel programming and explicit message passing, have been successful in addressing the needs of programs with simple static communication patterns. For these programs it is usually possible to carefully orchestrate communication and computation to statically optimize the overall performance.

On the other hand, it proves far more difficult to find static solutions leading to high machine utilizations for parallel applications whose communication patterns are either highly irregular or dependent on dynamic information. In this paper, we are mostly interested in investigating the needs and characteristics of these classes of programs, which must rely on runtime mechanisms to enable efficient solutions.

Multi-threaded computation models have typically been proposed as a general solution to exploit dynamic, unstructured parallelism. In such a model, dynamically created instances of sequential threads of execution cooperate in solving the problem at hand. To efficiently execute such an application, it is necessary to have efficient runtime thread placement and scheduling techniques. Although this general thread placement problem is known to be NP-hard [9], it is possible to implement schedulers based on simple heuristics, achieving good machine utilizations at reasonable cost. These heuristics usually work well for a broad class of applications, making it possible to implement the scheduling and placement task as a fairly generic service that resides at the core of the runtime system.

Unfortunately, current parallel architectures present severe obstacles to achieving acceptable results for multi-threaded applications. These machines typically consist of a collection of computing nodes that communicate with each other through a message-based communication network. The individual RISC-like nodes have been optimized for sequential computations with a high locality of reference, and the cost of network accesses are often one or more orders of magnitude more expensive than local memory accesses. Multi-threaded programs, however, often require frequent global communications. In addition, by exposing multiple threads a parallel program tends to both lose locality and incur extra runtime overhead in synchronization and scheduling.

Several research machines, such as the HEP [12], the Monsoon dataflow system [19], or the forthcoming Tera machine [2], 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 machines are commercially or technically viable, the problem of programming most of the current parallel machines, which have no special hardware support for multi-threading, still remains.

We are thus left with the only alternative solution, namely software. The programming challenge, in view of the above difficulties, is to minimize network communication and to provide longer sequential threads to offset the runtime scheduling and synchronization overhead.

The static set of sequential threads making up the multithreaded program can either be generated *implicitly* by a sophisticated compiler, or *explicitly* by the programmer. Programming languages advocating the implicit style, such as Id [16] and Sisal [14], usually take a high level description of the actual problem, extract the available parallelism from the declaration and partition it into sequential threads. While implicit programming languages simplify the programming task, they usually fail to produce threads long enough to adequately amortize the overhead introduced by a dynamic execution model.

In terms of reducing network communication, the execution of threads must exhibit communication locality. This precludes scheduling policies such as round-robin or random placement, but favors solutions such as *work stealing* where all threads are created locally per default, but may later migrate to other nodes upon demand. Other solutions may be built around explicit policies specialized for the application program at hand.

The PCM model presented in this paper is aimed at solving the aforementioned problems. The intended target architectures are simple message passing machines which support the implementation of low-overhead communication layers such as *Active Messages* [24]. We do not assume any additional hardware support.

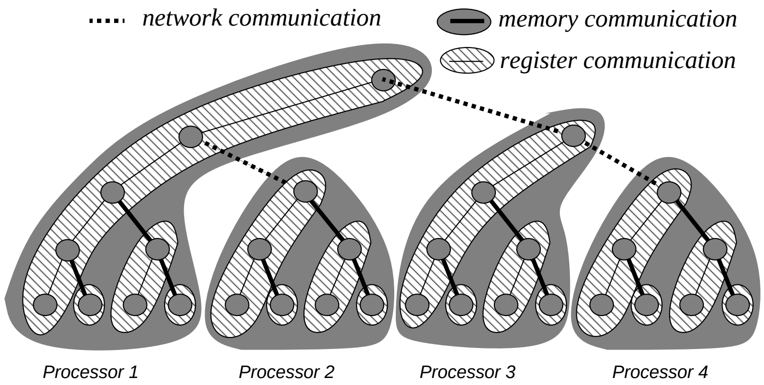

Figure 1: maximizing communication locality

We have provided C language extensions where threads can be specified along with conventional sequential C code. A program consists simply of a collection of *threads*, which are pieces of sequential code which are guaranteed to terminate once they have been scheduled. Threads represent the basic scheduling units that can be executed on any processor. By exposing threads in this way, we can experiment with various static and dynamic scheduling policies to optimize the overall machine utilization. In particular, we have found two strategies which can have a great effect on the performance of a program.

- A thread can be made to directly transfer control to another thread, similar to the control transfer mechanism used to implement tail-recursions. This mechanism bypasses dynamic scheduling entirely, thus avoiding all of its associated costs.

- The scheduler uses work stealing as its default policy. This creates patterns of locality in which threads can pass arguments through local memory rather than across the network most of the time, thus greatly reducing the communication frequency.

We show the effect of these optimizations in Figure 1. As this diagram suggests, there are three communication levels, namely register communication, memory communication and network communication. Preferably, we would like to transfer data through registers as much as possible, but the very nature of a dynamic execution model will force us to resort to memory communication or, even worse, to network communication. Note that an application increases the working set whenever it exposes additional parallelism, making it impossible to keep the entire working set in registers. The implementation goal will be to provide an optimal compromise between increasing the sequentiality of the application to increase its locality and exposing enough parallelism to enable dynamic load balancing. The second optimization, work stealing, will then attempt to minimize the work migration frequency, thereby minimizing network communication.

1.1 Related Work

The idea of continuation passing has been with us for some time now, since Steele and Sussman [21] forcefully argued its advantage in optimizing function calls. Several compilers for sequential languages convert programs into continuation passing style [3] before optimization and generating machine code. It is also used in parallel programming, for example, in [10]. We note that dataflow diagrams are essentially CPS in disguise [4], with additional structures to implement function calls.

Active Messages were popularized by von Eicken et al [24], although our current CM-5 implementation is based on a faster version in the Strata package [6]. There are numerous message passing libraries for conventional sequential languages, which implement the functionality of the underlying network hardware, but none of them matches Active Messages in terms of raw performance.

There also exist many parallel coordination languages, for example PVM [22] and Linda [7], which provide additional support for scheduling and synchronization. These packages typically arise from the needs of distributed computing, therefore are suitable mostly for coarser grained parallelism.

Parallel programming languages are more often modeled after communicating sequential processes, which gives rise to preemptive threads. For example, Mul-T [1] has blocking tasks which are context switched when an embeded future construct has not yet been computed to provide a value. An important feature of the Mul-T runtime system worth mentioning is that it uses lazy task creation, supported by work stealing, to increase the granularity of tasks [15].

The evolution of parallel computing based on the dataflow model has advocated the use of non-preemptive threads, also called micro-threads. Models based on this approach [19, 8] also superimposes upon threads a runtime environment consisting of a tree of frames (the parallel equivalent of the call stack in a sequential program) to support the function call abstraction. PCM is in fact the result of optimizing away this super-structure, replacing it with structures more resource-efficient and flexible in scheduling.

The rest of this paper is structured as follows: Section 2 introduces the PCM thread specification language and present in some detail all of its components. Section 3 introduces a cost model, intended to clarify the costs involved in dynamic execution. Section 4 contains in depth explanations of two actual problems, implemented with the PCM package. Section 5 states some possible future work and Section 6 concludes the paper.

2 The Parallel Continuation Machine

The parallel continuation machine implements an SPMD programming model, where all processors keep their own local copy of the entire code. The execution itself is completely asynchronous, meaning that each node may execute entirely different pieces of the program.

2.1 Elements of the PCM

A PCM program consists of a collection of *threads* that cooperatively solve a single problem. Statically, a thread identifies nothing more than a sequence of instructions, written in the machine language of the processor. At runtime, an application can create arbitrary numbers of dynamic instances of a static thread, each with its own set of arguments.[1]

The PCM thread specification language, which is explained in section 2.2, allows the programmer to define threads and to specify how threads communicate with each other. All machine specific execution details, such as dynamic load balancing or the mechanism require to enable transparent inter-thread communication, are part of the PCM runtime system and do not have to be specified by the user. This should simplify the task of writing explicit multithreaded applications without sacrificing the programmer's power for experimentation. The key elements of PCM's execution environment are illustrated in Figure 2.

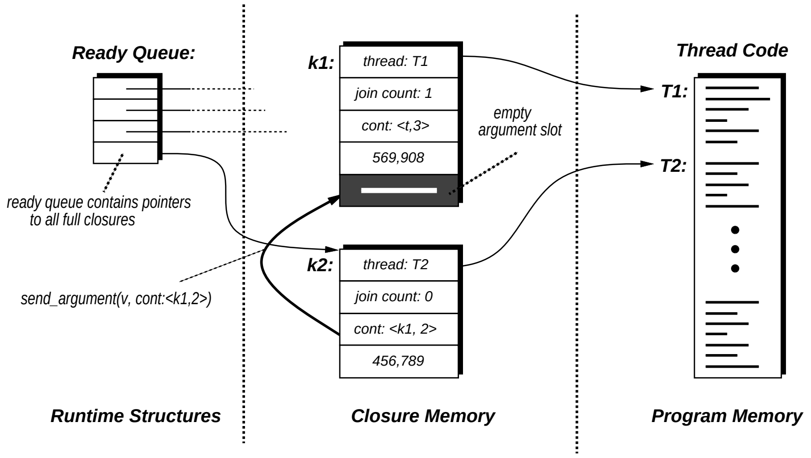

Figure 2: Elements of the PCM model

New dynamic instances of threads can be created by making closures. *Closures* form the contract between the application code and the runtime system. They contain a pointer to the thread code and all the arguments needed to execute the code. The thread is ready to execute when the closure becomes *full*; in other words, when it has obtained all the arguments. 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 of the arguments missing. These arguments are supplied by other threads in the future. A thread supplying an argument to a non-ready closure must obtain a reference to where the argument is to be sent. Such references will be called *continuations*; a continuation is just a pointer to a closure plus an integer offset into the closure. Note that we are slightly abusing the term continuation here, which is used to refer (in sequential computations) to *the rest of the computation* seen from a particular program control point. In parallel programs, however, there is usually no such thing as "the rest of the computation" from a single control point, and the issue of how a thread can be regarded as "continuing" from another thread, which sends one of the needed arguments, is further complicated by the synchronization involved.

In order to detect when a closure becomes full, every closure has an additional slot called the *join counter* that indicates the number of its outstanding arguments. The join counter is initialized with an integer equal to the number of missing arguments at closure creation

[1]We decided not to introduce a new term for these dynamic thread instances. No confusion should occur if the reader bears in mind that the word *thread* is used both in a static and in a dynamic sense.

time, and decremented each time the closure receives an argument. No closure is given to the scheduler until the join counter reaches zero.

2.2 The Thread Specification Language

The thread specification language described in this section is implemented as an extension to C. An example program computing the Fibonacci function is shown in Figure 3.

```
thread fib_fork (Cont parent, int n){
    if (n<2) send_argument (n, parent);
    else
    {    closure k1, s1, s2;

         k1 = make_closure (fib_sum, parent, _, _);
         s1 = make_closure (fib_fork, cont{k1,fib_sum:x}, n−1);
         s2 = make_closure (fib_fork, cont{k1,fib_sum:y}, n−2);
         post (s1);
         post (s2);
    }
}

thread fib_sum (Cont parent, int x, int y) {
    send_argument (x+y, parent);
}
```

Figure 3: PCM program to compute Fibonacci

In this language a program consists of threads, marked by the specifier **thread**, and normal C functions. A preprocessor expands just the threads into C functions, while copying the rest of the C code literally. The resulting C program can then be compiled and linked with the PCM runtime library.

Runtime primitives are used by the threads to create closures, send arguments, and transfer closures to the scheduler:

- `make_closure` (*Thread*, arg_1, ..., arg_n) Allocates a closure of size $n+2$, and returns a pointer to the closure. The two additional slots are reserved for the code pointer and the join counter. Closures can be created without specifying all the arguments, in which case a missing argument is indicated by "`_`". The join counter is always initialized to the the number of missing arguments.

- `post` (k) Hands the closure k over to the scheduler. Only a full closure can be posted inside the thread that created it. Closures with empty slots will be posted by `send_argument` when the join counter reaches zero [2].

[2]Note that for all examples shown in this paper except the protein folding problem of section 4, it is possible to let make_closure automatically post full closures, thus omit the explicit post operation.

- `send_argument` (v, c) Sends value v to continuation c and decrements the join counter of the target closure. The closure is posted if the join counter becomes zero. A continuation has the type **Cont** and contains two fields: a pointer to a closure, and an integer offset within the closure. A new continuation can be constructed, for example, using the expression **cont**`{k1, fib_sum:x}`, where `k1` is a closure pointer and `fib_sum:x` a symbolic reference to the first argument slot.

The preprocessor expands a thread, for example `fib_fork`, into a C function that takes a single argument, namely a closure, and fetches all specified arguments from the closure before starting its actual execution.

2.3 Executing a PCM Program

Figure 4 illustrates the sequence of events, when the Fibonacci program runs on a single processor. When there are multiple processors the only difference is that full closures may be migrated to ensure a balanced load across all available processors. These scheduling issues will be discussed in more detail in section 3.

The execution of a PCM program can be divided into three phases: initialization, computation, and termination. We did not show the initialization and termination code in Figure 3, since they are not really important. The first and third phases are usually very short, with the computation phase constituting the bulk of the overall execution.

During the *initialization* phase, shown in Figure 4.0, the program creates two closures. One specifies the start of the computation, in this example a ready instance of `fib_fork`, and one specifies the actions to be taken when the computation ends, in this example a non-ready instance of a special thread called `top`. This `top` thread is supplied by the runtime system and its responsibilities are to terminate the computation and to print the result(s). The `top` closure can be instructed to expect any number of results and must be the one to execute last.

During the *computation* phase, the scheduler enters a perpetual loop: popping a full closure from the ready queue; calling the thread function, such as `fib_fork` or `fib_sum`, as specified in the closure, with the closure pointer as its only argument. Figure 4.1–4.7 show snapshots of the machine state, once after each full closure has executed. For example, in figure 4.1, the thread `fib_fork` with argument 3 has just terminated. It created three closures: two full ones for `fib_fork` with arguments 2 and 1 respectively, and a closure for `fib_sum` waiting for two arguments. The full closures were immediately posted. These full closures contain continuations which point to the place in the sum closure where they will send their results. Similarly the sum closure contains a continuation pointing to the top closure, which is where its result should be sent.

During the *termination* phase, shown in Figure 4.8, the top thread is run. It will print the result of the computation and then cause the scheduler to exit the loop, thereby terminating the computation. For multiprocessor computations, the processor on which the `top` thread is executed also signals schedulers on other processors to exit their work loops.

2.4 Tail-Calls

A thread may directly call other threads via the following runtime primitive:

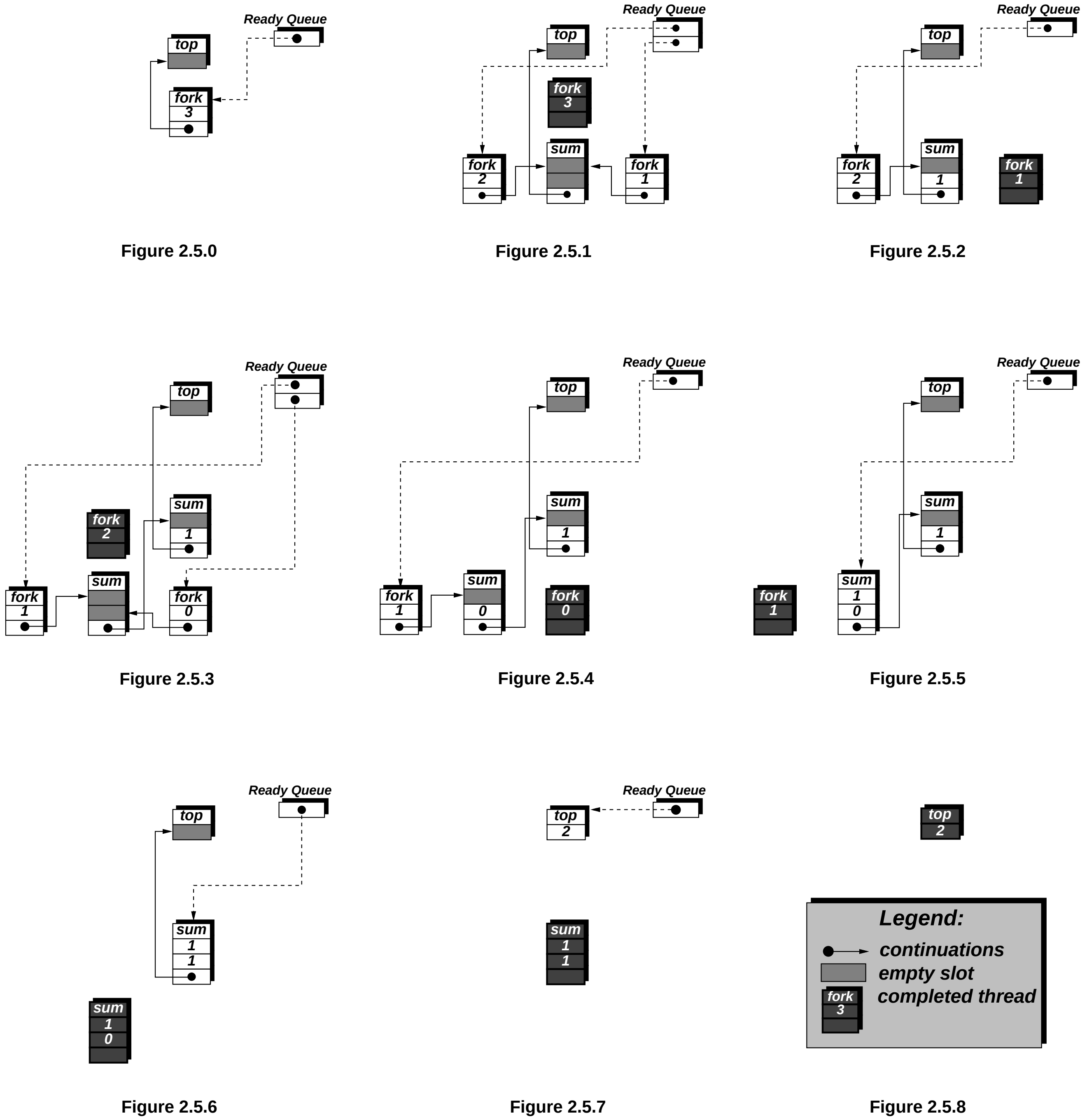

Figure 4: Snapshots of the state of PCM after each thread completion.

- `tail_call` (*Thread*, arg_1, …, arg_n)

A tail-call represents a more efficient invocation of a thread, avoiding any of the the dynamic execution overheads incurred otherwise. In the example shown in Figure 3, the thread `fib_fork` creates two full closures `s1` and `s2` and posts both of them before releasing control and returning to the scheduler. After receiving control, the immediate action of the scheduler will be to pop the `s2` closure and to call its thread function `fib_fork`, which in turn needs to unpack the `s2` closure prior to doing actual work. We can thus avoid this costly detour through the scheduler by rewriting the code with a tail-call as follows:

```
s1 = make_closure (fib_fork, cont{k1, fib_sum:x}, n−1);
post (s1);
tail_call (fib_fork, cont{k1, fib_sum:y}, n−2);
```

To implement this mechanism, the thread preprocessor actually expands a thread into two C functions: a *general entry version*, which is what we mentioned before; and a *fast entry version* which receives all arguments directly. A `tail_call` is thus converted into a standard C function call to the fast entry version. The actual performance improvements obtained with the tail-call mechanism can be quite impressive, especially for fine-grained applications, such as the Fibonacci example, where the performance improved by almost twenty-five percent. The effects of using the tail-call mechanism can be seen in Figure 1.

2.5 Passing Vectors in Closures

An additional mechanism provided by the thread language allows structures to be passed in closures. One of these structures may even be of arbitrary length. These vectors can then be referenced within a thread like any other local variable. The one vector argument which is allowed to be of arbitrary length needs to be specified as the last argument, to make sure it is packed into the tail of the closure.

`thread` *foo* (…, *type vect1*[10], …, *type vect2*[])

declares a vector argument *vect* of type *type*. It is the responsibility of the creator of the closure to initialize the vector. For example, the expression

`make_closure` (*foo*, …, *vect1*[10], …, *vect2* = [*size*])

creates a closure for *foo*, dynamically defining *vect2* to consist of *size* entries. In addition, *vect1* and *vect2* will be declared to be pointers initialized to the zeroth word of the corresponging vector arguments. These pointers must be used subsequently to initialize the vectors, which would otherwise be left empty. If we use "`_`" instead of a *vector name* when allocating a closure, then the vector is intentionally left empty, and its *size* will be added to the initial join count.

2.6 Partitioning Functions into Threads

Consider the following C program, which can be viewed as the "source" of the two threads shown in Figure 3.

```
int fib (int n) {
      if (n<2) return (n);
      else return (fib(n−1) + fib(n−2));
}
```

The function `fib` as shown has no parallelism. It order to expose parallelism, we must post one or both of the sub-calls to `fib`. But here we encounter a major difficulty: the addition depends on the return value of both calls. How can we set up the linkage between threads or instruct the scheduler to execute the addition after both sub-calls are complete?

One well-known solution is to use a *blocking thread model*. In such a model, the runtime system would suspend an executing `fib` thread after it forked off its two sub-calls and later, when the two sub-calls terminate, resumes the thread so that it can continue its execution. The costs for suspending and restoring a thread can be fairly expensive, depending on the number of registers that need to be saved[3].

The solution shown in Figure 3 uses a *non-blocking thread model*. In this model, new threads are created at points where context switches would otherwise occur. For example, thread `fib_fork` makes a closure for `fib_sum`, which is exactly saving the context needed for `fib_sum` to execute later. Consequently, threads always execute to completion once started. In other words, we are essentially following a caller-save convention, which tends to save fewer registers. For instance, in the `fib_sum` closure, only the continuation `parent` needs to be saved.

From a language point of view, threads provide an abstraction which is lower than the one of functions. As we have seen, a function such as `fib` translates into two static threads, which will then expand into three dynamic threads at runtime. As pointed out in the introduction, there are compilers with automatic partitioners that systematically convert and package a function into threads. Automatic partitioning is still a very active area of research (see for example, [23]), but we will not touch it further, since we are mainly interested in using the thread language in its own right in this paper.

3 Scheduling PCM Threads on a Multiprocessor

This section introduces a cost model for the PCM to motivate the work stealing scheduling system which we implemented to schedule PCM threads on the CM-5 parallel processor. All communication mechanisms required to implement the work stealer and the inter-thread communication have been built using a version of Active Messages [6].

As Figure 5 illustrates, we will equate useful computation with what a sequential program would have to do and classify everything else, such as communication, synchronization, and dynamic scheduling as additional overhead. The goal of this classification is to study the factors that determine the efficiency of a parallel computation with respect to its sequential counterpart.

To simplify the analysis, we will ignore the effect of idles and assume that each processing element is either executing a thread or one of the overhead tasks depicted in Figure 5. Under this assumption we can reduce the analysis to that of an average thread. The corresponding

[3]On the CM-5, for example, when suspending the current thread, it is necessary to save the entire context kept in the register windows. Generally, if the function call follows the callee-save convention, then all the callee-save registers must be saved.

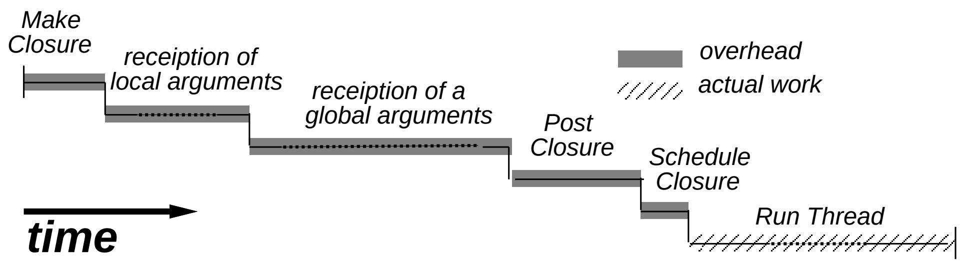

Figure 5: Anatomy of a PCM Thread

efficiency, ϵ, defined as the fraction of the overall execution time actually spent executing the useful computation, can then be calculated using equation (1).

There are three important ratios in the equation, reflecting the effects of tail-recursions (π_1), global send arguments (π_2) and closure migration (π_3) on the overall efficiency. π_1 equals the fraction of threads not using the tail-call mechanism, π_2 equals the fraction of arguments that have to be sent across the interconnection network and π_3 equals the fraction closures that migrate from one processing element to another, as a result of the work stealing scheduler. R_t defines the average run length of a PCM thread and k defines the threads arity.

Make Closure (M_c): At thread creation time, a closure must be allocated and initialized. Both closure allocation and initialization are constant and do not depend on the actual closure size ($M_c \approx 10$ cycles).

Send Local Argument (S_l): Local send arguments are fairly cheap and reduce to simple memory-to-memory transfer plus an additional check to see whether the closure becomes ready for execution (S_l) ≈ 10 cycles).

Send Global Argument ($S_l + \pi_2 \cdot T_a$): For arguments that must be sent across the network, we have to add an additional overhead factor T_a (≈ 100 cycles)[4] to the constant costs of S_l to account for the transfer costs.

Post closure ($P_c + \pi_3 \cdot T_c$): After receiving all of its arguments a closure is posted and becomes subject to dynamic scheduling ($P_c \approx 10$). If migrated to a remote processing element, additional transfer costs of T_c (≈ 500 cycles for a closure consisting of eight words) need to be charged in addition to constant cost P_t, reflecting the local posting.

Schedule closure (S_c): The costs for transferring control to the thread at the beginning of its execution and back to the scheduler after its termination are S_c (≈ 15 cycles).

$$\epsilon = \frac{R_t}{\pi_1 \cdot (M_c + k \cdot (S_l + \pi_2 \cdot T_a) + (P_c + \pi_3 \cdot T_c) + S_c) + R_t} \tag{1}$$

With the communication costs T_a and T_c ranging in the hundreds of cycles, it becomes imperative to reduce both π_2 and π_3 in order to avoid disappointing efficiencies. π_1, on the other hand, cannot be reduced to arbitrarily small values, thus the only remaining alternative to amortize the non-transfer related overhead is to increase the thread run length R_t.

[4]The transfer cost include both the sending and receiving overhead.

$$\epsilon = \frac{R_t}{\underbrace{M_c + P_c + S_c + k \cdot S_l}_{schedule\ dependent} + \underbrace{\pi_2 \cdot T_a \cdot k + \pi_3 \cdot T_c)}_{schedule\ independent} + R_t} \tag{2}$$

To illustrate the effect of π_2 and π_3 on the overall efficiency, we transformed equation(1) into equation(2), seperating schedule dependent and schedule independent costs and Figure 6 shows the resulting execution efficiency as a function of π_2 and π_3. For simplicity, we turned off the tail-call mechanism, which results into a value for π_1 equal to one, and set π_2 equal to π_3. In fact, for most computations we can expect π_2 and π_3 to be fairly dependent – higher migration frequencies will cleary force more of the arguments to be sent global.

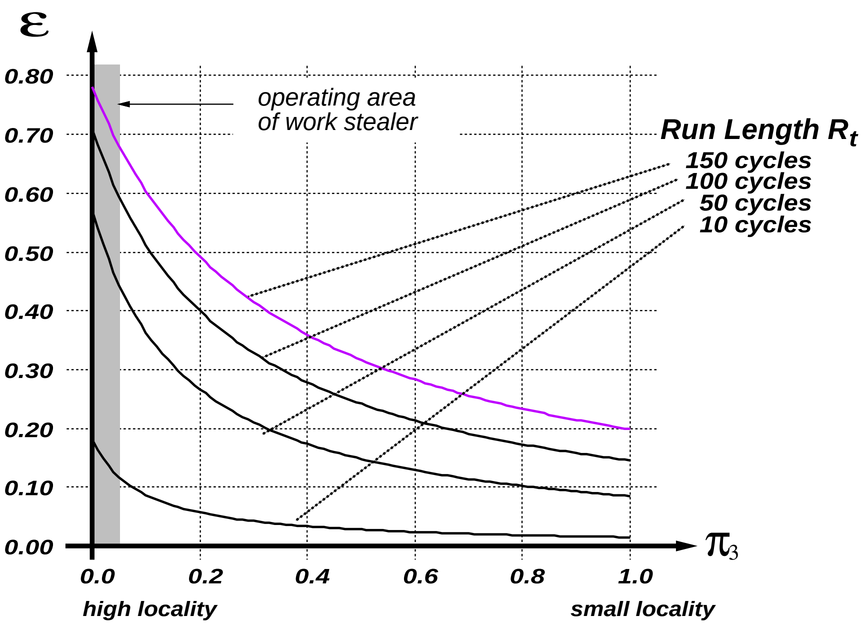

Figure 6: execution efficiency (ϵ) versus fraction of local global operations (π_2)

3.1 Implementation

To achieve minimal values for π_2 we have adopted a lazy scheduling policy known as *work stealing*. In such a system, closures are always scheduled locally per default, and will only be transferred across the network upon explicit requests initiated by starving processors.

Each processing element maintains a local queue of full closures, called the *ready queue*. When a closure becomes full, it is posted to its local ready queue. The local scheduler communicates with schedulers on other processing elements, and may later schedule the closure to execute locally, or may migrate it to another processing element. It is worth pointing out at this point that the actual schedule chosen is not going to affect the result of the computation. It will, however, have a significant impact on the efficiency of the computation.

In our current implementation, a computation executes locally using a depth-first scheduling policy. This heuristic can be expected to result in lower resource requirements for most computations when compared with a breadth-first policy. Steal requests, on the other hand, will always be served using a breadth-first policy (see Figure 1). Such a steal policy can be expected to result in significantly reduced steal frequencies for computations, such as recursions. For examples, both examples considered in the remainder of this paper typically migrated less than one percent of all dynamically created closures, as annotated in Figure 1. In addition, recent theoretical results described in [5] show that breadth-first stealing achieves linear speedup in the presence of modest parallel slackness.

4 Two Case Studies

In the following section we present two applications implemented with the PCM thread package.

4.1 Parallel Ray Tracing

The parallel ray tracer presented here is an optimal example to illustrate the virtues of the PCM thread package. First, the task of tracing a complicated picture with a reasonable size, like the one shown in Figure 9, contains enough parallelism to justify the use of a powerful parallel processor. Second, the variance in the amount of processor cycles required to trace individual rays necessitates the use of a dynamic load balancing schema to guarantee acceptable utilizations and to ensure scalability. Third, we can package the ray tracer into threads and obtain threads with grain-sizes coarse enough to offset the overhead introduced by a dynamic execution model.

The simplest of all ray tracing algorithms intersects a ray with every object surface and displays the object whose intersection is closest to the position of the observer. This algorithm is known as exhaustive ray tracing, since it calculates all possible ray-surface intersections. The ray-tracer we used improves upon this basic algorithm. It uses a bounding volume which requires relatively simple intersection calculations, such as a sphere, to enclose more complex objects. If a ray does not pierce the bounding volume then all the objects contained within can be eliminated from consideration which substantially reduces the average costs of ray surface calculations. This technique is further improved by arranging bounding volumes into hierarchies. In such a scheme a number of bounding volumes could themselves be enclosed within an even larger bounding volume so that many objects can be eliminated if a given ray does not intersect with a parent bounding volume.

This algorithmic improvement reduces the linear time complexity of exhaustive ray tracing to one which is logarithmic in the number of objects. However, this optimization also creates a large variation in the time needed to trace a ray, making it hard to find a static work distribution that sufficiently balances the available work.

4.1.1 Ray Tracer Parallelization

To show the power of our thread package as a tool to retarget existing sequential programs for parallel processors, we took the POV-Ray package that implements the optimized method described above and rewrote its kernel with our thread language. A simplified version of the original sequential kernel can be seen in Figure 7. The details of ray-object intersection caculation are not required to comprehend the transformations explained in the remainder

```
void Trace(){
    int x, y;

    for (y = First_Line; y < Last_Line; y++)
        for (x = First_Column ; x < Last_Column ; x++) {
            pixel = calculate_intersections(x, y);
            write_pixel(x, y, pixel);
        }
}
```

Figure 7: kernel of sequential ray tracer

of this section. In fact, the threaded version shown in Figure 8 uses the exact same function, `calculate_intersection` that the sequential version used.

The *Trace* thread traces a subwindow of the original picture, specified by the four coordinates in the argument list. *Trace* accomplishes this task by recursively splitting its subwindow into even smaller subwindows until the size of the newly created windows reaches the size of a single pixel. The implementation shown tries to split a trace job into four jobs by splitting the picture along both the x and y dimension.

Note that unlike the Fibonacci example, neither the sequential nor the parallel ray tracer return results. They simply write the RGB-values for the computed pixels into a preallocated buffer. The absence of a result forces us to resort to an alternative way to detect the termination of the program. The thread *Join* is used to implement a fairly general concept of a *signal tree*. The underlying idea is to use a tree to collect all the signals produced by the bottom cases of the recursion and to return a single signal to the top closure after all the signals have been collected.

4.1.2 Ray Tracer Results

To test our multithreaded implementation, we traced pictures of different complexities on various machine sizes. We adjusted the picture size so that enough parallelism would be generated to justify the use of the largest machine configuration used during our test runs.

We first compared the uniprocessor timings of the multithreaded code with those of the original sequential code, both running on the same CM-5 processing node. The results showed no measurable difference between the sequential and the multithreaded timings. To explain this, we need to have a look at the actual thread granularity. To trace the picture shown in Figure 9 with a resolution of 512×512 pixels, around 300,000 threads are created over the running time of about 1590 seconds, resulting in an average running time per thread of about 3 milliseconds ($\approx$ 100,000 Sparc cycles). This long thread run length makes the average overhead of 80 cycles/thread negligible.

In a second step, we compared the speedup behavior of our multithreaded ray tracer with that of an implementation using a static load balancing scheme (see Table 1). The static algorithm employs a simple work distribution that assigns exactly the same number of rays to each node. For the static case we listed two timings: the execution time of the fastest processor and the execution time of the slowest processor. We can see that already with two nodes, the slower processor requires 64% more compute cycles than the faster processor.

```
thread Join(Cont parent_join, int s1, int s2, int s3, int s4 ) {
      send_argument(SIGNAL, parent_join);
}

thread Trace(Cont parent_join, int sx, int ex, int sy, int ey) {
      if((sx == ex) && (sy == ey)) {
            pixel = calculate_intersections(sx, sy);
            write_pixel(sx,sy,pixel);
            send_argument(SIGNAL, parent_join);
      }
      else {
            closure k1, p1, p2, p3, p4;
            int xoff = (ex - sx) >> 1, yoff = (ey - sy) >> 1;

            k1 = make_closure(Join, parent_join, _ , _ , _ , _ );
            p1 = make_closure(Trace, cont{k1,Join:s1}, sx, (sx + xoff), sy, (sy + yoff));
            p2 = make_closure(Trace, cont{k1,Join:s2}, sx, (sx + xoff), (sy + yoff +1 ), ey);
            p3 = make_closure(Trace, cont{k1,Join:s3}, (sx + xoff + 1), ex, sy, (sy + yoff));
            p4 = make_closure(Trace, cont{k1,Join:s4}, (sx + xoff + 1), ex, (sy + yoff +1 ), ey);
            post(p1); post(p2); post(p3); post(p4);
      }
}
```

Figure 8: kernel of PCM ray tracer

Even worse, this gap widens as we increase the number of processors. With 64 nodes, the slowest processor is already taking 3.6 times longer than the fastest processor.

As pointed out at the beginning of this section, the time required to trace an individual ray can vary significantly, as shown by the execution timings obtained with the static load balancer in Table 1. To show this uneven work requirement, we calculated a work histogram for our example. The left part of Figure 9 shows the traced picture and the right part shows the work histogram. In the histogram brighter points represent higher workloads, darker points lighter workloads.

As a comparison, the multithreaded ray tracer not only performs better for all machine configurations than the static solution, it even achieves perfect linear speedup.

We added two additional columns to the data, showing the range of number of pixels traced per processor. Those numbers reflect the effect of the dynamic load balancer. As we expect, the difference between the maximum and minimum number of pixels traced per node increases as we move to larger machine configurations.

4.2 Protein Folding

A second application that we have implemented is protein folding. The reasons for choosing this application are similar to the reasons for choosing ray-tracing. The problems are large enough to warrant the use of parallelism. A common problem size takes over two hours when run sequentially, and we would like to run a series of problems. Also, an initial attempt to

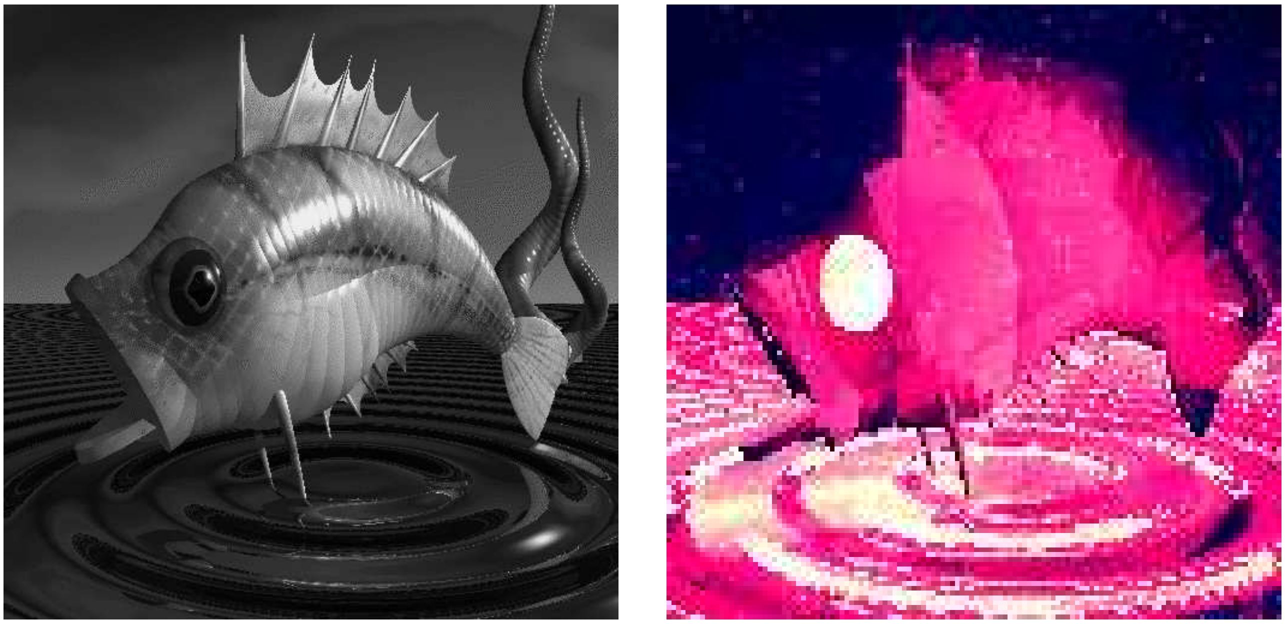

Figure 9: Traced picture and work histogram

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

Table 1: ray tracing result overview

parallelize the program did not make efficient use of the machine. This attempt statically broke the computation into subcomputations; but the subcomputations were too coarse, and their run times too variable, to keep all processors busy. An implementation using PCM avoids this problem.

The work on this problem was done in conjunction with Vijay Pande of the Center for Material Sciences and Engineering at MIT. In their work [18] they use the lattice model [20] for protein design. In this approach to the problem a protein is described as a chain of monomers (each monomer may represent a number of amino acids). It is assumed that each monomer will sit on a lattice point (*i.e.,* a point on a 3D grid). Each possible folding of the polymer can then be described as some path along the set of lattice points. Figure 10 shows a polymer of length 26; each shade represents a different type of monomer. The model assumes the polymer will take on the most compact possible paths, so it is only concerned with paths that completely fill some cube. In a folded polymer, two neighboring monomers will exert some attractive or repulsive force on one another. This force depends on the types of the two monomers. The energy of a folded polymer is the sum of the forces between all of the neighboring monomers. Of course, this energy value depends greatly on the way in which the polymer is folded. For this work it is necessary to consider all possible foldings of a given polymer and compute a histogram of the energy values. This is what we have implemented using the PCM model on the CM-5.

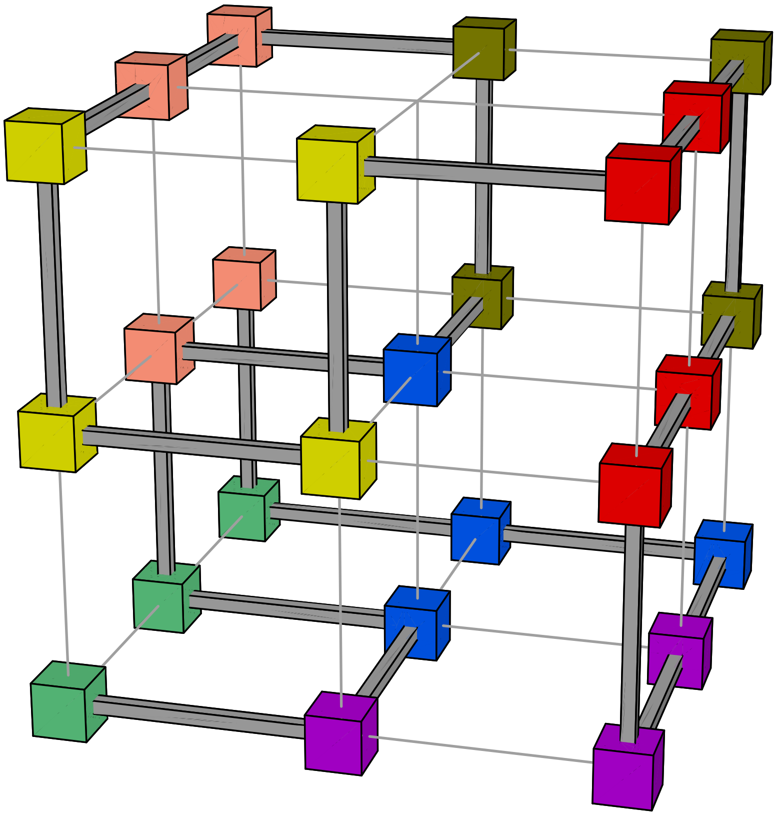

Figure 10: A folded polymer

For the rest of this section we will be concerned mainly with the implementation of this problem using PCM. For more details on the algorithms used and the results obtained see [17]. At its heart, this program is a search program that finds all possible unique paths in a cube; each path must traverse each point in the cube exactly once. Each complete path represents one possible folding of the polymer; so each time a path is found the energy value for that folding is calculated and a result histogram is updated with the value.

This algorithm works by incrementally building up paths through the cube until complete paths are reached. The function *Count_Entries* performs the core of the search. An outline of the sequential code for this function is given in Figure 11.

The first argument to this function is a STATE structure. This structure defines the partial path that has been constructed so far. This structure contains information describing which points are occupied, the type of monomer at each occupied point, and other data used to increase the efficiency of the search. The size of the STATE structure is on the order of 100 bytes. The second argument to the function is *point*, the lattice point to be added to the partial path. This function returns an integer, namely the number of paths found. The function first adds *point* to the partial path. If this completes the path it updates the result

```
int Count_Entries(STATE st, int point) {
    memcpy(state,st,sizeof(STATE));
    add_point_to_path(point,state);        /** add point to path **/
    if (complete_path(state)){
        update_result_histogram(state);
        return 1 ;
    }
    /** If no complete paths are possible return 0 ***/
    give_up = f(state,point);
    if (give_up) return 0;
    /** Otherwise call Count_Entries recursively on each neighbor **/
    else{
        sum = 0;
        for(i=0;i<num_neighbors;i++){
            next_neighbor=neighbor[i];
            if (not_occupied(next_neighbor,state)){
                sum += count_entries(state, next_neighbor);
            }
        }
    }
    return sum;
}
```

Figure 11: Kernel of sequential protein folding

histogram with the energy value of this new path and returns. Otherwise it applies some tests to see if it impossible for this state to lead to a complete path. If a complete path is impossible it prunes the search here. Otherwise it calls itself recursively for each empty neighbor of *point*. At the end it returns the total number of complete paths found. Typically we run the program on several polymers at a time. Each time we find a complete path we just calculate several energy values. This amortizes the time spent searching over several polymers.

We start the search by calling *Count_Entries* on a set of starting paths. The obvious way to start this search is to call *Count_Entries* once for each point in the cube, each time using a starting path consisting only of that point. The problem with this approach is that it would count duplicate paths. Two paths are considered duplicates if one path is a rotation and/or reflection of the other. Instead, we have a second (sequential) program which generates a set of starting paths that guarantees that no generated paths will be duplicated. The initial routine simply calls *Count_Entries* on each of these starting paths.

4.2.1 Protein Folding Parallelization

An earlier attempt at parallelization was made without using PCM. In this attempt the starting paths were statically divided between the nodes. Each node then ran the sequential code for its subset of the starting paths. At the end the result histograms on all the nodes are merged. This was trivial to implement but gave poor results because of inadequate load balancing. The amount of work involved for calculating with different starting paths can differ by many orders of magnitude.

To get a more efficient parallelization, the computation needed to be broken into finer grains. PCM was ideal for this task. The procedure that makes use of the PCM primitives is the *Count_Entries* procedure. A skeleton of the code for this procedure is given in Figure 12. The major changes to this code to run it using PCM are similar to the changes made to the ray-tracer. In particular, most of the recursive calls to *Count_Entries* are now made by creating and posting a closure. Also, this version takes an additional argument *parent*. This is the continuation to which the result will be sent.

There have been other changes to the code, however, most of these changes were made for performance reasons rather than correctness reasons. The first difference is that this version determines in advance the number of neighbors that will be visited. If there is just one neighbor that needs to be visited, then exactly one recursive call needs to be made. We do this by making use of a tail-call. In this instance the tail call eliminates two overheads: first, the posting and scheduling of the closure, and second, the copying of the state argument into the new closure.

If there are more than one recursive call to be made then a summation closure is created. This closure will add up the results of the subcomputations and send the result to the parent of this thread. The unusual syntax in the call to make_closure (*i.e.,* " $\_ = [num\_ntv]$") signifies that a specified number of empty slots (here *num_ntv*) should be left in the closure. These slots will be filled in later with the results of the subcomputations. Most of the recursive calls are made by making and posting closures. The final call makes use of a tail call, for the same reasons given above.

4.2.2 Protein Folding Results

Many variations of this program have been run on a range of problem and machine sizes. Results for two problem sizes are shown in Figure 13. The first figure shows a smaller problem size, namely a $3 \times 3 \times 3$ cube which has 103,346 paths. The second shows a larger size, 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 using PCM on various machine sizes.

The first observation is that the overhead added by the PCM model is fairly small. The PCM program running on one processor was 14 and 22 percent slower than the sequential version in these examples[5]. This overhead is due both to posting and scheduling closures and to the copying of the state structure into each closure.

The second observation is that the speedup curves are almost perfectly linear for both of these problems. (Even on the smallest problems we ran we saw nearly linear speedups. On 64 processors we achieved a speedup of 60 and a run time of only 0.25 seconds.) This is because the granularity of the threads is small enough that the workload can be evenly distributed.

[5] Due to its large runtime, for the second example the sequential version was run on a standard Sparc, not on one processor of the CM-5. The runtime on a single CM-5 node is likely to be slightly larger than this. This causes us to slightly overstate the overhead.

```
THREAD Count_Entries(Cont parent; int point, STATE state,) {
    add_point_to_path(point,state);
    if (complete_path(state)){
        update_result_histogram(state);
        send_argument(1, parent);
        return;
    }

    /** Determine number of neighbor nodes to be visited **/
    give_up = f(state,point);
    num_ntv = g(...);        /** num_ntv = num of neighbors to visit **/
                             /** nbrs_to_visit[i] = 'i'th neighbor to visit **/

    /** Case 0: If no paths to search, then return 0 (no paths found) **/
    if (give_up || (num_ntv==0)) send_argument(0, parent);

    /** Case 1: If exactly 1 neighbor to visit - only try that one  **/
    else if (num_ntv==1)
        tail_call(Count_Entries,parent,nbrs_to_visit[0],state);
    else{
        /** General case - n neighbors to try [n>1] **/
        /** create a closure to sum results of all sub-computations **/
        /** post num_ntv-1 threads and perform a tail call for the last **/
        sum_closure = make_closure(sum,parent,num_ntv,_=[num_ntv]);
        for(i=0;i<(num_ntv-1);i++){
            next_neighbor=nbrs_to_visit[i];
            k1 = make_closure(Count_Entries,
                              cont{sum_closure,sum:val[i]},
                              next_neighbor, new_st=[sizeof (STATE)]);
            memcpy(new_st,st,sizeof(STATE));
            post(k1);
            }
        /**Perform a tail call for final neighbor **/
        new_parent = Cont{sum_closure,sum:val[num_ntv-1]};
        tail_call(Count_Entries,new_parent,next_neighbor+1,state);
    }
}
```

Figure 12: Kernel of parallel protein folding

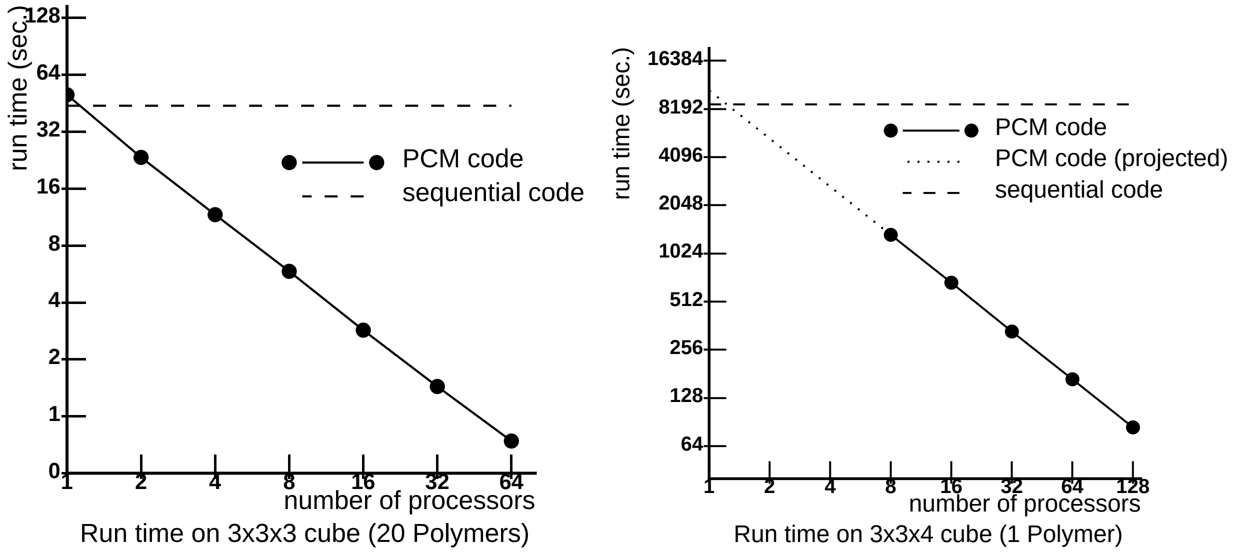

Figure 13: Run times for protein folding

5 Future Work

An area that needs to be explored further, which will unquestionably constitute the bulk of future research, is that of global data structures. Since global memory accesses typically involve communication, it is necessary to split threads at the boundary of remote memory accesses. As a consequence, it will be hard to scale applications with frequent global memory references on architectures such as the CM-5.

Two key issues need to be addressed to enable efficient implementations. First, data structures need to be evenly distributed across all processing elements and second, they need to be aligned with the computation. Data structures and computational threads can be aligned either by moving the data structure, as done by architectures based on globally coherent caches [11][13], or by moving the computation. Our preliminary results show that we need to use both techniques to achieve efficient solutions. However, no general solution has yet surfaced.

Although the distribution and alignment of data structures will undoubtedly interfere with the scheduling of computational threads, we excluded these issues in order to obtain a clean and simple execution model, thereby clarifying the issues of dynamic execution. We believe that this simplicity will make it easier to identify and to integrate additional scheduling mechanisms targeted to address the alignment problem.

6 Conclusions

The performance of any parallel program must scale over the performance of the best sequential program to be truly practical. Because of the high costs of dynamic scheduling and network communication in current message-passing architectures, this goal becomes a serious challenge when programming applications with unstructured parallelism.

Barring revolutionary hardware solutions, we have argued that there are effective solutions in software to (a) lengthen sequential threads in order to offset the costs of dynamic scheduling, and to (b) preserve the locality of computations to reduce the network traffic. We believe that these solutions have not yet been effectively explored by message passing extensions to conventional languages, nor by higher level parallel languages, because their level of abstraction is either too low or too high to enable the programmer to effectively tune an application toward these key optimizations.

The parallel continuation passing model presented in this paper represents an attempt to bridge this gap. The model can either serve as a compilation target for a higher level language, or it can be used directly in conjunction with a sequential language, such as C. In the latter case it comes as a simple extension, providing the essential structures needed to synchronize computational threads and to optimize scheduling decisions. Although it could be argued that PCM is difficult to program because of its explicit continuation-passing style, we found it often the case that a program just has a small kernel that needs to be parallelized, leaving the rest of the program in its original sequential form (see the ray-tracing example). Moreover, such a parallelization always increases the granularity of the sequential threads, resulting in more efficient parallel programs.

As the outcome of experimenting with PCM, we identified two scheduling policies of general use, increasing the efficiency of parallel applications based on a dynamic execution model. First, the tail-call mechanism gives the programmer the flexibility to glue short threads into longer ones. Tail-calls will undoubtedly become extremely important for finer grained parallel computation, as has been shown for the Fibonacci example. Second, a work stealing scheduling policy enables almost-all-local computation, resulting in linear and near-linear speedups of the ray-tracing and protein-folding examples. Since work stealing is often the determining factor in whether or not a parallel application executes efficiently, we built this mechanism into the PCM runtime system as the default scheduling policy.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *The 17th Annual International Symposium on Computer Architecture Conference Proceedings*, Seattle, Washington, May 1990.

[2] 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.

[3] Andrew Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[4] 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.

[5] Robert Blumofe and Charles E. Leiserson. Work-Stealing Algorithms for Scheduling Multithreaded Computations. Manuscript in progress.

[6] Eric Brewer and Robert Blumofe. *Strata Reference Manual.* Lab for Computer Science, MIT, version 2.0 edition, January 1994.

[7] N. C. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989.

[8] 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 4^{th} Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa-Clara, CA, April 1991.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W.H. Freeman and Company, 1979.

[10] J. F. Giorgi and D. Le Metayer. Continuation-Based Parallel Implementation of Functional Languages. In *Proc. POPL*, pages 209–217, 1990.

[11] Kendall Square Research. *KSR1 Technical Summary*, 1992.

[12] J. Kuehn and B. Smith. The Horizen Supercomputer System: Architecture and Software. In *Proc. of Supercomputing '88*, Orlando, Florida, November 1988.

[13] 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.

[14] 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.

[15] E. Mohr, D. Kranz, and R. Halstead. Lazy Task Creation. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[16] 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.

[17] V. Pande and C. Joerg.

[18] Vijay Pande, Alexander Yu, Grosberg, and Toyoichi Tanaka. Thermodynamic Procedur to Construct HeteroPolymers that can be Renatured to Recognize a Given Target Molecule. *submitted to Nature.*

[19] 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.

[20] E. Shaklmovich and A. Gutin. *J. Chem. Phys.*, 93:5967, 1990.

[21] G. L. Steele and G. J. Sussman. LAMBDA, the Ultimate Imperative. AI Memo 353, MIT AI Lab., March 1976.

[22] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2:315–339, April 1990.

[23] K. R. Traub, D. E. Culler, and K. E. Schauser. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads. In *Proceedings of the 1992 ACM Conf. on Lisp and Functional Programming*, pages 324–334, June 1992.

[24] 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.