OLDEN: PARALLELIZING PROGRAMS WITH DYNAMIC DATA STRUCTURES ON DISTRIBUTED-MEMORY MACHINES

Martin Christopher Carlisle

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

June 1996

Abstract

The goal of the *Olden* project is to build a system that provides parallelism for general-purpose C programs with minimal programmer annotations. We focus on programs using dynamic structures such as trees, lists, and DAGs. We describe a programming and execution model for supporting programs that use pointer-based dynamic data structures. The major differences between our model and the standard sequential model are that the programmer explicitly chooses a particular strategy to map the dynamic data structures over a distributed heap, and annotates work that can be done in parallel using futures. Remote data access is handled automatically using a combination of software caching and computation migration. We provide a compile-time heuristic that selects between them for each pointer dereference based on programmer hints regarding the data layout. Also, we provide a new local-knowledge coherence mechanism for the cache that outperforms traditional invalidation methods on our benchmarks. The Olden profiler allows the programmer to verify the data layout hints and to determine which operations in the program are expensive. We have implemented a prototype of Olden on the Thinking Machines CM-5. We describe our implementation and report on experiments with eleven benchmarks.

Acknowledgments

As with probably any dissertation, this document would never have come to be without the assistance and encouragement of an exceedingly large number of people. I shall endeavor to recognize at least some of them here.

I wish to thank my advisor, Anne Rogers, for her support throughout the process of creating this dissertation. Her striving for excellence and clarity of presentation have contributed greatly to my ability to write effectively. Also, her unselfish counsel throughout my graduate career has been especially appreciated.

Also, my readers, Andrew Appel and Ed Felten, are to be thanked for their diligence in proofreading this document. Their efforts enabled me to have a much better document, particularly in the discussion of cache coherence in Chapter 3. Andrea LaPaugh and Bob Sedgewick also contributed to the presentation of the work by their participation on my committee. Rishiyur Nikhil supervised a summer internship at Digital CRL, and provided advice on finding a job. Also of particular note are Dave Hanson and Chris Fraser, who allowed me to use early releases of lcc, and answered many questions.

The rest of the department has also made Princeton an enjoyable place to work. The technical staff has done a superb job of providing computing resources, answering questions, and saving me from my own blunders.

Errol Lloyd and M. Sandra Carberry from the University of Delaware played a key role in helping me decide to undertake this effort, and provided encouragement at key points along the way.

J. Barnes, J. Hummel, L. Guibas, S. Lumetta, J. Müller, and J. Stolfi graciously provided ideas and implementations of benchmarks that we ported to Olden. Ender Bilir and G. Scott Mackenzie made significant contributions to the system in independent work projects.

Special thanks go to my family and friends, who have been of great help to me as I have sought to be able to express my work to laymen, never letting on if they were bored. My parents have always showed an interest, and have been supportive in every conceivable way. Bill Kerwin, on the night before his wedding, helped me come up with the name for the express-checkout register. Sarah Crandall, Neil Evans, Bill Heller, Theresa Lupo, and Bryan Pyle have stayed in touch throughout my graduate studies. At Princeton, I have been privileged to have an enormous number of friends through my affiliation with the Princeton Evangelical Fellowship, and to be mentored by Bill Boyce, Clay Porr, Jamie Rankin and Wayne Wever. My roommates, Jon Johnson, John Quintanilla, and Chuck Vellenga, have provided both encouragement and practical assistance in both meals and a sane place to live. Jeff Cagle, Phil Coburn, Bill Fullilove, and Michael Marucheck have been constant friends beyond their years at Princeton.

Most importantly, I thank "the God and Father of our Lord Jesus Christ, who according to His great mercy caused [me] to be born again to a living hope through the resurrection of Jesus Christ from the dead." (1 Peter 1:3) My relationship to God through Jesus has been a continual source of strength and joy thoughout my time here.

On a less personal note, I gratefully acknowledge funding from the Fannie and John Hertz Foundation, a National Science Foundation Graduate Fellowship, NSF Research Initiation Grant ASC 91-10766, NSF FAW award MIP-9023542, and the Department of Computer Science at Princeton University. Also, the National Supercomputing Centers at the University of Illinois (NCSA) and Syracuse University (NPAC) provided access to the CM-5s.

Beginning July 1, I will be an assistant professor of computer science at the United States Air Force Academy.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

"Many hands make light work." Heywood's proverb [Hey92] succinctly states the notion that many people working together can accomplish a task more quickly and easily than a single person working alone. Similarly, parallel processing is based on the idea that a collection of connected computer processors can solve a problem much more quickly than a single computer. Unfortunately, the complexities of parallelism can often make the task more reminiscent of the phrase, "The best way to slow progress is to form a committee." [McK80]

To improve performance by parallelization, the programmer must find tasks that can be done in parallel, manage the creation of threads to perform these tasks and their assignment to processors, synchronize the threads, and communicate data between them. Although many different hardware platforms have been proposed using a single shared memory to communicate data between tasks, parallel machines are converging toward workstation-like nodes connected by a network [RLW94b]. These machines have the advantages that they are easy to build and keep with the technology curve (as they require little special purpose hardware), and that they are also easily scaled to large numbers of processors. Because the memory is distributed among the processors, *locality*, that is, making sure the data is at the same place as the computation, is a significant performance issue.

Handling all of these issues is a complex and time consuming task. Complicating matters further is the fact that program bugs may be timing-dependent, changing in nature or disappearing in the presence of monitoring. Consequently, there is a need for good abstractions to assist the programmer in performing parallelization. These abstractions must not only be expressive, but also efficient, lest the gain from parallelism be outweighed by the additional overhead introduced by the abstractions.

Olden is a compiler and run-time system that provides parallelism on distributed-memory machines for general purpose C programs with minimal programmer annotations. Specifically, Olden is intended for programs that use dynamic data structures, such as trees, lists and DAGs. Although much work has been done on compiling for distributed-memory machines, much of this work has concentrated on scientific programs that use arrays as their primary data structure and loops as their primary control structure.[1]

In Section 1.1, we review parallel code generation for scientific programs using arrays as their primary data structure. We also describe an attempt to apply these techniques directly to programs using pointer-based dynamic data structures, and explain why a better solution is needed. Then, in Section 1.2, we describe our solution, *Olden*, and its contributions. Finally, we give an outline of the rest of the thesis in Section 1.3.

1.1 Previous Work

Before we examine how one might generate parallel code for programs using dynamic data structures, let us first review why it is possible for scientific programs using arrays and loops, and then point out the fundamental problems that prevent us from applying the same techniques to programs with pointer-based dynamic data structures.

From a compilation standpoint, the most important property of a distributed-memory machine is that each processor has its own address space; remote references

[1] For example, see [AL93a], [AL93b], [CK88], [Ger90], [HKT91], [Koe90], [RP89], and [ZBG88].

are satisfied through explicitly passed messages, which are expensive. Therefore, arranging a computation so that most references are local is crucial to efficient execution. Scientific programs using arrays and loops tend to have the property that the arrays can be partitioned into relatively independent pieces, and therefore the operations performed on these pieces can proceed in parallel. These properties make them ideal applications for distributed-memory machines. Each group of related data can be placed on a separate processor, which allows operations on independent groups to be done in parallel with little interprocessor communication.

The key insight underlying recently developed methods for automatically parallelizing programs for distributed-memory machines is that the layout of a program's data should determine how the work in the program is assigned to processors. Typically, the programmer specifies a *mapping* of the program's data onto the target machine, and the compiler uses this mapping to decompose the program into processes. The simplest compilation strategy, sometimes called *run-time resolution*, inserts code to determine at run time which processor needs to execute a particular statement. Different policies for allocating work are possible, but the most popular is the *owner computes* rule: the work of an assignment statement (`v = e`), including the computation of `e`, is assigned to the processor that "owns" `v`. Control statements such as conditionals and loops are executed by all processors. The code produced by this method can be improved substantially using the arsenal of techniques developed for vectorizing compilers, such as data dependence analysis and loop restructuring [AK87, Wol89].

Run-time resolution works because arrays are static in nature, that is, names are available for all elements of an array at compile time. To determine the processor responsible for a given array element, the programmer-supplied mapping function is applied to the array element's global name. Since every processor knows the global name of every array element, this test can be done locally without communication. Techniques for improving run-time resolution code rely on the fact that the expressions used to reference array elements tend to be very simple and have nice mathematical properties.

Now let us return to our problem of parallelizing programs that use pointer-based dynamic data structures. Such programs often exhibit the required property that their data structures can be partitioned into relatively independent pieces. For example, a tree can be recursively partitioned into smaller, independent subtrees, and a list can be recursively partitioned into its head and its tail. Furthermore, this partitioning can often be used to distribute parallel tasks over the sub-pieces. Thus, so far, we see no fundamental problem in mapping these programs to distributed-memory machines. But, with further investigation, it becomes clear that the techniques used for array-based programs do not work for programs using dynamic data structures. The first problem is that determining the independence of operations on a dynamic data structure is substantially harder than determining the independence of array operations. This is partially because the nodes of a dynamic data structure do not have compile-time names, and therefore references to a structure do not share the nice mathematical properties of array references. Second, recursion, rather than looping with its easily partitionable index sets, is the primary control structure used with dynamic data structures. Finally, without compile-time names, the mapping of nodes to processors cannot be done statically, and the owner of a node cannot be determined, in general, without interprocessor communication.

Gupta [Gup92] suggests a mechanism for addressing the problem of global names so that an approach similar to run-time resolution can be used. In his approach, a global name is assigned to every element of a dynamic data structure, and this name is made known to all processors. To accomplish this, a name is assigned to each node as it is added to a data structure. This name is determined by the node's position in the structure and is registered with all processors as part of adding it to the structure. The mapping of a node to a processor is also based on its position in the structure. As an example, a breadth-first numbering of the nodes might be used as a naming scheme for a binary tree. Once a processor has a name for the nodes in a data structure, it can traverse the structure without further communication.

It is important to note that this new way of naming dynamic data structures leads to restrictions on how the data structures can be used. Since the name of a node is determined by its position, only one node can be added to a structure at a time (for example, two lists cannot be concatenated). Another ramification of Gupta's naming scheme is that node names may have to be reassigned when a new node is introduced. For example, consider a list in which a node's name is simply its position in the list. If a node is added to the front of the list, the rest of the list's nodes will have to be renamed to reflect their change in position.

The goal of the Olden project is to find a better solution to the problem of parallelizing pointer-based programs on distributed-memory machines. The collection of work just described provided motivation and insights that aided us in accomplishing this task. Other researchers have taken different approaches in creating systems that parallelize programs using dynamic data structures on distributed-memory machines. We defer discussion of these systems until Chapter 8, after we have described our system fully.

1.2 Olden's Contributions

In Olden, we use a dynamic mechanism to match the dynamic nature of the data structures. As in the previously described approaches, we rely on the programmer to specify a mapping of the data onto the processors. This mapping is done at run time by specifying a processor name with every memory allocation request. The key difference between our approach and previous approaches is the mechanism for assigning work to processors. We believe that run-time resolution, which was developed to support statically defined, directly addressable rectangular arrays, is inappropriate for pointer-based dynamic data structures, which are neither statically defined nor directly addressable.

We propose a more dynamic approach that is better matched to the dynamic nature of the data structures themselves. Instead of having a single thread running on each processor, we use *futures* to create parallel threads dynamically as processors become idle. The programmer marks a procedure call with a future if the procedure can be executed safely in parallel with its parent continuation. Also, to handle remote references, we use a combination of computation migration and caching rather than a static owner-computes rule. Computation migration is similar to using a static owner-computes rule, as the processor owning the data is responsible for reading and writing its value; however, computation migration sends the computation to the data. Software caching performs the dual function, bringing the data to the computation that needs it. Computation migration takes advantage of the spatial locality of nearby objects in the data structure; while caching takes advantage of temporal locality and also allows multiple objects on different processors to be accessed more efficiently.

Since distributed-memory machines do not support coherent caching directly, we implemented a software-caching scheme. Our caching method relies on the insertion of pointer-test code by the compiler and uses active messages [vECGS92] for communication between processors. We have implemented three different coherence mechanisms—*global knowledge, local knowledge, and bilateral*—and evaluated their performance. Our local-knowledge relaxed coherence protocol outperforms more traditional invalidation-based protocols for Olden programs.

Also, as selecting whether to use computation migration or software caching for each program point is very tedious, we provide a compile-time heuristic to make these choices automatically. The mechanism choice is dependent on the layout of the data; therefore, we provide an extension to the language, *local path lengths*, that allow the programmer to give a hint about the expected layout of the data. Given the local-path-length information (or using default information if none is specified), the compiler will analyze the way the program traverses the data structures, and, at each program point, select the appropriate mechanism for handling remote references.

What constitutes a good data layout and the appropriate local-path-length values for each data structure are not always obvious. To assist the programmer, we provide a profiler, which will compute the appropriate local-path-length values from the actual data layout at run time, and also report the number of communication events caused by each line of the program. Using this feedback, the programmer can focus on the key areas where optimization will improve performance and make changes in a directed rather than haphazard manner.

In summary, the Olden compiler and run-time system provide abstractions that greatly simplify the task of parallelizing programs using pointer-based dynamic data structures, and provide a programming model that requires minimal annotations beyond a sequential implementation. In combination, our computation migration and software caching mechanisms provide good performance, and our compiler can automatically choose the appropriate combination of these mechanisms with only a small amount of additional information from the programmer regarding the data layout. The programmer may obtain this information by analyzing the data layout directly, or by obtaining a run-time profile using the Olden profiler. The profiler also gives feedback regarding the data layout by identifying which portions of the program cause communication. We have demonstrated the effectiveness of our system on a suite of eleven benchmarks.

1.3 Organization

In Chapter 2, we describe Olden's programming model. We describe the execution model in Chapter 3 and its implementation in Chapter 4. This includes our mechanisms for migrating computation based on the layout of heap-allocated data, for sending remote data to the computation that requires it using software caching, and also for introducing parallelism using futures and lazy task creation [MKH91]. Then, in Chapter 5, we introduce the heuristic used by the compiler to choose between computation migration and software caching. In Chapter 6, we report results for a suite of eleven benchmarks using our implementation on the Thinking Machines CM-5, and in Chapter 7, describe the Olden profiler and how to use it to improve performance. Finally, in Chapter 8 we contrast Olden with other projects, and in Chapter 9, give conclusions and suggestions for future work. This dissertation expands on our previous work [CR95, CRRH93, RCRH95, RRH92].

Chapter 2

Programming Model

Olden's programming model is designed to facilitate the parallelization of C programs that use dynamic data structures on distributed-memory machines. The programmer converts a sequential C program into an Olden program by providing data-layout information, and marking work that can be done in parallel. The Olden compiler then generates an SPMD (single-program, multiple-data) program [Kar87]. The necessary communication and thread management is handled by the Olden run-time system. In this chapter, we describe the programmer's view of Olden.

In our SPMD model, each processor has an identical copy of the program, as well as a local stack that is used to store procedure arguments, local variables, and return addresses. Additionally, each processor owns one section of a distributed heap. We view heap addresses as consisting of a pair, $\langle p, l \rangle$, of a processor name and a local address. This information is encoded in a single 32-bit word. All pointers are assumed to be global pointers into the distributed heap.

2.1 Programming Language

Olden takes as input a program written in a restricted subset of C, with some additional Olden-specific annotations. For simplicity, we assume that there are no global variables (these could be put in the distributed heap). We also require that programs

```
typedef struct tree {
  int val;
  struct tree *left, *right;
} tree;

int TreeAdd (tree *t)
{
    if (t == NULL)
        return 0;
    else {
        return (TreeAdd(t->left) + TreeAdd(t->right) + t->val);
    }
}
```

Figure 1: `TreeAdd` function.

do not take the address of stack-allocated objects, which ensures that all pointers point into the heap.[1] The major differences between our programming model and the standard sequential model are that the programmer explicitly chooses a particular strategy to map the dynamic data structures over the distributed heap and annotates work that can be done in parallel. We provide three extensions to the language: `ALLOC`, *local path lengths*, and *futures*. `ALLOC` and local path lengths are used to map the dynamic data structures, and provide information to the compiler regarding the mapping. Futures are used to mark available parallelism, and are a variant of a construct used in parallel Lisps [Hal85].

Throughout this chapter, we will examine how a very simple function, `TreeAdd`, would be modified for Olden. `TreeAdd` recursively sums the values stored in each of the nodes of the tree. The program is given in Figure 1. As previously mentioned, the programmer is responsible both for describing the layout of the data, and for marking available parallelism. We examine data layout in Section 2.2, and parallelism in Section 2.3.

[1] We do provide structure return values, which can be used to handle many of the cases where & (address-of) is needed.

```
/* Allocate a tree with level levels on processors lo..lo+num_proc-1
   Assume num_proc is a power of 2 */

tree *TreeAlloc (int level, int lo, int num_proc)
{
  if (level == 0)
    return NULL;
  else {
    tree *new, *right, *left;
    int mid, lo_tmp;

    new = (tree *) ALLOC(lo, sizeof(struct tree));
    new->val = 1;
    new->left =  TreeAlloc(level-1, lo+num_proc/2, num_proc/2);
    new->right = TreeAlloc(level-1, lo, num_proc/2);
    return new;
  }
}
```

Figure 2: Allocation code

2.2 Data Layout

Olden uses data layout information provided by the programmer both at run-time and during compilation. The actual mapping of data to the processors is achieved by including a processor number in each allocation request. Olden provides a library routine, `ALLOC`, that allocates memory on a specified processor, and returns a pointer that encodes both the processor name and the local address of the allocated memory. To assist the compiler, the programmer also provides local path lengths, a hint about the expected run-time layout of the data.

Since the heap is distributed and communication is expensive, to get good performance, the programmer must place related pieces of data on the same processor explicitly. Using `ALLOC`, the programmer can specify the layout of the data structure across the processors. For a binary tree, it is often desirable to place large subtrees together on the same processor, as it is expected that subtrees contain related data.

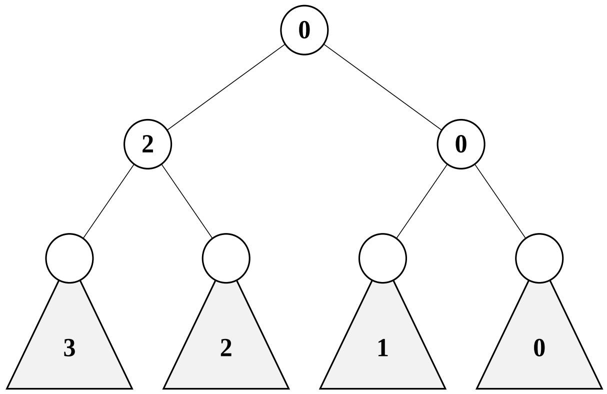

Figure 3: Result of allocation on 4 processors

In Figure 2, we give an example function that allocates a binary tree such that the subtrees at a fixed depth are distributed evenly across the processors (where the number of processors is a power of two). Figure 3 shows the distribution of a balanced binary tree that would be created by a call to `TreeAlloc` on four processors with `lo` equal to zero.

2.2.1 Local Path Lengths

To assist the compiler in managing communication, we allow the programmer to provide a quantified hint regarding the layout of a recursive data structure. A *local path length* represents the expected number of adjacent nodes in the structure that are all on the same processor, measured along a path. Each pointer field of a data structure has a local path length, either specified by the programmer or a compiler-supplied default. Associating a local path length, l, with a field, F, of a structure indicates that, on average, after traversing a pointer along field F that crosses a processor boundary, there will be a path of l adjacent nodes in the structure that reside on the same processor.

Determining the local path length is simplest for a linked list. Consider, for example, the linked list in Figure 4. In this list, there are two nodes on Processor 0,

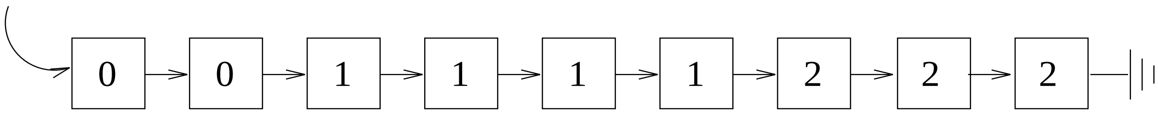

Figure 4: A simple linked list

```
typedef struct tree {
    int val;
    struct tree *left {6};
    struct tree *right {6};
} tree;
```

Figure 5: Sample local-path-length hint

followed by four nodes on Processor 1, and three nodes on Processor 2. Consequently, the local path length for the pointer field would be $\frac{2+4+3}{3}$, which is 3.

For structures with more than one pointer field, such as the tree used in `TreeAdd`, determining the appropriate value for the local path length is more complicated. Local path lengths, however, are merely a hint, and may be approximated or omitted (in which case a default value is used). The compiler's analyses are insensitive to small changes in the local path lengths, and incorrect values do not affect program correctness. In most cases, it will suffice to estimate the value using a high-level analysis of the structure. Recall the allocation shown in Figure 3. Suppose the height of the tree is 12. On average, a path from the root of the tree to a leaf crosses a processor boundary once. We can therefore estimate the local path length as 6 for both of the pointer fields. In Figure 5, we show a data structure declaration with local-path-length hints.

In Chapter 7, we describe the Olden profiler and also provide a mathematically rigorous presentation of local path lengths. The profiler automatically computes the local path lengths of a data structure by performing a run-time analysis. For nine of the eleven benchmarks we implemented, using the compiler-specified default local path lengths yielded the same performance as using the exact values computed by the profiler. In the two cases where the profiler-computed values differed significantly

```
int TreeAdd (tree *t)
{
    if (t == NULL)
        return 0;
    else {
        tree        *t_left;
        future_cell_int f_left;
        int         sum_right;


        t_left = t->left;
        f_left = futurecall (TreeAdd, t_left);

        sum_right = TreeAdd (t->right);

        return (touch(f_left) + sum_right + t->val);
    }
}
```

Figure 6: `TreeAdd` function using `future`.

from the defaults, it was straightforward to estimate the correct values. Since the programmer can guess or use default local-path-length values, and then verify these with the profiler, it has never been necessary to perform a detailed analysis of local path lengths.

2.3 Marking Available Parallelism

In addition to specifying a data layout, the programmer must also mark opportunities for parallelism. In Olden, this is done using futures, a variant of the construct found in many parallel Lisps [Hal85]. The programmer may mark a procedure call as a *futurecall*, if it may be evaluated safely in parallel with its parent context. A *touch* must also be inserted by the programmer before the return value is used.

In our `TreeAdd` example, the two recursive calls on the left and right subtrees do not interfere; therefore, they may be performed safely in parallel. To specify this, we mark the first recursive call as a *futurecall*. This indicates that the continuation of the first recursive call, namely the second recursive call, may be done in parallel with the futurecall. Since the continuation of the second recursive call contains no work that can be parallelized, we do not mark it as a future. Then, when the return value is needed, we must first explicitly touch the future cell. In Figure 6, we show the `TreeAdd` program as modified to use futures.

As we will discuss in Section 3.2, thread creation is closely tied to the data layout. If large groups of related data are placed together, as in the example in Figure 2, Olden will generate a small number of large-granularity tasks, and thus be more efficient. In this example, one thread will be generated for the subtree allocated on each processor. These threads should have large granularity, and good load balancing.

2.4 Summary

Our approach in Olden is to have the programmer specify the data distribution and mark available parallelism. The system then handles work distribution, communication, and synchronization. Given a suitably restricted C program with `ALLOC`, local-path-length hints, and futures as specified, the Olden compiler will generate an SPMD program that handles references to global heap pointers correctly and calls the Olden library routines as needed. These routines manage the threads, and perform the requisite communication between the processors.

Chapter 3

Execution Model

Once the programmer has specified the data layout and identified opportunities for parallelism, the system must still handle communication, work distribution, and synchronization. In this chapter, we describe how Olden handles remote references and how Olden extracts parallelism from the computation. To handle remote references, Olden uses a combination of computation migration, which moves the computation to the data, and software caching, which brings a copy of the data to the computation. Parallelism is introduced using futures.

3.1 Handling Remote References

When a thread of computation attempts to access data from another processor, communication must be performed to satisfy the reference. In Olden, we provide two mechanisms for accessing remote data: computation migration and software caching. These mechanisms may be viewed as duals. Computation migration sends the thread of computation to the data it needs; by contrast, software caching brings a copy of the data to the computation that needs it. The appropriate mechanism for each point in the program is selected automatically by the system. In this section, we describe these two mechanisms; in Chapter 5, we describe our compile-time heuristic that selects between them.

3.1.1 Computation Migration

The basic idea of computation migration is that when a thread executing on Processor P attempts to access a location residing on Processor Q, the thread is migrated from P to Q. Full thread migration entails sending the current program counter, the thread's stack, and the current contents of the registers to Q. Processor Q then sets up its stack, loads the registers, and resumes execution of the thread at the instruction that caused the migration. Processor P remains idle until another (or the same) thread migrates to it.

Recall that we view a memory address as consisting of a pair of a processor name and a local address. This information can be encoded as a single address, and the address-translation hardware can be used to detect non-local references [AL91]. When Processor P executes a load or store instruction that refers to Q's memory, the instruction traps to a library routine. The trap handler is responsible for packaging up the thread's state information and sending it to Processor Q. Notice that if the entire stack is sent with the thread, stack references will always be local and cannot cause a migration. In our current implementation, we do not use the address-translation hardware to detect non-local accesses. Instead, the compiler inserts explicit checks into the code to test the encoded pointer and migrate the thread as needed. Compiler checks are preferable when the ratio of cost of faults to cost of tests exceeds the ratio of tests to faults [AL91]. Preliminary experiments suggested the use of address-translation hardware would improve performance slightly; however, operating-system support was not available on our platform.

Full thread migration is quite expensive, because the thread's entire stack is included in the message. To make computation migration affordable, we send only the portion of the thread's state that is necessary for the current procedure to complete execution: the registers, program counter, and current stack frame. In Section 3.3, we will explain computation migration in the context of a real program; here, we use the example in Figure 7 to illustrate the concepts. During the execution of `H`, the computation migrates from P to Q. Q receives a copy of the stack frame for `H`, which it places on its stack. Note, however, that when it is time to return from

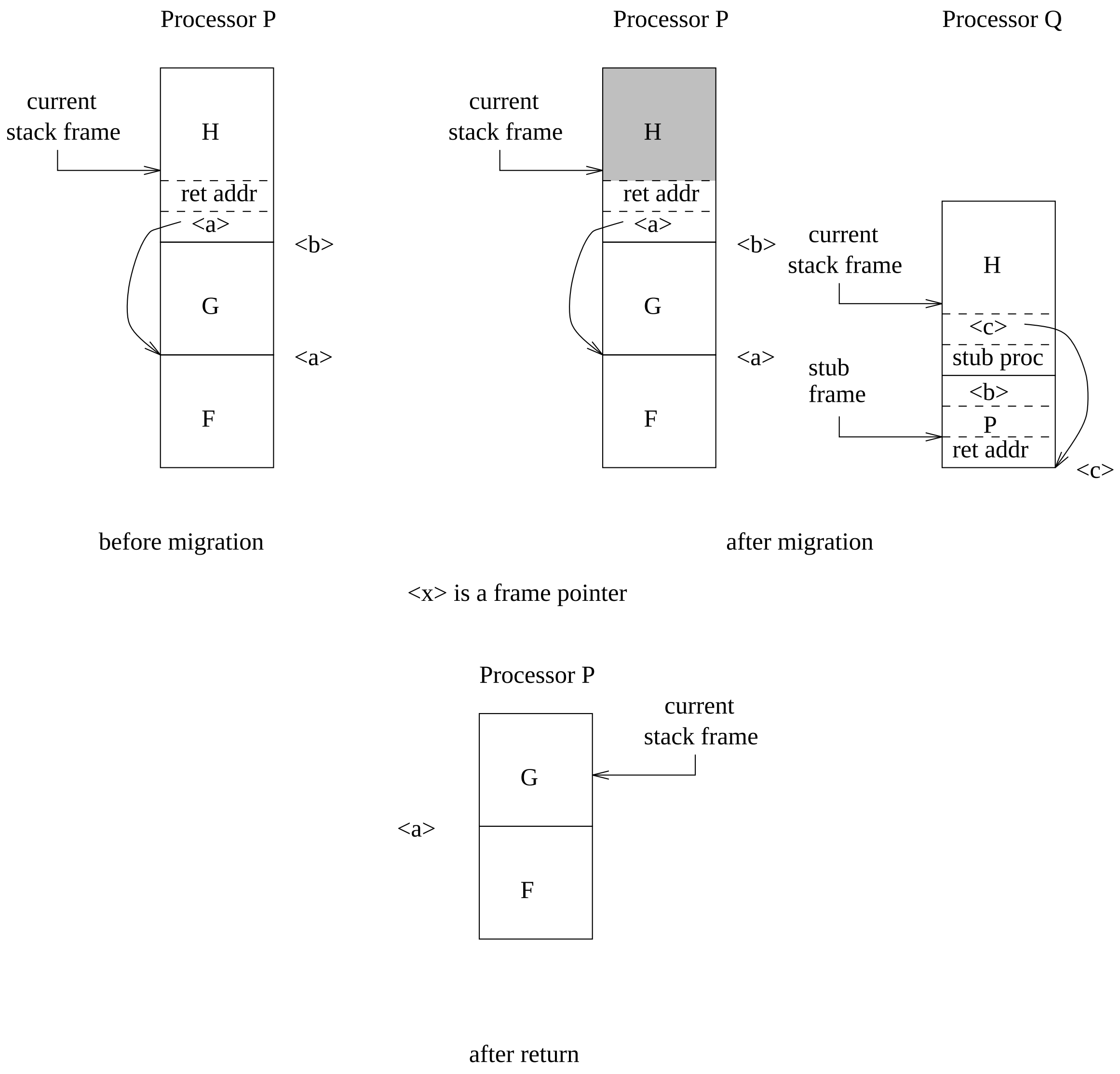

Figure 7: Stack during computation migration (stacks grow up)

the procedure, it is necessary to return control to Processor P, because it has the stack frame of the caller. To accomplish this, Q places a stack frame for a special return stub directly below the frame for `H`. This frame holds the return address and the return frame pointer for the currently executing function. The return address stored in the frame of `H` is modified to point to a stub procedure. The stub procedure migrates the thread of computation back to P by sending a message that contains the return frame pointer ($< b >$), and the contents of the registers. Processor P then completes the procedure return by loading the return address from its copy of the frame, deallocating its copy of the frame, and then restarting the thread at the return address. Note Q does not need to return the stack frame for `H` to P, as it will be deallocated immediately.

Allocation

Recall that the programmer is responsible for specifying a processor name with each memory-allocation request. The allocation routine, which is part of our run-time system, explicitly migrates the thread when a non-local allocation is requested. The Olden compiler expands this migration test inline to ensure that the object initialization can be also completed before the thread migrates back to the processor that initiated the allocation.

Discussion

The primary purpose of migrating the computation to the processor that owns the data rather than migrating the data is to exploit locality. Our system relies on the programmer to choose a layout that places related data together, for example, by placing all the data in a subtree at some fixed depth in the tree on the same processor. If the programmer has chosen a good data layout, then migrating the computation to the data allows the system to perform the computation on the processor that owns most of the required data. This can reduce the amount of interprocessor communication substantially and thereby reduce execution time. Recent work by Hsieh et al. [HWW93] supports this claim.

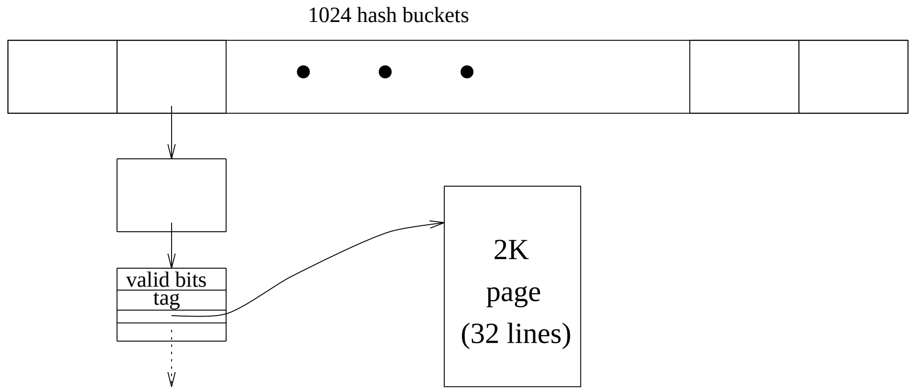

Figure 8: Olden's software cache

3.1.2 Software Caching

There are occasions where it is preferable to move the data rather than the computation. To accomplish this, we use a software-caching mechanism that is very similar to the caching scheme in Blizzard-S [SFL*94]. Each processor uses its local memory as a large, fully associative, write-through cache. A write-through cache is used because these messages can be sent cheaply from user level on the CM-5, and this allows us to overlap the update messages with computation. As in Blizzard-S, we perform allocation on the page level, and perform transfers at the line level.[1] The main difference between Olden's cache and that in Blizzard-S is that we do not rely on virtual-memory support. Consequently, rather than using a very large virtual table to translate addresses (pages of this table are allocated on demand by trapping page faults), we instead use a 1K hash table with a list of pages kept in each bucket. Figure 8 shows the translation table. Since entries are kept on a per-page basis, the chains in each bucket will tend to be quite short (in our experience, the average chain length is approximately one).

The Olden compiler directly inserts software checks before each heap reference that uses software caching. The table lookup itself is very similar to that of Blizzard-S, with additional code to search the lists stored in the hash table. In addition to

[1]In Olden, a page has 2K bytes, and a line has 64 bytes.

checking the valid bit for the line, as done in Blizzard-S, our lookup also returns a tag used to translate the address from a global to a local pointer (in Blizzard-S this translation is performed automatically by the virtual-memory hardware). For both systems, in the event that the page is not allocated or the line is not valid, the appropriate allocation or transfer is performed by a library routine.

Cache coherence

Once we introduce a local cache at each processor, we must provide a means to ensure that no processor sees stale data. For a uniprocessor, this is quite straightforward. One simply requires that any read on a location return the value of the most recently completed write to that location. Lamport's *sequential consistency* [Lam79] extends this idea to multiprocessor systems. A system is said to be sequentially consistent if the result of any execution is the same as if the operations of all the processors were interleaved in some sequential order. Each processor must see the same sequential order. A small two-processor example of sequential consistency is given in Figure 9. Since each processor's code must run sequentially, there are six possible traces. For each, we give the values of `A` and `B` after the program completes.

```
   P1          P2
1) A=0      3) A=1
2) B=A*2    4) B=A+3

Possible orderings
1,2,3,4:  A=1, B=4
1,3,2,4:  A=1, B=4
1,3,4,2:  A=1, B=2
3,4,1,2:  A=0, B=0
3,1,4,2:  A=0, B=0
3,1,2,4:  A=0, B=3
```

Figure 9: Sequentially consistent executions

Although sequential consistency is the most intuitive model, implementations of this model tend to have excessive latency associated with remote memory accesses.

As a result, many systems implement a weaker consistency model, such as *release consistency* [GLL*90]. Release consistency takes advantage of additional information about shared accesses to allow more memory operations to be outstanding. Specifically, the programmer marks accesses that are used for synchronization as acquires or releases of locks. Synchronization accesses are *performed* in a sequentially consistent manner. Other ordinary accesses must be performed before any subsequent releases complete, and after all preceding acquires have completed.

```
   P1              P2
1) A=2          7) acquire L1
2) B=1          8) C=A+B
3) acquire L1   9) release L1
4) A=3
5) B=4
6) release L1

Assume all variables are initially 0

Possible orderings seen by P2
1,7,8,9,2,3,4,5,6:  C=2
1,2,7,8,9,3,4,5,6:  C=3
1,2,3,4,5,6,7,8,9:  C=7
2,7,8,9,1,3,4,5,6:  C=1
```

Figure 10: Release consistency

Figure 10 gives an example of a program executing on a release-consistent system. Although on P1, statement 1 always occurs before statement 2, processor `P2` may see these two writes out of order, resulting in a possible value of 1 for `C` that could not be obtained on a sequentially consistent system. The synchronization accesses, however, are performed in a sequentially consistent manner, consequently statements 3-6 and 7-9 may not be interleaved. Gharachorloo et al. [GLL*90] prove that if a program is *synchronized*, that is, all accesses to shared data are protected by locks, release consistency will give the same results as sequential consistency.

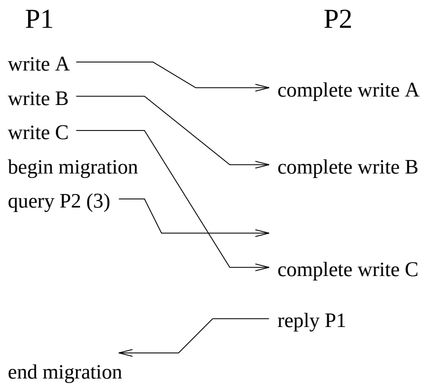

Figure 11: Committing writes

The idea of synchronized programs maps nicely onto Olden, as each migration sent may be viewed as releasing a lock, and each migration received as acquiring one. Since the semantics of Olden's futures require that no thread can read a data item while another thread is writing it, the "virtual locks" may be perceived as locking all of the data that is written by the thread. An Olden program running with any coherence protocol that provides release consistency with respect to these virtual locks will have the same semantics as the same program running on a sequentially consistent system.

To satisfy the first condition of release consistency, that all ordinary accesses complete before any subsequent release, we keep track of how many writes have been sent to each processor. Each processor maintains a counter of the number of writes sent to every other processor, and the number of writes received from every other processor. At an outgoing migration (release), we examine the counters for each processor that has been sent writes to verify that the writes have completed. The migration is not allowed to complete until all of the writes have completed. An example of this is given in Figure 11. `P1` writes three shared variables whose home is on `P2`. At each write, a message is sent to `P2` with the new value. When the thread attempts to migrate, a message is sent to `P2` indicating that three writes have been attempted. When `P2` has received three writes from `P1`, it sends a response. `P1` is then able to complete the migration.

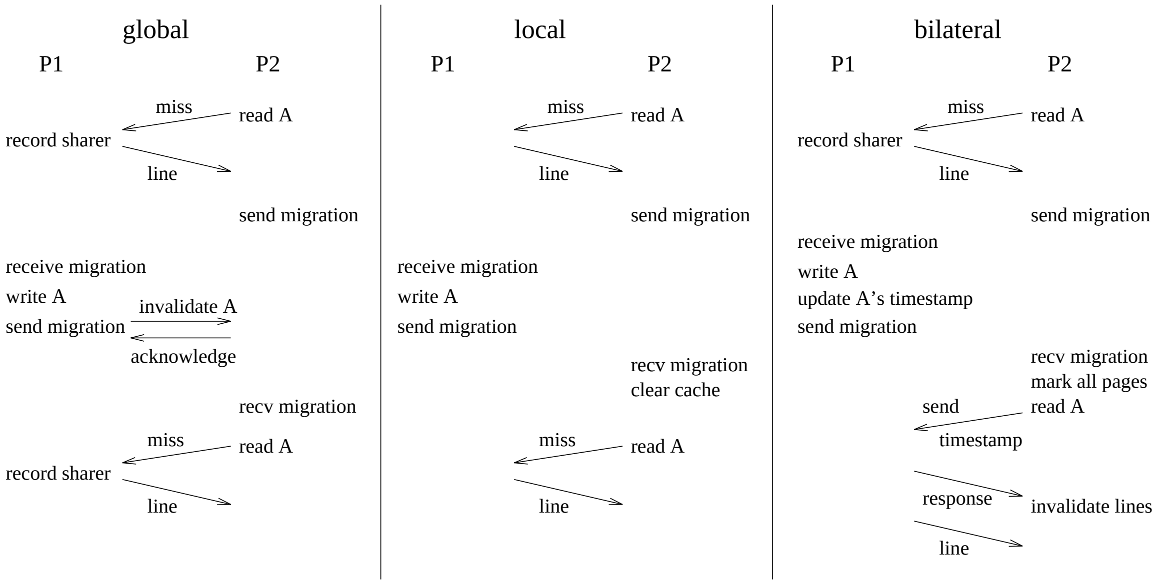

Figure 12: Example using three coherence protocols

To satisfy the second condition, that ordinary accesses complete after all preceding acquires complete, we provide three different protocols. Each of these must guarantee that after receiving a migration (an acquire), the processor sees the most up-to-date copy of the data. In Figure 12, we give an example where a shared variable is read by `P2`, written by `P1`, and then read again by `P2`. The events and messages for each of the three schemes are shown.

Our first coherence protocol is an adaptation of *eager release consistency* [CBZ91]. In an eager release-consistent program, at each release all of the information regarding invalidations is forwarded immediately to other processors, and acknowledgments are collected before the release is allowed to complete. We have implemented eager release consistency in Olden by having the compiler insert code to track each write into the heap. These writes are tracked at the line level, by keeping a vector of *dirty bits* for each shared page. Additionally, the run-time system tracks sharers when it receives cache requests. Just as we performed allocations at the page level, we also track sharers at the page level, to reduce the amount of state information required. At each migration, the run-time system sends invalidations to each sharer of a page, indicating which lines have been written on that page. This does not introduce

Table 1: Summary of Olden's coherence mechanisms

	global	local	bilateral
track writes	yes	no	yes
outgoing migration	commit writes invalidate sharers	commit writes	commit writes increment timestamps
incoming migration		clear cache	mark pages to miss on first access
first miss on page	get line	get line	send timestamp to home invalidate lines clear mark get line
subsequent misses	get line	get line	get line

false sharing, since writes are tracked at the line level, but could cause some spurious invalidation messages to be sent (an invalidation of a line might be sent to a processor that does not share that line). This scheme may be referred to as a *global-knowledge* scheme, as invalidations are performed on the basis of knowledge obtained from other processors.

A model that requires no communication is called a *local-knowledge* scheme. Prior local-knowledge schemes (for example, [CV88, DK93, MB92]) have relied on the compiler specifically inserting code to invalidate the local cache. We implement a local-knowledge scheme in Olden using the run-time system, by having each processor invalidate its entire cache upon receiving a migration (acquire). Since all writes were committed at the previous release, the processor performing the acquire will get up-to-date copies of all the shared data it accesses. This scheme will have unnecessary misses on data that was not written since the previous acquire. We have made a slight improvement by noticing that on returns we need only invalidate cached copies of lines from processors whose memories have been written by the returning thread. (Since the thread cannot read data written by other concurrently executing threads, it need only invalidate cache lines that it might have updated while on a remote processor.)

```
Data Structures.
Hash table for cached pages, with linked list of page information blocks
    each entry contains:
       tag
       pointer to local copy of page
       valid_vector
       remember_vector (bilateral only)
       pointer to next

Hash table for local writes (global and bilateral only)
    each entry contains:
       written_vector
       old_written, older_written (bilateral only)

WritesSent, WritesReceived:  arrays of size number_of_processors

Common Procedures.
RemoteWrite(address,value)
    SendRequest(owner(address),PerformWrite,address,value,MyNodeId)
    WritesSent[owner(address)] += 1

PerformWrite(address,value,nodeid)
    write value at address
    WritesReceived[nodeid] += 1
    TrackLocalWrite(address) (global and bilateral only)

CommitWrites()
    for i = 1 to number_of_processors do
       if (WritesSent[i] > 0)
         SendRequest(i,Wait_for_Writes,MyNodeId,WritesSent[i])
         WritesSent[i] = 0
         messages_sent += 1
    Wait until messages_sent replies are received

Wait_for_Writes(nodeid,num)
    while (WritesReceived[nodeid] < num) process incoming writes
    WritesReceived[nodeid] = 0
    Send reply to nodeid
```

Figure 13: Coherence pseudo-code

Common Procedures.
```
GetLine(address)
    SendRequest(owner(address),SendLine,MyNodeId,address)
    Await reply
    Fill in local cache copy of line
    Mark line valid

SendLine(nodeid,address)
    Send reply with line to nodeid
    Add nodeid to sharer list for page containing address (global only)
```

Protocol-Specific Procedures.
```
ClearCache() (local only)
    for each hash table entry
       for each page block in linked list
          valid_vector = 0

TrackLocalWrite(address) (global and bilateral only)
    Lookup address in hash table
    Set written bit for line

InvalidateSharers() (global only)
    foreach hash entry
       foreach page in list
          foreach sharer in sharer list
             SendRequest(sharer,Invalidate,page,written_vector,MyNodeId)
    Await acknowledgments

Invalidate(page,written_vector,nodeid) (global only)
    Find page in hash table lists
    valid_vector(page) = valid_vector(page) and not written_vector
    acknowledge invalidation

MarkPagesToMiss() (bilateral only)
    foreach hash entry
       foreach page in list
          remember_valid(page) = valid(page)
          valid(page) = 0
```

Figure 14: Coherence pseudo-code part 2

Protocol-Specific Procedures.

```
IncrementTimestamps() (bilateral only)
    foreach entry in local writes table
       if (written_vector(entry) is not 0)
          older_written(entry) = old_written(entry)
          old_written(entry) = written_vector(entry)
          written_vector(entry) = 0
          timestamp(entry) += 1

SendTimestampHome(address) (bilateral only)
    (called only on miss if remember_valid is non-zero)
    SendRequest(owner(address),GetTimestamp,address,
                timestamp(address),MyNodeId)
    Wait for response containing invalidate_vector and (possibly) line
    valid_vector(page) = remember_valid(page) and not invalidate_vector
    remember_valid(page) = 0 (clears mark)

GetTimestamp(address,timestamp,nodeid) (bilateral only)
    Find entry in local writes table corresponding to address
    if (timestamp = timestamp(entry))
       send response with written_vector(entry)
    else if (timestamp = timestamp(entry)-1)
       send response with (written_vector(entry) or old_written(entry))
    else if (timestamp = timestamp(entry)-2)
       send response with (written_vector(entry) or old_written(entry)
                          or older_written(entry))
    send line if nodeid's copy is no longer valid
```

Figure 15: Coherence pseudo-code part 3

Finally, we implemented a *bilateral scheme*, that combines both local and global knowledge. Again, the compiler inserts code to track writes; however, sharers are not tracked. Instead, a timestamp is kept for each page. Since each page has a different timestamp, and only one processor is the owner of each page, we do not have to keep timestamps consistent. These timestamps, which are counters, are incremented when a migration leaves a processor, if the corresponding pages have been written. On receiving a migration, a processor marks all of its pages, so that they miss on the first access. (This is similar to the use of epoch bits, as proposed by Darnell et al. [DK93].) On this miss, the handler sends the page's timestamp to its home, and is informed which lines need to be invalidated. The bilateral scheme greatly reduces the amount of invalidation traffic; however, it still must pay the cost of tracking writes.

A summary of the three coherence schemes is given in Table 1, and pseudo-code is given in Figures 13, 14 and 15. Each of the three schemes described will perform differently depending on the pattern of remote data references. We examine these tradeoffs in detail in Section 6.1.

3.2 Introducing Parallelism

While computation migration and software caching provide mechanisms for operating on distributed data, they do not provide a mechanism for extracting parallelism from the computation. When a thread of computation migrates from Processor P to Q, P is left idle. In this section, we describe a mechanism for introducing parallelism. Our approach is to introduce *continuation-capturing* operations at key points in the program. When a thread migrates from P to Q, Processor P can start executing one of the captured continuations. The natural place to capture continuations is at procedure calls, since the return linkage is effectively a continuation. This provides a fairly inexpensive mechanism for labeling work that can be done in parallel. In effect, this capturing technique chops the thread of execution into many pieces that can be executed out of order. Thus the introduction of continuation-capturing operations must be based on an analysis of the program, which can be done either by a parallelizing compiler targeted for Olden or by a programmer using Olden directly.

Futures

Our continuation-capturing mechanism is essentially a variant of the *future* mechanism found in many parallel Lisps [Hal85]. In the traditional Lisp context, the expression `(future` e`)` is an annotation to the system that says that e can be evaluated in parallel with its context. The result of this evaluation is a *future cell* that serves as a synchronization point between the child thread that is evaluating e and the parent thread. If the parent *touches* the future cell, that is, attempts to read its value, before the child is finished, then the parent blocks. When the child thread finishes evaluating e, it puts the result in the cell and restarts any blocked threads.

Our view of futures, which is influenced by the *lazy-task-creation* scheme of Mohr et al. [MKH91], is to save the futurecall's context (return continuation) on a work list and to evaluate the future's body directly.[2] If a migration occurs in the execution of the body, then we grab a continuation from the work list and start executing it; this is called *future stealing*. In addition, all touches in Olden are done explicitly using the `touch` operation.

Futures in More Detail

Our scheme for introducing parallelism consists of two operations: `futurecall` and `touch`. In the remainder of this section, we describe these operations and their related data structures. The two data structures are the future cells and the future stack, which is a stack of future cells that serves as a work list. A future cell can be in one of four states (Figure 16 gives the allowable state transitions for a future cell):

Empty: This is the initial state and the future cell contains the information necessary to restart the parent.

[2]This is also similar to the workcrews paradigm proposed by Roberts and Vandevoorde [RV89].

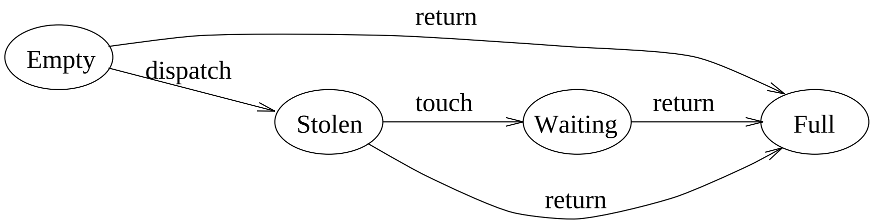

Figure 16: Future cell state transitions.

Stolen: This is the state when the parent continuation has been stolen.

Waiting: This is the state when the parent continuation has been stolen and subsequently touched the future. The future cell contains the continuation of the blocked thread (note: no future can be touched by more than one thread).

Full: This is the final state of a future cell, which contains the result of the computation.

The future stack is threaded through the execution stack. In the normal case, when no migration or stealing occurs, only a few extra instructions are required to allocate the future cell and touch the result (see Section 4.2.2 for details).

The `futurecall` operation is responsible for allocating a new future cell, pushing the cell on the future stack, and calling the specified routine. Upon return from the routine, the cell is filled with the return result, following an examination of the cell's state. If it is **Empty**, then the cell is popped off the future stack, and execution continues normally; otherwise the corresponding continuation has been stolen, and the cell is either in the **Waiting** or **Stolen** state. In the **Waiting** state, the waiting continuation is resumed; in the **Stolen** case, there is nothing more to do, and the processor must look for new work.

Whenever a processor needs new work, it calls the run-time system routine `MsgDispatch`. This routine does the actual stealing of continuations from the future stack. If the stack is not empty, it pops a future cell from the top of the stack and executes the resume continuation from the cell. To signal that the parent continuation has been stolen, the state of the future cell is changed to **Stolen**.

Touching a future that is not **Full** causes the currently executing thread of computation to block. The `touch` operation changes the state of the future from **Stolen** to **Waiting**, stores the continuation of the touch in the cell, and looks for other work to do by calling `MsgDispatch`. Olden's programming model requires that only one `touch` be attempted per future and that touch must be done by the future's parent thread of computation. As a result, the state of a future will be either **Stolen** or **Full** when a touch is attempted.

Discussion

The design of futures in Olden was influenced heavily by our intent to use parallelizing compiler technology [Hen90, HN90, HHN92, Lar91] to insert them automatically. Both the restrictions mentioned above follow from this intention. These two restrictions allow us to allocate future cells on the stack rather than the heap, which helps make futures less expensive. Although we currently insert futures by hand rather than automatically, in the applications we examined, we have not found any occasions when it would be desirable to violate these restrictions.

Another important fact to note about the use of futures in Olden is that when a processor becomes idle, it steals work only from its own worklist. A processor never removes work from the worklist of another processor. The motivation for this design decision was our expectation that most futures captured by a processor would operate on local data. Although allowing another processor to steal this work may seem desirable for load-balancing purposes, it would simply cause unnecessary communication. Instead, load balancing is achieved by a careful data layout. This is in contrast to Mohr et al.'s formulation, where an idle processor removes the oldest piece of work from another processor's work queue.

The choice of using a queue versus a stack to hold pending futures is related to the question of who can steal work from a processor's worklist. The breadth-first decomposition that results from using a work queue is desirable when work can be

```
int TreeMultAdd (tree *t, tree *u)
{
    if (t == NULL) {
        assert(u == NULL);
        return 0;
        }
    else {
        tree        *t_left, *u_left;
        future_cell f_left;
        int         sum_right;

        assert(u != NULL);
        t_left = t->left;                /* may cause a migration */
        u_left = u->left;                /* may cause cache miss */
        f_left = futurecall (TreeMultAdd, t_left, u_left);

        sum_right = TreeMultAdd (t->right,u->right);

        return (touch(f_left) + sum_right + t->val*u->val);
    }
}
```

Figure 17: `TreeMultAdd` with Olden annotations.

stolen by another processor, because it tends to provide large granularity tasks. A stack-based implementation is appropriate for Olden, because when a processor steals its own work, it is better for it to steal work that is needed sooner rather than later.

3.3 A Simple Example

To make the ideas of this chapter more concrete, we present a simple example.[3] `TreeMultAdd` is a prototypical divide-and-conquer program, which computes, for two identical binary trees, the sum of the products of the values stored at corresponding nodes in the two trees.

[3]This is an abstraction of the Union benchmark. We present Union and other benchmarks in Chapter 6.

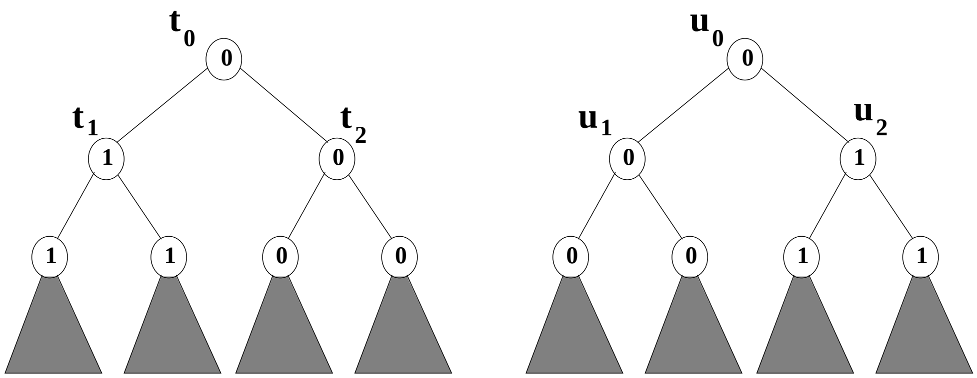

Figure 18: Example input for `TreeMultAdd`

Figure 17 gives an implementation of `TreeMultAdd` that has been annotated with futures. We mark the left recursive call to `TreeMultAdd` with a `futurecall`;[4] the result is not demanded until after the right recursive call. We assume that the compiler has chosen to use computation migration to satisfy remote dereferences of `t` and software caching to satisfy remote dereferences of `u`.

To understand what this means in terms of the program's execution, consider a call to `TreeMultAdd` on the two trees whose layout is shown in Figure 18. Consider the first call to `TreeMultAdd`, made on Processor 0 with t_0 and u_0 as arguments. Since both `t` and `u` are local, no communication is needed to compute `t_left` and `u_left`. Then, once the recursive call on the left subtrees is made, `t`, now pointing to t_1 is non-local. Since the compiler has selected computation migration for handling remote references to `t`, the statement `t_left = t->left` will cause the computation to migrate to Processor 1. After the migration, `u`, now pointing to u_1, is also non-local, and the statement `u = u->left` will cause a cache miss. As the computation continues on the subtrees rooted at t_1 and u_1, note that all references to `t` will be local and all references to `u` remote. Consequently, no further migrations will occur, and the subtree rooted at u_1 will be cached on Processor 1. Once the computation on the subtrees rooted at t_1 and u_1 is complete, a return message will be sent to Processor 0.

[4]Note that using a `futurecall` on the right subtree is not cost effective, since there is very little computation between the return from `TreeMultAdd` and the use of its result.

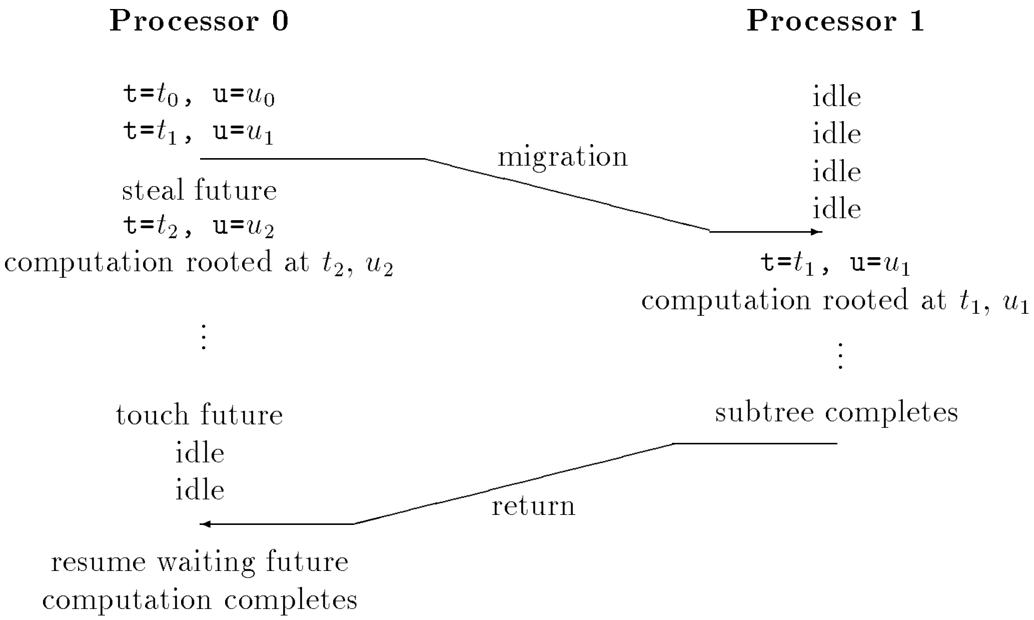

Figure 19: Execution trace for `TreeMultAdd`

Meanwhile, after the migration, Processor 0 is idle. Therefore, it will steal the continuation of the first recursive call. This continuation starts the recursive call on the right subtrees, rooted at t_2 and u_2. Note all of the dereferences of `t` are local, and all of the dereferences of `u` remote. The subtree rooted at u_2 will be cached on Processor 0, and no further migrations will occur. Once this computation completes, Processor 0 will attempt to touch the future cell associated with the call with arguments t_1 and u_1. At this point, execution must wait for the return message from Processor 1. An execution trace for both processors is shown in Figure 19.

As previously mentioned, since the compiler has chosen to use caching to satisfy remote dereferences of `u`, the subtree rooted at u_1 will end up being cached on Processor 1, and the subtree rooted at u_2 on Processor 0. Had the compiler chosen to use migration for dereferences of both `t` and `u`, the computation would have bounced back and forth between the two processors. Here we see the advantage of using both computation migration and software caching. By using migration, we obtain locality for all references to the tree `t`, and by using caching, we prevent the computation from migrating back and forth repeatedly between the processors. We examine further the benefits of using both computation migration and software caching in Chapters 5 and 6.

Neither the reference to `t->right` nor the reference to `t->val` can ever be the source of a migration. Once a non-local reference to `t->left` causes the computation to migrate to the owner of `t`, the computation for the currently executing function will remain on the owner of `t` until it has completed (since we assumed references to `u` will use caching rather than migration).

3.4 Summary

In this chapter, we have described the Olden execution model. Given an Olden program, the Olden compiler will generate an SPMD program that automatically handles remote references, and calls the appropriate run-time routines to perform communication and thread management. Remote references are satisfied using a combination of computation migration and software caching; while parallelism is introduced using futures. `TreeMultAdd` illustrates the use of the Olden programming model; we present more realistic benchmarks in Chapter 6.

Chapter 4

Implementation Details

We have implemented prototypes of our execution model that run on the Intel iPSC/860 hypercube and the Thinking Machines CM-5. The Intel implementation does not include software caching. The results on this system are similar to those on the CM-5 with caching disabled, and may be found in a prior paper [RCRH95].

As mentioned earlier, we call our system *Olden*. The input to Olden is a C program annotated with `futurecalls` and `touches`. Olden consists of a run-time system and a compiler that translates annotated C code into the assembly code for the target machine with embedded calls to the run-time system. The run-time system handles caching, migrations, returns from migrations, and starts new computations by stealing futures. The compiler generates code for futures, touches, and pointer tests, and it uses a non-standard stack discipline to support multiple active threads of computation. We describe the two parts of the system in detail in this chapter.

4.1 The Run-time System

Figure 20 provides an overview of the control structure of the run-time system. The main procedure is `MsgDispatch`, which is responsible for assigning work to a processor when it becomes idle. Captured empty futures are given first priority, then messages from other processors in the order received. We chose this priority scheme,

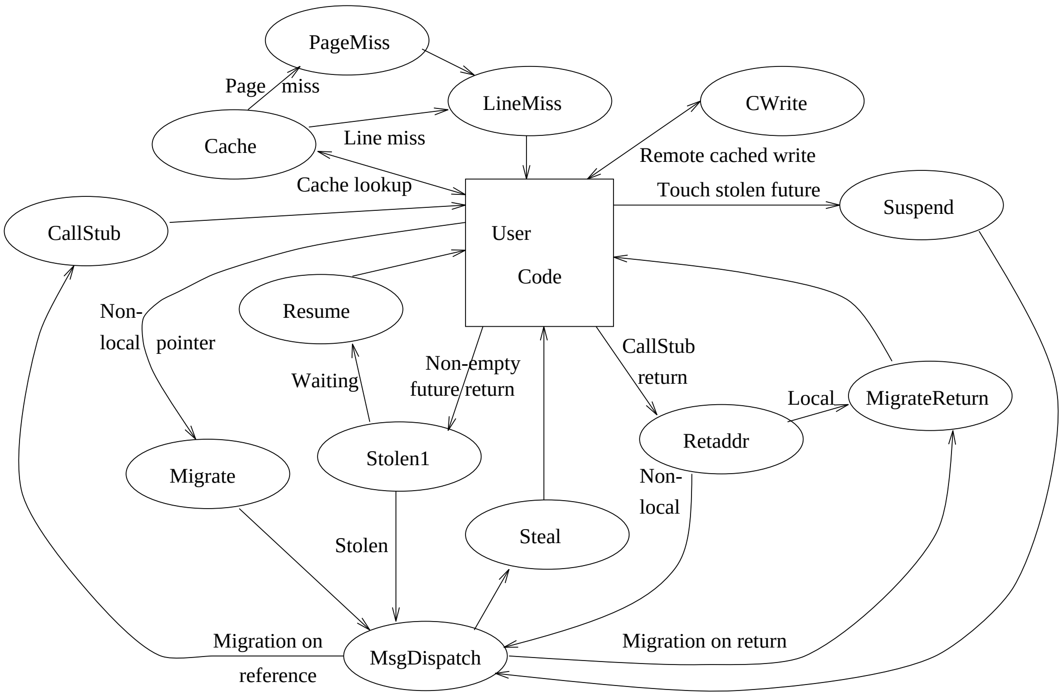

Figure 20: The run-time system (each oval represents a procedure).

because early in a computation a future is likely to spin off work to other processors. Processing futures first allows us to get work to idle processors sooner.

The run-time system processes six different events: cache miss, remote cache write, migrate on reference, migrate on return, steal future, and suspend touch. We focus on caching and migration first; we discuss the implementation of futures in the next section.

4.1.1 Cache Events

As previously mentioned, the compiler will select software caching to handle some remote references. To satisfy a non-local cached reference, the run-time-system caching routine, `Cache` (shown in Figures 21 and 22 with pseudo-code in Figure 23), is called. When there is a miss, `PageMiss` is called, if necessary, to allocate the page, and then `LineMiss` is called to fetch the line. `LineMiss` uses active messages [vECGS92] to obtain the cache line from the remote processor, and then updates the cache table

```
! Cache takes an argument which is an address.  First, it looks up
! this address in the page table (which is hashed).  We assume the
! address of this table is contained in a register (TBL_REG).  If
! the page is not present, we signal an allocation miss.  Otherwise,
! we look at the state information for the page to determine if the
! line is valid.  If so, we return its address, otherwise we have a
! cache line miss.
!
! addresses are of form:[14 bit page #|11 bit page offset|7 bit pn]
.global ___Cache
___Cache:
   and %o0,IDMASK,%l0         ! l0 <- o0 & IDMASK                   (pn)
   srl %o0,7,%l1              ! l1 <- o0 >> 7                 (strip pn)
   and %l1,(PAGE_SIZE-1),%l1  ! l1 <- l1 & PAGE_SIZE-1   (page offset)
   srl %o0,18,%l2             ! l2 <- o2 >> 18                  (page #)
   and %l2,(PG_TBL_SZ-1),%l3  ! l3 <- l2 & PG_TBL_SZ-1  (abbrev page #)
   sll %l0,3,%l4              ! l4 <- l0 << 3                 (pn << 3)
   xor %l3,%l4,%l3            ! l3 <- l4 ^ l3               (hash value)
   sll %l3,4,%l3              ! l3 <- l3 << 4                (hash*16)
   sll %l2,7,%l2              ! l2 <- l2 << 7             (page # << 7)
   or %l2,%l0,%l2             ! l2 <- l0 | l2                      (tag)
                              ! tag is [14 bit page # | 7 bit pn]
   ld [TBL_REG+%l3],%l4       ! l4 <- M[TBL_REG + l3]         (load tag)
   cmp %l4,%l2                ! l4 == l2 ?                (compare tags)
   be ___PageHit;             ! if = goto ___PageHit   (has delay slot)
   add TBL_REG,%l3,%l3        ! l3 <- l3 + TBL_REG       (entry pointer)
   ! At this point, we want to walk down linked list
   ! tag is in l2, page offset is in l1, entry pointer in l3
___PageLoop:
   ld [%l3+NEXT_OFFST],%l3 ! l3 <- M[l3 + NEXT_OFFST]  (entry pointer)
   cmp %l3,%g0             ! l3 == 0 ?                    (compare tags)
   be,a ___PageMiss;       ! if = goto ___PageMiss  (delay slot if goto)
   add %sp,-88,%sp         ! sp <- sp - 88
   ! use max stack offset for all-- coordinate with compiler
   ! If null pointer, goto page miss-- allocate frame ext.
   ld [%l3+TAG_OFFST],%l4  ! l4 <- M[l3 + TAG_OFFST]          (load tag)
   cmp %l4,%l2             ! l4 == l2 ?                      (check tag)
   bne ___PageLoop; nop    ! if not = goto ___PageLoop
   ! must be followed by ___PageHit
```

Figure 21: Cache page-lookup routine for CM-5 (SPARC assembly)

```
   ! immediately preceded by ___PageLoop code
___PageHit:
   ! require page offset to be in %l1, entry pointer in l3
   ld [%l3+PAGE_OFFST],%l0  ! l0 <- M[l3 + PAGE_OFFST]  (page location)
   ld [%l3+VALID_OFFST],%l2 ! l2 <- M[l3 + VALID_OFFST]     (state info)
   srl %l1,6,%l4            ! l4 <-  l1  >>  6          (cache line offset)
   srl %l2,%l4,%l2          ! l2 <-  l2  >>  l4           (place bit in LSB)
   andcc %l2,1,%g0          ! l2 & 1  ==  0?  (Check valid bit for line)
   be ___SignalLineMiss; nop! if = goto ___SignalLineMiss (0=>not valid)
   jmpl %o7+12,%g0;         ! return
   add %l1,%l0,%o0          ! o0 <-  l1 + l2 (add page offset to page addr)
___SignalLineMiss:
   ! At this point, we wish to signal cache line miss
   ! require page to be in l0, page offset in l1, entry in l3,
   ! cache line offset in l4
   add %l3,VALID_OFFST,%o1 ! o1 <-  l3 + VALID_OFFST (arg 2:valid bits)
   mov %l4,%o2             ! o2 <-  l4                 (arg 3:line offset)
   ! Warning--  pages are not page aligned -- align before adding
   andn %l1,63,%o3         ! o3 <-  l1  & ~63              (align with line)
   add %o3,%l0,%o3         ! o3 <-  o3 + l0        (page + lineaddr offset)
   ba ___LineMiss; add %sp,-88,%sp ! goto ___LineMiss
```

Figure 22: Cache line-lookup routine for CM-5 (SPARC assembly)

```
Cache(address)
   pn = address & IDMASK
   page_offset = (address >> 7) and (PAGE_SIZE-1)
   page_no = address >> 18
   abbrev_page_no = page_no and (PG_TBL_SZ-1)
   hash_val = (pn << 3) xor abbrev_page_no
   tag = (page_no << 7) or pn
   entry = PageTable[hash_val]
   while ((entry's tag is not tag) and (not at end of list))
      walk down entry list
   if at end of list, call PageMiss
   valid_vector = entry's valid vector
   line_offset = page_offset >> 6
   check line_offset bit of valid vector
   if not valid, call LineMiss
   page_location = entry's page location
   cache_location = page_location + page_offset
   return cache_location
```

Figure 23: Pseudo-code for cache line-lookup routine

to record that the line is present. Since the line is expected to be returned quickly, the computation suspends waiting for the return message rather than calling the dispatcher.

For cached writes to remote data, the routine `CWrite` is called. Olden uses a write-through cache policy; therefore, a message is sent immediately to the remote processor, which will update the home copy of the data. Since remote writes are infrequent and active messages are fairly cheap, this policy is effective. If the cost of sending a message were greater, the system could accumulate remote writes and send them in a batch. After sending the write, `CWrite` also updates the local cached copy.

4.1.2 Migrate on Reference

For other remote references, the compiler will select migration. The code to test the pointer and call `Migrate`, if necessary, is inserted by the compiler. `Migrate` packages the necessary data and sends it to the appropriate processor. A migration message contains the current stack frame, the argument-build area of the caller, the contents of callee-save registers,[1] and some bookkeeping information. We allocate extra space in each stack frame for the callee-save registers and bookkeeping information, which allows us to construct messages *in situ*, reducing the cost of migrations. Once the migration is complete, the sending processor calls `MsgDispatch` to assign a new task.

When `MsgDispatch` selects an incoming migration message as the next task, it transfers control to `CallStub` to process the message. `CallStub` allocates space on the stack for a return message, the frame, the argument-build area, and a stub frame. Then it copies the migration message into the allocated space. `CallStub` stores the frame size, return-value size, and frame pointer from the migration message into the space allocated for the return message. Then it loads the callee-save register values from the message, adjusts the pointers to the return area and the argument-build area (if they exist), and changes the return address stored in the frame to `retaddr`, the stub return procedure. Finally, it loads the pointer causing the migration, the frame pointer, and the program counter into the appropriate registers and resumes the migrated function on the new processor. When this procedure exits, it will go to the procedure `retaddr`. This code stores the values of the callee-save registers in the return message, sends the message to the processor that began execution of the procedure, and calls `MsgDispatch`.

A simple optimization is added to avoid a chain of trivial returns in the case that a thread migrates several times during the course of executing a single function. `Migrate` examines the current return address of the function to determine whether it points to the return stub. If so, the original return address, frame pointer, and node id are pulled from the stub's frame and passed as part of the migration message. An example of this is given in Figure 24. The procedure `H` migrates from P to Q to R. On the migration to R, Q may deallocate its frame, sending the information from its stub frame to R. When `H` completes on processor R, it may return directly to P, without sending a message through Q. This is analogous to a tail-call optimization

[1] We bypass the SPARC's register window mechanism in our CM-5 implementation. As a result, we have callee-save registers.

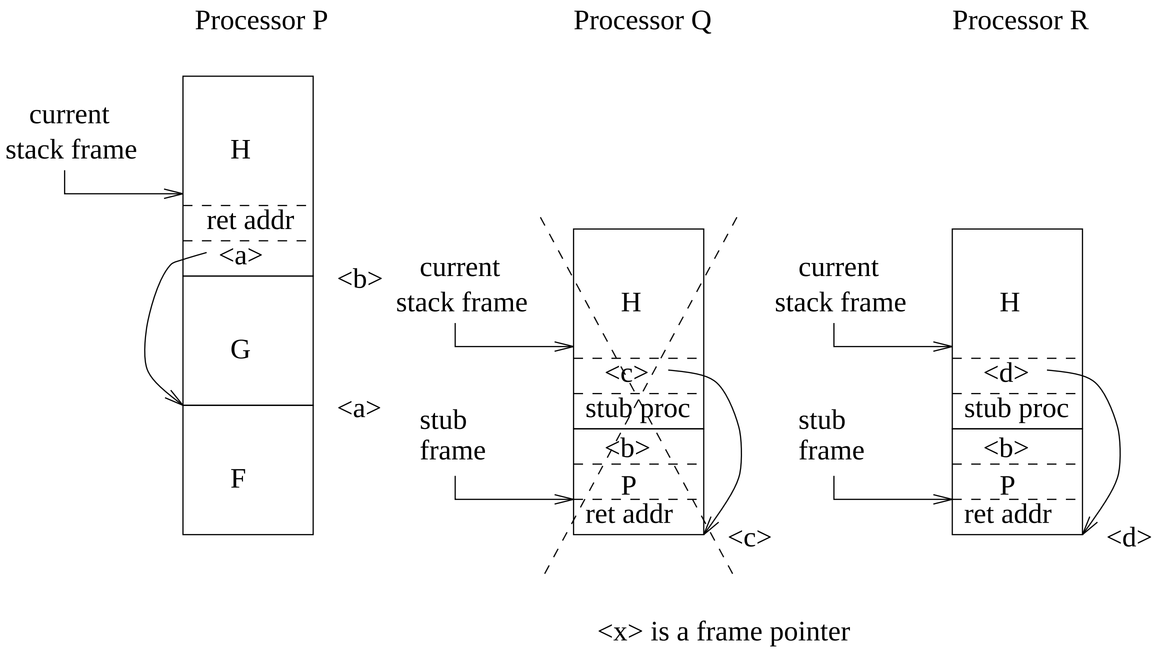

Figure 24: A chain of migrations

and is similar to the *tail-forwarding* optimization used in the Concurrent Smalltalk compiler [HCD89]. A similar optimization is that the return message is not sent if a migrated procedure exits on the same processor as it began (for example, it migrated from P to Q then back to P): `MigrateReturn` is called directly instead.

4.1.3 Migrate on Return

When `MsgDispatch` selects a return message as the next task, it calls `MigrateReturn`, which loads the contents of the callee-save registers from the message, deactivates the frame of the procedure that migrated, and copies the return value to the appropriate place, if necessary. It then resumes execution at the return address of the procedure that migrated. Note that since the procedure exited on the remote processor before the migrate on return, the state of the callee-save registers will be the same as before the procedure was called.

```
! Test pointer in %r28, local pointer returned in %r28
! %l6 contains processor id
xor %r28,%l6,%l0               ! xor pointer with id
andcc %l0,127,%g0              ! test for equality
be,a L.99; srl %r28,7,%r28     ! if equal done, strip off
                               ! processor name
set 176,%o1                    ! frame size
set 72,%o2                     ! frame extension size
mov %r28, %o0                  ! pointer causing migration
call ___Migrate; or %g0,0,%o3  ! Migrate:no structure return
mov %o0,%r28                   ! place return value in register
L.99:
! END of pointer test, using migration
```

Figure 25: Pointer test using migration on CM-5

4.2 The Compiler

The compiler is derived from **lcc** [FH95], an ANSI C compiler.[2] We ported the back end to the appropriate processor (SPARC) and added modifications to handle our execution model. These modifications generate code to test for non-local memory references and call `Migrate` or `Cache` as necessary, to capture futures, and to synchronize on touches.

4.2.1 Testing Memory References

For each dereference of a non-local heap pointer, it is necessary to call the appropriate run-time-system routine. In our implementation, heap pointers consist of a tag indicating the processor where the data resides[3] and a local address, so it is necessary to check the tag explicitly on each heap pointer dereference. **lcc** has a command-line option for generating null pointer checks on all pointer dereferences. We modified this mechanism to generate code that examines the tag to see if the pointer is local,

[2] **lcc** does not have an optimizer.

[3] The range of possible heap addresses dictates the maximum size of the processor tags. On the CM-5, we use the seven low-order bits to hold the tag.

conditionally calls the appropriate library routine to handle a remote reference, and removes the tag to reveal the local address.

The additional overhead of testing each pointer dereference is three integer instructions and a conditional branch on the SPARC. An additional integer instruction is needed to remove the processor name. The code sequence generated by the compiler for testing a pointer (where migration has been selected) is shown in Figure 25. This overhead can be reduced if we can guarantee that a pointer dereference will always be local. For example, in `TreeMultAdd` (on page 32), of the dereferences of `t`, only the reference to `t->left` can be non-local (as discussed in Section 3.3). All subsequent indirect references through `t` are guaranteed to be local, as the computation will have migrated to the processor that owns `t`.[4]

A simple compiler optimization can take advantage of this observation. We say that a dereference of some pointer variable t is *aligned* with another dereference of t, if it is *dominated* by that reference and if there are no intervening migrateable references to other pointers. For example, in `TreeMultAdd`, the references to `t->right` and `t->val` are both aligned with the reference to `t->left`. Even though there are intervening references to `u`, they do not affect the alignment, as references to `u` use caching. The first dereference in such a chain must be tested. Subsequent aligned dereferences are guaranteed to be local, and require only one integer instruction of overhead to obtain the local address from the global pointer. The compiler uses a standard data flow analysis to detect these cases and omit the test, leaving only a single instruction that removes the processor name (`srl %r28,7,%r28`).

For dereferences that use caching, the compiler also inserts code to test for locality. Again, however, the overhead can be reduced from five instructions to one instruction if the compiler can guarantee the reference will be local. If the reference is remote, the compiler calls a routine that checks to see if the line is present in the cache. We chose not to inline this test to reduce code size. The overhead (including the locality test) for a cache hit in the first bucket in the hash chain is 29 instructions. Once the

[4]Note that this observation relies on the fact that function calls return always to the original processor on exit from the called routine.

```
! BEGIN cached pointer test
! pointer and result in %r27
! processor name in %l6
xor %r27,%l6,%l0              ! xor with processor name
andcc %l0,127,%g0             ! check if local
be,a L.187; srl %r27,7,%r27   ! if local,
                              ! strip off processor name
call ___Cache; mov %r27,%o0   ! call cache routine
add %sp,88,%sp                ! spaghetti-stack code
mov %o0,%r27                  ! put result in register
L.187:
! END of cached pointer test
```

Figure 26: Pointer test using caching on CM-5

line has been found in the cache, or added to it by a call to `Cache`, a pointer to the cache copy is returned. A pointer dereference using caching is given in Figure 26.

4.2.2 Compiling `futurecall`

In Section 3.2, we presented the basic ideas underlying our use of futures. In this subsection, we return to the topic of futures to examine our implementation in detail. We review the steps necessary to implement future calls and then discuss several optimizations.

When the back end encounters a `futurecall` annotation, it generates inline code to store the continuation (which consists of the callee-save registers, the frame pointer, and the program counter) in the future cell, initialize the cell's state to **Empty**, and push the cell onto the future stack. Then the call is generated. After the call, code is generated inline to store the return value in the future cell, and check the state of the future. If it is **Empty**, the cell is popped from the stack, and execution continues.

In the case that the cell is not **Empty** (that is, the called function or one of its descendants migrated), we test to see whether the future is **Stolen** or **Waiting**. If it is **Stolen**, then `MsgDispatch` is called to assign work to the processor. If it is

Waiting, there must have been a touch on this future cell. We describe how to resume a waiting thread of computation in the next subsection.

As described, the overhead of future calls is quite large. It is imperative for the capturing of futures to be inexpensive, because most future calls do not contribute parallelism. A future generates parallelism only if the called function (or one of its descendants) migrates. Whether the called function migrates depends on the size and layout of the data structure. A common situation is for the called function to migrate early in a sequence of recursive calls, but then to remain fixed later in the sequence. We have implemented a series of optimizations to reduce the overhead of `futurecalls`.

Our first optimization, called *register unwinding*, eliminates the cost associated with saving callee-save registers at the expense of increasing the cost of steals. This tradeoff is profitable, because stealing a future is much less common than capturing one. Register unwinding exploits the fact that `MsgDispatch` gives stealing futures the highest priority. A descendant of the function, f, associated with the future cell must have been executing immediately before the steal. Therefore, the state of the callee-save registers may be restored by executing the restores of callee-save registers for each procedure in the call chain from the user code function last executing to the function f.

To implement register unwinding, we generate a callee-save restore sequence for each procedure, and store its address at each call site in the procedure. Then, when stealing a future, we traverse the list of frame pointers to the frame that contains the future cell, executing each callee-save restore sequence as we go. Since the address of a restore sequence is stored at the appropriate call site, it can be obtained by reading a word at a constant offset from the return address stored in the frame. For example, consider the scenario shown in Figure 27. When `H` migrates, the run-time system will execute the callee-save restore sequence for `H`, and then `G`. Once this is done, the callee-save registers will contain their original values.

A second optimization is to eliminate the need for storing the state **Empty** in the future cell. The return address for a future call is initially set to the sequence

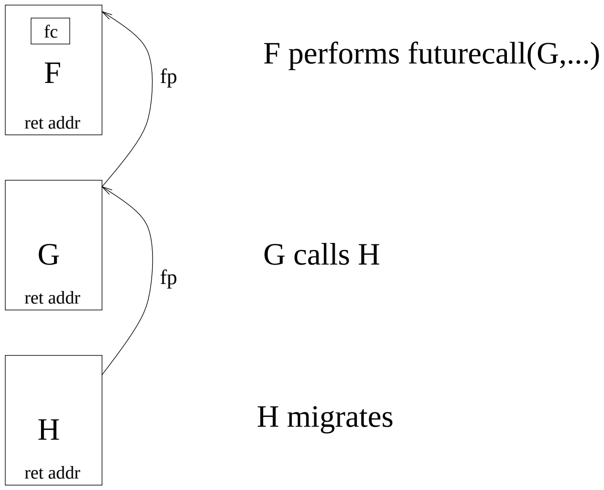

Figure 27: Register unwinding example.

of instructions for an **Empty** future cell. When a future cell is stolen, the return address of the called function is modified to point to code which tests for **Stolen** or **Waiting**, and perform the appropriate action.

Our third optimization allows us to eliminate the store of the frame pointer on entry. We notice that since the future cell is stored in the frame of the caller, we can determine the frame pointer during register unwinding. The frame pointer of the parent continuation is simply the first frame pointer in the chain with an address that precedes the address of the future cell (see Figure 27).

Our final optimization takes advantage of the facts that the future stack is threaded through the execution stack using a linked list and that future cells are four-byte aligned, to eliminate another store. We store the state information in the last two bits of the future cell's link field. By representing the **Empty** and **Full** states as zero, we get the state information stored for free when we set the link field (because it contains a four-byte-aligned pointer). Then, on a steal the link field is no longer necessary, and we may reuse the bottom two bits to hold state information.

```
! Future cell is located at %r10
! Future stack register is %l5

st   %l5,[%r10+4]                ! Store next field
mov  %r10,%l5                    ! Place this fc at top of stack
call _TreeAdd; add %o7,12,%o7    ! Call routine, modify return address
.word __TreeAdd.cs               ! Pointer to register unwinding code
ba   ___stolen; st %o0,[%fp+16]  ! Abnormal return
                                 ! (Future has been stolen)
                                 ! jump to steal & store ret val
.word __TreeAdd.cs               ! Pointer to register unwinding code
st   %o0,[%l5+16]                ! store return value
ld   [%l5+4],%l5                 ! Pop future cell stack
```

Figure 28: CM-5 Future call with integer return value

Given these optimizations, a `futurecall` on the CM-5 requires only five instructions (two stores, two integer instructions, and one load) in the case when the future is not stolen. Figure 28 contains sample code for a future on the CM-5.[5]

4.2.3 Compiling `touch`

The `touch` annotation generates inline code to synchronize on the state of the future cell. If the state is **Full**, then the touching thread continues; otherwise the library routine `Suspend` is called. `Suspend` stores the values of the callee-save registers, the frame pointer, and the program counter in the future cell, marks the cell as **Waiting**, and calls `MsgDispatch` to assign work to the processor. When the future call returns and finds the cell marked **Waiting**, it loads the registers, frame pointer, and program counter from the suspended cell, and resumes execution of the procedure. Figure 29 gives sample code for a touch of a future on the CM-5.

[5] According to the description of `call` on the SPARC [SPA92], the value of `o7` is computed before the delay instruction. We can therefore modify this value in the delay slot.

```
! BEGIN of TOUCH (future cell pointer in %o0)
ld [%o0+8],%l0      ! load future state
andcc %l0,3,%g0     ! check if full
be L.115
sethi %hi(__TreeAdd.cs),%o2
call ___Suspend; or %o2,%lo(__TreeAdd.cs),%o2
                    ! suspend-- pass callee-save sequence
L.115:              ! continue if full
! END of TOUCH
```

Figure 29: Touch future on CM-5

4.3 Stack Management

In the discussion thus far, we have glossed over certain details related to stack management. When a future is stolen, the portion of the stack between the frame belonging to the stolen continuation and the frame that migrated must be preserved for when the migrated thread returns. The stolen continuation may allocate new stack frames, which could overwrite the frames of the migrated thread if we used a simple stack management policy. To avoid this problem, we adopt a simplification of a single-stack technique for multiple environments by Bobrow and Wegbreit [BW73].

Bobrow and Wegbreit's method, called *spaghetti stacks*, splits a frame into a basic frame, which can be shared among several access modules, and an extension, which contains data local to an access module. Associated with each basic frame is a counter of the number of active access modules. Initially, a basic frame and extension are allocated contiguously at the end of the stack, with the counter set to one. The use of a coroutine, for example, would cause a copy of the frame extension to be made so the returns may go to different continuation points. On exit from an access module, the appropriate extension is deleted, and the basic frame is freed if no other extension references it (that is, the extension count is one). The end of stack pointer is then adjusted appropriately. When control returns to an extension, if active frames exist between the extension and the end of the stack, it is moved to the end to allow it to allocate new frames. Thus, the stack is split into various segments. Compaction is performed as necessary to prevent stack overflow, since the stack may have a tendency to be ever increasing.

Since our model does not require more than one extension per frame, we can simplify the method and reduce the overhead on function calls and returns. We maintain a single stack, called a *simplified spaghetti stack*, that contains the interleaved frames of the continuations.[6] In a simplified spaghetti stack, new stack frames are allocated off the global stack pointer. We maintain the invariant that *the global stack pointer always marks the end of a live frame.* The exit routine for a frame is split into two parts: deactivation and deallocation. If a procedure whose frame is in the middle of the stack exits, it is deactivated by marking it as garbage. If a frame at the end of the stack exits, the global stack pointer is adjusted, thus deactivating and deallocating the frame simultaneously. Additionally, the stack pointer is adjusted to the end of the live frame nearest the end of the stack, which deallocates any garbage frames immediately below it.

To implement deactivation, one word of memory in each frame, *splink*, is reserved for stack management information. If the frame is live, then a null pointer is stored at splink. Otherwise, splink contains a pointer to the beginning of the frame. By placing this word at the end of each frame, these pointers form linked lists of garbage frames. To maintain the invariant, after deallocating a live frame at the end of the stack, it is merely necessary to traverse the list until reaching a null pointer, and then set the global stack pointer to the location containing the null pointer.

Figure 30 provides an example of this process. Assume that initially the procedure F is executing (1). Notice that there are live frames between F and the end of the stack that belong to another thread of execution (d and e). When F calls G, a new frame is allocated at the end of the stack (2). When G exits, since the preceding frame (e) is live, the stack returns to the initial state (3). Figure 31 provides a more complex example. Again, F calls G (1,2). G migrates, and the run-time system gives control to e (3). The frame e is not at the end of the stack; hence, when it exits,

[6] A similar method, called a meshed stack, was developed by Hogen and Loogen [HL93] in the context of parallel implementations of functional or logic languages.

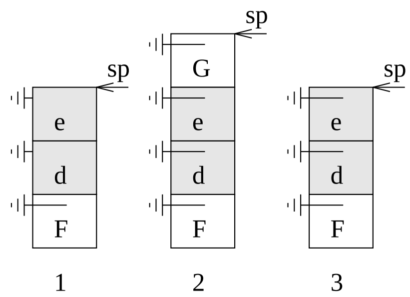

Figure 30: Simplified spaghetti stack — simple case.

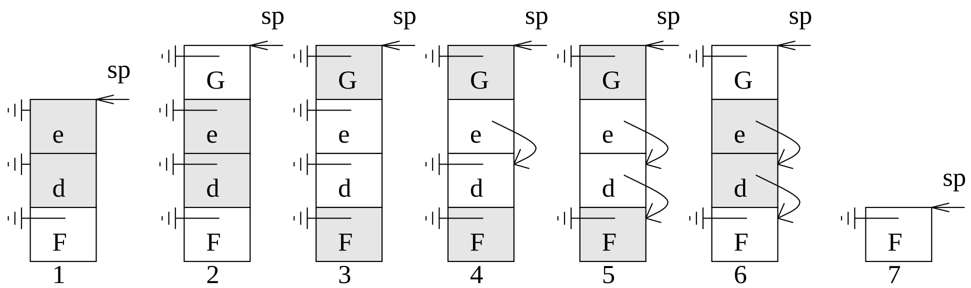

Figure 31: Simplified spaghetti stack — complex case.

a pointer is placed at the end of its frame marking it as garbage (4). The exit for d is similar (5). When G resumes execution and exits, its frame is deactivated (6). Since it is at the end of the stack, the list of garbage frames will be followed until a null pointer is reached (in the frame labeled F), thus deallocating all of the garbage frames (7).

The overhead resulting from our simplified spaghetti stacks is low. One extra instruction is required on entry to store zero at splink. The expected exit path (topmost frame, with live frame underneath), requires six additional instructions (three adds, one load, and two conditional branches) for the CM-5.

To reduce the overhead in the common case, we have implemented an additional optimization, called *express checkout*. We maintain a bit that indicates that the current stack frame is at the end of the stack with a live frame below it (the common case). When this bit is set, the cost of the stack exit sequence is only two instructions. Figure 32 shows the code related to the spaghetti stack, including the express-checkout

```
_TreeAdd:                               ! Beginning of procedure
add %sp,-176,%sp                        ! frame size 176, extension 72
st %fp,[%sp+172]
st %o7,[%sp+168]
add %sp,176,%fp
st %g0,[%sp+64]                         ! Store zero at splink

_END_TreeAdd:
andcc %l6,128,%g0                       ! Check express-checkout test
be _NoExpressEND_TreeAdd;               ! delay slot below
ld [%fp+-8],%l3                         ! Express-checkout code
ld [%fp+-4],%fp
jmpl %l3+12,%g0; add %sp,248,%sp

_NoExpressEND_TreeAdd:                  ! Slow checkout code
add %fp,-176,%l3
subcc %sp,%l3,%g0
bne _END_TreeAdd.2; ld [%fp+-8],%l4     ! Are we at stack bottom?
ld [%fp+136],%l1                        ! At stack bottom
subcc %l1,%g0,%g0
bne _END_TreeAdd.1; ld [%fp+-4],%fp     ! Is frame above live?
jmpl %l4+12,%g0; add %l3,248,%sp        ! If so, deallocate and exit
_END_TreeAdd.1:
add %l3,248,%sp
_END_TreeAdd.1.1:           ! Deallocate frames until finding one live
ld [%sp+64],%l3
subcc %l3,%g0,%g0
bne,a _END_TreeAdd.1.1; mov %l3,%sp
jmpl %l4+12,%g0; nop
_END_TreeAdd.2:             ! Not at bottom of stack
add %fp,72,%l1
st %l1,[%l3+64]             ! Deactivate frame
jmpl %l4+12,%g0; ld [%fp+-4],%fp
```

Figure 32: Spaghetti-stack code for CM-5

optimization. If the express-checkout optimization were disabled, the exit sequence would begin at `_NoExpressEND_TreeAdd`.

The express-checkout bit is set whenever a new thread is started, either by stealing a future or processing an incoming migrate on reference message. The bit is cleared when threads are combined (when processing migrate on return messages). Consequently, for a computation with reasonably large granularity, the express-checkout bit is set on the majority of procedure returns.

4.3.1 Discussion

There is one problem with the scheme that we have proposed. Since the simplified spaghetti stack scheme does not deallocate dead frames immediately, an Olden program could run out of stack space unnecessarily. Olden guarantees that at most two copies of any stack frame will be present in the system, one on the "home processor," and one on the current processor. We can only bound the size of a stack on a particular processor by the number of frames that could be live theoretically, even if, in an actual run, frames not at the end of the stack become dead sooner. We can achieve optimal stack use by performing compaction as necessary on entry to the run-time system. It suffices to check the stack explicitly in the run-time system, because dead frames that are not deallocated can only be generated by calls to the run-time system. A metric based on fragmentation of the stack or the remaining capacity of the stack can be used to determine when the stack needs to be compacted. The cost of such checks is small. In our experience, the additional stack growth is small, and in our implementation we do not check for stack overflow or perform compaction.

We chose spaghetti stacks as a compromise between multiple disjoint stacks and heap-allocated stacks. Our first implementation used multiple disjoint stacks, but this incurred excessive copying overhead and led to implementation difficulties. A heap-allocated approach might be the best design choice, since our stack usage patterns would allow a simple "deallocate-on-return" policy to be used. Our principal reason for not using heap allocation is historical; we started from a existing stack-based compiler and did not want to make more changes than necessary. Furthermore, the spaghetti stacks make it easy to use code compiled by Olden's compiler with existing system libraries.

4.4 Summary

In this chapter, we have outlined the major components of Olden: the run-time system and compiler. The run-time system handles communication and task management, using caching, migration and future stealing. The compiler manages pointer testing, futures, and multiple stacks by generating code inline. Using optimizations such as register unwinding and express checkout, we have implemented the Olden programming model with very little overhead. In Chapter 6, we will demonstrate the efficiency of this implementation by giving timing results for a suite of benchmarks.

Chapter 5

Selecting Between Mechanisms

The compiler must decide, for each pointer dereference, whether to use caching or migration for accessing remote data. Our goal is to minimize the total communication cost over the entire program. Consequently, although an individual thread migration is substantially more expensive than performing a single remote fetch (by a factor of about seven on the CM-5), it may still be desirable to pay the cost of the migration, if moving the thread will convert many subsequent references into local references. Consider a list of N elements, evenly divided among P processors (two possible configurations are given in Figure 33). First suppose the list items are distributed in a block fashion. A traversal of this list will require $N\frac{(P-1)}{P}$ remote accesses if software caching is used, but only $P-1$ migrations if the computation is allowed to follow the data (assuming $N >> P$). Hence, it is better to use computation migration for such

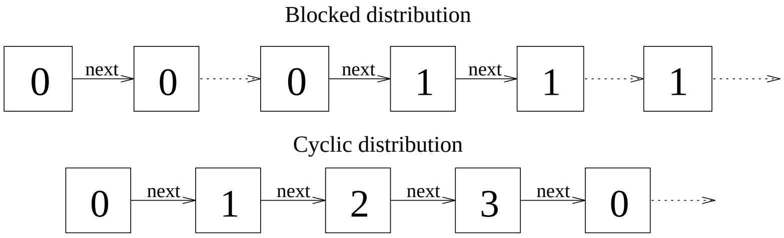

Figure 33: Two different list distributions. The numbers in the boxes are processor numbers. Dotted arrows represent a sequence of list items.

a data layout. Caching, however, performs better when the list items are distributed using a cyclic layout. In this case, using computation migration will require $N - 1$ migrations, whereas caching requires $N\frac{(P-1)}{P}$ remote accesses.

Olden uses a three-step process to select a mechanism for each program point. First, the programmer specifies *local path lengths*, which give hints to the compiler regarding the layout of the data. Second, a data flow analysis is used to find pointers that are traversing the data structure in a regular manner. In each loop (either iterative or recursive), at most one such variable is selected for computation migration. Finally, interactions between loops are considered, and additional variables are marked for caching, if it is determined that using computation migration for them may cause a bottleneck.

5.1 Using Local Path Lengths

Since the communication cost of a particular mechanism for a particular program fragment is highly dependent on the layout of the data, we allow the programmer to provide a quantified hint to the compiler regarding the layout of a recursive data structure. As previously discussed in Chapter 2, each pointer field of a structure may be marked with a local path length, which represents the expected number of adjacent nodes in the structure that are all on the same processor; if the programmer does not specify a local path length, a default value is supplied. To be more precise, a local path length of l associated with field, F, indicates that, on average, after traversing a pointer along field F that crosses a processor boundary, there will be a path of l adjacent nodes in the structure that reside on the same processor. Consequently, the local path length provides information regarding the relative benefit of using computation migration or software caching, as it provides information about how many future references will be made local by performing a migration. In Chapter 7, we discuss how to compute local path lengths in more detail, and present the Olden profiler, which will compute the local path lengths at run-time.

```
typedef struct tree {
    int val;
    struct tree *left {10};
    struct tree *right {3.33};
} tree;
```

Figure 34: Sample local-path-length hint

The compiler converts local path lengths into probabilities, called *path affinities*. Recall that a geometric distribution is obtained from an infinite sequence of independent coin flips [Ric87]. It takes one parameter, p, which is the probability of a success. The value of a geometric random variable is the number of trials up to and including the first success. If we consider traversing a local pointer to be a failure, then the path affinity is the probability of failure. The local path length is then the expected value of this geometric distribution; therefore, the path affinity is given by $1 - \frac{1}{local\ path\ length}$.

The intuition behind our heuristic's use of local path lengths and path affinities is illustrated by the examples given in Figure 33. In the blocked case, where computation migration is the preferred mechanism, the local path length is $\frac{N}{P}$ (there are N list items, distributed in a blocked fashion across P processors). The path affinity of the `next` field is then $1 - \frac{P}{N}$. In the cyclic case, where caching performs better, the `next` field has a local path length of one (each `next` pointer is to an object on a different processor), and a path affinity of zero. In general, computation migration is preferable when the local path lengths and path affinities are large, and software caching is preferable when the local path lengths and path affinities are small.

In the remainder of this chapter, we will describe how the compiler uses the path affinities computed from the programmer-specified local path lengths to select between computation migration and software caching. In our examples, we will refer to the structure declaration from Figure 34. For this declaration, the path affinity of the `left` field is $1 - \frac{1}{10}$ or 90%, and the path affinity of the `right` field is $1 - \frac{1}{3.33}$ or 70%.

```
struct tree *s, *t, *u;
while (s) {
  s = s->left;
  t = t->right->left;
  u = s->right;
}
```

Update Matrix (after \ before)	s	t	u
s	90		
t		63	
u	70		

Figure 35: A simple loop with induction variables

5.1.1 Discussion

In prior work [CR95], we had the programmer specify path affinities directly, rather than using local path lengths. While formalizing the definition of path affinities, we introduced local path lengths. Since we found it more natural to reason about local path lengths instead of probabilities, we chose to have the programmer specify a local path length, and then convert it to a path affinity.

Our notion of affinities differs from that used in COOL [CGH93]. In COOL, each method has affinities that are used by the runtime system for scheduling where the thread will be run. These affinities can specify collocation with an object or processor, or that certain methods should be run back-to-back. By contrast, our path affinities describe the probability that adjacent nodes along a path in the structure will reside on the same processor, and the information is used at compile time instead.

5.2 Update Matrices

We want to estimate how the program will, in general, traverse its recursively defined structures. To accomplish this, we examine the loops and recursive calls (hereafter referred to as *control loops*) checking how pointers are updated in each iteration. We say that `s` is updated by `t` along field `F` in a given loop, if the value of `s` at the end of an iteration is the value of `t` from the beginning of the iteration dereferenced through field `F` (that is, `s'=t->F`). This notion extends directly to paths of fields. Intuitively, variables that are updated by themselves in a control loop will

```
struct tree *TreeSearch(struct tree *t, int key)
{
  while(t && t->key != key) {
    if (t->key > key)
      t = t->left;
    else
      t = t->right;
  }
  return t;
}
```

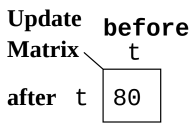

Figure 36: An example of updates with `if-then`

traverse the data structure in a regular fashion. We call such variables *induction variables*. This is similar to the notion of induction variables used in an optimizing compiler's analysis of for-loops [ASU88]. In both cases, an induction variable is a variable that is updated in a regular fashion in each iteration of the loop. In Figure 35, `s` and `t` are induction variables (since `s′=s->left` and `t′=t->right->left`), whereas `u` is not (since its value cannot be written as a path from its value in the previous iteration).

We summarize information on possible induction variables in an *update matrix*. The entry at location `(s,t)` of the matrix is the path affinity of the update, if `s` is updated by `t`, and is blank otherwise. In Figure 35, since `s` is updated by itself along the field `left`, the entry `(s,s)` in the update matrix is 90 (the affinity of the `left` field). Induction variables are then simply those pointers with entries along the diagonal (that is, they have been updated by themselves). In our example, the variables `s` and `t` have entries on the diagonal. We will consider only these for possible computation migration, as they traverse the structure in a regular manner.

The update matrices may be computed using standard data-flow methods. (Note again that exact or conservative information is not needed, as errors in the update matrices will not affect program correctness.) The only complications are that variables may have multiple updates or update paths of length greater than one. There are three cases. The first is a join point in the flow graph (for example, at the end of an `if-then` statement). Here we simply merge the two updates from each branch

```
int TreeAdd(struct tree *t)
{
  if (t == NULL) return 0;
  else
    return TreeAdd(t->left)+TreeAdd(t->right)+t->val;
}
```

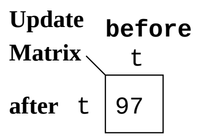

Figure 37: TreeAdd

by taking the average of their affinities. This corresponds to assuming each branch is taken about half of the time, and could be improved with better branch prediction information. If the update does not appear in both branches and we have no branch prediction information, rather than averaging the update, we omit it. We do this because we wish to consider only those updates that occur in every iteration of the loop, thus guaranteeing that the updated variable is actually traversing the structure. Having a matrix entry for a variable that is not updated might cause the compiler to incorrectly surmise that it could make a large number of otherwise remote references local by using migration for this variable. An example of the branch rule is given in Figure 36. The induction variable, `t`, has an update with path affinity 80, the average of the updates in the two branches (`t->right` 90%, `t->left` 70%).

Second, we must have a rule for multiple updates via recursion. Consider the simple recursive program in Figure 37. Note that `t` has two updates, one corresponding to each recursive call. The two recursive calls form a control loop. In this case, we define the path affinity of the update as the probability that either of the updates will be along a local path (since both are going to be executed). Because the path affinity of `left` is 90% and `right` is 70%, the probability that both are remote is 3% (assuming independence). Consequently, the path affinity of the update of `t` because of the recursive calls is 97%, the probability that at least one will be local. We will further motivate this choice in Section 5.3. Rather than combining the updates from the two branches of the `if-then-else` statement, we instead notice that the recursive calls occur within the `else` branch. This means we can predict that the `else` branch is almost always taken, and consequently, only the rule for recursion is used to compute the update of `t` in this control loop.

The final possibility is an update path of length greater than one (for example, `t=t->right->left`). The path affinity of this case is simply the product of the path affinities of each field along the path. An example of this is given in Figure 35. Here the path affinity of the update of `t` is the product of the path affinities of the `right` and `left` fields (90% * 70% = 63%).

So far, we have only discussed computing update matrices intraprocedurally. A full interprocedural implementation would need to be able to compute paths generated by the return values of functions, and handle control loops that span more than one procedure (for example, a mutual recursion). Our implementation performs only a limited amount of interprocedural analysis. In particular, we do not consider return values, or analyze loops that span multiple procedures. This limited interprocedural analysis is sufficient for all of our benchmarks; in the future, it may be possible to expand this analysis using techniques such as access path matrices [HHN95].

5.3 The Heuristic

Once the update matrices have been computed, the heuristic uses a two-pass process to select between computation migration and software caching. First, each control loop is considered in isolation. Then, in the second phase, we consider the interactions between nested control loops, and possibly decide to do additional caching. In addition to having the update matrix for each control loop, we also need information regarding whether or not the loop may be parallelized. In Olden, the compiler checks for the presence of programmer-inserted futures to determine when a control loop may be parallelized.

In the first pass, for each control loop, we select the induction variable whose update has the strongest path affinity. If a control loop has no induction variable, then it will select computation migration for the same variable as its parent (the smallest control loop enclosing this one). If the path affinity of the selected variable's

```
Traverse(tree *t) {
  if (t==NULL) return;
  else {
    Traverse(t->left);
    Traverse(t->right);
  }
}

WalkAndTraverse(list *l, tree *t) {
  for each body, b, in l do in parallel {
    Traverse(t);
  }
}
```

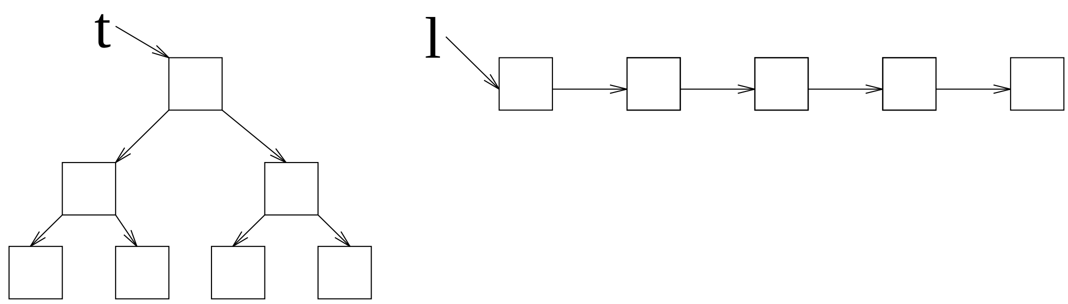

Figure 38: Example with a bottleneck.

update exceeds a certain threshold, or the control loop is parallelizable, then computation migration is chosen for this variable; otherwise, dereferences of this variable are cached. Dereferences of all other pointer variables are cached. We select computation migration for parallelizable loops with path affinities below the threshold because this mechanism allows us to generate new threads. (As described in Chapter 3, in Olden, new threads are only generated following a migration).

Considering control loops in isolation does not yield the best performance. Inside a parallel loop, it is possible to create a bottleneck by using computation migration. Consider the code fragments in Figures 38 and 39. `WalkAndTraverse` is a procedure that for each list item traverses the tree, and is derived from a portion of the Barnes-Hut benchmark, which will be discussed in Chapter 6. If computation migration

```
Walk(list *l) {
  while (l) {
    visit(l);
    l = l->next;
  }
}

TraverseAndWalk(tree *t) {
  if (t==NULL) return;
  else {
    do in parallel {
      TraverseAndWalk(t->left);
      TraverseAndWalk(t->right);
    }
  Walk(t->list);
  }
}
```

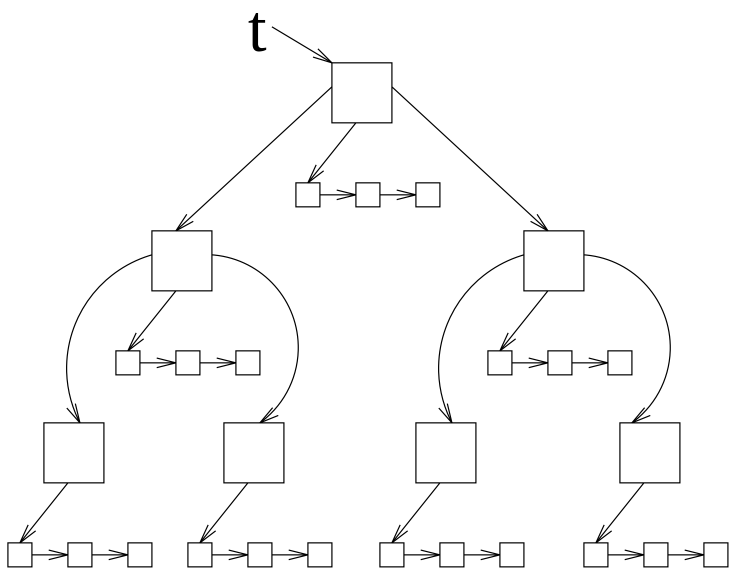

Figure 39: Example without a bottleneck.

were chosen for the tree traversal, the parallel threads for each item in the list would be forced to serialize on their accesses to the root of the tree, which becomes a bottleneck. In `TraverseAndWalk`, for each node in the tree, we walk the list stored at that node. Since there is a different list at each node of the tree, the parallel threads at different tree nodes are not forced to serialize, and there is no bottleneck. In general, a bottleneck occurs whenever the initial value of a variable selected for migration in an inner loop is the same over a large number of iterations of the outer loop. Returning to the examples, in `WalkAndTraverse`, `t` has the same value for each iteration of the parallel `for` loop, while in `TraverseAndWalk`, we assume `t->list` has a different value in each iteration (that is, at each node in the tree). Although in general this is a difficult aliasing problem, we do not need exact or conservative information. If incorrect information is used, the program will run correctly, but possibly more slowly than if more precise information were available. Our current approximation tests to see if the induction variable for the inner loop is updated in the parent loop. If so, we assume no bottleneck will occur; otherwise, we use caching in the inner loop to avoid the possibility of a bottleneck. Once the heuristic has analyzed the interactions between loops, the selection process is complete.

5.3.1 Threshold and Default Path Affinities

The threshold has been set to 86% for the CM-5 implementation. Since the cost of a migration is about seven times that of a cache miss on the CM-5, the break-even local path length is seven, which corresponds to a path affinity of 86%. For other platforms, the threshold would be $1 - \frac{1}{r}$, where r is the ratio of the cost of a migration to the cost of a cache miss. On platforms where latency is greater, such as networks of workstations, r (and consequently the threshold) will be smaller, and on platforms where latency is smaller, such as the Cray T3D [Cra93], r and the threshold will be larger.

We set the default path affinity to 70% (this corresponds to a default local path length of 3.33). This value was chosen so that, by default, list traversals will use caching, tree traversals will use computation migration, and tree searches will use caching. The averaging method for recursive calls was also designed to obtain this behavior. To see why this is desirable, recall the four processor tree allocation shown in Figure 3. Generally, as in this example, we expect large subtrees to be distributed evenly among the processors. In such a case, searches of the tree (such as the code in Figure 36), will traverse a relatively short path that may cross a processor boundary several times. A complete tree traversal (such as the code in Figure 37), however, will perform large local computations. Consequently, it is preferable to use migration for the traversal, and caching for the search. List traversals may be viewed as searches through a degenerate tree.

In our experience, these defaults provide the best performance most of the time. In those cases where the defaults are not appropriate, the programmer can specify local-path-length hints. (We do not allow the programmer to modify the threshold, but the same effect can be obtained by modifying the local path lengths.) We explicitly specified local path lengths in three of the eleven benchmarks (TSP, Perimeter, and MST) since the default affinity did not reflect the layout of the data structure, but in only one case (TSP) did it have a significant effect on performance. We will examine TSP more closely in Section 6.2.

Returning to the declaration in Figure 34 (page 57), we see why providing more precise local path length information is often unnecessary, and had no effect for Perimeter and MST. Using default local path lengths rather than those from Figure 34 changes the entry for `t` in the `TreeAdd` matrix from 97 to 91, and the entry for `t` in the `TreeSearch` matrix from 80 to 70. Since these changes do not cross the threshold, the heuristic will make the same selections. Our experience demonstrates that there is broad latitude for imprecision in the local-path-length information.

5.3.2 Example

We now return to the `TreeMultAdd` example from Figure 17 in Chapter 3. If we use the default local-path-length values, both the `left` and `right` fields have local path length 3.33. The path affinities of these fields are $1 - \frac{1}{3.33}$, or 70%.

Update Matrix		before	
		t	u
after	t	91	
	u		91

Figure 40: Update matrix for TreeMultAdd

There is one control loop in this program, consisting of the two recursive calls. Both `t` and `u` have two updates from the recursive calls. The path affinities of both of these updates are 70% (as both the `left` and `right` fields have path affinity 70%). As shown in Figure 40, using the rule for combining multiple updates via recursion, the entries for `t` and `u` in the update matrix will both be $1-(1-.7)(1-.7)$, or 91%. Since `t` and `u` have entries on the diagonal of the matrix, they are both induction variables.

When the heuristic examines this loop, it will select the induction variable whose update has the largest affinity for migration. Since the updates of both variables have the same affinity, one is arbitrarily chosen (we assume `t`). References to `u` are then cached. Since there is only one loop, the bottleneck analysis will not be performed.

5.4 Summary

Olden provides two mechanisms, computation migration and software caching, for accessing remote data. Based on local-path-length information, the compiler automatically selects the appropriate combination of these two mechanisms to minimize communication. The local path lengths may be obtained from the programmer, the Olden profiler (which will be presented in Chapter 7), or compiler defaults. The selection process, as shown in Figure 41, consists of four parts: converting local path lengths to probabilities, called *path affinities*; computing update matrices for each loop and recursion; analyzing loops in isolation; and using bottleneck analysis to analyze interactions between nested loops. We demonstrate the effectiveness of the heuristic on a suite of eleven benchmarks in Chapter 6.

Input. Olden program with some local-path-length hints.

Output. Program with each dereference marked for computation migration or software caching.

Method.

```
comment Compute path affinities
foreach structure declaration, D, do
   foreach pointer field, F, do
      if field F of structure D has no programmer-specified
         local path length then
             mark field F of structure D with default local path length
      Convert local path length to path affinity

comment Compute update matrices
foreach loop, L, do
   Compute an update matrix, M(L), for L

comment Single-loop analysis
foreach loop, L, do
   x ← induction variable with largest path affinity from M(L)
   foreach dereference, R, in L do
      if R dereferences x then
         if affinity(M(L),x) > threshold then
            mark R for computation migration
         else
            mark R for software caching
      else
         mark R for software caching

comment Bottleneck analysis
foreach parallel loop, P, do
   foreach loop, L, inside P, do
      x ← variable using computation migration in L
      if x is not updated in P then
         mark dereferences of x in L for software caching
```

Figure 41: Selecting between computation migration and software caching

Chapter 6

Experimental Results

In the previous chapters, we have described our prototype implementation of Olden for the Thinking Machines CM-5. In this chapter, we report results for eleven benchmarks. Each of these benchmarks is a C program annotated with futures, touches, calls to Olden's allocation routine, and, in some cases, data structure local path length information.

We performed our experiments on CM-5s at two National Supercomputing Centers: NPAC at Syracuse University and NCSA at the University of Illinois. The timings reported are averages over three runs done in dedicated mode. For each benchmark, we present two one-processor versions. The *sequential* version was compiled using our compiler, but without the overhead of futures, pointer testing, or the spaghetti stack. The one-processor Olden implementation (*one*) includes these overheads.

Table 2 briefly describes each benchmark. Before we discuss our overall results, we examine the three coherence mechanisms described in Section 3.1.2 and select the best for our benchmark suite. Then, in Section 6.2, we give overall results for the heuristic using this coherence mechanism.

Table 2: Benchmark Descriptions

Benchmarks	Description	Problem Size
TreeAdd	Adds the values in a tree	1024K nodes
MST	Computes the minimum spanning tree of a graph[Ben80]	1K nodes
Power	Solves the Power System Optimization problem [LML*93a]	10,000 customers
TSP	Computes an estimate of the best hamiltonian circuit for the Traveling-salesman problem[Kar77]	32K cities
Barnes-Hut	Solves the N-body problem using hierarchical methods [BH86]	8K bodies
Bisort	Sorts by creating two disjoint bitonic sequences and then merging them[BN89]	128K integers
EM3D	Simulates the propagation of electro-magnetic waves in a 3D object[CDG*93]	40K nodes
Health	Simulates the Colombian health-care system [LCUW88]	1365 villages
Perimeter	Computes the perimeter of a set of quad-tree encoded raster images [Sam81]	4K x 4K image
Union	Computes the union of two quad-tree encoded raster images [Shn81]	70,000 leaves
Voronoi	Computes the Voronoi Diagram of a set of points[GS85]	64K points

6.1 Analysis of Coherence Mechanisms

For our comparison of the three coherence schemes, we use six benchmarks for which the compiler chose to use both migration and caching (although the compiler chose to use both migration and caching for Union, the data layout causes the cache not to be used). Table 3 gives statistics on writes, reads, and the behavior of the three coherence schemes. For most of our benchmarks, the three schemes behave basically identically, as almost all shared data is written between migrations. Barnes-Hut has a slightly smaller number of misses for the global and bilateral schemes. Bitonic Sort and Perimeter both have a slightly smaller number of misses only for the global scheme. The percentage of misses in Health decreases dramatically by adding global knowledge, but since the total number of misses is small the local-knowledge scheme has the lowest running time. Overall, the number of unnecessary misses generated by the overzealous invalidation of the local scheme is quite small (less than two percent of the total number of cacheable references). Since the overhead of tracking writes is substantial (seven instructions for non-shared pages, and twenty-three instructions for shared pages),[1] the local-knowledge scheme has the best running times for our benchmark suite. These times range from 0 (EM3D, Perimeter, Union) to 72% (Health) faster than the global-knowledge scheme. The bilateral scheme had running times between those of the local and global schemes.

Although it may seem surprising that Olden programs run faster with such a coarse coherence scheme, it follows logically from the way Olden uses data. To get good performance, the programmer must lay out the data so that related data is stored on the same processor. Also, since threads are guaranteed not to interfere, we are able to postpone the invalidations significantly. Given these considerations, and the fact that the benchmarks do not have much long-term read-only data, it is not surprising that the local-knowledge scheme outperforms the others.

6.2 Heuristic Results

Table 5 lists the running time of a sequential implementation plus speedup numbers for up to 32 processors for each benchmark using the local-knowledge coherence scheme. We report whole program times (W) for three benchmarks, Power, Barnes-Hut, and Health, to allow for comparison with published results. We report kernel times only for the rest to avoid having their data structure building phases, which show excellent speedup, skew the results. We use a true sequential implementation compiled with our compiler for computing speedups. These sequential times are comparable to those using gcc with optimization turned off. Using an optimization level of two, the gcc code ranges from one (em3d, health and mst) to five (TSP) times faster. The gcc optimizations can be traced to better register allocation and handling of floating point arguments, both of which could be implemented in lcc/Olden by adding an optimization phase to the compiler.

[1] Zekauskas et al. [ZSB94] track writes, using nine MIPS instructions, by storing a code template in each shared memory region that sets the appropriate dirty bit. Even with this improvement, the extra overhead would still exceed the caching performance gain for our benchmarks.

Table 3: Caching Statistics on 32 Processors

Benchmarks	Cacheable Writes # (1000s)	Cacheable Writes % Remote	Cacheable Reads # (1000s)	Cacheable Reads % Remote	% of Remote references that miss: Local	Global	Bilateral	Total Pages Cached	% of Time waiting for misses (Local)
Barnes-Hut	2,707	18.3	73,601	55.6	0.815	0.563	0.792	21,749	1.7
Bisort	8,208	0.045	32,617	0.054	28.6	24.9	29.2	1,604	0.3
EM3D	0	0	839	19.4	6.18	6.18	6.18	1,995	25.6
Health	8,861	0.063	33,405	0.019	87.0	10.3	87.0	163	0.7
Perimeter	0	0	1,018	2.02	8.80	8.63	8.80	502	2.6
Union	0	0	567	0	0.0	0.0	0.0	0	0.0
Voronoi	9,825	1.57	42,359	1.26	5.89	5.89	5.89	2,982	1.4

Table 4: Running Times of Coherence Mechanisms (32 Processors)

Benchmarks	Running Time (sec.) Local	Global	Bilateral
Barnes-Hut	49.62	66.34	63.41
Bisort	4.96	9.93	9.79
EM3D	0.101	0.101	0.101
Health	1.94	7.00	4.93
Perimeter	0.175	0.175	0.175
Union	0.129	0.129	0.129
Voronoi	5.677	6.806	6.39

Table 5: Results

Benchmarks	Heuristic choice	Sequential time (sec.)	Speedup by number of processors: 1	2	4	8	16	32	Migrate-only speedup (32)
TreeAdd	M	4.49	0.73	1.47	2.93	5.90	11.81	23.4	
MST	M	9.81	0.96	1.36	2.20	3.43	4.56	5.14	
Power^W	M	286.59	0.96	1.94	3.81	6.92	14.85	27.5	
TSP	M	43.35	0.95	1.92	3.70	6.70	10.08	15.8	
Barnes-Hut^W	M+C	555.79	0.74	1.42	3.00	5.29	8.13	11.2	<0.01
Bisort	M+C	31.41	0.73	1.35	2.29	3.52	4.92	6.33	6.13
EM3D	M+C	1.21	0.86	1.51	2.69	4.48	6.72	12.0	0.05
Health^W	M+C	42.09	0.95	1.95	3.89	7.61	14.72	21.70	14.49
Perimeter	M+C	2.47	0.86	1.70	3.37	6.09	9.86	14.1	2.96
Union	M+C	1.46	0.76	1.48	2.88	5.11	7.97	11.30	11.30
Voronoi	M+C	49.73	0.75	1.38	2.41	4.23	6.88	8.76	0.47

^{W} – Whole program times

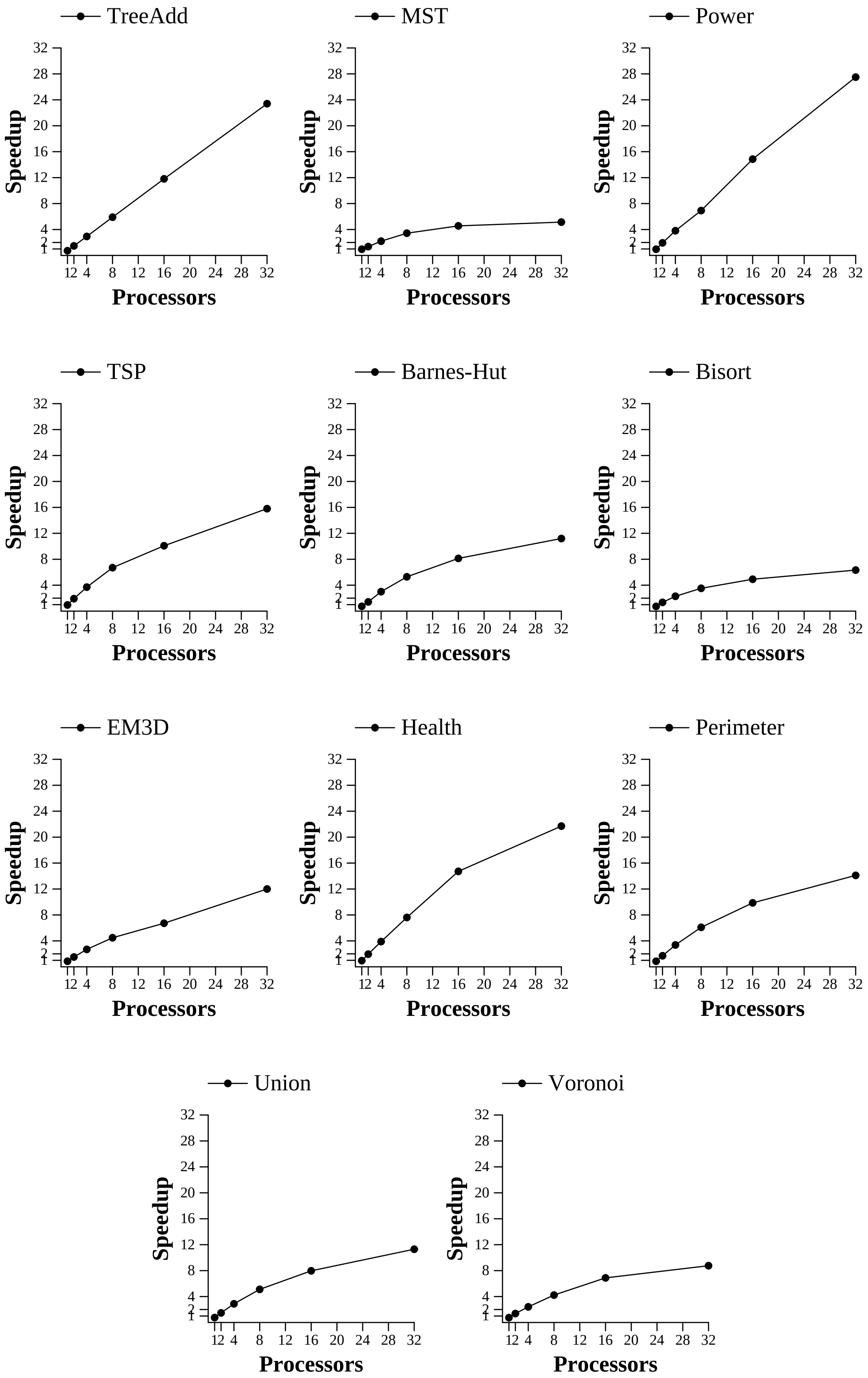
TreeAdd
MST
Power
TSP
Barnes-Hut
Bisort
EM3D
Health
Perimeter
Union
Voronoi
Speedup
Processors
1 2 4 8 12 16 20 24 28 32

Table 6: 32-Processor Overhead (Normalized)

Benchmark	Communication overhead (sec.)	Cache lookup overhead (sec.)	Pointer test overhead (sec.)	future/stack overhead (sec.)	"Available speedup"	Observed speedup
TreeAdd	0.01	0.00	0.08	0.16	32	23.4
MST	0.75	0.00	0.01	0.03	25	5.14
Power	0.03	0.00	<0.01	0.04	30	27.5
TSP	0.01	0.00	0.03	0.01	17	15.8
Barnes-Hut	0.08	<0.01	<0.01	0.04	13	11.2
Bisort	0.20	<0.01	0.09	0.04	9	6.33
EM3D	0.40	<0.01	0.10	0.10	24	12
Health	0.24	<0.01	0.03	0.01	30	21.7
Perimeter	0.16	<0.01	0.05	0.05	19	14.1
Union	0.08	0.00	0.08	0.08	16	11.3
Voronoi	0.17	0.03	0.10	0.04	13	8.76

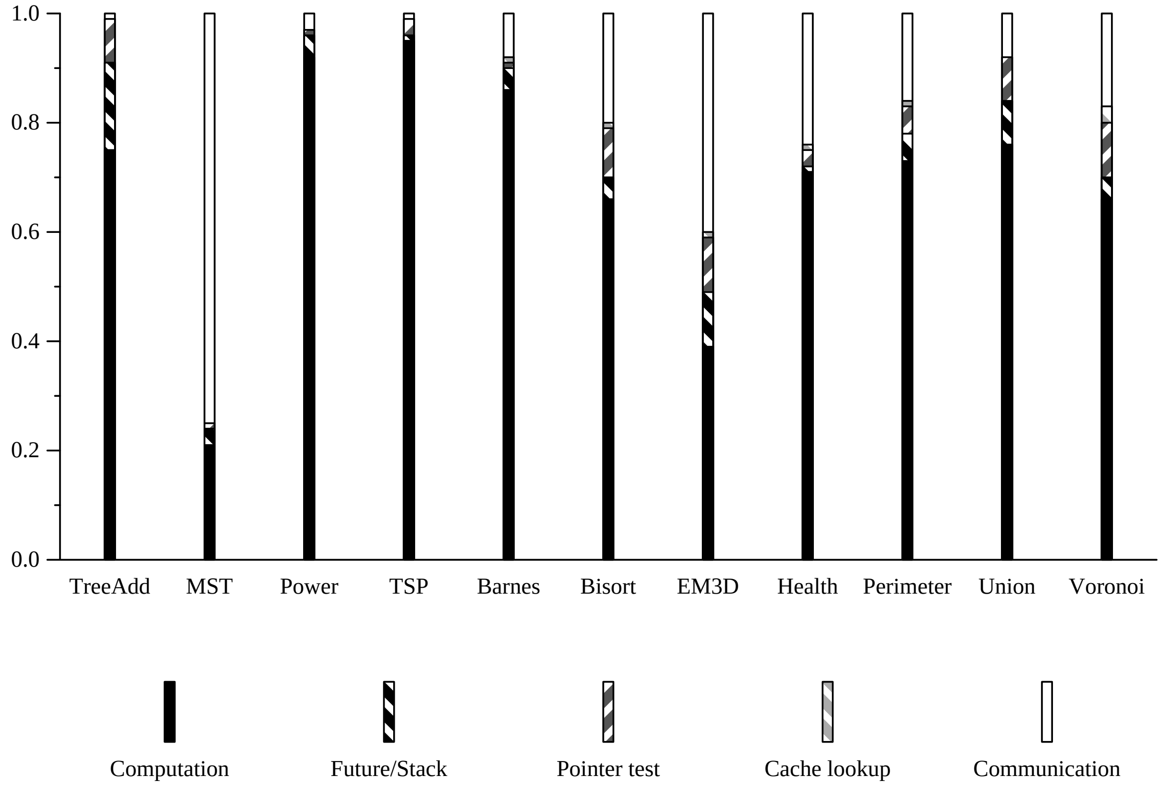

Figure 42: 32-Processor time breakdown

The speedup numbers for the one-processor version give a partial measure of Olden's overhead. Communication cost, and the cost of maintaining the cache table, however, do not appear in the one-processor version. Table 6 gives a breakdown of Olden's overhead on 32-processor runs, where each benchmark's running time has been normalized to one second. We give a bar-graph representation of this data in Figure 42. We divide the overhead into four categories: communication overhead, cache-lookup overhead, pointer-test overhead, and future/stack overhead. These numbers were computed by taking the maximum over all processors of the number of events on that processor multiplied by an experimentally obtained estimate of their costs. Since we do not do a critical path analysis, the numbers give a lower bound for the actual overhead. Communication overhead measures the amount of time spent sending and receiving messages. Cache-lookup overhead measures the amount of time spent determining if a remote reference is present in the cache, as described in Section 4.2. Pointer-test overhead measures the time spent determining if a pointer dereference is local or remote. Future/stack overhead is the overhead of having multiple threads, and measures the amount of time spent managing futures and the multiple stacks. The pointer-test overhead and future/stack overhead are also described in more detail in Section 4.2. We also provide a speedup calculation after subtracting out the measured overheads. This gives a rough estimate of the available parallelism in the algorithm as implemented in Olden.

6.2.1 TreeAdd

TreeAdd is a very simple program that recursively walks a tree, summing each subtree. We report results for a balanced, binary tree of size 1024K nodes. The tree is allocated using the allocation scheme presented in Figure 2. The subtrees at a fixed depth in the tree are equally distributed. Since this is a simple tree traversal, the Olden compiler chose to use migration alone to satisfy remote references. The Olden implementation of `TreeAdd` has an efficiency of 73%. The efficiency is not perfect as there is very little real work over which to amortize the overhead of the futures, pointer tests, and the spaghetti stack. If the overhead from futures, pointer tests, and the spaghetti stack were zero, this benchmark would display near-perfect speedup, because only $P-1$, where P is the number of processors, migrations occur.

6.2.2 Minimum Spanning Tree

MST computes the minimum spanning tree of a graph using Bentley's algorithm [Ben80]. The vertices of the graph are divided evenly among the processors, and at each step, each processor nominates one vertex to be added to the spanning tree. The closest vertex is added to the tree. This process repeats until all vertices have been added to the tree.

The performance for MST is poor and degrades sharply as the number of processors increases, because the number of migrations is $O(NP)$, where N is the number of vertices, and P the number of processors. Caching would not reduce communication costs for this program, because these migrations serve mostly as a mechanism for synchronization. As can be seen in Table 6, the communication overhead dominates, and we would obtain a speedup of 25 on 32 processors if this overhead were zero. In the future, we intend to explore ways to reduce communication overhead in these situations.

6.2.3 Power Pricing

Power solves the *Power-System-Optimization* problem, which can be stated as follows: given a power network represented by a tree with the power plant at the root and the customers at the leaves, use local information to determine the prices that will optimize the benefit to the community [LML*93b]. The computation executes a series of phases until convergence is reached. Each phase has a downward pass that propagates pricing information from the root to the customers and an upward pass that propagates demand information from the customers to the root. As with TreeAdd, the heuristic chose to use computation migration for this tree traversal. As shown in Figure 43, we use the input configuration from Lumetta et al. [LML*93b]: one substation; 10 main feeders from the root; each feeder branches to 20 lateral nodes;

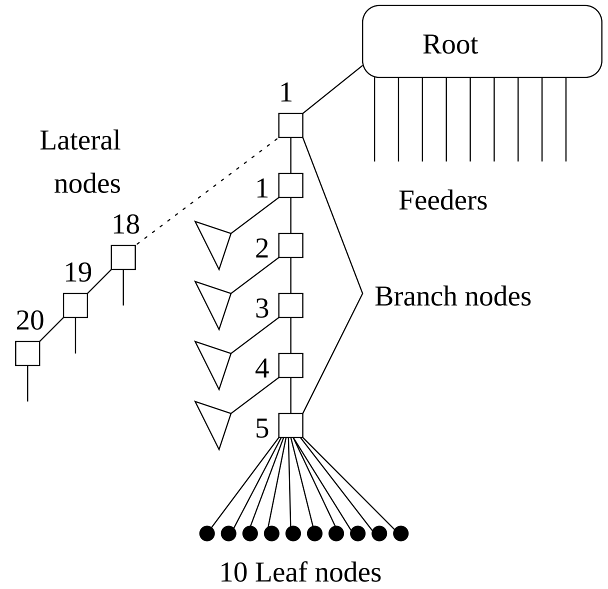

Figure 43: Power-pricing tree

each lateral node is the head of a line of five branch nodes; and each branch has 10 leaves. In all there are 1,201 internal nodes and 10,000 customers. The layout of the tree is determined in the following way: the branch nodes are divided evenly among the processors using a block distribution based on an in-order numbering of the tree. The leaves are distributed to the same processor as the corresponding branch; laterals are placed on the same processor as the head of their line of branches; and the root is placed on Processor 0.

We report speedups for the whole program. Power's behavior is very similar to TreeAdd's, but on a much smaller tree. The main difference between them is that, in Power, a substantial amount of work is done at the leaves of the tree. On 64 processors, our efficiency is about 80%, compared to 75% by Lumetta et al.[2] Also, Olden allows us to express the computation in a more natural recursive manner, rather than iteratively with locks.

[2]The speedups we report exceed those in prior work [RCRH95] due to an improved implementation of migration

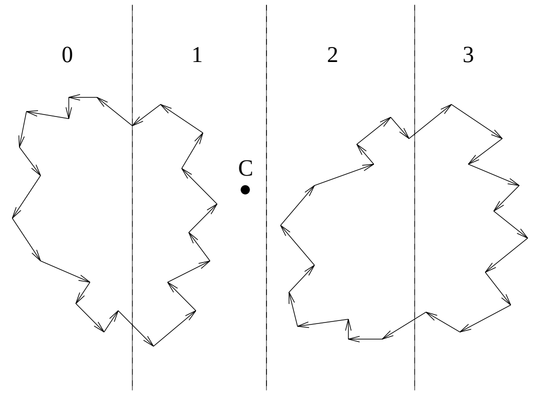

Figure 44: Merge phase of TSP

6.2.4 Traveling-Salesman Problem

TSP computes an estimate of the shortest hamiltonian circuit in a planar graph. The program is based on an implementation of Karp's [Kar77] partitioning algorithm by Müller [Mül93]. In our implementation, however, we assume the cities are given in a tree that alternately divides a set of points by X or Y coordinate at each level. The subtrees at a fixed depth are divided equally among the processors. Rather than computing an exact solution for small sizes (fewer than 150 cities), following Müller, we use the closest-point heuristic [CLR89].

Although TSP also uses a divide-and-conquer algorithm on a tree, unlike TreeAdd and Power, its merge phase is non-trivial. The merge phase of TSP combines two cycles (the subresults from the recursive calls on the subtrees) and the parent of these two subtrees into a single cycle. An example merge on four processors is given in Figure 44. The numbers indicate the layout of the cities across the processors. The cities are sorted by x-coordinate, and then divided evenly among the processors. Each merge is sequential and walks through the left subresult followed by the right subresult, which requires a migration for each participating processor. Even after subtracting out the measured overhead, the predicted speedup is only 17, as there are fewer nodes than processors in the top four levels of the tree (out of 15).

TSP is one case where allowing the programmer to specify local-path-length information yields a marked increase in performance. Using a specified local path length

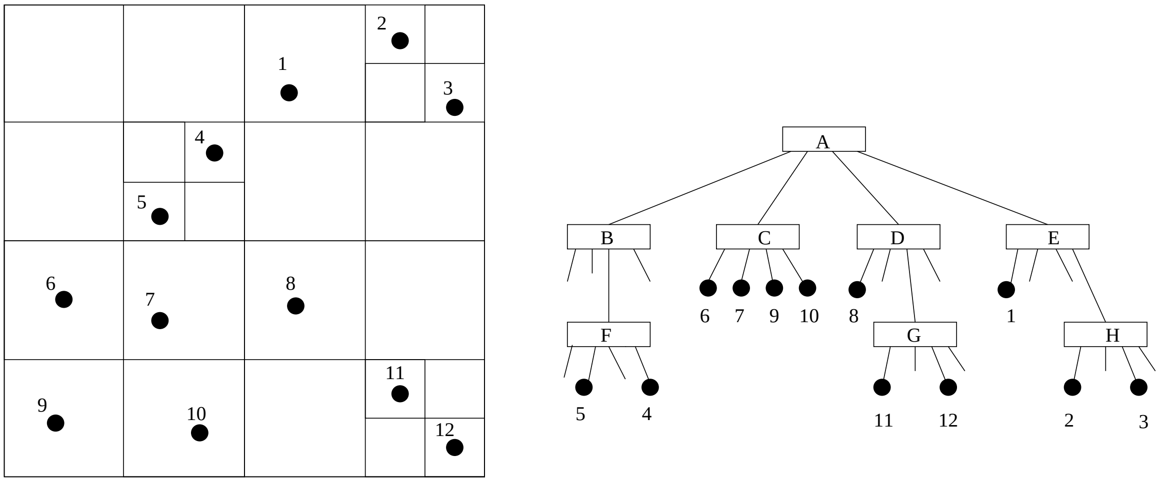

Figure 45: Barnes-Hut tree

yields a speedup of 15.8 on 32 processors, as opposed to 6.4 with the default local path length. We will discuss this benchmark in greater detail in Chapter 7.

6.2.5 Barnes-Hut

The Barnes-Hut[3] benchmark [BH86] simulates the motion of particles in space using an $O(n \log n)$ algorithm for computing the accelerations of the particles. This is a classic n-body problem. To speed the simulation, the particles are stored in an oct-tree, where each internal node contains summary data for all the particles in its subtree and each external node represents a particle. An oct-tree is simply the three-dimensional generalization of a quad-tree. An example quad-tree (which would be used if all the bodies were in the same plane) is given in Figure 45. Each particle computes the gravitational forces from other particles by traversing this tree, summarizing the force from groups of particles that are sufficiently far away. In our example, particle 9 might summarize the contributions of 1, 2, and 3, using only the information stored at node E.

[3]These results differ greatly from those given in [RCRH95]. Our prior implementation created a copy of the tree on each processor, whereas this implementation distributes the data structure. With a distributed data structure, we are able to run simulations on much larger systems.

We report the cost of generating the points using a uniform distribution and computing the new acceleration and position for each particle for ten time steps. This computation is broken into three pieces: building the tree used to represent the particles, calculating new accelerations for the particles by walking the tree, and then computing the new positions of the particles. The heuristic chose a combination of migration and caching for the first two phases: migration to distribute computation among the processors; caching to bring "distant" tree nodes to the computation as needed. Software caching is chosen for the tree even though it has high locality to avoid causing a bottleneck at the root, as discussed in Chapter 5. Migration alone suffices for computing the new positions of the particles, as this is done independently for each. Falsafi et al. [FLR*94] give results for six different implementations of this benchmark; our results using their parameters (approximately 36 secs/iter) fall near the middle of their range (from 15 to 80 secs/iter). In our implementation, however, the tree building phase is sequential and starts to represent a substantial fraction of the computation as the number of processors increases. Factoring out this cost, we predict a speedup of over 19 on 32 processors. If we omit the building phase and subtract out overhead, the predicted speedup for the remaining two phases is approximately 27 on 32 processors. To improve the performance of this and other programs, we plan to investigate ways to parallelize building incremental data structures in Olden.

6.2.6 Bitonic Sort

Bitonic Sort [BN89] allocates a random set of integers and performs two sorts on them: one forward and one backward. A binary tree is used to store the numbers. First, the algorithm creates bitonic sequences in both subtrees, and then merges them to generate the sorted result. The merge phase swaps subtrees to create two disjoint bitonic sequences, and then performs two recursive calls on these sequences. We have modified Bilardi and Nicolau's implementation to remove an unnecessary malloc on each recursive call. The data have been placed on the processors such that the subtrees at a fixed depth are evenly distributed over the available processors, and the

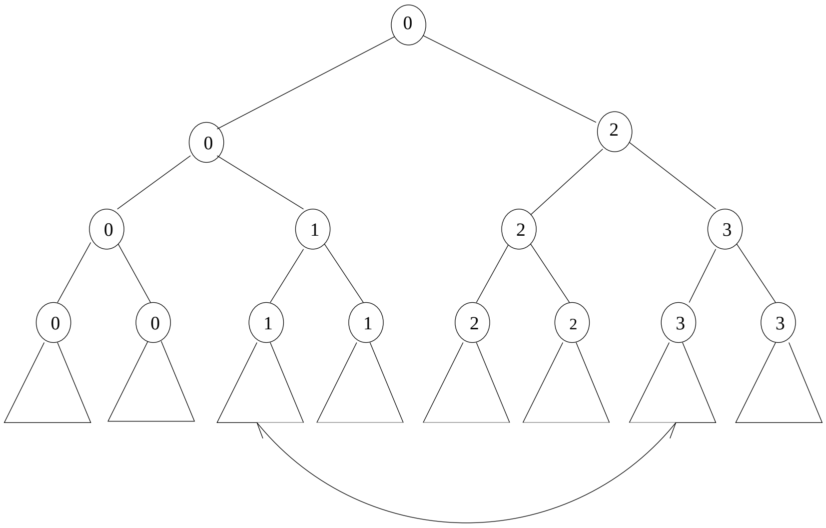

Figure 46: Bitonic Sort

benchmark has been modified to maintain this layout following the sort. Swapping the trees rather than pointers to the trees is expensive, but, as seen in Figure 46, helps maintain locality, which is crucial to the performance of the second sort and subsequent uses of the data.

A pair of pointers is used to search the subtrees during the merge phase. The migration heuristic is designed to use software caching for tree searches, so dereferences to these pointers use caching. The tree swaps, on the other hand, use migration, because a large amount of the data is touched on each processor between migrations. Again there is a large amount of communication overhead, but we also observe a loss of parallelism from performing the tree swaps to maintain locality.

6.2.7 EM3D

EM3D models the propagation of electromagnetic waves through objects in three dimensions [CDG*93]. This problem is cast into a computation on an irregular bipartite graph containing nodes representing electric and magnetic field values (`E` nodes and `H` nodes, respectively). At each time step, new values for the `E` nodes are computed from a weighted sum of the neighboring `H` nodes, and then the same is done for the

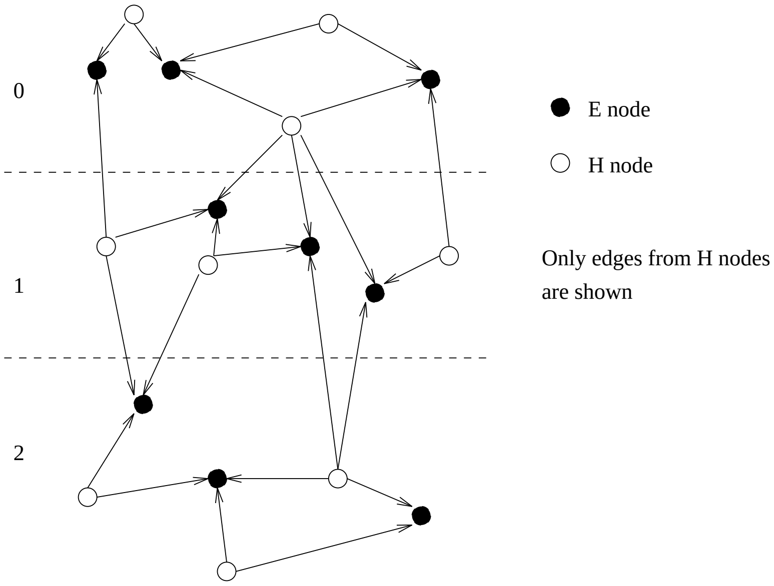

Figure 47: An EM3D graph on three processors

`H` nodes. Figure 47 gives a small example graph. The main computation loop consists of each processor walking down its list of nodes, reading the values from the neighbors and using them to update the current node. The heuristic chose to use caching to read the values of neighbors on remote processors. We report speedups on a graph with 40,000 nodes, each having degree 20, with 80% of the edges being local. For a 64 processor implementation with 320,000 nodes, our implementation performs comparably to the ghost node implementation of Culler et al. [CDG*93], yet does not require substantial modification to the sequential code.

6.2.8 Health

Health simulates the Colombian health-care system [LCUW88] using a four-way tree of the form shown in Figure 48. Health care is provided according to a multi-tiered system of services and referrals. At each level in the system, certain services are provided while other problems are referred to the next higher level. (It is assumed that all patients can be treated by the health center at the root of the tree). Each node of the tree represents a hospital, and at each node there is a list, where each list node

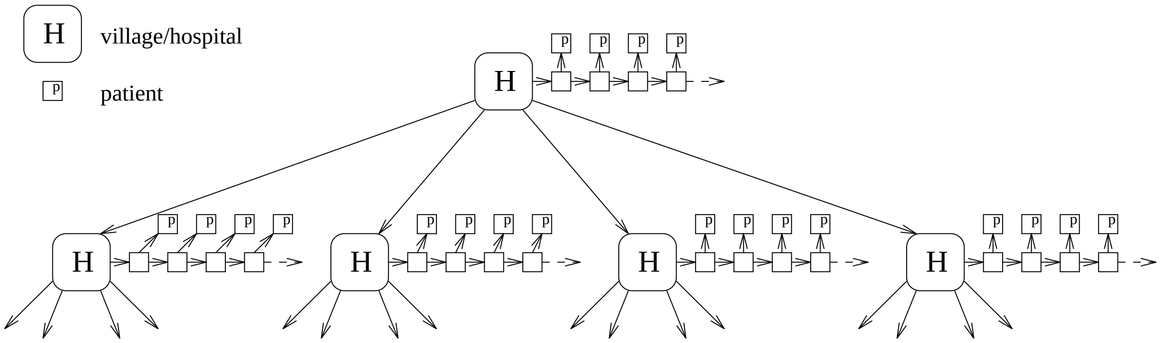

Figure 48: Colombian health-care system

contains a pointer to a patient. At each timestep, the tree is traversed, and patients, once assessed, are either treated or passed up the tree to the parent. Also, during each timestep, each hospital receives patients from its associated village according to a negative exponential distribution. The heuristic, according to its design, chose migration for the tree traversal, and caching to access patients from remote processors.

Although Health uses the same synchronization as MST, we obtain better speedup, as there is more work done in each iteration.[4] Again, subtracting the overhead yields a prediction of nearly perfect speedup for this benchmark. Note the linked list at each hospital is almost entirely local, but may have pointers to patients from remote processors. We will examine the performance impact of this layout in more detail in Chapter 7.

6.2.9 Perimeter

Perimeter computes the perimeter of a quad-tree encoded raster image [Sam81]. A small raster image and its encoding are given in Figure 49. The algorithm superficially looks similar to TreeAdd, but traverses the tree in a very different way when computing the contribution of a particular leaf. For example, to compute the contribution of X, we must determine which of its neighbors are white. This requires

[4]We report better speedups than in previous work [CR95] as we have corrected a bug that caused each hospital to only have one doctor.

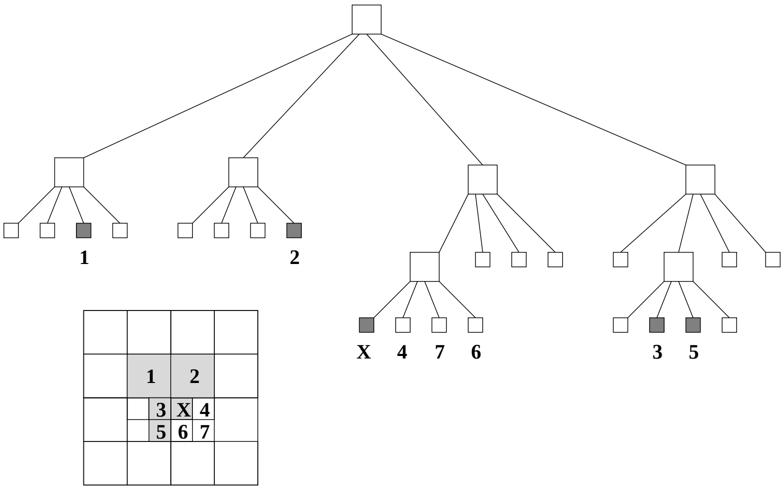

Figure 49: A quad-tree encoded raster image

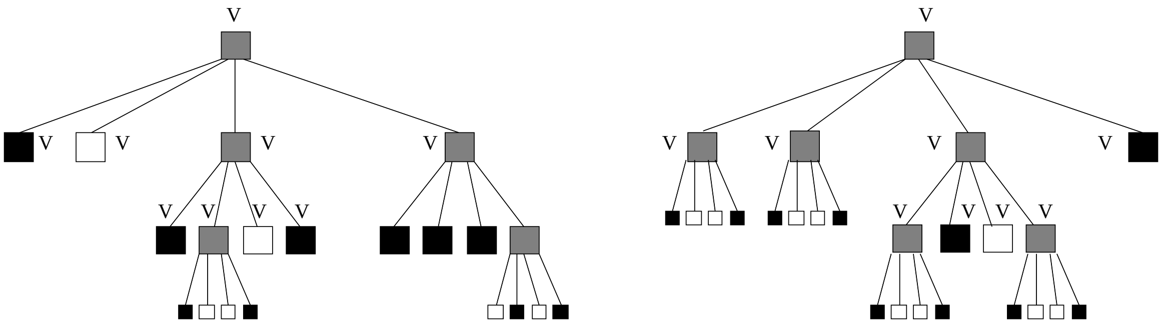

Figure 50: Union of two quad-tree images (V indicates node visited)

examining nodes 2, 3, 4, and 6. The heuristic chose to use caching when determining the neighbors of a quadrant, because they may be far away the tree.

6.2.10 Union

Union computes the union of two quad-tree encoded raster images [Shn81]. The trees are traversed simultaneously, and decisions are made whenever either of the traversals reaches a leaf. When a black leaf is encountered, this becomes the subtree at the current position in the union tree. A white leaf in one tree results in the corresponding subtree of the other tree being copied at the current position in the union tree. When there are two interior nodes, the procedure is called recursively for

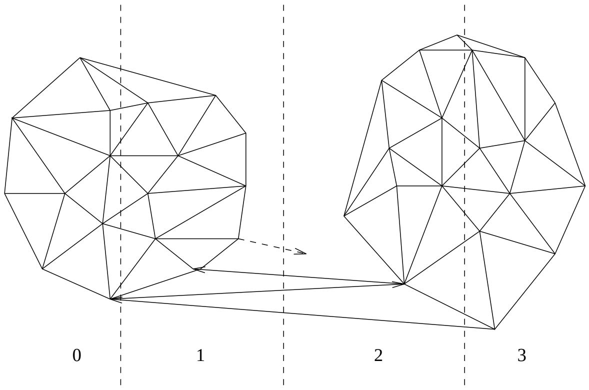

Figure 51: Merge phase of Voronoi

the subtrees rooted at these nodes. Because a decision can be made when either of the trees reaches a leaf, there is very little work done in this algorithm, corresponding to smaller speedups than in Perimeter. As seen in Figure 50, only 18 of the 46 nodes in the two quad-trees are visited in computing the union. For the input used to report results for this benchmark, only 4,531 of the approximately 93,000 nodes in each tree were traversed recursively.

As with `TreeMultAdd` (the example in Chapter 3), the heuristic will select to use migration for the traversal of one tree and caching for the other. Since we have placed the trees on the processors so that corresponding nodes are on the same processor, there are no cache misses. We will discuss this layout in further detail in Chapter 7.

6.2.11 Voronoi Diagram

The Voronoi Diagram benchmark [GS85, LS80] generates a random set of points and computes a Voronoi Diagram for these points. The points are stored in a binary tree sorted by X-coordinate. To compute a Voronoi Diagram, the algorithm computes Voronoi Diagrams of the two subtrees recursively (thereby dividing the set of points at the median), and merges them to form the final result. The merge phase walks along the convex hull of the two sub-diagrams, and adds edges to knit them together to form the Voronoi Diagram for the whole set. The points are assigned to processors so that the subtrees at a fixed depth are evenly distributed.

A partially completed merge phase (with points distributed among 4 processors) is shown in Figure 51. Walking along the convex hull of a single subresult is best done with migration, but the merge phase walks along two subresults, alternating between them in an irregular fashion. As a result, the heuristic chose to pin the computation on the processor that owns the root of one of the subresults and use software caching to bring remote subresults to the computation. This version performs dramatically better than an early version that used only migration [RCRH95]; nevertheless, the heuristic does not make the optimal choice in this situation; it would be better to traverse one of the subresults while caching the other (such a version has a speedup of over 12 on 32 processors). Even without the overhead, we would not obtain perfect speedup, because as with TSP, many of the merges leave some processors idle.

6.3 Summary

Our implementation of Olden on the Thinking Machines CM-5 demonstrates the effectiveness of the local-knowledge coherence scheme and our programming model. For the six benchmarks that cache data, the local-knowledge scheme performed best, as the number of unnecessary misses was small, and the savings from not tracking writes substantial.

Our results indicate that the heuristic makes good selections, and only occasionally requires the programmer to specify local path lengths to obtain the best performance. The performance of only one of the eleven benchmarks, TSP, improved when the default local path lengths were replaced by user-specified local path lengths. In one case, Voronoi, the heuristic was unable to select the best blend of computation migration and caching. Even so, we obtained a speedup of 8.76 on 32 processors.

Two other benchmarks, Barnes-Hut and MST, while obtaining speedups, demonstrated possibilities for improvement. In Barnes-Hut, we were unable to parallelize the tree-building phase. For MST, the synchronization time caused poor performance. In both cases, the programs would benefit from more inexpensive synchronization mechanisms.

Overall, the Olden implementations provided good performance and required few programmer modifications. Where timings from other systems were available (Power, Barnes-Hut, and EM3D), the Olden implementations performed comparably. In each case, the Olden implementation required less programmer effort. These results indicate that Olden provides a simple yet efficient model for parallelizing programs using dynamic data structures.

Chapter 7

Profiling in Olden

After getting a program to work correctly, the programmer's attention usually turns to making it run faster. The local path lengths in Olden provide a mechanism to allow the programmer to give hints to the system regarding the data layout. These hints allow the system to reduce communication by selecting the appropriate blend of computation migration and caching. As seen in TSP, changing these local path lengths may lead to dramatic changes in the speed of the algorithm. By having the system provide feedback regarding the choice of local path lengths, the programmer can avoid guesswork or a tedious search of the parameter space in selecting the local-path-length values. Yet, even after obtaining correct local-path-length values, the program may still perform poorly. For example, a program with a poor data layout may have excessive communication overhead. Determining where the program is performing expensive operations, such as cache misses and migrations, and how often these occur may help the programmer improve the layout of the data or the computation.

In Olden, we provide profiling tools that allow the programmer to check the local path lengths of the fields of the data structure and also examine the number and type of communication events caused by each line in the program. The local path lengths of the fields of a structure may be computed at run time by calling an Olden procedure with a pointer to the root of the structure. When the appropriate compile-time flag is

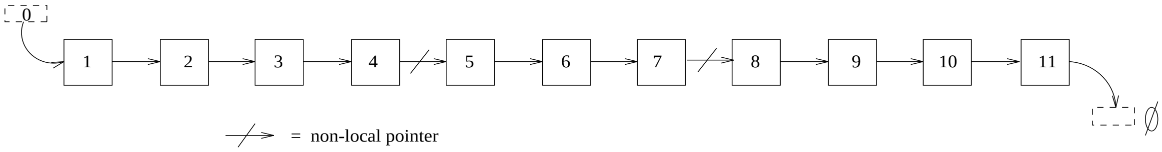

Figure 52: A sample linked list

used, the compiler will generate code to record the number of communication events corresponding to each line of the program.

7.1 Verifying Local Path Lengths

Recall from Chapter 5 that the local path length of a field of a structure is a hint that gives information about the expected number of structure items that will be made local by a migration following a traversal of a given field of the structure. In this section, we will describe formally how to compute a local path length for a particular field of a data structure.

To simplify our definitions, we assume that each leaf node in the graph has one child, a special node, ϕ. The home of ϕ is unique (i.e., $\forall v \in V, home(\phi) \neq home(v)$), and ϕ has no descendants. We ignore all null pointers. We define a *local path* to be a sequence of vertices, $v_1, v_2, \ldots, v_n$, along a traversable set of pointers in the graph such that $\forall i, j, 2 \leq i, j \leq n-1$, $home(v_i) = home(v_j)$. We say a local path is *maximal* if $home(v_1) \neq home(v_2)$ and $home(v_{n-1}) \neq home(v_n)$. The length of a local path, $v_1, v_2, \ldots, v_n$, is n-2.[1] The lengths of the maximal local paths give an estimate of the benefit of using migration, as they provide an estimate of how many future references will be made local by performing a migration. We define the local path length of a field, F, of a data structure to be the average length of maximal local paths beginning with a pointer along field F.

For an example of computing local path lengths, consider the data structure in Figure 52. For the `next` field of this list, there are two non-local pointers, $[v_4, v_5]$ and

[1] This differs from the normal notion of path length (see, for example, [PS82]). The length of a local path expresses the number of vertices in the path that have the same home.

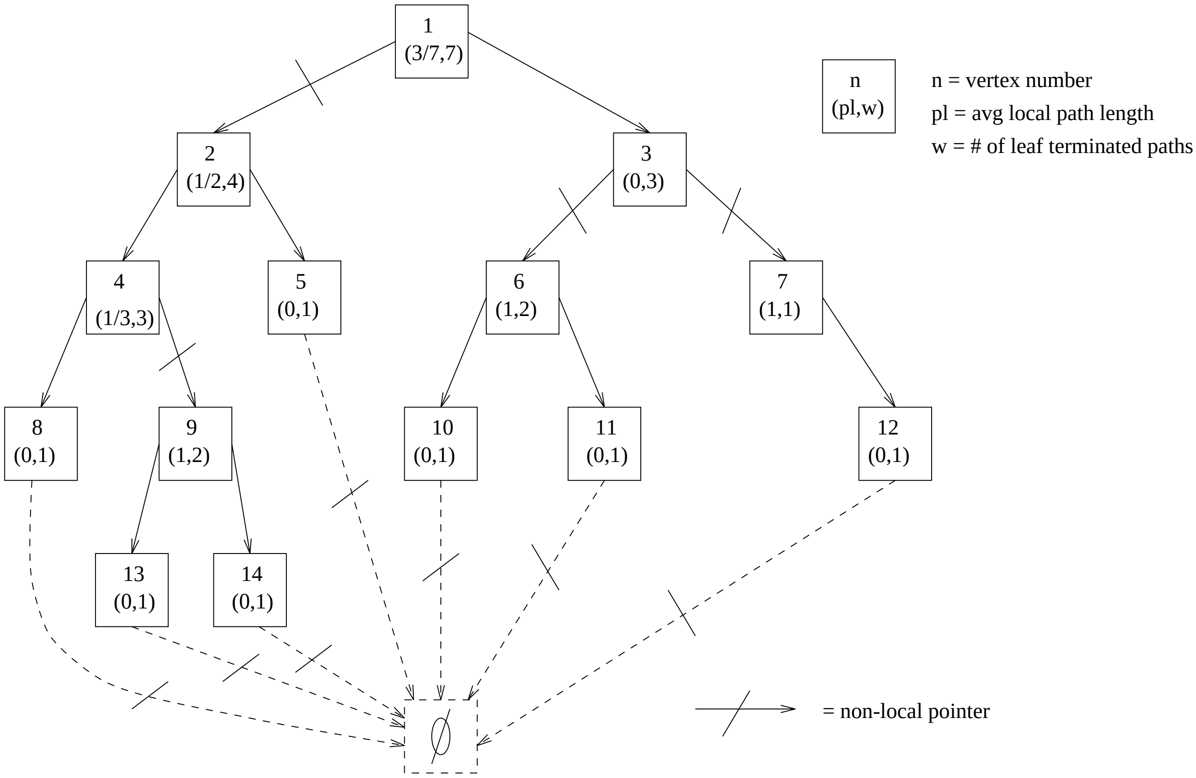

Figure 53: A sample tree with local-path-length information

$[v_7, v_8]$. The corresponding maximal local paths are $[v_4, \ldots, v_8]$ and $[v_7, \ldots, v_{11}, \phi]$, which have lengths of 3 and 4. Ignoring the initial pointer from v_0 to v_1, the average maximal local path length is 3.5, and therefore the local path length for the `next` field is 3.5. If the traversal of the structure were to begin on a different processor than the owner of v_1, $[v_0, v_1, \ldots, v_5]$ would be another maximal local path (where v_0 is a dummy vertex corresponding to the variable holding the pointer to v_1), and the average maximal local path length would be $\frac{11}{3}$. For simplicity, we will assume for the remainder of this chapter that traversals begin on the owner of the root; however, the profiler considers initial pointers when computing local path lengths.

For structures that have more than one field, there are multiple maximal local paths corresponding to a single non-local pointer. In this case, we weight each maximal local path, P, by the total number of leaf-terminated paths that have P as a subpath. This corresponds to assuming that all traversals are equally likely. Returning to the linked list example, each maximal local path is a subpath of exactly one leaf-terminated path; therefore, we compute a simple average of the maximal local path lengths. For an example of the weighted case, consider the tree in Figure 53. There are three maximal local paths beginning with a right edge ($[v_3, v_7, v_{12}, \phi]$, $[v_4, v_9, v_{13}, \phi]$, and $[v_4, v_9, v_{14}, \phi]$), and their weighted average local path length is 2.0. For the left field, there are five maximal local paths, $[v_1, v_2, v_5, \phi]$, $[v_1, v_2, v_4, v_9]$, $[v_1, v_2, v_4, v_8, \phi]$, $[v_3, v_6, v_{10}, \phi]$, and $[v_3, v_6, v_{11}, \phi]$, and their weights are 1,2,1,1 and 1 respectively (note there are two leaf-terminated paths from v_9, through v_{13} and v_{14}). Consequently, their weighted average local path length is $\frac{13}{6}$. For this tree, the local path length of the left field is 2, and the local path length of the right field is $\frac{13}{6}$.

We can compute these local path lengths in linear time with respect to the size of the graph using dynamic programming. For each node of the structure, using a simple depth-first traversal, we compute both the number of leaf-terminated paths from that node, and also the average local path length for paths beginning at that node. For a leaf node, the average local path length is zero, and the number of leaf-terminated paths is one. For any interior node, if it has already been visited (as might occur in a traversal of a DAG), we immediately return the computed values. Otherwise, we traverse all of its children. The number of paths from an interior node is then simply the sum of the number of paths from each of its children. The average local path length for an interior node, x, is given by:

$$\sum_{c \in children(x)} same\_proc(c, x) * (1 + avg\_local\_path\_length(c))$$

where $same\_proc(c, x)$ is 1 if c and x are on the same processor, and 0 otherwise. Since we can compute the average local path length of a node, x, incrementally from the average local path lengths of its children using $O(degree(x))$ operations, we can perform the computation for all the nodes in the graph in linear time with respect to the size of the graph.

During the same traversal, we can also compute the local path length for each field. This is shown in the program in Figure 54. During the traversal, if a non-local pointer is encountered, we increment the total weight and weighted sum for that field.

```
struct check_dag2 {
  double avg_lpath_len,leaf_paths,weight_fld1,sum_fld1,weight_fld2,sum_fld2;
};
struct check_dag2 Compute_Dag2_LPLs(dag2 p) {
  struct check_dag2 f1res, f2res, myres;

  if (p==NULL || IsLeaf(p)) {
    return {0.0, IsLeaf(p), 0.0, 0.0, 0.0, 0.0}
  }
  if (PreviouslyVisited(p)) return {Avg_Len(p),Paths(p),0.0,0.0,0.0,0.0};
  Record_Visited(p);

  f1res = Compute_Dag2_LPLs(p->fld1);
  f2res = Compute_Dag2_LPLs(p->fld2);
  myres = f1res + f2res;
  if (LocalPointer(p->fld1)) {
    myres.avg_lpath_len += (f1res.avg_lpath_len)*(1+f1res.leaf_paths);
  } else {
    myres.weight_fld1 += f1res.leaf_paths;
    myres.sum_fld1 += f1res.leaf_paths*(1+f1res.avg_lpath_len); }
  if (LocalPointer(p->fld2)) {
    myres.avg_lpath_len += (f2res.avg_lpath_len)*(1+f2res.leaf_paths);
  } else {
    myres.weight_fld2 += f2res.leaf_paths;
    myres.sum_fld2 += f2res.leaf_paths*(1+f2res.avg_lpath_len);  }
  myres.avg_lpath_len /= myres.leaf_paths;
  RecordResult(myres,p);
  return myres;
}
void Print_LPLs(dag2 p) {
  struct check_dag2 x;
  int LPL1, LPL2;

  x = Compute_Dag2_LPLs(p);
  LPL1 = (x.weight_fld1) ? (x.weight_fld1/x.sum_fld1) : 100;
  LPL2 = (x.weight_fld2) ? (x.weight_fld2/x.sum_fld2) : 100;
  printf("Local path length of fld1 is %d and fld2 is %d",LPL1,LPL2);
}
```

Figure 54: Sample program to compute local path lengths of DAG with degree ≤ 2

Note that if a node, x, is visited a second time, we do not double count the non-local pointers that are in the subgraph rooted at x. Once the entire structure has been traversed, the local path length is for the field is 100 if no non-local pointers are present along that field, or the computed value otherwise. Since the path affinity is given by $1 - \frac{1}{local\ path\ length}$ and we compute path affinities using integer arithmetic, all local path lengths greater than or equal to 100 will be converted to a path affinity of 99%.

The procedures used to compute local path lengths (such as the ones shown in Figure 54) are generated automatically using preprocessor macros. To use the profiler to compute local path lengths for a tree, the programmer needs to add only two lines of code. In the header, the statement `CHECK2(tree_t *,left,right,tree)` informs the preprocessor to generate code to compute local path lengths for a data structure having two pointer fields, `left` and `right`, of type `tree_t *`. The last argument, `tree`, is used to generate a unique name `tree`. Once the data structure has been built, a call to `Docheck_tree(root)` computes and prints the local path lengths for the two pointer fields.

In Table 7, we give the computed local path lengths for each field in all of our benchmarks. For two of these benchmarks, TSP and Health, the heuristic will make different choices if these values are used in place of the default local path lengths. Unfortunately, the results are mixed. For TSP we get a large performance gain by specifying a local path length other than the default; whereas for Health, we get a slight decrease in performance for large numbers of processors.

As noted in Chapter 6, the traversal of each subresult of TSP has high locality, and thus we can reduce substantially the communication by using migration to traverse these subresults. Using the profiler, we see that the run-time-computed local path length of the next field is 100, which is representative of this locality. By changing to the computed local path length value, we increase the speedup for TSP on 32 processors from 6.4 to 15.8. Thus, with the profiler, the programmer can discover this opportunity for increased performance without having to do a detailed analysis of the program.

Table 7: Computed Local Path Lengths

Benchmarks	Fields and Local Path Lengths						
TreeAdd	left 6.66	right 100					
Power	next_branch 50	next_lateral 100					
TSP†	next 100						
MST	next 100						
Bitonic Sort	left 5.55	right 100					
Voronoi	left 2.33	right 4	next 100				
EM3D	next 100						
Barnes-Hut	subp[0] 2.44	[1] 2.63	[2] 2.70	[3] 2.63	[4] 2.63	[5] 2.63	[6] 2.63	[7] 3.33
Perimeter	nw 100	ne 4	sw 5	se 5			
Health†	forward[0] 100	[1] 2.04	[2] 2.33	[3] 2.33	next 100		
Union	nw 100	ne 4.35	sw 5.55	se 5.88			

† indicates that computed local path lengths yield different heuristic choices than using defaults

```
while(p != patient)
  {
    p = list->patient;
    list = list->forward;
  }
```

Figure 55: Linked list traversal in Health

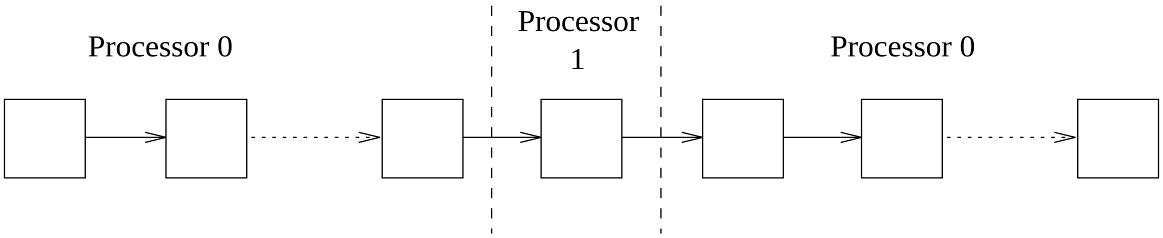

Figure 56: Example linked list

The performance tradeoff for Health is more subtle. For this benchmark, each list is located almost entirely on a single processor. Because of this, the use of the computed local path length values will reduce the number of pointer tests that must be performed. Consider the code fragment in Figure 55. Using computation migration, we can eliminate the pointer test for the reference to `list->patient`, as we know that `list` is local following the reference, `list->forward`. Using caching, we cannot eliminate the second test unless the programmer provides a guarantee that structures are aligned with cache-line boundaries. Changing the local path length value removes 6.8 million pointer tests and reduces the running time on one processor from 44.4 to 42.2 seconds. For large numbers of processors, however, the communication cost is increased. Consider the list shown in Figure 56. There is a single node on Processor 1, in the middle of a very long list on Processor 0. Because the list length is very large, the local path length will also be large. Nonetheless, using caching for this list traversal is preferable, because there is only one cache miss, instead of 2 migrations using computation migration. Since the lists in Health resemble the list in Figure 56, the default version, which uses caching, performs slightly better for large numbers of processors. On 32 processors, the running time is 1.94 seconds using the default

```
File: tsp.c Line: 00175 Local: 130684 Migrates: 133
File: tsp.c Line: 00182 Loc: 250 Hit: 0 Miss: 5 Rem_Wr: 0
File: tsp.c Line: 00183 Loc: 250 Hit: 5 Miss: 0 Rem_Wr: 0
File: tsp.c Line: 00236 Loc: 73 Hit: 11 Miss: 0 Rem_Wr: 11
```

Figure 57: Portion of program event count for TSP

version, and 2.27 seconds using the run-time-computed local path lengths. We can examine this tradeoff more carefully using the profiler's event counter.

7.2 Monitoring Program Events

Local path length information alone is not sufficient to get the best performance. It is also useful to be able to determine which portions of the program are performing expensive operations. The Olden compiler has a profiling flag (-p) that allows the user to get a count of the number of these events during execution. If the profiling flag is used, at the end of execution, each processor will print a table of events collected during that run. The table indicates, for each line of the program containing a pointer dereference, the number of times the pointer was local, and the number of migrations, cache hits, cache misses and remote writes that occurred at this dereference. Awk scripts are provided to merge these tables into a single overall table by program line, and to compute an overall total number of events. A portion of the overall table by program line for TSP is given in Figure 57.

To track the events, before each dereference, the compiler inserts code that calls a profiling routine in the run-time system. Along with the pointer, the line number, file name, and type of reference (either migration, cache read, or cache write) are passed as arguments. The run-time system then looks up this line in a hash table, inserting an entry if necessary. The pointer is then tested. If the pointer is local, the local counter is incremented and the system returns immediately to user code. For remote migrating references, it is also only necessary to increment the appropriate counter and return. For remote cache references, the profiler inquires if the line is present in the cache, and updates the appropriate counter.

```
for (a=a->next; a!=tmp; a=a->next) {
    test = distance(a,t);
    if (test < mindist) {
      mindist = test;
      min = a;
      } /* if */
    } /* for a... */
```

Figure 58: Closest-point loop from TSP

For the Health benchmark, the profiler allows us to determine why the computed local path length decreased performance for large numbers of processors. Although on 32 processors there are 6.8 million fewer pointer tests and 336 fewer cached remote writes using the computed local path length, the number of cache misses increases by 339, and there are 336 more migrations. Since migrations are the most expensive operation, their additional cost outweighs the savings from fewer pointer tests and cached remote writes.

Union demonstrates the importance of using a good data layout. If the two trees are not distributed across the processors in the same manner, the traversal will have to perform a large amount of caching. On a 32-processor run where the trees are laid out differently, there are 795 additional cache misses. Using the profiler, we can readily identify the source lines causing these misses, and can see that they result from the traversal of the second tree. By changing the layout of the data according to this information, we increase the speedup on 32 processors from 4.26 to 11.3.

Using the profiler, the programmer can also find ways to change the program slightly to reduce communication. Consider the loop from TSP shown in Figure 58. This loop traverses the subresult beginning at `a` to find the point closest to `t`. Since the local path length of the next field is large, the heuristic will use migration for references to `a` inside this loop. Consequently, references to `t` will have to be cached on many processors as the computation migrates. To avoid this, we lift the read of `t` out of the loop (since it is invariant) by reading the location of `t` into the stack frame (and modifying the distance function accordingly). On a 32-processor run, this

```
t_x = t->x;
t_y = t->y;

for (a=a->next; a!=tmp; a=a->next) {
  test = distance_xy(a,t_x,t_y);
  if (test < mindist) {
    mindist = test;
    min = a;
    } /* if */
  } /* for a... */
```

Figure 59: Modified closest-point loop from TSP

eliminates 570 cache misses and changes 264,618 cache hits into local references. This increases the speedup from 15.0 to 15.4. The modified code is shown in Figure 59.

7.3 Summary

With information from a run-time profile of a program, the programmer can find ways to improve performance, and can focus on those portions of the program that have expensive operations. The Olden profiler provides two types of information that are useful for analyzing the program's performance: local path length values and event counts. With the computed local path length values, the programmer can (most often) verify that the defaults are appropriate, or make changes that might dramatically increase performance. With the event count information, the programmer is able examine to the appropriateness of the data layout, and focus on changing portions of the program that will have the greatest impact on performance.

Chapter 8

Related Work

Much work has been done on providing support for programming parallel machines. In this chapter, we describe how our work relates to that of other groups. We divide the projects discussed into three categories: object-oriented systems, parallel versions of C using fork and join, and other related projects not fitting in the first two categories.

8.1 Object-Oriented Systems

Emerald [JLHB88] is an object-oriented language developed at the University of Washington for programming distributed systems. The language provides primitives to locate objects on processors explicitly, and also has constructs for moving objects. Each method is invoked on the processor owning that object, migrating the computation at the invocation. Arguments to the method may be moved to the invocation site on the basis of compile-time information.

Amber [CAL*89] is a subset of C++ designed for parallel programming with a distribution model and mobility primitives derived from Emerald; however, all object motion is done explicitly. Amber also adds replication, but only for immutable objects. As in Olden, a computation may migrate within an invocation, but Amber computations only migrate if the objects they reference are moved out from under them explicitly by another thread.

COOL [CGH93] and Mercury [FK92] are also extensions of C++. Unlike Emerald and Amber, they are designed to run on a shared-memory multiprocessor. In COOL, by default, method invocations run on the processor that owns the object; however, this can be overridden by specifying affinities for the methods. There are three different types of affinities: *object*, *processor*, and *task*. If the programmer specifies that a method has affinity with a particular object (not the base object), then it will be run on the processor that owns that object. A processor affinity is similar, and could be viewed as an object affinity to a dummy object on the specified processor. Object affinities are used to increase locality; processor affinities are specified to balance the load. To increase cache reuse of an object, COOL allows the programmer to specify a task affinity with respect to that object. For each processor, the COOL run-time system has an array of task queues. There is a queue for each object owned by that processor, for each collection of tasks with task affinity, and for the collection of tasks with processor affinity for that processor. Load balancing is accomplished by work stealing. If a processor is idle, it may steal a task-affinity task queue from another processor, as these tasks are not required to run on a particular processor. Mercury also has a notion of object affinity, but uses templates rather than a scheduler. The template associated with each method specifies how to find the next task to execute.

The Concert system [CKP93, KC93] provides compiler and run-time support for efficient execution of fine-grained concurrent object-oriented programs. This work is primarily concerned with efficiency rather than language features. They combine static analysis, speculative compilation, and dynamic compilation. If an object type and the assurance of locality can be inferred from analysis, method invocations on that object can be inlined, improving execution efficiency. If imprecise information is available, it is sometimes possible to optimize for the likely cases, selecting amongst several specialized versions at run time. When neither of these is applicable, dynamic compilation is sometimes used to optimize based on run-time information. The compiler will select which program points will be dynamically compiled. Concert provides a globally shared object space, common programming idioms (such as RPC and tail forwarding), inheritance, and some concurrency control. Objects are single threaded and communicate asynchronously through message passing (invocations). Concert also provides parallel collections of data, structures for parallel composition, first-class messages, and continuations. A major goal of the Concert project is to provide efficient support for fine-grain concurrency. The system seeks to accomplish this by automatically collecting invocations into groups using structure analysis [PCK93].

Each of these systems supports more general synchronization than Olden; however, they do not provide the automatic selection between caching and computation migration. Emerald does have heuristics to move objects automatically, but does not support replication. Additionally, with the exception of Concert, these languages do not support inlining; consequently, their use of objects adds additional overhead.

8.2 Extensions of C with Fork-Join Parallelism

Cid [Nik94], a recently proposed extension to C, supports a threads and locks model of parallelism. Cid threads are lightweight and the thread creation mechanism allows the programmer to name a specific processing element on which the thread should be run. Unlike Olden, Cid threads cannot migrate once they have begun execution. This makes it awkward to take advantage of data locality while traversing a structure iteratively. Cid also provides a global object mechanism that is based on global pointers. The programmer explicitly requests access to a global object using one of several sharing modes (for example, readonly) and is given a pointer to a local copy in return. Cid's global objects use implicit locking and the run-time system maintains consistency.

One of Cid's design goals is to use existing compilers. While this makes it easier to port to new systems, it does not allow it to take advantage of having access to the code generator and compile-time information. For example, once a Cid fork is inlined, the system cannot change its mind. In Olden, an inlined future may be stolen at a later time, should the processor become idle. Additionally, Olden programs more closely resemble their sequential C counterparts, as handling remote references is done implicitly.

Cilk [BFJ*95, BJK*95] is another parallel extension of C implemented using a preprocessor. The implementation introduces a substantial amount of overhead per call, as each call to a Cilk procedure requires a user-level spawn. Unlike Cid and Olden, these spawns are never inlined.

Cilk introduces dag-consistent shared memory, a novel technique to share data across processors. This is implemented as a stack, and shared memory allocations are similar to local variable declarations. Caching is done implicitly at the page level, and work is placed on processors using a work-stealing scheduler. By contrast, Olden does caching at the line-level, and is thus more efficient when small portions of a page are shared. Additionally, using computation migration, Olden can take advantage of data locality in placing the computation. Olden, however, gets its load balance from the data layout, and thus is only more efficient for programs where the data is placed so that the work is reasonably distributed among the processors. If the data is distributed non-uniformly, a Cilk program will perform better, as Cilk's work-stealing algorithm will generate a better load balance.

8.3 Other Related Work

Split-C [CDG*93] is a parallel extension of C that provides a global address space and maintains a clear concept of locality by providing both local and global pointers. Split-C provides a variety of primitives to manipulate global pointers efficiently. In a related piece of work, Lumetta et al. [LML*93a] describe a global object space abstraction that provides a way to decouple the description of an algorithm from the description of an optimized layout of its data structures.

Like Olden, Split-C is based on the modification of an existing compiler. Split-C, however, adopts a programming model where each processor has a single thread executing the same program. While this is perhaps convenient for expressing array-based parallelism, recursive programs must be written much more awkwardly.

Prelude [HWW93] is an explicitly parallel language that provides a computation model based on threads and objects. Annotations are added to a Prelude program to specify which of several mechanisms — remote procedure call, object migration, or computation migration — should be used to implement an object or thread. Hsieh later implemented MCRL [Hsi95], which, like Olden, provides both computation migration and data migration, and a mechanism that automatically selects between them. MCRL uses two heuristics to decide between computation and data migration, *static* and *repeat*. The static heuristic may be computed at compile-time for most functions and migrates computation for all non-local writes and data for all non-local reads. The repeat heuristic also always migrates computation for non-local writes, but makes a run-time decision for non-local reads based on the relative frequency of reads and writes. Computation migration is used for an object until two consecutive reads of that object have occured without any intervening writes. The heuristics used by MCRL are very different from those in Olden, as MCRL is designed for a different type of program. In Olden, we are concerned with traversals of data structures, and improving locality by migrating the computation if it will make future references local. The benchmarks we describe use threads that do not interfere. By contrast, MCRL is concerned with reducing coherence traffic for programs where the threads are performing synchronized accesses to the same regions with a large degree of interference. Olden cannot be used for such programs. Their heuristics for choosing between data and computation migration would perform poorly on our benchmarks, as they consider only the read/write patterns of accesses. On a read-only traversal of a large data structure on a remote processor, MCRL would choose to cache the accesses, as this causes no additional coherence traffic. Olden, however, would use migration, causing all references but the first to become local.

Orca [BK93, BKT92] also provides an explicitly parallel programming model based on threads and objects. Orca hides the distribution of the data from the programmer, but is designed to allow the compiler and run-time system to implement shared objects efficiently. The Orca compiler produces a summary of how shared objects are accessed that is used by its run-time system to decide if a shared object should be replicated, and if not, where it should be stored. Operations on replicated and local objects are processed locally; operations on remote objects are handled using a remote procedure call to the processor that owns the object. Orca performs a global broadcast at each fork. This restricts Orca to programs having coarse parallelism, and makes it awkward to express parallelism on recursive traversals of trees and DAGs. Also, Orca does not allow pointers, instead using a special graph type. This has the disadvantages of making it impossible to link together two different structures without copying one and also requiring that each dereference be done indirectly through a table.

Carriero et al.'s [CGL86] work on *distributed data structures* in Linda shares a common objective with Olden, namely, providing a mechanism for distributed processors to work on shared linked structures. In the details, however, the two approaches are quite different. The Linda model provides a global address space (tuple space), but no control over the actual assignment of data to processors. While Linda allows the flexibility of arbitrary distributed data structures, including arrays, graphs and sets, Olden's control over data layout will provide greater efficiency for programs using hierarchical structures.

Jégou [J9́4] uses the idea of computation migration to provide an environment for executing irregular codes on distributed memory multiprocessors. His system is implemented as an extension of FORTRAN. For each parallel loop, a single thread is started for each iteration. The data never moves; instead, when a thread needs non-local data, it is migrated to the processor that owns the data. The threads operate entirely asynchronously (i.e., no processor ever needs to wait for the result of a communication). This allows computation to be overlapped with communication. Because the threads migrate, detecting when the loop has terminated is non-trivial. Jégou provides an algorithm for detecting termination that requires $O(p^2)$ messages. Using computation migration alone for these types of programs is successful because there is sufficient parallelism to hide the communication latency; however, for divide-and-conquer programs, we have demonstrated that combining computation migration with software caching provides better performance.

	Global arrays	Global heap	Multi-threaded	Lazy task creation	Fine-grain synch.	Computation migration	Software caching	Auto selection	Implicit global pointers
Olden		✓	✓	✓		✓	✓	✓	✓
O-O systems	✓	✓	✓		✓	✓	*		
Cid	✓	✓	✓	**	✓		✓		
Cilk	✓	✓	✓				✓		
Split-C	✓	✓			✓				
MCRL	✓	✓	✓		✓	✓	✓	✓	
Orca	✓	✓	✓						
Linda	✓	✓	✓						
Jegou	✓					✓			

* supports data migration and replication of read-only objects
** has irrevocable inlining

Figure 60: Summary of related work

Chapter 9

Conclusions and Future Work

We have presented a new approach for supporting programs that use pointer-based dynamic data structures on distributed-memory machines. In developing our new approach, we have noted a fundamental problem with trying to apply run-time-resolution techniques, currently used to produce SPMD programs for array-based programs, to pointer-based programs. Array data structures are directly addressable. In contrast, dynamic data structures must be traversed to be addressable. This property of dynamic data structures precludes the use of simple local tests for ownership, and therefore makes the run-time resolution model ineffective.

Our solution avoids these fundamental problems by matching more closely the dynamic nature of the data structures. Rather than having a single thread on each processor, which decides if it should execute a statement by determining if it owns the relevant piece of the data structure, we instead have multiple threads of computation, which are allowed to migrate to the processor owning the data they need. Along with this computation migration technique we provide a `futurecall` mechanism, which introduces parallelism by allowing processors to split threads, and a software caching mechanism, which provides an efficient means for a thread to access data from multiple processors simultaneously. In addition, we have implemented a compiler heuristic requiring minimal programmer input that automatically chooses the whether to use computation migration or software caching for each pointer dereference. We have also built a profiler, which will compute the local-path-length information used by the heuristic, and allow the programmer to determine which lines of the program cause communication events.

We have implemented our execution mechanisms on the CM-5, and have performed experiments using this system on a suite of eleven benchmarks. Our results indicate that combining both computation migration and software caching produces better performance than either mechanism alone, and that our heuristic makes good selections with minimal additional information from the programmer. The Olden profiler can generate this information automatically by examining a sample run of the program. Where comparisons were available, our system's performance was comparable to implementations of the same benchmarks using other systems; however, the Olden implementation was more easily programmed.

There are several areas for future work. First, the performance improvements gained by combining computation migration and software caching are not limited to distributed-memory machines. Programs run on both networks of workstations and recently proposed hybrid shared-memory/message-passing machines, such as Alewife [A+91], FLASH [K+94], and Tempest/Typhoon [RLW94a], could benefit from the combination. Implementations of Olden for such machines would use different thresholds for choosing between computation migration and caching. The threshold for a network of workstations would favor computation migration, because of the network's high communication latency, whereas the threshold for machines with extensive hardware support would favor caching.

Second, porting Olden to a system with fine-grain access control, such as Tempest, will allow us to make pointer tests cheaper. There is, however, a tradeoff between the cost of the pointer test and the overhead of processing interrupts on misses. Experimentation is necessary to determine whether explicit pointer tests, hardware interrupts, or a combination of the two provides the best performance.

Also, we would like to explore incorporating fine-grain synchronization mechanisms in Olden. Currently, parallel tasks in Olden are not allowed to interfere. While this allows us to use a relaxed consistency model, the system is not able to parallelize the incremental building of data structures, such as the tree-building phase of the Barnes-Hut benchmark.

Finally, the techniques for parallelizing array and pointer-based programs differ greatly. For arrays, commonly a single thread per processor approach is used, whereas for pointer-based programs, we suggest using multiple threads per processor. We would like to find a unified framework for parallelizing both array and pointer-based codes, which would allow parallelization of programs that use both types of data structures.

In summary, the Olden system provides an efficient framework for parallelizing programs using pointer-based data structures on distributed-memory machines. Our implementation has yielded good performance on a suite of eleven benchmarks, and has also suggested several avenues for continued research. Olden should provide a useful core for exploring these new topics.

Bibliography

[A+91] A. Agarwal et al. The MIT Alewife Machine: A large scale distributed-memory multiprocessor. Technical Report MIT/LCS TM-454, MIT, 1991.

[AK87] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[AL91] A.W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, ACM, New York, April 1991.

[AL93a] S.P. Amarasinghe and M.S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 126–138, ACM, New York, June 1993.

[AL93b] J.M. Anderson and M.S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, ACM, New York, June 1993.

[ASU88] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1988.

[Ben80] J. Bentley. A Parallel Algorithm for Constructing Minimum Spanning Trees. *Journal of Algorithms*, 1:51–59, 1980.

[BFJ*95] R. Blumofe, M. Frigo, C. Joerg, C. Leiserson, and K. Randall. Dag-Consistent Distributed Shared Memory. Technical report, Massachusetts Institute of Technology, September 1995. See `http://theory.lcs.mit.edu/~cilk`.

[BH86] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[BJK*95] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 207–216, July 1995.

[BK93] H. E. Bal and M. F. Kaashoek. Object Distribution in Orca using Compile-time and Run-time techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93)*, September 1993.

[BKT92] H. Bal, M. F. Kaashoek, and A. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, Mar 1992.

[BN89] G. Bilardi and A. Nicolau. Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared-Memory Machines. *SIAM Journal of Computing*, 18(2):216–228, 1989.

[BW73] D. Bobrow and B. Wegbreit. A Model and Stack Implementation of Multiple Environments. *Communications of the ACM*, 16(10):591–603, 1973.

[CAL*89] J.S. Chase, F. G. Amador, E.D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of

Multiprocesors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, ACM, New York, December 1989.

[CBZ91] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 152–164, 1991.

[CDG*93] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing 93*, pages 262–273, 1993.

[CGH93] R. Chandra, A. Gupta, and J. Hennessy. Data Locality and Load Balancing in COOL. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 249–259, May 1993.

[CGL86] N. Carriero, D. Gelernter, and J. Leichter. Distributed data structures in Linda. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 236–242, January 1986.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2(2):151–169, October 1988.

[CKP93] A. Chien, V. Karamcheti, and J. Plevyak. The Concert System– Compiler and Runtime Support for Efficient, Fine-Grained Concurrent Object-Oriented Programs. Technical Report UIUCDCS-R-93-1815, Department of Computer Science, University of Illinois at Urbana-Champaign, June 1993.

[CLR89] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw-Hill, New York, 1989.

[CR95] M. Carlisle and A. Rogers. Software Caching and Computation Migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 29–38, 1995.

[Cra93] Cray Research, Inc. *Cray T3D System Architecture*. 1993.

[CRRH93] M. Carlisle, A. Rogers, J. Reppy, and L. Hendren. Early Experiences with Olden. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 1–20. Lecture Notes in Computer Science 768, Springer Verlag, 1993.

[CV88] H. Cheong and A. Veidenbaum. A Cache Coherence Scheme with Fast Selective Invalidation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 299–307, June 1988.

[DK93] E. Darnell and K. Kennedy. Cache Coherence Using Local Knowledge. In *Proceedings of Supercomputing 93*, pages 720–729, 1993.

[FH95] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, Redwood City, CA, 1995. ISBN 0-8053-1670-1.

[FK92] R. Fowler and L. Kontothanassis. Improving Processor and Cache Locality in Fine-Grain Parallel Computations using Object-Affinity Scheduling and Continuation Passing. Technical Report 411, University of Rochester– Computer Science Department, June 1992.

[FLR*94] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing 94*, 1994.

[Ger90] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, 1990.

[GLL*90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[GS85] L. Guibas and J. Stolfi. General Subdivisions and Voronoi Diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[Gup92] R. Gupta. SPMD Execution of Programs with Dynamic Data Structures on Distributed Memory Machines. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 232–241, Los Alamitos, California, April 1992. IEEE Computer Society.

[Hal85] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HCD89] W. Horwat, A.A. Chien, and W.J. Dally. Experience with CST: Programming and Implementation. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 101–108, ACM, New York, June 1989.

[Hen90] L. J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, Ithaca, N.Y., January 1990.

[Hey92] John Heywood. Proverbs (1546). In Justin Kaplan, editor, *Familiar quotations: a collection of passages, phrases, and proverbs traced to their sources in ancient and modern literature by John Bartlett.* Little Brown and Co., Boston, 16th edition, 1992.

[HHN92] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference* *on Programming Language Design and Implementation*, pages 249–260, ACM, New York, June 1992.

[HHN95] J. Hummel, L. Hendren, and A. Nicolau. Path Collection and Dependence Testing in the Presence of Dynamic, Pointer-Based Data Structures. In B. Szymanski and B. Sinharoy, editors, *Languages, Compilers and Run-Time Systems for Scalable Computers (Proceedings of the 3rd Workshop)*, pages 15–27, May 1995. Kluwer Academic Publishers, Boston, MA, 1995.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler Optimizations for FORTRAN D on MIMD Distributed Memory Machines. In *Proceedings of Supercomputing 91*, pages 86–100, Los Alamitos, California, November 1991. IEEE Computer Society.

[HL93] G. Hogen and R. Loogen. A New Stack Technique for the Management of Runtime Structures in Distributed Implementations. Technical Report 3, RWTH Aachen, 1993.

[HN90] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, 1990.

[Hsi95] Wilson C. Hsieh. *Dynamic Computation Migration in Distributed Shared Memory Systems*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, September 1995.

[HWW93] W. Hsieh, P. Wang, and W. Weihl. Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 239–248, 1993.

[J94́] Yvon Jégou. Exécution de codes irrégulier par migration de tâches. Technical Report 867, Institut de Recherche en Informatique et Systèmes Aléatoires, November 1994.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[K+94] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[Kar77] R. Karp. Probabilistic Analysis of Partitioning Algorithms for the Traveling-saleman Problem in the Plane. *Mathematics of Operations Research*, 2(3):209–224, August 1977.

[Kar87] A. Karp. Programming for Parallelism. In *IEEE Computer*, May 1987.

[KC93] V. Karamcheti and A. Chien. Concert – Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Proceedings of Supercomputing 93*, pages 598–607, November 1993.

[Koe90] C. Koelbel. *Compiling Programs for Nonshared Memory Machines*. PhD thesis, Purdue University, West Lafayette, Ind, August 1990.

[Lam79] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[Lar91] J. R. Larus. Compiling Lisp Programs for Parallel Execution. *Lisp and Symbolic Computation*, 4(1):29–99, 1991.

[LCUW88] G. Lomow, J. Cleary, B. Unger, and D. West. A performance study of Time Warp. In *SCS Multiconference on Distributed Simulation*, pages 50–55, February 1988.

[LML*93a] S. Lumetta, L. Murphy, X. Li, D. Culler, and I. Khalil. Decentralized optimal power pricing: The development of a parallel program. In *Proceedings of Supercomputing 93*, pages 240–249, 1993.

[LML*93b] S. Lumetta, L. Murphy, X. Li, D. Culler, and I. Khalil. Decentralized Optimal Power Pricing: The Development of a Parallel Program. In *Proceedings of Supercomputing 93*, pages 240–249, Los Alamitos, California, November 1993. IEEE Computer Society.

[LS80] D. T. Lee and B. J. Schachter. Two Algorithms for Constructing a Delaunay Triangulation. *International Journal of Computer and Information Sciences*, 9(3):219–242, 1980.

[MB92] S. Min and J. Baer. Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):25–44, January 1992.

[McK80] E. C. McKenzie. *14000 Quips & Quotes for Speakers, Writers, Editors, Preachers and Teachers.* Baker Book House, Grand Rapids, MI, 1980.

[MKH91] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[Mül93] J. Müller. Parallelverarbeitung auf Workstation-Clustern mittels Express und Network-Linda. Diplomarbeit in Elektrotechnik, RWTH-Aachen, 1993.

[Nik94] Rishiyur Nikhil. Cid: A Parallel, “Shared-Memory” C for Distributed-Memory Machines. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[PCK93] John Plevyak, Andrew Chien, and Vijay Karamcheti. Analysis of Dynamic Structures for Efficient Parallel Execution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 37–56, 1993.

[PS82] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, 1982.

[RCRH95] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[Ric87] John A. Rice. *Mathematical Statistics and Data Analysis.* Wadsworth and Brooks/Cole Advanced Books and Software, Pacific Grove, CA, 1987.

[RLW94a] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[RLW94b] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, ACM, New York, April 1994.

[RP89] A. Rogers and K. Pingali. Process Decomposition Through Locality of Reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 69–80, ACM, New York, June 1989.

[RRH92] A. Rogers, J. Reppy, and L. Hendren. Supporting SPMD execution for dynamic data structures. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing.* Lecture Notes in Computer Science 757, Springer Verlag, 1992.

[RV89] E. S. Roberts and M. T. Vandevoorde. WorkCrews: An Abstraction for Controlling Parallelism. Technical Report 42, DEC Systems Research Center, Palo Alto, CA, April 1989.

[Sam81] H. Samet. Computing Perimeters of Regions in Images Represented by Quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3(6):683–687, November 1981.

[SFL*94] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[Shn81] Michael Shneier. Calculations of Geometric Properties Using Quadtrees. *Computer Graphics and Image Processing*, 16(3):296–302, July 1981.

[SPA92] SPARC International Inc. *The SPARC Architecture Manual*. Prentice Hall, 1992. Version 8.

[vECGS92] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrating communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[Wol89] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman Publishing, London, 1989.

[ZBG88] H. Zima, H. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6(1):1–18, 1988.

[ZSB94] M. Zekauskas, W. Sawdon, and B. Bershad. Software Write Detection for a Distributed Shared Memory. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, 1994.