Chapter 20. Supporting Dynamic Data Structures with Olden

Martin C. Carlisle[1] and Anne Rogers[2]

[1] US Air Force Academy, 2354 Fairchild Drive, Suite 6K41, USAFA, CO 80840
[2] AT&T Labs-Research, 600 Mountain Avenue, Murray Hill, NJ 07974

Summary. 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. 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 report on experiments with eleven benchmarks.

1. Introduction

To use parallelism to improve performance, a 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. 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 supports 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] These techniques are not suited to programs that use dynamic data structures [47],

[1] For example, see [2], [3], [12], [23], [27], [36], [48], and [52].

S. Pande, D.P. Agrawal (Eds.): Compiler Optimizations for Scalable PS, LNCS 1808, pp. 709-749, 2001.

because they rely on the fact that arrays, unlike dynamic data structures, are statically defined and directly addressable.

Olden's approach, by necessity, is much more dynamic than approaches designed for arrays. Instead of having a single thread running on each processor as is common in array-based approaches, 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. To handle remote references, Olden uses a combination of computation migration and software caching. Computation migration sends the computation to the data, whereas caching brings the data to the computation. 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.

Selecting whether to use computation migration or software caching for each program point would be very tedious, so Olden includes a compile-time heuristic that makes these choices automatically. The mechanism choice is dependent on the layout of the data; therefore, we provide a language extension, called *local path lengths*, that allows 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.

The rest of this chapter proceeds as follows: in Sect. 2, we describe Olden's programming model. We describe the execution model, which 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, in Sect. 3. Then, in Sect. 4, we discuss the heuristic used by the compiler to choose between computation migration and software caching. In Sect. 5, we report results for a suite of eleven benchmarks using our implementation on the Thinking Machines CM-5, and in Sect. 6, describe the Olden profiler and how to use it to improve performance. Finally, we contrast Olden with other projects in Sect. 7 and conclude in Sect. 8.

This paper summarizes our work on Olden. Several other sources [13, 14, 47] describe this work in more detail.

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 [34]. The necessary communication and thread management are handled by Olden's run-time system. In this section, we describe the programmer's view of Olden.

Our underlying machine model assumes that 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. Addresses in this distributed heap are represented as a pair, $\langle p, l \rangle$, that contains a processor name and a local address, which can be encoded in a single 32-bit word. All pointers are assumed to point 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 do not take the address of stack-allocated objects, which ensures that all pointers point into the heap.[2] 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 C— `ALLOC`, *local path lengths*, and *futures*— to allow the programmer to specify this information. `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 many parallel Lisps [26]. Throughout this section, 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 Fig. 2.1.

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

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

```
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);
    }
}
```

Fig. 2.1. `TreeAdd` function.

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. Olden also provides a mechanism to allow the programmer to provide a hint about the expected run-time layout of the data.

```
/* 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;
  }
}
```

Fig. 2.2. Allocation code

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. 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. In Fig. 2.2, we give an example function that allocates a binary tree such that the subtrees at a fixed depth are distributed evenly across the pro-

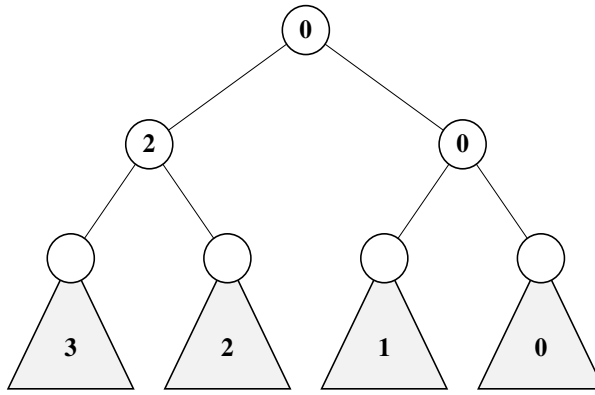

Fig. 2.3. Result of allocation on four processors

cessors (where the number of processors is a power of two). Figure 2.3 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.

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 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.

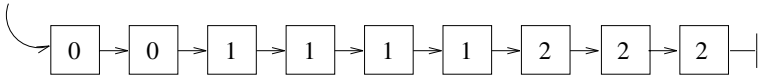

Fig. 2.4. A simple linked list

Determining the local path length is simplest for a linked list. Consider, for example, the linked list in Fig. 2.4. In this list, there are two nodes on Processor 0, 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 suffices to estimate the value using a high-level analysis of the structure. Recall the allocation shown in Fig. 2.3. 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 pointer fields. In Fig. 2.5, we show a data structure declaration with local-path-length hints.

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

Fig. 2.5. Sample local-path-length hint

In Sect. 6, 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 runtime 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 cases where the profiler-computed values differed significantly 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 [26]. The programmer may mark a procedure call as a *futurecall*, if it may be evaluated safely in parallel with its parent context. The result of a *futurecall* is a *future cell*, which serves as a synchronization point between the parent and the child. A *touch* of the future cell, which synchronizes the parent and child, must also be inserted by the programmer before the return value is used.

In our `TreeAdd` example, the 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 first call. Since the continuation of the second recursive call contains no work that can be parallelized, we do not mark it

```
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);
    }
}
```

Fig. 2.6. `TreeAdd` function using `future`.

as a future. Then, when the return value is needed, we must first explicitly touch the future cell. In Fig. 2.6, we show the `TreeAdd` program as modified to use futures.

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 section, 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. At the end of this section, we present an example, `TreeMultAdd`, that illustrates the execution model in action.

Due to space constraints, we do not discuss the implementation of Olden's run-time system in any detail. We refer the interested reader to Rogers et al. [47] and Carlisle [14] for details.

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; whereas,

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 at compile time using a mechanism that is described in Sect. 4.

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[3] 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. Once it migrates the thread, Processor P is free to do other work, which it gets from a work dispatcher in the run-time system.

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. Later we will explain computation migration in the context of a real program; here, we use the example in Fig. 3.1 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 the procedure, it is necessary to return control to Processor P, because it holds the stack frame of `H`'s 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.

Olden implements a simple optimization to circumvent a chain of trivial returns in the case that a thread migrates several times during the course of executing a single function. Upon a migration, the run-time system examines the current return address of the function to determine whether it points to the return procedure. 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

[3] The Olden compiler inserts explicit checks into the code to test a pointer dereference (recall that pointers encode both a processor name and a local address) to determine if the reference is local and to migrate the thread as needed. On machines that have appropriate support, the address translation hardware can be used to detect non-local references. This approach is preferable when the ratio of cost of faults to cost of tests is less than the ratio of tests to faults [4].

message. This allows the eventual return message to be sent directly to the original processor.

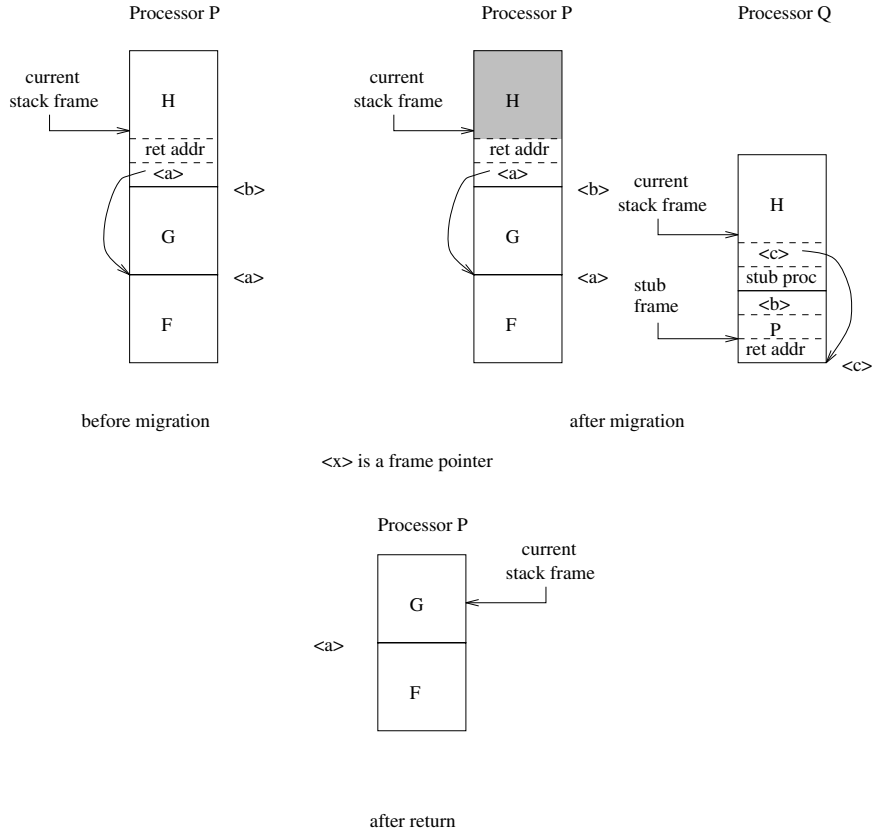

Fig. 3.1. Stack during computation migration (stacks grow up)

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 [50]. Each processor uses a portion of its local memory as a large, fully associative, write-through cache. A write-through cache is used because update messages can be sent cheaply from user level on the CM-5, and this allows us to overlap these messages with computation. As in Blizzard-S, we perform allocation on the page level, and perform transfers at the line level.[4] The main difference between Olden's cache and that in Blizzard-S is that we do not rely on virtual-memory support. We use a 1K hash table with a list

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

of pages kept in each bucket to translate addresses. 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 code before each heap reference that uses software caching. This code searches the lists stored in the hash table, checks the valid bit for the line, and returns a tag used to translate the address from a global to a local pointer. 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.

Once we introduce a local cache at each processor, we must provide a means to ensure that no processor sees stale data. Olden uses a relaxed coherence scheme where each processor invalidates its own cache at synchronization points. This coherence mechanism can be shown to be equivalent to sequential consistency [37] for Olden programs and is discussed in detail in Carlisle's dissertation [14].

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.

Our continuation-capturing mechanism is essentially a variant of the *future* mechanism found in many parallel Lisps [26]. 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. [41], is to save the futurecall's context (return continuation) on

a work list (in our case, a stack) and to evaluate the future's body directly.[5] If a processor becomes idle (either through a migration or through a blocked touch), then we grab a continuation from the work list and start executing it; this is called *future stealing*. In most parallel lisp systems, touches are implicit and may occur anywhere. In Olden, touches are done explicitly using the `touch` operation and there are restrictions on how they may be used. The first restriction is that only one `touch` can be attempted per future. The second is that the one allowed touch must be done by the future's parent thread of computation. These restrictions simplify the implementation of futures considerably and we have not found any occasions when it would be desirable to violate them.

Due to space constraints, we will not discuss the implementation of futures, except to note an important fact about how Olden uses the continuation work list. When a processor becomes idle, it steals work only from its own work list. A processor never removes work from the work list 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.

3.3 A Simple Example

To make the ideas of this chapter more concrete, we present a simple example. `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.

Figure 3.2 gives an implementation of `TreeMultAdd` that has been annotated with futures. We mark the left recursive call to `TreeMultAdd` with a `futurecall`; 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 Fig. 3.3. 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

[5] This is also similar to the workcrews paradigm proposed by Roberts and Vandevoorde [46].

`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, 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.

```
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);
    }
}
```

Fig. 3.2. `TreeMultAdd` with Olden annotations.

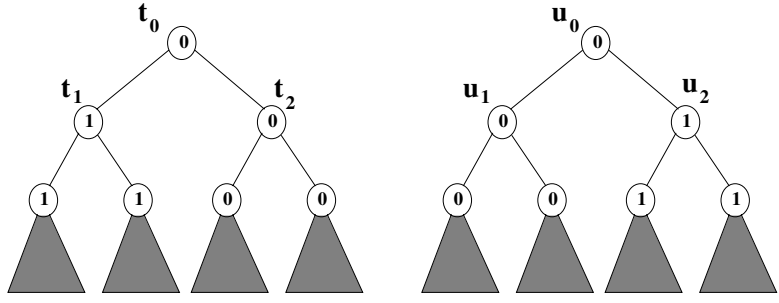

Fig. 3.3. Example input for `TreeMultAdd`

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 dereferences of `t` are local, and all dereferences of `u` remote. The subtree rooted at u_2 will be cached on Processor 0, and no 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 Fig. 3.4.

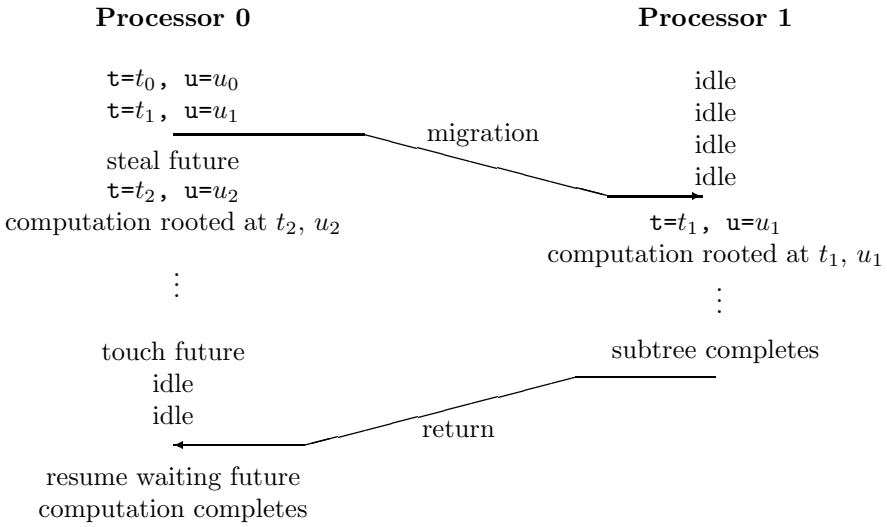

Fig. 3.4. Execution trace for `TreeMultAdd`

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 Sections 4 and 5.

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).

4. Selecting Between Mechanisms

As mentioned earlier, the Olden compiler decides, 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 Fig. 4.1). 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 $N \gg P$). Hence, it is better to use computation migration for such 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.

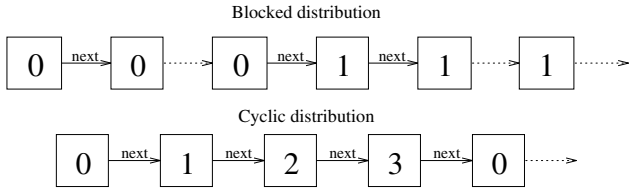

Fig. 4.1. Two different list distributions. The numbers in the boxes are processor numbers. Dotted arrows represent a sequence of list items.

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 traverse 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.

4.1 Using Local Path Lengths

Since the communication cost of using 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 Sect. 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 Sect. 6, we discuss how to compute local path lengths in more detail, and present the Olden profiler, which computes the local path lengths at run time.

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 [45]. 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 Fig. 4.1. 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.

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

Fig. 4.2. Sample local-path-length hint

In the remainder of this section, we 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 Fig. 4.2. 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%.

4.2 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 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 [1]. In both cases, an induction variable is a variable that is updated in a regular fashion in each iteration of the loop. In Fig. 4.3, `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).

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

Update Matrix

		before s	before t	before u
a f t e r	s	90		
	t		63	
	u	70		

Fig. 4.3. A simple loop with induction variables

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 Fig. 4.3, 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

```
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;
}
```

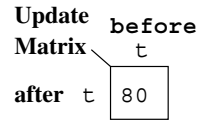

Fig. 4.4. An example of updates with `if-then`

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 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 surmise incorrectly 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 Fig. 4.4. 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%).

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

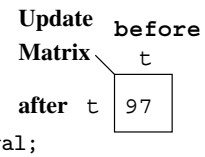

Fig. 4.5. TreeAdd

Second, we must have a rule for multiple updates via recursion. Consider the simple recursive program in Fig. 4.5. 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). Con-

sequently, the path affinity of the update of `t` because of the recursive calls is 97%, the probability that at least one will be local. 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 Fig. 4.3. 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 intra-procedurally. A full inter-procedural 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 inter-procedural analysis. In particular, we do not consider return values, or analyze loops that span multiple procedures. This limited inter-procedural 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 [30].

4.3 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 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. (Due to Olden's use of lazy rather than eager futures, new threads are generated only following migrations and blocked touches.)

```
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);
  }
}
```

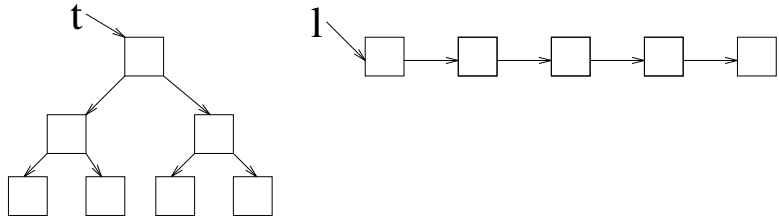

Fig. 4.6. Example with a bottleneck.

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 4.6 and 4.7. `WalkAndTraverse` is a procedure that for each list item traverses the tree.[6] If computation migration 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

[6] The syntax for specifying parallelism used in this example is not part of Olden and is used only to simplify the example.

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.

```
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);
  }
}
```

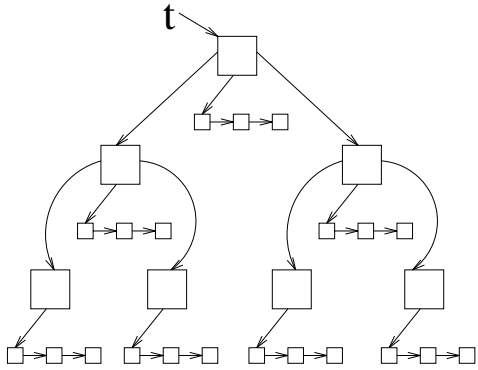

Fig. 4.7. Example without a bottleneck.

4.3.1 Threshold and Default Path Affinities. The migration 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 [19], 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 Fig. 2.3. 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 Fig. 4.4), will traverse a relatively short path that may cross a processor boundary several times. A complete tree traversal (such as the code in Fig. 4.5), 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 examine TSP more closely in the next section.

Returning to the declaration in Fig. 4.2 (page 723), we see why providing more precise local path length information is often unnecessary. Using default local path lengths rather than those from Fig. 4.2 changes the entry for `t` in the `TreeAdd` matrix (Figure 4.5) from 97 to 91, and the entry for `t` in the `TreeSearch` matrix (Figure 4.4) 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.

4.3.2 Example. We now return to the `TreeMultAdd` example from Fig. 3.2. 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%.

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 Fig. 4.8, 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`).

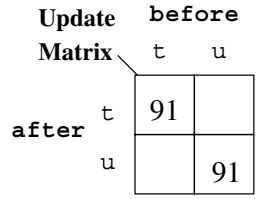

Fig. 4.8. Update matrix for TreeMultAdd

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) or (L is parallel) 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
```

Fig. 4.9. Selecting between computation migration and software caching

References to u are then cached. Since there is only one loop, the bottleneck analysis will not be performed.

4.3.3 Summary. In 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 Sect. 6), or compiler defaults. The selection process, as shown in Fig. 4.9, 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 the next section.

5. Experimental Results

In the previous sections, we have described Olden. In this section, we report results for eleven benchmarks using our Thinking Machines CM-5 implementation. 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 our special stack discipline, which is described in Rogers et al. [47]. The one-processor Olden implementation (*one*) includes these overheads. The difference between the two implementations provides a measure of the overhead.

Table 5.1 briefly describes each benchmark and Table 5.2 lists the running time of a sequential implementation plus speedup numbers for up to 32 processors for each benchmark. We report kernel times for most of the benchmarks to avoid having their data-structure building phases, which show excellent speedup, skew the results. Power, Barnes-Hut, and Health are the exceptions. We report whole program times (W) for Power and Barnes-Hut to allow for comparison with published results. We report whole program time for Health, because it does not have a data-structure build phase that would 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

Table 5.1. Benchmark Descriptions

Benchmarks	Description	Problem Size
TreeAdd	Adds the values in a tree	1024K nodes
MST	Computes the minimum spanning tree of a graph [8]	1K nodes
Power	Solves the Power System Optimization problem [39]	10,000 customers
TSP	Computes an estimate of the best Hamiltonian circuit for the Traveling-salesman problem [35]	32K cities
Barnes-Hut	Solves the N-body problem using a hierarchical method [7]	8K bodies
Bisort	Sorts by creating two disjoint bitonic sequences and then merging them [9]	128K integers
EM3D	Simulates the propagation of electro-magnetic waves in a 3D object [20]	40K nodes
Health	Simulates the Colombian health-care system [38]	1365 villages
Perimeter	Computes the perimeter of a set of quad-tree encoded raster images [49]	4K x 4K image
Union	Computes the union of two quad-tree encoded raster images [51]	70,000 leaves
Voronoi	Computes the Voronoi Diagram of a set of points [24]	64K points

Table 5.2. Results

Benchmarks	Heuristic choice	Seq. time (sec.)	Speedup by number of processors 1	2	4	8	16	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
Bisort	M+C	31.41	0.73	1.35	2.29	3.52	4.92	6.33
EM3D	M+C	1.21	0.86	1.51	2.69	4.48	6.72	12.0
Health^W	M+C	42.09	0.95	1.95	3.89	7.61	14.72	21.70
Perimeter	M+C	2.47	0.86	1.70	3.37	6.09	9.86	14.1
Union	M+C	1.46	0.76	1.48	2.88	5.11	7.97	11.30
Voronoi	M+C	49.73	0.75	1.38	2.41	4.23	6.88	8.76

^{W} – Whole program 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 Olden by adding an optimization phase to the compiler.

5.1 Comparison with Other Published Work

Several of our benchmarks (Barnes-Hut, EM3D, and Power) come from the parallel computing literature and have CM-5 implementations available. In this section, we briefly describe these benchmarks and compare our results to those available in the literature.

Barnes-Hut [7] simulates the motion of particles in space using a hierarchical ($O(n \log n)$) algorithm for computing the accelerations of the particles. The algorithm alternates between building an oct-tree that represents the particles in space and computing the accelerations of the particles. Falsafi et al. [21] 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. We discuss this more later.

EM3D models the propagation of electromagnetic waves through objects in three dimensions [20]. 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 `H` nodes. For a 64 processor implementation with 320,000 nodes, our implementation performs comparably to the ghost node implementation of Culler et al. [20], yet does not require substantial modification to the sequential code.

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 [40]. It was implemented originally on the CM-5 by Lumetta et al. in a variant of Split-C. On 64 processors, Olden's efficiency is about 80%, compared to Lumetta et al.'s 75%.

5.2 Heuristic Results

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. For most of the benchmarks, the default local path lengths were accurate. For three of the eleven benchmarks (Perimeter, MST, and TSP) the defaults were not accurate. In the case of Perimeter and MST,

specifying more accurate local path length information did not affect performance. Perimeter does a tree traversal and as we noted earlier the heuristic chooses migration for tree traversals by default. The local path length information that we specified did not contradict this choice. In the case of MST, the path length that we specified was for a data structure that was completely local. As a result, the benchmark's performance did not depend on whether the heuristic chose migration or caching.

In the case of TSP, changing the local path length information did affect performance. TSP is a divide-and-conquer algorithm with a non-trivial merge phase. The merge phase of TSP takes two Hamiltonian cycles and a single point, and combines them into a single cycle. Each merge is sequential and walks through the left sub-result followed by the right sub-result, which requires a migration for each participating processor. Given the default local path length, the heuristic will select caching for traversing the sub-results, as the procedure resembles a list traversal. This requires that all of the data be cached on a single processor. Because the sub-results have a long local paths, using migration results in O(number of processors) migrations rather than O(number of cities) remote fetches from using caching. As the number of cities greatly exceeds the number of processors, much less communication is required using migration. By specifying this higher local path length value, we obtain a speedup of 15.8 on 32 processors, as opposed to 6.4 with the default value.

The Voronoi Diagram benchmark is another case where the default values did not obtain the best performance; however, for this benchmark, changing the values did not improve performance. Voronoi is another divide-and-conquer algorithm with a non-trivial merge phase. 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. Since the sub-diagrams have long local paths, walking along the convex hull of a single sub-result is best done with migration, but the merge phase walks along two sub-results, alternating between them in an irregular fashion. The ideal choice is to use migration to traverse one sub-result and cache the other (such a version has a speedup of over 12 on 32 processors). As we do not have good branch-prediction in formation, the heuristic instead chooses to pin the computation on the processor that owns the root of one of the sub-results and use software caching to bring remote sub-results to the computation. This version is not optimal, but nonetheless performs dramatically better than a version that uses only migration, which had a speedup of 0.47 on 32 processors.

5.2.1 Other Performance Issues. Two benchmarks, MST and Barnes-Hut, while obtaining speedups, demonstrated possibilities for improvement. In both cases, the programs would benefit from more inexpensive synchronization mechanisms. For MST, time spent in synchronization caused poor performance. MST computes the minimum spanning tree of a graph using Bentley's algorithm [8]. 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 noted earlier, Barnes-Hut uses an oct-tree that is rebuilt every iteration. In our implementation, the tree is built sequentially, because Olden lacks the synchronization mechanisms necessary allow multiple updates to a data structure to occur in parallel. As the cost of computing the forces on the particles, which is done in parallel, decreases the time spent on the tree-building phase becomes significant.

5.3 Summary

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.

6. 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 computation or the layout of the data.

Olden provides profiling tools that allow the programmer to check the local path lengths of the fields of a 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 used, the compiler will generate

code to record the number of communication events corresponding to each line of the program. In this section, we discuss how local path lengths are verified. Olden's event profiler is relatively straightforward and is discussed in Carlisle's thesis [14].

6.1 Verifying Local Path Lengths

Recall from Sect. 4 that the local path length of a field of a structure is a hint that gives information about the expected number of nodes that will be made local by a migration following a traversal of a given field of the structure. In this section, we 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.[7] 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.

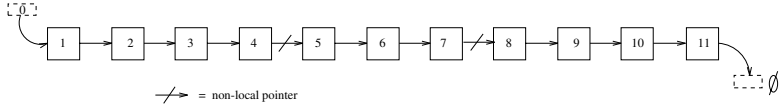

Fig. 6.1. A sample linked list

For an example of computing local path lengths, consider the data structure in Fig. 6.1. For the `next` field of this list, there are two non-local pointers, $[v_4, v_5]$ and $[v_7, v_8]$. 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

[7] This differs from the normal notion of path length (see, for example, [43]). The length of a local path expresses the number of vertices in the path that have the same home.

length would be $\frac{11}{3}$. For simplicity, we will assume for the remainder of this section that traversals begin on the owner of the root; however, the profiler considers initial pointers when computing local path lengths.

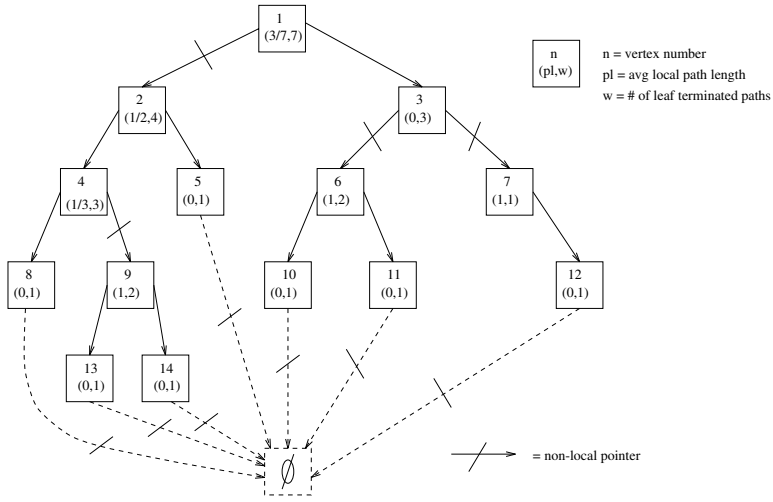

Fig. 6.2. A sample tree with local-path-length information

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 sub-path. This corresponds to assuming that all traversals are equally likely. Returning to the linked list example, each maximal local path is a sub-path 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 Fig. 6.2. 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 right field is 2, and the local path length of the left 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. If a non-local pointer is encountered during the traversal, we increment the total weight and weighted sum for that field. 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 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 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. 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.

We used the profiler to compute 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 the values computed by the profiler 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.

The traversal of each sub-result of TSP has high locality, and thus we can reduce the communication substantially by using migration to traverse these sub-results. Using the profiler, we see that the run-time-computed local path length of the relevant field is 100, which is representative of this locality. By changing to the computed local path length value, we increase the speedup

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

Fig. 6.3. Linked list traversal in Health

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.

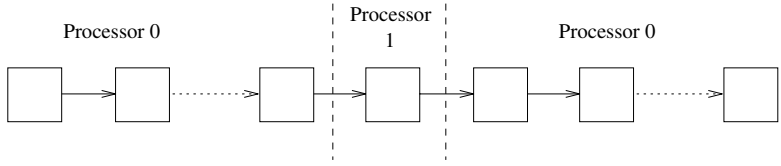

Fig. 6.4. Example linked list

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 Fig. 6.3. 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 Fig. 6.4. 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, as opposed to two migrations using computation migration. Since the lists in Health resemble the list in Fig. 6.4, 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 version, and 2.27 seconds using the run-time-computed local path lengths.

7. Related Work

Much work has been done on providing support for programming parallel machines. In this section, we describe how our work relates to that of other groups. First, we discuss some work by Rajiv Gupta that motivated our work on Olden. Then we divide other projects 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. Figure 7.1 summarizes the differences between Olden and the other systems described in this section.

	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

Fig. 7.1. Summary of related work

7.1 Gupta's Work

Our work on Olden was motivated originally by Rajiv Gupta's work supporting dynamic data structures on distributed memory machines [25]. His approach was to create global names for the nodes in a data structure and then to apply a variant of run-time resolution [48] to the program. His naming scheme assigns a name to every node in a data structure as it is added to the structure and makes this name known to all processors (thereby producing a global name for the node). The name assigned to a node is determined by

its position in the structure, as is the mapping of nodes to processors. For 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.

This method 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). Also, 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.

Gupta's decision to use a variant of run-time resolution to handle non-local references also has several problems, which are caused by the ownership tests that are done to allocate work. First, the overhead of these tests is high. And second, they prevent many important optimizations, such as vectorization and pipelining, from being applied. Run-time resolution was never intended to be a primary method of allocating work, instead it was designed as the fall back position for compile-time resolution, which resolves who will do the work at compile time rather than at run time. In Olden, we take a much more dynamic approach to allocating work. We still test most references to determine where the work should be done, but each test is done by at most one processor and in general the other processors will be busy with their own work.

7.2 Object-Oriented Systems

Emerald [32] 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 [17] 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 [16] and Mercury [22] 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 [18, 33] 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 in-lined, 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 [44].

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 in-lining; consequently, their use of objects adds additional overhead.

7.3 Extensions of C with Fork-Join Parallelism

Cid [42], 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 in-lined, the system cannot change its mind. In Olden, an in-lined 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 [10, 11] is another parallel extension of C implemented using a pre-processor. 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 in-lined.

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.

7.4 Other Related Work

Split-C [20] 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. [39] 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 [28] 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 [29], 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 occurred 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 [5, 6] 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 [15] 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 [31] 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.

8. Conclusions

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.

Acknowledgement. Laurie Hendren of McGill University and John Reppy of AT&T Labs contributed to the early design of Olden. Both authors were members of the Department of Computer Science at Princeton University when this work was done. Martin Carlisle was supported by the Fannie and John Hertz Foundation, a National Science Foundation Graduate Fellowship, NSF Grant ASC-9110766, NSF FAW award MIP-9023542, and the Department of Computer Science at Princeton University. Anne Rogers was supported, in part, by NSF Grant ASC-9110766. Also, the National Supercomputing Centers at the University of Illinois (NCSA) and Syracuse University (NPAC) provided access to their Think Machines CM-5s.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1988.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, December 1986.
8. J. Bentley. A parallel algorithm for constructing minimum spanning trees. *Journal of Algorithms*, 1:51–59, 1980.
9. 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.
10. 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`.
11. 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.
12. D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2(2):151–169, October 1988.
13. M. Carlisle and A. Rogers. Software caching and computation migration in Olden'. *Journal of Parallel and Distributed Computing*, 38(2):248–255, November 1996.
14. Martin C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University Department of Computer Science, June 1996.
15. 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.
16. 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.
17. 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.
18. 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.
19. Cray Research, Inc. *Cray T3D System Architecture*. 1993.
20. 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.
21. 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.
22. 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.
23. M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, 1990.
24. L. Guibas and J. Stolfi. General subdivisions and voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

25. 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.
26. R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
27. 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.
28. 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.
29. Wilson C. Hsieh. *Dynamic Computation Migration in Distributed Shared Memory Systems*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, September 1995.
30. 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.
31. Yvon Jégou. Exécution de codes irrégulier par migration de tâches. Technical Report 867, Institut de Recherche en Informatique et Systèmes Aléatoires, November 1994.
32. 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.
33. 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.
34. A. Karp. Programming for parallelism. In *IEEE Computer*, May 1987.
35. R. Karp. Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane. *Mathematics of Operations Research*, 2(3):209–224, August 1977.
36. C. Koelbel. *Compiling Programs for Nonshared Memory Machines*. PhD thesis, Purdue University, West Lafayette, Ind, August 1990.
37. L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), September 1979.
38. 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.
39. 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.
40. 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.
41. 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.

42. 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.
43. C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.
44. 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.
45. John A. Rice. *Mathematical Statistics and Data Analysis*. Wadsworth and Brooks/Cole Advanced Books and Software, Pacific Grove, CA, 1987.
46. 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.
47. 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.
48. 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.
49. 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.
50. 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.
51. Michael Shneier. Calculations of geometric properties using quadtrees. *Computer Graphics and Image Processing*, 16(3):296–302, July 1981.
52. H. Zima, H. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6(1):1–18, 1988.