To appear in ACM Transactions on Programming Languages and Systems

Supporting Dynamic Data Structures on Distributed Memory Machines

ANNE ROGERS and MARTIN C. CARLISLE
Princeton University
JOHN H. REPPY
AT&T Bell Laboratories
and
LAURIE J. HENDREN
McGill University

Compiling for distributed memory machines has been a very active research area in recent years. Much of this work has concentrated on programs that use arrays as their primary data structures. To date, little work has been done to address the problem of supporting programs that use pointer-based dynamic data structures. The techniques developed for supporting SPMD execution of array-based programs rely on the fact that arrays are statically defined and directly addressable. Recursive data structures do not have these properties, so new techniques must be developed. In this paper, we describe an execution model for supporting programs that use pointer-based dynamic data structures. This model uses a simple mechanism for migrating a thread of control based on the layout of heap-allocated data and introduces parallelism using a technique based on futures and lazy task creation. We intend to exploit this execution model using compiler analyses and automatic parallelization techniques. We have implemented a prototype system, which we call *Olden*, that runs on the Intel iPSC/860 and the Thinking Machines CM-5. We discuss our implementation and report on experiments with five benchmarks.

Categories and Subject Descriptors: D.1.3 [**Software**]: Concurrent Programming—*Parallel Programming*; D.3.4 [**Software**]: Processors—*Compilers*; *Run-time Environments*

General Terms: Languages, Performance

Additional Key Words and Phrases: Dynamic data structures

A. Rogers was supported, in part, by NSF Grant ASC-9110766. M. C. Carlisle was supported, in part, by a National Science Foundation Graduate Fellowship, the Fannie and John Hertz Foundation, and NSF Grant ASC-9110766. L. J. Hendren was supported, in part, by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.
Authors' addresses: A. Rogers and M. C. Carlisle, Department of Computer Science, Princeton University, Princeton, NJ 08540. email: {amr, mcc}@cs.princeton.edu; J. H. Reppy, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974. email: jhr@research.att.com; L. J. Hendren, School of Computer Science, McGill University, Montreal, Que, Canada, H3A 2A7. email: hendren@cs.mcgill.ca.

1. INTRODUCTION

Compiling for distributed memory machines has been a very active area of research in recent years.[1] Much of this work has concentrated on scientific programs that use arrays as their primary data structure and loops as their primary control structure. Such programs tend to have the property that the arrays can be partitioned into relatively independent pieces and therefore the operations performed on these pieces can proceed in parallel. It is this property of scientific programs that has led to impressive results in the development of vectorizing and parallelizing compilers [Allen et al. 1988; Allen and Kennedy 1987; Wolfe 1989]. More recently, this property has been exploited by researchers investigating methods for automatically generating parallel programs for SPMD (Single-Program, Multiple-Data) execution on distributed memory machines. In this paper, we address the problem of supporting programs that operate on recursively-defined dynamic data structures. Such programs typically use list-like or tree-like data structures, and have recursive procedures as their primary control structure.

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

From a compilation standpoint, the most important property of a distributed memory machine is that each processor has its own address space; remote references are satisfied through explicitly passed messages, which are expensive. Therefore, arranging a computation so that most references are local is crucial to efficient execution. The aforementioned properties of scientific programs make them ideal applications for distributed memory machines. Each group of related data can be placed on a separate processor, which allows operations on independent groups to be done in parallel with little interprocessor communication.

The key insight underlying recently developed methods for automatically parallelizing programs for distributed memory machines is that the layout of a program's data should determine how the work in the program is assigned to processors. Typically, the programmer specifies a *mapping* of the program's data onto the target machine and the compiler uses this mapping to decompose the program into processes. The simplest compilation strategy, sometimes called *runtime resolution*, inserts code to determine at runtime which processor needs to execute a particular statement. Different policies for allocating work are possible, but the most popular is the *owner computes rule*: the work of an assignment statement (`v = e`), including the computation of `e`, is assigned to the processor that "owns" `v`. Control statements such as conditionals and loops are executed by all processors. The code produced by this method can be improved substantially using the arsenal of techniques developed for vectorizing compilers, such as data dependence analysis and loop restructuring [Allen and Kennedy 1987; Wolfe 1989].

---

[1] For example see, [Amarasinghe and Lam 1993; Anderson and Lam 1993; Callahan and Kennedy 1988; Gerndt 1990; Hiranandani et al. 1991; Koelbel 1990; Rogers and Pingali 1989; Zima et al. 1988].

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

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

A recent paper by Gupta [1992] suggests a mechanism for addressing the problem of global names so that an approach similar to runtime resolution can be used. In his approach, a global name is assigned to every element of a dynamic data structure and this name is made known to all processors. To accomplish this, a name is assigned to each node as it is added to a data structure. This name is determined by the node's position in the structure and is registered with all processors as part of adding it to the structure. The mapping of a node to a processor is also based on its position in the tree. As an example, a breadth-first numbering of the nodes might be used as a naming scheme for a binary tree. Once a processor has a name for the nodes in a data structure, it can traverse the structure without further communication.

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

As in earlier approaches, we rely on the programmer to specify a mapping of the data onto the processors. This mapping is done at runtime by specifying a processor name with every memory allocation request. They key difference between our approach and earlier approaches is the mechanism for assigning work to processors. We believe that runtime resolution, which was developed to support statically-defined, directly-addressable, rectangular arrays, is inappropriate for pointer-based dynamic data structures, which are neither statically defined nor directly addressable. We propose a more dynamic approach that is better matched to the dynamic nature of the data structures themselves. Rather than making each processor decide if it owns the data, we send the computation to the processors that own the data. Thus, as a dynamic structure is recursively traversed, the computation migrates to the processor that owns that part of the structure.

Before presenting our execution model, we review the basic SPMD model and our programming model. In Section 3, we present the two parts of our execution model: a simple mechanism for migrating a thread of control based on the layout of heap-allocated data and a technique for introducing parallelism based on futures and lazy task creation [Mohr et al. 1991]. We consider an example in Section 4 and discuss our prototype system, which we call *Olden*, in Section 5. In Section 6, we report results for five benchmarks using implementations of the execution model for the Intel iPSC/860 and the Thinking Machines CM-5. In Section 7, we briefly sketch the compiler analyses and parallelization techniques that we intend to use to transform a sequential program automatically into a program that can take advantage of our execution model. Finally, we discuss related work in Section 8 and conclusions in Section 9. This paper expands upon our earlier work [Rogers et al. 1992; Carlisle et al. 1993].

2. THE SPMD MODEL

Before we explain the details of our approach, let us review the basic SPMD and programming models that we are using.

In our SPMD model, each processor has an identical copy of the program, as well as a local stack that is used to store procedure arguments, local variables, and return addresses. In addition to these local stacks, there is a distributed heap; each processor owns part of the heap. We view a heap address as consisting of a pair of a processor name and a local address (<p, l>). This information is encoded as a single machine address.

Our programming model is based on a restricted form of C. 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 there are no pointers into the processor stacks. The major difference between our programming model and the standard sequential model is that the programmer explicitly chooses a particular strategy to map the dynamic data structures over the distributed heap. This mapping is achieved by including a processor number in each allocation request. Figure 1 contains an example allocation routine that builds a tree such that the sub-trees at some fixed depth are equally distributed over all processors. `ALLOC` is a library routine that allocates space on a specified processor. `ALLOC`'s result is a pointer that encodes both the processor name and the local address on that processor where the requested space has been allocated. Figure 2 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.

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

/* Allocate a tree with level levels on processors lo..lo+num_proc-1 */

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. 1. Allocation code

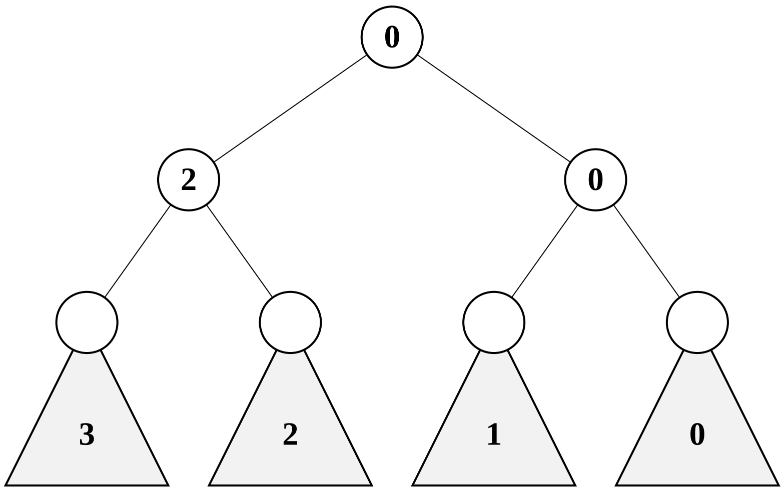

Fig. 2. Allocation example

3. EXECUTION MODEL

This section describes the two parts of our execution model: thread migration and thread splitting. Thread migration is a mechanism for migrating a thread of control through a set of processors based on the layout of heap-allocated data. Thread splitting, which is based on continuation capturing operations, is used to introduce parallelism by providing work for a processor when the thread it is executing migrates.

3.1 Thread migration

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

Recall that we view a memory address as consisting of a pair of a processor name and a local address. This information can be encoded as a single address, and the address translation hardware can be used to detect non-local references [Appel and Li 1991]. When Processor P executes a load or store instruction that refers to Q's memory, the instruction traps to a library routine. The trap handler is responsible for packaging up the thread's state information and sending it to Processor Q. Notice that if the entire stack is sent with the thread, stack references will always be local and cannot cause a migration. In our current implementation, we do not use the address translation hardware to detect non-local accesses. Instead, the compiler inserts explicit checks into the code to test the encoded pointer and migrate the thread as needed.

Full thread migration is quite expensive, because the thread's entire stack is included in the message. To make thread 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. When it is time to return from the procedure, it is necessary to return control to Processor P, because it has the stack frame of the caller. To accomplish this, Q sets up a stack frame for a special return stub to be used in place of the return to the caller. This frame holds the return address and the return frame pointer for the currently executing function. The stub migrates the thread of computation back to P by sending a message that contains the return address, the return frame pointer, and the contents of the registers. Processor P then completes the procedure return by restarting the thread at the return address. Note that the stack frame is not returned, because it is no longer needed.

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

3.1.2 *Discussion.* The primary purpose of migrating the thread of computation to the processor that owns the data rather than migrating the data is to exploit locality. Our system relies on the programmer to choose a layout that places related data together. For example, by placing all the data in a subtree at some fixed depth in the tree on the same processor. If the programmer has chosen a good data layout, then migrating the computation to the data allows the system to perform the computation on the processor that owns most of the required data. This can reduce the amount of interprocessor communication substantially and thereby reduce execution time. Recent work by Hsieh, Wang, and Weihl [1993] supports this claim.

3.2 Thread Splitting

While the migration scheme provides a mechanism for operating on distributed data, it does not provide a mechanism for extracting parallelism from the computation. When a thread 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. This capturing technique effectively 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 analysis of the program, which can be done either by a parallelizing compiler targeted for Olden or by a programmer using Olden directly.

3.2.1 *Futures.* Our continuation capturing mechanism is essentially a variant of the *future* mechanism found in many parallel Lisps [Halstead, Jr. 1985]. 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, Kranz, and Halstead [1991], is to save the future call's context (return continuation) on a work list and to evaluate the future's body directly.[2] If a migration occurs in the execution of the body, then we grab a continuation from the work list and start executing it; this is called *future stealing*. In addition, all touches in Olden are done explicitly using the `touch` operation.

3.2.2 *Futures in more detail.* Our scheme for introducing parallelism consists of two operations: `futurecall` and `touch`. In the remainder of this section, we describe these operations and their related data structures. The two data structures are the future cells and the future stack, which is a stack of future cells that serves

[2] This is also similar to the workcrews paradigm proposed by Roberts and Vandevoorde [1989].

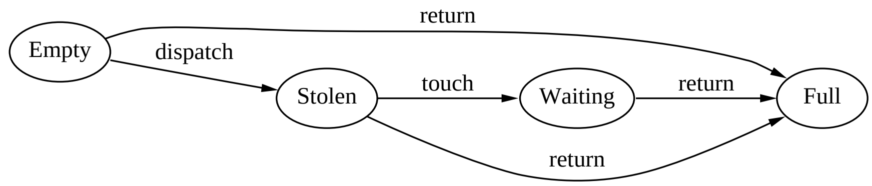

Fig. 3. Future cell state transitions

as a work list. A future cell can be in one of four states (Figure 3 gives the allowable state transitions for a future cell):

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

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

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

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

The future stack is threaded through the execution stack. In the normal case, when no migration or stealing occurs, only a few extra instructions are required to allocate the future cell and touch the result (see Section 5 for details).

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

When a processor has no work to do, such as just after a thread migration, it calls the runtime system routine `MsgDispatch`. This routine does the actual stealing of continuations from the future stack. If the stack is not empty, it pops a future cell from the top of the stack and executes the resume continuation in the cell. To signal that the parent continuation has been stolen, the state of the future cell is changed to **Stolen**.

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

3.2.3 *Discussion.* The design of futures in Olden was influenced heavily by our intent to use parallelizing compiler technology [Hendren 1990; Hendren and Nicolau 1990; Hendren et al. 1992; Larus 1991] to insert them automatically. Both the restrictions mentioned in the previous paragraph follow from this intention. These two restrictions allow us to allocate future cells on the stack rather than the heap, which helps make futures less expensive.

Another important fact to note about the use of futures in Olden is that when a process becomes idle, it steals work only from its own worklist. A process never removes work from the worklist of another process. The motivation for this design decision was our expectation that most futures captured by a process would operate on local data. Although allowing another process to steal this work may seem desirable for load balancing purposes, it would simply cause unnecessary migrations. 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 work from the tail of another process's work queue.

The choice of using a queue versus a stack to hold pending futures is related to the question of who can steal work from a processor's worklist. The breadth-first decomposition that results from using a work queue is desirable when work can be stolen by another process, because it tends to provide large granularity tasks. A stack-based implementation is appropriate for Olden, because when a process steals its own work, it is better for it to steal work that is needed sooner rather than later.

4. A SIMPLE EXAMPLE

To make the ideas of the previous two sections more concrete, we present a simple example.[3] Figure 4 gives the C code for a prototypical divide-and-conquer program, which computes the sum of the values stored in a binary tree.

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

Fig. 4. `TreeAdd`

Figure 5 gives a semantically equivalent program that has been annotated with futures. In the annotated version, the left recursive call to `TreeAdd` has been replaced by a `futurecall`,[4] and the result is not demanded until after the right recursive call.

To understand what this means in terms of the program's execution, consider the case where a tree node t_P (on Processor P) has a left child t_Q (on Processor

[3] We present more realistic benchmarks in Section 6.

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

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

        t_left = t->left;              /* this can cause a migration */
        f_left = futurecall (TreeAdd, t_left);

        sum_right = TreeAdd (t->right);

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

Fig. 5. `TreeAdd` code with compiler annotations

Q). When `TreeAdd`, executing on P, is recursively called on t_Q, it attempts to fetch the left child of t_Q, which causes a trap to the runtime system. This will cause the thread of control to migrate to Processor Q, where it can continue executing. Meanwhile, execution resumes on Processor P at the return point of the `futurecall`. Assuming that the right subtree of t_P is on Processor P, execution of the stolen future continues until it attempts to touch the future cell associated with the call on t_Q. At this point, execution must wait for the thread of control to return from Processor Q.

Please note that neither the reference to `t->right` nor the reference to `t->val` can 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.

5. EXECUTION MODEL IMPLEMENTATION

We have implemented prototypes of our execution model that run on the Intel iPSC/860 hypercube and the Thinking Machines CM-5. As mentioned earlier, we call our system *Olden*. The input to Olden is a C program annotated with `futurecall`s and `touch`es. Olden consists of a runtime system and a compiler that translates annotated C code into the assembly code for the target machine with embedded calls to the runtime system. The runtime system handles migrations, returns from migrations, and starting new computations by stealing futures. The compiler generates code for futures, touches, and pointer tests, and it uses a non-standard stack discipline to support multiple active threads of computation. We describe the two parts of the system in detail in this section.

5.1 The Runtime System

Figure 6 provides an overview of the control structure of the runtime system. The main procedure is `MsgDispatch`, which is responsible for assigning work to a pro-

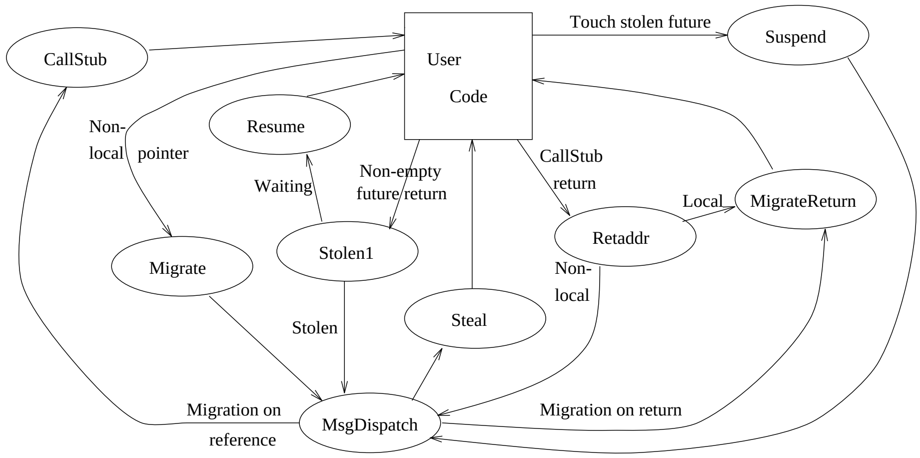

Fig. 6. The Runtime System

cessor when it becomes idle. Captured empty futures are given first priority, then messages from other processors in the order received. We choose this priority scheme, because early in a computation, a future is likely to spin off work to other processors. Processing futures first allows us to get work to idle processors sooner.

The runtime system processes four different events: migrate on reference, migrate on return, steal future, and suspend touch. We focus on migration first; we discuss the implementation of futures in the next subsection.

5.1.1 *Migrate on Reference.* As previously mentioned, a thread will migrate if it attempts to dereference a non-local pointer. The code to test the pointer and call `Migrate`, if necessary, is inserted by the compiler. `Migrate` packages the necessary data and sends it to the appropriate processor. A migration message contains the current stack frame, the argument build area of the caller, the contents of callee-save registers,[5] and some bookkeeping information. We allocate extra space in each stack frame for the callee-save registers and bookkeeping information, which allows us to construct messages *in situ*, reducing the cost of migrations. Once the migration is complete, the sending processor calls `MsgDispatch` to assign a new task.

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

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

the migrated function on the new processor. When this procedure exits, it will go to the procedure `retaddr`. This code stores the values of the callee-save registers in the return message, sends the message to the processor that began execution of the procedure, and calls `MsgDispatch`.

A simple optimization is added to avoid a chain of trivial returns in the case that a thread migrates several times during the course of executing a single function. `Migrate` examines the current return address of the function to determine whether it points to the return stub. If so, the original return address, frame pointer, and node id are pulled from the stub's frame and passed as part of the migration message. This is analogous to a tail-call optimization and is similar to the *tail forwarding* optimization used in the Concurrent Smalltalk compiler [Horwat et al. 1989]. A similar optimization is that the return message is not sent if a migrated procedure exits on the same processor as it began (for example, it migrated from P to Q then back to P): `MigrateReturn` is called directly instead.

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

5.2 The Compiler

The compiler is derived from **lcc** [Fraser and Hanson 1995], an ANSI C compiler.[6] We ported the backend to the appropriate processors (i860 and SPARC) and added modifications to handle our execution model. These modifications generate code to test for non-local memory references and call `Migrate` as necessary, to capture futures, and to synchronize on touches.

5.2.1 *Testing Memory References.* As previously mentioned, a process will migrate on a reference if it attempts to dereference a non-local heap pointer. In our implementation, heap pointers consist of a tag indicating the processor where the data resides[7] and a local address, so it is necessary to check the tag explicitly on each heap pointer dereference. **lcc** has a command-line option for generating null pointer checks on all pointer dereferences. We modified this mechanism to generate code that examines the tag to see if the pointer is local, conditionally calls the library routine `Migrate` to move the thread, and removes the tag to reveal the local address.

The additional overhead of testing each pointer dereference is three integer instructions and a conditional branch on the i860 and four integer instructions and a conditional branch on the SPARC. This overhead can be reduced by observing that only the first reference in a sequence of references to the same pointer can cause a

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

[7] The range of possible heap addresses dictates the maximum size of the processor tags. The CM-5 implementation uses the seven high order bits to hold the tag and the iPSC/860 implementation uses the eight high order bits.

migration. For example, in `TreeAdd`, only the reference `t->left` can cause a migration (as discussed in Section 4). All subsequent indirect references through `t` are guaranteed to be local.[8] The Olden compiler provides an annotation, `local`, which allows allows us to take advantage of this observation. The expression `local(`t`)` removes the pointer tag and forces a migration if t is non-local. Subsequent indirect references can use the resulting local pointer.

A simple compiler optimization can take advantage of this construct. We say that a dereference of some pointer variable t is *aligned* with another dereference of t, if it is *dominated* by that reference and if there are no intervening references to other pointers. A dominating reference, that is the first reference in the sequence, can be annotated with `local`. The resulting local address can be used by the aligned references directly. The code in Figure 7 shows the result of applying this optimization to the `TreeAdd` example. Note that the deferences of `local_t` require no special handling.

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

        local_t = local(t);      /* this can cause a migration */
        t_left = local_t->left;
        f_left = futurecall (TreeAdd, t_left);

        sum_right = TreeAdd (local_t->right);

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

Fig. 7. `TreeAdd` code using `local`

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

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

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

check the state of the future. If it is **Empty**, the cell is popped from the stack, and execution continues.

In the case that the cell is not **Empty** (that is, the called function or one of its descendants migrated), we test to see whether the future is **Stolen** or **Waiting**. If it is **Stolen**, then `MsgDispatch` is called to assign work to the processor. If it is **Waiting**, there must have been a touch on this future cell. We describe how to resume a waiting thread of computation in the next subsection.

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

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

To implement register unwinding, we generate a callee-save restore sequence for each procedure, and store its address at each call site in the procedure. Then, when stealing a future, we traverse the list of frame pointers to the frame that contains the future cell, executing each callee-save restore sequence as we go. Since the address of a restore sequence is stored at the appropriate call site, it can be obtained by reading a word at a constant offset from the return address stored in the frame. For example, consider the scenario show in Figure 8. When `H` migrates, the runtime system will execute the callee-save restore sequence for `H`, and then `G`. Once this is done, the callee-save registers will contain their original values.

A second optimization is to eliminate the need for storing the state **Empty** in the future cell. The return address for a future call is initially set to the sequence of instructions for an **Empty** future cell. When a future cell is stolen, the return address of the called function is modified to point to code which will test for **Stolen** or **Waiting**, and perform the appropriate action.

Given these optimizations, the overhead of a future call on the i860 is only seven instructions (four stores, two integer instructions, and one load) in the expected case where the future cell is not stolen.

On the CM-5, we have implemented two additional optimizations. The first allows us to eliminate the store of the frame pointer on entry. We notice that since the future cell is stored in the frame of the caller, we can determine the frame pointer during register unwinding. The frame pointer of the parent continuation is simply the first frame pointer in the chain with an address that precedes the address of the future cell (see Figure 8).

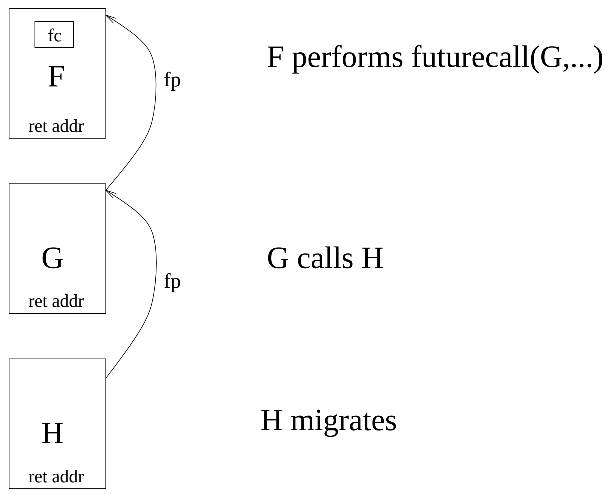

Fig. 8. Register Unwinding Example

The second takes advantage of the facts that the future stack is threaded through the execution stack using a linked list and that future cells are four-byte aligned, to eliminate another store. We store the state information in the last two bits of the future cell's link field. By representing the **Empty** and **Full** states as zero, we get the state information stored for free when we set the link field (because it contains a 4-byte aligned pointer). Then, on a steal the link field is no longer necessary, and we may reuse the bottom two bits to hold state information.

Given these optimizations, a `futurecall` on the CM-5 requires only five instructions (two stores, two integer instructions, and one load) in the case when the future is not stolen. Appendix A contains sample `futurecall` code for both the iPSC/860 and CM-5 implementations.

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

5.3 Stack Management

In the discussion thus far, we have glossed over certain details related to stack management. When a future is stolen, the portion of the stack between the frame belonging to the stolen continuation and the frame that migrated must be preserved for when the migrated thread returns. The stolen continuation may allocate new stack frames, which could overwrite the frames of the migrated thread if we used a simple stack management policy. To avoid this problem, we adopt a simplification of a single stack technique for multiple environments by Bobrow and Wegbreit [1973].

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

Since our model does not require more than one extension per frame, we can simplify the method and reduce the overhead on function calls and returns. We maintain a single stack, called a *simplified spaghetti stack*, that contains the interleaved frames of the continuations.[9] In a simplified spaghetti stack, new stack frames are allocated off the global stack pointer. We maintain the invariant that *the global stack pointer always marks the end of a live frame.* The exit routine for a frame is split into two parts: deactivation and deallocation. If a procedure whose frame is in the middle of the stack exits, it is deactivated by marking it as garbage. If a frame at the end of the stack exits, the global stack pointer is adjusted, thus deactivating and deallocating the frame simultaneously. Additionally, the stack pointer is adjusted to the end of the live frame nearest the end of the stack, which deallocates any garbage frames immediately below it.

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

Figure 9 provides an example of this process. Assume that initially the procedure F is executing. When F calls G, a new frame is allocated at the end of the stack. Notice that there are live frames between F and the end of the stack that belong to another thread of execution. When G exits, since the preceding frame (e) is live, the stack returns to the initial state. Figure 10 provides a more complex example. Again, F calls G. G migrates, and the runtime system gives control to e. The frame e is not at the end of the stack; hence, when it exits, a pointer is placed at the end of its frame marking it as garbage. The exit for d is similar. When G resumes execution and exits, its frame is deactivated. Since it is at the end of the stack, the list of garbage frames will be followed until a null pointer is reached (in the frame labelled F), thus deallocating all of the garbage frames.

[9]A similar method, called a meshed stack, was developed by Hogen and Loogen in the context of parallel implementations of functional or logic languages [Hogen and Loogen 1993].

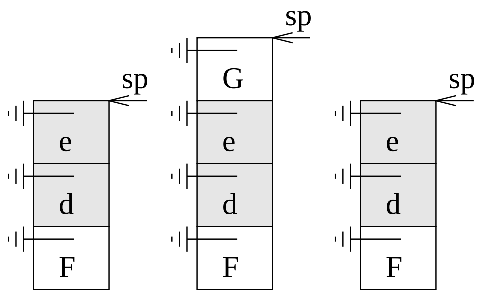

Fig. 9. Simplified Spaghetti Stack — simple case

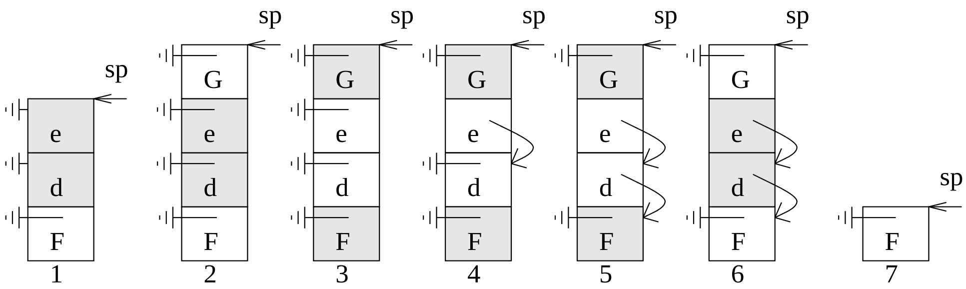

Fig. 10. Simplified Spaghetti Stack — complex case

The overhead resulting from our simplified spaghetti stacks is low. One extra instruction is required on entry to store zero at splink. The expected exit path (topmost frame, with live frame underneath), requires four additional instructions (an add, one load, and two conditional branch instructions) over the basic stack exit code for the i860 and six additional instructions (three adds, one load, and two conditional branches) for the CM-5.

On the CM-5, we have implemented an additional optimization, called *express checkout*, that maintains a bit that indicates that the current stack frame is at the end of the stack with a live frame below it (the common case). This optimization reduces the cost of the stack exit sequence from six instructions to three instructions.

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

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

6. RESULTS

The previous section described our prototype implementations of Olden for the Intel iPSC/860 and the Thinking Machines CM-5. In this section, we report results for five benchmarks: TreeAdd, Power, Bitonic Sort, Traveling Salesman, and Voronoi Diagram. TreeAdd is very simple and has ideal locality. Power has a similar structure, but is more computationally intensive. Bitonic Sort is a relatively complex algorithm that has two-levels of recursion and complex locality patterns. We experimented with two versions of Bitonic Sort, which we call Original Bitonic Sort and Bitonic Sort. The difference between the two is an extra call to malloc in Original Bitonic Sort. Traveling Salesman is a classic divide-and-conquer algorithm. And finally, Voronoi Diagram is classic divide-and-conquer algorithm that uses a different method for merging subresults. We include it because illustrates a problem with our model. All of these benchmarks use trees as their primary data structures.

We performed our experiments on a 16-node iPSC/860 at Princeton and on CM-5s at two National Supercomputing Centers: NPAC at Syracuse University and NCSA at the University of Illinois. The raw data from the experiments is given in Tables I and II. The timings given for the iPSC/860 are from single runs. Timings on this machine do not vary significantly between runs. The timings reported for the CM-5 are averages over three runs done in dedicated mode. Each benchmark includes a timing for a sequential implementation (*sequential*), which was compiled using our compiler, but without the overhead of futures, pointer testing, or the spaghetti stack, and a one processor implementation (*one*) which includes this overhead.

The benchmarks were hand-annotated to include future calls, touches, and a variant of the local annotation. The programs were annotated aggressively. Our goal was to show that the execution model allows efficient parallelization. We do not claim that the annotated programs could be generated automatically using existing compiler technology.

6.1 TreeAdd

TreeAdd is a very simple program that recursively walks a tree, summing each subtree (see Figure 7). We report results for a balanced, binary tree of size 1024K in Figure 11. The tree is allocated using the allocation scheme presented in Figure

Table I. iPSC/860 timings in seconds

Benchmarks	sequential	one	Processors 2	4	8	16
TreeAdd (1024K)	1.94	2.62	1.31	0.66	0.33	0.17
Power (10,000)	517.28	517.46	260.23	130.43	65.88	34.20
Original Bitonic Sort (32 K)	4.87	5.69	3.23	2.29	2.04	2.09
Original Bitonic Sort (128 K)	29.21	64.67	15.18	9.38	6.69	5.36
Bitonic Sort (32 K)	3.60	4.45	2.64	2.01	1.91	2.02
Bitonic Sort (128 K)	17.46	21.21	11.98	7.96	6.21	5.05
Traveling Salesman (32,767)	65.79	66.02	33.49	17.48	9.61	5.77
Voronoi Diagram (64K)	13.60	20.17	13.06	11.35	16.00	33.16

Table II. CM-5 timings in seconds

Benchmarks	sequential	one	Processors 2	4	8	16	32
TreeAdd (1024K)	4.49	5.94	2.99	1.50	0.75	0.37	0.19
Power (10,000)	286.59	313.04	160.75	79.73	43.87	25.03	11.87
Original Bitonic Sort (32 K)	8.50	10.79	5.80	3.49	2.40	1.88	1.67
Original Bitonic Sort (128 K)	40.66	52.56	27.15	15.72	9.86	6.65	4.89
Bitonic Sort (32 K)	6.55	8.48	4.93	2.90	2.10	1.75	1.64
Bitonic Sort (128 K)	31.83	41.18	21.81	13.08	8.52	5.97	4.40
Traveling Salesman (32,767)	43.35	45.45	22.58	11.72	6.47	3.91	2.74
Voronoi Diagram (64K)	48.46	62.88	38.03	25.87	24.83	39.33	69.30

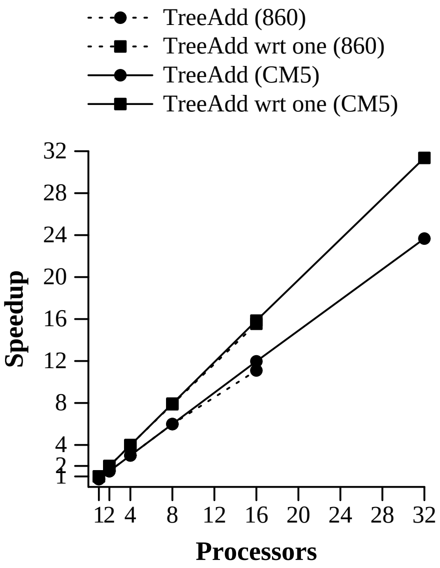

Fig. 11. `TreeAdd` performance

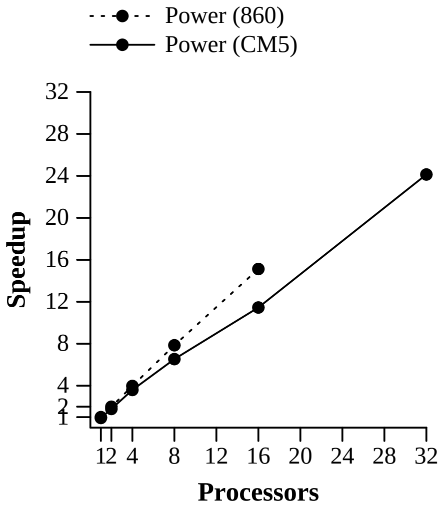

Fig. 12. `Power` performance

1. The subtrees at a fixed depth in the tree are equally distributed. The lower two curves plot the true speedup, that is, sequential time/parallel time. They represent parallel efficiencies of 72% for the i860 and 74% for the CM-5. The efficiencies are not perfect because there is very little real work over which to amortize the overhead of the futures, pointer tests, and the spaghetti stack. The upper curves plot speedup using the one processor implementation, rather than the sequential implementation, as the baseline. These curves display near-perfect linear speedup, because only $P-1$, where P is the number of processors, migrations occur. This demonstrates that the execution model can exploit available parallelism.

6.2 Power Pricing

Power solves the *Power System Optimization* problem, which can be stated as follows: given a power network represented by a tree with the power plant at the root and the customers at the leaves, use local information to determine the prices that will optimize the benefit to the community [Lumetta et al. 1993]. The computation executes a series of phases until convergence is reached. Each phase has a downward pass that propagates pricing information from the root to the customers and an upward pass that propagates demand information from the customers to the root. We use the input configuration from Lumetta et al. [1993]: one substation; ten main feeders from the root; each feeder branches to twenty lateral nodes; each lateral node is the head of a line of five branch nodes; and each branch has ten leaves. In all there are 1,201 internal nodes and 10,000 customers. The layout of the tree is determined in the following way: the branch nodes are divided evenly among the processors using a block distribution based on an in-order numbering of the tree. The leaves are distributed on the same processor as the corresponding branch, laterals are placed on the same processor as the head of their line of branches, and the root is placed on Processor 0.

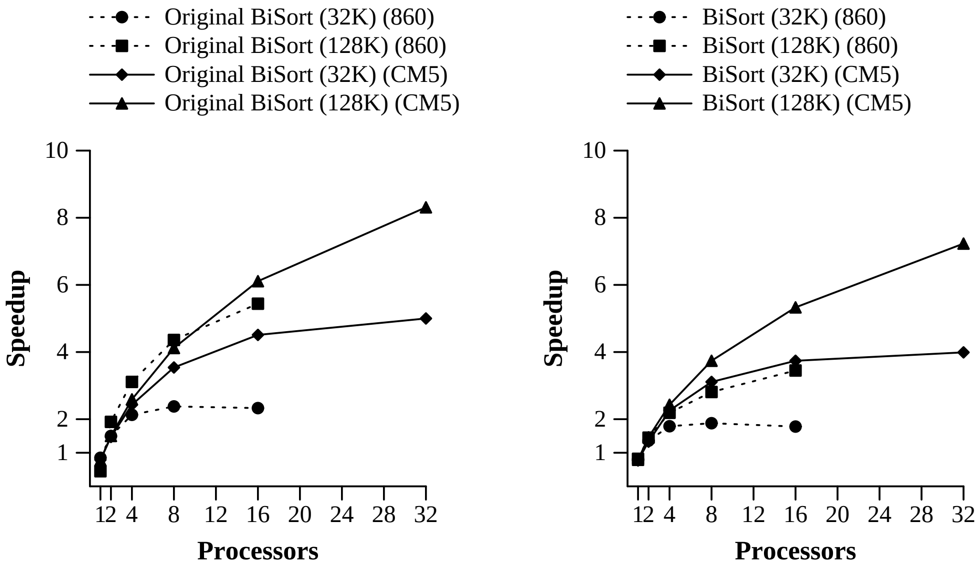

Fig. 13. Bitonic Sort performance

We report speedups for the whole program (including the time to build the tree) in Figure 12. Power's behavior is very similar to TreeAdd's, but on a much smaller tree. The main difference between them is that a substantial amount of work is done at the leaves of the tree in Power. Our results are comparable to Lumetta et al.'s results.

6.3 Bitonic Sort

Bitonic Sort [Bilardi and Nicolau 1989] allocates a random set of integers and performs two sorts on them: one forward and one backward. A binary tree is used to store the numbers. The algorithm first creates bitonic sequences in both subtrees, and then merges them to generate the sorted result. The merge phase swaps subtrees to create two disjoint bitonic sequences, and then performs two recursive calls on these sequences. The data has been placed on the processors such that the subtrees at a fixed depth are evenly distributed over the available processors, and the benchmark has been modified to maintain this layout following the sort. We report speedup only for the sorting phases to avoid having the expensive and easily parallelizable tree building phase skew the results. The graph in the left half of Figure 13 contains four curves: two different problem sizes (32K, 128K) for each implementation of Original Bitonic Sort. These implementations perform respectably on a medium sized problem (128K numbers), but they perform an unnecessary malloc for each recursive sort, which increases the amount of work per call. The figure in the right half of Figure 13 displays the curves for an improved Bitonic Sort that does not include the extra call to malloc. It still displays parallelism, but not as much as the original version.

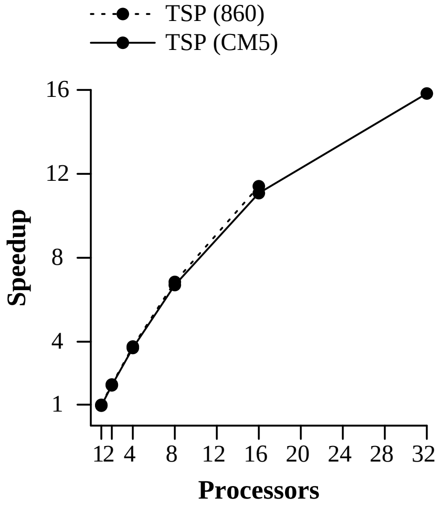

Fig. 14. `Traveling Salesman` performance

System overhead accounts for a loss of about 30% of the speedup. Even without this overhead, this benchmark would not show perfect speedup because the cost of moving the subtrees to maintain locality for the second sort is expensive.

6.4 Traveling Salesman

This benchmark computes an estimation of the best hamiltonian circuit for the Traveling-salesman problem. The program is based on an implementation of Karp's partitioning algorithm [Karp 1977] by Müller [Müller 1993]. In our implementation, however, we assume the cities are given in a tree that alternately divides a set of points by X or Y coordinate at each level. Rather than computing an exponential exact solution for small sizes, following Müller, we use the closest-point heuristic [Cormen et al. 1989].

The tree is laid out using the allocation scheme presented in Figure 1. We report results for computing a tour for 32,767 uniformly distributed points in Figure 14. These results do not include the cost of the tree building phase.

6.5 Building Voronoi Diagrams

The Voronoi Diagram benchmark [Guibas and Stolfi 1985; Lee and Schachter 1980] generates a random set of points and computes a Voronoi Diagram for these points. The points are stored in a balanced binary tree sorted by X coordinate. To compute a Voronoi Diagram, the algorithm recursively computes Voronoi Diagrams of the two subtrees (thereby dividing the set of points at the median), and merges them to form the final result. The merge phase walks along the convex hull of the two sub-diagrams, and adds edges to knit them together to form the Voronoi Diagram for the whole set. The points are assigned to processors so that the subtrees at a fixed depth are evenly distributed.

In Figure 15, we report the speedup obtained for building the Voronoi Diagram for 64K points. The reported speedup does not include the cost of generating

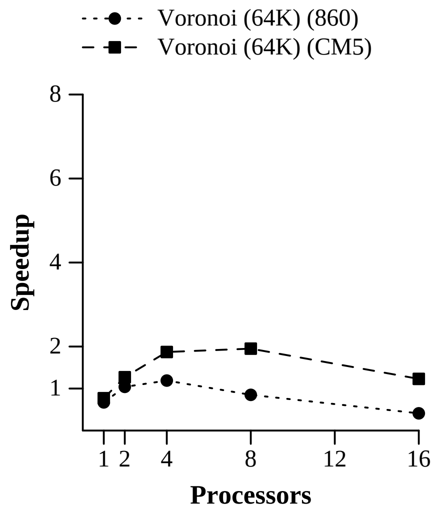

Fig. 15. `Voronoi Diagram` performance

the points or building the tree used to represent the sets of points. This example displays almost no parallelism for two reasons. First, the merge phase is sequential and represents a substantial fraction of the computation. But more importantly, since later merge phases alternately reference subsets on different processors, the available parallelism is swamped by the cost of the "ping-pong" migrations that occur during these merges. Migrations are expensive in our current implementation, because messages are expensive.[10]

6.6 Discussion and Future Work

In early experiments, we observed super-linear speedup for several of these benchmarks. We traced this to the quadratic behavior of the iPSC/860's memory allocation routines. The CM-5 suffers from a related problem: allocating memory from the operating system requires a global synchronization. We reduced the effect of these problems by calling malloc only a few times and managing the acquired storage explicitly.

The overhead of our technique includes the cost of pointer testing, capturing futures, and managing the spaghetti stack. We can measure the overhead for a benchmark by computing the difference between the sequential and one processor implementations. Pointer testing represents a significant component of this overhead. On the iPSC/860, pointer testing accounts for between 44% and 80% of the overhead. On the CM-5, it accounts for 24% to 80% of the overhead. It may be possible to reduce this overhead by using the address translation hardware and a user-level trap handler to detect and manage non-local references. The effectiveness of such a scheme will depend heavily on the cost of servicing a user-level trap [Appel and Li 1991]. We are currently experimenting with an implementation of Olden built on top of the Tempest/Blizzard system from the Wisconsin Wind Tunnel

[10] We have measured the cost of sending a 250 byte message, which is the average size of a migration message, as 496μs on the iPSC/860 and 546μs on the CM-5.

group [Schoinas et al. 1994], which provides a simple mechanism for registering user-level handlers that are invoked on a non-local reference.

Finally, the Voronoi Diagram benchmark points out that our model is not the final solution to the problem of parallelizing pointer-based programs for distributed memory machines. Our results indicate that in this case, and others, it may be better to use software caching rather than computation migration. The messages used for caching are much cheaper, and when data from multiple processors needs to be examined simultaneously, they can reduce communication costs substantially. We have implemented a simple user-invalidate cache using Active Messages [von Eicken et al. 1992] on the CM-5, and are exploring how this can be used to improve performance. Initial results have been quite encouraging. At present we are experimenting with compile-time heuristics that automatically choose between computation migration and software caching for each pointer dereference.

7. AUTOMATIC PARALLELIZATION

This paper has concentrated on Olden's execution model. As mentioned earlier, its design was influenced heavily by our intention to use it as the target for a parallelizing compiler. In this section, we briefly outline the compiler analyses that are needed to determine which sub-computations may be executed in parallel safely, and where it is best to place the `futurecall` and `touch` operations. See Rogers et al. [1992] for more details.

In order to support this analysis, we can build upon the techniques previously proposed in the context of parallelizing imperative programs with recursive data structures [Hendren 1990; Hendren and Nicolau 1990; Hendren et al. 1992], and the CURARE restructuring compiler for Lisp [Larus and Hilfinger 1988a; Larus and Hilfinger 1988b; Larus 1989]. In both these cases, the objective is to analyze the program to determine which computations refer to disjoint pieces of the hierarchical data structure, and then to use this information to insert parallel function calls or futures automatically.

For our purposes, we plan to build on the *path matrix* analysis [Hendren 1990; Hendren and Nicolau 1990], an interprocedural analysis designed to determine statically if the data structures are indeed tree-like, and to approximate the relationships between different parts of the data structure. A typical analysis provides information about the disjointness of sub-trees, or the non-circularity of lists. Based on information available from this analysis, we have defined the appropriate tests to determine when it is safe to introduce futures. For a typical divide-and-conquer type recursive procedure, we say that the procedure consists of *pre-computation*, followed by the recursive calls implementing the *conquer part*, followed by *post-computation*. Our analysis must be used to: determine if it is safe to execute the various sub-computations for the conquer step in parallel, determine if the pre-computation can be overlapped with the conquer step, and place the touches in the latest possible position to ensure maximum concurrency.

8. RELATED WORK

Programming support for parallel machines is a very active area of research. Out of necessity we restrict our discussion to papers that seem particularly relevant to our goal of supporting pointer-based dynamic data structures.

Carriero et al.'s [1986] work on *distributed data structures* on Linda shares a common objective with Olden, namely providing a mechanism for distributed processors to work on a shared linked structures. In the details, however, the two approaches are quite different. The Linda model provides a global shared address space (tuple space), but no control over the actual assignment of data to processors. Also, while Linda allows the flexibility of arbitrary distributed data structures, including arrays, graphs and sets, it is an explicitly parallel model that requires the programmer to parallelize a program by hand. The distributed data structures in Olden are restricted to hierarchical structures at present, but we provide exact control over data layout and have a runtime model that we believe is amenable to automatic parallelization.

Hudak and Smith's "Para-functional programming" model [1986] has some similarity with Olden. In both cases, sequential semantics are preserved and annotations are used to distribute the computation. In para-functional programming, however, the annotations are *mapping expressions*, which specify the processor to run the annotated expression, whereas in Olden, we annotate the allocations and let the data layout determine the location of computations. The other main difference is that Hudak and Smith start from a referentially transparent language, which makes preserving sequential semantics trivial.

Emerald [Jul et al. 1988] and Amber [Chase et al. 1989], which are object-oriented languages developed at the University of Washington, employ thread and object migration mechanisms to improve locality. Emerald was designed for programming distributed system. Amber permits an application program to use a network of homogeneous processors as an integrated multiprocessor. These languages provide primitives for object location and mobility, and constructs to allow the programmer to indicate whether the thread or the object(s) should move to satisfy an invocation that references a remote object.

Prelude, a language being developed at MIT [Weihl et al. 1991; Hsieh et al. 1993], includes a migration mechanism similar to ours. The goal of the Prelude project is to develop a language and support system for writing portable parallel programs. Prelude is an explicitly parallel language that provides a computation model based on threads and objects. Annotations are added to a Prelude program to specify architecture-specific implementation details. These annotations specify which of several mechanisms should be used to implement an object or thread. The language supports a variety of mechanisms including remote procedure call, object migration, and computation migration. The goals of the Olden and Prelude projects are different. Our focus is on effectively parallelizing sequential C programs; thread migration is simply part of the underlying model. Their goal is to develop a new language that facilitates writing portable parallel programs using computation migration as one possible tool.

Orca[Bal and Kaashoek 1993] 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 runtime system to implement shared objects efficiently. The Orca compiler produces a summary of how shared objects are accessed that is used by its runtime system to decide if a shared object will be replicated, and if not, where it will 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.

Split-C [Culler et al. 1993] is a parallel extension of C that provides shared-memory, message-passing, and data parallel abstractions to the programmer. Split-C provides a global address space that maintains a clear concept of locality by providing both local and global pointers. Local pointers are guaranteed to point to the local processor, whereas, global pointers may point to any (possibly remote) location. Split-C provides a variety of primitives to manipulate global pointers efficiently. The way work is allocated is a major difference between Olden and Split-C. In Olden, work follows the data. In Split-C, the programmer places the work and uses the provided routines to retrieve any necessary data.

In a related piece of work, Lumetta et al. [1993] 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. The system is also based on a global pointer abstraction and provides routines for allocating data objects, for asynchronous reading of a data object, and for synchronous writing of an object. This object system is intended to be used in explicitly parallel programs and like Split-C has a work allocation model different from Olden's.

The Concert system [Chien et al. 1993; Karamcheti and Chien 1993] provides compiler and runtime support for efficient execution of fine-grained concurrent object-oriented programs. 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.

Cid [Nikhil 1994], 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. Unlike Split-C, these pointers cannot be dereferenced directly. Instead, the programmer explicitly requests access to the 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 runtime system maintains consistency. A major design goal of Cid is that Cid programs be able to run on existing hardware using existing compilers. This goal supports portability, but may hamper efficiency.

9. CONCLUSIONS

We have presented a new approach supporting programs that use pointer-based dynamic data structures on message passing machines. In developing our new approach we have noted a fundamental problem with trying to apply runtime 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 runtime resolution model ineffective.

Our mechanism avoids these fundamental problems by more closely matching the dynamic nature of the data structures. Rather than making each processor decide if it should execute a statement by determining if it owns the relevant piece of the data structure, we use a thread migration strategy that migrate the thread of computation to the processor that owns the data automatically. Coupled with the thread migration technique is our futurecall mechanism, which introduces parallelism by allowing processors to split the thread of computation.

We have implemented our execution mechanism on the iPSC/860 and the CM-5, and have used this system to run some sample programs. Our results are encouraging, especially in light of the poor message-passing performance of current machines. We believe that our model will achieve better results as communication systems improve. Our experiences also point out the importance of merging results computed on different processors to the performance of divide-and-conquer programs. We plan to focus on this issue as part of our continuing research.

ACKNOWLEDGMENTS

Dave Hanson and Chris Fraser wrote `lcc` and answered many questions about it. Dave Hanson suggested the use of spaghetti stacks. Kai Li and the Intel Supercomputer Systems Division provided access to the iPSC/860 that we used for our experiments. The National Supercomputing centers at the University of Illinois (NCSA) and Syracuse University (NPAC) provided access to their CM-5s. Josh Barnes provided an implementation of Barnes-Hut. Steve Lumetta provided an implementation of Power. Joe Hummel suggested Voronoi Diagram as a possible application; we obtained the initial implementation, written by L. Guibas and J. Stolfi, from Netlib at ORNL. Jörg Müller recommended TSP and provided an initial implementation. The many suggestions made by Mark Hill, Mukund Raghavachari, and the anonymous reviewers helped to improve this paper.

APPENDIX A. SAMPLE FUTURECALL CODE

The following is an example of the code sequence for `futurecalls` in Olden's CM-5 implementation.

```
! CM-5 Future call with integer return value

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

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

The following is an example of the code sequence for **futurecalls** in Olden's iPSC/860 implementation.

```
// Intel i860 Future call with integer return value
// Future cell is in r30
// r14 is future cell stack pointer

st.l r0, 0(r30)            // Mark future cell empty/full
st.l r14, 4(r30)           // Store next field
st.l fp, 12(r30)           // Store fp
addu 0, r30, r14           // Push future cell
call _TreeAdd              // Call TreeAdd
addu 32, r1, r1            // Delay slot of TreeAdd call
                           // Modify return address, which
                           // is stored in r1, to point to
                           // the first subsequent st.l

// Abnormal Return (Future had been stolen)
.long _TreeAdd.cs          // Pointer to register unwinding code
st.l r16, 16(fp)           // Store return value
mov  fp, r19               // Pass fc pointer as argument to stolen
ld.l 12(fp), fp            // Load fp from future cell
orh  h%_TreeAdd.cs, r0, r18
br   ___stolen             // Call stolen with register unwinding
or   l%_TreeAdd.cs, r18,r18 // address as argument

// Normal Return (Future has not been stolen)
.long _TreeAdd.cs          // Pointer to register unwinding code
st.l r16, 16(r14)          // Store value in future cell
ld.l 4(r14), r14           // Pop future stack
L.14:
```

REFERENCES

Allen, F., Burke, M., Charles, P., Cytron, R., and Ferrante, J. 1988. An overview of the PTRAN analysis system for multiprocessing. *J. of Parallel and Distributed Computing 5,* 5 (Oct), 617–640.

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

Amarasinghe, S. and Lam, M. 1993. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 126–138.

Anderson, J. and Lam, M. 1993. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 112–125.

Appel, A. and Li, K. 1991. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 96–107.

Bal, H. E. and Kaashoek, M. F. 1993. 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)*.

Bilardi, G. and Nicolau, A. 1989. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal of Computing 18,* 2, 216–228.

Bobrow, D. and Wegbreit, B. 1973. A model and stack implementation of multiple environments. *Communications of the ACM 16,* 10, 591–603.

Callahan, D. and Kennedy, K. 1988. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing 2,* 2 (Oct.), 151–169.

Carlisle, M., Rogers, A., Reppy, J., and Hendren, L. 1993. Early experiences with Olden. In *Conference Record of the Sixth Workshop on Languages and Compilers for Parallel Computing*. Also appears in Springer Verlag LNCS 768 (pp. 1-20).

Carriero, N., Gelernter, D., and Leichter, J. 1986. Distributed data structures in Linda. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 236–242.

Chase, J., Amador, F. G., Lazowska, E., Levy, H. M., and Littlefield, R. J. 1989. The Amber system: Parallel programming on a network of multiprocesors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pp. 147–158.

Chien, A., Karamcheti, V., and Plevyak, J. 1993. The Concert system–compiler and runtime support for efficient, fine-grained concurrent object-oriented programs. Tech. Rep. UIUCDCS-R-93-1815 (June), Department of Computer Science, University of Illinois, Urbana, Illinois.

Cormen, T., Leiserson, C., and Rivest, R. 1989. *Introduction to Algorithms.* McGraw Hill.

Culler, D. E., Dusseau, A., Goldstein, S. C., Krishnamurthy, A., Lumetta, S., von Eicken, T., and Yelick, K. 1993. Parallel programming in Split-C. In *Proceedings of Supercomputing 93*, pp. 262–273.

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

Gerndt, M. 1990. Automatic parallelization for distributed-memory multiprocessing systems. Ph. D. thesis, University of Bonn.

Guibas, L. and Stolfi, J. 1985. General subdivisions and voronoi diagrams. *ACM Transactions on Graphics 4,* 2, 74–123.

Gupta, R. 1992. SPMD execution of programs with dynamic data structures on distributed memory machines. In *Proceedings of the 1992 International Conference on Computer Languages*, pp. 232–241.

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

Hendren, L. J. 1990. Parallelizing programs with recursive data structures. Ph. D. thesis, Cornell University.

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

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

Hiranandani, S., Kennedy, K., and Tseng, C. 1991. Compiler optimizations for FORTRAN D on MIMD distributed memory machines. In *Proceedings of Supercomputing 91*, pp. 86–100.

Hogen, G. and Loogen, R. 1993. A new stack technique for the management of runtime structures in distributed implementations. Tech. Rep. 3, RWTH Aachen.

Horwat, W., Chien, A., and Dally, W. 1989. Experience with CST: Programming and implementation. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 101–108.

Hsieh, W., Wang, P., and Weihl, W. 1993. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 239–248.

Hudak, P. and Smith, L. 1986. Para-functional programming: A paradigm for programming multiprocessor systems. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 243–254.

JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. 1988. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems 6,* 1, 109–133.

KARAMCHETI, V. AND CHIEN, A. 1993. Concert – efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Proceedings of Supercomputing 93*, pp. 598–607.

KARP, R. 1977. Probabilistic analysis of partitioning algorithms for the traveling-saleman problem in the plane. *Mathematics of Operations Research 2,* 3 (August), 209–224.

KOELBEL, C. 1990. Compiling programs for nonshared memory machines. Ph. D. thesis, Purdue University, West Lafayette, IN.

LARUS, J. R. 1989. Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors. Ph. D. thesis, University of California, Berkeley.

LARUS, J. R. 1991. Compiling lisp programs for parallel execution. *Lisp and Symbolic Computation 4,* 1, 29–99.

LARUS, J. R. AND HILFINGER, P. N. 1988a. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 21–34.

LARUS, J. R. AND HILFINGER, P. N. 1988b. Restructuring Lisp programs for concurrent execution. In *Proceedings of the ACM/SIGPLAN PPEALS 1988 - Parallel Programming: Experience with Applications, Languages and Systems*, pp. 100–110.

LEE, D. T. AND SCHACHTER, B. J. 1980. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer and Information Sciences 9,* 3, 219–242.

LUMETTA, S., MURPHY, L., LI, X., CULLER, D., AND KHALIL, I. 1993. Decentralized optimal power pricing: The development of a parallel program. In *Proceedings of Supercomputing 93*, pp. 240–249.

MOHR, E., KRANZ, D. A., AND HALSTEAD, JR., R. H. 1991. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems 2,* 3 (July), 264–280.

MÜLLER, J. 1993. Parallelverarbeitung auf workstation-clustern mittels express und network-linda. Diplomarbeit in Elektrotechnik, RWTH-Aachen.

NIKHIL, R. 1994. *Cid*: A parallel, "shared-memory" C for distributed-memory machines. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing.*

ROBERTS, E. S. AND VANDEVOORDE, M. T. 1989. WorkCrews: An abstraction for controlling parallelism. Tech. Rep. 42 (April), DEC Systems Research Center.

ROGERS, A. AND PINGALI, K. 1989. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 69–80.

ROGERS, A., REPPY, J., AND HENDREN, L. 1992. Supporting SPMD execution for dynamic data structures. In *Conference Record of the Fifth Workshop on Languages and Compilers for Parallel Computing.* Also appears in Springer Verlag LNCS 757 (pp. 192-207).

SCHOINAS, I., FALSAFI, B., LEBECK, A. R., REINHARDT, S. K., LARUS, J. R., AND WOOD, D. A. 1994. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems.* To appear.

VON EICKEN, T., CULLER, D. E., GOLDSTEIN, S. C., AND SCHAUSER, K. E. 1992. Active messages: a mechanism for integrating communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 256–266.

WEIHL, W., BREWER, E., COLBROOK, A., DELLAROCAS, C., HSIEH, W., JOSEPH, A., WALDSPURGER, C., AND WANG, P. 1991. Prelude: A system for portable parallel software. MIT/LCS 519, Massachusetts Institute of Technology, Cambridge, Massachusetts.

WOLFE, M. 1989. *Optimizing Supercompilers for Supercomputers.* Pitman Publishing, London.

ZIMA, H., BAST, H., AND GERNDT, M. 1988. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing 6,* 1, 1–18.