Early Experiences with Olden

Martin C. Carlisle,[1*]
Anne Rogers,[1**]
John H. Reppy,[2] and
Laurie J. Hendren[3***]

[1] Princeton University, Princeton NJ 08544, USA
[2] AT&T Bell Laboratories, Murray Hill NJ 07974, USA
[3] McGill University, Montreal, Quebec, Canada H3A 2A7

Abstract. In an earlier paper[RRH92], we presented a new technique for the SPMD parallelization of programs that use dynamic data structures. Our approach is based on migrating the thread of computation to the processor that owns the data, and annotating the program with *futures* to introduce parallelism. We have implemented this approach for the Intel iPSC/860. This paper reports on our implementation, called *Olden*, and presents some early performance results from a series of non-trivial benchmarks.

1 Introduction

Compiling for distributed memory machines has been a very active area of research in recent years; particularly in the area of 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 operations performed on these pieces can proceed in parallel. This property of scientific programs has been exploited in vectorizing compilers [ABC+88, AK87, PW86, Wol89]. More recently, this property has been used by researchers investigating methods for automatically generating parallel programs for SPMD (Single-Program, Multiple-Data) execution on distributed memory machines.[4] To date, this research has largely ignored the question of how to compile programs that use trees as their primary data structures and recursion as their primary control structure. In part, this is because there are fundamental problems with trying to apply the techniques, such

* Supported, in part, by a National Science Foundation Graduate Fellowship and the Fannie and John Hertz Foundation.

** This work was supported, in part, by NSF Grant ASC-9110766, by DARPA and ONR under contracts N00014-91-J-4039, and by the Intel Supercomputer Systems Division.

*** The research supported, in part, by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.

[4] For examples, see [CK88, ZBG88, RP89, Ger90, KMvR90, Koe90, Rog90, RSW90, HKT91, AL93, AM93].

as runtime resolution, currently used to produce SPMD programs for scientific programs, to programs that use dynamic data structures. In the case of scientific programs, the array data structures are statically allocated, statically mapped, and directly addressable. Dynamic data structures, on the other hand, are dynamically allocated, dynamically mapped, and must be recursively traversed to be addressable. These properties of dynamic data structures preclude the use of simple local tests for ownership, and therefore make the runtime resolution model ineffective.

One exception is a recent paper by Gupta [Gup92] that suggests a mechanism for introducing global names for each element of a data structure at runtime to allow an approach similar to runtime resolution. In his approach, a name, which is determined by the node's position in the structure, is assigned to each node as it is added to a data structure. This name is then registered with the other processors. Once a processor has a name for the nodes in a data structure, it can traverse the structure without further communication. A problem is that this new way of naming dynamic data structures leads to restrictions on how they may be used. For example, because the name of a node is determined by its position, only one node can be added to a structure at a time. Another problem 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. These problem arise from trying to retrofit runtime resolution, a method designed for statically allocated, directly addressable structures, to handle dynamic data structures, which are neither directly addressable nor statically allocated.

In a previous paper [RRH92], we presented an execution model with associated compiler techniques for parallelizing programs that use dynamic data structures using an SPMD approach. We have since implemented our execution model on the Intel iPSC/860. This paper starts with a review of our proposed execution model and compiler techniques. This review is followed by a description of our implementation, which we call *Olden*, and a discussion of our early experiences with using Olden on a series of non-trivial benchmarks.

2 Data-driven Execution

In this section, we review our technique for executing SPMD programs on hierarchical data structures.[5] This technique consists of two mechanisms: thread migration and thread splitting.

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. We view a heap address as consisting of a pair of a processor name and a local address. This information is encoded as a single address. For simplicity

[5] We suggest that readers unfamiliar with our proposed execution model consult our original paper[RRH92], which contains an in-depth discussion of the model.

we assume that there is no global data. We also require that programs do not take the address of stack-allocated objects; thus there are no pointers into the processor stacks.

The basic programming model may be summarized as follows. The programmer writes a normal sequential program except for a slight difference in how dynamic data structures are allocated. In our programming model, 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. A typical choice of mapping is to build a tree such that the sub-trees at some fixed depth are equally distributed over all processors. Appendix A contains a sample allocation routine.

2.1 Thread Migration — Exploiting Locality

This section briefly describes our mechanism for migrating a single thread of control through a set of processors based on the placement of heap-allocated data. When a thread executing on Processor P attempts to access a word 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 access that caused the migration. Notice that because the stack is sent with the thread, stack references are always local and cannot cause a migration.

Full thread migration is potentially quite expensive, since 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: namely, 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, since 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 code 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.

2.2 Thread Splitting — Introducing Parallelism

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 use compiler transformations 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 splits the thread of execution into many pieces that can be executed out of order, thus the introduction of continuation capturing must be based on an analysis of the program (see Section 3).

Our continuation capturing mechanism is essentially a variant of the *future* mechanism found in many parallel Lisps [Hal85]. In the traditional Lisp context, the expression `(future` e`)` is an annotation to the system that says that e can be evaluated in parallel with its context. The result of this evaluation is a *future cell* that serves as a synchronization point between the child thread that is evaluating e and the parent thread. If the parent attempts to read the value of the future cell, called a *touch*, 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, Krantz, and Halstead [MKH91], is to save the future call's context (return continuation) on a work list and to evaluate the future's body directly.[6] 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*.

3 Compiler Issues

In order to generate a data-driven SPMD program automatically, one needs compiler analysis to determine which sub-computations may be executed in parallel safely, and where it is best to place the `futurecall` and `touch` operations.

To support this analysis, we can build upon the techniques previously proposed in the context of parallelizing imperative programs with recursive data structures [Hen90, HN90, HHN92], and the CURARE restructuring compiler for Lisp [LH88a, LH88b, Lar89]. 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 [Hen90, HN90], 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*.

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

Our analysis must be used to: (1) determine if it is safe to execute the various sub-computations for the conquer step in parallel, (2) determine if the precomputation can be overlapped with the conquer step, and (3) place the touches in the latest possible position to ensure maximum concurrency. Our previous paper describes this in more detail [RRH92].

4 A Simple Example

To make the ideas of the previous two sections more concrete, we present a simple example. Figure 1(a) gives the C code for a prototypical divide-and-conquer program. Using the analyses outlined in Section 3, a compiler would generate the semantically equivalent program annotated with futures give in Figure 1(b). In the annotated version, the left recursive call to `TreeAdd` has been replaced by a `futurecall`,[7] 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 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.

5 Implementation

We have implemented a prototype of our execution model at Princeton on a 16 node Intel iPSC/860 hypercube. The input to the system is a C program annotated with `futurecall`s and `touch`es. Our system consists of a runtime system and a compiler for the annotated C code; the compiler also generates tests for pointer locality.

5.1 The Runtime System

Figure 2 provides an overview of the control structure of the runtime system. The main procedure is `MsgDispatch`, which is responsible for assigning work to a processor when it becomes idle. Captured futures are given first priority, then messages from other processors in the order received. The runtime system processes four different events: migrate on reference, migrate on return, steal future, and suspend touch. We discuss futures in the next section, so we focus on migration in this section.

[7] 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 where the result is needed. Also note that the fetching of `t`'s `right` field cannot cause a migration.

```
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);
}
```

(a) Original code

```
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);
    }
}
```

(b) Annotated code

Fig. 1. `TreeAdd` code

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, and some bookkeeping information: the name of the processor that originated the current function, an address to continue

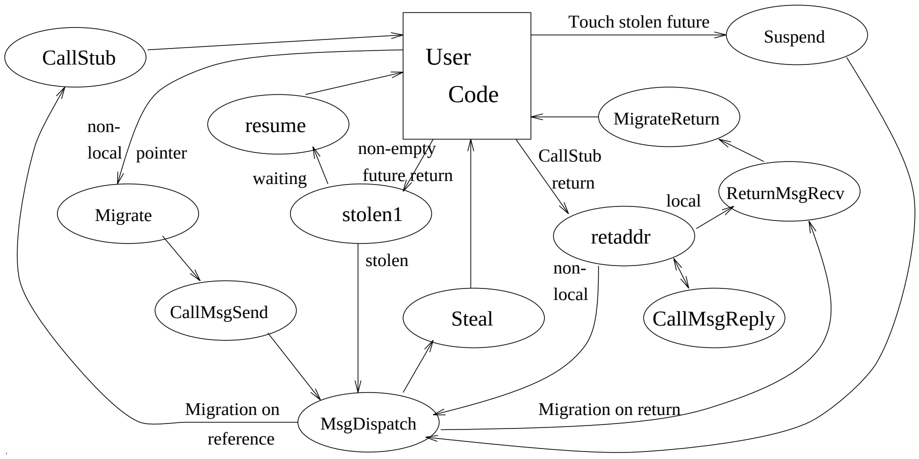

Fig. 2. The Runtime System

execution, the frame pointer, the pointer causing the migration, the return value size, the framesize, and the argument build area size. We allocate extra space in 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.

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[HCD89]. A similar optimization is that if a migrated procedure exits on the same processor as it began (for example, it migrated from P to Q then back to P), the return message is not sent, rather `ReturnMsgRecv` is called directly.

When `MsgDispatch` selects an incoming migration message as the next task, it transfers control to `CallStub` to process the message. `CallStub` allocates space for a return message, in which it stores the framesize, return value size, and frame pointer from the migration message. Then space on the stack is allocated for the frame, the argument build area and a stub frame. The callee-save register values from the message are loaded, and the pointers to the return area and the argument build area are adjusted (if they exist). The return address stored in the frame is changed to `retaddr`, the stub return procedure. Finally, the pointer causing the migration, the frame pointer and the program counter are loaded, and execution resumes 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, and sends the message to the processor that began execution of the procedure.

Migrate on Return When `MsgDispatch` selects a return message as the highest priority task, it calls `ReturnMsgRecv`, which loads the contents of the callee-save registers from the message, and deactivates the frame of the procedure that migrated. 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 a port of **lcc** [Fra91, FH91], an ANSI C compiler, to the i860, with modifications to handle our execution model. **lcc** does not have an optimizer.

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 and a local address, so it is necessary to check the tag on each heap pointer dereference.[8] **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 `Migrate`, 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. This overhead can be reduced by observing that only the first reference in a string of references to the same pointer can cause a migration. In the `TreeAdd` example, only the reference `t->left` can cause a migration. All subsequent references are guaranteed to be local.[9] 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 references use the resulting local pointer.

A simple compiler optimization can take advantage of this construct. A pointer dereference is *aligned* with another pointer dereference 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 string, can be tested using local. The local address that results can be used by the aligned references. The code in Figure 3 shows the result of applying this optimization to the `TreeAdd` example. Note that the deferences of `local_t` require no special handling.

[8] Note that the address translation hardware could be used to detect non-local references [AL91].

[9] Note that this observation relies on the fact that function calls always return to the original processor on return.

```
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. 3. `TreeAdd` code using `local`

Compiling `futurecall` As noted previously, future calls allow the introduction of parallelism. We use two data structures to support future calls: future cells and the future stack, which serves as work list. A future cell can be in one of four states:

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: when inserting futures, the compiler insures that no future can be touched by more than one thread).

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

Figure 4 gives the allowable state transitions for a future cell. The future stack is allocated as part of the processor's stack. In particular, this means that no migrations may occur between a future call and its corresponding touch.

When the backend (unoptimized) encounters a **`futurecall`** annotation, it generates in-line code to store the continuation, which consists of the callee-save registers, the frame pointer, the state of the future cell (**Empty**), and the program counter. This information is stored in the future cell. Then the call is generated. After the call, code is generated in-line to store the return value in the future cell, and check the state of the future. If it is **Empty**, the cell is popped from the stack, and execution continues.

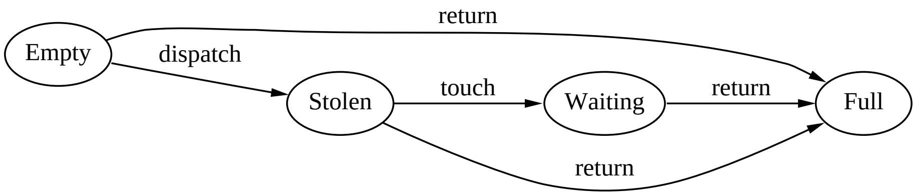

Fig. 4. Future cell state transitions

In the case that the cell is not **Empty**, (that is, the 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. In Section 5.3, we describe how to resume this thread of computation.

As described, the overhead of future calls is quite large. Ideally, we would like to avoid having to save all of the callee-save registers, and instead just save those that are live using an interprocedural analysis to determine the subset of live registers. But, even this would be a duplication, since the callee-save registers that are used by the **`futurecall`** will be stored upon entry to the function. We can do better by exploiting the fact that MsgDispatch gives stealing futures the highest priority. As a result, a descendant of the future call must have been executing immediately before the steal. Therefore, the state of the callee-save registers may be restored by a process called *register unwinding*, which involves executing the restores of callee save registers for each procedure in the call chain from the user code function last executing to the function called in the future call.

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.

A second optimization is to eliminate the necessity of 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 body of the future is modified to point to code which will test for **Stolen** or **Waiting**, and perform the appropriate function.

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.

5.3 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 procedure continues, otherwise the library routine `Suspend` is called. `Suspend` stores the values of the callee-save registers 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 and program counter from the suspended cell, and resumes execution of the procedure.

5.4 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, overwriting the frames of the migrated thread. To avoid this problem, we adopt a simplification of a single stack technique for multiple environments by Bobrow and Wegbreit [BW73].

Their method maintains counters for the number of uses and frame extensions, which arise from the use of backtracking, for each frame. On entry, a basic frame and extension are allocated contiguously at the end of the stack, with the counters set to 1. 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 1). The end of stack pointer is then adjusted appropriately. When control returns to an extension, if active frames exist between this 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 will 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 frames from the continuations interleaved.[10] 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 the frame is at the end of the stack, 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 garbage frames.

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

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 5 provides an example of this process. Assume that initially the pro-

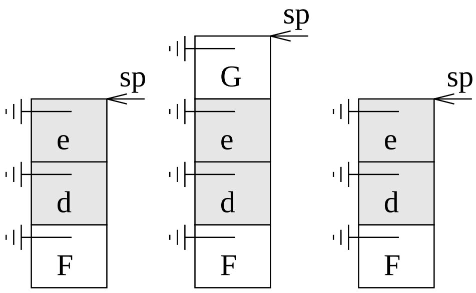

Fig. 5. Simplified Spaghetti Stack — simple case

cedure F is executing. 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 6 provides a more complex example. Again, F calls G. G migrates, and the runtime system gives control to

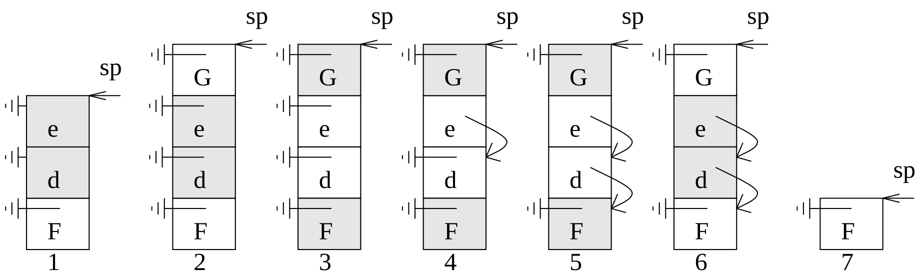

Fig. 6. Simplified Spaghetti Stack — complex case

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.

The overhead resulting from our simplified spaghetti stacks is minimal. 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, load, and two conditional branch instructions) over the basic stack exit code for the i860.

There is one problem with the scheme that we have proposed. Since the simplified spaghetti stack scheme does not deallocate dead frames immediately, we cannot give a bound for how the size of the simplified spaghetti stack in terms of the number of live frames. In certain degenerate communication patterns, there is no bound. For example, in the case where p threads numbered $[1..p]$ alternately migrate between Processor 0 and Processor p, the stack size on Processor 0 is bounded only by the number of migrations. In our experience, it has not been necessary or desirable for programs to have this property. However, we must be able to detect and correct this problem should it arise. 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 one based on the remaining capacity of the stack can be used to determine when the stack needs to be compacted. The cost of the checks is minimal. Our prototype implementation does not check for stack overflow.

6 Results

This section describes out early experiments with our prototype implementation of Olden. We report results for four benchmarks: TreeAdd, Bitonic Sort, Barnes-Hut, and Voronoi Diagram. For each benchmark, the speedups are reported with respect to an implementation that was compiled using our compiler, but without the overhead of futures, pointer testing, or the spaghetti stack. We refer to this as the *plain* implementation. The speedup curves include a data point for one processor. This implementation, which we refer to as *one*, includes the cost of futures, pointer testing, and spaghetti stacks.

The benchmarks were hand-annotated to include future calls and touches. The programs were annotated aggressively. Our goal is to show that the execution model is not a barrier to efficient parallelization. We do not claim that all of the annotations can be generated automatically using existing compiler technology.

6.1 TreeAdd

Figure 7 shows the speedup for running `TreeAdd` on a 1,048,575 node tree. The speedup is not as large as one might hope because there is very little work over which to amortize the overhead of futures, pointer tests, and the spaghetti stack. The second curve represents the speedup using one rather than plain

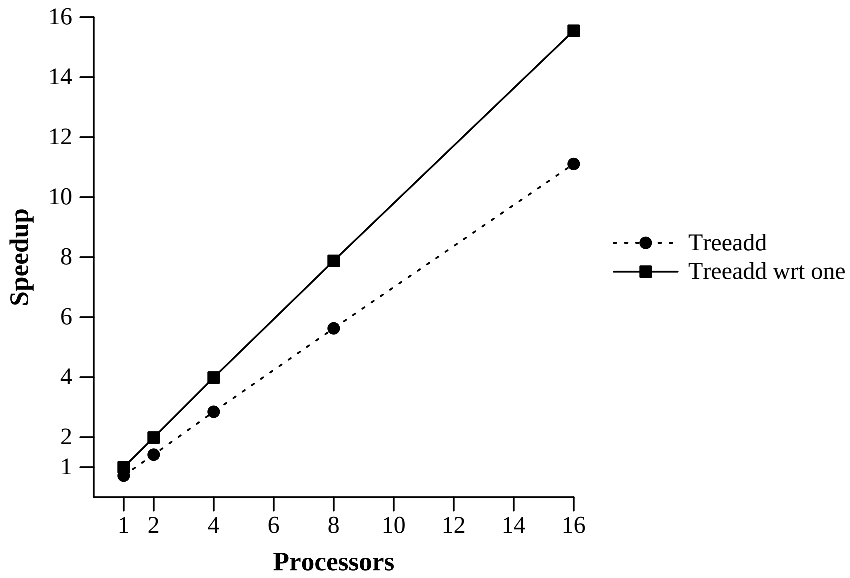

Fig. 7. `TreeAdd` speedup

as the baseline. This curve demonstrates that the execution model can exploit parallelism. The problem for this benchmark is that there is not enough real work to amortize the overhead of the system. The difference between plain and one is mostly attributable to the overhead of futurecalls. This is the only benchmark in which futurecalls represent the dominant factor in the overhead.

6.2 Bitonic Sort

The bitonic sort benchmark[BN89] allocates a random tree and performs two bitonic sorts: one forward and one backward. The benchmark has been modified to maintain data locality across the processors. We report speedup only for the sorting phases to avoid having the easily parallelizable build phase skew the results. Figure 8 contains four curves: two runs of the original implementation and two runs of an improved implementation. The original implementation, which does quite respectably on a medium sized problem (128K numbers), performs an unnecessary malloc for each recursive sort, thereby artificially increasing the amount of work per call. The improved Bitonic Sort, which removed this extra call, still displays parallelism but not nearly as much as the original version.

The difference between plain and one is substantial for this benchmark. The overhead from futures, pointer tests, and the spaghetti stack causes a .75 slowdown. Seventy-five percent of this overhead is attributable to pointer tests.

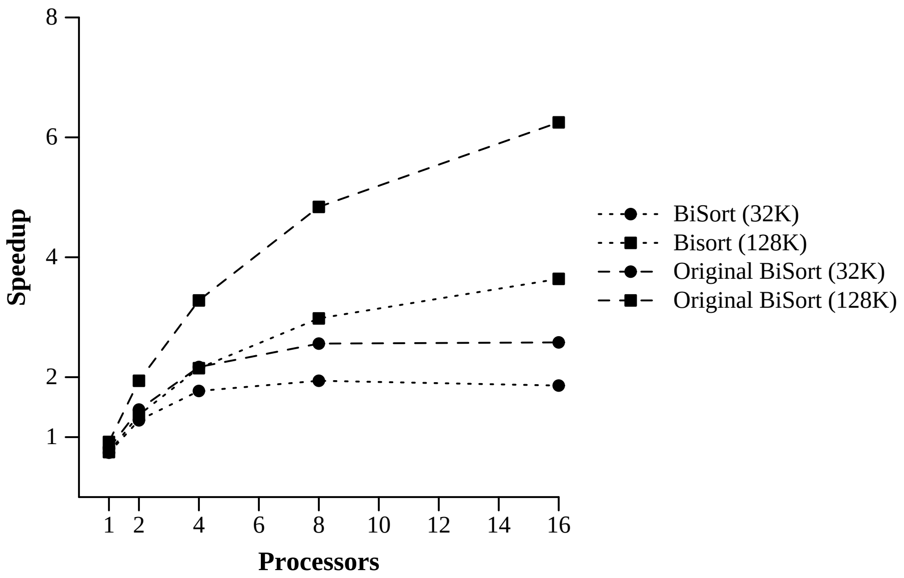

Fig. 8. Bitonic Sort speedup

6.3 Building Voronoi Diagrams

The Voronoi Diagram benchmark[GS85, LS80] generates a random set of points and computes a Voronoi Diagram for these points. To compute a Voronoi diagram, the algorithm splits the point set into two sets using the median point as the dividing line, it recursively computes the Voronoi Diagrams of the smaller sets, and merges them to form the final result. The merge phase is sequential and may require a linear number of migrations when the subsets are on different processors. In Figure 9, we report the speedup obtained for building the Voronoi Diagram for 64K points. The reported speedup does not include the cost of generating 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, migrations are expensive in our current implementation because the speed of message passing is not well matched to the speed of the processor.

As in Bitonic Sort, we suffer from a significant slowdown in going from plain to one (.68), which is attributable to the cost of testing for non-local pointers.

6.4 Barnes Hut

The Barnes Hut benchmark[BH86] simulates the motion of particles in space using an $O(n \log n)$ algorithm for computing the accelerations of the particles. This is a classic n-body problem. We report the cost of generating the points using a uniform distribution and computing the new acceleration and position for each particle for ten time steps. The results for this benchmark, shown in

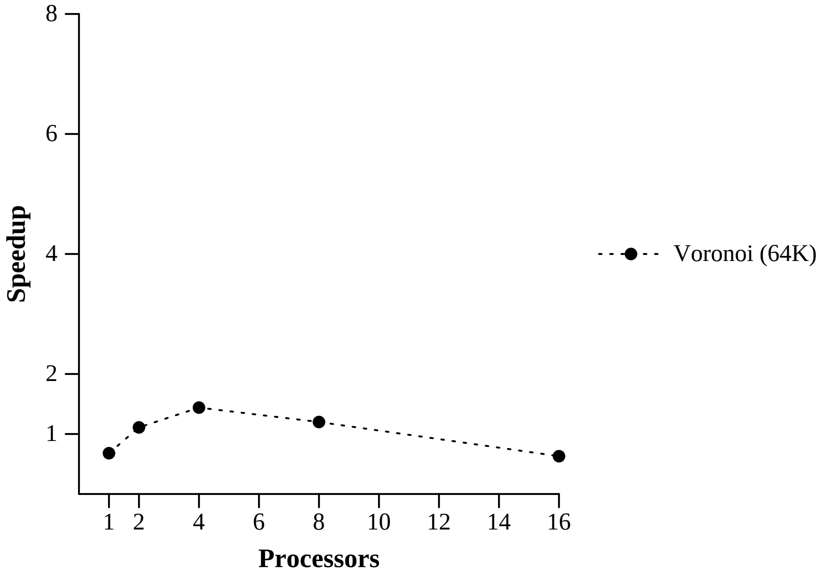

Fig. 9. Voronoi Diagram speedup

Figure 10, are encouraging. For a relatively small problem (4K particles), we achieve a speedup of almost eight for 16 processors. This computation is broken into three pieces: building the tree used to represent the particles, the acceleration calculation, and computing the new positions of the particles. The second two pieces perform quite well, in part, because we build a copy of the tree of every processor. This will not work for larger problem sizes. The tree building is sequential and starts to represent a substantial amount of the computation as the number of processors increases. Another factor is the cost of transmitting the particle and acceleration information to the processors. This cost increases as the number of processors increases.

6.5 Discussion

In our current implementations, migrations are expensive. Sending the data accounts for roughly ninety percent of the cost of a migration. We believe that by switching a machine with a better communication system, we will see a substantial improvement in the results for the Bitonic Sort and Voronoi Diagram benchmarks. We expect to see a less dramatic improvement in Barnes-Hut.

In early experiments, we observed super-linear speedup for several of these benchmarks. We traced this to the quadratic behavior of the i860's memory allocation routines. We reduced the effect of this problem by calling malloc only a few times and managing the acquired storage explicitly.

Both the Voronoi Diagram and Bitonic Sort benchmarks suffer from substantial overhead due to pointer testing. It may be possible to reduce this overhead

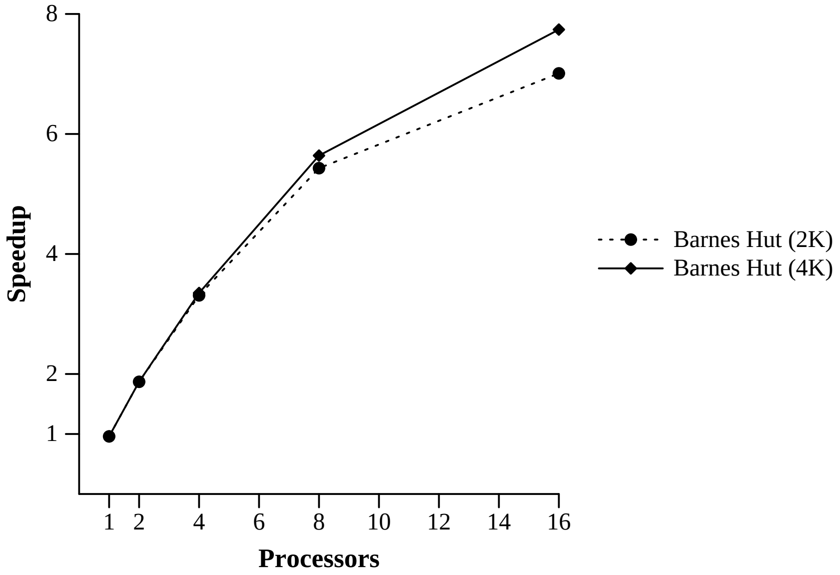

Fig. 10. Barnes-Hut speedup

pointers 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[AL91].

7 Conclusions

We have presented a new approach for generating SPMD parallel programs automatically from sequential programs that use hierarchical data structures. In developing our new approach, we have noted fundamental problems with trying to apply runtime resolution techniques, currently used to produce SPMD programs for scientific programs, to programs that use dynamic data structures. In the case of scientific programs, the array data structures are statically allocated, statically mapped, and directly addressable. Dynamic data structures, on the other hand, are dynamically allocated, dynamically mapped, and must be recursively traversed to be addressable. These properties of dynamic data structures preclude the use of simple local tests for ownership, and therefore make 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 migrates the 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.

In order to illustrate how a compiler can make use of the thread migration and lazy future mechanisms, we also outlined a strategy to parallelize divide-and-conquer type programs. This compilation method relies heavily on accurate alias and dependency analysis for hierarchical dynamic data structures.

We have implemented our mechanism for the iPSC/860, and have used this system to run some example programs. Our results are encouraging, especially in light of the poor message-passing performance of the iPSC/860, and we believe that our model can achieve substantially better results on a machine with better communication. 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.

References

[ABC+88] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *J. of Parallel and Distributed Computing*, 5:617–640, 1988.

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

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

[AL93] 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*, June 1993.

[AM93] 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*, June 1993.

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

[BN89] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal of Computing*, 18(2):216–228, 1989.

[BW73] D. Bobrow and B. Wegbreit. A model and stack implementation of multiple environments. *Communications of the ACM*, 16(10):591–603, 1973.

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

[FH91] C. Fraser and D. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, 1991.

[Fra91] C. Fraser. A code generation interface for ANSI C. *Software–Practice & Experience*, 21(9):963–988, 1991.

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

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

[Gup92] R. Gupta. SPMD execution of programs with dynamic data structures on distributed memory machines. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 232–241, April 1992.

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

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

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

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

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for FORTRAN D on MIMD distributed memory machines. In *Proceedings of Supercomputing 91*, pages 86–100, November 1991.

[HL93] Guido Hogen and Rita Loogen. A new stack technique for the management of runtime structures in distributed implementations. Technical Report 3, RWTH Aachen, {ghogen,rita}@zeus.informatik.rwth-aachen.de, 1993.

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

[KMvR90] C. Koelbel, P. Mehrotra, and J. van Rosendale. Supporting shared data structures on distributed memory architectures. In *Proceedings of the Second ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1990.

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

[Lar89] James R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD thesis, University of California, Berkeley, 1989.

[LH88a] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.

[LH88b] James R. Larus and Paul N. Hilfinger. Restructuring Lisp programs for concurrent execution. In *Proceedings of the ACM/SIGPLAN PPEALS 1988 - Parallel Programming: Experience with Applications, Languages and Systems*, pages 100–110, July 1988.

[LS80] D. T. Lee and B. J. Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer and Information Sciences*, 9(3):219–242, 1980.

[MKH91] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[PW86] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12), December 1986.

[Rog90] A. Rogers. *Compiling for Locality of Reference.* PhD thesis, Cornell University, August 1990.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989.

[RRH92] A. Rogers, J. H. Reppy, and L. J. Hendren. Supporting SPMD execution for dynamic data structures. In *Conference Record of the Fifth Workshop on Languages and Compilers for Parallel Computing*, August 1992. To appear as volume of Springer's Lecture Notes in Computer Science series.

[RSW90] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, University of Colorado at Boulder, April 1990.

[RV89] E. S. Roberts and M. T. Vandevoorde. WorkCrews: An abstraction for controlling parallelism. Technical Report 42, DEC Systems Research Center, April 1989.

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

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

A Allocation Example

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

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

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