# Supporting SPMD Execution for Dynamic Data Structures


Anne Rogers* — Princeton University — amr@cs.princeton.edu

John H. Reppy — AT&T Bell Laboratories — jhr@research.att.com

Laurie J. Hendren† — McGill University — hendren@cs.mcgill.ca



**Abstract**

In this paper, we address the problem of supporting SPMD execution of programs that use recursively-defined dynamic data structures on distributed memory machines. The techniques developed for supporting SPMD execution of array-based programs rely on the fact that arrays are statically defined and directly addressable. As a result, these techniques do not apply to recursive data structures, which are neither statically defined nor directly addressable. We propose a three pronged approach. First, we describe a simple mechanism for migrating a thread of control based on the layout of heap allocated data. Second, we explain how to introduce parallelism into the model using a technique based on futures and lazy task creation[MKH91]. Third, we present the compiler analyses and parallelization techniques that are required to exploit the proposed mechanism.


## 1 Introduction

Compiling for distributed memory machines has been a very active area of research in recent years[CK88, Ger90, HKT91, KMvR90, Koe90, RP89, Rog90, RSW90, ZBG88]. 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 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 vectorizing and parallelizing compilers[ABC+88, AK87, PW86, Wol89]. 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 wish to address the problem of the automatic generation of SPMD parallel 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.

In order to determine 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 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; non-local references are satisfied through explicitly passed messages, which are expensive. Therefore, arranging a computation so that most references are local is crucial to producing efficient code. 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.


*This work was supported, in part, by NSF Grant ASC-9110766.

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

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*[RP89], 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 ownership rule: the work of an assignment statement (`v := e`), including the computation of `e`, is assigned to the processor that `v` is mapped to. Control statements such as conditionals and loop are executed by all processors. The code produced by this method can be improved substantially using the arsenal of techniques developed for vectorizing compilers, such as data dependence analysis and loop restructuring[AK87, PW86, Wol89].

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 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, often this partitioning can be used to distribute parallel tasks over the sub-pieces. One such natural parallel sub-division is used in many divide-and-conquer programs. Thus, so far, we see no fundamental problem in mapping these programs to distributed memory machines. However, 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 that operations on a dynamic data structure are independent is substantially harder than determining that operations on an array are independent. This is partially due to the fact 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 loops with their easily partionable index sets, is the primary control structure for use 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[Gup92] 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 every processor has a name for the nodes in a data structure, it can traverse the structure without any further communication.

It is important to note that this new way of naming dynamic data structures leads to restrictions on how the dynamic data structures are 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 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.

It is our belief that the difficulty of applying runtime resolution to dynamic data structures is because this strategy was developed for statically-defined, directly-addressable, rectangular arrays. It is not the proper strategy for dynamic data structures that are not statically defined, nor directly addressable. We propose a more dynamic approach that is better matched to the dynamic nature of the data structures themselves. As in the earlier approaches, the programmer is responsible for mapping data to processors. However, we propose that this be done at runtime, by specifying a processor name with every memory allocation request. In addition, rather than making each processor decide if it owns the data, we find that it is much more natural to send computation to the processors that own the data explicitly. Thus, as a dynamic structure is recursively traversed the computation will migrate to the processor that owns that part of the structure. Before presenting our three-pronged approach to the problem of supporting SPMD execution for programs that use recursively-defined dynamic data structures, we review the basic SPMD model and our programming model. In Section 3 we present the first part of our solution - a simple mechanism for migrating a thread of control based on the layout of heap allocated data. In Section 4 we discuss the second important insight by explaining how to introduce parallelism into the thread migration model using a technique based on futures and lazy task creation[MKH91]. Finally, in Section 5 we present the third prong - the compiler analyses and parallelization techniques that are required to transform a sequential program into an SPMD program for a distributed memory machine. The paper ends with a discussion of conclusions and future work.

## 2 The SPMD model

Before we explain the details of our approach, let us review the basic SPMD model and programming model 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. For simplicity we assume that there is no global data (this could be put in the distributed heap).

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.

## 3 Thread migration

This section describes a mechanism for migrating a single thread of control through a set of processors based on the placement of data. Following a brief overview of thread migration, we describe the mechanism in detail, then present a simple example, and finally discuss how allocation fits into our model.

The basic idea is that when a thread executing on Processor $P$ attempts to access a word residing on Processor $Q$, the thread is migrated from $P$ to $Q$. For an SPMD machine, 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 executing the thread at the instruction that caused the migration. Processor $P$ remains idle until another (or the same) thread migrates to it.

We view a memory address as consisting of a pair of a processor name and a local address. This information can be encoded as a single address, and the address translation hardware can be used to detect non-local references [AL91]. When Processor $P$ executes a load or store instruction that refers to $Q$'s memory, the instruction traps. The trap handler is then run on $P$, which is responsible for packaging up the thread's state information and sending it to Processor $Q$. Notice that because we send the entire stack with the thread, stack references are always local and cannot cause a migration.

### 3.1 The migration mechanism

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 top-most 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 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 sent because it is no longer needed.

Figure 1 contains pseudo-code for the various parts of thread migration. Notice that the receiver's

```
Trap on Memory Reference : pointer <proc, addr>
    Send (proc, REF, <PC, Top-most Stack Frame, Registers, <retPC,retFP>>)
    dispatch()


Return Stub:
    Retrieve retPC  from stack
    Retrieve return frame pointer, <proc, addr>,  from stack
    Send(InCH.proc, RET, <retPC, <proc, addr>, Registers>)
    dispatch()


Receiver:
    if (peek(InCh) = REF) {
       Recv(InCh, REF, <NewPC, Stack_Frame, NewRegisters, <retPC,retFP>>)
       Allocate stack space for stub
       Store retPC  on stack
       Store retFP  on stack
       Allocate stack space and fill with contents of Stack_frame
       Copy NewRegisters data into Registers
       retFP = stubFP
       retPC = stubPC
       PC = NewPC
    } else {
       Recv(InCH, RET, <NewPC,NewFP, NewRegisters>)
       Copy NewRegisters into Registers
       FP = NewFP
       PC = NewPC
    }
```

Figure 1: Pseudo-code for thread migration

actions depend on the type of the migration message. `Peek` blocks until a message is available to be received and returns the type of that message.

A small optimization is possible. When thread migration is forced by a non-local reference, the trap handler examines the current return address to determine whether it points to the return stub. If so, the original return address and return frame pointer are pulled from the stub's frame and passed as part of the migration message. This is analogous to a tail-call optimization, and avoids the possibility of a chain of trivial returns should a thread migrate several times during the course of executing a single function.

### 3.2 An example

To illustrate the process of thread migration, we present an instruction-level trace of a thread on a two-processor machine. Figure 2 gives the "Simple C" code for a simple function that adds the values stored in a binary tree. Simple C is essentially three-address code with control structures written in C syntax [HSSW92, HDE$^{+}$92].

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

int TreeAdd (tree *t)
{
    if (t == NULL)
        return 0;
    else {
        tree *t_left, *t_right;
        int  r_left, r_right;

        t_left = t->left;                /* this can trap */
        r_left = TreeAdd (t_left);
        t_right = t->right;              /* this can trap */
        r_right = TreeAdd (t_right);

        return (r_left + r_right + t->value);
    }
}
```

Figure 2: `TreeAdd` C code

Figure 3 gives a translation of this into an idealized machine code, for a processor with word addressing. Figure 4 shows the beginning of a sample execution of `TreeAdd` on the tree described in Figure 5. Only user instructions are shown in the trace; except for sends and receives thread management instructions are not shown. At step 8, Processor 0 executes a `LOAD` instruction that traps on a non-local memory reference (to $<1, 0>$). This causes the thread to be migrated from Processor 0 to Processor 1.[1] Processor 1 receives the message at step 10 and restarts the trapping instruction. The processor's stacks before and after this thread migration are shown in Figure 6. At step 16, Processor 1 executes a `RET` instruction that causes the thread to migrate back to Processor 0. Figure 7 displays the processor's stacks before and after the return. Note that in between the two migrations there are two function calls that are entirely local to Processor 1.

[1] Obviously, on a real processor, thread migration takes much longer than a couple of cycles.

```
000:  ENTRY  2           % allocate space for 2 locals
001:  BNEQ   r0,4        % test t for NULL pointer
002:  SET    r0,0
003:  RET
004:  STORE  [fp+1],r0   % save t in stack frame
005:  LOAD   r0,[r0+1]   % r0 := t->left
006:  CALL   0           % call TreeAdd on t_left (in r0)
007:  STORE  [fp+2],r0   % save TreeAdd(left) in stack frame
008:  LOAD   r0,[fp+1]
009:  LOAD   r0,[r0+2]   % r0 := t->right
010:  CALL   0           % call TreeAdd on t_right (in r0)
011:  LOAD   r1,[fp+1]
012:  LOAD   r1,[r1]     % r1 := t->val
013:  ADD    r0,r0,r1
014:  LOAD   r1,[fp+2]
015:  ADD    r0,r0,r1
016:  RET
```

Figure 3: `TreeAdd` machine code

### 3.3 Allocation

One of the nice aspects of this mechanism is that it handles remote 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. By inline expanding the allocation code, we can insure that the object initialization is also completed before the thread migrates back to the processor that initiated the allocation. If the allocation is done in a library routine, then the thread will migrate back when the allocation routine returns. This will force an additional migration when the allocating processor attempts to initialize the remote data.

## 4 Introducing parallelism

While the migration scheme described in the previous section provides a mechanism for operating on distributed data in an SPMD machine, 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 chops the thread of execution into many pieces that can be executed out of order, thus the introduction of continuation capturing must be based on analysis of the program. We postpone the discussion of the compiler technology until the next section.

### 4.1 Futures

Our continuation capturing mechanism is essentially a variant of the *future* mechanism found in many parallel Lisps [Hal85]. In the traditional Lisp context, the expression `(future` $e$`)` is an annotation to the system that says that $e$ can be evaluated in parallel with its context. The result of this evaluation is a *future cell* that serves as a synchronization point between the child thread that is

| | Processor 0 | | Processor 1 | |
|---|---|---|---|---|
| **Step** | PC | Instruction | PC | Instruction |
| 0 | 0 | `ENTRY 2` | | |
| 1 | 1 | `BNEQ r0,4` | | |
| 2 | 4 | `STORE [fp+1],r0` | | |
| 3 | 5 | `LOAD r0,[r0+1]` | | |
| 4 | 6 | `CALL 0` | | |
| 5 | 0 | `ENTRY 2` | | |
| 6 | 1 | `BNEQ r0,4` | | |
| 7 | 4 | `STORE [fp+1],r0` | | |
| 8 | 5 | `LOAD r0,[r0+1]` | | |
| 9 | | *send to 1* | | |
| 10 | | | | *receive from 0* |
| 11 | | | 5 | `LOAD r0,[r0+1]` |
| 12 | | | 6 | `CALL 0` |
| 13 | | | 0 | `ENTRY 2` |
| 14 | | | 1 | `BNEQ r0,4` |
| 15 | | | 2 | `SET r0,0` |
| 16 | | | 3 | `RET` |
| 17 | | | 7 | `STORE [fp+2],r0` |
| 18 | | | 8 | `LOAD r0,[fp+1]` |
| 19 | | | 9 | `LOAD r0,[r0+2]` |
| 20 | | | 10 | `CALL 0` |
| 21 | | | 0 | `ENTRY 2` |
| 22 | | | 1 | `BNEQ r0,4` |
| 23 | | | 2 | `SET r0,0` |
| 24 | | | 3 | `RET` |
| 25 | | | 11 | `LOAD r1,[fp+1]` |
| 26 | | | 12 | `LOAD r1,[fp+1]` |
| 27 | | | 13 | `ADD r0,r0,r1` |
| 28 | | | 14 | `LOAD r1,[fp+2]` |
| 29 | | | 15 | `ADD r0,r0,r2` |
| 30 | | | 16 | `RET` |
| 31 | | | | *send to 1* |
| 32 | | *receive from 1* | | |
| 33 | 7 | `STORE [fp+1],r0` | | |
| | | ... | | |

Figure 4: Execution trace for TreeAdd Example

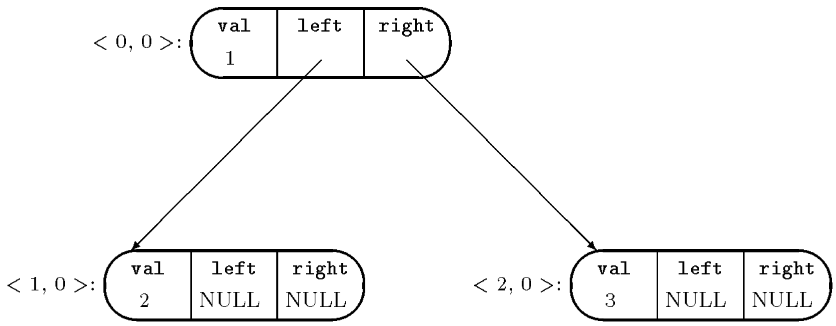



Figure 5: A simple distributed tree

evaluating $e$ and the parent thread. If the parent attempts to read the value (called *touching*) of the future cell 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.[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*. To make this concrete, consider the `TreeAdd` example again. Figure 8 gives pseudo-C code for the program annotated with futures. In this example, the `futurecall` operation takes a procedure and argument list and calls the procedure while saving the return context in the work list. The result of a `futurecall` is a future cell; the operation `touch` returns the value in a cell, blocking if necessary. Consider the example tree given in Figure 5 again. The first call to `TreeAdd` does a future call to `TreeAdd` on the left child (which is on Processor 2). When the second call traps on the reference to `t->left`, the thread migrates to Processor 2, leaving Processor 1 idle. At this point, Processor 1 can resume execution by storing an empty future cell in `f_left` and continuing execution with the right subtree. When the original thread returns from Processor 2, it will fill in the future cell and exit.

## 4.2 Implementation of futures

Our scheme for introducing parallelism consists of three operations: `futurecall`, `touch`, and `dispatch`. The first two of these are introduced into the code by the compiler, the `dispatch` operation is used internally when the processor needs something to do. In the remainder of this section, we describe these operations and their related data structures in detail.

The two data structures are the future cells and the future stack, which is a stack of future cells that serves as work list. A future cell can be in one of four states:

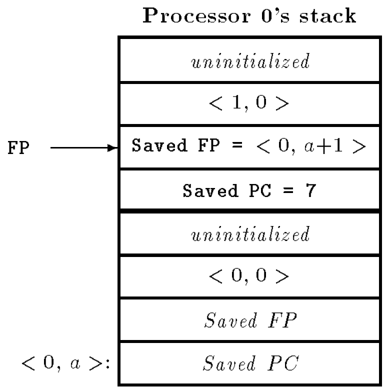
**Empty** This is the initial state and contains the information necessary to restart the parent.

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

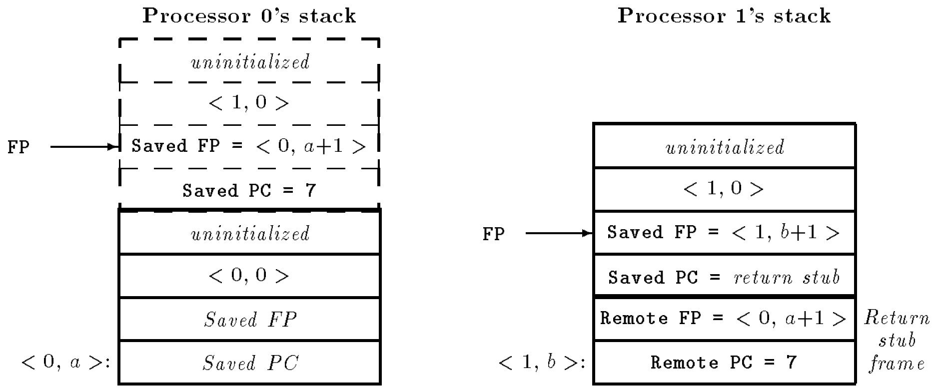
**Waiting** This is the state when the parent continuation has been stolen and subsequently touched the future. It contains the continuation of the blocked thread (Note: when inserting futures, the compiler insures that no future can be touched by more than one thread).

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

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




(a) Before migration.



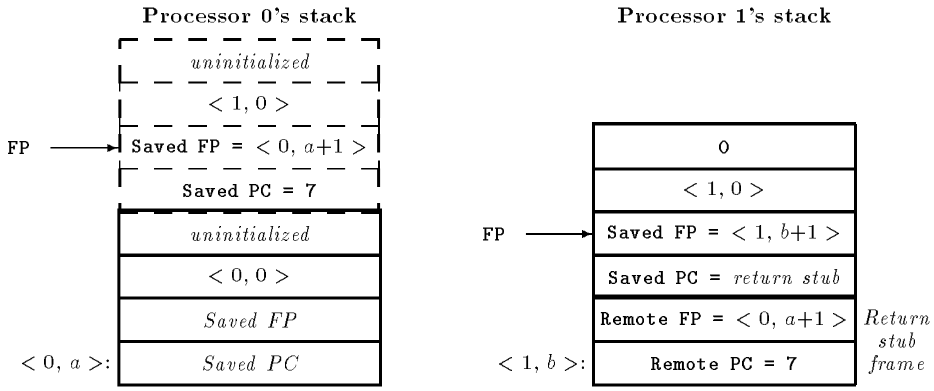

(b) After migration.

Figure 6: Thread migration on memory trap

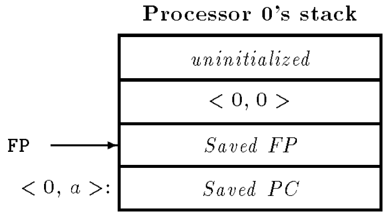
Figure 9 gives the allowable state transitions for a future cell. The future stack can be allocated as part of the processor's stack. In the normal case, when no migration or stealing occurs, only a few extra instructions are required to allocate the future cell and touch the result.

The `futurecall` operation is responsible for allocating a new future cell, pushing the cell on the future stack, and calling the given routine. Upon return from the routine, the cell is filled with the return result following the 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 `dispatch` 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 `dispatch` operation. This operation 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`.




(a) Before migration.




(b) After migration.

Figure 7: Thread migration on return

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 `dispatch`. Because the compiler rather than the user inserts futures and touches, we know that only only one touch will be attempted per future and that touch will be done by the future's parent thread of computation. This implies that the state of a future will be either **Stolen** or **Full** when a touch is attempted.

Figure 10 contains pseudo-code for `futurecall`, `touch`, and `dispatch`. Note that we parameterize the `futurecall` operation by the return program counter (`PC`) and frame pointer (`FP`) of the `futurecall` site. The operation `throw` transfers control to a continuation.

## 4.3 `TreeAdd` revisited

We now present idealized machine code and execution trace for the `TreeAdd` example from Section 3.2 augmented with futures. Figure 11 contains the idealized machine code.

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

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

        t_right = t->right;              /* this can trap */
        f_right = futurecall (TreeAdd, t_left);

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

Figure 8: `TreeAdd` C code with futures

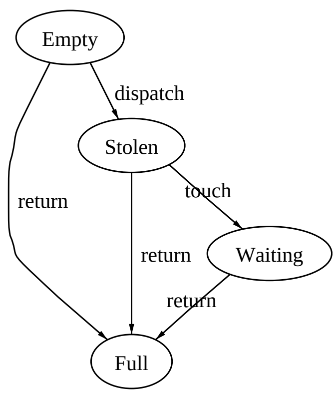



Figure 9: Future cell state transitions

```
futurecall (f, args, PC, FP):
    cell := new future cell
    cell.state := Empty
    cell.cont := <PC, FP, Registers>
    push  cell on the Continuation stack
    call  f on  args
    cell.val := result of  f
    case cell.state
      of Empty => (cell.state := Full; pop cell)
       | Stolen => (cell.state := Full; dispatch())
       | Waiting => (cell.state := Full; throw(cell.cont))
end


touch (cell):
    if (cell.state = Full)
        return cell.val
    else {
        cell.state := Waiting
        cell.cont := <PC, FP, Registers>
        dispatch();
    }
end


dispatch():
    if (the Continuation stack is empty)
        receive a message
    else {
      /* execute the next parent continuation */
        cell := pop from the Continuation stack
        cell.state := Stolen;
        throw (cell.cont)
    }
end
```

Figure 10: Pseudo-code for futures

```
000:  ENTRY  2
001:  BNEQ   r0,4
002:  SET    r0,0
003:  RET
004:  STORE  [fp+1],r0  % save t in stack frame
005:  LOAD   r0,[r0+1]  % r0 := t->left
006:  FUTURE 9
007:  CALL   0          % call TreeAdd on t_left (in r0)
008:  BACKTO
009:  STORE  [fp+2],r0  % save left future on stack
010:  LOAD   r0,[fp+1]
011:  LOAD   r0,[r0+2]  % r0 := t->right
012:  FUTURE 15
013:  CALL   0          % call TreeAdd on t_right (in r0)
014:  BACKTO
015:  LOAD   r1,[fp+2]  % retrieve left future
016:  TOUCH  r1,r1      % touch left future
017:  TOUCH  r0,r0      % touch right future
018:  ADD    r0,r0,r1
019:  LOAD   r1,[fp+1]
020:  LOAD   r1,[r1]    % r1 := t->val
021:  ADD    r0,r0,r1
022:  RET
```

Figure 11: `TreeAdd` augmented with Futures

This code contains three instructions that are used to implement our future mechanism: `FUTURE`, `BACKTO`, and `TOUCH`. A `futurecall` of a procedure *f* is translated into the following code sequence:

```
    set up arguments
    FUTURE  L
    CALL    f
    BACKTO
L:  % r0 is the future cell
```

The instruction "`FUTURE L`" adds a new future cell to the continuation stack, initializes the cell with a continuation formed from the code address `L`, the current registers and frame pointer. Following the return from *f*, the `BACKTO` instruction fills in the result of the future with the value in register `r0` and checks the state of future cell. If the future cell was **Empty**, then execution proceeds normally; otherwise, a new thread must be dispatched.[3] The instruction "`TOUCH Ra,Rb`" touches the future in `Rb`, putting the result in register `Ra`. The complete execution trace of `TreeAdd` running on a three processor machine for the tree given in Figure 5 is given in Figure 12.

## 4.4 Stack management

In the discussion so far, we have glossed over certain details related to stack management. When a continuation is stolen, the portion of the stack between the frame belonging to the stolen continuation and the frame that migrated[4] must be preserved for when the migrated thread returns. The problem is that stolen continuation may allocate new stack frames, overwriting the frames of the migrated thread. To avoid this problem we *split* the stack by copying the frames above the stolen continuation. We choose to copy the frames belonging to the migrated continuation, since we suspect that typically

[3] This can be optimized to just handle the non-stolen case (see page 15).

[4] Note that a continuation can only be stolen if some descendent of the `futurecall` migrates.

| | **Processor 0** | | **Processor 1** | | **Processor 2** | |
|---|---|---|---|---|---|---|
| **Step** | PC | Instruction | PC | Instruction | PC | Instruction |
| 0 | 0 | ENTRY 2 | | | | |
| 1 | 1 | BNEQ r0,4 | | | | |
| 2 | 4 | STORE [fp+1],r0 | | | | |
| 3 | 5 | LOAD r0,[r0+1] | | | | |
| 4 | 6 | FUTURE 9 | | | | |
| 5 | 7 | CALL 0 | | | | |
| 6 | 0 | ENTRY 2 | | | | |
| 7 | 1 | BNEQ r0,4 | | | | |
| 8 | 4 | STORE [fp+1],r0 | | | | |
| 9 | 5 | LOAD r0,[r0+1] | | | | |
| 10 | | *send to 1* | | | | |
| 11 | | *steal* | | *receive from 1* | | |
| 12 | 9 | STORE [fp+2],r0 | 5 | LOAD r0,[r0+1] | | |
| 13 | 10 | LOAD r0,[fp+1] | 6 | FUTURE 9 | | |
| 14 | 11 | LOAD r0,[r0+2] | 7 | CALL 0 | | |
| 15 | 12 | FUTURE 15 | 0 | ENTRY 2 | | |
| 16 | 13 | CALL 0 | 1 | BNEQ r0,4 | | |
| 17 | 0 | ENTRY 2 | 2 | SET r0,0 | | |
| 18 | 1 | BNEQ r0,4 | 3 | RET | | |
| 19 | 4 | STORE [fp+1],r0 | 8 | BACKTO | | |
| 20 | 5 | LOAD r0,[r0+1] | 9 | STORE [fp+2],r0 | | |
| 21 | | *send to 2* | 10 | LOAD r0,[fp+1] | | |
| 22 | | *steal* | 11 | LOAD r0,[r0+2] | | *receive from 0* |
| 23 | 15 | LOAD r1,[fp+2] | 12 | FUTURE 15 | 5 | LOAD r0,[r0+1] |
| 24 | 16 | TOUCH r1,r1 | 13 | CALL 0 | 6 | FUTURE 9 |
| 25 | | | 0 | ENTRY 2 | 7 | CALL 0 |
| 26 | | | 1 | BNEQ r0,4 | 0 | ENTRY 2 |
| 27 | | | 2 | SET r0,0 | 1 | BNEQ r0,4 |
| 28 | | | 3 | RET | 2 | SET r0,0 |
| 29 | | | 14 | BACKTO | 3 | RET |
| 30 | | | 15 | LOAD r1,[fp+2] | 8 | BACKTO |
| 31 | | | 16 | TOUCH r1,r1 | 9 | STORE [fp+2],r0 |
| 32 | | | 17 | TOUCH r0,r0 | 10 | LOAD r0,[fp+1] |
| 33 | | | 18 | ADD r0,r0,r1 | 11 | LOAD r0,[r0+2] |
| 34 | | | 19 | LOAD r1,[fp+1] | 12 | FUTURE 15 |
| 35 | | | 19 | LOAD r1,[r1] | 13 | CALL 0 |
| 36 | | | 21 | ADD r0,r0,r1 | 0 | ENTRY 2 |
| 37 | | | 22 | RET | 1 | BNEQ r0,4 |
| 38 | | | | *send to 0* | 2 | SET r0,0 |
| 39 | | *receive from 1* | | | 3 | RET |
| 40 | 8 | BACKTO | | | 14 | BACKTO |
| 41 | 16 | TOUCH r1,r1 | | | 15 | LOAD r1,[fp+2] |
| 42 | 17 | TOUCH r0,r0 | | | 16 | TOUCH r1,r1 |
| 43 | | | | | 17 | TOUCH r0,r0 |
| 44 | | | | | 18 | ADD r0,r0,r1 |
| 45 | | | | | 19 | LOAD r1,[fp+1] |
| 46 | | | | | 19 | LOAD r1,[r1] |
| 47 | | | | | 21 | ADD r0,r0,r1 |
| 48 | | | | | 22 | RET |
| 49 | | | | | | *send to 0* |
| 50 | | *receive from 2* | | | | |
| 51 | 14 | BACKTO | | | | |
| 52 | 17 | TOUCH r0,r0 | | | | |
| 53 | 18 | ADD r0,r0,r1 | | | | |
| 54 | 19 | LOAD r1,[fp+1] | | | | |
| 55 | 19 | LOAD r1,[r1] | | | | |
| 56 | 21 | ADD r0,r0,r1 | | | | |
| 57 | 22 | RET | | | | |

Figure 12: Execution trace for `TreeAdd` augmented with futures

there will only a few of these. This adds a minor complication to the return migration: instead of using an explicit return frame pointer, we use a unique ID that is mapped back to the current position of the stack when a return message is received.

We can exploit this copying to optimize the return of a `futurecall` for the normal (non-stolen) case. The idea is that when copying the portion of the stack belonging to the migrated thread, we replace the return PC with the address of a piece of code that tests the state for **Stolen** or **Waiting** and takes the appropriate action. In effect, we are encoding part of the future cell's state (i.e., is it **Empty** or not) in the return PC. Since the return PC installed by the `futurecall` will only be in place if the future has not been stolen, it is not necessary to test the future cell's state.

# 5 Compiler Analyses and Parallelizing Transformations

In the previous two sections, we have outlined a mechanism for thread migration based on continuations, and a technique, based on futures, for introducing parallelism into the computation. In this section, we describe some compiler analysis techniques that can be used to determine when it is safe to introduce `futurecall` operations. We also discuss compiler transformations that can increase the amount of parallel computation.

In programs that use recursively-defined dynamic data structures, a fundamental source of parallelism is the *divide-and-conquer* paradigm in which a function is recursively applied to independent sub-pieces of the data structure. As illustrated by the `TreeAdd` program given in Figure 2, this type of parallelism is often available in programs operating on tree-like or list-like data structures. In general, a divide-and-conquer algorithm has three phases: *pre-computation*, *divide-and-conquer*, and *post-computation*. Figure 13 gives this general form for tree-like data structures.

```
int DivideConquer (tree *t)
{
   if (t != NULL) {
     /* pre-computation */
       ...

     /* divide-and-conquer  */
       subtree_1 = t->s1;
       t_1 = DivideConquer (subtree_1);
       ...
       subtree_n = t->sn;
       t_n = DivideConquer (subtree_n);

     /* post-computation */
       ...

       return result;
   }
   else
       return 0;
}
```

Figure 13: An abstract divide-and-conquer procedure

In addition to the `TreeAdd` example given above, we provide two further examples of divide-and-conquer problems, `ReverseTree` and `SumAll`, in Figure 14. Unlike `TreeAdd`, which does not have side effects, the `ReverseTree` example is an imperative procedure that recursively reverses the children at each level of a binary tree. The `SumAll` procedure is used to illustrate our techniques on list-like structures. At each step, the function SumList is applied to the head of the list, and the function SumAll is recursively applied to the tail of the list.

```
void ReverseTree (tree *t)
{
    if (t != NULL) {
        tree *t_left, *t_right;

      /* pre-computation (none) */

      /* divide-and-conquer */
        t_left = t->left;
        ReverseTree (t_left);
        t_right = t->right;
        ReverseTree (t_right);

      /* post-computation */
        t->left = t_right;
        t->right = t_left;
    }
}
```

```
int SumAll (list *l)
{
    if (l != NULL) {
      /* pre-computation */
        l_head = l->head;
        head_sum = SumList(l_head)

      /* divide-and-conquer */
        l_tail = l->tail;
        rest_sum = SumAll(l_tail);

      /* post-computation */
        return (col_sum + rest_sum);
    }
    else
    return(0);
}
```

Figure 14: Additional program examples

We can summarize the major steps in parallelizing sequential procedures as follows:

1. determine the three phases of the function,
2. test if the divide-and-conquer calls can be issued in parallel and insert the appropriate future calls and return values,
3. determine if all or part of the pre-computation can be moved after the divide-and-conquer phase, and
4. insert the appropriate touch operations to enforce synchronization.

In the following sections, we discuss each of these steps in more detail using our three example programs to illustrate the process.

## 5.1 Determining the three phases

Given a procedure body, we first locate the recursive procedure calls. These call statements, along with the immediate calculations of their arguments form the main divide-and-conquer part of the procedure. The appropriate statements may be selected by performing a program slice[HRB90] relative to the arguments of the procedure calls. The statements that occur before the procedure calls, and that are not included in the slice, form the pre-computation. And the statements that follow the recursive calls form the post-computation.

## 5.2 Inserting futures

Once the recursive calls have been identified, the major problem is to determine whether or not the recursive calls *interfere*. If the recursive calls do not interfere, then they are candidates for future insertion. In general, we say that two statements *interfere* if one statement reads or writes a location that the other statement writes. Let us consider a typical function call statement, $S$, as follows:

$$res = f(t_1, \ldots, t_m, v_1, \ldots, v_n)$$

in which a scalar variable $res$ is assigned the resulting value of a call to function $f$ with sub-tree pointer arguments $t_1$, ..., $t_m$ and scalar arguments $v_1$, ..., $v_n$. If we could describe the set of locations read and written for each such call, then we would be able to determine whether or not two function calls interfere. This problem of computing read and write sets can be divided into two sub-problems: (1) the set of scalar (stack-allocated) locations read and written, and (2) the set of pointer (heap-allocated) locations read and written.

The first sub-problem of calculating locations read and written by scalar variables is very straightforward. Since all scalars read or written by the called function, $f$, are guaranteed to be local to $f$ and not visible to any other function call, we only need to include the scalars read or written directly by the call statement itself. Assuming a call-by-value parameter mechanism, we can associate each variable name with an abstract location, and define the following *ReadScalar* and *WriteScalar* sets for $S$:

$$\begin{aligned} \mathit{ReadScalar}(S) &= \{v_1, \ldots, v_n\} \\ \mathit{WriteScalar}(S) &= \{res\} \end{aligned}$$

The second sub-problem of calculating locations read and written by pointer variables is substantially harder. These pointer variables refer to objects in the heap (which in turn may point to other objects in the heap). Thus, we need to include not only the reads and writes performed by the call statement, but also the reads and writes to the heap that are performed by executing the called function. One way of estimating these reads and writes is to associate the pointer variable name $t_i$ with a set of heap locations. More precisely, we say that abstract location $t_i$ is *read* if the node pointed to by $t_i$ is read during the execution of the call to $f$, **or** if any node reachable from $t_i$ is read. Similarly, we say that abstract location $t_i$ is *written* if the the node pointed to by $t_i$ is written **or** any node reachable from $t_i$ is written. Using this notion of abstract location, we define the following *ReadHeap* and *WriteHeap* sets for $S$. Note that the notation read$(t_i, S)$ denotes that execution of the call $S$ reads abstract location $t_i$, and the notation $t_i \leadsto p$ means that there exits a path from $t_i$ to $p$.

$$\begin{aligned} \mathit{ReadHeap}(S) &= \{t_i \mid \mathrm{read}(t_i, S) \vee (\mathrm{read}(p, S) \wedge t_i \leadsto p)\} \\ \mathit{WriteHeap}(S) &= \{t_i \mid \mathrm{write}(t_i, S) \vee (\mathrm{write}(p, S) \wedge t_i \leadsto p)\} \end{aligned}$$

To make the idea of read and write sets more concrete, consider our initial program example `TreeAdd`. Since the procedure `TreeAdd` only reads values from the tree, we can easily compute the following read and write sets.

| | *ReadScalar* | *WriteScalar* | *ReadHeap* | *WriteHeap* |
|---|---|---|---|---|
| `r_left = TreeAdd(t_left)` | {} | {`r_left`} | {`t_left`} | {} |
| `r_right = TreeAdd(t_right)` | {} | {`r_right`} | {`t_right`} | {} |

In order to determine if the two procedure calls interfere we must check that one procedure call does not read or write a location that the other procedure writes. In this example there is clearly no interference due to scalar variables (the first call writes `r_left` while the second call writes a different location `r_right`). Furthermore, since both of the *WriteHeap* sets are empty, there cannot be any interference due to heap variables. Given these two facts, the two recursive calls to `TreeAdd` may be executed safely in parallel.

Now let us consider a more complicated example, `ReverseTree`. By analyzing the body of `ReverseTree` we note that it both reads and writes locations in the tree. This gives us the following read and write sets for the two recursive calls.

| | *ReadScalar* | *WriteScalar* | *ReadHeap* | *WriteHeap* |
|---|---|---|---|---|
| `ReverseTree(t_left)` | {} | {} | {`t_left`} | {`t_left`} |
| `ReverseTree(t_right)` | {} | {} | {`t_right`} | {`t_right`} |

In the case of ReverseTree, note that there are no scalar accesses, and so interference cannot occur due to scalars. The situation for heap references is more complicated. Here we find that the call ReverseTree(t_left) both reads and writes locations reachable from t_left, and the call ReverseTree(t_right) both read and writes locations reachable from t_right. Therefore, to determine whether these two calls interfere or not, we must be able to determine if all nodes accessible from t_left are distinct from all nodes accessible from t_right. This information is available through the use of *path matrix* analysis [Hen90, HN90]. Path matrix analysis computes, for each point in the program, a summary of all possible paths between each pair of handles (pointer variables live at that program point). In addition, the analysis determines if the data structures are tree-like (each node has at most one parent, and there are no cycles). Using path matrix analysis we can calculate the following path matrix for the program point just before the two recursive calls to ReverseTree.

| | t | t_left | t_right |
|---|---|---|---|
| t | $S$ | $L^1$ | $R^1$ |
| t_left | | $S$ | |
| t_right | | | $S$ |

Using the information in this path matrix we note that: (1) at this point in the procedure the data structure is a tree (there are no entries for DAG nodes[5] in the matrix), (2) the only paths are an $L^1$ (exactly 1 left link) path from t to t_left and an $R^1$ (exactly 1 right link) path from t to t_right, and (3) there is no path between t_left and t_right (the entries $PM$[t_left, t_right] and $PM$[t_right, t_left] are empty). This information from the path matrix guarantees that for *every* recursive invocation of ReverseTree, the two sub-trees t_left and t_right are independent and therefore the two recursive procedure calls may be processed in parallel.

For our third example, SumAll we note that there is only one recursive call to SumAll and we will leave the discussion of how to handle this case to the next subsection on reordering the pre-computation.

Once we have determined that the recursive calls can be safely executed in parallel, we may replace the procedure calls with futurecall operations. Figure 8 shows the result of this transformation for the TreeAdd example. In this case, the statements

```
r_left = TreeAdd(t_left);
r_right = TreeAdd(t_right);
```

are replaced by

```
f_left = futurecall(TreeAdd,t_left);
f_right = futurecall(TreeAdd,t_right);
```

Note that the integer result values r_left and r_right are replaced by the future cells, f_left and f_right.

Figure 15 gives the transformation for the imperative procedure ReverseTree. In this case the original procedure (Figure 14) did not have a return value. In order to introduce the futurecall operations we have introduced a dummy return value, and the appropriate future-cell variables, f_left and f_right.

It should be noted that in these compilation methods we are assuming that either: all recursive calls will access non-local sub-trees, or all recursive calls will access local sub-trees. Given this assumption, the order in which the future calls are placed is not important.

[5] A DAG node is any node that has more than one parent. If the data structure is tree-like, then no such node will exist, and the path matrix will contain only the ordinary handles. If DAG nodes do exist, then there will be special *DAG-handles* that appear in the path matrix. In this case there are no special DAG-handles, and therefore the structure is guaranteed to be tree-like.

```
int ReverseTree(tree *t)
{
    if (t != NULL) {
        tree *t_left, *t_right;
        future_cell f_left, f_right;

      /* pre-computation (none) */

      /* divide-and-conquer */
        t_left = t->left;
        f_left = futurecall (ReverseTree, t_left);
        t_right = t->right;
        f_right = futurecall (ReverseTree, t_right);

      /* post-computation */
        t->left = t_right;
        t->right = t_left;

        touch (f_left);
        touch (f_right);
    }

    return 0;
}
```

Figure 15: `ReverseTree` augmented with futures

Note, however, that it is also possible that the programmer chooses to map a program's data structures in another way; in that case, the order in which the future calls are listed may be of importance. Since our model specifies that the future computation is executed immediately, we would like to list the future calls that are most likely to migrate to other processors first, and those that are more likely to stay on the current processor last. This strategy exposes more parallelism since the first futures will move to other processors quickly leaving the current processor available to process the later futures or the post-computation. It is therefore important, in our next phases of research, to explore different programmer-defined mapping strategies that could aid in the ordering of future calls.

## 5.3 Reordering the pre-computation

In the previous section, we have illustrated how we can expose parallelism among the recursive calls. In this section we explore how we can expose parallelism between the recursive calls and the pre-computation. Let us consider the example `SumAll` from Figure 14. Since this procedure is operating on a list, we only find one recursive call, and therefore, even if we make this call a `futurecall`, we do not expose any parallelism. However, we note that there is a significant pre-computation in the form of the statement `head_sum = SumList(l_head)`. If we can overlap this pre-computation with the recursive call `rest_sum = SumAll(l_tail)`, then we can expose some parallelism. We can determine that these two computations do not interfere by examining the following read and write sets.

| | *ReadScalar* | *WriteScalar* | *ReadHeap* | *WriteHeap* |
|---|---|---|---|---|
| `head_sum = SumList(l_head)` | {} | {`head_sum`} | {`l_head`} | {} |
| `rest_sum = SumAll(l_tail)` | {} | {`rest_sum`} | {`l_tail`} | {} |

Given the fact that the pre-computation does not interfere with the recursive call, we can safely move the pre-computation after the future recursive call as indicated in Figure 16. This exposes parallelism because when the future call migrates to another processor, the pre-computation can be executed in parallel.

```
int SumAll (list *l)
{
    if (l != NULL) {
      /* divide-and-conquer */
        l_tail = l->tail;
        rest_sum = futurecall (SumAll, l_tail);

      /* pre-computation (moved after the divide-and-conquer) */
        l_head = l->head;
        head_sum = SumList (l_head)

      /* post-computation */
        return (col_sum + touch (rest_sum));
    }
    else
        return 0;
}
```

Figure 16: `SumAll` augmented with futures

## 5.4 Inserting touches

The final step in the parallelization is the insertion of the `touch` operations, which provide the synchronization between the parallel tasks that have been created by the future/migration mechanisms. The most straightforward way of introducing `touch` operations is to place one touch for each future cell immediately following the `futurecall` statments. This introduces a barrier at this point, and is guaranteed to be safe because we have checked that the recursive calls can be executed in parallel safely. But this may not be the approach that exposes the most parallelism. The best strategy is to place the `touch` operations as far down in the post-process section as possible, without violating any dependencies. For example, let us consider `ReverseTree` in Figure 15. In this case, the post-process statements do not depend on the results from the recursive calls and the touches can therefore be placed at then end of the post-process section. This allows the post-process statements to be executed in parallel with the recursive calls and it achieves an effect somewhat like the fuzzy barrier mechanism [GE90].

## 5.5 Parallelization Summary

In this section we have outlined the four major steps in analyzing and parallelizing recursive divide-and-conquer procedures: (1) identifying the three phases of the computation, (2) inserting futures for the recursive calls, (3) moving the pre-computation, and (4) inserting touch operations. In order to perform these four steps one requires an accurate alias and dependence analysis that can be used to determine when computations may be reordered or executed in parallel. We have outlined how one such analysis, path matrix analysis, can be used for programs operating on list or tree-like data structures. Other analysis methods that can handle more complex pointer data structures include a storeless analysis based on finite representations of right-rectangular equivalence relations [Deu92] or extended path matrix analysis [HHN92]. By employing one of these more advanced analyses one could also apply these techniques to more complex data structures.

# 6 Conclusions and Further Work

In this paper we have presented a new approach for automatically generating SPMD parallel programs from sequential programs that use dynamic 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.

Given these fundamental problems, we have proposed a new computation model that more closely matches 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 have suggested a thread migration strategy that allows the computation to migrate automatically to the processor that owns the data. We also noted that supporting thread migration is not enough, one must also introduce a new mechanism to expose parallelism. Thus, the second component of our approach is to introduce a continuation capturing future mechanism. By combining this future mechanism with notions of lazy task creation and thread migration we have provided a mechanism that allows many processors to operate on different pieces of a data structure in parallel.

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 recursive dynamic data structures. Using the dependency analysis we show how futures can be introduced automatically to expose the divide-and-conquer parallelism. In addition, we show how the computation can be reordered to expose parallelism between the body of the computation and the divide-and-conquer step.

Currently we have a small simulator that we have used to produce the execution traces provided in this paper. After further refinement of both the execution model and compilation methods, we intend to implement this model on a distributed memory machine, and perhaps examine how the thread migration model can be supported on some of the new multi-threaded architectures. In addition, we would like to study the interaction between different dynamic data structure mapping strategies and the compilation strategies used to expose parallelism.

## 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.

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

[Deu92] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13, April 1992.

[GE90] R. Gupta and M. Epstein. High speed synchronization of processors using fuzzy barriers. *International Journal of Parallel Programming*, 19(1), 1990.

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

[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.

[HDE+92] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured representations. ACAPS Technical Memo 46, McGill University, 1992.

[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.

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

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Languages and Systems*, 12(1):26–60, January 1990.

[HSSW92] L. J. Hendren, B. Sridharan, V. Sreedhar, and Y. Wong. The SIMPLE AST – McCAT compiler. ACAPS Technical Memo 36, McGill University, 1992.

[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.

[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.

[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.