McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Compiling C for the EARTH Multithreaded Architecture†

Laurie J. Hendren, Guang R. Gao, Xinan Tang,
Yingcun Zhu, Xun Xue‡,
Haiying Cai and Pierre Ouellet

ACAPS Technical Memo 101
March 18, 1996

†This work support, in part, by NSERC and FCAR. Also, many thanks to GMD-FIRST, Berlin, Germany for the use of their MANNA multiprocessor.

‡Currently at IBM Toronto

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7
FTP: ftp-acaps.cs.mcgill.ca • WWW: http://www-acaps.cs.mcgill.ca/

Abstract

Multithreaded architectures provide an opportunity for efficiently executing programs with irregular parallelism and/or irregular locality. This paper presents a strategy that makes use of the multithreaded execution model without exposing multithreading to the programmer.

Our approach is to design simple extensions to C, and to provide compiler support that automatically translates high-level C programs into lower-level threaded programs. In this paper we introduce EARTH-C, our extended C language which contains simple constructs for specifying control parallelism and data locality. Based on EARTH-C, we describe compiler techniques that are used for translating to lower-level Threaded-C programs for the EARTH multithreaded architecture.

We demonstrate our approach with six benchmark programs. We show that even naive EARTH-C programs can lead to reasonable performance, and that more advanced EARTH-C programs can give performance very close to hand-coded threaded-C programs.

Contents

List of Figures

List of Tables

1 Introduction

Multithreaded architectures provide one approach to high-performance parallel computing. By supporting many threads of control and fast switching among threads, multithreaded architectures can tolerate inherent communication and synchronization latencies by switching to a new ready thread of control whenever a long-latency operation is encountered [1, 4, 13, 17, 20]. As long as there is enough parallelism in an application, a multithreaded architecture can hide these latencies and effectively utilize available communication bandwidth. For a review of the principle of multithreading and some representative multithreaded architecture projects, readers are referred to [7].

However, providing support for multithreading at the architecture level is not enough, we must also be able to effectively program such architectures. One alternative is to make the threads and synchronization between threads explicit in the programming language, and ask the programmer to handle all of the details of thread creation, communication, and synchronization between threads. While this approach gives the full flexibility of multithreading to the user, it tends to introduce many low-level details that are hard for the programmer to handle. Thus, it is unlikely that such a programming model would become generally acceptable. A preferable approach is to give the programmer a more traditional high-level language that does not explicitly support threads, and to use selected language extensions and compilation techniques to automatically translate high-level programs into lower-level threaded programs. Thus, with minimal effort, the programmer should be able to benefit from at least some of the power of multithreading.

In this paper we present EARTH-C, our parallel dialect of C, and we address the problem of translating programs written EARTH-C into lower-level Threaded-C programs that run on the EARTH (Efficient Architecture for Running Threads) multiprocessor [11, 10, 15]. Our goal is to provide simple language extensions along with advanced compilation techniques that allow us to automatically produce reasonably efficient threaded programs. Our solution is based on the design of parallel extensions to C and compilation techniques.

Our main emphasis has been on compiling benchmarks that have irregular parallelism, and those that use dynamic data structures. These types of applications are difficult to deal with using standard techniques that rely on **static** computation and data partitioning. Our programs have inherently dynamic behaviour, and thus to execute our class of benchmarks, we need an architecture that can tolerate a reasonable amount of communication, and can adapt to uneven loads. This is precisely the kind of programs for which multithreaded architectures should be useful. However, as we noted earlier, we do not want to require the programmer to specify the threads explicitly. Rather, we want to capture the correct high-level language features that allow the compiler to generate the appropriate threads.

As multithreaded architectures use parallelism (multiple threads per processor) to mask communication latency, we need to be able to find all possible parallelism in a program. EARTH-C supports constructs for specifying control parallelism in the form of a parallel `forall` loop and parallel statement sequences. The programmer uses these constructs to specify coarse-grain parallelism, while the compiler uses dependence analysis to detect fine-grain parallelism. Our language only supports fork/join type parallelism where the parallel computations do not interfere (one computation cannot write a location read or written by the other computation). We have made this restriction to keep the language simple for the programmer, and to allow us to safely apply our advanced dependence analyses to the EARTH-C programs.

Even with multithreading, it is often advantageous to reduce (although not necessarily minimize) communication. Thus, the EARTH-C type system distinguishes between local and remote memory references. Thus, if the programmer wants to improve performance, he/she can use appropriate type declarations plus dynamic data and computation allocation to reduce communication. As multithreading can tolerate some communication, and some unevenness in load, the programmer need not spend a lot of effort to get the best distribution, a reasonable distribution should give reasonable performance.

The EARTH-C language extensions are not radical and should be easy for programmers to grasp. The main point is not the novelty of the extensions themselves, but rather the fact that such simple extensions are enough to allow the compiler to generate reasonable threaded programs.

We have implemented a version of the McCAT C compiler that accepts EARTH-C programs as input, and generates Threaded-C programs. Threaded-C is a lower-level language where the programmer specifies all threads and synchronization among threads [11, 10].[1] The McCAT compiler uses automatic parallelization techniques to augment the coarse-grain parallelism specified by the programmer. The compiler also uses loop transformations and thread-generation strategies to produce threaded code with the appropriate synchronization inserted. The EARTH run-time system handles the actual scheduling of threads and load-balancing among processors.

In order to evaluate our approach, we have examined six benchmark programs. We examine how much performance we can achieve with a *naive* and *advanced* EARTH-C version of the benchmark. The *naive* version corresponds to minimal programmer effort, where the programmer expresses simple control parallelism using EARTH-C constructs. The *advanced* version corresponds to more programmer effort, where the programmer refines the naive version by introducing the appropriate type declarations to expose data locality, and some dynamic data and computation allocation. For each benchmark we compare the speedup of the two EARTH C versions relative to a sequential version and a hand-coded low-level Threaded-C version. Our results show that reasonable speedup can be achieved by the naive EARTH-C versions, where only minimal programmer effort was required. However, in some cases, the additional effort spent developing the advanced EARTH-C version results in significantly better performance. Furthermore, the advanced EARTH-C versions come close to matching the performance of the hand-coded Threaded-C versions, but with less programmer effort than is required for the Threaded-C versions.

[1]See Figure 15(b) for an example of Threaded-C code.

The remainder of this paper is organized as follows. Section 2 gives an overview of our EARTH-C language, including several small examples. Section 3 provides an overview of the EARTH-McCAT compiler, and describes the parallelization and thread-generation strategies that we have developed. Our benchmarks and experimental results are presented in Section 4, a discussion of related work is given in Section 5, and conclusions are given in Section 6.

2 The EARTH-C Language

The design of EARTH-C has been driven by two factors: (1) the desire to keep the extensions to C simple, but yet expressive; (2) the need to to provide some information about control parallelism and data locality to the compiler. Another design criteria is that we want to be able to correctly compile C programs that contain no EARTH-C extensions. Thus, the programmer should be able to start with a sequential C program, and let the compiler produce a parallel threaded program. If the programmer is not satisfied with the performance, he/she can then incrementally improve the original C program by using the appropriate EARTH-C constructs that expose parallelism and improve locality. The following subsections describe the features of EARTH-C, and explain the purpose of the features.

2.1 Statement Sequences

EARTH-C supports three kinds of statement sequence: *ordinary*, *parallel*, and *strictly sequential*. Figure 1 illustrates the three cases. In ordinary sequences, denoted using the standard C block syntax `{ ... }`, statements may be executed in any order, as long as all data dependences are obeyed. In Figure 1(a) an ordinary sequence is given. In this case `a = f(b);` must precede `foo(a)` because of the flow-dependence on `a`. It is expected that programmers will usually use ordinary statement sequences, and leave it to the compiler to infer most of the statement-level parallelism. In the case of ordinary C (no EARTH-C constructs) programs as input, all statement sequences will be ordinary.

`{ a = f(b);`
`  foo(a);`
`  goo(b);`
`}`
	`{^ f(t->left);`
	`   f(t->right);`
	`^}`
		`{! a = f(x);`
		`   b = g(y);`
		`   foo(a,b);`
		`!}`

(a) ordinary	(b) parallel	(c) strictly sequential

Figure 1: Types of statement sequences

However, in some cases a programmer may want to explicitly specify parallel or strictly sequential sequences. Figure 1(b) gives a typical use of a parallel sequence, denoted using the syntactic construct `{^ ... ^}`. Parallel statement sequences are used to explicitly specify that there are no dependences among the statements, and they may be safely executed in parallel.

This construct is used primarily when the compiler's dependence analysis cannot detect the parallelism automatically. In Figure 1(b) it is used to specify that the function calls to `f` can be done in parallel (i.e. the programmer guarantees that there is no interference between the calls).

Figure 1(c) gives an example of a strictly sequential sequence, as denoted by the syntactic construct `{! ... !}`. In strictly sequential sequences, statements must be executed in their lexical order. They are used when the data dependences would allow the compiler to rearrange the statements, but the programmer wants to ensure a specific order. In Figure 1(c) the programmer is specifying that the call to `f` must be executed before the call to `g`, even though there is no dependence between the two calls.

The different sorts of statement sequences can be used to group any type of statements, including compositional statements such as `if`, `switch`, `while`, `for` and `do` statements. Figure 2(a) gives an example of two `for` loops that can execute in parallel. Furthermore, just like blocks in C, statement sequences can be arbitrarily nested. Figure 2(b) gives an example with a parallel sequence nested inside an ordinary sequence.

```
{^  for (i1=0; i1 < n/2; i1++)
      f(i1);
    for (i2=n/2+1; i2<n; i2++)
      f(i2);
^}
```

(a) sequence of `for` statements

```
{ t_l = t->left;
  t_r = t->right;
  {^ f(t_l);
     f(t_r);
  ^}
}
```

(b) nested sequences

Figure 2: Examples of statement sequences

2.2 Parallel Loops

EARTH-C supports a `forall` loop in which the header may have inter-iteration dependences, but the loop body must have no inter-iteration dependences[2]. We can express the `forall` loop construct as follows:

$$\texttt{forall}\ (\ v = \mathit{init\_expr}\ ;\ \mathit{test\_expr}\ (\ v\ );\ v = \mathit{update\_expr}\ (\ v\ )\ )$$
$$\mathit{loop\_body}\ (\ v\ )$$

The operational semantics of the `forall` loop can be stated as follows. Let $v_1 \ldots v_n$ be the values generated by executing the loop header. Then all loop bodies *loop_body*(v_1), *loop_body*(v_2), ..., *loop_body*(v_n) **may** be executed in parallel. Figure 3 gives two examples of `forall` loops. The first, in Figure 3(a), is a typical parallel loop in scientific computing, where the different iteration values for `i` can be computed with a closed form. The second example, in Figure 3(b), gives an example of traversing a linked list, where the different values of `p` must be calculated sequentially, but many iterations of the body may proceed in parallel.

[2] Dependences within an iteration are allowed as the compiler will perform the appropriate renaming in the parallelization phases.

`forall (i=0; i<N; i++)` `  { a[i] = b[i] * s; }`	`forall (p=head; p != NULL; p = p->next)` `  { f(p); }`
(a) completely parallel `forall`	(b) `forall` with sequential header

Figure 3: Examples of `forall` loops

2.3 Local and Remote Memory Accesses

In the underlying EARTH execution model we assume that memory is physically distributed among the processors, and so there is a crucial difference in the cost of *remote* memory accesses (on a different processor) and *local* memory accesses (memory is on the local processor). From the compiler's view, it is always safe to treat memory accesses as remote. However, for efficiency, the compiler should recognize when memory accesses are local whenever possible. If the programmer uses standard C declarations using no EARTH-C extensions, then the compiler assumes the following:

Global Variables: All global variables are by default allocated on processor 0. Thus, all direct references (i.e. references via the **name** of the variable, and not indirect references through pointers) to global variables are considered to be remote.

Formal Parameters and Local Variables: All formal parameters and local variables for a function invocation are allocated in an activation frame that resides in the memory of the processor executing the invocation. Thus, all direct references to parameters and local variables are to local memory.

Indirect References via Pointers: Pointer dereferences may refer to globals, to local variables outside of the current function activation, or to heap-allocated objects. Thus, the most conservative assumption is that all pointer dereferences refer to remote memory. Of course, pointer analyses can be used to improve upon this conservative rule.

With these assumptions, the programmer can write their program using ordinary C declarations, and the compiler is guaranteed to produce correct, although possibly inefficient, code. Consider the example in Figure 4. In `main`, the references to local variables `i` and `zeros` are local memory accesses, but the references to the global array `a` are remote memory accesses.[3] In `count_k`, references to formal parameters `n` and `k`, and to locals `i` and `count` are all local memory accesses. However, the indirect accesses via pointer `x` are remote memory accesses.

Note that the communication in EARTH-C is implicit. For example, in the program in Figure 4, the programmer referenced remote memory read via the use of `x[i]` and a remote memory write via the assignment to `a[i]`. However, the underlying communication is hidden, and it is the EARTH-C compiler's responsibility to produce the correct threads and communication code.

[3] Of course, if the compiler can determine that `main` is always invoked on processor 0, then the reference to `a` could be considered a local memory access.

```
int a[10]; /* global variable declaration */

/* find number of times k occurs in the first n items of array x */
int count_k(int *x, int n, int k)
{ int i, count=0;
  for(i=0; i<n; i++) if (x[i] == k) count++;
  return(count);
}

int main()
{ int i, zeros;

  /* init array a */
  for(i=0; i<10; i++) a[i] = init_val();

  /* find number of zeros in a */
  zeros = count_k(a,10,0);
}
```

Figure 4: Example of local vs remote memory accesses

We have extended the type system in EARTH-C to allow the user to provide additional information about data mapping and locality. There are 2 extensions, the first extends the type declarations for global variables, while the second gives new types for pointers.

2.3.1 Global Variables

Global variables can be declared as *ordinary* global variables or *replicated* global variables. An ordinary global variable has only **one** copy, and by default it is located on processor 0. If the user wishes to allocate the variable on another processor, then they must provide the processor number along with the declaration. Consider the declaration of globals `a` and `b` in Figure 5(a). Array `a` will be allocated in processor 0, whereas array `b` will be allocated in processor `MAXP`. Accesses to ordinary globals are local from the processor on which they reside, but remote from all other processors.

A replicated global variable is indicated by the storage class keyword `replicated`, and a copy of the global variable is allocated on each processor. An example is the declaration of array `c` in Figure 5(a). Accesses to a replicated global are always made to the local copy of the global, and so the compiler considers them to be local accesses. Note that once declared as replicated global variables, the compiler treats each copy of the global as a distinct variable, and so there is no consistency between the copies.

2.3.2 Pointers

The rules for locally-scoped and global variables allow the compiler to determine if a direct memory reference is local or remote. In summary, references to locally-scoped variables (local declaration or formal parameter), and replicated globals are local references, while references to ordinary globals are remote references. However, this does not solve the problem when memory references are made via indirections (i.e. `*p`, `p->f`, or `p[i]`, where `p` is a pointer type). In order to be able to express the case when a pointer is guaranteed to point to local memory, we have introduced an extension to pointer types in C. Each pointer type can be either *ordinary* or *local*. An ordinary pointer type is assumed to point to a remote location, whereas a local pointer type is guaranteed to point to a local location. Local pointers are created by using `* local` instead of `*` in type declarations. Thus, in Figure 5(b) we are declaring `p` to be an ordinary pointer, `q` to be a local pointer, `r` to be an ordinary pointer to an ordinary pointer, `s` to be a local pointer to an ordinary pointer, and `t` to be a local pointer to a local pointer. Thus, assuming that `p`, `q`, `r`, `s` and `t` are locally-scoped variables, the memory accesses to `p`, `q`, `*q`, `r`, `s`, `*s`, `t`, `*t` and `**t` would be local, whereas the memory accesses to `*p`, `*r`, `**r` and `**s` would be remote. A typical example of the use of local pointers is given in Figure 6. In this case, the parameter `root` in the function `incr_tree` can be declared as a local pointer since it `incr_tree` is always invoked on the processor owning the node pointed to by `root`. We will return to this example when discussing function invocations in Section 2.4.

`int a[4], b[4]@MAXP;`
`replicated int c[4];`
	`int * p;`
	`int * local q;`
	`int * * r;`
	`int * * local s;`
	`int * local * local t;`

(a) global declarations	(b) local-pointer declarations

Figure 5: Examples of declarations

2.4 Remote and Local Functions

In order to support parallel execution of programs, it is necessary to be able to invoke functions on different processors. In fact, function invocation is the only way for an EARTH-C programmer to explicitly send computation to another processor.

Remote functions are declared as normal C functions, and these functions may be invoked on any processor, and the function invocation may access remote memory locations. If a call to a remote function does not specify on which processor to invoke the function, then the run-time load balancer makes the assignment to a processor. If the programmer wants to invoke a function on a specific processor, then the call may decorated with an expression that denotes the processor number (i.e. `foo(a1,a2)@p`, `bar(a1,a2)@my_ld_balancer()` or `baz(a1)@OWNER_OF(a1)`). The expression after the `@` symbol must denote an integer value where a positive value, $p \geq 0$, will cause the function to be invoked on processor p, whereas a negative value, $p < 0$, indicates that the run-time load balancer should be used to select the appropriate processor.

Of course, there is a substantial overhead associated with invoking functions on other processors, so we have defined *local* functions which have more restrictions, but less overhead. In general, the programmer should make a function local if it contains no useful parallelism and it accesses only local memory. Local functions are indicated by including the keyword `local` at the beginning of the function declaration. A local function will be invoked on the processor from which it was called, and thus calls to local functions may be inlined. Furthermore, local functions must access only memory local to the current processor, and they may only call local functions. If the programmer states that a function is local, then the compiler may assume that all direct and indirect memory references are local.

Consider a typical example in Figure 6. In this example, the function `incr_tree` is a remote function that traverses a binary tree, incrementing the key field of each node by k. The two recursive calls to `incr_tree` are invoked on the processors that own the roots of the left and right sub-trees using the built-in primitive `OWNER_OF` which returns the processor that owns the address. The function `incr_node` is declared as a local function, so it will be invoked on the current processor, and all memory accesses to `root` will be assumed to be local. Note that inside local functions all memory references are implicitly local, so we do not need to explicitly give a local pointer type to the parameter `node` in the function `incr_node`, although it would also be correct to give the explicit local declaration.

```
void incr_tree(nodeptr * local root, int k)
{ if (root != NULL)
    {^ incr_tree(root->left,k)@OWNER_OF(root->left);
       incr_tree(root->right,k)@OWNER_OF(root->right);
       incr_node(root,k);
    ^}
}

local void incr_node(nodeptr * node, int k)
{  node->key = node->key + k;
}
```

Figure 6: Example of local pointers and local functions

The main idea of the program in Figure 6 is that the remote invocations for the recursive traversal via `incr_tree` expose parallelism and send computation to the processor owning the data, thus it is worth the extra overhead. Whereas, it is preferable to make the computation for `incr_node` less expensive by indicating that it is a local function. If we had omitted the keyword `local` in the declaration of `incr_node` then the program is still correct, but the call to `incr_node` would be more expensive and would be assigned by the run-time load-balancer, the call could not be inlined, and some memory accesses would be assumed to be remote.

2.5 EARTH Library Functions and Primitives

The programmer may sometimes want to access blocks of data, and to copy blocks of data between processors. For these cases we provide the primitive `blkmov(source, dest, nbytes)` which is used primarily to copy arrays and records between processors. From the programmer's perspective, `blkmov` is the EARTH-C version of the usual C library function `memcpy`.

Other useful built-in primitives include `OWNER_OF(addr)` which returns the processor owning the `addr`, `NODE_ID` which returns the processor number currently executing, and `NUM_NODES` that returns the number of processors allocated to the program.

3 The EARTH-C Compiler

We have implemented a version of the McCAT compiler, called EARTH-McCAT, that translates EARTH-C programs into low-level Threaded-C programs. As illustrated in Figure 7, the compiler can be divided into three main phases. Phase I is composed of the standard McCAT transformations and analyses that have been slightly modified to handle EARTH-C extensions. Although these analyses are very important for the subsequent phases, they are not the main point of this paper. The important point to note is that after this first phase we have a structured intermediate representation called SIMPLE-EARTH-C and appropriate dependence information for scalars, indirect references via pointers, and array references. SIMPLE-EARTH-C is a slight extension of SIMPLE, the standard intermediate representation for McCAT.[4] The salient features here are: the program is now compositional (`gotos` have been eliminated); each statement has a simple form which has at most one indirection or array reference; expressions in the tests of conditional and loop constructs are simplified so that they refer only to scalar values; and arguments to functions are either constants or scalar variables.

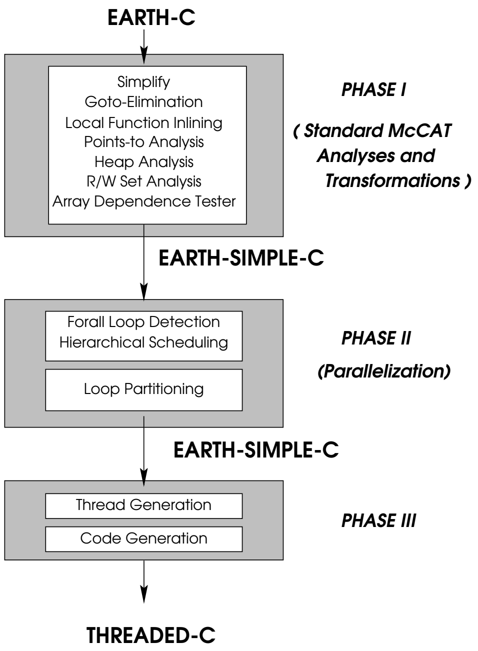


Figure 7: Overall structure of the compiler

The main focus of this paper is on phases II and III. Phase II takes EARTH-SIMPLE-C as input and performs parallelization and loop transformations and produces EARTH-SIMPLE-C as output. Phase III takes EARTH-SIMPLE-C as input and does the thread partitioning and code generation, producing Threaded-C as output. The main components of these phases are discussed in subsequent subsections.

[4]We will use the terms SIMPLE and SIMPLE-EARTH-C interchangeably.

3.1 PHASE II - Parallelization and Loop Transformations

3.1.1 Hierarchical Scheduling

Although the programmer may have explicitly specified coarse-grain parallelism using `forall` and parallel sequences, the compiler can augment this parallelism by automatically detecting when two or more statements can execute in parallel. The purpose of the hierarchical scheduler is to find those statements that can safely execute in parallel, and to enclose them in a parallel statement sequence. The parallelism that we detect may be fine-grain (when we detect that several assignment statements may execute in parallel), or coarse-grain (when we detect that loops or function calls may execute in parallel).

Figure 8 gives a small example using a piece of code extracted from the partitioning phase of a quicksort program. Figure 8(a) gives the original C source code, Figure 8(b) gives the SIMPLE version, and Figure 8(c) gives the SIMPLE version after scheduling. Note that when scheduling is applied to the outermost ordinary statement sequence it detects that the two `do` loops could be executed in parallel (coarse-grain parallelism). In the inner statement sequences, it only finds one opportunity for parallelism in the statements for swapping the values (fine-grain parallelism).

```
while (true) do
 { do  /* scan left */
     j--;
   while (a[j] > x);

   do   /* scan right */
     i++;
   while (a[i] < x);

   if (i < j) /* swap */
     { t = a[i];
       a[i] = a[j];
       a[j] = t;
     }
   else
     return(j);
 }
```

(a) input C code

```
while (true) do
 { do /* Read = {j,a,x}
         Write = {j,temp1} */
    { j = j - 1;
      temp1 = a[j];
    }
   while (temp1 > x);

   do /* Read = {i,a,x}
         Write = {i,temp2} */
    { i = i + 1;
      temp2 = a[i];
    }
   while (temp2 < x);

   if (i < j)
     { t = a[i];
       temp3 = a[j]
       a[i] = temp3;
       a[j] = t;
     }
   else
     return(j);
 }
```

(b) SIMPLE code

```
while (true) do
 { {^ do
       { j = j - 1;
        temp1 = a[j];
       }
      while (temp1 > x);

      do
       { i = i + 1;
         temp2 = a[i];
       }
      while (temp2 < x);
    ^}

   if (i < j)
     { {^ t = a[i];
          temp3 = a[j];
       ^}
       a[i] = temp3;
       a[j] = t;
     }
   else
     return(j);
 }
```

(c) Scheduled SIMPLE

Figure 8: Example of Scheduling

The main scheduling algorithm, called *Schedule*, is given in Figure 9. The basic idea is that we recursively traverse the SIMPLE AST, applying the percolation scheduler to each ordinary statement sequence as they are encountered. The percolation scheduler merely moves operations up in the sequence as early as possible, while maintaining data dependences. A statement S_i may be moved up as long as it does not interfere with statement S_{i-1}. Two statements do not interfere if their *interference set* is empty. As PHASE I of the compiler computes read/write sets for all statements (including compositional statements and function calls), the interference set is computed directly using the read/write sets. Given that $W(S)$ is the set of locations written by statement S, and $R(S)$ is the set of locations read by statement S, the interference set of two statements S_i and S_j is defined as follows: $inter(S_i, S_j) = (W(S_i) \cap (R(S_j) \cup W(S_j))] \cup [W(S_j) \cap (R(S_i) \cup W(S_i)))$. As the read/write set is supported by our points-to analysis, this correctly handles all memory references, even those that are indirect references through a pointer.

```
proc schedule(S)
/* Given a SIMPLE representation of
   stmt S, process S top-down, and apply
   the percolation scheduler to each
   sequential sequence encountered. */
{ case S of
  { /* process each Si, then percolate S */
    ordinary_seq(S1,S2,...,Sn):
      { foreach Si in S do schedule(Si);
        percolate(S);
      }
    /* process each Si, no percolation */
    sequential_seq(S1,S2,...,Sn):
    parallel_seq(S1,S2,...,Sn):
      foreach Si in S do schedule(Si);

    /* process the body of loop */
    while_stmt(Exp,Sbody):
    do_stmt(Sbody,Exp):
    for_stmt(init,test,incr,Sbody):
      schedule(Sbody);

    /* process each side of conditional */
    if_stmt(Exp,Sthen,Selse):
      { schedule(Sthen);
        schedule(Selse);
      }
    /* process each case in switch */
    switch_stmt(Exp,Scase):
      foreach Si in Scase do schedule(Si);
  }
}
```

```
proc percolate(stmt_seq)
/* Given a statement sequence (stmt_seq),
   percolate each statement up as far
   as possible. */
{ /* for each statment in the sequence */
  for i := 2 to n do
    {  /* find index of first interfering stmt */
       inter_index := find_inter(i,stmt_seq);

       /* if not the stmt just above stmt i */
       if (inter_index < i-1)
          /* move statement at index i into a
             parallel stmt sequence with stmts
             already at index inter_index+1 */
         move_stmt(i,inter_index+1,stmt_seq);
    }
}

fun find_inter(i,stmt_seq)
/* Given a statement sequence (stmt_seq) and
   an index into that sequence (i), scan each
   stmt above i until a statement at index k
   is found that interferes with the statement
   at i,  or k is 0. */
{ k := i;
  repeat
    k := k - 1;
  until (k = 0) or (inteferes(i,k,stmt_seq));
  return(k);
}
```

Figure 9: Hierarchical Scheduling

3.1.2 Forall Loop Detection

As described in section 10, the EARTH-C `forall` loop specifies that once all iteration values of the header have been computed, all iterations of the body may proceed in parallel. In some cases the programmer will explicitly use the `forall` construct. However, we also use our compiler's dependence analysis to automatically locate `for` loops that can be automatically translated to `forall` loops. A `for` loop may be translated to a `forall` look when there are no inter-iteration dependences in the body of the loop. We currently use the read/write sets to check that there are no inter-iteration dependences due to scalars and pointer dereferences, and we use the McCAT array dependence tester to verify that that all dependences between array references have a dependence distance of 0.

3.1.3 Forall Loop Partitioning

Once all possible `forall` loops have been detected, there are two main ways that we can exploit the parallelism available in these loops: *function partitioning* and *thread partitioning*. Function partitioning is used to partition the iteration space across **all processors**, whereas thread partitioning allocates different iterations to different threads on the **same processor**. Consider, for example, the `forall` loop in Figure 10(a). This loop iterates over array `a` which is an array of pointers to linked lists. In each iteration of the `forall` loop, a `while` loop is used to traverse the linked list whose head is stored in `a[i]`.

Figure 10(b) shows the result of function partitioning. The body of the loop is packaged into a function `forloop0`, and the body of the loop is replaced by a call to the function with the appropriate parameters. As function calls in EARTH-C are spread among all processors, this effectively spreads the iteration space across processors.

Function partitioning is not the only possible use of parallelism. As mentioned earlier, parallelism can also be used on the same processor. By creating multiple threads on a processor, the processor can switch to a ready thread when a long-latency operation is encountered. In the original `forall` loop, the access `l->next` is remote, and is thus a long-latency operation. We would like to ensure that there is another thread, on the same processor, ready to execute. As illustrated in Figure 10(c), this is accomplished by thread partitioning the original loop. In this case the iteration space is divided into k segments (in this example $k = 2$) by creating a parallel sequence that contains k `forall` loops, one `forall` loop for each segment of the iteration space. Each `forall` loop now corresponds to a thread of control, and all threads are on the same processor. Thus, when one of these smaller `forall` loops encounters a remote remote access (`l->next` or `l2->next`), there is likely to be another thread available. The compiler chooses k by examining the number of remote accesses in the body of the loop, and the estimated running time of one iteration of the loop.

3.2 PHASE III - Thread Creation and Code Generation

3.2.1 EARTH Execution Model

One of the fundamental ideas of the underlying EARTH multithreaded architecture model is that remote memory accesses are handled in a split-phase manner. That is, a remote memory request is made by one thread, and when the request is satisfied a synchronization is made with another thread that uses the value. Remote function calls are also handled in a split-phase manner, the function call is issued, and when the invocation terminates it signals its completion. Thus, when a remote function (i.e. not a local function) is invoked, its invocation

```
forall(i=0; i<N; i++)
  { l = a[i];
     while (l != NULL)
       { f(l,k);
         l = l->next;
       }
  }
```

(a) original loop

```
void
forall0(LIST *a[],int i,int k)
{ LIST *l;
  l = a[i];
  while (l != NULL)
    { f(l,k);
      l = l->next;
    }
}

...

forall(i=0; i<N; i++)
  forall0(a,i,k);
```

(b) function partitioning

```
temp1 = N / 2;
{^ forall(i=0; i<temp1; i++)
     { l = a[i];
       while (l != NULL)
         { f(l,k);
           l = l->next;
         }
     }
   forall(i2=temp1; i2<N; i2++)
     { l2 = a[i2];
       while (l2 != NULL)
         { f(l2,k);
           l2 = l2->next;
         }
     }
^}
```

(c) thread partitioning

Figure 10: Partitioning `forall` Loops

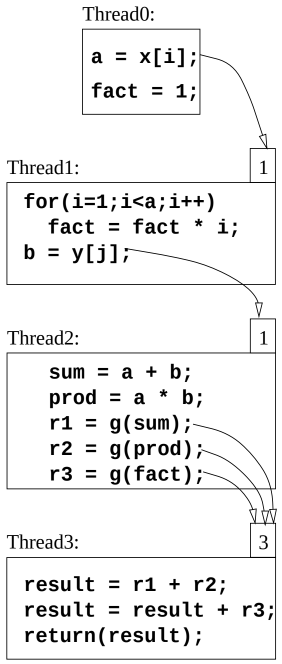
is issued (most likely to another processor) as a new thread, and control moves immediately to the next statement in the current thread. When the result of the function is ready, a synchronization is made with a thread that that uses the value. Consider the example C program in Figure 11(a). In this program the remote operations are the reads from array `x` and the calls to function `g`. All other operations are local since they refer to formal parameters or local variables.

Consider how the threads must be generated in order to satisfy the constraints due to split-phase memory operations and remote function calls. The request (remote read/write, or function invocation) must appear in a different thread from the use or redefinition of the value, and the correct synchronization must be made. Figure 11(b) shows a naive thread generation. In this case the body of the function was processed from top to bottom, and a thread boundary was inserted each time a statement used or redefined the result of a split-phase operation that appears in the current thread. For example, the end of Thread 0 was placed just before the `for` loop because the `for` loop uses `a` which is the result of a split-phase read of `x[i]` within Thread 0. Similarly, the end of Thread 1 was placed just before the assignment to `sum` because it uses `b` which is the result of a split phase read of `x[j]` from within Thread 1. Finally, the end of Thread 2 was placed just before the assignment to `result` because it uses `r1` and `r2` which are results of function calls to `g` which occurred within Thread 2.

Following the EARTH model, synchronization between threads is implemented via so-called *synchronization slots* attached to each thread. The number in each slot determines how many signals must be received before the thread can execute, while the arcs associate split-phase operations with their appropriate slot. For example, the completion of the remote read of `x[i]` in Thread 0 signals Thread 1. Thread 1 is ready as soon as it receives one signal. In this naive scheme, the slot for $Thread_i$ is associated with all the split-phase operations that occur in $Thread_{i-1}$.

```
int f(int *x, int i, int j)
{ int a,b,sum,prod,fact;
  int r1,r2,r3,result;

  /* compute fact */
  a = x[i];
  fact = 1;
  for(i=1; i<a; i++)
    fact = fact * i;

  /* compute sum and prod */
  b = x[j];
  sum  = a + b;
  prod = a * b;

  /* apply g */
  r1 = g(sum);
  r2 = g(prod);
  r3 = g(fact);

  /* sum results */
  result = r1 + r2;
  result = result + r3;
  return(result);
}
```

(a) EARTH C



(b) Threads and Slots

Figure 11: Remote memory accesses and naive thread creation

3.2.2 Thread Creation via List Scheduling

We can improve upon the thread creation scheme given in the previous section using a list scheduling strategy. For each statement sequence we build a Data Dependence Graph (DDG) with one node for each statement, and arcs showing dependences between the statements (transitive edges are removed). Currently the dependences are calculated using the results of Read/Write and points-to analysis[8]. This gives us quite precise DDGs, even in the presence of pointers. However, we will soon improve upon this even further by integrating a symbolic array dependence tester, and more refined heap dependence testing. Each node in the DDG is given two labels: *statement type* and *earliest thread number*. Figure 12(a) gives the labeled DDG for the example program from Figure 11(a). The rules for computing the labels are described in the following subsections.

Statement Type: Each node is labeled with one of the types summarized in Table 1. The basic idea is that a statement is given a LOCAL type if it requires no split-phase operations, whereas any statement containing a split-phase operation (remote read, remote write, remote function call) is given a REMOTE type. We can determine the type of each statement by looking at the type and scope of each variable referenced. Direct references to ordinary globals, and indirect references via ordinary pointers are remote. Direct references to replicated globals, locals, parameters, and indirect references via local pointers are local. If a statement contains

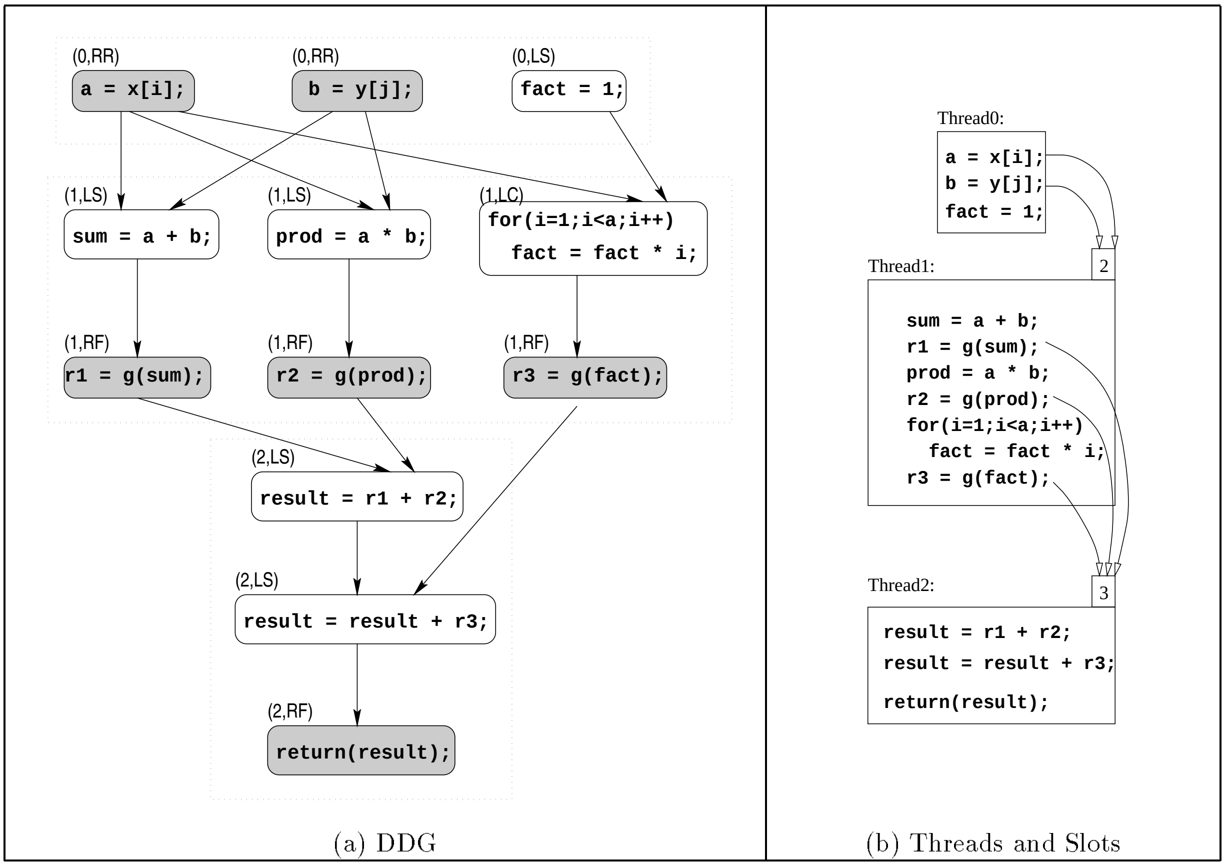


Figure 12: DDG and List Scheduling

any variable reference that is remote, the statement is labeled as REMOTE. The DDG along with the statement types for the example program of Figure 11(a) is given in Figure 12(a). Note that each statement of the main body of the function is represented by one node (even compositional statements, like the `for loop`. The `for` loop in this program contains only local operations, so it is given the type LOCAL_COMPOUND. If it had contained a split-phase operation, it would be given the type REMOTE_COMPOUND.

Earliest Thread Number: In addition to a statement type, each node in the DDG is labeled with a number corresponding to the earliest thread in which the statement could be placed. The fundamental observation is that if a node P has a REMOTE type, and $thread\_num(P) = i$, then the earliest thread number for all successors of P is thread $i + 1$. Similarly, if node Q has many REMOTE ancestors, say $P_1 \ldots P_k$, then the earliest thread number for Q is $max(thread\_num(P_1)+1, \ldots, thread\_num(P_k)+1)$. Based on these observations, we can assign earliest thread numbers to each node in the DDG using the algorithm in Figure 13(a) and an example of applying the algorithm is given in Figure 13(b) (each node in the DDG is labeled with its earliest thread number, nodes with REMOTE type are shaded). Since this algorithm places each statement in the earliest possible thread, then total number of threads needed for

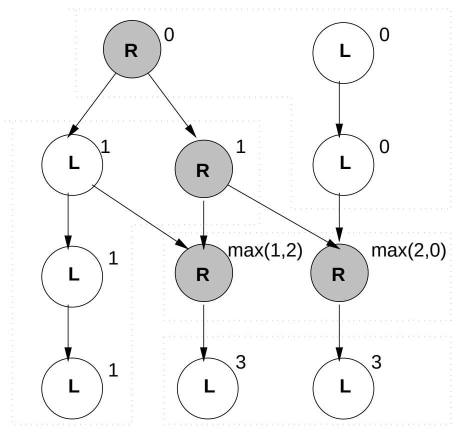
Statement Type	Abbrev.	Description
LOCAL_SIMPLE	LS	a simple assignment statement accessing only local memory
LOCAL_FN_CALL	LF	a call to a local function
LOCAL_COMPOUND	LC	a conditional or loop statement that only accesses local memory and calls only local functions
REMOTE_READ	RR	an assignment statement that reads a remote value
REMOTE_WRITE	RW	an assignment statement that writes a remote value
REMOTE_FN_CALL	RF	a call to a remote function
REMOTE_COMPOUND	RC	a conditional or loop statement that accesses remote memory or calls a remote function

Table 1: Statement Types for Scheduling

the statement sequence is minimized and thus the average thread length is maximized.

```
proc assign_all(D)
/* Given a DDG, assign earliest thread
   numbers to each node.  Assume each
   node initialized with thread num of 0 */
{ /* for each root in DDG */
  foreach root R in D do
    assign_num(R,0);
}

proc assign_num(N,i)
/* Given a node N in the DDG, and the
   predecessors thread num (i). Assign
   numbers to successors */
{ N.threadnum := max(N.threadnum,i);
  foreach successor S of N do
    if N.type = R then  /* N is a Remote */
      assign_num(S,N.threadnum + 1);
    else
      assign_num(S,N.threadnum);
}
```

(a) Algorithm



(b) Example

Figure 13: Earliest Thread Number Assignment

Our list scheduling strategy is based on the the earliest thread number and statement type. Statements with the smallest thread number are chosen first. This implies that all statements with earliest thread number i will be scheduled before statements with earliest thread number $i + 1$. If there are many operations ready with the same thread number the statement with the lowest statement type is chosen. The ordering on statement types is: REMOTE_READ < REMOTE_WRITE < REMOTE_FN_CALL < LOCAL_SIMPLE < REMOTE_COMPOUND

< LOCAL_COMPOUND < LOCAL_FN_CALL. The basic idea behind this ordering is that we want to issue the split phase operations as early as possible, and then do all the computation that is local. By issuing REMOTE_READ and REMOTE_WRITE statements early, local computation is overlapped with the communication. By issuing REMOTE_FN_CALL statements early, function-level parallelism is exposed as early as possible. We place small LOCAL_SIMPLE statements quite early in the ordering since they take very little time to execute, and scheduling them early often "uncovers" important REMOTE operations. Local computations that likely take a lot of time (LOCAL_FN_CALL and LOCAL_COMPOUND) are scheduled last. The actual scheduling algorithm is given in Figure 14. Note that all of the remote operations placed in thread number i will be associated with the synchronization slot in thread $i+1$. Figure 12(b) shows the result of our list scheduling algorithm when applied to the DDG in Figure 12(a). Note that compared to the naive thread generation approach of Figure 11(b), our list scheduling approach gives only 3 threads rather than 4. Furthermore, many of the remote operations are issued earlier than in the naive schedule. For example, the statement `b = y[i];` has moved from Thread 1 in the naive schedule to Thread 0 in the good schedule. Furthermore, the invocations `r1 = g(sum);` and `r2 = g(prod);` are scheduled earlier in the good schedule.

```
proc list_schedule(D)
/* Given a DDG, generate threads using the earliest thread number
   and statement type. */
{  put all roots of D on the ready_list;
   i = 0; /* start with thread 0 */
   threadsync = 0;
   start thread 0;
   while (not empty(ready_list)) do
     { N = node from ready_list with min value;
       /* check to see if a new thread should be started */
       if (N.thread_num != i) then
         {  mark the end of thread i;
            i++; /* increment thread counter */
            start thread i;
            set the sync count for thread i to threadsync;
            threadsync = 0;
         }
       add N to thread i;
       add successors of N that are now ready to ready_list;
       /* if a remote op, then increment sync count */
       if N.type = R then threadsync++;
     }
   mark the end of thread i;
}
```

Figure 14: List Scheduling Algorithm

This basic list scheduling approach is applied hierarchically in order to generate threads for each function body. The scheduler is applied first to the outermost statement list, and then recursively applied to the bodies of any statement with type REMOTE_COMPOUND. A small variation on the strategy is used when compiling the bodies of `forall` loops. In this case the synchronization associated with the last thread in the body is moved outside of the loop. This allows multiple iterations to run in parallel and enforces a barrier synchronization at the end of the loop.

3.2.3 Code Generation

Once the threads have been partitioned, and the synchronization counts computed, the threaded-C code is generated. Each *remote* function is compiled as a `THREADED` function, using the keyword `THREADED` at the beginning and `END_FUNCTION` at the end. The threaded code for our example program is given in Figure 15(b). Each threaded function has two extra parameters: `ret_parm` which points to the destination for the return value, and `*rsync_parm` that points to the synchronization slot associated with the caller of the function. In addition to all the regular local variable declarations, we declare an array of synchronization slots, one per thread (Thread 0 starts at the beginning of the body, and it has no synchronization slot). These slots are initialized using the `INIT_SYNC` threaded-C instruction. For example, in our generated code `INIT_SYNC(0, 2, 2, 1)` is used to indicate that Thread 1 is associated with SLOT 0, and it should have an initial synchronization count of 2, and a reset synchronization count of 2. Each thread starts with the appropriate thread label, and terminated with the `END_THREAD();` statement. Each remote read is translated into an Threaded-C `GET_SYNC` and each remote write is translated into a `DATA_SYNC`. Calls to `local` functions are not modified, but calls to `remote` functions are compiled into `TOKEN` or `INVOKE` calls. The `TOKEN` mechanism is used when the run-time load balancer should be used, and the `INVOKE` mechanism is used when an explicit processor number is given.

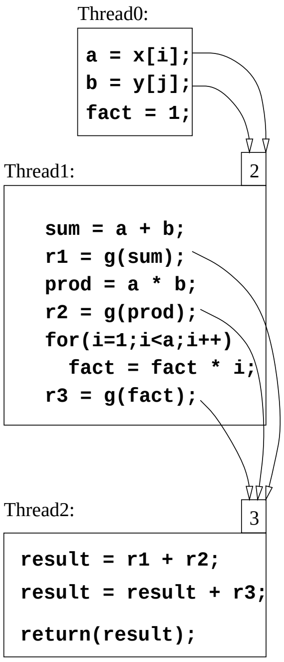
4 Experimental Results

In order to evaluate our approach we have experimented with six benchmarks as described in Table 2. The first four benchmarks are representative of our target class of benchmarks as they use recursion and/or dynamic data structures. The last two benchmarks, `mmult` is a small example, more typical of scientific codes.

Benchmark	Description	Problem Size
tsp	Find sub-optimal tour for traveling salesperson problem	32K cites
power	Optimization Problem based on a variable k-nary tree	10,000 leaves
nqueen	Search for all legal queen configurations on a chess board	12x12
quicksort	Parallel version of quicksort	512K integers
mmult	Matrix Multiplication	512x512 floats
tomcatv	SPEC benchmark	258x258

Table 2: Benchmark Programs and EARTH-C Features

For the first four benchmarks, we started with the original sequential programs and introduced EARTH constructs for parallelism and locality. In Table 3, we summarize the EARTH-C features used in this group of benchmarks. We give two versions for each benchmark, a *naive* version and an *advanced* version. In the *naive* versions we made relatively few changes to the



(a) Threads

```
THREADED f(int *ret_parm, SLOT *rsync_parm,
                               int *x, int i, int j)
{ SLOTS SYNC_SLOTS[2];
  int a,b,sum,prod,fact,r1,r2,r3,result;

/* THREAD_0:; */
  INIT_SYNC(0, 2, 2, 1);
  INIT_SYNC(1, 3, 3, 2);

  GET_SYNC_L(&x[i], &a, 0);
  GET_SYNC_L(&x[j], &b, 0);
  fact = 1;
  END_THREAD();

THREAD_1:;
  sum = a + b;
  TOKEN(g, &r1, (SLOT *) SLOT_ADR(1), sum);
  prod = a * b;
  TOKEN(g, &r2, (SLOT *) SLOT_ADR(1), prod);
  for (i = 1;i < a; i = i + 1)
    { fact = fact * i; }
  TOKEN(g, &r3, (SLOT *) SLOT_ADR(1), fact);
  END_THREAD();

THREAD_2:;
  result = r1 + r2;
  result = result + r3;
  DATA_RSYNC_L(result, ret_parm, rsync_parm);
  goto end_of_function;

end_of_function:;
  END_FUNCTION();
}
```

(b) Threaded-C code

Figure 15: Generated Threaded-C Code

original sequential programs, and we used only the most relevant EARTH-C features that concentrated on exposing parallelism. For the *advanced* versions we changed the program more dramatically, using more EARTH-C constructs, most often by adding appropriate uses of local pointer declarations and/or local functions.

For each benchmark we executed the original sequential version, the naive and advanced EARTH-C versions, and a hand-coded Threaded-C benchmark. For the benchmarks `queens`, `mmult` and `tomcatv` we used Threaded-C codes developed and hand-optimized by other programmers. For `power`, `tsp` and `quicksort` had to develop and optimize our own Threaded-C programs. Table 4 gives the results for program size and program performance. The column labeled *sequential C* gives the time for running the original sequential version, whereas the columns labeled *Naive EARTH-C*, *Advanced EARTH-C*, and *Threaded-C* give the speedups for 1, 8 and 16 processors (speedup relative to the sequential program). The columns labeled *Naive/Threaded* and *Advanced/Threaded* give the ratio of the time for the EARTH-C versions with respect to the hand-coded Threaded-C versions.

Benchmark	EARTH-C Features					
	`{^ ... ^}` for Function-level Parallelism	`forall` for Loop-Level Parallelism	Block Move for Collective Communication	Local Functions for Throttling Parallelism	Local Functions for Data Locality	Local Pointers & OWNER_OF for Data Locality
power	•	•			∘	∘
queens	⊙		•	•		∘
tsp	•	•	•	•	∘	∘
quicksort	•		•	•	•	∘

Table 3: Benchmark Features: Compiler (⊙), Naive(•), Advanced(∘)

Benchmark		Sequential C	Naive EARTH-C	Advanced EARTH-C	Threaded C	Naive/ Threaded-C	Advanced/ Threaded-C
power	1 proc	62.4 sec	0.92	0.96	0.99	0.93	0.97
	8 procs		6.61	7.13	7.35	0.90	0.97
	16 procs		12.76	13.26	14.17	0.90	0.94
queens	1 proc	37.7 sec	0.96	0.96	1.00	0.96	0.96
	8 procs		7.65	7.65	8.01	0.96	0.96
	16 procs		15.10	15.21	15.89	0.95	0.96
tsp	1 proc	23.8 sec	0.72	0.99	0.99	0.73	1.00
	8 procs		3.86	6.37	6.67	0.58	0.96
	16 procs		4.70	9.66	11.31	0.42	0.85
quicksort	1 proc	2.6 sec	0.77	0.99	1.00	0.77	0.99
	8 procs		1.73	3.18	3.21	0.54	0.99
	16 procs		1.81	3.30	3.40	0.53	0.97
mmult	1 proc	17.1 sec	0.99	0.99	0.99	1.00	1.00
	8 procs		7.28	7.28	7.28	1.00	1.00
	16 procs		12.99	13.14	13.14	0.99	1.00
tomcatv	1 proc	64.9 sec	1.17	1.19	1.33	0.88	0.89
	8 procs		9.01	9.32	9.59	0.88	0.91
	16 procs		17.69	18.27	17.90	0.99	1.02

Table 4: Experimental Measurements

Power System Optimization: The `power` benchmark solves a power system optimization by computing the best price for the benefit of the community[14]. The problem is based on a four-level tree structure, with different branching widths at each level. The program executes a series of phases until convergence is reached. In each phase, a top-down pass broadcasts the price information to the leaves while bottom-up pass sends the consumers' demand up to the root. In the naive EARTH-C version, we specified the parallelism at each level in the tree using the `{^ ... ^}` construct for recursive calls, and `forall` construct for iteration over arrays of pointers to children. This leads to 10,000 parallel leaf computations which are balanced by the EARTH run-time system. As the run-time system does the allocation for load-balancing, data locality is not exploited. In the advanced version we exploit data locality, and only expand the parallelism on the first two levels giving 200 sub-trees. The computation on the sub-trees is allocated to the processor owning the complete sub-tree. Thus, computations on the lower two levels are specified as `local` functions.

The results in Table 4 is quite encouraging. Even the naive version gets 90% of the performance of the hand-coded Threaded-C version. Thus, the programmer can get very good speedup with only minimal programming effort. Furthermore, it appears that by exposing a lot of parallelism, the EARTH run-time system did mask most of the communication, and did a good job of load balancing. The result for the advanced version is even better, achieving 94% to 97% of the speedup of the hand-coded Threaded-C version. Thus, it seems that mapping the computation to the owner of the data works well in this case (as has been previously reported in [18]). It should be noted that it was very simple to change the EARTH-C program to try different variations of this benchmark.

Queens: The `queens` benchmark is a classic AI search benchmark which requires N queens to be put on a $N \times N$ chess board so that no queen may be taken by another. We use a recursive descent search to find all legal queen positions. At each search step, a small array recording the partial search space is copied from a parent node to a child. The parallelism in this benchmark is recognized by the compiler automatically (of course, it could also be specified using `{^ ... ^}`). For the naive version, we use `blkmov` to copy the partial search space, and we invoke local versions of the functions to throttle parallelism at a specified search depth. In the advanced version, we optimize the solution by recognizing when a parent and child are on the same processor. In this case we avoid the remote `blkmov` and use a local `memcpy` instead. The results in Table 4 show that both EARTH-C versions perform very well, with almost perfect speedup and at least 95% of the performance of the Threaded-C version. It appears that the optimization done in the advanced version is not very important.

Heuristic Traveling Salesperson: The `tsp` benchmark is a typical NP-complete problem. Our benchmark uses a heuristic divide-and-conquer approach based on close-point algorithm[12, 5]. This heuristic algorithm first searches a suboptimal tour for each subtree(region) and then merges subtours into bigger ones. The heuristic for merging attempts to avoid creating crossed edges. The tour found is built as a circular linked list sitting on the top of the root nodes of subtrees. The linked lists are distributed in segments and there are only few links across processors. This provides a great opportunity to use data locality. The advanced version makes full use of this partial data locality using the `OWNER_OF` primitive and `local` pointers. Both the naive and advanced versions use `local` functions to throttle the recursive parallelism. The data in Table 4 indicates that the locality optimization in the advanced version is important, and it becomes more important as the number of processors increases. With 16 processors the naive version gets only 42% of the performance of the Threaded-C version, whereas the advanced version achieves 85%. The Threaded-C version appears to perform better as it does even more locality optimization and it takes advantage of a cheaper procedure call mechanism when calling a remote function on the same processor.

Quicksort: This benchmark is a parallel implementation of the standard quicksort algorithm. The naive version simply invokes the two recursive calls in parallel, with no explicit processor assignment (the EARTH run-time system does the assignment). The advanced version keeps the bigger partition on the local processor to reduce interprocessor communication. Because the size of subarrays in each recursive sorting phase is unknown in advance, dynamically-allocated arrays are employed. In the advanced version, local pointers are used to point to the dynamically-allocated arrays and *memcpy* is used if the source and destination are found locating in the same processor. Examining the data in Table 4 indicates that all versions of `quicksort` have relatively poor speedup (as was expected). However, the advanced EARTH version performs almost as well as the Threaded-C version, and is considerably better than the naive version. Both the `tsp` and `quicksort` benchmarks demonstrate that the locality features in EARTH-C are very important.

Matrix Multiplication and Tomcatv: Unlike the previous benchmarks, where we started with the sequential algorithm, and made relatively few changes to the algorithm for the EARTH-C versions, the `mmult` and `tomcatv` benchmarks needed explicit blocking and partitioning. Our EARTH-C versions were inspired from the algorithms found in the existing Threaded-C versions.

For `mmult`, the naive EARTH-C version uses `blkmov` and `replicated` globals for collective communication and locality. The advanced EARTH-C version overlaps the communication of one iteration with the communication of another. The speedup of all versions is very good, and there seems to be very little difference between the EARTH-C versions and the Threaded-C version.

The `tomcatv` program is a data-parallel floating-point SPEC89 benchmark. From an initial pair of 258*258 meshes, each iteration updates the meshes, first using independent nearest-neighbour calculations, then by performing calculations with horizontal loop-carried dependences. Iterations continue until the maximum change drops below a threshold. In the Threaded-C version, a separate function invocation is created for each row and this invocation lives for the lifetime of all iterations of the algorithm. Invocations for separate rows synchronize with each other and the top-level function using a pure dataflow paradigm. However, the EARTH-C language doesn't provide explicit control of synchronization as in Threaded-C. Thus, the EARTH-C program each iteration is expressed as a simple fork-join parallelism that is easy to express as several `forall` constructs in sequence. As the EARTH-C approach requires new function invocations for each iteration of the main algorithm, the EARTH-C versions may suffer greater overhead. As in the Threaded-C version, the EARTH-C versions use a row-based data distribution and replicated global data to reduce communication cost. The advanced EARTH-C version differs from the naive version in that it overlaps communication with computation by issuing remote communication as early as possible. The performance of all versions is very good. It should be noted the super-linear speedup for the parallel versions is achieved due to cache effects, and better register allocation. Also note that as the number of processors increases, the advanced EARTH-C version performs better with respect to the Threaded-C version, and even outperforms the Threaded-C version for 16 processors. This is likely due to the fact that EARTH-C version takes advantage of substituting local `memcpy` operations for more expensive `blkmov` operations whenever possible, and the overlap of communications and computation. Thus, it appears that the complicated synchronization strategy in the Threaded-C version is not really necessary for large numbers of processors.

In general, for both `mmult` and `tomcatv`, the performance of the EARTH-C versions is very good. Furthermore, we found that the EARTH-C versions were simpler to express, and easier to read. Thus, there appears to be no big advantage in programming in Threaded-C for this type of benchmark.

5 Related Work

In this section, we briefly outline some related work. Due to space limitations, we focus on language extensions based on C with multithreading as target program execution models.

Split-C is a parallel extension to C and provides shared-memory, message-passing, and data parallel abstractions to programmers[6]. Split-C provides a global address space supporting both local and global pointers. Split-C follows an SPMD model of program execution. As the authors stated that Split C "is in stark contrast to languages that rely on extensive program transformation at compile time to obtain performance on parallel machines".

Cid is designed to be a parallel, shared-memory superset of C [16]. Cid provides global and local pointers similar to Split-C. Cid also supports fork-join parallel control structures. Load balancing can be directly controlled by programmers through explicit specification of sites of function invocations. Latency of remote forks and remote accesses is made visible by making them asynchronous operations [16].

Cilk is a C-based language and runtime system for multithreaded parallel programming [2], especially for explicit continuation-passing style programming[9]. It is more like our target language Threaded-C, with a rich set of thread scheduling primitives to assist in load balancing. Like EARTH-C, it has local function and local pointer concepts and programmers can use call sites to control load balance. Since it is based on explicit continuation-passing programming model, continuations must be explicitly specified by the programmer.

EM-C is a parallel extension of the C languages designed for efficient parallel programming in the EM-4 multiprocessor[19]. It provides simple data distribution to support global data memory and the I-structures for global data synchronization. It supports parallel sequences and `forall` parallel constructs similar to those in EARTH-C, in addition to several other parallel block extensions. Remote access and synchronization must be explicitly specified by the programmer.

Olden is a compiler and runtime system for C programs on distributed memory machines[18, 3]. In Olden, the programmer uses a *future* notation to expose parallelism to the compiler and the runtime system uses lazy task creation and thread migration strategies to exploit parallelism.

One distinct feature of EARTH-C is to provide the programmer a more traditional high-level language that does not explicitly support threads, and to use selected language extensions as well as compilation techniques to automatically translate high-level programs into lower-level threaded programs. Another unique experience with the EARTH-C development is that the language extensions have been evolving through an integrated project in which the EARTH architecture and runtime system, and the EARTH-C compiler are developed together, along with the parallel development of a substantial suite of benchmarks in both EARTH-C and Threaded-C. This experience has helped us to understand what is essential for users to express parallelism in their programs, and what the compiler should do to generate efficient code and exploit the power of the EARTH architecture and its program execution model.

6 Conclusions and Further Work

We have presented a methodology for programming multithreaded architectures. We have focused on providing a parallel dialect of C, called EARTH-C, that exposes parallelism and locality to the programmer, but does **not** expose threads or communication. By using only very simple extensions to C, we provide a straight-forward programming model for the programmer, and a language can be effectively analyzed by the compiler. Thus, the programmer can focus on high-level issues like specifying coarse-grain parallelism and improving locality, while the compiler does the work of detecting fine-grain parallelism and creating the lower-level threads and synchronization. We presented a thread generation strategy based on list scheduling that minimizes the number of threads, and schedules remote operations as early as possible.

We tested our approach with six benchmark programs, and we found that the EARTH-C programs were simple to write, and that it was easy to create different versions of programs that explored the effect of throttling parallelism, exploiting locality, collective communication, and so on. Thus, it seems to be a good language for rapid prototyping.

Even with naive EARTH-C versions of the benchmarks, where we did not focus on maximizing data locality, we achieved very good speedup in some cases, and reasonable speedup in others. This was, in part, due to the fact that the underlying multithreaded model can tolerate some communication if there exists adequate parallelism. Thus, the programmer need not always worry about data locality.

Our advanced EARTH-C versions of the benchmarks did use more locality, and in several cases this does result in significant performance improvement. Thus, the features for expressing locality in EARTH-C were useful, and are definitely necessary. Finally, we found that the advanced EARTH-C versions almost match the performance of the hand-coded Threaded-C versions. Thus, for this class of benchmarks, there is no significant performance penalty for programming in the EARTH-C language that hides the underlying multithreaded architecture.

Based on our experiences we plan to improve the compiler to do more locality analysis and collective communication automatically. We would also like to study more variations on our list-scheduling heuristics. These additions should further improve the performance of naive EARTH-C programs. We also plan to study other language features that can increase the expressibility of EARTH-C, while keeping the language analyzable for the compiler.

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 23(2), May 1995.

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In

Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), pages 207–216, Santa Barbara, California, July 19–21, 1995. *SIGPLAN Notices,* 30(8), August 1995.

[3] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 29–38, Santa Barbara, California, July 19–21, 1995. *SIGPLAN Notices,* 30(8), August 1995.

[4] Derek Chiou, Boon S. Ang, Robert Greiner, Arvind, James C. Hoe, Michael J. Beckerle, James E. Hicks, and Andy Boughton. StarT-NG: Delivering seamless parallel computing. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *Proceedings of the First International EURO-PAR Conference*, number 966 in Lecture Notes in Computer Science, pages 101–116, Stockholm, Sweden, August 29–31, 1995. Springer-Verlag.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press; McGraw-Hill Book Company, Cambridge, Massachusetts; New York, New York, 1990.

[6] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Portland, Oregon, November 15–19, 1993. ACM SIGARCH and IEEE Computer Society.

[7] Jack B. Dennis and Guang R. Gao. Multithreaded architectures: Principles, projects, and issues. ACAPS Technical Memo 29, School of Computer Science, McGill University, Montréal, Québec, February 1994. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[8] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 20–24, 1994. *SIGPLAN Notices,* 29(6), June 1994.

[9] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. Mimd-style parallel programming based on continuation-passing threads. CSG Memo 355, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, Massachusetts, March 1994.

[10] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 1996. To appear.

[11] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Laurie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank S. Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. A design study of the EARTH multiprocessor. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 59–68, Limassol, Cyprus, June 27–29, 1995. ACM Press.

[12] Richard M. Karp. Probabilistic analysis of partitioning algorithms for the travelling-salesman problem in the plan. *Mathematics of Operations Research*, 2(3):209–224, August 1977.

[13] Yuetsu Kodama, Hirohumi Sakane, Mitsuhisa Sato, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi. The EM-X parallel computer: Architecture and basic performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 14–23, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 23(2), May 1995.

[14] S. Lumetta, L. Murphy, X. Li, D. Culler, and I. Khalil. Decentralized optimal power pricing: The development of a parallel program. In *Proceedings of Supercomputing '93*, pages 240–249, Portland, Oregon, November 15–19, 1993. ACM SIGARCH and IEEE Computer Society.

[15] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xinmin Tian. Polling watchdog: Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, Philadelphia, Pennsylvania, May 22–24, 1996. ACM SIGARCH and IEEE Computer Society. To appear.

[16] Rishiyur S. Nikhil. Cid: A parallel, "shared-memory" C for distributed-memory machines. Technical report, Cambridge Research Laboratory, Digital Equipment Corporation, July 1994.

[17] Michael D. Noakes, Deborah A. Wallah, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 224–235, San Diego, California, May 17–19, 1993. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 21(2), May 1993.

[18] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[19] Mitsuhisa Sato, Yuetsu Kodama, Shuichi Sakai, and Yoshinori Yamaguchi. EM-C: Programming with explicit parallelism and locality for the EM-4 multiprocessor. In Michel Cosnard, Guang R. Gao, and Gabriel M. Silberman, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '94*, pages 3–14, Montréal, Québec, August 24–26, 1994. North-Holland Publishing Company.

[20] Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally. Evaluation of mechanisms for fine-grained parallel programs in the J-Machine and the CM-5. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 302–313, San Diego, California, May 17–19, 1993. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 21(2), May 1993.