International Journal of Parallel Programming, Vol. 25, No. 4, 1997

Compiling C for the EARTH Multithreaded Architecture[1]

Laurie J. Hendren,[2] Xinan Tang,[2] Yingchun Zhu,[2] Shereen Ghobrial,[2] Guang R. Gao,[2] Xun Xue,[2] Haiying Cai,[2] and Pierre Ouellet[2]

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 present EARTH-C our extended C language which contains simple constructs for specifying control parallelism, data locality, shared variables and atomic operations. 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 threated-C programs.

KEY WORDS: Parallel languages; multithreaded architecture; compiling for parallel architectures.

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.[1-5] As

[1] This work supported, in part, by NSERC and FCAR.
[2] School of Computer Science, McGill University, Montreal, Quebec H3A 2A7, Canada.

0885-7458/97/0800-0305$12.50/0 © 1997 Plenum Publishing Corporation

long as there is enough parallelism in an application, a multithreaded architecture can hide these latencies and effectively utilize available communication bandwidth. 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 in EARTH-C into lower-level Threaded-C programs that run on the EARTH (Efficient Architecture for Running Threads) multiprocessor.(6–8) 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 behavior, 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. The main programming paradigm in EARTH-C is fork/join type parallelism where the parallel computations must not interfere (one computation cannot write a location read or written by the other computation). We relax this restriction for the case of shared variables which must be accessed via atomic operations. We have made these restrictions 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. The McCAT C compiler was designed to support parallelization and optimization,(9) and it provides: a simple high-level intermediate representation for dataflow analysis;(10) advanced pointer analyses;(11–13) and array dependence testing.(14, 15) Threaded-C is a lower-level language where the programmer specifies all threads and synchronization among threads.(6) [See Fig. 13c for an example of Threaded-C code.] 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 runtime system handles the actual scheduling of threads find 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.

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 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 Fig. 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,

```
{  a = f(b);
   foo(a);
   goo(b);
}
```

(a) ordinary

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

(b) parallel

```
{!  a = f(x);
    b = g(y);
    foo(a,b);
!}
```

(c) strictly sequential

Fig. 1. Types of 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.

However, in some cases a programmer may want to explicitly specify parallel or strictly sequential sequences. Figure 1b gives a typical use of a parallel sequence, denoted using the syntactic construct $\{\,^\wedge \ldots {}^\wedge\}$. Parallel statement sequences are used to explicitly specify that there are no dependences among the statements (except dependences on shared variables as explained in Section 2.6), and they may be safely executed in parallel. This conduct is used primarily when the compiler's dependence analysis cannot detect the parallelism automatically. In Fig. 1b 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 1c gives an example of a strictly sequential sequence, as denoted by the syntactic construct $\{! \ldots !\}$. 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 (for example, they may want to limit the amount of parallelism exposed). In Fig. 1c 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 2a gives an example of two for loops

```
{^  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

Fig. 2. Examples of statement sequences.

that can execute in parallel. Furthermore, just like blocks in C, statement sequences can be arbitrarily nested. Fig. 2b gives an example with a parallel sequence nested inside an ordinary sequence.

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. [Dependence within an iteration are allowed as the compiler will perform the appropriate renaming in the parallelization phases.] We can express the forall loop construct as follows:

forall ($v = init\_expr$; $test\_expr(v)$; $v = update\_expr(v)$) $loop\_body(v)$

The operational semantics of the forall loop can be stated as follows. Let $v_1 \dots v_n$ be the values generated by executing the loop header. Then all loop bodies $loop\_body(v_1)$, $loop\_body(v_2), \dots, loop\_body(v_n)$ **may** be executed in parallel. Figure 3 gives two examples of forall loops. The first, in Fig. 3a, 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 Fig. 3b, 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.

2.3. 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:

`forall (i=0; i<N; i++)`
`  { a[i] = b[i] * s; }`
	`forall (p=hd; p!=NULL; p=p->next)`
	`    { f(p); }`

(a) completely parallel	(b) with sequential header

Fig. 3. Examples of forall loops.

• **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.

The programmer with these assumptions can write their program using ordinary C declarations, and the compiler is guaranteed to produce correct, although possibly inefficient, code. Consider the example in Fig. 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. [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.] 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.

```
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);
}
```

Fig. 4. Example of local vs. remote memory accesses.

Note that the communication in EARTH-C is implicit. For example, in the program in Fig. 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.

We have extended the type system in EARTH-C to allow the user to provide additional information about data mapping and locality. There are two 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 Fig. 5a. 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 Fig. 5a. 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

`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

Fig. 5. Examples of declarations.

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 *remote* or *local.* By default a pointer is remote and can point to a remote memory location. Local pointers point to local memory locations, and are created by using local * instead of * in type declarations. As pointer declarations are read from right to left in C, in Fig. 5b we are declaring p to be a remote pointer, q to be a local pointer, r to be an remote pointer to a remote pointer, s to be a local pointer to a remote 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 Fig. 6. In this case, the parameter root in the function incr_tree can be declared as a local pointer since 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.

2.4. Remote and Basic 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, they may be invoked on any processor, and the function invocation may access remote

```
void incr_tree(NODE 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);
    ^}
}

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

Fig. 6. Example of local pointers and basic functions.

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 be 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, $p \geqslant 0$, 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 *basic* functions which have more restrictions, but less overhead. In general, the programmer should make a function basic if it contains no useful parallelism and it accesses only local memory. Basic functions are indicated by including the keyword basic at the beginning of the function declaration. A basic function will be invoked on the processor from which it was called, and thus calls to basic functions may be inlined. Furthermore, basic functions must access only memory local to the current processor, and they may only call basic functions. If the programmer states that a function is basic, then the compiler may assume that all direct and indirect memory references are local.

Consider a typical example in Fig. 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 basic 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 basic 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.

The main idea of the program in Fig. 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 basic function. If we had omitted the keyword basic 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 runtime 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.

2.6. Shared Variables and Atomic Functions

As indicated in Sections 2.1 and 2.2, EARTH-C computations that are executed in parallel must not interfere. That is, statements *S*1 and *S*2 cannot be executed in parallel if *S*1 writes to a location read or written by *S*2, or if *S*2 writes to a location read or written by *S*1. This restriction leads to a very simple language, and it also allows us to apply our advanced pointer and dependence analyses to EARTH-C programs. If we allowed interfering computations, then our flow analysis would have to allow for arbitrary interleavings of updates to program variables in parallel computations. However, this restriction also limits the expressibility of EARTH-C. Thus, we have extended the core EARTH-C language to include support for shared variables and atomic functions. The basic idea is that statements *S*1 and *S*2 may be executed in parallel if they interfere only on shared variables. Further, shared variables may only be read or written using atomic operations. We provide built-in atomic operations to read, write and perform simple arithmetic/logical operations on shared variables. We also provide a mechanism for user-defined atomic functions which operate on shared variables. We first give two simple examples, and then give a more detailed account of shared variables and user-defined atomic functions.

2.6.1. Simple Examples

Consider the function find_sum in Fig. 7a. The for loop in this function must be executed sequentially since there is a loop-carried dependence on the variable sum. However, the order in which the elements are summed is immaterial. If we could provide a way of updating sum atomically, then the iterations could proceed in parallel. Figure 7b gives the parallel EARTH-C program that uses a forall loop. Note that the variable

```
double find_sum(double a[], int n)
{ int i;
  double sum;

  sum = 0;
  for (i=0; i<n; i++)
    sum += a[i];

  return(sum);
}
```

(a) Sequential Version

```
double find_sum(double a[], int n)
{ int i;
  shared double sum;

  writeto(&sum,0);
  forall (i=0; i<n; i++)
    addto(&sum,a[i]);

  return(valueof(&sum));
}
```

(b) Parallel Version

Fig. 7. Summing an array using a shared variable.

sum has been declared as shared, and thus the loop can be considered a parallel loop, even though there are loop-carried dependences on sum. Also note that all reads and writes to sum must be done via atomic functions. In this case the initialization of sum is done via writeto, the addition to sum has been done via a built-in atomic primitive called addto and the final value of sum is accessed via the built-in primitive value_of.

In many cases simple uses of shared variables, and built-in atomic operations are sufficient. However, the programmer may sometimes want to specify a more complex atomic function. Consider the example in Fig. 8a. The function find_furthest_distance iterates through an array of points, updating the variable furthest_distance each time distance(p[i]) is greater than the current value of furthest_distance. In order to perform this loop in parallel it is necessary to declare furthest_distance as a shared variable (since there is a loop-carried dependence), and to perform the check and update on furthest_distance atomically. The EARTH-C solution is given in Fig. 8b. Note that the parallel version defines an atomic function, update_furthest, that checks and updates the shared variable furthest_distance. Also note that outside of the atomic function the shared variable must be written/read using the built-in operations writeto/valueof.

2.6.2. Declaring shared *Variables*

In general, shared variables are declared using the keyword shared which is a type-modifier which affects the type to its left. The syntax of declaring a *shared* variable is as follows: ⟨type⟩ shared ⟨var_id⟩. Examples of different declarations of shared variables are given in Fig. 9. The first group of declarations shows examples of basic types like int, double, char, etc. The second group of declarations gives examples of pointers and shared variables. Note that p points to a shared integer. Thus, *p is considered to be a shared variable and the programmer must use atomic operations to operate on *p. The declaration of p1 is a shared pointer to

```
double find_furthest_distance(point p[])
{ int i;
  double furthest_distance;

  furthest_distance = distance(p[0]);
  for ( i = 1 ; i < MAXP ; i++)
    if (distance(p[i]) > furthest_distance)
      furthest_distance = distance(p[i]);

  return (furthest_distance);
}
```

(a) Sequential Version

```
atomic void update_furthest(shared double *furthest, double d)
{ if (d > *furthest) *furthest = d; }

double find_furthest_distance(point p[])
{ int i;
  shared double furthest_distance;

  writeto(&furthest_distance,distance(p[0]));
  forall (i = 1; i < MAXP; i++)
    update_furthest(&furthest_distance, distance(p[i]));

  return (valueof(&furthest_distance));
}
```

(b) Parallel Version

Fig. 8. Finding the furthest distance.

an integer. Hence, *p1 is not a shared object and we can access it without using atomic operations. However, we have first to get the value of p1 through an atomic operation, then we can de-reference the result. Another example is p2, which is a shared pointer to a shared integer. In this case, p2 and *p2 are considered shared objects and their access should be done through atomic functions. The fourth group of declarations is for shared arrays. When we specify an array to be shared, that means that the array elements are shared. In the last group of declarations, we have a shared structure and a pointer to a shared structure. In this case, we have each field of the structure as shared. We do not support any operations on the structure as a whole; we can not assign a normal structure to a shared structure.

Shared variables must be accessed via built-in atomic operations or user-defined atomic functions. Further, shared pointers may not be copied or cast to non-shared pointers. All such errors are caught at compile-time by the EARTH-C compiler.

2.6.3. User-Defined atomic *Functions*

As illustrated in Fig. 8b, it is possible for the user to define atomic functions. In order to guarantee that the underlying EARTH runtime system can execute the function atomically, we place the following restrictions on any atomic function. An atomic function F must have the following properties:

1. F must have a declaration of the form:
   atomic *ReturnType* F (*Type* shared *S, Q_1,..., Q_n), where:
   - The first parameter, S must be a pointer to a shared variable.
   - The *ReturnType* and the types of parameters Q_1,..., Q_n must NOT be structures or shared variables.

```
int shared y;             /* shared integer */
double shared r;          /* shared double */

int shared * p;           /* pointer to a shared integer    */
int * shared p1;          /* shared pointer to int */
int shared * shared p2;   /* shared pointer to shared int */

int shared  M [100];      /* array of shared integers */
char shared * names[10];  /* array of pointers to shared characters */
int * shared values[5];   /* array of shared pointers to integers */

struct ss shared * p_st  ;/* pointer to a shared structure */
struct ss shared my_st;   /* shared structure */
```

Fig. 9. Examples of shared variable declarations.

2. The body of F must not contain any remote accesses.
3. The body of F must not contain any calls to remote functions or other atomic functions. It may, however, contain calls to basic functions.
4. F may not be called using a user-specified call site (i.e., a call of the form F(...)@exp is not allowed). F will always be automatically invoked at the owner of the shared variable given as the first argument to F.

These restrictions on atomic functions guarantee that the generated code will be a single thread, and will thus be executed atomically.

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 Fig. 10, the compiler can be divided into three main phases. Phase I is composed of the standard

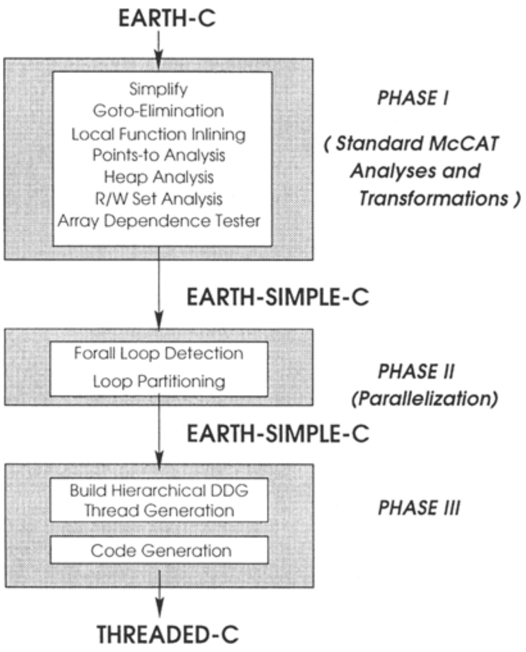


Fig. 10. Overall structure of the compiler.

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. [Here, we will use the terms SIMPLE and SIMPLE-EARTH-C interchangeably.] 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.

In phase II we detect for loops that can be safely transformed to forall loops, and we restructure some forall loops so they expose function-level and/or thread-level parallelism. The main focus of this paper is on phase III, the translation from SIMPLE-EARTH-C to Threaded-C. This process consists of partitioning the program into appropriate threads, and then generating the target Threaded-C code. In order to understand the constraints on generating threads, it is important to understand the underlying EARTH execution model.

3.1. EARTH Execution Model and EARTH/MANNA

In the EARTH model, a multiprocessor consists of multiple EARTH nodes and an interconnection network.[6, 8] As illustrated in Fig. 11, each EARTH node consists of an *Execution Unit* (EU) and a *Synchronization Unit* (SU), linked together by buffers. The SU and EU share a local memory, which is part of a *distributed shared memory* architecture in which

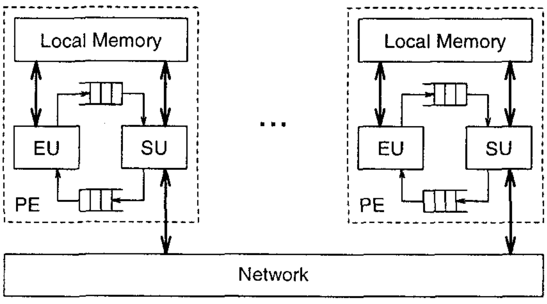


Fig. 11. The EARTH architecture.

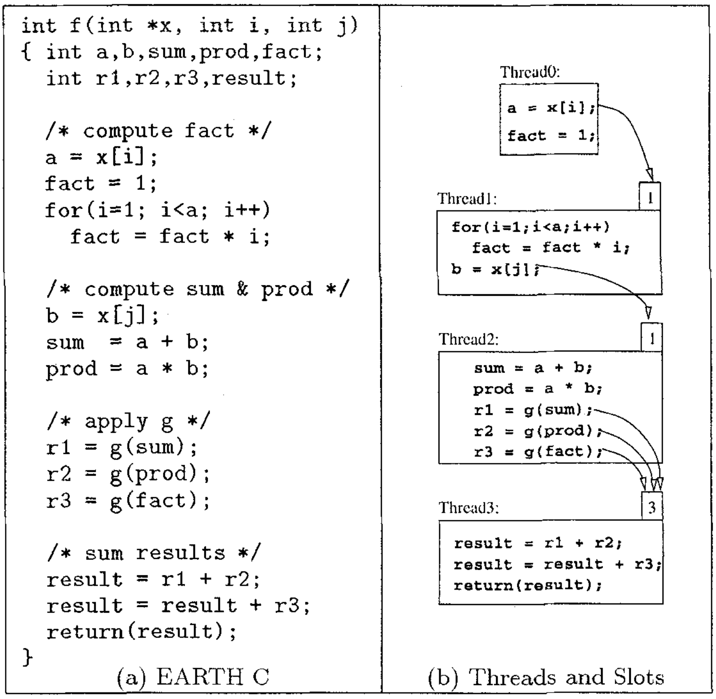
the aggregate of the local memories of all the nodes represents a global memory address space. The EU processes instructions in an active thread, where an active thread is initiated for execution when the EU fetches its *thread id* from the ready queue. The EU executes a thread to **completion** before moving to another thread. It interacts with the SU and the network by placing messages in the event queue. The SU fetches these messages, plus messages coming from remote processors via the network. The SU responds to remote synchronization commands and requests for data, and also determines which threads are to be run and adds their thread ids to the ready queue.

Our experiments have been done using a multithreaded emulator built on top of the MANNA parallel machine.(16) Each MANNA node consists of two Intel i860 XP CPUs, clocked at 50 MHz, 32 MB of dynamic RAM and a bidirectional network interface capable of transferring 50 MB/S in each direction. The two processors on each node are mapped to the EARTH EU and SU. The EARTH runtime system supports efficient remote operations. Sequentially, loading a remote word takes about 7 μs, calling a remote function can be performed in 9 μs, and spawning a new remote thread takes about 4 μs. When issued in a pipeline these operation take only one third of these times. Thus, placing many independent remote operations into one thread and issuing them in a pipeline may reduce the total execution time. However, even when pipelined, each remote operation and each spawn does impose significant overhead. Thus, the thread-generator must try and reduce the number of remote operations and the number of threads, while attempting to overlap communication with computation.

Another constraint on the thread generator is that **all** 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. The request and the use **must** be in different threads. 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 basic function) is invoked, its invocation 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 uses the value. Consider the example C program in Fig. 12a. 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 12b 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



```
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 & 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

```
Thread0:
  a = x[i];
  fact = 1;

Thread1:                    1
  for(i=1;i<a;i++)
    fact = fact * i;
  b = x[j];

Thread2:                    1
  sum = a + b;
  prod = a * b;
  r1 = g(sum);
  r2 = g(prod);
  r3 = g(fact);

Thread3:                    3
  result = r1 + r2;
  result = result + r3;
  return(result);
```

(b) Threads and Slots

Fig. 12. Naive thread creation.

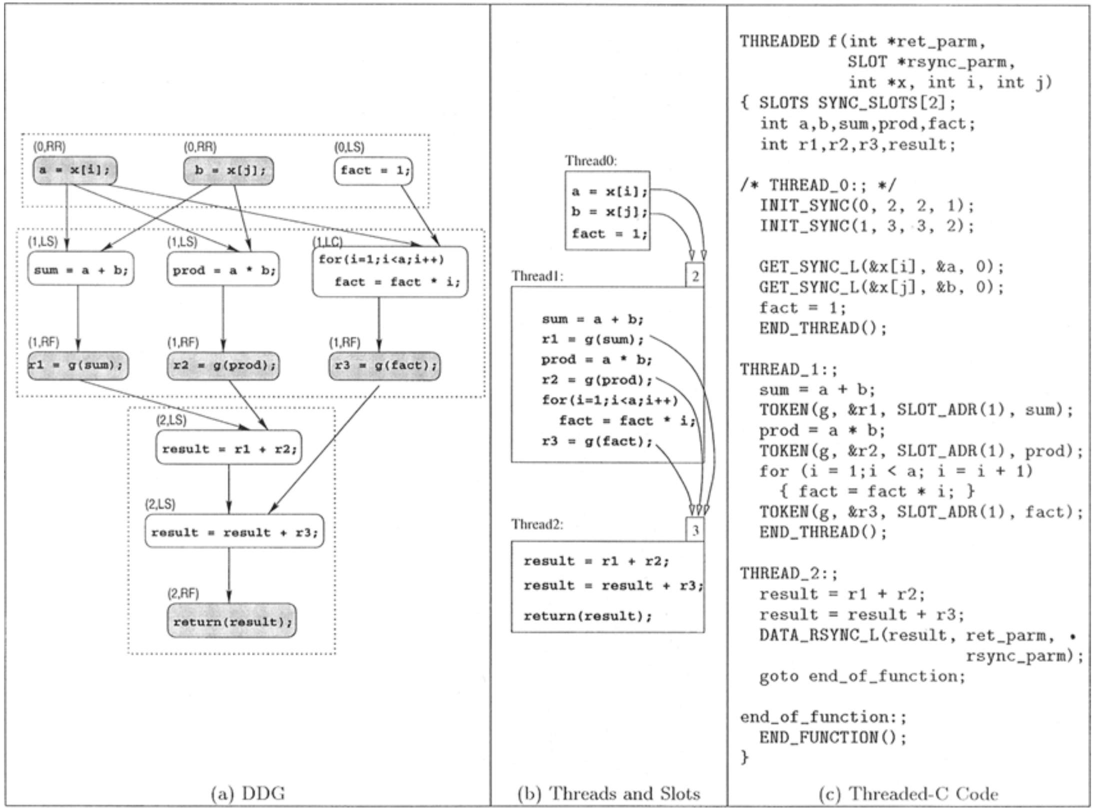
with their appropriate slot. For example, the completion of the remote read of x[i] in Thread 0 signals Thread 1. Thread 1 is ready as soon as it receives one signal. In this naive scheme, the slot for $Thread_i$ is associated with all the split-phase operations that occur in $Thread_{i-1}$.

3.2. 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.[11] This gives us quite precise DDGs, and exposes parallelism even in the presence of pointers. Each node in the DDG is given two labels: *statement type* and *earliest thread number*. Figure 13a gives the labeled DDG for the example program from Fig. 12a. The rules for computing the labels and methods for generating the threaded-C code are described in the following subsections.

Statement Type: Each node is labeled with one of the types summarized in Table I. 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 any variable reference that is remote, the statement is labeled as REMOTE. The DDG along with the statement types for the example program of Fig. 12a is given in Fig. 13a. 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, and the list scheduler would be recursively applied to it.

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 $max(thread\_num(P_1) + 1, \ldots, thread\_num(P_k) + 1)$.



```
THREADED f(int *ret_parm,
           SLOT *rsync_parm,
           int *x, int i, int j)
{ SLOTS SYNC_SLOTS[2];
  int a,b,sum,prod,fact;
  int 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_ADR(1), sum);
  prod = a * b;
  TOKEN(g, &r2, SLOT_ADR(1), prod);
  for (i = 1;i < a; i = i + 1)
    { fact = fact * i; }
  TOKEN(g, &r3, 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();
}
```

Fig. 13. DDG and list scheduling.

Table I. Statement Types for Scheduling

Statement Type	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/loop stmt. that accesses only local memory & 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/loop statement that accesses remote memory or calls a remote function

Based on these observations, we can assign earliest thread numbers to each node in the DDG using a top-down search, assigning the maximum thread numbers based on the thread numbers of the predecessors. An example of applying our algorithm is given in Fig. 14 (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 the statement sequence is minimized and thus the average thread length is maximized.

After assigning the earliest thread number and statement type, we use a list scheduling strategy to actually form the threads, and to calculate the

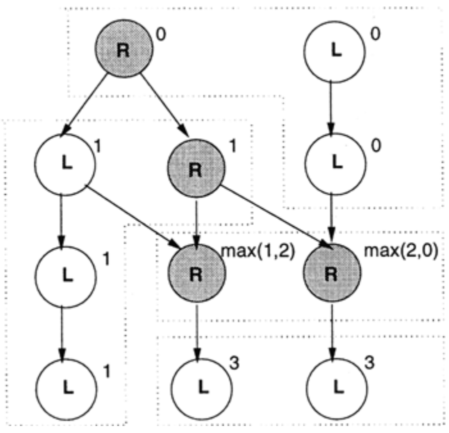


Fig. 14. Earliest thread number assignment.

synchronization counts. 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 $i+1$. Furthermore, we schedule all statements with the same earliest thread number i together into thread number i. 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 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 Fig. 15. Note that when we schedule a remote operation in thread i, we must increment the synchronization count of the earliest thread which depends on the result of the remote operation. This can be done by looking at all successors in the DDG, and taking the smallest of the earliest thread numbers stored in those successors.

Figure 13b shows the result of our list scheduling algorithm when applied to the DDG in Fig. 13a. Note that compared to the naive thread generation approach of Fig. 12b, 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.

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. If the number of loop iterations can be computed before the loop executes, the synchronization associated with the last thread in the body is moved outside of the loop. Thus, for loops which have only one thread in the body, multiple iterations can run in parallel and a barrier synchronization is enforced at the end of the loop.

```
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;
   init all threads with sync count = 0;
   i = 0; /* start with thread 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;
         }
       add N to thread i;
       add successors of N that are now ready to ready_list;
       /* if a remote op, then increment sync count of
          earliest thread to depend on result of this op */
       if N.type = R then
         { s_thread_num = min thread_num of successors of N;
           increment sync count of s_thread_num;
         }
     }
   /* mark end of last thread */
   mark the end of thread i;
}
```

Fig. 15. List scheduling algorithm.

3.3. 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 Fig. 13c. 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 a Threaded-C GET_SYNC and each remote write is translated into a DATA_SYNC. Calls to basic functions are not modified, but calls to remote functions are compiled into TOKEN or INVOKE calls. The TOKEN mechanism is used when the runtime load balancer should be used, and the INVOKE mechanism is used when an explicit processor number is given.

Shared variables and atomic functions are handled slightly differently. Built-in shared operations like addto are compiled into the appropriate primitive based on the type of the argument. For example, a statement of the form addto(&x, 3) is transformed into addto_i(MAKE_GPTR(&x, OWNER_OF(x)), 3)@OWNER_OF(x), where addto_i is a built-in atomic function for integers. Atomic functions are translated into THREADED functions, and the body of each atomic function is compiled into exactly one thread. Since atomic functions are always executed on the processor owning the shared variable, and atomic functions execute as one uninterrupted thread, we can guarantee atomic computations on the shared variable.

4. EXPERIMENTAL RESULTS

In this section we examine the effectiveness of our approach. To evaluate the core part of EARTH-C (as described in Sections 2.1 to 2.5), we have experimented with six benchmarks as described in Table II. 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 and tomcatv are small examples more typical of scientific codes.

Table II. Benchmark Programs

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

Table III. Benchmark Features: Compiler (⊙), Naive (●), Advanced (○)

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

For the first four benchmarks, we started with the original sequential programs and introduced EARTH constructs for parallelism and locality. In Table III, 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 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 basic 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 we had to develop and optimize our own Threaded-C programs. Table IV 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 last two columns give the ratio of the time for the EARTH-C versions with respect to the hand-coded Threaded-C versions.

Power System Optimization: The power benchmark solves a power system optimization by computing the best price for the benefit of the community.[17] The problem is based on a four-level tree structure, with different branching widths at each level. The program executes a series of

Table IV. Experimental Measurements

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	45.7 sec	0.82	0.85	0.94	0.87	0.90
	8 procs		6.35	6.56	6.76	0.94	0.97
	16 procs		12.46	12.86	12.61	0.99	1.02

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 $\{^\wedge \dots ^\wedge\}$ 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 runtime system. As the runtime 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 subtrees is allocated to the processor owning the complete sub-tree. Thus, computations on the lower two levels are specified as **basic** functions.

The results in Table IV is quite encouraging. Even the naive version gets 90% of the performance of the handcoded 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 runtime 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 Ref. 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 basic 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 IV 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.(19, 20) This heuristic algorithm first searches a suboptimal tour for each subtree(region) and then merges subtours into bigger ones. The tour found is built as a circular linked list sitting on 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 basic functions to throttle the recursive parallelism. The data in Table IV 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 runtime 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 IV 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. In general, for both mmult and tomcatv, the performance of the EARTH-C versions is quite reasonable. 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.

4.1. Using Shared Variables and Atomic Functions

We have also experimented with using shared variables and atomic functions. We found that two of our benchmarks, power and tomcatv could use shared variables to simplify the code. In power, a shared variable is used to accumulate the sum of several subcomputations that proceed in parallel. In tomcatv shared variables are used to accumulate the maximum of several parallel subcomputations. In both cases the original version used an array to store partial sums/maximums for the parallel computations, and then computed the final sum/maximum by iterating through this array. We found that using a shared variable made the program more readable, and had very little impact on performance.

However, further experiments showed that one must be careful in the use of shared variables. We experimented with shared variables in the perimeter benchmark, which finds the perimeter of a raster imagage that is encoded using a quad-tree.(21, 22) There are two places where shared variables can be used. The first is in a recursive routine that computes the number of leaf nodes in the quadtree, and the second is in actually computing the perimeter. We found that using a shared variable for computing the number of nodes resulted in a slight performance loss, whereas using shared variables for computing the perimeter resulted in a slight performance gain.

In the case of computing the number of nodes, the code was not simplified by using shared variables. The performance loss was due to contention for the shared variable, and the overhead of many accesses to shared variables. Since each access to a shared variable is compiled into a function call to a THREADED function, one must be careful not to overuse shared variables, particularly when a local variable can be used instead (as was the case in this benchmark).

In the case of computing the perimeter, the code using shared variables was simpler and smaller than the code without shared variables. The performance gain was due to the reduction in the amount of code that needed to be executed (for example, a loop summing the results of four recursive calls was not needed in the shared variable version).

We also experimented with using shared variables and atomic functions to implement a version of Barnes-Hut that uses a master/slave type of synchronization. For this experiment we took a Threaded-C program that implemented this mechanism, and developed a similar higher-level EARTH-C program using shared variables in atomic operations. One interesting result was that our EARTH-C version was about 20%–30% faster than the hand-coded Threaded-C version. This appears to be because the hand-coded version had not been fine-tuned for performance, whereas our EARTH-C compiler generated reasonably good threads automatically.

In summary, it seems that shared variables and atomic functions are quite useful in simplifying some types of computations, and in allowing the user to express more parallel paradigms. However, with the current implementation of shared variables and atomic functions, there is a definite cost for each access to a shared variable, and they should be used with care. A more detailed description of experiments using shared variables may be found in Ref. 23.

5. RELATED WORK

In this section, we outline some related work, focusing mostly 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.(24) Split-C provides a global address space supporting both local and global pointers. Split-C follows an SPMD model of program execution. The authors state 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.(25) 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.

Cilk is a C-based language and runtime system for multithreaded parallel programming,(26) especially for explicit continuation-passing style programming.(27) 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 basic 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.(28) 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 accesses and synchronization must be explicitly specified by the programmer.

Olden is a compiler and runtime system for C programs on distributed memory machines.(18, 22, 29) The philosophy of the design of Olden is very similar to that of EARTH-C. In both cases they were designed to minimize the burden on the programmer, and to have the compiler automatically handle communication and synchronization. In Olden, the programmer uses a *future* notation to expose parallelism, and the Olden compiler uses sophisticated compile-time techniques to combine computation migration and software caching to exploit locality. Unlike EARTH-C, Olden is not specifically targeted towards a multithreaded architecture. Thus, the compilation problems are different in the two approaches. In compiling EARTH-C, a large part of the problem is in generating the appropriate threads, whereas in Olden it is deciding when to use software caching and when to do computation migration. The target applications of the two projects are quite similar, and many of the benchmarks used in our experiments come from the Olden benchmark suite.

Overall, a distinct feature of EARTH-C is that it provides the programmer a more traditional high-level language that does not explicitly support threads, and it uses 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 simple 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 that 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.

Our experiments with shared variables and atomic operations showed that these features did help in simplifying some programs, and in providing support for new parallel paradigms. We also showed that shared variables must be used carefully, and overuse can cause performance degradation.

Based on our experiences we plan to improve the compiler to do more locality analysis and collective communication automatically. We are currently implementing a dataflow analysis based on type inference that detects when pointer variables must be local, must be remote, or could be either local or remote. This analysis will be used to decrease the number of split-phase operations required, and to selectively clone procedures to improve locality and automatically create basic procedures. 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.

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, *Proc. of the 22nd Ann. Intl. Symp. on Computer Architecture*, Santa Margherita Ligure, Italy, pp. 2–13 (June 1995).
2. 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, *Proc of the First Intl. EURO-PAR Conf.*, No 996, *Lec. Notes in Comp. Sci.*, Stockholm, Sweden, Springer-Verlag, pp. 101–116 (August 1995).
3. Yuetsu Kodama, Hirohumi Sakane, Mitsuhisa Sato, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi, The EM-X Parallel Computer: Architecture and Basic Performance, *Proc. of the 22nd Ann. Intl. Symp. on Computer Architecture*, Santa Margherita Ligure, Italy, pp. 14–23 (June 1994).
4. Michael D. Noakes, Deborah A. Wallah, and William J. Dally, The J-Machine Multicomputer: An Architectural Evaluation, *Proc. of the 20th Ann. Intl. Symp. on Computer Architecture*, San Diego, California, pp. 224–235 (May 1993).
5. 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, *Proc. of the 20th Ann. Intl. Symp. on Computer Architecture*, San Diego, California, pp. 302–313 (May 1993).
6. 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, *I. J. P. P.* **24**(4): 319–347 (August 1996).
7. Olivier C. Maquelin, Herbert H. J. Hum, and Guang R. Gao, Costs and Benefits of Multithreading with Off-the-shelf RISC Processor, Springer-Verlag, Stockholm, Sweden, *Proc. of the First Intl. EURO-PAR Conf.*, No. 966, *Lec. Notes in Comp. Sci.*, pp. 117–128 (August 1995).

8. Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian, Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling, *Proc. of the 23rd Ann. Intl. Symp. on Computer Architecture*, Philadelphia, Penn., pp. 178–188 (May 1996).
9. L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan, Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations, *Proc. of the Fifth Intl. Work. on Languages and Compilers for Parallel Computing, Lec. Notes in Comp. Sci.*, No. 757, New Haven, Conn., pp. 406–420 (August 1992); Springer-Verlag (1993).
10. Bhama Sridharan, An Analysis Framework for the McCAT Compiler. Master's Thesis, McGill University, Montréal, Québec (September 1992).
11. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren, Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers, *Proc. of the ACM SIGPLAN '94 Conf. on Progr. Lang. Design and Implementation*, Orlando, Florida, pp. 242–256 (June 1994).
12. Rakesh Ghiya and Laurie J. Hendren, Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C, *Conf. Rec. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles Progr. Lang.*, St. Petersburg, Florida, pp. 1–15 (Jan. 1996).
13. Rakesh Ghiya and Laurie J. Hendren, Connection Analysis: A Practical Interprocedural Heap Analysis for C, *IJPP* **24**(6) 547–578 (December 1996).
14. Justiani and Laurie J. Hendren, Supporting Array Dependence Testing for an Optimizing/Parallelizing C Compiler, *Proc. of the Fifth Intl. Conf. on Compiler Construction, Lec. Notes in Comp. Sci.*, No. 786, Springer-Verlag, Edinburgh, Scotland, pp. 309–323, (April 1994).
15. Christopher Lapkowski, A practical symbolic analysis for array dependences. Master's Thesis, McGill University, Montréal, Québec (April 1997).
16. U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat, Latency hiding in Message-Passing Architectures, *Proc. of the 8th Intl. Parallel Processing Symp.* Cancún, Mexico, IEEE Comp. Soc., pp. 704–709 (April 1994).
17. S. Lumetta, L. Murphy, X. Li, D. Culler, and I. Khalil, Decentralized Optimal Power Pricing: The Development of a Parallel Program. ACM SIGARCH and IEEE Comp. Soc. *Proc. of Supercomputing '93*, Portland, Oregon, pp. 240–249 (November 1993).
18. Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren, Supporting Dynamic Data Structures on Distributed-Memory Machines, *ACM Trans. Progr. Lang. Syst.* **17**(2):233–263 (March 1995).
19. Richard M. Karp, Probabilistic Analysis of Partitioning Algorithms for the Travelling-Salesman Problem in the Plane, *Mathematics of Operations Research* **2**(3): 209–224 (August 1977).
20. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*. MIT Press, McGraw-Hill Book Co., Cambridge, Massachusetts (1990).
21. Hanan Samet, Computing Perimeters of Regions in Images Represented by Quadtrees, *IEEE Trans. Pattern Analysis and Machine Intelligence* **3**(6):683–687 (November 1981).
22. Martin C. Carlisle, Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines. Ph.D. Thesis, Princeton University Department of Computer Science (June 1996).
23. Shereen Ghobrial, Extending EARTH-C with Shared Variables and Atomic Operations. Master's Thesis, McGill University, Montréal, Québec (April 1997).
24. David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick, Parallel Programming in Split-C. Portland, Oregon, ACM SIGARCH and IEEE Comp. Soc., *Proc. of Supercomputing '93*, pp. 262–273, (November 1993).

25. Rishiyur S. Nikhil, Cid: A Parallel, "Shared-Memory" C for Distributed-Memory Machines, *Proc. of the 7th Intl. Work. on Languages and Compilers for Parallel Computing*, No. 892, *Lec. Notes in Comp. Sci.*, pp. 376–390, Ithaca, New York, (August 1994); Springer-Verlag (1995).
26. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou, Cilk: An Efficient Multithreaded Runtime System, *Proc. of the Fifth ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming* (*PPOPP*), Santa Barbara, California, pp. 207–216 (July 1995).
27. Michael Halbherr, Yuli Zhou, and Chris F. Joerg, MIMD-Style Parallel Programming Based on Continuation-Passing Threads. CSG Memo 355, Computation Structures Group, MIT Lab. for Comp. Sci. (March 1994).
28. Mitsuhisa Sato, Yuetsu Kodama, Shuichi Sakai, and Yoshinori Yamaguchi, EM-C: Programming with Explicit Parallelism and Locality for the EM-4 Multiprocessor, *Proc. of the IFIP WG 10.3 Working Conf. on Parallel Architectures and Compilation Techniques, PACT '94*, North-Holland Pub. Co., Montréal, Québec, pp. 3–14 (August 1994).
29. Martin C. Carlisle and Anne Rogers, Software Caching and Computation Migration in Olden, *Proc. of the Fifth ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming* (*PPOPP*), Santa Barbara, California, pp. 29–38 (July 1995).

Printed in Belgium