Compiling C for the EARTH Multithreaded Architecture

Laurie J. Hendren
Xinan Tang
Yingchun Zhu
Guang R. Gao (Deleware)
Xun Xue (IBM Toronto)
Haiying Cai
Pierre Ouellet

School of Computer Science

McGill University

Montreal, Canada

Goals

1. Design simple high-level extensions for C that allow programmers to write programs that will run efficiently on multi-threaded architectures.
   (EARTH-C)

2. Develop compiler techniques to automatically translate programs written in EARTH-C to multi-threaded programs.
   (EARTH C $\rightarrow$ Threaded-C)

3. Determine if EARTH-C + compiler can compete with hand-coded Threaded-C programs.

EARTH C (Coarse Grain Parallelism)

Statement Sequences

Ordinary	`{  s1; s2; ...; sn  }`
Parallel	`{^ s1; s2; ...; sn ^}`
Sequential	`{! s1; s2; ...; sn !}`

Forall Loops

forall (*v* = *init_expr* **;**
test_expr **(** *v* **);**
v = *update_expr* **(** *v* **)**
)
loop_body **(** *v* **)**

Example:

```
forall (p=head; p != NULL; p = p->next)
  { f(p); }
```

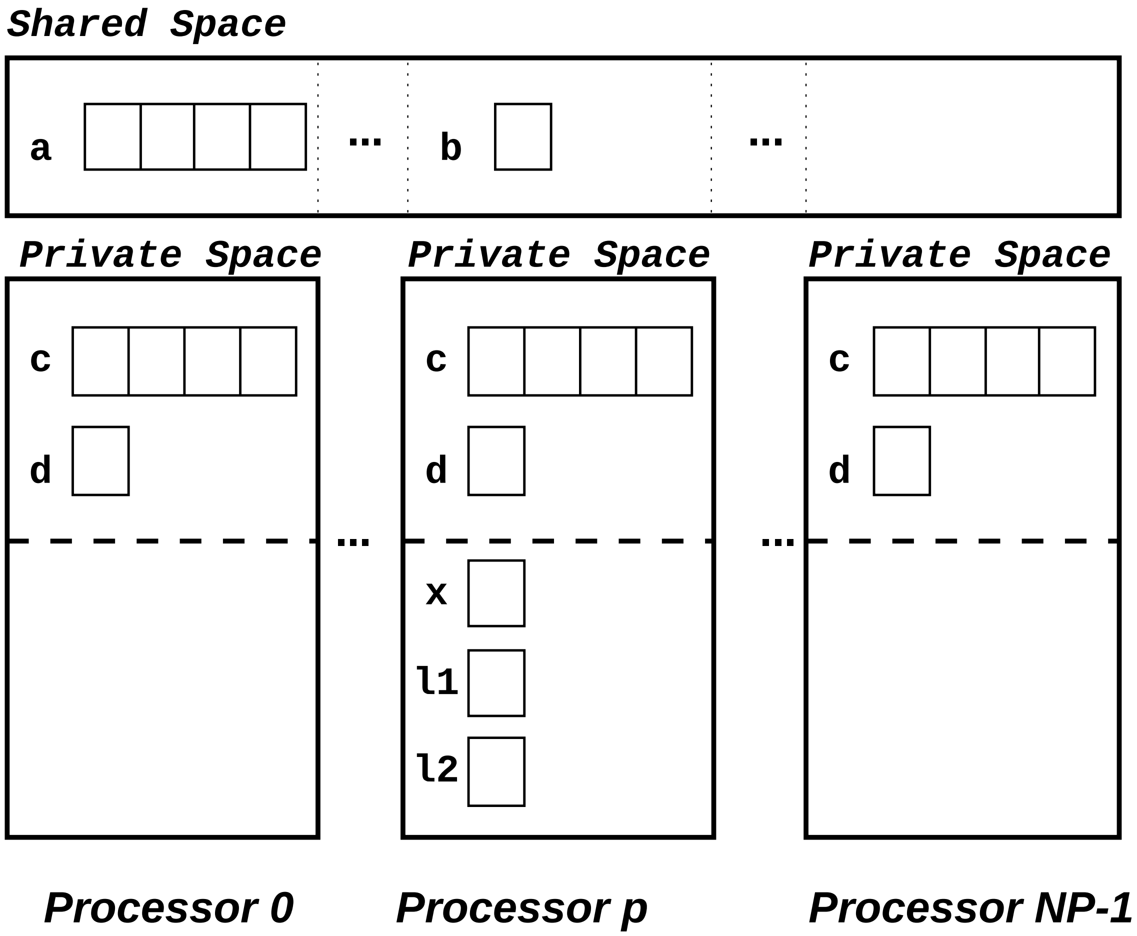
Remote Functions

- Function calls are assumed to remote, unless the function is declared as basic.
- Memory accesses in the function may be remote.
- An invocation `foo(a1,a2,..,an)` will be assigned to a processor by the run-time system. An invocation `foo(a1,a2,...,an)@my_exp()` will be assigned to the processor denoted by `my_exp()`.

Basic Functions

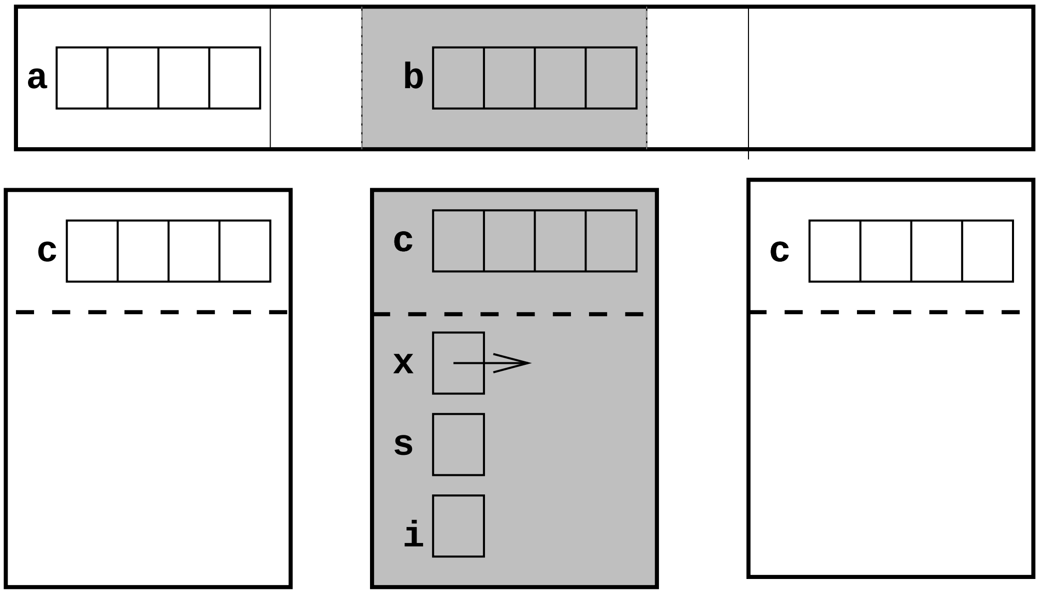
- `basic int foo() ...`
- All memory accesses in a basic function must be local.
- A call to a basic function will be executed on the processor that calls it.
- A basic function may be inlined.

Declarations and Memory Model




```
int a[4], b@p;
replicated int c[4], d;

foo(int x)
{ int l1, l2;
   ...
}
```

Pointers to Local Objects



Processor p

```
scale_vector(int *x, int s)
{ int i;
  for(i=0; i<10; i++)
    x[i] = x[i] * s;
}
```

```
scale_vector(a,10);
scale_vector(b,10)@ANY;
scale_vector(a,10)@myldbal();
```

```
lscale_vector(int local *x, int s)
{ int i;
  for(i=0; i<10; i++)
    x[i] = x[i] * s;
}
```

```
lscale_vector(c,20);
lscale_vector(b,10)@p;
lscale_vector(a,30)@OWNER_OF(a);
```

Summary of EARTH C Extensions

Declarations

```
replicated int x[10];
int a[100]@N;
int local * p;
basic foo ( ... ) { ... }
```

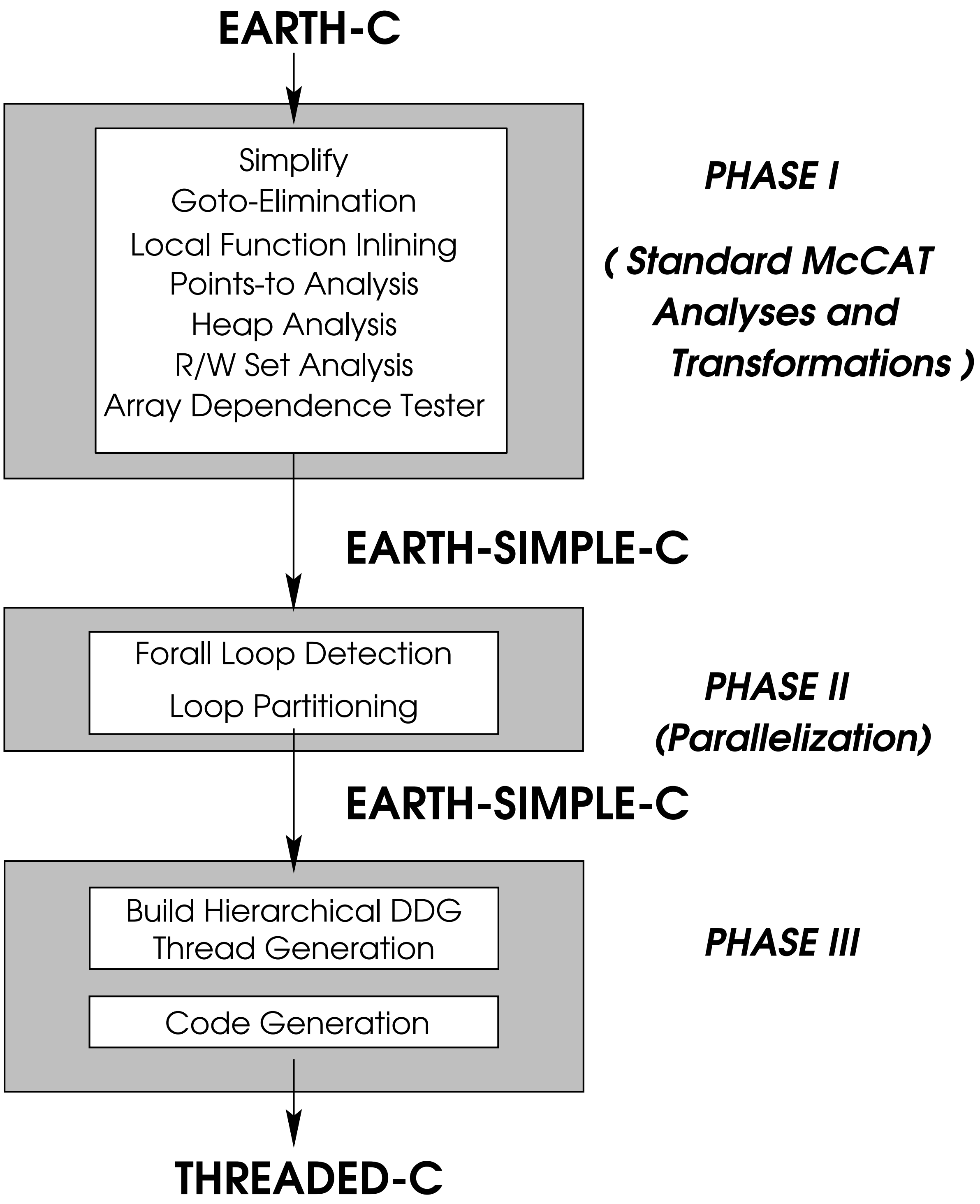
Control Constructs

```
{ ... }
{! ... !}
{^ ... ^}
forall(...) { ... }
p(...) @ OWNER_OF(x)
```

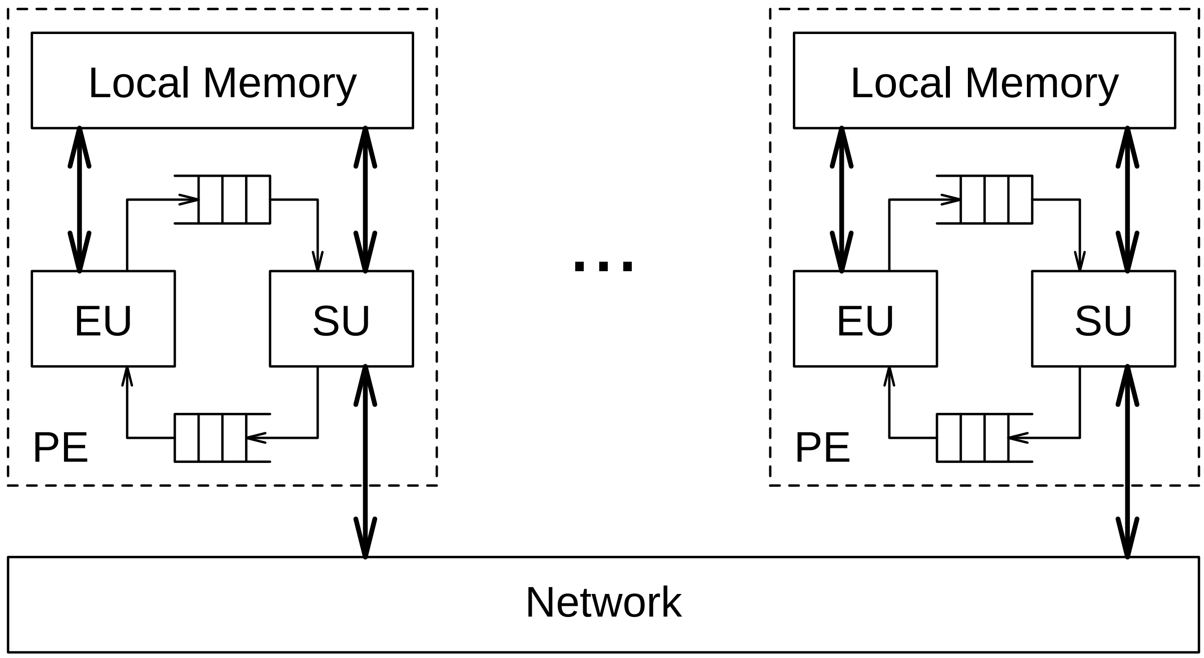
“Built-in” Const/Variables/Functions

```
#define ANY (-1)
replicated int NODE_ID;
int NUM_NODES;
MoveData(&A[i],buf,20)
OWNER_OF(A)
```

The McCAT/EARTH Compiler



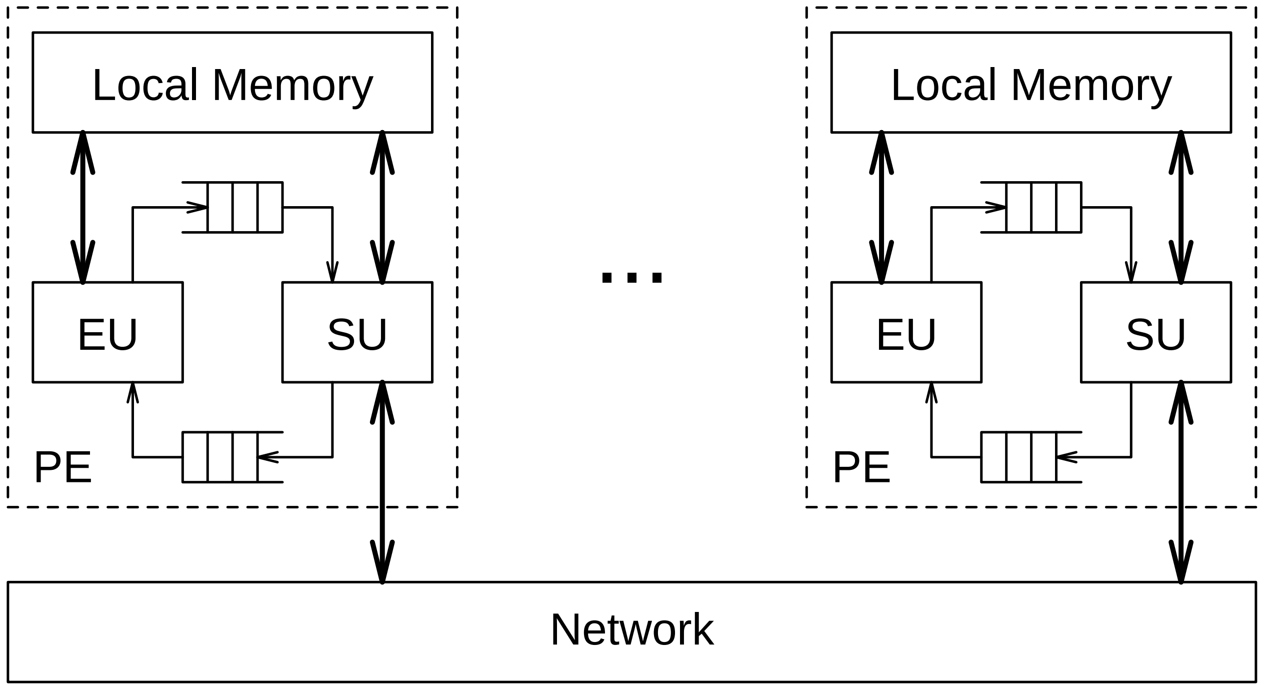
The EARTH Multithreaded Architecture (Model)



- The architecture consists of multiple EARTH nodes and an interconnection network.

- Each processing node consists of an *Execution Unit* (EU) and a *Synchronization Unit* (SU), which are linked together via buffers.

The EARTH Multithreaded Architecture (Model)



- The EU processes instructions of the active thread, until **completion** of the thread.

- The SU responds to synchronization commands, requests for data, and determines which threads are ready to run.

The EARTH Multithreaded Architecture (Testbed)

- Multithreaded emulator build on top of the GMD Manna parallel machine.

- Each processing node consists two Intel i860 CPUs, clocked at 50MHz, with 32 MB of dynamic RAM, and a bidirectional network interface capable of transferring 50MB/s in each direction.

- Efficient remote operations
  loading a remote word takes about 7μs
  calling a remote function about 9μs
  spawning a new thread about 4μs.

- Pipelining remote operations reduces the time to about 1/3rd.

EARTH C → Threaded C (Thread Generation)

Given a sequence of statements, $s_1, s_2, \ldots s_n$, we wish to create threads such that:

- maximize thread length (minimize thread switching overhead)

- retain *sufficient* parallelism

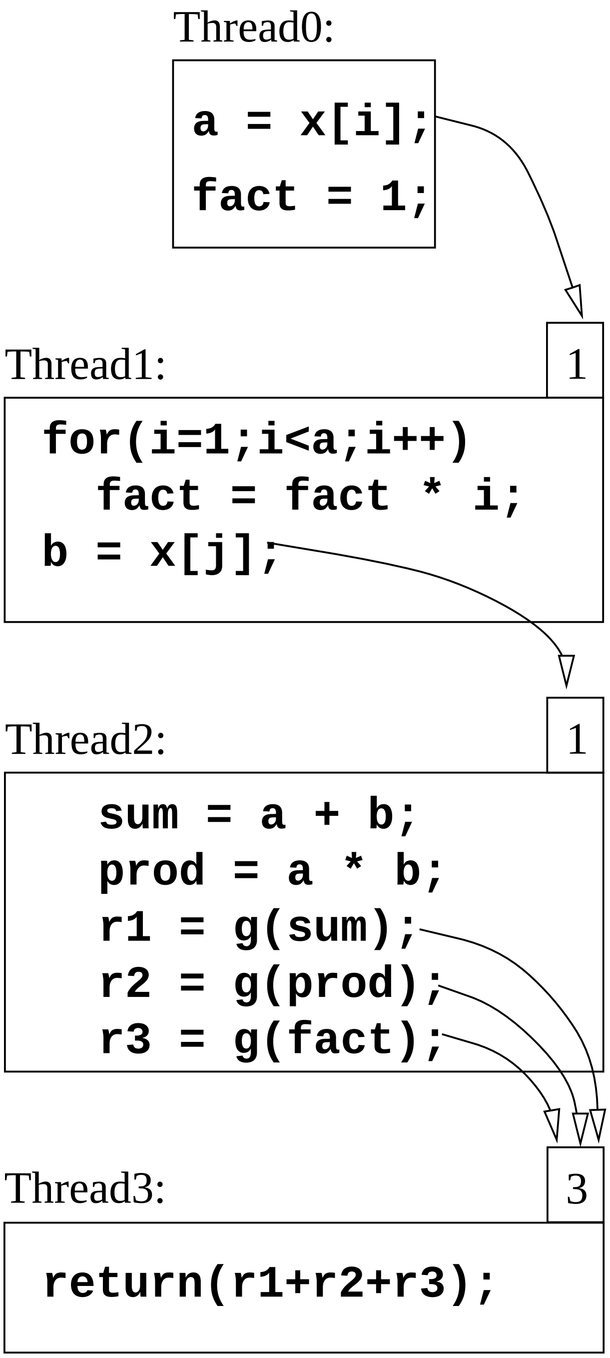
- issue remote memory requests as early as possible (prefetching)

- compile split-phase remote memory operations and remote function calls correctly

Example

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

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

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

  r1 = g(sum);
  r2 = g(prod);
  r3 = g(fact);

  return(r1 + r2 + r3);
}
```

Naive, Top-down Thread Generation



Better Strategy Using List Scheduling

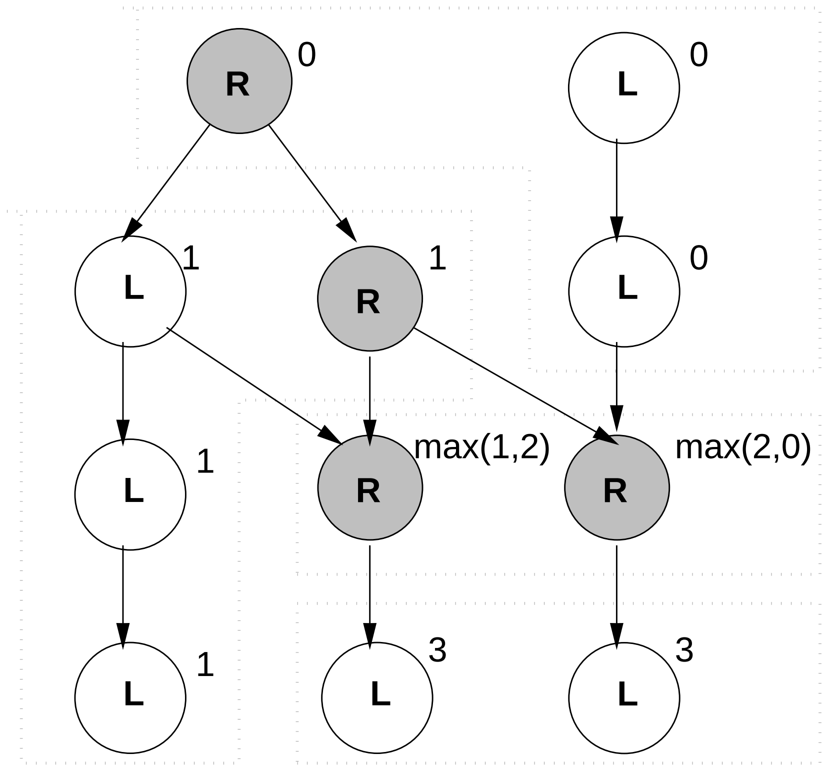
- Put each instruction in the **earliest** possible thread.
- Within a thread, put the remote operations are executed as early as possible.

Build a Data Dependence Graph (DDG), and use a list scheduling strategy, where the selection of instructions is guided by *Earliest Thread Number* and *Statement Type*.

Earliest Thread Number

If instruction Q has n remote ancestors, $P_1 \ldots P_n$, then the earliest possible thread for Q is:

$$1 + max(tnum(P_1), \ldots, tnum(P_n)).$$



Instruction Types

SCHEDULE FIRST

REMOTE_READ,REMOTE_WRITE

REMOTE_FN_CALL

LOCAL_SIMPLE

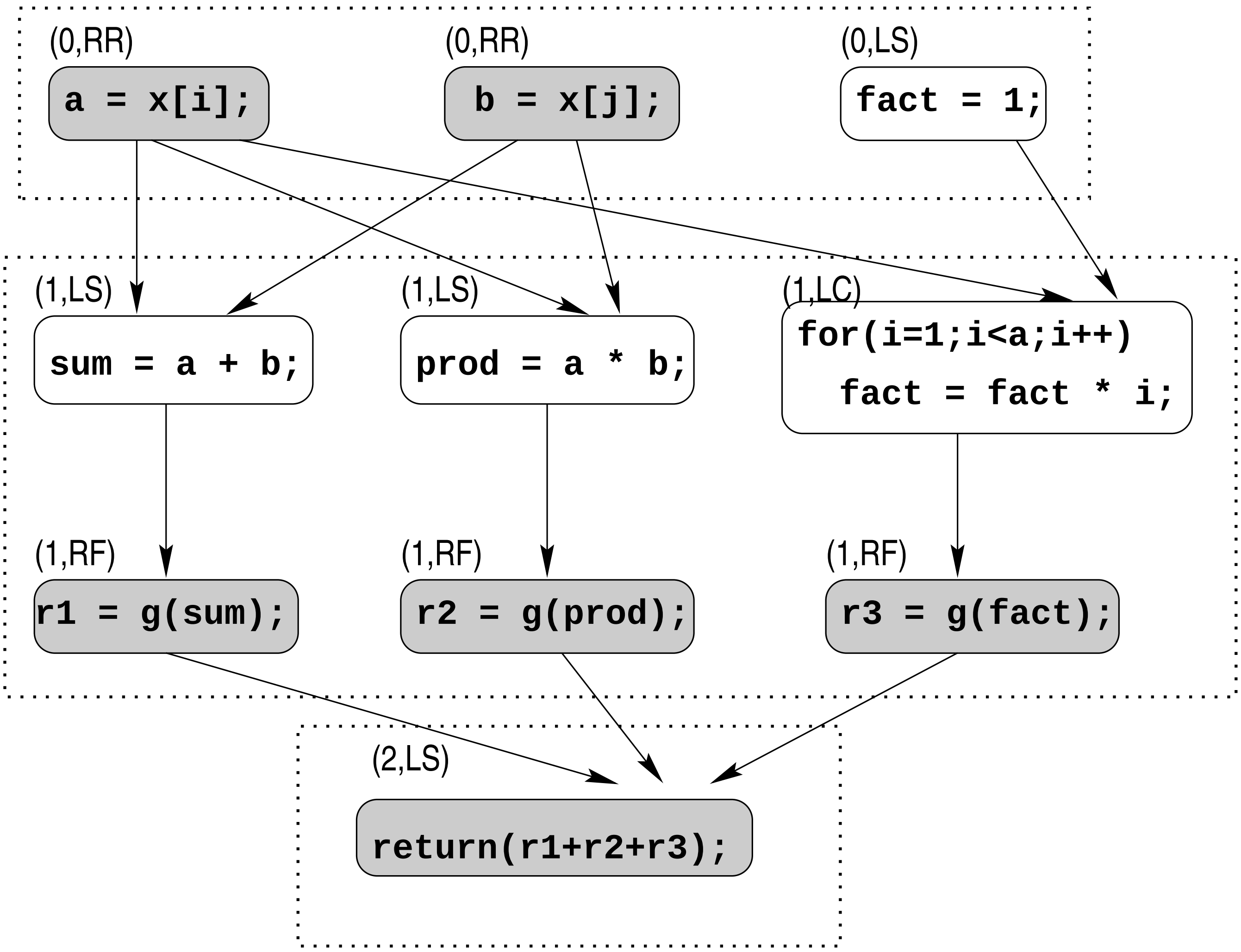
REMOTE_COMPOUND

LOCAL_COMPOUND

BASIC_FN_CALL

SCHEDULE LAST

List Scheduling Previous Example



Resulting List-scheduled Threads

Thread0:

```
a = x[i];
b = x[j];
fact = 1;
```

Thread1: 2

```
sum = a + b;
r1 = g(sum);
prod = a * b;
r2 = g(prod);
for(i=1;i<a;i++)
  fact = fact * i;
r3 = g(fact);
```

Thread2: 3

```
return(r1+r2+r3);
```

Generating Threaded-C Code

```
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;

/* 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:;
  DATA_RSYNC_L(r1+r2+r3, ret_parm, rsync_parm);
  END_FUNCTION();
}
```

Experimental Results

- Develop three versions of each benchmark:

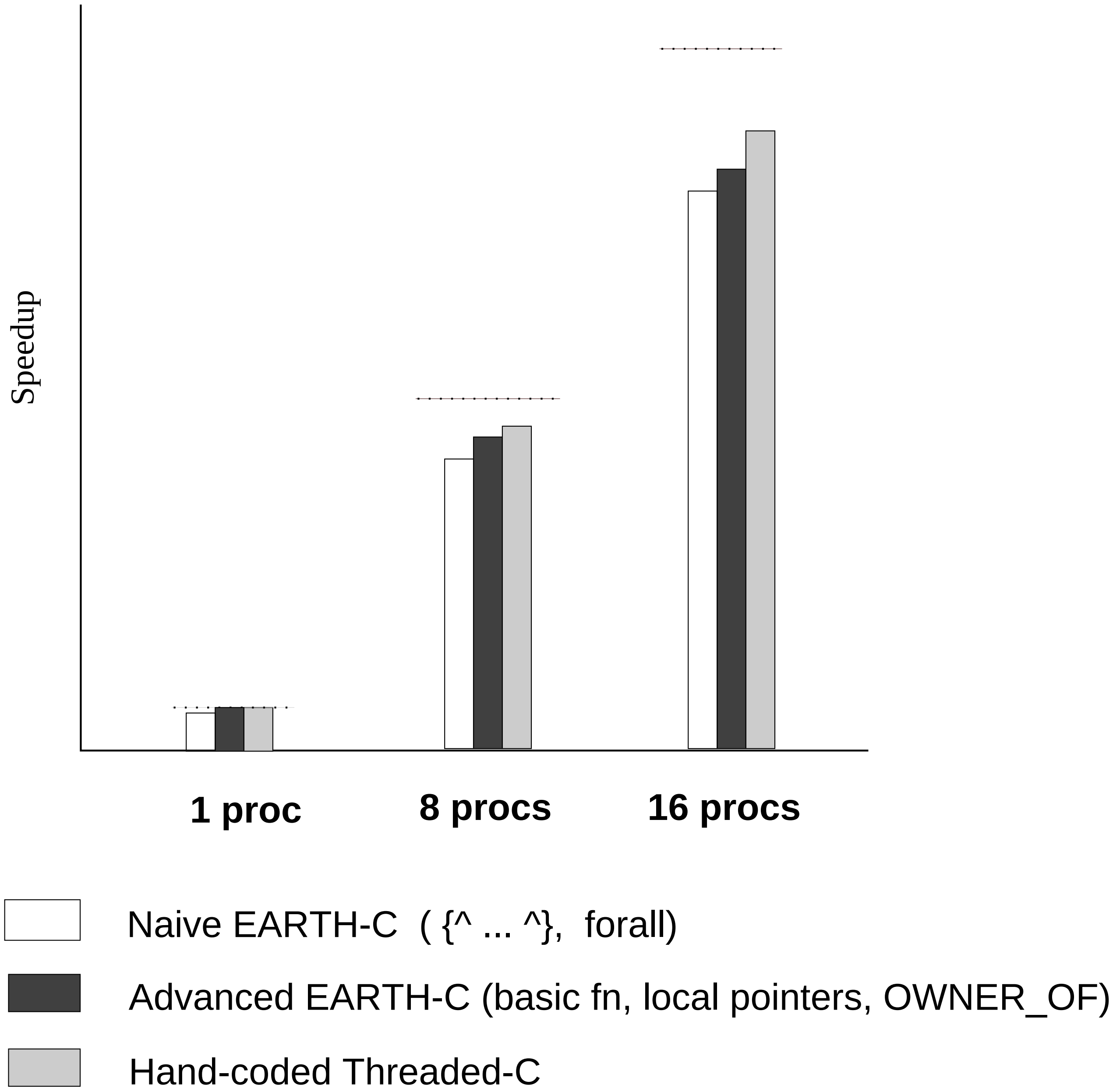
  Naive EARTH-C: Develop a simple parallel version of the program using relatively few EARTH-C constructs.

  Advanced EARTH-C: Improve the performance of the program by using more advanced EARTH-C features like local pointers and basic functions to enhance data locality.

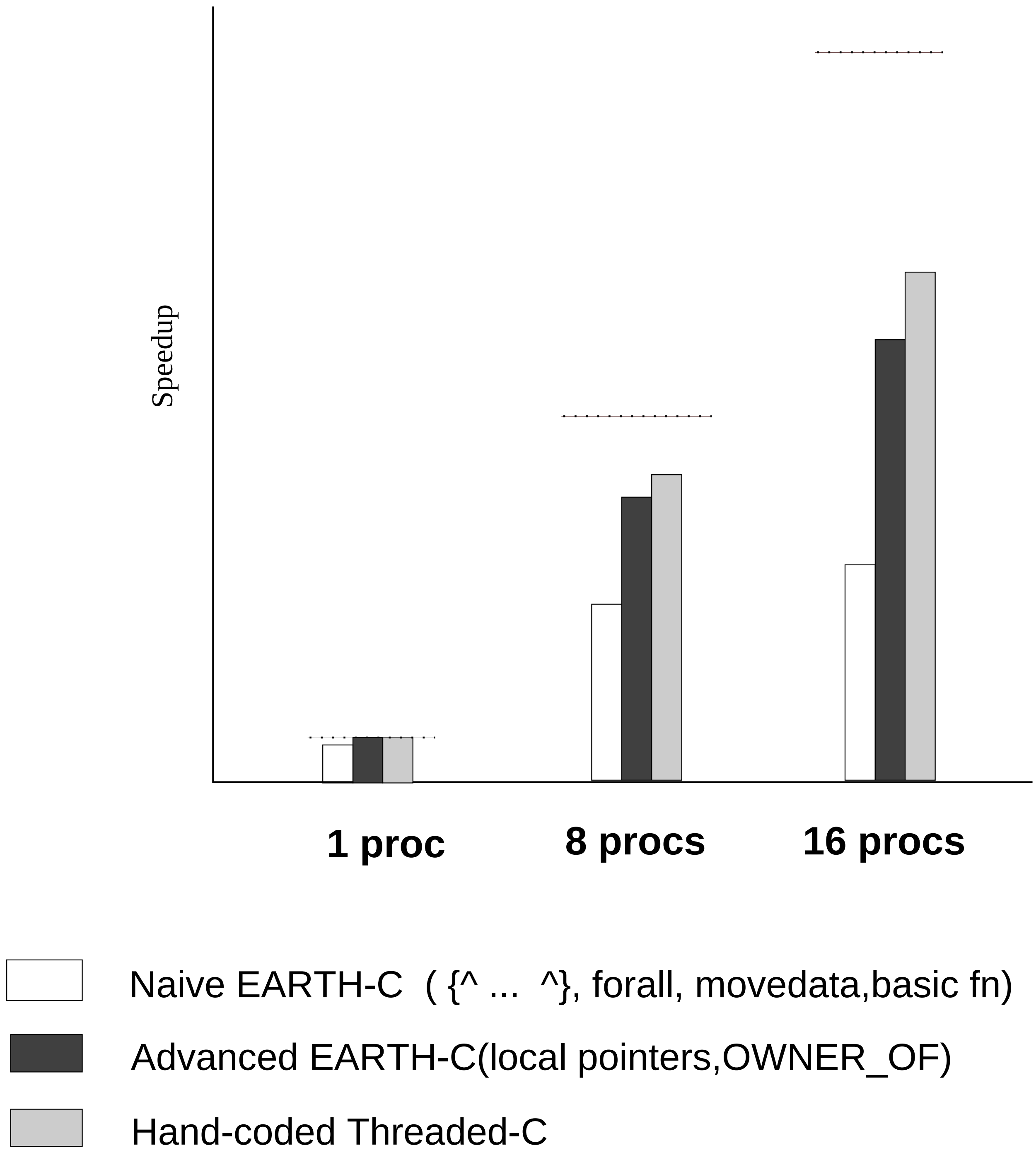
  Hand-coded Threaded-C: The best existing hand-coded parallel program.

- Can we approach the performance of the hand-coded Threaded-C programs, using only EARTH-C + McCAT/EARTH compiler?

Power Benchmark (k-ary tree with 10,000 leaves)



Heuristic TSP Benchmark (32K cities)



What's new and what's not

- There are many related C dialects that are targeted to multi-threaded archtectures including SPLIT-C, Cid, Cilk,EM-C.

- EARTH-C is different in that the language is intended to be very simple, and does **NOT** expose multithreading to the user.

- The McCAT/EARTH compiler is different in that it uses dependence analysis (including pointer analysis) plus information provided via EARTH-C constructs to automatically generate low-level threaded programs.

- We have developed the language and compiler in conjunction with the architecture and run-time system.

Ongoing and Future Work

- further refinements to EARTH-C, motivated by our experiences with more applications (shared variables and atomic functions)

- develop more sophisticated thread generation strategies based on a cost model

- design new pointer analyses that can infer locality information, and automatically produce specialized functions

- continue to experiment with new applications.