Optimizing and Parallelizing Compilers:

How good are they?

Laurie J. Hendren

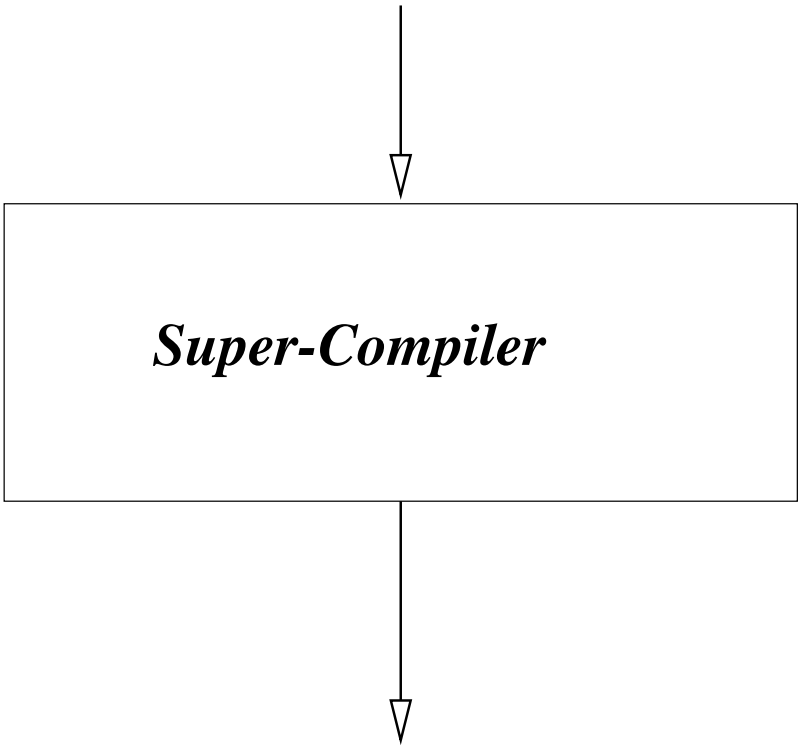
School of Computer Science

McGill University, Montréal, Canada

What should an optimizing/parallelizing compiler do?

Program written in a very high-level, modern, (parallel) programming language

"Good for the programmer"



Efficient assembler/machine code for a modern, high-performance, (parallel) architecture

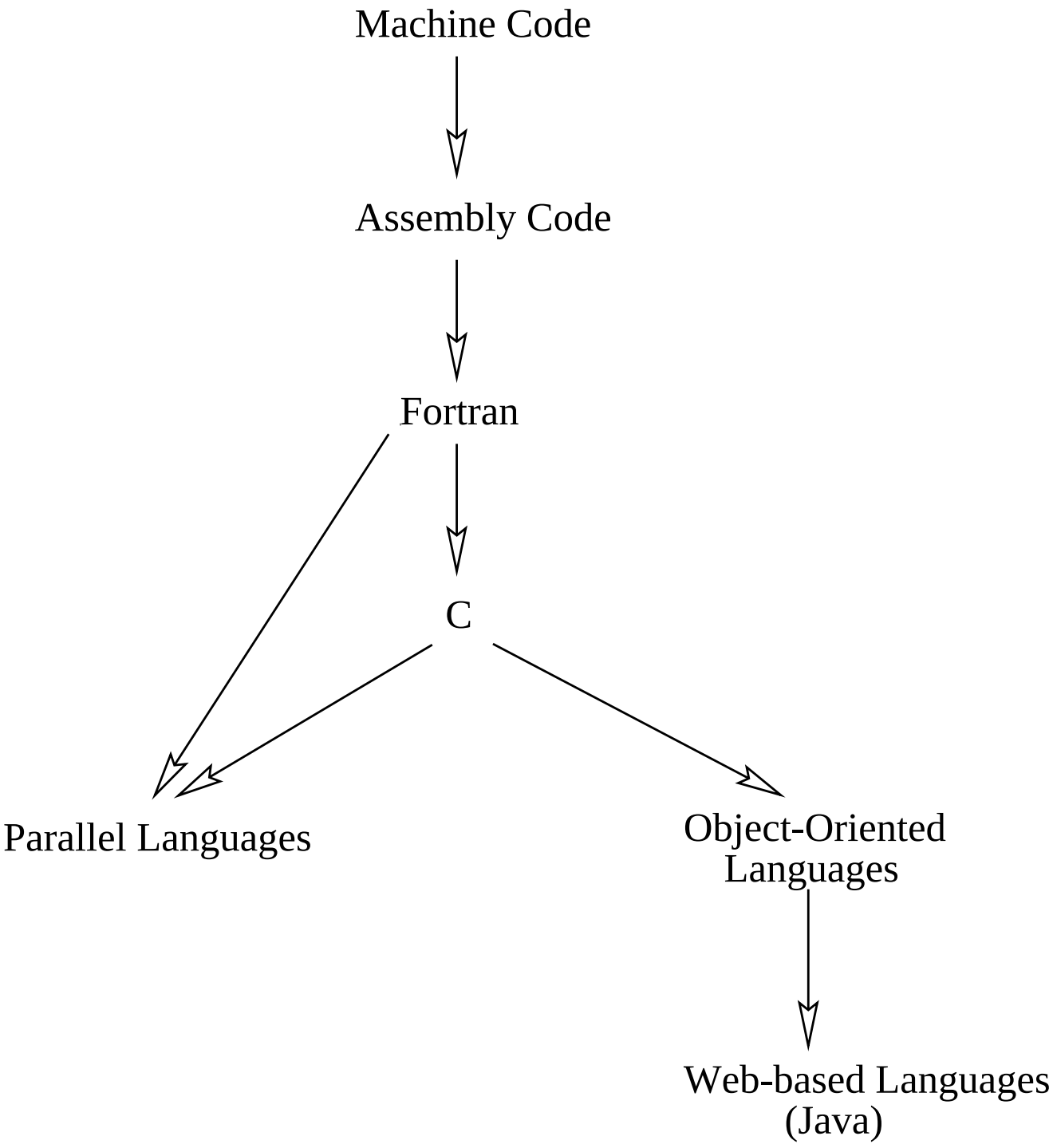
"Good for the computer"

Can such an optimizing/parallelizing compiler be built?

Outline

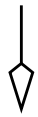
- Historical Perspective
- An Optimizing Compiler (C/McCAT)
- A Parallelizing Compiler (C/EARTH-C)
- Conclusions

Increasing Level of Programming Language Abstraction



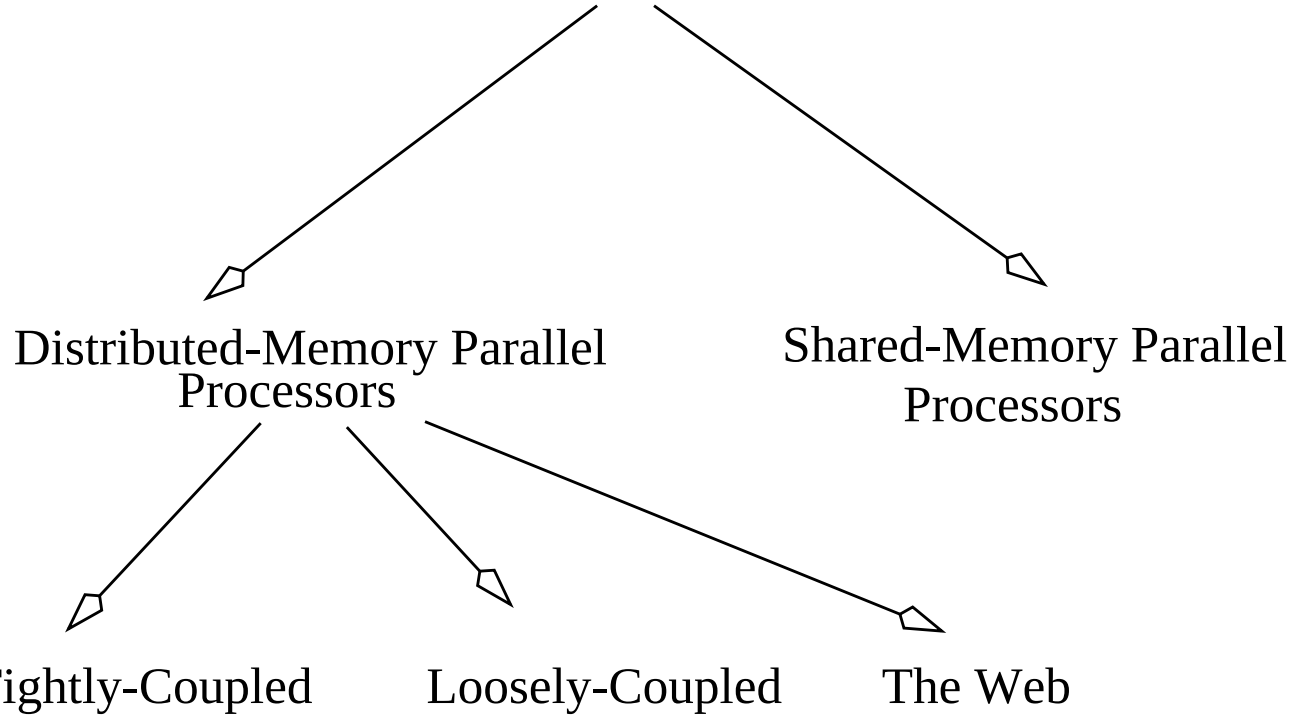
Increasing Architecture Complexity

Simple "von Neumann" Architectures
(small register sets, small main memory)

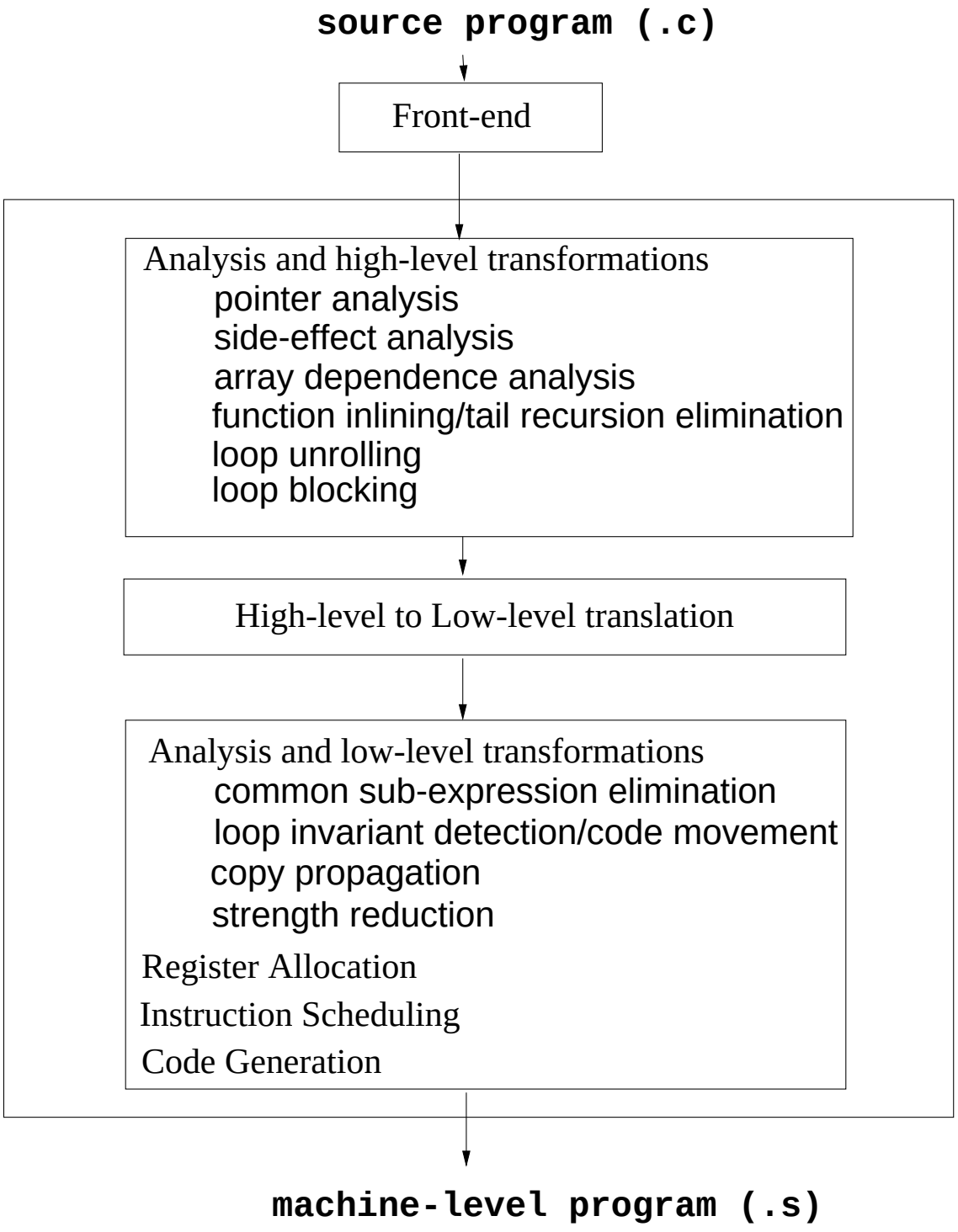


"Simple" RISC/CISC
(larger register sets, more main memory)

"Advanced" RISC/CISC
(register windows, pipelined, superscalar,
multi-level caches,enormous main memory)



Anatomy of an Optimizing Compiler



Naive Code generation

```
for (i = 0; i < 10000; i++)
  { a[i] = a[i] * 2000;
    a[i] = a[i] / 10000;
  }
```

```
L6: ld [%fp-12],%o0       // %o0 = i
    sethi %hi(9999),%o2
    or %o2,%lo(9999),%o1 // %o1 = 9999
    cmp %o0,%o1
    ble L9                // if i <= 9999 goto L9
    nop
    b L7                  // goto L7
    nop
L9: // ---- a[i] = a[i] * 2000; ----
    // ---- a[i] = a[i] / 10000 ----;
    ld [%fp-12],%o1       // %o1 = i
    add %o1,1,%o0         // %o1 = %o1 + 1
    mov %o0,%o1
    st %o1,[%fp-12]       // i = %o1
    b L6                  // goto L6
    nop
L7:
```

Naive code (continued)

```
for (i = 0; i < 10000; i++)
  { a[i] = a[i] * 2000;
    a[i] = a[i] / 10000;
  }
```

```
L9: // ---- a[i] = a[i] * 2000; ----
    ld [%fp-12],%o0     // %o0 = i
    mov %o0,%o1
    sll %o1,2,%o0       // %o0 = i*4
    sethi %hi(_a),%o2
    or %o2,%lo(_a),%o1 // %o1 = &a
    ld [%fp-12],%o2     // %o2 = i
    mov %o2,%o3
    sll %o3,2,%o2       // %o2 = i*4
    sethi %hi(_a),%o4
    or %o4,%lo(_a),%o3 // %o3 = &a
    ld [%o2+%o3],%o2    // %o2 = a[i]
    mov %o2,%o4         // ..
    sll %o4,5,%o3       // ..
    sub %o3,%o2,%o3     // ..
    sll %o3,2,%o4       // ..
    add %o4,%o2,%o4     // ..
    sll %o4,4,%o2       // %o2 = %o2 *2000
    st %o2,[%o0+%o1]i   // a[i] = %o2
```

Optimized Code

```
for (i = 0; i < 10000; i++)
  { a[i] = a[i] * 2000;
    a[i] = a[i] / 10000;
  }
```

```
L8: // %l2 = base address of a, %l1 = i
    sll %l1,2,%l0           // %l0=i*4
    ld [%l0+%l2],%o1        // %o1 = a[i]
    sll %o1,5,%o0           // ..
    sub %o0,%o1,%o0         // ..
    sll %o0,2,%o0           // ..
    add %o0,%o1,%o0         // ..
    sll %o0,4,%o0           // %o0 = a[i]*2000
    st %o0,[%l0+%l2]        // a[i] = %o0
    call .div,0
    or %l4,%lo(10000),%o1 //  %o0 = a[i]/10000
    add %l1,1,%l1           // i++
    cmp %l1,%l5             // i<10000
    ble L8
    st %o0,[%l0+%l2]        // a[i] = %o0
```

Optimized Code with Scalar Replacement

```
for (i = 0; i < 10000; i++)
  { temp = a[i] * 2000;
    a[i] = temp / 10000;
  }
```

```
L8: // %l2 = base address of a, %l1 = i
    sll %l1,2,%l0          // %l0=i*4
    ld [%l0+%l2],%o1       // %o1 = a[i]
    sll %o1,5,%o0          // ..
    sub %o0,%o1,%o0        // ..
    sll %o0,2,%o0          // ..
    add %o0,%o1,%o0        // ..
    sll %o0,4,%o0          // %o0 = a[i]*2000
    call .div,0
    or %l4,%lo(10000),%o1 //  %o0 = %o0/10000
    add %l1,1,%l1          // i++
    cmp %l1,%l5            // i<10000
    ble L8
    st %o0,[%l0+%l2]       // a[i] = %o0
```

Does the optimization work?

opt. level	SPARC		MIPS	
	Original	Scalar	Original	Scalar
no opt	20.5	16.9	21.6	18.4
opt	8.8	8.1	12.3	12.3
super opt	7.9	7.1	11.2	11.3

- Standard compiler optimizations work very well for loops and arrays.

- Adding additional optimizations may not always lead to further speedup.

Should the programmer write "Optimized C"?

If you wanted to write a fast C program, which one should you use, Loop #1 or Loop #2?

```
/* LOOP 1 */
for (i = 0; i < N; i++)
  { a[i] = a[i] * 2000;
    a[i] = a[i] / 10000;
  }
```

```
b = a;
/* LOOP 2 */
for (i = 0; i < N; i++)
  { *b = *b * 2000;
    *b = *b / 10000;
    b++;
   }
```

If you said LOOP #2 ...

You were wrong!

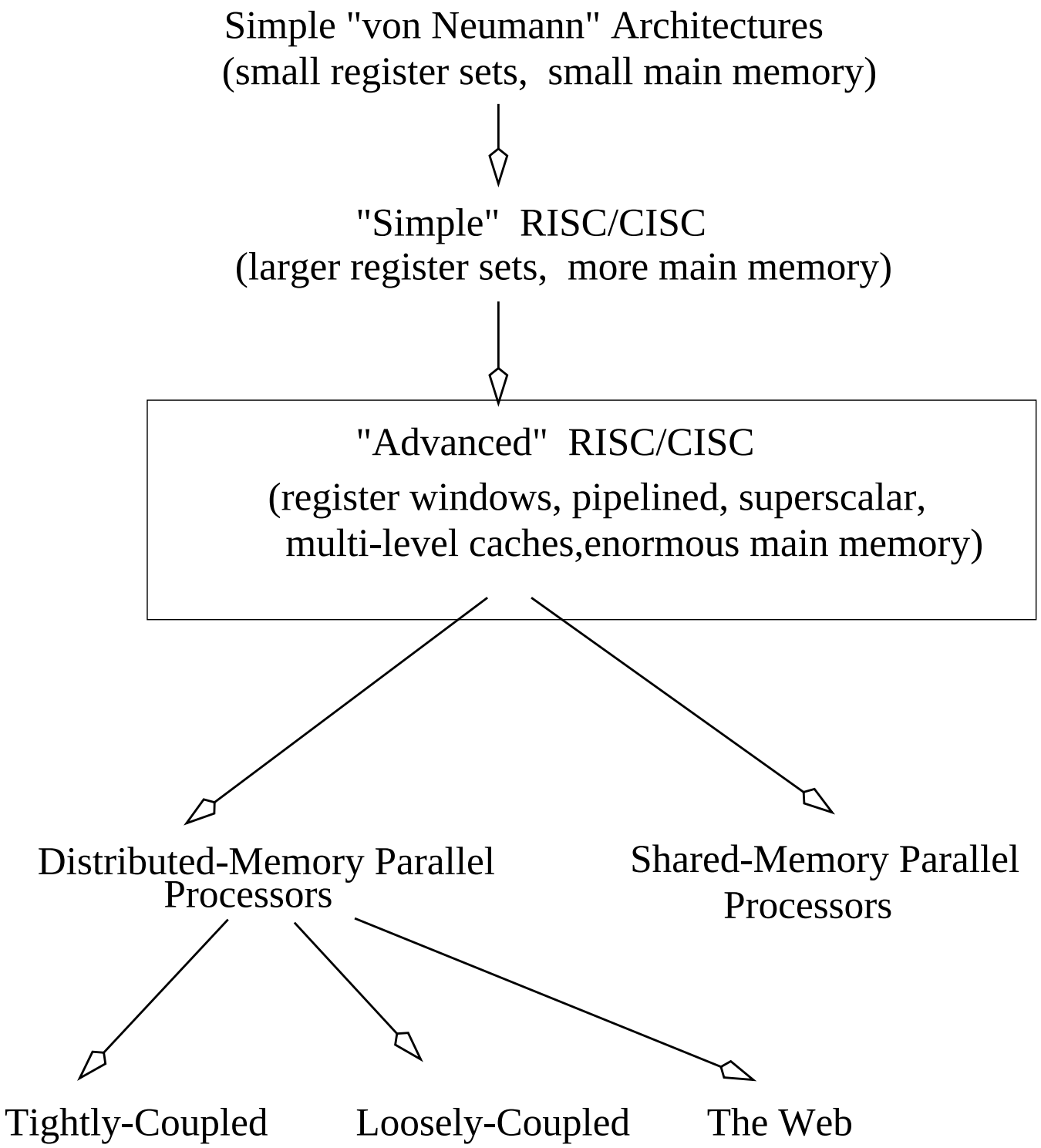
LOOP	opt. level	SPARC	MIPS	Alpha
#1 (array)	no opt	20.5	21.6	7.85
#1 (array)	opt	8.8	12.3	3.26
#1 (array)	super	7.9	11.2	2.96
#2 (ptr)	no opt	19.5	17.6	7.55
#2 (ptr)	opt	12.4	15.4	4.09
#2 (ptr)	super	10.7	12.9	3.94

- Pointers confuse most C compilers. Don't use pointers instead of array references.

- Compilers do a good job of register allocation, don't try to allocate registers in your C program.

- In general, write clear C code, it is easier for both the programmer and the compiler to understand.

Problems due to Architectures



Scheduling example: `a`, `b`, `c` and `d` are scalar variables

```
a = a + b;
c = c + d;
```

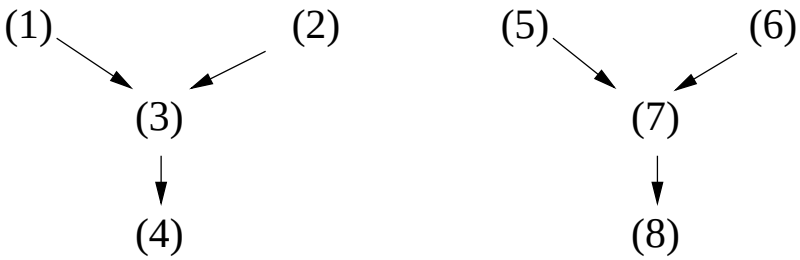
(a) C Program

```
(1)  r1 <-- a
(2)  r2 <-- b
(3)  r2 <-- r1 + r2
(4)  a <--  r2
(5)  r3 <-- c
(6)  r4 <-- d
(7)  r4 <-- r3 + r4
(8)  c <-- r4
```

(b) Instructions

```
(1)  r1 <-- a
(2)  r2 <-- b
     nop
(3)  r2 <-- r1 + r2
     nop
(4)  a <--  r2
(5)  r3 <-- c
(6)  r4 <-- d
     nop
(7)  r4 <-- r3 + r4
     nop
(8)  c <-- r4
```

(c) RISC schedule



(d) Dependencies between instructions

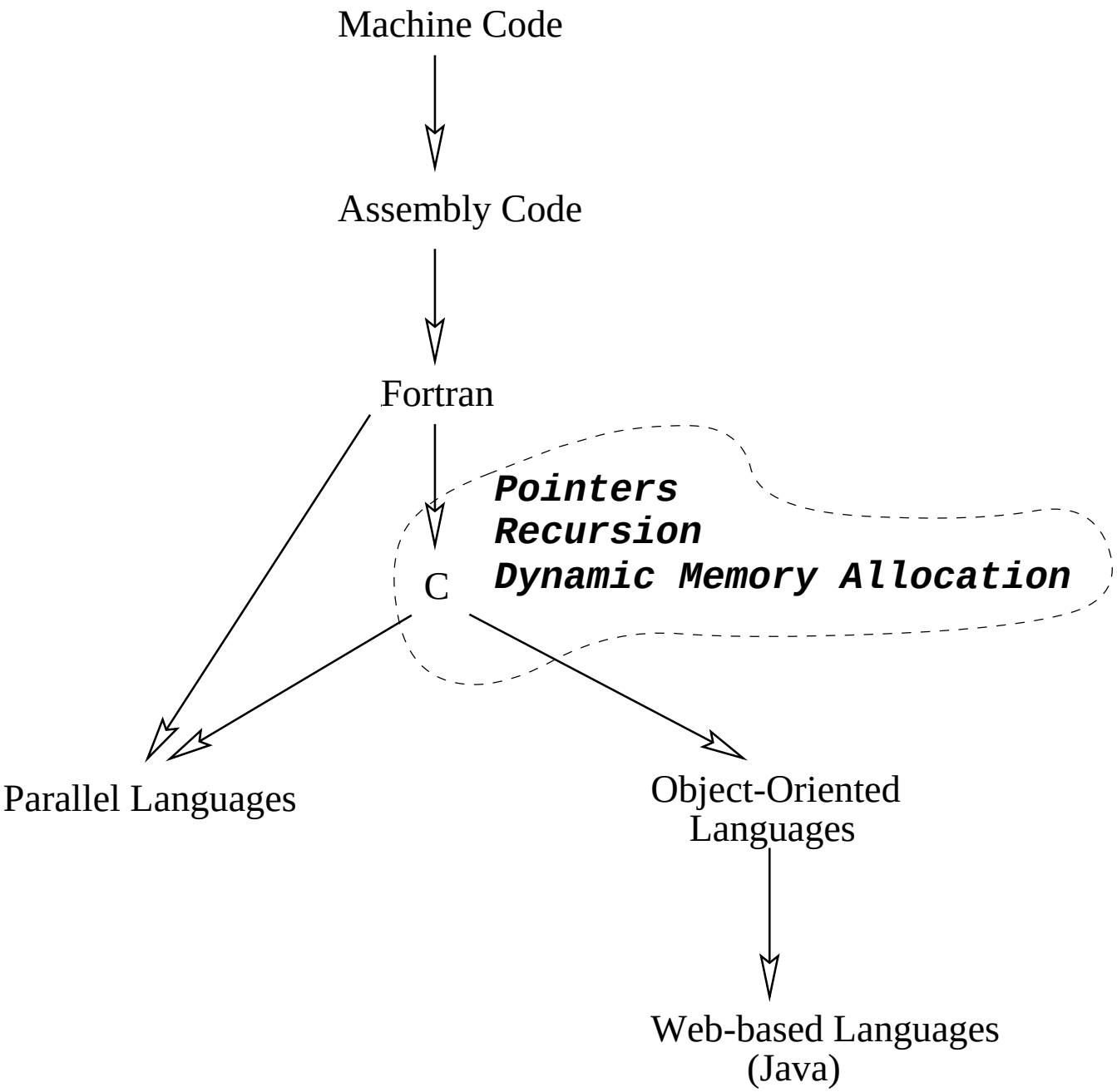
```
(1)  r1 <-- a
(5)  r3 <-- c
(2)  r2 <-- b
(6)  r4 <-- d
(3)  r2 <-- r1 + r2
(7)  r4 <-- r3 + r4
(4)  a <--  r2
(8)  c <-- r4
```

(e) Good RISC schedule

```
cycle 1:  (1) (2) (5) (6)
cycle 2:  (3) (7)
cycle 3:  (4) (8)
```

(f) A parallel schedule

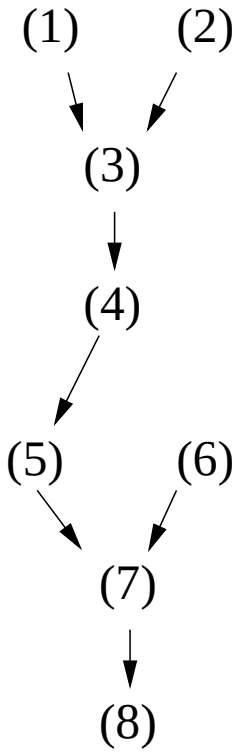
Problems due to Increased Programming Language Abstraction



Scheduling example: `a` and `c` are pointer variables

```
a->val =
   a->val+b;

c->val=
   c->val+d;
```

```
(1) r1 <-- 0[a]
(2) r2 <-- b
(3) r2 <-- r1+r2
(4) 0[a] <--  r2
(5) r3 <-- 0[c]
(6) r4 <-- d
(7) r4 <-- r3+r4
(8) 0[c] <-- r4
```



(a) C Program (b) Instructions (c) Dep. Graph

```
(1)  r1 <-- 0[a]
(2)  r2 <-- b
(6)  r4 <-- d
(3)  r2 <-- r1 + r2
     nop
(4)  0[a] <--  r2
(5)  r3 <-- 0[c]
     nop
(7)  r4 <-- r3 + r4
     nop
(8)  0[c] <-- r4
```

```
cycle 1:  (1) (2) (6)
cycle 2:  (3)
cycle 3:  (4)
cycle 4:  (5)
cycle 5:  (7)
cycle 6:  (8)
```

(d) A "good" RISC schedule (e) A parallel schedule

Pointer Analysis

- The big question is: *Is there a dependence between* `p->f` *at statement* `S` *and* `q->f` *at statement* `T`*?*

```
S:   p->f =
     ...
T:        = q->f
```

- Pointer analysis is a static analysis that can be used to answer these sorts of questions.

- Points-to Analysis for stack-allocated data: (Emami,Ghiya and Hendren, PLDI94)

```
     NODE x,y,*p,*q;
     p = &x;        {p → x}
     q = &y;        {p → x, q → y}
S:   p->f =
     ...
T:        = q->f
```

Pointer analysis of heap-allocated data is harder

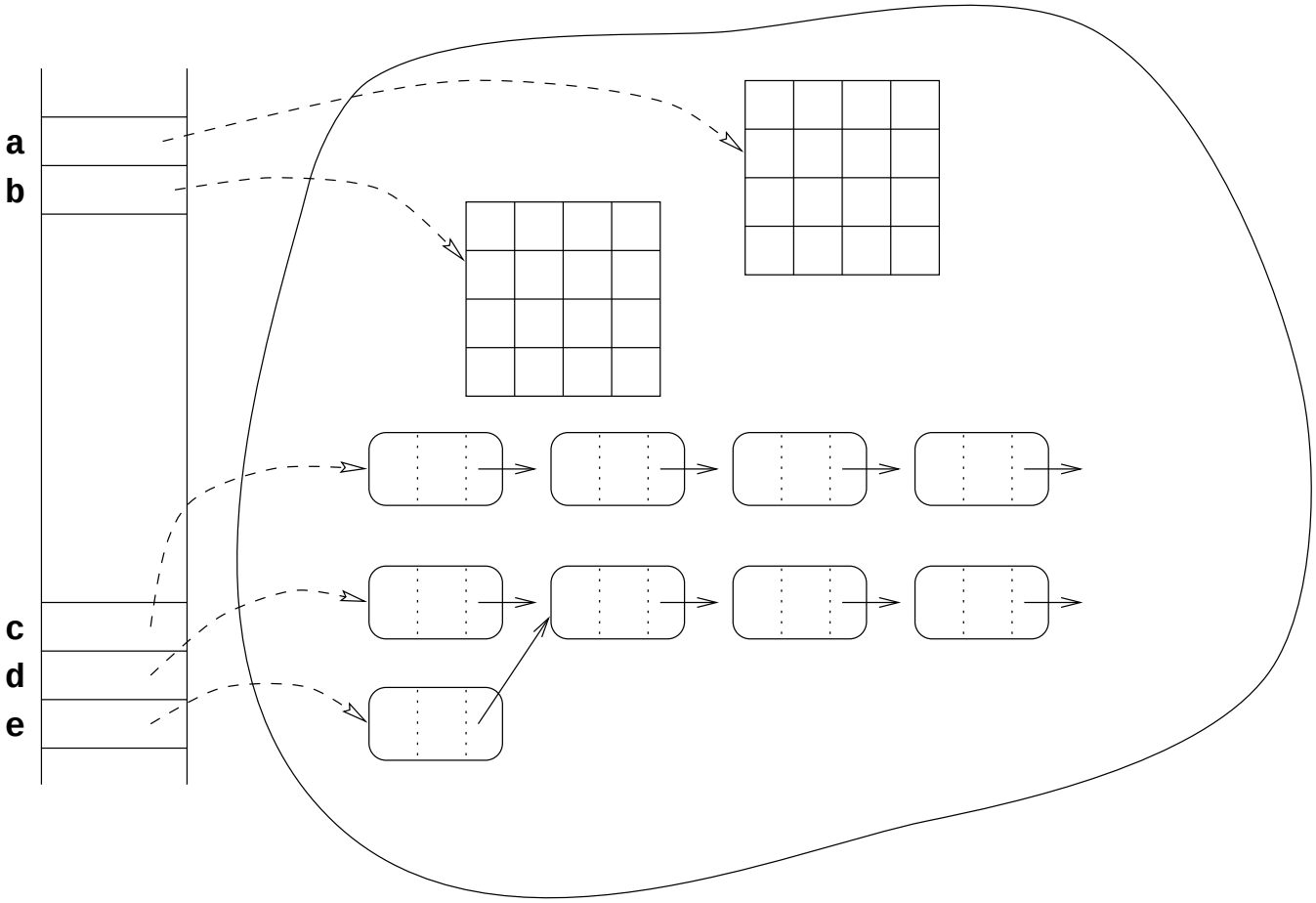
- Dynamically-allocated memory does not have a static name.

- Static analyses need to capture some "property" that is useful for dependence testing.

- *Connection Analysis* is used to estimate when two heap-directed pointers point to the same data structure.

- *Shape Analysis* is used to determine when the data structure pointed-to is a TREE, DAG or CYCLIC GRAPH.

Example of Connection Analysis (Ghiya and Hendren, IJPP96)

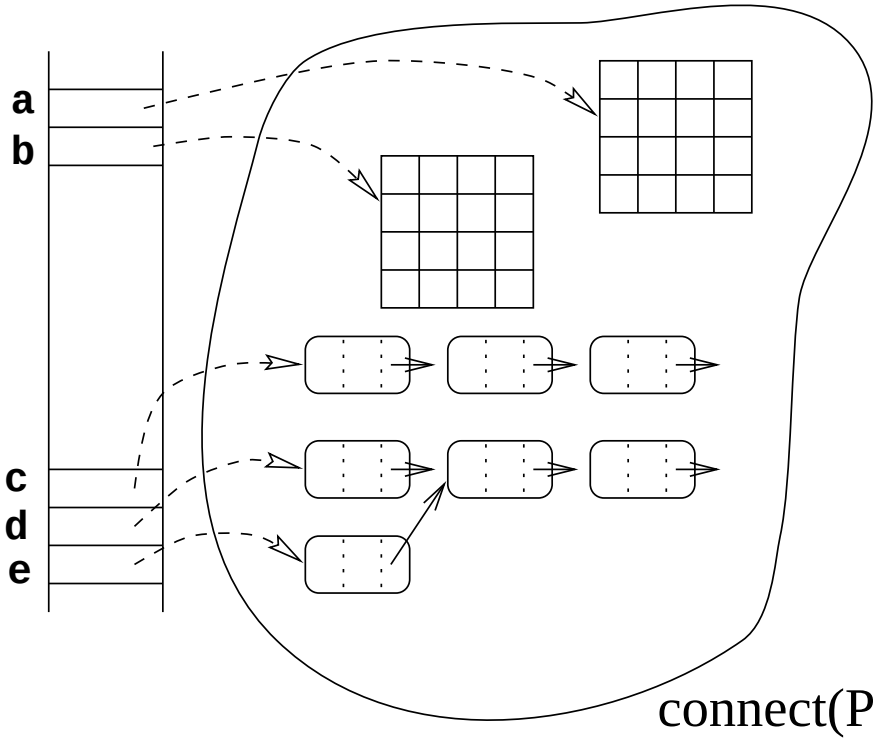


Stack, showing heap-directied pointers

Heap

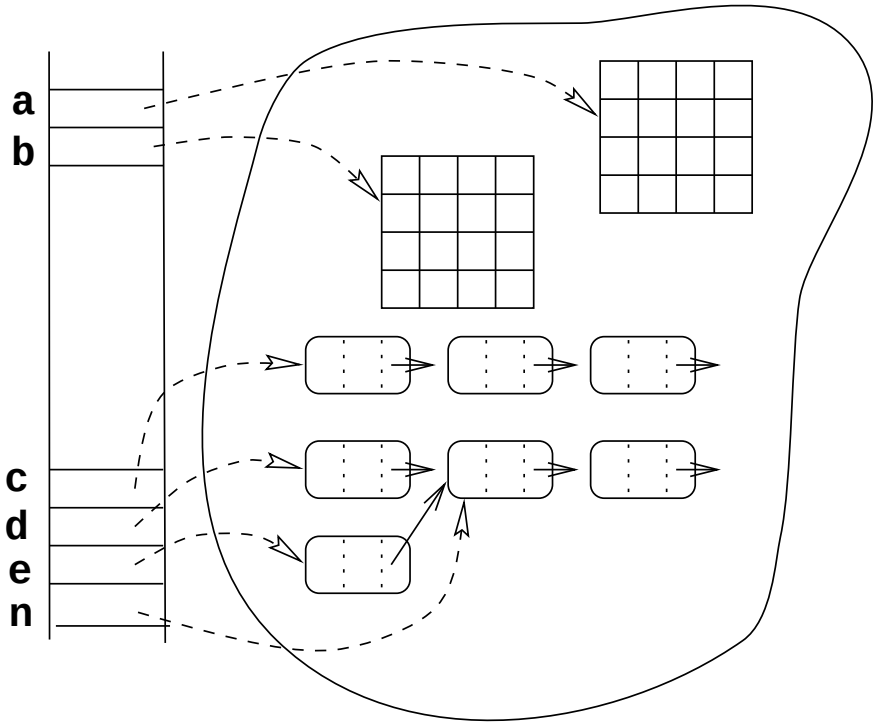
Not connected: (a,b), (c,d), (c,e)

Connected: (d,e)



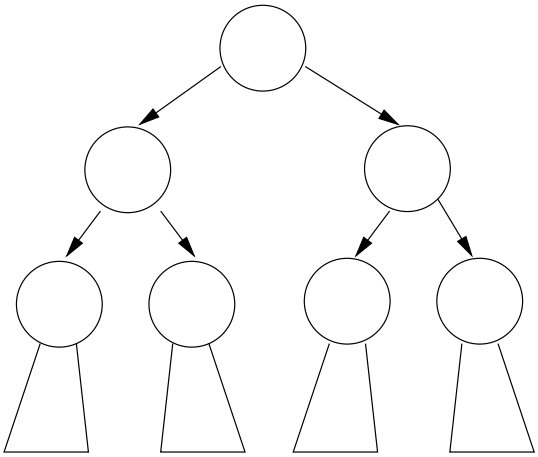
connect(P) = *{ (d,e) }*

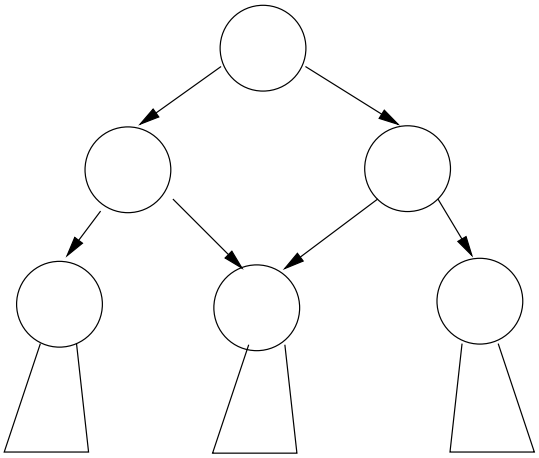
`n = d->next`

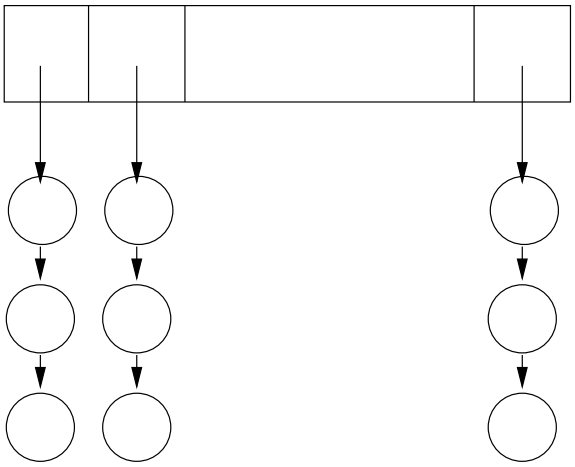


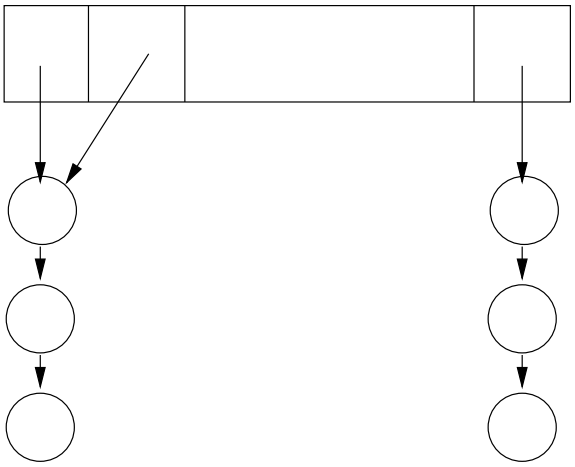
connect(P') = *{ (d,e), (d,n), (e,n) }*

Shape Analysis
(Ghiya and Hendren, POPL96)



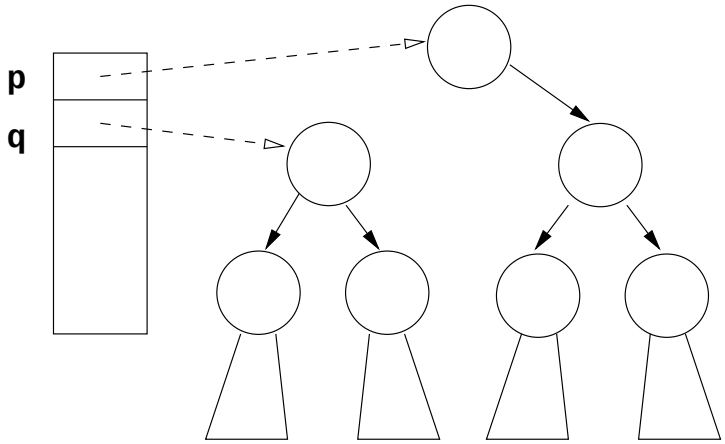




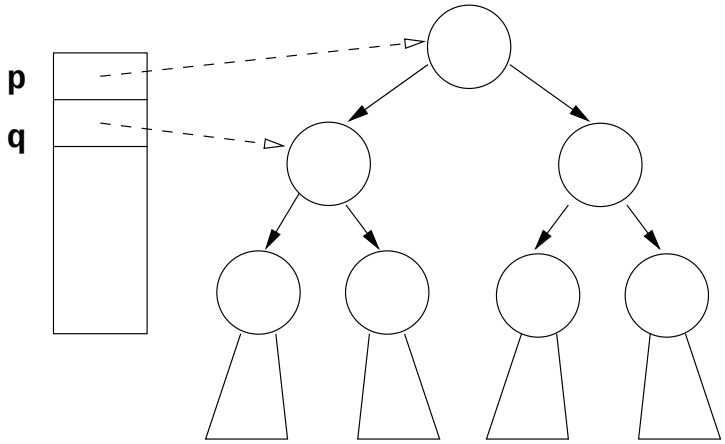


TREE

DAG




```
p->left = q;
```



Pointer `p` points to a TREE after the assignment if: `p` and `q` are TREE and `p` is not connected to `q` before the assignment.

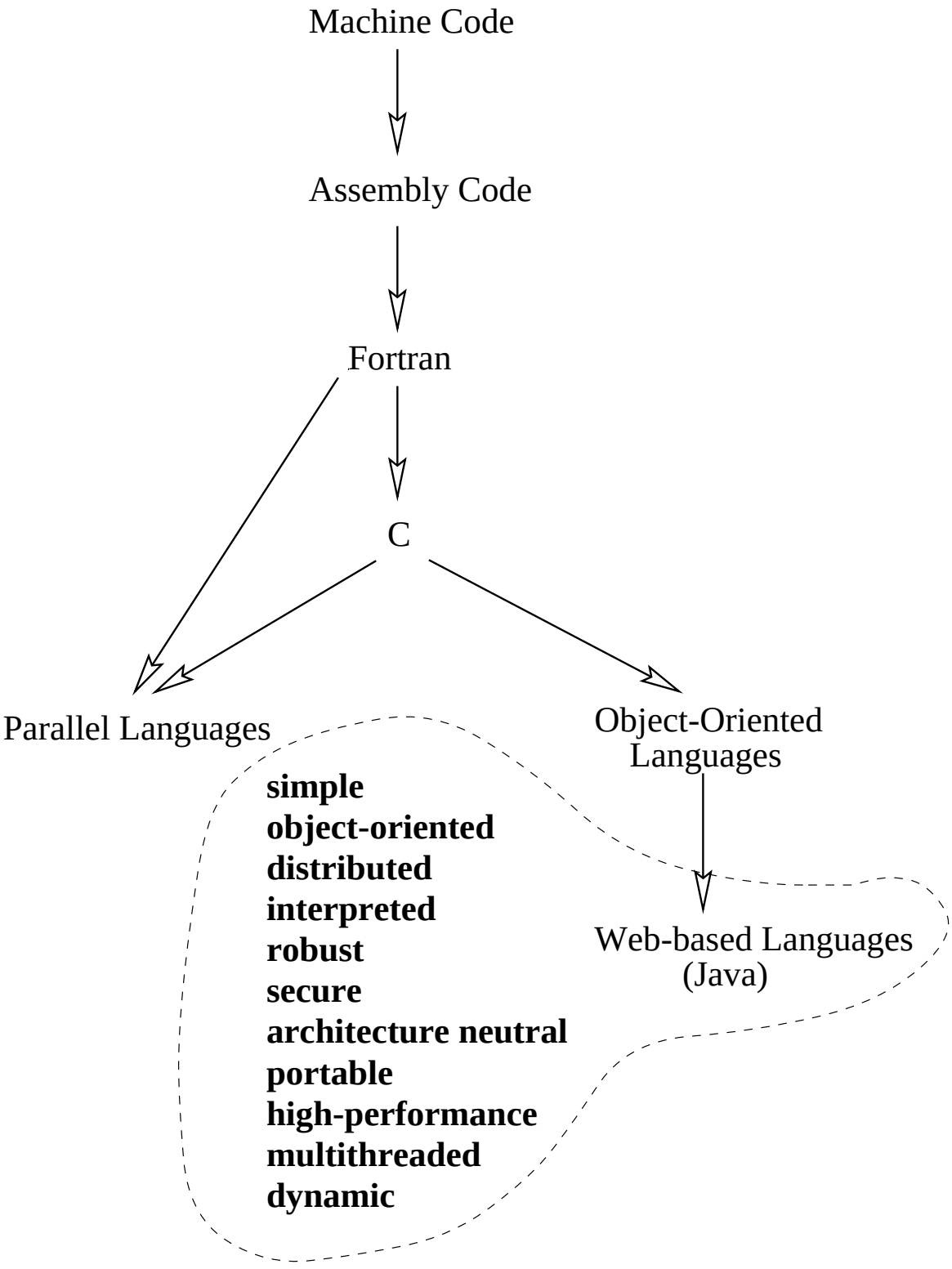
Observations

- Optimizing C compilers do a very good job on programs with well-structured for loops, using arrays in a regular manner.

- Programs with many small function calls and heavy use of dynamic data structures are harder to optimize. Need new analyses including pointer analysis for dynamically-allocated data structures.

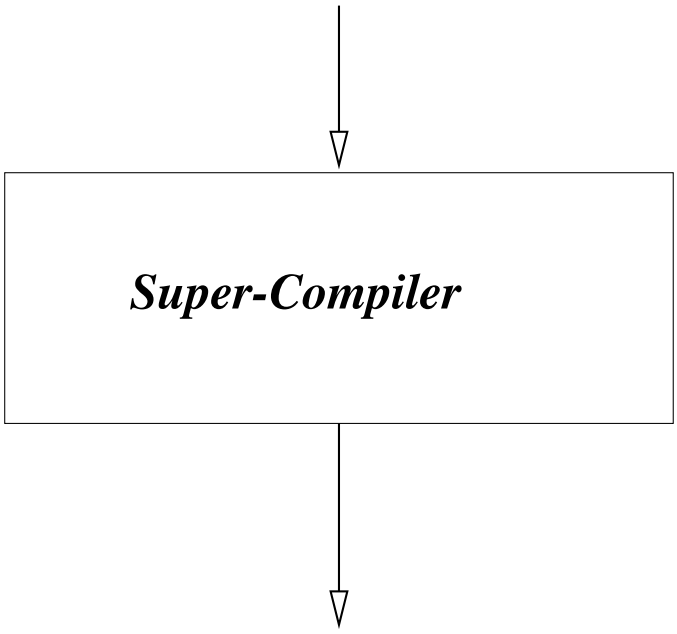
- In general, we need to add new optimizations to compilers as languages and as architectures evolve.

Problems with Abstractions in Java



Remember ... language, compiler, and architecture design interact

Program written in a very high-level, modern, (parallel) programming language



Efficient assembler/machine code for a modern, high-performance, (parallel) architecture

"Good for the programmer"

...

but, don't make the compiler's job impossible

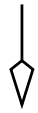
"Good for the computer"

...

but, also remember to provide support for optimizations that cannot be done at compile-time

Compiling for Parallel Machines

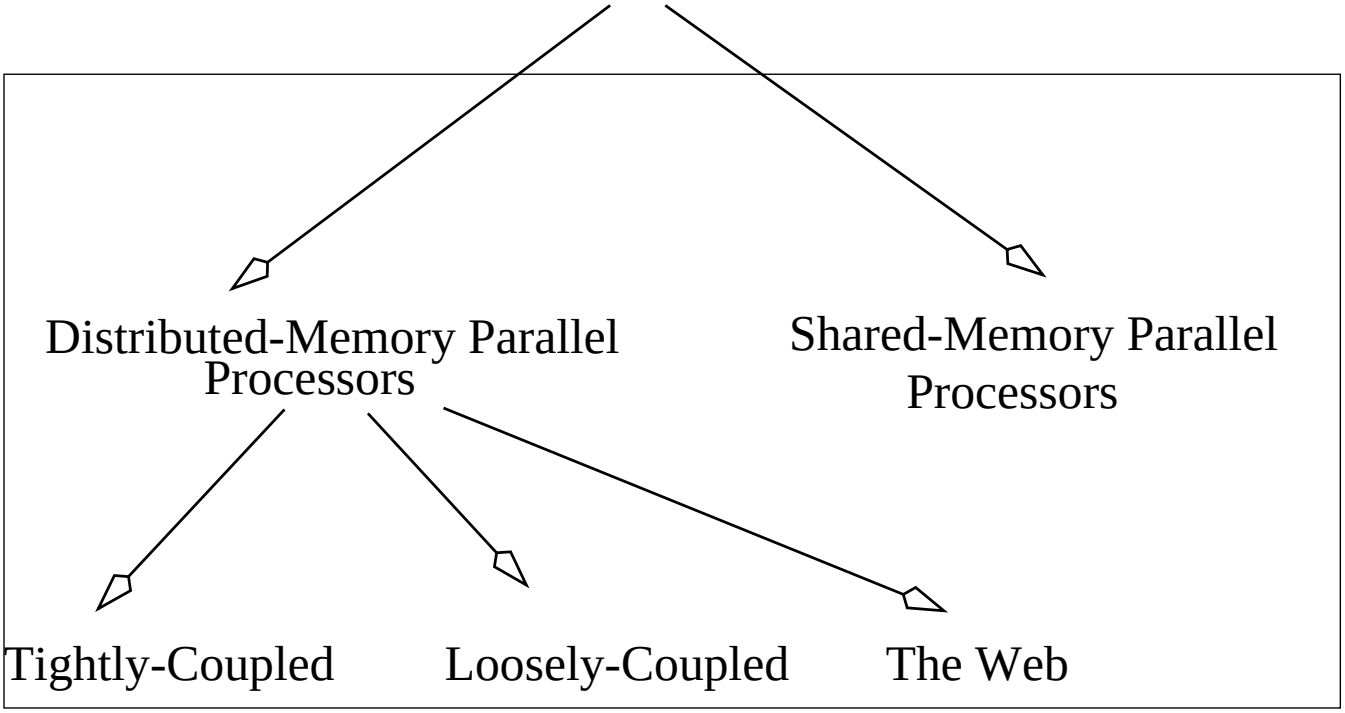
Simple "von Neumann" Architectures
(small register sets, small main memory)



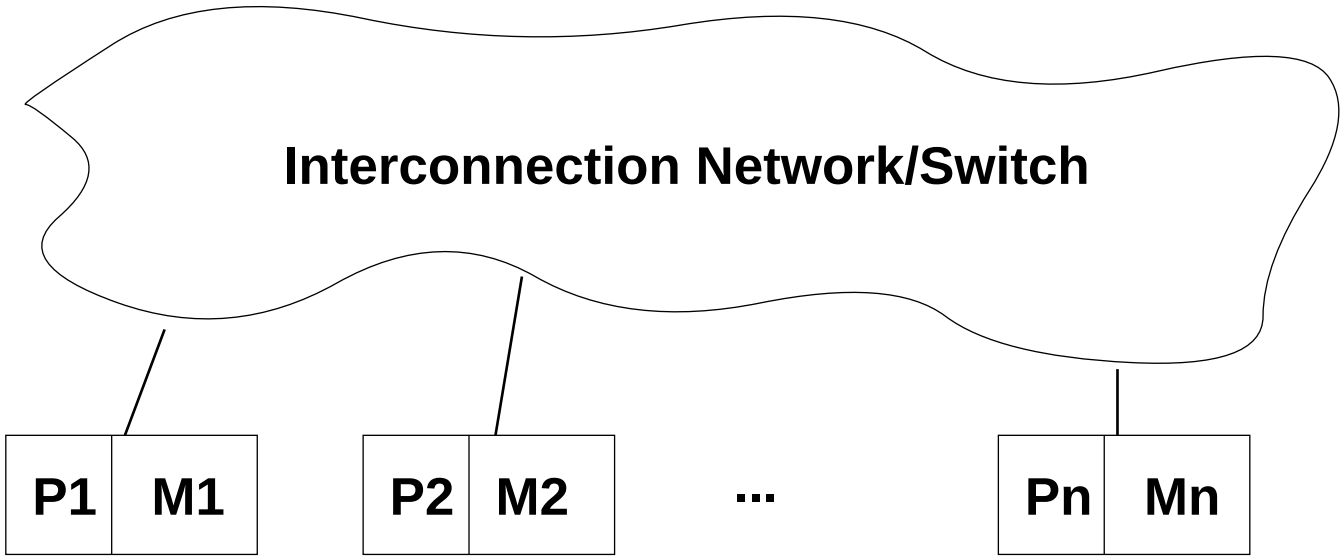
"Simple" RISC/CISC
(larger register sets, more main memory)

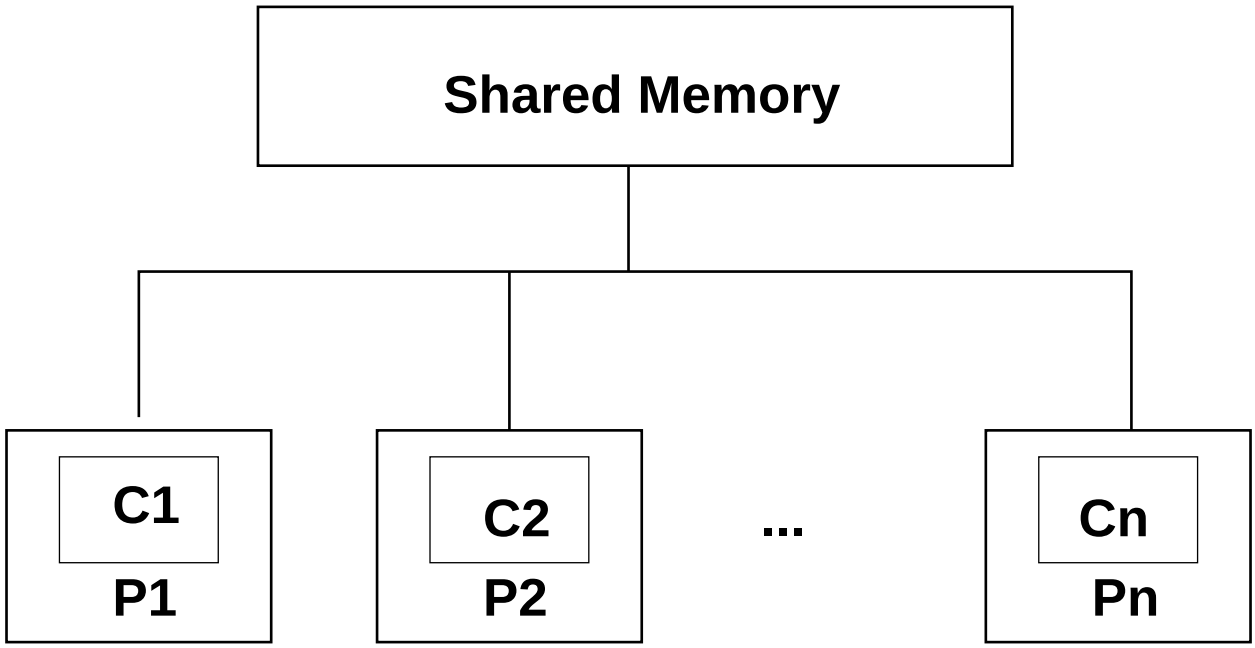
"Advanced" RISC/CISC

(register windows, pipelined, superscalar,
multi-level caches,enormous main memory)



Distributed vs. Shared Memory





An efficient Parallel Program must ...

- have sufficient parallelism
- not too much parallelism (computations must be coarse-grain)
- balanced parallelism
- relatively few remote memory accesses (data and computation should reside on the same processor)

Not all sequential programs can be parallelized!

The Big Question ...

- Can we design compilers to automatically convert sequential C programs to efficient parallel programs for distributed machines?

- In general, no!

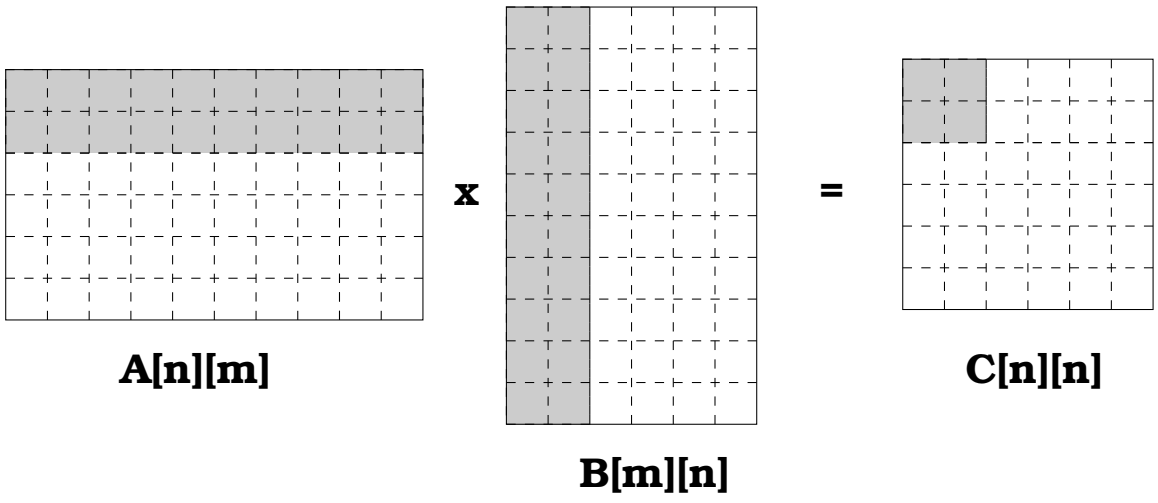
- But, we can minimize programmer effort, by careful design of the language, compiler, and run-time system.

Scientific Code - SPMD compilers

(High Performance Fortran, Fortran D (Rice))

- The programmer gives the sequential program + **static** data decompositions.

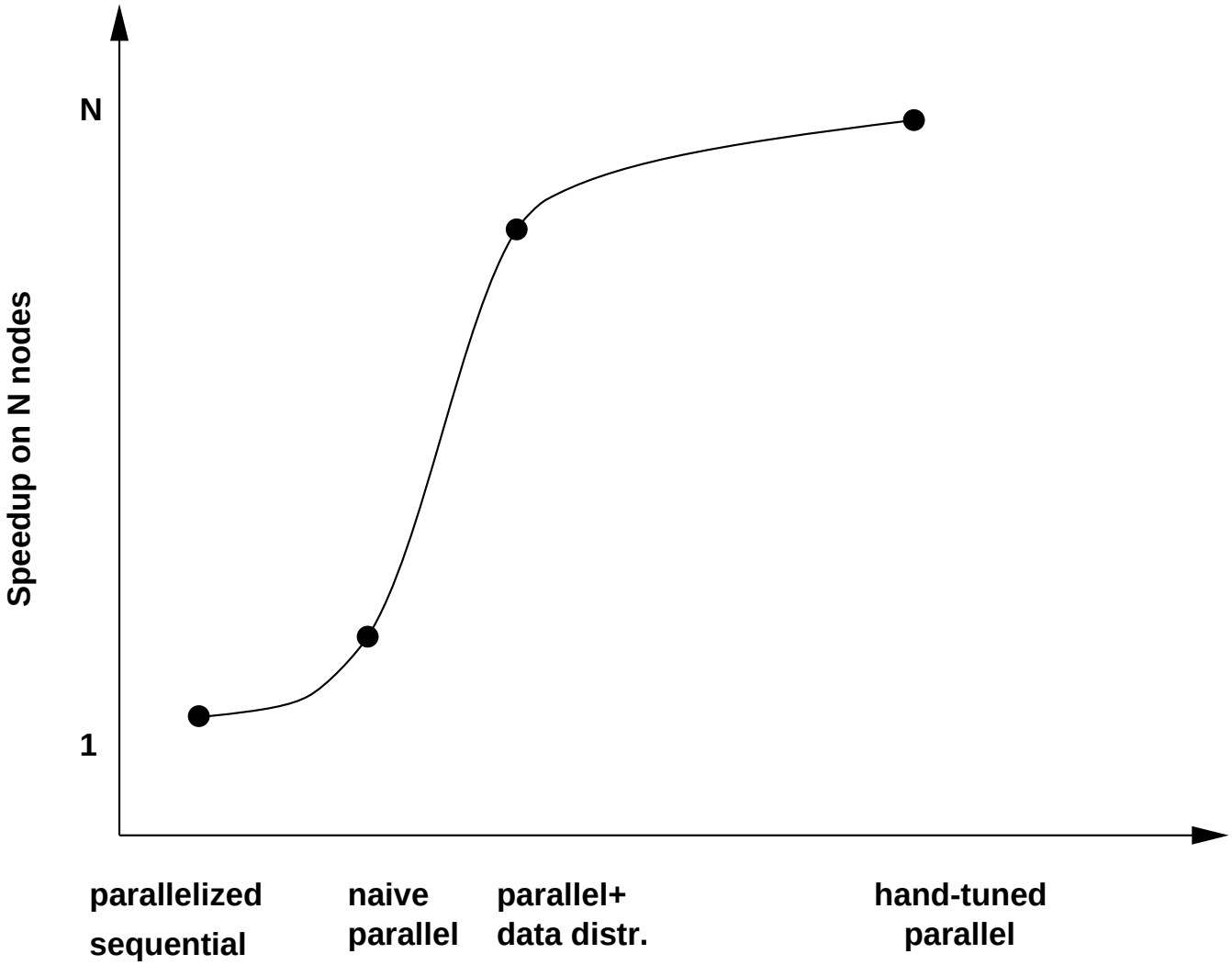
- The compiler **statically** maps the program to each processor, specializing it to the computations on data that it owns, and optimizing communication.



Regular Numerical Computations vs. Irregular Symbolic Computations

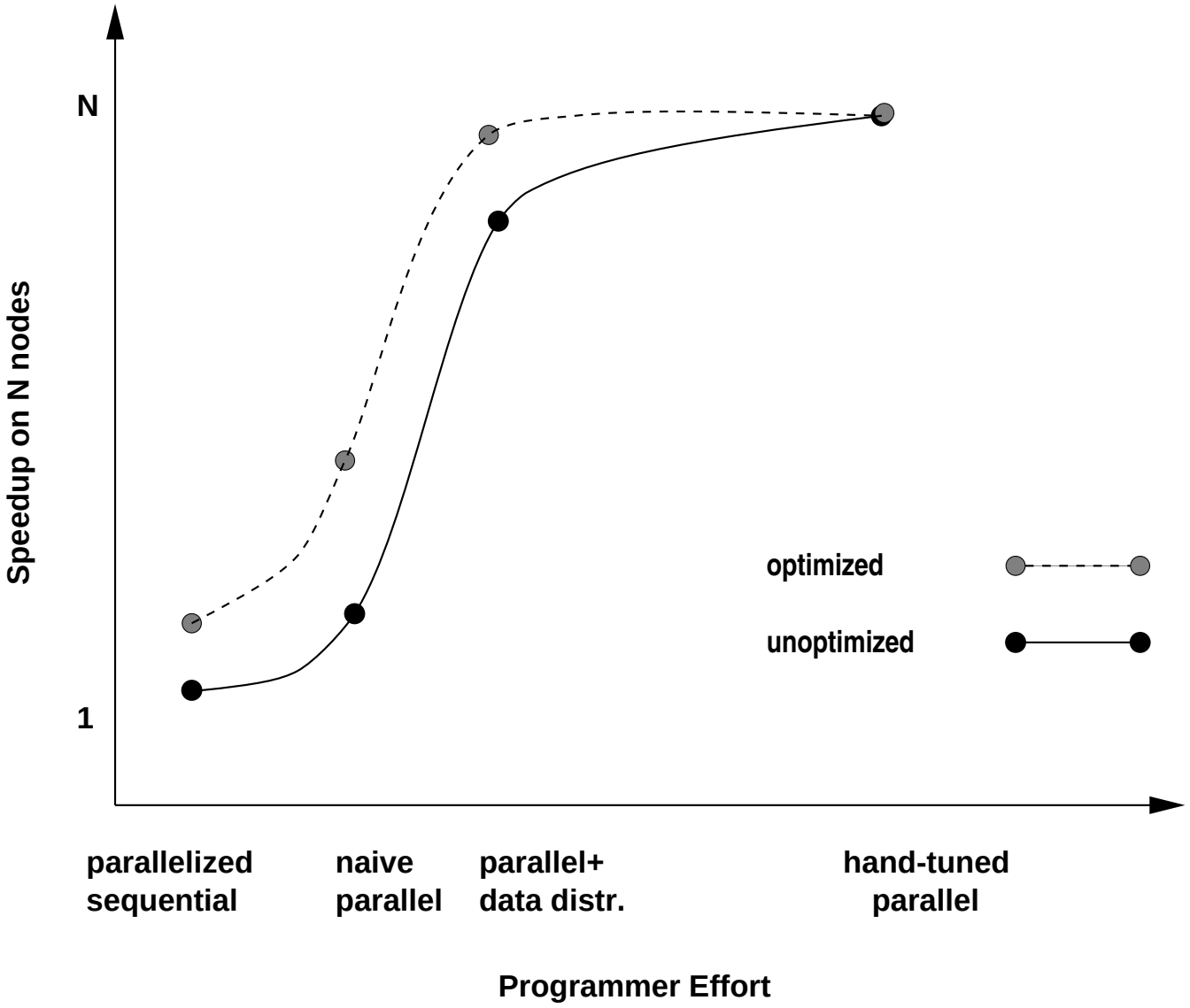
- Data is dynamic (dynamically allocated arrays, lists, trees)
- Control flow is more irregular
- Control flow is often via (recursive) function calls.
- Static data and computation mapping does not work for symbolic applications.
- Need good language + runtime system + compiler.

Programmer Effort vs. Speedup



- Programmer needs to make some contribution for good performance.
- Diminishing returns on programmer effort.

Effect of Compiler Optimizations



- Optimizations make larger proportional improvement on bad programs.
- Optimizations can provide the extra performance instead of hand-tuning.

Compiling C programs for a Multithreaded Parallel Machine

EARTH-C (McGill, Hendren et. al., IJPP97)

- Provide **minimal** language extensions that allow the programmer to express parallelism and data locality.
- Any C program is a valid EARTH-C program, and the compiler will attempt to parallelize.
- Language extensions should not hamper ordinary compiler analyses (pointer analysis).
- Compile the high-level program into a lower-level multithreaded program applying compiler optimizations.
- Execute the low-level program with an lean-and-mean run-time system that performs efficient communication, context-switches, and load balancing.

Summary of EARTH C Extensions

Declarations

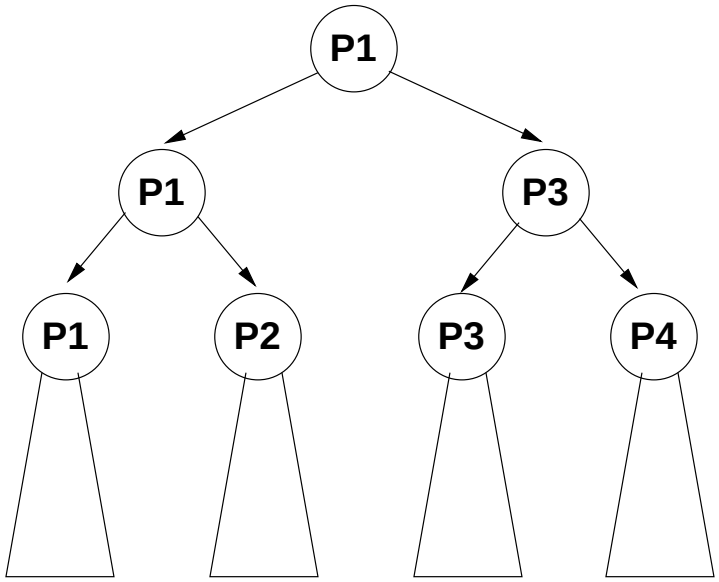
```
replicated int x[10];
int a[100]@N;
int local * p;
basic foo ( ... ) { ... }
shared int p;
atomic bar( ... ) { ... }
```

Control Constructs

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

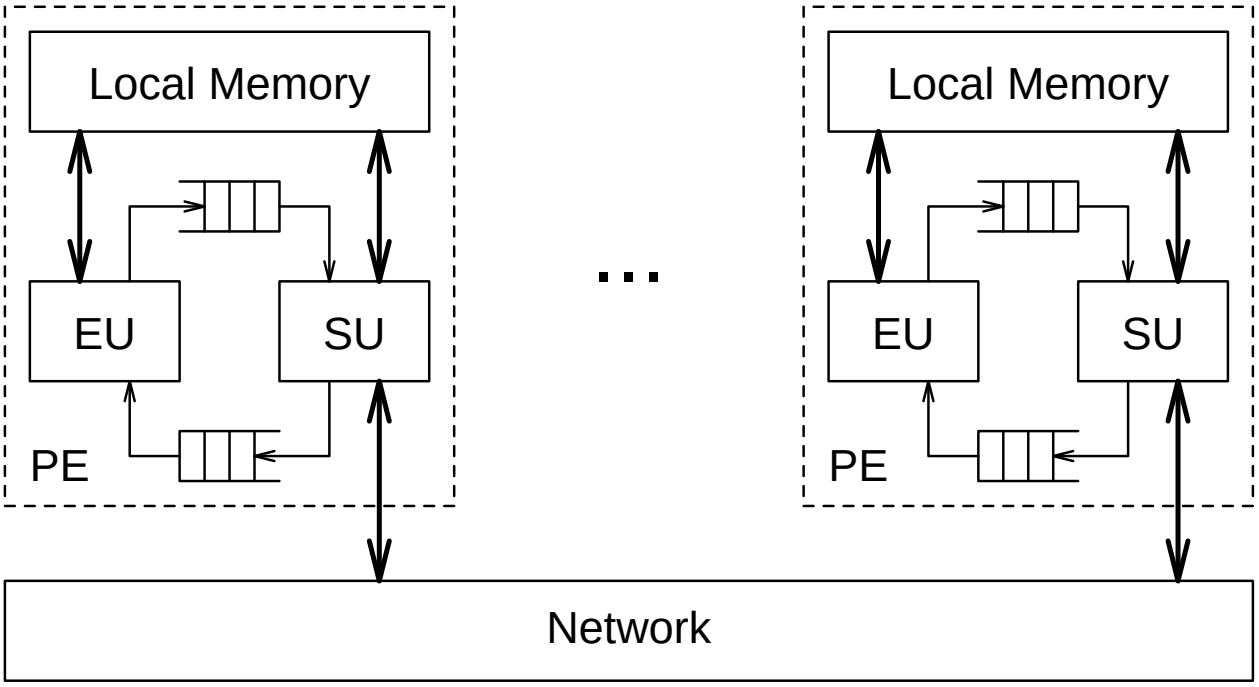
"Built-in" Const/Variables/Functions

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




```
int f(NODE local *root) {
int a,b,c;

if (root != NULL)
  { {^ a = g(root->value)@HOME;
       b = f(root->left)@OWNER_OF(root->left);
       c = f(root->right)@OWNER_OF(root->right);
    ^}
    return(a+b+c);
  }
return(0);
}
```

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)

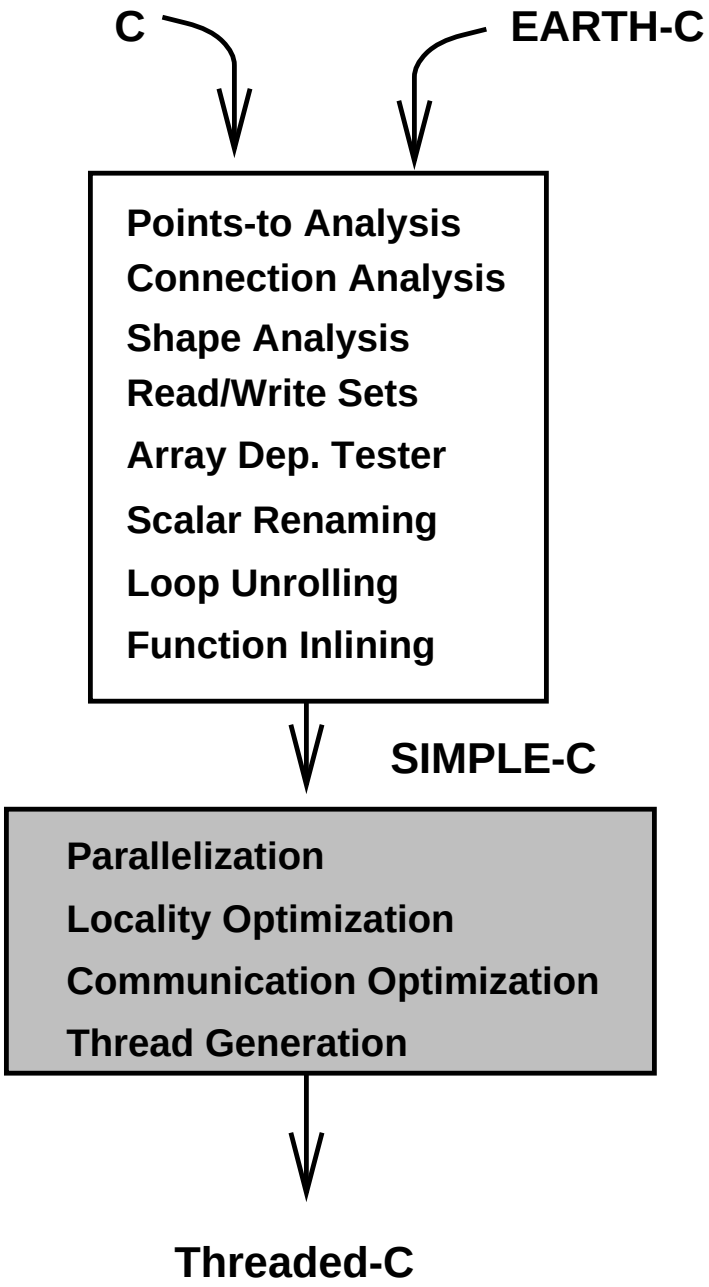
- Multithreaded emulator build on top of the GMD Manna parallel machine.

- 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.

McCAT/EARTH (Overview)



Parallelism Detection (Ghiya, Zhu and Hendren, CC98)

- Function-level Parallelism

```
void treeAdd(tree *t)
 {
   if (t == NULL)
    return;
Q: tl = t->left;
L: treeAdd(tl);
M: tr = t->right;
N: treeAdd(tr);
   t->val = tl->val + tr->val;
 }
```

- Forall Parallelism

```
for (i = 0; i < N; i++)
{ t = list_arr[i];
  foo(t, x, y);
}
```

- Foreach Parallelism

```
while (p != NULL)
{ S: lp->x = lp->y * 5;
  T: lp->y = lp->x * 6;
  U: lp = lp->next;
}
```

Locality Analysis (Zhu and Hendren, PACT97)

- Each indirect memory reference (`p->f`) must be assumed to be remote unless explicitly declared to be local.

- Remote memory references are MUCH slower than local memory references.

- The programmer can specify local pointers, but we want to automate this process as much as possible.

- Infer additional locality using static analysis.

Sources of Locality Information

Pointers to locally-scoped variables:

```
{ int x, *p;
  p = &x;
}
```

Dynamic allocation:

```
{ NODE *p;
  p = malloc(sizeof(NODE));
}
```

Mapping Computation using `OWNER_OF` and `HOME`:

```
void foo(NODE *r)
{  ...
   bar(r->f)@OWNER_OF(r->f);
   ...
}
```

```
void bar(NODE *p)
{  p->data = ...
}
```

Locality Inference

- Has two phases:
  - Intraprocedural inference (based on linear points-to);
  - Interprocedural propagation and specialization.
- Gives minimal (5%) to significant (80%) performance improvement.

Communication Optimization (Zhu and Hendren (PLDI98), Tang, Ghiya and Hendren(PACT97)

Fact: Each remote memory operation is costly.

Opt.: Redundant memory references should be stored in local scalars.

Fact: There is a delay between when a remote piece of data is requested and when it is available locally.

Opt.: Move requests for data as early as possible.

Fact: Moving a block of data is only marginally more expensive than moving a small piece of data.

Opt.: Block related data requests together.

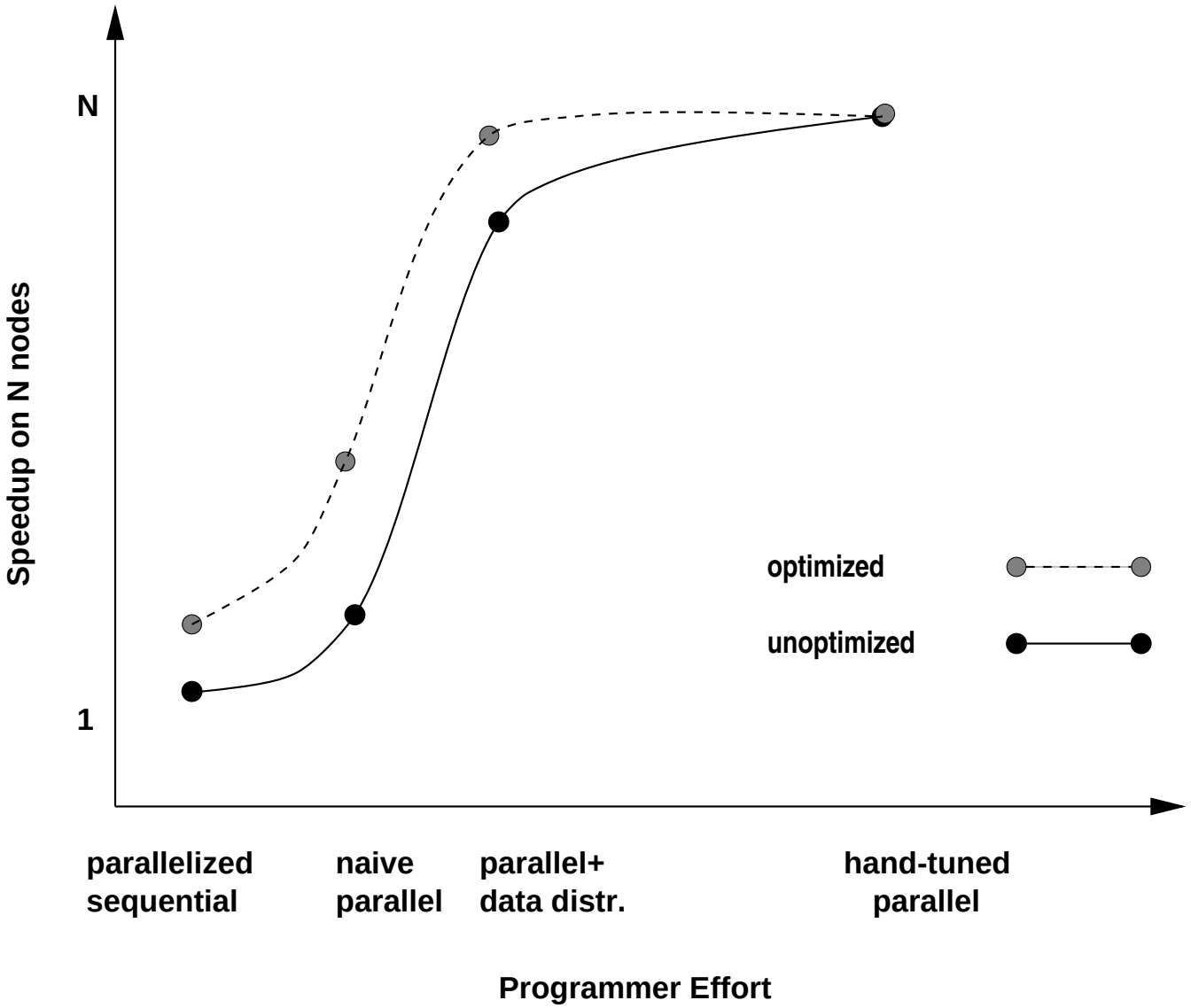
Example of Communication Analysis

```
int weird(NODE *p)
{ int a,b,c;


  foo(p);
  a = p->x;
  b = p->y;
  if (p != NULL)
    { c = bar(p->x);
      return(a+b+c);
    }
  return(0);
}
```

```
int weird(NODE *p)
{ int a,b,c;
  NODE loc_p;
  loc_p = blk_read(p);

  foo(p);
  a = loc_p.x;
  b = loc_p.y;
  if (p != NULL);
    { c = bar(loc_p.x);
      return(a+b+c);
    }
  return(0);
 }
```

- Gives from 2% to 15% performance improvement.
- Programs with very good locality won't benefit much.

Effect of Compiler Optimizations



- Optimizations make larger proportional improvement on bad programs.
- Optimizations can provide the extra performance instead of hand-tuning.
- EARTH-C optimizations approaching hand-tuned programs.

Conclusions

- Optimizing and Parallelizing compilers can do a very good job with scientific programs with simple, regular loops. **Don't** confuse the compiler by using fancy pointer operations.
- New language and architecture features require new compiler optimizations.
- For good performance, languages, compilers, run-time systems and architectures must be designed together.
- Totally automatic parallelization is not generally possible.
- Parallel languages and run-time systems must be designed along with the compiler.
- Minimize the impact on the programmer, but provide sufficient language support to express coarse-grain parallelism and locality.
- Make the compiler do the low-level optimizations.

Other Slides - not part of main presentation

Experimental Results

- benchmarks written exploiting EARTH C constructs
- compiled using EARTH/McCAT (EARTH C $\Rightarrow$ Threaded-C) using a simple thread generation strategy

	Description	Problem Size
TreeAdd	Recursive addition of a balanced binary tree	1M nodes
Power	Optimization Problems Based on variable K-nary tree	10,000 consumers (leaves)
TSP	Searching a sub-optimal tour for Travelling-salesman problem	32K cites
Voronoi	Compute the Voronoi Diagram of a set of points	64K points
N-queen	Searching all of legal queen configurations on a chessboard	12x12

	Seq. time(s)	Speedups				
		1	2	4	8	16
TreeAdd	0.81	0.94	1.88	3.77	7.51	15.01
Power	59.65	0.97	1.93	3.85	7.37	13.64
TSP	24.24	1.00	2.00	3.93	7.59	13.10
Voronoi	7.81	0.93	1.59	2.77	5.04	9.40
N-queen	52.95	1.00	1.99	3.94	7.91	15.84

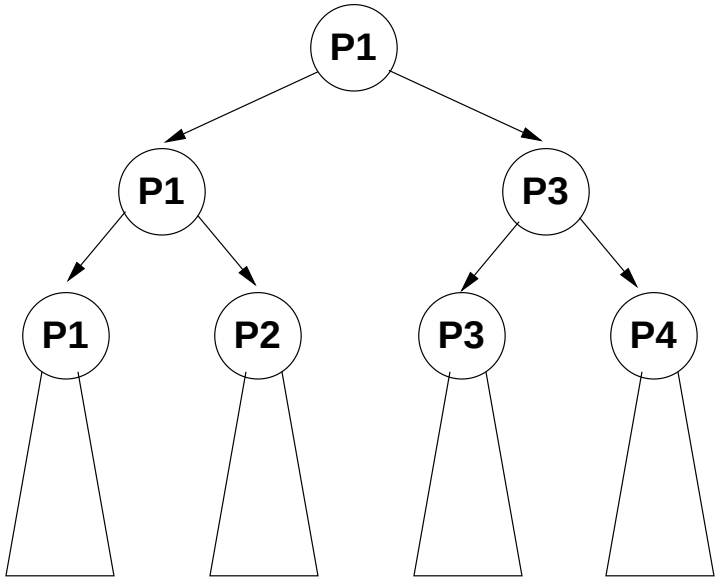
Symbolic Code - DSPMD compiler/runtime system

Olden (Princeton, Rogers, Carlisle, Reppy and Hendren, TOPLAS 95)

- The programmer **dynamically** allocates his/her data structures so that subcomputations will be on the same hunk.

- The programmer specifies function calls that can execute in parallel (using futures).

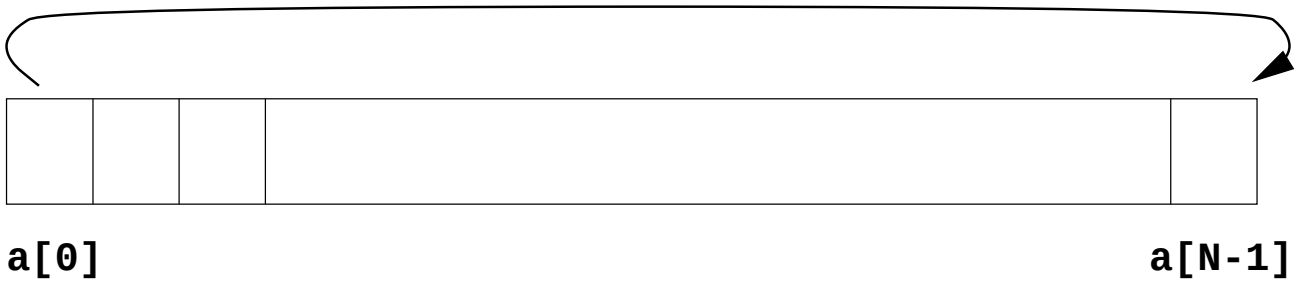
- The compiler and run-time system **dynamically** moves the computation to the owner of the data structure.



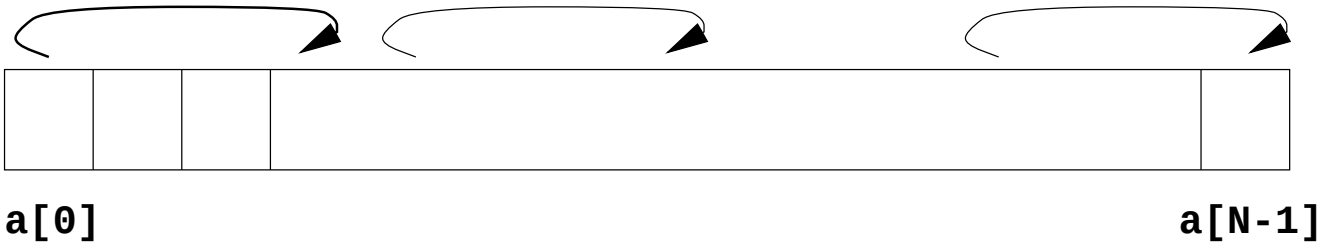

```
int f(NODE *root)
   { future_call a,b;
     int c;

     if (root != NULL)
      { a = future_call(g,root->value);
        b = future_call(f,root->left);
        c = f(root->right);
        return(touch(a)+touch(b)+c);
      }
     return(0);
   }
```

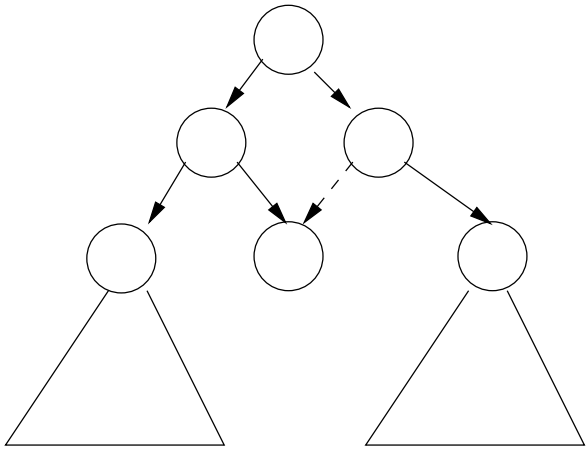
Optimizing Data Caches

```
for (n=0; n<REPS; n++)
  for (i=0; i<N; i++)
    a[i] = f(a[i],n);
```




```
for (b=0; b<N; b=b+BSIZE)
  for (n=0; n<REPS; n++)
    for (i=b; i<b+BSIZE; i++)
      a[i] = f(a[i],n);
```



Optimizing Data Caches with Dynamic Data Structures?

```
for (n=0; n<REPS; n++)
  traverse_tree(root,n);


traverse_tree(NODE *root, int n)
{  if (root != NULL)
     { root->key = f(root->key,n);
       traverse_tree(root->left,n);
       traverse_tree(root->right,n);
     }
}
```