Final Exam!

- Thursday May 3 in class
- Closed book, closed notes
Single-Threaded Performance Not Improving
What about Parallel Programming? –or-
What is Good About the Sequential Model?

❖ Sequential is easier
  » People think about programs sequentially
  » Simpler to write a sequential program

❖ Deterministic execution
  » Reproducing errors for debugging
  » Testing for correctness

❖ No concurrency bugs
  » Deadlock, livelock, atomicity violations
  » Locks are not composable

❖ Performance extraction
  » Sequential programs are portable
    Ÿ Are parallel programs? Ask GPU developers 😊
  » Performance debugging of sequential programs straight-forward
Compilers are the Answer? - Proebsting’s Law

- “Compilers are the Answer!”
- “Compiler Advances Double Computing Power Every 18 Years”

Run your favorite set of benchmarks with your favorite state-of-the-art optimizing compiler. Run the benchmarks both with and without optimizations enabled. The ratio of those numbers represents the entirety of the contribution of compiler optimizations to speeding up those benchmarks. Let’s assume that this ratio is about 4X for typical real-world applications, and let’s further assume that compiler optimization work has been going on for about 36 years. Therefore, compiler optimization advances double computing power every 18 years. QED.

Conclusion – Compilers not about performance!
Are We Doomed?

A Step Back in Time: Old Skool Parallelization
Parallelizing Loops In Scientific Applications

Scientific Codes (FORTRAN-like)

```
for(i=1; i<=N; i++) // C
  a[i] = a[i] + 1; // X
```

Independent Multithreading (IMT)

Example: DOALL parallelization
What Information is Needed to Parallelize?

❖ Dependences within iterations are fine
❖ Identify the presence of cross-iteration data-dependences
  » Traditional analysis is inadequate for parallelization. For instance, it does not distinguish between different executions of the same statement in a loop.
❖ Array dependence analysis enables optimization for parallelism in programs involving arrays.
  » Determine pairs of iterations where there is a data dependence
  » Want to know all dependences, not just yes/no

```
for(i=1; i<=N; i++) // C
a[i] = a[i] + 1;  // X
```
```
for(i=1; i<=N; i++) // C
a[i] = a[i-1] + 1;  // X
```
Affine/Linear Functions

- $f(i_1, i_2, ..., i_n)$ is **affine**, if it can be expressed as a sum of a constant, plus constant multiples of the variables. i.e.

$$f = c_0 + \sum_{i=1}^{n} c_i x_i$$

- Array subscript expressions are usually affine functions involving loop induction variables.

- Examples:
  - `a[i]` **affine**
  - `a[i+j-1]` **affine**
  - `a[i*j]` non-linear, not affine
  - `a[2*i+1, i*j]` linear/non-linear, not affine
  - `a[b[i] + 1]` non linear (indexed subscript), not affine
for (i = 1; i < 10; i++) {
    X[i] = X[i-1]
}

To find all the data dependences, we check if

1. $X[i-1]$ and $X[i]$ refer to the same location;
2. different instances of $X[i]$ refer to the same location.

For 1, we solve for $i$ and $i'$ in

$1 \leq i \leq 10, 1 \leq i' \leq 10$ and $i - 1 = i'$

For 2, we solve for $i$ and $i'$ in

$1 \leq i \leq 10, 1 \leq i' \leq 10, i = i'$ and $i \neq i'$ (between different dynamic accesses)

There is a dependence since there exist integer solutions to 1. e.g. $(i=2, i'=1), (i=3,i'=2)$. 9 solutions exist.

There is no dependences among different instances of $X[i]$ because 2 has no solutions!
Array Dependence Analysis - Summary

- Array data dependence basically requires finding integer solutions to a system (often referred to as dependence system) consisting of equalities and inequalities.
- Equalities are derived from array accesses.
- Inequalities from the loop bounds.
- It is an integer linear programming problem.
- ILP is an NP-Complete problem.
- Several Heuristics have been developed.
  » Omega – U. Maryland
Loop Parallelization Using Affine Analysis Is Proven Technology

❖ DOALL Loop
   » No loop carried dependences for a particular nest
   » Loop interchange to move parallel loops to outer scopes

❖ Other forms of parallelization possible
   » DOAcross, DOpipe

❖ Optimizing for the memory hierarchy
   » Tiling, skewing, etc.

❖ Real compilers available – KAP, Portland Group, gcc

❖ For better information, see
Back to the Present – Parallelizing C and C++ Programs
Loop Level Parallelization

**Bad news:** limited number of parallel loops in general purpose applications

– 1.3x speedup for SpecINT2000 on 4 cores
DOALL Loop Coverage

![Graph showing DOALL Loop Coverage for different benchmarks and categories. The x-axis represents benchmarks like SPEC FP, SPEC INT, Mediabench, and Utilities, while the y-axis represents the fraction of sequential execution. The graph uses bars to indicate coverage percentages for each benchmark.]
What’s the Problem?

1. Memory dependence analysis

```c
for (i=0; i<100; i++) {
    ... = *p;
    *q = ...;
}
```

Memory dependence profiling and speculative parallelization
DOALL Coverage – Provable and Profiled

Still not good enough!
What’s the Next Problem?

2. Data dependences

while (ptr != NULL) {

... 

ptr = ptr->next;

sum = sum + foo;
}

Compiler transformations
We Know How to Break Some of These Dependences – Recall ILP Optimizations

Apply accumulator variable expansion!

sum += x

Thread 0

sum1 += x

Thread 1

sum2 += x

sum = sum1 + sum2

sum = sum1 + sum2
Data Dependences Inhibit Parallelization

- Accumulator, induction, and min/max expansion only capture a small set of dependences

- 2 options
  - 1) Break more dependences – New transformations
  - 2) Parallelize in the presence of dependences – more than DOALL parallelization

- We will talk about both, but for now ignore this issue
3. C/C++ too restrictive

```c
char *memory;

void * alloc(int size);

void * alloc(int size) {
    void * ptr = memory;
    memory = memory + size;
    return ptr;
}
```
char *memory;

void * alloc(int size);

void * alloc(int size) {
    void * ptr = memory;
    memory = memory + size;
    return ptr;
}

Loops cannot be parallelized even if computation is independent
Commutative Extension

- Interchangeable call sites
  - Programmer doesn’t care about the order that a particular function is called
  - Multiple different orders are all defined as correct
  - Impossible to express in C

- Prime example is memory allocation routine
  - Programmer does not care which address is returned on each call, just that the proper space is provided

- Enables compiler to break dependences that flow from 1 invocation to next forcing sequential behavior
char *memory;

@Commutative
void * alloc(int size);

void * alloc(int size) {
    void * ptr = memory;
    memory = memory + size;
    return ptr;
}
char *memory;

@Commutative
void * alloc(int size);

void * alloc(int size) {
    void * ptr = memory;
    memory = memory + size;
    return ptr;
}

Implementation dependences should not cause serialization.
What is the Next Problem?

- **4. C does not allow any prescribed non-determinism**
  - Thus sequential semantics must be assumed even though they not necessary
  - Restricts parallelism (useless dependences)

- **Non-deterministic branch → programmer does not care about individual outcomes**
  - They attach a probability to control how statistically often the branch should take
  - Allow compiler to tradeoff ‘quality’ (e.g., compression rates) for performance
    - When to create a new dictionary in a compression scheme
#define CUTOFF 100

dict = create_dict();
count = 0;
while((char = read(1))) {
    profitable = compress(char, dict)
    if (!profitable) {
        dict = restart(dict);
    }
}
finish_dict(dict);

profitable = compress(char, dict)
if (!profitable) {
    dict = restart(dict);
}
if (count == CUTOFF) {
    dict = restart(dict);
    count = 0;
}

count++;

finish_dict(dict);
dict = create_dict();
while((char = read(1))) {
    profitable =
        compress(char, dict)

    @YBRANCH(probability=.01)
    if (!profitable) {
        dict = restart(dict);
    }
}
finish_dict(dict);

Compilers are best situated to make
the tradeoff between output quality
and performance
Capturing Output/Performance Tradeoff: *Y*-Branches in 164.gzip

dict = create_dict();
while((char = read(1))) {
    profitable =
        compress(char, dict)

    if(!profitable) {
        dict = restart(dict);
    }
} } 
finish_dict(dict);

#define CUTOFF 100000

dict = create_dict();

count = 0;
while((char = read(1))) {
    profitable =
        compress(char, dict)

    if(!profitable) {
        dict = restart(dict);
    }
    if(count == CUTOFF){
        dict = restart(dict);
        count = 0;
    }
    count++;
} }

finish_dict(dict);
**256.bzip2**

```c
unsigned char *block;
int last_written;

compressStream(in, out) {
    while (True) {
        loadAndRLEsource(in);
        if (!last) break;
        doReversibleTransform();
        sendMTFValues(out);
    }
}

doReversibleTransform() {
    sortIt();
    ...
}

sortIt() {
    printf(...);
    ...
}
```

Parallelization techniques must look inside function calls to expose operations that cause synchronization.
197.parser

batch_process() {
    while (True) {
        sentence = read();
        if (!sentence) break;
        parse(sentence);
        print(sentence);
    }
}

High-Level View:
Parsing a sentence is independent of any other sentence.

Low-Level Reality:
Implementation dependences inside functions called by parse lead to large sequential regions.

char *memory;

void *xalloc(int size) {
    void *ptr = memory;
    memory = memory + size;
    return ptr;
}
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>26</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>175.vpr</td>
<td>1</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>X</td>
</tr>
<tr>
<td>176.gcc</td>
<td>18</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>181.mcf</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>186.crafty</td>
<td>9</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>197.parser</td>
<td>3</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>254.gap</td>
<td>3</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>255.vortex</td>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>300.twolf</td>
<td>1</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Modified only 60 LOC out of ~500,000 LOC**
What prevents the automatic extraction of parallelism?

Lack of an Aggressive Compilation Framework

Sequential Programming Model
What About Non-Scientific Codes???

Scientific Codes (FORTRAN-like)

```c
for(i=1; i<=N; i++) // C
    a[i] = a[i] + 1; // X
```

General-purpose Codes (legacy C/C++)

```c
while(ptr = ptr->next) // LD
    ptr->val = ptr->val + 1; // X
```

Independent Multithreading (IMT)

Example: DOALL parallelization

Cyclic Multithreading (CMT)

Example: DOACROSS [Cytron, ICPP 86]
Alternative Parallelization Approaches

while(ptr = ptr->next)    // LD
    ptr->val = ptr->val + 1; // X

Cyclic Multithreading (CMT)

Pipelined Multithreading (PMT)

Example: DSWP [PACT 2004]
Comparison: IMT, PMT, CMT

**IMT**

Core 1
- C:1
- X:1
- C:3
- X:3
- C:5
- X:5

Core 2
- C:2
- X:2
- C:4
- X:4
- C:6
- X:6

**PMT**

Core 1
- LD:1
- X:1
- LD:3
- X:2
- LD:4
- X:3
- LD:5
- X:4
- LD:6
- X:5

Core 2
- LD:2
- X:1
- LD:3
- X:2
- LD:4
- X:3
- LD:5
- X:4
- LD:6
- X:5

**CMT**

Core 1
- LD:1
- X:1
- LD:3
- X:2
- LD:4
- X:3
- LD:5
- X:4
- LD:6
- X:5

Core 2
- LD:2
- X:1
- LD:3
- X:2
- LD:4
- X:3
- LD:5
- X:4
- LD:6
- X:5
Comparison: IMT, PMT, CMT

For IMT:
- Core 1: X:1, C:1, C:3, X:3, C:5, X:5
- Core 2: X:2, C:2, C:4, X:4, C:6, X:6
- Latency (comm) = 1: 1 iter/cycle

For PMT:
- Core 1: LD:1, LD:2, LD:3, LD:4, LD:5, LD:6
- Core 2: X:1, X:2, X:3, X:4, X:5
- Latency (comm) = 1: 1 iter/cycle

For CMT:
- Core 1: LD:1, LD:3, LD:5
- Core 2: X:1, X:2, X:3, X:4, X:5, LD:6
- Latency (comm) = 1: 1 iter/cycle
Comparison: IMT, PMT, CMT

**IMT**

Core 1
- C:1 --> X:1
- C:3 --> X:3
- C:5 --> X:5

Core 2
- C:2 --> X:2
- C:4 --> X:4
- C:6 --> X:6

lat(comm) = 1: 1 iter/cycle
lat(comm) = 2: 1 iter/cycle

**PMT**

Core 1
- LD:1 --> X:1
- LD:3 --> X:3
- LD:5 --> X:5

Core 2
- LD:2 --> X:2
- LD:4 --> X:4
- LD:6 --> X:6

lat(comm) = 1: 1 iter/cycle
lat(comm) = 2: 1 iter/cycle

**CMT**

Core 1
- LD:1 --> X:1
- LD:3 --> X:3

Core 2
- LD:2 --> X:2
- LD:6 --> X:6

lat(comm) = 1: 1 iter/cycle
lat(comm) = 2: 0.5 iter/cycle
Comparison: IMT, PMT, CMT

Thread-local Recurrences ➔ Fast Execution

Cross-thread Dependences ➔ Wide Applicability
Our Objective: Automatic Extraction of Pipeline Parallelism using DSWP

Decoupled Software Pipelining

Find English Sentences → Parse Sentences (95%) → Emit Results

PS-DSWP (Spec DOALL Middle Stage)
Decoupled Software Pipelining
Decoupled Software Pipelining (DSWP)

A: while(node)
B: ncost = doit(node);
C: cost += ncost;
D: node = node->next;

Inter-thread communication latency is a one-time cost
Implementing DSWP

DFG

A: r1 = M[r1]

F: p1 = r1 != 0

G: br p1, Loop

B: r2 = r1 + 4

C: r3 = M[r2]

D: r4 = r3 + 1

E: M[r2] = r4

L1:

SPAWN(Aux)
A: r1 = M[r1] PRODUCE [1] = r1
F: p1 = r1 != 0
G: br p1, L1

Aux:

CONSUME r1 = [1]
B: r2 = r1 + 4
C: r3 = M[r2]
D: r4 = r3 + 1
E: M[r2] = r4

register control memory
→ intra-iteration
- - - loop-carried
Optimization: Node Splitting
To Eliminate Cross Thread Control

L1
A: r1 = M[r1]
   PRODUCE [1] = r1
F: p1 = r1 != 0
   PRODUCE [2] = p1
G: br p1, L1

L2
CONSUME r1 = [1]
B: r2 = r1 + 4
C: r3 = M[r2]
D: r4 = r3 + 1
E: M[r2] = r4
   CONSUME p1 = [2]
G': br p1, L2

register
control
memory

intra-iteration
loop-carried
Optimization: Node Splitting To Reduce Communication

A: \( r1 = M[r1] \)
  PRODUCE \([1] = r1\)
F: \( p1 = r1 \neq 0 \)
G: \( \text{br } p1, \text{L1} \)

\(\text{CONSUME } r1 = [1]\)
B: \( r2 = r1 + 4 \)
C: \( r3 = M[r2] \)
D: \( r4 = r3 + 1 \)
E: \( M[r2] = r4 \)
F': \( p1 = r1 \neq 0 \)
G': \( \text{br } p1, \text{L2} \)

register  
control  
memory  
\(\rightarrow\) intra-iteration  
\(--\rightarrow\) loop-carried
Constraint: Strongly Connected Components

Consider:

Solution: $\text{DAG}_{\text{SCC}}$

- **Spawn (Aux)**
  - A: $r_1 = M[r_1]$
  - B: $r_2 = r_1 + 4$
  - C: $r_3 = M[r_2]$
  - Produce $[1] = r_3$
  - Consume $r_0 = [2]$
- **Control**
  - F: $p_1 = r_1 \neq 0$
  - G: $\text{br } p_1, L1$

- **Register Memory**
  - D: $r_4 = r_3 + 1$
  - E: $M[r_2] = r_4$
  - Produce $[2] = r_0$

Eliminates pipelined/decoupled property
2 Extensions to the Basic Transformation

❖ Speculation
  » Break statistically unlikely dependences
  » Form better-balanced pipelines

❖ Parallel Stages
  » Execute multiple copies of certain “large” stages
  » Stages that contain inner loops perfect candidates
Why Speculation?

A: while(node)
B: ncost = doit(node);
C: cost += ncost;
D: node = node->next;

Dependence Graph

DAG_{scc}

register
control

intra-iteration

loop-carried
communication queue
Why Speculation?

A: while(cost < T && node)
B: ncost = doit(node);
C: cost += ncost;
D: node = node->next;

Dependence
Graph

Predictable
Dependences

DAG_{sc}

Register
communication queue
control

register
control
intra-iteration
loop-carried
Why Speculation?

A: `while(cost < T && node)`
B: `ncost = doit(node);`
C: `cost += ncost;`
D: `node = node->next;`
Execution Paradigm

- DAG
- SCC
- Misspeculation
- Misspeculation Recovery
- Rerun Iteration 4
- Core 1
- Core 2
- Core 3
- Core 4
- D:1
- D:2
- D:3
- D:4
- D:5
- D:6
- D:7
- B:1
- B:2
- B:3
- B:4
- B:5
- B:6
- C:1
- C:2
- C:3
- C:4
- C:5
- A:1
- A:2
- A:3
- A:5
- A:6

- register communication queue
- control
- intra-iteration
- loop-carried

Misspeculation detected
Understanding PMT Performance

$T \propto \max(t_i)$

1. Rate $t_i$ is at least as large as the longest dependence recurrence.

2. NP-hard to find longest recurrence.

3. Large loops make problem difficult in practice.

Slowest thread: 1 cycle/iter
Iteration Rate: 1 iter/cycle

2 cycle/iter
0.5 iter/cycle
Selecting Dependences To Speculate

A: while(cost < T && node)
B:   ncost = doit(node);
C:   cost += ncost;
D:   node = node->next;

Dependence
Graph

DAG

DAG_{\text{SCC}}

Thread 1
Thread 2
Thread 3
Thread 4

register
control

intra-iteration

loop-carried

communication queue
Detecting Misspeculation

A¹: while(consume(4))
D: node = node->next
   produce({0,1},node);

A²: while(consume(5))
B: ncost = doit(node);
   produce(2,ncost);
D²: node = consume(0);

A³: while(consume(6))
B³: ncost = consume(2);
C: cost += ncost;
   produce(3,cost);

A: while(cost < T && node)
B⁴: cost = consume(3);
C⁴: node = consume(1);
   produce({4,5,6},cost < T && node);
Detecting Misspeculation

Thread 1

A¹: while(TRUE)
D: node = node->next
   produce({0,1},node);

A²: while(TRUE)
B: ncost = doit(node);
   produce(2,ncost);
D²: node = consume(0);

A³: while(TRUE)
B³: ncost = consume(2);
C: cost += ncost;
   produce(3,cost);

A: while(cost < T && node)
B⁴: cost = consume(3);
C⁴: node = consume(1);
   produce({4,5,6},cost < T && node);
Detecting Misspeculation

Thread 1
A^1: while(TRUE)
D: node = node->next
    produce({0,1},node);

Thread 2
A^2: while(TRUE)
B: ncost = doit(node);
    produce(2,ncost);
D^2: node = consume(0);

Thread 3
A^3: while(TRUE)
B^3: ncost = consume(2);
C: cost += ncost;
    produce(3,cost);

Thread 4
A: while(cost < T && node)
B^4: cost = consume(3);
C^4: node = consume(1);
if(!(cost < T && node))
    FLAG_MISSPEC();
Breaking False Memory Dependences

Dependence Graph

false memory
register
control
intra-iteration
loop-carried
communication queue

Oldest Version
Committed by
Recovery Thread

Memory
Version 3
Adding Parallel Stages to DSWP

while(ptr = ptr->next) // LD
ptr->val = ptr->val + 1; // X

LD = 1 cycle
X = 2 cycles

Comm. Latency = 2 cycles

Throughput
DSWP: 1/2 iteration/cycle
DOACROSS: 1/2 iteration/cycle
PS-DSWP: 1 iteration/cycle
Thread Partitioning

```c
p = list;
sum = 0;
A: while (p != NULL) {
B:    id = p->id;
E:    q = p->inner_list;
C:    if (!visited[id]) {
D:        visited[id] = true;
F:        while (foo(q))
G:            q = q->next;
H:        if (q != NULL)
I:            sum += p->value;
        }
J:    p = p->next;
}
```
Thread Partitioning: $\text{DAG}_{\text{SCC}}$
Thread Partitioning

Merging Invariants

- No cycles
- No loop-carried dependence inside a doall node

- register dependence
- control dependence

- doall
- sequential
Thread Partitioning

Treated as sequential
Thread Partitioning

- Modified MTCG[Ottoni, MICRO’05] to generate code from partition
Discussion Point 1 – Speculation

❖ How do you decide what dependences to speculate?
  » Look solely at profile data?
  » How do you ensure enough profile coverage?
  » What about code structure?
  » What if you are wrong? Undo speculation decisions at run-time?

❖ How do you manage speculation in a pipeline?
  » Traditional definition of a transaction is broken
  » Transaction execution spread out across multiple cores
Discussion Point 2 – Pipeline Structure

❖ When is a pipeline a good/bad choice for parallelization?

❖ Is pipelining good or bad for cache performance?
  » Is DOALL better/worse for cache?

❖ Can a pipeline be adjusted when the number of available cores increases/decreases?
CFGs, PCs, and Cross-Iteration Deps

1. \( r_1 = 10 \)

1. \( r_1 = r_1 + 1 \)

2. \( r_2 = \text{MEM}[r_1] \)

3. \( r_2 = r_2 + 1 \)

4. \( \text{MEM}[r_1] = r_2 \)

5. Branch \( r_1 < 1000 \)

No register live outs
Loop-Level Parallelization: DOALL

1. \( r1 = 10 \)
2. \( r1 = r1 + 1 \)
3. \( r2 = \text{MEM}[r1] \)
4. \( \text{MEM}[r1] = r2 \)
5. Branch \( r1 < 1000 \)

1. \( r1 = 9 \)
2. \( r1 = r1 + 2 \)
3. \( r2 = \text{MEM}[r1] \)
4. \( \text{MEM}[r1] = r2 \)
5. Branch \( r1 < 999 \)

1. \( r1 = 10 \)
2. \( r1 = r1 + 2 \)
3. \( r2 = \text{MEM}[r1] \)
4. \( \text{MEM}[r1] = r2 \)
5. Branch \( r1 < 1000 \)

No register live outs
Another Example

1. $r_1 = 10$

2. $r_1 = r_1 + 1$

3. $r_2 = MEM[r_1]$

4. $r_2 = r_2 + 1$

5. $MEM[r_1] = r_2$

6. Branch $r_2 == 10$

No register live outs
Another Example

1. \( r_1 = 10 \)

1. \( r_1 = r_1 + 1 \)  
2. \( r_2 = \text{MEM}[r_1] \)  
3. \( r_2 = r_2 + 1 \)  
4. \( \text{MEM}[r_1] = r_2 \)  
5. Branch \( r_2 == 10 \)

1. \( r_1 = 9 \)

1. \( r_1 = r_1 + 2 \)  
2. \( r_2 = \text{MEM}[r_1] \)  
3. \( r_2 = r_2 + 1 \)  
4. \( \text{MEM}[r_1] = r_2 \)  
5. Branch \( r_2 == 10 \)

1. \( r_1 = 10 \)

1. \( r_1 = r_1 + 2 \)  
2. \( r_2 = \text{MEM}[r_1] \)  
3. \( r_2 = r_2 + 1 \)  
4. \( \text{MEM}[r_1] = r_2 \)  
5. Branch \( r_2 == 10 \)

No register live outs
Speculation

1. \( r1 = 9 \)  
2. \( r2 = MEM[r1] \)  
3. \( r2 = r2 + 1 \)  
4. \( MEM[r1] = r2 \)  
5. Branch \( r2 == 10 \)

1. \( r1 = 10 \)  
2. \( r2 = MEM[r1] \)  
3. \( r2 = r2 + 1 \)  
4. \( MEM[r1] = r2 \)  
5. Branch \( r2 == 10 \)

No register live outs
Speculation, Commit, and Recovery

1. \( r_1 = 9 \)
2. \( r_1 = r_1 + 2 \)
3. \( r_2 = MEM[r_1] \)
4. \( r_2 = r_2 + 1 \)
5. \( \text{Send}\{1\} \text{ r2} \)
6. \( \text{Send}\{2\} \text{ r2} \)
7. \( \text{Send}\{2\} \text{ r2} \)

1. \( r_2 = \text{Receive}\{1\} \)
2. \( \text{Branch} \ r_2 \neq 10 \)
3. \( MEM[r_1] = r_2 \)
4. \( r_2 = \text{Receive}\{2\} \)
5. \( \text{Branch} \ r_2 \neq 10 \)
6. \( MEM[r_1] = r_2 \)
7. \( \text{Branch} \ r_2 \neq 10 \)
8. \( \text{Branch} \ r_2 \neq 10 \)
9. \( \text{Branch} \ r_2 \neq 10 \)
10. \( \text{Branch} \ r_2 \neq 10 \)

1. \( r_1 = 10 \)
2. \( r_1 = r_1 + 2 \)
3. \( r_2 = MEM[r_1] \)
4. \( r_2 = r_2 + 1 \)
5. \( \text{Branch} \ r_2 \neq 10 \)
6. \( MEM[r_1] = r_2 \)
7. \( \text{Branch} \ r_2 \neq 10 \)
8. \( MEM[r_1] = r_2 \)
9. \( \text{Branch} \ r_2 \neq 10 \)
10. \( \text{Branch} \ r_2 \neq 10 \)
11. \( \text{Branch} \ r_2 \neq 10 \)
12. \( \text{Branch} \ r_2 \neq 10 \)

No register live outs

1. \( \text{Kill and Continue} \)
Difficult Dependences

1. r1 = Head

1. r1 = MEM[r1]
1. Branch r1 == 0
2. r2 = MEM[r1 + 4]
1. r3 = Work (r2)
2. Print ( r3 )
3. Jump

No register live outs
1. \( r1 = \text{Head} \)

1. \( r1 = \text{MEM}[r1] \)

1. Branch \( r1 == 0 \)

2. \( r2 = \text{MEM}[r1 + 4] \)

1. \( r3 = \text{Work}(r2) \)

2. Print (\( r3 \))

3. Jump

No register live outs
1. \( r1 = \text{Head} \)

1. \( r1 = \text{MEM}[r1] \)

1. Branch \( r1 == 0 \)

2. \( r2 = \text{MEM}[r1 + 4] \)

1. \( r3 = \text{Work}(r2) \)

2. Print (r3)

3. Jump

No register live outs
Era of DIY:

- Multicore
- Reconfigurable
- GPUs
- Clusters

10 Cores!

10-Core Intel Xeon
“Unparalleled Performance”
P6 SUPERSCALAR ARCHITECTURE (CIRCA 1994)

- Automatic Speculation
- Automatic Pipelining
- Commit
- Parallel Resources
- Automatic Allocation/Scheduling
<table>
<thead>
<tr>
<th>ABCPL</th>
<th>CORRELATE</th>
<th>GLU</th>
<th>GUARD</th>
<th>ATM</th>
<th>Mentat</th>
<th>Parafrase2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>CPS</td>
<td>HAAsL.</td>
<td>Haskell</td>
<td>Legion</td>
<td>SCHEDULE</td>
<td></td>
</tr>
<tr>
<td>ACT++</td>
<td>CRL</td>
<td>HPC</td>
<td>JAVAR.</td>
<td>Meta Chaos</td>
<td>ScrTL</td>
<td></td>
</tr>
<tr>
<td>Active messages</td>
<td>CSP</td>
<td>HORUS</td>
<td>JAVAR.</td>
<td>Midway</td>
<td>POET</td>
<td></td>
</tr>
<tr>
<td>Adl</td>
<td>Cthreads</td>
<td>HPc</td>
<td>IMPACT</td>
<td>Millipede</td>
<td>SDDA.</td>
<td></td>
</tr>
<tr>
<td>Adsmith</td>
<td>CUMULVS</td>
<td>JAPE</td>
<td>IRAc</td>
<td>CparPar</td>
<td>SHMEM</td>
<td></td>
</tr>
<tr>
<td>ADDAP</td>
<td>DAGGER</td>
<td>Java RMI</td>
<td>Mirage</td>
<td>ParC</td>
<td>SIMPLE</td>
<td></td>
</tr>
<tr>
<td>AFAPI</td>
<td>DAPPLE</td>
<td>javaPG</td>
<td>MpC</td>
<td>ParLib++</td>
<td>Sina</td>
<td></td>
</tr>
<tr>
<td>ALWAN</td>
<td>Data Parallel C</td>
<td>JavaSpace JOYce</td>
<td>MOSIX</td>
<td>ParLin</td>
<td>SISAL.</td>
<td></td>
</tr>
<tr>
<td>AM</td>
<td>DC++</td>
<td>JIDL</td>
<td>MODula-P</td>
<td>MuP</td>
<td>distributed smalltalk</td>
<td></td>
</tr>
<tr>
<td>AMDC</td>
<td>DCE++</td>
<td>Karma</td>
<td>ModuLa-2*</td>
<td>MuP++</td>
<td>SMI.</td>
<td></td>
</tr>
<tr>
<td>AppLeS</td>
<td>DDDD</td>
<td>Khoros</td>
<td>Nexus</td>
<td>MuN</td>
<td>SONiC</td>
<td></td>
</tr>
<tr>
<td>Amoeba</td>
<td>DICE.</td>
<td>Nexus</td>
<td>Nimrod</td>
<td>PEACE</td>
<td>Split-C.</td>
<td></td>
</tr>
<tr>
<td>ARTS</td>
<td>DIPC</td>
<td>NIHroS</td>
<td>Now</td>
<td>PCU</td>
<td>SRT</td>
<td></td>
</tr>
<tr>
<td>Athapascan-0b</td>
<td>DOLIB</td>
<td>JIDL</td>
<td>Occam</td>
<td>PEAT</td>
<td>S threads</td>
<td></td>
</tr>
<tr>
<td>Aurora</td>
<td>DOME</td>
<td>JADa</td>
<td>Omega</td>
<td>PET</td>
<td>Strand.</td>
<td></td>
</tr>
<tr>
<td>Automap</td>
<td>DOSMOS.</td>
<td>WWWinda</td>
<td>OpenMP</td>
<td>PETSc</td>
<td>SUIF.</td>
<td></td>
</tr>
<tr>
<td>bb_threads</td>
<td>DRL</td>
<td>Karma</td>
<td>ORca</td>
<td>PENNY</td>
<td>Synergy.</td>
<td></td>
</tr>
<tr>
<td>Blaze</td>
<td>DSM-Threads</td>
<td>Khoros</td>
<td>ORca</td>
<td>Phosphorus</td>
<td>Telephos</td>
<td></td>
</tr>
<tr>
<td>BSP</td>
<td>Ease.</td>
<td>Khoros</td>
<td>OpenMP</td>
<td>POE.</td>
<td>SuperPascal</td>
<td></td>
</tr>
<tr>
<td>BlockComm</td>
<td>ECO</td>
<td>Karma</td>
<td>ParLin</td>
<td>Polaris</td>
<td>TCGMSG.</td>
<td></td>
</tr>
<tr>
<td>C*</td>
<td>Eiffel</td>
<td>Lilac</td>
<td>ParLin</td>
<td>POOMA</td>
<td>Threads.h++.</td>
<td></td>
</tr>
<tr>
<td>&quot;C* in C</td>
<td>eilean</td>
<td>Linda</td>
<td>ParLin</td>
<td>POOL-T</td>
<td>TreadMarks</td>
<td></td>
</tr>
<tr>
<td>C**</td>
<td>Emerald</td>
<td>Linda</td>
<td>ParLin</td>
<td>PRESTO</td>
<td>TRAPPER</td>
<td></td>
</tr>
<tr>
<td>CarIoS</td>
<td>EPL</td>
<td>Linda</td>
<td>Para++</td>
<td>P-rio</td>
<td>uC++</td>
<td></td>
</tr>
<tr>
<td>Cashmere</td>
<td>Excalibur</td>
<td>Linda</td>
<td>Para++</td>
<td>Prospero</td>
<td>UC</td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>Express</td>
<td>Linda</td>
<td>Para++</td>
<td>Proteus</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>CC++</td>
<td>Falcon</td>
<td>Linda</td>
<td>Para++</td>
<td>QPC++</td>
<td>ViC*</td>
<td></td>
</tr>
<tr>
<td>Chu</td>
<td>Filaments</td>
<td>Linda</td>
<td>Para++</td>
<td>PVM</td>
<td>Visifold V-NUS</td>
<td></td>
</tr>
<tr>
<td>Charlotte</td>
<td>FM</td>
<td>Linda</td>
<td>Para++</td>
<td>PSI</td>
<td>VPE</td>
<td></td>
</tr>
<tr>
<td>Charm</td>
<td>FLASH</td>
<td>Objective-Linda</td>
<td>Para++</td>
<td>PSDM</td>
<td>Win32 threads</td>
<td></td>
</tr>
<tr>
<td>Charm++</td>
<td>The FORCE</td>
<td>Objective-Linda</td>
<td>Para++</td>
<td>Quake</td>
<td>WinPar</td>
<td></td>
</tr>
<tr>
<td>Cid</td>
<td>Fork</td>
<td>Objective-Linda</td>
<td>Para++</td>
<td>Quark</td>
<td>WWWinda</td>
<td></td>
</tr>
<tr>
<td>Cilk</td>
<td>Fortran-M</td>
<td>Objective-Linda</td>
<td>Para++</td>
<td>Quick Threads</td>
<td>XENOOPS</td>
<td></td>
</tr>
<tr>
<td>CM-Fortran</td>
<td>FX</td>
<td>Objective-Linda</td>
<td>Para++</td>
<td>Sage++</td>
<td>XPC</td>
<td></td>
</tr>
<tr>
<td>Converse</td>
<td>GA</td>
<td>Objective-Linda</td>
<td>Para++</td>
<td>SCANDAL</td>
<td>Zounds</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>GAMMA</td>
<td>Objective-Linda</td>
<td>Para++</td>
<td>SAM</td>
<td>ZPL</td>
<td></td>
</tr>
<tr>
<td>COOL</td>
<td>Glenda</td>
<td>Objective-Linda</td>
<td>Para++</td>
<td>Sam</td>
<td>ZPL</td>
<td></td>
</tr>
</tbody>
</table>
Parallel Library Calls

Realizable parallelism

Credit: Jack Dongarra
“Compiler Advances Double Computing Power Every 18 Years!”
– Proebsting’s Law
P6 SUPERSCALAR ARCHITECTURE

Spec-PS-DSWP

Core 1  Core 2  Core 3  Core 4

Spec-PS-DSWP

LD:1  LD:2  LD:3  LD:4  LD:5

W:1  W:2  W:3  W:4

C:1  C:2  C:3

P6 SUPERSCALAR ARCHITECTURE
Example
A: while (node) {
B:   node = node->next;
C:   res = work(node);
D:   write(res);
}

Program Dependence Graph

Control Dependence
Data Dependence

Time

Core 1 | Core 2 | Core 3
-------|-------|-------
A1     | B1    | C1    | D1
A2     | B2    | C2    | D2
A: while (node) {
B:   node = node->next;
C:   res = work(node);
D:   write(res);
}
Example
A: while (node) {
B:   node = node->next;
C:   res = work(node);
D:   write(res);
}

Program Dependence Graph

Control Dependence
Data Dependence
Example

A: while (node) {
B:   node = node->next;
C:   res = work(node);
D:   write(res);
}

Program Dependence Graph

Spec-DOALL

<table>
<thead>
<tr>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>A2</td>
<td>A3</td>
</tr>
<tr>
<td>B1</td>
<td>B2</td>
<td>B3</td>
</tr>
<tr>
<td>C1</td>
<td>C2</td>
<td>C3</td>
</tr>
<tr>
<td>D1</td>
<td>D2</td>
<td>D3</td>
</tr>
<tr>
<td>B2</td>
<td>B3</td>
<td>B4</td>
</tr>
<tr>
<td>C2</td>
<td>C3</td>
<td>C4</td>
</tr>
<tr>
<td>D2</td>
<td>D3</td>
<td>D4</td>
</tr>
</tbody>
</table>

Slowdown

Number of Threads

Control Dependence

Data Dependence

197.parser

1000

100

10

1

Time
Spec-DOACROSS
Throughput: 1 iter/cycle

Spec-DSWP
Throughput: 1 iter/cycle
Comparison: Spec-DOACROSS and Spec-DSWP

Comm. Latency = 1: 1 iter/cycle
Comm. Latency = 2: 0.5 iter/cycle
Comm. Latency = 2: 1 iter/cycle

Core 1 | Core 2 | Core 3
--- | --- | ---
B1 | C1 | B2
C1 | D1 | C2
D1 | B2 | C3
B2 | C2 | D2
C2 | D2 | C3
B3 | C3 | D3

Time

Pipeline Fill time
Spec-DOACROSS vs. Spec-DSWP

[MICRO 2010]

Geomean of 11 benchmarks on the same cluster
Performance relative to Best Sequential
128 Cores in 32 Nodes with Intel Xeon Processors [MICRO 2010]
“Compiler Advances Double Computing Power Every 18 Years!”
– Proebsting’s Law

Era of DIY:
• Multicore
• Reconfigurable
• GPUs
• Clusters

Compiler technology inspired class of architectures?
CFGs and PCs