Topic 14: Parallelism

COS 320
Compiling Techniques
Princeton University
Spring 2018
Prof. David August

Final Exam!

- Thursday May 3 in class
- Closed book, closed notes

Moore’s Law
Source: Intel/Wikipedia

Single-Threaded Performance Not Improving
Source: Intel/Wikipedia
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
    ✶ Are parallel programs? Ask GPU developers 😊
  » Performance debugging of sequential programs straight-forward

Compilers are the Answer? - Proebsting’s Law

❖ "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!

Parallelizing Loops In Scientific Applications

Scientific Codes (FORTRAN-like)

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

Example: DOALL parallelization

A Step Back in Time: Old Skool 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
```

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

Array Dependence Analysis

```
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 refers 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

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

Back to the Present – Parallelizing C and C++ Programs
What’s the Problem?

1. Memory dependence analysis
   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!
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

What’s the Next Problem?

3. C/C++ too restrictive

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

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

Loops cannot be parallelized even if computation is independent
char *memory;

@Commutative
void * alloc(int size);

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

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

int main() {
    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++;
        if (profitable) {
            dict = restart(dict);
        }
    }
    finish_dict(dict);
    return 0;
}
```
```
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.

```
dict = create_dict();
while((char = read(1))) {
    profitable = compress(char, dict)
    @YBRANCH(probability=.00001)
    if (!profitable) {
        dict = restart(dict);
    }
} finish_dict(dict);
```

Capturing Output/Performance Tradeoff: Y-Branches in 164.gzip

```c
#define CUTOFF 100000
dict = create_dict();
count = 0;
while((char = read(1))) {
    profitable = compress(char, dict)
    @YBRANCH(probability=.00001)
    if (!profitable) {
        dict = restart(dict);
    }
    if (count == CUTOFF) {
        dict = restart(dict);
        count = 0;
    }
    count++;
} finish_dict(dict);
```

```
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.

```
unsigned char *block;
int last_written;
compressStream(in, out) {
    while (True) {
        loadAndRLEsource(in);
        if (!last) break;
        doReversibleTransform();
        sendMTFValues(out);
    }
}
doReversibleTransform() {
    sortIt();
    ...
    sortIt() {
        printf(...);
    }
}
```

```
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.
What prevents the automatic extraction of parallelism?

Lack of an Aggressive Compilation Framework

Sequential Programming Model

Performance Potential

Existing
Framework + Annotations

164.gzip 26 x x x
175.vpr 1 x x x
176.gcc 18 x x x
181.mcf 0 x
186.crafty 9 x x x x
197.parser 3 x x
253.perlbmk 0 x x x x
254.gap 3 x x
255.vortex 0 x x x
256.bzip2 0 x x
300.twolf 1 x x

Modified only 60 LOC out of ~500,000 LOC

What About Non-Scientific Codes???

Scientific Codes (FORTRAN-like)

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

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

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

Alternative Parallelization Approaches

Independent Multithreading (IMT)
Example: DOALL parallelization

Cyclic Multithreading (CMT)
Example: DOACROSS
[Cytron, ICPP 86]

Pipelined Multithreading (PMT)
Example: DSWP
[PACT 2004]
Our Objective: Automatic Extraction of Pipeline Parallelism using DSWP

Decoupled Software Pipelining (DSWP)

[MICRO 2005]

Implementing DSWP

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

 DFG

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

Inter-thread communication latency is a one-time cost
Optimization: Node Splitting To Eliminate Cross Thread Control

L1
L2

Optimization: Node Splitting To Reduce Communication

L1
L2

Constraint: Strongly Connected Components

Consider:

Solution: DAG_{SCC}

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

Execution Paradigm

DAG_{SCC}
Understanding PMT Performance

- Core 1
  A:1
  A:2
  A:3
  A:4
  A:5
  A:6
- Core 2
  B:1
  B:2
  B:3
  B:4
  B:5

$T \propto \max(t_i)$

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.

Selecting Dependences To Speculate

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

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

- 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

\[ A^1: \text{while(TRUE)} \]
\[ D: \text{node} = \text{node} \rightarrow \text{next} \]
\[ \quad \text{produce}((0,1),\text{node}); \]

\[ A^2: \text{while(TRUE)} \]
\[ B: \text{ncost} = \text{doit(\text{node})}; \]
\[ \quad \text{produce}(2,\text{ncost}); \]
\[ D^2: \text{node} = \text{consume}(0); \]

\[ A^3: \text{while(TRUE)} \]
\[ B^3: \text{ncost} = \text{consume}(2); \]
\[ C: \text{cost} += \text{ncost}; \]
\[ \quad \text{produce}(3,\text{cost}); \]

\[ A: \text{while(cost < T && node)} \]
\[ B^4: \text{cost} = \text{consume}(3); \]
\[ C^4: \text{node} = \text{consume}(1); \]
\[ \quad \text{if}(!\text{(cost < T && node)}) \]
\[ \quad \quad \text{FLAG_MISSPEC();} \]

Breaking False Memory Dependences

Adding Parallel Stages to DSWP

Thread Partitioning

\[ p = \text{list}; \]
\[ \text{sum} = 0; \]
\[ A: \text{while (p !\= NULL) \{} \]
\[ B: \quad \text{id} = p\rightarrow\text{id}; \]
\[ E: \quad q = p\rightarrow\text{inner_list}; \]
\[ C: \quad \text{if} (!\text{visited[id]}) \{ \]
\[ D: \quad \text{visited[id]} = \text{true}; \]
\[ F: \quad \text{while (foo(q))} \]
\[ G: \quad \text{q} = q\rightarrow\text{next}; \]
\[ H: \quad \text{if} (q !\= \text{NULL}) \]
\[ I: \quad \text{sum} += p\rightarrow\text{value}; \]
\[ J: \quad p = p\rightarrow\text{next}; \}

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: \( \text{DAG}_{\text{SCC}} \)

- \( A \) to \( J \)
- \( B \) to \( D \)
- \( C \) to \( E \)
- \( F \) to \( G \)
- \( H \) to \( I \)

- \( \text{register dependence} \)
- \( \text{control dependence} \)

Thread Partitioning

Merging Invariants

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

- \( \text{register dependence} \)
- \( \text{control dependence} \)
- \( \text{doall} \)
- \( \text{sequential} \)

Thread Partitioning

- \( A \) to \( J \)
- \( B \) to \( D \)
- \( C \) to \( D \)
- \( E \) to \( F \)
- \( G \) to \( H \)
- \( H \) to \( I \)

- \( \text{Treated as sequential} \)

Thread Partitioning

- \( A \) to \( B \) to \( C \) to \( D \) to \( I \)
- \( E \) to \( F \) to \( G \) to \( H \) to \( I \)

- \( \text{Modified MTCG[Ottoni, MICRO’05]} \) 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. \(r_1 = 10\)
2. \(r_1 = r_1 + 1\)
3. \(r_2 = \text{MEM}[r_1]\)
4. \(r_2 = r_2 + 1\)
5. \(\text{MEM}[r_1] = r_2\)
6. Branch \(r_1 < 1000\)

Loop-Level Parallelization: DOALL

1. \(r_1 = 10\)
2. \(r_1 = r_1 + 1\)
3. \(r_2 = \text{MEM}[r_1]\)
4. \(r_2 = r_2 + 1\)
5. \(\text{MEM}[r_1] = r_2\)
6. Branch \(r_1 < 1000\)

1. \(r_1 = 9\)
2. \(r_1 = r_1 + 2\)
3. \(r_2 = \text{MEM}[r_1]\)
4. \(r_2 = r_2 + 1\)
5. \(\text{MEM}[r_1] = r_2\)
6. Branch \(r_1 < 1000\)

1. \(r_1 = 10\)
2. \(r_1 = r_1 + 2\)
3. \(r_2 = \text{MEM}[r_1]\)
4. \(r_2 = r_2 + 1\)
5. \(\text{MEM}[r_1] = r_2\)
6. Branch \(r_1 < 1000\)

No register live outs

No register live outs
Another Example

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

No register live outs

Speculation

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

No register live outs

Speculation, Commit, and Recovery

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

No register live outs

1. \( r2 = Receive\{1\} \)
2. Branch \( r2 != 10 \)
3. \( MEM[r1] = r2 \)
4. \( r2 = Receive\{2\} \)
5. Branch \( r2 != 10 \)
6. \( MEM[r1] = r2 \)
7. Jump

No register live outs

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

1. Kill and Continue
Difficult Dependences

1. \( r1 = \text{MEM}[r1] \)
2. \( \text{Branch } r1 == 0 \)
3. \( r2 = \text{MEM}[r1 + 4] \)
4. \( r3 = \text{Work}(r2) \)
5. \( \text{Print}(r3) \)
6. \( \text{Jump} \)

No register live outs

DOACROSS

1. \( r1 = \text{Head} \)
2. \( \text{Branch } r1 == 0 \)
3. \( r2 = \text{MEM}[r1 + 4] \)
4. \( r3 = \text{Work}(r2) \)
5. \( \text{Print}(r3) \)
6. \( \text{Jump} \)

No register live outs

PS-DSWP

1. \( r1 = \text{Head} \)
2. \( \text{Branch } r1 == 0 \)
3. \( r2 = \text{MEM}[r1 + 4] \)
4. \( r3 = \text{Work}(r2) \)
5. \( \text{Print}(r3) \)
6. \( \text{Jump} \)

No register live outs

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

10 Cores!

10-Core Intel Xeon "Unparalleled Performance"
Compiler Advances Double Computing Power Every 18 Years!
– Proebsting’s Law

P6 SUPERSCALAR ARCHITECTURE

Example
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);
}
Example

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

Spec-DOALL

Throughput: 1 iter/cycle

Spec-DOACROSS

Comparison: Spec-DOACROSS and Spec-DSWP

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

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: 1 iter/cycle

Pipeline

Fill time

0.5 iter/cycle

1 iter/cycle

0.5 iter/cycle

1 iter/cycle
Performance relative to Best Sequential
128 Cores in 32 Nodes with Intel Xeon Processors [MICRO 2010]

Performance Speedup (X)
(Number of Total Cores, Number of Nodes)

Spec-DOACROSS vs. Spec-DSWP
Geomean of 11 benchmarks on the same cluster

Spec-DOACROSS vs. Spec-DSWP [MICRO 2010]

TLS
Spec-PS-DSWP
Spec-DOACROSS vs. Spec-DSWP

Geomean of 11 benchmarks on the same cluster

Compiler Advances Double Computing Power Every 18 Years!
— Proebsting’s Law

Compiler Technology
Architecture/Devices
Era of DIY:
• Multicore
• Reconfigurable
• GPUs
• Clusters

Compiler technology inspired class of architectures?