CIS 431/531
Parallel Computing
Dependencies

Department of Computer and Information Science
Winter 2018
Logistics

- Programming lab
  - Instructions for lab assignment submission
  - TBB programming problem posted by Friday COB
  - Brief introduction to the TBB programming problem at the end of lecture
Outline

- Parallel programming models
- Dependencies
Parallel Models 101

- Sequential models
  - von Neumann (RAM) model

- Parallel model
  - A parallel computer is simply a collection of processors interconnected in some manner to coordinate activities and exchange data.
  - Models that can be used as general frameworks for describing and analyzing parallel algorithms:
    - Simplicity: description, analysis, architecture independence
    - Implementability: able to be realized, reflect performance

- Three common parallel models
  - Directed acyclic graphs, shared-memory, network
Directed Acyclic Graphs (DAG)

- Captures data flow parallelism
- Nodes represent operations to be performed
  - Inputs are nodes with no incoming arcs
  - Output are nodes with no outgoing arcs
  - Think of nodes as *tasks* to be performed
- Arcs are paths for flow of data (in and out of tasks)
- DAG represents the algorithm operations and implies precedent constraints on their order of execution

```c
for (i=1; i<100; i++)
a[i] = a[i-1] + 100;
```

- Where is the parallelism?
  - Think about when a node can execute

Can this DAG be executed in parallel?
Shared Memory Model

- Parallel extension of RAM model (PRAM)
  - Memory size is infinite
  - Number of processors is unbounded
  - Processors communicate via the memory
  - Every processor accesses in 1 cycle
  - Synchronous execution
    - All processors execute the same algorithm synchronously
      - READ phase
      - COMPUTE phase
      - WRITE phase
    - Some subset of processors can stay idle (implicit synchronization)
  - Asynchronous execution
    - Processors run independently
    - Synchronization using memory
Network Model

- \( G = (N,E) \)
  - \( N \) are processing nodes
  - \( E \) are bidirectional communication links
  - Edges define a network “topology”
- Each processor has its own memory
- No shared memory is available
- Network operation
  - Synchronous or asynchronous
- Requires communication primitives
  - \textit{Send} \((X, i)\)
  - \textit{Receive} \((Y, j)\)
- Captures message passing model for algorithm design
Parallelism

- Ability to execute different parts of a computation concurrently on different machines

- Why do you want parallelism?
  - Shorter running time (time to solution)
  - Handling more work (bigger problems, throughput)

- What is being parallelized?
  - Task: instruction, statement, procedure, …
  - Data: data flow, size, replication
  - Parallelism granularity (both in task and data)
    - coarse-grain versus fine-grained

- Thinking about parallelism (concurrency, efficiency)

- Evaluation
Why is parallel programming important?

- Parallel programming has matured
  - Standard programming models
  - Common machine architectures
  - Programmer can focus on computation and use suitable programming model for implementation

- Increase portability between models and architectures

- Reasonable hope of portability across platforms

- Problem
  - Performance optimization is still platform-dependent
  - Performance portability is a problem
  - Parallel programming methods are still evolving
Parallel Algorithm

- Recipe to solve a problem “in parallel” on multiple processing elements

- Standard steps for constructing a parallel algorithm
  - Identify work that can be performed concurrently
  - Partition the concurrent work on separate processors
  - Properly manage input, output, and intermediate data
  - Coordination to satisfy dependencies
    - data consistency
  - Coordination to manage parallelism

- Which are hard to do?
Parallelism Views

- Where can we find parallelism?

- Program (task) view
  - Statement level
    - Between program statements
    - Which statements can be executed at the same time?
  - Block level / Loop level / Routine level / Process level
    - Larger-grained program statements

- Data view
  - How is data operated on?
  - Where does data reside?

- Resource view
Parallelism, Correctness, and Dependence

- Parallel execution, from any point of view, will be constrained by the sequence of operations needed to be performed for a correct result.
- Parallel execution must address control, data, and system dependences.
- A *dependency* arises when one operation depends on an earlier operation to complete and produce a result before this later operation can be performed.
- We extend this notion of dependency to resources since some operations may depend on certain resources:
  - Due to where data is located
  - Due to what processing resources might be available.
Executing Two Statements in Parallel

- Want to execute two statements in parallel
  - Execute *at the same time on different processors*

- On one processor:
  - Statement 1;
  - Statement 2;

- On two processors:
  - Processor 1: Statement 1;
  - Processor 2: Statement 2;

- Fundamental (*concurrent*) execution assumption
  - Processors execute independent of each other
  - No assumptions made about speed of processor execution
  - why do we care?
Sequential Consistency in Parallel Execution

- **Case 1:**
  - Processor 1: Statement 1;
  - Processor 2: Statement 2;

- **Case 2:**
  - Processor 1: Statement 2;
  - Processor 2: Statement 1;

**Sequential consistency**
- Statements execution does not interfere with each other
- Computation results are the same (independent of order)
Independent versus Dependent

- In other words the execution of
  Statement1;
  Statement2;
  must be equivalent to (i.e., producing the same outcome)
  Statement2;
  Statement1;

- Sequential consistency means that the statement order of execution must not matter!

- If true, the statements are *independent* of each other

- Two statements are *dependent* when the order of their execution affects the computation outcome
Examples

- **Example 1**
  
  \( S_1: a=1; \)
  
  \( S_2: b=1; \)

- **Example 2**
  
  \( S_1: a=1; \)
  
  \( S_2: b=a; \)

- **Example 3**
  
  \( S_1: a=f(x); \)
  
  \( S_2: a=b; \)

- **Example 4**
  
  \( S_1: a=b; \)
  
  \( S_2: b=1; \)

Independent or dependent?

- **Example 1**
  
  \((a=5, b=6) S_1 \) then \( S_2 (a=1, b=1) \)
  
  \((a=5, b=6) S_2 \) then \( S_1 (a=1, b=1) \)

- **Example 2**
  
  \((a=5, b=6) S_1 \) then \( S_2 (a=1, b=1) \)
  
  \((a=5, b=6) S_2 \) then \( S_1 (a=1, b=5) \)

- **Example 3**
  
  \((a=5, b=6) S_1 \) then \( S_2 (a=6) \)
  
  \((a=5, b=6) S_2 \) then \( S_1 (a=f(x)) \)

- **Example 4**
  
  \((a=5, b=6) S_1 \) then \( S_2 (a=6, b=1) \)
  
  \((a=5, b=6) S_2 \) then \( S_1 (a=1, b=1) \)
Examples

Example 1
S₁: a=1;
S₂: b=1;
 Statements are independent

Example 2
S₁: a=1;
S₂: b=a;
 Dependent (true (flow) dependence)
  Second is dependent on first
  Can you remove dependency?

Example 3
S₁: a=f(x);
S₂: a=b;
 Dependent (output dependence)
  Second is dependent on first
  Can you remove dependency? How?

Example 4
S₁: a=b;
S₂: b=1;
 Dependent (anti-dependence)
  First is dependent on second
  Can you remove dependency? How?
True Dependence and Anti-Dependence

- Given statements $S_1$ and $S_2$,
  
  $S_1$;
  
  $S_2$;

- $S_2$ has a true (flow) dependence on $S_1$ if and only if
  
  $S_2$ reads a value written by $S_1$

- $S_2$ has an anti-dependence on $S_1$ if and only if
  
  $S_2$ writes a value read by $S_1$
Output Dependence

- Given statements \( S_1 \) and \( S_2 \),
  
  \[
  S_1; \\
  S_2; 
  \]

- \( S_2 \) has an *output dependence* on \( S_1 \) if and only if
  
  \( S_2 \) writes a variable written by \( S_1 \)

- Anti- and output dependences are “name” dependencies
  
  - Are they “true” dependences?

- How can you get rid of output dependences?
  
  - Are there cases where you can not?
Statement Dependency Graphs

- Can use graphs to show dependence relationships
- Example
  
  \[ S_1: \ a = 1; \]
  
  \[ S_2: \ b = a; \]
  
  \[ S_3: \ a = b + 1; \]
  
  \[ S_4: \ c = a; \]

- \( S_2 \delta \ S_3 \): \( S_3 \) is flow-dependent on \( S_2 \)
- \( S_1 \delta^0 \ S_3 \): \( S_3 \) is output-dependent on \( S_1 \)
- \( S_2 \delta^{-1} \ S_3 \): \( S_3 \) is anti-dependent on \( S_2 \)
When can two statements execute in parallel?

- Statements $S_1$ and $S_2$ can execute in parallel if and only if there are \textit{no dependences} between $S_1$ and $S_2$
  - True dependences
  - Anti-dependences
  - Output dependences

- Some dependences can be removed by modifying the program
  - Rearranging statements
  - Eliminating statements

- Dependence analysis came from compilers looking for opportunities for improving ILP
How do you compute dependence?

- Data dependence relations can be found by comparing the \(IN\) and \(OUT\) sets of each node.
- The \(IN\) and \(OUT\) sets of a statement \(S\) are defined as:
  - \(IN(S)\): set of memory locations (variables) that may be used in \(S\)
  - \(OUT(S)\): set of memory locations (variables) that may be modified by \(S\)
- Note that these sets include all memory locations that may be fetched or modified.
- As such, the sets can be conservatively large.
IN / OUT Sets and Computing Dependence

Assuming that there is a path from $S_1$ to $S_2$, the following shows how to intersect the IN and OUT sets to test for data dependence

\[ \text{out}(S_1) \cap \text{in}(S_2) \neq \emptyset \quad S_1 \delta S_2 \quad \text{flow dependence} \]
\[ \text{in}(S_1) \cap \text{out}(S_2) \neq \emptyset \quad S_1 \delta^{-1} S_2 \quad \text{anti-dependence} \]
\[ \text{out}(S_1) \cap \text{out}(S_2) \neq \emptyset \quad S_1 \delta^0 S_2 \quad \text{output dependence} \]
Loop-Level Parallelism

- Significant parallelism can be identified within loops

```c
for (i=0; i<100; i++)
   S1: a[i] = i;
```

- Dependencies? What about \( i \), the loop index?

- **DOALL** loop (a.k.a. *foreach* loop)
  - All iterations are independent of each other
  - All statements be executed in parallel at the same time
    - is this really true?

- Imagine that the loop is *unrolled*
Iteration Space

- Unroll loop into separate statements / iterations
- Show dependences between iterations

```plaintext
for (i=0; i<100; i++)
  S_1: a[i] = i;
for (i=0; i<100; i++) {
  S_1: a[i] = i;
  S_2: b[i] = 2*i;
}
```
Multi-Loop Parallelism

- Significant parallelism can be identified between loops.

```c
for (i=0; i<100; i++) a[i] = i;
for (i=0; i<100; i++) b[i] = i;
```

- Dependencies?
- How much parallelism is available?
- Given 4 processors, how much parallelism is possible?
- What parallelism is achievable with 50 processors?
Multi-Loop Parallelism

- Significant parallelism can be identified *between* loops

```c
for (i=0; i<100; i++) a[i] = i;
c = 5
for (i=0; i<100; i++) b[i] = c;
```

- Dependencies?
- How much parallelism is available?
- Given 4 processors, how much parallelism is possible?
- What parallelism is achievable with 50 processors?
Loops with Dependencies

Case 1:
for (i=1; i<100; i++)
    a[i] = a[i-1] + 100;

- Dependencies?
  - What type?
- Is the Case 1 loop parallelizable?

Case 2:
for (i=5; i<100; i++)
a[i-5] = a[i] + 100;

- Is the Case 2 loop parallelizable?
Another Loop Example

for (i=1; i<100; i++)
    a[i] = f(a[i-1]);

- Dependencies?
  - What type?

- Loop iterations are not parallelizable
  - Why not?
Loop Dependencies

- A *loop-carried* dependence is a dependence that is present only if the statements are part of the execution of a loop (i.e., between two statements instances in two different iterations of a loop)
- Otherwise, it is *loop-independent*, including between two statements instances in the same loop iteration.
- Loop-carried dependences can prevent loop iteration parallelization.
- The dependence is *lexically forward* if the source comes before the target or *lexically backward* otherwise.
  - Use *loop unrolling* to see
Loop Dependence Example

for (i=0; i<100; i++)
    a[i+10] = f(a[i]);

- Dependencies?
  - Between a[10], a[20], …
  - Between a[11], a[21], …

- Some parallel execution is possible
  - How much?
Dependences Between Iterations

for (i=1; i<100; i++) {
    S1: a[i] = …;
    S2: … = a[i-1];
}

- Dependencies?
  - Between a[i] (S1) and a[i-1] (S2)

- Is parallelism possible?
  - Statements can be executed in “pipeline” manner
  - What else?
Another Loop Dependence Example

```c
for (i=0; i<100; i++)
    for (j=1; j<100; j++)
        a[i][j] = f(a[i][j-1]);
```

- **Dependencies?**
  - Loop-independent dependence on \( i \)
  - Loop-carried dependence on \( j \)

- **Which loop can be parallelized?**
  - Outer loop parallelizable
  - Inner loop cannot be parallelized
Still Another Loop Dependence Example

for (j=1; j<100; j++)
    for (i=0; i<100; i++)
        a[i][j] = f(a[i][j-1]);

- Dependencies?
  - Loop-independent dependence on $i$
  - Loop-carried dependence on $j$

- Which loop can be parallelized?
  - Inner loop parallelizable
  - Outer loop cannot be parallelized
  - Less desirable (why?)
Key Ideas for Dependency Analysis

- To identify parallelism opportunities:
  - Statement order must not matter
  - Statements must not have dependences
- Some dependences can be removed
- Some dependences may not be obvious
Dependencies and Synchronization

- How is parallelism achieved when have dependencies?
  - Think about concurrency
  - Some parts of the execution are independent
  - Some parts of the execution are dependent

- Must control ordering of events on different processors (cores)
  - Dependencies pose constraints on parallel event ordering
  - Partial ordering of execution action

- Use synchronization mechanisms
  - Need for concurrent execution too
  - Maintains *partial order*
TBB Programming Assignment

- Play off of the ispc programming assignment
- Given multiple 2D images of different sizes ...
  - Each in its own file
- ... apply filtering to each one
  - Write an output file

Approach

1) Use TBB tasking to create 1 task for each image
   - Filtering is done sequentially within each task
2) Use parallel for to process filtering in parallel
   - Each image is done sequentially one after the other
3) Combine 1) and 2)