CIS 631
Parallel Processing

Lecture 5: Parallel Programming

Allen D. Malony

malony@cs.uoregon.edu

Department of Computer and Information Science

University of Oregon

Acknowledgements

- Portions of the lectures slides were adopted from:
  - I. Foster, "Designing and Building Parallel Programs," 1995.
  - John Mellor-Crummey, COMP 422, "Parallel Computing," Rice University, Spring 2010.

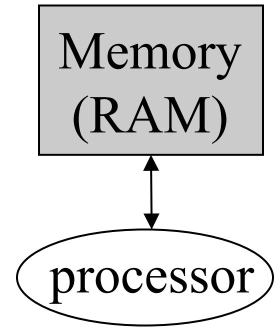
Term Project

- Work on a code for parallel particle advection
  - Contribute by Prof. Hank Childs
  - Start by reading "GPU Acceleration of Particle Advection Workloads in a Parallel, Distributed Memory Setting"
  - Skype with Prof. Childs on Wednesday during class meeting to discuss
  - Verbal agreement needed to keep confidential the paper, the code, the ideas contained in each
- Work with David Ozog's Framework for Parallel Task Operation (FRAMPTON)
  - Extend how FRAMPTON can process task DAGs
  - Develop new tasks adaptors

Outline

- Quick look at parallel models
- Parallelism
  - Where can you find parallelism in a computation?
  - Dependencies
- Different types of parallelism
  - data parallelism
  - task parallelism
- Parallel programming
  - Creating parallel programs
- Standard models of parallelism and parallel programs

Parallel Models 101

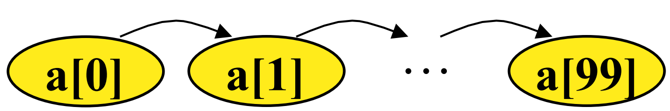


- Sequential models
  - von Neumann (RAM) model
- Parallel model
  - A parallel computer is simple a collection of *processors interconnected* in some manner to *coordinate* activities and *exchange data*
  - Models that can be used as general frameworks for describing and analyzing parallel algorithms
    - *Simplicity*: description, analysis, architecture independence
    - *Implementability*: able to be realized, reflect performance
- Three common parallel models
  - Directed acyclic graphs, shared-memory, network

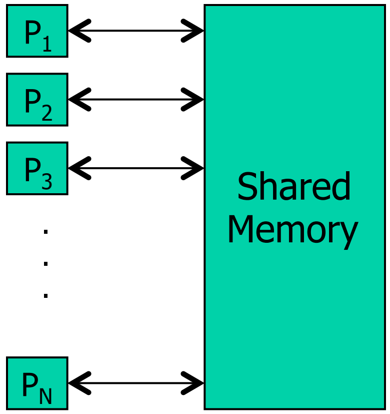
Directed Acyclic Graphs (DAG)

- 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
- Arcs are paths for flow of data results
- DAG represents the operations of the algorithm and implies precedent constraints on their order

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



Shared Memory Model



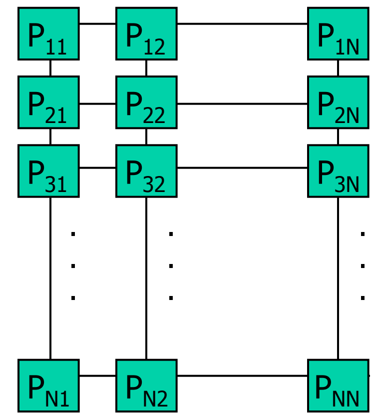
- Parallel extension of RAM model (PRAM)
  - Memory size is infinite
  - Number of processors in unbounded
  - Processors communicate via the memory
  - Every processor accesses any memory location in 1 cycle
  - Synchronous
    - All processors execute same algorithm synchronously
      - READ phase
      - COMPUTE phase
      - WRITE phase
    - Some subset of the processors can stay idle
  - Asynchronous

Memory Access in PRAM

- Exclusive Read (ER): p processors can simultaneously read the content of p distinct memory locations
- Concurrent Read (CR): p processors can simultaneously read the content of p' memory locations, where $p' < p$
- Exclusive Write (EW): p processors can simultaneously write the content of p distinct memory locations
- Concurrent Write (CW): p processors can simultaneously write the content of p' memory locations, where $p' < p$
- EREW and ERCW (weird)
- CREW and CRCW

Network Model

- G = (N,E)
  - N are processing nodes
  - E are bidirectional communication links
- Each processor has its own memory
- No shared memory is available
- Network operation may be synchronous or asynchronous
- Requires communication primitives
  - Send (X, i)
  - 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 or handling more work
- What is being parallelized?
  - Task: instruction, statement, procedure, …
  - Data: data flow, size, replication
  - Parallelism granularity
    - Coarse-grain versus fine-grainded
- Thinking about parallelism
- Evaluation

Why is parallel programming important today?

- Parallel programming has matured
  - Standard programming models
  - Common machine architectures
  - Programmer can focus on computation and use suitable programming model for implementation
- Increasing 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
  - Coordinate data accesses and work to satisfy dependencies
- 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
  - For example, due to where data is located

Executing Two Statements in Parallel

- Want to execute two statements in parallel
- On one processor:

  Statement 1;

  Statement 2;
- On two processors:

Processor 1:	Processor 2:
Statement 1;	Statement 2;

- Fundamental (*concurrent*) execution assumption
  - Processors execute independent of each other
  - No assumptions made about speed of processor execution

Sequential Consistency in Parallel Execution

- Case 1:

  Processor 1: Processor 2: time

  statement 1;

  statement 2;

- Case 2:

  Processor 1: Processor 2: time

  statement 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

  statement2;

  statement1;

- Their 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

  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 S1 and S2,

  S1;

  S2;

- S2 has a *true (flow) dependence* on S1

  if and only if

  S2 reads a value written by S1

$$\begin{array}{l} X = \\ \quad \vdots \qquad \delta \\ \quad = X \end{array}$$

- S2 has an *anti-dependence* on S1

  if and only if

  S2 writes a value read by S1

$$\begin{array}{l} \quad = X \\ \quad \vdots \qquad \delta^{-1} \\ X = \end{array}$$

Output Dependence

- Given statements S1 and S2,

  S1;

  S2;

- S2 has an *output dependence* on S1

  if and only if

  S2 writes a variable written by S1

$$\begin{array}{l} X = \\ \vdots \\ X = \end{array} \quad \delta^0$$

- 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

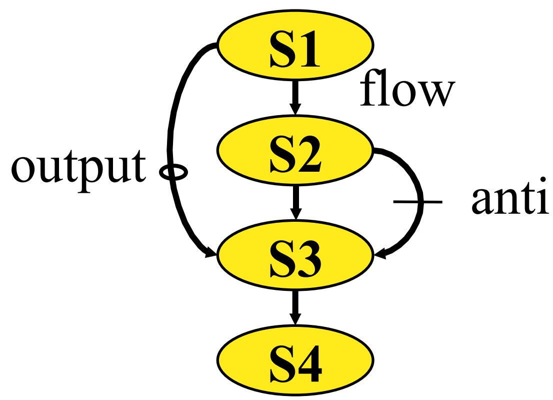
- Can use graphs to show dependence relationships
- Example

  S1: a=1;

  S2: b=a;

  S3: a=b+1;

  S4: 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 S1 and S2 can execute in parallel if and only if there are *no dependences* between S1 and S2
  - True dependences
  - Anti-dependences
  - Output dependences
- Some dependences can be remove by modifying the program
  - Rearranging statements
  - Eliminating statements

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 and OUT Sets and Computing Dependence

- Assuming that there is a path from S1 to S2 , the following shows how to intersect the IN and OUT sets to test for data dependence

$$out(S_1) \cap in(S_2) \neq \emptyset \qquad S_1\ \delta\ S_2 \qquad \text{flow dependence}$$

$$in(S_1) \cap out(S_2) \neq \emptyset \qquad S_1\ \delta^{-1}\ S_2 \qquad \text{anti-dependence}$$

$$out(S_1) \cap out(S_2) \neq \emptyset \qquad S_1\ \delta^{0} S_2 \qquad \text{output dependence}$$

Loop-Level Parallelism

- Significant parallelism can be identified <u>within</u> loops

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

```
for (i=0; i<100; i++) {
    S1: a[i] = i;
    S2: b[i] = 2*i;
}
```

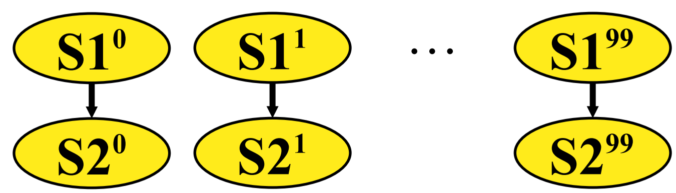
- Dependencies? What about *i*, the loop index?
- *DOALL* loop
  - All iterations are independent of each other
  - All statements be executed in parallel at the same time
    - Is this really true?

Iteration Space

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

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

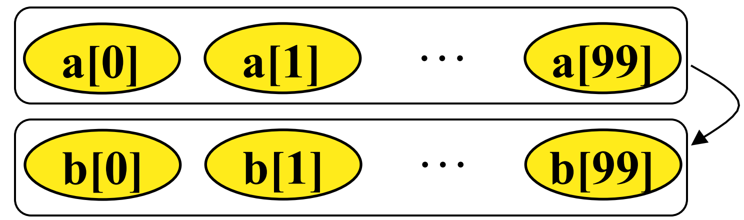
```
for (i=0; i<100; i++) {
    S1: a[i] = i;
    S2: b[i] = 2*i;
}
```



Multi-Loop Parallelism

- Significant parallelism can be identified <u>between</u> loops

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

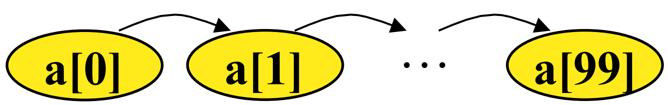


- Dependencies?
- How much parallelism is available?
- Given 4 processors, how much parallelism is possible?
- What parallelism is achievable with 50 processors?

Loops with Dependencies

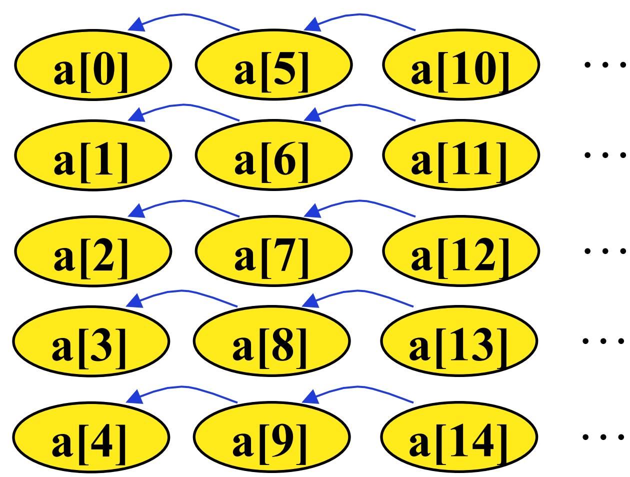
Case 1:

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



Case 2:

```
for (i=5; i<100; i++)
    a[i-5] = a[i] + 100;
```



- Dependencies?
  - What type?
- Is the Case 1 loop parallelizable?
- 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
  - Unroll the loop to see

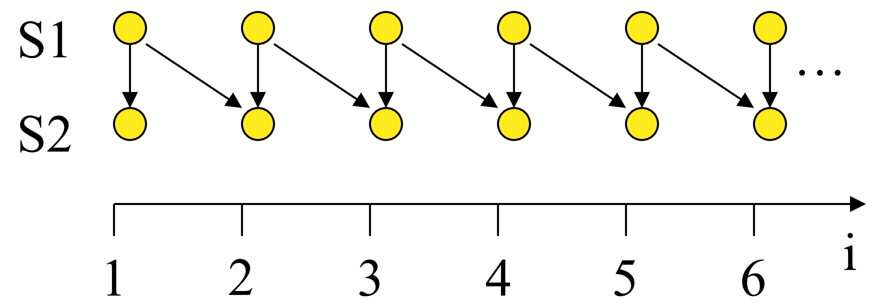
Loop Dependence Example

```
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?

Iteration Dependence and Pipelining

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



- Dependencies?
  - Between a[i] and a[i-1]
- Is parallelism possible?
  - Statements can be executed in pipelined parallel

Another Loop Dependence Example

```
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?)

Indirect Indexing and Dependences

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

- Dependencies?
  - Cannot tell for sure
- Parallelization depends on knowledge of index values
  - User may know
  - Compiler does not know
  - User could inform the compiler

Hidden Dependencies – Printing

```
printf("a");
printf("b");
```

- Statements have a hidden output dependence
  - Due to the serial output stream

Hidden Dependences – Functions

```
a = f(x);
b = g(x);
```

- Statements could have hidden dependence if f() and g() update the same variable through side effects

Parallelizing Compilers

- Parallelizing compilers analyze program dependences to decide parallelization
- In parallelization by hand, user does the same analysis.
- Compiler more convenient and more correct
- User more knowledgable
  - Can analyze more patterns

Key Ideas for Dependency Analysis

- To execute in parallel:
  - 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
  - Dependencies pose constraints on parallel event ordering
  - Partial ordering of execution action
- Use synchronization mechanisms
  - Need for concurrent execution too
  - Maintains partial order

Synchronization Primitives

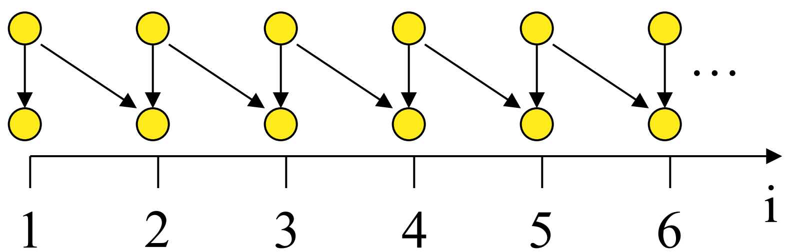
- Suppose we had a set of primitives, signal(x) and wait(x)
- wait(x) blocks unless a signal(x) has occurred.
- signal(x) does not block, but causes a wait(x) to unblock, or causes a future wait(x) not to block

```
f() {
  a=1; b=2; c=3;
}
g() {
  d=4; e=5; a=6;
}
main() { f(); g(); }
```

```
f() {
  a=1; signal(e_a); b=2; c=3;
}
g() {
d=4; e=5; wait(e_a); a=6;
}
main() { f(); g(); }
```

Synchronization in Loops

```
for (i=0; i<100; i++) {
    a[i] = …;
    …;
    … = a[i-1];
}
```



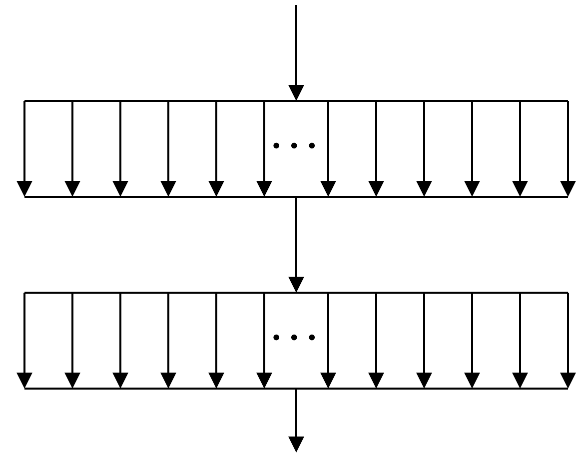
⇒

```
for (i=0; i<100; i++) {
    a[i] = …;
    signal(e_a[i]);
    …;
    wait(e_a[i-1]);
    … = a[i-1];
}
```

- Loop cannot be parallelized unless have synchronization!
- Does it matters which processors get which iterations?
- This is called a *DOACROSS* loop
- How could you parallelize this without synchronization?

Fork-Join Parallelism

```
x = g(a);
for( i=0; i<100; i++ ) a[i] = f(i);
y = h(a);
for( i=0; i<100; i++ ) b[i] = x + h( a[i]);
```



- First loop is a DOALL loop
- Middle statement is sequential
- Second loop is a DOALL loop
- Execution moves between sequential and parallel phases
- Call this *fork-join* parallelism
- Fork-join, loop-level parallelism is basis for OpenMP

Fork-Join and Barrier Synchronization

- fork() causes a number of processes to be created and to be run in parallel
- join() causes all these processes to wait until all of them have executed a join() (*barrier* synchronization)

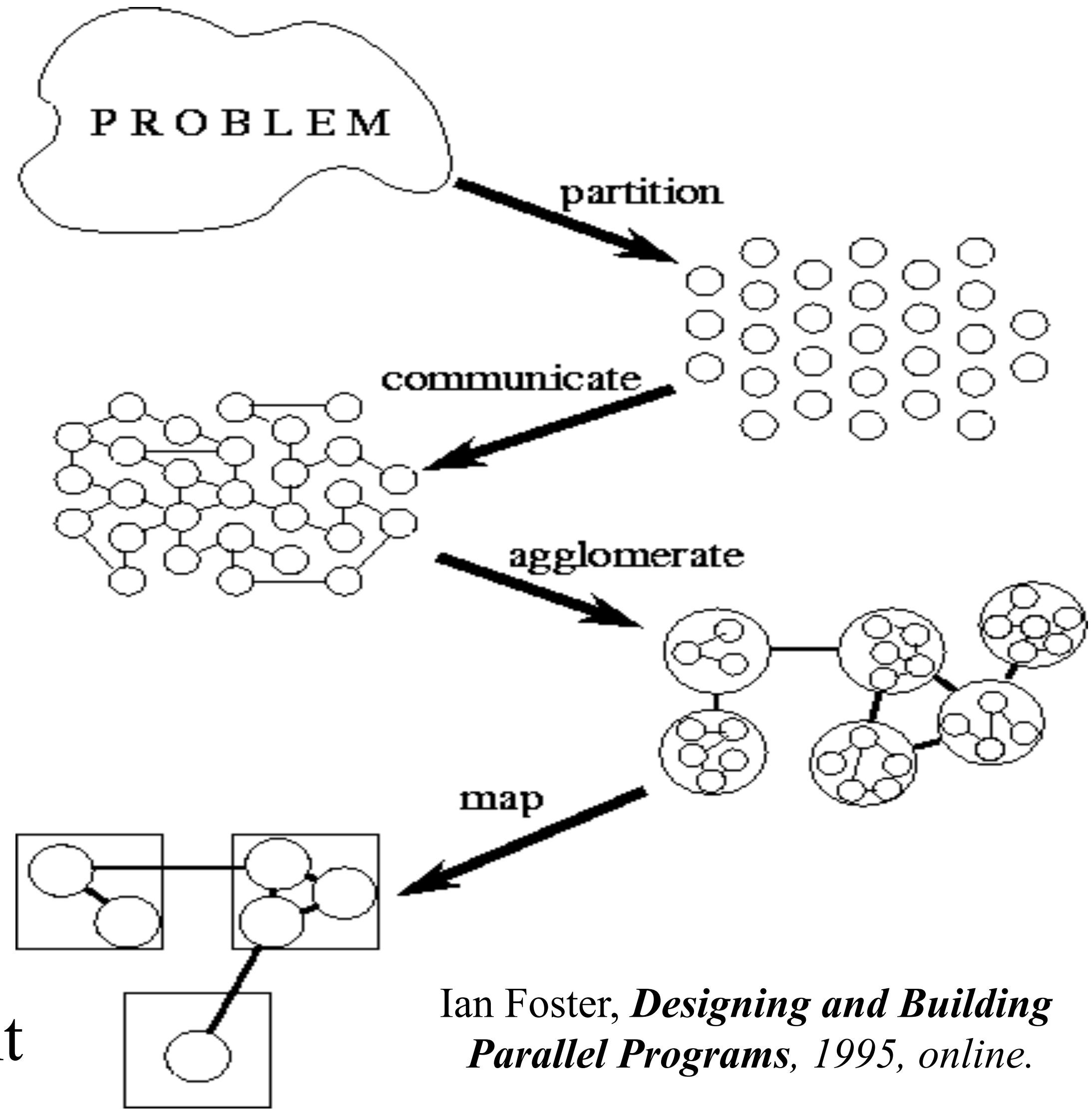
```
fork();
for( i=0; i<100; i++ ) a[i] = f(i);
join();
y = h(a);
fork();
for( i=0; i<100; i++ ) b[i] = x + h(a[i]);
join();
```

Synchronization Issues

- Synchronization is necessary to make some programs execute correctly in parallel
- Dependences have to be "covered" by appropriate synchronization operations
- Different sychronization constructs exist in different parallel programming models
- However, synchronization is expensive
- To reduce synchronization
  - May need to limit parallelization
  - Look for opportunities to increase parallelism granularity

Methodological Design

- Partition:
  - Task/data decomposition
- Communication
  - Task execution coordination
- Agglomeration
  - Evaluation of the structure
- Mapping
  - Resource assignment



Ian Foster, ***Designing and Building Parallel Programs****, 1995, online.*

Next Class

- Parallel programming models
- Introduction to HPC Linux and LiveDVD