Parallel Programming Patterns
Overview and Map Pattern

Parallel Computing
CIS 410/510
Department of Computer and Information Science
Outline

- Parallel programming models
- Dependencies
- Structured programming patterns overview
  - Serial / parallel control flow patterns
  - Serial / parallel data management patterns
- Map pattern
  - Optimizations
    - sequences of Maps
    - code Fusion
    - cache Fusion
  - Related Patterns
  - Example: Scaled Vector Addition (SAXPY)
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
- 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 is unbounded
  - Processors communicate via the memory
  - Every processor accesses any memory location in 1 cycle
  - Synchronous
    - All processors execute the same algorithm synchronously
      - READ phase
      - COMPUTE phase
      - WRITE phase
    - Some subset of the processors can stay idle
  - Asynchronous
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-grained
- Thinking about parallelism
- 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
  - 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:  
  
  Statement 1;

  Processor 2:  

  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: 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
  
  ```
  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;

- **Example 2**
  S1: a=1;
  S2: b=a;

- **Example 3**
  S1: a=f(x);
  S2: a=b;

- **Example 4**
  S1: a=b;
  S2: b=1;

  - Statements are independent
  - Dependent (*true (flow) dependence*)
    - Second is dependent on first
    - Can you remove dependency?
  - Dependent (*output dependence*)
    - Second is dependent on first
    - Can you remove dependency? How?
  - 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

- S2 has a **anti-dependence** on S1 if and only if
  
  S2 writes a value read by S1
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
  
  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 removed 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:
  - $\text{IN}(S)$: set of memory locations (variables) that may be used in $S$
  - $\text{OUT}(S)$: 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 flow dependence
- $\text{in}(S_1) \cap \text{out}(S_2) \neq \emptyset$ \quad $S_1 \delta^{-1} S_2$ \quad anti-dependence
- $\text{out}(S_1) \cap \text{out}(S_2) \neq \emptyset$ \quad $S_1 \delta^0 S_2$ \quad output dependence
Loop-Level Parallelism

- Significant parallelism can be identified within loops

```plaintext
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** (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?
Iteration Space

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

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

```c
for (i=0; i<100; i++) {
    S1: a[i] = i;
    S2: 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?
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**

```c
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?
Dependences Between Iterations

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 “pipeline” manner
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?)
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 (cores)
  - Dependencies pose constraints on parallel event ordering
  - Partial ordering of execution action

- Use synchronization mechanisms
  - Need for concurrent execution too
  - Maintains partial order
Parallel Patterns

- **Parallel Patterns**: A recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design.
- Patterns provide us with a “vocabulary” for algorithm design
- It can be useful to compare parallel patterns with serial patterns
- Patterns are universal – they can be used in *any* parallel programming system
Parallel Patterns

- Nesting Pattern
- Serial / Parallel Control Patterns
- Serial / Parallel Data Management Patterns
- Other Patterns
- Programming Model Support for Patterns
Nesting Pattern

- **Nesting** is the ability to hierarchically compose patterns.
- This pattern appears in both serial and parallel algorithms.
- “Pattern diagrams” are used to visually show the pattern idea where each “task block” is a location of general code in an algorithm.
- Each “task block” can in turn be another pattern in the nesting pattern.
Nesting Pattern: A compositional pattern. Nesting allows other patterns to be composed in a hierarchy so that any task block in the above diagram can be replaced with a pattern with the same input/output and dependencies.
Serial Control Patterns

- Structured serial programming is based on these patterns: sequence, selection, iteration, and recursion

- The nesting pattern can also be used to hierarchically compose these four patterns

- Though you should be familiar with these, it’s extra important to understand these patterns when parallelizing serial algorithms based on these patterns
Serial Control Patterns: Sequence

- **Sequence**: Ordered list of tasks that are executed in a specific order

- Assumption – program text ordering will be followed (obvious, but this will be important when parallelized)

```
1. T = f(A);
2. S = g(T);
3. B = h(S);

1. T = f(A);
2. S = g(A);
3. B = h(S,T);
```
Serial Control Patterns: Selection

- **Selection**: condition $c$ is first evaluated. Either task $a$ or $b$ is executed depending on the true or false result of $c$.

- Assumptions – $a$ and $b$ are never executed before $c$, and only $a$ or $b$ is executed - never both

```java
1 if (c) {
2   a;
3 } else {
4     b;
5 }
```

Diagram:
```
        T
          ↓
        c
          ↓
        F
```

```
  a
```

```
  b
```
Serial Control Patterns: Iteration

- **Iteration**: a condition $c$ is evaluated. If true, $a$ is evaluated, and then $c$ is evaluated again. This repeats until $c$ is false.

- Complication when parallelizing: potential for dependencies to exist between previous iterations

```plaintext
for (i = 0; i < n; )
  a;
}

while (c) {
  a;
}
```
Serial Control Patterns: Recursion

- **Recursion**: dynamic form of nesting allowing functions to call themselves
- Tail recursion is a special recursion that can be converted into iteration – important for functional languages
Parallel Control Patterns

- Parallel control patterns extend serial control patterns
- Each parallel control pattern is related to at least one serial control pattern, but relaxes assumptions of serial control patterns
- Parallel control patterns: fork-join, map, stencil, reduction, scan, recurrence
Parallel Control Patterns: Fork-Join

- **Fork-join**: allows control flow to fork into multiple parallel flows, then rejoin later
- **Cilk Plus** implements this with **spawn** and **sync**
  - The call tree is a parallel call tree and functions are spawned instead of called
  - Functions that spawn another function call will continue to execute
  - Caller *syncs* with the spawned function to join the two
- A “join” is different than a “barrier”
  - Sync – only one thread continues
  - Barrier – all threads continue
Parallel Control Patterns: Map

- **Map**: performs a function over every element of a collection
- Map replicates a serial iteration pattern where each iteration is independent of the others, the number of iterations is known in advance, and computation only depends on the iteration count and data from the input collection
- The replicated function is referred to as an “elemental function”
Parallel Control Patterns: Stencil

- **Stencil**: Elemental function accesses a set of “neighbors”, stencil is a generalization of map
- Often combined with iteration – used with iterative solvers or to evolve a system through time
- Boundary conditions must be handled carefully in the stencil pattern
- See stencil lecture…
Parallel Control Patterns: Reduction

- **Reduction**: Combines every element in a collection using an associative “combiner function”
- Because of the associativity of the combiner function, different orderings of the reduction are possible
- Examples of combiner functions: addition, multiplication, maximum, minimum, and Boolean AND, OR, and XOR
Parallel Control Patterns: Reduction

Serial Reduction

Parallel Reduction
Parallel Control Patterns: Scan

- **Scan**: computes all partial reduction of a collection
- For every output in a collection, a reduction of the input up to that point is computed
- If the function being used is associative, the scan can be parallelized
- Parallelizing a scan is not obvious at first, because of dependencies to previous iterations in the serial loop
- A parallel scan will require more operations than a serial version
Parallel Control Patterns: Scan

Serial Scan

Parallel Scan
Parallel Control Patterns: Recurrence

- **Recurrence**: More complex version of map, where the loop iterations can depend on one another.
- Similar to map, but elements can use outputs of adjacent elements as inputs.
- For a recurrence to be computable, there must be a serial ordering of the recurrence elements so that elements can be computed using previously computed outputs.
Serial Data Management Patterns

- Serial programs can manage data in many ways
- Data management deals with how data is allocated, shared, read, written, and copied
- Serial Data Management Patterns: random read and write, stack allocation, heap allocation, objects
Serial Data Management Patterns: random read and write

- Memory locations indexed with addresses
- Pointers are typically used to refer to memory addresses
- Aliasing (uncertainty of two pointers referring to the same object) can cause problems when serial code is parallelized
Serial Data Management Patterns: Stack Allocation

- Stack allocation is useful for dynamically allocating data in LIFO manner.
- Efficient – arbitrary amount of data can be allocated in constant time.
- Stack allocation also preserves locality.
- When parallelized, typically each thread will get its own stack so thread locality is preserved.
Serial Data Management Patterns: Heap Allocation

- Heap allocation is useful when data cannot be allocated in a LIFO fashion
- But, heap allocation is slower and more complex than stack allocation
- A parallelized heap allocator should be used when dynamically allocating memory in parallel
  - This type of allocator will keep separate pools for each parallel worker
Serial Data Management Patterns: Objects

- Objects are language constructs to associate data with code to manipulate and manage that data
- Objects can have member functions, and they also are considered members of a class of objects
- Parallel programming models will generalize objects in various ways
Parallel Data Management Patterns

- To avoid things like race conditions, it is critically important to know when data is, and isn’t, potentially shared by multiple parallel workers
- Some parallel data management patterns help us with data locality
- Parallel data management patterns: **pack, pipeline, geometric decomposition, gather, and scatter**
Parallel Data Management Patterns: Pack

- **Pack** is used to eliminate unused space in a collection.
- Elements marked *false* are discarded, the remaining elements are placed in a contiguous sequence in the same order.
- Useful when used with map.
- **Unpack** is the inverse and is used to place elements back in their original locations.

![Diagram of Pack and Unpack operations]
Parallel Data Management Patterns: Pipeline

- **Pipeline** connects tasks in a producer-consumer manner.
- A linear pipeline is the basic pattern idea, but a pipeline in a DAG is also possible.
- Pipelines are most useful when used with other patterns as they can multiply available parallelism.
Parallel Data Management Patterns: Geometric Decomposition

- **Geometric Decomposition** – arranges data into subcollections
- Overlapping and non-overlapping decompositions are possible
- This pattern doesn’t necessarily move data, it just gives us another view of it
Parallel Data Management Patterns: Gather

- **Gather** reads a collection of data given a collection of indices
- Think of a combination of map and random serial reads
- The output collection shares the same type as the input collection, but it shares the same shape as the indices collection
Parallel Data Management Patterns: Scatter

- **Scatter** is the inverse of gather
- A set of input and indices is required, but each element of the input is written to the output at the given index instead of read from the input at the given index
- Race conditions can occur when we have two writes to the same location!
Other Parallel Patterns

- **Superscalar Sequences:** write a sequence of tasks, ordered only by dependencies
- **Futures:** similar to fork-join, but tasks do not need to be nested hierarchically
- **Speculative Selection:** general version of serial selection where the condition and both outcomes can all run in parallel
- **Workpile:** general map pattern where each instance of elemental function can generate more instances, adding to the “pile” of work
Other Parallel Patterns

- **Search**: finds some data in a collection that meets some criteria
- **Segmentation**: operations on subdivided, non-overlapping, non-uniformly sized partitions of 1D collections
- **Expand**: a combination of pack and map
- **Category Reduction**: Given a collection of elements each with a label, find all elements with same label and reduce them
# Programming Model Support for Patterns

Table 3.1 Summary of programming model support for the serial patterns discussed in this book. Note that some of the parallel programming models we consider do not, in fact, support all the common serial programming patterns. In particular, note that recursion and memory allocation are limited on some model.

<table>
<thead>
<tr>
<th>Serial Pattern</th>
<th>TBB</th>
<th>Cilk Plus</th>
<th>OpenMP</th>
<th>ArBB</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Serial) Nesting</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Sequence</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Selection</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Iteration</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Recursion</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>?</td>
</tr>
<tr>
<td>Random Read</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Random Write</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Stack Allocation</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Heap Allocation</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Closures</td>
<td>F</td>
<td>F</td>
<td>F(w/C++)</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>Objects</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Programming Model Support for Patterns

**Table 3.2** Summary of programming model support for the patterns discussed in this book. F: Supported directly, with a special feature. I: Can be implemented easily and efficiently using other features. P: Implementations of one pattern in terms of others, listed under the pattern being implemented. Blank means the particular pattern cannot be implemented in that programming model (or that an efficient implementation cannot be implemented easily). When examples exist in this book of a particular pattern with a particular model, section references are given.

<table>
<thead>
<tr>
<th>Parallel Pattern</th>
<th>TBB</th>
<th>Cilk Plus</th>
<th>OpenMP</th>
<th>ArBB</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel nesting</td>
<td>F</td>
<td>F</td>
<td>F 4.2.6;</td>
<td>F 4.2.9;</td>
<td></td>
</tr>
<tr>
<td>Map</td>
<td>F 4.2.3;</td>
<td>4.2.4;4.2.5;</td>
<td>4.3.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.3.3</td>
<td>4.3.4;4.3.5</td>
<td>4.3.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>11</td>
<td>4.3.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stencil</td>
<td>I 10</td>
<td>I 10</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workpile</td>
<td>F</td>
<td>F 10</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reduction</td>
<td>F 5.3.4;</td>
<td>F 5.3.5;</td>
<td>F 5.3.6;</td>
<td>F 5.3.7;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>11</td>
<td>5.3.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scan</td>
<td>F 5.6.5;</td>
<td>I 5.6.3;</td>
<td>I 5.6.4;</td>
<td>I 5.6.6;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>P 8.11;</td>
<td>5.6.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>P 5.4.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fork-join</td>
<td>F 8.9.2;</td>
<td>F 8.7;</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>8.9.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recurrence</td>
<td>I 14</td>
<td>I 14</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Superscalar sequence</td>
<td>I 15</td>
<td>I 15</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Futures</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speculative selection</td>
<td></td>
<td></td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pack</td>
<td>I 14</td>
<td>I 14</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expand</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipeline</td>
<td>F 12</td>
<td>I 12</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geometric decomposition</td>
<td>I 15</td>
<td>I 15</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Category reduction</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gather</td>
<td>I</td>
<td>F</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic scatter</td>
<td>F</td>
<td>I</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Permutation scatter</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge scatter</td>
<td>I</td>
<td>I</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Priority scatter</td>
<td>I</td>
<td>I</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Programming Model Support for Patterns

<table>
<thead>
<tr>
<th>Parallel Pattern</th>
<th>TBB</th>
<th>Cilk Plus</th>
<th>OpenMP</th>
<th>ArBB</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar sequence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Futures</td>
<td>I</td>
<td></td>
<td></td>
<td>I</td>
<td>F</td>
</tr>
<tr>
<td>Speculative selection</td>
<td>I</td>
<td></td>
<td></td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Workpile</td>
<td>F</td>
<td></td>
<td></td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Expand</td>
<td>I</td>
<td></td>
<td></td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Search</td>
<td>I</td>
<td></td>
<td></td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Category reduction</td>
<td>I</td>
<td></td>
<td></td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Atomic scatter</td>
<td>F</td>
<td></td>
<td></td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Permutation scatter</td>
<td>F</td>
<td>F</td>
<td></td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Merge scatter</td>
<td>I</td>
<td></td>
<td></td>
<td>I</td>
<td>F</td>
</tr>
<tr>
<td>Priority scatter</td>
<td></td>
<td></td>
<td></td>
<td>I</td>
<td></td>
</tr>
</tbody>
</table>
Map Pattern - Overview

- Map
- Optimizations
  - Sequences of Maps
  - Code Fusion
  - Cache Fusion
- Related Patterns
- Example Implementation: Scaled Vector Addition (SAXPY)
  - Problem Description
  - Various Implementations
Mapping

- “Do the same thing many times”
  ```python
  foreach i in foo:
    do something
  ```
- Well-known higher order function in languages like ML, Haskell, Scala
  ```
  \[
  \text{map} : \forall a b. (a \rightarrow b) \text{List}<a> \rightarrow \text{List}<b>
  \]
  applies a function each element in a list and returns a list of results
**Example Maps**

Add 1 to every item in an array

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>5</th>
<th>3</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Double every item in an array

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>7</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>14</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

**Key Point:** An operation is a map if it can be applied to each element without knowledge of neighbors.
Key Idea

- Map is a “foreach loop” where each iteration is independent.

Embarrassingly Parallel

Independence is a big win. We can run map completely in parallel. Significant speedups! More precisely: $T(\infty)$ is $O(1)$ plus implementation overhead that is $O(\log n)$…so $T(\infty) \in O(\log n)$.
### Sequential Map

```java
for(int n=0; n< array.length; ++n){
    process(array[n]);
}
```
Parallel Map

parallel_for_each(
    x in array) {
    process(x);
}
Comparing Maps

Serial Map

Parallel Map

Data → Task → Data → Task → Data → Task → Data

Data → Task → Data → Task → Data → Task → Data

Task → Data → Task → Data → Task → Data → Task → Data

Task → Data → Task → Data → Task → Data → Task → Data
Comparing Maps

Serial Map

Parallel Map

Speedup

The space here is speedup. With the parallel map, our program finished execution early, while the serial map is still running.
Independence

- The key to (embarrassing) parallelism is independence

**Warning:** No shared state!

Map function should be “pure” (or “pure-ish”) and should not modify shared states

- Modifying shared state breaks perfect independence

- Results of accidentally violating independence:
  - non-determinism
  - data-races
  - undefined behavior
  - segfaults
Implementation and API

- OpenMP and CilkPlus contain a parallel `for` language construct
- Map is a mode of use of parallel `for`
- TBB uses `higher order functions` with lambda expressions/“funtors”
- Some languages (CilkPlus, Matlab, Fortran) provide array notation which makes some maps more concise

Array Notation

```c
A[:] = A[:]*5;
```

is CilkPlus array notation for “multiply every element in A by 5”
Unary Maps

So far we have only dealt with mapping over a single collection…
## Map with 1 Input, 1 Output

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>result</td>
<td>6</td>
<td>14</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>10</td>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

```c
int oneToOne ( int x[11] ) {
  return x*2;
}
```
N-ary Maps

But, sometimes it makes sense to map over multiple collections at once…
Map with 2 Inputs, 1 Output

```
int twoToOne ( int x[11], int y[11] ) {  
  return x+y;  
}
```
Often several map operations occur in sequence

- Vector math consists of many small operations such as additions and multiplications applied as maps

- A naïve implementation may write each intermediate result to memory, wasting memory BW and likely overwhelming the cache

**Optimization – Sequences of Maps**

- If we fuse the operations used in a sequence of maps into a sequence inside a single map, we can load only the input data at the start of the map and keep intermediate results in registers rather than wasting memory bandwidth on them. We will call this approach **code fusion**, and it can be applied to other patterns as well. Code fusion is demonstrated in Figure 4.2.
Optimization – Code Fusion

- Can sometimes “fuse” together the operations to perform them at once
- Adds arithmetic intensity, reduces memory/cache usage
- Ideally, operations can be performed using registers alone

A sequence of map operations over collections of the same shape should be combined whenever possible into a single larger operation. In particular, vector operations are really map operations using very simple operations like addition and multiplication. Implementing these one by one, writing to and from memory, would be inefficient, since it would have low arithmetic intensity. If this organization was implemented literally, data would have to be read and written for each operation, and we would consume memory bandwidth unnecessarily for intermediate results. Even worse, if the maps were big enough, we might exceed the size of the cache and so each map operation would go directly to and from main memory.

If we fuse the operations used in a sequence of maps into a sequence inside a single map, we can load only the input data at the start of the map and keep intermediate results in registers rather than wasting memory bandwidth on them. We will call this approach code fusion, and it can be applied to other patterns as well. Code fusion is demonstrated in Figure 4.2.

**Figure 4.2**

Code fusion optimization: Convert a sequence of maps into a map of sequences, avoiding the need to write intermediate results to memory. This can be done automatically by ArBB and explicitly in other programming models.
**Optimization – Cache Fusion**

- Sometimes impractical to fuse together the map operations
- Can instead break the work into blocks, giving each CPU one block at a time
- Hopefully, operations use cache alone
Related Patterns

Three patterns related to map are discussed here:

- Stencil
- Workpile
- Divide-and-Conquer

More detail presented in a later lecture
**Stencil**

- Each instance of the map function accesses neighbors of its input, offset from its usual input
- Common in imaging and PDE solvers
Workpile

- Work items can be added to the map while it is in progress, from inside map function instances
- Work grows and is consumed by the map
- Workpile pattern terminates when no more work is available
Divide-and-Conquer

- Applies if a problem can be divided into smaller subproblems recursively until a base case is reached that can be solved serially.
Example: Scaled Vector Addition (SAXPY)

- \( y \leftarrow ax + y \)
  - Scales vector \( x \) by \( a \) and adds it to vector \( y \)
  - Result is stored in input vector \( y \)

- Comes from the BLAS (Basic Linear Algebra Subprograms) library

- Every element in vector \( x \) and vector \( y \) are independent
What does $y \leftarrow ax + y$ look like?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td>9</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>11</td>
<td>23</td>
<td>8</td>
<td>5</td>
<td>36</td>
<td>12</td>
<td>36</td>
<td>49</td>
<td>50</td>
<td>7</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>
Visual: $y \leftarrow ax + y$

Twelve processors used $\rightarrow$ one for each element in the vector
**Visual:** \( y \leftarrow ax + y \)

Six processors used \( \rightarrow \) one for every two elements in the vector
**Visual**: \( y \leftarrow ax + y \)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td>9</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>11</td>
<td>23</td>
<td>8</td>
<td>5</td>
<td>36</td>
<td>12</td>
<td>36</td>
<td>49</td>
<td>50</td>
<td>7</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

Two processors used → one for every six elements in the vector
Serial SAXPY Implementation

```c
void saxpy_serial(
    size_t n,       // the number of elements in the vectors
    float a,        // scale factor
    const float x[], // the first input vector
    float y[]       // the output vector and second input vector
) {
    for (size_t i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

TBB uses thread parallelism but does not, by itself, vectorize the code. It depends on the underlying C++ compiler to do that. On the other hand, tiling does expose opportunities for vectorization, so if the basic serial algorithm can be vectorized then typically the TBB code can be, too. Generally, the serial SAXPY implementation is a good starting point for understanding how to parallelize a program.
**TBB SAXPY Implementation**

```c
void saxpy_tbb(
    int n,       // the number of elements in the vectors
    float a,     // scale factor
    float x[],   // the first input vector
    float y[]    // the output vector and second input vector
) {
    tbb::parallel_for(
        tbb::blocked_range<int>(0, n),
        [&](tbb::blocked_range<int> r) {
            for (size_t i = r.begin(); i != r.end(); ++i)
                y[i] = a * x[i] + y[i];
        }
    );
}
```
Cilk Plus SAXPY Implementation

```c
void saxpy_cilk(
    int n,    // the number of elements in the vectors
    float a,  // scale factor
    float x[], // the first input vector
    float y[]   // the output vector and second input vector
) {
    cilk_for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

4.2.4 Cilk Plus

A basic Cilk Plus implementation of the SAXPY operation is given in Listing 4.3. The "parallel for" syntax approach is used here, as with TBB, although the syntax is closer to a regular for loop. In fact, an ordinary for loop can often be converted to a cilk_for construct if all iterations of the loop body are independent—that is, if it is a map. As with TBB, the cilk_for is not explicitly vectorized but the compiler may attempt to auto-vectorize. There are restrictions on the form of a cilk_for loop. See Appendix B.5 for details.

4.2.5 Cilk Plus with Array Notation

It is also possible in Cilk Plus to explicitly specify vector operations using Cilk Plus array notation, as in Listing 4.4. Here x[0:n] and y[0:n] refer to n consecutive elements of each array, starting with x[0] and y[0]. A variant syntax allows specification of a stride between elements, using x[start:length:stride]. Sections of the same length can be combined with operators. Note that there is no cilk_for in Listing 4.4.

```c
void saxpy_array_notation(
    int n,    // the number of elements in the vectors
    float a,  // scale factor
    float x[], // the input vector
    float y[]   // the output vector and offset
) {
    y[0:n] = a * x[0:n] + y[0:n];
}
```

Listing 4.3 SAXPY in Cilk Plus using cilk_for.

Listing 4.4 SAXPY in Cilk Plus using cilk_for and array notation for explicit vectorization.
**OpenMP SAXPY Implementation**

```c
void saxpy_openmp(
    int n,          // the number of elements in the vectors
    float a,        // scale factor
    float x[],      // the first input vector
    float y[]       // the output vector and second input vector
) {

    #pragma omp parallel for
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
```
OpenMP SAXPY Performance

Vector size = 500,000,000