Logistics

- Homework 1
  - Due 12:00 today
- ispc programming assignment
  - Due midnight today
- TBB programming assignment
  - Due February 8, 2018 midnight
- Piazza
  - [piazza.com/uoregon/winter2018/cis431531/home](http://piazza.com/uoregon/winter2018/cis431531/home)
  - Please use fellow classmates as additional resources for asking programming questions and getting help
- GitLab repository for assignment submission
- Programming lab this afternoon
  - Introduction to OpenMP (25 minutes) then assignment help
Outline

- Structured programming patterns overview
  - Serial / parallel control flow patterns
  - Serial / parallel data management patterns
- Map pattern
Parallel Patterns

- A parallel pattern is 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 Overview

- 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: It is a compositional pattern that allows other patterns to be composed in a hierarchy. 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 **serial control patterns** that order statement execution
  - *Imperative* programming changes program state and controls the flow of instruction execution
  - Sequence
  - Selection
  - Iteration
  - Recursion

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

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

- Ordered list of tasks that are executed in a specific order constitute a **sequence** control pattern

- 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);
```

Assume executing `f()` and `g()` is sequentially consistent
Serial Control Patterns: Selection

- A selection control pattern evaluates a condition $c$ first, and then executes task $a$ or $b$ 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

```
1   if (c) {
2     a;
3   } else {
4     b;
5   }
```
Serial Control Patterns: Iteration

- With the **iteration** control pattern …
  - … 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 between iterations

```plaintext
1 for (i=0; i<n; i++) {
  1 while (c) {
    2 a
    3 }
  2 a
  3 }
```
Serial Control Patterns: Recursion

- The **recursion** control pattern is a dynamic form of nesting allowing functions to call themselves.
- Tail recursion is a special recursion that can be converted into iteration.
  - Particularly important for functional languages.

```c++
void print(int n) {
    if (n < 0) return;
    cout << " " << n;
    // last executed statement is recursive call
    print(n-1);
}
```

```c++
void print(int n) {
    while (n >= 0) {
        cout << " " << n;
        n--;
    }
}
```
Parallel Control Patterns

- Parallel control patterns extend serial control patterns
- Each parallel control pattern is related to at least one serial control pattern
  - 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`
  - Calling 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
  - Replicates a serial iteration pattern
  - Each iteration is independent of the others
  - Number of iterations is known in advance
  - Computation only depends on iteration count and data from input collection

- The replicated function is referred to as an “elemental function”
Parallel Control Patterns: Stencil

- **Stencil**: elemental function accesses “neighbors”
  - A stencil is a generalization of map
- Often combined with iteration
  - Used with iterative solvers
  - Evolve a simulation through time (e.g., heat diffusion)
- 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:
  - Arithmetic (add, multiply, max, min)
  - Boolean (AND, OR, 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:
  - Scan can be parallelized.
- Parallelizing a scan is not obvious at first:
  - 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: Read/Write

- Random memory addressing
- 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.
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.
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.
(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 share the same shape as the indices collection.

![Diagram showing the Gather pattern](image)
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></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>F</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</td>
<td>F</td>
<td>F 4.2.9;</td>
</tr>
<tr>
<td>Map</td>
<td>F 4.2.3;</td>
<td>4.2.4;4.2.5;</td>
<td>4.2.6;</td>
<td>4.2.7;</td>
<td>4.2.8;</td>
</tr>
<tr>
<td></td>
<td>4.3.3;</td>
<td>4.3.4;4.3.5;</td>
<td>4.3.6;</td>
<td>4.3.7;</td>
<td>4.3.8;</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stencil</td>
<td>F 10</td>
<td>F 10</td>
<td>I</td>
<td>F 10</td>
<td>I</td>
</tr>
<tr>
<td>Workpile</td>
<td>F</td>
<td></td>
<td></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>I</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scan</td>
<td>F 5.6.5;</td>
<td>F 5.6.3;</td>
<td>F 5.6.4;</td>
<td>F 5.6.6;</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td></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>Recurrence</td>
<td>F 8.12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Superscalar sequence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Futures</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speculative selection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>Pack</td>
<td>F 14</td>
<td>F 14</td>
<td>I</td>
<td>F</td>
<td>I</td>
</tr>
<tr>
<td>Expand</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>Pipeline</td>
<td>F 12</td>
<td>F 12</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Geometric decomposition</td>
<td>F 15</td>
<td>F 15</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Search</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Category reduction</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Gather</td>
<td>I</td>
<td>F</td>
<td>I</td>
<td>F</td>
<td>I</td>
</tr>
<tr>
<td>Atomic scatter</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Permutation scatter</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Merge scatter</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>F</td>
<td>I</td>
</tr>
<tr>
<td>Priority scatter</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>F</td>
<td>I</td>
</tr>
</tbody>
</table>
## Programming Model Support for Patterns

**Table 3.3** Additional patterns discussed. **F:** Supported directly, with a special feature. **I:** Can be implemented easily and efficiently using other features. Blank means the particular pattern cannot be implemented in that programming model (or that an efficient implementation cannot be implemented easily).

<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>I</td>
<td>I</td>
<td>I</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>Futures</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>Speculative selection</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workpile</td>
<td>F</td>
<td>I</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expand</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Search</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Category reduction</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Atomic scatter</td>
<td>F</td>
<td>I</td>
<td>I</td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>Permutation scatter</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Merge scatter</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>F</td>
<td>I</td>
</tr>
<tr>
<td>Priority scatter</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>F</td>
<td>I</td>
</tr>
</tbody>
</table>