CIS 31/531
Parallel Computing
Map and Collective Patterns

Department of Computer and Information Science
Spring 2017

UNIVERSITY OF OREGON
Outline

- Map pattern
  - Optimizations
    - sequences of Maps
    - code Fusion
    - cache Fusion
  - Related Patterns
  - Example: Scaled Vector Addition (SAXPY)

- Collectives pattern
  - Reduce Pattern
  - Scan Pattern
  - Example: Sorting
Map Pattern - Overview

- What is map(ping)?
- 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

  $$ \text{map} : \forall ab.(a \rightarrow b)\text{List}<a> \rightarrow \text{List}<b> $$

  applies a function to each element in a list and returns a list of results
Example Maps

Add 1 to every item in an array

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

+1

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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

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

*2

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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 (in a collection) without knowledge of neighbors. (Well, not exactly. It is more a case of independence. We come to this later.)
**Key Idea**

- Map is a “foreach” loop (a.k.a. “doall” loop) where each iteration is independent.
- These are embarrassingly parallel!
Sequential Map

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

process each iteration in sequence
Parallel Map

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

CIS 431/531: Parallel Computing, University of Oregon
Comparing Maps

Serial Map

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

Parallel Map

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

Serial Map

- Task
- Data
- Task
- Data
- Task
- Data
- Task
- Data
- Task
- Data

Parallel Map

- Task
- Data
- Task
- Data
- Task
- Data
- Task
- Data
- Task
- Data

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 (lead to violation of sequential consistency)
  - Undefined behavior
  - Segfaults
Implementation and API

- OpenMP and CilkPlus contain a parallel \textit{for} language construct
  - OpenMP has a version for Fortran and C/C+
- Map is a mode of use of parallel \textit{for}
- TBB uses \textbf{higher order functions} with lambda expressions and “functors”
- Some languages (CilkPlus, Matlab, Fortran) provide array notation which makes some maps more concise

Array Notation

\begin{verbatim}
A[:] = A[:]*5;
\end{verbatim}

is CilkPlus array notation for “multiply every element in \textit{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>*2</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

<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>y</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>
</tbody>
</table>

result

5  11  2  2  12  3  9  9  10  4  3  1

```cpp
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 – 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.
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.

These diagrams show only the simplest case, where the sections of the partition fit exactly into the domain. In practice, there may be boundary conditions where partial sections are required along the edges. These may need to be treated with special-purpose code, but even in this case the majority of the sections will be regular, which lends itself to vectorization. Ideally, to get good memory behavior and to allow efficient vectorization, we also normally want to partition data, especially for writes, so that it aligns with cache line and vectorization boundaries. You should be aware of how data is actually laid out in memory when partitioning data. For example, in a multidimensional partitioning, typically only one dimension of an array is contiguous in memory, so only this one benefits directly from spatial locality. This is also the only dimension that benefits from alignment with cache lines and vectorization unless the data will be transposed as part of the computation. Partitioning is related to strip-mining the stencil pattern, which is discussed in Section 7.3.

Partitioning can be generalized to another pattern that we will call segmentation. Segmentation still requires non-overlapping sections, but now the sections can vary in size. This is shown in Figure 6.18.

Various algorithms have been designed to operate on segmented data, including segmented versions of scan and reduce that can operate on each segment of the array but in a perfectly load-balanced fashion, regardless of the irregularities in the lengths of the segments [BHC+93]. These segmented algorithms can actually be implemented in terms of the normal scan and reduce algorithms by using a suitable combiner function and some auxiliary data. Other algorithms, such as quicksort [Ble90, Ble96], can in turn be implemented in a vectorized fashion with a segmented data structure using these primitives.

In order to represent a segmented collection, additional data is required to keep track of the boundaries between sections. The two most common representations are shown in Figure 6.19.
Example: Scaled Vector Addition (SAXPY)

- \( y \leftarrow a x + y \)
  - 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>4</td>
<td>4</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:** \( y \leftarrow ax + y \)

<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><strong>a</strong></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><strong>x</strong></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><strong>+</strong></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><strong>y</strong></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>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>

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>( * )</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>( 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>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>( 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 \( \rightarrow \) one for every six elements in the vector
Serial SAXPY Implementation

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 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];
        });
}
```
4.2 Scaled Vector Addition (SAXPY)

The performance of the serial code inside TBB tasks will depend on the performance of the code generated by the C++ compiler with which it is used.

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.

LISTING 4.3

```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];
}
```

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];
}
```
Map

Uniform inputs are handled by scalar promotion:

When a scalar and an array are combined with an operator, the scalar is conceptually “promoted” to an array of the same length by replication.

4.2.6 OpenMP

Like TBB and Cilk Plus, the map pattern is expressed in OpenMP using a “parallel for” construct. This is done by adding a `pragma` as in Listing 4.5 just before the loop to be parallelized. OpenMP uses a “team” of threads and the work of the loop is distributed over the team when such a pragma is used. How exactly the distribution of work is done is given by the current scheduling option. The advantage of the OpenMP syntax is that the code inside the loop does not change, and the annotations can usually be safely ignored and a correct serial program will result. However, as with the equivalent Cilk Plus construct, the form of the `for` loop is more restricted than in the serial case. Also, as with Cilk Plus and TBB, implementations of OpenMP generally do not check for incorrect parallelizations that can arise from dependencies between loop iterations, which can lead to race conditions.

4.2.7 ArBB Using Vector Operations

ArBB operates only over data stored in ArBB containers and requires using ArBB types to represent elements of those containers. The ArBB `dense` container represents multidimensional arrays. It is a template with the first argument being the element type and the second the dimensionality. The dimensionality default is 1 so the second template argument can be omitted for 1D arrays.

The simplest way to implement SAXPY in ArBB is to use arithmetic operations directly over `dense` containers, as in Listing 4.6. Actually, this gives a sequence of maps. However, as will be explained in Section 4.4, ArBB automatically optimizes this into a `map` of a `sequence`.

In ArBB, we have to include some extra code to move data into “ArBB data space” and to invoke the above function. Moving data into ArBB space is required for two reasons: safety and offload.

Data stored in ArBB containers can be managed in such a way that race conditions are avoided. For example, if the same container is both an input and an output to a function, ArBB will make sure that

```c
1 void saxpy_openmp(
2     int n,   // the number of elements in the vectors
3     float a, // scale factor
4     float x[], // the first input vector
5     float y[]  // the output vector and second input vector
6 ) {
7     #pragma omp parallel for
8         for (int i = 0; i < n; ++i)
9             y[i] = a * x[i] + y[i];
10 }
```
OpenMP SAXPY Performance

Vector size = 500,000,000
Collectives

- Collective operations deal with a *collection* of data as a whole, rather than as separate elements.

- Collective patterns include:
  - Reduce
  - Scan
  - Partition
  - Scatter
  - Gather
Collectives

- Collective operations deal with a *collection* of data as a whole, rather than as separate elements.

- Collective patterns include:
  - Reduce
  - Scan
  - Partition
  - Scatter
  - Gather

Reduce and Scan will be covered in this lecture.
Reduce

- **Reduce** is used to combine a collection of elements into one summary value
- A combiner function combines elements pairwise
- A combiner function only needs to be *associative* to be parallelizable

- **Example combiner functions:**
  - Addition
  - Multiplication
  - Maximum / Minimum
How do we actually implement the parallel reduction?
Reduce

- Vectorization
Reduce

- **Tiling** is used to break chunks of work up for workers to reduce serially
Reduce – Add Example
Reduce – Add Example

1 2 5 4 9 7 0 1

3 8 12 21 28 28 29

29
Reduce – Add Example
Reduce – Add Example

1  2  5  4  9  7  0  1

3  9 16  1
12 17 29

29
Reduce

- We can “fuse” the map and reduce patterns
Reduce

- Precision can become a problem with reductions on floating point data
- Different orderings of floating point data can change the reduction value
Reduce Example: Dot Product

- 2 vectors of same length
- Map (*) to multiply the components
- Then reduce with (+) to get the final answer

\[ \mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i. \]

Also:

\[ \mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \cos(\theta) |\mathbf{b}| \]
**Dot Product – Example Uses**

- Essential operation in physics, graphics, video games,…
- Gaming analogy: in Mario Kart, there are “boost pads” on the ground that increase your speed
  - Red vector is your speed (x and y direction)
  - Blue vector is the orientation of the boost pad (x and y direction)
  - Larger numbers are more power

How much boost will you get? For the analogy, imagine the pad multiplies your speed:
  - If you come in going 0, you’ll get nothing
  - If you cross the pad perpendicularly, you’ll get 0 [just like the banana obliteration, it will give you 0x boost in the perpendicular direction]

\[
\text{Total} = \text{speed}_x \cdot \text{boost}_x + \text{speed}_y \cdot \text{boost}_y
\]

Ref: http://betterexplained.com/articles/vector-calculus-understanding-the-dot-product/
Scan

- The **scan** pattern produces partial reductions of input sequence, generates new sequence
- Trickier to parallelize than reduce
- Inclusive scan vs. exclusive scan
  - Inclusive scan: includes current element in partial reduction
  - Exclusive scan: excludes current element in partial reduction, partial reduction is of all prior elements prior to current element
Scan – Example Uses

- Lexical comparison of strings – e.g., determine that “strategy” should appear before “stratification” in a dictionary
- Add multi-precision numbers (those that cannot be represented in a single machine word)
- Evaluate polynomials
- Implement radix sort or quicksort
- Delete marked elements in an array
- Dynamically allocate processors
- Lexical analysis – parsing programs into tokens
- Searching for regular expressions
- Labeling components in 2-D images
- Some tree algorithms
  - Example: finding the depth of every vertex in a tree
Scan

Serial Scan

Parallel Scan
Scan

- One algorithm for parallelizing scan is to perform an “up sweep” and a “down sweep”
- Reduce the input on the up sweep
- The down sweep produces the intermediate results
Scan – Maximum Example
Scan – Maximum Example

[Diagram showing the process of a scan operation to find the maximum value in parallel computing. The diagram illustrates the reduction of a sequence of numbers through parallel processing, with arrows indicating the flow of data and the accumulation of the maximum value.]
Scan

- Three phase scan with tiling
Scan

- Just like reduce, we can also fuse the **map** pattern with the **scan** pattern
Scan
Merge Sort as a Reduction

- We can sort an array via a map and a reduce
- Map each element into a vector
  - Contains just that element
- Merge vectors
  - \(<>\) is the merge operation
    - \([1,3,5,7] <> [2,6,15] = [1,2,3,5,6,7,15]\)
  - \([\ ]\) is the empty list
- How fast is this?
Right Biased Sort

Start with [14,3,4,8,7,52,1]
Map to  [[14],[3],[4],[8],[7],[52],[1]]
Reduce:

\[\begin{align*}
14 & \leftrightarrow (3 \leftrightarrow (4 \leftrightarrow (8 \leftrightarrow (7 \leftrightarrow (52 \leftrightarrow [1]))))) \\
= 14 & \leftrightarrow (3 \leftrightarrow (4 \leftrightarrow (8 \leftrightarrow (7 \leftrightarrow [1,52])))) \\
= 14 & \leftrightarrow (3 \leftrightarrow (4 \leftrightarrow (8 \leftrightarrow [1,7,52]))) \\
= 14 & \leftrightarrow (3 \leftrightarrow (4 \leftrightarrow [1,7,8,52])) \\
= 14 & \leftrightarrow (3 \leftrightarrow [1,4,7,8,52]) \\
= 14 & \leftrightarrow [1,3,4,7,8,52] \\
= [1,3,4,7,8,14,52]
\end{align*}\]
Right Biased Sort (Continued)

- How long did that take?
- We did $O(n)$ merges…but each one took $O(n)$ time
- $O(n^2)$
- We wanted merge sort, but instead we got insertion sort!
Tree Shaped Sort

Start with \([14,3,4,8,7,52,1]\)
Map to \([[14],[3],[4],[8],[7],[52],[1]]\)
Reduce:

\[
([14] \leftrightarrow [3]) \leftrightarrow ([4] \leftrightarrow [8]) \leftrightarrow ([7] \leftrightarrow [52]) \leftrightarrow [1] \\
= ([3,14] \leftrightarrow [4,8]) \leftrightarrow ([7,52] \leftrightarrow [1]) \\
= [3,4,8,14] \leftrightarrow [1,7,52] \\
= [1,3,4,7,8,14,52]
\]
Tree Shaped Sort Performance

- Even if we only had a single processor this is better
  - We do $O(\log n)$ merges
  - Each one is $O(n)$
  - So $O(n*\log(n))$

- But opportunity for parallelism is not so great
  - $O(n)$ assuming sequential merge
  - Takeaway: the shape of reduction matters!