Logistics

- Programming lab
  - Look at the LLNL OpenMP tutorial (see webpage)
- Start thinking about potential project ideas
  - Elevator pitches on Thursday, February 8
- Read SPP Chapter 3 – Patterns
- Read SPP Chapter 4 – Map
- Read SPP Chapters 5 – Collectives
Outline

- Map pattern
  - Optimizations
    - sequences of maps
    - Code fusion
    - Cache fusion
  - Related Patterns
  - Example: Scaled Vector Addition (SAXPY)
    - problem description
    - various implementations

- Collectives pattern
  - Reduce pattern
  - Scan pattern
  - Example: Sorting
Mapping

- What is map(ping)?
- “Do the same thing many times”
  ```python
def do_something(i):
    pass
for i in foo:
  do_something(i)
  ```

- Well-known higher order function in languages like ML, Haskell, Scala
  ```math
  \text{map} : \forall a,b. (a \rightarrow b) \times \text{List}\langle a\rangle \rightarrow \text{List}\langle b\rangle
  ```
  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>

\[\begin{array}{cccccc}
+1 \\
\downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \\
1 & 5 & 6 & 4 & 2 & 1
\end{array}\]

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

\[\begin{array}{cccccc}
*2 \\
\downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \\
6 & 14 & 0 & 2 & 8 & 0
\end{array}\]

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

\[
\text{parallel\_for\_each}(x \text{ in array})\
\begin{align*}
\text{process}(x); \\
\end{align*}
\]
Comparing Maps

**Serial Map**

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

**Parallel Map**

- Data
- Task
- Task
- Task
- Task
- Data
- Data
- Data
- Data
- Data
- Data
- Data
Comparing Maps

Serial Map

Parallel Map

The space here represents 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
  - Segmentation faults (segfaults)
Implementation and API

- OpenMP and CilkPlus contain a parallel `for` language construct
  - OpenMP has a version for Fortran and C/C+

- Map is a mode of use of parallel `for`

- TBB uses **higher order functions** with lambda expressions and “functors”

- Some languages (CilkPlus, Matlab, Fortran) provide **array notation** which makes some maps more concise

Array Notation

```
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

```
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>
<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>5</td>
<td>11</td>
<td>2</td>
<td>2</td>
<td>12</td>
<td>3</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

```c
int twoTOone ( int x[11], int y[11] ){
    return x+y;
}
```
Optimization – Sequences of Maps

- 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
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
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

Another approach that is often almost as effective as code fusion is **cache fusion**, shown in Figure 4.3. If the maps are broken into tiles and the entire sequence of smaller maps for one tile is executed sequentially on one core, then if the aggregate size of the tiles is small enough intermediate data will be resident in cache. In this case at least it will be possible to avoid going to main memory.

Both kinds of fusion also reduce the cost of synchronization, since when multiple maps are fused only one synchronization is needed after all the tiles are processed, instead of after every map. However, code fusion is preferred when it is possible since registers are still faster than cache, and with cache fusion there is still the "interpreter" overhead of managing the multiple passes. However, cache fusion is useful when there is no access to the code inside the individual maps—for example, if they are provided as precompiled user-defined functions without source access by the compiler. This is a common pattern in, for example, image processing plugins.

In Cilk Plus, TBB, OpenMP, and OpenCL the reorganization needed for either kind of fusion must generally be done by the programmer, with the following notable exceptions:

**OpenMP:**
- Cache fusion occurs when all of the following are true:
  - A single parallel region executes all of the maps to be fused.
  - The loop for each map has the same bounds and chunk size.
  - Each loop uses the `static` scheduling mode, either as implied by the environment or explicitly specified.
Inputs and Outputs in Map

- Need to be concerned about whether the input and output collections are the same
- If the input and output collections are different
  - Map writes (output) operations will not affect reads
- If the input and output collections are different
  - Map writes (output) operations should be careful to not overwrite data in the input collection before it is read
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

![Stencil Diagram]

Note, here we have to be concerned about the input and output collections.
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
- Think of a “bag” of data to be processed and new data can be added to the “bag” during execution
  - Also known as a “bag of tasks”
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
- The base cases are processed as a map

FIGURE 6.17

Partitioning in 2D. The partition pattern can be extended to multiple dimensions. 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)

- \( y \leftarrow ax + y \)
  - 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>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>$y$</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>$y'$</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:** \( y \leftarrow ax + y \)

<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>

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>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>

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

Two processors used \( \rightarrow \) one for every six elements in the vector
# Serial SAXPY Implementation

1. `void saxpy_serial(
   2.   size_t n,          // the number of elements in the vectors
   3.   float a,           // scale factor
   4.   const float x[],   // the first input vector
   5.   float y[]          // the output vector and second input vector
   6. ) {
   7.   for (size_t i = 0; i < n; ++i)
   8.     y[i] = a * x[i] + y[i];
   9. }

Listing 4.1 Serial implementation of SAXPY in C.

1. `void saxpy_tbb(
   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.   tbb::parallel_for(
   8.     tbb::blocked_range<int>(0, n),
   9.     [] (tbb::blocked_range<int> r)
   10.    {
   11.       for (size_t i = r.begin(); i != r.end(); ++i)
   12.         y[i] = a * x[i] + y[i];
   13.     });
   14. }

Listing 4.2 Tiled implementation of SAXPY in TBB. Tiling not only leads to better spatial locality but also exposes opportunities for vectorization by the host compiler.

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

functions for brevity throughout the book, though they are not required for using TBB. Appendix D.2 discusses lambda functions and how to write the equivalent code by hand if you need to use an old C++ compiler.

The TBB code exploits tiling. The parallel_for breaks the half-open range \([0, n)\) into subranges and processes each subrange \(r\) with a separate task. Hence, each subrange \(r\) acts as a tile, which is processed by the serial for loop in the code. Here the range and subrange are implemented as blocked_range objects. Appendix C.3 says more about the mechanics of parallel_for. 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
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];
        }
    );
}
```

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
OpenMP SAXPY Implementation

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

SAXPY on a NUC

Time (seconds)

Number of Threads

Vector size = 500,000,000
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.

```c
1 void saxpy_cilk(
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     cilk_for (int i = 0; i < n; ++i)
8         y[i] = a * x[i] + y[i];
9 }
```
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
Lecture 7 – Map and Collective Patterns

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 Overview

- **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 with Vectorization
Reduce with Tiling

- **Tiling** is used to break chunks of work up for workers to reduce serially
Reduce – Add Example (serial)
Reduce – Add Example (serial)
Reduce – Add Example (parallel)
Reduce – Add Example (parallel)
Reduce with Fusing

- We can “fuse” the map and reduce patterns

![Diagram showing the fusion of map and reduce patterns]
Reduce Precision and Ordering

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

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

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]

\[ Total = speed_x \cdot boost_x + speed_y \cdot boost_y \]

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

- 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 Diagrams

Serial Scan

Parallel Scan
Scan Parallelization Method

- 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
Three-phase Scan with Tiling
Scan with Fusing

- Just like reduce, we can also fuse the map pattern with the scan pattern
Scan with Fusing
Scan – Example Uses

- Lexical comparison of strings
  - Does “strategy” 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
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:

\[
[14] \Leftrightarrow ([3] \Leftrightarrow ([4] \Leftrightarrow ([8] \Leftrightarrow ([7] \Leftrightarrow ([52] \Leftrightarrow [1]))))))
\]

= [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]
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:

$(((14) <> [3]) <> ([4] <> [8])) <> (([7] <> [52]) <> [1])$

$= ([3,14] <> [4,8]) <> ([7,52] <> [1])$

$= [3,4,8,14] <> [1,7,52]$

$= [1,3,4,7,8,14,52]$
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 $O(n \times \log(n))$

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