Logistics

- Homework exercise posted
  - Due Tuesday, April 21, 17:00
- Paper instructions for graduate students
- Begin patterns programming in the lab on Friday
- “Special Report: 50 Years of Moore’s Law, The glorious history and inevitable decline of one of technology’s greatest winning streaks”
  IEEE Spectrum

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
  ```
  map : ∀ab.(a → b)List<a> → 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

<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>

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>

| 3 | 7 | 0 | 1 | 4 | 0 |

**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” 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

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

Serial Map

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

Parallel Map

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

Serial Map

Data

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 `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

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

```
 0 1 2 3 4 5 6 7 8 9 10 11
x  3 7 0 1 4 0 0 4 5 3 1 0
y  2 4 2 1 8 3 9 5 5 1 2 1
*  ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
result  5 11 2 2 12 3 9 9 10 4 3 1
```

```
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

*Diagram:*

1. Green block represents the map operations.
2. Blue block represents the intermediate results.
3. Arrows indicate the flow of data from one operation to the next.

*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 – 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
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>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:** \( y \leftarrow ax + y \)

<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>
</tbody>
</table>

12 processors used → 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>
</tbody>
</table>

\[ y \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \]

|   | 11 | 23 | 8 | 5 | 36 | 12 | 36 | 49 | 50 | 7 | 9 | 4  |

**Six processors used ⇒ 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></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>

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. }
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];
        }
    );
}
```

This implementation uses TBB's `parallel_for` function to parallelize the SAXPY operation over a range of elements in the vectors. Tiling is employed to improve spatial locality, which exposes opportunities for vectorization by the host compiler. TBB uses thread parallelism but does not vectorize the code itself; it relies on the underlying C++ compiler for vectorization. If the basic serial algorithm can be vectorized, then the TBB code can typically be vectorized as well.
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];
}
```

```c
void saxpy_array_notation(
    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
) {
    y[0:n] = a * x[0:n] + y[0:n];
}
```
OpenMP SAXPY Implementation

```cpp
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
Reduce

Serial Reduction

Parallel Reduction

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
Reduce – Add Example

1 2 5 4 9 7 0 1
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
Scan

- Three phase scan with tiling
Scan
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:

\[ [14] \leftrightarrow ([3] \leftrightarrow ([4] \leftrightarrow ([8] \leftrightarrow ([7] \leftrightarrow ([52] \leftrightarrow [1]))))) \]
= [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 Cont

- 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 Shape 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 $O(n\log(n))$

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