Lecture 8 – Map and Collective Patterns

CIS 410/510
Parallel Computing
Map and Collective Patterns

Prof. Allen D. Malony
Department of Computer and Information Science
Spring 2016
Logistics

- Homework exercise posted
  - Due Tuesday, April 19, 17:00

- Lab
  - Look at the OpenMP tutorial
  - Keep working on OpenMP exercises
  - Next up is Cilk Plus
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
  ```plaintext
  map : \forall ab.(a \to b)List\langle a\rangle \to 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

<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>

\[ +1 \]

| 1 | 5 | 6 | 4 | 2 | 1 |

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>

\[ \times 2 \]

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

| 6 | 14 | 0 | 2 | 8 | 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 (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]);
}
```
Parallel Map

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

Serial Map

Parallel Map
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 (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

```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

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\hline
x & 3 & 7 & 0 & 1 & 4 & 0 & 0 & 4 & 5 & 3 & 1 & 0 \\
*2 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
result & 6 & 14 & 0 & 2 & 8 & 0 & 0 & 8 & 10 & 6 & 2 & 0 \\
\end{array}
\]

```c
int oneTOone ( int x[11] ) {
    return x*2;
}
```
But, sometimes it makes sense to map over multiple collections at once…
Map with 2 Inputs, 1 Output

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\hline
x & 3 & 7 & 0 & 1 & 4 & 0 & 0 & 4 & 5 & 3 & 1 & 0 \\
y & 2 & 4 & 2 & 1 & 8 & 3 & 9 & 5 & 5 & 1 & 2 & 1 \\
\hline
result & 5 & 11 & 2 & 2 & 12 & 3 & 9 & 9 & 10 & 4 & 3 & 1 \\
\end{array}
\]

```c
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.
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
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

Implementing Stencil with Shift

The regular data access pattern used by stencils can be implemented using shifts. For a group of elemental functions, a vector of inputs for each offset in the stencil can be collected by shifting the input by the amount of the offset. This is diagrammed in Figure 7.2.

Implementing a stencil in this way is really only beneficial for one-dimensional stencils or the memory-contiguous dimension of a multidimensional stencil. Also, it does not reduce total memory traffic to external memory since, if random scalar reads are used, data movement from external memory will still be combined into block reads by the cache. Shifts, however, allow vectorization of the data reads, and this can reduce the total number of instructions used. They may also place data in vector registers ready for use by vectorized elemental functions.
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.

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>$+$</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>$\downarrow$</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>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>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>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
Two processors used $\Rightarrow$ 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];
}
```

Listing 4.1 Serial implementation of SAXPY in C.
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];
        });
}
```

Tiled implementation of SAXPY in TBB. Tiling not only leads to better spatial locality but also exposes opportunities for vectorization by the host compiler. 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 with a separate task. Hence, each subrange 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...
4.2 Scaled Vector Addition (SAXPY)

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 }
```

LISTING 4.3 SAXPY in Cilk Plus using cilk_for.

```c
1 void saxpy_array_notation(
2     int n,       // the number of elements in the vectors
3     float a,    // scale factor
4     float x[],  // the input vector
5     float y[]   // the output vector and offset
6 ) {
7     y[0:n] = a * x[0:n] + y[0:n];
8 }
```

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
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

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
Reduce – Add Example
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]

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

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
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.

Up sweep – compute reduction

Down sweep – compute intermediate values

Scan
Scan – Maximum Example
Scan – Maximum Example
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
  - $\langle\rangle$ is the merge operation
    - $[1,3,5,7] \langle\rangle [2,6,15] = [1,2,3,5,6,7,15]$  
  - $[]$ 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:

\[
\begin{align*}
( ([14] &\triangleleft [3]) \triangleleft ([4] &\triangleleft [8])) \triangleleft ([7] &\triangleleft [52]) \triangleleft [1] \\
= ([3,14] &\triangleleft [4,8]) \triangleleft ([7,52] &\triangleleft [1]) \\
= [3,4,8,14] &\triangleleft [1,7,52] \\
= [1,3,4,7,8,14,52]
\end{align*}
\]
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 $O(n\cdot \log(n))$

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