Compsci 590.3:
Introduction to Parallel Computing

Alvin R. Lebeck

Slides based on those from the University of Oregon
Admin

Logistics
• Homework 0
  ▪ Serial Hello World
  ▪ Parallel Hello World
• Homework 1
  ▪ Available tonight or tomorrow..
  ▪ I’ll create group repository
• Use Piazza for questions…
  ▪ I do read email, but it might get lost…
  ▪ Others can possibly answer.
  ▪ Others can learn from answer.

• Homework Groups
  ▪ spread sheet
  ▪ Use piazza if you need to find partners
• HW groups assigned to specific machines

Outline
• Dependencies
• Patterns
  ▪ Map
  ▪ Reduce
  ▪ Scan
Independent versus Dependent

• Two statements are independent if execution of
  statement1;
  statement2;
  ▪ must be equivalent to
  statement2;
  statement1;

• Their order of execution must not matter!
• If true, the statements are independent of each other
• Two statements are dependent when the order of their execution affects the computation outcome
Dependence Examples

1. Example
   S1: a=1;
   S2: b=1;

2. Example
   S1: a=1;
   S2: b=a;

3. Example
   S1: a=f(x);
   S2: a=b;

4. Example
   S1: a=b;
   S2: b=1;

1. Statements are independent

2. Dependent (*true (flow) dependence*)
   - Second is dependent on first
   - Can you remove dependency?

3. Dependent (*output dependence*)
   - Second is dependent on first
   - Can you remove dependency?

4. Dependent (*anti-dependence*)
   - First is dependent on second
   - Can you remove dependency?
True Dependence and Anti-Dependence

- Given statements $S_1$ and $S_2$,
  $S_1$;
  $S_2$;

- $S_2$ has a \textit{true (flow) dependence} on $S_1$
  if and only if
  $S_2$ reads a value written by $S_1$

- $S_2$ has a \textit{anti-dependence} on $S_1$
  if and only if
  $S_2$ writes a value read by $S_1$
Output Dependence

- Given statements S1 and S2,
  S1;
  S2;
- S2 has an output dependence on S1
  if and only if
  S2 writes a variable written by S1

- Anti- and output dependences are “name” dependencies
  - Are they “true” dependences?
- How can you get rid of output dependences?
  - Are there cases where you can not?
Directed Acyclic Graphs (DAG)

• Captures data flow parallelism
• Nodes represent operations to be performed
  ▪ Inputs are nodes with no incoming arcs
  ▪ Output are nodes with no outgoing arcs
  ▪ Think of nodes as tasks
• Arcs are paths for flow of data results
• DAG represents the operations of the algorithm and implies precedent constraints on their order

\[
\text{for } (i=1; i<100; i++)
\]
\[
a[i] = a[i-1] + 100;
\]
Statement Dependency Graphs

- Can use graphs to show dependence relationships
- Example
  
  S1: \( a=1; \)  
  S2: \( b=a; \)  
  S3: \( a=b+1; \)  
  S4: \( c=a; \)

- \( S_2 \delta S_3 : S_3 \) is flow-dependent on \( S_2 \)
- \( S_1 \delta^0 S_3 : S_3 \) is output-dependent on \( S_1 \)
- \( S_2 \delta^{-1} S_3 : S_3 \) is anti-dependent on \( S_2 \)
When can two statements execute in parallel?

• Statements S1 and S2 can execute in parallel if and only if there are no dependences between S1 and S2
  ▪ True dependences
  ▪ Anti-dependences
  ▪ Output dependences

• Some dependences can be remove by modifying the program
  ▪ Rearranging statements
  ▪ Eliminating statements

• Be very careful about reordering statements! This can affect result given limited precision in computers!
Parallel Patterns

- **Parallel Patterns**: 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.
Map Pattern - Overview

• Map
• Optimizations
  ▪ Sequences of Maps
  ▪ Code Fusion
• Related Patterns
• Example Implementation: Scaled Vector Addition (SAXPY)
  ▪ Problem Description
  ▪ Various Implementations
Parallel Control Patterns: Map

- Map: performs a function over every element of a collection
- Map replicates a serial iteration pattern where each iteration is independent of the others, and computation only depends on the iteration count and data from the input collection
- The replicated function is referred to as an “elemental function”
Mapping

- “Do the same thing many times”
  
  foreach i in foo:
    
    do something
  

  while (!done)
    
    done = f(element i)

- Well-known higher order function in languages like ML, Haskell, Scala

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

- applies a function 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>3</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Key Point: An operation is a map if it can be applied to each element without knowledge of neighbors.
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 \( T(\infty) \in O(\log n) \).
Performance side track (one slide)

• $T = \text{execution time}$

• Speedup $= \frac{T_{\text{old}}}{T_{\text{new}}}$
• Speedup on $P$ processors $= \frac{T_{\text{serial}}}{T_p}$

• More in a few lectures…
for(int n=0; n< array.length; ++n) {
    process(array[n]);
}
```
parallel_for_each(x in array)
{
    process(x);
}
```
Comparing Maps

Serial Map

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

Parallel Map

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

© Alvin R. Lebeck slides from U Oregon material Compsci 590.3 Parallel Computing
Comparing Maps

Serial Map

Parallel Map

Speedup

The space here produces 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:
  - undefined behavior, Segfaults, non-determinism, data-races
  - Your program crashes or produces incorrect result!
- Potential exception is the shared variable to signal termination event. But can use the parallel task cancel also.
Implementation and API

- OpenMP and CilkPlus contain a parallel for language construct
- Map is a mode of use of parallel for
- TBB uses higher order functions with lambda expressions/“funtors”
- Some languages (CilkPlus, Matlab, Fortran) provide array notation which makes some maps more concise

Array Notation

\[ A[:,] = A[:,] \times 5; \]

is CilkPlus array notation for “multiply every element in \( A \) by 5”
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

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.
4.4 Sequence of Maps versus Map of Sequence

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.
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.

Compsci 590.3 Parallel Computing
Related Patterns

- Three patterns related to map are discussed here:
  - Stencil
  - Workpile
  - Divide-and-Conquer

- More detail presented in a later lecture
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?

$$\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\hline
a & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
\times & 2 & 4 & 2 & 1 & 8 & 3 & 9 & 5 & 5 & 1 & 2 & 1 \\
\hline
+ & 3 & 7 & 0 & 1 & 4 & 0 & 0 & 4 & 5 & 3 & 1 & 0 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
y & 11 & 23 & 8 & 5 & 36 & 12 & 36 & 49 & 50 & 7 & 9 & 4
\end{array}$$
Visual: $y \leftarrow ax + y$

Twelve processors used $\Rightarrow$ one for each element in the vector
Visual:  \( y \leftarrow ax + y \)

Six processors used \( \rightarrow \) one for every two elements in the vector

© Alvin R. Lebeck  slides from U Oregon material  Compsci 590.3 Parallel Computing
**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>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><strong>y</strong></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

```c
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 }
```
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];
}
```

Chapter 4: 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. If these exist and are not correctly accounted for in the pragma, an incorrect parallelization will result.

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.
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];
}
```

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

```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];
}
```

LISTING 4.3 SAXPY in Cilk Plus using `cilk_for`.

LISTING 4.4 SAXPY in Cilk Plus using `cilk_for` and array notation for explicit vectorization.
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];
        }
    );
}
```
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

• Chapter 5 in book
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
Reduce

- Vectorization
Reduce

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

© Alvin R. Lebeck  slides from U Oregon material
Comp sci 590.3 Parallel Computing
Reduce – Add Example
Reduce – Add Example
Reduce – Add Example
Reduce

- We can “fuse” the map and reduce patterns
Side Note: Precision

- Precision can become a problem with reductions on floating point data
- Different orderings of floating point data can change the reduction value
- Limited # of bits to represent numbers -> rounding errors -> order matters!

- You may not get exact same answer as serial version of program
- Entire field of Numerical Analysis studies these error bounds
Reduce Example: Dot Product

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

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

Also:
\[
\vec{a} \cdot \vec{b} = |\vec{a}| \cos(\theta) |\vec{b}|
\]
Reductions: OpenMP & CilkPlus

- **Openmp**
  #pragma parallel for reduction(+:sum)
  for (int i=0; i < n; i++)
      sum += (a[i] * b[i]);
  printf("sum is %f\n", sum);

- **CilkPlus**
  cilk::reducer_opadd<float> sum(0));
  cilk_for (int i = 0; i < n; i++)
      sum += (a[i] * b[i]);
  printf("sum is %f\n", sum.get_value());
Reductions: TBB

- **TBB**
  
  ```cpp
double my_simple_reduce(int n) {
    return parallel_reduce(
      blocked_range<int>(0,n), double(0),
      [&](blocked_range<int> r, double in) {
        for (int i=r.begin(); i<r.end(); i++){
          in += A[i];
        }
        return(in);
      },
      std::plus<double>()
    );
  }
```

- **Reduce syntax**

  ```cpp
type result = tbb::parallel_reduce(
  range, identity,
  subrange reduction,
  combine);
```

  - **Identity value**: $x \ op \ value = x$
  - **Subrange reduction**: function applied to the subrange (tile) of values to obtain partial result
  - **Combine**: function applied to partial results to get final result
Scan

• Section 5.4 in Book

• The `scan` pattern produces partial reductions of input sequence, generates new sequence
  ▪ Sometimes called prefix computation

• Trickier to parallelize than reduce
  ▪ Also called “parallel prefix”

• 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

• Cilk: `cilk_scan`

• TBB: `parallel_scan`

• OpenMP: write your own…
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
Summary

- Dependencies
  - True, Anti and Output
- Patterns
- Map
- Reduce
- Scan

- Homework partners…
- Homework 1 will be up soon