Logistics

- Lectures (audio/video) up-to-date on website
- MPI exercise is posted (see Assignments webpage)
  - Deadline change to Sunday, January 23, midnight
Contents

- What is the stencil pattern?
  - Update alternatives
  - 2D Jacobi iteration
  - SOR and Red/Black SOR

- Partitioning and parallelization
Stencil Pattern

- A stencil pattern is a map where each output depends on a “neighborhood” of inputs

- These inputs are a set of fixed offsets relative to the output position

- A stencil output is a function of a “neighborhood” of elements in an input collection
  - Applies the stencil to select the inputs

- Data access patterns of stencils are regular
  - Stencil is the “shape” of “neighborhood”
  - Stencil remains the same for all parallel operations
Where is the stencil pattern used?

- Recurrences (iterative computations) over multidimensional arrays with regular neighborhood access patterns
- Image and signal processing
  - Convolution – input samples are combined using a weighted sum (linear)
- Bilateral filtering for noise reduction (nonlinear)
- Solving PDEs discretized over regular grids
  - scientific and engineering simulations
  - photography, satellite imaging, medical imaging, seismic reconstruction (used in oil and gas exploration)
Serial Stencil Example (part 1)

```c++
template<
    int NumOff,  // number of offsets
    typename In,  // type of input locations
    typename Out,  // type of output locations
    typename F  // type of function/functor
>
void stencil(
    int n,       // number of elements in data collection
    const In a[],  // input data collection (n elements)
    Out r[],       // output data collection (n elements)
    In b,          // boundary value
    F func,        // function/functor from neighborhood inputs to output
    const int offsets[]  // offsets (NumOff offsets elements)
) { ... }
```
Serial Stencil Example (part 2)

```java
// array to hold neighbors
In neighborhood[NumOff];
// loop over all output locations
for (int i = 0; i < n; ++i) {
    // loop over all offsets and gather neighborhood
    for (int j = 0; j < NumOff; ++j) {
        // get index of jth input location
        int k = i + offsets[j];
        if (0 <= k && k < n) {
            // read input location
            neighborhood[j] = a[k];
        } else {
            // handle boundary case
            neighborhood[j] = b;
        }
    }
    // compute output value from input neighborhood
    r[i] = func(neighborhood);
}
```

How would we parallelize this?
What is the stencil pattern?
What is the stencil pattern?

Input array
What is the stencil pattern?
What is the stencil pattern?

Output Array
What is the stencil pattern?

This stencil has 3 elements in the neighborhood: i-1, i, i+1
What is the stencil pattern?

Applies some function to them…

i-1  i  i+1

neighborhood
What is the stencil pattern?

And outputs to the $i^{th}$ position of the output array
Stencil Patterns

- Stencils can operate on one dimensional and multi-dimensional data
- Stencil neighborhoods can range from compact to sparse, square to cube, and anything else!
- It is the pattern of the stencil that determines how the stencil operates in an application
- Stencil patterns derive from the numerical solution requirements of particular computational problems
  - Partial differential equations are a main type
2-Dimensional Stencils

4-point stencil
Center cell (P) is not used

5-point stencil
Center cell (P) is used as well

9-point stencil
Center cell (C) is used as well

Source: http://en.wikipedia.org/wiki/Stencil_code
3-Dimensional Stencils

6-point stencil
(7-point stencil)

24-point stencil
(25-point stencil)

Source: http://en.wikipedia.org/wiki/Stencil_code
**Stencil Example**

- Here is our array, A

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 9 & 7 & 0 & 0 & 0 \\
0 & 6 & 4 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
## Stencil Example

- Here is our array $A$
- Assume an output array $B$
  - Initialize to all 0
- Apply a stencil operation to the inner square of the form:
  
  \[ B(i,j) = \text{avg}( A(i,j), \ A(i-1,j), A(i+1,j), \ A(i,j-1), A(i,j+1) ) \]

- What is the stencil?

![Stencil Example](image.png)
1) Average all blue squares
# Stencil Pattern Procedure

1) Average all blue squares
2) Store result in B

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Stencil Pattern Procedure

1) Average all blue squares
2) Store result in B
3) Repeat Step 1 and Step 2 for all green squares
Serial Stencil Example (part 1)

```cpp
template<
    int NumOff, // number of offsets
typename In,  // type of input locations
typename Out, // type of output locations
typename F    // type of function/functor
>

void stencil(
    int n,      // number of elements in data collection
    const In a[], // input data collection (n elements)
    Out r[],    // output data collection (n elements)
    In b,       // boundary value
    F func,     // function/functor from neighborhood inputs to output
    const int offsets[] // offsets (NumOffsets elements)
)

```
Serial Stencil Example (part 2)

```c
// array to hold neighbors
In neighborhood[NumOff];
// loop over all output locations
for (int i = 0; i < n; ++i) {
    // loop over all offsets and gather neighborhood
    for (int j = 0; j < NumOff; ++j) {
        // get index of jth input location
        int k = i + offsets[j];
        if (0 <= k && k < n) {
            // read input location
            neighborhood[j] = a[k];
        } else {
            // handle boundary case
            neighborhood[j] = b;
        }
    }
    // compute output value from input neighborhood
    a[i] = func(neighborhood);
}
```

How would we parallelize this?

Updates occur in place!!!
Stencil Pattern with In Place Update
Stencil Pattern with In Place Update

Input array
Stencil Pattern with In Place Update

Function
Stencil Pattern with In Place Update

Input Array!!

Problems?


**Stencil Example**

- Here is our array, $A$

\[
A = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 9 & 7 & 0 & 0 \\
0 & 6 & 4 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]
Stencil Example

• Here is our array $A$
• Update $A$ in place
• Apply a stencil operation to the inner square of the form:

$$A(i,j) = \text{avg}(A(i,j), A(i-1,j), A(i+1,j), A(i,j-1), A(i,j+1))$$

• What is the stencil?
### Stencil Pattern Procedure

1) **Average all blue squares**

![Stencil Pattern Diagram]

- **A**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0</strong></td>
<td>9</td>
<td><strong>7</strong></td>
</tr>
<tr>
<td><strong>0</strong></td>
<td><strong>6</strong></td>
<td><strong>4</strong></td>
</tr>
<tr>
<td><strong>0</strong></td>
<td><strong>0</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Stencil Pattern Procedure

1) Average all blue squares
2) Store result in red square
Stencil Pattern Procedure

1) Average all blue squares
2) Store result in red square
3) Repeat Step 1 and Step 2 for all green squares

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 9 & 7 & 0 \\
0 & 6 & 4 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]
### Different Cases

**Input**
- Separate output array
  - 9 7
  - 6 4

**Output**
- 4.4 4.0
- 3.8 3.4

**Input**
- Updates occur in place
  - 9 7
  - 6 4

**Output**
- 4.4 3.1
- 2.9 2.0
Which is correct?

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 7</td>
<td>4.4 3.1</td>
</tr>
<tr>
<td>6 4</td>
<td>2.9 2.0</td>
</tr>
</tbody>
</table>

- Is this output incorrect?
- Hmm, probably not, but it really depends on what the algorithm intended to do
- It might be ok some algorithms to do "in place" updates
Iterative Codes

- Iterative codes are ones that update their data in steps
  - At each step, a new value of an element is computed using a formula based on other elements
  - Once all elements are updated, the computation proceeds to the next step or completes

- Iterative codes are commonly found in computer simulations of physical systems for science and engineering applications
  - Computational fluid dynamics
  - Electromagnetics modeling

- They are often applied to solve partial differential equations
  - Jacobi iteration
  - Gauss-Seidel iteration
  - Successive over relaxation (SOR)
Iterative Codes and Stencils

- Stencils essentially define which elements are used in the update formula.
- Because the data is organized in a regular manner, stencils can be applied across the data uniformly.
- Iterations in iterative codes are typically associated with steps forward in time.
  - Stencils are applied to compute a new value at a later time from old values at an earlier time.
**Simple 2D Example**

- Consider the following code

```plaintext
def k=1, 1000
  for i=1, N-2
    for j = 1, N-2
      a[i][j] = 0.25 * (a[i][j] + a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1])

Do you see anything interesting?

How would you parallelize?
```
2-Dimension Jacobi Iteration

- Consider a 2D array of elements
- Initialize each array element to some value
- At each step, update each array element to the arithmetic mean of its N, S, E, W neighbors
- Iterate until array values converge
  - What is the convergence criteria?
- Here we are using a 4-point stencil
- It is different from before because we want to update all array elements simultaneously ... How?
2-Dimension Jacobi Iteration

- Consider a 2D array of elements
- Initialize each array element to some value
- At each step, update each array element to the arithmetic mean of its N, S, E, W neighbors
- Iterate until array values converge
Successive Over Relaxation (SOR)

- SOR is an alternate method of solving partial differential equations
- While the Jacobi iteration scheme is very simple and parallelizable, its slow convergent rate renders it impractical for any "real world" applications
- One way to speed up the convergent rate would be to "over predict" the new solution by linear extrapolation
- It also allows a method known as Red-Black SOR to be used to enable parallel updates in place
Red / Black SOR

Pass 1: Writing to RED cells, Reading from BLACK cells

Pass 1: Writing to BLACK cells, Reading from RED cells
Red / Black SOR
Partitioning

- Data is divided into
  - Non-overlapping regions (avoid write conflicts, data races)
  - Equal-sized regions (improve load balancing)
Partitioning

Data is divided into
- Non-overlapping regions (avoid write conflicts, data races)
- Equal-sized regions (improve load balancing)
Conway’s Game of Life

- The Game of Life is a cellular automaton created by John Conway in 1970
- The evolution of the game is entirely based on the input state – zero player game
- To play: create initial state, observe how the system evolves over successive time steps

2D landscape
Conway’s Game of Life

- Typical rules for the Game of Life
  - Infinite 2D grid of square cells, each cell is either “alive” or “dead”
  - Each cell will interact with all 8 of its neighbors
    - Any cell with < 2 live neighbors dies (under-population)
    - Any cell with 2 or 3 live neighbors lives to next gen.
    - Any cell with > 3 live neighbors dies (overcrowding)
    - Any dead cell with 3 live neighbors becomes a live cell
Conway’s Game of Life: Examples
Conway’s Game of Life

- The Game of Life computation can easily fit into the stencil pattern!
- Each larger, bold black box is “owned” by a thread (process)
- What will happen at the boundaries?
Conway’s Game of Life

- We need some way to preserve information from the previous iteration without overwriting it.
- *Ghost Cells* are one solution to the boundary and update issues of a stencil computation.
- Each thread keeps a copy of neighbors’ data to use in its local computations.
- These ghost cells must be updated after each iteration of the stencil.
Conway’s Game of Life

- Working with ghost cells
Conway’s Game of Life

- Working with ghost cells
Conway’s Game of Life

- Working with ghost cells

Compute the new value for this cell
Conway’s Game of Life

- Working with ghost cells

Five of its eight neighbors already belong to this thread.

But three of its neighbors belong to a different thread.
Conway’s Game of Life

- Working with ghost cells

Before any updates are done in a new iteration, all threads must update their ghost cells.
Conway’s Game of Life

- Working with ghost cells

Data this thread can use (including ghost cells from neighbors)
Conway’s Game of Life

- Working with ghost cells
Conway’s Game of Life

- Things to consider…
  - What might happen to our ghost cells as we increase the number of threads?
    - ghost cells to total cells ratio will rapidly increase causing a greater demand on memory
  - What would be the benefits of using a larger number of ghost cells per thread? Negatives?
    - allows several iterations to occur without stopping for a ghost cell update
Stencil and Communication Optimizations

- When data is distributed, ghost cells must be **explicitly** communicated among nodes between loop iterations.

- Darker cells are PE 0’s ghost cells.

- After first iteration of stencil computation:
  - PE 0 must request PE 1 & PE 2’s stencil results.
  - PE 0 can perform another iteration of stencil.
Stencil and Communication Optimizations

- Generally better to replicate ghost cells in each local memory and swap after each iteration than to share memory
  - Fine-grained sharing can lead to increased communication cost
Stencil and Communication Optimizations

- **Halo**: set of all ghost cells
- Halo must contain all neighbors needed for one iteration
- Larger halo *(deep halo)*
  - Trade off
    - less communications and more independence, but…
    - more redundant computation and more memory used

- **Latency Hiding**: Compute interior of stencil while waiting for ghost cell updates
Seismic Raytracing


Calculate shortest travel time paths to sensors

818 points!