High-Performance Scientific Computing
Lecture 13: Parallel Patterns

MATH-GA 2011 / CSCI-GA 2945 · December 5, 2012
Today

Tool of the day: 3D Visualization

Parallel Patterns
Bits and pieces

- HW6: soon
- Dec 12: No class—good luck on finals!
- Dec 17?/18?/19: Project presentations
  - Will announce precise date, watch email
- Project guidelines posted
- Need help with project? Ask/come see us!
- Class evaluations
Tool of the day: 3D Visualization

Parallel Patterns
3D vis demo time
Software links:

- **libsilo** (LLNL “WCI”, BSD license)
- **VisIt** (LLNL “WCI”, BSD license)

Alternative:

- **Paraview** (KitWare/LANL, BSD license)
- TecPlot ($$$)
Tool of the day: 3D Visualization

Parallel Patterns
- Partition
  - Obtaining partitions
- Pipelines
- Reduction
- Map-Reduce
- Scan
- Divide-and-Conquer
- General Data Dependencies
Tool of the day: 3D Visualization

Parallel Patterns
  Partition
    Obtaining partitions
  Pipelines
  Reduction
  Map-Reduce
  Scan
  Divide-and-Conquer
  General Data Dependencies
Partition

\[ y_i = f_i(x_{i-1}, x_i, x_{i+1}) \]

where \( i \in \{1, \ldots, N\} \).
\[ y_i = f_i(x_{i-1}, x_i, x_{i+1}) \]

where \( i \in \{1, \ldots, N\} \).

Includes straightforward generalizations to dependencies on a larger (but not \( O(P) \)-sized!) set of neighbor inputs.
\[ y_i = f_i(x_{i-1}, x_i, x_{i+1}) \]

where \( i \in \{1, \ldots, N\} \).

Includes straightforward generalizations to dependencies on a larger (but not \( O(P) \)-sized!) set of neighbor inputs.

**Point:** Processor \( i \) owns \( x_i \). ("owns" = is "responsible for updating")
Partition: Graph

Visualization Parallel Patterns
Mapping to Mechanisms

- OpenMP?
Mapping to Mechanisms

- OpenMP?
- MPI?
Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than \# ranks?
Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than \# ranks?
- GPU?
Mapping to Mechanisms: Stencils

Common example ("5-point stencil"): \[ u_{i,j}^{n+1} = \frac{1}{h^2} (-4u_{i,j}^n + u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n) \]
Mapping to Mechanisms: Stencils

Common example ("5-point stencil"): 

\[ u_{i,j}^{n+1} = \frac{1}{h^2} \left( -4u_{i,j}^n + u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n \right) \]

- Sequential

Visualization Parallel Patterns
Mapping to Mechanisms: Stencils

Common example ("5-point stencil"): 

\[ u_{i,j}^{n+1} = \frac{1}{h^2} ( -4u_{i,j}^n + u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n ) \]

- Sequential
- OpenMP?
- MPI?
- GPU — 2D?
- GPU — 3D?

What if there's geometry?

Visualization Parallel Patterns
Common example ("5-point stencil"): 

$$u_{i,j}^{n+1} = \frac{1}{h^2} \left(-4u_{i,j}^n + u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n\right)$$

- Sequential
- OpenMP?
- MPI?
Mapping to Mechanisms: Stencils

Common example ("5-point stencil"): \[
u_{i,j}^{n+1} = \frac{1}{h^2}(-4u_{i,j}^n + u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n)
\]

- Sequential
- OpenMP?
- MPI?
- GPU — 2D?
Common example ("5-point stencil"): 

\[
\begin{align*}
\frac{u_{i,j}^{n+1}}{h^2} &= -4u_{i,j}^{n} + u_{i-1,j}^{n} + u_{i+1,j}^{n} + u_{i,j-1}^{n} + u_{i,j+1}^{n} \\
\end{align*}
\]

- Sequential
- OpenMP?
- MPI?
- GPU — 2D?
- GPU — 3D?
Mapping to Mechanisms: Stencils

Common example ("5-point stencil"): 

\[
\frac{u_{i,j}^{n+1}}{h^2} = \frac{1}{h^2}(-4u_{i,j}^n + u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n)
\]

- Sequential
- OpenMP?
- MPI?
- GPU — 2D?
- GPU — 3D?

What if there's geometry?
Partition: Issues

- Same computation often repeated many times
  - As time steps in a simulation
  - Until ‘convergence’
- → Synchronization?
- Main structures: Array (image, grid), Graph (mesh)
- Performance impact of partition?
- Granularity?
- Only useful when the computation is mainly local
- Load Balancing: Thorny issue (next)
Rendezvous Trick

- Assume an irregular partition.
- Assume problem components \(i, j\) on unknown partitions \(p_i, p_j\) need to communicate.
- How can \(p_i\) find \(p_j\) (and vice versa)?

Communicate via a third party, \(p_f(i, j)\).

For \(f\): think ‘hash function’.

"I'm in \(p_i\)."

"I'm in \(p_j\)."
Rendezvous Trick

- Assume an irregular partition.
- Assume problem components $i, j$ on unknown partitions $p_i, p_j$ need to communicate.
- How can $p_i$ find $p_j$ (and vice versa)?

Communicate via a third party, $p_f(i, j)$.

For $f$: think ‘hash function’.
Rendezvous Trick

- Assume an irregular partition.
- Assume problem components $i, j$ on unknown partitions $p_i, p_j$ need to communicate.
- How can $p_i$ find $p_j$ (and vice versa)?

Communicate via a third party, $p_f(i,j)$.
For $f$: think ‘hash function’.
Rendezvous Trick

- Assume an irregular partition.
- Assume problem components $i, j$ on unknown partitions $p_i, p_j$ need to communicate.
- How can $p_i$ find $p_j$ (and vice versa)?

Communicate via a third party, $pf(i,j)$.

For $f$: think ‘hash function’.
Rendezvous Trick

- Assume an irregular partition.
- Assume problem components $i, j$ on unknown partitions $p_i, p_j$ need to communicate.
- How can $p_i$ find $p_j$ (and vice versa)?

Communicate via a third party, $p_f(i,j)$.

For $f$: think ‘hash function’.
Rendezvous Trick

- Assume an irregular partition.
- Assume problem components $i$, $j$ on unknown partitions $p_i$, $p_j$ need to communicate.
- How can $p_i$ find $p_j$ (and vice versa)?

Communicate via a third party, $p_{f(i,j)}$.

For $f$: think ‘hash function’.
Example

Finite difference method.
- Assign equal numbers of grid points to processors.
- Keep amount of data communicated small.

7x5 grid
5-point stencil
4 processors
Example

Finite difference method.

- Assign equal numbers of grid points to processors.
- Keep amount of data communicated small.

Max Data Comm: 14
Total Volume: 42
Max Nbor Proc: 2
Max Imbalance: 3%

First 35/4 points to processor 0; next 35/4 points to processor 1; etc.

E. Boman, K. Devine (Sandia)
Example

- Finite difference method.
  - Assign equal numbers of grid points to processors.
  - Keep amount of data communicated small.

Max Data Comm: 10
Total Volume: 30
Max Nbor Proc: 2
Max Imbalance: 14%

One-dimensional striped partition

E. Boman, K. Devine (Sandia)
A simple strategy

Recursive Coordinate Bisection (‘RCB’) [Berger, Bokhari ‘87]

Simple
Easy to update for changed geometry (‘incremental’)
Easy to fool

E. Boman, K. Devine (Sandia)
Space-filling curves

Hilbert curve
Space-filling curves

Morton curve ("Z curve")

Easily obtained by bit interleaving!
Space-filling curves

Carlo H. Sequin, UC Berkeley / Wikipedia
Space-filling curves

Simple, even for adaptive meshes
Weight-able
Cache-happy
Easy to update for changed geometry (‘incremental’)
Communication volume?

M. Berger
Space-filling curves: Examples

M. Berger, M. Aftosmis
Space-filling curves: Examples

M. Berger, M. Aftosmis
Partitioning: Objectives

Main goals:
- Even distribution of work
- Minimize neighbor communication

Criteria:
- Cheap! (General problem: NP-complete)
- Incremental
- Partitioning itself is parallel
Partitioning: Objectives

Main goals:
- Even distribution of work
- Minimize neighbor communication

Criteria:
- Cheap! (General problem: NP-complete)
- Incremental
- Partitioning itself is parallel

What if we don’t have geometry/coordinates?
Chopping up the communication graph

Here is a partitioned graph with an edge-cut of seven. Three sub domains, A, B, and C, are shown. The edge-cut of the three-way partitioning is seven. During parallel computation, the processor corresponding to sub domain A will need to send the data for vertices 1 and 3 to the processor corresponding to sub domain B, and the data for vertex 4 to the processor corresponding to sub domain C. Similarly, B needs to send the data for vertices 5 and 7 to A and the data for vertices 7 and 8 to C. Finally, C needs to send the data for vertices 9 to B and the data for vertex 10 to A. This equals nine units of data to be sent, while the edge-cut is seven. The reason that edge-cut and total communication volume are not the same is because the edge-cut counts every edge cut, while data is required to be sent only once if two or more edges of a single vertex are cut by the same sub domain (as is the case, for example, between vertex 3 and sub domain B in Figure 3). It should also be noted that total communication volume alone cannot accurately predict inter-processor communication overhead. A more precise measure is the maximum time required by any of the processors to perform communication (because computation and communication occur in alternating phases). This depends on a number of factors, including the amount of data to be sent out of any one processor, as well as the number of processors with which a processor must communicate. In particular, on message-passing architectures, minimizing the maximum number of message startups that any one processor must perform can sometimes be more important than minimizing the communications volume. Nevertheless, there still tends to be a strong correlation between edge-cuts and inter-processor communication costs for graphs of uniform degree (i.e., graphs in which most vertices have about the same number of edges). This is a typical characteristic of graphs derived from scientific simulations. For this reason, the traditional min-cut partitioning problem is a reasonable approximation to the problem of minimizing the inter-processor communications that underly many scientific simulations.

Computing a k-way partitioning via Recursive Bisection

Graphs are frequently partitioned into k sub domains by recursively computing two-way partitionings (i.e., bisections) of the graph. This method requires the computation of k/2 bisections. If k is not a power of two, then for each bisection, the appropriate sub domain weights need to be specified in order to ensure that the resulting k-way partitioning is balanced.

It is known that for a large class of graphs derived from scientific simulations, recursive bisection algorithms are able to compute k-way partitionings that are within a constant factor of the optimal solution. Furthermore, if the balance constraint is sufficiently relaxed, then recursive bisection methods can be used to compute k-way partitionings that are within log p of the optimal for all graphs. Since the direct computation of a good k-way partitioning is harder in general than the computation of a good bisection (although both problems are NP-complete), recursive bisection has become a widely used technique.
Chopping up the communication graph

CONTENTS

1
2
4
3
5
7 8
6
10
9
11
12
B

A

C

Figure 3: A partitioned graph with an edge-cut of seven. Here, nine communications are incurred during parallel processing. In Figure 3, three sub domains, A, B, and C are shown. The edge-cut of the (three-way) partitioning is seven. During parallel computation, the processor corresponding to sub domain A will need to send the data for vertex 1 and 3 to the processor corresponding to sub domain B, and the data for vertex 4 to the processor corresponding to sub domain C. Similarly, B needs to send the data for vertex 5 and 7 to A and the data for vertex 7 and 8 to C. Finally, C needs to send the data for vertex 9 to B and the data for vertex 10 to A. This equals nine units of data to be sent, while the edge-cut is seven. The reason that edge-cut and total communication volume are not the same is because the edge-cut counts every edge cut, while data is required to be sent only one time if two or more edges of a single vertex are cut by the same sub domain (as is the case, for example, between vertex 3 and sub domain B in Figure 3). It should also be noted that total communication volume alone cannot accurately predict inter-processor communication overhead. A more precise measure is the maximum time required by any of the processors to perform communication (because computation and communication occur in alternating phases). This depends on a number of factors, including the amount of data to be sent out of any one processor, as well as the number of processors with which a processor must communicate. In particular, on message-passing architectures, minimizing the maximum number of message startups that any one processor must perform can sometimes be more important than minimizing the communications volume. Nevertheless, there still tends to be a strong correlation between edge-cuts and inter-processor communication costs for graphs of uniform degree (i.e., graphs in which most vertices have about the same number of edges). This is a typical characteristic of graphs derived from scientific simulations. For this reason, the traditional min-cut partitioning problem is a reasonable approximation to the problem of minimizing the inter-processor communications that underly many scientific simulations.

Computing a k-way partitioning via Recursive Bisection

Graphs are frequently partitioned into k sub domains by recursively computing two-way partitionings (i.e., bisections) of the graph. This method requires the computation of k^2/1 bisections. If k is not a power of two, then for each bisection, the appropriate sub domain weights need to be specified in order to ensure that the resulting k-way partitioning is balanced.

It is known that for a large class of graphs derived from scientific simulations, recursive bisection algorithms are able to compute k-way partitionings that are within a constant factor of the optimal solution. Furthermore, if the balance constraint is sufficiently relaxed, then recursive bisection methods can be used to compute k-way partitionings that are within log p of the optimal for all graphs. Since the direct computation of a good k-way partitioning is harder in general than the computation of a good bisection (although both problems are NP-complete), recursive bisection has become a widely used technique.

Static Graph Partitioning Techniques

The graph partitioning problem is known to be NP-complete. Therefore, in general it is not possible to compute optimal partitionings for graphs of interesting size in a reasonable amount of time. This fact, combined...
Chopping up the communication graph

Figure 3: A partitioned graph with an edge-cut of seven. Here, nine communications are incurred during parallel processing.

In Figure 3, three subdomains, A, B, and C are shown. The edge-cut of the (three-way) partitioning is seven. During parallel computation, the processor corresponding to subdomain A will need to send the data for vertices 1 and 3 to the processor corresponding to subdomain B, and the data for vertex 4 to the processor corresponding to subdomain C. Similarly, B needs to send the data for 5 and 7 to A and the data for 7 and 8 to C. Finally, C needs to send the data for 9 to B and the data for 10 to A. This equals nine units of data to be sent, while the edge-cut is seven. The reason that edge-cut and total communication volume are not the same is because the edge-cut counts every edge cut, while data is required to be sent only one time if two or more edges of a single vertex are cut by the same subdomain (as is the case, for example, between vertex 3 and subdomain B in Figure 3). It should also be noted that total communication volume alone cannot accurately predict inter-processor communication overhead. A more precise measure is the maximum time required by any of the processors to perform communication (because computation and communication occur in alternating phases). This depends on a number of factors, including the amount of data to be sent out of any one processor, as well as the number of processors with which a processor must communicate. In particular, on message-passing architectures, minimizing the maximum number of message startups that any one processor must perform can sometimes be more important than minimizing the communications volume. Nevertheless, there still tends to be a strong correlation between edge-cuts and inter-processor communication costs for graphs of uniform degree (i.e., graphs in which most vertices have about the same number of edges). This is a typical characteristic of graphs derived from scientific simulations. For this reason, the traditional min-cut partitioning problem is a reasonable approximation to the problem of minimizing the inter-processor communications that underly many scientific simulations.

Computing a $k$-way partitioning via Recursive Bisection

Graphs are frequently partitioned into $k$ subdomains by recursively computing two-way partitionings (i.e., bisections). This method requires the computation of $k^2/1$ bisections. If $k$ is not a power of two, then for each bisection, the appropriate subdomain weights need to be specified in order to ensure that the resulting $k$-way partitioning is balanced.

It is known that for a large class of graphs derived from scientific simulations, recursive bisection algorithms are able to compute $k$-way partitionings that are within a constant factor of the optimal solution. Furthermore, if the balance constraint is sufficiently relaxed, then recursive bisection methods can be used to compute $k$-way partitionings that are within $\log p$ of the optimal for all graphs. Since the direct computation of a good $k$-way partitioning is harder in general than the computation of a good bisection (although both problems are NP-complete), recursive bisection has become a widely used technique.

Great model? How often do we send vertex 1 to B?

Perhaps: assign weight to vertices, edges
Spectral partitioning demo
Metis demo
Finer points

- What if \( \# \text{ inputs} \neq \# \text{ outputs} \)?
- Might want to balance multiple objectives
  - Types of work
  - Types of communication

Software packages to look for:
- Zoltan (free, LGPL)
- PT-Scotch (free, copyleft)
- Metis (free to use, proprietary, some source available)
Finer points

- What if \# inputs \neq \# outputs? (\rightarrow hypergraphs)
- Might want to balance multiple objectives
  - Types of work
  - Types of communication

Software packages to look for:
- Zoltan (free, LGPL)
- PT-Scotch (free, copyleft)
- Metis (free to use, proprietary, some source available)
Tool of the day: 3D Visualization

Parallel Patterns
  Partition
    Obtaining partitions
  Pipelines
  Reduction
  Map-Reduce
  Scan
  Divide-and-Conquer
  General Data Dependencies
Pipelined Computation

\[ y = f_N(\cdots f_2(f_1(x))\cdots) \]
\[ = (f_N \circ \cdots \circ f_1)(x) \]

where \( N \) is fixed.
Pipelined Computation: Graph

Visualization Parallel Patterns
Pipelined Computation: Graph

Processor Assignment?
Pipelined Computation: Examples

- Image processing
- Any multi-stage algorithm
  - Pre/post-processing or I/O
- Out-of-Core algorithms

Specific simple examples:
- Sorting (insertion sort)
- Triangular linear system solve (‘backsubstitution’)
  - Key: Pass on values as soon as they’re available

(Will see more efficient algorithms for both later)
Pipelined Computation: Issues

- Non-optimal while pipeline fills or empties
- Often communication-inefficient
  - for large data
- Needs some attention to synchronization, deadlock avoidance
- Can accommodate some asynchrony
  But don’t want:
  - Pile-up
  - Starvation
Mapping to Mechanisms

- OpenMP?
Mapping to Mechanisms

- OpenMP?
- MPI?
Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than \# ranks?
Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than # ranks?
- GPU?
Tool of the day: 3D Visualization

Parallel Patterns
  Partition
    Obtaining partitions
  Pipelines
  Reduction
  Map-Reduce
  Scan
  Divide-and-Conquer
  General Data Dependencies
Reduction

\[ y = f(\cdots f(f(x_1, x_2), x_3), \ldots, x_N) \]

where \( N \) is the input size.
Reduction

\[ y = f(\cdots f(f(x_1, x_2), x_3), \ldots, x_N) \]

where \( N \) is the input size.

Also known as...

- Lisp/Python function `reduce` (Scheme: `fold`)
- C++ STL `std::accumulate`
Reduction: Graph

Painful! Not parallelizable.

Visualization Parallel Patterns
Reduction: Graph

Painful! Not parallelizable.
Approach to Reduction

Can we do better?

“Tree” very imbalanced. What property of $f$ would allow ‘rebalancing’?

$$f(x, y)$$

$$(x \circ y) = (x \circ (y \circ z))$$

Looks less improbable if we let $x \circ y = f(x, y)$:

$$(x \circ (y \circ z)) = ((x \circ y) \circ z)$$

Has a very familiar name: Associativity
Approach to Reduction

Can we do better?

“Tree” very imbalanced. What property of $f$ would allow ‘rebalancing’?

\[
f(f(x, y), z) = f(x, f(y, z))
\]

Looks less improbable if we let $x \circ y = f(x, y)$:

\[
x \circ (y \circ z)) = (x \circ y) \circ z
\]

Has a very familiar name: **Associativity**
Reduction: A Better Graph

[x₀] → [x₁] → [x₂] → [x₃] → [x₄] → [x₅] → [x₆] → [x₇] → [y]
Reduction: A Better Graph

Processor allocation?
Mapping to Mechanisms

- Single threads?
Mapping to Mechanisms

- Single threads?
- OpenMP?
Mapping to Mechanisms

- Single threads?
- OpenMP?
- MPI?
Mapping to Mechanisms

- Single threads?
- OpenMP?
- MPI?
- MPI: Larger than # ranks?
Mapping to Mechanisms

- Single threads?
- OpenMP?
- MPI?
- MPI: Larger than # ranks?
- GPU?
Mapping Reduction to the GPU

- Obvious: Want to use tree-based approach.
- Problem: Two scales, Work group and Grid
  - Need to occupy both to make good use of the machine.
- In particular, need synchronization after each tree stage.
Mapping Reduction to the GPU

- Obvious: Want to use tree-based approach.
- Problem: Two scales, Work group and Grid
  - Need to occupy both to make good use of the machine.
- In particular, need synchronization after each tree stage.
- Solution: Use a two-scale algorithm.

In particular: Use multiple grid invocations to achieve inter-workgroup synchronization.

With material by M. Harris (Nvidia Corp.)
__kernel void reduce0(__global T *g_idata, __global T *g_odata, unsigned int n, __local T* ldata)
{
    unsigned int lid = get_local_id(0);
    unsigned int i = get_global_id(0);

    ldata[lid] = (i < n) ? g_idata[i] : 0;
    barrier(CLK_LOCAL_MEM_FENCE);

    for(unsigned int s=1; s < get_local_size(0); s *= 2)
    {
        if ((lid % (2*s)) == 0)
            ldata[lid] += ldata[lid + s];
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if (lid == 0) g_odata[get_group_id(0)] = ldata[0];
}
Interleaved Addressing

Values (shared memory)

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 1</td>
<td>0 2 4 6 8 10 12 14</td>
<td>11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 2</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 2</td>
<td>0 4 8 12</td>
<td>18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 3</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 4</td>
<td>0 8</td>
<td>24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 4</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 8</td>
<td>0</td>
<td>41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2</td>
</tr>
</tbody>
</table>

With material by M. Harris (Nvidia Corp.)
Interleaved Addressing

Values (shared memory)

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>1</th>
<th>8</th>
<th>-1</th>
<th>0</th>
<th>-2</th>
<th>3</th>
<th>5</th>
<th>-2</th>
<th>-3</th>
<th>2</th>
<th>7</th>
<th>0</th>
<th>11</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
</table>

**Step 1**
- Stride 1
- Thread IDs: 0, 2, 4, 6, 8, 10, 12, 14
- Values: 11, 1, 7, -1, -2, -2, 8, 5, -5, -3, 9, 7, 11, 11, 2, 2

**Step 2**
- Stride 2
- Thread IDs: 0, 4, 8, 12
- Values: 18, 1, 7, -1, 6, -2, 8, 5, 4, -3, 9, 7, 13, 11, 2, 2

**Step 3**
- Stride 4
- Thread IDs: 0, 8
- Values: 24, 1, 7, -1, 6, -2, 8, 5, 17, -3, 9, 7, 13, 11, 2, 2

**Step 4**
- Stride 8
- Thread IDs: 0
- Values: 41, 1, 7, -1, 6, -2, 8, 5, 17, -3, 9, 7, 13, 11, 2, 2

**Issue:** Slow modulo, Divergence

With material by M. Harris (Nvidia Corp.)
__kernel void reduce2(__global T *g_idata, __global T *g_odata, unsigned int n, __local T *ldata) {

    unsigned int lid = get_local_id(0);
    unsigned int i = get_global_id(0);

    ldata[lid] = (i < n) ? g_idata[i] : 0;
    barrier(CLK_LOCAL_MEM_FENCE);

    for(unsigned int s = get_local_size(0)/2; s > 0; s >>= 1) {
        if (lid < s)
            ldata[lid] += ldata[lid + s];
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if (lid == 0) g_odata[get_local_size(0)] = ldata[0];
}
Sequential Addressing

Values (shared memory)

<table>
<thead>
<tr>
<th>Values (shared memory)</th>
<th>10</th>
<th>1</th>
<th>8</th>
<th>-1</th>
<th>0</th>
<th>-2</th>
<th>3</th>
<th>5</th>
<th>-2</th>
<th>-3</th>
<th>2</th>
<th>7</th>
<th>0</th>
<th>11</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
</table>

Step 1
Stride 8
Thread IDs
0 → 1
Values
8 → 10 → 6 → 0 → 9 → 3 → 7 → -2 → -3 → 2 → 7 → 0 → 11 → 0 → 2

Step 2
Stride 4
Thread IDs
0 → 1
Values
8 → 7 → 13 → 13 → 0 → 9 → 3 → 7 → -2 → -3 → 2 → 7 → 0 → 11 → 0 → 2

Step 3
Stride 2
Thread IDs
0 → 1
Values
21 → 20 → 13 → 13 → 0 → 9 → 3 → 7 → -2 → -3 → 2 → 7 → 0 → 11 → 0 → 2

Step 4
Stride 1
Thread IDs
0
Values
41 → 20 → 13 → 13 → 0 → 9 → 3 → 7 → -2 → -3 → 2 → 7 → 0 → 11 → 0 → 2

Better!
But still not “efficient.”
Only half of all work items after first round, then a quarter, . . .

With material by M. Harris (Nvidia Corp.)
**Sequential Addressing**

![Diagram of Sequential Addressing]

Better! But still not “efficient”.

Only half of all work items after first round, then a quarter, ...
Recap: Parallel Complexity

Distinguish:

- **Time on** $T$ processors: $T_P$
- **Step Complexity/Span** $T_\infty$: Minimum number of steps taken if an infinite number of processors are available
- Work per step $S_t$
- **Work Complexity/Work** $T_1 = \sum_{t=1}^{T_\infty} S_t$: Total number of operations performed
- **Parallelism** $T_1 / T_\infty$: average amount of work along span
  - $P > T_1 / T_\infty$ doesn't make sense.

Algorithm-specific!
Number of Items $N$
Actual work to be done: $W = O(N)$ additions.

Step Complexity: Let $d = \lceil \log_2 N \rceil$. Then $T_{\infty} = d$, $S_t = O(2^{d-t})$.

Work Complexity:

$$T_1 = \sum_{t=1}^{T} S_t = O \left( \sum_{t=1}^{T} 2^{d-t} \right) = O(2^d) = O(N)$$
Number of Items $N$
Actual work to be done: $W = O(N)$ additions.

Step Complexity: Let $d = \lceil \log_2 N \rceil$. Then $T_\infty = d$, $S_t = O(2^{d-t})$.

Work Complexity:

$$T_1 = \sum_{t=1}^{T} S_t = O \left( \sum_{t=1}^{T} 2^{d-t} \right) = O(2^d) = O(N)$$

“Work-efficient:” $T_1 \sim W$. 
Greedy Scheduling

Theorem (Graham ‘68, Brent ‘75)
A parallel algorithm with span $T_\infty$ and work complexity $T_1$ can be executed on a shared-memory machine with $P$ processors in no more than

$$T_P \leq \frac{T_1}{P} + T_\infty$$

steps.

Observations:

- Think of $T_\infty$ as the length of the “critical path”.
- The first summand can be made to go away by increasing $P$.
- Only valid for shared-memory.
Greedy Scheduling

Theorem (Graham ‘68, Brent ‘75)
A parallel algorithm with span $T_\infty$ and work complexity $T_1$ can be executed on a shared-memory machine with $P$ processors in no more than

$$T_P \leq \frac{T_1}{P} + T_\infty$$

steps.

Observations:
- Think of $T_\infty$ as the length of the "critical path".
- The first summand can be made to go away by increasing $P$.
- Only valid for shared-memory.

Estimate for $P = 1$?
Proof sketch?
What about reduction?
What is $P$ for a GPU?
```c
__kernel void reduce6(__global T *g_idata, __global T *g_odata, unsigned int n, volatile __local T* ldata) {

    unsigned int lid = get_local_id(0);
    unsigned int i = get_group_id(0) * (get_local_size(0) * 2) + get_local_id(0);
    unsigned int gridSize = GROUP_SIZE * 2 * get_num_groups(0);
    ldata[lid] = 0;

    while (i < n) {
        ldata[lid] += g_idata[i];
        if (i + GROUP_SIZE < n)
            ldata[lid] += g_idata[i + GROUP_SIZE];
        i += gridSize;
    }

    barrier(CLK_LOCAL_MEM_FENCE);
}
```

**Kernel V3 Part 1**
if (GROUP_SIZE >= 512) {
    if (lid < 256) { ldata[lid] += ldata[lid + 256]; }
    barrier (CLK_LOCAL_MEM_FENCE);
}
// ...
if (GROUP_SIZE >= 128) {
    /* ... */
}

if (lid < 32) {
    if (GROUP_SIZE >= 64) { ldata[lid] += ldata[lid + 32]; }
    if (GROUP_SIZE >= 32) { ldata[lid] += ldata[lid + 16]; }
    // ...
    if (GROUP_SIZE >= 2) { ldata[lid] += ldata[lid + 1]; }
}

if (lid == 0) g_odata[get_group_id(0)] = ldata[0];
Performance Comparison

With material by M. Harris (Nvidia Corp.)
Reduction: Examples

- Sum, Inner Product, Norm
  - Occurs in iterative methods
- Minimum, Maximum
- Data Analysis
  - Evaluation of Monte Carlo Simulations
- List Concatenation, Set Union
- Matrix-Vector product (but...)
Reduction: Issues

• When adding: floating point cancellation?
• Serial order goes faster: can use registers for intermediate results
• Requires availability of neutral element
• GPU-Reduce: Optimization sensitive to data type
Tool of the day: 3D Visualization

Parallel Patterns

Partition
  Obtaining partitions
Pipelines
Reduction
Map-Reduce
Scan
Divide-and-Conquer
General Data Dependencies
Map-Reduce

Sounds like this:

\[ y = f(\cdots f(f(g(x_1), g(x_2)), g(x_3)), \ldots, g(x_N)) \]

where \( N \) is the input size.

- Lisp naming, again
- Mild generalization of reduction
Map-Reduce

But no. Not even close.

Sounds like this:

\[ y = f(\cdots f(f(g(x_1), g(x_2)), g(x_3)), \ldots, g(x_N)) \]

where \( N \) is the input size.

- Lisp naming, again
- Mild generalization of reduction
Map-Reduce: Graph

Visualization Parallel Patterns
MapReduce: Discussion

MapReduce $\geq$ map + reduce:

- Used by Google (and many others) for large-scale data processing
- Map generates (key, value) pairs
  - Reduce operates only on pairs with identical keys
  - Remaining output sorted by key
- Represent all data as character strings
  - User must convert to/from internal repr.
- Messy implementation
  - Parallelization, fault tolerance, monitoring, data management, load balance, re-run “stragglers”, data locality
- Works for Internet-size data
- Simple to use even for inexperienced users
MapReduce: Examples

- String search
- (e.g. URL) Hit count from Log
- Reverse web-link graph
  - desired: (target URL, sources)
- Sort
- Indexing
  - desired: (word, document IDs)
- Machine Learning, Clustering, ...
Outline

Tool of the day: 3D Visualization

Parallel Patterns

- Partition
  - Obtaining partitions
- Pipelines
- Reduction
- Map-Reduce
- Scan
- Divide-and-Conquer
- General Data Dependencies
Scan

\[
\begin{align*}
  y_1 &= x_1 \\
  y_2 &= f(y_1, x_2) \\
  &\vdots \quad \vdots \\
  y_N &= f(y_{N-1}, x_N)
\end{align*}
\]

where \( N \) is the input size. (Think: \( N \) large, \( f(x, y) = x + y \))

- Prefix Sum/Cumulative Sum
- Abstract view of: loop-carried dependence
- Also possible: Segmented Scan
This can't possibly be parallelized. Or can it?
This can’t possibly be parallelized. Or can it?
This can’t possibly be parallelized. Or can it? Again: Need assumptions on $f$. Associativity, commutativity.
Scan: Implementation

Visualization Parallel Patterns
Scan: Implementation

Work-efficient?
Scan: Implementation II

Two sweeps: Upward, downward, both tree-shape

On upward sweep:
- Get values L and R from left and right child
- Save L in local variable Mine
- Compute \( Tmp = L + R \) and pass to parent

On downward sweep:
- Get value \( Tmp \) from parent
- Send \( Tmp \) to left child
- Sent \( Tmp + Mine \) to right child
Two sweeps: Upward, downward, both tree-shape

On upward sweep:
- Get values $L$ and $R$ from left and right child
- Save $L$ in local variable $\text{Mine}$
- Compute $\text{Tmp} = L + R$ and pass to parent

On downward sweep:
- Get value $\text{Tmp}$ from parent
- Send $\text{Tmp}$ to left child
- Send $\text{Tmp} + \text{Mine}$ to right child

Work-efficient?
Span rel. to first attempt?
Scan: Examples

- Anything with a loop-carried dependence
- One row of Gauss-Seidel
- One row of triangular solve
- Segment numbering if boundaries are known
- Low-level building block for many higher-level algorithms
- FIR/IIR Filtering
- G.E. Blelloch: Prefix Sums and their Applications
Scan: Issues

- Subtlety: Inclusive/Exclusive Scan
- Pattern sometimes hard to recognize
  - But shows up surprisingly often
  - Need to prove associativity/commutativity
- Useful in Implementation: algorithm cascading
  - Do sequential scan on parts, then parallelize at coarser granularities
Mapping to Mechanisms

- OpenMP?
Mapping to Mechanisms

- OpenMP?
- MPI?
Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than \# ranks?
Mapping to Mechanisms

• OpenMP?
• MPI?
• MPI: Larger than number of ranks?
• GPU?
Sort (fixed-size) integers using scan
Tool of the day: 3D Visualization

Parallel Patterns
  Partition
    Obtaining partitions
  Pipelines
  Reduction
  Map-Reduce
  Scan
  Divide-and-Conquer
  General Data Dependencies
Divide and Conquer

\[ y_i = f_i(x_1, \ldots, x_N) \]

for \( i \in \{1, \ldots, M\} \).

**Main purpose:** A way of partitioning up fully dependent tasks.
Divide and Conquer

\[ y_i = f_i(x_1, \ldots, x_N) \]

for \( i \in \{1, \ldots, M\} \).

**Main purpose:** A way of partitioning up fully dependent tasks.

Processor allocation?
Divide and Conquer: Examples

- GEMM, TRMM, TRSM, GETRF (LU)
- FFT
- Sorting: Bucket sort, Merge sort
- \( N \)-Body problems (Barnes-Hut, FMM)
- Adaptive Integration

More fun with work and span:

D&C analysis lecture
Mapping to Mechanisms

- OpenMP?
Mapping to Mechanisms

- OpenMP?
- MPI?
Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than \# ranks?
Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than \# ranks?
- GPU?
Divide and Conquer: Issues

- “No idea how to parallelize that”
  - → Try D&C
- Non-optimal during partition, merge
  - But: Does not matter if deep levels do heavy enough processing
- Subtle to map to fixed-width machines (e.g. GPUs)
  - Varying data size along tree → Scan!
- Bookkeeping nontrivial for non-\(2^n\) sizes
- Side benefit: D&C is generally cache-friendly
Tool of the day: 3D Visualization

Parallel Patterns

- Partition
  - Obtaining partitions
- Pipelines
- Reduction
- Map-Reduce
- Scan
- Divide-and-Conquer

General Data Dependencies
B = f(A)
C = g(B)
E = f(C)
F = h(C)
G = g(E,F)
P = p(B)
Q = q(B)
R = r(G,P,Q)

Great: All patterns discussed so far can be reduced to this one.
B = f(A)
C = g(B)
E = f(C)
F = h(C)
G = g(E,F)
P = p(B)
Q = q(B)
R = r(G,P,Q)

Great: All patterns discussed so far can be reduced to this one.
Mapping to Mechanisms

- OpenMP?
Mapping to Mechanisms

- OpenMP?
- MPI?
Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than \# ranks?
Mapping to Mechanisms

- OpenMP?
- MPI?
- MPI: Larger than \# ranks?
- GPU?
cilk int fib (int n) {
    if (n < 2) return n;
    else {
        int x, y;

        x = spawn fib (n−1);
        y = spawn fib (n−2);

        sync;

        return (x+y);
    }
}

Features:

- Adds keywords spawn, sync, (inlet, abort)
- Remove keywords → valid (seq.) C

Timeline:

- Developed at MIT, starting in ‘94
- Commercialized in ‘06
- Bought by Intel in ‘09
- Available in the Intel Compilers
Cilk

cilk int fib (int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib (n−1);
    y = spawn fib (n−2);
    sync;
    return (x+y);
  }
}

Features:

- Adds keywords spawn, sync, (inlet, abort)
- Remove keywords → valid (seq.) C

Timeline:

- Developed at MIT, starting in ‘94
- Commercialized in ‘06
- Bought by Intel in ‘09
- Available in the Intel Compilers

Efficient implementation?
Each processor maintains a **work deque** of ready threads, and it manipulates the bottom of the deque like a stack.
Each processor maintains a work deque of ready threads, and it manipulates the bottom of the deque like a stack.

With material by Charles E. Leiserson (MIT)
Each processor maintains a *work deque* of ready threads, and it manipulates the bottom of the deque like a stack.
Work-Stealing

Each processor maintains a *work deque* of ready threads, and it manipulates the bottom of the deque like a stack.

With material by Charles E. Leiserson (MIT)
Work-Stealing

Each processor maintains a work deque of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it steals a thread from the top of a random victim’s deque.

With material by Charles E. Leiserson (MIT)
Work-Stealing

Each processor maintains a *work deque* of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it *steals* a thread from the top of a *random* victim’s deque.

With material by Charles E. Leiserson (MIT)
Work-Stealing

Each processor maintains a *work deque* of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it *steals* a thread from the top of a *random* victim’s deque.

With material by Charles E. Leiserson (MIT)
Work-Stealing

Each processor maintains a *work deque* of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it *steals* a thread from the top of a *random* victim’s deque.

With material by Charles E. Leiserson (MIT)
Work-Stealing

Each processor maintains a **work deque** of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it steals a thread from the top of a random victim's deque.

Why is Work-Stealing better than a Task Queue?
General Graphs: Implementations

- Intel Cilk(+) (also: vector math)
- OpenCL (“Events”, Out-of-order queues)
- Intel Thread Building Blocks
- StarPU
- (Charm++)
- Many more
General Graphs: Issues

- Model can accommodate ‘speculative execution’
  - Launch many different ‘approaches’
  - Abort the others as soon as one satisfactory one emerges.
- Discover dependencies, make up schedule at run-time
  - Usually less efficient than the case of known dependencies
  - Map-Reduce absorbs many cases that would otherwise be general
- On-line scheduling: complicated
- Not a good fit if a more specific pattern applies
- Good if inputs/outputs/functions are (somewhat) heavy-weight
Questions?
Image Credits

- Pipe: sxc.hu/mterraza
- Tree: sxc.hu/bertvthul
- Radar: sxc.hu/KimPouss
- Quadtree: flickr.com/ethanhein [CC]