Lecture 8:
Parallel Programming
Case Studies

Parallel Computer Architecture and Programming
CMU 15-418, Spring 2013
Today

- Parallel application case studies!

- Five examples
  - Ocean
  - Galaxy simulation (Barnes-Hut algorithm)
  - Parallel scan
  - Data-parallel segmented scan (Bonus material!)
  - Ray tracing (Bonus material!)

- Will be describing key aspects of the implementations
  - Focus on: optimization techniques, analysis of workload characteristics
Assumption: shared address space

- For the purposes of today’s lecture I encourage you to think about the example applications in the context of a large NUMA shared address space machine.
  
  (single address space, but each processor can access a local region of the address space more quickly)

- Issues we discuss also arise in a distributed address space setting.
Example 1: simulation of ocean currents

- Discretize ocean into slices represented as 2D grids
- Discretize time evolution: $\Delta t$
- High accuracy simulation requires small $\Delta t$ and high resolution grids
First question to ask: where are the dependencies?

Dependencies in one time step of ocean simulation

Potential parallelism within a grid (data-parallelism) and across operations on the different grids. Implementation only leverages data-parallelism. (simplicity)
Another question to ask: what are the critical working sets?

1. Local neighborhood for cell
2. Three rows of processor’s local partition
3. Processor’s local partition
Ocean implementation details

Recall shared-memory implementation from previous classes:

- **Synchronization**
  - Barriers (each grid computation is a phase)
  - Locks for mutual exclusion when updating shared variables (atomic update of ‘diff’)

- **Assignment**
  - Static assignment via block decomposition
Ocean: execution time breakdown
Execution on 32-processor SGI Origin 2000 (1026x1026 grids)

- Static assignment is sufficient (approximately equal busy time per thread)
- 4D blocking of grid reduces communication (reflected on graph as data wait time)
- Synchronization cost largely due to waiting at barriers
Example 2: Galaxy evolution
Barnes-Hut algorithm

- Represent galaxy as a collection of particles (think: particle = star)
- Compute forces due to gravity
  - Gravity has infinite extent: naive algorithm is $O(N^2)$
  - Magnitude of gravitational force falls off with distance
    (so approximate forces from groups of far away stars)
  - Result is an $O(N \log N)$ algorithm for computing gravitational forces between all stars
Barnes-Hut tree

- Interior nodes store center of mass, aggregate mass of child bodies
- For each body, traverse tree... accumulating forces from all other bodies
  - Compute forces using aggregate interior node if \( \frac{L}{D} < \Theta \), else descend to children
- Expected number of nodes touched \( \sim \log n / \Theta^2 \)
Application structure

- Challenges:
  - Amount of work per body, and communication pattern of work is non-uniform (depends on local density of bodies)
  - The bodies move: so costs and communication patterns change over time
  - Irregular, fine-grained computation

- But, there is a lot of locality in the computation (bodies near in space require similar data to compute forces -- should co-locate these computations!)
Work assignment

- Challenge:
  - Equal number of bodies per processor ≠ equal work per processor
  - Want equal work per processor AND assignment should preserve locality

- Observation: spatial distribution of bodies evolves slowly

- Use semi-static assignment
  - Each time step, for each body, record number of interactions with other bodies (the application profiles itself)
    - Cheap to compute. Just increment local per-body counters
    - Use values to periodically recompute assignment
Assignment using cost zones

- Leverage locality inherent in tree
- Compute total work estimate $W$ for all bodies (computed by summing per-body costs)
- Each processor is assigned $W/P$ of the total work ($P =$ num processors)
- Each processor performs depth-first (post-order) traversal of tree (accumulates work seen so far)
- Processor $P_i$ responsible for processing bodies corresponding to work: $iW/P$ to $(i+1)W/P$
- Each processor can independently compute its assignment of bodies. (The only synchronization required is the reduction to compute total amount of work.)
Barnes-Hut: working sets

- Working set 1: data needed to compute forces between body-body (or body-node) pairs
- Working set 2: data encountered in an entire tree traversal
  - Expected number of nodes touched for one body: $\sim \lg n / \Theta^2$
  - Computation has high locality: consecutively processed bodies are nearby, so its processing touches almost exactly the same nodes!
Barnes-hut: data distribution

- Cost zones technique computes a good work assignment. What about data distribution?

- Difficult to distribute data
  - Work assignment changes with time: would have to dynamically redistribute all simulation data
  - Data accessed at fine granularity (single tree node)

- Luckily: high temporal locality
  - Bodies assigned to same processor are nearby in space, so tree nodes accessed during force computations are very similar.
  - Data for traversal already in cache (Barnes-Hut benefits from large caches, smaller cache line size)

- Result: Unlike OCEAN, data distribution in Barnes-Hut does not significantly impact performance
  - Use static distribution (interleave particles throughout the machine)
Barnes-hut: synchronization

- A few barriers between phases of force computation
- Fine-grained synchronization needed during tree build phase
  - Lock per tree cell
Barnes-hut: execution time

Execution on 32-processor SGI Origin 2000 (512K bodies)

- Load balance is good even with static assignment because of random assignment
  - On average, each processor does approximately the same amount of work

- But random assignment yields poor locality
  - Significant amount of inherent communication
  - Significant amount of artifactual communication (fine-grained accesses to tree nodes)

- Common tension: work balance vs. locality (cost-zones get us both!)
  (similar to work balance vs. synchronization trade-offs in “work distribution” lecture)
Summary

- Today so far: two examples of parallel program optimization
- Key issues when discussing the applications
  - How to balance the work?
  - How to exploit locality inherent in the problem?
  - What synchronization is necessary?
Example 3: data-parallel scan

let \(a = [a_0, a_1, a_2, a_3, \ldots, a_{n-1}]\)

let \(\oplus\) be an associative binary operator with identity element \(I\)

\[
\text{scan\_inclusive}(\oplus, a) = [a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots]
\]

\[
\text{scan\_exclusive}(\oplus, a) = [I, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots]
\]

If operator is +, then \(\text{scan\_inclusive}(+, a)\) is a prefix sum

\[
\text{prefix\_sum}(a) = [a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots]
\]
Data-parallel exclusive scan
(Just subtract original vector to get inclusive scan result)

Work: $O(N \lg N)$
Inefficient compared to sequential algorithm!
Span: $O(\lg N)$
### Work-efficient parallel exclusive scan (O(N) work)

<table>
<thead>
<tr>
<th>a₀</th>
<th>a₁</th>
<th>a₂</th>
<th>a₃</th>
<th>a₄</th>
<th>a₅</th>
<th>a₆</th>
<th>a₇</th>
<th>a₈</th>
<th>a₉</th>
<th>a₁₀</th>
<th>a₁₁</th>
<th>a₁₂</th>
<th>a₁₃</th>
<th>a₁₄</th>
<th>a₁₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>a₂-3</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>a₆-7</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₁₀-11</td>
<td>a₁₂</td>
<td>a₁₂-13</td>
<td>a₁₄</td>
<td>a₁₄-15</td>
</tr>
<tr>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>a₀-3</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>a₄-7</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₈-11</td>
<td>a₁₂</td>
<td>a₁₂-13</td>
<td>a₁₄</td>
<td>a₁₂-15</td>
</tr>
<tr>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>a₀-3</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>a₀-7</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₈-11</td>
<td>a₁₂</td>
<td>a₁₂-13</td>
<td>a₁₄</td>
<td>a₈-15</td>
</tr>
<tr>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>a₀-3</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>a₀-7</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₈-11</td>
<td>a₁₂</td>
<td>a₁₂-13</td>
<td>a₁₄</td>
<td>0</td>
</tr>
<tr>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>a₀-3</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>0</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₈-11</td>
<td>a₁₂</td>
<td>a₁₂-13</td>
<td>a₁₄</td>
<td>a₀-7</td>
</tr>
<tr>
<td>a₀</td>
<td>a₀-1</td>
<td>a₂</td>
<td>0</td>
<td>a₄</td>
<td>a₄-5</td>
<td>a₆</td>
<td>a₀-3</td>
<td>a₈</td>
<td>a₈-9</td>
<td>a₁₀</td>
<td>a₀-7</td>
<td>a₁₂</td>
<td>a₁₂-13</td>
<td>a₁₄</td>
<td>a₀-11</td>
</tr>
<tr>
<td>a₀</td>
<td>0</td>
<td>a₂</td>
<td>a₀-1</td>
<td>a₄</td>
<td>a₀-3</td>
<td>a₆</td>
<td>a₀-5</td>
<td>a₈</td>
<td>a₀-7</td>
<td>a₁₀</td>
<td>a₀-9</td>
<td>a₁₂</td>
<td>a₀-11</td>
<td>a₁₄</td>
<td>a₀-13</td>
</tr>
<tr>
<td>0</td>
<td>a₀</td>
<td>a₀-1</td>
<td>a₀-2</td>
<td>a₀-3</td>
<td>a₀-4</td>
<td>a₀-5</td>
<td>a₀-6</td>
<td>a₀-7</td>
<td>a₀-8</td>
<td>a₀-9</td>
<td>a₀-10</td>
<td>a₀-11</td>
<td>a₀-12</td>
<td>a₀-13</td>
<td>a₀-14</td>
</tr>
</tbody>
</table>
Work efficient exclusive scan algorithm

Up-sweep:

for \( d=0 \) to \( (\log_2 n - 1) \) do 
  forall \( k=0 \) to \( n-1 \) by \( 2^{d+1} \) do 
    \( a[k + 2^{d+1} - 1] = a[k + 2^d - 1] + a[k + 2^{d+1} - 1] \)

Down-sweep:

\( x[n-1] = 0 \)

for \( d=(\log_2 n - 1) \) down to \( 0 \) do 
  forall \( k=0 \) to \( n-1 \) by \( 2^{d+1} \) do 
    \( \text{tmp} = a[k + 2^d - 1] \)
    \( a[k + 2^d - 1] = a[k + 2^{d+1} - 1] \)
    \( a[k + 2^{d+1} - 1] = \text{tmp} + a[k + 2^{d+1} - 1] \)

Work: \( O(N) \) (but what is the constant?)
Span: \( O(\log N) \) (but what is the constant?)
Locality: ??
Scan implementation: two cores

\[ a_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6 \ a_7 \]

\[ a_8 \ a_9 \ a_{10} \ a_{11} \ a_{12} \ a_{13} \ a_{14} \ a_{15} \]

\[ a_0 \ a_0-1 \ a_2 \ a_{0-3} \ a_4 \ a_{4-5} \ a_6 \ a_{4-7} \]

\[ a_8 \ a_8-9 \ a_{10} \ a_{8-11} \ a_{12} \ a_{12-13} \ a_{14} \ a_{12-15} \]

\[ a_0 \ a_0-1 \ a_2 \ a_{0-3} \ a_4 \ a_{4-5} \ a_6 \ a_{0-7} \]

\[ a_8 \ a_8-9 \ a_{10} \ a_{8-11} \ a_{12} \ a_{12-13} \ a_{14} \ a_{8-15} \]

\[ a_0 \ a_0-1 \ a_2 \ a_{0-3} \ a_4 \ a_{4-5} \ a_6 \ a_{0-7} \]

\[ a_8 \ a_8-9 \ a_{10} \ a_{8-11} \ a_{12} \ a_{12-13} \ a_{14} \ 0 \]

\[ a_0 \ a_0-1 \ a_2 \ a_{0-3} \ a_4 \ a_{4-5} \ a_6 \ 0 \]

\[ a_8 \ a_8-9 \ a_{10} \ a_{8-11} \ a_{12} \ a_{12-13} \ a_{14} \ a_{0-7} \]

\[ a_0 \ a_0-1 \ a_2 \ 0 \ a_4 \ a_{4-5} \ a_6 \ a_{0-3} \]

\[ a_8 \ a_8-9 \ a_{10} \ a_{0-7} \ a_{12} \ a_{12-13} \ a_{14} \ a_{0-11} \]

\[ a_0 \ 0 \ a_2 \ a_{0-1} \ a_4 \ a_{0-3} \ a_6 \ a_{0-5} \]

\[ a_8 \ a_{0-7} \ a_{10} \ a_{0-9} \ a_{12} \ a_{0-11} \ a_{14} \ a_{0-13} \]

\[ 0 \ a_0 \ a_0-1 \ a_0-2 \ a_{0-3} \ a_{0-4} \ a_{0-5} \ a_{0-6} \]

\[ a_{0-7} \ a_{0-8} \ a_{0-9} \ a_{0-10} \ a_{0-11} \ a_{0-12} \ a_{0-13} \ a_{0-14} \]
Exclusive scan: two processor implementation

| a₀ | a₁ | a₂ | a₃ | a₄ | a₅ | a₆ | a₇ | a₈ | a₉ | a₁₀ | a₁₁ | a₁₂ | a₁₃ | a₁₄ | a₁₅ |

**P1**

Sequential scan on elements [0-7]

Let base = a₀-7
(a₀-6 already computed as part of exclusive scan, just add a₇)

Add base to elements a₈ thru a₈-11

**P2**

Sequential scan on elements [8-15]

Add base to elements a₈-12 thru a₈-15

Work: O(N) (but constant is now only 1.5)

Data-access:
- Very high spatial locality (contiguous memory access)
- P1’s access to a₈ through a₈-11 may be more costly on large NUMA system, but on small-scale system access likely same cost as from P2
Exclusive scan: wide SIMD implementation

Example: perform exclusive scan on 32-element array: 32-wide GPU execution (SPMD program)
When scan_warp is run by a group of 32 CUDA threads, each thread returns the exclusive scan result for element ‘idx’ (note: upon completion ptr[] stores inclusive scan result)

```
template<class OP, class T>
__device__ T scan_warp(volatile T *ptr, const unsigned int idx)
{
    const unsigned int lane = idx & 31; // index of thread in warp (0..31)

    if (lane >= 1) ptr[idx] = OP::apply(ptr[idx - 1], ptr[idx]);
    if (lane >= 2) ptr[idx] = OP::apply(ptr[idx - 2], ptr[idx]);
    if (lane >= 4) ptr[idx] = OP::apply(ptr[idx - 4], ptr[idx]);
    if (lane >= 8) ptr[idx] = OP::apply(ptr[idx - 8], ptr[idx]);
    if (lane >= 16) ptr[idx] = OP::apply(ptr[idx - 16], ptr[idx]);

    return (lane>0) ? ptr[idx-1] : OP::identity();
}
```

```
CUDA thread index
```

```
Work: ??
```
Wide SIMD implementation

Example: exclusive scan 32-element array
32-wide GPU execution (SPMD program)

```
template<class OP, class T>
__device__ T scan_warp(volatile T *ptr, const unsigned int idx)
{
    const unsigned int lane = idx & 31; // index of thread in warp (0..31)

    if (lane >= 1) ptr[idx] = OP::apply(ptr[idx - 1], ptr[idx]);
    if (lane >= 2) ptr[idx] = OP::apply(ptr[idx - 2], ptr[idx]);
    if (lane >= 4) ptr[idx] = OP::apply(ptr[idx - 4], ptr[idx]);
    if (lane >= 8) ptr[idx] = OP::apply(ptr[idx - 8], ptr[idx]);
    if (lane >= 16) ptr[idx] = OP::apply(ptr[idx - 16], ptr[idx]);

    return (lane>0) ? ptr[idx-1] : OP::identity();
}
```

Work: $N \log(N)$

Work-efficient formulation of scan is not beneficial in this context because it results in low SIMD utilization. It would require more than 2x the number of instructions as the implementation above!
Building scan on larger array

Example: 128-element scan using four-warp thread block
Multi-threaded, SIMD implementation

Example: cooperating threads in a CUDA thread block
(We provided similar code in assignment 2, assumes length of array given by ptr is same as number of threads per block)

```cpp
template<class OP, class T>
__device__ void scan_block(volatile T *ptr, const unsigned int idx) {
    const unsigned int lane = idx & 31; // index of thread in warp (0..31)
    const unsigned int warpid = idx >> 5;

    T val = scan_warp<OP,T>(ptr, idx); // Step 1. per-warp partial scan

    if (lane == 31) ptr[warpid] = ptr[idx]; // Step 2. copy partial-scan bases
    __syncthreads();

    if (warpid == 0) scan_warp<OP,T>(ptr, idx); // Step 3. scan to accumulate bases
    __syncthreads();

    if (warpid > 0) // Step 4. apply bases to all elements
        val = OP::apply(ptr[warpid-1], val);
    __syncthreads();

    ptr[idx] = val;
}
```
Building a larger scan

Example: 1 million element scan (1024 elements per block)

Exceeding 1 million elements requires partitioning phase 2 into multiple blocks
Scan implementation

- **Parallelism**
  - Scan algorithm features $O(N)$ parallel work
  - But efficient implementations only leverage as much parallelism as required to make good utilization of the machine
    - Reduce work and reduce communication/synchronization

- **Locality**
  - Multi-level implementation matches memory hierarchy
    (Per-block implementation carried out in local memory)

- **Heterogeneity: different strategy at different machine levels**
  - Different algorithm for intra-warp scan than inter-thread scan
Segmented scan

- Generalization of scan
- Simultaneously perform scans on arbitrary contiguous partitions of input collection

```plaintext
let a = [[1,2],[6],[1,2,3,4]]
let ⊕ = +
segmented_scan_exclusive(⊕,a) = [[0,1], [0], [0,1,3,6]]
```

Assume simple “head-flag” representation:

```
a = [[1,2,3],[4,5,6,7,8]]
flag: 0 0 0 1 0 0 0 0
data: 1 2 3 4 5 6 7 8
```
Work-efficient segmented scan

Up-sweep:

for \( d = 0 \) to \((\log_2 n - 1)\) do
  for all \( k = 0 \) to \( n-1 \) by \( 2^{d+1} \) do
    if \( \text{flag}[k + 2^{d+1} - 1] == 0: \)
      \( \text{data}[k + 2^{d+1} - 1] = \text{data}[k + 2^d - 1] + \text{data}[k + 2^{d+1} - 1] \)
      \( \text{flag}[k + 2^{d+1} - 1] = \text{flag}[k + 2^d - 1] || \text{flag}[k + 2^{d+1} - 1] \)

Down-sweep:

\( \text{data}[n-1] = 0 \)

for \( d = (\log_2 n - 1) \) down to \( 0 \) do
  for all \( k = 0 \) to \( n-1 \) by \( 2^{d+1} \) do
    \( \text{tmp} = \text{data}[k + 2^d - 1] \)
    \( \text{data}[k + 2^d - 1] = \text{data}[k + 2^{d+1} - 1] \)
    if \( \text{flag_original}[k + 2^d] == 1: \) // must maintain copy of original flags
      \( \text{data}[k + 2^{d+1} - 1] = 0 \)
    else if \( \text{flag}[k + 2^d - 1] == 1: \)
      \( \text{data}[k + 2^{d+1} - 1] = \text{tmp} \)
    else:
      \( \text{data}[k + 2^{d+1} - 1] = \text{tmp} + \text{data}[k + 2^{d+1} - 1] \)
      \( \text{flag}[k + 2^d - 1] = 0 \)
Segmented scan

\[ a_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6 \ a_7 \ a_8 \ a_9 \ a_{10} \ a_{11} \ a_{12} \ a_{13} \ a_{14} \ a_{15} \]

\[ a_0 \ a_{\theta-1} \ a_2 \ a_{\theta-3} \ a_4 \ a_5 \ a_6 \ a_{\theta-7} \ a_8 \ a_{\theta-9} \ a_{10} \ a_{10-11} \ a_{12} \ a_{12-13} \ a_{14} \ a_{14-15} \]

\[ a_0 \ a_{\theta-1} \ a_2 \ a_{\theta-3} \ a_4 \ a_5 \ a_6 \ a_{\theta-7} \ a_8 \ a_{\theta-9} \ a_{10} \ a_{10-11} \ a_{12} \ a_{12-13} \ a_{14} \ a_{12-15} \]

\[ a_0 \ a_{\theta-1} \ a_2 \ a_{\theta-3} \ a_4 \ a_5 \ a_6 \ a_{\theta-7} \ a_8 \ a_{\theta-9} \ a_{10} \ a_{10-11} \ a_{12} \ a_{12-13} \ a_{14} \ a_{10-15} \]

\[ a_0 \ a_{\theta-1} \ a_2 \ a_{\theta-3} \ a_4 \ a_5 \ a_6 \ a_{\theta-7} \ a_8 \ a_{\theta-9} \ a_{10} \ a_{10-11} \ a_{12} \ a_{12-13} \ a_{14} \ a_{10-13} \]

\[ a_0 \ a_{\theta-1} \ a_2 \ a_{\theta-3} \ a_4 \ a_{\theta-3} \ a_6 \ a_5 \ a_8 \ a_{10} \ a_{10} \ a_{12} \ a_{12-13} \ a_{14} \ a_{10-11} \ a_{10-13} \]

\[ a_0 \ a_{\theta-3} \ a_4 \ a_5 \ a_6 \ a_{\theta-3} \ a_8 \ a_{10} \ a_{10} \ a_{12} \ a_{12-13} \ a_{14} \ a_{10-11} \ a_{10-13} \]

\[ a_0 \ a_{\theta-1} \ a_2 \ a_{\theta-2} \ a_{\theta-3} \ a_5 \ a_5 \ a_6 \ a_{\theta-6} \ a_8 \ a_{10} \ a_{10} \ a_{10-13} \ a_{10-14} \]
Sparse matrix multiplication example

\[
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
\vdots \\
y_{n-1}
\end{bmatrix}
\begin{bmatrix}
3 & 0 & 1 & \cdots & 0 \\
0 & 2 & 0 & \cdots & 0 \\
0 & 0 & 4 & \cdots & 0 \\
\vdots \\
0 & 2 & 6 & \cdots & 8
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
\vdots \\
x_{n-1}
\end{bmatrix}
\]

- Most values in matrix are zero
  - Note: logical parallelization is across per-row dot products
  - But different amounts of work per row (complicates wide SIMD execution)

- Example sparse storage format: compressed sparse row
  
  values = [ [3,1], [2], [4], ..., [2,6,8] ]
  
  row_starts = [0, 2, 3, 4, ...]
  
  cols = [ [0,2], [1], [2], ...., ]
Sparse matrix multiplication with scan

values = \([3,1], [2], [4], [2,6,8]\) 
cols = \([0,2], [1], [2], [1,2,3]\) 
row_starts = \([0, 2, 3, 4]\)

\[
\begin{bmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3 \\
\end{bmatrix} = \begin{bmatrix}
3 & 0 & 1 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & 4 & 0 \\
0 & 2 & 6 & 8 \\
\end{bmatrix} \begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
\end{bmatrix}
\]

1. Map over all non-zero values: \(\text{products}[i] = \text{values}[i] \times x[\text{cols}[i]]\)
   - \(\text{products} = [3x_0, x_2, 2x_1, 4x_2, 2x_1, 6x_2, 8x_3]\)
2. Create flags vector from row_starts: \(\text{flags} = [1,0,1,1,0,0]\)
3. **Inclusive** segmented-scan on (multiples, flags) using addition operator
   - \([3x_0, 3x_0+x_2, 2x_1, 4x_2, 2x_1, 2x_1+6x_2, 2x_1+6x_2+8x_2]\)
4. Take last element in each segment:
   - \(y = [3x_0+x_2, 4x_2, 2x_1, 2x_1+6x_2+8x_2]\)
Scan/segmented scan summary

- **Scan**
  - Parallel implementation of (intuitively sequential application)
  - Theory: parallelism linear in number of elements
  - Practice: exploit locality, use only as much parallelism as necessary to fill the machine
    - Great example of applying different strategies at different levels of the machine

- **Segmented scan**
  - Express computation and operate on irregular data structures (e.g., list of lists) in a regular, data parallel way
Example 4: ray tracing

Problem statement:
Given a “ray”, find closest intersection with scene geometry

Simplest ray tracer:
For each image pixel, shoot ray from camera through pixel into scene. Color pixel according to first surface hit.
Accelerating ray-scene intersection

Preprocess scene to build data structure that accelerates finding “closest” geometry along ray

Idea: group objects with spatial proximity (like quad-tree in Barnes-Hut)

- Hierarchical grouping adapts to non-uniform density of scene objects

**Scene objects (in 2D)**

**Bounding Volume Hierarchy (BVH)**
(Binary tree organizing the scene)
Tracing a ray (accelerated using BVH)

// ClosestIsectInfo stores information about closest intersection found so far along a ray

struct ClosestIsectInfo {
    Primitive primitive;    // primitive that is the closest intersection
    float distance;        // distance along ray to closest intersection
};

void traceRay(Ray ray, BVHNode node, ClosestIsectInfo isectInfo)
{
    if (!intersect(ray, node.bbox) ||
        (closest point on box is farther than hitInfo.distance))
    {
        return;
    }

    if (node.leaf) {
        for (each primitive in leaf node) {
            (does_isect, distance) = intersect(ray, primitive);
            if (does_isect && distance < isectInfo.distance) {
                // check if new isect is closest
                isectInfo.primitive = primitive;
                isectInfo.distance = distance;
            }
        }
    } else {
        traceRay(ray, node.leftChild, hitInfo);
        traceRay(ray, node.rightChild, hitInfo);
    }
}
Decomposition and assignment

- Spatial decomposition of image (2D blocks)
  - 2D blocks maximize spatial locality of rays
- Create many more tiles than processors (just like assignment 1, problem 2)
- Use simple work queue to dynamically assign tiles to processors
  - Cost to render a block is large, therefore synchronization overhead is trivial
Ray tracing using ray packets

Code traces a collection of rays (a “packet”) through BVH at once

```c
RayPacket
{
    Ray rays[PACKET_SIZE];
    bool active[PACKET_SIZE];
};

trace(RayPacket rays, BVHNode node, ClosestHitInfo packetHitInfo)
{
    if (!ANY_ACTIVE_intersect(rays, node.bbox) ||
        (closest point on box is farther than hitInfo.distance for all active rays))
    {
        return;
    }

    update packet active mask to de-activate rays that do not intersect node bbox (these
    rays would not have visited children nodes in a non-packet implementation)

    if (node.leaf) {
        for (each primitive in node) {
            for (each ACTIVE ray r in packet) {
                (hit, distance) = intersect(ray, primitive);
                if (hit && distance < hitInfo.distance) {
                    hitInfo[r].primitive = primitive;
                    hitInfo[r].distance = distance;
                }
            }
        }
    } else {
        trace(rays, node.leftChild, hitInfo);
        trace(rays, node.rightChild, hitInfo);
    }
}
```
Ray tracing using packets

Blue = ray is active after node-box test

Note: r6 intersects F, but does not pass node F box test due to closest-so-far check
Advantages of packets

- Enables SIMD execution of tree traversal logic for multiple rays in parallel
  - One vector lane per ray

- Amortize node data fetch (since all rays in packet visit node at same time)
  - Load BVH node once for all rays in packet
  - A reason to make packets much bigger than SIMD width!

- Amortize work
  - Packets are an algorithmic improvement: some computations can be shared across rays in a packet (this help sequential algorithm as well)
  - A reason to make packets much bigger than SIMD width!
Disadvantages of packets

- If *any* ray must visit a node in BVH, it drags all rays in the packet along with it!

- **Implications:**
  - Diminished SIMD execution efficiency during traversal due to divergence
  - Node traversal, intersection, etc. are now amortized over less than a full packet's worth of rays
When rays in a packet are “incoherent” (travel in different directions), the benefit of packets can decrease significantly. In this example: packet visits all tree nodes. (All rays in packet visit all tree nodes)
Improving packet tracing with ray reordering

Idea: when packet utilization drops below threshold, resort rays and continue with smaller packet

- Increases SIMD utilization
- But loses amortization benefits of large packets

Example: 8-wide SIMD processor, 16-ray packets
2 SIMD instructions required to perform operation for all 16 rays in full packet

16-ray packet: 7 of 16 rays active
2 SIMD instructions do meaningful work for only 7 rays

Reorder rays
Recompute intervals/bounds for active rays

Continue tracing with 8-ray packet: 7 of 8 rays active
(Now: 1 SIMD instruction to perform operation for 7 active rays)
Giving up on packets

- Even with reordering, ray coherence during BVH traversal will diminish
  - Little benefit to packets (can even decrease performance compared to single ray code due to overhead of managing packet)

- Idea: exploit SIMD execution for parallel work within single ray-BVH intersection query
  - Interior: use wider-branching BVH (e.g., use degree four rather than binary BVH) (test single ray against multiple node bounding boxes in parallel)
  - Leaf: test single ray against multiple triangles in parallel
Ray tracing best practices on multi-core CPU

- **Trivial assignment of screen regions to CPU cores**
  - Use work queue to dynamically assign regions of screen to cores (some regions may be more expensive than others)

- **Effective use of SIMD execution is harder**
  - Use large packets for higher levels of BVH
    - Ray coherence always high at the top of the tree (visited by all rays)
  - Can switch to single ray (use SIMD for intra-ray parallelism) when packet utilization drops below threshold
  - Can use dynamic reordering to postpone time of switch
    - Reordering allows packets to provide benefit deeper into tree

- **Building a BVH in parallel is a tricky problem**
  - Multiple threads must cooperate to build a shared structure
  - Not addressed here
Summary

Today we looked at several different parallel programs

Key questions:
- What are the dependencies?
- What synchronization is necessary?
- How to balance the work?
- How to exploit locality inherent in the problem?

Trends
- Only need enough parallelism to keep all processing elements busy (e.g., data-parallel scan vs. simple multi-core scan)
- Different parallelization strategies may be applicable under different workloads (packets vs. no packets) or different locations in the machine (different implementations of scan internal and external to warp)