Data Parallel Computations

- PP Section 6.2.1
- forall statement:
  ```
  forall( j = 0; j < n; j++ )
  { … body done in parallel for all j … }
  ```
- Barrier implied at end among parallel bodies

Example: Prefix-Sum

- Assume that \( n \) is a power of 2.
- Assume shared memory.
- ```
  for( j = 0; j < log(n); j++ )
  forall( i = 2^j; i < n; i++)
  x[i] += x[i - 2^j];
  ```

Implementing forall using SPMD:

- ```
  for( j = 0; j < log(n); j++ )
  forall( i = 0; i < n; i++)
  Body(i);
  barrier();
  ```

Example: Prefix-Sum

- ```
  for( j = 0; j < log(n); j++ )
  x[i] += x[i - 2^j];
  ```

Example: Iterative Linear Equation Solver

- ```
  for( iter = 0; iter < numIterations; iter++ )
  forall( i = 0; i < n; i++ )
  {
    double sum = 0;
    for( j = 1; j < n; j++ )
      sum += a[i][j]*x[j];
    x[i] = sum;
  }
  ```

Iterative Linear Equation Solver: Translation to SPMD

- ```
  for( iter = 0; iter < numIterations; iter++ )
  {
    i = my_process_rank();
    double sum = 0;
    for( j = 1; j < n; j++ )
      sum += a[i][j]*x[j];
    new_x[i] = sum;
    all gather new_x to x (implied barrier)
  }
  ```
### Nested forall's

- `forall( i = 0; i < m; i++)
  forall( j = 0; j < n; i++)
  Body(i, j)`

### Example of nested forall's: Laplace equation

- `forall( i = 0; i < m; i++)
  forall( j = 0; j < n; i++)
  x[i][j] = (x[i-1][j] + x[i][j-1] + x[i+1][j] + x[i][j+1]) / 4.0;

### Exercise

- How would you translate nested forall's to SPMD?

### Cellular Automata

- Synchronous computation
- Infinitely-large grid (finite occupancy)
- Typically fine-grain
- If distributed, still need to communicate and boundaries, once per cycle.

### PRAM Model (PP Appendix D)

- **PRAM = Parallel, Random-Access Machine**
- Idealized model introduced in 1978, based on theoretical RAM model
- Unbounded number of processors, to fit problem
- Shared common memory + local memories per processor
- Processors operate synchronously, could be loosened to SPMD with synchronization routines
- Writing to common memory is synchronous

### Use of PRAM Model

- Simple and elegant for some problems
- Can tell us certain things about structuring, especially for synchronous computation
- Can be simulated on parallel machines (e.g. by rescheduling, Brent’s lemma, etc.)
- At least one is being constructed
The project goal is to achieve a 64 physical (2048 virtual) processor machine with 2 GByte of global memory and 256 hard disks. The machine will be connected to the internet for free access.

**Memory-Conflicts**

- All processors can read or write to distinct shared memory locations in one time step.
- What if two processors try to read from the same memory location in the same time step?
- What if two processors try to write to the same memory location in the same time step?

**PRAM Varieties Based on Memory-Conflict Models**

- Generally concurrent reading and writing to a single location is disallowed.
- **EREW** (Exclusive-Read, Exclusive-Write)
  Concurrent reading or writing to a location is disallowed.
- **CREW** (Concurrent-Read, Exclusive-Write)
  Concurrent writing to a location is disallowed.
- **CRCW** (Concurrent-Read, Concurrent-Write)
  Concurrent writing to a location is allowed.

**Sub-varieties of CRCW (1) indicate how conflict is resolved**

- **CRCW-Common**: Concurrent writing is allowed only if it is known that all processors will be writing the same value (writing no value is always an option).
- **CRCW-Arbitrary**: If multiple processors attempt to write, one will be chosen as the winner and the others ignored.

**Sub-varieties of CRCW (2) indicate how conflict is resolved**

- **CRCW-Priority**: If multiple processors attempt to write, the highest-priority will be chosen as the winner and the others ignored.
- **CRCW-Sum**: If multiple processors attempt to write, the values will be summed and the sum written instead.
- **Variants on Sum**: Any binary operator (or, and, xor, min, max, product, …)

**Why does it matter?**

- To physically realize any approximation to a PRAM requires an understanding of the memory conflict model.
- There is a time cost to resolving memory conflicts, which varies depending on the model.
PRAM Preferences

- It is preferable to use as little machinery as possible for algorithms.
- Therefore, prefer
  - CREW over CRCW
  - CRCW-arbitrary over CRCW-common
  - CRCW-common over CRCW-sum
  - etc.

PRAM Algorithm Examples

- Computing max of n numbers:
  - log n time on EREW (and by implication CREW, CRCW, ...)
  - Assume the numbers are in shared memory locations 0, 1, ..., n-1.
  - Even numbered processors fetch “their” numbers to their local memory (other processors are idle).
  - Even numbered processes fetch “their neighbors numbers to their local memory.
  - Even numbered processors write the max of the two numbers to “their” locations.
  - Repeat with processors divisible by 4, 8, 16, ...

PRAM Max

Essentially we have a subtree of the prefix-sum tree (using max instead of add).

Better (?) ways to do max

- Intuitively Ω(log n) seems like a lower bound on the max computation of n numbers.
- However, a CRCW-arbitrary PRAM can do better.

PRAM Prefix Sum

- Obviously an EREW PRAM can compute any prefix-sum type computation in O(log n).
- More processors are busy than in the max case.

CRCW-arbitrary max computation

- O(1)
- using $n^2$ processors
**CRCW-arbitrary max computation setup**

- Let the data be in shared memory locations \( x[0], \ldots, x[n-1] \).
- Use \( n \) bit locations: \( b[0], \ldots, b[n-1] \), all set to 1 (in one step).
- \( b[i] \) is associated with \( x[i] \).

**CRCW-arbitrary max computation**

- The meaning is that, at the end of the computation, \( b[i] \) will be 0 iff \( x[i] \) is less than some \( x[j] \) where \( j \neq i \).
- So elements \( x[i] \) where \( b[i] == 1 \) will be the max.
- In three steps: \( n(n-1)/2 \) processors each fetch, then compare a different \( x[i] \) with an \( x[j] \). If \( x[i] < x[j] \), the processor sets \( b[i] \) to 0, and vice-versa.

**CRCW-arbitrary max computation**

- Each processor either writes 0 or does nothing.
- If two processors write to the same location, they will both be writing the same thing.
- Therefore the CRCW-arbitrary assumption is honored.

**What happened to \( \Omega(\log n) \)?**

- In an implementation of CRCW-common, it isn’t physically realizable to have an arbitrary number of processors write to the same location at once, even if they do write the same value.
- We have replaced what would have been binary ops with a single op of arbitrary arity.
- We could implement this op as a fan-in tree, which would recover the \( \Omega(\log n) \).

\( [O(n^2) \text{ processors fanning in, } \log(n^2) = O(\log n)]. \)

**Simulation Theorem (see Cormen, et al., p706-708)**

- Any CRCW-common PRAM algorithm using \( p \) processors can be simulated by an EREW PRAM with a slowdown factor of \( \log(p) \).

**Array Compression**

- Problem:
  Given an array in shared memory and a bit vector indicating the elements to be compressed, create an array containing only those elements contiguously.

```
1 0 1 1 1 0 0 1 0 0 0
0 1 1 1 1 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0 0
```
Technique

- Use prefix-sum + indexing (parallel)
- Compute the prefix sum of the bit array
- Use the values as indexes of where to store the corresponding item.
- Only use the index at transitions (to avoid the need to use CRCW).

Array Compression Exposed

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>l</th>
<th>m</th>
<th>n</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Prefix sum

<table>
<thead>
<tr>
<th>a</th>
<th>e</th>
<th>i</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>111</td>
<td>111</td>
<td>222</td>
</tr>
</tbody>
</table>

Transitions (use as indices)

<table>
<thead>
<tr>
<th>p</th>
<th>a</th>
<th>e</th>
<th>i</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Parallel stores

Exercise

- How would you do array expansion: distribute array elements according to a bit vector?