The PRAM (Parallel Random Access Memory) model

- All processors operate synchronously under the control of a common CPU.
The PRAM (Parallel Random Access Memory) model

- All processors operate synchronously under the control of a common CPU.
- Shared memory. Common random access (array) memory, which can be accessed for read and write.
The PRAM (Parallel Random Access Memory) model

- All processors operate synchronously under the control of a common CPU.
- Shared memory. Common random access (array) memory, which can be accessed for read and write
- SIMD. Each active processor executes the same program. (Single Instruction Multiple Data).
The PRAM (Parallel Random Access Memory) model

- All processors operate synchronously under the control of a common CPU.
- Shared memory. Common random access (array) memory, which can be accessed for read and write
- SIMD. Each active processor executes the same program. (Single Instruction Multiple Data).
- Several models of how processors can access the same shared memory location. EREW, CREW, CRCW.
The PRAM (Parallel Random Access Memory) model

- All processors operate synchronously under the control of a common CPU.
- Shared memory. Common random access (array) memory, which can be accessed for read and write.
- SIMD. Each active processor executes the same program. (Single Instruction Multiple Data).
- Several models of how processors can access the same shared memory location. EREW, CREW, CRCW.
- CRCW is more powerful than EREW.
The PRAM (Parallel Random Access Memory) model

- All processors operate synchronously under the control of a common CPU.
- Shared memory. Common random access (array) memory, which can be accessed for read and write.
- SIMD. Each active processor executes the same program. (Single Instruction Multiple Data).
- Several models of how processors can access the same shared memory location. EREW, CREW, CRCW.
- CRCW is more powerful than EREW.
- Various protocols for CW exist. The simplest is PRIORITY. The processor with the lowest (or highest) index has priority.
The PRAM (Parallel Random Access Memory) model

- All processors operate synchronously under the control of a common CPU.
- Shared memory. Common random access (array) memory, which can be accessed for read and write
- SIMD. Each active processor executes the same program. (Single Instruction Multiple Data).
- Several models of how processors can access the same shared memory location. EREW, CREW, CRCW.
- CRCW is more powerful than EREW.
- Various protocols for CW exist. The simplest is PRIORITY. The processor with the lowest (or highest) index has priority.
- Others are ARBITRARY (choose processor at random) and COMMON (only write if values same).
Close look at an example. Array total on EREW-PRAM.

Various things to consider in understanding a parallel algorithm:
In no particular order, there include
- Pseudocode 'Syntax'
- Algorithm design (why is it done this way?)
- Flowchart (what is it doing?)
- Number of sequential steps
- Best sequential algorithm
- Work-Time analysis
- Efficiency metrics
The problem

- An array $A[1..8]$, with entries
  
  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

  
  We want to add up the entries in parallel.

  We want the final answer to be stored at memory location 1.

  We add the contents of locations $2i-1$ and $2i$ into location $i$.

  This halves the effective array length in one (parallel) step.

  
  | 1+2=3 | 3+4=7 | 5+6=11 | 7+8=15 |

  We now repeat.

  | 3+7 | 1+15 |

  And so on. The algorithm is about halving the number of memory locations we need at each step.
The problem

- An array $A[1..8]$, with entries
  
  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
  
- We want to add up the entries in parallel. We want the final answer to be stored at memory location 1.
The problem

- An array $A[1..8]$, with entries
  \[
  \begin{array}{cccccccc}
  1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
  \end{array}
  \]
- We want to add up the entries in parallel.
  We want the final answer to be stored at memory location 1.
- We add the contents of locations $2i - 1$ and $2i$ into location $i$.
  This halves the effective array length in one (parallel) step.
The problem

- An array $A[1..8]$, with entries
  
  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
  |

- We want to add up the entries in parallel. We want the final answer to be stored at memory location 1.

- We add the contents of locations $2i - 1$ and $2i$ into location $i$.
  This halves the effective array length in one (parallel) step.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1+2=3</td>
<td>3+4=7</td>
<td>5+6=11</td>
<td>7+8=15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The problem

- An array \( A[1..8] \), with entries

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

- We want to add up the entries in parallel. We want the final answer to be stored at memory location 1.

- We add the contents of locations \( 2i - 1 \) and \( 2i \) into location \( i \).

This halves the effective array length in one (parallel) step.

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1+2=3 & 3+4=7 & 5+6=11 & 7+8=15 \\
\end{array}
\]

- We now repeat.

\[
\begin{array}{cccc}
3 & 7 & 11 & 15 \\
3+7 & 1+15 \\
\end{array}
\]

- And so on. The algorithm is about halving the number of memory locations we need at each step.
We can use this for any associative binary operation, e.g. $+$, $\ast$, $\max$, $\min$.

Associative basically means 'order of evaluation doesn’t matter'.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can use this for any associative binary operation ‘+’
e.g. +, *, max, min
Associative basically means ‘order of evaluation doesn’t matter’
We can use this for any associative binary operation '+'
e.g. +, *, max, min
Associative basically means 'order of evaluation doesn't matter'

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a+b</th>
<th>c+d</th>
<th>e+f</th>
<th>g+h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
begin P-Sum-EREW
Input: $n = 2^k$ numbers stored in Array $A[1..n]$
Output: $S = \sum_{i=1}^{n} A[i]$
Note $k = \log_2 n$

begin
1. for $i = 1...n$ in parallel do
   $B[i] = A[i]$

2. for $h = 1..k$ do
   for $1 \leq i \leq n/2^h$ in parallel do

end P-Sum-EREW
A word about indexing

- Our array size was $n = 2^k$
- Our main (sequential) index $h$ was 'for $h = 1..k$ do'
A word about indexing

- Our array size was $n = 2^k$
- Our main (sequential) index $h$ was 'for $h = 1..k$ do'
- Explain 'for $1 \leq i \leq n/2^h$ in parallel do'
A word about indexing

- Our array size was $n = 2^k$
- Our main (sequential) index $h$ was 'for $h = 1..k$ do'
- Explain 'for $1 \leq i \leq n/2^h$ in parallel do'
- When $h = 1$ our parallel index $i$ ran from $1 \leq i \leq n/2$
A word about indexing

- Our array size was $n = 2^k$
- Our main (sequential) index $h$ was 'for $h = 1..k$ do'
- Explain 'for $1 \leq i \leq n/2^h$ in parallel do'
- When $h = 1$ our parallel index $i$ ran from $1 \leq i \leq n/2$
- When $h = 2$ our array size is $n/2$ and our parallel index $i$ ran from $1 \leq i \leq n/4$
A word about indexing

- Our array size was $n = 2^k$
- Our main (sequential) index $h$ was 'for $h = 1..k$ do'
- Explain 'for $1 \leq i \leq n/2^h$ in parallel do'
- When $h = 1$ our parallel index $i$ ran from $1 \leq i \leq n/2$
- When $h = 2$ our array size is $n/2$ and our parallel index $i$ ran from $1 \leq i \leq n/4$
- In the final round, our array size is 2, our parallel index is $i = 1$
Q: Could we have chosen a different indexing algorithm?

Either way, some care is needed.
Q: Could we have chosen a different indexing algorithm?
A: Yes. Here is one from last years course. (Exercise: Check it)
Q: Could we have chosen a different indexing algorithm?
A: Yes. Here is one from last years course. (Exercise: Check it)

for 1 \leq i \leq n \text{ in parallel do}
  \text{for all } j, \ 1 \leq j \leq \log_2 n \text{ do}
    \text{if } i \mod 2^j = 0 \text{ then}
      B[i] = B[i] + B[i - 2^{j-1}]

Either way some care is needed.
The algorithm is about choosing the indexing
begin P-Sum-CRCW
Input: \( n = 2^k \) numbers stored in Array \( A[1..n] \)
Output: \( S = \sum_{i=1}^{n} A[i] \)
Note \( k = \log_2 n \)

PRIORITY-CW

1. \( S = 0 \)
2. for \( i = 1 \ldots n \) in parallel do
   \( S = S + A[i] \)
end for
Analysis of algorithms

- Order of magnitude (approximate) cost estimates.
- Uniform cost model: We charge 1 unit for "basic operations". What we charge for depends on the model.
Analysis of algorithms

- Order of magnitude (approximate) cost estimates.
- Uniform cost model: We charge 1 unit for "basic operations". What we charge for depends on the model.
- Multi-threading model: Charge for recursive calls (including base cases), spawn and sync operations. We ignore addition etc.
Analysis of algorithms

- Order of magnitude (approximate) cost estimates.
- Uniform cost model: We charge 1 unit for "basic operations". What we charge for depends on the model.
- Multi-threading model: Charge for recursive calls (including base cases), spawn and sync operations. We ignore addition etc.
- PRAM model: Large variety of basic operations; adding, writing to memory, calling a procedure etc. Normally we charge 1 for each operation
Analysis of algorithms

- Order of magnitude (approximate) cost estimates.
- Uniform cost model: We charge 1 unit for ”basic operations”. What we charge for depends on the model.
- Multi-threading model: Charge for recursive calls (including base cases), spawn and sync operations. We ignore addition etc.
- PRAM model: Large variety of basic operations; adding, writing to memory, calling a procedure etc. Normally we charge 1 for each operation.
- Interconnection network model: Charge for the communication between processors.
Analysis of algorithms

- Order of magnitude (approximate) cost estimates.
- Uniform cost model: We charge 1 unit for ”basic operations”. What we charge for depends on the model.
- Multi-threading model: Charge for recursive calls (including base cases), spawn and sync operations. We ignore addition etc.
- PRAM model: Large variety of basic operations; adding, writing to memory, calling a procedure etc. Normally we charge 1 for each operation.
- Interconnection network model: Charge for the communication between processors.
- We use crude performance measures which do not distinguish between $n, 2n, 3n$ etc but distinguish $n, n^2, n^3$, eg $O(\cdot), \Theta(\cdot)$.
Analysis of Array Total: PRIORITY-CRCW

1. \( S = 0 \)
2. for \( i = 1 \ldots n \) in parallel do
   \[ S = S + A[i] \]
end for
Analysis of Array Total: PRIORITY-CRCW

1. \( S = 0 \)
2. for \( i = 1 \ldots n \) in parallel do
   \[ S = S + A[i] \]
   end for

**CRCW Array Total**
- Line 1. \( \Theta(1) \) time
- Line 2. One round of parallel \( \Theta(1) \) time
Analysis of Array Total: EREW

1. for $i = 1 \ldots n$ in parallel do
   \[ B[i] = A[i] \]

2. for $h = 1 \ldots k$ do
   for $1 \leq i \leq n/2^h$ in parallel do


EREW Array Total

- Line 1: Parallel $\Theta(1)$ time
- Line 2: $k = \log_2 n$ rounds of parallel $\Theta(1)$ time
- Line 3: $\Theta(1)$ time
1. for $i = 1 \ldots n$ in parallel do
   $B[i] = A[i]$

2. for $h = 1 \ldots k$ do
   for $1 \leq i \leq n/2^h$ in parallel do


EREW Array Total
   - Line 1. Parallel $\Theta(1)$ time
   - Line 2. $k = \log_2 n$ rounds of parallel $\Theta(1)$ time
   - Line 3. $\Theta(1)$ time
Notation: Work and Span; Work and Time

- In either model: Work $W = T_1$ is the total number of instructions in sequential execution.
- Span $S = T_\infty$ refers to the Multi-Threading model.
- Time $T = T_\infty$ refers to the PRAM model. It assumes we have as many processors $p$ as we need (i.e. $p = \infty$).
- In either model we define Parallelism as $Par = W/S$ or $T_1/T_\infty$ depending on which notation we use.
- Parallelism is a measure of the maximum number of processors $p$ we can use efficiently in the algorithm.
- Variables $W, S, T$ are a function of the problem size $n$. 
Work-Time scheduling principle

Another name for the greedy scheduling principle we introduced in multi-threading
Assumption: $p$ processors can do $p$ units of work in one time step,

$$T_p(n) \leq \frac{W(n)}{p} + T(n)$$

Proof: Decompose the time (span) $T = T = T_\infty$ into steps $i = 1, \ldots, T$. These are the 'unavoidably sequential' statements in the program. $W_i$ number of work operations in step $i$. Simulate $W_i$ ops in at most $\lceil W_i/p \rceil$ parallel steps on $p$ processors.

$$T_p \leq \sum_{i=1}^{T} \left\lceil \frac{W_i}{p} \right\rceil \leq \sum_{i=1}^{T} \left( \frac{W_i}{p} + 1 \right) = \frac{W}{p} + T$$
Examples

Algorithm has $T = T_\infty = 50$, $W = 10000$ on some data input. How much better is $p = 10000$ than $p = 1000$ than $p = 200$? Remember $T_\infty \leq T_p$.

$$T = T_\infty \leq T_p \leq \frac{W}{p} + T.$$ 

For $p = 200$,

- $50 \leq T_{200} \leq 100$,
- $50 \leq T_{1000} \leq 60$,
- $50 \leq T_{10000} \leq 51$ etc
Examples

Algorithm has $T(n) = \Theta(n^{2/3})$ and $W(n) = \Theta(n)$.
What is maximum useful value of $p$?

Max useful $p \sim n^{1/3}$. When $p$ is this large $\Theta(n) = \Theta(n^{2/3})$ is more than $\Theta(n^{1/3})$ wastes resources (Why?)
Ans: Still stuck with the final $\Theta(n^{2/3})$ on RHS
Algorithm has $T(n) = \Theta(n^{2/3})$ and $W(n) = \Theta(n)$. What is maximum useful value of $p$?

$$T = T_\infty \leq T_p \leq \frac{W}{p} + T.$$
Algorithm has $T(n) = \Theta(n^{2/3})$ and $W(n) = \Theta(n)$. What is maximum useful value of $p$?

$$T = T_\infty \leq T_p \leq \frac{W}{p} + T.$$  

$$T(n) = \Theta(n^{2/3}) \leq T_P \leq \frac{\Theta(n)}{p} + \Theta(n^{2/3})$$

What is max useful $p$?
Algorithm has $T(n) = \Theta(n^{2/3})$ and $W(n) = \Theta(n)$. What is maximum useful value of $p$?

$$T = T_{\infty} \leq T_p \leq \frac{W}{p} + T.$$  

$$T(n) = \Theta(n^{2/3}) \leq T_P \leq \frac{\Theta(n)}{p} + \Theta(n^{2/3})$$

What is max useful $p$? Max useful $p \sim n^{1/3}$. When $p$ is this large $\frac{\Theta(n)}{p} = \Theta(n^{2/3})$ More than $\Theta(n^{1/3})$ wastes resources (Why?)
Algorithm has $T(n) = \Theta(n^{2/3})$ and $W(n) = \Theta(n)$. What is maximum useful value of $p$?

$$T = T_{\infty} \leq T_p \leq \frac{W}{p} + T.$$  

$T(n) = \Theta(n^{2/3}) \leq T_P \leq \frac{\Theta(n)}{p} + \Theta(n^{2/3})$

What is max useful $p$?
Max useful $p \sim n^{1/3}$.
When $p$ is this large $\frac{\Theta(n)}{p} = \Theta(n^{2/3})$
More than $\Theta(n^{1/3})$ wastes resources (Why?)
Ans: Still stuck with the final $\Theta(n^{2/3})$ on RHS
Other efficiency metrics

- In the PRAM model, as in the Multi-Threading we can also use comparative measures based on the number of processors $p$ available. For example Speedup, Cost, Efficiency.
- Let $T_p(n)$ be the running time of our algorithm $A$ with $p$ processors, where $W = T_1$ is the work.
- The Cost is defined as $C_p(n) = pT_p(n)$.
- The efficiency obtained by $A$ on an input of size $n$ is

$$E_p(n) = \frac{T_1(n)}{pT_p(n)} = \frac{T_1(n)}{C_p(n)}$$

- $T_p(n)$ would usually be obtained experimentally.
There is a tricky detail about what exactly we should use to make the comparative measure of speedup.

For parallelism etc we used $W = T_1$, i.e. how our algorithm works on one processor.

The speedup measure for PRAM uses $T^*$ the run time of the best possible sequential algorithm for comparison

$T_p(n)$, running time of our algorithm $A$ with $p$ processors

The speedup obtained by $A$ on an input of size $n$ is

$$S_p(n) = \frac{T^*(n)}{T_p(n)}.$$ 

For a lot of problems, we do not know $T^*$, in which case we will use $T_1$ instead
The end of the PRAM introduction