Last Time

- Open Addressing Hashing
Today

- Multithreading
Two Styles of Threading

- Shared Memory – Every thread can access the same memory (data).
- Distributed Memory – Each thread has its own partition of the memory
Types of Parallelism

Nested Parallelism

- Call a method.
- Don’t wait for it to return.
Parallel Loops

- Just like a for loop
- Loop executions run concurrently
Parallelization keywords

- **spawn** – Start a new thread running
- **sync** – Wait here until all threads finish
- **parallel** – indicates a parallel loop
Fibonacci example for parallelism

<table>
<thead>
<tr>
<th>FIB(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if $n \leq 1$ then</td>
</tr>
<tr>
<td>return $n$</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>$x = \text{FIB}(n-1)$</td>
</tr>
<tr>
<td>$y = \text{FIB}(n-2)$</td>
</tr>
<tr>
<td>return $x + y$</td>
</tr>
<tr>
<td>end if</td>
</tr>
</tbody>
</table>
Fibonacci example for parallelism

\[
\text{FIB}(n) = \begin{cases} 
\text{return } n & \text{if } n \leq 1 \\
\text{spawn FIB}(n-1) + \text{FIB}(n-2) & \text{else}
\end{cases}
\]
Recursion of Fib(6)
While uses 17 commands, the critical path – the longest from initial strand to the final strand – is 8 units long. (Can be found using a BFS.)
Some Terminology

- **Work** is the number of calls to a function that are made ($T_1$).
- **Span** is the length of the critical path ($T_\infty$).

- The fastest a process can run on a multicore processor with $P$ threads is $T_P \geq T_1/P$.
- The **speedup** is $T_1/T_P$. If the speedup is $P$ we have perfect linear speedup.
- The **parallelism** of an algorithm is $T_1/T_\infty$. This is the maximum possible speedup that an algorithm can achieve by adding more processors.
Analyzing multithreaded algorithms

- Most of what we’ve done so far has been the analysis and optimization of work.
- Analyzing span is different. The total span of a parallel algorithm with two components A and B is \( \max(T_\infty(A), T_\infty(B)) \).

For P-Fib(x):

\[
T_\infty(n) = \max(T_\infty(n - 1), T_\infty(n - 2)) + \Theta(1) \tag{1}
\]

\[
= T_\infty(n - 1) + \Theta(1) \tag{2}
\]
Analyzing multithreaded algorithms

- Most of what we’ve done so far has been the analysis and optimization of work.
- Analyzing span is different. The total span of a parallel algorithm with two components A and B is \( \max(T_\infty(A), T_\infty(B)) \).

For P-Fib(x):

\[
T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \tag{1}
\]
\[
= T_\infty(n-1) + \Theta(1) \tag{2}
\]
\[
= \Theta(n) \tag{3}
\]

Huge Parallelism: \( T_1(n) / T_\infty(n) = \theta(\phi^n / n) \)
If the iterations of a loop don't depend on each other, they can run in parallel.

For example, multiply a matrix $A$ by a vector $x$.

$$y_i = \sum_{j=1}^{n} A_{ij}x_j$$
\[ y_i = \sum_{j=1}^{n} A_{ij}x_j \]

**Mat-Vec(A,x)**

- \( n = A\text{.rows} \)
- \( y = \text{new vector with n cells} \)
- \( \text{for } i = 1 \text{ to } n \text{ do} \)
  - \( y_i = 0 \)
- \( \text{end for} \)
- \( \text{for } i = 1 \text{ to } n \text{ do} \)
  - \( \text{for } j = 1 \text{ to } n \text{ do} \)
    - \( y_i = y_i + a_{ij}x_j \)
  - \( \text{end for} \)
- \( \text{end for} \)
\[ y_i = \sum_{j=1}^{n} A_{ij}x_j \]

**MAT-VEC(A,x)**

\[ n = A\text{.rows} \]
\[ y = \text{new vector with n cells} \]
\[ \text{for parallel } i = 1 \text{ to } n \text{ do} \]
\[ y_i = 0 \]
\[ \text{end for} \]
\[ \text{for parallel } i = 1 \text{ to } n \text{ do} \]
\[ \text{for } j = 1 \text{ to } n \text{ do} \]
\[ y_i = y_i + a_{ij}x_j \]
\[ \text{end for} \]
\[ \text{end for} \]
Race Conditions

\begin{verbatim}
\textbf{Race()}

\begin{verbatim}
x = 0
for parallel i = 1 to 2 do
  x = x + 1
end for
print x
\end{verbatim}

Need to make sure that the two threads are \textbf{independent} – no writing to memory that other threads are reading from.
\end{verbatim}
Multithreaded Merge Sort

**Merge-Sort**(A, p, r)

if \( p < r \) then

\[ q = \lfloor (p + r/2) \rfloor \]

spawn **Merge-Sort**(A, p, q)
**Merge-Sort**(A, q + 1, r)

sync

**Merge**(A, p, q, r)

end if

Spawn two recursive calls to Merge Sort.
No problem
Multithreaded Merge Sort

Work

\[ MS_1(n) = 2MS_1(n/2) + \Theta(n) \]
\[ = \Theta(n \log n) \]

Span

\[ MS_\infty(n) = MS_\infty(n/2) + \Theta(n) \]
\[ = \Theta(n) \]

Parallelization

\[ \frac{MS_1}{MS_\infty} = \frac{\Theta(n \log n)}{\Theta(n)} = \Theta(\log n) \]
Parallel Merge

- Serial Merge is dominating the performance.
- How can we parallelize merge?
- Divide-and-conquer Merge
  - Put the middle element, $z$, of the smaller of the two lists in the correct position
  - Merge the subarrays containing elements smaller than $z$
  - Merge the subarrays containing elements greater than $z$
Parallel Merge

**P-Merge**\((T, p_1, r_1, p_2, r_2, A, p_3)\)

\[
\begin{align*}
  n_1 &= r_1 - p_1 + 1 \\
  n_2 &= r_2 - p_2 + 1 \\
  \text{if } n_1 < n_2 \text{ then} & \\
  & \quad \text{swap } p\text{'s, } r\text{'s and } n\text{'s} \\
  \text{end if} \\
  \text{if } n_1 == 0 \text{ then} & \\
  & \quad \text{return} \\
  \text{else} & \\
  & \quad q_1 = \lfloor (p_1 + r_1)/2 \rfloor \\
  & \quad q_2 = \text{Binary-Search}(T[q_1], T, p_2, r_2) \\
  & \quad q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2) \quad \text{// Where to put } T[q_1] \\
  & \quad A[q_3] = T[q_1] \\
  & \quad \text{spawn } P\text{-Merge}(T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3) \\
  & \quad \text{P-Merge}(T, q_1 + 1, r_1, q_2 + 1, r_2, A, q_3 + 1) \\
  \text{sync} & \\
  \text{end if}
\]
Span

- Identify the **maximum** number of elements in the largest call to P-Merge.
- The worst case merges $n_1/2$ elements (from the larger subarray) with all $n_2$ elements (from the smaller subarray).

\[
\lfloor n_1/2 \rfloor + n_2 \leq n_1/2 + n_2/2 + n_2/2 \\
= (n_1 + n_2)/2 + n_2/2 \\
\leq n/2 + n/4 \\
= 3n/4
\]

\[
PM_\infty(n) = PM_\infty(3n/4) + \Theta(\log n) \\
= \Theta(\log^2 n)
\]
Parallel Merge

Work

\[ PM_1(n) = PM_1 = PM_1(\alpha n) + PM_1((1 - \alpha)n) + O(\log n) \]

- \( PM_1 \) is clearly \( \Omega(n) \).
- Can show that \( PM_1(n) \leq c_1 n - c_2 \log n \) for constants \( c_1, c_2 \) proving that \( PM_1 = O(n) \).
- Thus, \( PM_1 = \Theta(n) \).
Parallel Merge Sort

**P-MergeSort**(*A, p, r, B, s*)

\[
n = r - p + 1
\]

**if** *n == 1** then

\[
B[s] = A[p]
\]

**else**

\[
\text{let } T[n] \text{ be a new array}
\]
\[
q = \lfloor (p + r)/2 \rfloor
\]
\[
q' = q - p + 1
\]

**spawn** **P-Merge-Sort**(*A, p, q, T, 1*)

**P-Merge-Sort**(*A, q + 1, r, T, q' + 1*)

**sync**

**P-Merge**(*T, 1, q', q' + 1, n, B, s*)

**end if**
Parallel Merge Sort

Work

\[ PMergeSort_1(n) = 2PMergeSort_1(n/2) + PM_1(n) \]
\[ = 2PMergeSort_1(n/2) + \Theta(n) \]
\[ = \Theta(n \log n) \]
Parallel Merge Sort

Span

\[ PMergeSort_\infty(n) = PMergeSort_\infty(n/2) + PM_{infty}(n) \]
\[ = PMergeSort_\infty(n/2) + \Theta(\log^2 n) \]
\[ = \Theta(\log^3 n) \]

Parallelism

\[ PMergeSort_1(n) / PMergeSort_\infty(n) = \Theta(n \log n) / \Theta(\log^3 n) \]
\[ = \Theta(n / \log^2 n) \]
Bye

- Next time
- NP-Completeness