CSE 638: Advanced Algorithms

Lectures 2, 3 & 4
(Analytical Modeling of Parallel Algorithms)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2013
Parallel running time on \( p \) processing elements,

\[
T_P = t_{\text{end}} - t_{\text{start}},
\]

where, \( t_{\text{start}} \) = starting time of the processing element that starts first

\( t_{\text{end}} \) = termination time of the processing element that finishes last

Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition
Sources of overhead (w.r.t. serial execution)

- Interprocess interaction
  - Interact and communicate data (e.g., intermediate results)

- Idling
  - Due to load imbalance, synchronization, presence of serial computation, etc.

- Excess computation
  - Fastest serial algorithm may be difficult/impossible to parallelize
Overhead function or total parallel overhead,

\[ T_o = pT_p - T, \]

where, \( p \) = number of processing elements

\( T \) = time spent doing useful work

( often execution time of the fastest serial algorithm )
Let $T_p = \text{running time using } p \text{ identical processing elements}$

Speedup, $S_p = \frac{T_1}{T_p}$

Theoretically, $S_p \leq p$ \text{(why?)}

*Perfect or linear or ideal* speedup if $S_p = p$
Consider adding $n$ numbers using $n$ identical processing elements.

Serial runtime, $T_1 = \Theta(n)$

Parallel runtime, $T_n = \Theta(\log n)$

Speedup, $S_n = \frac{T_1}{T_n} = \Theta\left(\frac{n}{\log n}\right)$

Speedup not ideal.
Superlinear Speedup

Theoretically, $S_p \leq p$

But in practice *superlinear speedup* is sometimes observed, that is, $S_p > p$ (why?)

Reasons for superlinear speedup

- Cache effects
- Exploratory decomposition
Superlinear Speedup
(Cache Effects)

Let cache access latency = 2 ns
DRAM access latency = 100 ns

Suppose we want solve a problem instance that executes \( k \) FLOPs.

**With 1 Core:** Suppose cache hit rate is 80%.
If the computation performs 1 FLOP/memory access, then each FLOP will take \( 2 \times 0.8 + 100 \times 0.2 = 21.6 \) ns to execute.

**With 2 Cores:** Cache hit rate will improve. (why?)
Suppose cache hit rate is now 90%.
Then each FLOP will take \( 2 \times 0.9 + 100 \times 0.1 = 11.8 \) ns to execute.
Since now each core will execute only \( k / 2 \) FLOPs,
\[
\text{Speedup, } S_2 = \frac{k \times 21.6}{(k/2) \times 11.8} \approx 3.66 > 2!
\]
Superlinear Speedup
(Due to Exploratory Decomposition)

Consider searching an array of $2n$ unordered elements for a specific element $x$.

Suppose $x$ is located at array location $k > n$ and $k$ is odd.

Serial runtime, $T_1 = k$

Parallel running time with $n$ processing elements, $T_n = 1$

Speedup, $S_n = \frac{T_1}{T_n} = k > n$

Speedup is superlinear!
Parallelism & Span Law

We defined, $T_p = \text{runtime on } p \text{ identical processing elements}$

Then span, $T_\infty = \text{runtime on an infinite number of identical processing elements}$

Parallelism, $P = \frac{T_1}{T_\infty}$

Parallelism is an upper bound on speedup, i.e., $S_p \leq P$ (why?)

**Span Law**

$T_p \geq T_\infty$
Work Law

The cost of solving (or work performed for solving) a problem:

On a Serial Computer: is given by $T_1$

On a Parallel Computer: is given by $pT_p$

$$\text{Work Law}$$

$$T_p \geq \frac{T_1}{p}$$
Work Optimality

Let $T_s$ = runtime of the optimal or the fastest known serial algorithm

A parallel algorithm is *cost-optimal* or *work-optimal* provided

$$pT_p = \Theta(T_s)$$

Our algorithm for adding $n$ numbers using $n$ identical processing elements is clearly not work optimal.
Adding $n$ Numbers Work-Optimality

We reduce the number of processing elements which in turn increases the granularity of the subproblem assigned to each processing element.

Suppose we use $p$ processing elements.

First each processing element locally adds its $\frac{n}{p}$ numbers in time $\Theta\left(\frac{n}{p}\right)$.

Then $p$ processing elements adds these $p$ partial sums in time $\Theta(\log p)$.

Thus $T_p = \Theta\left(\frac{n}{p} + \log p\right)$, and $T_s = \Theta(n)$.

So the algorithm is work-optimal provided $n = \Omega(p \log p)$. 

Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition
Scaling Laws
Scaling of Parallel Algorithms

( Amdahl’s Law )

Suppose only a fraction $f$ of a computation can be parallelized.

Then parallel running time, $T_p \geq (1 - f)T_1 + f \frac{T_1}{p}$

Speedup, $S_p = \frac{T_1}{T_p} \leq \frac{p}{f + (1-f)p} = \frac{1}{(1-f) + \frac{f}{p}} \leq \frac{1}{1-f}$
Scaling of Parallel Algorithms (Amdahl’s Law)

Suppose only a fraction $f$ of a computation can be parallelized.

Speedup, $S_p = \frac{T_1}{T_p} \leq \frac{1}{(1-f) + \frac{f}{p}} \leq \frac{1}{1-f}$

Scaling of Parallel Algorithms
(Gustafson-Barsis’ Law)

Suppose only a fraction $f$ of a computation was parallelized.

Then serial running time, $T_1 = (1 - f)T_p + pfT_p$

Speedup, $S_p = \frac{T_1}{T_p} = \frac{(1-f)T_p + pfT_p}{T_p} = 1 + (p - 1)f$
Suppose only a fraction \( f \) of a computation was parallelized.

Speedup, \( S_p = \frac{T}{T_p} \leq \frac{T_1}{T_p} = \frac{(1-f)T_p + pfT_p}{T_p} = 1 + (p - 1)f \)

**Strong Scaling vs. Weak Scaling**

**Strong Scaling**

How $T_p$ (or $S_p$) varies with $p$ when the problem size is fixed.

**Weak Scaling**

How $T_p$ (or $S_p$) varies with $p$ when the problem size per processing element is fixed.
A parallel algorithm is called *scalable* if its efficiency can be maintained at a fixed value by simultaneously increasing the number of processing elements and the problem size.

Scalability reflects a parallel algorithm’s ability to utilize increasing processing elements effectively.

**Efficiency,** \( E_p = \frac{S_p}{p} = \frac{T_1}{pT_p} \)
In order to keep $E_p$ fixed at a constant $k$, we need

$$E_p = k \Rightarrow \frac{T_1}{pT_p} = k \Rightarrow T_1 = kpT_p$$

For the algorithm that adds $n$ numbers using $p$ processing elements:

$$T_1 = n \quad \text{and} \quad T_p = \frac{n}{p} + 2 \log p$$

So in order to keep $E_p$ fixed at $k$, we must have:

$$n = kp \left( \frac{n}{p} + 2 \log p \right) \Rightarrow n = \frac{2k}{1 - k} p \log p$$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$p = 1$</th>
<th>$p = 4$</th>
<th>$p = 8$</th>
<th>$p = 16$</th>
<th>$p = 32$</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1.0</td>
<td>0.80</td>
<td>0.57</td>
<td>0.33</td>
<td>0.17</td>
</tr>
<tr>
<td>192</td>
<td>1.0</td>
<td>0.92</td>
<td>0.80</td>
<td>0.60</td>
<td>0.38</td>
</tr>
<tr>
<td>320</td>
<td>1.0</td>
<td>0.95</td>
<td>0.87</td>
<td>0.71</td>
<td>0.50</td>
</tr>
<tr>
<td>512</td>
<td>1.0</td>
<td>0.97</td>
<td>0.91</td>
<td>0.80</td>
<td>0.62</td>
</tr>
</tbody>
</table>

**Fig:** Efficiency for adding $n$ numbers using $p$ processing elements

**Source:** Grama et al., “Introduction to Parallel Computing”, 2nd Edition
Greedy Scheduling Theorem
Loop Parallelism (Data Parallelism)

\[
\begin{pmatrix}
a_{11} & a_{12} & \ldots & a_{1n} \\
a_{21} & a_{22} & \ldots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \ldots & a_{nn}
\end{pmatrix}
\]

\[
\begin{pmatrix}
a_{11} & a_{21} & \ldots & a_{n1} \\
a_{12} & a_{22} & \ldots & a_{n2} \\
\vdots & \vdots & \ddots & \vdots \\
a_{1n} & a_{2n} & \ldots & a_{nn}
\end{pmatrix}
\]

in-place transpose

for (int i = 1; i < n; ++i)
    for (int j = 0; j < i; ++j)
    {
        double t = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = t;
    }

Serial Code

```
parallel for (int i = 1; i < n; ++i)
    for (int j = 0; j < i; ++j)
    {
        double t = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = t;
    }
```

Parallel Code

Allows all iterations of the loop to be executed in parallel.
Loop Parallelism (Data Parallelism)

\[
\begin{pmatrix}
    a_{11} & a_{12} & \ldots & a_{1n} \\
    a_{21} & a_{22} & \ldots & a_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{n1} & a_{n2} & \ldots & a_{nn}
\end{pmatrix}
\]

\[
\begin{pmatrix}
    a_{11} & a_{21} & \ldots & a_{n1} \\
    a_{12} & a_{22} & \ldots & a_{n2} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{1n} & a_{2n} & \ldots & a_{nn}
\end{pmatrix}
\]

in-place transpose

Serial Code

```c
for ( int i = 1; i < n; ++i )
    for ( int j = 0; j < i; ++j )
    {
        double t = A[ i ][ j ];
        A[ i ][ j ] = A[ j ][ i ];
        A[ j ][ i ] = t;
    }
```

Parallel Code

```c
parallel for ( int i = 1; i < n; ++i )
    parallel for ( int j = 0; j < i; ++j )
    {
        double t = A[ i ][ j ];
        A[ i ][ j ] = A[ j ][ i ];
        A[ j ][ i ] = t;
    }
```

Allows all iterations of the loop to be executed in parallel.
Nested Parallelism (Task Parallelism)

Serial Code

```c
int comb ( int n, int r )
{
    if ( r > n ) return 0;
    if ( r == 0 || r == n ) return 1;
    int x, y;
    x = comb( n - 1, r - 1 );
    y = comb( n - 1, r );
    return ( x + y );
}
```

Parallel Code

```c
int comb ( int n, int r )
{
    if ( r > n ) return 0;
    if ( r == 0 || r == n ) return 1;
    int x, y;
    x = spawn comb( n - 1, r - 1 );
    y = comb( n - 1, r );
    sync;
    return ( x + y );
}
```

Grant permission to execute the called (spawned) function in parallel with the caller.

Control cannot pass this point until all spawned children have returned.
Loop Parallelism (Data Parallelism)

\[
\begin{pmatrix}
  a_{11} & a_{12} & \ldots & a_{1n} \\
  a_{21} & a_{22} & \ldots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{n1} & a_{n2} & \ldots & a_{nn}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
  a_{11} & a_{21} & \ldots & a_{n1} \\
  a_{12} & a_{22} & \ldots & a_{n2} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{1n} & a_{2n} & \ldots & a_{nn}
\end{pmatrix}
\]

in-place transpose

\[
\begin{align*}
  &\text{Serial Code} \\
  \text{for ( int } i = 1; i < n; ++i ) \\
  &\quad \text{for ( int } j = 0; j < i; ++j ) \\
  &\quad \quad \{
  &\quad \quad \quad \text{double } t = A[ i ][ j ]; \\
  &\quad \quad \quad A[ i ][ j ] = A[ j ][ i ]; \\
  &\quad \quad \quad A[ j ][ i ] = t;
  &\quad \quad \}\n\end{align*}
\]

\[
\begin{align*}
  &\text{Parallel Code} \\
  \text{parallel for ( int } i = 1; i < n; ++i ) \\
  &\quad \text{for ( int } j = 0; j < i; ++j ) \\
  &\quad \quad \{
  &\quad \quad \quad \text{double } t = A[ i ][ j ]; \\
  &\quad \quad \quad A[ i ][ j ] = A[ j ][ i ]; \\
  &\quad \quad \quad A[ j ][ i ] = t;
  &\quad \quad \}\n\end{align*}
\]

Allows all iterations of the loop to be executed in parallel.

Can be converted to spawns and syncs using recursive divide-and-conquer.
int comb ( int n, int r )
{
    if ( r > n ) return 0;
    if ( r == 0 || r == n ) return 1;
    int x, y;
    x = spawn comb( n - 1, r - 1 );
    y = comb( n - 1, r );
    sync;
    return ( x + y );
}

Parallel Execution Model
int comb ( int n, int r )
{
    if ( r > n ) return 0;
    if ( r == 0 || r == n ) return 1;
    int x, y;
    x = spawn comb( n - 1, r - 1 );
    y = comb( n - 1, r );
    sync;
    return ( x + y );
}
Parallel Execution Model

```c
int comb ( int n, int r )
{
    if ( r > n ) return 0;
    if ( r == 0 || r == n ) return 1;
    int x, y;
    x = spawn comb( n - 1, r - 1 );
    y = comb( n - 1, r );
    sync;
    return ( x + y );
}
```

(3, 1) → (4, 2)
```c
int comb ( int n, int r )
{
    if ( r > n ) return 0;
    if ( r == 0 || r == n ) return 1;
    int x, y;
    x = spawn comb( n – 1, r - 1 );
    y = comb( n – 1, r );
    sync;
    return ( x + y );
}
```
int comb ( int n, int r ) {
    if ( r > n ) return 0;
    if ( r == 0 || r == n ) return 1;
    int x, y;
    x = spawn comb( n - 1, r - 1 );
    y = comb( n - 1, r );
    sync;
    return ( x + y );
}
```c
int comb ( int n, int r )
{
    if ( r > n ) return 0;
    if ( r == 0 || r == n ) return 1;
    int x, y;
    x = spawn comb( n - 1, r - 1 );
    y = comb( n - 1, r );
    sync;
    return ( x + y );
}
```
int comb ( int n, int r )
{
    if ( r > n ) return 0;
    if ( r == 0 || r == n ) return 1;
    int x, y;
    x = spawn comb( n – 1, r - 1 );
    y = comb( n – 1, r );
    sync;
    return ( x + y );
}
Parallel Execution Model

```c
int comb ( int n, int r ) {
    if ( r > n ) return 0;
    if ( r == 0 || r == n ) return 1;
    int x, y;
    x = spawn comb( n - 1, r - 1 );
    y = comb( n - 1, r );
    sync;
    return ( x + y );
}
```
int comb ( int n, int r )
{
    if ( r > n ) return 0;
    if ( r == 0 || r == n ) return 1;
    int x, y;
    x = spawn comb( n - 1, r - 1 );
    y = comb( n - 1, r );
    sync;
    return ( x + y );
}
int comb ( int n, int r )
{
    if ( r > n ) return 0;
    if ( r == 0 || r == n ) return 1;
    int x, y;
    x = spawn comb( n - 1, r - 1 );
    y = comb( n - 1, r );
    sync;
    return ( x + y );
}
A parallel instruction stream is represented by a DAG $G = (V, E)$.

Each vertex $v \in V$ is a strand – a maximal sequence of instructions ending with a spawn, sync or return (implicit or explicit).

Each edge $e \in E$ is a spawn, call, continue or return edge.
Parallelism in \text{comb}(4, 2)

\text{work: } T_1 = \text{#nodes in the DAG} = 21

\text{span: } T_\infty = \text{#nodes on the longest path in the DAG} = 9

\text{parallelism} = \frac{T_1}{T_\infty} = \frac{21}{9} \approx 2.33

Only marginal performance gains with more than 2 cores!
A runtime/online scheduler maps tasks to processing elements dynamically at runtime.

The map is called a *schedule*.

An offline scheduler prepares the schedule prior to the actual execution of the program.
A strand / task is called *ready* provided all its parents (if any) have already been executed.

- grey: executed task
- green: ready to be executed
- white: not yet ready

A *greedy scheduler* tries to perform as much work as possible at every step.
A Centralized Greedy Scheduler

Let \( p \) = number of cores

At every step:
- if \( \geq p \) tasks are ready: execute any \( p \) of them (complete step)
- if \( < p \) tasks are ready: execute all of them (incomplete step)
A Centralized Greedy Scheduler

Let $p = \text{number of cores}$

At every step:

- if $\geq p$ tasks are ready: execute any $p$ of them (complete step)
- if $< p$ tasks are ready: execute all of them (incomplete step)
Let $p = \text{number of cores}$

At every step:

- if $\geq p$ tasks are ready:
  execute any $p$ of them
  (complete step)

- if $< p$ tasks are ready:
  execute all of them
  (incomplete step)
A Centralized Greedy Scheduler

Let $p =$ number of cores

At every step:
- if $\geq p$ tasks are ready:
  execute any $p$ of them
  ( complete step )
- if $< p$ tasks are ready:
  execute all of them
  ( incomplete step )
A Centralized Greedy Scheduler

Let $p =$ number of cores

At every step:

- if $\geq p$ tasks are ready: execute any $p$ of them (complete step)
- if $< p$ tasks are ready: execute all of them (incomplete step)
A Centralized Greedy Scheduler

Let \( p \) = number of cores

At every step:
- if \( \geq p \) tasks are ready:
  execute any \( p \) of them
  (complete step)
- if \(< p \) tasks are ready:
  execute all of them
  (incomplete step)
A Centralized Greedy Scheduler

Let $p =$ number of cores

At every step:
- if $\geq p$ tasks are ready:
  execute any $p$ of them  
  (complete step)
- if $< p$ tasks are ready:
  execute all of them  
  (incomplete step)
A Centralized Greedy Scheduler

Let $p = \text{number of cores}$

At every step:

- if $\geq p$ tasks are ready:
  execute any $p$ of them
  (complete step)

- if $< p$ tasks are ready:
  execute all of them
  (incomplete step)
A Centralized Greedy Scheduler

Let $p = \text{number of cores}$

At every step:

- if $\geq p$ tasks are ready: execute any $p$ of them (complete step)
- if $< p$ tasks are ready: execute all of them (incomplete step)
A Centralized Greedy Scheduler

Let $p =$ number of cores

At every step:

- if $\geq p$ tasks are ready:
  execute any $p$ of them
  (complete step)

- if $< p$ tasks are ready:
  execute all of them
  (incomplete step)
A Centralized Greedy Scheduler

Let $p =$ number of cores

At every step:

- if $\geq p$ tasks are ready:
  execute any $p$ of them
  ( complete step )

- if $< p$ tasks are ready:
  execute all of them
  ( incomplete step )
Let $p = \text{number of cores}$

At every step:

- if $\geq p$ tasks are ready: execute any $p$ of them (complete step)
- if $< p$ tasks are ready: execute all of them (incomplete step)
A Centralized Greedy Scheduler

Let $p =$ number of cores

At every step:
- if $\geq p$ tasks are ready:
  execute any $p$ of them
  ( complete step )
- if $< p$ tasks are ready:
  execute all of them
  ( incomplete step )
Theorem [Graham’68, Brent’74]:
For any greedy scheduler,
\[ T_p \leq \frac{T_1}{p} + T_\infty \]

Proof:
\[ T_p = \text{#complete steps} + \text{#incomplete steps} \]
- Each complete step performs \( p \) work:
  \[ \text{#complete steps} \leq \frac{T_1}{p} \]
- Each incomplete step reduces the span by 1:
  \[ \text{#incomplete steps} \leq T_\infty \]
Corollary 1: For any greedy scheduler $T_p \leq 2T_p^*$, where $T_p^*$ is the running time due to optimal scheduling on $p$ processing elements.

Proof:

Work law: $T_p^* \geq \frac{T_1}{p}$

Span law: $T_p^* \geq T_\infty$

∴ From Graham-Brent Theorem:

$$T_p \leq \frac{T_1}{p} + T_\infty \leq T_p^* + T_p^* = 2T_p^*$$
Optimality of the Greedy Scheduler

Corollary 2: Any greedy scheduler achieves $S_p \approx p$ (i.e., nearly linear speedup) provided parallelism, $P = \frac{T_1}{T_\infty} \gg p$.

Proof:

Given, $P = \frac{T_1}{T_\infty} \gg p \implies \frac{T_1}{p} \gg T_\infty$

$\therefore$ From Graham-Brent Theorem:

$$T_p \leq \frac{T_1}{p} + T_\infty \approx \frac{T_1}{p}$$

$$\implies \frac{T_1}{T_p} \approx p \implies S_p \approx p$$
Parallel Matrix Multiplication
**Parallel Iterative MM**

**Iter-MM** \((Z, X, Y)\)  \{X, Y, Z are \(n \times n\) matrices, where \(n\) is a positive integer\}

1. \(\text{for } i \leftarrow 1 \text{ to } n \text{ do}\)
2. \(\quad \text{for } j \leftarrow 1 \text{ to } n \text{ do}\)
3. \(\quad Z[i][j] \leftarrow 0\)
4. \(\quad \text{for } k \leftarrow 1 \text{ to } n \text{ do}\)
5. \(\quad Z[i][j] \leftarrow Z[i][j] + X[i][k] \cdot Y[k][j]\)

**Par-Iter-MM** \((Z, X, Y)\)  \{X, Y, Z are \(n \times n\) matrices, where \(n\) is a positive integer\}

1. \(\text{parallel for } i \leftarrow 1 \text{ to } n \text{ do}\)
2. \(\quad \text{parallel for } j \leftarrow 1 \text{ to } n \text{ do}\)
3. \(\quad Z[i][j] \leftarrow 0\)
4. \(\quad \text{for } k \leftarrow 1 \text{ to } n \text{ do}\)
5. \(\quad Z[i][j] \leftarrow Z[i][j] + X[i][k] \cdot Y[k][j]\)
Parallel Iterative MM

Par-Iter-MM (Z, X, Y) \{ X, Y, Z are n \times n matrices, where n is a positive integer \}

1. parallel for i ← 1 to n do
2. parallel for j ← 1 to n do
3. Z[i][j] ← 0
4. for k ← 1 to n do
5. Z[i][j] ← Z[i][j] + X[i][k] \cdot Y[k][j]

Work: \( T_1(n) = \Theta(n^3) \)

Span: \( T_\infty(n) = \Theta(n) \)

Parallel Running Time: \( T_p(n) = O\left(\frac{T_1(n)}{p} + T_\infty(n)\right) = O\left(\frac{n^3}{p} + n\right) \)

Parallelism: \( \frac{T_1(n)}{T_\infty(n)} = \Theta(n^2) \)
Parallel Recursive MM

\[
\begin{array}{c}
\begin{array}{c}
\text{Z} \\
\begin{array}{c}
Z_{11} & Z_{12} \\
Z_{21} & Z_{22}
\end{array}
\end{array}
\end{array}
= 
\begin{array}{c}
\begin{array}{c}
X \\
\begin{array}{c}
X_{11} & X_{12} \\
X_{21} & X_{22}
\end{array}
\end{array}
\end{array}
\times 
\begin{array}{c}
\begin{array}{c}
Y \\
\begin{array}{c}
Y_{11} & Y_{12} \\
Y_{21} & Y_{22}
\end{array}
\end{array}
\end{array}
\end{array}
\]
Parallel Recursive MM

Par-Rec-MM ( Z, X, Y )
{ X, Y, Z are n × n matrices, where n = 2^k for integer k ≥ 0 }

1. if n = 1 then
2.   Z ← Z + X · Y
3. else
4.    spawn Par-Rec-MM ( Z_{11}, X_{11}, Y_{11} )
5.    spawn Par-Rec-MM ( Z_{12}, X_{11}, Y_{12} )
6.    spawn Par-Rec-MM ( Z_{21}, X_{21}, Y_{11} )
7.    Par-Rec-MM ( Z_{21}, X_{21}, Y_{12} )
8.    sync
9.    spawn Par-Rec-MM ( Z_{11}, X_{12}, Y_{21} )
10.   spawn Par-Rec-MM ( Z_{12}, X_{12}, Y_{22} )
11.   spawn Par-Rec-MM ( Z_{21}, X_{22}, Y_{21} )
12.   Par-Rec-MM ( Z_{22}, X_{22}, Y_{22} )
13.   sync
14. endif
Parallel Recursive MM

Par-Rec-MM (Z, X, Y) { X, Y, Z are n × n matrices, where n = 2^k for integer k ≥ 0 }

1. if n = 1 then
2. Z ← Z + X · Y
3. else
4. spawn Par-Rec-MM (Z_{11}, X_{11}, Y_{11})
5. spawn Par-Rec-MM (Z_{12}, X_{11}, Y_{12})
6. spawn Par-Rec-MM (Z_{21}, X_{21}, Y_{11})
7. Par-Rec-MM (Z_{21}, X_{21}, Y_{12})
8. sync
9. spawn Par-Rec-MM (Z_{11}, X_{12}, Y_{21})
10. spawn Par-Rec-MM (Z_{12}, X_{12}, Y_{22})
11. spawn Par-Rec-MM (Z_{21}, X_{22}, Y_{21})
12. Par-Rec-MM (Z_{22}, X_{22}, Y_{22})
13. sync
14. endif

Work:

\[ T_1(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 8T_1(\frac{n}{2}) + \Theta(1), & \text{otherwise.} \end{cases} \]

\[ = \Theta(n^3) \quad \text{[MT Case 1]} \]

Span:

\[ T_\infty(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 2T_\infty(\frac{n}{2}) + \Theta(1), & \text{otherwise.} \end{cases} \]

\[ = \Theta(n) \quad \text{[MT Case 1]} \]

Parallelism: \[ \frac{T_1(n)}{T_\infty(n)} = \Theta(n^2) \]

Additional Space:

\[ s_\infty(n) = \Theta(1) \]
Recursive MM with More Parallelism

\[ Z_{11} \quad Z_{12} \]

\[ Z_{21} \quad Z_{22} \]

\[ X_{11}Y_{11} + X_{12}Y_{21} \quad X_{11}Y_{12} + X_{12}Y_{22} \]

\[ X_{21}Y_{11} + X_{22}Y_{21} \quad X_{21}Y_{12} + X_{22}Y_{22} \]

\[ X_{12}Y_{21} \quad X_{12}Y_{22} \]

\[ X_{22}Y_{21} \quad X_{22}Y_{22} \]
Recursive MM with More Parallelism

\[ \text{Par-Rec-MM}_2 \left( Z, X, Y \right) \quad \{ X, Y, Z \text{ are } n \times n \text{ matrices, where } n = 2^k \text{ for integer } k \geq 0 \} \]

1. \textit{if} \ n = 1 \ \textit{then}
2. \quad Z \leftarrow Z + X \cdot Y
3. \textit{else} \quad \{ T \text{ is a temporary } n \times n \text{ matrix} \}
4. \quad \text{spawn} \ \text{Par-Rec-MM}_2 \left( Z_{11}, X_{11}, Y_{11} \right)
5. \quad \text{spawn} \ \text{Par-Rec-MM}_2 \left( Z_{12}, X_{11}, Y_{12} \right)
6. \quad \text{spawn} \ \text{Par-Rec-MM}_2 \left( Z_{21}, X_{21}, Y_{11} \right)
7. \quad \text{spawn} \ \text{Par-Rec-MM}_2 \left( Z_{21}, X_{21}, Y_{12} \right)
8. \quad \text{spawn} \ \text{Par-Rec-MM}_2 \left( T_{11}, X_{12}, Y_{21} \right)
9. \quad \text{spawn} \ \text{Par-Rec-MM}_2 \left( T_{12}, X_{12}, Y_{22} \right)
10. \quad \text{spawn} \ \text{Par-Rec-MM}_2 \left( T_{21}, X_{22}, Y_{21} \right)
11. \quad \text{spawn} \ \text{Par-Rec-MM}_2 \left( T_{22}, X_{22}, Y_{22} \right)
12. \quad \text{sync}
13. \quad \text{parallel for } i \leftarrow 1 \ \text{to} \ n \ \text{do}
14. \quad \quad \text{parallel for } j \leftarrow 1 \ \text{to} \ n \ \text{do}
15. \quad \quad \quad Z[i][j] \leftarrow Z[i][j] + T[i][j]
16. \quad \quad \text{endif}
Recursive MM with More Parallelism

\[
\text{Par-Rec-MM2 (} Z, X, Y \text{)} \quad \{ X, Y, Z \text{ are } n \times n \text{ matrices, where } n = 2^k \text{ for integer } k \geq 0 \}
\]

1. if \( n = 1 \) then
2. \( Z \leftarrow Z + X \cdot Y \)
3. else \{ \( T \) is a temporary \( n \times n \) matrix \}
4. spawn Par-Rec-MM2 (\( Z_{11}, X_{11}, Y_{11} \))
5. spawn Par-Rec-MM2 (\( Z_{12}, X_{11}, Y_{12} \))
6. spawn Par-Rec-MM2 (\( Z_{21}, X_{21}, Y_{11} \))
7. spawn Par-Rec-MM2 (\( Z_{21}, X_{21}, Y_{12} \))
8. spawn Par-Rec-MM2 (\( T_{11}, X_{12}, Y_{21} \))
9. spawn Par-Rec-MM2 (\( T_{12}, X_{12}, Y_{22} \))
10. spawn Par-Rec-MM2 (\( T_{21}, X_{22}, Y_{21} \))
11. spawn Par-Rec-MM2 (\( T_{22}, X_{22}, Y_{22} \))
12. sync
13. parallel for \( i \leftarrow 1 \) to \( n \) do
14. \quad parallel for \( j \leftarrow 1 \) to \( n \) do
15. \quad \( Z[i][j] \leftarrow Z[i][j] + T[i][j] \)
16. endif

Work:
\[
T_1(n) = \begin{cases} 
\Theta(1), & \text{if } n = 1, \\
8T_1\left(\frac{n}{2}\right) + \Theta(n^2), & \text{otherwise.} 
\end{cases}
\]
\[
= \Theta(n^3) \quad \text{[MT Case 1]} 
\]

Span:
\[
T_\infty(n) = \begin{cases} 
\Theta(1), & \text{if } n = 1, \\
T_\infty\left(\frac{n}{2}\right) + \Theta(\log n), & \text{otherwise.} 
\end{cases}
\]
\[
= \Theta(\log^2 n) \quad \text{[MT Case 2]} 
\]

Parallelism:
\[
\frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{n^3}{\log^2 n}\right)
\]

Additional Space:
\[
s_\infty(n) = \begin{cases} 
\Theta(1), & \text{if } n = 1, \\
8s_\infty\left(\frac{n}{2}\right) + \Theta(n^2), & \text{otherwise.} 
\end{cases}
\]
\[
= \Theta(n^3) \quad \text{[MT Case 1]} 
\]
Parallel Merge Sort
**Parallel Merge Sort**

\[
\text{Merge-Sort} \ (A, \ p, \ r) \quad \{ \text{sort the elements in } A[p \ldots r] \}
\]

1. if \( p < r \) then
2. \( q \leftarrow \lfloor (p + r) / 2 \rfloor \)
3. \( \text{Merge-Sort} \ (A, \ p, \ q) \)
4. \( \text{Merge-Sort} \ (A, \ q + 1, \ r) \)
5. \( \text{Merge} \ (A, \ p, \ q, \ r) \)

---

\[
\text{Par-Merge-Sort} \ (A, \ p, \ r) \quad \{ \text{sort the elements in } A[p \ldots r] \}
\]

1. if \( p < r \) then
2. \( q \leftarrow \lfloor (p + r) / 2 \rfloor \)
3. spawn \( \text{Merge-Sort} \ (A, \ p, \ q) \)
4. \( \text{Merge-Sort} \ (A, \ q + 1, \ r) \)
5. sync
6. \( \text{Merge} \ (A, \ p, \ q, \ r) \)
Parallel Merge Sort

Par-Merge-Sort (A, p, r) { sort the elements in A[p...r] }

1. if p < r then
2. q ← ⌊(p + r) / 2⌋
3. spawn Merge-Sort (A, p, q)
4. Merge-Sort (A, q + 1, r)
5. sync
6. Merge (A, p, q, r)

Work: \[ T_1(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 2T_1 \left(\frac{n}{2}\right) + \Theta(n), & \text{otherwise}. \end{cases} \]

\[ = \Theta(n \log n) \quad \text{[MT Case 2]} \]

Span: \[ T_\infty(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ T_\infty \left(\frac{n}{2}\right) + \Theta(n), & \text{otherwise}. \end{cases} \]

\[ = \Theta(n) \quad \text{[MT Case 3]} \]

Parallelism: \[ \frac{T_1(n)}{T_\infty(n)} = \Theta(\log n) \]
Parallel Merge

\[ n_1 = r_1 - p_1 + 1 \quad \quad n_2 = r_2 - p_2 + 1 \]

subarrays to merge:

\[ T[p_1..r_1] \quad \quad T[p_2..r_2] \]

suppose: \( n_1 \geq n_2 \)

merged output:

\[ n_3 = r_3 - p_3 + 1 = n_1 + n_2 \]

Source: Cormen et al., "Introduction to Algorithms", 3rd Edition
Parallel Merge

\[ n_1 = r_1 - p_1 + 1 \quad \text{and} \quad n_2 = r_2 - p_2 + 1 \]

subarrays to merge:

\[ T[p_1...r_1] \quad \text{and} \quad T[p_2...r_2] \]

suppose: \( n_1 \geq n_2 \)

merged output:

\[ A[p_3...r_3] \]

\[ n_3 = r_3 - p_3 + 1 = n_1 + n_2 \]

**Step 1:** Find \( x = T[q_1] \), where \( q_1 \) is the midpoint of \( T[p_1...r_1] \)

Source: Cormen et al., "Introduction to Algorithms", 3rd Edition
Parallel Merge

\[ n_1 = r_1 - p_1 + 1 \quad n_2 = r_2 - p_2 + 1 \]

subarrays to merge:

\[ T[p_1..r_1] \quad T[p_2..r_2] \]

suppose: \( n_1 \geq n_2 \)

merged output:

\[ A[p_3..r_3] \]

\[ n_3 = r_3 - p_3 + 1 = n_1 + n_2 \]

Step 2: Use binary search to find the index \( q_2 \) in subarray \( T[p_2..r_2] \) so that the subarray would still be sorted if we insert \( x \) between \( T[q_2 - 1] \) and \( T[q_2] \).
**Parallel Merge**

\[
\begin{align*}
    n_1 &= r_1 - p_1 + 1 \\
    n_2 &= r_2 - p_2 + 1
\end{align*}
\]

subarrays to merge:

\[
T[p_1...r_1] \quad T[p_2...r_2]
\]

\[
T \quad \cdots \quad \leq x \quad \boxed{x} \quad \geq x \quad \cdots \quad < x \quad \geq x \quad \cdots
\]

suppose: \( n_1 \geq n_2 \)

\[
A \quad \cdots \quad \leq x \quad \boxed{x} \quad \geq x \quad \cdots
\]

merged output:

\[
A[p_3...r_3] \\
    n_3 = r_3 - p_3 + 1 = n_1 + n_2
\]

**Step 3:** Copy \( x \) to \( A[q_3] \), where \( q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2) \)
Parallel Merge

\[ n_1 = r_1 - p_1 + 1 \quad \text{and} \quad n_2 = r_2 - p_2 + 1 \]

subarrays to merge: \( T[p_1..r_1] \) \( T[p_2..r_2] \)

Suppose: \( n_1 \geq n_2 \)

merged output: \( A[p_3..r_3] \)

\[ n_3 = r_3 - p_3 + 1 = n_1 + n_2 \]

Perform the following two steps in parallel.

**Step 4(a):** Recursively merge \( T[p_1..q_1 - 1] \) with \( T[p_2..q_2 - 1] \), and place the result into \( A[p_3..q_3 - 1] \)
### Parallel Merge

\[
\begin{align*}
 n_1 &= r_1 - p_1 + 1 \\
 n_2 &= r_2 - p_2 + 1 \\
 T[p_1 \ldots r_1] &\quad T[p_2 \ldots r_2]
\end{align*}
\]

**subarrays to merge:**

**merged output:**

\[
\begin{align*}
 n_3 &= r_3 - p_3 + 1 = n_1 + n_2
\end{align*}
\]

Perform the following two steps in parallel.

**Step 4(a):** Recursively merge \(T[p_1 \ldots q_1 - 1]\) with \(T[p_2 \ldots q_2 - 1]\), and place the result into \(A[p_3 \ldots q_3 - 1]\)

**Step 4(b):** Recursively merge \(T[q_1 + 1 \ldots r_1]\) with \(T[q_2 + 1 \ldots r_2]\), and place the result into \(A[q_3 + 1 \ldots r_3]\)

---

*Source: Cormen et al., “Introduction to Algorithms,” 3rd Edition*
Parallel Merge

**Par-Merge** (T, p₁, r₁, p₂, r₂, A, p₃)

1. \( n₁ \leftarrow r₁ - p₁ + 1, \quad n₂ \leftarrow r₂ - p₂ + 1 \)
2. \textbf{if} \( n₁ < n₂ \) \textbf{then}
3. \( p₁ \leftarrow p₂, \quad r₁ \leftarrow r₂, \quad n₁ \leftarrow n₂ \)
4. \textbf{if} \( n₁ = 0 \) \textbf{then return}
5. \textbf{else}
6. \( q₁ \leftarrow \lfloor (p₁ + r₁) / 2 \rfloor \)
7. \( q₂ \leftarrow \text{Binary-Search} (T[q₁], \ T, \ p₂, \ r₂) \)
8. \( q₃ \leftarrow p₃ + (q₁ - p₁) + (q₂ - p₂) \)
9. \( A[q₃] \leftarrow T[q₁] \)
10. \textbf{spawn} **Par-Merge** (T, p₁, p₁⁻¹, p₂, p₂⁻¹, A, p₃)
11. **Par-Merge** (T, q₁⁺¹, r₁, q₂⁺¹, r₂, A, q₃⁺¹)
12. \textbf{sync}
We have,

\[ n_2 \leq n_1 \Rightarrow 2n_2 \leq n_1 + n_2 = n \]

In the worst case, a recursive call in lines 9-10 merges half the elements of \( T[p_1..r_1] \) with all elements of \( T[p_2..r_2] \).

Hence, \#elements involved in such a call:

\[
\left\lfloor \frac{n_1}{2} \right\rfloor + n_2 \leq \frac{n_1}{2} + \frac{n_2}{2} + \frac{n_2}{2} = \frac{n_1 + n_2}{2} + \frac{2n_2}{4} \leq \frac{n}{2} + \frac{n}{4} = \frac{3n}{4}
\]
### Parallel Merge

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

1. \( n_1 \leftarrow r_1 - p_1 + 1, \quad n_2 \leftarrow r_2 - p_2 + 1 \)
2. *if* \( n_1 < n_2 \) *then*
3. \( p_1 \leftarrow p_2, \quad r_1 \leftarrow r_2, \quad n_1 \leftarrow n_2 \)
4. *if* \( n_1 = 0 \) *then return*
5. *else*
6. \( q_1 \leftarrow \left\lfloor \frac{p_1 + r_1}{2} \right\rfloor \)
7. \( q_2 \leftarrow \text{Binary-Search} \left( T[q_1], \ T, p_2, r_2 \right) \)
8. \( q_3 \leftarrow p_3 + (q_1 - p_1) + (q_2 - p_2) \)
9. \( A[q_3] \leftarrow T[q_1] \)
10. *spawn Par-Merge* \( T, p_1, q_1^{-1}, p_2, q_2^{-1}, A, p_3 \)
11. *Par-Merge* \( T, q_1+1, r_1, q_2+1, r_2, A, q_3+1 \)
12. *sync*

---

**Span:**

\[
T_\infty(n) = \begin{cases} 
\Theta(1), & \text{if } n = 1, \\
T_\infty \left( \frac{3n}{4} \right) + \Theta(\log n), & \text{otherwise.} 
\end{cases}
\]

\[= \Theta(\log^2 n) \quad [\text{MT Case 2}]\]

**Work:**

Clearly, \( T_1(n) = \Omega(n) \)

We show below that, \( T_1(n) = O(n) \)

For some \( \alpha \in \left[ \frac{1}{4}, \frac{3}{4} \right] \), we have the following recurrence,

\[
T_1(n) = T_1(\alpha n) + T_1((1 - \alpha)n) + O(\log n)
\]

Assuming \( T_1(n) \leq c_1 n - c_2 \log n \) for positive constants \( c_1 \) and \( c_2 \), and substituting on the right hand side of the above recurrence gives us: \( T_1(n) \leq c_1 n - c_2 \log n = O(n) \).

Hence, \( T_1(n) = \Theta(n) \).
Parallel Merge Sort with Parallel Merge

*Par-Merge-Sort*( A, p, r )   \{ sort the elements in \( A[p \ldots r] \) \}

1. if \( p < r \) then
2. \( q \leftarrow \lceil (p + r) / 2 \rceil \)
3. spawn \( \text{Merge-Sort}( A, p, q ) \)
4. \( \text{Merge-Sort}( A, q + 1, r ) \)
5. sync
6. \( \text{Par-Merge}( A, p, q, r ) \)

**Work:** \( T_1(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 2T_1 \left( \frac{n}{2} \right) + \Theta(n), & \text{otherwise.} \end{cases} \)

\[ = \Theta(n \log n) \quad [\text{MT Case 2}] \]

**Span:** \( T_\infty(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ T_\infty \left( \frac{n}{2} \right) + \Theta(\log^2 n), & \text{otherwise.} \end{cases} \)

\[ = \Theta(\log^3 n) \quad [\text{MT Case 2}] \]

**Parallelism:** \( \frac{T_1(n)}{T_\infty(n)} = \Theta \left( \frac{n}{\log^2 n} \right) \)