CSE 548: Analysis of Algorithms

Lectures 22 & 23
(Analyzing Parallel Algorithms)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2012
Why Parallelism?
Unicore Performance Has Hit a Wall!

Some Reasons

- Lack of additional ILP
  (Instruction Level Hidden Parallelism)
- High power density
- Manufacturing issues
- Physical limits
- Memory speed
Exhausted all ideas to exploit hidden parallelism?

- Multiple simultaneous instructions
- Dynamic instruction scheduling
- Branch prediction
- Out-of-order instructions
- Speculative execution
- Pipelining
- Non-blocking caches, etc.
Unicore Performance: High Power Density

- Dynamic power, $P_d \propto V^2 f C$
  - $V = \text{supply voltage}$
  - $f = \text{clock frequency}$
  - $C = \text{capacitance}$
- But $V \propto f$
- Thus $P_d \propto f^3$

Source: Patrick Gelsinger, Intel Developer Forum, Spring 2004 (Simon Floyd)
Unicore Performance: High Power Density

- Changing $f$ by 20% changes performance by 13%
- So what happens if we overclock by 20%?
- And underclock by 20%?

Source: Andrew A. Chien, Vice President of Research, Intel Corporation
Unicore Performance: High Power Density

- Changing $f$ by 20% changes performance by 13%
- So what happens if we overclock by 20%?
- And underclock by 20%?

Source: Andrew A. Chien, Vice President of Research, Intel Corporation
Unicore Performance: High Power Density

- Changing $f$ by 20% changes performance by 13%
- So what happens if we overclock by 20%?
- And underclock by 20%?

Source: Andrew A. Chien, Vice President of Research, Intel Corporation
Unicore Performance: Manufacturing Issues

- Frequency, $f \propto 1 / s$
  - $s = \text{feature size (transistor dimension)}$
- Transistors / unit area $\propto 1 / s^2$
- Typically, die size $\propto 1 / s$
- So, what happens if feature size goes down by a factor of $x$?
  - Raw computing power goes up by a factor of $x^4$!
  - Typically most programs run faster by a factor of $x^3$ without any change!

Source: Kathy Yelick and Jim Demmel, UC Berkeley
Unicore Performance: Manufacturing Issues

As feature size decreases

- Manufacturing cost goes up
  - Cost of a semiconductor fabrication plant doubles every 4 years (Rock’s Law)
- Yield (\% of usable chips produced) drops

Source: Kathy Yelick and Jim Demmel, UC Berkeley
Unicore Performance: Physical Limits

Execute the following loop on a serial machine in 1 second:

\[
for ( i = 0; i < 10^{12}; ++i )
\]
\[
z[ i ] = x[ i ] + y[ i ];
\]

- We will have to access \(3\times10^{12}\) data items in one second
- Speed of light is, \(c \approx 3\times10^8\) m/s
- So each data item must be within \(c / 3\times10^{12} \approx 0.1\) mm from the CPU on the average
- All data must be put inside a \(0.2\) mm \(\times\) \(0.2\) mm square
- Each data item ( \(\geq 8\) bytes ) can occupy only \(1\) Å\(^2\) space! ( size of a small atom! )

Source: Kathy Yelick and Jim Demmel, UC Berkeley
Unicore Performance: Memory Wall

Relative Performance


Source: Rick Hetherington, Chief Technology Officer, Microelectronics, Sun Microsystems
Moore’s Law Reinterpreted

Source: Report of the 2011 Workshop on Exascale Programming Challenges
Cores / Processor (General Purpose)

Source: Andrew A. Chien, Vice President of Research, Intel Corporation
No Free Lunch for Traditional Software

Source: Simon Floyd, Workstation Performance: Tomorrow's Possibilities (Viewpoint Column)
Insatiable Demand for Performance

Some Useful Classifications of Parallel Computers
Parallel Computer Memory Architecture (Shared Memory)

- All processors access all memory as global address space
- Changes in memory by one processor are visible to all others
- Two types:
  - Uniform Memory Access (UMA)
  - Non-Uniform Memory Access (NUMA)

Source: Blaise Barney, LLNL
Parallel Computer Memory Architecture (Distributed Memory)

- Each processor has its own local memory — no global address space
- Changes in local memory by one processor have no effect on memory of other processors
- Communication network to connect inter-processor memory

Source: Blaise Barney, LLNL
Parallel Computer Memory Architecture (Hybrid Distributed-Shared Memory)

- The share-memory component can be a cache-coherent SMP or a Graphics Processing Unit (GPU)
- The distributed-memory component is the networking of multiple SMP/GPU machines
- Most common architecture for the largest and fastest computers in the world today

Source: Blaise Barney, LLNL
Analyzing Parallel Algorithms
Let \( T_p = \) running time using \( p \) 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, $T = \Theta(n)$

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
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., $S_p \leq P$ \(\text{why?}\)

**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 $pT_p$
Let $T_s = \text{runtime of the optimal or the fastest known serial algorithm}$

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. The speedup, $S_p$, is given by:

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

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, $S_p = \frac{T}{T_p} \leq \frac{T_1}{T_p} = \frac{(1-f)T_p + pfT_p}{T_p} = 1 + (p - 1)f$
Greedy Scheduling Theorem
Nested Parallelism

```
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 );
}
```

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

Control cannot pass this point until all spawned children have returned.

```
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 );
}
```
### Loop Parallelism

**Serial Code**

```
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**

```
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;
    }
```

**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 );
}
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 );
}
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 );
}
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);
}
A parallel instruction stream is represented by a DAG $G = (V, E)$.

Each vertex $v \in V$ is a *strand* which is a sequence of instructions without a spawn, call, return or exception.

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

**Work:** $T_1 = 21$

**Span:** $T_\infty = 9$

**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.

- executed task
- ready to be executed
- not yet ready

A *greedy scheduler* tries to perform as much work as possible at every 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)
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 $p = \text{number of cores}$

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)
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 $p =$ 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)
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 )
Greedy Scheduling Theorem

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: $T_p^* \geq T_\infty$

\[ T_p \leq \frac{T_1}{p} + T_\infty \leq T_p^* + T_p^* = 2T_p^* \]
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, $P = \frac{T_1}{T_\infty} \gg p \Rightarrow \frac{T_1}{p} \gg T_\infty$

∴ From Graham-Brent Theorem:

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

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

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

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

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

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

Par-Iter-MM (Z, X, Y) { X, Y, Z are n × 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]·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

\[
\begin{align*}
Z & = Z_{11} & Z_{12} \\
    & Z_{21} & Z_{22} \\
\end{align*}
\]

\[
\begin{align*}
X & = X_{11} & X_{12} \\
    & X_{21} & X_{22} \\
\end{align*}
\]

\[
\begin{align*}
Y & = Y_{11} & Y_{12} \\
    & Y_{21} & Y_{22} \\
\end{align*}
\]

\[
\begin{align*}
Z & = X_{11} Y_{11} + X_{12} Y_{21} & X_{11} Y_{12} + X_{12} Y_{22} \\
    & X_{21} Y_{11} + X_{22} Y_{21} & X_{21} Y_{12} + X_{22} Y_{22} \\
\end{align*}
\]
Parallel Recursive MM

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

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

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

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

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

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

Additional Space:
\( s_\infty(n) = \Theta(1) \)
Recursive MM with More Parallelism

\[ Z \]

\[ n/2 \rightarrow \]

\[ \begin{array}{c c}
Z_{11} & Z_{12} \\
Z_{21} & Z_{22}
\end{array} \]

\[ n/2 \downarrow \]

\[ n \]

\[ \rightarrow \]

\[ n \]

\[ \uparrow \]

\[ n/2 \]

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

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

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

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

\[ \rightarrow \]

\[ n/2 \rightarrow \]

\[ \begin{array}{c c}
X_{11} Y_{11} & X_{11} Y_{12} \\
X_{21} Y_{11} & X_{21} Y_{12}
\end{array} \]

\[ + \]

\[ \begin{array}{c c}
X_{12} Y_{21} & X_{12} Y_{22} \\
X_{22} Y_{21} & X_{22} Y_{22}
\end{array} \]
Recursive MM with More Parallelism

Par-Rec-MM2 (Z, X, Y)  \{ X, Y, Z are n \times n matrices, 
where n = 2^k 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. parallel for j \leftarrow 1 to n do
15. Z[i][j] \leftarrow Z[i][j] + T[i][j]
16. endif
Recursive MM with More Parallelism

\[
\begin{align*}
\text{Par-Rec-MM2}(Z, X, Y) & \quad \{X, Y, Z \text{ are } n \times n \text{ matrices, where } n = 2^k \text{ for integer } k \geq 0\} \\
1. \text{ if } n = 1 \text{ then} & \\
2. & Z \leftarrow Z + X \cdot Y \\
3. \text{ else} \quad \{T \text{ is a temporary } n \times n \text{ matrix}\} & \\
4. & \text{spawn Par-Rec-MM2}(Z_{11}, X_{11}, Y_{11}) \\
5. & \text{spawn Par-Rec-MM2}(Z_{12}, X_{11}, Y_{12}) \\
6. & \text{spawn Par-Rec-MM2}(Z_{21}, X_{21}, Y_{11}) \\
7. & \text{spawn Par-Rec-MM2}(Z_{21}, X_{21}, Y_{12}) \\
8. & \text{spawn Par-Rec-MM2}(T_{11}, X_{12}, Y_{21}) \\
9. & \text{spawn Par-Rec-MM2}(T_{12}, X_{12}, Y_{22}) \\
10. & \text{spawn Par-Rec-MM2}(T_{21}, X_{22}, Y_{21}) \\
11. & \text{spawn Par-Rec-MM2}(T_{22}, X_{22}, Y_{22}) \\
12. & \text{sync} \\
13. & \text{parallel for } i \leftarrow 1 \text{ to } n \text{ do} \\
14. & \quad \text{parallel for } j \leftarrow 1 \text{ to } n \text{ do} \\
15. & \quad Z[i][j] \leftarrow Z[i][j] + T[i][j] \\
16. & \text{endif}
\end{align*}
\]

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

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

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

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

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

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

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

**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

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

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:

\[ A[p_3..r_3] \]

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

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:

\[ A[p_3..r_3] \quad \text{with} \quad n_3 = r_3 - p_3 + 1 = n_1 + n_2 \]

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

\[ A[p_3..r_3] \quad \text{where} \quad n_3 = r_3 - p_3 + 1 = n_1 + n_2 \]

**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

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

Subarrays to merge:

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

Suppose: \( n_1 \geq n_2 \)

Merged output:

\[ A[p_3 \ldots r_3] \]

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

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

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

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

subarrays to merge: \[ T[p_1 \ldots r_1] \quad \text{and} \quad T[p_2 \ldots r_2] \]

\[
\begin{array}{ccc}
  & \leq x & \geq x \\
p_1 & \ldots & x & \ldots \\
q_1 & \geq x & \ldots \\
r_1 & \ldots & \leq x & \ldots
\end{array}
\]

suppose: \( n_1 \geq n_2 \)

merged output: \[ A[p_3 \ldots r_3] \]

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

Perform the following two steps in parallel.

**Step 4(a):** Recursively merge \( T[p_1 \ldots q_1 - 1] \) 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 \( A[q_3 + 1 \ldots r_3] \)
Parallel Merge

\[ \text{Par-Merge} \left( T, p_1, r_1, p_2, r_2, A, p_3 \right) \]

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. \( p_1 \leftrightarrow p_2, \quad r_1 \leftrightarrow r_2, \quad n_1 \leftrightarrow n_2 \)
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 \( \text{Par-Merge} \left( T, p_1, q_1-1, p_2, q_2-1, A, p_3 \right) \)
11. \( \text{Par-Merge} \left( T, q_1+1, r_1, q_2+1, r_2, A, q_3+1 \right) \)
12. 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:

\[
\left\lceil \frac{n_1}{2} \right\rceil + n_2 \leq \frac{n_1}{2} + \frac{n_2}{2} + \frac{n_2}{2} = \frac{n_1 + n_2}{2} + \frac{2n_2}{4} \leq \frac{n}{2} + \frac{n}{4} = \frac{3n}{4}
\]
Parallel Merge

Par-Merge (T, p₁, r₁, p₂, r₂, A, p₃)
1. \( n_1 \leftarrow r_1 - p_1 + 1 \), \( n_2 \leftarrow r_2 - p_2 + 1 \)
2. if \( n_1 < n_2 \) then
3. \( p_1 \leftrightarrow p_2, \ r_1 \leftrightarrow r_2, \ n_1 \leftrightarrow n_2 \)
4. if \( n_1 = 0 \) then return
5. else
6. \( q_1 \leftarrow \lfloor (p_1 + r_1) / 2 \rfloor \)
7. \( q_2 \leftarrow \text{Binary-Search} (T[q_1], T, p_2, r_2) \)
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, p₁, q₁⁻¹, p₂, q₂⁻¹, A, p₃)
11. Par-Merge (T, q₁⁺¹, r₁, q₂⁺¹, r₂, A, q₃⁺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,

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

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