CSE 548 / AMS 542: Analysis of Algorithms

Lecture 12
(Analyzing Parallel Algorithms)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2023
Why Parallelism?
Moore’s Law

The number of transistors on an integrated circuit for minimum component cost doubles every 24 months. (Gordon Moore, 1965)
Unicore Performance

Single-Threaded Floating-Point Performance

Based on adjusted SPECfp® results

Some Reasons

- Lack of additional ILP
  ( Instruction Level Hidden Parallelism )
- High power density
- Manufacturing issues
- Physical limits
- Memory speed
Unicore Performance: No Additional ILP

“Everything that can be invented has been invented.”

— Charles H. Duell

Commissioner, U.S. patent office, 1899

Exhausted all ideas to exploit hidden parallelism?

— Multiple simultaneous instructions

— Instruction Pipelining

— Out-of-order instructions

— Speculative execution

— Branch prediction

— Register renaming, etc.
ILP: Instruction Pipelining

**Unicore Performance: High Power Density**

- Dynamic power, \( P_d \propto V^2 f C \)
  
  - \( V \) = supply voltage
  - \( f \) = clock frequency
  - \( C \) = capacitance

- But \( V \propto f \)
- Thus \( P_d \propto f^3 \)

*Source: Patrick Gelsinger, Intel Developer Forum, Spring 2004 (Simon Floyd)*
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

- Manufacturing cost goes up as feature size decreases
  - Cost of a semiconductor fabrication plant doubles every 4 years (Rock’s Law)
- CMOS feature size is limited to 5 nm (at least 10 atoms)

Source: Kathy Yelick and Jim Demmel, UC Berkeley
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 $3 \times 10^{12}$ data items in one second
- Speed of light is, $c \approx 3 \times 10^8$ m/s
- So each data item must be within $c / 3 \times 10^{12} \approx 0.1$ mm from the CPU on the average
- All data must be put inside a $0.2 \times 0.2$ 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


Source: Rick Hetherington, Chief Technology Officer, Microelectronics, Sun Microsystems
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

“Oh Sinnerman, where you gonna run to?”

— Sinnerman (recorded by Nina Simone)
Where You Gonna Run To?

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

Source: Andrew A. Chien, Vice President of Research, Intel Corporation
Where You Gonna Run To?

– 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
Where You Gonna Run To?

– 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
Moore’s Law Reinterpreted

Source: Report of the 2011 Workshop on Exascale Programming Challenges
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 (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
- Programming
  - Message Passing Interface (MPI)
  - Many once available: PVM, Chameleon, MPL, NX, etc.

Source: Blaise Barney, LLNL
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)
- Programming
  - Open Multi-Processing (OpenMP)
  - Cilk/Cilk++ and Intel Cilk Plus
  - Intel Thread Building Block (TBB), etc.

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.

- Programming
  - OpenMP / Cilk + CUDA / OpenCL + MPI, etc.

Source: Blaise Barney, LLNL
Types of Parallelism
Nested Parallelism

```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 );
}
```

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

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

### 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 );
}
```
Loop Parallelism

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 )
    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.
Recursive D&C Implementation of Parallel Loops

\texttt{parallel for ( int i = s; i < t; ++i )}
\texttt{BODY( i );}

daive-and-conquer
implementation

\texttt{void recur( int lo, int hi )}
{\texttt{
  if ( hi - lo > GRAINSIZE )}
  {\texttt{
      int mid = lo + ( hi - lo ) / 2;}
      \texttt{spawn recur( lo, mid );}
      \texttt{recur( mid, hi );}
      \texttt{sync;}
  }\texttt{
  else}
  {\texttt{
      for ( int i = lo; i < hi; ++i )}
      \texttt{BODY( i );}
  }\texttt{
}
\texttt{recur( s, t );}
Analyzing Parallel Algorithms
Speedup

Let $T_p = \text{running time using } p \text{ identical processing elements}$

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

Theoretically, $S_p \leq p$

*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, $S_n = \frac{T_1}{T_n} = \Theta\left(\frac{n}{\log n}\right)$
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
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., $S_p \leq P$

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 $pT_p$

\[
T_p \geq \frac{T_1}{p}
\]
Bounding Parallel Running Time \((T_p)\)

A runtime/online scheduler maps tasks to processing elements dynamically at runtime.

A greedy scheduler never leaves a processing element idle if it can map a task to it.

**Theorem [Graham’68, Brent’74]:** For any greedy scheduler,

\[
T_p \leq \frac{T_1}{p} + T_\infty
\]

**Corollary:** For any greedy scheduler,

\[
T_p \leq 2T_p^* ,
\]

where \(T_p^*\) is the running time due to optimal scheduling on \(p\) processing elements.
**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.
Suppose we use $p$ processing elements. First each processing element locally adds its $n/p$ 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 $n = \Omega(p \log p)$. 

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.

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition
Scaling Law
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, $S_p = \frac{T_1}{T_p} \leq \frac{1}{(1-f) + \frac{f}{p}} \leq \frac{1}{1-f}$

**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.
Races
A determinacy race occurs if two logically parallel instructions access the same memory location and at least one of them performs a write.

```c
int x = 0;
parallel for ( int i = 0; i < 2; i++ )
    x++;
printf("%d", x);
```

A determinacy race occurs if two logically parallel instructions access the same memory location and at least one of them performs a write.

```
int x = 0;
parallel for ( int i = 0; i < 2; i++ )
    x++;
printf("%d", x);
```
Critical Sections and Mutexes

```
int r = 0;
parallel for ( int i = 0; i < n; i++ )
    r += eval( x[ i ] );
```

```
mutex mtx;
parallel for ( int i = 0; i < n; i++ )
    mtx.lock( );
    r += eval( x[ i ] );
    mtx.unlock( );
```

**Race**

Critical section

Two or more strands must not access at the same time

**Mutex (Mutual Exclusion)**

An attempt by a strand to lock an already locked mutex causes that strand to block (i.e., wait) until the mutex is unlocked.

Problems

- Lock overhead
- Lock contention
Critical Sections and Mutexes

```c
int r = 0;
parallel for ( int i = 0; i < n; i++ )
    r += eval( x[ i ] );
```

mutex mtx;

```c
parallel for ( int i = 0; i < n; i++ )
    mtx.lock( );
    r += eval( x[ i ] );
    mtx.unlock( );
```

mutex mtx;

```c
parallel for ( int i = 0; i < n; i++ )
    int y = eval( x[ i ] );
    mtx.lock( );
    r += y;
    mtx.unlock( );
```

- slightly better solution
- but lock contention can still destroy parallelism
Recursive D&C Implementation of Loops
Recursive D&C Implementation of Parallel Loops

\[
\text{parallel for ( int } i = s; i < t; ++i ) \]
\[
\text{BODY( } i \text{ );}
\]

**divide-and-conquer implementation**

```c
void recur( int lo, int hi )
{
    if ( hi - lo > GRAINSIZE )
    {
        int mid = lo + ( hi - lo ) / 2;
        spawn recur( lo, mid );
        recur( mid, hi );
        sync;
    }
    else
    {
        for ( int i = lo; i < hi; ++i )
            BODY( i );
    }
}
recur( s, t );
```

Let \( n = t - s \)

\( m = \) running time of a single call to \( \text{BODY} \)

**Span:** \( T_\infty(n) = \Theta(\log n + \text{GRAINSIZE} \times m) \)
Parallel Iterative MM

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

1. for i ← 1 to n do
2.    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]

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]
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

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

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

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

1. \( \text{if } n = 1 \text{ then} \)
2. \( Z \leftarrow Z + X \cdot Y \)
3. \( \text{else} \)
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. \( \text{endif} \)
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:

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

= \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

\[ Z = \begin{array}{cc}
Z_{11} & Z_{12} \\
Z_{21} & Z_{22}
\end{array} \]

\[ = \begin{array}{cc}
X_{11} Y_{11} + X_{12} Y_{21} & X_{11} Y_{12} + X_{12} Y_{22} \\
X_{21} Y_{11} + X_{22} Y_{21} & X_{21} Y_{12} + X_{22} Y_{22}
\end{array} \]

\[ = \begin{array}{cc}
X_{11} Y_{11} & X_{11} Y_{12} \\
X_{21} Y_{11} & X_{21} Y_{12}
\end{array} +
\begin{array}{cc}
X_{12} Y_{21} & X_{12} Y_{22} \\
X_{22} Y_{21} & X_{22} Y_{22}
\end{array} \]
Recursive MM with More Parallelism

Par-Rec-MM2 (Z, X, Y)  { X, Y, Z are n × n matrices, where n = 2^k for integer k ≥ 0 }

1. if n = 1 then
2.  Z ← Z + X · Y
3. else { T is a temporary n × 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 ← 1 to n do
14.  parallel for j ← 1 to n do
15.  Z[i][j] ← Z[i][j] + T[i][j]
16. endif
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

**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**

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

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. \( \text{Merge} \ (A, \ p, \ q, \ r) \)

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

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

= \( \Theta(n \log n) \) [ MT Case 2 ]

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

= \( \Theta(n) \) [ MT Case 3 ]

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

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

subarrays to merge:

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

suppose: \( n_1 \geq n_2 \)

merged output:

\[ A[p_3..r_3] \]

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

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

subarrays to merge:

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

suppose: \( n_1 \geq n_2 \)

merged output:

\[ A[p_3..r_3] \]

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

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

subarrays to merge:

$$n_1 = r_1 - p_1 + 1$$
$$n_2 = r_2 - p_2 + 1$$

merged output:

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

Suppose: $$n_1 \geq n_2$$

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

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

subarrays to merge:

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

suppose: \( n_1 \geq n_2 \)

merged output:

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

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

Subarrays to merge:

\[ n_1 = r_1 - p_1 + 1 \]

\[ T[p_1..r_1] \]

\[ n_2 = r_2 - p_2 + 1 \]

\[ T[p_2..r_2] \]

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

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

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

subarrays to merge:

converted output:

\[ n_1 = o_1 - p_1 + 1 \]
\[ n_2 = o_2 - p_2 + 1 \]
\[ n_3 = o_3 - p_3 + 1 = n_1 + n_2 \]

suppose: \( n_1 \geq n_2 \)

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 \( A[p_3..q_3 - 1] \)

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

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

1. \( n₁ \leftarrow r₁ - p₁ + 1, \quad n₂ \leftarrow r₂ - p₂ + 1 \)
2. if \( n₁ < n₂ \) then
3. \( p₁ \leftrightarrow p₂, \quad r₁ \leftrightarrow r₂, \quad n₁ \leftrightarrow n₂ \)
4. if \( n₁ = 0 \) then return
5. 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. 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
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. \(\text{if } n_1 < n_2 \text{ then}\)
3. \(p_1 \leftrightarrow p_2, \quad r_1 \leftrightarrow r_2, \quad n_1 \leftrightarrow n_2\)
4. \(\text{if } n_1 = 0 \text{ then return}\)
5. \(\text{else}\)
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. \(\text{spawn Par-Merge} (T, p_1, q_1-1, p_2, q_2-1, A, p_3)\)
11. \(\text{Par-Merge} (T, q_1+1, r_1, q_2+1, r_2, A, q_3+1)\)
12. \(\text{sync}\)

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. \text{if } n_1 < n_2 \text{ then}
3. \(p_1 \leftrightarrow p_2, \quad r_1 \leftrightarrow r_2, \quad n_1 \leftrightarrow n_2\)
4. \text{if } n_1 = 0 \text{ then return}
5. \text{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. \text{spawn Par-Merge}(T, p_1, q_1^{-1}, p_2, q_2^{-1}, A, p_3)\)
11. \text{Par-Merge}(T, q_1+1, r_1, q_2+1, r_2, A, q_3+1)\)
12. \text{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

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

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{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:**

\[
\frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{n}{\log^2 n}\right)
\]
Parallel Prefix Sums
Parallel Prefix Sums

Input: A sequence of $n$ elements $\{x_1, x_2, ..., x_n\}$ drawn from a set $S$ with a binary associative operation, denoted by $\oplus$.

Output: A sequence of $n$ partial sums $\{s_1, s_2, ..., s_n\}$, where

$$s_i = x_1 \oplus x_2 \oplus ... \oplus x_i \text{ for } 1 \leq i \leq n.$$
Parallel Prefix Sums

\[ \text{Prefix-Sum} \left( \langle x_1, x_2, \ldots, x_n \rangle, \oplus \right) \{ n = 2^k \text{ for some } k \geq 0. \]

Return prefix sums \( \langle s_1, s_2, \ldots, s_n \rangle \}

1. if \( n = 1 \) then
2. \( s_1 \leftarrow x_1 \)
3. else
4. parallel for \( i \leftarrow 1 \) to \( n/2 \) do
5. \( y_i \leftarrow x_{2i-1} \oplus x_{2i} \)
6. \( \langle z_1, z_2, \ldots, z_{n/2} \rangle \leftarrow \text{Prefix-Sum} \left( \langle y_1, y_2, \ldots, y_{n/2} \rangle, \oplus \right) \)
7. parallel for \( i \leftarrow 1 \) to \( n \) do
8. if \( i = 1 \) then \( s_1 \leftarrow x_1 \)
9. else if \( i = \text{even} \) then \( s_i \leftarrow z_{i/2} \)
10. else \( s_i \leftarrow z_{(i-1)/2} \oplus x_i \)
11. return \( \langle s_1, s_2, \ldots, s_n \rangle \)
Parallel Prefix Sums
Parallel Prefix Sums
Parallel Prefix Sums

Prefix-Sum (\( \langle x_1, x_2, \ldots, x_n \rangle, \oplus \) ) \( \{ n = 2^k \) for some \( k \geq 0 \).
Return prefix sums \( \langle s_1, s_2, \ldots, s_n \rangle \} \)

1. \( \text{if } n = 1 \text{ then} \)
2. \( s_1 \leftarrow x_1 \)
3. \( \text{else} \)
4. \( \text{parallel for } i \leftarrow 1 \text{ to } n/2 \text{ do} \)
5. \( y_i \leftarrow x_{2i-1} \oplus x_{2i} \)
6. \( \langle z_1, z_2, \ldots, z_{n/2} \rangle \leftarrow \text{Prefix-Sum}(\langle y_1, y_2, \ldots, y_{n/2} \rangle, \oplus) \)
7. \( \text{parallel for } i \leftarrow 1 \text{ to } n \text{ do} \)
8. \( \text{if } i = 1 \text{ then } s_1 \leftarrow x_1 \)
9. \( \text{else if } i = \text{even} \text{ then } s_i \leftarrow z_{i/2} \)
10. \( \text{else } s_i \leftarrow z_{(i-1)/2} \oplus x_i \)
11. \( \text{return } \langle s_1, s_2, \ldots, s_n \rangle \)

Observe that we have assumed here that a parallel for loop can be executed in \( \Theta(1) \) time. But recall that cilk_for is implemented using divide-and-conquer, and so in practice, it will take \( \Theta(\log n) \) time. In that case, we will have \( T_\infty(n) = \Theta(\log^2 n) \), and parallelism = \( \Theta(n/ \log^2 n) \).
Parallel Partition
Parallel Partition


**Output:** Rearrange the elements of $A[ q : r ]$, and return an index $k \in [ q, r ]$, such that all elements in $A[ q : k - 1 ]$ are smaller than $x$, all elements in $A[ k + 1 : r ]$ are larger than $x$, and $A[ k ] = x$.

```plaintext
Par-Partition ( A[ q : r ], x )
1.  n ← r − q + 1
2.  if n = 1 then return q
3.  array B[ 0: n − 1 ], lt[ 0: n − 1 ], gt[ 0: n − 1 ]
4.  parallel for i ← 0 to n − 1 do
6.    if B[ i ] < x then lt[ i ] ← 1 else lt[ i ] ← 0
7.    if B[ i ] > x then gt[ i ] ← 1 else gt[ i ] ← 0
8.    lt[ 0: n − 1 ] ← Par-Prefix-Sum ( lt[ 0: n − 1 ], + )
9.    gt[ 0: n − 1 ] ← Par-Prefix-Sum ( gt[ 0: n − 1 ], + )
10.   k ← q + lt[ n − 1 ], A[ k ] ← x
11.   parallel for i ← 0 to n − 1 do
13.    else if B[ i ] > x then A[ k + gt[ i ] ] ← B[ i ]
14.   return k
```
Parallel Partition

\[ A: \begin{bmatrix} 9 & 5 & 7 & 11 & 1 & 3 & 8 & 14 & 4 & 21 \end{bmatrix} \quad x = 8 \]
Parallel Partition

A: 9 5 7 11 1 3 8 14 4 21
B: 9 5 7 11 1 3 8 14 4 21

x = 8

lt: 0 1 1 0 1 1 0 0 1 0
gt: 1 0 0 1 0 0 0 1 0 1
Parallel Partition

\[ x = 8 \]

\[
\begin{array}{cccccccccccc}
9 & 5 & 7 & 11 & 1 & 3 & 8 & 14 & 4 & 21 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
9 & 5 & 7 & 11 & 1 & 3 & 8 & 14 & 4 & 21 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 2 & 3 & 4 & 4 & 4 & 5 & 5 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 4 \\
\end{array}
\]

prefix sum
Parallel Partition

\[ A: \begin{array}{cccccccccccc}
9 & 5 & 7 & 11 & 1 & 3 & 8 & 14 & 4 & 21 \\
\end{array} \]

\[ B: \begin{array}{cccccccccccc}
9 & 5 & 7 & 11 & 1 & 3 & 8 & 14 & 4 & 21 \\
\end{array} \]

\[ x = 8 \]

\[ \text{lt:} \begin{array}{cccccccccccc}
0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
\end{array} \]

\[ \text{gt:} \begin{array}{cccccccccccc}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
\end{array} \]

\[ \text{prefix sum} \]

\[ \text{lt:} \begin{array}{cccccccccccc}
0 & 1 & 2 & 2 & 3 & 4 & 4 & 4 & 5 & 5 \\
\end{array} \]

\[ \text{gt:} \begin{array}{cccccccccccc}
1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 4 \\
\end{array} \]

\[ \text{prefix sum} \]

\[ A: \begin{array}{cccccccccccc}
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\hline
\end{array} \]
Parallel Partition

A: 9 5 7 11 1 3 8 14 4 21  
B: 9 5 7 11 1 3 8 14 4 21  

t: 0 1 1 0 1 1 0 0 1 0  
gt: 1 0 0 1 0 0 0 1 0 1  

prefix sum

lt: 0 1 2 2 3 4 4 4 5 5  
gt: 1 1 1 2 2 2 2 3 3 4  

prefix sum

k = 5

A: 5 7 1 3 4 8  

x = 8
Parallel Partition

A: $\begin{bmatrix} 9 & 5 & 7 & 11 & 1 & 3 & 8 & 14 & 4 & 21 \end{bmatrix}$ \quad x = 8

B: $\begin{bmatrix} 9 & 5 & 7 & 11 & 1 & 3 & 8 & 14 & 4 & 21 \end{bmatrix}$

lt: \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}

gt: \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}

lt: \begin{bmatrix} 0 & 1 & 2 & 2 & 3 & 4 & 4 & 4 & 5 & 5 \end{bmatrix}

prefix sum

k = 5

gt: \begin{bmatrix} 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 4 \end{bmatrix}

prefix sum

A: $\begin{bmatrix} 5 & 7 & 1 & 3 & 4 & 8 & 9 & 11 & 14 & 21 \end{bmatrix}$
Parallel Partition

A: 9 5 7 11 1 3 8 14 4 21  \( x = 8 \)

B: 9 5 7 11 1 3 8 14 4 21

\( k = 5 \)

prefix sum

\( \lt: \) 0 1 1 0 1 1 0 0 1 0

\( gt: \) 1 0 0 1 0 0 0 1 0 1

prefix sum
Parallel Partition: Analysis

\[ T_1(n) = \Theta(n) \quad [\text{lines 1–7 }] \]
\[ + \Theta(n) \quad [\text{lines 8–9 }] \]
\[ + \Theta(n) \quad [\text{lines 10–14 }] \]
\[ = \Theta(n) \]

Work:

Span:

Assuming \( \log n \) depth for parallel for loops:

\[ T_\infty(n) = \Theta(\log n) \quad [\text{lines 1–7 }] \]
\[ + \Theta(\log^2 n) \quad [\text{lines 8–9 }] \]
\[ + \Theta(\log n) \quad [\text{lines 10–14 }] \]
\[ = \Theta(\log^2 n) \]

Parallelism:

\[ \frac{T_1(n)}{T_\infty(n)} = \Theta\left( \frac{n}{\log^2 n} \right) \]
Parallel Quicksort
Randomized Parallel QuickSort

**Input:** An array $A[q : r]$ of distinct elements.

**Output:** Elements of $A[q : r]$ sorted in increasing order of value.

```
Par-Randomized-QuickSort ( A[q : r] )
1.  $n \leftarrow r - q + 1$
2.  if $n \leq 30$ then
4.  else
5.      select a random element $x$ from $A[q : r]$
6.      $k \leftarrow Par-Partition ( A[q : r], x )$
7.      spawn Par-Randomized-QuickSort ( A[q : k - 1] )
8.      Par-Randomized-QuickSort ( A[k + 1 : r] )
9.      sync
```
Randomized Parallel QuickSort: Analysis

Par-Randomized-QuickSort \( (A[ q : r ] ) \)
1. \( n \leftarrow r - q + 1 \)
2. \( \text{if } n \leq 30 \text{ then} \)
3. sort \( A[ q : r ] \) using any sorting algorithm
4. \( \text{else} \)
5. select a random element \( x \) from \( A[ q : r ] \)
6. \( k \leftarrow \text{Par-Partition} \ (A[ q : r ], \ x) \)
7. \( \text{spawn Par-Randomized-QuickSort} \ (A[ q : k - 1 ] ) \)
8. \( \text{Par-Randomized-QuickSort} \ (A[ k + 1 : r ] ) \)
9. \( \text{sync} \)

Lines 1—6 take \( \Theta(\log^2 n) \) parallel time and perform \( \Theta(n) \) work.

Also the recursive spawns in lines 7—8 work on disjoint parts of \( A[ q : r ] \). So the upper bounds on the parallel time and the total work in each level of recursion are \( \Theta(\log^2 n) \) and \( \Theta(n) \), respectively.

Hence, if \( D \) is the recursion depth of the algorithm, then

\[
T_1(n) = O(nD) \quad \text{and} \quad T_\infty(n) = O(D \log^2 n)
\]
We already proved that w.h.p. recursion depth, $D = O(\log n)$. Hence, with high probability,

$$T_1(n) = O(n \log n) \text{ and } T_\infty(n) = O(\log^3 n)$$