

# Multithreaded Algorithms

Ashwin Kolhatkar and Harshit Gautam

# Introduction

Nowadays, many computers have multiple “cores” in CPU, each of which is a full-fledged processor that can access common memory.

Thus multiple threads can be used to efficiently utilize these multiple cores. Thus, the role of multithreaded algorithms comes into play.

In this research assignment, we shall explain and analyse some common algorithms in a multithreaded environment.

One common means of programming chip multiprocessors is by using *static threading*, which provides a software abstraction of virtual processors or *threads*, sharing a common memory.

# **Part I**

## Basics of Dynamic Multithreading

# Dynamic Multithreading

It allows programmers to specify parallelism in applications without worrying about communication protocols or load balancing.

It contains a ***scheduler***, which load balances the computation automatically, hence simplifying the programmer's chore.

A parallel loop is like an ordinary ***for loop***, except that iterations can occur concurrently.

Multithreading platforms: **Cilk , OpenMP, Task Parallel Library, Threading Building Blocks [Intel TBB]**

## Some notation...

We can describe a multithreaded algorithm by adding to our pseudocode just three concurrency keywords:

**parallel**

**spawn**

**sync**

These keywords express the *logical parallelism* of the computation, indicating which parts of the computation may proceed in parallel.

## Example: Multithreading the computation of Fibonacci numbers

$$F_0 = 0,$$

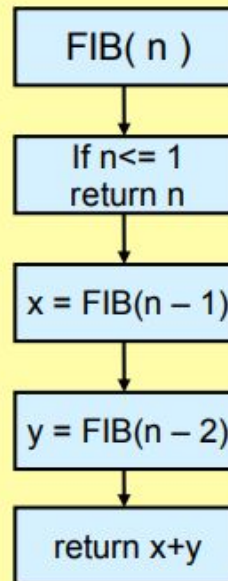
$$F_1 = 1,$$

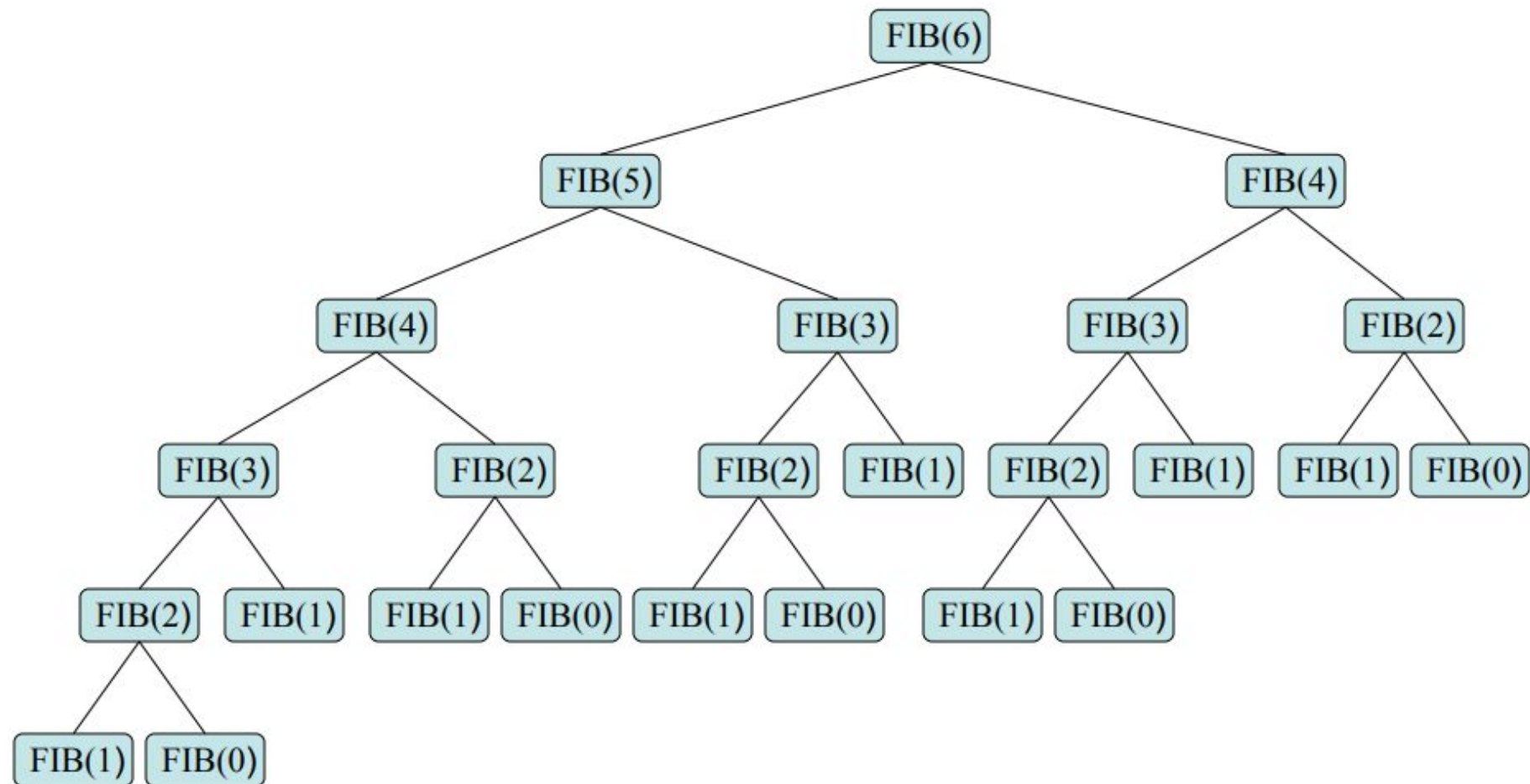
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i \geq 2.$$

### Sequential version

**FIB( n )**

1. **if**  $n \leq 1$
2.   **return**  $n$
3. **else**
4.    $x = \text{FIB}(n - 1)$
5.    $y = \text{FIB}(n - 2)$
6.   **return**  $x + y$





The tree of recursive calls to compute FIB(6).

**Running time analysis of  $FIB(n)$ :**

$$T(n) = T(n - 1) + T(n - 2) + \Theta(1)$$

This recurrence has solution  $T(n) = \Theta(F_n)$

$$T(n) = \Theta(\phi^n) \quad \text{where } \phi = (1 + \sqrt{5})/2 \text{ is the golden ratio}$$



Poor complexity of algorithm, but illustrates the key concepts for analyzing multithreaded algorithms.

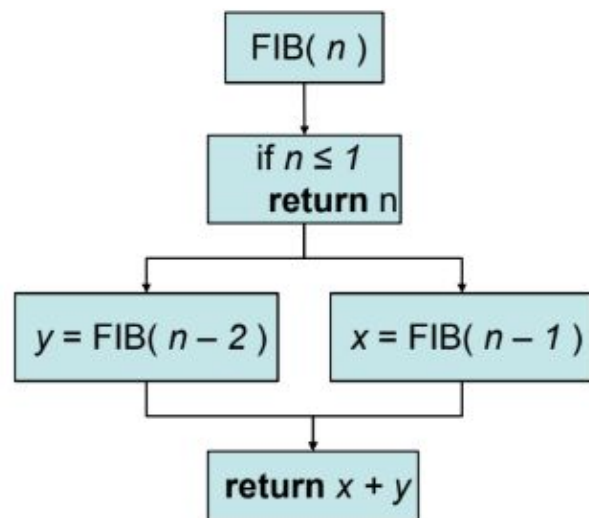
**Since  $T(n)$  grows exponentially in  $n$ , this procedure is a slow way to compute Fibonacci numbers.**

Now we shall see how dynamic multithreading can reduce complexity, and under what situations.

## Multithreading the computation of Fibonacci numbers

Recursive calls to FIB( $n-1$ ) and FIB( $n-2$ ) in lines 4 and 5 are independent to each other.

```
FIB(  $n$  )  
1  if  $n \leq 1$   
2      return  $n$   
3  else  
4       $x = \text{FIB}( n - 1 )$   
5       $y = \text{FIB}( n - 2 )$   
6      return  $x + y$ 
```



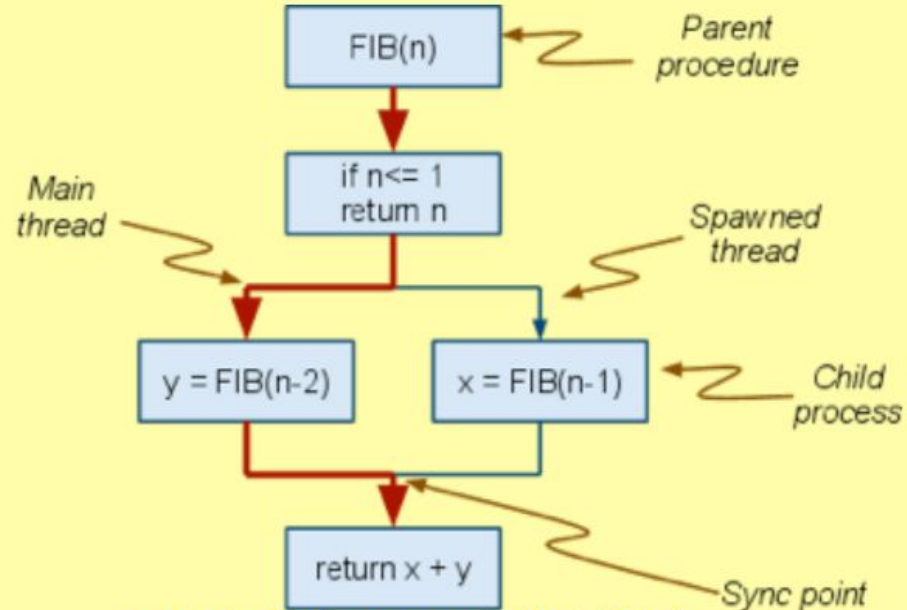
Task dependency graph for FIB( $n$ )

We augment our pseudocode to indicate parallelism by using the concurrency keywords **spawn** and **sync**.

### Parallel version

**P-FIB( n )**

1. **if**  $n \leq 1$
2.   **return**  $n$
3. **else**
4.    $x = \text{spawn P-FIB}( n - 1 )$
5.    $y = \text{P-FIB}( n - 2 )$
6.   **sync**
7.   **return**  $x + y$



Task dependency graph for  $\text{FIB}( n )$

# Multithreaded Execution Model

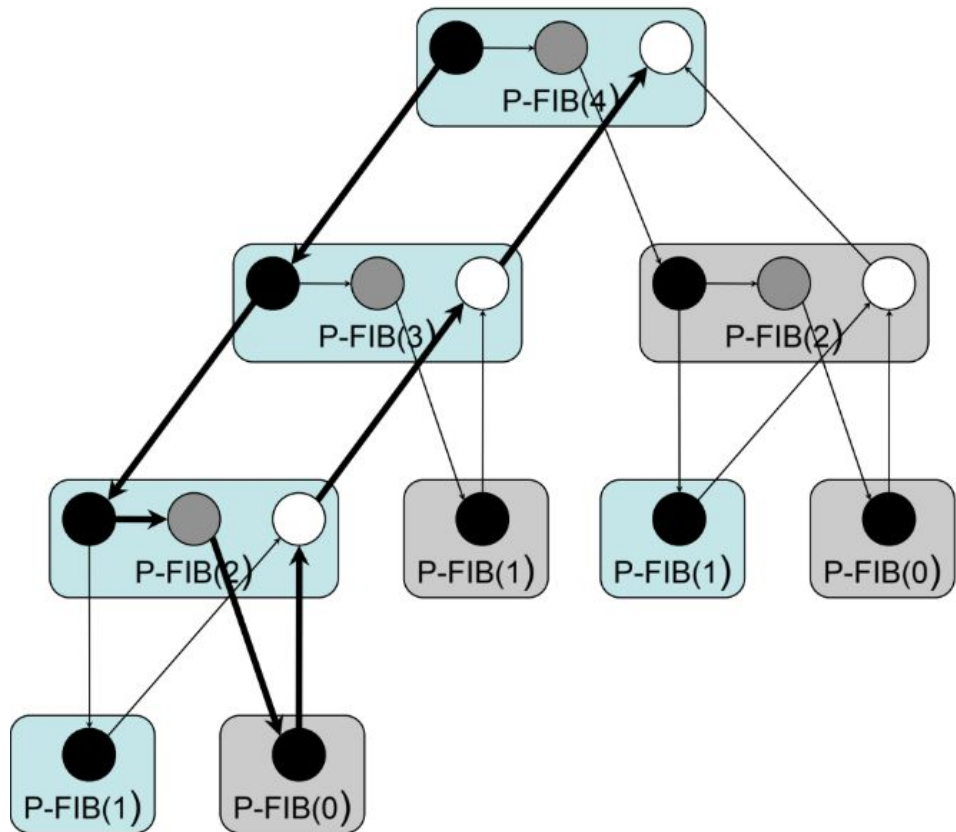
A directed acyclic graph  $G = (V, E)$  called a **computation dag**, is used to represent multithreaded computation.

Vertices in  $V$  represent instructions.






Edges in  $E$  represent dependencies between instructions.

Edge  $(u, v)$  in  $E$  means instruction  $u$  must execute before instruction  $v$ .

# A model of multithreaded execution



Directed acyclic graph representing P-FIB(4)

- Circles represent strands.
-  base case or the part of the procedure up to the spawn of P-FIB( $n-1$ ) in line 4.
-  part of the procedure that calls P\_FIB( $n-2$ ) in line 5 up to the **sync** in line 6, where it suspends until the spawn of P-FIB( $n-1$ ) returns.
-  part of the procedure after the **sync** where it sums  $x$  and  $y$  up to the point where it returns the result.
- Rectangles contain strands belonging to the same procedure
-  for spawned procedures
-  for called procedures.

# Performance Measures

**Work:** Total time to execute the entire computation on a single processor.

$Work = \sum t_i$ , where  $t_i$  is the time taken by strand  $i$ .

It can also be defined as vertices in the dag if each strand takes unit time.

**Span:** Longest time to execute the strands along any path in the dag.

It can also be defined as vertices on a longest or **critical path** in the dag if each strand takes unit time.

**Speedup:** Tells how many times faster computation is on  $P$  processors rather than 1 processor.

The running time of a multithreaded computation:

$T_p$  = depends on work, span,  $p$  and scheduler, where  $p$  is the no. of processors available.

$T_1$  = **Work** on a single processor.

$T_\infty$  = **Span** is the running time if we could run each strand on its own processor

(i.e., *unlimited number of processors*)

### Work Law:

$$T_P \geq T_1/P$$

When the speedup is linear in the number of processors i.e. when the computation exhibits  $T_1/T_\infty = \Theta(P)$  it is called **linear speedup**.

When  $T_1/T_\infty = P$ , we have **perfect linear speedup**.

### Span Law:

$$T_P \geq T_\infty$$

The ratio  $T_1/T_\infty$  of the work to the span gives the **parallelism** of the multithreaded computation.

For an ideal parallel computer with  $P$  processors, a greedy scheduler executes a multithreaded computation within time:

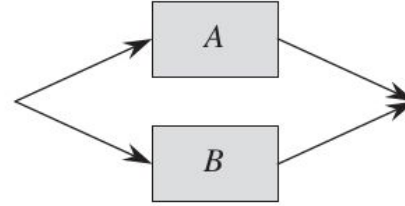
$$T_P \leq T_1/P + T_\infty$$





Work:  $T_1(A \cup B) = T_1(A) + T_1(B)$   
Span:  $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

(a)



Work:  $T_1(A \cup B) = T_1(A) + T_1(B)$   
Span:  $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

(b)

If two sub-computations are joined in series , their spans add to form the spans of their compositions.

If they are joined in parallel, span of their composition is the maximum of the two sub-computations.

# Parallelism Interpretations

As a ratio, the parallelism denotes the average amount of work that can be performed in parallel for each step along the critical path.

As an upper bound, parallelism gives the maximum possible speedup that can be achieved on any number of processors.

Finally, it also provides a limit on the possibility of attaining perfect linear speedup.

**Slackness** is the *parallelism* divided by  $P$  i.e.  $T_1 / PT_\infty$

A slackness less than one implies (by the span law) that perfect linear speedup is impossible on  $p$  processors.

# Analysis of Multithreaded Algorithms

## Analysis of Work:

For  $P\text{-FIB}(n)$ ,  $T_1(n) = T(n) = \Theta(\phi^n)$

## Analysis of Span:

For  $P\text{-FIB}(n)$ , the spawned call to  $P\text{-FIB}(n-1)$  in line 3 runs in parallel with the call to  $P\text{-FIB}(n-2)$  in line 4. Hence, we can express the span of  $P\text{-FIB}(n)$  as the recurrence:

$$\begin{aligned} T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1), \end{aligned}$$

$$T_\infty(n) = \Theta(n)$$

# Scheduling

Good performance depends upon more than just minimising work and span.

Strands must also be scheduled efficiently on the CPU's cores.

A multithreaded scheduler must schedule the computation with no advance knowledge of when strands will be spawned or completed i.e. it must operate *on-line*.

*Centralised schedulers* know the global state of the computation at any given time.

*Greedy schedulers* assign as many strands as possible in each step.

# Parallel Loops

**Example:** *matrix-vector multiplication.*

Let  $A = (a_{ij})$  be an  $n \times n$  matrix and let  $x = (x_j)$  be an  $n$ -vector. The product of  $A$  times  $x$  is other  $n$ -vector  $y = (y_i)$  defined by

$$y_i = \sum_{j=1}^n a_{ij} x_j$$

MAT-VEC-SEQ(  $A, x$  )

```
1   $n = A.rows$ 
2  Let  $y$  be a new vector of length  $n$ 
3  for  $i = 1$  to  $n$ 
4       $y_i = 0$ 
5  for  $i = 1$  to  $n$ 
6      for  $j = 1$  to  $n$ 
7           $y_i = y_i + a_{ij} x_j$ 
8  return  $y$ 
```

Serial code for matrix-vector multiplication

MAT-VEC(  $A, x$  )

```
1   $n = A.rows$ 
2  Let  $y$  be a new vector of length  $n$ 
3  parallel for  $i = 1$  to  $n$ 
4       $y_i = 0$ 
5  parallel for  $i = 1$  to  $n$ 
6      for  $j = 1$  to  $n$ 
7           $y_i = y_i + a_{ij} x_j$ 
8  return  $y$ 
```

Parallel code for matrix-vector multiplication

This code recursively spawns the first half of iterations of the loop to execute in parallel with the second half of the iterations followed by **sync** operation, thereby creating binary tree of executions where the leaves are individual loop iterations.

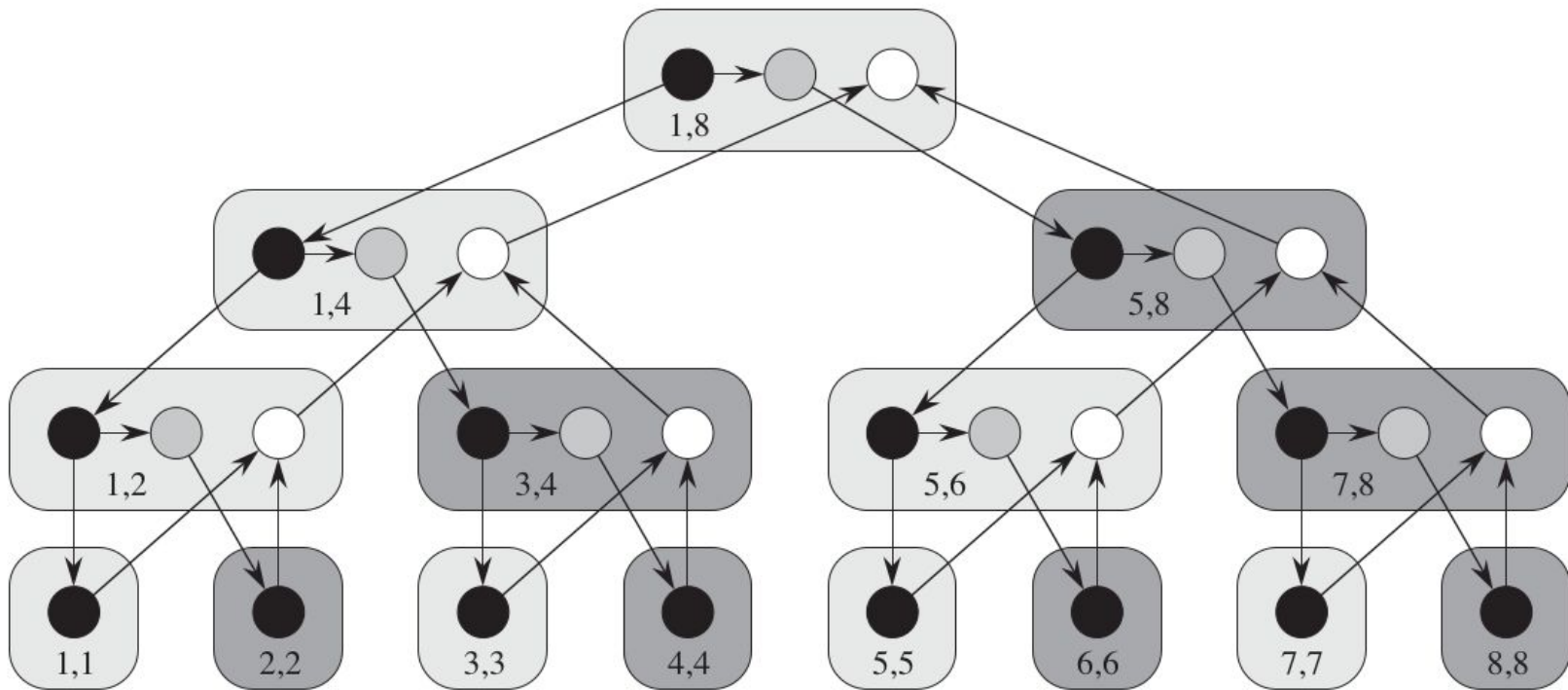
$$\begin{bmatrix} a_{11} & L & a_{1n} \\ a_{mid+1} & L & a_{mid+1n} \\ a_{n1} & L & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ M \\ x_{mid+1} \\ M \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ M \\ y_{mid+1} \\ M \\ y_n \end{bmatrix}$$

MAT-VEC-MAIN-LOOP(  $A, x, y, n, i, i'$  )

```

1  if  $i == i'$ 
2    for  $j = 1$  to  $n$ 
3       $y_i = y_i + a_{ij} x_j$ 
4  else  $mid = \text{floor}((i + i')/2)$ 
5    spawn MAT-VEC-MAIN-LOOP(  $A, x, y, n, i, mid$  )
6    MAT-VEC-MAIN-LOOP(  $A, x, y, n, mid + 1, i'$  )
7    sync

```



**Fig.** *dag* for  $\text{MAT-VEC-MAIN-LOOP}(A, x, y, 8, 1, 8)$

Span becomes,

$$T_{\infty}(n) = \Theta(\lg n) + \max_{1 \leq i \leq n} \text{iter}_{\infty}(i)$$

As a practical matter, dynamic-multithreading concurrency platforms sometimes *coarsen* the leaves of the recursion by executing several iterations in a single leaf, either automatically or under programmer control, thereby reducing the overhead of recursive spawning.

This reduced overhead comes at the expense of also reducing the parallelism, however, but if the computation has sufficient parallel slackness, near-perfect linear speedup need not be sacrificed.



# Race Conditions

A multithreaded algorithm is *deterministic* if it always does the same thing on the same input.

It is *nondeterministic* if its behaviour might vary from run to run.

Often, a multithreaded algorithm intended to be deterministic fails to be, because it contains a *determinacy race*. Race bugs are notoriously hard to find.

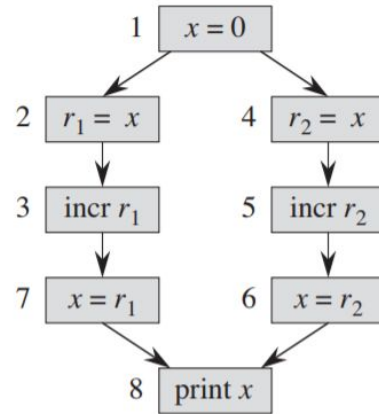
Famous race bugs include the **Therac-25** radiation therapy machine, which killed 3 people and injured several others.

The **North American Blackout of 2003** left over 50 million people without power.

A **determinacy race** occurs when two logically parallel instructions access the same memory locations and at least one of the instructions performs a write.

### RACE-EXAMPLE()

```
1   $x = 0$   
2  parallel for  $i = 1$  to 2  
3       $x = x + 1$   
4  print  $x$ 
```



(a)

step	$x$	$r_1$	$r_2$
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

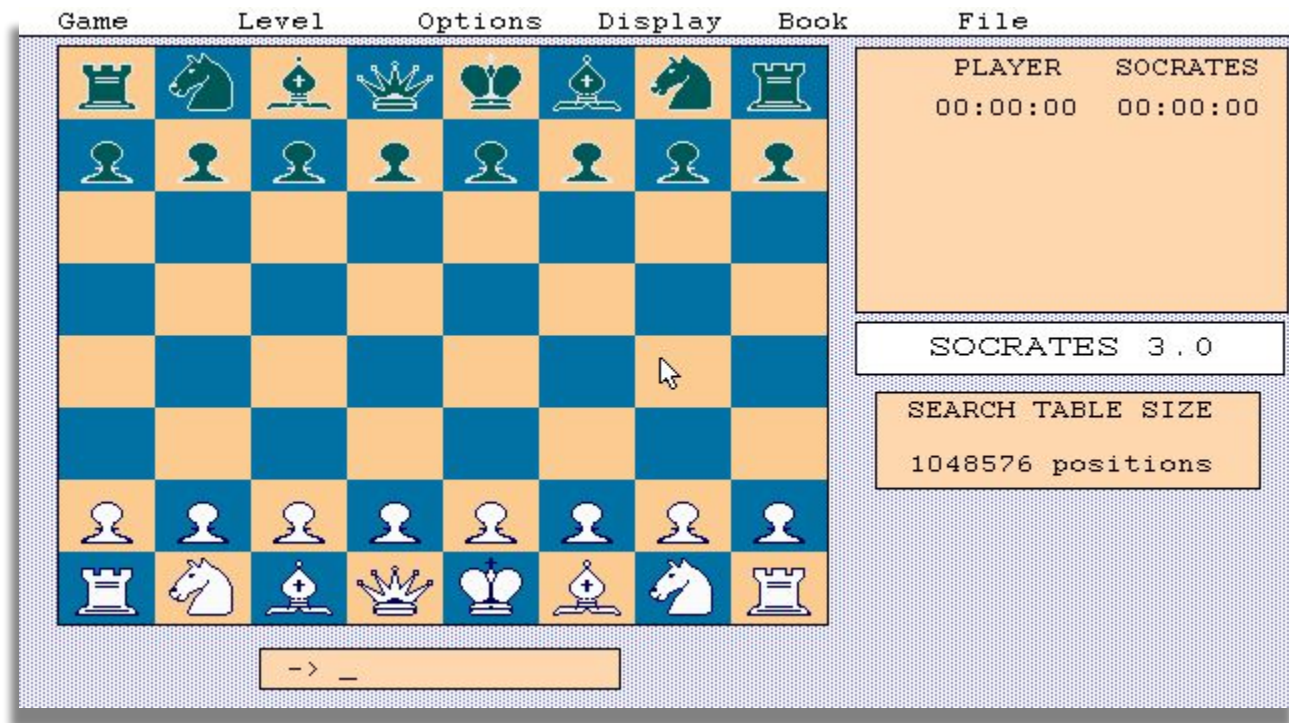
(b)

Between a spawn and the corresponding sync, *the code of the spawned child should be independent of the code of the parent*, including code executed by additional spawned or called children.

MAT-VEC-WRONG( $A, x$ )

```
1   $n = A.rows$ 
2  let  $y$  be a new vector of length  $n$ 
3  parallel for  $i = 1$  to  $n$ 
4       $y_i = 0$ 
5  parallel for  $i = 1$  to  $n$ 
6      parallel for  $j = 1$  to  $n$ 
7           $y_i = y_i + a_{ij}x_j$ 
8  return  $y$ 
```

# Case Study: Socrates Chess Game



A true story that occurred during the development of the world-class multithreaded chess-playing program **Socrates**.

The program was prototyped on a 32-processor computer but was ultimately to run on a supercomputer with 512 processors.

At one point, the developers incorporated an optimization into the program that reduced its running time on an important benchmark on the 32-processor machine from  $T_{32} = 65$  sec to  $T_{32} = 40$  sec.

Yet, the developers used the work and span performance measures to conclude that the optimized version, which was faster on 32 processors, *would actually be slower than the original version on 512 processors!* As a result, they abandoned the “optimization.”

Their proof was as follows:

**For 32-processor computer:**

*Originally,*

$$T_1 = 2048 \text{ s and } T_\infty = 1 \text{ s}$$

$$\text{Assuming, } T_\square = T_1 / P + T_\infty \quad \therefore T_{32} = 2048/32 + 1 = 65 \text{ s}$$

*Upon Optimisation,*

$$T_1' = 1024 \text{ s and } T_\infty' = 8 \text{ s} \quad \therefore T_{32} = 1024/32 + 8 = 40 \text{ s}$$

**For 512-processor computer:**

$$T_{512} = 2048/512 + 1 = 5 \text{ s}$$

$$T_{512}' = 2048/512 + 8 = 10 \text{ s}$$

The optimization that sped up the program on a 32-processor system slows down by 2 times on a 512-processor machine!

*Explanation:* The optimised version span of 8 which was not dominant term in  $T_{32}'$ , became the dominant term in  $T_{512}'$ .

**Thus, work and span can provide a better means of extrapolating performance rather than measured running times.**

## **Part II**

# Design and Analysis of Multithreaded Algorithms



# Analysis I: Multithreaded Matrix Multiplication

Standard matrix multiplication takes  $O(n^3)$  time.

SQUARE-MATRIX-MULTIPLY( $A, B$ )

```
1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5           $c_{ij} = 0$ 
6          for  $k = 1$  to  $n$ 
7               $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
8  return  $C$ 
```

P-SQUARE-MATRIX-MULTIPLY( $A, B$ )

```
1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  parallel for  $i = 1$  to  $n$ 
4      parallel for  $j = 1$  to  $n$ 
5           $c_{ij} = 0$ 
6          for  $k = 1$  to  $n$ 
7               $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
8  return  $C$ 
```

The work of *P-SQUARE-MATRIX-MULTIPLY*(*A*, *B*) is:

$$T_1(n) = \Theta(n^3) \quad (\text{obviously})$$

The span of *P-SQUARE-MATRIX-MULTIPLY*(*A*, *B*) is:

$$T_\infty(n) = \Theta(n)$$

Because it follows a path down the tree of recursion for the parallel for loop starting in line 3, then down the tree of recursion for the parallel for loop starting in line 4, and then executes all  $n$  iterations of the ordinary for loop starting in line 6, resulting in a total span of  $\Theta(\lg n) + \Theta(\lg n) + \Theta(n) = \Theta(n)$  .

Thus, the parallelism is  $\Theta(n^3)/\Theta(n) = \Theta(n^2)$

Using Divide and Conquer strategy:

SQUARE-MATRIX-MULTIPLY-RECURSIVE( $A, B$ )

```
1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  if  $n == 1$ 
4       $c_{11} = a_{11} \cdot b_{11}$ 
5  else partition  $A, B$ , and  $C$  as in equations (4.9)
6       $C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11})$ 
            $+ \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})$ 
7       $C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12})$ 
            $+ \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22})$ 
8       $C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11})$ 
            $+ \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21})$ 
9       $C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12})$ 
            $+ \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22})$ 
10 return  $C$ 
```

# P-MATRIX-MULTIPLY-RECURSIVE( $C, A, B$ )

```
1   $n = A.rows$ 
2  if  $n == 1$ 
3       $c_{11} = a_{11}b_{11}$ 
4  else let  $T$  be a new  $n \times n$  matrix
5      partition  $A, B, C$ , and  $T$  into  $n/2 \times n/2$  submatrices
           $A_{11}, A_{12}, A_{21}, A_{22}; B_{11}, B_{12}, B_{21}, B_{22}; C_{11}, C_{12}, C_{21}, C_{22};$ 
          and  $T_{11}, T_{12}, T_{21}, T_{22};$  respectively
6      spawn P-MATRIX-MULTIPLY-RECURSIVE( $C_{11}, A_{11}, B_{11}$ )
7      spawn P-MATRIX-MULTIPLY-RECURSIVE( $C_{12}, A_{11}, B_{12}$ )
8      spawn P-MATRIX-MULTIPLY-RECURSIVE( $C_{21}, A_{21}, B_{11}$ )
9      spawn P-MATRIX-MULTIPLY-RECURSIVE( $C_{22}, A_{21}, B_{12}$ )
10     spawn P-MATRIX-MULTIPLY-RECURSIVE( $T_{11}, A_{12}, B_{21}$ )
11     spawn P-MATRIX-MULTIPLY-RECURSIVE( $T_{12}, A_{12}, B_{22}$ )
12     spawn P-MATRIX-MULTIPLY-RECURSIVE( $T_{21}, A_{22}, B_{21}$ )
13     P-MATRIX-MULTIPLY-RECURSIVE( $T_{22}, A_{22}, B_{22}$ )
14     sync
15     parallel for  $i = 1$  to  $n$ 
16         parallel for  $j = 1$  to  $n$ 
17              $c_{ij} = c_{ij} + t_{ij}$ 
```

Recurrence for work becomes,

$$\begin{aligned}M_1(n) &= 8M_1(n/2) + \Theta(n^2) \\ &= \Theta(n^3)\end{aligned}$$

Recurrence for span becomes,

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n)$$

$$\therefore M_\infty(n) = \Theta(\lg^2 n)$$

Thus, the parallelism is,

$$M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$$

which is pretty high.

P-MATRIX-MULTIPLY-RECURSIVE( $C, A, B$ )

```
1   $n = A.rows$ 
2  if  $n == 1$ 
3       $c_{11} = a_{11}b_{11}$ 
4  else let  $T$  be a new  $n \times n$  matrix
5      partition  $A, B, C$ , and  $T$  into  $n/2 \times n/2$  submatrices
            $A_{11}, A_{12}, A_{21}, A_{22}; B_{11}, B_{12}, B_{21}, B_{22}; C_{11}, C_{12}, C_{21}, C_{22};$ 
           and  $T_{11}, T_{12}, T_{21}, T_{22}$ ; respectively
6      spawn P-MATRIX-MULTIPLY-RECURSIVE( $C_{11}, A_{11}, B_{11}$ )
7      spawn P-MATRIX-MULTIPLY-RECURSIVE( $C_{12}, A_{11}, B_{12}$ )
8      spawn P-MATRIX-MULTIPLY-RECURSIVE( $C_{21}, A_{21}, B_{11}$ )
9      spawn P-MATRIX-MULTIPLY-RECURSIVE( $C_{22}, A_{21}, B_{12}$ )
10     spawn P-MATRIX-MULTIPLY-RECURSIVE( $T_{11}, A_{12}, B_{21}$ )
11     spawn P-MATRIX-MULTIPLY-RECURSIVE( $T_{12}, A_{12}, B_{22}$ )
12     spawn P-MATRIX-MULTIPLY-RECURSIVE( $T_{21}, A_{22}, B_{21}$ )
13     P-MATRIX-MULTIPLY-RECURSIVE( $T_{22}, A_{22}, B_{22}$ )
14     sync
15     parallel for  $i = 1$  to  $n$ 
16         parallel for  $j = 1$  to  $n$ 
17              $c_{ij} = c_{ij} + t_{ij}$ 
```

## Analysis II: Multithreaded merge sort

Merge Sort already uses Divide and Conquer paradigm, making it a good candidate for multithreading using nested parallelism.

MERGE-SORT'(A, p, r)

```
1  if  $p < r$   
2       $q = \lfloor (p + r)/2 \rfloor$   
3      spawn MERGE-SORT'(A, p, q)  
4      MERGE-SORT'(A, q + 1, r)  
5      sync  
6      MERGE(A, p, q, r)
```

Because *MERGE* is serial, both its work and span are  $\Theta(n)$ . Thus, the following recurrence characterizes the work  $MS'_1(n)$  of *MERGE-SORT* of  $n$  elements.

$$\begin{aligned} MS'_1(n) &= 2 MS'_1(n/2) + \Theta(n) \\ &= \Theta(n \lg n) , \end{aligned}$$

Since two recursive calls of *MERGE-SORT*' can run in parallel, the span  $MS'_\infty$  is given by the recurrence:

$$\begin{aligned} MS'_\infty(n) &= MS'_\infty(n/2) + \Theta(n) \\ &= \Theta(n) . \end{aligned}$$

Thus, the parallelism of MERGE-SORT' comes to,

$$MS'_1(n)/MS'_\infty(n) = \Theta(\lg n)$$

This result is pretty unimpressive amount of parallelism.

To sort 10 million elements, it might work for a few processors, but will not scale up effectively for hundreds of processors.

The parallelism bottleneck in this merge sort is the serial MERGE procedure. Merge operation may seem inherently serial, but we can create a multithreaded version too!



# Algorithm for Multithreaded merge operation

Two sorted subarrays of an array  $T$ .

$T[p_1....r_1]$  of length  $n_1 = r_1 - p_1 + 1$  and  $T[p_2....r_2]$  of length  $n_2 = r_2 - p_2 + 1$

They're being merged into a subarray  $A[p_3....r_3]$  of length  $n_3 = r_3 - p_3 + 1 = n_1 + n_2$ .

We assume  $n_1 \geq n_2$ .

The middle element  $x = T[q_1]$  of the subarray  $T[p_1....r_1]$ , where  $q_1 = (p_1 + r_1)/2$ .

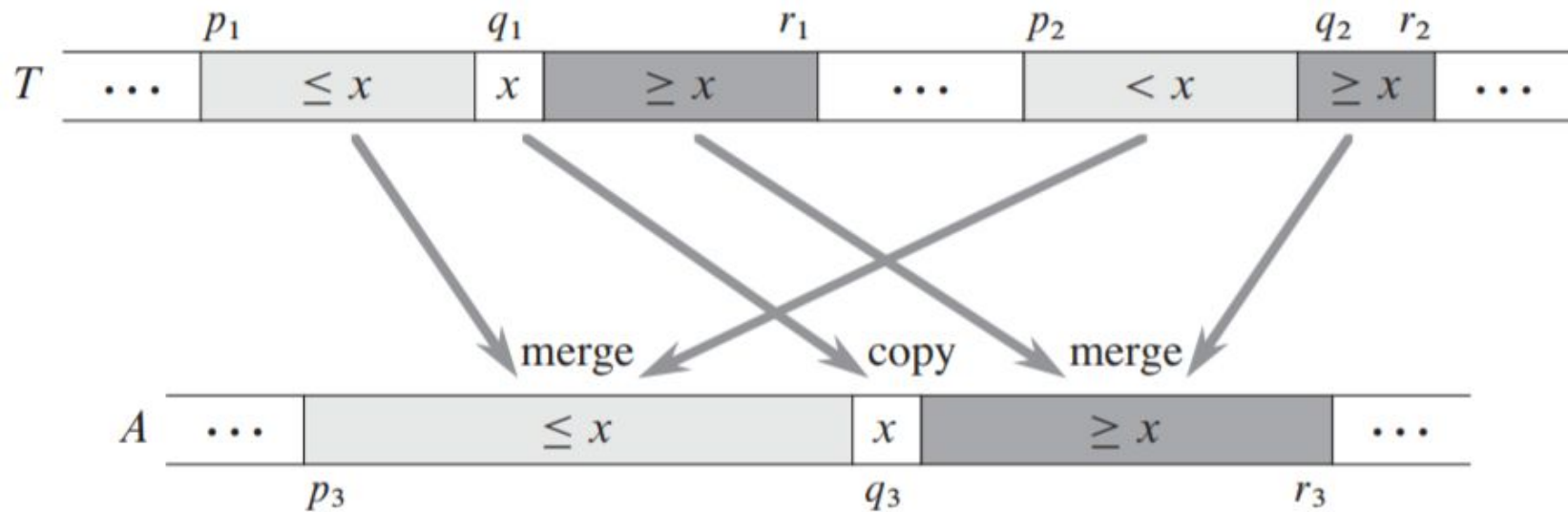
Because subarray is sorted,  $x$  is a median of  $T[p_1....r_1]$  : every element in  $T[p_1....q_1-1]$  is no more than  $x$ , and every element in  $T[q_1+1....r_1]$  is no less than  $x$ .

Now, we shall use Binary Search to find index  $q_2$  in the subarray  $T[p_2 \dots r_2]$ , so that the array will still be sorted if we inserted  $x$  between  $T[q_2 - 1]$  and  $T[q_2]$ .

We then merge the original subarrays  $T[p_1 \dots r_1]$  and  $T[p_2 \dots r_2]$  into  $A[p_3 \dots r_3]$  as follows:

1. Set  $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$ .
2. Copy  $x$  into  $A[q_3]$ .
3. Recursively merge  $T[p_1 \dots q_1 - 1]$  with  $T[p_2 \dots q_2 - 1]$ , and place the result into the subarray  $A[p_3 \dots q_3 - 1]$ .
4. Recursively merge  $T[q_1 + 1 \dots r_1]$  with  $T[q_2 \dots r_2]$ , and place the result into the subarray  $A[q_3 + 1 \dots r_3]$ .

When we compute  $q_3$ , the quantity  $q_1 - p_1$  is the number of elements in the subarray  $T[p_1 \dots q_1 - 1]$  and  $q_2 - p_2$  is the number of elements in subarray  $T[p_2 \dots q_2 - 1]$ . Thus, their sum is the number of elements that end up before  $x$  in the subarray  $A[p_3 \dots r_3]$ .



The base case occurs when  $n_1 = n_2 = 0$ , in which case we have no work to do to merge the two empty subarrays. Now, we must implement procedure *BINARY-SEARCH*( $x, T, p, r$ ):

Pseudocode is as follows:

*BINARY-SEARCH*( $x, T, p, r$ )

```
1   $low = p$ 
2   $high = \max(p, r + 1)$ 
3  while  $low < high$ 
4       $mid = \lfloor (low + high) / 2 \rfloor$ 
5      if  $x \leq T[mid]$ 
6           $high = mid$ 
7      else  $low = mid + 1$ 
8  return  $high$ 
```

**Note:**

*BINARY-SEARCH*( $x, T, p, r$ ) takes  $\Theta(\lg n)$  serial time in the worst case, where  $n = r - p + 1$  is the size of the subarray on which it runs. Since it is a serial procedure, work and span both are  $\Theta(\lg n)$ .

P-MERGE( $T, p_1, r_1, p_2, r_2, A, p_3$ )

```
1   $n_1 = r_1 - p_1 + 1$ 
2   $n_2 = r_2 - p_2 + 1$ 
3  if  $n_1 < n_2$                                 // ensure that  $n_1 \geq n_2$ 
4      exchange  $p_1$  with  $p_2$ 
5      exchange  $r_1$  with  $r_2$ 
6      exchange  $n_1$  with  $n_2$ 
7  if  $n_1 == 0$                                 // both empty?
8      return
9  else  $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$ 
10      $q_2 = \text{BINARY-SEARCH}(T[q_1], T, p_2, r_2)$ 
11      $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$ 
12      $A[q_3] = T[q_1]$ 
13     spawn P-MERGE( $T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3$ )
14     P-MERGE( $T, q_1 + 1, r_1, q_2, r_2, A, q_3 + 1$ )
15     sync
```

# Analysis of Multithreaded Merging

Because of the spawn in line 13 and the call in line 14 operate logically in parallel, we need examine only the costlier of the two calls. In the worst case, the maximum no. of elements in either of the recursive calls can be at most  $3n/4$  as shown below.

Lines 3-6 ensure that  $n_2 \leq n_1$ , it follows that  $n_2 = 2n_2/2 \leq (n_1 + n_2)/2 = n/2$ .

In the worst case, one of the two recursive calls merges  $\lfloor n_1/2 \rfloor$  elements of  $T[p_1 \dots r_1]$  with all  $n_2$  elements of  $T[p_2 \dots r_2]$ , and hence, no. of elements involved in the call are:

$$\begin{aligned} \lfloor n_1/2 \rfloor + n_2 &\leq n_1/2 + n_2/2 + n_2/2 \\ &= (n_1 + n_2)/2 + n_2/2 \\ &\leq n/2 + n/4 \\ &= 3n/4 . \end{aligned}$$

Adding in the  $O(\log n)$  cost of the call to *BINARY-SEARCH* in line 10, we obtain the following recurrence for the worst-case span.

$$PM_{\infty}(n) = PM_{\infty}(3n/4) + \Theta(\lg n) .$$

For the base case, the span is  $O(1)$ .

Solution to above recurrence is found and thus worst-case span is,

$$PM_{\infty}(n) = \Theta(\lg^2 n)$$

To calculate the work,

We can observe that even though recursions at lines 13 and 14 may each merge different no. of elements, they will always merge at most  $n$  elements.

While analyzing the span, we saw that a recursive call operates on at most  $3n/4$  elements. Thus, we obtain the recurrence as,

$$PM_1(n) = PM_1(\alpha n) + PM_1((1 - \alpha)n) + O(\lg n) \quad (\text{where } 1/4 \leq \alpha \leq 3/4)$$

$$\begin{aligned} PM_1(n) &\leq (c_1 \alpha n - c_2 \lg(\alpha n)) + (c_1(1 - \alpha)n - c_2 \lg((1 - \alpha)n)) + \Theta(\lg n) \\ &= c_1(\alpha + (1 - \alpha))n - c_2(\lg(\alpha n) + \lg((1 - \alpha)n)) + \Theta(\lg n) \\ &= c_1 n - c_2(\lg \alpha + \lg n + \lg(1 - \alpha) + \lg n) + \Theta(\lg n) \\ &= c_1 n - c_2 \lg n - (c_2(\lg n + \lg(\alpha(1 - \alpha)))) + \Theta(\lg n) \\ &\leq c_1 n - c_2 \lg n, \end{aligned}$$



We can choose  $c_2$  large enough that dominates the term.

Furthermore, we can choose  $c_1$  large enough to satisfy the base conditions of the recurrence. Since the work  $PM_1(n)$  of P-MERGE is both  $O(n)$  and  $\Omega(n)$ .

Thus,  $PM_1(n) = \Theta(n)$

Thus, parallelism of P-MERGE is

$$PM_1(n)/PM_\infty(n) = \Theta(n/\lg^2 n).$$

# Multithreaded Merge Sort

P-MERGE-SORT( $A, p, r, B, s$ )

```
1   $n = r - p + 1$ 
2  if  $n == 1$ 
3       $B[s] = A[p]$ 
4  else let  $T[1..n]$  be a new array
5       $q = \lfloor (p + r) / 2 \rfloor$ 
6       $q' = q - p + 1$ 
7      spawn P-MERGE-SORT( $A, p, q, T, 1$ )
8      P-MERGE-SORT( $A, q + 1, r, T, q' + 1$ )
9      sync
10     P-MERGE( $T, 1, q', q' + 1, n, B, s$ )
```

# Analysis of multithreaded merge sort

Now that we have a nicely parallelized multithreaded merging procedure, we can incorporate it into a multithreaded merge sort.

This version of merge sort is similar to the *MERGE-SORT'* procedure we saw earlier, but unlike *MERGE-SORT'*, it takes as an argument an output subarray  $B$ , which will hold the sorted result.

In particular, the call  $P\text{-MERGE-SORT}(A, p, r, B, s)$  sorts the elements in  $A[p \dots r]$  and stores them in  $B[s \dots s+r-p]$ .

The work is given by the recurrence,

$$\begin{aligned}PMS_1(n) &= 2 PMS_1(n/2) + PM_1(n) \\&= 2 PMS_1(n/2) + \Theta(n) .\end{aligned}$$

Solution is same as sequential merge sort i.e

$$PMS_1(n) = \Theta(n \lg n)$$

Thus, parallelism for multithreaded merge sort is,

$$\begin{aligned}PMS_1(n)/PMS_\infty(n) &= \Theta(n \lg n)/\Theta(\lg^3 n) \\&= \Theta(n/\lg^2 n) ,\end{aligned}$$

The span is given by the recurrence,

$$\begin{aligned}PMS_\infty(n) &= PMS_\infty(n/2) + PM_\infty(n) \\&= PMS_\infty(n/2) + \Theta(\lg^2 n) .\end{aligned}$$

A good implementation would sacrifice some parallelism by coarsening base cases to reduce constants hidden by the asymptotic notation.

The direct way to coarsen the base case is to switch to an ordinary serial sort, like quicksort, when array size is sufficiently small.

Thank You!