Multithreaded Algorithms

Motivation

Serial algorithms are suitable for running on a uniprocessor computer.

We will now extend our model to parallel algorithms that can run on a multiprocessor computer.

Computational Model

There exist many competing models of parallel computation that are essentially different. For example, one can have shared or distributed memory.

Since multicore processors are ubiquitous, we focus on a parallel computing model with shared memory.

Threads

- A thread in computer science is short for a thread of execution.
- A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.
- Threads are a way for a program to divide (termed "split") itself into two or more simultaneously (or pseudo-simultaneously) running tasks.
- The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources.
- Threads are lightweight, in terms of the system resources they consume, as compared with processes.

Threading Types

- ***Two types feasible:***
  - ***Static threading: OS controls, typically for single-core CPU's.***

    but multi-core CPU's use it if compiler guarantees safe execution
  - ***Dynamic threading: Program controls explicitly, threads are created/destroyed as needed, parallel computing model***

Threads allow concurrent execution of two or more parts of a program for maximum utilization of CPU.

Dynamic Multithreading

Programming a shared-memory parallel computer can be difficult and error-prone. In particular, it is difficult to partition the work among several threads so that each thread approximately has the same load.

A concurrency platform is a software layer that coordinates, schedules, and manages parallel-computing resources. We will use a simple extension of the serial programming model that uses the concurrency instructions parallel, spawn, and sync.

Spawn

Spawn: If spawn proceeds a procedure call, then the procedure instance that executes the spawn (the parent) may continue to execute in parallel with the spawned subroutine (the child), instead of waiting for the child to complete.

The keyword spawn does not say that a procedure must execute concurrently, but simply that it may.

At runtime, it is up to the scheduler to decide which subcomputations should run concurrently.

Sync

The keyword sync indicates that the procedure must wait for all its spawned children to complete.

Parallel

Many algorithms contain loops, where all iterations can operate in parallel. If the parallel keyword proceeds a for loop, then this indicates that the loop body can be executed in parallel.

Fibonacci Numbers

Definition

The Fibonacci numbers (0,1,2,3,5,8,13…) are defined by the recurrence:

$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-1} + F_{i-2}$

for $i > 1$.

Naive Algorithm

Computing the Fibonacci numbers can be done with the following algorithm:

```
Fibonacci(n)
if n < 2 then return n;
   x = Fibonacci(n-1);
   y = Fibonacci(n-2) ;
      return x + y;
```

Running Time

Let T(n) denote the running time of Fibonacci(n). Since this procedure contains two recursive calls and a constant amount of extra work, we get

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

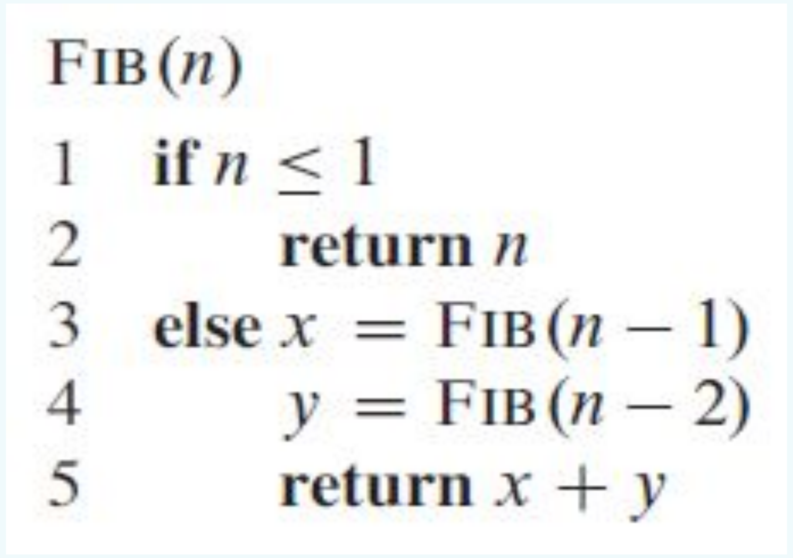
which yields $T(n) = \theta(F_n) = \theta(((1+\mathrm{sqrt}(5))/2)^n)$

Since this is an exponential growth, this is a particularly bad way to calculate Fibonacci numbers.

How would you calculate the Fibonacci numbers?

Fibonacci Example

Observe that within FIB(n), the two recursive calls in lines 3 and 4 to FIB(n-1) and FIB(n-2), respectively, are independent of each other: they could be called in either order, and the computation performed by one has no way affects the other. Therefore, the two recursive calls can run in parallel.

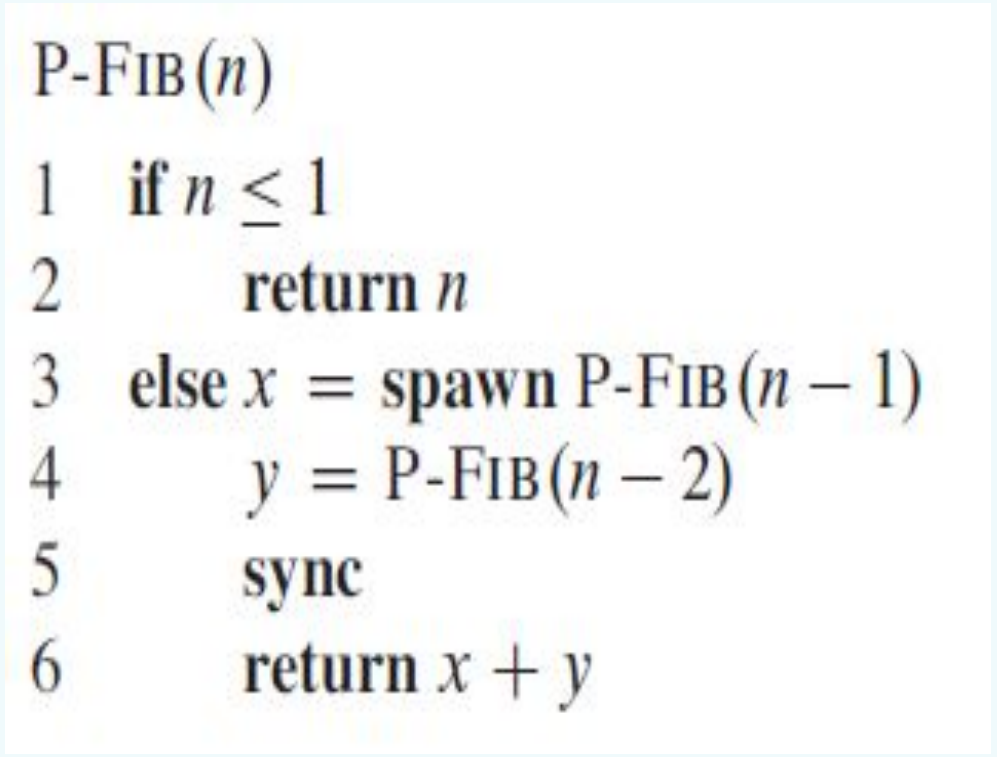


```
FIB($n$)
1  if $n \le 1$
2      return $n$
3  else $x$ = FIB($n - 1$)
4      $y$ = FIB($n - 2$)
5      return $x + y$
```

Fibonacci Example

Parallel algorithm to compute Fibonacci numbers:

We augment our pseudocode to indicate parallelism by adding the *concurrency keywords spawn and sync. Here is how we can rewrite the FIB procedure to use* dynamic multithreading:



```
P-FIB(n)
if $n \leq 1$
    return $n$
else $x$ = spawn P-FIB($n - 1$)
    $y$ = P-FIB($n - 2$)
    sync
    return $x + y$
```

Spawn, Sync & Parallel

Notice that if we delete the concurrency keywords **spawn and sync from P-FIB,** the resulting pseudocode text is identical to FIB (other than renaming the procedurein the header and in the two recursive calls).

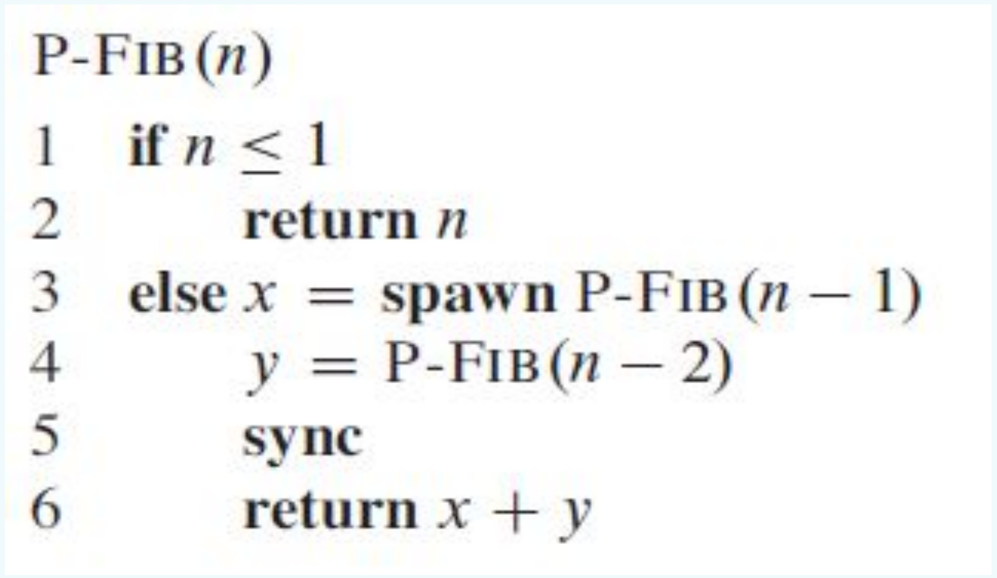
We define the ***serialization of a multithreaded*** algorithm to be the serial algorithm that results from deleting the multithreaded keywords:

spawn, sync, and parallel.

Spawn

Nested parallelism occurs when the keyword spawn precedes a procedure call, as in line 3.

The semantics of a spawn differs from an ordinary procedure call in that the procedure instance that executes the spawn—the ***parent—may continue*** to execute in parallel with the spawned subroutine—its ***child—instead of waiting*** for the child to complete, as would normally happen in a serial execution.

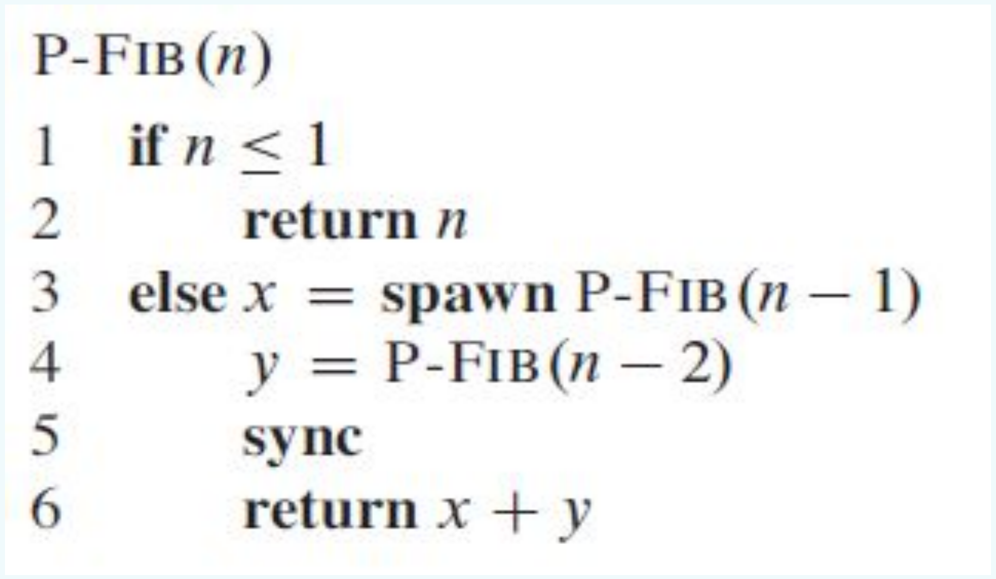


```
P-FIB($n$)
1  if $n \le 1$
2      return $n$
3  else $x = $ spawn P-FIB($n - 1$)
4      $y = $ P-FIB($n - 2$)
5      sync
6      return $x + y$
```

Spawn

In this case, while the spawned child is computing P-FIB(n-1), the parent may go on to compute P-FIB(n-2)in line 4 in parallel with the spawned child.

Since the P-FIB procedure is recursive, these two subroutine calls themselves create nested parallelism, as do their children, thereby creating a potentially vast tree of subcomputations, all executing in parallel.



```
P-FIB($n$)
1  if $n \leq 1$
2      return $n$
3  else $x$ = spawn P-FIB($n - 1$)
4      $y$ = P-FIB($n - 2$)
5      sync
6      return $x + y$
```

Spawn

The keyword **spawn does not say, however, that a procedure *must* *execute concurrently*** with its spawned children, only that it *may*.

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

At runtime, it is up to a ***scheduler to determine*** which subcomputations actually run concurrently by assigning them to available processors as the computation unfolds.

Sync

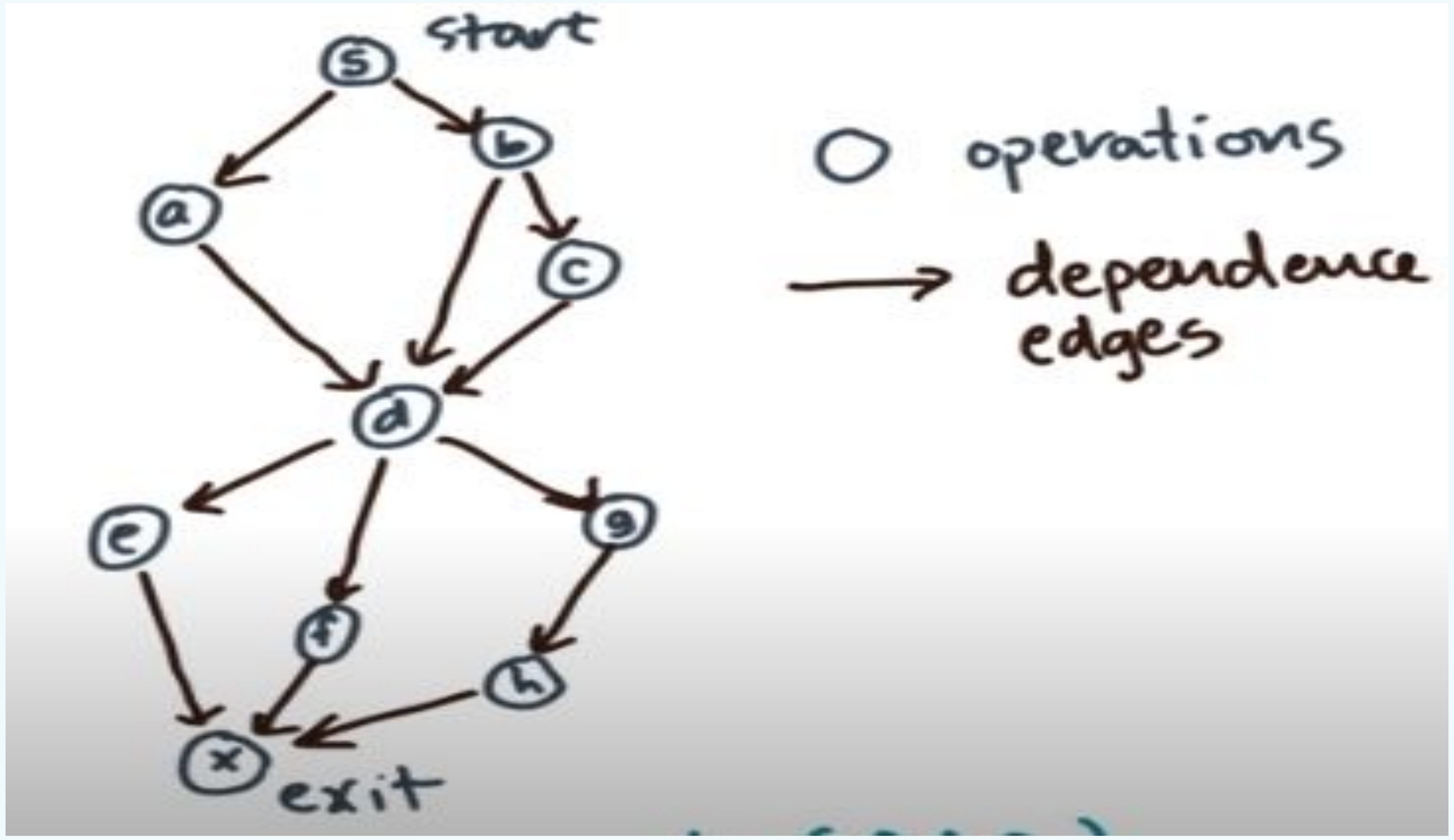
A procedure cannot safely use the values returned by its spawned children until after it executes a **sync statement, as in line 5. The keyword sync indicates that** the procedure must wait as necessary for all its spawned children to complete before proceeding to the statement after the **sync**.

In the P-FIB procedure, a sync is required before the **return statement in line 6 to avoid the anomaly that would** occur if x and y were summed before x was computed.

In addition to explicit synchronization provided by the **sync statement, every procedure executes a sync** implicitly before it returns, thus ensuring that all its children terminate before it does.

A model for multithreaded execution

It helps to think of a ***multithreaded computation—the set of runtime instructions*** executed by a processor on behalf of a multithreaded program—as a directed acyclic graph G =(V,E), called a ***computation dag.***



Computation DAG

Multithreaded computation can be better understood with the help of a computation directed acyclic graph G=(V,E).

The vertices V in the graph are the instructions.

The edges E represent dependencies between instructions.

An edge (u,v) is in E means that the instruction u must execute before instruction v.

Strand and Threads

A sequence of instructions containing no parallel control (spawn, sync, return from spawn, parallel) can be grouped into a single strand.

A strand of maximal length will be called a thread.

Computation DAG

A computation directed acyclic graph G=(V,E) consists a vertex set V that comprises the threads of the program.

The edge set E contains an edge (u,v) if and only if the thread u need to execute before thread v.

If there is an edge between thread u and v, then they are said to be (logically) in series. If there is no thread, then they are said to be (logically) in parallel.

Edge Classification

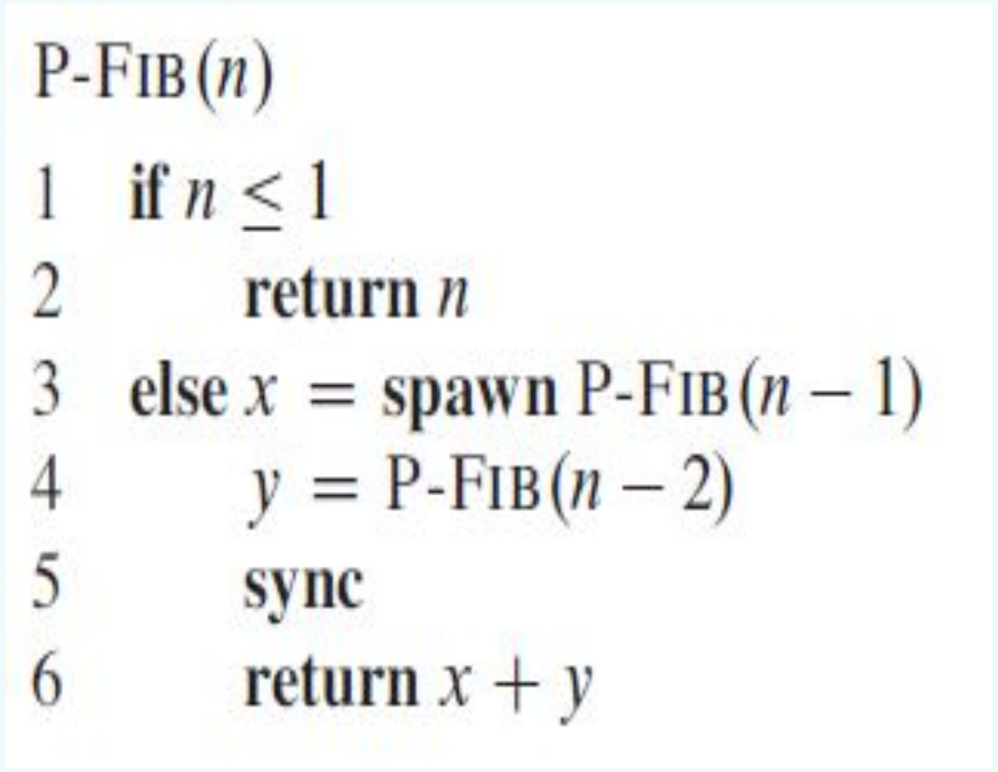
A continuation edge (u,v) connects a thread u to its successor v within the same procedure instance.

When a thread u spawns a new thread v, then (u,v) is called a spawn edge.

When a thread v returns to its calling procedure and x is the thread following the parallel control, then the return edge (v,x) is included in the graph.

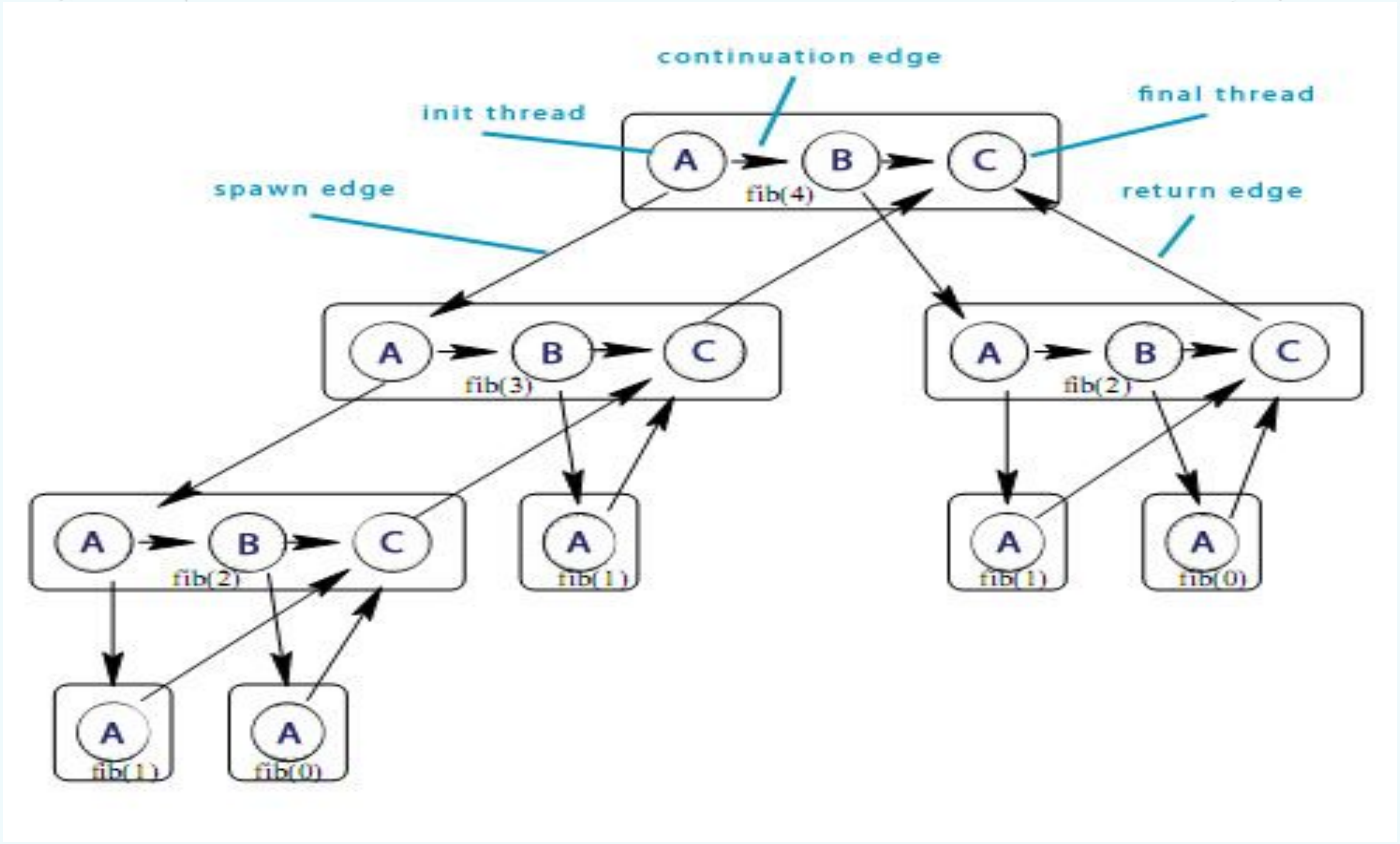
Fibonacci Example

Parallel algorithm to compute Fibonacci numbers:

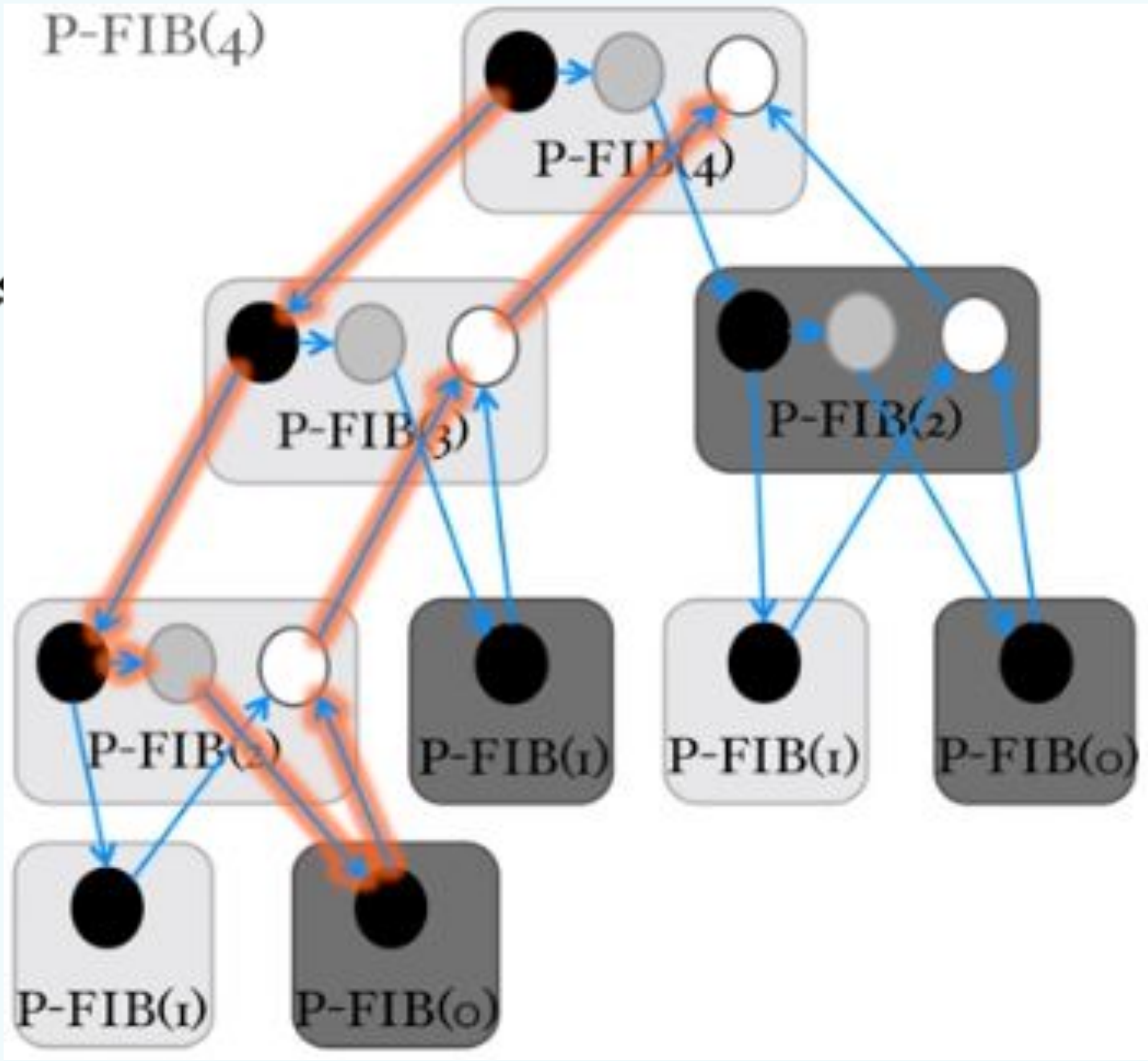


```
P-FIB(n)
if $n \le 1$
    return $n$
else $x$ = spawn P-FIB($n - 1$)
    $y$ = P-FIB($n - 2$)
    sync
    return $x + y$
```

Fibonacci(4)


continuation edge
init thread
final thread
spawn edge
return edge
A
B
C
fib(4)
fib(3)
fib(2)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)

Parallel Algorithms



```
P-FIB(n)
1  if $n \le 1$
2      return $n$
3  else $x$ = spawn P-FIB($n - 1$)
4      $y$ = P-FIB($n - 2$)
5      sync
6      return $x + y$
```

- *Each circle represents one strand (a chain of instructions which contains no parallel control).*
- *Black dots : base case or part of the procedure up to the spawn of P-FIN(n-1) in line 3.*
- *Grey dots: regular execution ie the part of the procedure that calls P-FIN(n-2) in line 4 up to the sync in line 5.*
- *White dots: part of the procedure after sync up to the point where it returns the result.*

Performance Measures

DAG: directed acyclic graph. Vertices are the circles for spawn, sync or procedure call. For a problem of size n:

- **Span** S or $T_\infty(n)$. Number of vertices on the longest directed path from start to finish in the computation DAG. (The critical path).

  The run time if each vertex of the DAG has its own processor.

- **Work** W or $T_1(n)$. Total time to execute the entire computation on one processor. Defined as the number of vertices in the computation DAG

- Tp(n). Total time to execute entire computation with p processors

- Speed up = T1/Tp. How much faster it is.

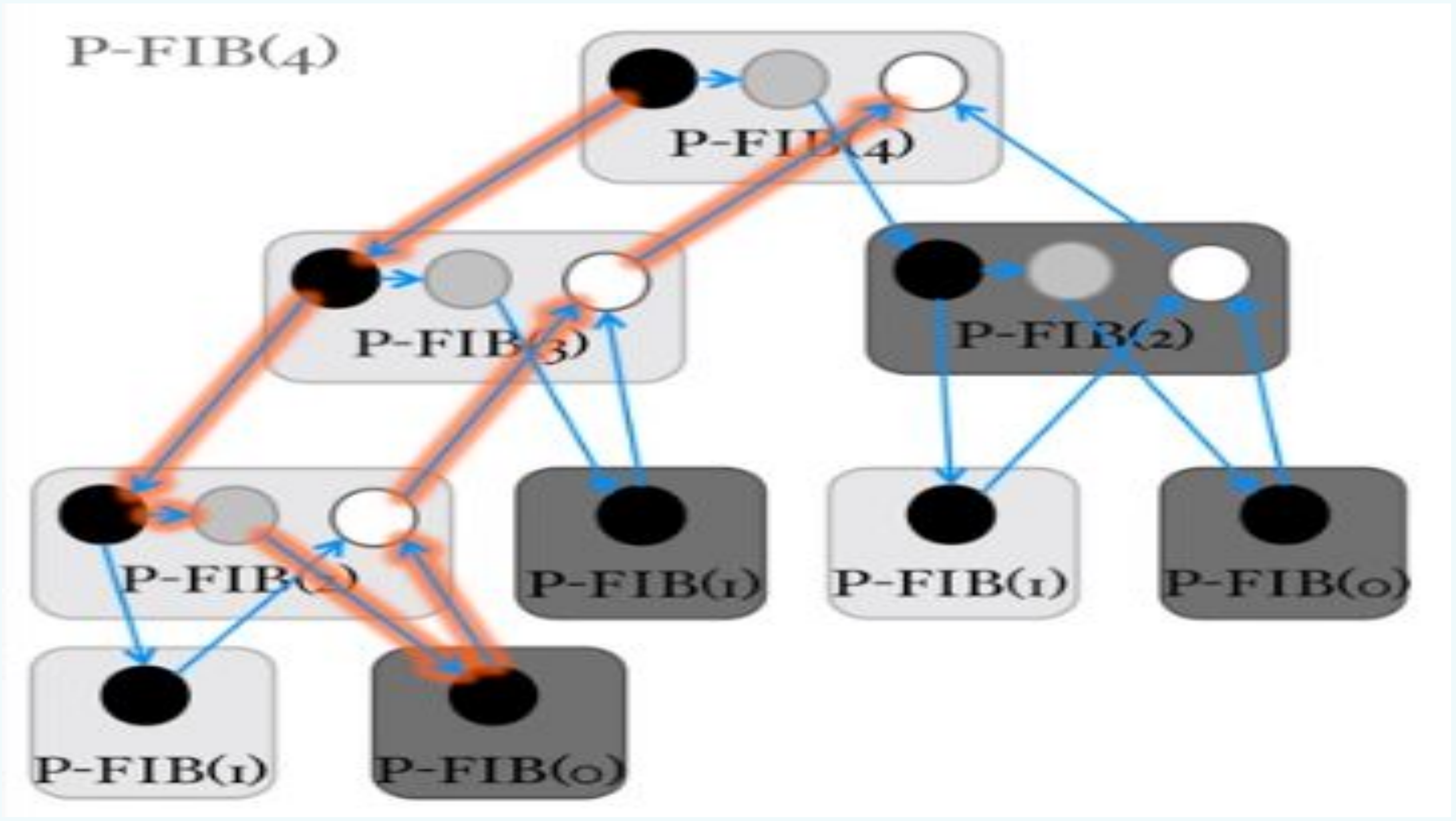
- Parallelism = T1/ T_∞. The maximum possible speed up.

Performance Measures

The work of a multithreaded computation is the total time to execute the entire computation on one processor.

Work = sum of the times taken by each thread

= 17 time units

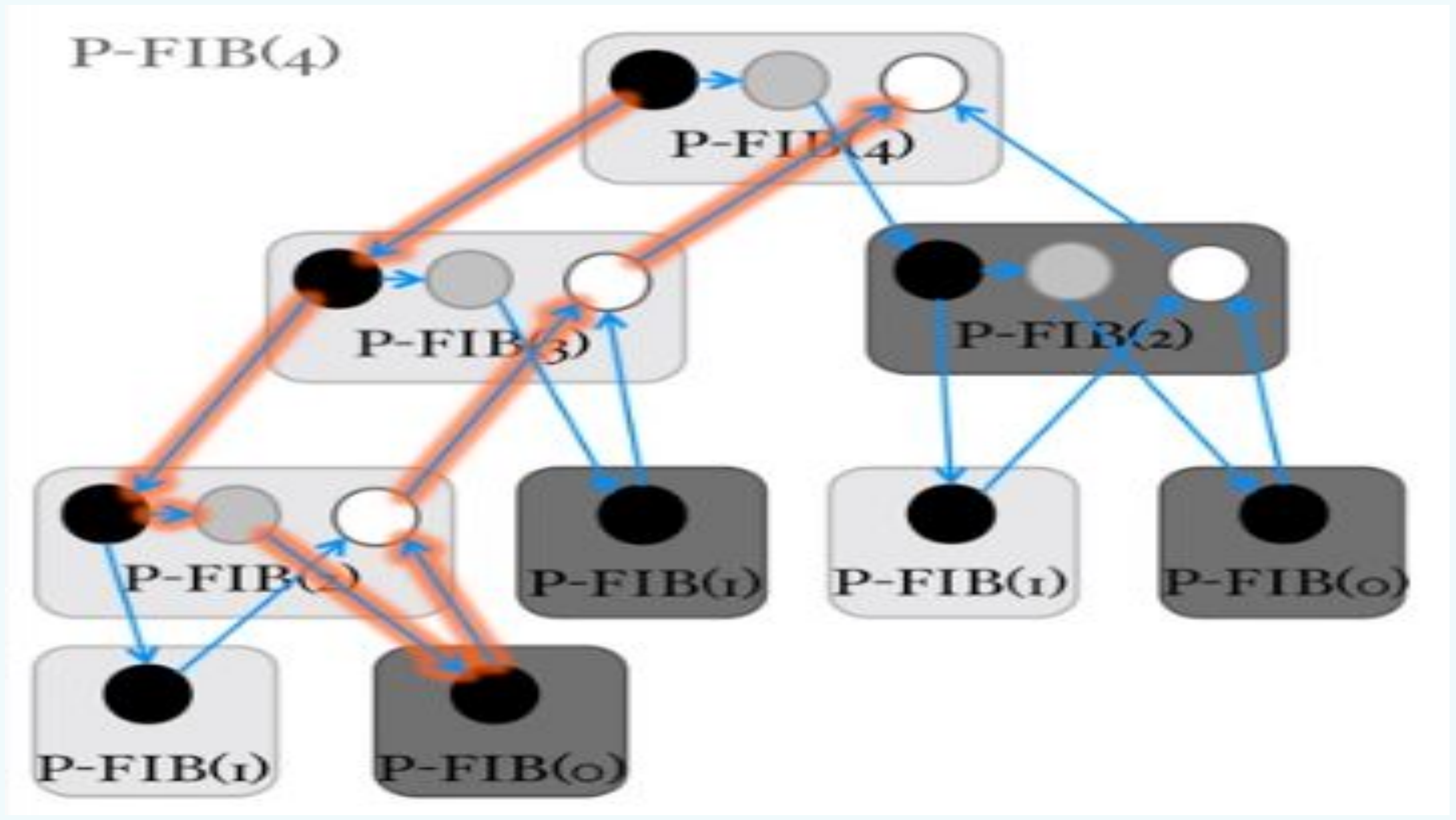


Performance Measures

The span is the longest time to execute the threads along any path of the computational directed acyclic graph.

Span =the number of vertices on a longest or critical path

span = 8 time units



Performance Measure Example

In Fibonacci(4), we have

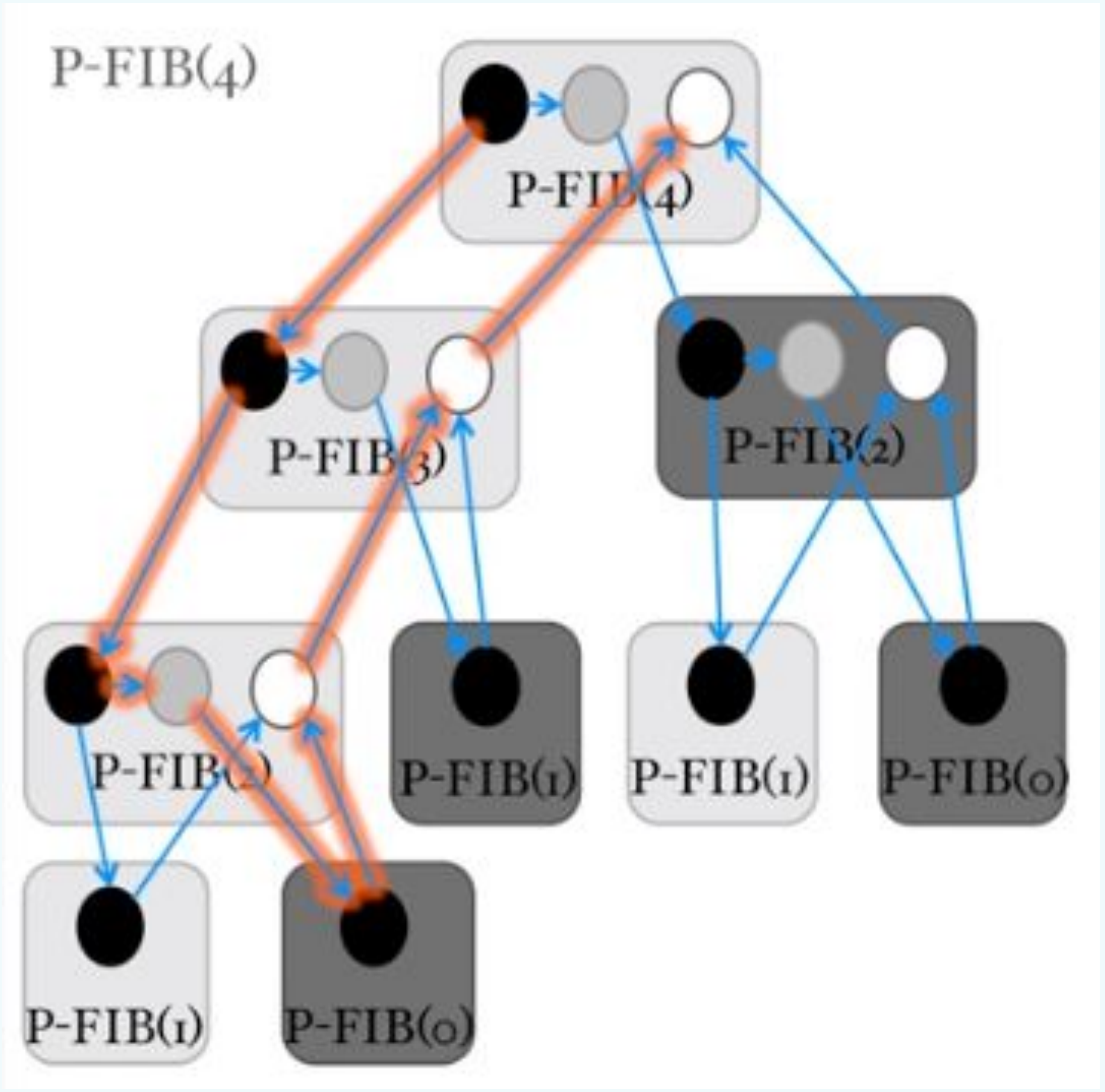
17 vertices = 17 threads.

8 vertices on longest path.

Assuming unit time for each thread, we get

work = 17 time units

span = 8 time units



The actual running time of a multithreaded computation depends not just on its work and span, but also on how many processors (cores) are available, and how the scheduler allocates strands to processors.

Running time on P processors is indicated by subscript P

- T_1 running time on a single processor
- T_P running time on P processors
- T_∞ running time on unlimited processors

Work Law

An ideal parallel computer with P processors can do at most P units of work, and thus in T_p time, it can perform at most PT_p work.

Since the total work to do is T1, we have $PT_p >= T1$.

Dividing by P yields the ***work law:***

$T_p >= T_1/P$

Span Law

A P-processor ideal parallel computer cannot run faster than a machine with unlimited number of processors.

However, a computer with unlimited number of processors can emulate a P-processor machine by using just P of its processors. Therefore,

$T_p >= T_\infty$

which is called the span law.

- T_p running time on P processors
- T_∞ running time on unlimited processors

Speedup and Parallelism

The speed up of a computation on P processors is defined as T_1 / T_p

i.e. how many times faster the computations on P processors than on 1 processor (How much faster it is).

Thus, speedup on P processors can be at most P.

Speedup and Parallelism

The parallelism (max possible speed up). of a multithreaded computation is given by T_1 / T_∞

We can view the parallelism from three perspectives.

- 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, the parallelism gives the maximum possible speedup that can be achieved on any number of processors.
- Finally, and perhaps most important, the parallelism provides a limit on the possibility of attaining perfect linear speedup. Specifically, once the number of processors exceeds the parallelism, the computation cannot possibly achieve perfect linear speedup.

Speedup and Parallelism

Consider the computation P-FIB(4) and assume that each strand takes unit time.

Since the work is T1 = 17 and the span is $T_\infty = 8$,

the parallelism is $T1/T_\infty = 17/8 = 2.125$.

Consequently, achieving much more than double the speedup is impossible, no matter how many processors we employ to execute the computation.

Scheduling

The performance depends not just on the work and span. Additionally, the strands must be scheduled efficiently onto the processors of the parallel machines.

The strands must be mapped to static threads, and the operating system schedules the threads on the processors themselves.

The scheduler must schedule the computation with no advance knowledge of when the strands will be spawned or when they will complete; it must operate online.

Greedy Scheduler

We will assume a greedy scheduler in our analysis, since this keeps things simple. A greedy scheduler assigns as many strands to processors as possible in each time step.

On P processors, if at least P strands are ready to execute during a time step, then we say that the step is a complete step; otherwise we say that it is an incomplete step.

Greedy Scheduler Theorem

On an ideal parallel computer with P processors, a greedy scheduler executes a multithreaded computation with work T_1 and span T_∞ in time

$T_P <= T_1 / P + T_\infty$

[Given the fact the best we can hope for on P processors is $T_P = T_1 / P$ by the work law, and $T_P = T_\infty$ by the span law, the sum of these two bounds]

Corollary

The running time of any multithreaded computation scheduled by a greedy scheduler on an ideal parallel computer with P processors is within a factor of 2 of optimal.

Proof.: The T_P^* be the running time produced by an optimal scheduler. Let T_1 be the work and T_∞ be the span of the computation. Then $T_P^* >= \max(T_1 /P, T_\infty)$. By the theorem,

$$T_P <= T_1 /P + T_\infty <= 2 \max(T_1 /P, T_\infty) <= 2\ T_P^*$$

Slackness

The parallel slackness of a multithreaded computation executed on an ideal parallel computer with P processors is the ratio of parallelism by P.

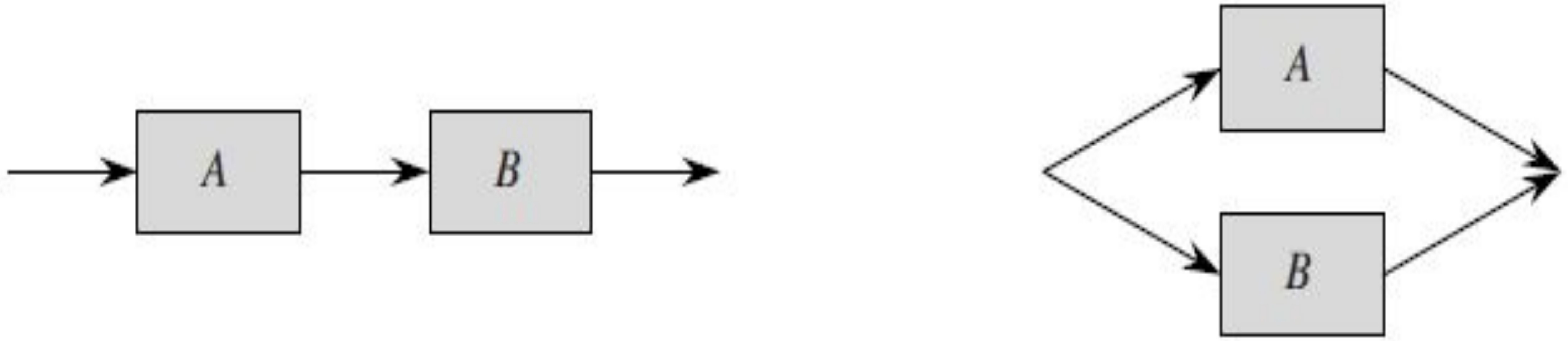
Slackness = $(T_1 / T_\infty) / P$

If the slackness is less than 1, we cannot hope to achieve a linear speedup.

Speedup

Let T_P be the running time of a multithreaded computation produced by a greedy scheduler on an ideal computer with P processors. Let T_1 be the work and T_∞ be the span of the computation. If the slackness is big, $P \ll (T_1 / T_\infty)$, then

T_P is approximately T_1 / P.



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)

Figure 27.3 The work and span of composed subcomputations. **(a)** When two subcomputations are joined in series, the work of the composition is the sum of their work, and the span of the composition is the sum of their spans. **(b)** When two subcomputations are joined in parallel, the work of the composition remains the sum of their work, but the span of the composition is only the maximum of their spans.

Back to Fibonacci

Parallel Fibonacci Computation

Parallel algorithm to compute Fibonacci numbers:

```
Fibonacci(n)
if n < 2 then return n;
x = spawn Fibonacci(n-1);   // parallel execution
y = spawn Fibonacci(n-2) ;  // parallel execution
sync;  // wait for results of x and y
return x + y;
```

Work of Fibonacci

We want to know the work and span of the Fibonacci computation, so that we can compute the parallelism (work/span) of the computation.

The work T_1 is straightforward, since it amounts to compute the running time of the serialized algorithm.

$T_1 = \theta(((1+\mathrm{sqrt}(5))/2)^n)$

Span of Fibonacci

Recall that the span T_∞ in the longest path in the computational DAG. Since Fibonacci(n) spawns

- Fibonacci(n-1)
- Fibonacci(n-2)

we have

$T_\infty(n) = \max(T_\infty(n-1) , T_\infty(n-2)) + \theta(1) = T_\infty(n-1) + \theta(1)$

which yields $T_\infty(n) = \theta(n)$.

Parallelism of Fibonacci

The parallelism of the Fibonacci computation is

$T_1(n)/T_\infty(n) = \theta(((1+sqrt(5))/2)^n / n)$

which grows dramatically as n gets large.

Therefore, even on the largest parallel computers, a modest value of n suffices to achieve near perfect linear speedup, since we have considerable parallel slackness.

Parallel loops

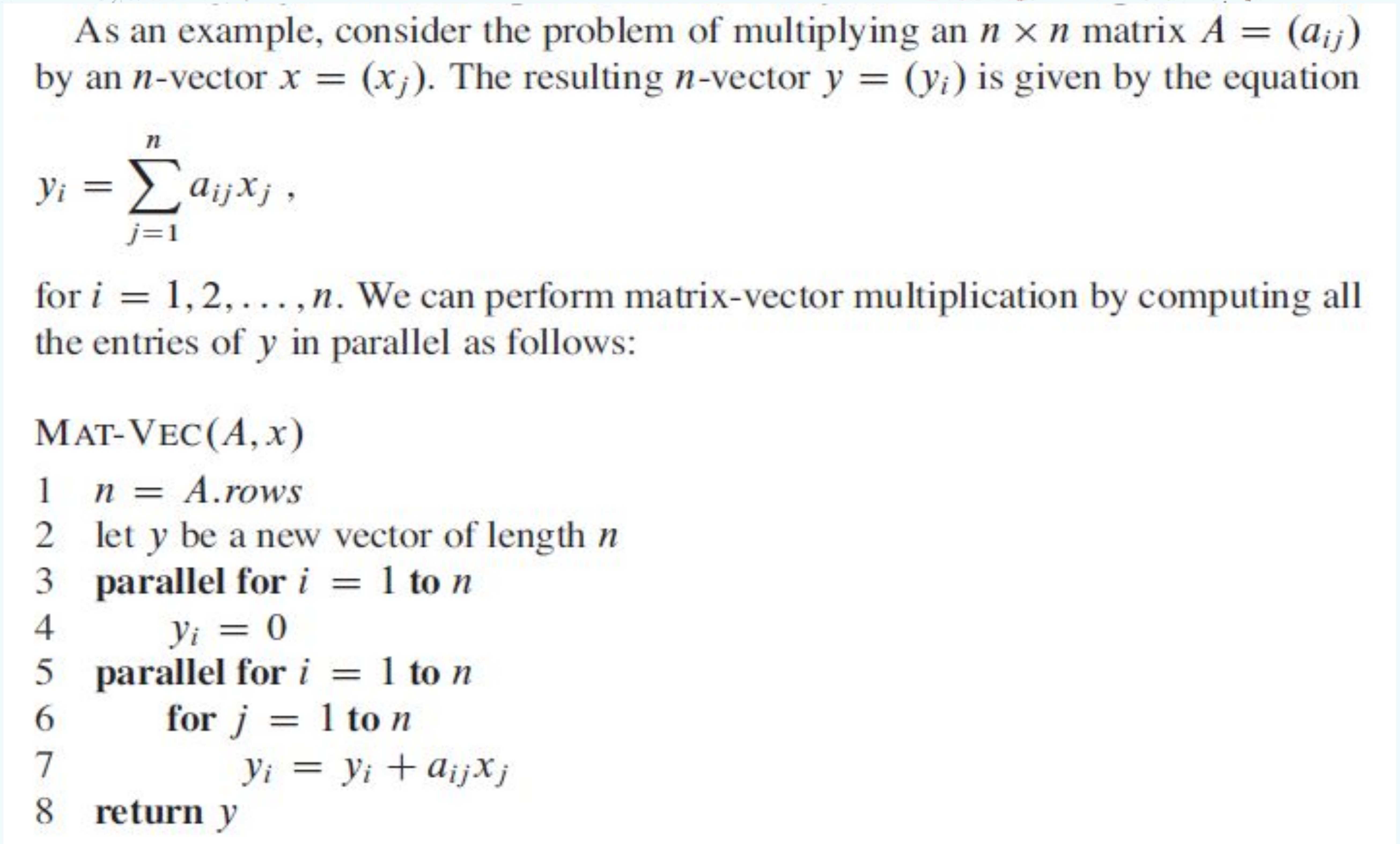
Parallel loops

Many algorithms contain loops all of whose iterations can operate in parallel.

We can parallelize such loops using the **spawn and sync keywords**, but it is much more convenient to specify directly that the iterations of such loops can run concurrently.

The pseudocode provides this functionality via the **parallel** concurrency keyword, which precedes the **for keyword in a for loop statement.**

Parallel loops



As an example, consider the problem of multiplying an $n \times n$ matrix $A = (a_{ij})$ by an n-vector $x = (x_j)$. The resulting n-vector $y = (y_i)$ is given by the equation

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

for $i = 1, 2, \ldots, n$. We can perform matrix-vector multiplication by computing all the entries of y in parallel as follows:

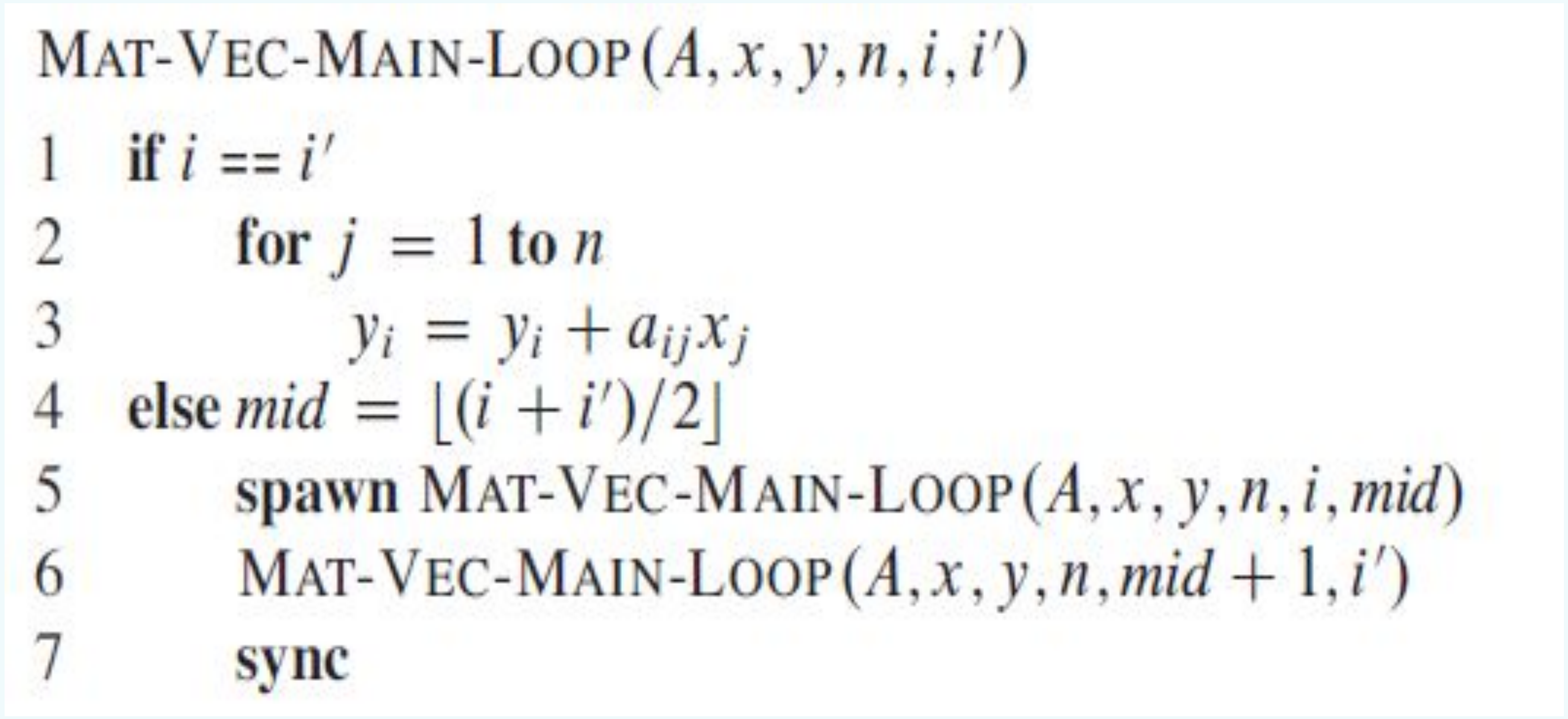
```
MAT-VEC($A, x$)
$n = A.rows$
let $y$ be a new vector of length $n$
**parallel for** $i = 1$ **to** $n$
    $y_i = 0$
**parallel for** $i = 1$ **to** $n$
    **for** $j = 1$ **to** $n$
        $y_i = y_i + a_{ij} x_j$
**return** $y$
```

Parallel loops

The **parallel for** keywords in lines 3 and 5 **indicate that the iterations** of the respective loops may be run concurrently.

A compiler can implement each **parallel for loop as a divide-and-conquer subroutine using nested parallelism.**

For example, the **parallel for loop in lines 5–7 can be implemented with the call** MAT-VEC-MAIN-LOOP(A, x, y, n, 1, n)



```
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 $mid = \lfloor (i + i')/2 \rfloor$
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
```

Race Conditions

Race Conditions

A multithreaded algorithm is deterministic if and only if it does the same thing on the same input, no matter how the instructions are scheduled.

A multithreaded algorithm is nondeterministic if its behavior might vary from run to run.

Often, a multithreaded algorithm that is intended to be deterministic fails to be.

Determinacy Race

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

```
Race-Example()
x = 0
parallel for i = 1 to 2 do
        x = x+1
print x
```

Determinacy Race

When a processor increments x, the operation is not indivisible, but composed of a sequence of instructions.

1) Read x from memory into one of the processor's registers

2) Increment the value of the register

3) Write the value in the register back into x in memory

Determinacy Race

x = 0

assign r1 = 0

incr r1, so r1=1

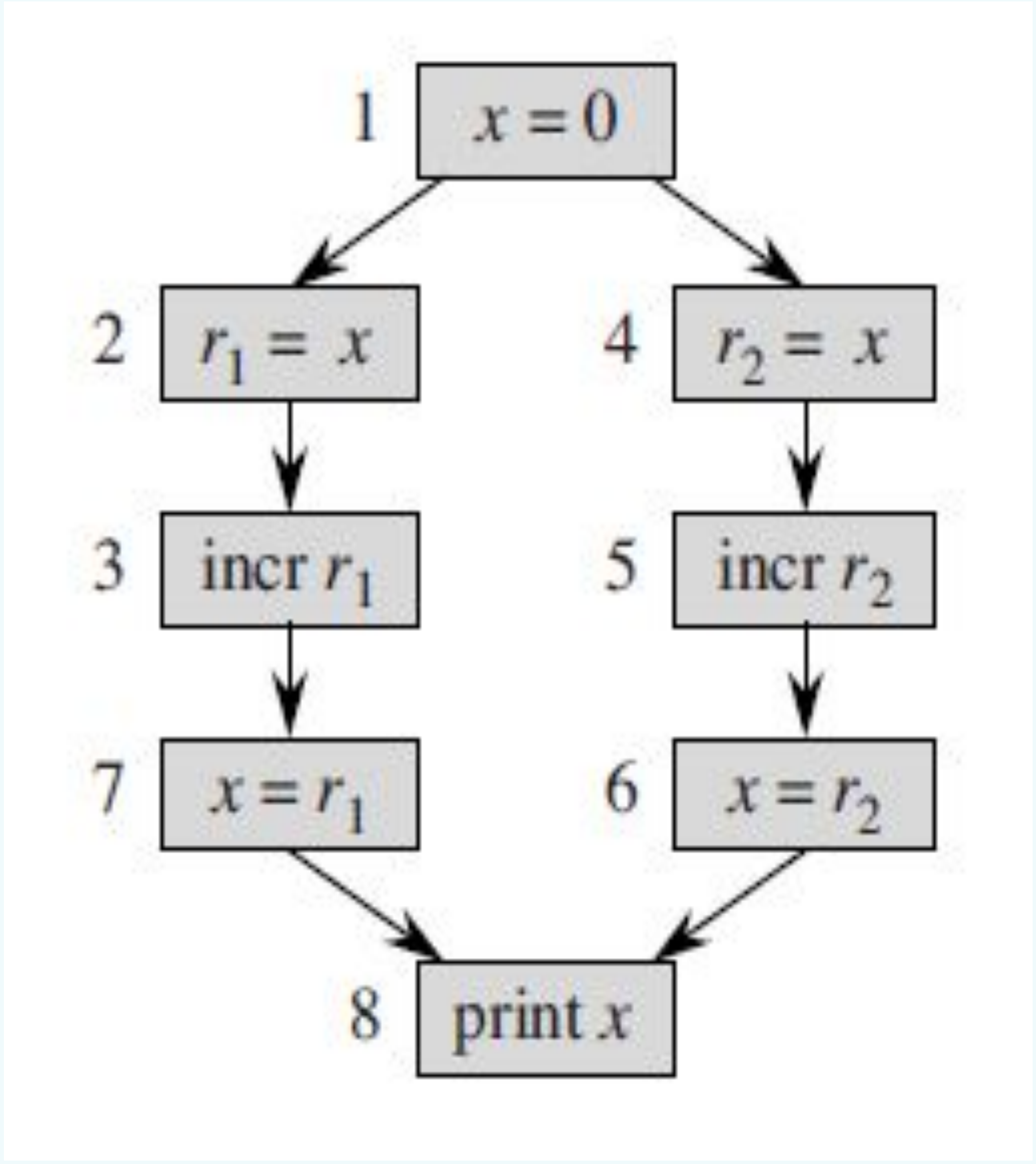
assign r2 = 0

incr r2, so r2 = 1

write back x = r1

write back x = r2

print x // now prints 1 instead of 2



Determinacy Race

If the effect of the parallel execution were that processor 1 executed all its instructions before processor 2, the value 2 would be printed.

Conversely, if the effect were that processor 2 executed all its instructions before processor 1, the value 2 would still be printed.

When the instructions of the two processors execute at the same time, however, it is possible, as in this example execution, that one of the updates to x is lost.

Determinacy Race

Generally, most orderings produce correct . But some orderings generate improper results when the instructions interleave. Consequently, races can be extremely hard to test for. You can run tests for days and never see the bug, only to experience a catastrophic system crash in the field when the outcome is critical.

Although we can cope with races in a variety of ways, including using **mutual exclusion locks** and other methods of synchronization, for our purposes, we shall simply ensure that strands that operate in parallel are ***independent: they have no*** determinacy races among them.

Thus, in a **parallel for construct, all the iterations** should be independent.

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.

Matrix Multiplication

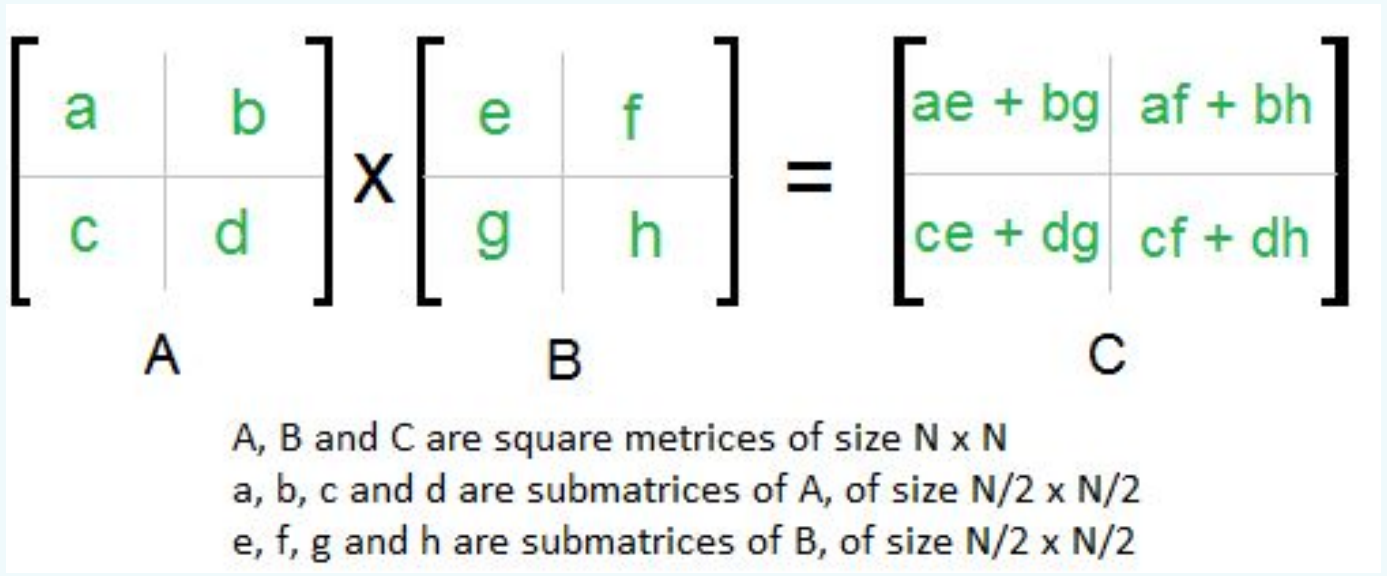
Matrix Multiplication: Naïve Method

```
void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

Time Complexity of above method is $O(N^3)$

Matrix Multiplication: Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.
1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.

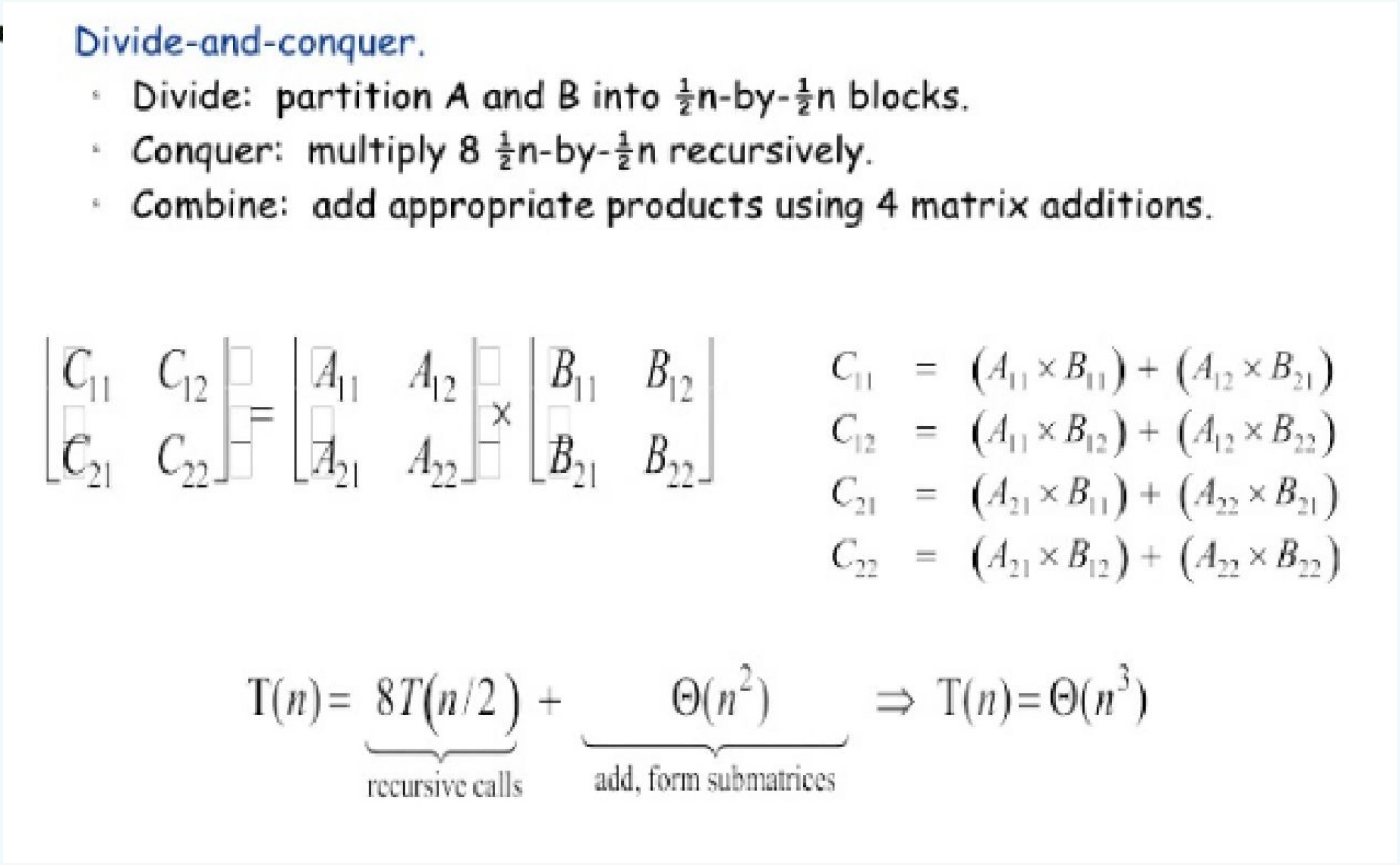

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}_{C}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

$T(N) = 8T(N/2) + O(N^2)$ From Master's Theorem, time complexity of above method is **$O(N^3)$** which is unfortunately same as the above naive method.

Matrix Multiplication: Divide and Conquer



Divide-and-conquer.

- Divide: partition A and B into $\frac{1}{2}$n-by-$\frac{1}{2}$n blocks.
- Conquer: multiply 8 $\frac{1}{2}$n-by-$\frac{1}{2}$n recursively.
- Combine: add appropriate products using 4 matrix additions.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

$$\mathrm{T}(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow \mathrm{T}(n) = \Theta(n^3)$$

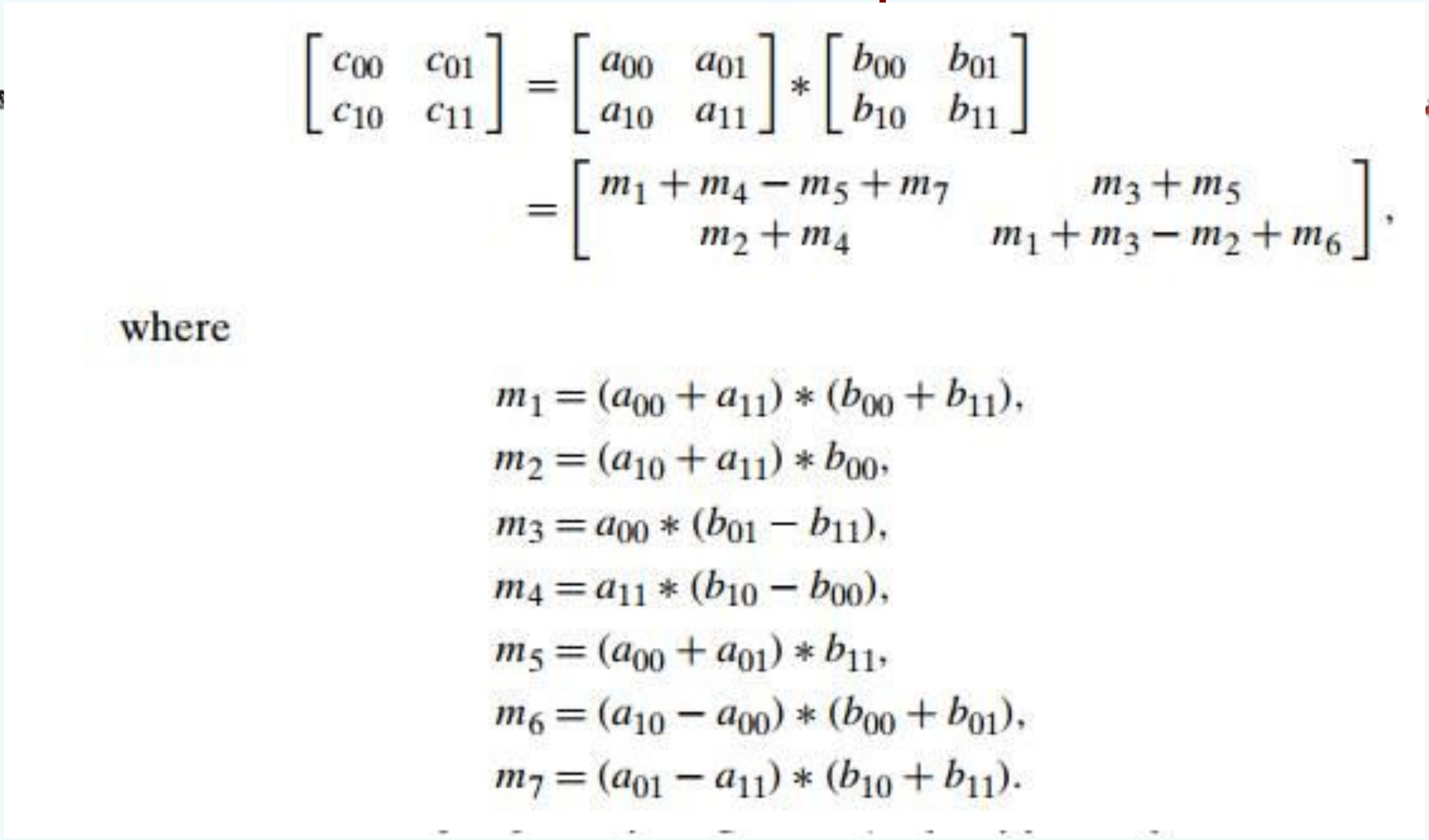
Strassen’s Matrix Multiplication Method

-In divide and conquer method, the main component for high time complexity is 8 recursive calls.

- The idea of **Strassen’s method** is to reduce the number of recursive calls to 7.

- Strassen’s method is similar to simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown, but in Strassen’s method, the four sub-matrices of result are calculated using following formulae.

Strassen's Matrix Multiplication Method



$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

where

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$
$$m_2 = (a_{10} + a_{11}) * b_{00},$$
$$m_3 = a_{00} * (b_{01} - b_{11}),$$
$$m_4 = a_{11} * (b_{10} - b_{00}),$$
$$m_5 = (a_{00} + a_{01}) * b_{11},$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Thus, to multiply two 2 × 2 matrices, Strassen's algorithm makes ***seven multiplications and 18 additions/subtractions****, whereas the normal algorithm requires eight multiplications and four additions.*

Matrix Multiplication

One can multiply nxn matrices serially in time

$\Theta(n^{\log 7}) = O(n^{2.81})$ using Strassen's divide-and-conquer method.

We will use multithreading for a simpler divide-and-conquer algorithm.

Simple Divide-and-Conquer

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Then, we can write the matrix product as

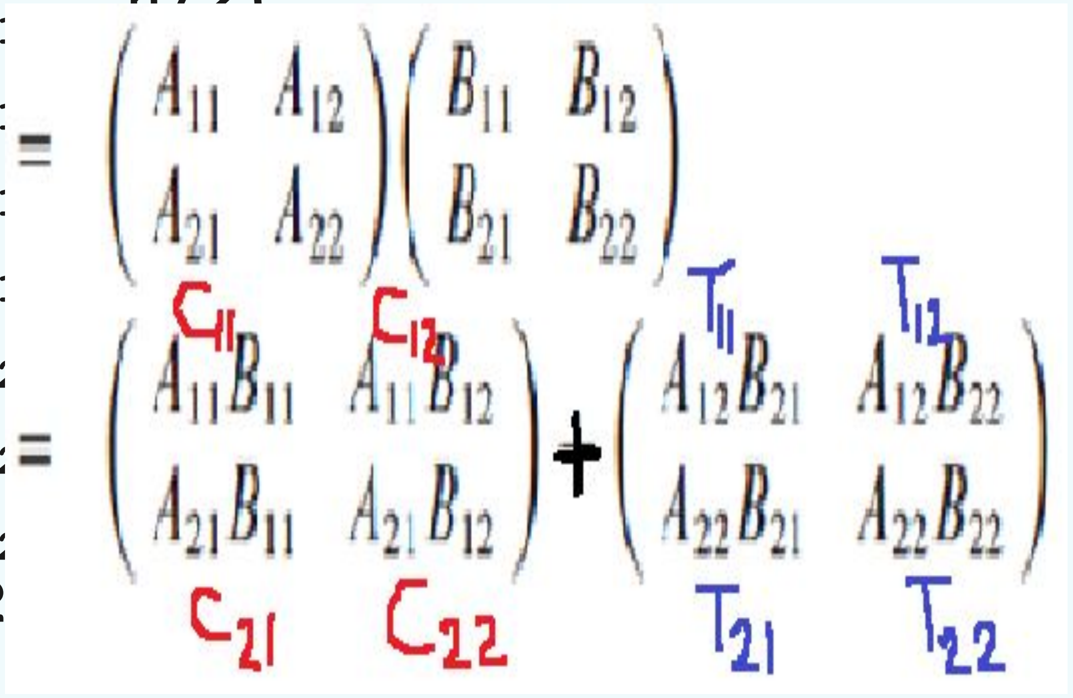
$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}.$$

To multiply two nxn matrices, we perform 8 matrix multiplications of n/2 x n/2 matrices and one addition of n x n matrices.

Matrix Multiplication

```
Matrix-Multiply(C, A, B, n):
    // Multiplies matrices A and B, storing the result in C.
    // n is power of 2 (for simplicity).
    if  n == 1:
      C[1, 1] = A[1, 1]  · B[1, 1]
    else:
      allocate a temporary matrix T[1...n, 1...n]
      partition A, B, C, and T into (n/2)x(n/2) submatrices
      spawn Matrix-Multiply($C_{11}$,$A_{11}$,B    n/2)
      spawn Matrix-Multiply($C_{12}$,$A_{11}$,B
      spawn Matrix-Multiply($C_{21}$,$A_{21}$,B
      spawn Matrix-Multiply($C_{22}$,$A_{21}$,B
      spawn Matrix-Multiply($T_{11}$,$A_{12}$,B
      spawn Matrix-Multiply($T_{12}$,$A_{12}$,B
      spawn Matrix-Multiply($T_{21}$,$A_{22}$,B
      Matrix-Multiply($T_{22}$,$A_{22}$,$B_{22}$, n/2
      sync
      Matrix-Add(C, T, n)
```

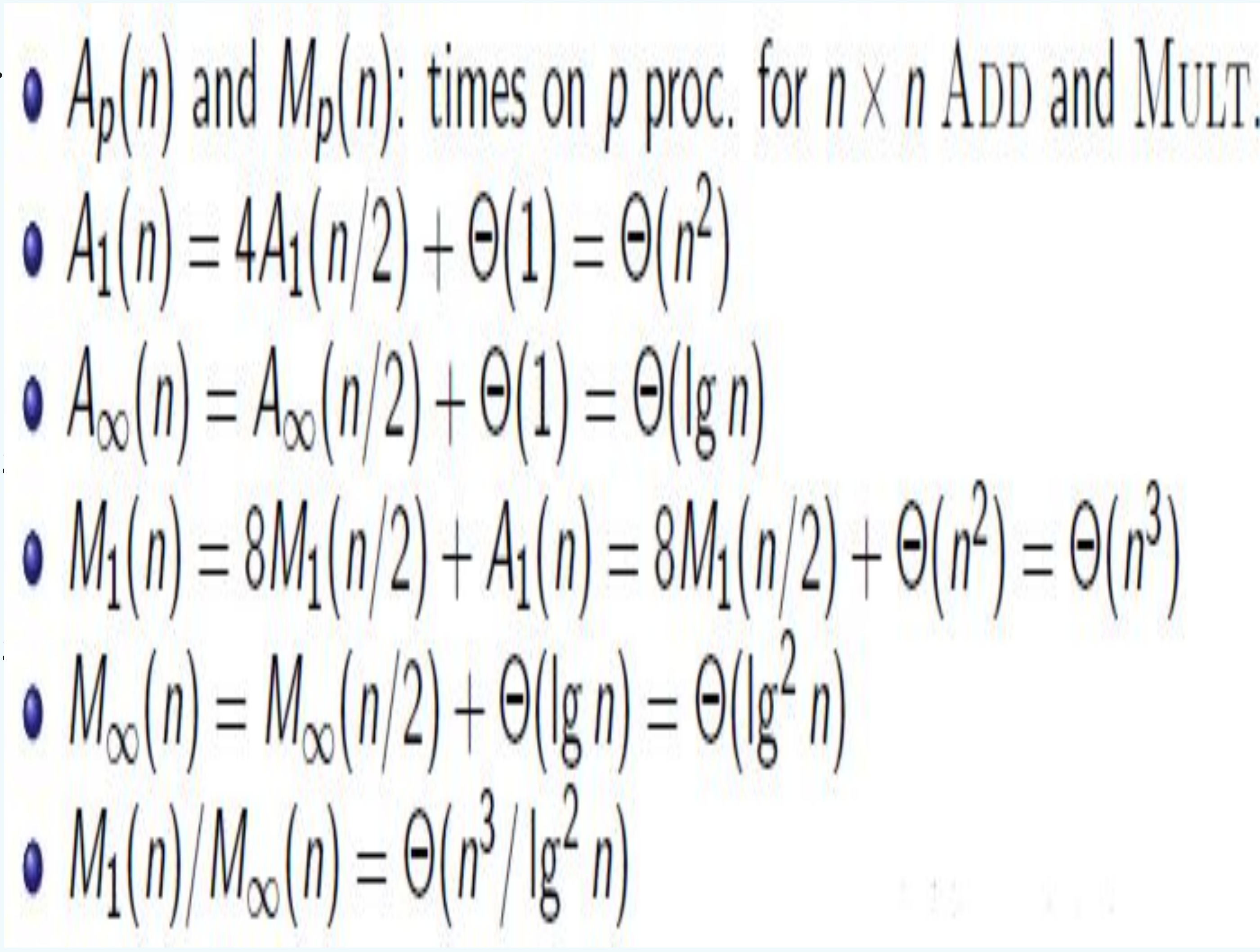


$$= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}$$

Addition of Matrices

```
Matrix-Add(C, T, n):
    // Adds matrices C and T in-place, producing C = C + T
    // n is power of 2 (for simplicity).
    if  n == 1:
      C[1, 1] = C[1, 1] + T[1
    else:
      partition C and T into
      spawn Matrix-Add($C_{11}$, $T_1$
      spawn Matrix-Add($C_{12}$, $T_1$
      spawn Matrix-Add($C_{21}$, $T_2$
      spawn Matrix-Add($C_{22}$, $T_2$
      sync
```



- $A_p(n)$ and $M_p(n)$: times on p proc. for $n \times n$ ADD and MULT
- $A_1(n) = 4A_1(n/2) + \Theta(1) = \Theta(n^2)$
- $A_\infty(n) = A_\infty(n/2) + \Theta(1) = \Theta(\lg n)$
- $M_1(n) = 8M_1(n/2) + A_1(n) = 8M_1(n/2) + \Theta(n^2) = \Theta(n^3)$
- $M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n)$
- $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$

Work of Matrix Multiplication

The work $T_1(n)$ of matrix multiplication satisfies the recurrence

$T_1(n) = 8\ T_1(n/2) + \theta(n^2) = \theta(n^3)$

by case 1 of the Master theorem.

The parallelism (max possible speed up). of a multithreaded computation is given by T_1 / T_∞

Span of Matrix Multiplication

The span $T_\infty(n)$ of matrix multiplication is determined by

- the span for partitioning $\theta(1)$

- the span of the parallel nested for loops at the end $\theta(\log n)$

- the maximum span of the 8 matrix multiplications

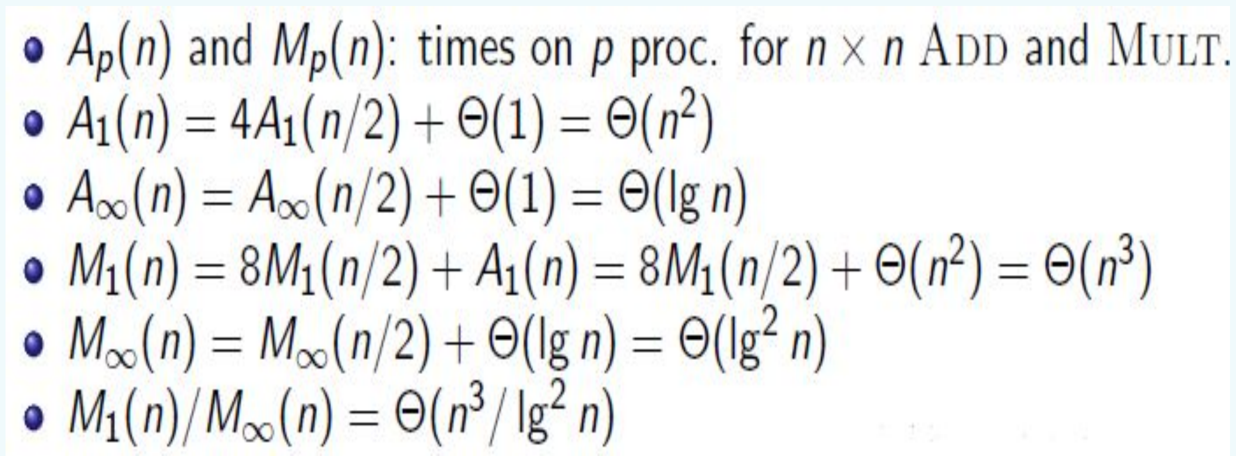
$T_\infty(n) = T_\infty(n/2) + \theta(\log n)$

Solving this recurrence we get

$T_\infty(n) = \theta((\log n)^2)$

The parallelism of matrix multiplication is given by

$T_1(n) / T_\infty(n) = \theta(n^3 / (\log n)^2)$



- $A_p(n)$ and $M_p(n)$: times on p proc. for $n \times n$ ADD and MULT.
- $A_1(n) = 4A_1(n/2) + \Theta(1) = \Theta(n^2)$
- $A_\infty(n) = A_\infty(n/2) + \Theta(1) = \Theta(\lg n)$
- $M_1(n) = 8M_1(n/2) + A_1(n) = 8M_1(n/2) + \Theta(n^2) = \Theta(n^3)$
- $M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n)$
- $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$

Merge Sort

Merge Sort- Serial version

The procedure MERGE-SORT(A, p, r) sorts the elements in the subarray $A[p..r]$. If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p..r]$ into two subarrays: $A[p..q]$, containing $\lceil n/2 \rceil$ elements, and $A[q+1..r]$, containing $\lfloor n/2 \rfloor$ elements.[8]

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

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

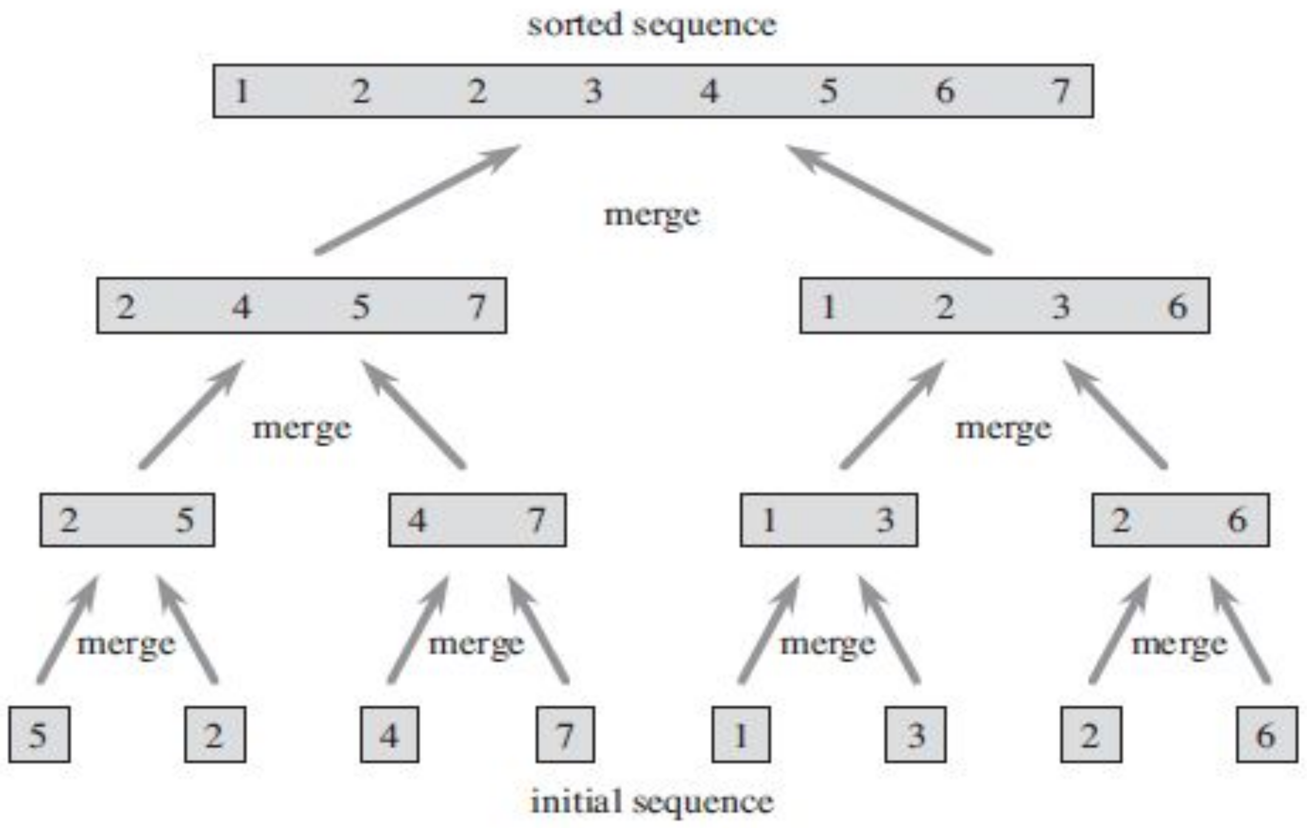
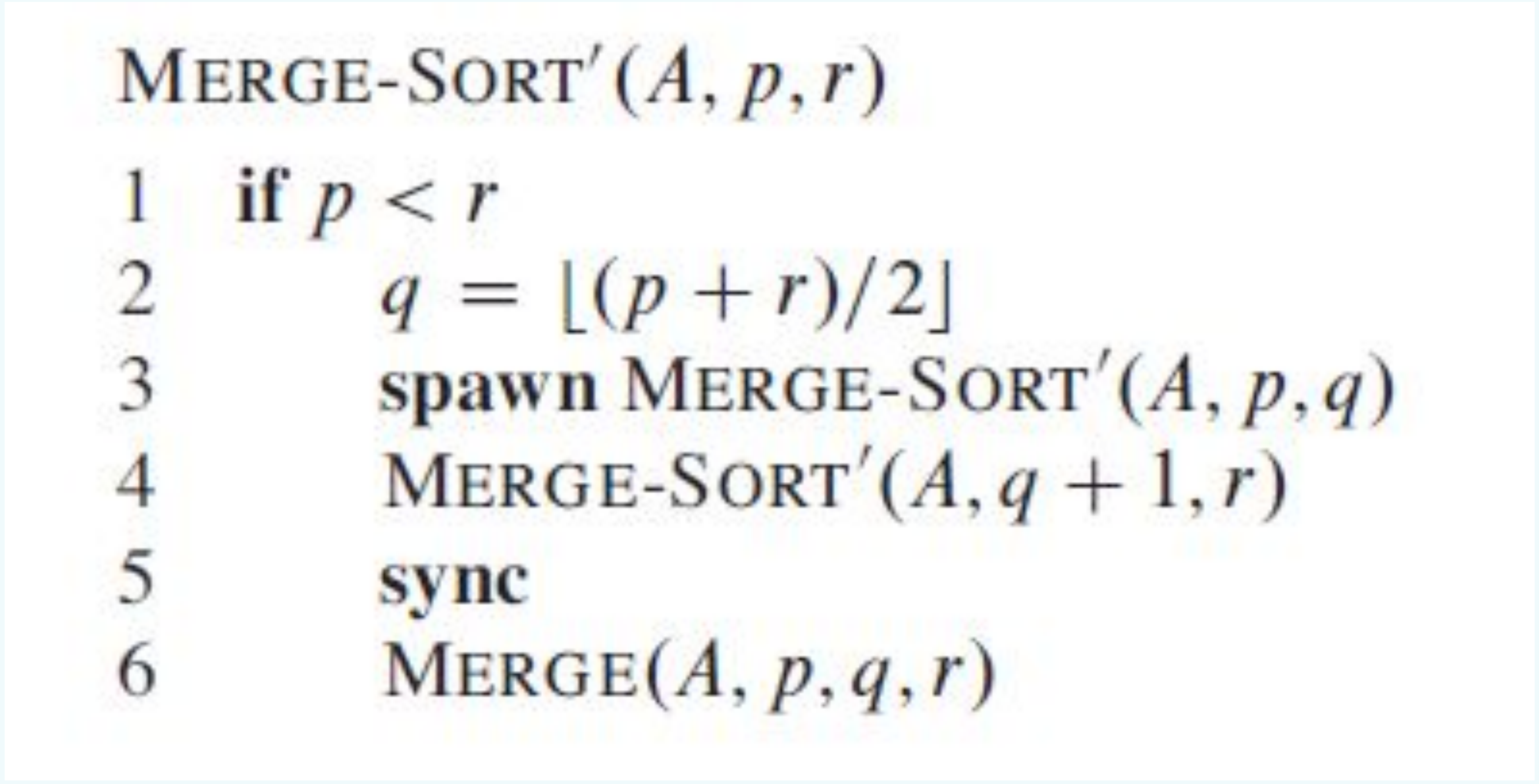


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Multithreaded Merge Sort



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

Multithreaded Merge Sort

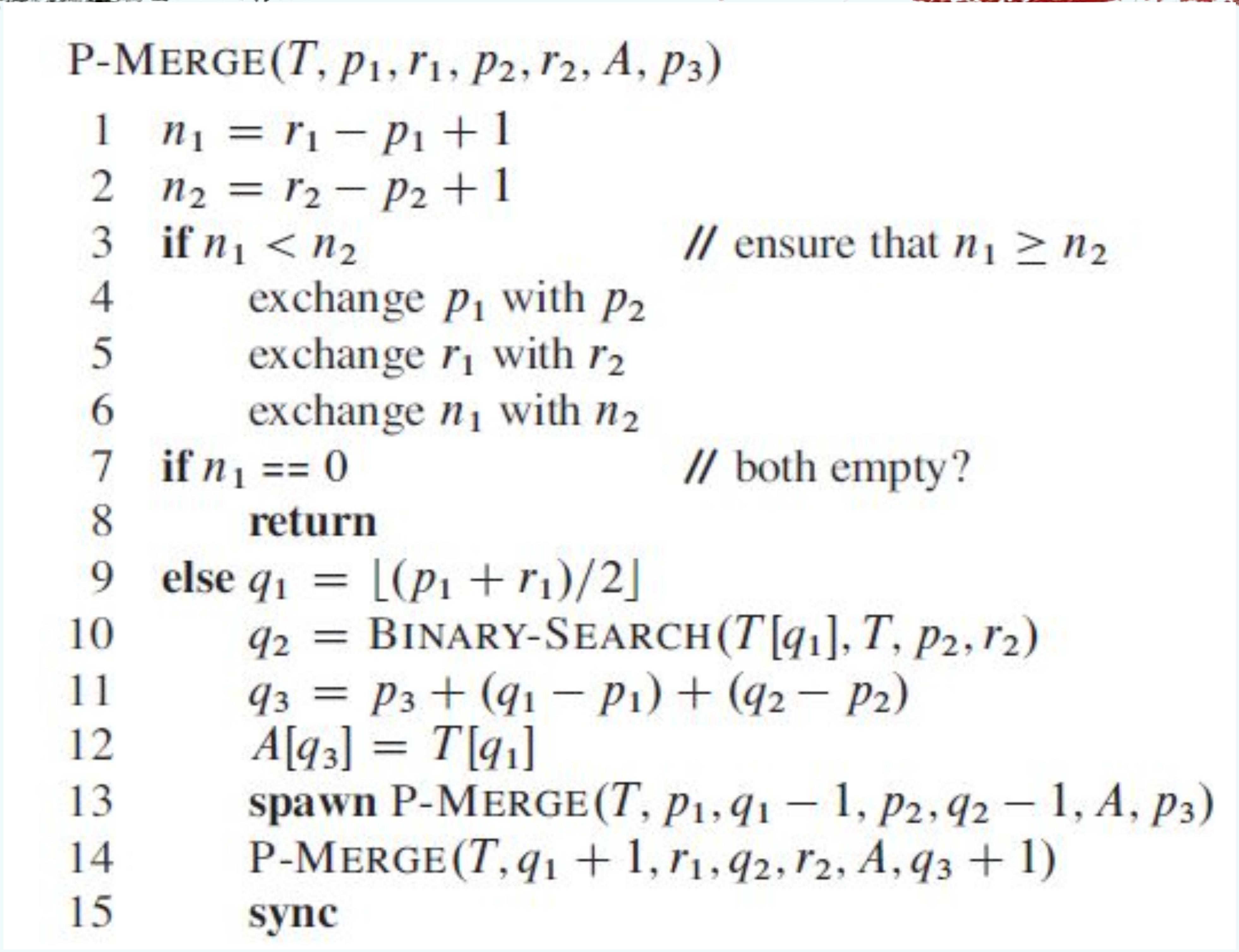
$$
\begin{aligned}
MS'_1(n) &= 2MS'_1(n/2) + \Theta(n) \\
&= \Theta(n \lg n) ,
\end{aligned}
$$

calls of MERGE-SORT′ can run in parallel, the span MS'_∞ is given by the recurrence

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

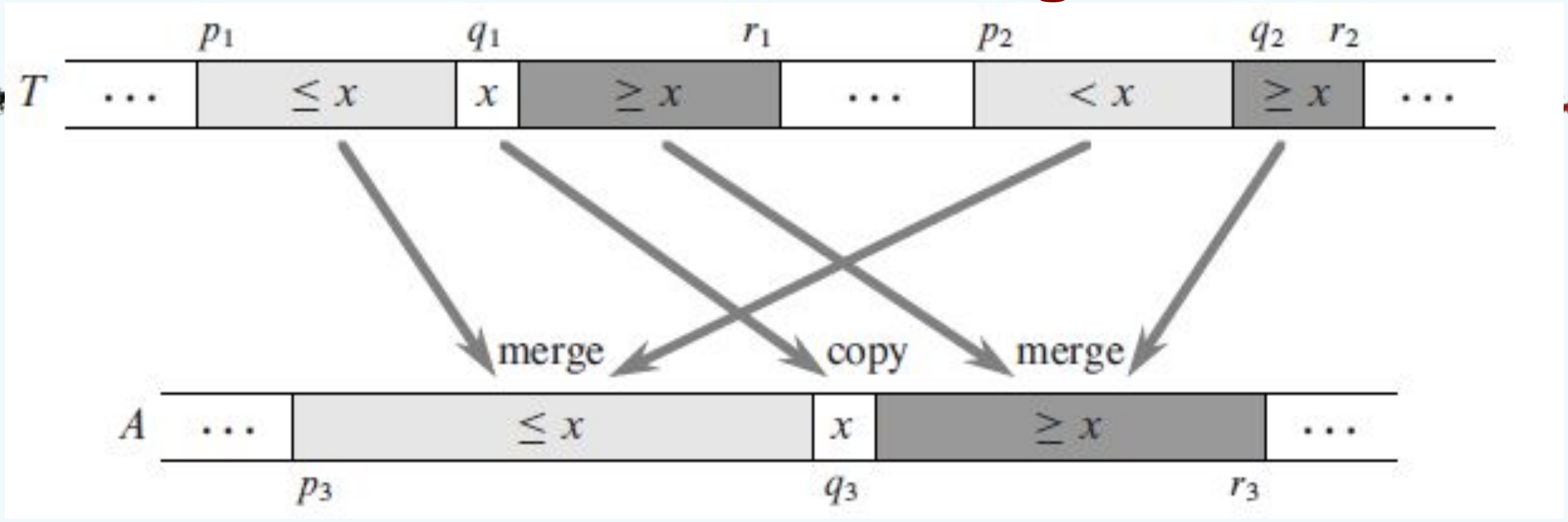
Thus, the parallelism of MERGE-SORT′ comes to $MS'_1(n)/MS'_\infty(n) = \Theta(\lg n)$,

Multithreaded Merge Sort



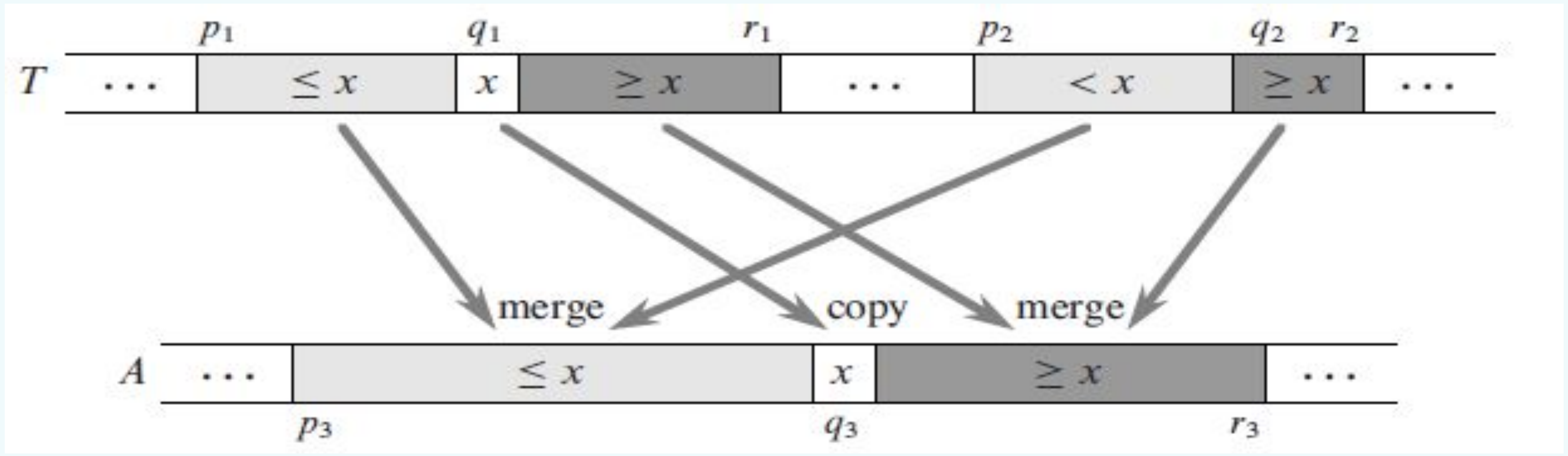
```
P-MERGE($T, p_1, r_1, p_2, r_2, A, p_3$)
$n_1 = r_1 - p_1 + 1$
$n_2 = r_2 - p_2 + 1$
if $n_1 < n_2$                      // ensure that $n_1 \geq n_2$
    exchange $p_1$ with $p_2$
    exchange $r_1$ with $r_2$
    exchange $n_1$ with $n_2$
if $n_1 == 0$                       // both empty?
    return
else $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$
    $q_2 =$ BINARY-SEARCH($T[q_1], T, p_2, r_2$)
    $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$
    $A[q_3] = T[q_1]$
    spawn P-MERGE($T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3$)
    P-MERGE($T, q_1 + 1, r_1, q_2, r_2, A, q_3 + 1$)
    sync
```

Multithreaded Merge Sort



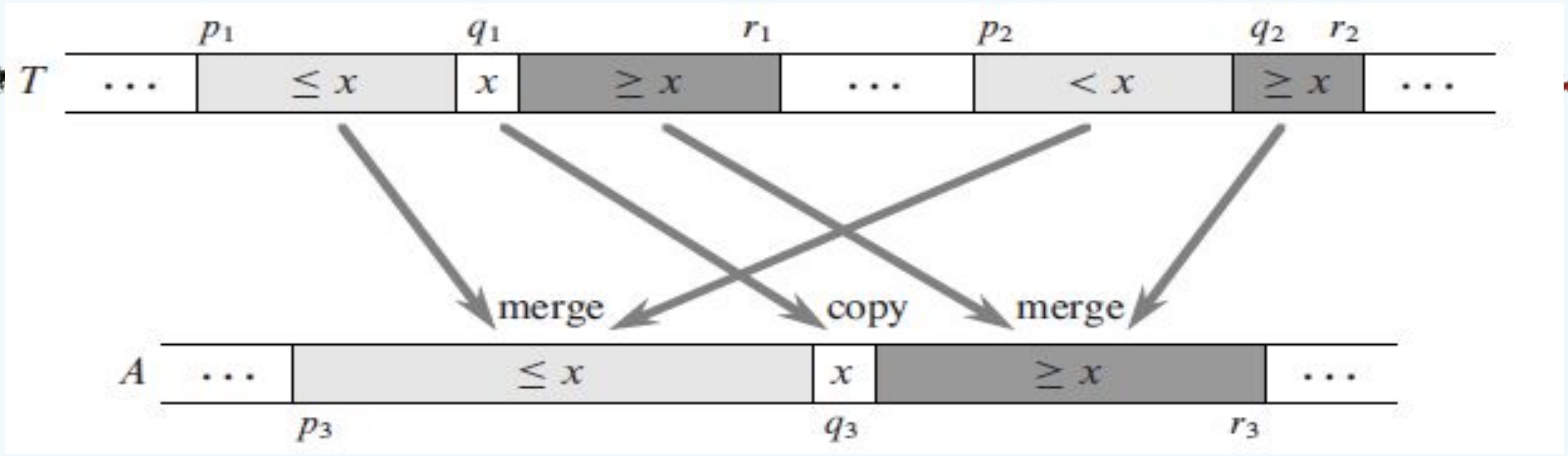
Our divide-and-conquer strategy for multithreaded merging, which is illustrated in Figure 27.6, operates on subarrays of an array T. Suppose that we are merging the two sorted subarrays $T[p_1 \ldots r_1]$ of length $n_1 = r_1 - p_1 + 1$ and $T[p_2 \ldots r_2]$ of length $n_2 = r_2 - p_2 + 1$ into another subarray $A[p_3 \ldots r_3]$, of length $n_3 = r_3 - p_3 + 1 = n_1 + n_2$. Without loss of generality, we make the simplifying assumption that $n_1 \geq n_2$.

Multithreaded Merge Sort



We first find the middle element $x = T[q_1]$ of the subarray $T[p_1 \ldots r_1]$, where $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$. Because the subarray is sorted, x is a median of $T[p_1 \ldots r_1]$: every element in $T[p_1 \ldots q_1 - 1]$ is no more than x, and every element in $T[q_1 + 1 \ldots r_1]$ is no less than x. We then use binary search to find the index q_2 in the subarray $T[p_2 \ldots r_2]$ so that the subarray would still be sorted if we inserted x between $T[q_2 - 1]$ and $T[q_2]$.

Multithreaded Merge Sort



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

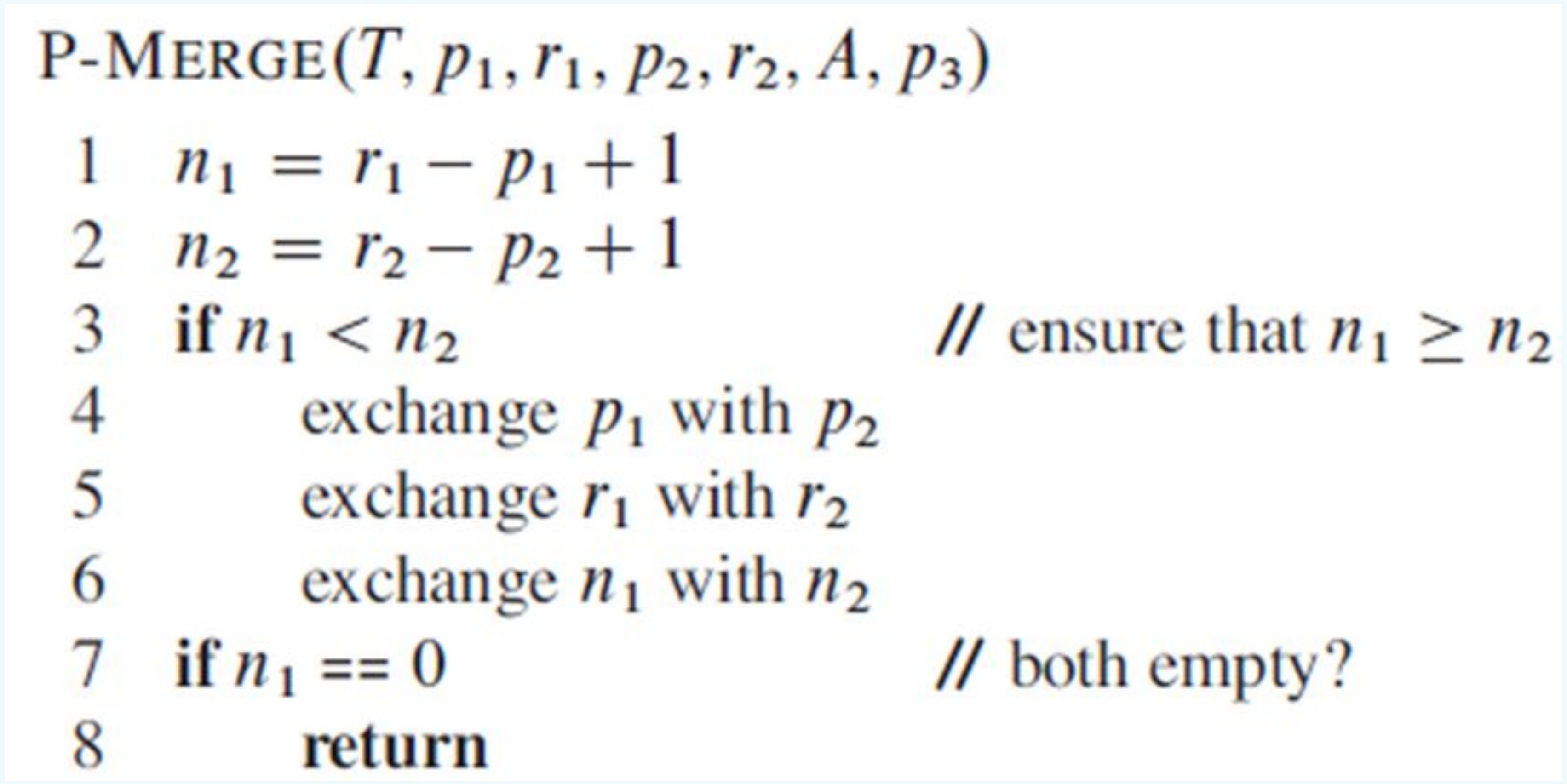
1. Set $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$.
2. Copy x into $A[q_3]$.
3. Recursively merge $T[p_1 \ldots q_1 - 1]$ with $T[p_2 \ldots q_2 - 1]$, and place the result into the subarray $A[p_3 \ldots q_3 - 1]$.
4. Recursively merge $T[q_1 + 1 \ldots r_1]$ with $T[q_2 \ldots r_2]$, and place the result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

Multithreaded Merge Sort

P-MERGE procedure assumes that the two subarrays to be merged lie within the same array.

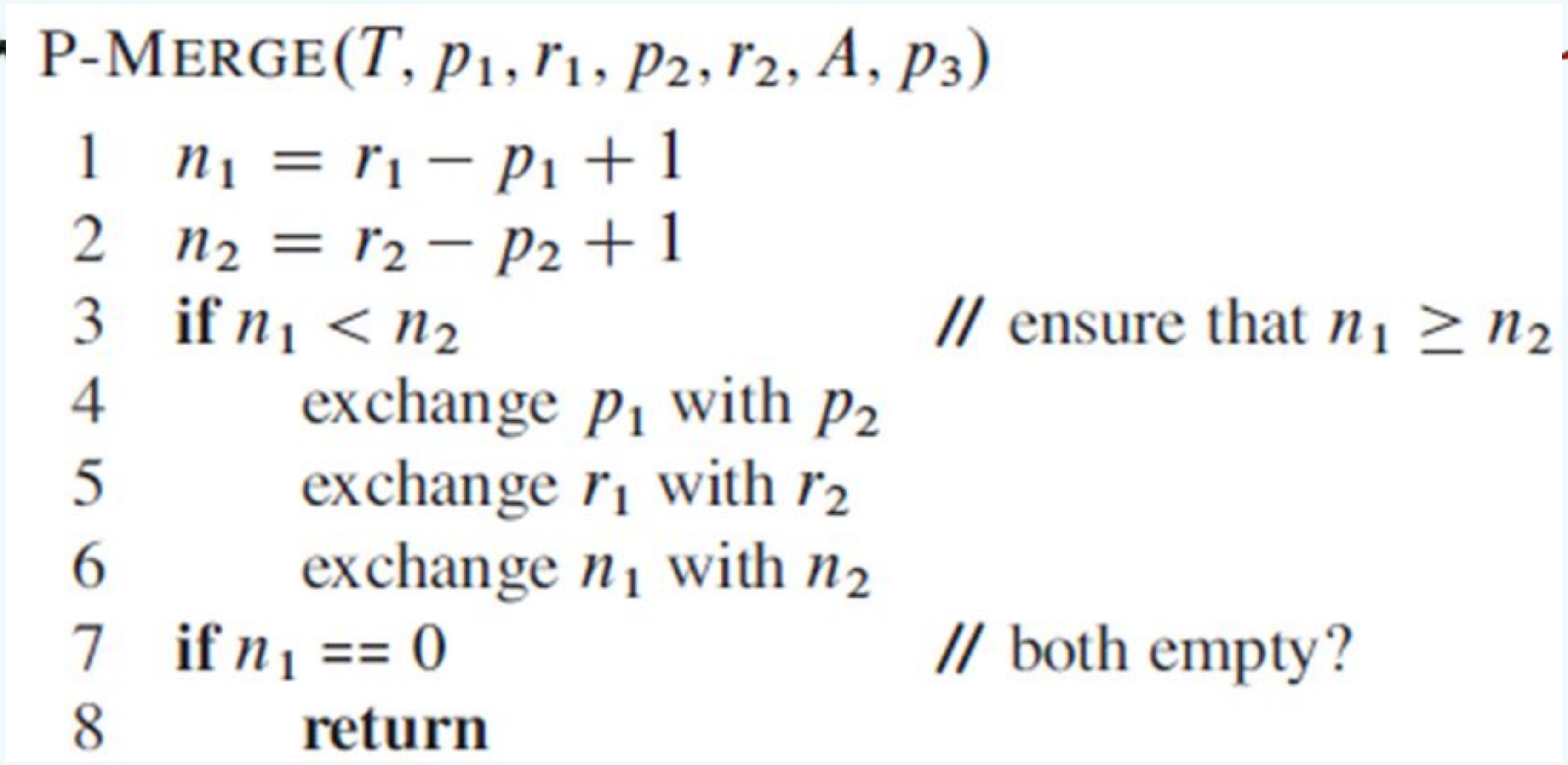
P-MERGE takes as an argument an output subarray A into which the merged values should be stored.

The call P-MERGE(T, p1, r1, p2, r2,A, p3) merges the sorted subarrays T[p1 ..r1] and T[p2 .. r2] into the subarray A[p3 .. r3], where r3=p3 +(r1 -p1 + 1) +(r2- p2+ 1)- 1 = p3+(r1- p1)+(r2 - p2)+ 1 and is not provided as an input.

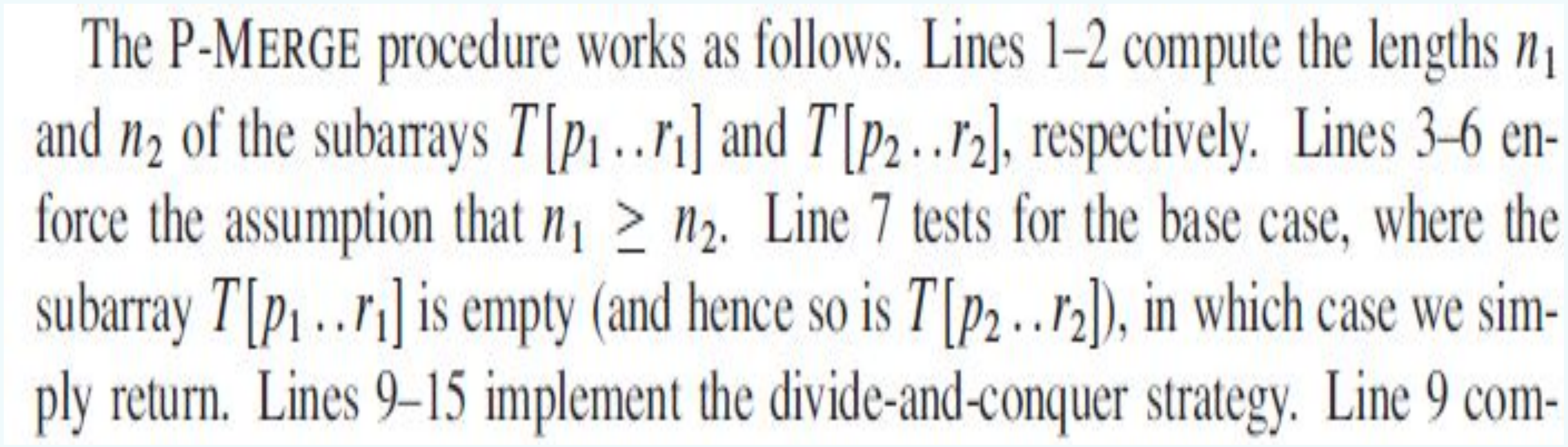


```
P-MERGE($T, p_1, r_1, p_2, r_2, A, p_3$)
$n_1 = r_1 - p_1 + 1$
$n_2 = r_2 - p_2 + 1$
if $n_1 < n_2$                    // ensure that $n_1 \geq n_2$
    exchange $p_1$ with $p_2$
    exchange $r_1$ with $r_2$
    exchange $n_1$ with $n_2$
if $n_1 == 0$                     // both empty?
    return
```

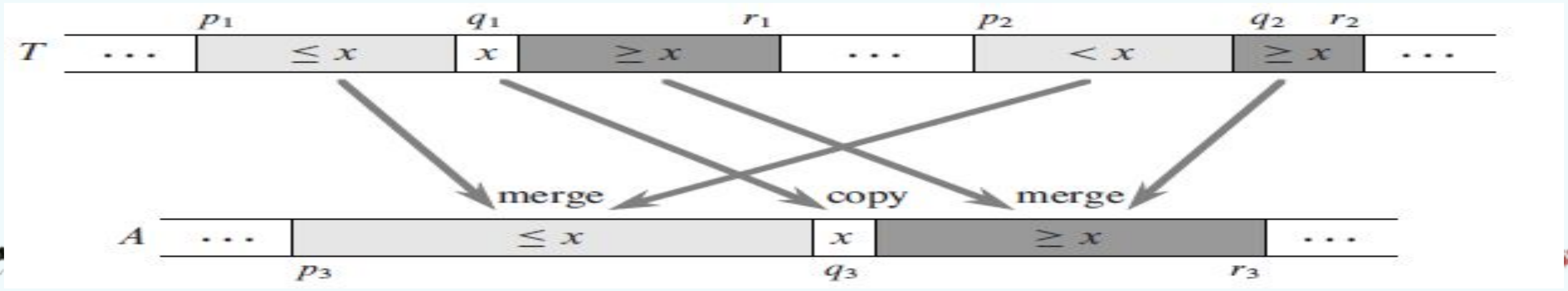
Multithreaded Merge Sort



```
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
```



The P-MERGE procedure works as follows. Lines 1–2 compute the lengths n_1 and n_2 of the subarrays $T[p_1 \,..\, r_1]$ and $T[p_2 \,..\, r_2]$, respectively. Lines 3–6 enforce the assumption that $n_1 \geq n_2$. Line 7 tests for the base case, where the subarray $T[p_1 \,..\, r_1]$ is empty (and hence so is $T[p_2 \,..\, r_2]$), in which case we simply return. Lines 9–15 implement the divide-and-conquer strategy. Line 9 com-



```
9   else $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$
10       $q_2 =$ 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
```

Lines 9–15 implement the divide-and-conquer strategy. Line 9 computes the midpoint of $T[p_1 \mathinner{..} r_1]$, and line 10 finds the point q_2 in $T[p_2 \mathinner{..} r_2]$ such that all elements in $T[p_2 \mathinner{..} q_2 - 1]$ are less than $T[q_1]$ (which corresponds to x) and all the elements in $T[q_2 \mathinner{..} p_2]$ are at least as large as $T[q_1]$. Line 11 computes the index q_3 of the element that divides the output subarray $A[p_3 \mathinner{..} r_3]$ into $A[p_3 \mathinner{..} q_3 - 1]$ and $A[q_3{+}1 \mathinner{..} r_3]$, and then line 12 copies $T[q_1]$ directly into $A[q_3]$.

Multithreaded Merge Sort

```
 9  else $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$
10      $q_2 = \textsc{Binary-Search}(T[q_1], T, p_2, r_2)$
11      $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$
12      $A[q_3] = T[q_1]$
13      spawn $\textsc{P-Merge}(T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3)$
14      $\textsc{P-Merge}(T, q_1 + 1, r_1, q_2, r_2, A, q_3 + 1)$
15      sync
```

Then, we recurse using nested parallelism. Line 13 spawns the first subproblem, while line 14 calls the second subproblem in parallel. The **sync** statement in line 15 ensures that the subproblems have completed before the procedure returns.

Multithreaded Merge Sort

```
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
```

Because 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 $T[p_1 \mathinner{..} r_1]$ with all n_2 elements of $T[p_2 \mathinner{..} r_2]$, and hence the number of elements involved in the call is

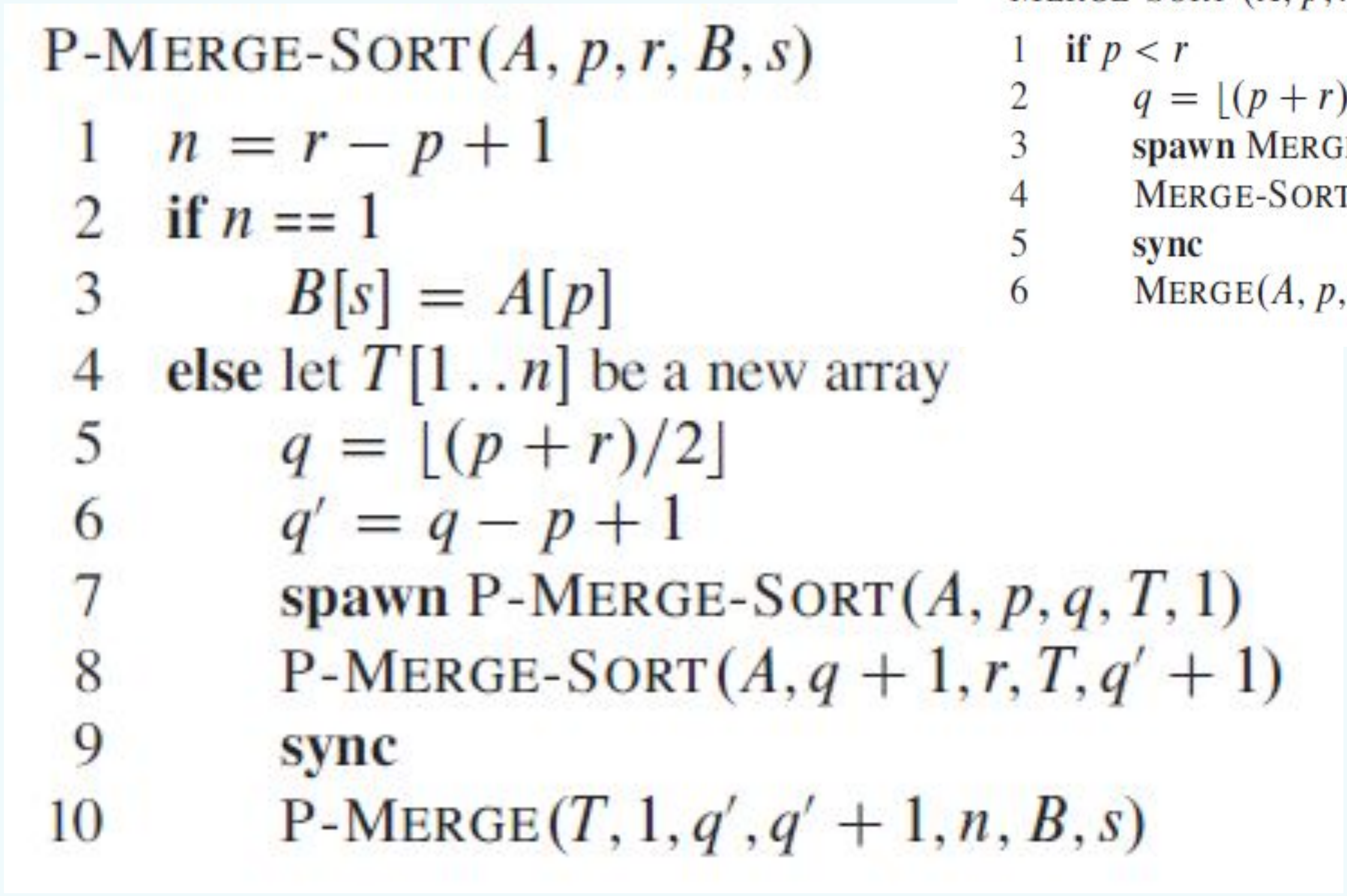
$$\begin{aligned} \lfloor n_1/2 \rfloor + n_2 &\leq n_1/2 + n_2/2 + n_2/2 \\ &= (n_1 + n_2)/2 + n_2/2 \\ &\leq n/2 + n/4 \\ &= 3n/4 . \end{aligned}$$

Adding in the $\Theta(\lg n)$ cost of the call to BINARY-SEARCH in line 10, we obtain the following recurrence for the worst-case span:

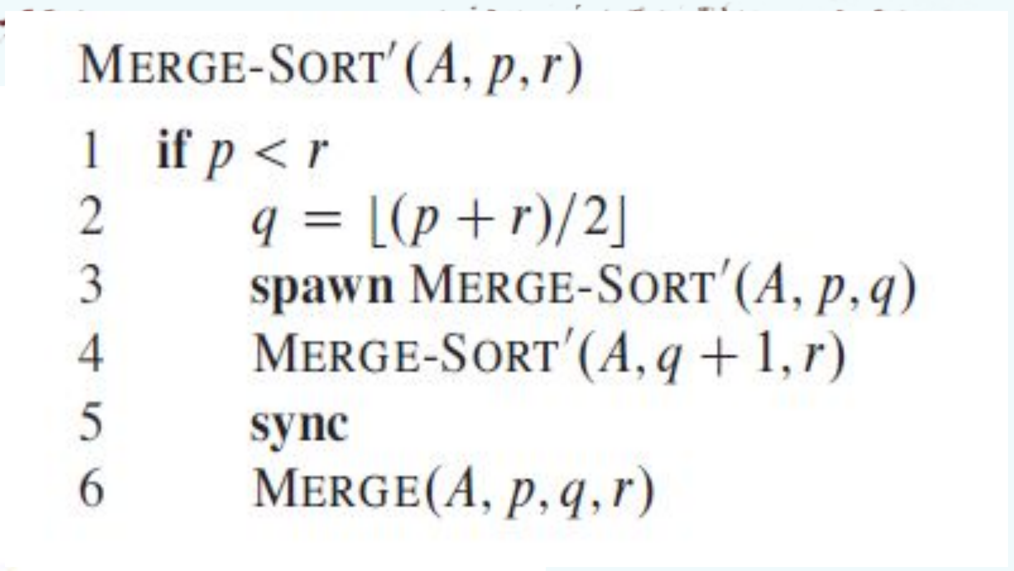
$$PM_\infty(n) = PM_\infty(3n/4) + \Theta(\lg n) .$$

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

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($T, 1, q', q' + 1, n, B, s$)
```



```
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($A, p, q, r$)
```

```
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($T, 1, q', q' + 1, n, B, s$)
```

After line 1 computes the number n of elements in the input subarray $A[p \,.\,.\, r]$, lines 2–3 handle the base case when the array has only 1 element. Lines 4–6 set up for the recursive spawn in line 7 and call in line 8, which operate in parallel. In particular, line 4 allocates a temporary array T with n elements to store the results of the recursive merge sorting. Line 5 calculates the index q of $A[p \,.\,.\, r]$ to divide the elements into the two subarrays $A[p \,.\,.\, q]$ and $A[q + 1 \,.\,.\, r]$ that will be sorted recursively, and line 6 goes on to compute the number q' of elements in the first subarray $A[p \,.\,.\, q]$, which line 8 uses to determine the starting index in T of where to store the sorted result of $A[q + 1 \,.\,.\, r]$. At that point, the spawn and recursive call are made, followed by the **sync** in line 9, which forces the procedure to wait until the spawned procedure is done. Finally, line 10 calls P-MERGE to merge the sorted subarrays, now in $T[1 \,.\,.\, q']$ and $T[q' + 1 \,.\,.\, n]$, into the output subarray $B[s \,.\,.\, s + r - p]$.

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)
```

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

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

Parallelism:

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

Acknowledgements

Andreas Klappenecker

https://www.youtube.com/watch?v=UaCX8Iy0ODA

https://www.youtube.com/watch?v=VD8hY7kWjdc

https://www.youtube.com/watch?v=7T-gjX24FR0

https://www.slideshare.net/AndresMendezVazquez/24-multithreaded-algorithms

- https://homes.luddy.indiana.edu/achauhan/Teaching/B403/LectureNotes/11-multithreaded.html
- https://catonmat.net/mit-introduction-to-algorithms-part-thirteen
- https://www.youtube.com/watch?v=iFrmLRr9ke0
- https://www.youtube.com/watch?v=GvtgV2NkdVg&t=31s
- https://www.youtube.com/watch?v=_XOZ2IiP2nw
- Analysis of merge sort : https://www.youtube.com/watch?v=0nlPxaC2lTw