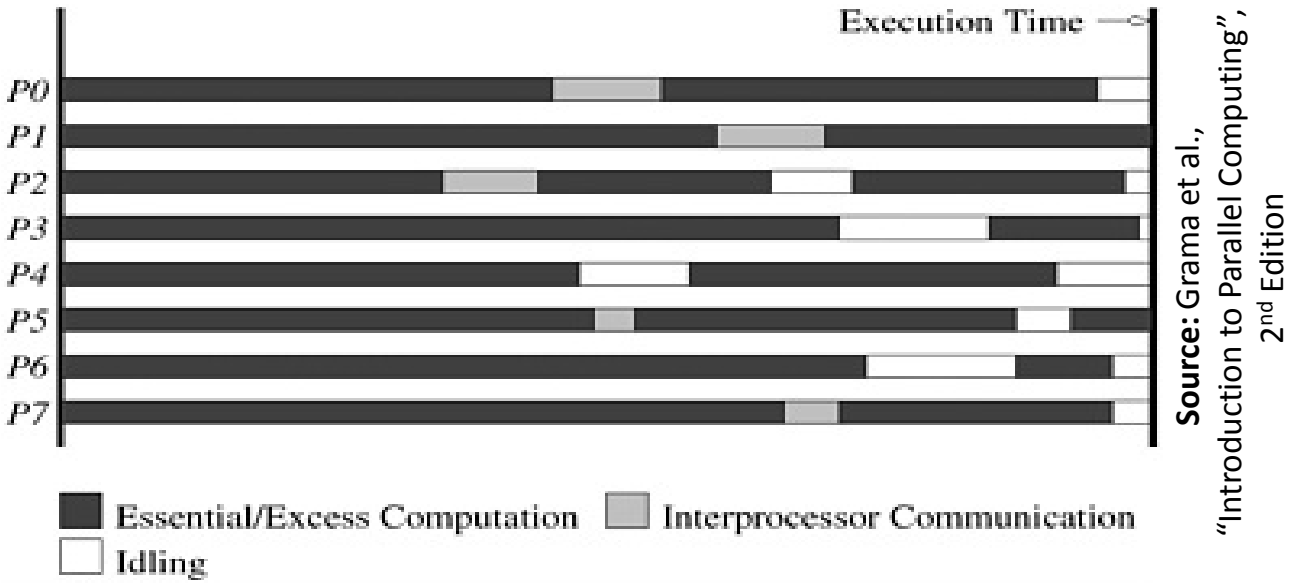
CSE 638: Advanced Algorithms

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

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2013

Parallel Execution Time & Overhead



Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

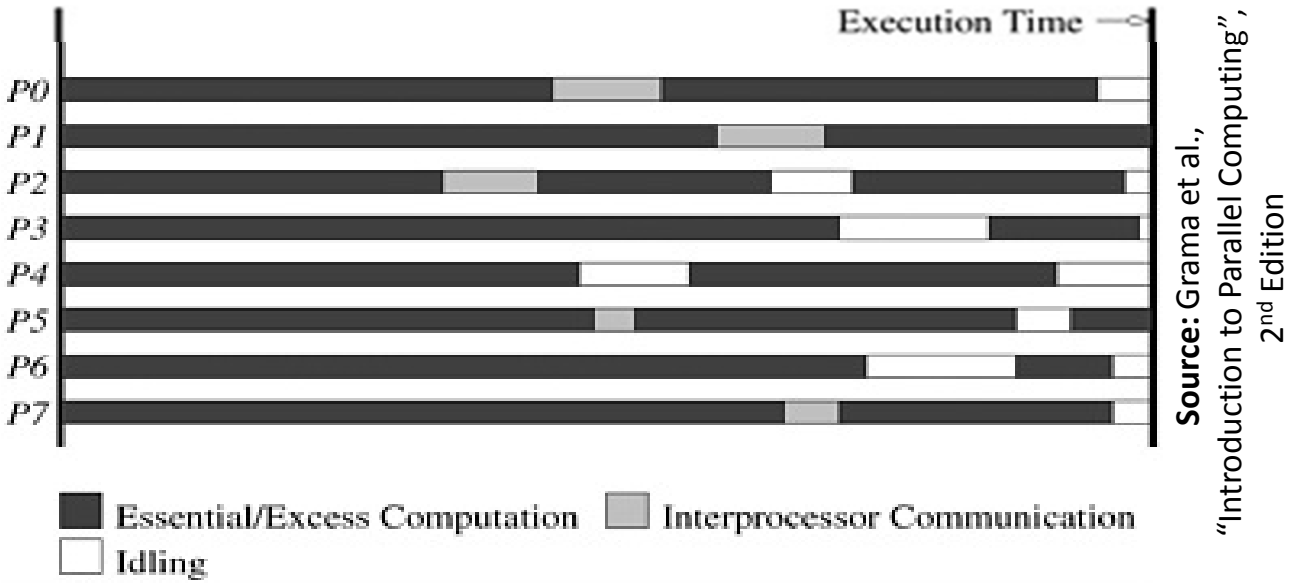
Parallel running time on p processing elements,

$$T_P = t_{end} - t_{start},$$

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

t_{end} = termination time of the processing element that finishes last

Parallel Execution Time & Overhead

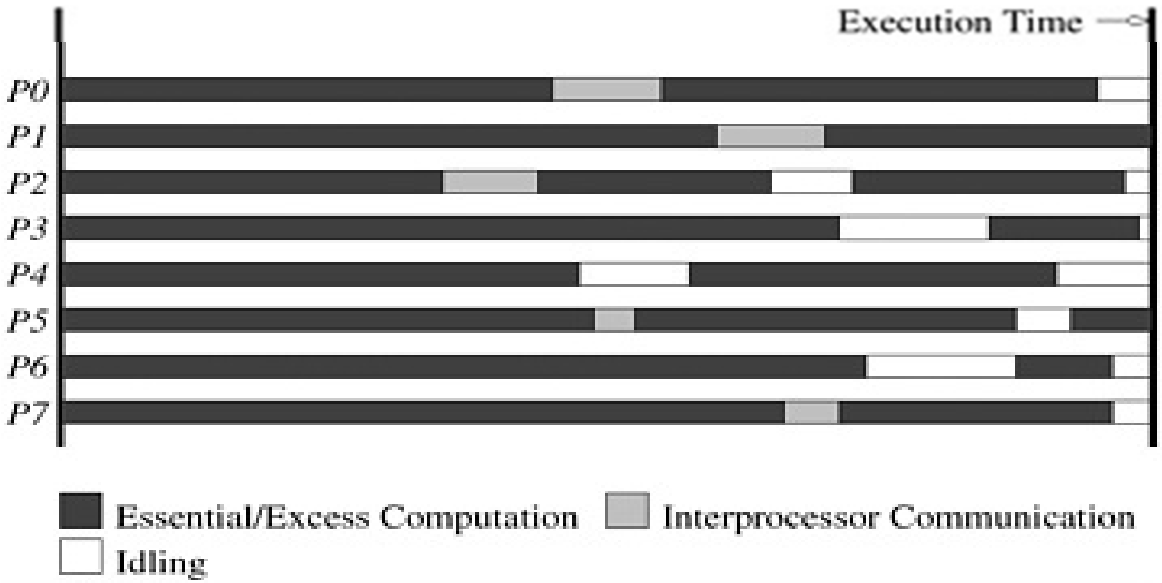


Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Sources of overhead (w.r.t. serial execution)

- Interprocess interaction
  - Interact and communicate data (e.g., intermediate results)
- Idling
  - Due to load imbalance, synchronization, presence of serial computation, etc.
- Excess computation
  - Fastest serial algorithm may be difficult/impossible to parallelize

Parallel Execution Time & Overhead



Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Overhead function or total parallel overhead,

$$T_O = pT_p - T,$$

where, p = number of processing elements

T = time spent doing useful work

(often execution time of the fastest serial algorithm)

Speedup

Let T_p = running time using p identical processing elements

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

Theoretically, $S_p \leq p$ (why?)

Perfect or *linear* or *ideal* speedup if $S_p = p$

Speedup

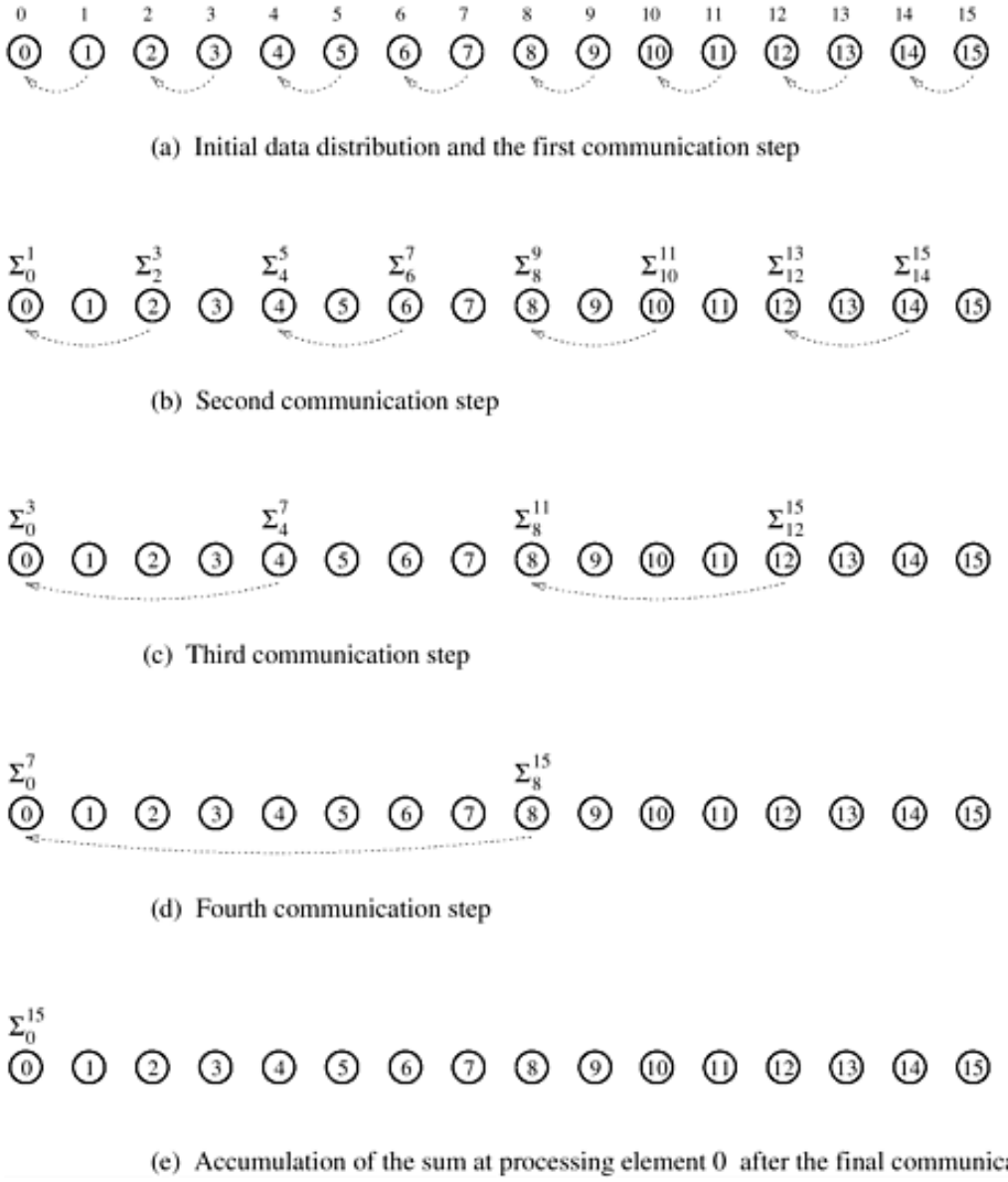
Consider adding n numbers using n identical processing elements.

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

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

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

Speedup not ideal.



Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Superlinear Speedup

Theoretically, $S_p \leq p$

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

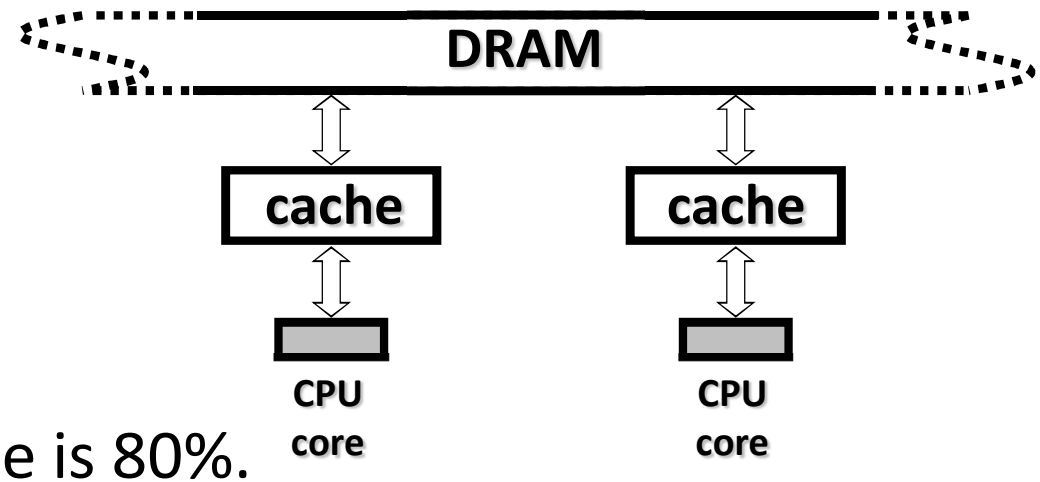
Reasons for superlinear speedup

- Cache effects
- Exploratory decomposition

Superlinear Speedup (Cache Effects)

Let cache access latency = 2 ns

DRAM access latency = 100 ns



Suppose we want solve a problem instance that executes *k* FLOPs.

With 1 Core: Suppose cache hit rate is 80%.

If the computation performs 1 FLOP/memory access, then each FLOP will take 2 × 0.8 + 100 × 0.2 = 21.6 ns to execute.

With 2 Cores: Cache hit rate will improve. (why?)

Suppose cache hit rate is now 90%.

Then each FLOP will take 2 × 0.9 + 100 × 0.1 = 11.8 ns to execute.

Since now each core will execute only *k* / 2 FLOPs,

Speedup, $S_2 = \frac{k \times 21.6}{(k/2) \times 11.8} \approx 3.66 > 2!$

Superlinear Speedup
(Due to Exploratory Decomposition)

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

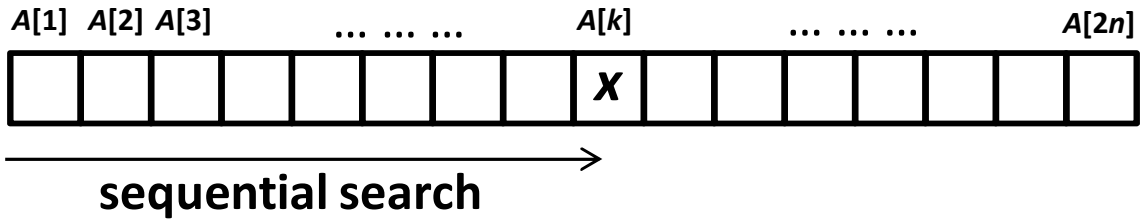
Suppose *x* is located at array location *k* > *n* and *k* is odd.

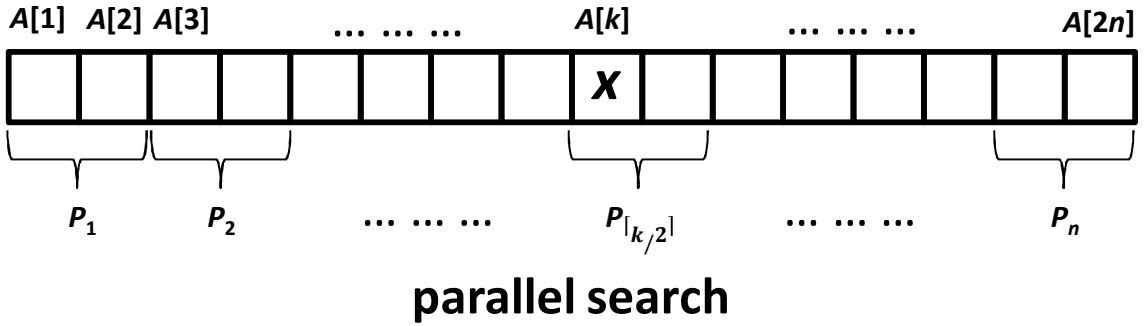
Serial runtime, $T_1 = k$

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

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

Speedup is superlinear!





Parallelism & Span Law

We defined, T_p = runtime on p identical processing elements

Then span, T_∞ = runtime on an infinite number of identical processing elements

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

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

Span Law

$$T_p \geq T_\infty$$

Work Law

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

On a Serial Computer: is given by T_1

On a Parallel Computer: is given by pT_p

Work Law

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

Work Optimality

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

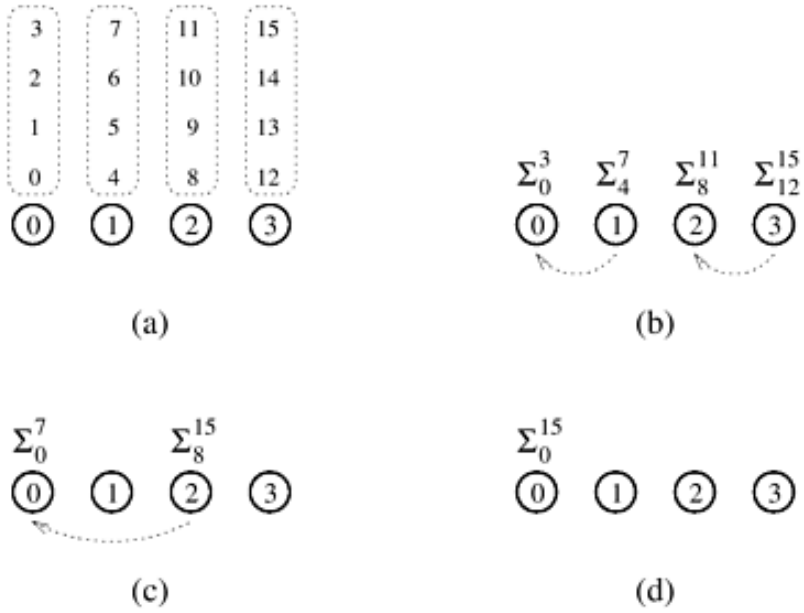
A parallel algorithm is *cost-optimal* or *work-optimal* provided

$$pT_p = \Theta(T_s)$$

Our algorithm for adding n numbers using n identical processing elements is clearly not work optimal.

Adding *n* Numbers Work-Optimality

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



Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Suppose we use p processing elements.

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

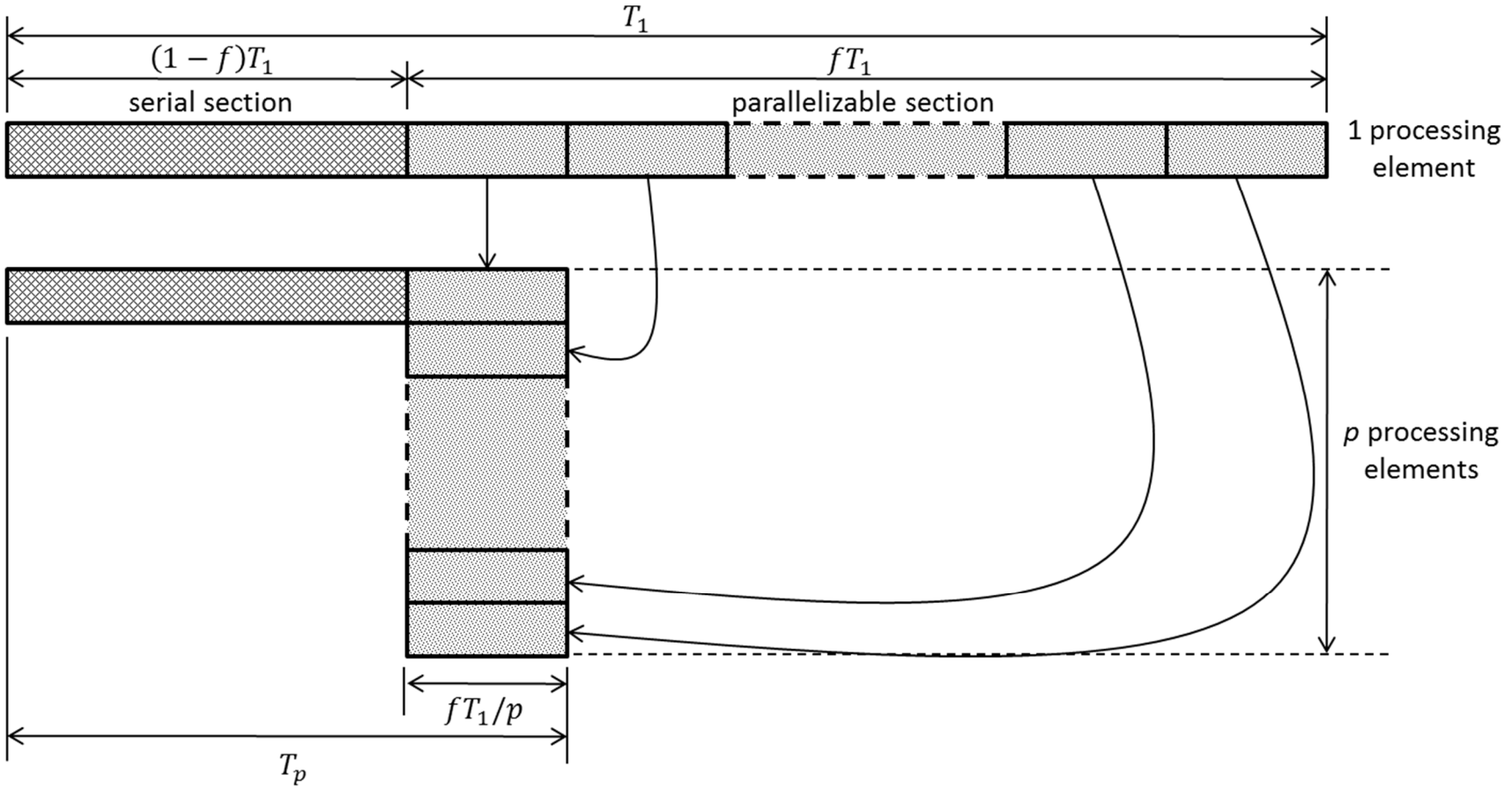
Then p processing elements adds these p partial sums in time $\Theta(\log p)$.

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

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

Scaling Laws

Scaling of Parallel Algorithms (Amdahl's Law)



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

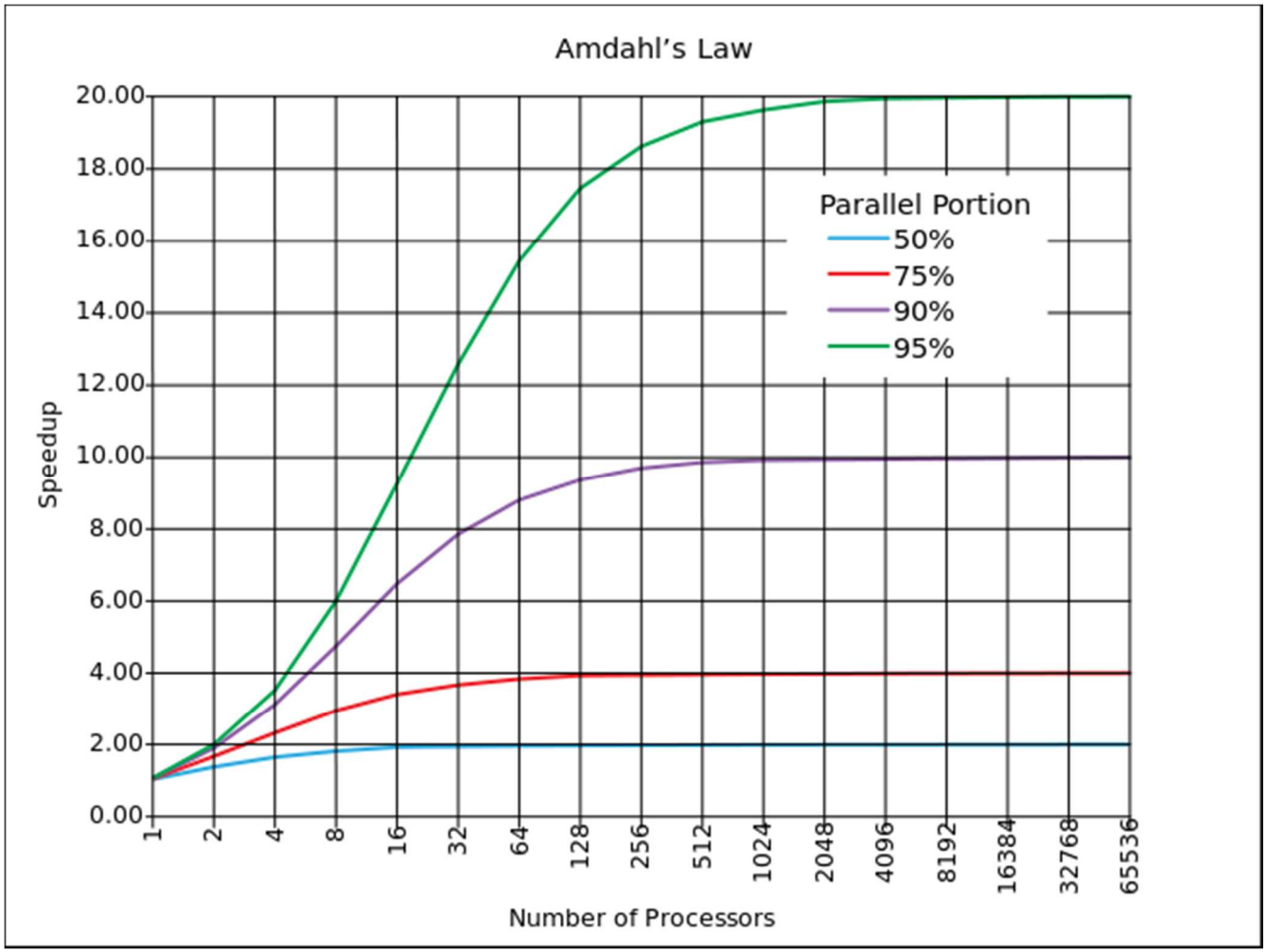
Then parallel running time, $T_p \geq (1-f)T_1 + f\frac{T_1}{p}$

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

Scaling of Parallel Algorithms (Amdahl's Law)

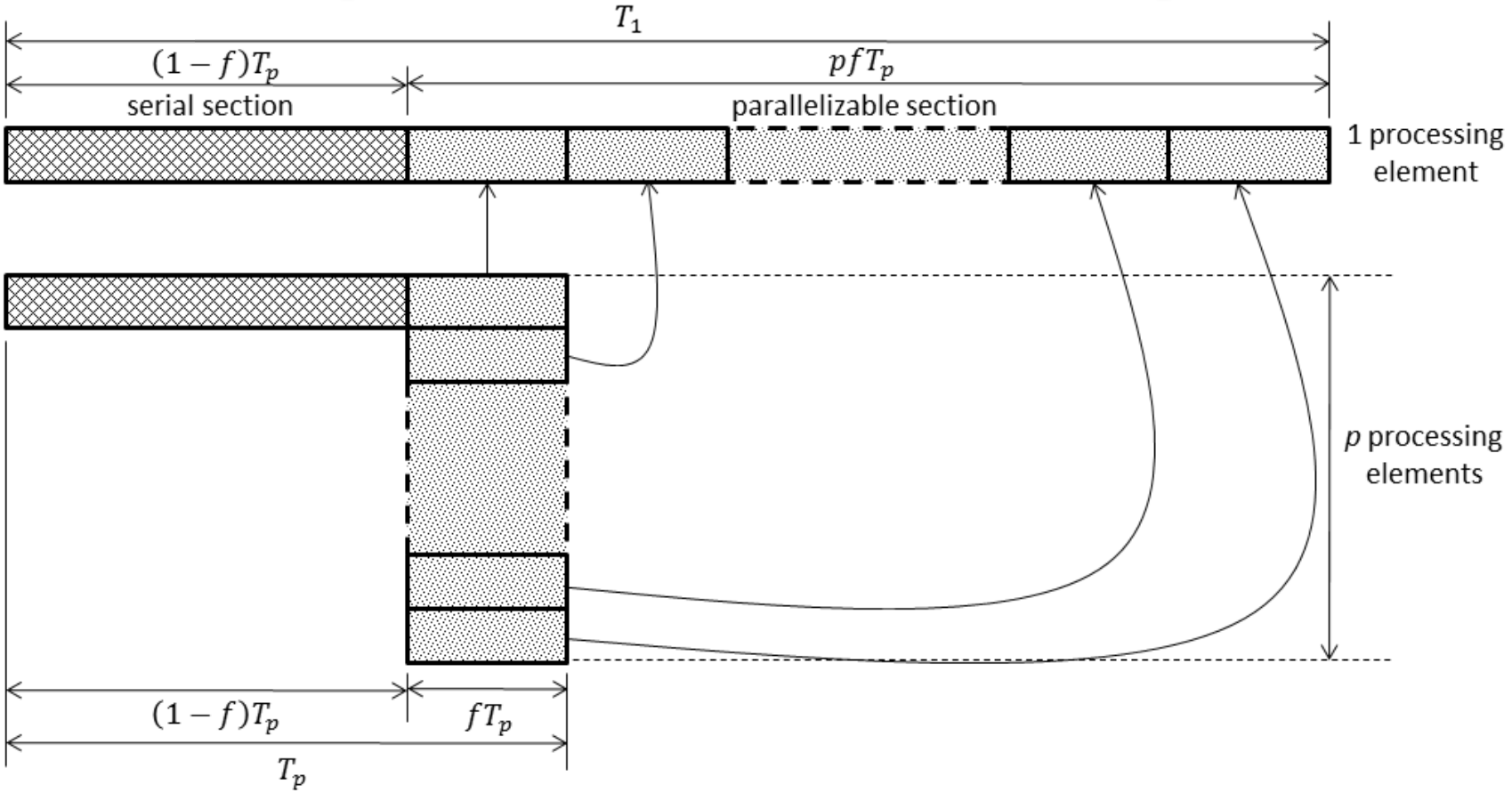
Suppose only a fraction f of a computation can be parallelized.

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



Source: Wikipedia

Scaling of Parallel Algorithms (Gustafson-Barsis' Law)



Suppose only a fraction f of a computation was parallelized.

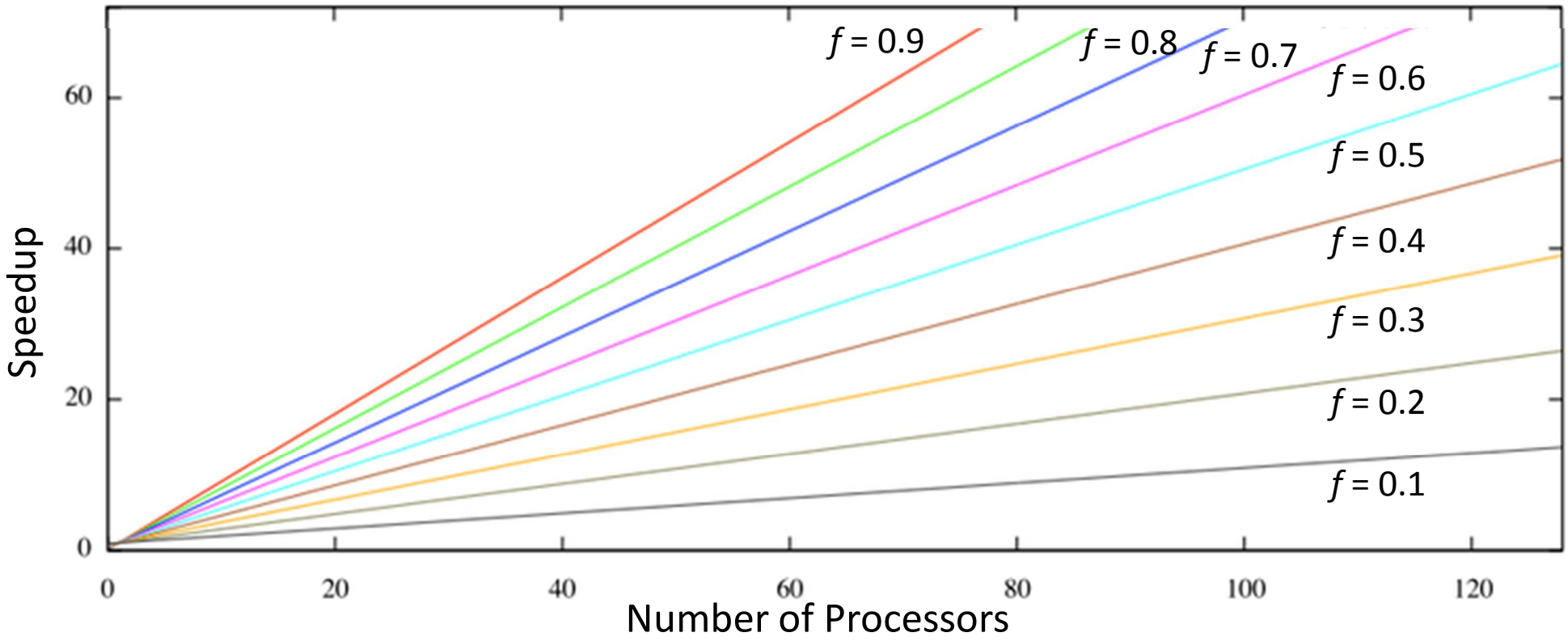
Then serial running time, $T_1 = (1-f)T_p + pfT_p$

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

Scaling of Parallel Algorithms (Gustafson-Barsis' Law)

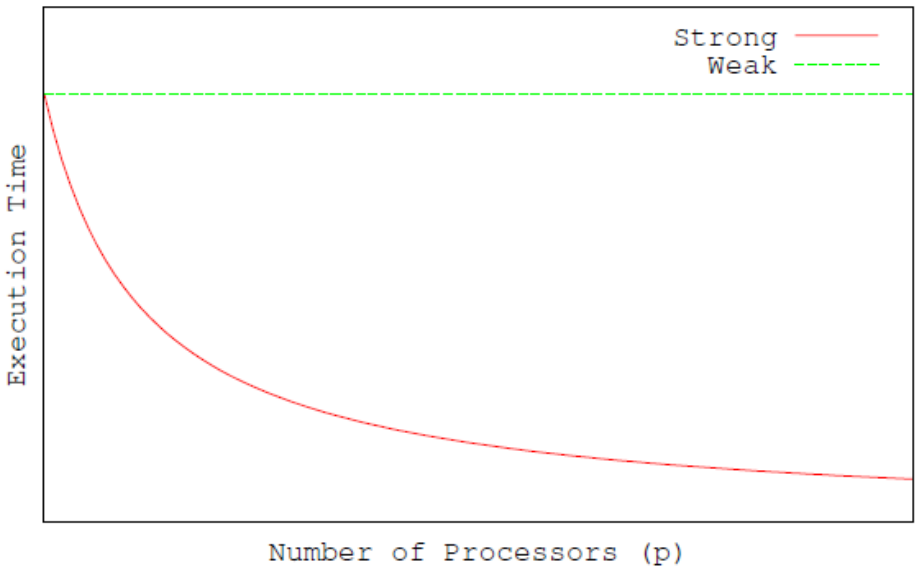
Suppose only a fraction f of a computation was parallelized.

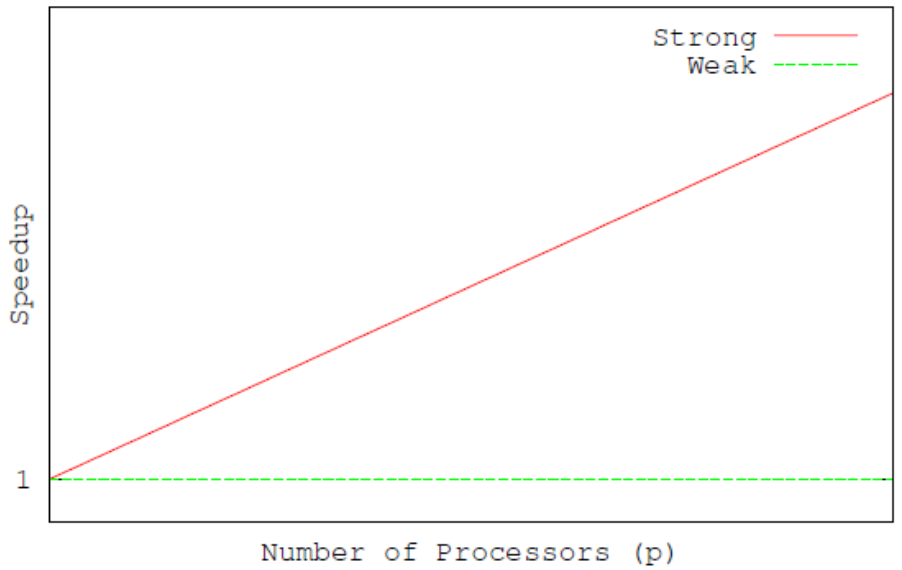
Speedup, $S_p = \frac{T}{T_p} \leq \frac{T_1}{T_p} = \frac{(1-f)T_p + pfT_p}{T_p} = 1 + (p-1)f$



Source: Wikipedia

Strong Scaling vs. Weak Scaling





Source: Martha Kim, Columbia University

Strong Scaling

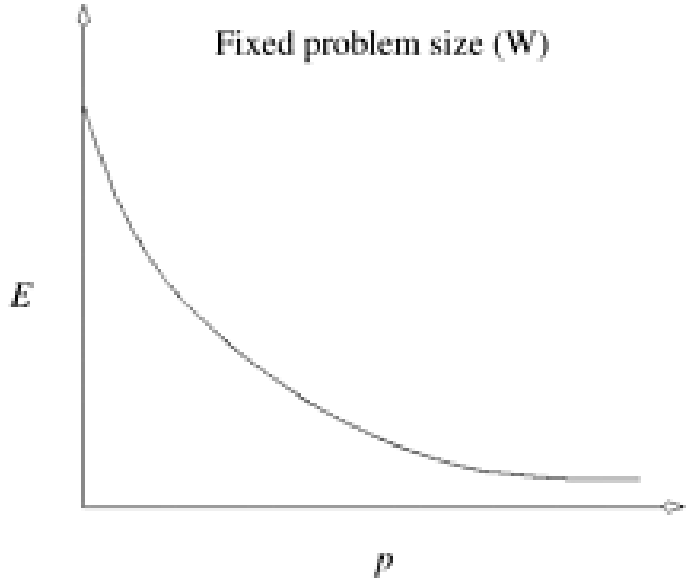
How T_p (or S_p) varies with p when the problem size is fixed.

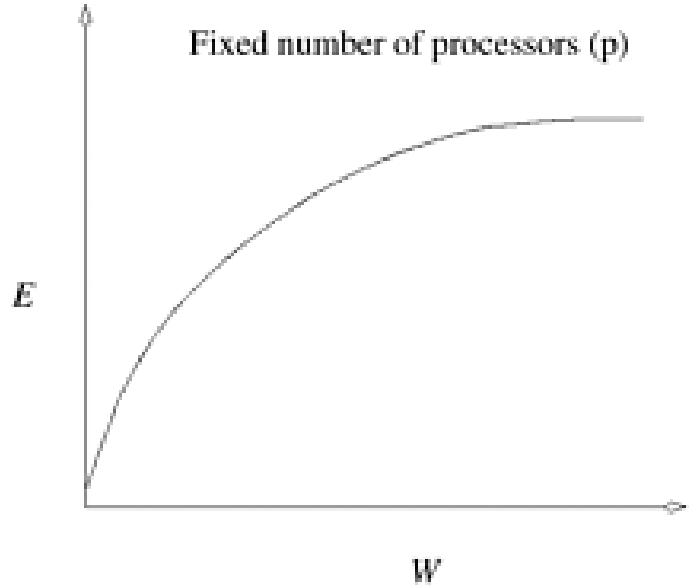
Weak Scaling

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

Scalable Parallel Algorithms

Efficiency, $E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$





Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

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.

Scalable Parallel Algorithms

In order to keep E_p fixed at a constant *k*, we need

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

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

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

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

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

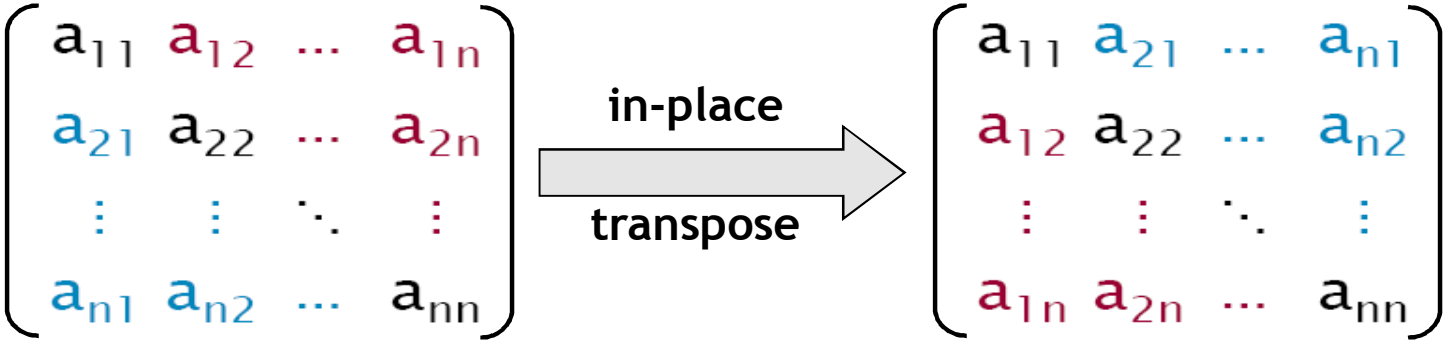
n	*p* = 1	*p* = 4	*p* = 8	*p* = 16	*p* = 32
64	1.0	***0.80***	0.57	0.33	0.17
192	1.0	0.92	***0.80***	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	***0.80***	0.62

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

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Greedy Scheduling Theorem

Loop Parallelism (Data Parallelism)




```
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;
      }
```

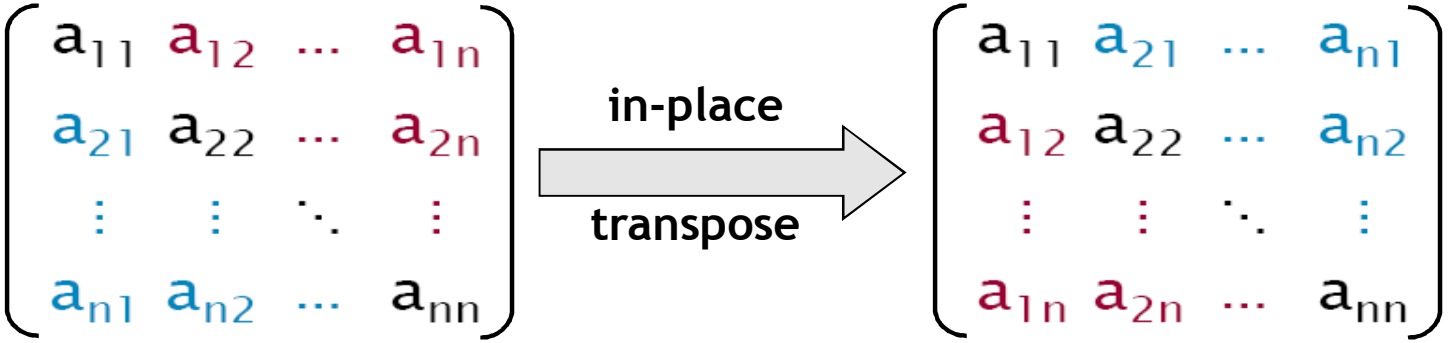
Allows all iterations of the loop to be executed in parallel.

Serial Code

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

Parallel Code

Loop Parallelism (Data Parallelism)




```
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.

Serial Code

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

Parallel Code

Nested Parallelism (Task Parallelism)

$$ {}^{n}C_{r} = {}^{n-1}C_{r-1} + {}^{n-1}C_{r} $$

```
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 );
}
```

Serial Code

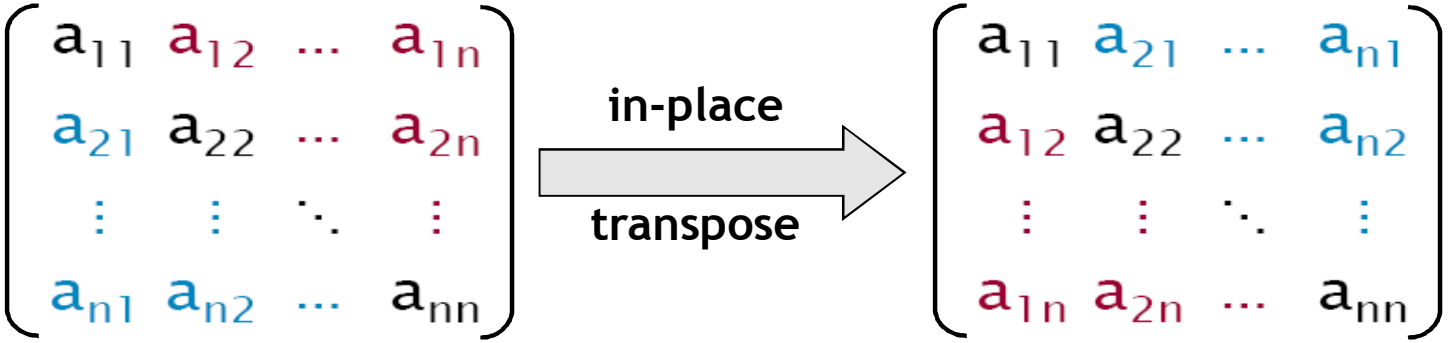
Grant permission to execute the called (spawned) function in parallel with the caller.

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

```
comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  sync;

  return ( x + y );
}
```

Parallel Code

Loop Parallelism (Data Parallelism)




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

Serial Code

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

Can be converted to spawns and syncs using recursive divide-and-conquer.

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

Parallel Code

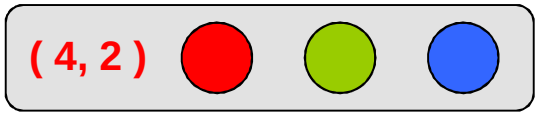
Parallel Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  sync;

  return ( x + y );
}
```

Parallel Execution Model

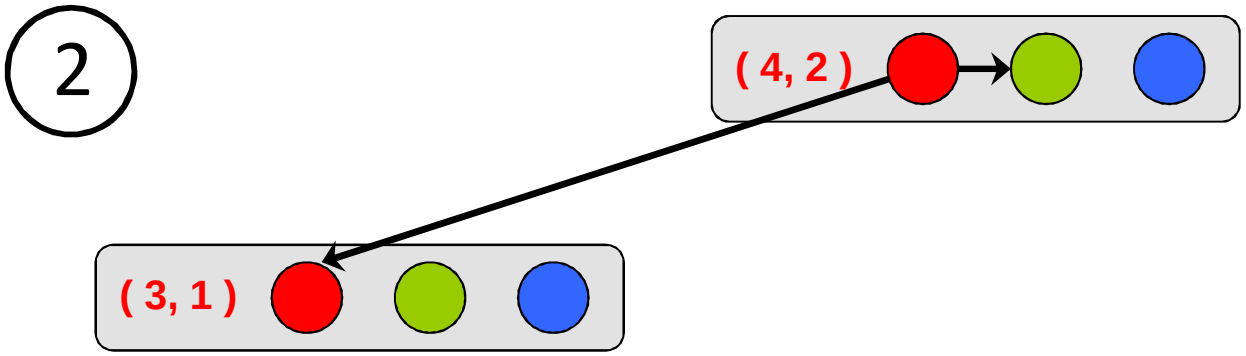
```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  sync;

  return ( x + y );
}
```

1



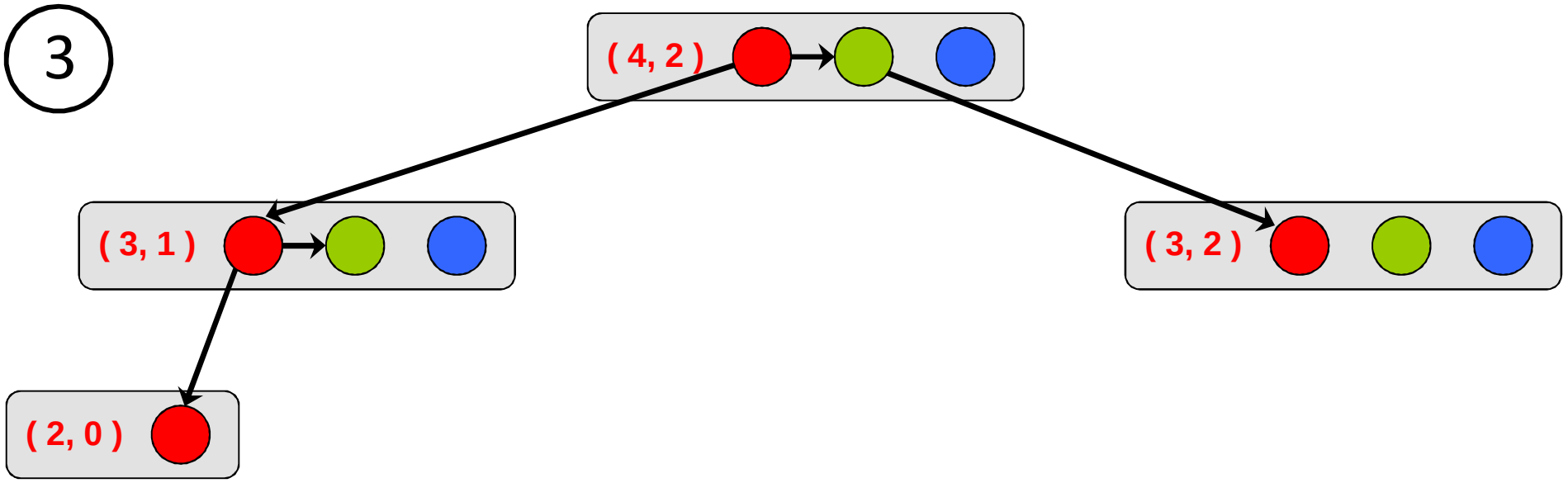
Parallel Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  sync;

  return ( x + y );
}
```



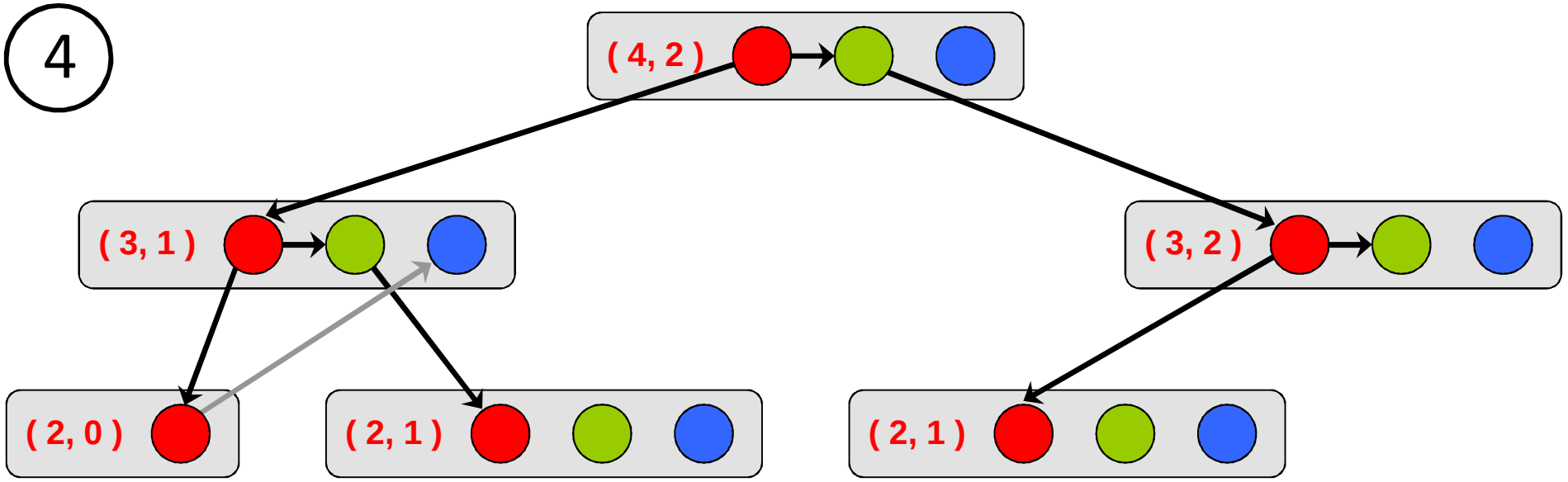
Parallel Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  sync;

  return ( x + y );
}
```



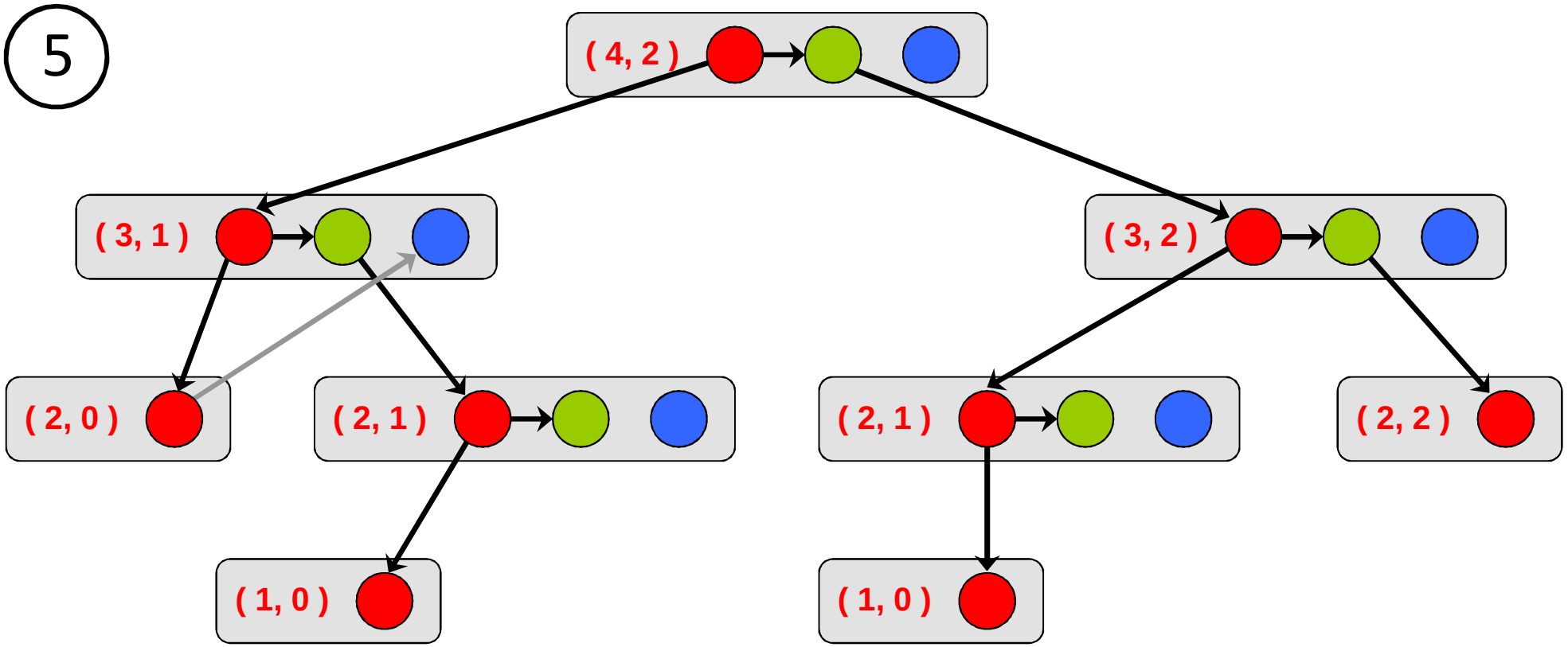
Parallel Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  sync;

  return ( x + y );
}
```



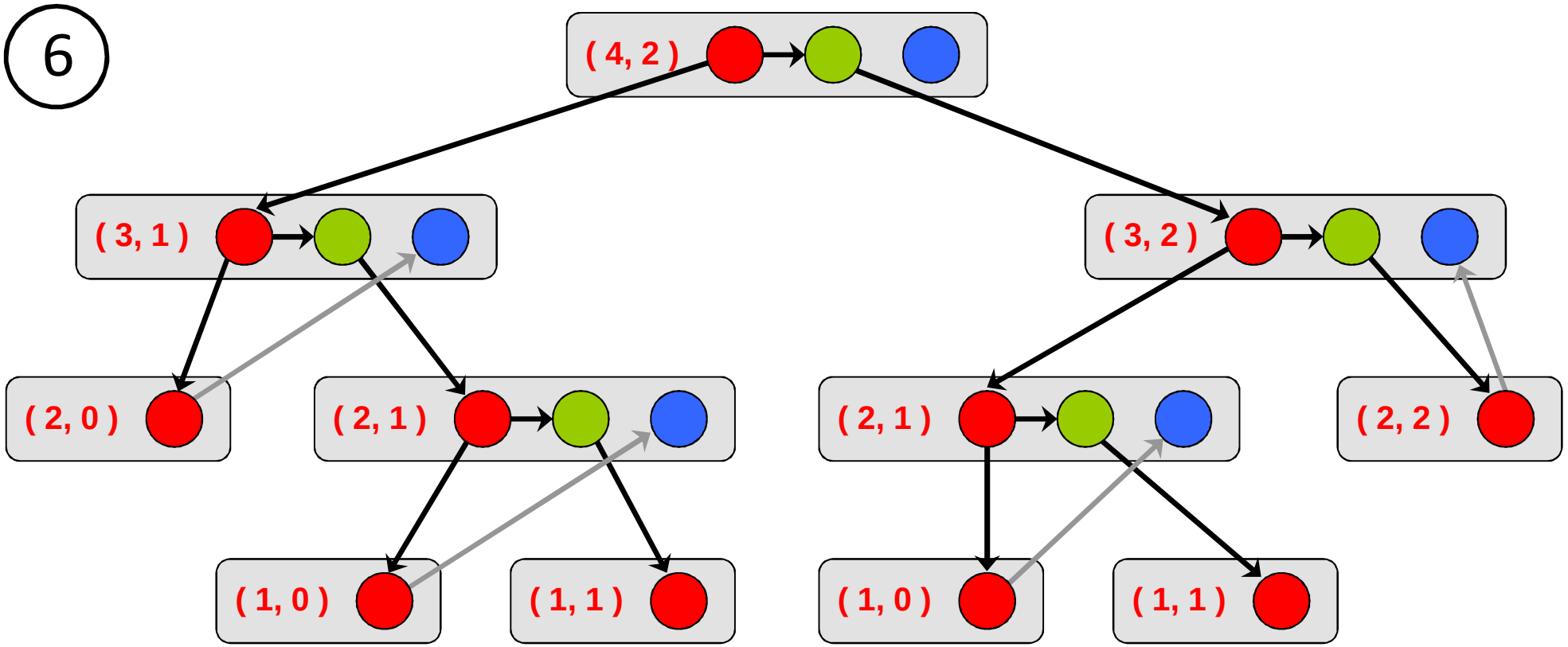
Parallel Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  sync;

  return ( x + y );
}
```



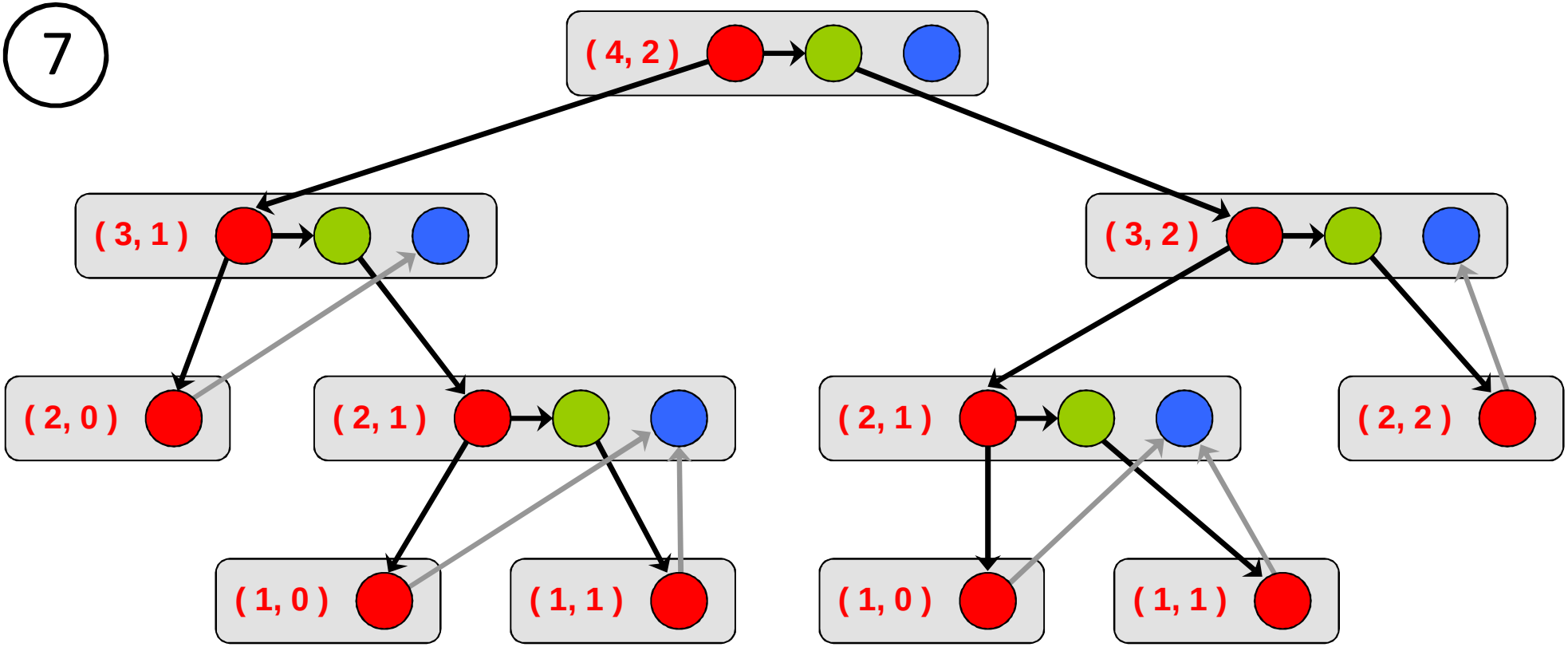
Parallel Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  sync;

  return ( x + y );
}
```



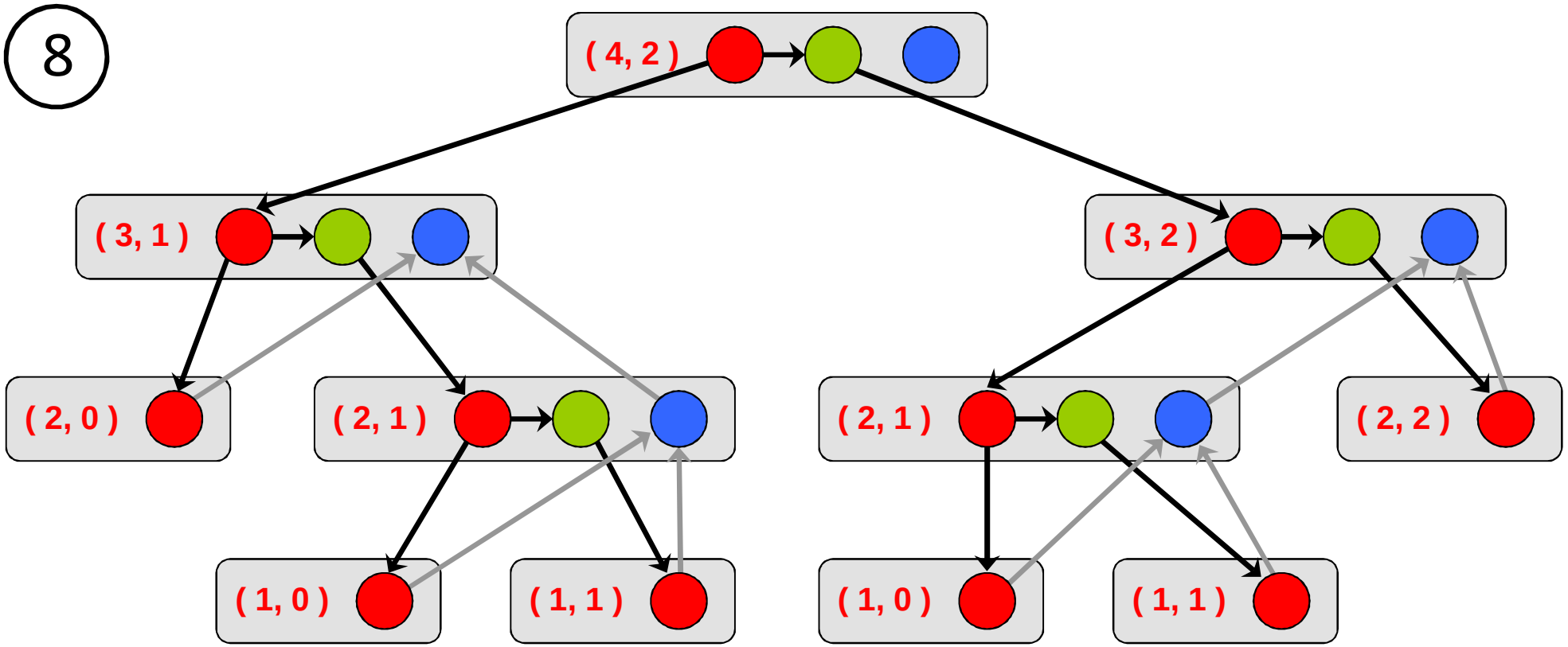
Parallel Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  sync;

  return ( x + y );
}
```

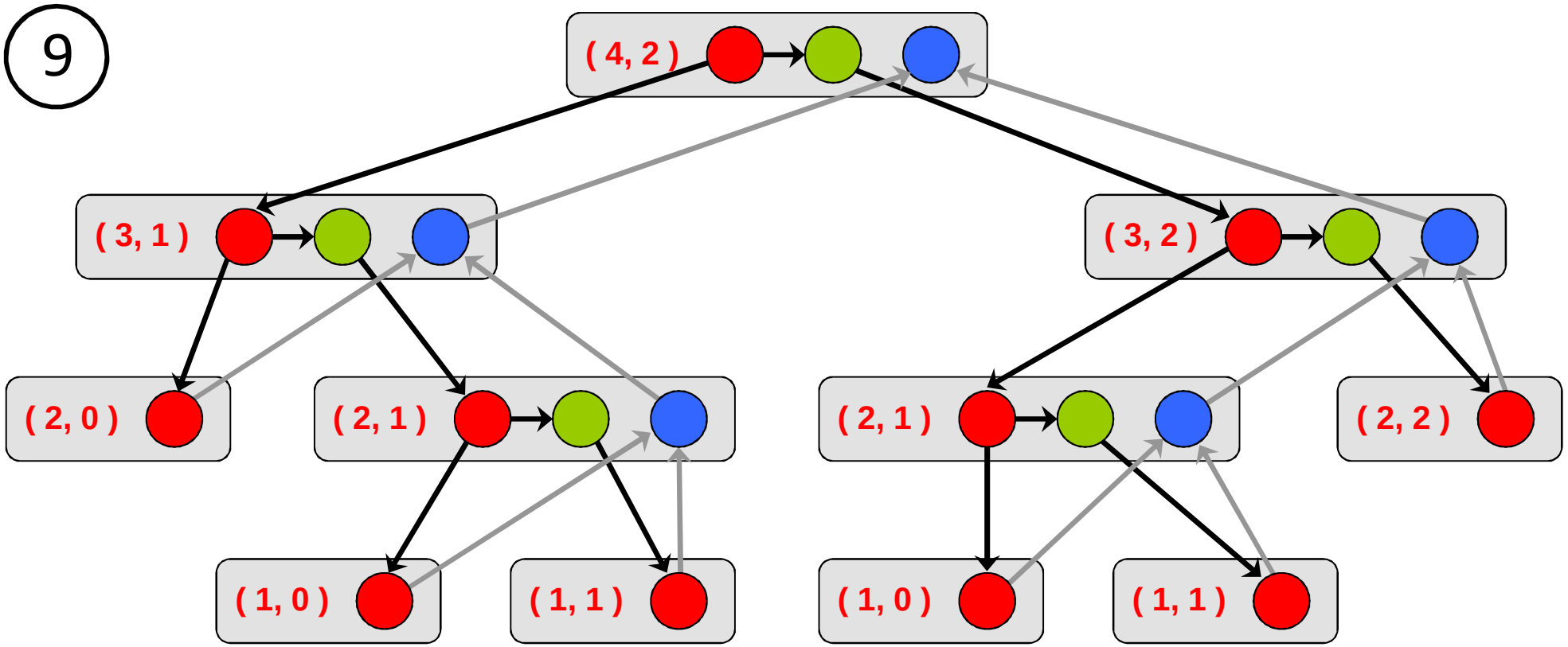
Parallel Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  sync;

  return ( x + y );
}
```

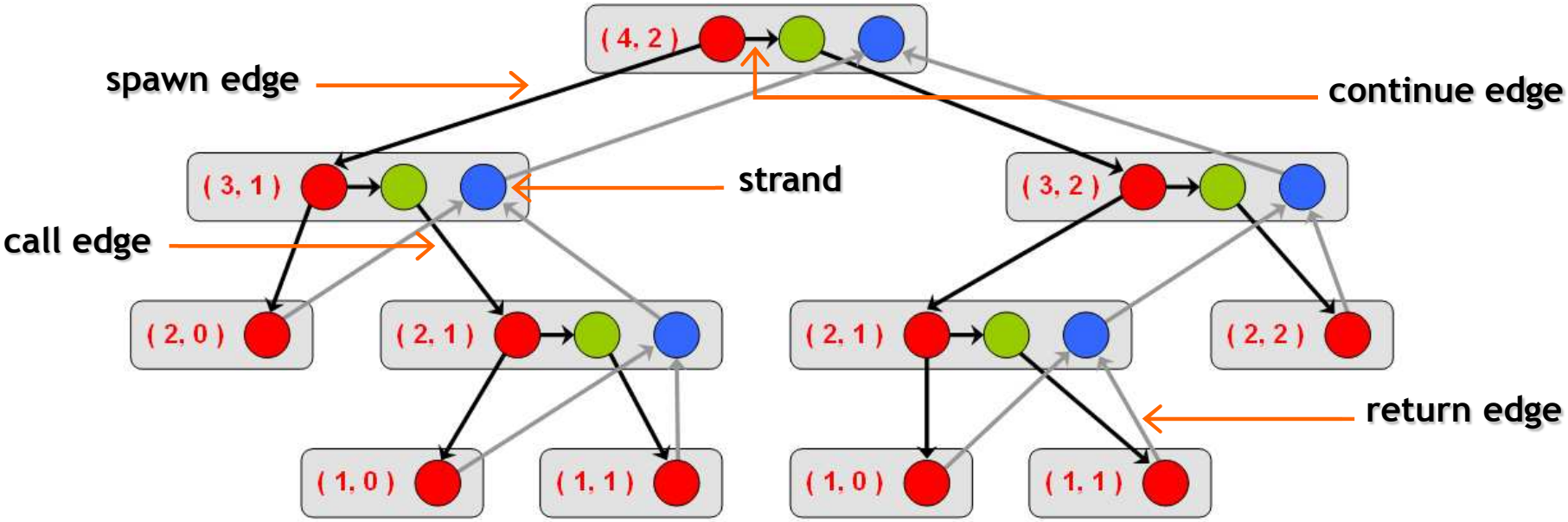


Parallel Execution Model

```
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 );
}
```

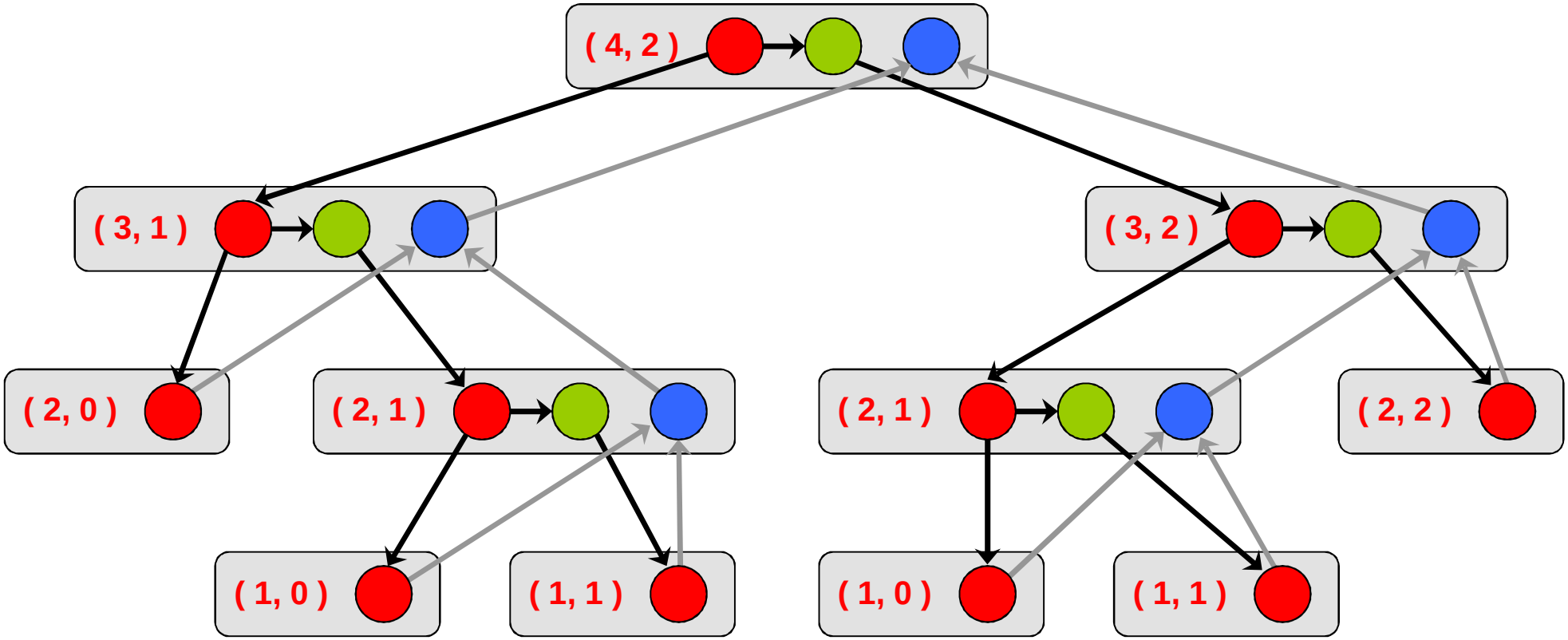


Computation DAG



- A parallel instruction stream is represented by a DAG $G = (V, E)$.
- Each vertex $v \in V$ is a *strand* – a maximal sequence of instructions ending with a spawn, sync or return (implicit or explicit).
- Each edge $e \in E$ is a *spawn*, *call*, *continue* or *return* edge.

Parallelism in comb(4, 2)



work: T_1 = #nodes in the DAG = 21

Critical path

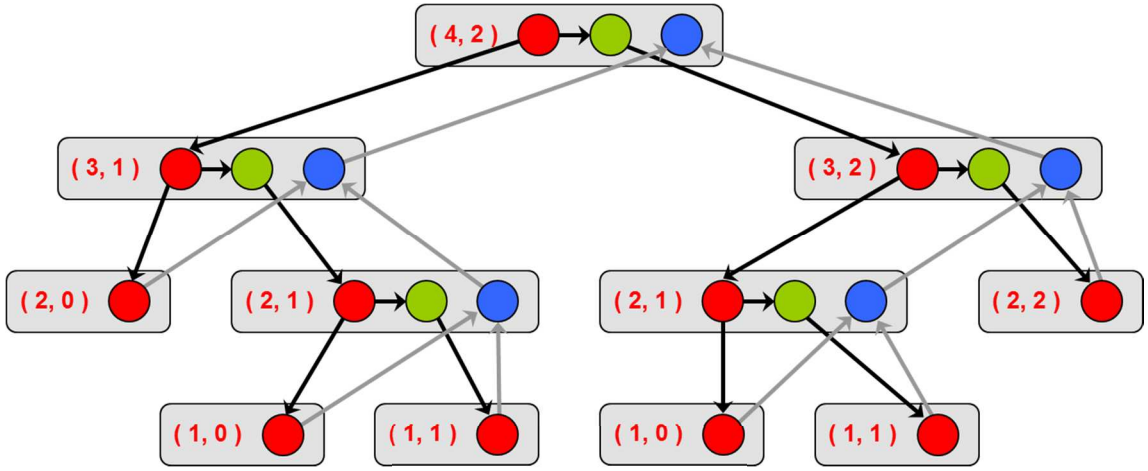
span: T_∞ = #nodes on the longest path in the DAG = 9

parallelism = T_1 / T_∞ = 21 / 9 ≈ 2.33

Only marginal performance gains with more than 2 cores!

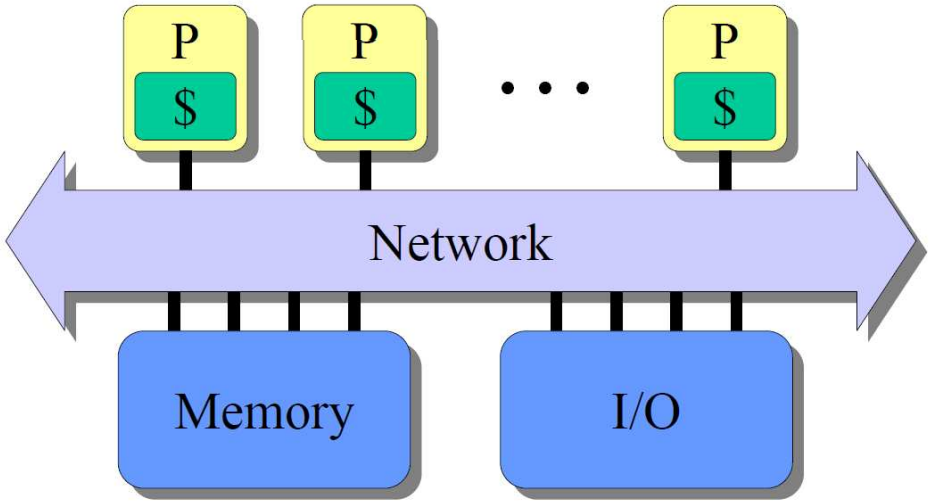
Scheduler

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



The map is called a *schedule*.

An *offline scheduler* prepares the schedule prior to the actual execution of the program.

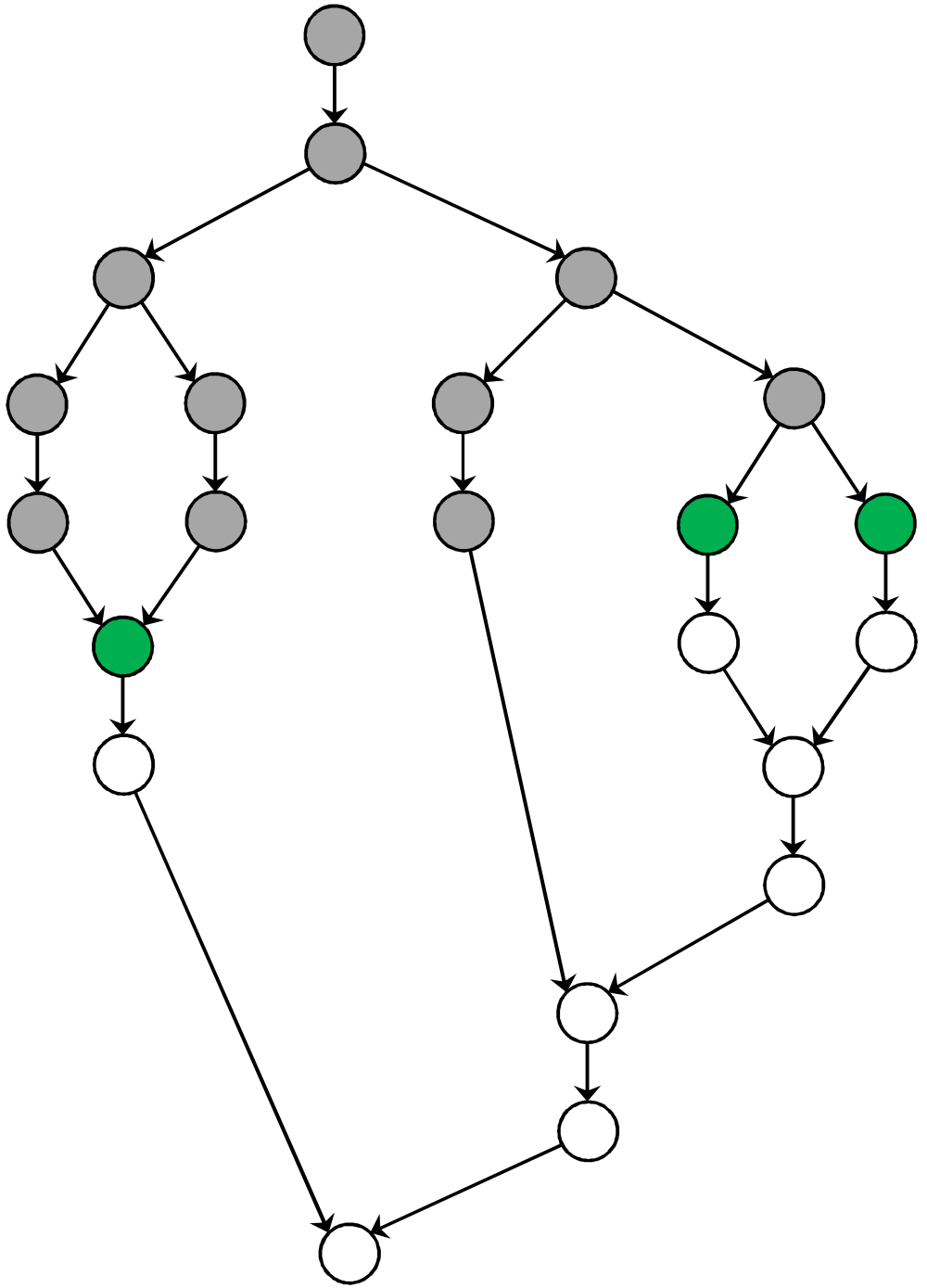


Greedy Scheduling

A strand / task is called *ready* provided all its parents (if any) have already been executed.

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

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

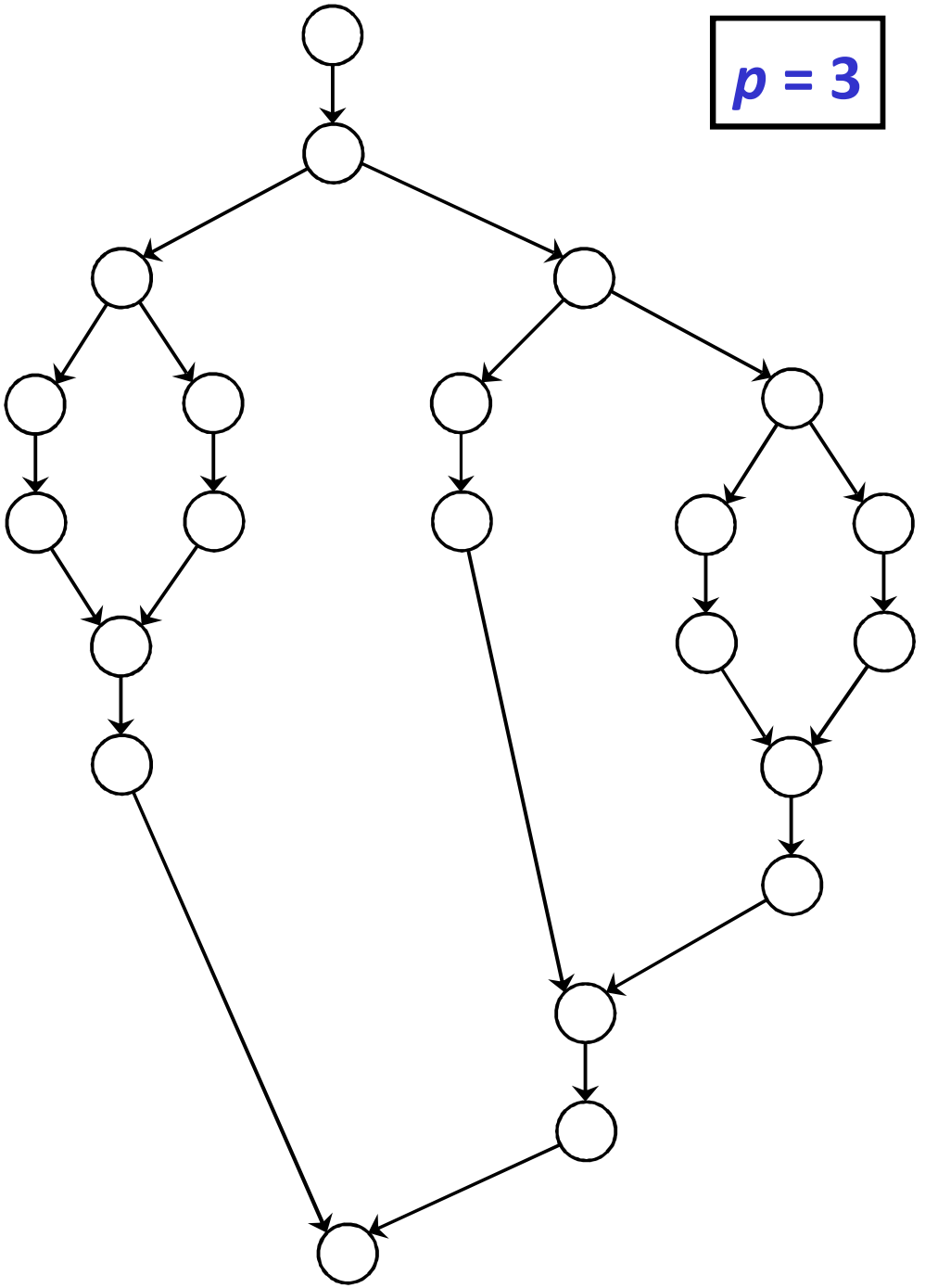


A Centralized Greedy Scheduler

Let p = number of cores

At every step:

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

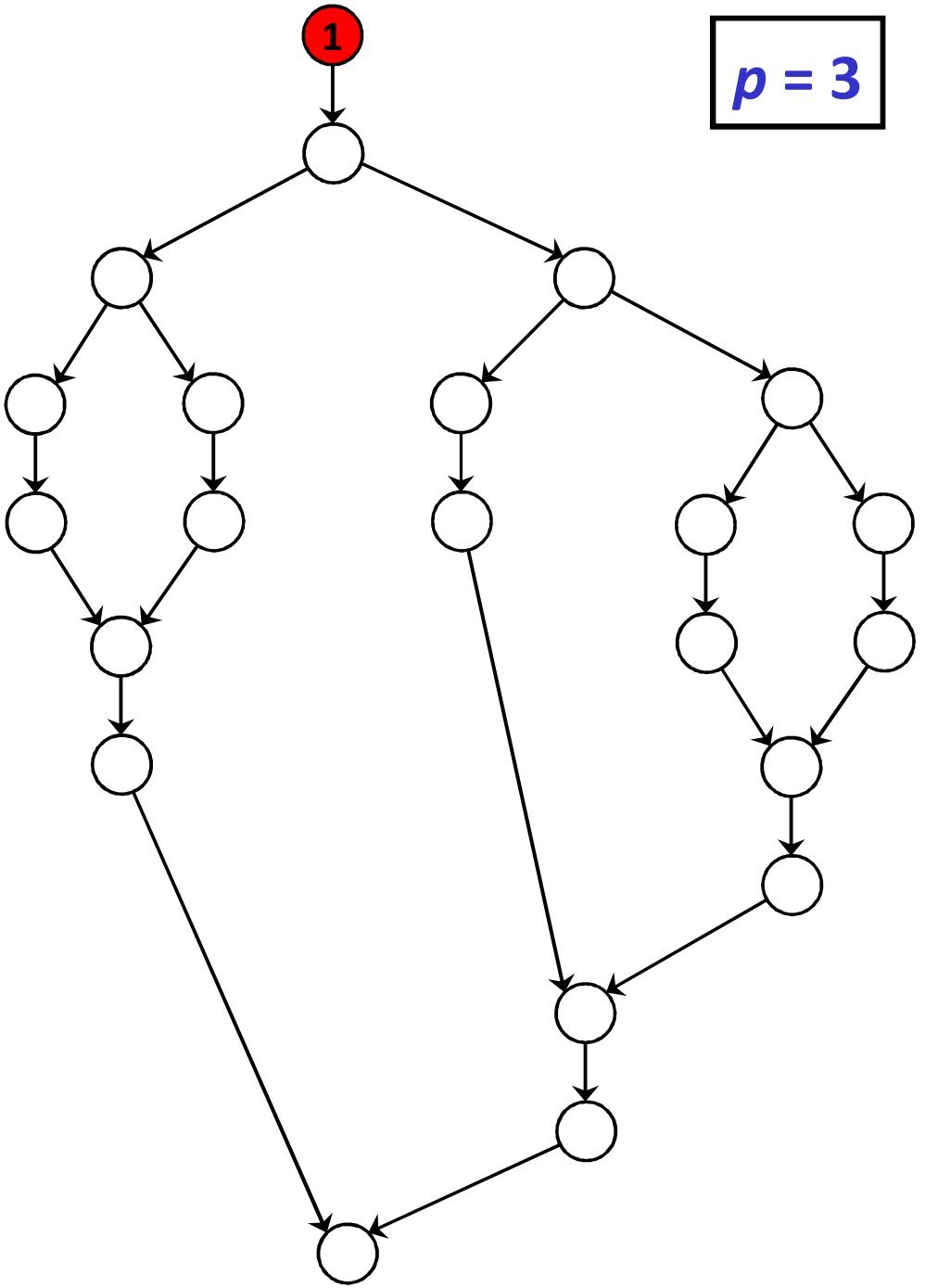


A Centralized Greedy Scheduler

Let p = number of cores

At every step:

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

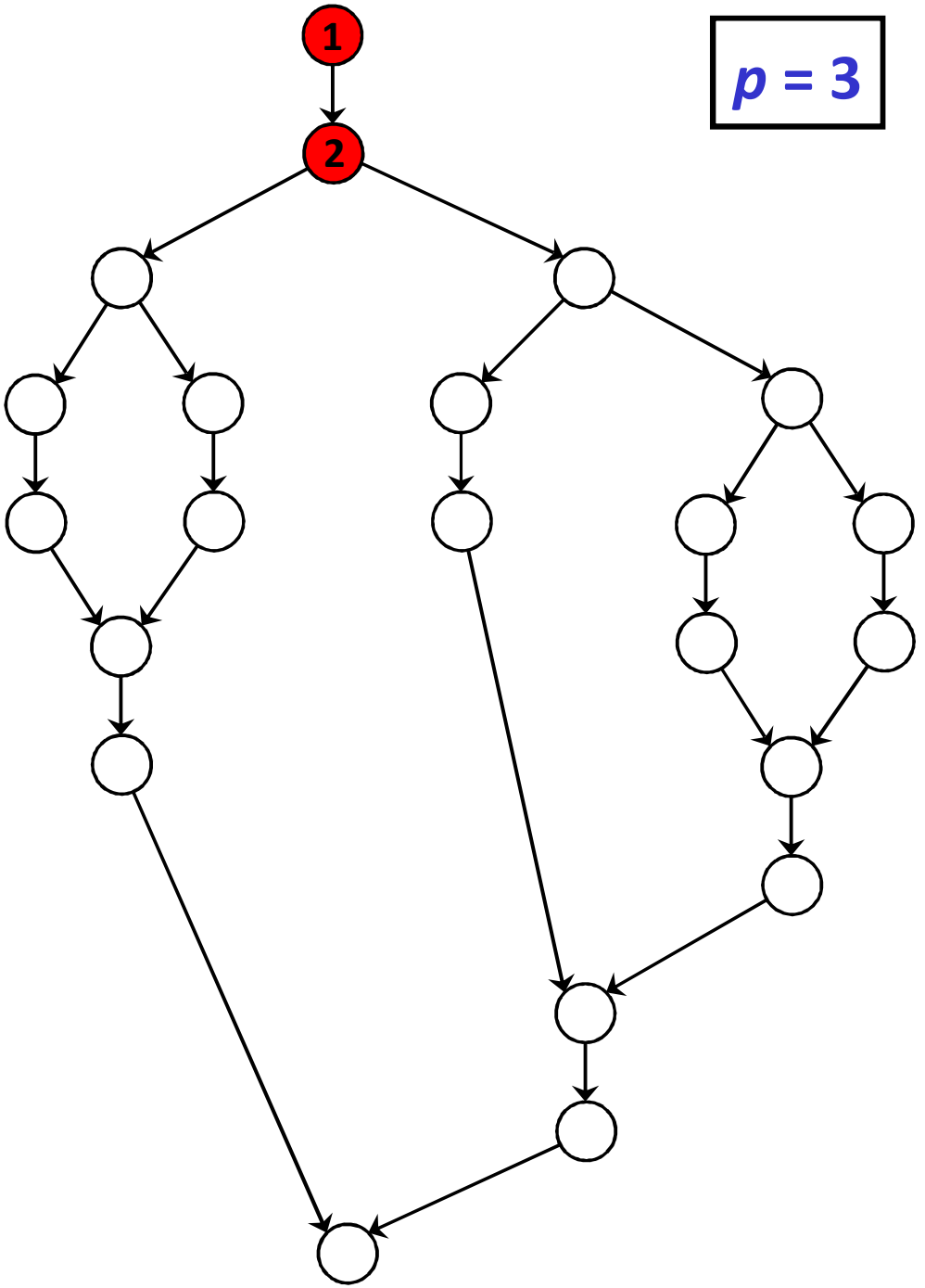


A Centralized Greedy Scheduler

Let p = number of cores

At every step:

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

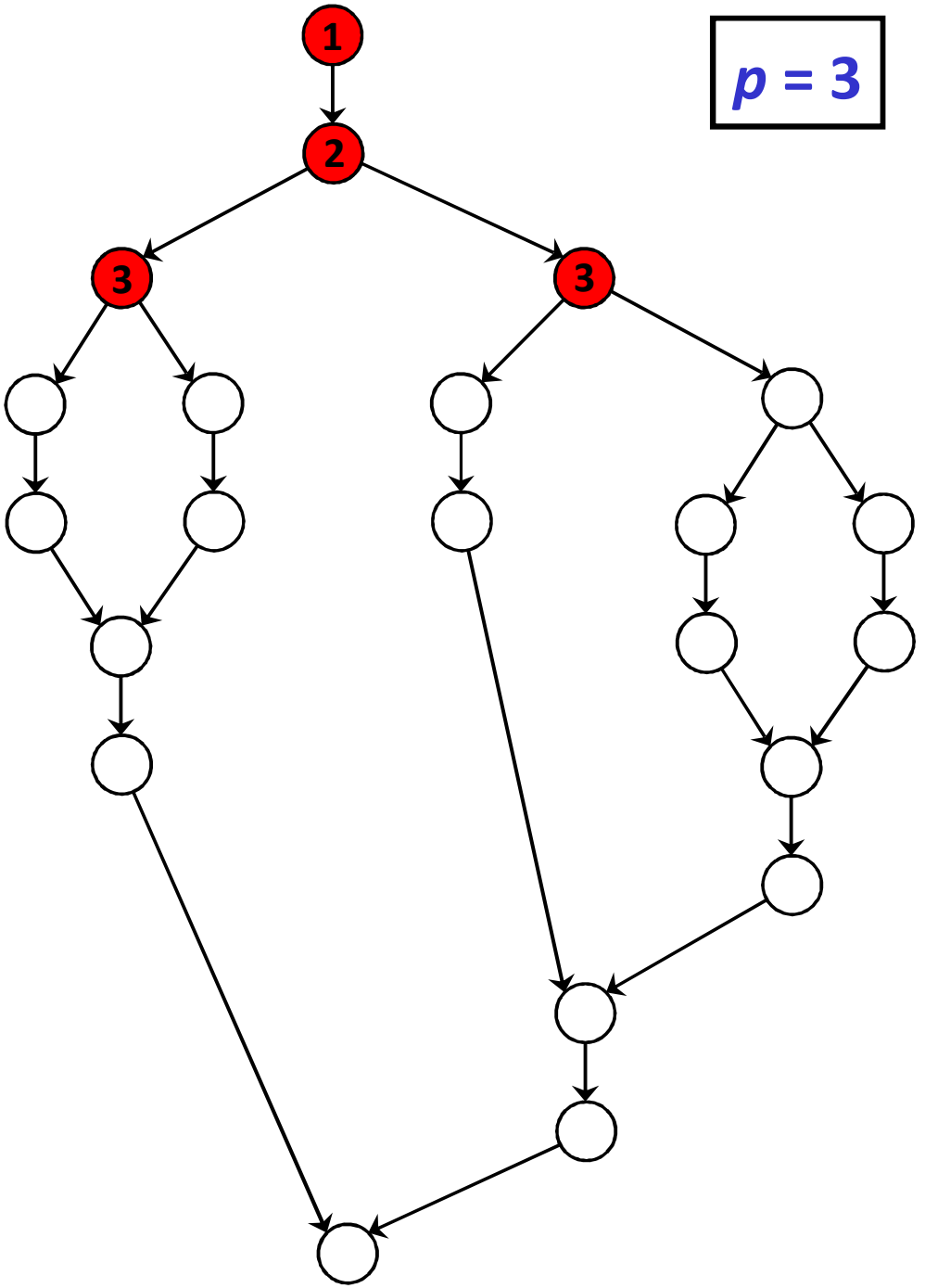


A Centralized Greedy Scheduler

Let p = number of cores

At every step:

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

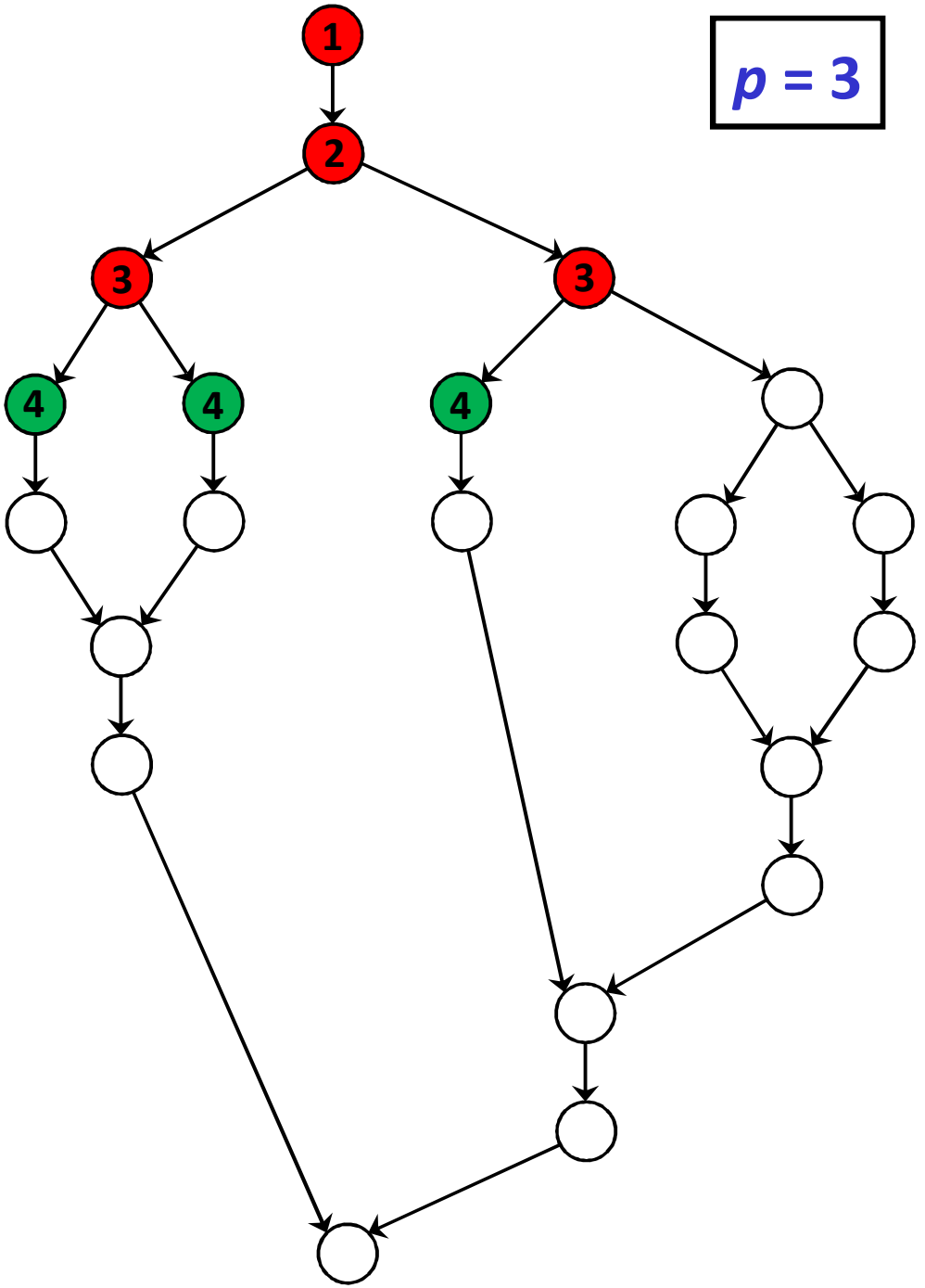


A Centralized Greedy Scheduler

Let p = number of cores

At every step:

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

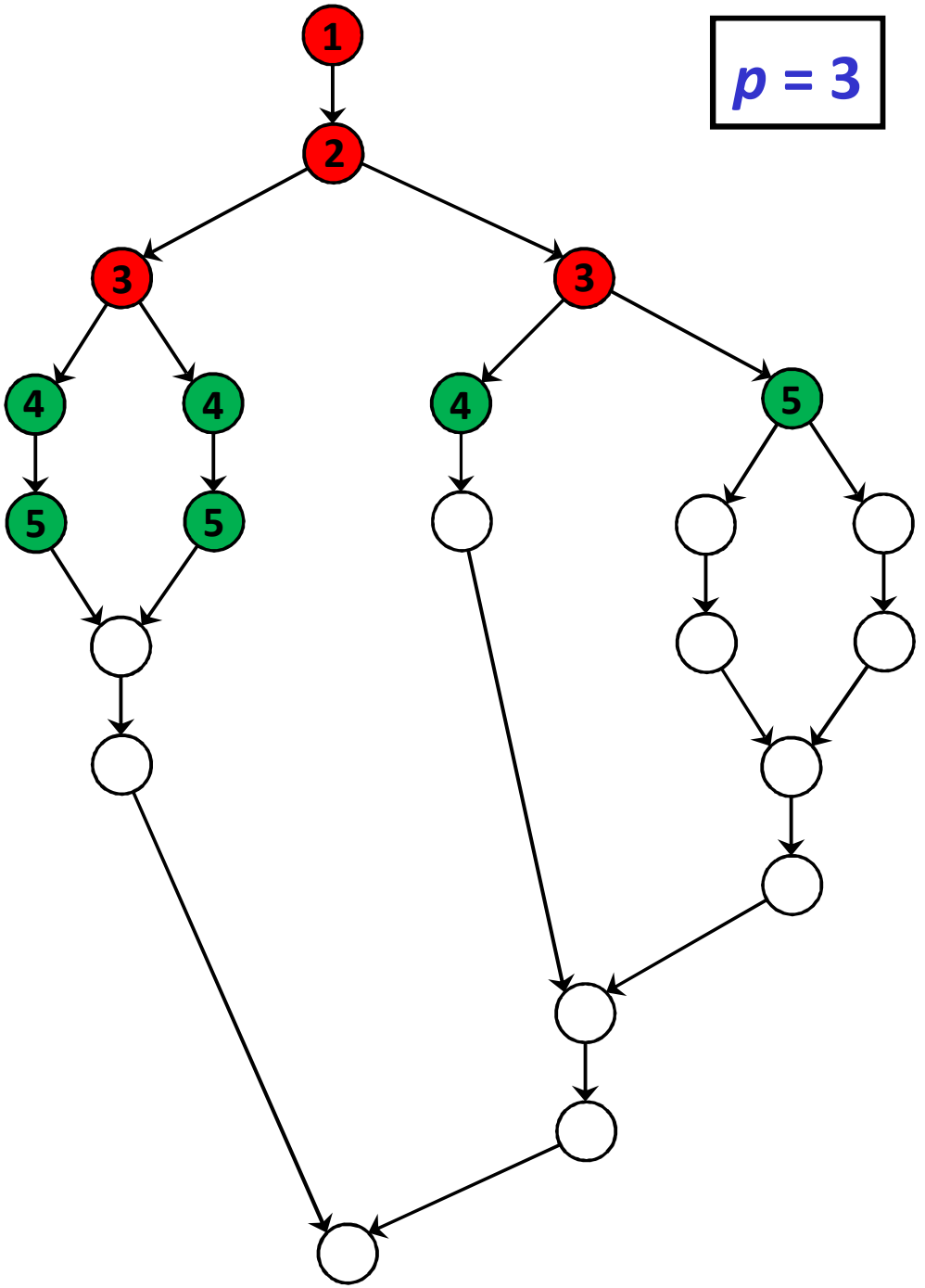


A Centralized Greedy Scheduler

Let p = number of cores

At every step:

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

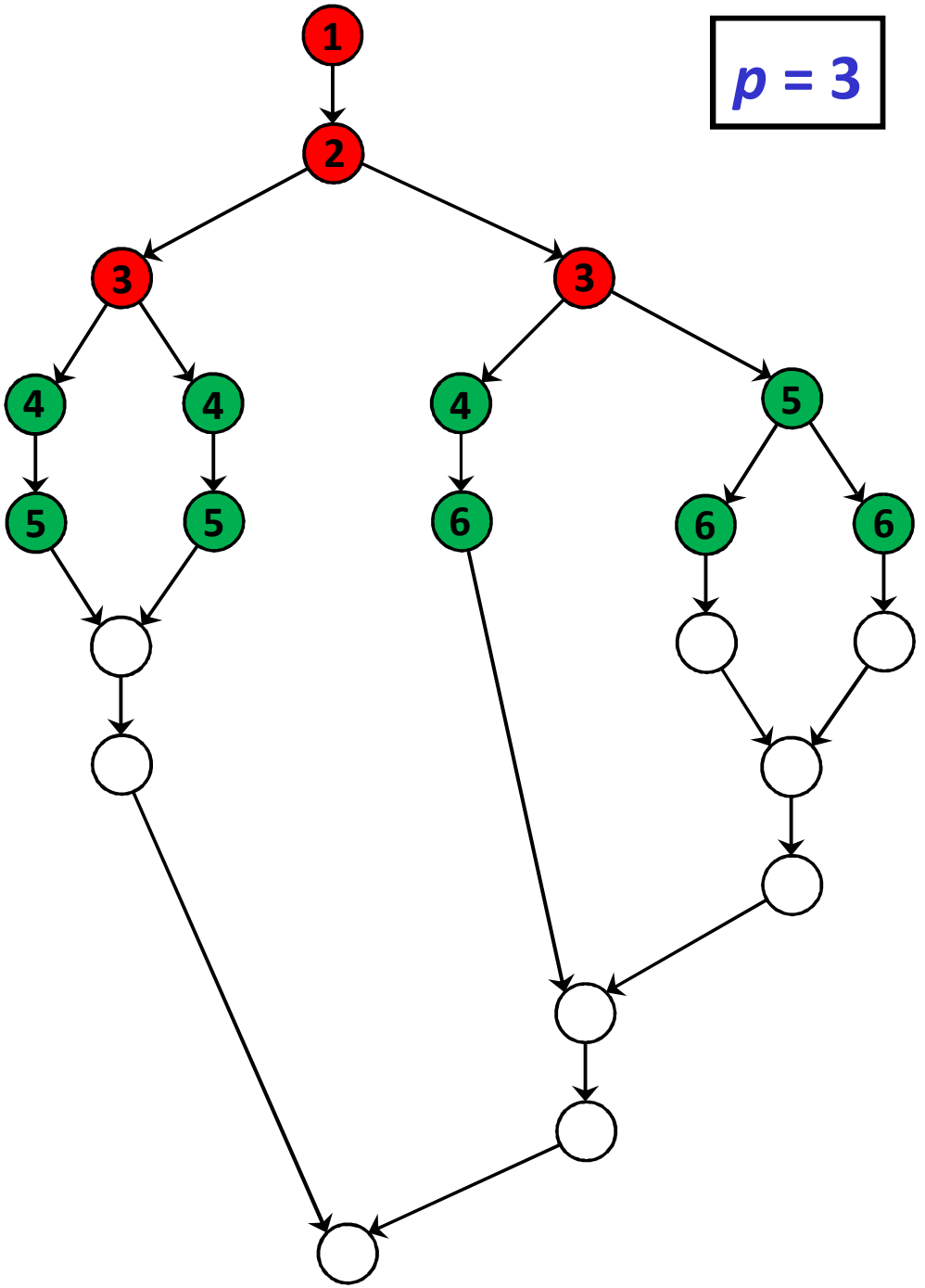


A Centralized Greedy Scheduler

Let p = number of cores

At every step:

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

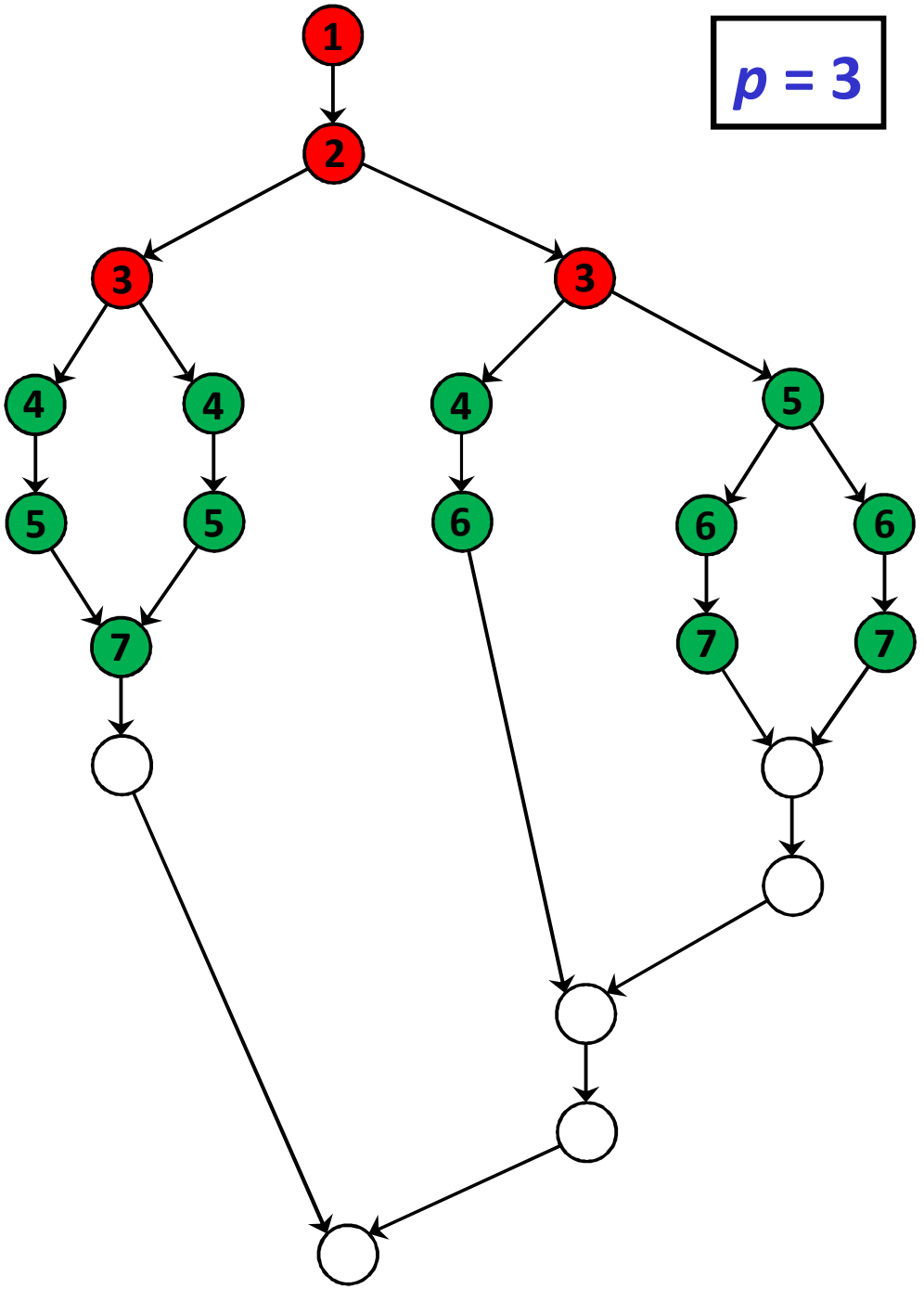


A Centralized Greedy Scheduler

Let p = number of cores

At every step:

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

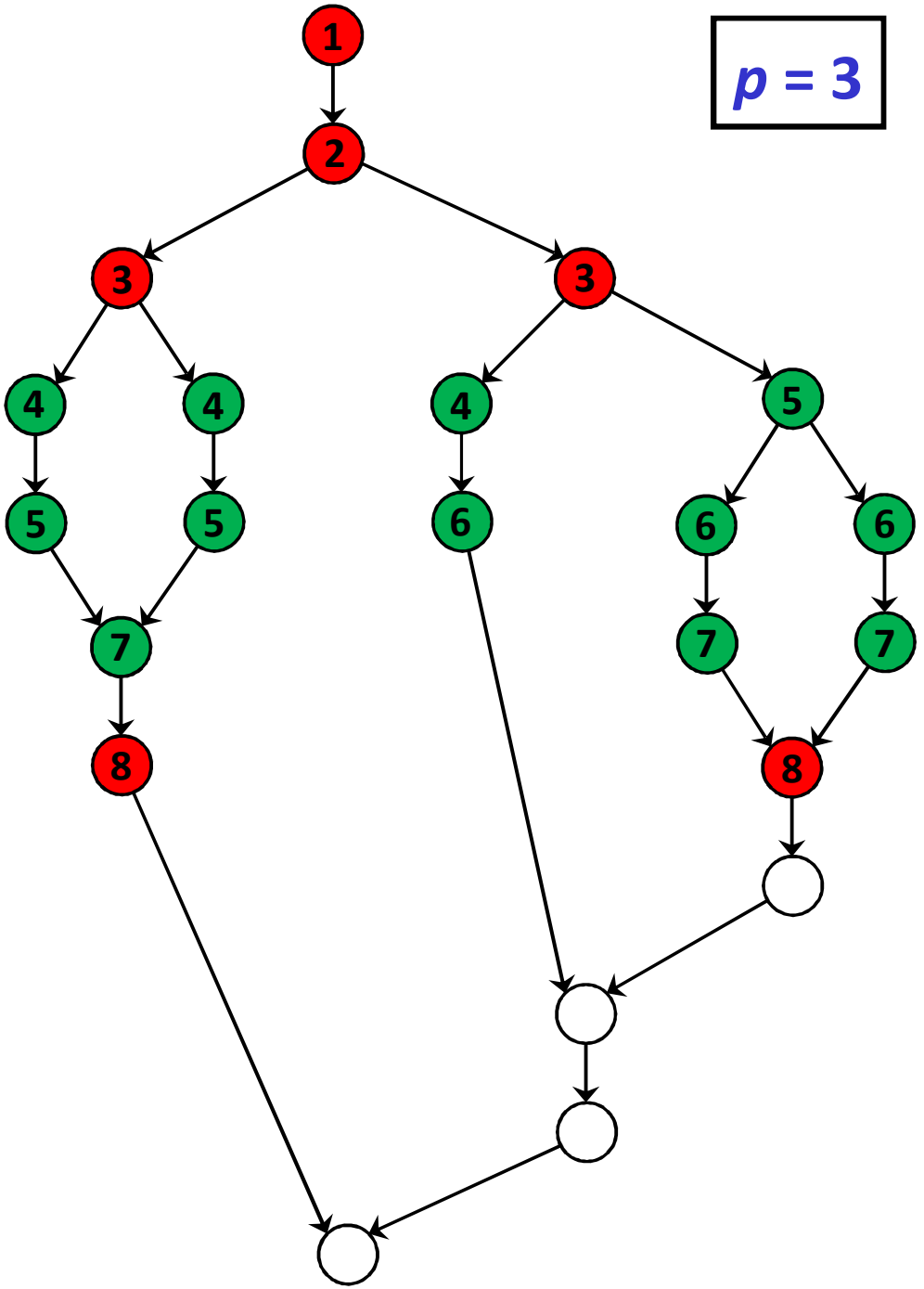


A Centralized Greedy Scheduler

Let p = number of cores

At every step:

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

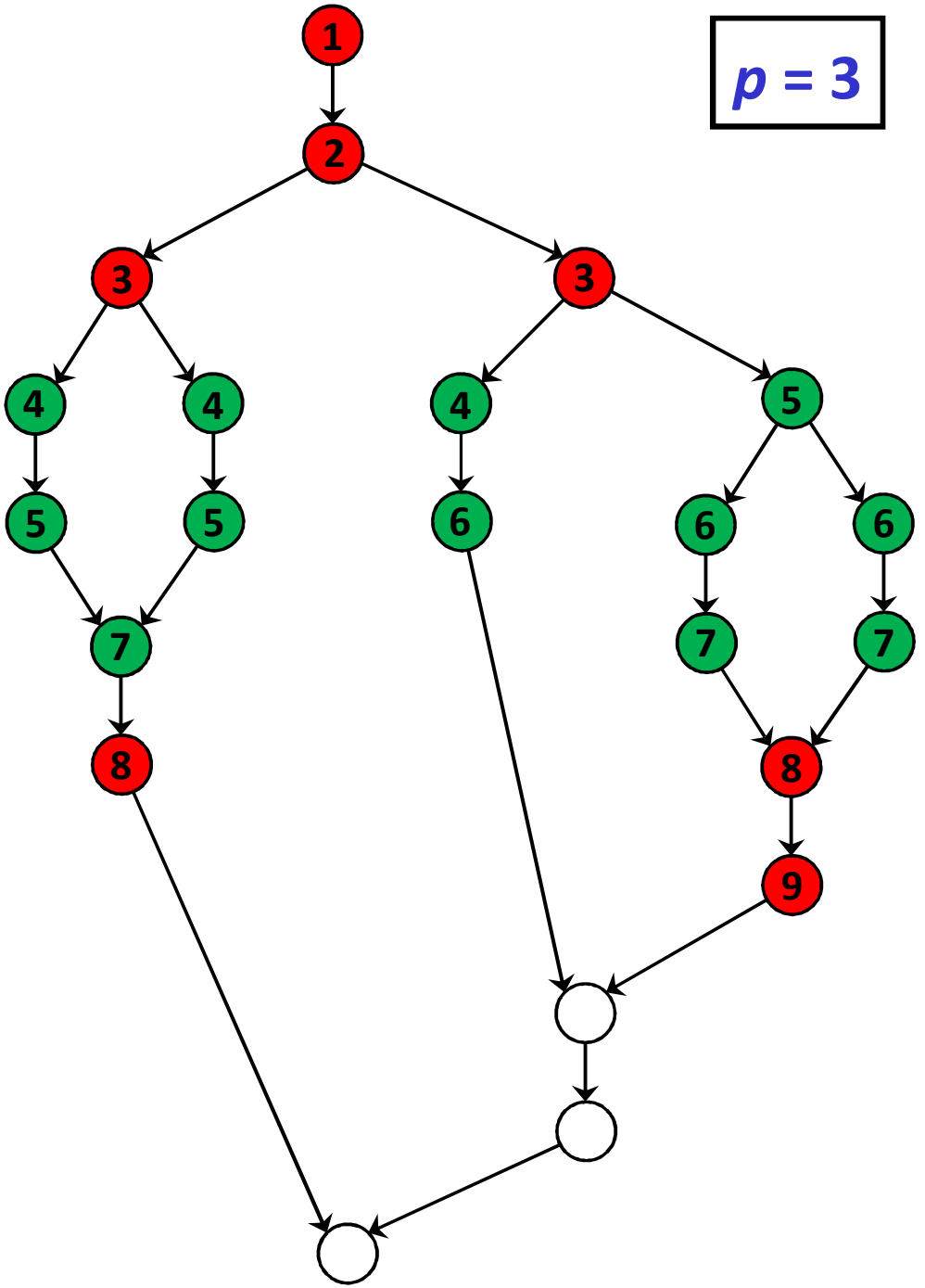


A Centralized Greedy Scheduler

Let p = number of cores

At every step:

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

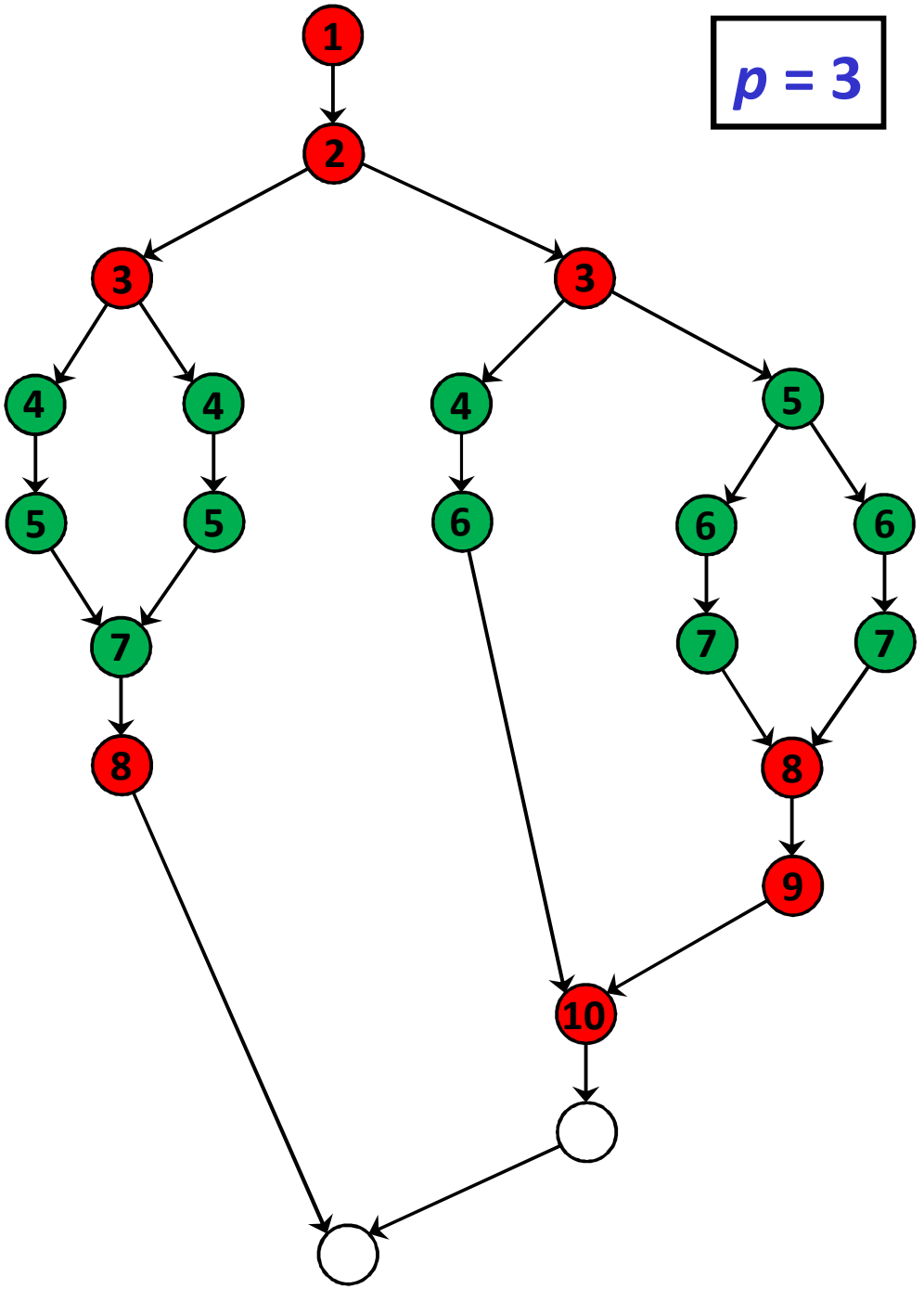


A Centralized Greedy Scheduler

Let p = number of cores

At every step:

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

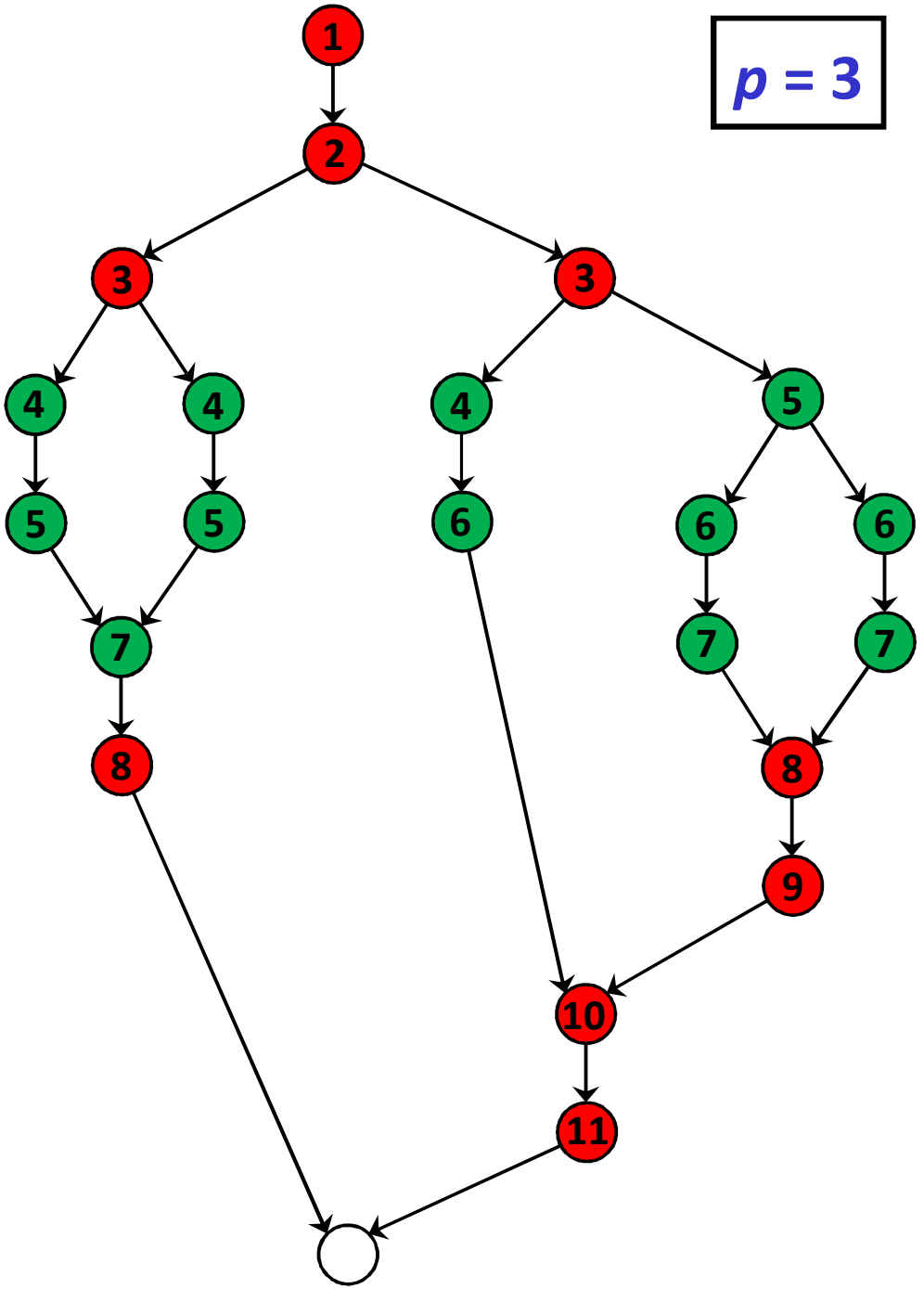


A Centralized Greedy Scheduler

Let p = number of cores

At every step:

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

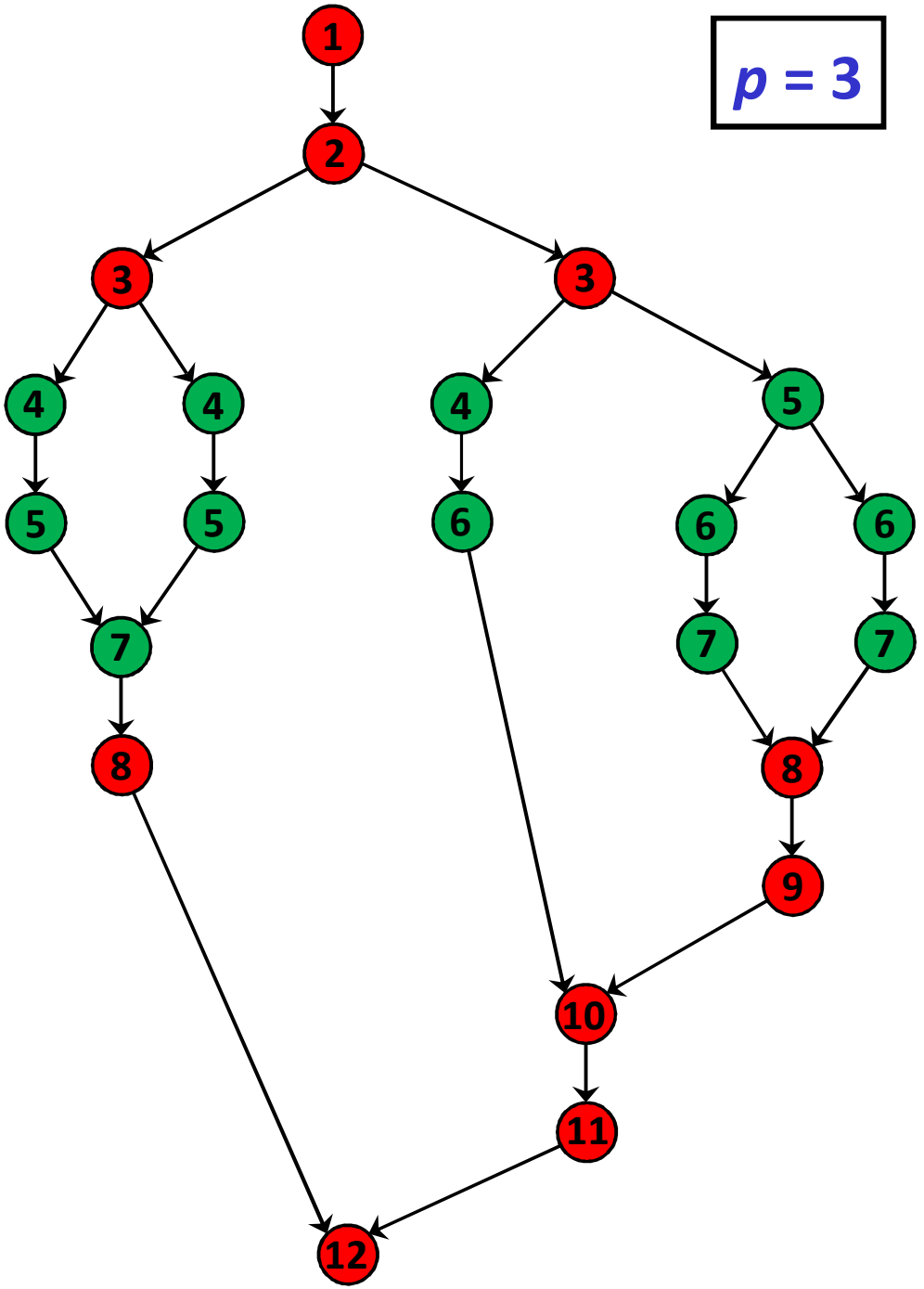


A Centralized Greedy Scheduler

Let p = number of cores

At every step:

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



Greedy Scheduling Theorem

Theorem [Graham'68, Brent'74]:

For any greedy scheduler,

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

Proof:

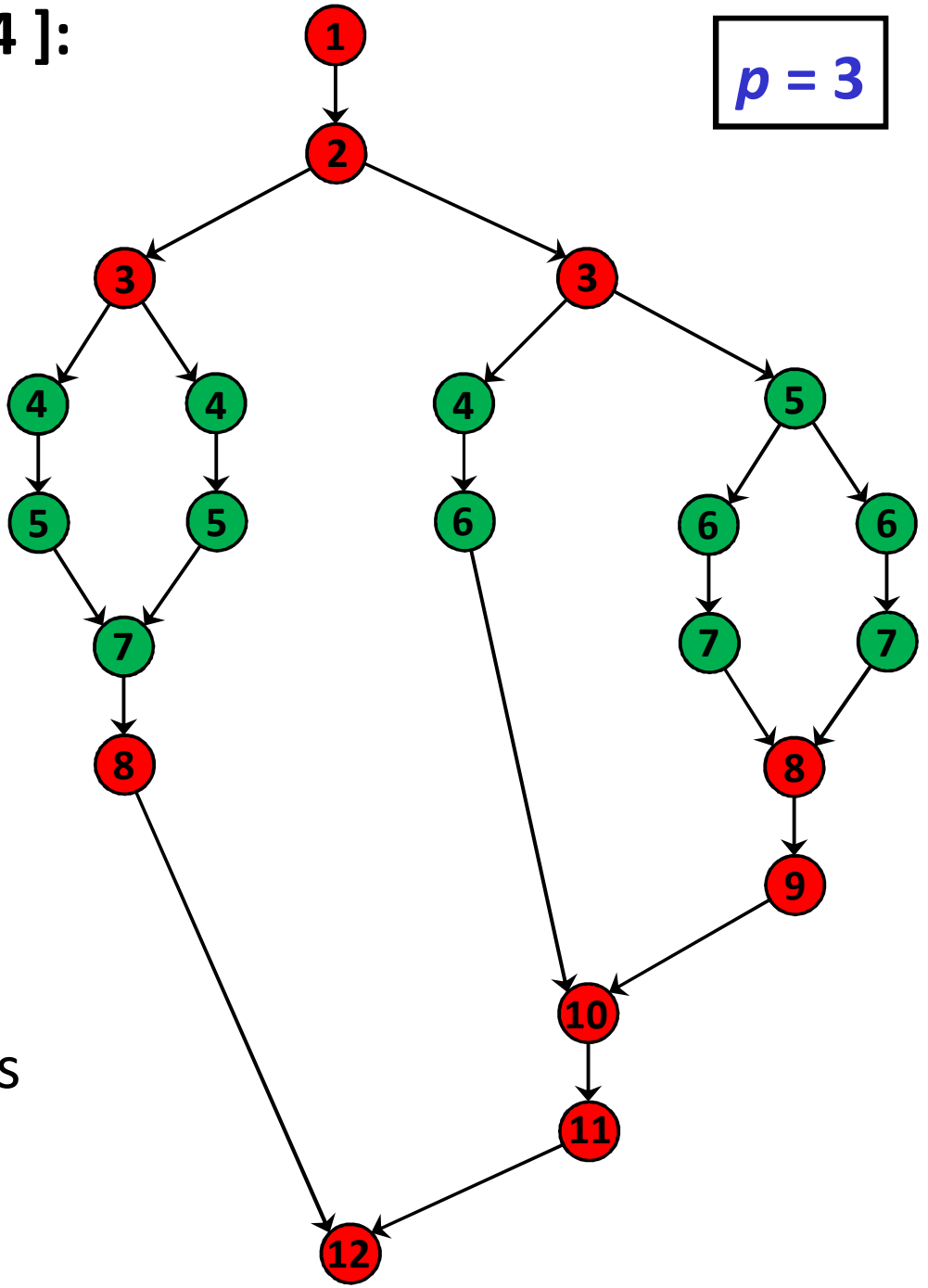
T_p = #complete steps
+ #incomplete steps

– Each complete step performs p work:

#complete steps $\leq \frac{T_1}{p}$

– Each incomplete step reduces the span by 1:

#incomplete steps $\leq T_\infty$



Optimality of the Greedy Scheduler

Corollary 1: For any greedy scheduler $T_p \leq 2T_p^*$, where T_p^* is the running time due to optimal scheduling on *p* processing elements.

Proof:

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

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

$\therefore$ From Graham-Brent Theorem:

$$T_p \leq \frac{T_1}{p} + T_\infty \leq T_p^* + T_p^* = 2T_p^*$$

Optimality of the Greedy Scheduler

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

Proof:

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

$\therefore$ From Graham-Brent Theorem:

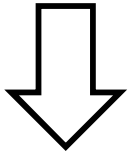
$$T_p \leq \frac{T_1}{p} + T_\infty \approx \frac{T_1}{p}$$

$$\Rightarrow \frac{T_1}{T_p} \approx p \Rightarrow S_p \approx p$$

Parallel
Matrix Multiplication

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 × n matrices,
                                 where n is a positive integer }
1.  parallel for i ← 1 to n do
2.      parallel for j ← 1 to n do
3.          Z[ i ][ j ] ← 0
4.          for k ← 1 to n do
5.              Z[ i ][ j ] ← Z[ i ][ j ] + X[ i ][ k ] · Y[ k ][ j ]
```

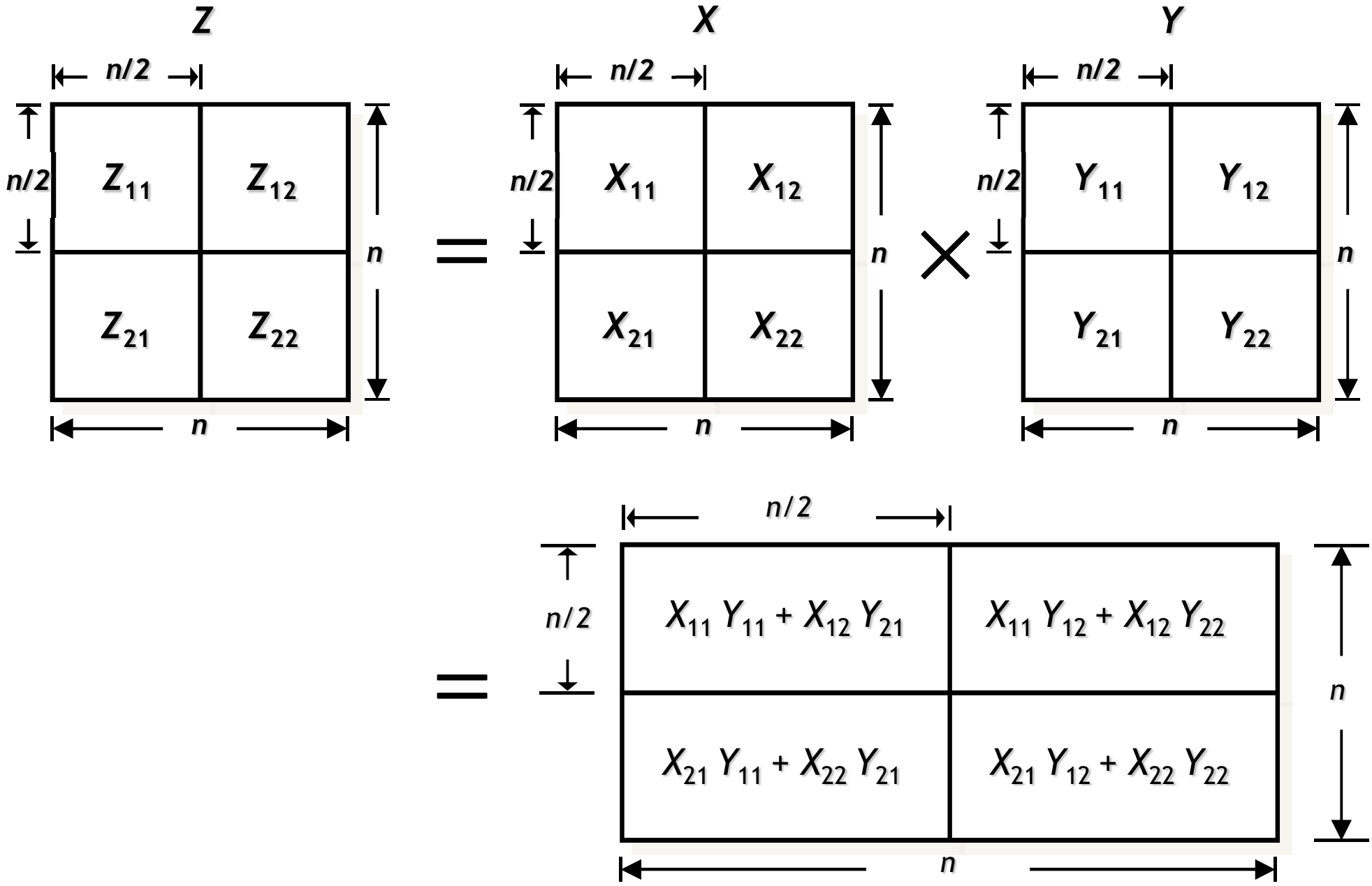
Work: $T_1(n) = \Theta(n^3)$

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

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

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

Parallel Recursive MM



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 ( Z11, X11, Y11 )
 5.     spawn Par-Rec-MM ( Z12, X11, Y12 )
 6.     spawn Par-Rec-MM ( Z21, X21, Y11 )
 7.           Par-Rec-MM ( Z21, X21, Y12 )
 8.     sync
 9.     spawn Par-Rec-MM ( Z11, X12, Y21 )
10.     spawn Par-Rec-MM ( Z12, X12, Y22 )
11.     spawn Par-Rec-MM ( Z21, X22, Y21 )
12.           Par-Rec-MM ( Z22, X22, Y22 )
13.     sync
14. endif
```

Parallel Recursive MM

```
Par-Rec-MM ( Z, X, Y )    { X, Y, Z are n × n matrices,
                              where n = 2^k for integer k ≥ 0 }
1.  if n = 1 then
2.      Z ← Z + X · Y
3.  else
4.      spawn Par-Rec-MM ( Z11, X11, Y11 )
5.      spawn Par-Rec-MM ( Z12, X11, Y12 )
6.      spawn Par-Rec-MM ( Z21, X21, Y11 )
7.            Par-Rec-MM ( Z21, X21, Y12 )
8.      sync
9.      spawn Par-Rec-MM ( Z11, X12, Y21 )
10.     spawn Par-Rec-MM ( Z12, X12, Y22 )
11.     spawn Par-Rec-MM ( Z21, X22, Y21 )
12.           Par-Rec-MM ( Z22, X22, Y22 )
13.     sync
14. endif
```

Work:

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

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

Span:

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

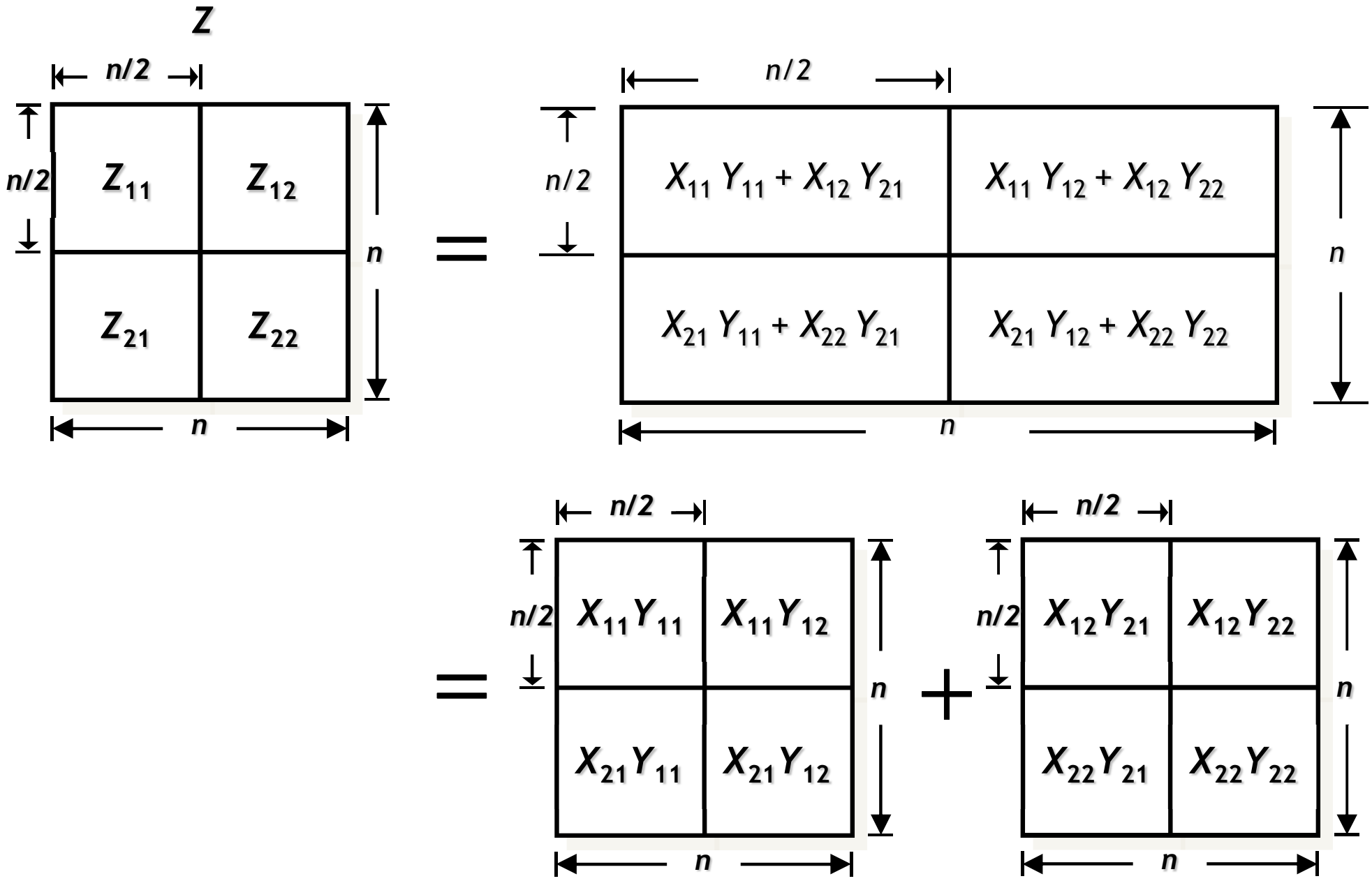
$$= \Theta(n) \qquad \text{[MT Case 1]}$$

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

Additional Space:

$$s_\infty(n) = \Theta(1)$$

Recursive MM with More Parallelism



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 ( Z11, X11, Y11 )
 5.     spawn Par-Rec-MM2 ( Z12, X11, Y12 )
 6.     spawn Par-Rec-MM2 ( Z21, X21, Y11 )
 7.     spawn Par-Rec-MM2 ( Z21, X21, Y12 )
 8.     spawn Par-Rec-MM2 ( T11, X12, Y21 )
 9.     spawn Par-Rec-MM2 ( T12, X12, Y22 )
10.     spawn Par-Rec-MM2 ( T21, X22, Y21 )
11.           Par-Rec-MM2 ( T22, X22, Y22 )
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 × 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 ( Z11, X11, Y11 )
 5.     spawn Par-Rec-MM2 ( Z12, X11, Y12 )
 6.     spawn Par-Rec-MM2 ( Z21, X21, Y11 )
 7.     spawn Par-Rec-MM2 ( Z21, X21, Y12 )
 8.     spawn Par-Rec-MM2 ( T11, X12, Y21 )
 9.     spawn Par-Rec-MM2 ( T12, X12, Y22 )
10.     spawn Par-Rec-MM2 ( T21, X22, Y21 )
11.           Par-Rec-MM2 ( T22, X22, Y22 )
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
```

Work:

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

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

Span:

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

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

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

Additional Space:

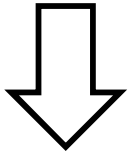
$$s_\infty(n) = \begin{cases} \Theta(1), & if\ n = 1, \\ 8s_\infty\left(\frac{n}{2}\right) + \Theta(n^2), & otherwise. \end{cases}$$

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

Parallel Merge Sort

Parallel Merge Sort

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

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



```
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 )
```

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

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

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

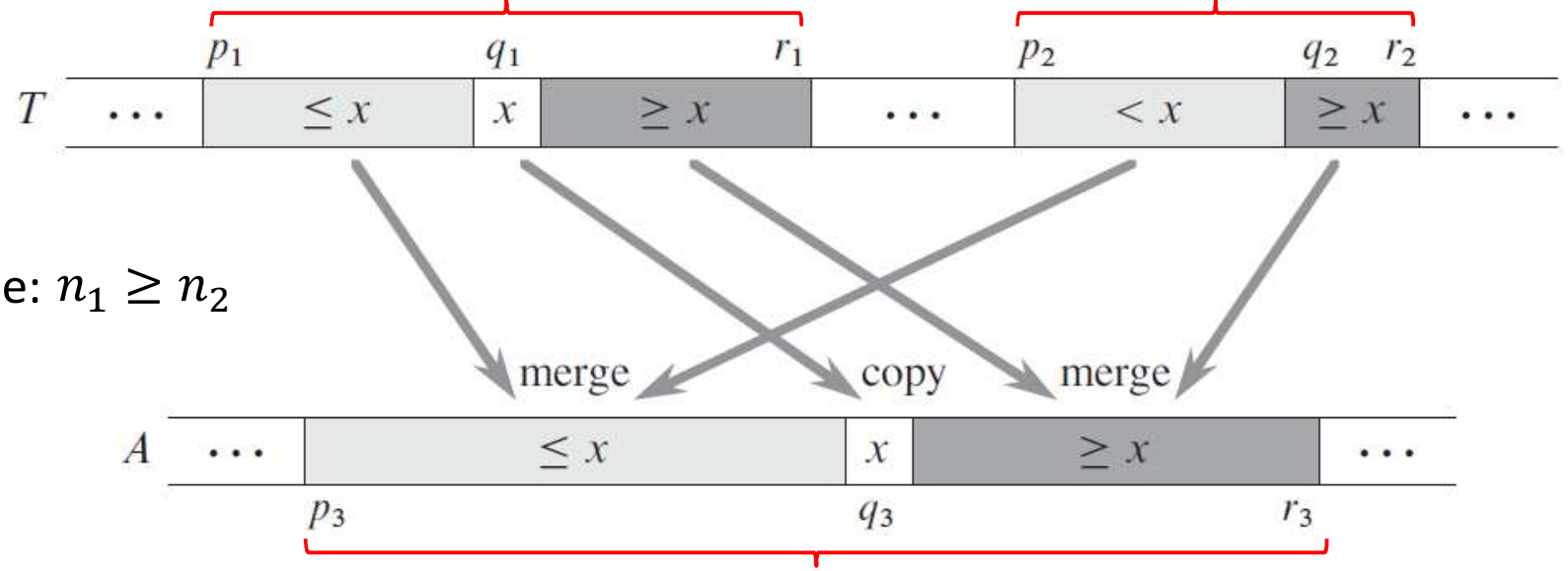
$= \Theta(n)$ [MT Case 3]

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

Parallel Merge

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

subarrays to merge: $T[p_1..r_1]$ $\quad$ $T[p_2..r_2]$



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

suppose: $n_1 \geq n_2$

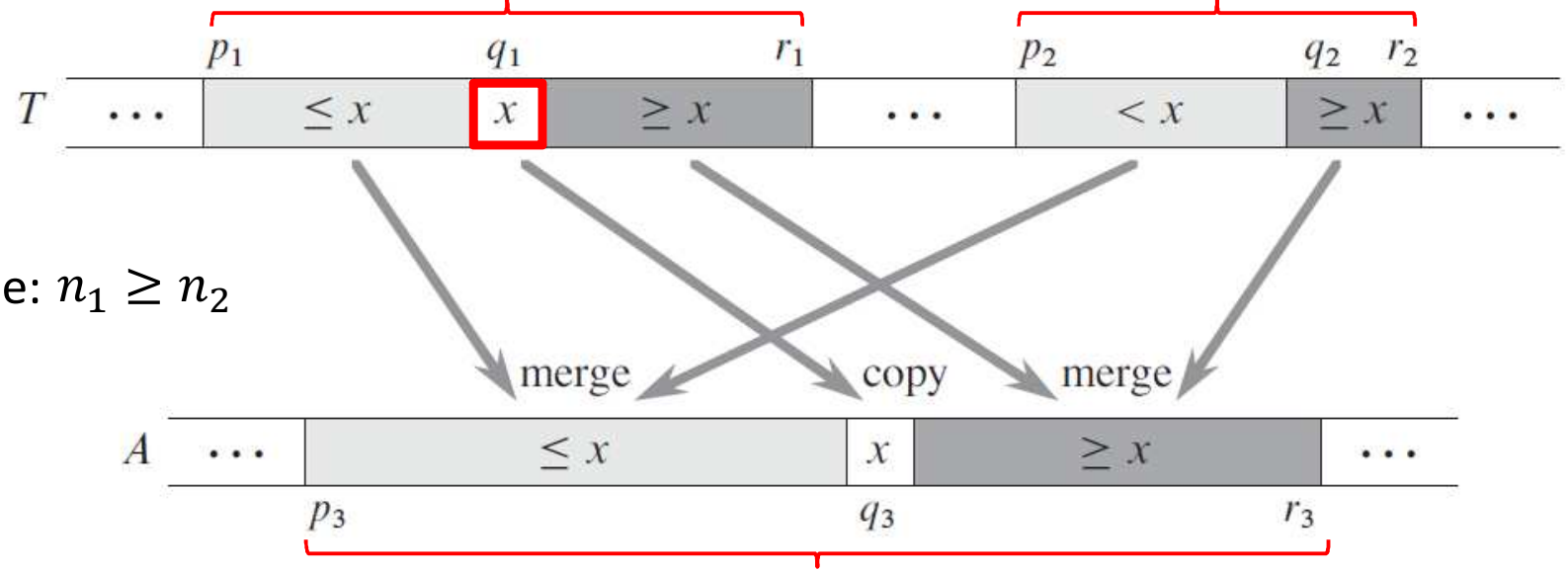
merged output: $A[p_3..r_3]$

$n_3 = r_3 - p_3 + 1 = n_1 + n_2$

Parallel Merge

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

subarrays to merge: $T[p_1..r_1]$ $\quad$ $T[p_2..r_2]$



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

suppose: $n_1 \geq n_2$

merged output: $A[p_3..r_3]$

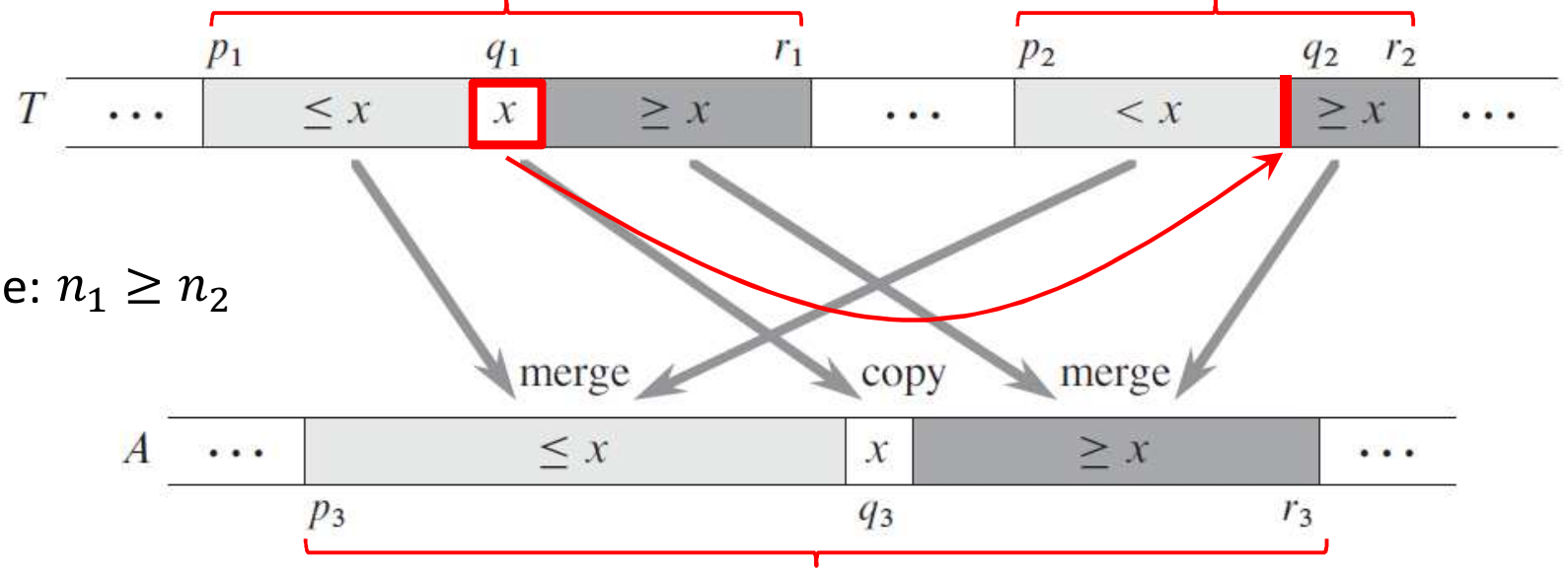
$n_3 = r_3 - p_3 + 1 = n_1 + n_2$

Step 1: Find $x = T[q_1]$, where q_1 is the midpoint of $T[p_1..r_1]$

Parallel Merge

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

subarrays to merge: $T[p_1..r_1]$ $\quad$ $T[p_2..r_2]$



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

suppose: $n_1 \geq n_2$

merged output: $A[p_3..r_3]$

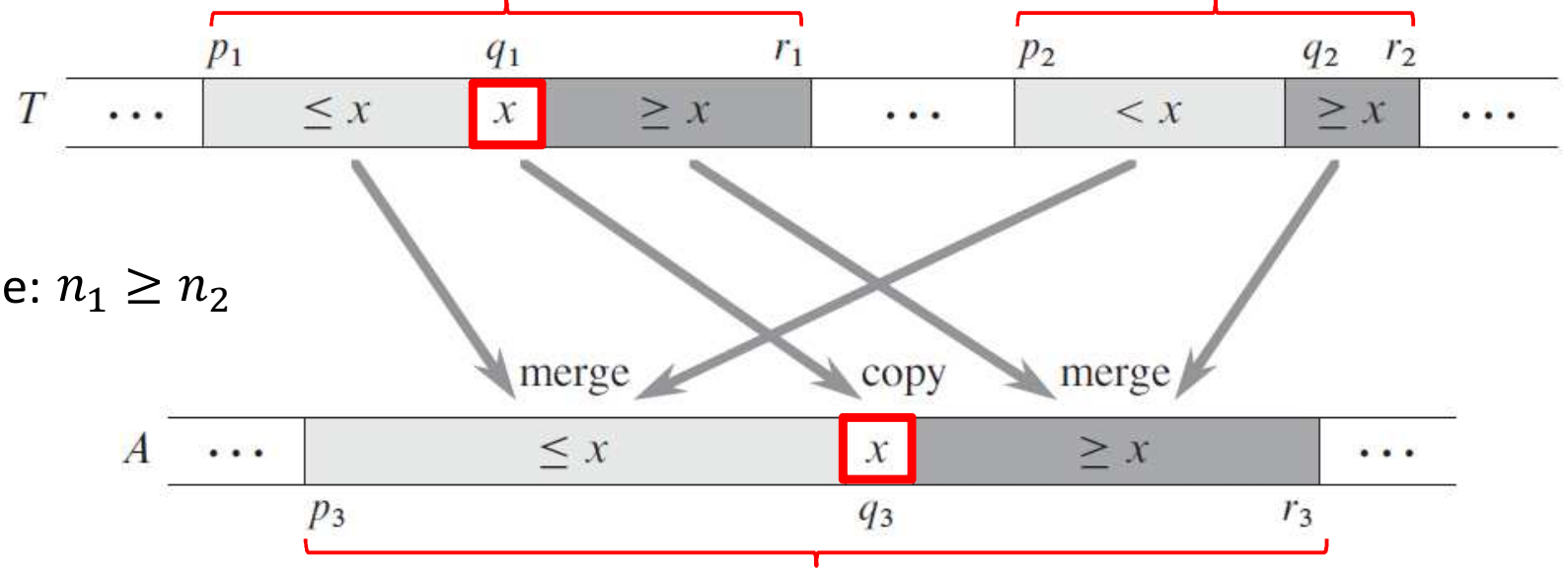
$n_3 = r_3 - p_3 + 1 = n_1 + n_2$

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

Parallel Merge

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

subarrays to merge: $T[p_1..r_1]$ $\quad$ $T[p_2..r_2]$



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

suppose: $n_1 \geq n_2$

merged output: $A[p_3..r_3]$

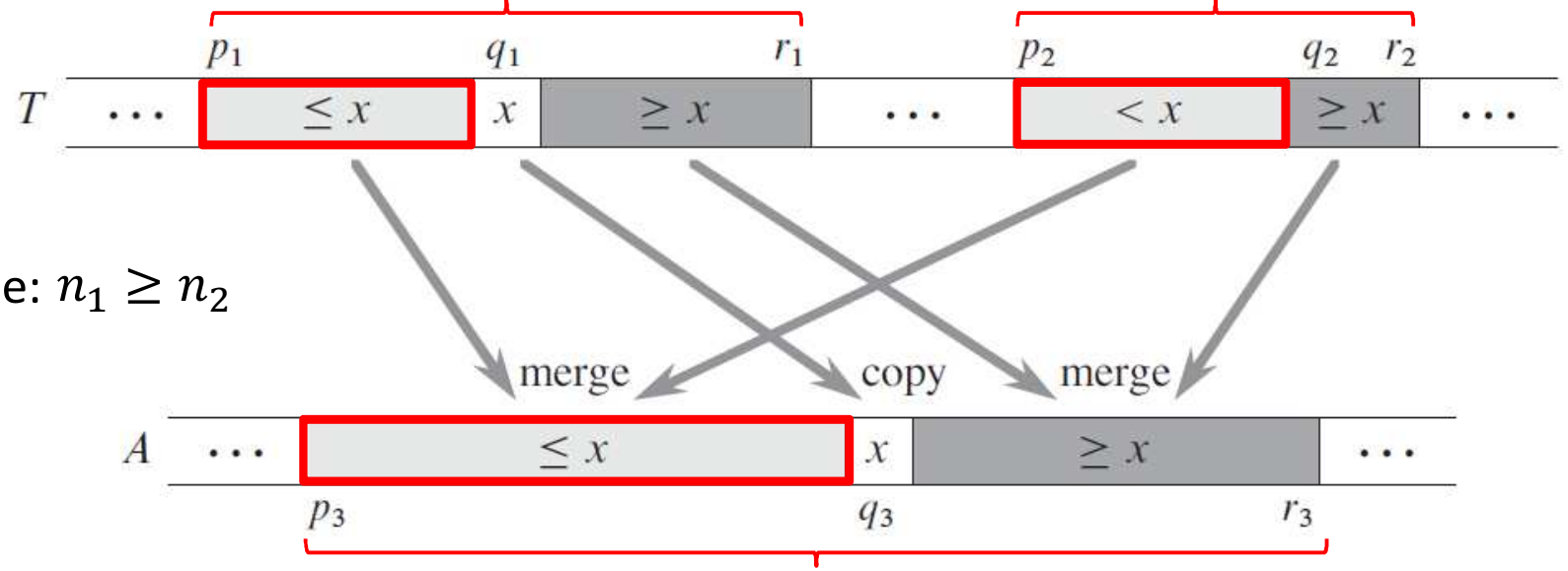
$n_3 = r_3 - p_3 + 1 = n_1 + n_2$

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

Parallel Merge

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

subarrays to merge: $T[p_1..r_1]$ $\quad$ $T[p_2..r_2]$



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

suppose: $n_1 \geq n_2$

merged output: $A[p_3..r_3]$

$n_3 = r_3 - p_3 + 1 = n_1 + n_2$

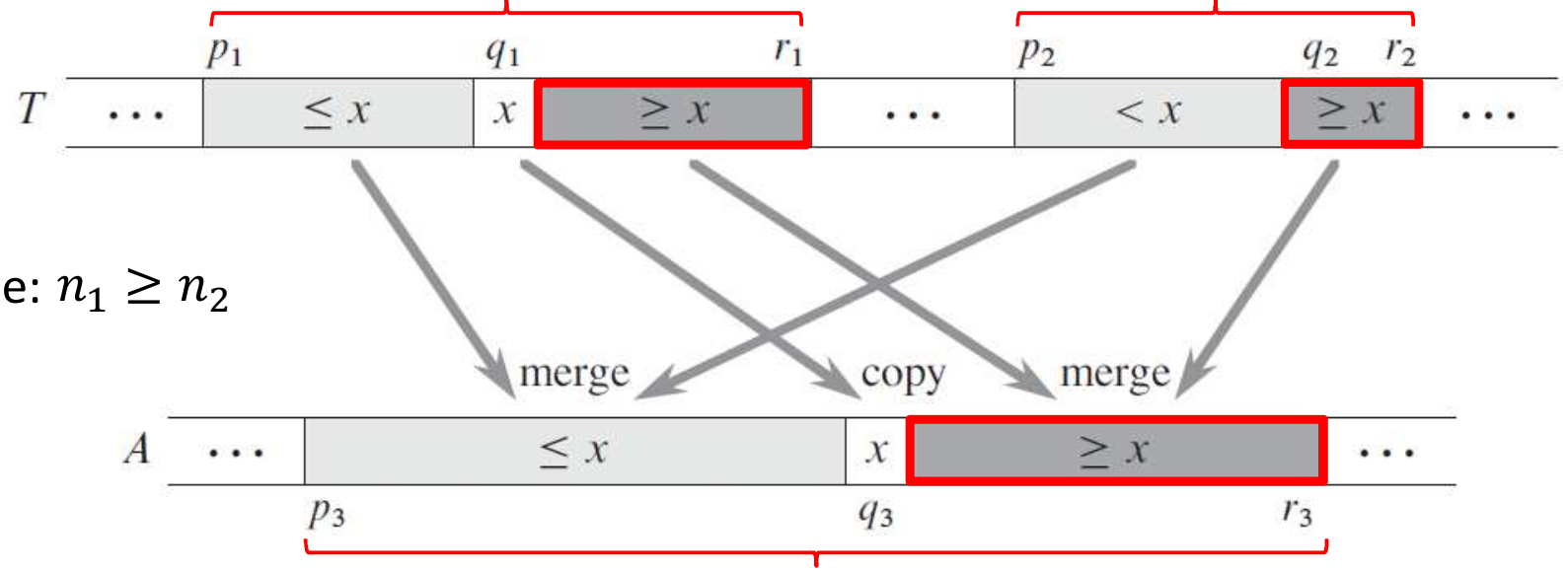
Perform the following two steps in parallel.

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

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

subarrays to merge: $T[p_1..r_1]$ $\quad$ $T[p_2..r_2]$



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

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 $A[p_3..q_3 - 1]$

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 ← r1 - p1 + 1,   n2 ← r2 - p2 + 1
 2. if n1 < n2 then
 3.     p1 ↔ p2,  r1 ↔ r2,  n1 ↔ n2
 4. if n1 = 0 then return
 5. else
 6.     q1 ← ⌊ ( p1 + r1 ) / 2 ⌋
 7.     q2 ← Binary-Search ( T[ q1 ],  T,  p2, r2 )
 8.     q3 ← p3 + ( q1 - p1 ) + ( q2 - p2 )
 9.     A[ q3 ] ← 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, p1, r1, p2, r2, A, p3 )
1.  n1 ← r1 - p1 + 1,   n2 ← r2 - p2 + 1
2.  if n1 < n2 then
3.      p1 ↔ p2,  r1 ↔ r2,  n1 ↔ n2
4.  if n1 = 0 then return
5.  else
6.      q1 ← ⌊ ( p1 + r1 ) / 2 ⌋
7.      q2 ← Binary-Search ( T[ q1 ],  T,  p2, r2 )
8.      q3 ← p3 + ( q1 - p1 ) + ( q2 - p2 )
9.      A[ q3 ] ← 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
```

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, p1, r1, p2, r2, A, p3 )
1.  n1 ← r1 - p1 + 1,   n2 ← r2 - p2 + 1
2.  if n1 < n2 then
3.      p1 ↔ p2,  r1 ↔ r2,  n1 ↔ n2
4.  if n1 = 0 then return
5.  else
6.      q1 ← ⌊ ( p1 + r1 ) / 2 ⌋
7.      q2 ← Binary-Search ( T[ q1 ],  T,  p2, r2 )
8.      q3 ← p3 + ( q1 - p1 ) + ( q2 - p2 )
9.      A[ q3 ] ← 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
```

Span:

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

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

Work:

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

We show below that, $T_1(n) = \mathrm{O}(n)$

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

$$T_1(n) = T_1(\alpha n) + T_1\big((1-\alpha)n\big) + \mathrm{O}(\log n)$$

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

Hence, $T_1(n) = \Theta(n)$.

Parallel Merge Sort with Parallel Merge

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

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

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

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

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

$= \Theta(\log^3 n)$ [MT Case 2]

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