CSE 548: Analysis of Algorithms

Lectures 22 & 23
(Analyzing Parallel Algorithms)

Rezaul A. Chowdhury

Department of Computer Science
SUNY Stony Brook
Fall 2012

Why Parallelism?

Unicore Performance Has Hit a Wall!

Some Reasons

- Lack of additional ILP
  (Instruction Level Hidden Parallelism)
- High power density
- Manufacturing issues
- Physical limits
- Memory speed

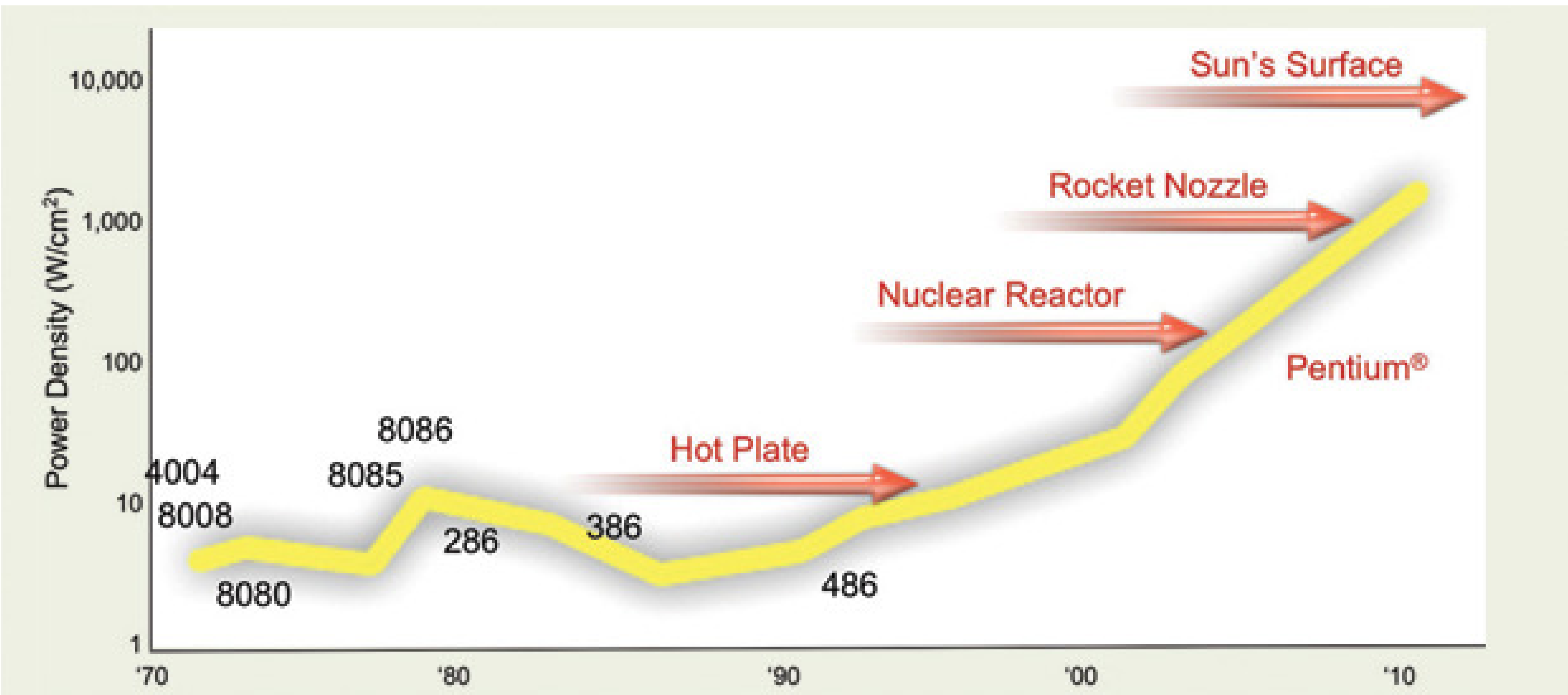
Unicore Performance: No Additional ILP

Exhausted all ideas to exploit hidden parallelism?

- Multiple simultaneous instructions
- Dynamic instruction scheduling
- Branch prediction
- Out-of-order instructions
- Speculative execution
- Pipelining
- Non-blocking caches, etc.

Unicore Performance: High Power Density

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



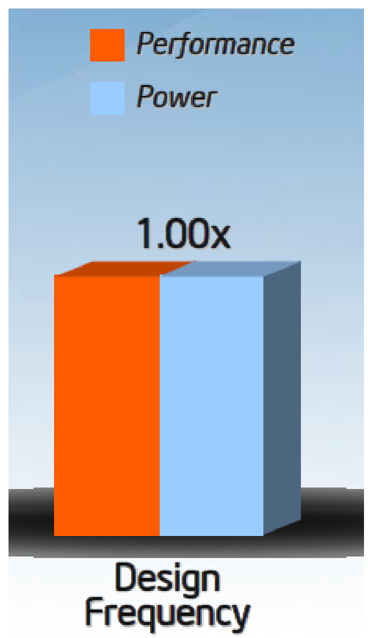
Source: Patrick Gelsinger, Intel Developer Forum, Spring 2004 (Simon Floyd)

Unicore Performance: High Power Density

– Changing f by 20% changes performance by 13%

– So what happens if we overclock by 20%?

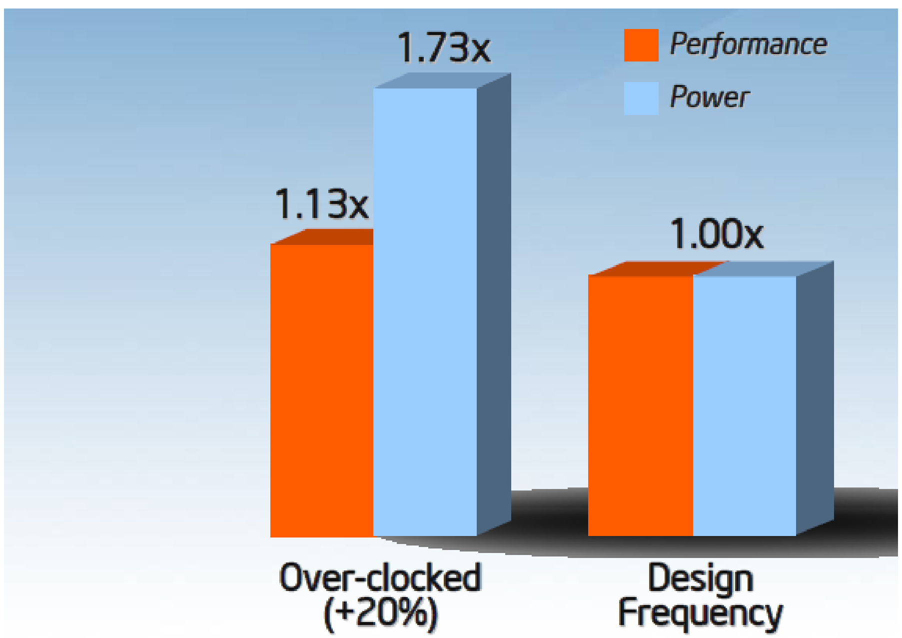
– And underclock by 20%?



Source: Andrew A. Chien, Vice President of Research, Intel Corporation

Unicore Performance: High Power Density

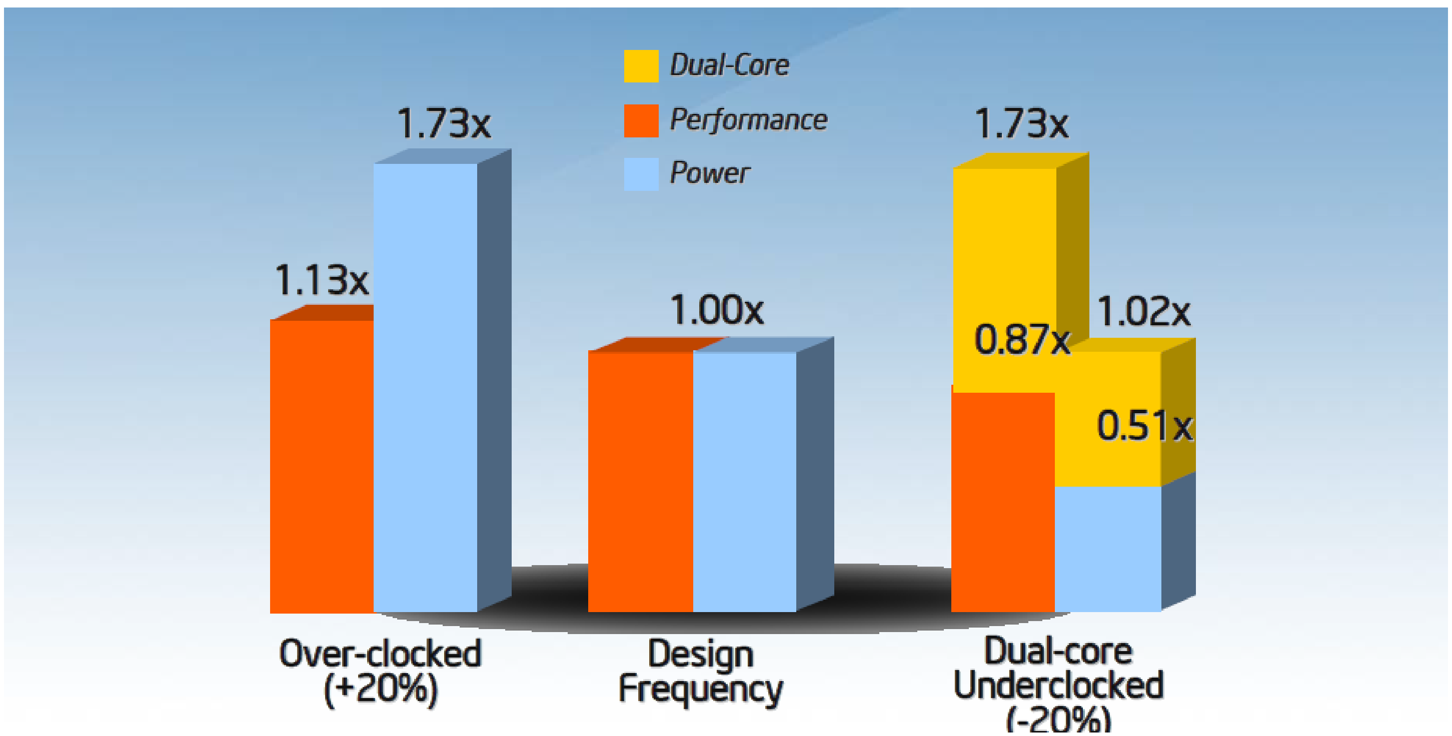
- Changing f by 20% changes performance by 13%
- So what happens if we overclock by 20%?
- And underclock by 20%?



Source: Andrew A. Chien, Vice President of Research, Intel Corporation

Unicore Performance: High Power Density

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



Source: Andrew A. Chien, Vice President of Research, Intel Corporation

Unicore Performance: Manufacturing Issues

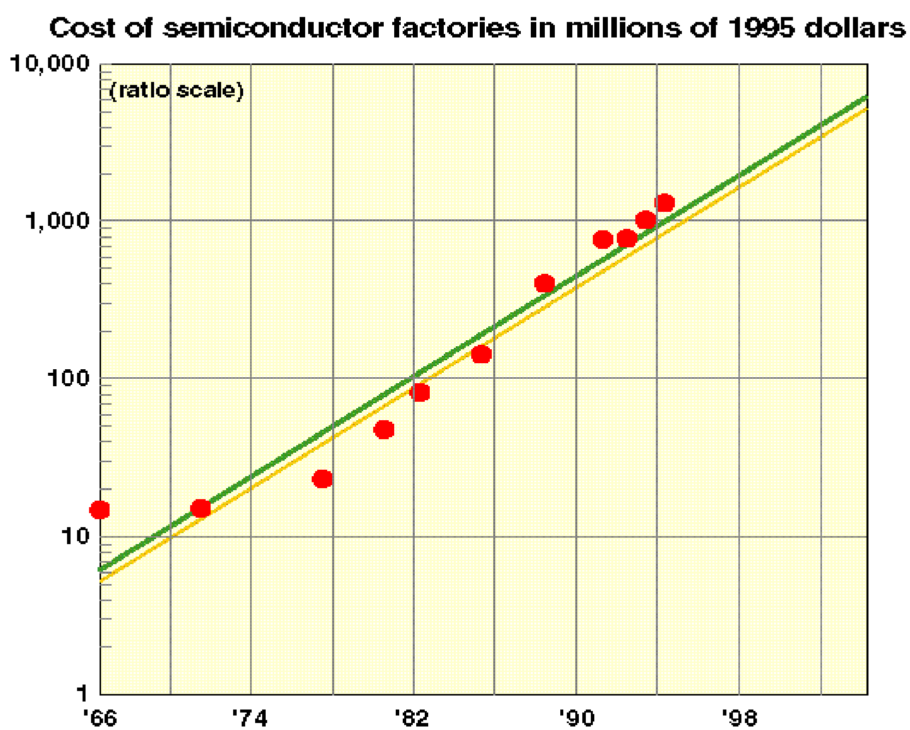
- Frequency, $f \propto 1 / s$
    - *s = feature size (transistor dimension)*
- Transistors / unit area $\propto 1 / s^2$
- Typically, die size $\propto 1 / s$
- So, what happens if feature size goes down by a factor of x?
    - Raw computing power goes up by a factor of x^4 !
    - Typically most programs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

Source: Kathy Yelick and Jim Demmel, UC Berkeley

Unicore Performance: Manufacturing Issues

As feature size decreases

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



Source: Kathy Yelick and Jim Demmel, UC Berkeley

Unicore Performance: Physical Limits

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

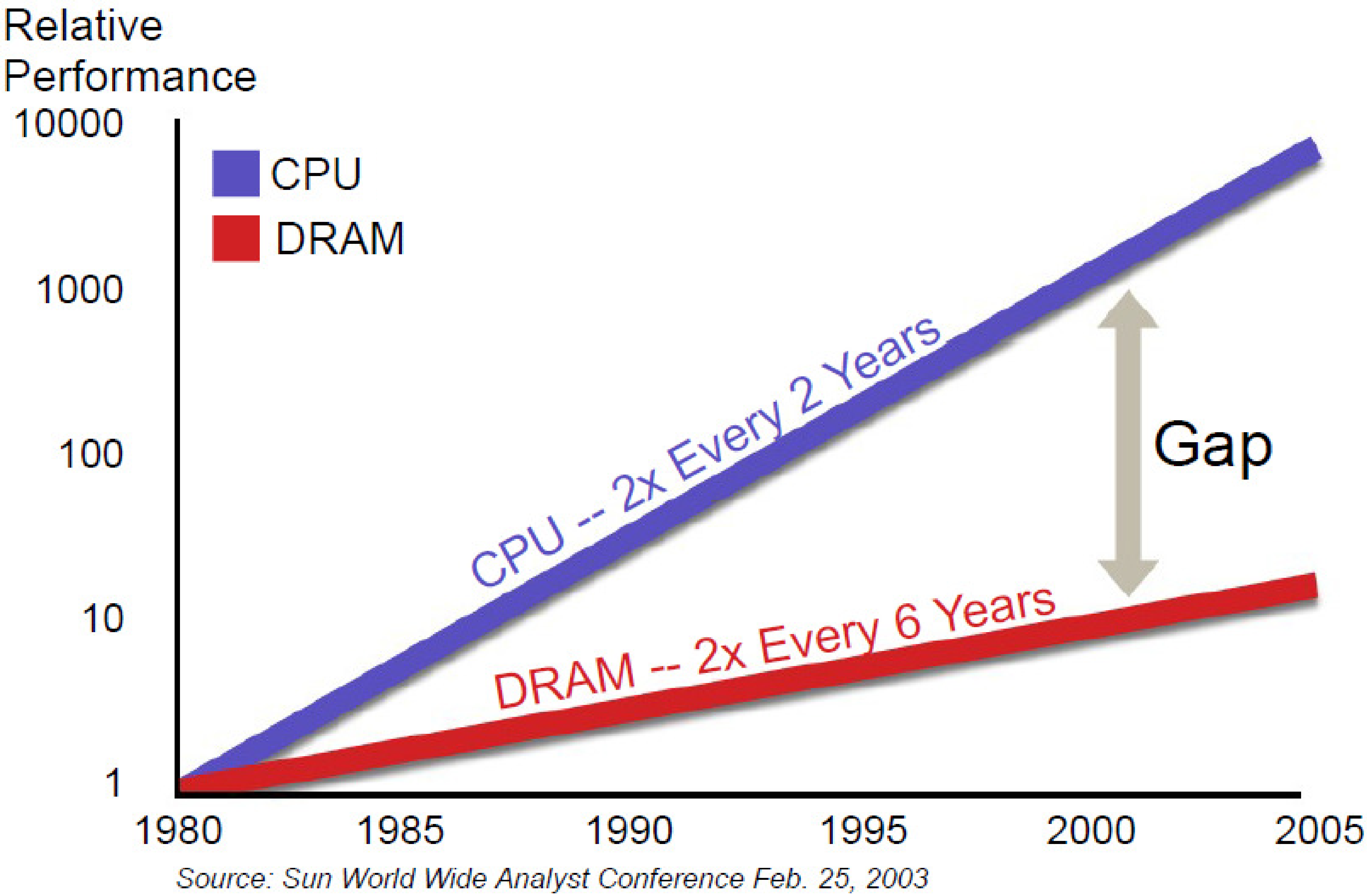
$$\textit{for}\ (\ i = 0;\ i < 10^{12};\ ++i\ )$$
$$z[\ i\ ] = x[\ i\ ] + y[\ i\ ];$$

- We will have to access 3×10^{12} data items in one second
- Speed of light is, $c \approx 3 \times 10^{8}$ m/s
- So each data item must be within $c\ /\ 3 \times 10^{12} \approx 0.1$ mm from the CPU on the average
- All data must be put inside a 0.2 mm × 0.2 mm square
- Each data item (≥ 8 bytes) can occupy only 1 Å^2 space!

  (size of a small atom!)

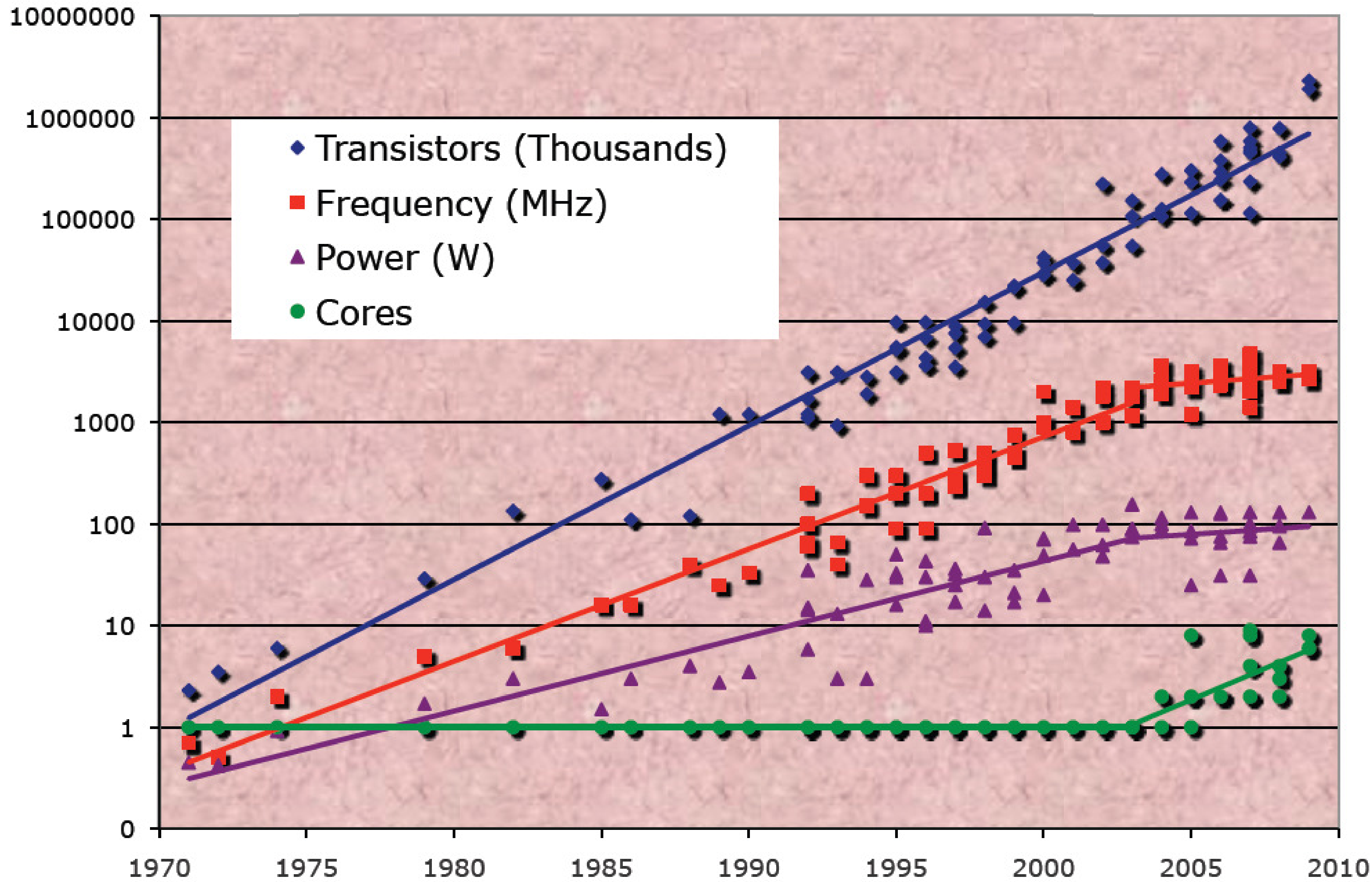
Source: Kathy Yelick and Jim Demmel, UC Berkeley

Unicore Performance: Memory Wall



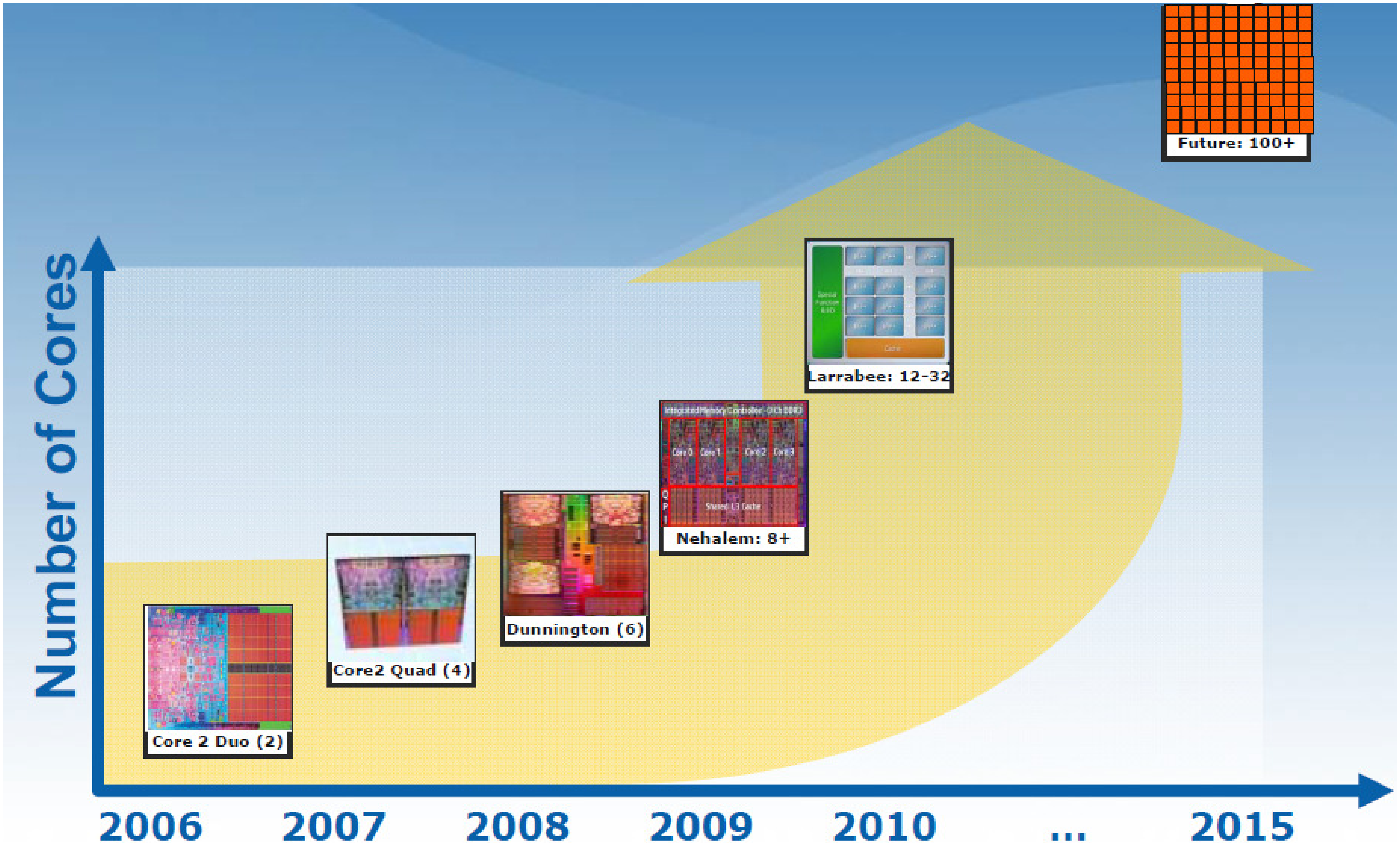
Source: Rick Hetherington, Chief Technology Officer, Microelectronics, Sun Microsystems

Moore's Law Reinterpreted



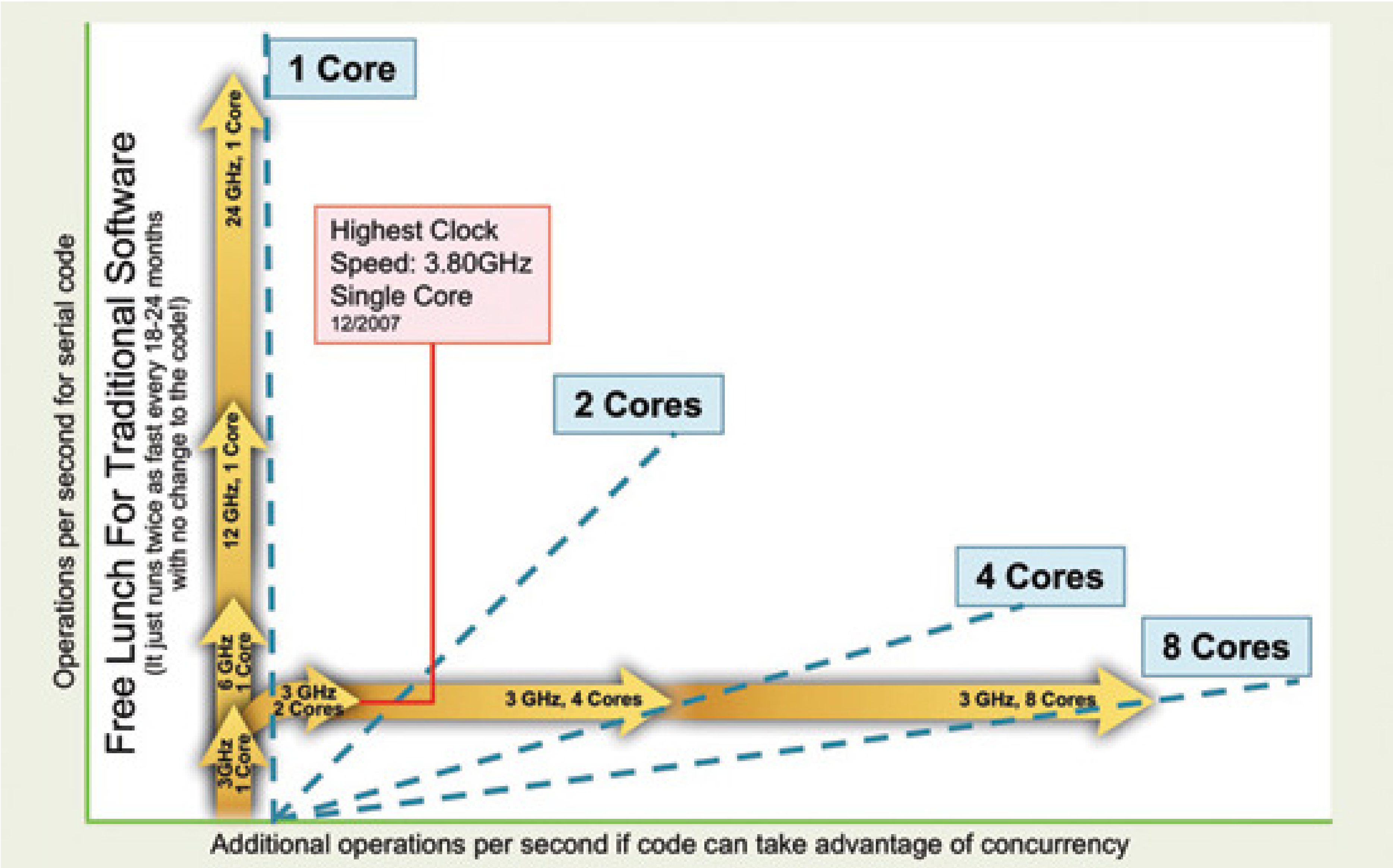
Source: Report of the 2011 Workshop on Exascale Programming Challenges

Cores / Processor (General Purpose)



Source: Andrew A. Chien, Vice President of Research, Intel Corporation

No Free Lunch for Traditional Software



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

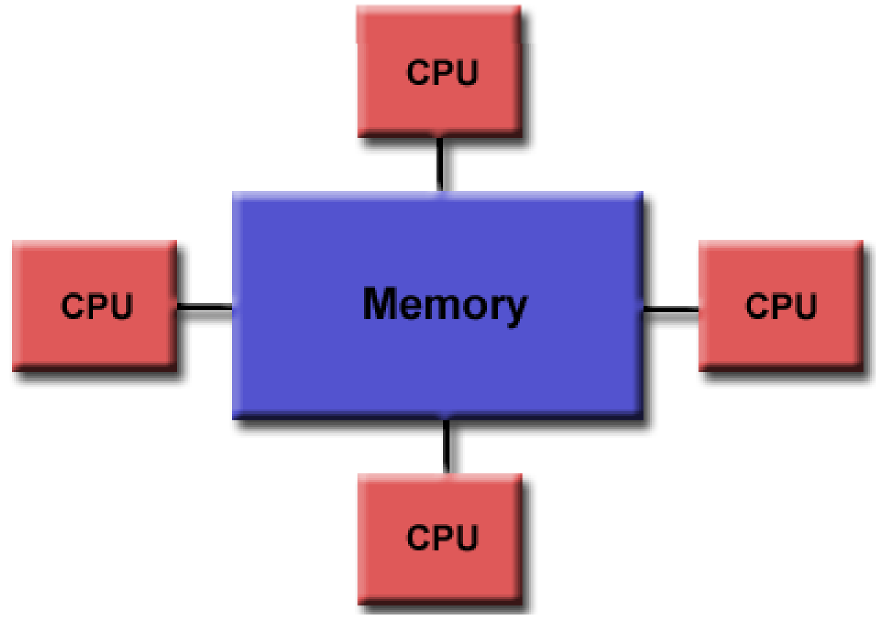
Insatiable Demand for Performance

Source: Patrick Gelsinger, Intel Developer Forum, 2008

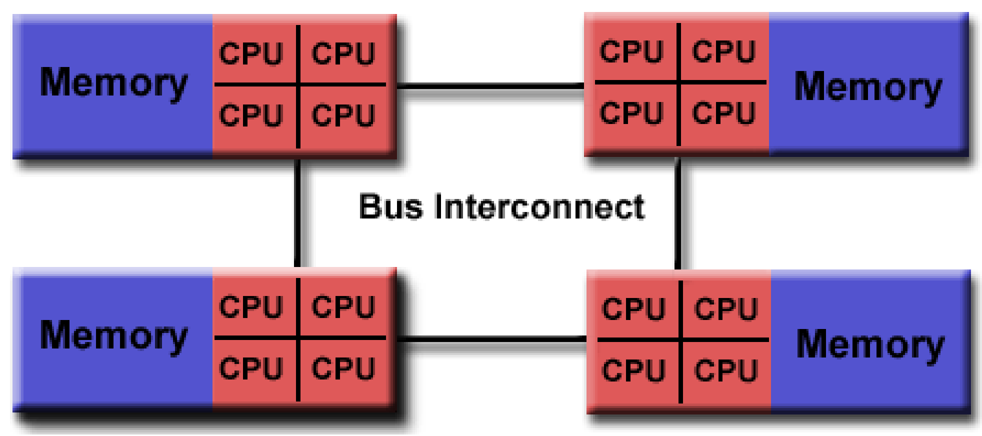
Some Useful Classifications of Parallel Computers

Parallel Computer Memory Architecture (Shared Memory)

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



UMA



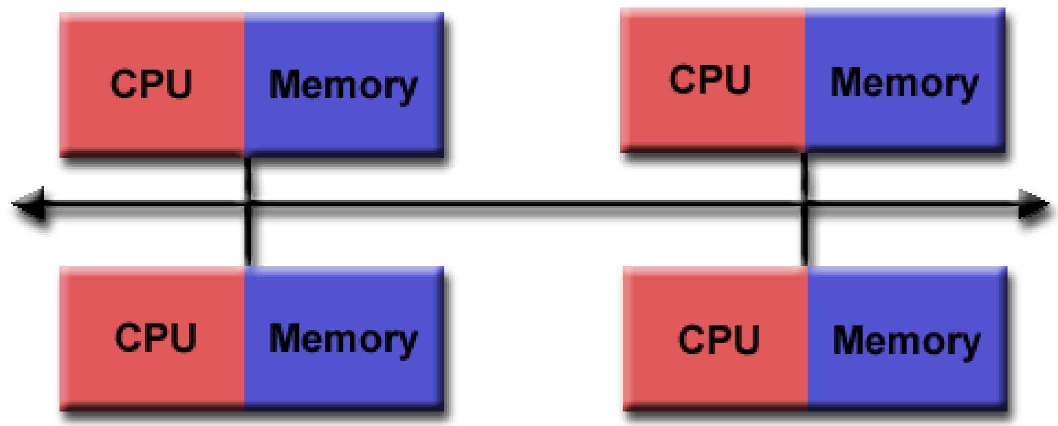
NUMA

Source: Blaise Barney, LLNL

Parallel Computer Memory Architecture (Distributed Memory)

– Each processor has its own local memory — no global address space

– Changes in local memory by one processor have no effect on memory of other processors

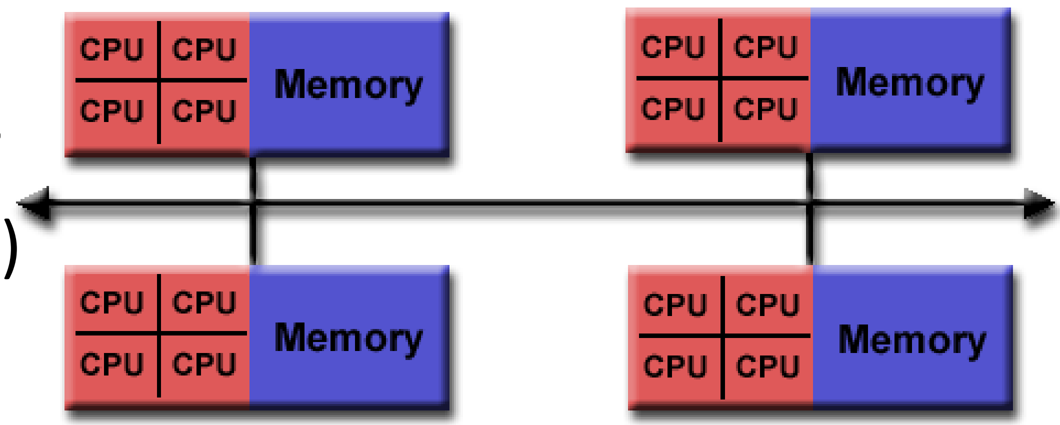


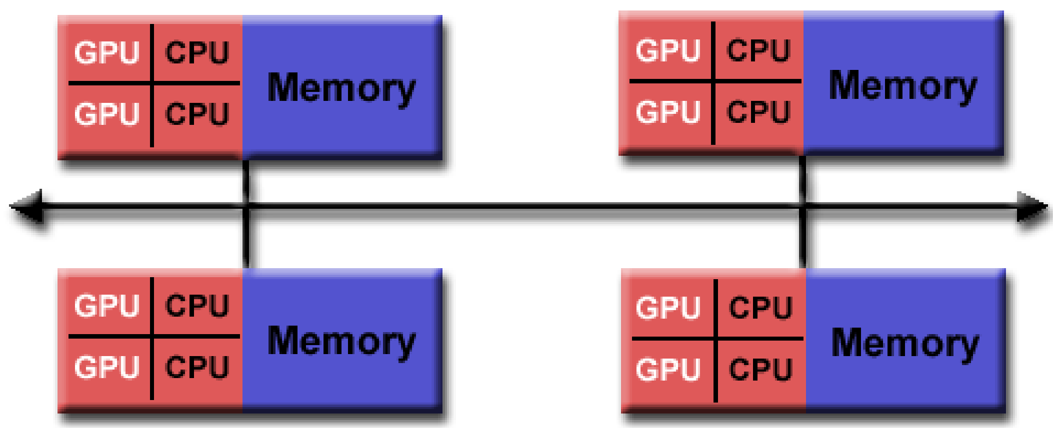
Source: Blaise Barney, LLNL

– Communication network to connect inter-processor memory

Parallel Computer Memory Architecture (Hybrid Distributed-Shared Memory)

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





Source: Blaise Barney, LLNL

Analyzing Parallel Algorithms

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

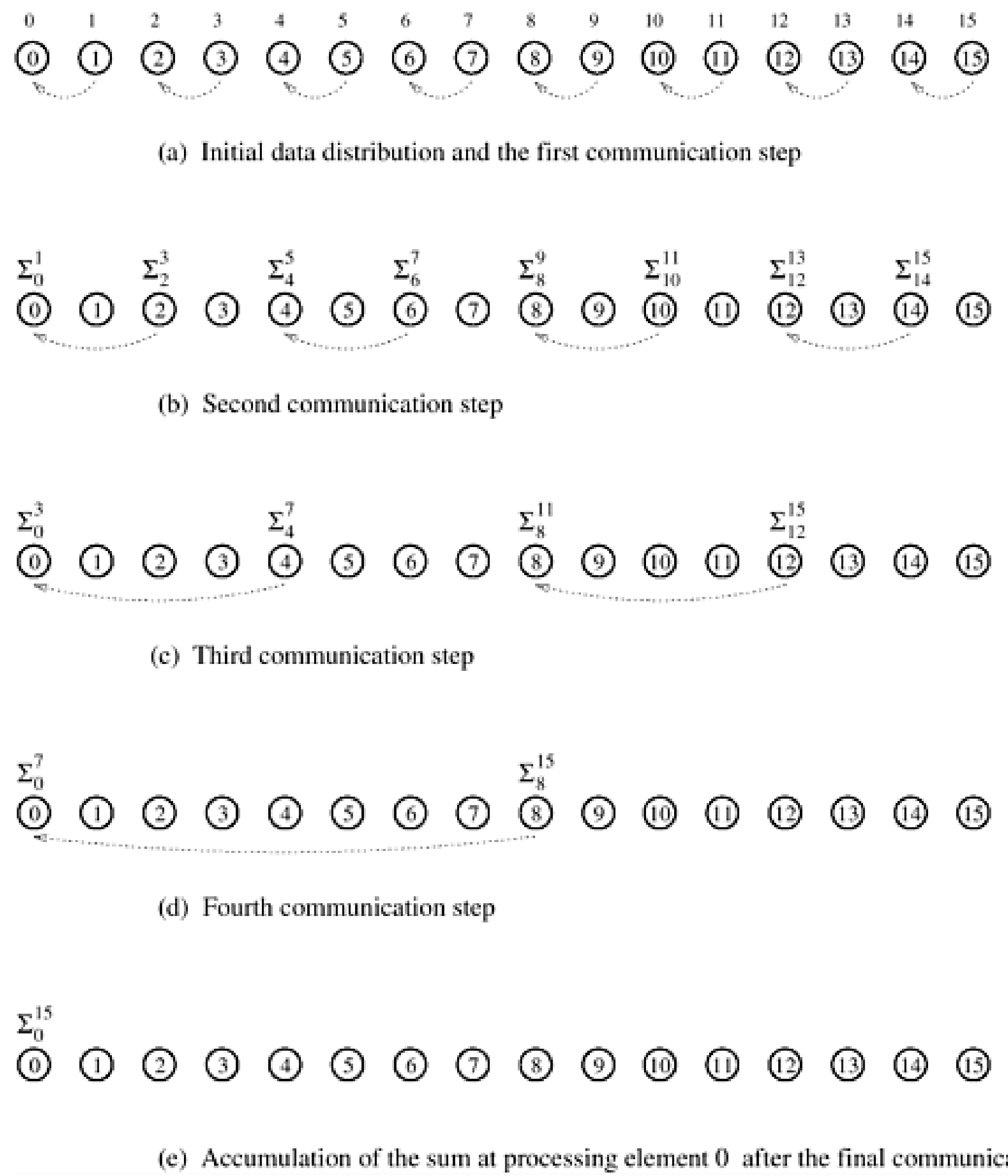
Consider adding n numbers using n identical processing elements.

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

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

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

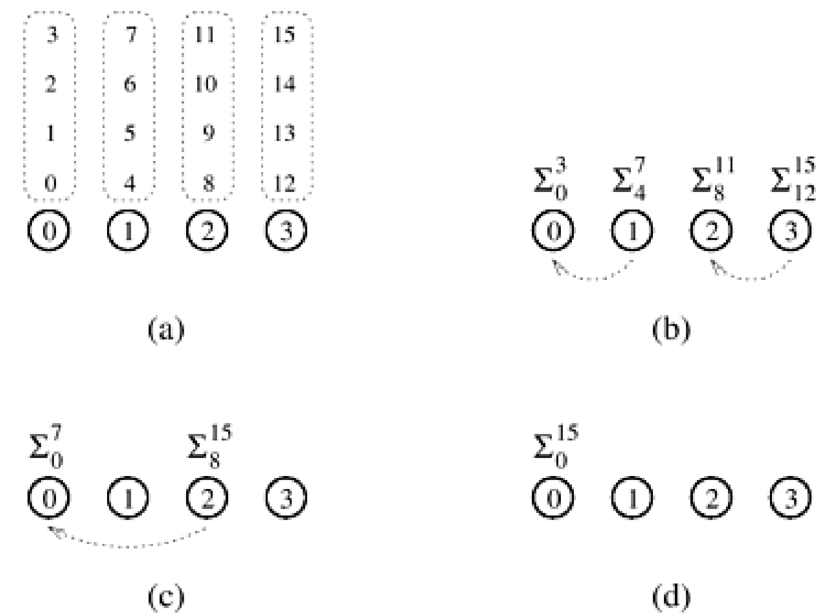
A parallel algorithm is *cost-optimal* or *work-optimal* provided

$$pT_p = \Theta(T_s)$$

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 $\Theta\left(\frac{n}{p}\right)$.

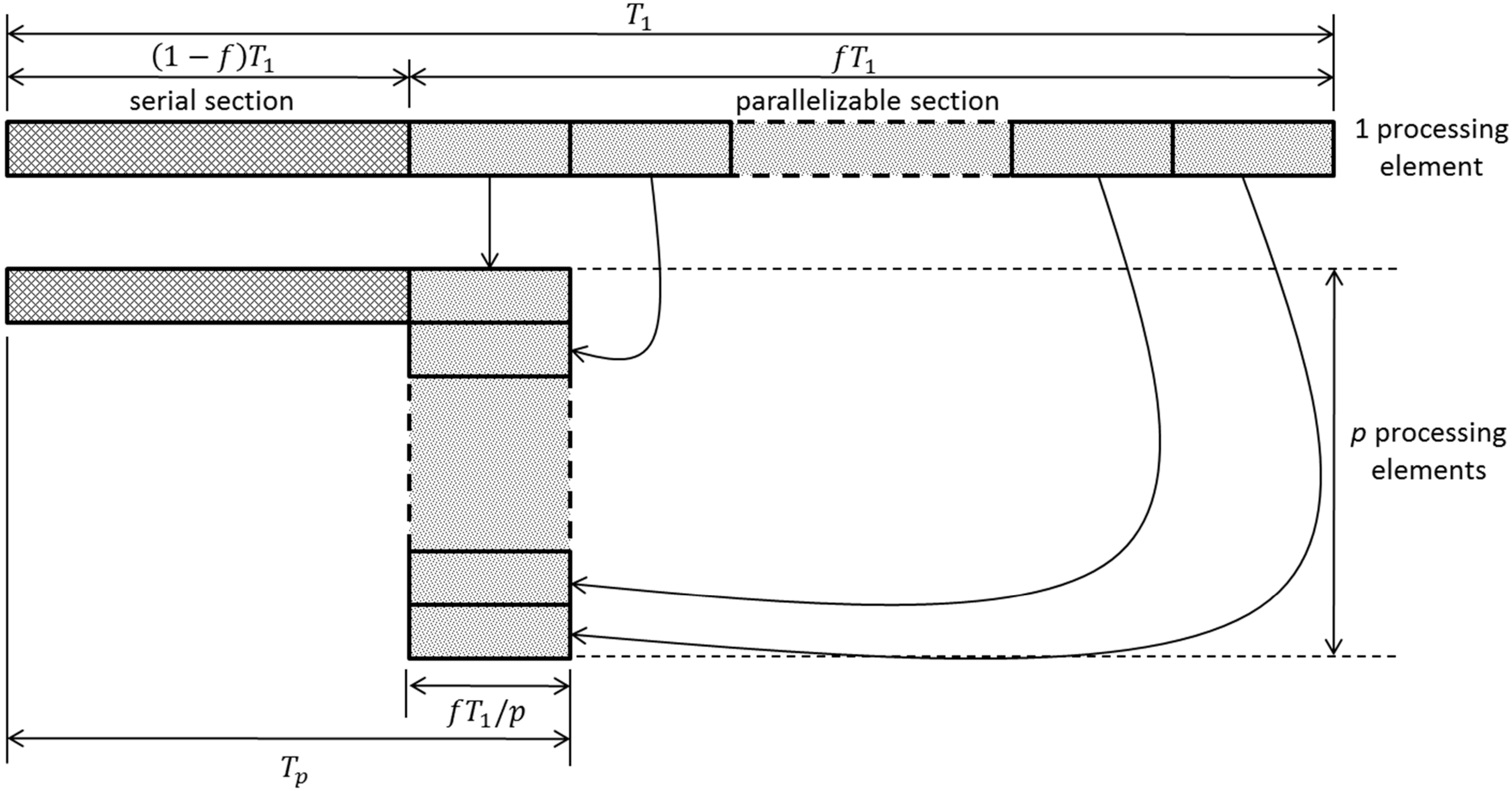
Then p processing elements adds these p partial sums in time $\Theta(\log p)$.

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.

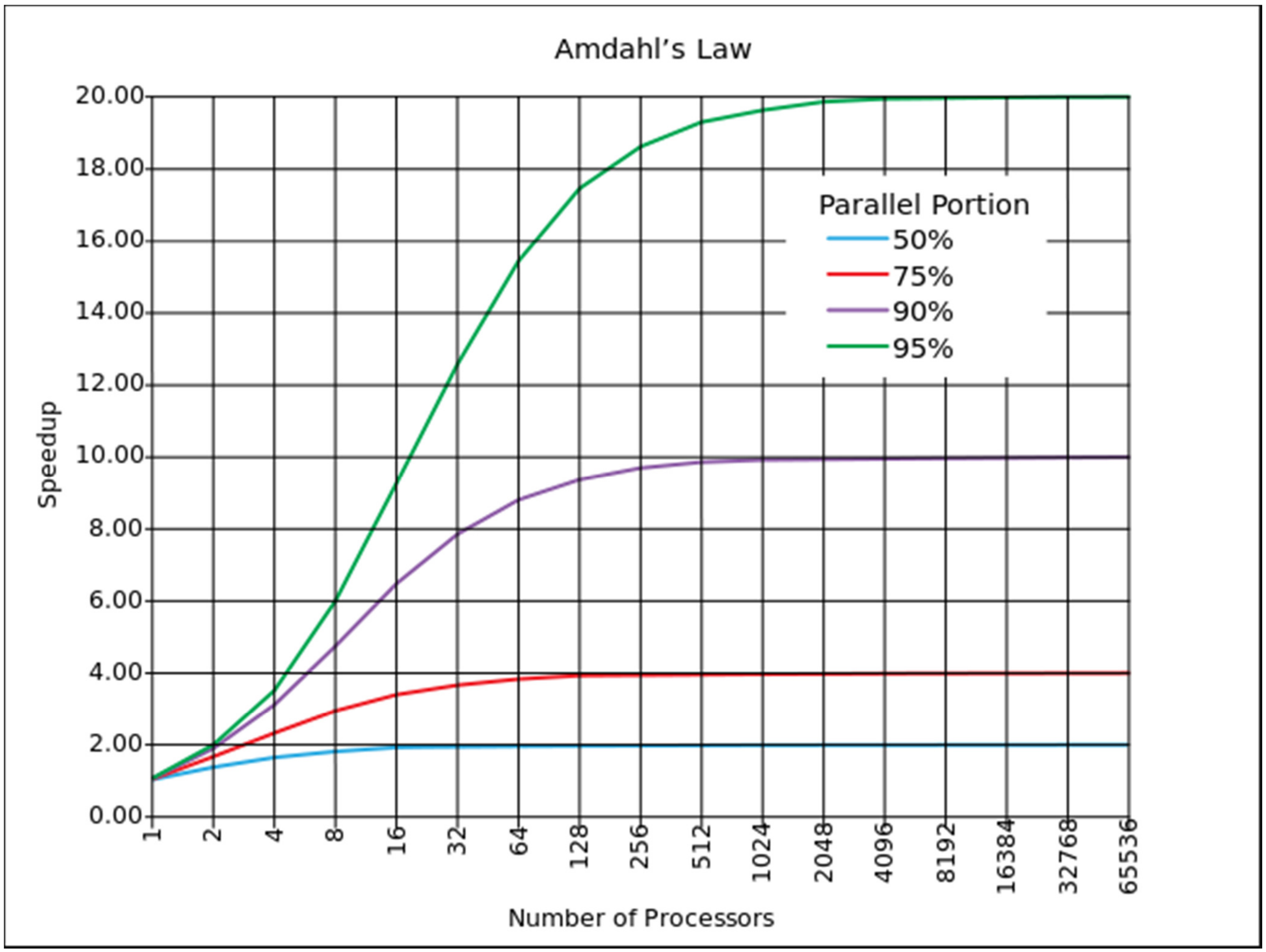
Then parallel running time, $T_p \geq (1-f)T_1 + f\,\frac{T_1}{p}$

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)

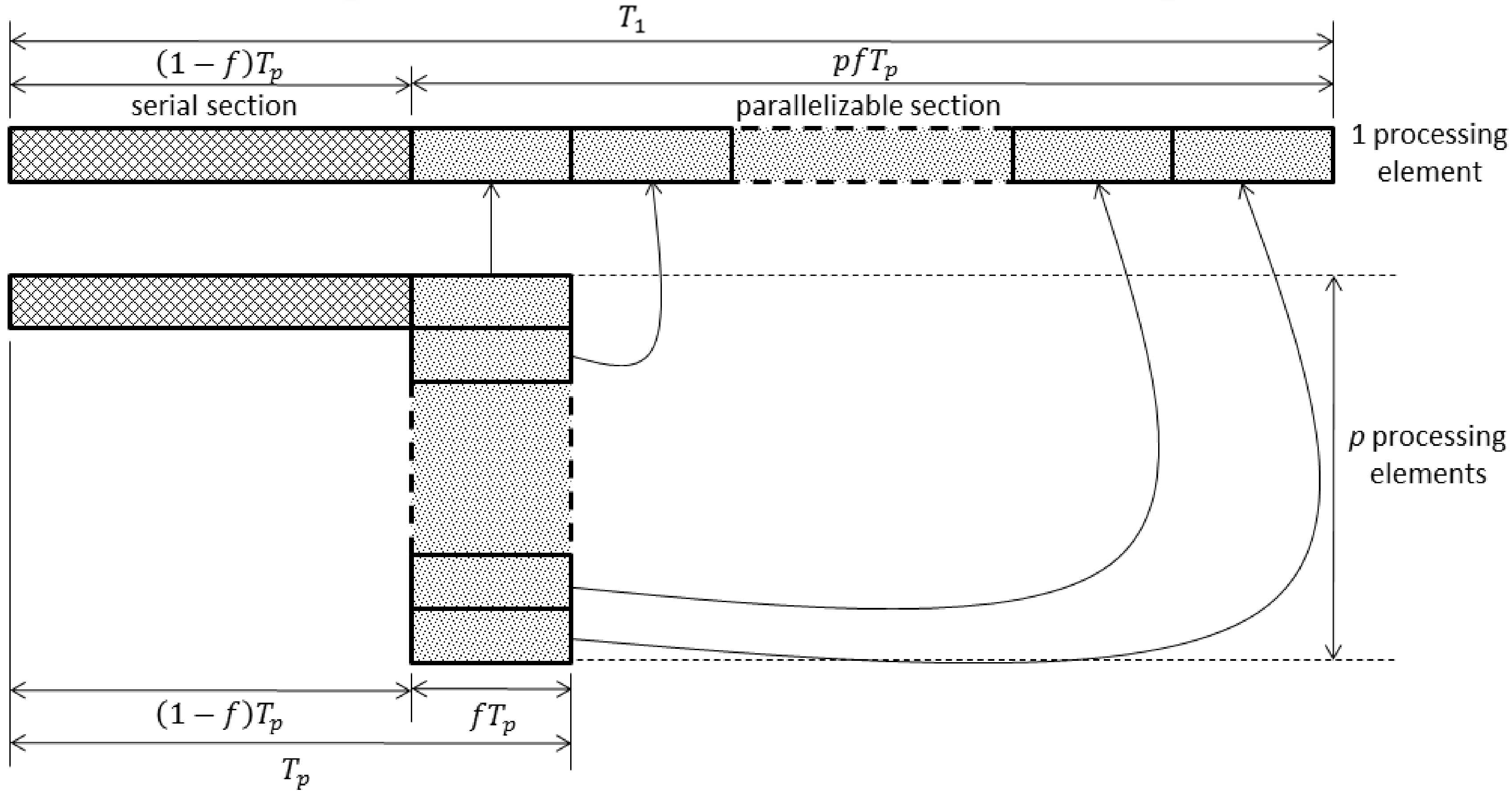
Suppose only a fraction f of a computation can be parallelized.

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.

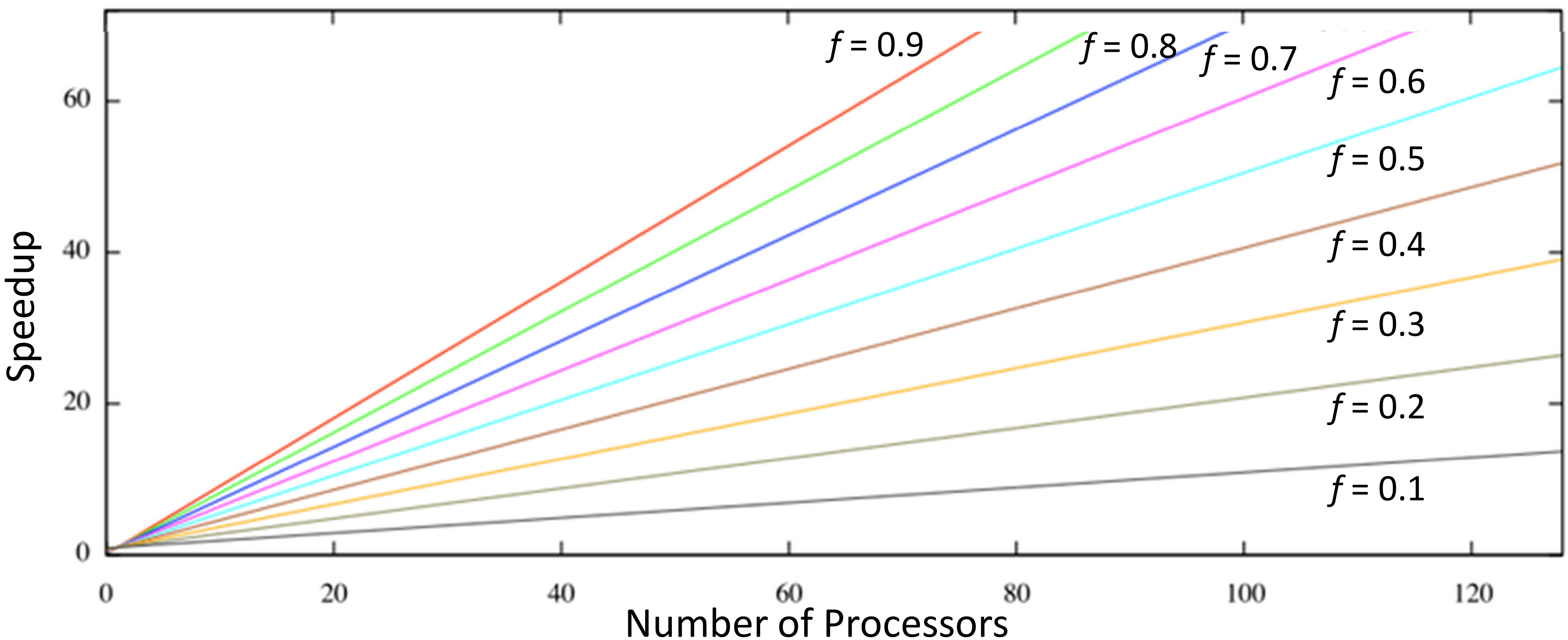
Then serial running time, $T_1 = (1-f)T_p + pfT_p$

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

Greedy Scheduling Theorem

Nested 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.

```
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 Code

Loop Parallelism

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \xrightarrow[\text{transpose}]{\text{in-place}} \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix}$$

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

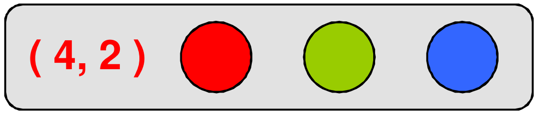
Parallel Execution Model

```
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

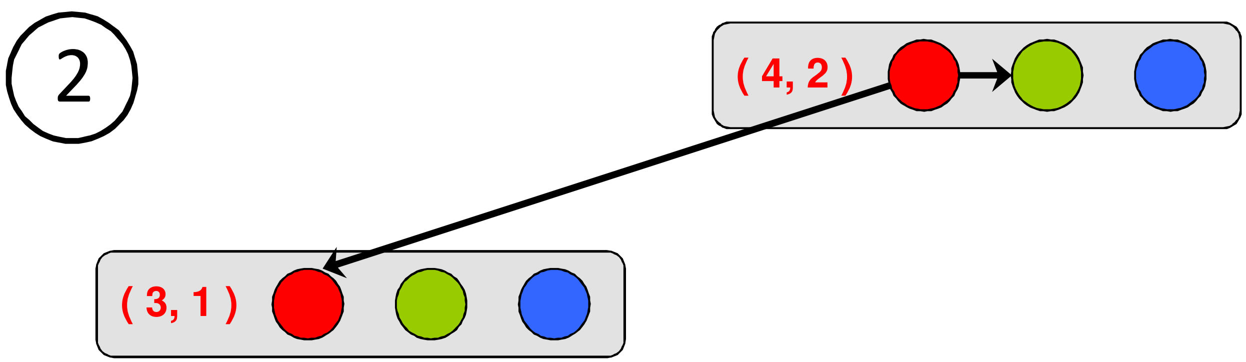
```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

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

  sync;

  return ( x + y );
}
```

1



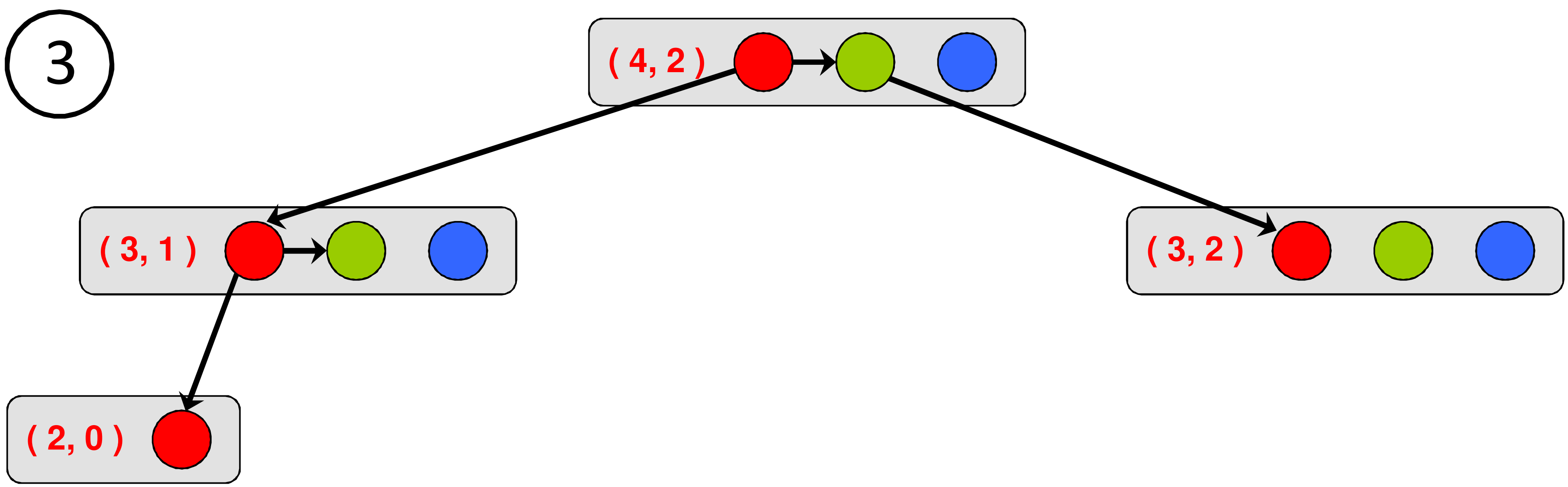
Parallel Execution Model

```
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

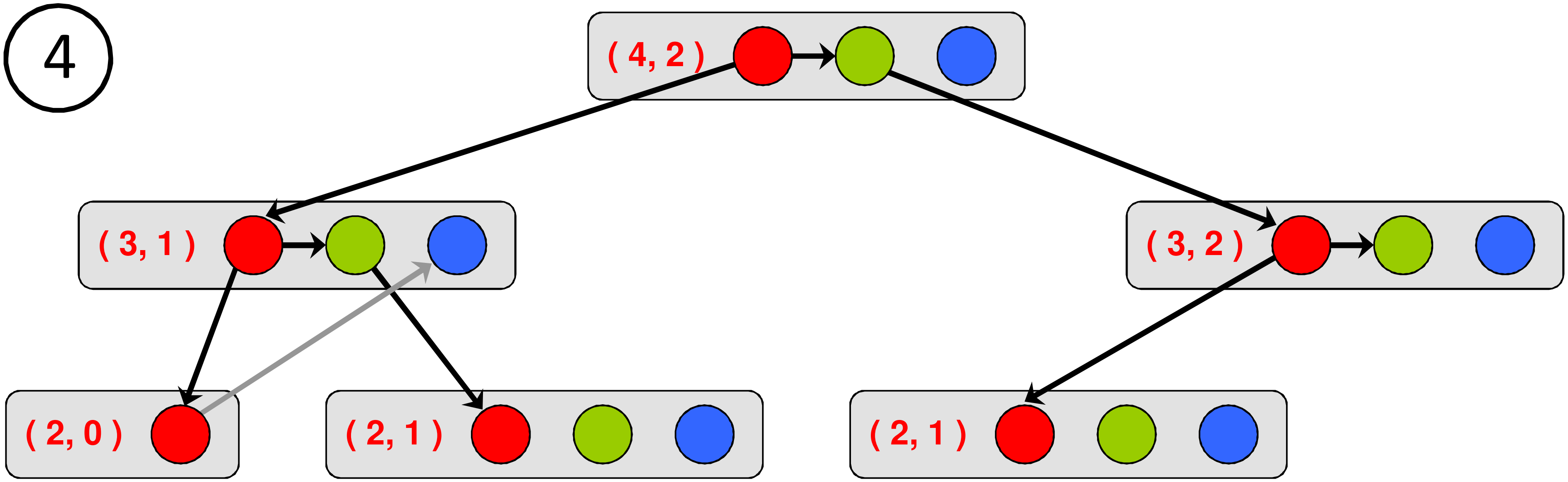
```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

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

  sync;

  return ( x + y );
}
```



Parallel Execution Model

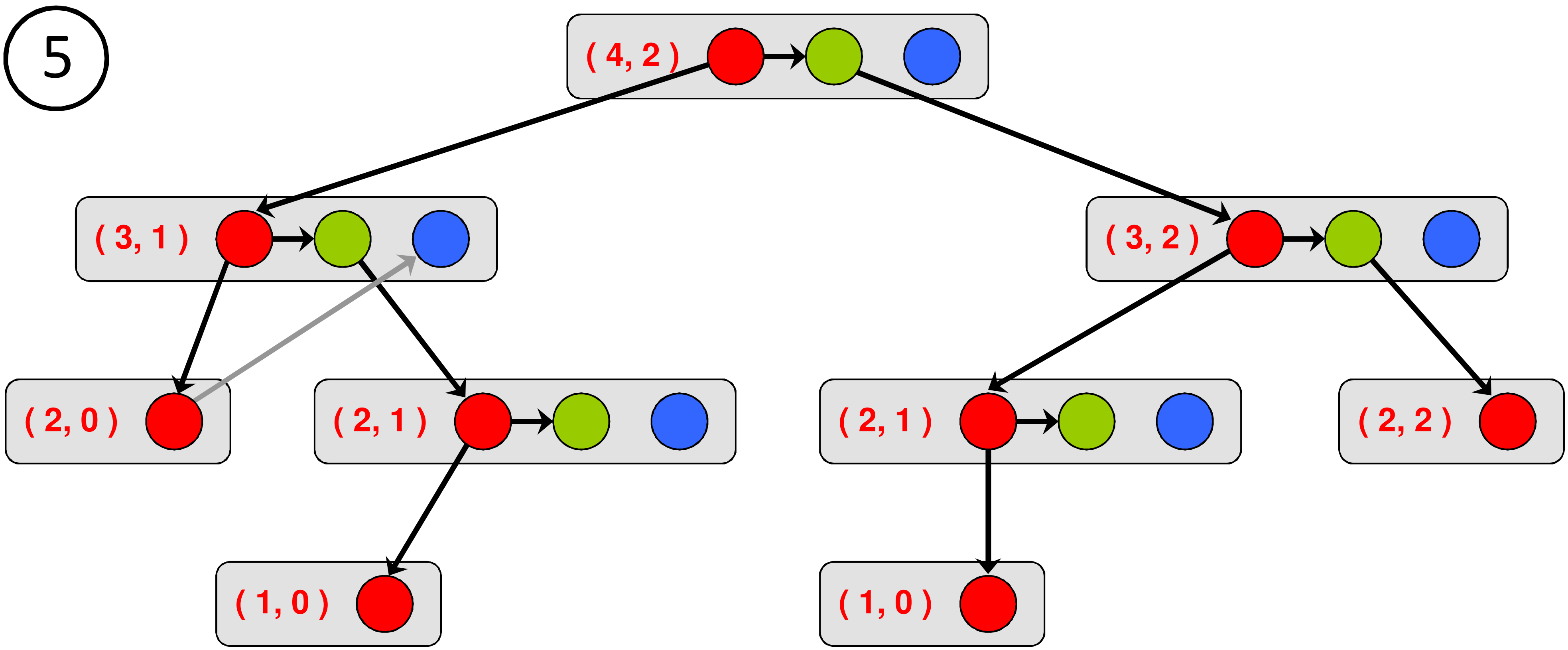
```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

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

  sync;

  return ( x + y );
}
```

4



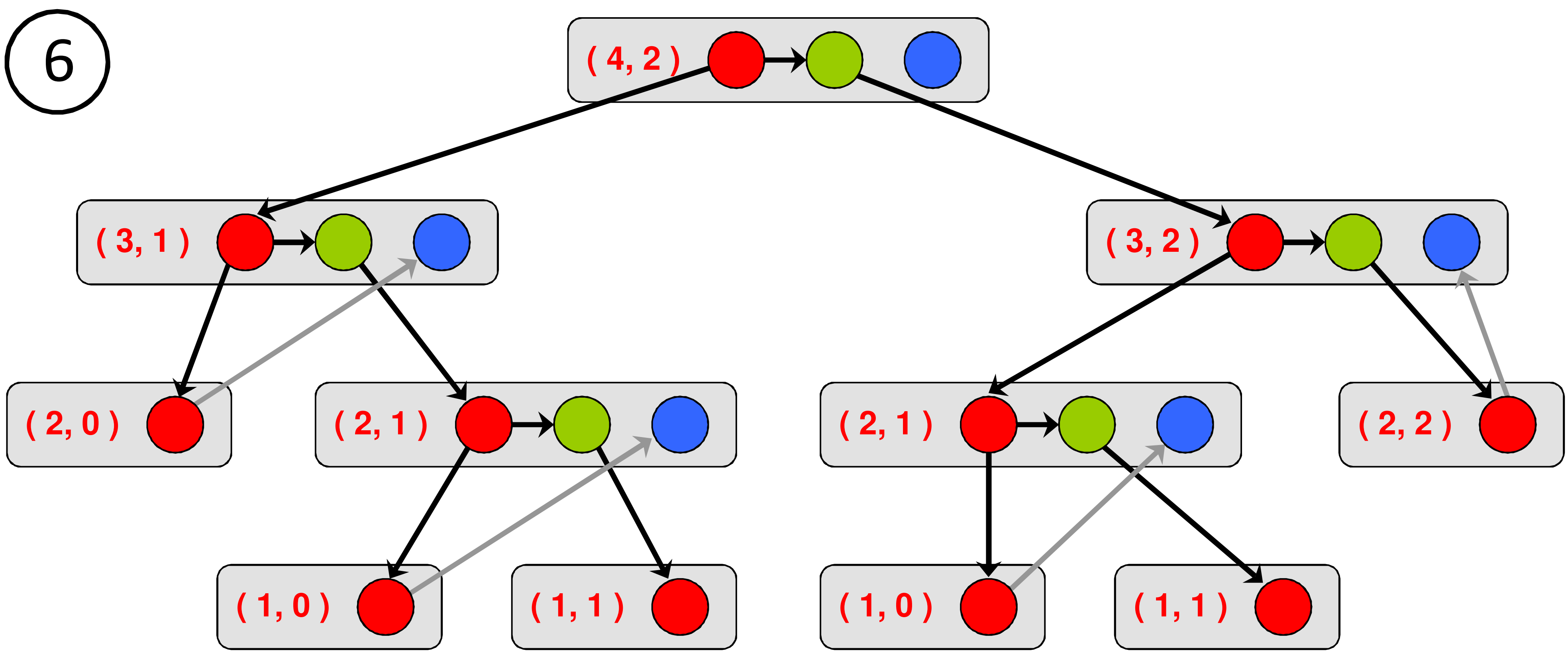
Parallel Execution Model

```
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

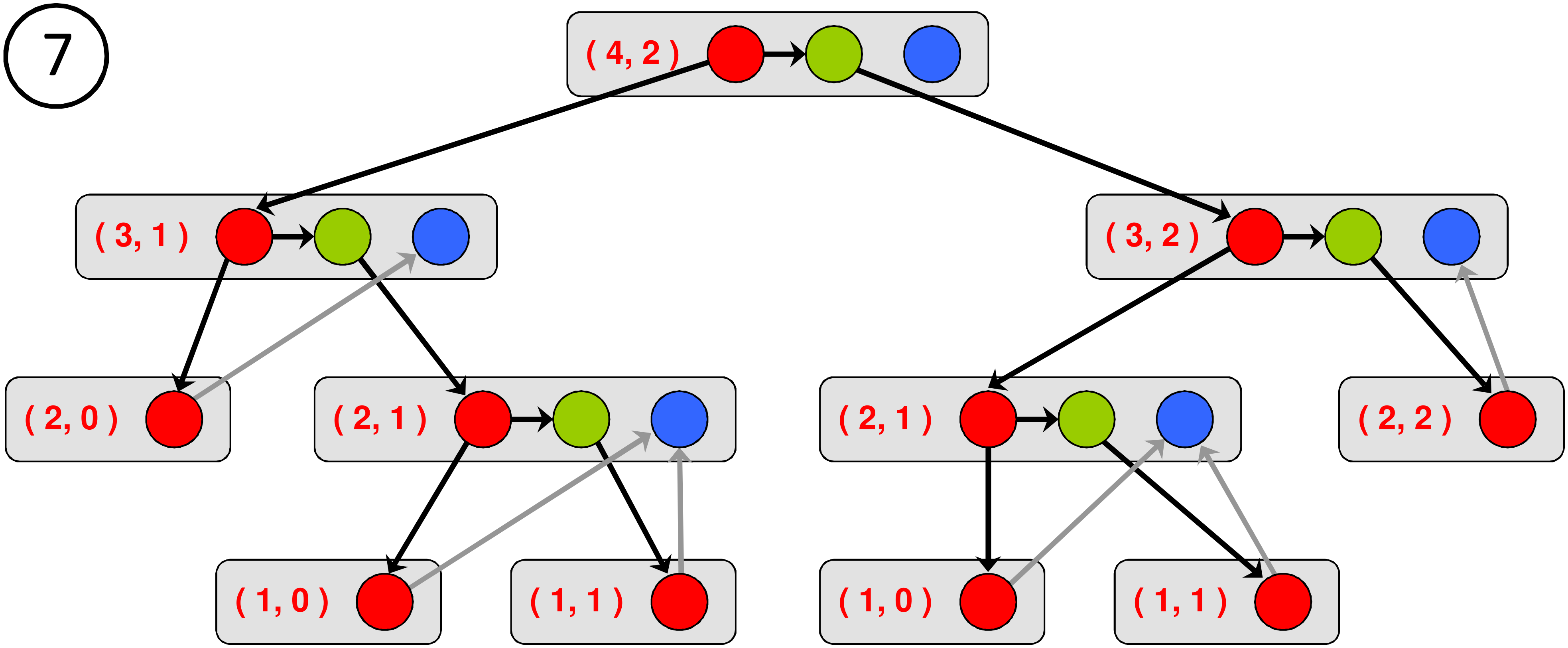
```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;
  int x, y;
  x = spawn comb( n – 1, r – 1 );
  y = comb( n – 1, r );
  sync;
  return ( x + y );
}
```



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 );
}
```

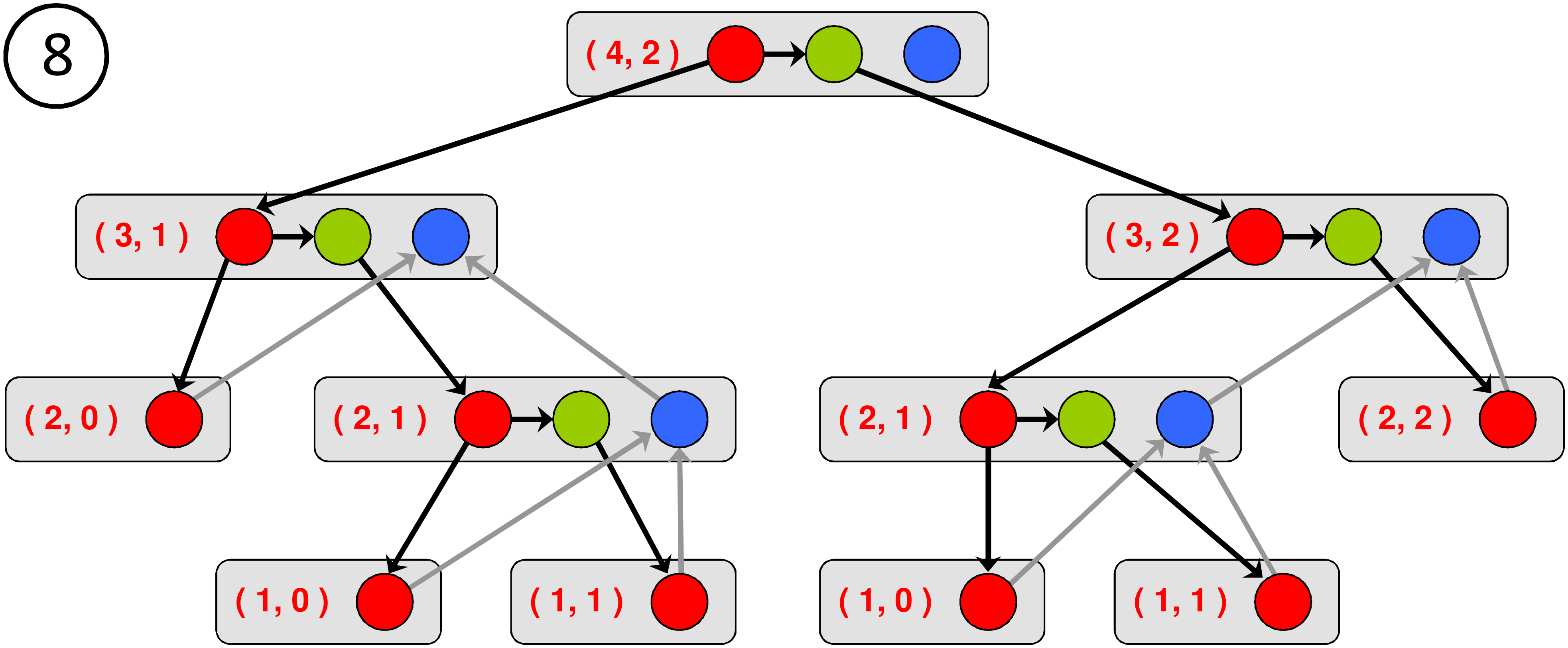
7



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 );
}
```

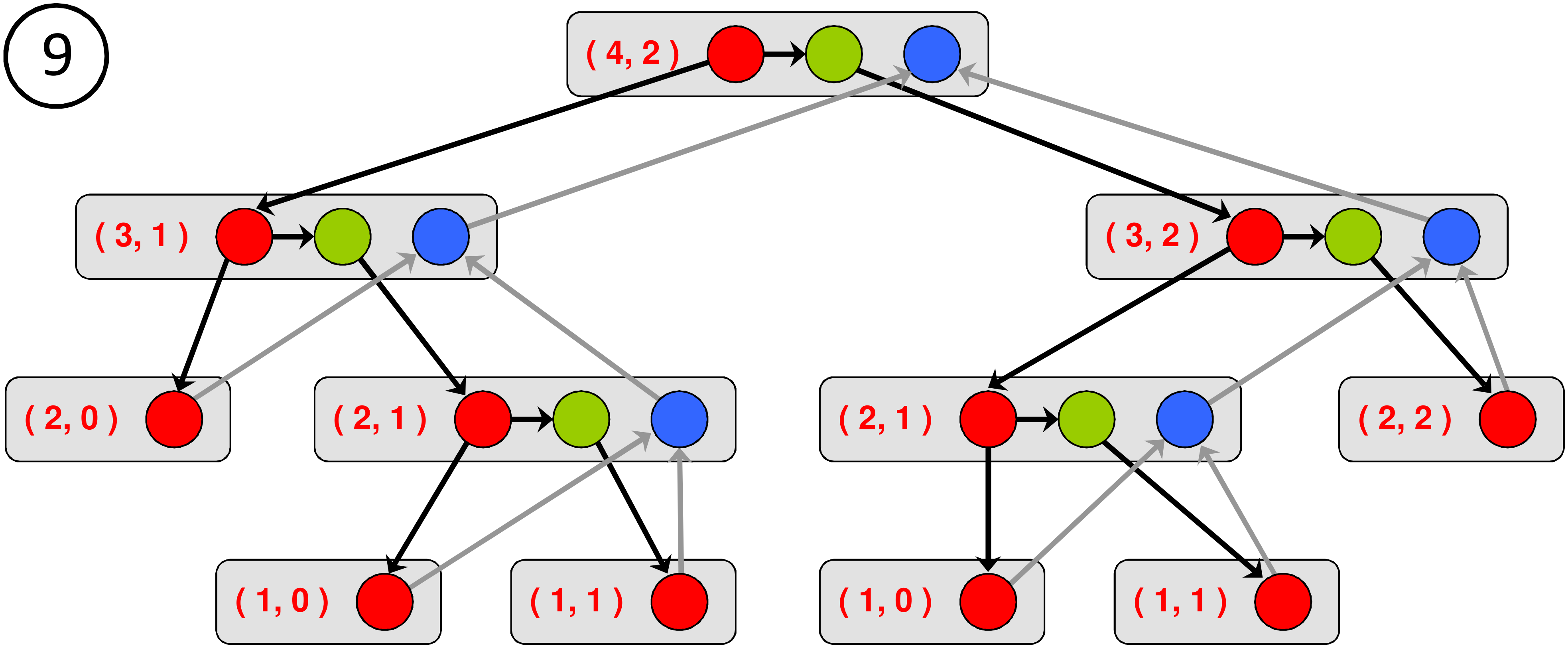
8



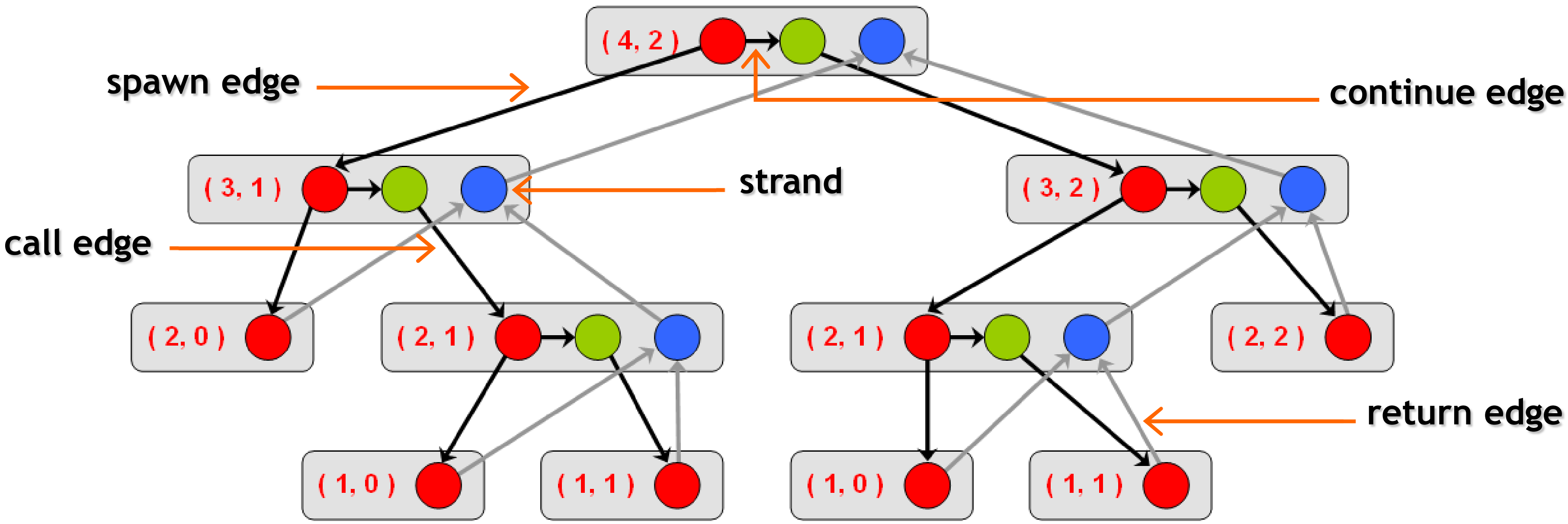
Parallel Execution Model

```
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 );
}
```

9



Computation DAG

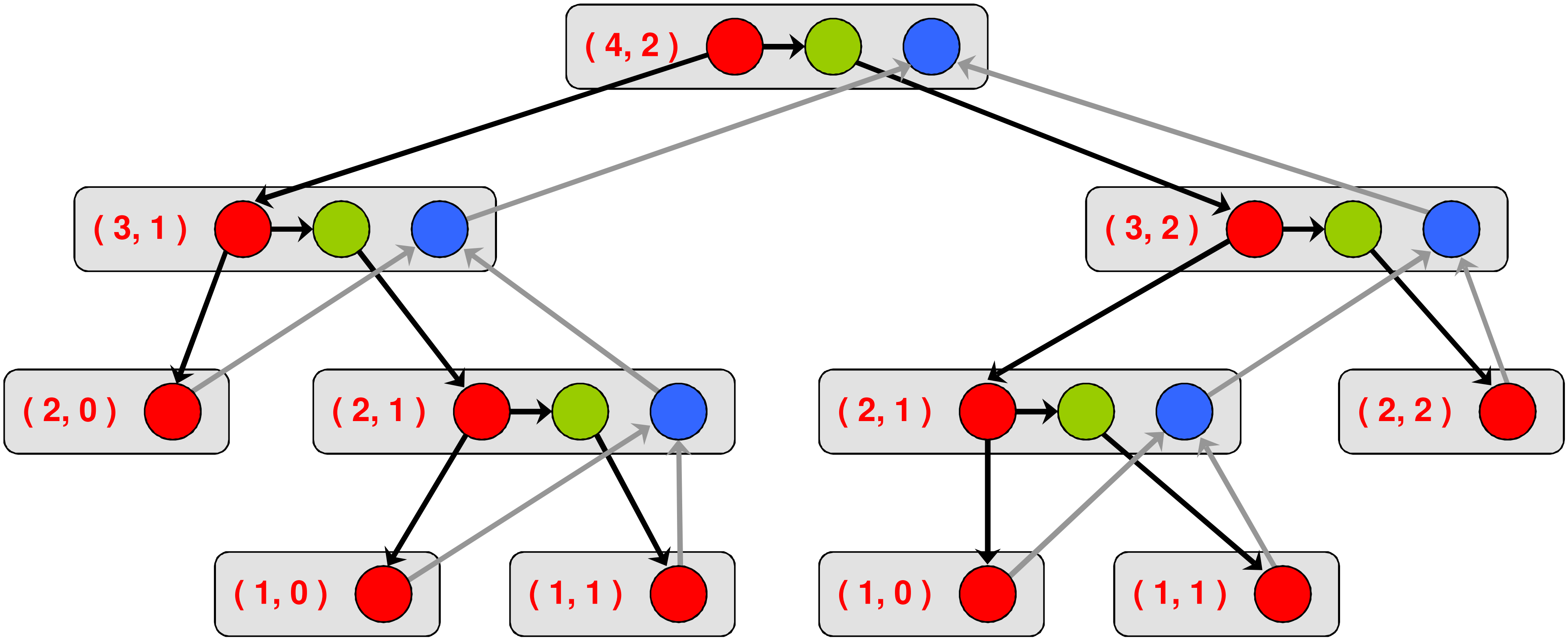


– A parallel instruction stream is represented by a DAG $G = (V, E)$.

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

– Each edge $e \in E$ is a *spawn*, *call*, *continue* or *return* edge.

Parallelism in comb(4, 2)



work: $T_1 = 21$

span: $T_\infty = 9$

parallelism $= T_1 / T_\infty = 21 / 9 \approx \boxed{2.33}$

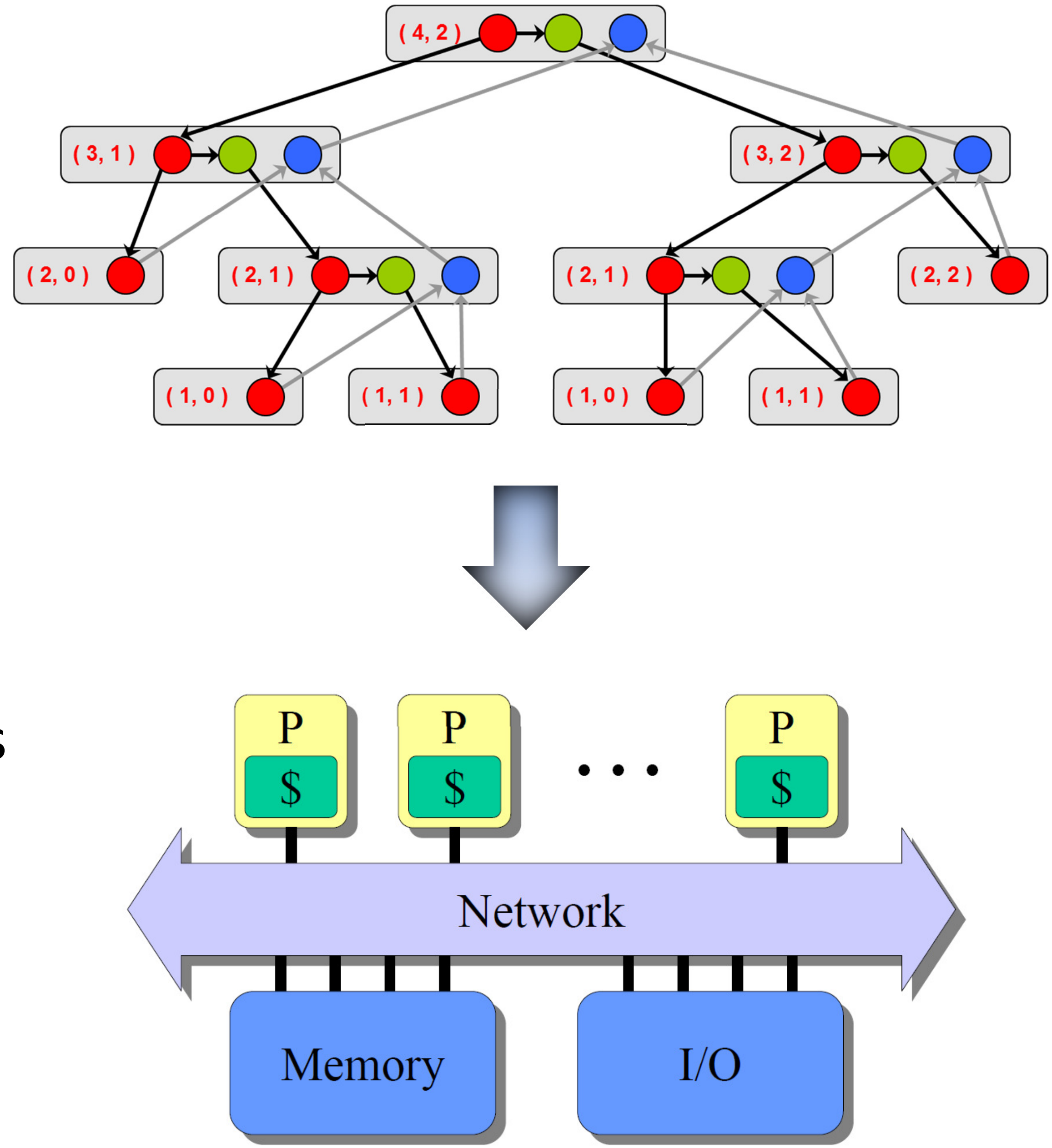
Only marginal performance gains with more than 2 cores!

Scheduler

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

The map is called a *schedule*.

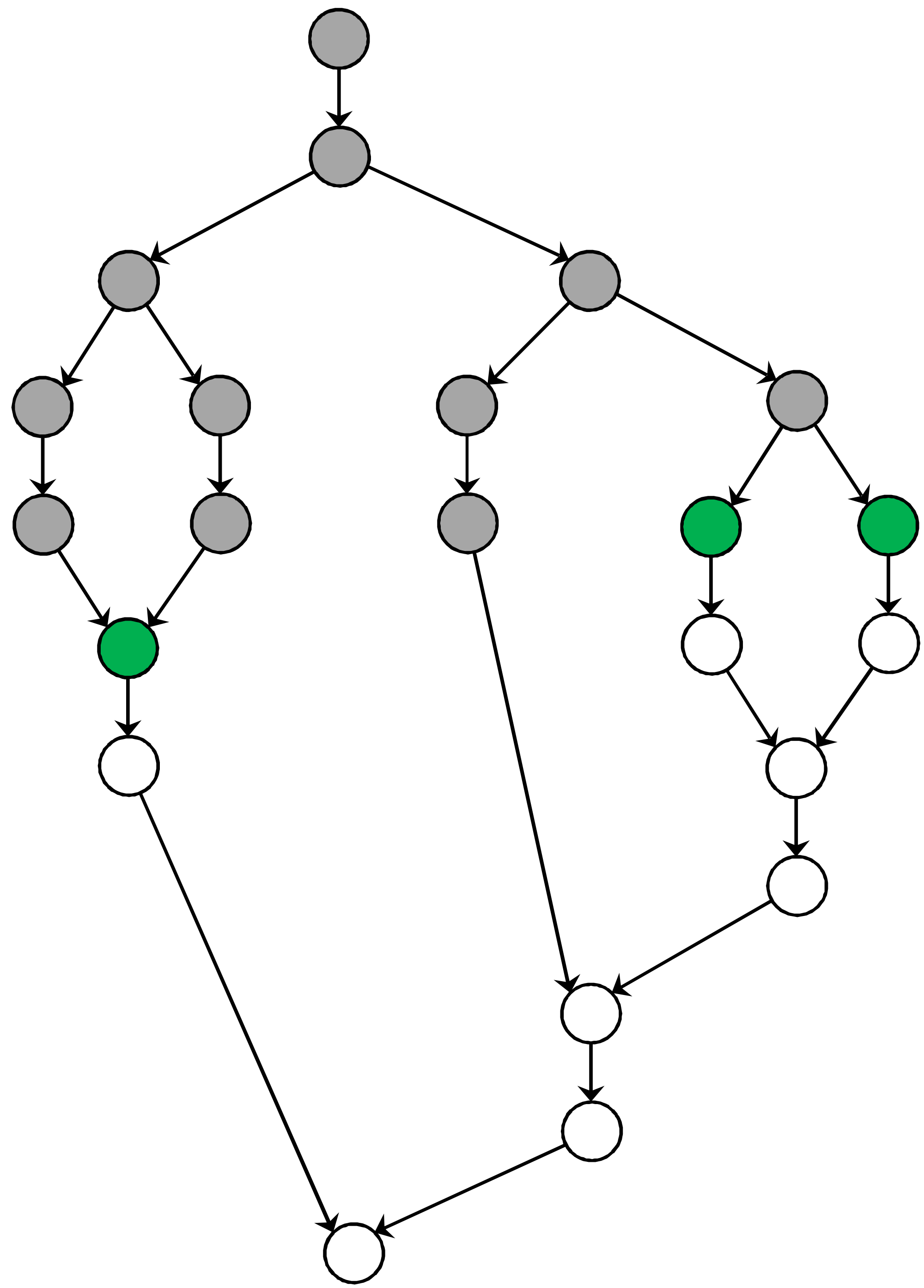
An *offline scheduler* prepares the schedule prior to the actual execution of the program.



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.

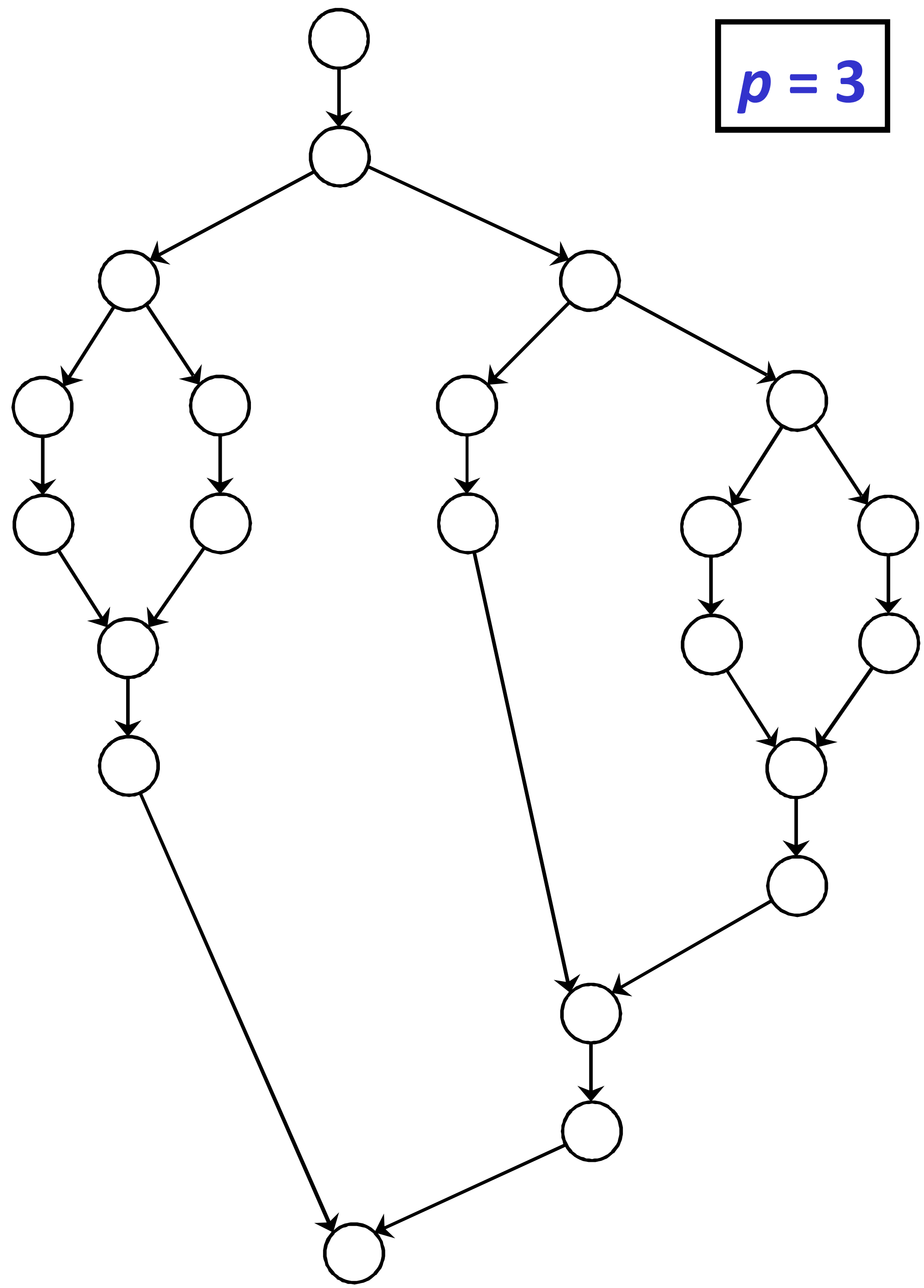
A Centralized Greedy Scheduler

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)



$p = 3$

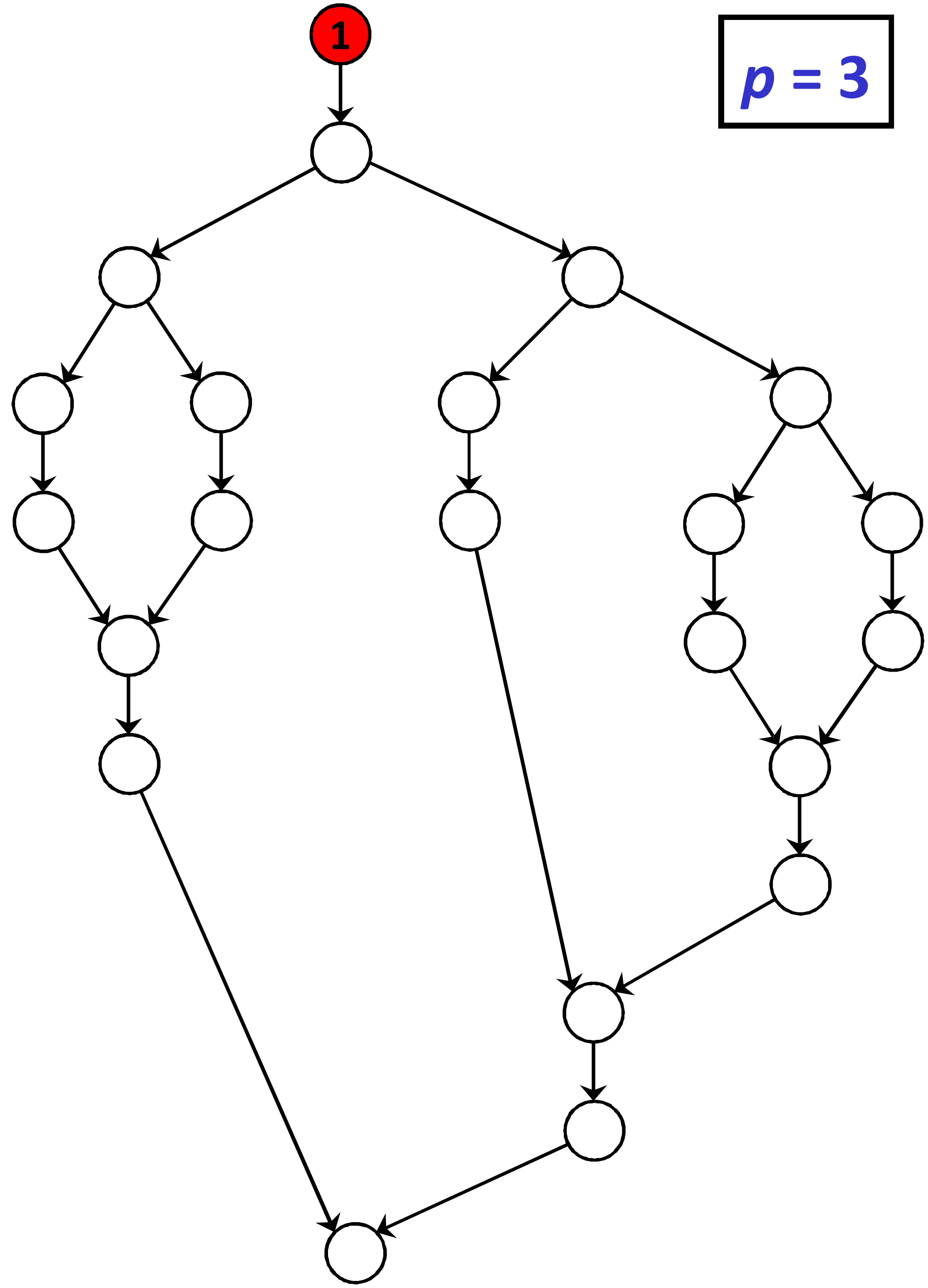
A Centralized Greedy Scheduler

Let p = number of cores

At every step:

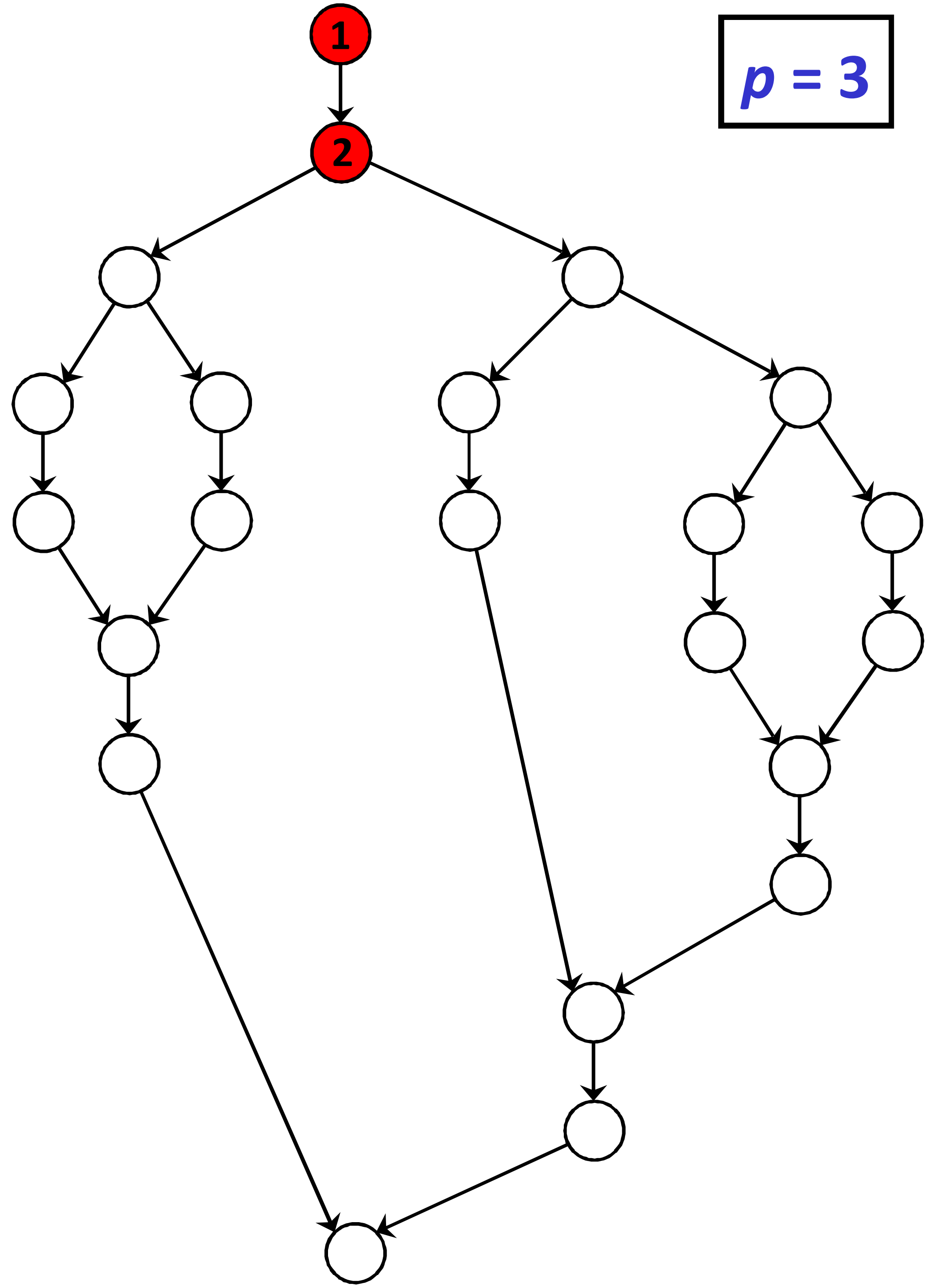
– if $\geq p$ tasks are ready: execute any p of them (complete step)

– if $< p$ tasks are ready: execute all of them (incomplete step)



$p = 3$

A Centralized Greedy Scheduler



$p = 3$

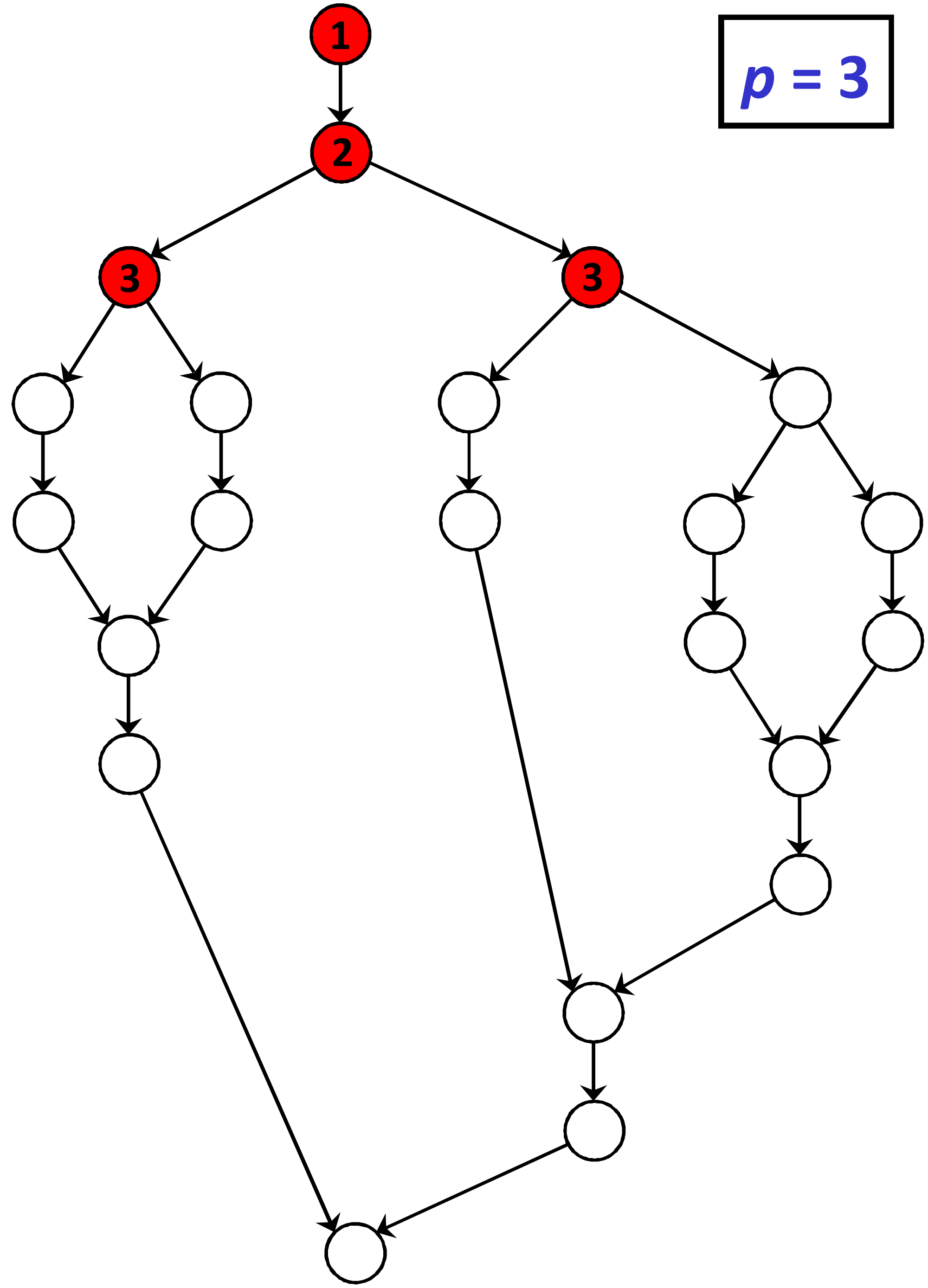
Let p = number of cores

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)

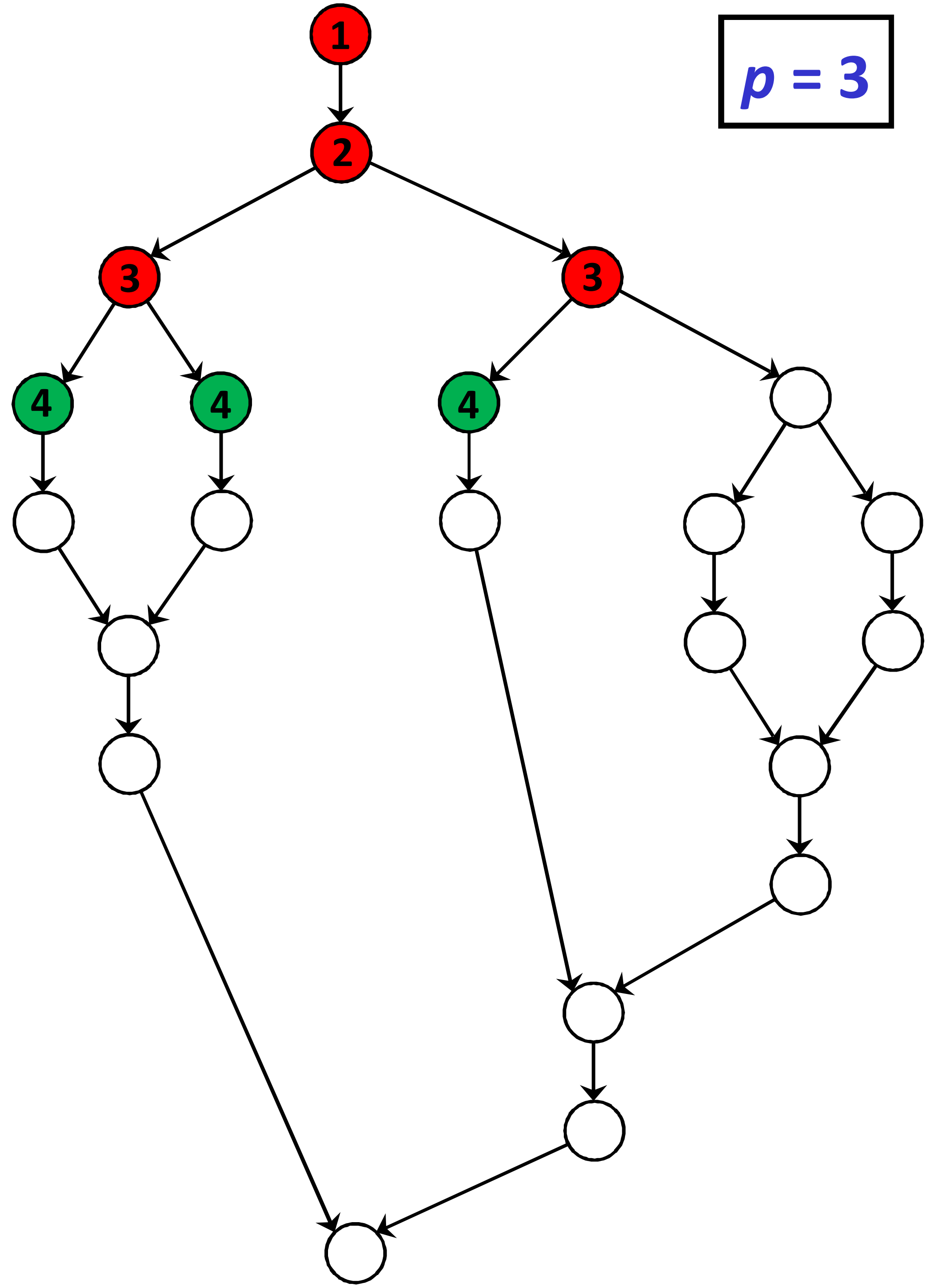
A Centralized Greedy Scheduler

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)



$p = 3$

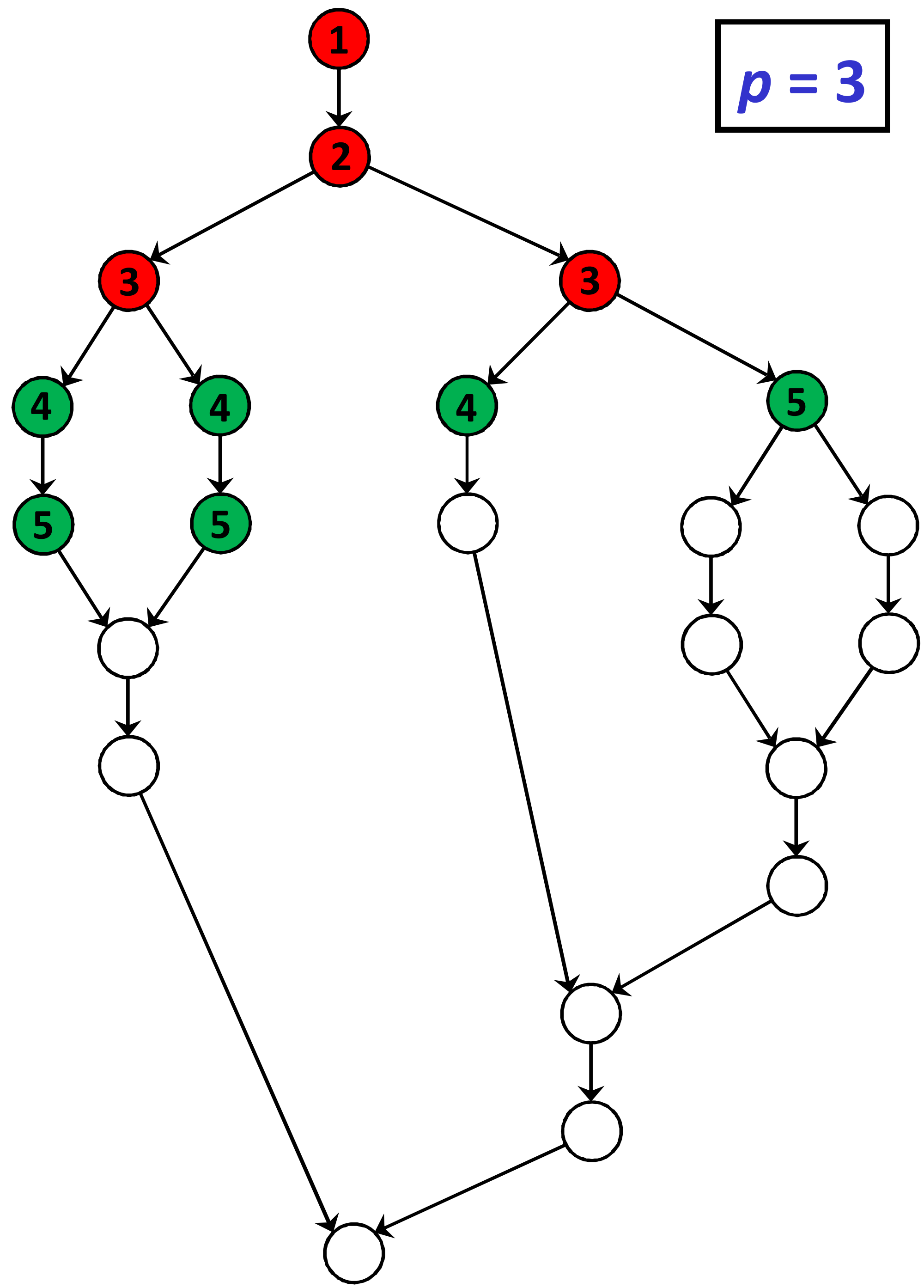
A Centralized Greedy Scheduler

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)



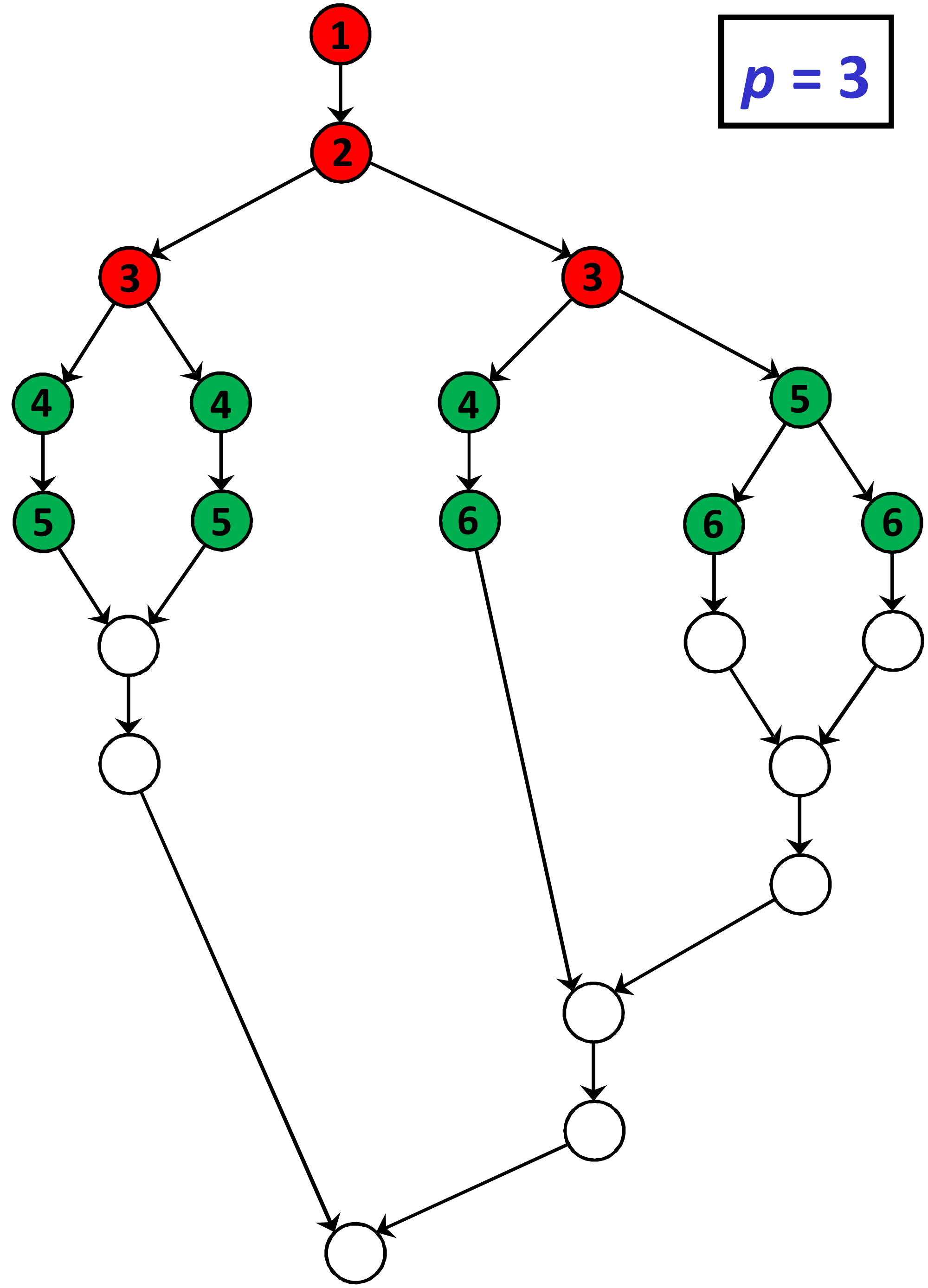
A Centralized Greedy Scheduler

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)



$p = 3$

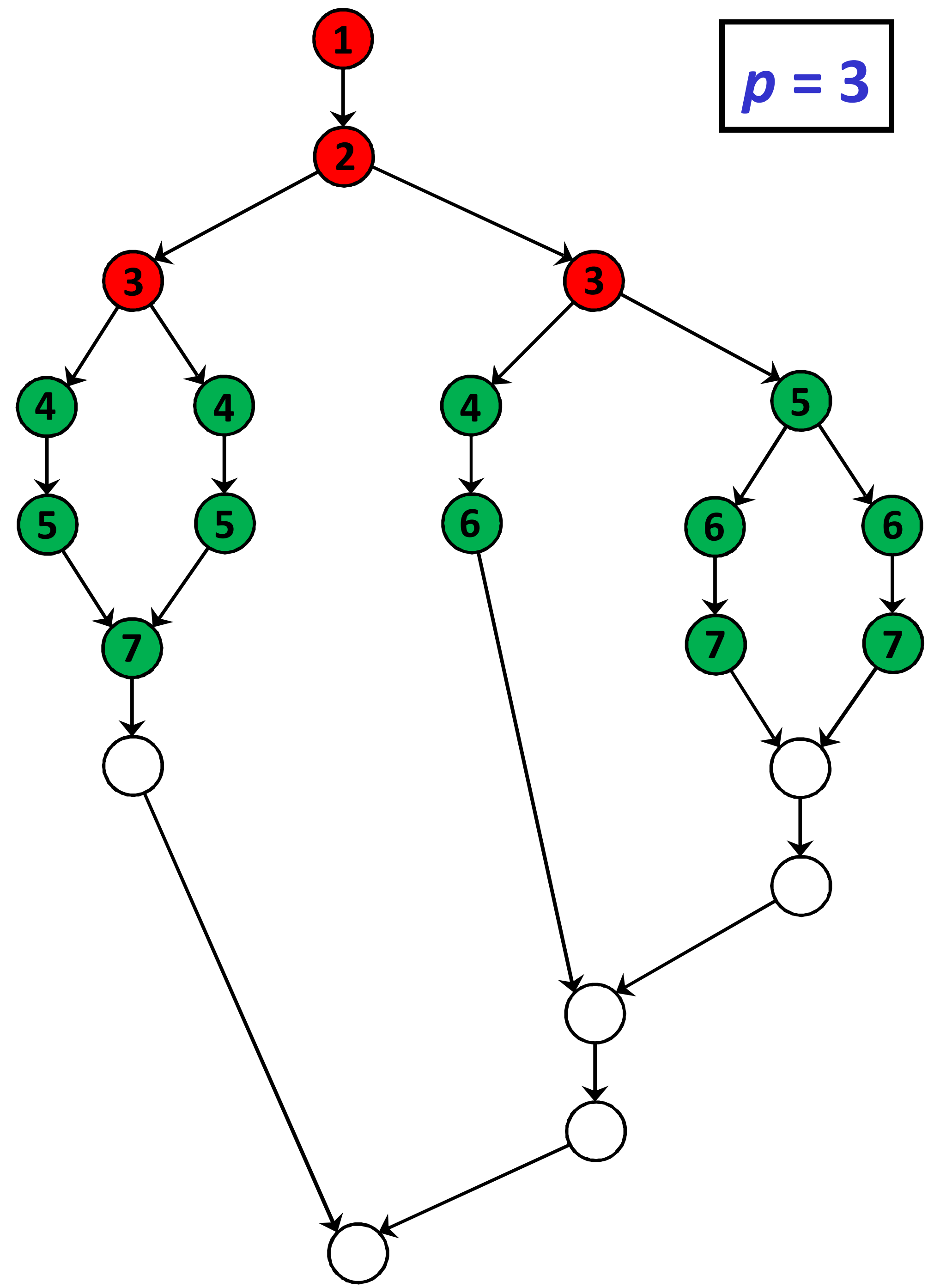
A Centralized Greedy Scheduler

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)



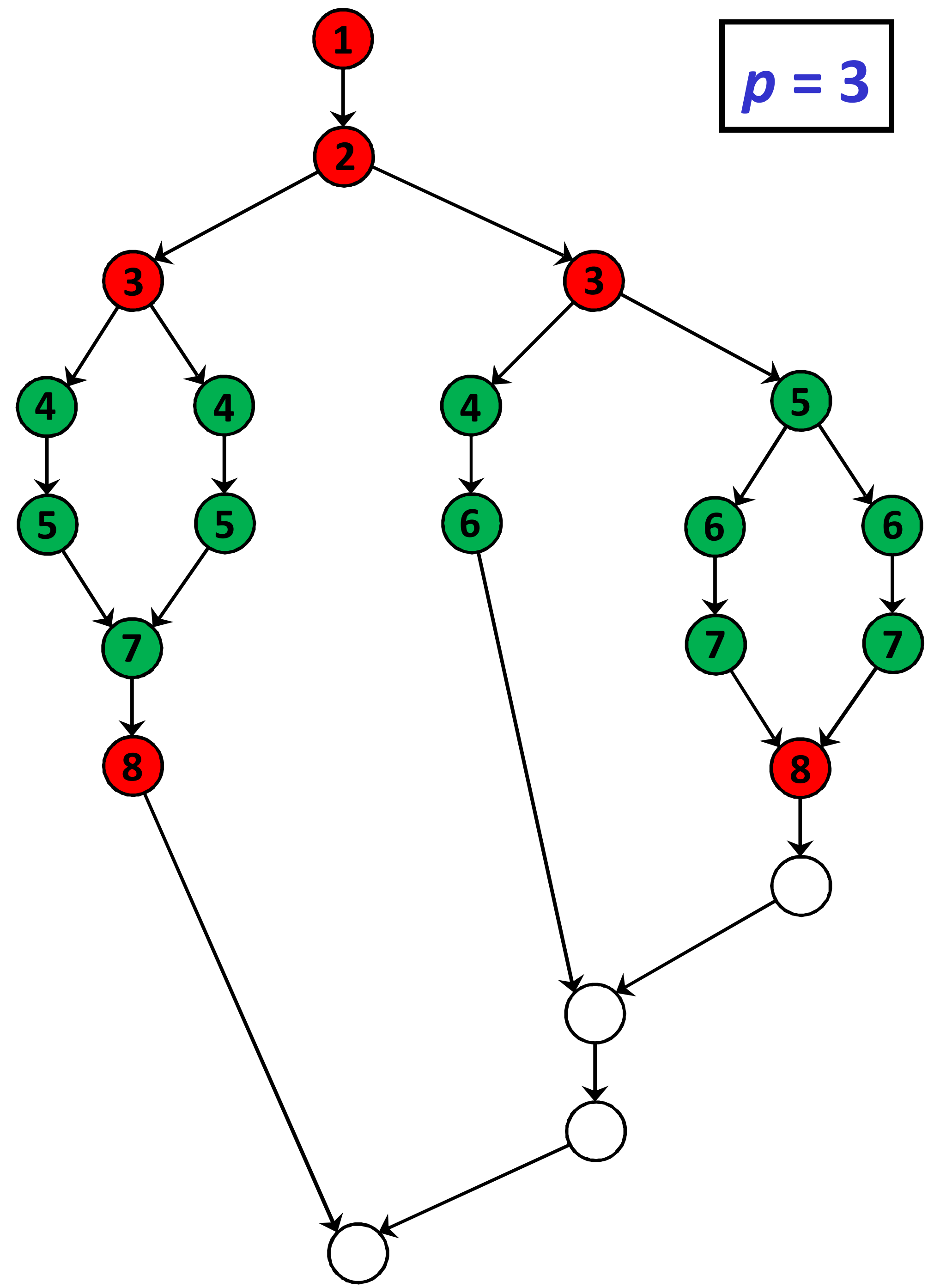
A Centralized Greedy Scheduler

Let p = number of cores

At every step:

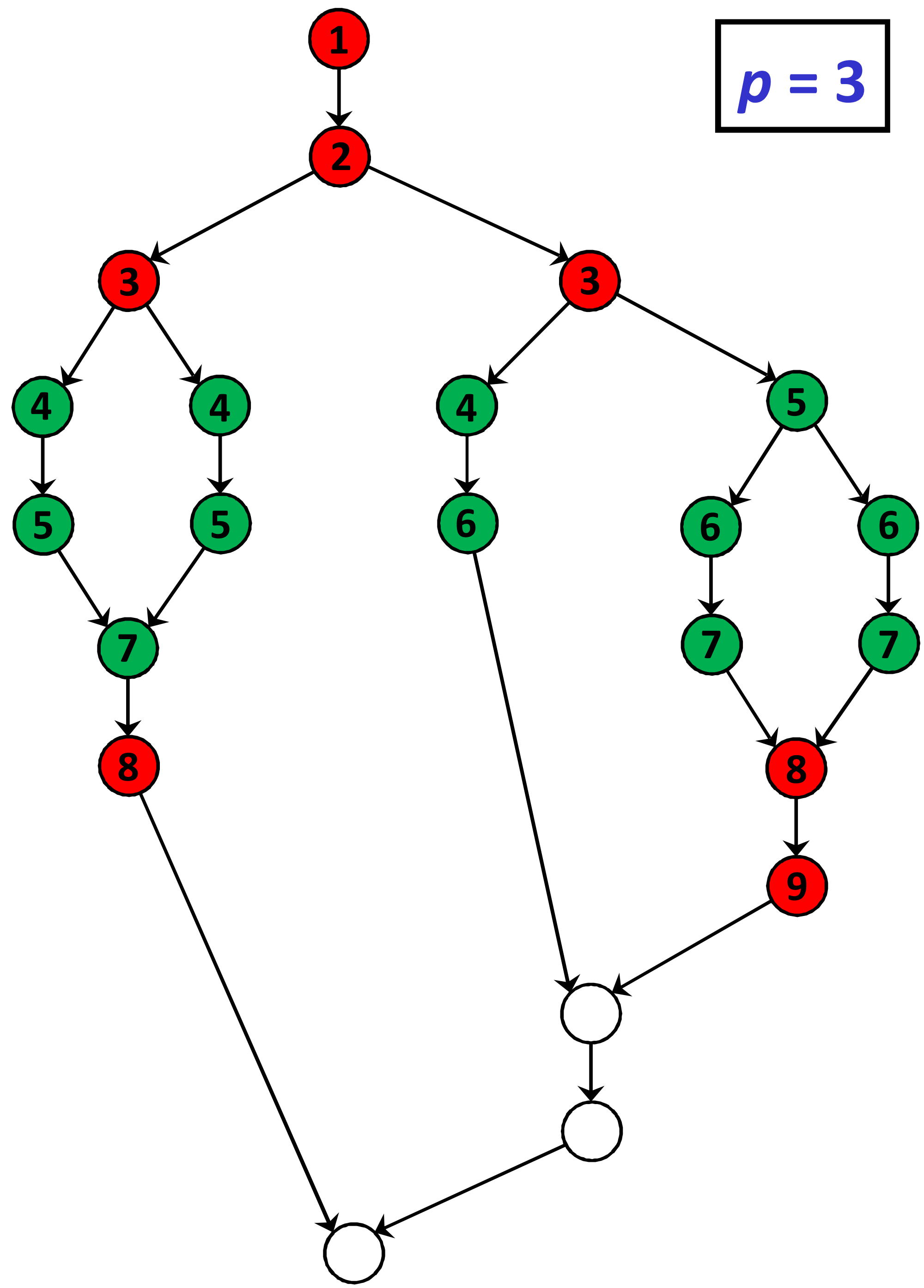
– if $\geq p$ tasks are ready: execute any p of them (complete step)

– if $< p$ tasks are ready: execute all of them (incomplete step)



$p = 3$

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)

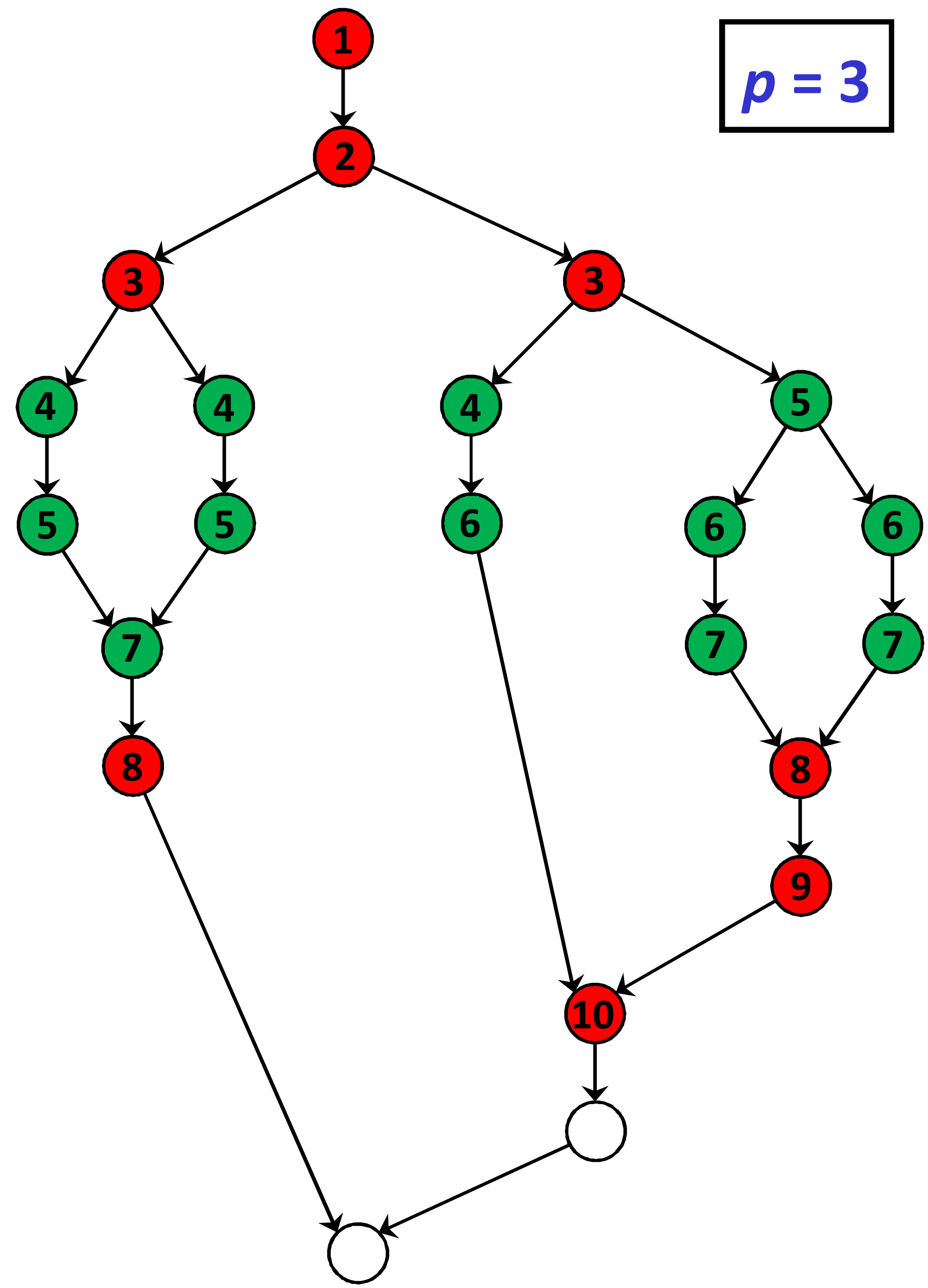
A Centralized Greedy Scheduler

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)



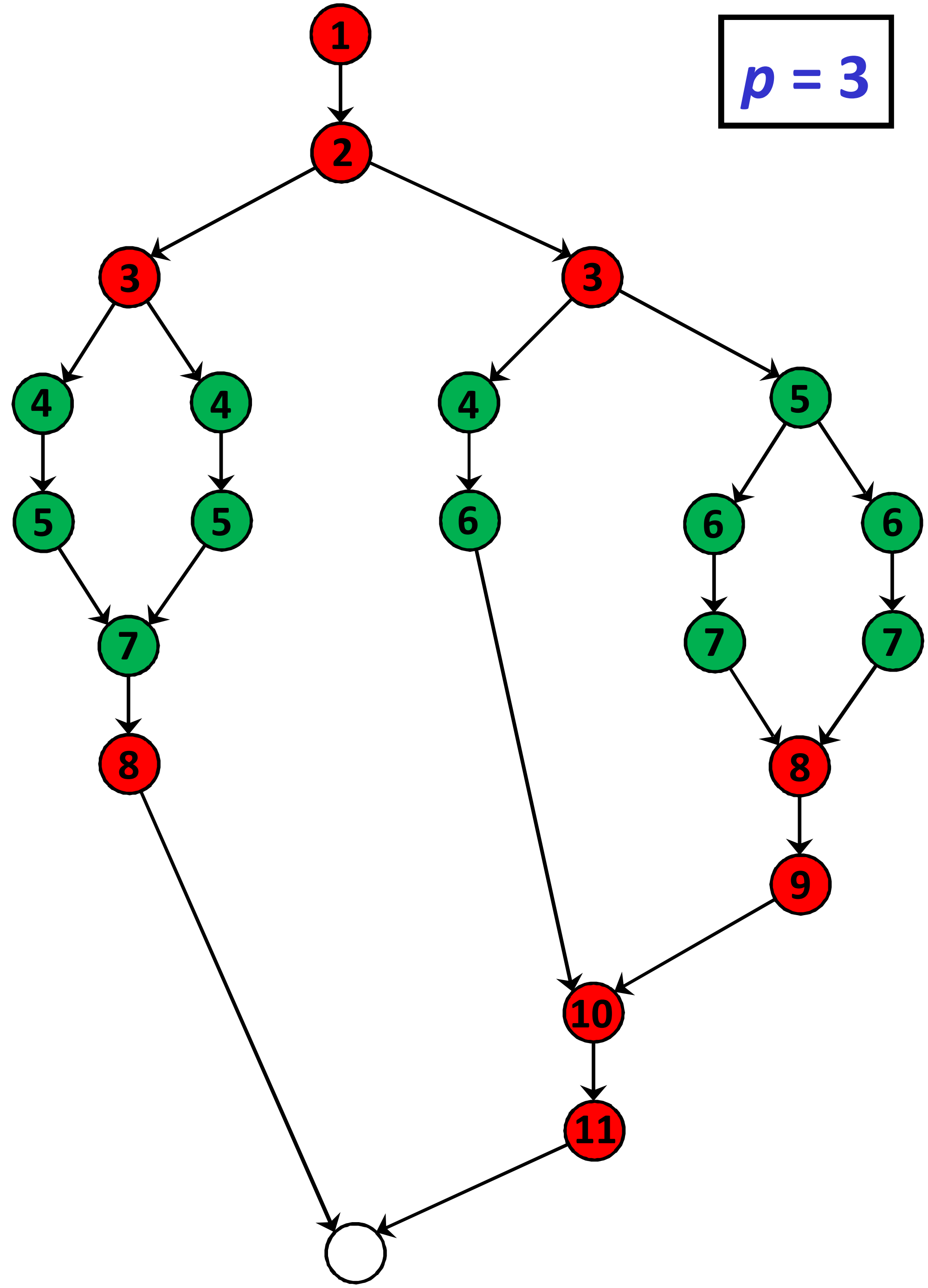
A Centralized Greedy Scheduler

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)



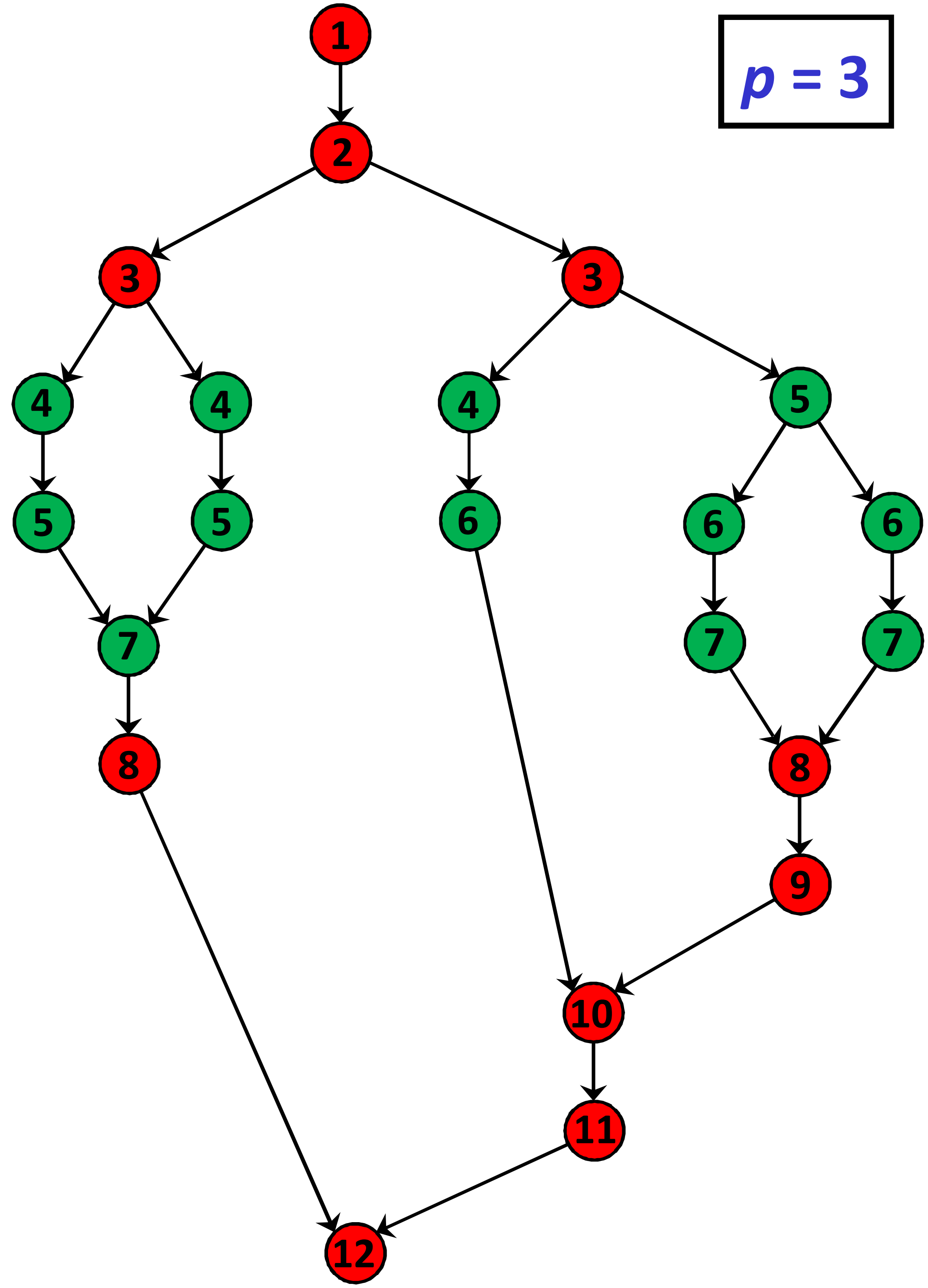
A Centralized Greedy Scheduler

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

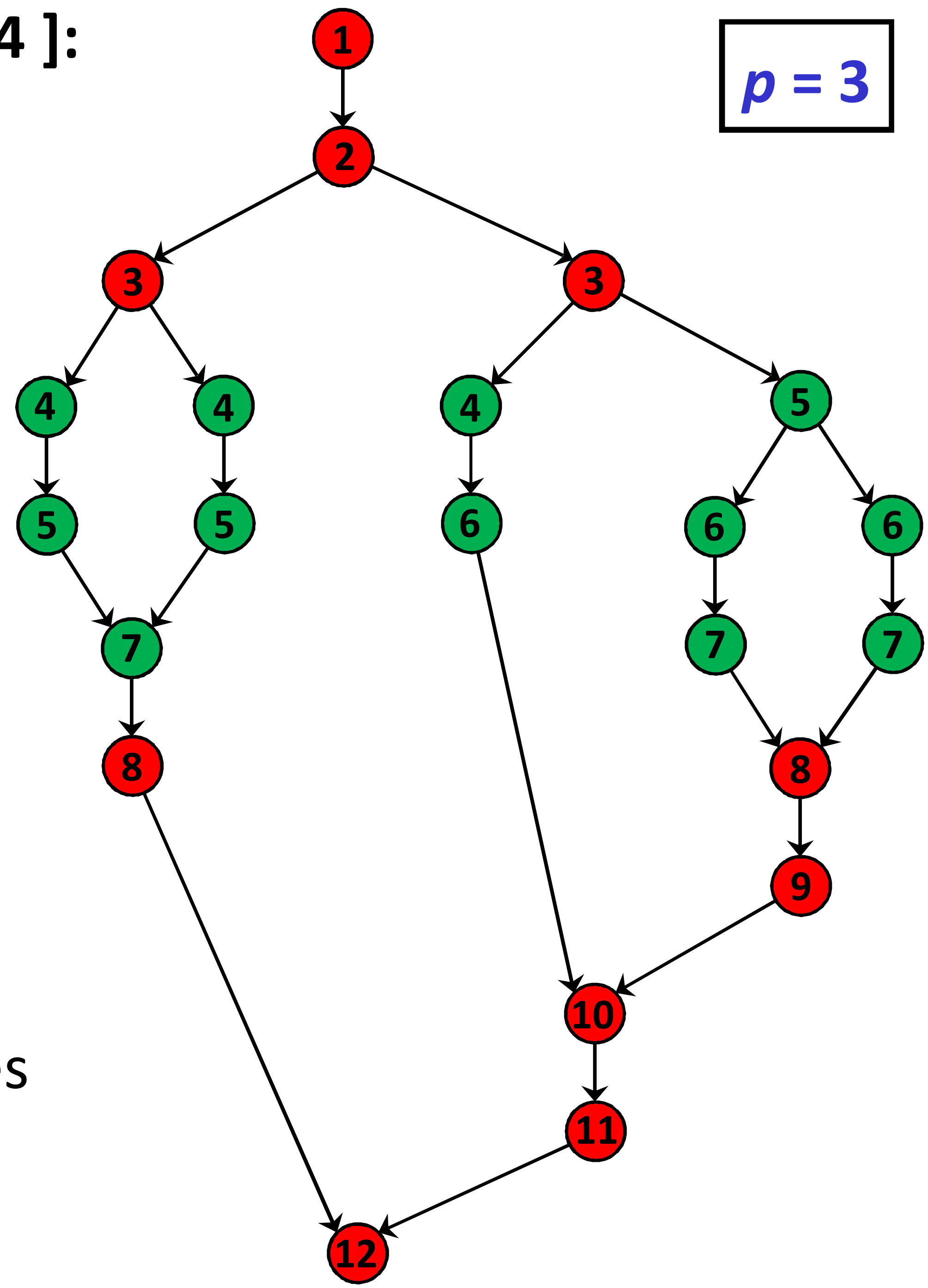
+ #incomplete steps

– 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 }
for i ← 1 to n do
    for j ← 1 to n do
        Z[ i ][ j ] ← 0
        for k ← 1 to n do
            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 }
parallel for i ← 1 to n do
    parallel for j ← 1 to n do
        Z[ i ][ j ] ← 0
        for k ← 1 to n do
            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: $T_p(n) = O\left(\frac{T_1(n)}{p} + T_\infty(n)\right) = O\left(\frac{n^3}{p} + n\right)$

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

Parallel Recursive MM

$$\underbrace{\begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix}}_{Z} = \underbrace{\begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix}}_{X} \times \underbrace{\begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix}}_{Y}$$

$$= \begin{bmatrix} X_{11}\,Y_{11} + X_{12}\,Y_{21} & X_{11}\,Y_{12} + X_{12}\,Y_{22} \\ X_{21}\,Y_{11} + X_{22}\,Y_{21} & X_{21}\,Y_{12} + X_{22}\,Y_{22} \end{bmatrix}$$

Parallel Recursive MM

```
Par-Rec-MM ( Z, X, Y )     { X, Y, Z are n × n matrices,
                             where n = 2^k for integer k ≥ 0 }
if n = 1 then
    Z ← Z + X · Y
else
    spawn Par-Rec-MM ( Z11, X11, Y11 )
    spawn Par-Rec-MM ( Z12, X11, Y12 )
    spawn Par-Rec-MM ( Z21, X21, Y11 )
          Par-Rec-MM ( Z21, X21, Y12 )
    sync
    spawn Par-Rec-MM ( Z11, X12, Y21 )
    spawn Par-Rec-MM ( Z12, X12, Y22 )
    spawn Par-Rec-MM ( Z21, X22, Y21 )
          Par-Rec-MM ( Z22, X22, Y22 )
    sync
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 }
if n = 1 then
    Z ← Z + X · Y
else
    spawn Par-Rec-MM ( Z11, X11, Y11 )
    spawn Par-Rec-MM ( Z12, X11, Y12 )
    spawn Par-Rec-MM ( Z21, X21, Y11 )
          Par-Rec-MM ( Z21, X21, Y12 )
    sync
    spawn Par-Rec-MM ( Z11, X12, Y21 )
    spawn Par-Rec-MM ( Z12, X12, Y22 )
    spawn Par-Rec-MM ( Z21, X22, Y21 )
          Par-Rec-MM ( Z22, X22, Y22 )
    sync
endif
```

Work:

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

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

Span:

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

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

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

Additional Space:

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

Recursive MM with More Parallelism

$$Z = \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix} = \begin{bmatrix} X_{11}Y_{11} + X_{12}Y_{21} & X_{11}Y_{12} + X_{12}Y_{22} \\ X_{21}Y_{11} + X_{22}Y_{21} & X_{21}Y_{12} + X_{22}Y_{22} \end{bmatrix}$$

$$= \begin{bmatrix} X_{11}Y_{11} & X_{11}Y_{12} \\ X_{21}Y_{11} & X_{21}Y_{12} \end{bmatrix} + \begin{bmatrix} X_{12}Y_{21} & X_{12}Y_{22} \\ X_{22}Y_{21} & X_{22}Y_{22} \end{bmatrix}$$

Recursive MM with More Parallelism

Par-Rec-MM2 (Z, X, Y) { X, Y, Z are $n \times n$ matrices, where $n = 2^k$ for integer $k \geq 0$ }

if $n = 1$ *then*
    $Z \leftarrow Z + X \cdot Y$
else { T is a temporary $n \times n$ matrix }
    spawn Par-Rec-MM2 (Z_{11}, X_{11}, Y_{11})
    spawn Par-Rec-MM2 (Z_{12}, X_{11}, Y_{12})
    spawn Par-Rec-MM2 (Z_{21}, X_{21}, Y_{11})
    spawn Par-Rec-MM2 (Z_{21}, X_{21}, Y_{12})
    spawn Par-Rec-MM2 (T_{11}, X_{12}, Y_{21})
    spawn Par-Rec-MM2 (T_{12}, X_{12}, Y_{22})
    spawn Par-Rec-MM2 (T_{21}, X_{22}, Y_{21})
    Par-Rec-MM2 (T_{22}, X_{22}, Y_{22})
    sync
    parallel for $i \leftarrow 1$ *to* n *do*
        parallel for $j \leftarrow 1$ *to* n *do*
            $Z[i][j] \leftarrow Z[i][j] + T[i][j]$
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 }
if n = 1 then
    Z ← Z + X · Y
else          { T is a temporary n × n matrix }
    spawn Par-Rec-MM2 ( Z11, X11, Y11 )
    spawn Par-Rec-MM2 ( Z12, X11, Y12 )
    spawn Par-Rec-MM2 ( Z21, X21, Y11 )
    spawn Par-Rec-MM2 ( Z21, X21, Y12 )
    spawn Par-Rec-MM2 ( T11, X12, Y21 )
    spawn Par-Rec-MM2 ( T12, X12, Y22 )
    spawn Par-Rec-MM2 ( T21, X22, Y21 )
          Par-Rec-MM2 ( T22, X22, Y22 )
    sync
    parallel for i ← 1 to n do
        parallel for j ← 1 to n do
            Z[ i ][ j ] ← Z[ i ][ j ] + T[ i ][ j ]
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 ] }

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

⇓

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

if p < r then
    q ← ⌊ ( p + r ) / 2 ⌋
    spawn Merge-Sort ( A, p, q )
          Merge-Sort ( A, q + 1, r )
    sync
    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: $T_\infty(n) = \begin{cases} \Theta(1), & if\ n = 1, \\ T_\infty\left(\frac{n}{2}\right) + \Theta(n), & otherwise. \end{cases}$

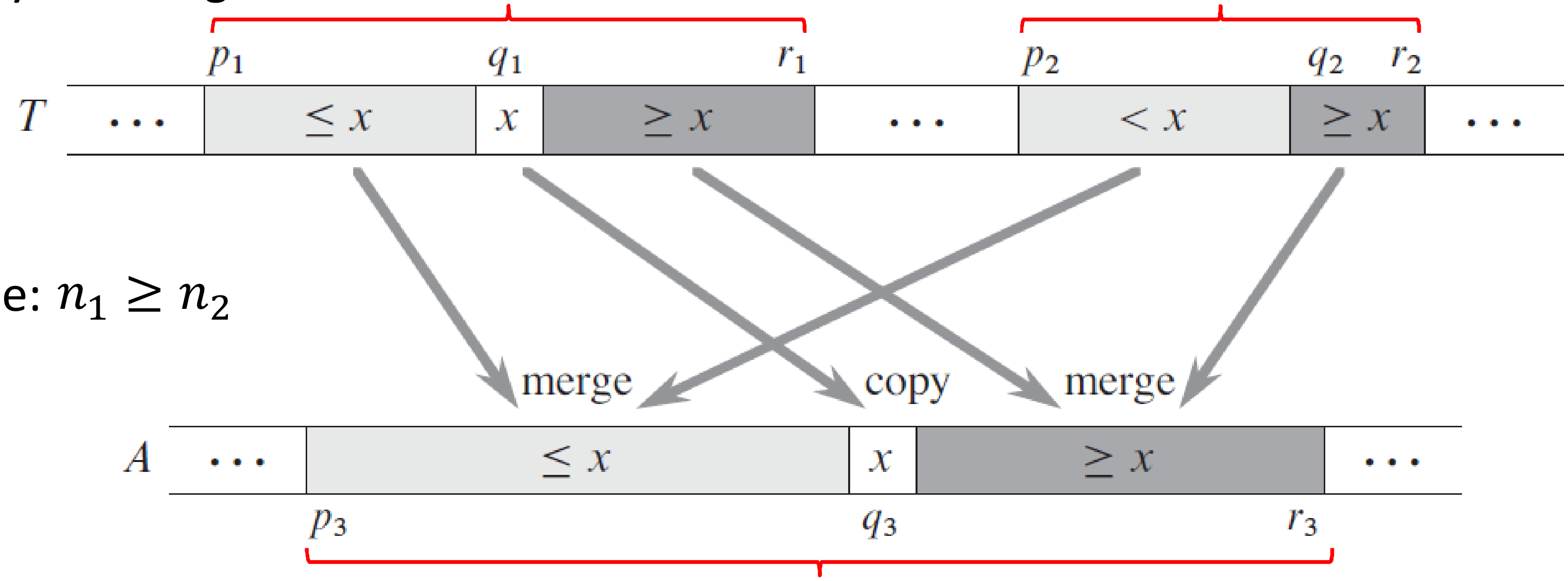
$= \Theta(n)$ [MT Case 3]

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

Parallel Merge

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

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



suppose: $n_1 \geq n_2$

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

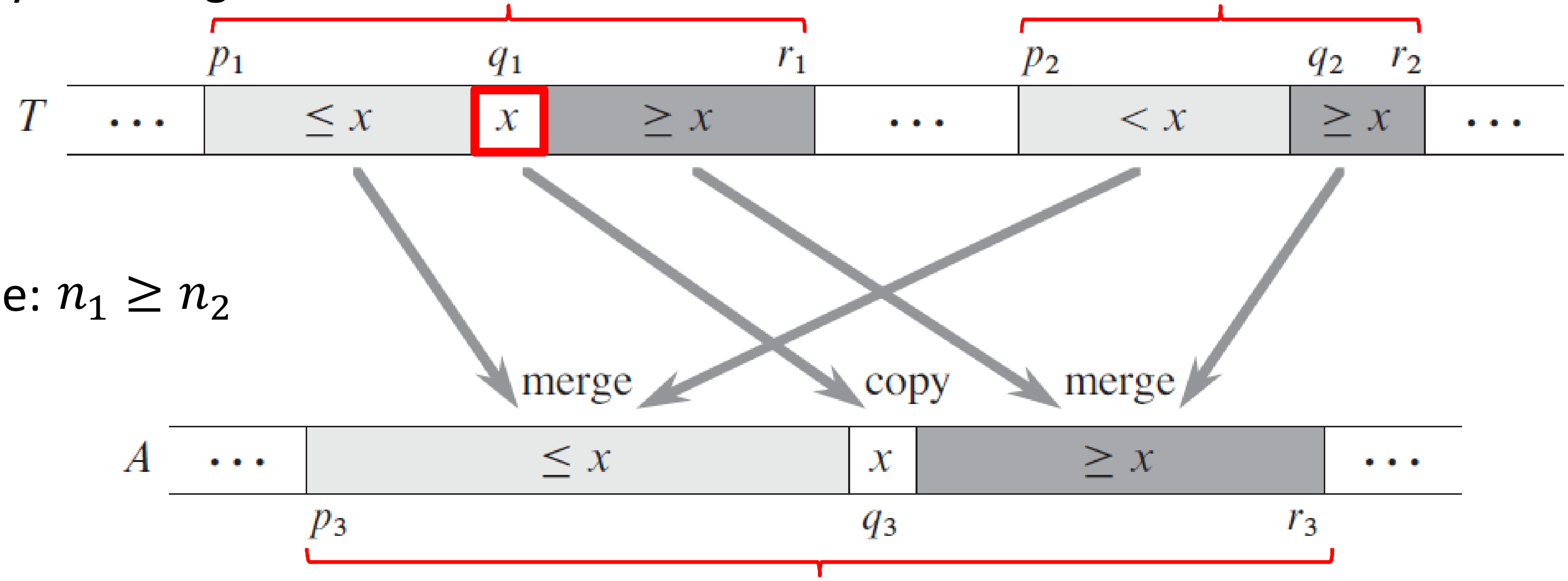
$n_3 = r_3 - p_3 + 1 = n_1 + n_2$

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

Parallel Merge

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

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



suppose: $n_1 \ge n_2$

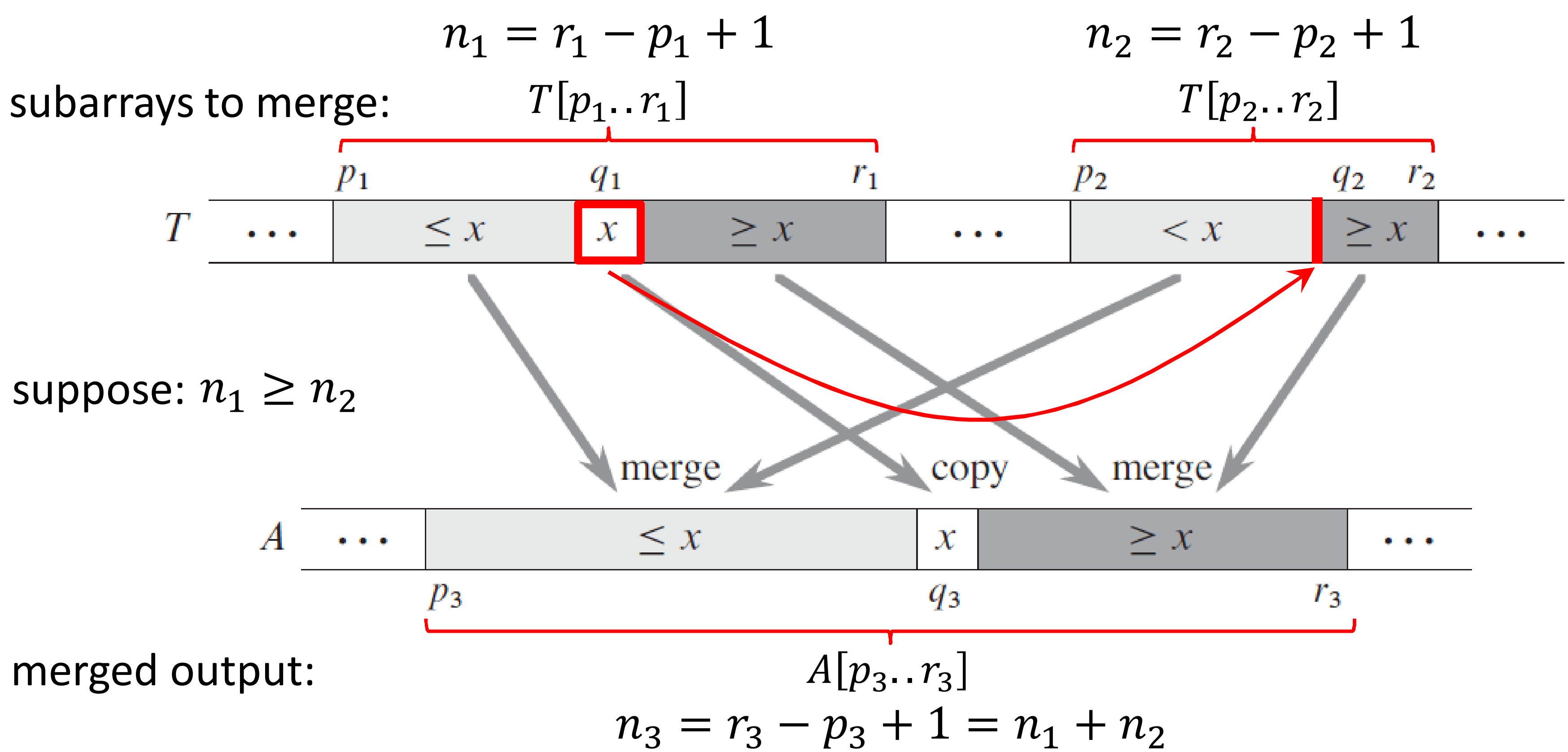
merged output: $A[p_3 \mathinner{..} r_3]$

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

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

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

Parallel Merge



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

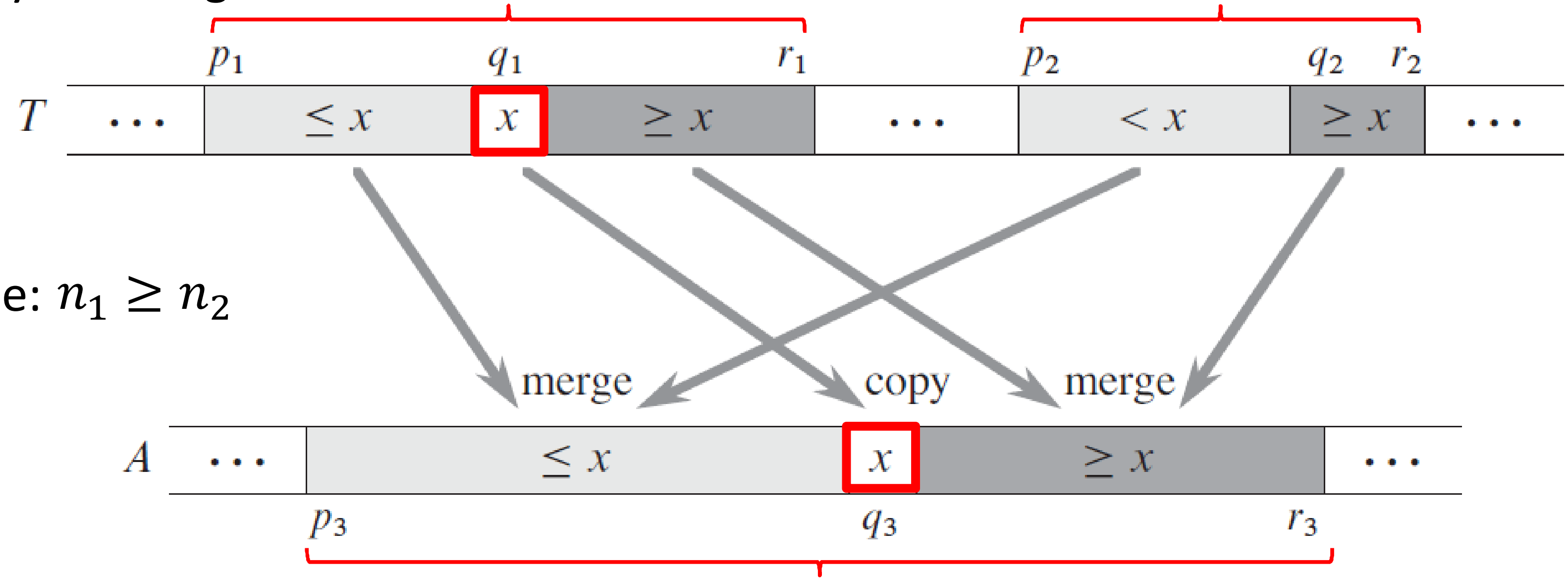
Step 2: Use binary search to find the index q_2 in subarray $T[p_2..r_2]$ so that the subarray would still be sorted if we insert x between $T[q_2 - 1]$ and $T[q_2]$

Parallel Merge

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

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

suppose: $n_1 \geq n_2$



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

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

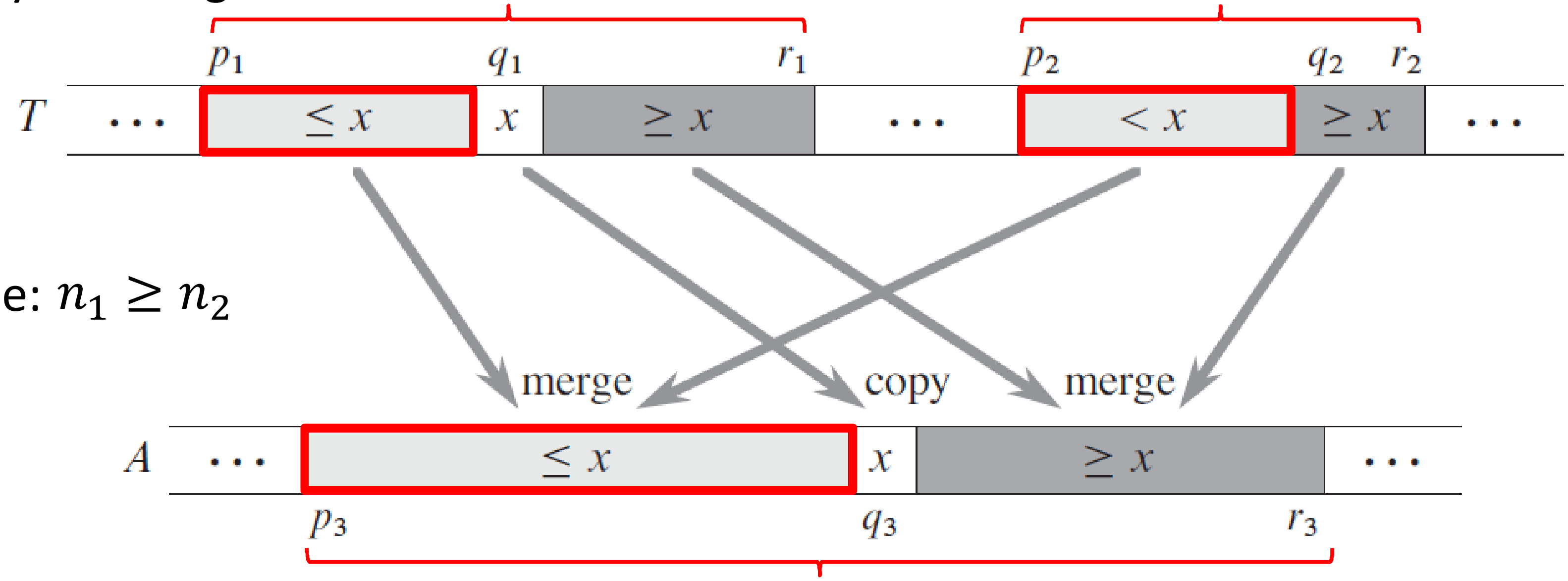
Source: Cormen et al., "Introduction to Algorithms", 3rd Edition

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: $T[p_1 \mathinner{..} r_1]$ $\quad$ $T[p_2 \mathinner{..} r_2]$



suppose: $n_1 \geq n_2$

merged output: $A[p_3 \mathinner{..} r_3]$

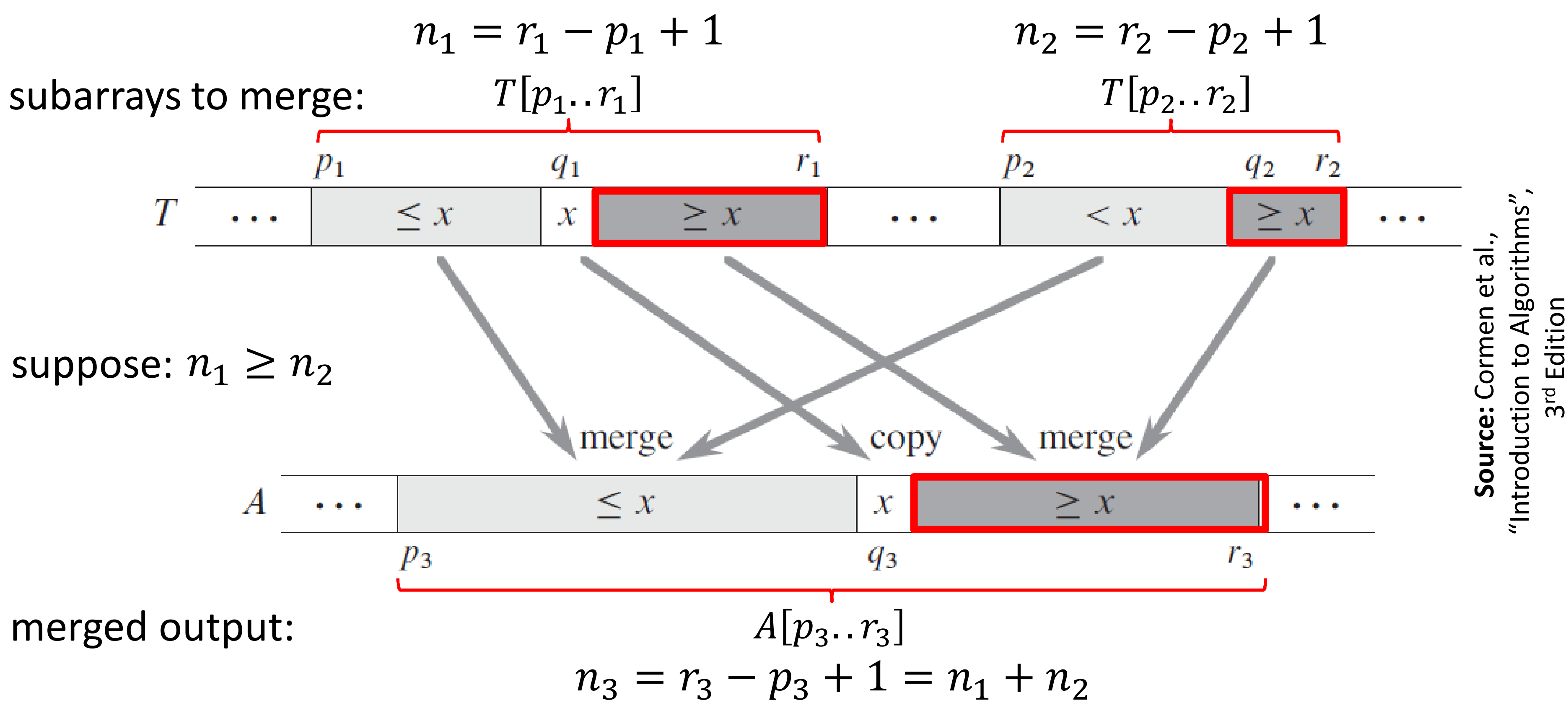
$n_3 = r_3 - p_3 + 1 = n_1 + n_2$

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

Perform the following two steps in parallel.

Step 4(a): Recursively merge $T[p_1 \mathinner{..} q_1 - 1]$ with $T[p_2 \mathinner{..} q_2 - 1]$, and place the result into $A[p_3 \mathinner{..} q_3 - 1]$

Parallel Merge



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 ($T, p_1, r_1, p_2, r_2, A, p_3$)

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

if $n_1 < n_2$ *then*

  $p_1 \leftrightarrow p_2, \quad r_1 \leftrightarrow r_2, \quad n_1 \leftrightarrow n_2$

if $n_1 = 0$ *then return*

else

  $q_1 \leftarrow \lfloor (p_1 + r_1) / 2 \rfloor$

  $q_2 \leftarrow$ *Binary-Search* ($T[q_1], T, p_2, r_2$)

  $q_3 \leftarrow p_3 + (q_1 - p_1) + (q_2 - p_2)$

  $A[q_3] \leftarrow T[q_1]$

  spawn *Par-Merge* ($T, p_1, q_1-1, p_2, q_2-1, A, p_3$)

    Par-Merge ($T, q_1+1, r_1, q_2+1, r_2, A, q_3+1$)

  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 $(T, p_1, r_1, p_2, r_2, A, p_3)$

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

if $n_1 < n_2$ *then*

    $p_1 \leftrightarrow p_2, \quad r_1 \leftrightarrow r_2, \quad n_1 \leftrightarrow n_2$

if $n_1 = 0$ *then return*

else

    $q_1 \leftarrow \lfloor (p_1 + r_1) / 2 \rfloor$

    $q_2 \leftarrow$ *Binary-Search* $(T[q_1], T, p_2, r_2)$

    $q_3 \leftarrow p_3 + (q_1 - p_1) + (q_2 - p_2)$

    $A[q_3] \leftarrow T[q_1]$

    spawn Par-Merge $(T, p_1, q_1-1, p_2, q_2-1, A, p_3)$

    Par-Merge $(T, q_1+1, r_1, q_2+1, r_2, A, q_3+1)$

    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, $T_1(n) = O(n)$

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

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

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: $T_1(n) \le c_1 n - c_2 \log n \ = O(n)$.

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 ] }

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

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

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

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

$$= \Theta(\log^3 n) \qquad \text{[MT Case 2]}$$

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