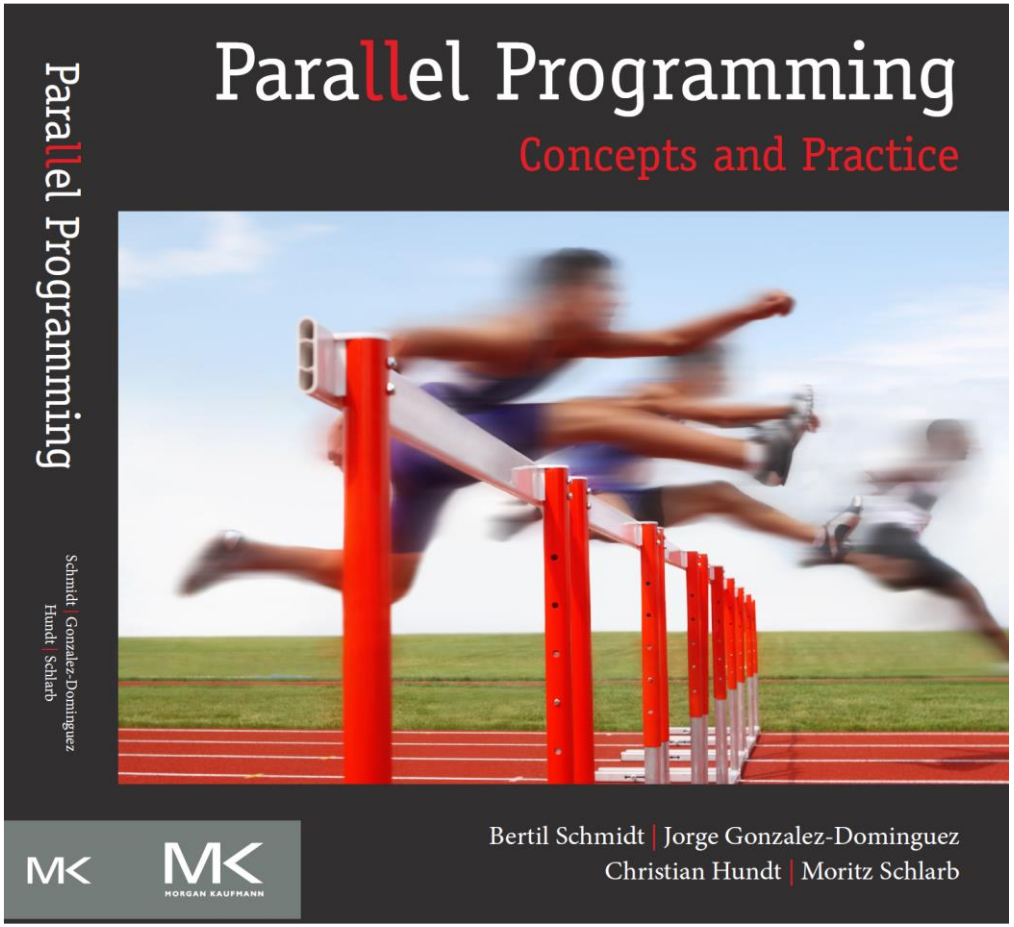
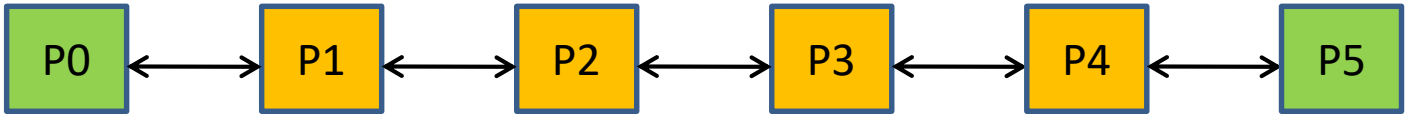
Chapter 02: Theoretical Background



Learning Outcomes Today

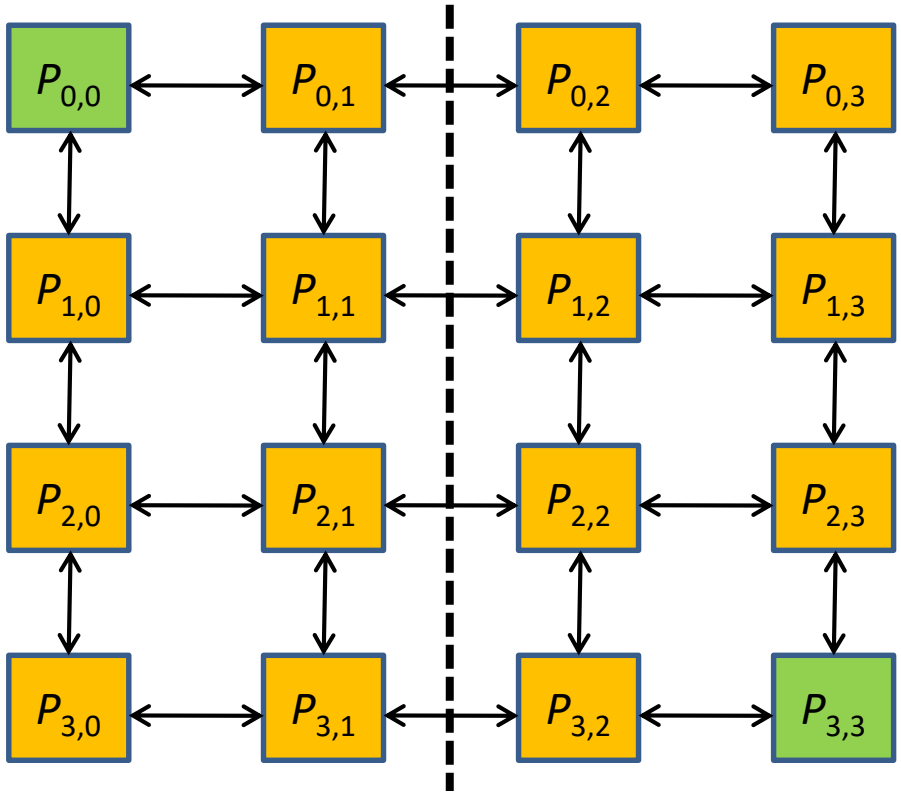
- Learn about some typical topologies of distributed memory systems and network architectures
- Compare their qualities in terms of the graph theoretic concepts degree, bisection width, and diameter
- Use Amdahl's law and Gustafson's law to derive an upper bound on the achievable speedup of parallel program
- Derive both laws as special cases of a more general scaled speedup formulation
- Understand and apply Foster's methodology to parallel program design in order to explore possible parallelization approaches for distributed memory architecture

Linear Array



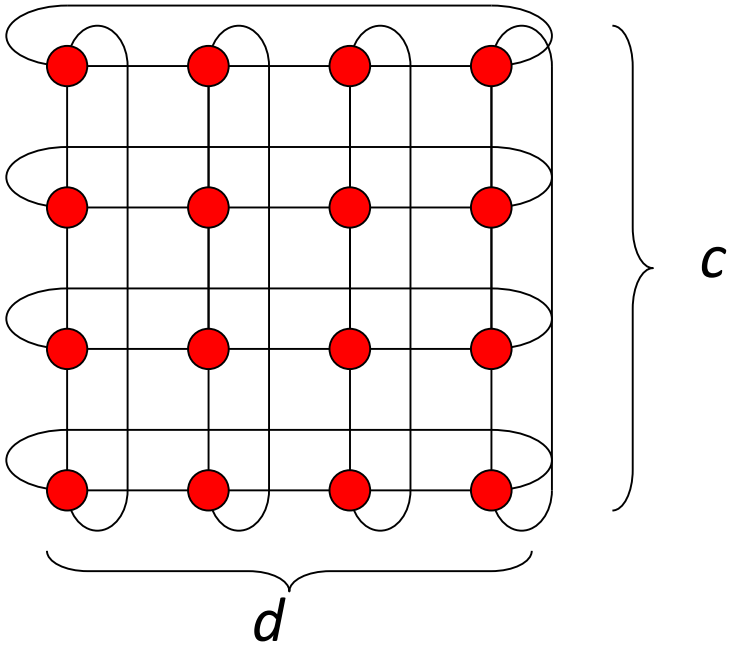
- Assume a linear array with *n* processors, denoted as L_n
- The **degree** of a network is the number maximum number of neighbors of any processor
  - $degree(L_n) = 2$
- The **diameter** of a network is the length of the longest distance between any two pairs of processors
  - $diameter(L_n) = n-1$
- The **bisection-width** of a network is the minimum number of links to be removed to disconnect the network into two halves of equal size
  - $bw(L_n) = 1$

2D Mesh



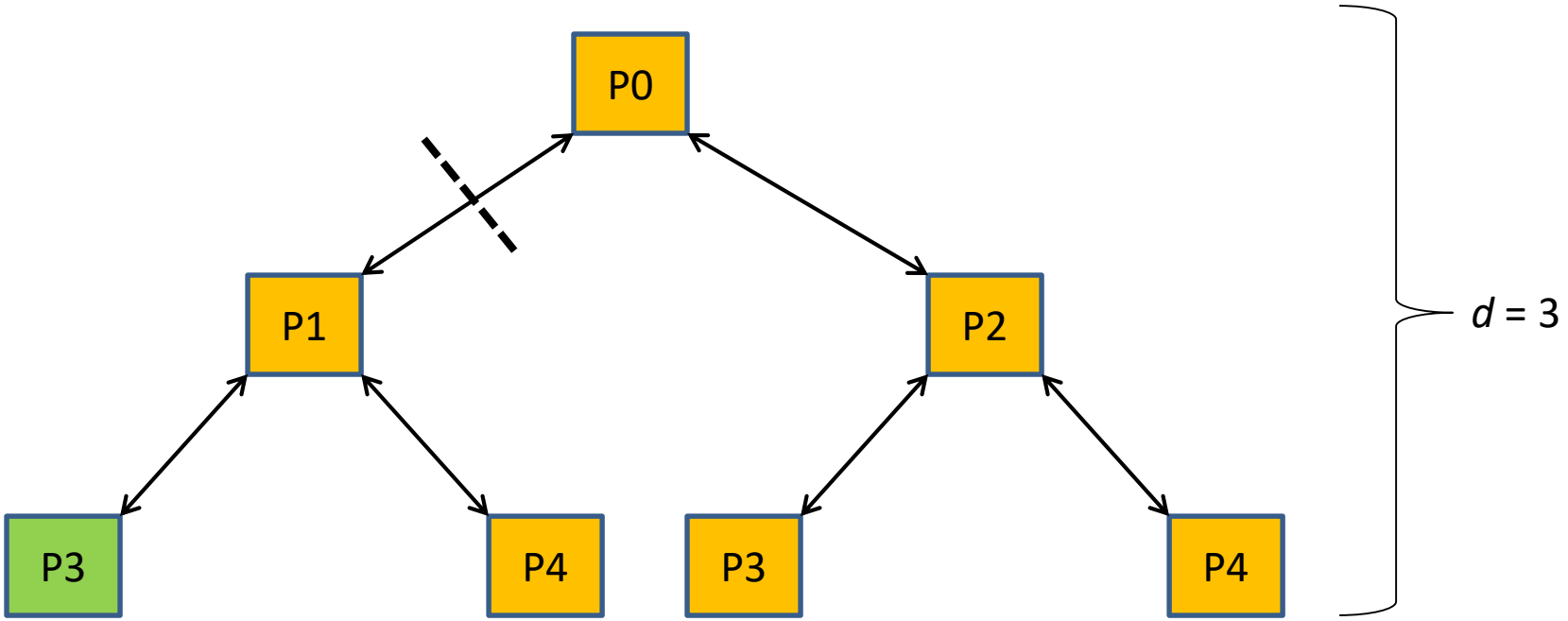
- $M(d,d)$ has $n = d^2$ processors
- degree($M(d,d)$) = 4
  - $\boldsymbol{O(1)}$
- diameter($M(d,d)$) = $2(d-1)$ =
  - $\boldsymbol{O(\sqrt{n})}$
- bw($M(d,d)$) = d
  - $\boldsymbol{O(\sqrt{n})}$

2D Torus



- A **Torus** $T(c,d)$ is a Mesh augmented by wraparound edges at the border of the mesh.
- $T(c,d)$ has $\boldsymbol{n = c \cdot d}$ processors
- degree($T(c,d)$) = 4
- diameter($T(c,d)$) = $d/2 + c/2$
- bw($T(c,d)$) = $\min\{2 \cdot c, 2 \cdot d\}$

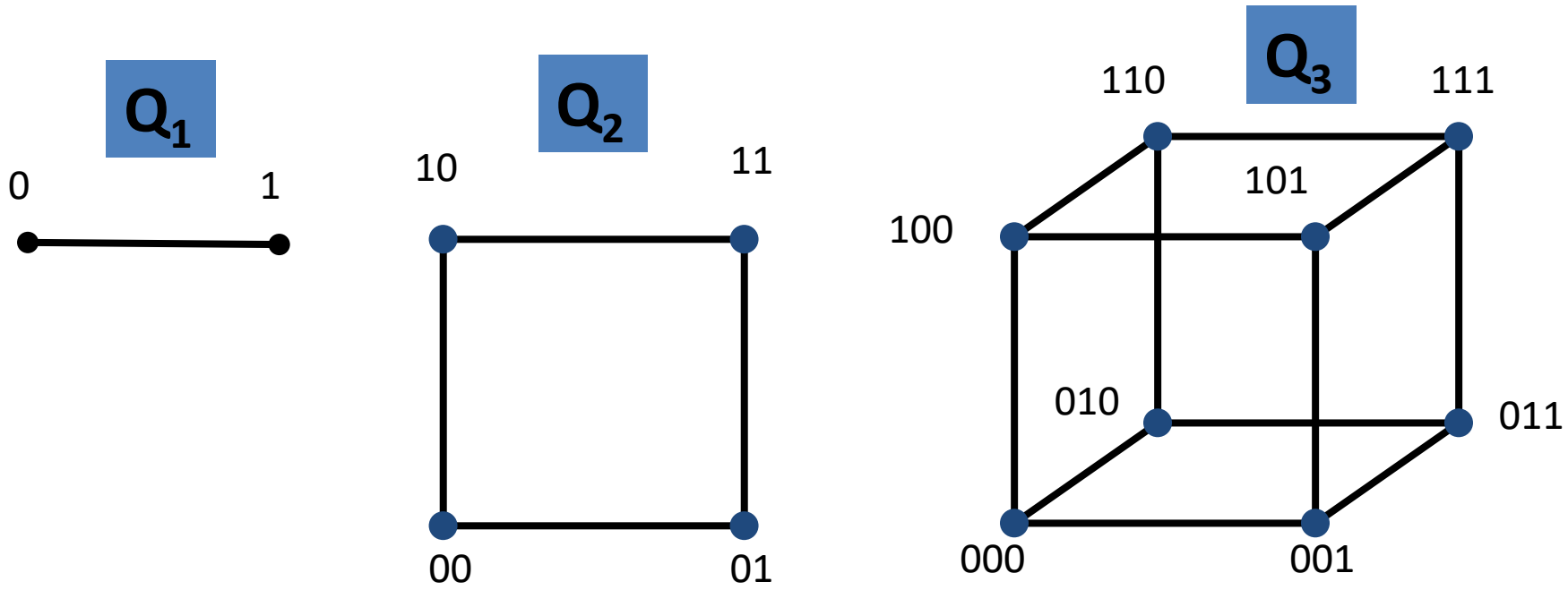
Binary Tree



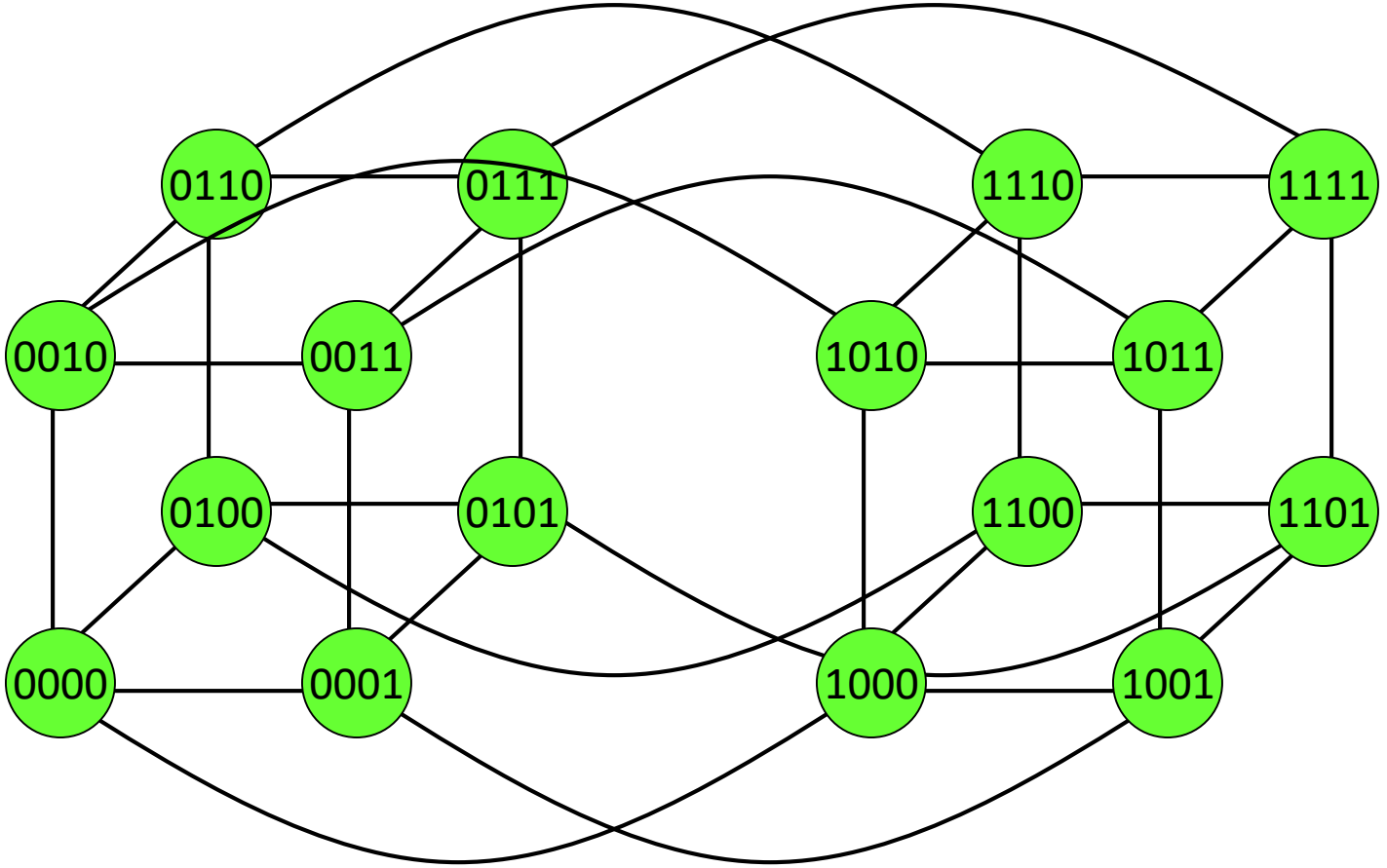
- BT(d) has $n = 2^d - 1$ processors
- degree(*BT(d)*) = 3
  - **O(1)**
- diameter(BT(*d*)) = 2(*d*–1)
  - **O(log(*n*))**
- bw(*BT(d)*) = 1
  - **O(1)**

Hypercube

The ***Hypercube***, denoted by Q_d ($d \geq 1$), is the graph that has vertices representing the 2^d bit strings of length d. Two vertices are adjacent if and only if the bit strings that they represent differ in exactly one bit position.



Hypercube



- HC(d) = Hypercube of degree d
  - Number of processors: $n = 2^d$
  - degree(*HC(d)*) = $d = O(\log(n))$
  - diameter(HC(d)) = $d = O(\log(n))$
  - bw(*HC(d)*) = $n/2 = O(n)$

Criteria to Evaluate Network Topologies

Topology	Degree	Diameter	Bisection-Width
Linear Array	$O(1)$	$O(n)$	$O(1)$
2D Mesh/Torus	$O(1)$	$O(\sqrt{n})$	$O(\sqrt{n})$
3D Mesh/Torus	$O(1)$	$O(\sqrt[3]{n})$	$O(n^{2/3})$
Binary Tree	$O(1)$	$O(\log(n))$	$O(1)$
Hypercube	$O(\log(n))$	$O(\log(n))$	$O(n)$

- ***Low*** **Diameter**
  - In order to support efficient communication between any pair of processors
- ***High*** **Bisection Width**:
  - A low bisection width can slow down many collective communication operations and thus can severely limit the performance of applications
  - However, achieving high bisection width may require non-constant network degree
- ***Constant*** degree (i.e. independent of network size)
  - allows a network to scale to a large number of nodes without the need of adding an excessive number of connections

Amdahl’s Law

- A formula for estimating speedup is named Amdahl's Law
- It states that no matter how many processors are used in a parallel run, a program's speedup will be limited by its fraction of sequential code.
- That is, almost every program has a fraction of the code that doesn't lend itself to parallelism. This is the fraction of code that will have to be run with just one processor, even in a parallel run.
- Gives an **upper bound** on the speedup that can be achieved.

Derivation of Amdahl's Law

$$T(1) = T_{\text{ser}} + T_{\text{par}}$$

part of the program that does not benefit from parallelization

part of the program that does can benefit from parallelization

- We now assume that the best possible speedup we can achieve is linear
- Then, we derive an upper bound for achievable speedup

$$T(p) \geq T_{\text{ser}} + \frac{T_{\text{par}}}{p}$$

$$S(p) = \frac{T(1)}{T(p)} \leq \frac{T_{\text{ser}} + T_{\text{par}}}{T_{\text{ser}} + \frac{T_{\text{par}}}{p}}$$

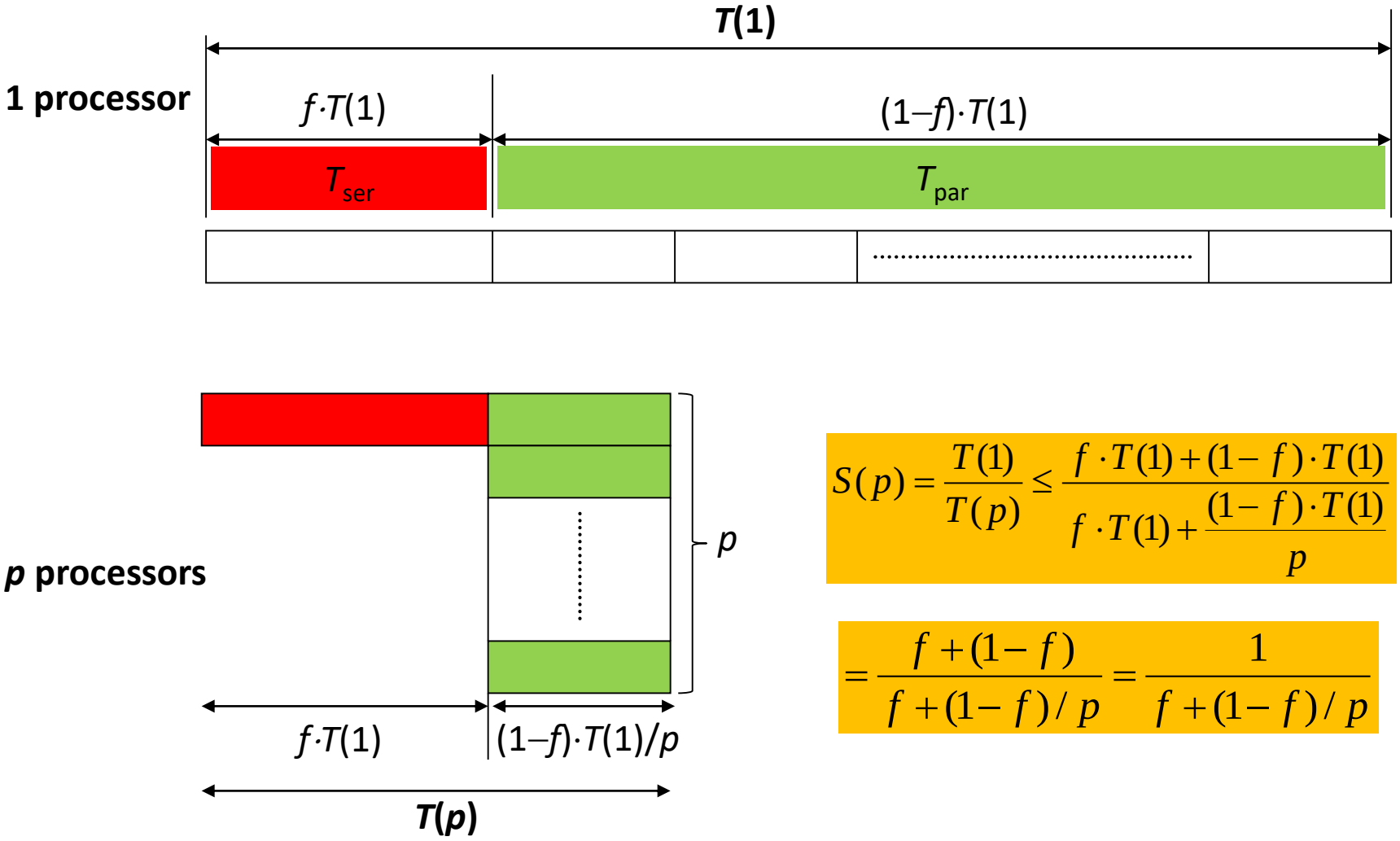
Derivation of Amdahl's Law

$$T_{\text{ser}} = f \cdot T(1) \text{ and } T_{\text{par}} = (1-f) \cdot T(1);\ \ (0 \leq f \leq 1)$$

- Instead of using absolute runtimes (T_{ser} and T_{par}), we now use their **fraction *f***
- Substituting this in the previously derived upper bound, results in **Amdahl's Law**; i.e. upper bound for the speedup that only depends on *f* and *p*

$$S(p) = \frac{T(1)}{T(p)} \leq \frac{T_{\text{ser}} + T_{\text{par}}}{T_{\text{ser}} + \frac{T_{\text{par}}}{p}} = \frac{f \cdot T(1) + (1-f) \cdot T(1)}{f \cdot T(1) + \frac{(1-f) \cdot T(1)}{p}} = \frac{1}{f + \frac{(1-f)}{p}}$$

Amdahl's Law



$$S(p) = \frac{T(1)}{T(p)} \le \frac{f \cdot T(1) + (1-f) \cdot T(1)}{f \cdot T(1) + \frac{(1-f) \cdot T(1)}{p}}$$

$$= \frac{f + (1-f)}{f + (1-f)/p} = \frac{1}{f + (1-f)/p}$$

Amdahl's Law

$$S(p) \leq \frac{1}{f + \frac{(1-f)}{p}}$$

- Where the term ***f*** stands for the fraction of operations done sequentially with just one processor, and the term **(1–*f*)** stands for the fraction of operations that can potentially be parallelized.
- The sequential fraction of code, ***f***, is a unit-less measure between 0 and 1.
- Amdahl's Law can be used to predict the performance of parallel programs

Amdahl's Law – Example

1. 95% of a program's execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 6 CPUs?

$$S(6) \leq \frac{1}{0.05 + \frac{(1 - 0.05)}{6}} = 4.8$$

2. 10% of a program's execution time is spent within inherently sequential code. What is the limit to the speedup achievable by a parallel version of the program?

$$S(\infty) \leq \lim_{p \to \infty} \frac{1}{0.1 + \frac{(0.9)}{p}} = 10$$

Scaled Speedup

- **Limitation of Amdahl's law**: only applies in situation where the problem size is constant and the number of processors varies ($\Rightarrow$ **strong scalability**)
- However, when using more processors we may also use larger problems sizes ($\Rightarrow$ **weak scalability**)
  - In this case the time spent in the parallelizable part (T_{par}) may grow faster in comparison to T_{ser}
- **Scaled Speedup:** incorporates such scenarios in the calculation of the achievable speedup.
- We now derive a more general law which allows for scaling with respect to the problem's complexity
  - **Gustafson's Law** is just a special case which can be used to predict the theoretically achievable speedup using multiple processors when the parallelizable part scales linearly with the problem size while serial part remains constant

Derivation of Scaled Speedup Law

$$T_{\alpha\beta}(1) = \alpha \cdot T_{\text{ser}} + \beta \cdot T_{\text{par}} = \alpha \cdot f \cdot T(1) + \beta \cdot (1 - f) \cdot T(1)$$

α: scaling function of the part of the program that does not benefit from parallelization with respect to the complexity of the problem size

β: scaling function of the part of the program that benefits from parallelization with respect to the complexity of the problem size

- We derive a **scaled upper bound** for the achievable speedup:

$$S_{\alpha\beta}(p) = \frac{T_{\alpha\beta}(1)}{T_{\alpha\beta}(p)} \leq \frac{\alpha \cdot f \cdot T(1) + \beta \cdot (1 - f) \cdot T(1)}{\alpha \cdot f \cdot T(1) + \frac{\beta \cdot (1 - f) \cdot T(1)}{p}}$$

$$= \frac{\alpha \cdot f + \beta \cdot (1 - f)}{\alpha \cdot f + \frac{\beta \cdot (1 - f)}{p}}$$

Derivation of Gustafson‘s Law

$$\gamma = \frac{\beta}{\alpha}$$

Ratio of the problem complexity scaling between the parallelizable part and the non-parallelizable part.

$$S_\gamma(p) \leq \frac{f + \gamma \cdot (1 - f)}{f + \frac{\gamma \cdot (1 - f)}{p}}$$

Using different functions for γ yields the following special cases:

1. γ **= 1** (i.e. $\alpha = \beta$): **Amdahl‘s Law**
2. $\gamma = p$ (e.g. $\alpha = 1$; $\beta = p$): Parallelizable part grows linear in p while the non-parallelizable part remains constant $\Rightarrow$ **Gustafson’s law:**

$$S(p) \leq f + p \cdot (1 - f) = p + f \cdot (1 - p)$$

Scaled Speedup – Example

- Suppose we have a parallel program that is 15% serial and 85% linearly parallelizable for a given problem size. Assume that the (absolute) serial time does not grow as the problem size is scaled.
  i. How much speedup can we achieve if we use 50 processors without scaling the problem?

$$S_{\gamma=1}(50) \leq \frac{f + \gamma \cdot (1-f)}{f + \frac{\gamma \cdot (1-f)}{p}} = \frac{1}{0.15 + \frac{0.85}{50}} = 5.99$$

  ii. Suppose we scale up the problem size by a factor of 100. How much speedup could we achieve with 50 processors?

$$S_{\gamma=100}(50) \leq \frac{f + \gamma \cdot (1-f)}{f + \frac{\gamma \cdot (1-f)}{p}} = \frac{0.15 + 100 \cdot 0.85}{0.15 + \frac{100 \cdot 0.85}{50}} = 46.03$$

Exercise

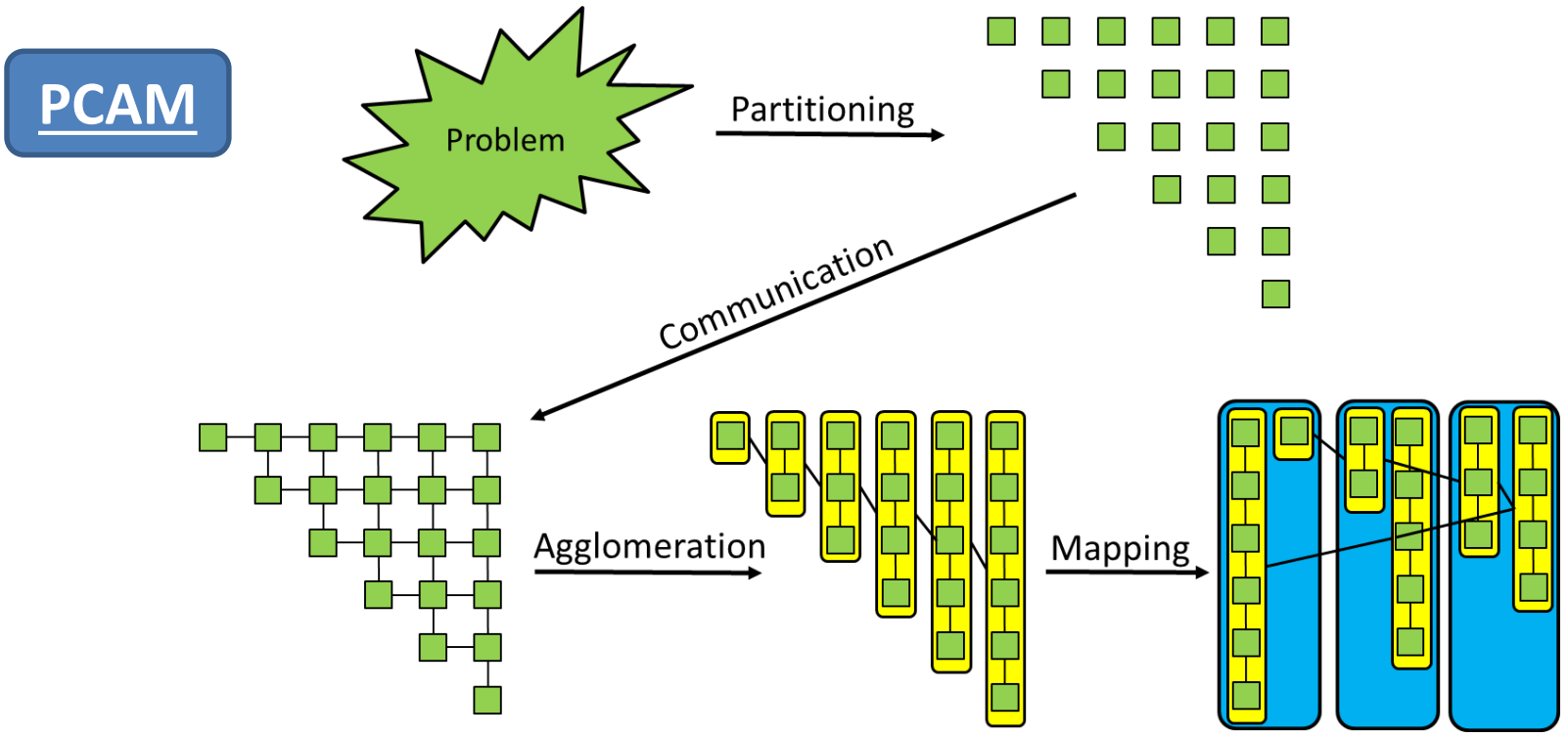
- Assume that you want to write a program that should achieve a speedup of 100 on 128 processors.
  i. What is the maximum sequential fraction of the program when this speedup should be achieved under the assumption of strong scalability?

$$100 = \frac{1}{f + \frac{(1-f)}{128}} = \frac{128}{128 \cdot f + 1 - f} = \frac{128}{127 \cdot f + 1} \Rightarrow f = 0.0022$$

  ii. What is the maximum sequential fraction of the program when this speedup should be achieved under the assumption of weak scalability whereby γ scales linearly?

$$100 = 128 + f \cdot (1 - 128) = 128 - 127 \cdot f \Rightarrow f = \frac{28}{127} = 0.22$$

Foster‘s Parallel Algorithm Design Method



1. **Partitioning:** decompose the problem into a large amount of small (*fine-grained*) tasks that can be executed in parallel
2. **Communication:** determine the required communication between tasks
3. **Agglomeration**: combine identified tasks into larger (*coarse-grained*) tasks in order to reduce communication by improving data locality
4. **Mapping:** assign the agglomerated to processes/threads in order to minimize communication, enable concurrency, and balance workload

Example: Jacobi Iteration

- Replace each value in the matrix by the average of its four neighbors
- Boundaries remain constant

Input Matrix: *x*[i][j]

–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
3	3	3	3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

xnew[i][j]

2	2	2	2	2	2	2	2	2	2
2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8
2.3	2.3	2.3	2.3	2.3	2.3	2.3	2.3	2.3	2.3
2	2	2	2	2	2	2	2	2	2
1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8	1.8
1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3
1	1	1	1	1	1	1	1	1	1
0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8
0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3
–0.3	–0.3	–0.3	–0.3	–0.3	–0.3	–0.3	–0.3	–0.3	–0.3

Jacobi Iteration

- Replaces all points of a given 2D matrix by the average of the values around it in **every iteration** step until convergence:

```
while (not converged) {
    for (int i=1; i<rows-1; i++)
        for (int j=1; j<cols-1; j++)
            buff[i*cols+j] = 0.25f*(data[(i+1)*cols+j] + data[i*cols+j-1]
                                  + data[i*cols+j+1] + data[(i-1)*cols+j]);

    memcpy(data,buff,rows*cols*sizeof(float));
}
```

- Boundary values are fixed:

```
x[0][j]
x[n-1][j]
x[i][0]
x[i][n-1]
```

Example: Jacobi Iteration

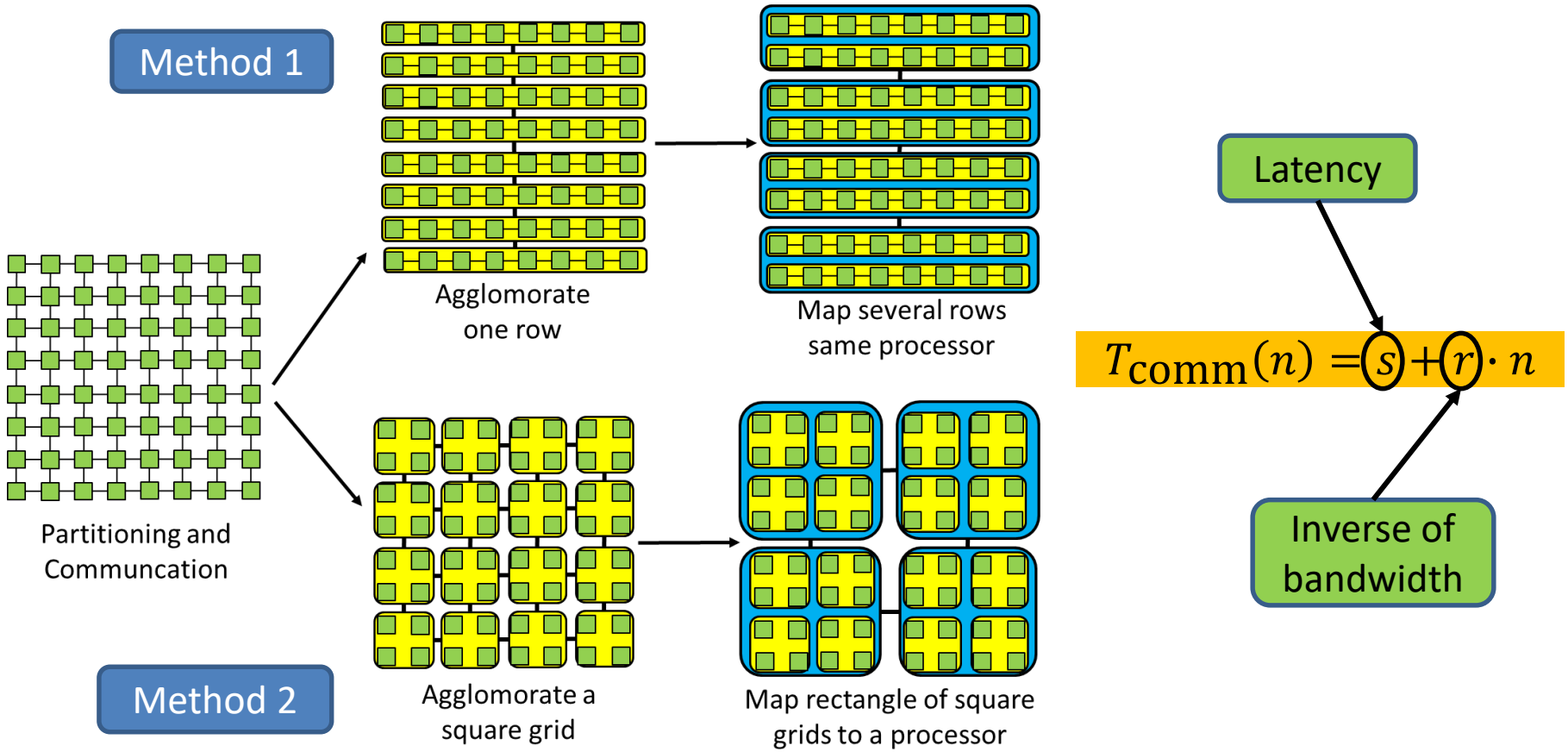
x[i][j] after **25** iterations

–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
3	1.0	0.2	–0.1	–0.3	–0.3	–0.3	–0.3	–0.1	0.2	1.0	3
3	1.6	0.9	0.5	0.4	0.3	0.3	0.4	0.5	0.9	1.6	3
2	1.6	1.2	0.9	0.8	0.7	0.7	0.8	0.9	1.2	1.6	2
2	1.6	1.3	1.1	1.0	0.9	0.9	1.0	1.1	1.3	1.6	2
2	1.5	1.3	1.1	1.0	1.0	1.0	1.0	1.1	1.3	1.5	2
1	1.1	1.0	1.0	0.9	0.9	0.9	0.9	1.0	1.0	1.1	1
1	0.9	0.8	0.7	0.7	0.7	0.7	0.7	0.7	0.8	0.9	1
1	0.6	0.5	0.4	0.3	0.3	0.3	0.3	0.4	0.5	0.6	1
0	0.1	0.0	0.0	–0.1	–0.1	–0.1	–0.1	0.0	0.0	0.1	0
0	–0.3	–0.5	–0.5	–0.5	–0.5	–0.5	–0.5	–0.5	–0.5	–0.3	0
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

x[i][j] after **75** iterations

–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
3	0.9	0.1	–0.3	–0.4	–0.5	–0.5	–0.4	–0.3	0.1	0.9	3
3	1.5	0.7	0.3	0.1	0.0	0.0	0.1	0.3	0.7	1.5	3
2	1.5	1.0	0.6	0.4	0.3	0.3	0.4	0.6	1.0	1.5	2
2	1.5	1.1	0.8	0.6	0.5	0.5	0.6	0.8	1.1	1.5	2
2	1.4	1.0	0.8	0.6	0.5	0.5	0.6	0.8	1.0	1.4	2
1	1.0	0.8	0.6	0.5	0.5	0.5	0.5	0.6	0.8	1.0	1
1	0.8	0.6	0.4	0.4	0.3	0.3	0.4	0.4	0.6	0.8	1
1	0.5	0.3	0.2	0.1	0.1	0.1	0.1	0.2	0.3	0.5	1
0	0.0	–0.1	–0.2	–0.2	–0.3	–0.3	–0.2	–0.2	–0.1	0.0	0
0	–0.4	–0.5	–0.6	–0.6	–0.6	–0.6	–0.6	–0.6	–0.5	–0.4	0
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

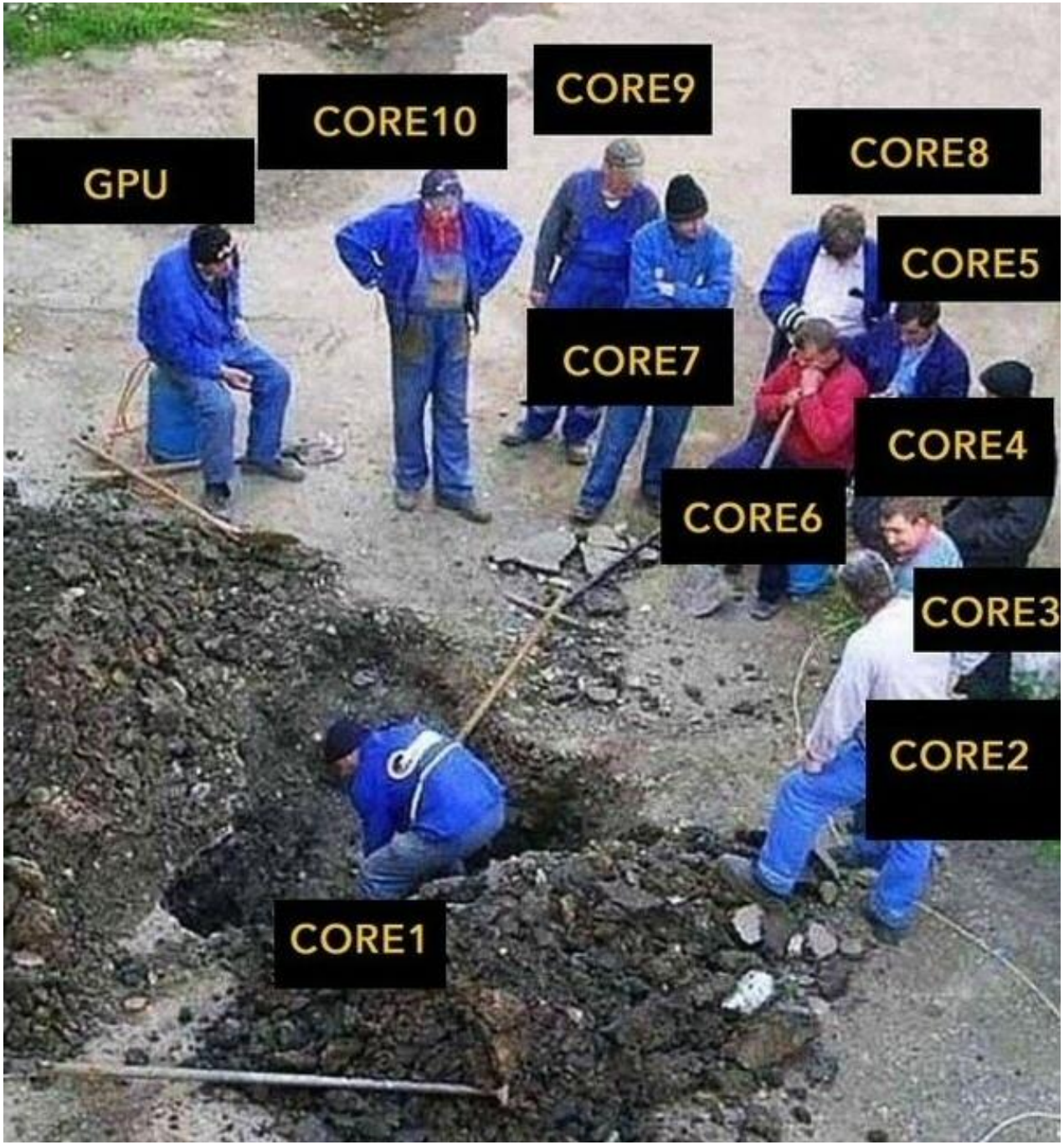
Two Schemes for Jacobi Iteration



- **Method 1:** Communication between two procs $\approx 2(s + r \cdot n)$
- **Method 2:** Communication between two procs $\approx 4\left(s + r\left(\frac{n}{\sqrt{p}}\right)\right)$
- $\Rightarrow$Method 2 superior for large *p* since communication time decreases with *p* while it remains constant for Method 1.

Review Questions

- What are diameter and bisection width of a hypercube?
- How do you derive Amdahl‘s and Gustafson’s law?
- What are some of their limitations?
- Explain Foster‘s Parallel Algorithm Design Method
- Why is 2D decomposition superior to 1D decomposition for Jacobi Iteration?


GPU
CORE10
CORE9
CORE8
CORE5
CORE7
CORE4
CORE6
CORE3
CORE2
CORE1