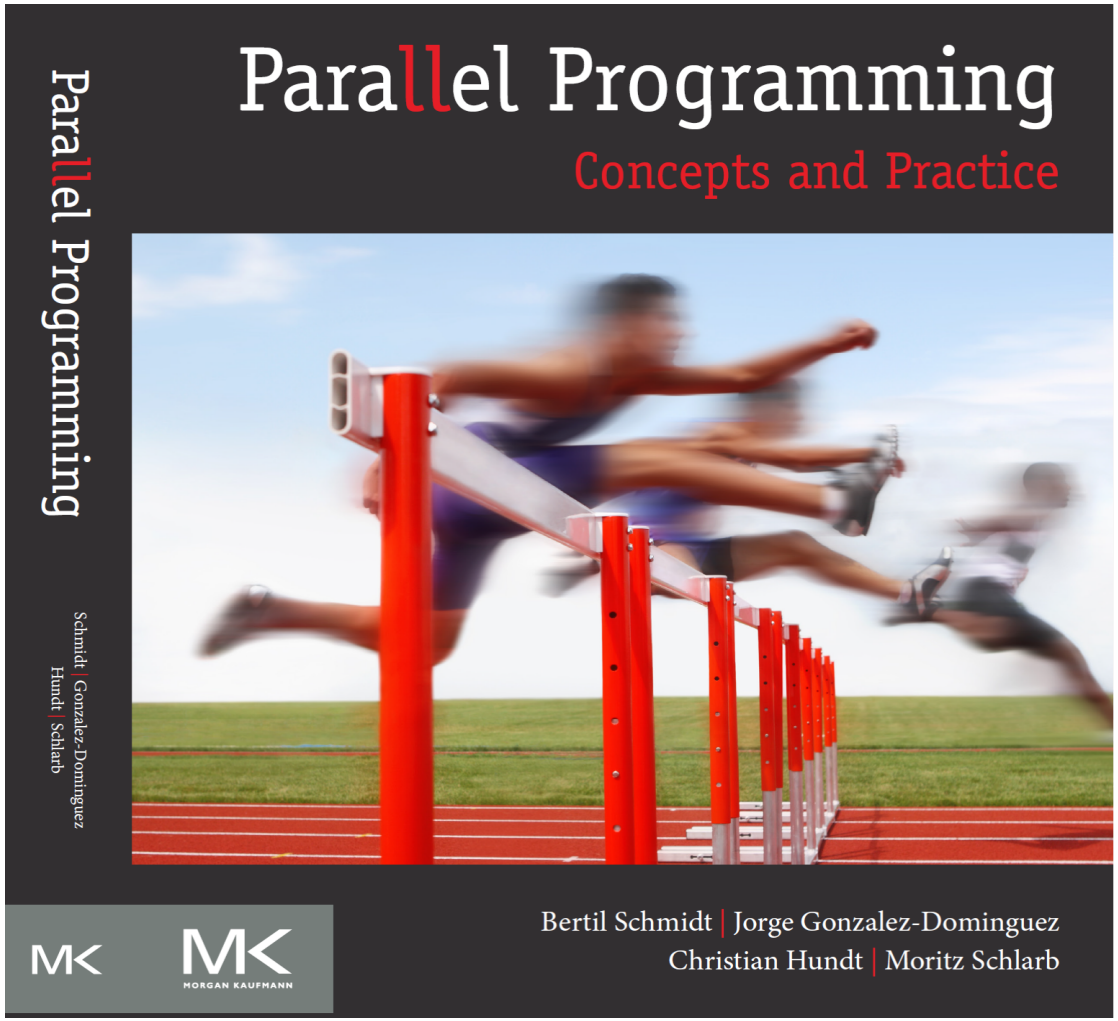
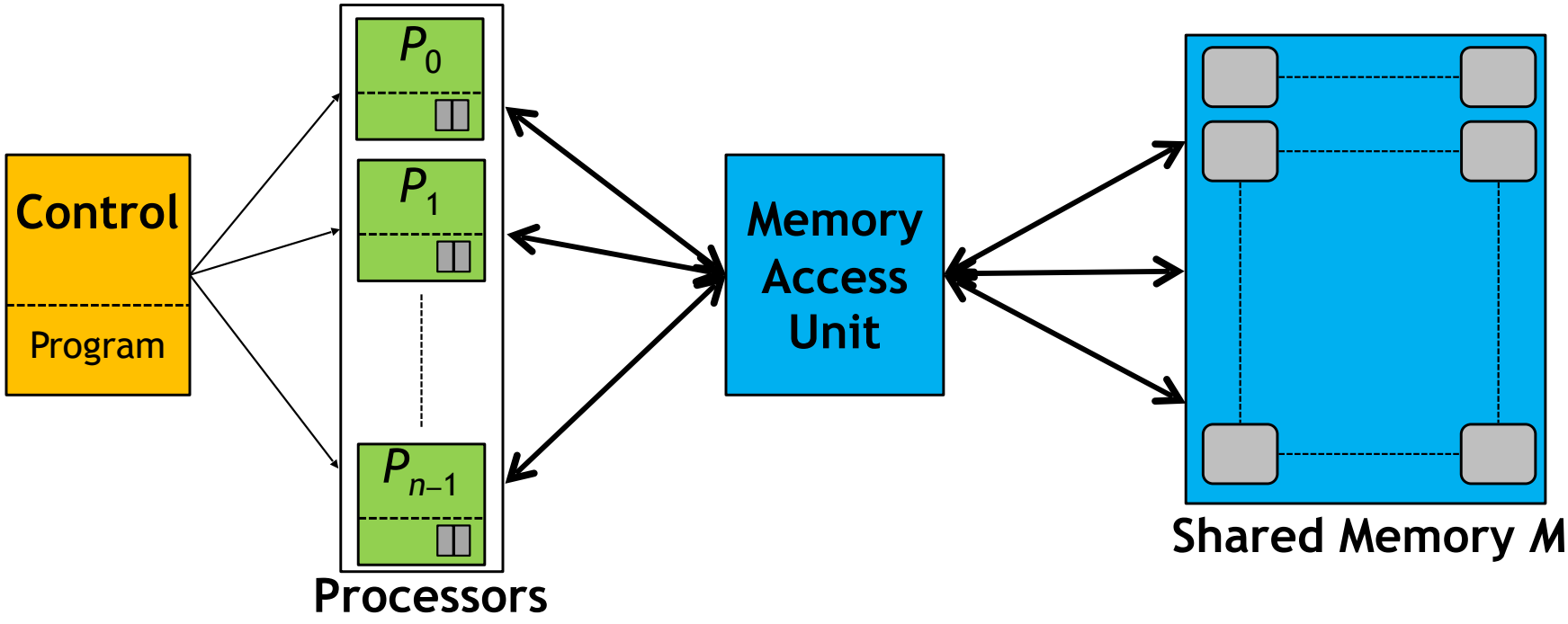
Chapter 02: Theoretical Background



Parallel Random Access Machine (PRAM)



- *n* processors $P_0,\ldots,P_{n-1}$ are connected to a global shared memory *M*
- Any memory location is uniformly accessible from any processor in constant time
- Communication between processors can be implemented by reading and writing to the globally accessible shared memory.

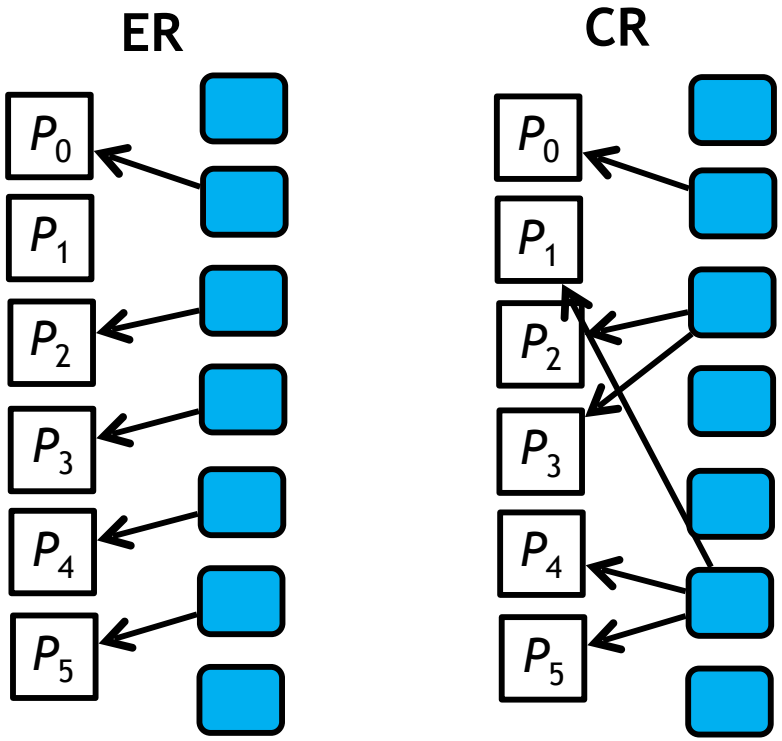
PRAM

- n identical processors P_i, $i = 0,\ldots,n-1$ operate in lock-step
- In every step each processor executes an instruction cycle in three phases:

1. **Read phase**: Each processor can simultaneously read a single data item from a (distinct) shared memory cell and store it in a local register.
2. **Compute phase:** Each processor can perform a fundamental operation on its local data and subsequently stores the result in a register.
3. **Write phase:** Each processor can simultaneously write a data item to a shared memory cell, whereby the exclusive write PRAM variant allows writing only distinct cells while concurrent write PRAM variant also allows processors to write to the same location (race conditions).

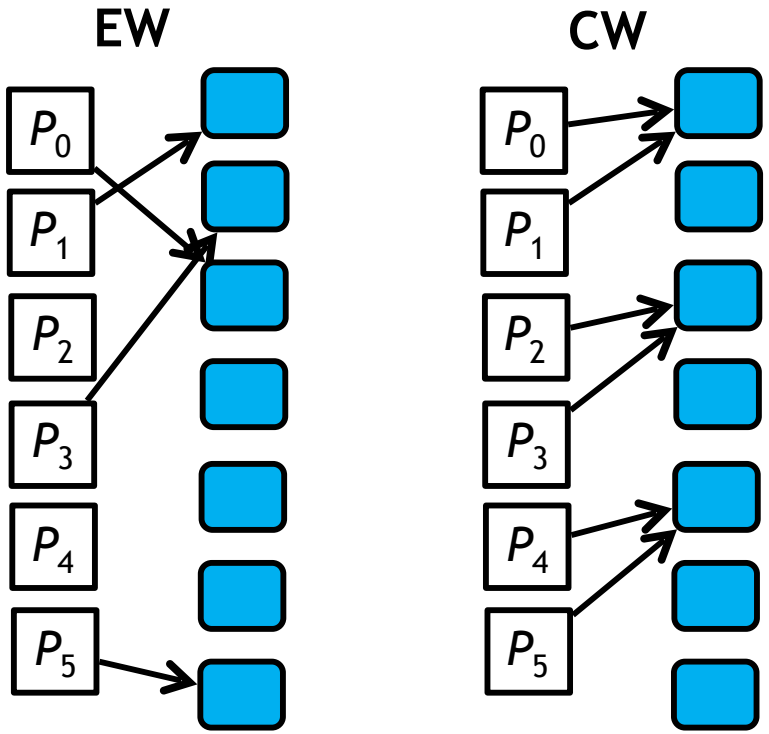
- **Uniform complexity analysis:** Each step on the RAM takes $O(1)$ time

PRAM Variants



- **Exclusive Read Exclusive Write (EREW):** No two processors are allowed to read or write to the same shared memory cell during any cycle
- **Concurrent Read Exclusive Write (CREW):** Several processors may read data from the same shared memory cell simultaneously. Still, different processors are not allowed to write to the same shared memory cell

PRAM Variants



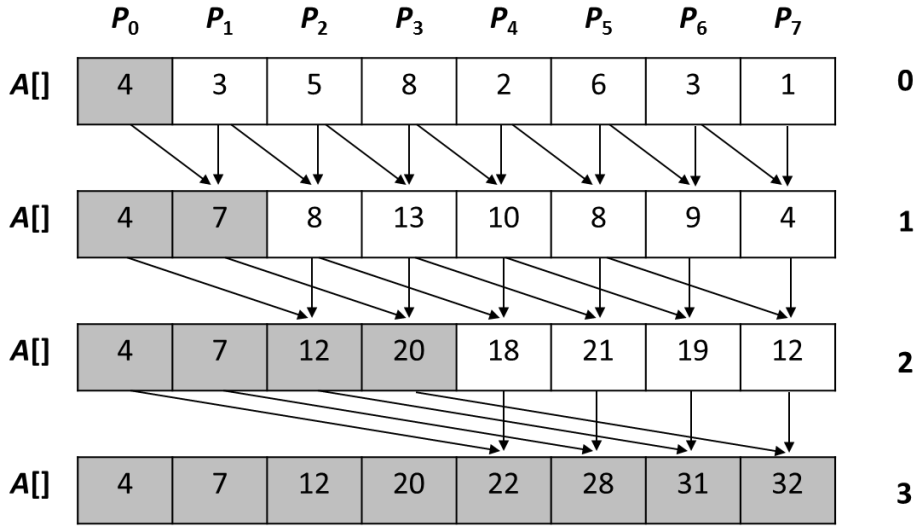
- **Concurrent Read Concurrent Write (CRCW):** Both simultaneous reads and writes to the same shared memory cell are allowed. In case of a simultaneous write (analogous to a race condition) we further specify which value will actually be stored:

1. **Priority CW:** Processors have been assigned distinct priorities and the one with highest priority succeeds writing
2. **Arbitrary CW:** A randomly chosen processor succeeds writing its value.
3. **Common CW:** If the values are all equal, then this common value is written, otherwise, the memory location is unchanged.
4. **Combining CW:** All values to be written are combined into a single value by means of an associative binary operations (e.g. sum, product, minimum, logical AND)

Parallel Prefix Computation

- Binary **associative** operation $\circ$ on the set $\mathbb{X}$; i.e. $\circ: \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{X}$
  - $(x_i \circ x_j) \circ x_k = x_i \circ (x_j \circ x_k)$ for all $x_i, x_j, x_k \in \mathbb{X}$.
  - Examples: Addition, Multiplication, Minimum, Maximum, String concatenation, boolean AND/OR
- $X = \{x_0, \ldots, x_{n-1}\}$, $x_i \in \mathbb{X}$ for all $i = 0,\ldots,n-1$. We want to compute
  - $s_0 = x_0$
  - $s_1 = x_0 \circ x_1$
  - :
  - $s_{n-1} = x_0 \circ x_1 \circ \ldots \circ x_{n-1}$.
  - In other words, $s_0 = x_0$ and $s_i = s_{i-1} \circ x_i$ for $i = 1,\ldots,n-1$
- Obtaining $S = \{s_0, \ldots, s_{n-1}\}$ from $X = \{x_0, \ldots, x_{n-1}\}$ is called **prefix computation**
- Example: "MINIMUM"
  - Input: {39, 21, 20, 50, 13, 18, 2, 33, 49, 39, 47, 15, 30, 47, 24, 1}
  - Output: {39, 21, 20, 20, 13, 13, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1}

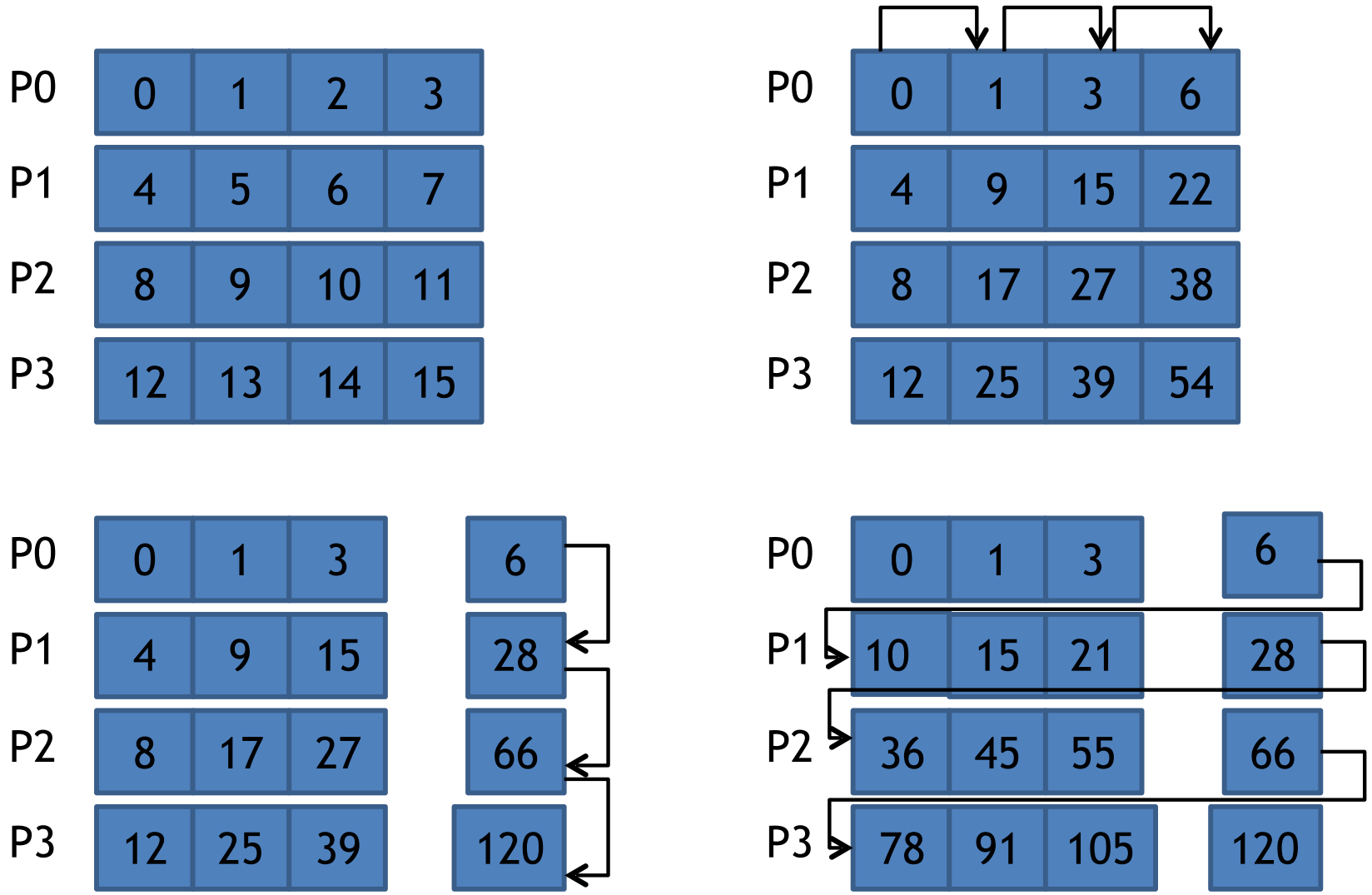
Parallel Prefix on a PRAM using recursive doubling on *n* Processors




```
for (j = 0; j<n; j++) do_in_parallel        // each processor j
    reg_j = A[j];                           // copies one value to a local register
for (i = 0; i<ceil(log(n)); i++) do             // sequential outer loop
    for (j = pow(2, i); j<n; j++) do_in_parallel // each proc. j
        reg_j += A[j - pow(2, i)];              // performs computation
        A[j] = reg_j;                           // writes result to shared memory
}
```

- $C(n) = T(\mathrm{p},n) \times p = O(\log n) \times n = O(n \times \log n)$
- Thus, this algorithm is **NOT** cost-optimal

Cost-optimal Parallel Prefix on a PRAM



- Use $p = n/\log(n)$ processors
- $C(n) = T(\mathrm{p},n) \times p = O(\log n) \times n/\log(n) = O(n)$
- Thus, this algorithm is cost-optimal

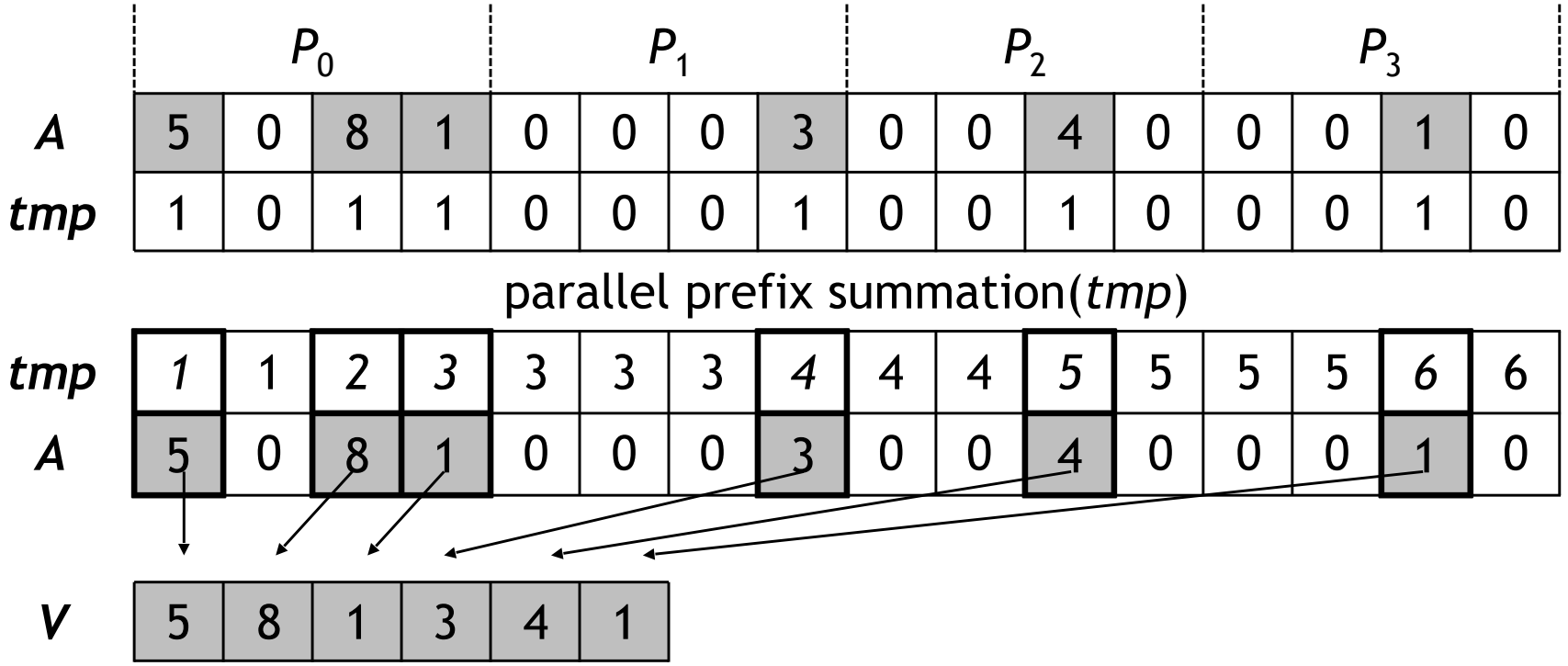
Cost-optimal Parallel Prefix on an EREW PRAM using $p = n/\log(n)$ processors

```
// Stage 1: each Processor i computes a local prefix sum
// of a subarray of size n/p = log(n) = k
for (i = 0; i<n / k; i++) do_in_parallel
    for (j = 1; j<k; j++) do
        A[i*k+j] += A[i*k+j-1];

// Stage 2: Prefix summation using only the rightmost value
// of each subarray (O(log(n/k)))
for (i = 0; i<log(n / k); i++) do
    for (j = pow(2, i); j<n / k; j++) do_in_parallel
        A[j*k-1] += A[(j-pow(2, i))*k-1];

// Stage 3: each Proc i adds the value computed in Step 2 by Proc i-1 to
// each subarray element except for the last one
for (i = 1; i<n / k; i++) do_in_parallel
    for (j = 0; j<k - 1; j++) do
        A[i*k+j] += A[i*k+j-1];
```

Sparse Array Compaction on a PRAM

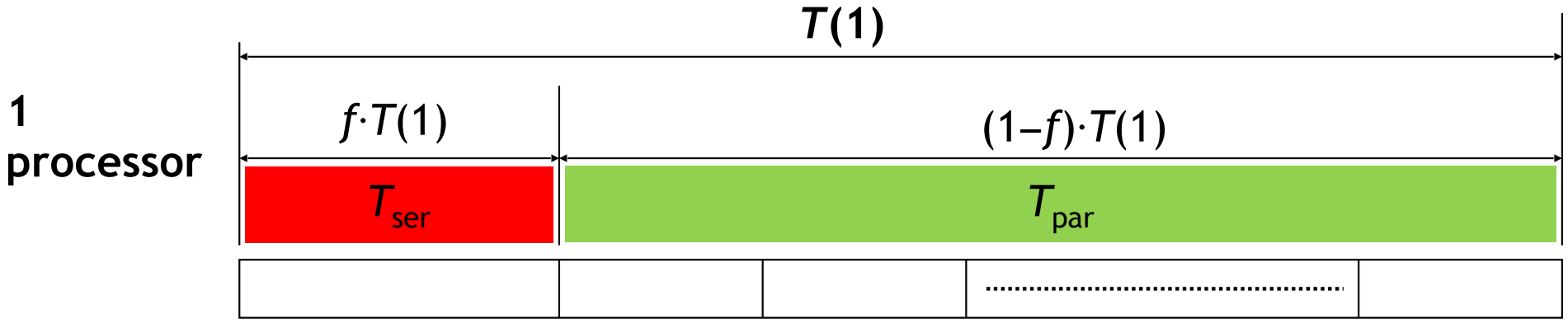


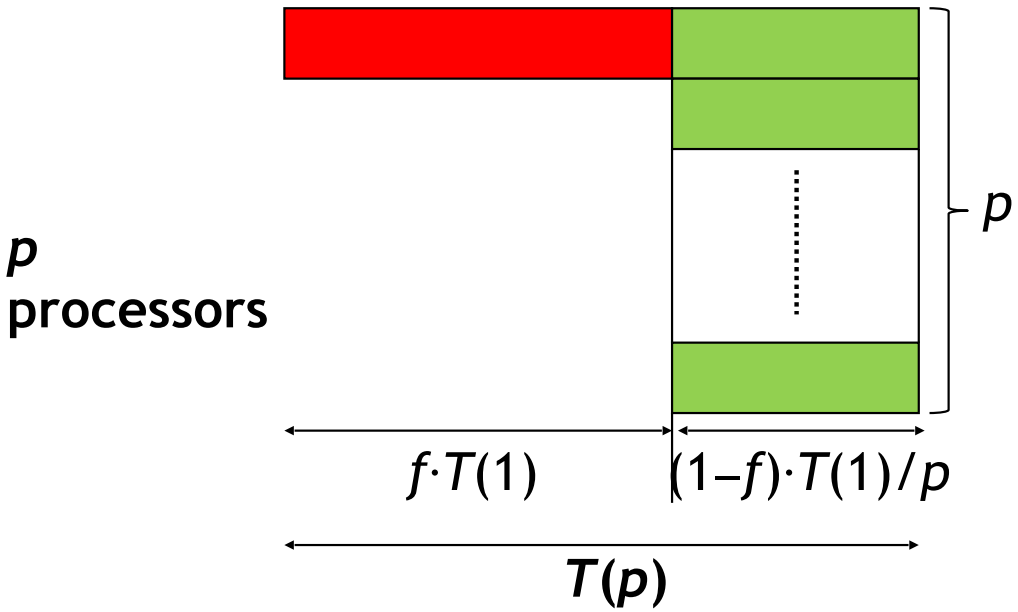
- Use $p = n/\log(n)$ processors
- $C(n) = T(\mathrm{p},n) \times p = O(\log n) \times n/\log(n) = O(n)$

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.

Amdahl's Law





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

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

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?

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

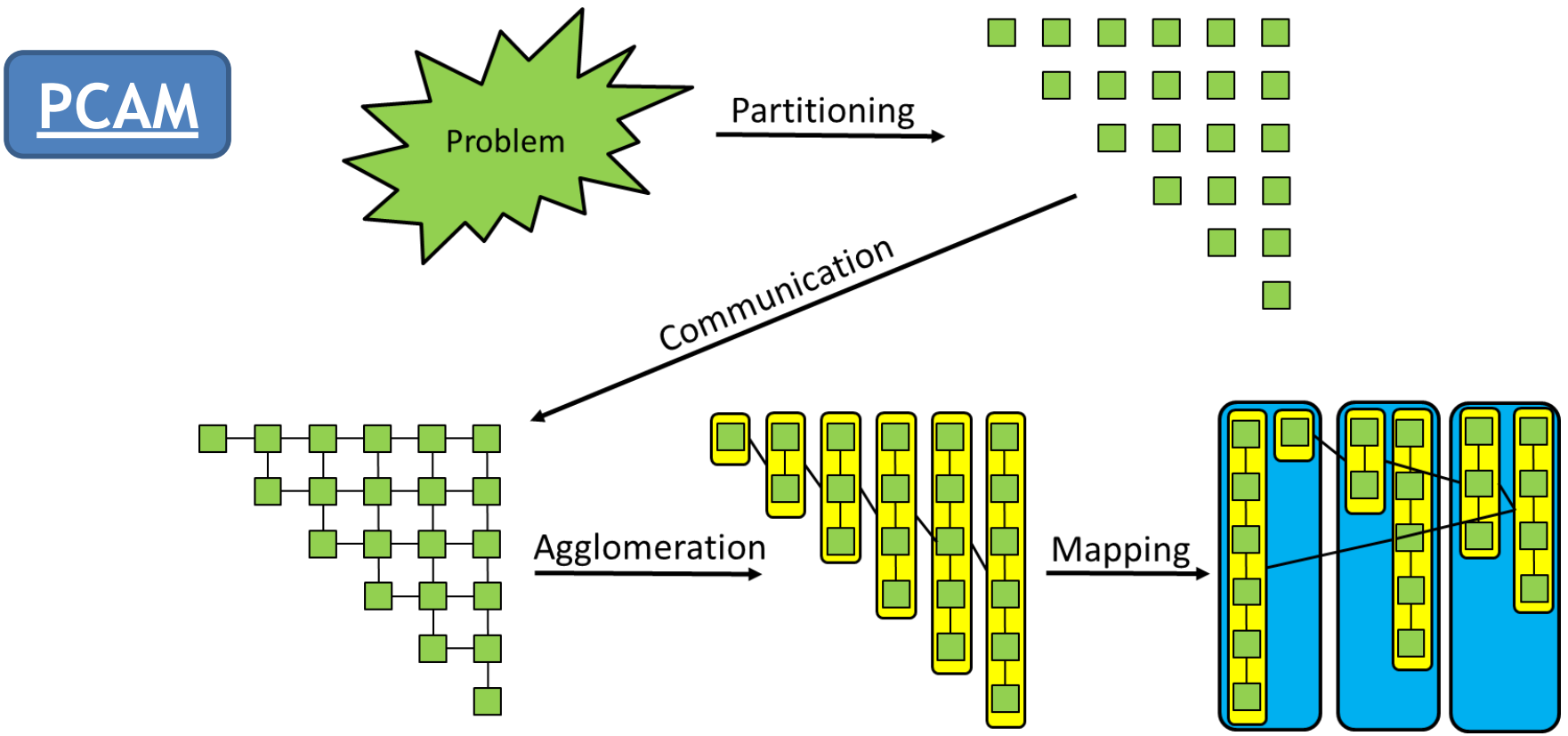
2. 10% of a program's execution time is spent within inherently sequential code. What is the limit to the speedup achievable by a parallel version of the program?

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

Scaled Speedup

- **Limitation of Amdahl's law**: only applies in situation where the problem size is constant and the number of processors varies (⇒ **strong scalability**)
- However, when using more processors we may also use larger problems sizes (⇒ **weak scalability)**
- **Scaled Speedup:** incorporates such scenarios in the calculation of the achievable speedup.

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

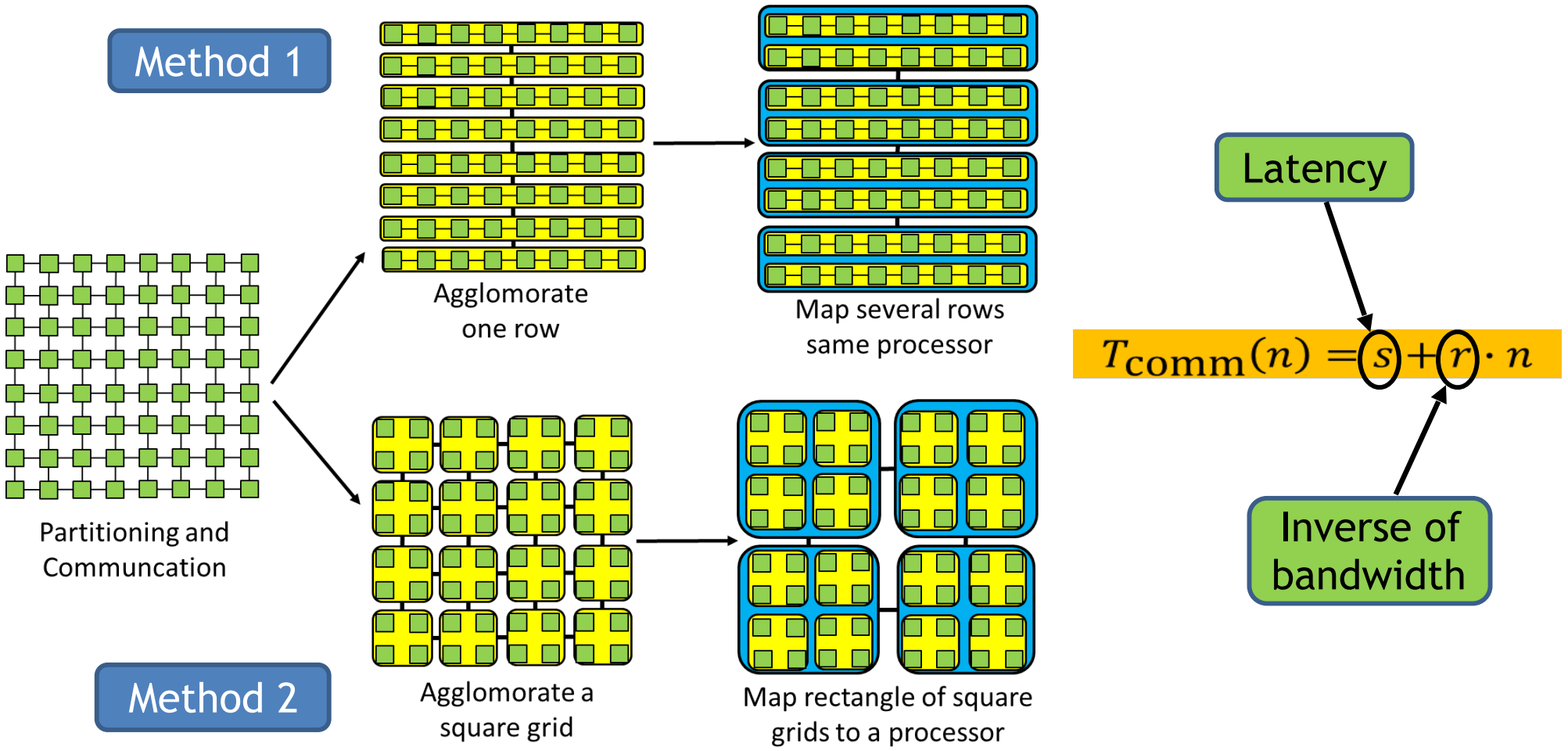
x[i][j] after **25** iterations

–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.