Parallel Computing Theory

Week 3

Amittai Aviram – 22 September 2020

Background: System Concepts

Processes and Threads

Definitions (1)

- **Program:** A stored sequence of instructions.
- **Process:** A program *in flight*. This implies
  - Creation of a memory *address space*
  - Creation of a *stack*
  - Loading of the address of the first instruction into the processor’s *instruction pointer register*
  - The *scheduling* of the process onto an available processor by the OS.
- **Forking a Process:** When the OS creates a *child process* from an existing *parent process*.
  - The OS creates an exact copy of the parent process’s stack for the child process in its state at the moment of forking, which is now *private to the child process*.
  - The OS provides a new, *independent*, memory address space for the child process.
  - The OS starts execution of the child process at the instruction given as the new start instruction in the *fork* call.
  - From now on, the two processes are independent and do not share any data unless they communicate data across files.
-

Processes and Threads
Definitions (2)

- **Kernel Thread:** a “lightweight process.”
  - The OS devotes a processor to execute instructions starting at the given new start instruction.
  - The address space and stack are *shared*, except for special *thread-local storage*.
  - When a thread executes a function, it creates its own private stack frame, just as a normal process would do.
- **User-Level Thread:** a sequence of instructions performed within a process.
  - A user-level library manages the switching of threads.
  - The containing process does not execute more than one instruction at the same time, but the program may appear to the programmer as if it had simultaneous threads.

Process Address Space (Cloned)

What a Process Sees

```
from multiprocessing import Process, Value


def foo(x: int, y: int, z: Value):
    print(z.value)
    z.value = x + y


if __name__ == '__main__':
    arg0 = 43
    arg1 = 37
    result = Value('i')
    proc = Process(target=foo, args=(arg0, arg1, result))
    # Still in the main process:
    print(result.value)
    proc.start()
    result.value = 17
    print(result.value)
    proc.join()
    print(result.value)
```

The child process *proc* inherits these.

Copies the stack for *proc*
but creates a new (private) stack
frame for *foo*.

proc sees this
because *result* is
shared (communicated).

proc's stack is lost,
but *result* is communicated
back.

Output:
0
0
17
80

Thread Address Space (Shared)

What a Thread Sees

```python
from threading import Thread
from typing import List


def foo(x: int, y: int, z: List[int]):
    print(z[0])
    z[0] = x + y


if __name__ == '__main__':
    arg0 = 43
    arg1 = 37
    result = [0]
    thread = Thread(target=foo, args=(arg0, arg1, result))
    # Still in the main thread:
    print(result[0])
    thread.start()
    print(result[0])
    result[0] = 17
    thread.join()
    print(result[0])
```

Passed by reference and mutable.

The child thread executes within the same address space.

Creates a new (private) stack frame for *foo*.

Race condition!

thread's stack frame for foo is lost, but *result* is visible to the main thread because it is all in the same address space.

Output (nondeterministic):
0
0
80
17

Processes, Threads, and the OS

Scheduling

- A process is itself an *abstraction*.
- Real OSs schedule the currently-running processes on the available processors.
- We may usually simplify by assuming a one-to-one correspondence between processes and processors.
- This simplifying assumption often makes the terms *process*, *processor*, and *thread* all interchangeable *when we speak at a high level of abstraction*.
- It is when we *implement* a parallel algorithm that we must observe the differences!

Python and C++

Parallel Execution Support

- Python
  - Parallel processes — multiprocessing.Process
  - No kernel threads!
  - User-level threads — threading.Thread —
    These *look like* kernel threads, but do not execute at the same time.
  - The Python interpreter requires that any running thread acquire a mutual-exclusion (mutex) *lock*, which guarantees that threads cannot run concurrently.
  - See https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock for details.
- C++
  - Parallel processes — <unistd> fork
  - Kernel threads — <threading> thread
  - User-level threads — various libraries, e.g.,
    http://homepage.divms.uiowa.edu/~jones/opsys/threads/

The Parallel Random Access Machine (PRAM)

The Parallel Random Access Machine

A Tool for Analyzing High-Level Parallel Behavior

- The PRAM has n *identical* processors—$P_0, P_1, \ldots, P_{n-1}$.
- All processors *share memory*.
- The processors work in *lock-step*.
- Each step has three optional phases, which all processors perform simultaneously:
    - Read
    - Compute
    - Write
- Communication occurs only through reading from and writing to shared memory.
- Any processor can access any location in shared memory in the same amount of time (one unit).

PRAM Memory Access

Schemes

- Concurrent Read (CR)
- Exclusive Read (ER)
- Concurrent Write (CW)
- Exclusive Write (EW)

PRAM Variants

Working with Memory Access Rules

- EREW — safest and slowest
- CREW
- CRCW — fastest and most dangerous — to manage race conditions:
  - Give each process a fixed *priority*
  - Only write if all writers agree on the new value
  - Write an *aggregate* according to some aggregation function.

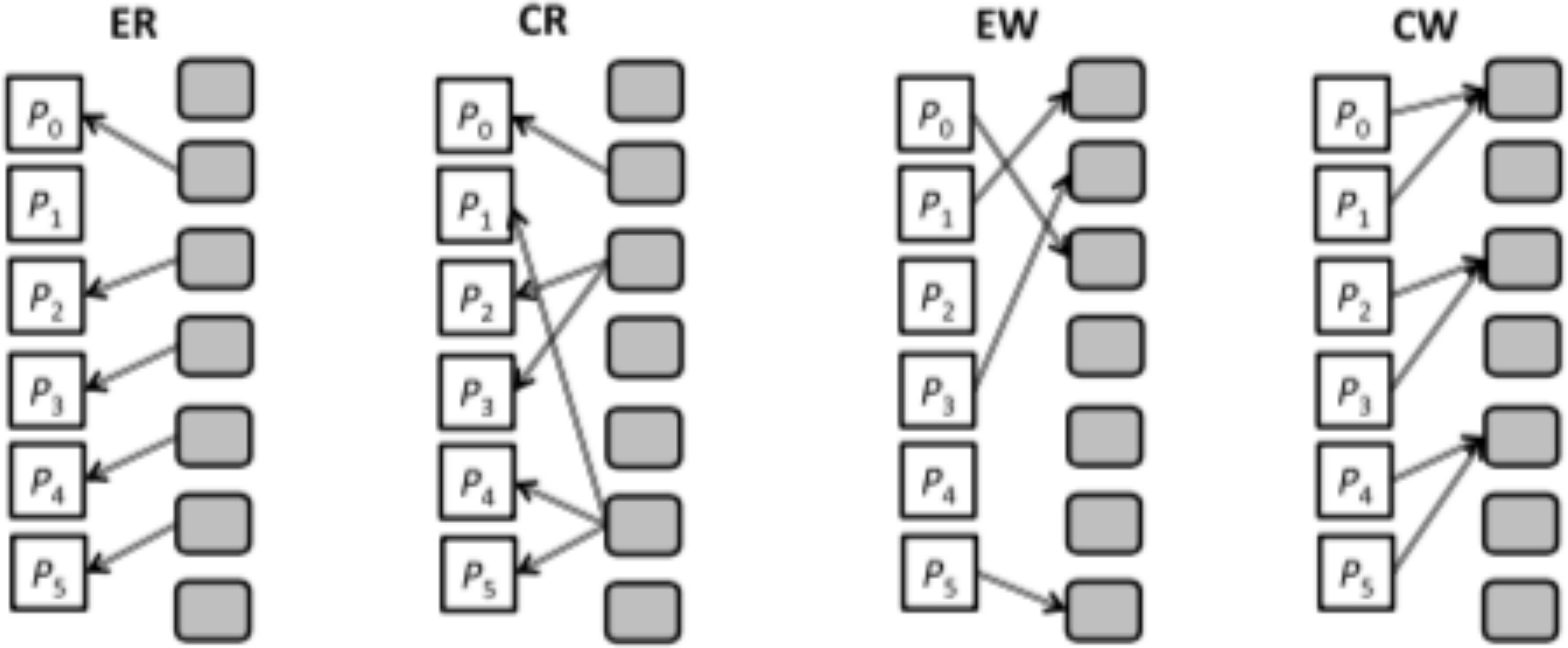


FIGURE 2.2

The four different variants of reading/writing from/to shared memory on a PRAM.

Example: Parallel Prefix Sum (Again!)

From a Sequential to a (Naïve Parallel) Implementation

- Sequential:
  `for (i = 0; i < n; i++)  A[i] += A[i - 1]` $C(n) \in \mathcal{O}(n) \times 1 = \mathcal{O}(n)$
- Parallel:
  - Simply assign one processor to each item *i*.
  - $p = n$ processors.
  - Shared memory makes communication back unnecessary.

Parallel Prefix Sum

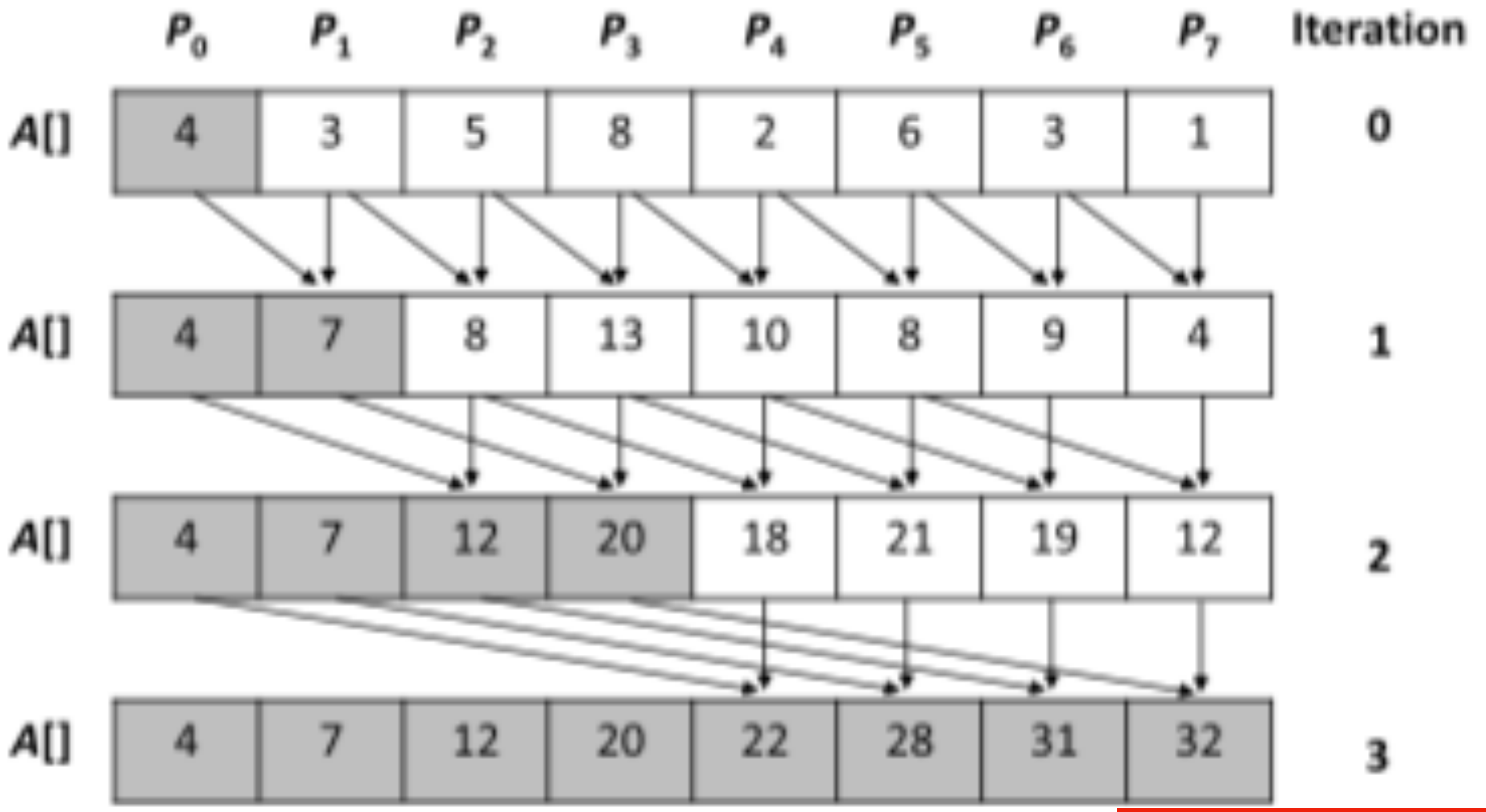
Revealing Inefficiency with a PRAM

```
// With n processors:

// Each processor copies an array entry to a local register.
for (j = 0; j < n; j++) do_in_parallel
    reg_j = A[j];

// Sequential outer loop:
for (i = 0; i < ceil(log2(n)); i++) do
    // Parallel inner loop performed by Processor j:
    for (j = pow(2, i); j < n; j++) do_in_parallel {
        reg_j += A[j - pow(2, i)];
        A[j] = reg_j;
    }
}
```

Parallel prefix summation of an array *A* of size n stored in shared memory on an EREW PRAM with *n* processors.

Parallel Prefix Sum

Revealing Inefficiency with a PRAM



$$C(n) \in \mathcal{O}\left(\log(n)\right) \times n = \mathcal{O}\left(n \log(n)\right)$$

FIGURE 2.3

Parallel prefix summation of an array *A* of size 8 on a PRAM with eight processors in three iteration steps based on recursive doubling.

Parallel Prefix Sum

Improvements — $p = \frac{n}{\log_2(n)}$

1. Partition the n input values into chunks of size $log_2(n)$. Each processor computes local prefix sums of the values in one chunk in parallel (takes time $O(\log(n))$).

2. Perform the old non-cost-optimal prefix sum algorithm on the $\frac{n}{\log_2(n)}$ partial results (takes time $O\left(\log\left(n/\log(n)\right)\right)$).

3. Each processor adds the value computed in Stage 2 by its left neighbor to all values of its chunk (takes time $O(n)$).

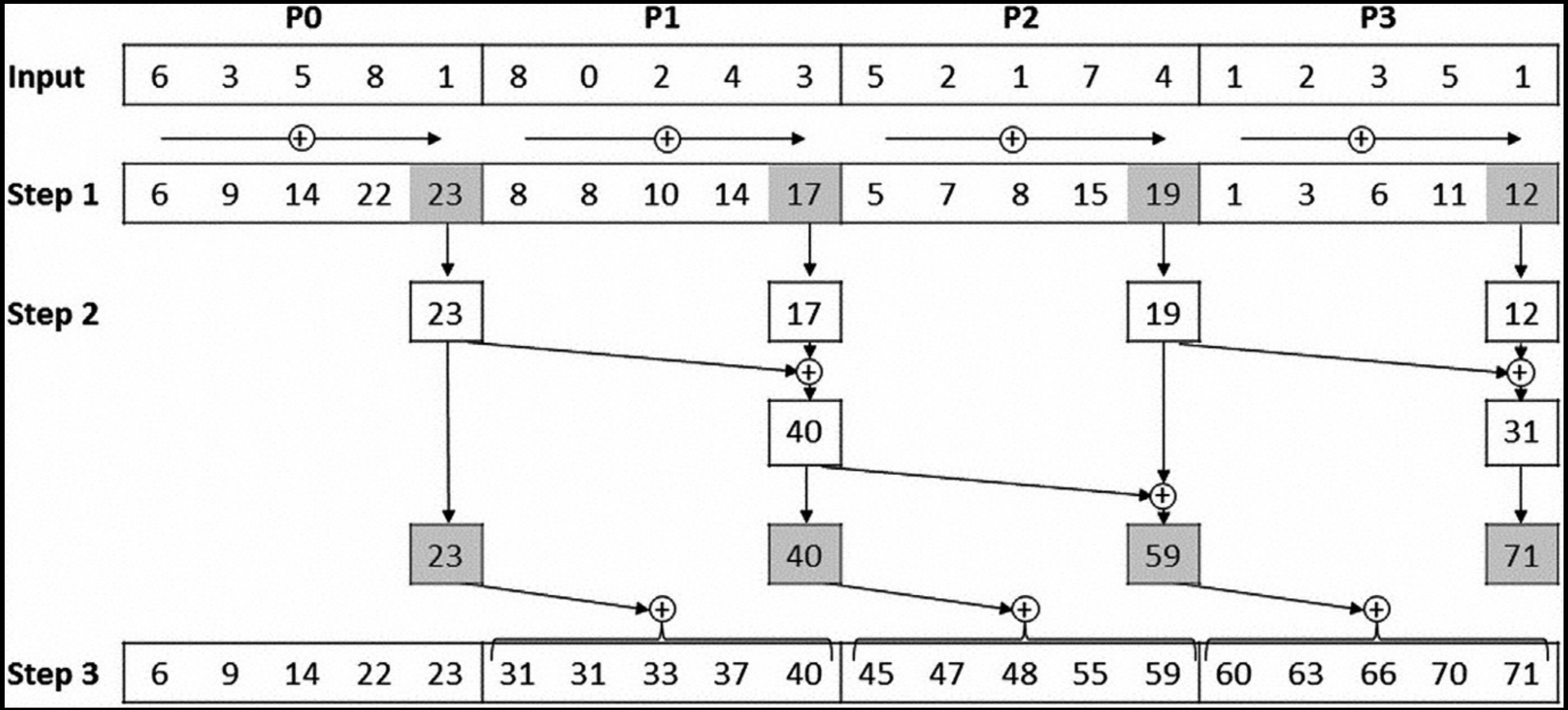
Parallel Prefix Sum

Improved Version — $p = \frac{n}{\log_2(n)}$

```
// Stage 1: each processor i computes a local
// prefix sum of an array of size n/p = log2(n) = k
for (i = 0; i < n/k; i++) do_in_parallel
    A[i * k + j] += A[i * k + j - 1]

// Stage 2: prefix sum computation using only the rightmost value
// of each subarray, which takes O(log(n/k)) steps
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 - k * pow(2, i)) * k - 1];

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

Parallel Prefix Sum

Improved Version



$$C(n) \in \mathcal{O}\left(\log(n)\right) \times \frac{n}{\log(n)} = \mathcal{O}(n)$$

Sparse Array Compaction

Sequential Algorithm

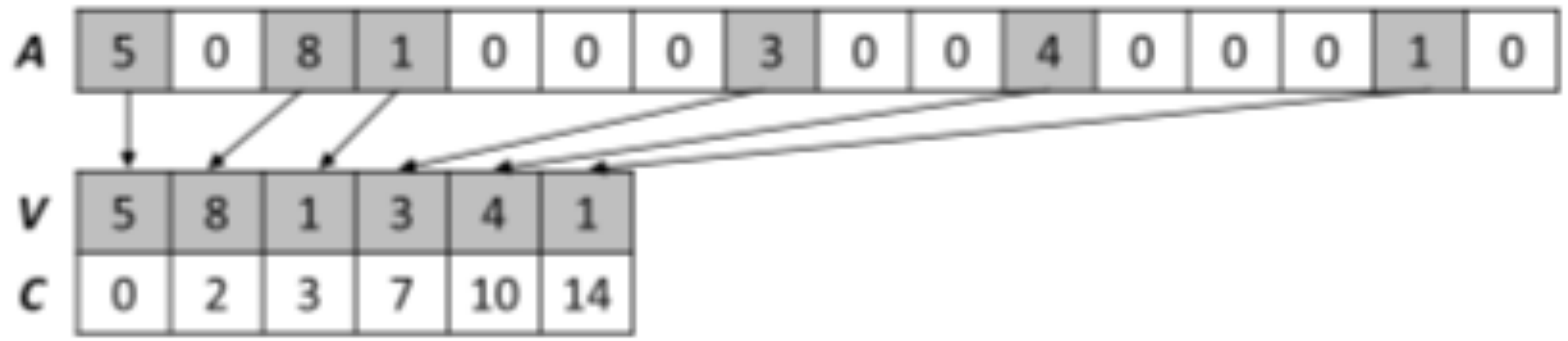


FIGURE 2.4

Example of compacting a sparse array A of size 16 into two smaller arrays: V (values) and C (coordinates).

$$\boxed{C(n) \in \mathcal{O}(n)}$$

Sparse Array Compaction

Cost-Effective Parallel Version

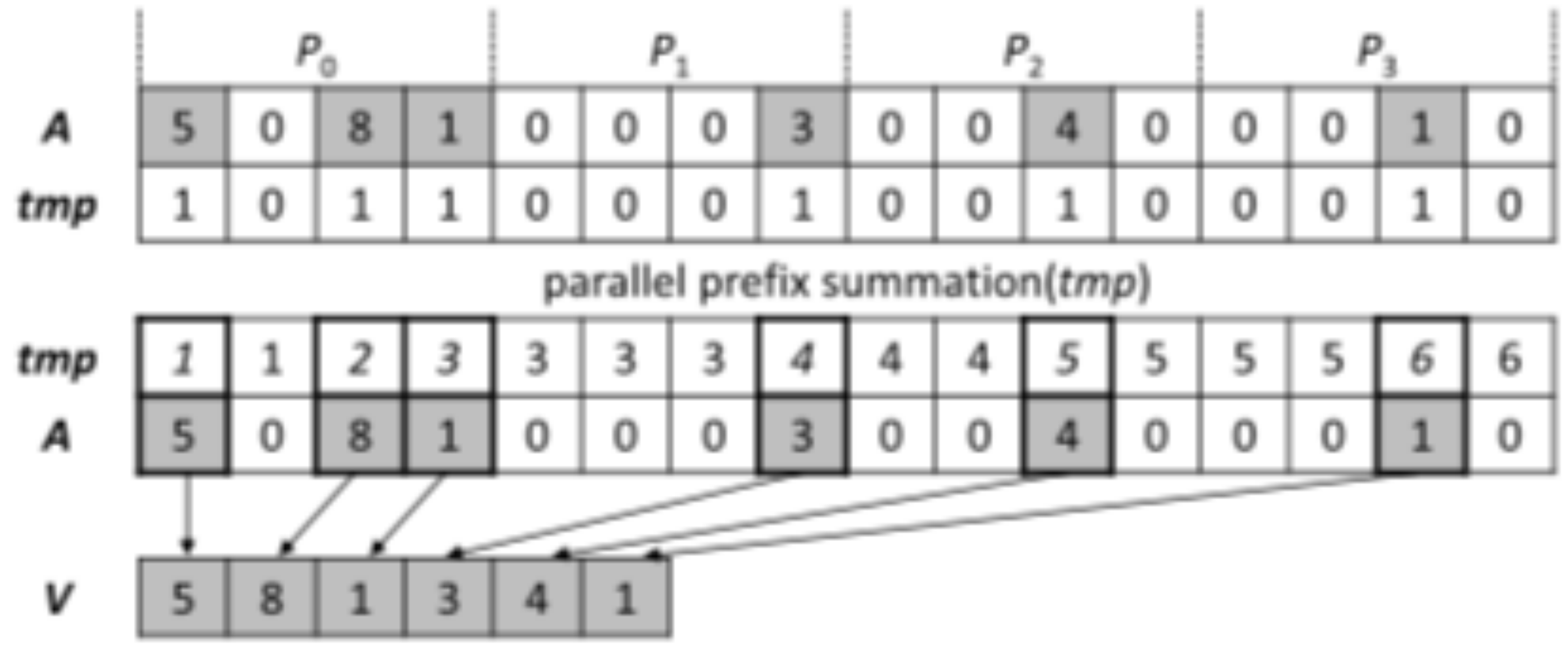


FIGURE 2.5

Example of compacting the values of a sparse array A of size 16 into the array V on a PRAM with four processors.

$$C(n) \in \mathcal{O}\left(\log(n)\right) \times \frac{n}{\log(n)} = \mathcal{O}(n)$$

Network Topologies

Metrics

Distributed Memory Parallel Networks

- Degree
  The maximum number of neighbors of any node.
- Diameter
  The maximum length of all shortest paths between any two nodes.
- Bisection Width
  The minimum number of connections to be removed so as to partition the network into two *equal halves* (where, if the total number of nodes is odd, one "half" may contain one more node than the other "half").

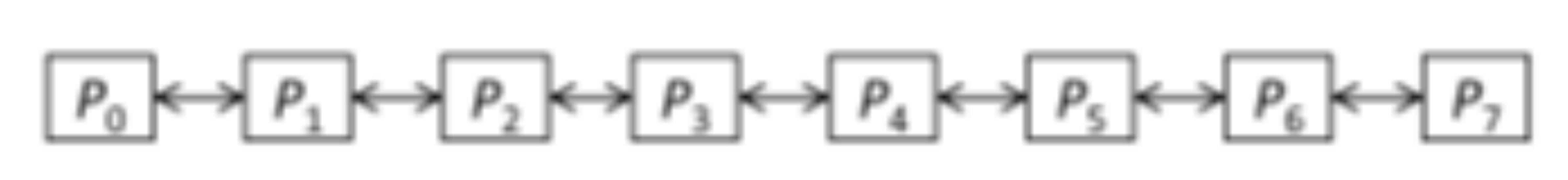
Ideal Metrics

Goals of Optimizing a Network Topology

- Constant degree
- Low diameter
- High bisection width.

Linear Array

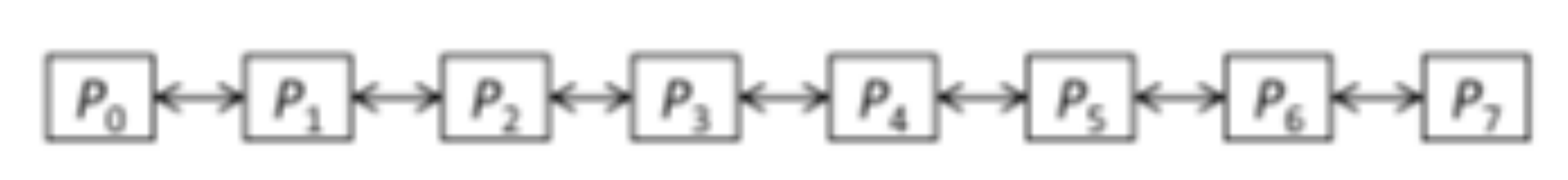
Array L of n Nodes



- Degree — $\deg(L_n)$?
- Diameter — $\mathrm{diam}(L_n)$?
- Bisection Width — $\mathrm{bw}(L_n)$?

Linear Array

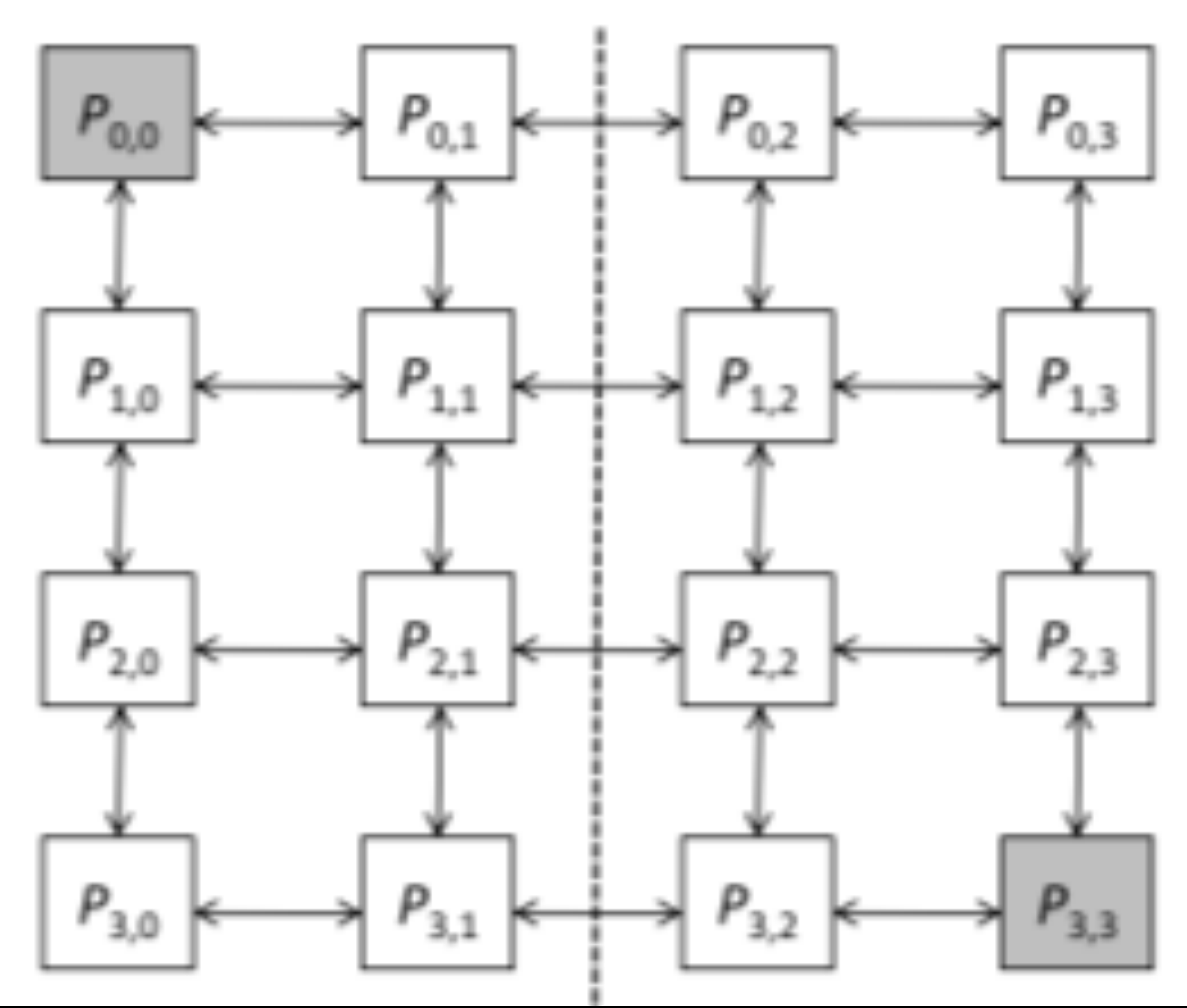
Array L of n Nodes



- Degree — $\deg(L_n) = 2$
- Diameter — $\mathrm{diam}(L_n) = n - 1$
- Bisection Width — $\mathrm{bw}(L_n) = 1$

2-Dimensional Mesh

$M_{x,y}$



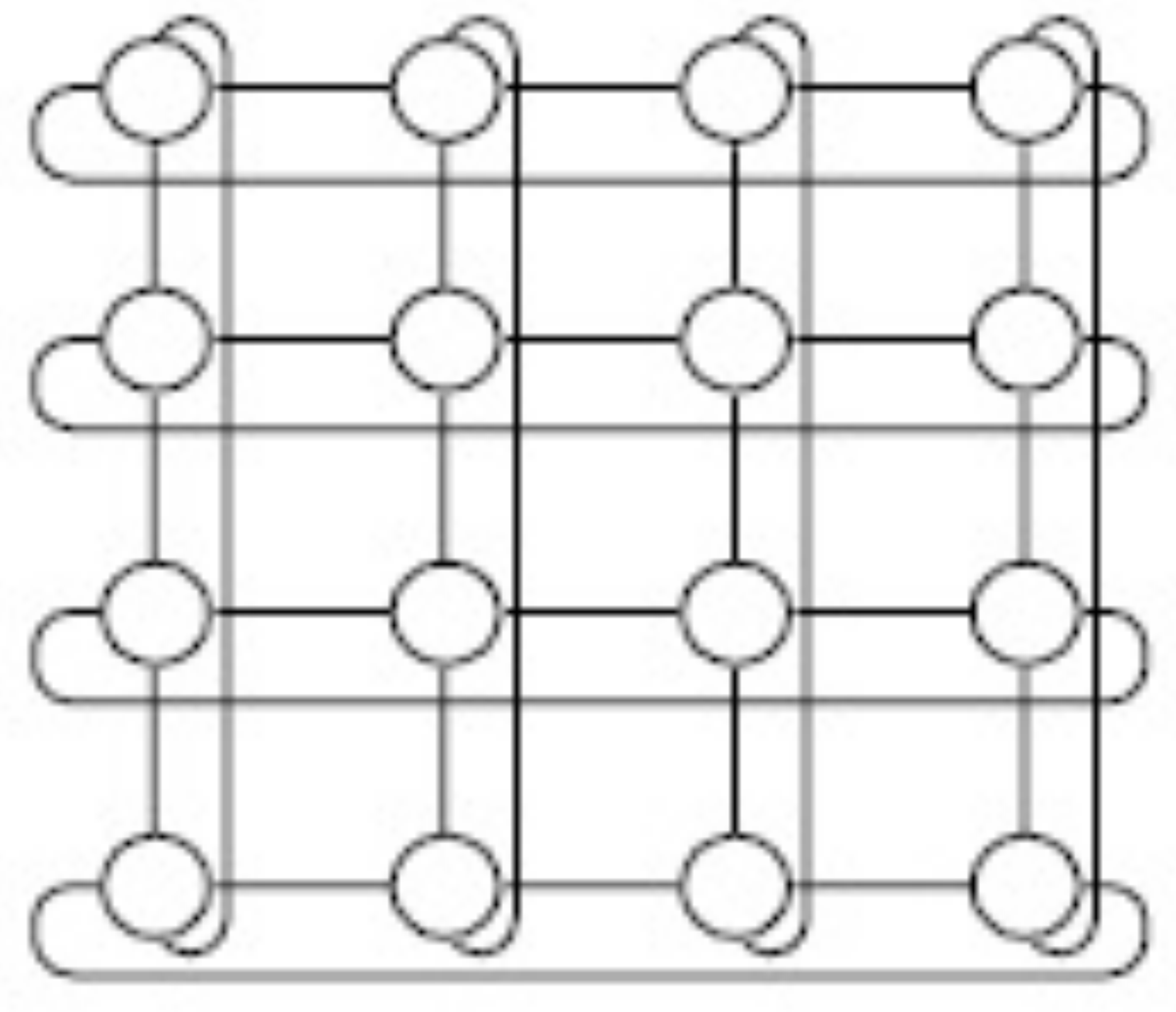
2-Dimensional Mesh

$M_{x,y}$

- $\text{deg}(M_{x,y} = 4$
- $\text{diam}(M_{x,y}) = (x-1) + (y-1) = x + y - 2$
- $\text{bw}(M_{x,y}) = \min(x, y)$

2-Dimensional Torus

$T_{k,k}$ — Square for Simplicity



2D torus

2-Dimensional Torus

- $\deg(T_{k,k}) = 4$
- $\mathrm{diam}(T_{k,k}) = k$
- $\mathrm{bw}(T_{k,k}) = 2k$

3-Dimensional Mesh

$M_{k,k,k}$

- $\mathrm{deg}(M_{k,k,k} = 6$
- $\mathrm{diam}(M_{k,k,k} = 3(k - 1)$
- $\mathrm{bw}(M_{k,k,k}) = k^2$

3-Dimensional Torus

$T_{k,k,k}$

- $\mathrm{deg}(T_{k,k,k} = 6$
- $\mathrm{diam}(M_{k,k,k} = \dfrac{3k}{2}$
- $\mathrm{bw}(M_{k,k,k}) = \dfrac{2k^2}{2} = k^2$
- https://web.ece.ucsb.edu/~parhami/pubs_folder/parh04-mcm-four-torus-nets.pdf

Binary Tree

BT_d of Depth d

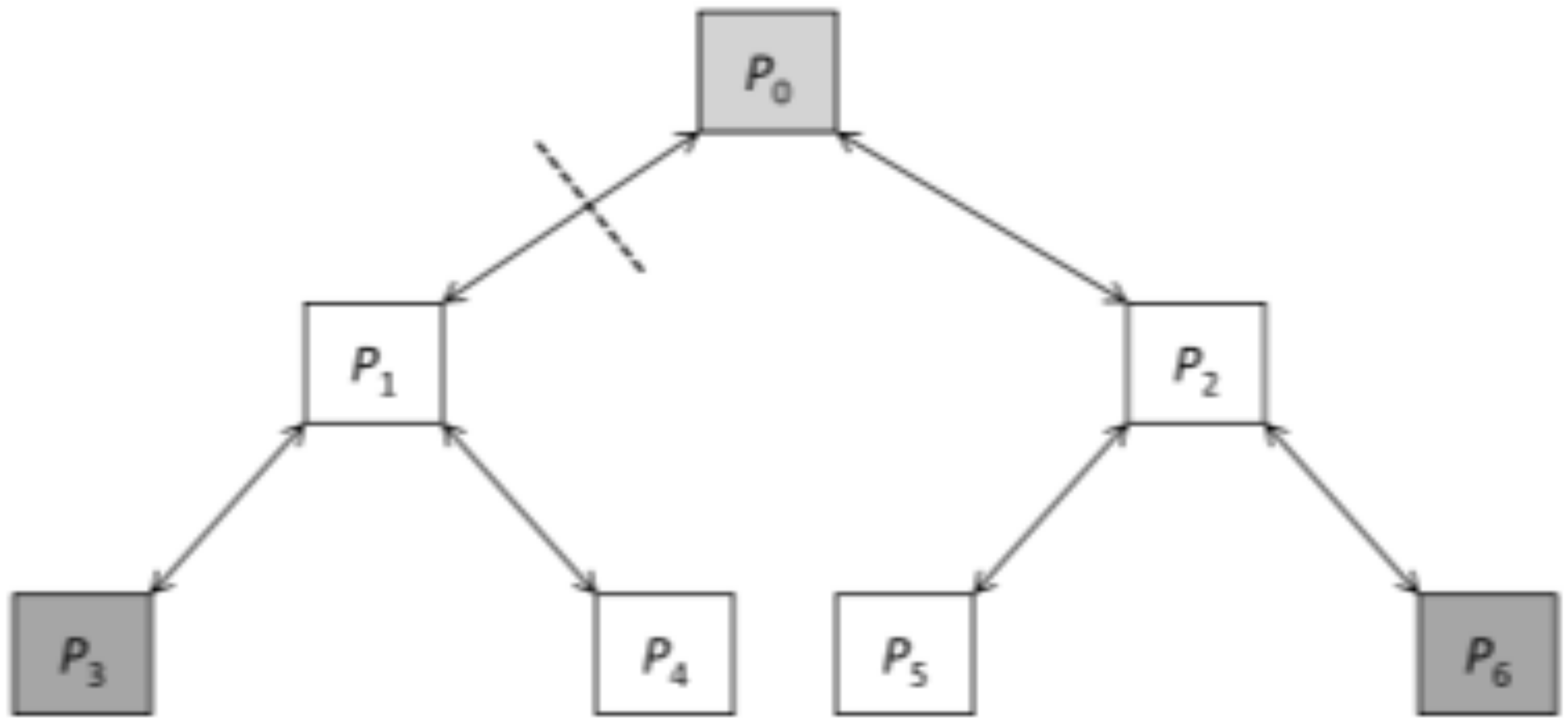


FIGURE 2.8

A binary tree of depth 3 (BT_3). Each node has at most three neighbors; i.e. $\mathbf{deg}(BT_3) = 3$. The longest distance is for example between P_3 and P_6, leading to a diameter of 4. Removing a single link adjacent to the root disconnects the tree into two (almost) equal-sized halves; i.e. $\mathbf{bw}(BT_3) = 1$.

Binary Tree

BT_d of Depth d

- $\deg(BT_d) = 3$
- $\operatorname{diam}(BT_d) = d \in \mathcal{O}\left(\log(n)\right)$ for a binary tree with n nodes ($n = 2^d - 1$)
- $\operatorname{bw}(BT_d) = 1$

Hypercube of Dimension d and size $n = 2^d$

Q_d

- Each node is labeled with a distinct bit string — e.g., for Q_4, 0000, 0001, 0010, 0011, etc.
- Nodes are connected if and only if their bit strings differ in one bit — e.g., 0100 is connected to 0000, 0101, 0110, and 1100, but not to 1110, etc.
- $\deg(Q_d) = d = \log_2(n)$.
- $\mathrm{diam}(Q_d) = d$ — the longest path occurs when every bit differs.
- $\mathrm{bw}(Q_d) = 2^{d-1} = n/2$, where $n = 2^d$ is the number of nodes, because a minimal bisection removes all connections between all nodes whose addresses start with 0 and all nodes whose addresses start with 1.

Hypercube of Dimension d and size $n = 2^d$

Q_d

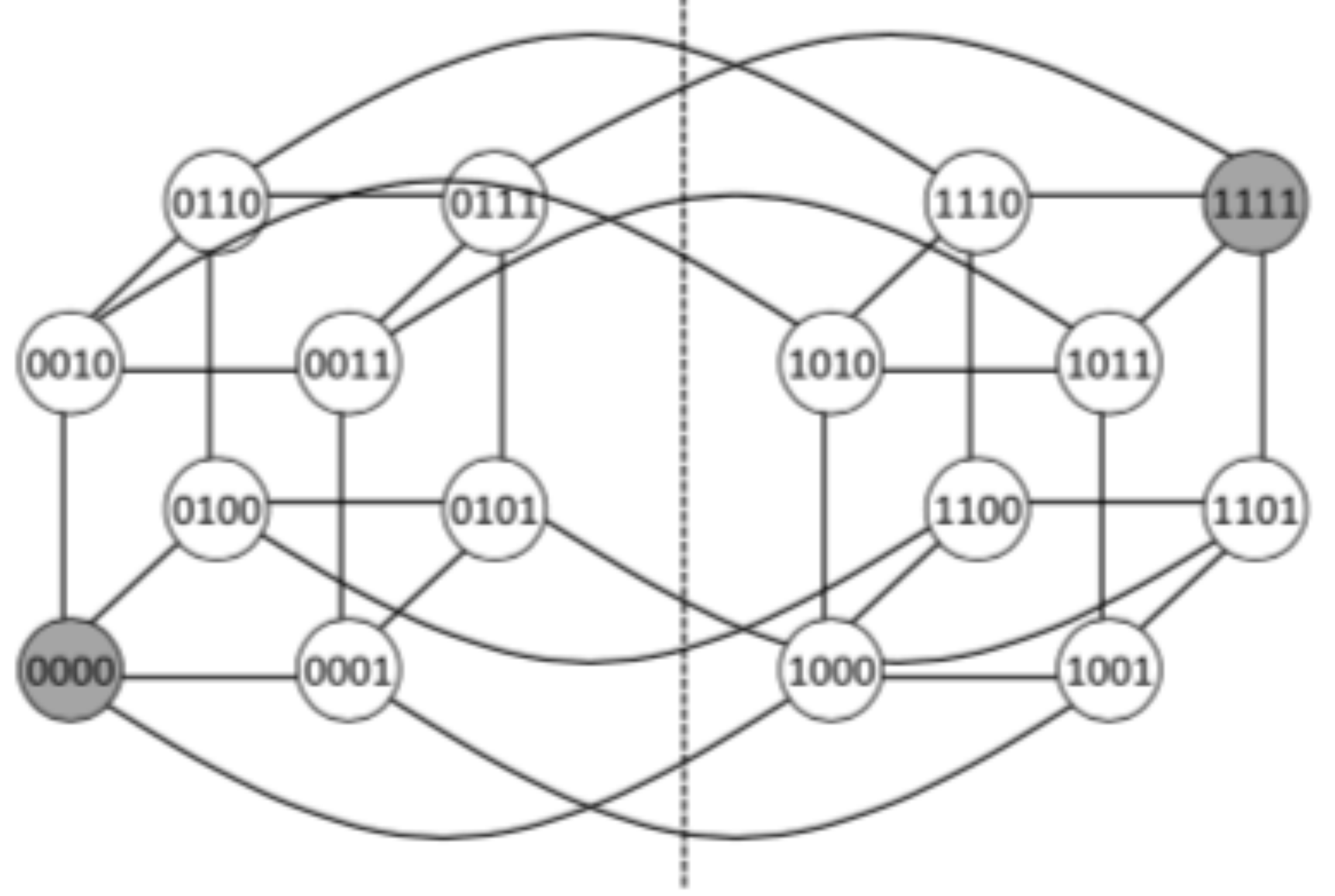


FIGURE 2.9

A 4-dimensional hypercube (Q_4). Each node has exactly four neighbors; i.e. $\mathbf{deg}(Q_4) = 4$. The longest distance is, for example, between the node labeled 0000 and the one labeled 1111, leading to a diameter of 4. Removing all links between the nodes starting with label 0 and all nodes starting with 1 disconnects H_4 into two equal-sized halves; i.e. $\mathbf{bw}(Q_4) = 8$.

Summary

Network Topology Metrics and Trade-Offs

Table 2.1 Degree, diameter, and bisection-width of the discussed interconnection network topologies in terms of the number of nodes (n) using asymptotic notation.

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

Laws of Parallel Processing

Amdahl's Law
Definitions

- Every parallel algorithm has a portion that must be run *serially* (sequentially).
- Let
  - T_{ser} be the time required for a *single processor* to run the serial portion
  - T_{par} be the time required for a *single processor* to run the portion that *can be parallelized*
- $T(1) = T_{\text{ser}} + T_{\text{par}}$

Amdahl's Law
Upper Bound on Speedup

- Assume ideal (linear) speedup
  - $T(2) = \dfrac{T(1)}{2}, T(4) = \dfrac{T(1)}{4}, T(p) = \dfrac{T(1)}{p}$
  - $S(2) = 2, S(4) = 4, S(p) = p$
- Then, the ideal parallel execution *time* has the lower bound
  $T(p) \geq T_{\text{ser}} + \dfrac{T_{\text{par}}}{p}$
- The ideal parallel execution *speedup* has the upper bound
  $S(p) = \dfrac{T(1)}{T(p)} \leq \dfrac{T_{\text{ser}} + T_{\text{par}}}{T_{\text{ser}} + \frac{T_{\text{par}}}{p}}$

Amdahl's Law
Fractional Formulation

- Let f be the fraction of the total running time taken up by the serial portion
  - $0 < f < 1$
  - $T_{\text{ser}} = T(1) \times f$
  - $T_{\text{par}} = T(1) \times (1 - f)$
- Therefore,

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

$$= \frac{f \cdot T(1) + (1-f) \cdot T(1)}{f \cdot T(1) + \frac{(1-f) \cdot T(1)}{p}} = \frac{f + (1-f)}{f + \frac{(1-f)}{p}} = \frac{1}{f + \frac{(1-f)}{p}} \quad \text{(Amdahl's Law)}$$

Amdahl's Law

Examples

- Suppose a program contains a loop that takes up 75% of execution time and that we can parallelize.
- Suppose we plan to use 8 processors.

$$S(8) \leq \frac{1}{0.25 + \frac{0.75}{8}} = 2.9$$

- Suppose you can scale up to unlimited processors?

$$S(\infty) \leq \lim_{p \to \infty} \frac{1}{0.25 + \frac{0.75}{p}} = 4$$

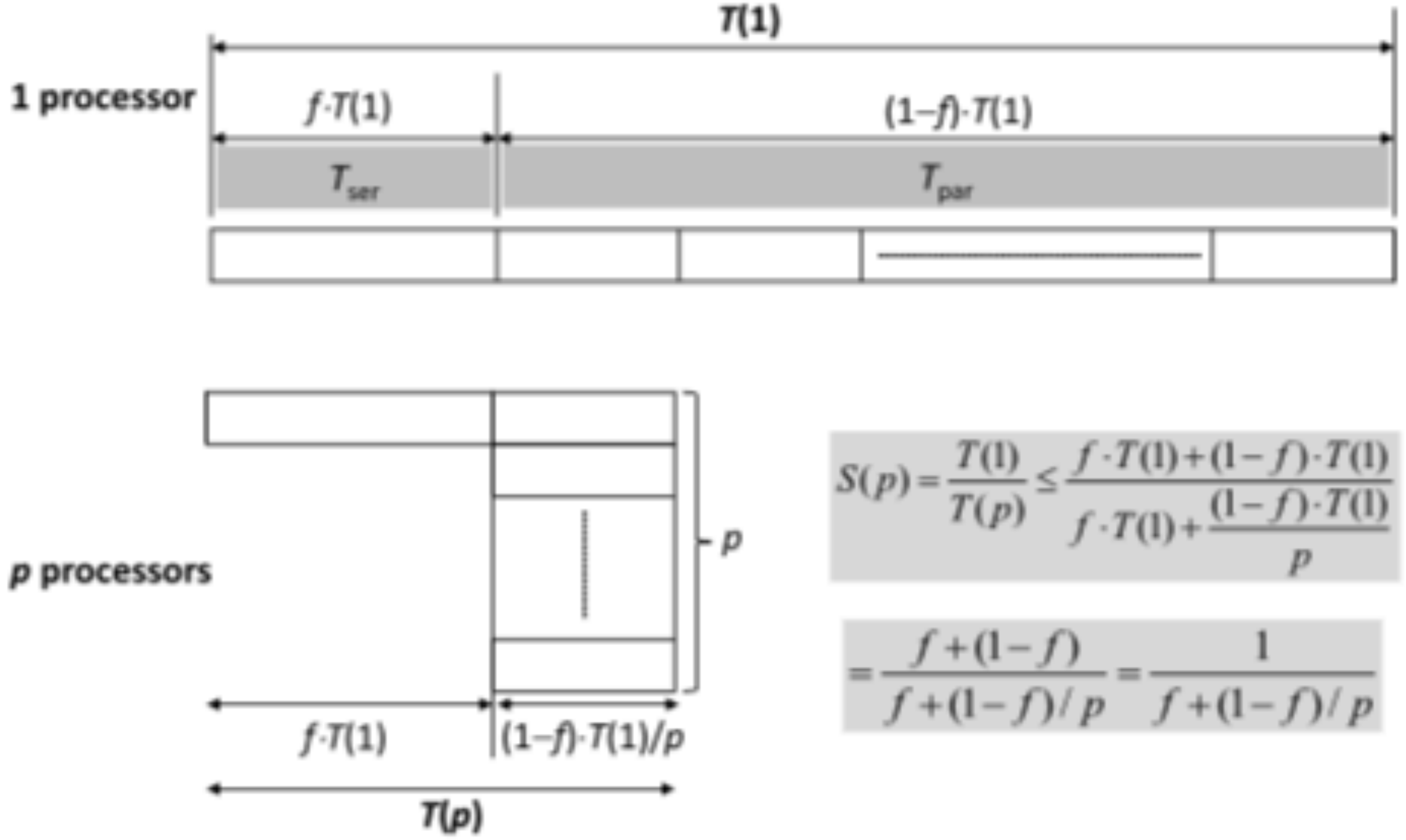


FIGURE 2.10

Illustration of Amdahl's law for the establishing an upper bound for the speedup with constant problem size.

Gustafson's Law

Context

- Amdahl's law assumes that the portion f of the program's running time that must be run serially is fixed.
- This corresponds to a strongly scalable system: the input size remains constant while the number of processors increases.
- In many algorithms, the portion of the running time f *changes* (shrinks) as the input size increases, providing *weak scalability*.
- In the Distributed Sum algorithm, f (communication) *increased* with input size n and number of processors p, but only as $log_2(p)$.
- Can you think of an algorithm where f shrinks as n increases?

Gustafson's Law

Derivation from a Generalization

- Let α be the complexity of the serial portion as a function of n.
- Let β be the complexity of the parallelizable portion as a function of n.
- $$S_{\alpha\beta}(p) = \frac{T_{\alpha\beta}(1)}{T_{\alpha\beta}(p)} \leq \frac{\alpha \cdot f \cdot T(1) + \beta \cdot (1-f) \cdot T(1)}{\alpha \cdot f \cdot T(1) + \frac{\beta \cdot (1-f) \cdot T(1)}{p}} = \frac{\alpha \cdot f + \beta \cdot (1-f)}{\alpha \cdot f + \frac{\beta \cdot (1-f)}{p}}$$

Gustafson's Law

Derivation

- Now let $\gamma = \dfrac{\beta}{\alpha}$ be the ratio of the complexity function of the parallelizable portion to the complexity function of the serial portion.
- $$S_\gamma(p) \leq \frac{f + \gamma(1-f)}{f + \dfrac{\gamma(1-f)}{p}}$$
- Note:

$$S_{\alpha\beta} \leq \frac{\alpha \cdot f + \beta \cdot (1-f)}{\alpha \cdot f + \dfrac{\beta \cdot (1-f)}{p}} = \frac{\alpha/\alpha \cdot f + \beta/\alpha \cdot (1-f)}{\alpha/\alpha \cdot f + \dfrac{\beta/\alpha \cdot (1-f)}{p}} = \frac{f + \gamma \cdot (1-f)}{f + \dfrac{\gamma \cdot (1-f)}{p}}$$

Gustafson's Law

Derivation from a Special Case

- Special cases
  - $\gamma = 1$
    - The ratio never changes.
    - You add processors without changing the input size.
    - **Amdahl's law**.
  - $\gamma = p$
    - The parallel share of computation increases in proportion to the number of processors.
    - This happens automatically if the ratio of complexities increases as p increases.
    - **Gustafson's law**— $S(p) \leq f + f \cdot (1 - f) = \boxed{p + f \cdot (1 - p)}$.
  - This is the *ideal speedup* for a system that scales up *weakly.*

Gustafson's Law

Example

- Suppose we have a program with a portion that can be linearly parallelized (with ideal speedup) taking up 85% of the total.
- How much speedup would we have if the ratio of parallel to serial computation remains fixed for any input size n (by Amdahl's law), with $p = 50$ processors?

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

- What if the program is such that the ratio of computation time between the paralleizable and the serial portions increases as p increases?

$$S_{\gamma=p}(p = 50) \leq p + f \cdot (1 - p) = 50 + 0.15 \cdot (-49) = 42.65$$

Gustafson’s Law

Original Observation

As a first approximation, we have found that it is the *parallel or vector* part of a program that scales with the problem size. Times for vector startup, program loading, serial bottlenecks and I/O that make up the *s* component of the run do not grow with problem size. When we double the number of degrees of freedom in a physical simulation, we double the number of processors. But this means that, as a first approximation, the amount of work that can be done in parallel *varies linearly with the number of processors*. For the three applications mentioned above, we found that the parallel portion scaled by factors of 1023.9969, 1023.9965, and 1023.9965. If we use s' and p' to represent serial and parallel time spent on the *parallel* system, then a serial processor would require time $s' + p' \times N$ to perform the task. This reasoning gives an alternative to Amdahl's law suggested by E. Barsis at Sandia:

$$\begin{aligned} \textit{Scaled speedup} &= (s' + p' \times N) / (s' + p') \\ &= s' + p' \times N \\ &= N + (1 - N) \times s' \end{aligned}$$

http://www.johngustafson.net/pubs/pub13/amdahl.htm

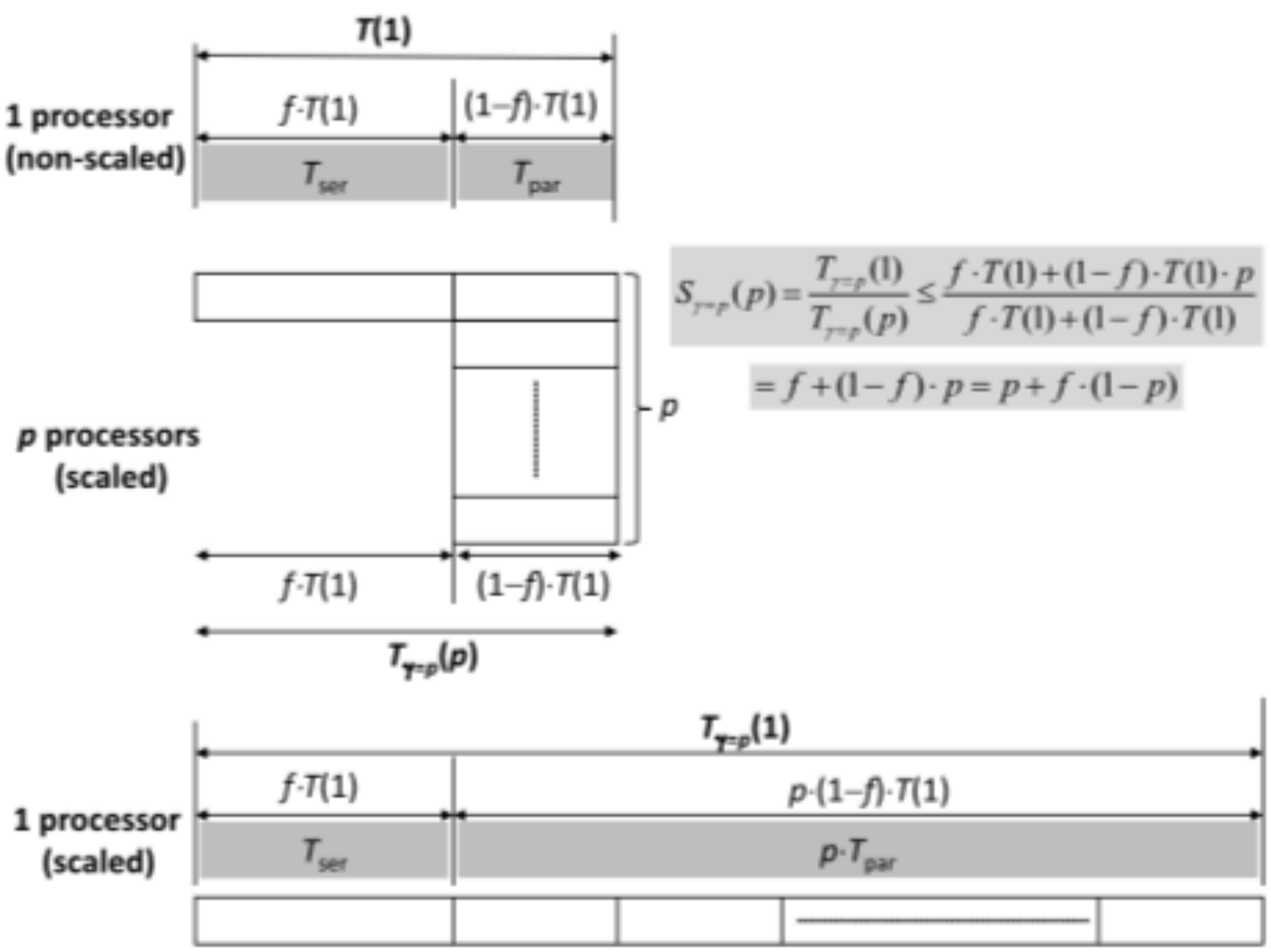


FIGURE 2.11

Illustration of Gustafson's law for establishing an upper bound for the scaled speedup.

Designing a Parallel Algorithm

Considerations

How Will We Handle These?

- Partitioning
- Communication
- Synchronization
- Load Balancing

Ian Foster's Method

PCAM

- Partitioning
  How can we break the problem up into elementary sub-tasks among independent processors?
- Communication
  What data does each processor need and how can we provide it with those data?
- Agglomeration
  Should we combine the sub-tasks identified in "Partitioning" for efficiency?
- Mapping
  How do we map tasks to processors so as to
  - Minimize communication?
  - Maximize parallelization?
  - Balance the workload among processors?