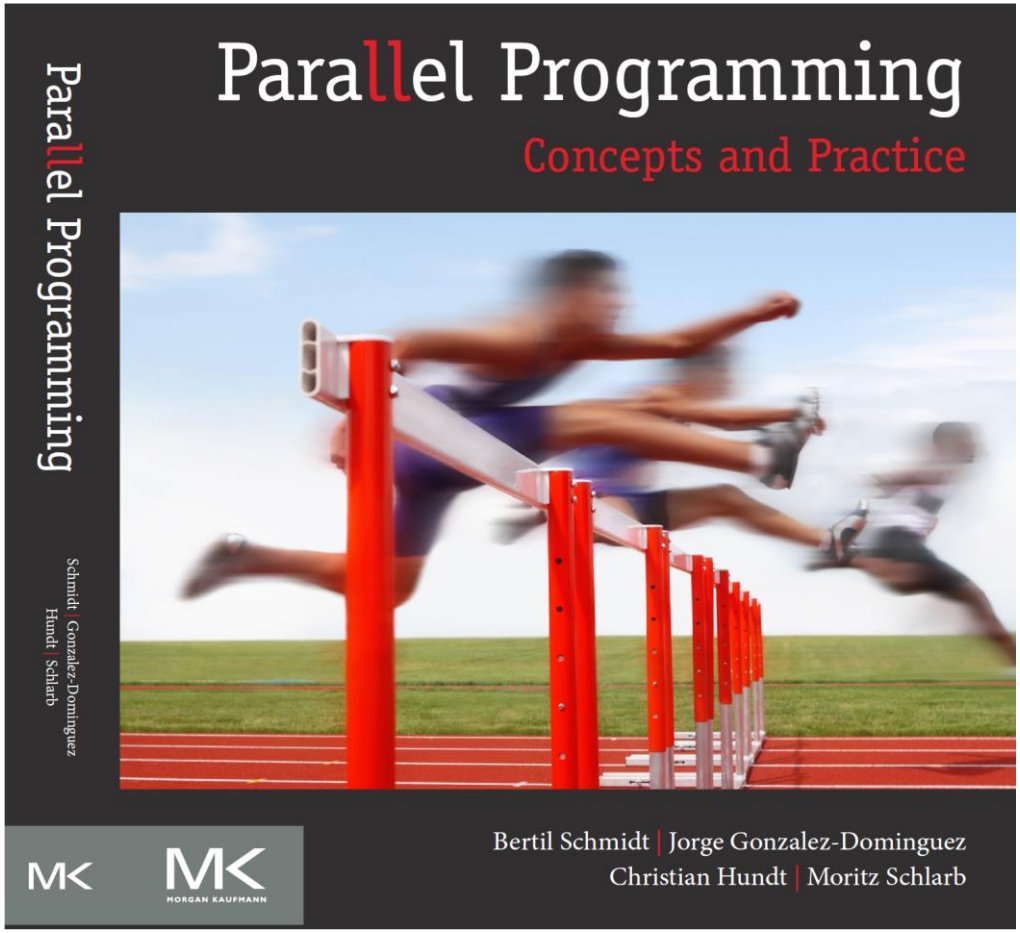
Chapter 01: Introduction



What we'll be doing

- Modern CPUs contain many cores
- An even higher degree of parallelism is available multiple CPUs, clusters, and supercomputers.
- Thus, the ability to program these types of systems efficiently and effectively is an essential skill
- You will learn how to write programs that are explicitly parallel
- Using the C++ language for
  - Message-Passing Interface (MPI)
  - Multithreading
  - OpenMP
  - Vectorization

Learning Outcomes Today

- Understand and analyze a parallel algorithm for adding numbers
- Learn about the terms **Speedup**, **Efficiency**, **Scalability**, and **Compute-to-Computation Ratio**
- Know the difference between *shared-memory* and *distributed-memory* architectures and some corresponding parallel programming languages
- Understand important considerations for parallel program design:
  - Partitioning, Communication, Synchronization, Load Balancing
- Learn about important HPC Trends and Rankings
  - Top 500, Green 500, Graph 500

Motivational Example and its Analysis

- **Speedup.** You have designed a parallel algorithm or written a parallel code.
- Now you want to know how much faster it is than your sequential approach.
- The speedup (*S*) is usually measured or calculated for almost every parallel code or algorithm and is simply defined as the quotient of the time taken using a single processor (*T (1)*) over the time measured using *p* processors (*T (p)*).

$$S = \frac{T(1)}{T(p)}$$

Motivational Example and its Analysis

- The Efficiency E measures exactly that by dividing S by p.

$$E = \frac{S}{p} = \frac{T(1)}{T(p) \times p}$$

- The cost C is similar but relates the runtime T (p) to the number of utilized processors (or cores) by multiplying T (p) and p.

$$C = T(p) \times p$$

Motivational Example and its Analysis

- **Scalability**. Often we do not only want to measure the efficiency for one particular number of processors or cores but for a varying number; e.g. P = 1, 2, 4, 8, 16, 32, 64, 128, etc.
- This is called scalability analysis and indicates the behavior of a parallel program when the number of processors increases.
- Besides varying the number of processors, the input data size is another parameter that you might want to vary when executing your code.
- Thus, there are two types of scalability: **strong** scalability and **weak** scalability.
- In the case of **strong** scalability we measure efficiencies for a varying number of processors and keep the input data size fixed.
- In contrast, **weak** scalability shows the behavior of our parallel code for varying both the number of processors and the input data size; i.e. when doubling the number of processors we also double the input data size.

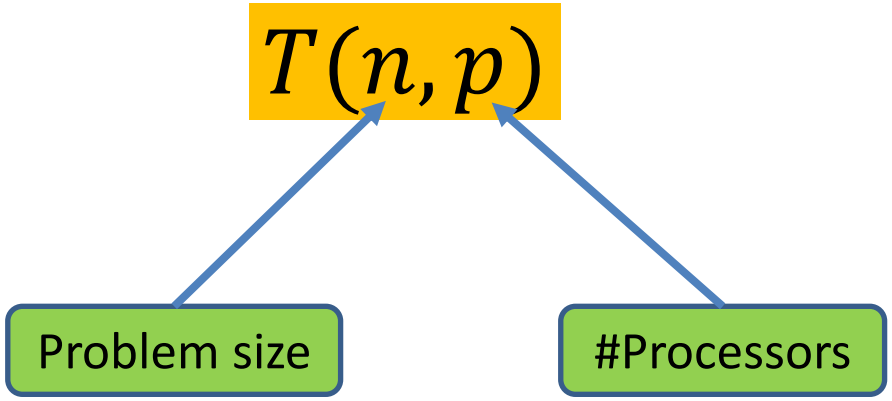
Motivational Example and its Analysis

- **Computation-to-communication ratio**. This is an important metric influencing the achievable scalability of a parallel implementation.
- It can be defined as the time spent calculating divided by the time spent communicating messages between processors.
- A higher ratio often leads to improved speedups and efficiencies.

Motivational Example and its Analysis

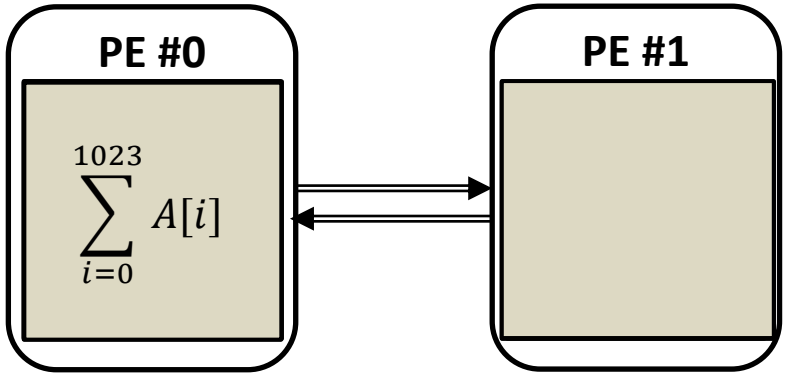
- **Input:** Array *A* of *n* numbers
- **Output:** $\sum_{i=0}^{n-1} A[i]$
- **Task:** Parallelize this problem efficiently using an array of ***processing elements*** (**PEs**)
- **Assumptions:**
  1. **Computation:** Each PE can add two numbers stored in its local memory in **1 sec**
  2. **Communication:** A PE can send data from its local memory to the local memory of any other PE in **3 sec** (independent of the size of the data)
  3. **Input and Output:** At the beginning of the program the whole input array *A* is stored in PE #0. At the end the result should be gathered in PE #0
  4. **Synchronization:** All PEs operate in lock-step manner; i.e. they can either compute, communicate or be idle. Thus, it is not possible to overlap computation and communication on this architecture.

Motivational Example and its Analysis



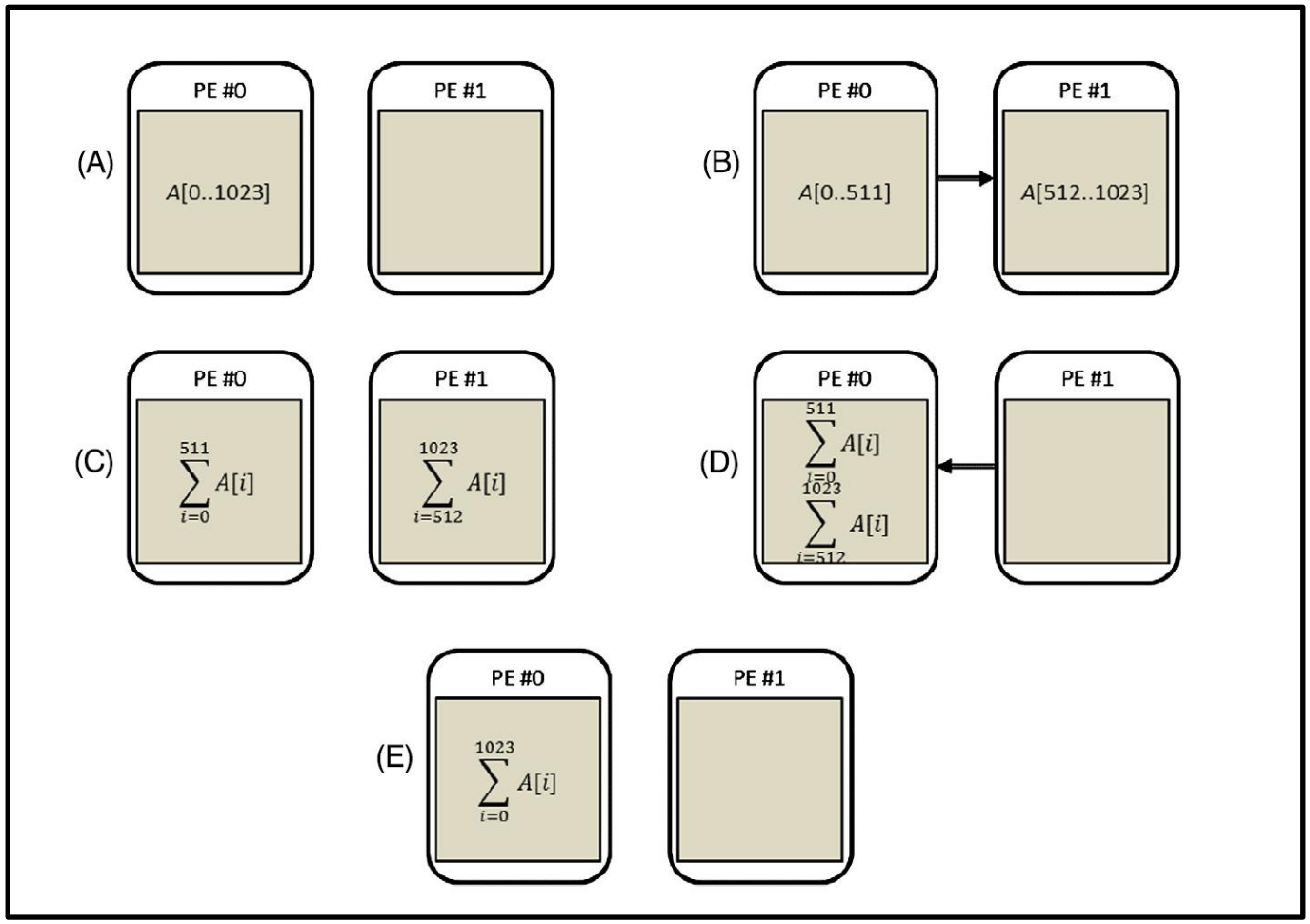
- Establish sequential runtime as a baseline (p = 1):
  - For our example: ***T*(1,*n*) = *n*–1 sec**
- What are we interested in?
  - **Speedup:** How much faster can we get with $p > 1$ processors?
  - **Efficiency:** Is our parallel program efficient?
  - **Scalability:** How does our parallel program behave for varying number of processors (for fixed/varying problem size)?

Motivational Example and its Analysis

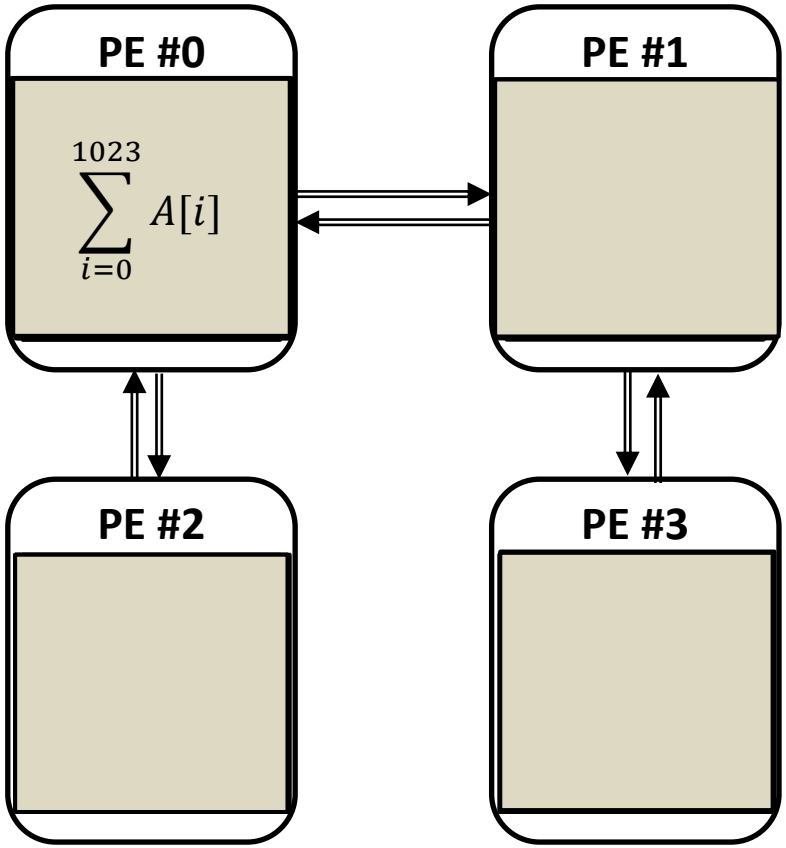


- Establish runtime for 2 PEs (p = 2) and 1024 numbers (n = 1024):
  - $T(2,1024) = 3 + 511 + 3 + 1 =$ **518 sec**
  - **Speedup:** $T(1,1024)/T(2,1024) = 1023/518 = 1.975$
  - **Efficiency:** 1.975/2 = 98.75%

Motivational Example and its Analysis

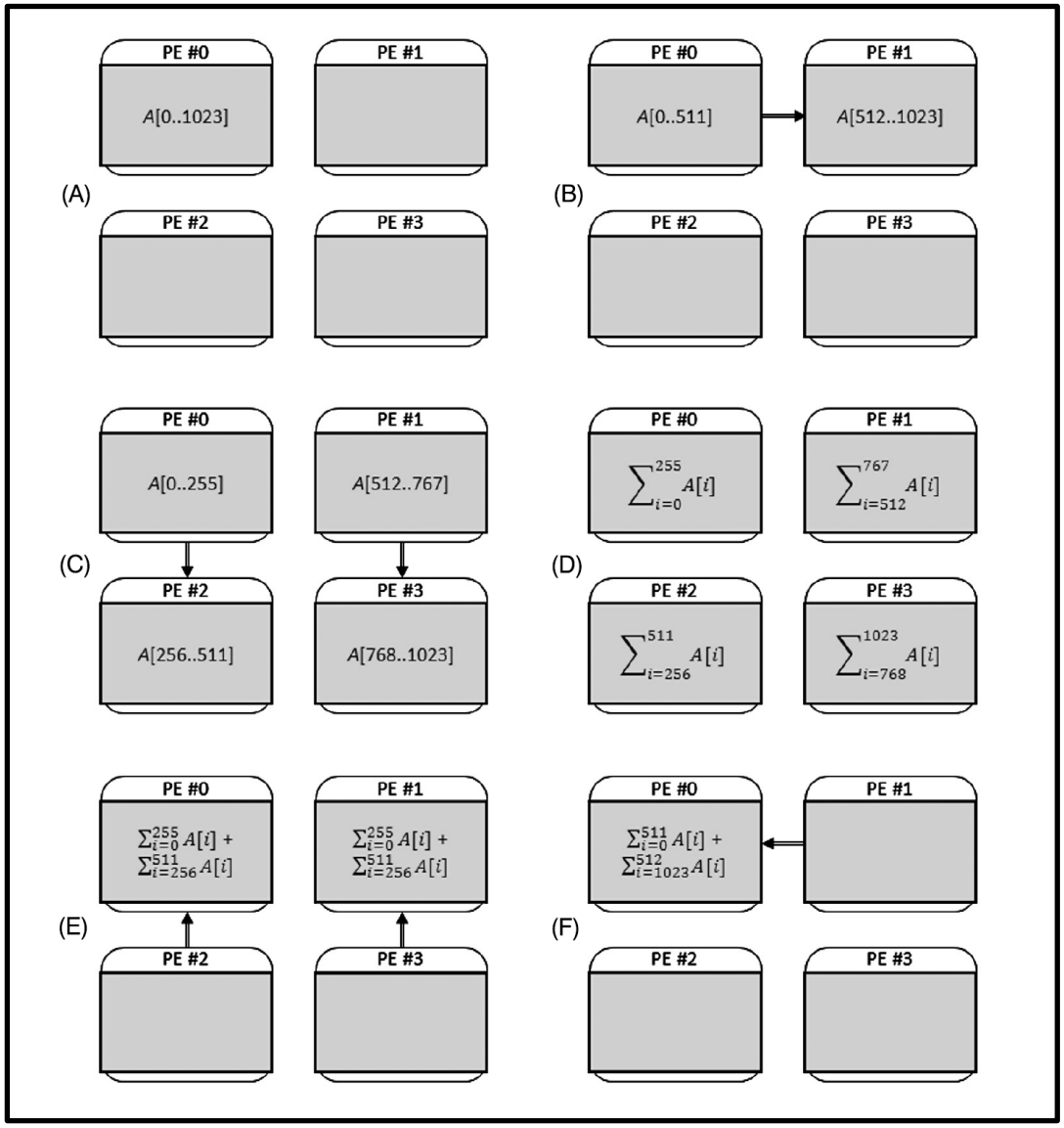


Motivational Example and its Analysis

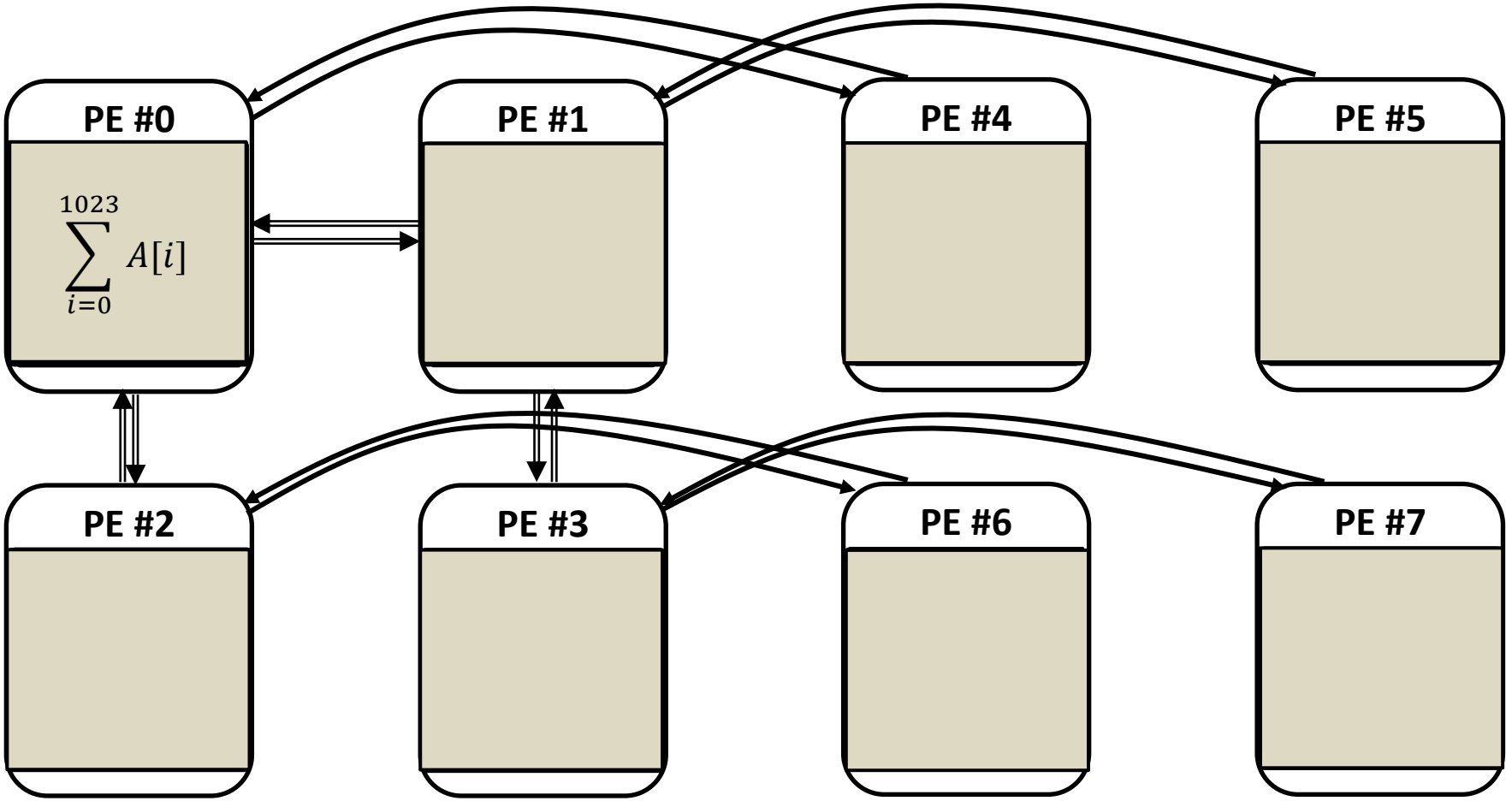


- $T(4,1024) = 3\times2 + 255 + 3\times2 + 2 =$ **269** seconds
- **Speedup:** $T(1,1024)/T(4,1024) = 1023/269 = 3.803$
- **Efficiency:** 3.803/4 = 95.07%

Motivational Example and its Analysis



Motivational Example and its Analysis



- $T(8,1024) = 3\times3 + 127 + 3\times3 + 3 =$ **148** seconds
- **Speedup:** $T(1,1024)/T(8,1024) = 1023/148 = 6.91$
- **Efficiency:** 6.91/8 = 86%

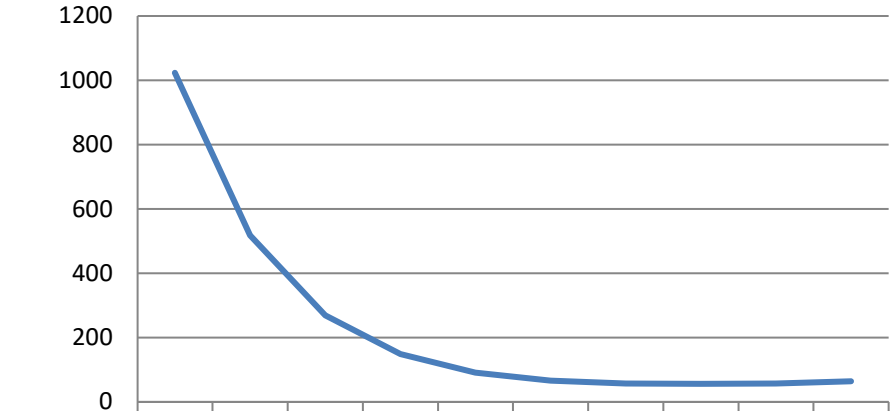
Motivational Example and its Analysis

- Timing analysis using $p = 2^q$ PEs and $n = 2^k$ input numbers:
  - Data distribution: $\mathbf{3 \cdot q}$
  - Computing local sums: $\mathbf{n/p - 1 = 2^{k-q} - 1}$
  - Collection partial results: $\mathbf{3 \cdot q}$
  - Adding partial results: $\mathbf{q}$

$$T(p,n) = T(2^q,2^k) = 3q + 2^{k-q} - 1 + 3q + q = \mathbf{2^{k-q} - 1 + 7q}$$

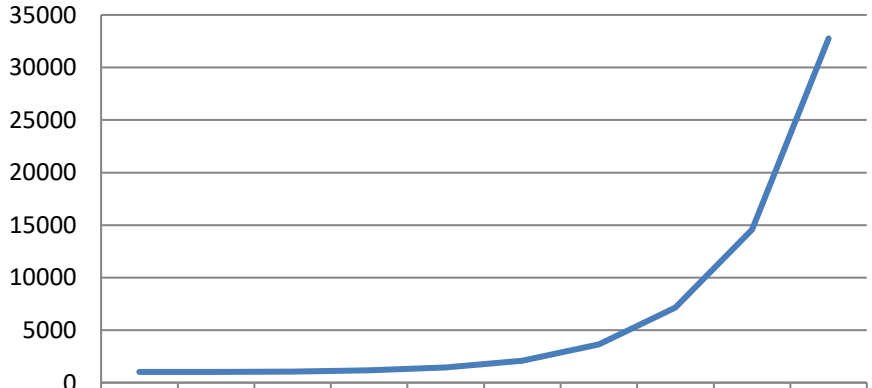
Strong Scalability Analysis for $n = 1024$

Runtime $T(1024,p)$



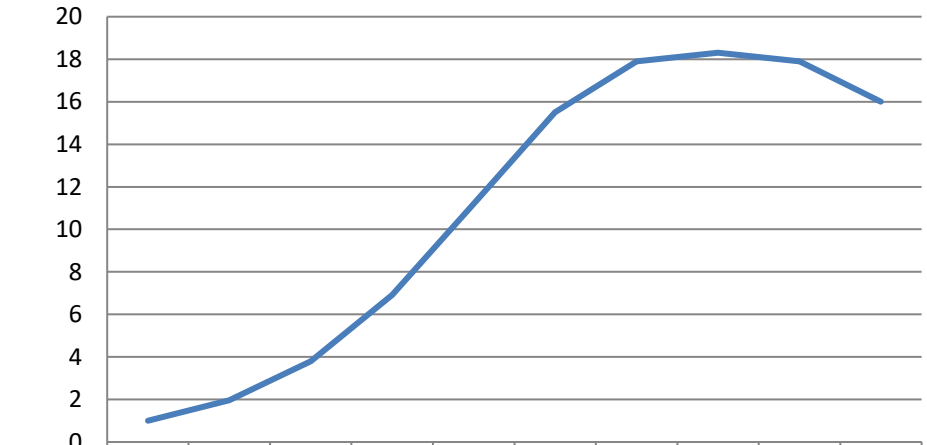
	1	2	4	8	16	32	64	128	256	512
Runtime	1023	518	269	148	91	66	57	56	57	64

Cost = $T(1024,p)$ x p



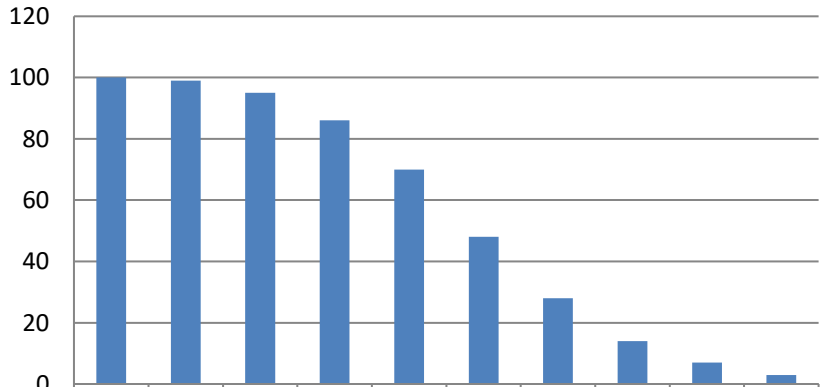
	1	2	4	8	16	32	64	128	256	512
Cost	1023	1036	1076	1184	1456	2112	3648	7168	14592	32768

Speedup = $T(1024,1)/T(1024,p)$



	1	2	4	8	16	32	64	128	256	512
Speedup	1	1,97	3,8	6,91	11,2	15,5	17,9	18,3	17,9	16

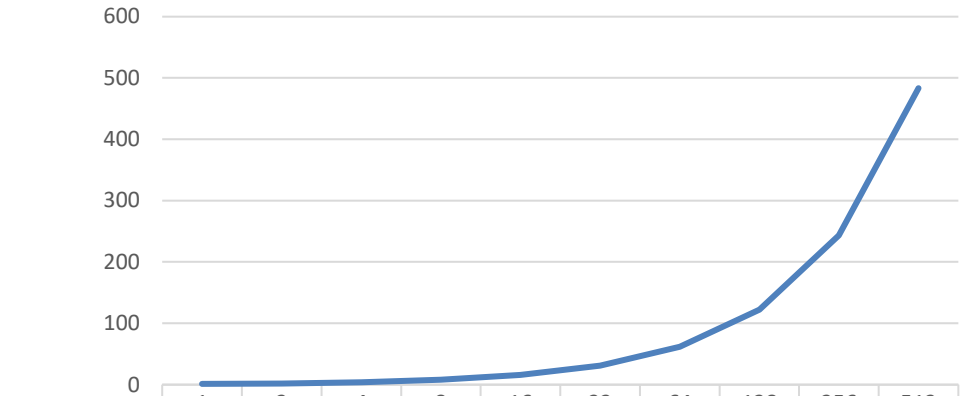
Efficiency (in %) = Speedup/p



	1	2	4	8	16	32	64	128	256	512
Efficiency	100	99	95	86	70	48	28	14	7	3

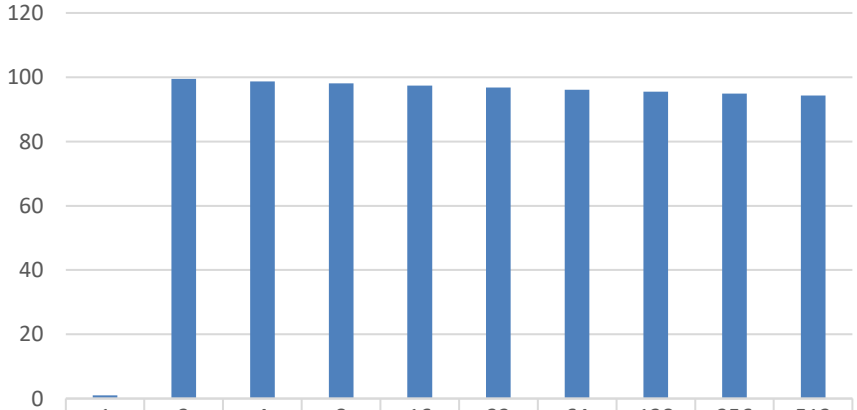
Weak Scalability Analysis for $n = 1024 \times p$

Speedup = $T(1024 \times p, 1)/T(1024 \times p, p)$



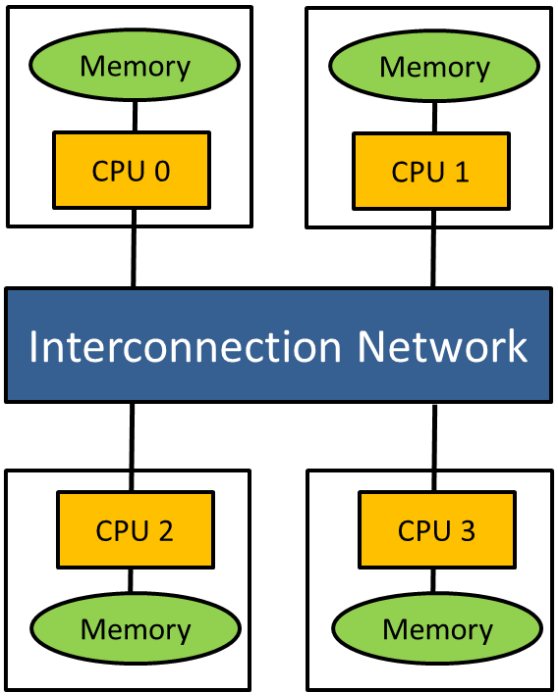
	1	2	4	8	16	32	64	128	256	512
Speedup	1	1,99	3,95	7,85	15,6	31	61,5	122	243	483

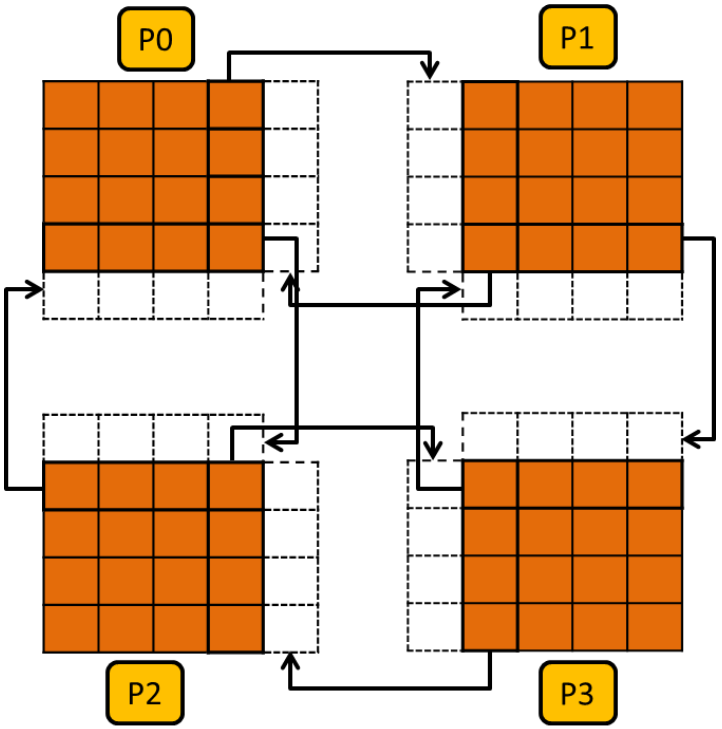
Efficiency (in %) =Speedup/p



	1	2	4	8	16	32	64	128	256	512
Efficiency (in %)	1	99,5	98,7	98,1	97,4	96,8	96,1	95,5	94,9	94,3

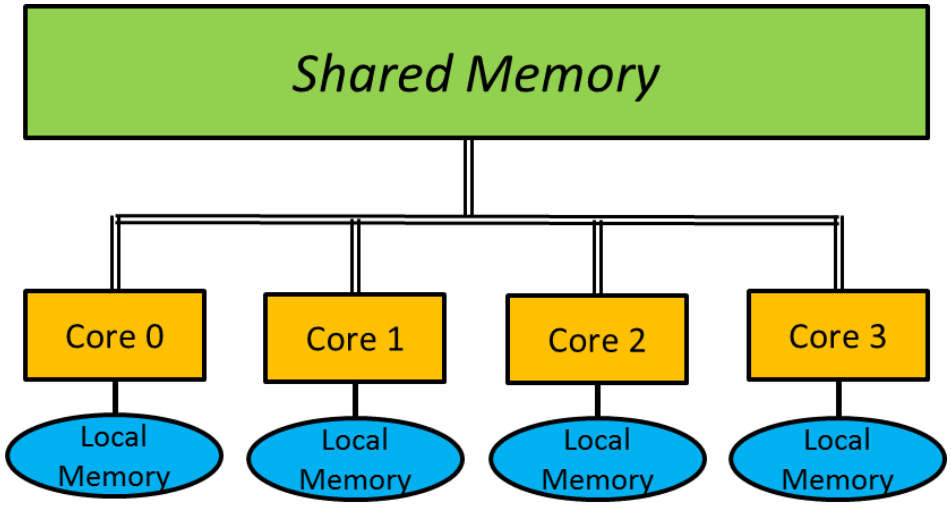
Distributed Memory Systems





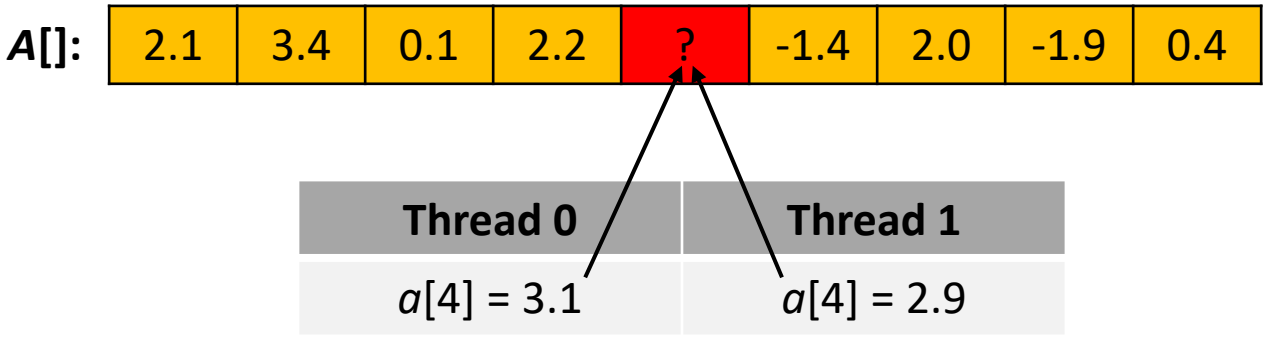
- Each node has its own, private memory. Processors communicate explicitly by sending messages across a network
  - Most popular language: **MPI** (e.g. `MPI_Send`, `MPI_Recv`, `MPI_Bcast`, `MPI_Reduce`)
  - Alternative: PGAS languages (e.g. UPC++)
- **Example:** Compute Clusters
  - Collection of commodity systems such as CPUs connected by an interconnection network (e.g . Infiniband)

Shared Memory Systems



- All cores can access a common memory space through a shared bus or crossbar switch
  - e.g. multi-core CPU-based workstations in which all cores share main memory
- In addition to the shared main memory each core can also contains a smaller local memory (e.g. L1-cache) in order to reduce expensive accesses to main memory (*von-Neumann bottleneck*)
  - Modern multi-core CPU systems support cache coherence
  - *ccNUMA*: cache coherent non-uniform access architectures
- Popular languages: C++11 multithreading, OpenMP, CUDA

Shared Memory Systems



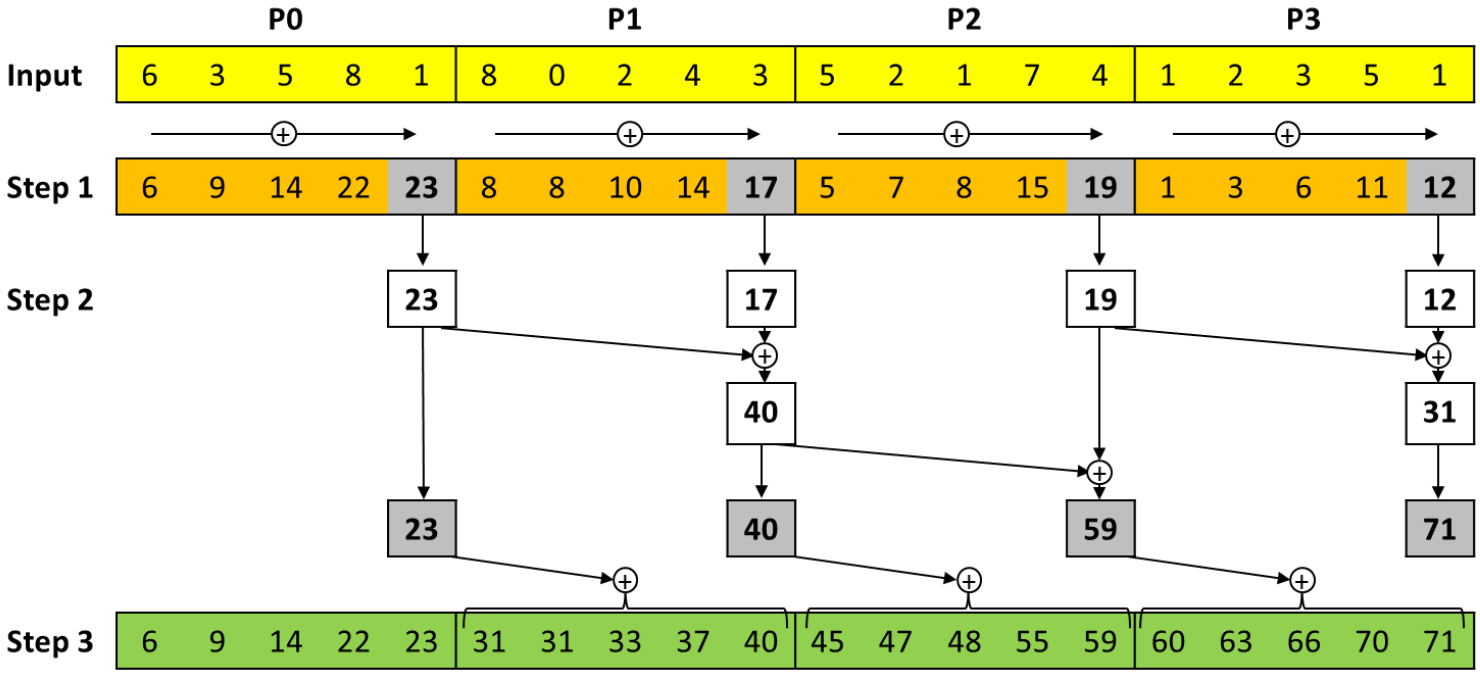
- Parallelism created by starting threads running concurrently on the system
- Exchange of data implemented by threads reading from and writing to shared memory locations.
- **Race condition:** occurs when two threads access a shared variable simultaneously
  - Corresponding programming techniques: *mutexes*, *condition variables*, *atomics*
- Thread creation more lightweight and faster compared to process creation:
  - `CreateProcess()`: 12.76 ms
  - `CreateThread()`: 0.038 ms

Parallel Program Design

- **Partitioning:**
  - Given problem needs to be decomposed into pieces; e.g. data parallelism, task parallelism, model parallelism
- **Communication:**
  - Chosen partitioning scheme determines the amount and types of required communication
- **Synchronization:**
  - In order to cooperate in an appropriate way, threads or processes may need to be synchronized
- **Load Balancing:**
  - Work needs to be equally divided among threads or processes in order to balance the load and minimize idle times
- $\Rightarrow$ Foster‘s Design methodology (see Chapter 2)

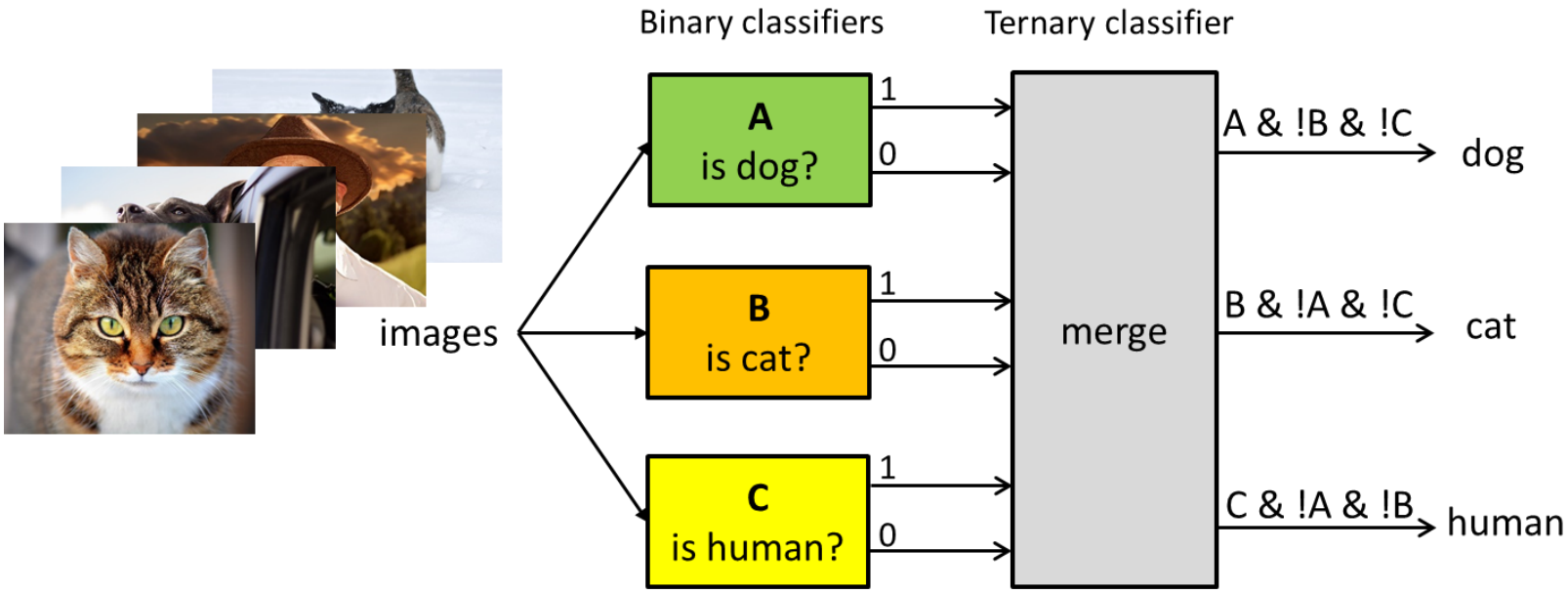
Discovering Parallelism (Prefix Sum)

```
for (i=1; i<n; i++) A[i]=A[i]+A[i-1];
```



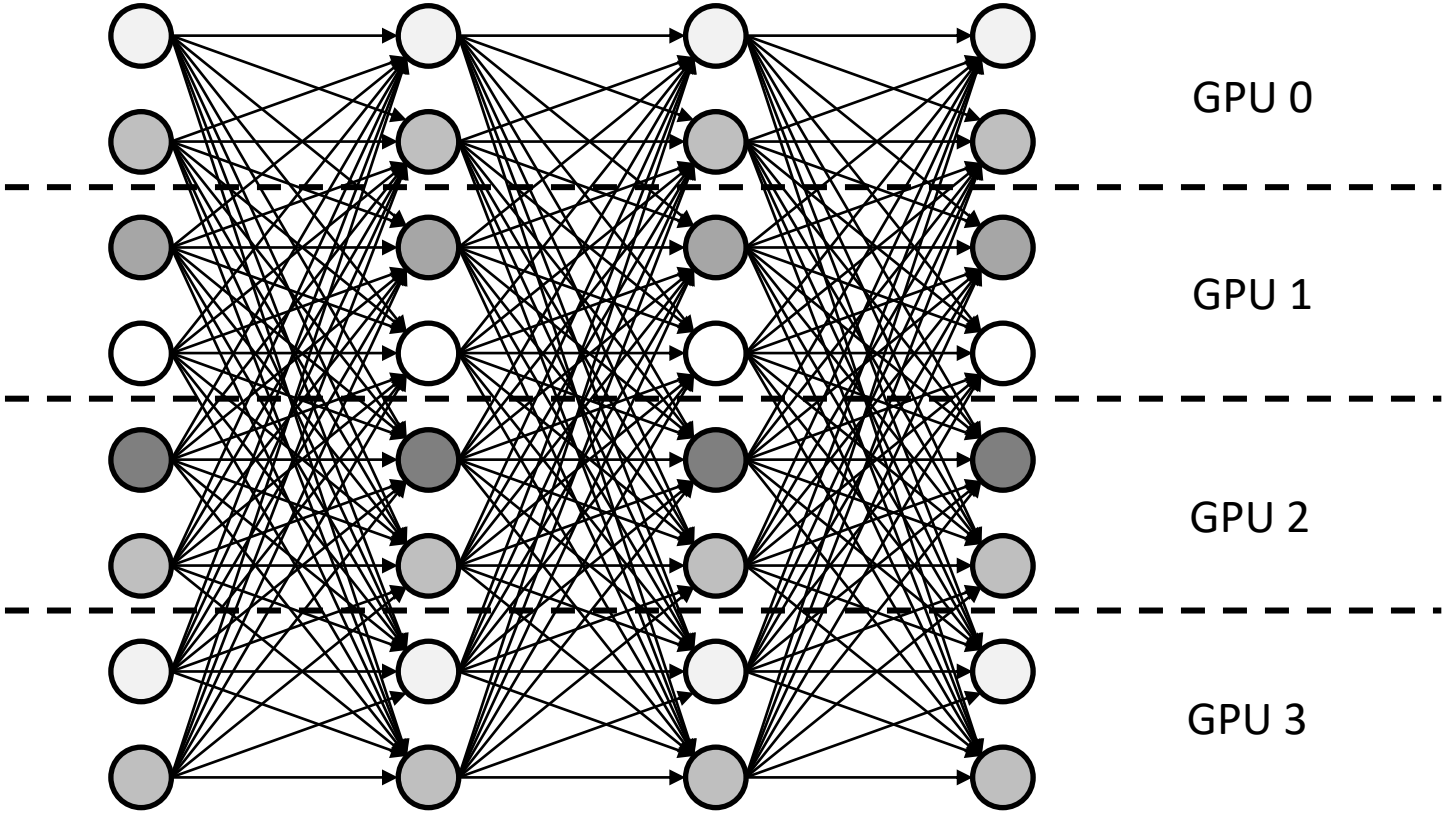
- **Step 1:** local summation within each processor
- **Step 2:** Prefix sum computation using only the rightmost value of each local array
- **Step 3:** Addition of the value computed in Step2 from the left neighbor to each local array element

Partitioning Strategies

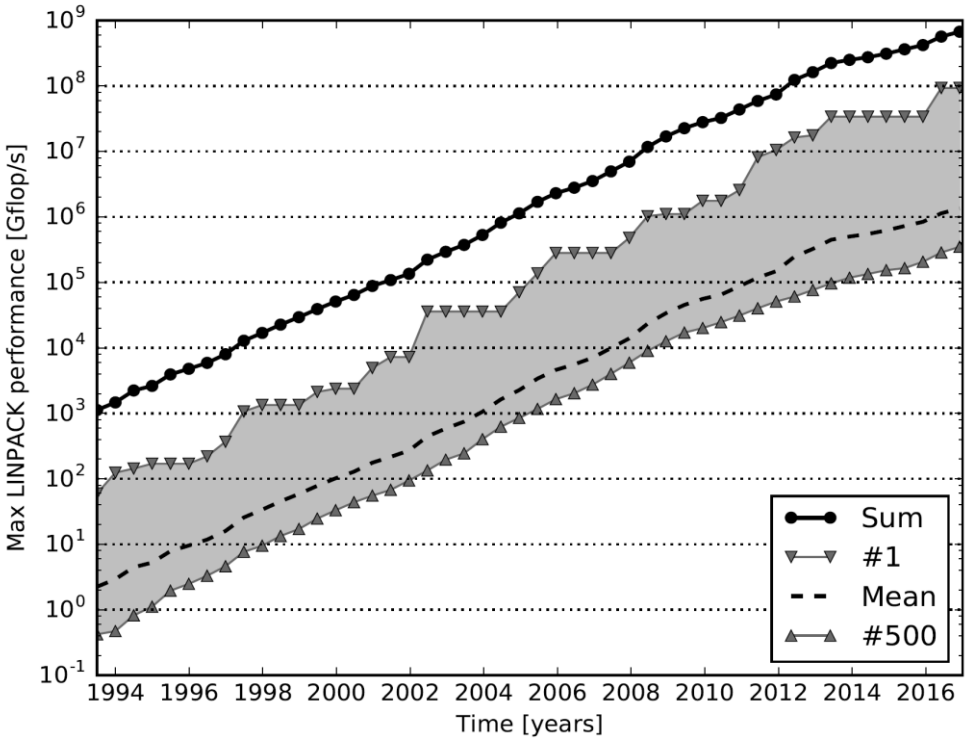


- **Task parallelism**
  - Different binary classifier assigned to a different process (P0, P1, P2)
  - Every process classifies each image using the assigned classifier.
  - Binary classification results for each image are send to P0 and merged
  - Limited parallelism and possible load imbalance
- **Data parallelism**
  - Input images could be divided into a number of batches.
  - Once a process has completed the classification of its assigned batch, a scheduler dynamically assigns a new batch

Model Parallelism for Deep Learning



TOP500 Trends



- Supercomputers ranked according to their maximally achieved LINPACK performance (top500.org)
  - Measures the performance for solving a dense system of linear equations ($A \cdot x = b$) in terms of Flop/s
  - Top-ranked system in 2017: *Sunway Taihu Light* contains over 10 million cores (hybrid system) and achieves 93 PFlop/s
- Green500: Flop/s-per-Watt of achieved LINPACK performance