Parallel & Distributed Computing

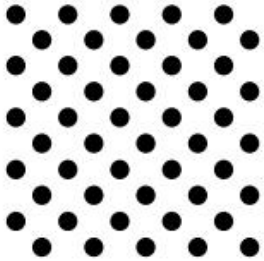
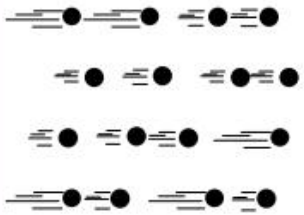
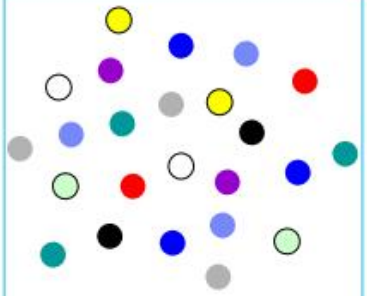
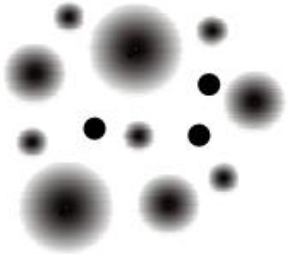
Thoai Nam

What's Big Data?

"**Massive** amounts of **diverse**, **unstructured** data produced by **high-performance** applications."

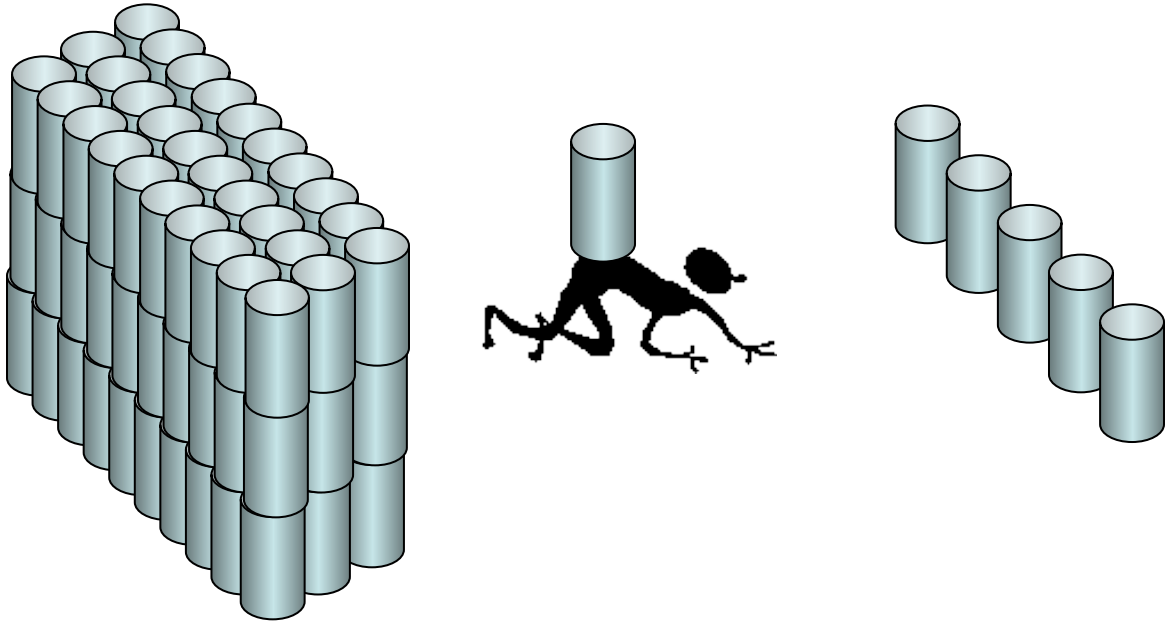
"Data **too large & complex** to be effectively handled by standard database technologies currently founded in most organizations

Some Make it 4V's

Volume	Velocity	Variety	Veracity*
			
Data at Rest	**Data in Motion**	**Data in Many Forms**	**Data in Doubt**
Terabytes to exabytes of existing data to process	Streaming data, milliseconds to seconds to respond	Structured, unstructured, text, multimedia	Uncertainty due to data inconsistency & incompleteness, ambiguities, latency, deception, model approximations

Sequential Processing

- 1 CPU
- Simple
- Big problems?
- **Big data problems???**

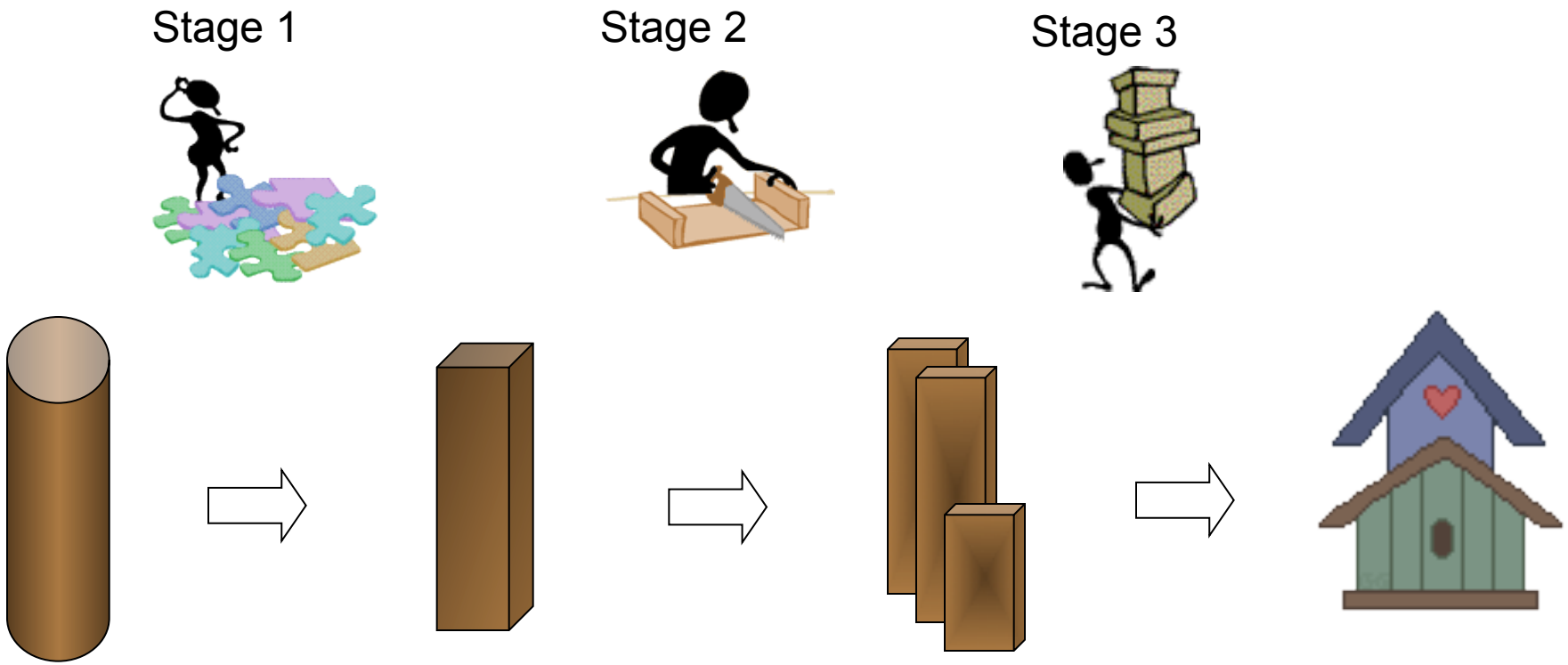


Solutions

- Power processor
  - 50 Hz -> 100 Hz -> 1 GHz -> 4 Ghz -> ... -> Upper bound?
- Smart worker
  - Better algorithms
- Parallel processing

Pipeline

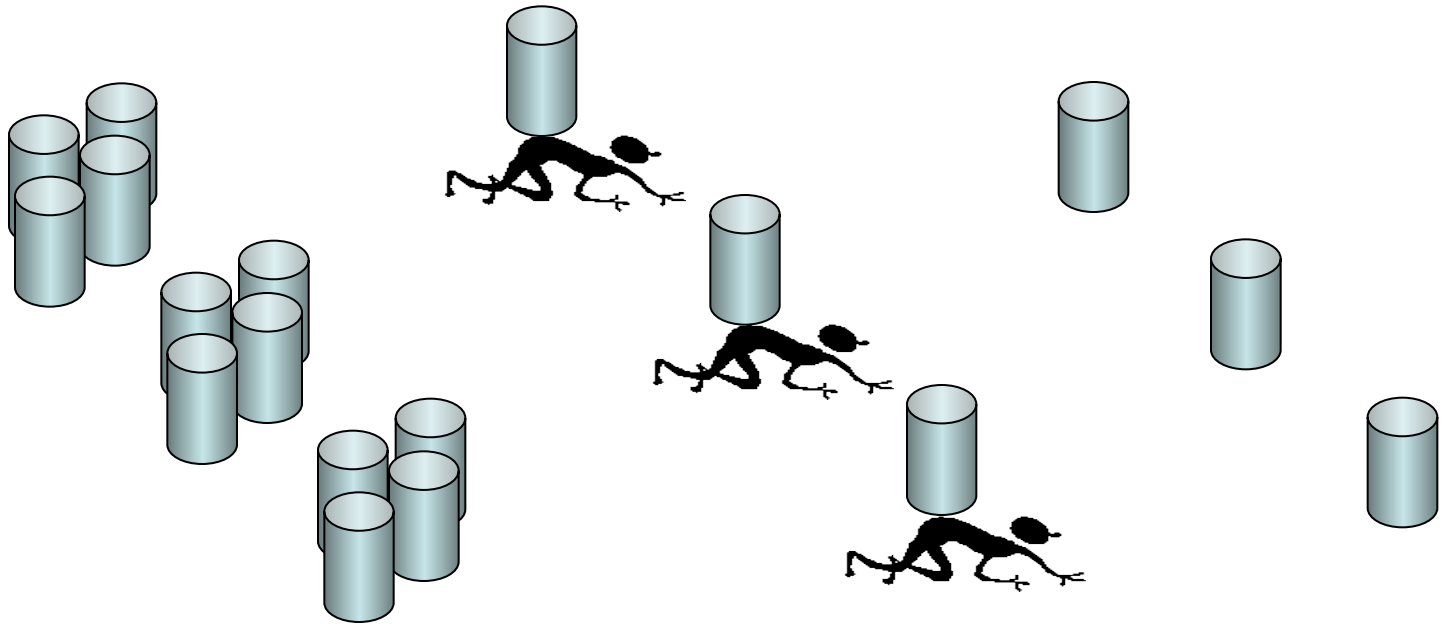
- A number of steps called **segments** or **stages**
- The output of one segment is the input of other segment



Data Parallelism

- Distributing the data across different parallel computing nodes

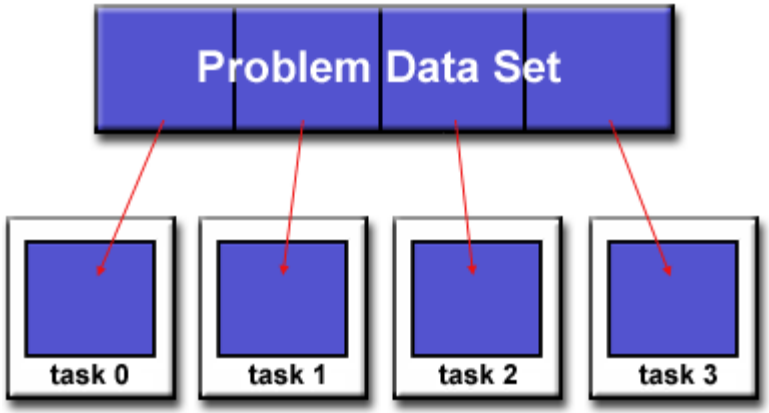
Applying the same operation simultaneously to elements of a data set



Control Parallelism

- Task/Function parallelism
- Distributing execution processes (threads) across different parallel computing nodes

Applying different operations to different data elements simultaneously



Throughput

- The **throughput** of a device is the number of results it produces per unit time

- **High Performance Computing (HPC)**
  - ✧ Needing large amounts of computing power for short periods of time in order to completing the task as soon as possible
- **High Throughput Computing (HTC)**
  - ✧ How many jobs can be completed over a long period of time instead of how fast an individual job can complete

Throughput: Woodhouse problem

- 5 persons complete 1 woodhouse in 3 days
- 10 persons complete 1 woodhouse in 2 days
- How to build 2 houses with 10 persons?

  (1) 10 persons building the 1st woodhouse and then the 2nd one later (sequentially)

  (2) 10 persons building 2 woodhouses concurrently; it means that each group of 5 persons complete a woodhouse

Speedup & Efficiency

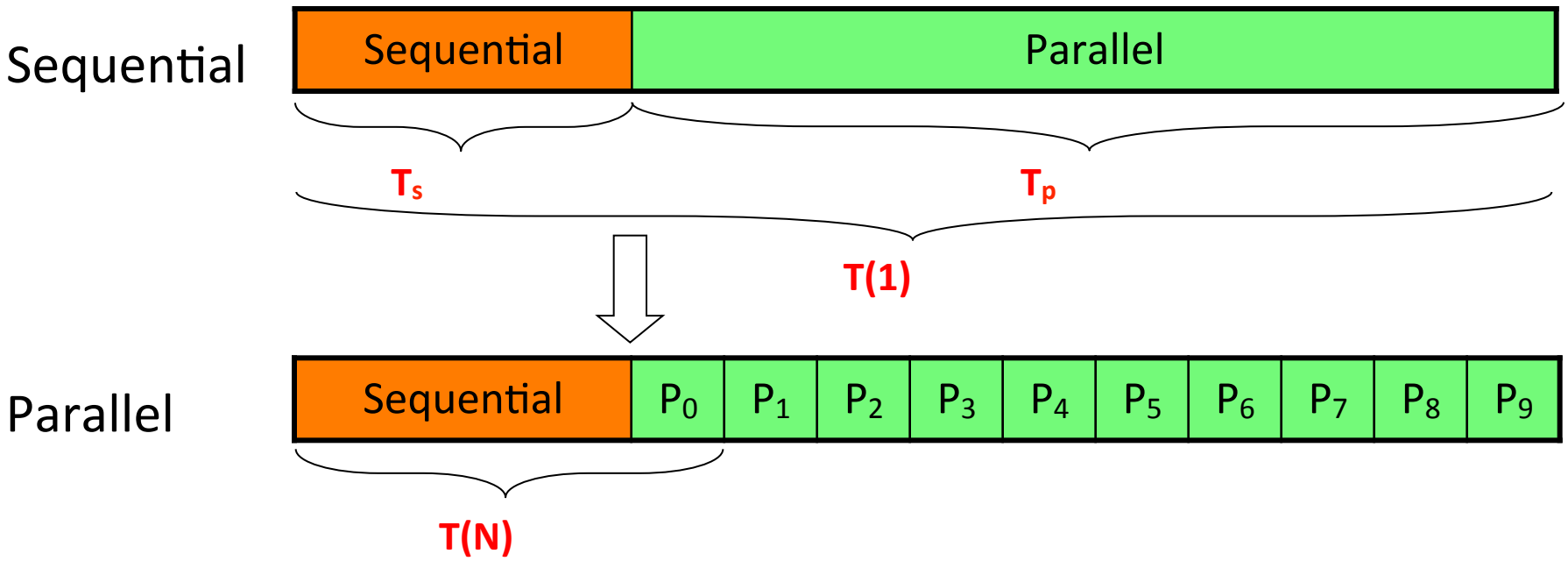
- **Speedup**:

  S = Time(*the most efficient sequential algorithm*) / Time(*parallel algorithm*)

- **Efficiency**:

  E = S / N with N is the number of processors

Amdahl's Law – Fixed Problem Size



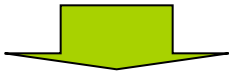
$T_s = \alpha T(1) \Rightarrow T_p = (1-\alpha)T(1)$

$$\mathbf{T(N)} = \alpha T(1) + (1-\alpha)T(1)/N$$

Number of processors

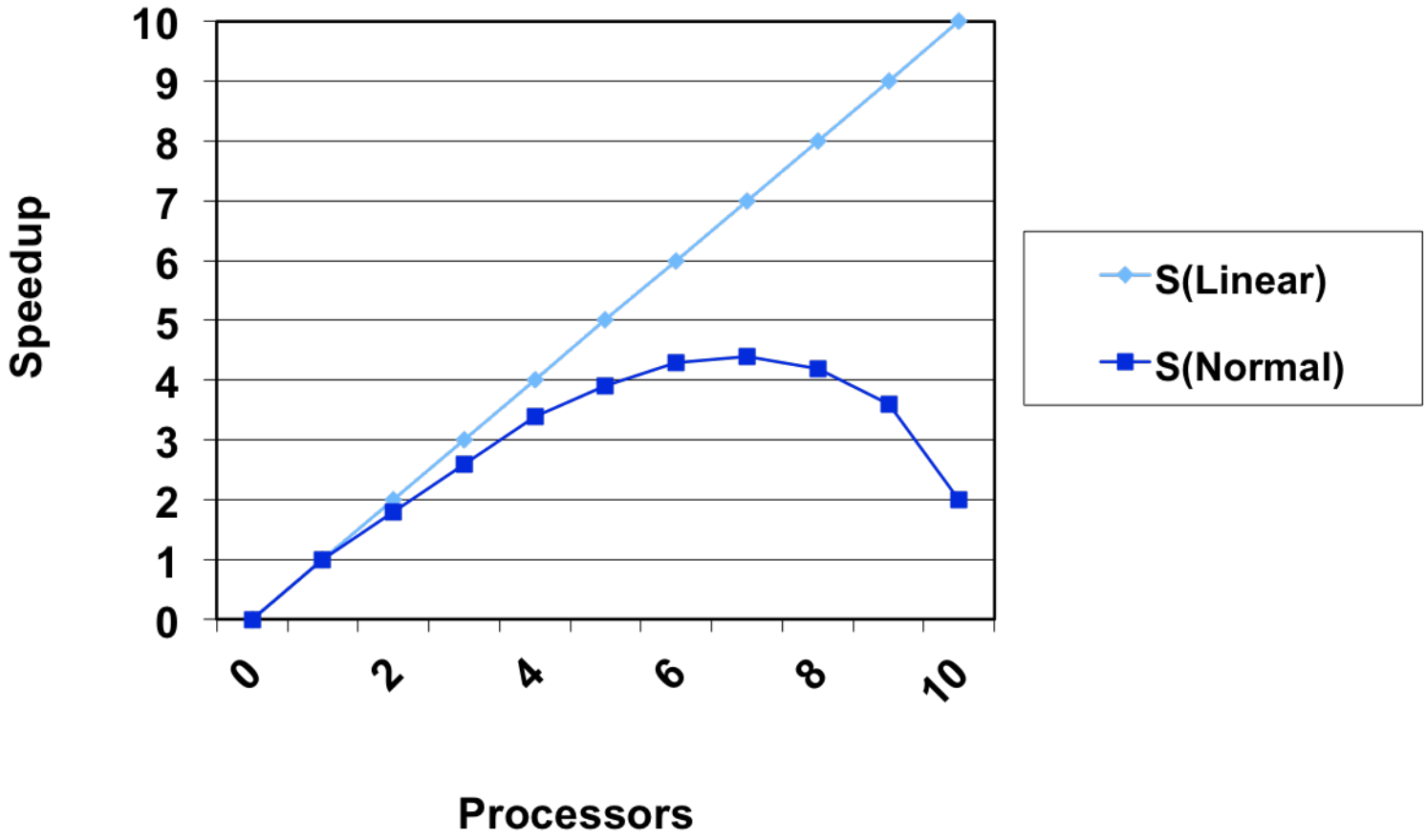
Amdahl's Law – Fixed Problem Size (2)

$$Speedup = \frac{Time(1)}{Time(N)}$$

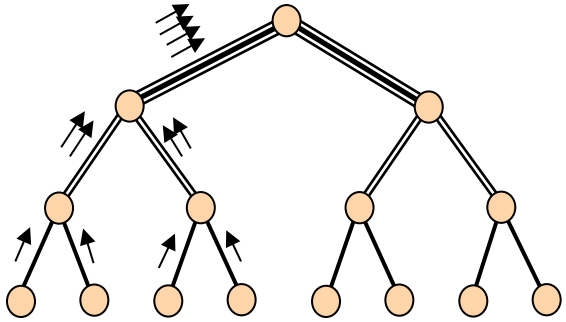


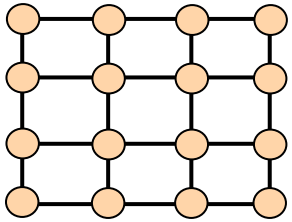
$$Speedup = \frac{T(1)}{\alpha T(1) + \frac{(1-\alpha)T(1)}{N}} = \frac{1}{\alpha + \frac{(1-\alpha)}{N}} \rightarrow \frac{1}{\alpha} \; as \; N \rightarrow \infty$$

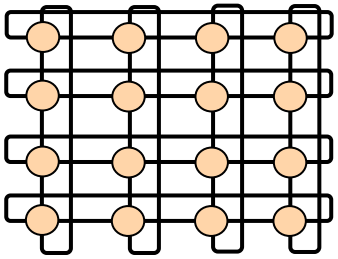
Speedup

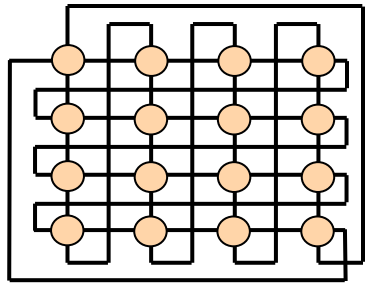

Speedup
10
9
8
7
6
5
4
3
2
1
0
0
2
4
6
8
10
Processors
S(Linear)
S(Normal)

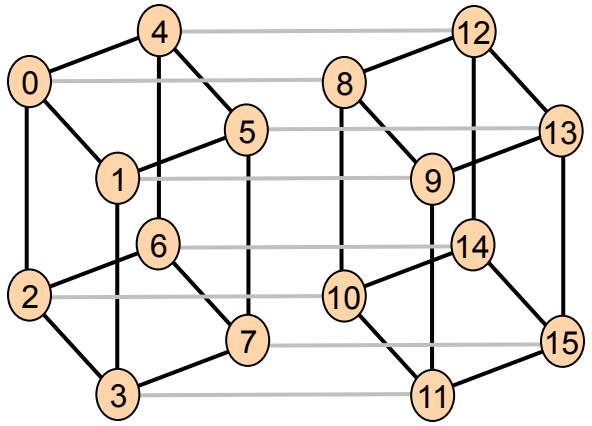
Processor organization









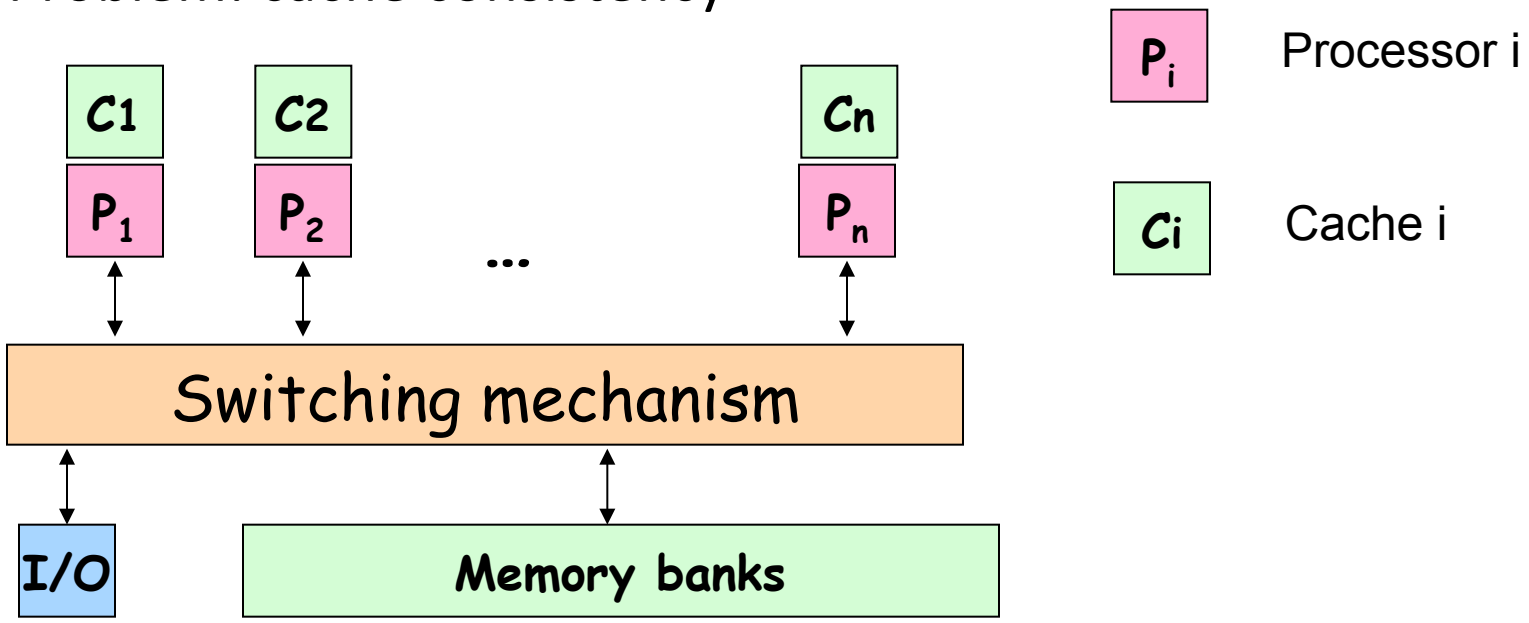


Multiprocessor

- Consists of many fully programmable processors each capable of executing its own program
- Shared address space architecture
- Classified into 2 types
  - Uniform Memory Access (UMA) Multiprocessors
  - Non-Uniform Memory Access (NUMA) Multiprocessors

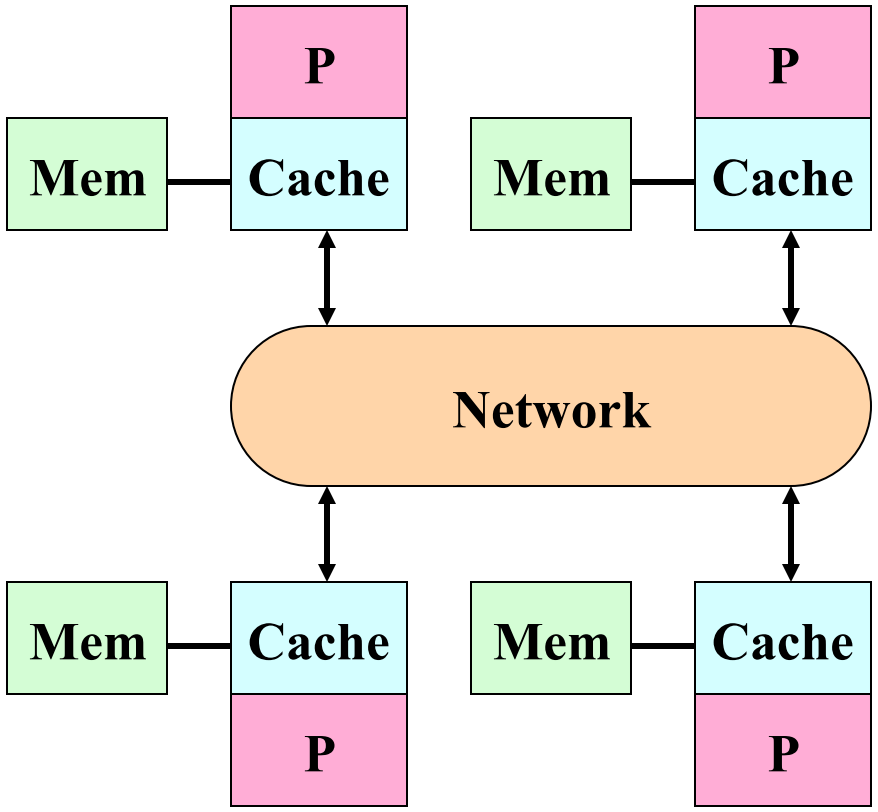
UMA Multiprocessor

- Uses a central switching mechanism to reach a centralized shared memory
- All processors have equal access time to global memory
- Tightly coupled system
- Problem: cache consistency



NUMA Multiprocessor

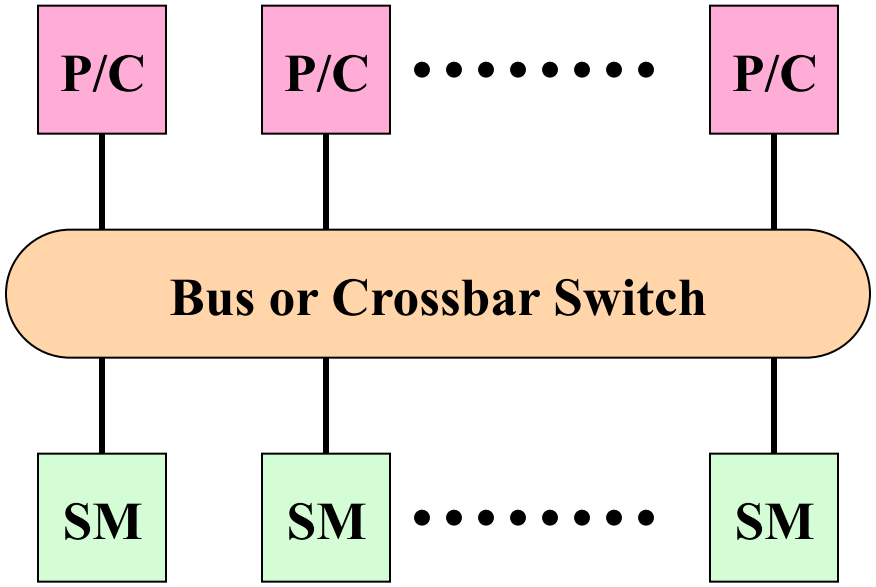
- Distributed shared memory combined by local memory of all processors
- Memory access time depends on whether it is local to the processor
- Caching shared (particularly nonlocal) data?



Distributed Memory

SMP (Symmetric Multi-Processor)

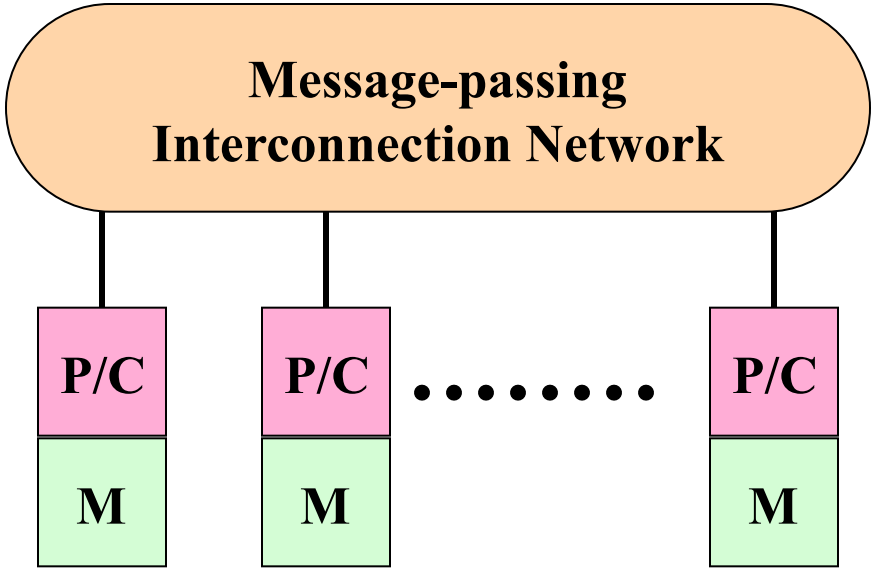
P/C : Microprocessor and Cache
SM: Shared Memory



Multicomputer

- Consists of many processors with their own memory
- No shared memory
- Processors interact via message passing → loosely coupled system

P/C: Microprocessor & Cache
M: Memory



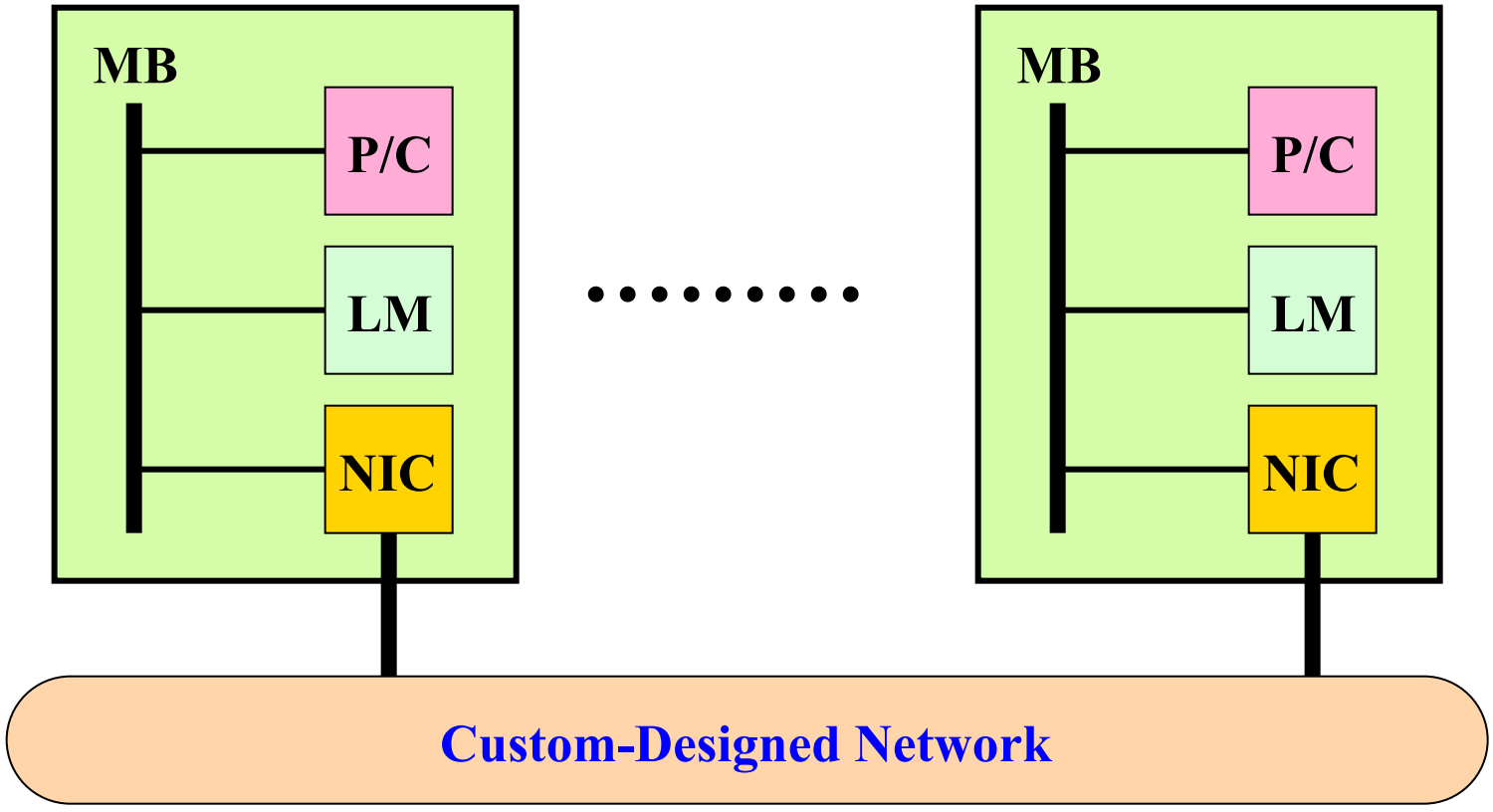
MPP (Massively Parallel Processing)

P/C: Microprocessor & Cache

MB: Memory Bus

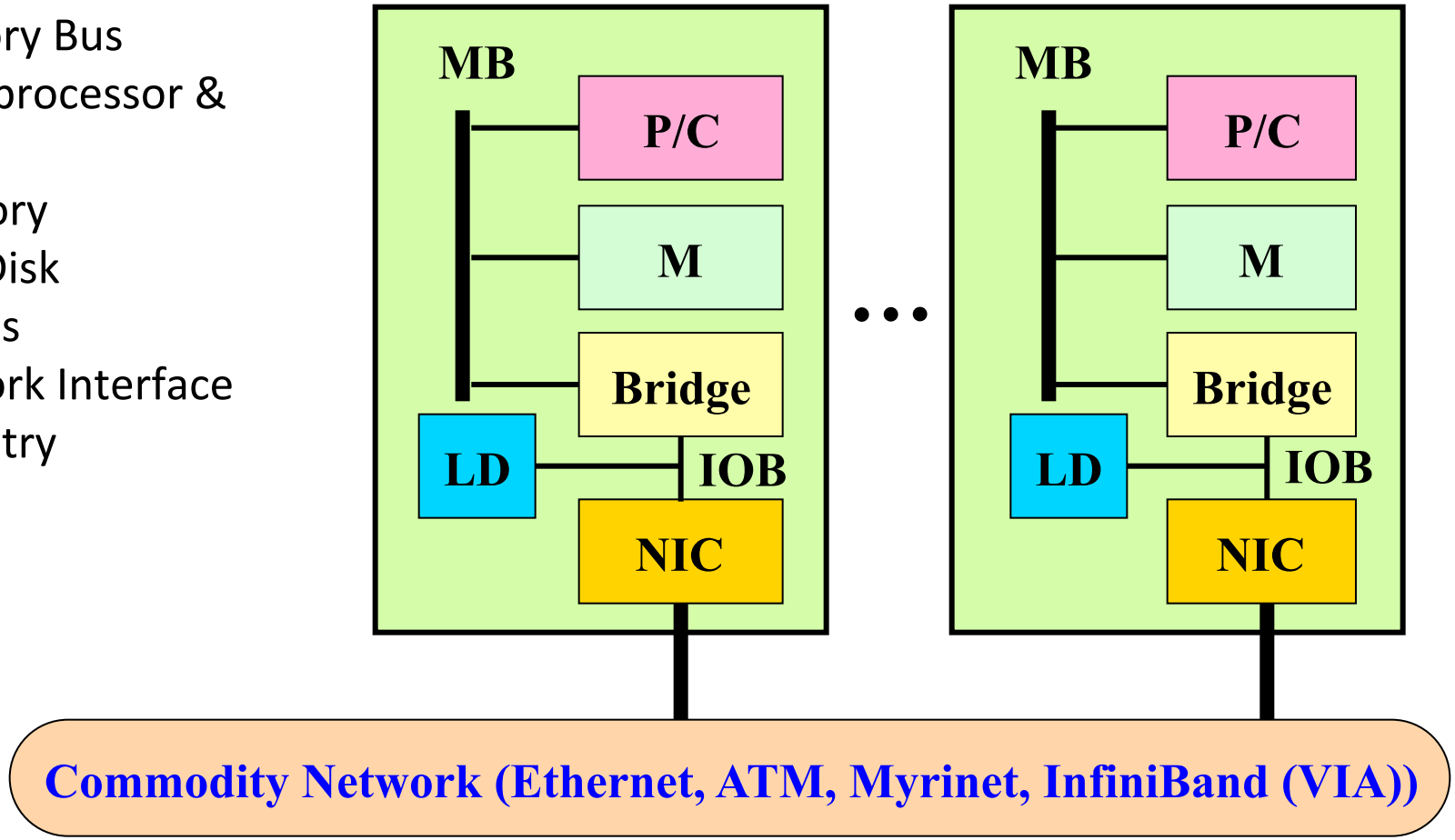
NIC: Network Interface Circuitry

LM: Local Memory



Clusters

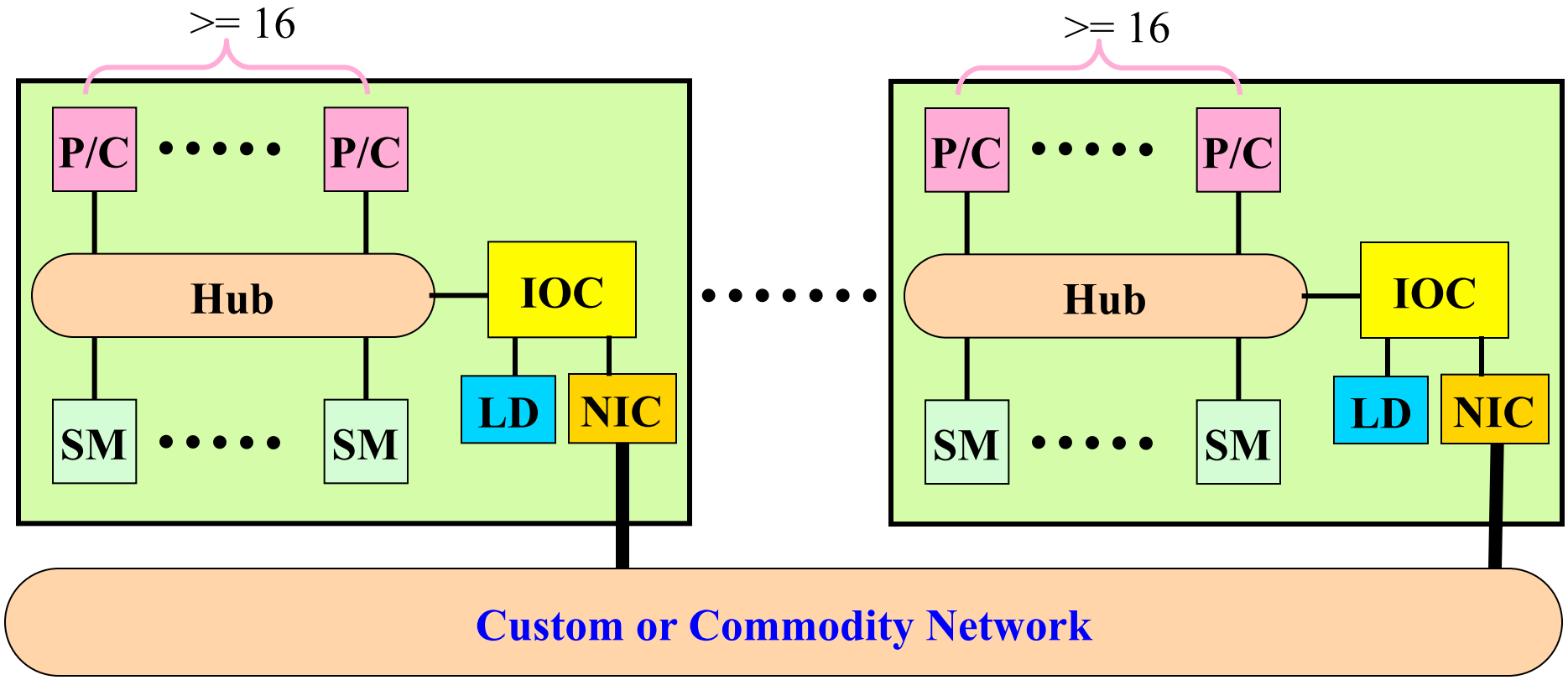
MB: Memory Bus
P/C: Microprocessor & Cache
M: Memory
LD: Local Disk
IOB: I/O Bus
NIC: Network Interface Circuitry



Constellations

P/C: Microprocessor & Cache
NIC: Network Interface Circuitry
IOC: I/O Controller

MB: Memory Bus
SM: Shared Memory
LD: Local Disk

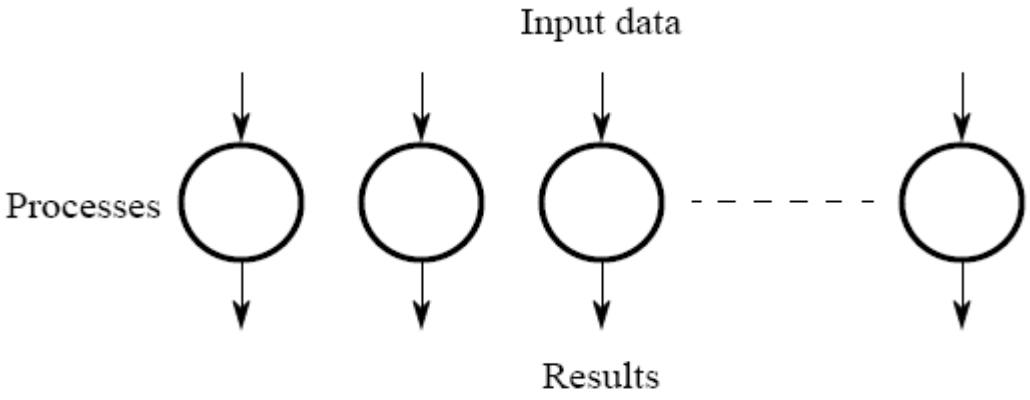


Parallel Techniques

- Embarrassingly Parallel Computations
- Partitioning and Divide-and-Conquer Strategies
- Pipelined Computations
- Synchronous Computations

Embarrassingly Parallel Computations

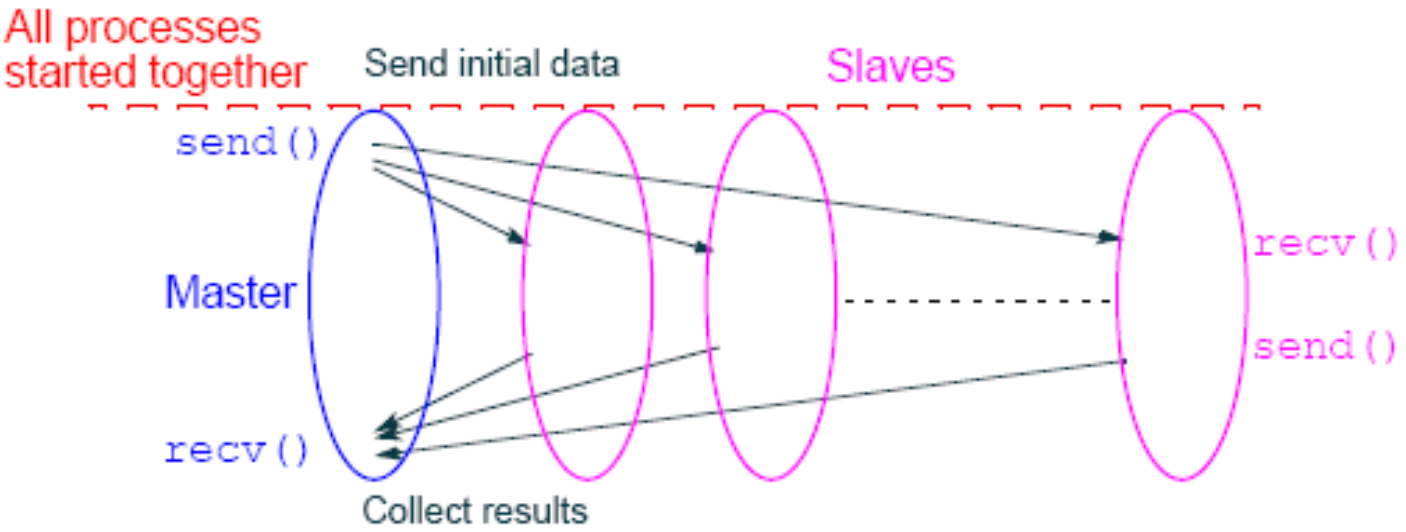
- A computation that can obviously be divided into a number of completely independent parts, each of which can be executed by a separate process(or).



No communication or very little communication between processes
Each process can do its tasks without any interaction with other processes

Embarrassingly Parallel Computations

Practical embarrassingly parallel computation with static process creation and master-slave approach



Usual MPI approach

Low level image processing

Many low level image processing operations only involve local data with very limited if any communication between areas of interest.

Some geometrical operations

Shifting

Object shifted by Δx in the x-dimension and Δy in the y-dimension:

$$x' = x + \Delta x$$
$$y' = y + \Delta y$$

where x and y are the original and x' and y' are the new coordinates.

Scaling

Object scaled by a factor S_x in x-direction and S_y in y-direction:

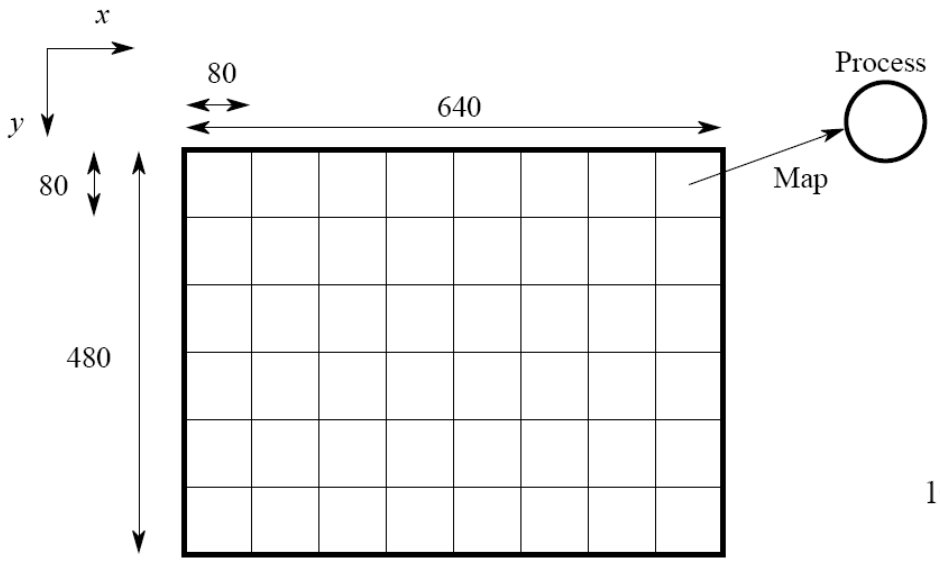
$$x' = xS_x$$
$$y' = yS_y$$

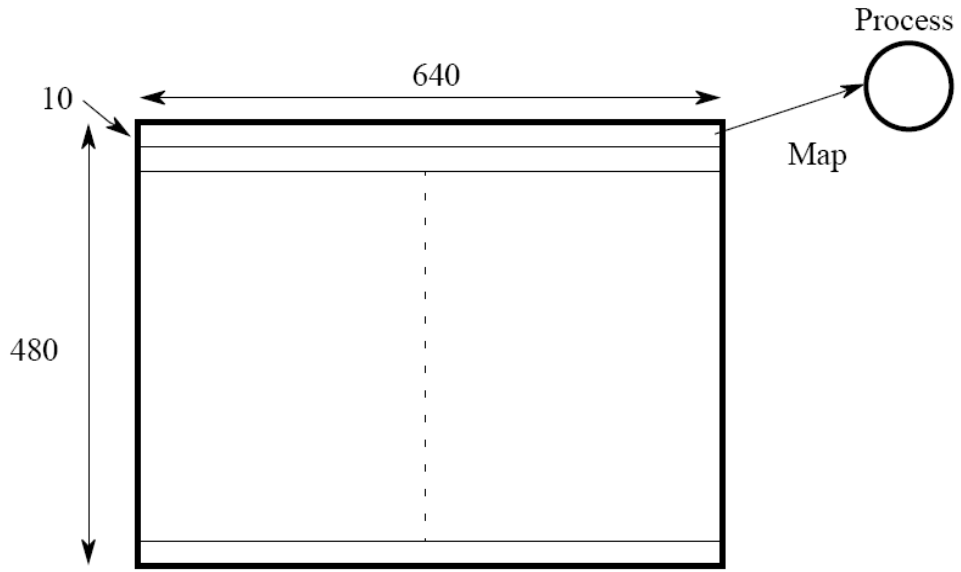
Rotation

Object rotated through an angle θ about the origin of the coordinate system:

$$x' = x \cos\theta + y \sin\theta$$
$$y' = -x \sin\theta + y \cos\theta$$

Partitioning into regions for individual processes





Partitioning and Divide-and-Conquer Strategies

Partitioning

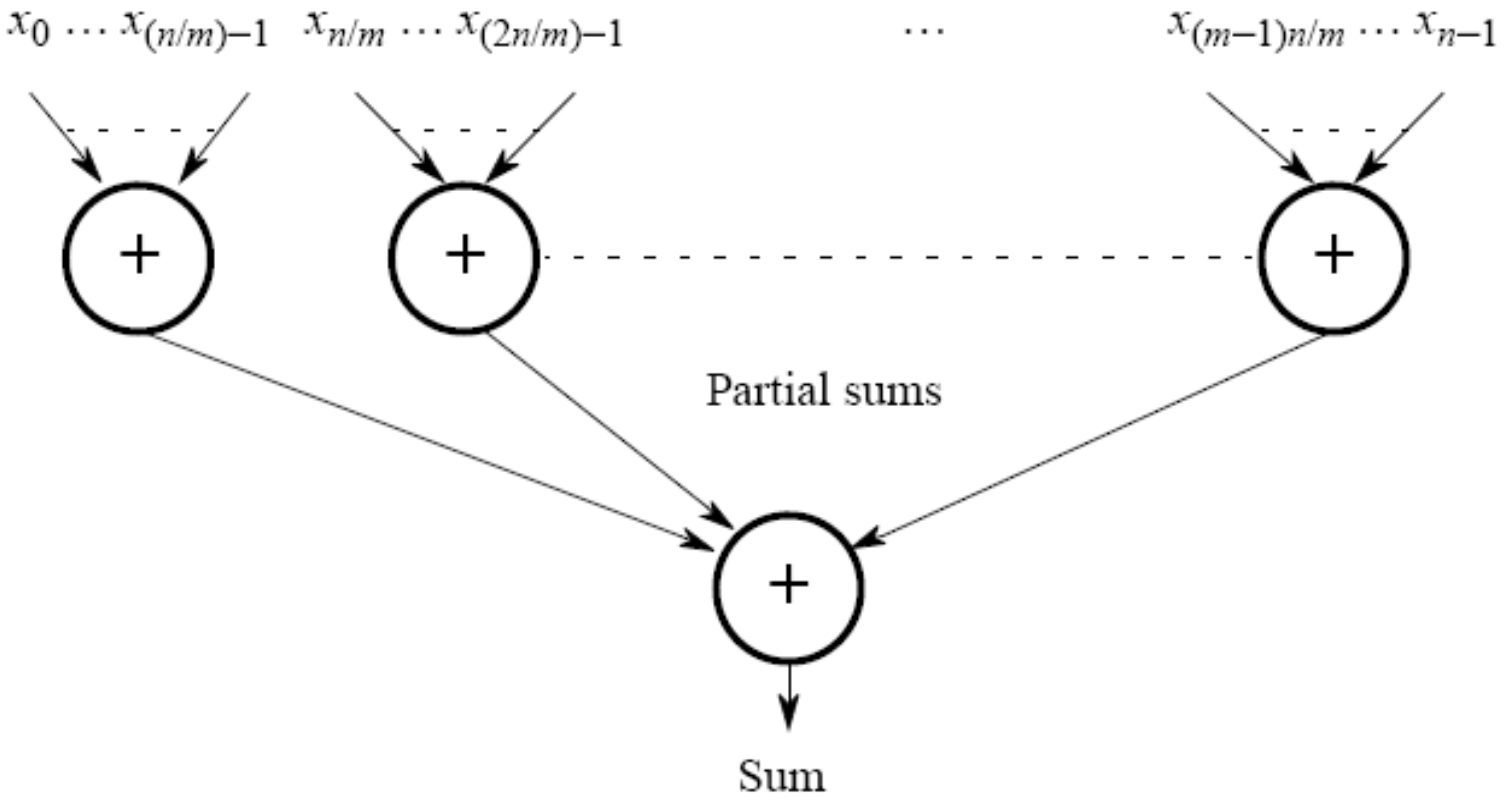
Partitioning simply divides the problem into parts.

Divide and Conquer

Characterized by dividing problem into sub-problems of same form as larger problem.

Further divisions into still smaller sub-problems, usually done by recursion.Recursive divide and conquer amenable to parallelization because separate processes can be used for divided parts. Also usually data is naturally localized.

Partitioning a sequence of numbers into parts and adding the parts



Bucket sort

One "bucket"assigned to hold numbers that fall within each region. Numbers in each bucket sorted using a sequential

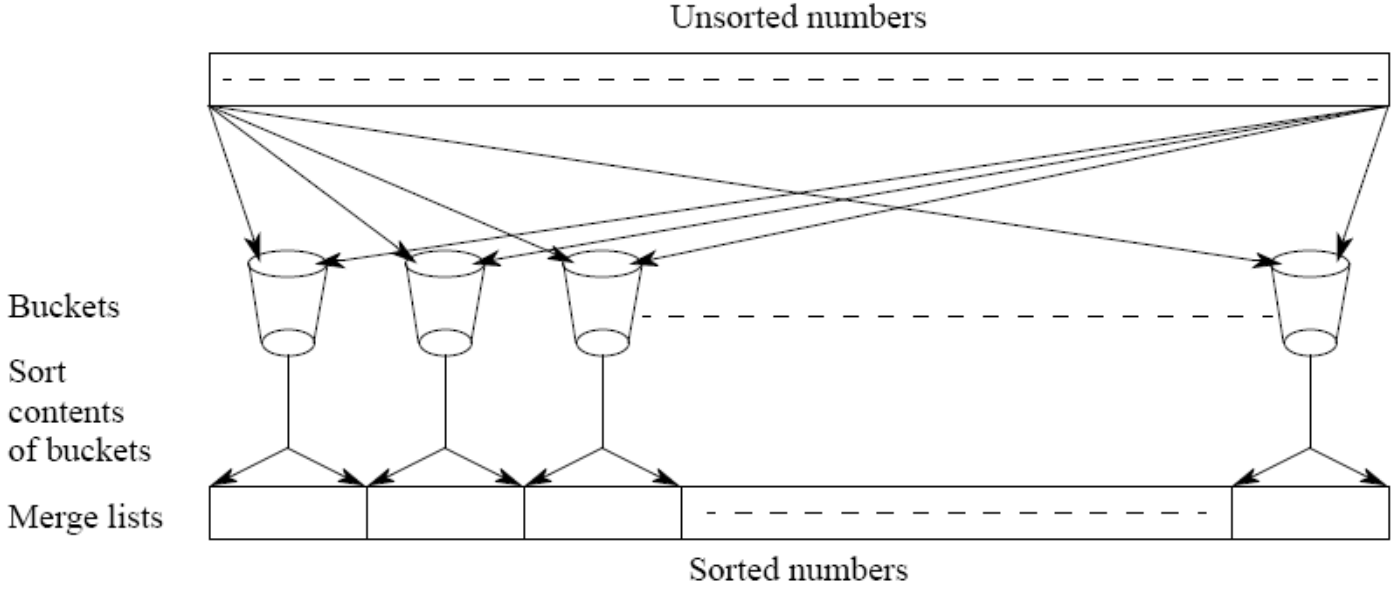
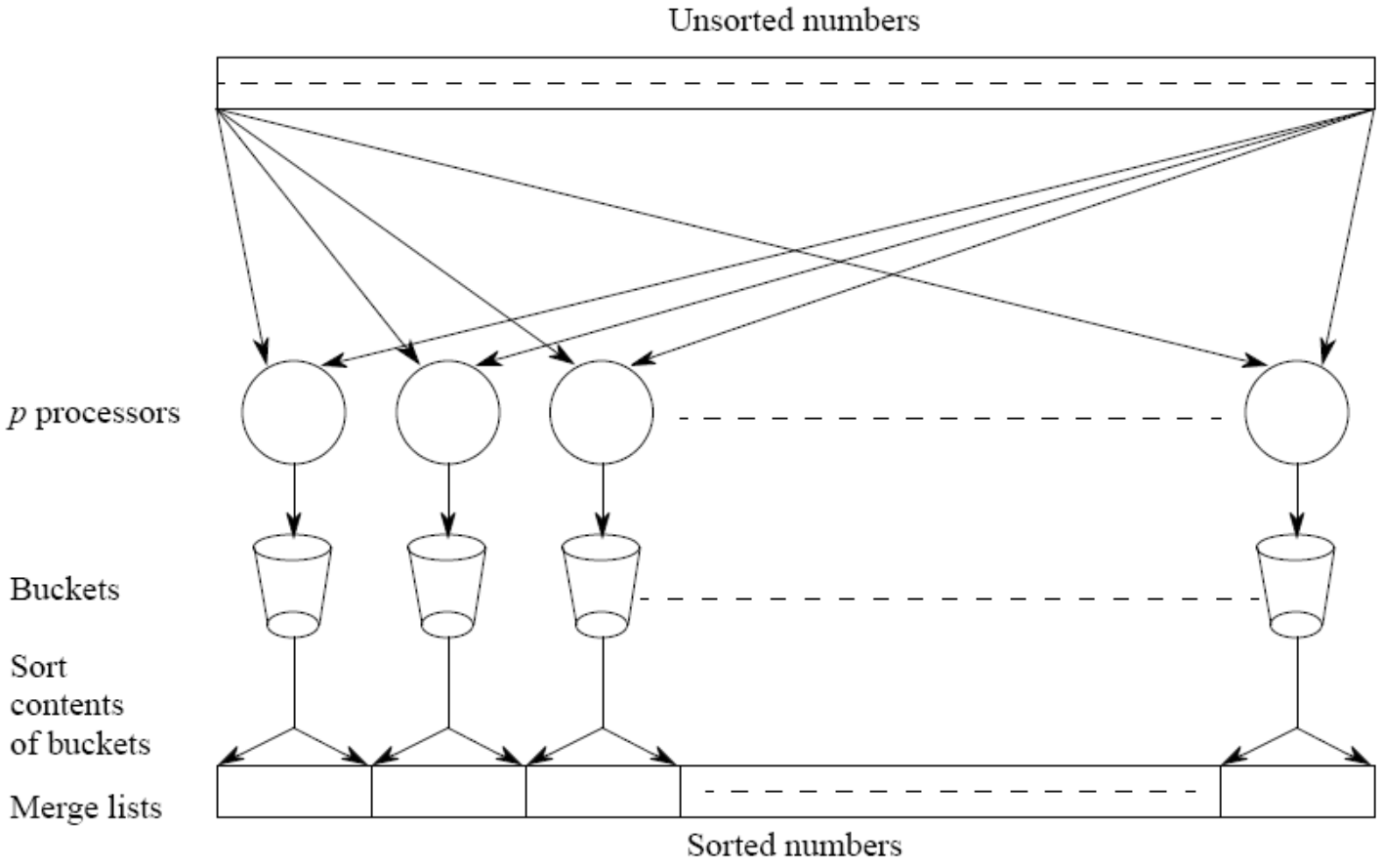


Figure 4.8 Bucket sort.

Sequential sorting time complexity: O(nlog(n/m).Works well if the original numbers uniformly distributed across a known interval, say 0 to *a* -1.

Parallel version of bucket sort
Simple approach

Assign one processor for each bucket.



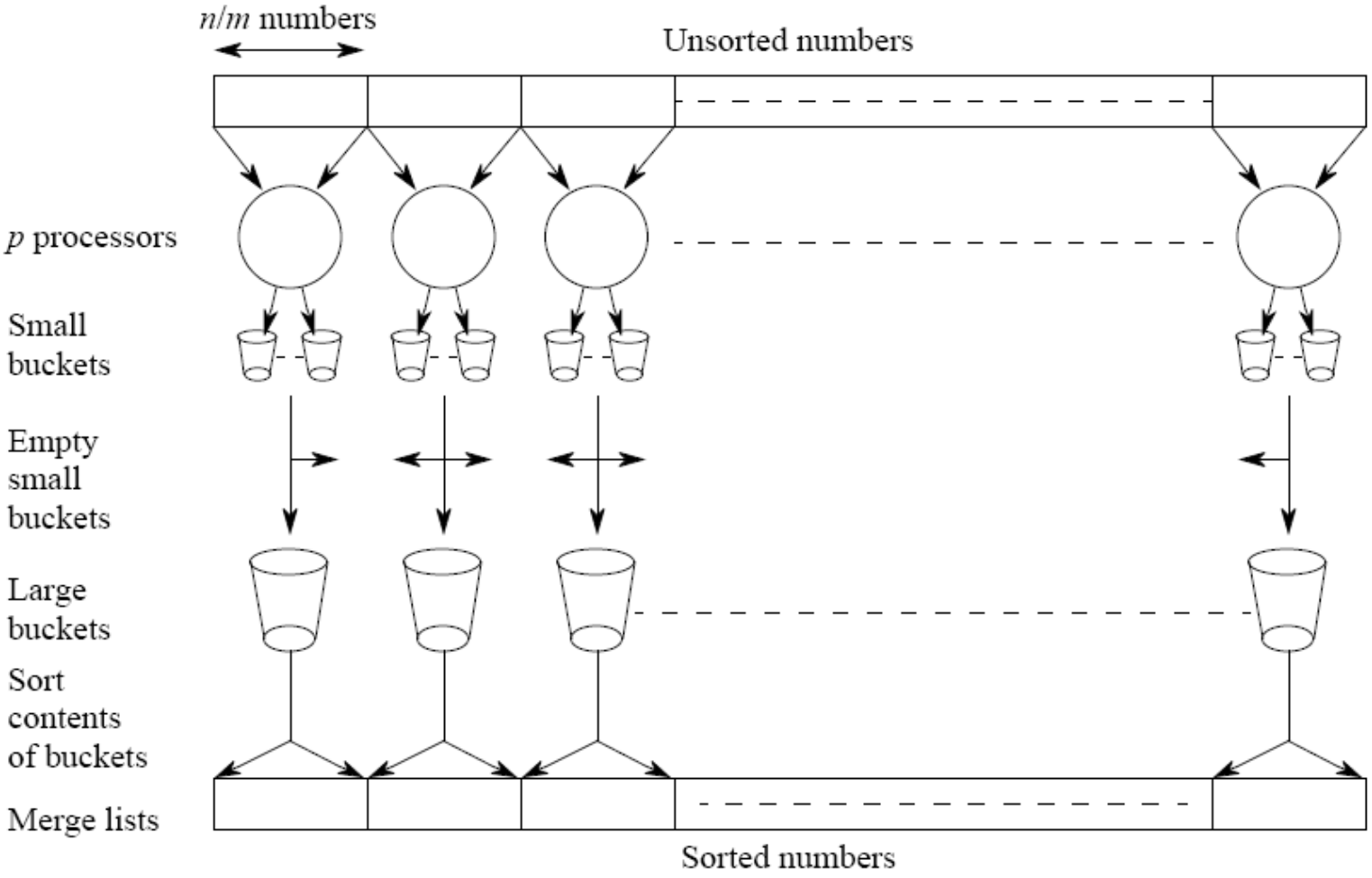
Further Parallelization

- Partition sequence into *m* regions, one region for each processor.

- Each processor maintains *p* "small"buckets and separates numbers in its region into its own small buckets.

- Small buckets then emptied into *p* final buckets for sorting, which requires each processor to send one small bucket to each of the other processors (bucket *i* to processor *i*).

Another parallel version of bucket sort

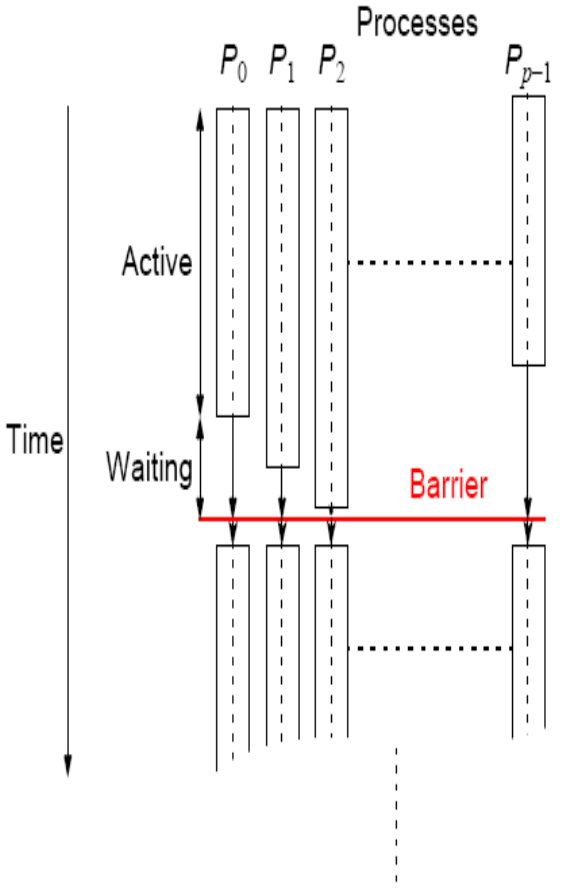


Synchronous Computations

- Barrier concept
- Data parallel computations
- Examples of global and local barriers

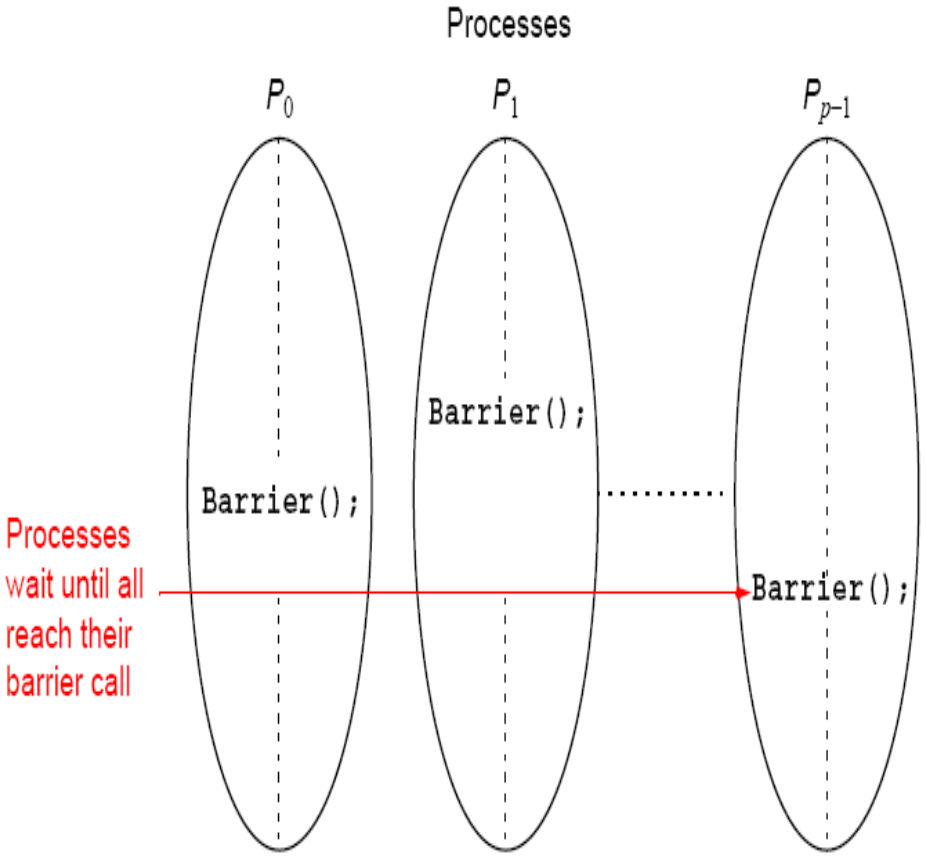
Barrier

- A basic mechanism for synchronizing processes - inserted at the point in each process where it must wait
- All processes can continue from this point when all the processes have reached it (in some implementations, when a stated number of processes have reached this point)
- Processes reaching barrier at different times

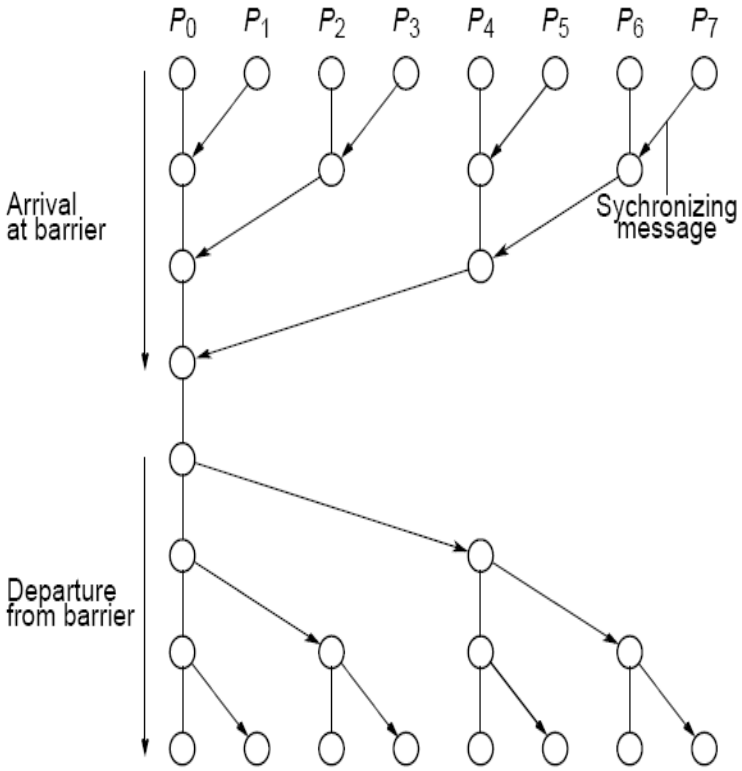


Barrier in message-passing

In message-passing systems, barriers provided with library routines:



Tree barrier

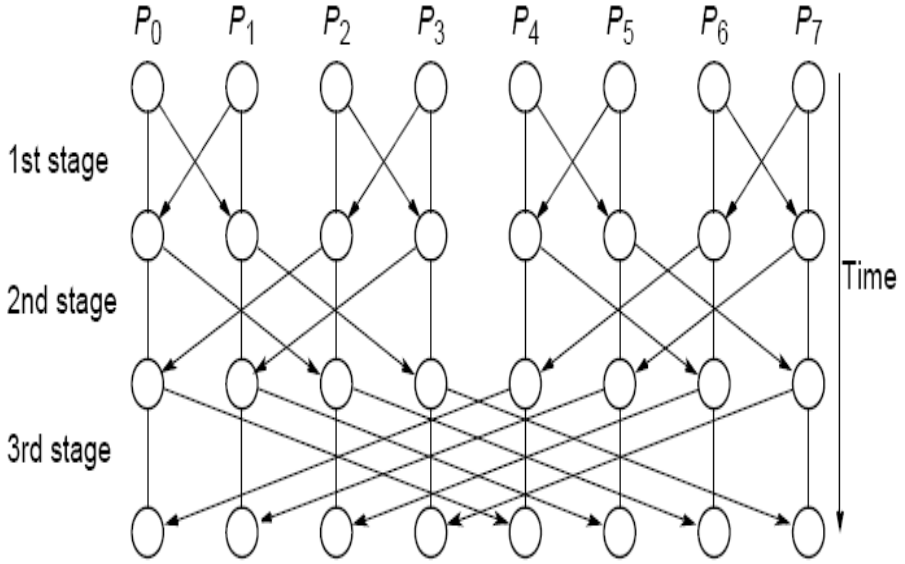


Butterfly barrier

1st stage $P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$

2nd stage $P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$

3rd stage $P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$



Synchronized Computations

- Can be classified as:
  - Fully synchronous
  - Locally synchronous
- In **fully synchronous**, all processes involved in the computation must be synchronized.
- In **locally synchronous**, processes only need to synchronize with a set of logically nearby processes, not all processes involved in the computation

Fully Synchronized Computation Examples

- **Data Parallel Computations**
  - Same operation performed on different data elements simultaneously; i.e., in parallel.
  - Particularly convenient because:
    - Ease of programming (essentially only one program).
    - Can scale easily to larger problem sizes.

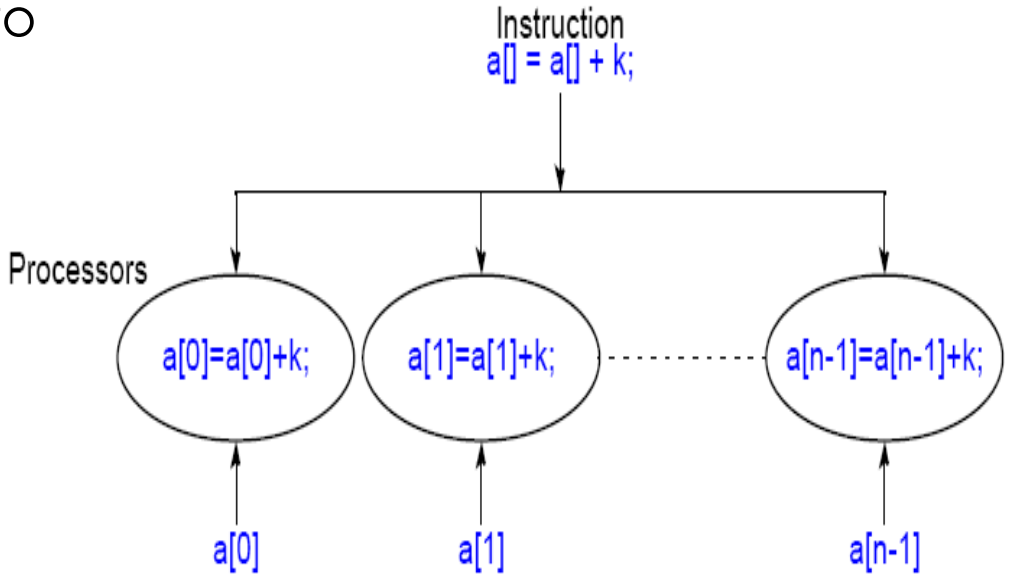
Data Parallel Computations

- To add the same constant to each element of an array:

  for (i = 0; i < n; i++)
  a[i] = a[i] + k;

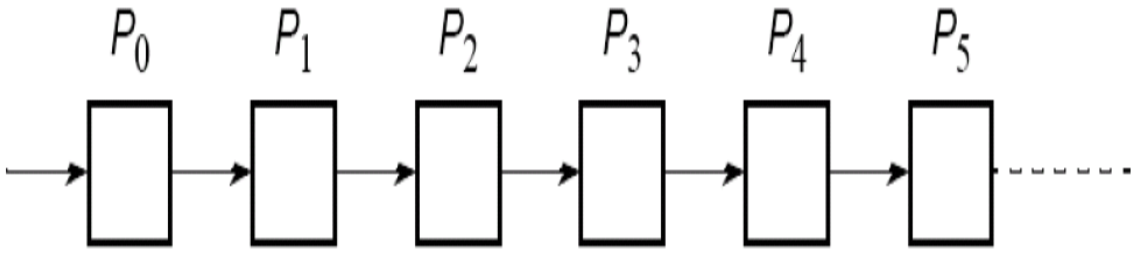
- The statement:
  - **a[i] = a[i] + k;**

  could be executed simultaneously by multiple processors, each using a different index **i (0 < i <= n)**.



Pipeline Technique

- Problem divided into a series of tasks that have to be completed one after the other (the basis of sequential programming). Each task executed by a separate process or processor.



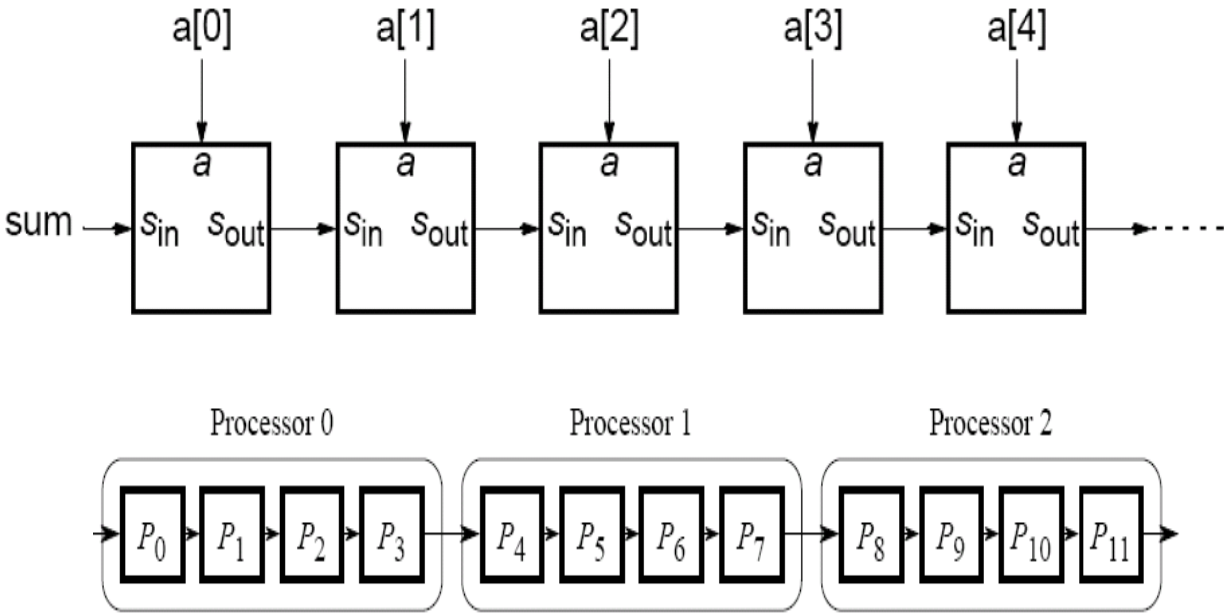
Pipeline for an unfolded loop

Add all the elements of array **a** to an accumulating sum:

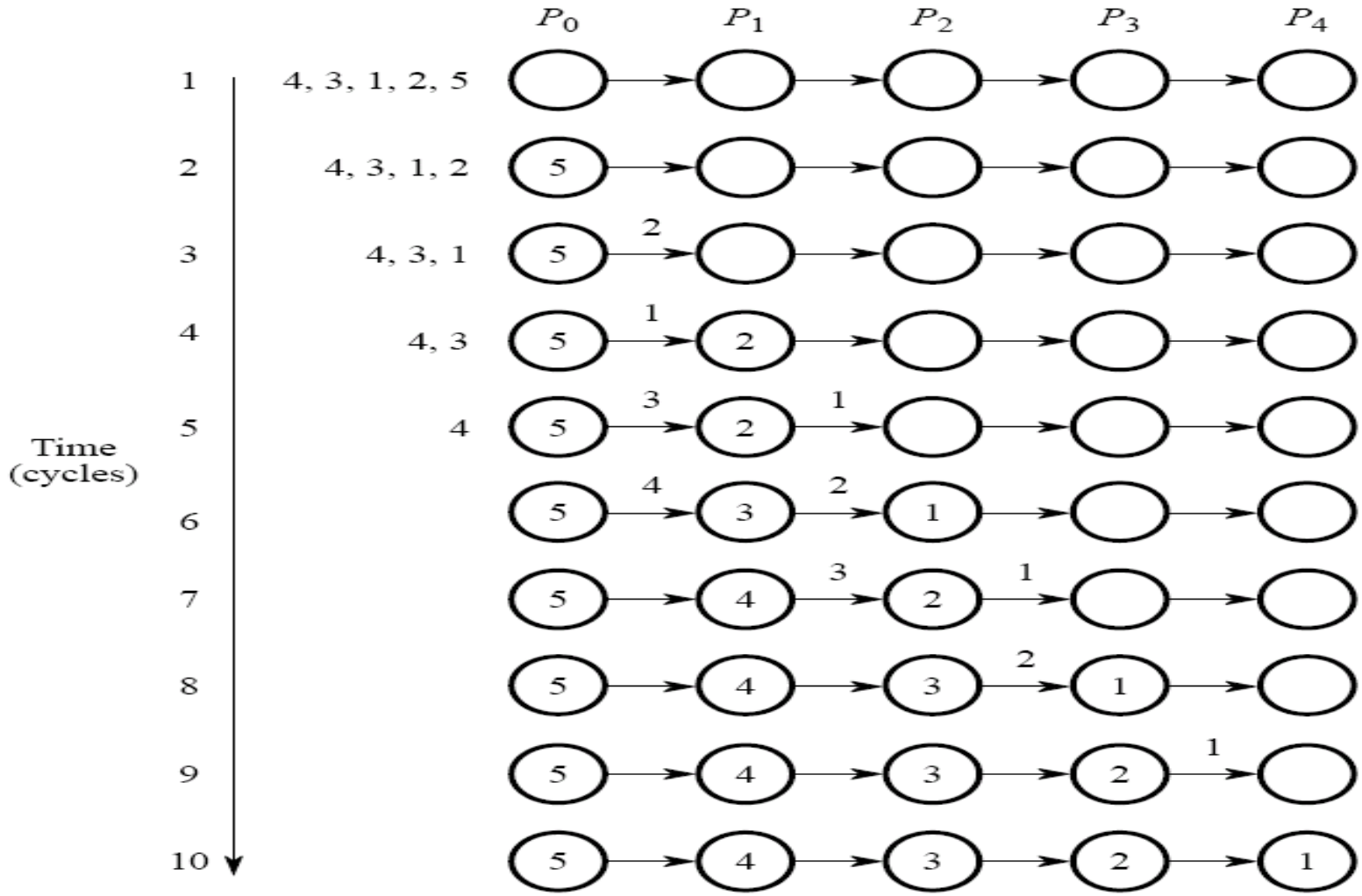
```
for (i = 0; i < n; i++)
sum = sum + a[i];
```

The loop could be "unfolded" to yield

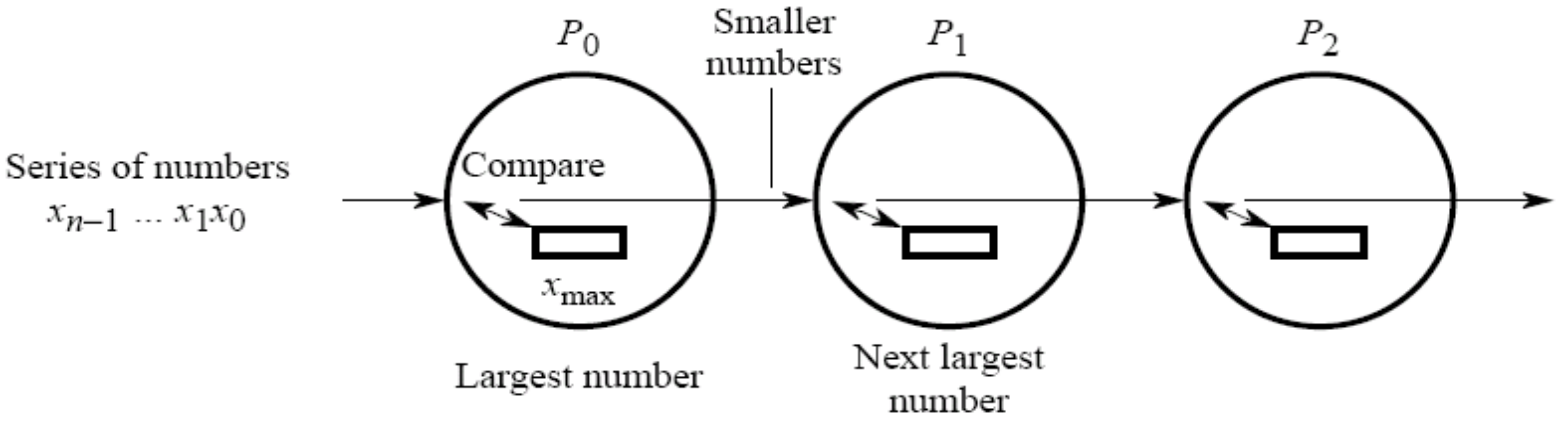
```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
                ...
```



Sorting Numbers



Sorting Numbers: pipeline computation



The basic algorithm for process *Pi* is:

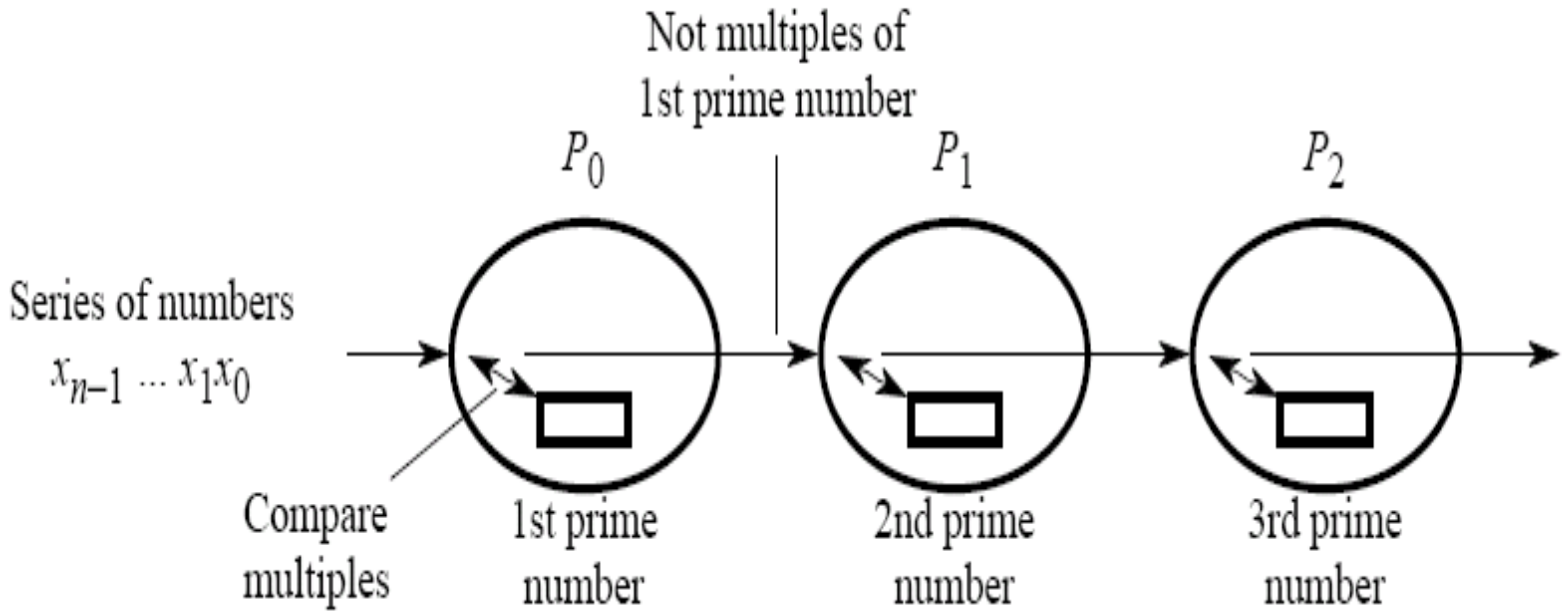
```
recv (&number, Pi-1);
if (number > x) {
        send (&x, Pi+1);
        x = number;
} else
        send (&number, Pi+1);
```

With *n* numbers, number *i*th process is to accept = *n-i*. Number of passes onward = *n-i*-1Hence, a simple loop could be used.

Prime Number Generation
Sieve of Eratosthenes

- Series of all integers generated from 2
- First number, 2, is prime and kept
- All multiples of this number deleted as they cannot be prime
- Process repeated with each remaining number
- The algorithm removes non-primes, leaving only primes
- Type 2 pipeline computation

Prime Number Generation



Prime Number Generation

- The code for a process, *Pi*, could be based upon:

```
recv(&x, Pi-1);/* repeat following for each number */
recv(&number, Pi-1);
if ((number % x) != 0)
        send(&number, P i+1);
```

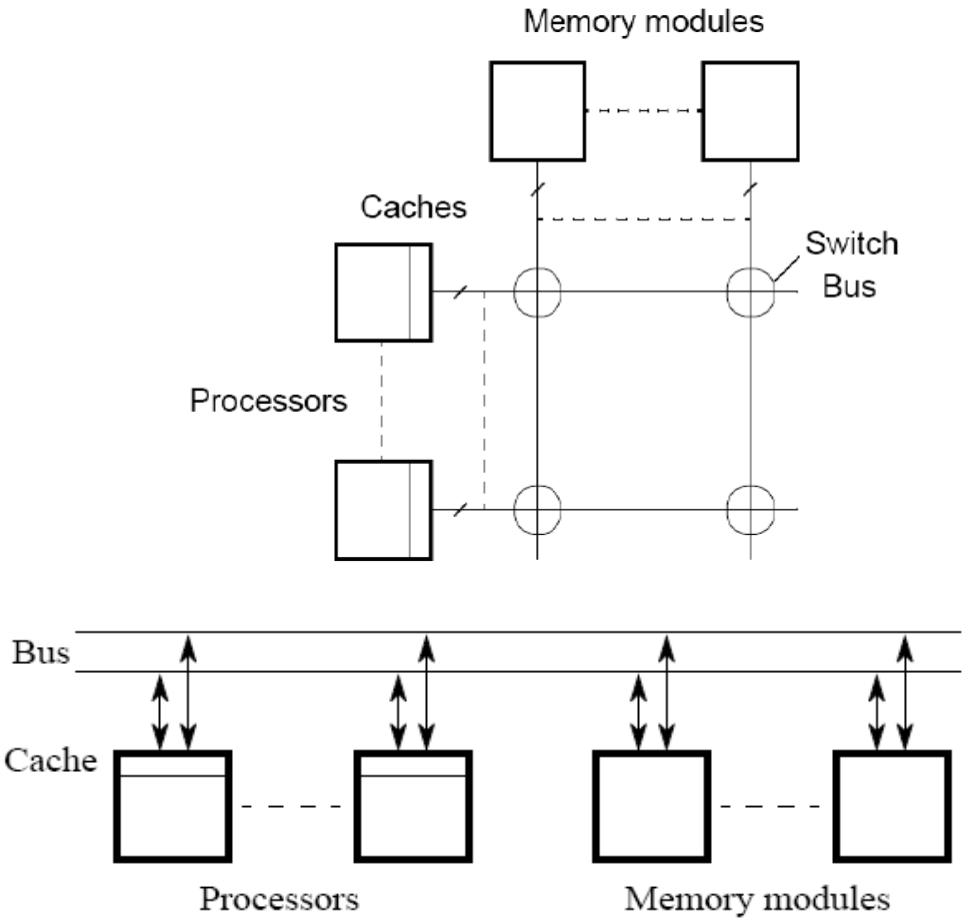
- Each process will not receive the same number of numbers and is not known beforehand. Use a "terminator" message, which is sent at the end of the sequence:

```
recv(&x, Pi-1);
for (i = 0; i < n; i++) {
        recv(&number, Pi-1);
        If (number == terminator)
                break;
        If (number % x) != 0)
                send(&number, P i+1);
}
```

Programming with Shared Memory

Shared memory multiprocessor system

- Any memory location can be accessible by any of the processors.
- A *single address space* exists, meaning that each memory location is given a unique address within a single range of addresses.
- Generally, shared memory programming more convenient although it does require access to shared data to be controlled by the programmer (using critical sections etc.)

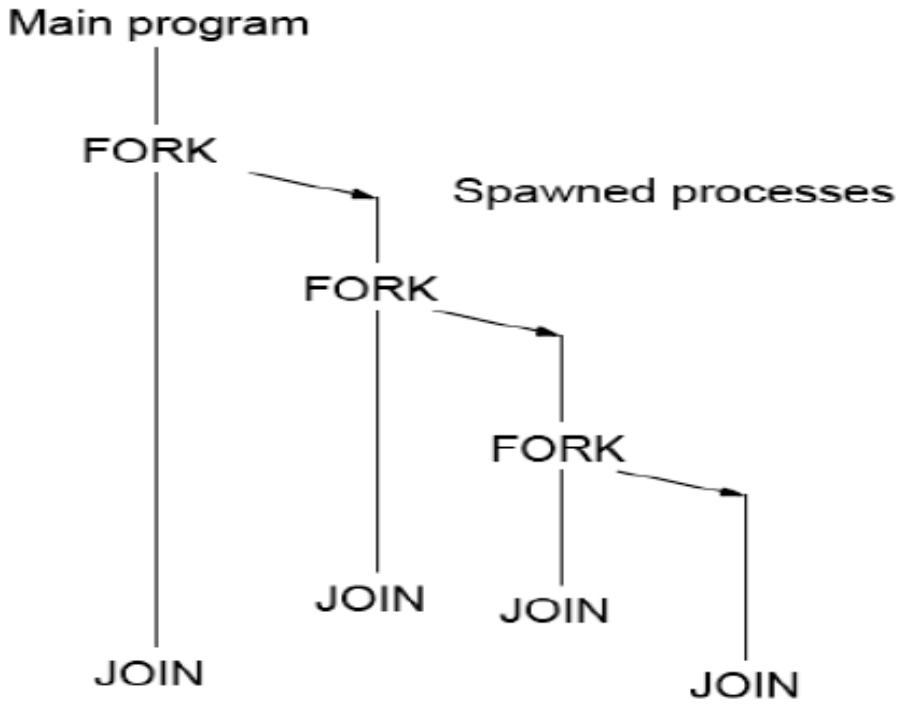


Alternatives for Programming Shared Memory Multiprocessors

- Using heavyweight processes.
- Using threads. Example **Pthreads**
- Using a completely new programming language for parallel programming -not popular. Example Ada.
- Using an existing sequential programming language supplemented with compiler directives for specifying parallelism. Example **OpenMP**

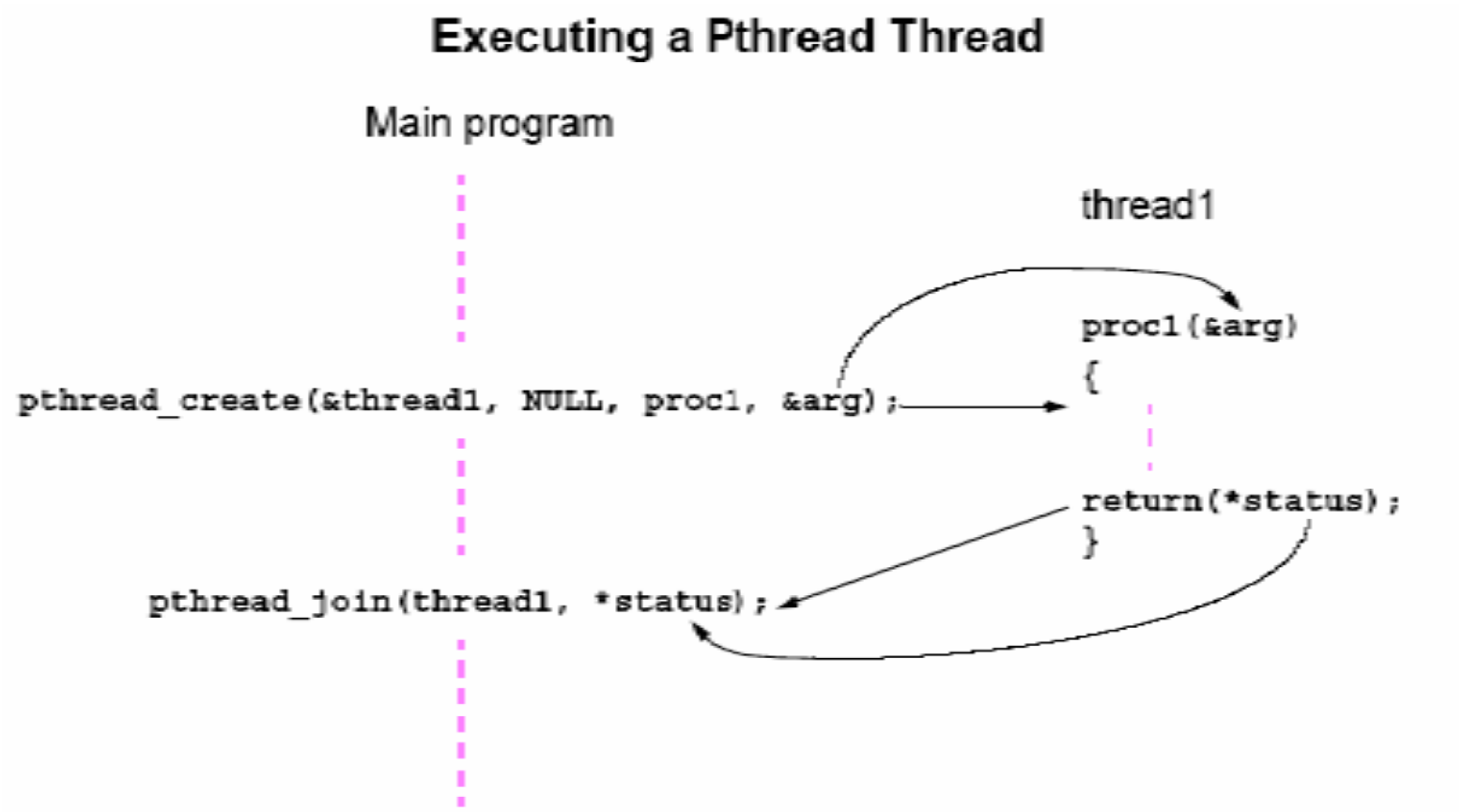
Heavyweight Processes

- Processor time shares between processes, switching from one process to another.



Pthreads

- IEEE Portable Operating System Interface, POSIX, sec. 1003.1 standard



OpenMP

- An accepted standard developed in the late 1990s by a group of industry specialists.
- Consists of a small set of compiler directives, augmented with a small set of library routines and environment variables using the base language Fortran and C/C++.
- The compiler directives can specify such things as the par and forall operations described previously.
- Several OpenMP compilers available.

Hello world

```
#include <omp.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    int id, nthreads;
#pragma omp parallel private(id) {
    id = omp_get_thread_num();
    printf("Hello World from thread %d\n", id);
#pragma omp barrier
    if ( id == 0 ) {
    nthreads = omp_get_num_threads();
    printf("There are %d threads\n",nthreads);
 } }
return 0; }
```