

CS432 Parallel Computing

By
Rajeev Wankar

wankarcs@uohyd.ac.in

It is a 4 credit course: so we
will meet 4 hrs. in a week
(video lectures on YouTube for later
viewing after a week)

What to learn?

- **Design Paradigms of Parallel Algorithms**
- **Parallel Architectures**
- **Distributed Memory Computing**
- **Shared Memory Computing**
- **Software Multi-core Programming**
- **Programming GPGPUs using CUDA (earlier known as Compute Unified Device Architecture) and OpenACC**

What to know?

- **Syllabus**
 - Download from the course web site
 - Will be flexible

What to know?

- Books
- Quinn, M. J. (2003), *Parallel Programming in C with MPI and OpenMP*, First ed., McGraw-Hill Education.
- Bary Wilkinson and Michael Allen (1999), *Parallel Programming Techniques using networked of workstations and Parallel Computers*, First edition, Pearson.
- W. Gropp, E. Lusk, N. Doss, A. Skjellum(1996), *A high performance portable implementation of the message passing Interface (MPI) standard*, Parallel Computing 22 (6).
- Gibbons, A., W. Rytter (1989), *Efficient Parallel Algorithms*, first Edition Cambridge Uni. Press.
- Thomas Rauber, Gudula Rünger(2023), *Parallel Programming for Multicore and Cluster Systems*, Springer.

What to know?

- **Books**
- Ananth Grama et al. (2004), *An Introduction to Parallel Computing: Design and Analysis of Algorithms*, Second edition, Pearson Education India.
- David B. Kirk, Wen-mei W. Hwu (2010), *Programming Massively Parallel Processors: A Hands-on Approach*, First Edition, Morgan Kaufmann (This book is only on NVIDIA GPUs and CUDA programming despite its title)
- Jason Sanders and Edwards Kandrot (2011), *CUDA by Example: An Introduction to General-Purpose GPU Programming*, First Edition, Addison-Wesley.
- Shane Cook, Morgan Kaufmann (2012), *CUDA Programming A Developer's Guide to Parallel Computing with GPUs*, First Edition, Morgan Kaufmann.

What to know?

- **Course Material**
 - <https://studio.learnx.uohyd.ac.in/course/course-v1:UoH+CS751+2020-21>
 - <https://scislearn2.uohyd.ac.in/moodle/login/index.php>
 - <http://scis.uohyd.ac.in/~wankarcs>
 - <https://rajeevwankar.wixsite.com/mysite>

**Why do we need
powerful computers?**

Some Challenging Computations

- **Science**
 - Global climate modeling
 - Astrophysical modeling
 - Biology: genomics; protein folding; drug design
 - Computational Chemistry
 - Computational Material Sciences
- **Engineering**
 - Crash simulation
 - Semiconductor design
 - Earthquake and structural modeling
 - Computation fluid dynamics (airplane design)
 - Combustion (engine design)
- **Business**
 - Financial and economic modeling
 - Transaction processing, web services and search engines
- **Defense**
 - Nuclear weapons -- test by simulation
 - Cryptography

Some New Challenges

- **Artificial Intelligence & Data Science**
 - Training large AI/ML models (e.g., GPT, LLMs require supercomputers).
 - Deep learning in healthcare (medical image diagnostics, precision medicine).
 - Reinforcement learning for robotics & autonomous vehicles.
- **Health & Life Sciences**
 - Epidemiological simulations (COVID-19, disease spread at global scale).
 - Personalized/precision medicine (genome-to-drug pipeline).
 - Brain simulations (Human Brain Project, Blue Brain).
- **Security & Defense (new frontiers)**
 - Cybersecurity simulations (network attack/defense at scale).
 - Quantum-safe cryptography testing.
 - Blockchain optimization & cryptographic simulations.
- **Frontier Computing**
 - Quantum computing simulation on HPC (until real QC scales up).
 - Neuromorphic computing integration.

Modeling

- **Definition:** Modeling is the process of **creating an abstract representation** (a *model*) of a real-world system, object, or process.
- A **model** captures the essential features, rules, and relationships of the system but leaves out unnecessary details.
- Models can be **mathematical, physical, or computational**.
- **Example:**
- Writing down the equations of motion of a pendulum → that's a **model** of how it behaves.
- Designing a set of chemical equations to describe a reaction → a **model** of chemistry.

Simulation

- **Definition:** Simulation is the process of running experiments on the model to study its behavior under different conditions.
- It means using the model dynamically to imitate the operation of the real system over time.
- **Example:**
- Using a computer program to calculate and visualize the swing of the pendulum over 10 seconds → that's a simulation.
- Running a weather forecast on a climate model → simulation of weather.

Simulation

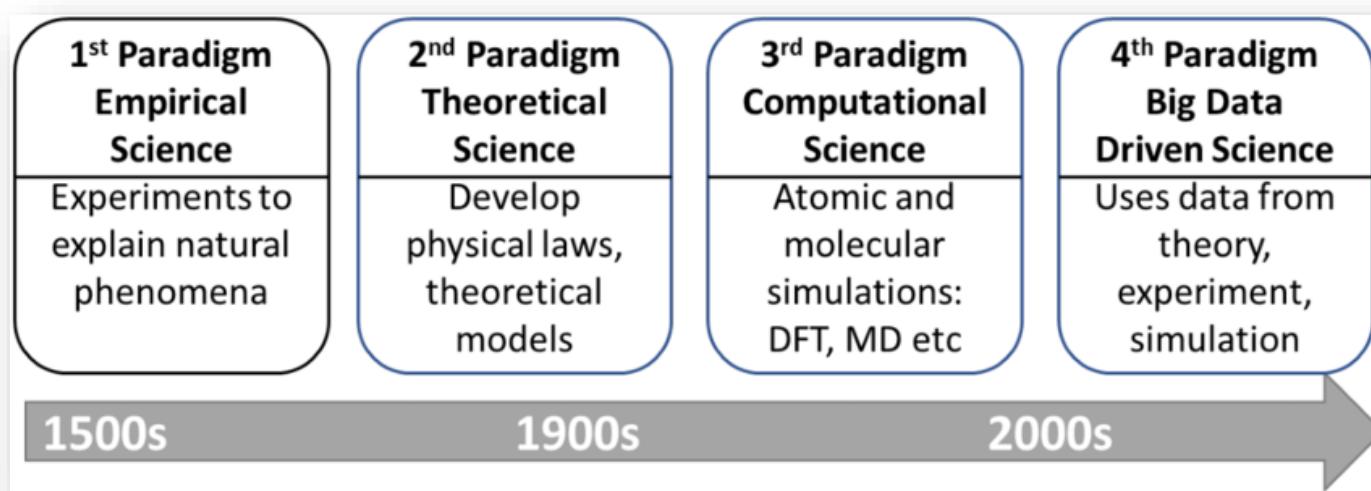
- Traditional scientific and engineering paradigm:
 - 1) Do theory or paper design.
(First pillar of Science)
 - 2) Perform experiments or build system.
(Second pillar of Science)
- Limitations:
 - Too difficult -- build large wind tunnels.
 - Too expensive -- build an experimental passenger jet.
 - Too slow -- wait for climate evolution.
 - Too dangerous -- weapons, drug design experiments.

Simulation

- Computational science paradigm: (Third pillar of Science)
 - 3) Use high performance computer systems to simulate the phenomenon.
 - Based on known physical laws and efficient numerical methods.

beyond

- What is **forth pillar** of Science?
- **Wisdom** is the ability to know what is true or right, common sense or the collection of one's knowledge.

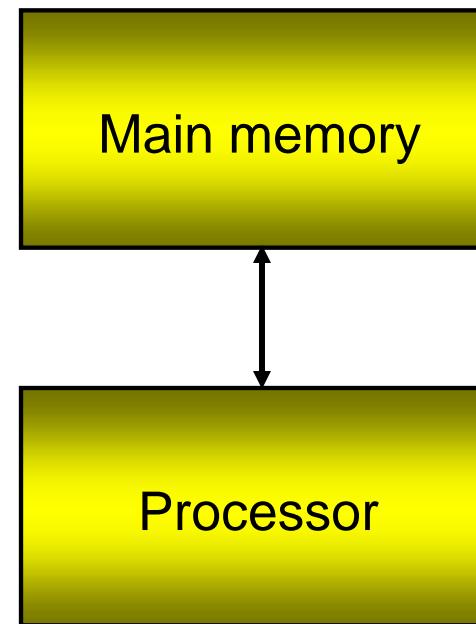


Hey, A. J. G., Tansley, S., & Tolle, K. M. (2009). The fourth paradigm: data-intensive scientific discovery. Redmond, WA: Microsoft Research.

Conventional Computer

Consists of a processor executing a program stored in a (main) memory:

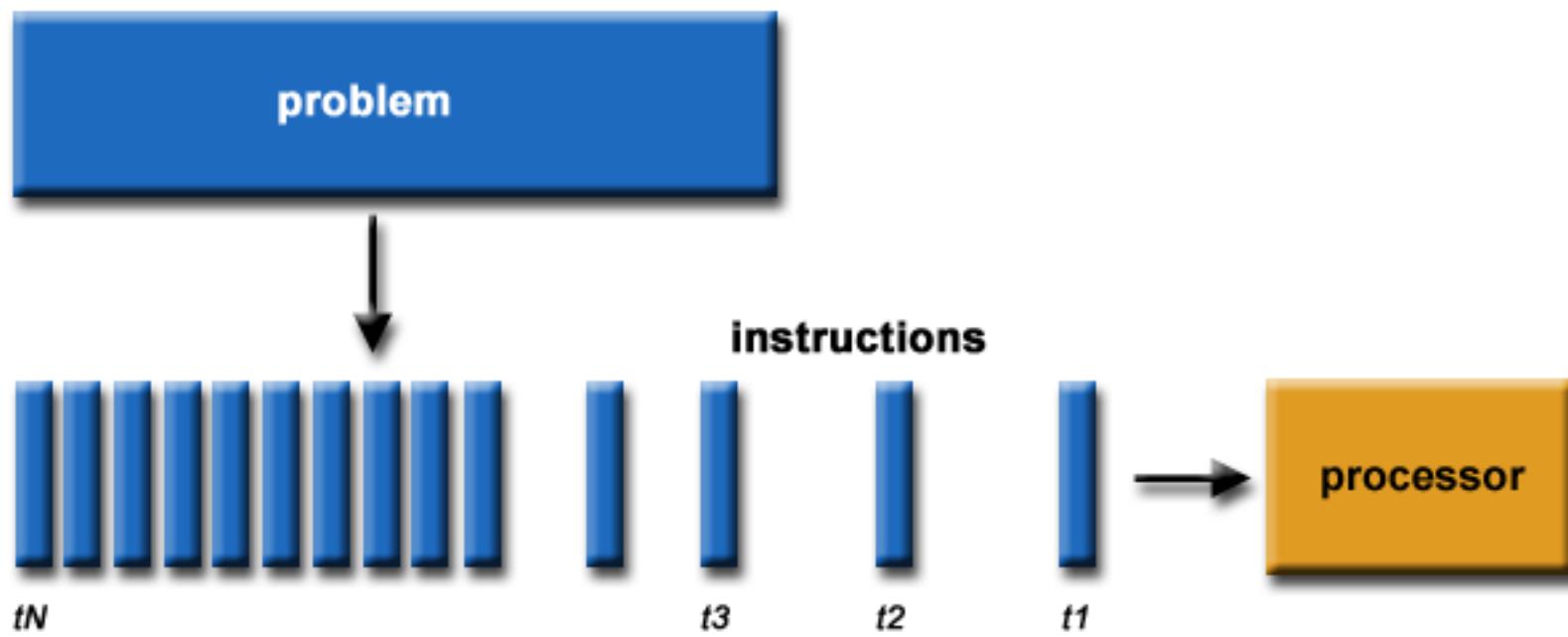
- All data and programs are stored in **one memory unit**.
- The processor fetches data from memory, works on it, and puts results back.

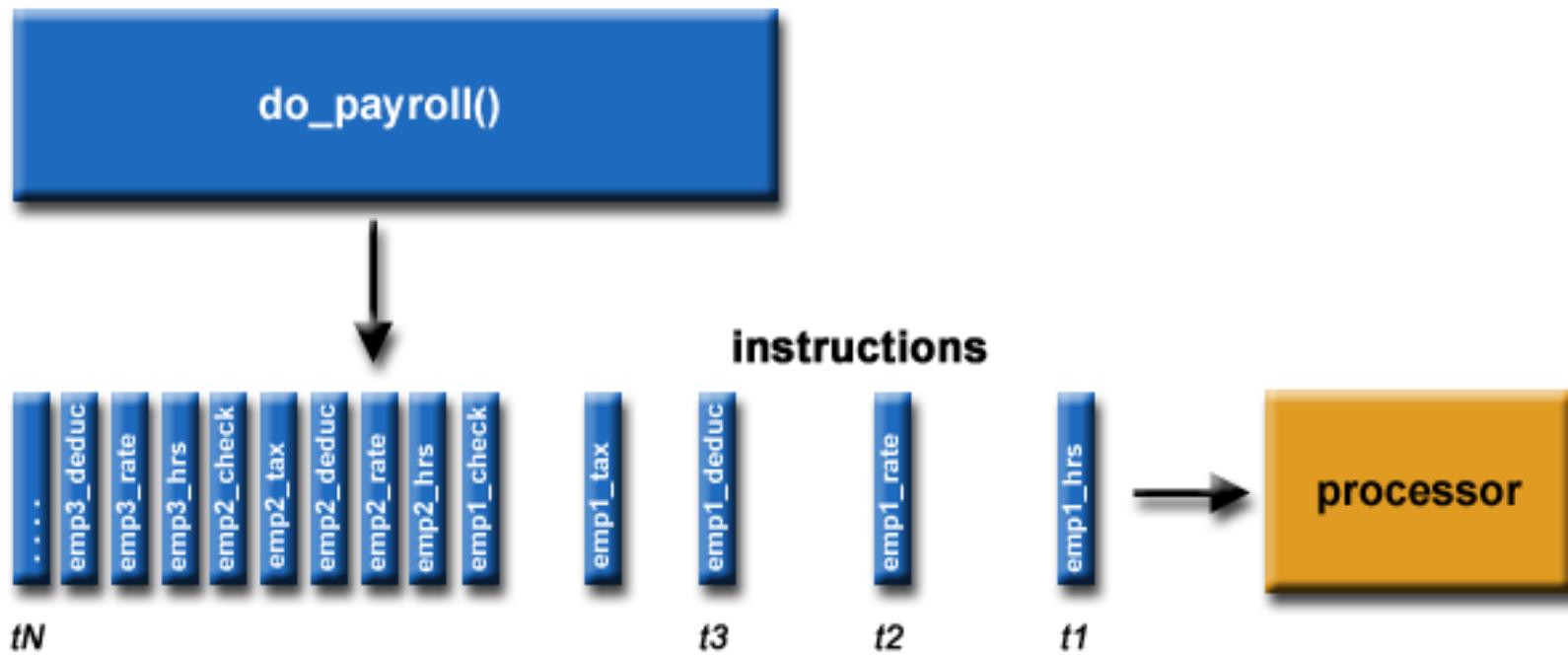


“Single processor + Single memory = Single memory space (step-by-step execution)”

Serial Computing

- Traditionally, software has been written for *serial* computation:
 - A problem is broken into a discrete series of instructions
 - Instructions are executed sequentially one after another
 - Executed on a single processor
 - Only one instruction may execute at any moment in time





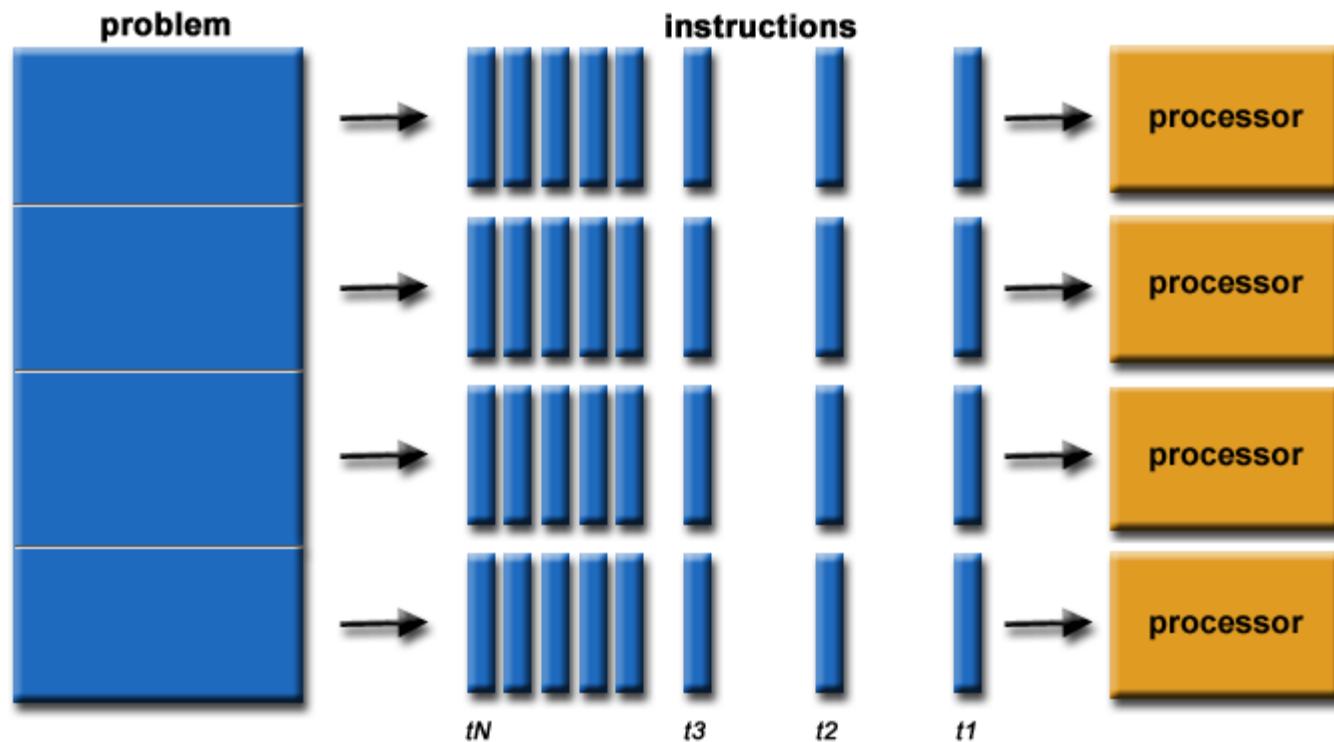
High Performance Computing (HPC)

- *Traditionally*, achieved by using the multiple computers together - parallel computing.
- Simple idea! -- Using multiple computers (or processors) simultaneously should be able to solve the problem faster than a single computer.

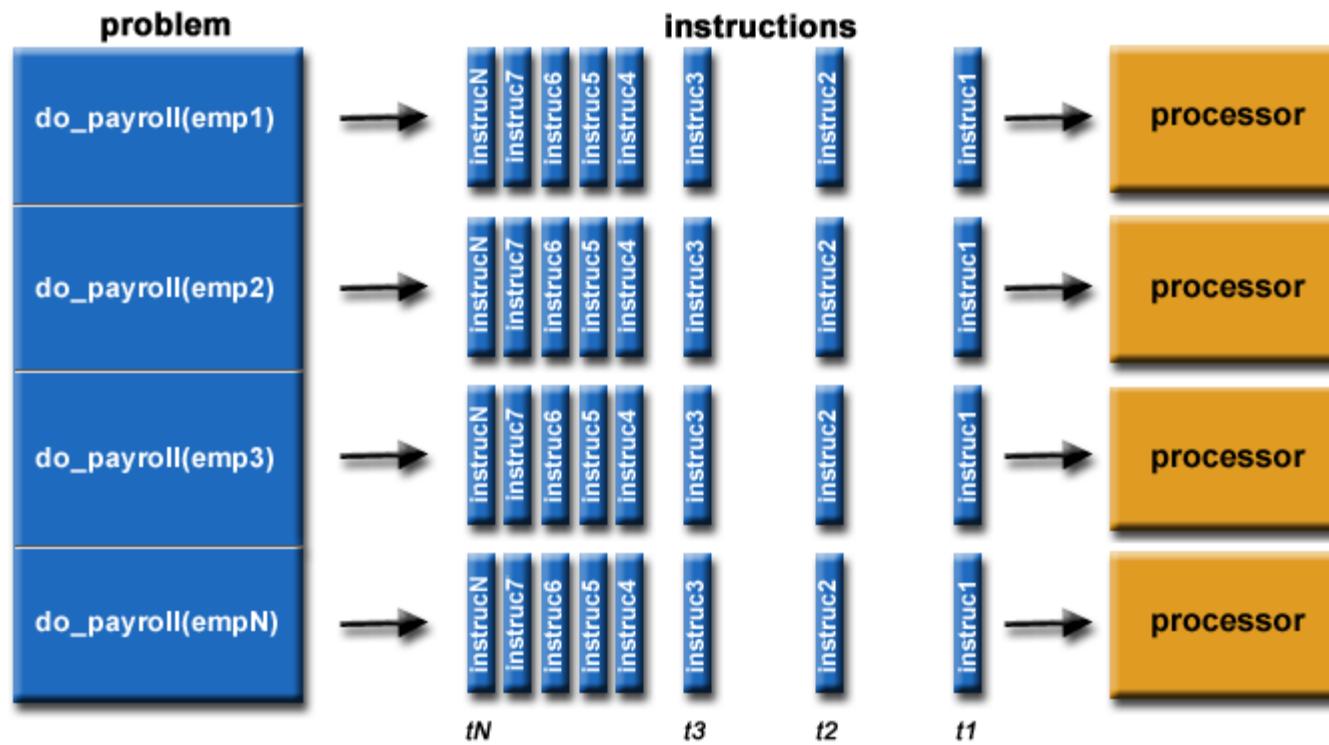
Using multiple computers or processors

- Key concept - dividing problem into parts that can be computed simultaneously.
- Parallel programming – solution of a problem using multiple processes or multi computers.
- Concept very old (65 years).

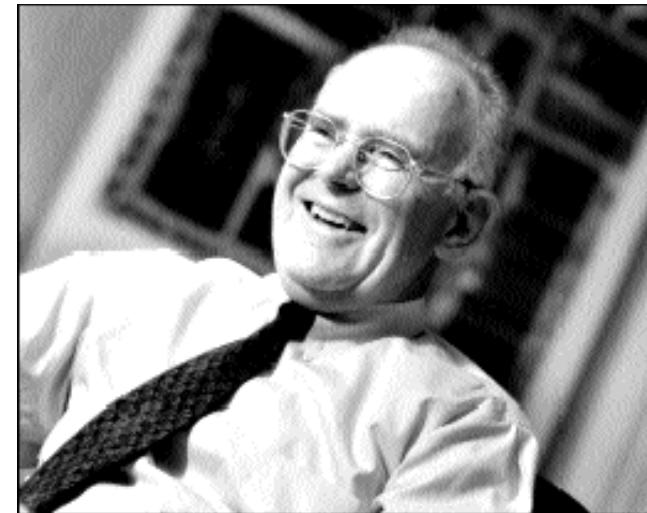
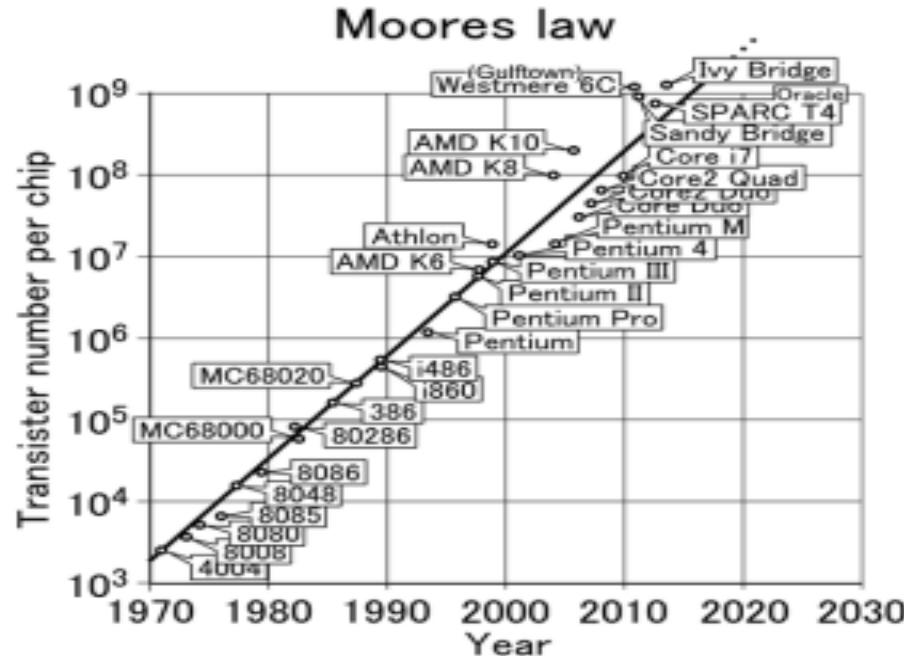
Parallel Processing



Parallel Processing



Technology Trends: Microprocessor Capacity



Moore's Law:

#transistors/chip doubles
every 18 months

Microprocessors have
become smaller, denser,
and more powerful.

Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

Slide source: Jack Dongarra

High Performance Computing

- Long History:
 - Multiprocessor system of various types (1950's onwards)
 - Supercomputers (1960s-80's)
 - Cluster computing (1990's)
 - Grid computing (2000's)
 - Multi-core architecture (2003 onwards)
 - **(brings supercomputer on a desk)**
 - GPUs, GPGPUs

High Performance Computing

Practical Parallel Programming and Machines:

- 1960's ILLIAC IV, Solomon
- 1970's, C.mmp, Cm*
- 1980's Cosmic tube
- 1990's Beowulf Cluster

ILLIAC IV (1960)



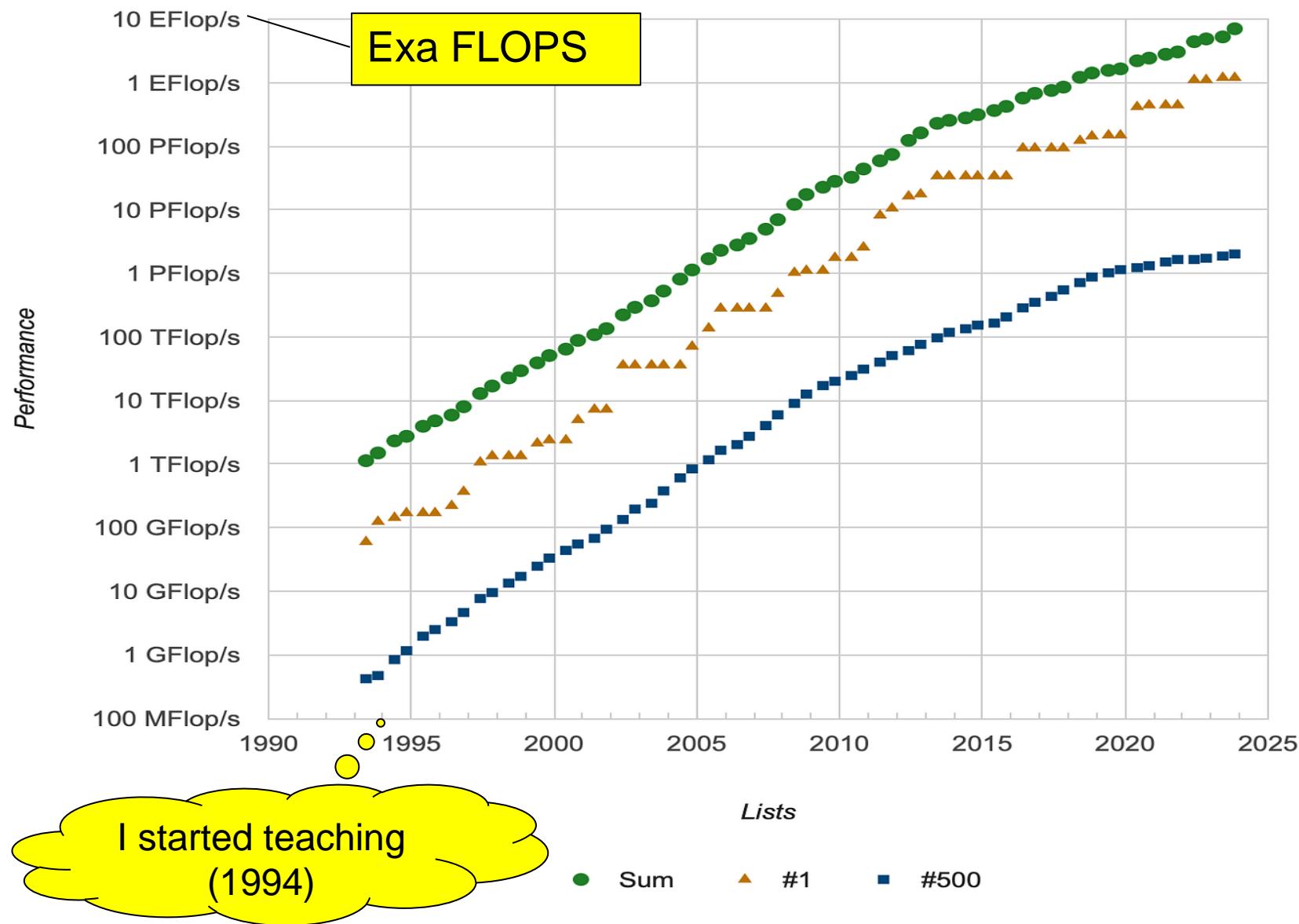
The performance of the single proc. can be improved

- **Architectural** (amt. of work performed per instruction cycle) bit parallel memory, bit parallel arithmetic, cache, instruction pipelining, multiple functional units etc.
- **Tech. advances** (reduce time needed per instruction cycle). Speed of the electronic device is limited by the speed of the light (It travels appox. a foot in a nanosec.)

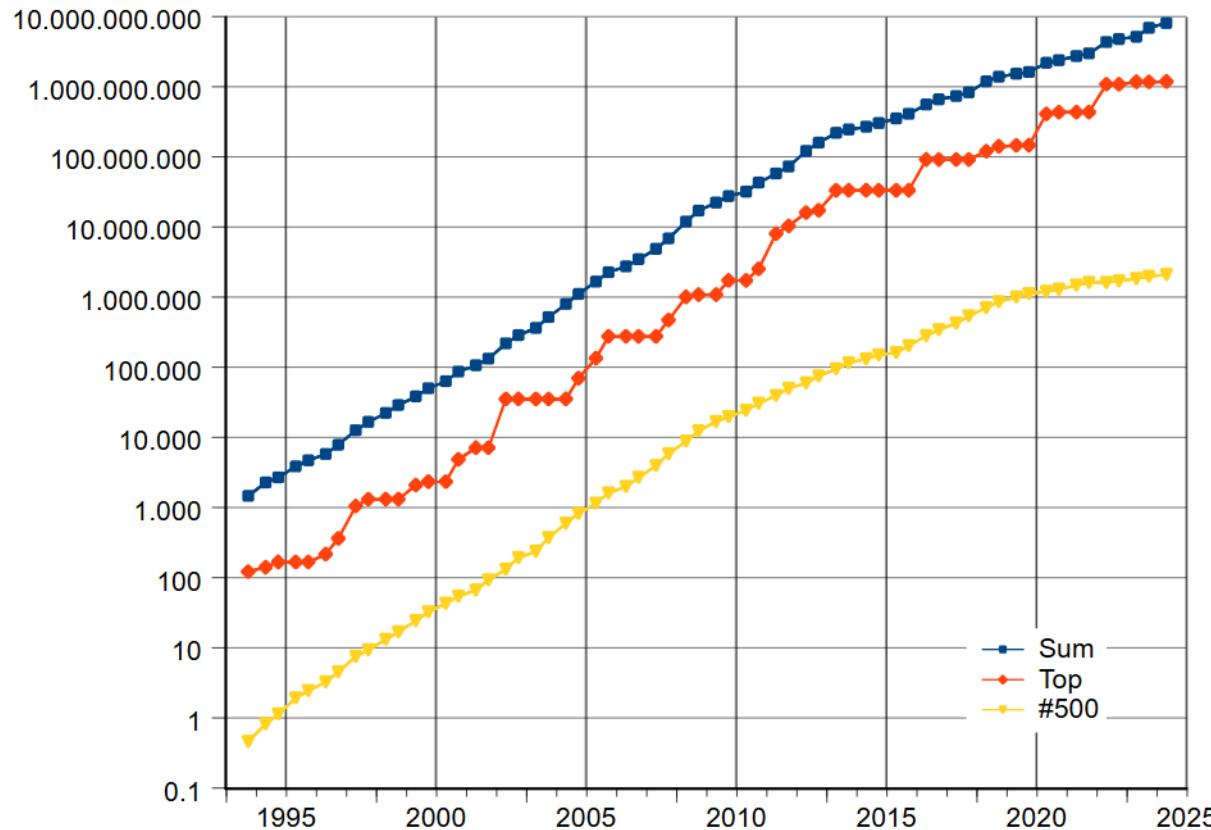
“Automatic” Parallelism in Modern Machines

- Bit level parallelism
 - within floating point operations, etc.
- Instruction level parallelism
 - multiple instructions execute per clock cycle
- Memory system parallelism
 - overlap of memory operations with computation
- OS parallelism
 - Multiple threads run in parallel on SMPs

There are limits to all of these -- for very high performance, user must identify, schedule and coordinate parallel tasks



Courtesy:
<https://commons.wikimedia.org/w/index.php?curid=33540287>



Rapid growth of supercomputer performance, based on data from top500.org site. The logarithmic y-axis shows performance in **GFLOPS**. **Blue** combined performance of 500 largest supercomputers **red** Fastest supercomputer **yellow** Supercomputer in 500th place

Top Supercomputer

- According to the TOP500 list, the world's fastest supercomputer is **EL CAPITAN** - HPE CRAY EX255A, AMD 4TH GEN EPYC 24C 1.8GHZ, AMD INSTINCT MI300A, SLINGSHOT-11, TOSS (TRI-LAB'S OS STACK)
- As of Nov 2024 EL CAPITAN was the world's fastest supercomputer using AMD CPUs and GPUs.
- It achieved an Rmax of 1.742 exaFLOPS (1,742.00 peta flops on LINPACK) using 11,039,616 cores

Units of Measure in HPC

- High Performance Computing (HPC) units are:
 - Flops: floating point operations
 - Flop/s: floating point operations per second

Units of Measure in HPC

Mega	$\text{Mflop/s} = 10^6 \text{ flop/sec}$	$\text{Mbyte} = 10^6 \text{ byte}$ (also $2^{20} = 1048576$)
Giga	$\text{Gflop/s} = 10^9 \text{ flop/sec}$	$\text{Gbyte} = 10^9 \text{ byte}$ (also $2^{30} = 1073741824$)
Tera	$\text{Tflop/s} = 10^{12} \text{ flop/sec}$	$\text{Tbyte} = 10^{12} \text{ byte}$ (also $2^{40} = 10995211627776$)
Peta	$\text{Pflop/s} = 10^{15} \text{ flop/sec}$	$\text{Pbyte} = 10^{15} \text{ byte}$ (also $2^{50} = 1125899906842624$)
Exa	$\text{Eflop/s} = 10^{18} \text{ flop/sec}$	$\text{Ebyte} = 10^{18} \text{ byte } (2^{60})$ (also $2^{60} = 115292150\ 4606846976$)
Zetta	$\text{Eflop/s} = 10^{21} \text{ flop/sec}$	$\text{Zbyte} = 10^{21} \text{ byte } (2^{70})$ (also $2^{70} = 1180591620717411303424$)
Yotta	$\text{Eflop/s} = 10^{24} \text{ flop/sec}$	$\text{Ybyte} = 10^{24} \text{ byte } (2^{80})$ (also $2^{80} = 1208925819614629174706176$)

Parallel Processing Terminology

Parallel Processing is the information processing that emphasizes on the concurrent manipulation of data elements belonging to one or more processes solving a single problem.

A multiple processor computer capable of **parallel processing** is a parallel computer.

A general purpose computer capable of solving the individual problem at a very high speed is a **Supercomputer**.

Throughput of the device is the number of results it produces per unit time.

Parallel Processing Terminology

- ❑ *Parallelism* is a condition that arises when at least two processes are executing simultaneously
- ❑ *Concurrency* is a condition that exists when at least two processes are making progress.

Parallel Processing Terminology

- *Concurrency is a more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.*
- *In a multithreaded process on a single processor, the processor can switch execution resources between threads, resulting in concurrent execution.*
- *In the same multithreaded process in a shared-memory multiprocessor environment, each thread in the process can run on a separate processor at the same time, resulting in parallel execution.*

Parallel Processing Terminology

Control parallelism: Applying different operations to different data elements simultaneously. **Pipelining** is a special case of it in which the computation is divided in to segments or stages.

Ex. Assembly line of any car manufacturing process.

Data parallelism: Multiple functional units apply same operation simultaneously to elements of a data set.

Ex. Matrix addition.

Sieve of Eratosthenes (240 BC)

- Sequential Approach
- Control Parallel Approach
- Data Parallel Approach

Composite Numbers: Any whole number having more than two factors

Prime Numbers: A number that has only two factors 1 and itself

Sieve of Eratosthenes (240 BC)

Input: an integer $n > 1$

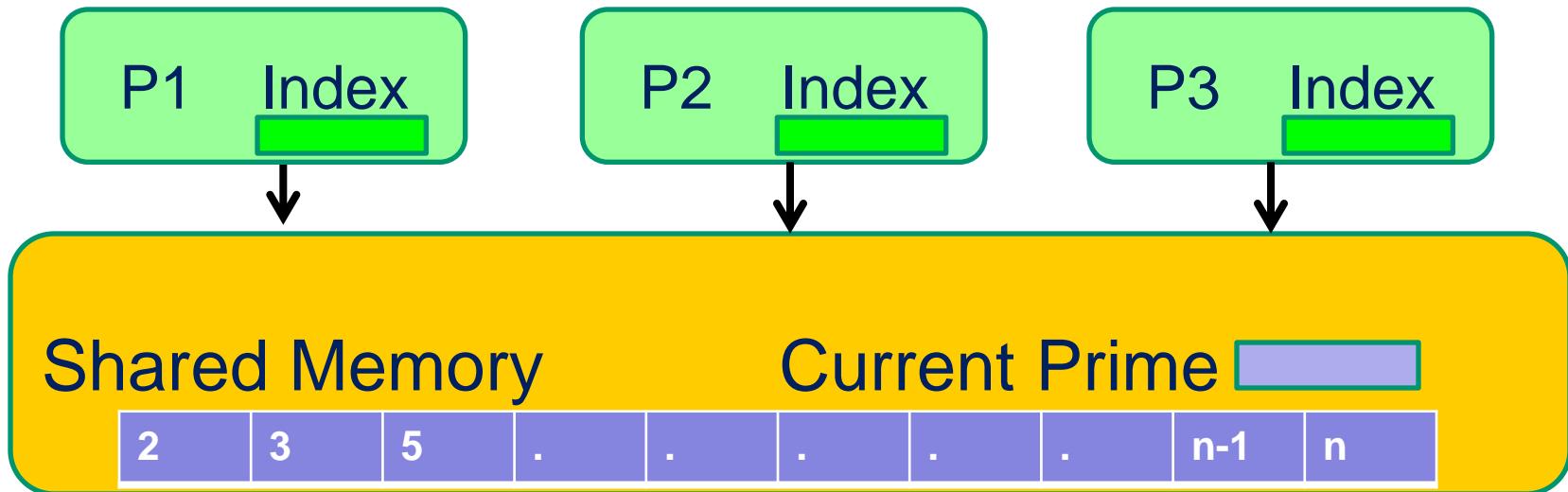
```
//Let A be an array of Boolean values,  
//indexed by integers 2 to n,  
//initially all set to true.
```

```
for i = 2, 3, 4, ..., not exceeding  $\sqrt{n}$ :  
    if A[i] is true:  
        for j =  $i^2$ ,  $i^2+i$ ,  $i^2+2i$ , ..., not exceeding n :  
            A[j] := false
```

Output: all i such that $A[i]$ is **true**.

	2	3	4	5	6	7	8	9	10	Prime numbers
11	12	13	14	15	16	17	18	19	20	
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	
41	42	43	44	45	46	47	48	49	50	
51	52	53	54	55	56	57	58	59	60	
61	62	63	64	65	66	67	68	69	70	
71	72	73	74	75	76	77	78	79	80	
81	82	83	84	85	86	87	88	89	90	
91	92	93	94	95	96	97	98	99	100	
101	102	103	104	105	106	107	108	109	110	
111	112	113	114	115	116	117	118	119	120	

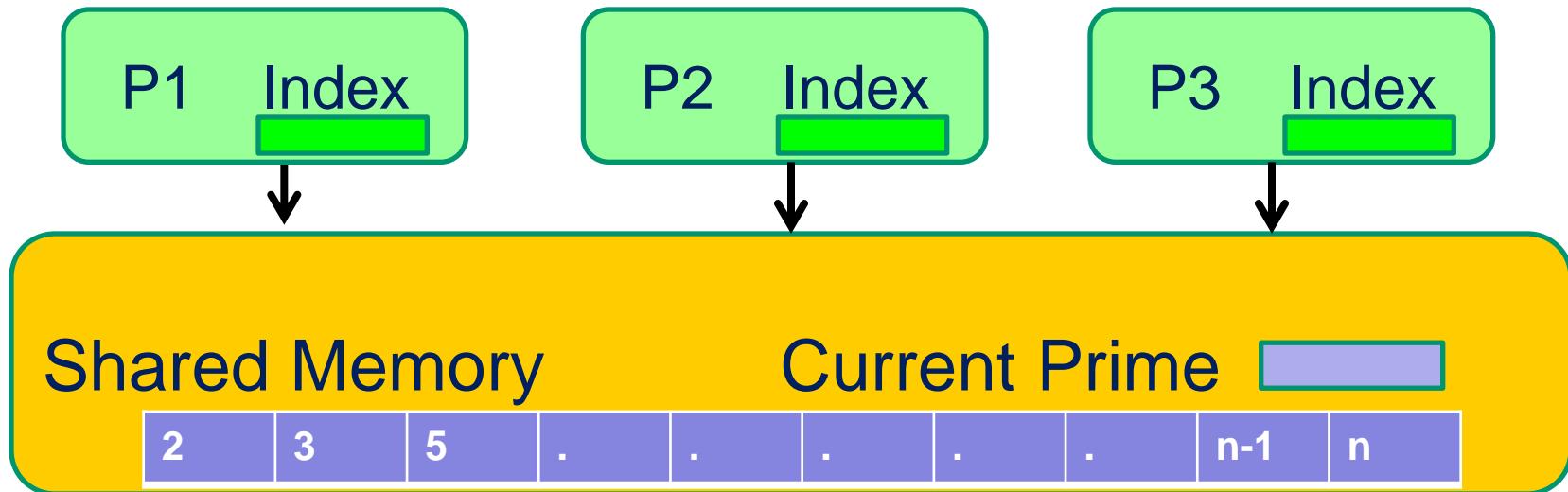
Figure curtesy: <http://en.wikipedia.org>



Control Parallel Approach:

- Two step process, striking from the list **multiple of that prime(not itself)**, beginning from its square
- The processors continue **until** a prime is found whose value is **greater than \sqrt{n}**

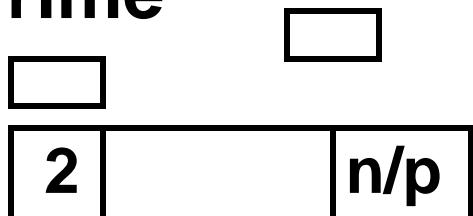
Things to observe:



1. Each processor has its own **private loop index** and remaining variables are shared with all processors.
2. Two processors may start with the **same prime** to sieve.
3. Instead of taking next prime they can choose the **composite number**

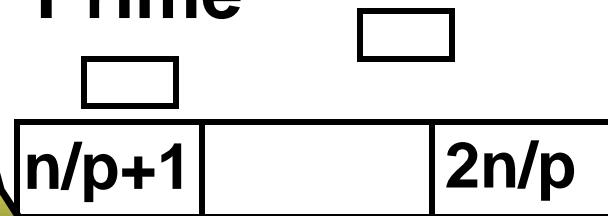
P1

Current Index
Prime



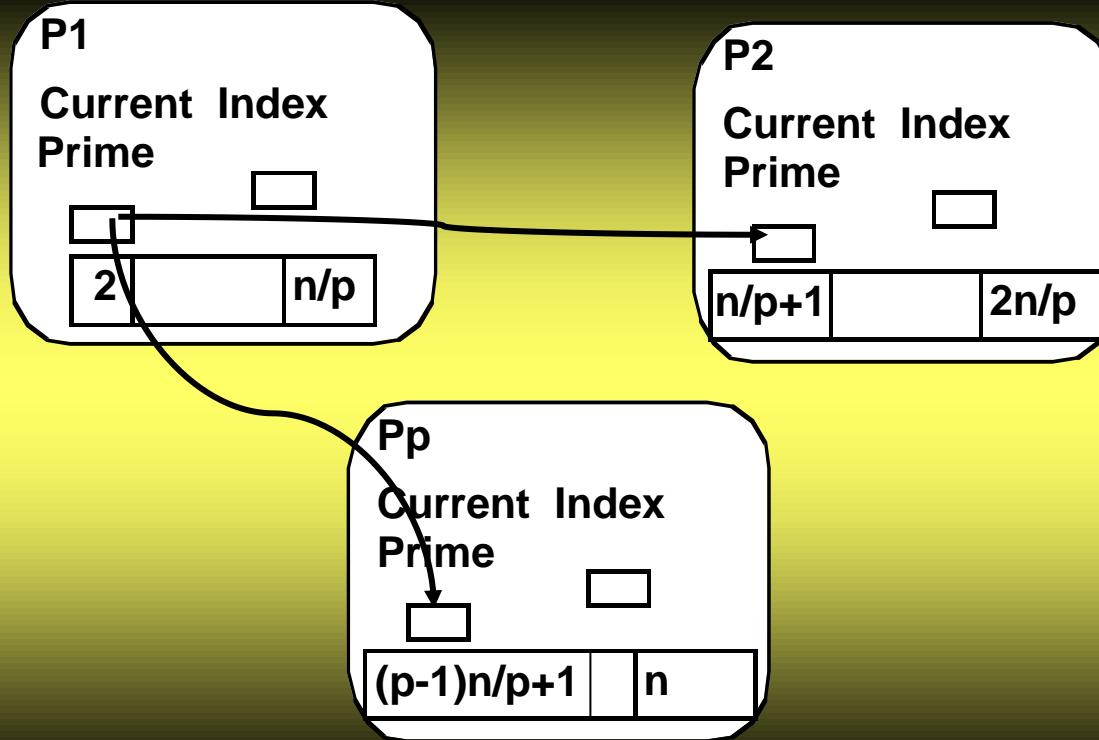
P2

Current Index
Prime



Data Parallel Approach

Every processor will be responsible for striking a segment of the array representing the natural numbers



1. Each processor has its own copy of the variables containing the current prime and the loop index.
2. P1 finds prime and **communicates them to other processors using message passing**
3. Each processor iterates through its own portion of the array.

Let us Compare Control and Data Parallelism

Control Parallel	Data Parallel
Associating data (same or different) to processes	Associating a Process to different data (analogous to SIMD)
Different operations are performed on the same or different data.	Same operations are performed on different subsets of same data.
Asynchronous computation	Synchronous computation
Speedup is limited by number of parallel processes	Speedup is more as there is only one execution process operating on all sets of data. (up to some critical no.)
Amount of parallelization is proportional to the number of independent tasks to be performed.	Amount of parallelization is proportional to the input data size.
Load balancing depends on the availability of the hardware and scheduling algorithms like static and dynamic scheduling.	Designed for optimum load balance

Performance Measures

Speedup

Measure of execution time T , when we have a problem of size N and P number of processors

Speedup is a number that measures the relative performance of two systems processing the same problem.

More technically, it is the improvement in speed of execution of **a task** executed on two similar architectures with **different resources**.

Speedup

Measure of execution time T , when we have a problem of size N and P number of processors

Speedup $S(N, P) = T(N, 1)/T(N, P)$,

often Speedup < P

Parallel Efficiency

Measure of execution time T , when we have a problem of size N and P number of processors

Parallel Efficiency is the ratio between speedup and number of Processors

Parallel Efficiency

Measure of execution time T, when we have a problem of size N and P number of processors

Parallel efficiency $PE(N, P) = S(N, P)/P$,

often Parallel Efficiency < 1 (max value is 1, can be expressed as % also)

Serial efficiency

Measure of execution time T , when we have a problem of size N and P number of processors

Serial Efficiency is the ratio between the time needed for the most efficient sequential algorithm to perform a computation and the time needed to perform the same computation on a machine incorporating data/control computation using a **single processor** (may be using the different algorithm).

Serial efficiency

Measure of execution time T, when we have a problem of size N and P number of processors

Serial Efficiency

$$SE(N) = T_{\text{Best}}(N)/T(N, 1),$$

often Serial Efficiency ≤ 1

Scalability

The **scalability** of a **parallel** algorithm is a measure of its capacity to effectively utilize an increasing number of processors, i.e. how its performance changes with increased number of processor.

Scalability

- Strong Scaling: ***Keeping the problem size fixed*** and increasing the processors
 - Aim is to: Minimize time to solution for a given problem
- Weak Scaling: ***Keeping the work per processor fixed*** and adding more processors (the overall problem size increases)
 - Aim is to: solve the larger problems

Strong scaling is more useful in Parallel Computing and more difficult to achieve

Speedup Factor

$$S(p) = \frac{\text{Execution time using one processor}}{\text{Execution time using multiprocessors with } p \text{ processors}} = \frac{t_s}{t_p}$$

t_s is execution time on a single processor

t_p is execution time on a multiprocessor.

$S(p)$ gives increase in speed by using multiprocessor.

Best sequential algorithm for single processor and *Parallel algorithm are usually different.*

Maximum speedup is usually p with p processors (**linear speedup**).

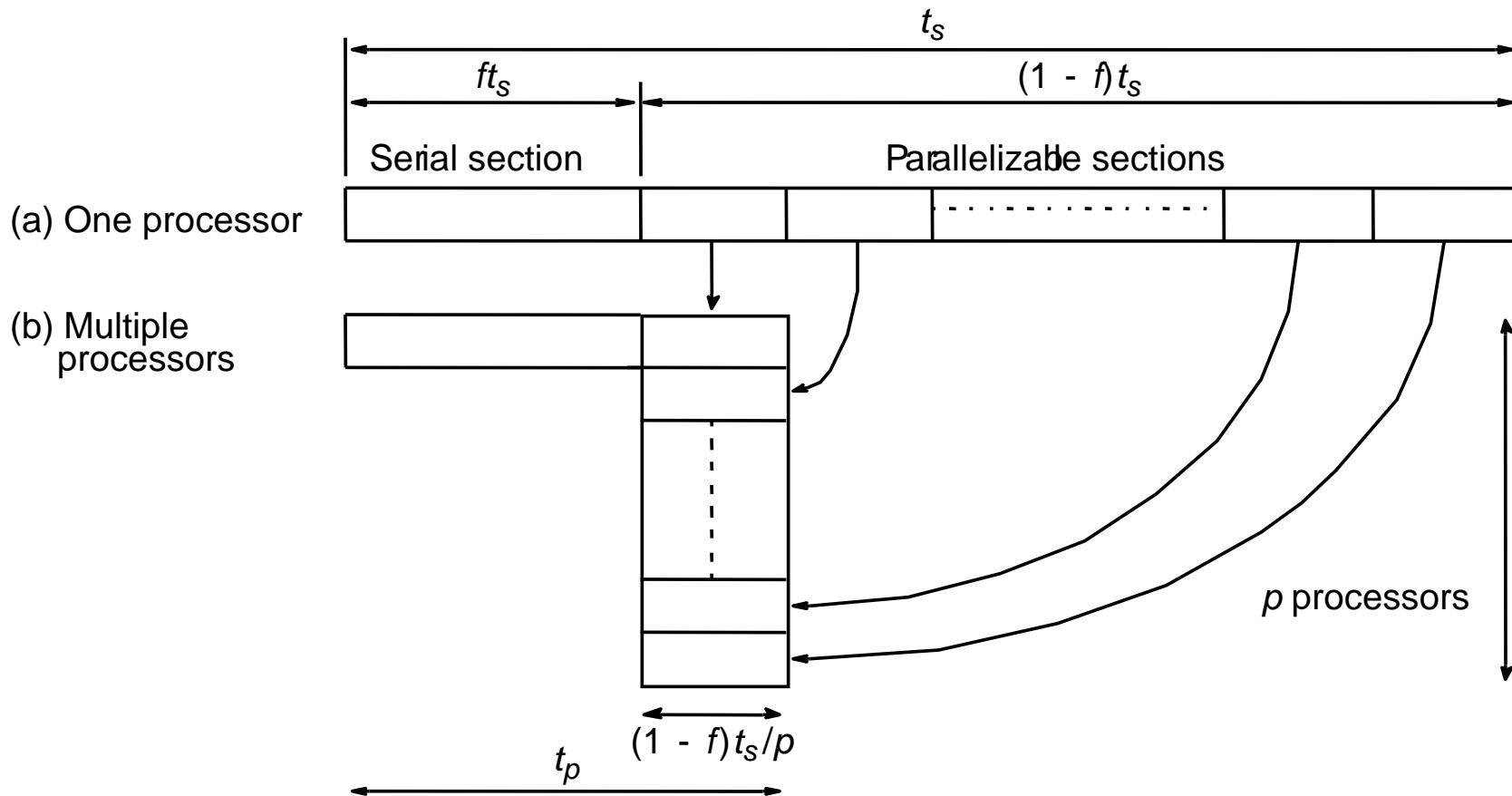
Amdhal's law: Let f be a fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$. The maximum speedup S achievable by a parallel computer with P processors performing the computation is

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

Gene Myron Amdahl (1967)

Maximum Speedup

Amdahl's law



Example 1

- If 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 8 CPUs?

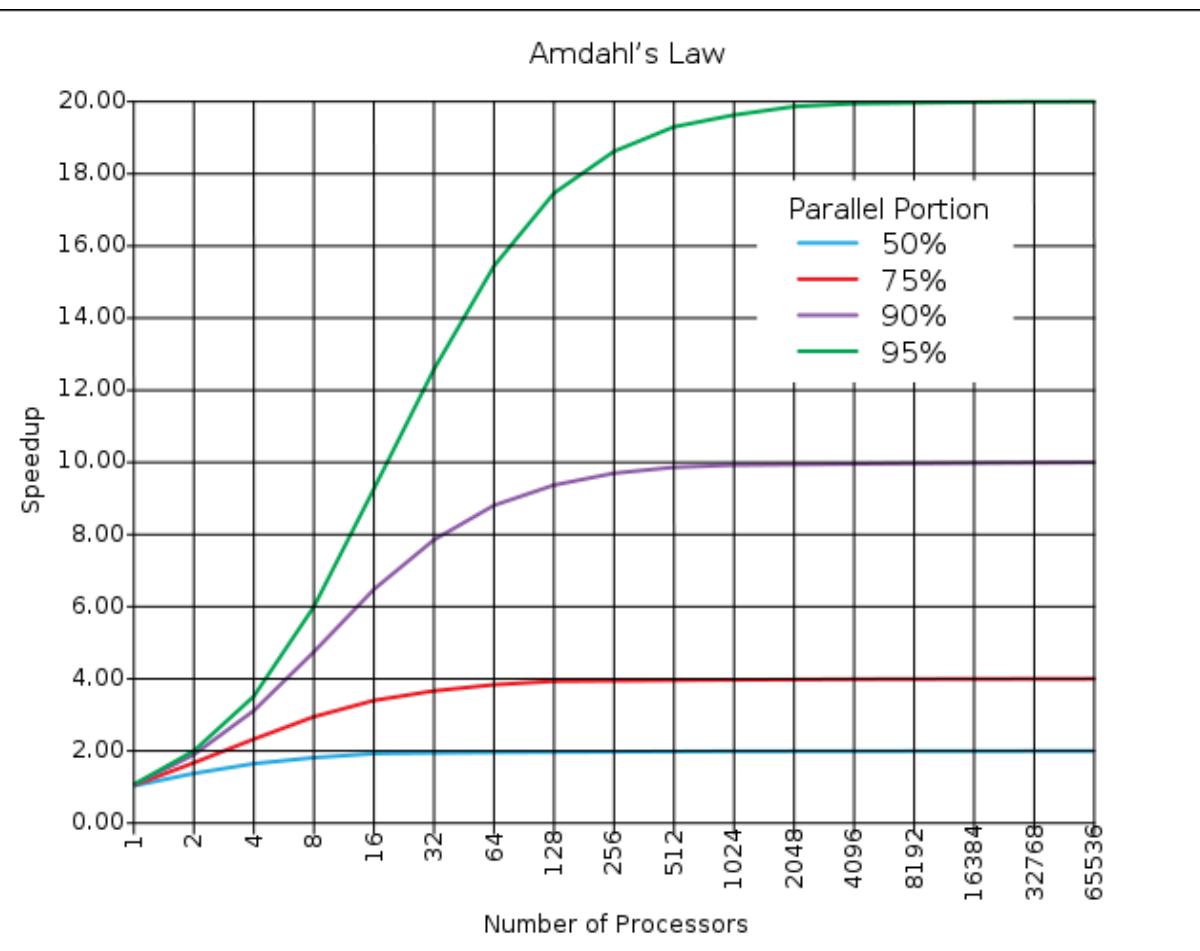
$$S \leq \frac{1}{0.05 + (1 - 0.05)/8} \cong 5.9$$

Example 1

- 5% of a parallel program's execution time is spent within **inherently sequential** code.
- The maximum speedup achievable by this program, regardless of how many PEs are used, is

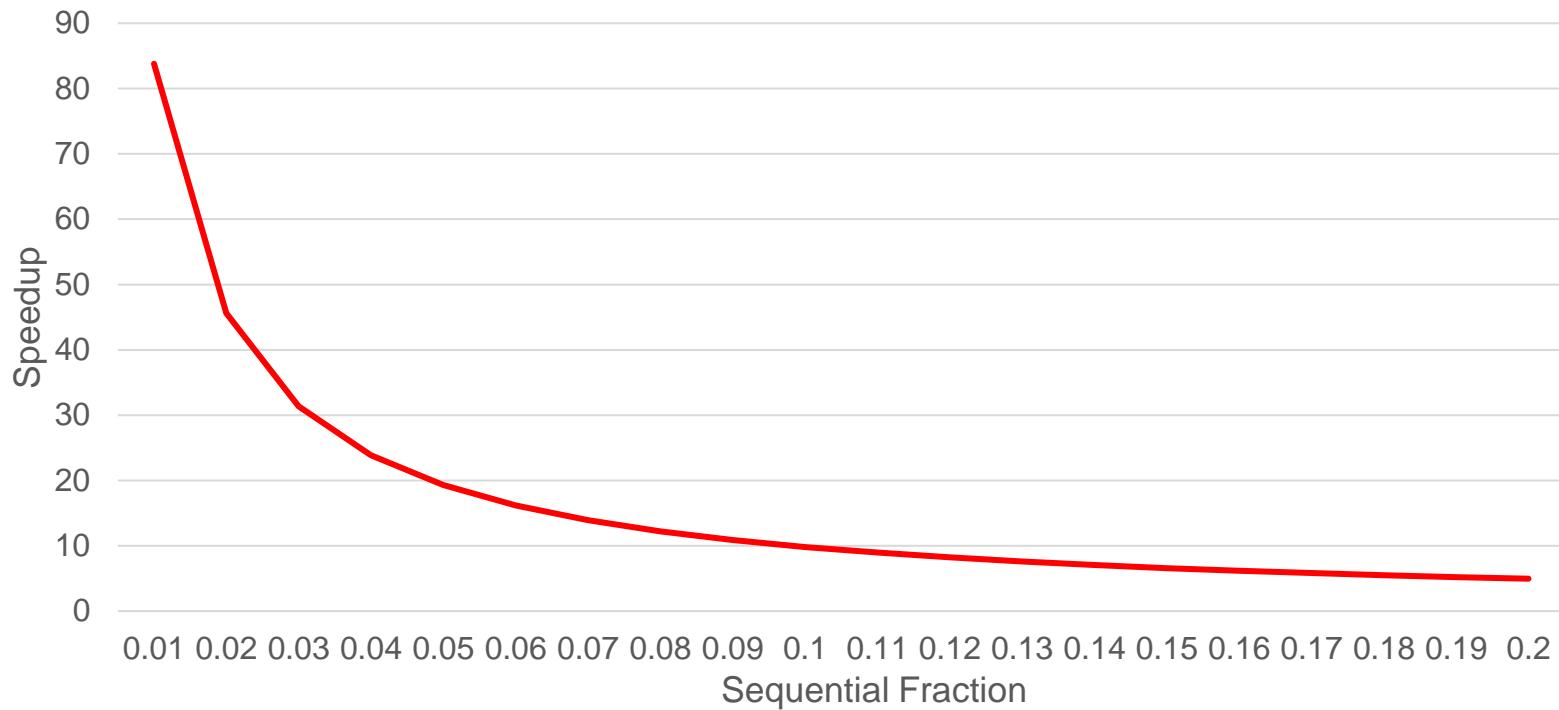
$$\lim_{p \rightarrow \infty} \frac{1}{0.05 + (1 - 0.05)/p} = \frac{1}{0.05} = 20$$

Example 1

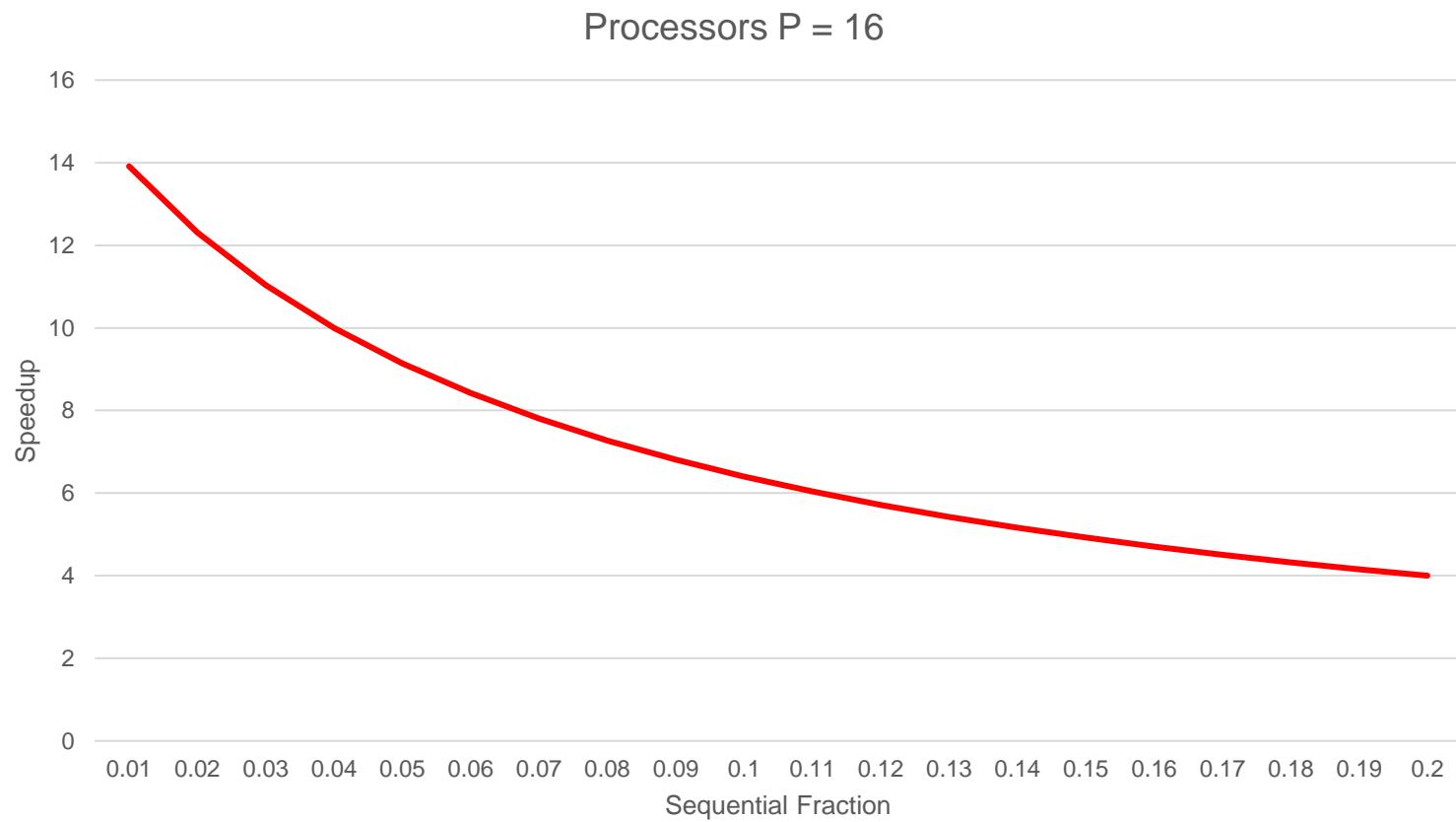


Example 2

Processors P = 512

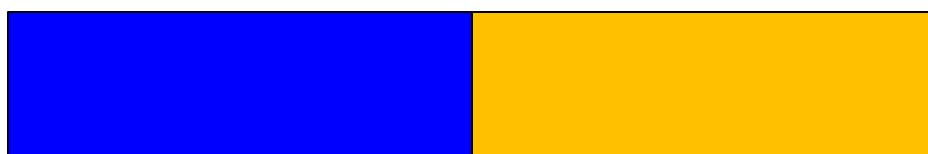


Example 3



Serial Portion

Parallel Portion



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

Processors	Speedup
1	1
2	1.33
4	1.6
8	1.8

If Amdahl's Law is applied

“The speedup is always limited by the sequential portion of the code”

Amdahl's Law Continue...

- Gustafson's law addresses the **shortcomings** of Amdahl's law, which cannot scale to match availability of computing power as the machine size increases.
- It removes the **fixed problem size or fixed computation load** constrains on the parallel processors: instead, he proposed a **fixed time** concept which leads to scaled speed up for larger problem sizes.

Gustafson's Law

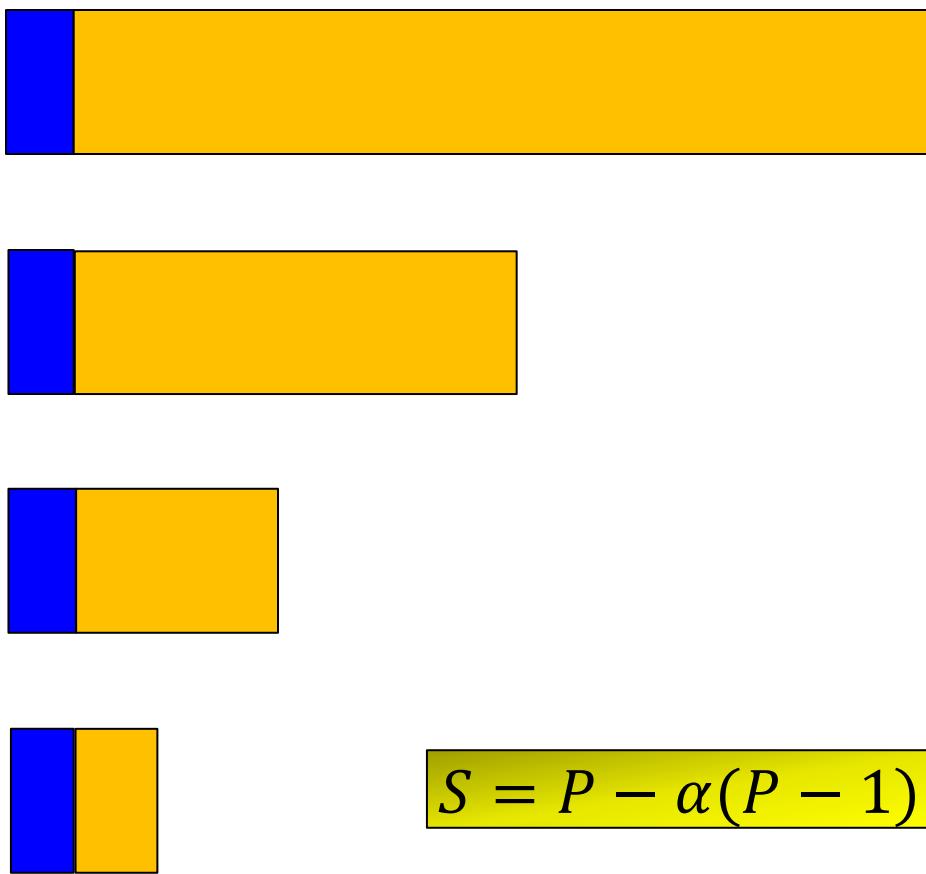
- **Gustafson's Law** (also known as **Gustafson-Barsis' law, 1988**) states that *any sufficiently large problem can be efficiently parallelized*. Gustafson's Law is closely related to Amdahl's law, which gives a limit to the degree to which a program can be sped up due to parallelization.

$$S = P - \alpha(P - 1)$$

where P is the number of processors, S is the speedup, and α the sequential part of the process.

We require larger problem for large processors

Even if it still limited by serial portion,
It is less important



Processors	Speedup
1	1
2	1.8
4	3.8
8	4.5

Amdahl's Law approximately suggests:

- Suppose a car is traveling between two cities 60 kilometer apart, and has already spent one hour traveling half the distance at 30 km/h.
- No matter how fast we drive the last half, it is impossible to achieve 90 km/h average before reaching the second city. Since it has already taken one hour and we only have a distance of 60 km total; going infinitely fast it would only achieve 60 km/h.

Gustafson's Law approximately states:

- Suppose a car has already been traveling for some time at less than 90 km/h. Given enough time and distance to travel, the car's average speed can always eventually reach 90 km/h, no matter how long or how slowly it has already traveled.
- For example, if the car spent one hour at 30 km/h, it could achieve this by driving at 120 km/h for two additional hours, or at 150 km/h for an hour, and so on.

Comparing Amdahl and Gustafson's Law

Amdahl's Law

- Assumes the fix execution time on a single Processor
- $f + (1-f) = 1$ that is normalized serial time
- Whole problem fits into the memory of serial computer
- Fixed Speedup =

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

Gustafson's Law

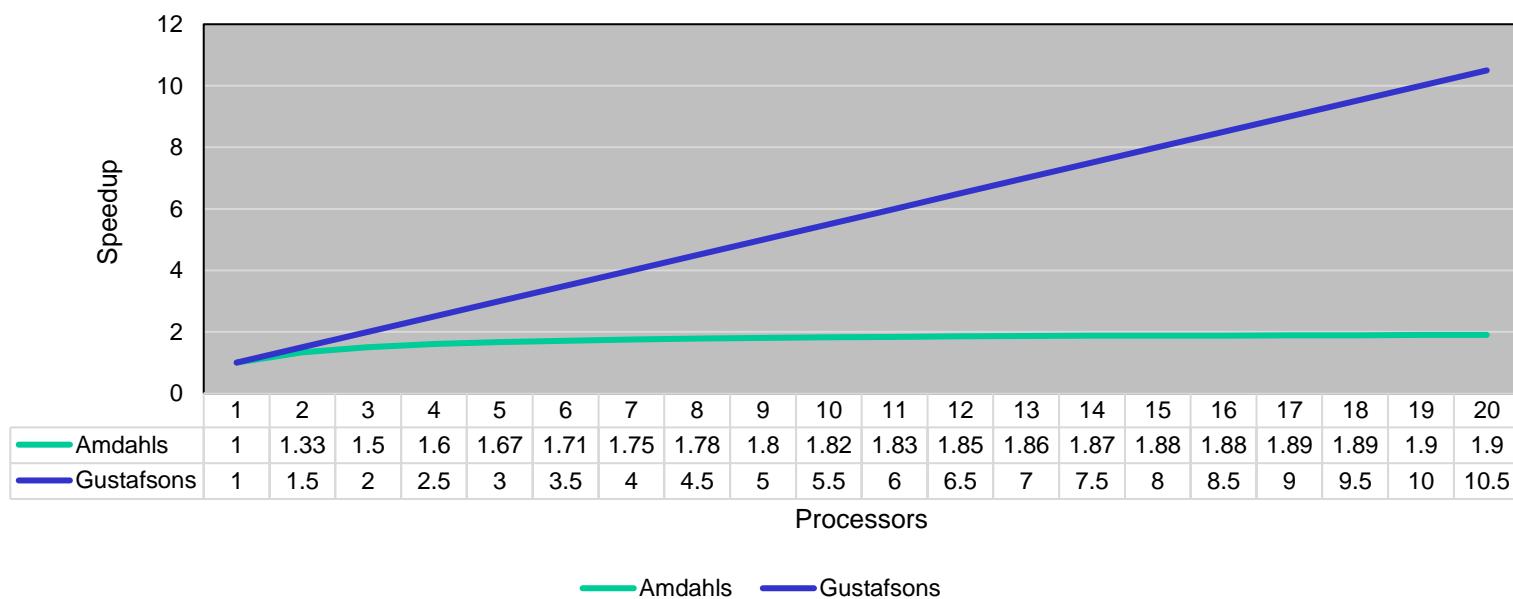
- Assumes the fix execution time on a parallel machine
- $f + (1-f) = 1$ that is normalized parallel time
- $f + (1-f)P$ is serial time on single processor
- Whole problem fits into the memory of a parallel computer
- Scaled Speedup =

$$S = P - \alpha(P - 1)$$

Comparing Amdahl and Gustafson's Law



Amdahls Vs Gustafson



- **Karp-Flatt metric:** Given a parallel computation exhibiting speedup S on p processors, where $p > 1$, experimentally determined serial fraction e is defined to be the Karp - Flatt Metric as

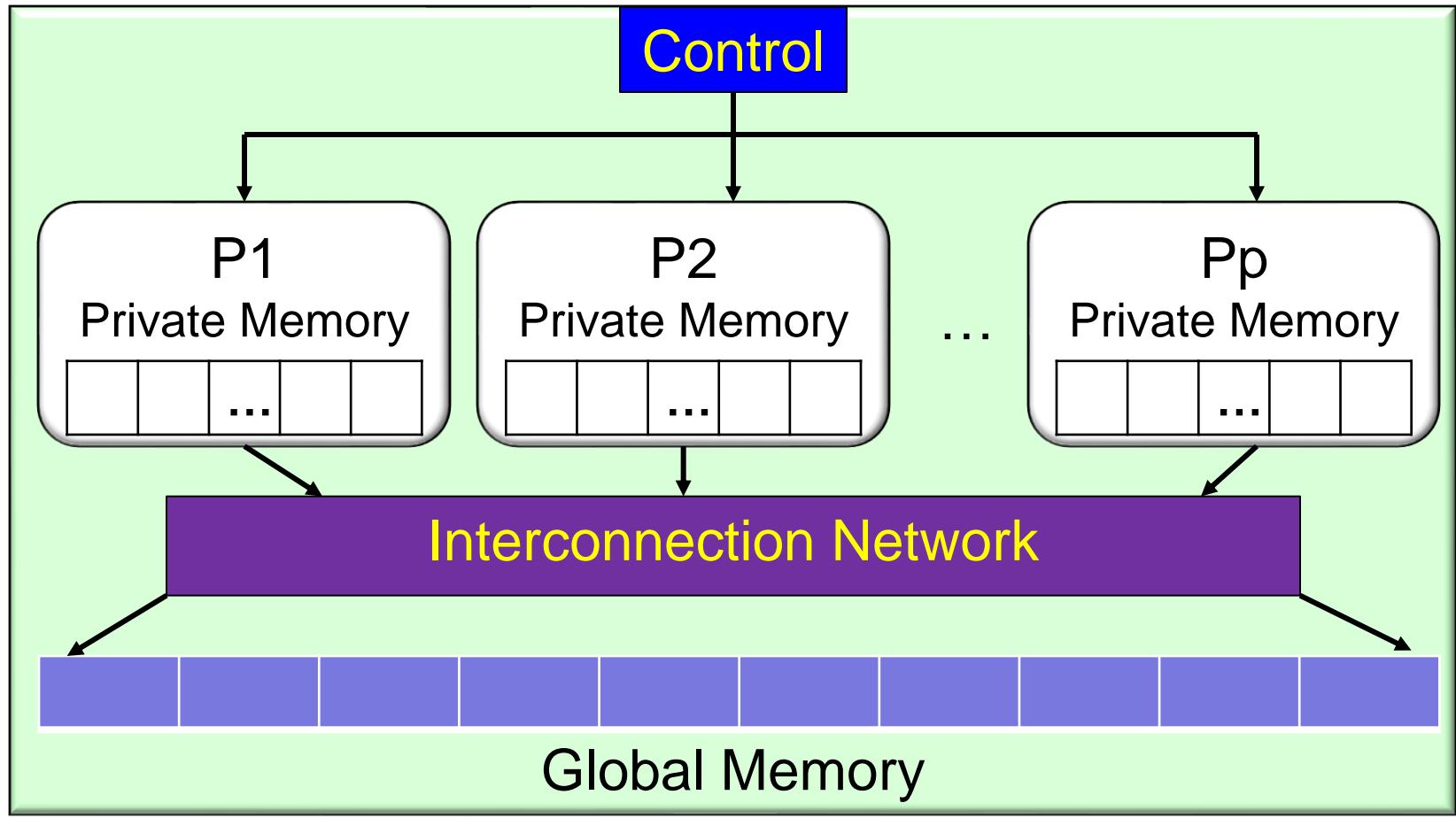
$$e = \frac{\frac{1}{S} - \frac{1}{p}}{1 - \frac{1}{p}}$$

- The less the value of e the better the parallelization.

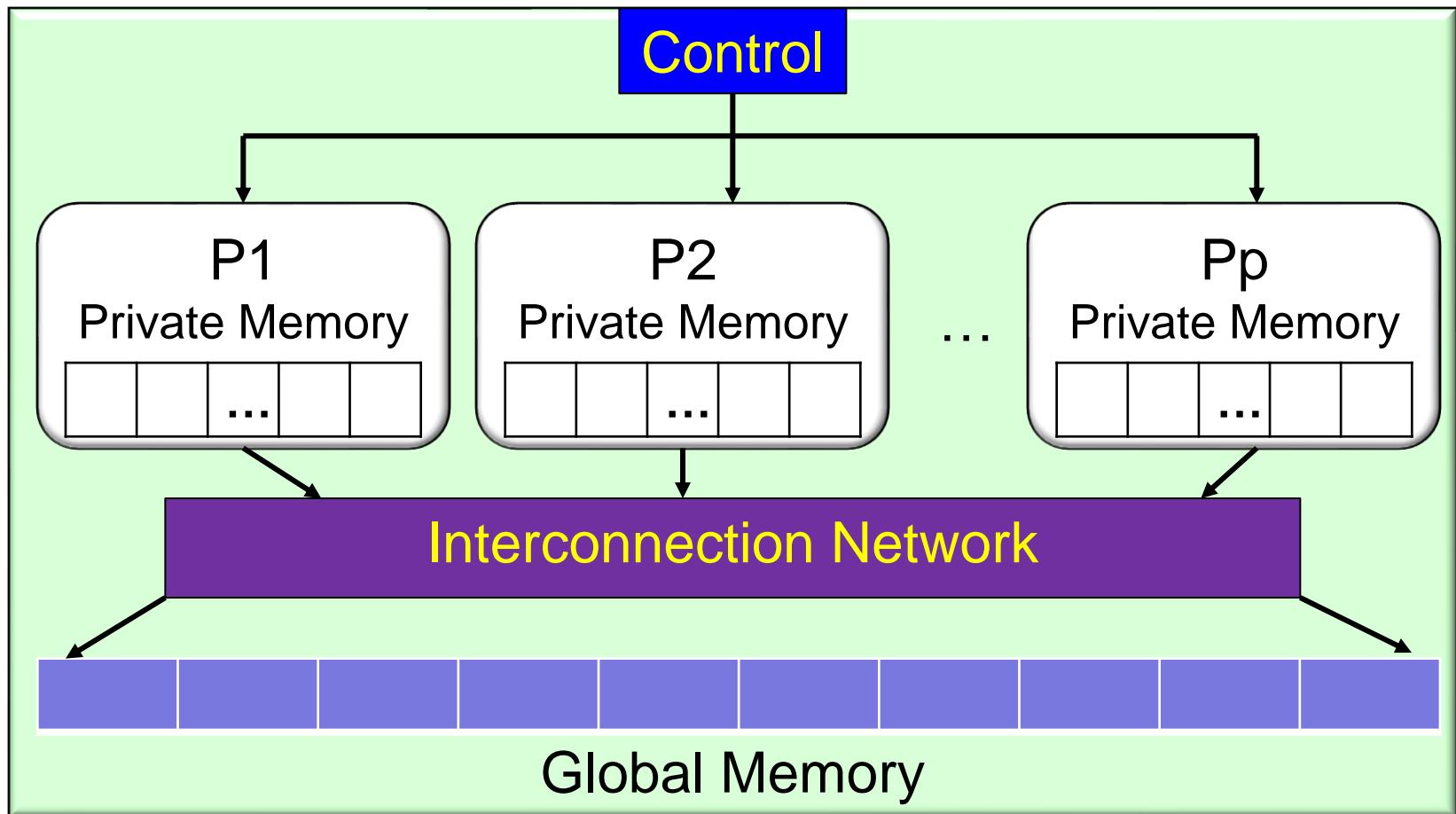
Go to next ppt

PRAM model of parallel computation

(Introduced by Fortune and Wyllie, 1978)

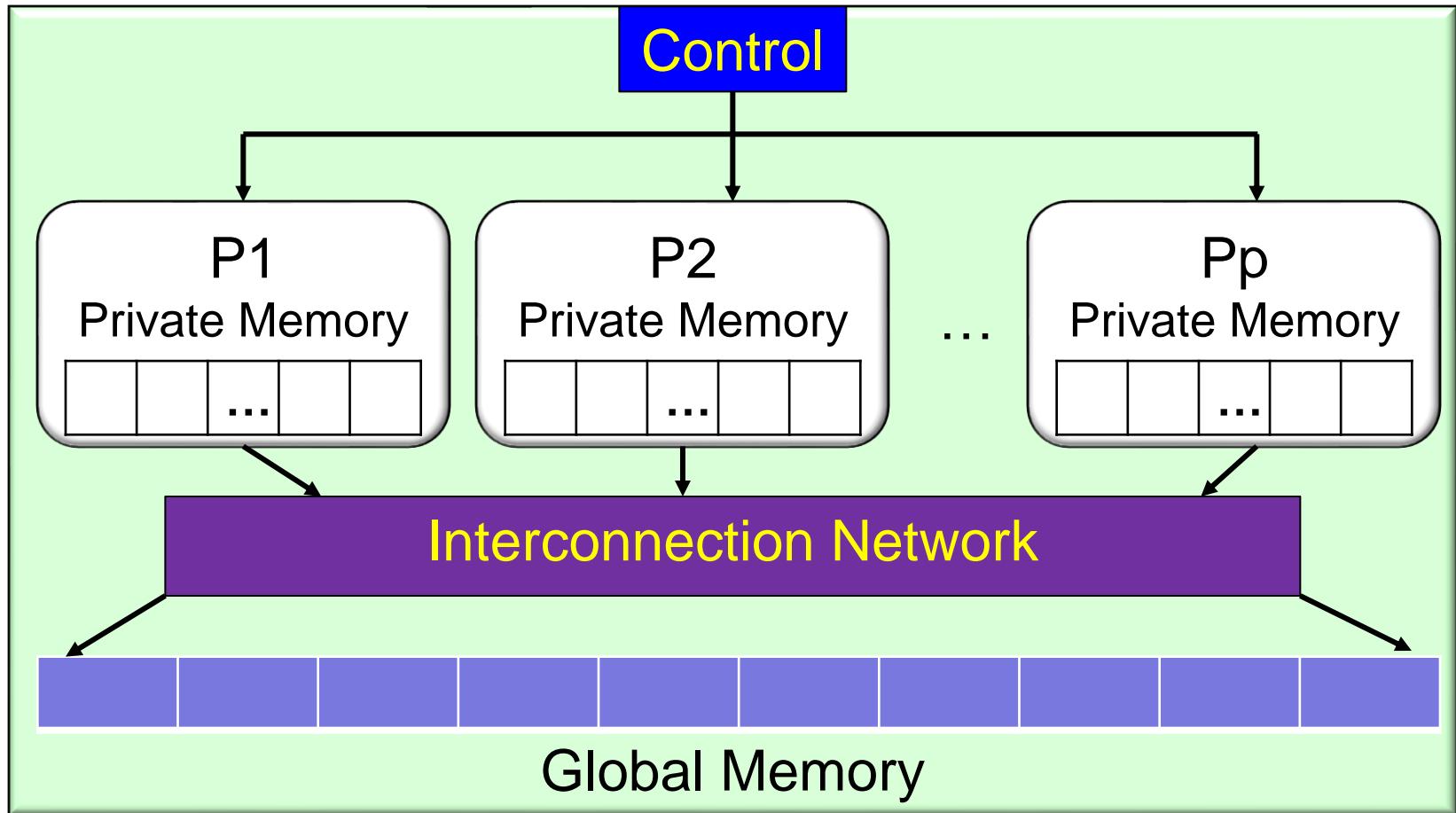


PRAM model of parallel computation



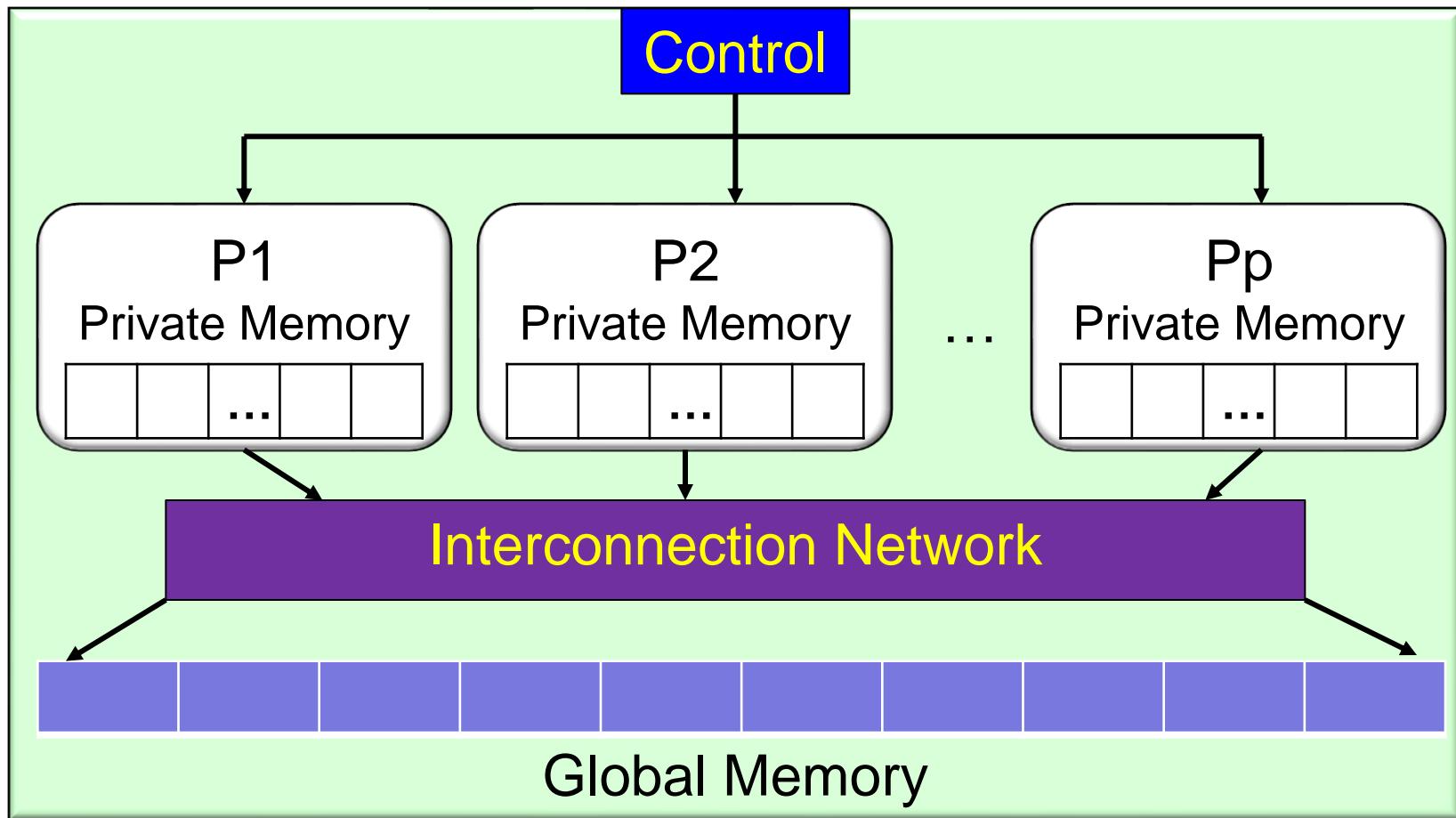
- Shared memory model with unbounded set of processors (RAM) having own memory and usual operations and instructions

PRAM model of parallel computation



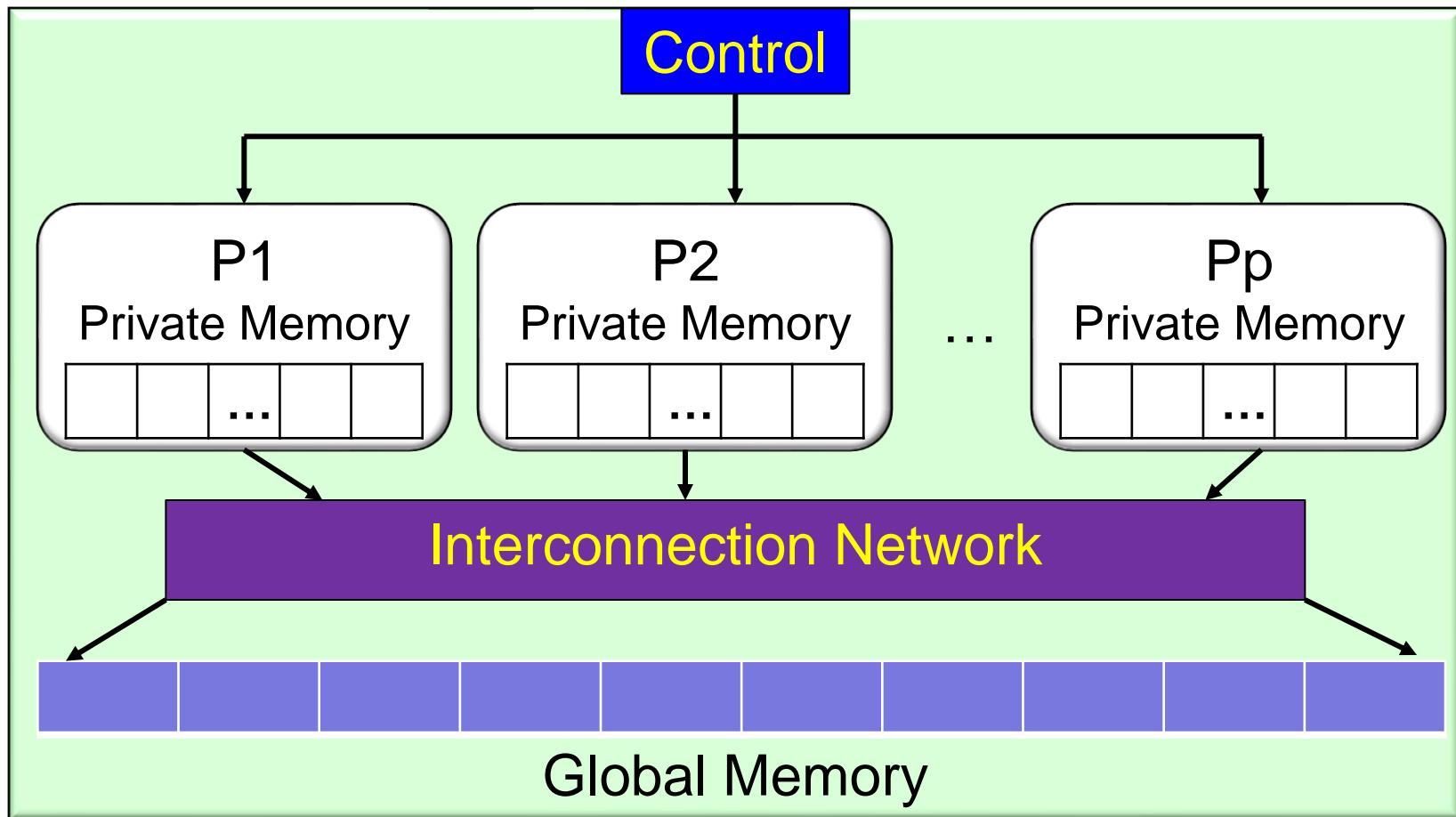
- The cost of arithmetic operation is constant
- Each processor is indexed by a natural number

PRAM model of parallel computation



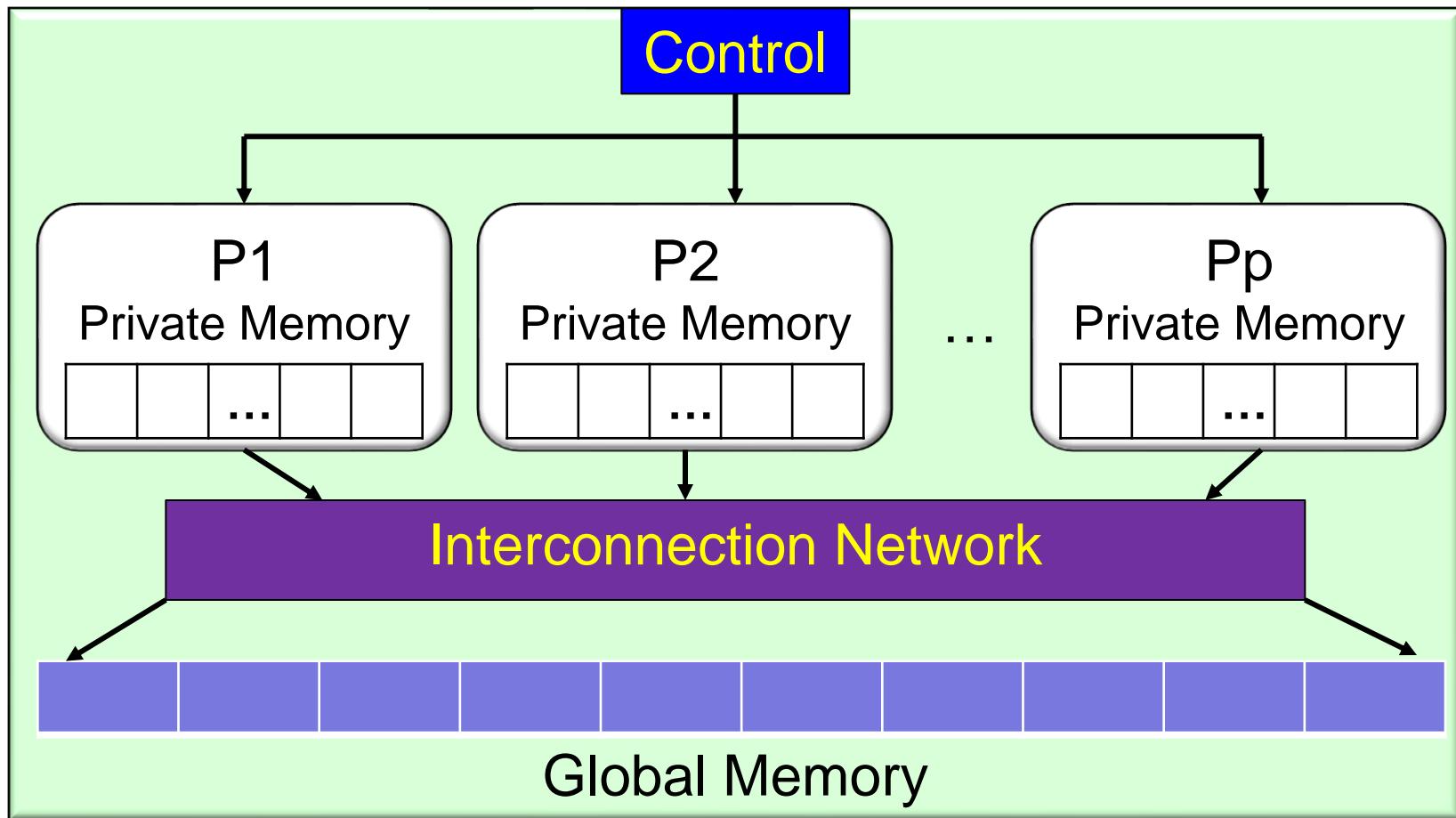
- Computation starts with single active processing element with input stored in global memory, **log p** time needed to activate **p** processors

PRAM model of parallel computation



- Any number of processors can read from the same memory location simultaneously

PRAM model of parallel computation



- No two processor can write into the same memory location

PRAM model of parallel computation

- Shared memory model with unbounded set of processors (RAM) having own memory and usual operations and instructions
- The cost of arithmetic operation is constant
- Each processor is indexed by a natural number
- Computation starts with single active processing element with input stored in global memory, **log p** time needed to activate **p** processors
- Any number of processors can read from the same memory location simultaneously
- No two processor can write into the same memory location

Definition: The cost of a PRAM computation is the product of the parallel time complexity and the **number of processors** used.

Various PRAM models **differ** in how they handle the read or write conflicts;

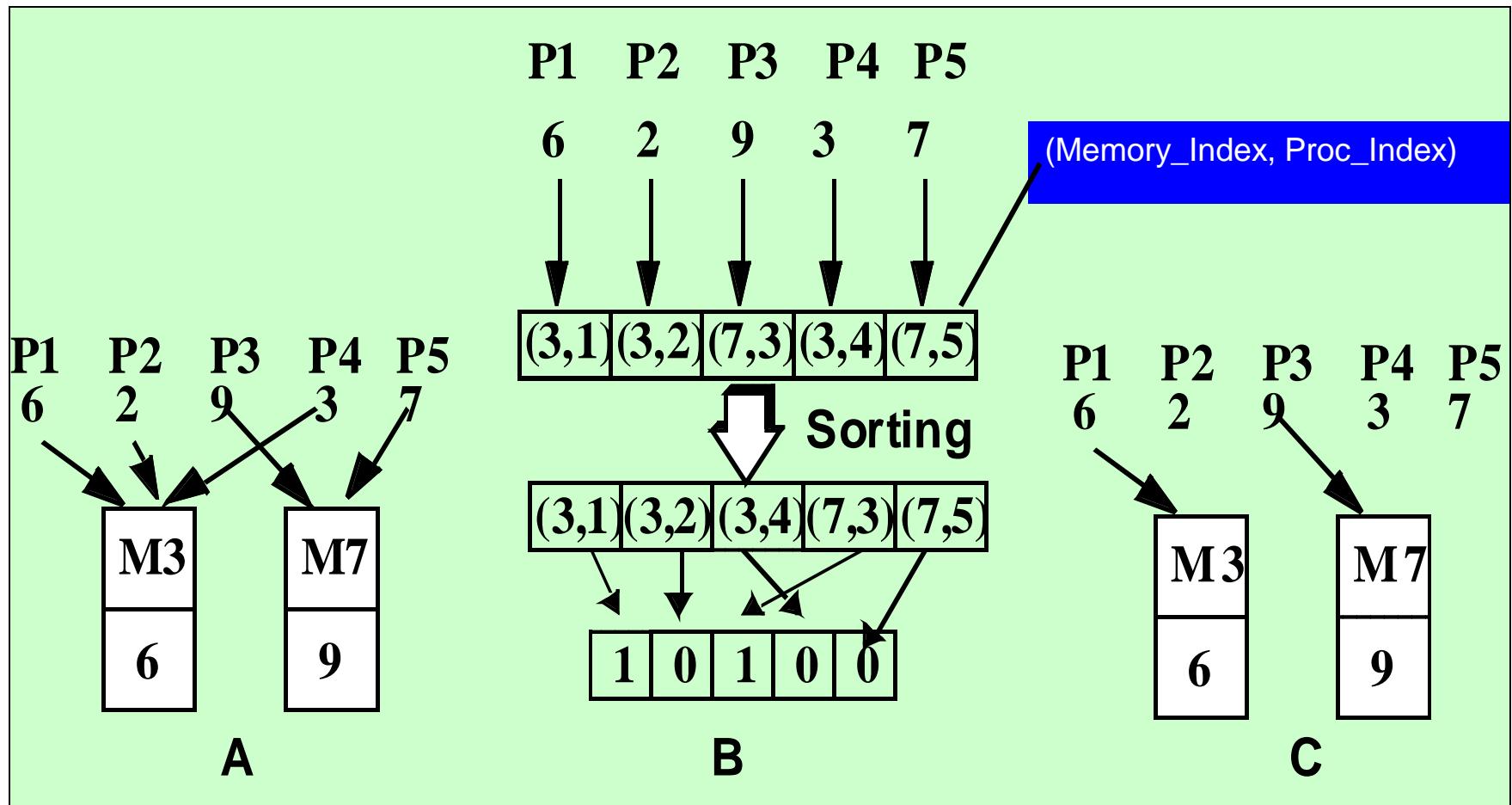
- **EREW (Exclusive Read Exclusive Write):** Read and write conflicts are **not allowed**
- **CREW (Concurrent Read Exclusive Write):** Concurrent read allowed (i.e. multiple processors are allowed to read from the same memory location), but concurrent write is **not allowed** (*Default PRAM*)
- **CRCW:** Concurrent read and concurrent write is allowed (W-RAM). There are different policies to handle the concurrent write operation.

- **COMMON:** All processors writing to the same memory location **must write same value**.
- **ARBITRARY:** If multiple processors concurrently write to the same global address, one of the competing processors is **arbitrarily chosen and its value is written into the register**.
- **PRIORITY:** If multiple processors concurrently write to the same global address, the processor with the **lowest index succeeds** writing its value into the memory location.

Relative strength of the models

Lemma: (Cole [88]) A p -processor EREW PRAM can sort a p -element array stored in global memory in $(\log n)$ time.

Theorem: A p -processors PRIORITY PRAM can be simulated by a p -processor EREW PRAM with the time complexity increased by a factor of $(\log n)$.



We build request records: (addr, processorID, value)

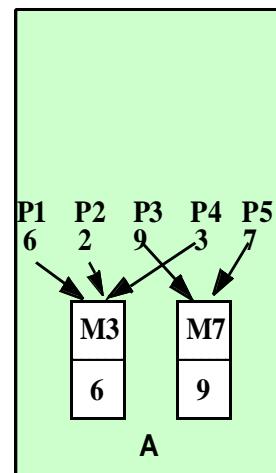
R1 = (M3, P1, 6)

R2 = (M3, P2, 2)

R3 = (M7, P3, 9)

R4 = (M3, P4, 3)

R5 = (M7, P5, 7)



Step A: Sort by address, then processorID

(M3, P1, 6)

(M3, P2, 2)

(M3, P4, 3)

(M7, P3, 9)

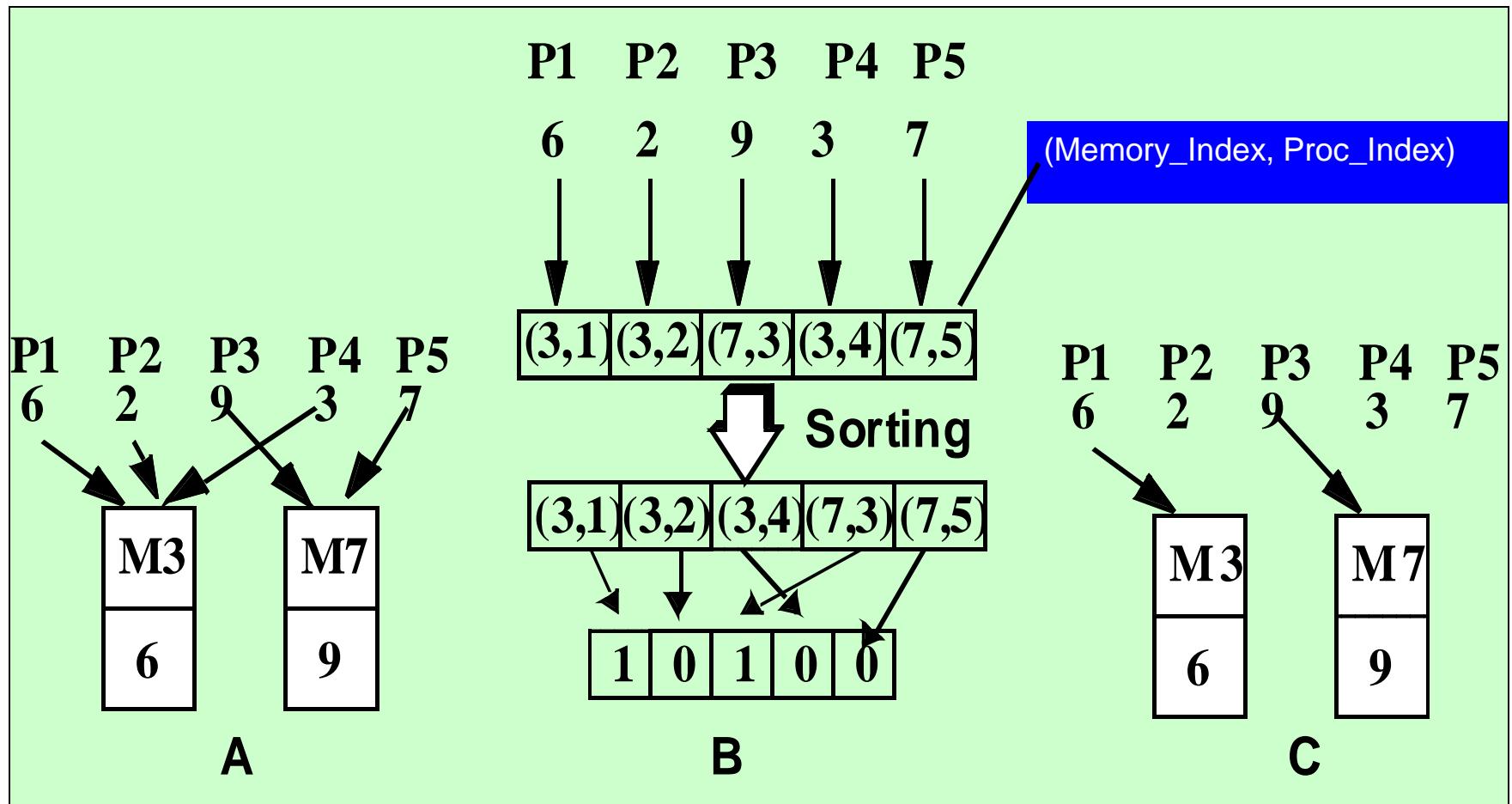
(M7, P5, 7)

Step B: For each block of same address, pick the *first record* (lowest processor ID).

Block M3 → (M3, P1, 6) → Winner = P1 → Write 6 to M3

Block M7 → (M7, P3, 9) → Winner = P3 → Write 9 to M7

M3 = 6, M7 = 9



Two statements are used in the algorithm description

1. **spawn** (<processor names>) (log p) time needed
2. **for all** <processor list> **do** <{statement name}>
endfor

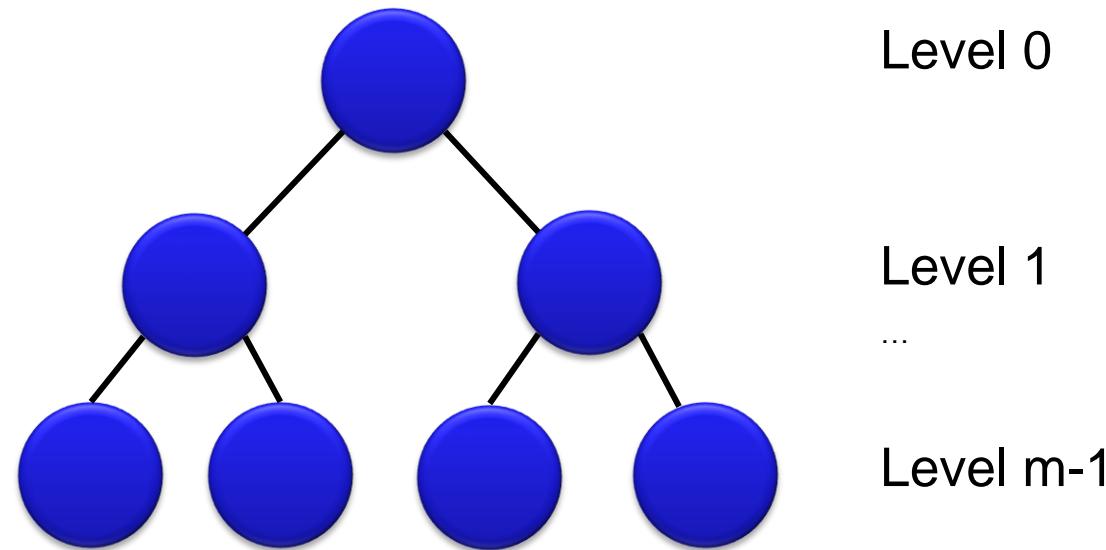
Binary tree is one of the most important data structures which can be exploited for the parallel algorithm design.

- **top down**
 1. Broadcast
 2. Divide and Conquer

- bottom up

fan-in or reduction

Balanced Binary tree method



Balanced Binary tree method structure

```
for levels m-1, m-2,..., 0 do
    for each vertex v at internal node in parallel
do
    value[v] = value[LeftChild[v]] op
                value[RightChild[v]]
output = value[root];
```

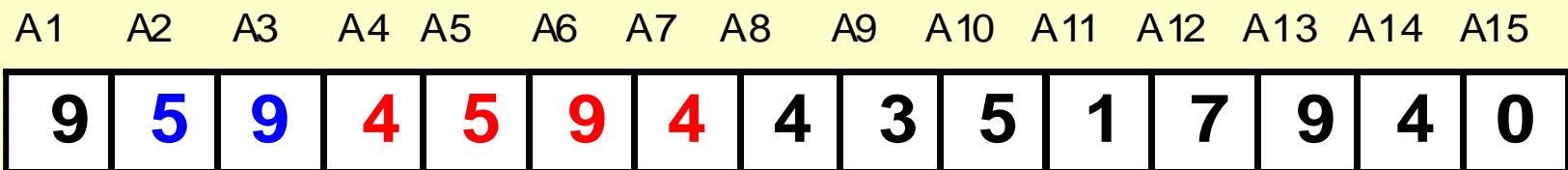
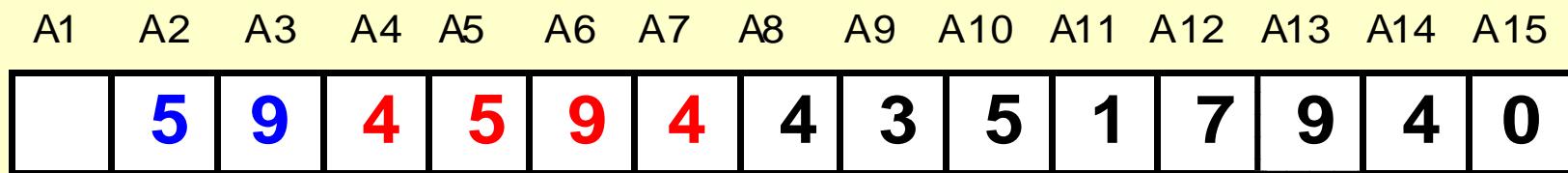
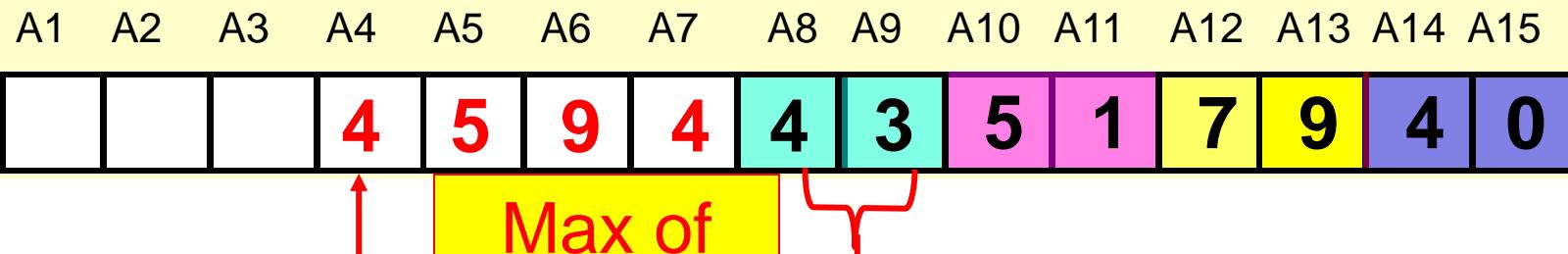
Maximum of $n = 2^m$ numbers stored in an array A of dimension $(2n-1)$ from $A(n), A(n+1), \dots, A(2n-1)$. At the end $A(1)$ stores the result.

for $k = m-1$ step -1 to 0 do

for all j , $2^k \leq j \leq 2^{k+1} - 1$, in parallel do

$$A(j) = \max\{A(2j), A(2j+1)\}$$

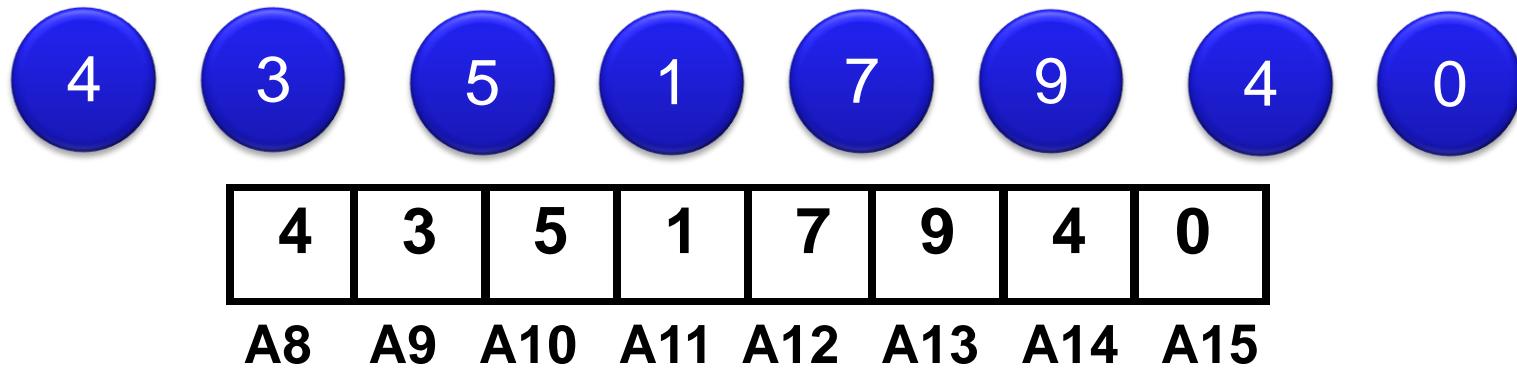
A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
							4	3	5	1	7	9	4	0



- No of processors used, Parallel time required

Maximum of $n = 2^m$ numbers stored in an array A of dimension $(2n-1)$ from $A(n), A(n+1), \dots, A(2n-1)$. At the end $A(1)$ stores the result.

Here $n = 8$ so $m = 3$, which is $\log_2 8$



for $k = m-1$ step -1 to 0 do

for all j , $2^k \leq j \leq 2^{k+1} - 1$ in parallel do

$$A(j) = \max\{A(2j), A(2j+1)\}$$

Here $n = 8$ so $m = 3$, which is $\log_2 8$ and values of $k = 2, 1, 0$

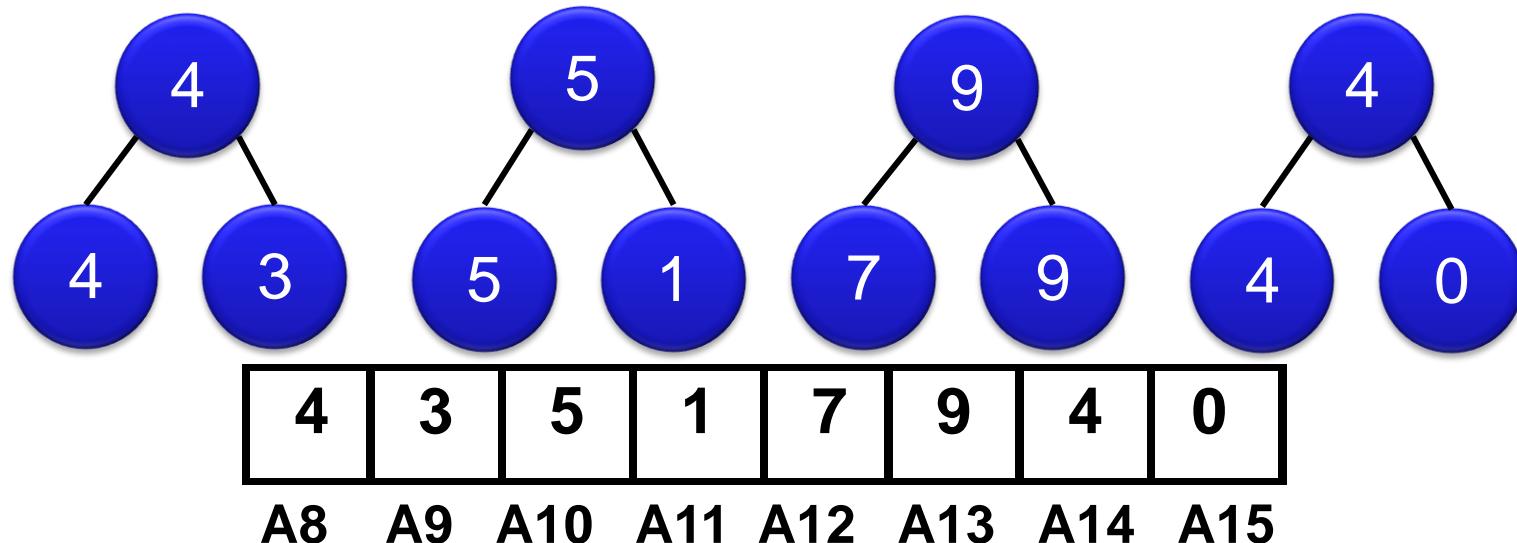
When $k = 2$, then values of $j = 4, 5, 6, 7$, four || operations are

$$A(4) = \max \{A(8), A(9)\}$$

$$A(5) = \max \{A(10), A(11)\}$$

$$A(6) = \max \{A(12), A(13)\}$$

$$A(7) = \max \{A(14), A(15)\}$$

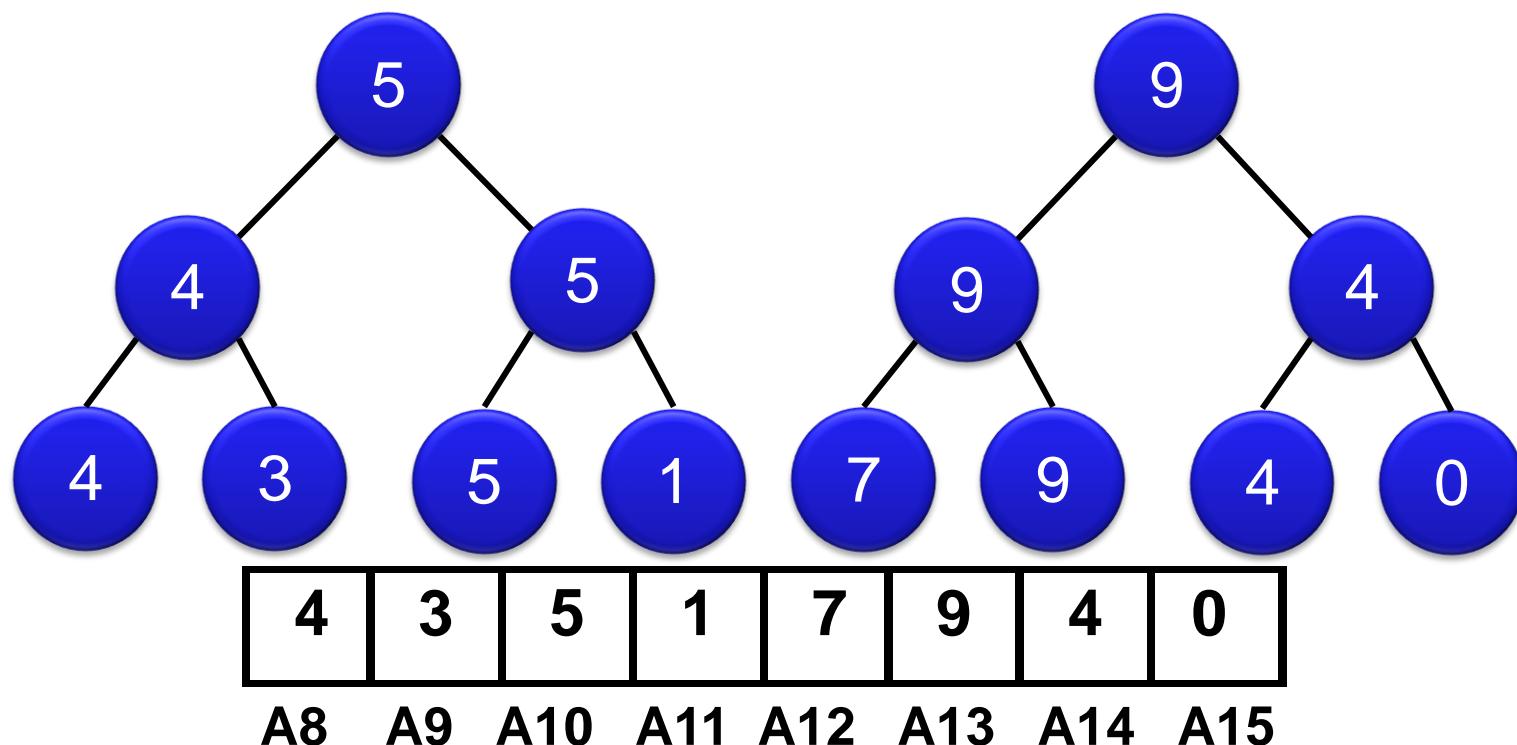


Here $n = 8$ so $m = 3$, which is $\log_2 8$ and values of $k = 2, 1, 0$

When $k = 1$, then values of $j = 2, 3$, two \parallel operations are

$$A(2) = \max \{A(4), A(5)\}$$

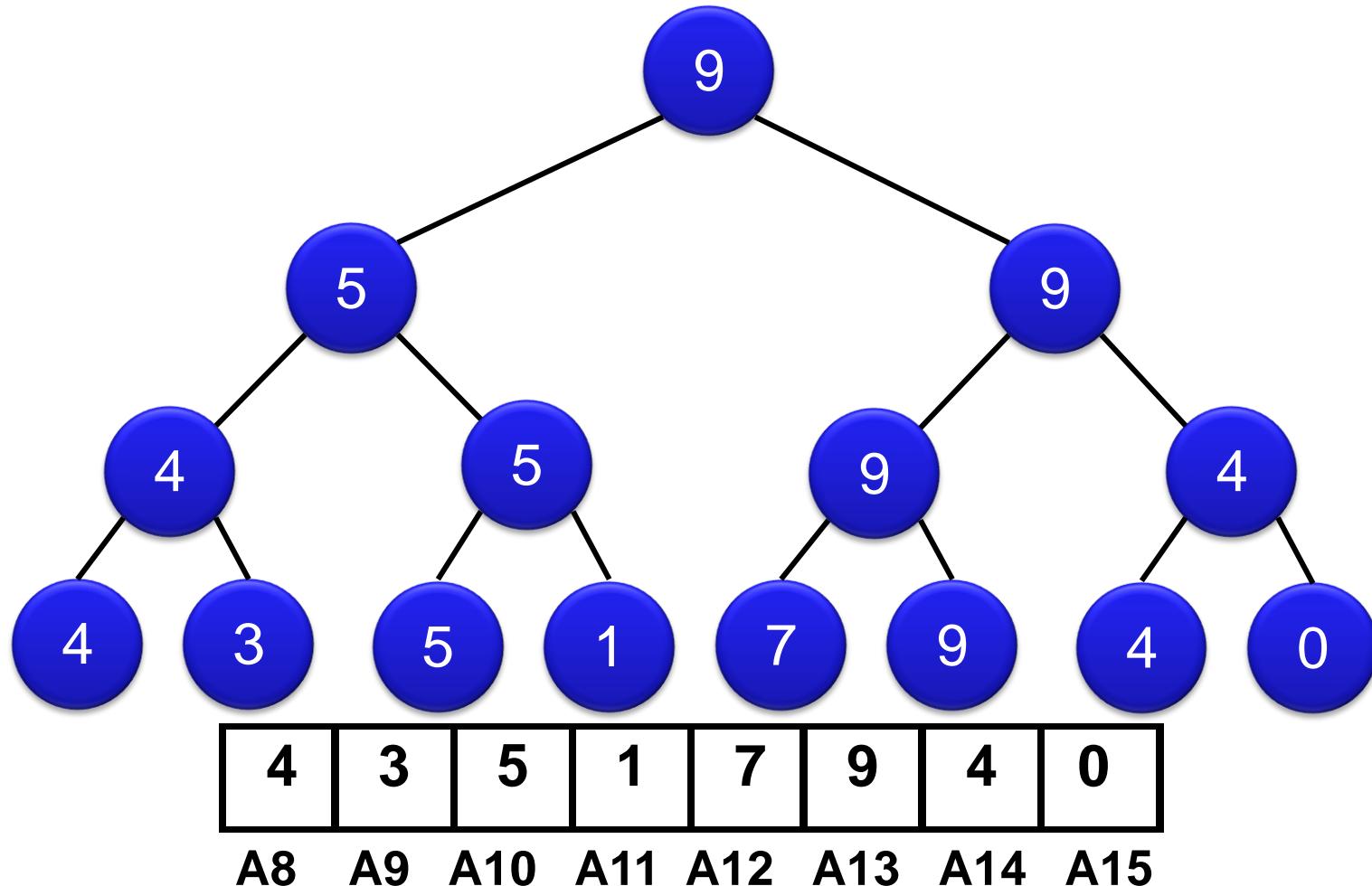
$$A(3) = \max \{A(6), A(7)\}$$



Here $n = 8$ so $m = 3$, which is $\log_2 8$ and values of $k = 2, 1, 0$

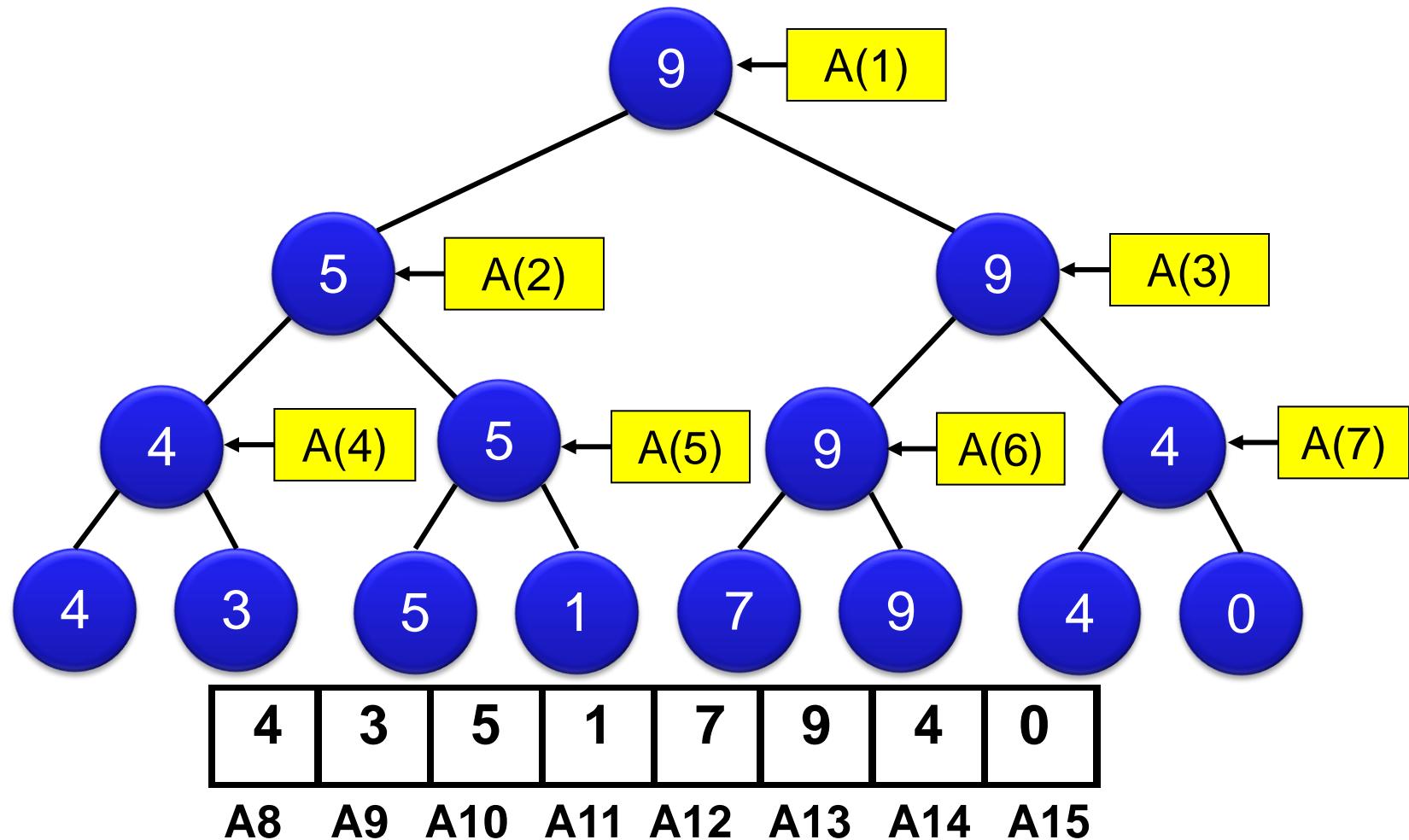
When $k = 0$, then values of $j = 1$ one || operation is

$$A(1) = \max \{A(2), A(3)\}$$



A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15

9	5	9	4	5	9	4	4	3	5	1	7	9	4	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



SUM (EREW PRAM)

Global n, A[0,...,(n-1)],j

begin

spawn (P0,P1,...,P $\lfloor n/2 \rfloor - 1$)

for all Pi where $0 \leq i \leq \lfloor n/2 \rfloor - 1$ do

 for j = 0 to $\lceil \log_2 n - 1 \rceil$ do

 if ($i \bmod 2^j$) = 0 and $(2i + 2^j) < n$ then

$A[2i] = A[2i] + A[2i + 2^j]$

 endif

 endfor

endfor

end

SUM (EREW PRAM)

Global n, A[0,...,(n-1)],j

begin

 spawn (P0,P1,...,P_{n-1})

for all Pi where $0 \leq i \leq \lfloor n/2 \rfloor - 1$ do

for j = 0 to $\lceil \log_2 n - 1 \rceil$ do

 if $(i \bmod 2^j) = 0$ and $(2i + 2^j) < n$ then

 A[2i] = A[2i] + A[2i + 2^j]

endif

endfor

endfor

end

n = 7, spawn P0, P1, P2, $\lceil \log_2 n - 1 \rceil = 2$

At P0	At P1	At P2
j = 0 if $(i \bmod 2^j) = 0$ and $(2i + 2^j) < n$ then A[2i] = A[2i] + A[2i + 2 ^j]	j = 0 if $(i \bmod 2^j) = 0$ and $(2i + 2^j) < n$ then A[2i] = A[2i] + A[2i + 2 ^j]	j = 0 if $(i \bmod 2^j) = 0$ and $(2i + 2^j) < n$ then A[2i] = A[2i] + A[2i + 2 ^j]
j = 1 if $(i \bmod 2^j) = 0$ and $(2i + 2^j) < n$ then A[2i] = A[2i] + A[2i + 2 ^j]	j = 1 if $(i \bmod 2^j) = 0$ and $(2i + 2^j) < n$ then A[2i] = A[2i] + A[2i + 2 ^j]	j = 1 if $(i \bmod 2^j) = 0$ and $(2i + 2^j) < n$ then A[2i] = A[2i] + A[2i + 2 ^j]
j = 2 if $(i \bmod 2^j) = 0$ and $(2i + 2^j) < n$ then A[2i] = A[2i] + A[2i + 2 ^j]	j = 2 if $(i \bmod 2^j) = 0$ and $(2i + 2^j) < n$ then A[2i] = A[2i] + A[2i + 2 ^j]	j = 2 if $(i \bmod 2^j) = 0$ and $(2i + 2^j) < n$ then A[2i] = A[2i] + A[2i + 2 ^j]

SUM (EREW PRAM)

Global n, A[0,...,(n-1)],j

begin

 spawn (P0,P1,...,P $\lfloor n/2 \rfloor - 1$)

for all Pi where $0 \leq i \leq \lfloor n/2 \rfloor - 1$ **do**

for j = 0 to $\lceil \log_2 n - 1 \rceil$ **do**

if (i mod 2^j) = 0 and ($2i + 2^j$) < n **then**

 A[$2i$] = A[$2i$] + A[$2i + 2^j$]

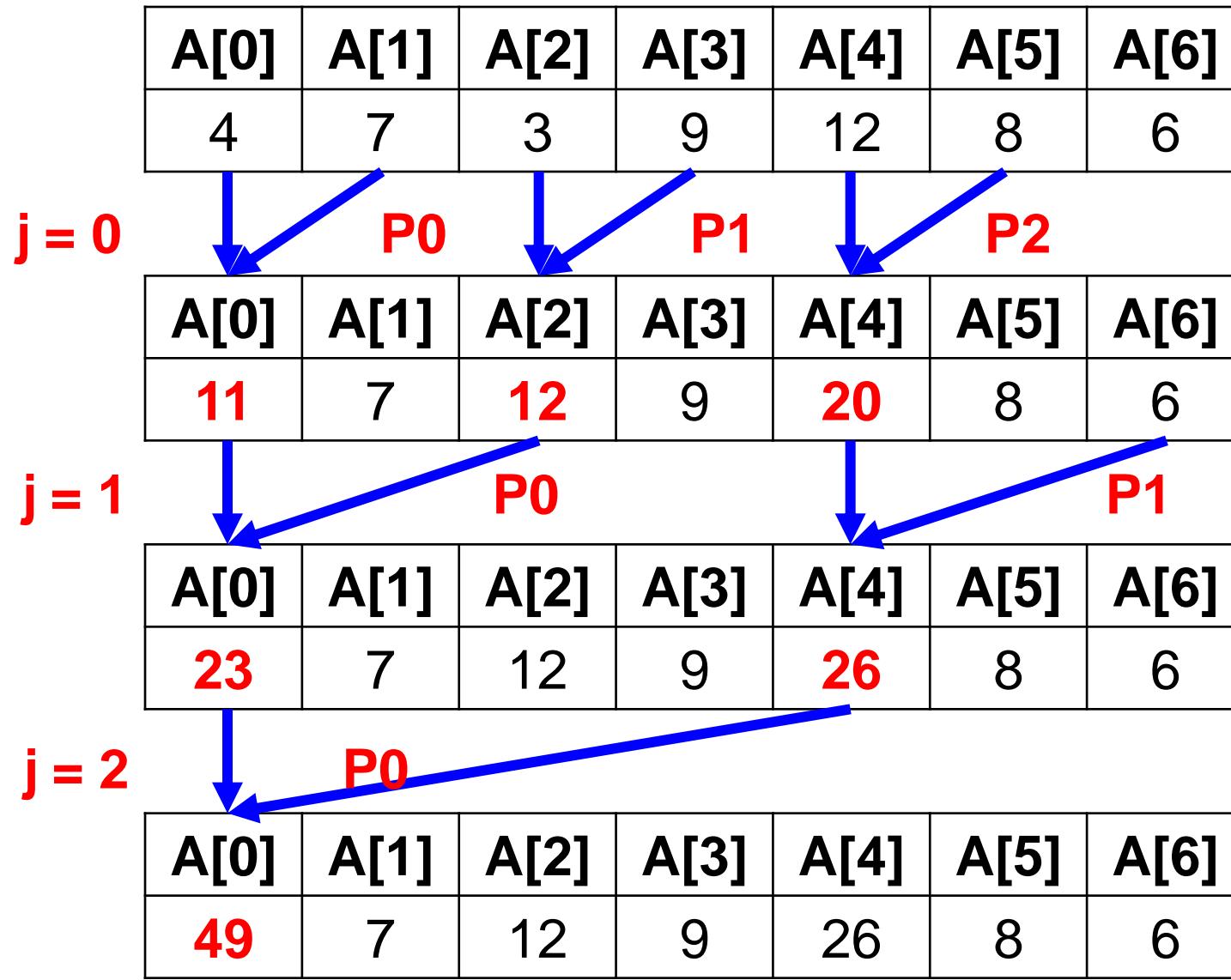
endif

endfor

endfor

end

$n = 7$, spawn P0, P1, P2



Prefix Computations

In computer science, the prefix sum, cumulative sum of a sequence of numbers a_0, a_1, a_2, \dots is a second sequence of numbers b_0, b_1, b_2, \dots , the sums of prefixes of the input sequence:

$$b_0 = a_0$$

$$b_1 = a_0 + a_1$$

$$b_2 = a_0 + a_1 + a_2$$

$$b_3 = a_0 + a_1 + a_2 + a_3$$

...

Prefix Computations

n numbers are stored in an array A of dimension $(n-1)$ from $A(0), A(1), \dots, A(n-1)$. At the end, $A(i)$ stores $A[0], A[1] \dots A[i]$.

Applications of Prefix Computation

- Knapsack Problem
- Job Sequencing with deadline
- Compiler Design
- Computational Biology
- Evaluation of Polynomials
- Solving System of Linear Equations
- Polynomial Interpolation

PREFIX.SUMS (CREW PRAM)

Global n, A(0), A(1),...,A(n-1), j

begin

spawn(P_0, P_1, \dots, P_{n-1})

for all P_i where $0 \leq i \leq n - 1$ do

 for $j = 0$ to $\lceil \log_2 n \rceil - 1$ do

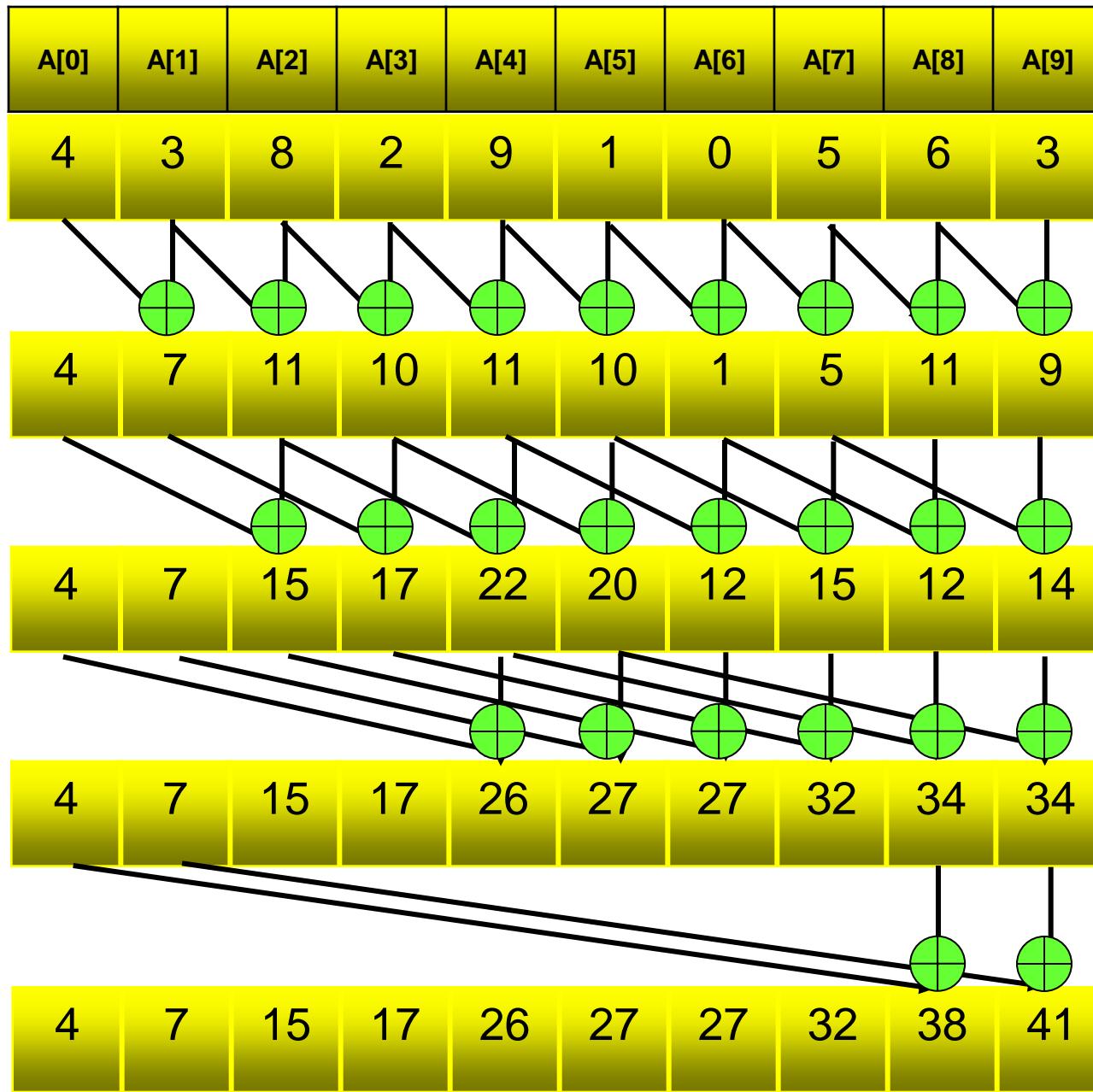
 if $(i - 2^j) \geq 0$ then

$A[i] = A[i] + A[i - 2^j]$

 endif

 endfor

endfor end



Seq. Steps

$j = 0$

$j = 1$

$j = \lceil \log_2 n \rceil - 1$

Exercise: $n = 2^m$ numbers stored in an array A of dimension $(2n-1)$ from $A(n), A(n+1), \dots, A(2n-1)$. Write an algorithm for obtaining the prefix sum of these numbers, at the end $A(i), 1 \leq i \leq n-1$ stores the result.

Doubling techniques

Normally applied to an array or to a list of elements. The computation proceeds by a recursive application of the computation in hand to all the elements.

Linked List Ranking

- Given a linked list, stored in an array, compute the distance of each element from the end (either end) of the list.
- Called **Pointer Jumping** when using pointers.
- Don't destroy original list!

The distance doubles in successive steps.
Thus after k iterations computation to all
elements at distance 2^k is performed.

Value in an array **next** represents linked list

Value in an array **position** contain original
distance of each element from end of the list.

Global: n, position[0..(n-1)], next[0..(n-1)], j

LIST.RANKING(CREW PRAM)

begin

 spawn(P_0, P_1, \dots, P_{n-1})

for all P_i where $0 \leq i \leq n - 1$ **do**

if $\text{next}[i] = i$ **then** $\text{position}[i] := 0$

else $\text{position}[i] := 1$

endif

for $j = 1$ to $\lceil \log_2 n \rceil - 1$ **do**

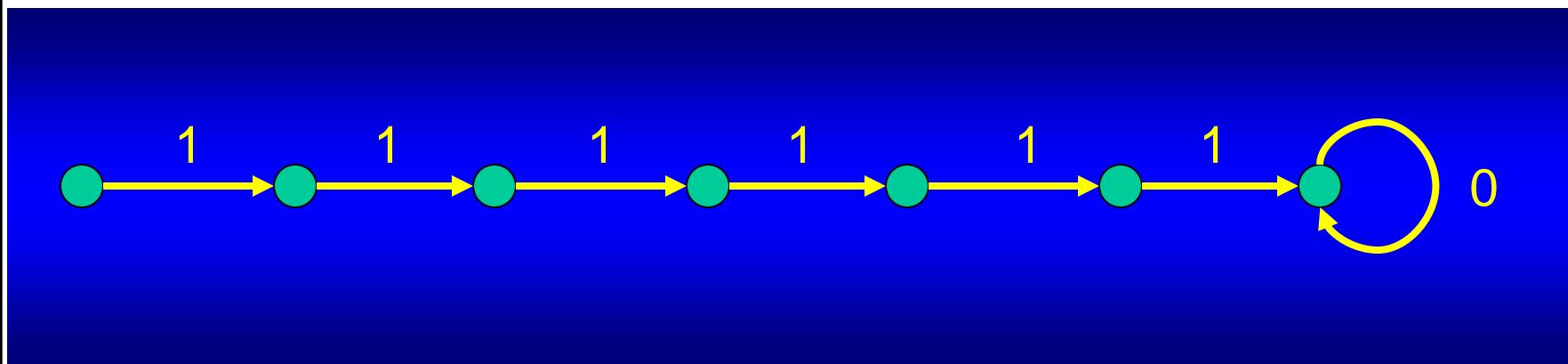
$\text{position}[i] := \text{position}[i] + \text{position}[\text{next}[i]]$

$\text{next}[i] = \text{next}[\text{next}[i]]$

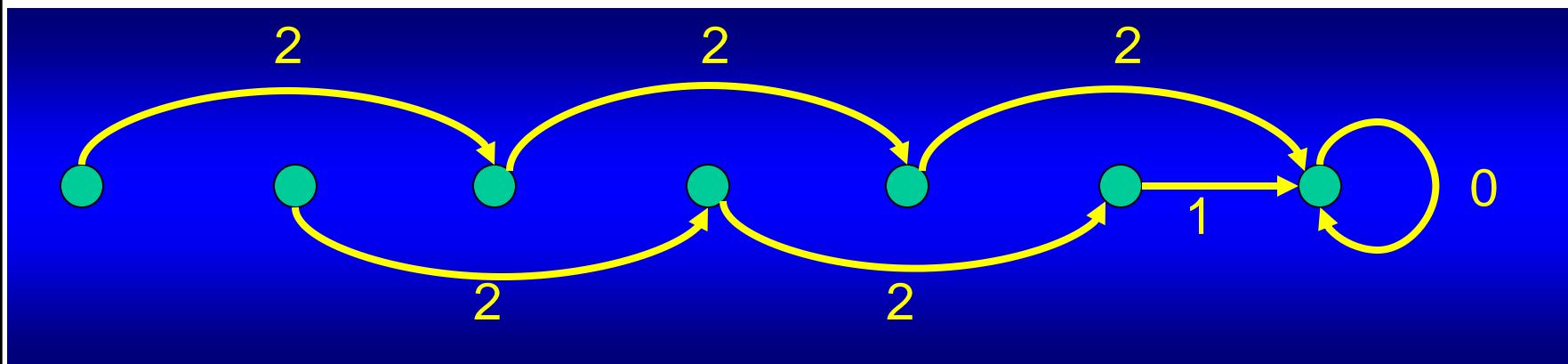
endfor

endfor **end**

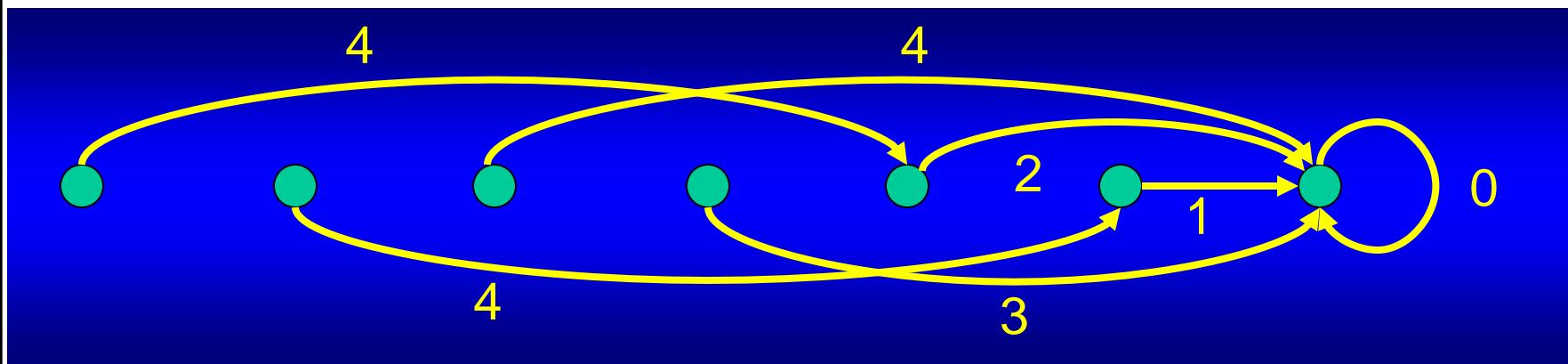
Step 1:



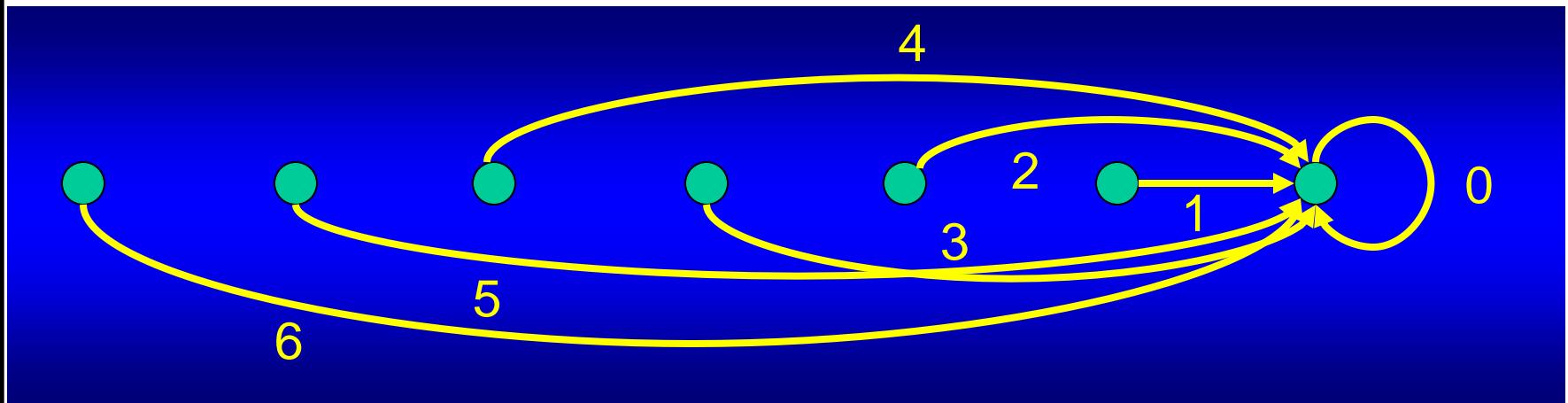
Step 2:



Step 3:



Step 4:



Parallel time complexity, Processors 

Work Analysis

- Number of Steps: $T_p = O(\log N)$
- Number of Processors: N
- Work = $O(N \log N)$
- Sequential = $O(N)$
- Optimal??

Applications of List Ranking

- Expression Tree Evaluation
- Parentheses Matching
- Tree Traversals
- Ear–Decomposition of Graphs
- Euler tour of trees
- - - - many others

● **Merging two sorted lists**

- Best known sequential algorithm needs $O(n)$ time.
- Every processor finds the position of *its own* element on the other list using *binary search*, making an algorithm that takes $O(\log n)$ parallel time.

Assumption: Two lists and their unions have disjoint values.

Global A[1..n]

MERGE.LISTS(CREW PRAM):

Local x, low, high, index

begin

spawn(P_1, P_2, \dots, P_n)

for all P_i where $1 \leq i \leq n$ do

if ($i \leq n/2$) then

low := $(n/2)+1$

high := n

else

low := 1

high := $n/2$

endif

{Each processor performs binary search}

x := A[i]

repeat

index:= $\lfloor (low + high)/2 \rfloor$

if $x < A[index]$ then

high := index-1

else

low := index + 1

endif

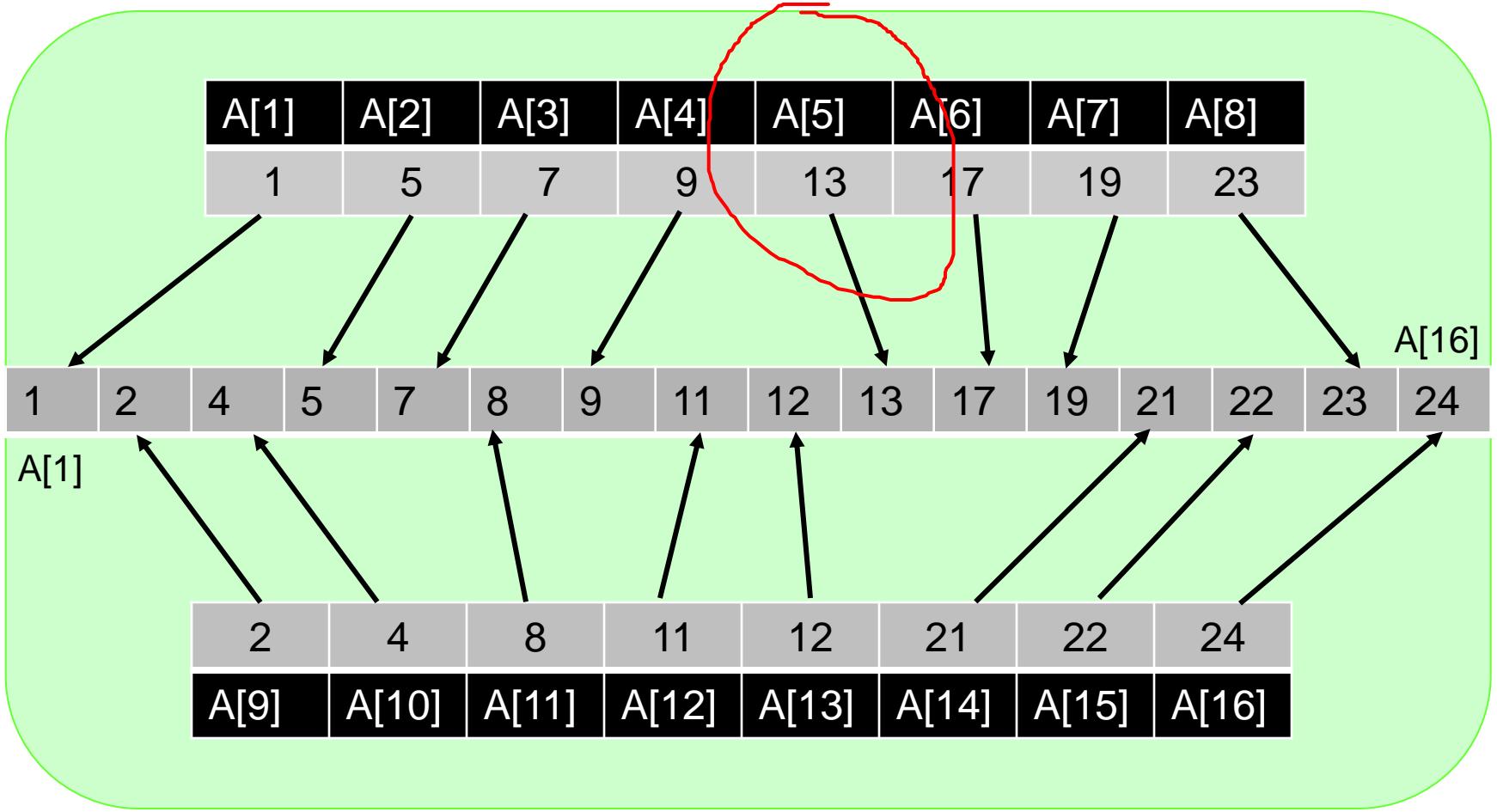
until ($low > high$)

{put values in correct position on merged list}

A[high+i-n/2] := x

endfor

end



Parallel time = ?,
 No. of processors = ?

Global A[1..n]

MERGE.LISTS(CREW PRAM):

Local x, low, high, index

begin

spawn(P_1, P_2, \dots, P_n)

for all P_i where $1 \leq i \leq n$ do

if ($i \leq n/2$) then

low := $(n/2)+1$

high := n

else

low := 1

high := $n/2$

endif

P_1	...	P_5	...	P_{16}
-------	-----	-------	-----	----------

Let us discuss at P_5

$i = 5;$

if ($i \leq n/2$) i.e. ($5 \leq 8$, as $n = 16$)

low := 8+1=9

high := 16

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
1	5	7	9	13	17	19	23

{Each processor performs

binary search}

$x := A[i]$

repeat

index:= $\lfloor (low + high)/2 \rfloor$

if $x < A[index]$ then

 high := index-1

else

 low := index + 1

endif

until (low > high)

$A[high+i-n/2] := x$

endfor

end



Let us discuss at P_5

$i = 5;$

if ($i \leq n/2$) i.e. ($5 \leq 8$, as $n = 16$)

low:= 8+1=9

high:=16

$x := A[5] = 13$

repeat

index:= $\text{floor}((9+16)/2) = 12$

if ($13 < A[index]$) i.e. $A[12]$

{here $13 < 11$ } NO, else part

low:=index+1 = $12+1 = 13$

Until (low > high)

{Each processor performs

binary search}

$x := A[i]$

repeat

$\text{index} := \lfloor (low + high)/2 \rfloor$

if $x < A[\text{index}]$ **then**

$\text{high} := \text{index}-1$

else

$\text{low} := \text{index} + 1$

endif

until ($low > high$)

$A[\text{high}+i-n/2] := x$

endfor

end



Let us discuss at P_5

$i = 5;$

if ($i \leq n/2$) i.e. ($5 \leq 8$, as $n = 16$)

low:=8+1=9

high:=16

$x := A[5] = 13$

repeat

$\text{index} := \text{floor}((9+16)/2) = 12$

if ($13 < A[\text{index}]$) i.e. $A[12]$

{here $13 < 11$ } **NO, else part**

low:=index+1 = 12+1 = 13

Until ($low > high$)

Next loop

$\text{index} := \text{floor}((13+16)/2) = 14$

if ($13 < A[\text{index}]$) i.e. $A[14]$

{here $13 < 21$ } **YES**

high:= index-1 = 14-1 = 13

Next loop

$\text{index} := \text{floor}((13+13)/2) = 13$

if ($13 < A[\text{index}]$) i.e. $A[13]$

{here $13 < 12$ } **NO**

low:=index+1 = 13+1 = 14

{Each processor performs

binary search}

$x := A[i]$

repeat

$\text{index} := \lfloor (low + high)/2 \rfloor$

if $x < A[\text{index}]$ **then**

$\text{high} := \text{index}-1$

else

$\text{low} := \text{index} + 1$

endif

until ($low > high$)

$A[\text{high}+i-n/2] := x$

endfor

end



Let us discuss at P_5

$i = 5;$

if ($i \leq n/2$) i.e. ($5 \leq 8$, as $n = 16$)

:

$x := A[5] = 13$

:

Next loop

$\text{index} := \text{floor}((13+16)/2) = 14$

if ($13 < A[\text{index}]$) i.e. $A[14]$

{here $13 < 21$ } **YES**

high := $\text{index}-1 = 14-1 = 13$

Next loop

$\text{index} := \text{floor}((13+13)/2) = 13$

if ($13 < A[\text{index}]$) i.e. $A[13]$

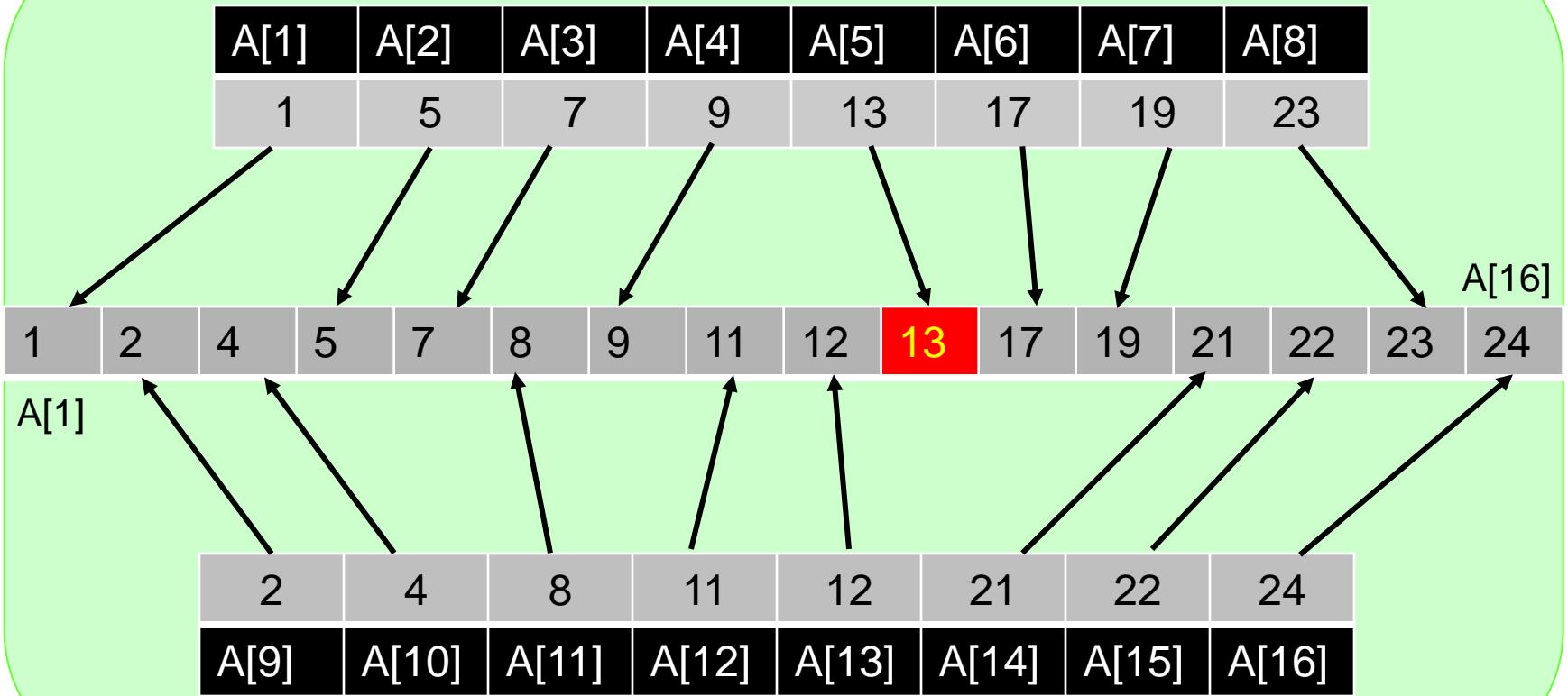
{here $13 < 12$ } **NO**

low := $\text{index}+1 = 13+1 = 14$

as ($low > high$) **then** loop exit

$A[\text{high}+i-n/2] := x$ i.e.

$A[13+5-8] := x$, i.e. $A[10] := 13$



Parallel time = ?,
 No. of processors = ?

Do we need so many processors



(Cost) Optimal parallel algorithm: One in which the product of number of processor p used and parallel time t is linear in problem size S , i.e. $pt = O(S)$

Reducing the number of processors

Suppose we have designed an algorithm working in parallel time t with p processors, here we assume that p is the maximum number of operations executed in the same parallel step.

Maximum finding algorithm takes $O(\log n)$ time with the $p \geq n/2$ processors, in fact $n/2$ processors are required only at the beginning of the procedure. Most of the processors are sitting idle.

suppose we have $p < n/2$ processors. Partition n elements in p groups. $p - 1$ such group will be having $[n/p]$ elements and remaining group contains

$(n-(p-1)[n/p] \leq ?)$ elements.

suppose we have $p < n/2$ processors. Partition n elements in p groups. $p - 1$ such group will be having $\lceil n/p \rceil$ elements and remaining group contains

$$(n-(p-1) \lceil n/p \rceil) \leq \lceil n/p \rceil \text{ elements.}$$

Assign a processor to each group which finds maximum in $\lceil n/p \rceil - 1$ time each, in parallel, later $\log p$ time using balanced binary tree method.

Case: n is exactly divisible by p



Here $n = 20$, p (processors) = 5, $[n/p] = 4$



At p1, exactly n/p elements



At p2, exactly n/p elements



At p3, exactly n/p elements



At p4, exactly n/p elements



At p5, exactly n/p elements

Case: n is exactly NOT divisible by p



Here $n = 20$, $p = 3$, $\lceil n/p \rceil = 7$ so $(p-1) = 2$ processors having $n/p = 7$ elements each



At p_1 , n/p elements = 7



At p_2 , n/p elements = 7



At p_3 , $(n-(p-1))$ $\lceil n/p \rceil \leq \lceil n/p \rceil = 6$ elements.

Thus overall time is $[n/p] - 1 + \log p$ with $p < n/2$ processors.

What if $p = n / \log n$ 

Brent's theorem: Let A be a given parallel algorithm with computation time t , if parallel algorithm performs m computational operations then p processors can execute algorithm A in time $O(m/p + t)$.

Definition: The set $(\log n)^{O(n)}$ is called the set of **poly-logarithmic function**.

Theorem(Parallel computation thesis): The class of problems solvable in time $T(n)^{O(n)}$ by a PRAM is equal to the class of problems solvable in work space $T(n)^{O(n)}$ by a RAM, if $T(n) \geq \log n$.

Bulk Synchronous Parallel

What with PRAM?

- PRAM can emulate a message-passing computer by dividing the memory into private memories with each processor
- Several PRAM based papers (fine-grained) algorithmic techniques
 - Results seem irrelevant, **posterior time is away**
 - Performance predictions are **inaccurate**
 - Hasn't lead to programming languages
 - Hardware doesn't have fine-grained synchronous steps

BSP

- The **Bulk Synchronous Parallel (BSP)** abstract computer is a bridging model for designing parallel algorithms.
- It serves a purpose similar to the Parallel Random Access Machine (PRAM) model. It is generalization of PRAM model.
- BSP does not take communication and synchronization for granted.
- An important part of analyzing a BSP algorithm rests on **quantifying the synchronization and communication** needed.

BSP-History

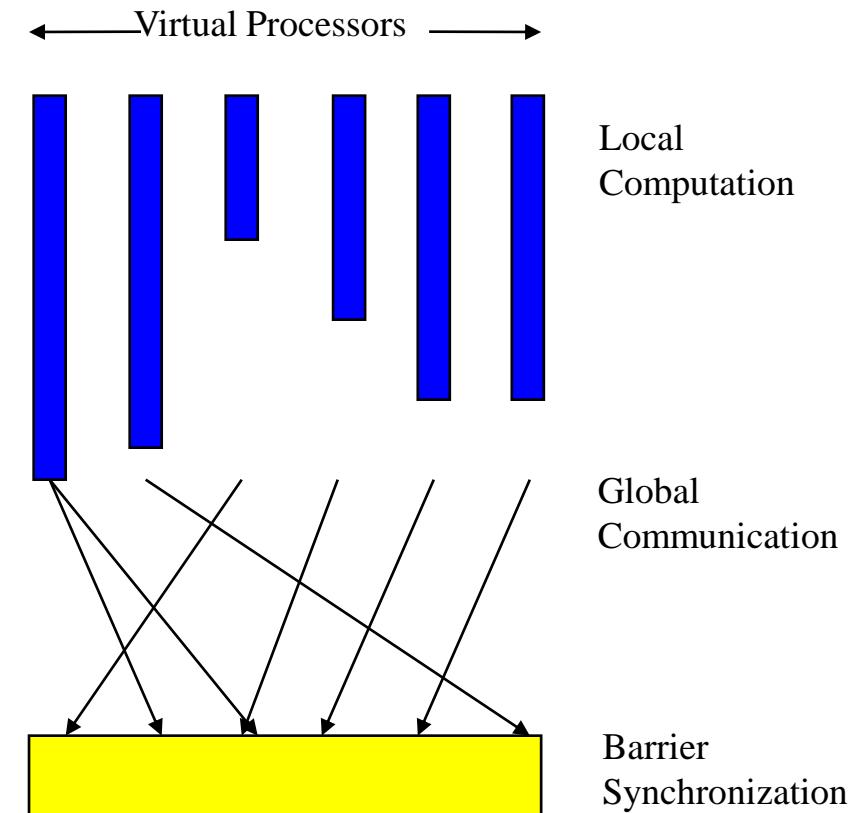
- **BSP: Bulk-Synchronous Parallel**
 - Valiant, Leslie G., “*A Bridging Model for Parallel Computation*”, Communications of the ACM, Aug., 1990, Vol. 33, No. 8, pp. 103-111.
- BSP is designed to be **architecture independent**
 - Portable programs
- BSP considers at a global level (*bulk*) computation and communication
- Execution time of a BSP program is computed by the local execution time and from few parameters tied to the particular architecture that is used

BSP

- A BSP computer consists of
 - **Components** capable of processing and/or local memory transactions
 - a **network** that **routes** messages between pairs of such components, and
 - a **hardware facility** that allows for the *synchronization* of all or a subset of components. I.e. **Periodicity parameter L** : to facilitate synchronization at regular intervals of L time units.

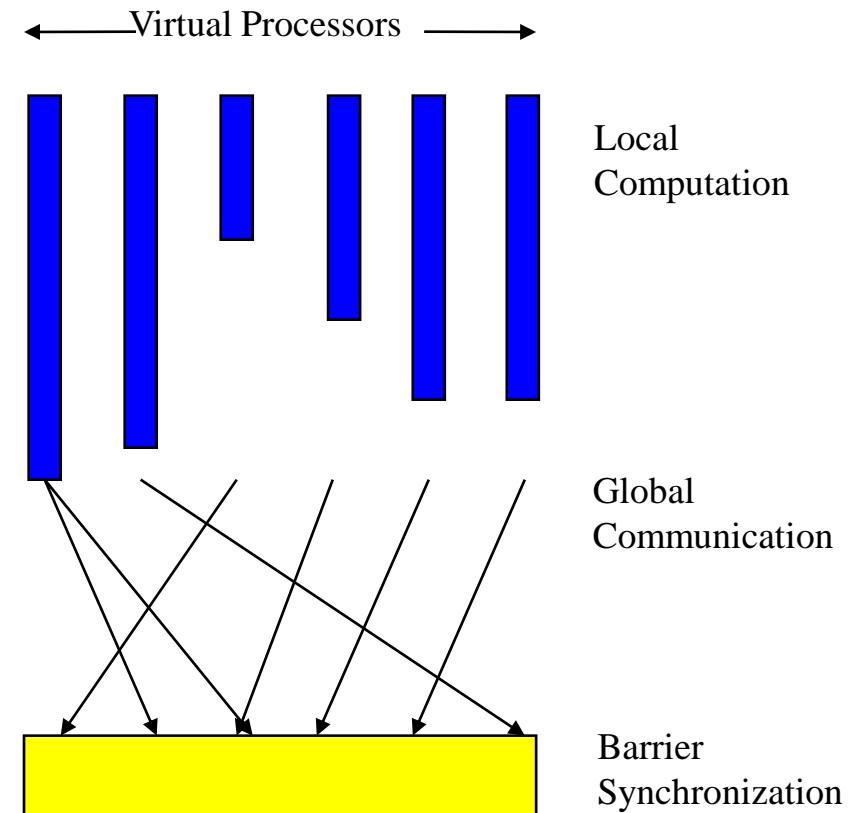
Continued..

- The *components* could be processors
- *The inter-connection network* could be router
- The *periodicity* parameter could be barrier.



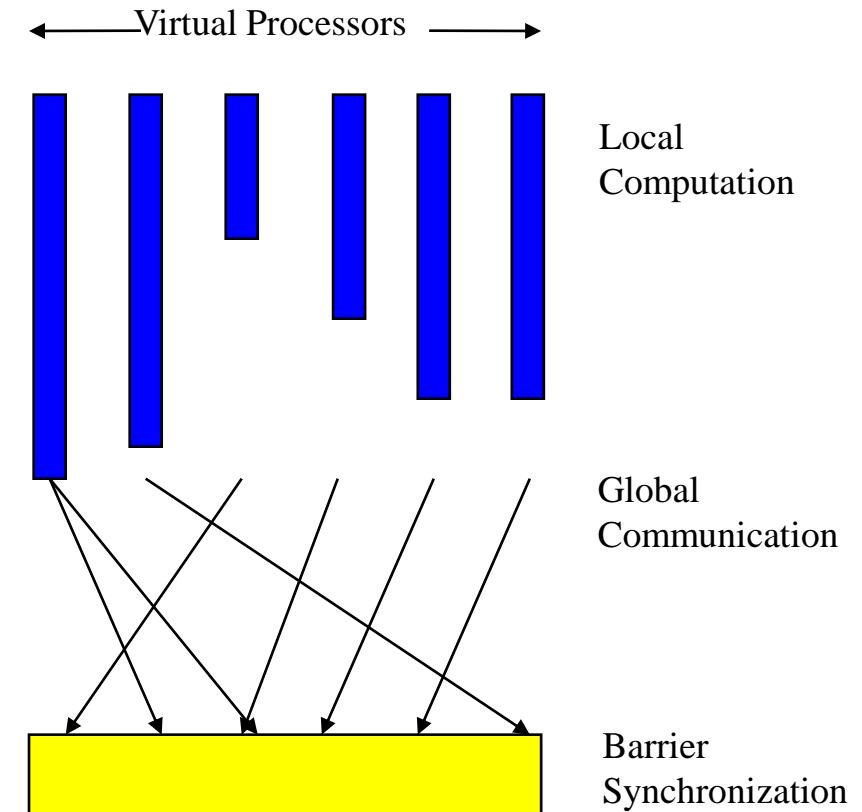
Computation on BSP Model

- A computation consists of **several supersteps**
- A **superstep** consists of:
 - A computation where each processor **uses only locally held values**
 - A global message transmission from each processor to any subset of others
 - A barrier synchronization



Computation on BSP Model

- At the end of a superstep, the transmitted messages become available as local data for the next *superstep*

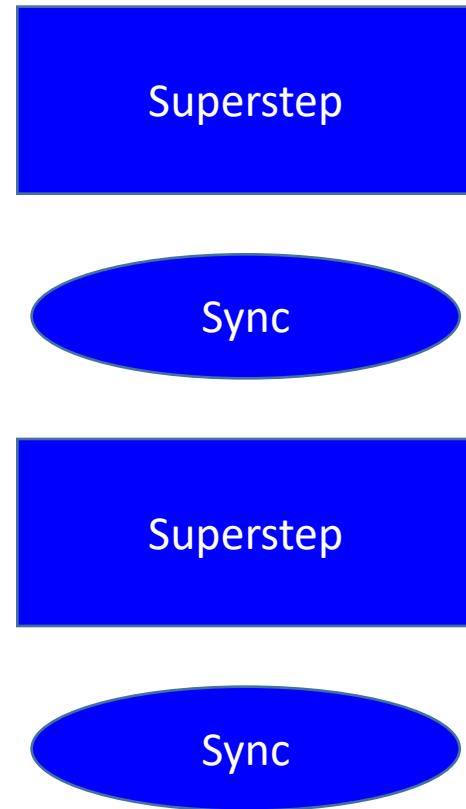


Communication on BSP Model

- A communication is always realized in a point-to point manner
 - Thus it is **not allowed** for multiple processes to read or write the same memory location **in the same cycle**
 - All memory and communication operations in a *superstep* must completely finish before any operation of the next *superstep* begins

Communication on BSP Model

- In BSP, each processor has local memory
 - “One-sided”* communication style is advocated
 - There are globally-known “symbolic addresses”
 - Data may be inconsistent until next barrier synchronization



*allow a process to access another process address space without any explicit participation in that communication operation by the remote process. One-sided *put* and *get* Direct Remote Memory Access (DRMA) calls, rather than paired two-sided *send* and *receive* message passing calls

The BSP Model

- Compute → Communicate → Synchronize → repeat
- The BSP computer is a MIMD system
- It is **loosely synchronous** at the ***superstep*** level
 - While the PRAM model was synchronous atwhich level??

The BSP Model

- Compute → Communicate → Synchronize → repeat
- The BSP computer is a MIMD system
- It is **loosely synchronous** at the ***superstep*** level
 - While the PRAM model was synchronous at **instruction level**
- Within a superstep, different processes execute asynchronously at their own paces

BSP Basics

- A BSP program runs in **supersteps**:
 1. Do local work (computation)
 2. Send/receive messages (communication)
 3. Wait until everyone is done (synchronization)
- The cost of a BSP program = **computation**
 + **communication**
 + **synchronization**.

Components (Processors)

- No need for programmers to manage memory, assign communication and perform low-level synchronization.
- This is achieved by programs written with **sufficient parallel slackness**.
- When programs written for **v virtual processors** are **run on p real processors** with $v \gg p$ (e.g. $v = p \log p$) then there is parallel slackness.
- Parallel slackness makes work distribution more balanced (than in cases such as $v = p$ OR $v < p$).

The BSP Model – w

- To account for load imbalance, the computation time **w is the maximum time** spent on computation operations by any processor

The BSP Model – h

- The BSP model **abstracts the communication operations** in a BSP superstep by the ***h-relation*** concept
- An ***h-relation*** is an abstraction of any communication operation, where each node **sends at most h words** to various nodes and each node **receives at most h words**

The BSP Model – gh

- Parameter g measures the permeability of the network to continuous traffic to uniformly random destinations
 - The parameter g is defined such that an **h -relation** will be delivered in time gh
 - The communication overhead is gh cycles, where g is the proportional coefficient for realizing an **h -relation**
- The value of g is platform-dependent, **but independent of the communication pattern**
 - In other words, gh is the time to execute the most time-consuming **h -relation**

The BSP Model – mg

- BSP does not distinguish between sending 1 message of length m , or m messages of length 1
 - Cost is mg

The BSP Model – I

- The synchronization overhead is I , which has a lower bound of the communication network latency (i.e., the time for a word to propagate through the physical network) and is always greater than zero

Barrier

- “Often expensive and should be used as sparingly as possible”
 - Developers of BSP claim that barriers are not as expensive as they are believed to be in high performance computing community
- The cost of a barrier synchronization **has two parts**
 - The cost caused by the variation in the completion time of the computation steps that participate
 - **The cost of reaching a globally-consistent state in all processors**

Barrier

- The parameter \mathcal{I} captures the latter of these costs
 - Lower bound on \mathcal{I} is the **diameter of the network**
 - However, it is also affected by many other factors, so that, in practice, an accurate value of \mathcal{I} for each parallel architecture is obtained **empirically**

The two parts of barrier cost

1. Variation in completion times (load imbalance):

1. If one processor is slower (more work, more messages, slower hardware), others must **wait**.
2. This is “waiting for the slowest processor.”

2. Reaching a globally consistent state:

1. Even if all processors finish at the same time, the system must ensure that:
 1. All **messages** sent in this superstep are delivered to the right processors.
 2. All processors agree that “**everyone is done**” and it’s safe to start the next superstep.
2. This requires **synchronization overhead**: exchanging small control signals, acknowledgments, or using a global clock.
3. In real systems, this is the **latency (l)** part of the BSP cost model.

Parameters (in simple words)

p → number of processors (workers).

w → **work** per processor in one superstep.

- **Example:** how many additions/multiplications each processor does locally.

h → number of **messages** a processor sends or receives in a superstep.

g → **gap per message** = cost of sending one word of data.

- If each message is 100 words long, cost = $100g$.

I → **latency** = time for barrier synchronization (the “global consistency overhead” cost).

- Like waiting for the slowest worker to arrive before moving on.

gh → total communication cost for a processor in a superstep.

- If a worker sends h words, each word costs g , total = gh .

The BSP Model

- h : communication time
- w : computation time
- I : synchronization time
- gh : communication overhead
- The time for a superstep is estimated by the sum
- ????

The BSP Model

- h : communication time
- w : computation time
- I : synchronization time (2nd part)
- gh : communication overhead
- The time for a superstep is estimated by the sum
 - $\text{Max}_i w_i + \text{Max}_i gh_i + I$

The BSP Model

- The BSP model **allows** the overlapping of the computation, the communication, and the synchronization operations within a superstep
 - If all three types of **operations are fully overlapped**, the time for a superstep becomes $\max(w, gh, l)$
 - However, the more conservative $w + gh + l$ is typically used

Example: Maximum of n element

- Algorithm to compute the maximum of a n-elements array.
On a BSP, since there is **no shared memory**, we have to say where the data are
 - $A[0..n-1]$ is distributed block-wise across **p** processors
 - For instance, each processor can have a portion of the array
 - n/p elements
- To describe an algorithm on a BSP machine, we have to define all supersteps
 - Local computing operations
 - Communication operations
 - Synchronization barrier

Maximum

- Superstep1
 - Local computation phase
 - $m = -\infty$;
 - for all $A[i]$ in my local partition of A , $m = \max(m, A[i])$;
 - Communication phase
 - if $\text{myPID} \neq 0$ send $(m, 0)$;
 - else // on P_0 :
 - for each i in $\{1..p-1\}$ recv (m_i, i) ;
- Superstep2
 - if $\text{myPID} = 0$ for each i in $\{1..p-1\}$ $m = \max(m, m_i)$

Maximum

- Superstep1

- Local computation phase **Time?**

- $m = -\infty$;
 - for all $A[i]$ in my local partition of A , $m = \max(m, A[i])$;

- Communication phase **Time?**

- if $\text{myPID} \neq 0$ send $(m, 0)$;
 - else // on P_0 :
 - for each i in $\{1..p-1\}$ recv (m_i, i) ;

- Superstep2

- if $\text{myPID} = 0$ for each i in $\{1..p-1\}$ $m = \max(m, m_i)$ **Time?**

Maximum

- Superstep1
 - Local computation phase **(n/p)**
 - $m = -\infty$;
 - for all $A[i]$ in my local partition of A , $m = \max(m, A[i])$;
 - Communication phase **Time?**
 - if $\text{myPID} \neq 0$ send $(m, 0)$;
 - else **// on P_0 :**
 - for each i in $\{1..p-1\}$ recv (m_i, i) ;
- Superstep2
 - if $\text{myPID} = 0$ for each i in $\{1..p-1\}$ $m = \max(m, m_i)$ **Time?**

Maximum

- Superstep1

- Local computation phase **(n/p)**

- $m = -\infty$;
 - for all $A[i]$ in my local partition of A , $m = \max(m, A[i])$;

- Communication phase **(gh, with $h = p-1$)**

- if $\text{myPID} \neq 0$ send $(m, 0)$;
 - else // on P_0 :
 - for each i in $\{1..p-1\}$ recv (m_i, i) ;

- Superstep2

- if $\text{myPID} = 0$ for each i in $\{1..p-1\}$ $m = \max(m, m_i)$ **Time?**

Maximum

- Superstep1
 - Local computation phase (n/p)
 - $m = -\infty;$
 - for all $A[i]$ in my local partition of A , $m = \max(m, A[i]);$
 - Communication phase (*gh*, with $h=p-1$, P_0 receives $p-1$ messages)
 - if $\text{myPID} \neq 0$ send $(m, 0);$
 - else $// \text{on } P_0:$
 - for each i in $\{1..p-1\}$ $\text{recv } (m_i, i);$
- Superstep2
 - if $\text{myPID} = 0$ for each i in $\{1..p-1\}$ $m = \max(m, m_i)$ **Time?**

Maximum

- Superstep1
 - Local computation phase (n/p)
 - $m = -\infty;$
 - for all $A[i]$ in my local partition of A , $m = \max(m, A[i]);$
 - Communication phase (*gh*, with $h=p-1$, P_0 receives $p-1$ messages)
 - if $\text{myPID} \neq 0$ send $(m, 0);$
 - else $// \text{on } P_0:$
 - for each i in $\{1..p-1\}$ $\text{recv } (m_i, i);$
- Superstep2
 - if $\text{myPID} = 0$ for each i in $\{1..p-1\}$ $m = \max(m, m_i) \quad p$

Maximum

- Total
- $\Theta(n/p + g(p-1) + l + p) = \Theta(n/p + gp + l)$

Example

- Algorithm for inner-product with 8 processors
- Given two arrays **x** and **y**, we want to compute $\sum x_i y_i$
- In a BSP program, it is crucial to define **how data are split among processors**
 - For instance, in this example, **how the vectors' elements can be divided?**

Example

- Algorithm for inner-product with 8 processors
- Given two arrays **x** and **y**, we want to compute $\sum x_i y_i$
- In a BSP program, it is crucial to define **how data are split among processors**
 - For instance, in this example, the vectors' elements can be **divided cyclically or in blocks**

	0	1	2	3	4	5	6	7	8	9
Cyclic:	P0	P1	P2	P3	P0	P1	P2	P3	P0	P1
Block:	P0	P0	P0	P1	P1	P1	P2	P2	P2	P3

- In any case, it is better having both x_i and y_i on the same processor!

Example

- Algorithm for inner-product using 8-processor BSP computer in 4 supersteps (“small” communication):
- Superstep 1
 - Computation?
 - Communication?
 - Barrier synchronization

Example

- Algorithm for inner-product using 8-processor BSP computer in 4 supersteps (“small” communication):
- Superstep 1
 - Computation: Each processor computes its local sum in $w = 2N/8$ time (actually $2N-1/8$) (*N multiplications + N-1 additions*)
 - Communication: Processors 0, 2, 4, 6 send their local sums to processors 1, 3, 5, 7 respectively
 - Apply 1-relation here
 - Barrier synchronization

Example

- Superstep 2
 - Computation?
 - Communication?
 - Barrier synchronization

Example

- Superstep 2
 - Computation: Processors 1, 3, 5, 7 each perform one addition ($w = 1$)
 - Communication: Processors 1 and 5 send their intermediate results to processors 3 and 7 respectively
 - 1-relation is applied here
 - Barrier synchronization

Example

- Superstep 3
 - Computation?
 - Communication?
 - Barrier synchronization

Example

- Superstep 3
 - Computation: Processors 3 and 7 each perform one addition ($w = 1$)
 - Communication: Processor 3 sends its intermediate result to processor 7
 - Apply 1-relation here
 - Barrier synchronization

Example

- Superstep 4
 - Computation?
 - Communication?

Example

- Superstep 4
 - Computation: Processor 7 performs one addition ($w=1$) to generate the final sum
 - No more communication or synchronization is needed

Example

- The total execution time (8 processors) is?

Example

- The total execution time is (8 processors) is $2N/8 + 3g + 3l + 3$ cycles
- In general, the execution time is  supersteps on an p -processor BSP
 - **How much is the parallel time on PRAM computer with p processors?**

Example

- The total execution time is (8 processors) is $2N/8 + 3g + 3l + 3$ cycles
- In general, the execution time is $2N/p + (g+l+1)\log p$ cycles on an p -processor BSP
 - **How much is the parallel time on PRAM computer with p processors?**

Example

- The total execution time is (8 processors) is $2N/8 + 3g + 3l + 3$ cycles
- In general, the execution time is $2N/p + (g+l+1)\log p$ cycles on an p -processor BSP
- This is in contrast to the time $2N/p + \log p$ on a PRAM computer
 - The two extra terms, $\log p$ and $l \log p$ correspond to communication and synchronization overheads, respectively

Matrix Multiplication

- We want to multiply two matrices, A and B
 - $A_{(nxn)} \times B_{(nxn)} = C_{(nxn)}$
- The standard algorithm uses $p \leq n^2$ processors
 - If $p = n^2$, then each processor can compute the **value of a single element** in C

Matrix Multiplication

- **Each element** of C can be computed in parallel using **n** processors on a CREW PRAM
 - **O(log n)** parallel time
 - Basically, it's a SUM in parallel
- **All** c_{ij} can be computed in parallel using **n^3** processors in **O(log n)** time

Matrix Multiplication

- In the BSP model we need to find a way of dividing the input among processors, and to optimize the communication
- Since we have only p processors every processor gets n^2/p elements.
- To each processor we assign the sub-problem of computing a sub-matrix of C , of size $n/\sqrt{p} \times n/\sqrt{p}$
 - Each processor computes $n/\sqrt{p} \times n/\sqrt{p} = n^2/p$ elements of C
 - Thus, each processor receives in input n/\sqrt{p} rows of A and n/\sqrt{p} columns of B

Matrix Multiplication

Let $n = 4$

A =

a ₁₁	a ₁₂	a ₁₃	a ₁₄
a ₂₁	a ₂₂	a ₂₃	a ₂₄
a ₃₁	a ₃₂	a ₃₃	a ₃₄
a ₄₁	a ₄₂	a ₄₃	a ₄₄

B =

b ₁₁	b ₁₂	b ₁₃	b ₁₄
b ₂₁	b ₂₂	b ₂₃	b ₂₄
b ₃₁	b ₃₂	b ₃₃	b ₃₄
b ₄₁	b ₄₂	b ₄₃	b ₄₄

C = A \times B =

c ₁₁	c ₁₂	c ₁₃	c ₁₄
c ₂₁	c ₂₂	c ₂₃	c ₂₄
c ₃₁	c ₃₂	c ₃₃	c ₃₄
c ₄₁	c ₄₂	c ₄₃	c ₄₄

Matrix Multiplication

Let $p=4$ (p_1, p_2, p_3, p_4)

p_1 computes

c_{11}	c_{12}
c_{21}	c_{22}

p_2 computes

c_{13}	c_{14}
c_{23}	c_{24}

p_3 computes

c_{31}	c_{32}
c_{41}	c_{42}

p_4 computes

c_{33}	c_{34}
c_{43}	c_{44}

Matrix Multiplication

p_1 computes

c_{11}	c_{12}
c_{21}	c_{22}

with input

a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}

b_{11}	b_{12}
b_{21}	b_{22}
b_{31}	b_{32}
b_{41}	b_{42}

p_2 computes

c_{13}	c_{14}
c_{23}	c_{24}

with input

a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}

b_{13}	b_{14}
b_{23}	b_{24}
b_{33}	b_{34}
b_{43}	b_{44}

Matrix Multiplication

p_3 computes

c_{31}	c_{32}
c_{41}	c_{42}

with input

a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}

b_{11}	b_{12}
b_{21}	b_{22}
b_{31}	b_{32}
b_{41}	b_{42}

p_4 computes

c_{33}	c_{34}
c_{43}	c_{44}

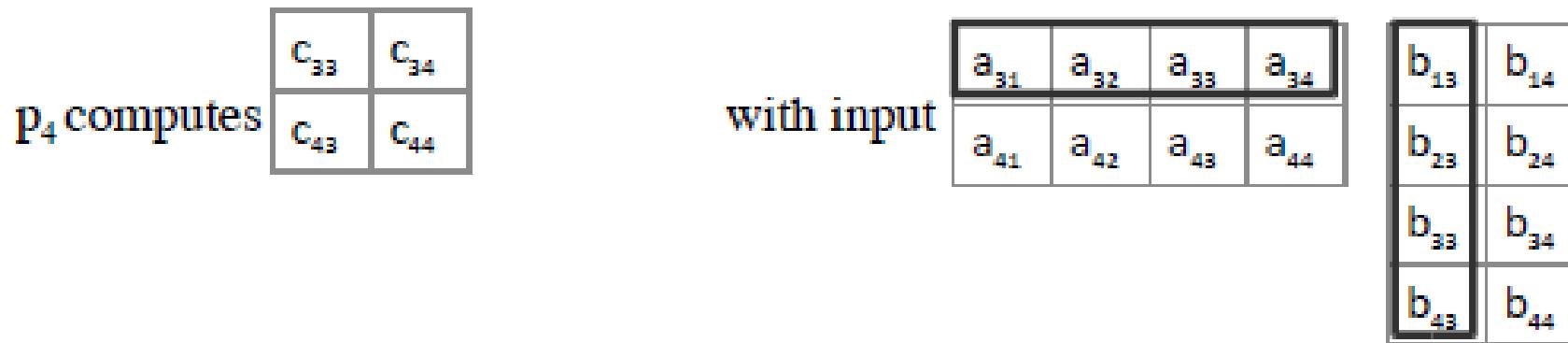
with input

a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}

b_{13}	b_{14}
b_{23}	b_{24}
b_{33}	b_{34}
b_{43}	b_{44}

Matrix Multiplication

- Let us compute the number of local operations performed by a processor, say p_4
 - Given a local row and a local column of p_4
 - »How many sums does it perform?
 - »How many multiplications does it perform?



Matrix Multiplication

- Let us compute the number of local operations performed by a processor, say p_4
 - Given **a local row and a local column** of p_4
 - »How many sums does it perform **$n-1$**
 - »How many multiplications does it perform **n**

p_4 computes

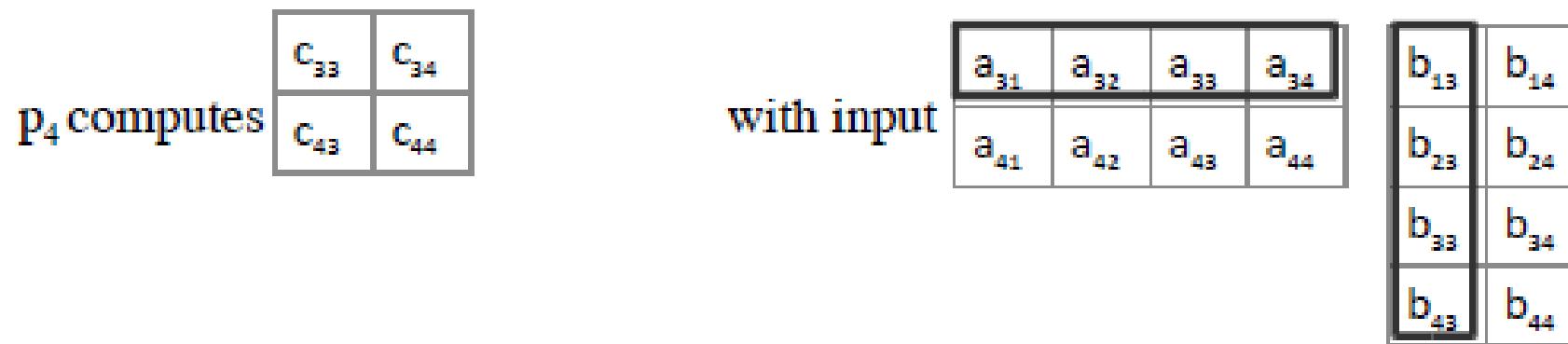
c_{33}	c_{34}
c_{43}	c_{44}

with input

a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}
b_{13}	b_{14}		
b_{23}	b_{24}		
b_{33}	b_{34}		
b_{43}	b_{44}		

Matrix Multiplication

- Let us compute the number of local operations performed by a processor, say p_4
 - How many row-by-column inner products p_4 does perform locally?**



Matrix Multiplication

- Let us compute the number of local operations performed by a processor, say p_4
 - Summing **over all inner products** performed by p_4
 - » How many sums does it perform?
 - » How many multiplications does it perform?

p_4 computes

c_{33}	c_{34}
c_{43}	c_{44}

with input

a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}

b_{13}	b_{14}
b_{23}	b_{24}
b_{33}	b_{34}
b_{43}	b_{44}

Matrix Multiplication

- Let us compute the number of local operations performed by a processor, say p_4
 - Summing **over all inner products** performed by p_4
 - » How many sums does it perform $(n-1) \times n/\sqrt{p} \times n/\sqrt{p} = (n-1)n^2/p$
 - » How many multiplications does it perform $n \times n/\sqrt{p} \times n/\sqrt{p} = n^3/p$

p_4 computes

C_{33}	C_{34}
C_{43}	C_{44}

with input

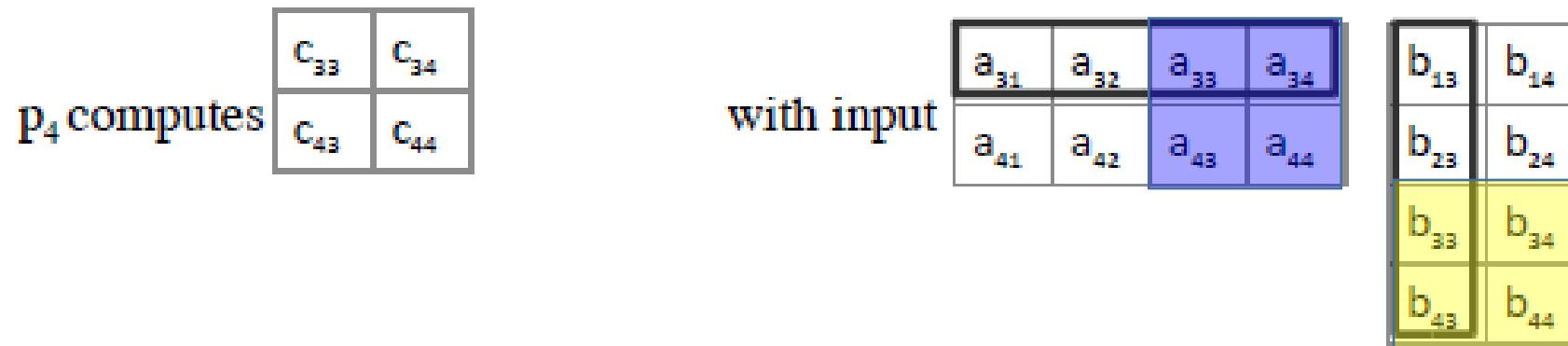
a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}
b_{13}	b_{14}		
b_{23}	b_{24}		
b_{33}	b_{34}		
b_{43}	b_{44}		

Matrix Multiplication

- Thus, each processor executes locally $(n-1)n^2/p$ sums + n^3/p multiplications
- That is, $(2n-1)n^2/p$ operations

Matrix Multiplication

- Now, let us analyze the complexity of the communication phase
 - In order to execute its local operations, how many messages does each processor needs to receive?

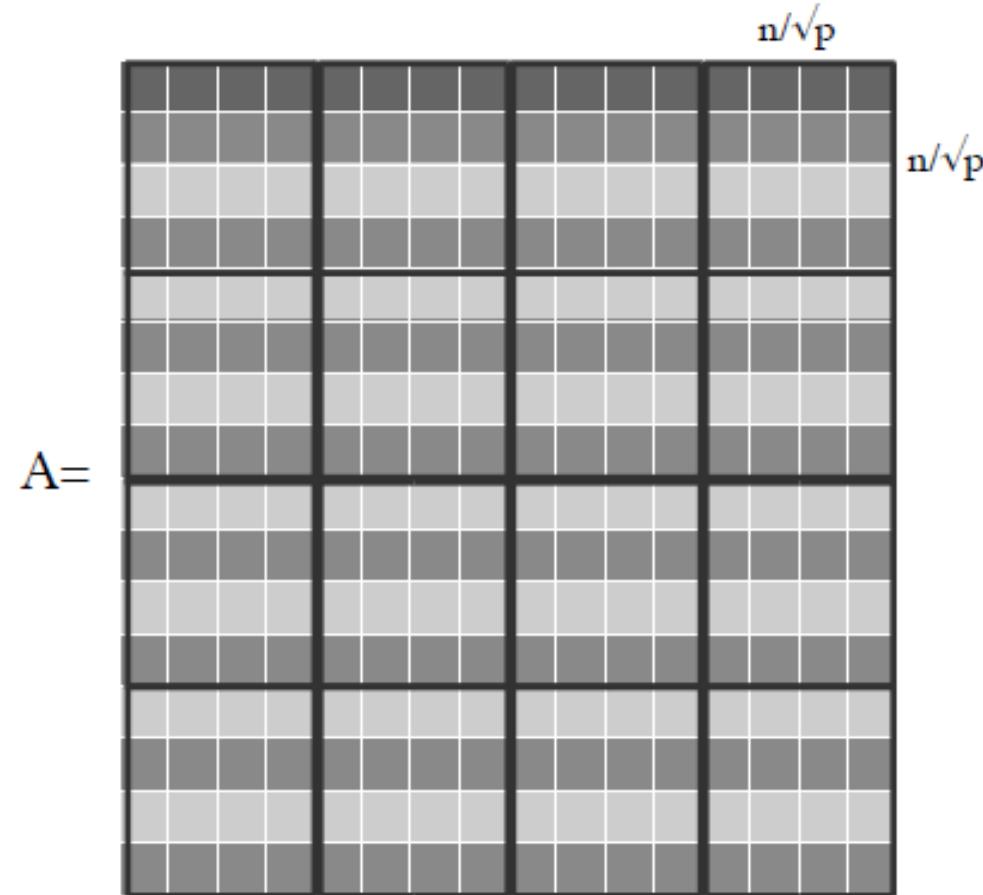


Matrix Multiplication

- How many of its local elements each processor needs to send, so that the other processors can receive the elements they need?
- For instance, to which processor p_2 has to send the elements of A it locally has?
 - To p_1

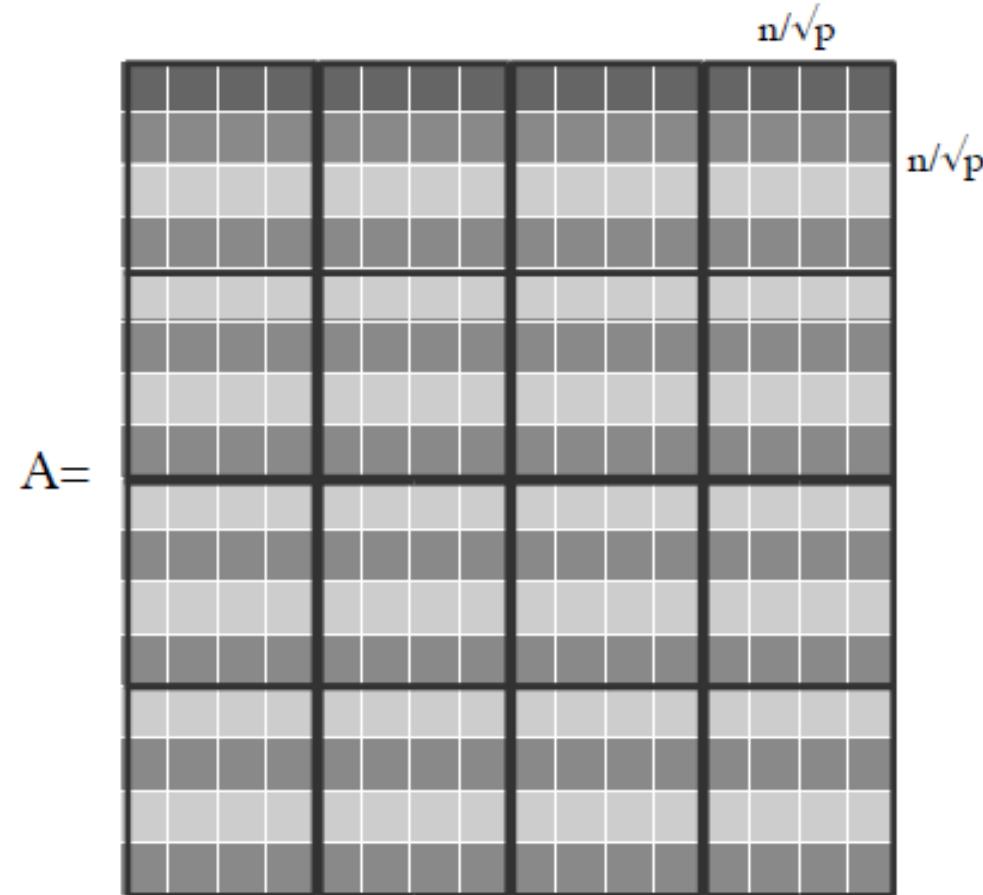
$$A = \begin{array}{|c|c|c|c|} \hline a_{11} & a_{12} & a_{13} & a_{14} \\ \hline a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ \hline a_{41} & a_{42} & a_{43} & a_{44} \\ \hline \end{array}$$

Matrix Multiplication



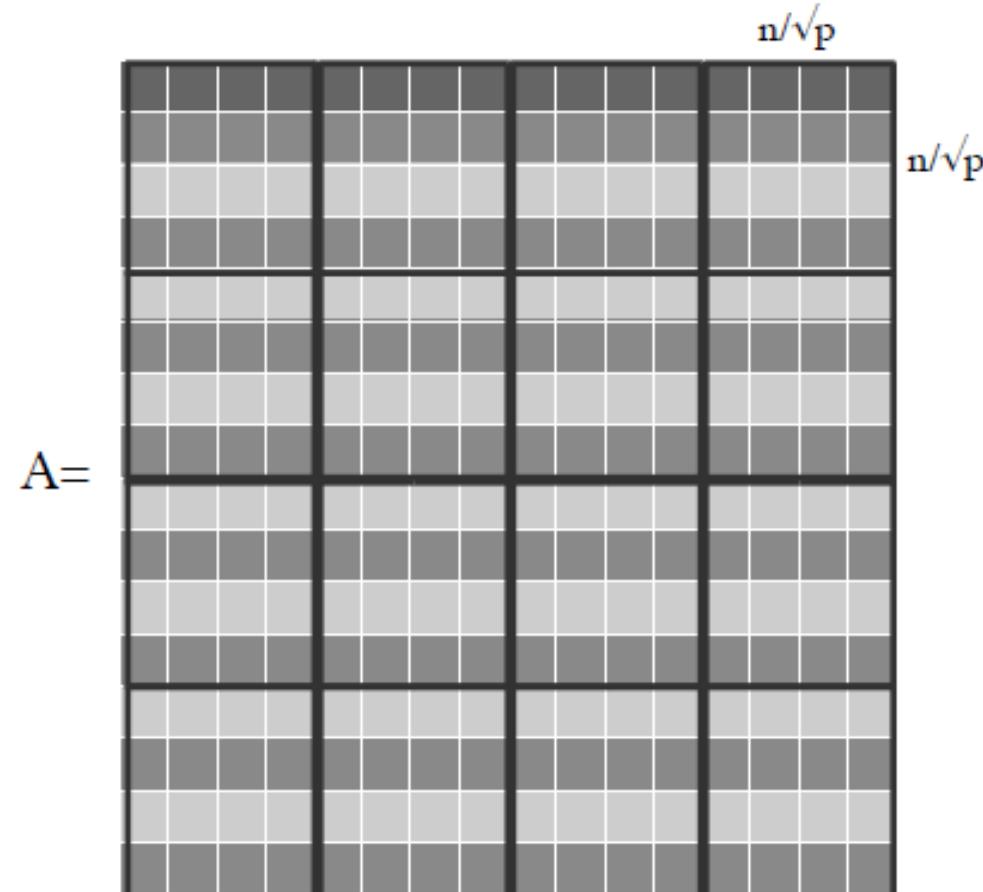
- In general, each processor has to send each one of its local values to how many processors?

Matrix Multiplication



- In general, each processor has to send each one of its local values **to how many processors?**
 - **\sqrt{p} (at most)**
 - So, how many messages will send each processor in total?

Matrix Multiplication



- In general, each processor has to send each one of its local values to how many processors?
 - **\sqrt{p} (at most)**
 - So, how many messages will send each processor in total?
 - **$(2n^2/p) \times \sqrt{p}$ (From A and B)**

Matrix Multiplication

- Clearly, we cannot expect to have the elements spread over the processors exactly as we need!!
- Thus, we can assume that the elements of A and B are uniformly distributed among processors
 - $2n^2/p$ for each processor
 - Each processor *replicates locally* each one of its elements at most **\sqrt{p}** times

Matrix Multiplication

- Finally, \sqrt{p} processors send the appropriated replicated elements to the processors that need them
- Thus, the number of transmissions, for each processor, is this number of messages: $(2n^2/p) \times \sqrt{p} = 2n^2/\sqrt{p}$

Matrix Multiplication

- The cost of this BSP algorithm is
 - $(2n-1)n^2/p + (2n^2/p^{1/2})g + l$
- The optimal cost $O(n^3/p)$, with n^2/p memory for each processor, is achieved when
 - $g = O(n/p^{1/2})$
 - $l = O(n^3/p)$

Example

- let's take your BSP model matrix multiplication example with $n = 8$ (matrix size 8×8) and $p = 4$ (4 processors).

A

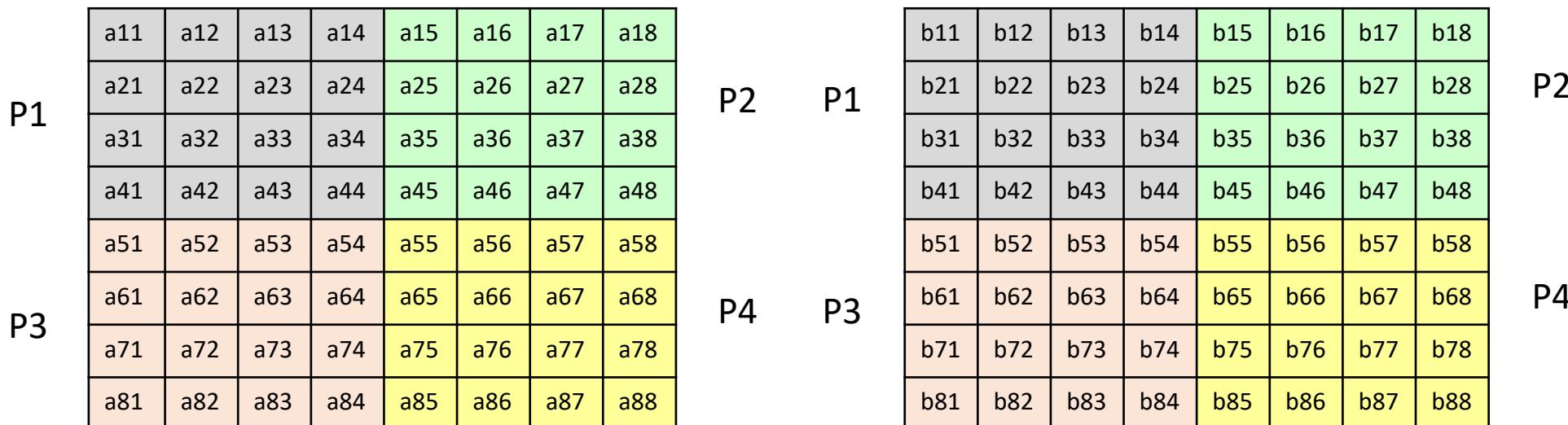
a11	a12	a13	a14	a15	a16	a17	a18
a21	a22	a23	a24	a25	a26	a27	a28
a31	a32	a33	a34	a35	a36	a37	a38
a41	a42	a43	a44	a45	a46	a47	a48
a51	a52	a53	a54	a55	a56	a57	a58
a61	a62	a63	a64	a65	a66	a67	a68
a71	a72	a73	a74	a75	a76	a77	a78
a81	a82	a83	a84	a85	a86	a87	a88

B

b11	b12	b13	b14	b15	b16	b17	b18
b21	b22	b23	b24	b25	b26	b27	b28
b31	b32	b33	b34	b35	b36	b37	b38
b41	b42	b43	b44	b45	b46	b47	b48
b51	b52	b53	b54	b55	b56	b57	b58
b61	b62	b63	b64	b65	b66	b67	b68
b71	b72	b73	b74	b75	b76	b77	b78
b81	b82	b83	b84	b85	b86	b87	b88

Step 1: Divide the work

- We have a matrix $C = A \times B$, size **8×8** , **$p = 4$ processors**, so each gets $n^2/p = 64/4 = 16$ elements of C .
- Each processor computes a sub-matrix of size $(n/\sqrt{p}) \times (n/\sqrt{p}) = (8/2) \times (8/2) = 4 \times 4$. So each processor works on a **4×4 block of C** .



Step 2: Input required

- To compute its 4×4 block of C, each processor needs:
 - 4 rows of A, 4 columns of B
- Processor **P1** (top-left block of C) needs: rows 1–4 of A and columns 1–4 of B.
- Processor **P2** (top-right block of C) needs: rows 1–4 of A and columns 5–8 of B.
- Processor **P3** (bottom-left block of C) needs: rows 5–8 of A and columns 1–4 of B.
- Processor **P4** (bottom-right block of C) needs: rows 5–8 of A and columns 5–8 of B.

	a11	a12	a13	a14	a15	a16	a17	a18								
	a21	a22	a23	a24	a25	a26	a27	a28								
P1	a31	a32	a33	a34	a35	a36	a37	a38								
	a41	a42	a43	a44	a45	a46	a47	a48								
	a51	a52	a53	a54	a55	a56	a57	a58								
P3	a61	a62	a63	a64	a65	a66	a67	a68								
	a71	a72	a73	a74	a75	a76	a77	a78								
	a81	a82	a83	a84	a85	a86	a87	a88								
	b11	b12	b13	b14	b15	b16	b17	b18								
	b21	b22	b23	b24	b25	b26	b27	b28								
P2	b31	b32	b33	b34	b35	b36	b37	b38								
	b41	b42	b43	b44	b45	b46	b47	b48								
	b51	b52	b53	b54	b55	b56	b57	b58								
P4	b61	b62	b63	b64	b65	b66	b67	b68								
	b71	b72	b73	b74	b75	b76	b77	b78								
	b81	b82	b83	b84	b85	b86	b87	b88								

Step 2: Input required

- To compute its 4×4 block of C, each processor needs:
 - 4 rows of A, 4 columns of B
- Processor **P1** (top-left block of C) needs: rows 1–4 of A and columns 1–4 of B.
- Processor **P2** (top-right block of C) needs: rows 1–4 of A and columns 5–8 of B.
- Processor **P3** (bottom-left block of C) needs: rows 5–8 of A and columns 1–4 of B.
- Processor **P4** (bottom-right block of C) needs: rows 5–8 of A and columns 5–8 of B.

P1

P3

a11	a12	a13	a14	a15	a16	a17	a18
a21	a22	a23	a24	a25	a26	a27	a28
a31	a32	a33	a34	a35	a36	a37	a38
a41	a42	a43	a44	a45	a46	a47	a48
a51	a52	a53	a54	a55	a56	a57	a58
a61	a62	a63	a64	a65	a66	a67	a68
a71	a72	a73	a74	a75	a76	a77	a78
a81	a82	a83	a84	a85	a86	a87	a88

C

a11	a12	a13	a14	a15	a16	a17	a18
a21	a22	a23	a24	a25	a26	a27	a28
a31	a32	a33	a34	a35	a36	a37	a38
a41	a42	a43	a44	a45	a46	a47	a48
a51	a52	a53	a54	a55	a56	a57	a58
a61	a62	a63	a64	a65	a66	a67	a68
a71	a72	a73	a74	a75	a76	a77	a78
a81	a82	a83	a84	a85	a86	a87	a88

A

b11	b12	b13	b14	b15	b16	b17	b18
b21	b22	b23	b24	b25	b26	b27	b28
b31	b32	b33	b34	b35	b36	b37	b38
b41	b42	b43	b44	b45	b46	b47	b48
b51	b52	b53	b54	b55	b56	b57	b58
b61	b62	b63	b64	b65	b66	b67	b68
b71	b72	b73	b74	b75	b76	b77	b78
b81	b82	b83	b84	b85	b86	b87	b88

B

Step 2: Input required

- To compute its 4×4 block of C, each processor needs:
 - 4 rows of A, 4 columns of B
- Processor **P1** (top-left block of C) needs: rows 1–4 of A and columns 1–4 of B.
- Processor **P2** (top-right block of C) needs: rows 1–4 of A and columns 5–8 of B.
- Processor **P3** (bottom-left block of C) needs: rows 5–8 of A and columns 1–4 of B.
- Processor **P4** (bottom-right block of C) needs: rows 5–8 of A and columns 5–8 of B.

P1

P3

a11	a12	a13	a14	a15	a16	a17	a18
a21	a22	a23	a24	a25	a26	a27	a28
a31	a32	a33	a34	a35	a36	a37	a38
a41	a42	a43	a44	a45	a46	a47	a48
a51	a52	a53	a54	a55	a56	a57	a58
a61	a62	a63	a64	a65	a66	a67	a68
a71	a72	a73	a74	a75	a76	a77	a78
a81	a82	a83	a84	a85	a86	a87	a88

C

a11	a12	a13	a14	a15	a16	a17	a18
a21	a22	a23	a24	a25	a26	a27	a28
a31	a32	a33	a34	a35	a36	a37	a38
a41	a42	a43	a44	a45	a46	a47	a48
a51	a52	a53	a54	a55	a56	a57	a58
a61	a62	a63	a64	a65	a66	a67	a68
a71	a72	a73	a74	a75	a76	a77	a78
a81	a82	a83	a84	a85	a86	a87	a88

A

b11	b12	b13	b14	b15	b16	b17	b18
b21	b22	b23	b24	b25	b26	b27	b28
b31	b32	b33	b34	b35	b36	b37	b38
b41	b42	b43	b44	b45	b46	b47	b48
b51	b52	b53	b54	b55	b56	b57	b58
b61	b62	b63	b64	b65	b66	b67	b68
b71	b72	b73	b74	b75	b76	b77	b78
b81	b82	b83	b84	b85	b86	b87	b88

B

Step 2: Input required

- To compute its 4×4 block of C, each processor needs:
 - 4 rows of A, 4 columns of B
- Processor **P1** (top-left block of C) needs: rows 1–4 of A and columns 1–4 of B.
- Processor **P2** (top-right block of C) needs: rows 1–4 of A and columns 5–8 of B.
- Processor **P3** (bottom-left block of C) needs: rows 5–8 of A and columns 1–4 of B.
- Processor **P4** (bottom-right block of C) needs: rows 5–8 of A and columns 5–8 of B.

P1

P3

a11	a12	a13	a14	a15	a16	a17	a18
a21	a22	a23	a24	a25	a26	a27	a28
a31	a32	a33	a34	a35	a36	a37	a38
a41	a42	a43	a44	a45	a46	a47	a48
a51	a52	a53	a54	a55	a56	a57	a58
a61	a62	a63	a64	a65	a66	a67	a68
a71	a72	a73	a74	a75	a76	a77	a78
a81	a82	a83	a84	a85	a86	a87	a88

C

a11	a12	a13	a14	a15	a16	a17	a18
a21	a22	a23	a24	a25	a26	a27	a28
a31	a32	a33	a34	a35	a36	a37	a38
a41	a42	a43	a44	a45	a46	a47	a48
a51	a52	a53	a54	a55	a56	a57	a58
a61	a62	a63	a64	a65	a66	a67	a68
a71	a72	a73	a74	a75	a76	a77	a78
a81	a82	a83	a84	a85	a86	a87	a88

A

b11	b12	b13	b14	b15	b16	b17	b18
b21	b22	b23	b24	b25	b26	b27	b28
b31	b32	b33	b34	b35	b36	b37	b38
b41	b42	b43	b44	b45	b46	b47	b48
b51	b52	b53	b54	b55	b56	b57	b58
b61	b62	b63	b64	b65	b66	b67	b68
b71	b72	b73	b74	b75	b76	b77	b78
b81	b82	b83	b84	b85	b86	b87	b88

B

Step 2: Input required

- To compute its 4×4 block of C, each processor needs:
 - 4 rows of A, 4 columns of B
- Processor **P1** (top-left block of C) needs: rows 1–4 of A and columns 1–4 of B.
- Processor **P2** (top-right block of C) needs: rows 1–4 of A and columns 5–8 of B.
- Processor **P3** (bottom-left block of C) needs: rows 5–8 of A and columns 1–4 of B.
- **Processor P4 (bottom-right block of C)** needs: rows 5–8 of A and columns 5–8 of B.

P1

P3

a11	a12	a13	a14	a15	a16	a17	a18
a21	a22	a23	a24	a25	a26	a27	a28
a31	a32	a33	a34	a35	a36	a37	a38
a41	a42	a43	a44	a45	a46	a47	a48
a51	a52	a53	a54	a55	a56	a57	a58
a61	a62	a63	a64	a65	a66	a67	a68
a71	a72	a73	a74	a75	a76	a77	a78
a81	a82	a83	a84	a85	a86	a87	a88

C

a11	a12	a13	a14	a15	a16	a17	a18
a21	a22	a23	a24	a25	a26	a27	a28
a31	a32	a33	a34	a35	a36	a37	a38
a41	a42	a43	a44	a45	a46	a47	a48
a51	a52	a53	a54	a55	a56	a57	a58
a61	a62	a63	a64	a65	a66	a67	a68
a71	a72	a73	a74	a75	a76	a77	a78
a81	a82	a83	a84	a85	a86	a87	a88

A

b11	b12	b13	b14	b15	b16	b17	b18
b21	b22	b23	b24	b25	b26	b27	b28
b31	b32	b33	b34	b35	b36	b37	b38
b41	b42	b43	b44	b45	b46	b47	b48
b51	b52	b53	b54	b55	b56	b57	b58
b61	b62	b63	b64	b65	b66	b67	b68
b71	b72	b73	b74	b75	b76	b77	b78
b81	b82	b83	b84	b85	b86	b87	b88

B

Step 3: Local computation (per processor)

- Each processor has to compute **16 elements (4×4)**.
- Each element of C is an inner product of 8 terms.
- So, per processor:
 - Multiplications = $n \times 16 = 8 \times 16 = 128$
 - Additions = $(n-1) \times 16 = 7 \times 16 = 112$
- Total = **128 + 112 = 240 operations** per processor.
 - Multiplications = $n^3/p = 8^3/4 = 512/4 = 128$.
 - Additions (sums) = $(n-1)n^2/p = 7 \cdot 16 = 112$

Step 4: Communication

- Each processor does not initially have all the rows/columns it needs.
- So, processors must **share rows of A and columns of B** with others.
- **Who sends to whom?**
 - **P1:** Has rows 1–4 of A. Must share them with **P2** (since P2 also needs rows 1–4).
 - **P3:** Has rows 5–8 of A. Must share them with **P4**.
 - **P1:** Has columns 1–4 of B. Must share them with **P3**.
 - **P2:** Has columns 5–8 of B. Must share them with **P4**.

Step 4: Communication

- Each processor does not initially have all the rows/columns it needs.
- So, processors must **share rows of A and columns of B** with others.
- So:
- Each processor shares with **$\sqrt{p} = 2$ processors at most**.
- Messages per processor = $(2n^2/p) \times \sqrt{p} = (2 \times 64/4) \times 2 = 32 \times 2 = 64$
- 8 sends total; each processor sends exactly two blocks.
- Each processor shares with **$\sqrt{p} = 2$ processors at most**.

Matrix Multiplication

- There exists a more sophisticated algorithm, by McColl and Valiant, that solves the problem with less messages
 - $n^3/p + (n^2/p^{2/3})g + l$
- That is optimal when
 - $g = O(n/p^{1/3})$
 - $l = O(n^3/(p \log n))$

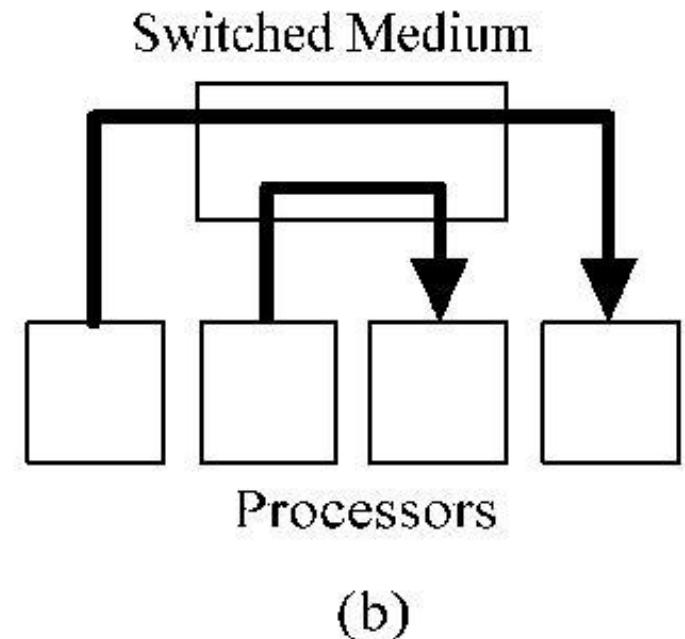
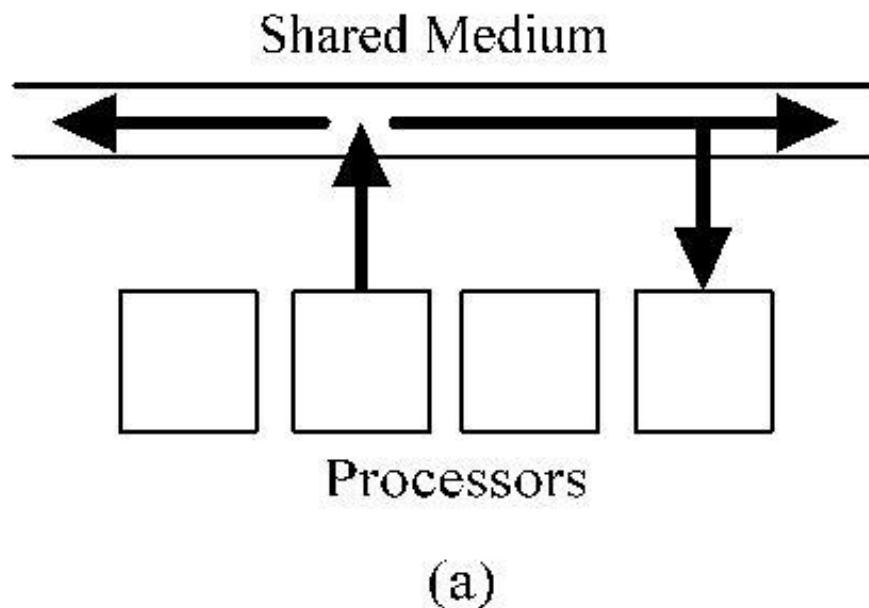
Slides prepared based on book by
M.J. Quinn et al.

Parallel programming in C with
MPI and OpenMP

Interconnection Networks

- Using interconnection networks we can
 - Connect processors to shared memory
 - Connect processors to each other
- Interconnection media types
 - Shared medium
 - Switched medium

Shared versus Switched Media



Shared Medium

- Allows **only message** at a time
- Messages are broadcast
- Each processor “**listens**” to every message
- Collisions require resending of messages
- Ethernet is an example

Switched Medium

- Supports **point-to-point messages** between pairs of processors
- Each processor has its own path to switch
- Advantages over shared media
 - Allows **multiple messages** to be sent simultaneously
 - Allows scaling of network to accommodate increase in processors

Switch Network Topologies

- View switched network as a **graph**
 - Vertices = processors **or switches**
 - Edges = communication paths
- Two kinds of topologies
 - Direct
 - Indirect

Direct Topology

- Ratio of switch nodes to processor nodes is 1:1
- Every switch node is connected to
 - 1 processor node
 - At least 1 other switch node

Indirect Topology

- Ratio of switch nodes to processor nodes is greater than 1:1
- Some switches simply connect other switches

Multiprocessors and Multicomputers

- Criteria to understand effectiveness in implementing efficient parallel algorithms on real architecture are:

1. Diameter: It is the largest distance between two nodes in the network. **Low diameter is better** as it puts a lower bound on the complexity of parallel algorithms.

2. Bisection width of the network: It is the minimum number of edges that must be removed in order to divide the network into two halves. **High bisection width is better**. Data set/Bisection width puts a lower bound on the complexity of parallel algorithms.

3. Number of edges per node: It is better if the number of edges per node **is a constant** independent of the network size. Processor organization scale well with a organization having more processors.

4. Maximum edge length: For better scalability, it is best if the nodes and edges are laid out in 3-D space so that the **maximum edge length is constant** independent of the network size.

Together they define **scalability** of the interconnection network, which directly impacts how efficiently parallel algorithms can run on real hardware.

Topological / Static Networks

- These are **direct topologies** where processors are laid out in space, and connections are mostly *deterministic and neighbor-based*.

Processor Organizations

Mesh Network: (1 of 8)

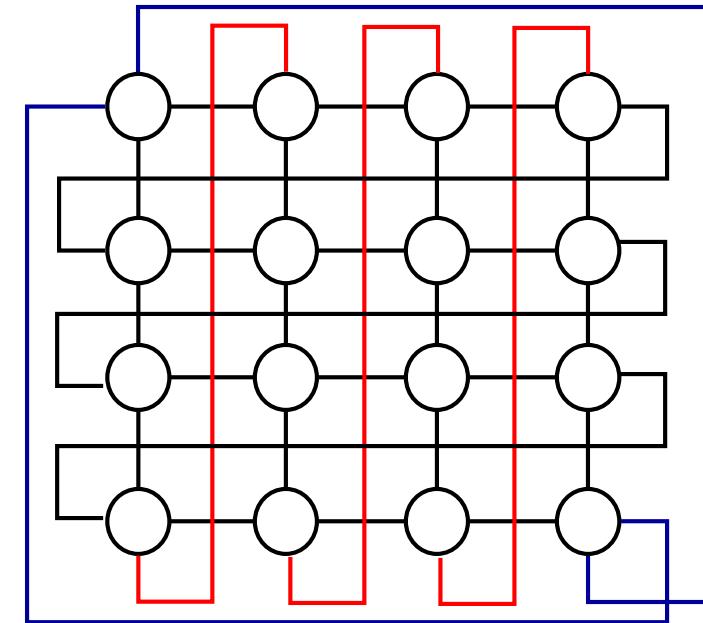
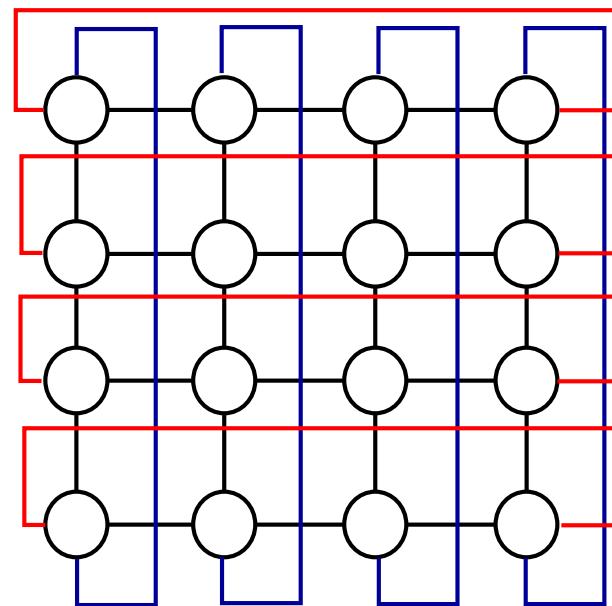
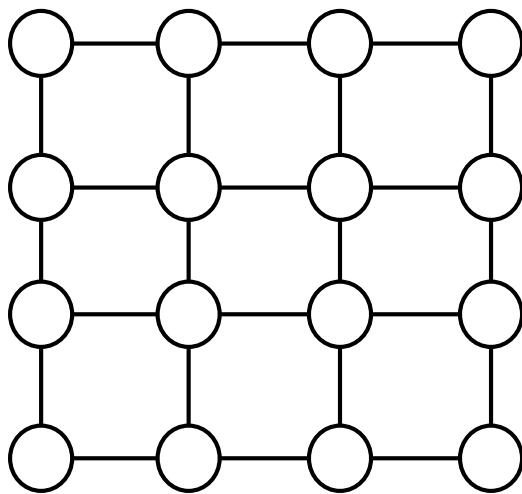
1. q-D lattice
2. Communication is allowed only between neighboring nodes
3. May allow wrap around connections
4. Diameter of a q-D mesh with k^q nodes is $q(k-1)$ (Difficult to get polylogarithmic time algorithm)



5. Bisection width of a q-D mesh with k^q nodes is k^{q-1}
6. Maximum edges per nodes is $2q$
7. Maximum edge length is a constant

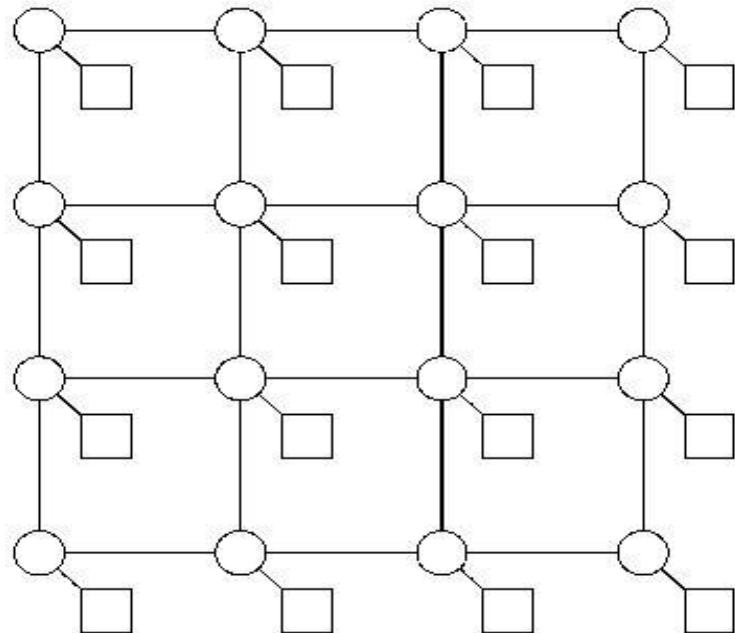


Ex. MarPar's MP-1, Intel's Paragon XP/S

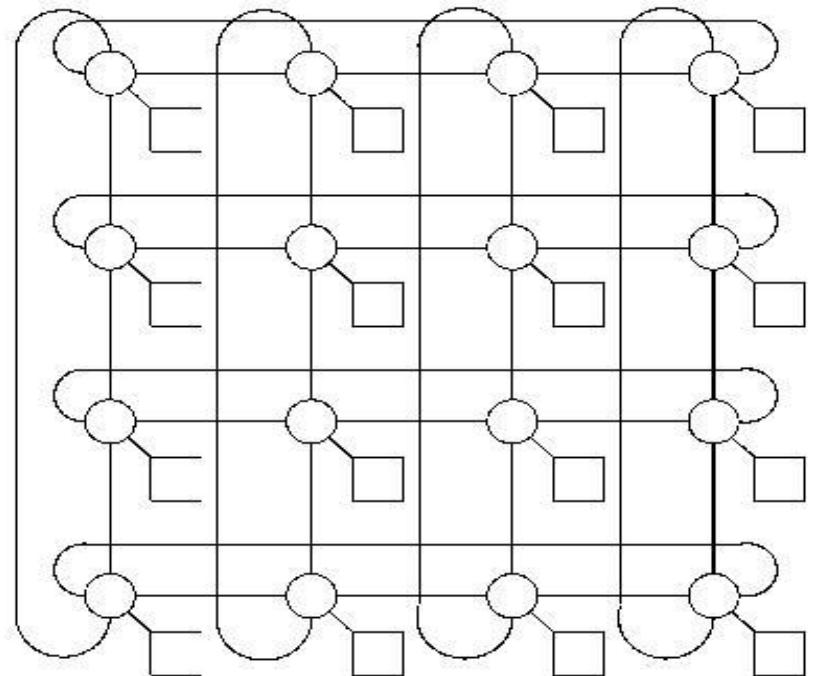


Mesh Networks

2-D Meshes



(a)

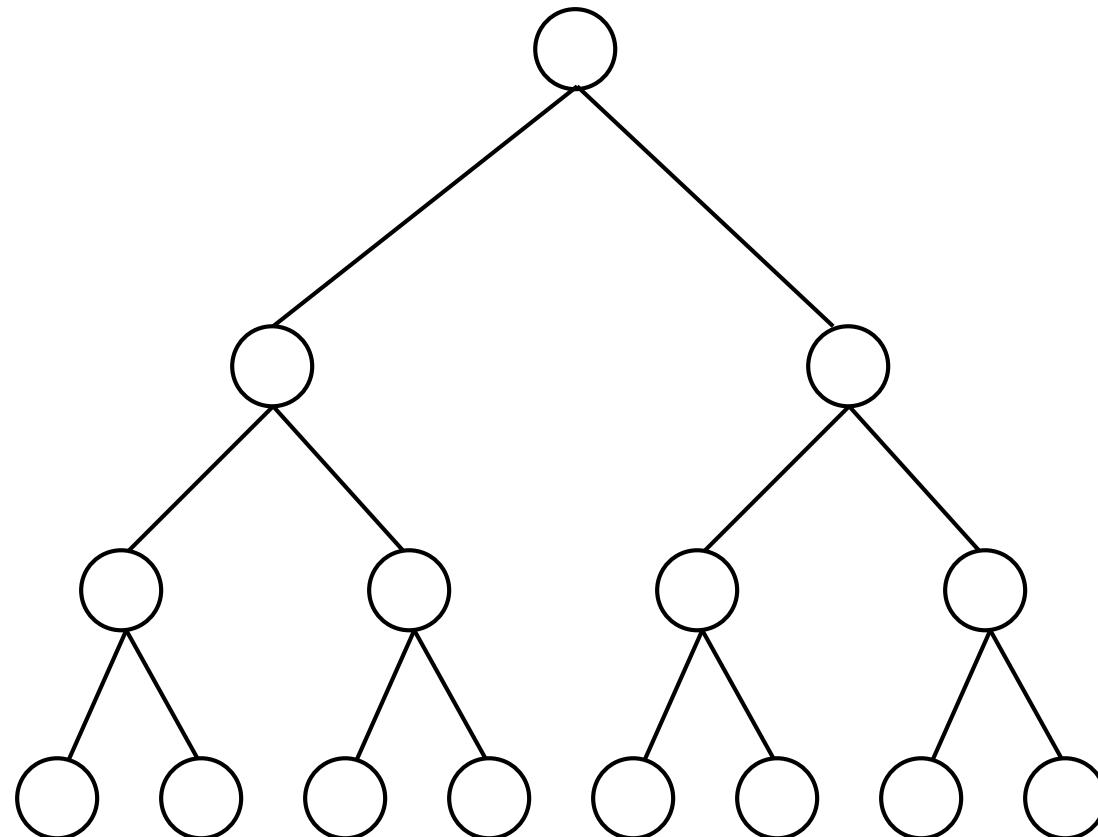


(b)

Binary tree: (2 of 8)

1. $2^k - 1$ nodes are arranged into a complete binary tree of depth k .
2. A node has at most 3 links
3. Low diameter of $2(k-1)$
4. **Poor bisection width**



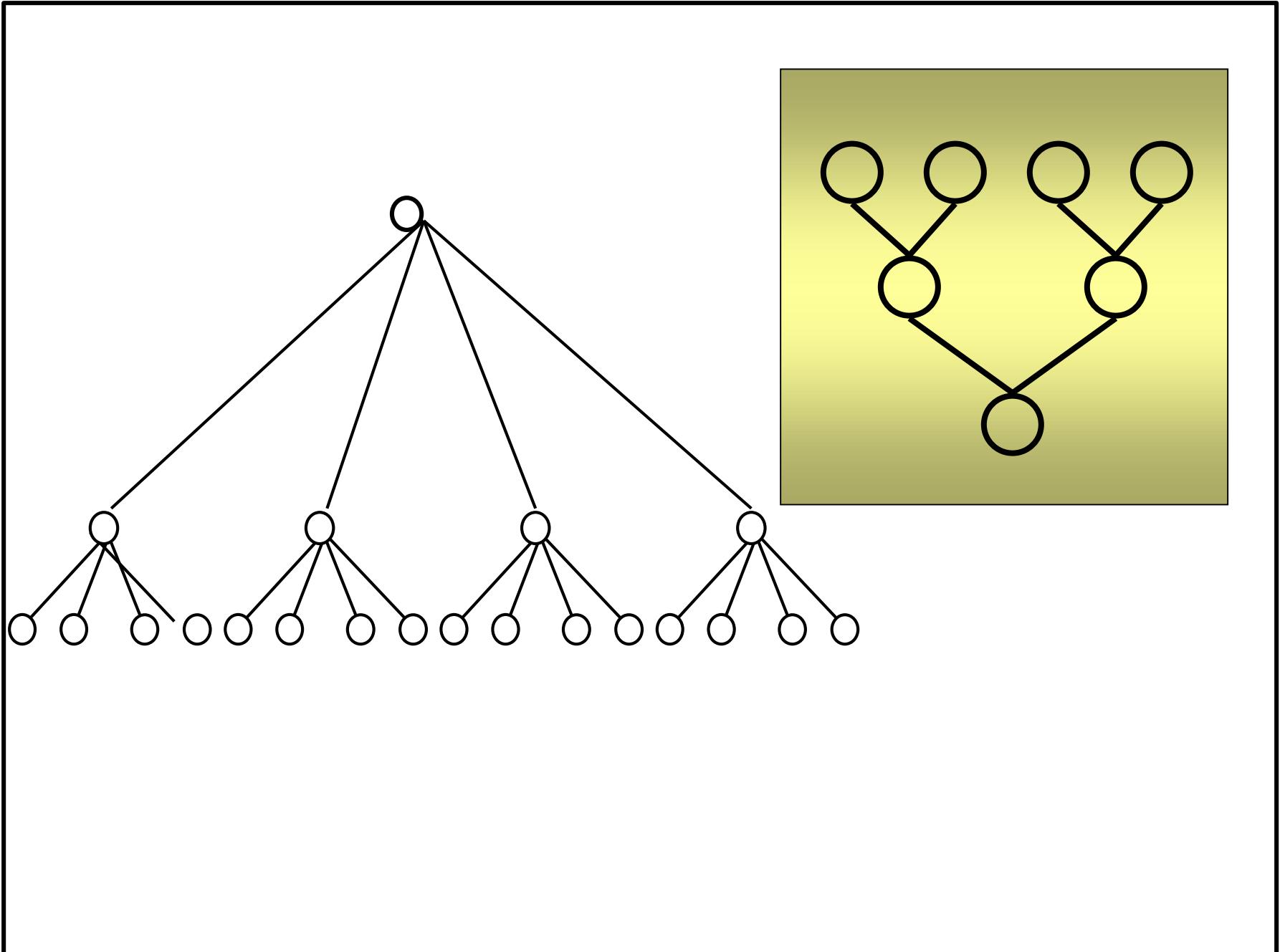


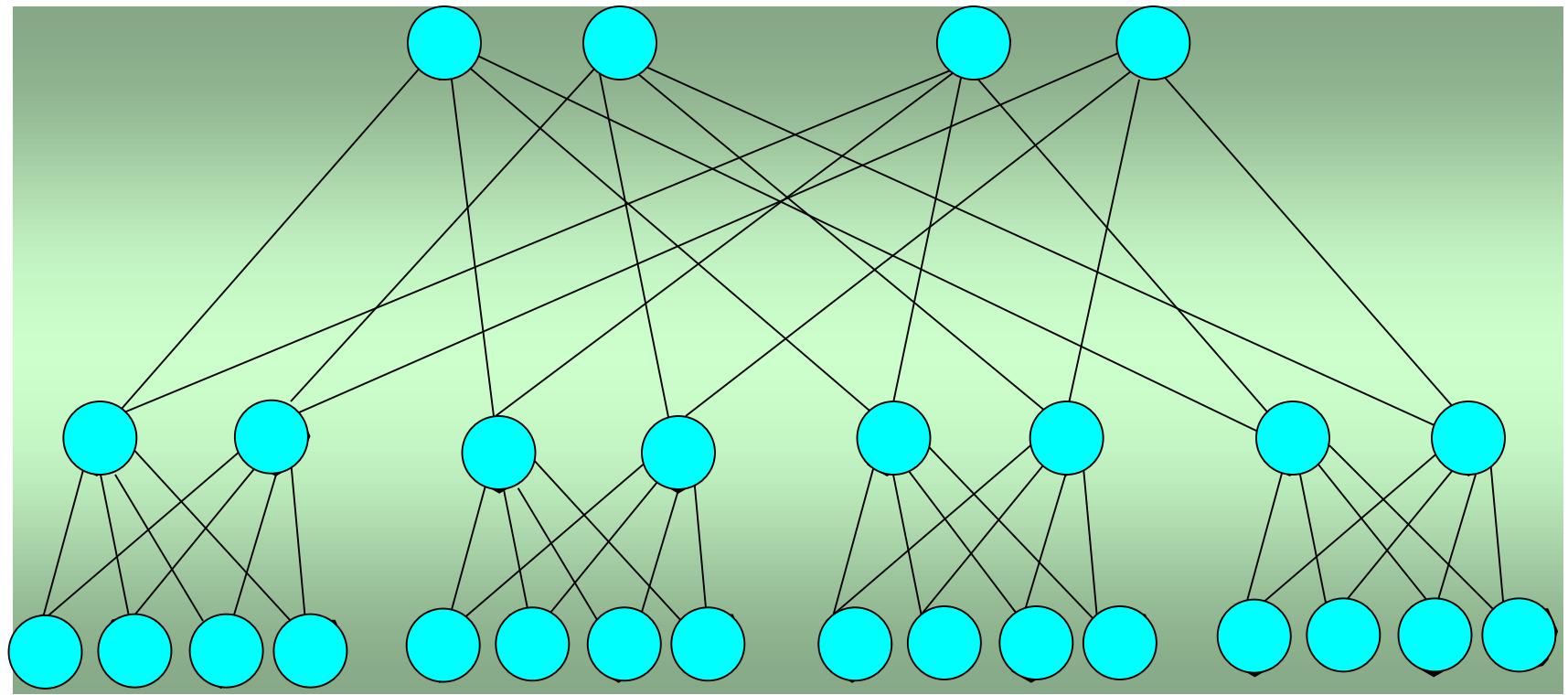
Tree Network

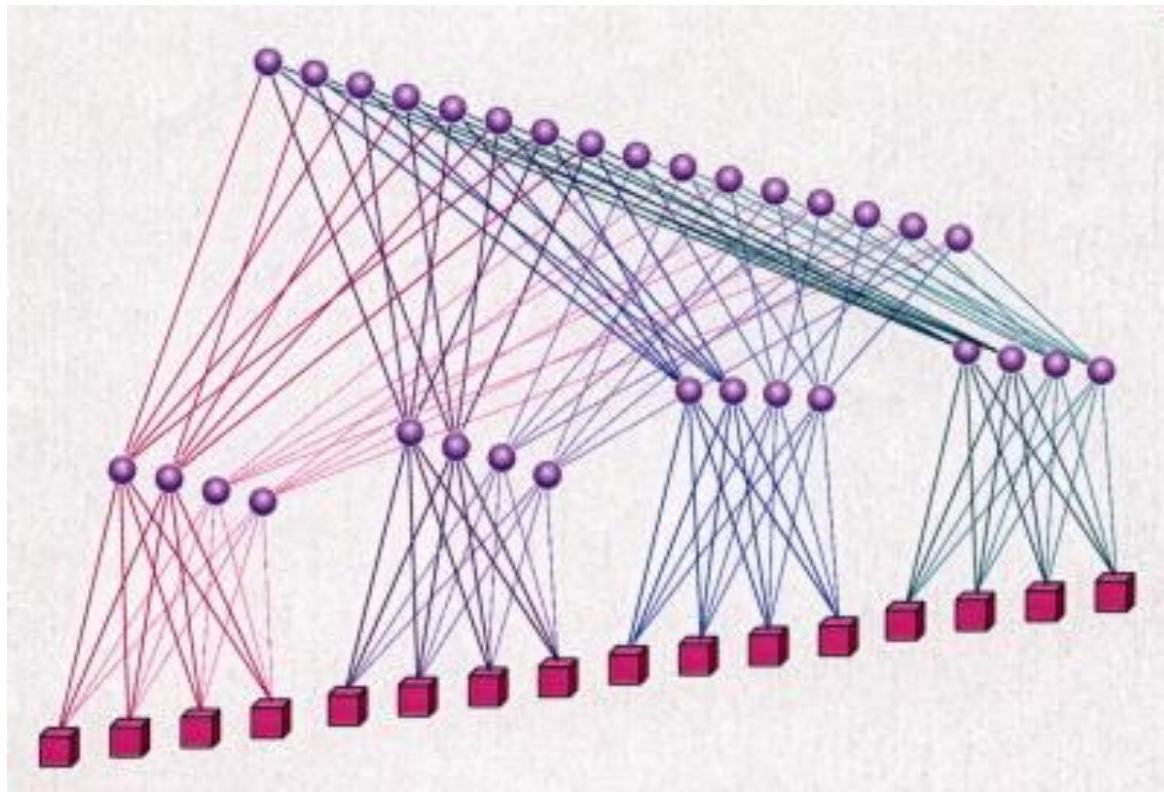
Hypertree Network: (Ex. data routine net of CM-5) (3 of 8)

1. Low diameter of binary tree with Improved bisection width
2. A 4-ary hypertree with depth d has 4^d leaves and $2^d(2^{d+1}-1)$ nodes
3. Diameter is $2d$ and bisection width is 2^{d+1}
4. No. of edges per node is never more than 6
5. Maximum edge length is an increasing function of the problem size.





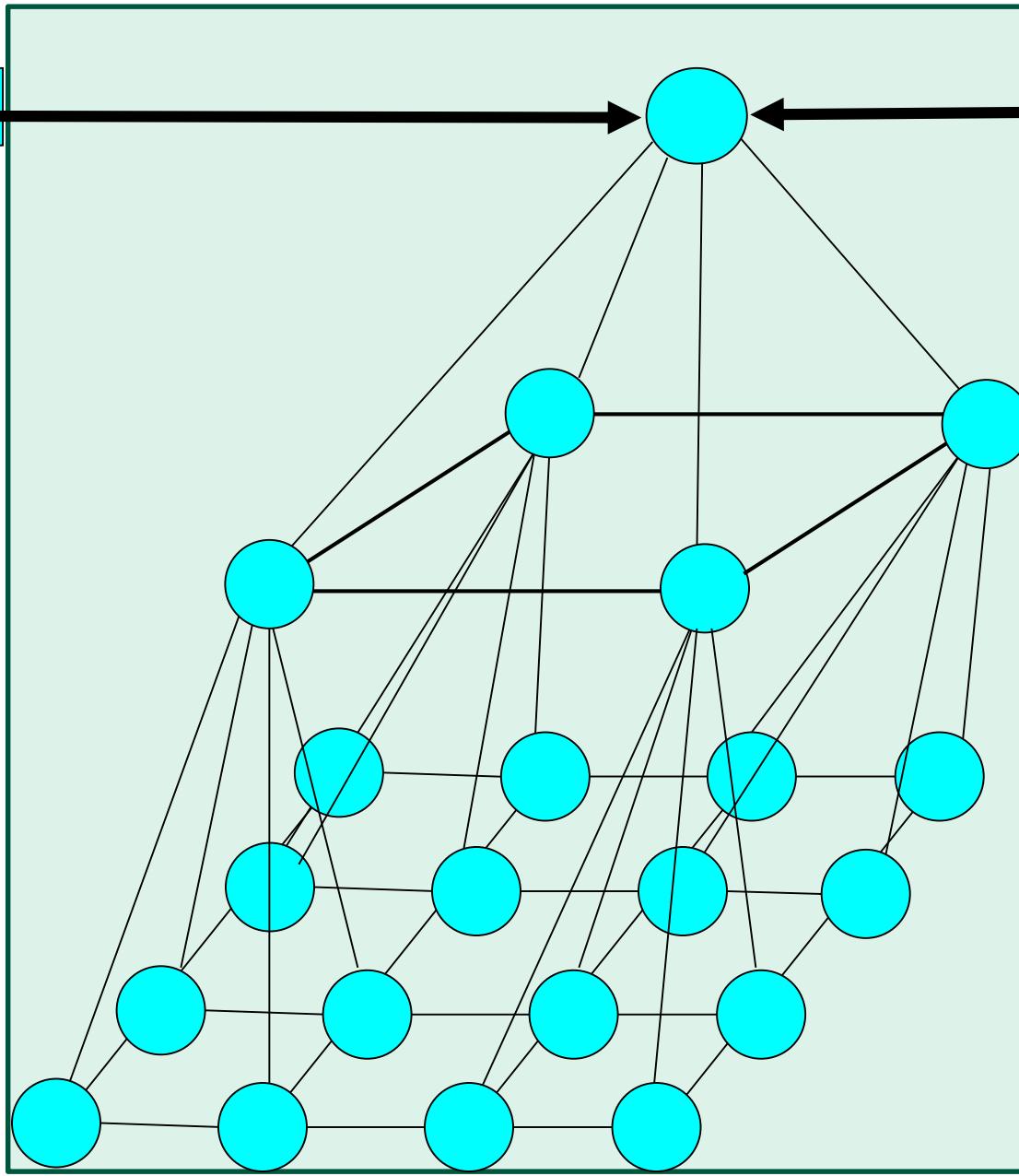




Pyramid Network: (4 of 8)

1. Mesh Network + Tree Network
2. Network of size k^2 is a complete 4-ary rooted tree of height $\log_2 k$
3. Total no. of processors of size k^2 is $(4/3)k^2 - (1/3)$
4. Level of the base is 0, apex of the pyramid has level $\log_2 k$.
5. Every interior processor is connected to 9 other processors
6. Pyramid reduces the diameter, $2 \log k$
7. Bisection width is $2k$





Multistage Interconnection Networks (MINs)

- These are **indirect topologies** designed for **high-performance routing**.
- Processors do not connect directly to all processors instead, messages pass through *stages of switch nodes*.

Butterfly Network: (Ex. BBN TC2000) (5 of 8)

1. It consists of $(k+1)2^k$ nodes divided into $k+1$ rows or **ranks**
2. Each row contains 2^k nodes
3. If node(i,j) denotes j^{th} node on i^{th} rank $0 \leq i \leq k$ and $0 \leq j < n$ then node(i,j) on rank $i>0$ is connected to two nodes on rank $i-1$, nodes $(i-1,j)$ and $(i-1,m)$, where m is the integer found by inverting the i^{th} **msb** in binary representation of j .

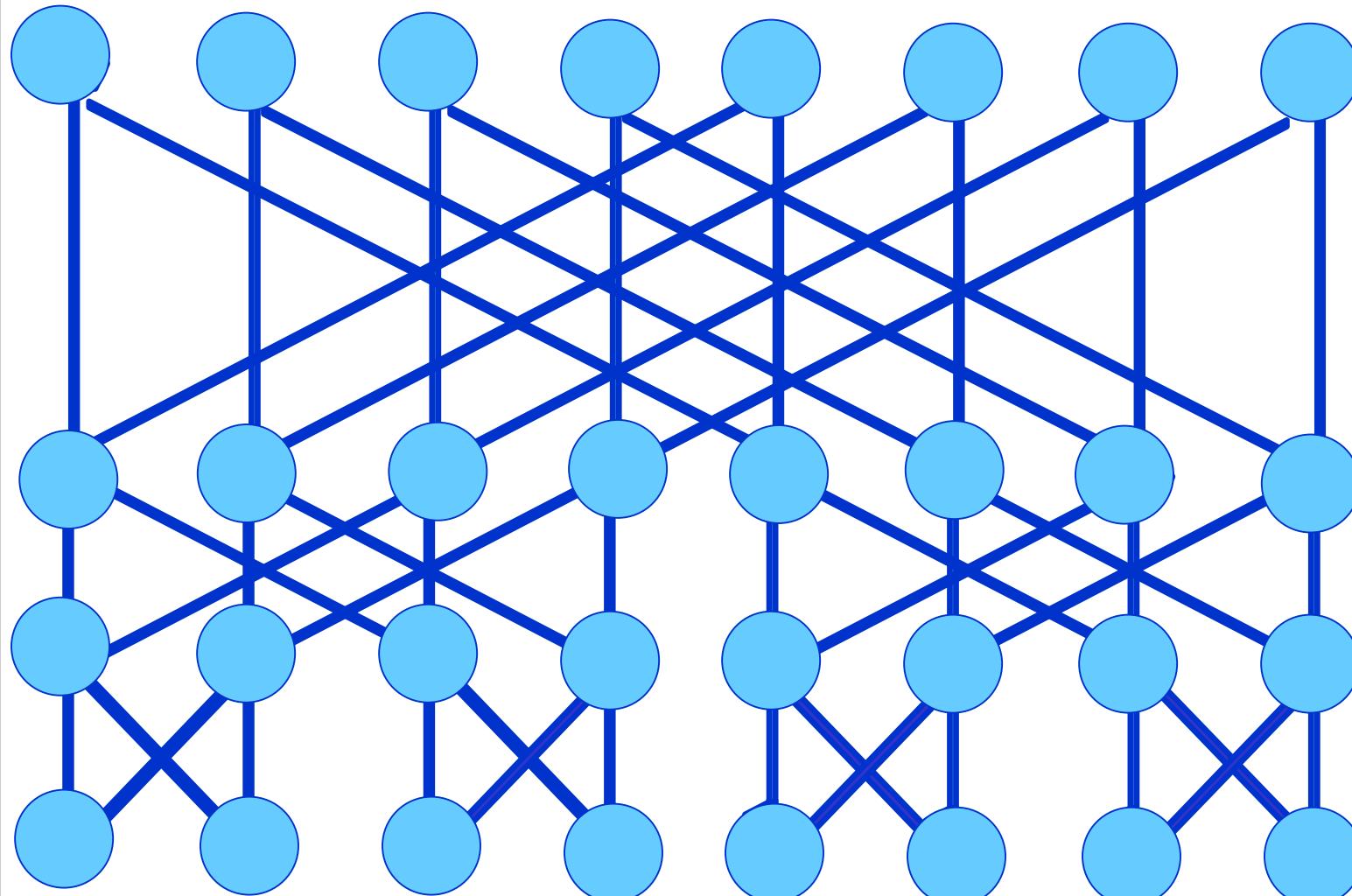
4. Diameter of the net is $2k$



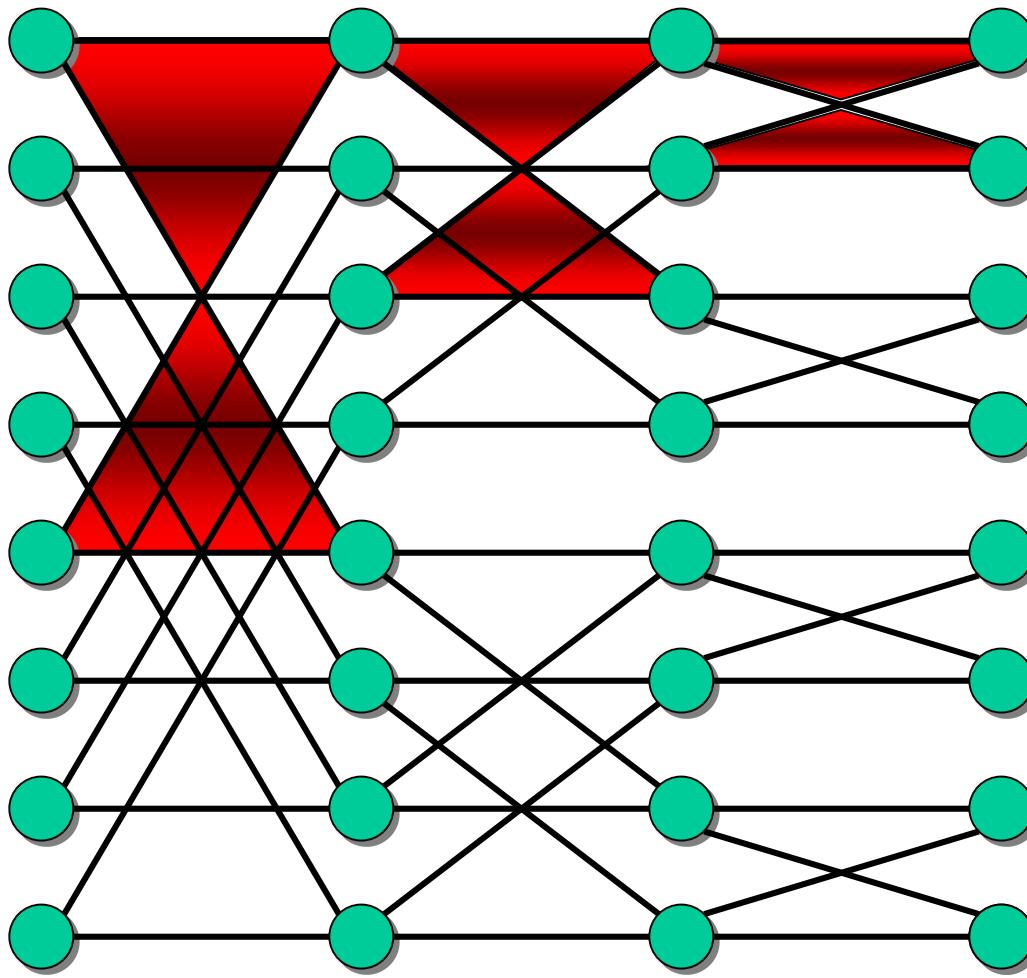
5. Bisection width is 2^{k-1}



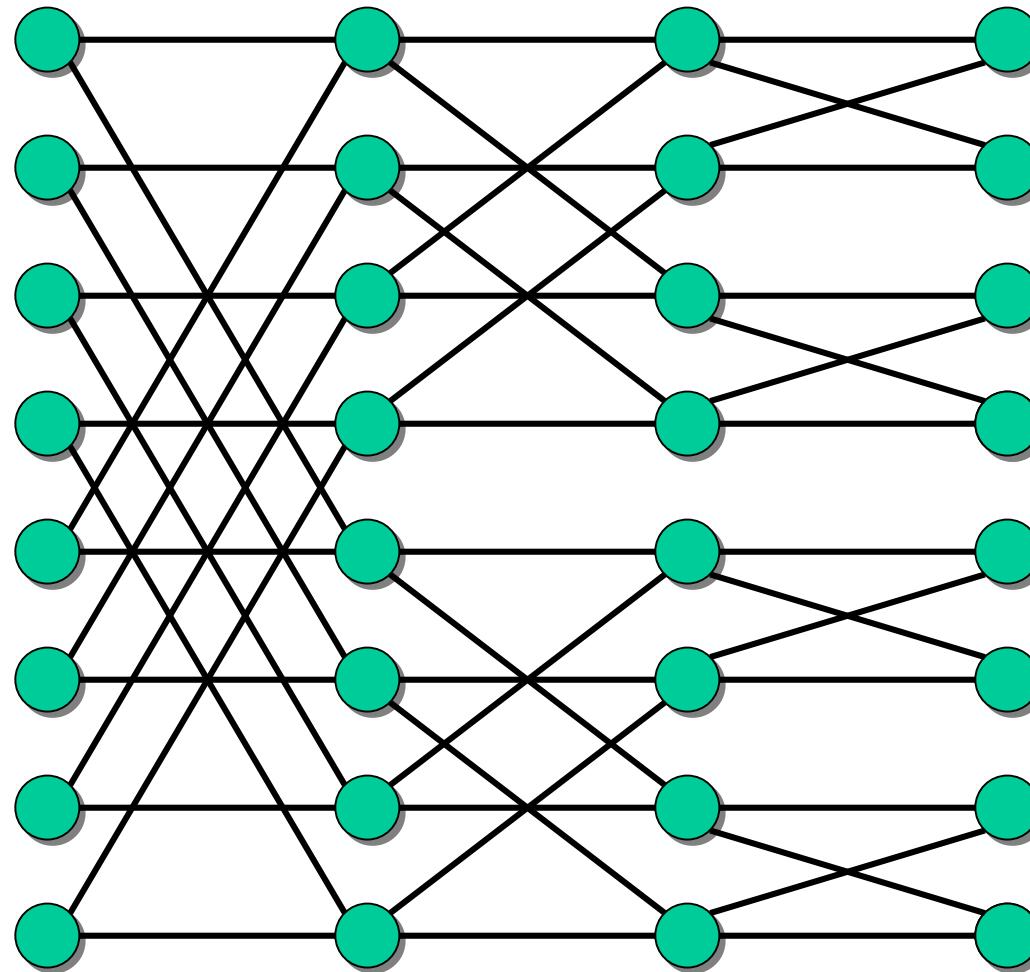
$k = 3, k + 1 = 4, 8$ nodes in each row



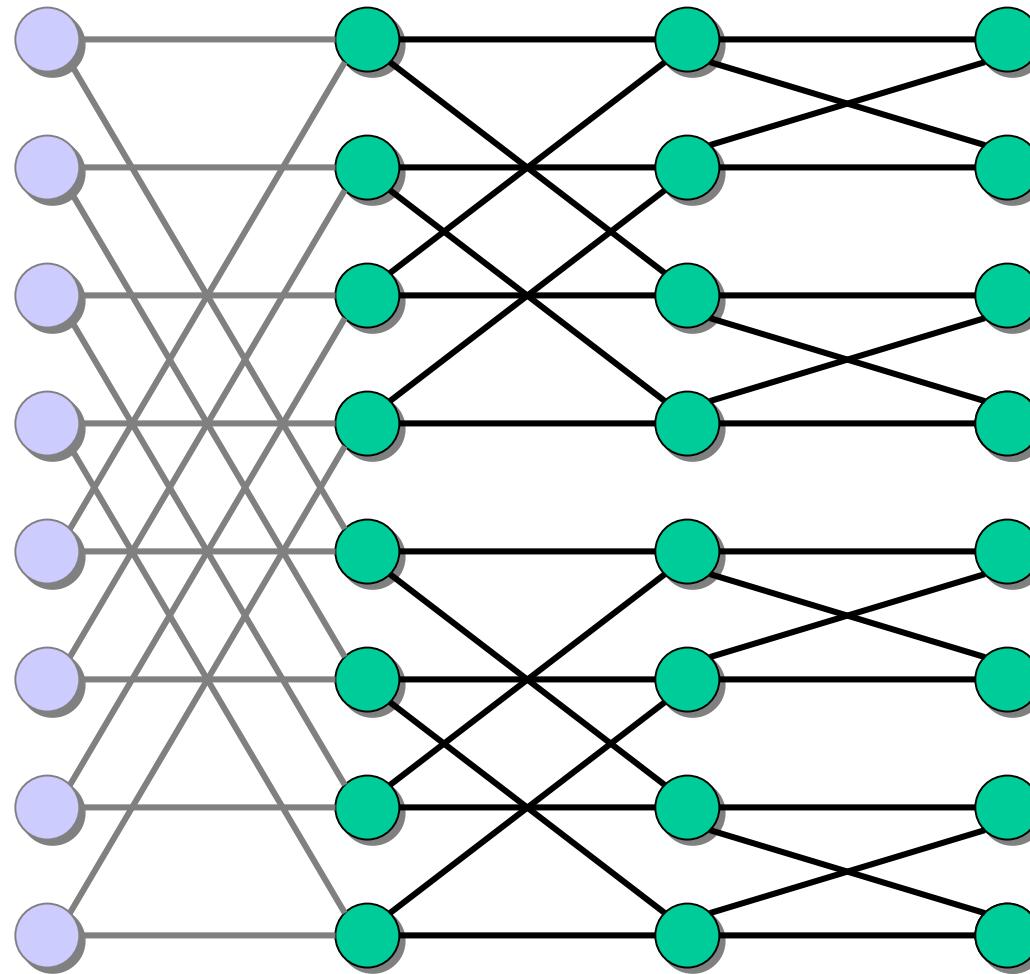
Butterflies



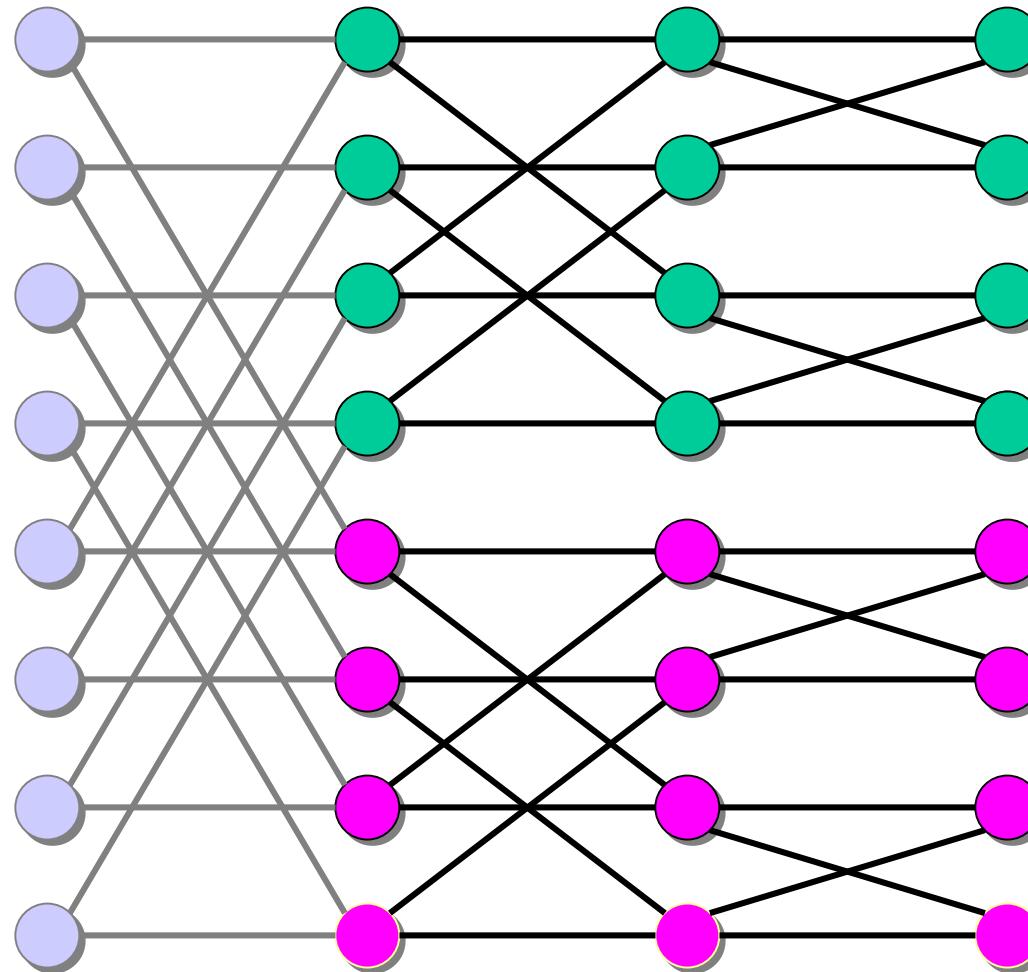
Decomposing a Butterfly



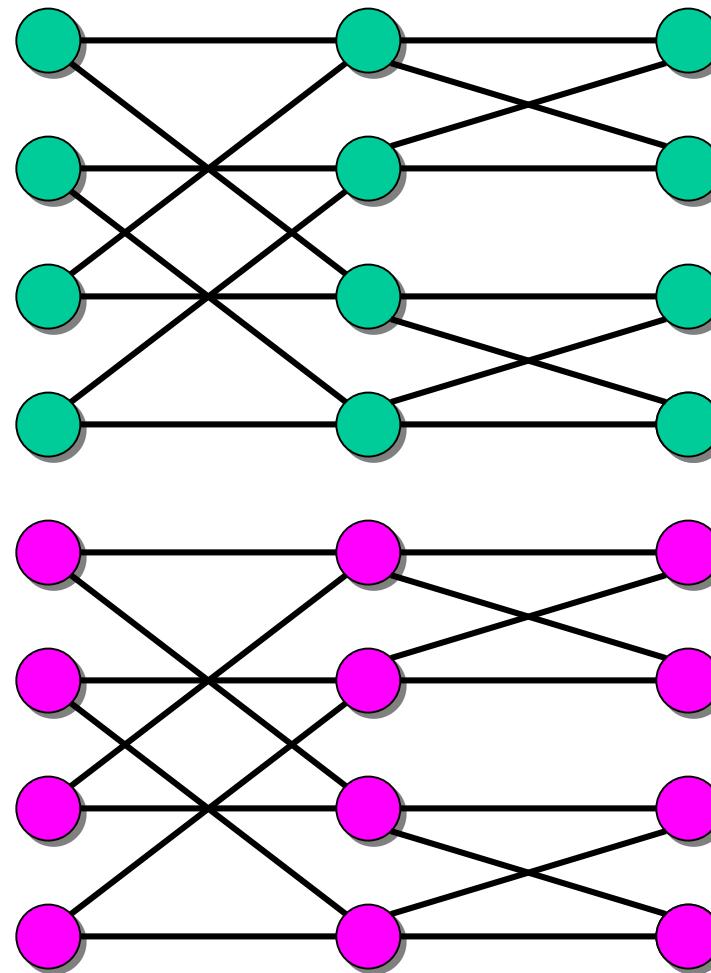
Decomposing a Butterfly



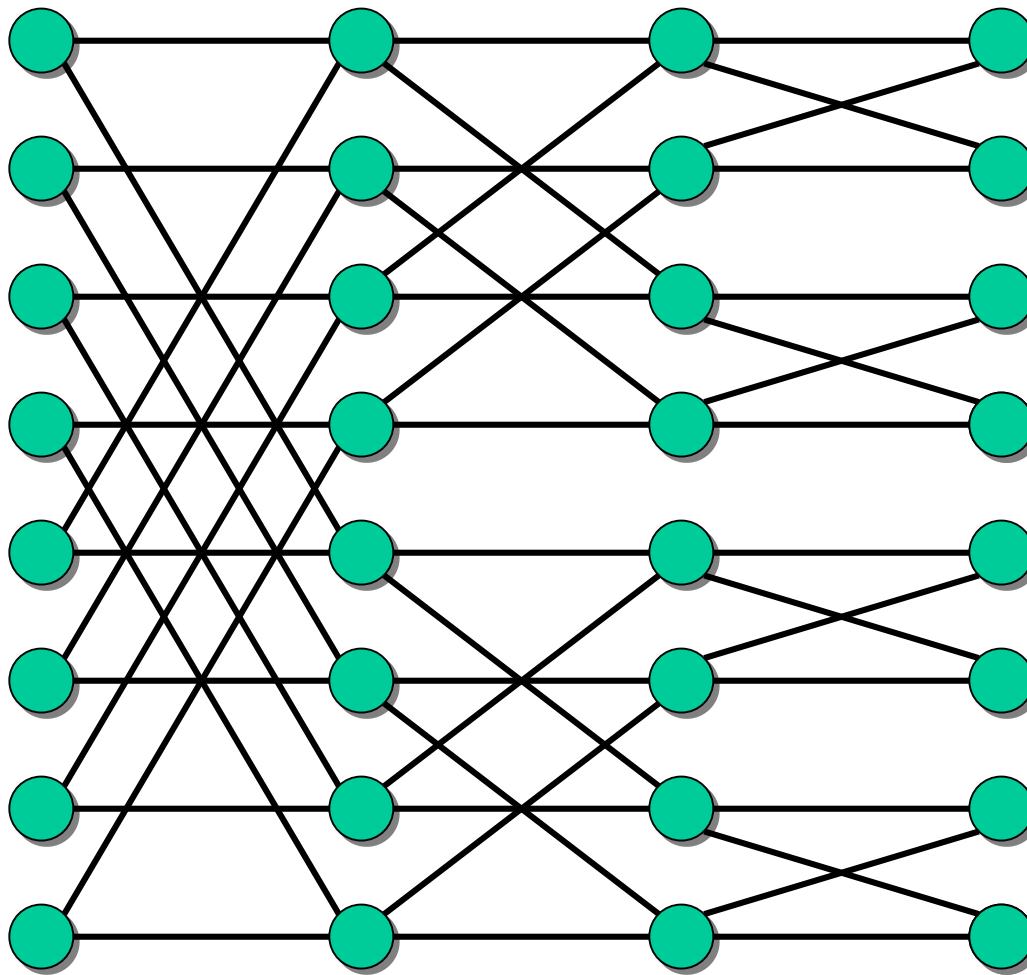
Decomposing a Butterfly



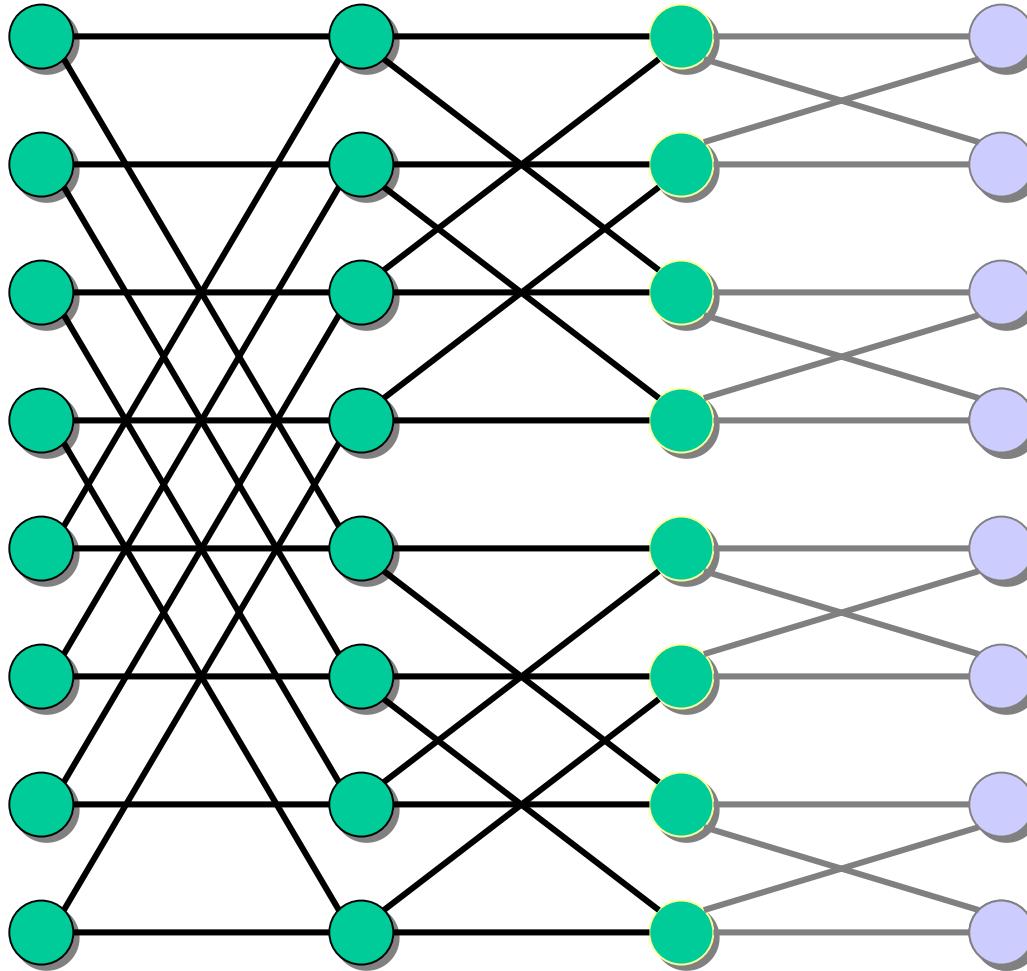
Decomposing a Butterfly



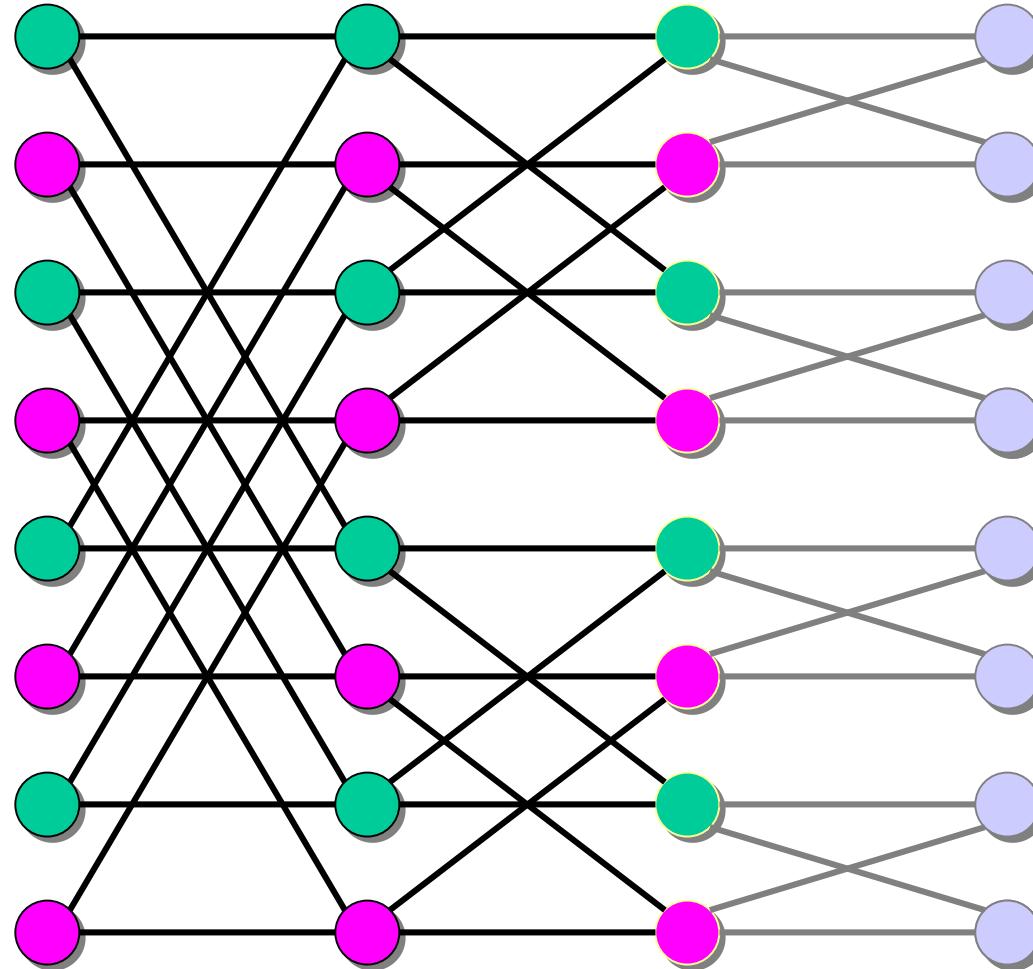
Decomposing a Butterfly II



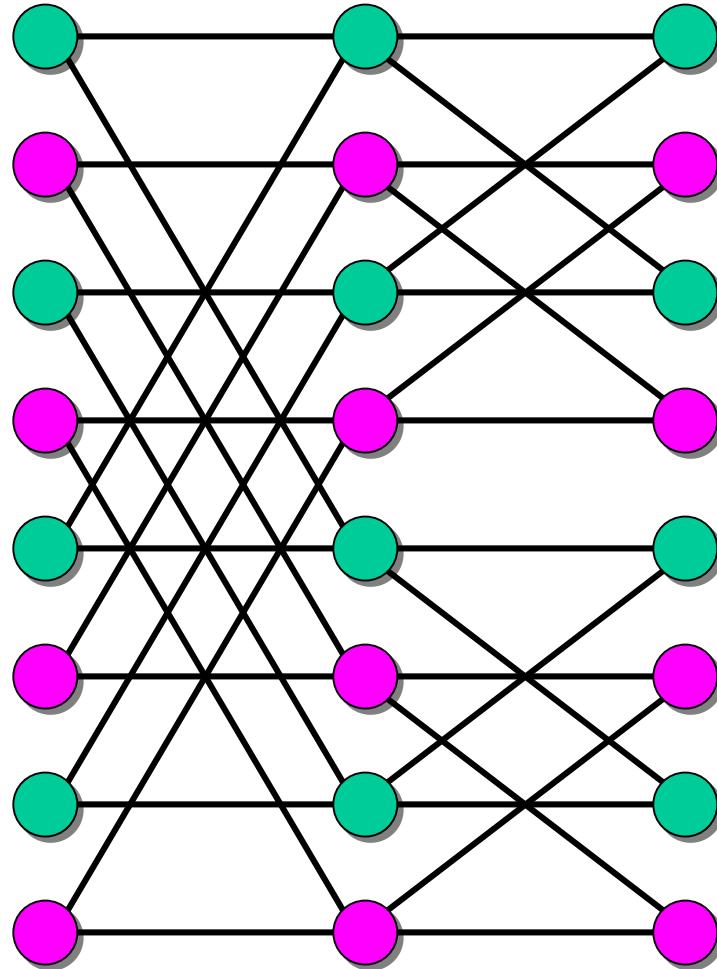
Decomposing a Butterfly II



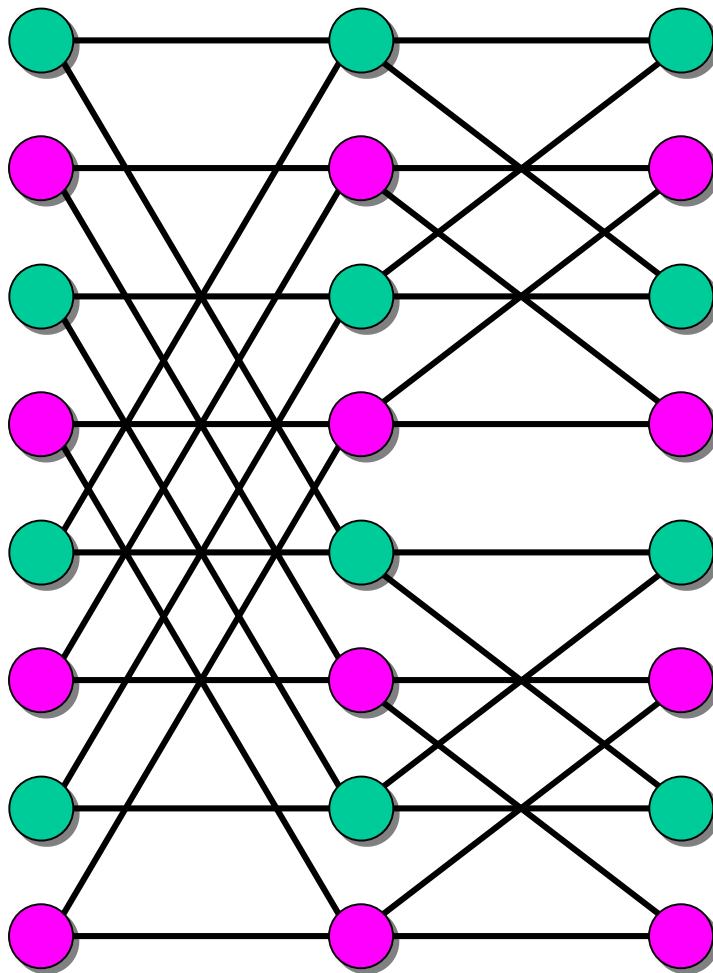
Decomposing a Butterfly II



Decomposing a Butterfly II



Decomposing a Butterfly II



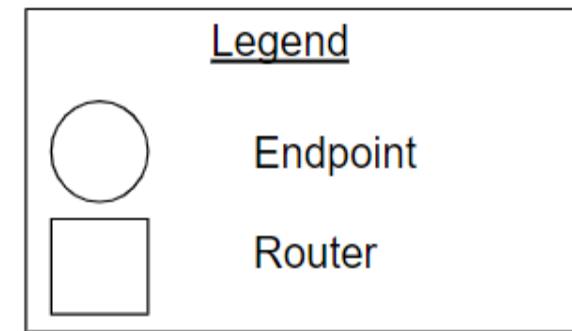
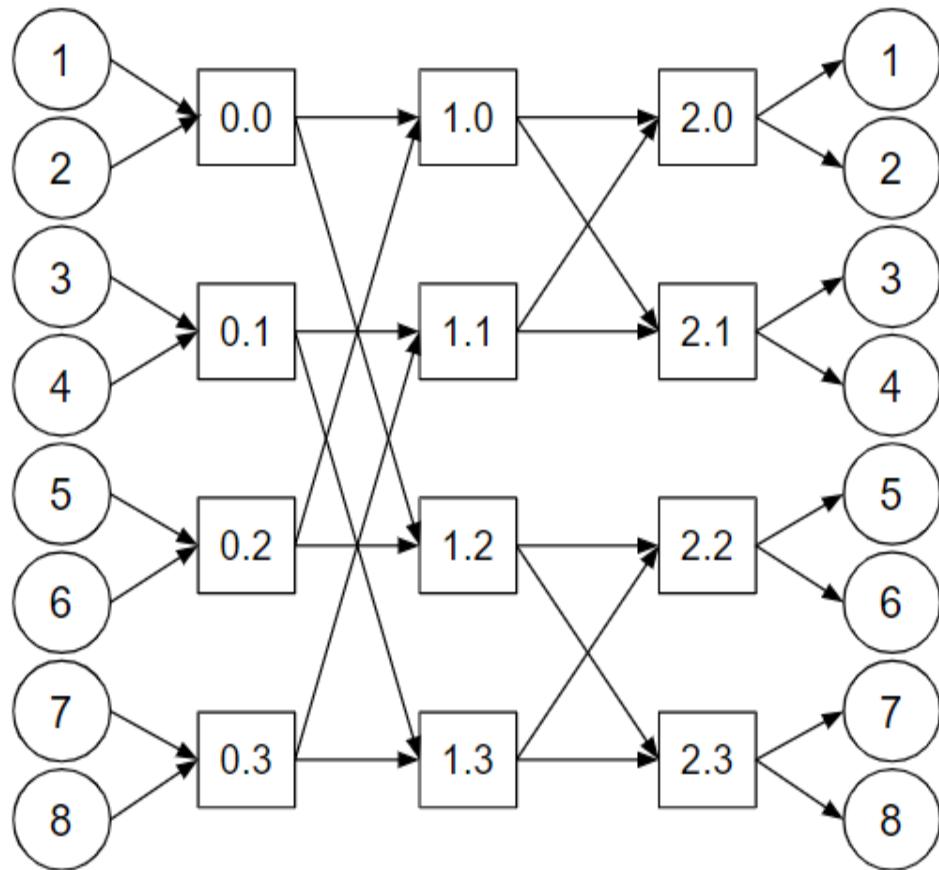
Computer Architecture / Parallel Computing view (MIN)

- In real hardware, those “nodes” are not processors — they are **switching elements (SEs)**.
- The processors sit **only at the first and last rows** of the butterfly.
- The middle rows (intermediate ranks) are **pure switches** that just forward data.

This is why in architecture books, the **Butterfly network is classified as an MIN (indirect)**.

The processors never directly connect to each other; their messages travel through a *stage-by-stage set of switches*.

A 2-ary 3-fly butterfly



Ref: https://google.github.io/xls/noc/xls_noc_butterfly_topology/

Hypercube (Cube Connected) Networks: (6 of 8)

1. 2^k nodes form a k-D network
2. Node addresses $0, 1, \dots, 2^k-1$
3. Diameter with 2^k nodes is k
4. Bisection width is 2^{k-1}
5. Low diameter and high bisection width
6. Node i connected to k nodes whose addresses differ from i in exactly one bit position
7. No. of edges per node is k-the logarithmic of the no. of nodes in the network (**Ex. CM-200**)

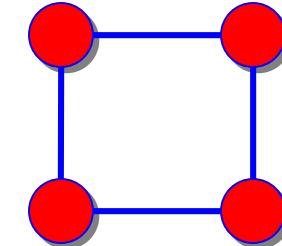
Hypercube



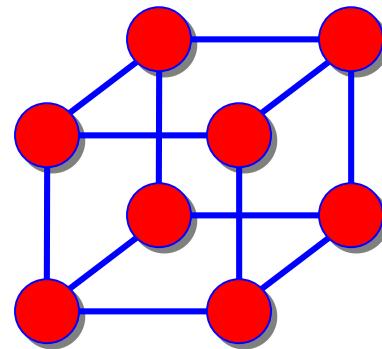
$k = 0$
 $N = 1 (2^k)$



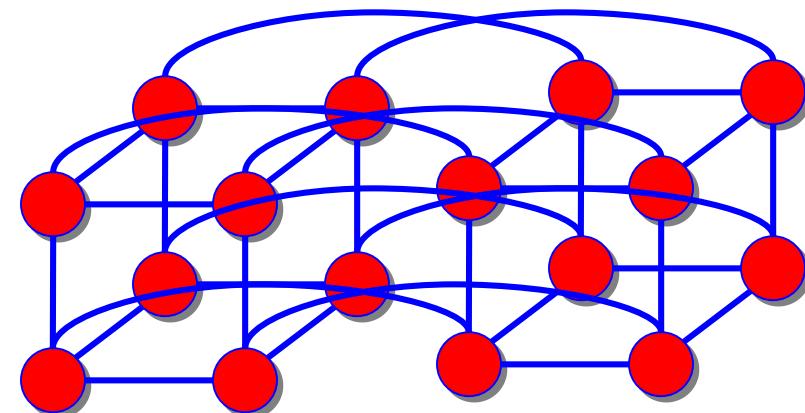
$k = 1$
 $N = 2$



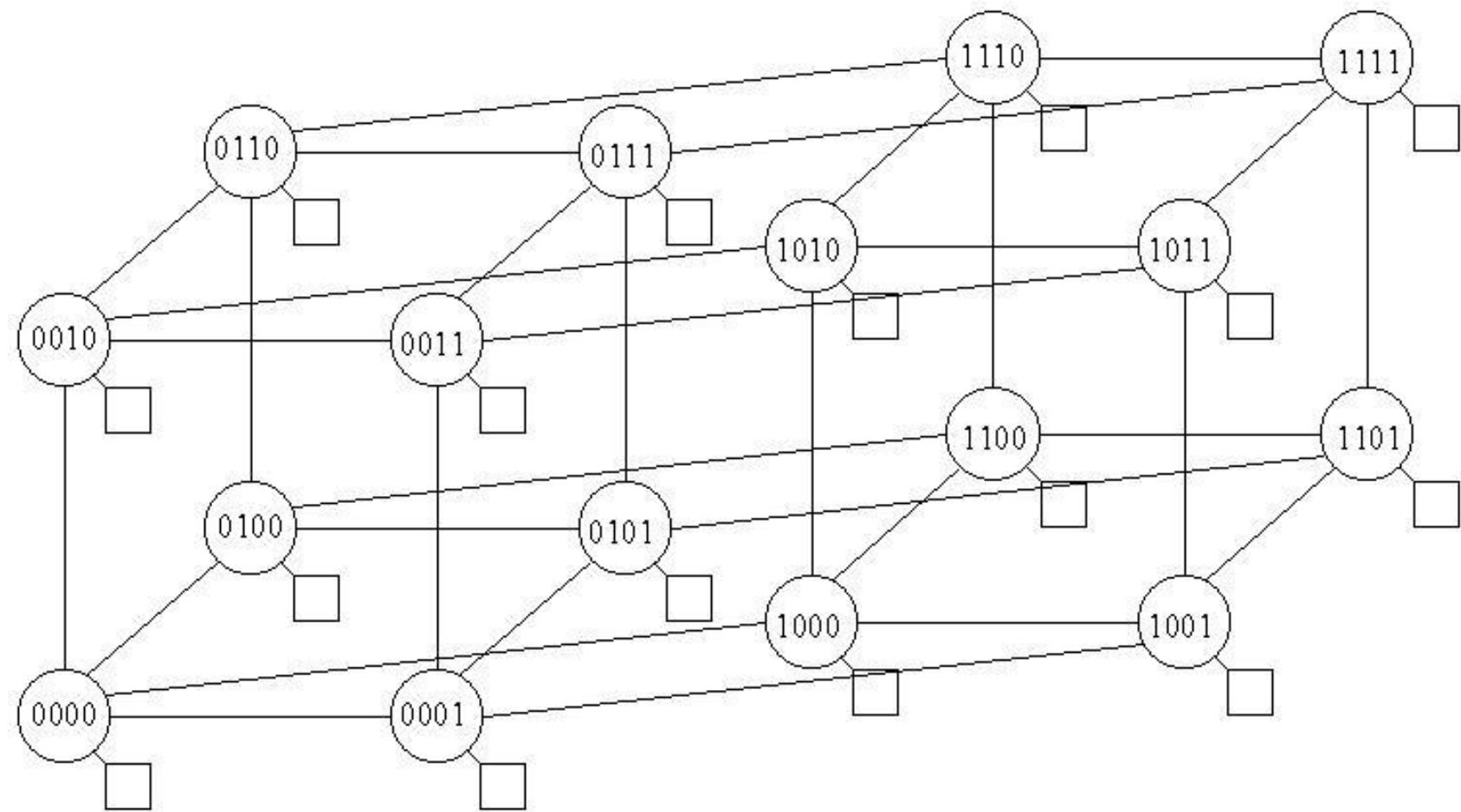
$k = 2$
 $N = 4$



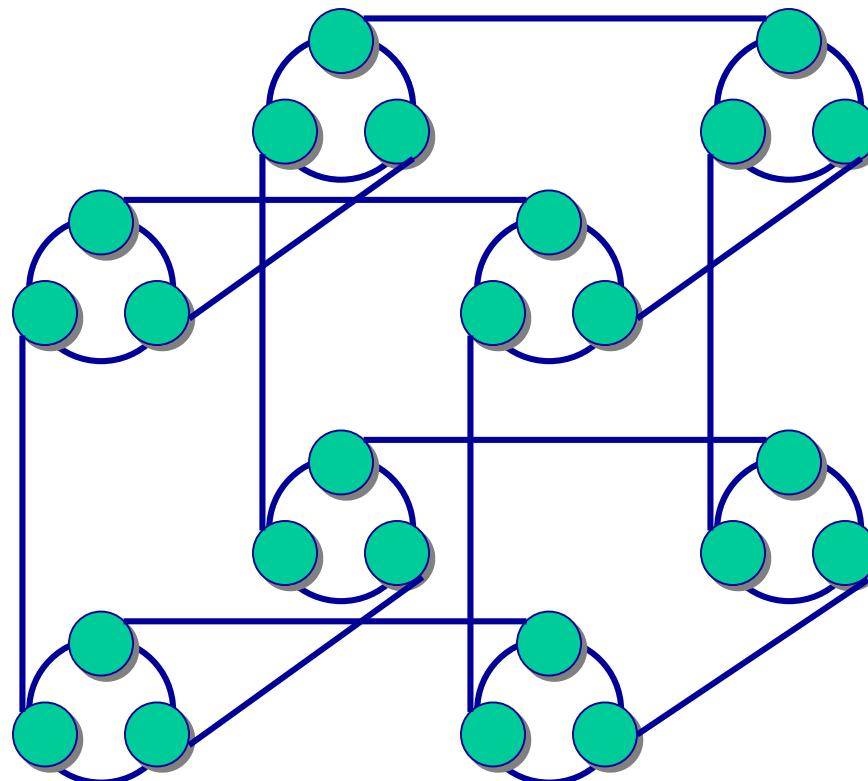
$k = 3$
 $N = 8$



$k = 4$
 $N = 16$



Cube-Connected Cycles



Shuffle Exchange Network: (7 of 8)

1. Consist of $n = 2^k$ nodes numbered $0, \dots, n-1$ having two kind of connections called **shuffle** and **exchange**.
2. Exchange connections link pairs of nodes whose numbers differ in their last significant bit.
3. Shuffle connection link node i with node $2i \bmod (n-1)$, with the exception that node $n-1$ is connected to itself.

4. Let $a_{k-1}a_{k-2}\dots a_0$ be the address of a node in a perfect shuffle network, expressed in binary. A datum at this address will be at address $a_{k-2}\dots a_0a_{k-1}$.



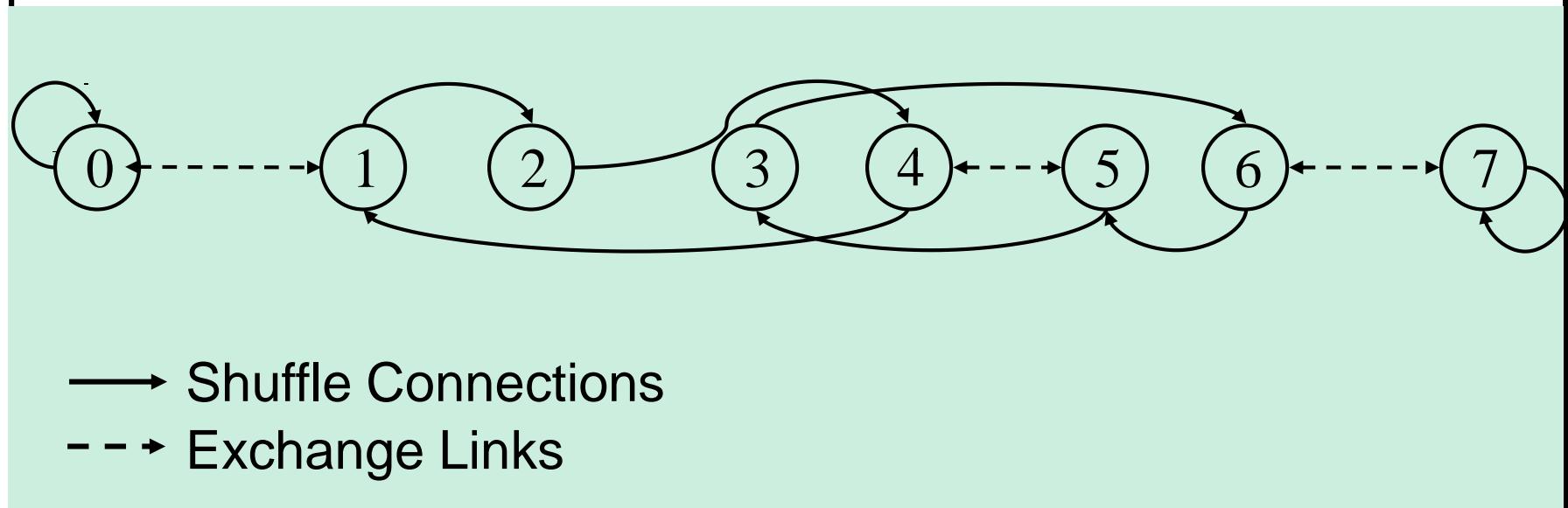
5. **Length of the longest link increases as a function of network size.**



6. Diameter of the network with 2^k nodes is $2k-1$



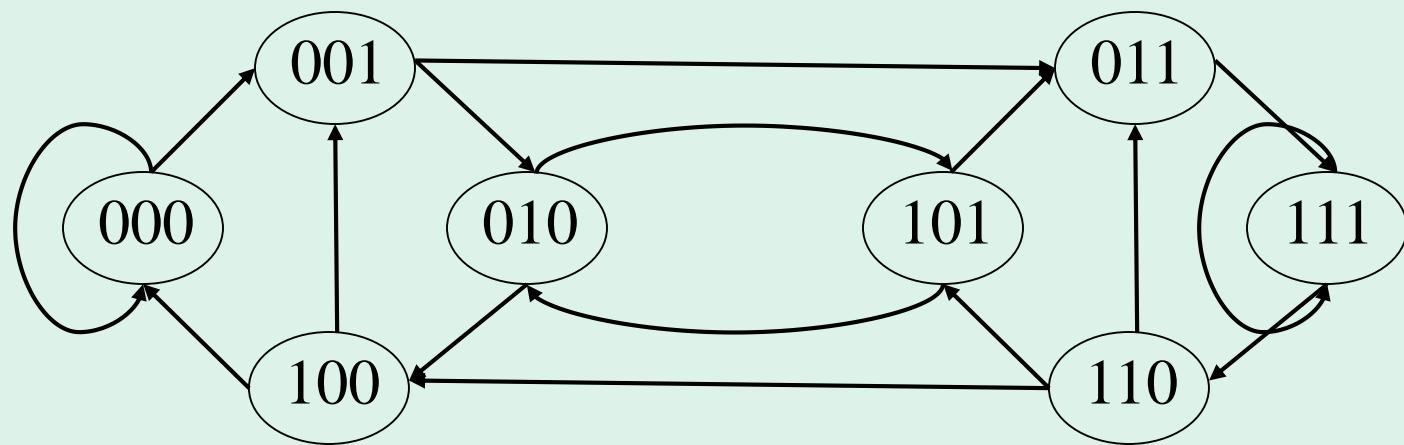
7. Bisection width is $2^{k-1}/k$



de Bruijn network: (8 of 8)

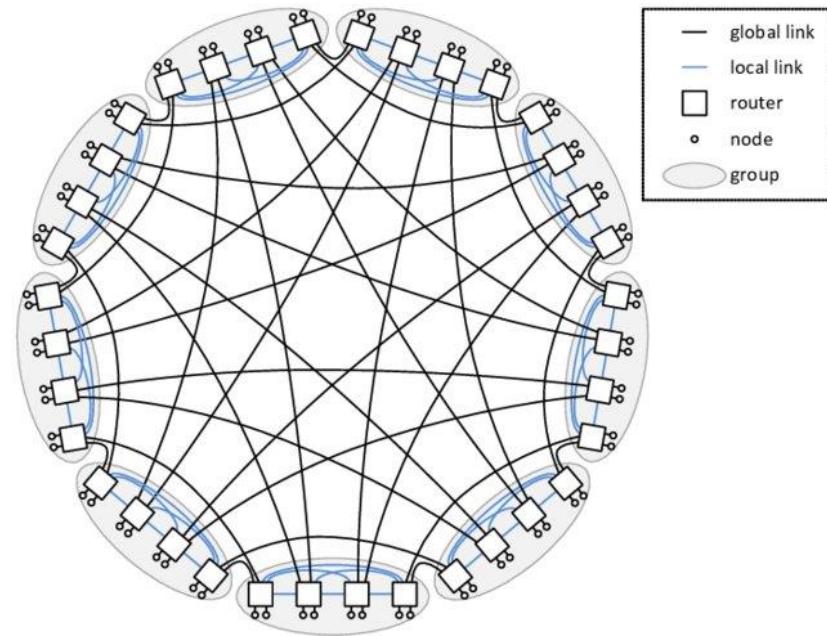
1. Let $n = 2^k$ nodes and $a_{k-1}a_{k-2}\dots a_0$ be the addresses
2. Two nodes reachable via directed edges are
 $a_{k-2}a_{k-3}\dots a_00$ and $a_{k-2}a_{k-3}\dots a_01$
3. The number of edges per node are constant independent of the network size.
4. Bisection width with 2^k nodes is $2^k/k$
5. Diameter is k





Facts about El Capitan's Network

- The interconnect used is HPE Slingshot.
- The topology is a **1-D Dragonfly** topology among the compute groups/racks.
- Switches are high-radix, 64-port, with high bandwidth (each port delivering ~200 Gbps)
- At most **three switch-to-switch hops** between any two endpoints (nodes) in the system



Dragonfly (1-D Dragonfly Topology)

1. Network consists of groups of routers; each router connects to local nodes, other routers in its group, and routers in different groups.
2. Node addresses are of the form (group, router, terminal).
3. Diameter of the network is at most 3 hops (source node → local router → global router → destination node).
4. Bisection bandwidth is high because many global links connect groups evenly.
5. Provides low diameter and high throughput, ideal for exascale HPC systems.

Dragonfly (1-D Dragonfly Topology)

7. Each router of radix k divides its ports into:
 - a. p ports for terminals (compute nodes),
 - b. $a-1$ ports to routers within the same group,
 - c. h ports to routers in other groups.
8. Scales efficiently: with high-radix routers, supports **hundreds of thousands of nodes** with only 2–3 hops between them (e.g., El Capitan supercomputer).

Dragonfly (1-D Dragonfly Topology)

We have the formula

$$k = p + (a - 1) + h$$

where:

- p = terminals per router (compute nodes)
- $(a - 1)$ = local connections to other routers in the same group
- h = global connections to routers in other groups
- k = total router radix (number of ports per router)

Dragonfly (1-D Dragonfly Topology)

Example with $p = 8$

Let's assume:

- Each router has **radix $k = 64$** (common in practice, like Slingshot switches).
- $p = 8$ means 8 terminals (compute nodes) connected per router.
- Suppose each group has $a = 16$ routers.
 - Then each router needs $(a - 1) = 15$ intra-group (local) connections. Within a group of **a routers**, each router connects to every other router in that group. These are **short, fast local links** (usually copper).

Dragonfly (1-D Dragonfly Topology)

$$p + (a - 1) = 8 + 15 = 23$$

That leaves:

$$h = k - (p + (a - 1)) = 64 - 23 = 41$$

Interpretation

- 8 ports go to compute nodes.
- 15 ports connect to other routers in the same group.
- 41 ports connect to routers in other groups (global optical links).
- So with $p = 8$, you still have plenty of global links ($h = 41$) ensuring high inter-group bandwidth.

Fundamental Difference

- Tree, Mesh, Hypertree → Direct Networks
 - Each processor directly connects to other processors via fixed links.
 - Communication = path along physical layout.
 - Simple to design, scalable in wiring.
 - Often large diameter (longer paths).
- Butterfly, de Bruijn, MINs → Indirect Networks
 - Communication goes through switch nodes (not processors).
 - Messages routed through multiple stages.
 - Very low diameter ($\log p$), highly efficient routing.
 - But more complex, can suffer from congestion and need more switches.

Multiprocessors

- Multiple-CPU computers consist of a number of fully programmable processors, each capable of executing its own program
- Multiprocessors are multiple CPU computers with a shared memory.

- Based on the amount of time a processor takes to access local or global memory, shared address-space computers are classified into two categories.
- If the time taken by a processor to access any memory word is identical, the computer is classified as ***Uniform Memory Access (UMA)*** computer

- If the time taken to access a remote memory bank is longer than the time to access a local one, the computer is called a ***NonUniform Memory Access (NUMA)*** computer.

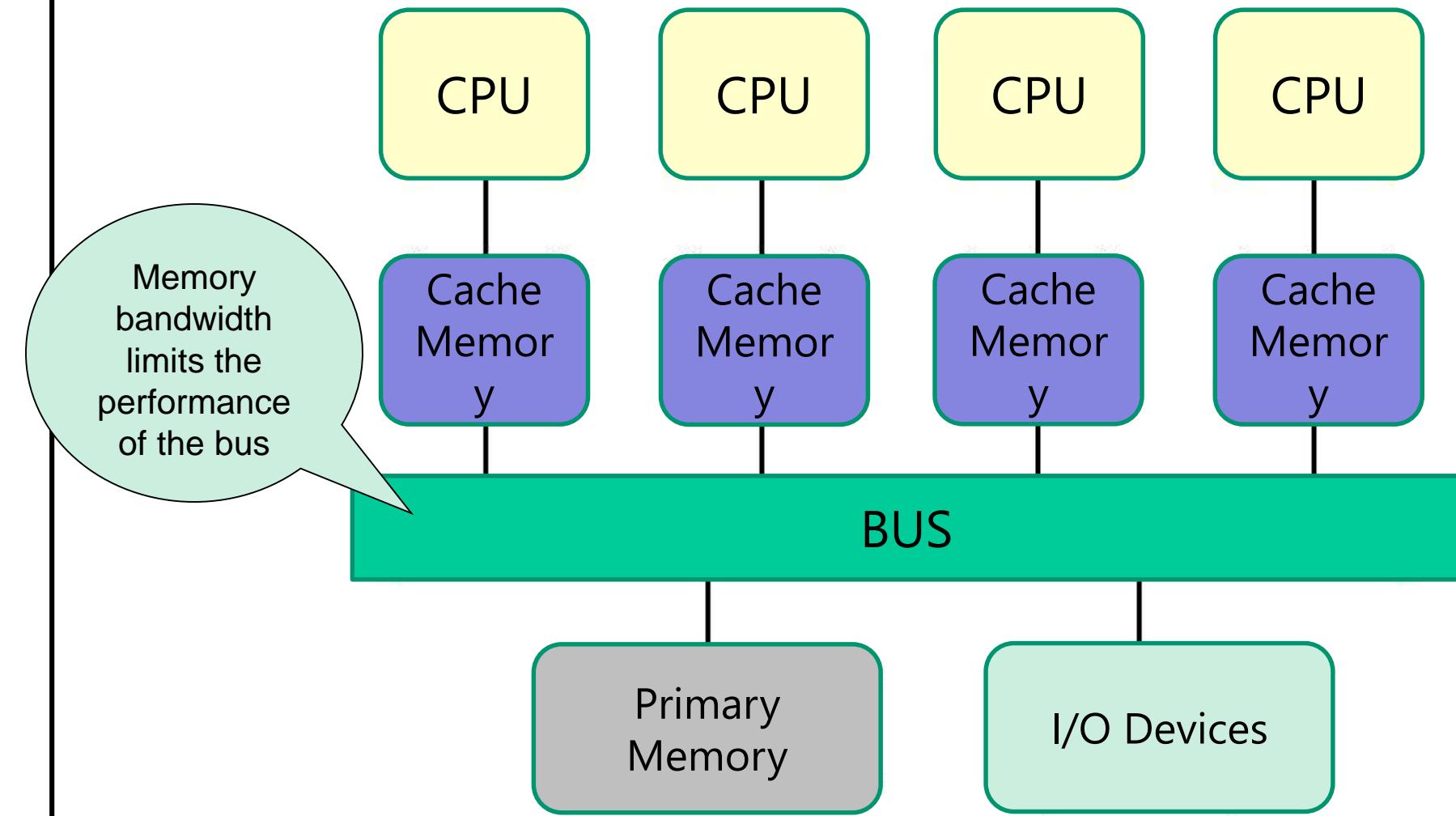
UMA

- Central switching mechanism to reach shared centralized memory
- Switching mechanisms are Common bus, crossbar switch and packet switch net

Centralized Multiprocessor

- Straightforward extension of uniprocessor
- Add CPUs to bus
- All processors share same primary memory
- Memory access time same for all CPUs
 - Uniform memory access (UMA) multiprocessor
 - Symmetrical multiprocessor (SMP)

Centralized Multiprocessor



SMP's in CMSD IBM Power Series Ex. P595....



Private and Shared Data

- **Private data:** items used only by a single processor
- **Shared data:** values used by multiple processors
- *In a multiprocessor, processors communicate via shared data values*

Problems Associated with Shared Data

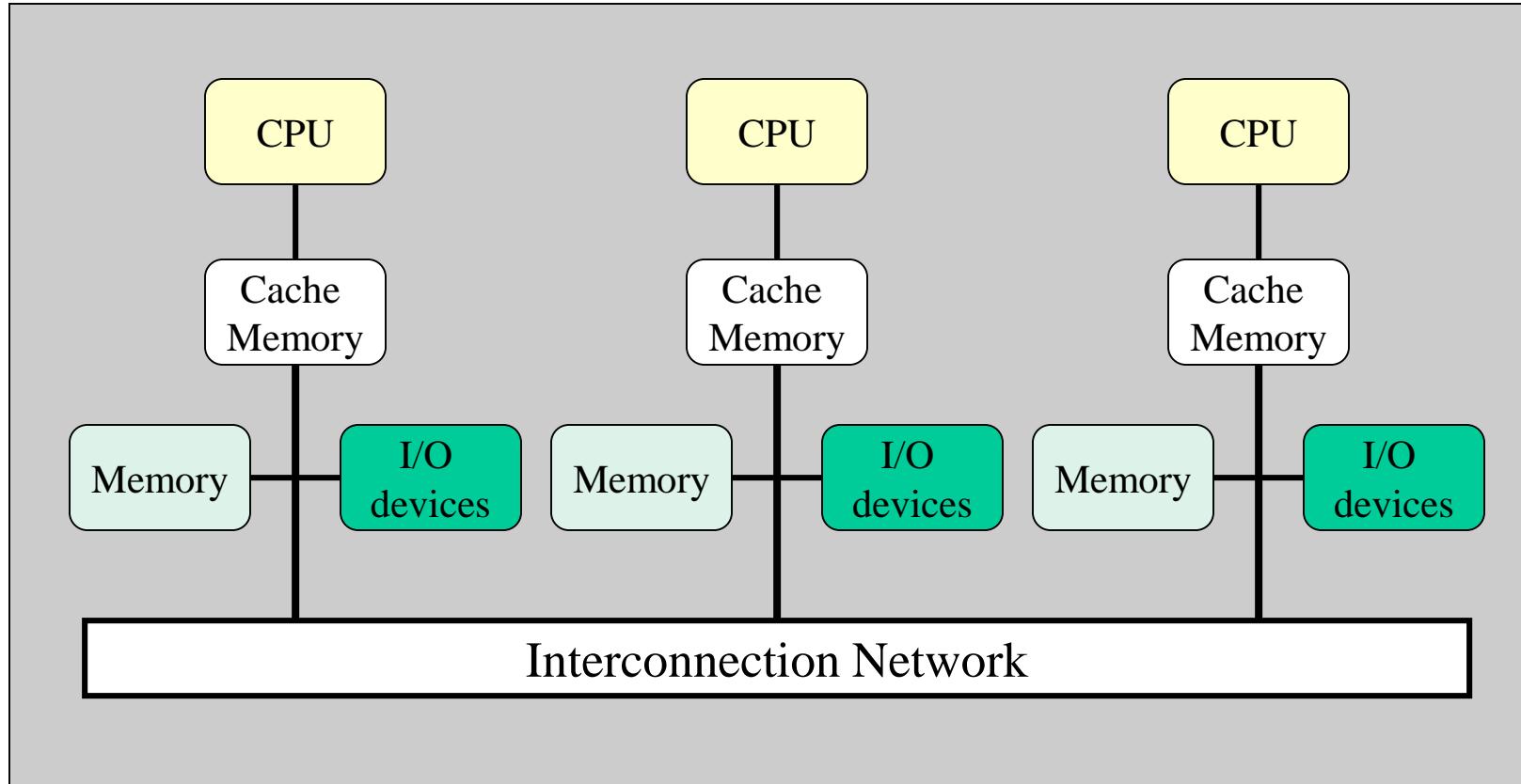
- Cache coherence
 - Replicating data across multiple caches reduces contention
 - How to ensure different processors have same value for same address?
 - Snooping/Snarfing protocol
 - (Each CPU's **cache controller monitor** snoops bus/address-line)
 - Write invalidate protocol (processor sends an invalidation signal over the bus)
 - Write update protocol (processor broadcasts a new data without issuing the invalidation signal)
- Processor Synchronization
 - Mutual exclusion
 - Barrier

- **NUMA Multiprocessors**
- Memory is **distributed**, every processor has some nearby memory, and the shared address space on a NUMA multiprocessor is formed by combining these memories

Distributed Multiprocessor

- Distribute primary memory among processors
- Possibility to distribute instruction and data among memory unit so the memory reference is local to the processor
- Increase aggregate memory bandwidth and lower average memory access time
- Allow greater number of processors
- Also called non-uniform memory access (NUMA) multiprocessor

Distributed Multiprocessor



Cache Coherence

- Some NUMA multiprocessors do not have cache coherence support in hardware
 - Only instructions, private data in cache
 - Large memory access time variance
- Implementation more difficult
 - No shared memory bus to “snoop”
 - Snooping methods does not scale well
 - ***Directory-based protocol needed***

Directory-based Protocol

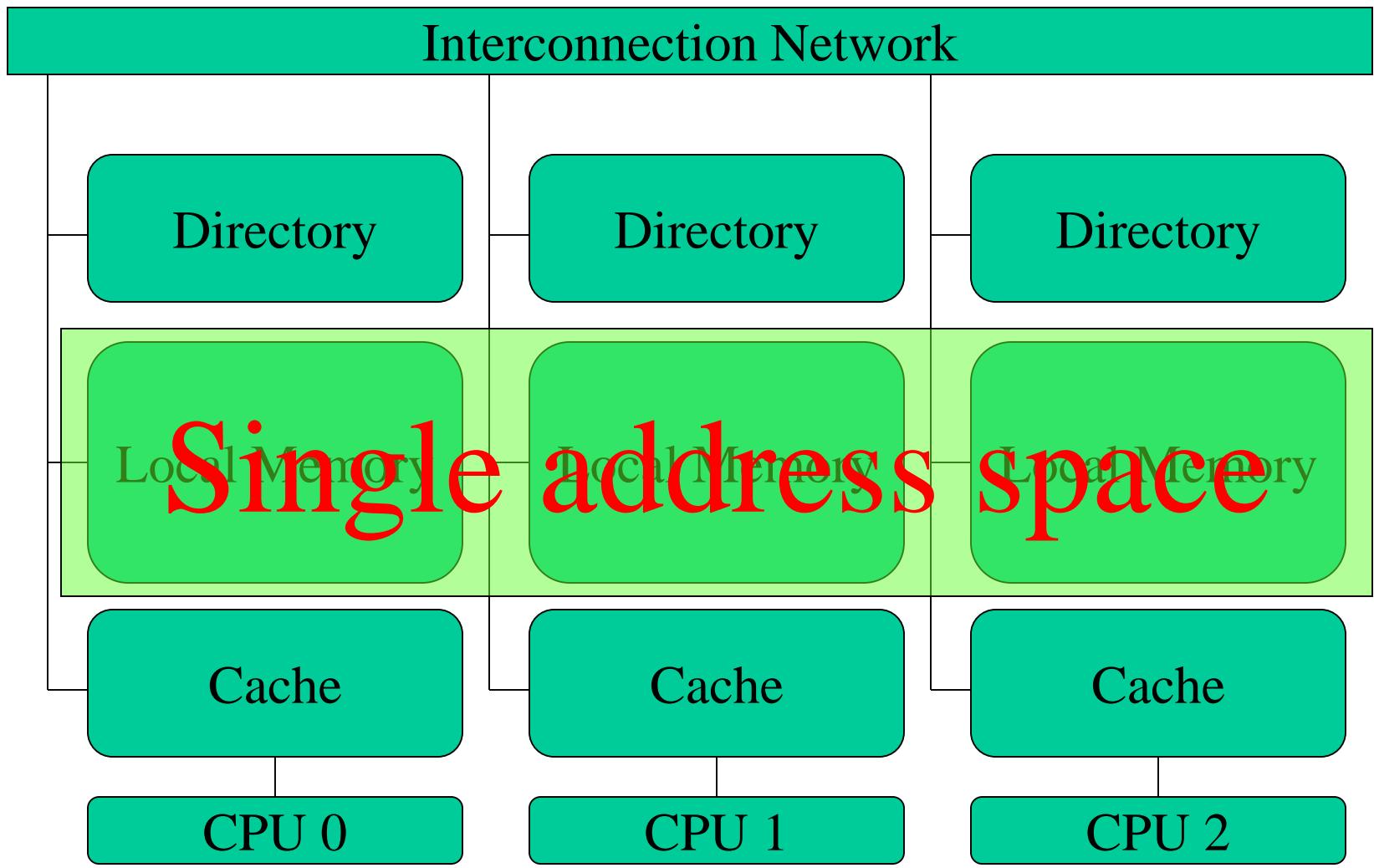
- Distributed directory contains information about cacheable memory blocks
- One directory entry for each cache block
- Each entry has
 - Sharing status
 - Which processors have copies

Sharing Status

- **Uncached**
 - Block not in any processor's cache
- **Shared**
 - Cached by one or more processors
 - Read only
- **Exclusive**
 - Cached by exactly one processor
 - Processor has written block
 - Copy **in memory** is obsolete

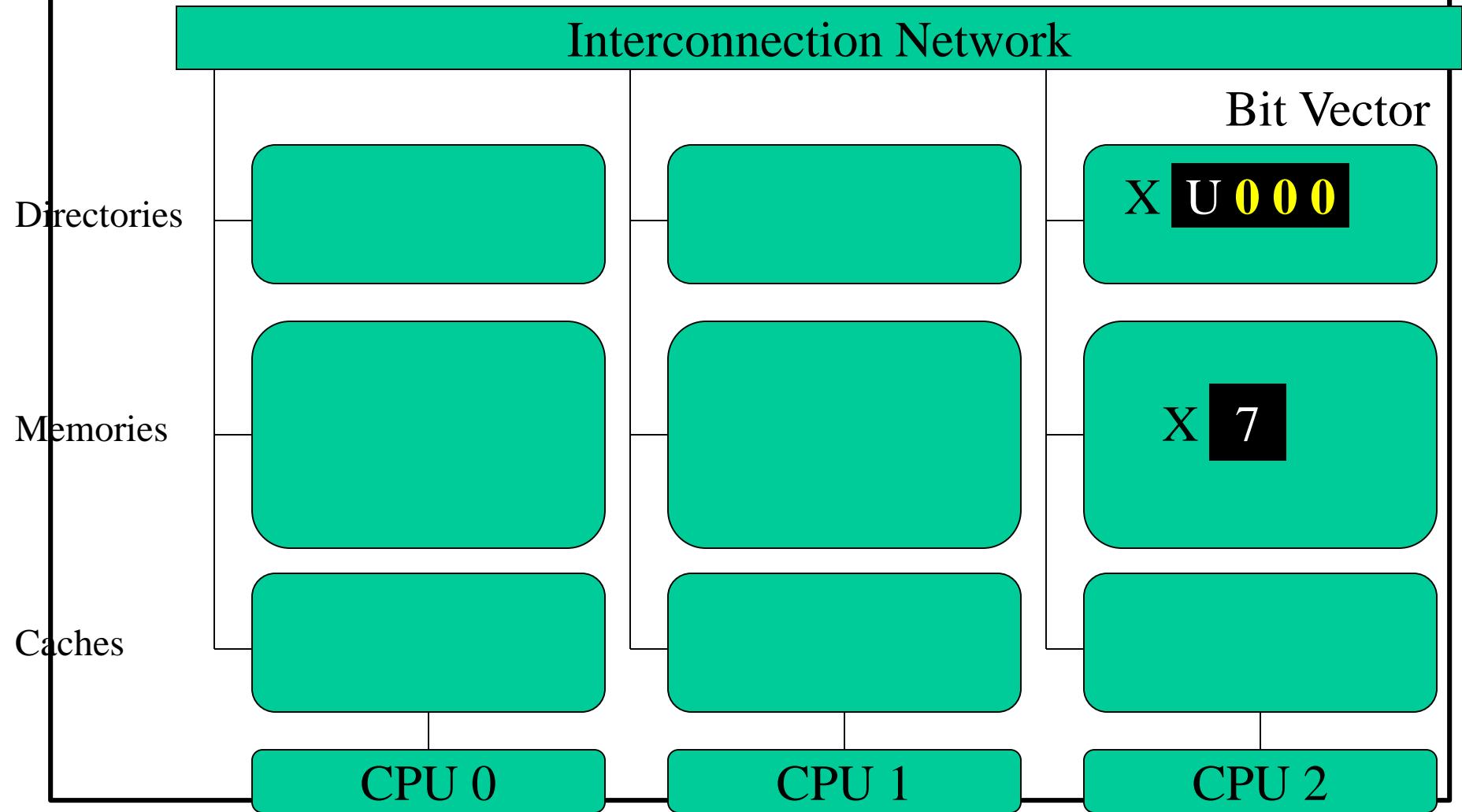
Uncached
Shared
Exclusive

Directory-based Protocol



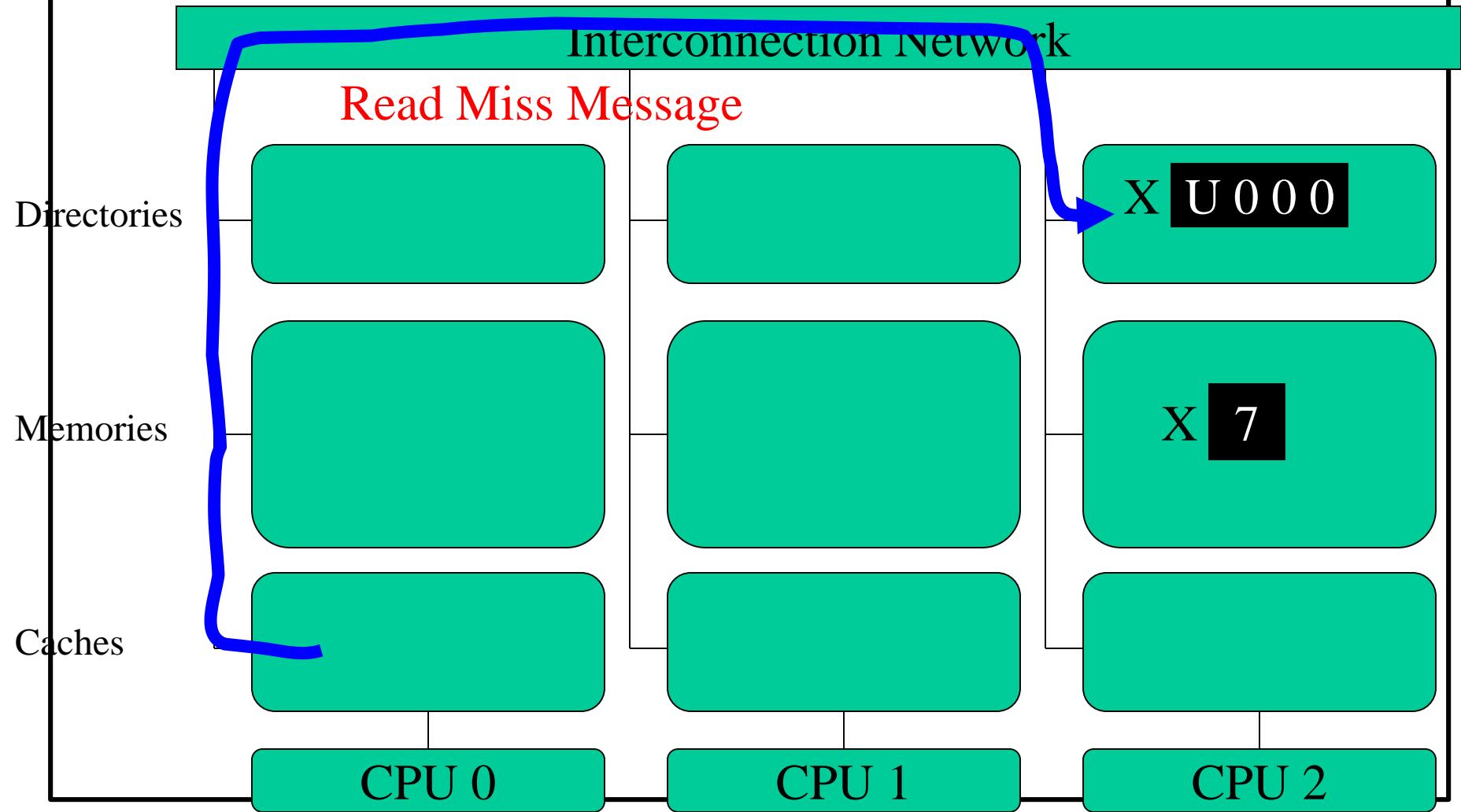
Uncached
Shared
Exclusive

Directory-based Protocol



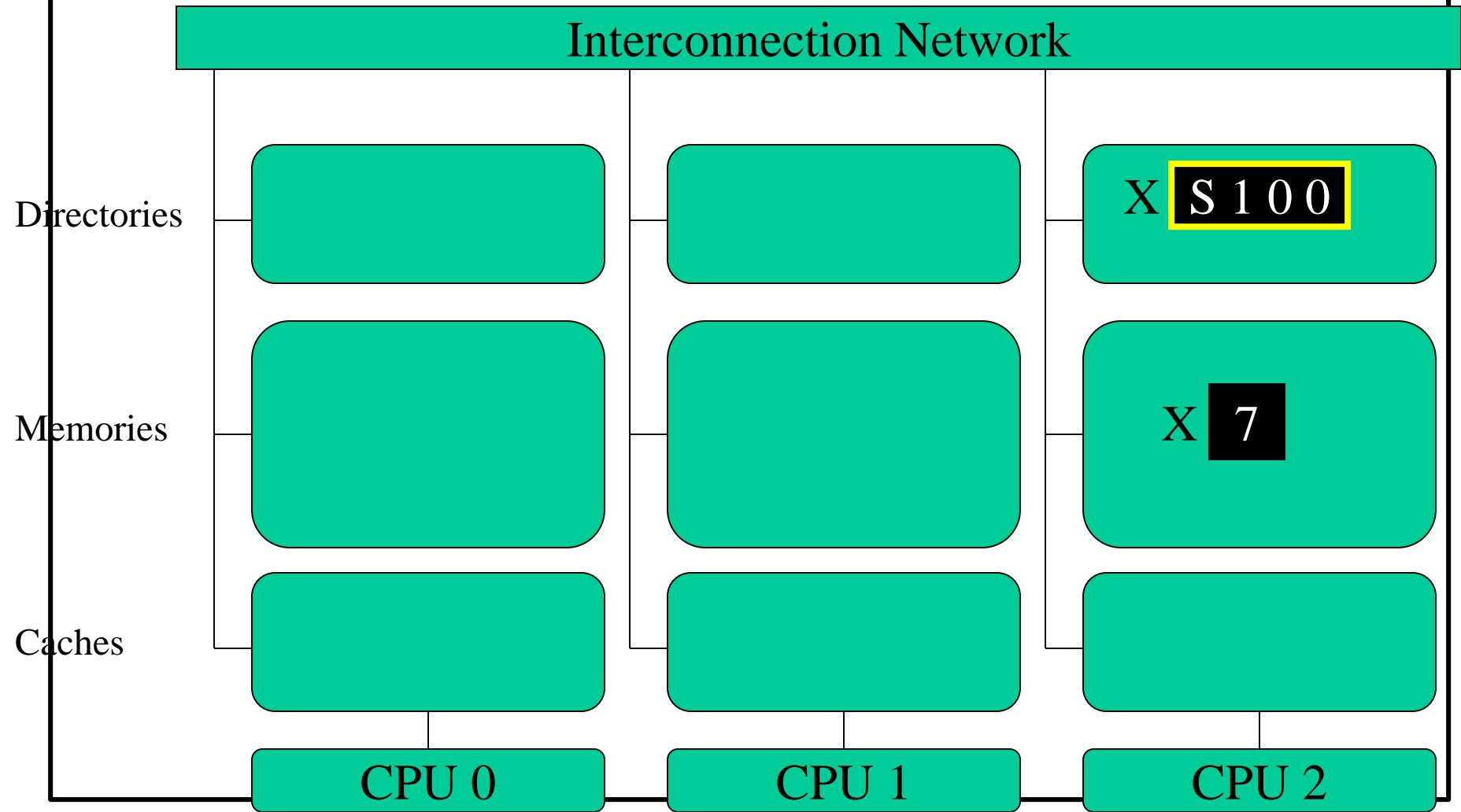
Uncached
Shared
Exclusive

CPU 0 Reads X



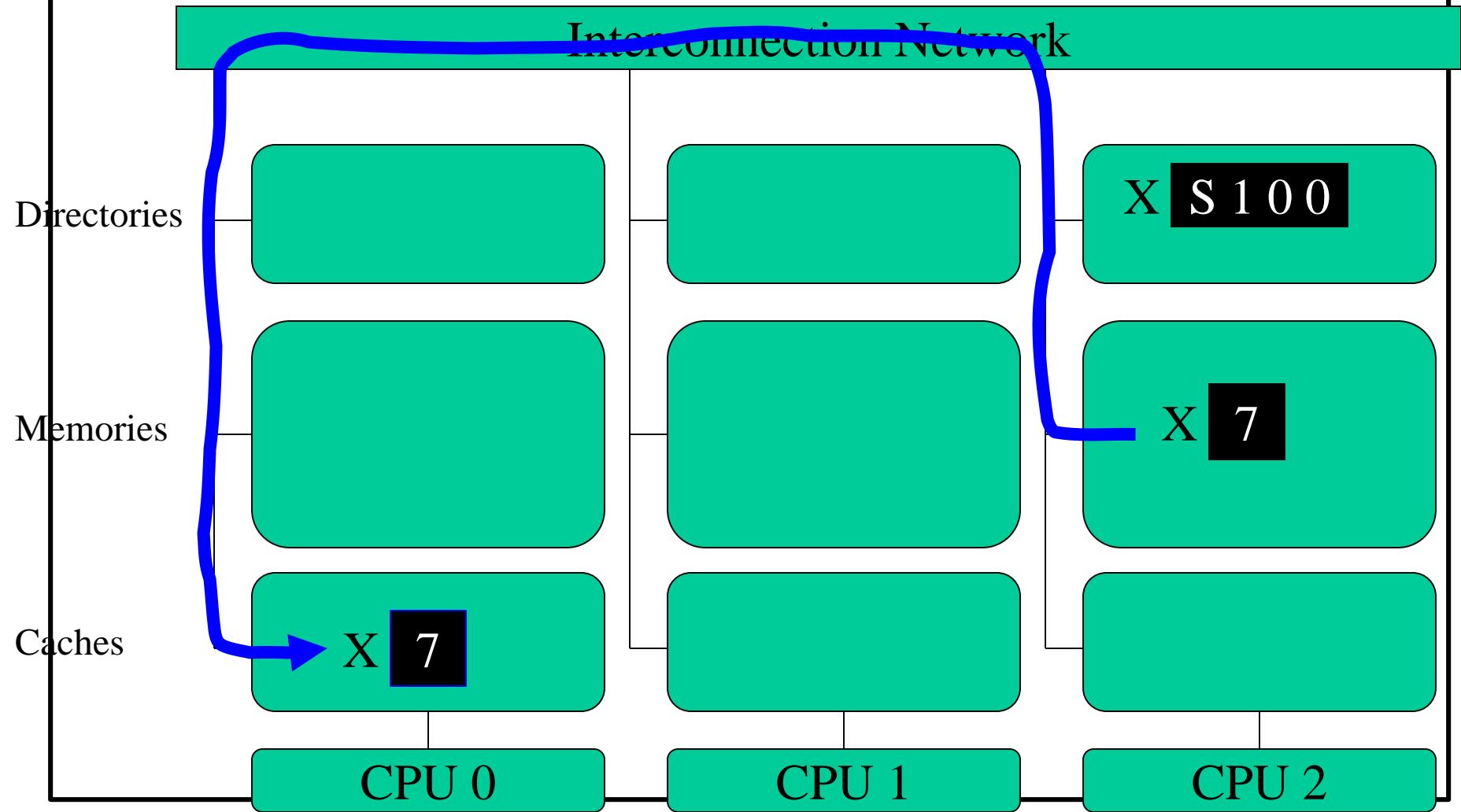
Uncached
Shared
Exclusive

CPU 0 Reads X



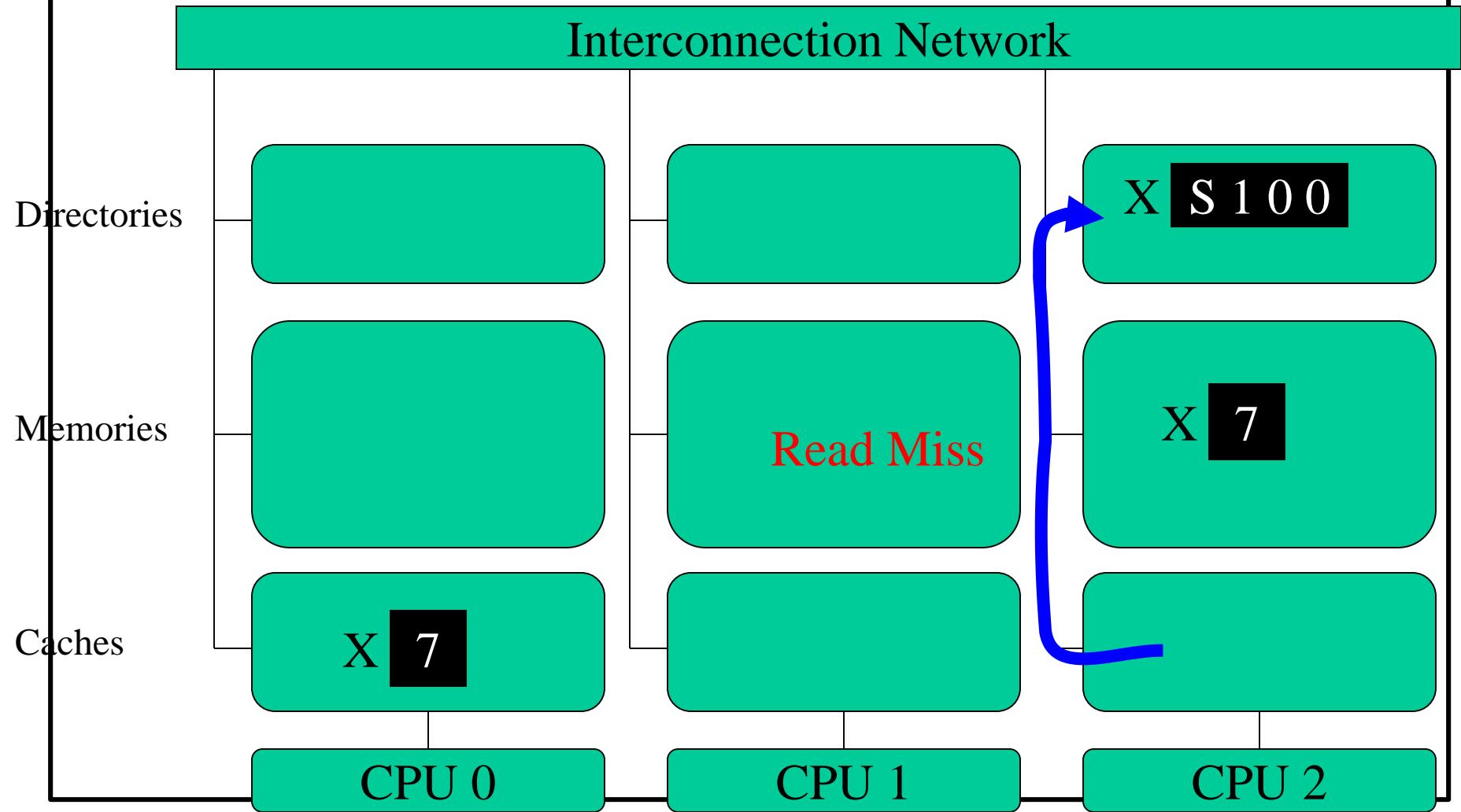
Uncached
Shared
Exclusive

CPU 0 Reads X



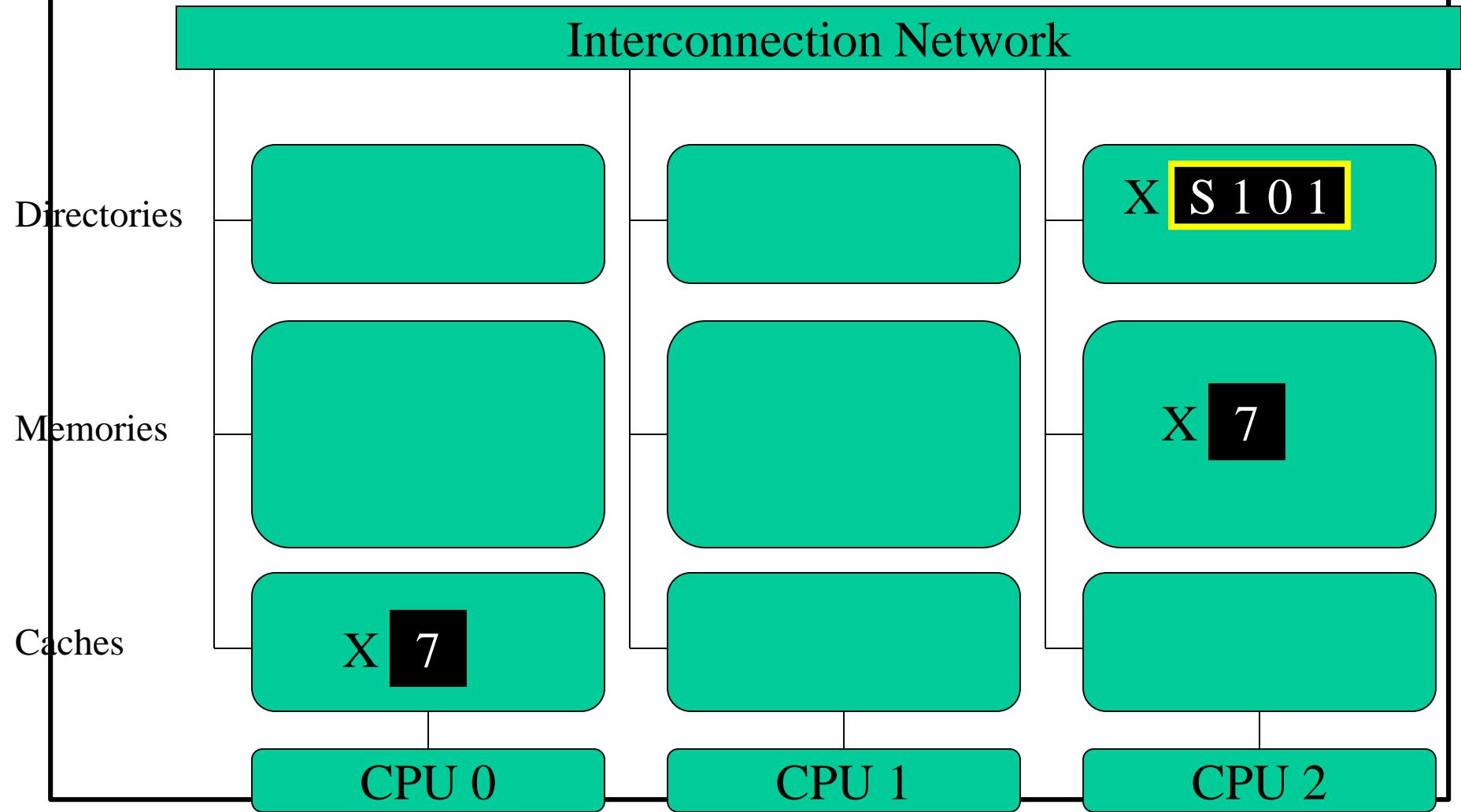
Uncached
Shared
Exclusive

CPU 2 Reads X



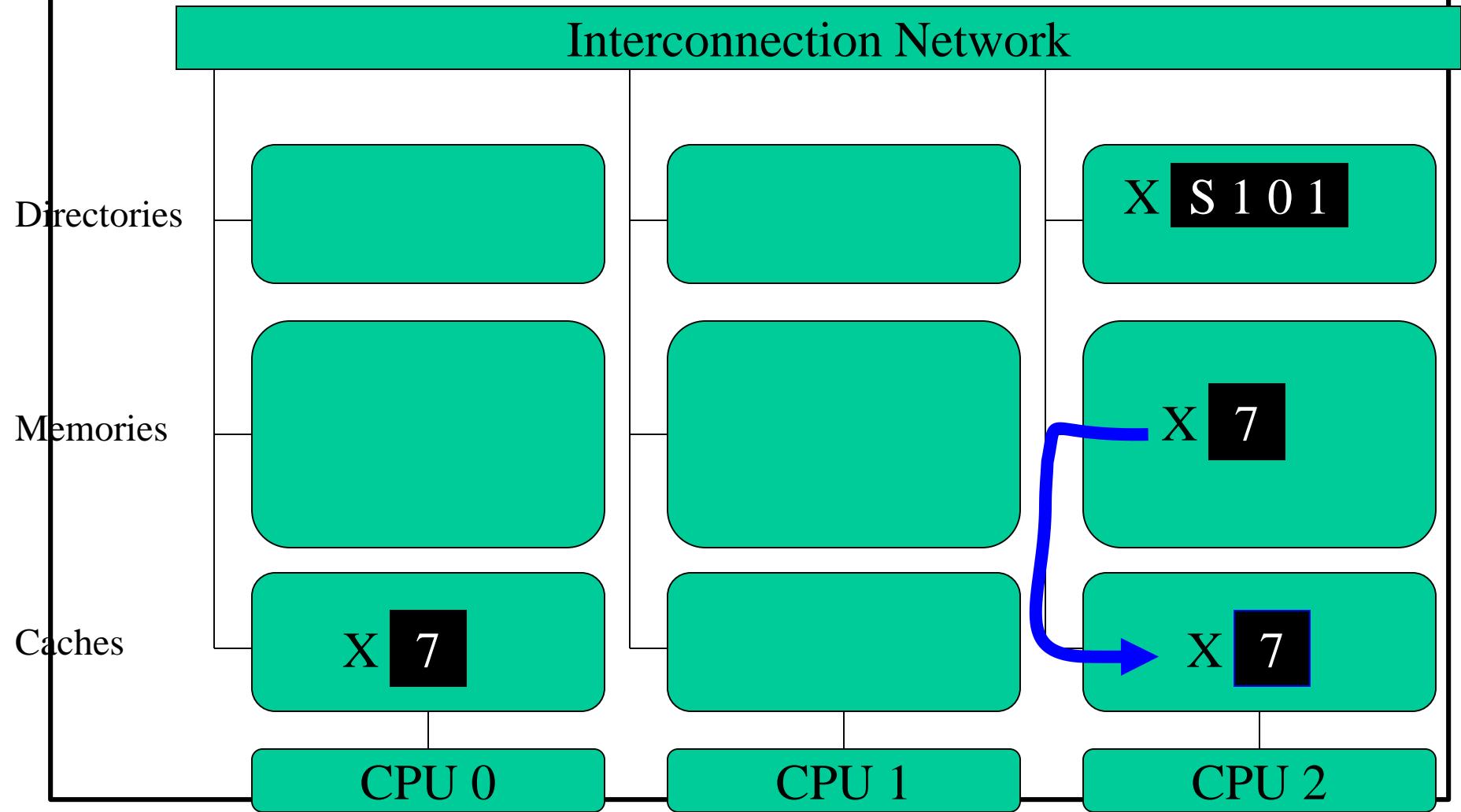
Uncached
Shared
Exclusive

CPU 2 Reads X



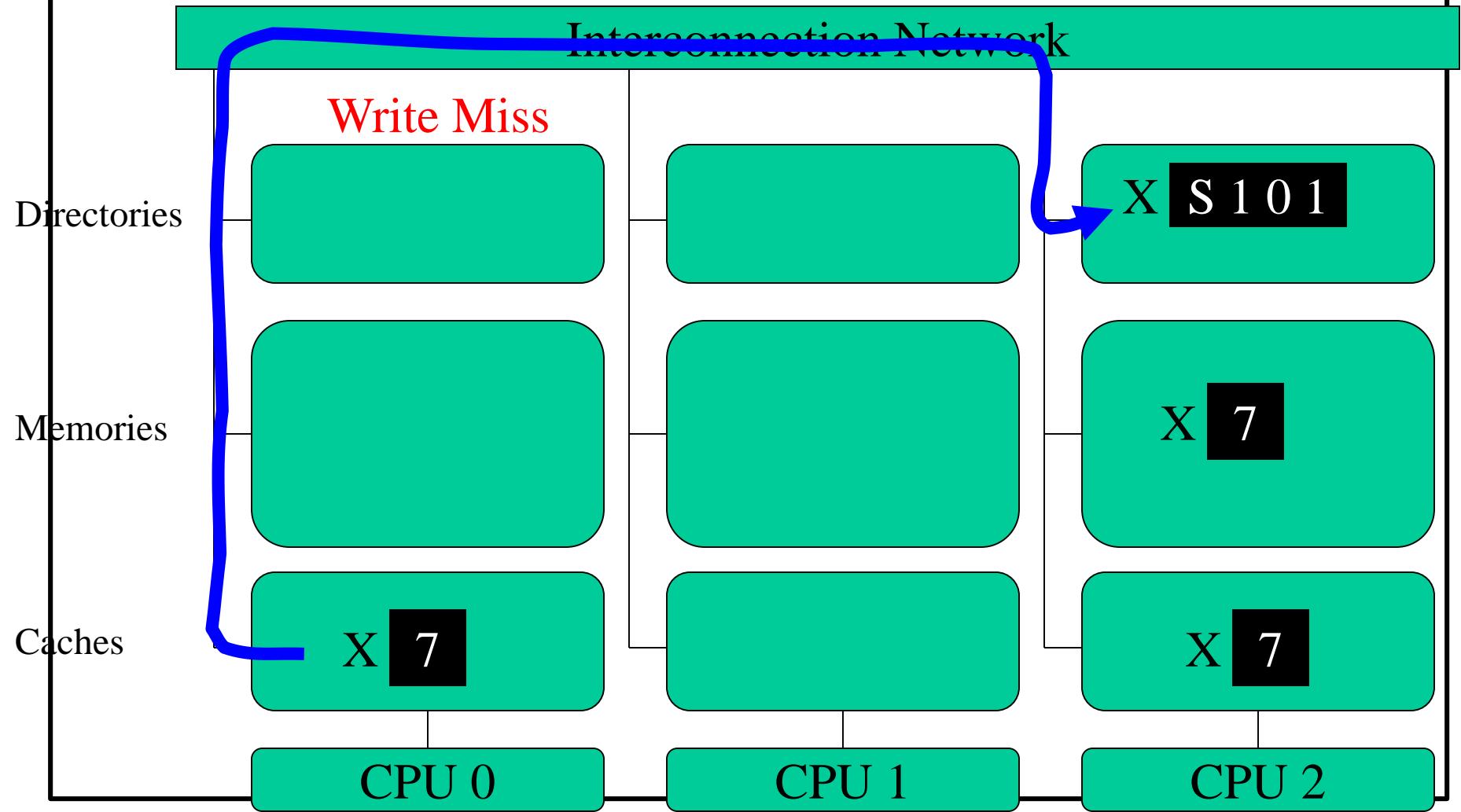
Uncached
Shared
Exclusive

CPU 2 Reads X



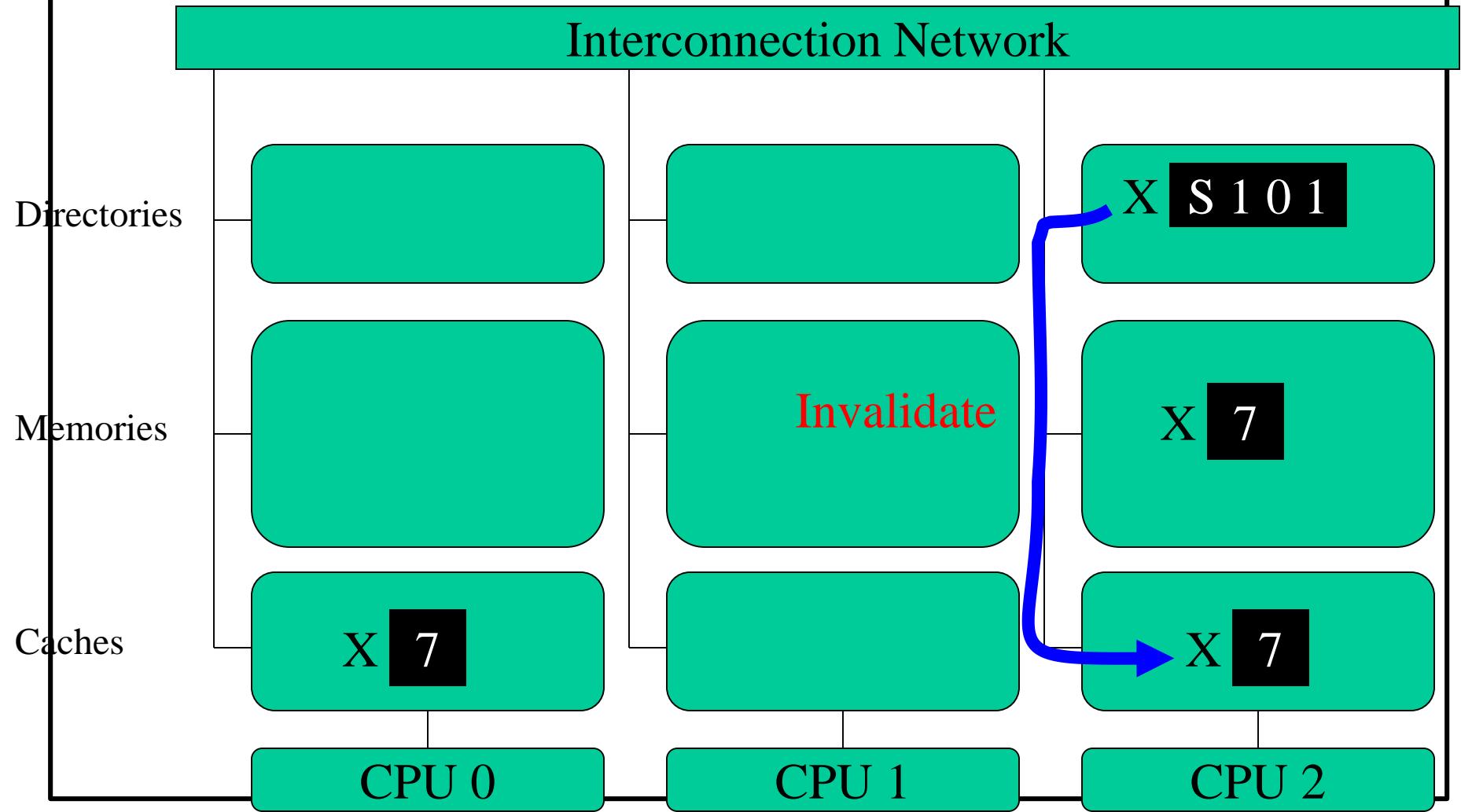
Uncached
Shared
Exclusive

CPU 0 Writes 6 to X



Uncached
Shared
Exclusive

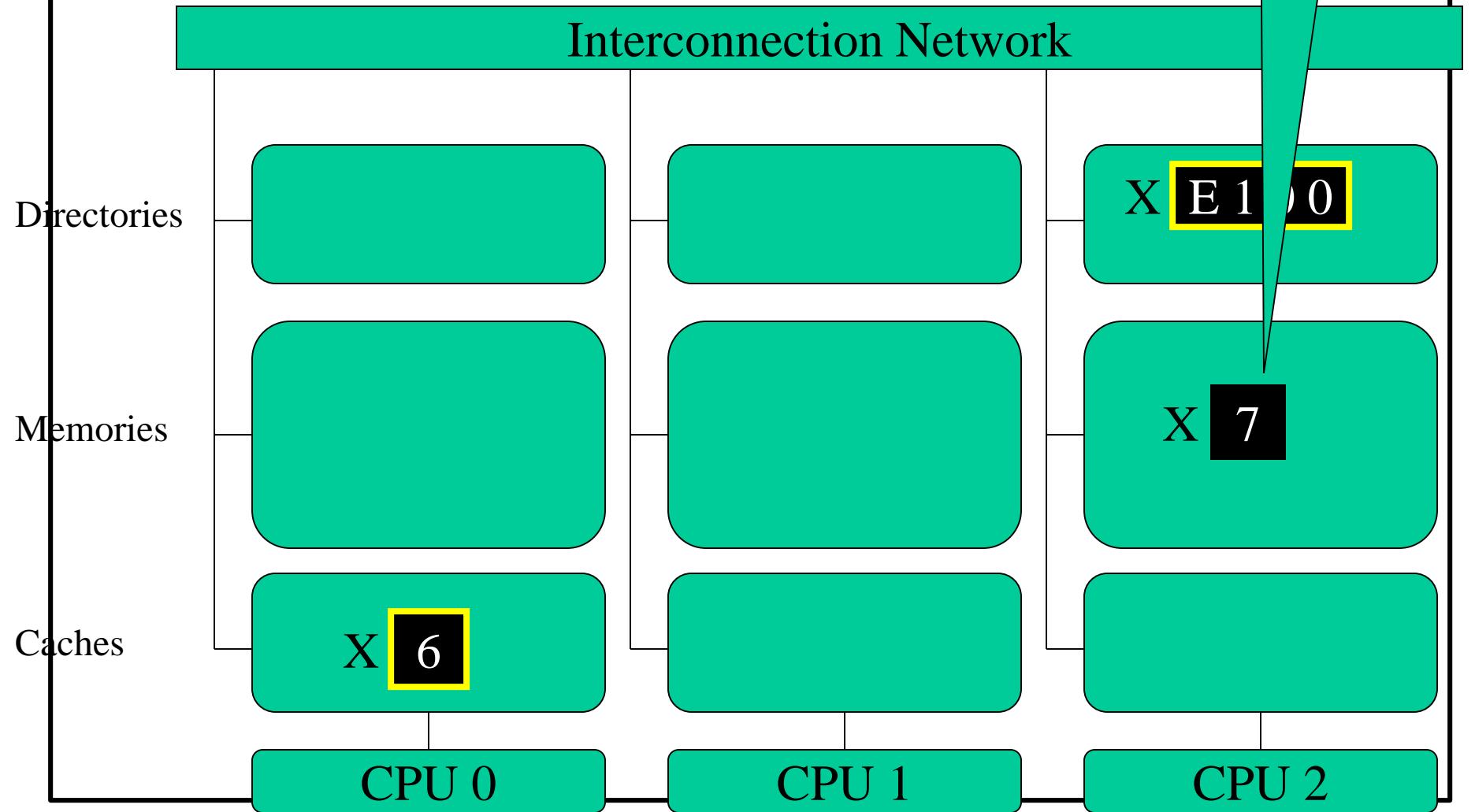
CPU 0 Writes 6 to X



Uncached
Shared
Exclusive

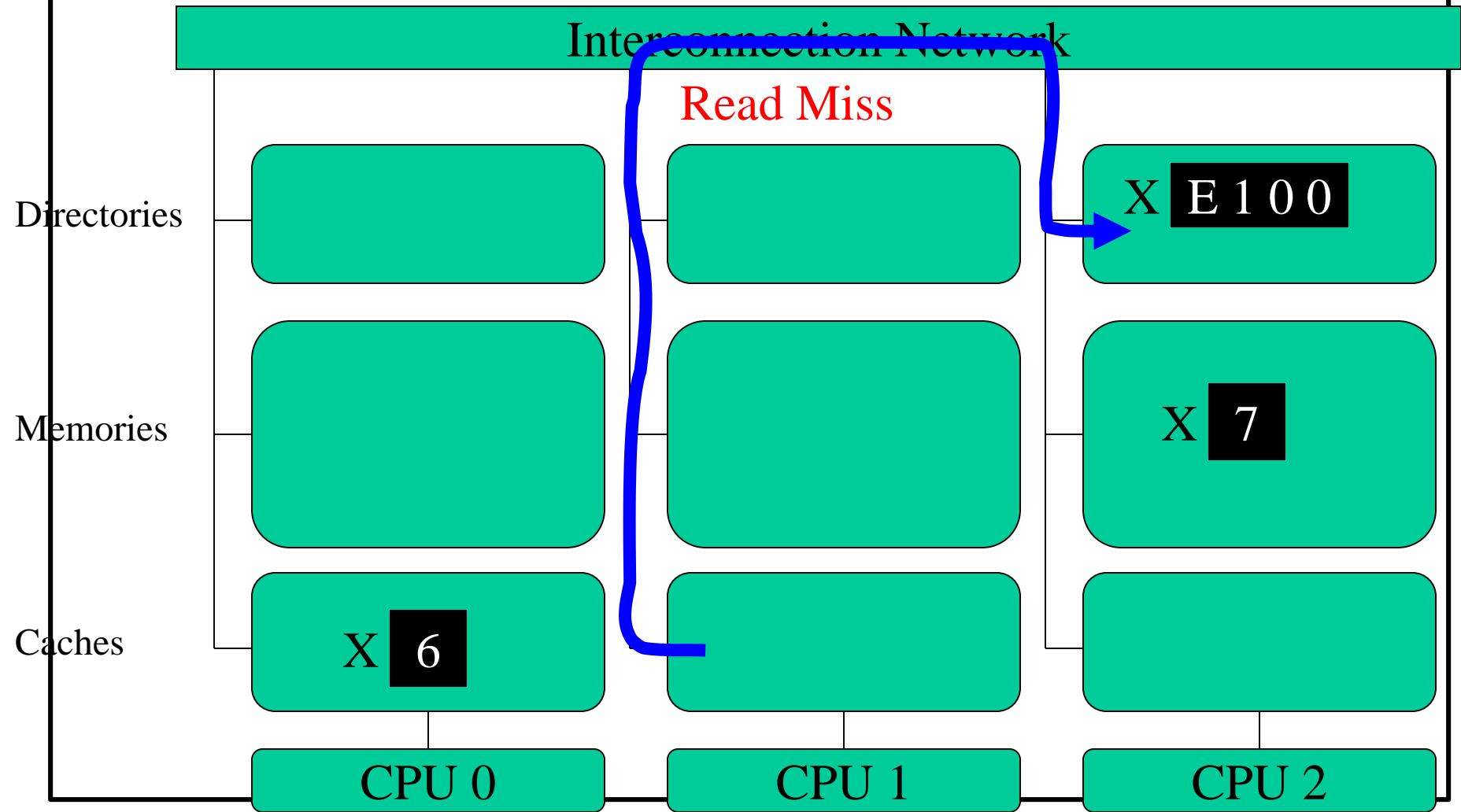
Obsolete

CPU 0 Writes 6 to X



Uncached
Shared
Exclusive

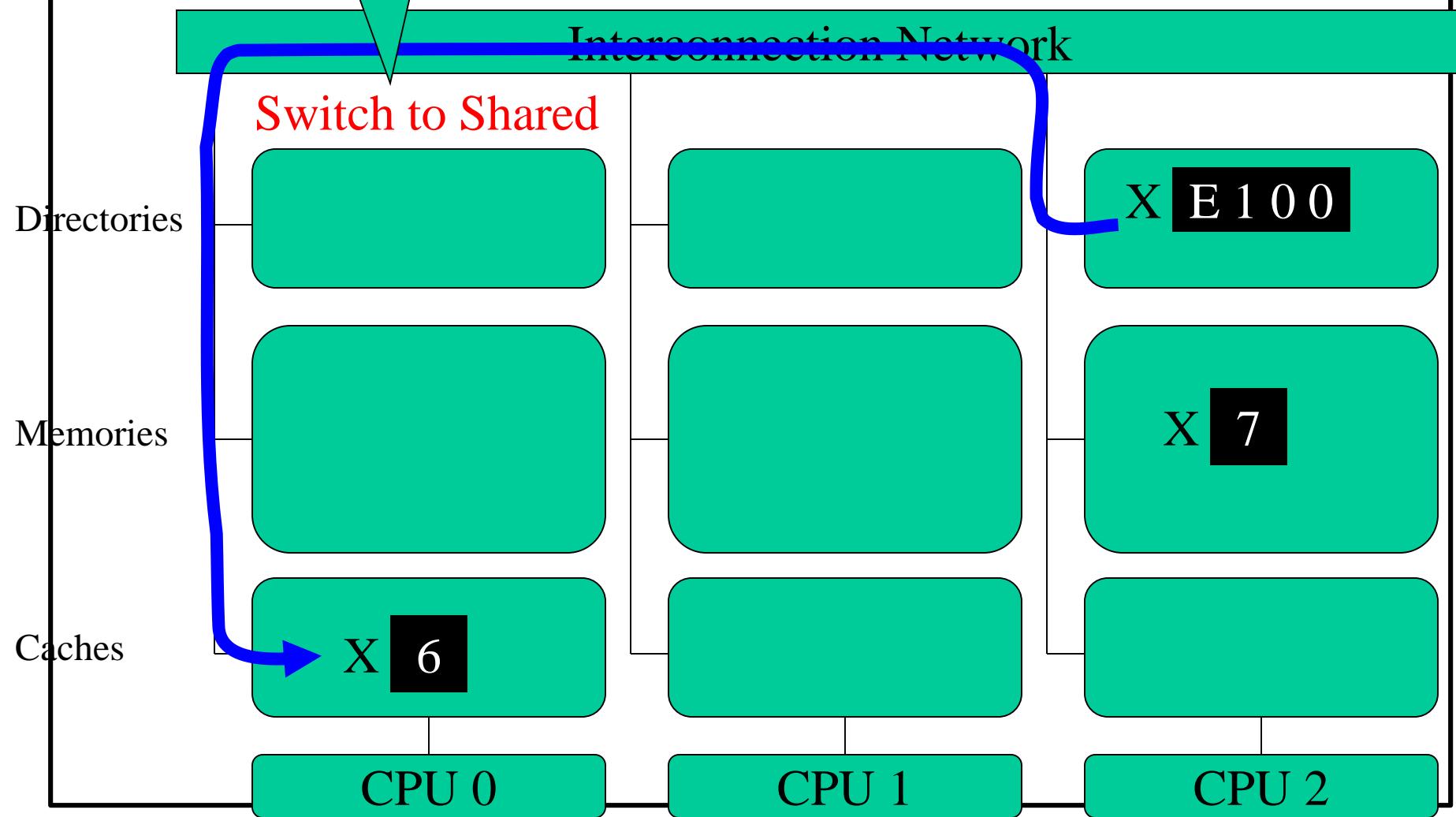
CPU 1 Reads X



This message is sent by
Dir. Con. For CPU 2

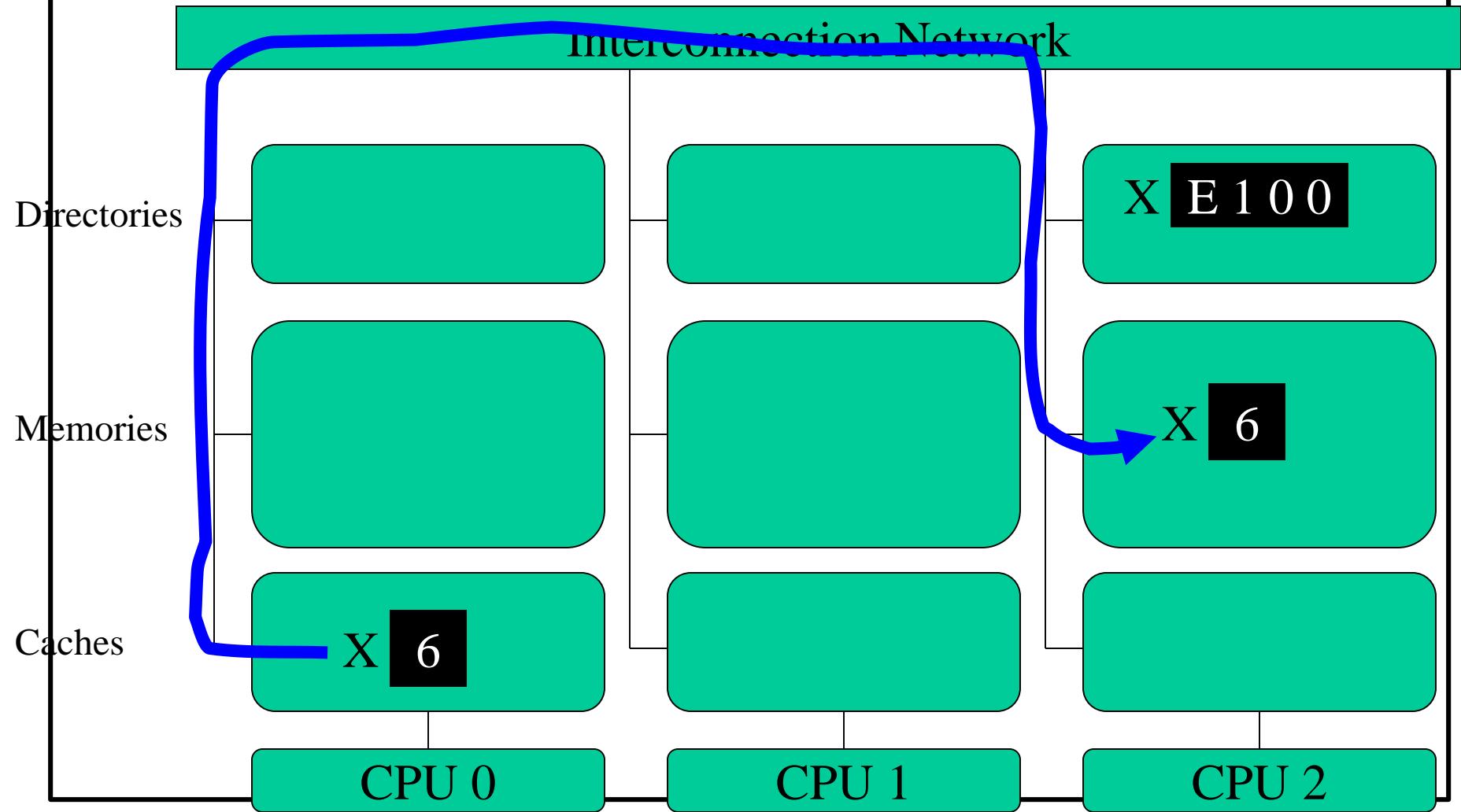
Uncached
Shared
Exclusive

CPU 1 Reads X



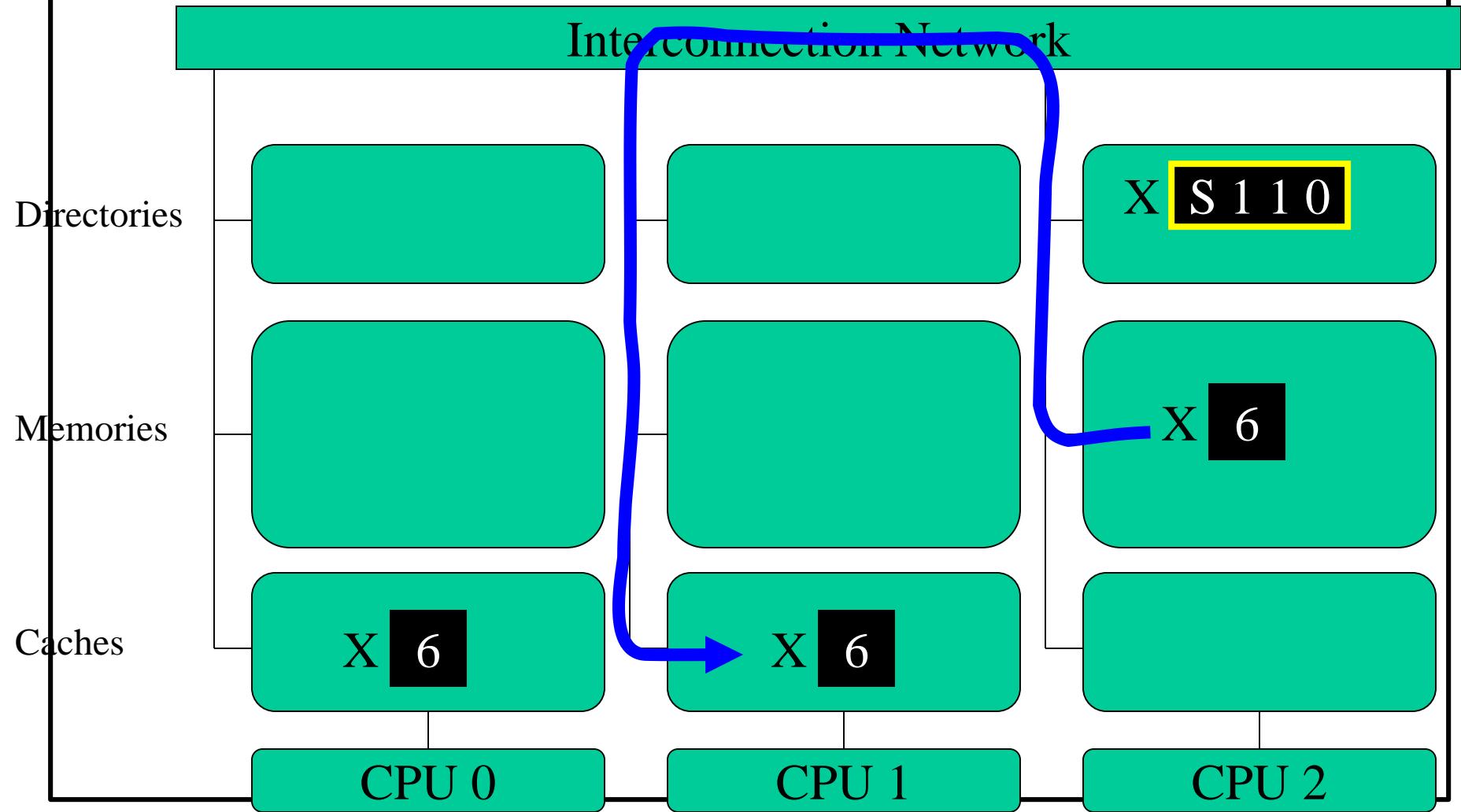
Uncached
Shared
Exclusive

CPU 1 Reads X



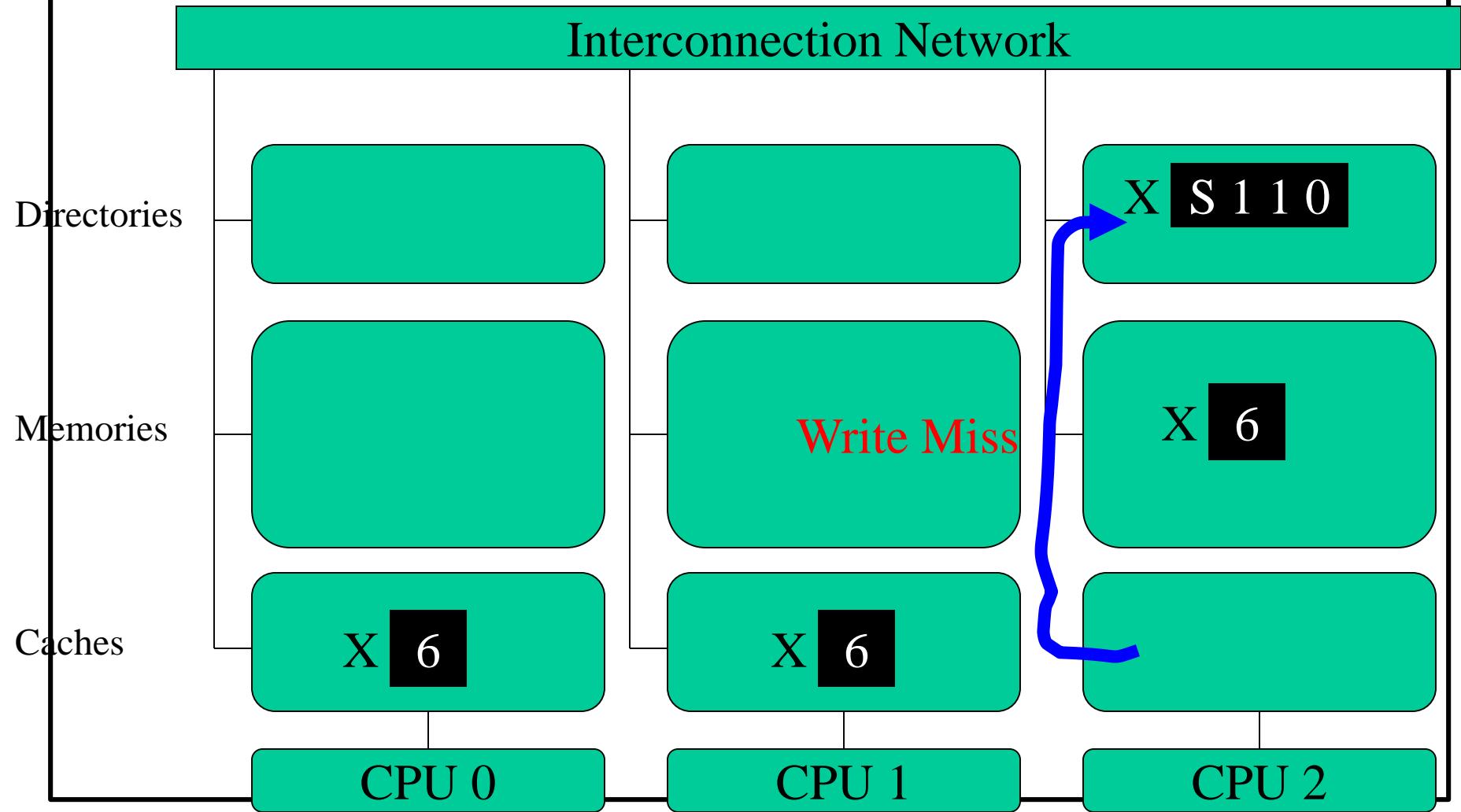
Uncached
Shared
Exclusive

CPU 1 Reads X



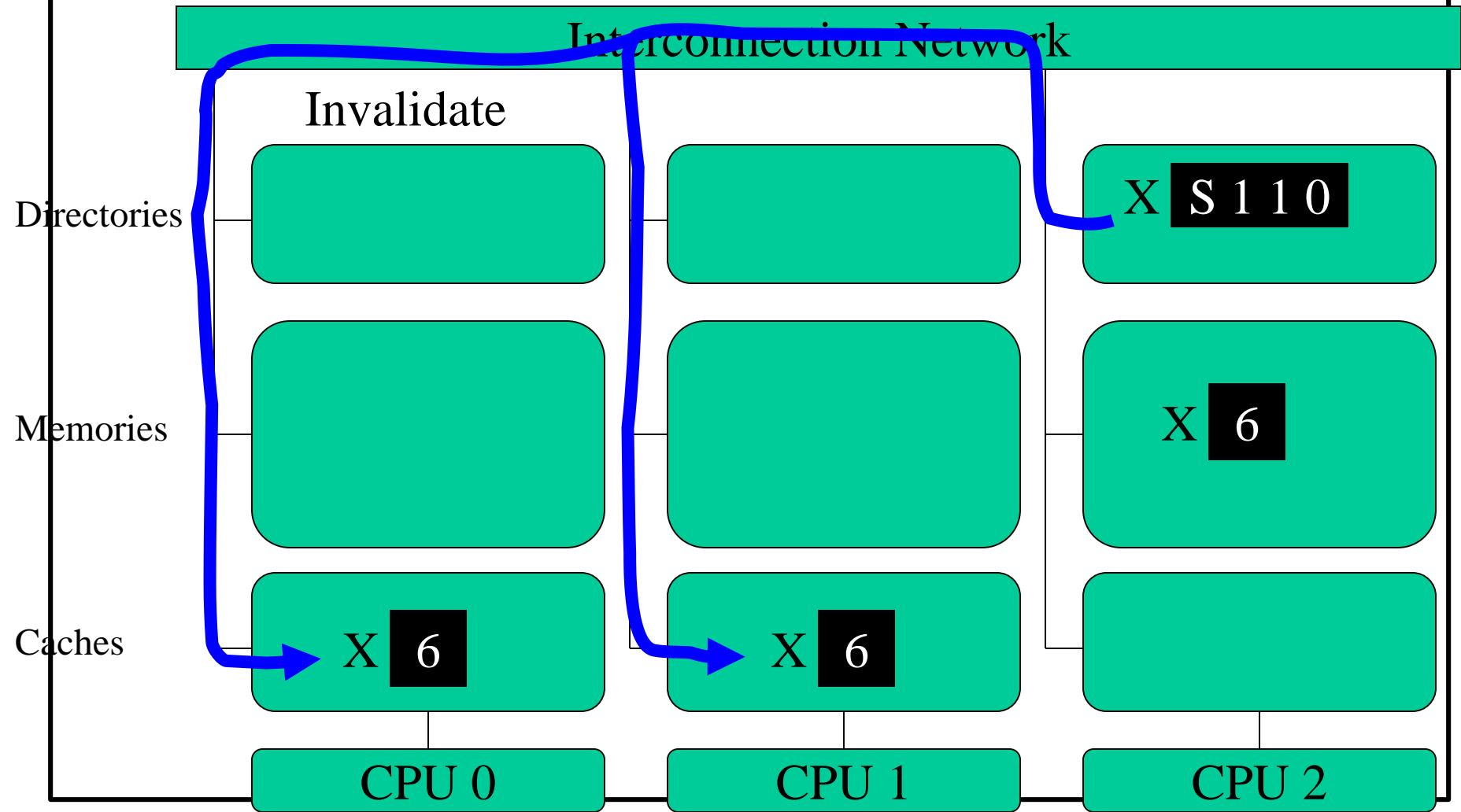
Uncached
Shared
Exclusive

CPU 2 Writes 5 to X



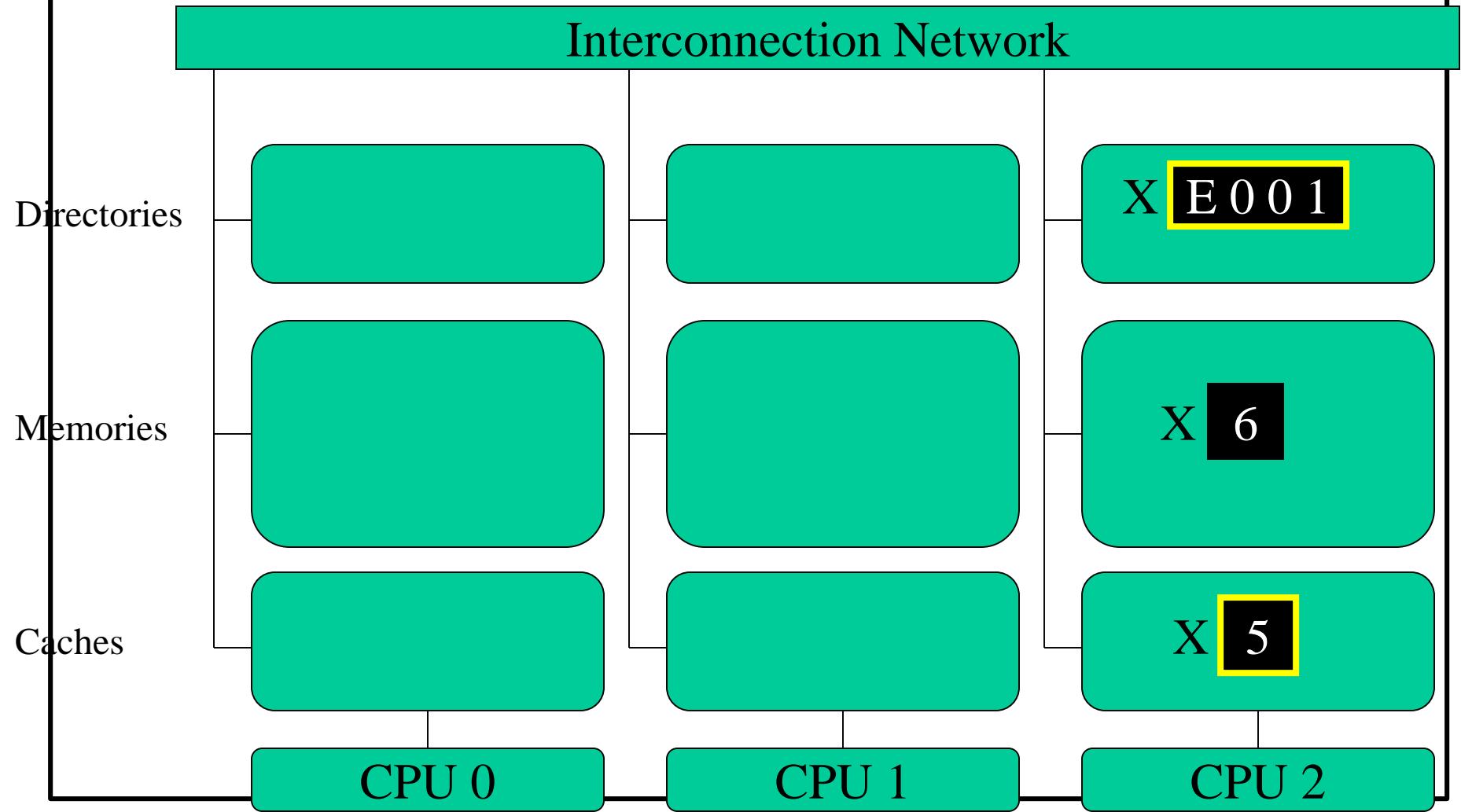
Uncached
Shared
Exclusive

CPU 2 Writes 5 to X



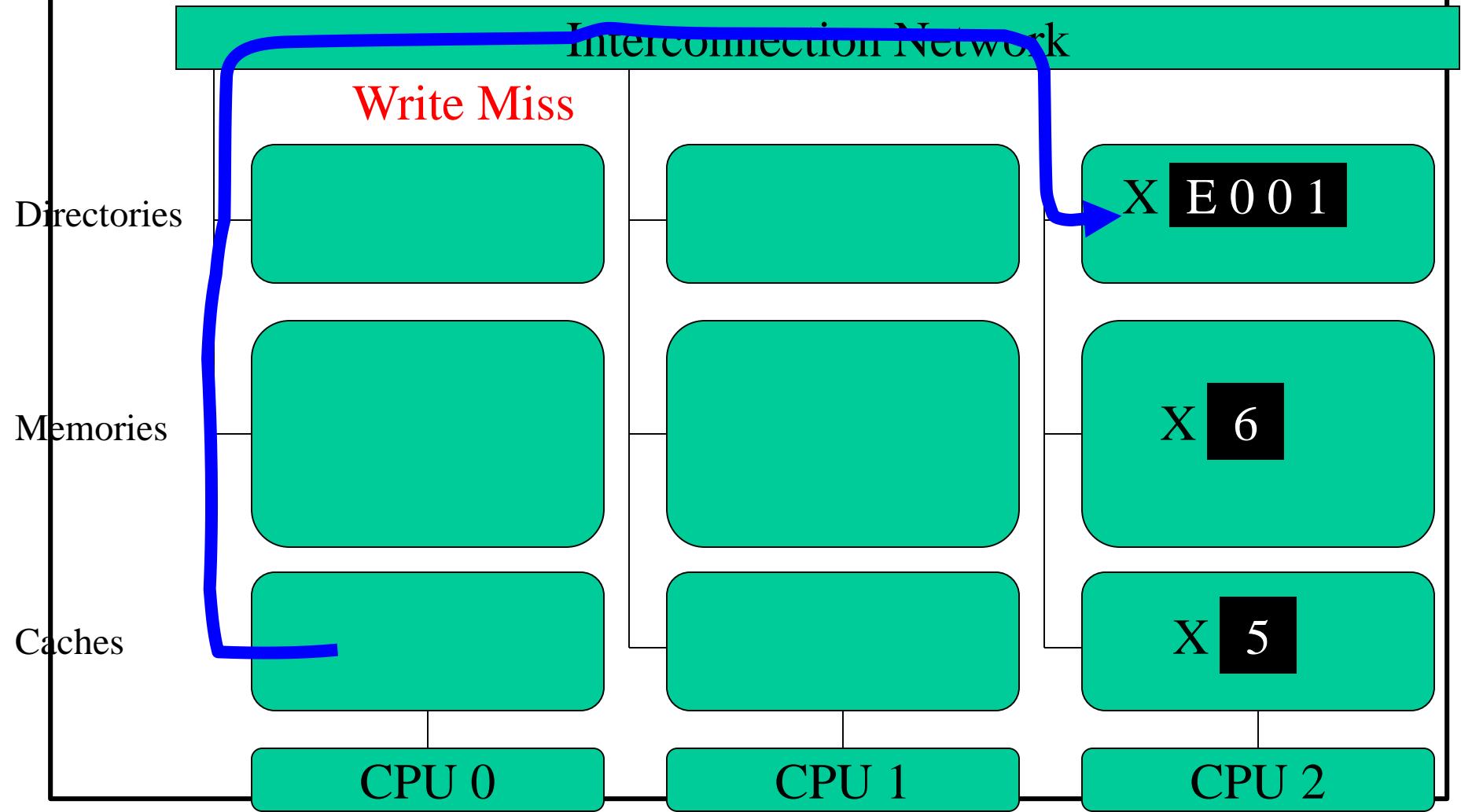
Uncached
Shared
Exclusive

CPU 2 Writes 5 to X



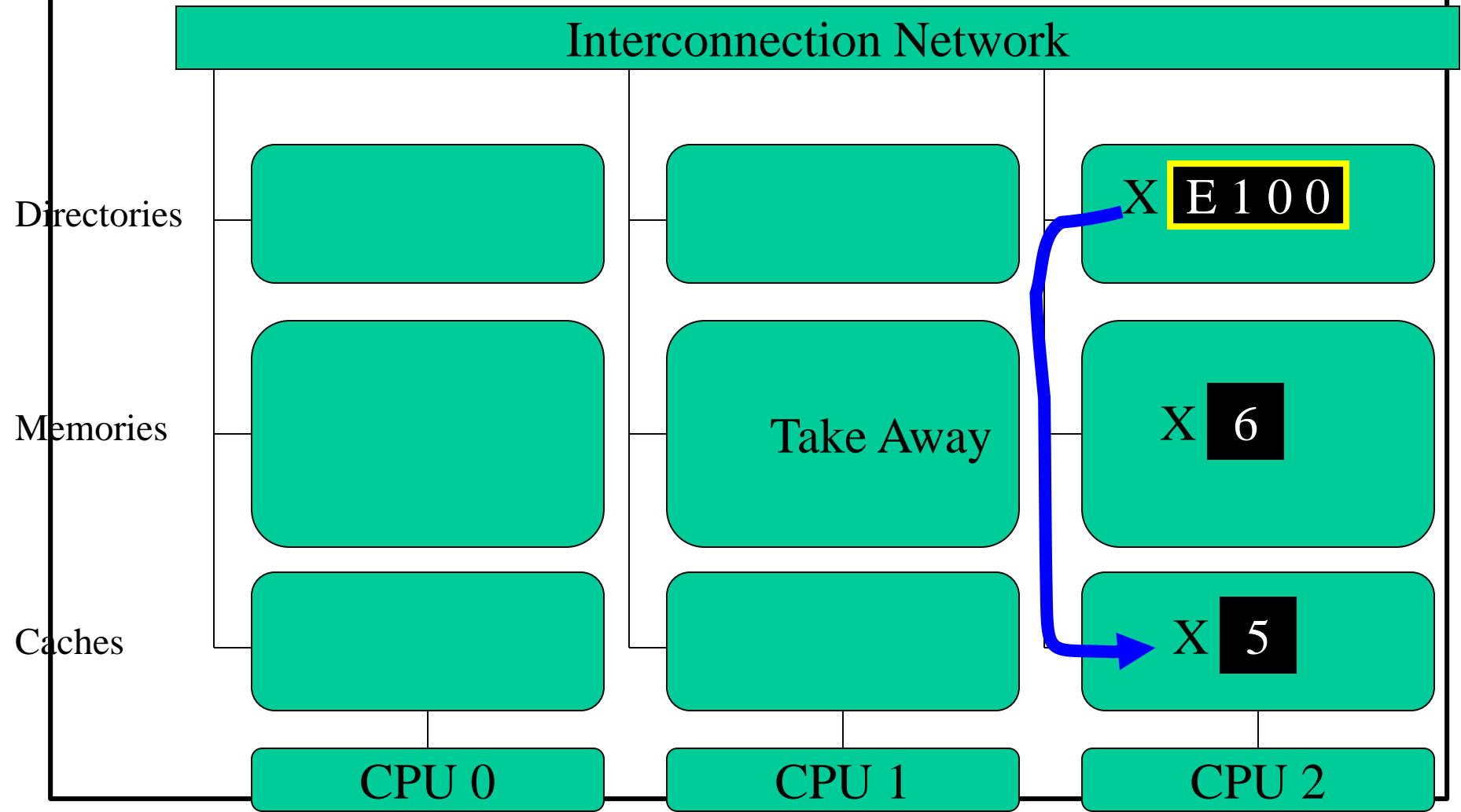
Uncached
Shared
Exclusive

CPU 0 Writes 4 to X



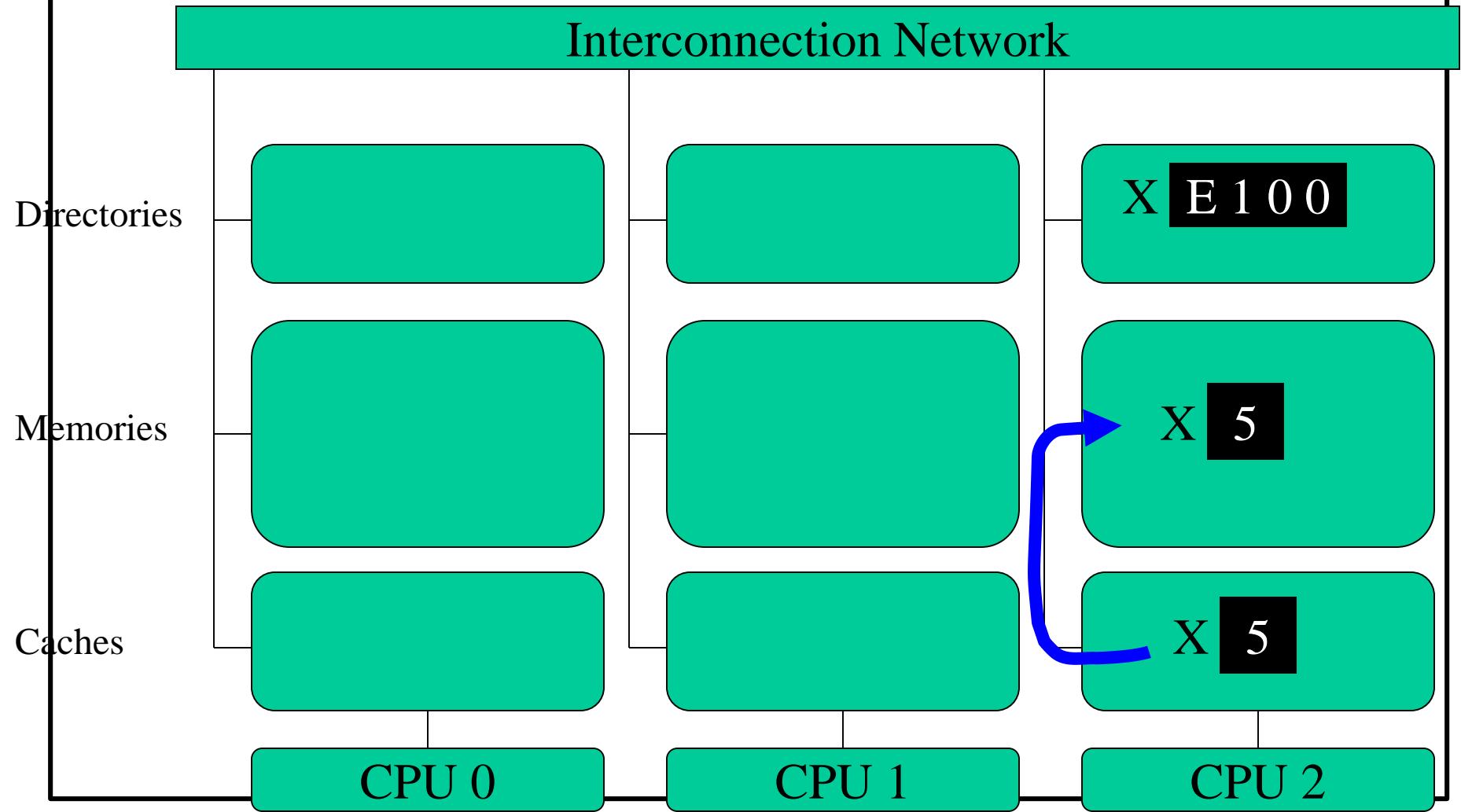
Uncached
Shared
Exclusive

CPU 0 Writes 4 to X



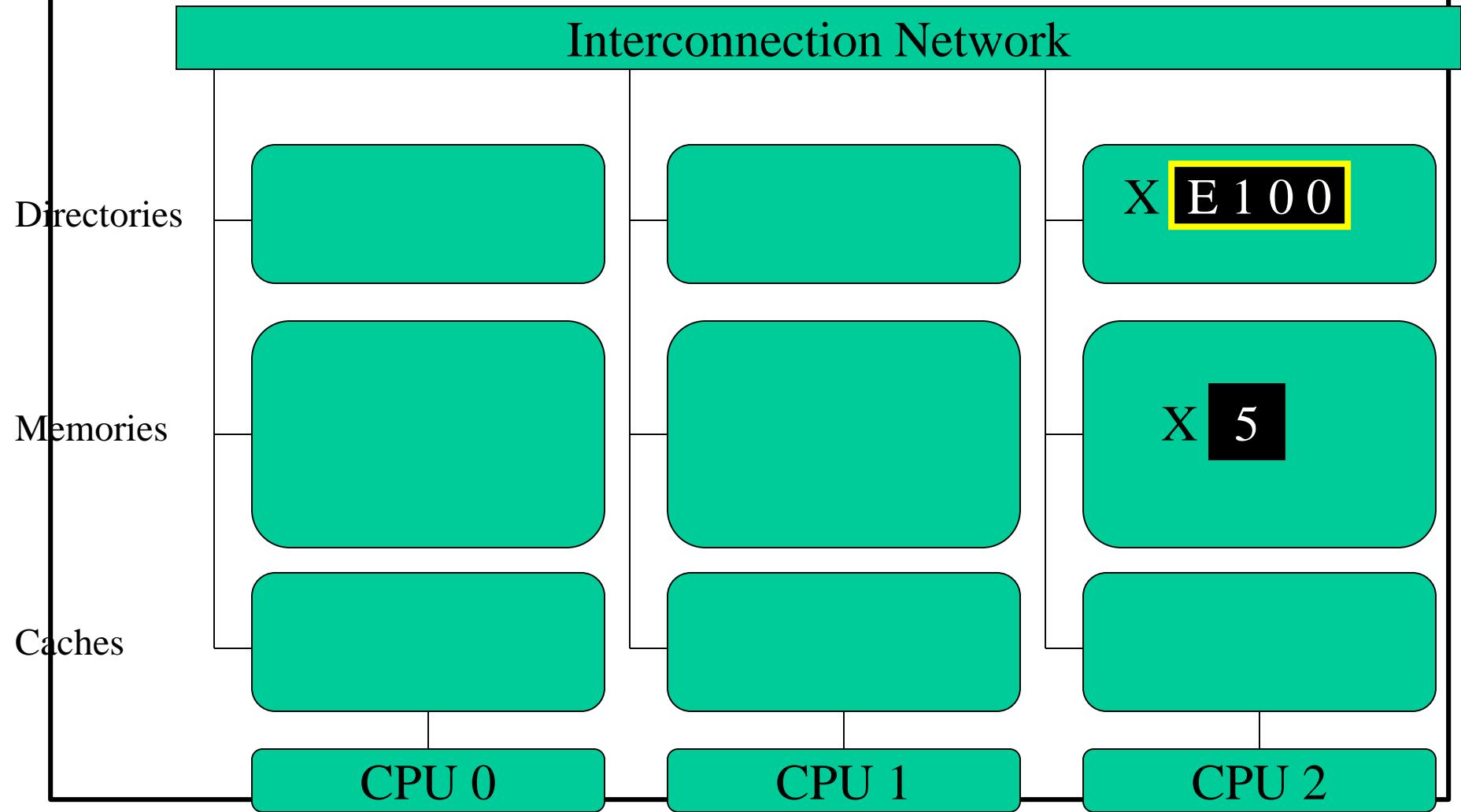
Uncached
Shared
Exclusive

CPU 0 Writes 4 to X



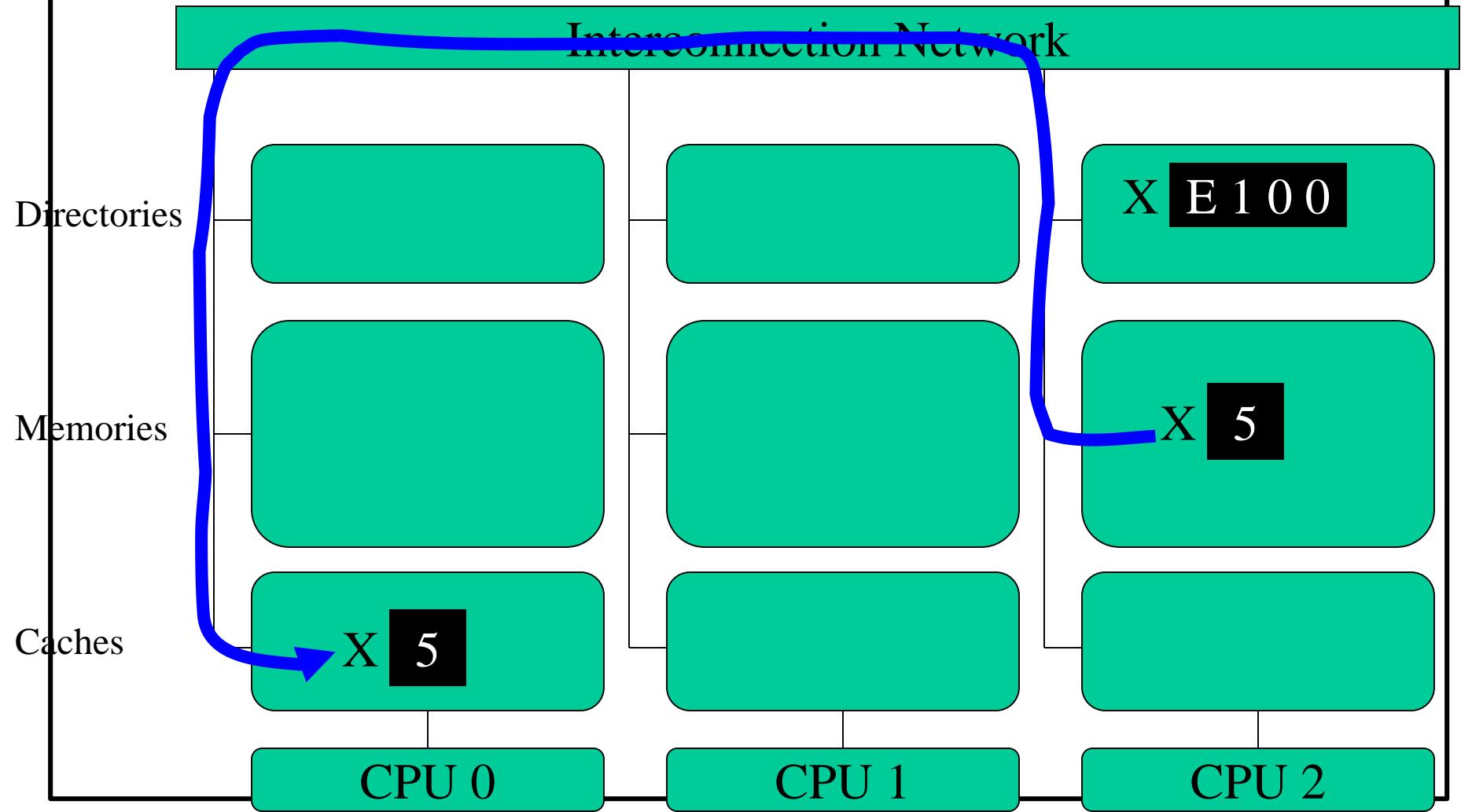
Uncached
Shared
Exclusive

CPU 0 Writes 4 to X



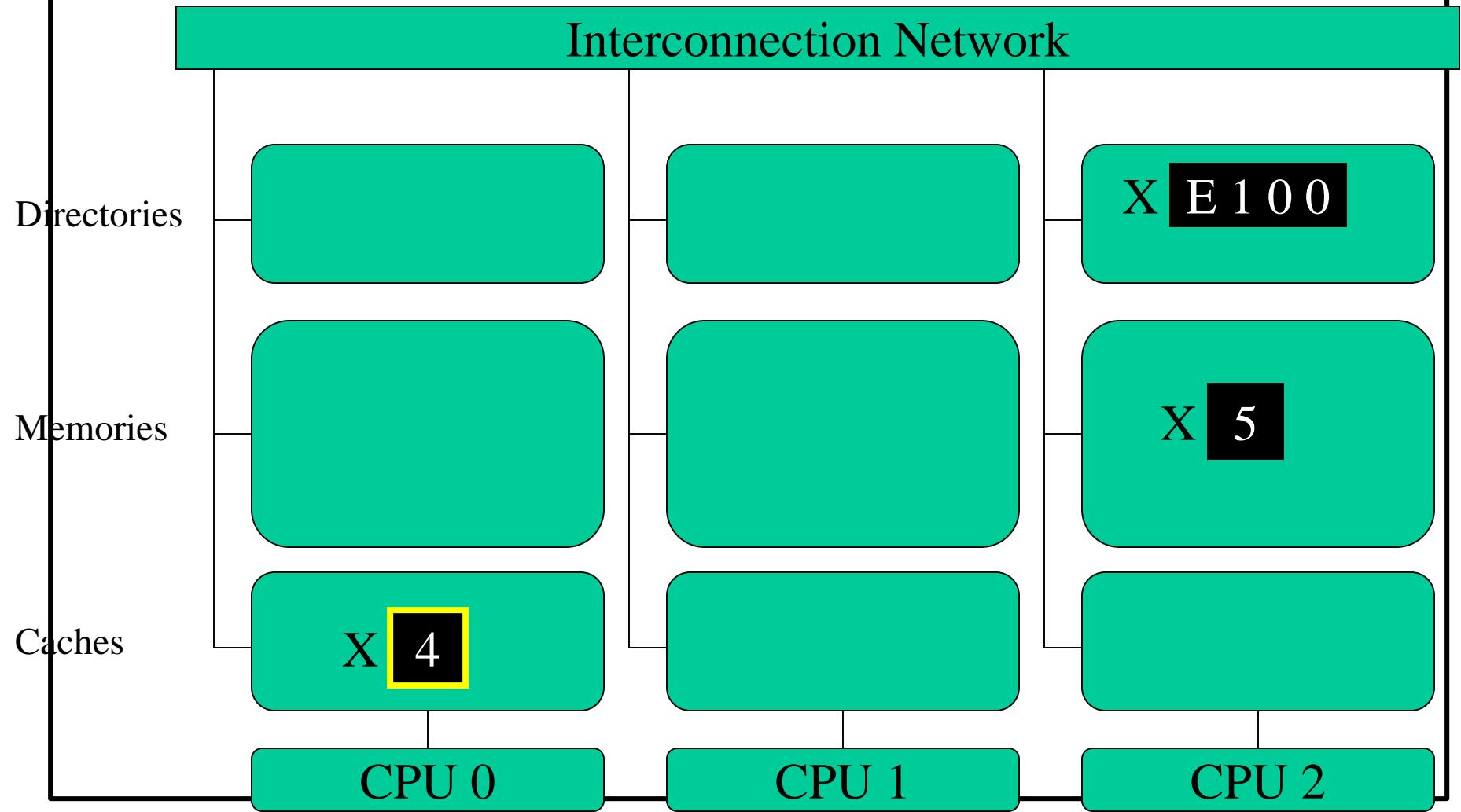
Uncached
Shared
Exclusive

CPU 0 Writes 4 to X



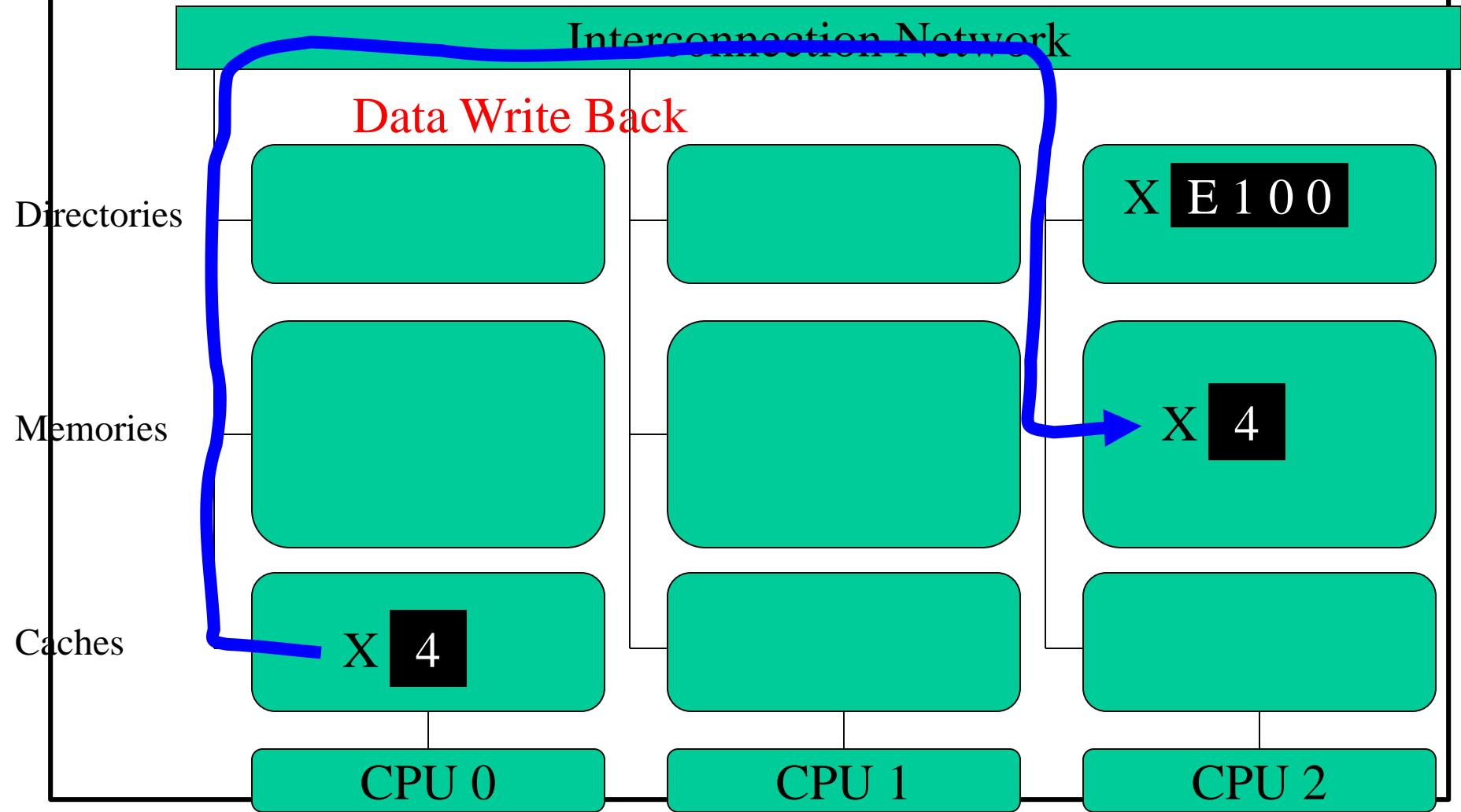
Uncached
Shared
Exclusive

CPU 0 Writes 4 to X



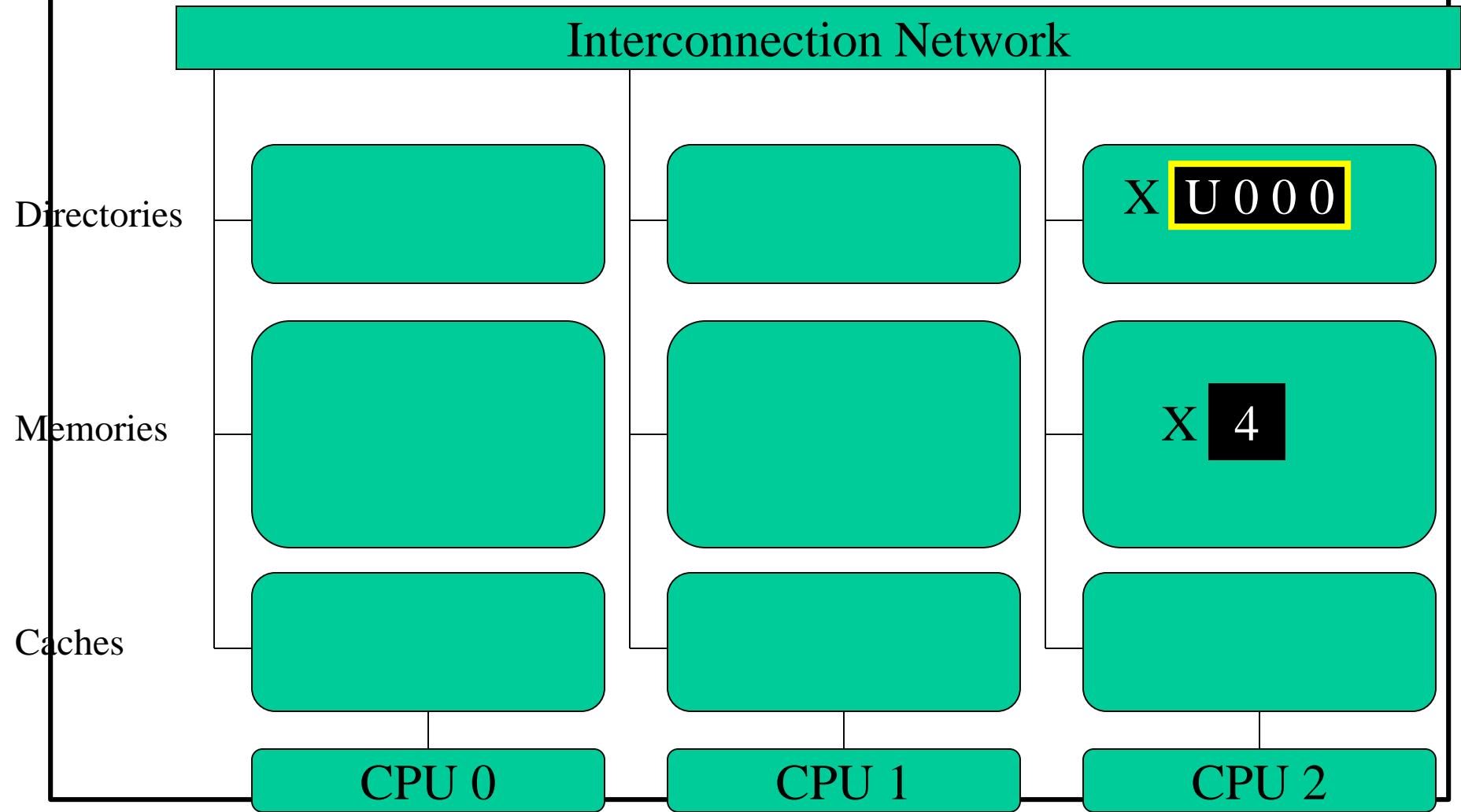
Uncached
Shared
Exclusive

CPU 0 Writes Back X Block



Uncached
Shared
Exclusive

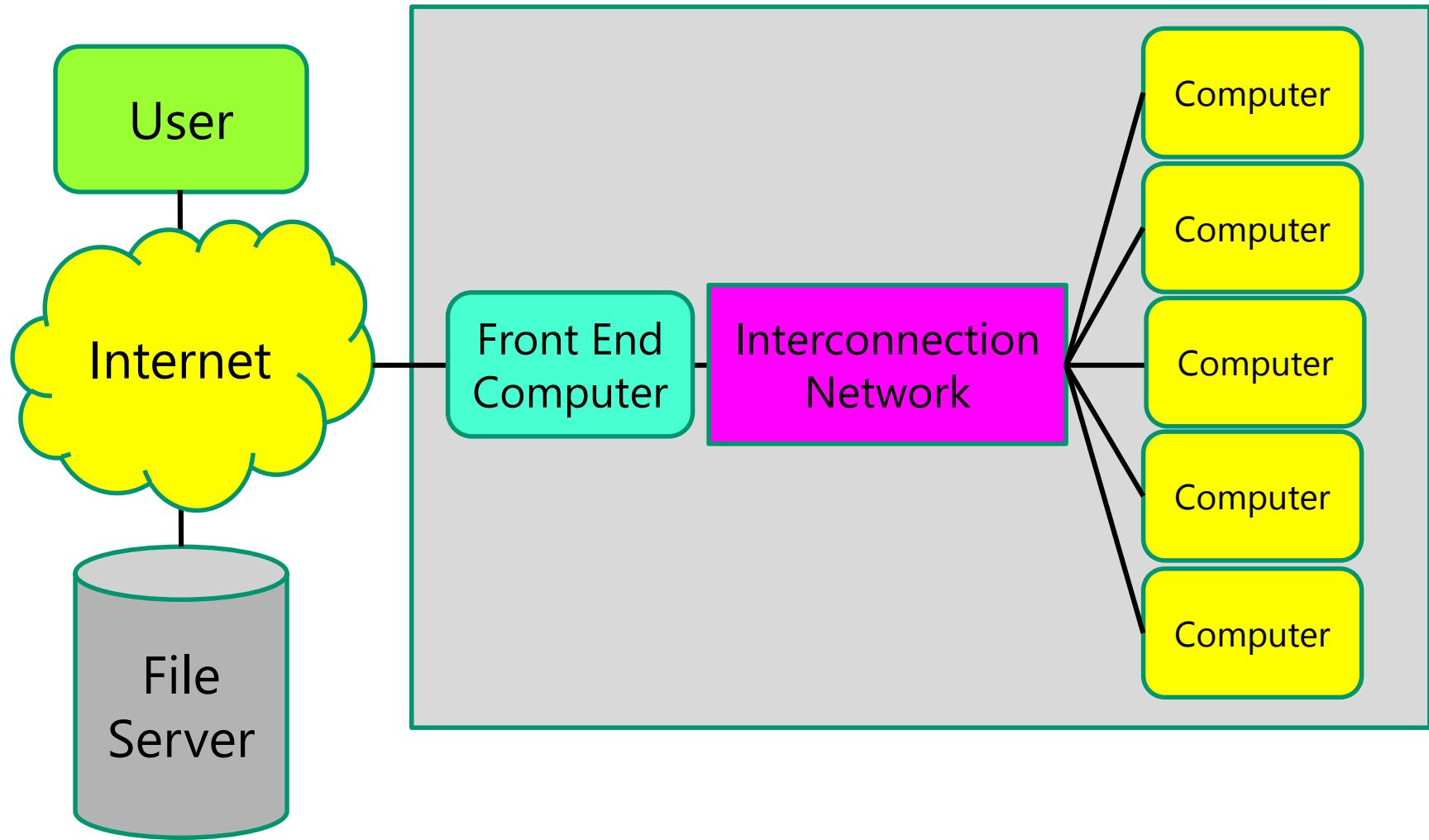
CPU 0 Writes Back X Block



Multicomputers

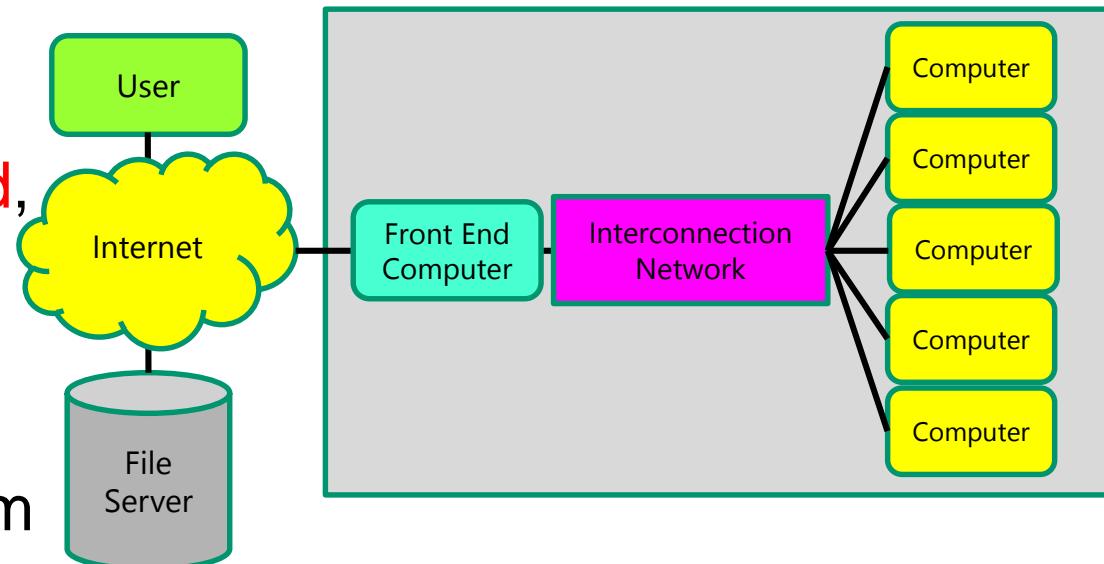
- It has **no shared memory**, each processor has its own memory
 - Interaction is done through the message passing
 - Distributed memory multiple-CPU computer
 - Same address on different processors refers to different physical memory locations
 - Commodity clusters
 - Store and forward message passing
- ★ Cluster Computing, Grid Computing

Asymmetrical Multicomputer



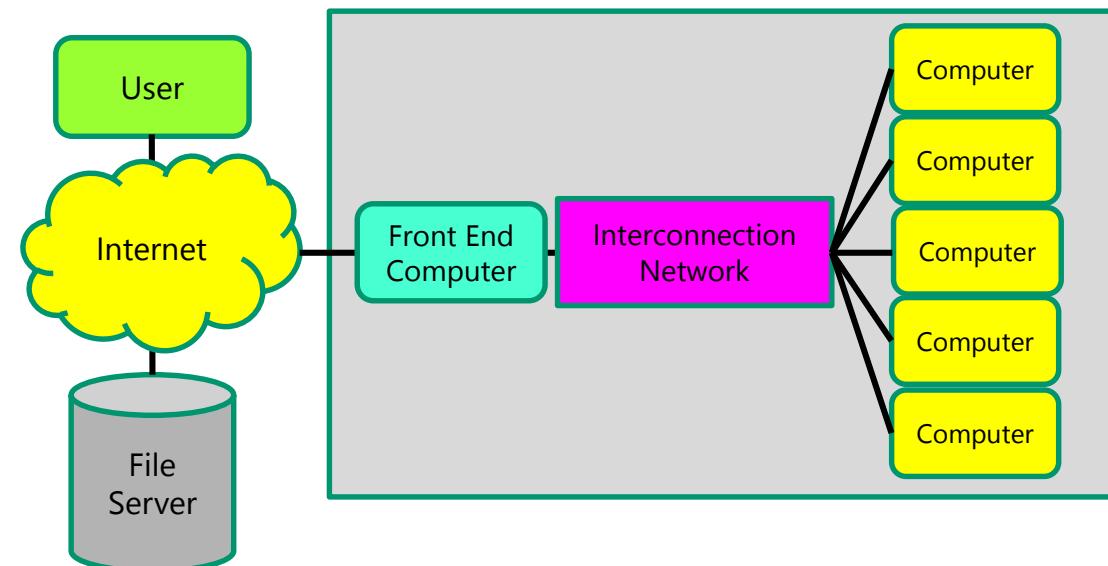
Asymmetrical MC Advantages

- Back-end processors dedicated to parallel computations ⇒ **Easier to understand, model, tune performance**
- Only a simple back-end operating system needed ⇒ **Easy for a vendor to create**

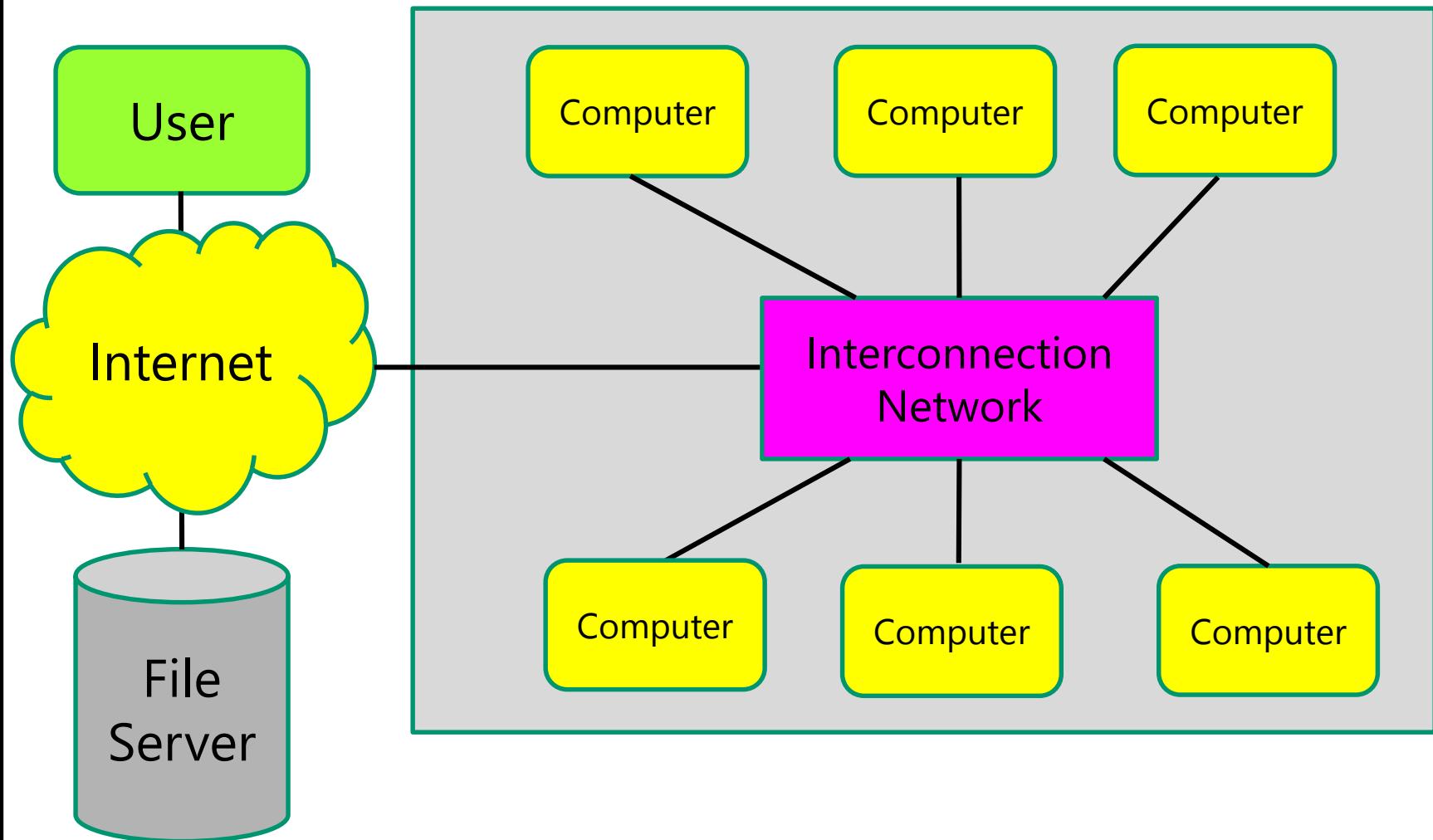


Asymmetrical MC Disadvantages

- Front-end computer is a **single point of failure**
- Single front-end computer **limits scalability of system**
- Primitive operating system in back-end processors makes **debugging difficult**
- Every application requires development of both **front-end and back-end program**

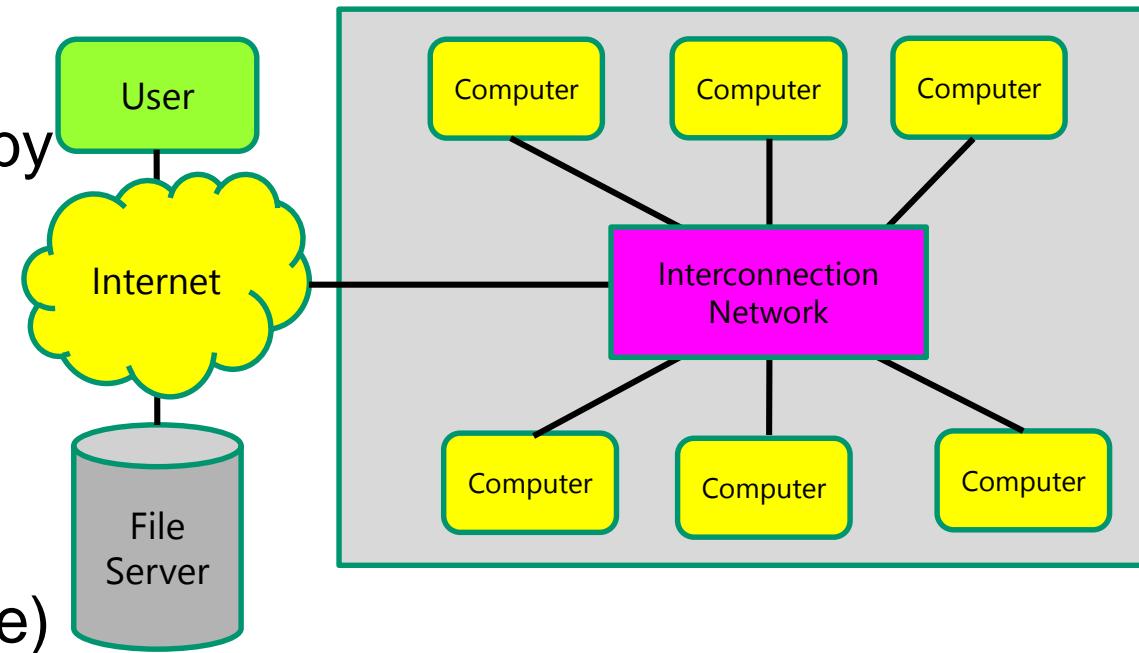


Symmetrical Multicomputer



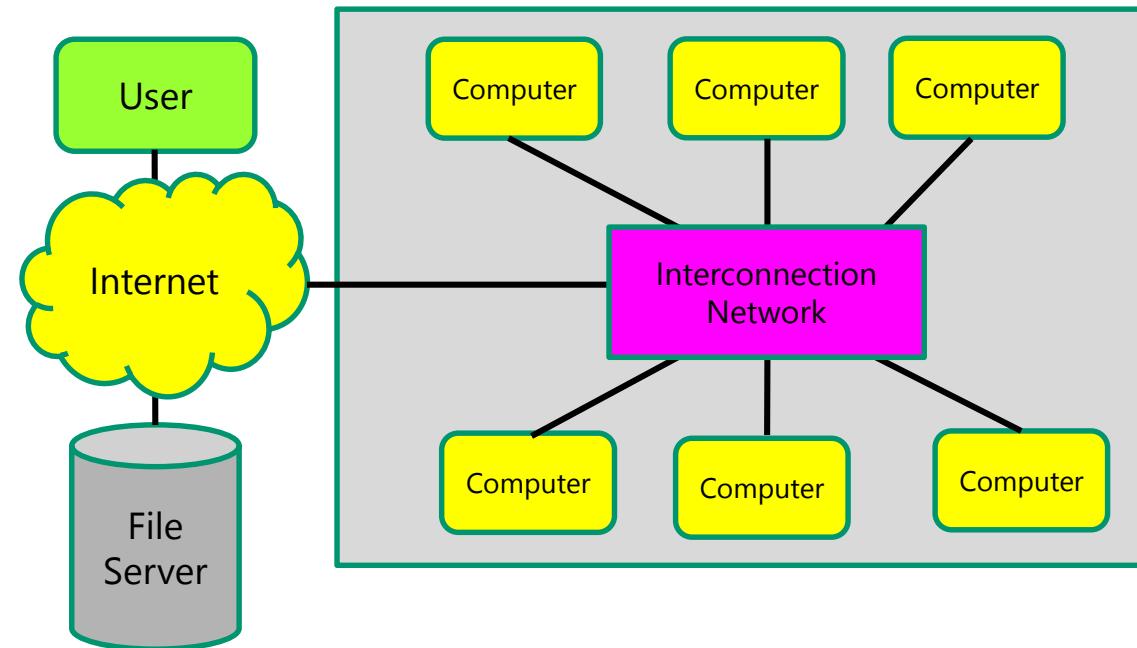
Symmetrical MC Advantages

- Improve performance bottleneck caused by single front-end computer
- Better support for debugging (each node can print debugging message)
- Every processor executes same program

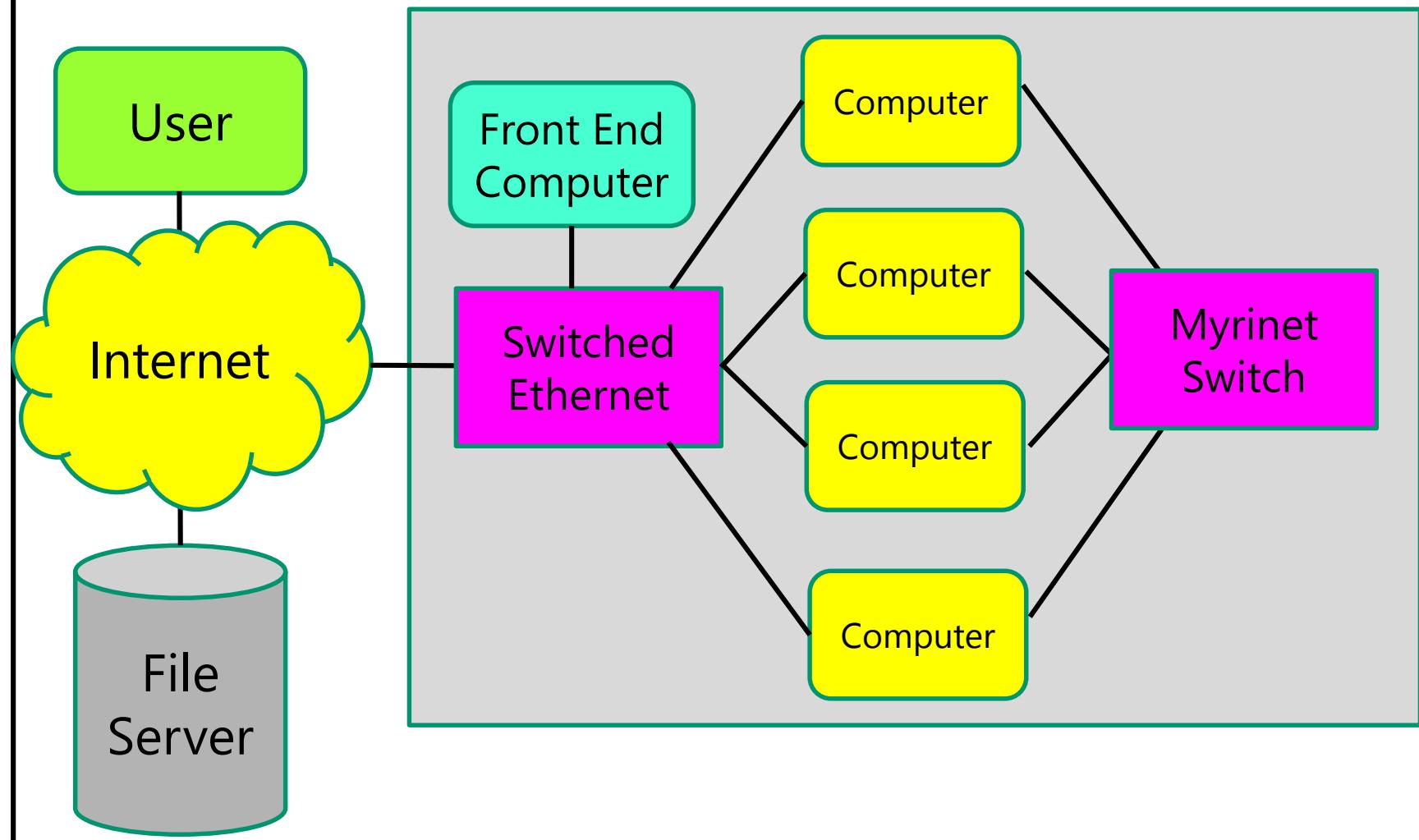


Symmetrical MC Disadvantages

- More difficult to maintain illusion of single “parallel computer”
- No simple way to balance program development workload among processors
- More difficult to achieve high performance when multiple processes on each processor



ParPar Cluster, A Mixed Model



Commodity Cluster

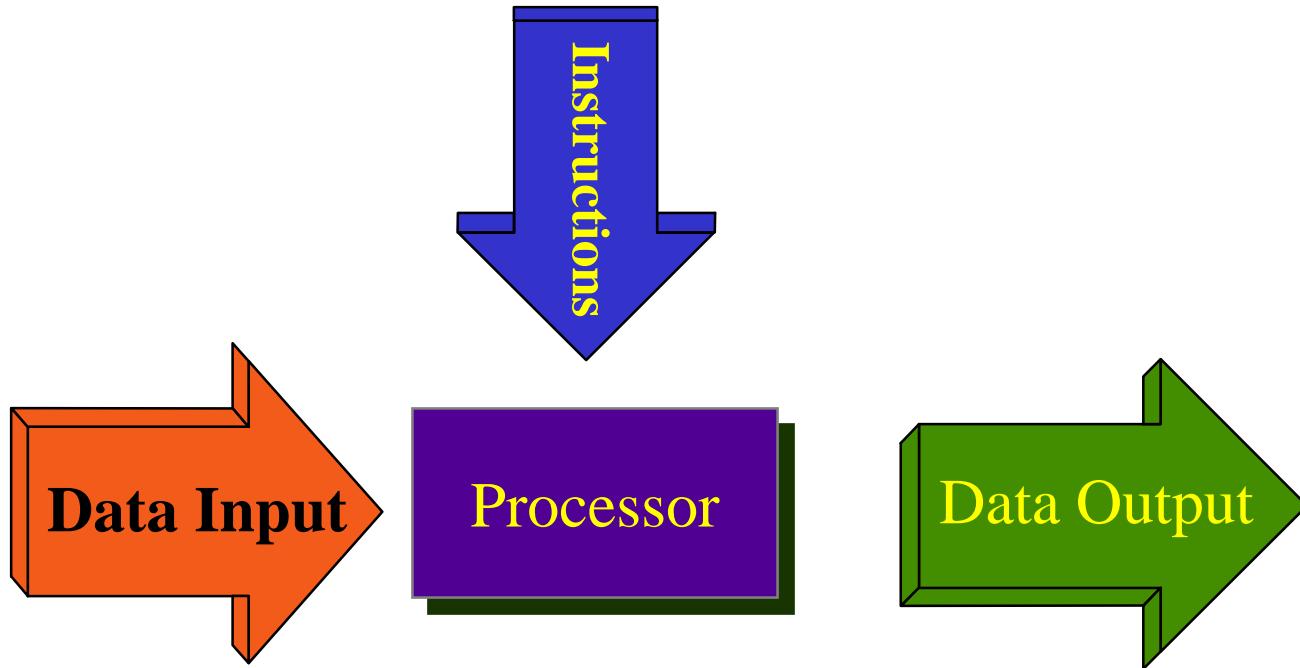
- Co-located computers
- Dedicated to running parallel jobs
- No keyboards or displays
- Identical operating system
- Identical local disk images
- Administered as an entity

Processing Elements Architecture

Processing Elements

- ↗ Simple classification by Flynn (**FLYNN'S TAXONOMY**):
(dual concept of Instruction stream and Data stream.)
 - Ⓐ **SISD** - **conventional**
 - Ⓑ **SIMD** - **data parallel, vector computing, GPU's**
 - Ⓒ **MISD** - **systolic arrays**
 - Ⓓ **MIMD** - **very general, multiple approaches.**
- ↗ Current development is on MIMD model, using general purpose processors.

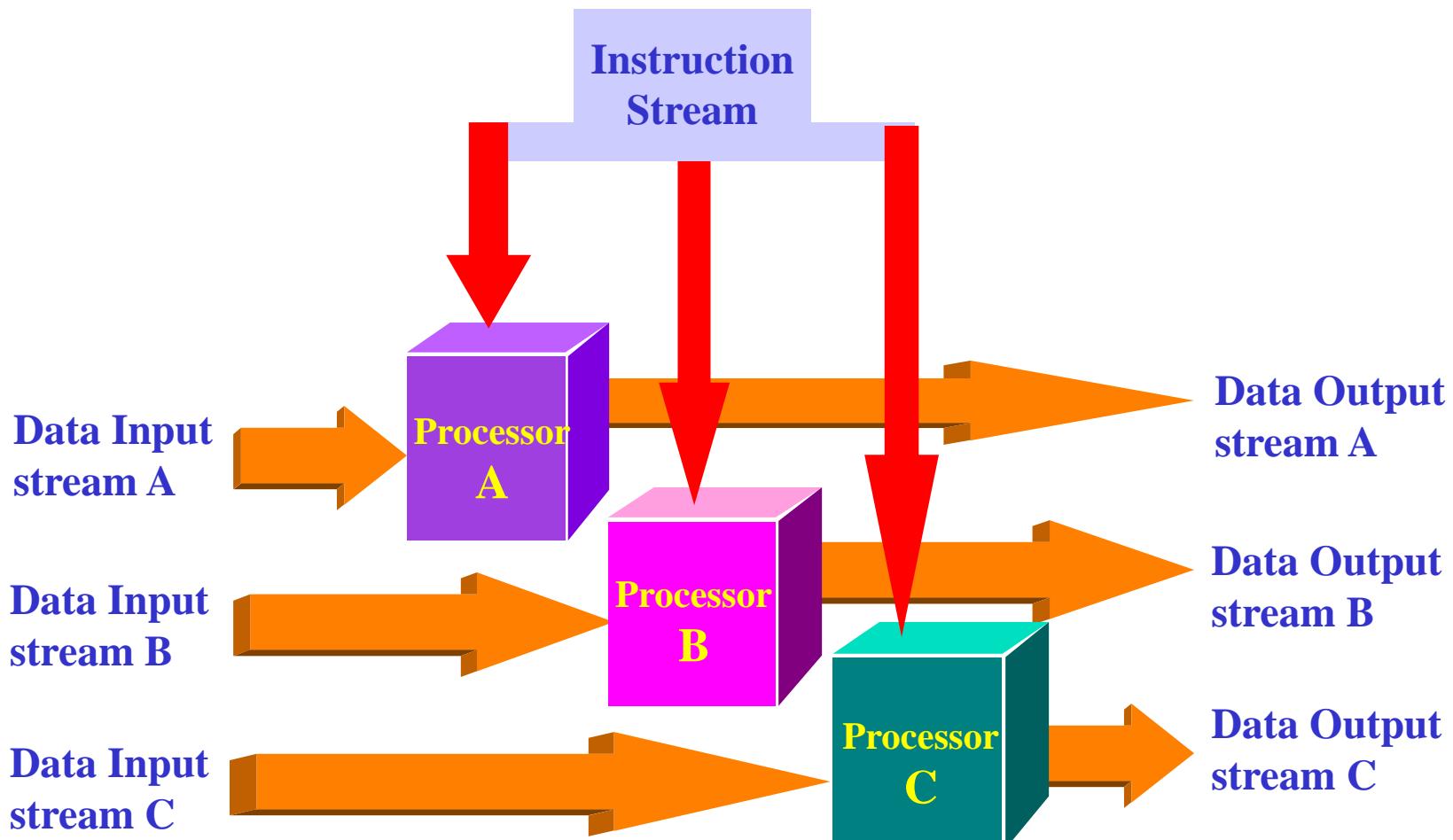
SISD : A Conventional Computer



→ Speed is limited by the rate at which computer can transfer information internally.

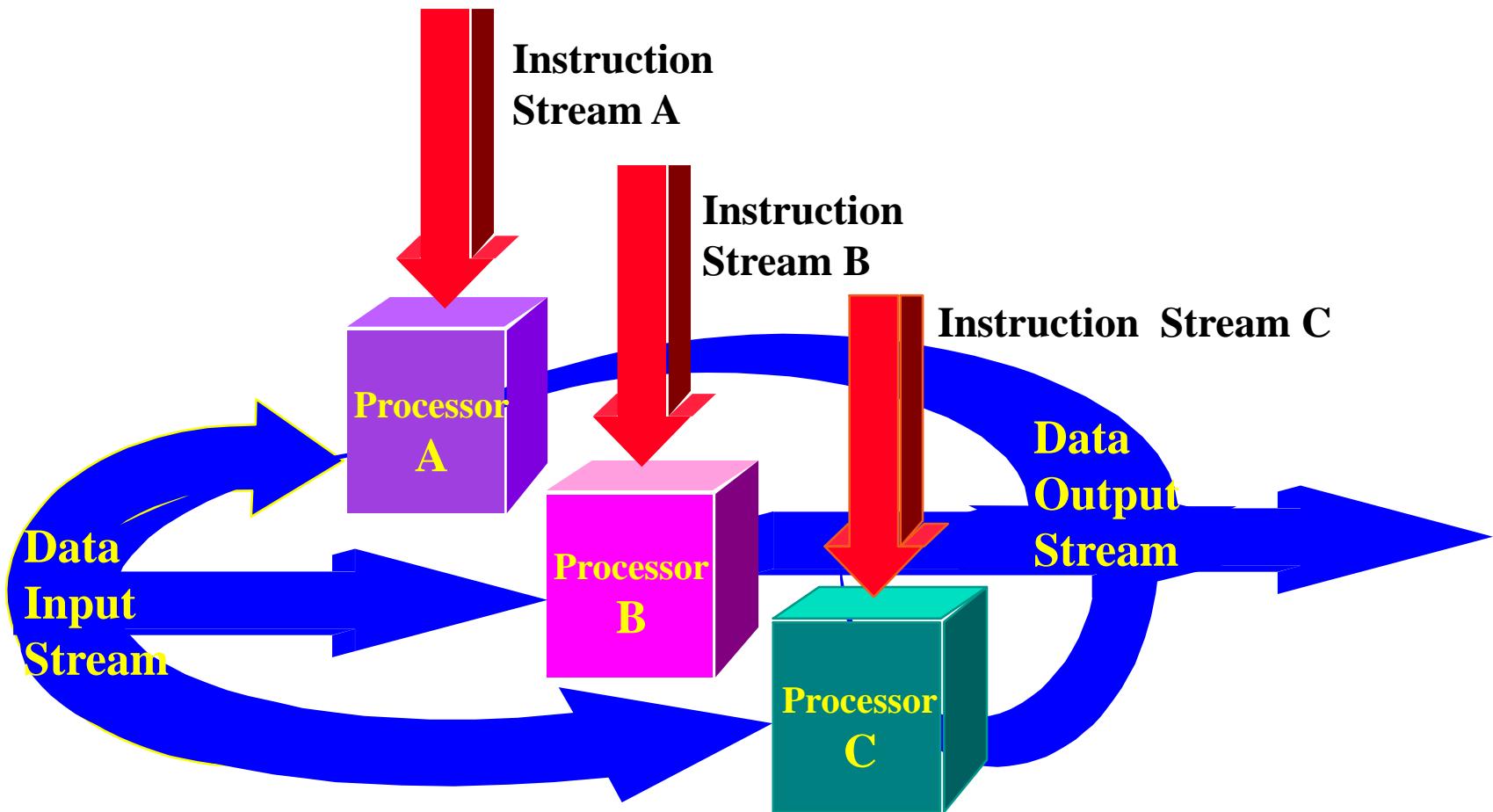
Ex:PC, Macintosh, Workstations

SIMD Architecture



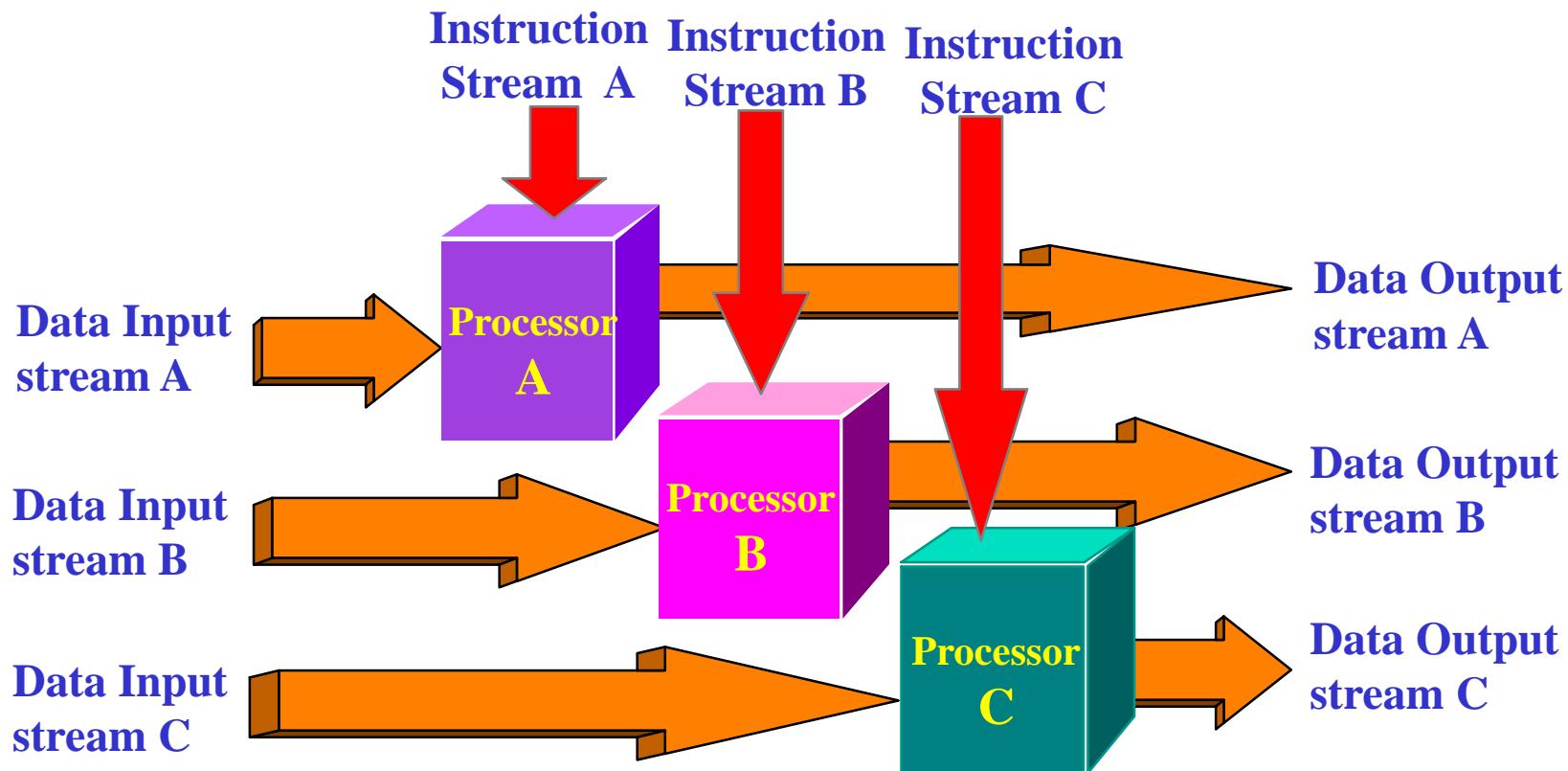
Ex: CRAY machine vector processing, Thinking machine cm*

The MISD Architecture



- Only an intellectual exercise. Few built, but commercially not available

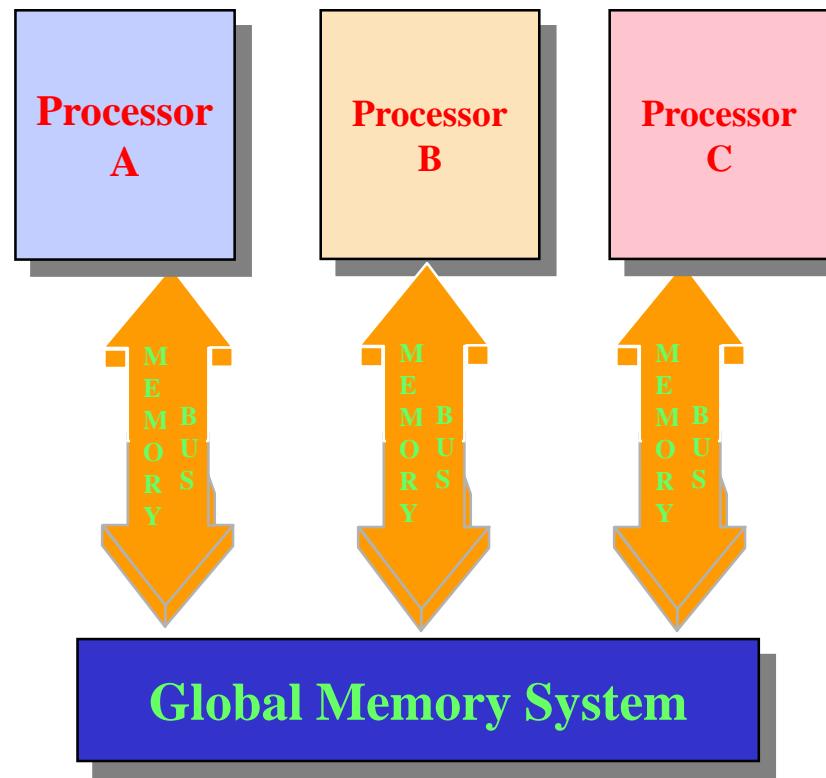
MIMD Architecture



Unlike SISD and MISD a MIMD computer works asynchronously.

- Shared memory (tightly coupled) MIMD
- Distributed memory (loosely coupled) MIMD

Shared Memory MIMD machine

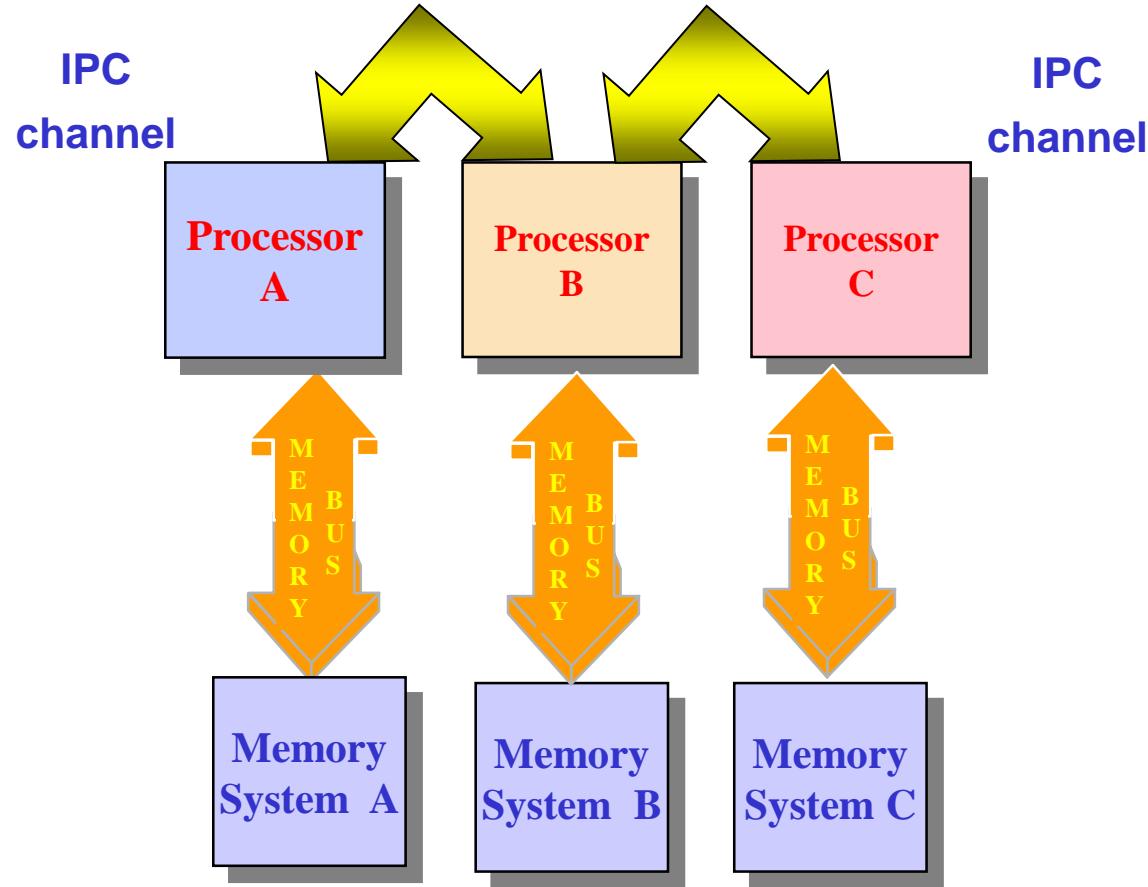


Shared Memory MIMD machine

Comm: Source Processing Elements(PE) writes data to Global Memory(GM) & destination retrieves it

- Easy to build, conventional OSes of SISD can easily be ported
- **Limitation:** reliability & expandability. A memory component or any processor failure affects the whole system.
- Increase of processors leads to memory contention.
Ex. : Silicon graphics supercomputers....

Distributed Memory MIMD



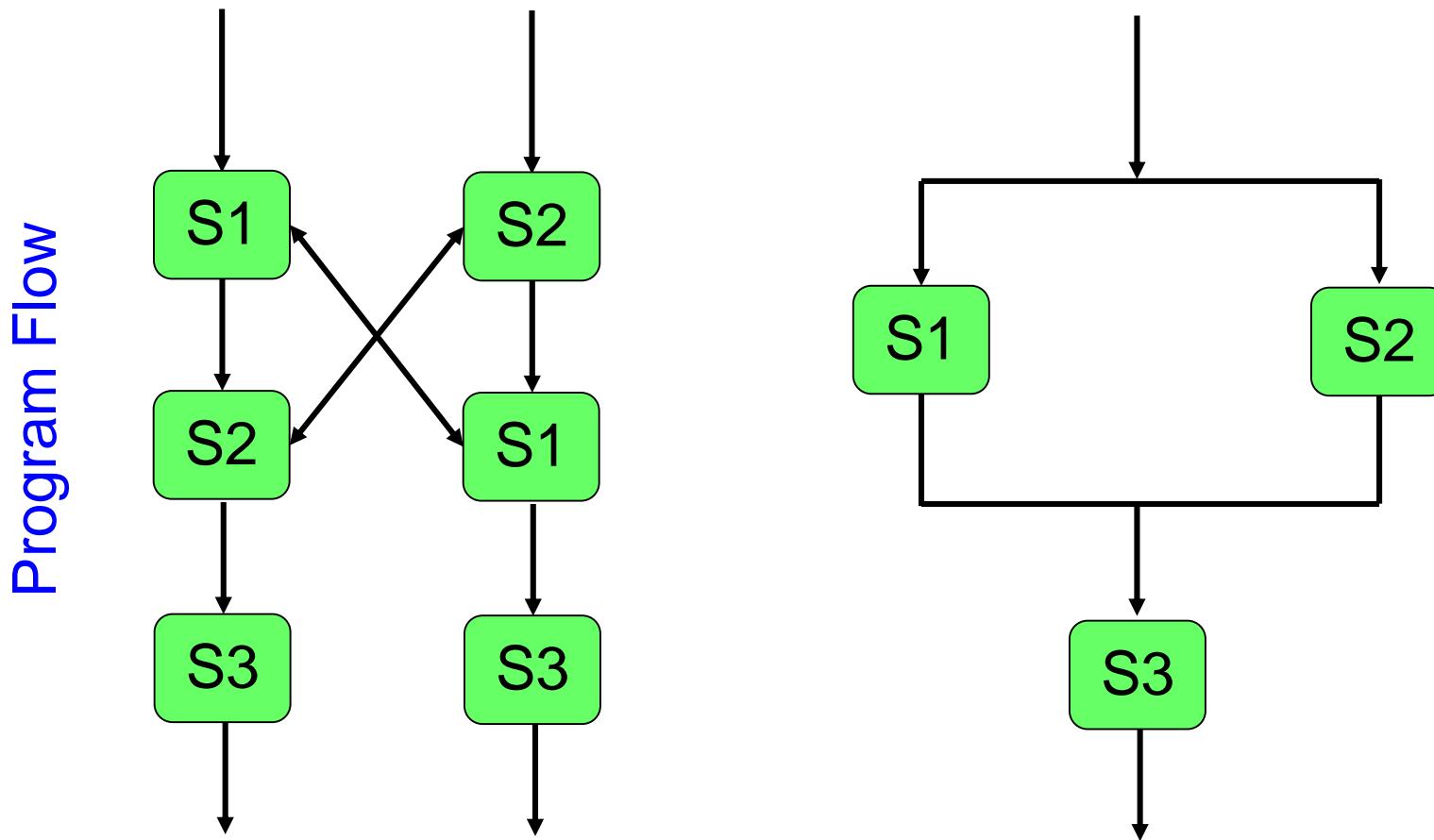
Distributed Memory MIMD

- Communication : IPC on High Speed Network.
- Network can be configured to ... Tree, Mesh, Cube, etc.
- Unlike Shared MIMD
 - easily/ readily expandable
 - Highly reliable (any CPU failure does not affect the whole system)

Classification Based on Grain Size

- The idea is to identify the sub-tasks or instructions in a program that can be executed **in parallel**.
- For example, there are 3 statements in a program and statements S1 and S2 can be exchanged.
- That means, these need not be executed sequentially.
- Then S1 and S2 can be executed in parallel.

Shared memory multiprocessor system



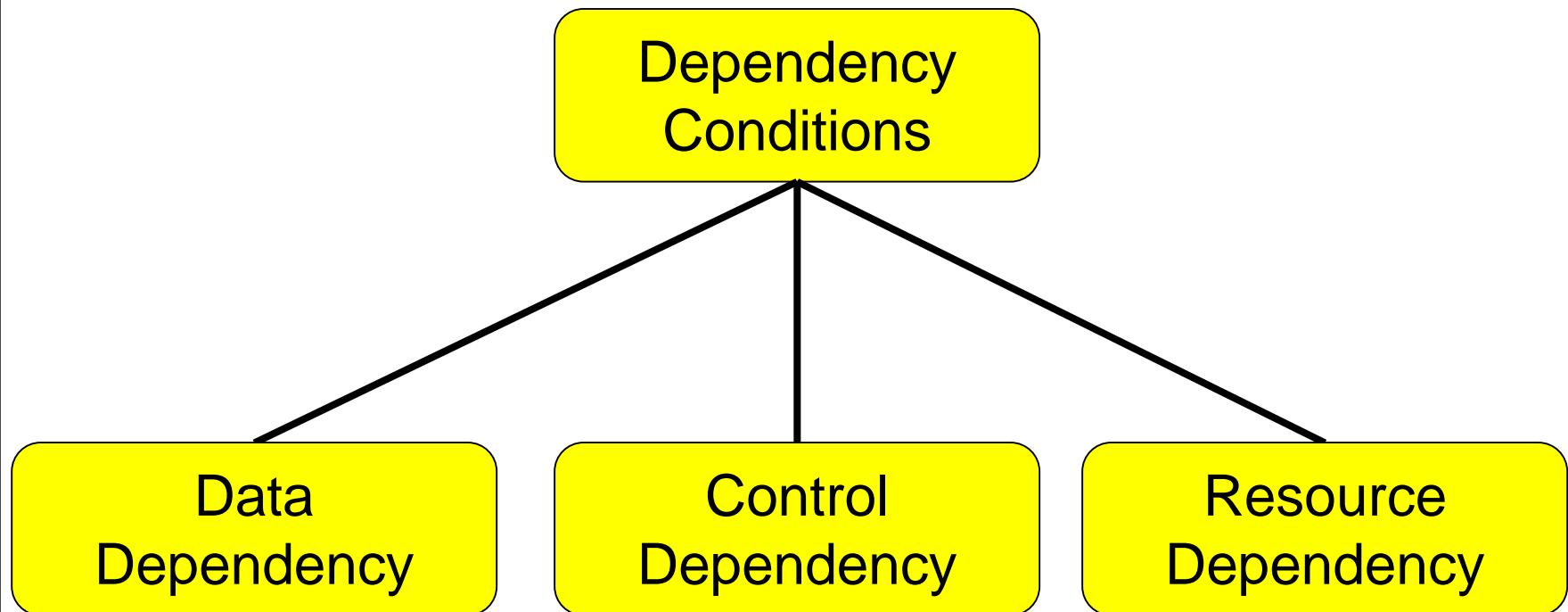
Shared memory multiprocessor system

- Only checking for the parallelism between statements or processes in a program is **not sufficient**. The decision of parallelism also depends on the following factors:
 - Number and types of processors available, i.e., architectural features of host computer
 - Memory organization
 - Dependency of data, control and resources

Parallelism Conditions

- Parallel computing requires that the **segments** to be executed in parallel **must be independent** of each other.
- So, before executing parallelism, all the conditions of parallelism between the segments must be analyzed.
- Three types of dependency conditions between the segments

Parallelism Conditions



Data Dependency

- **Data Dependency:** It refers to the situation in which two or more instructions share same data.
- The instructions in a program can be arranged based on the relationship of data dependency; this means how two instructions or segments are data dependent on each other.
- Four types of data dependencies are recognized:

Data Dependency

- I. **Flow Dependence:** If instruction I_2 follows I_1 and output of I_1 becomes input of I_2 , then I_2 is said to be flow dependent on I_1 .
- II. **Antidependence:** When instruction I_2 follows I_1 such that output of I_2 overlaps with the input of I_1 on the same data.
- III. **Output dependence:** When output of the two instructions I_1 and I_2 overlap on the same data, the instructions are said to be output dependent.
- IV. **I/O dependence:** When read and write operations by two instructions are invoked on the same file, it is a situation of I/O dependence.

Data Dependency

- Consider the following program instructions:
- $I_1: a = b$
- $I_2: c = a + d$
- $I_3: a = c$
- In this program segment instructions I_1 and I_2 are *Flow dependent* because variable a is generated by I_1 as output and used by I_2 as input.

Data Dependency

- Consider the following program instructions:
- $I_1: a = b$
- $I_2: c = a + d$
- $I_3: a = c$
- Instructions I_2 and I_3 are *Antidependent* because variable a is generated by I_3 but used by I_2 and in sequence I_2 comes first.

Data Dependency

- Consider the following program instructions:
- $I_1: a = b$
- $I_2: c = a + d$
- $I_3: a = c$
- I_3 is **flow dependent** on I_2 because of variable c .

Data Dependency

- Consider the following program instructions:
- $I_1: a = b$
- $I_2: c = a + d$
- $I_3: a = c$
- Instructions I_3 and I_1 are **Output dependent** because variable a is generated by both instructions.

Control Dependence

- **Control Dependence:** Instructions or segments in a program may contain control structures. Therefore, dependency among the statements can be in control structures also.
- The order of execution in control structures is not known before the run time. Thus, control structures dependency among the instructions must be analyzed carefully.

Control Dependence

```
For ( i= 1; i<= n ; i++)  
{  
    if (x[i - 1] == 0)  
        x[i] =0  
    else  
        x[i] = 1;  
}
```

Resource Dependence

- **Resource Dependence:** The parallelism between the instructions may also be affected due to the shared resources. If two instructions are using the same shared resource then it is a resource dependency condition.
- For example,
 - floating point units or registers are shared, and this is known as *ALU dependency*.
 - When memory is being shared, then it is called *Storage dependency*.

Bernstein Conditions for Detection of Parallelism

- For execution of instructions or block of instructions in parallel, it should be ensured that the instructions are independent of each other. These instructions can be data dependent / control dependent / resource dependent on each other.
- **A.J. Bernstein** has elaborated the work of data dependency and derived some conditions based on which we can decide the parallelism of instructions or processes.

Bernstein Conditions for Detection of Parallelism

- Bernstein conditions are based on the following two sets of variables:
 - i. The Read set or input set R_i that consists of memory locations read by the statement of instruction I_i .
 - ii. The Write set or output set W_i that consists of memory locations written into by instruction I_i .

The sets R_i and W_i are **not disjoint** as the same locations are used for reading and writing by S_i .

Bernstein Conditions for Detection of Parallelism

- The following are Bernstein Parallelism conditions which are used to determine whether statements are parallel or not:
 1. Locations in R_1 from which S_1 reads and the locations W_2 onto which S_2 writes must be mutually exclusive. That means S_1 does not read from any memory location onto which S_2 writes. It can be denoted as:

$$R_1 \cap W_2 = \emptyset$$

Bernstein Conditions for Detection of Parallelism

2. Similarly, locations in R_2 from which S_2 reads and the locations W_1 onto which S_1 writes must be **mutually exclusive**. That means S_2 does not read from any memory location onto which S_1 writes. It can be denoted as: $R_2 \cap W_1 = \emptyset$
3. The memory locations W_1 and W_2 onto which S_1 and S_2 write, should not be read by S_1 and S_2 . That means R_1 and R_2 should be independent of W_1 and W_2 . It can be denoted as : $W_1 \cap W_2 = \emptyset$

Bernstein Conditions for Detection of Parallelism

- To show the operation of Bernstein's conditions, consider the following instructions of sequential program:

$$I_1 : x = (a + b) / (a * b)$$

$$I_2 : y = (b + c) * d$$

$$I_3 : z = x^2 + (a * e)$$

- Now, the read set and write set of I_1 , I_2 and I_3 are as follows:

$$R_1 = \{a, b\} \quad W_1 = \{x\}$$

$$R_2 = \{b, c, d\} \quad W_2 = \{y\}$$

$$R_3 = \{x, a, e\} \quad W_3 = \{z\}$$

Bernstein Conditions for Detection of Parallelism

$$R_1 = \{a, b\} \quad W_1 = \{x\}$$

$$R_2 = \{b, c, d\} \quad W_2 = \{y\}$$

$$R_3 = \{x, a, e\} \quad W_3 = \{z\}$$

- Now let us find out whether I_1 and I_2 are parallel or not

$$R_1 \cap W_2 = \emptyset$$

$$R_2 \cap W_1 = \emptyset$$

$$W_1 \cap W_2 = \emptyset$$

- That means I_1 and I_2 are independent of each other.

Bernstein Conditions for Detection of Parallelism

$$R_1 = \{a, b\} \quad W_1 = \{x\}$$

$$R_2 = \{b, c, d\} \quad W_2 = \{y\}$$

$$R_3 = \{x, a, e\} \quad W_3 = \{z\}$$

- Similarly for $I_1 \parallel I_3$,

$$R_1 \cap W_3 = \emptyset$$

$$R_3 \cap W_1 \neq \emptyset$$

$$W_1 \cap W_3 = \emptyset$$

- Hence I_1 and I_3 are not independent of each other.

Bernstein Conditions for Detection of Parallelism

$$R_1 = \{a, b\} \quad W_1 = \{x\}$$

$$R_2 = \{b, c, d\} \quad W_2 = \{y\}$$

$$R_3 = \{x, a, e\} \quad W_3 = \{z\}$$

- For $I_2 \parallel I_3$,

$$R_2 \cap W_3 = \emptyset$$

$$R_3 \cap W_2 = \emptyset$$

$$W_3 \cap W_2 = \emptyset$$

- Hence, I_2 and I_3 are independent of each other.
- Thus, I_1 and I_2 , I_2 and I_3 are parallelizable **but I_1 and I_3 are not**.

Bernstein Conditions for Detection of Parallelism

OpenMP & Loop Iteration Data Dependency

- Why it matters?
- OpenMP splits **loop iterations** among threads.
- Safe parallelization only if **no data dependency** exists between iterations.
- For iterations i and j :
- $W_i \cap R_j = \emptyset, W_j \cap R_i = \emptyset, W_i \cap W_j = \emptyset$

Bernstein Conditions for Detection of Parallelism

OpenMP & Loop Iteration Data Dependency

```
#pragma omp parallel for  
for (int i=0; i<n; i++)  
    a[i] = b[i] + c[i];
```

- Each iteration:
 $R_i = \{b[i], c[i]\}, W_i = \{a[i]\}$
- All disjoint → **Safe to parallelize**

Bernstein Conditions for Detection of Parallelism

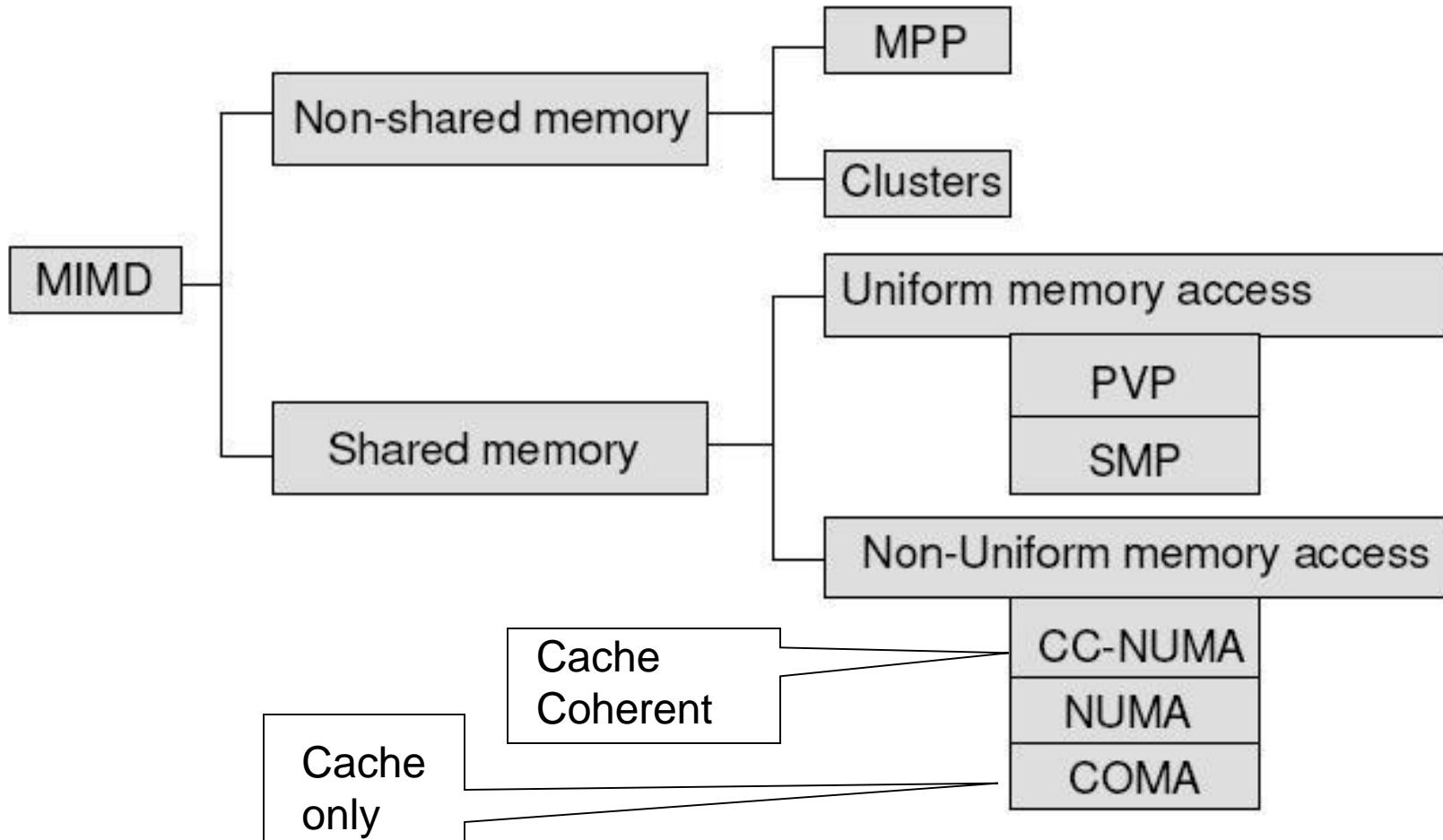
OpenMP & Loop Iteration Data Dependency

```
for (int i=1; i<n; i++)  
    a[i] = a[i-1] + b[i];
```

- $R_i = \{a[i-1], b[i]\}$, $W_i = \{a[i]\}$
- $W_{i-1} \cap R_i = \{a[i-1]\} \neq \emptyset \rightarrow \text{Loop-carried dependency}$

Unit 2.2 Follows

Architectural Model of MIMD



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 is more convenient although it does require access to shared data to be controlled by the programmer (using critical sections etc.)

Several Alternatives for Programming Shared Memory Multiprocessors:

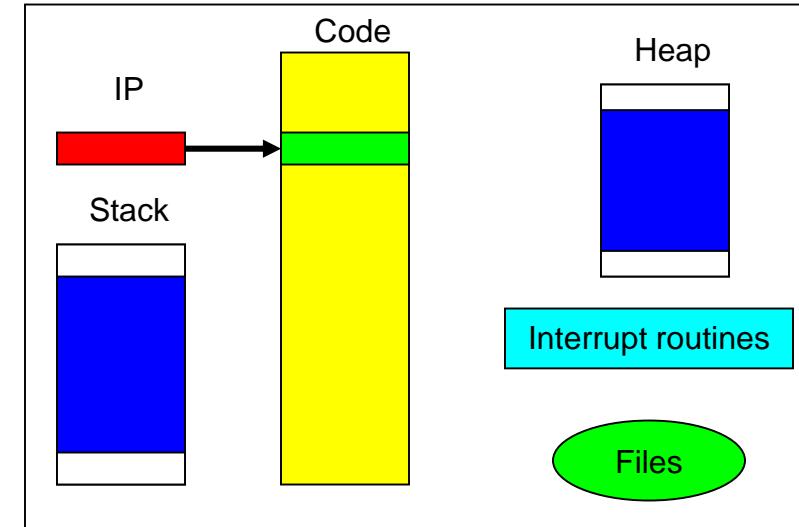
- Using heavy weight processes.
- Using threads. Example **Pthreads**.
- Using a completely new programming language for parallel programming Example **Ada** or **CilkPlus**.
- Using library routines with an existing sequential programming language.
- Modifying the syntax of an existing sequential programming language to create a \parallel programming language. Example **UPC**.
- Using an existing sequential programming language supplemented with compiler directives for specifying parallelism. Example **OpenMP**.
- Using **Threading Building Block (TBB)** from Intel.

Memory Layout Summary

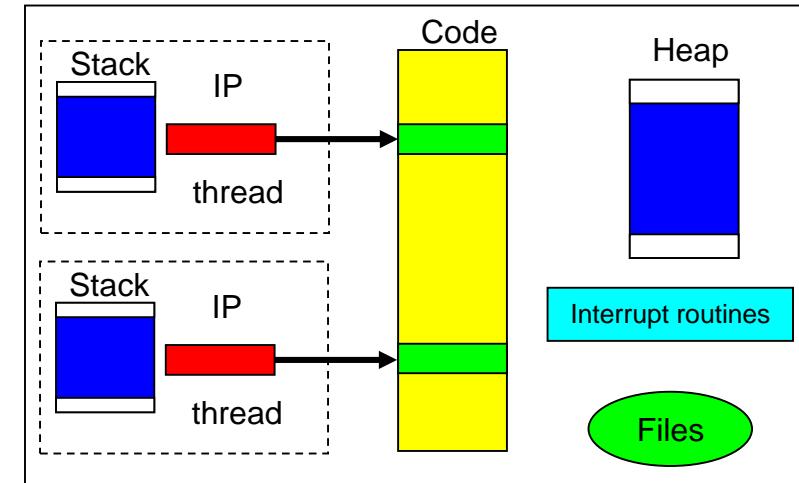
- **Text Segment:** Contains the program's compiled instructions (read-only code).
- **Data Segment:** Stores global and static variables.
- **Heap:** Holds dynamically allocated memory created at runtime. (Grows upwards)
- **Stack:** Manages function calls, local variables, and return addresses (Grows downward)

Differences between a process and threads

process - completely separate program with its own variables, stack, and memory allocation.



Threads - shares the same memory space and global variables between routines.



“lightweight” process is a kernel or some O.S. thread

Shared Memory Systems and Programming

Topics:

- Regular shared memory systems and programming
- Distributed shared memory on a cluster

Pthreads

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

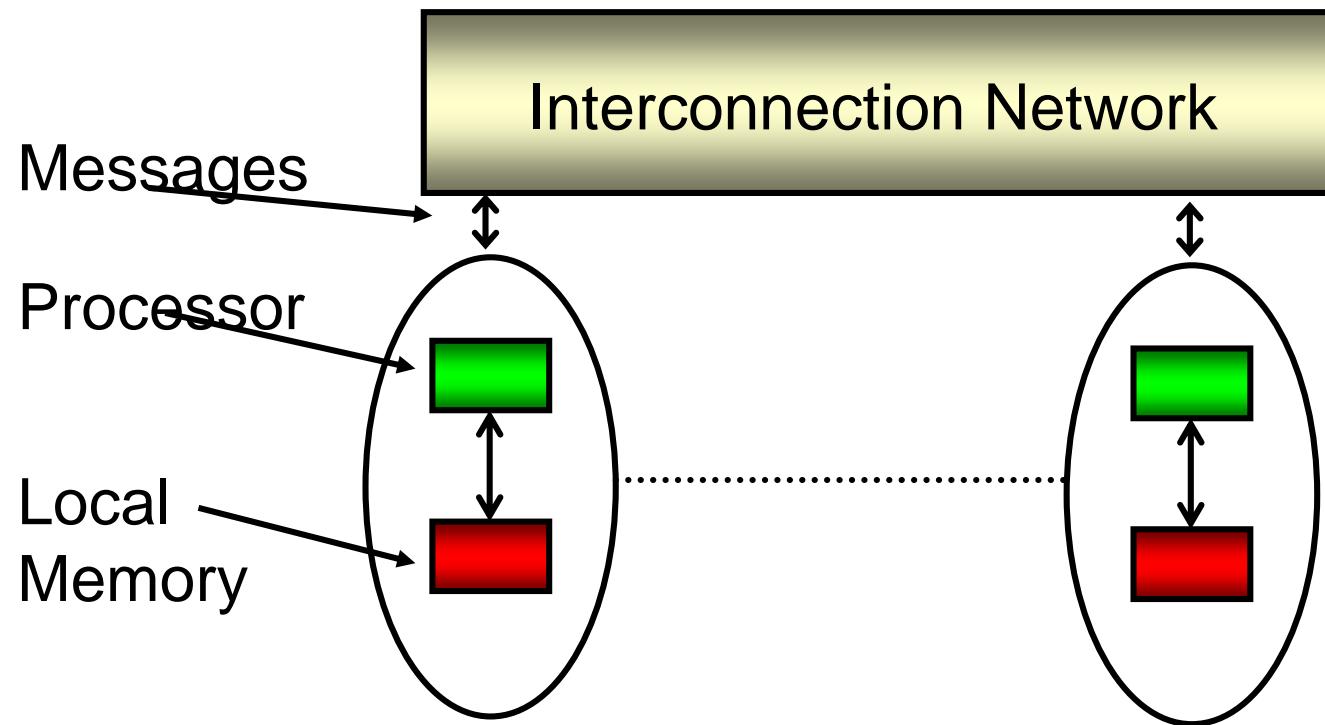
In Pthread, the main program is a thread itself, a separate
thread is created and terminated with the routine itself.

```
pthread_t thread1; /*handle of special Pthread data type*/  
pthread_create(&thread1, NULL, (void *)proc1, (void *)&argv);  
pthread_join(thread1, void *status);
```

A thread ID or handle is assigned and obtained from &thread

Message-Passing Multicomputer

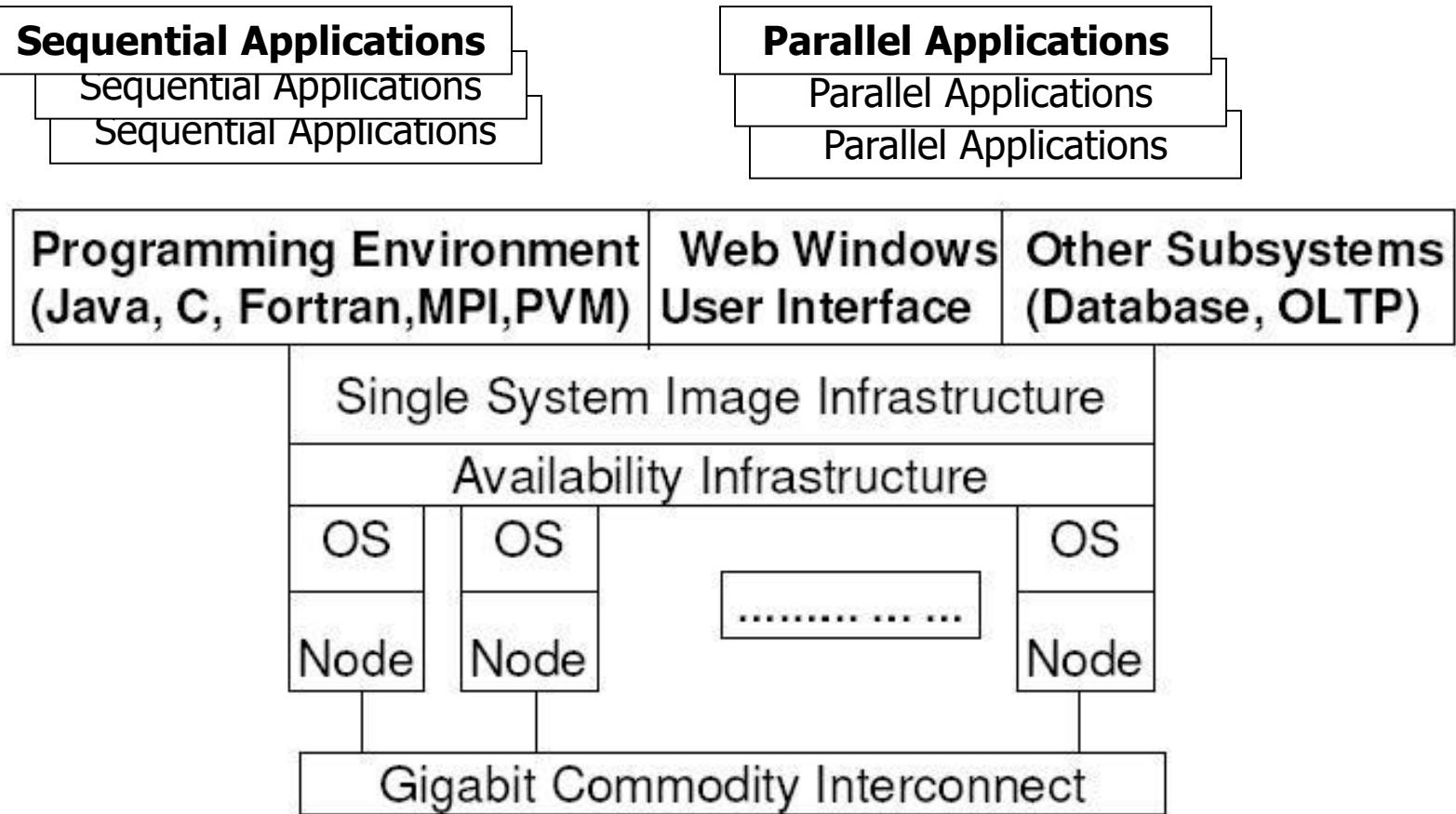
MIMD Complete computers connected through an interconnection network:

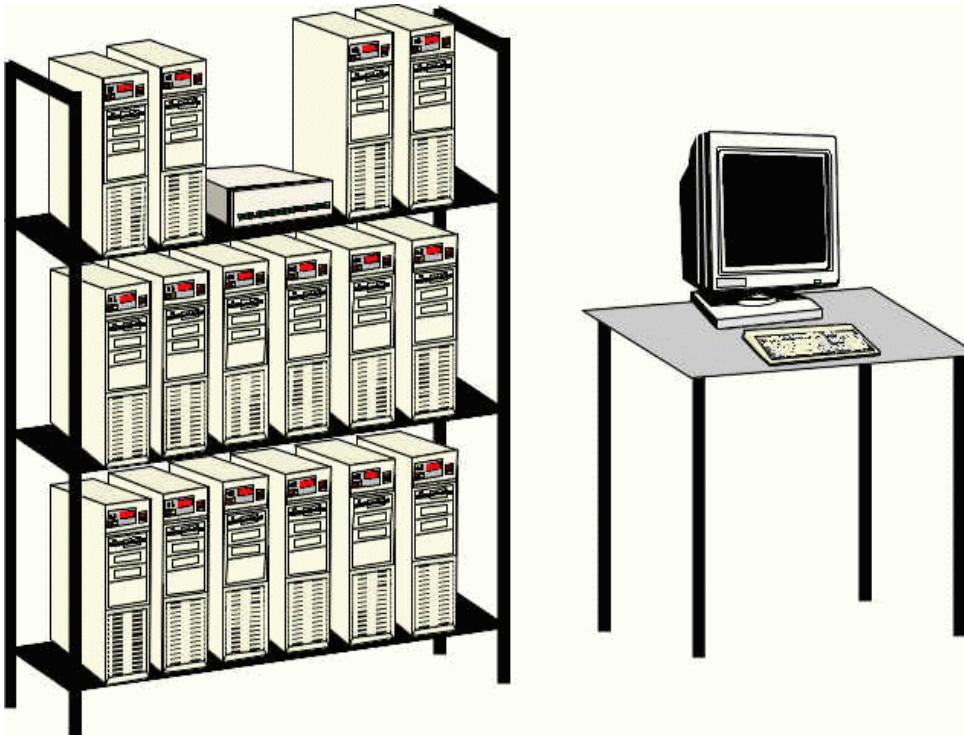


Cluster of Computers – Features

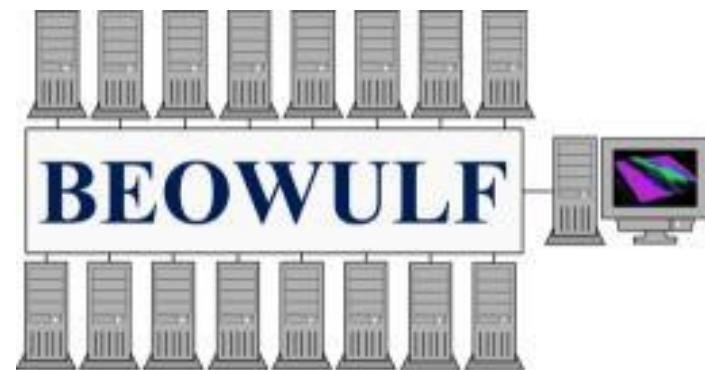
- A **cluster** is a type of parallel or distributed processing system, which consists of a collection of interconnected ***stand-alone computers*** cooperatively working together as a single, integrated computing resources. “***stand-alone***” (whole) computer that can be used on its own (full hardware and OS)
- Collection of nodes physically connected over commodity/ proprietary network
- Cluster computer is a collection of complete independent workstations or Symmetric Multi Processors
- Network is a decisive factors for scalability issues (especially for fine grain applications)
- High volumes driving high performance
- Network using commodity components and proprietary architecture is becoming the trend

Cluster system architecture





**120 nodes Jupiter in
CMSD Lab**



DIGITAL TEMP INDICATOR
295 °

DIGITAL TEMP INDICATOR
148 °
DS17500 Equipment



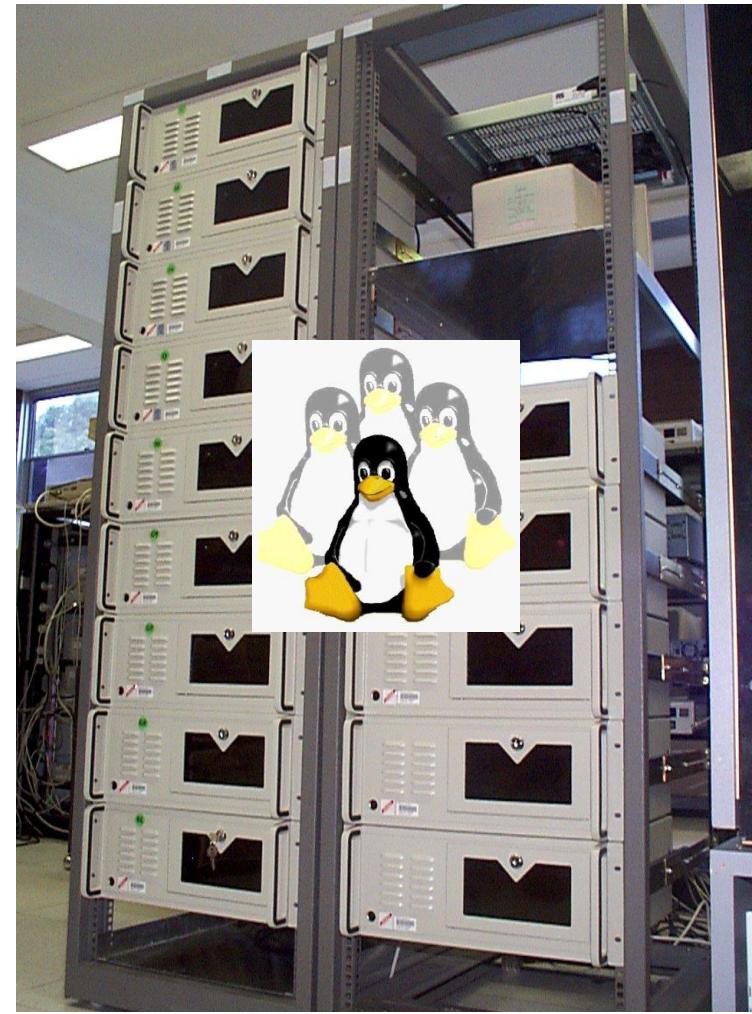
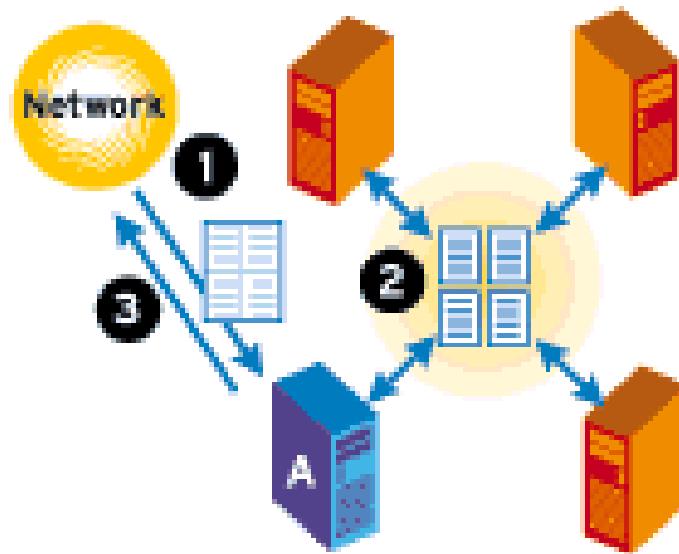


1680 CPU cores, achieving peak **120-Tflops** speed, fully networked and consists of the following hardware: HPE Master Nodes(2), Computing Nodes(42), Storage Nodes(2) and SAN storage.

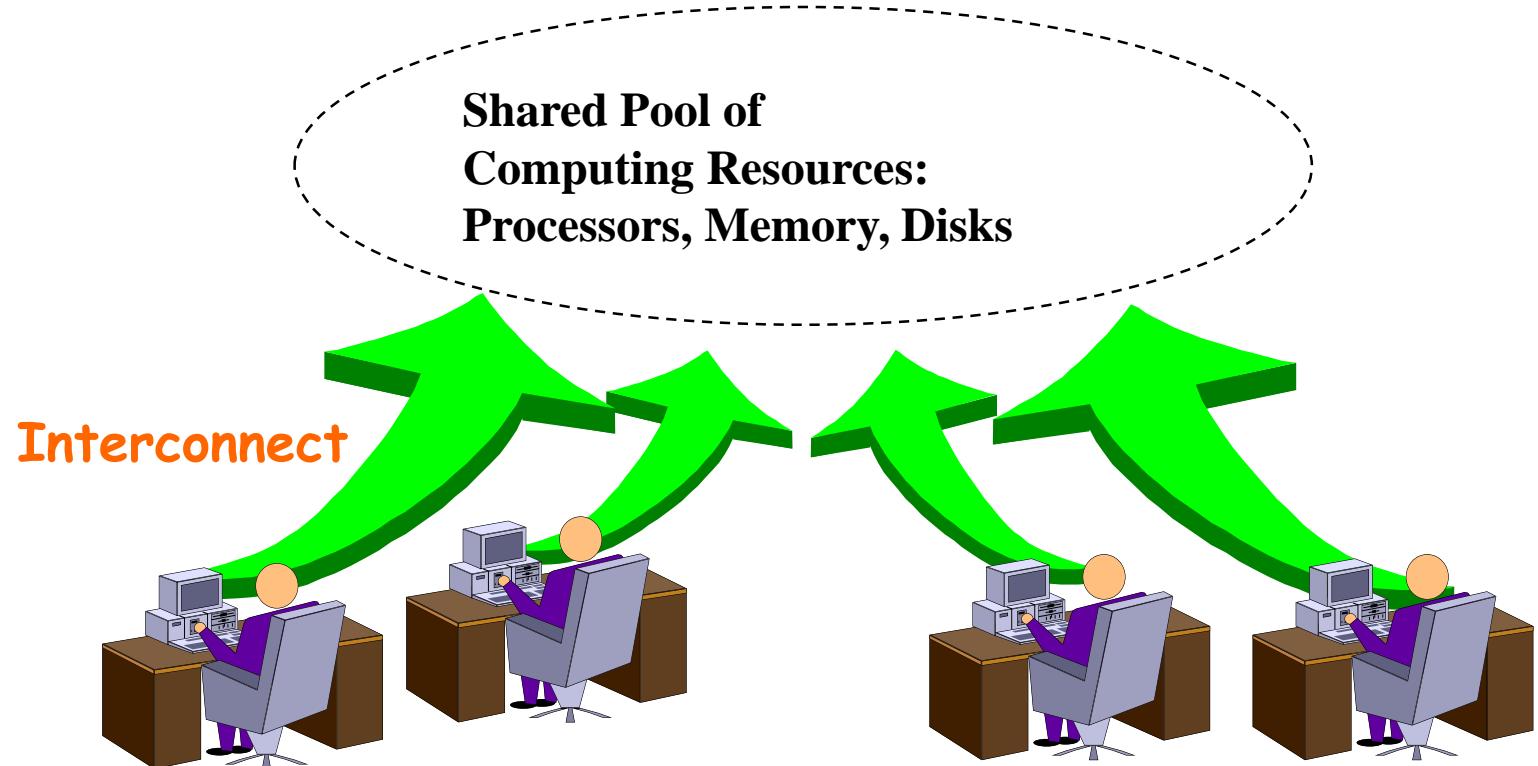
Common Cluster Modes

- **High Performance (dedicated)**
- **High Throughput (idle cycle collection)**
- **High Availability**

High Performance Cluster (dedicated mode)



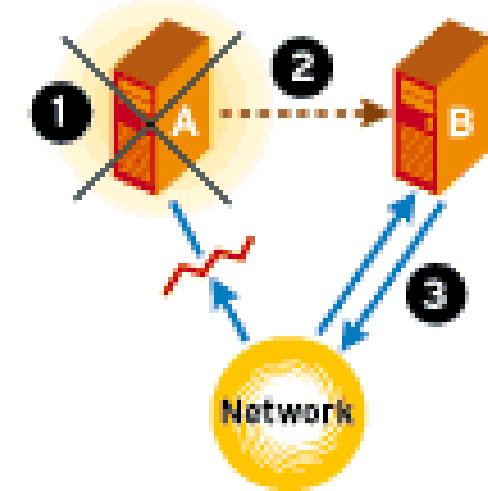
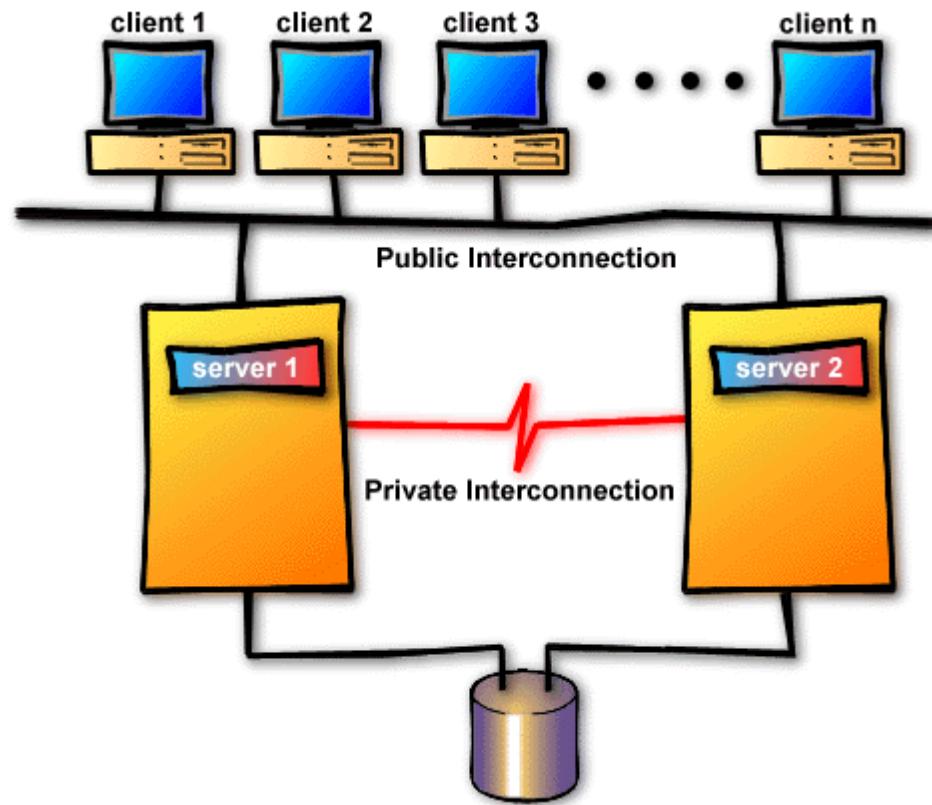
High Throughput Cluster (Idle Resource Collection)



Guarantee at least one workstation to many individuals (when active)

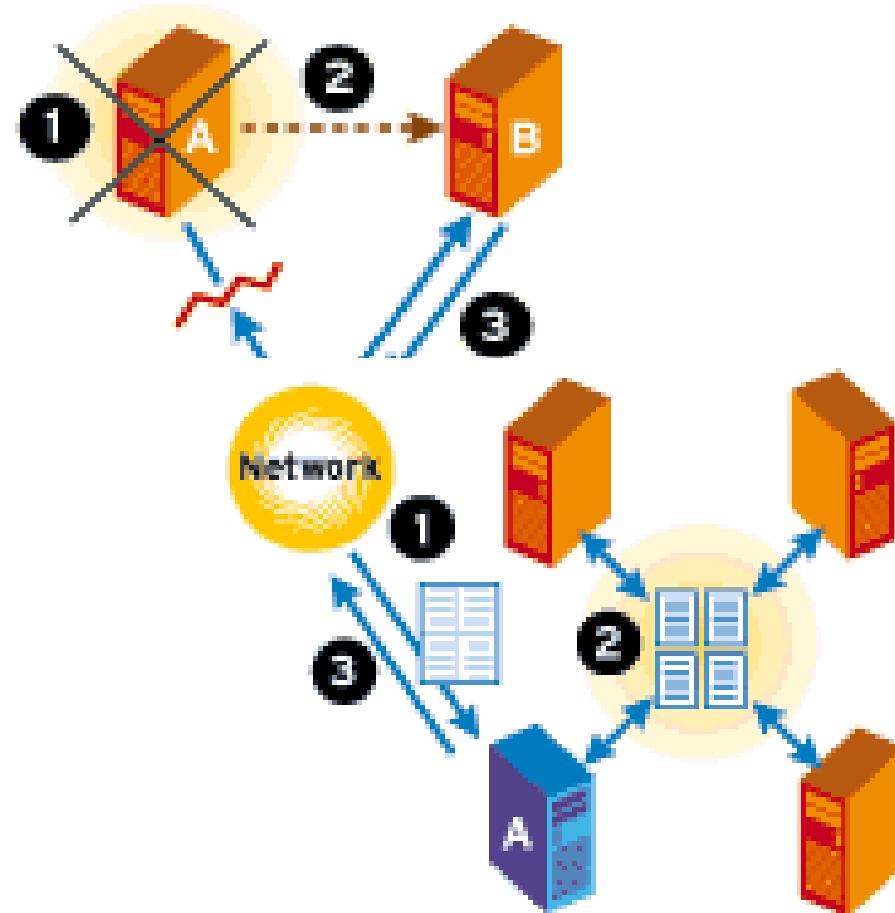
Deliver large % of collective resources to few individuals at any one time

High Availability Clusters



HA and HP in the same Cluster

- Best of both Worlds:
world is heading towards
this configuration)



HPC(Compute) Cluster Components

Prominent Components of Cluster Computers

- Multiple High Performance Computers
 - PCs
 - Workstations
 - SMPs (CLUMPS)
 - Distributed HPC Systems leading to Cloud Computing

Prominent Components of Cluster Computers

- State of the art Operating Systems
 - Linux (MOSIX, Beowulf, and many more)
 - Microsoft NT (Illinois HPVM, Cornell Velocity)
 - SUN Solaris (Berkeley NOW, C-DAC PARAM)
 - IBM AIX (IBM SP2)
 - HP UX (Illinois - PANDA)
 - Mach (Microkernel based OS) (CMU)
 - Cluster Operating Systems (Solaris MC, SCO Unixware, MOSIX (academic project))
 - OS gluing layers (Berkeley Glunix)

Prominent Components of Cluster Computers

- High Performance Networks/Switches
 - Ethernet (10Mbps),
 - Fast Ethernet (100Mbps),
 - Gigabit Ethernet (1Gbps)
 - SCI (Scalable Coherent Interface- MPI- 12 μ sec latency)
 - ATM (Asynchronous Transfer Mode)
 - Myrinet (1.2Gbps)
 - QsNet (Quadrics Supercomputing World, 5 μ sec latency for MPI messages)
 - Digital Memory Channel
 - FDDI (Fiber Distributed Data Interface)
 - InfiniBand

Prominent Components of Cluster Computers

- Fast Communication Protocols and Services (User Level Communication):
 - Active Messages (Berkeley)
 - Fast Messages (Illinois)
 - U-net (Cornell)
 - XTP (Virginia)
 - Virtual Interface Architecture (VIA)

Prominent Components of Cluster Computers

- Parallel Programming Environments and Tools
 - MPI (Message Passing Interface), standard
 - Linux, NT, on many Supercomputers
 - Implementations MPICH, Open MPI
 - PVM (Parallel Virtual Machine)
 - Software DSMs (Shmem)
 - Threads (PCs, SMPs, NOW..)
 - POSIX Threads
 - Java Threads
 - Compilers
 - C/C++/Java/FORTRAN
 - RAD (rapid application development tools)
 - GUI based tools for PP modeling
 - Debuggers
 - Performance Analysis Tools (PAPI, Performance Application Programming Interface)
 - Visualization Tools

Classification of Clusters

- ❖ Packaging
 - Compact/Slack
- ❖ Control
 - Centralized/Decentralized
- ❖ Homogeneity
 - Homogenous/Heterogeneous
- ❖ Security
 - Enclosed/Exposed

Classification of Clusters

- ❖ Packaging
 - Compact/Slack
- ❖ Control
 - Centralized/Decentralized
- ❖ Homogeneity
 - Homogenous/Heterogeneous
- ❖ Security
 - Enclosed/Exposed

Classify using four orthogonal attributes packaging, control, homogeneity and security

Dedicated v/s Enterprise Clusters

- Dedicated
 - Compact, Centralized, Homogenous, Enclosed
- Enterprise
 - Geographically distributed (slack), Decentralized, heterogeneous, Exposed

Users View of Cluster

The users view the entire cluster as **Single system**, which has multiple processors. The user could say: “Execute my application using five processors.” This is different from a distributed system.

- Single Entry
- Single File Hierarchy
- Single Networking
- Single Input/Output
- Single Point of Control
- Single Memory Space
- Single Job Management System
- Single User Interface
- Single Process Space
- Single System
- Symmetry
- Location Transparent

Job Management

- ❖ Global job management
- ❖ Global system management and configuration
- ❖ Group based scheduling and resource allocation
- ❖ Idle resource detection
- ❖ Co-scheduling of parallel programs
- ❖ Load Sharing Facility (LSF)

High Availability

- ❖ Fault tolerance
- ❖ Check-pointing

Some Cluster Systems: Comparison

Project	Platform	Communications	OS	Other
Beowulf	PCs	Multiple Ethernet with TCP/IP	Linux and Grendel	MPI/PVM/PGAS Sockets and HPF
Berkeley Now	Solaris-based PCs and workstations	Myrinet and Active Messages	Solaris + GLUnix + xFS	AM, PVM, MPI, HPF, Split-C
HPVM	PCs	Myrinet with Fast Messages	NT or Linux connection and global resource manager + LSF	Java-fronted, FM, Sockets, Global Arrays, SHMEM and MPI
Solaris MC	Solaris-based PCs and workstations	Solaris-supported	Solaris + Globalization layer	C++ and CORBA

Compare HP Cluster and Data Cluster

System Components	HPC Cluster	Data Cluster
Standards/APIs/Model	MPI/PVM/Partitioned Global Address Space(PGAS)*	Hadoop/Spark
Programming Languages	C/C++/FORTRAN/Python	Java/Python/R/Scala
Scheduler/Resource Managers	PBS/SLURM/LSF/TORQUE	YARN/Apache Mesos/Nirmod
File Systems	General Parallel File System (GPFS)/NFS/Luster/GFS	HDFS/Databricks/Google BigQuery/Cloudera/Amazon EMR/Azure HDINSIDE....

*Shared memory space may have an affinity for a particular process, thereby exploiting locality of reference (tendency of a processor (or process) to access the same set of memory locations repetitively over a short period of time)

Parallel & Distributed File Systems

- A parallel file system is a type of distributed file system.
- Both distributed and parallel file systems can spread data across multiple storage servers
- Scale to accommodate petabytes of data, and support high bandwidth
- They only differ in their objectives so mechanism to access differs

Parallel & Distributed File Systems

Parallel File Systems	Distributed File Systems
Support a shared global namespace	Support a shared global namespace
The client systems have direct access to all of the storage nodes for data transfer without having to go through a single coordinating server	All client systems accessing a given portion of the namespace generally go through the same storage node to access the data and metadata
The paradigm in HPC is to separate compute from storage	They do not separate

Parallel & Distributed File Systems

Parallel File Systems	Distributed File Systems
Distribute data across multiple nodes by breaking up the file and stripes the data blocks across (mostly round robin)	Store entire object on a single node
It reads and writes data to distributed storage devices using multiple I/O paths concurrently, to provide a significant performance benefit	achieve its performance through its clever insight of shipping code to data

Parallel & Distributed File Systems

Parallel File Systems	Distributed File Systems
Requires the installation of client-based software drivers to access the <u>shared storage</u> via high-speed networks such as Ethernet, InfiniBand, and OmniPath.	Uses a standard network file access protocol, such as NFS or SMB (Server Message Block), to access a storage server
It focuses on high-performance workloads that can benefit from coordinated I/O access and significant bandwidth	It targets loosely coupled, data-heavy applications
Run on shared storage.	three-way replication to provide <u>fault tolerance</u>

Major References

Kai Hwang, Zhiwei Xu, Scalable Parallel Computing (Technology Architecture Programming) McGraw Hill Newyork (1997).

Culler David E, Jaswinder Pal Singh with Anoop Gupta, Parallel Computer Architecture, A Hardware/Software Approach, Morgan Kaufmann Publishers, Inc, (1999), Reprinted in 2004.

Barry Wilkinson And Michael Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Prentice Hall, Upper Saddle River, NJ, 1999.

RajKumar Buyya, High Performance Cluster Computing, Programming and Applications, Prentice Hall, 1999.

Parallel Algorithm Design

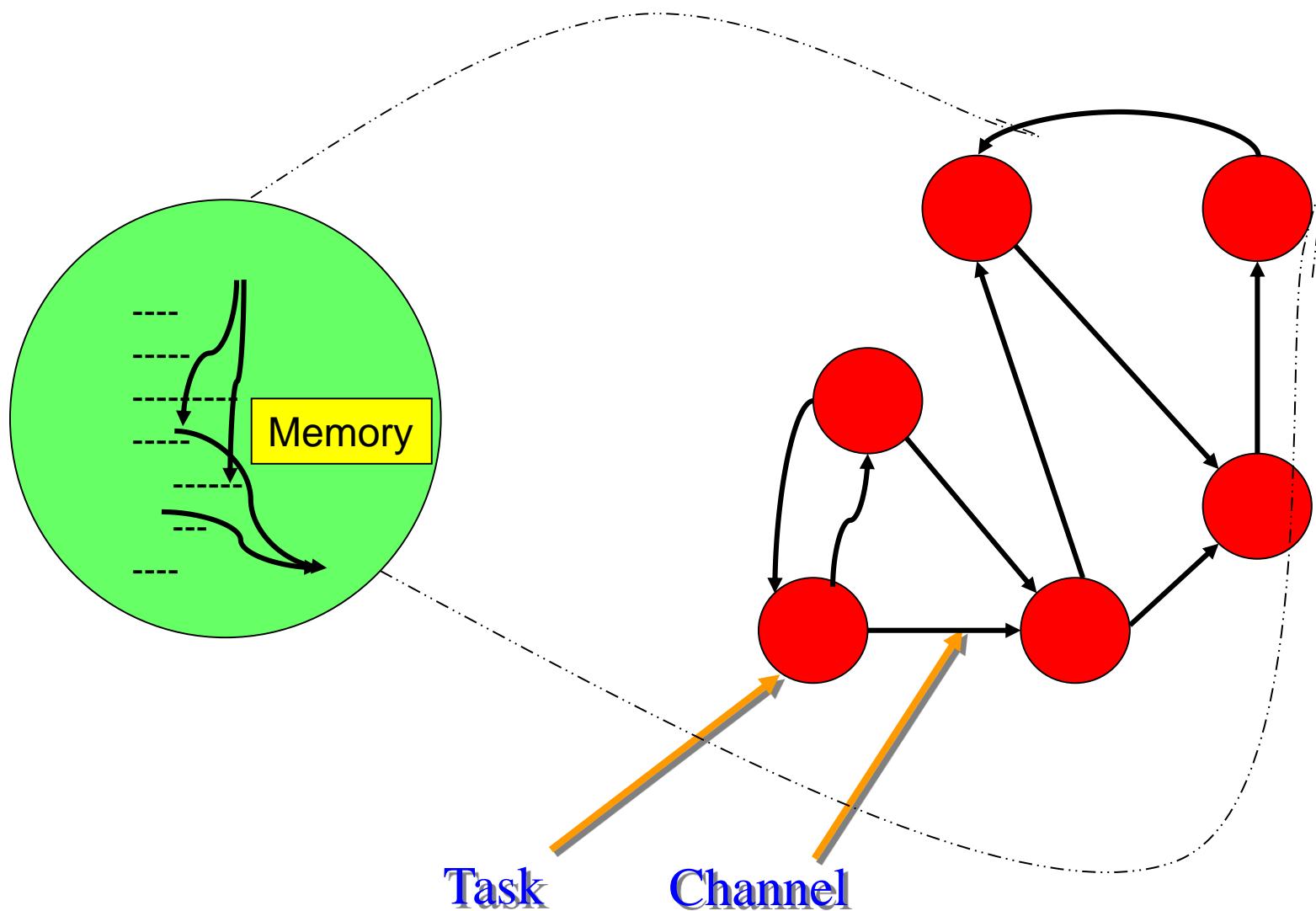
Outline

- Task/channel model
- Algorithm design methodology
- Case studies

Task/Channel Model

- Parallel computation = set of tasks
- Task
 - Program
 - Local memory
 - Collection of I/O ports
- Tasks interact by sending messages through channels
- At input port task must wait until the value appears, means task is **blocked**
- In this model receiving is a **synchronous**, while sending is an **asynchronous** operation

Task/Channel Model



Foster's Design Methodology*

- Partitioning
- Communication (Concentration on inherent parallelism)
- Agglomeration
- Mapping (Concentration on implementation on real HW)

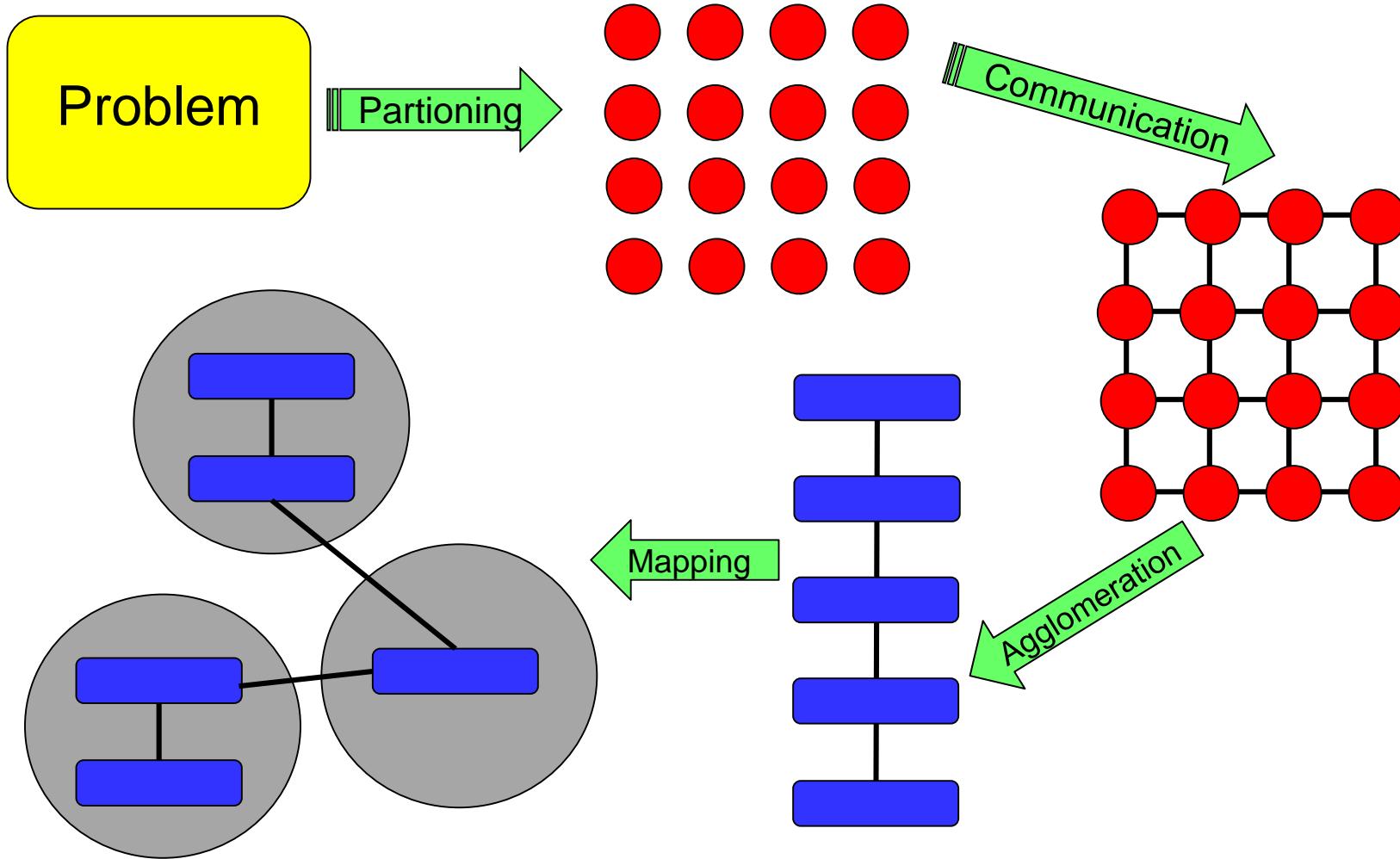
“It delays the machine dependent considerations at the later stage “

* Foster, Ian. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Reading, MA: Addison-Wesley, 1995

Foster's Design Methodology

A well-known design methodology that is used to architect and implement distributed-memory systems (DMS) particularly on Multi computer systems.

Foster's Methodology



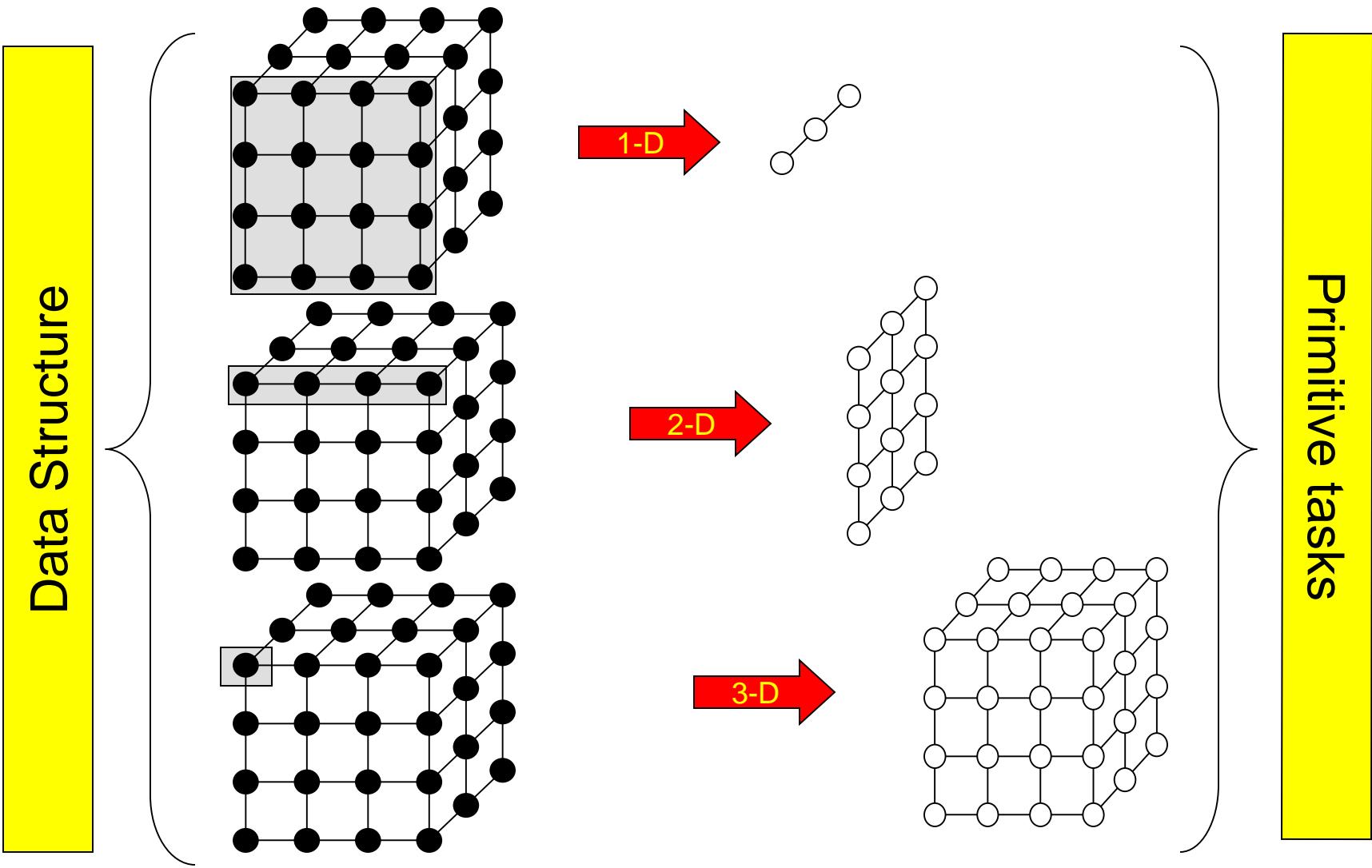
1. Partitioning

- Dividing computation and data into pieces
- Domain decomposition
 - Divide data into pieces
 - Determine how to associate computations with the data
- Functional decomposition
 - Divide computation into pieces
 - Determine how to associate data with the computations

1. Partitioning

- Focus is on most frequently accessed data structure
- In the matrix, we can partition data into
 - Collection of 2-D slice: resulting in a 1-D collection of primitive tasks
 - Collection of 1-D slice: resulting in a 2-D collection of primitive tasks
 - Consider each elements of the matrix individually: 3-D collection of primitive tasks

Domain Decompositions



1. Partitioning Checklist

- At least 10x more primitive tasks than processors in target computer
- Minimize redundant computations and redundant data storage (*If not, design does not work well when problem size increases*)
- Primitive tasks roughly the same size (*If not, load balancing problem*)
- Number of tasks an increasing function of problem size (*If not, not scale well*)

2.Communication

- Determine values passed among tasks
- Local communication
 - A task needs values from a small number of other tasks
 - Create channels illustrating data flow
- Global communication
 - Significant number of tasks contribute data to perform a computation
 - Don't create channels for them early in design

2. Communication Checklist

Communications are overhead in parallel algorithms,
minimizing them is an important goal

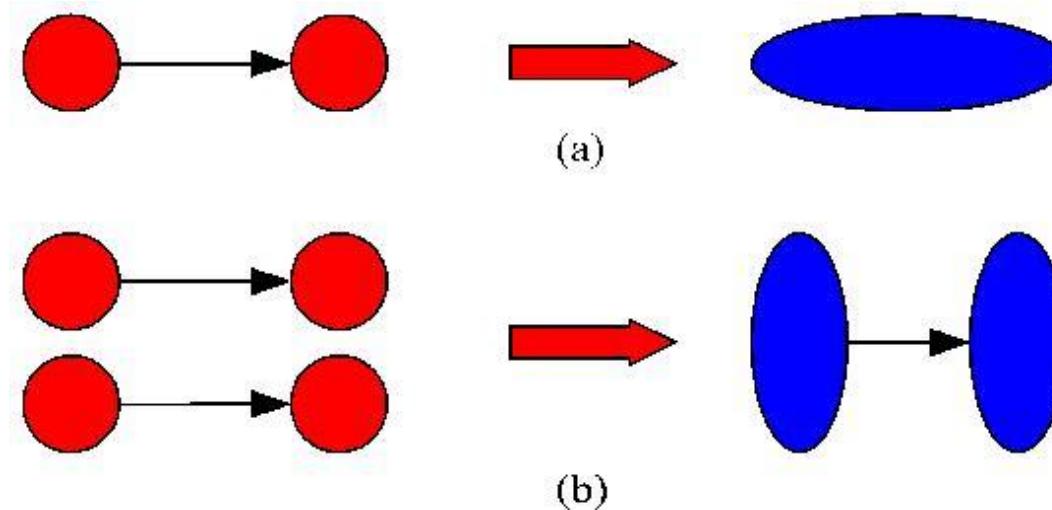
- Communication operations balanced among tasks
- Each task communicates with only small group of neighbors
- Tasks can perform communications concurrently
- Task can perform computations concurrently

3. Agglomeration

- Grouping tasks into larger tasks
- Goals
 - Improve performance
 - Maintain scalability of program
 - Simplify programming
- *In MPI programming, goal often is to create one agglomerated task per processor*

Agglomeration Can Improve Performance

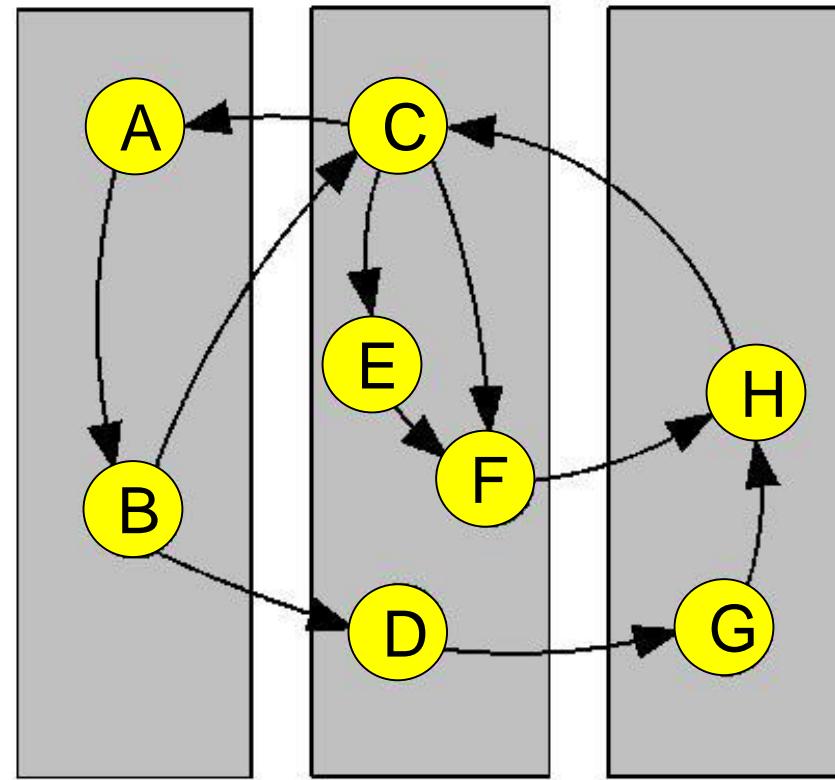
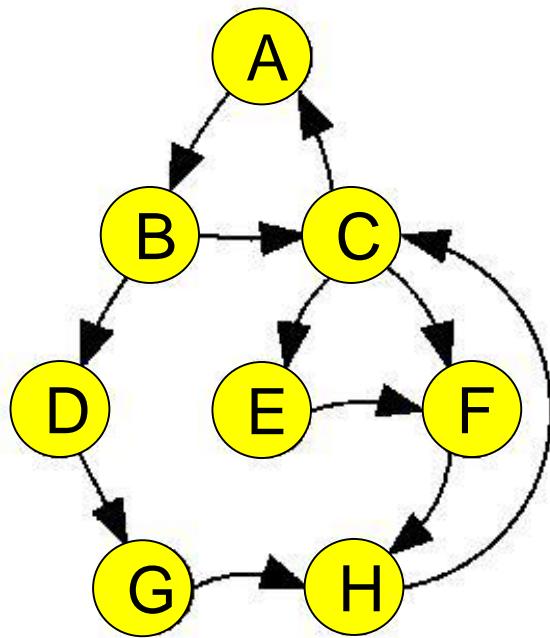
- Eliminate communication between primitive tasks agglomerated into consolidated task (specially when the tasks cannot perform their operations in parallel)
- Combine groups of sending and receiving tasks (for reducing message latency)



4.Mapping

- Process of assigning tasks to processors
- *Centralized multiprocessor: mapping done by operating system*
- *Distributed memory system: mapping done by user*
- Goals of mapping
 - Maximize processor utilization
 - Minimize inter-processor communication

Mapping Example



Optimal Mapping

- Finding optimal mapping is NP-hard
- Must rely on heuristics

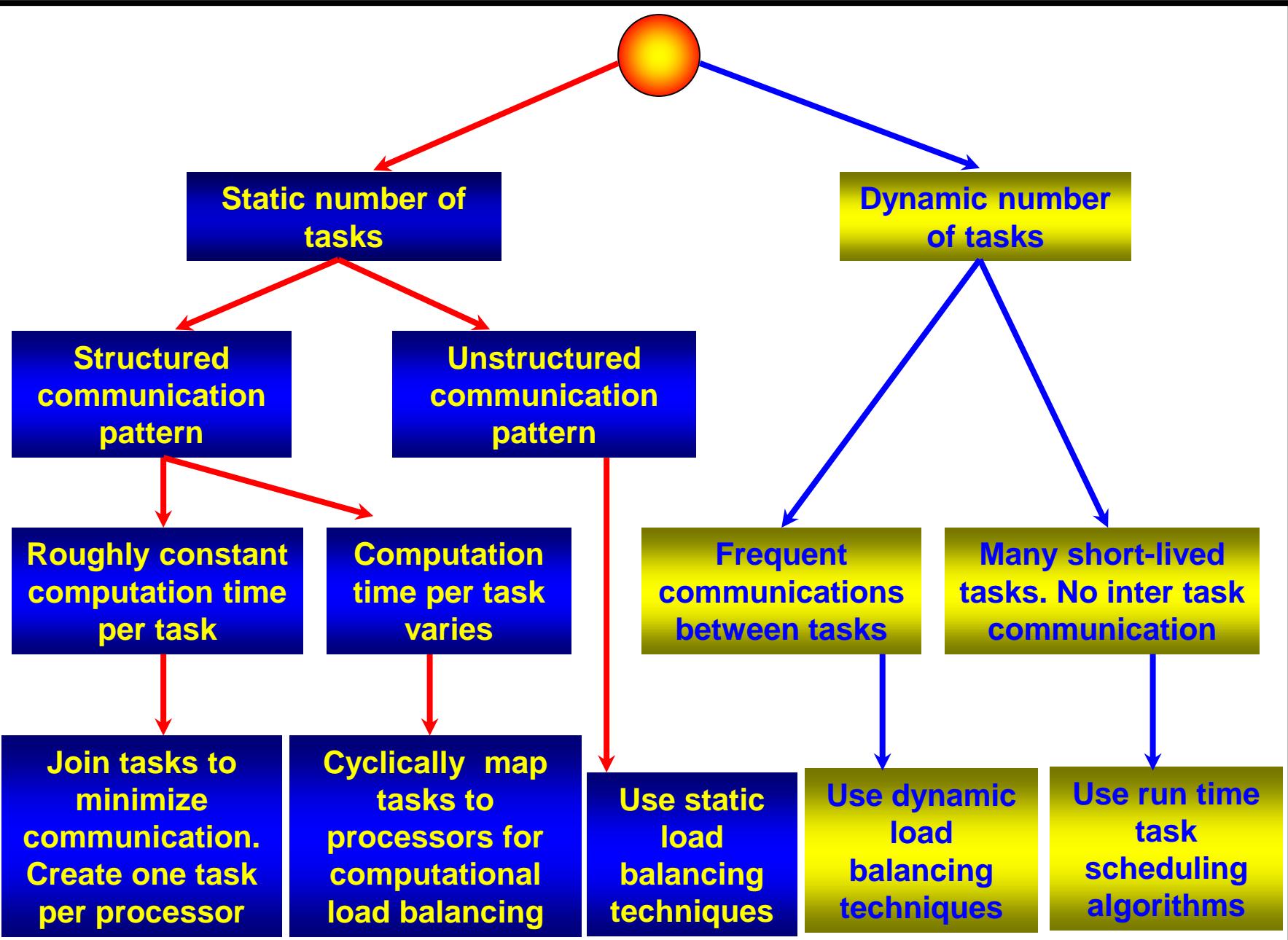
Definition

- In *local* communication, each task communicates with a small set of other tasks (its “neighbors”); in contrast, *global* communication requires each task to communicate with many tasks.
- In *structured* communication, a task and its neighbors form a regular structure, such as a tree or grid; in contrast, *unstructured* communication networks may be arbitrary graphs.

Definition

- In *static* communication, the identity of communication partners does not change over time; in contrast, the identity of communication partners in *dynamic* communication structures may be determined by data computed at runtime and may be highly variable.
- In *synchronous* communication, producers and consumers execute in a coordinated fashion, with producer/consumer pairs cooperating in data transfer operations; in contrast, *asynchronous* communication may require that a consumer obtain data without the cooperation of the producer.

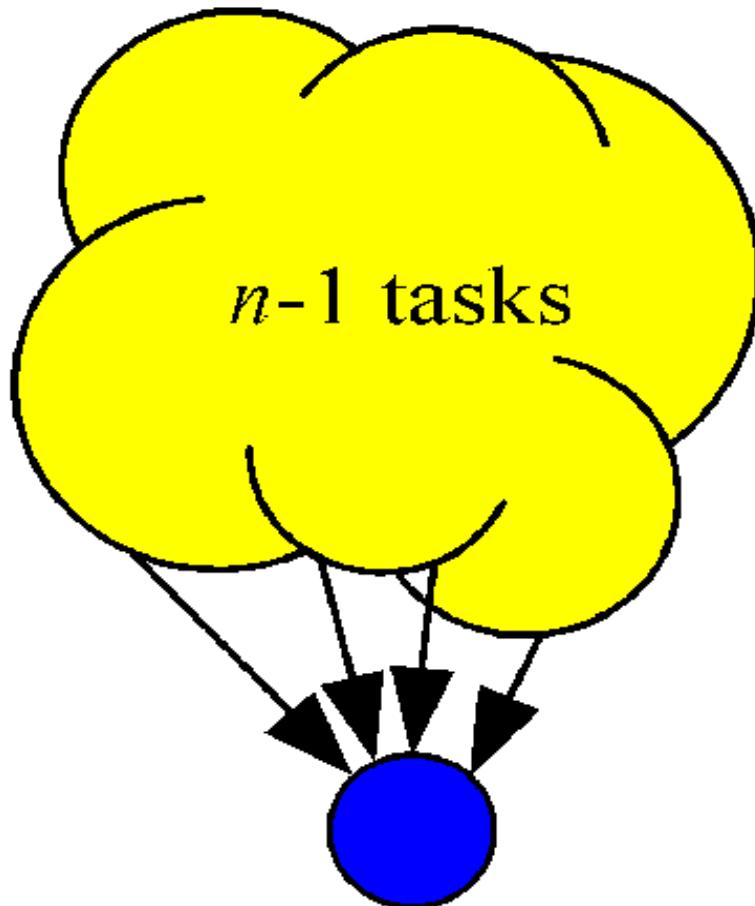
Decision tree to choose a mapping strategy



Reduction

- Given associative operator \oplus
- $a_0 \oplus a_1 \oplus a_2 \oplus \dots \oplus a_{n-1}$
- Examples
 - Add
 - Multiply
 - And, Or
 - Maximum, Minimum

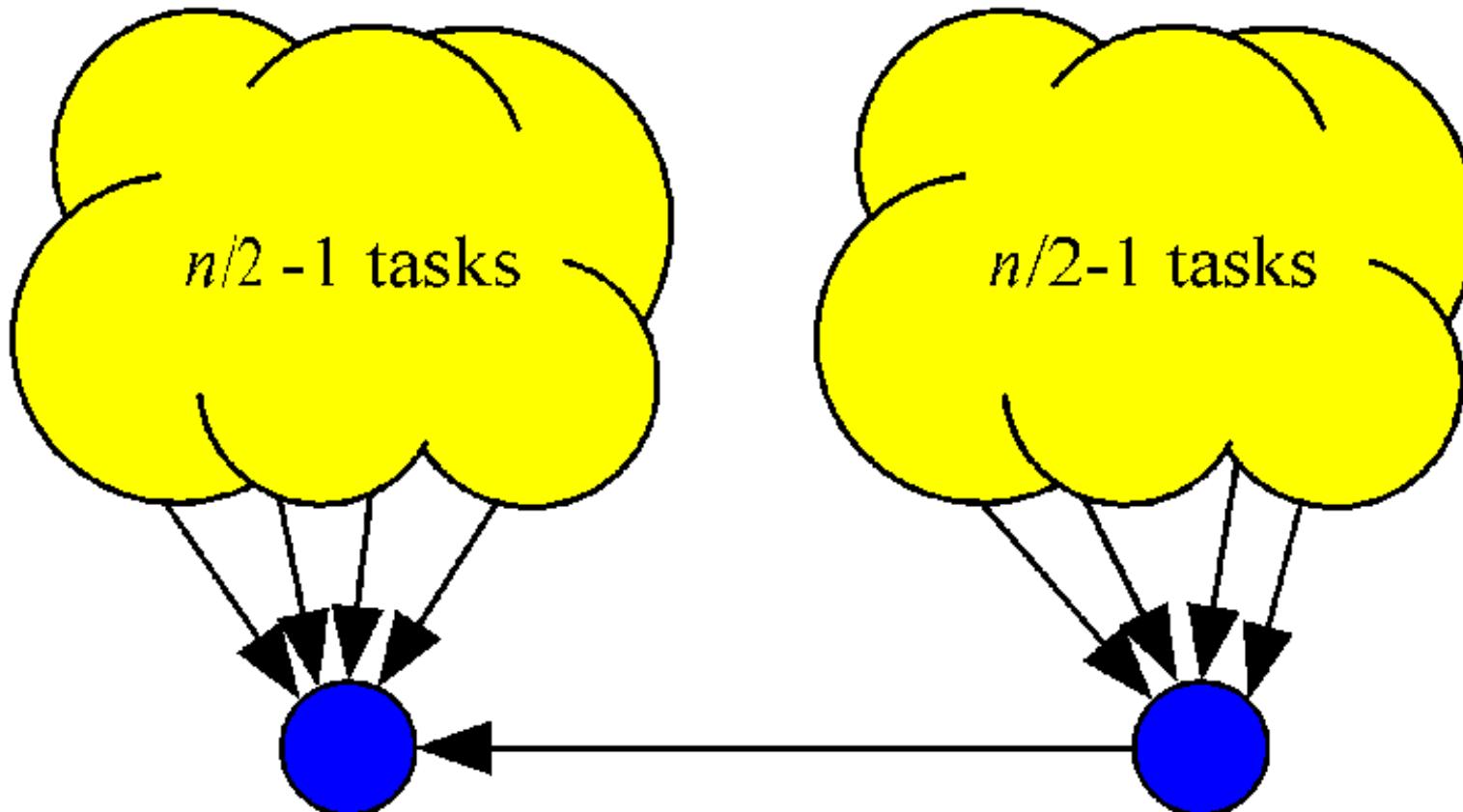
Parallel Reduction Evolution



Observations:

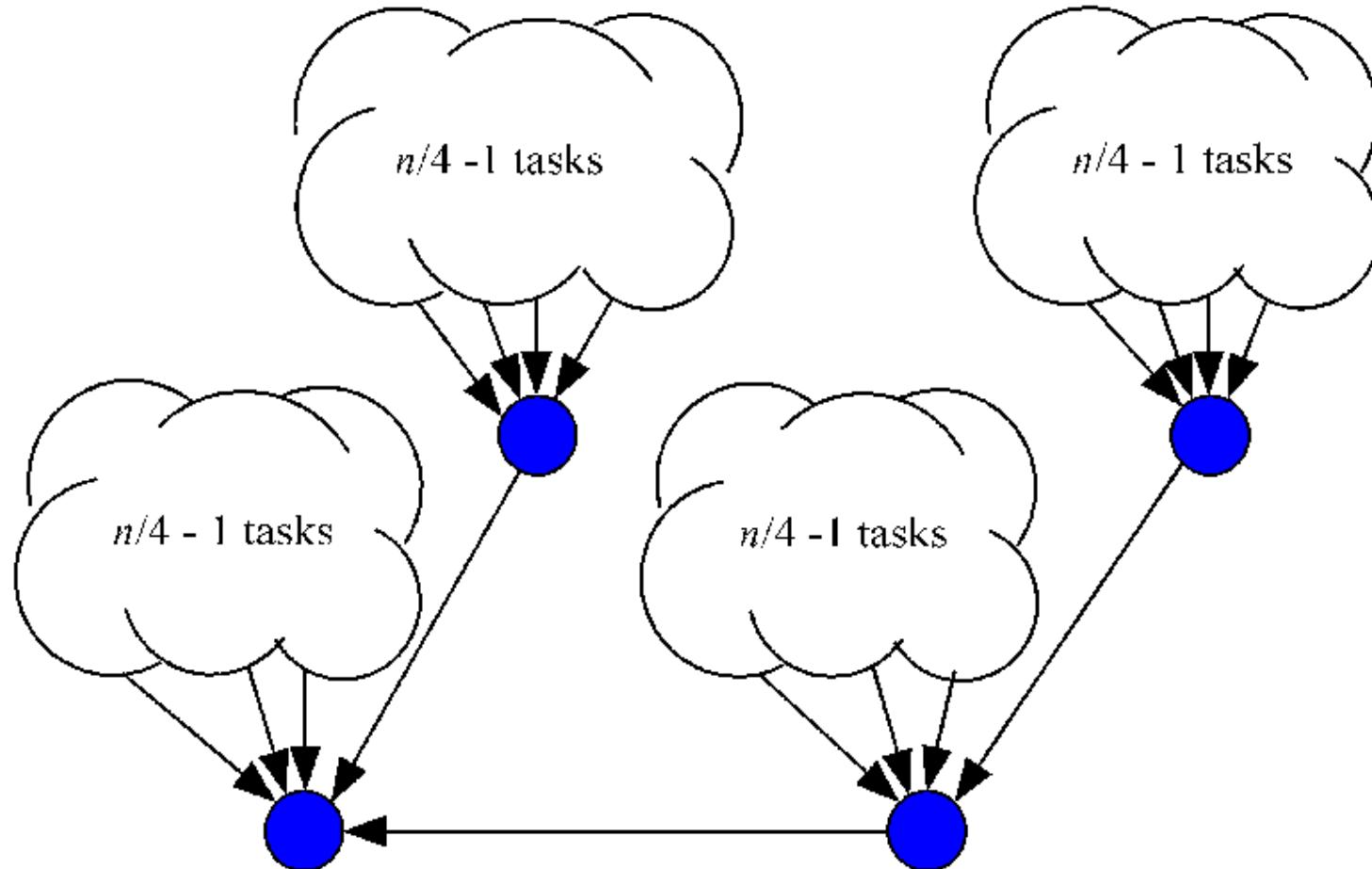
1. If x time required for a task to communicate another task
2. y time is required for addition then
3. Total time will be $(n-1)(x+y)$

Parallel Reduction Evolution

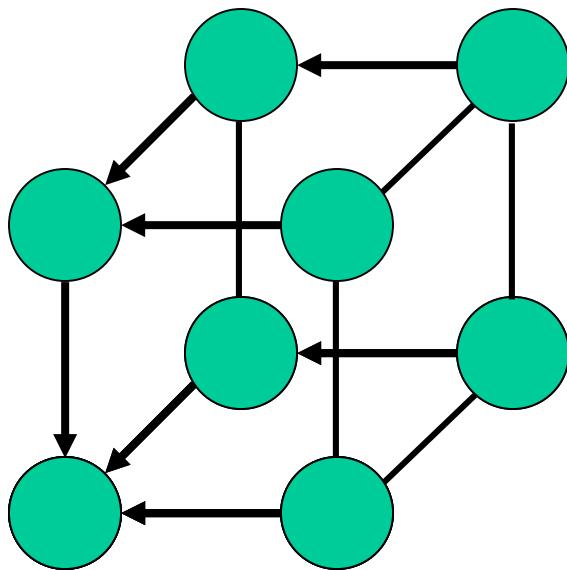


Observations: Total time will be now $(n/2-1)(x+y)$

Parallel Reduction Evolution



Binomial Trees



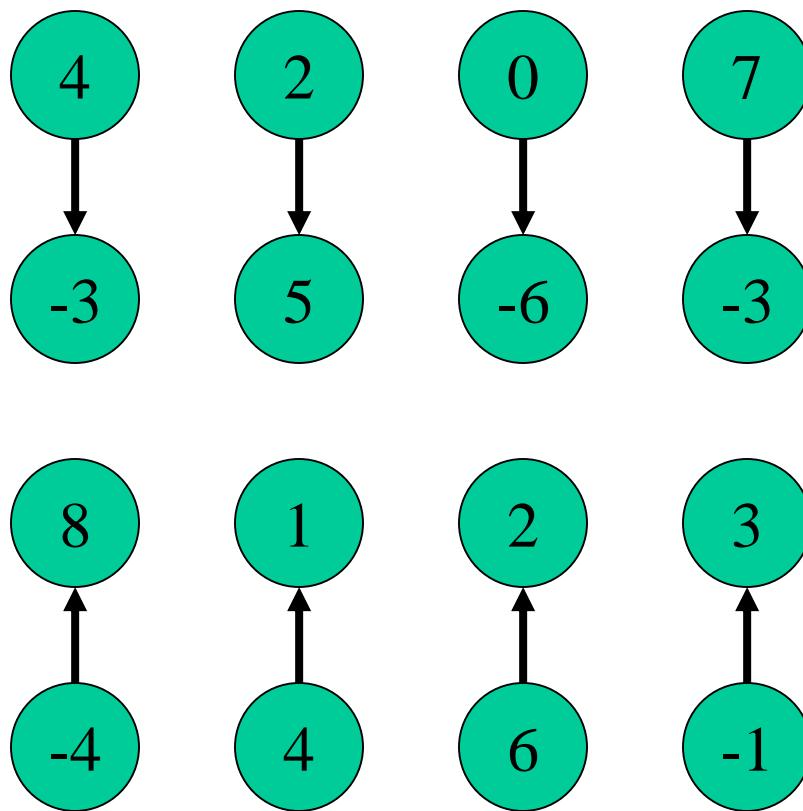
Subgraph of hypercube

Observations:

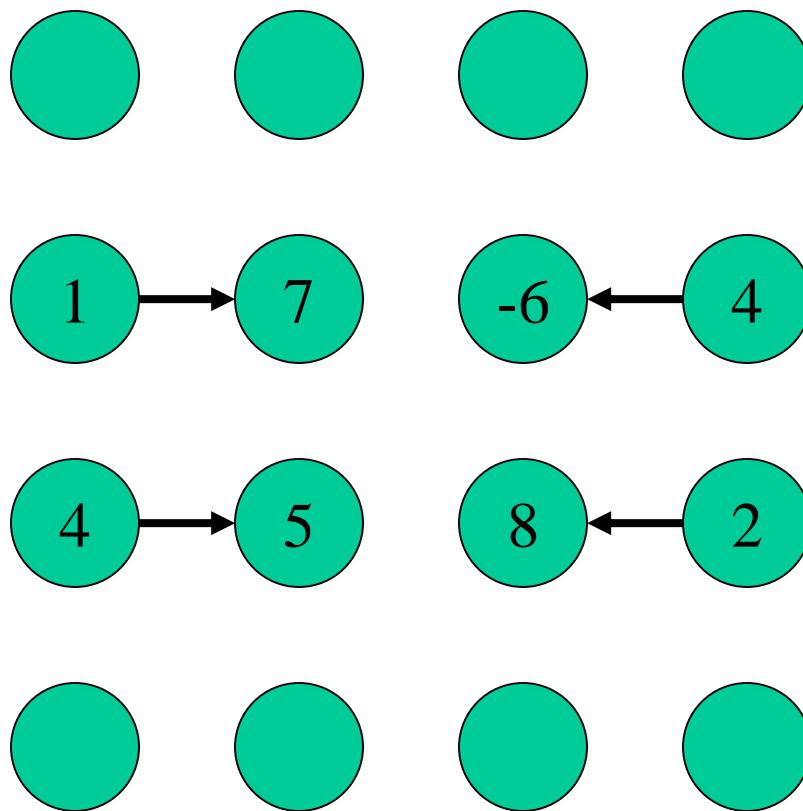
Continuing this way we have $n/2$ semi-root tasks

Total time will be **$\log n$**

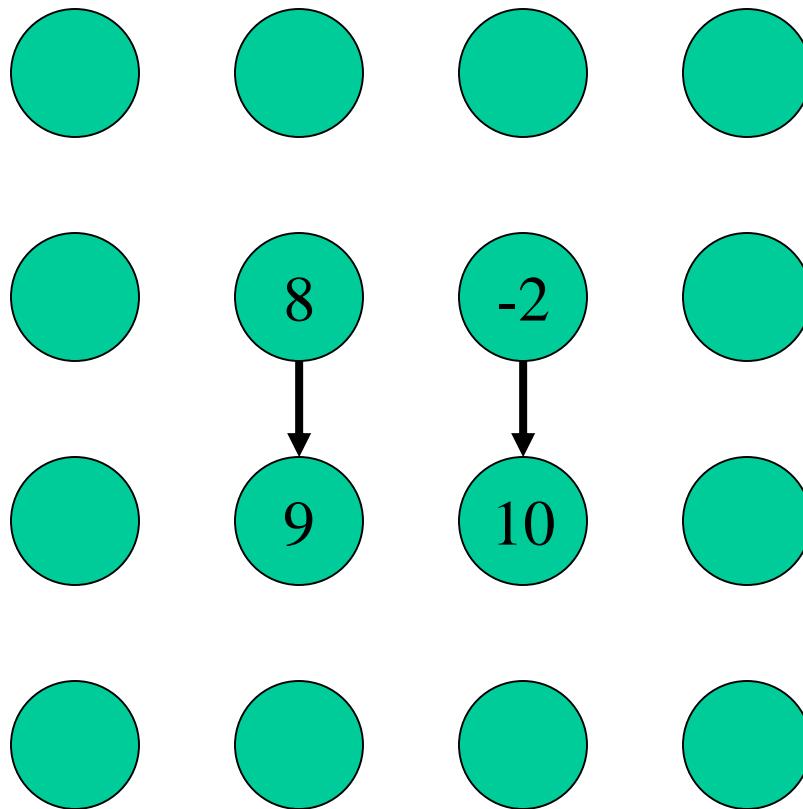
Finding Global Sum



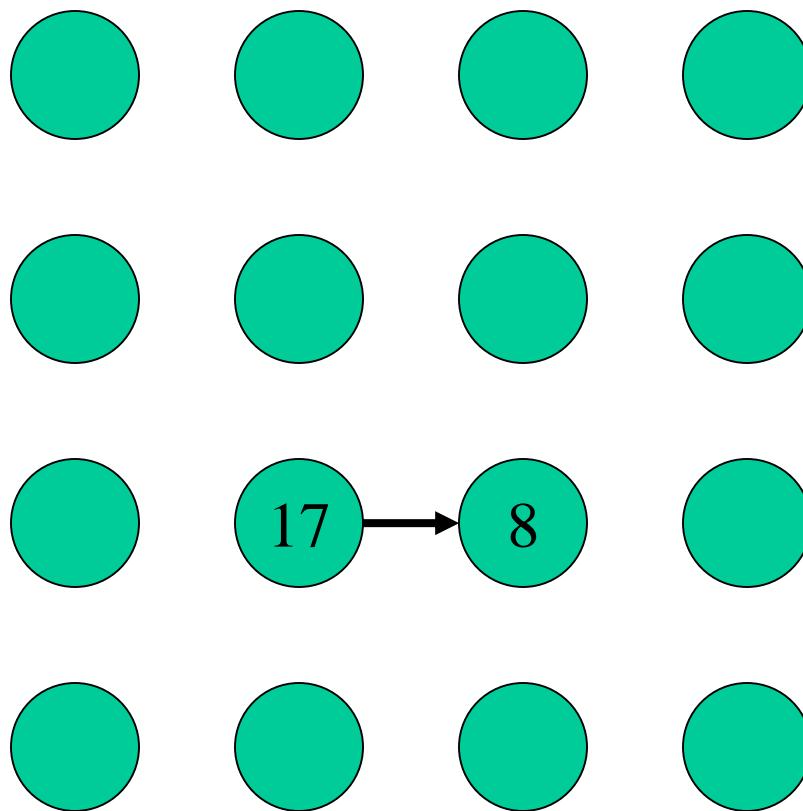
Finding Global Sum



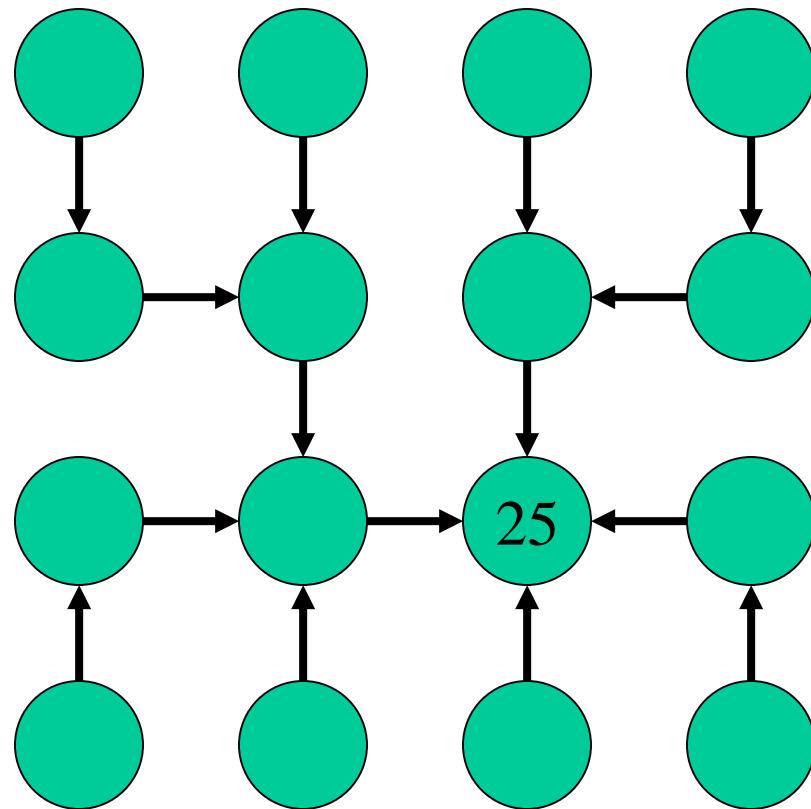
Finding Global Sum



Finding Global Sum

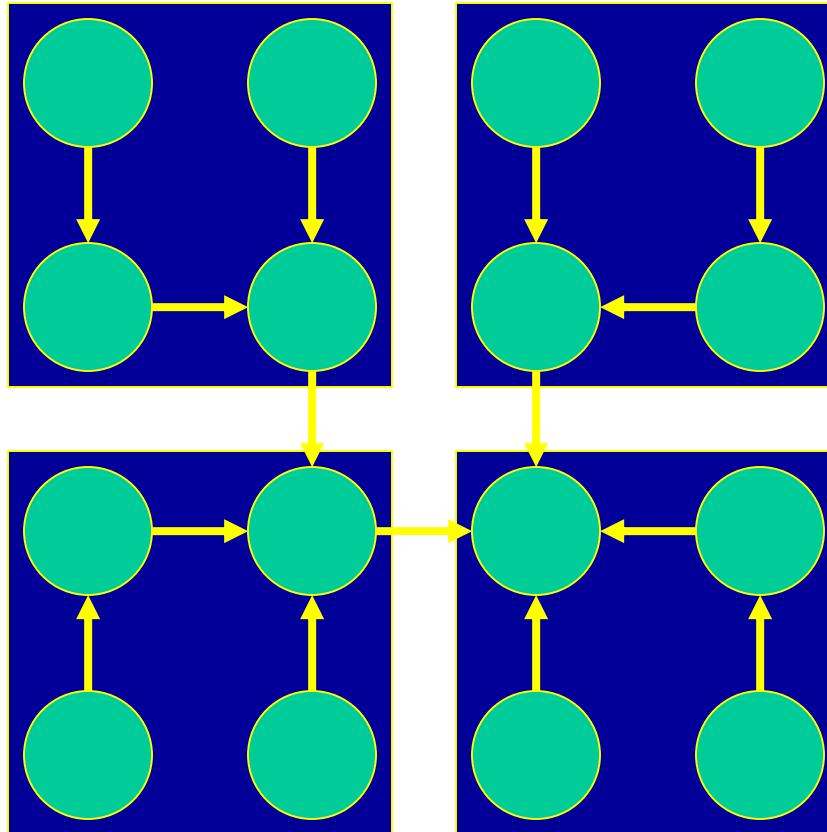


Finding Global Sum



Binomial Tree

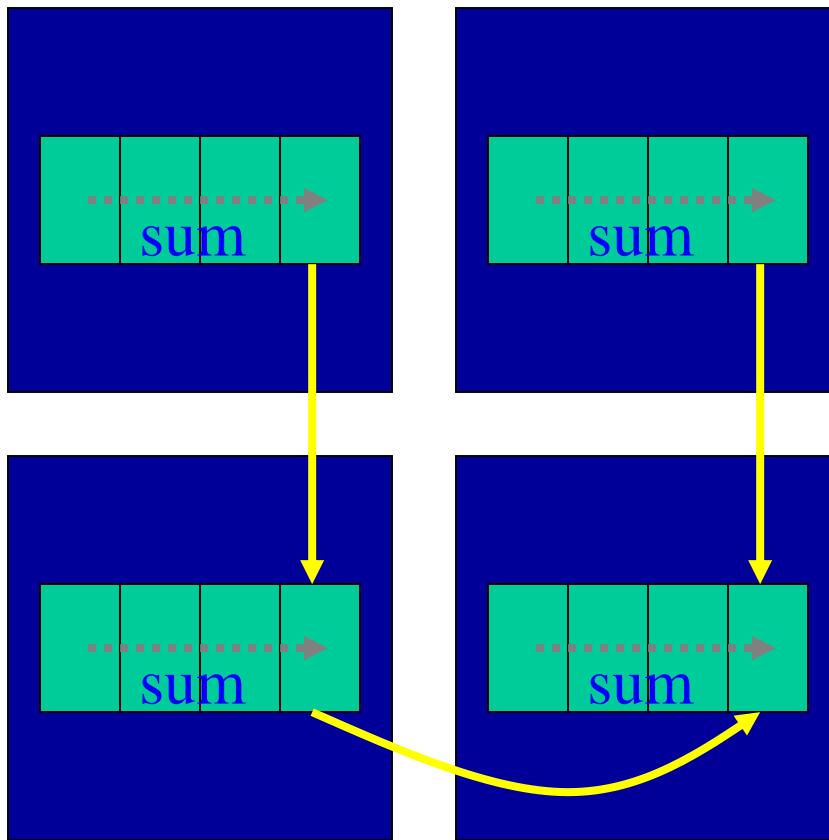
Agglomeration



Observations:

1. Number of tasks are static
2. Computations per task are trivial
3. Communication pattern is regular

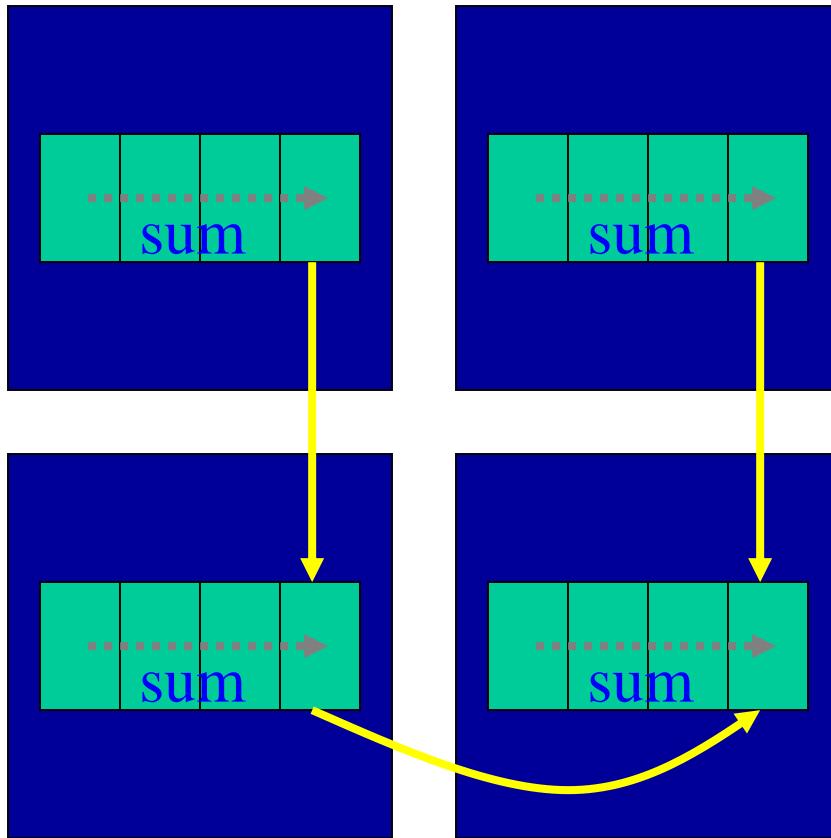
Agglomeration



Observations:

1. If x time required for a task to communicate another task
2. y time is required for binary operation then
3. Total time will be $(n-1)(x+y)$

Agglomeration



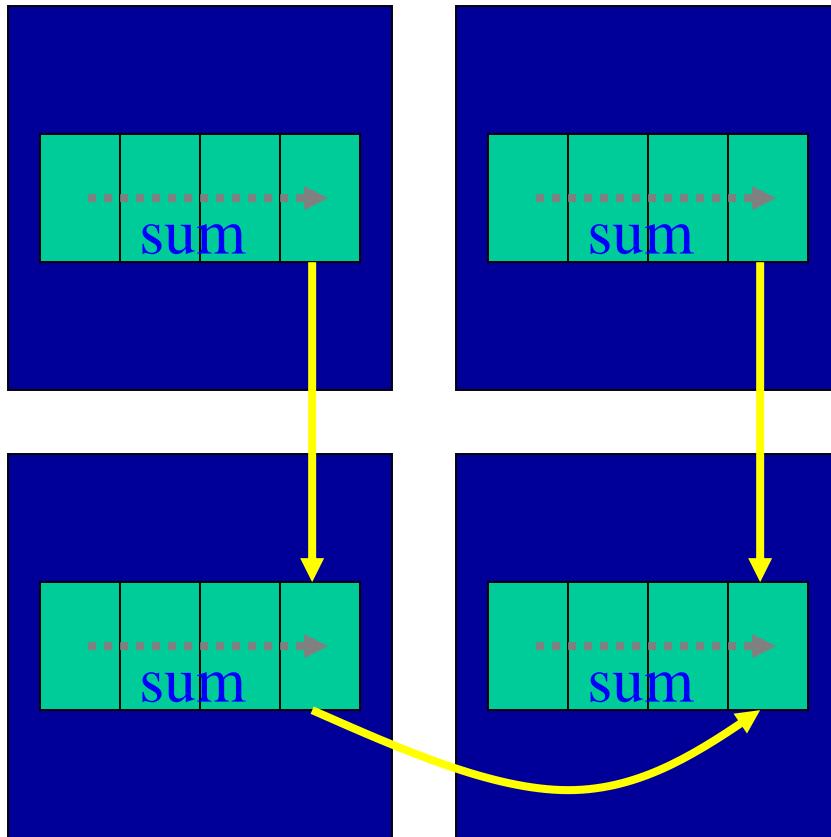
Observations:

1. If we have p processors
2. Time require to compute sub total is $(\lceil n/p \rceil - 1)y$
3. Reduction require $x+y$ time
4. With $\lceil \log p \rceil$ communications overall time

Agglomeration

Total time:

$$(\lceil n/p \rceil - 1)y + \lceil \log p \rceil(x+y)$$

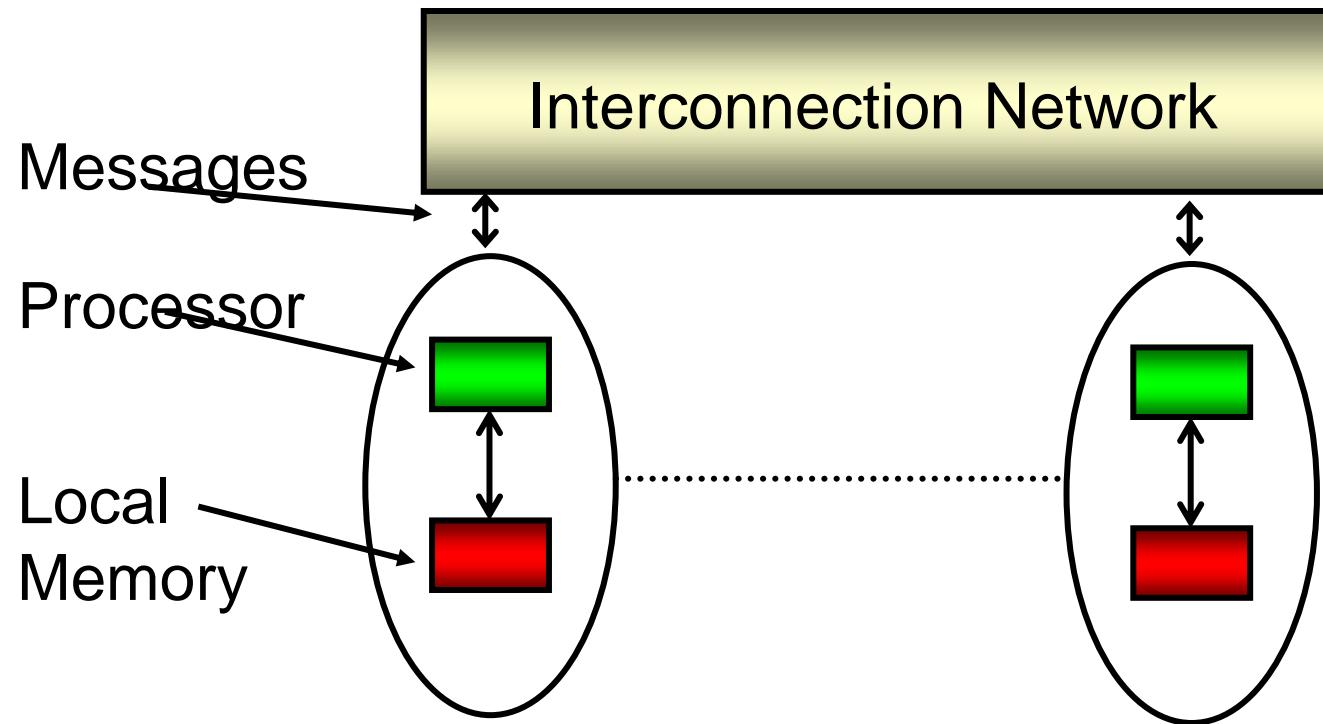


Message-Passing Computing for Distributed Multi-Computers

A review of basic concepts

Message-Passing Multicomputer

Complete computers connected through an interconnection network:



Programming

Programming a message-passing multicomputer can be achieved by

- ▶ Designing a special parallel programming language
- ▶ Extending the syntax/reserved words of an existing sequential high-level language to handle message passing
- ▶ Using an existing sequential high-level language and providing a ***library of external procedures for message passing***

Programming

Involves dividing problem into parts (domain or functional) that are intended to be executed **simultaneously** to solve the problem

Each part executed by **separate computers**

Parts (processes) communicate by sending **messages** - the only way to distribute data and collect result

Message Passing Parallel Programming Software Tools

Parallel Virtual Machine (PVM) - developed in late 1980's.
Became very popular.

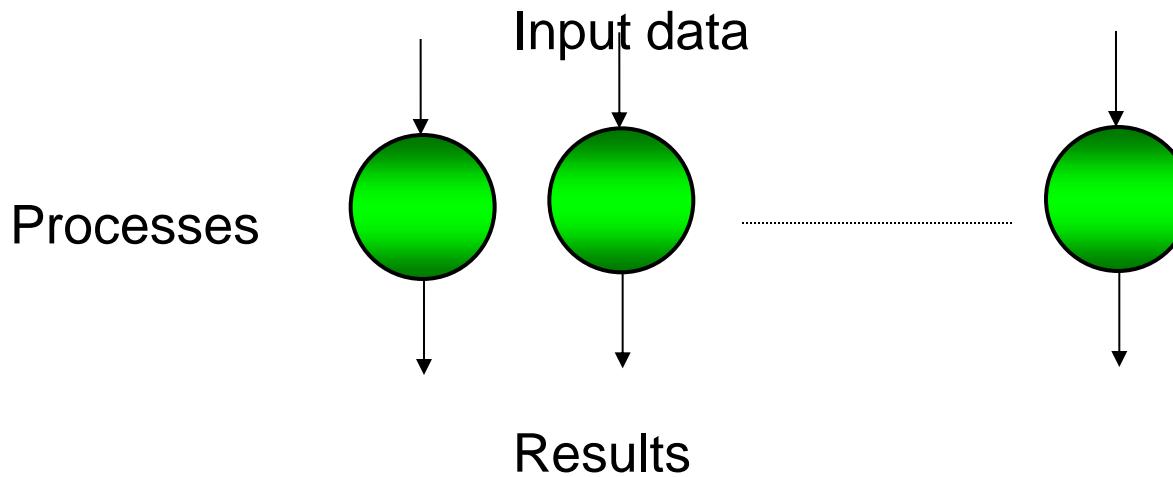
Message-Passing Interface (MPI) - standard defined in
1990s.

Both provide a set of user-level libraries for message
passing. Use with regular programming languages
(FORTRAN, C, C++, ...).

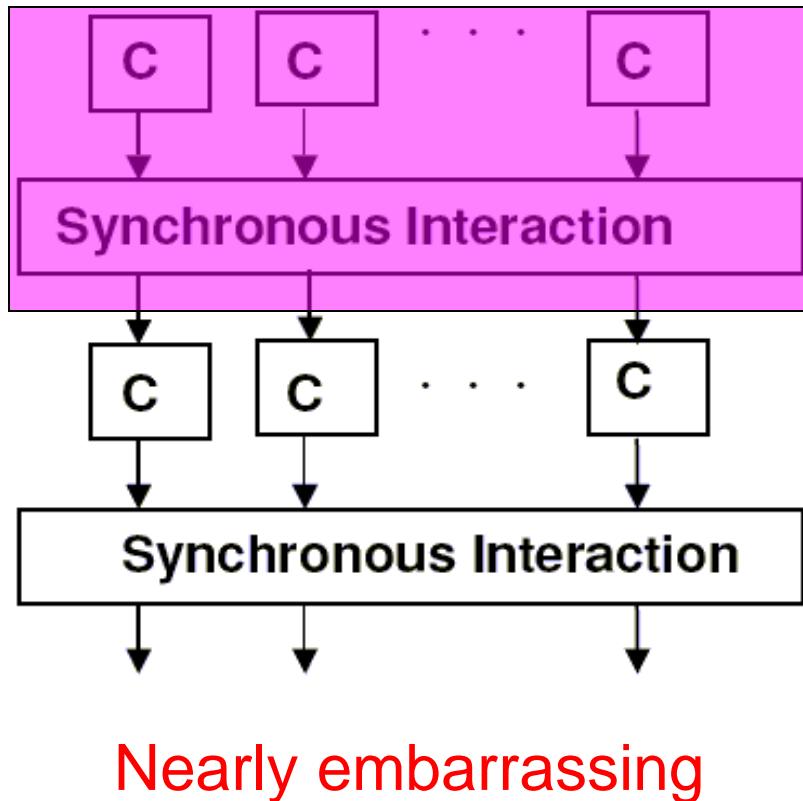
Embarrassingly Parallel Computations

A computation that can 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



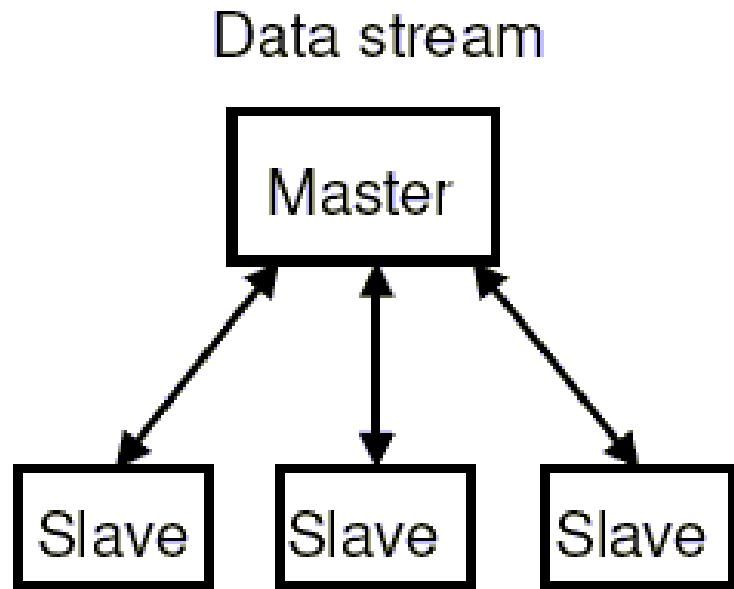
Phase Parallel Model



Nearly embarrassing
One can see how close
it is to BSP

- ❖ The phase-parallel model offers a paradigm that is widely used in parallel programming.
- ❖ The parallel program consists of a number of super steps, and each has two phases.
- ❖ In a computation phase, multiple processes each perform an independent computation C.
- ❖ In the subsequent interaction phase, the processes perform one or more synchronous interaction operations, such as a barrier or a blocking communication.
- ❖ Then next super step is executed.

Process farm



- ❖ This paradigm is also known as the **master-slave** paradigm.
- ❖ A master process executes the essentially sequential part of the parallel program and spawns a number of slave processes to execute the parallel workload.
- ❖ When a slave finishes its workload, it informs the master which assigns a new workload to the slave.
- ❖ This is a very simple paradigm, where the coordination is done by the master.

MPI (Message Passing Interface)

Standard developed by group of academics and industrial partners to foster more widespread use and portability

Defines routines, not implementation

Several free implementations of MPI standard exist.

List of Few active products/projects

- CRI/EPCC
- Hitachi MPI
- HP MPI
- IBM Parallel Environment for AIX-MPI Library
- LAM/MPI (Supplier: Indiana University)
- MPI for UNICOS Systems
- MPICH (Supplier: Argonne National Laboratory)
- OS/390 Unix System Services Parallel
- Intel MPI
- SGI Message Passing Toolkit
- Sun MPI
- Open MPI

List of Few active products/projects

- CRI/EPCC
- Hitachi MPI
- HP MPI
- IBM Parallel Environment for AIX-MPI Library
- LAM/MPI (Supplier: Indiana University)
- MPI for UNICOS Systems
- **MPICH** (Supplier: Argonne National Laboratory)
- OS/390 Unix System Services Parallel
- Intel MPI
- SGI Message Passing Toolkit
- Sun MPI
- **Open MPI**

What is MPI

- ❖ A **message-passing library** specification
 - Not a compiler specification
 - Not a specific product
- ❖ Used for parallel computers, clusters, and heterogeneous networks as a message passing library
- ❖ Designed to be used for the development of parallel software libraries
- ❖ Designed to provide access to advanced parallel hardware for
 - End users
 - Library writers
 - Tool developers

Where to use MPI?

- We need a portable parallel program
- We are writing a parallel Library

Why learn MPI?

- Portable
- Expressive
- Good way to learn about subtle issues in parallel computing
- Universal acceptance

- The attractiveness of the message-passing paradigm is its wide portability.
- Programs expressed this way may run on distributed-memory multiprocessors, networks of workstations, and combinations of all of these.
- In addition, shared-memory implementations are possible.

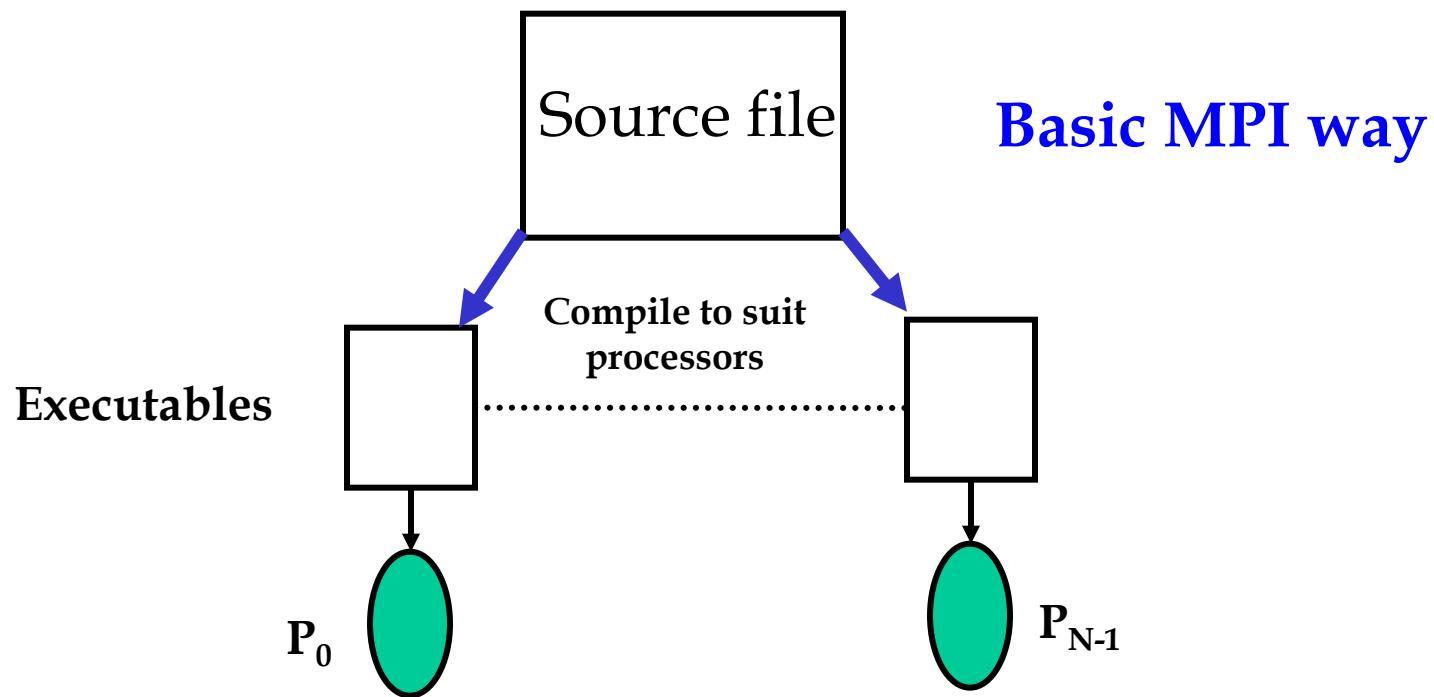
Basics of Message-Passing Programming

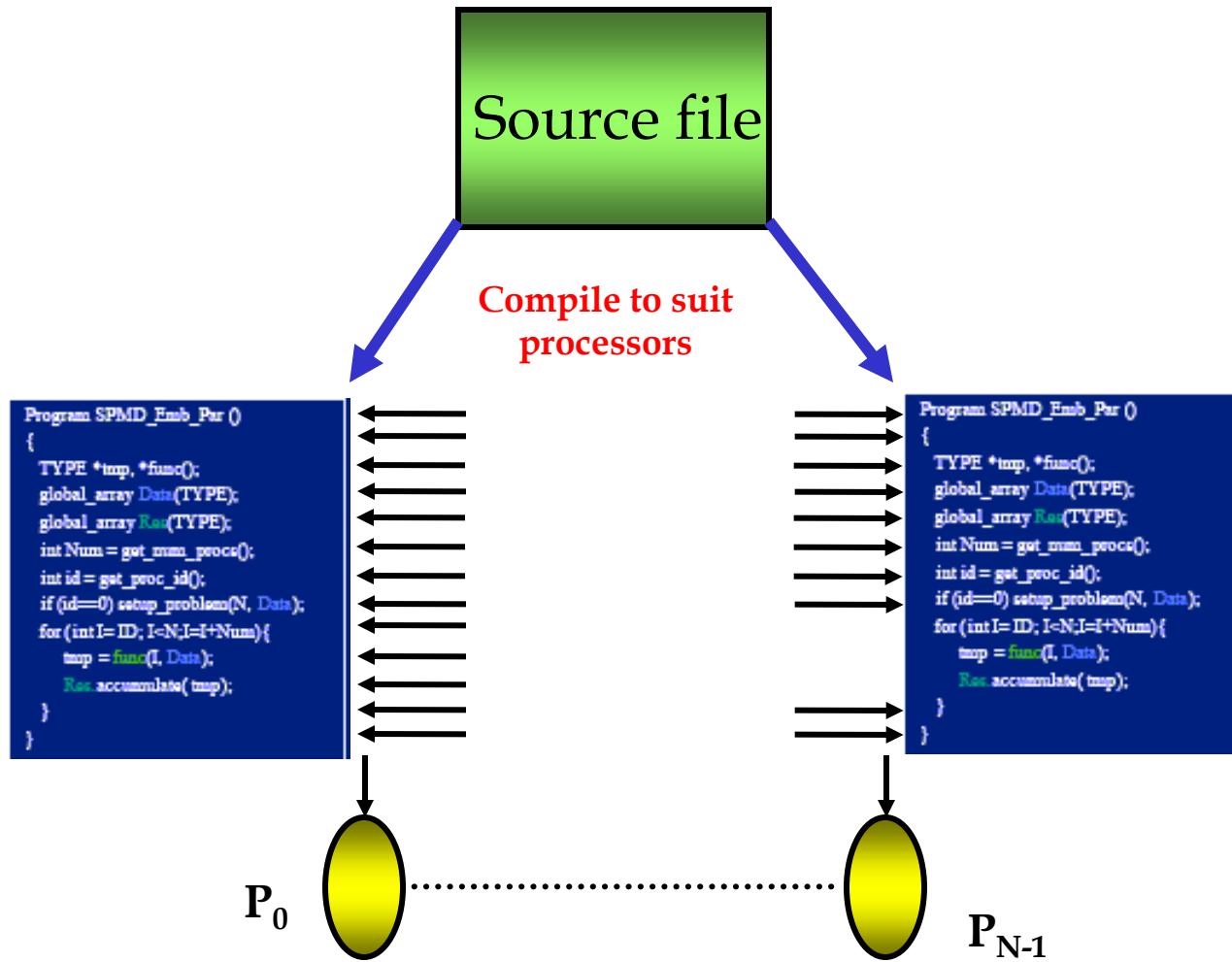
Two primary mechanisms needed:

1. A method of creating separate **processes** for execution on different computers
2. A method of sending and receiving messages

Single Program Multiple Data (SPMD) model

Different processes merged into one program. Within program, control statements select different parts **for each processor to execute**. All executables start together - static process creation.





Evaluating General Message

Message Passing SPMD : C program

```
main (int argc, char **argv)
{
    if (process is to become a controller process)
    {
        Controller /* Arguments */;
    }
    else
    {
        Worker /* Arguments */;
    }
}
```

A History of MPICH

- ❖ MPICH was developed during the MPI standards process to provide feedback to the MPI forum on implementation and usability issues.
- ❖ With the release of the MPI standard, MPICH was designed to provide an implementation of the MPI standard.
- ❖ It supported both MIMD programming and heterogeneous clusters from the very beginning.

MPICH

- ❖ MPICH is an open-source, portable implementation of the Message-Passing Interface Standard.
- ❖ Designed at Mathematics and Computer Science Division of **Argonne National Laboratory**.
- ❖ The “CH” in MPICH stands for **Chameleon** symbol of adaptability to one's environment and thus of portability.

MPICH

William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.

It contains a complete implementation of version 3.1 of the MPI Standard and also significant parts of MPI-2, particularly in the area of parallel I/O.

MPICH

- ❖ MPICH is the latest implementation of MPI.
Present version in 3.4a.
- ❖ The features in MPICH also include support for one-side communication, dynamic processes, inter communicator collective operations, and expanded MPI-IO functionality.
- ❖ Clusters consisting of both single-processor and SMP nodes are supported.

MPICH Architecture

- Most code is completely portable
- An Abstract Device defines the communication layer
- The abstract device (The central mechanism for achieving the goals of portability and performance is a specification we call the abstract device interface (ADI)) can have widely varying instantiations, using:
 - sockets
 - shared memory
 - other special interfaces
 - e.g. Myrinet, Quadrics, InfiniBand, Grid protocols

MPICH implementations

MPICH is freely available and is distributed as open source.

- The Unix/Linux (all flavors) version of MPICH
- The Microsoft Windows version of MPICH (mpich-3.2.1)
- MVAPICH2-GDR, support for InfiniBand and NVIDIA CUDA GPUs

Is MPICH Large or Small?

MPICH is large (~396 Functions)

- MPICH's extensive functionality requires many functions
- Number of functions not necessarily a measure of complexity

MPICH is small (6 Functions)

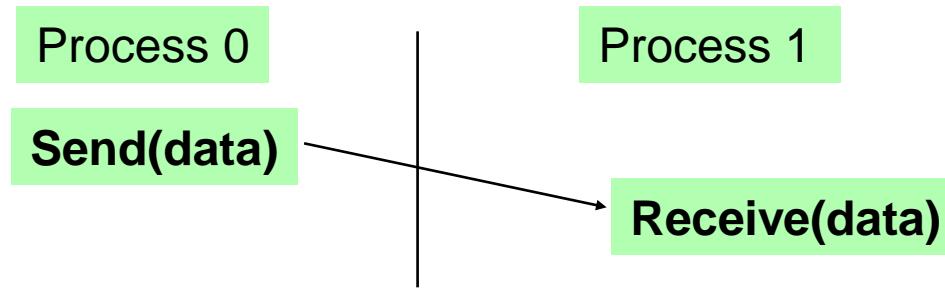
- Many parallel programs can be written with just 6 basic functions

MPICH is just **right** candidate for message passing

- One **need not** master all parts of MPICH to use it

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
 - How will “data” be described?
 - How will processes be identified?
 - How will the receiver recognize/screen messages?
 - What will it mean for these operations to complete?

Some Basic Concepts

- Processes can be collected into **groups**
- Each message is sent in a **context**, and must be received in the same context
- A group and context together form a **communicator**
- A process is identified by its **rank** in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called
MPI_COMM_WORLD

Begin programming with 6 MPI function calls

- MPI_INIT *Initializes MPI*
- MPI_COMM_SIZE *Determines number of processes*
- MPI_COMM_RANK *Determines the label of the calling process*
- MPI_SEND *Sends a message*
- MPI_RECV *Receives a message*
- MPI_FINALIZE *Terminates MPI*

MPI Datatypes

- The data in a message to send or receive is described by a triple (**address, count, datatype**), where
- An MPI *datatype* is recursively defined as:
 - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
 - a contiguous array of MPI datatypes
 - a strided block of datatypes
 - an indexed array of blocks of datatypes
 - an arbitrary structure of datatypes
- There are MPI functions to **construct custom datatypes**, in particular ones for subarrays

C data types

- **MPI_CHAR**
char
- **MPI_BYTE**
like unsigned char
- **MPI_SHORT**
short
- **MPI_INT**
int
- **MPI_LONG**
long
- **MPI_FLOAT**
float
- **MPI_DOUBLE**
double
- **MPI_UNSIGNED_CHAR**
unsigned char
- **MPI_UNSIGNED_SHORT**
unsigned short
- **MPI_UNSIGNED**
unsigned int
- **MPI_UNSIGNED_LONG**
unsigned long
- **MPI_LONG_DOUBLE**
long double (some systems may not implement)
- **MPI_LONG_LONG_INT**
long long (some systems may not implement)

FORTRAN data types

- **MPI_REAL**
REAL
- **MPI_INTEGER**
INTEGER
- **MPI_LOGICAL**
LOGICAL
- **MPI_DOUBLE_PRECISION**
DOUBLE PRECISION
- **MPI_COMPLEX**
COMPLEX
- **MPI_DOUBLE_COMPLEX**
complex*16 (or
complex*32) where
supported

The following data types are optional

- **MPI_INTEGER1**
integer*1 if supported
- **MPI_INTEGER2**
integer*2 if supported
- **MPI_INTEGER4**
integer*4 if supported
- **MPI_REAL4**
real*4 if supported
- **MPI_REAL8**
real*8 if supported

Caution!

- 👉 Fortran types should only be used in Fortran programs,
- 👉 C types should only be used in C programs.

For example, it is in error to use `MPI_INT` for a Fortran `INTEGER`, which should be `MPI_INTEGER`.

MPI_Op Options (Collective Operation)

- MPI_BAND
- MPI_BOR
- MPI_BXOR
- MPI_LAND
- MPI_LOR
- MPI_LXOR
- MPI_MAX
- MPI_MAXLOC*
- MPI_MIN
- MPI_MINLOC
- MPI_PROD
- MPI_SUM

* Maximum and Location

Communicators

Defines *scope* of a communication operation.

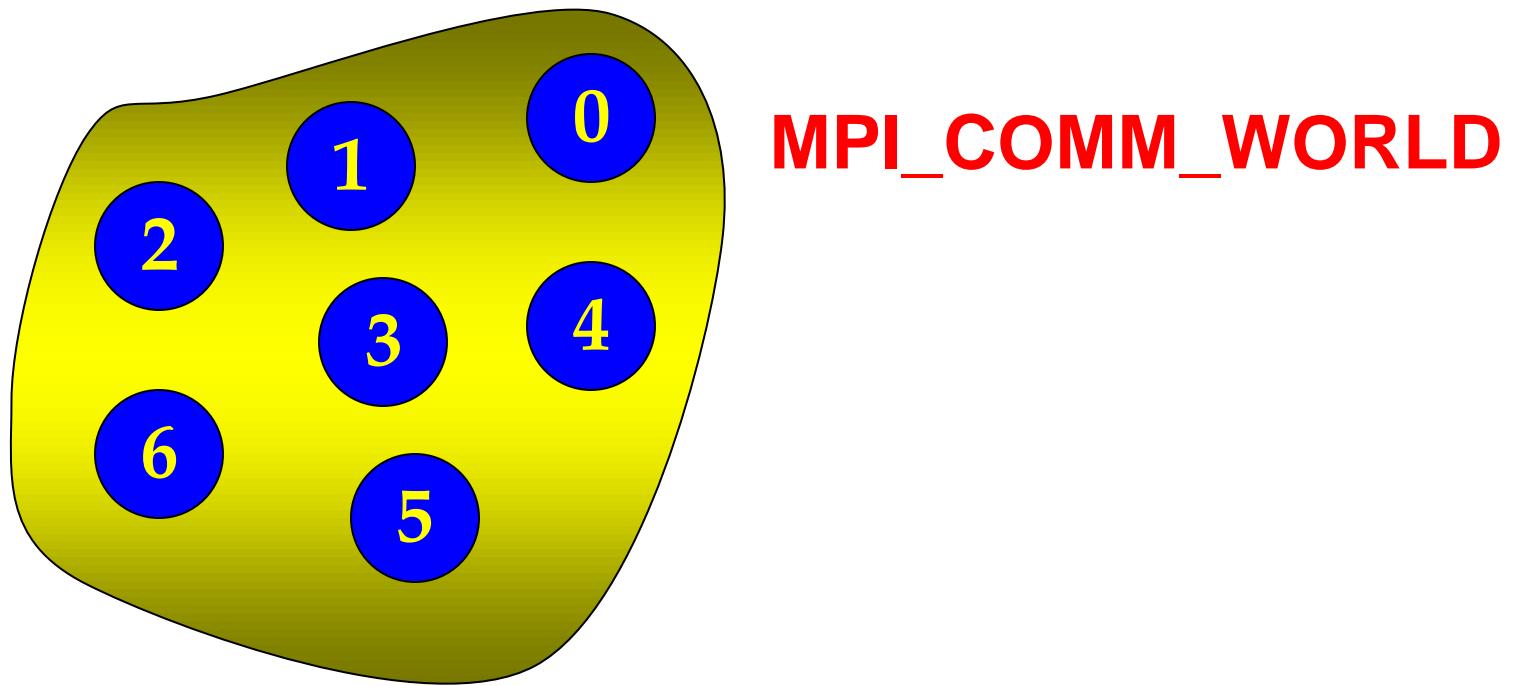
Processes have **ranks** associated with the communicator.

Initially, all processes enrolled in a “**universe**” called **MPI_COMM_WORLD** and each process is given a unique rank, a number from 0 to $n - 1$, where there are n processes.

Other communicators can be established for groups of processes.

MPI_COMM_WORLD communicator

This **Default Communicator** is MPI's mechanism for establishing individual communication universes



MPI Messages

Message : data (3 parameters) + envelope (3 parameters)

Data: startbuf, count, datatype

- **Startbuf:** address where the data starts
- **Count:** number of elements (items) of data in the message

Envelope: dest, tag, comm

- **Destination or Source:** Sending or Receiving processes
- **Tag:** Integer to distinguish messages

Communicator:

The communicator is communication “universe.”

Messages are sent or received within a given “universe.”

Synchronous Message Passing (Blocking)

Routines that actually return when message transfer completed.

Synchronous send routine Waits until complete message can be accepted by the receiving process before sending the next message.

Synchronous receive routine Waits until the message it is expecting arrives.

Synchronous routines fundamentally perform two actions: They **transfer data** and they **synchronize** processes.

Asynchronous Message Passing (Non-Blocking)

Routines that do not wait for actions to complete before returning. Usually require **local storage** for messages.

More than one version depending upon the actual semantics for returning.

In general, they do not synchronize processes but allow processes to move forward sooner. **Must be used with care.**

MPICH Send and Recv

- Communication between two processes
- *Source* process sends message to *destination* process
- Communication takes place within a *communicator*
- Destination process is identified by its *rank* in the communicator

Parameters of the blocking send

`MPI_Send(buf, count, datatype, dest, tag, comm)`

Address of Send buffer Number of items to send Data type of each item Rank of destination process Message tag Communicator

MPI Basic (Blocking) Send

- When this function returns, the data has been delivered to the system and the buffer can be reused.

Parameters of the blocking receive

MPI_Recv(buf, count, datatype, src, tag, comm, status)

Address of
receive buffer

Data type of
each item

Message tag

Status after
operation

Maximum number
of items to receive

Rank of source
process

Communicator

MPI Basic (Blocking) Receive

- Waits until a matching (both **source** and **tag**) message is received from the system, and the buffer can be used
- **source** is rank in communicator specified by **comm**, or **MPI_ANY_SOURCE**
- **tag** is a tag to be matched on or **MPI_ANY_TAG**
- receiving fewer than **count** occurrences of **datatype** is OK, but **receiving more is an error**
- **status** contains further information (e.g. size of message)

Message Tag

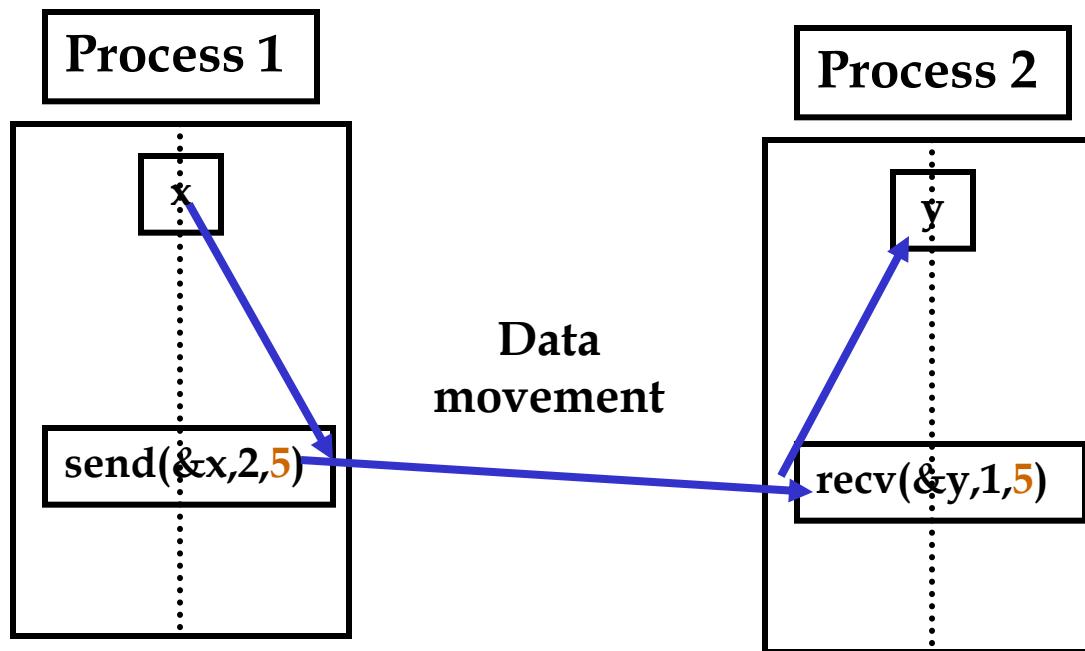
Used to differentiate between different types of messages being sent.

Message tag is carried within message.

If special type matching is not required, a *wild card* message tag is used, so that the `recv()` will match with any `send()`.

Message Tag Example

To send a message **x** with message tag **5** from a source process 1 to a destination process 2 and assign to **y**:



Waits for a message from process 1 with a tag of 5

Initializing MPICH

- Must be first routine called
- `int MPI_Init(int *argc, char **argv);`

What makes an MPICH Program?

Include files

- mpi.h (c)
- mpif.h (Fortran)

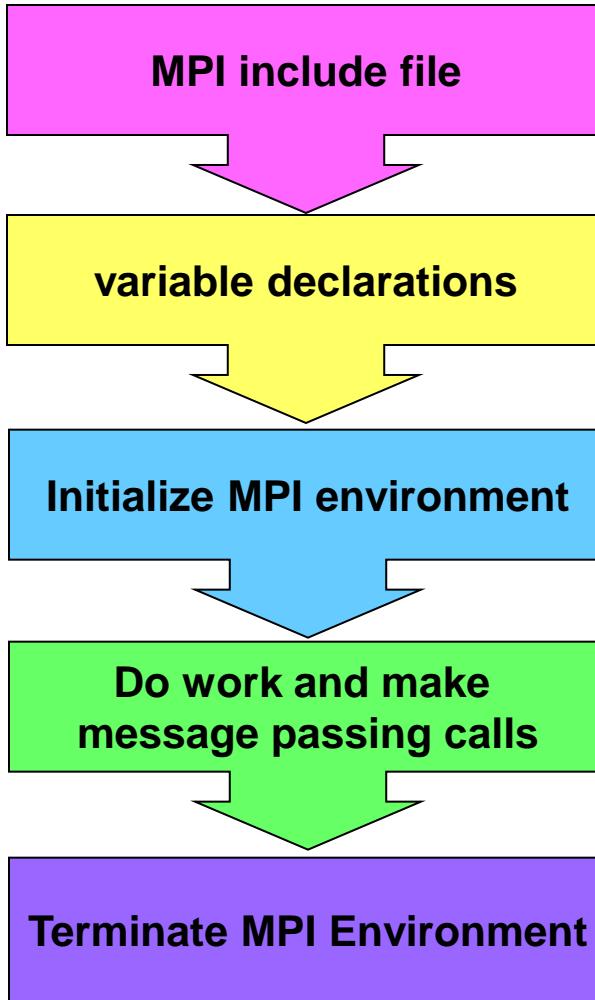
Initiation of MPI

- MPI_INIT

Completion of MPI

- MPI_FINALIZE

General MPI Program Structure



```
#include <mpi.h>
void main (int argc, char **argv)
{
    int np, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    /*      Do Some Works           */
    MPI_Finalize();
}
```

MPI_Init

- ❖ The command line arguments are provided to **MPI_Init** to allow an MPI implementation to use them in initializing *the MPI environment*.
- ❖ They are passed **by reference** to allow an MPI implementation to provide them in environments where the command-line arguments are not provided to **main function**.

MPI_Init

- ❖ At least one process has access to `stdin`, `stdout`, and `stderr`
- ❖ The user can find out which process this is by querying the attribute `MPI_IO` on `MPI_COMM_WORLD`
- ❖ In MPICH all processes have access to `stdin`, `stdout`, and `stderr` and on networks these I/O streams are routed back to the process with rank 0 in `MPI_COMM_WORLD`.

MPI_Init

- ❖ On most systems, these streams also can be redirected through mpirun, as follows

```
mpirun -np 64 myprog -myarg 13 <data.in>  
results.out
```

- ❖ Here we assume that –myarg 13 are command-line arguments processed by the application myprog. After MPI_Init, each process will have these arguments in its argv

Example

To send an integer x from process 0 to process 1,

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find rank */  
if (myrank == 0) {  
    int x;  
    MPI_Send(&x, 1, MPI_INT, 1, msgtag,  
             MPI_COMM_WORLD);  
} else if (myrank == 1) {  
    int x;  
    MPI_Recv(&x, 1, MPI_INT, 0, msgtag,  
             MPI_COMM_WORLD, status);  
}
```

MPI_Init

- **MPI_Init** is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- Calling **MPI_Init** more than once during the execution of a program **will lead to an error**.
- **MPI_Finalize** is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- No MPI calls may be performed after **MPI_Finalize** has been called, not even **MPI_Init**.

MPI_Init

- Both **MPI_Init** and **MPI_Finalize** must be called by all the processes, otherwise MPI's behavior will be undefined.
- The exact calling sequences of these two routines for C are as follows:
 - `int MPI_Init(int *argc, char ***argv)`
 - `int MPI_Finalize()`

MPI_Init

- The arguments `argc` and `argv` of `MPI_Init` are the command-line arguments of the C program.
- An MPI implementation is expected to remove from the `argv` array any command-line arguments that should be processed by the implementation before returning back to the program, and to decrement `argc` accordingly.

MPI_Init

- Thus, command-line processing should be performed only after **MPI_Init** has been called.
- Upon successful execution, **MPI_Init** and **MPI_Finalize** return **MPI_SUCCESS**; otherwise they return an implementation-defined error code.

Let us write the first Complete C/MPICH program

Write a simple parallel program in which every process with **rank greater than 0** sends a message “Hello-Participants” to a process with **rank 0**. The processes with **rank 0** receives the message and prints it.

```
#include "mpi.h"           ←
main (int argc, char **argv) {
    int MyRank, Numprocs, tag, ierror, i;   ←
    MPI_Status status;   ←
    char send_message[20], recv_message[20];
    MPI_Init (&argc, &argv);   ←
    MPI_Comm_size (MPI_COMM_WORLD, &Numprocs); ←
    MPI_Comm_rank (MPI_COMM_WORLD, &MyRank);   ←
    tag = 100;   ←
    strcpy (send_message, "Hello-Participants");
    if (MyRank==0) {   ←
        for (i=1; i<Numprocs; i++) {   ←
            MPI_Recv (recv_message,20, MPI_CHAR, i, tag, MPI_COMM_WORLD,&status); ←
            printf ("node %d : %s \n", i, recv_message);   ←
        }
    } else
        MPI_Send(send_message, 20, MPI_CHAR,0, tag, MPI_COMM_WORLD);   ←
    MPI_Finalize();   ←
}
```

Basic instructions for compiling/executing MPICH programs

Preliminaries

- Set up paths
- Create required directory structure
- Modify makefile to match your source file
- Create a file (hostfile) listing machines to be used (required)

Compiling/executing (SPMD) C/MPICH program

To compile MPI programs:

`mpicc -o file file.c`

Or

`mpiCC -o file file.cpp`

To execute MPI program: `mpirun -np no_processes file`

Basic instructions for compiling/executing MPI programs

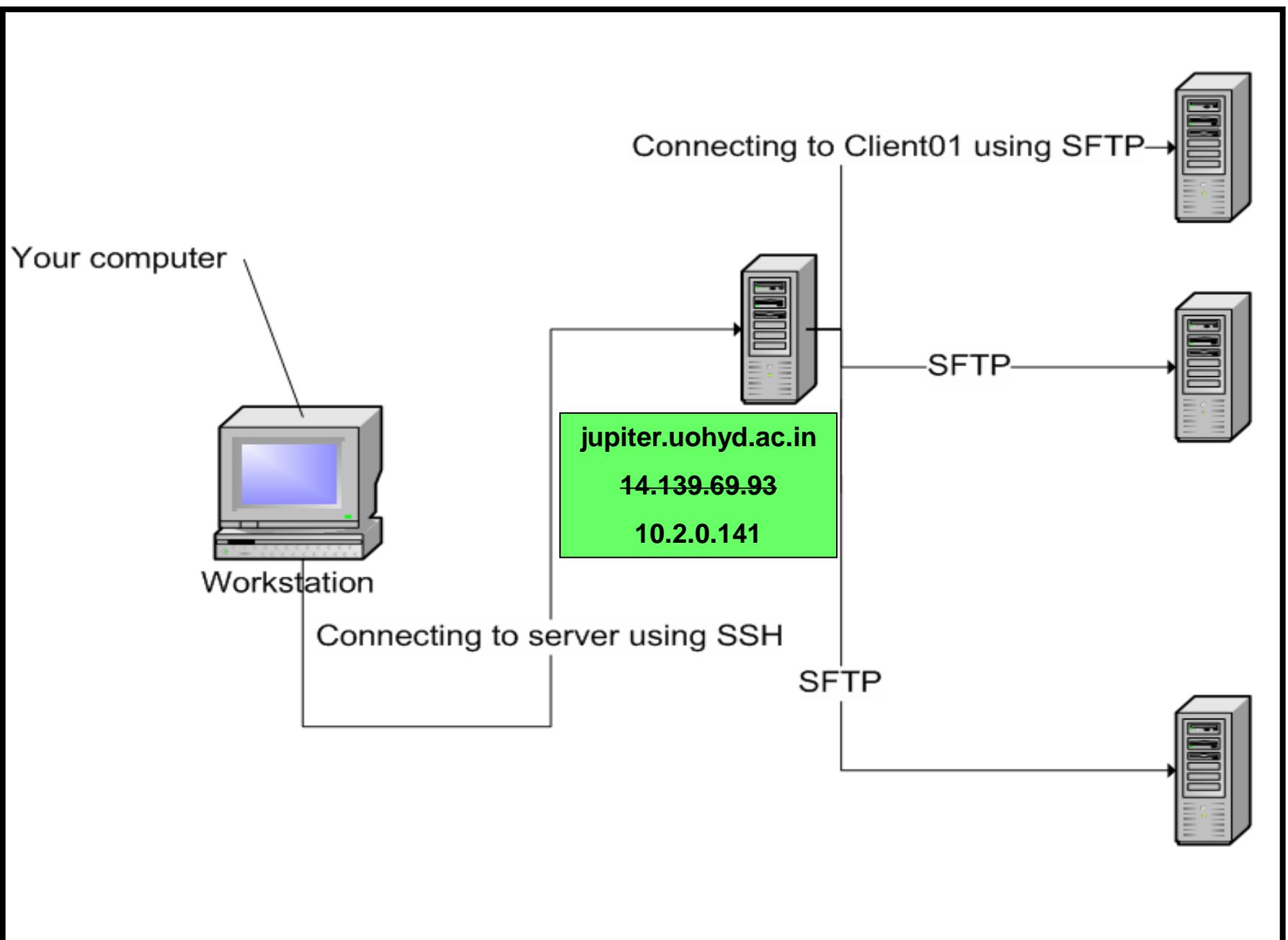
Depends on your implementation of MPI

❑ For Mpich/OpenMPI:

- mpirun -np 2 a001 program # run MPI

❑ For lam:

- lamboot -v lamhosts # starts LAM
- mpirun -v -np 2 a001 # run MPI program
- lamclean -v # rm all user processes
- mpirun ... # run another program
- lamclean ...
- lamhalt # stop LAM



Procedure Speciation

MPI procedures are specified using a language independent notation. The arguments of procedure calls are marked as IN, OUT or INOUT. The meanings of these are:

- ❖ the call uses but does not update an argument marked IN,
- ❖ the call may update an argument marked OUT,
- ❖ the call both uses and updates an argument marked INOUT.

Procedure Speciation

- ❖ There is one special case: if an argument is a **handle** to an **opaque object** and the object is updated by the procedure call, then the argument is marked OUT.
- ❖ It is marked this way even though the handle itself is not modified we use the OUT attribute to denote that the **handle references is updated**.

Opaque objects

- ❖ MPI manages system memory that is used for buffering messages and for storing internal representations of various MPI objects such as groups, communicators, datatypes, etc.
- ❖ This memory is not directly accessible to the user, and **objects stored there are opaque**: their size and shape is not visible to the user. Opaque objects are accessed via **handles**, which exist in user space.
- ❖ In Fortran, all handles have type INTEGER. In C, a different handle type is defined for each category of objects.

Collective Communications

The sending and/or receiving of messages to/from **groups of processes**. A collective communication implies that all processes need to participate in the communication.

- Involves coordinated communication within a group of processes
- No message tags used
- All collective routines block until they are locally complete

Collective Communication

- Two broad classes:
 - Data movement routines
 - Global computation routines
- Called by all processes in a communicator

Examples:

- Barrier synchronization
- Broadcast, scatter, gather
- Global sum, global maximum, etc.

P0	A			
P1				
P2				
P3				

Broadcast

P0	A			
P1	A			
P2	A			
P3	A			

P0	A	B	C	D
P1				
P2				
P3				

Scatter

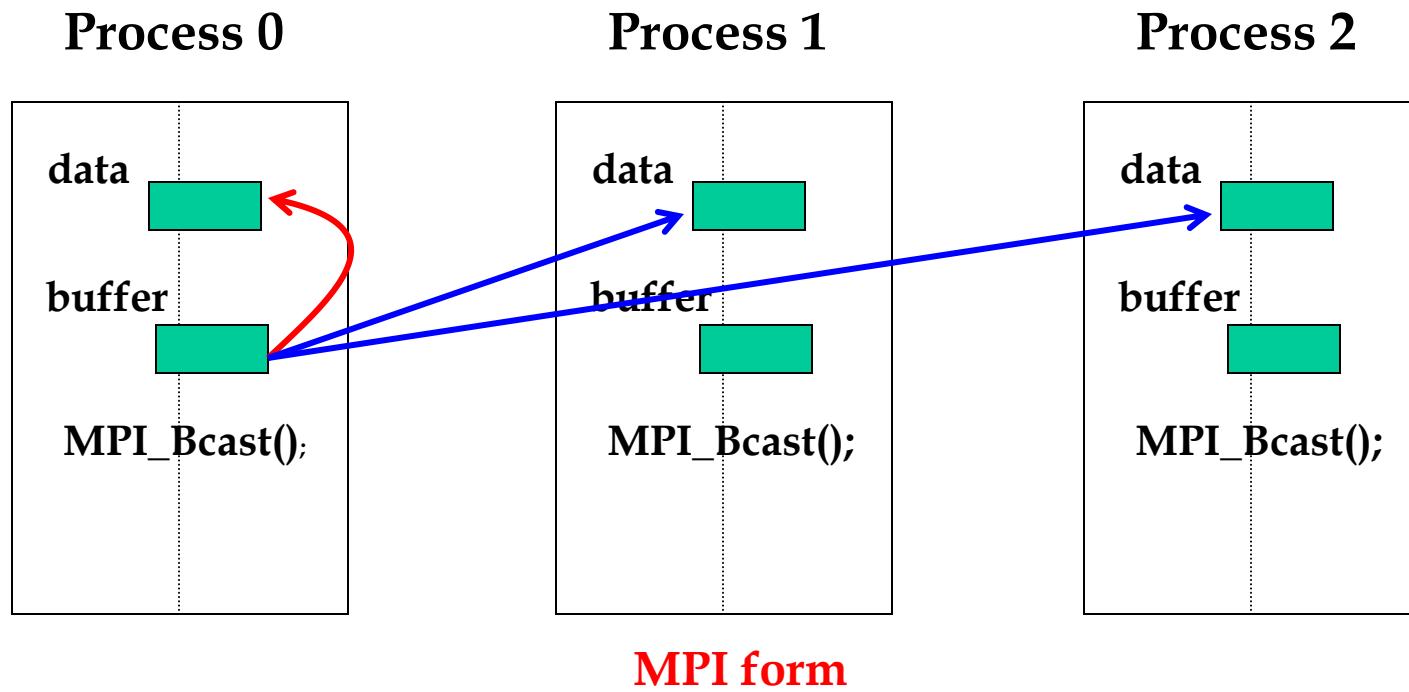
Gather

P0	A			
P1	B			
P2	C			
P3	D			

Representation of collective data movement in MPI

Broadcast

Sending same message to all processes concerned with problem.
Multicast - sending same message to defined group of processes.



Broadcast

A broadcast sends data from one processor to all other processors, **including itself**.

C:

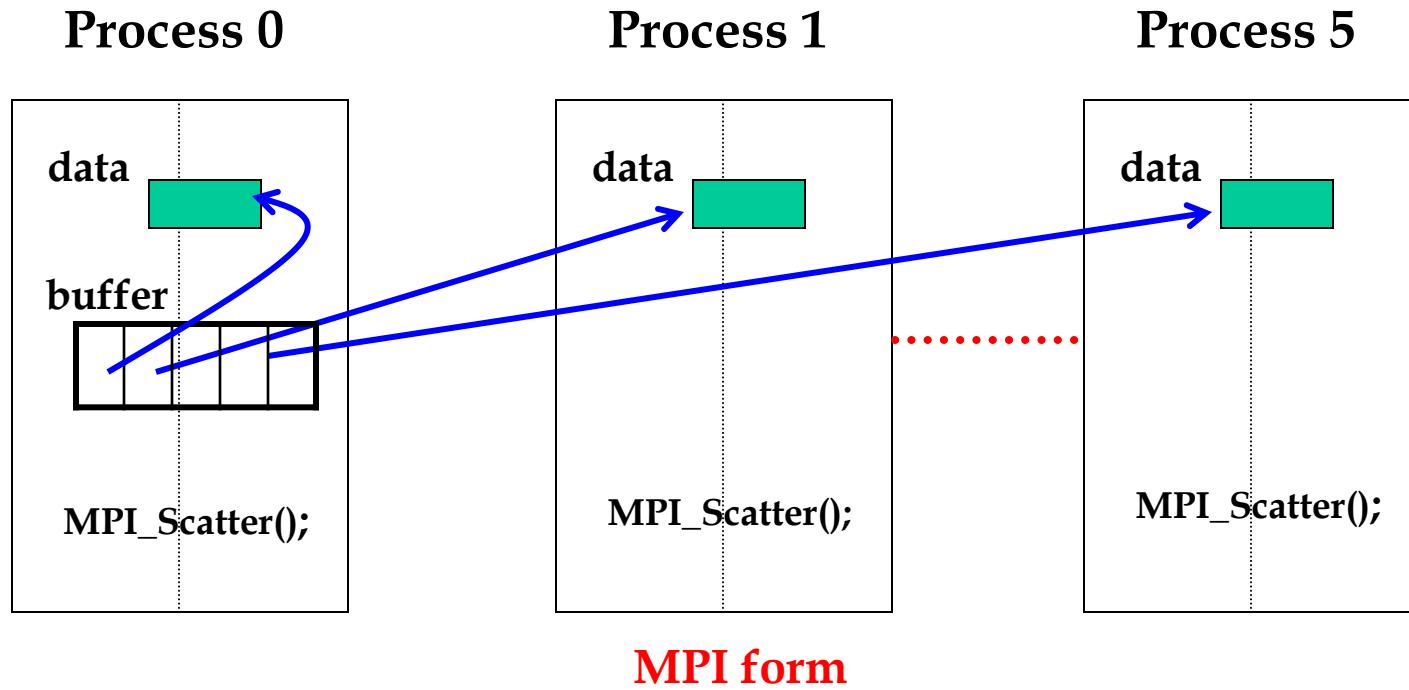
```
int MPI_Bcast ( void *buffer, int count, MPI_Datatype  
datatype, int root, MPI_Comm comm);
```

Input/output Parameters

INOUT	buffer	starting address of buffer
IN	count	number of entries in buffer
IN	Datatype	data type of buffer
IN	root	rank of broadcast root
IN	comm	communicator

Scatter

In its simplest form sending each element of an array in **root** process to a separate process. Contents of i^{th} location of array sent to i^{th} process.



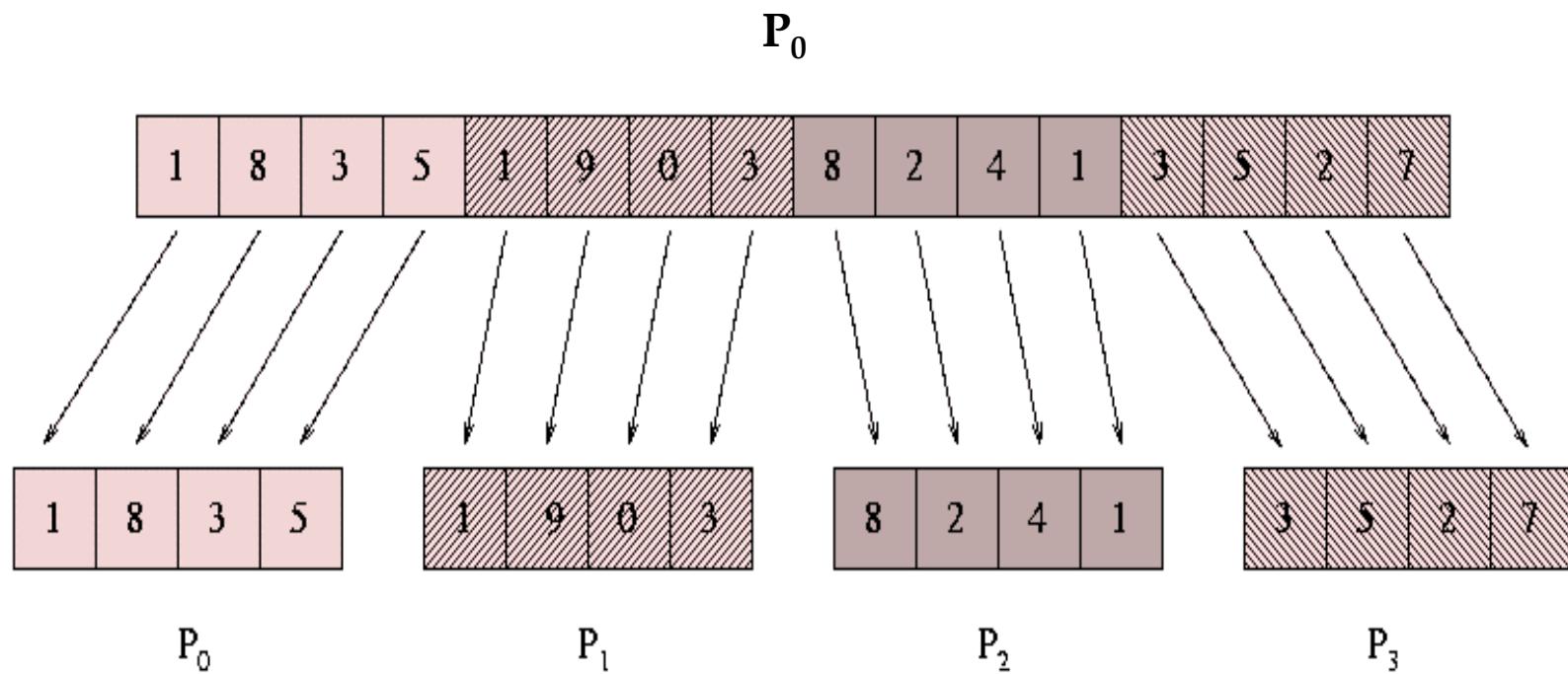
Scatter

The process with rank *root* distributes the contents of *send-buffer* among the processes. The contents of *send-buffer* are split into *p* segments each consisting of *send_count* elements. The first segment goes to process 0, the second to process 1, etc... **The send arguments are significant only on process root.**

C:

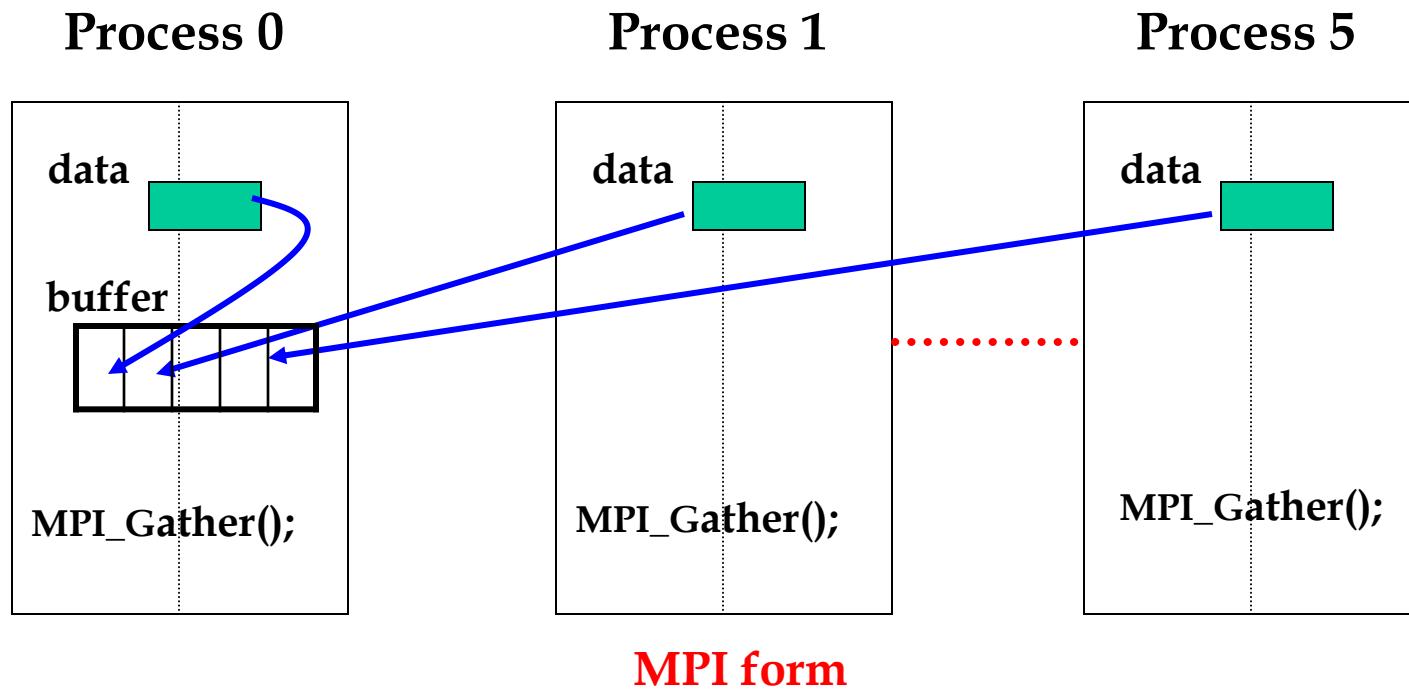
```
int MPI_Scatter( void *send_buffer, int send_count,  
MPI_Datatype send_type, void *recv_buffer, int recv_count,  
MPI_Datatype recv_type, int root, MPI_Comm comm);
```

Scatter



Gather

Having one process collect individual values from set of processes.



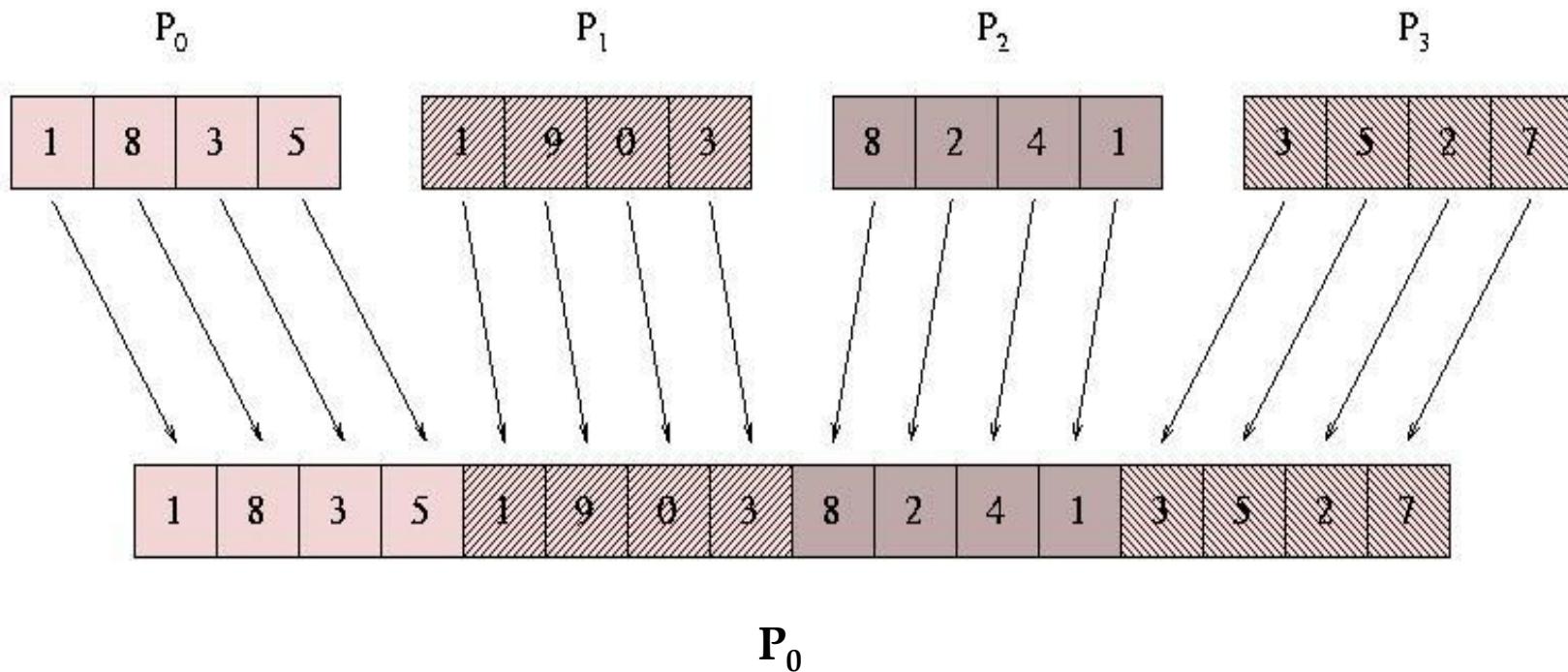
Gather

Each process in *comm* sends the contents of *send-buffer* to the process with rank *root*. The process with rank *root* concatenates the received data in the process rank order in *recv-buffer*. The argument *recv_count* indicates the number of items received from each process – not the total number received.

C:

```
int MPI_Gather( void *send_buffer, int send_count,  
MPI_Datatype send_type, void *recv_buffer, int recv_count,  
MPI_Datatype recv_type, int root, MPI_Comm comm);
```

Gather

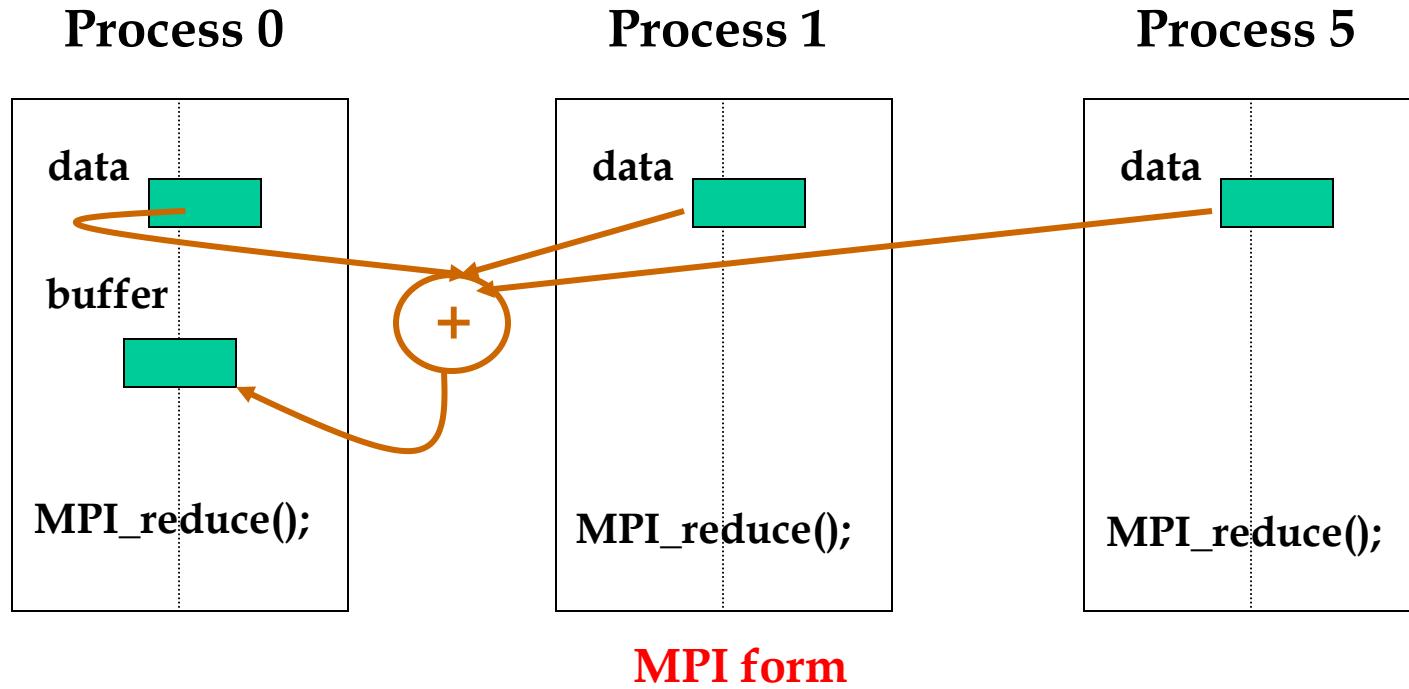


- Your ID on jupiter is your registration number
- Password is test123, change immediately
- Press “Enter” if it asks any key to enter
- jupiter.uohyd.ac.in
- 14.139.69.93 (from outside) **shh only on port 8223**
- 10.2.0.141 (from inside UoH)
- Browser: <http://jupiter.uohyd.ac.in/wordpress/>

Reduce

It is a Gather operation combined with specified arithmetic/logical operation.

Values could be gathered and then added together by root:



MPI_Reduce

```
int MPI_Reduce (void *sendbuf, void *recvbuf, int count,  
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm  
comm )
```

Input Parameters

sendbuf	address of send buffer (choice)
count	number of elements in send buffer
datatype	data type of elements of send buffer (handle)
op	reduce operation
root	rank of root process (integer)
comm	communicator (handle)

Output Parameter

recvbuf address of receive buffer (significant only at root)

Predefined reduce operations

[MPI_Max]	maximum
[MPI_Min]	minimum
[MPI_Sum]	sum
[MPI_Prod]	product
[MPI_Land]	logical and
[MPI_Band]	bit-wise and
[MPI_Lor]	logical or
[MPI_Bor]	bit-wise or
[MPI_Lxor]	logical xor
[MPI_Bxor]	bit-wise xor
[MPI_Maxloc]	max value and location
[MPI_Minloc]	min value and location

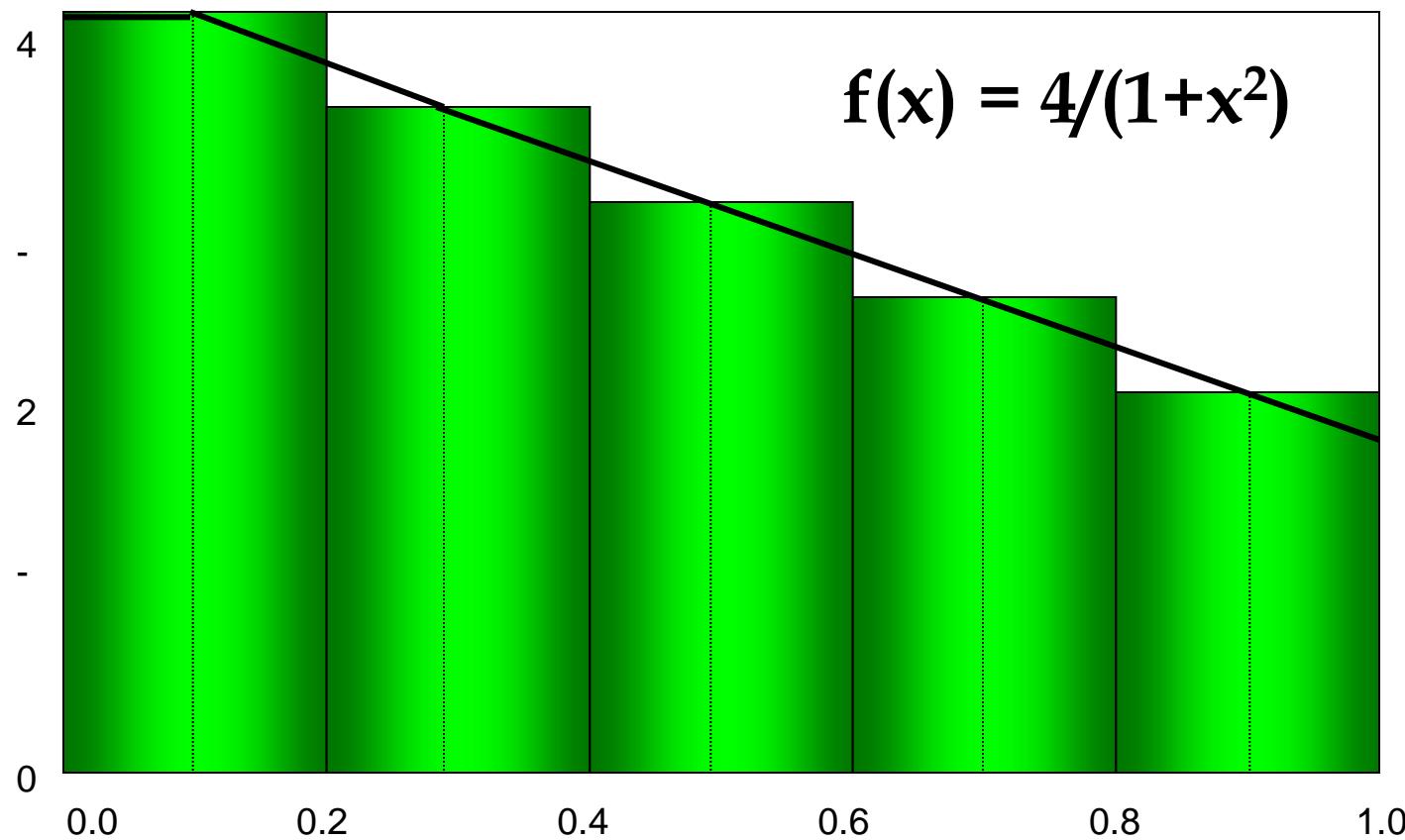
Example

Write a simple parallel program to calculate value of **pi** by numerical integration. Since

$$\int_0^1 \frac{1}{1+x^2} dx = \tan^{-1}(x) \Big|_0^1 = \tan^{-1}(1) - \tan^{-1}(0) = \tan^{-1}(1) = \frac{\pi}{4}$$

We will integrate the function $f(x) = 4/(1 + x^2)$

Graph



```
#include "mpi.h"
#include <math.h>
double f(a)
double a;
{
    return (4.0 / (1.0 + a*a));
}
int main(argc,argv)
int argc;
char *argv[];
{
    int n, myid, numprocs, i, rc;
    double PI25DT = 3.14159265;
    double mypi, pi, h, sum, x, a;
    double startwtime, endwtime;
```

```
MPI_Init(&argc,&argv) ;  
MPI_Comm_size(MPI_COMM_WORLD , &numprocs) ;  
MPI_Comm_rank(MPI_COMM_WORLD , &myid) ;  
if (myid == 0)  
{  
    printf("Enter the number of intervals:") ;  
    scanf("%d" , &n) ;  
    startwtime = MPI_Wtime() ;  
}
```

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);  
h    = 1.0 / (double) n;  
sum = 0.0;  
for (i = myid + 1; i <= n; i += numprocs)  
{  
    x = h * ((double)i - 0.5);  
    sum += f(x);  
}  
mypi = h * sum;
```

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,  
MPI_COMM_WORLD) ;  
  
    if (myid == 0)  
    {  
  
        printf("pi is approximately  
        %.16f, Error is %.16f\n",pi, fabs(pi  
        - PI25DT));  
  
        endwtime = MPI_Wtime();  
        printf("wall clock time = %f\n",  
              endwtime-startwtime);  
    }  
}  
}  
  
MPI_Finalize();  
}
```

```
#include "mpi.h"
#include <math.h>
double f(a)
double a;
{
    return (4.0 / (1.0 + a*a));
}
int main(argc,argv)
int argc;
char *argv[];
{
    int n, myid, numprocs, i, rc;
    double PI25DT = 3.14159265;
    double mypi, pi, h, sum, x, a;
    double startwtime, endwtime;
```

Some variable declarations:

myid, mypi, sum, x, i
are local to process.

n, numproc are global

```
MPI_Init(&argc,&argv) ;  
MPI_Comm_size(MPI_COMM_WORLD , &numprocs) ;  
MPI_Comm_rank(MPI_COMM_WORLD , &myid) ;  
if (myid == 0)  
{  
    printf("Enter the number of intervals:") ;  
    scanf("%d" , &n) ;  
    startwtime = MPI_Wtime() ;  
}
```

```

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

h    = 1.0 / (double) n;
sum = 0.0;

for (i = myid + 1; i <= n; i += numprocs)
{
    x = h * ((double)i - 0.5);
    sum += f(x);
}

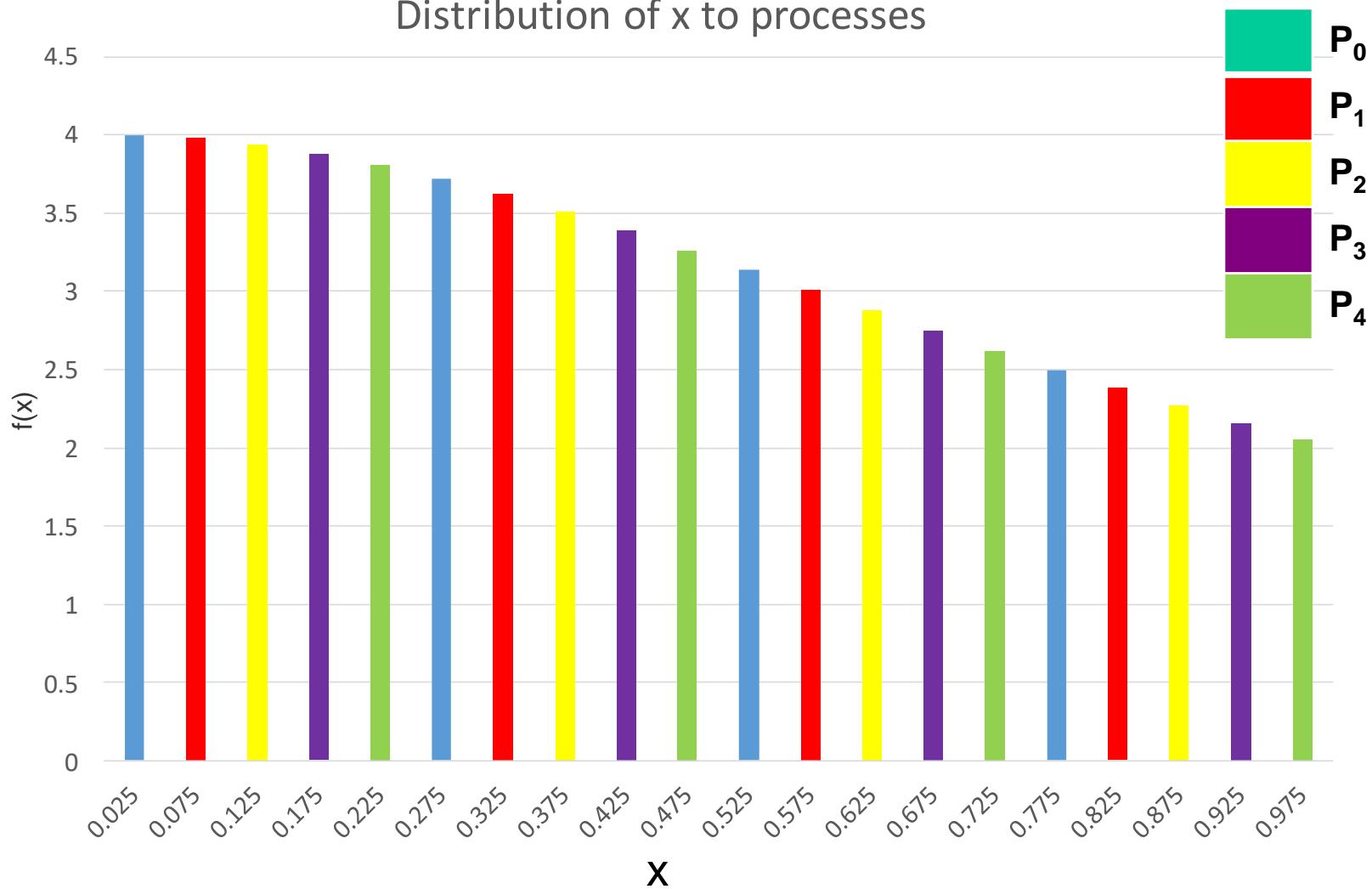
mypi = h * sum;      /area of the rectangle formula

```

Suppose $n = 20$, $numproc = 5$ then it will be broadcasted to all processes, $h = 1/20 = .05$

P ₀			P ₁			P ₂			P ₃			P ₄		
sum = 0.0														
i	x	f(x)												
1	0.025	3.99	2	0.075	3.98	3	0.125	3.93	4	0.175	3.88	5	0.225	3.81
6	0.275	3.71	7	0.325	3.62	8	0.375	3.50	9	0.425	3.39	10	0.475	3.26
11	0.525	3.13	12	0.575	3.01	13	0.625	2.87	14	0.675	2.75	15	0.725	2.62
16	0.775	2.49	17	0.825	2.38	18	0.875	2.26	19	0.925	2.16	20	0.975	2.05
sum = 13.35			sum = 12.99			sum = 12.58			sum = 12.18			sum = 11.74		

Distribution of x to processes



```

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD) ;

    if (myid == 0)

    {

        printf("pi is approximately
            %.16f, Error is %.16f\n",pi, fabs(pi
            - PI25DT)) ;

        endwtime = MPI_Wtime() ;
        printf("wall clock time = %f\n",
            endwtime-startwtime) ;

    }

}

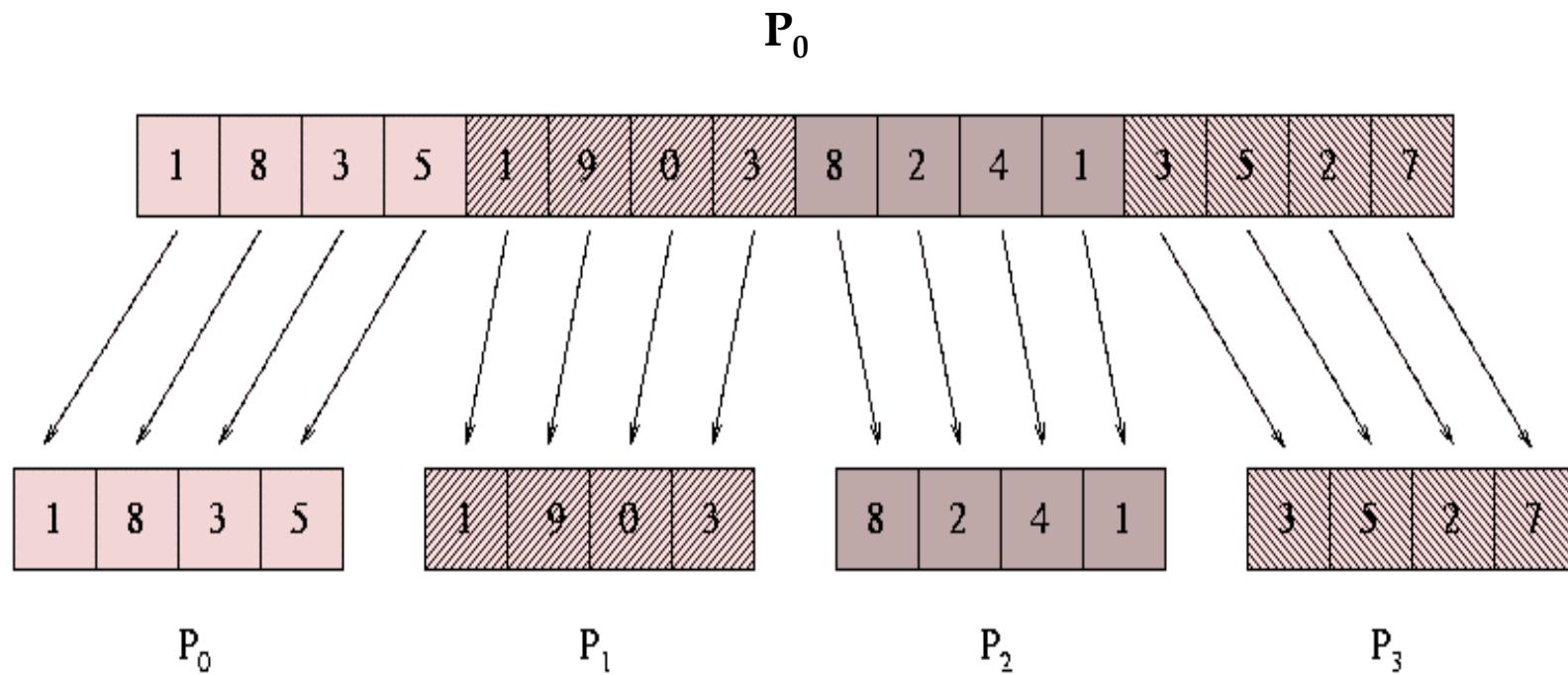
MPI_Finalize() ;

}

```

Simple Scatter Program

revisiting Scatter



Simple Scatter program

```
#include "mpi.h"

#include <stdio.h>

#include <stdlib.h>

#define SIZE 4

//on 14.139.69.97 program name is scatter.c

int main (int argc, char *argv[])

{

int numtasks, rank, sendcount, recvcount, source;

float sendbuf[SIZE][SIZE] = {

    {1.0, 2.0, 3.0, 4.0} ,

    {5.0, 6.0, 7.0, 8.0} ,

    {9.0, 10.0, 11.0, 12.0} ,

    {13.0, 14.0, 15.0, 16.0} };
```

Simple Scatter program

```
float recvbuf[SIZE];

MPI_Init(&argc,&argv);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {

    source = 1; //it can be any source

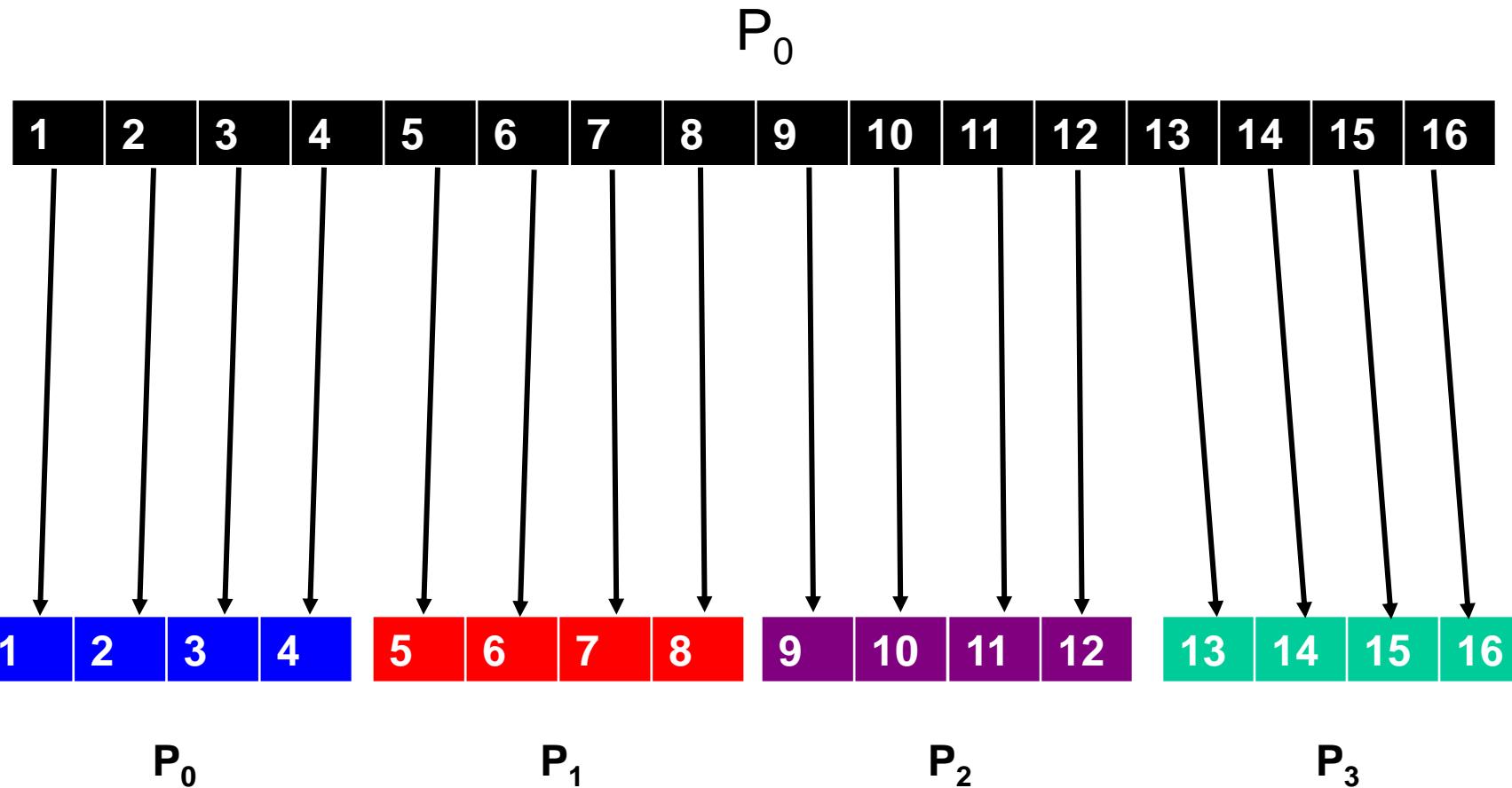
    sendcount = SIZE;

    recvcount = SIZE;

    MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,r
    ecvcount, MPI_FLOAT,source,MPI_COMM_WORLD);
```

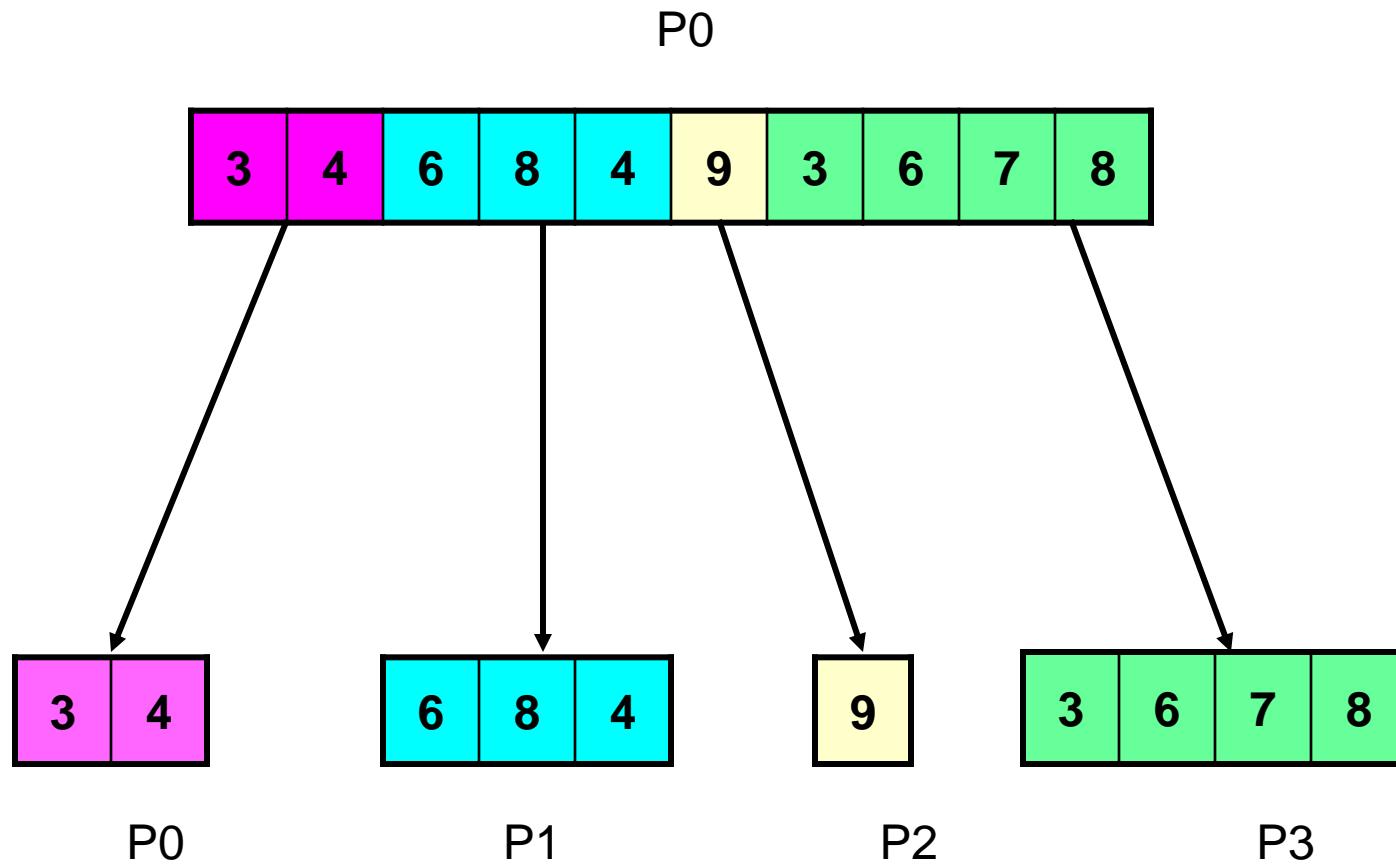
Simple Scatter program

```
printf("rank= %d  Results: %f %f %f  
%f\n",rank,recvbuf[0],  
recvbuf[1],recvbuf[2],recvbuf[3]);  
}  
  
else  
  
    printf("Must specify %d processors.  
Terminating.\n",SIZE);  
  
MPI_Finalize();  
}
```



For $n = 16$, $P = 4$

Scatterv Operation



MPI SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each processor
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf from which to take the outgoing data to process i)
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	recvcount	number of elements in receive buffer (integer)
IN	recvtype	data type of receive buffer elements
IN	root	rank of sending process (integer)
IN	comm	communicator

Header for MPI_Scatterv

```
int MPI_Scatterv (
    void           *send_buffer,
    int            *send_cnt,
    int            *send_disp,
    MPI_Datatype   send_type,
    void           *receive_buffer,
    int             receive_cnt,
    MPI_Datatype   receive_type,
    int             root,
    MPI_Comm       communicator)
```

Scatterv Example

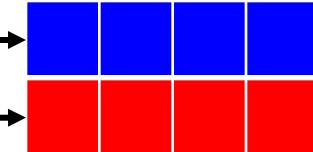
```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
//program name on hpc is scattervv.c
int main (int argc, char *argv[])
{
    int numproc, i, rank, recvcount, source;
    float sendbuf[SIZE][SIZE] = {
        {1.0, 2.0, 3.0, 4.0},
        {5.0, 6.0, 7.0, 8.0},
        {9.0, 10.0, 11.0, 12.0},
        {13.0, 14.0, 15.0, 16.0} };
```

1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0	16.0
-----	-----	-----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------	------

Scatterv Example

```
int sendcount[SIZE];
```

```
int displ[SIZE];
```



```
float recvbuf[SIZE*SIZE];
```

```
MPI_Init(&argc,&argv);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &numproc);
```

```
if (numproc == SIZE) {
```

```
    source = 0;
```

```
    for (i = 0; i < SIZE; i++) {
```

```
        sendcount[i] = rand() % (2*SIZE);
```

```
        displ[i] = rand() % (2*SIZE);
```

```
}
```



Scatterv Example

```
recvcount = SIZE*SIZE;
```

```
for (i = 0; i < SIZE*SIZE; i++)
```

```
    recvbuf[i] = 0; → 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
```

```
if (rank==0){
```

```
    for (i = 0; i < SIZE; i++){
```

```
        printf ("sendcount[%d] = %d, displ[%d] = %d\n", i, sendcount[i], i,  
displ[i]);
```

```
}
```

```
}
```

```
printf("\n");
```



Scatterv Example

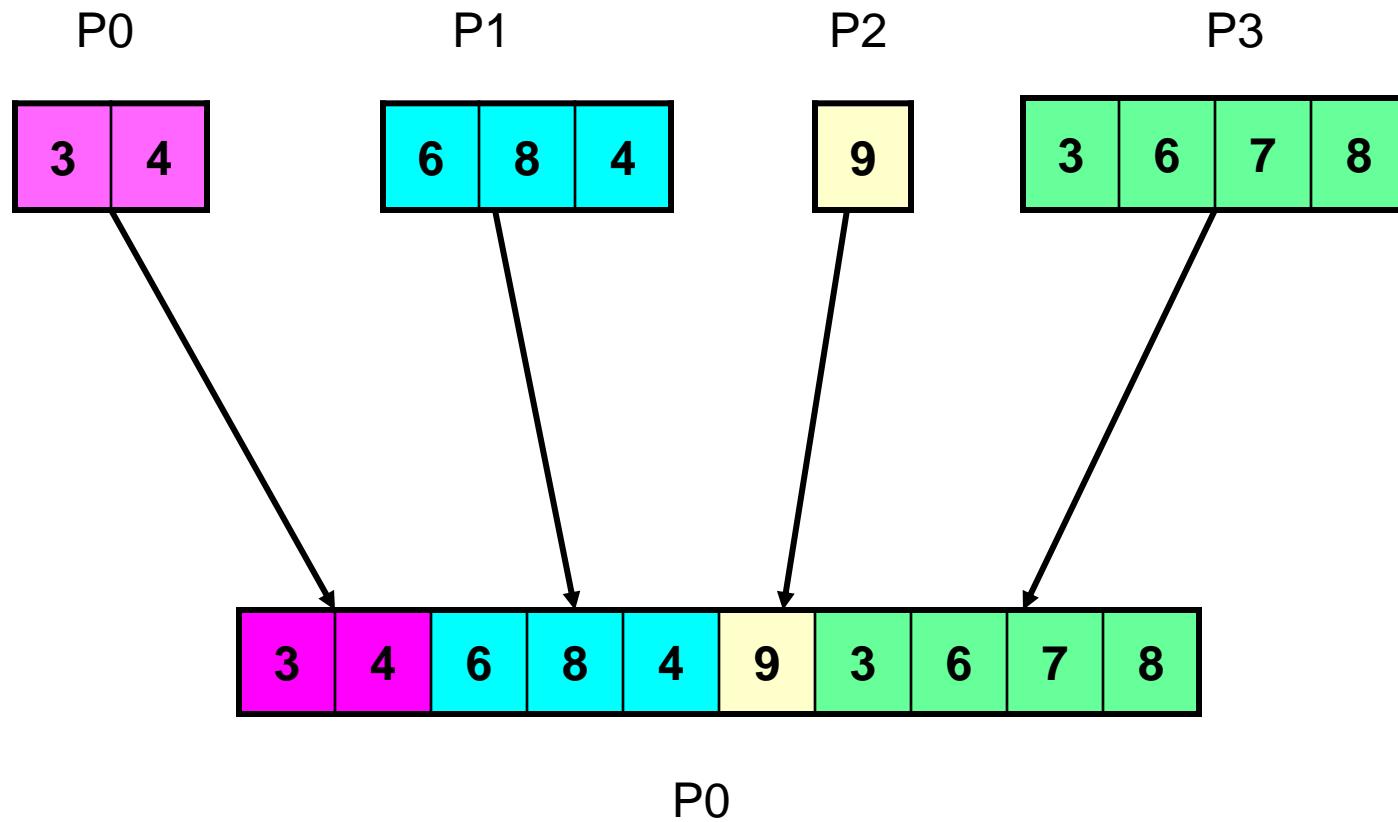
```
MPI_Scatterv(sendbuf,sendcount,displ,  
MPI_FLOAT,recvbuf,recvcount, MPI_FLOAT,source,  
MPI_COMM_WORLD);
```

```
printf("rank= %d\n", rank);  
for (i=0; i<SIZE*SIZE; i++){  
    printf("%.1f ",recvbuf[i]);  
}  
  
    7.0 | 8.0 | 9.0 | 10.0 | 11.0 | 12.0 | 13.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  
        4.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  
    8.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  
        5.0 | 6.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  
printf("\n");  
}  
else  
printf("Must specify %d processors. Terminating.\n",SIZE);  
MPI_Finalize();  
}
```

Steps

- Download Putty software, free
- Connect to 14.139.69.93
- **ssh** port number **8223** and **not 22**
- Use any editor (Ex. **vi or nano**) to write c program
- mpicc -o scattervv scattervv.c
- mpirun -np 4 scattervv

Gatherv Operation

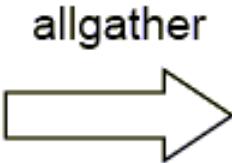


Header for MPI_Gatherv

```
int MPI_Gatherv (  
    void *send_buffer,  
    int send_cnt,  
    MPI_Datatype send_type,  
    void *receive_buffer,  
    int receive_cnt,  
    int receive_disp,  
    MPI_Datatype receive_type,  
    int root,  
    MPI_Comm communicator)
```

Allgather

A ₀					
B ₀					
C ₀					
D ₀					
E ₀					
F ₀					

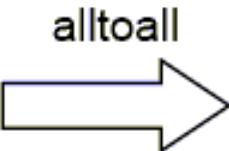


A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₀	B ₀	C ₀	D ₀	E ₀	F ₀

Gathers data from all tasks and distribute it to all

Alltoall

A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
B ₀	B ₁	B ₂	B ₃	B ₄	B ₅
C ₀	C ₁	C ₂	C ₃	C ₄	C ₅
D ₀	D ₁	D ₂	D ₃	D ₄	D ₅
E ₀	E ₁	E ₂	E ₃	E ₄	E ₅
F ₀	F ₁	F ₂	F ₃	F ₄	F ₅



A ₀	B ₀	C ₀	D ₀	E ₀	F ₀
A ₁	B ₁	C ₁	D ₁	E ₁	F ₁
A ₂	B ₂	C ₂	D ₂	E ₂	F ₂
A ₃	B ₃	C ₃	D ₃	E ₃	F ₃
A ₄	B ₄	C ₄	D ₄	E ₄	F ₄
A ₅	B ₅	C ₅	D ₅	E ₅	F ₅

- Sends data from all to all processes
- Useful in getting transpose of a matrix

MPI_Alltoall

```
int MPI_Alltoall( void *sendbuf, int sendcount,  
MPI_Datatype sendtype, void *recvbuf, int recvcnt,  
MPI_Datatype recvtype, MPI_Comm comm )
```

Input Parameters

sendbuf

starting address of send buffer
number of elements to send to
each process (integer)

sendtype

data type of send buffer elements
number of elements received from
any process (integer)

recvcount

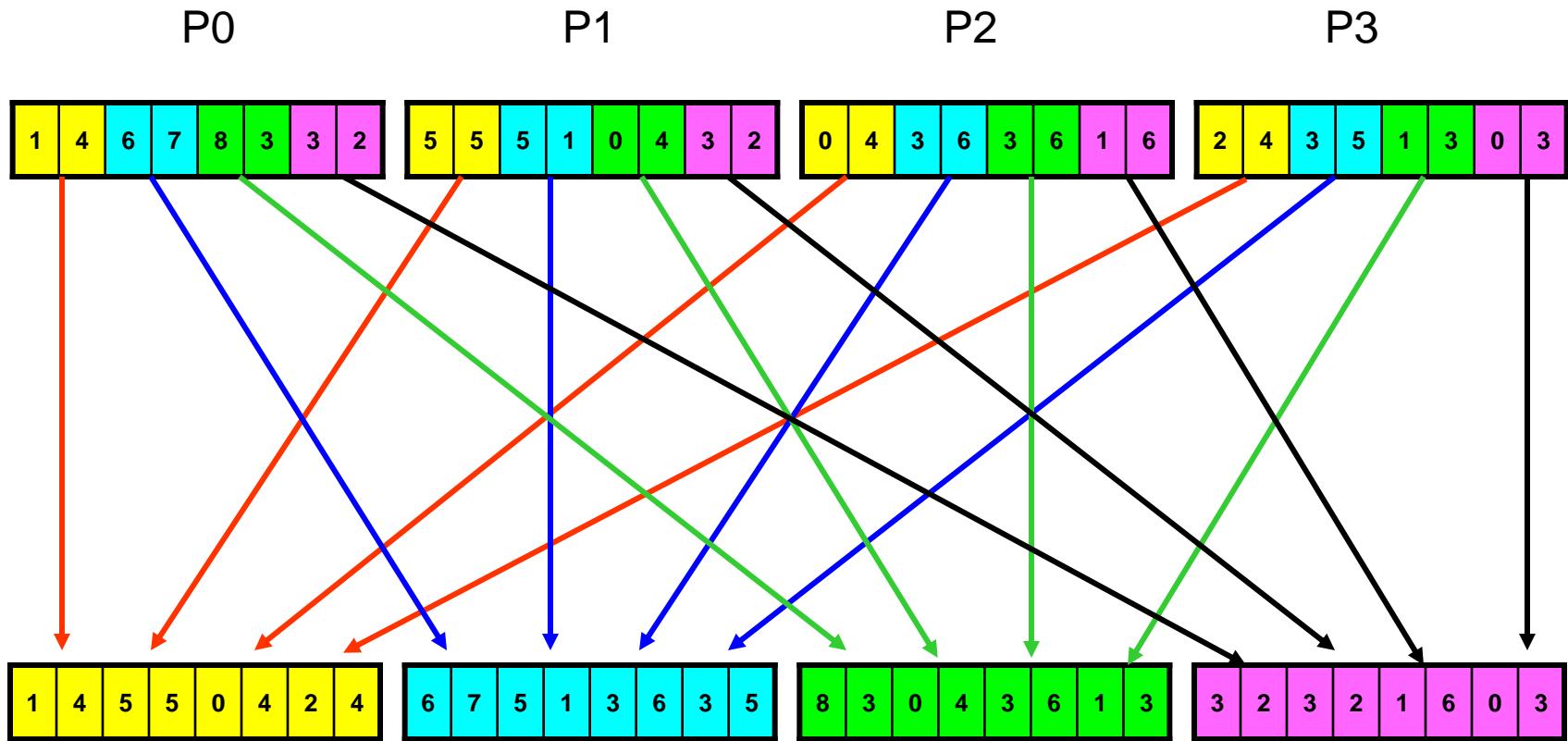
data type of receive buffer
elements

recvtype

communicator

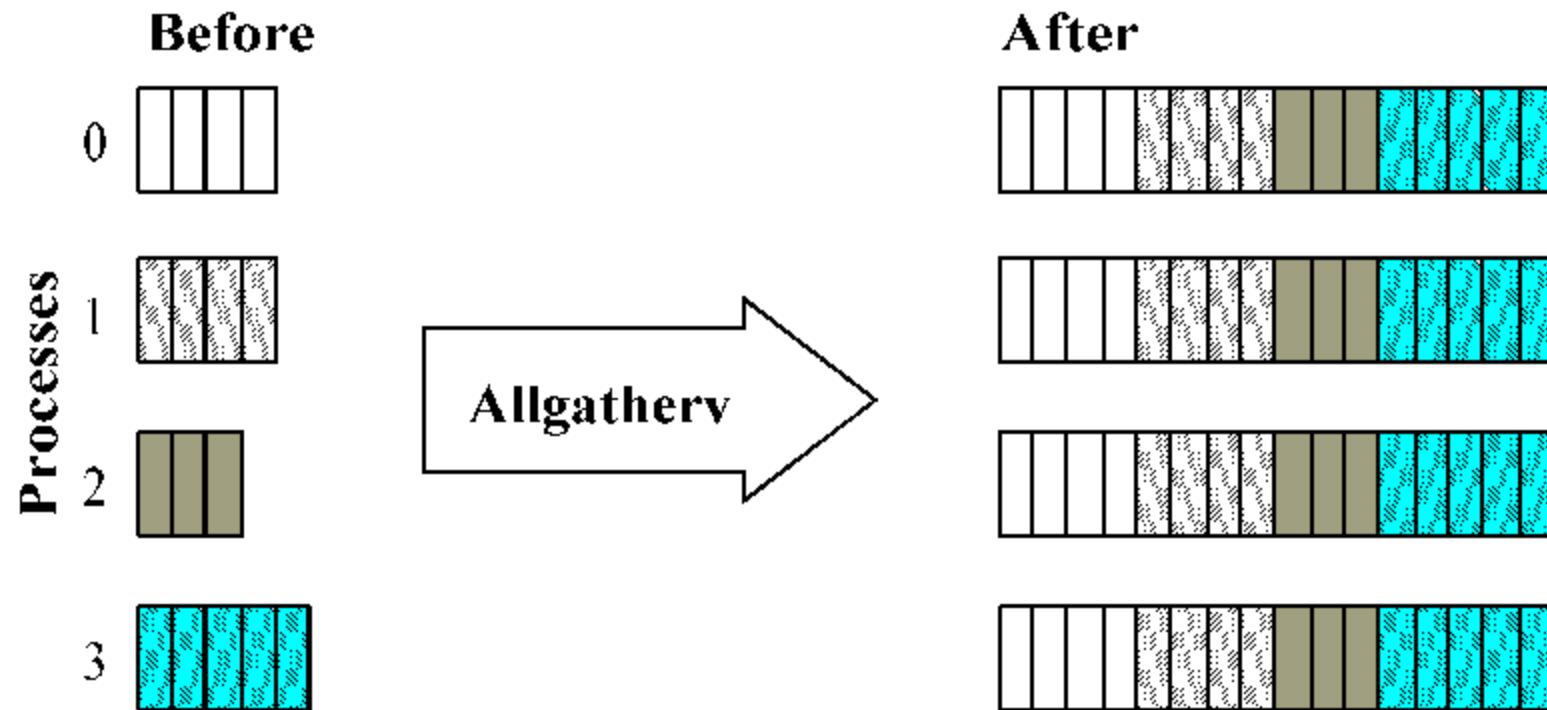
comm

Alltoall Operation



Sends data from all to all processes
All-to-All operation for an integer array of size 8 on 4 processors

MPI_Allgatherv



MPI_Allgatherv

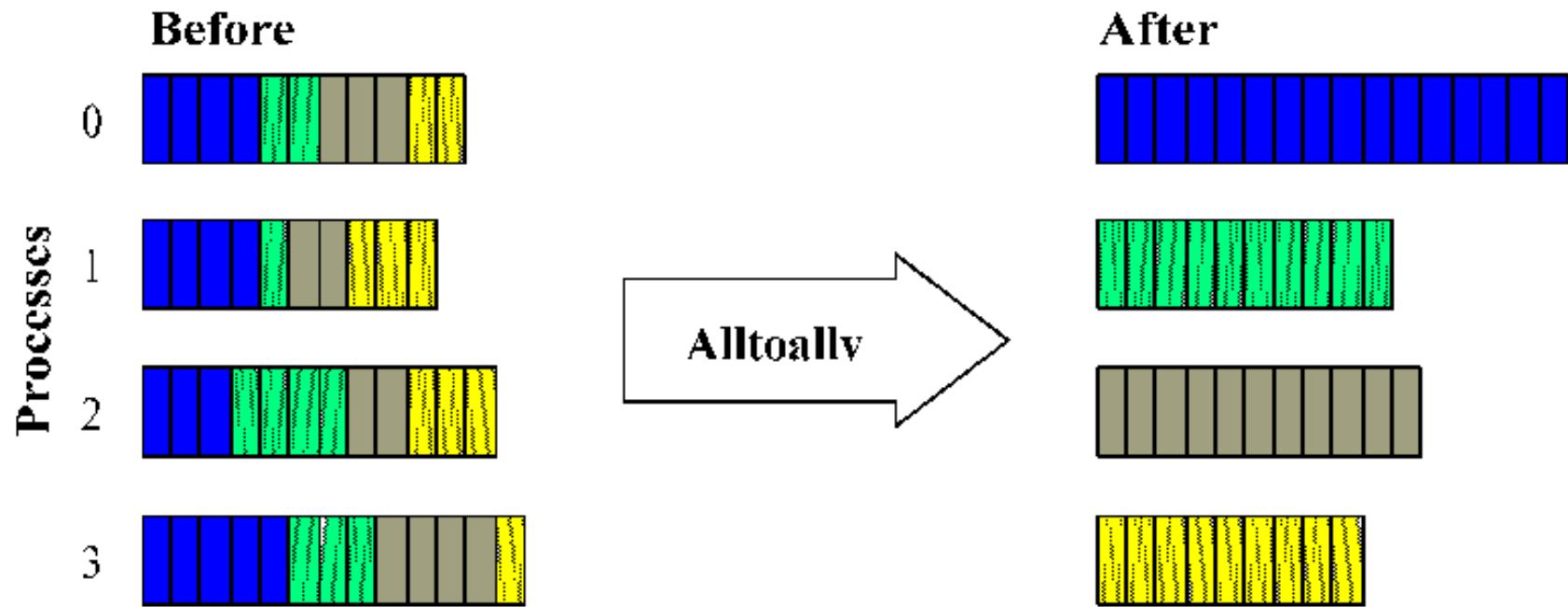
```
int MPI_Allgatherv (
    void                 *send_buffer,
    int                  send_cnt,
    MPI_Datatype        send_type,
    void                *receive_buffer,
    *receive_cnt,
    *receive_disp,
    MPI_Datatype        receive_type,
    MPI_Comm            communicator)
```

Input Parameters (MPI_Allgatherv)

sendbuf	starting address of send buffer (choice)
sendcount	number of elements in send buffer (integer)
sendtype	data type of send buffer elements
recvcounts	integer array containing the number of elements that are received from each process
displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
recvtype	data type of receive buffer elements
comm	communicator
recvbuf	address of receive buffer (choice)

The block of data sent from the jth process is received by every process and placed in the jth block of the buffer recvbuf.

Function MPI_Alltoally



Header for MPI_Alltoallv

```
int MPI_Gatherv (  
    void *send_buffer,  
    int send_cnt,  
    int send_disp,  
    MPI_Datatype send_type,  
    void *receive_buffer,  
    int receive_cnt,  
    int receive_disp,  
    MPI_Datatype receive_type,  
    MPI_Comm communicator)
```

Matrix-vector Multiplication

Outline

- At least three parallel implementation are possible
 - Rowwise block striped
 - Columnwise block striped
 - Block decomposition

Storing Vectors

- Divide vector elements among processes
- Replicate vector elements
- Vector replication acceptable because vectors have only n elements, versus n^2 elements in matrices

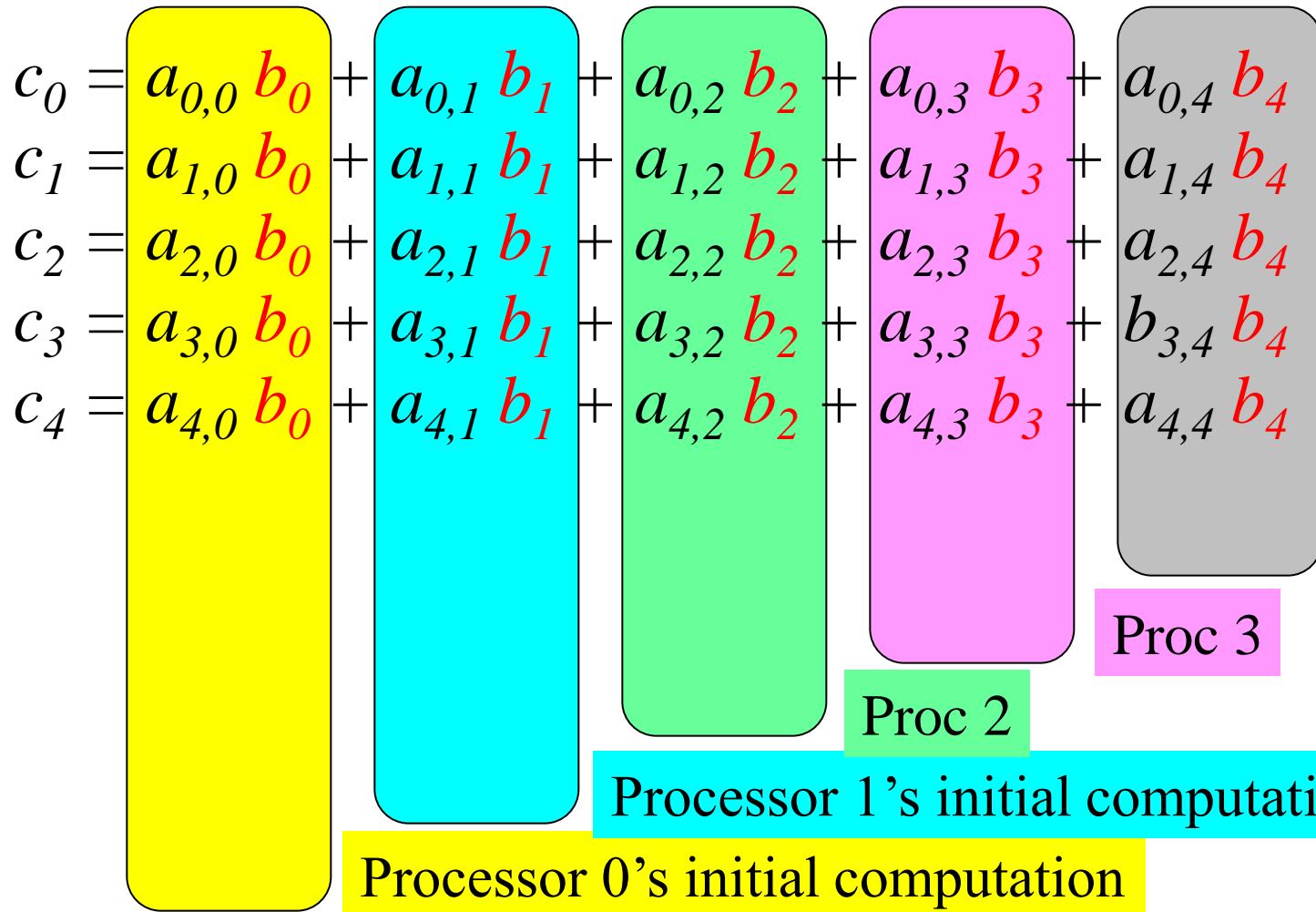
Rowwise Block Striped Matrix

- Partitioning through domain decomposition
- Primitive task associated with
 - Row of matrix
 - Entire vector

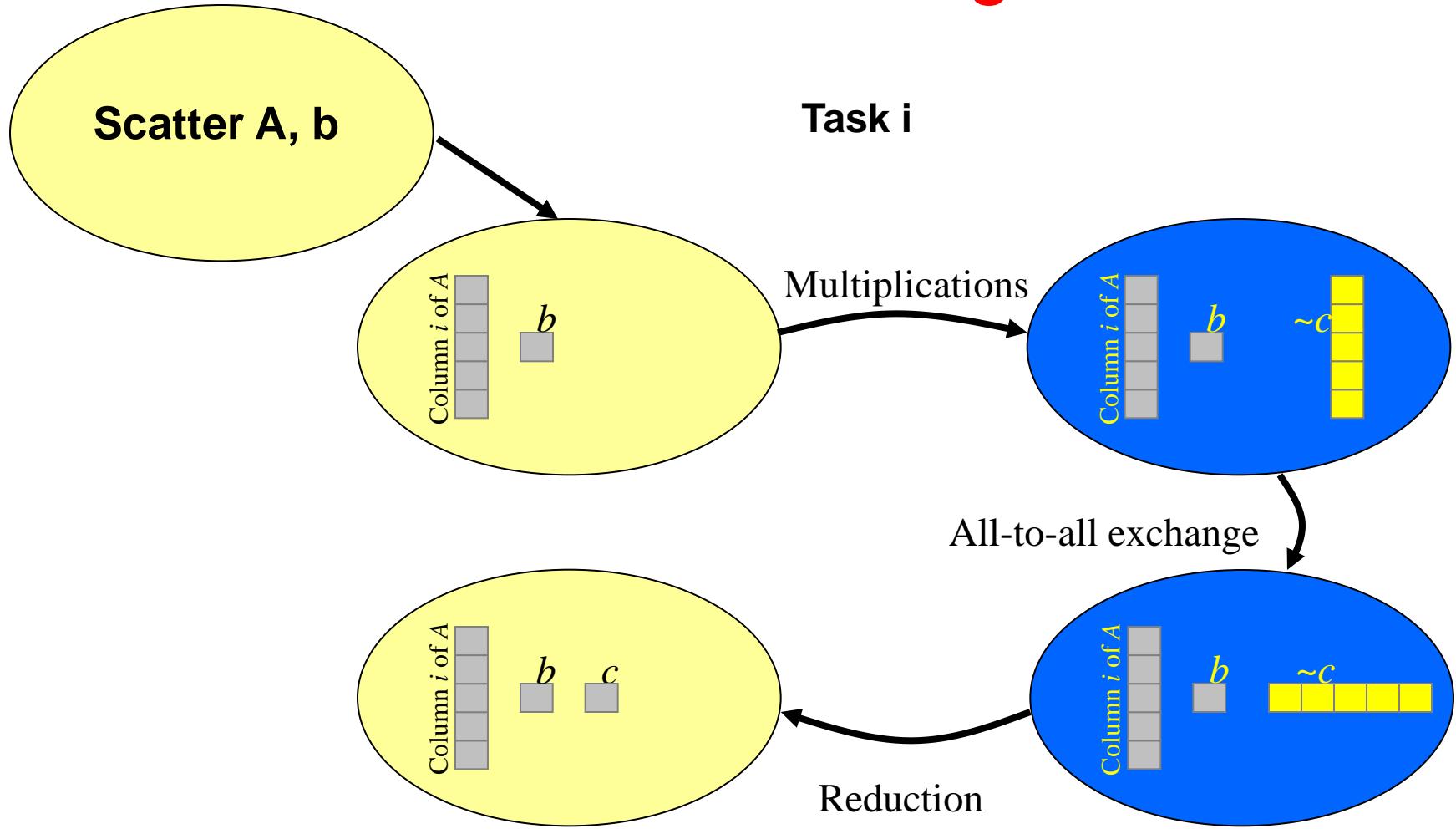
Columnwise Block Striped Matrix

- Partitioning through domain decomposition
- Task associated with
 - Column of matrix
 - Vector element
- MPICH function used are MPI_Scatter, MPI_Gather, MPI_alltoall

Matrix-Vector Multiplication



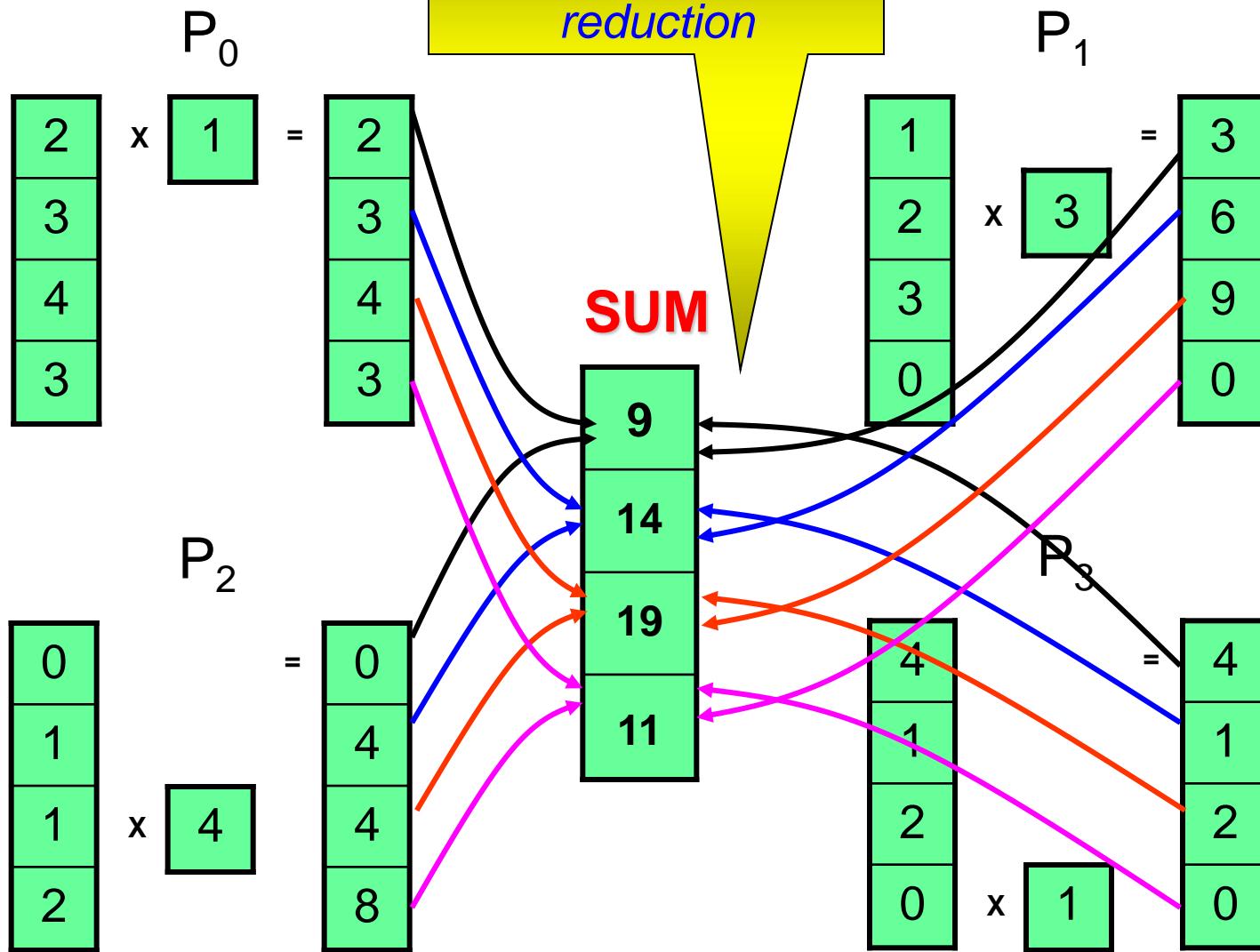
Phases of Parallel Algorithm



Matrix-Vector

$$\begin{matrix} & \begin{matrix} 2 & 1 & 0 & 4 \\ 3 & 2 & 1 & 1 \\ 4 & 3 & 1 & 2 \\ 3 & 0 & 2 & 0 \end{matrix} & \times & \begin{matrix} 1 \\ 3 \\ 4 \\ 1 \end{matrix} \end{matrix}$$

This is *alltoall*
operation with
reduction



Evaluating Programs Empirically

Measuring Execution Time

To measure the execution time between point L1 and point L2 in the code, we might have a construction such as.

```
L1:    time(&t1); /* start timer */..
```

```
L2:    time(&t2); /* stop timer */.
```

```
elapsed_time = difftime(t2, t1); /* elapsed_time = t2 - t1 */
```

```
printf("Elapsed time = %5.2f seconds", elapsed_time);
```

MPICH provides the routine [MPI_Wtime\(\)](#) for returning time (in seconds).

References

- William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.
- William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd edition. MIT Press, Cambridge, MA, 1999.
- William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

References

- Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.
- Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufman, 1997.
- Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI—The Complete Reference: Volume 1, The MPI Core*, 2nd edition. MIT Press, Cambridge, MA, 1998.
- <http://www.mcs.anl.gov/mpi/mpich>

References

- William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd edition. MIT Press, Cambridge, MA, 1999.
- Michael J Quinn. *Parallel Programming in C with MPI and OpenMP*, Tata-McGraw-Hill Edition, 2003.
- Barry Wilkinson And Michael Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, Upper Saddle River, NJ, 1999.

Practical Techniques & Examples

Main Source: M. J. Quinn's book and Berry Wilkinson's book

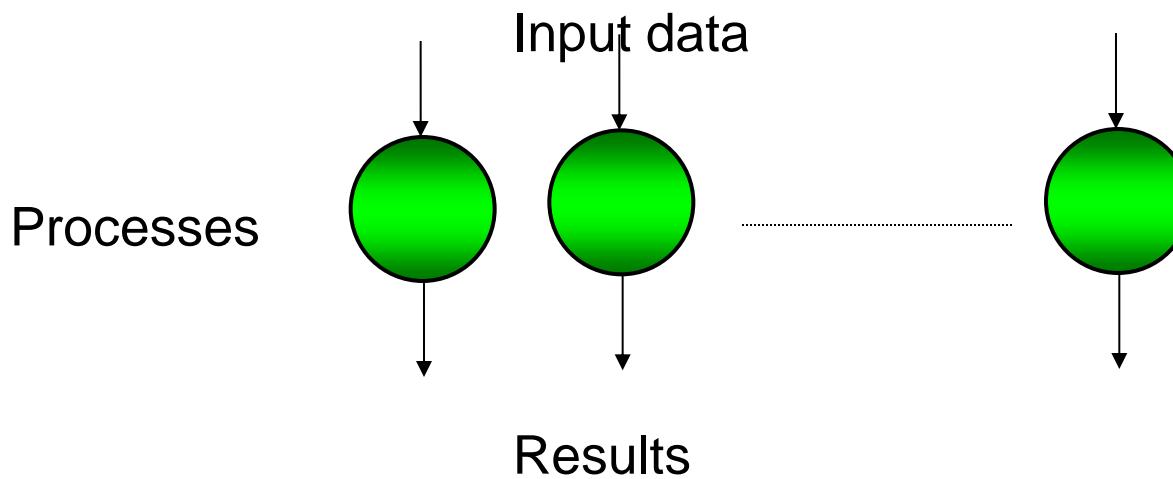
Parallel Techniques

- Embarassingly Parallel Computations
- Partitioning and Divide-and-Conquer Strategies
- Pipelined Computations
- Synchronous Computations
- Asynchronous Computations
- Load Balancing and Termination Detection

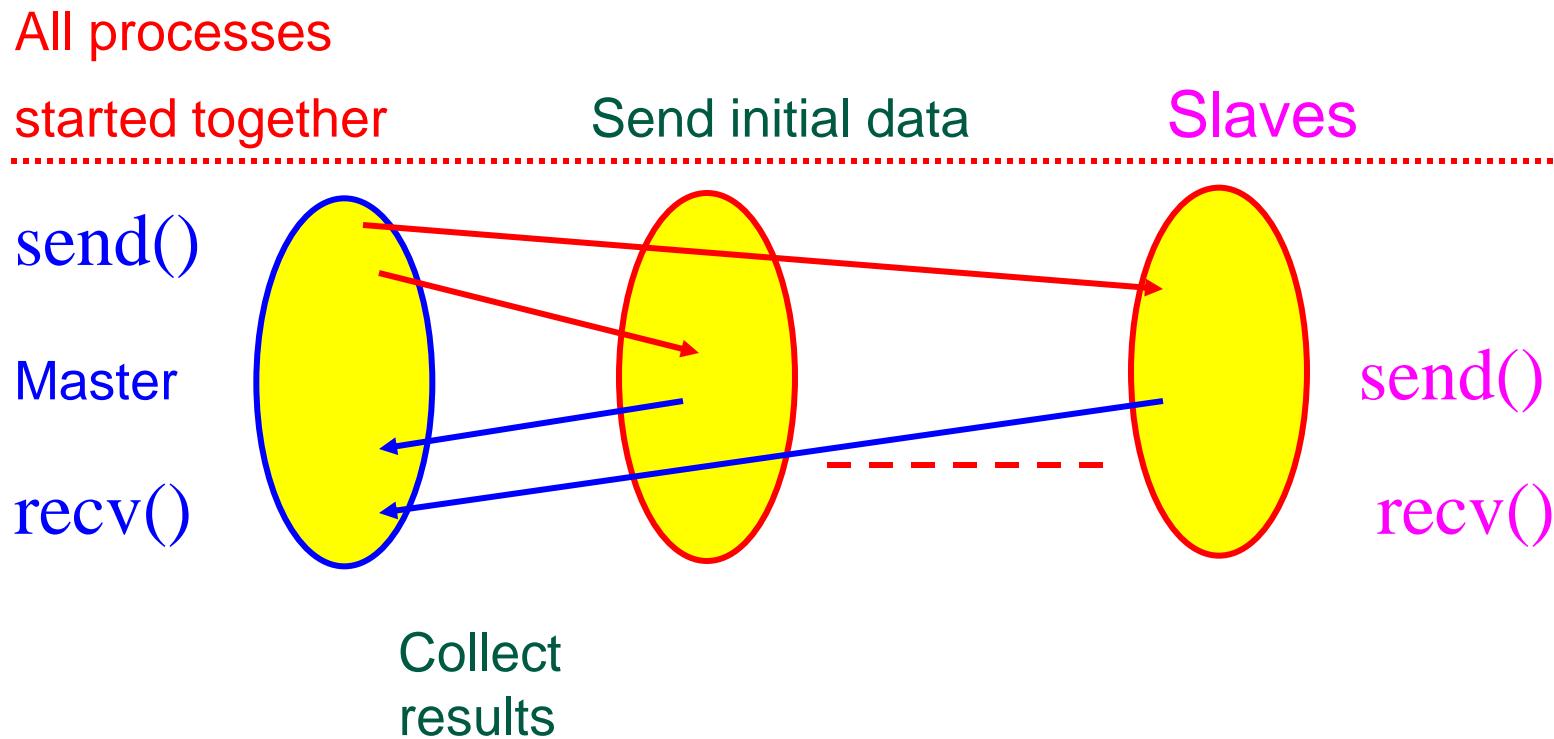
Embarrassingly Parallel Computations

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

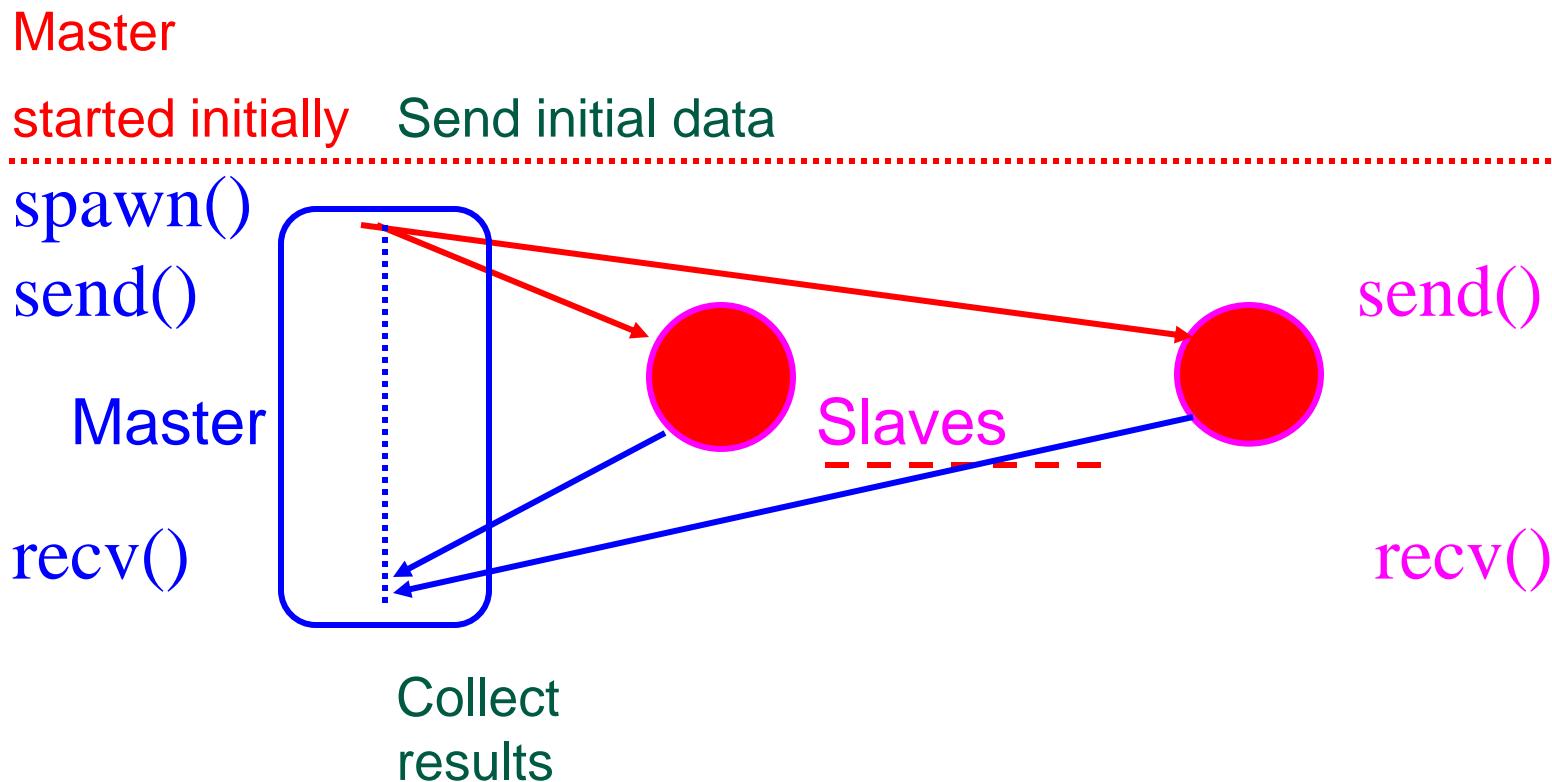
No communication or very little communication between processes



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



Practical embarrassingly parallel computation with **dynamic** process creation and master slave **PVM** approach



Examples

- Circuit Satisfiability
- Low level image processing (assignment)
- Mandelbrot set
- Monte Carlo Calculations

MPI programs for Circuit Satisfiability

Case Study

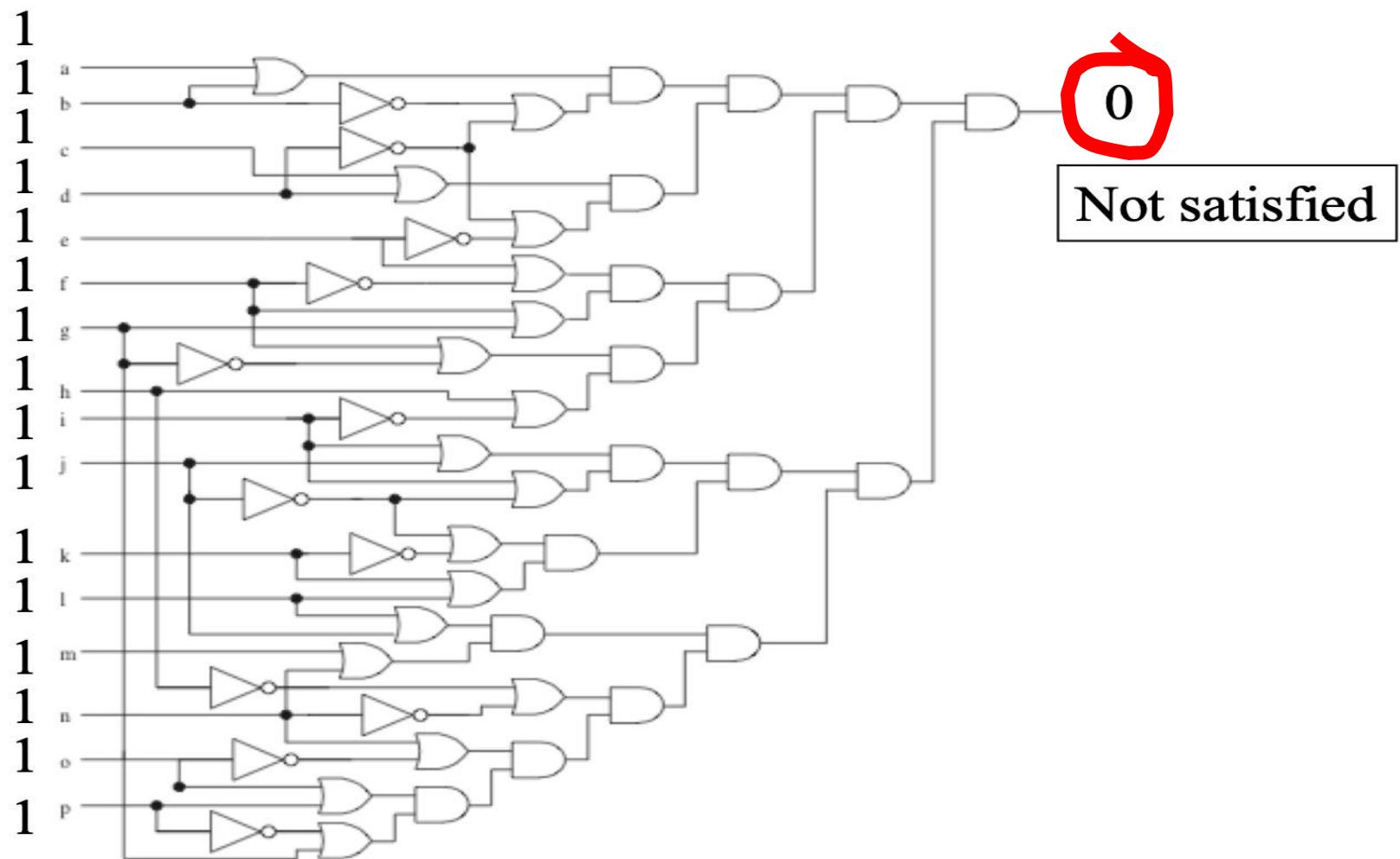
Outline

- Coding MPI programs
- Compiling MPI programs
- Running MPI programs
- Benchmarking MPI programs

Processes

- Number is specified at start-up time
- Remains constant throughout execution of program
- All execute same program
- Each has unique ID number
- Alternately performs computations and communicates

Circuit Satisfiability



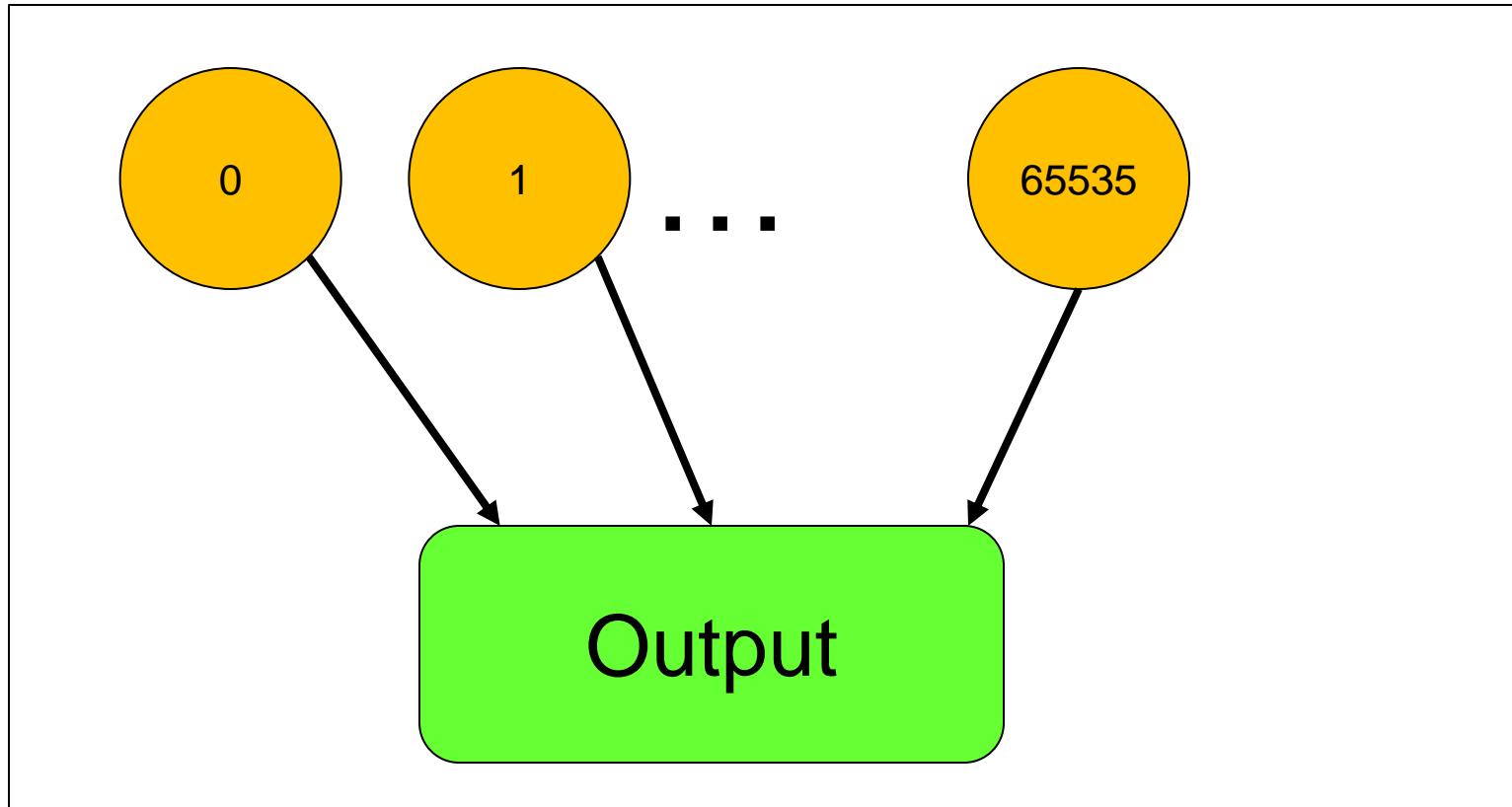
Equation in Boolean Normal form (CNF)

$$F = (v_0 \vee v_1) \wedge (\neg v_1 \vee \neg v_3) \wedge (v_2 \vee v_3) \wedge (\neg v_3 \vee \neg v_4) \wedge (v_4 \vee \neg v_5) \wedge (v_5 \vee \neg v_6) \wedge (v_5 \vee v_6) \wedge (v_6 \vee \neg v_{15}) \wedge (v_7 \vee \neg v_8) \wedge (\neg v_7 \vee \neg v_{13}) \wedge (v_8 \vee v_9) \wedge (v_8 \vee \neg v_9) \wedge (\neg v_9 \vee \neg v_{10}) \wedge (v_9 \vee v_{11}) \wedge (v_{10} \vee v_{11}) \wedge (v_{12} \vee v_{13}) \wedge (v_{13} \vee \neg v_{14}) \wedge (v_{14} \vee v_{15})$$

Solution Method

- Circuit satisfiability is **NP-complete**
- No known algorithms to solve in polynomial time
- We seek all solutions
- We find through exhaustive search
- 16 inputs \Rightarrow 65,536 combinations to test

Partitioning: Functional Decomposition



- **Embarrassingly parallel:** No channels between tasks

Properties and Mapping

- Properties of parallel algorithm
 - Fixed number of tasks
 - No communications between tasks
 - Time needed per task is variable
- Mapping strategy
 - decision tree
 - Map tasks to processors in a cyclic fashion

Cyclic (interleaved) Allocation

- Assume p processes
- Each process gets every p^{th} piece of work
- Example: 5 processes and 12 pieces of work
 - P_0 : 0, 5, 10
 - P_1 : 1, 6, 11
 - P_2 : 2, 7
 - P_3 : 3, 8
 - P_4 : 4, 9

Program Design

- Program will consider all 65,536 combinations of 16 boolean inputs
- Combinations allocated in cyclic fashion to processes
- Each process examines each of its combinations
- If it finds a satisfiable combination, it will print it

Include Files

```
#include <mpi.h>
```

- MPI header file

```
#include <stdio.h>
```

Local Variables

```
int main (int argc, char *argv[]) {  
    int i;  
    int id; /* Process rank */  
    int p; /* Number of processes */  
    void check_circuit (int, int);
```

- One copy of every variable for each process running this program

Initialize MPI

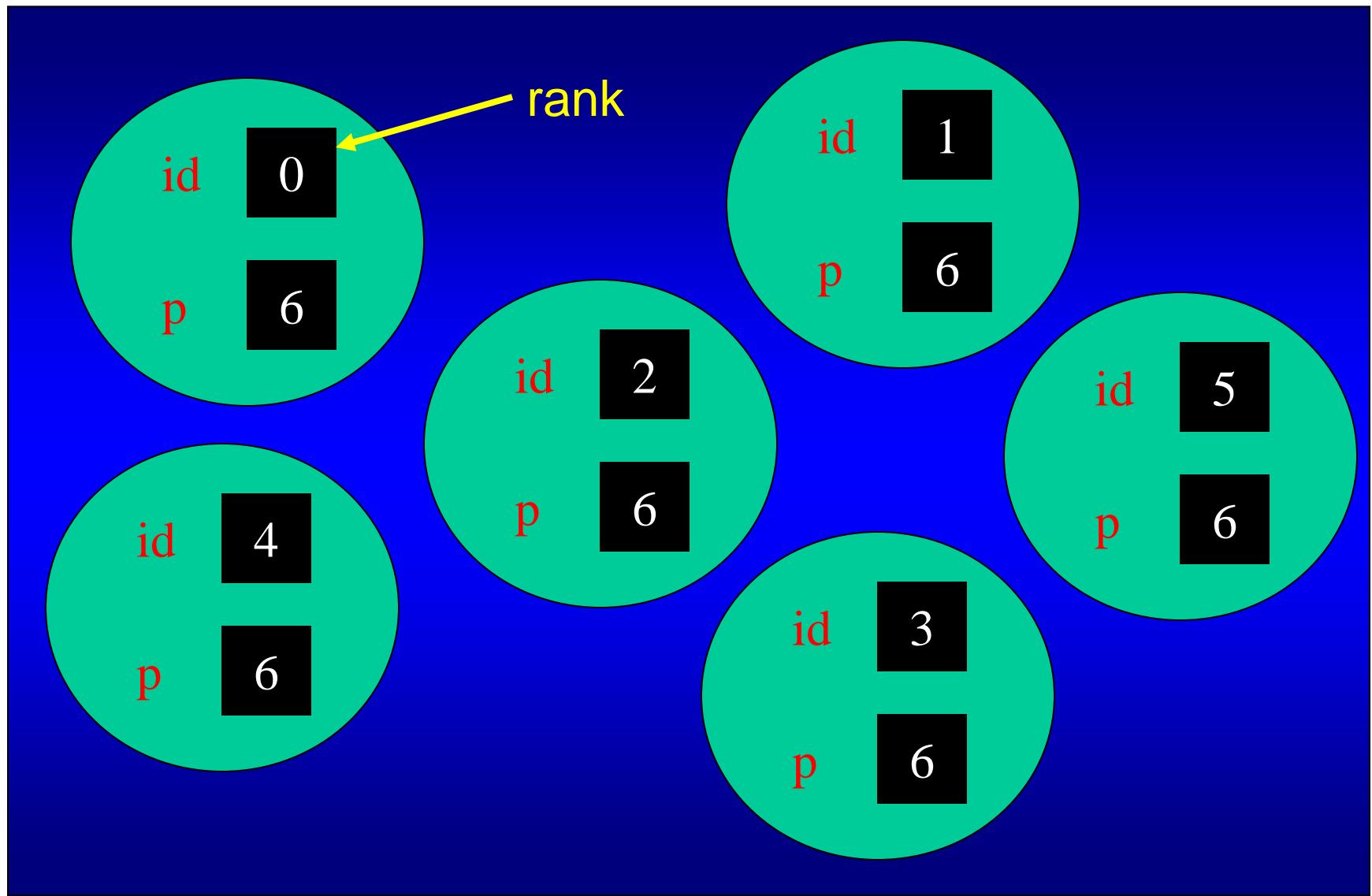
```
MPI_Init(&argc, &argv) ;
```

Determine Number of Processes and Rank

```
MPI_Comm_size(MPI_COMM_WORLD, &p) ;
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &id) ;
```

Replication of Automatic Variables



What about External Variables?

```
int total;
```

```
int main (int argc, char *argv[]) {  
    int i;  
    int id;  
    int p;  
    ...
```

- Where is variable **total** stored?

Cyclic Allocation of Work

```
for (i = id; i < 65536; i += p)
    check_circuit (id, i);
```

- Parallelism is outside function **check_circuit**
- It can be an ordinary, sequential function

Shutting Down MPI

```
MPI_Finalize();
```

- Call after all other MPI library calls
- Allows system to free up MPI resources

```
#include <mpi.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i;
    int id;
    int p;
    void check_circuit (int, int);

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);

    for (i = id; i < 65536; i += p)
        check_circuit (id, i);

    printf ("Process %d is done\n", id);
    fflush (stdout);
    MPI_Finalize();
    return 0;
}
```

Put fflush() after every printf() if you prefer not to wait until the buffer fills up (8Kb buffers are common) to see what has been happening

```

/* Return 1 if 'i'th bit of 'n' is 1; 0 otherwise */
#define EXTRACT_BIT(n,i) ((n&(1<<i))?1:0)

void check_circuit (int id, int z) {
    int v[16];           /* Each element is a bit of z */
    int i;

    for (i = 0; i < 16; i++) v[i] = EXTRACT_BIT(z,i);

    if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
        && (!v[3] || !v[4]) && (v[4] || !v[5])
        && (v[5] || !v[6]) && (v[5] || v[6])
        && (v[6] || !v[15]) && (v[7] || !v[8])
        && (!v[7] || !v[13]) && (v[8] || v[9])
        && (v[8] || !v[9]) && (!v[9] || !v[10])
        && (v[9] || v[11]) && (v[10] || v[11])
        && (v[12] || v[13]) && (v[13] || !v[14])
        && (v[14] || v[15])) {
        printf ("%d %d%d%d%d%d%d%d%d%d%d%d%d%d\n", id,
                v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8],v[9],
                v[10],v[11],v[12],v[13],v[14],v[15]);
        fflush (stdout);
    }
}

```

```
#define EXTRACT_BIT(n,i) ((n&(1<<i))?1:0)
```

//This macro **extracts the i-th bit** of number n.

- $(1 << i)$ shifts the number 1(binary representation of 1) left by i bits.
 - e.g., $1 << 3 = 00001000$ (binary)
 - $n \& (1 << i)$ performs **bitwise AND**:
 - If the i -th bit of n is 1, the result is nonzero.
 - Otherwise, result is 0.
 - $? 1 : 0$ converts that into a **1 or 0**.

EXTRACT_BIT(13,1)

Compiling MPI Programs

```
mpicc -O -o cnf cnf.c
```

- **mpicc**: script to compile and link C + MPI programs
- Flags: same meaning as C compiler
 - **-O** — optimize
 - **-o <file>** — where to put executable

Running MPI Programs

- **mpirun -np <p> <exec> <arg1> ...**
 - **-np <p>** — number of processes
 - **<exec>** — executable
 - **<arg1> ...** — command-line arguments

Execution on 1 CPU

```
% mpirun -np 1 sat
0) 101011110011001
0) 011011110011001
0) 111011110011001
0) 101011111011001
0) 011011111011001
0) 111011111011001
0) 1010111110111001
0) 0110111110111001
0) 1110111110111001
Process 0 is done
```

Execution on 2 CPUs

```
% mpirun -np 2 sat
0) 0110111110011001
0) 0110111111011001
0) 0110111110111001
1) 1010111110011001
1) 1110111110011001
1) 101011111011001
1) 111011111011001
1) 1010111110111001
1) 1110111110111001
Process 0 is done
Process 1 is done
```

Execution on 3 CPUs

```
% mpirun -np 3 sat
0) 0110111110011001
0) 1110111111011001
2) 1010111110011001
1) 1110111110011001
1) 1010111111011001
1) 0110111110111001
0) 1010111110111001
2) 0110111111011001
2) 11101111110111001
Process 1 is done
Process 2 is done
Process 0 is done
```

Interpreting Output

- Output order only partially reflects order of output events inside parallel computer
- If process A prints two messages, first message will appear before second
- If process A calls `printf` before process B, there is no guarantee process A's message will appear before process B's message

Enhancing the Program (Phase parallel model)

- We want to find total number of solutions
- Incorporate sum-reduction into program
- Reduction is a **collective communication**

Modifications

- Modify function `check_circuit`
 - Return 1 if circuit satisfiable with input combination
 - Return 0 otherwise
- Each process keeps local count of satisfiable circuits it has found
- Perform reduction after `for` loop

New Declarations and Code

```
int count; /* Local sum */  
int global_count; /* Global sum */  
int check_circuit (int, int);  
  
count = 0;  
for (i = id; i < 65536; i += p)  
    count += check_circuit (id, i);
```

Prototype of **MPI_Reduce()**

```
int MPI_Reduce (
    void          *operand,
                  /* addr of 1st reduction element */
    void          *result,
                  /* addr of 1st reduction result */
    int           count,
                  /* reductions to perform */
    MPI_Datatype type,
                  /* type of elements */
    MPI_Op        operator,
                  /* reduction operator */
    int           root,
                  /* process getting result(s) */
    MPI_Comm      comm
                  /* communicator */
)
```

Our Call to **MPI_Reduce ()**

```
MPI_Reduce (&count,  
            &global_count,  
            1,  
            MPI_INT,  
            MPI_SUM,  
            0,  
            MPI_COMM_WORLD) ;
```

Only process '0'
will get the result

```
if (!id) printf ("There are %d different solutions\n",  
    global_count);
```

Execution of Second Program

```
% mpirun -np 3 seq2
0) 0110111110011001
0) 1110111111011001
1) 1110111110011001
1) 1010111111011001
2) 1010111110011001
2) 0110111111011001
2) 11101111110111001
1) 01101111110111001
0) 10101111110111001
Process 1 is done
Process 2 is done
Process 0 is done
There are 9 different solutions
```

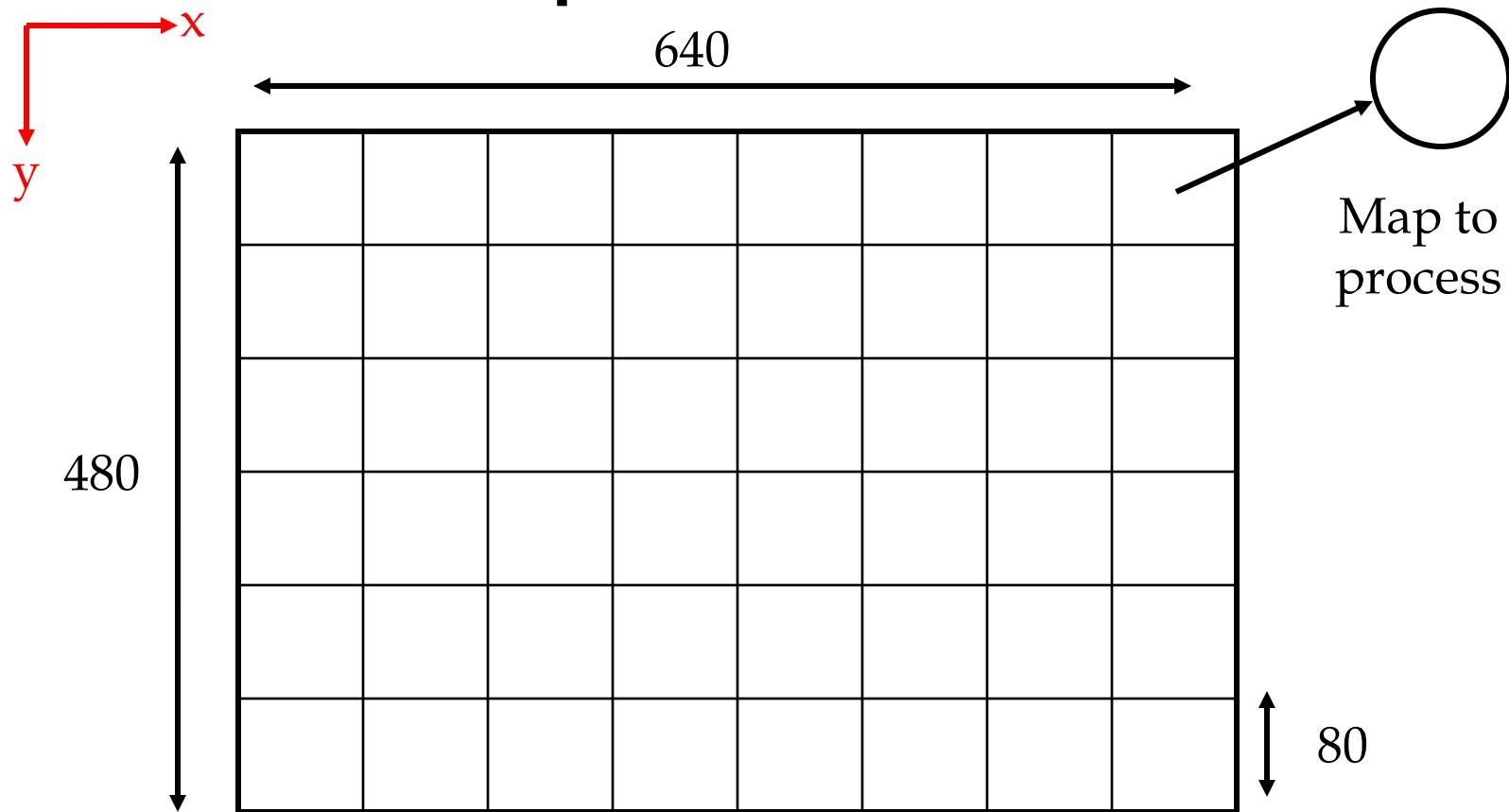
Benchmarking the Program

- **`MPI_Barrier`** — barrier synchronization
- **`MPI_Wtick`** — timer resolution
- **`MPI_Wtime`** — current time

Benchmarking Code

```
double elapsed_time;  
...  
MPI_Init (&argc, &argv);  
MPI_Barrier (MPI_COMM_WORLD);  
elapsed_time = - MPI_Wtime();  
...  
MPI_Reduce (...);  
elapsed_time += MPI_Wtime();
```

Partitioning into regions for individual processes.



Square region for each process (can also use strips)

Mandelbrot Set

Set of points in a complex plane that are quasi-stable (will increase and decrease, but not exceed some limit) when computed by iterating the function

$$z_{k+1} = z_k^2 + c$$

where z_{k+1} is the $(k + 1)^{\text{th}}$ iteration of the complex number $z = a + bi$ and c is a complex number giving position of the point in the complex plane. The initial value for z is zero.

Iterations continued until magnitude of z is greater than 2 or number of iterations reaches arbitrary limit. Magnitude of z is the length of the vector given by

$$z_{\text{length}} = \sqrt{a^2 + b^2}$$

Sequential routine computing value of one point returning numbers of iterations

```
structure complex {  
    float real;  
    float imag;  
};  
int cal_pixel(complex c)  
{  
int count, max;  
complex z;  
float temp, lengthsq;  
max = 256;  
z.real = 0; z.imag = 0;  
count = 0; /* number of iterations */  
do{  
    temp = z.real * z.real - z.imag * z.imag + c.real;  
    z.imag = 2 * z.real * z.imag + c.imag;  
    z.real = temp;  
    lengthsq = z.real * z.real + z.imag * z.imag;  
    count++;  
} while ((lengthsq <4.0) && (count < max));  
return count;  
}
```

To define complex
number

Mandelbrot set

Imaginary

+2

0

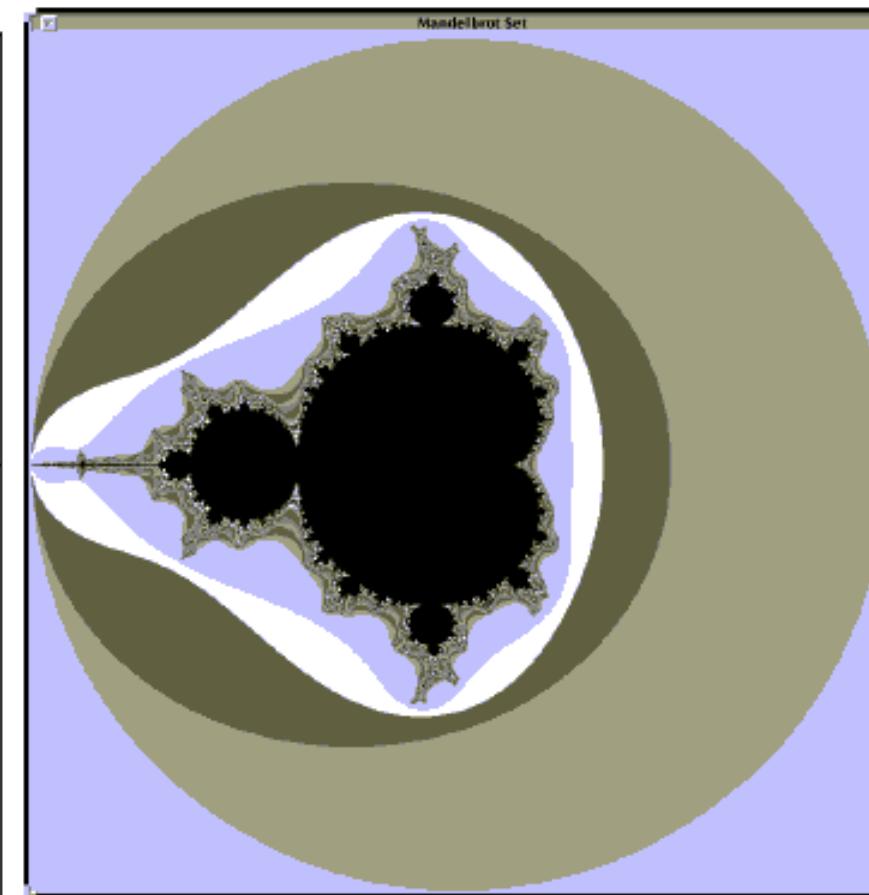
-2

-2

0

+2

Real



Mandelbrot Set

If the window is to display the complex plane with minimum values of (real_min, imag_min) and (real_max, imag_max). Each (x,y) point needs to be scaled by the factors

```
c.real = real_min + x * (real_max-real_min)/disp_width;
```

```
c.imag = imag_min + y * (imag_max-imag_min)/disp_height;
```

or

```
scale_real = (real_max-real_min)/disp_width;
```

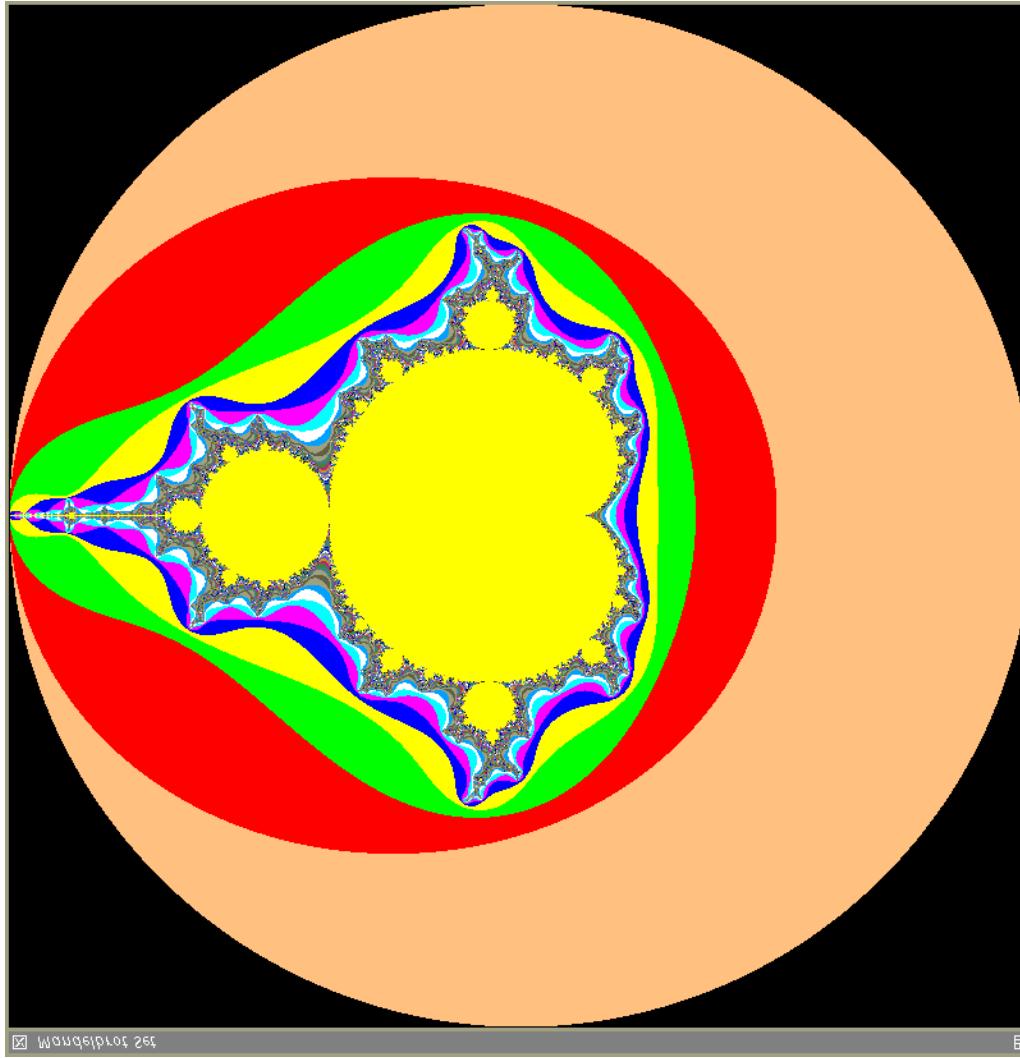
```
scale_imag = (imag_max-imag_min)/disp_height;
```

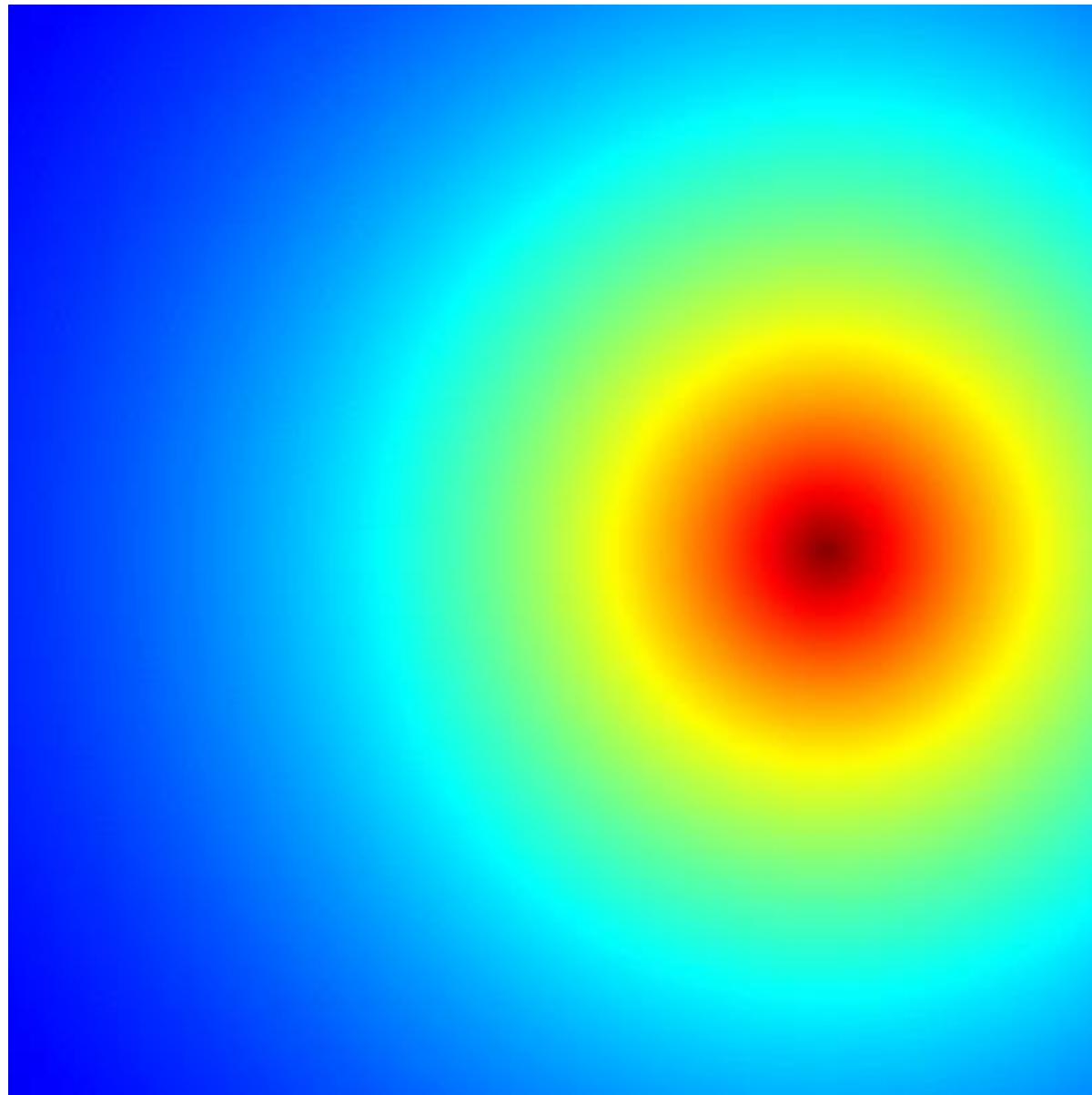
Mandelbrot Set

Including scaling, the code could be of the form

```
for (x = 0; x < disp_width; x++)  
    for (y = 0; y < disp_height; y++) {  
        c.real = real_min +((float) x *scale_real);  
        c.imag = imag_min +((float) y*scale_imag);  
        color = cal_pixel(c);  
        display(x,y, color);}
```

Sample Results





Parallelizing Mandelbrot Set Computation

Static Task Assignment

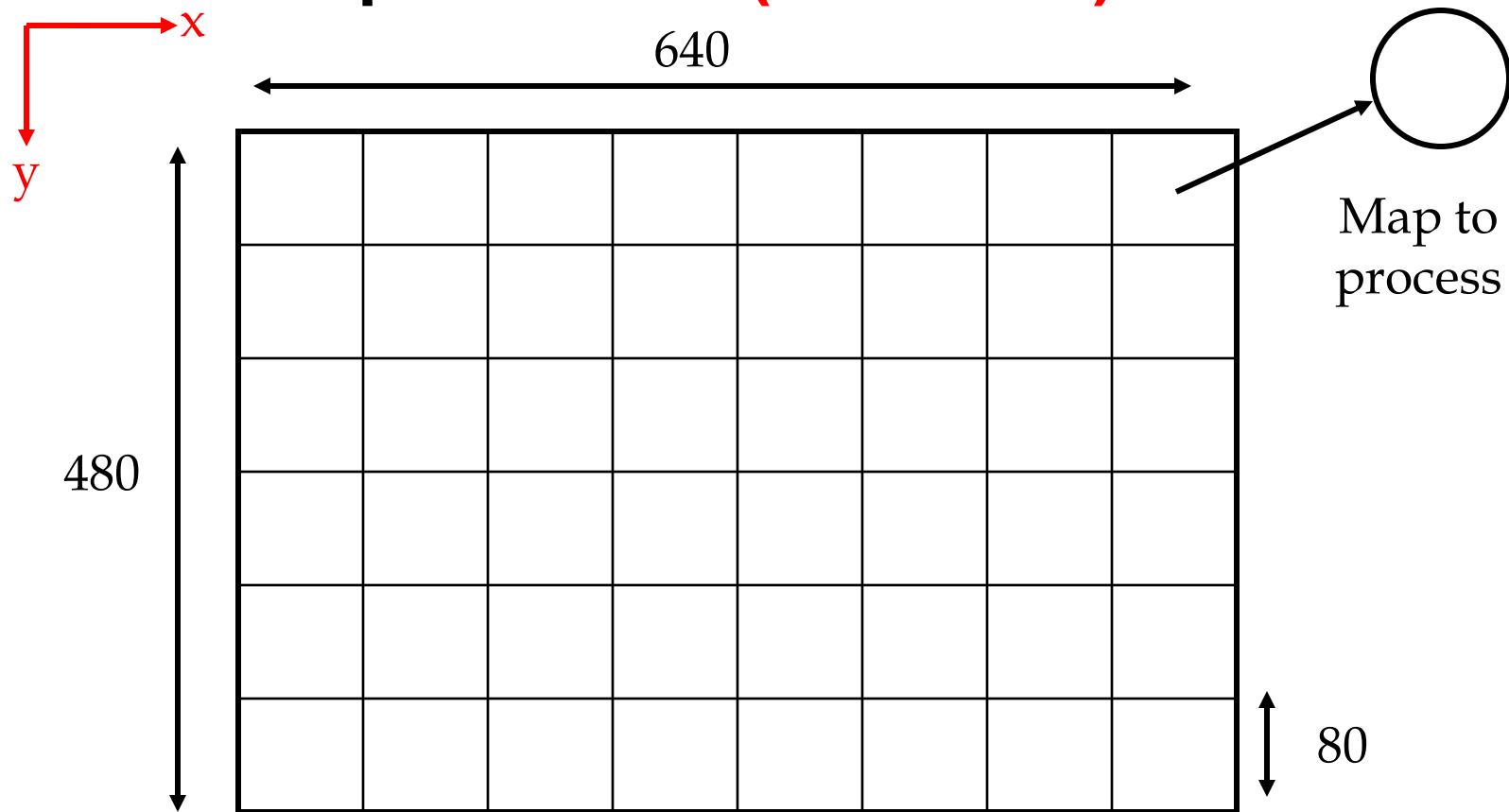
Simply divide the region in to fixed number of parts, each computed by a separate processor.

Not very successful because different regions require different numbers of iterations and time.

Dynamic Task Assignment

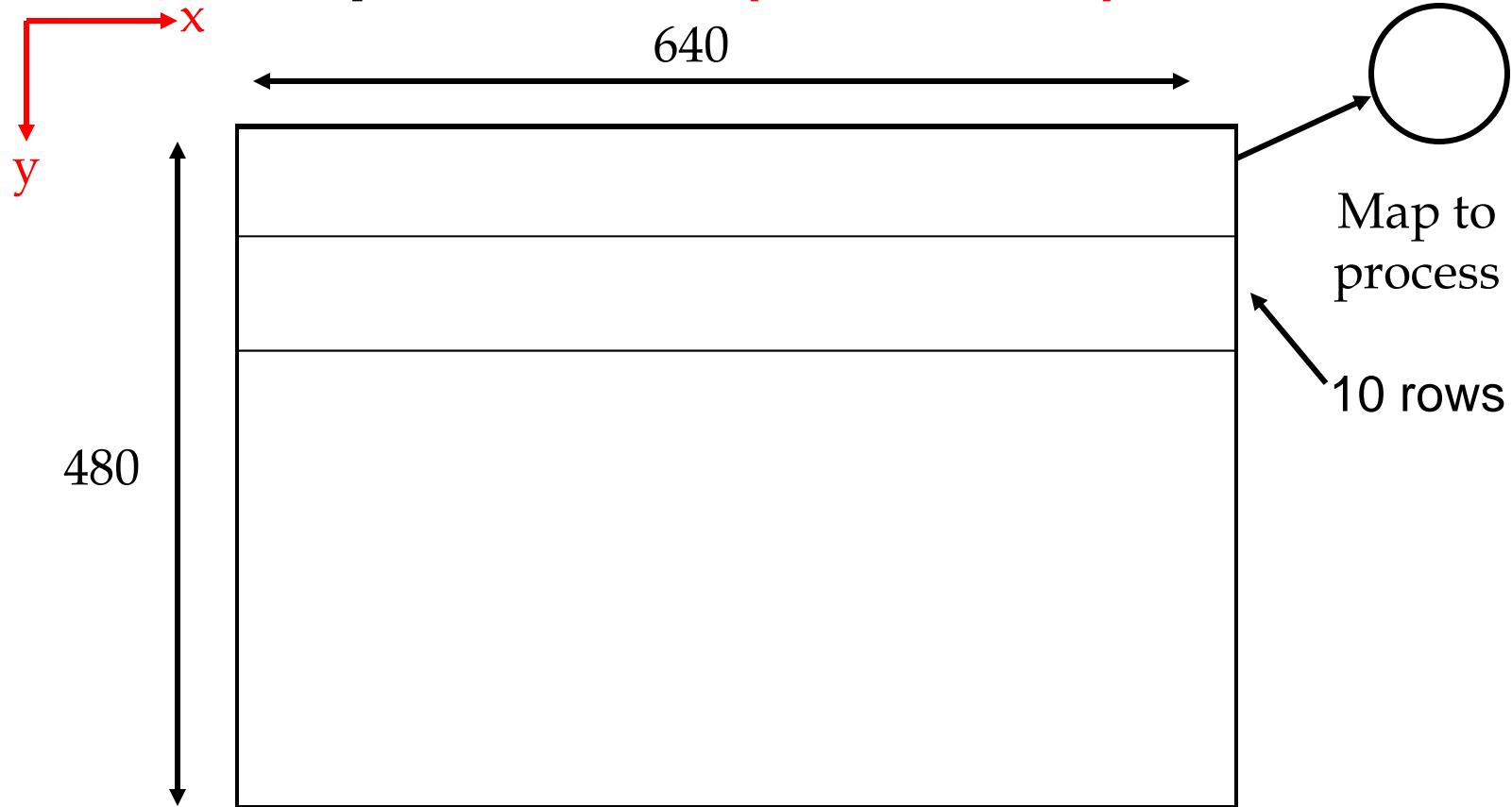
Have processor request regions after computing previous regions

Partitioning into regions for individual processes. (Solution I)



Square region for each process (can also use strips):
48 processes

Partitioning into regions for individual processes. (Solution II)



Row region for each process (again we require 48 processes)

Master

Process
id

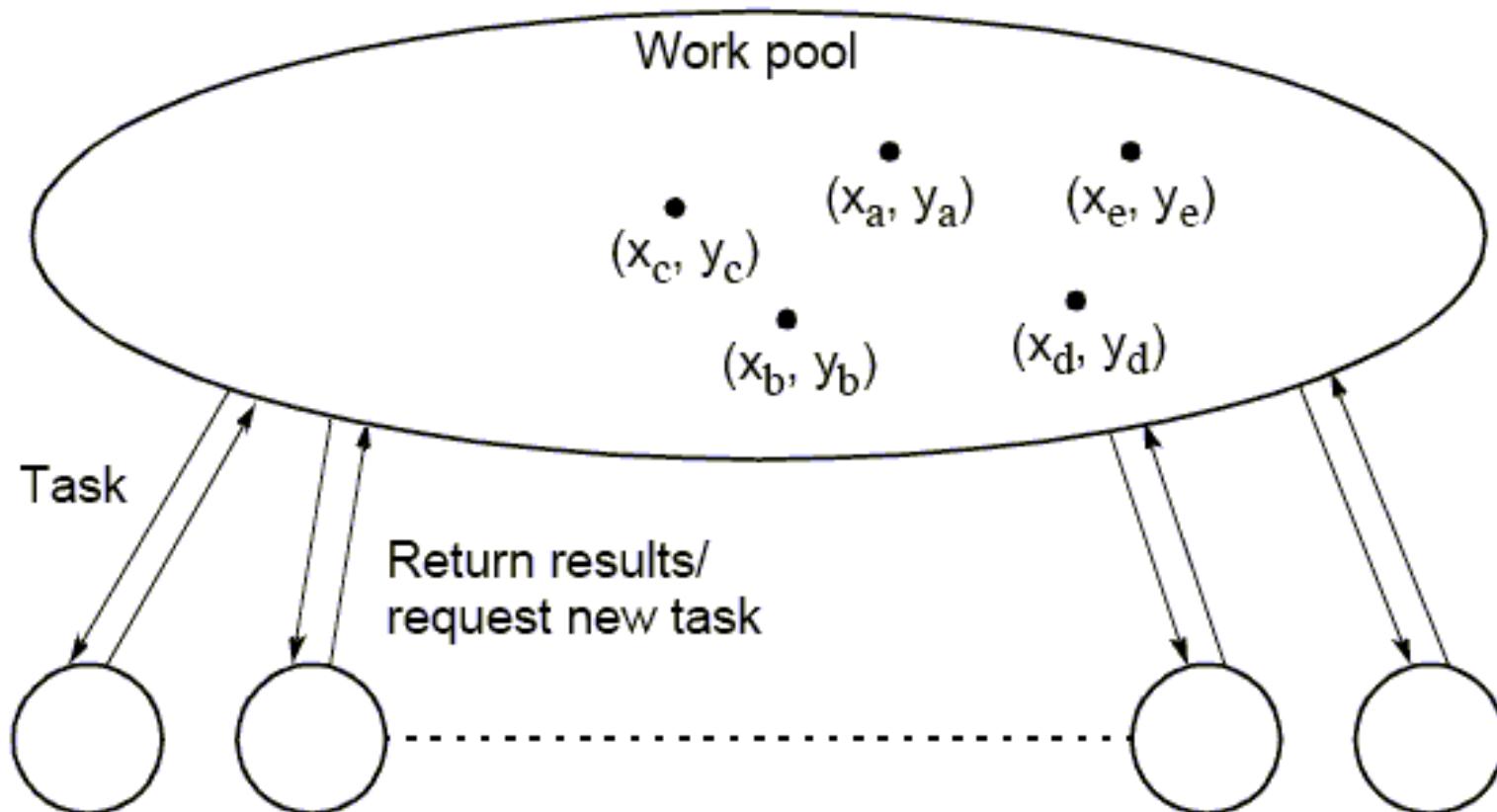
```
for (i = 0; row = 0; i < 48; i++, row = row + 10){ /*for each process*/  
    send( &row, Pi); /*send row number to processor*/  
for (i = 0; i <(480 * 640); ++i) /*from processor any order */  
    recv(&c, &color, PAny) /*receive coordinate /colors*/  
    display(c, color) /*display pixel on screen */
```

Slave (process i)

```
recv(&row, PMaster); /*receive row number*/  
for (x = 0; x < disp_width; x++)  
    for (y = row; y <(row + 10); y++) {  
        c.real = real_min + ((float) x * scale_real);  
        c.img = img_min + ((float) y * scale_imag);  
        color = cal_pixel(c);  
        send(&c, &color, PMaster); /*send coordinates and color*/  
    }
```

Dynamic Task Assignment

Work Pool/Processor Farms



Master

```
count = 0;                                /*counter for termination*/
row = 0;                                    /*row being sent*/
for (k = 0; k < procno; k++){               /*procno < disp_height*/
    send( &row, Pk, data_tag);           /*send initial row to processor*/
    count++;
row++;
}
do {
    recv(&slave, &r, color, Pany, result_tag);
    count--;                                /*reduce count as row received*/
    if (row < disp_height){                  /*send next row*/
        send(&row, Pslave, data_tag);      /*next row*/
        row++;
        count++;
    } else
        send(&row, Pslave, terminator_tag);/*terminate*/
    rows_recv++;
    display (r, color);
} while (count > 0);
```

Slave

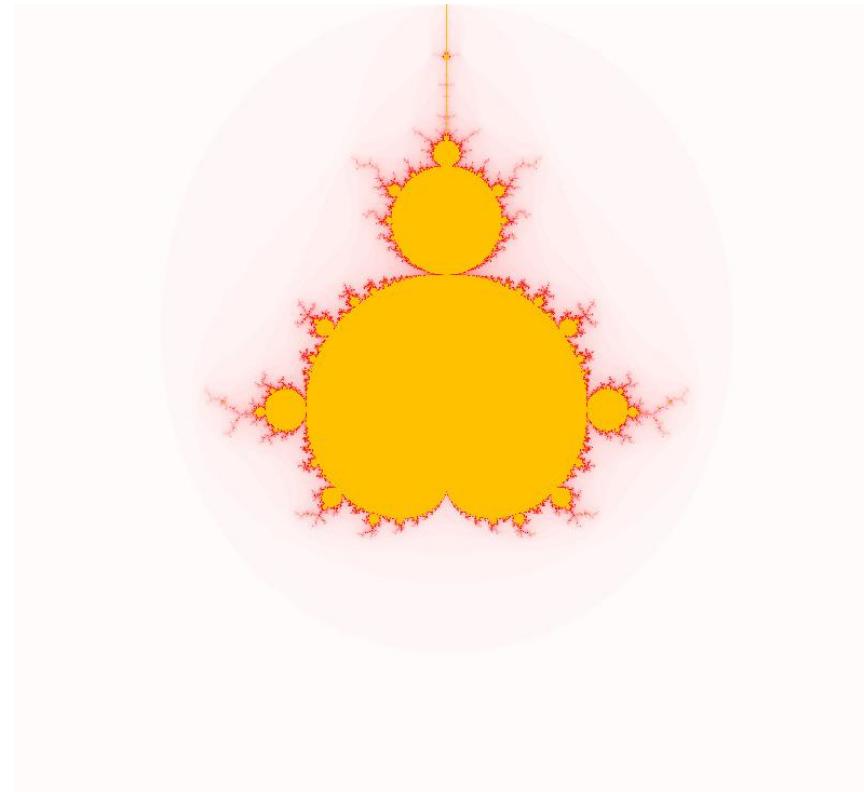
```
recv(y, Pslave, ANYTAG, source_tag);           /*receive 1st row to compute*/  
while (source_tag == data_tag) {  
    c.img = imag_min + ((float) y * scale)imag;  
    for (x = 0; c < disp_width; x++) /*compute row colors*/  
        c.real = real_min +((float) x *scale_real);  
        color[x] = cal_pixel(c);  
    }  
    send(&id, &y,color, Pslave, result_tag);/*send row colors to master*/  
    recv(y, Pmaster, source_tag);           /*receive next row*/  
};
```

Master

```
count = 0;                                /*counter for termination*/
row = 0;                                    /*row being sent*/
for (k = 0; k < procno; k++) {              /*procno < disp_height*/
    send( &row, Pk, data_tag);           /*send initial row to processor*/
    count++;

row++;
}
do {
    recv(&slave, &r, color, Pany, result_tag);
    count--;                                /*reduce count as row received*/
    if (row < disp_height){                  /*send next row*/
        send(&row, Pslave, data_tag);      /*next row*/
        row++;
        count++;
    } else
        send(&row, Pslave, terminator_tag);/*terminate*/
    rows_recv++;
    display (r, color);
} while (count > 0);
```

mpimand.c name on 14.139.69.97



Initially written as a portable arbitrary map format (PAM) file and then converted in jpg file and displayed

Julia set formed by $f(z) = z^2 + c$ where $c = 0.687 + 0.312i$

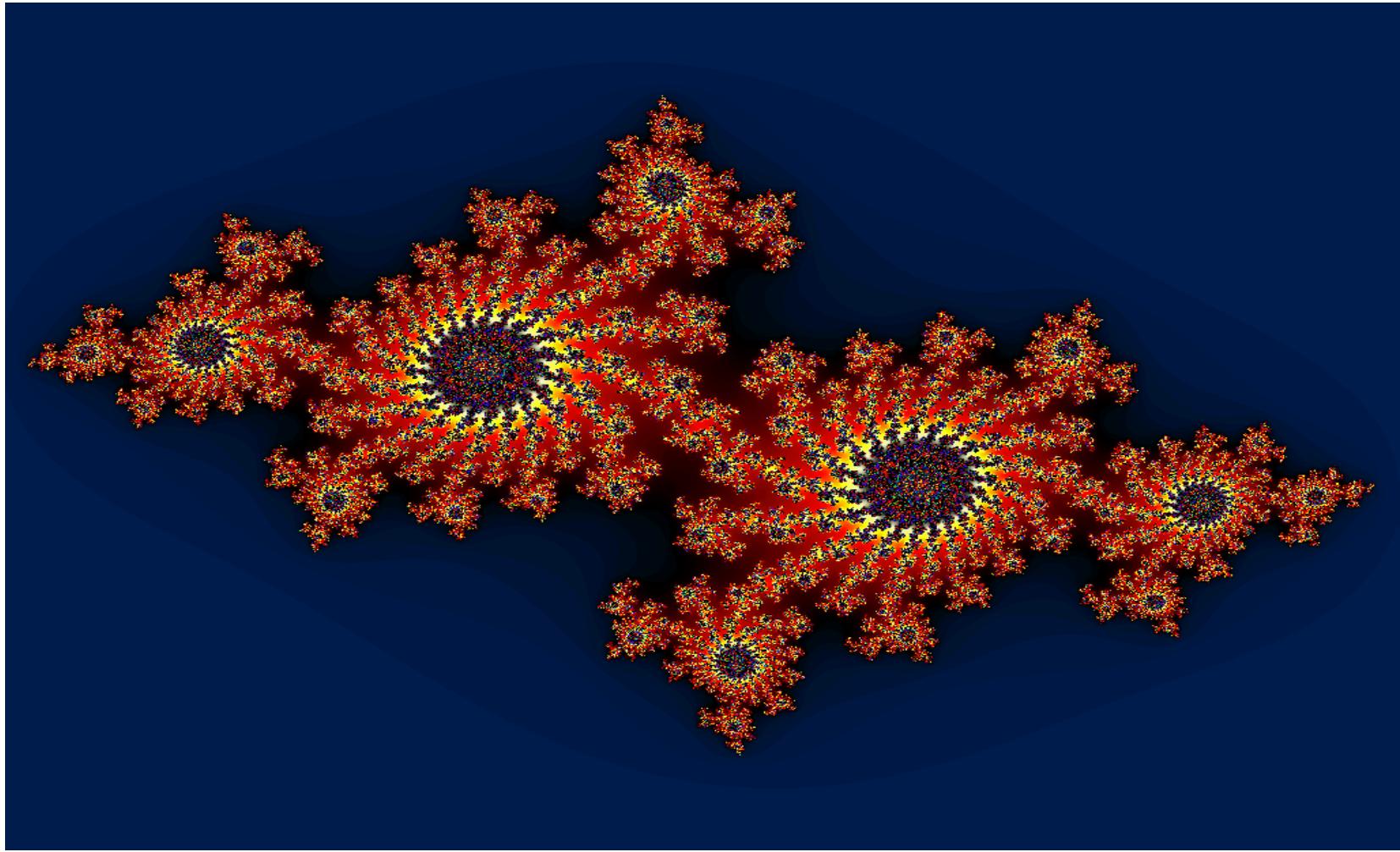
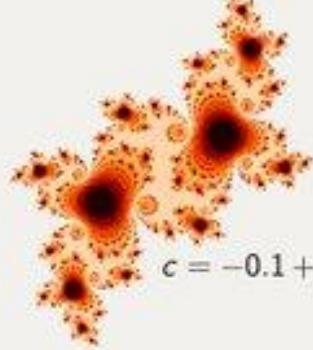
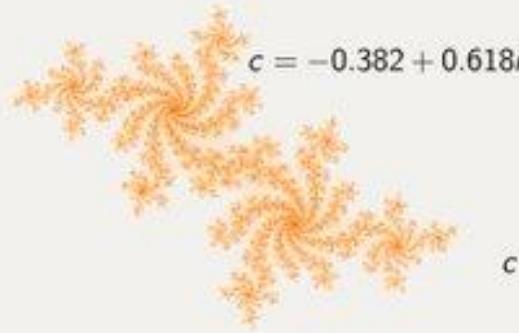


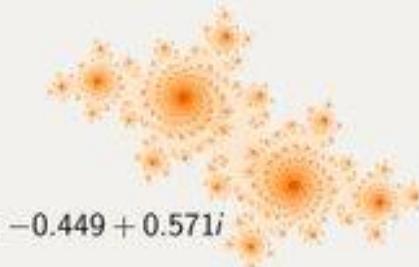
Image courtesy: Julia Sets (mcgoodwin.net)



$$c = -0.1 + 0.65i$$



$$c = -0.382 + 0.618i$$

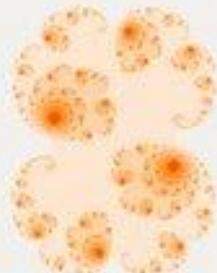


$$c = -0.449 + 0.571i$$

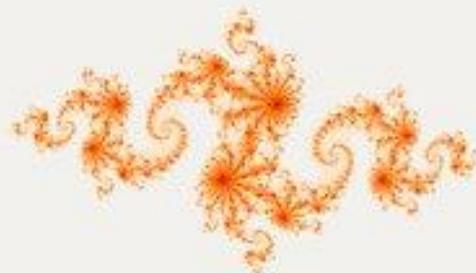
Julia Sets

The Julia Set associated to the function $f_c(z) = z^2 + c$ for a constant $c \in \mathbb{C}$

is the set $\mathcal{J}(f_c)$ of points $z \in \mathbb{C}$ such that $|f_c^n(z)| = |f_c(f_c(\dots f_c(z)))| \leq 2 \forall n$



$$c = 0.285 + 0.01i$$



$$c = -0.8 + 0.156i$$



$$c = 0.4 + 0.6i$$

Image courtesy: MathType

Monte Carlo Method

- **Basic:** Use random selections in calculations that lead to the solution to numerical and physical problems. It solves a problem using *statistical sampling*
- Name comes from Monaco's gambling resort city
- First important use in development of atomic bomb during World War II

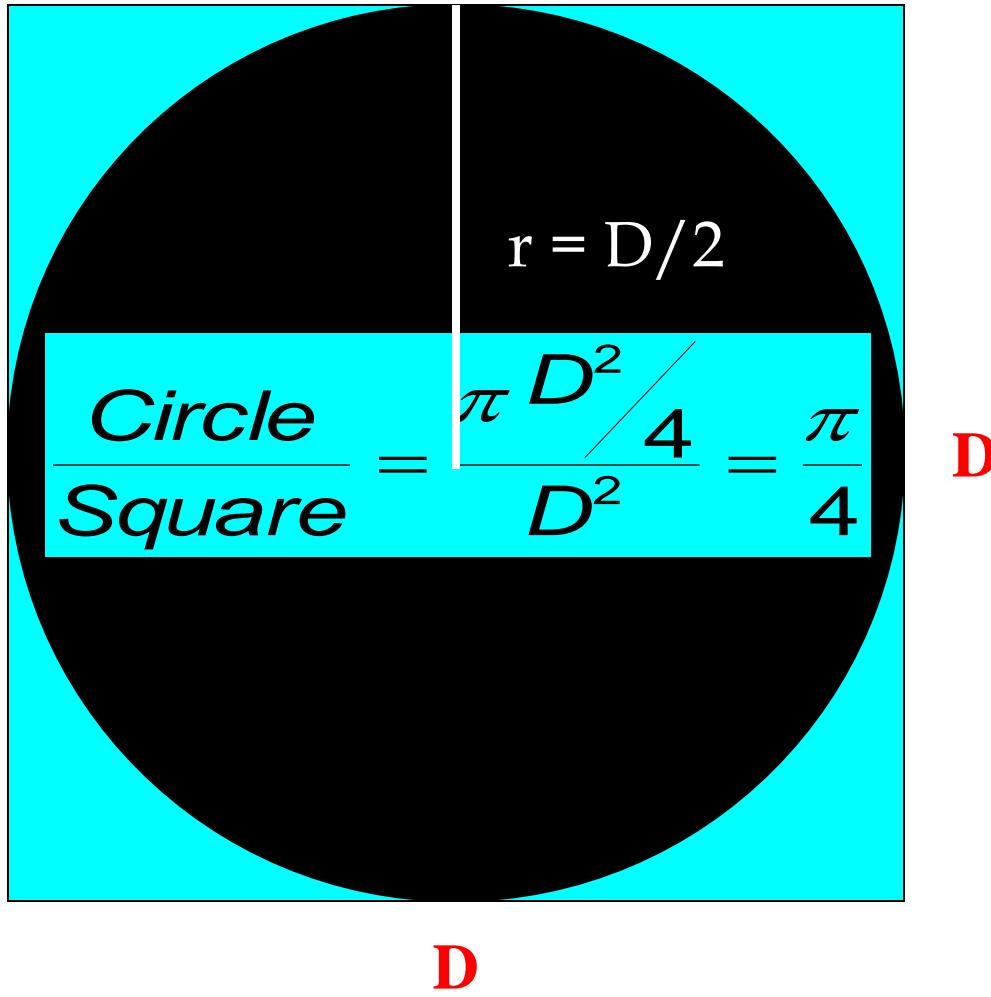
LV vs MC

- **Las Vegas (LV) Algorithms** - Are randomized algorithms which always give the correct answer. The running time however is not fixed (not deterministic), that is it can vary for the same input.
- Randomized Quick Sort always gives a correctly sorted array as its output. However it takes $O(n \log n)$ time on average but can be as bad as $O(n^2)$ in the worst case.

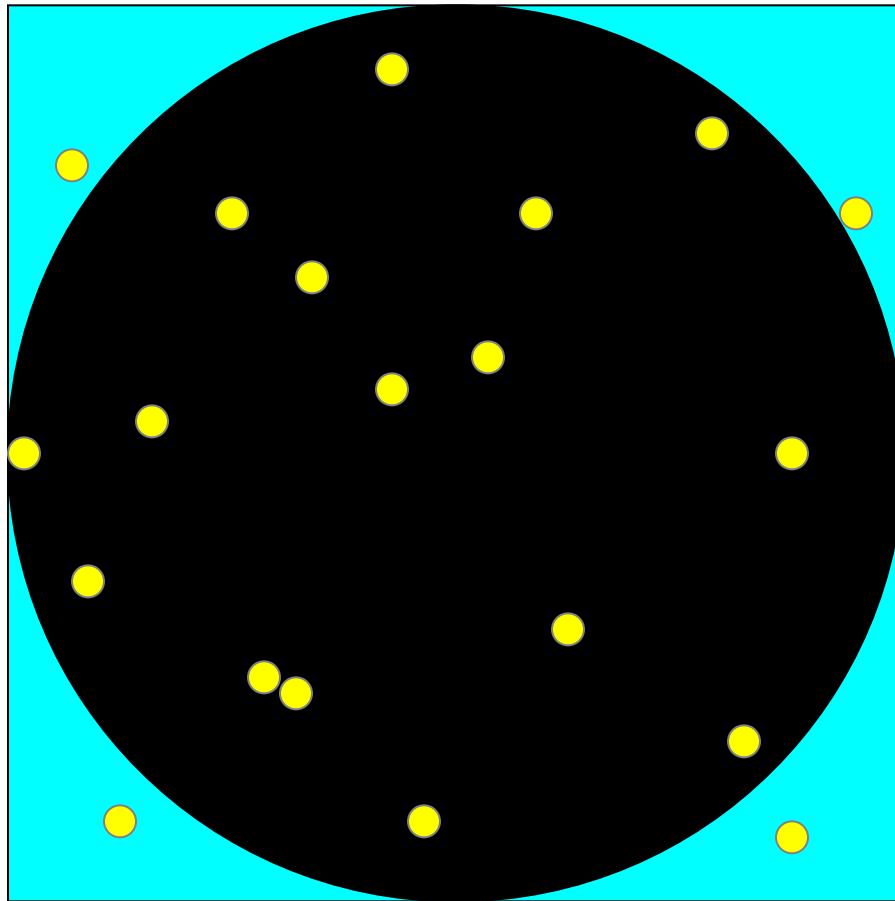
LV vs MC

- **Monte Carlo (MC) Algorithms** - Are randomized algorithms which may not always give the right answer. The running time for these algorithms is fixed however. There is also generally a probability guarantee that the answer is right.

Example of Monte Carlo Method



Example of Monte Carlo Method



```
count = 0  
for i = 1 to N:  
    x = random(0, D)  
    y = random(0, D)  
    if (x - D/2)2 + (y - D/2)2 ≤ (D/2)2:  
        count = count + 1  
 $\pi \approx 4 \times (\text{count} / N)$ 
```

$$\frac{16}{20} \approx \frac{\pi}{4} \Rightarrow \pi \approx 3.2$$

Applications of Monte Carlo Method

- Evaluating integrals of arbitrary functions of higher dimensions
- Predicting future values of stocks
- Predicting weather
- Solving partial differential equations
- Sharpening satellite images
- Modeling cell populations
- Finding approximate solutions to NP-hard problems

MC Example: Computing the integral

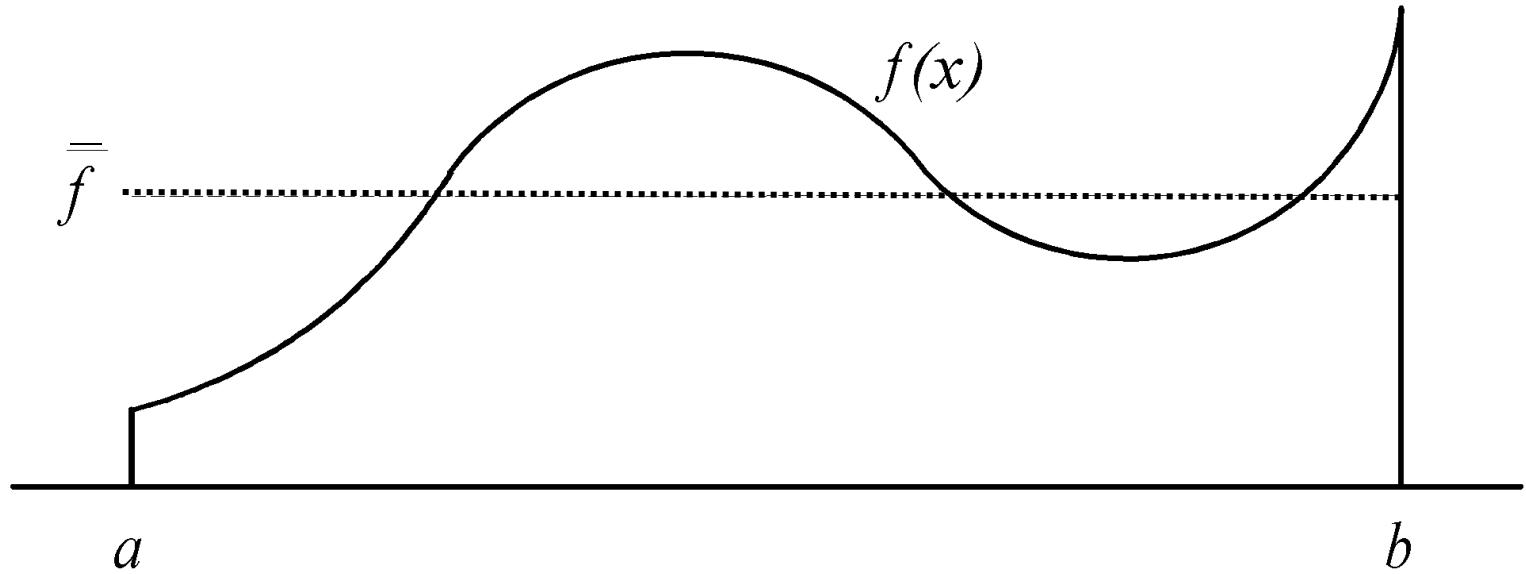
For integrals that can't be solved analytically, estimate by random sampling:

The probabilistic method to find an integral is to use random values of x to computer $f(x)$ and sum the values of $f(x)$

$$\text{Area} = \int_{x_1}^{x_2} f(x)dx = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{r=1}^N f(x_r)(x_2 - x_1)$$

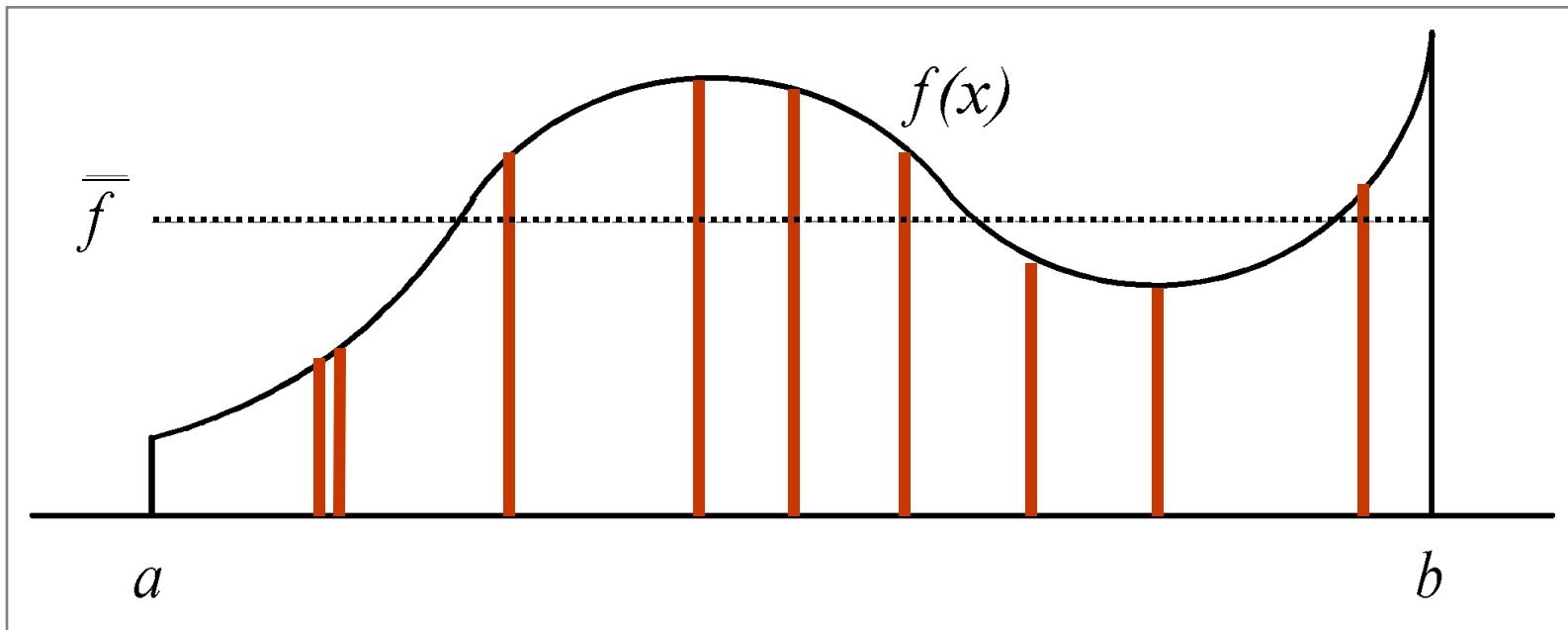
Here x_r is random number generated between x_1 and x_2 .

Mean Value Theorem



$$\int_a^b f(x)dx = (b-a) \bar{f}$$

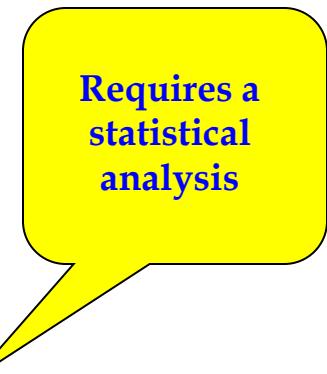
Estimating Mean Value



The expected value of $(1/n)(f(x_1) + \dots + f(x_n))$ is \bar{f}

Example: Computing the integral

Sequential Code $I = \int_{x_1}^{x_2} (x^2 - 3x) dx$



Requires a statistical analysis

```
sum = 0;  
  
for (i = 0; i < N; i++) {  
  
    x_r = rand_v(x1, x2);      /* generate next random value */  
  
    sum = sum + x_r * x_r - 3 * x_r;  /* compute f(x_r) */  
  
}  
  
area = (sum / N) * (x2 - x1);
```

/* N random samples */

Routine `rand_v(x1, x2)` returns a pseudorandom number between x_1 and x_2 .

Variable	Meaning
N	Total number of random samples for the whole computation
n	Number of random samples assigned to each slave per batch
N/n	Number of batches master must create to generate all N samples
slave_no	Total number of slave processors
xr[j]	Array of n random numbers generated by master and sent to slave j
sum (in slave)	Partial sum of $f(xr)$ for its batch
reduce_add(&sum, Pgroup)	Collective addition of all partial sums to master

If $N = 1,000,000$ and $n = 10,000$ then master will send **100 batches**.

Master

```
for (i = 0; i < N/n; i++) {
    for (j = 0; j < n; j++)
        xr[j] = rand(); /* n = no of random no for slaves*/
        /*load numbers to be sent*/

    recv(PANY, req_tag, Psource); /* wait for the slave to req*/
}

send(xr, &n, Psource, compute_tag);
}

for (i = 0; i < slave_no; ++i) /*terminate computation*/
    recv(PAny, req_tag)
    send(Pi, stop_tag);

}
sum = 0;
reduce_add(&sum, Pgroup);
```

Slave

```
sum = 0;
send(Pmaster, req_tag);
recv(xr, &n, Pmaster, source_tag) {
while(source_tag == compute_tag){
    for (i = 0, i < n; i++)
        sum = sum + xr[i] + xr[i] - 3 * xr[i];
    send(Pmaster, req_tag);
    recv(xr, &n, Pmaster, source_tag);
};
reduce_add(&sum, Pgroup)
```

SPDM Program

Input: x_1, x_2, N

// Integration limits and total random samples

p = total number of processes

id = rank of this process ($0 \dots p-1$)

// Divide total samples equally

$local_N = N / p$

// Each process computes its own partial sum

$local_sum = 0$

for $i = 1$ to $local_N$:

$xr = random(x_1, x_2)$

$local_sum = local_sum + (xr*xr - 3*xr)$

// Combine all partial sums from all processes

$global_sum = \text{SUM}$ of all $local_sum$ across processes

(using collective reduction)

// Only one process (say $id = 0$) computes final result

if $id == 0$:

$area = (global_sum / N) * (x_2 - x_1)$

print "Estimated integral =", area

Partitioning and Divide-and-Conquer Strategies

Partitioning

Partitioning simply divides the problem into parts-**Domain decomposition** & **Functional decomposition**.

Divide and Conquer

Characterized by dividing a problem into sub-problems that are of the same form as the larger problem. Further divisions into still smaller sub-problems are usually done by recursion.

Recursive divide and conquer amenable to parallelization because separate processes can be used for the divided parts.

Also usually data is naturally localized.

Divide and conquer



- ❖ A parent process divides its workload into several smaller pieces and assigns them to a number of child processes.
- ❖ The child processes then compute their workload in parallel and the results are merged by the parent.
- ❖ The dividing and the merging procedures are done recursively.
- ❖ This paradigm is very natural for computations such as merge sort. **Its disadvantage is the difficulty in achieving good load balance.**

Partitioning/Divide and Conquer Examples

Many possibilities.

- Operations on sequences of numbers such as simply adding them together
- Several sorting algorithms can often be partitioned or constructed in a recursive fashion
- Numerical integration

Outline

- The bucket sort makes assumptions about the data being sorted
- Consequently, we can achieve better than $O(n \ln(n))$ run times
- We will look at:
 - a supporting example
 - the algorithm (sequential and parallel)
 - run times (no best-, worst-, or average-cases)
 - summary and discussion

Bucket Sort: Supporting Example

- Suppose we are sorting a large number of local phone numbers, for example, all residential phone numbers in 040 code region (over two million)
- We could use heap sort, however that would require an array which is kept entirely in memory

Bucket Sort: Supporting Example

- Instead, consider the following scheme:
 - create a vector with 10 000 000 bits
 - this requires 9.54 MiB
 - set each bit to 0 (indicating false)
 - for each phone number, set the bit indexed by the phone number to 1 (true)
 - once each phone number has been checked, walk through the array and for each bit which is 1, record that number

Bucket Sort: Supporting Example

- In this example, the number of phone numbers (2 000 000) is comparable to the size of the array (10 000 000)
- The run time of such an algorithm is $\mathbf{O}(n)$:
 - we make one pass through the data,
 - we make one pass through the array and extract the phone numbers which are true

Bucket Sort

- This approach uses very little memory and allows the entire structure to be kept in main memory at all times
- We will term each entry in the bit vector a *bucket*
- We fill each bucket as appropriate

Example

- Consider sorting the following set of unique integers in the range 0, ..., 31:

20	1	31	8	29	28	11	14	6	16	15
27	10	4	23	7	19	18	0	26	12	22

- Create an bit-vector with 32 buckets
- This requires 4 bytes

00	<input type="checkbox"/>
01	<input type="checkbox"/>
02	<input type="checkbox"/>
03	<input type="checkbox"/>
04	<input type="checkbox"/>
05	<input type="checkbox"/>
06	<input type="checkbox"/>
07	<input type="checkbox"/>
08	<input type="checkbox"/>
09	<input type="checkbox"/>
10	<input type="checkbox"/>
11	<input type="checkbox"/>
12	<input type="checkbox"/>
13	<input type="checkbox"/>
14	<input type="checkbox"/>
15	<input type="checkbox"/>
16	<input type="checkbox"/>
17	<input type="checkbox"/>
18	<input type="checkbox"/>
19	<input type="checkbox"/>
20	<input type="checkbox"/>
21	<input type="checkbox"/>
22	<input type="checkbox"/>
23	<input type="checkbox"/>
24	<input type="checkbox"/>
25	<input type="checkbox"/>
26	<input type="checkbox"/>
27	<input type="checkbox"/>
28	<input type="checkbox"/>
29	<input type="checkbox"/>
30	<input type="checkbox"/>
31	<input type="checkbox"/>

Example

- For each number, set the bit of the corresponding bucket to 1
- Now, just traverse the list and record only those numbers for which the bit is 1 (true):

0	1	4	6	7	8	10	11	12	14	15
16	18	19	20	22	23	26	27	28	29	31

00	✓
01	✓
02	
03	
04	✓
05	
06	✓
07	✓
08	✓
09	
10	✓
11	✓
12	✓
13	
14	✓
15	✓
16	✓
17	
18	✓
19	✓
20	✓
21	
22	✓
23	✓
24	
25	
26	✓
27	✓
28	✓
29	✓
30	
31	✓

Bucket Sort

- How is this so fast?
- An algorithm which can **sort arbitrary data** must be $\Omega(n \ln(n))$
- In this case, **we don't have arbitrary data**: we have one further constraint, that the items being sorted fall within a certain range
- Using this assumption, we can reduce the run time

Bucket Sort

- Modification: what if there are **repetitions in the data**
- In this case, a bit vector is insufficient
- Two options, each bucket is either:
 - a counter, or
 - a linked list
- The first is better if objects in the bin are the same

Example

- Sort the digits

0 3 2 8 5 3 7 5 3 2 8 2 3 5 1 3 2 8 5 3 4 9 2 3 5 1 0 9 3 5 2 3 5 4 2
1 3

- We start with an array of 10 counters, each initially set to zero:

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Example

- Moving through the first 10 digits

0 3 2 8 5 3 7 5 3 2 8 2 3 5 1 3 2 8 5 3 4 9 2 3 5 1 0 9 3 5 2 3 5 4 2
1 3

we increment the corresponding
buckets

0	1
1	0
2	2
3	3
4	0
5	2
6	0
7	1
8	1
9	0

Example

- Moving through remaining digits

0 3 2 8 5 3 7 5 3 2 8 2 3 5 1 3 2 8 5 3 4 9 2 3 5 1 0 9 3 5 2 3 5 4 2
1 3

we continue incrementing the corresponding buckets

0	2
1	3
2	7
3	10
4	2
5	7
6	0
7	1
8	3
9	2

Example (using Counter)

- We now simply read off the number of each occurrence:

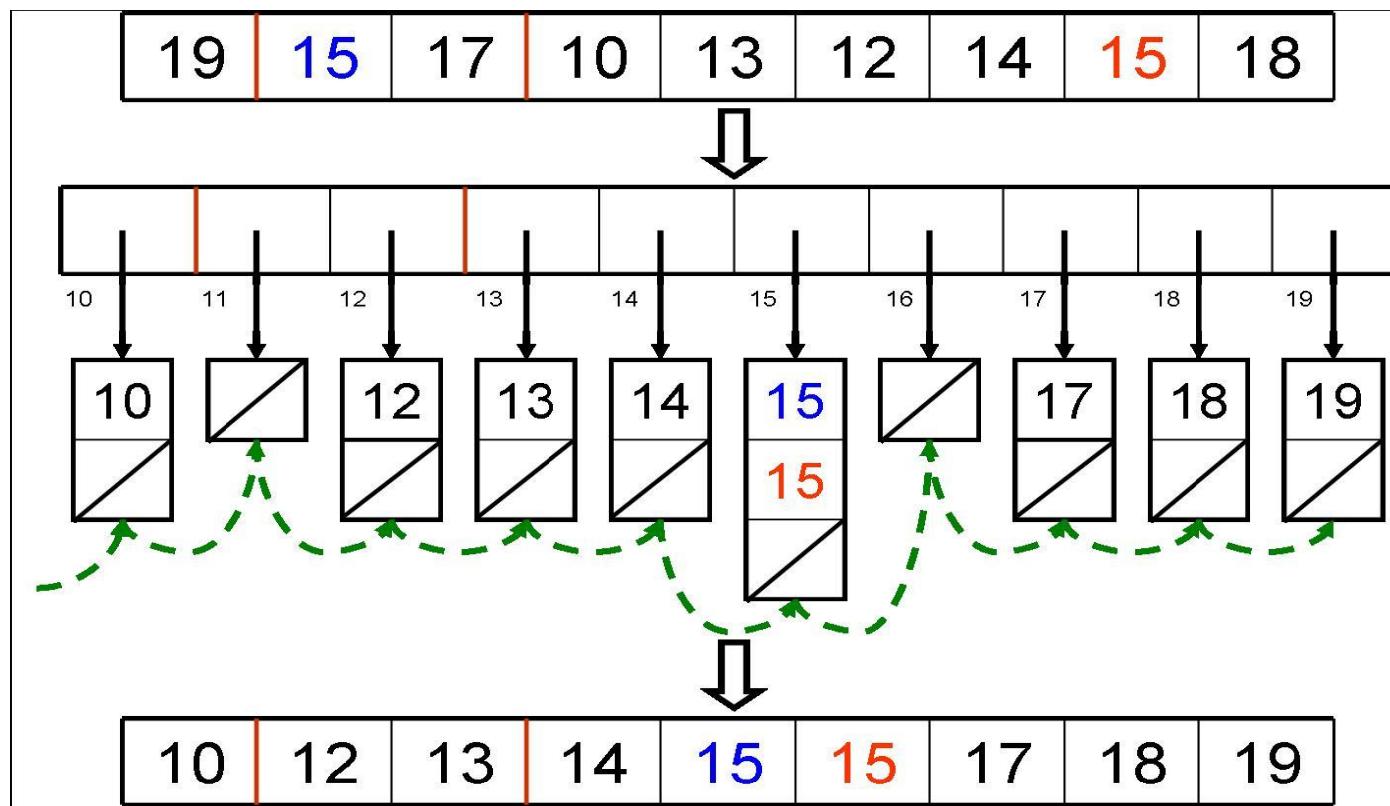
0 0 1 1 1 **2 2 2 2 2 2** 3 3 3 3 3 3 3 3 3 3 4 4 5 5 5 5 5 5 5 5 7 8 8 8
9 9

0	2
1	3
2	7
3	10
4	2
5	7
6	0
7	1
8	3
9	2

- For example:
 - there are seven **2s**
 - there are two **4s**

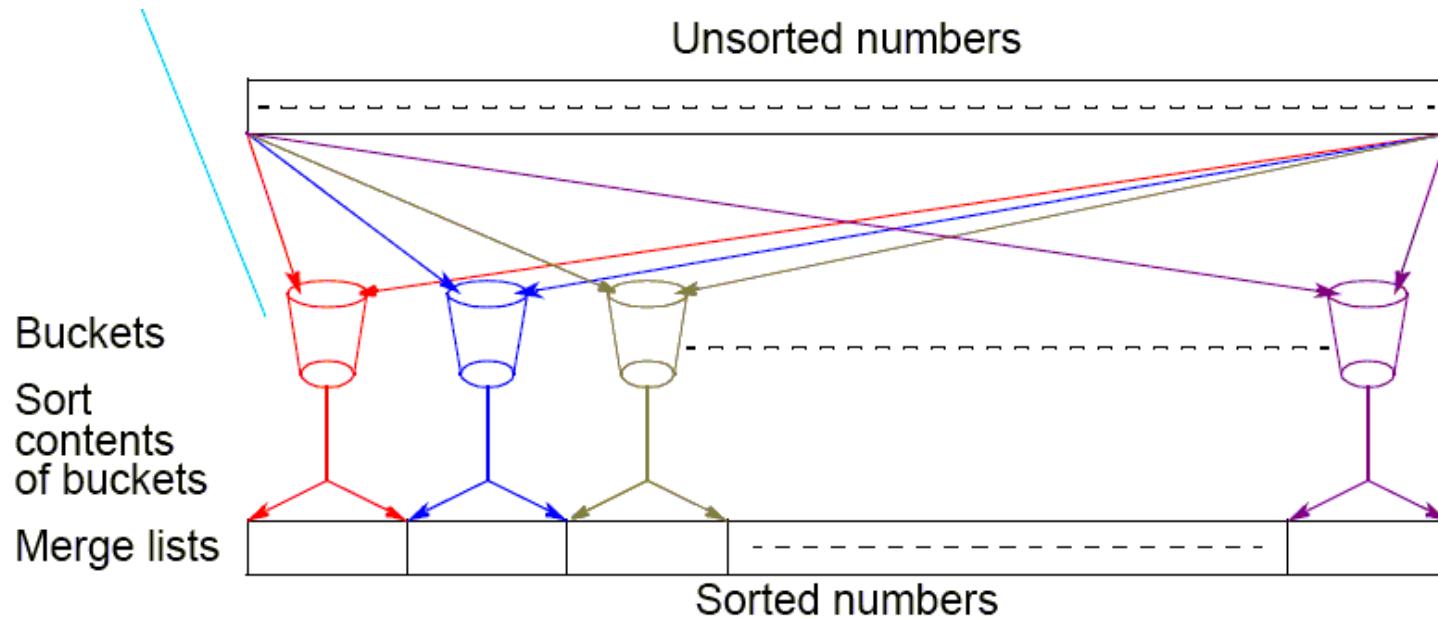
Example (using Linked List)

The example uses an input array of 9 elements. Key values are in the range from 10 to 19. It uses an auxiliary array of linked lists which is used as buckets.



Bucket sort

One “bucket” assigned to hold numbers that fall within each region. Numbers in each bucket sorted using a sequential algorithm (Insertion sort, if required!)

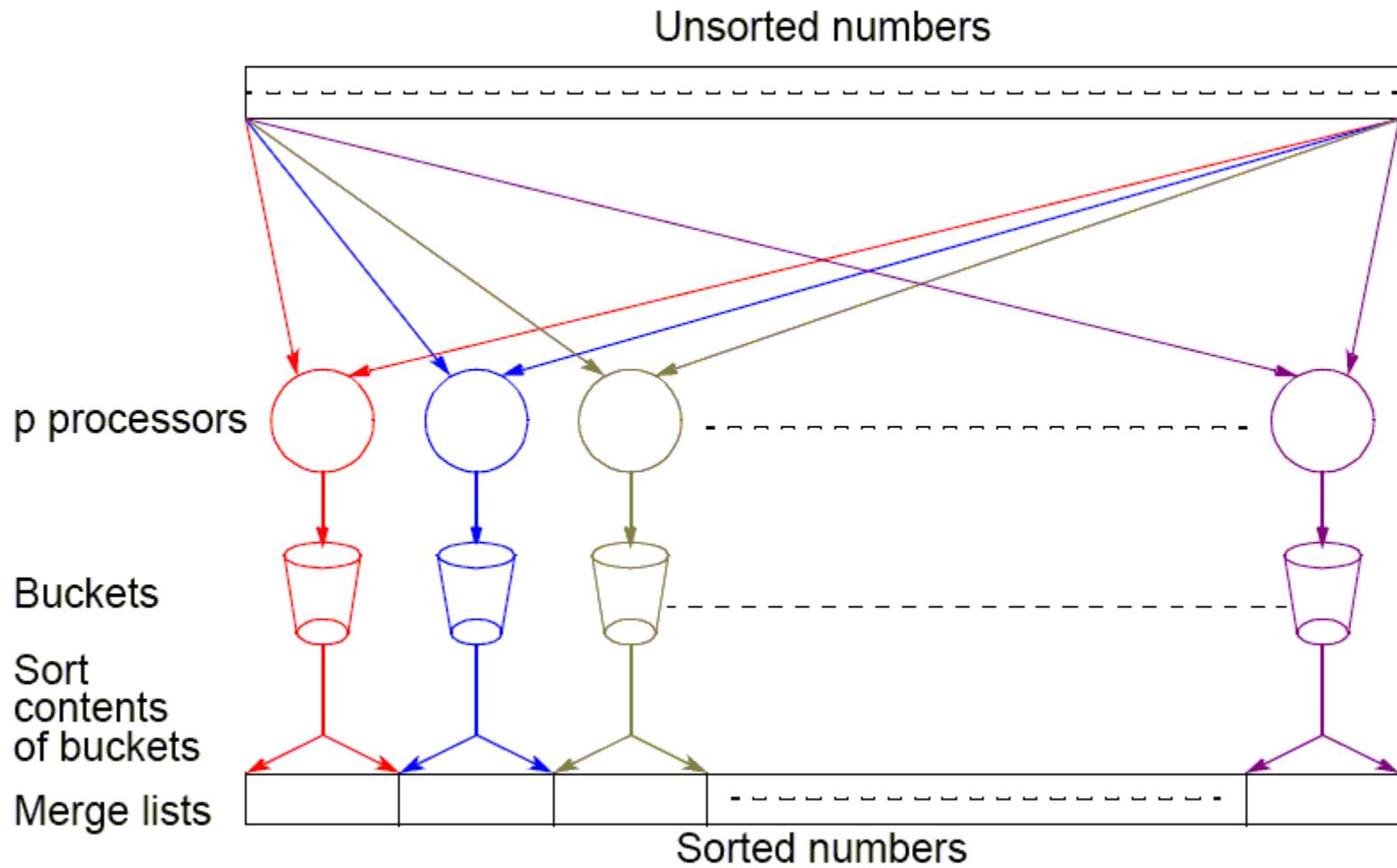


Sequential sorting time complexity: $O(n \log(n/m))$.

Works well only if the original numbers uniformly distributed across a known interval, say $[0, 1]$.

Parallel version of Bucket sort Simple approach

Assign one processor for each bucket.



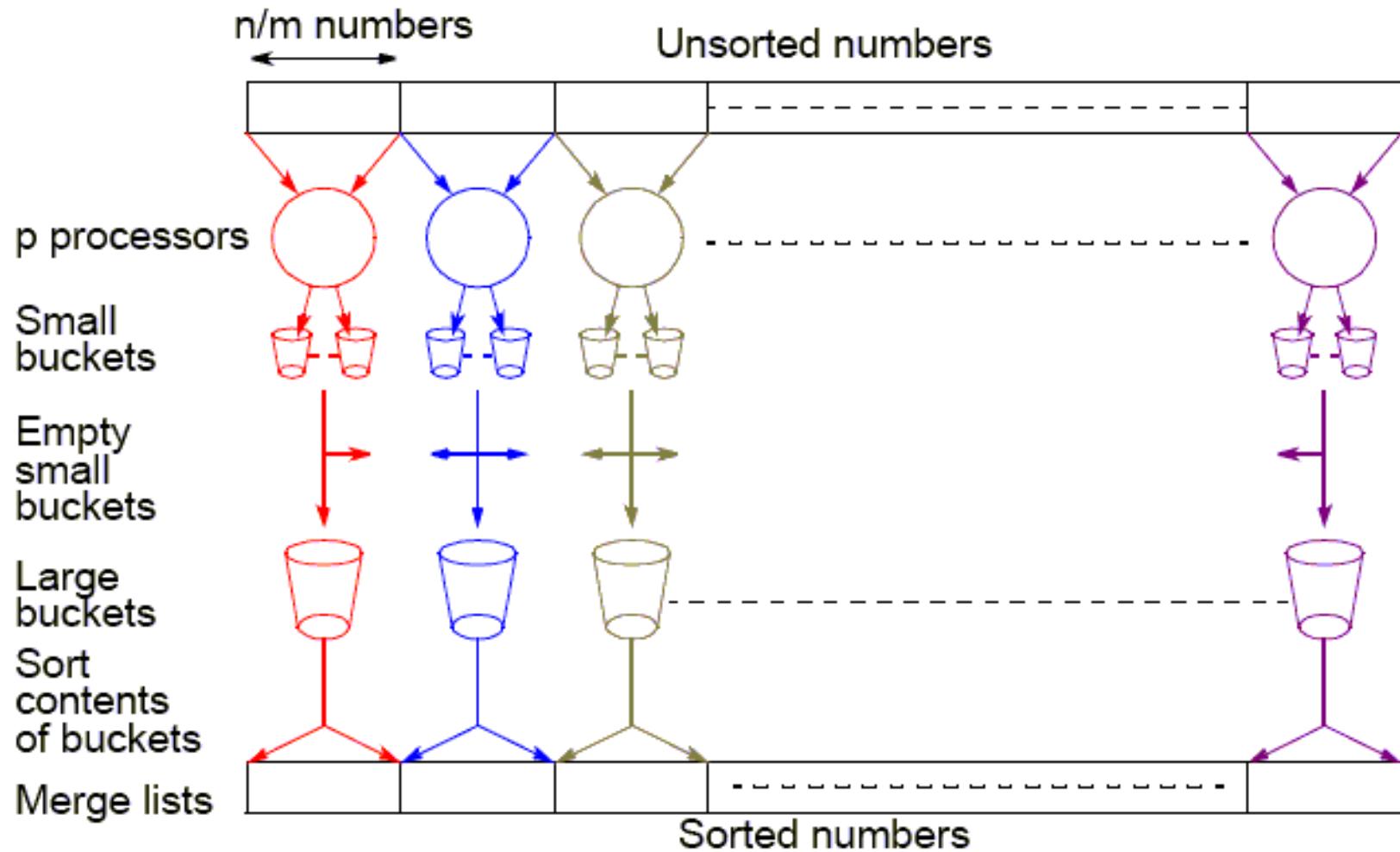
Further Parallelization

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

Each processor maintains p “small” buckets and separates the 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 j).

Another parallel version of bucket sort



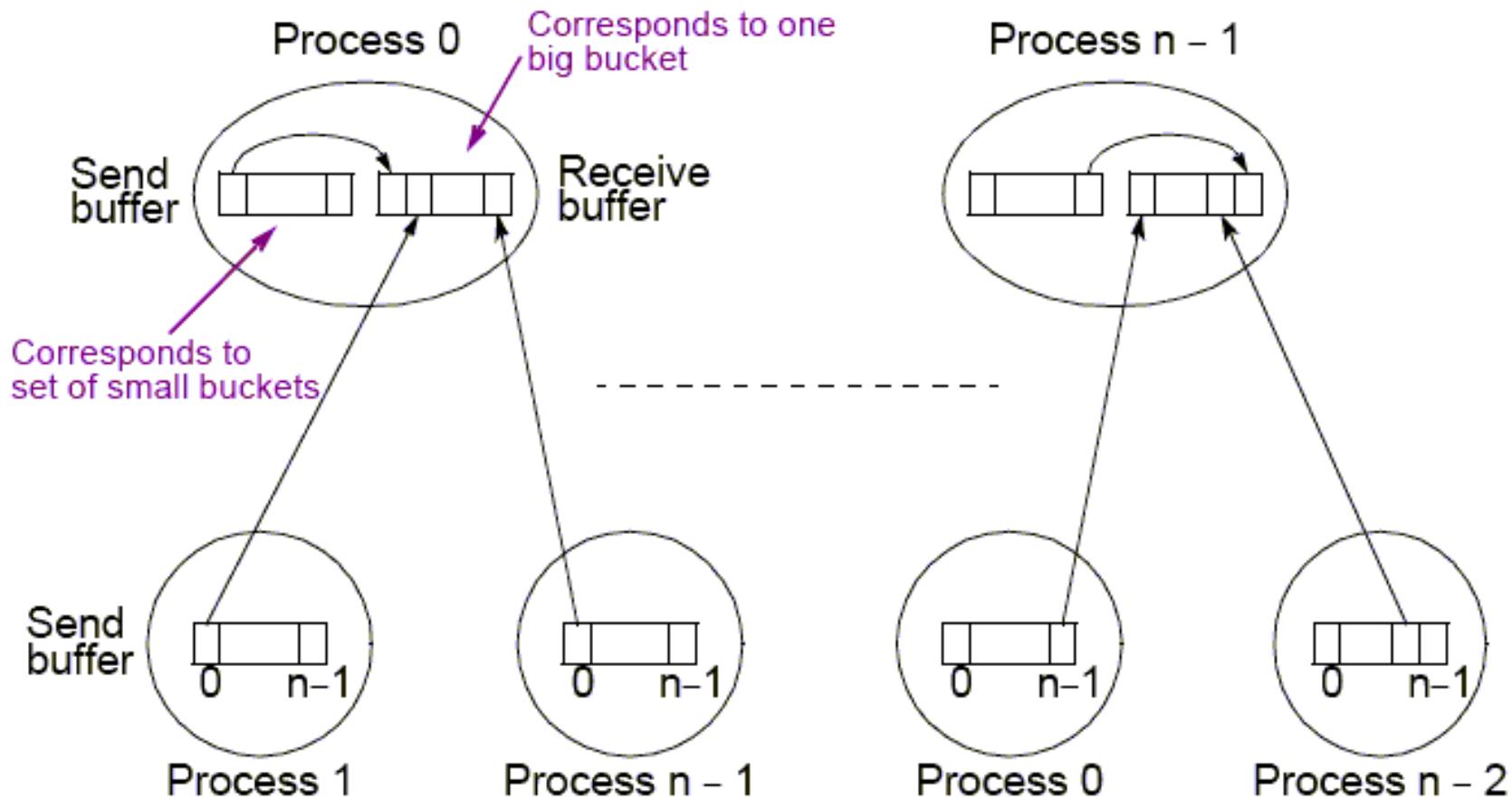
The following phases are needed

- Partition numbers
- Sort into small buckets
- Send to large buckets
- Sort large buckets

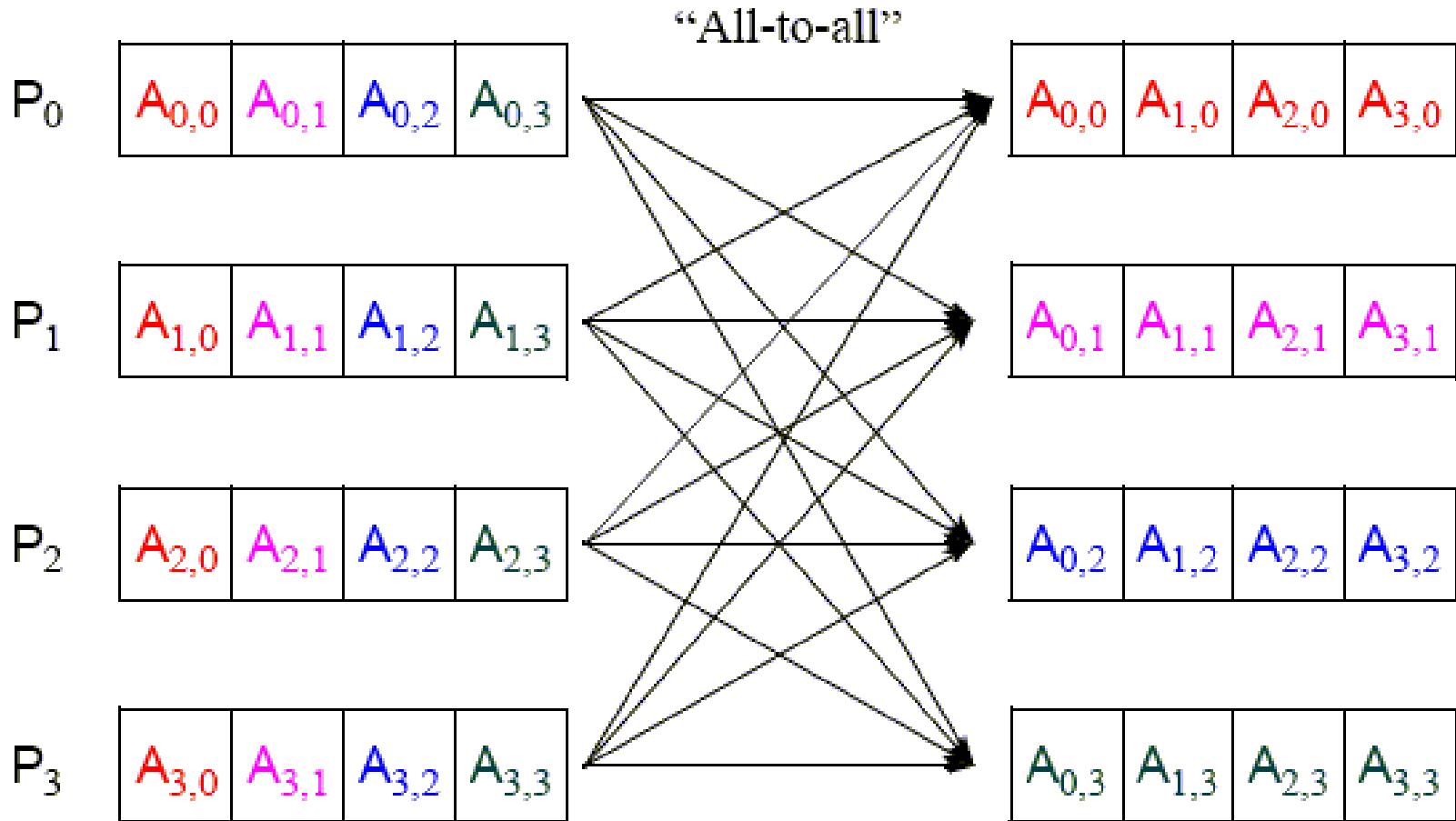
We require – all-to-all broadcast.

“all-to-all” broadcast routine

Sends data from each process to every other process



“all-to-all” routine actually transfers rows of an array to columns



Effect of “all-to-all” on an array

Data Parallel Example - Prefix Sum Problem

Given a list of numbers, x_0, \dots, x_{n-1} , compute all the partial summations (i.e., $x_0 + x_1$; $x_0 + x_1 + x_2$; $x_0 + x_1 + x_2 + x_3$; ...).

Can also be defined with associative operations other than addition.

Widely studied. Practical applications in areas such as processor allocation, data compaction, sorting, and polynomial evaluation.

Sequential code

```
for(i = 0; i < n; i++) {  
    sum[i] = 0;  
  
    for (j = 0; j <= i; j++)  
        sum[i] = sum[i] + x[j];  
}
```

An $O(n^2)$ algorithm.

Parallel Method

For obtaining the partial sum of 16 numbers, a different number of computations occurs in each step.

First $15(16-1)$ additions occur in which $x[i-1]$ is added to $x[i]$, $1 \leq i < 16$

Second $14(16-2)$ additions occur in which $x[i-2]$ is added to $x[i]$, $2 \leq i < 16$

Then $12(16-4)$ additions occur in which $x[i-4]$ is added to $x[i]$, $4 \leq i < 16$

In General, method requires $\log n$ steps in which

$n - 2^j$ additions occur in which $x[i-2^j]$ is added to $x[i]$, $2^j \leq i < n$

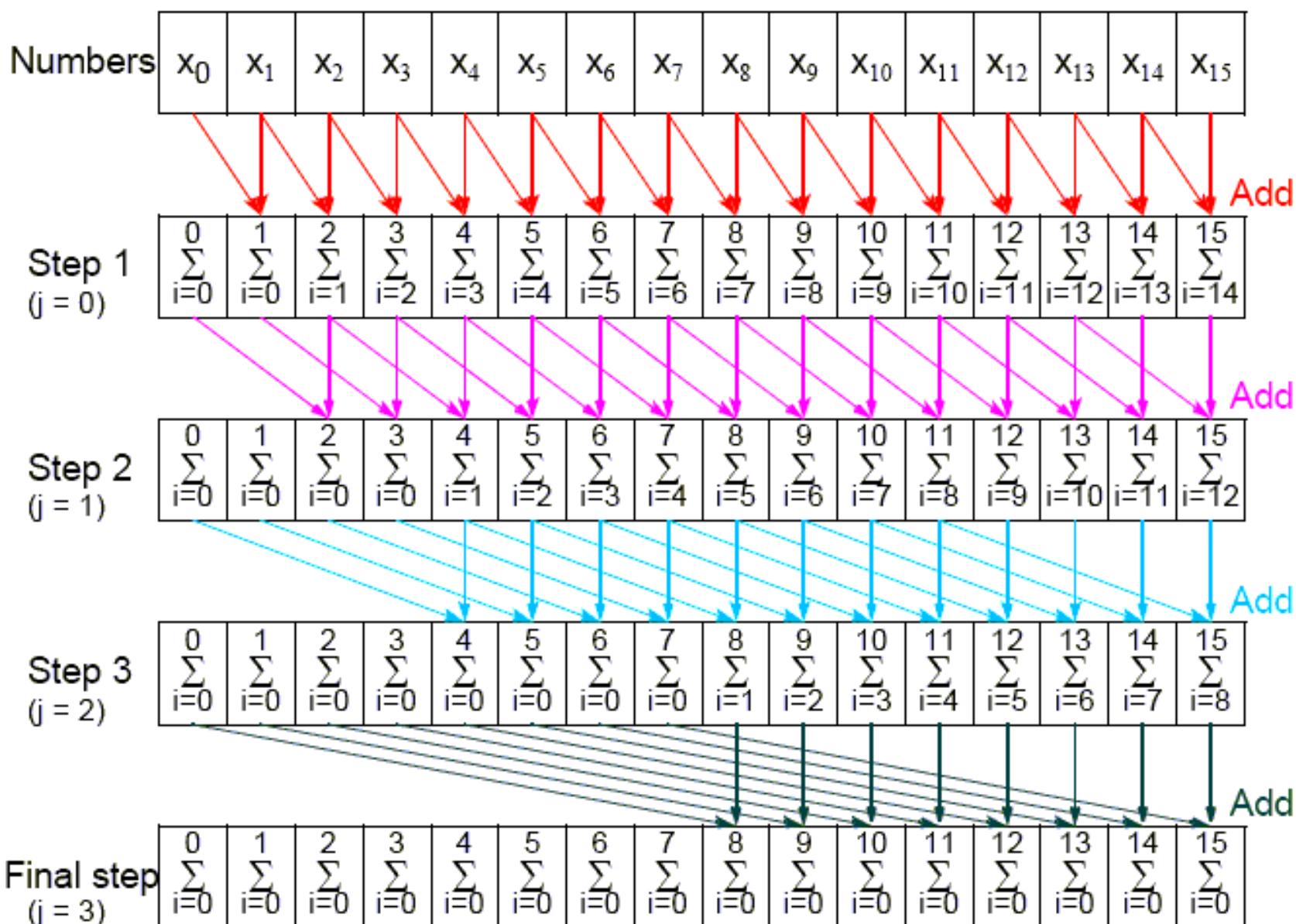
Sequential code

```
for(j = 0; j < log(n); j++) {  
    for (i = 2j; i < n; i++)  
        x[i] = x[i] + x[i-2j];  
}
```

Parallel code

```
for(j = 0; j < log(n); j++) {  
    forall (i = 0; i < n; i++)  
        if (i >= 2j)  
            x[i] = x[i] + x[i-2j];  
}
```

Data parallel prefix sum operation



Another fully synchronous computation example

Solving a General System of Linear Equations by Iteration

Suppose the equations are of a general form with n equations and n unknowns

$$\begin{array}{lll} a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \dots & + a_{n-1,n-1}x_{n-1} & = b_{n-1} \\ . & . & . \\ . & . & . \\ a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \dots & + a_{2,n-1}x_{n-1} & = b_2 \\ a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \dots & + a_{1,n-1}x_{n-1} & = b_1 \\ a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \dots & + a_{0,n-1}x_{n-1} & = b_0 \end{array}$$

where the unknowns are $x_0, x_1, x_2, \dots, x_{n-1}$ ($0 \leq i < n$).

One way to solve these equations for the unknowns is by iteration. By rearranging the i^{th} equation:

$$a_{i,0}x_0 + a_{i,1}x_1 + \dots + a_{i,n-1}x_{n-1} = b_i$$

to

$$x_i = (1/a_{i,i}) [b_i - (a_{i,0}x_0 + a_{i,1}x_1 + \dots + a_{i,n-1}x_{n-1})]$$

or

$$x_i = (1/a_{i,i}) [b_i - \sum_{j \neq i} a_{i,j}x_j]$$

Termination

A simple, common approach. Compare values computed in one iteration to values obtained from the previous iteration.

Terminate computation when all values are within given tolerance; i.e., when

$$|x_i^t - x_i^{t-1}| < \text{error tolerance}$$

for all i , where x_i^t is the value of x_i after the t th iteration and x_i^{t-1} is the value of x_i after the $(t-1)$ th iteration.

However, this does not guarantee the solution to that accuracy.

Sequential Code

```
for (i = 0; i < n; i ++)
    x[i] = b[i];                                /*initialize unknown*/
for (iteration = 0; iteration < limit; iteration++) {
    for (i = 0; i < n; i++){
        sum = 0;
        for (j = 0; j < n; j++)                /* compute summation */
            if (i != j) sum = sum + a[i][j] * x[j];
        new_x[i] = (b[i] - sum) / a[i][i];      /* compute unknown */
    }
    for (i = 0; i < n; i++)
        x[i] = new_x[i];
}
```

Parallel Code

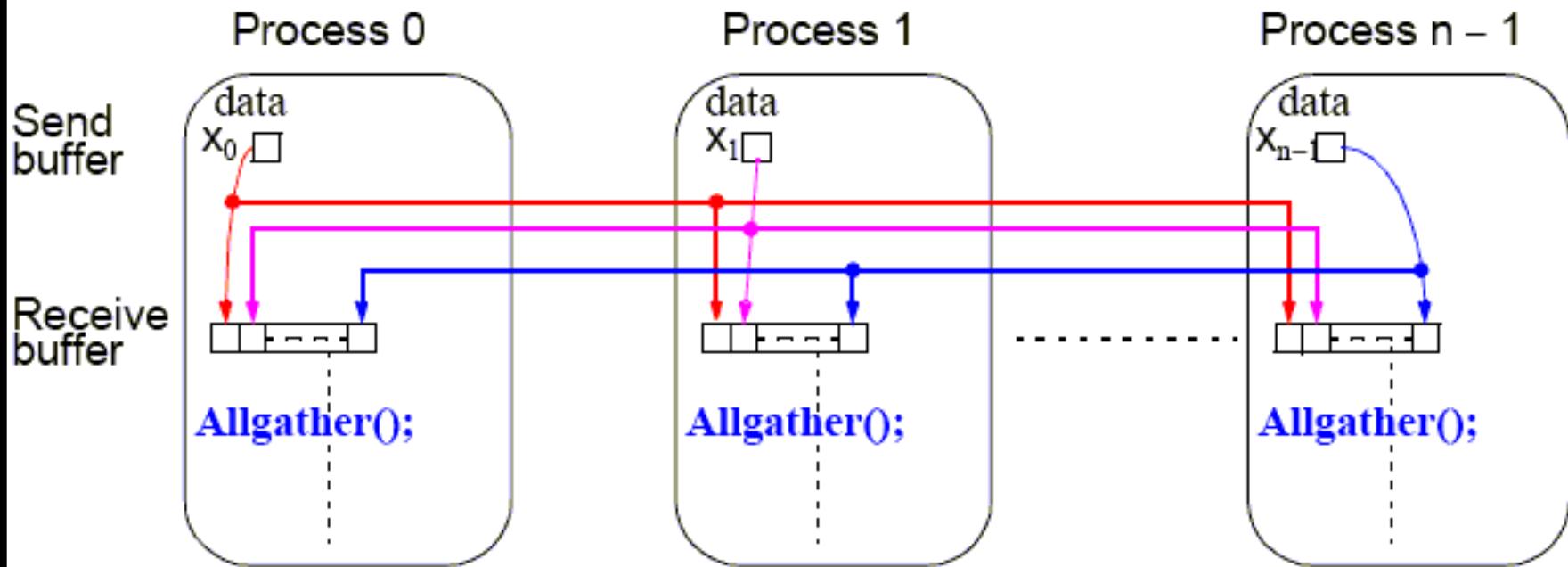
Process P_i could be of the form

```
x[i] = b[i];                                /*initialize unknown*/  
for (iteration = 0; iteration < limit; iteration++) {  
    sum = -a[i][i] * x[i];  
    for (j = 0; j < n; j++)                  /* compute summation */  
        sum = sum + a[i][j] * x[j];  
    new_x[i] = (b[i] - sum) / a[i][i];        /* compute unknown */  
    allgather(&new_x[i]);                     /*bcast or rec values */  
    global_barrier();                         /* wait for all procs */  
}
```

allgather() sends the newly computed value of $x[i]$ from process i to every other process and collects data broadcast from the other processes.

Allgather

Broadcast and gather values in one composite construction.



Textbooks

Ian T. Foster, Designing and Building Parallel Programs, Concepts and tools for Parallel Software Engineering, Addison-Wesley Publishing Company (1995).

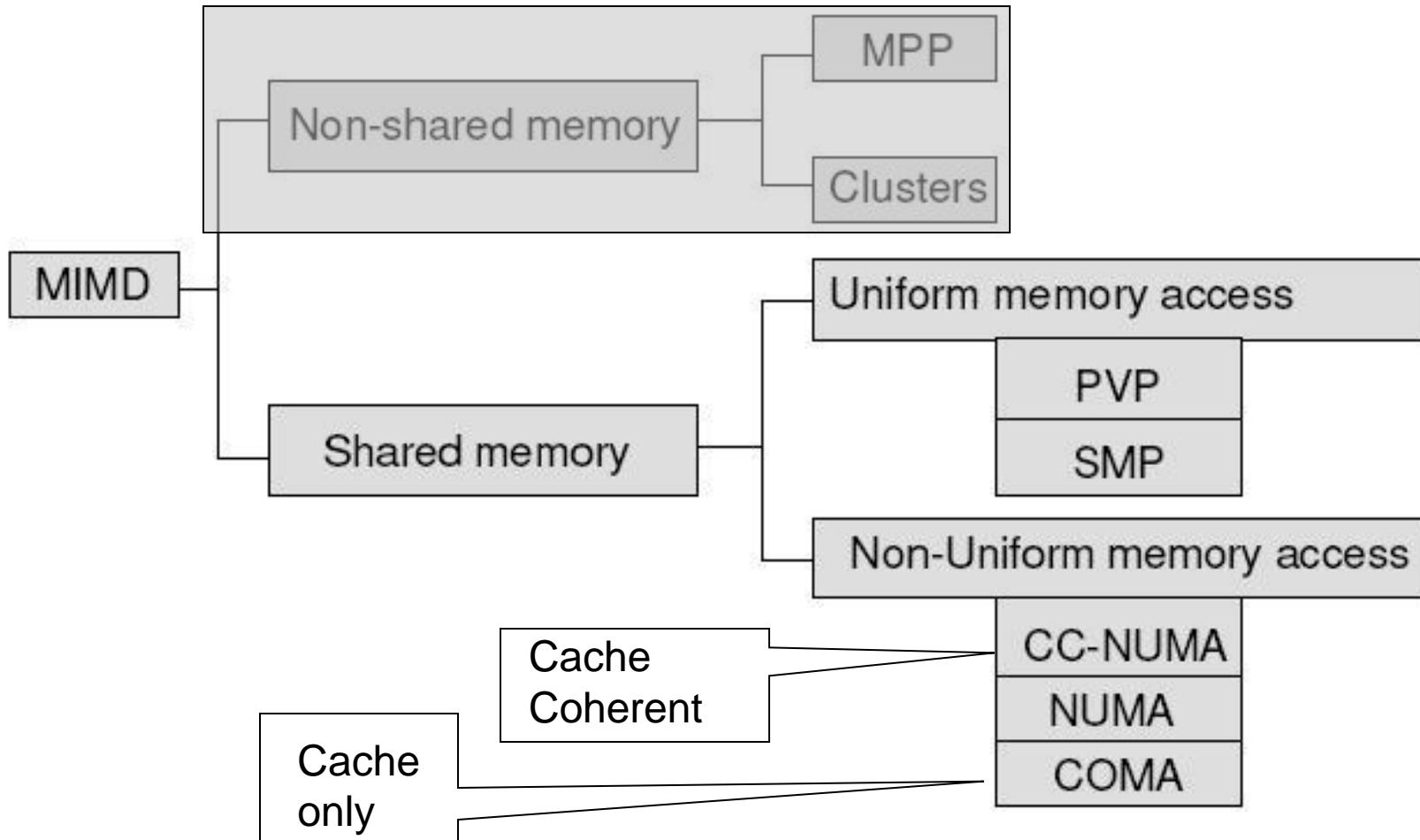
Kai Hwang, Zhiwei Xu, Scalable Parallel Computing (Technology Architecture Programming) McGraw Hill Newyork (1997)

Parallel Programming in C with MPI and OpenMP – Michael J Quinn, Tata-McGraw-Hill Edition, 2003

Barry Wilkinson And Michael Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, Upper Saddle River, NJ, 1999.

Shared Memory Programming

Architectural Model of MIMD



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 is more convenient although it does require access to shared data to be controlled by the programmer (using critical sections etc.)

Important Alternatives for Programming ShMemMps

- Threads ([Pthreads](#), Java, ..) in which the programmer decomposes the program into individual parallel sequences, each being thread, and each being able to access variables declared outside the threads.
- A sequential programming language with [preprocessor compiler directives](#) to declare shared variables and specify parallelism. Example: [OpenMP](#)
- C++ runtime library TBB Stands for “Threading Building Block”

Pthreads

IEEE Portable Operating System Interface for UniX, POSIX,
sec. 1003.1c (1995) standard

The POSIX standard (IEEE 1003.1c) defines
the *specification* for Pthread, not the *implementation*

In Pthread, the main program is a thread itself. A separate
thread is created and terminated with the routine itself.

```
pthread_t thread1;          /*handle of special Pthread data type*/  
pthread_create(&thread1, NULL, (void *)proc1, (void *)&argv);  
pthread_join(thread1, void *status);
```

A thread ID or handle is assigned and obtained from &thread

Thread Basics

- Thread operations include thread creation, termination, synchronization (joins, blocking), scheduling, data management and process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.
- Threads in the same process share:
 - Process instructions
 - Most data
 - open files (descriptors)
 - signals and signal handlers
 - current working directory
 - User and group id

Thread Basics

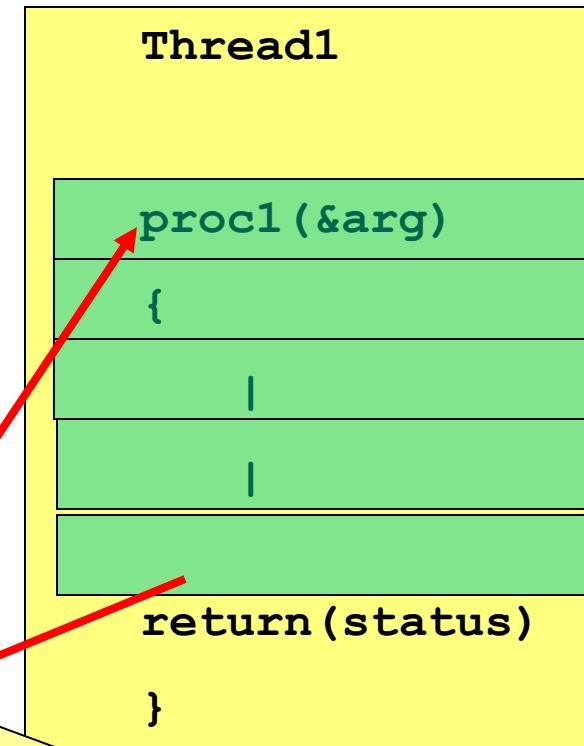
- Each thread has a unique:
 - Thread ID
 - set of registers, stack pointer
 - stack for local variables, return addresses
 - signal mask
 - Priority
 - Return value: errno
- pthread functions return "0" if OK.

Execution of a Pthread thread

Main program

```
pthread_t thread1;  
pthread_create(&thread1, NULL, proc1, &arg);
```

```
pthread_join(thread1, *status);
```



If attr is NULL, the default attributes are used
Upon successful completion,
pthread_create()
stores the ID of the created
thread in the location referenced by
thread1

A thread can obtain its ID by routine pthread_self().

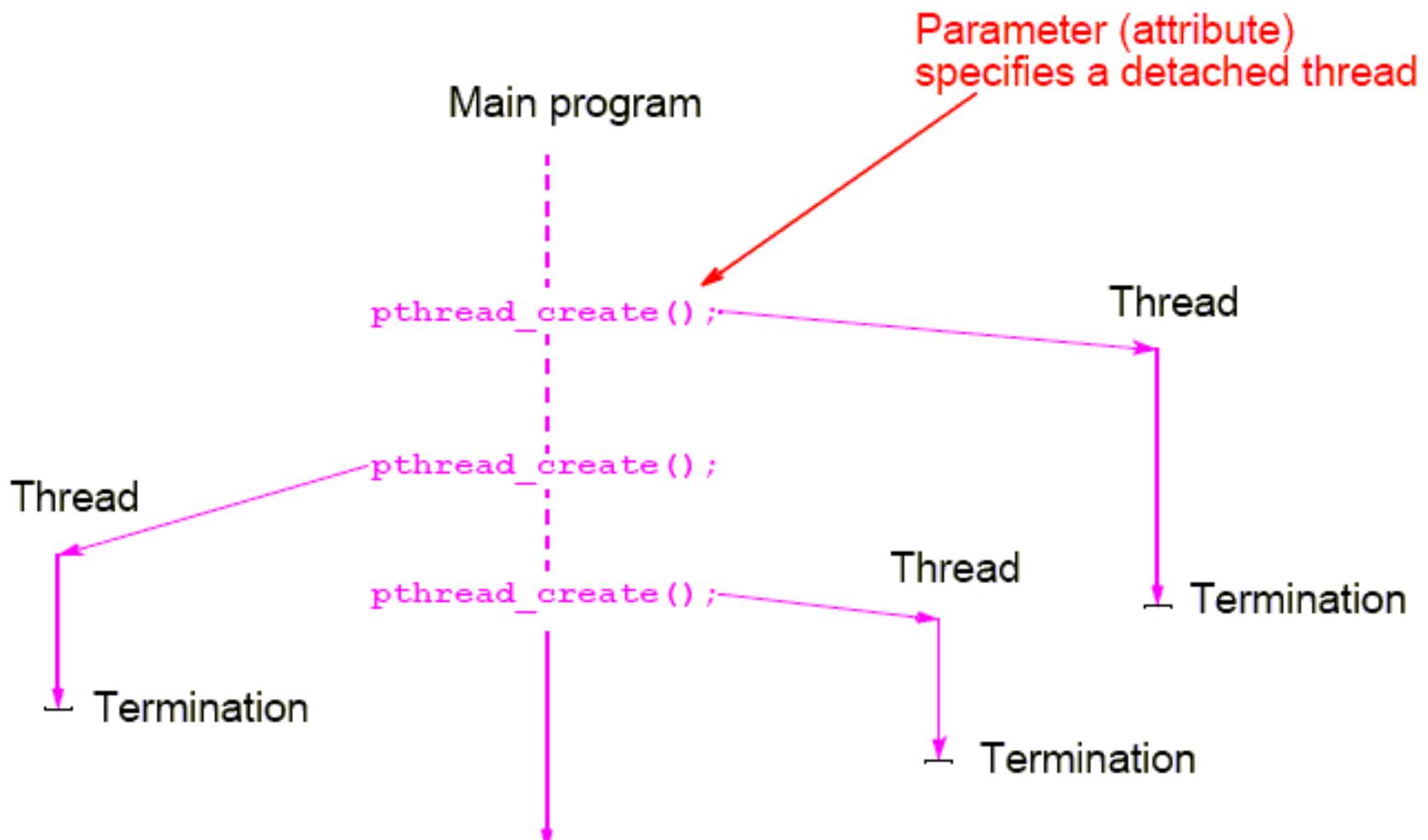
Detached Threads

It may be the case that threads are not bothered when a thread creates terminates and then a **join not needed**.

Threads not joined are called *detached threads*.

When detached threads terminate, they are destroyed and their resource released.

Pthreads Detached Threads



Thread-Safe Routines

Routines are thread safe if they can be called from multiple threads simultaneously and always produce correct results.

Standard I/O routines are thread safe (prints messages without interleaving the characters).

System routines that return time **may not be thread safe**.

Routines that access shared data may require special care to be made thread safe.

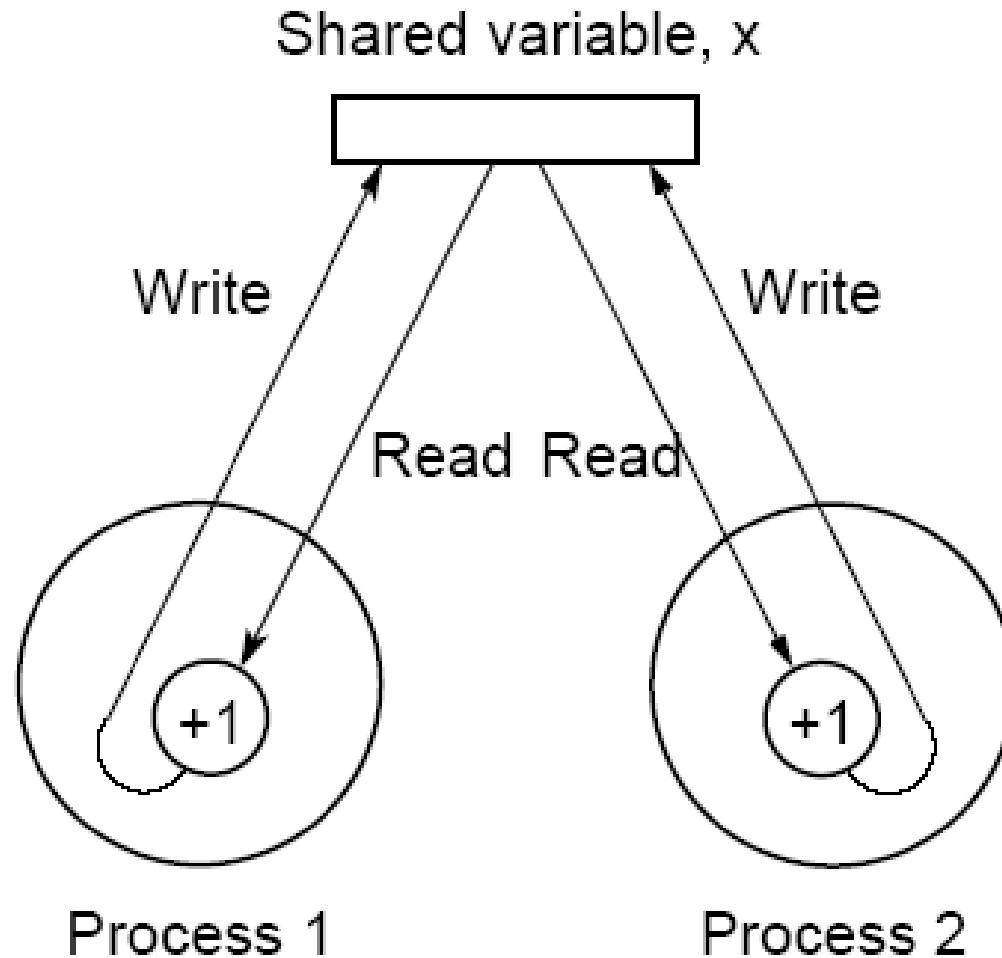
Accessing Shared Data

Accessing shared data needs careful control.

Consider two processes each of which is to add one to a shared data item, x . Necessary for the contents of the location x to be read, $x + 1$ computed, and the result written back to the location

Instruction	Process 1	Process 2
$x = x + 1;$	read x	read x
	compute $x+1$	compute $x+1$
	write to x	write to x

Conflict in accessing shared variable



Critical Section

A mechanism for ensuring that only one process accesses a particular resource at a time is to establish sections of code involving the resource as so-called *critical sections* and make sure that only one such critical section is executed at a time for that resource.

This mechanism is known as *mutual exclusion*.

We learn this concept in an operating system course. Here we simply review in context of shared memory programming.

Locks

Simplest mechanism for ensuring mutual exclusion of critical sections.

A lock is a 1-bit variable that is a **1** to indicate that a process has entered the critical section and a **0** to indicate that no process is in the critical section.

Operates much like that of a door lock:

Locks

A process coming to the “door” of a critical section and finding it open may enter the critical section, locking the door behind it to prevent other processes from entering. Once the process has finished the critical section, it unlocks the door and leaves.

Binary semaphores can achieve similar results, but usually locks are sufficient.

Mutexes can be applied only to threads in a single process and do not work between processes as do semaphores.

Pthread Routines

Pthreads API can be informally grouped into three major classes:

Thread management: The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)

Mutexes: The second class of functions deal with **synchronization**, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provided for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

Pthread Routines

Condition variables: The third class of functions address communications between threads that **share a mutex**. They are based upon programmer specified conditions.

This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Mutexes

- Mutexes are used to **prevent data inconsistencies due to race conditions.**
- A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results depends on the order of operations
- Mutexes are used for **serializing shared resources.**
- Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it.
- One can apply a mutex to protect a segment of memory ("critical region") from other threads.

Pthread Lock Routines

Locks are implemented in Pthreads with *mutually exclusive lock variables*, or “mutex” variables:

`pthread_mutex_lock(&mutex1);`

critical section

`pthread_mutex_unlock(&mutex1);`

If a thread reaches a mutex lock and finds it locked, it will wait for the lock to open. If more than one threads are waiting for the lock to open when it opens, the system will select one thread to be allowed to proceed. Only the thread that locks a mutex can unlock it.

Mutexes

Without Mutex

```
int counter=0;

/* Function C */
void function C()
{
    counter++

}
```

Mutexes

Without Mutex

```
int counter=0;  
  
/* Function C */  
void function C()  
{  
  
    counter++  
  
}
```

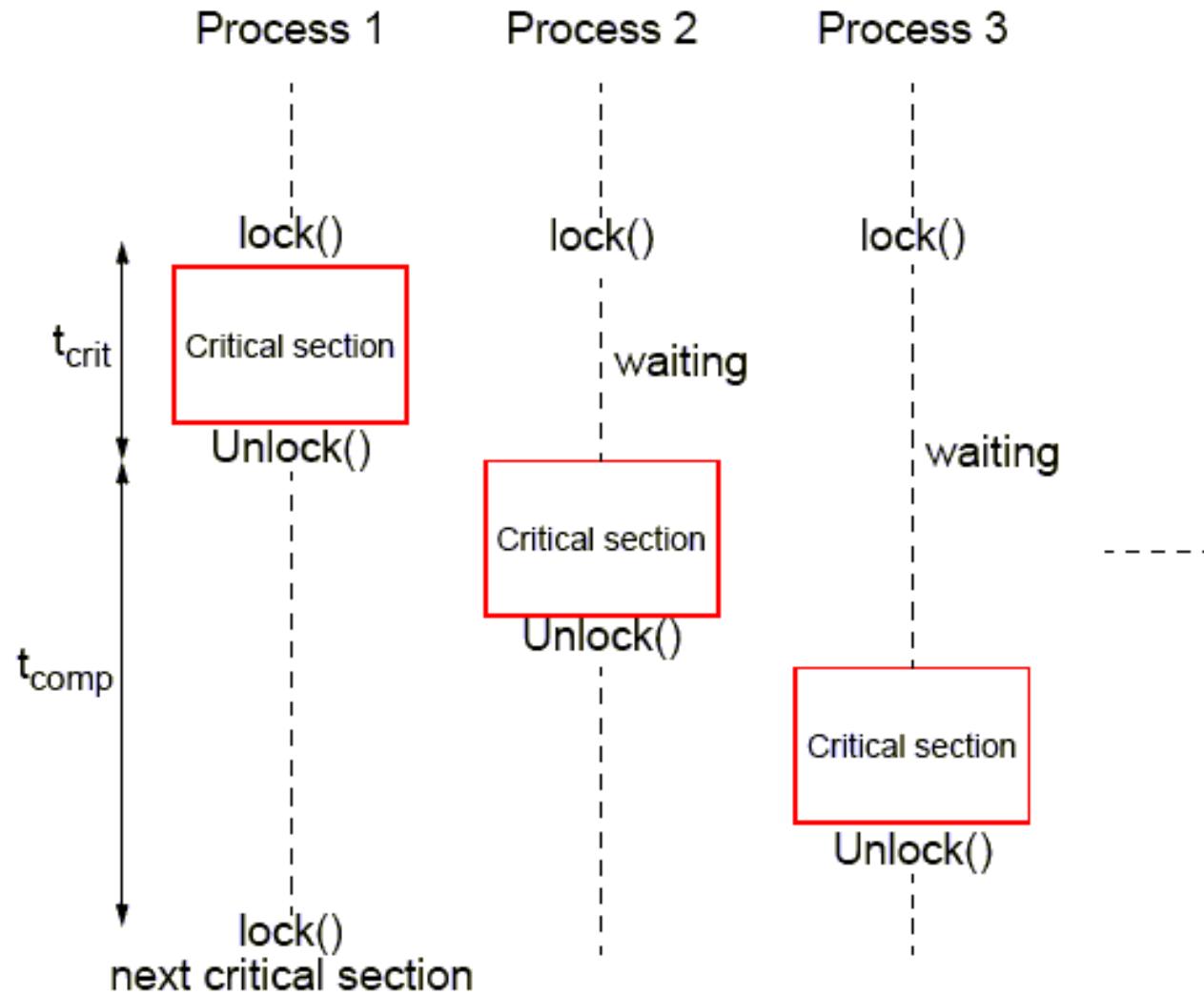
With Mutex

```
pthread_mutex_t mutex1 =  
PTHREAD_MUTEX_INITIALIZER;  
int counter=0;  
  
/* Function C */  
void function C()  
{  
  
    pthread_mutex_lock( &mutex1 );  
    counter++  
  
    pthread_mutex_unlock( &mutex1 );  
}
```

Critical Sections Can Serialize Code

- High performance programs should have **as few as possible** critical sections as their use can serialize the code.
- Suppose, all processes happen to come to their critical section together.
- They will execute their critical sections one after the other.
- In that situation, the execution time becomes almost that of a single processor.

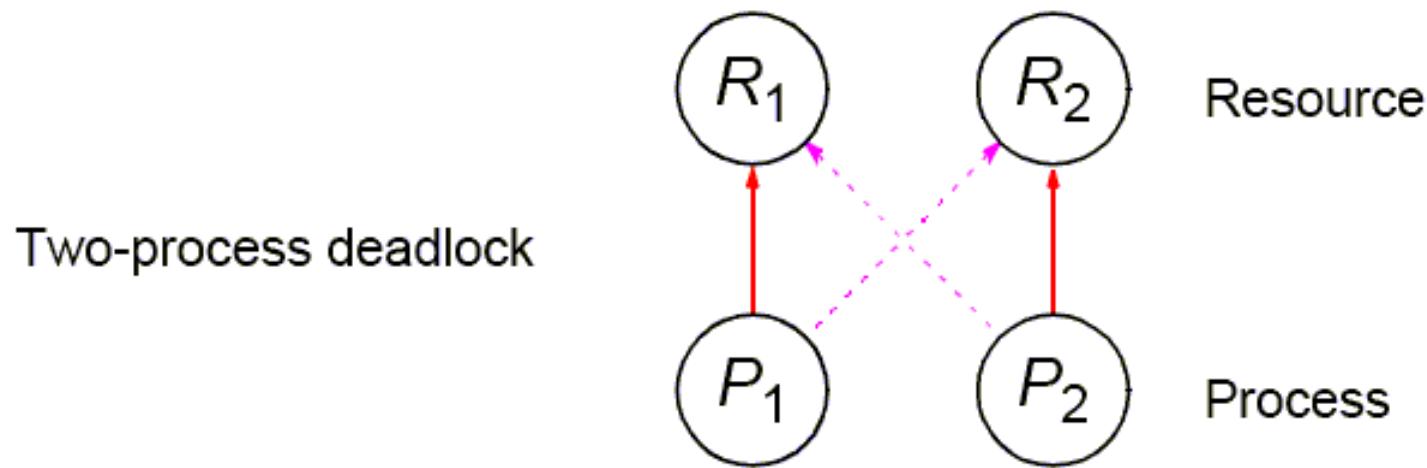
Illustration



When $t_{comp} < p t_{crit}$, less than p processor will be active

Deadlock

Can occur with two processes when one requires a resource held by the other, and this process requires a resource held by the first process.



Can also occur in a circular fashion with several processes having a resource wanted by another.

Pthreads

Offers one routine that can test whether a lock is actually closed without blocking the thread:

pthread_mutex_trylock()

This routine will lock an unlocked mutex and return **0** or will return with **EBUSY** if the mutex is already locked – might find a use in overcoming deadlock.

Monitor

Suite of procedures that provides only way to access shared resource. Only one process can use a monitor procedure at any instant.

Could be implemented using a semaphore or lock to protect its entry; i.e.,

```
monitor_proc1()
{
    lock(x);
    .
    monitor body
    .
    unlock(x);
    return;
}
```

Java, of course, has this built-in.

Condition Variables

- In a critical section (i.e. where a mutex has been used), a thread can suspend itself on a *condition variable* if the state of the computation is not right for it to proceed.
 - It will suspend by *waiting* on a condition variable.
 - **It will, however, release the critical section lock (mutex) .**
 - When that condition variable is *signaled*, it will become ready again; it will attempt to reacquire **that critical section lock** and only then will be able proceed.
- With Posix threads, a condition variable can be associated with only one mutex variable!

Condition Variables

Often, a critical section is to be executed if a specific global condition exists; for example, if a certain value of a variable has been reached.

With locks, the global variable would need to be examined at frequent intervals (“polled”) within a critical section.

This is a very time-consuming and unproductive exercise.

Can be overcome by introducing so-called *condition variable*

Condition Variables

Main Thread

- o Declare and initialize global data/variables which require synchronization (such as "count")
- o Declare and initialize a condition variable object
- o Declare and initialize an associated mutex
- o Create threads A and B to do work

Thread A

- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call `pthread_cond_wait()` to perform a blocking wait for signal from Thread-B. *Note that a call to `pthread_cond_wait()` automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.*
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

Thread B

- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue

Main Thread

Join / Continue

Pthread Condition Variables

Associated with a specific mutex. Given declarations:

```
pthread_cond_t cond1;
pthread_mutex_t mutex1;
pthread_cond_init(&cond1, NULL);
pthread_mutex_init(&mutex1, NULL);
```

the Pthreads arrangement for signal and wait is as follows:

```
action()
{
    .
    .
    .
    pthread_mutex_lock(&mutex1);
    while (c <> 0)
        pthread_cond_wait(cond1, mutex1); ←
    pthread_mutex_unlock(&mutex1);
    take_action();
    .
    .
}

counter()
{
    .
    .
    .
    pthread_mutex_lock(&mutex1);
    c--;
    if (c == 0) pthread_cond_signal(cond1);
    pthread_mutex_unlock(&mutex1);
    .
    .
}
```

Signals are *not* remembered - threads must already be waiting for a signal to receive it.

Thread Interaction Primitives in Pthreads

Function	Meaning
<code>pthread_mutex_init(...)</code>	Creates a new mutex variable
<code>pthread_mutex_destroy(...)</code>	Destroy a mutex variable
<code>pthread_mutex_lock(...)</code>	Lock (acquire) a mutex variable
<code>pthread_mutex_trylock(...)</code>	Try to acquire a mutex variable
<code>pthread_mutex_unlock(...)</code>	Unlock (release) a mutex variable
<code>pthread_cond_init(...)</code>	Creates a new conditional variable
<code>pthread_cond_destroy(...)</code>	Destroy a conditional variable
<code>pthread_cond_wait(...)</code>	Wait (block) on a conditional variable
<code>pthread_cond_timedwait(...)</code>	Wait on a conditional variable up to a time limit
<code>pthread_cond_signal(...)</code>	Post an event, unlock one waiting process
<code>pthread_cond_broadcast(...)</code>	Post an event, unlock all waiting processes

The POSIX Threads (Pthreads) Model

Function Prototype	Meaning
<pre>int pthread_create(pthread_t* thread_id, pthread_attr_t* attr, void* (*myroutine)(void*), void* arg)</pre>	Create a thread
<pre>void pthread_exit(void* status)</pre>	A thread exits
<pre>int pthread_join(pthread_t thread, void** status)</pre>	Join a thread
<pre>pthread_t pthread_self(void)</pre>	Returns the calling thread ID

Let us write Pthread & MPI Pthread programs

```

#include <pthread.h>

void *message(char *s)

{
    printf(" %s", s);
    return;
}

main(int argc, char *argv[])
{
    pthread_t thread1, thread2, thread3;
    pthread_attr_t x;           /*thread attribute object*/
    pthread_attr_init(&x);    /*Initializes a thread attribute to a default value*/
    pthread_create(&thread1, &x, (void *(*)(void *)) message, (void *) "Central");
    pthread_create(&thread2, &x, (void *(*)(void *)) message, (void *) "University");
    pthread_create(&thread3, &x, (void *(*)(void *)) message, (void *) "Hyderabad");
    /*Join the threads*/
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    pthread_attr_destroy(&x);
    printf("\n");
}

```

Compile and run on 14.139.69.97

- nano pthello.c
- gcc -o pthello pthello.c -pthread
- ./pthello

Compile and run on 14.139.69.93

- `nano pthello.c`
- `gcc -o pthello pthello.c -lpthread`
- `./pthello`

```
#include <pthread.h>
#include <mpi.h>
#include <stdio.h>

int MyRank, NumProcs;

void * Message(int MyID)
{
    printf("Hello World! from Thread:%d on Process: %d. \n",
           MyID, MyRank);

    return;
}

void main(int argc, char *argv[])
{
    pthread_t thread1, thread2;
    MPI_Status status;
```

```
MPI_Init(&argc, &argv) ;  
MPI_Comm_rank(MPI_COMM_WORLD, &MyRank) ;  
MPI_Comm_size(MPI_COMM_WORLD, &NumProcs) ;  
  
pthread_create(&thread1, NULL, (void *(*) (void *))  
    Message, (void *) 1);  
pthread_create(&thread2, NULL, (void *(*) (void *))  
    Message, (void *) 1);  
  
pthread_join(thread1, NULL);  
pthread_join(thread2, NULL);  
MPI_Finalize();  
return;  
}
```

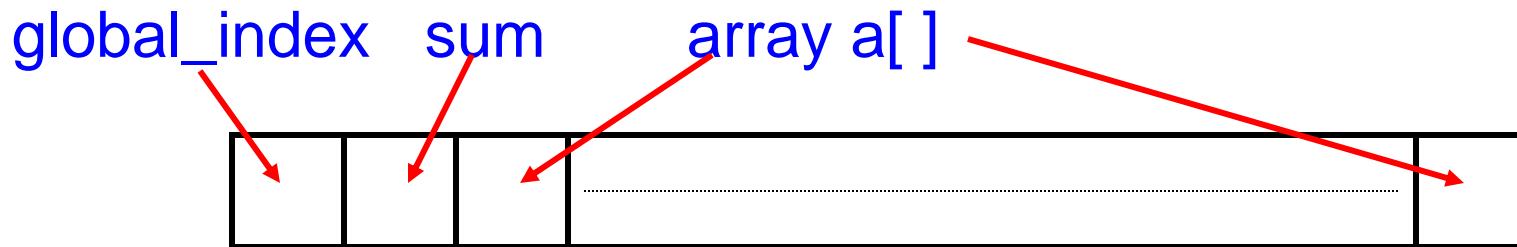
Compile and run on 14.139.69.97

- `nano ptmpi.c`
- `mpicc -o ptmpi ptmpi.c -pthread`
- `mpirun -np 10 ptmpi`

Assignment

N thread are to be created, each taking numbers from the list to add to their partial sums. When all numbers have been taken, the threads can add their partial results to a shared location **sum**. A shared location **global_index** can be used by each thread to select next element of **a[]**.

Shared variables



You may use `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_join`, `pthread_mutex_init` etc.

Resources for further reading

- <https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html>
- <https://computing.llnl.gov/tutorials/pthreads/#Thread>

Numerical Integration: Pthreads (1 of 3)

f(): the function returns $4/(1+a^2)$

```
#include <pthread.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
pthread_mutex_t mutex1;
pthread_t *tid;
int n, num_thread;
double pi, h;
```

Lock variable

Thread attribute

Intervals & number of threads

Global variables on heaps

```
double f(a)
double a;
{
    return(4.0 / (1.0 + a*a));
}
```

Numerical Integration: Pthreads (2 of 3)

mypical(): the function run by the threads

```
void *mypical(void *arg) {  
    int i, myid;  
    double sum, mypi, x;  
  
    myid = *(int*)arg; h = 1.0 / (double) n; sum = 0.0;  
    for (i = myid + 1; i <= n; i += num_thread)  
    {  
        x = h * ((double)i - 0.5);  
        sum += f(x);  
    }  
    mypi = h * sum;  
    pthread_mutex_lock(&mutex1);  
    pi += mypi;  
    pthread_mutex_unlock(&mutex1);  
  
    return(0);  
}
```

Argument used for thread ID passed by the pthread_create, gives integer pointer pointing to

Cyclic loop distribution

Put any code you want in between the Mutex_lock and unlock. This is called a **Critical section** ... only one thread at a time can execute this code

Numerical Integration: Pthreads (3 of 3)

mypical(): the function run by the threads

```
void main (int argc, char **argv)
{
    int i; double PI = 3.14159265; int tRank[num_thread];
    pthread_mutex_init(&mutex1,NULL);
    printf("Enter the number of threads you want to create : ");
    scanf("%d",&num_thread);
    tid =(pthread_t *)malloc(num_thread * sizeof(pthread_t));
    printf("Enter the number of intervals you want ");
    scanf("%d",&n);
    for(i=0;i<num_thread;i++)
    {
        tRank[i] = i;
        pthread_create(&tid[i],NULL, mypical, &tRank[i]);
    }
    for(i=0;i<num_thread;i++)
        pthread_join(tid[i],NULL);
    pthread_mutex_destroy(&mutex1);
    printf("pi is approximately %.16f, Error is %.16f\n",pi, fabs(pi- PI));
}
```

Argument used for thread ID passed by the pthread_create

Initializing Mutex

Create space

Create (fork) the threads ... passing each thread its rank

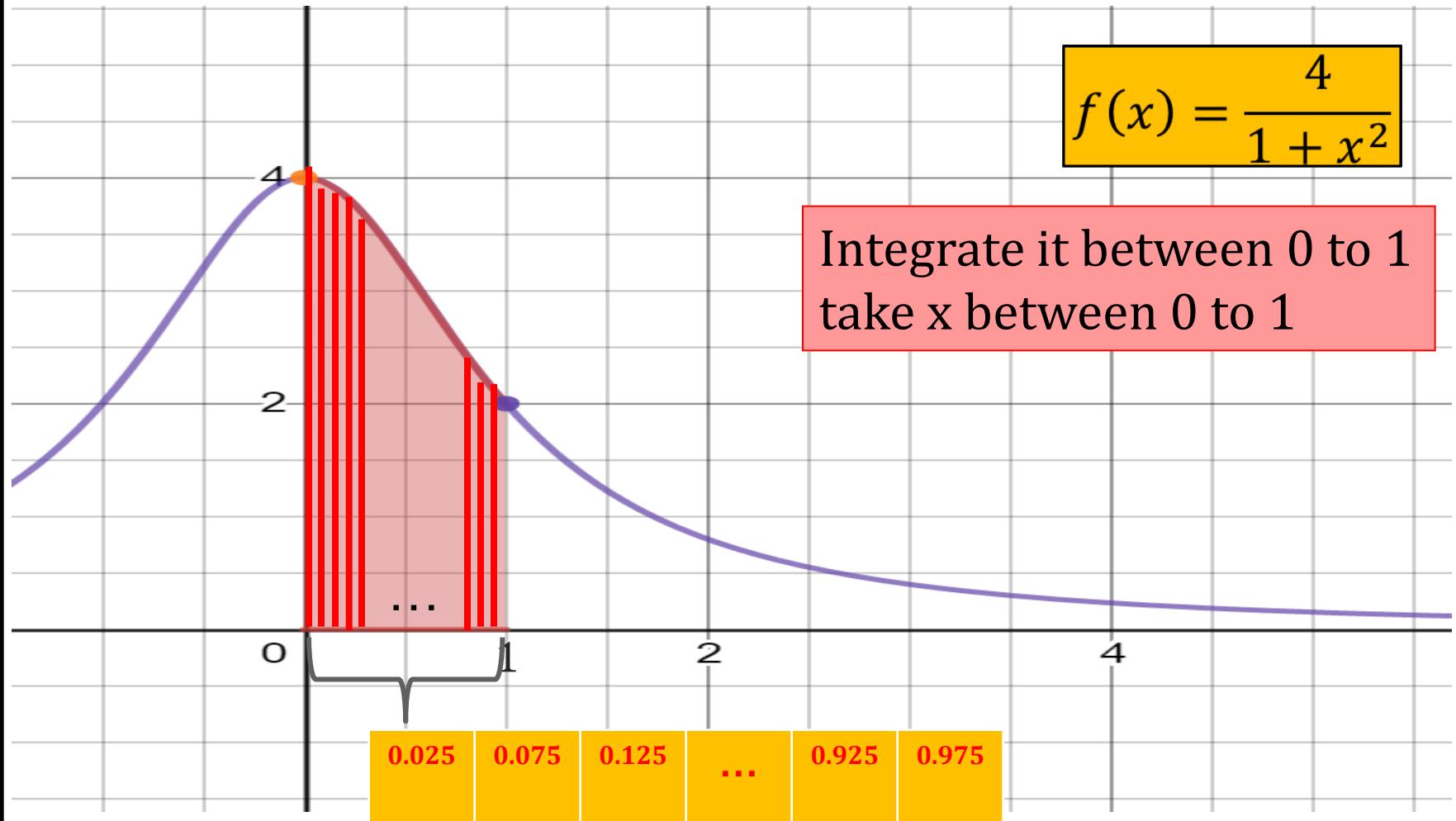
Post a join for each thread hence waiting for all of them to finish before proceeding

Destroy Mutex lock

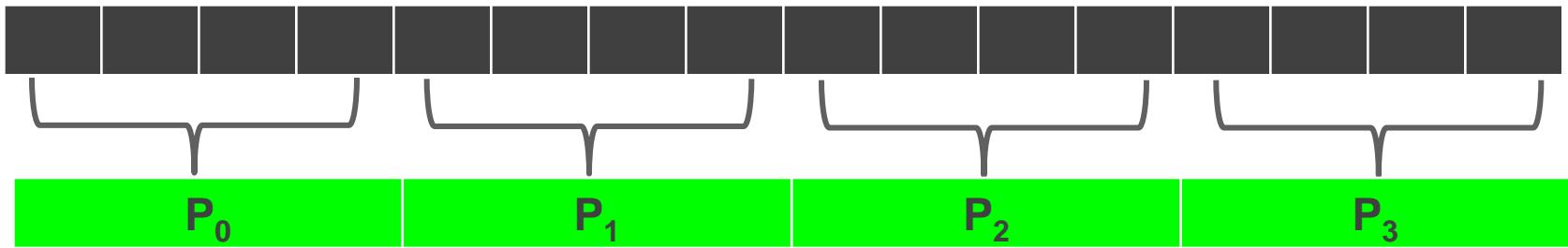
How to run

- login to cluster system (14.139.69.97)
- nano pthreadapi.c -c
- gcc -o pthreadapi pthreadapi.c -pthread
- ./pthreadapi

Numerical Integration (pi calculations): Mix mode MPI/pthread

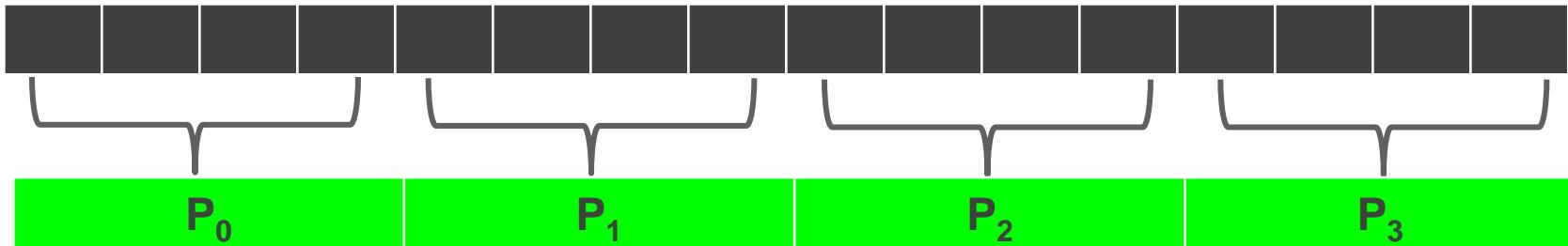


pi calculation: Mix mode MPI/pthread

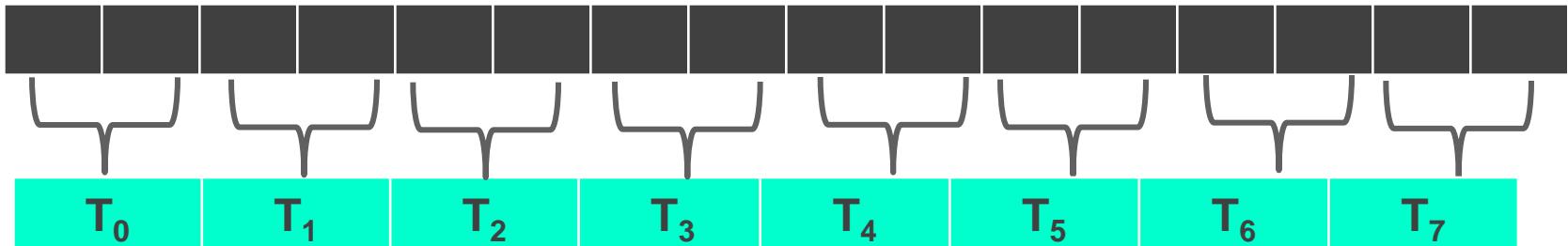


- Course grain implementation using MPI

pi calculation: Mix mode MPI/pthread

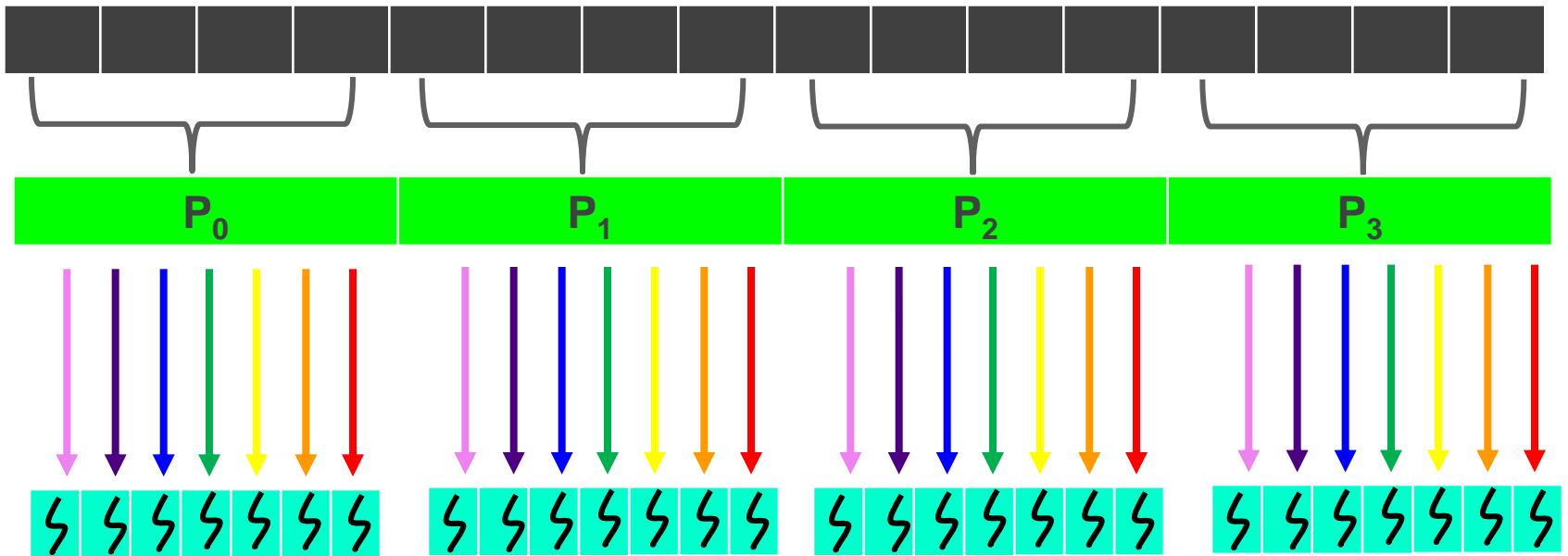


- Course grain implementation using MPI



- Fine grain implementation using pthread

pi calculation: Mix mode MPI/pthread



- Implementation using MPI + pthread

pi calculation: Mix mode MPI/pthread-01

- Input (interval) is used to find step size $h = 1/\text{interval}$
- num_proc denotes number of processes (processors) used to execute program
- Every process gets $\text{interval}/\text{num_proc}$ portion to perform task
- NUM_THREADS denotes the number of threads per process
- We assume that (easy modification is possible to remove these constraints, left as an exercise)
 - interval is exactly divisible by num_proc .
 - NUM_THREADS is always less than $\text{interval}/\text{num_proc}$
- Inside the local portion at a process, threads get work using cyclic rotation

pi calculation: Mix mode MPI/pthread-02

```
#include <mpi.h>
#include <math.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define NUM_THREADS 6
pthread_mutex_t mutex1;
int tRank [NUM_THREADS];
double localpi;
double pi, h;
```

Used for sleep() in threads, pause the execution for few milliseconds

Setting the number of threads per process

Lock variable for threads

Local value of pi at a processes obtained by threads

pi calculation: Mix mode MPI/pthread-03

```
double pi, h;
```

```
struct thread_data {
```

```
    int myrank;
```

```
    int nbyp;
```

```
    int interval;
```

```
    int *tRank;
```

```
};
```

```
struct thread_data
```

```
thread_data_array[NUM_THREADS];
```

Defining a structure to pass to a procedure, with which threads are created

Passing rank of a process on which threads are created

Portion of every process

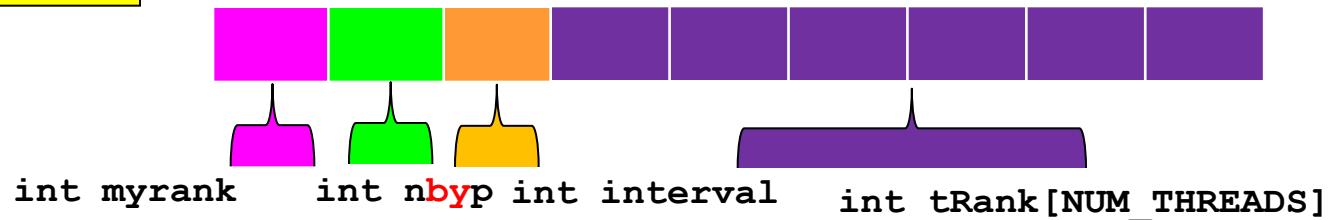
Total interval

Array holding index of threads/process

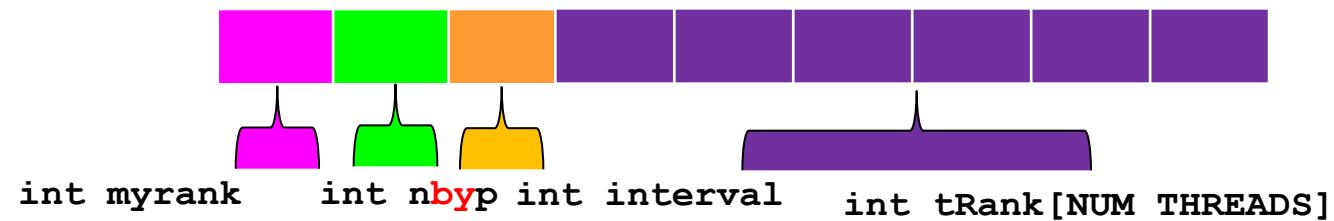
Array of type thread data

```
struct thread_data {  
    int myrank;  
    int nbyp;  
    int interval;  
    int *tRank;  
};
```

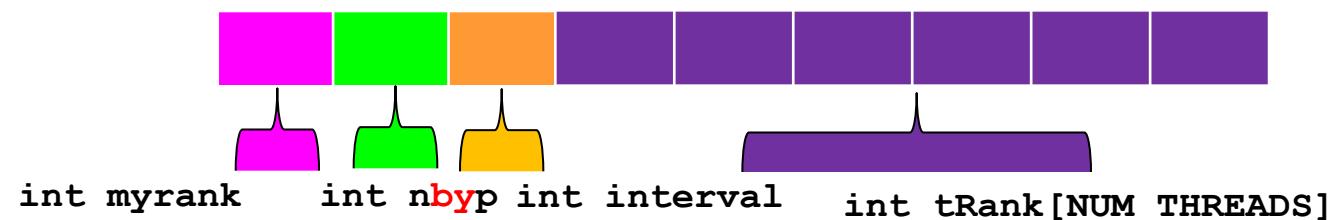
thread_data_array[0]



thread_data_array[1]



thread_data_array[2]



pi calculation: Mix mode MPI/pthread-04

```
double f(a)
double a;
{
    return(4.0 / (1.0 + a*a));
}
void *mypical(void *arg) {
    int pid, i, y;
    int m; int n;
    int *ttrank;
    struct thread_data *my_data;
    double sum, mypi, x;
```

Original function to be called by the thread

Name of the procedure called by all threads

Array of thread ranks

Structure type variable

mypi is local pi calculated by thread

pi calculation: Mix mode MPI/pthread-05

```
double f(a)
double a;
{
    return(4.0 / (1.0 + a*a));
}
void *mypical(void *arg) {
    int pid, i, y;
    int m; int n;
    int *ttrank;
    struct thread_data *my_data;
    double sum, mypi, x;
```

Original function to be called by the thread

Name of the procedure called by all threads

Array of thread ranks

Structure type variable

mypi is local pi calculated by thread

pi calculation: Mix mode MPI/pthread-06

```
sleep(1);  
  
my_data = (struct thread_data *) arg;  
pid = my_data->myrank;  
m = my_data -> nbyp;  
n = my_data -> interval;  
ttrank = my_data->tRank;
```

Passed arguments will
be available in my_data

pid: process rank, m:
interval/num_proc, n:interval,
ttrank is thread's rank

```
h = 1.0 / (double) n;
```

Step size

pi calculation: Mix mode MPI/pthread-07

```
y = *ttrank;  
for (i = ((y + 1) + (m*pid));  
     i <= (m * (pid + 1));  
     i += NUM_THREADS)  
{  
    x = h * ((double)i - 0.5);  
    sum += f(x);  
}  
mypi = h * sum;  
pthread_mutex_lock(&mutex1);  
    localpi += mypi;  
pthread_mutex_unlock(&mutex1);  
pthread_exit(NULL);  
return 0; }
```

Assign to every thread work in cyclic fashion inside a process

Calculate the area under the curve for given x and sum

Local pi calculation

Sum of pi to local to that process

pi calculation: Mix mode MPI/pthread-08

```
int main(int argc, char *argv[]) {
```

Main program where we declare all variables

```
    int i, num_proc, rank;
```

```
    double PI = 3.14159265;
```

```
    pthread_t threads[NUM_THREADS];
```

```
    int rc, t, sum;
```

```
    int interval, nbyp;
```

```
    double pi, h;
```

pthread_mutex_init(&mutex1,NULL);
Mutex lock variable to stop multiple thread to execute critical section

```
    pthread_mutex_init(&mutex1,NULL);
```

```
    double startwtime, endwtime;
```

Clock variable to calculate total execution time

pi calculation: Mix mode MPI/pthread-09

```
if (rank == 0)
{
    printf("Please enter the intervals:\n");
    scanf("%d", &interval);
    startwtime = MPI_Wtime();
}

MPI_Bcast(&interval, 1,
MPI_INT, 0, MPI_COMM_WORLD);

for(t=0;t< NUM_THREADS;t++)
    tRank[t] = (int)t;
```

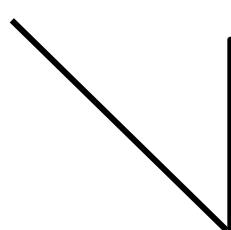
Asking the interval value to be given

Broadcasting interval to all processes

Giving ranks to threads

pi calculation: Mix mode MPI/pthread-10

```
for(t=0;t< NUM_THREADS;t++) {  
  
    thread_data_array[t].myrank = rank;  
    thread_data_array[t].tRank = tRank + t;  
    thread_data_array[t].nbyp =  
        interval/num_proc;  
    thread_data_array[t].interval = interval;
```



Passing different variables
as process rank, thread
rank array, portion of each
processes and total interval

pi calculation: Mix mode MPI/pthread-11

```
rc = pthread_create(&threads[t], NULL,  
myiscal, (void *) &thread_data_array[t]);  
  
if (rc) {  
    printf("ERR; pthread_create() ret =  
%d\n", rc);  
    exit(-1);  
}  
}
```

Creating threads

pi calculation: Mix mode MPI/pthread-12

```
for(i=0;i<NUM_THREADS;i++) {  
    pthread_join(threads[i],NULL);  
    pthread_mutex_destroy(&mutex1);  
}  
  
MPI_Reduce(&localpi, &pi, 1, MPI_DOUBLE,  
MPI_SUM, 0, MPI_COMM_WORLD);
```

Joint the threads and
destroy lock

MPI Reduce to collect
results from different
processes

pi calculation: Mix mode MPI/pthread-13

```
if (rank == 0)
{
    printf("pi is approximately %.16f,
Error is %.16f\n",pi, fabs(pi- PI));
    endwtime = MPI_Wtime();
    printf("wall clock time = %f\n",
endwtime-startwtime);
}
MPI_Finalize();
```

Producing results

Pithreadmpi.c

How to run

- ssh to cluster system
- nano pithreadmpi.c -c
- mpicc -o pithreadmpi pithreadmpi.c -pthread
- mpirun -np 10 pithreadmpi

Implicit/Explicit Parallel Programming Models

Implicit parallel programming models

Automatic Parallelization of sequential programs using compiler technology.

Explicit parallel programming models

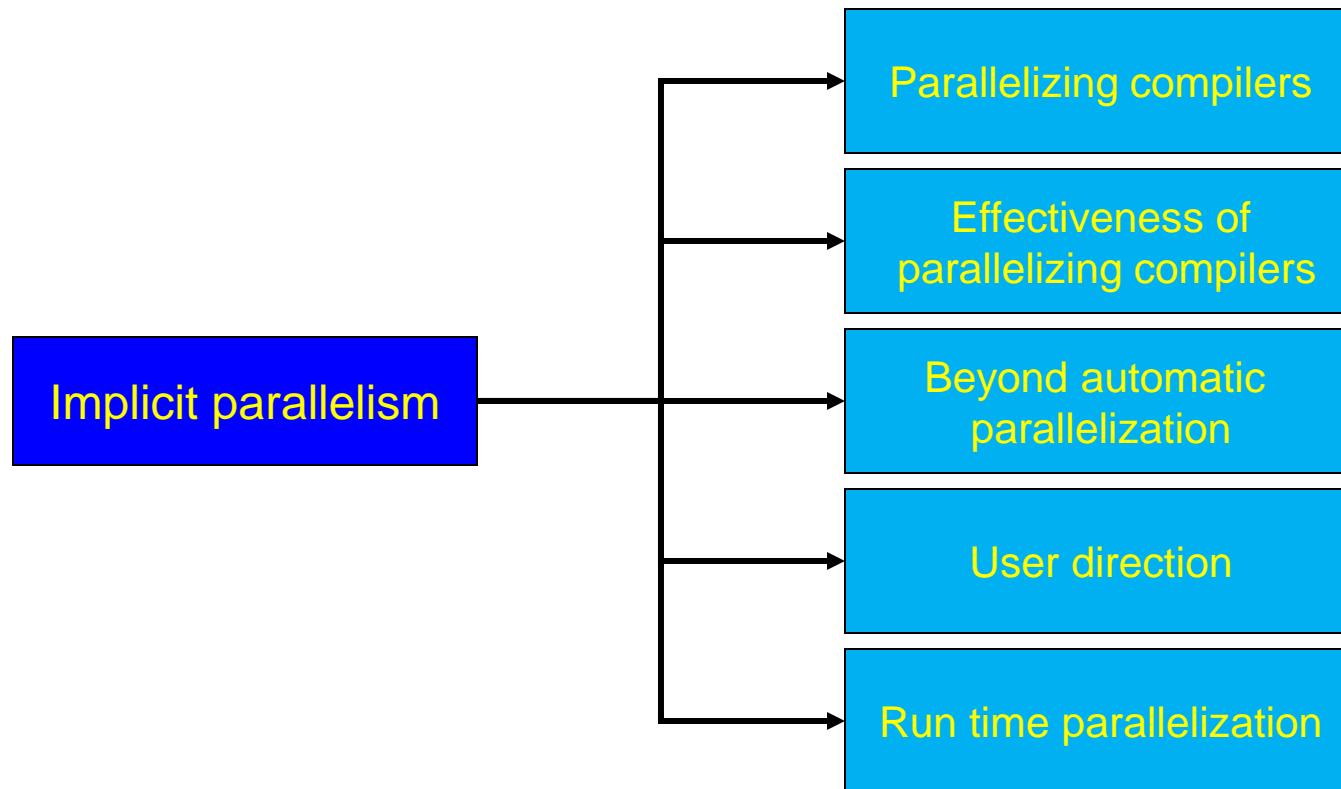
Three dominant parallel programming models are :

- ✗ • Data-parallel model (f90/HPF)
- ✓ • Message-passing model (MPI/PVM)
- ✓ • Shared-variable model (OpenMP/Pthreads)

Note : All the parallel programming models share a common computational model ; style - they are *imperative*

Implicit Parallel Programming Models

Q. Are parallelizing compilers effective in generating efficient code from real sequential programs?



Parallelizing Compilers

- ❖ Compiler performs **dependence analysis** on sequential program's source code.
- ❖ It uses a suite for program transformation techniques to convert the sequential code into native parallel code.
- ❖ A key in parallelizing a sequential code is **dependence analysis**, which defines **data dependence** and **control dependence**. (we have already seen it earlier)
- ❖ When dependencies do exist, **program transformation** (also known as **code re-structuring**) techniques are used to either eliminate them or otherwise make the code parallelizable, if at all possible.

Effectiveness of Parallelizing Compilers

- ❖ Most existing analyzing and restructuring techniques focus on loop level, i.e., how to exploit parallelism in Fortran **do** loops or C **for** loops.
- ❖ Are parallelizing compilers effective in generating efficient code from real sequential program?
- ❖ Performance studies indicate they **may not be effective**.

Optimizing Techniques for Eliminating Data Dependency

❖ Example

- Compare the speed-ups for automatic, manual, privatization, reduction, induction using complete parallelization and vectorization of 32 Processor parallel machine
 - Programs : ADM, ARC2D, BDNA, FLO52, OCEAN, TRACK, TRFD,
- ❖ The main reason for disappointing performance is the compiler's inability to exploit parallelism.

The performance could be further reduced on MPPs and COWs

Optimizing Techniques for Eliminating Data Dependency

```
do I = 1, N  
  P : A = ....  
  Q : X(I) = A + ....  
  .....  
end do
```

Privatization technique eliminates this dependency by making A into an array such that each iteration I has its own private copy

```
pardo I = 1, N  
  P : A (i) = ....  
  Q : X(i) = A (i)+ ....  
  .....  
end pardo
```

```
do I = 1, N  
  P : X(I) = ....  
  Q : Sum = Sum +X(I)  
  .....  
end do
```

The reduction technique eliminates this dependency by turning the loop-carried sequence of N sequential additions into an explicit reduction

```
pardo I = 1, N  
  P : X (i) = ....  
  Q :  
  Sum=sum_reduce(A(I))  
  .....  
end pardo
```

Bernstein's Theorem

- It is not decidable whether two operations in an imperative sequential program can be executed in parallel.

Dr. Allen, Frances E (Born: 4 August 1932, New York, US,
Died: 4 August 2020, New York, US)

- She is IBM fellow and is a recipient (first woman) of [Turing Award \(2006\)](#) for pioneering contributions to the theory and practice of optimizing compiler techniques that laid the foundation for modern optimizing compilers and automatic parallel execution.

- “I kind of stopped when C came out. That was a big blow ... We were making so much good progress on optimizations and transformations ... We have seriously regressed, since C developed. C has destroyed our ability to advance the state of the art in automatic optimization, automatic parallelization, automatic mapping ...”

https://www.noulakaz.net/2020/05/30/was-the-c-programming-language-one-of-the-best-and-at-the-same-time-one-of-the-worst-things-to-happen/?utm_source=chatgpt.com

Turing Award

- Any Indian to receive?
- Prof. Raj Reddy
- From United AP?
- IIITH, RGUKT
- Year?
- Work?

Compiler based Parallelization

Allow compiler to do automatic and directive – based parallelization

- ❖ `xautopar`, `xexplicitpar`, `xparallel`: tell the compiler to parallelize our program.
 - `xautopar`: tells the compiler to do only those parallelization that it can do automatically
 - `xexplicitpar`: tells the compiler to do only those parallelization that you have directed it to do with programs in the source
 - `parallel`: tells the compiler to parallelize both automatically and under `pragma` control
 - `xreduction`: tells the compiler that it may parallelize reduction loops. A reduction loop is a loop that produces output with smaller dimension than the input.

Source code directives

Compiler directives are put into the code as comments with a special form.

- ❖ For Fortran 77 and Fortran 90, the directives start with **C\$OMP**
- ❖ For C, the directives are supplied as **# pragma OMP**
- ❖ The three directives are DO ALL, DOSERIAL, and DOSERIAL*
 - DOALL – directs the compiler **to parallelize** the loop
 - DOSERIAL – directs the compiler **not to parallelize** the loop
 - DOSERIAL* – directs the compiler **not to parallelize** the loop **nest**.

Remark : DOSERIAL and DOSERIAL* are often used when the user knows in advance that the loops are too small to parallelize efficiently.

❖ Augmenting directives *Explicitly* with qualifiers

- We can use qualifiers to explicitly qualify every variable in a parallel loop
 - **PRIVATE** : The compiler creates private copies of these variables for each loop iteration.
 - **SHARED** : The compiler shares these variables among all loop iterations.
 - **READONLY** : A superset of SHARED that also means that the variable is **never written**.
 - **REDUCTION** : A special form of SHARED to indicate a reduction variable.

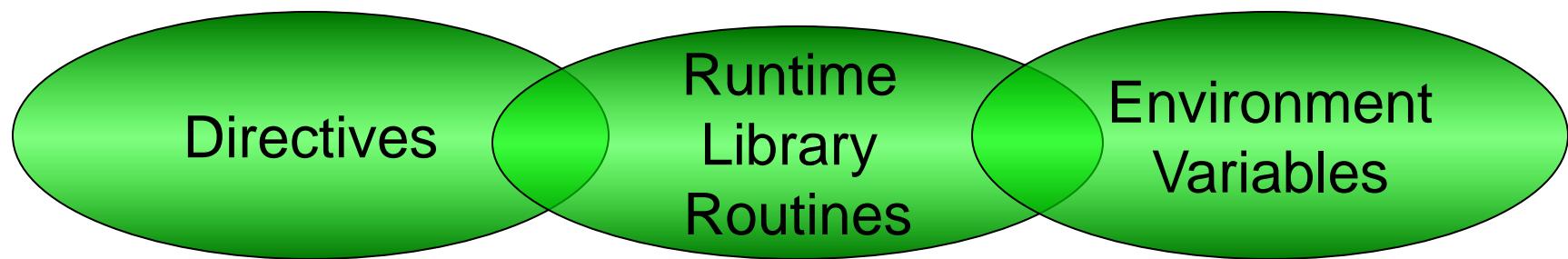
Shared Memory Programming With OpenMP

Open Multi-Processing

OpenMP: Introduction

OpenMP is an API for writing Multithreaded Applications

- Comprised of **three** primary API components: (will be explained in detail later)



- Portable : Makes it easy to create multi-threaded programs in C,C++ and Fortran.
- **Standardizes the SMP practice**

What does OpenMP stand for?

Open specifications for Multi Processing as a collaborative work of hardware vendors, software industry & academia

OpenMP Is

- **Not** meant for distributed memory parallel systems (by itself)
- **Not** necessarily implemented identically by all vendors
- **Not** guaranteed to make the most efficient use of shared memory

Then why OpenMP?

- ❖ Relatively easy to do parallelization for small parts of an application at a time
- ❖ Impact on code **quantity** (e.g., amount of additional code required) **and** code **quality** (e.g., readability of parallel code)
- ❖ Feasibility of scaling an application to a large number of processes
- ❖ Readability of the parallel code is high
- ❖ Availability of application development and debugging environment
- ❖ Standard and portable API

History

- ❖ First standard ANSI X3H5 in 1994
- ❖ OpenMP Standard SPECs started in 1997
- ❖ The **current version** is 6.0, released in 2024
- ❖ OpenMP Architecture review board
 - Compaq, HP, IBM, Sun Micro System, Intel Corp, Kuck & Associate Inc (KAI), SGI, US Dept. of Energy, etc.

Goals of OpenMP

Standardization:

- Provide a standard among a variety of shared memory architectures/platforms

Lean and Mean:

- Establish a simple and limited set of directives for programming shared memory machines
- Significant parallelism can be implemented by using just 3 or 4 directives

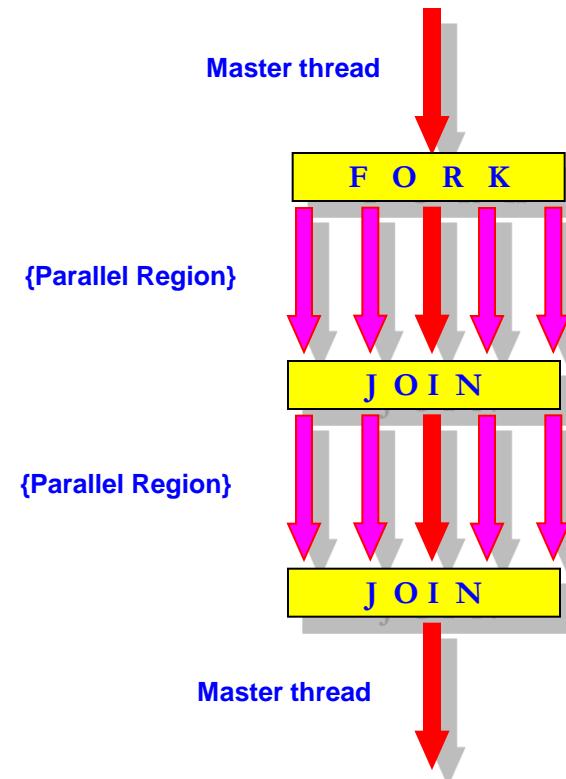
Ease of Use:

- Provide capability to **incrementally** parallelize a serial program, unlike message-passing libraries which typically require an **all or nothing** approach
- Provide the capability to implement both coarse-grain and fine-grain parallelism
- Multi-platform shared-memory multiprocessing **programming** supporting Fortran (77, 90, and 95), C, and C++
- No support for Java
- **JOMP, JaMP**—an OpenMP-like interface for Java

OpenMP: Programming Model

Fork – Join Model

- OpenMP uses fork and join model for parallel execution
- OpenMP programs begin with single process: **master thread**.
- **FORK** : Master thread creates a **team** of parallel threads
- **JOIN**: When the team threads complete the statements in parallel region, they synchronize and **terminate** leaving master thread.
- Parallelism is added incrementally conversion from a sequential to parallel is a little bit at a time



Threads based parallelization

- Open MP is based on the existence of **multiple threads** in the shared memory programming paradigm

Explicit parallelization

- It is an **explicit** (not automatic) programming model, and offers full control over parallelization to the programmer

Compiler directive based

- All of OpenMP parallelization is supported through the use of compiler directives

Nested parallelism support

- The API support placement of parallel construct inside other parallel construct

Dynamic threads

- The API provides dynamic altering of number of threads
(Depends on the implementation)

How do threads interact?

- OpenMP is shared memory model. Threads communicate by **sharing variables**

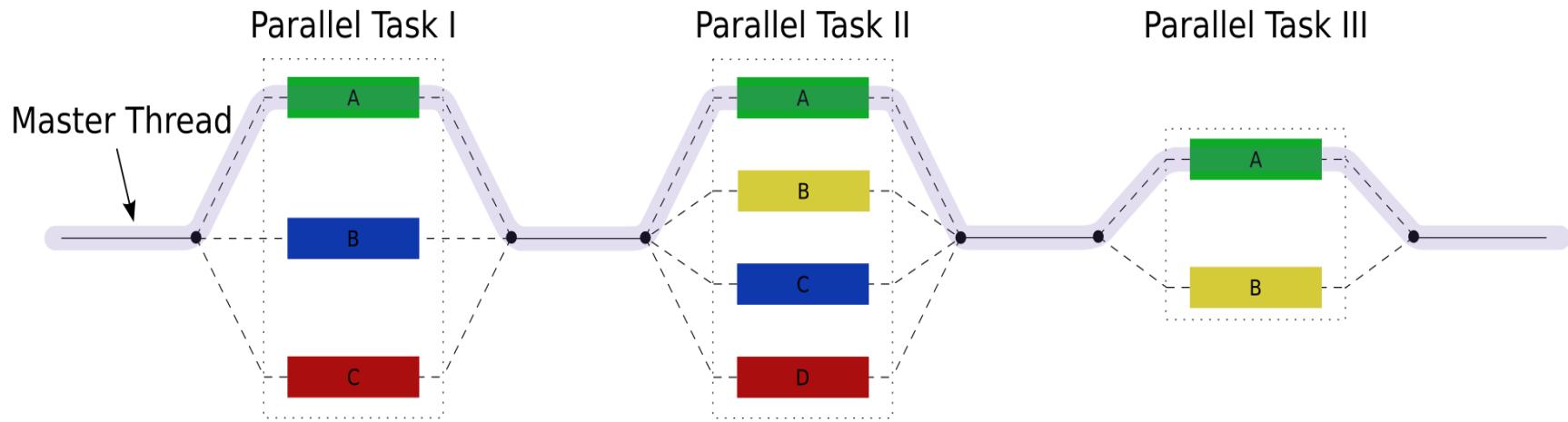
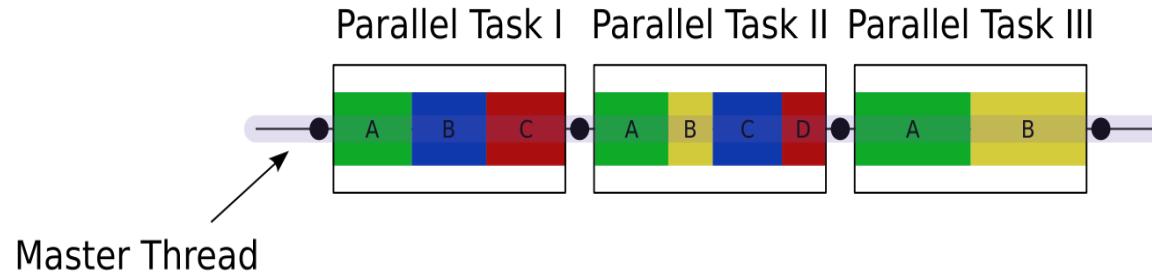
Compilers - Freeware

- **Intel C++/Fortran Compiler** for Linux. These compilers are free for academics.
- **INTONE project** A prototype version of Fortran and C compilers, as well as runtime support library for OpenMP
- **Nanos** NanosCompiler is a source-to-source parallelizing compiler for OpenMP applications written in Fortran 77
- **Odium** A free, Portable OpenMP Implementation for C. Developed at The Department of Information Technology, Lund University, Sweden
- **OdinMP2** an upgrade to the open-source OdinMP distribution that supports OpenMP 2.5 (except threadprivate). The upgrade converts the runtime to Posix Threads on both Apple/Intel OS/X and Windows XP. The distribution also contains a conformance test for OpenMP.
- **Omni** It is a collection of programs and libraries (in C and Java) that allows researchers to build code transformation systems (compilers). Omni OpenMP compiler is a part of this software that translates C and Fortran77 programs with OpenMP pragmas into C code suitable for compiling with a native compiler linked with the Omni OpenMP runtime library.

Compilers - Freeware

- **Ompi** OMPI is a light and portable source-to-source OpenMP compiler for C, conforming to version 2.0 of the specifications
- **OpenUH** is an open source compiler suite for OpenMP 2.5 in conjunction with C, C++, and Fortran 77/90 (and some Fortran 95) and the IA-64 Linux ABI and API standards.
- **SLIRP** is a vectorizing wrapper generator aimed at creating modules for the **S-Lang** interpreter.
A distinguishing feature of SLIRP is that it can generate parallelizable wrappers for OpenMP-aware compilers, enabling simplified use of OpenMP from a powerful scripting language well suited for scientific and engineering work
- **Sun Studio Compiler and Tools** for Linux and Solaris.
These compilers are for free.

OpenMP Structure



C / C++ - General Code Structure

```
#include <omp.h>
```

```
main()
```

```
{
```

```
int var1, var2, var3;
```

```
Serial code...
```

Beginning of parallel section. Fork a team of threads.

Specify variable scoping

```
#pragma omp parallel private(var1, var2) shared(var3)
```

```
{
```

Parallel section executed by all threads...

All threads join master thread and disband

```
}
```

```
Resume serial code...
```

```
}
```

Pragmas

- Pragma: a compiler directive in C or C++
- Stands for “**pragmatic information**”
- A way for the programmer to communicate with the compiler
- Syntax:
#pragma omp <rest of pragma>

Execution Context

- Every thread has its own **execution context**
- Execution context: address space containing all of the variables a thread may access
- Contents of execution context:
 - ◆ static variables
 - ◆ dynamically allocated data structures in the heap
 - ◆ variables on the run-time stack
 - ◆ additional run-time stack for functions invoked by the thread

OpenMP : C/C++ Directives Format

sentinel	directive-name	[clause ...]	newline
Required for all OpenMP C/C++ directives #pragma omp	A valid OpenMP directive. Must appear after the pragma and before any clauses.	Optional. Clauses can be in any order and repeated as necessary otherwise restricted.	Required. Proceeds the structured block which is enclosed by this directive.

Example

#pragma omp parallel shared (alpha), private(beta)

General Rules:

- Directives follow conventions of the C/C++ standards for compiler directives
- Case sensitive
- Only one directive-name may be specified per directive
- Each **directive applies to** at most one succeeding statement, which must be **a structured block**.
- Long directive lines can be “continued” on succeeding lines by escaping the newline character with a backslash (“\”) at the end of a directive line.

OpenMP : Constructs

Main categories of OpenMP's constructs:

➤ **Directives**

- ❖ Parallel Regions
- ❖ Work-sharing
- ❖ Data Environment
- ❖ Synchronization

➤ **Runtime library functions**

- ❖ Execution Environment Functions
- ❖ Lock functions
- ❖ Timing routines

➤ **Environment variables**

OpenMP : PARALLEL Region Construct

Purpose

A parallel region is a block of code that will be executed by multiple threads concurrently.

```
#pragma omp parallel [ clause .....] newline  
    if (scalar_logical) expression  
    private (list)  
    shared(list)  
    default (private | shared | none)  
    firstprivate (list)  
    reduction(operator : list)  
    structured_block
```

- When a thread reaches a PARALLEL directive, it creates a team of threads and **becomes the master** of the team. The master is also a member of that team and has **thread number 0** within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an **implied barrier** at the end of a parallel section. Only the master thread continues execution past this point.

OpenMP : Parallel Regions

Example

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_thread_num();
    xyz(ID,A);
}
```

Printf("all done\n")

Each thread
redundantly
executes
the code
within the
structured
block

*Runtime
function
returning the
thread ID*

Each thread executes the same code redundantly.



double A[1000];



omp_set_num_threads(4);



a single copy of A is shared between all threads

xyz(0,A) xyz(1,A) xyz(2,A) xyz(3,A)

printf("all done\n");

Threads wait here for all threads to finish before proceeding (i.e a barrier)

Let us write the first C/OpenMP program

OpenMP – 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;
}
```

OpenMP – Hello World

On hpc cluster machine

{this will work only on the head node and not spawned on any other node}

```
[wankarcs@master2 ~] $ gcc -fopenmp op1.c -o op1  
[wankarcs@master2 ~] $ ./op1
```

Processor Number Intel Core i5-4310M (Dual core)

of Cores 2 (Hyper threading Technology)

of Threads 4

OpenMP : Work-sharing Construct

It distributes the execution of the associated statement among the members of the team that encounter it

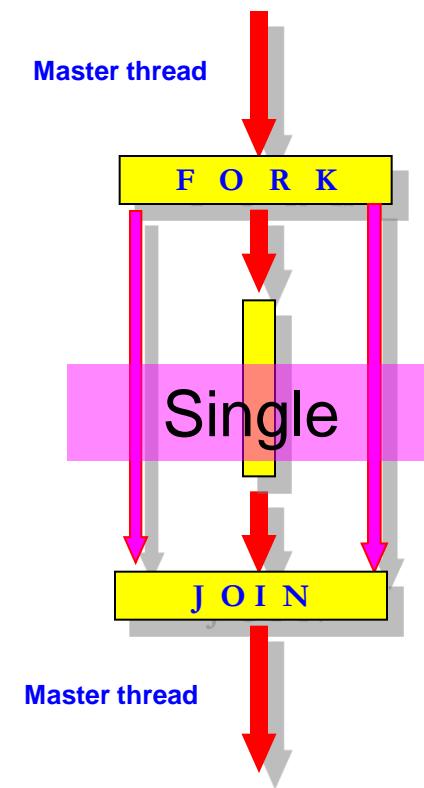
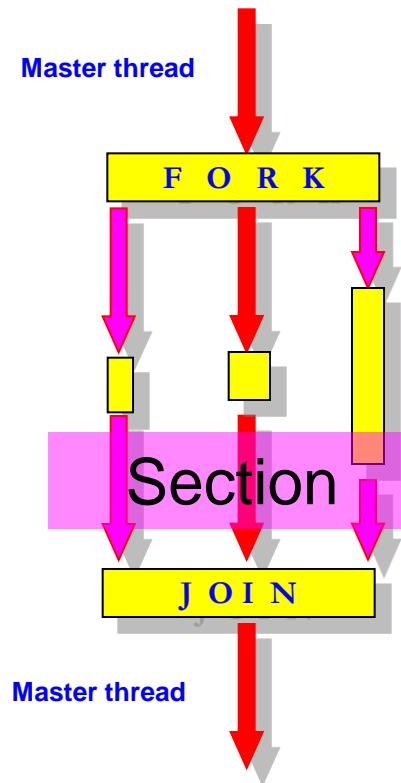
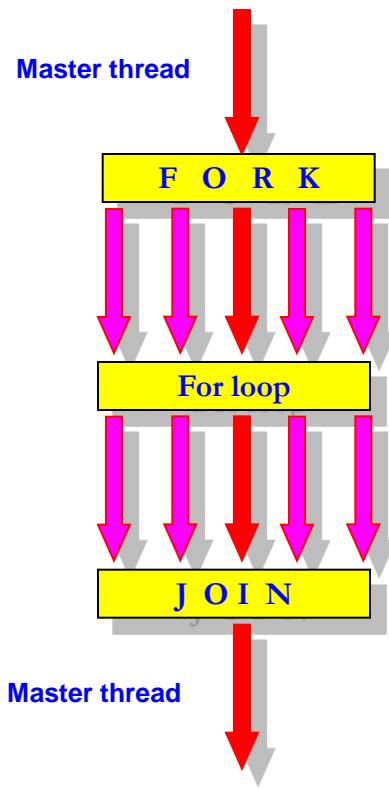
- Work sharing construct do not launch new threads
- There is no barrier upon entry to work-sharing construct.
- There is an implied barrier at the end of a work sharing construct

Restrictions

- Must be enclosed in the parallel region for parallel execution
- Must be encountered by all the members of the team or none of them

OpenMP defines the following work-sharing constructs.

- **for** directive
- **sections** directive
- **single** directive



for directive (Represents a type of “data parallelism”)

For directive identifies the iterative work-sharing construct.

#pragma omp for [clause[,]clause]...] new-line

for-loop Clause is one of the following:

scheduled (**type [,chunk]**)

private(*variable list*)

firstprivate (*variable list*)

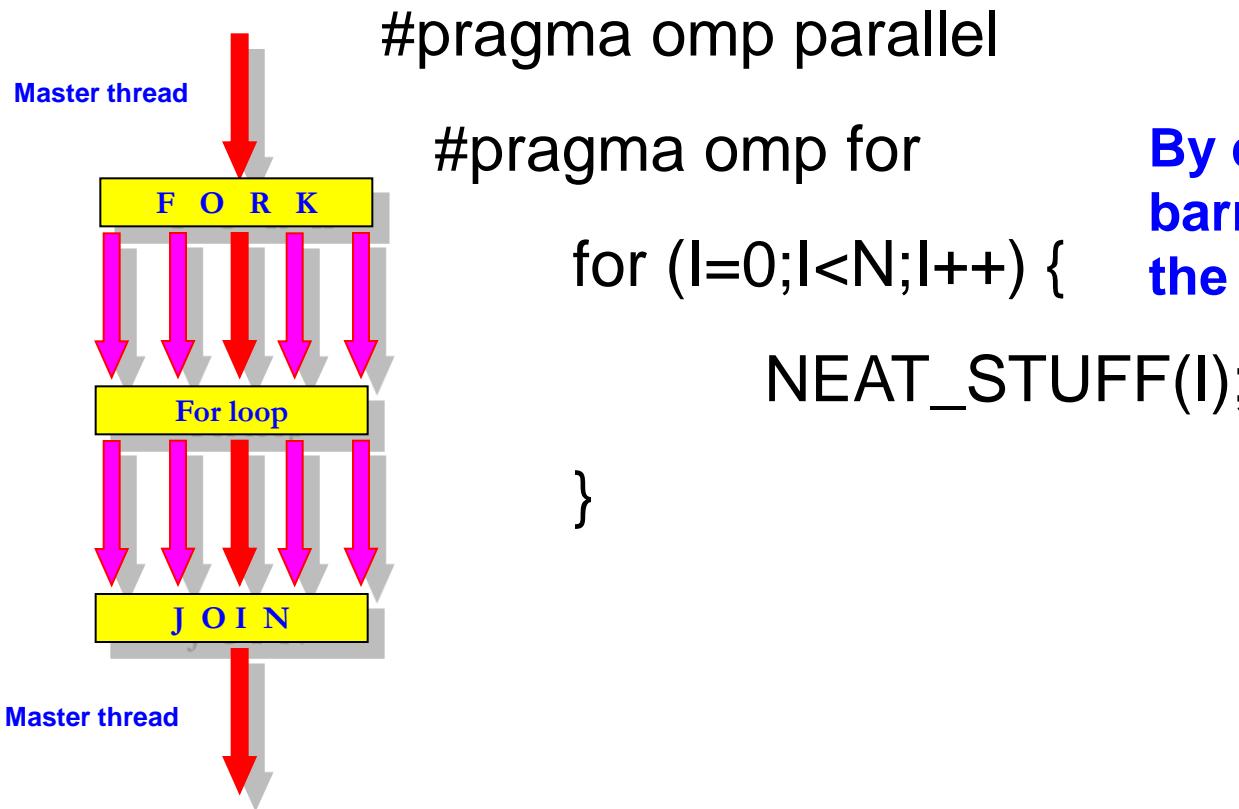
lastprivate (*variable list*)

reduction (*variable list*)

ordered, nowait

for directive

The “for” Work-Sharing construct splits up loop iterations among the threads in a team



By default, there is a barrier at the end of the “omp for”.

Important

The **for** directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, **otherwise it executes in serial on a single processor.**

```

#include <omp.h>

#define CHUNKSIZE 100

#define N 1000

main()
{
    /* program name openvadd.c on jupiter */

    int i, chunk; float a[N], b[N], c[N];

    /* Some initializations */

    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;

    chunk = CHUNKSIZE;

#pragma omp parallel for shared(a,b,c,chunk) private(i)

#pragma omp schedule(dynamic,chunk) nowait

    for (i=0; i < N; i++)
        c[i] = a[i] + b[i]; /* end of || section */

}

```

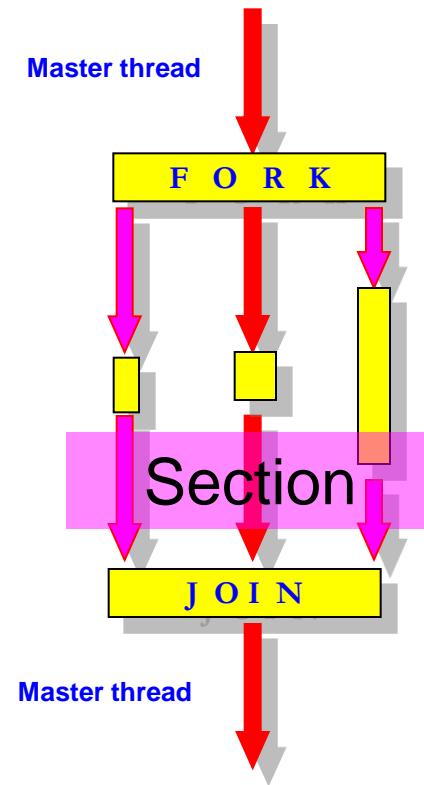
sections directive

Can be used to implement a type of "functional parallelism".

sections directive gives different structured blocks to each thread.

```
#pragma omp parallel  
#pragma omp sections {  
    #pragma omp section  
        x_calculation();  
    #pragma omp section  
        y_calculation();  
    .....  
}
```

- ❖ Independent **SECTION** directives are nested within a **SECTIONS** directive.
- ❖ Each SECTION is executed **once** by a thread in the team. Different sections will be executed by different threads.



Simple vector-add program

- ❖ The first $n/2$ iterations of the `for` loop will be distributed to the first thread, and the rest will be distributed to the second thread
- ❖ When each thread finishes its block of iterations, it proceeds with whatever code comes next (NOWAIT)

```

#include <omp.h>

#define N 1000

main()

{ int i;      float a[N], b[N], c[N];

for (i=0; i < N; i++)
a[i] = b[i] = i * 1.0;

#pragma omp parallel shared(a,b,c) private(i)  {

#pragma omp sections nowait      {

#pragma omp section

        for (i=0; i < N/2; i++) c[i] = a[i] + b[i];

#pragma omp section

        for (i=N/2; i < N; i++) c[i] = a[i] + b[i];
    }

}

}

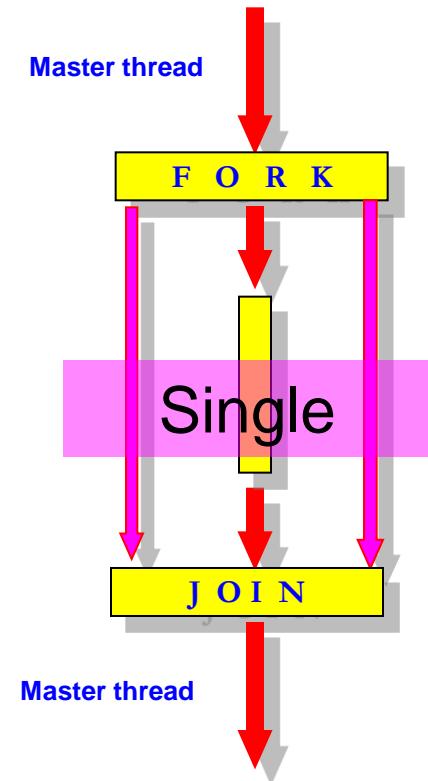
```

single directive

This identifies that the associated structured block is to be executed by only one thread in the team (It can be any thread including master thread).

```
#pragma omp single [clause[,] clause] ...  
new-line
```

structured-block



Restrictions

- ❖ A work-sharing construct must be enclosed dynamically **within a parallel region** in order for the directive to execute in parallel
- ❖ Work-sharing constructs must be encountered by **all members of a team or none at all**
- ❖ Successive work-sharing constructs must be encountered in the same order by all members of a team

OpenMP : Data Environment

OpenMP : Data Environment

Default storage attributes

- ❖ Shared Memory programming model:
 - Most variables are shared by default
- ❖ Global variables & SHARED are shared among threads
 - FORTRAN : COMMON blocks, SAVE variables, MODULE variables.
 - C: File scope variables, static
- ❖ But not everything is shared...
 - Stack variables in sub-programs called from parallel regions are PRIVATE
 - Automatic variables within a statement block are PRIVATE.

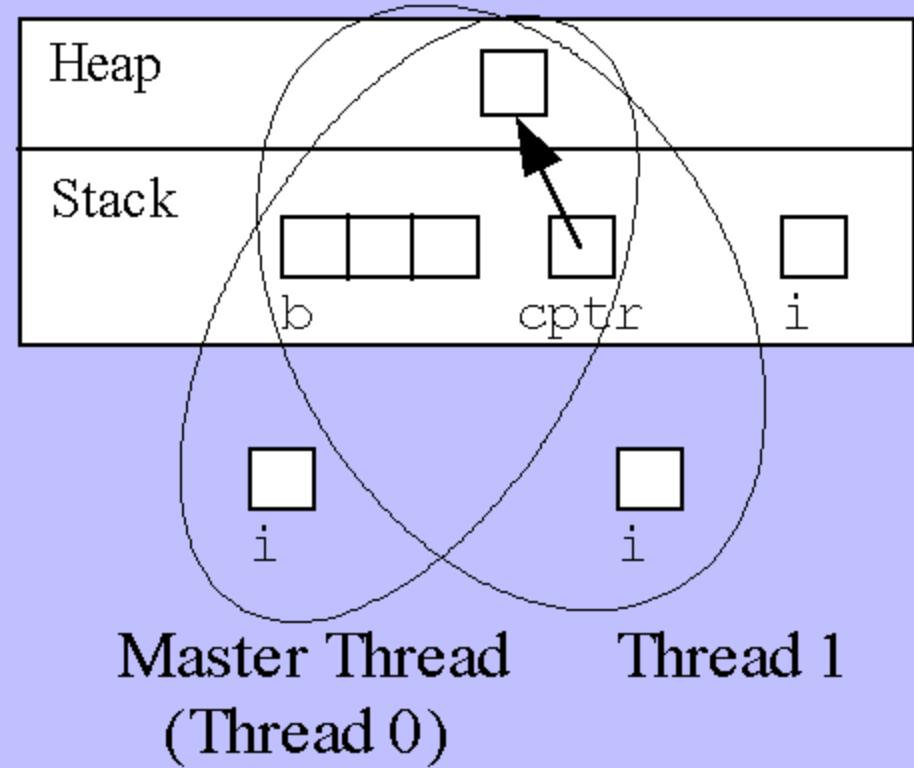
Shared and Private Variables

- **Shared variable** has same address in the execution context of every thread
- **Private variable** has different address in the execution context of every thread
- *A thread cannot access the private variables of another thread*

Shared and Private Variables

```
int main (int argc, char *argv[])
{
    int b[3];
    char *cptr;
    int i;

    cptr = malloc(1);
    #pragma omp parallel for
    for (i = 0; i < 3; i++)
        b[i] = i;
```



Changing the storage attributes

One can selectively change storage attributes constructs using the following clauses

SHARED	declares variables to be shared among all threads in the team
PRIVATE	declares variables to be private to each thread.
FIRSTPRIVATE	performs initialization of private variables
LASTPRIVATE	performs finalization of private variables
REDUCTION	performs a reduction on the variables subject to given operator.

The default status can be modified with:

DEFAULT (PRIVATE | SHARED | NONE)

Combined Parallel Work-Sharing Constructs

parallel for Directive

- ❖ The parallel for (C/C++) directives specify a parallel region that **contains a single for directive**. The single for directive must follow immediately as the very next statement
- ❖ Iterations of the for loop will be distributed in equal sized blocks to each thread in the team (SCHEDULE STATIC)

Simple vector-add program

- ❖ Arrays A, B, C, and variable N will be shared by all threads.
- ❖ Variable I will be private to each thread; each thread will have its own unique copy.
- ❖ The iterations of the loop will be distributed statically in CHUNK sized pieces.
- ❖ Threads will not synchronize upon completing their individual pieces of work (NOWAIT).

```

#include <omp.h>

#define N 1000

#define CHUNKSIZE 100

main ()
{
    int i, chunk;
    float a[N], b[N], c[N];
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
    #pragma omp parallel for shared(a,b,c,chunk) private(i) \
schedule(static,chunk)
        for (i=0; i < n; i++)
            c[i] = a[i] + b[i];
}

```

Function `omp_get_num_procs`

- Returns number of physical processors (in case Hyper threading (intel), it is logical processors) available for use by the parallel program

```
int omp_get_num_procs(void)
```

Function `omp_set_num_threads`

- Uses the parameter value to set the number of threads to be active in parallel sections of code
- May be called at multiple points in a program

```
void omp_set_num_threads (int t)
```

Private Clause

- Clause: an optional, additional component to a pragma
- Private clause: directs compiler to make one or more variables private

```
private ( <variable list> )
```

Example Use of private Clause

```
#pragma omp parallel for private(j)
for (i = 0; i < BLOCK_SIZE(id,p,n) ; i++)
    for (j = 0; j < n; j++)
        a[i][j] = MIN(a[i][j],a[i][k]+tmp) ;
```

firstprivate Clause

- Used to create private variables having initial values identical to the variable controlled by the master thread as the loop is entered
- Variables are initialized once per thread, not once per loop iteration
- If a thread modifies a variable's value in an iteration, subsequent iterations will get the modified value

lastprivate Clause

- Sequentially last iteration: iteration that occurs last when the loop is executed sequentially
- **lastprivate** clause: used to copy back to the master thread's copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration

Reductions

- Reductions are so common that OpenMP provides support for them
- May add reduction clause to `parallel for` pragma
- Specify reduction operation and reduction variable
- **OpenMP takes care of storing partial results in private variables and combining partial results after the loop**
- Local copies are reduced into a single global copy at the end of the construct

reduction Clause

- The reduction clause has this syntax:
reduction (<op> :<variable>)
- Operators

+	Sum
*	Product
&	Bitwise and
 	Bitwise or
^	Bitwise exclusive or
&&	Logical and
 	Logical or

π -finding Code with Reduction Clause

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for \
    private(x) reduction(+:area)
for (i = 0; i < n; i++) {
    x = (i + 0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Performance Improvement #1

- Too many fork/joins can lower performance
- Inverting loops may help performance if
 - Parallelism is in inner loop
 - After inversion, the outer loop can be made parallel

Performance Improvement #2

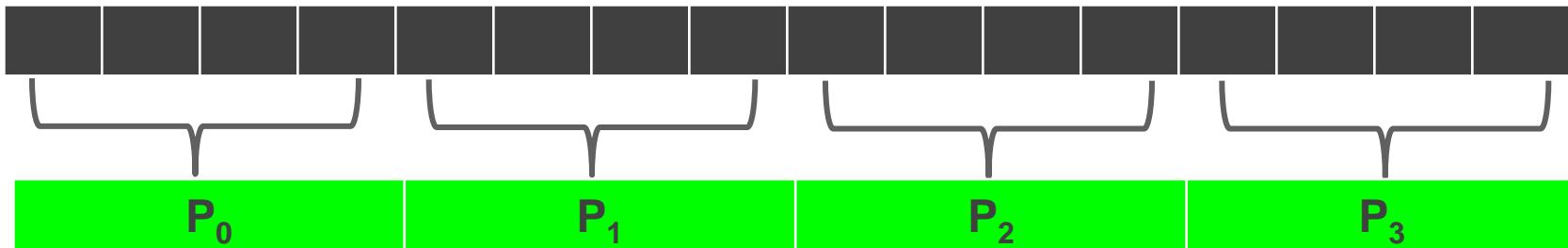
- If loop has too few iterations, fork/join overhead is greater than time savings from parallel execution
- The **if** clause instructs compiler to insert code that determines at run-time whether loop should be executed in parallel; e.g.,

```
#pragma omp parallel for if(n > 5000)
```

Performance Improvement #3

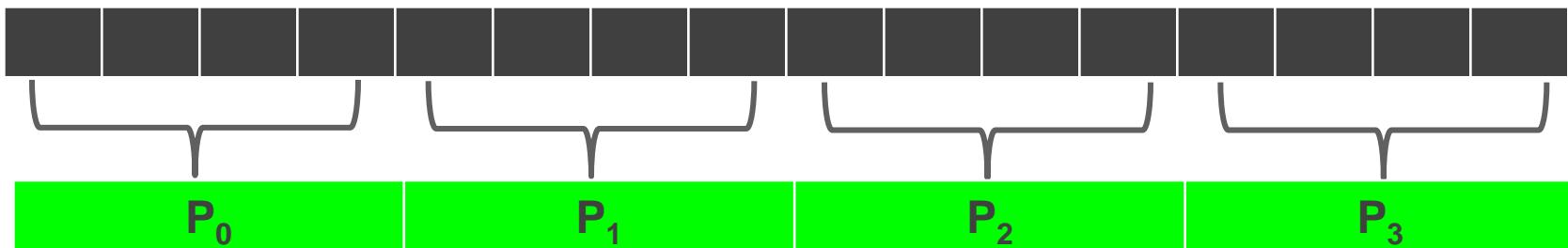
- We can use **schedule** clause to specify how iterations of a loop should be allocated to threads
- **Static schedule:** all iterations allocated to threads before any iterations executed
- **Dynamic schedule:** only some iterations allocated to threads at beginning of loop's execution. Remaining iterations allocated to threads that complete their assigned iterations.

pi calculation: Mix mode MPI/pthread (revisit)

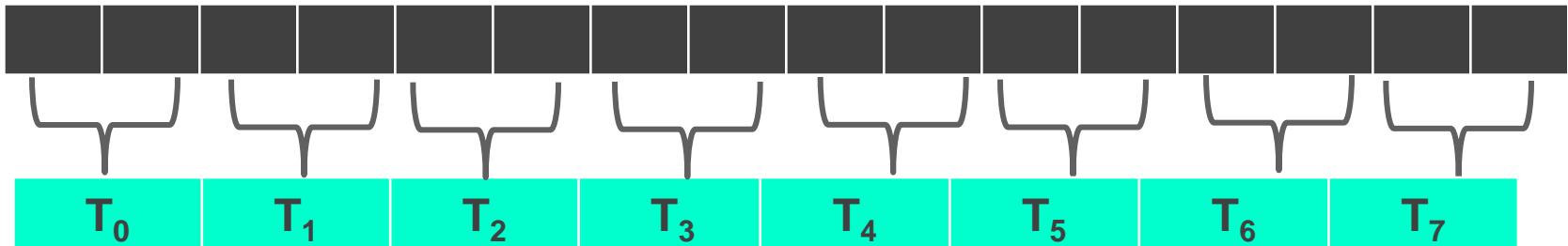


- Course grain implementation using MPI

pi calculation: Mix mode MPI/pthread (revisit)

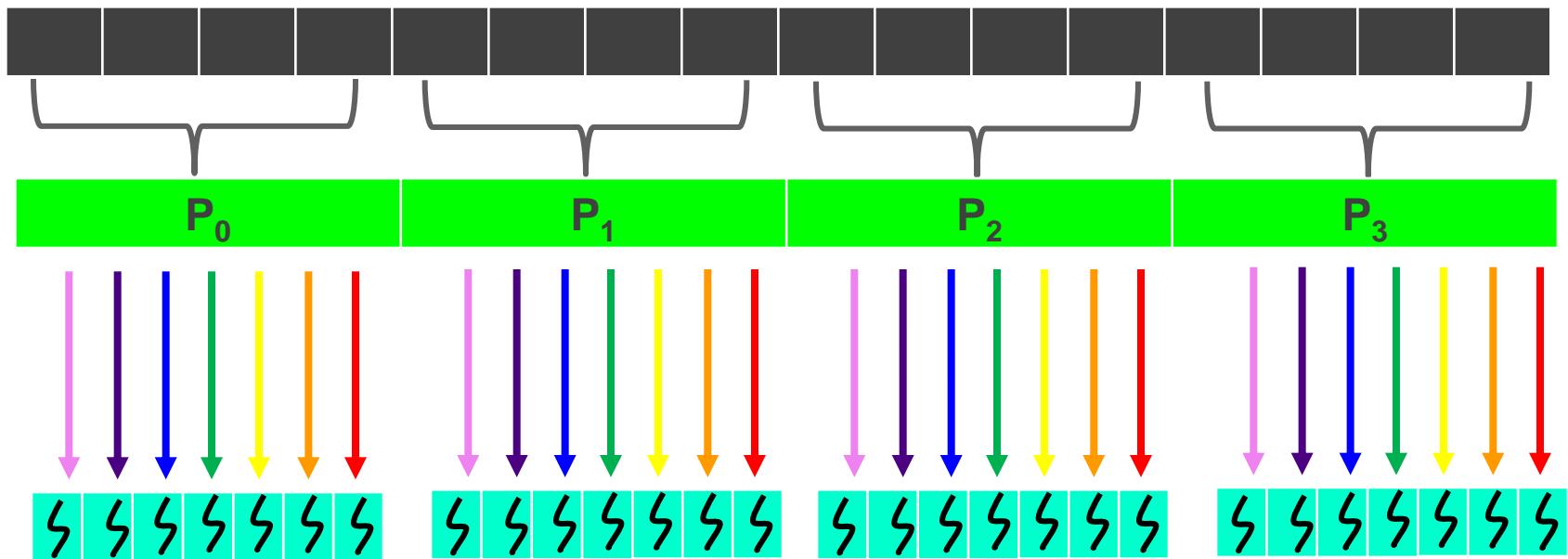


- Course grain implementation using MPI



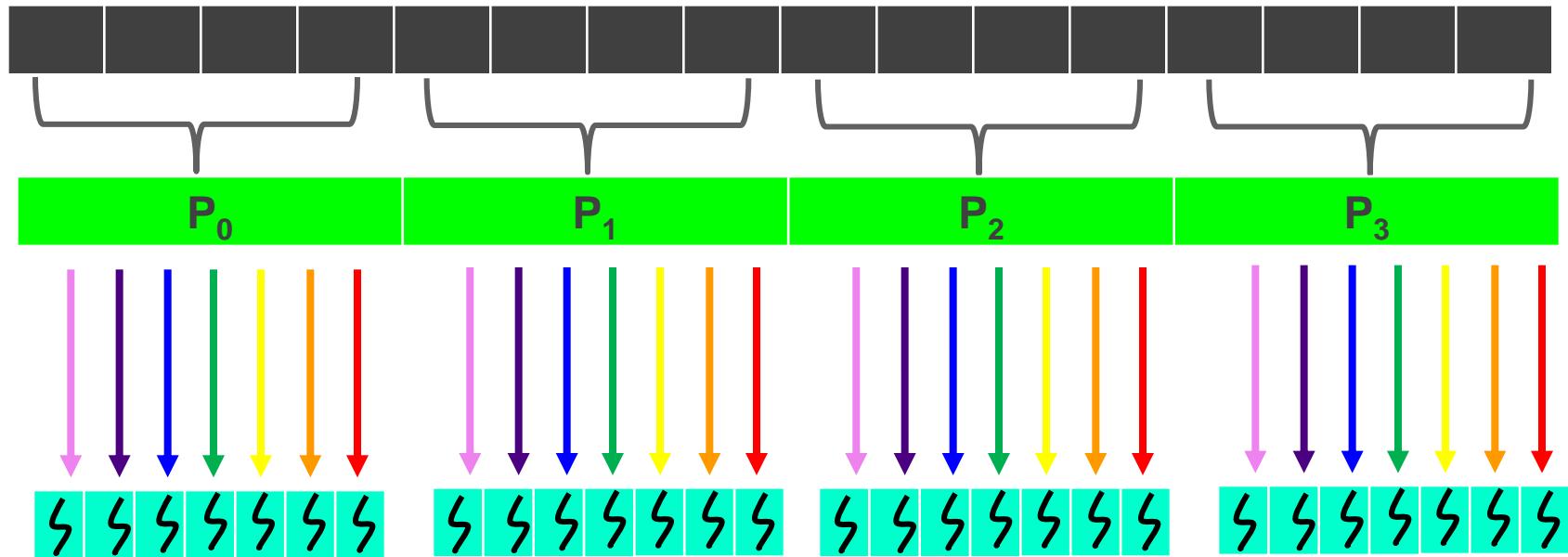
- Fine grain implementation using pthread

pi calculation: Mix mode MPI/pthread (revisit)



- Implementation using MPI + pthread

pi calculation: Mix mode MPI/OpenMP



- Implementation using MPI + OpenMP
- Loop index is divided among the threads which are scheduled by OS based on Compiler Directives

pi calculation: Mix mode MPI/OpenMP-01

- How to divide the computation so that there is a load balancing?

pi calculation: Mix mode MPI/OpenMP-01

- Input (*interval*) is used to find step size $h = 1/\text{interval}$
- *num_proc* denotes number of processes (processors) used to execute program
- First element of process *i* will be $\left\lfloor \frac{i * \text{interval}}{\text{num_proc}} \right\rfloor$
- Last element of process *i* will be $\left\lfloor \frac{(i+1) * \text{interval}}{\text{num_proc}} \right\rfloor - 1$
- This scheme try to give ~equal number of chunks to different processes
- Inside the local portion at a process, OpenMP's **work-sharing** constructs take an amount of work and distribute it over the available threads in a parallel region.

pi calculation: Mix mode MPI/OpenMP-02

- Input (*interval*) = 22, *num_proc* = 05



Process ID	First Element is $\left\lfloor \frac{i * interval}{num_proc} \right\rfloor$	Last Element is $\left\lfloor \frac{(i+1) * interval}{num_proc} \right\rfloor - 1$	Total Elements
0	0	3	4
1	4	7	4
2	8	12	5
3	13	16	4
4	17	21	5

pi calculation: Mix mode MPI/OpenMP-03

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
//mpicc -o exe exe.c -lm for compiling
int main(int argc, char *argv[]) {
    int interval, i, num_proc, rank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&num_proc);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

pi calculation: Mix mode MPI/OpenMP-04

```
if (rank == 0)
{
    printf("Please enter the intervals: \n");
    scanf("%d",&interval);
}
MPI_Bcast(&interval, 1, MPI_INT, 0, MPI_COMM_WORLD);
int x = floor((rank*interval)/num_proc);
int y = floor(((rank+1)*interval)/num_proc)-1;
for (i=x; i<=y; i++)
    printf("my index on process %d is %d\n", rank, i);
MPI_Finalize();
return 0;
}
```

pi calculation: Mix mode MPI/OpenMP-05

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
//mpicc -o omp-mpi-pi omp-mpi-pi.c -lm for compiling
int main(int argc, char *argv[]) {

    int interval, i, num_proc, rank;
    double startwtime, endwtime; double pi, area = 0.0;
    double PI = 3.14159265;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&num_proc);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

pi calculation: Mix mode MPI/OpenMP-06

```
if (rank == 0)
{
    printf("Please enter the intervals: \n");
    scanf("%d",&interval);
}

startwtime = MPI_Wtime();
MPI_Bcast(&interval, 1, MPI_INT, 0, MPI_COMM_WORLD);

int x = floor((rank*interval)/num_proc);
int y = floor(((rank+1)*interval)/num_proc)-1;
double mypi,a;
```

pi calculation: Mix mode MPI/OpenMP-07

```
#pragma omp parallel for private(a) reduction(+:area)
schedule(static)

for (i=x; i<=y; i++){
    a = (i+0.5)/interval;
    area += 4.0/(1.0+a*a);
}

mypi=area/interval;

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
endwtime = MPI_Wtime();
```

pi calculation: Mix mode MPI/OpenMP-08

```
if(rank == 0){  
    printf("pi is approximately %.40f, Error is %.16f\n",pi,  
fabs(pi- PI));  
    endwtime = MPI_Wtime();  
    printf("wall clock time = %f\n", endwtime-startwtime);  
}  
MPI_Finalize();  
return 0;  
}
```

mpirun command

- Please remember that `mpirun` automatically binds processes as of the start of the v1.8 series. Three binding patterns are used in the absence of any further directives:
- Bind to core:
 - when the number of processes is ≤ 2
- Bind to socket:
 - when the number of processes is > 2
- Bind to none:
 - when oversubscribed

Functions for SPMD-style Programming

- The parallel pragma allows us to write SPMD-style programs
- In these programs we often need to know number of threads and thread ID number
- OpenMP provides functions to retrieve this information

Function `omp_get_thread_num`

- This function returns the thread identification number
- If there are t threads, the ID numbers range from 0 to $t-1$
- The master thread has ID number 0

```
int omp_get_thread_num (void)
```

Function omp_get_num_threads

- Function `omp_get_num_threads` returns the number of **active** threads
- If call this function from sequential portion of program, it will return 1

```
int omp_get_num_threads (void)
```

nowait Clause

- Compiler puts a barrier synchronization at end of every **parallel for** statement
- In our example, this is necessary: if a thread leaves loop and changes `low` or `high`, it may affect behavior of another thread
- If we make these private variables, then it would be okay to let threads move ahead, which could reduce execution time

Use of nowait Clause

```
#pragma omp parallel private(i,j,low,high)
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
        #pragma omp single
        printf ("Exiting (%d)\n", i);
        break;
    }
#pragma omp for nowait
for (j = low; j < high; j++)
    c[j] = (c[j] - a[i])/b[i];
}
```

OpenMP: Library Routines

Lock routines

- `omp_init_lock()`, `omp_set_lock()`,
- `omp_destroy_lock()`,
- `omp_unset_lock()`,
- `omp_test_lock()`

Runtime environment routines:

Modify/Check the number of threads

- `omp_set_num_threads()`,
- `omp_get_num_threads()`,
- `omp_get_thread_num()`,
- `omp_get_max_threads()`

OpenMP: Library Routines

Are we in a parallel region?

`omp_in_parallel()`

How many processors in the system?

`omp_num_procs()`

Turn on/off nesting and dynamic mode

`omp_set_nested()`,

`omp_get_nested()`,

`omp_set_dynamic()`,

`omp_get_dynamic()`

Summary

<i>Characteristic</i>	<i>OpenMP</i>	<i>MPI</i>
Suitable for multiprocessors	Yes	Yes
Suitable for multiccomputers	No	Yes
Supports incremental parallelization	Yes	No
Minimal extra code	Yes	No
Explicit control of memory hierarchy	No	Yes