

# Parallel Processing

---

Thoai Nam

Faculty of Computer Science and Engineering

HCMC University of Technology

# What we are doing?



# Internet of Things (IoT)

(Timothy Chou)

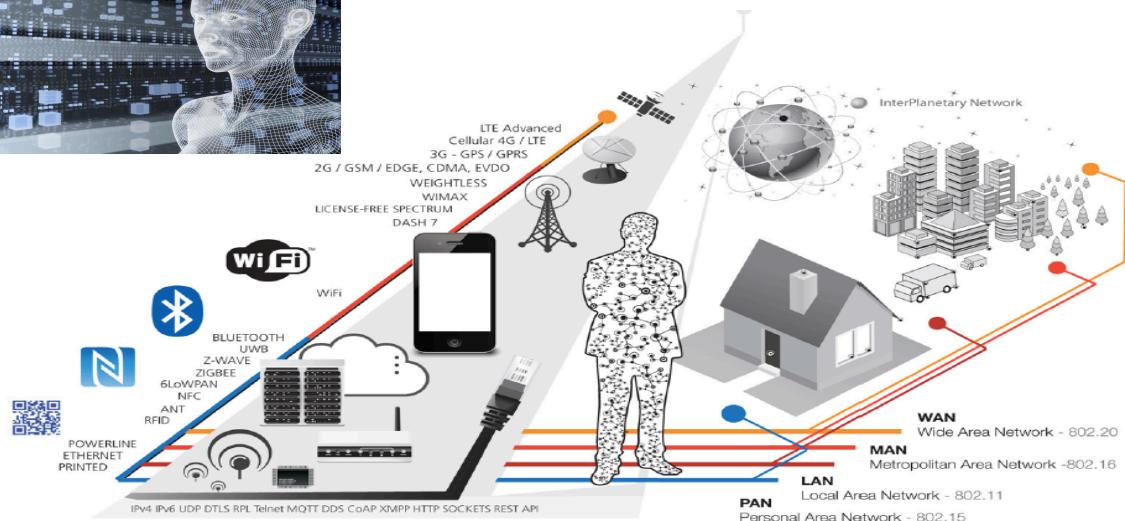
Do



Learn



Collect



Connect



Things

- (1) Smart city problems
- (2) Industry 4.0 problems

Real problems

Traffic jam  
Urban flooding  
Pollution: waster & air  
Security

ITS

GV

CE

IoT

D-STAR

HCMUT-UTS  
JRC

HCMUT-  
VNPT

Applications

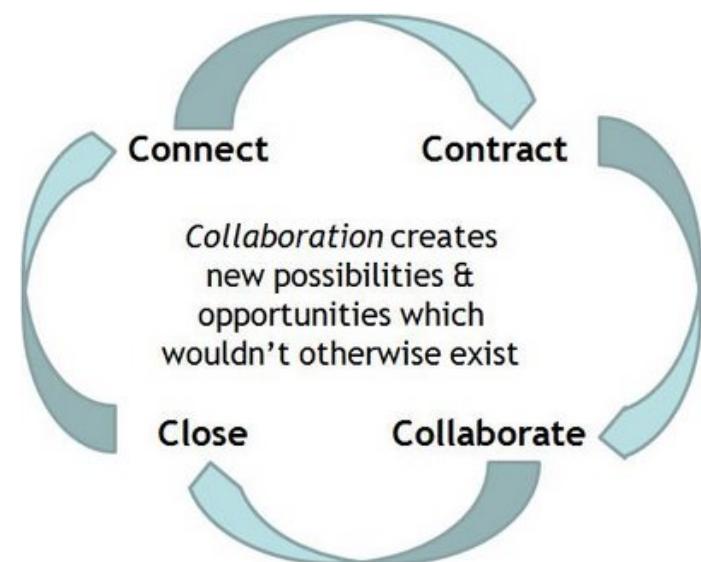
HPC Lab

Computing  
Infrastructure



# HPC Lab

- **Partners**
  - **VNU-HCM (HCMUT)**
  - **Intel**
  - Ho Chi Minh city
- **Plan: 2012-2022**
  - HPC Lab: set up in 2013
  - Strengthen HPC in Vietnam
  - Solving big problems
  - Leading in technology
  - **Partners: HPE (2015) & Nvidia (2017)**
- **Applications: 2013-2022**
  - **Traffic analysis**
  - **Urban flooding**
  - **Big data analytics**





# SuperNode I & II



SuperNode I in 1998-2000



The screenshot displays two Firefox windows. The left window shows a file browser titled 'Service View vaty' with a list of files in the current directory (~/). The right window shows a terminal session titled 'Browsing directory of hungnq - Mozilla Firefox' with the command 'ls' executed, displaying the same list of files.

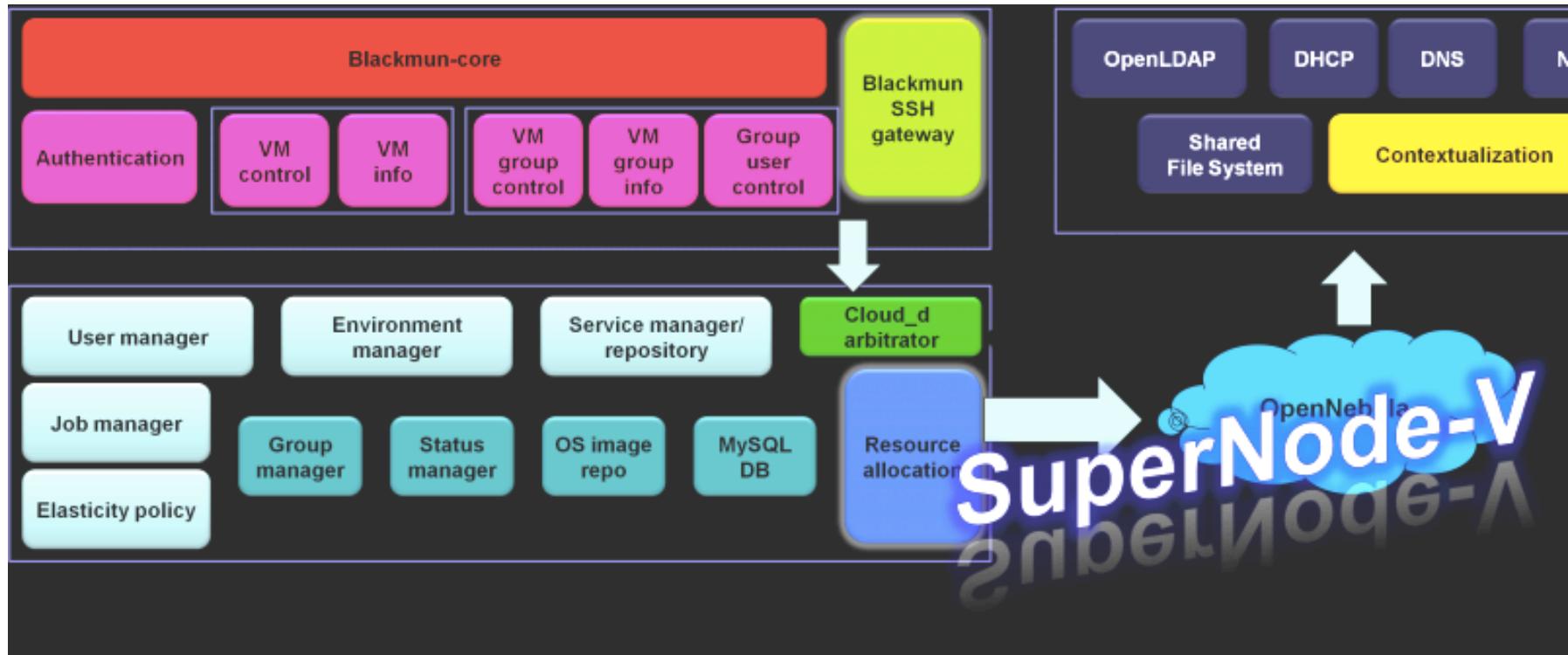
filename	Size	Permission	OwnerID	GroupID	Function
.bash_logout	24	-r--r--r--	539	1000	
.bash_profile	191	-r--r--r--	539	1000	
.bashrc	124	-r--r--r--	539	1000	
.gSRC	120	-r--r--r--	539	1000	
.kde	4096	d rwx r-x r-x	539	1000	
prime.c	7455	-r--r--r--	539	1000	
prime-seq.c	2295	-r--r--r--	539	1000	
prime.o	5156	-r--r--r--	539	539	



SuperNode II in 2003-2005



# SuperNode V



**SuperNode-V project: 2010-2012**



# EDA-Grid & VN-Grid

SuperNode II



Applications  
Chip design  
Data mining  
Airfoil optimization

Security

Monitoring

User Management

**Campus/VN-Grid (GT)**

Resource Management

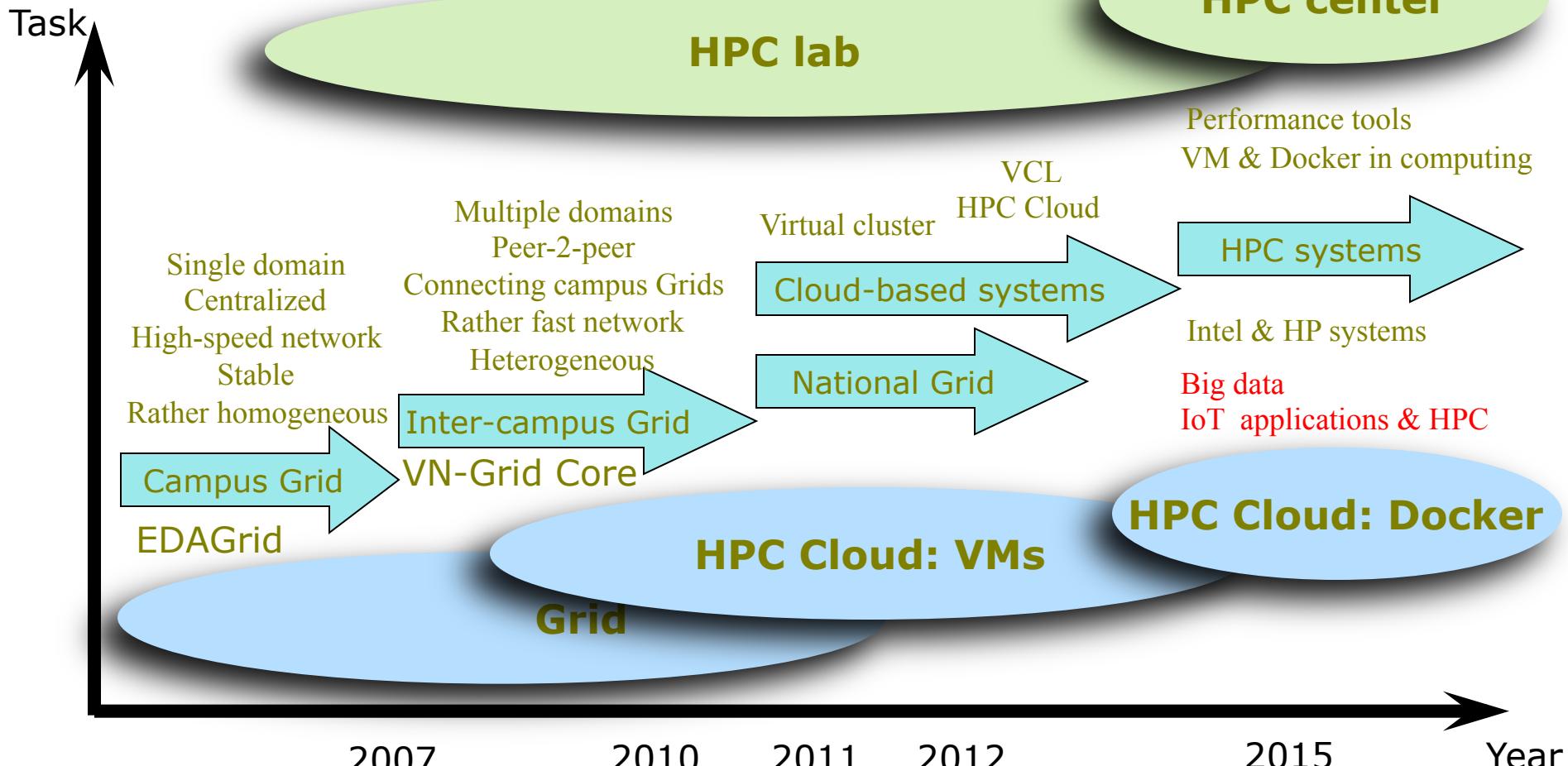
Information Service

Data Service

POP-C++



# HPC plan at HCMUT



# Parallel Processing

---

Thoai Nam

Faculty of Computer Science and Engineering

HCMC University of Technology

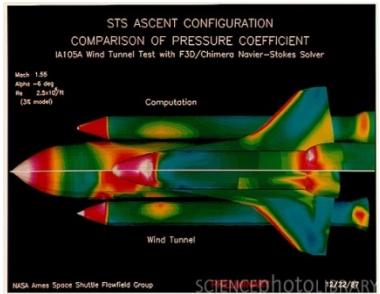


# Chapter 1: Introduction

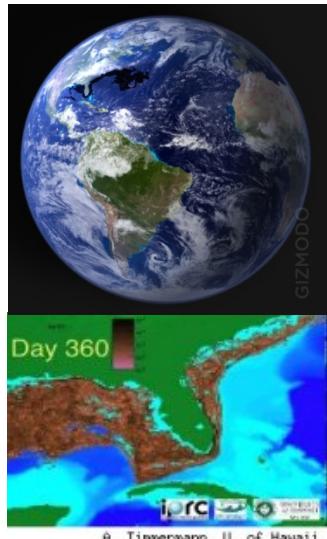
---

- HPC and applications
- New trends
- Introduction
  - What is parallel processing?
  - Why do we use parallel processing?
- Parallelism

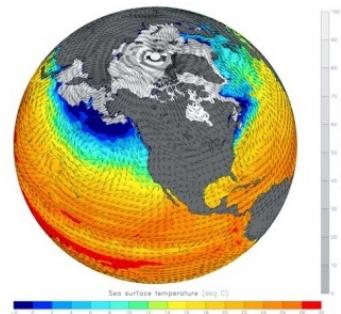
# Applications (1)



## Fluid dynamics



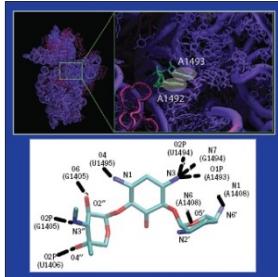
## Simulation of oil spill In in BP oil ship problem



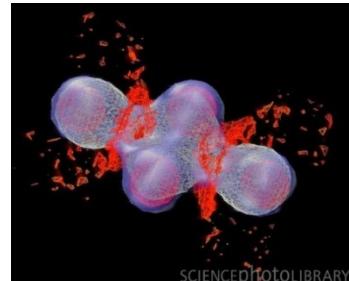
## Weather forecast (PCM)



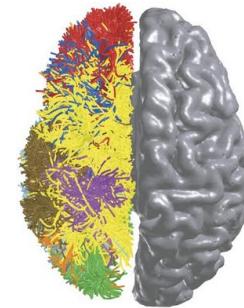
## Astronomy



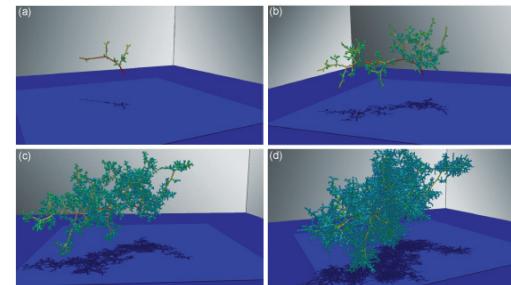
Medicine



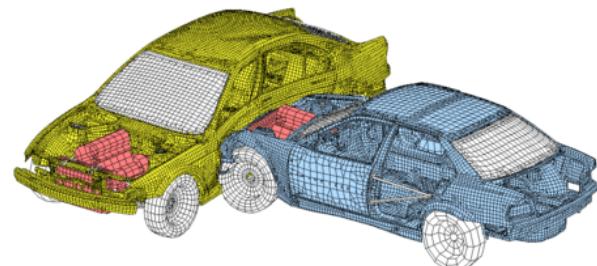
## Simulation i.e. Lithium atom



## Brain simulation



## Simulation of Uranium-235 created from Plutonium-239 decay



## Simulation of car accident



# Applications (2)

---

## □ Critical HPC issues

- Global warming
- Alternative energy
- Financial disaster modeling
- Healthcare

## □ New trends

- Big Data
- Internet of Things (IoT)
- 3D movies and large scale games are fun
- Homeland security
- Smart cities

? TBs of  
data every day



12+ TBs  
of tweet data  
every day



25+ TBs of  
log data  
every day

30 billion RFID  
tags today  
(1.3B in 2005)



76 million  
smart meters in 2009...  
200M by 2014

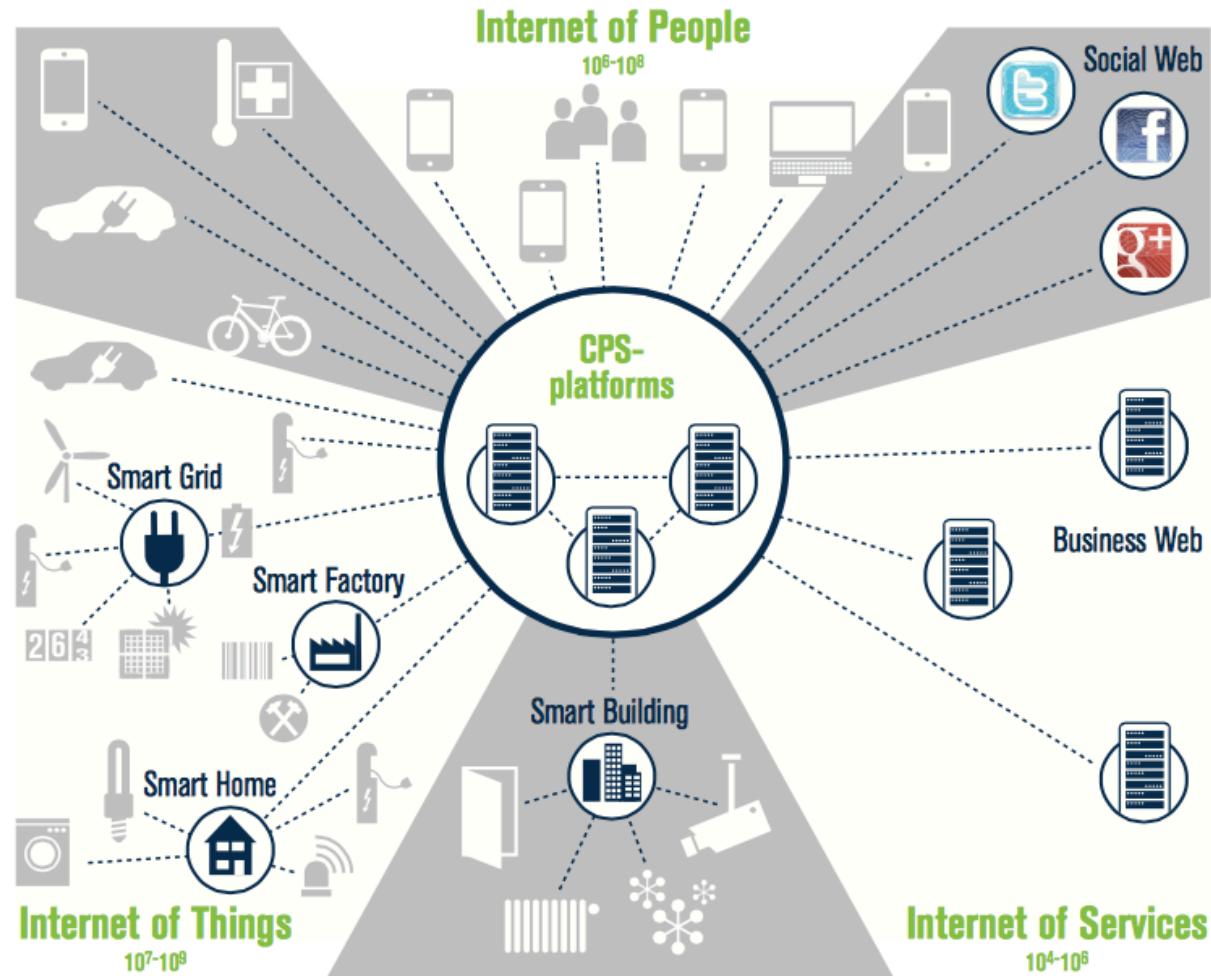
100s of  
millions of  
GPS  
enabled  
devices sold  
annually

2+ billion  
people on the Web by end  
2011

4.6  
billion  
camera  
phones  
world wide

# IoT and Services

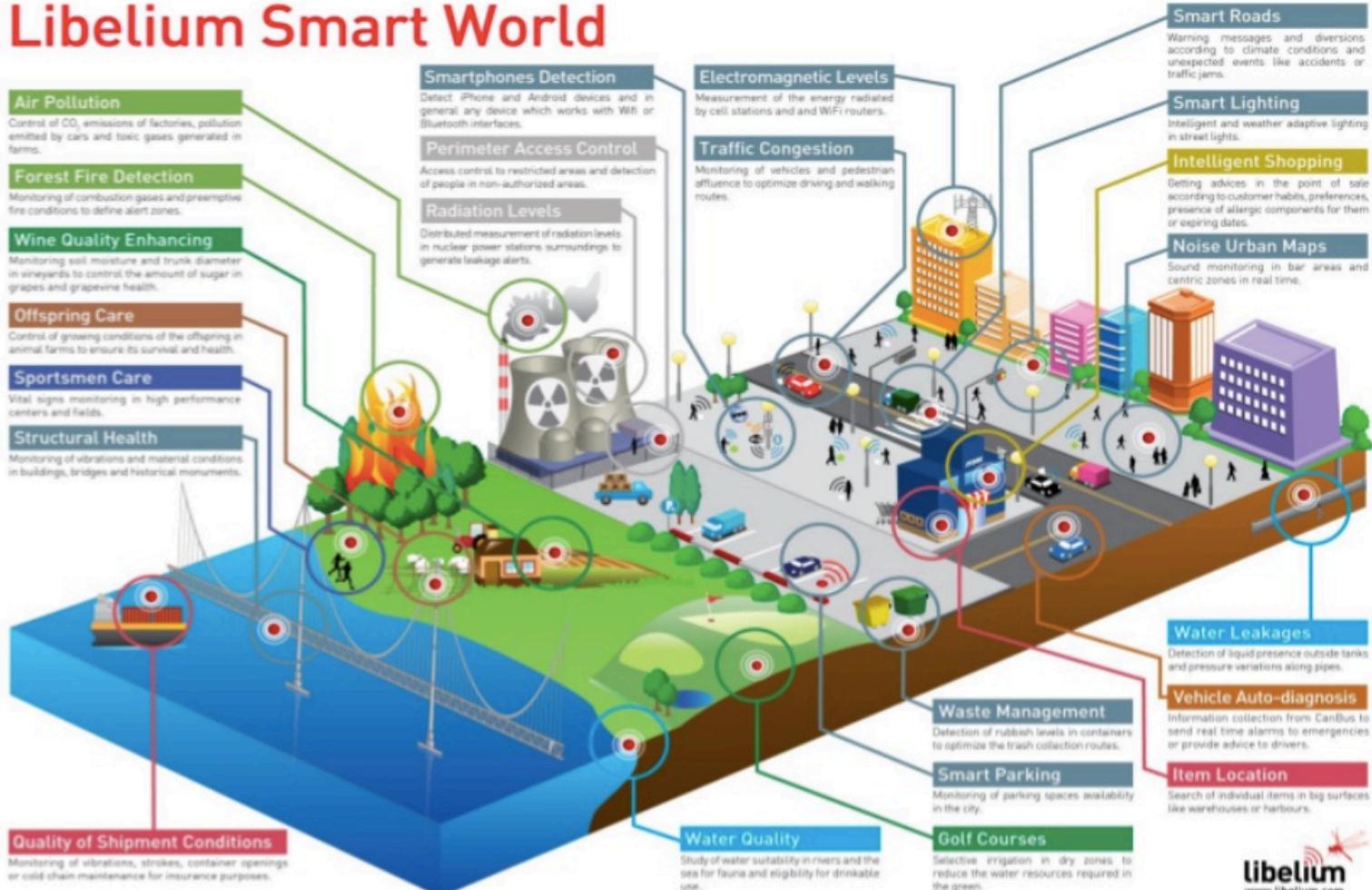
Figure 4:  
The Internet of Things and  
Services - Networking  
people, objects and systems



Source: Bosch Software Innovations 2012

# Smart cities

## Libelium Smart World



<http://www.libelium.com/libelium-smart-world-infographic-smart-cities-internet-of-things/>



# Different thinking

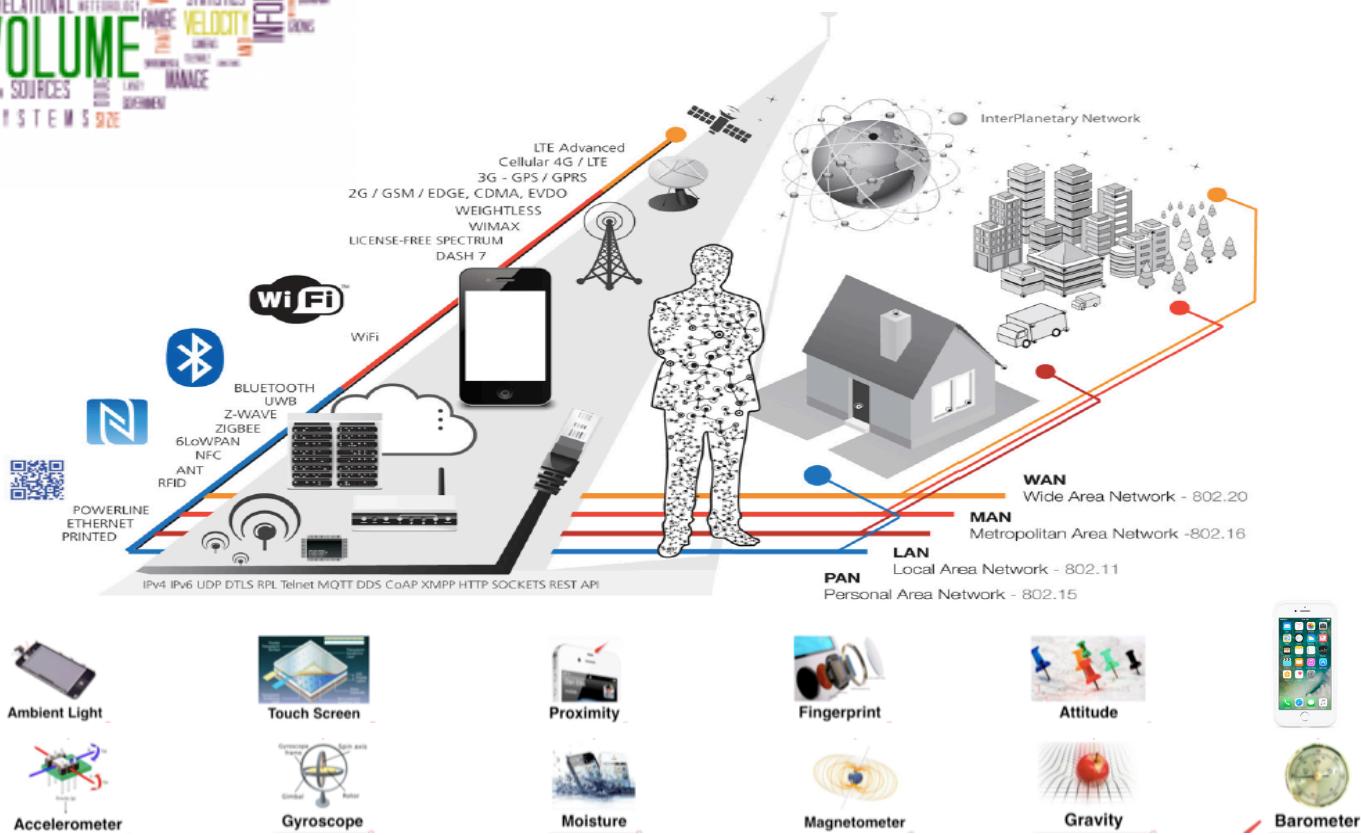
- Smart cities: 2008 <> 2018
  - Industry: 4.0 <> 3.0



# Data collection



# Connect



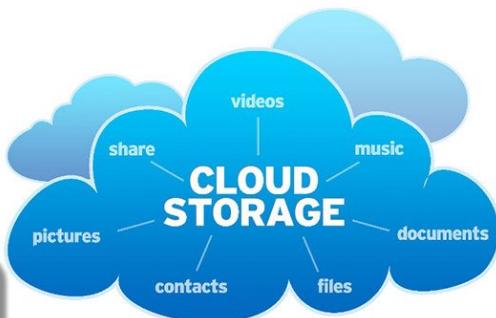


# Data analytics

Learn



Collect



Supercomputing  
Artificial Intelligence  
Data Mining  
**HPC**  
Big Data  
Scientific Simulations  
Business Intelligence  
Deep Learning



# High Performance Computing - HPC

## HPC wire

Since 1986 - Covering the Fastest Computers in the World and the People Who Run Them



Search this site  Search

Subscribe to receive

Home News ▾ Technologies ▾ Sectors ▾ Exascale Resources ▾ Specials ▾ Events

July 30, 2015

### White House Launches National HPC Strategy

John Russell and Tiffany Trader



Yesterday's executive order by President Barack Obama creating a National Strategic Computing Initiative (NSCI) is not only powerful acknowledgement of the vital role HPC plays in modern society but is also indicative of government's mounting worry that failure to coordinate and nourish HPC development on a broader scale would put the nation at risk. Not surprisingly, early reaction from the HPC community has been largely positive.



**Oakforest-PACS**  
13.55 Petaflops  
556,104 cores



**Piz Daint**  
19.59 Petaflops  
361,760 cores

## The European HPC Strategy

The Commission recognised the need for an EU-level policy in HPC to optimise national and European investments, addressing the entire HPC ecosystem. The Commission adopted its HPC Strategy on 15 February 2012 in the [Communication "High Performance Computing \(HPC\): Europe's place in a global race"](#) to ensure European leadership in the supply and use of HPC systems and services by 2020. The Competitiveness Council on 29/30 May 2013 adopted [conclusions](#) on this Communication, highlighting the role of HPC in the EU's innovation capacity and stressing its strategic importance to the EU's industrial and scientific capabilities as well as to its citizens.

High-Performance Computing (HPC) is a strategic resource for Europe's future. Mastering advanced computing technologies from hardware to software has become essential for innovation, growth and jobs.



**Summit** 122.3 Petaflops  
2,282,544 cores



**Sunway TaihuLight**  
93.0 Petaflops  
10,649,600 cores



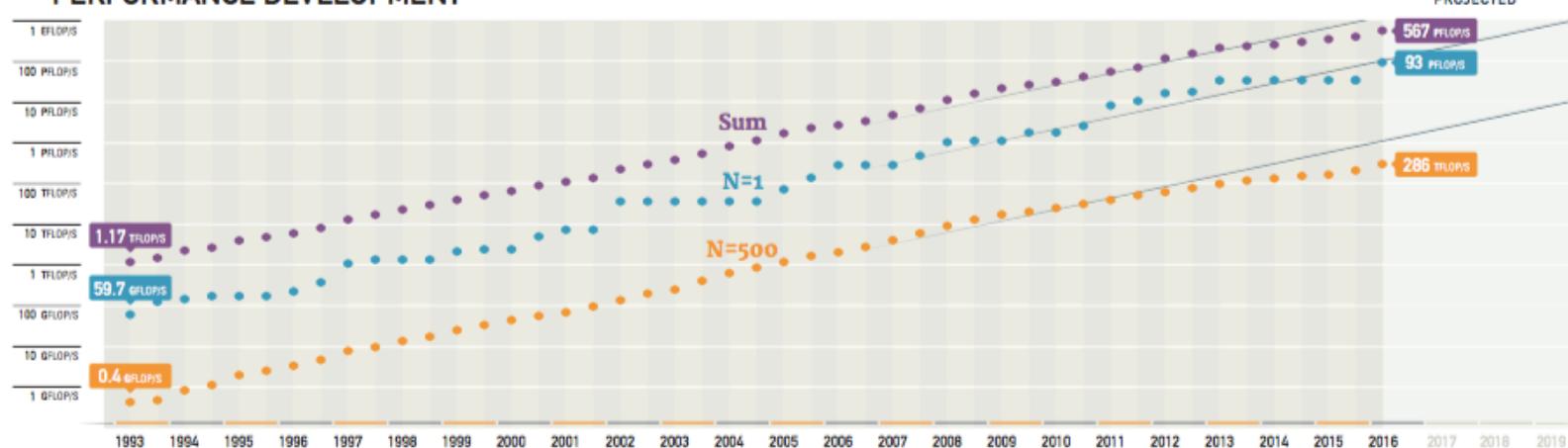
# <http://www.TOP500.org/>



FIND OUT MORE AT  
[top500.org](http://top500.org)

NAME	SPECS	SITE	COUNTRY	CORES	R <sub>MAX</sub> PFLOPS	POWER MW
1 Sunway TaihuLight	Shenwei SW26010 (260C 1.45 GHz) Custom interconnect	NSCC in Wuxi	China	10,649,600	93.0	15.4
2 Tianhe-2 (Milkyway-2)	Intel Ivy Bridge (12C 2.2 GHz) & Xeon Phi (57C 1.1 GHz), Custom interconnect	NSCC in Guangzhou	China	3,120,000	33.9	17.8
3 Titan	Cray XK7, Opteron 6274 (16C 2.2 GHz) + Nvidia Kepler GPU, Custom interconnect	DOE/SC/ORNL	USA	560,640	17.6	8.2
4 Sequoia	IBM BlueGene/Q, Power BQC (16C 1.60 GHz), Custom interconnect	DOE/NNSA/LLNL	USA	1,572,864	17.2	7.9
5 K computer	Fujitsu SPARC64 VIIIfx (8C 2.0 GHz), Custom interconnect	RIKEN AICS	Japan	705,024	10.5	12.7

## PERFORMANCE DEVELOPMENT



# Performance development (1)

June 2018



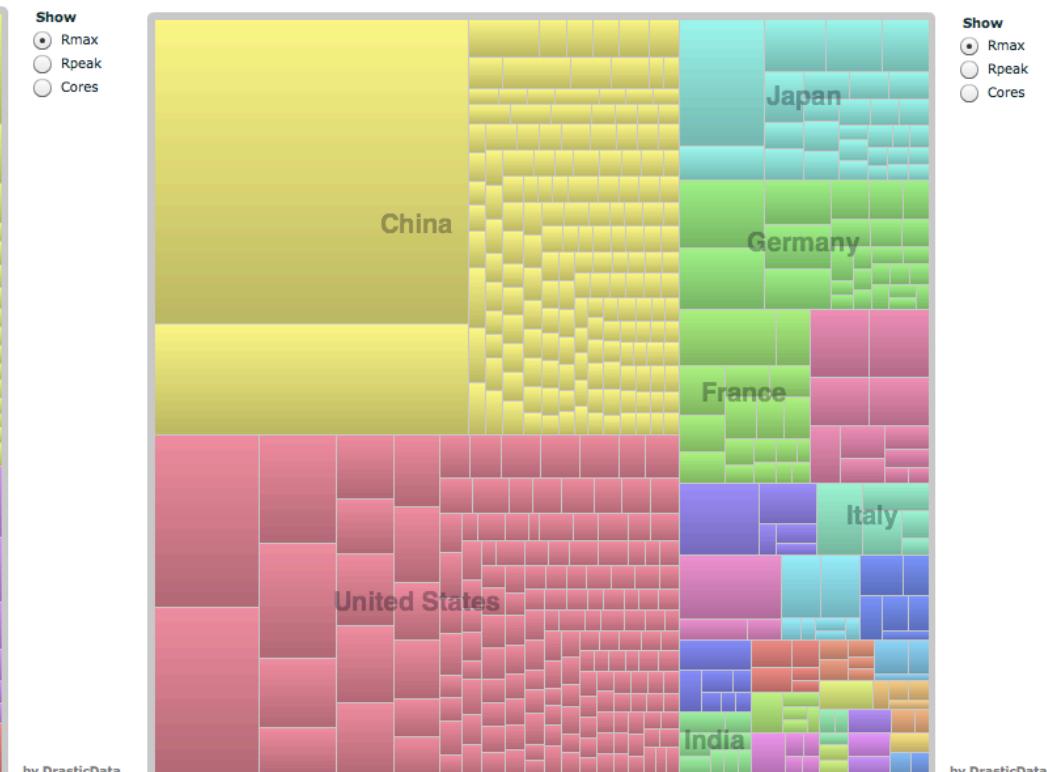
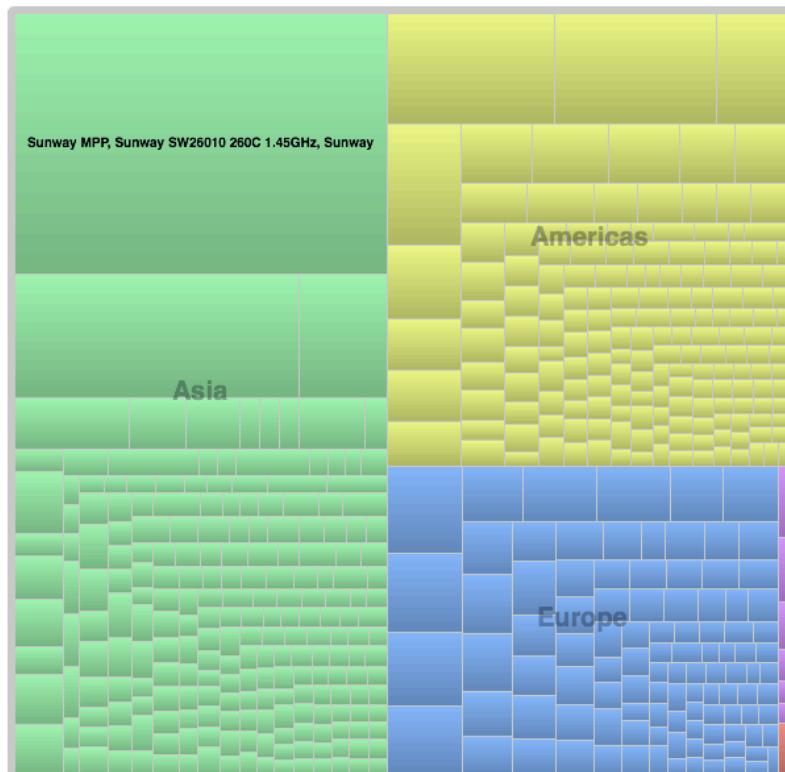
# Performance development (2)

June 2018





# HPC distribution in TOP500 (Jun 2016)

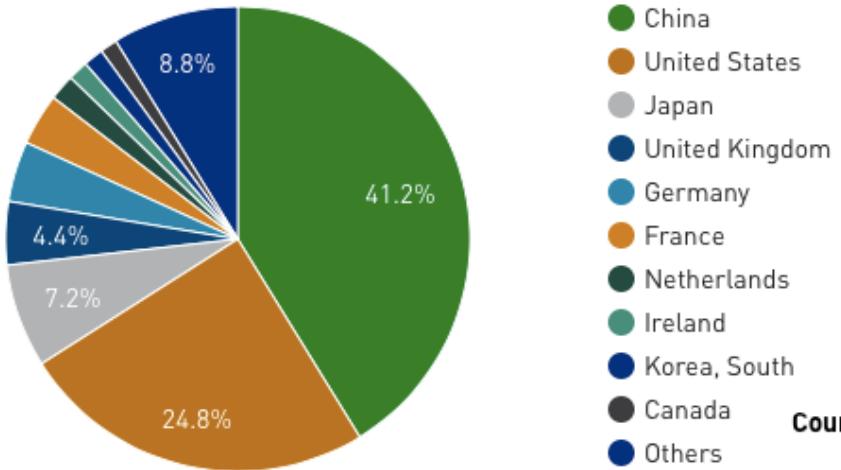




# HPC distribution in TOP500

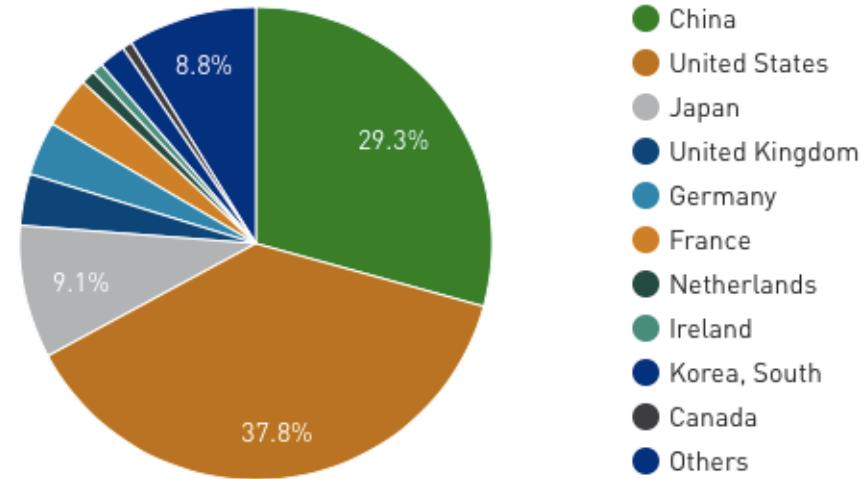
June 2018

Countries System Share



- China
- United States
- Japan
- United Kingdom
- Germany
- France
- Netherlands
- Ireland
- Korea, South
- Canada
- Others

Countries Performance Share

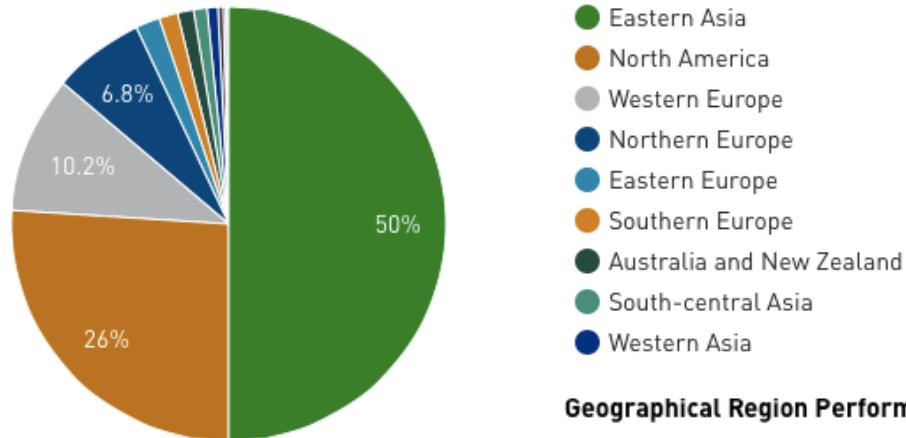


- China
- United States
- Japan
- United Kingdom
- Germany
- France
- Netherlands
- Ireland
- Korea, South
- Canada
- Others

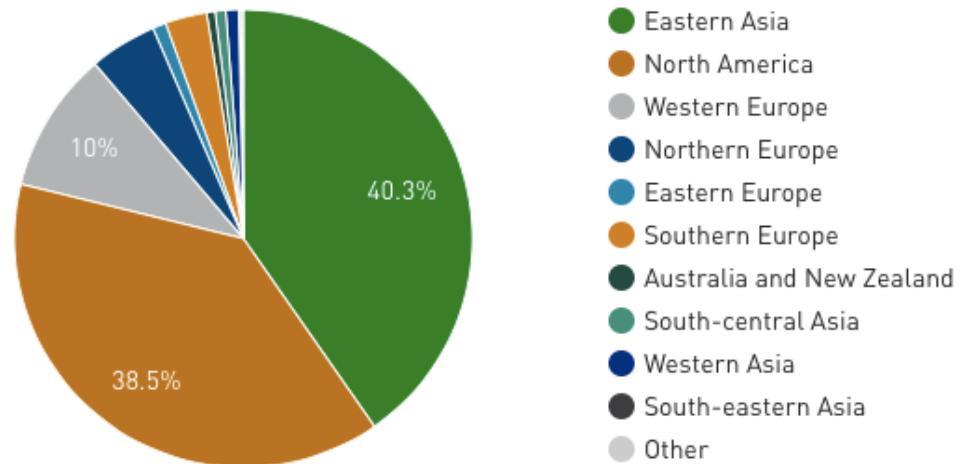
# HPC distribution in TOP500

June 2018

Geographical Region System Share



Geographical Region Performance Share

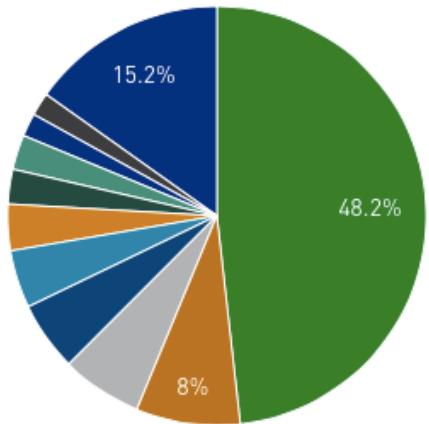




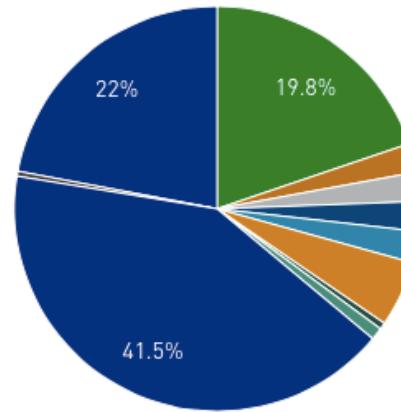
# HPC distribution in TOP500

## June 2018

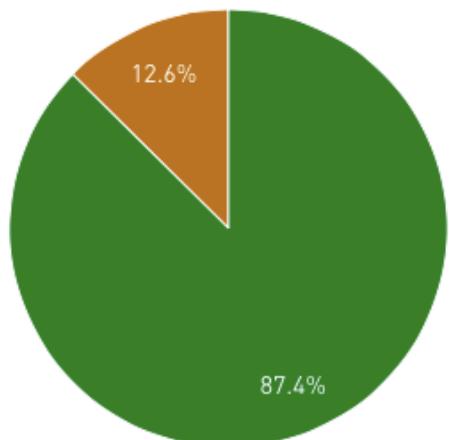
Accelerator/Co-Processor System Share



Accelerator/Co-Processor Performance Share



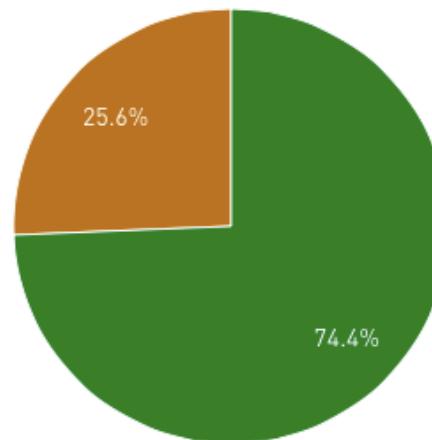
Architecture System Share



- NVIDIA Tesla P100
- NVIDIA Tesla V100
- NVIDIA Tesla K80
- NVIDIA Tesla K40
- NVIDIA Tesla P100 NVLink
- NVIDIA Tesla K20x
- PEZY-SC2 500Mhz
- NVIDIA 2050
- NVIDIA Volta GV100
- NVIDIA Tesla P40
- Others

- NVIDIA Tesla P100
- NVIDIA Tesla V100
- NVIDIA Tesla K80
- NVIDIA Tesla K40
- NVIDIA Tesla P100 NVLink
- NVIDIA Tesla K20x
- PEZY-SC2 500Mhz
- NVIDIA 2050
- NVIDIA Volta GV100
- NVIDIA Tesla P40
- Others

Architecture Performance Share

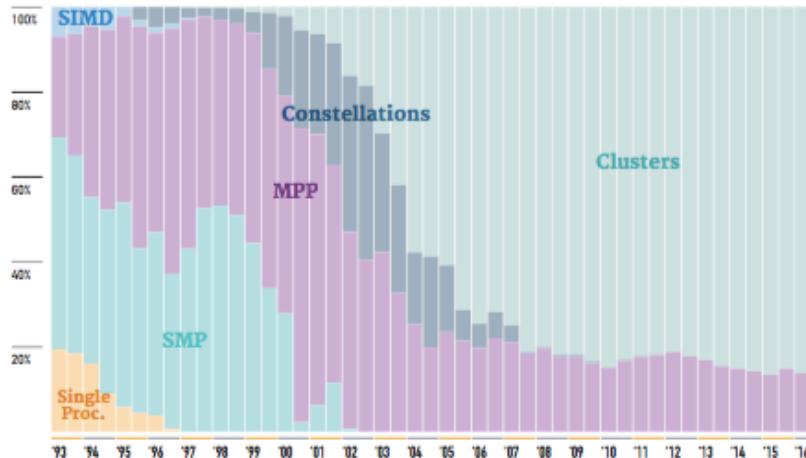


- Cluster
- MPP

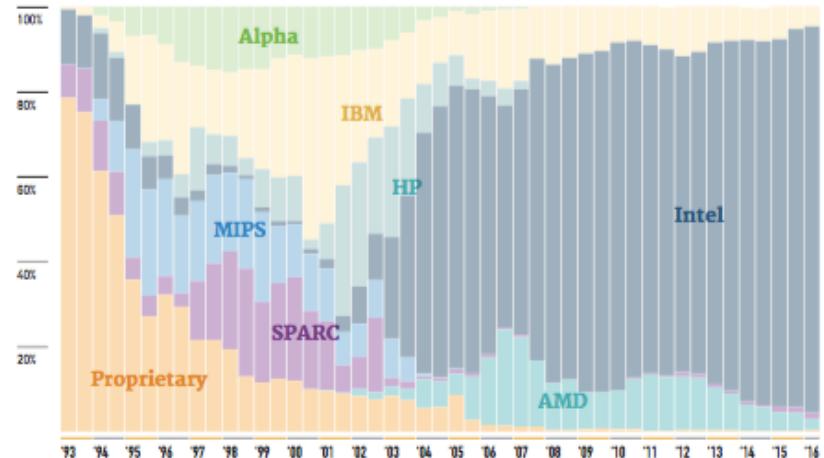


# TOP500 (Jun 2016)

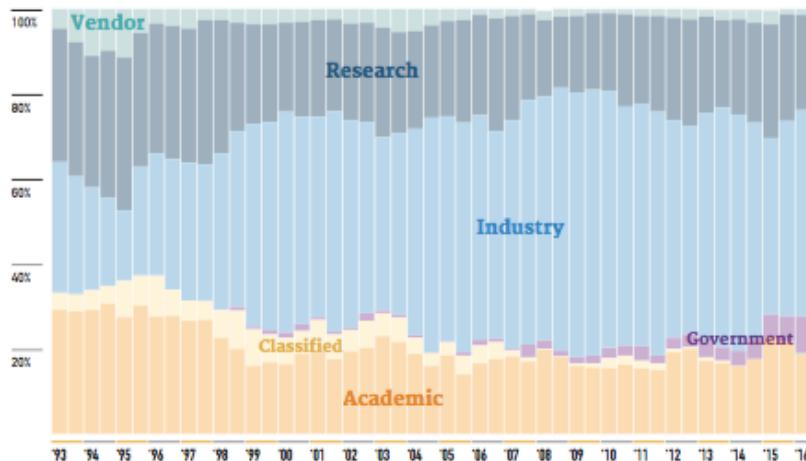
## ARCHITECTURES



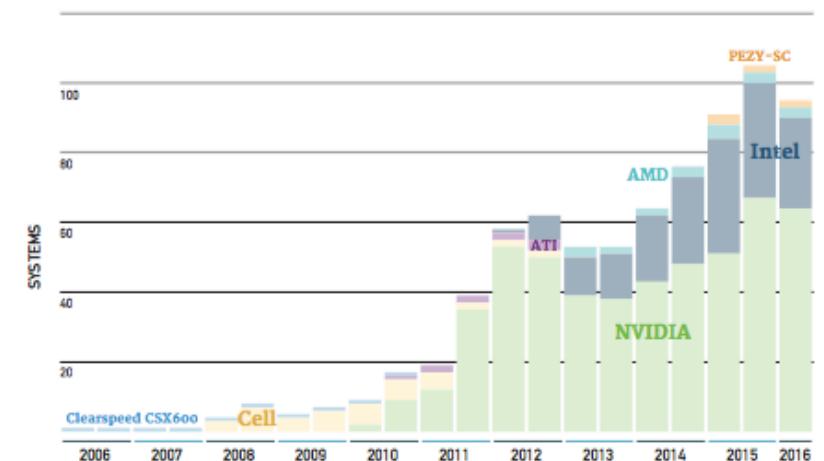
## CHIP TECHNOLOGY



## INSTALLATION TYPE



## ACCELERATORS/CO-PROCESSORS





## Exascale Race/Technologies

# IDC-Projected Exascale Dates and Suppliers

### U.S.



- Sustained ES: 2023
- Peak ES: 2021
- Vendors: U.S.
- Processors: U.S.
- Initiatives: NSCI/ECP
- Cost: \$300-500M per system, plus heavy R&D investments

### EU



- Sustained ES: 2023-24
- Peak ES: 2021
- Vendors: U.S., Europe
- Processors: U.S., ARM
- Initiatives: PRACE, ETP4HPC
- Cost: \$300-\$350 per system, plus heavy R&D investments

### China



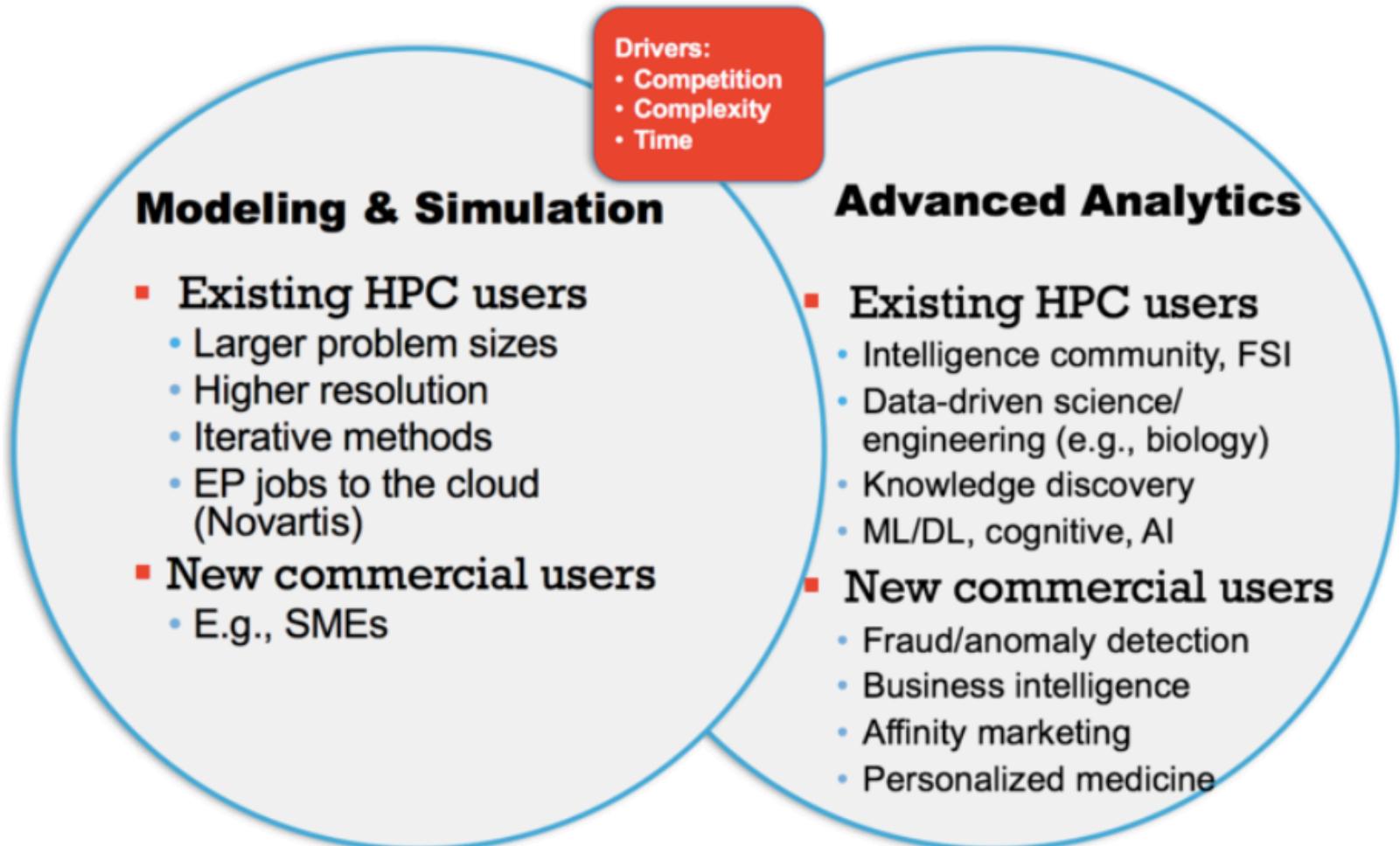
- Sustained ES: 2023
- Peak ES: 2020
- Vendors: Chinese
- Processors: Chinese (plus U.S.?)
- 13<sup>th</sup> 5-Year Plan
- Cost: \$350-500M per system, plus heavy R&D

### Japan



- Sustained ES: 2023-24
- Peak ES: Not planned
- Vendors: Japanese
- Processors: Japanese
- Cost: \$600-850M, this includes both 1 system and the R&D costs...will also do many smaller size systems

# HPDA = Data-Intensive Computing Using HPC

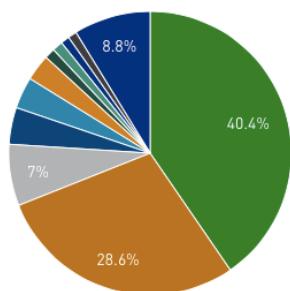




# Supercomputing Conference (SC)

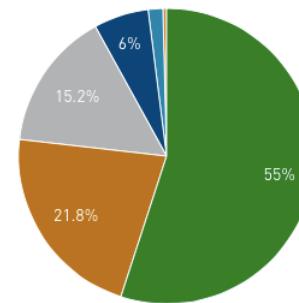
- TOP500 (1993) -> Green500 (SC06) -> Graph500 (SC10)
- SC17: 12-17 Nov 2017, Denver, Colorado - US
  - IO-500
  - Fog/Edge computing for smart cities

Countries System Share



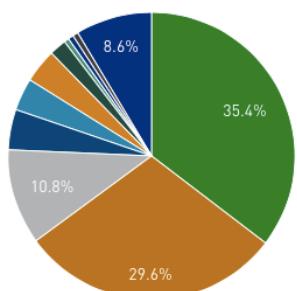
- China
- United States
- Japan
- Germany
- France
- United Kingdom
- Italy
- Netherlands
- Canada
- Poland
- Others

Segments System Share



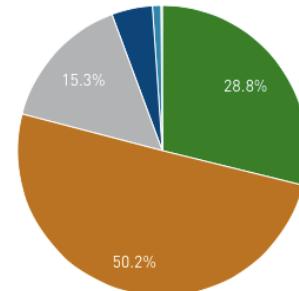
- Industry
- Research
- Academic
- Government
- Vendor
- Classified

Countries Performance Share



- China
- United States
- Japan
- Germany
- France
- United Kingdom
- Italy
- Netherlands
- Canada
- Poland
- Others

Segments Performance Share



- Industry
- Research
- Academic
- Government
- Vendor
- Classified

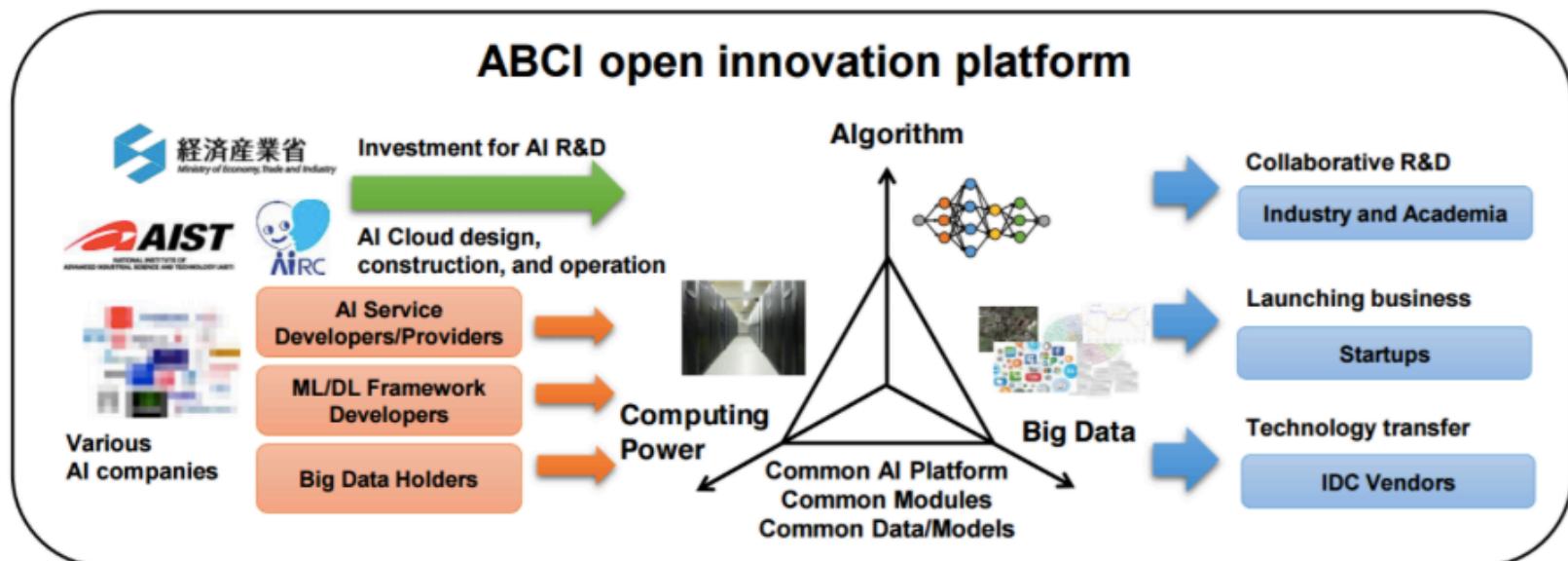


# Japan Plans Super-Efficient AI Supercomputer

By Tiffany Trader

November 28, 2016

*Editor's note: a source familiar with the project is reporting that the target of 130 petaflops is actually half-precision (16-bit). The target for double precision (64-bit) is 33 petaflops. Additionally, the budget of 19.5 billion yen is for the machine and the building and facility. The target for installation is now 2018 Q1. The story has been updated to reflect these changes and we will report further as it develops.*



Source: AIST document

# Parallel Processing

---

Thoai Nam

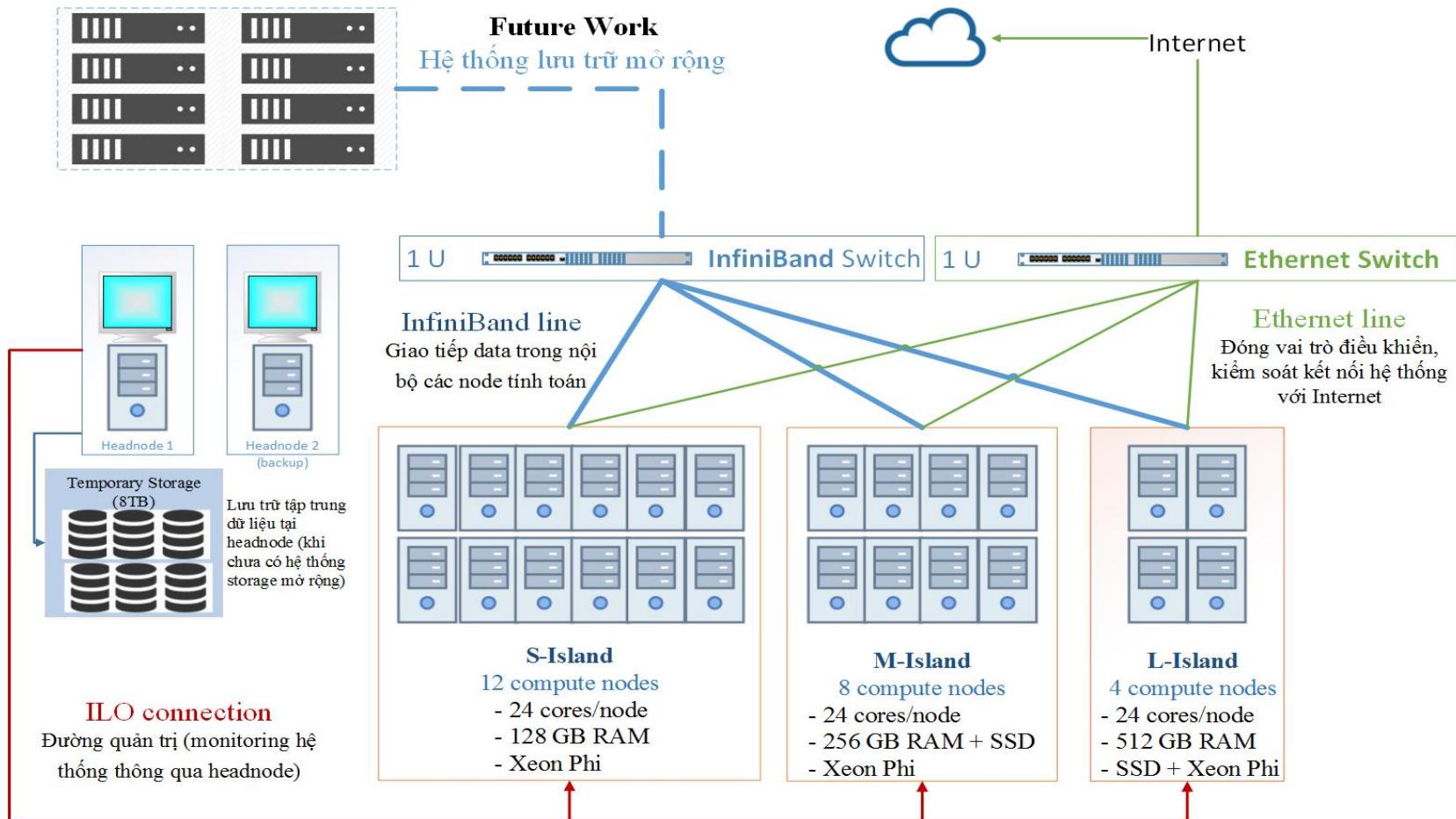
Faculty of Computer Science and Engineering

HCMC University of Technology



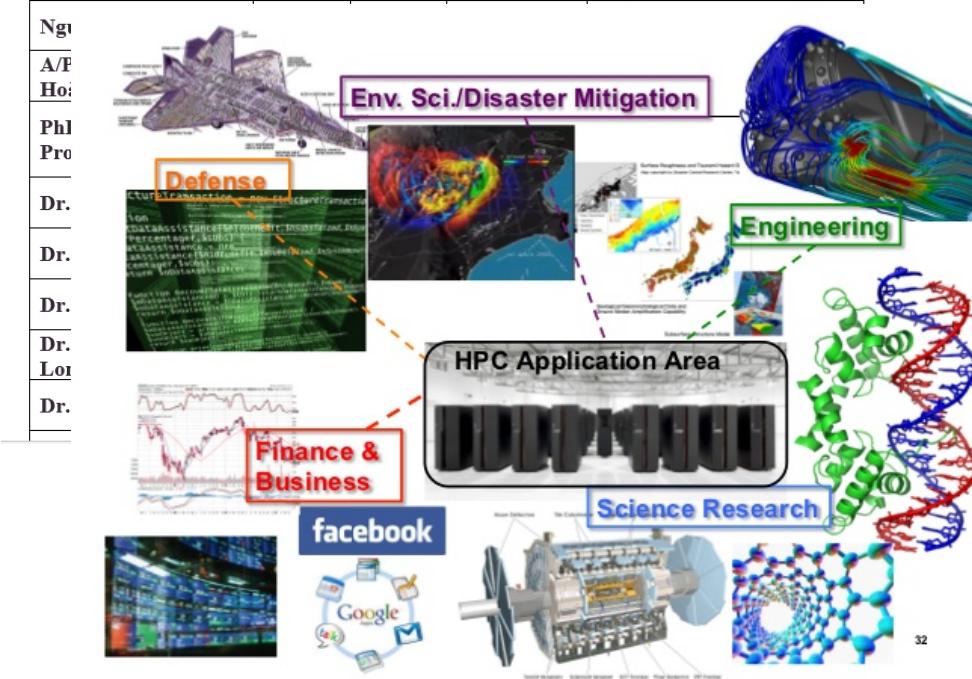
# SuperNode-XP

## 50 TFlops machine

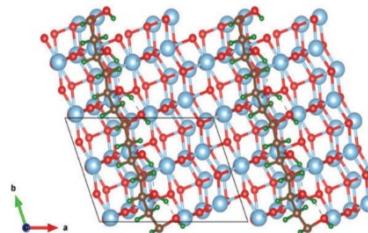


# Applications on SuperNode-XP

User	Start Time	End Time	Apps	Organization/Unit
A/Prof. Nguyễn Thông	30/09/14		OpenTelemac	HCMUT – Faculty of Civil Engineering
Ms. Phạm Ngọc Thanh, Dr. Ing Jorg Franke	06/06/16		OpenFoam	Vietnam Germany University
Dr. Lê Thành Văn	20/08/16		Hadoop	HCMUT – Faculty of Computer Science
A/Prof. Trần Văn Hoài	25/10/16		BLAS	HCMUT – Faculty of Computer Science
Mr. Quân	01/11/16		DFFT	HCMUT – Faculty of Electrical and Electronic Engineering



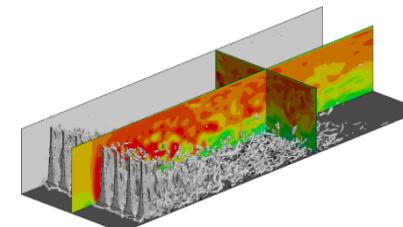
**INOMAR center**



VASP, Quantum Espresso Simulations

A/Prof. Dzung Hoang

**Vietnam Germany University**



OpenFOAM sims

Prof. Joerg Franke



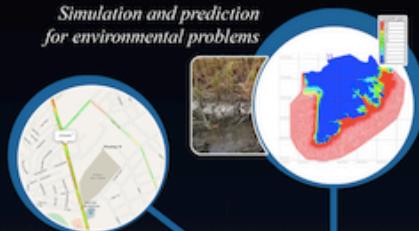
# High Performance Computing LAB



Hewlett Packard  
Enterprise

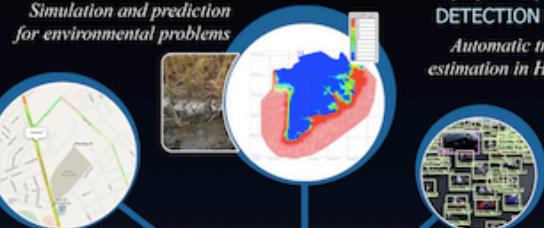
SALINIZATION IN  
MEKONG DELTA

*Simulation and prediction  
for environmental problems*



AUTOMATIC MOTORBIKE  
DETECTION IN VIETNAM

*Automatic traffic density  
estimation in Ho Chi Minh city*



ITS HCMUT  
*Traffic information for  
Ho Chi Minh city*

Internet  
&  
Cloud Services

## LIBRARIES

ANSYS (128 cores)  
CADENCE  
OpenFOAM  
BLAS

HADOOP & SPARK  
OpenTELEMAC  
Intel Parallel Studio  
...

INTEL XEON PHI (MIC)  
CO-PROCESSOR



2 MIC CARDS/NODE  
61 CORES/CARD

## SUPERNODE-XP & HPC APPLICATIONS

INFINIBAND  
SWITCHES  
56 (GBPS)



PCIe  
WIDTH - x16  
SPEED - 5 GT/s

S-Island



12 COMPUTE NODES

M-Island



8 COMPUTE NODES

L-Island



4 COMPUTE NODES

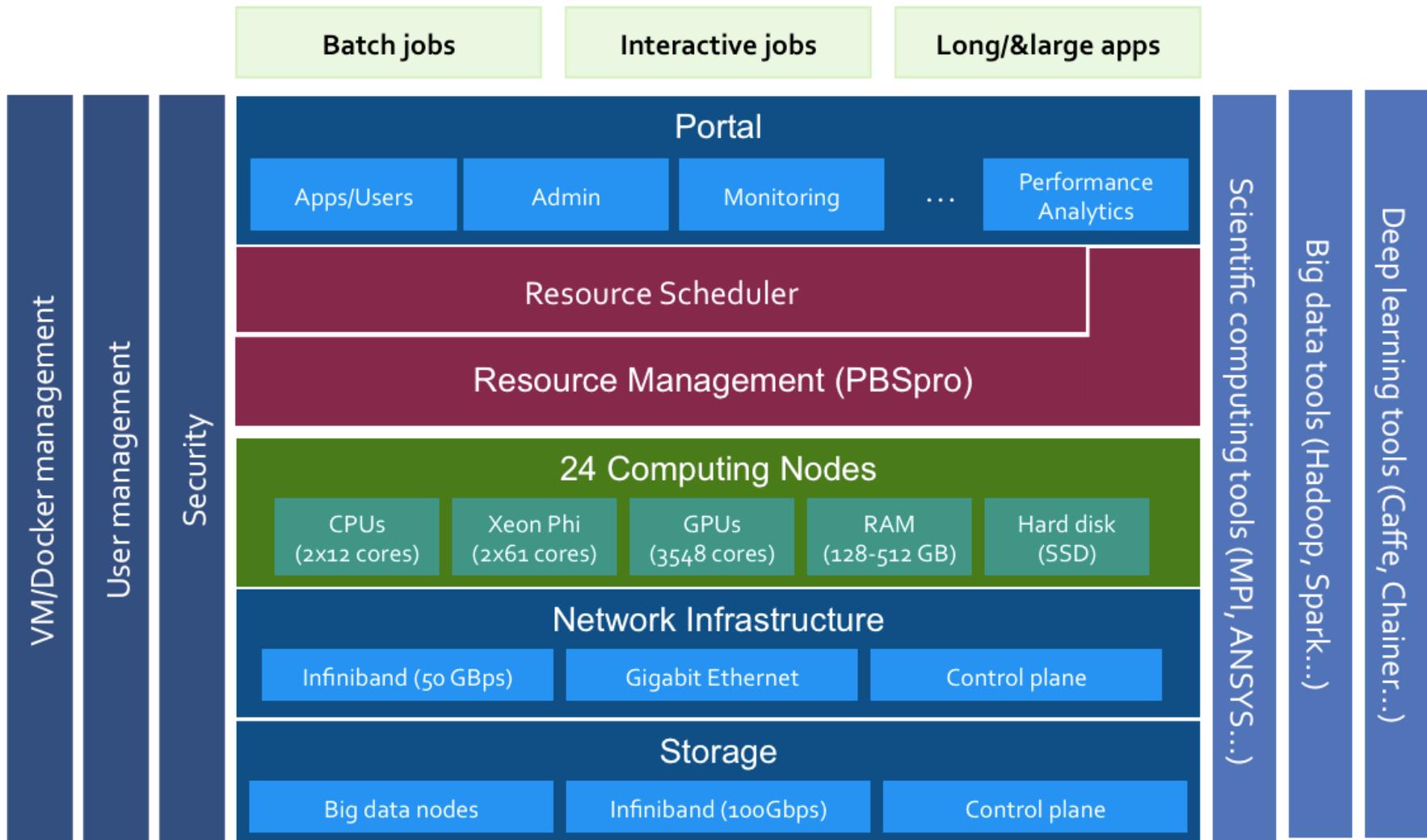
24 CORES/NODE  
128 GB RAM

24 CORES/NODE  
256 GB RAM + SSD

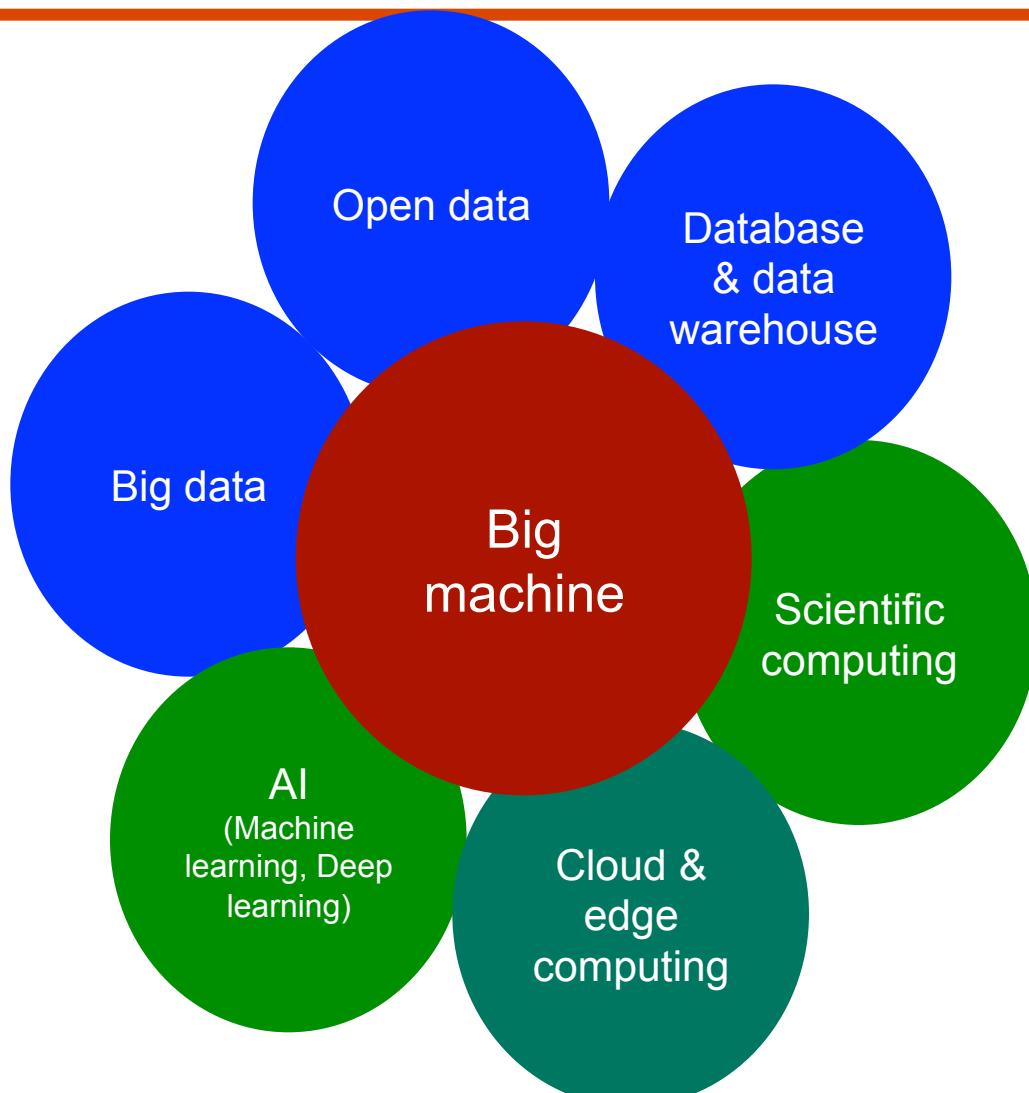
24 CORES/NODE  
512 GB RAM +SSD



# SuperNode-XP Architecture



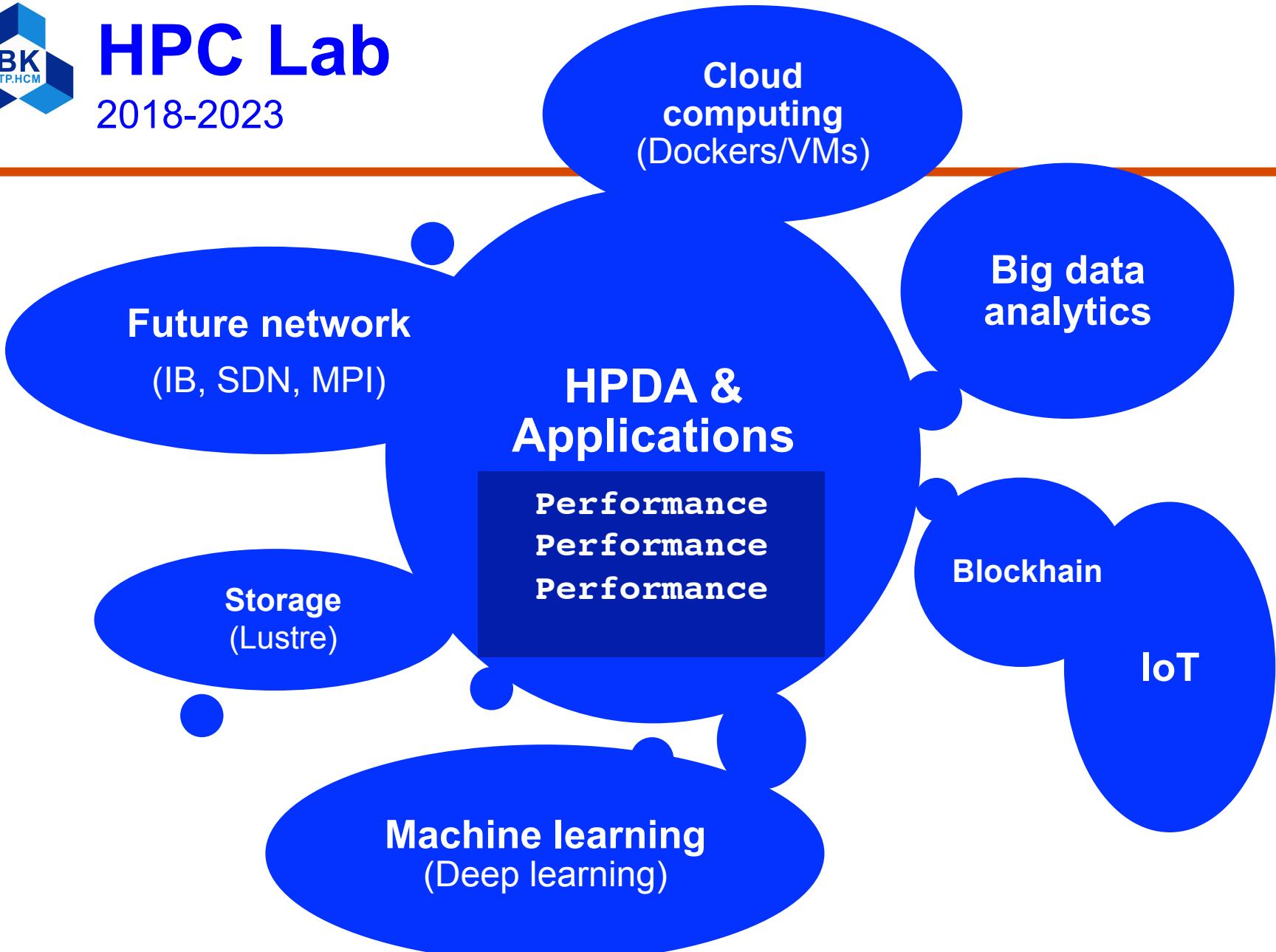
# Smart cities: Data center





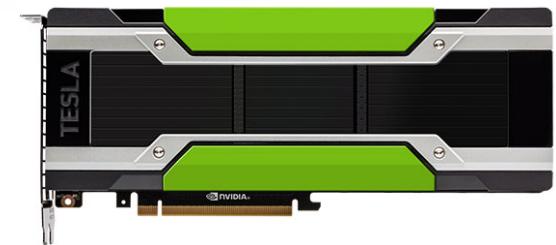
# HPC Lab

2018-2023



# HPDA: Big data analytics (Lrz-Germany, Intel+HPE)

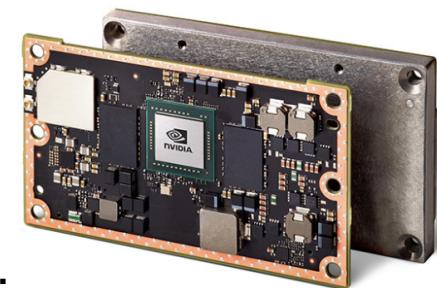
- SuperNode-XP
  - Xeon + Xeon Phi, GPUs
  - IB, Lustre storage, SSD
- Technology
  - Hadoop, Spark, PBSpro
  - Edge computing, Docker/VMs
  - Intel@ Parallel Studio XE
  - Libraries: ANSYS, OpenTelemac, Gromacs...
- Applications
  - (1) Data mining for large-scale data sets: Association rules, K-means, SVM...
  - (2) Security analytics
  - (3) Real problems in HCNC & VN on SuperNode-XP



# Machine Learning + IoT

## (Dr. Lê Thành Sách + Nvidia)

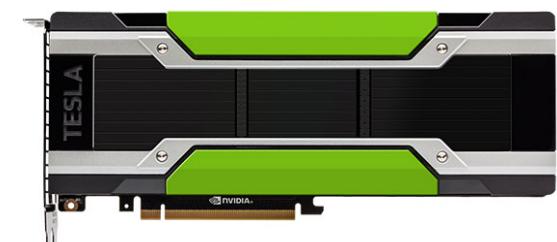
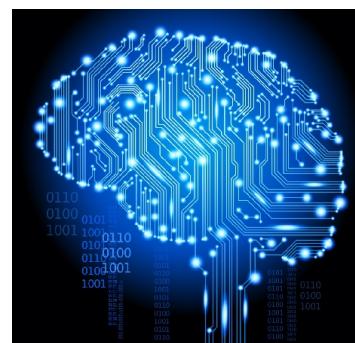
- SuperNode-XP + GPUs
  - P100, P4, GTX 1080Ti, Jetson TX2
- ML: Deep Learning
  - TensorFlow, Torch, Theano, Caffe, Chainer...
- Applications
  - (1) Image/data analytics
  - (2) Computing box (Edge computing)
  - (3) Real problems in HCMC & VN



Jetson TX2



GTX 1080 Ti



P100, P4



# Future Network

## (Dr. Phan Trường Khoa (UCL – UK))

- SDN (Software-Defined Networking)
  - P4: a language for programming the data plane of network devices
  - OpenFlow, OpenDayLight
- MPI
  - MPI one-sided communication
- Applications:
  - Routing analysis
  - Security
  - Optimization for big data applications
- Ref:
  - P4: <http://p4.org/>
  - IEEE SDN: <http://sdn.ieee.org>
  - OpenFlow: <http://archive.openflow.org>
  - OpenDayLight: <https://www.opendaylight.org>



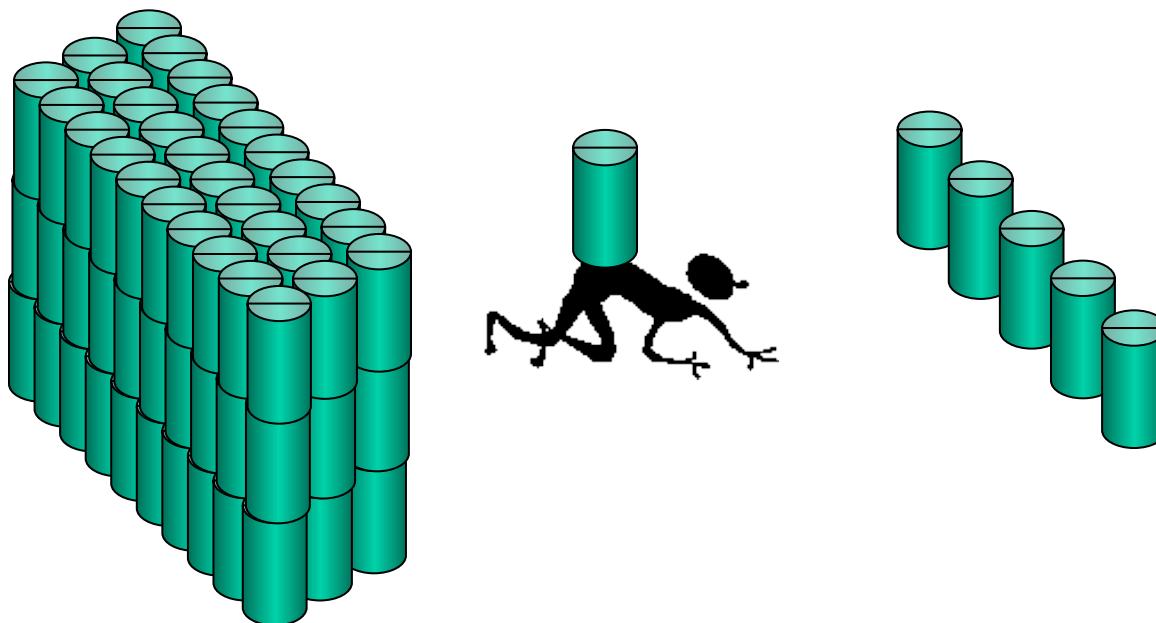
# How to do

---

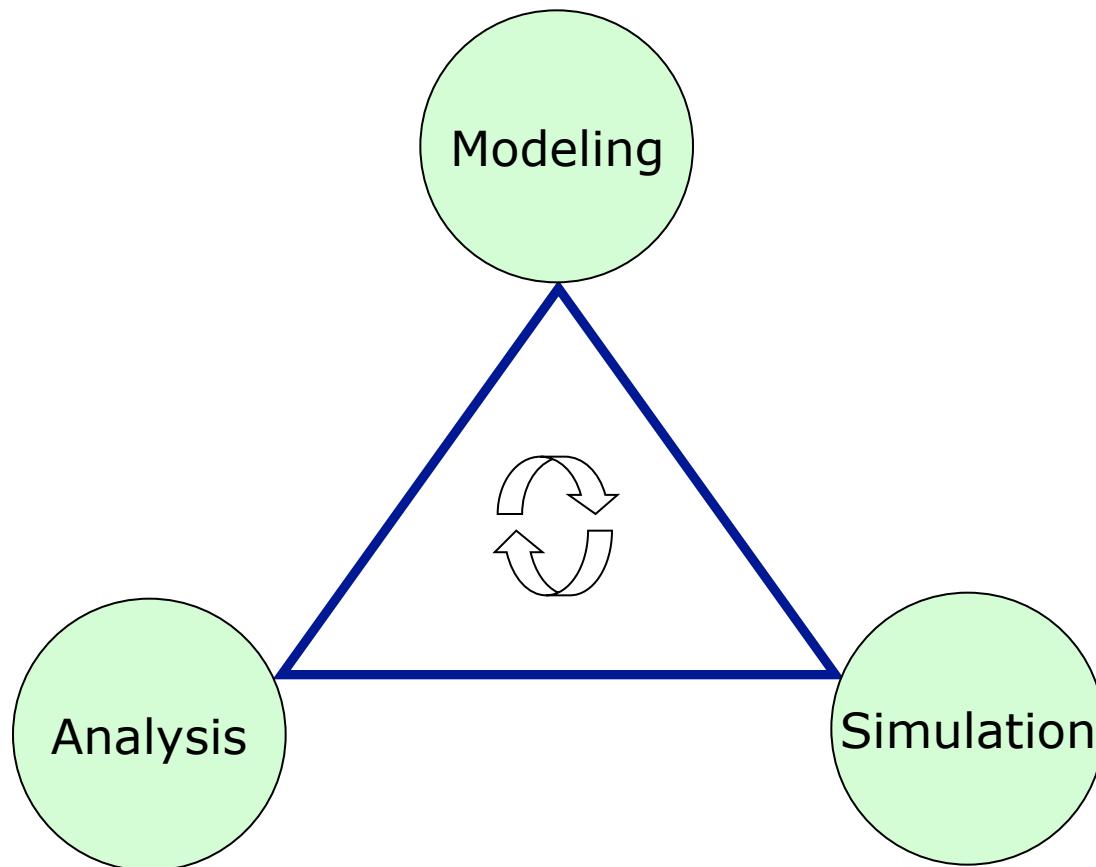
## Parallel processing

# Sequential Processing

- 1 CPU
- Simple
- Big problems???



# New Approach





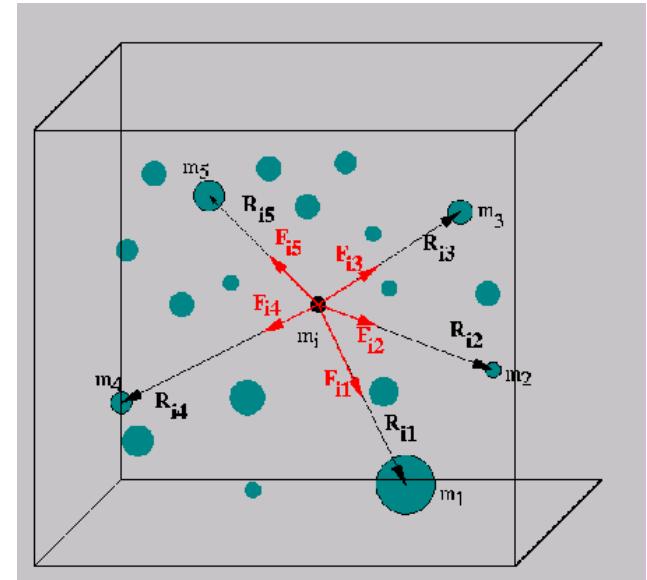
# Grand Challenge Problems

---

- A grand challenge problem is one that cannot be solved in a reasonable amount of time with today's computers
- Ex:
  - Modeling large DNA structures
  - Global weather forecasting
  - Modeling motion of astronomical bodies

## □ The N<sup>2</sup> algorithm:

- N bodies
- N-1 forces to calculate for each bodies
- N<sup>2</sup> calculations in total
- After the new positions of the bodies are determined, the calculations must be repeated





# Galaxy



- **$10^7$**  stars and so  **$10^{14}$**  calculations have to be repeated
- Each calculation could be done in  $1\mu\text{s}$  ( $10^{-6}\text{s}$ )
- It would take **~3 years** for one iteration (~26800 hours)
- But it only takes **10 hours** for one iteration with **2680** processors



# Solutions

---

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



# Parallel Processing Terminology

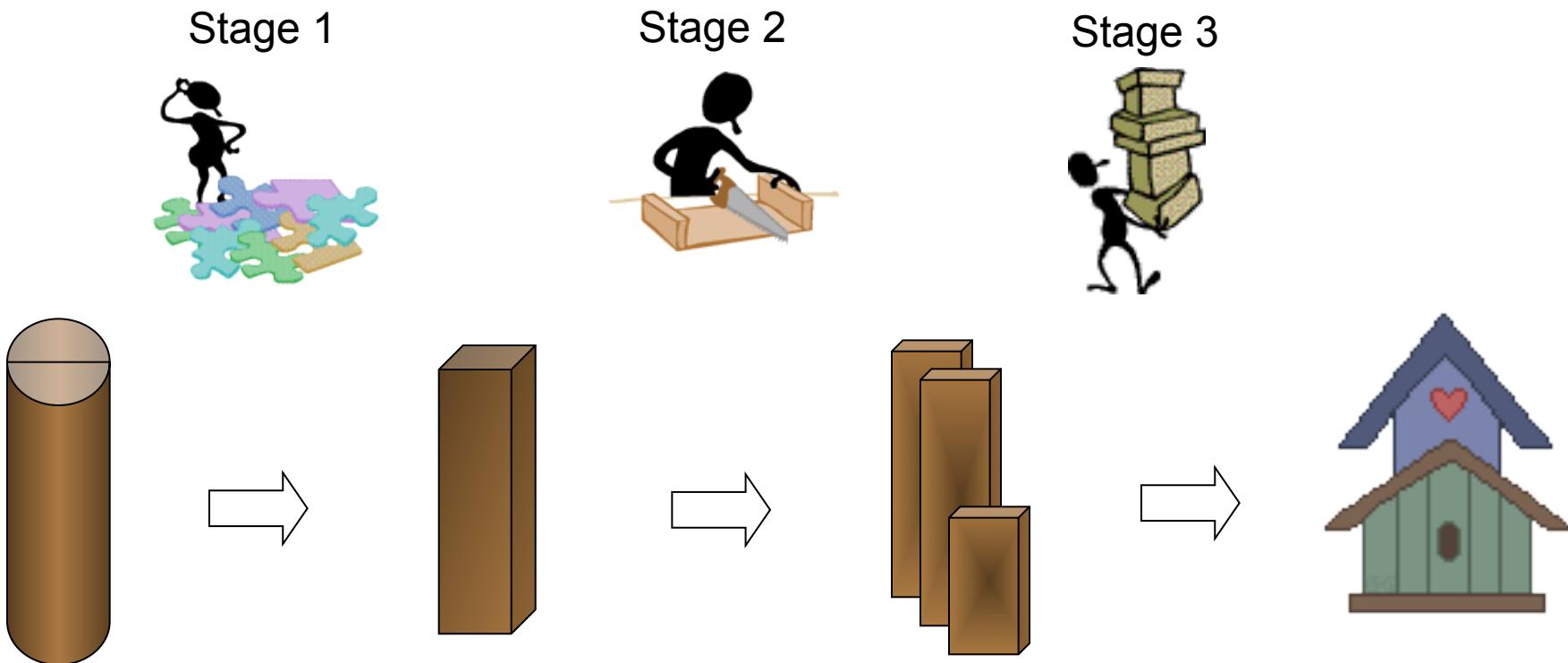
---

- Parallel processing
- Parallel computer
  - Multi-processor computer capable of parallel processing
- Throughput:
  - The throughput of a device is the number of results it produces per unit time.
- Speedup

$S = \text{Time}(\text{the most efficient sequential algorithm}) / \text{Time}(\text{parallel algorithm})$
- Parallelism:
  - Pipeline
  - Data parallelism
  - Control parallelism

# Pipeline

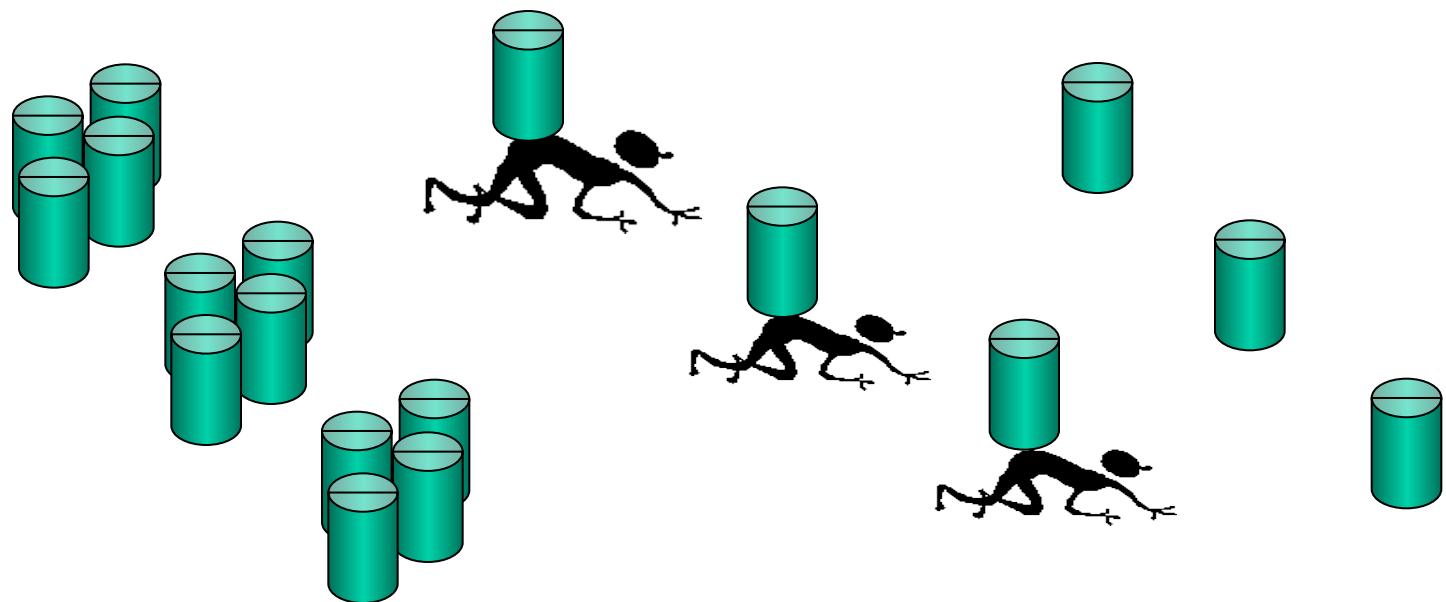
- A number of steps called **segments** or **stages**
- The output of one segment is the input of other segment



# Data Parallelism

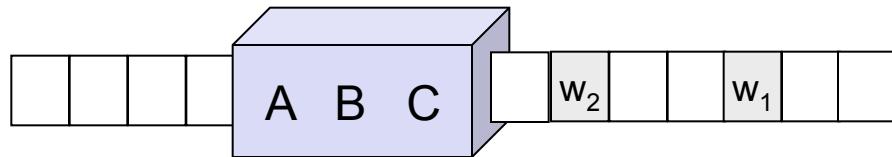
- Distributing the data across different parallel computing nodes

Applying the same operation simultaneously to elements of a data set

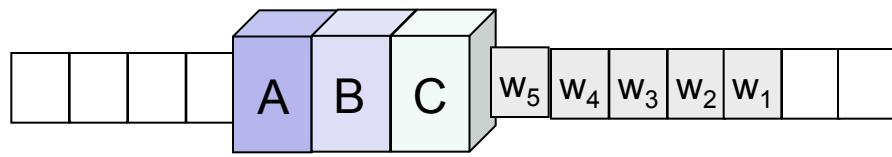


# Pipeline & Data Parallelism

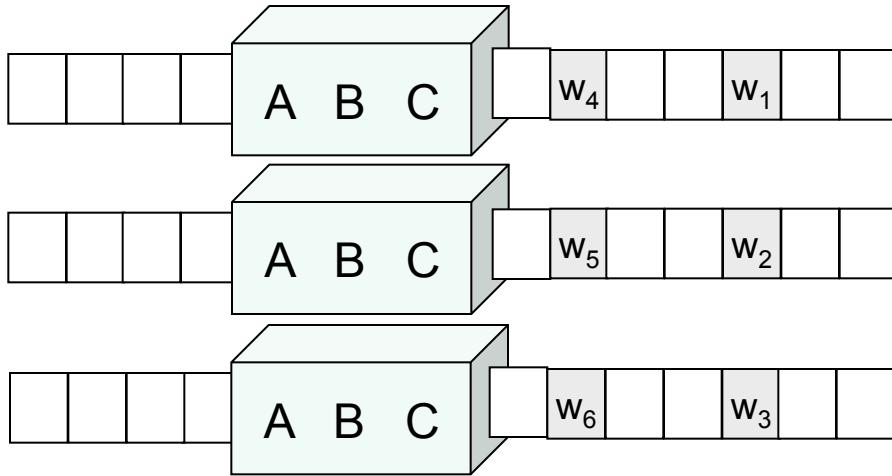
1. Sequential execution



2. Pipeline



3. Data Parallelism





# Pipeline & Data Parallelism

□ Pipeline is a special case of control parallelism

□  $T(s)$ : Sequential execution time

$T(p)$ : Pipeline execution time (with 3 stages)

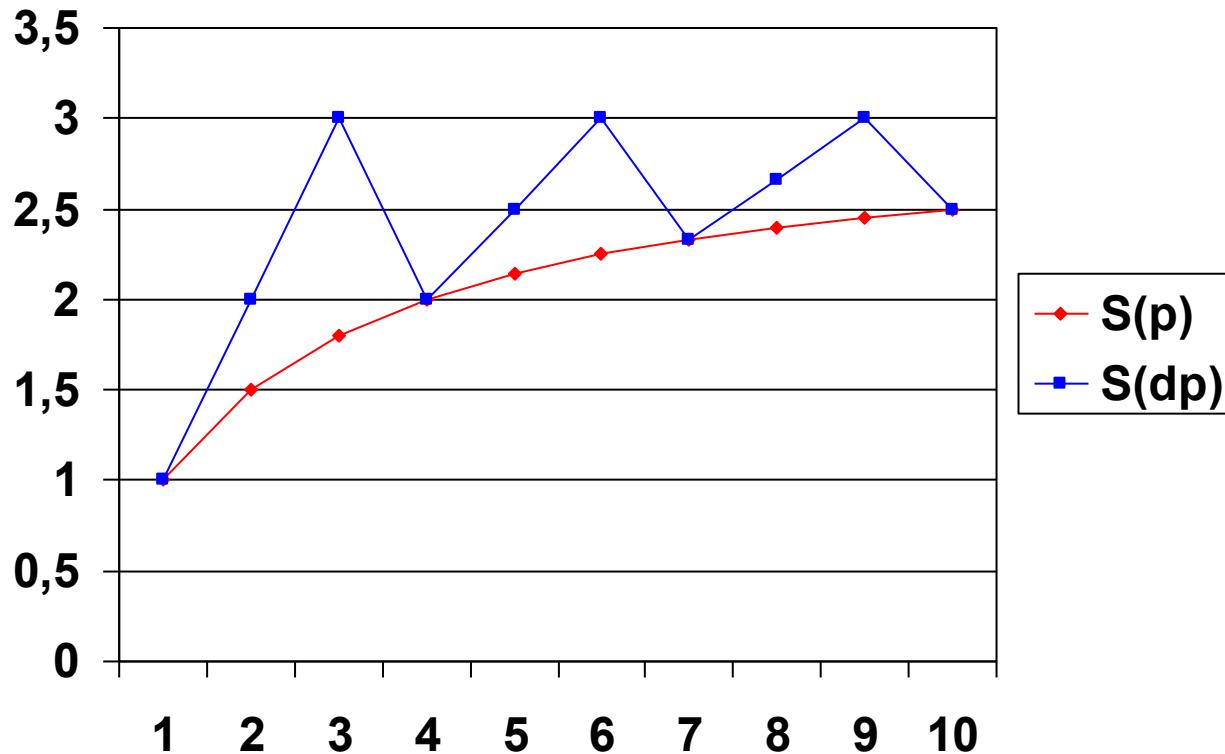
$T(dp)$ : Data-parallelism execution time (with 3 processors)

$S(p)$ : Speedup of pipeline

$S(dp)$ : Speedup of data parallelism

Widget	1	2	3	4	5	6	7	8	9	10
$T(s)$	3	6	9	12	15	18	21	24	27	30
$T(p)$	3	4	5	6	7	8	9	10	11	12
$T(dp)$	3	3	3	6	6	6	9	9	9	12
$S(p)$	1	$1+1/2$	$1+4/5$	2	$2+1/7$	$2+1/4$	$2+1/3$	$2+2/5$	$2+5/11$	$2+1/2$
$S(dp)$	1	2	3	2	$2+1/2$	3	$2+1/3$	$2+2/3$	3	$2+1/2$

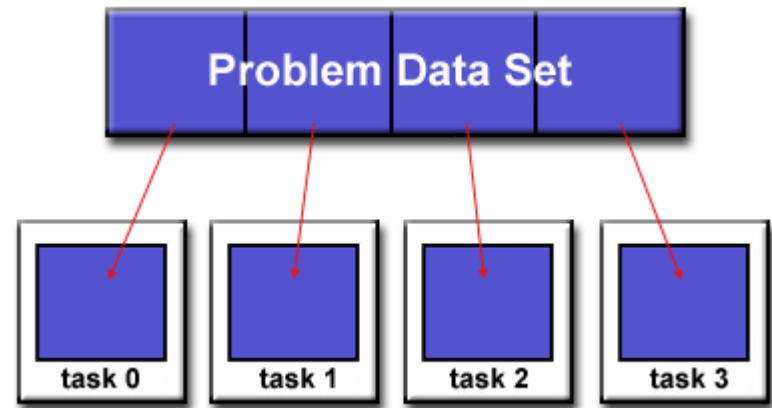
# Pipeline & Data Parallelism



# Control Parallelism

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

*Applying different operations to different data elements simultaneously*





# Example

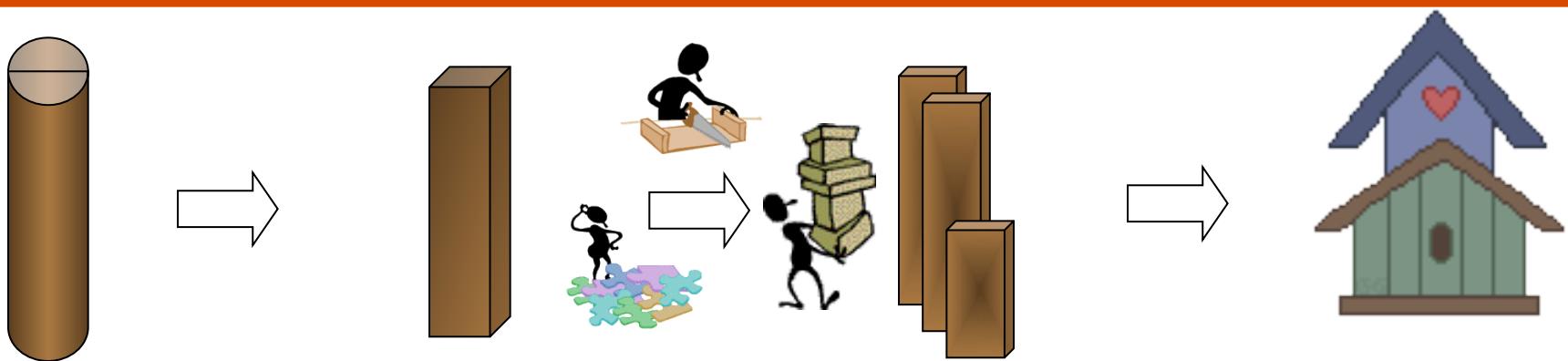
---

{Milk, Sugar, Bread}

{Milk, Sugar, Bread, Tea}  
{Bread, Milk, Coffee, Meat}  
{Milk, Sugar}  
{Milk, Bread, Sugar, Salt}  
{Apple, Orange, Banana, Sugar, Milk}  
...  
{Milk, Bread, Sugar, Beer}

- Pipeline?
- Control parallelism?
- Data parallelism?

# Throughput: Woodhouse problem



- ❑ 5 persons complete 1 woodhouse in 3 days
- ❑ 10 persons complete 1 woodhouse in 2 days
- ❑ How to build 2 houses with 10 persons?
  - (1) 10 persons building the 1<sup>st</sup> woodhouse and then the 2<sup>nd</sup> one later (sequentially)
  - (2) 10 persons building 2 woodhouses concurrently; it means that each group of 5 persons complete a woodhouse



# Throughput

---

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



# Scalability

---

- An algorithm is scalable if the level of parallelism increases at least linearly with the problem size.
- An architecture is scalable if it continues to yield the same performance per processor, albeit used in large problem size, as the number of processors increases.
  
- Data-parallelism algorithms are more scalable than control-parallelism algorithms

# OpenMP

Kenjiro Taura

# Contents

- 1 Overview
- 2 parallel pragma
- 3 Work sharing constructs
  - loops (**for**)
  - scheduling
  - task parallelism (**task** and **taskwait**)
- 4 Data sharing clauses
- 5 SIMD constructs

# Contents

1 Overview

2 parallel pragma

3 Work sharing constructs

- loops (`for`)
- scheduling
- task parallelism (`task` and `taskwait`)

4 Data sharing clauses

5 SIMD constructs

# Goal

- learn OpenMP, by far the most widespread standard API for shared memory parallel programming
- learn that various schedulers execute your parallel programs differently

# OpenMP

- *de fact* standard model for programming shared memory machines
- C/C++/Fortran + parallel directives + APIs
  - by `#pragma` in C/C++
  - by comments in Fortran
- many free/vendor compilers, including GCC

# OpenMP reference

- official home page: <http://openmp.org/>
- specification:  
<http://openmp.org/wp/openmp-specifications/>
- latest version is 4.5  
(<http://www.openmp.org/mp-documents/openmp-4.5.pdf>)
- section numbers below refer to those in OpenMP spec 4.0  
(<http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>)

# GCC and OpenMP

- <http://gcc.gnu.org/wiki/openmp>
- gcc 4.2 → OpenMP spec 2.5
- gcc 4.4 → OpenMP spec 3.0 (task parallelism)
- gcc 4.7 → OpenMP spec 3.1
- gcc 4.9 → OpenMP spec 4.0 (SIMD)

# Compiling/running OpenMP programs with GCC

- compile with `-fopenmp`

```
1 $ gcc -Wall -fopenmp program.c
```

- run the executable specifying the number of threads with `OMP_NUM_THREADS` environment variable

```
1 $ OMP_NUM_THREADS=1 ./a.out # use 1 thread  
2 $ OMP_NUM_THREADS=4 ./a.out # use 4 threads
```

- see 2.5.1 “Determining the Number of Threads for a parallel Region” for other ways to control the number of threads

# Contents

1 Overview

2 **parallel** pragma

3 Work sharing constructs

- loops (`for`)
- scheduling
- task parallelism (`task` and `taskwait`)

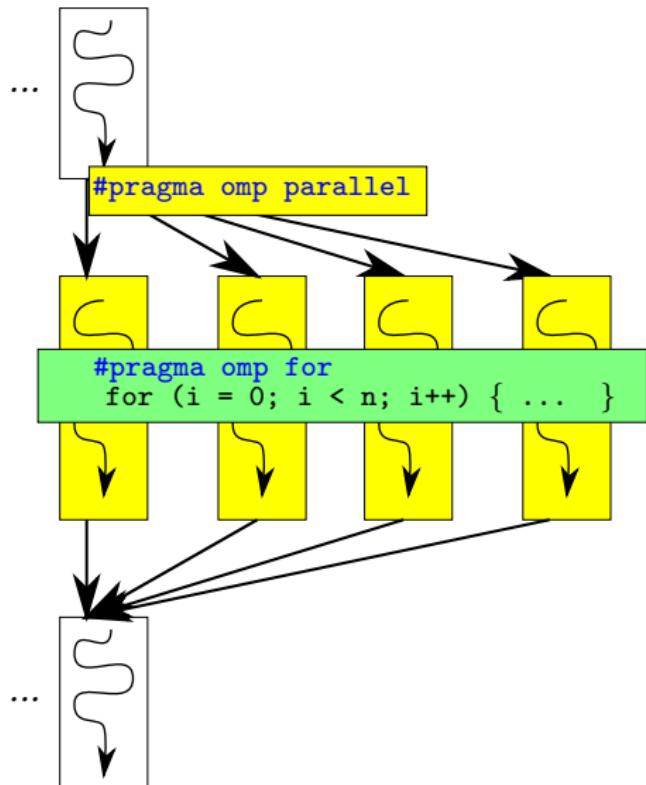
4 Data sharing clauses

5 SIMD constructs

# Two pragmas you must know first

- `#pragma omp parallel` to launch a team of threads (2.5)
- then `#pragma omp for` to distribute iterations to threads (2.7.1)

Note: all OpenMP pragmas have the common format: `#pragma omp ...`



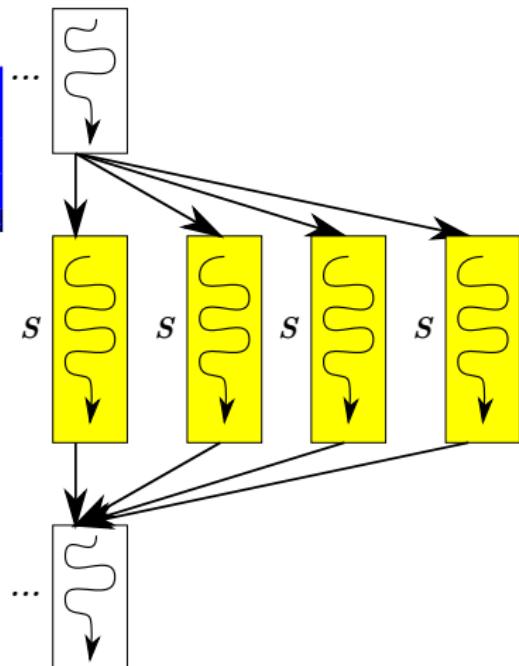
# #pragma parallel

- basic syntax:

```
1 ...  
2 #pragma omp parallel  
3   S  
4 ...
```

- basic semantics:

- create a team of `OMP_NUM_THREADS` threads
- the current thread becomes the *master* of the team
- *S will be executed by each member of the team*
- the master thread waits for all to finish *S* and continue



# parallel pragma example

```
1 #include <stdio.h>
2 int main() {
3     printf("hello\n");
4 #pragma omp parallel
5     printf("world\n");
6     return 0;
7 }
```

```
1 $ OMP_NUM_THREADS=1 ./a.out
2 hello
3 world
4 $ OMP_NUM_THREADS=4 ./a.out
5 hello
6 world
7 world
8 world
9 world
```

## Remarks : what does `parallel` do?

- you may assume an OpenMP thread  $\approx$  OS-supported thread (e.g., Pthread)
- that is, if you write this program

```
1 int main() {  
2     #pragma omp parallel  
3         worker();  
4 }
```

and run it as follows,

```
1 $ OMP_NUM_THREADS=50 ./a.out
```

you will get 50 OS-level threads, each doing `worker()`

# How to distribute work among threads?

- `#pragma omp parallel` creates threads, *all executing the same statement*
- it's not a means to parallelize work, *per se*, but just a means to create a number of similar threads (**SPMD**)
- so how to distribute (or partition) work among them?
  - ➊ do it yourself
  - ➋ use *work sharing* constructs

# Do it yourself: functions to get the number/id of threads

- `omp_get_num_threads()` (3.2.2) : the number of threads *in the current team*
- `omp_get_thread_num()` (3.2.4) : the current thread's id (0, 1, ...) in the team
- they are primitives with which you may partition work yourself by whichever ways you prefer
- e.g.,

```
1 #pragma omp parallel
2 {
3     int t = omp_get_thread_num();
4     int nt = omp_get_num_threads();
5     /* divide n iterations evenly among nt threads */
6     for (i = t * n / nt; i < (t + 1) * n / nt; i++) {
7         ...
8     }
9 }
```

# Contents

1 Overview

2 parallel pragma

3 Work sharing constructs

- loops (for)
- scheduling
- task parallelism (task and taskwait)

4 Data sharing clauses

5 SIMD constructs

# Contents

1 Overview

2 parallel pragma

3 Work sharing constructs

- loops (**for**)
- scheduling
- task parallelism (task and taskwait)

4 Data sharing clauses

5 SIMD constructs

# Work sharing constructs

- in theory, `parallel` construct is all you need to do things in parallel
- but it's too inconvenient
- OpenMP defines ways to `partition` work among threads (*work sharing constructs*)
  - for
  - task
  - section

# #pragma omp for (work-sharing for)

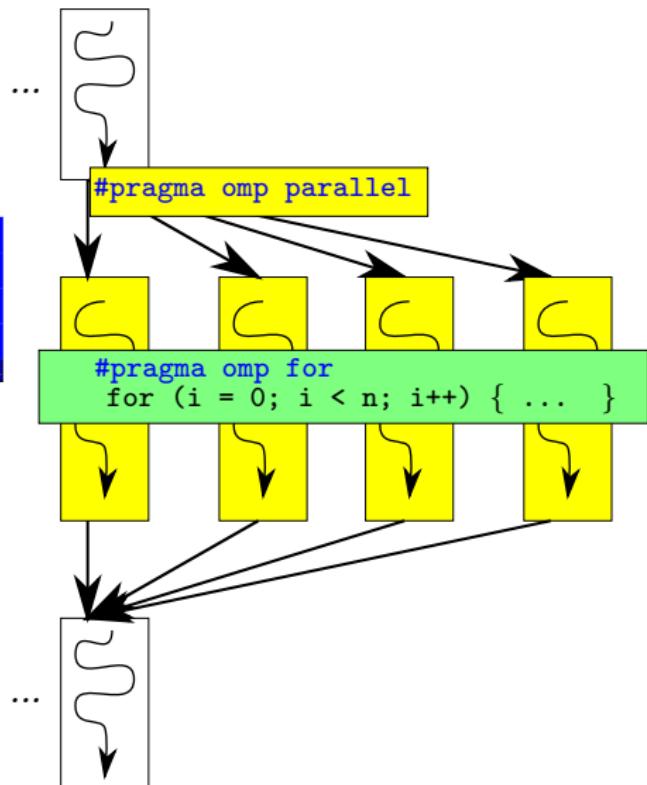
- basic syntax:

```
1 #pragma omp for
2 for(i=...; i...; i+=...){
3     S
4 }
```

- basic semantics:

the threads in the team  
divide the iterations among  
them

- but how? ⇒ scheduling



## #pragma omp for restrictions

- not arbitrary for statement is allowed after a for pragma
- strong syntactic restrictions apply, so that *the iteration counts can easily be identified at the beginning* of the loop
- roughly, it must be of the form:

```
1 #pragma omp for
2 for(i = init; i < limit; i += incr)
3     S
```

except < and += may be other similar operators

- *init*, *limit*, and *incr* must be loop invariant

# Contents

1 Overview

2 parallel pragma

3 Work sharing constructs

- loops (for)
- scheduling
- task parallelism (task and taskwait)

4 Data sharing clauses

5 SIMD constructs

## Scheduling (2.7.1)

- `schedule` clause in work-sharing for loop determines how iterations are divided among threads
- There are three alternatives (`static`, `dynamic`, and `guided`)

# static, dynamic, and guided

- `schedule(static,[chunk]):`  
predictable round-robin
- `schedule(dynamic,[chunk]):`  
each thread repeats fetching  
*chunk* iterations
- `schedule(guided,[chunk]):`  
threads grab many iterations  
in early stages; gradually  
reduce iterations to fetch at a  
time
- *chunk* specifies the minimum  
granularity (iteration counts)

```
#pragma omp for schedule(static)
    0   1   2   3
# pragma omp for schedule(static,3)
    0   1   2   3   0   1   2   3
# pragma omp for schedule(dynamic)
    0   1   2   3   0   1   2   3
# pragma omp for schedule(dynamic,2)
    0   1   2   3   0   1   2   3
# pragma omp for schedule(guided)
    0   1   2   3   0   1   2   3
# pragma omp for schedule(guided,2)
    0   1   2   3   0   1   2   3
```

# Other scheduling options and notes

- `schedule(runtime)` determines the schedule by `OMP_SCHEDULE` environment variable. e.g.,

```
1 $ OMP_SCHEDULE=dynamic,2 ./a.out
```

- `schedule(auto)` or `no schedule` clause choose an implementation dependent default

# Parallelizing loop nests by `collapse`

- `collapse(l)` can be used to partition nested loops. e.g.,

```
1 #pragma omp for collapse(2)
2 for (i = 0; i < n; i++)
3     for (j = 0; j < n; j++)
4         S
```

will partition  $n^2$  iterations of the doubly-nested loop

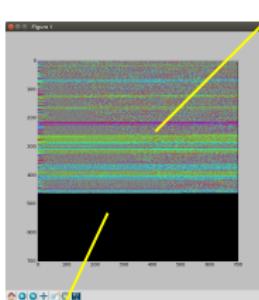
- `schedule` clause applies to nested loops as if the nested loop is an equivalent flat loop
- restriction: the loop must be “*perfectly nested*” (the iteration space must be a rectangular and no intervening statement between different levels of the nest)

# Visualizing schedulers

- seeing is believing. let's visualize how loops are distributed among threads
- write a simple doubly nested loop and run it under various scheduling options

```
1 #pragma omp for collapse(2) schedule(runtime)
2 for (i = 0; i < 1000; i++)
3     for (j = 0; j < 1000; j++)
4         unit_work(i, j);
```

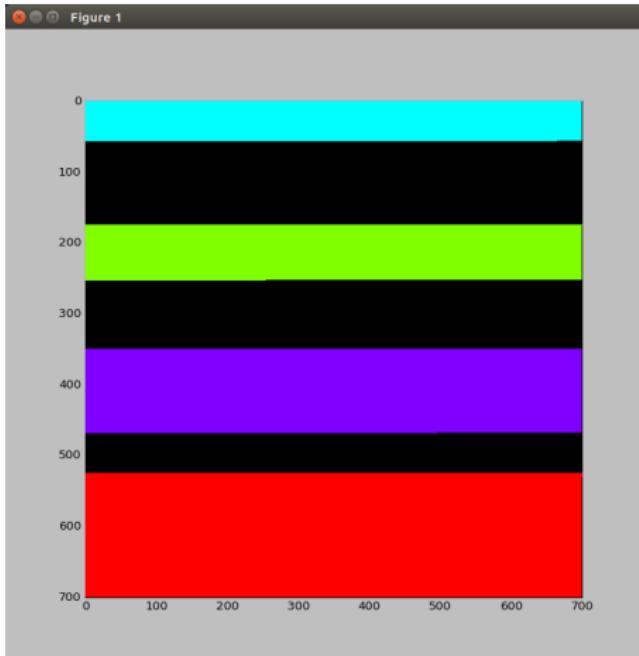
- load per point is systematically skewed:
  - $\approx 0$  in the lower triangle
  - drawn from  $[100, 10000]$  (clocks) in the upper triangle



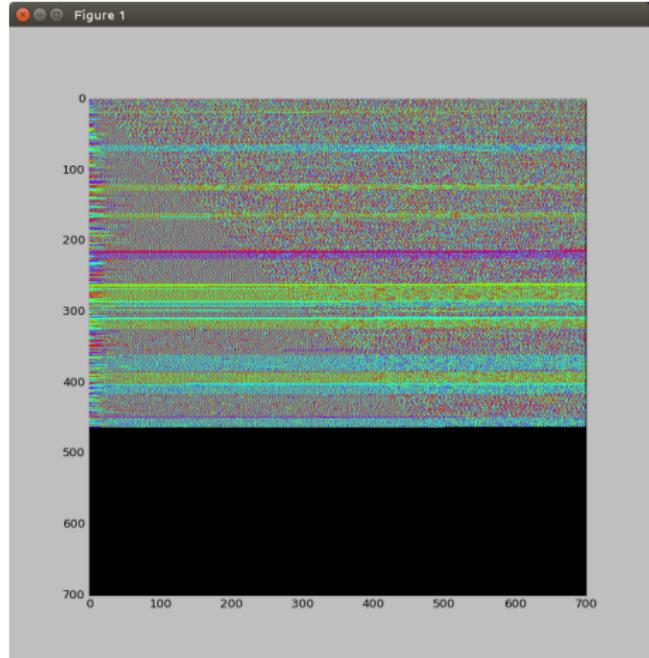
load  $\sim [100, 10000]$  clocks

load  $\approx 0$

# Visualizing schedulers



static



dynamic

# Contents

1 Overview

2 parallel pragma

3 Work sharing constructs

- loops (for)
- scheduling
- task parallelism (**task** and **taskwait**)

4 Data sharing clauses

5 SIMD constructs

# Task parallelism in OpenMP

- OpenMP's initial focus was simple parallel loops
- since 3.0, it supports task parallelism
- but why it's necessary?
- aren't `parallel` and `for` all we need?

# Limitation of parallel for

- what if you have a parallel loop inside another

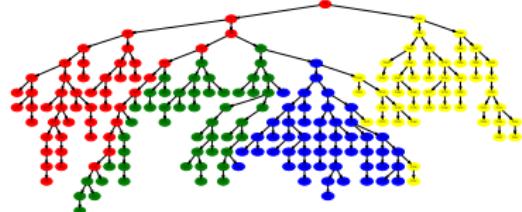
```
1 for ( ... ) {  
2     ...  
3     for ( ... ) ...  
4 }
```

- perhaps inside a separate function?

```
1 main() {  
2     for ( ... ) {  
3         ...  
4         g();  
5     }  
6 }  
7 g() {  
8     for ( ... ) ...  
9 }
```

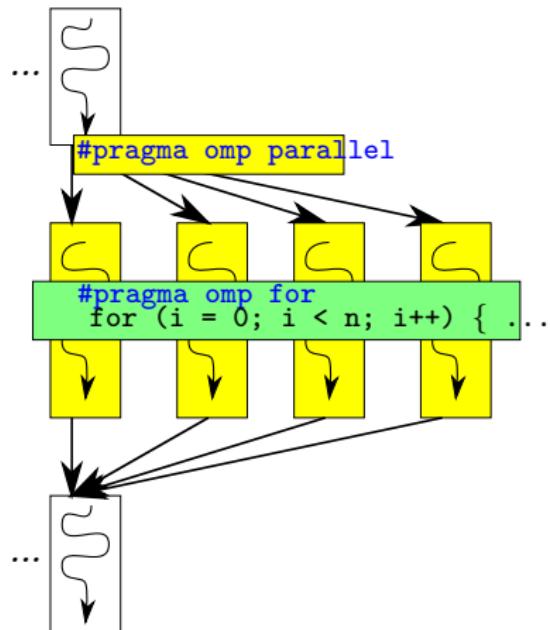
- what for parallel recursions?

```
1 qs() {  
2     if ( ... ) { ... }  
3     else {  
4         qs();  
5         qs();  
6     }  
7 }
```



# parallel for can't handle nested parallelism

- OpenMP generally ignores nested `parallel` pragma when enough threads have been created by the outer `parallel` pragma, for good reasons
- the fundamental limitation is its simplistic work-sharing mechanism
- tasks address these issues, by allowing tasks to be created at arbitrary points of execution (and a mechanism to distribute them across cores)



# Task parallelism in OpenMP

- syntax:

- create a task  $\approx$  TBB's `task_group::run`

```
1 #pragma omp task  
2   S
```

- wait for tasks  $\approx$  TBB's `task_group::wait`

```
1 #pragma omp taskwait
```

# OpenMP task parallelism template

- don't forget to create a **parallel** region
- don't also forget to enter a **master** region, which says only the master executes the following statement and others "stand-by"

```
1 int main() {  
2 #pragma omp parallel  
3 #pragma omp master  
4 // or #pragma omp single  
5     ms(a, a + n, t, 0);  
6 }
```

- and create tasks in the master region

```
1 void ms(a, a_end, t, dest) {  
2     if (n == 1) {  
3         ...  
4     } else {  
5         ...  
6 #pragma omp task  
7         ms(a, c,      t,      1 - dest);  
8 #pragma omp task  
9         ms(c, a_end, t + nh, 1 - dest);  
10 #pragma omp taskwait  
11     ...  
12 }
```

# Visualizing task parallel schedulers

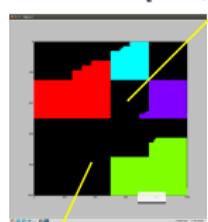
- the workload is exactly the same as before

```
1 #pragma omp for collapse(2) schedule(runtime)
2 for (i = 0; i < 1000; i++)
3     for (j = 0; j < 1000; j++)
4         unit_work(i, j);
```

- but we rewrite it into recursions

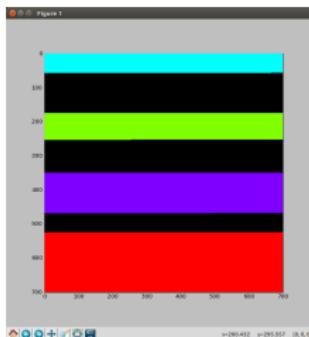
```
1 void work_rec(rectangle b) {
2     if (small(b)) {
3         ...
4     } else {
5         rectangle c[2][2];
6         split(b, c); // split b into 2x2 sub-rectangles
7         for (i = 0; i < 2; i++) {
8             for (j = 0; j < 2; j++) {
9 #pragma omp task
10             work_rec(b[i][j]);
11         }
12     }
13 #pragma omp taskwait
14 }
```

load  $\sim [100, 10000]$  clock

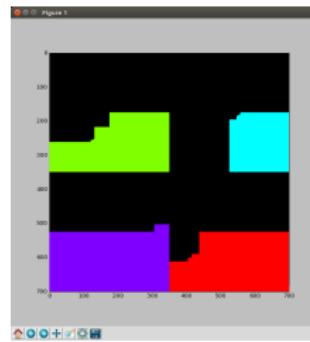


load  $\approx 0$

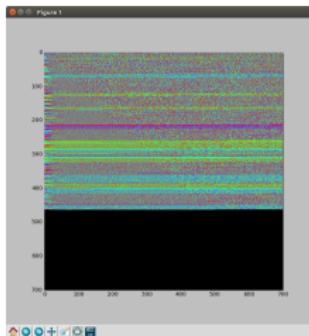
# Visualizing schedulers



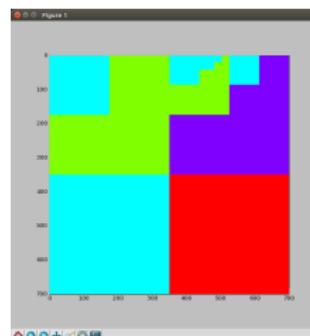
static



2D recursive (midway)



dynamic



2D recursive (end)

# A note on current GCC task implementation

- the overhead seems high and the scalability seems low, so it's not very useful now
- TBB, Cilk, and MassiveThreads-backed TBB are all much better
- Intel implementation of OpenMP tasks is good too
- we'll come back to the topic of efficient implementation of task parallelism later

# Pros/cons of schedulers

- **static:**

- partitioning iterations is simple and does not require communication
- may cause load imbalance (leave some threads idle, even when other threads have many work to do)
- mapping between work  $\leftrightarrow$  thread is deterministic and predictable (why it's important?)

- **dynamic:**

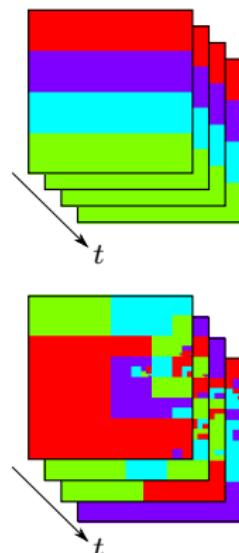
- no worry about load imbalance, if chunks are sufficiently small
- partitioning iterations needs communication (no two threads execute the same iteration) and may become a bottleneck
- mapping between iterations and threads is non-deterministic
- OpenMP's dynamic scheduler is inflexible in partitioning loop nests

# Pros/cons of schedulers

- **recursive (divide and conquer + tasks):**
  - no worry about load imbalance, as in dynamic
  - distributing tasks needs communication, but efficient implementation techniques are known
  - mapping between work and thread is non-deterministic, as in dynamic
  - you can flexibly partition loop nests in various ways (e.g., keep the space to square-like)
  - need boilerplate coding efforts (easily circumvented by additional libraries; e.g., TBB's `blocked_range2d` and `parallel_for`)

# Deterministic and predictable schedulers

- programs often execute the same for loops many times, with the same trip counts, and with the same iteration touching a similar region
- such *iterative* applications may benefit from reusing data brought into cache in the previous execution of the same loop
- a deterministic scheduler achieves this benefit



# Contents

1 Overview

2 parallel pragma

3 Work sharing constructs

- loops (`for`)
- scheduling
- task parallelism (`task` and `taskwait`)

4 Data sharing clauses

5 SIMD constructs

# Data sharing

- `parallel`, `for`, `task` pragma accept clauses specifying which variables should be shared among threads or between the parent/child tasks
- 2.14 “Data Environments”
  - `private`
  - `firstprivate`
  - `shared`
  - `reduction` (only for `parallel` and `for`)
  - `copyin`

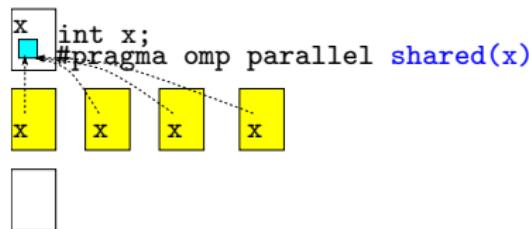
# Data sharing example

```
1 int main() {
2     int S; /* shared */
3     int P; /* made private below */
4 #pragma omp parallel private(P) shared(S)
5     {
6         int L; /* automatically private */
7         printf("S at %p, P at %p, L at %p\n",
8                &S, &P, &L);
9     }
10    return 0;
11 }
```

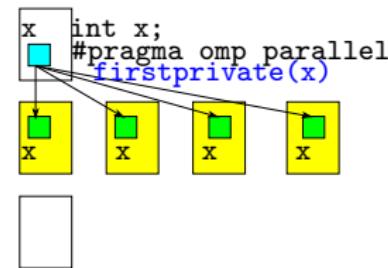
```
1 $ OMP_NUM_THREADS=2 ./a.out
2 S at 0x..777f494, P at 0x..80d0e28, L at 0x..80d0e2c
3 S at 0x..777f494, P at 0x..777f468, L at 0x..777f46c
```

# Data sharing behavior

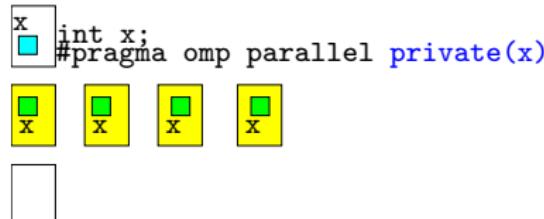
shared



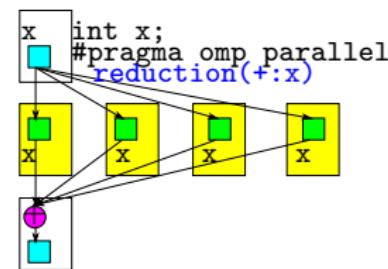
firstprivate



private



reduction



# Reduction

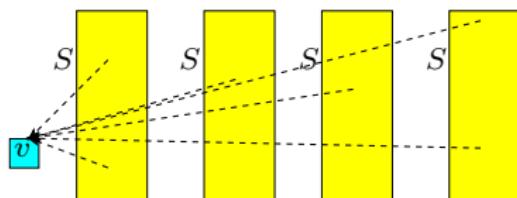
- in general, “reduction” refers to an operation to combine many values into a single value. e.g.,

- $v = v_1 + \dots + v_n$
- $v = \max(v_1, \dots, v_n)$
- ...

- simply sharing the variable ( $v$ ) does not work (race condition)
- even if you make updates atomic, it will be slow (by now you should know how slow it will be)

```
1  v = 0.0;
2  for (i = 0; i < n; i++) {
3      v += f(a + i * dt) * dt;
4 }
```

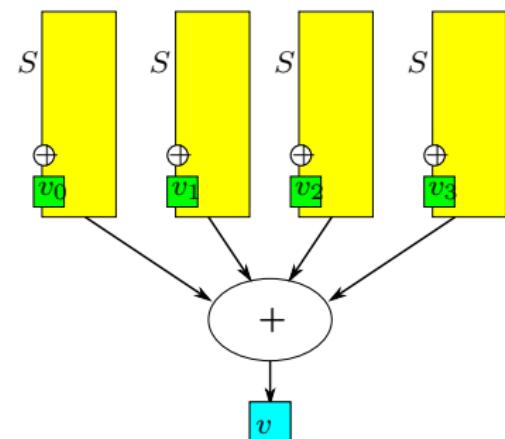
```
1  v = 0.0;
2  #pragma omp parallel shared(v)
3  #pragma omp for
4  for (i = 0; i < n; i++) {
5      #pragma omp atomic
6      v += f(a + i * dt) * dt;
7 }
```



# Reduction clause in OpenMP

- a more efficient strategy is to
  - let each thread reduce on its private variable, and
  - when threads finish, combine their partial results into one
- reduction clause in OpenMP does just that

```
1  v = 0.0;  
2  #pragma omp parallel shared(v)  
3  #pragma omp for reduce(+:v)  
4  for (i = 0; i < n; i++) {  
5      v += f(a + i * dt) * dt;  
6  }
```



# Simple reduction and user-defined reduction (2.7.1)

- reduction syntax:

```
1 #pragma omp parallel reduction(op:var,var,...)  
2     S
```

- builtin reductions
  - *op* is one of +, \*, -, &, ^, |, &&, and ||
  - (Since 3.1) min or max
- (Since 4.0) a user-defined reduction name

- user-defined reduction syntax:

```
1 #pragma omp declare reduction (name : type : combine statement)
```

# User-defined reduction example

```
1  typedef struct {
2      int x; int y;
3  } point;
4  point add_point(point p, point q) {
5      point r = { p.x + q.x, p.y + q.y };
6      return r;
7  }
8  // declare reduction "ap" to add two points
9  #pragma omp declare reduction(ap: point:  omp_out=add_point(omp_out,
10                                omp_in))
11
11 int main(int argc, char ** argv) {
12     int n = atoi(argv[1]);
13     point p = { 0.0, 0.0 };
14     int i;
15     #pragma omp parallel for reduction(ap : p)
16     for (i = 0; i < n; i++) {
17         point q = { i, i };
18         p = add_point(p, q);
19     }
20     printf("%d %d\n", p.x, p.y);
21     return 0;
22 }
```

# Contents

1 Overview

2 parallel pragma

3 Work sharing constructs

- loops (`for`)
- scheduling
- task parallelism (`task` and `taskwait`)

4 Data sharing clauses

5 SIMD constructs

# SIMD constructs

- **simd** pragma
  - allows an explicit vectorization of for loops
  - syntax restrictions similar to `omp for` pragma apply
- **declare simd** pragma
  - instructs the compiler to generate vectorized versions of a function
  - with it, loops with function calls can be vectorized

# simd pragma

- basic syntax (similar to `omp for`):

```
1 #pragma omp simd clauses
2 for (i = ...; i < ...; i += ...)
3     S
```

- clauses
  - `aligned(var,var,...:align)`
  - `uniform(var,var,...)` says variables are loop invariant
  - `linear(var,var,...:stride)` says variables have the specified stride between consecutive iterations

# declare simd pragma

- basic syntax (similar to `omp for`):

```
1 #pragma omp declare simd clauses
2 function definition
```

- clauses
  - those for `simd` pragma
  - `notinbranch`
  - `inbranch`

# SIMD pragmas, rationales

- most automatic vectorizers give up vectorization in many cases
  - ➊ conditionals (lanes may branch differently)
  - ➋ inner loops (lanes may have different trip counts)
  - ➌ function calls (function bodies are not vectorized)
  - ➍ iterations may not be independent
- `simd` and `declare simd` directives should eliminate obstacles 3 and 4 and significantly enhance vectorization opportunities

# A note on current GCC OpenMP SIMD implementation

- as of now (version 4.9), GCC `simd` and `declare simd` ≈ existing auto vectorizer – dependence analysis
- `declare simd` functions are first converted into a loop over all vector elements and then passed to the loop vectorizer

```
1 #pragma omp declare simd
2 float f(float x, float y) {
3     return x + y;
4 }
```

→

```
1 float8 f(float8 vx, float8 vy) {
2     float8 r;
3     for (i = 0; i < 8; i++) {
4         float x = vx[i], y = vy[i]
5         r[i] = x + y;
6     }
7     return r;
8 }
```

- the range of vectorizable loops in current GCC (4.9) implementation seems very limited
  - innermost loop with no conditionals
  - doubly nested loop with a very simple inner loop

# Chapter 2

## Abstract Machine Models & Multi-threading

---

**Thoai Nam**

Faculty of Computer Science and Engineering  
HCMC University of Technology



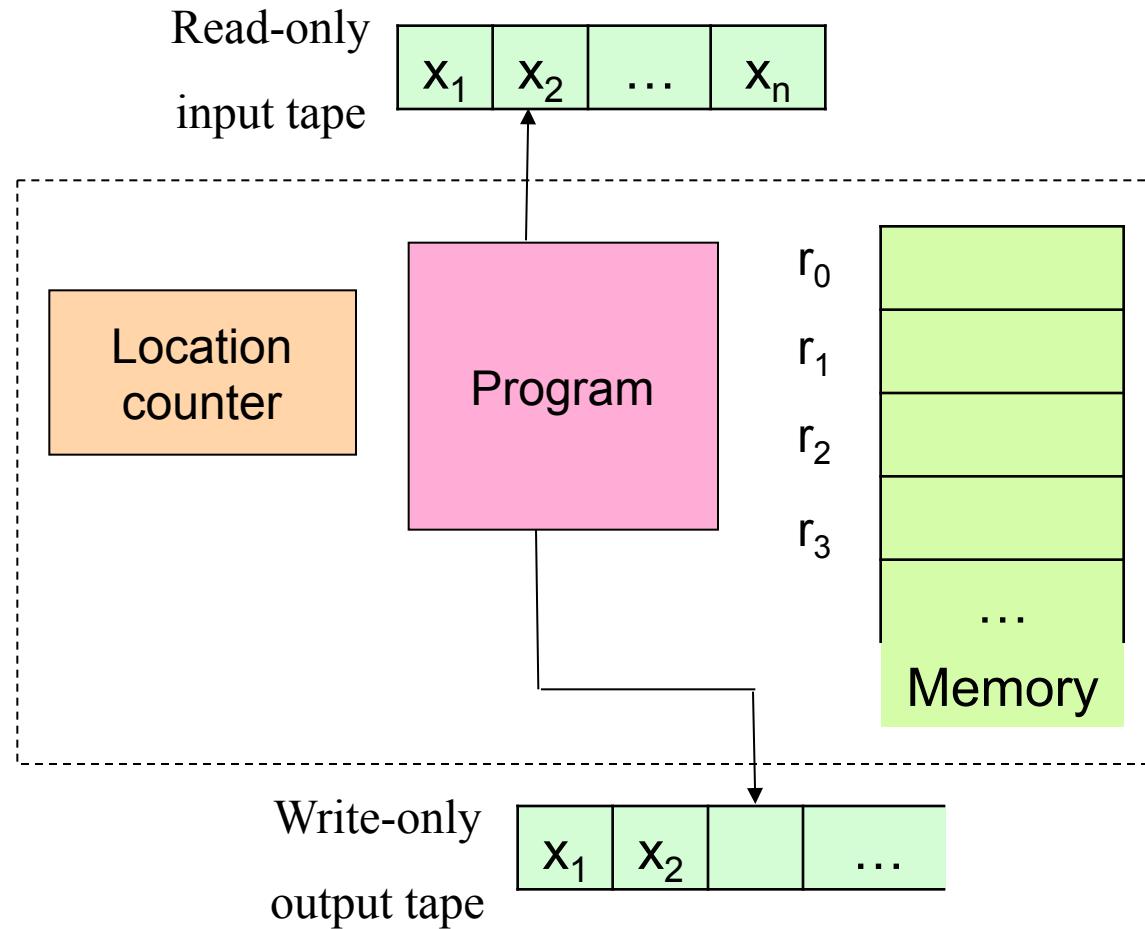
# Abstract Machine Models

---

- An abstract machine model is mainly used in the design and analysis of parallel algorithms without worry about the details of physics machines.
- Three abstract machine models:
  - PRAM
  - BSP
  - Phase Parallel

# RAM (1)

## □ RAM (Random Access Machine)





# RAM (2)

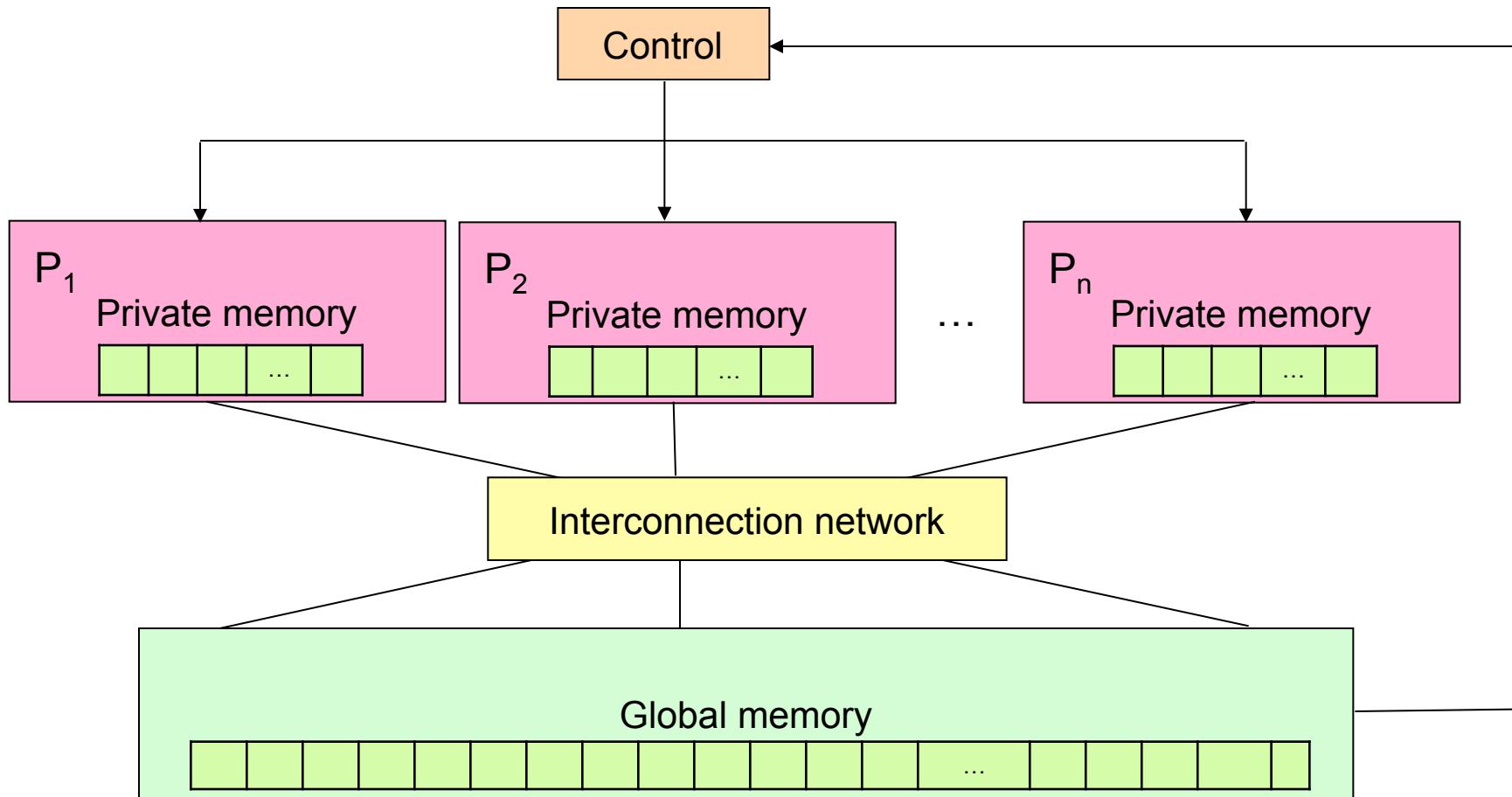
---

## RAM model of serial computers

- Memory is a sequence of words, each capable of containing an integer
- Each memory access takes one unit of time
- Basic operations (add, multiply, compare) take one unit time
- Instructions are not modifiable
- Read-only input tape, write-only output tape

# PRAM (1)

Parallel Random Access Machine (Introduced by Fortune and Wyllie, 1978)





# PRAM (2)

---

- A control unit
- An unbounded set of processors, each with its own private memory and an unique index
- Input stored in global memory or a single *active* processing element
- Step: (1) read a value from a single private/global memory location
  - (2) perform a RAM operation
  - (3) write into a single private/global memory location
- During a computation step: a processor may activate another processor
- All active, enable processors must execute ***the same instruction*** (albeit on different memory location)???
- Computation terminates when the last processor halts



# PRAM (3)

---

PRAM composed of:

- P processors, each with its own unmodifiable program
- A single shared memory composed of a sequence of words, each capable of containing an arbitrary integer
- a read-only input tape
- a write-only output tape

PRAM model is a synchronous, MIMD, shared address space parallel computer

- Processors share a common clock but may execute different instructions in each cycle



# PRAM(4)

---

## □ Definition:

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

Ex: a PRAM algorithm that has time complexity  $O(\log p)$  using  $p$  processors has cost  $O(p \log p)$



# Time Complexity Problem

---

- Time complexity of a PRAM algorithm is often expressed in the big- $O$  notation
- Machine size  $n$  is usually small in existing parallel computers
- Ex:
  - Three PRAM algorithms A, B and C have time complexities if  $7n$ ,  $(n \log n)/4$ ,  $n \log \log n$ .
  - Big- $O$  notation:  $A(O(n)) < C(O(n \log \log n)) < B(O(n \log n))$
  - Machines with no more than 1024 processors:  
 $\log n \leq \log 1024 = 10$  and  $\log \log n \leq \log \log 1024 < 4$   
and thus:  $B < C < A$



# Conflicts Resolution Schemes (1)

---

- PRAM execution can result in simultaneous access to the same location in shared memory.
  - Exclusive Read (ER)
    - » No two processors can simultaneously read the same memory location.
  - Exclusive Write (EW)
    - » No two processors can simultaneously write to the same memory location.
  - Concurrent Read (CR)
    - » Processors can simultaneously read the same memory location.
  - Concurrent Write (CW)
    - » Processors can simultaneously write to the same memory location, using some conflict resolution scheme.



# Conflicts Resolution Schemes (2)

---

## ❑ Common/Identical CRCW

- All processors writing to the same memory location must be writing the same value.
- The software must ensure that different values are not attempted to be written.

## ❑ Arbitrary CRCW

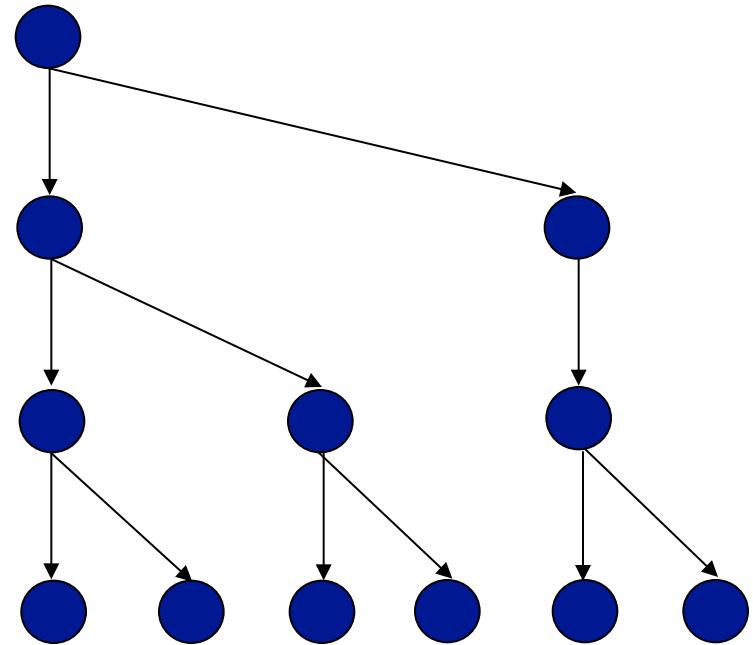
- Different values may be written to the same memory location, and an arbitrary one succeeds.

## ❑ Priority CRCW

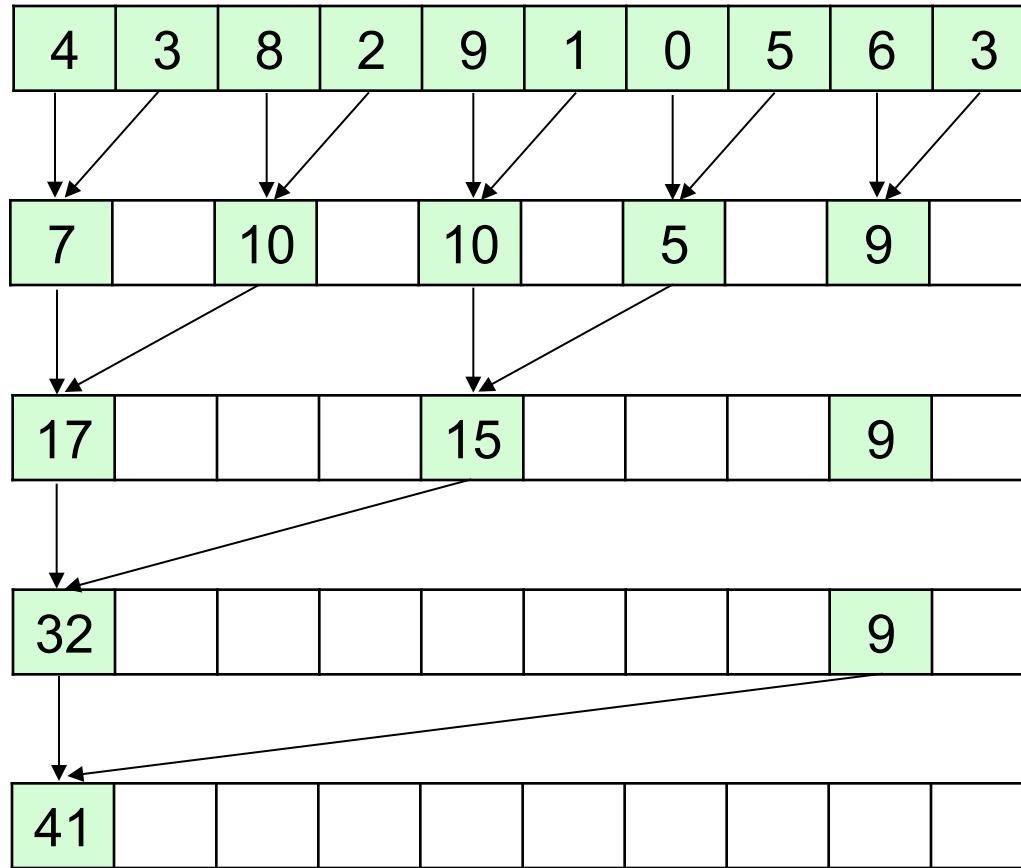
- An index is associated with the processors and when more than one processor write occurs, the lowest-numbered processor succeeds.
- The hardware must resolve any conflicts

# PRAM Algorithm

- Begin with a single active processor active
- Two phases:
  - A sufficient number of processors are activated
  - These activated processors perform the computation in parallel
- $\lceil \log p \rceil$  activation steps:  $p$  processors to become active
- The number of active processors can be double by executing a single instruction



# Parallel Reduction (1)





# Parallel Reduction (2)

---

(EREW PRAM Algorithm in Figure2-7, page 32, book [1])

Ex:     SUM(EREW)

Initial condition: List of  $n \geq 1$  elements stored in  $A[0..(n-1)]$

Final condition: Sum of elements stored in  $A[0]$

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

begin

    spawn ( $P_0, P_1, \dots, P_{\lfloor n/2 \rfloor - 1}$ )

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

        for  $j \leftarrow 0$  to  $\lceil \log n \rceil - 1$  do

            if  $i$  modulo  $2^j = 0$  and  $2i+2^j < n$  then

$A[2i] \leftarrow A[2i] + A[2i+2^j]$

            endif

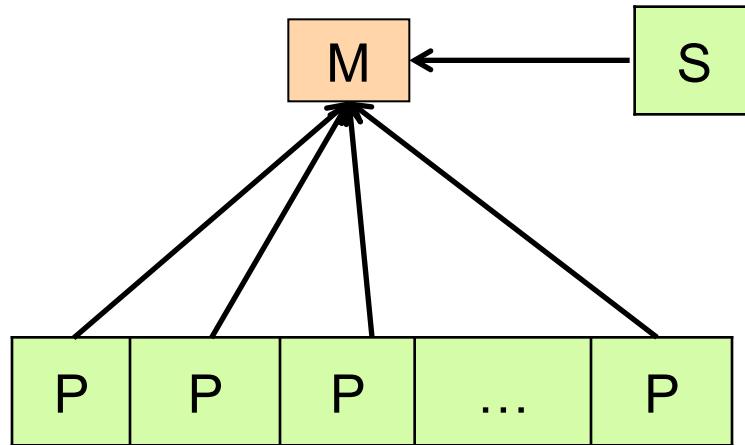
        endfor

    endfor

end

# Broadcasting on a PRAM

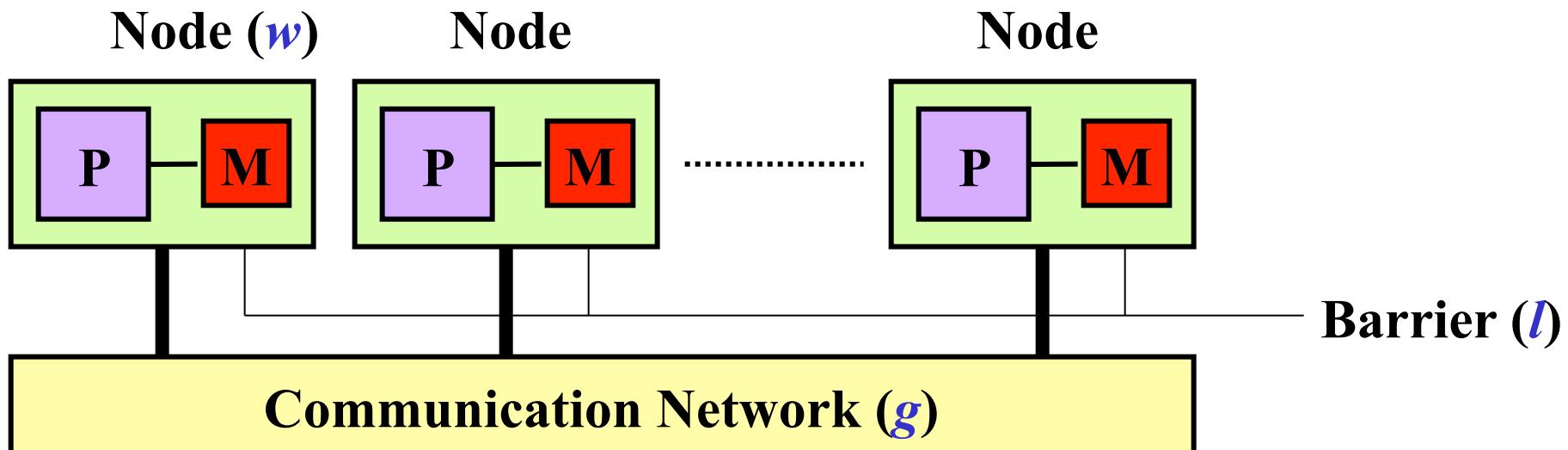
- “Broadcast” can be done on CREW PRAM in  $O(1)$  steps:
  - Broadcaster sends value to shared memory
  - Processors read from shared memory
- Requires  $\log P$  steps on EREW PRAM



# BSP – Bulk Synchronous Parallel

## □ BSP Model

- Proposed by Leslie Valiant of Harvard University
- Developed by W.F.McColl of Oxford University





# BSP Model

---

- A set of n nodes (processor/memory pairs)
- Communication Network
  - Point-to-point, message passing (or shared variable)
- Barrier synchronizing facility
  - All or subset
- Distributed memory architecture

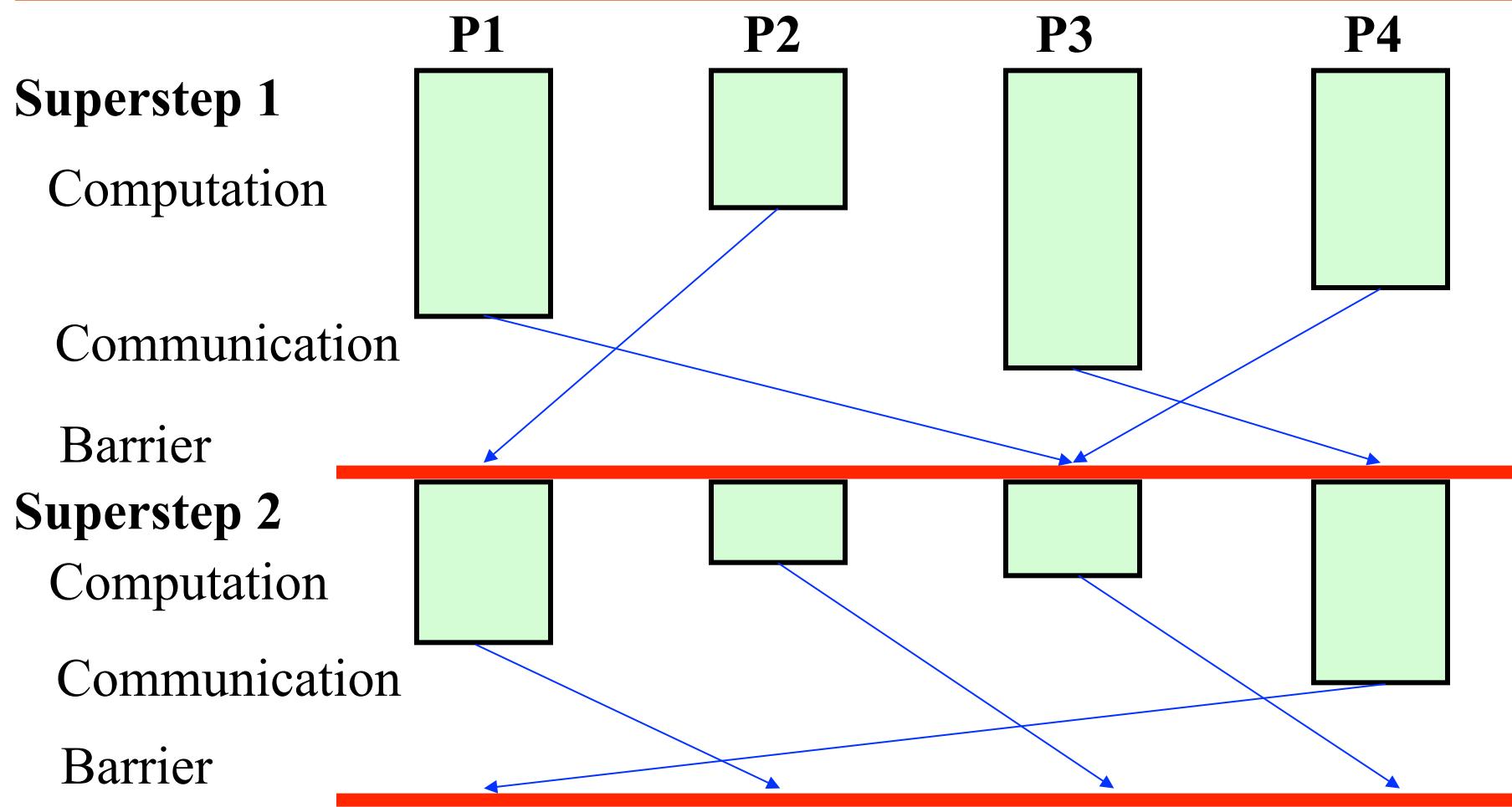


# BSP Programs

---

- A BSP program:
  - $n$  processes, each residing on a node
  - Executing a strict sequence of *supersteps*
  - In each superstep, a process executes:
    - » Computation operations:  $w$  cycles
    - » Communication:  $gh$  cycles
    - » Barrier synchronization:  $l$  cycles

# A Figure of BSP Programs





# Three Parameters

- The basic time unit is a cycle (or time step)
- **w** parameter
  - Maximum computation time within each superstep
  - Computation operation takes at most **w** cycles.
- **g** parameter
  - Number of cycles for communication of unit message when all processors are involved in communication - network bandwidth
  - (total number of local operations performed by all processors in one second) / (total number of words delivered by the communication network in one second)
  - **h** relation coefficient
  - Communication operation takes **gh** cycles.
- **I** parameter
  - Barrier synchronization takes **I** cycles.



# Time Complexity of BSP Algorithms

---

- Execution time of a superstep:
  - Sequence of the computation, the communication, and the synchronization operations:  $w + gh + l$
  - Overlapping the computation, the communication, and the synchronization operations:  $\max\{w, gh, l\}$

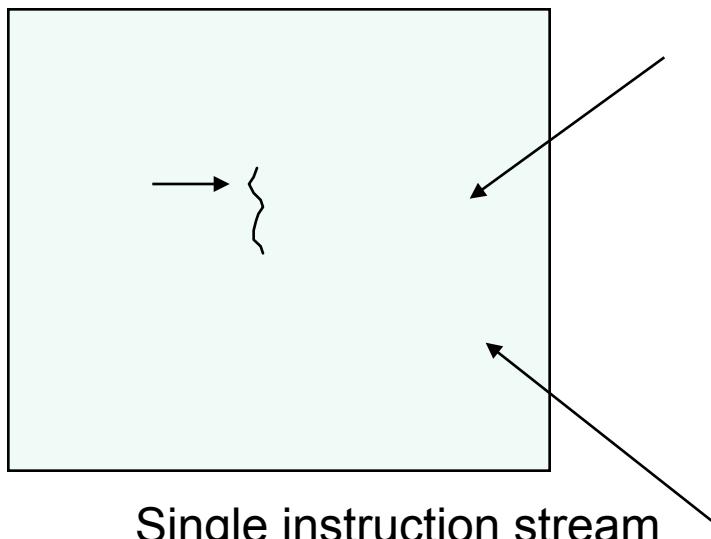


# Phase Parallel

- Proposed by Kai Hwang & Zhiwei Xu
- Similar to the BSP:
  - A parallel program: sequence of phases
  - Next phase cannot begin until all operations in the current phase have finished
  - Three types of phases:
    - » **Parallelism phase**: the overhead work involved in process management, such as process creation and grouping for parallel processing
    - » **Computation phase**: local computation (data are available)
    - » **Interaction phase**: communication, synchronization or aggregation (e.g., reduction and scan)
- Different computation phases may execute different workloads at different speed.

# Process: single & multithreaded

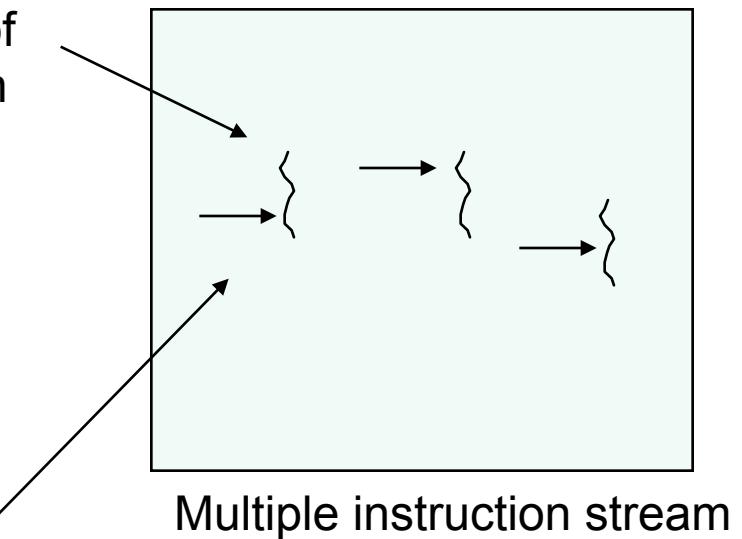
Single-threaded Process



Threads of  
Execution

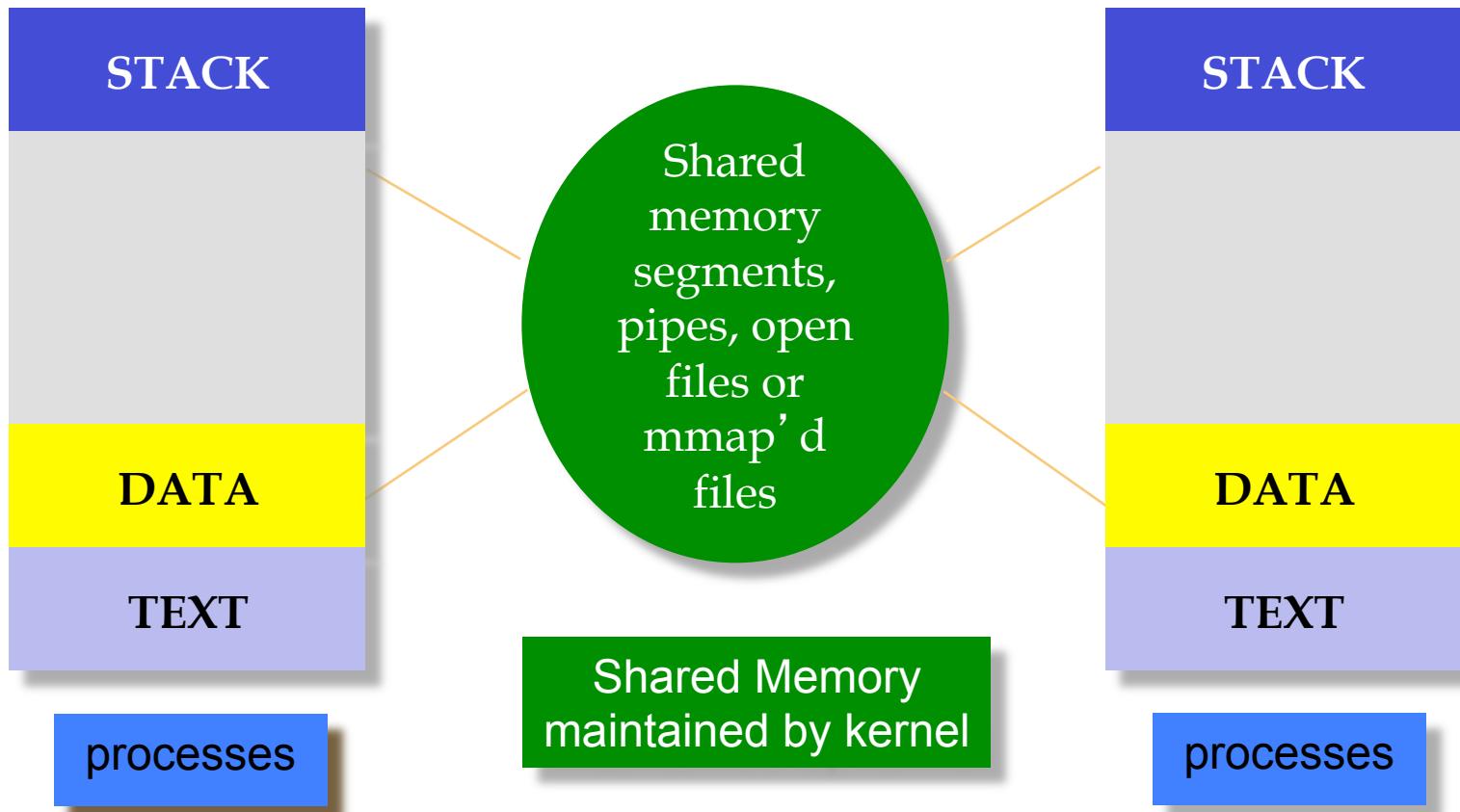
Common  
Address Space

Multiplethreaded Process



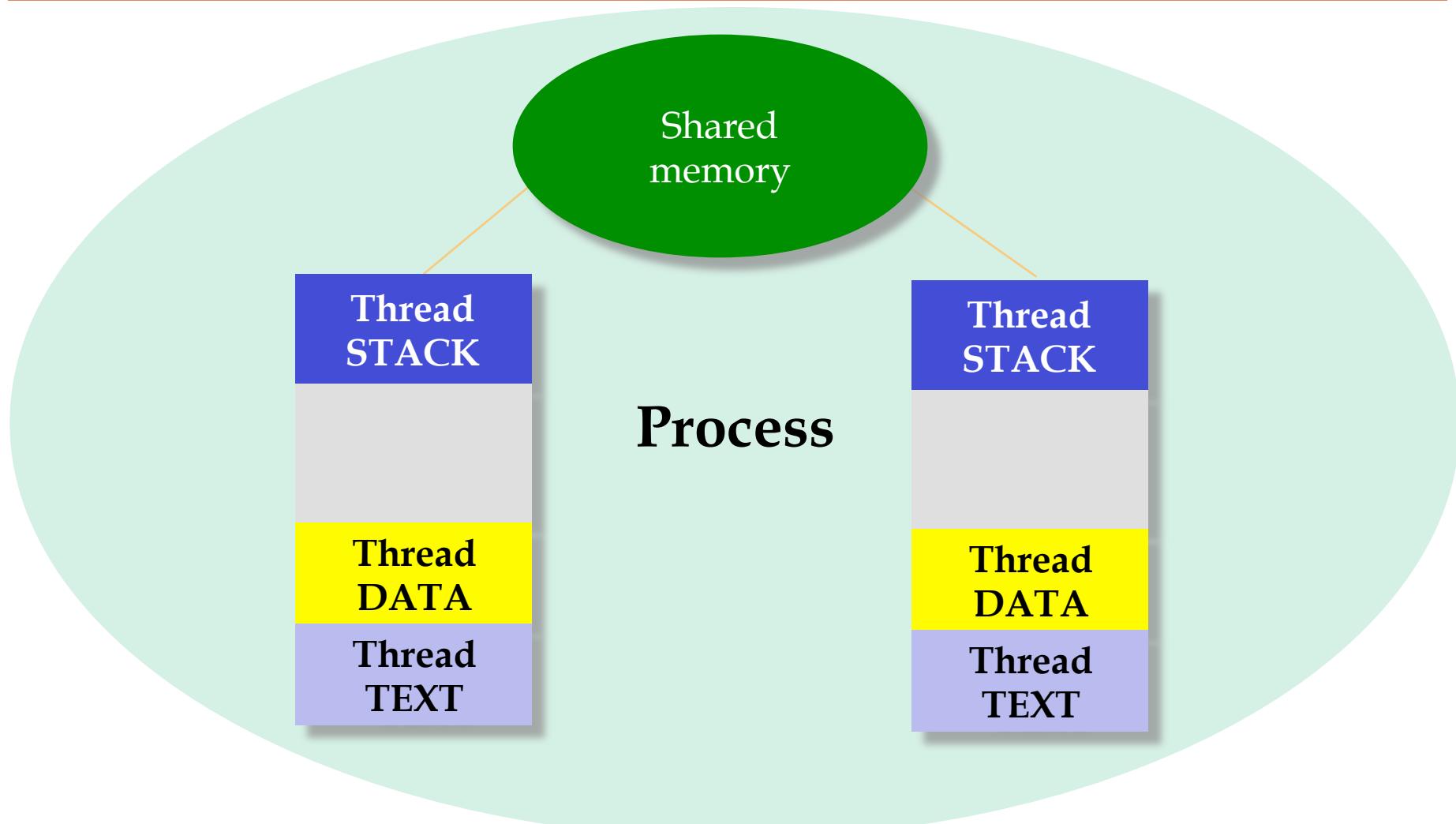
Multiple instruction stream

# Process Model



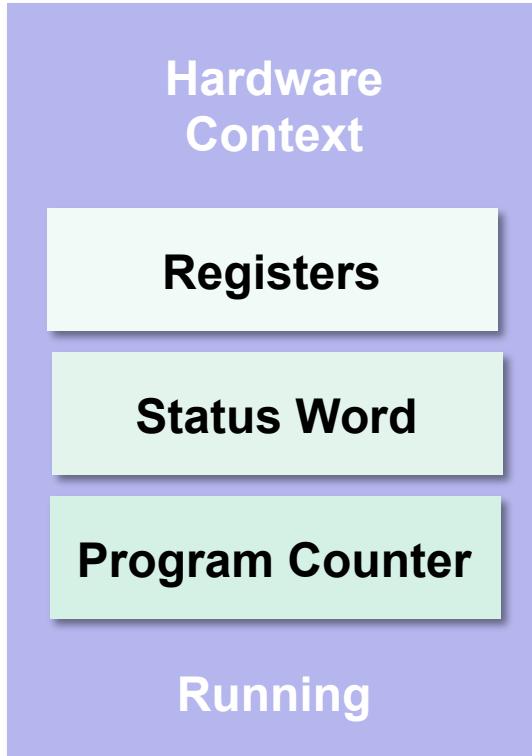


# Threaded Process Model



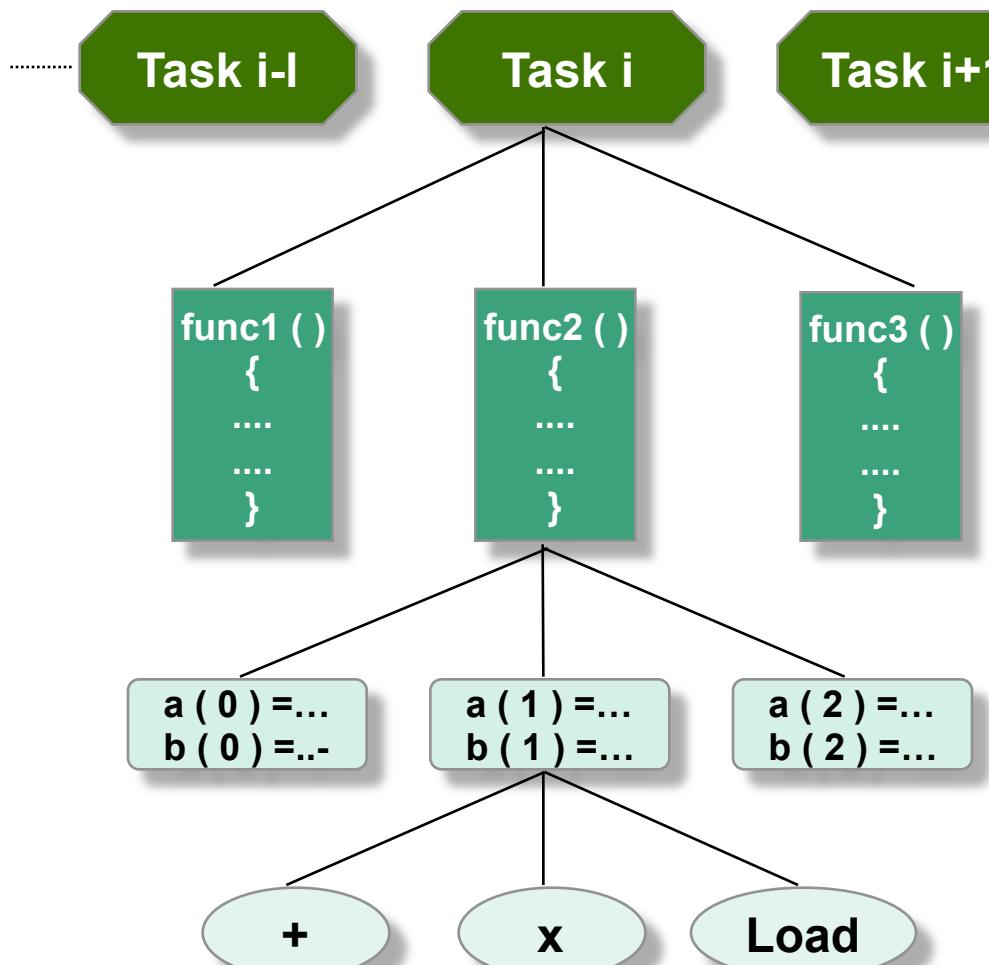


# What are Threads



- ❑ Thread is a piece of code that can execute in concurrence with other **threads**
- ❑ It is a schedule entity on a processor
  - Local state
  - Global/shared state
  - PC
  - Hard/Software context

# Levels of Parallelism



**Code-Granularity**  
**Code Item**  
**Large grain**  
**(task level)**  
**Program**

**Medium grain**  
**(control level)**  
**Function (thread)**

**Fine grain**  
**(data level)**  
**Loop**

**Very fine grain**  
**(multiple issue)**  
**With hardware**



# Thread Example

```
void *func ( )
{
    /* define local data */
    - - - - - - - - - -
    - - - - - - - - - - /* function code */
    - - - - - - - - - -
    thr_exit(exit_value);
}

main ( )
{
    thread_t tid;
    int exit_value;
    - - - - - - - - - -
    thread_create (0, 0, func (), NULL, &tid);
    - - - - - - - - - -
    thread_join (tid, 0, &exit_value);
    - - - - - - - - - -
}
```



# Ref

---

- BSP: [http://www.computingreviews.com/hottopic/hottopic\\_essay.cfm?htname=BSP](http://www.computingreviews.com/hottopic/hottopic_essay.cfm?htname=BSP)



# Bubble Sort

---

```
procedure BubbleSort( A : list of sortable items )
```

```
    n = length(A)
```

```
    repeat
```

```
        newn = 0
```

```
        for i = 1 to n-1 inclusive do
```

```
            if A[i-1] > A[i] then
```

```
                swap(A[i-1], A[i])
```

```
                newn = i
```

```
            end if
```

```
        end for
```

```
        n = newn
```

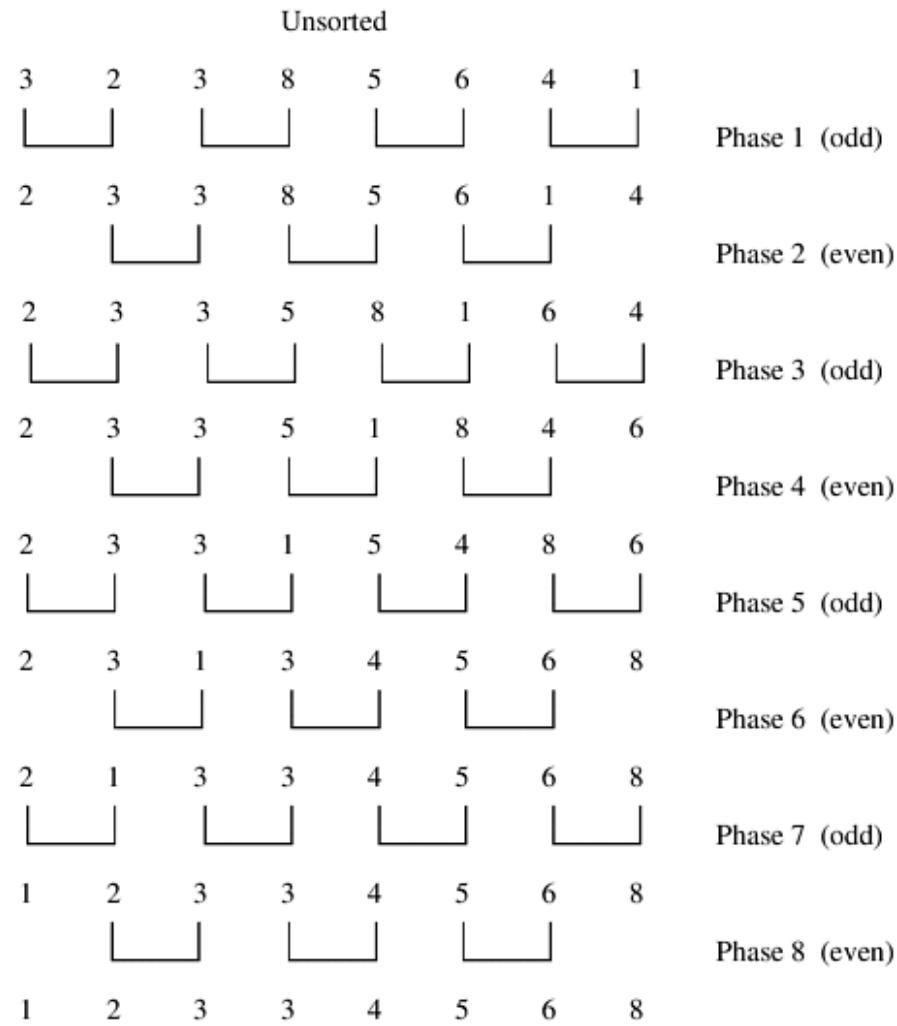
```
    until n = 0
```

```
end procedure
```



# Parallel Bubble Sort

[http://srmcse.weebly.com/uploads/8/9/0/9/8909020/introduction\\_to\\_parallel\\_computing\\_second\\_edition-ananth\\_grama..pdf](http://srmcse.weebly.com/uploads/8/9/0/9/8909020/introduction_to_parallel_computing_second_edition-ananth_grama..pdf)  
<http://parallelcomp.uw.hu/ch09lev1sec3.html>





# Quick Sort

---

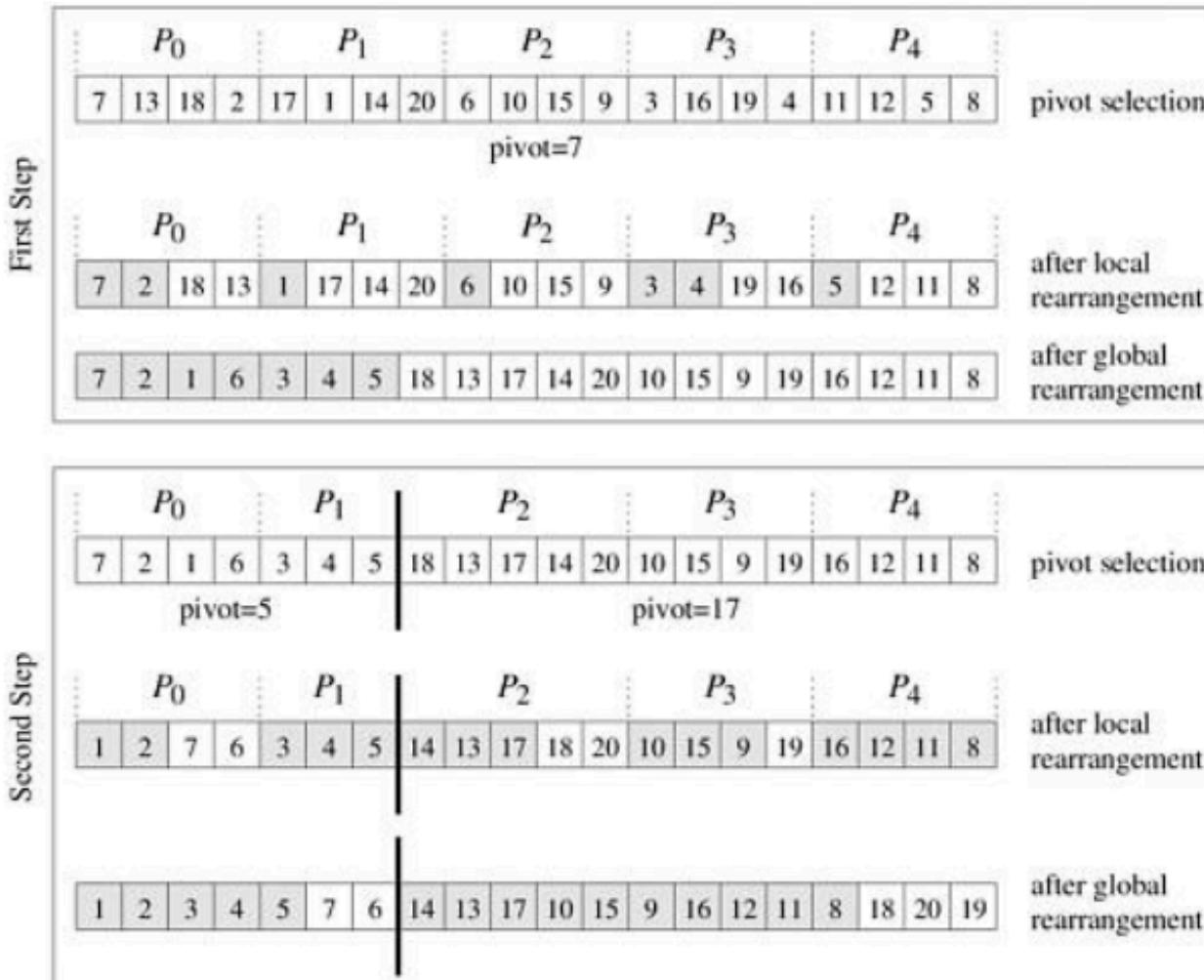
```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p – 1)
        quicksort(A, p + 1, hi)
```

```
algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo    // place for swapping
    for j := lo to hi – 1 do
        if A[j] ≤ pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```



# Parallel Quick Sort (1)

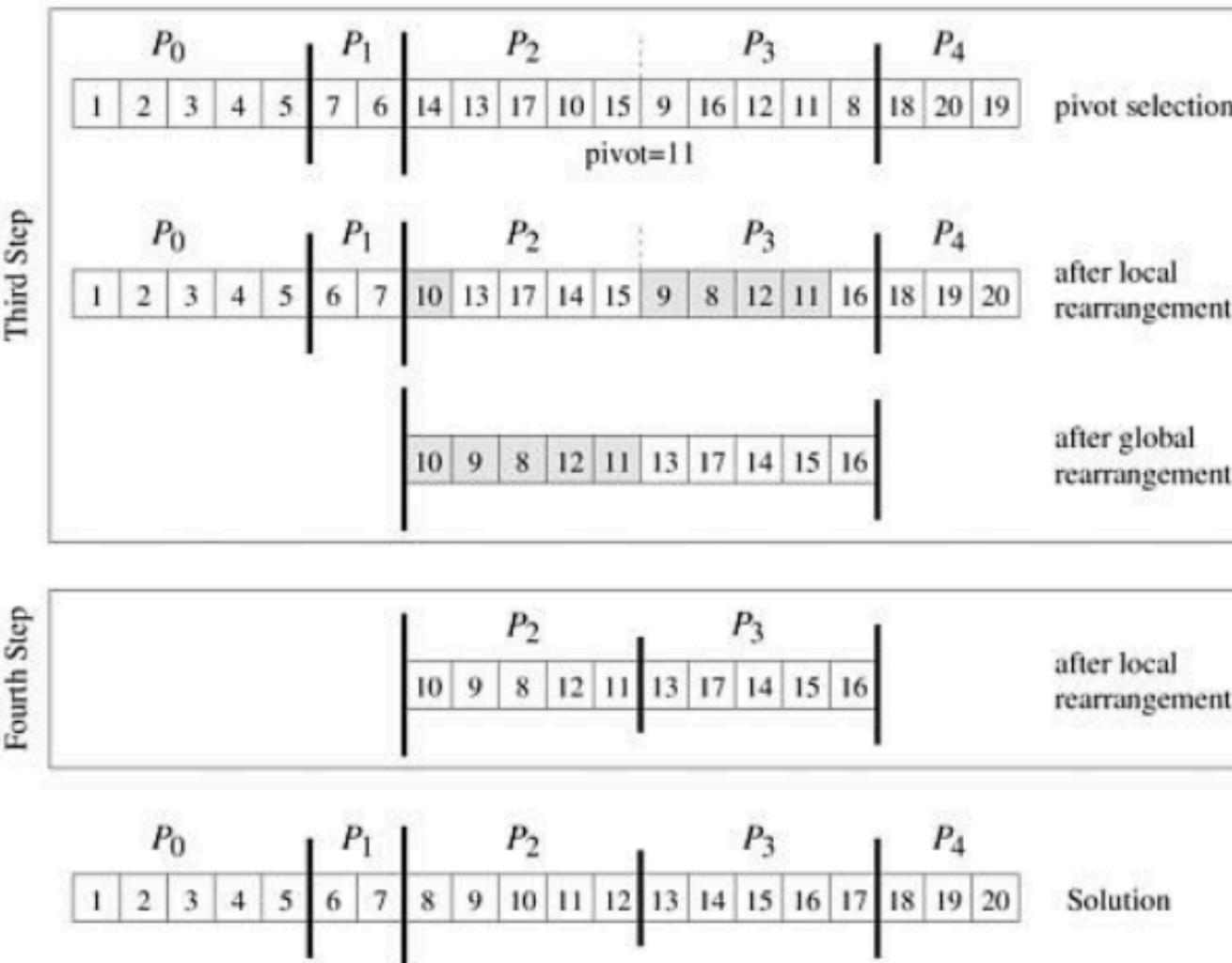
[http://srmcse.weebly.com/uploads/8/9/0/9/8909020/introduction\\_to\\_parallel\\_computing\\_second\\_edition-ananth\\_grama..pdf](http://srmcse.weebly.com/uploads/8/9/0/9/8909020/introduction_to_parallel_computing_second_edition-ananth_grama..pdf)  
<http://parallelcomp.uw.hu/ch09lev1sec4.html>





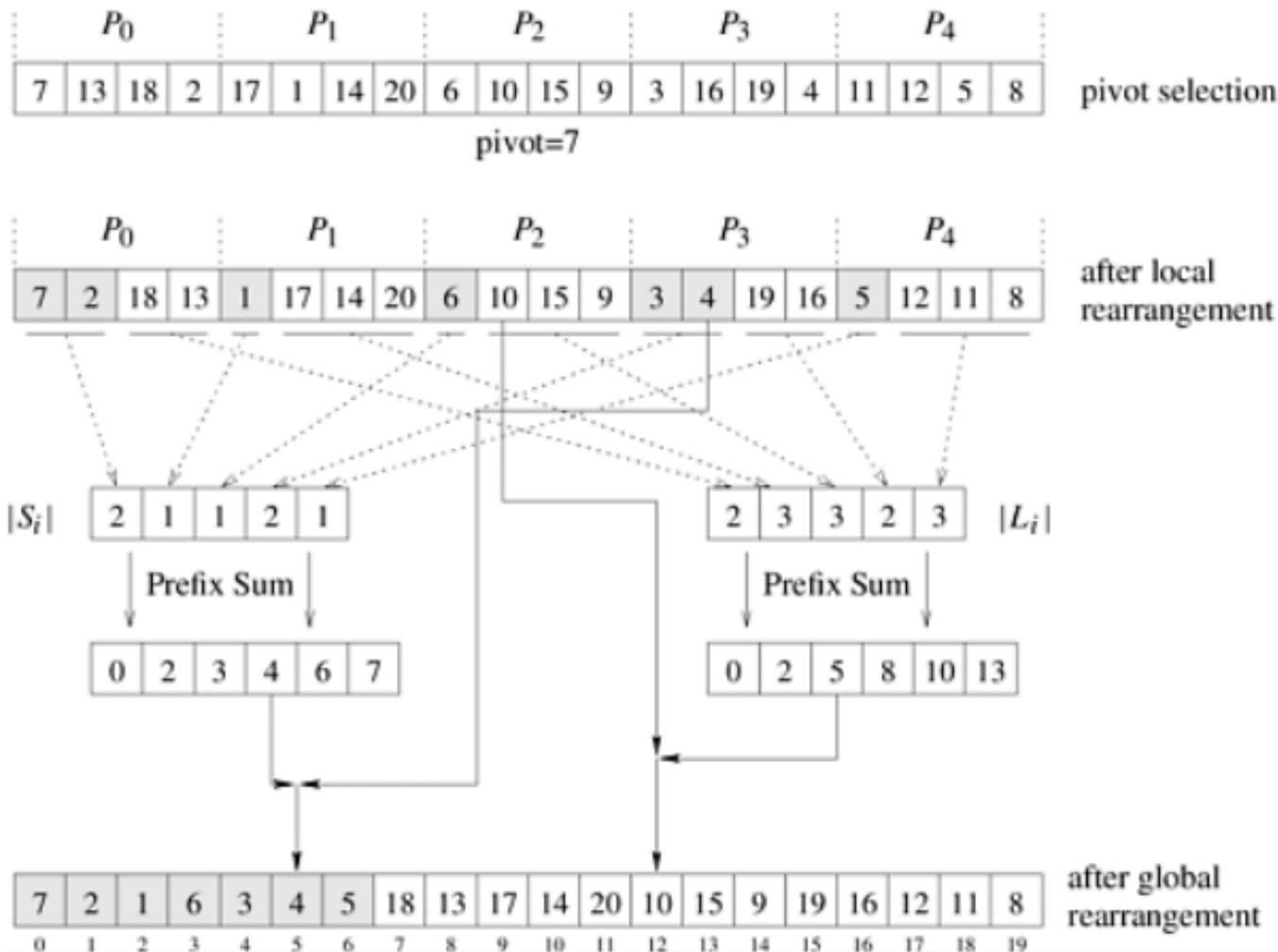
# Parallel Quick Sort (2)

[http://srmcse.weebly.com/uploads/8/9/0/9/8909020/introduction\\_to\\_parallel\\_computing\\_second\\_edition-ananth\\_grama..pdf](http://srmcse.weebly.com/uploads/8/9/0/9/8909020/introduction_to_parallel_computing_second_edition-ananth_grama..pdf)  
<http://parallelcomp.uw.hu/ch09lev1sec4.html>



# Parallel Quick Sort (3)

[http://srmcse.weebly.com/uploads/8/9/0/9/8909020/introduction\\_to\\_parallel\\_computing\\_second\\_edition-ananth\\_grama..pdf](http://srmcse.weebly.com/uploads/8/9/0/9/8909020/introduction_to_parallel_computing_second_edition-ananth_grama..pdf)  
<http://parallelcomp.uw.hu/ch09lev1sec4.html>



# **Chapter 4**

# **Parallel Computer Architectures**

---

**Thoai Nam**

Faculty of Computer Science and Engineering  
HCMC University of Technology



# Outline

---

- Flynn's Taxonomy
- Classification of Parallel Computers Based on Architectures



# Flynn's Taxonomy

---

- Based on notions of **instruction** and **data streams**
  - **SISD** (a **Single Instruction stream**, a **Single Data stream** )
  - **SIMD** (**Single Instruction stream**, **Multiple Data streams** )
  - **MISD** (**Multiple Instruction streams**, a **Single Data stream**)
  - **MIMD** (**Multiple Instruction streams**, **Multiple Data stream**)
- Popularity
  - **MIMD > SIMD > MISD**

## ❑ SISD

### – Conventional sequential machines

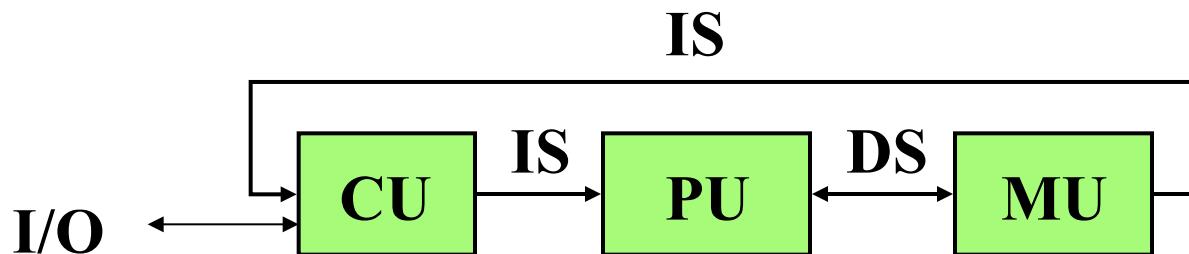
IS : Instruction Stream

CU : Control Unit

MU : Memory Unit

DS : Data Stream

PU : Processing Unit

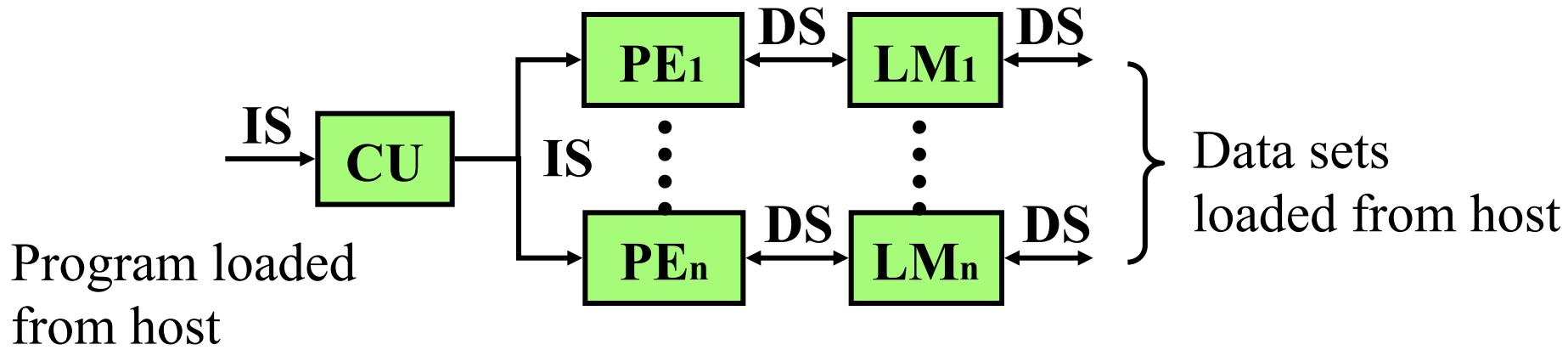


## ❑ SIMD

- Vector computers, processor arrays
- Special purpose computations

PE : Processing Element

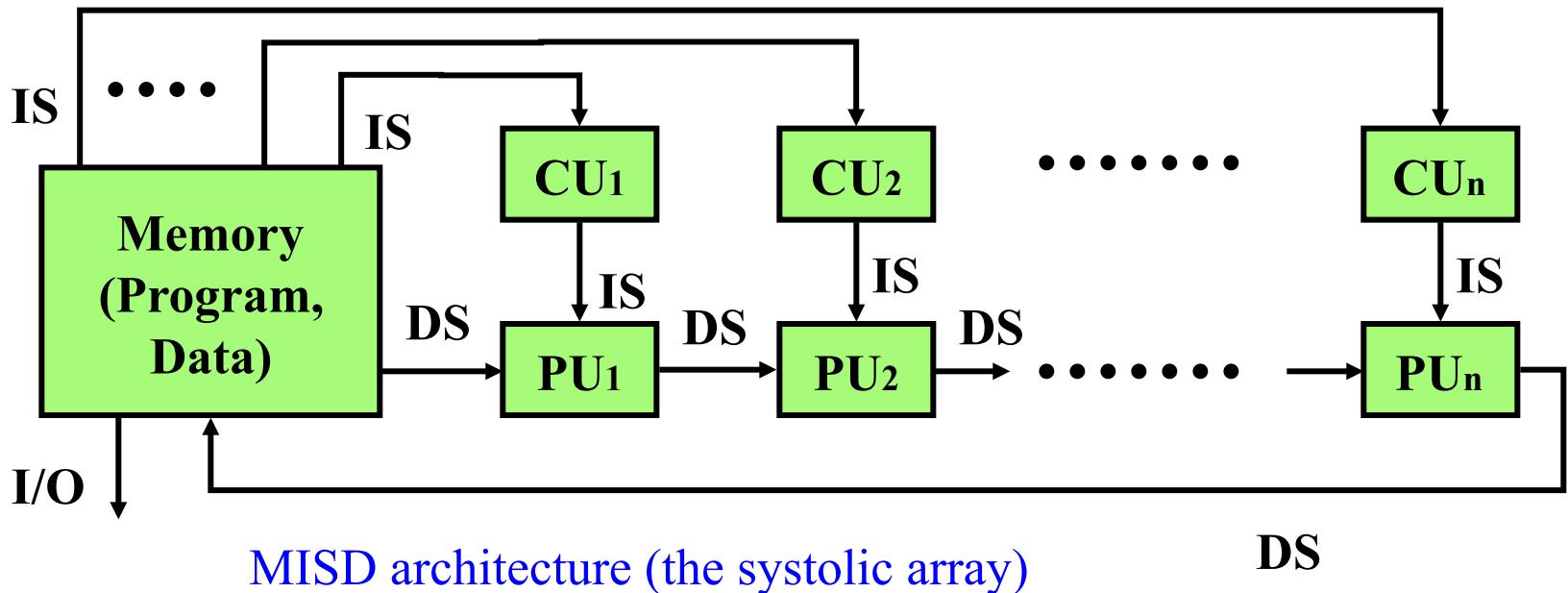
LM : Local Memory



SIMD architecture with distributed memory

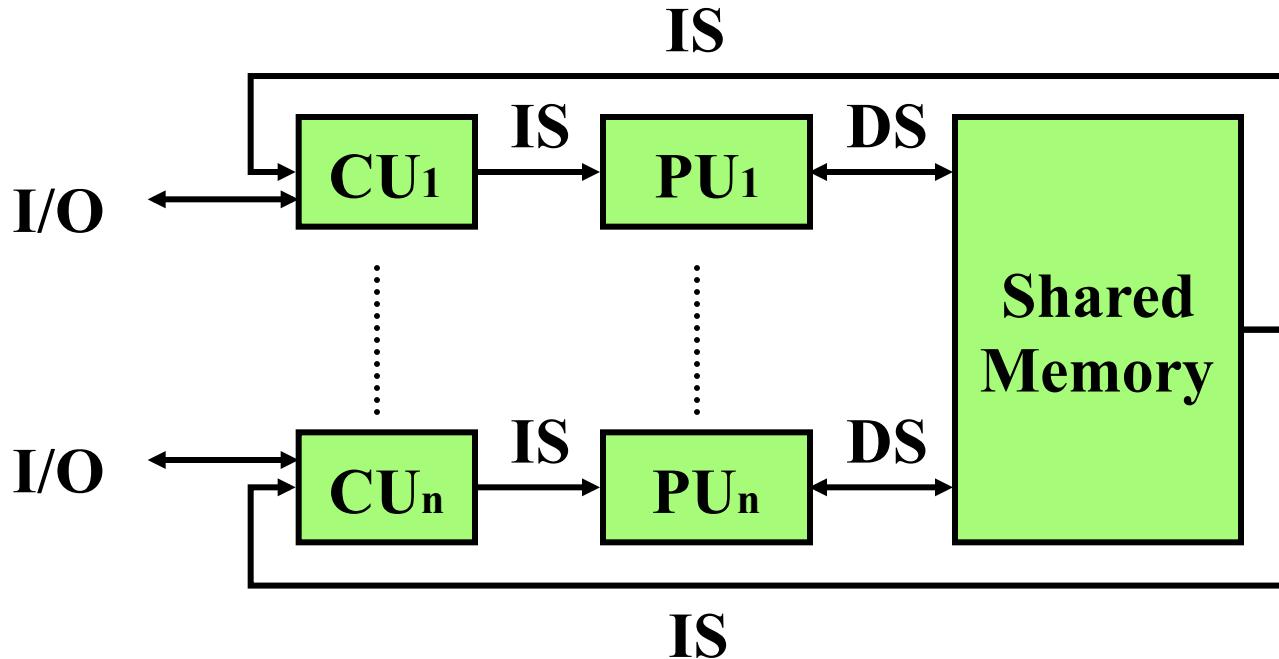
## ❑ MISD

- Systolic arrays
- Special purpose computations



## ❑ MIMD

- General purpose parallel computers



MIMD architecture with shared memory



# Classification based on Architecture

---

- Pipelined Computers
- Dataflow Architectures
- Data Parallel Systems
- Multiprocessors
- Multicomputers



# Pipelined Computers (1)

---

- Instructions are divided into a number of steps (segments, stages)
- At the same time, several instructions can be loaded in the machine and be executed in different steps

# Pipeline Computers (2)

- **IF**: instruction fetch
- **ID**: instruction decode and register fetch
- **EX**: execution and effective address calculation
- **MEM**: memory access
- **WB**: write back

Cycles

Instruction #	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB



# Dataflow Architecture

---

## □ Data-driven model

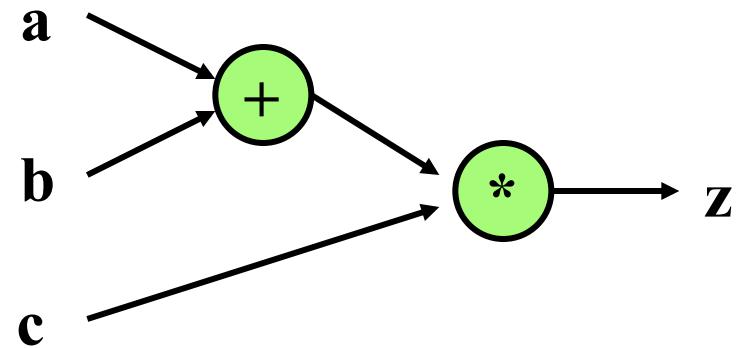
- A program is represented as a directed acyclic graph in which **a node represents an instruction** and **an edge represents the data dependency relationship** between the connected nodes
- Firing rule
  - » A node can be scheduled for execution if and only if its input data become valid for consumption

## □ Dataflow languages

- Id, SISAL, Silage, LISP,...
- Single assignment, applicative(functional) language
- Explicit parallelism

# Dataflow Graph

$$z = (a + b) * c$$



The dataflow representation of an arithmetic expression



# Dataflow Computer

- Execution of instructions is driven by data availability
  - What is the difference between this and normal (control flow) computers?
- Advantages
  - Very high potential for parallelism
  - High throughput
  - Free from side-effect
- Disadvantages
  - Time lost waiting for unneeded arguments
  - High control overhead
  - Difficult in manipulating data structures

# Dataflow Representation

input d,e,f

$c_0 = 0$

for i from 1 to 4 do

o

begin

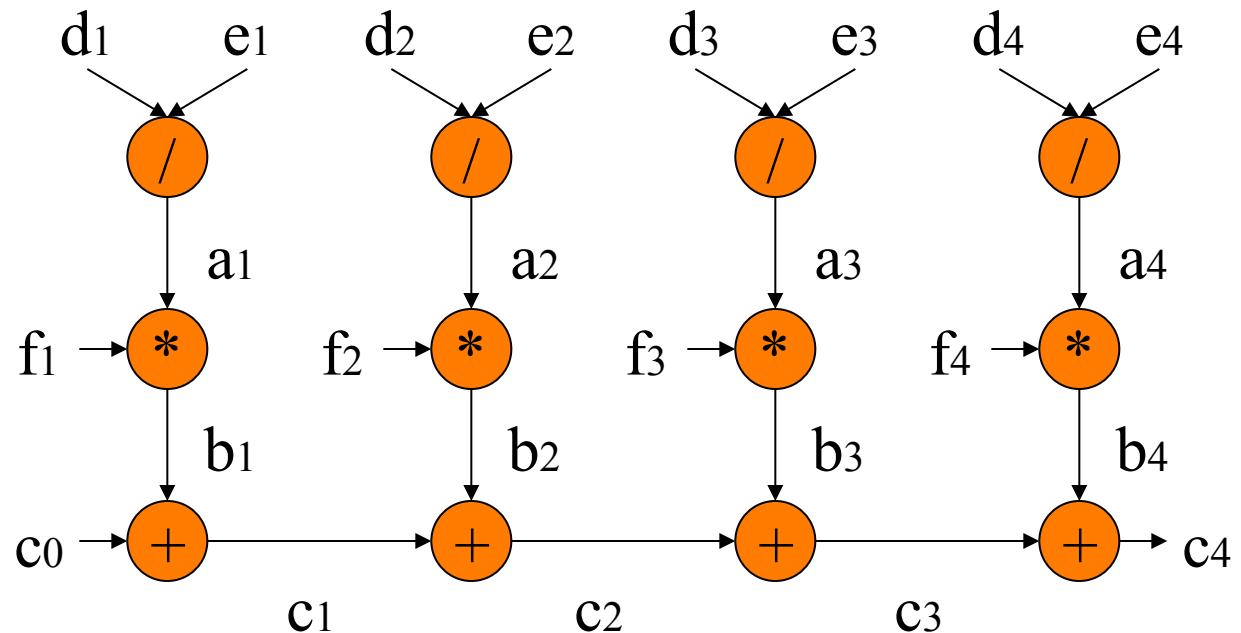
$a_i := d_i / e_i$

$b_i := a_i * f_i$

$c_i := b_i + c_{i-1}$

end

output a, b, c





# Execution on a Control Flow Machine

---

Assume all the external inputs are available before entering do loop

+ : 1 cycle, \* : 2 cycles, / : 3 cycles,

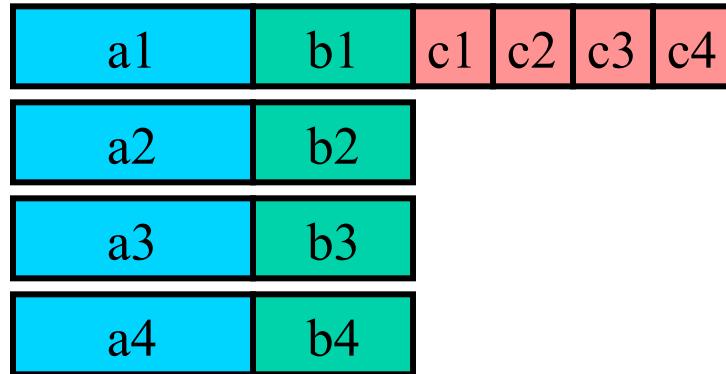


Sequential execution on a uniprocessor in 24 cycles

How long will it take to execute this program on a dataflow computer with 4 processors?

# Execution on a Dataflow Machine

---



Data-driven execution on a 4-processor dataflow computer in 9 cycles

Can we further reduce the execution time of this program ?



# Data Parallel Systems (1)

---

## □ Programming model

- Operations performed in parallel on each element of data structure
- Logically single thread of control, performs sequential or parallel steps
- Conceptually, a processor associated with each data element



# Data Parallel Systems (2)

---

## □ SIMD Architectural model

- Array of many simple, cheap processors with little memory each
  - » Processors don't sequence through instructions
- Attached to a control processor that issues instructions
- Specialized and general communication, cheap global synchronization



# Vector Processors

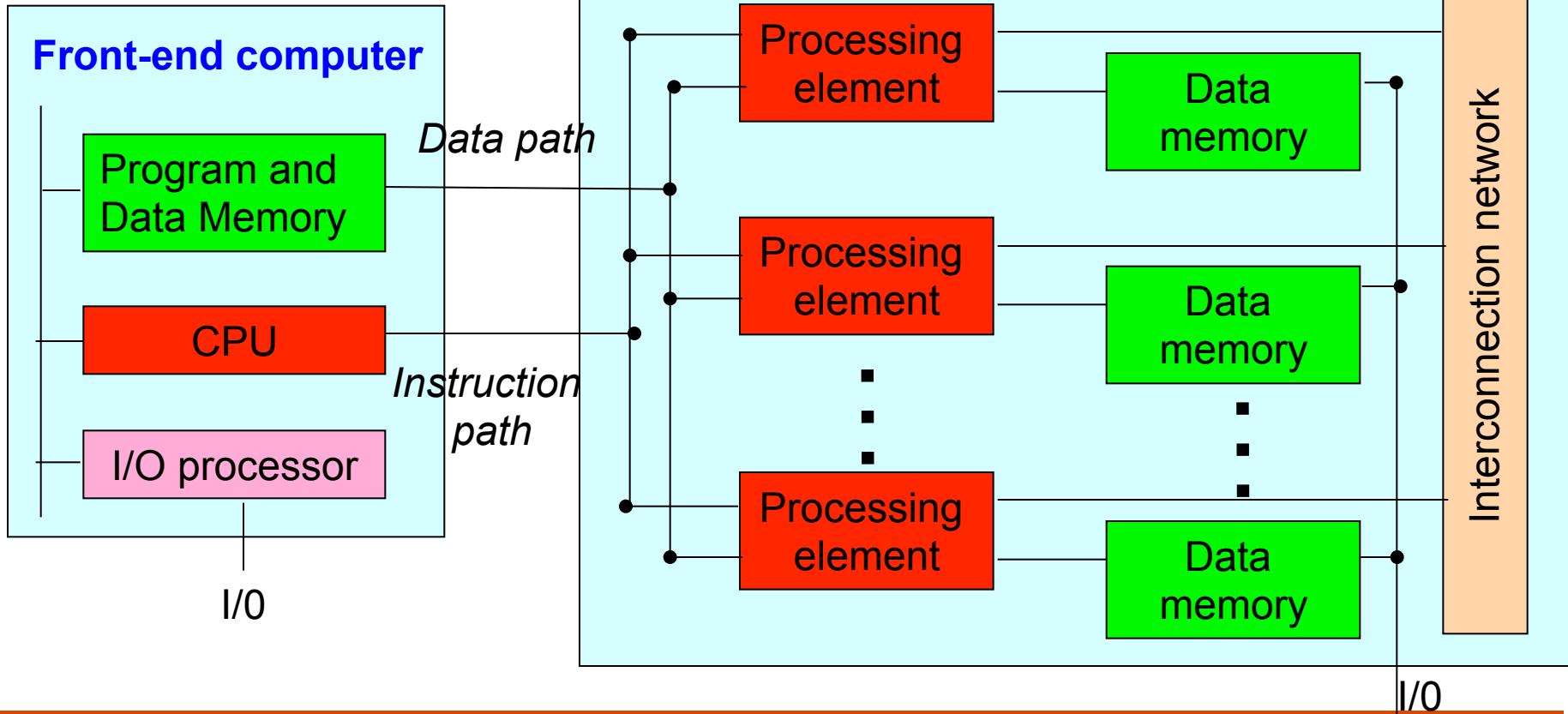
---

- Instruction set includes operations on vectors as well as scalars
- 2 types of vector computers
  - Processor arrays
  - Pipelined vector processors

# Processor Array

A sequential computer connected with a set of identical processing elements simultaneously doing the same operation on different data.

Eg CM-200



# Pipeline Vector Processor

- Stream vector from memory to the CPU
- Use pipelined arithmetic units to manipulate data
- Eg: Cray-1, Cyber-205





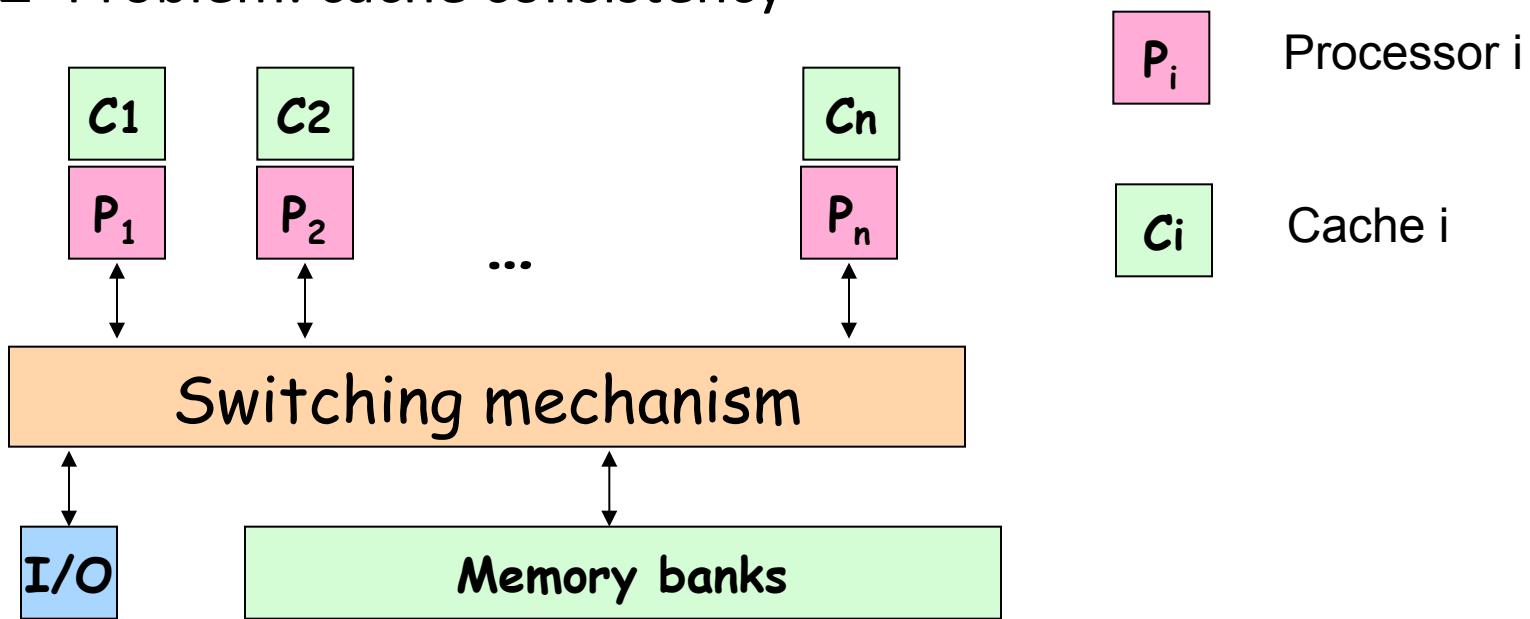
# Multiprocessor

---

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

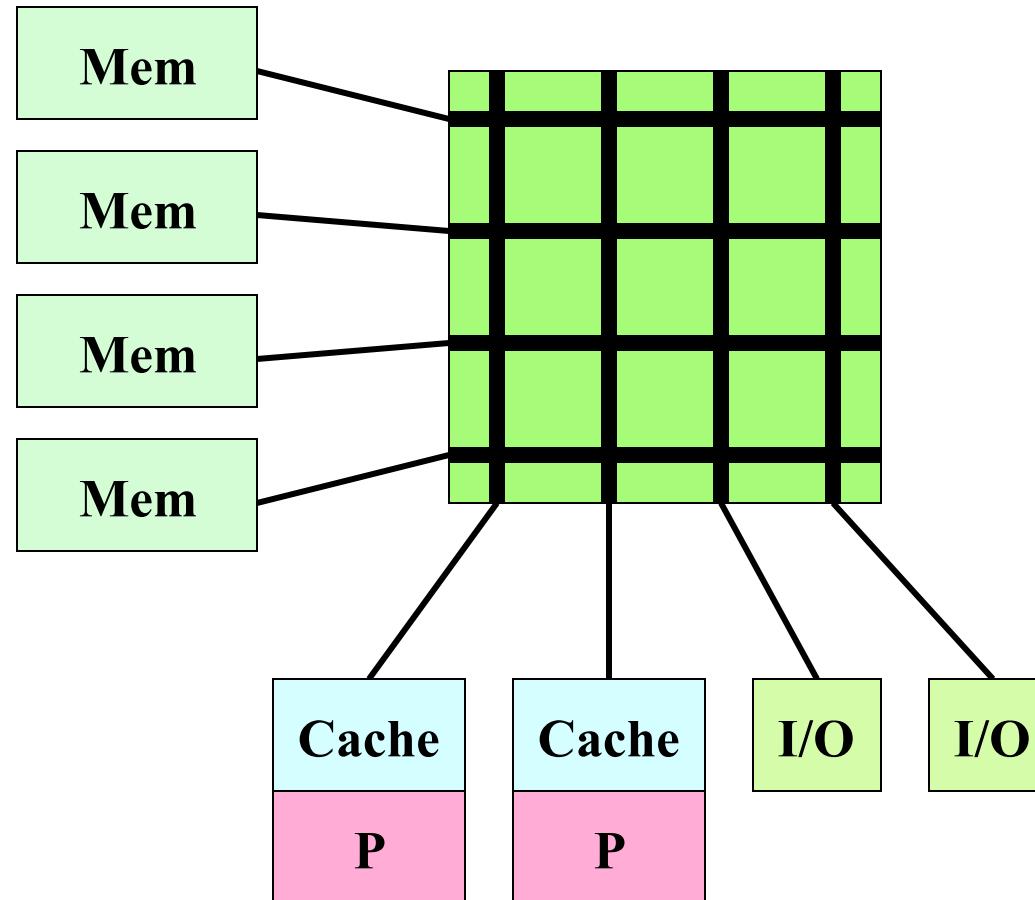
# UMA Multiprocessor (1)

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



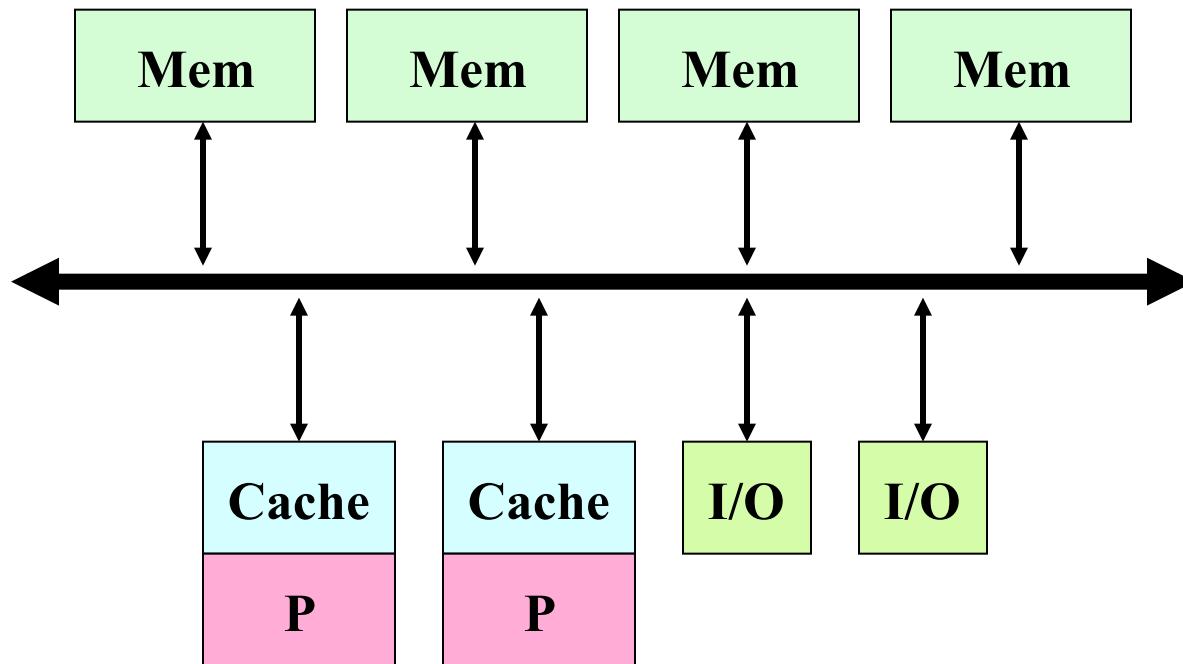
# UMA Multiprocessor (2)

- Crossbar switching mechanism



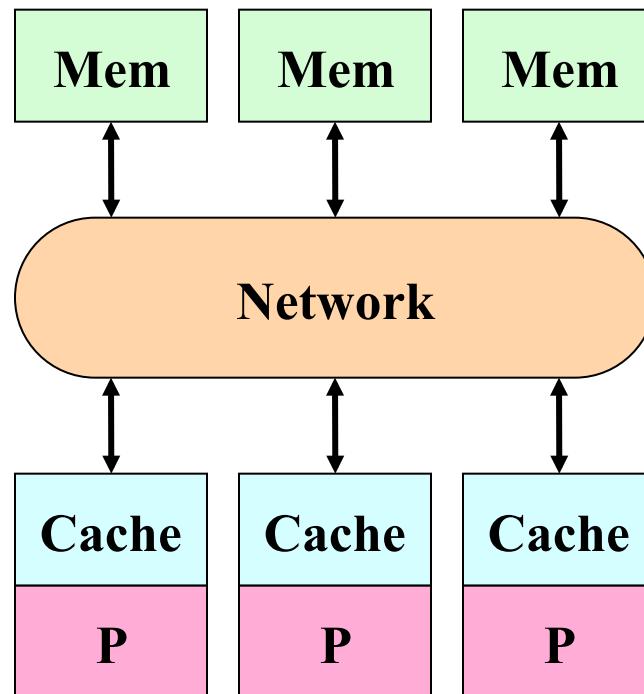
# UMA Multiprocessor (3)

- Shared-bus switching mechanism



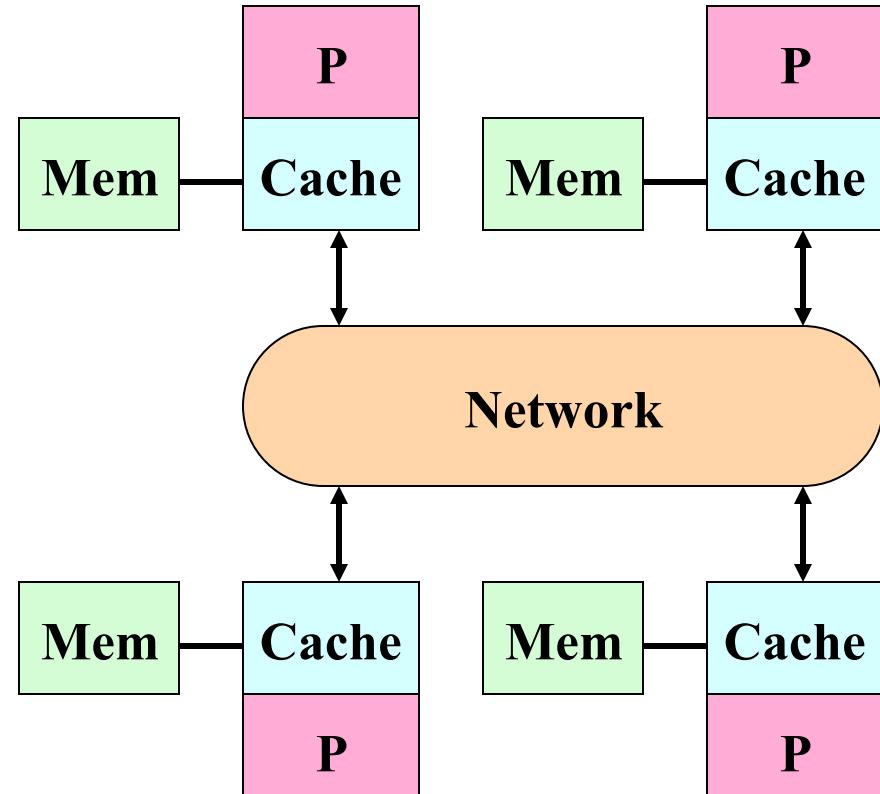
# UMA Multiprocessor (4)

- Packet-switched network



# NUMA Multiprocessor

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



Distributed Memory



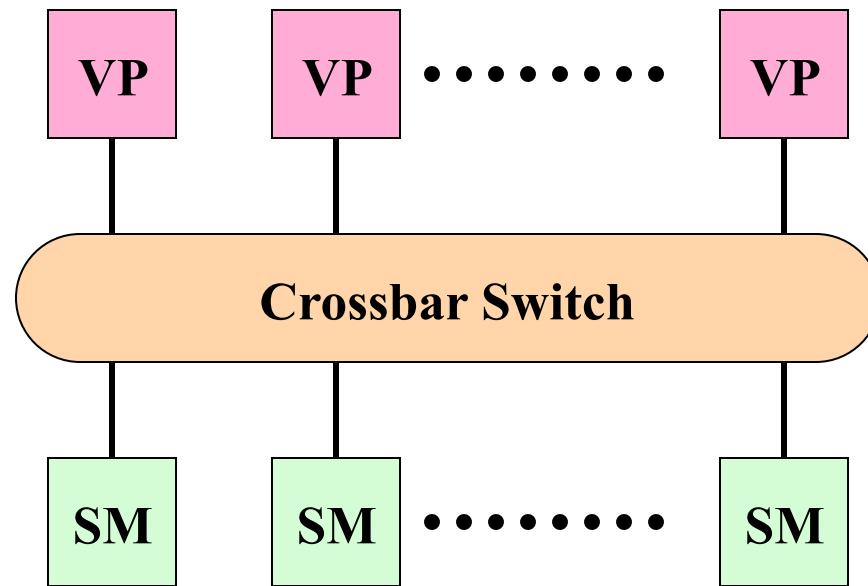
# Current Types of Multiprocessors

---

- PVP (Parallel Vector Processor)
  - A small number of proprietary vector processors connected by a high-bandwidth crossbar switch
- SMP (Symmetric Multiprocessor)
  - A small number of COST microprocessors connected by a high-speed bus or crossbar switch
- DSM (Distributed Shared Memory)
  - Similar to SMP
  - The memory is physically distributed among nodes.

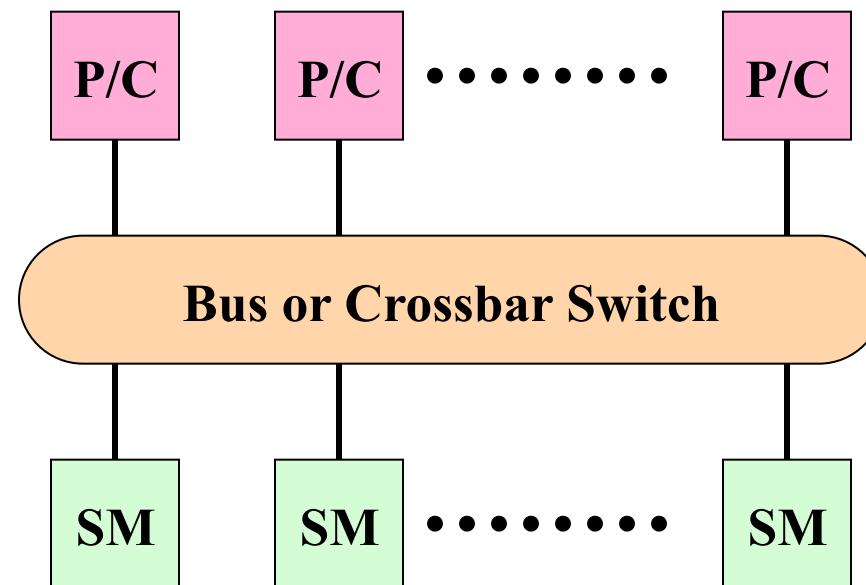
# PVP (Parallel Vector Processor)

VP : Vector Processor  
SM : Shared Memory



# SMP (Symmetric Multi-Processor)

P/C : Microprocessor and Cache  
SM: Shared Memory



# DSM (Distributed Shared Memory)

MB: Memory Bus

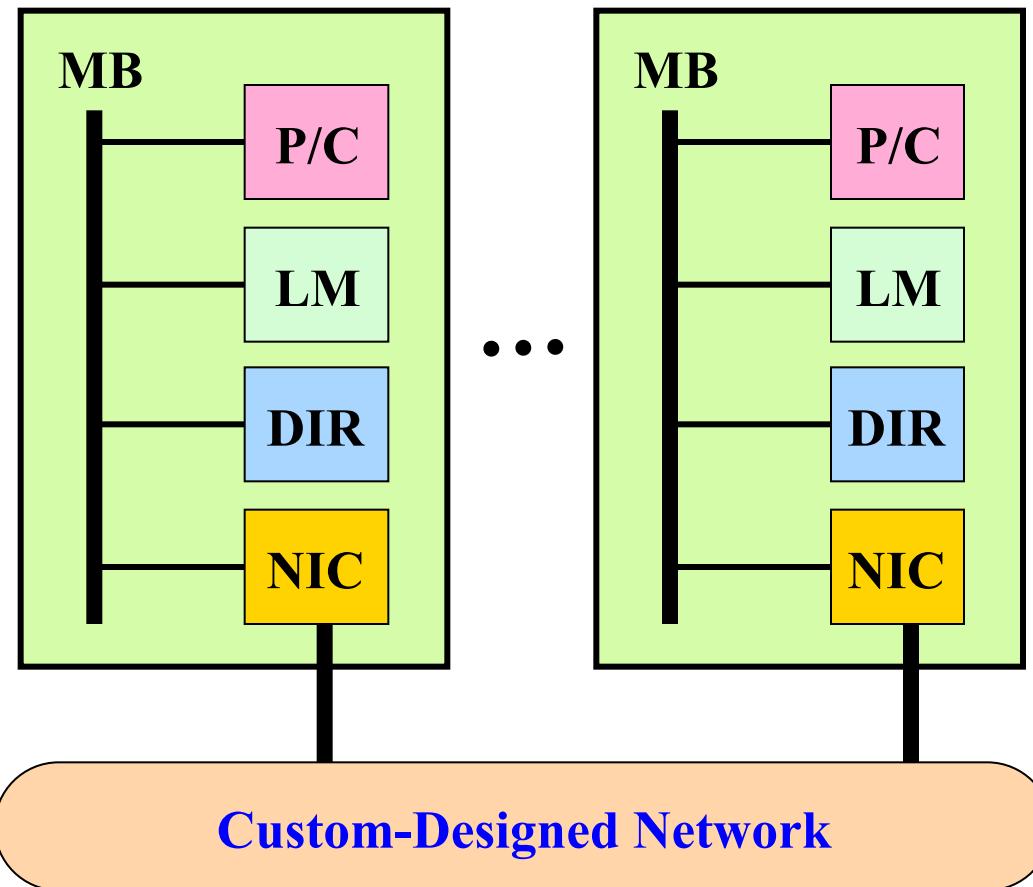
P/C: Microprocessor & Cache

LM: Local Memory

DIR: Cache Directory

NIC: Network Interface

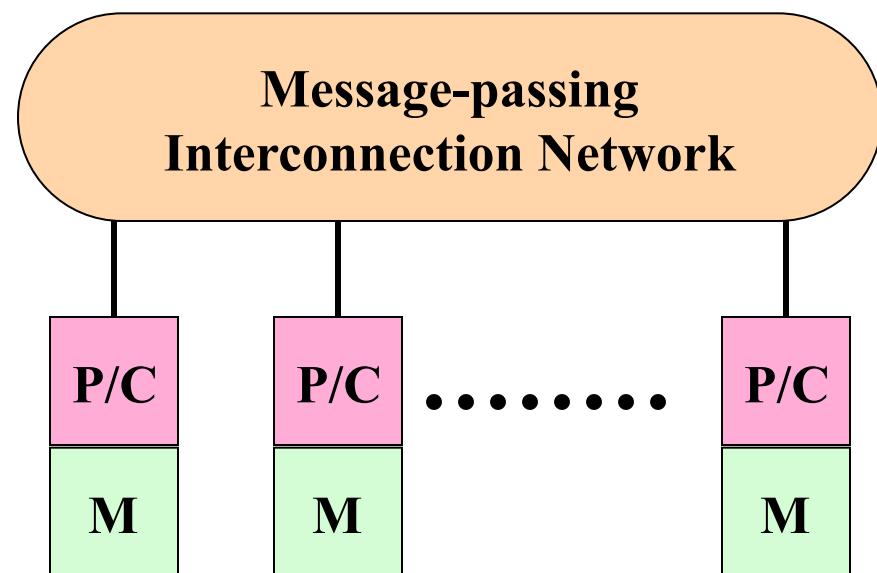
Circuitry



# Multicomputers

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

P/C: Microprocessor & Cache  
M: Memory





# Current Types of Multicomputers

---

- MPP (Massively Parallel Processing)
  - Total number of processors > 1000
- Cluster
  - Each node in system has less than 16 processors
- Constellation
  - Each node in system has more than 16 processors

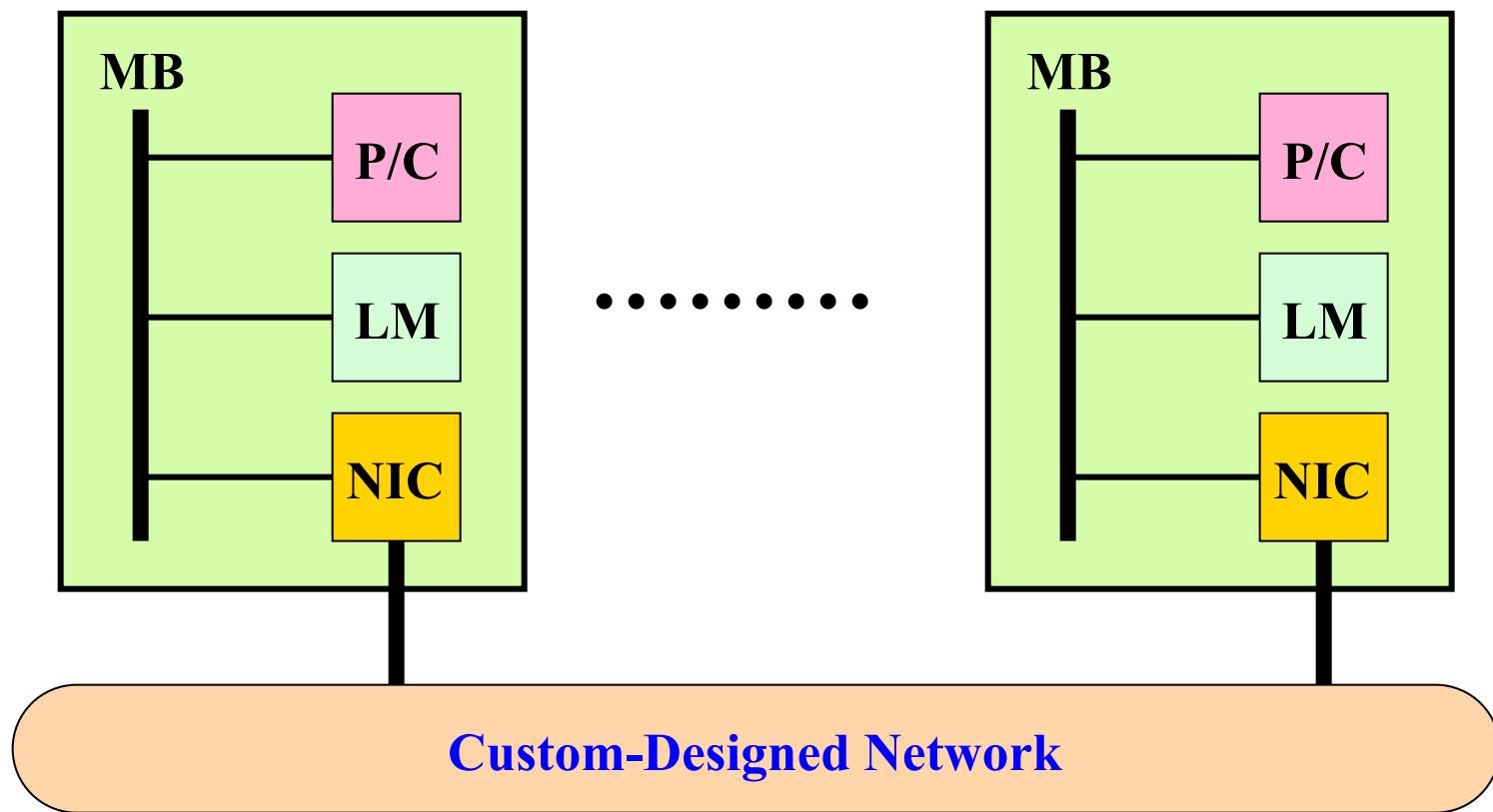
# MPP (Massively Parallel Processing)

P/C: Microprocessor & Cache

NIC: Network Interface Circuitry

MB: Memory Bus

LM: Local Memory



# Clusters

MB: Memory Bus

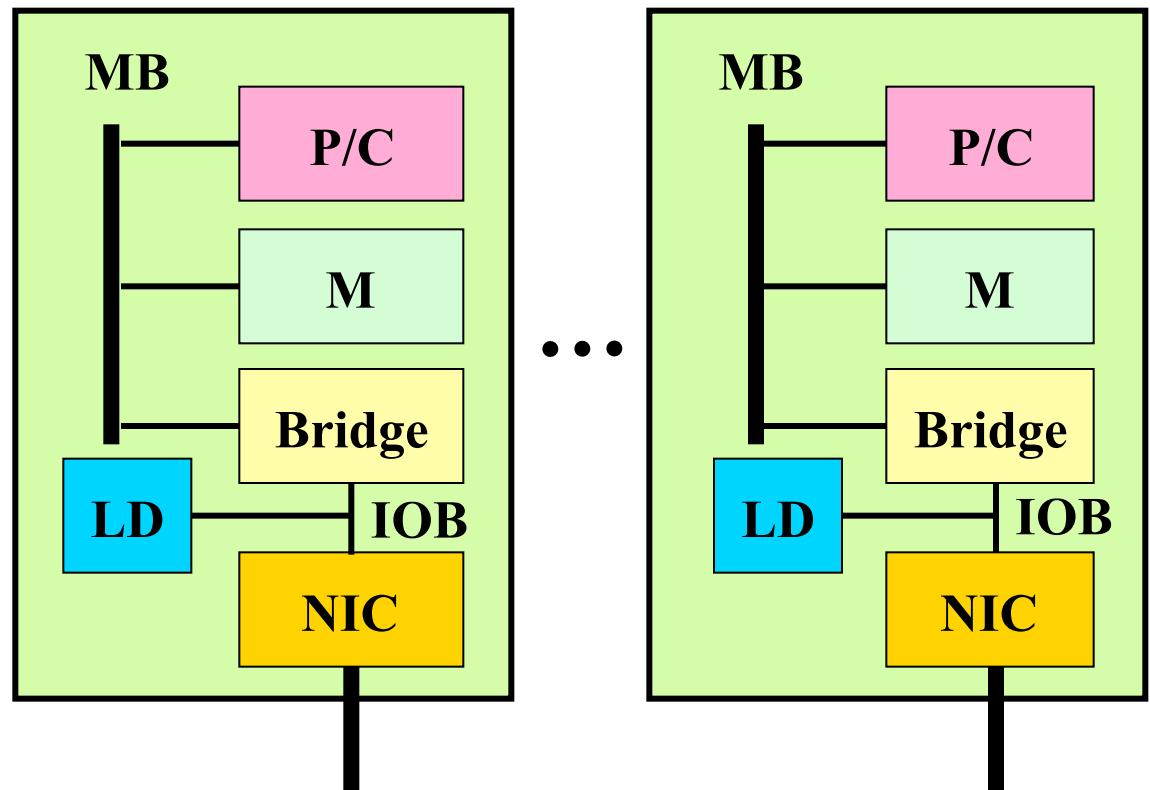
P/C: Microprocessor &  
Cache

M: Memory

LD: Local Disk

IOB: I/O Bus

NIC: Network Interface  
Circuitry



**Commodity Network (Ethernet, ATM, Myrinet, InfiniBand (VIA))**



# Constellations

P/C: Microprocessor & Cache

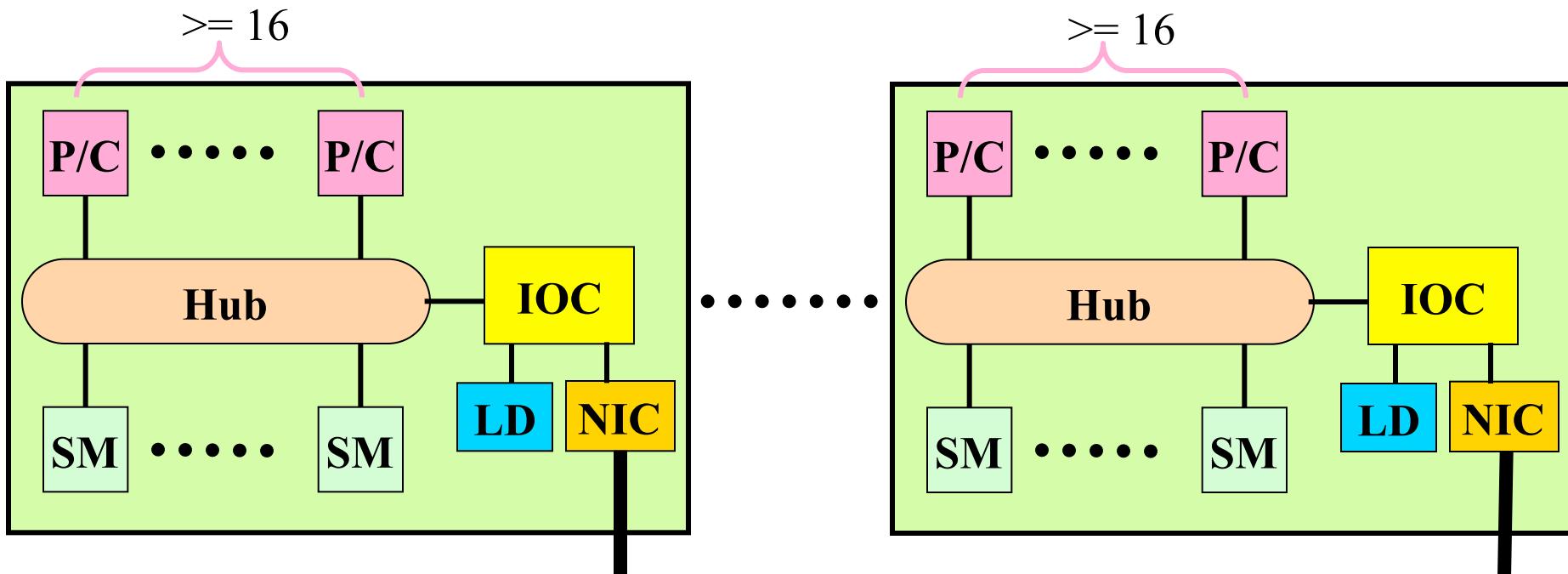
NIC: Network Interface Circuitry

IOC: I/O Controller

MB: Memory Bus

SM: Shared Memory

LD: Local Disk



Custom or Commodity Network

# Speedup

---

Thoai Nam

Faculty of Computer Science and Engineering

HCMC University of Technology



# Outline

---

- Speedup & Efficiency
- Amdahl's Law
- Gustafson's Law
- Sun & Ni's Law



# Speedup & Efficiency

---

## □ Speedup:

$S = \frac{\text{Time}(\text{the most efficient sequential algorithm})}{\text{Time}(\text{parallel algorithm})}$

## □ Efficiency:

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



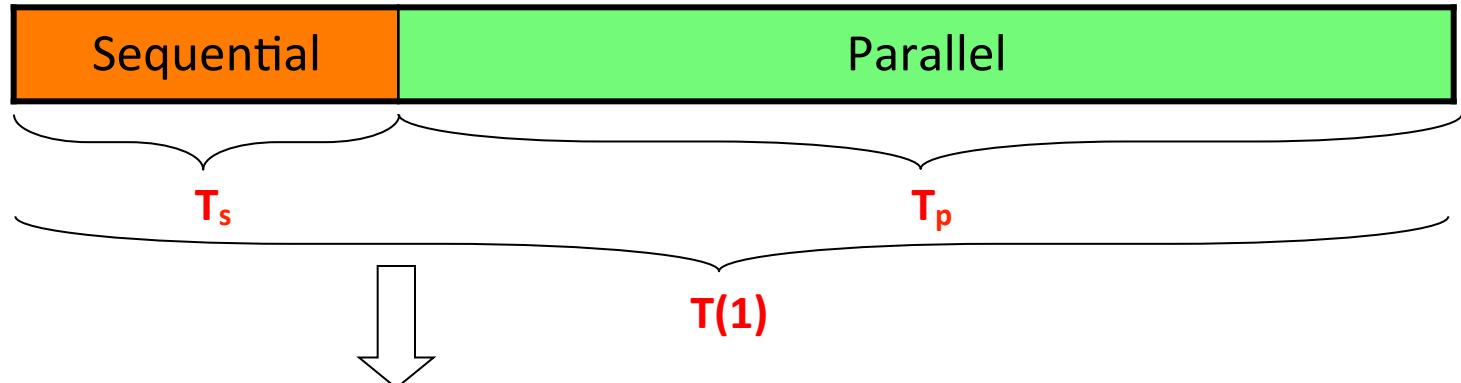
# Amdahl's Law – Fixed Problem Size (1)

---

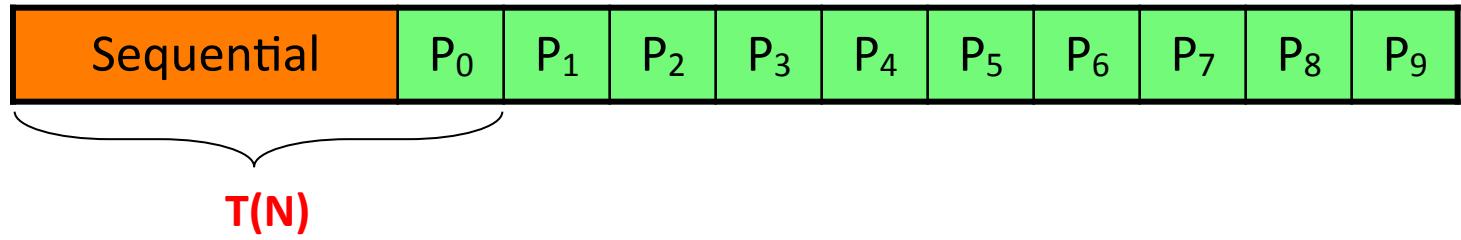
- The main objective is to produce the results as soon as possible
  - (ex) video compression, computer graphics, VLSI routing, etc
- Implications
  - Upper-bound is
  - Make Sequential bottleneck as small as possible
  - Optimize the common case
- Modified Amdahl's law for **fixed problem size** including the overhead

# Amdahl's Law – Fixed Problem Size (2)

Sequential



Parallel



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

Number of  
processors

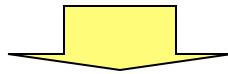
$$T(N) = \alpha T(1) + (1-\alpha)T(1)/N$$



# Amdahl's Law – Fixed Problem Size (3)

---

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



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



# Enhanced Amdahl's Law

---

The overhead includes parallelism  
and interaction overheads

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

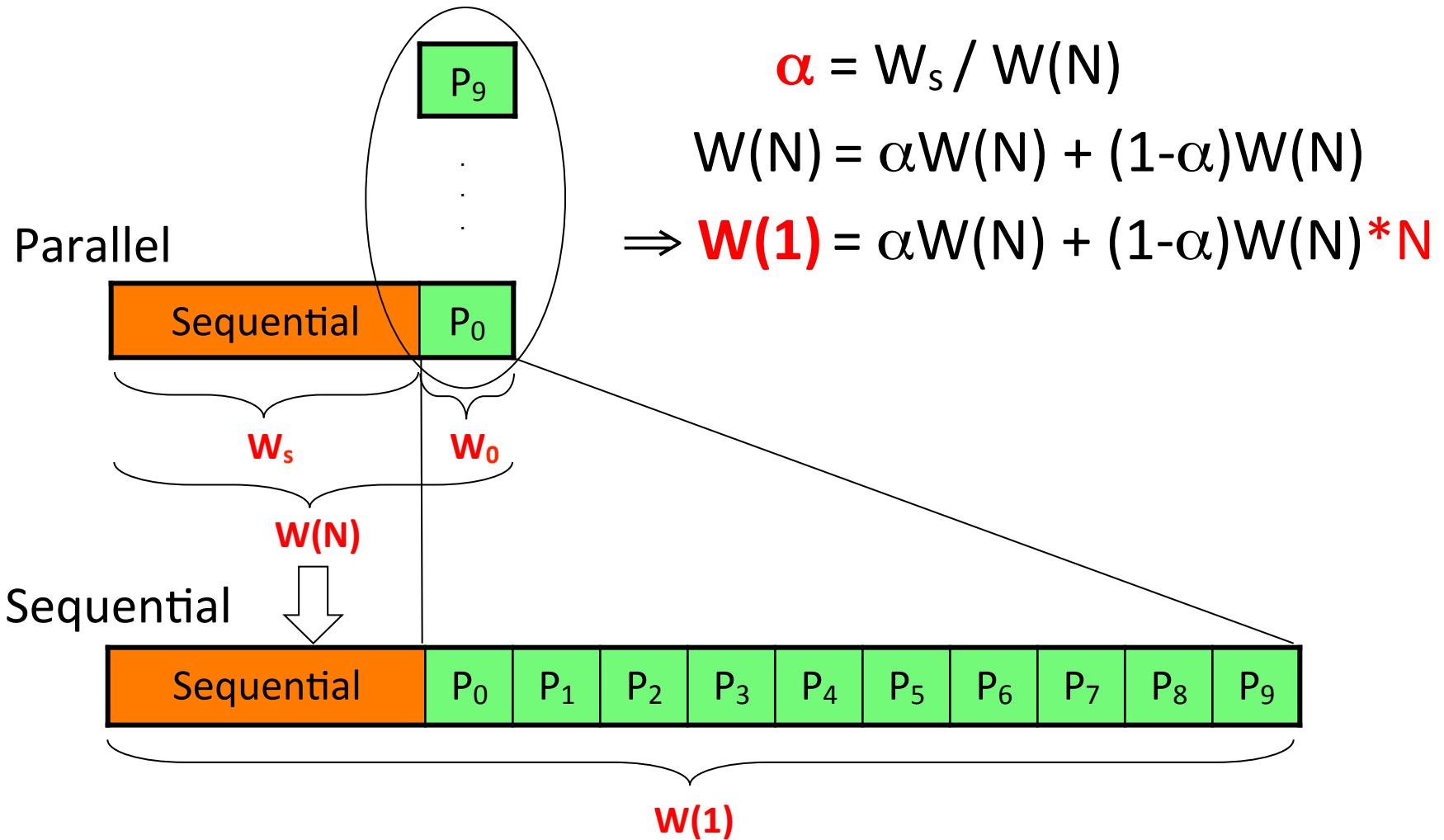


# Gustafson's Law – Fixed Time (1)

---

- User wants more accurate results within a time limit
    - Execution time is fixed as system scales
    - (ex) FEM (Finite element method) for structural analysis, FDM (Finite difference method) for fluid dynamics
  - Properties of a work metric
    - Easy to measure
    - Architecture independent
    - Easy to model with an analytical expression
    - No additional experiment to measure the work
    - The measure of work should scale linearly with sequential time complexity of the algorithm
  - Time constrained seems to be most generally viable model!
-

# Gustafson's Law – Fixed Time (2)





# Gustafson's Law – Fixed Time without overhead

---

Time = Work \* k

$W(N) = W$

$$Speedup = \frac{T(1)}{T(N)} = \frac{W(1)*k}{W(N)*k} = \frac{\alpha W + (1-\alpha)NW}{W} = \alpha + (1-\alpha)N$$



# Gustafson's Law – Fixed Time with overhead

---

$$W(N) = W + W_0$$

$$\text{Speedup} = \frac{T(1)}{T(N)} = \frac{W(1)*k}{W(N)*k} = \frac{\alpha W + (1-\alpha)NW}{W + W_0} = \frac{\alpha + (1-\alpha)N}{1 + \frac{W_0}{W}}$$



# Sun and Ni's Law – Fixed Memory (1)

---

- Scale the largest possible solution limited by the memory space. Or, fix memory usage per processor
- Speedup
  - $\text{Time}(1)/\text{Time}(N)$  for scaled up problem is not appropriate
  - For simple profile, and **G(N)** is the increase of parallel workload as **the memory capacity** increases N times



# Sun and Ni's Law – Fixed Memory (2)

---

- $W = \alpha W + (1 - \alpha)W$
- Let  $M$  be the memory capacity of a single node
- $N$  nodes:
  - the increased memory  $N * M$
  - The scaled work:  $W = \alpha W + (1 - \alpha)W * G(N)$

$$Speedup_{MC} = \frac{\alpha + (1 - \alpha)G(N)}{\alpha + (1 - \alpha)\frac{G(N)}{N}}$$



# Sun and Ni's Law – Fixed Memory (3)

---

## □ Definition:

A function  $g$  is homomorphism if there exists a function such that  $\bar{g}$  for any real number  $c$  and variable  $x$ ,

$$g(cx) = \bar{g}(c) * g(x)$$

## □ Theorem:

If  $W = g(M)$  for some homomorphism function  $g$ , then with all data being shared by all available processors, the simplified memory-bounced speedup is

$$S_N^* = \frac{W_1 + \bar{g}(N)W_N}{W_1 + \frac{\bar{g}(N)}{N}W_N} = \frac{\alpha + (1 - \alpha)G(N)}{\alpha + (1 - \alpha)\frac{G(N)}{N}}$$



# Sun and Ni's Law – Fixed Memory (4)

---

Proof:

Let the memory requirement of  $W_n$  be  $M$ ,  $W_n = g(M)$ .

$M$  is the memory requirement when 1 node is available.

With  $N$  nodes available, the memory capacity will increase to  $N*M$ .

Using all of the available memory, for the scaled parallel portion  $W_N^*$ :  $W_N^* = g(N*M) = \bar{g}(N)*g(M) = \bar{g}(N)*W_N$

$$S_N^* = \frac{W_1^* + W_N^*}{W_1^* + \frac{W_N^*}{N}} = \frac{W_1 + \bar{g}(N)W_N}{W_1 + \frac{\bar{g}(N)}{N}W_N}$$

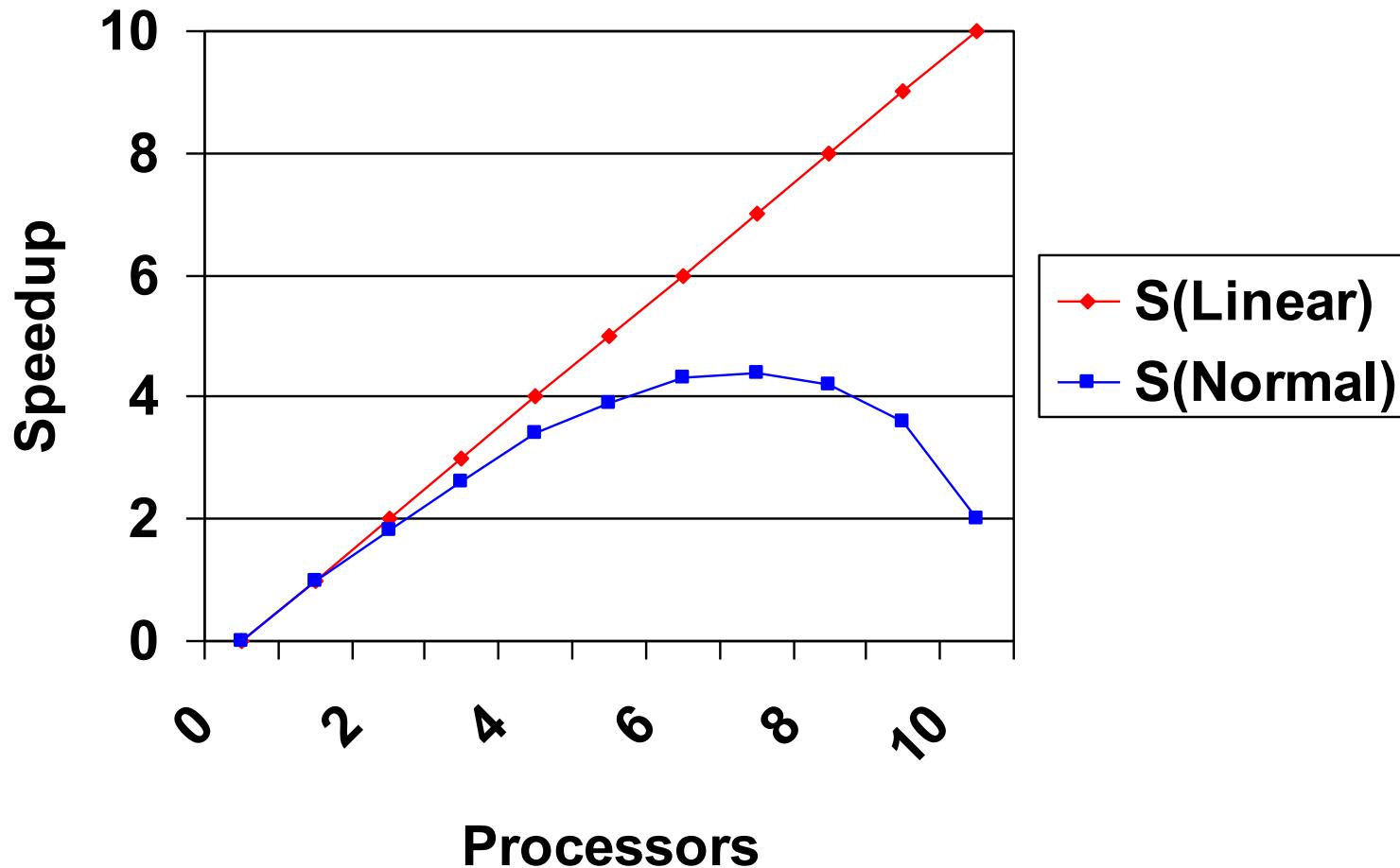


# Speedup

$$S_N^* = \frac{W_1 + G(N)W_N}{W_1 + \frac{G(N)}{N}W_N}$$

- When the problem size is independent of the system, the problem size is fixed,  $G(N)=1 \Rightarrow$  Amdahl's Law.
- When memory is increased N times, the workload also increases N times,  $G(N)=N \Rightarrow$  Gustafson's Law
- For most of the scientific and engineering applications, the computation requirement increases faster than the memory requirement,  $G(N)>N$ .

# Examples





# Scalability

---

- Parallelizing a code does not always result in a speedup; sometimes it actually slows the code down! This can be due to a poor choice of algorithm or to poor coding
- The best possible speedup is **linear**, i.e. it is proportional to the number of processors:  $T(N) = T(1)/N$  where **N = number of processors,  $T(1)$  = time for serial run.**
- A code that continues to speed up reasonably close to linearly as the number of processors increases is said to be **scalable**. Many codes scale up to some number of processors but adding more processors then brings no improvement. Very few, if any, codes are indefinitely scalable.



# Factors That Limit Speedup

---

## □ Software overhead

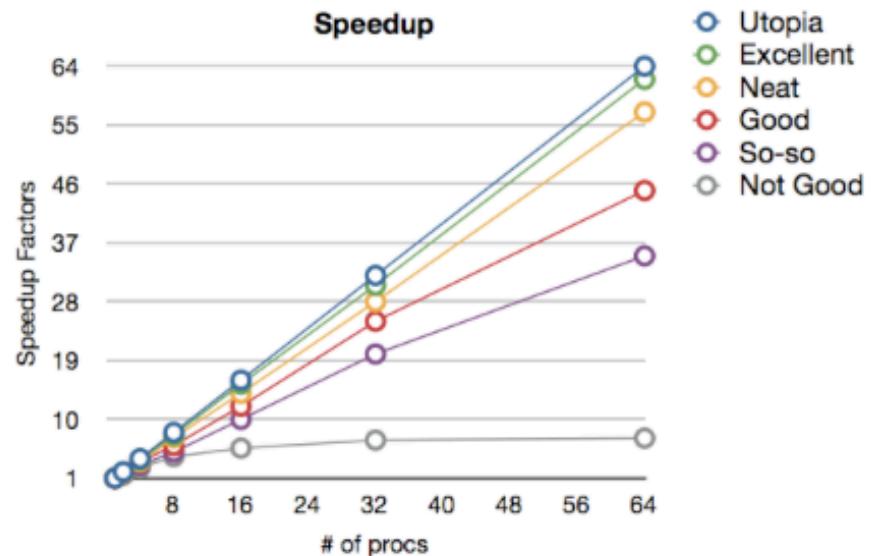
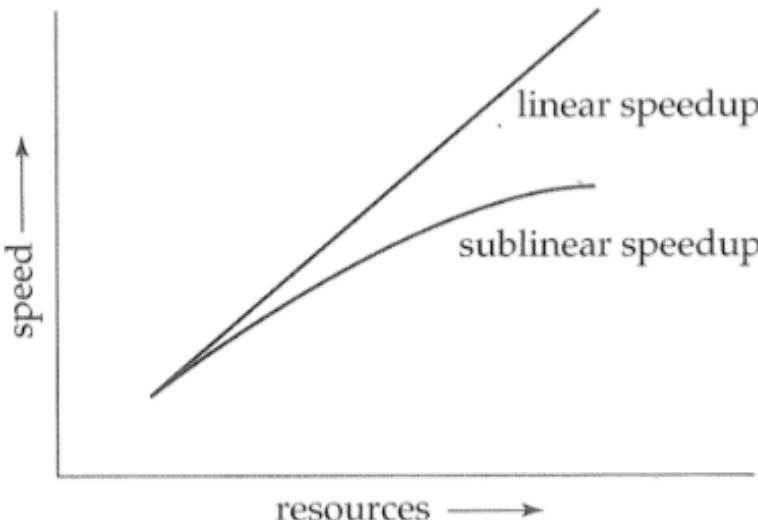
Even with a completely equivalent algorithm, software overhead arises in the concurrent implementation. (e.g. there may be additional index calculations necessitated by the manner in which data are "split up" among processors.) i.e. there is generally more lines of code to be executed in the parallel program than the sequential program.

## □ Load balancing

## □ Communication overhead

# Speedup

- The fundamental concept in parallelism
  - $T(1)$  = time to execute task on a single resource
  - $T(n)$  = time to execute task on  $n$  resources
  - Speedup =  $T(1)/T(n)$



[http://web.eecs.utk.edu/~huangj/hpc/hpc\\_intro.php](http://web.eecs.utk.edu/~huangj/hpc/hpc_intro.php)

# Parallel Techniques

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

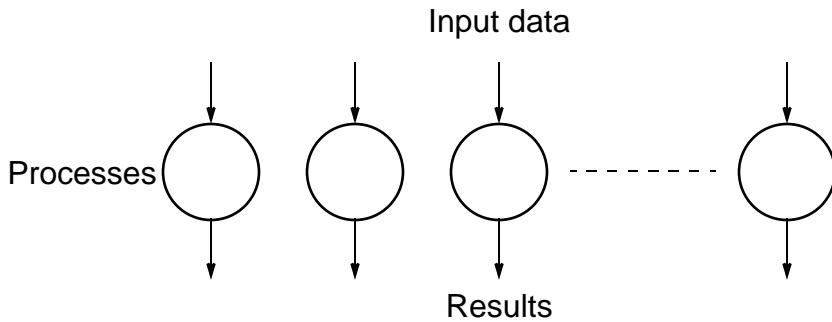
Not covered in 1st edition  
of textbook

## Chapter 3

# Embarrassingly Parallel Computations

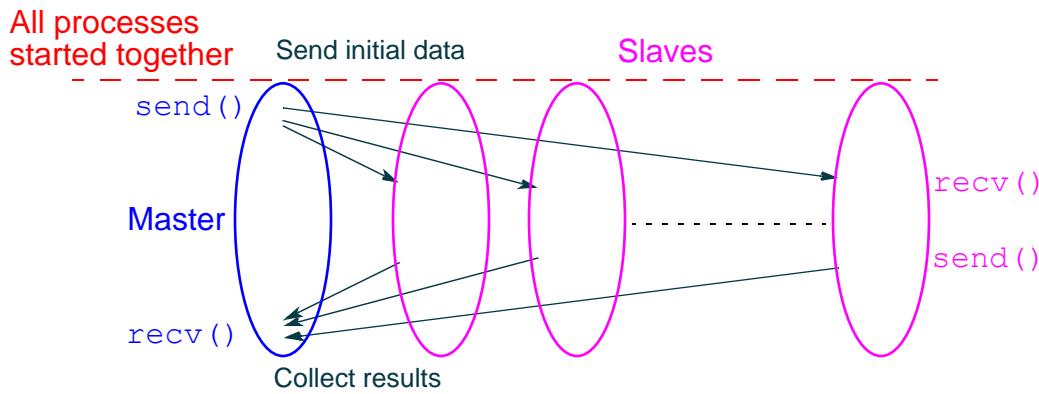
# Embarrassingly Parallel Computations

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



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

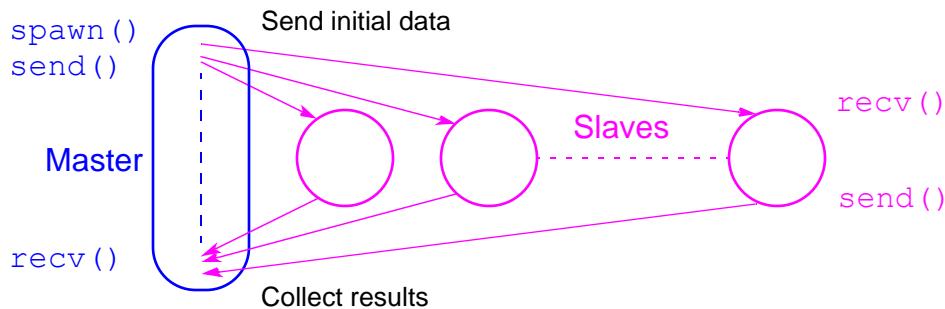
# Practical embarrassingly parallel computation with static process creation and master-slave approach



MPI approach

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

Start Master initially



PVM approach

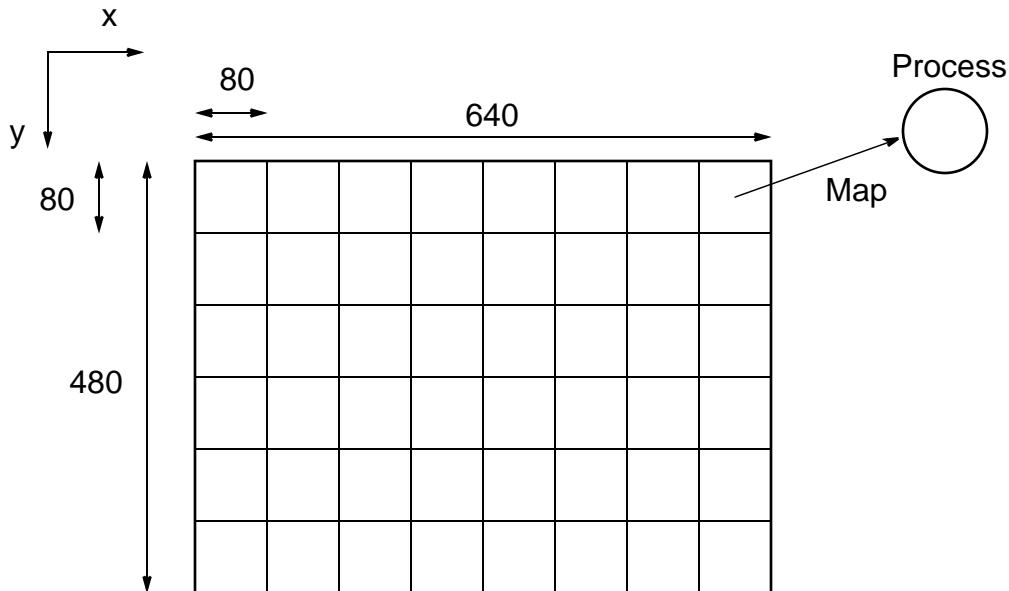
## Embarrassingly Parallel Computation Examples

- Low level image processing
- Mandelbrot set
- Monte Carlo Calculations

## Low level image processing

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

# Partitioning into regions for individual processes.



Square region for each process (can also use strips)

# Some geometrical operations

## Shifting

Object shifted by  $x$  in the  $x$ -dimension and  $y$  in the  $y$ -dimension:

$$x = x + x$$

$$y = y + y$$

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

## Scaling

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

$$x = xS_x$$

$$y = yS_y$$

## Rotation

Object rotated through an angle  $\theta$  about the origin of the coordinate system:

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

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

## 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  $(k + 1)$ th iteration of the complex number  $z = a + bi$  and  $c$  is a complex number giving position of 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

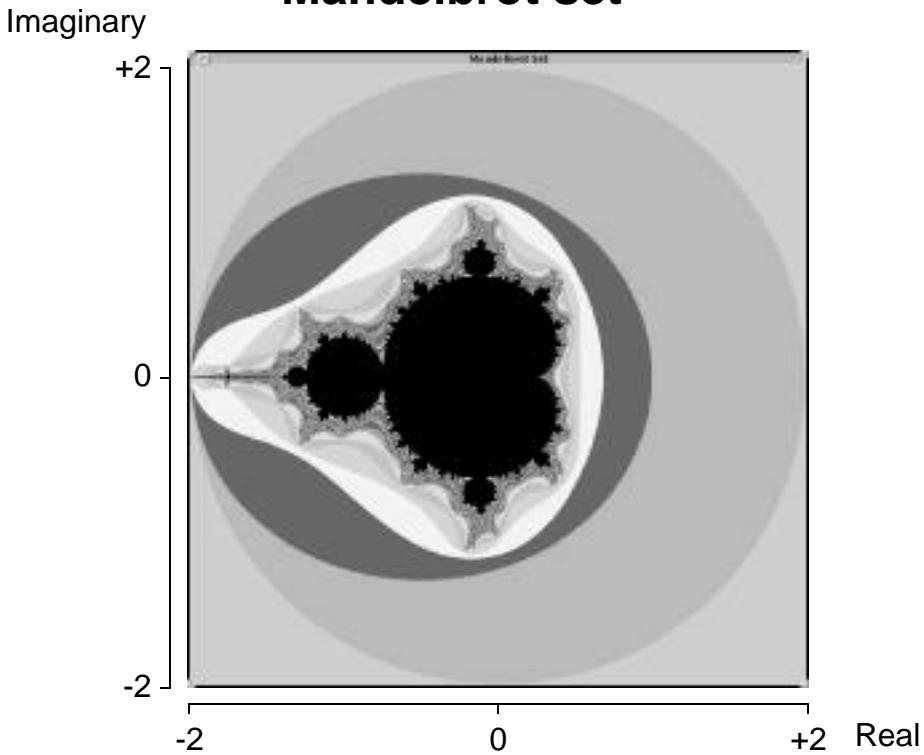
$$z_{\text{length}} = \sqrt{a^2 + b^2}$$

## Sequential routine computing value of one point returning number 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;
}
```

# Mandelbrot set



# Parallelizing Mandelbrot Set Computation

## Static Task Assignment

Simply divide the region in to fixed number of parts, each computed by a separate processor.

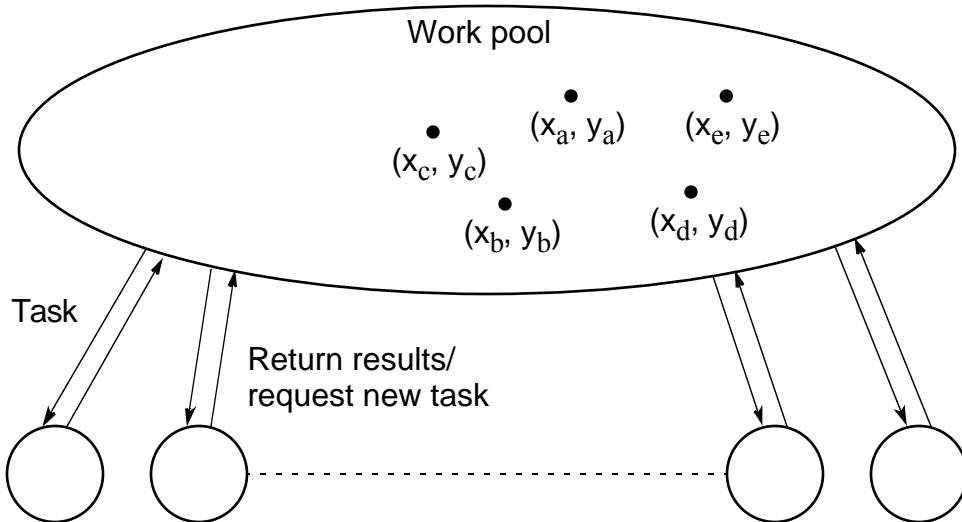
Not very successful because different regions require different numbers of iterations and time.

## Dynamic Task Assignment

Have processor request regions after computing previous regions

# Dynamic Task Assignment

## Work Pool/Processor Farms



## Monte Carlo Methods

Another embarrassingly parallel computation.

Monte Carlo methods use of random selections.

## Example - To calculate

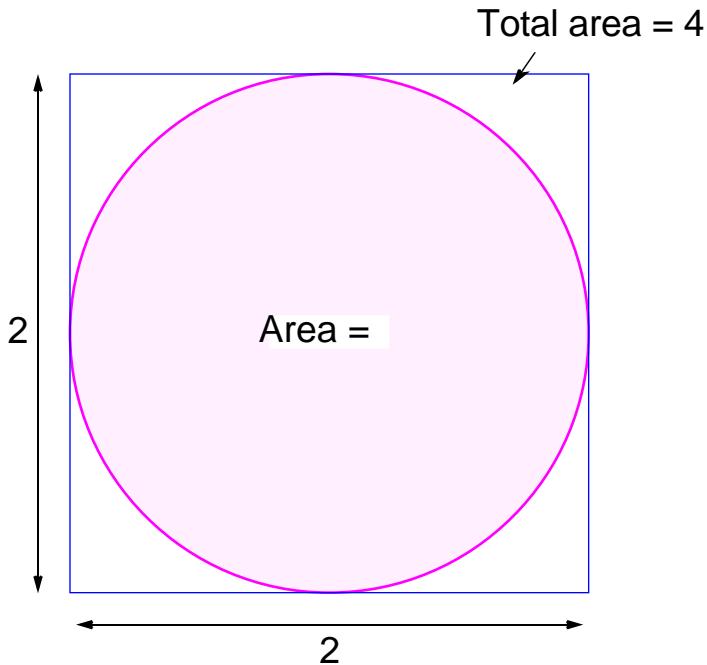
Circle formed within a square, with unit radius so that square has sides  $2 \times 2$ . Ratio of the area of the circle to the square given by

$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{(1)^2}{2 \times 2} = \frac{1}{4}$$

Points within square chosen randomly.

Score kept of how many points happen to lie within circle.

Fraction of points within the circle will be  $/4$ , given a sufficient number of randomly selected samples.



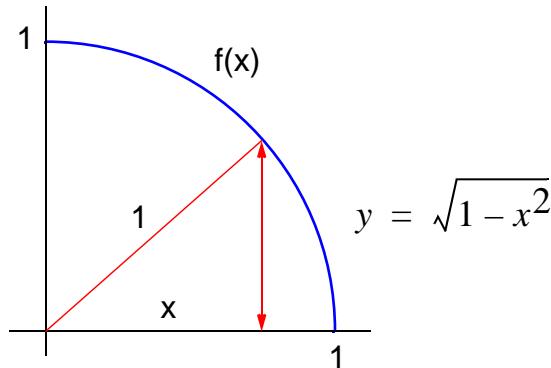
# Computing an Integral

One quadrant of the construction can be described by integral

$$\int_0^1 \sqrt{1 - x^2} dx = \frac{\pi}{4}$$

Random pairs of numbers,  $(x_r, y_r)$  generated, each between 0 and 1.

Counted as in circle if  $y_r \leq \sqrt{1 - x_r^2}$ ; that is,  $y_r^2 + x_r^2 \leq 1$ .



## Alternative (better) Method

Use random values of  $x$  to compute  $f(x)$  and sum values of  $f(x)$ :

$$\text{Area} = \int_{x_1}^{x_2} f(x) dx = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{i=1}^N f(x_i)(x_2 - x_1)$$

where  $x_i$  are randomly generated values of  $x$  between  $x_1$  and  $x_2$ .

Monte Carlo method very useful if the function cannot be integrated numerically (maybe having a large number of variables)

# Example

Computing the integral

$$I = \int_{x_1}^{x_2} (x^2 - 3x) dx$$

## Sequential Code

```
sum = 0;
for (i = 0; i < N; i++) { /* N random samples */
    xr = rand_v(x1, x2); /* generate next random value */
    sum = sum + xr * xr - 3 * xr; /* compute f(xr) */
}
area = (sum / N) * (x2 - x1);
```

Routine `randv(x1, x2)` returns a pseudorandom number between `x1` and `x2`.

*For parallelizing Monte Carlo code, must address best way to generate random numbers in parallel - see textbook*

# Intentionally blank

## Chapter 4

# Partitioning and Divide-and-Conquer Strategies

## Partitioning

Partitioning simply divides the problem into parts.

## Divide and Conquer

Characterized by dividing problem into subproblems of same form as larger problem. Further divisions into still smaller sub-problems, usually done by recursion.

Recursive divide and conquer amenable to parallelization because separate processes can be used for divided parts.

Also usually data is naturally localized.

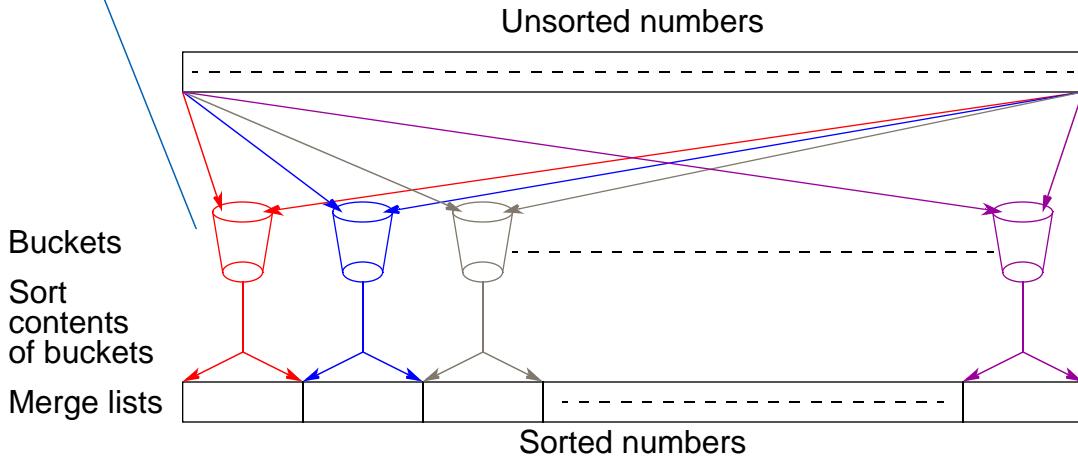
## Partitioning/Divide and Conquer Examples

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
- $N$ -body problem

# Bucket sort

One “bucket” assigned to hold numbers that fall within each region.  
Numbers in each bucket sorted using a sequential sorting algorithm.



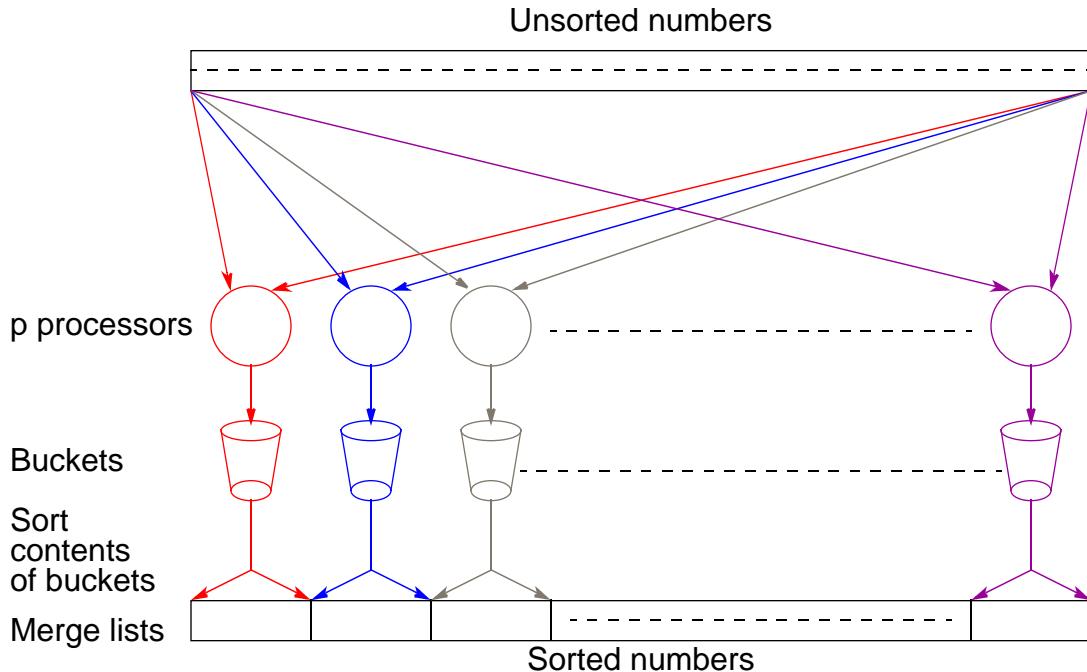
Sequential sorting time complexity:  $O(n \log(n/m))$ .

Works well if the original numbers uniformly distributed across a known interval, say 0 to  $a - 1$ .

# Parallel version of bucket sort

## Simple approach

Assign one processor for each bucket.



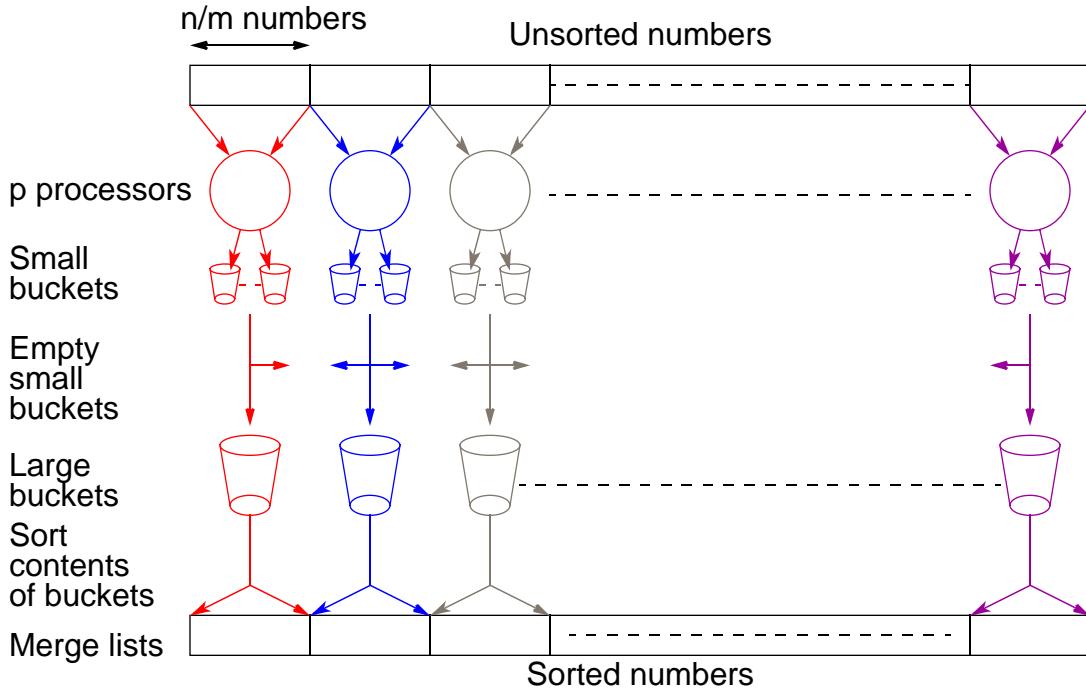
## Further Parallelization

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

Each processor maintains  $p$  “small” buckets and separates 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  $j$ ).

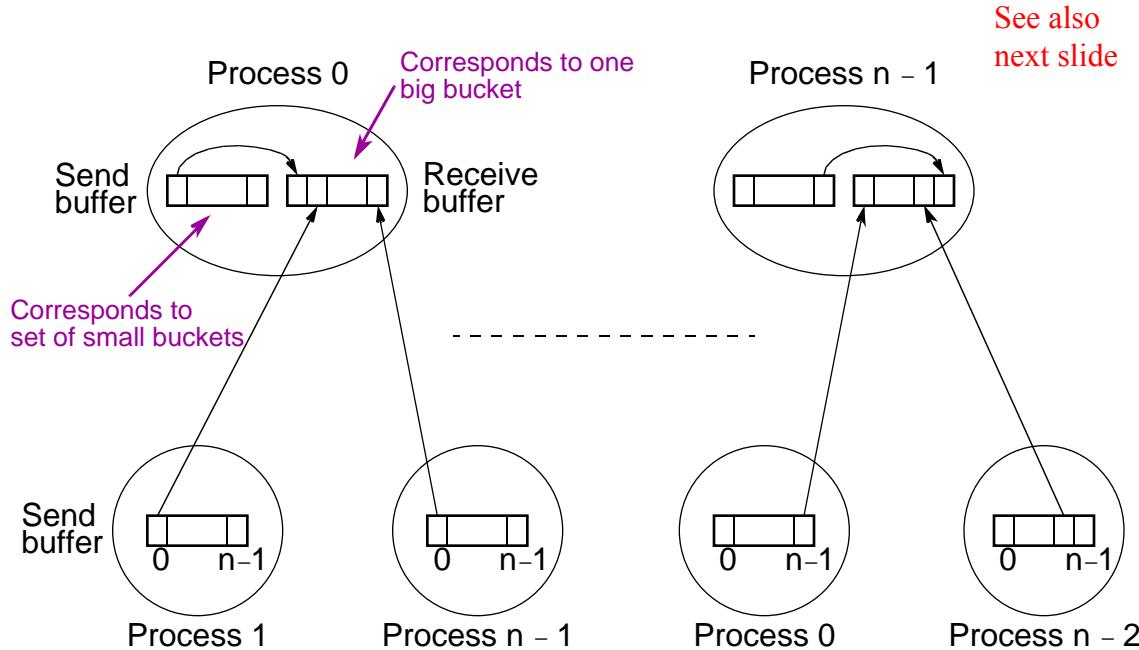
# Another parallel version of bucket sort



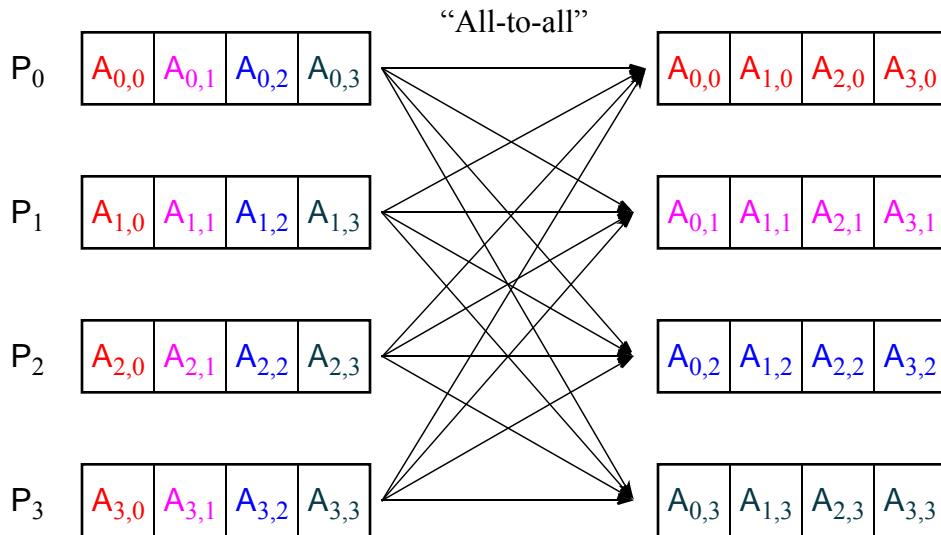
Introduces new message-passing operation - all-to-all broadcast.

# “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:  
Tranposes a matrix.

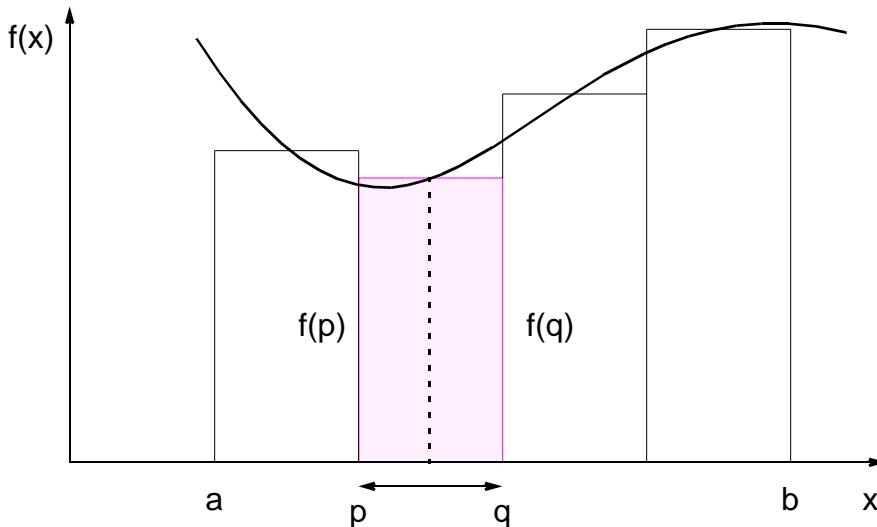


### Effect of “all-to-all” on an array

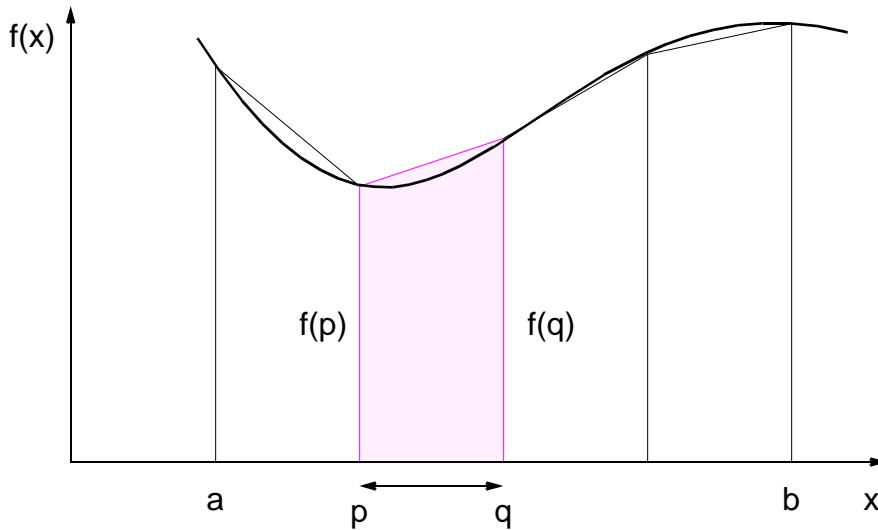
# Numerical integration using rectangles.

Each region calculated using an approximation given by rectangles:

Aligning the rectangles:



# Numerical integration using trapezoidal method

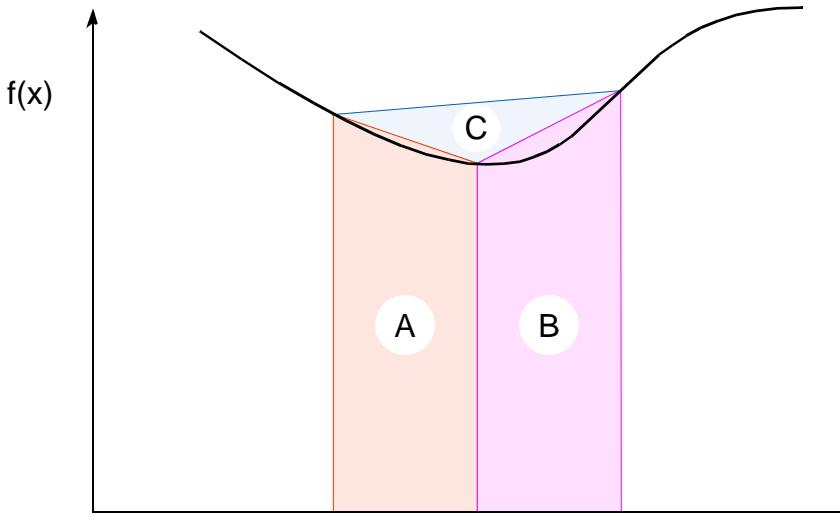


May not be better!

# Adaptive Quadrature

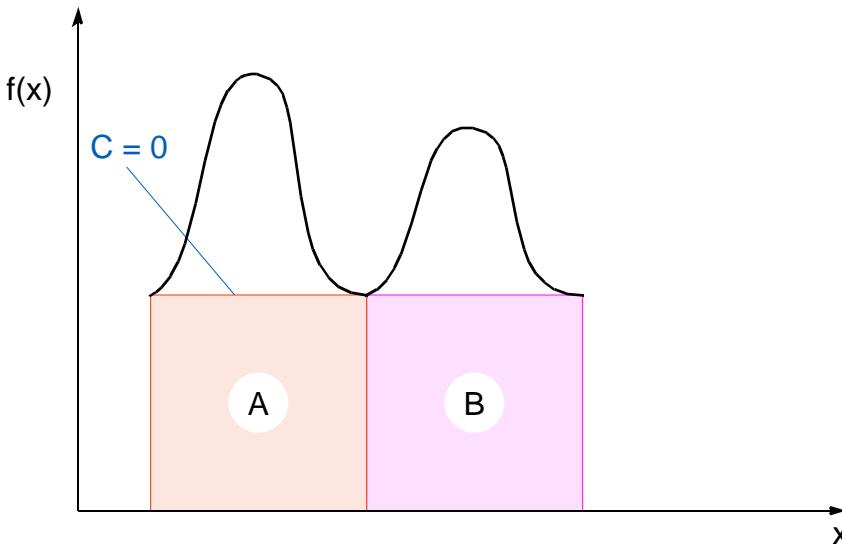
Solution adapts to shape of curve. Use three areas,  $A$ ,  $B$ , and  $C$ .

Computation terminated when largest of  $A$  and  $B$  sufficiently close to sum of remain two areas .



## Adaptive quadrature with false termination.

Some care might be needed in choosing when to terminate.



Might cause us to terminate early, as two large regions are the same (i.e.,  $C = 0$ ).

# **Simple program to compute**

## **Using C++ MPI routines**

```
*****  
pi_calc.cpp calculates value of pi and compares with actual value (to 25  
digits) of pi to give error. Integrates function f(x)=4/(1+x^2).  
July 6, 2001 K. Spry CSCI3145  
*****  
  
#include <math.h>           //include files  
#include <iostream.h>  
#include "mpi.h"  
void printit();           //function prototypes  
int main(int argc, char *argv[])  
{  
    double actual_pi = 3.141592653589793238462643;    //for comparison later  
    int n, rank, num_proc, i;  
    double temp_pi, calc_pi, int_size, part_sum, x;  
    char response = 'y', resp1 = 'y';  
    MPI::Init(argc, argv); //initiate MPI  
    num_proc = MPI::COMM_WORLD.Get_size();  
    rank = MPI::COMM_WORLD.Get_rank();  
    if (rank == 0) printit();          /* I am root node, print out welcome */  
    while (response == 'y') {  
        if (resp1 == 'y') {  
            if (rank == 0) {           /*I am root node*/  
                cout << "_____ " << endl;  
                cout << "\nEnter the number of intervals: (0 will exit)" << endl;  
                cin >> n;  
            }  
        } else n = 0;  
    }  
}
```

```
MPI::COMM_WORLD.Bcast(&n, 1, MPI::INT, 0);      //broadcast n
if (n==0) break;                                //check for quit condition
else {
    int_size = 1.0 / (double) n;//calcs interval size
    part_sum = 0.0;
    for (i = rank + 1; i <= n; i += num_proc) { //calcs partial sums
        x = int_size * ((double)i - 0.5);
        part_sum += (4.0 / (1.0 + x*x));
    }
    temp_pi = int_size * part_sum;
    //collects all partial sums computes pi
MPI::COMM_WORLD.Reduce(&temp_pi,&calc_pi, 1, MPI::DOUBLE, MPI::SUM, 0);
```

```
if (rank == 0) {    /*I am server*/
    cout << "pi is approximately " << calc_pi
    << ". Error is " << fabs(calc_pi - actual_pi)
    << endl
    << "____"
    << endl;
}
} //end else
if (rank == 0) { /*I am root node*/
    cout << "\nCompute with new intervals? (y/n)" << endl; cin >> respl;
}
}//end while
MPI::Finalize(); //terminate MPI
return 0;
} //end main
```

```
//functions
void printit()
{
    cout << "\n*****" << endl
        << "Welcome to the pi calculator!" << endl
        << "Programmer: K. Spry" << endl
        << "You set the number of divisions \nfor estimating the integral:
\n\tf(x)=4/(1+x^2)"
        << endl
        << "*****" << endl;
}//end printit
```

## Gravitational N-Body Problem

Finding positions and movements of bodies in space subject to gravitational forces from other bodies, using Newtonian laws of physics.

# Gravitational N-Body Problem Equations

Gravitational force between two bodies of masses  $m_a$  and  $m_b$  is:

$$F = \frac{Gm_a m_b}{r^2}$$

$G$  is the gravitational constant and  $r$  the distance between the bodies. Subject to forces, body accelerates according to Newton's 2nd law:

$$F = ma$$

$m$  is mass of the body,  $F$  is force it experiences, and  $a$  the resultant acceleration.

## Details

Let the time interval be  $t$ . For a body of mass  $m$ , the force is:

$$F = \frac{m(v^{t+1} - v^t)}{t}$$

New velocity is:

$$v^{t+1} = v^t + \frac{F}{m} t$$

where  $v^{t+1}$  is the velocity at time  $t + 1$  and  $v^t$  is the velocity at time  $t$ .

Over time interval  $t$ , position changes by

$$x^{t+1} - x^t = v^t t$$

where  $x^t$  is its position at time  $t$ .

Once bodies move to new positions, forces change. Computation has to be repeated.

## Sequential Code

Overall gravitational  $N$ -body computation can be described by:

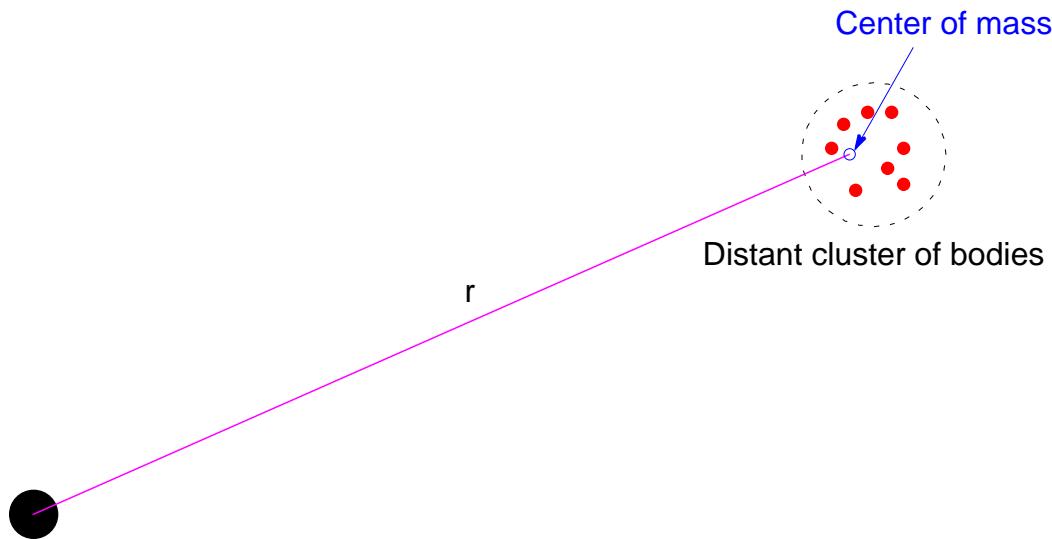
```
for (t = 0; t < tmax; t++)      /* for each time period */
    for (i = 0; i < N; i++) {    /* for each body */
        F = Force_routine(i);    /* compute force on ith body */
        v[i]_new = v[i] + F * dt / m; /* compute new velocity */
        x[i]_new = x[i] + v[i]_new * dt; /* and new position */
    }
    for (i = 0; i < nmax; i++) { /* for each body */
        x[i] = x[i]_new;           /* update velocity & position */
        v[i] = v[i]_new;
    }
```

## Parallel Code

The sequential algorithm is an  $O(N^2)$  algorithm (for one iteration) as each of the  $N$  bodies is influenced by each of the other  $N - 1$  bodies.

Not feasible to use this direct algorithm for most interesting  $N$ -body problems where  $N$  is very large.

Time complexity can be reduced using observation that a cluster of distant bodies can be approximated as a single distant body of the total mass of the cluster sited at the center of mass of the cluster:



## Barnes-Hut Algorithm

Start with whole space in which one cube contains the bodies (or particles).

- First, this cube is divided into eight subcubes.
- If a subcube contains no particles, the subcube is deleted from further consideration.
- If a subcube contains one body, this subcube retained
- If a subcube contains more than one body, it is recursively divided until every subcube contains one body.

Creates an *octtree* - a tree with up to eight edges from each node.

The leaves represent cells each containing one body.

After the tree has been constructed, the total mass and center of mass of the subcube is stored at each node.

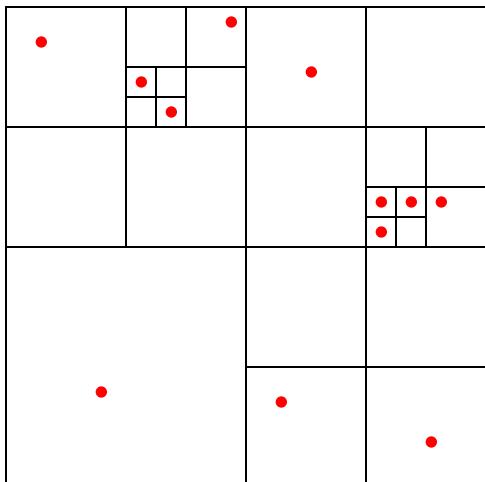
Force on each body obtained by traversing tree starting at root, stopping at a node when the clustering approximation can be used, e.g. when:

$$r \quad d$$

where  $r$  is a constant typically 1.0 or less.

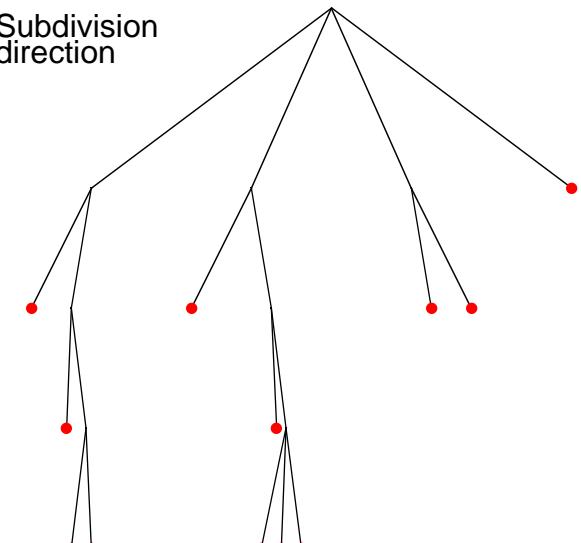
Constructing tree requires a time of  $O(n \log n)$ , and so does computing all the forces, so that the overall time complexity of the method is  $O(n \log n)$ .

# Recursive division of two-dimensional space



Particles

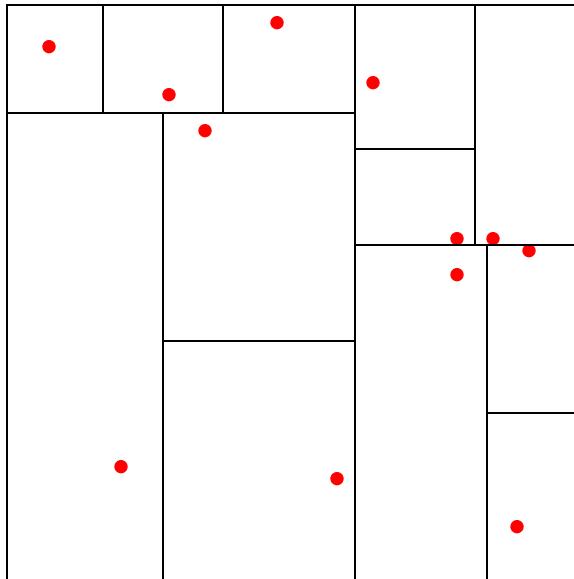
Subdivision direction



Partial quadtree

## Orthogonal Recursive Bisection

(For 2-dimensional area) First, a vertical line found that divides area into two areas each with equal number of bodies. For each area, a horizontal line found that divides it into two areas each with equal number of bodies. Repeated as required.



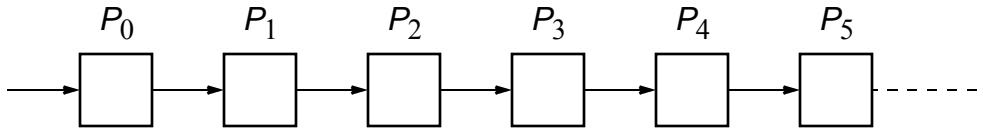
# Intentionally blank

## Chapter 5

# Pipelined Computations

# Pipelined Computations

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



## Example

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

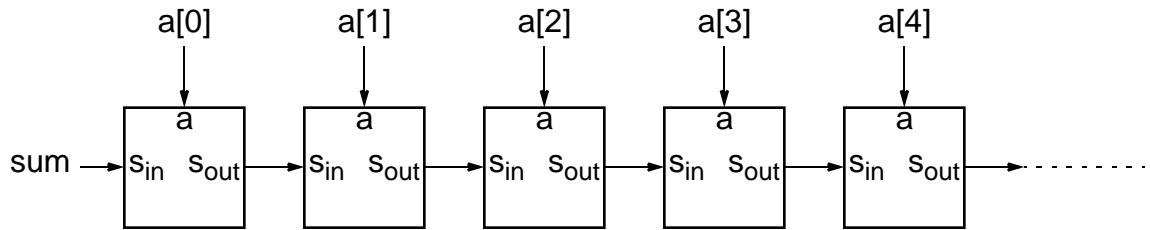
```
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

The loop could be “unfolded” to yield

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

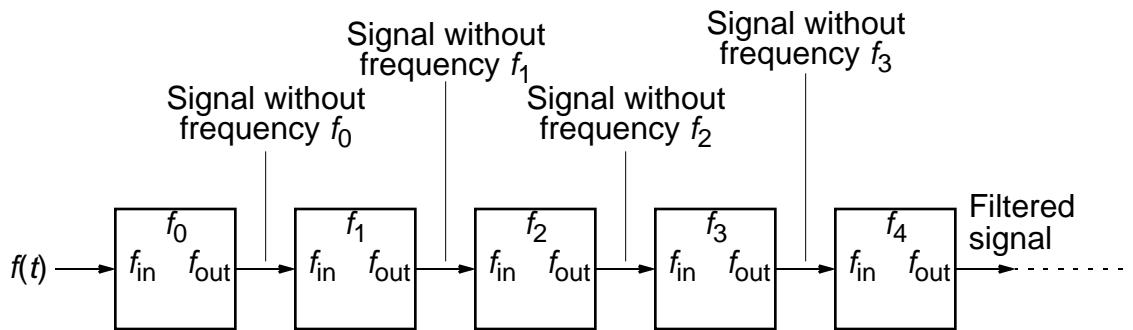
.

## Pipeline for an unfolded loop



## Another Example

**Frequency filter** - Objective to remove specific frequencies ( $f_0, f_1, f_2, f_3$ , etc.) from a digitized signal,  $f(t)$ . Signal enters pipeline from left:

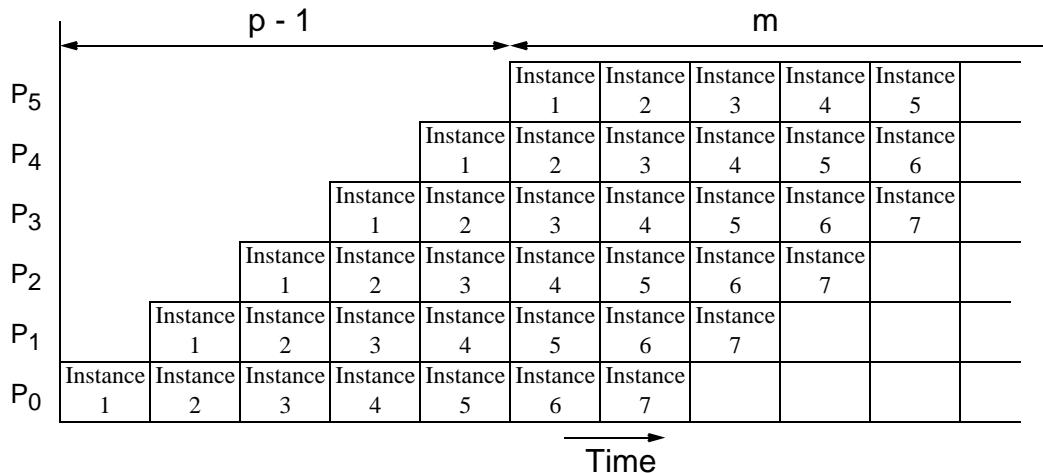


## Where pipelining can be used to good effect

Assuming problem can be divided into a series of sequential tasks, pipelined approach can provide increased execution speed under the following three types of computations:

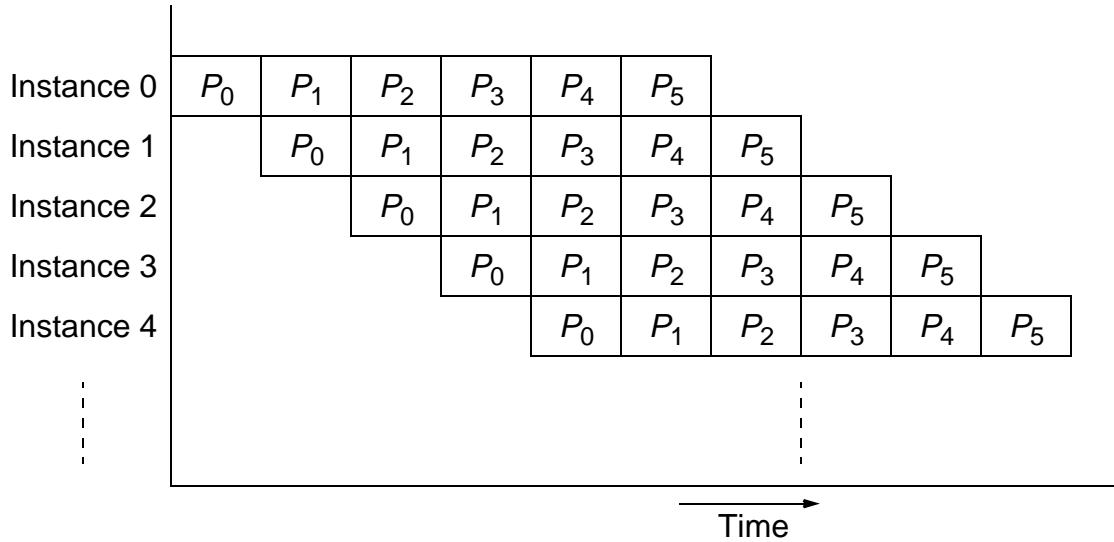
1. If more than one instance of the complete problem is to be executed
2. If a series of data items must be processed, each requiring multiple operations
3. If information to start the next process can be passed forward before the process has completed all its internal operations

# “Type 1” Pipeline Space-Time Diagram

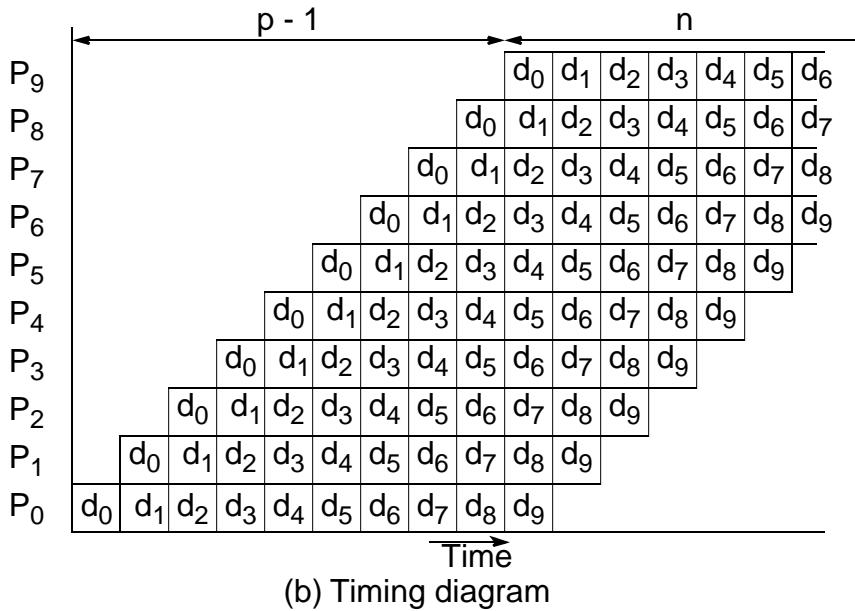
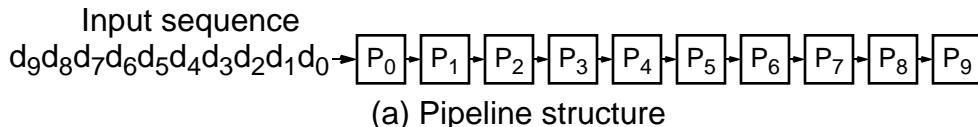


Execution time =  $m + p - 1$  cycles for a  $p$ -stage pipeline and  $m$  instances.

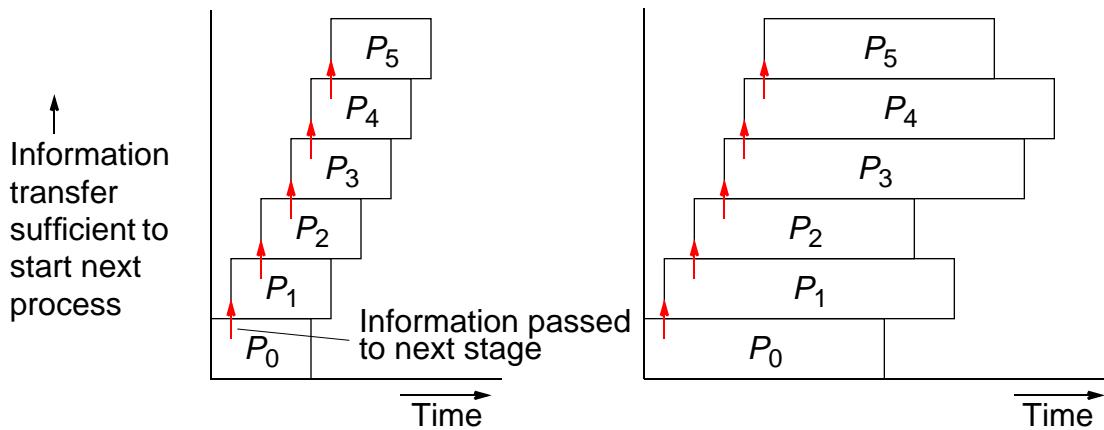
## Alternative space-time diagram



# “Type 2” Pipeline Space-Time Diagram

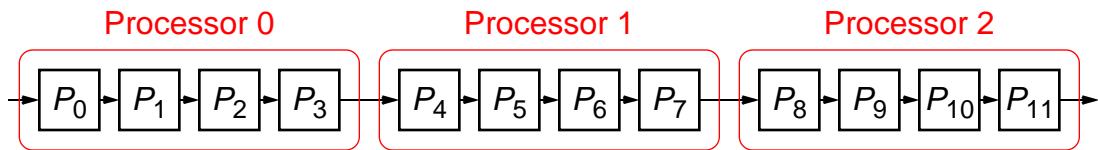


# “Type 3” Pipeline Space-Time Diagram



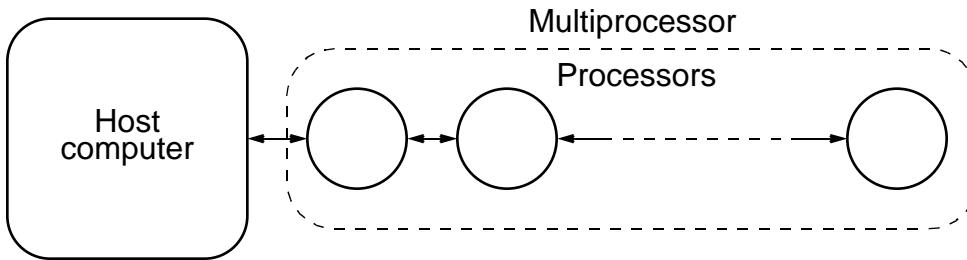
Pipeline processing where information passes to next stage before end of process.

If the number of stages is larger than the number of processors in any pipeline, a group of stages can be assigned to each processor:



# Computing Platform for Pipelined Applications

Multiprocessor system with a line configuration.



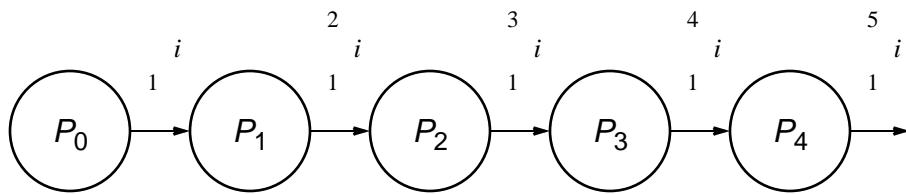
Strictly speaking pipeline may not be the best structure for a cluster - however a cluster with switched direct connections, as most have, can support simultaneous message passing.

# Example Pipelined Solutions

(Examples of each type of computation)

# Pipeline Program Examples

## Adding Numbers



Type 1 pipeline computation

Basic code for process  $P_i$ :

```
recv(&accumulation, Pi-1);  
accumulation = accumulation + number;  
send(&accumulation, Pi+1);
```

except for the first process,  $P_0$ , which is

```
send(&number, P1);
```

and the last process,  $P_{n-1}$ , which is

```
recv(&number, Pn-2);  
accumulation = accumulation + number;
```

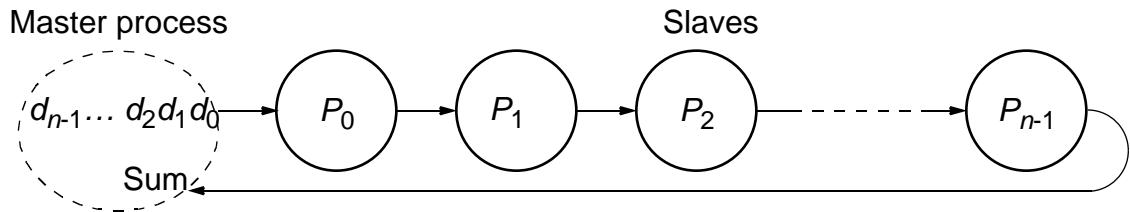
## SPMD program

```
if (process > 0) {  
    recv(&accumulation, p_1);  
    accumulation = accumulation + number;  
}  
if (process < n-1) send(&accumulation, i, p_1);
```

The final result is in the last process.

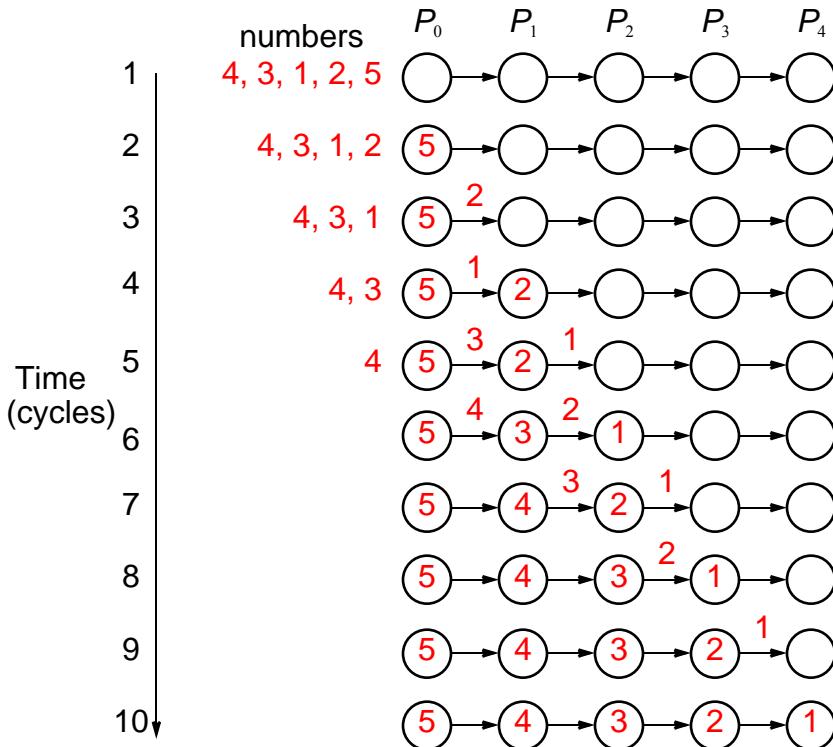
Instead of addition, other arithmetic operations could be done.

## Pipelined addition numbers with a master process and ring configuration

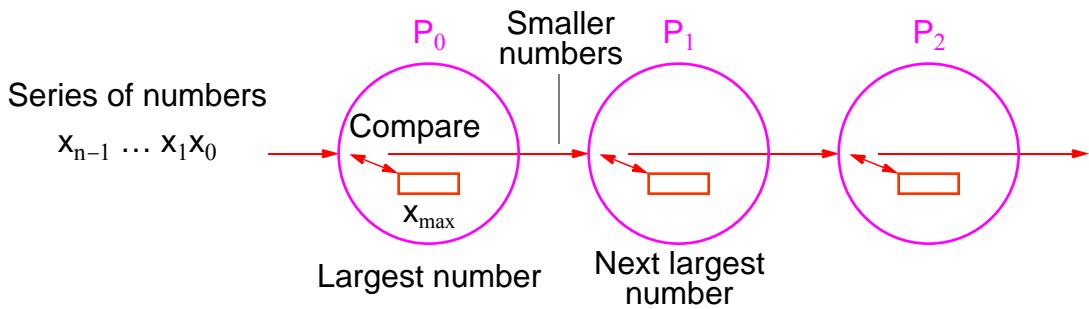


# Sorting Numbers

A parallel version of *insertion sort*.



# Pipeline for sorting using insertion sort



Type 2 pipeline computation

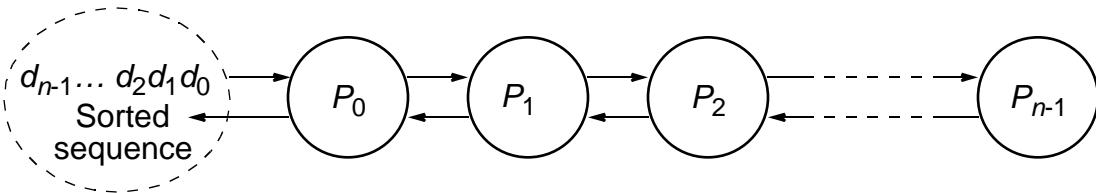
The basic algorithm for process  $P_i$  is

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

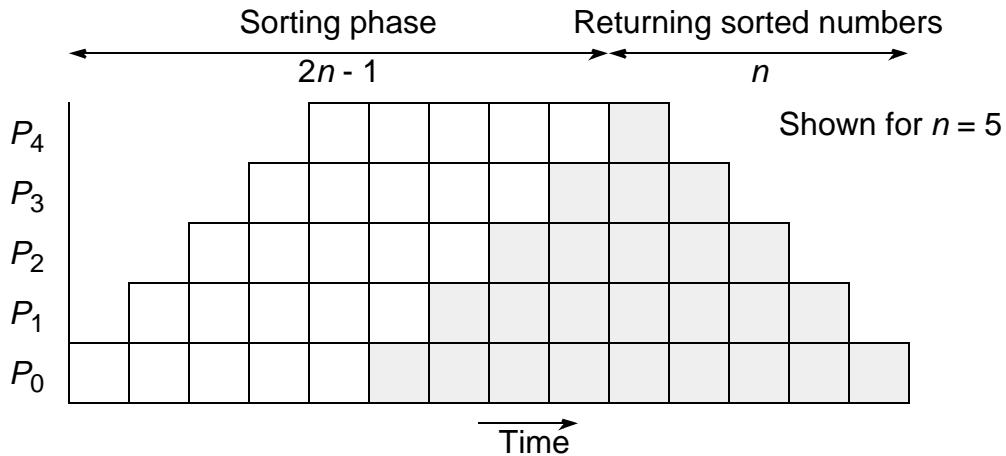
With  $n$  numbers, how many the  $i$ th process is to accept is known; it is given by  $n - i$ . How many to pass onward is also known; it is given by  $n - i - 1$  since one of the numbers received is not passed onward. Hence, a simple loop could be used.

## Insertion sort with results returned to the master process using a bidirectional line configuration

Master process



## Insertion sort with results returned

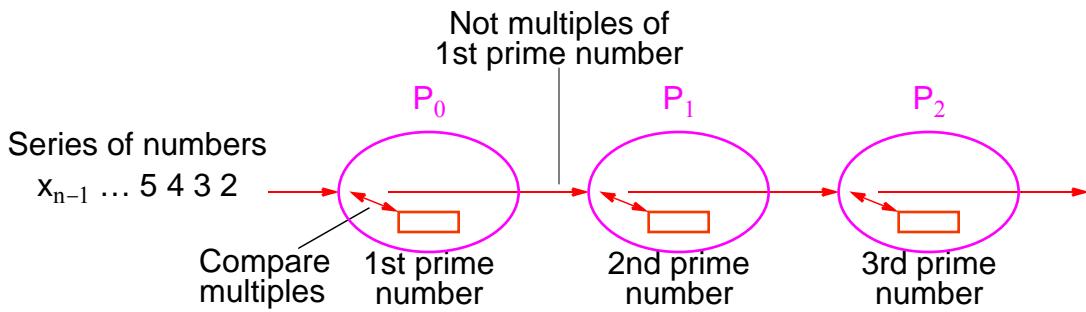


# Prime Number Generation

## Sieve of Eratosthenes

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

# Pipeline for Prime Number Generation



Type 2 pipeline computation

The code for a process,  $P_i$ , could be based upon

```
recv(&x, Pi-1);
/* repeat following for each number */
recv(&number, Pi-1);
if ((number % x) != 0) send(&number, iP1);
```

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

```
recv(&x, Pi-1);
for (i = 0; i < n; i++) {
    recv(&number, Pi-1);
    if (number == terminator) break;
    if (number % x) != 0) send(&number, iP1);
}
```

# Solving a System of Linear Equations

## Upper-triangular form

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \dots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

.

.

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 = b_1$$

$$a_{0,0}x_0 = b_0$$

where the  $a$ 's and  $b$ 's are constants and the  $x$ 's are unknowns to be found.

## Back Substitution

First, the unknown  $x_0$  is found from the last equation; i.e.,

$$x_0 = \frac{b_0}{a_{0,0}}$$

Value obtained for  $x_0$  substituted into next equation to obtain  $x_1$ ; i.e.,

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

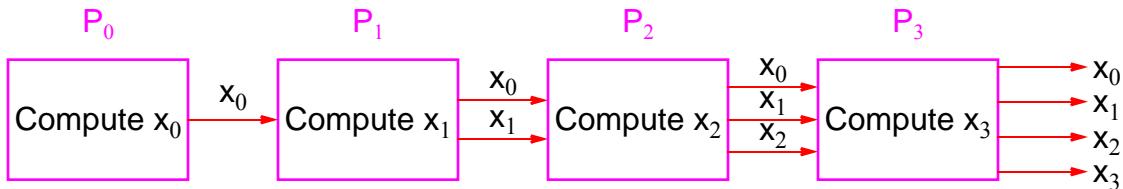
Values obtained for  $x_1$  and  $x_0$  substituted into next equation to obtain  $x_2$ :

$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}$$

and so on until all the unknowns are found.

# Pipeline Solution

First pipeline stage computes  $x_0$  and passes  $x_0$  onto the second stage, which computes  $x_1$  from  $x_0$  and passes both  $x_0$  and  $x_1$  onto the next stage, which computes  $x_2$  from  $x_0$  and  $x_1$ , and so on.



Type 3 pipeline computation

The  $i$ th process ( $0 < i < n$ ) receives the values  $x_0, x_1, x_2, \dots, x_{i-1}$  and computes  $x_i$  from the equation:

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j} x_j}{a_{i,i}}$$

## Sequential Code

Given the constants  $a_{i,j}$  and  $b_k$  stored in arrays  $\mathbf{a}[][]$  and  $\mathbf{b}[]$ , respectively, and the values for unknowns to be stored in an array,  $\mathbf{x}[]$ , the sequential code could be

```
x[0] = b[0]/a[0][0]; /* computed separately */
for (i = 1; i < n; i++) /* for remaining
unknowns */
    sum = 0;
    for (j = 0; j < i; j++)
        sum = sum + a[i][j]*x[j];
    x[i] = (b[i] - sum)/a[i][i];
}
```

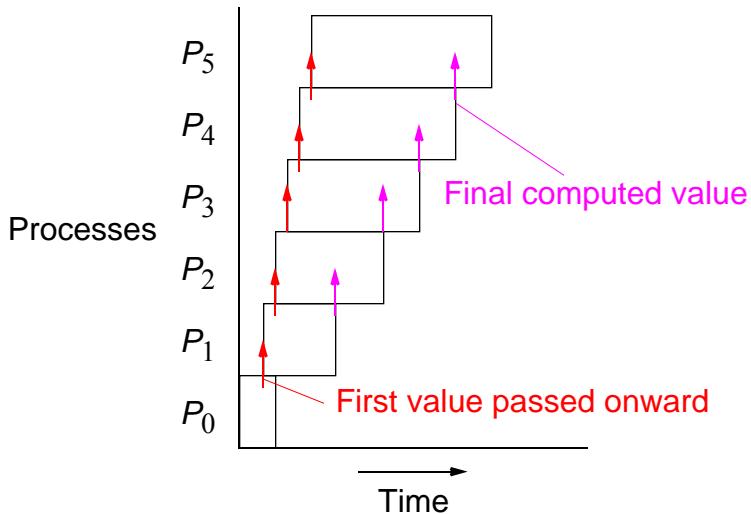
## Parallel Code

Pseudocode of process  $P_i$  ( $1 < i < n$ ) of could be

```
for (j = 0; j < i; j++) {
    recv(&x[j], Pi-1);
    send(&x[j], Pi+1);
}
sum = 0;
for (j = 0; j < i; j++)
    sum = sum + a[i][j]*x[j];
x[i] = (b[i] - sum)/a[i][i];
send(&x[i], Pi+1);
```

Now additional computations after receiving and resending values.

# Pipeline processing using back substitution



# Algorithms and Applications

# Evaluating Algorithms

## Cost

*Processor-time* product or *cost* (or *work*) of a computation can be defined as

Cost = execution time  $\times$  total number of processors used

Cost of a sequential computation simply its execution time,  $t_s$ .

Cost of a parallel computation is  $t_p \times n$ . Parallel execution time,  $t_p$ , is given by  $t_s/S(n)$ .

Hence, the cost of a parallel computation given by

$$\text{Cost} = \frac{t_s n}{S(n)} = \frac{t_s}{E}$$

## Cost-Optimal Parallel Algorithm

One in which the cost to solve a problem on a multiprocessor is proportional to the cost (i.e., execution time) on a single processor system.

Can be used to compare algorithms.

## Parallel Algorithm Time Complexity

Can derive the time complexity of a parallel algorithm in a similar manner as for a sequential algorithm by counting the steps in the algorithm (worst case) .

Following from the definition of cost-optimal algorithm

$$\text{(Cost) optimal parallel time complexity} = \frac{\text{sequential time complexity}}{\text{number of processors}}$$

But this does not take into account communication overhead. In textbook, calculated computation and communication separately.

Areas done in textbook:

- Sorting Algorithms
- Numerical Algorithms
- Image Processing
- Searching and Optimization

## Chapter 9

# Sorting Algorithms

- rearranging a list of numbers into increasing (strictly non-decreasing) order.

## Potential Speedup

$(n \log n)$  optimal for any sequential sorting algorithm without using special properties of the numbers.

Best we can expect based upon a sequential sorting algorithm using  $n$  processors is

$$\text{Optimal parallel time complexity} = \frac{O(n \log n)}{n} = O(\log n)$$

Has been obtained but the constant hidden in the order notation extremely large.

Also an algorithm exists for an  $n$ -processor hypercube using random operations.

**But, in general, a realistic  $(\log n)$  algorithm with  $n$  processors not be easy to achieve.**

# Sorting Algorithms Reviewed

- Rank sort  
(to show that an non-optimal sequential algorithm may in fact be a good parallel algorithm)
- Compare and exchange operations  
(to show the effect of duplicated operations can lead to erroneous results)
- Bubble sort and odd-even transposition sort
- Two dimensional sorting - Shearsort (with use of transposition)
- Parallel Mergesort
- Parallel Quicksort
- Odd-even Mergesort
- Bitonic Mergesort

## Rank Sort

The number of numbers that are smaller than each selected number is counted. This count provides the position of selected number in sorted list; that is, its “rank.”

First  $a[0]$  is read and compared with each of the other numbers,  $a[1] \dots a[n-1]$ , recording the number of numbers less than  $a[0]$ . Suppose this number is  $x$ . This is the index of the location in the final sorted list. The number  $a[0]$  is copied into the final sorted list  $b[0] \dots b[n-1]$ , at location  $b[x]$ . Actions repeated with the other numbers.

Overall sequential sorting time complexity of  $(n^2)$  (not exactly a good sequential sorting algorithm!).

## Sequential Code

```
for (i = 0; i < n; i++) {                  /* for each number */
    x = 0;
    for (j = 0; j < n; j++) /* count number less than it */
        if (a[i] > a[j]) x++;
    b[x] = a[i];      /* copy number into correct place */
}
```

This code will fail if duplicates exist in the sequence of numbers.

## Parallel Code Using $n$ Processors

One processor allocated to each number. Finds final index in  $(n)$  steps. With all processors operating in parallel, parallel time complexity  $(n)$ .

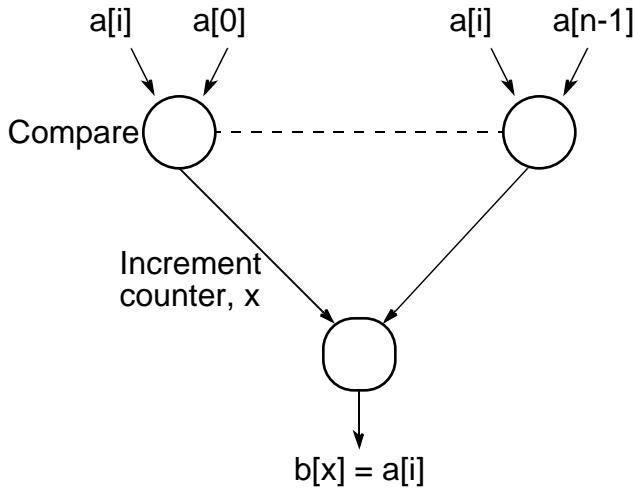
In **forall** notation, the code would look like

```
forall (i = 0; i < n; i++) /* for each no in parallel*/
    x = 0;
    for (j = 0; j < n; j++) /* count number less than it */
        if (a[i] > a[j]) x++;
    b[x] = a[i];           /* copy no into correct place */
}
```

Parallel time complexity,  $(n)$ , better than any sequential sorting algorithm. Can do even better if we have more processors.

# Using $n^2$ Processors

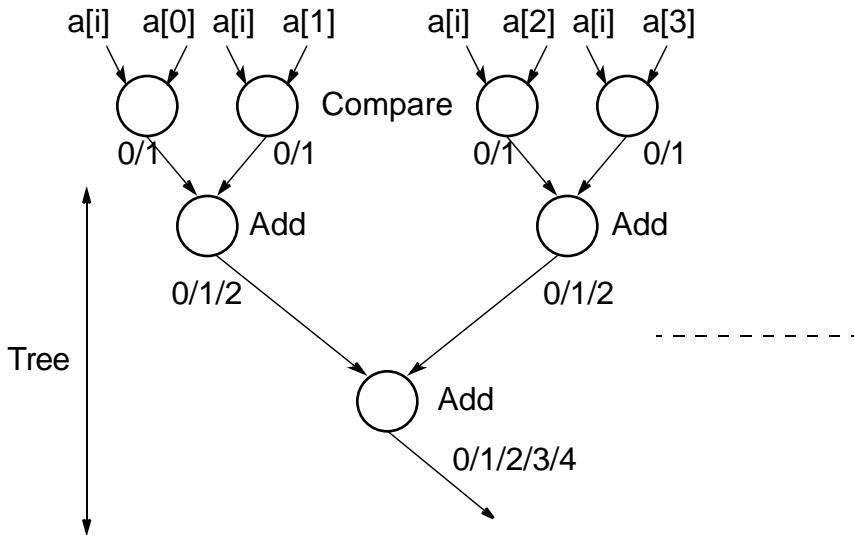
Comparing one number with the other numbers in list using multiple processors:



$n - 1$  processors used to find rank of one number. With  $n$  numbers,  $(n - 1)n$  processors or (almost)  $n^2$  processors needed. Incrementing the counter done sequentially and requires maximum of  $n$  steps.

# Reduction in Number of Steps

Tree to reduce number of steps involved in incrementing counter:



$(\log n)$  algorithm with  $n^2$  processors.  
Processor efficiency relatively low.

## Parallel Rank Sort Conclusions

Easy to do as each number can be considered in isolation.

Rank sort can sort in:

( $n$ ) with  $n$  processors

or

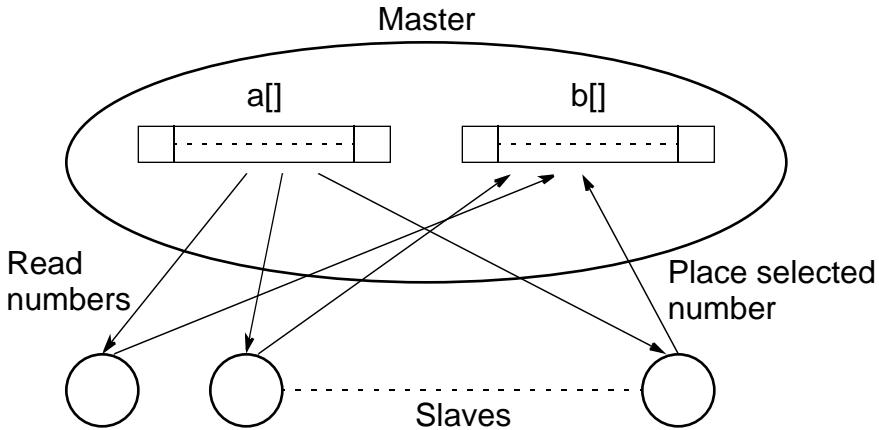
( $\log n$ ) using  $n^2$  processors.

In practical applications, using  $n^2$  processors prohibitive.

Theoretically possible to reduce time complexity to (1) by considering all increment operations as happening in parallel since they are independent of each other.

# Message Passing Parallel Rank Sort

## Master-Slave Approach



Requires shared access to list of numbers. Master process responds to request for numbers from slaves. Algorithm better for shared memory

# Compare-and-Exchange Sorting Algorithms

## Compare and Exchange

Form the basis of several, if not most, classical sequential sorting algorithms.

Two numbers, say  $A$  and  $B$ , are compared. If  $A > B$ ,  $A$  and  $B$  are exchanged, i.e.:

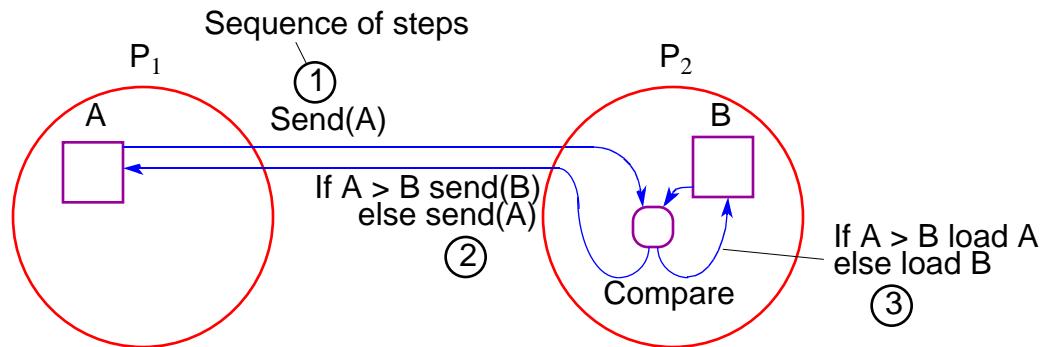
```
if (A > B) {  
    temp = A;  
    A = B;  
    B = temp;  
}
```

# Message-Passing Compare and Exchange

## Version 1

$P_1$  sends  $A$  to  $P_2$ , which compares  $A$  and  $B$  and sends back  $B$  to  $P_1$

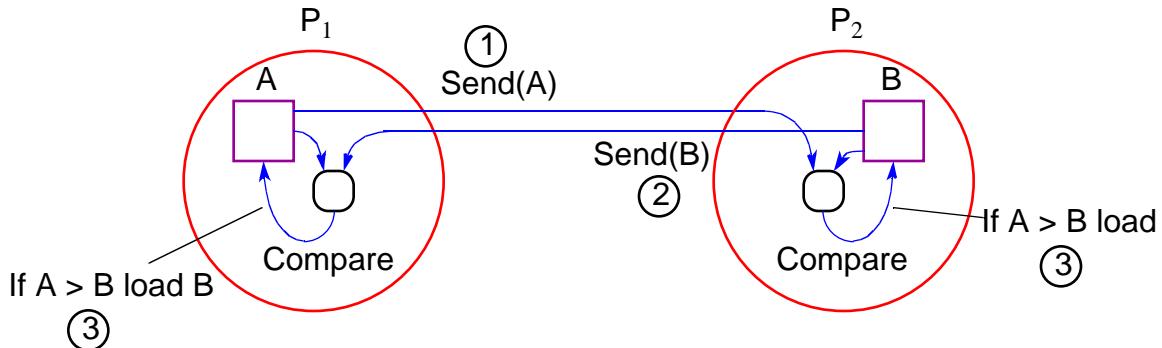
if  $A$  is larger than  $B$  (otherwise it sends back  $A$  to  $P_1$ ):



# Alternative Message Passing Method

## Version 2

For  $P_1$  to send  $A$  to  $P_2$  and  $P_2$  to send  $B$  to  $P_1$ . Then both processes perform compare operations.  $P_1$  keeps the larger of  $A$  and  $B$  and  $P_2$  keeps the smaller of  $A$  and  $B$ :



## Note on Precision of Duplicated Computations

Previous code assumes that the `if` condition, `A > B`, will return the same Boolean answer in both processors.

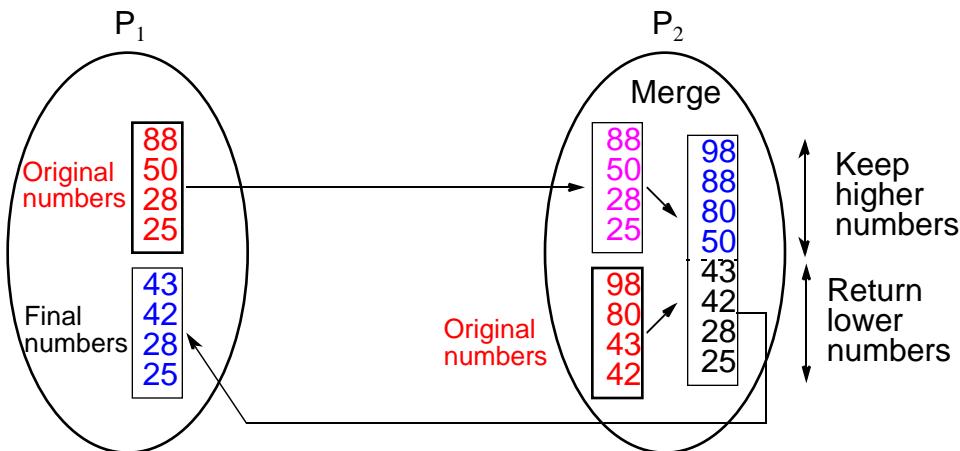
Different processors operating at different precision could conceivably produce **different answers** if real numbers are being compared.

This situation applies to anywhere computations are duplicated in different processors to reduce message passing, or to make the code SPMD.

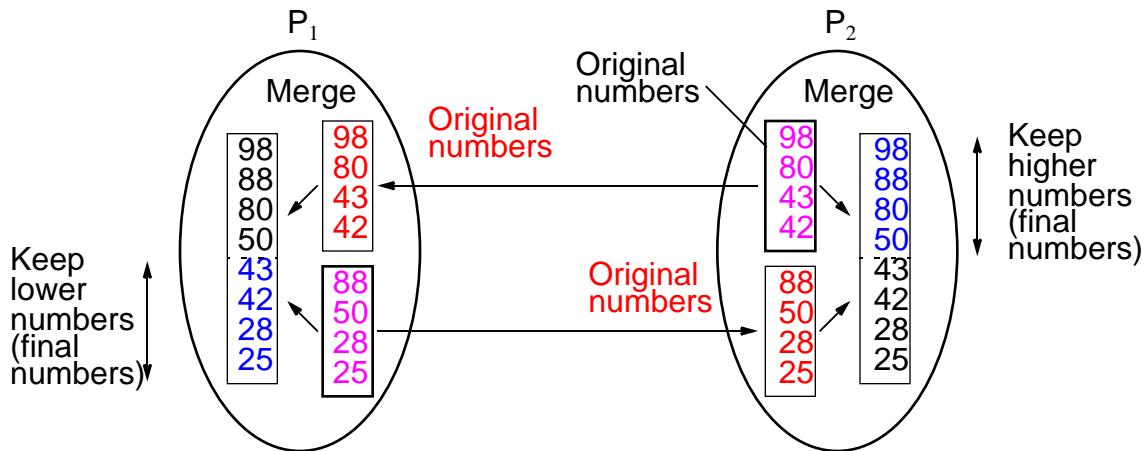
# Data Partitioning

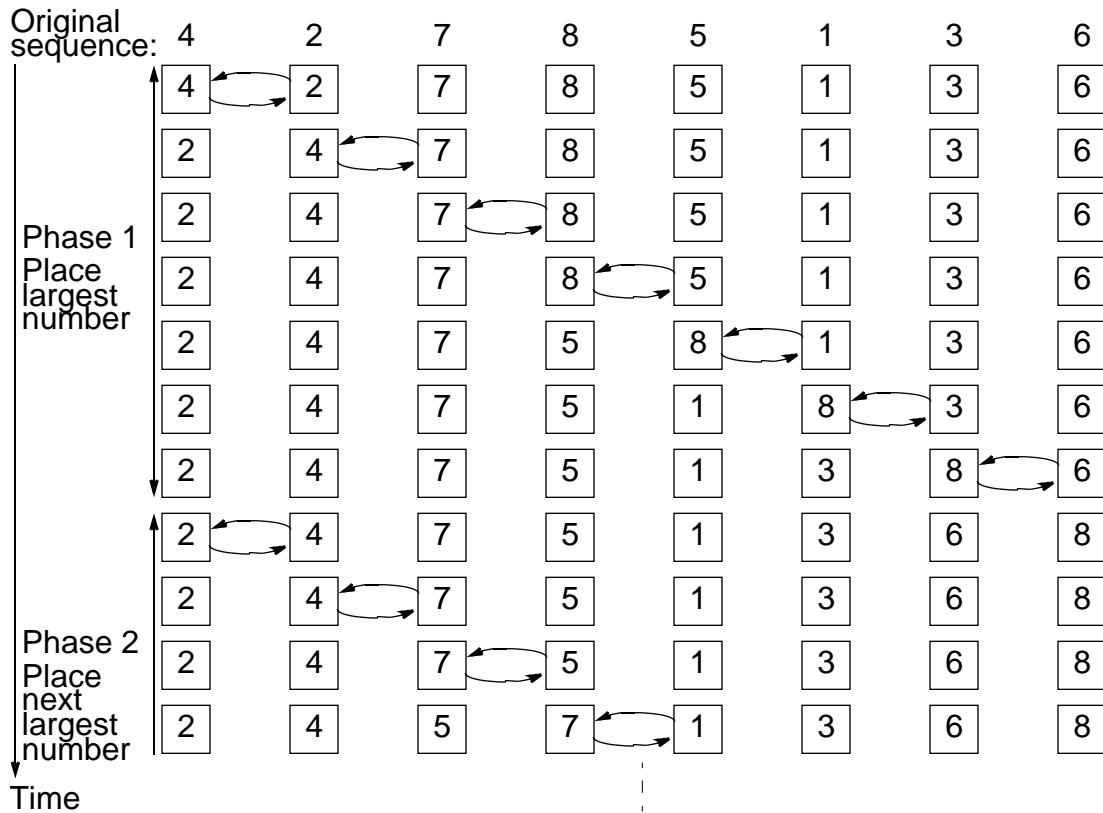
## (Version 1)

$p$  processors and  $n$  numbers.  $n/p$  numbers assigned to each processor:



# Merging Two Sublists — Version 2





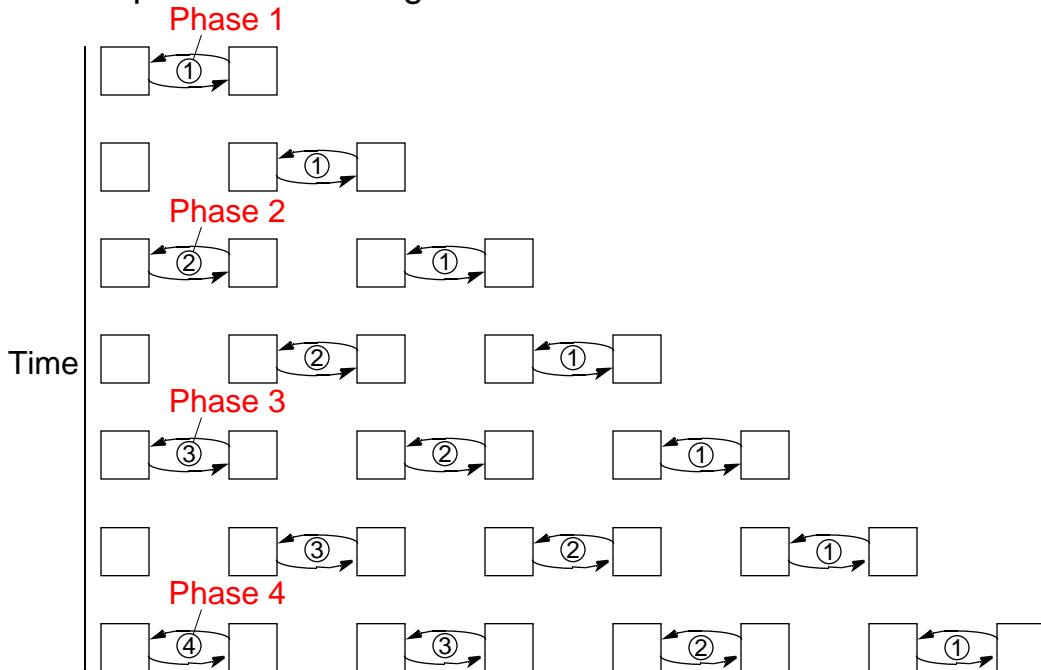
## Time Complexity

$$\text{Number of compare and exchange operations} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

which indicates a time complexity of  $(n^2)$  given that a single compare-and-exchange operation has a constant complexity, (1).

# Parallel Bubble Sort

Iteration could start before previous iteration finished if does not overtake previous bubbling action:



## Odd-Even (Transposition) Sort

Variation of bubble sort.

Operates in two alternating phases, *even* phase and *odd* phase.

### Even phase

Even-numbered processes exchange numbers with their right neighbor.

### Odd phase

Odd-numbered processes exchange numbers with their right neighbor.

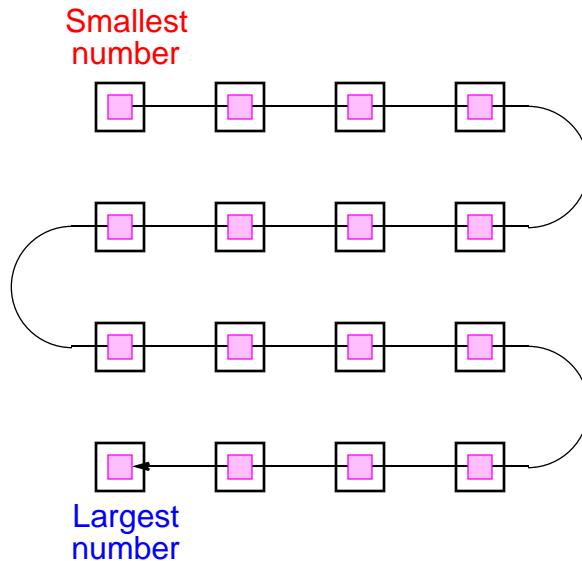
# Odd-Even Transposition Sort

## Sorting eight numbers

	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>	P <sub>5</sub>	P <sub>6</sub>	P <sub>7</sub>
Step	4 ↔ 2	7 ↔ 8	5 ↔ 1	3 ↔ 6				
Time	2 ↔ 4	4 ↔ 7	8 ↔ 1	5 ↔ 3	6			
0	4 ↔ 2	7 ↔ 8	5 ↔ 1	3 ↔ 6				
1	2 ↔ 4	4 ↔ 7	8 ↔ 1	5 ↔ 3	6			
2	2 ↔ 4	7 ↔ 1	8 ↔ 3	5 ↔ 6				
3	2 ↔ 4	4 ↔ 1	7 ↔ 3	8 ↔ 5	6			
4	2 ↔ 1	4 ↔ 3	7 ↔ 5	8 ↔ 6				
5	1 ↔ 2	2 ↔ 3	4 ↔ 5	7 ↔ 6	8			
6	1 ↔ 2	3 ↔ 4	5 ↔ 6	7 ↔ 8				
7	1	2 ↔ 3	4 ↔ 5	6 ↔ 7	8			

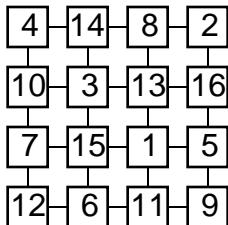
# Two-Dimensional Sorting

The layout of a sorted sequence on a mesh could be row by row or *snakelike*. Snakelike:

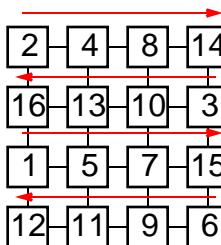


# Shearsort

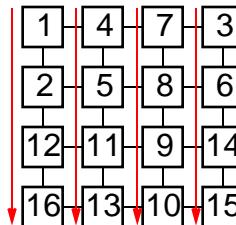
Alternate row and column sorting until list fully sorted. Row sorting alternative directions to get snake-like sorting:



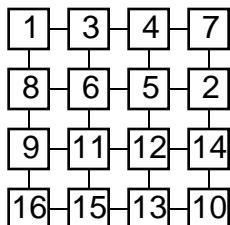
(a) Original placement of numbers



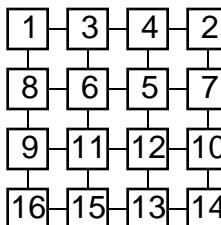
(b) Phase 1 — Row sort



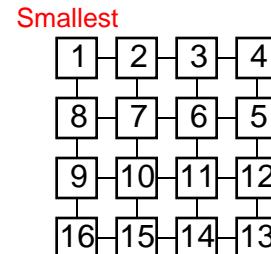
(c) Phase 2 — Column sort



(d) Phase 3 — Row sort



(e) Phase 4 — Column sort



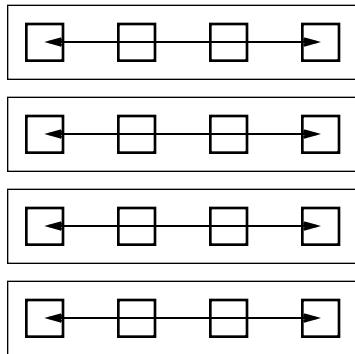
(f) Final phase — Row sort

## Shearsort

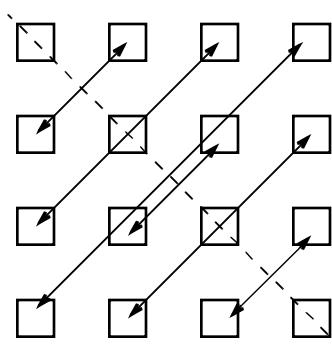
Requires  $\sqrt{n}(\log n + 1)$  steps for  $n$  numbers on a  $\sqrt{n} \times \sqrt{n}$  mesh.

# Using Transposition

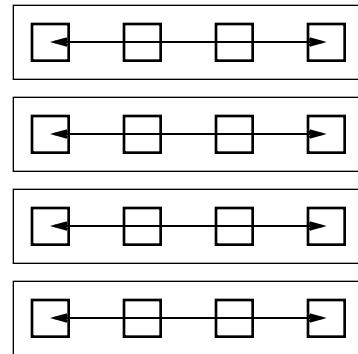
Causes the elements in each column to be in positions in a row.  
Can be placed between the row operations and column operations:



(a) Operations between elements  
in rows



(b) Transpose operation

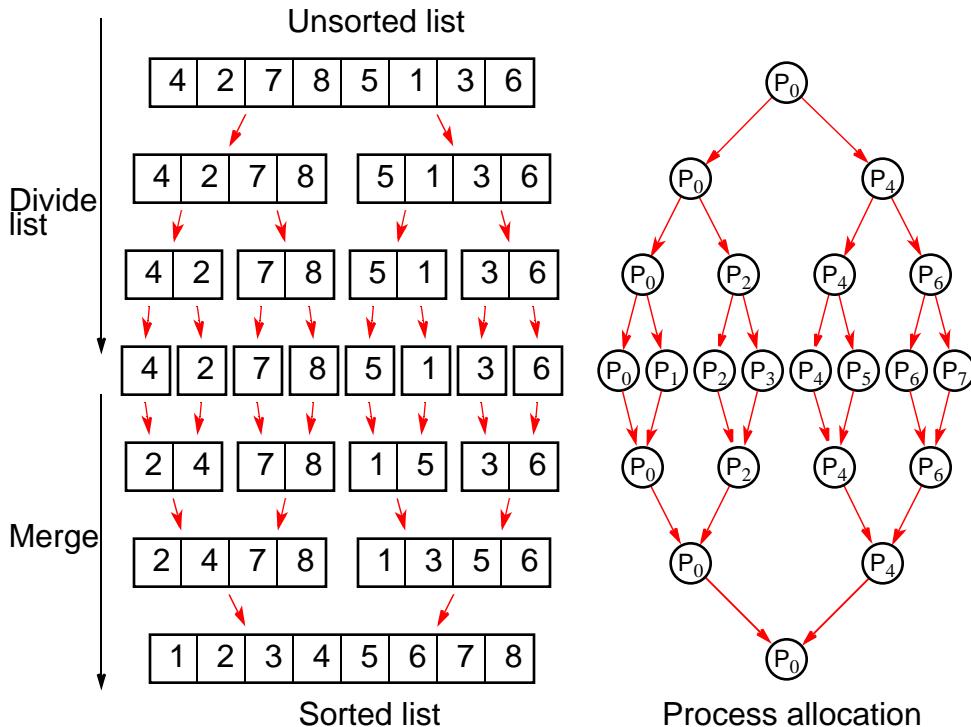


(c) Operations between elements  
in rows (originally columns)

Transposition can be achieved with  $\sqrt{n}(\sqrt{n} - 1)$  communications ( $n$ ). An *all-to-all* routine could be used to reduce this.

# Parallelizing Mergesort

Using tree allocation of processes



# Analysis

## Sequential

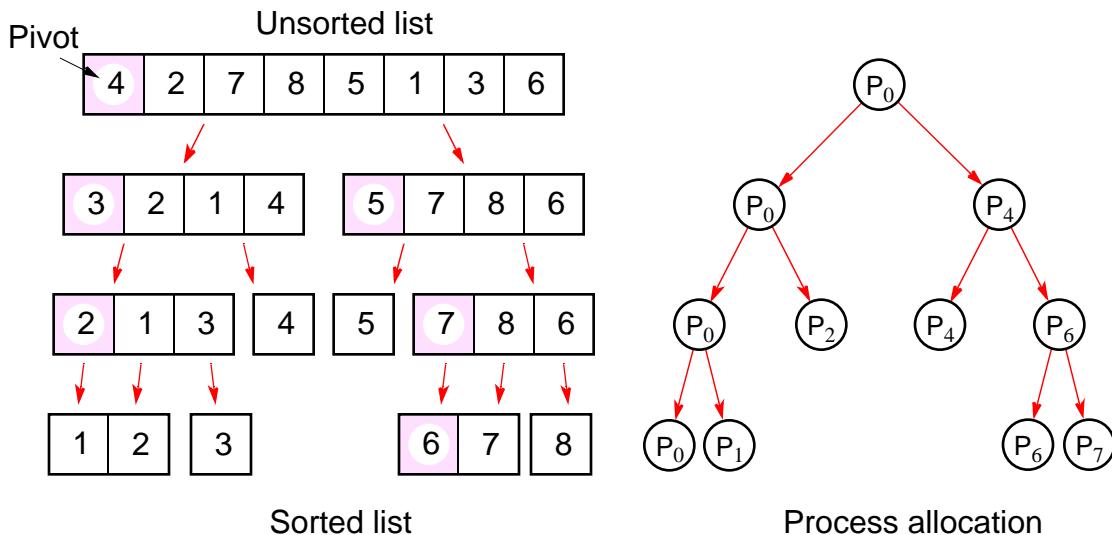
Sequential time complexity is  $(n \log n)$ .

## Parallel

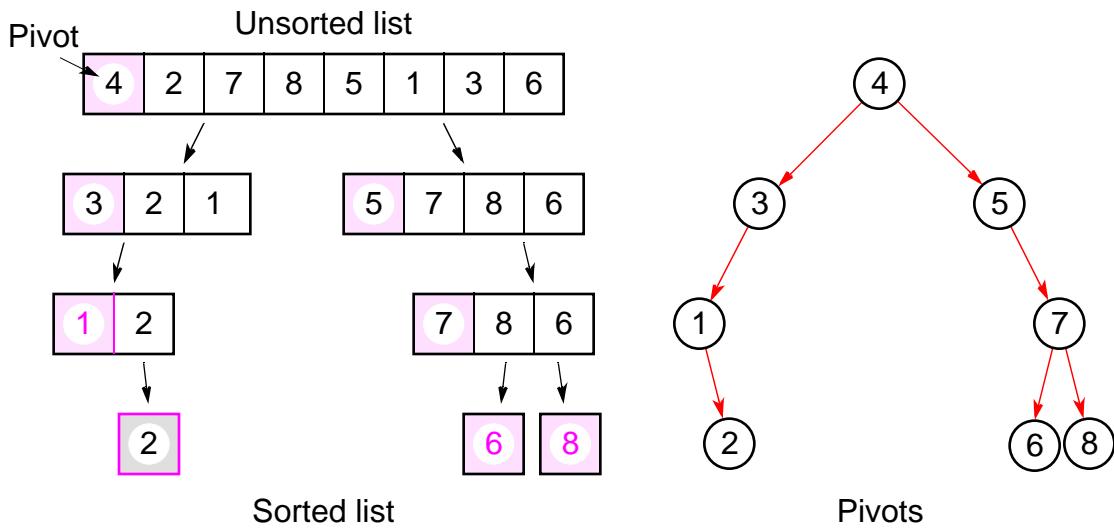
2  $\log n$  steps in the parallel version but each step may need to perform more than one basic operation, depending upon the number of numbers being processed - see text.

# Parallelizing Quicksort

Using tree allocation of processes



With the pivot being withheld in processes:



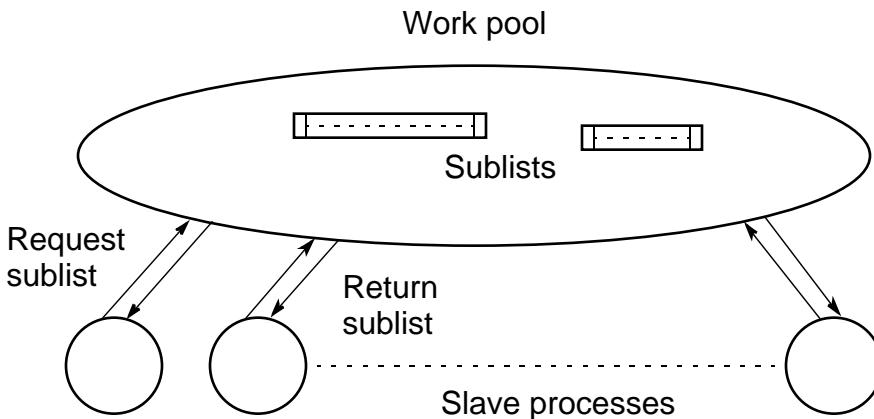
## Analysis

Fundamental problem with all tree constructions – initial division done by a single processor, which will seriously limit speed.

Tree in quicksort will not, in general, be perfectly balanced Pivot selection very important to make quicksort operate fast.

# Work Pool Implementation of Quicksort

First, work pool holds initial unsorted list. Given to first processor which divides list into two parts. One part returned to work pool to be given to another processor, while the other part operated upon again.



Neither Mergesort nor Quicksort parallelize very well as the processor efficiency is low (see book for analysis).

Quicksort also can be very unbalanced. Can use load balancing techniques

Parallel **hypercube** versions of quicksort in textbook - however hypercubes not now of much interest.

# Batcher's Parallel Sorting Algorithms

- Odd-even Mergesort
- Bitonic Mergesort

Originally derived in terms of switching networks.

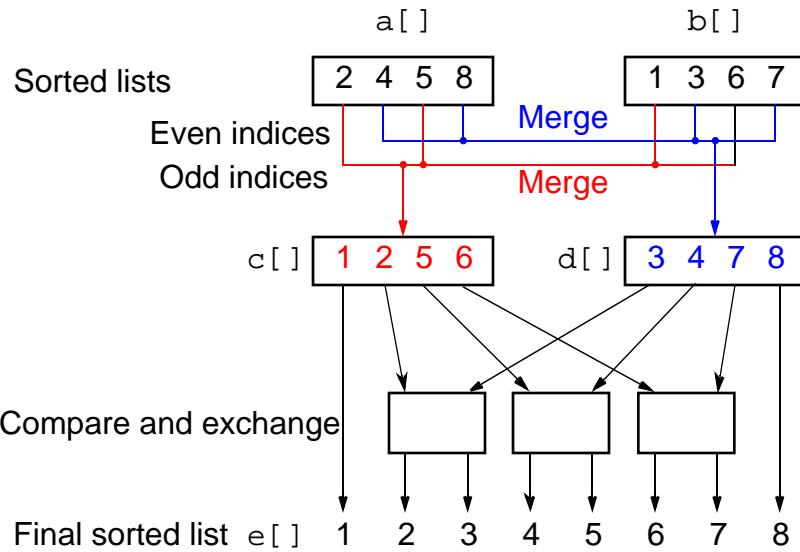
Both are well balanced and have parallel time complexity of  $O(\log^2 n)$  with  $n$  processors.

# **Odd-Even Mergesort**

## **Odd-Even Merge Algorithm**

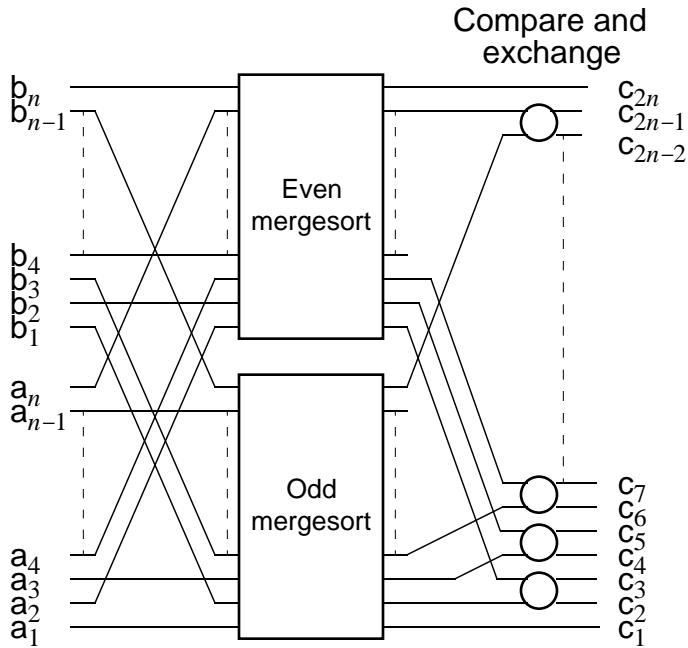
Start with odd-even merge algorithm which will merge two *sorted* lists into one sorted list. Given two sorted lists  $a_1, a_2, a_3, \dots, a_n$  and  $b_1, b_2, b_3, \dots, b_n$  (where  $n$  is a power of 2)

# Odd-Even Merging of Two Sorted Lists



# Odd-Even Mergesort

Apply odd-even merging recursively



# Bitonic Mergesort

## Bitonic Sequence

A **monotonic** increasing sequence is a sequence of increasing numbers.

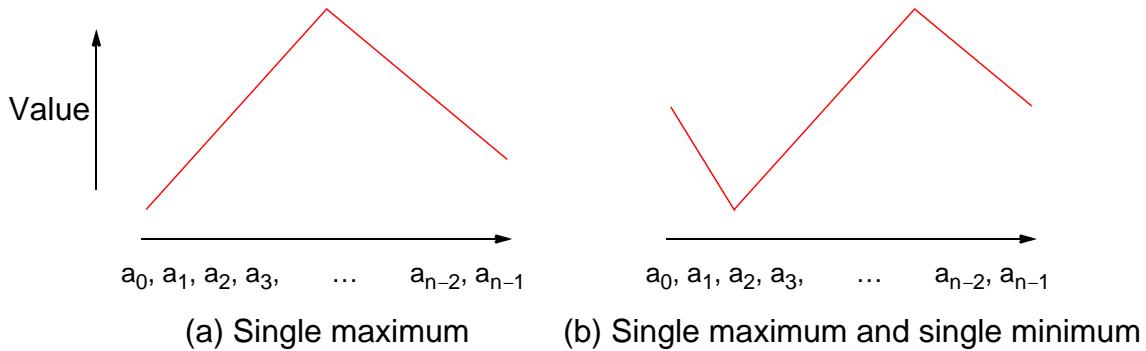
A **bitonic sequence** has two sequences, one increasing and one decreasing. e.g.

$$a_0 < a_1 < a_2, a_3, \dots, a_{i-1} < a_i > a_{i+1}, \dots, a_{n-2} > a_{n-1}$$

for some value of  $i$  ( $0 \leq i < n$ ).

A sequence is also bitonic if the preceding can be achieved by shifting the numbers cyclically (left or right).

# Bitonic Sequences



## “Special” Characteristic of Bitonic Sequences

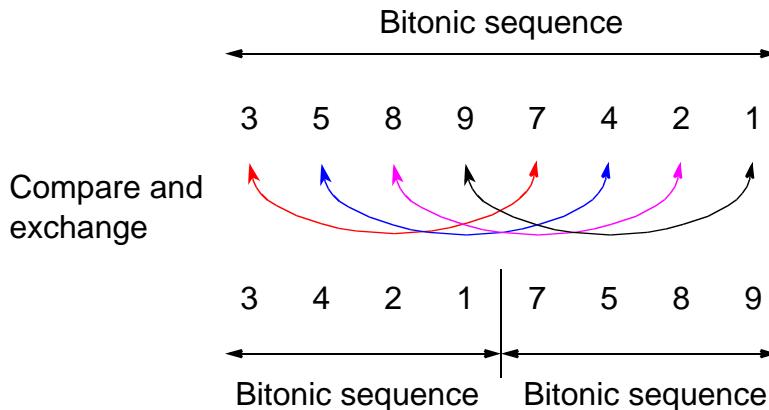
If we perform a compare-and-exchange operation on  $a_i$  with  $a_{i+n/2}$  for all  $i$ , where there are  $n$  numbers in the sequence, get **TWO** bitonic sequences, where the numbers in one sequence are **all less than the numbers in the other sequence.**

# Example - Creating two bitonic sequences from one bitonic sequence

Starting with the bitonic sequence

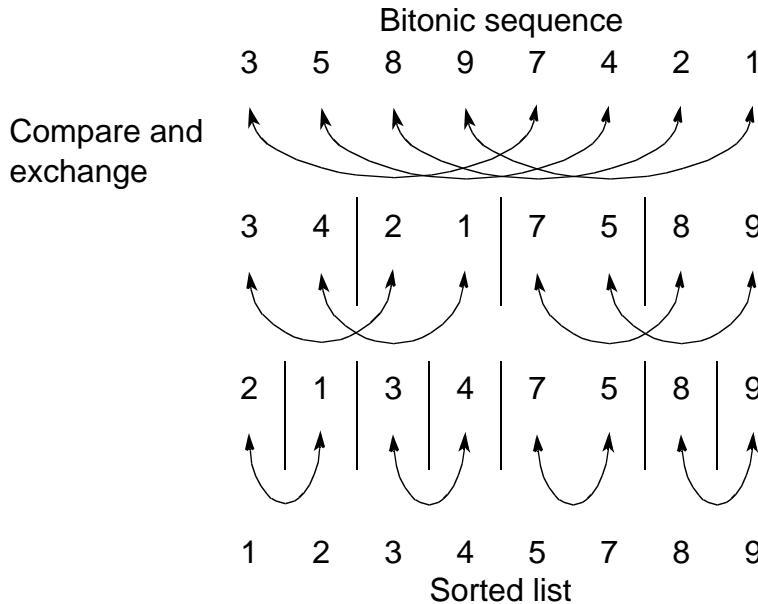
3, 5, 8, 9, 7, 4, 2, 1

we get:



# Sorting a bitonic sequence

Compare-and-exchange moves smaller numbers of each pair to left and larger numbers of pair to right. **Given a bitonic sequence**, recursively performing operations will sort the list.



## Sorting

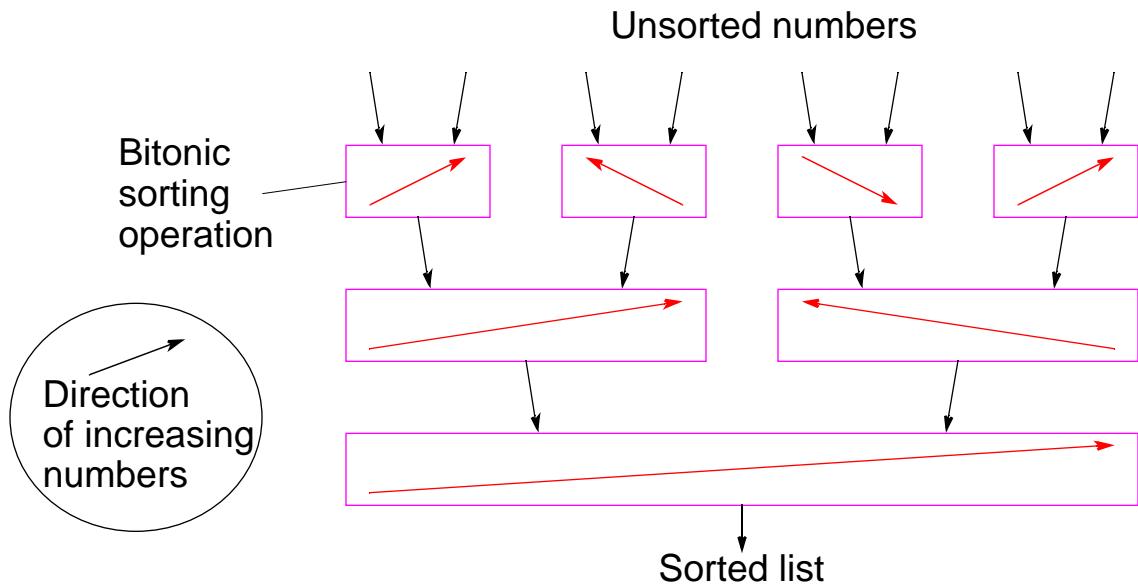
To sort an **unordered sequence**, sequences are merged into larger bitonic sequences, starting with pairs of adjacent numbers.

By a compare-and-exchange operation, pairs of adjacent numbers are formed into increasing sequences and decreasing sequences, pairs of which form a bitonic sequence of twice the size of each of the original sequences.

By repeating this process, bitonic sequences of larger and larger lengths are obtained.

In the final step, a single bitonic sequence is sorted into a single increasing sequence.

# Bitonic Mergesort



# Bitonic Mergesort on Eight Numbers

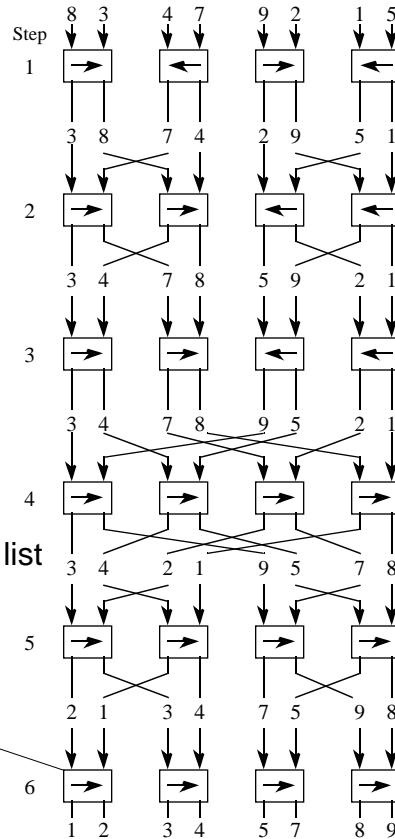
Form bitonic lists of four numbers

Form bitonic list of eight numbers

Sort bitonic list

Compare and exchange

Lower      Higher



# Phases

The six steps (for eight numbers) are divided into three phases:

Phase 1 (Step 1) Convert pairs of numbers into increasing/decreasing sequences and hence into 4-bit bitonic sequences.

Phase 2 (Steps 2/3) Split each 4-bit bitonic sequence into two 2-bit bitonic sequences, higher sequences at center.

Sort each 4-bit bitonic sequence increasing/decreasing sequences and merge into 8-bit bitonic sequence.

Phase 3 (Steps 4/5/6) Sort 8-bit bitonic sequence

## Number of Steps

In general, with  $n = 2^k$ , there are  $k$  phases, each of 1, 2, 3, ...,  $k$  steps. Hence the total number of steps is given by

$$\text{Steps} = \sum_{i=1}^k i = \frac{k(k+1)}{2} = \frac{\log n (\log n + 1)}{2} = (\log^2 n)$$

# Sorting Conclusions

Computational time complexity using n processors

- Ranksort  $O(n)$
- Odd-even transposition sort-  $O(n)$
- Parallel mergesort -  $O(n)$  but unbalanced processor load and communication
- Parallel quicksort -  $O(n)$  but unbalanced processor load, and communication can generate to  $O(n^2)$
- Odd-even Mergesort and Bitonic Mergesort  $O(\log^2 n)$

Bitonic mergesort has been a popular choice for a parallel sorting.

## Chapter 10

# Numerical Algorithms

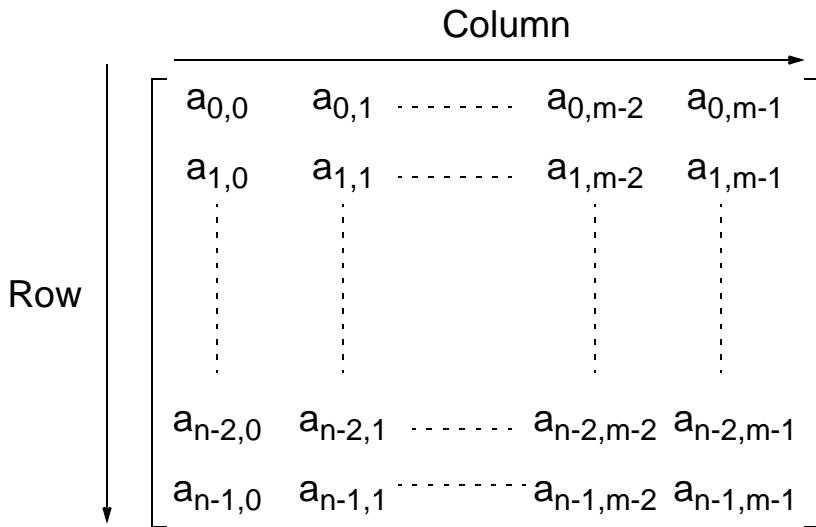
# Numerical Algorithms

In textbook do:

- Matrix multiplication
- Solving a system of linear equations

# Matrices — A Review

An  $n \times m$  matrix



## Matrix Addition

Involves adding corresponding elements of each matrix to form the result matrix.

Given the elements of **A** as  $a_{i,j}$  and the elements of **B** as  $b_{i,j}$ , each element of **C** is computed as

$$c_{i,j} = a_{i,j} + b_{i,j}$$

$$(0 \leq i < n, 0 \leq j < m)$$

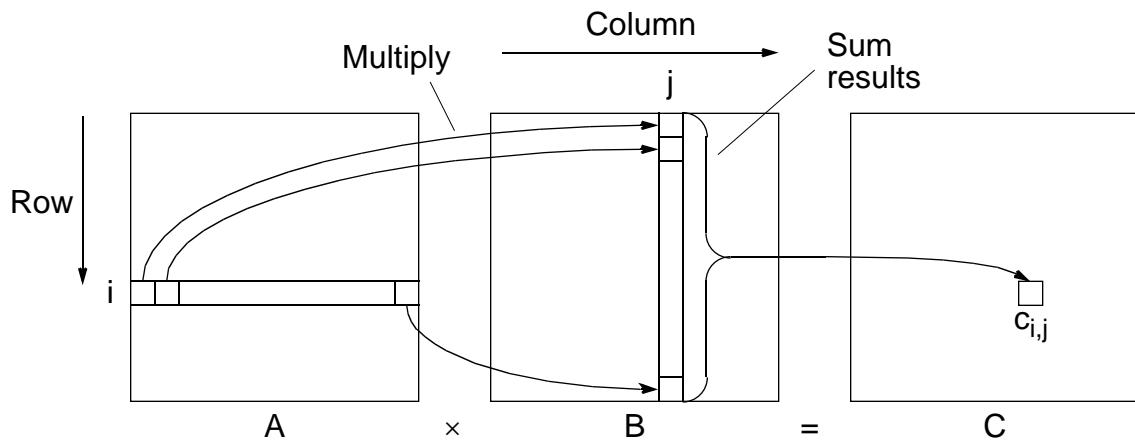
## Matrix Multiplication

Multiplication of two matrices, **A** and **B**, produces the matrix **C** whose elements,  $c_{i,j}$  ( $0 \leq i < n$ ,  $0 \leq j < m$ ), are computed as follows:

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$

where **A** is an  $n \times l$  matrix and **B** is an  $l \times m$  matrix.

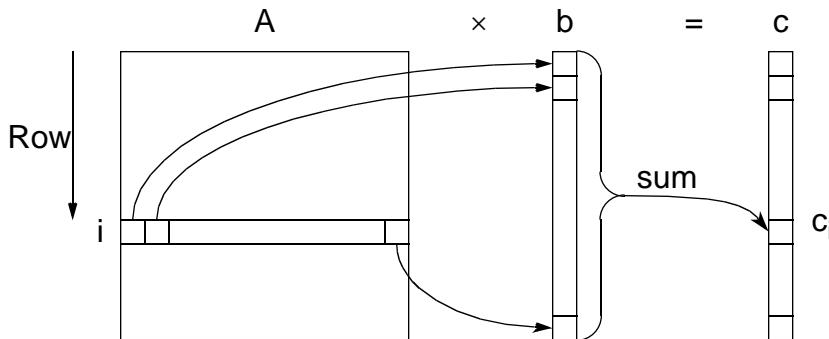
# Matrix multiplication, $C = A \times B$



# Matrix-Vector Multiplication

$$\mathbf{c} = \mathbf{A} \times \mathbf{b}$$

Matrix-vector multiplication follows directly from the definition of matrix-matrix multiplication by making  $\mathbf{B}$  an  $n \times 1$  matrix (vector). Result an  $n \times 1$  matrix (vector).



## Relationship of Matrices to Linear Equations

A system of linear equations can be written in matrix form:

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

Matrix **A** holds the *a* constants

**x** is a vector of the unknowns

**b** is a vector of the *b* constants.

# Implementing Matrix Multiplication

## Sequential Code

Assume throughout that the matrices are square ( $n \times n$  matrices).

The sequential code to compute  $\mathbf{A} \times \mathbf{B}$  could simply be

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        c[i][j] = 0;
        for (k = 0; k < n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
```

This algorithm requires  $n^3$  multiplications and  $n^3$  additions, leading to a sequential time complexity of  $(n^3)$ . Very easy to parallelize.

## Parallel Code

With  $n$  processors (and  $n \times n$  matrices), can obtain:

- Time complexity of  $O(n^2)$  with  $n$  processors  
Each instance of inner loop independent and can be done by a separate processor
- Time complexity of  $O(n)$  with  $n^2$  processors  
One element of  $A$  and  $B$  assigned to each processor.  
Cost optimal since  $O(n^3) = n \times O(n^2) = n^2 \times O(n)$ .
- Time complexity of  $O(\log n)$  with  $n^3$  processors  
By parallelizing the inner loop. Not cost-optimal since  $O(n^3) \propto n^3 \times O(\log n)$ .

$O(\log n)$  lower bound for parallel matrix multiplication.

# Partitioning into Submatrices

Suppose matrix divided into  $s^2$  submatrices. Each submatrix has  $n/s \times n/s$  elements. Using notation  $A_{p,q}$  as submatrix in submatrix row  $p$  and submatrix column  $q$ :

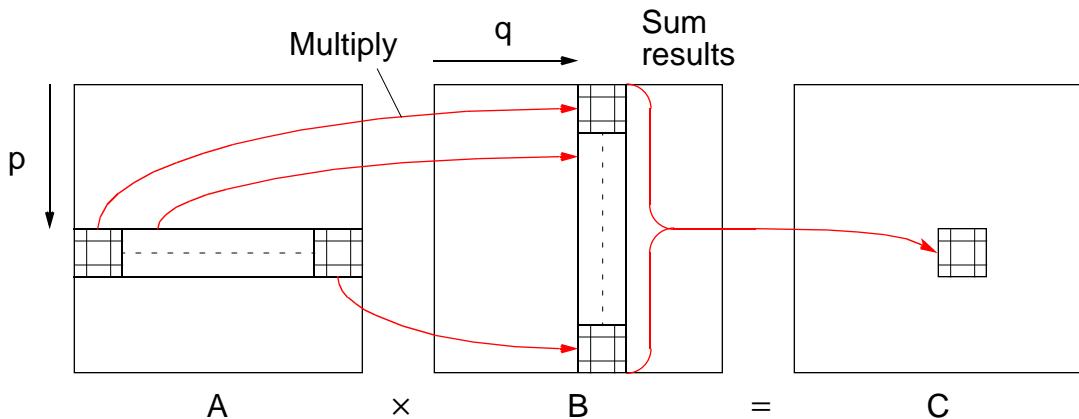
```
for (p = 0; p < s; p++)
    for (q = 0; q < s; q++) {
        Cp,q = 0; /* clear elements of submatrix */
        for (r = 0; r < m; r++)/* submatrix multiplication */
            Cp,q = Cp,q + Ap,r * Br,q/*add to accum. submatrix*/
    }
```

The line

$$C_{p,q} = C_{p,q} + A_{p,r} * B_{r,q};$$

means multiply submatrix  $A_{p,r}$  and  $B_{r,q}$  using matrix multiplication and add to submatrix  $C_{p,q}$  using matrix addition. Known as *block matrix multiplication*.

# Block Matrix Multiplication



# Submatrix multiplication

(a) Matrices

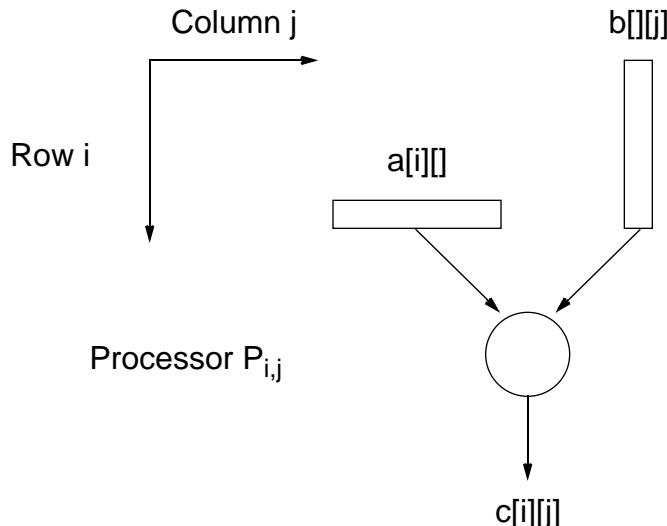
$$\begin{array}{c} \left[ \begin{array}{cc|cc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \end{array} \right] \times \left[ \begin{array}{cc|cc} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \end{array} \right] \\ \hline \left[ \begin{array}{cc|cc} a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{array} \right] \quad \left[ \begin{array}{cc|cc} b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{array} \right] \end{array}$$

(b) Multiplying  $A_{0,0} \times B_{0,0}$  to obtain  $C_{0,0}$ 

$$\begin{aligned} & A_{0,0} \qquad \qquad \qquad B_{0,0} \qquad \qquad \qquad A_{0,1} \qquad \qquad \qquad B_{1,0} \\ & \left[ \begin{array}{cc} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{array} \right] \times \left[ \begin{array}{cc} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{array} \right] + \left[ \begin{array}{cc} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{array} \right] \times \left[ \begin{array}{cc} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{array} \right] \\ & = \left[ \begin{array}{cc} a_{0,0}b_{0,0}+a_{0,1}b_{1,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1} \end{array} \right] + \left[ \begin{array}{cc} a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{array} \right] \\ & = \left[ \begin{array}{cc} a_{0,0}b_{0,0}+a_{0,1}b_{1,0}+a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1}+a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0}+a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1}+a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{array} \right] \\ & \qquad \qquad \qquad = C_{0,0} \end{aligned}$$

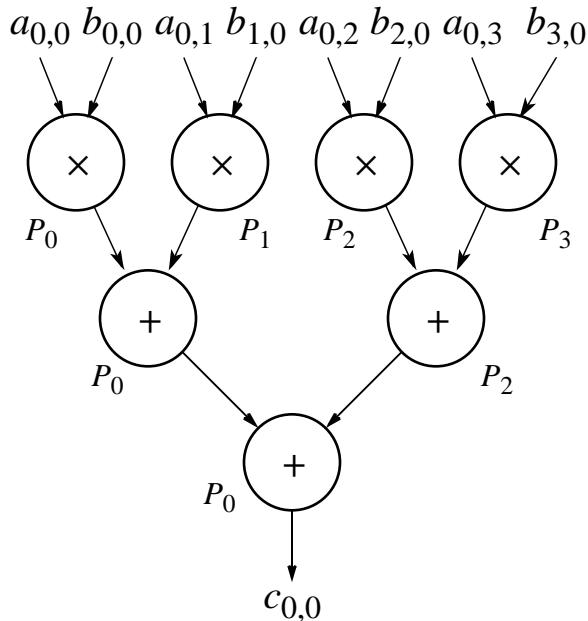
## Direct Implementation

One processor to compute each element of  $\mathbf{C}$  -  $n^2$  processors would be needed. One row of elements of  $\mathbf{A}$  and one column of elements of  $\mathbf{B}$  needed. Some of same elements sent to more than one processor. Can use submatrices.



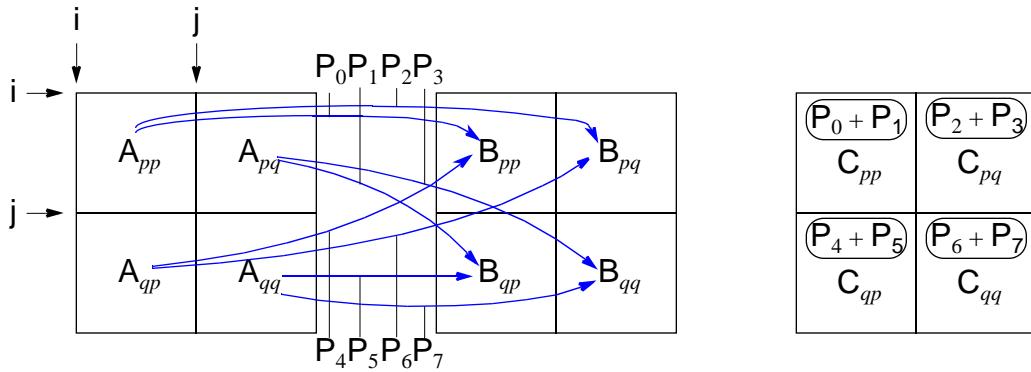
# Performance Improvement

Using tree construction  $n$  numbers can be added in  $\log n$  steps using  $n$  processors:



Computational time complexity of  $(\log n)$  using  $n^3$  processors.

# Recursive Implementation



Apply same algorithm on each submatrix recursively.

Excellent algorithm for a shared memory systems because of locality of operations.

# Recursive Algorithm

```
mat_mult(App, Bpp, s)
{
    if (s == 1)          /* if submatrix has one element */
        C = A * B;      /* multiply elements */
    else {
        /* continue to make recursive calls */
        s = s/2;         /* no of elements in each row/column */
        P0 = mat_mult(App, Bpp, s);
        P1 = mat_mult(Apq, Bqp, s);
        P2 = mat_mult(App, Bpq, s);
        P3 = mat_mult(Apq, Bqq, s);
        P4 = mat_mult(Aqp, Bpp, s);
        P5 = mat_mult(Aqq, Bqp, s);
        P6 = mat_mult(Aqp, Bpq, s);
        P7 = mat_mult(Aqq, Bqq, s);
        Cpp = P0 + P1;    /* add submatrix products to */
        Cpq = P2 + P3;    /* form submatrices of final matrix */
        Cqp = P4 + P5;
        Cqq = P6 + P7;
    }
    return (C);           /* return final matrix */
}
```

## Mesh Implementations

- Cannon's algorithm
- Fox's algorithm (not in textbook but similar complexity)
- Systolic array

All involve using processor arranged a mesh and shifting elements of the arrays through the mesh. Accumulate the partial sums at each processor.

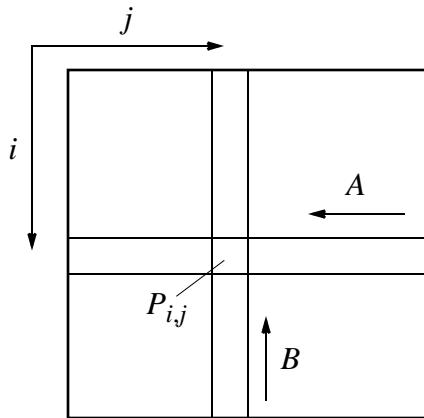
# Mesh Implementations

## Cannon's Algorithm

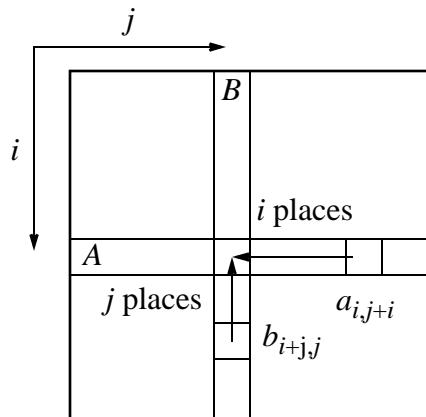
Uses a mesh of processors with wraparound connections (a torus) to shift the A elements (or submatrices) left and the B elements (or submatrices) up.

1. Initially processor  $P_{i,j}$  has elements  $a_{i,j}$  and  $b_{i,j}$  ( $0 \leq i < n, 0 \leq j < n$ ).
2. Elements are moved from their initial position to an “aligned” position. The complete  $i$ th row of A is shifted  $i$  places left and the complete  $j$ th column of B is shifted  $j$  places upward. This has the effect of placing the element  $a_{i,j+i}$  and the element  $b_{i+j,j}$  in processor  $P_{i,j}$ . These elements are a pair of those required in the accumulation of  $c_{i,j}$ .
3. Each processor,  $P_{i,j}$ , multiplies its elements.
4. The  $i$ th row of A is shifted one place right, and the  $j$ th column of B is shifted one place upward. This has the effect of bringing together the adjacent elements of A and B, which will also be required in the accumulation.
5. Each processor,  $P_{i,j}$ , multiplies the elements brought to it and adds the result to the accumulating sum.
6. Step 4 and 5 are repeated until the final result is obtained ( $n - 1$  shifts with  $n$  rows and  $n$  columns of elements).

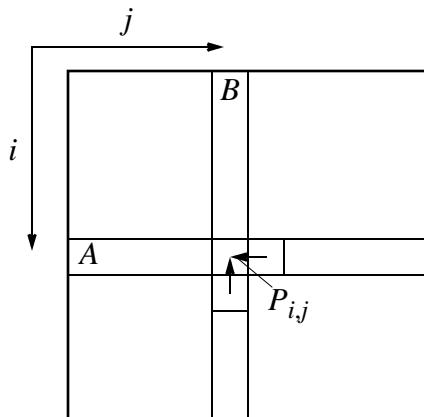
# Movement of *A* and *B* elements



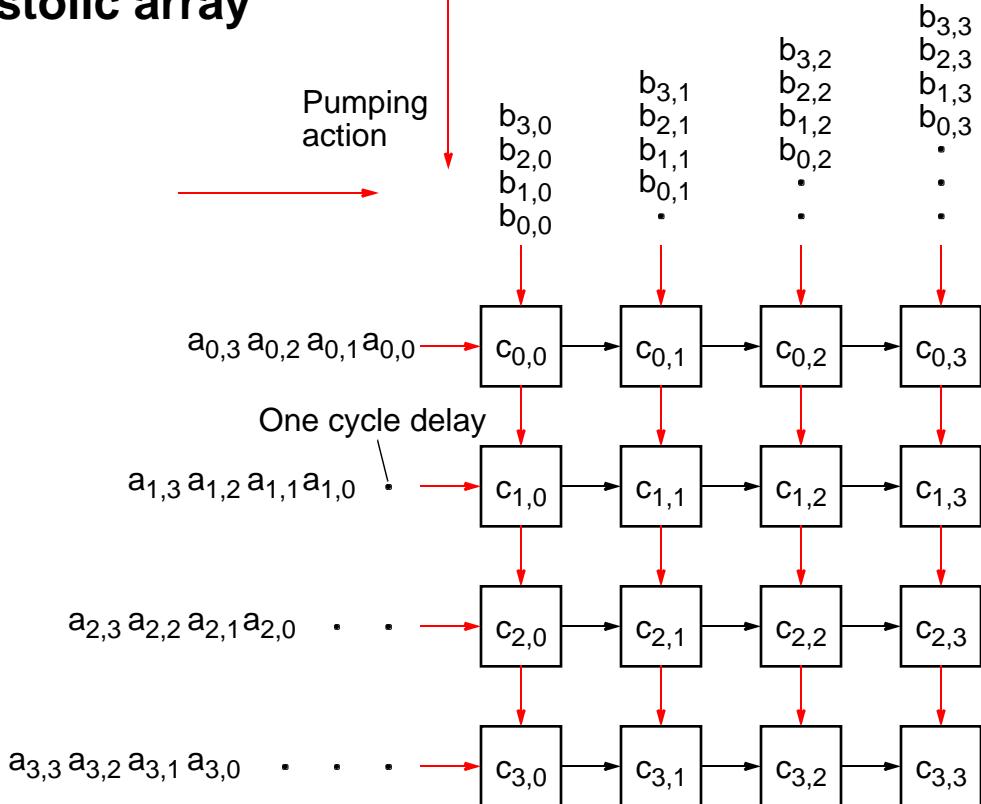
## Step 2 — Alignment of elements of $A$ and $B$



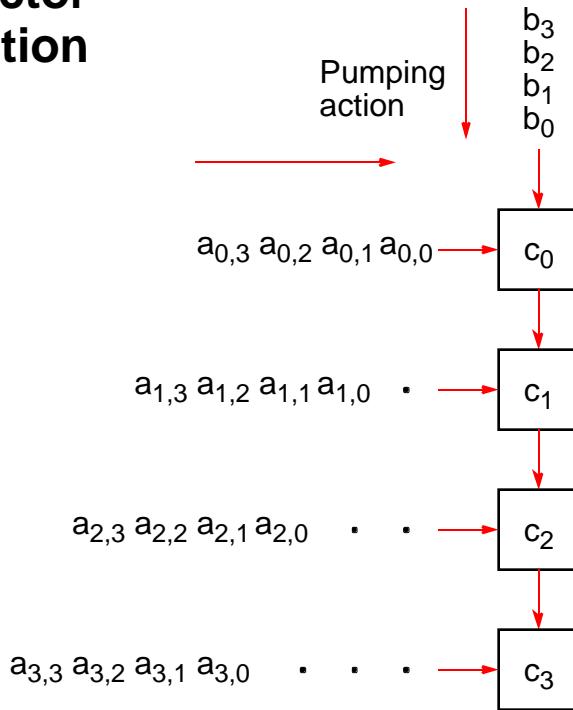
## Step 4 — One-place shift of elements of $A$ and $B$



# Systolic array



# Matrix-Vector Multiplication



# Solving a System of Linear Equations

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \quad \dots \quad + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

.

.

.

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \quad \dots \quad + a_{2,n-1}x_{n-1} = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \quad \dots \quad + a_{1,n-1}x_{n-1} = b_1$$

$$a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \quad \dots \quad + a_{0,n-1}x_{n-1} = b_0$$

which, in matrix form, is

$$\mathbf{Ax} = \mathbf{b}$$

Objective is to find values for the unknowns,  $x_0, x_1, \dots, x_{n-1}$ , given values for  $a_{0,0}, a_{0,1}, \dots, a_{n-1,n-1}$ , and  $b_0, \dots, b_n$ .

# Solving a System of Linear Equations

## Dense matrices

Gaussian Elimination - parallel time complexity  $O(n^2)$

## Sparse matrices

By iteration - depends upon iteration method and number of iterations but typically  $O(\log n)$

- Jacobi iteration
- Gauss-Seidel relaxation (not good for parallelization)
- Red-Black ordering
- Multigrid

## Gaussian Elimination

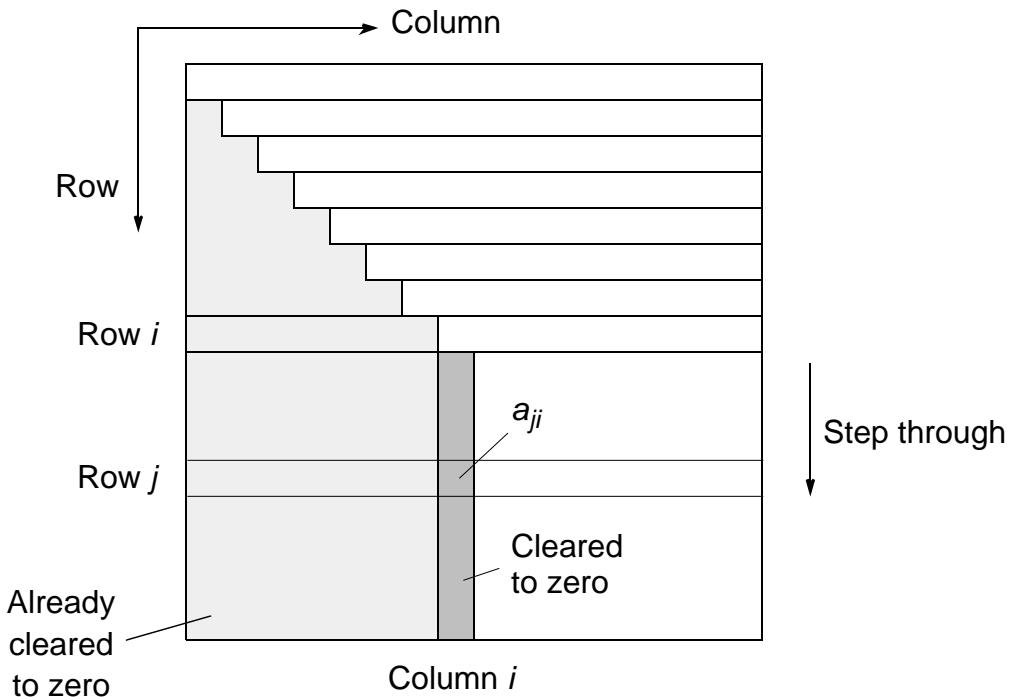
Convert general system of linear equations into triangular system of equations. Then be solved by Back Substitution.

Uses characteristic of linear equations that any row can be replaced by that row added to another row multiplied by a constant.

Starts at the first row and works toward the bottom row. At the  $i$ th row, each row  $j$  below the  $i$ th row is replaced by row  $j + (\text{row } i) \cdot (-a_{j,i}/a_{i,i})$ . The constant used for row  $j$  is  $-a_{j,i}/a_{i,i}$ . Has the effect of making all the elements in the  $i$ th column below the  $i$ th row zero because

$$a_{j,i} = a_{j,i} + a_{i,i} \frac{-a_{j,i}}{a_{i,i}} = 0$$

# Gaussian elimination



## Partial Pivoting

If  $a_{i,i}$  is zero or close to zero, we will not be able to compute the quantity  $-a_{j,i}/a_{i,i}$ .

Procedure must be modified into so-called *partial pivoting* by swapping the  $i$ th row with the row below it that has the largest absolute element in the  $i$ th column of any of the rows below the  $i$ th row if there is one. (Reordering equations will not affect the system.)

In the following, we will not consider partial pivoting.

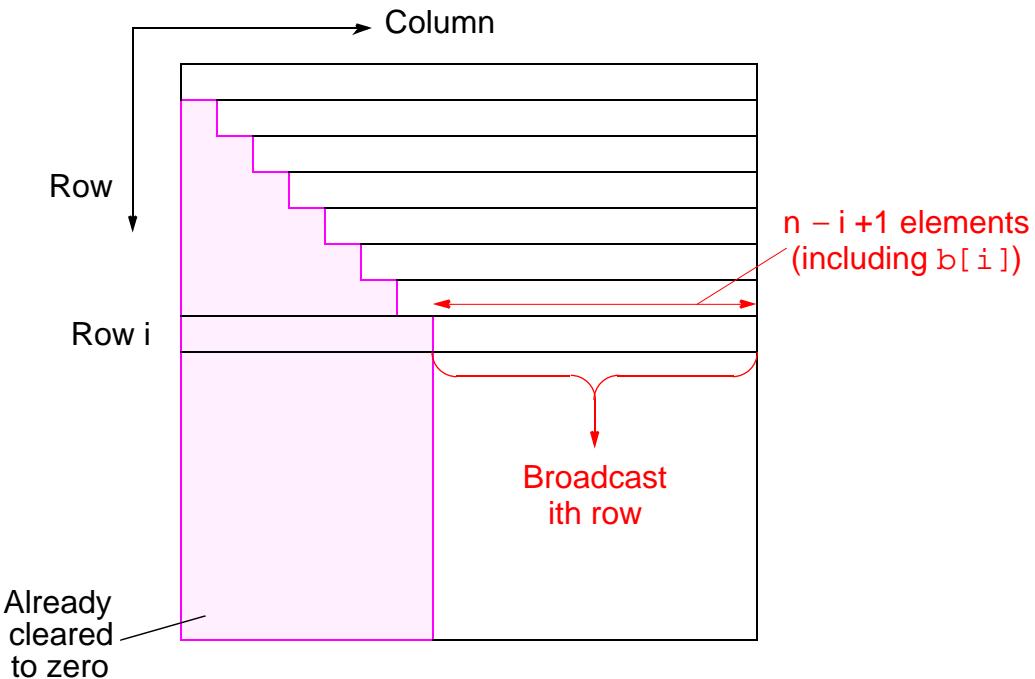
## Sequential Code

Without partial pivoting:

```
for (i = 0; i < n-1; i++) /* for each row, except last */
    for (j = i+1; j < n; j++) /*step thro subsequent rows */
        m = a[j][i]/a[i][i]; /* Compute multiplier */
        for (k = i; k < n; k++)/*last n-i-1 elements of row j*/,
            a[j][k] = a[j][k] - a[i][k] * m;
        b[j] = b[j] - b[i] * m/* modify right side */
    }
```

The time complexity is  $O(n^3)$ .

# Parallel Implementation



## Analysis Communication

$n - 1$  broadcasts performed sequentially.  $i$ th broadcast contains  $n - i + 1$  elements.

Time complexity of  $(n^2)$  (see textbook)

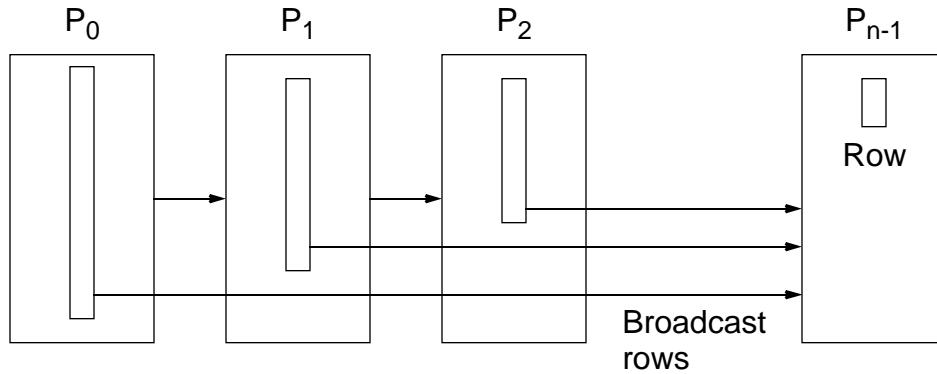
## Computation

After row broadcast, each processor  $P_j$  beyond broadcast processor  $P_i$  will compute its multiplier, and operate upon  $n - j + 2$  elements of its row. Ignoring the computation of the multiplier, there are  $n - j + 2$  multiplications and  $n - j + 2$  subtractions.

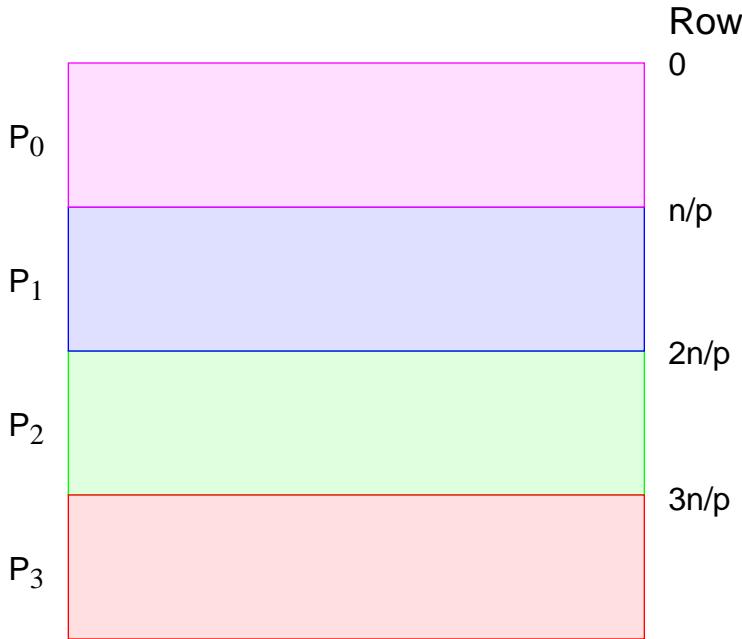
Time complexity of  $(n^2)$  (see textbook).

Efficiency will be relatively low because all the processors before the processor holding row  $i$  do not participate in the computation again.

# Pipeline implementation of Gaussian elimination



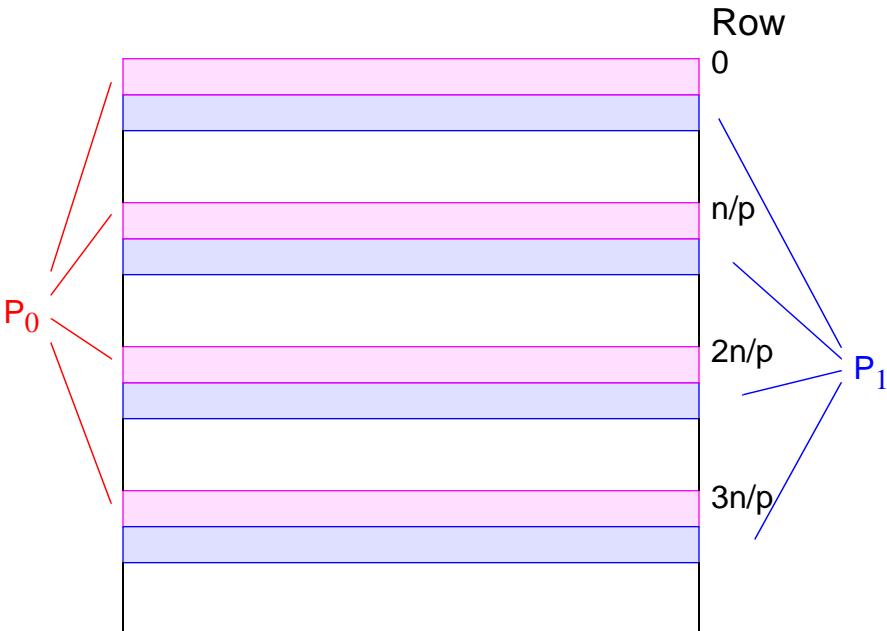
# Strip Partitioning



Poor processor allocation! Processors do not participate in computation after their last row is processed.

# Cyclic-Striped Partitioning

An alternative which equalizes the processor workload:



## Iterative Methods

Time complexity of direct method at  $(N^2)$  with  $N$  processors, is significant.

Time complexity of iteration method depends upon:

- the type of iteration,
- number of iterations
- number of unknowns, and
- required accuracy

but can be less than the direct method especially for a few unknowns i.e a sparse system of linear equations.

## Jacobi Iteration

Iteration formula -  $i$ th equation rearranged to have  $i$ th unknown on left side:

$$x_i^k = \frac{1}{a_{i,i}} \left[ b_i - \sum_j a_{i,j} x_j^{k-1} \right]$$

Superscript indicates iteration:

$x_i^k$  is  $k$ th iteration of  $x_i$ ,  $x_j^{k-1}$  is  $(k-1)$ th iteration of  $x_j$ .

# Example of a Sparse System of Linear Equations

## Laplace's Equation

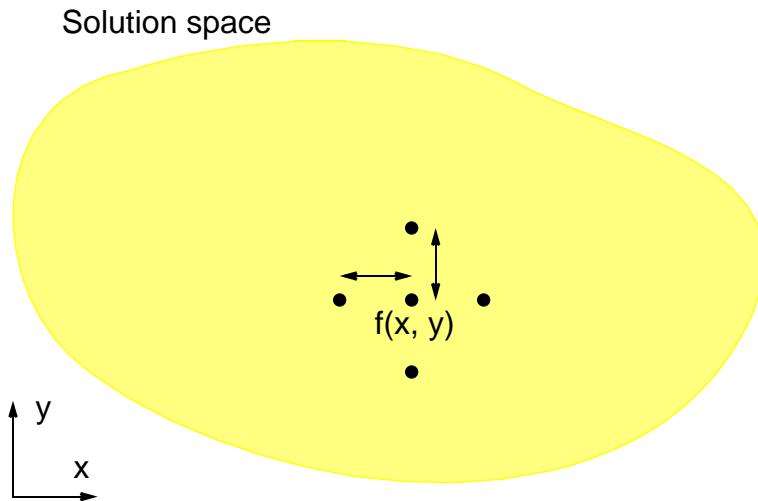
$$\frac{\frac{\partial^2 f}{\partial x^2}}{} + \frac{\frac{\partial^2 f}{\partial y^2}}{} = 0$$

Solve for  $f$  over the two-dimensional x-y space.

For a computer solution, *finite difference* methods are appropriate

Two-dimensional solution space is “discretized” into a large number of solution points.

# Finite Difference Method



If distance between points, , made small enough:

$$\frac{\frac{2}{x^2}f - \frac{1}{2}[f(x + , y) - 2f(x, y) + f(x - , y)]}{}$$

$$\frac{\frac{2}{y^2}f - \frac{1}{2}[f(x, y + ) - 2f(x, y) + f(x, y - )]}{}$$

Substituting into Laplace's equation, we get

$$\frac{1}{2}[f(x + , y) + f(x - , y) + f(x, y + ) + f(x, y - ) - 4f(x, y)] = 0$$

Rearranging, we get

$$f(x, y) = \frac{[f(x - , y) + f(x, y - ) + f(x + , y) + f(x, y + )]}{4}$$

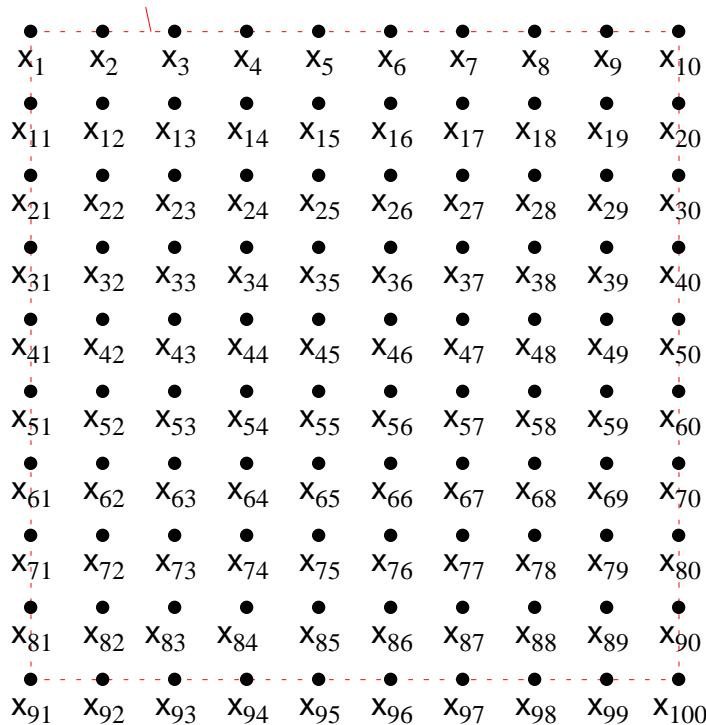
Rewritten as an iterative formula:

$$f^k(x, y) = \frac{[f^{k-1}(x - , y) + f^{k-1}(x, y - ) + f^{k-1}(x + , y) + f^{k-1}(x, y + )]}{4}$$

$f^k(x, y)$  -  $k$ th iteration,  $f^{k-1}(x, y)$  -  $(k - 1)$ th iteration.

# Natural Order

Boundary points (see text)



# Relationship with a General System of Linear Equations

Using natural ordering,  $i$ th point computed from  $i$ th equation:

$$x_i = \frac{x_{i-n} + x_{i-1} + x_{i+1} + x_{i+n}}{4}$$

or

$$x_{i-n} + x_{i-1} - 4x_i + x_{i+1} + x_{i+n} = 0$$

*which is a linear equation with five unknowns* (except those with boundary points).

In general form, the  $i$ th equation becomes:

$$a_{i,i-n}x_{i-n} + a_{i,i-1}x_{i-1} + a_{i,i}x_i + a_{i,i+1}x_{i+1} + a_{i,i+n}x_{i+n} = 0$$

where  $a_{i,i} = -4$ , and  $a_{i,i-n} = a_{i,i-1} = a_{i,i+1} = a_{i,i+n} = 1$ .

A

Those equations with a boundary point on diagonal unnecessary for solution

To include boundary values and some zero entries (see text)

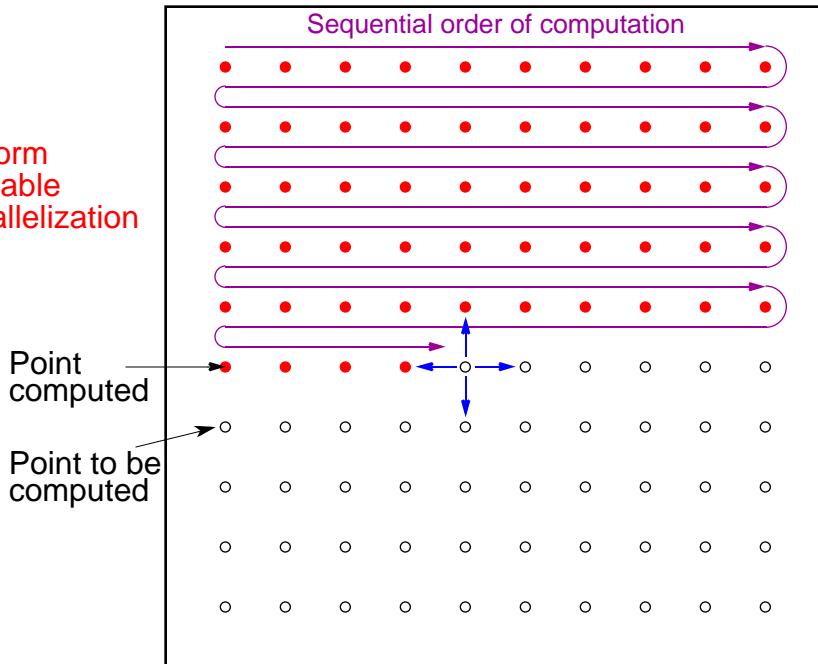
$$\begin{array}{ccccccccc}
 & 1 & 1 & -4 & 1 & 1 & & \\
 & 1 & & 1 & -4 & 1 & & \\
 \text{\color{magenta} } i\text{\color{black} th equation} & \boxed{1} & \color{magenta} 1 & \color{magenta} -4 & \color{magenta} 1 & \color{magenta} 1 & \color{magenta} 1 \\
 & a_{i,i-n} & a_{i,i-1} & a_{i,i} & a_{i,i+1} & a_{i,i+n} & \\
 & 1 & 1 & -4 & 1 & 1 & \\
 & 1 & & 1 & -4 & 1 & \\
 & & & 1 & -4 & 1 & \\
 & & & & 1 & 1 &
 \end{array}$$

$$\begin{bmatrix} x \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

# Gauss-Seidel Relaxation

Uses some newly computed values to compute other values in that iteration.

Basic form  
not suitable  
for parallelization



## Gauss-Seidel Iteration Formula

$$x_i^k = \frac{1}{a_{i,i}} \left[ b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^k - \sum_{j=i+1}^N a_{i,j} x_j^{k-1} \right]$$

where the superscript indicates the iteration.

With natural ordering of unknowns, formula reduces to

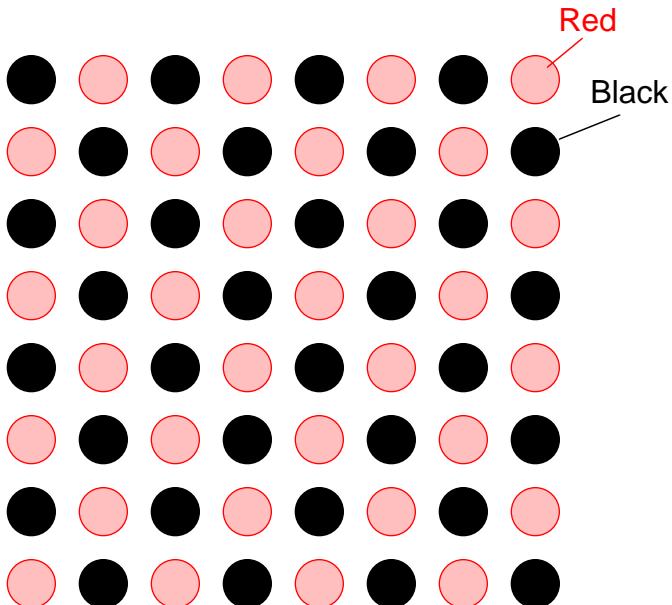
$$x_i^k = (-1/a_{i,i}) [a_{i,i-n} x_{i-n}^k + a_{i,i-1} x_{i-1}^k + a_{i,i+1} x_{i+1}^{k-1} + a_{i,i+n} x_{i+n}^{k-1}]$$

At the  $k$ th iteration, two of the four values (before the  $i$ th element) taken from the  $k$ th iteration and two values (after the  $i$ th element) taken from the  $(k-1)$ th iteration. We have:

$$f^k(x, y) = \frac{[f^k(x-, y) + f^k(x, y-) + f^{k-1}(x+, y) + f^{k-1}(x, y+)]}{4}$$

## Red-Black Ordering

First, black points computed. Next, red points computed. Black points computed simultaneously, and red points computed simultaneously.



## Red-Black Parallel Code

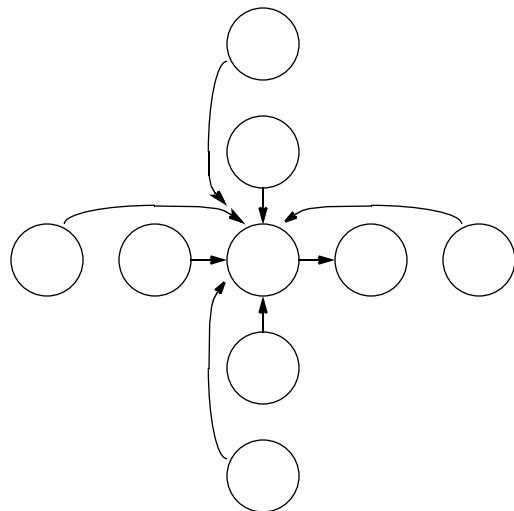
```
forall (i = 1; i < n; i++)
    forall (j = 1; j < n; j++)
        if ((i + j) % 2 == 0)          /* compute red points */
            f[i][j] = 0.25*(f[i-1][j] + f[i][j-1] + f[i+1][j] + f[i][j+1]);
forall (i = 1; i < n; i++)
    forall (j = 1; j < n; j++)
        if ((i + j) % 2 != 0)          /* compute black points */
            f[i][j] = 0.25*(f[i-1][j] + f[i][j-1] + f[i+1][j] + f[i][j+1]);
```

## Higher-Order Difference Methods

More distant points could be used in the computation. The following update formula:

$$\begin{aligned} f^k(x, y) = \\ \frac{1}{60} \left[ 16f^{k-1}(x - \Delta x, y) + 16f^{k-1}(x, y - \Delta y) + 16f^{k-1}(x + \Delta x, y) + 16f^{k-1}(x, y + \Delta y) \right. \\ \left. - f^{k-1}(x - 2\Delta x, y) - f^{k-1}(x, y - 2\Delta y) - f^{k-1}(x + 2\Delta x, y) - f^{k-1}(x, y + 2\Delta y) \right] \end{aligned}$$

# Nine-point stencil



# Overrelaxation

Improved convergence obtained by adding factor  $(1 - \omega)x_i$  to Jacobi or Gauss-Seidel formulae. Factor  $\omega$  is the *overrelaxation parameter*.

## Jacobi overrelaxation formula

$$x_i^k = \frac{1}{a_{ii}} \left[ b_i - \sum_{j \neq i} a_{ij} x_j^{k-1} \right] + (1 - \omega) x_i^{k-1}$$

where  $0 < \omega < 1$ .

## Gauss-Seidel successive overrelaxation

$$x_i^k = \frac{1}{a_{ii}} \left[ b_i - \sum_{j=1}^{i-1} a_{ij} x_j^k - \sum_{j=i+1}^N a_{ij} x_j^{k-1} \right] + (1 - \omega) x_i^{k-1}$$

where  $0 < \omega \leq 2$ . If  $\omega = 1$ , we obtain the Gauss-Seidel method.

## Multigrid Method

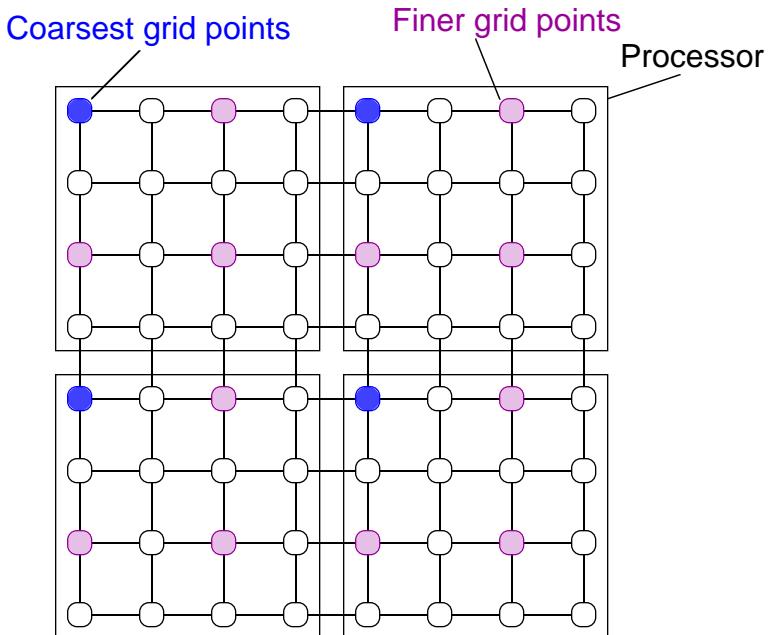
First, a coarse grid of points used. With these points, iteration process will start to converge quickly.

At some stage, number of points increased to include points of the coarse grid and extra points between the points of the coarse grid. Initial values of extra points found by interpolation. Computation continues with this finer grid.

Grid can be made finer and finer as computation proceeds, or computation can alternate between fine and coarse grids.

Coarser grids take into account distant effects more quickly and provide a good starting point for the next finer grid.

# Multigrid processor allocation



## (Semi) Asynchronous Iteration

As noted early, synchronizing on every iteration will cause significant overhead - best if one can is to synchronize after a number of iterations.

## Additional material for Chapter 6 -7

### **Asynchronous Computations**

Computations in which individual processes operate without needing to synchronize with other processes.

In 1st edition of textbook, asynchronous computations not dealt with except under load balancing and termination detection (Chapter 7)

Asynchronous computations important because synchronizing processes is an expensive operation which very significantly slows the computation - A major cause for reduced performance of parallel programs is due to the use of synchronization.

Global synchronization is done with barrier routines. Barriers cause processor to wait sometimes needlessly.

# Asynchronous Example

## Heat Distribution Problem Re-visited

To solve heat distribution problem, solution space divided into a two-dimensional array of points. The value of each point computed by taking average of four points around it repeatedly until values converge on the solution to a sufficient accuracy.



The waiting can be reduced by not forcing synchronization at each iteration.

# Sequential code

```
do {  
    for (i = 1; i < n; i++)  
        for (j = 1; j < n; j++)  
            g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);  
  
    for (i = 1; i < n; i++)      /* find max divergence/update points */  
        for (j = 1; j < n; j++) {  
            dif = h[i][j] - g[i][j];  
            if (dif < 0) dif = -dif;  
            if (dif < max_dif) max_dif = dif;  
            h[i][j] = g[i][j];  
        }  
    } while (max_dif > tolerance);    /* test convergence */
```

First section of code computing the next iteration values based on the immediate previous iteration values is traditional Jacobi iteration method.

Suppose however, processes are to continue with the next iteration before other processes have completed.

Then, the processes moving forward would use values computed from not only the previous iteration but maybe from earlier iterations.

Method then becomes an *asynchronous iterative method*.

## Asynchronous Iterative Method - Convergence

Mathematical conditions for convergence may be more strict.

Each process may not be allowed to use any previous iteration values if the method is to converge.

## Chaotic Relaxation

A form of asynchronous iterative method introduced by Chazan and Miranker (1969) in which the conditions are stated as “there must be a fixed positive integer  $s$  such that, in carrying out the evaluation of the  $i$ th iterate, a process cannot make use of any value of the components of the  $j$ th iterate if  $j < i - s$ ” (Baudet, 1978).

The final part of the code, checking for convergence of every iteration can also be reduced. It may be better to allow iterations to continue for several iterations before checking for convergence.

## Overall Parallel Code

Each process allowed to perform  $s$  iterations before being synchronized and also to update the array as it goes. At  $s$  iterations, maximum divergence recorded. Convergence is checked then.

The actual iteration corresponding to the elements of the array being used at any time may be from an earlier iteration but only up to  $s$  iterations previously. There may be a mixture of values of different iterations as the array is updated without synchronizing with other processes - truly a **chaotic** situation.

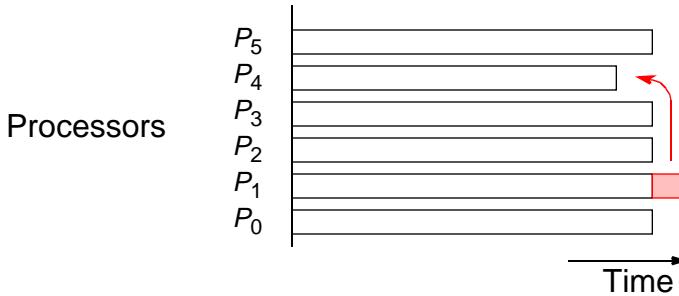
## Chapter 7

# Load Balancing and Termination Detection

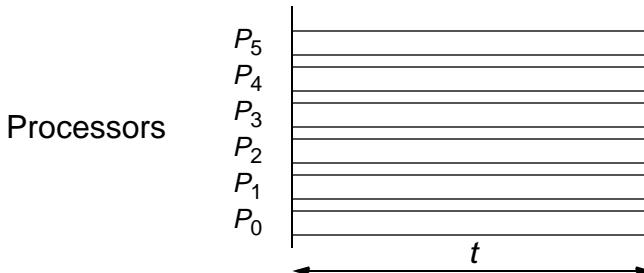
*Load balancing* – used to distribute computations fairly across processors in order to obtain the highest possible execution speed.

*Termination detection* – detecting when a computation has been completed. More difficult when the computation is distributed.

# Load balancing



(a) Imperfect load balancing leading to increased execution time



(b) Perfect load balancing

# Static Load Balancing

Before the execution of any process. Some potential static load-balancing techniques:

- *Round robin algorithm* — passes out tasks in sequential order of processes coming back to the first when all processes have been given a task
- *Randomized algorithms* — selects processes at random to take tasks
- *Recursive bisection* — recursively divides the problem into subproblems of equal computational effort while minimizing message passing
- *Simulated annealing* — an optimization technique
- *Genetic algorithm* — another optimization technique, described in Chapter 12

# Static Load Balancing

Balance load prior to the execution. Various static load-balancing algorithms.

Several **fundamental flaws** with static load balancing even if a mathematical solution exists:

- Very difficult to estimate accurately the execution times of various parts of a program without actually executing the parts.
- Communication delays that vary under different circumstances
- Some problems have an indeterminate number of steps to reach their solution.

## Dynamic Load Balancing

Vary load during the execution of the processes.

All previous factors are taken into account by making the division of load dependent upon the execution of the parts as they are being executed.

Does incur an additional overhead during execution, but it is much more effective than static load balancing

## Processes and Processors

Computation will be divided into *work* or *tasks* to be performed, and *processes* perform these tasks. *Processes* are mapped onto *processors*.

Since our objective is to keep the processors busy, we are interested in the activity of the processors.

However, we often map a single process onto each processor, so we will use the terms *process* and *processor* somewhat interchangeably.

## Dynamic Load Balancing

Can be classified as:

- Centralized
- Decentralized

## Centralized dynamic load balancing

Tasks handed out from a centralized location. Master-slave structure.

## Decentralized dynamic load balancing

Tasks are passed between arbitrary processes.

A collection of worker processes operate upon the problem and interact among themselves, finally reporting to a single process.

A worker process may receive tasks from other worker processes and may send tasks to other worker processes (to complete or pass on at their discretion).

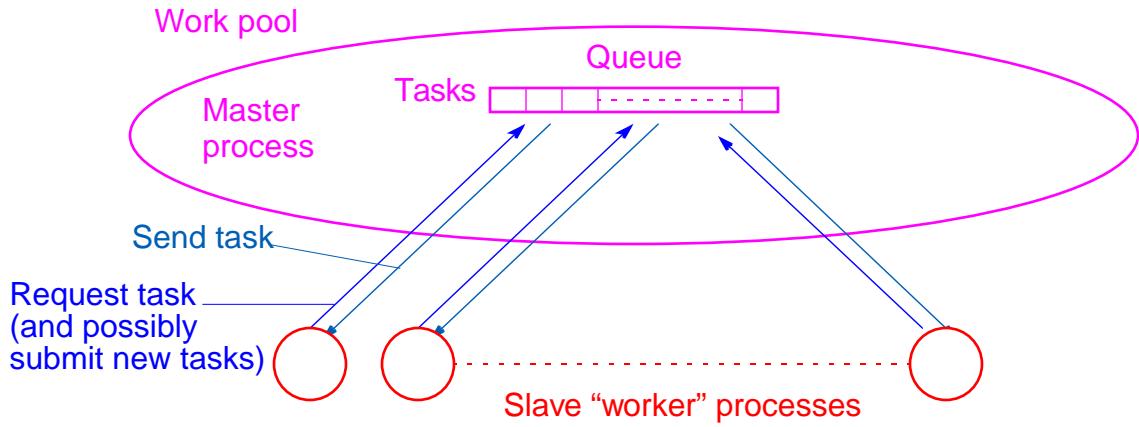
## Centralized Dynamic Load Balancing

Master process(or) holds the collection of tasks to be performed.

Tasks are sent to the slave processes. When a slave process completes one task, it requests another task from the master process.

Terms used : *work pool, replicated worker, processor farm.*

# Centralized work pool



# Termination

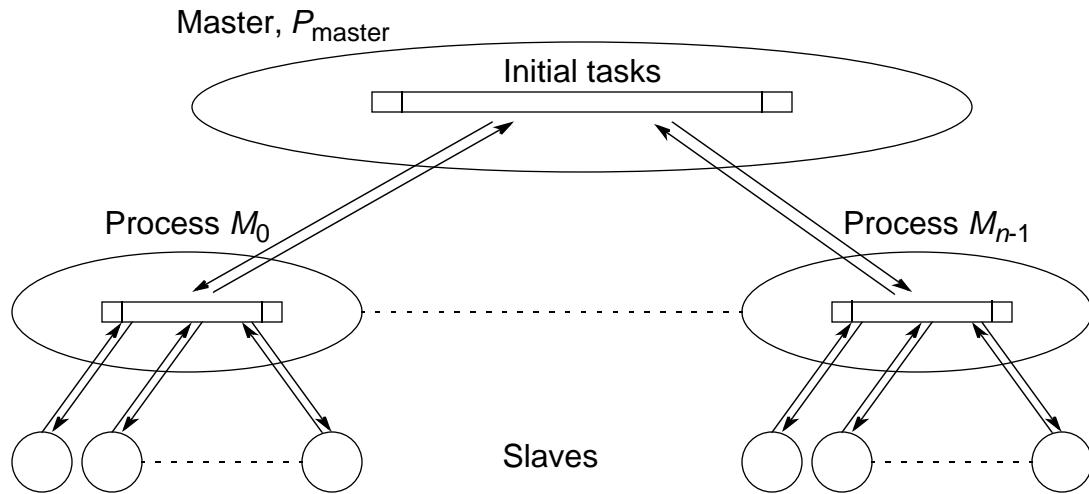
Computation terminates when:

- The task queue is empty and
- Every process has made a request for another task without any new tasks being generated

*Not sufficient* to terminate when task queue empty if one or more processes are still running if a running process may provide new tasks for task queue.

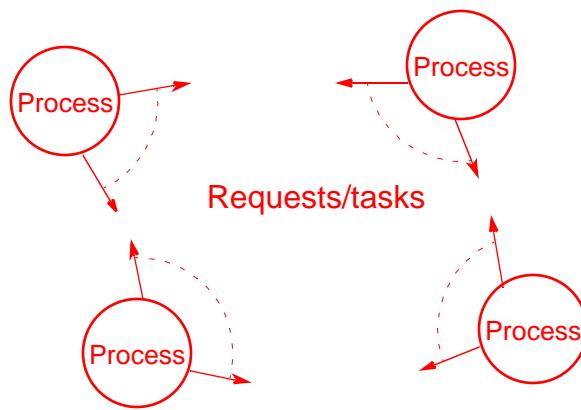
# Decentralized Dynamic Load Balancing

## Distributed Work Pool



# Fully Distributed Work Pool

Processes to execute tasks from each other



# Task Transfer Mechanisms

## Receiver-Initiated Method

A process requests tasks from other processes it selects.

Typically, a process would request tasks from other processes when it has few or no tasks to perform.

Method has been shown to work well at high system load.

Unfortunately, it can be expensive to determine process loads.

## Sender-Initiated Method

A process sends tasks to other processes it selects.

Typically, a process with a heavy load passes out some of its tasks to others that are willing to accept them.

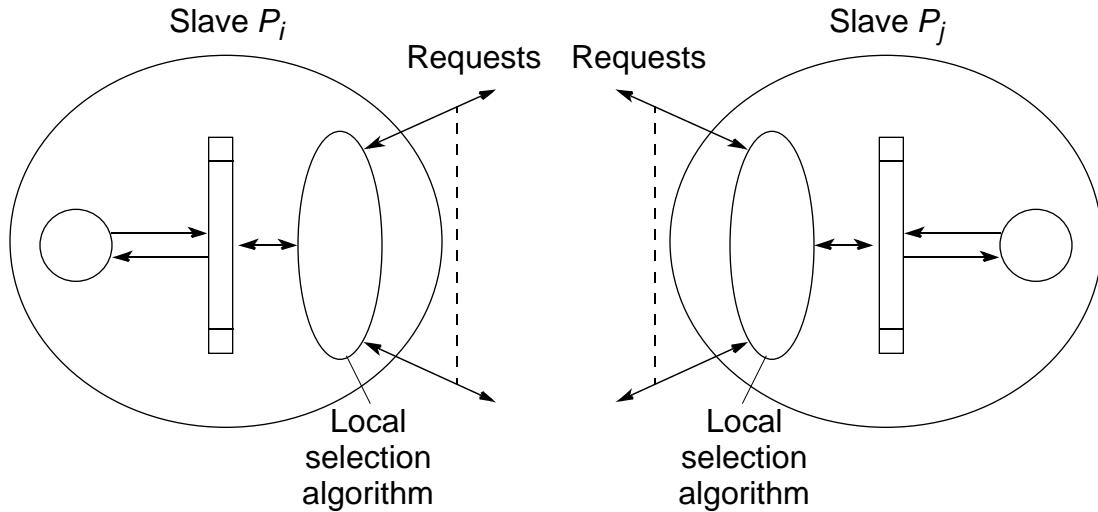
Method has been shown to work well for light overall system loads.

Another option is to have a mixture of both methods.

Unfortunately, it can be expensive to determine process loads.

In very heavy system loads, load balancing can also be difficult to achieve because of the lack of available processes.

# Decentralized selection algorithm requesting tasks between slaves



## Process Selection

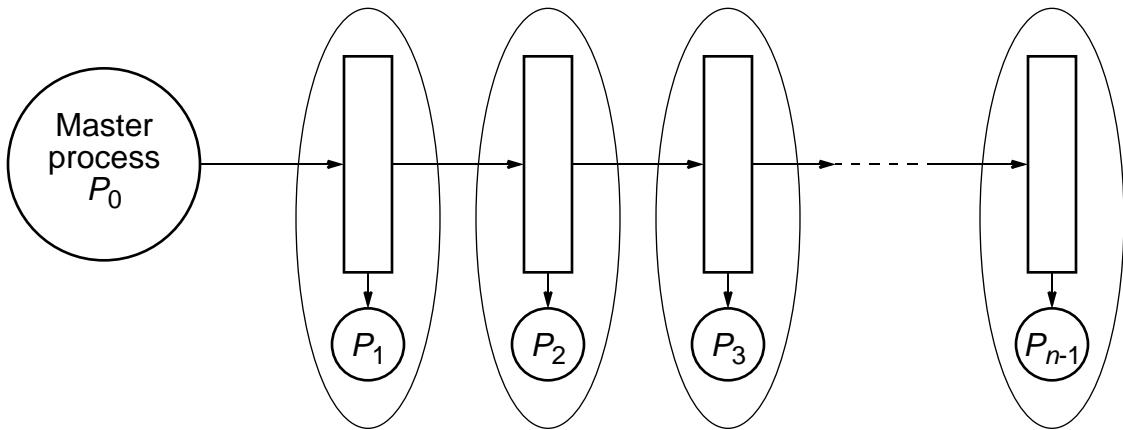
Algorithms for selecting a process:

*Round robin algorithm* – process  $P_i$  requests tasks from process  $P_x$ , where  $x$  is given by a counter that is incremented after each request, using modulo  $n$  arithmetic ( $n$  processes), excluding  $x = i$ .

.

*Random polling algorithm* – process  $P_i$  requests tasks from process  $P_x$ , where  $x$  is a number that is selected randomly between 0 and  $n - 1$  (excluding  $i$ ).

# Load Balancing Using a Line Structure



The master process ( $P_0$  in Figure 7.6) feeds the queue with tasks at one end, and the tasks are shifted down the queue.

When a “worker” process,  $P_i$  ( $1 \leq i < n$ ), detects a task at its input from the queue and the process is idle, it takes the task from the queue.

Then the tasks to the left shuffle down the queue so that the space held by the task is filled. A new task is inserted into the left side end of the queue.

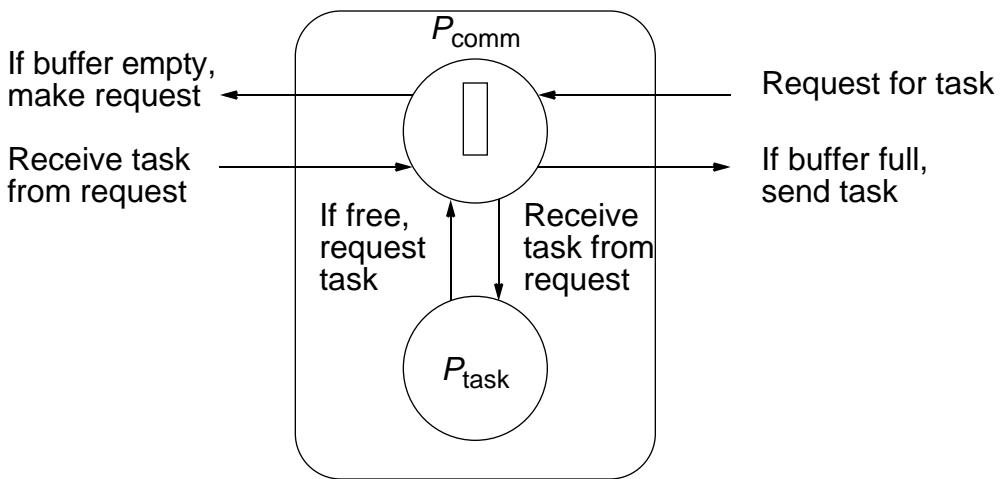
Eventually, all processes will have a task and the queue is filled with new tasks.

High-priority or larger tasks could be placed in the queue first.

# Shifting Actions

could be orchestrated by using messages between adjacent processes:

- For left and right communication
- For the current task



## Code Using Time Sharing Between Communication and Computation

Master process ( $P_0$ )

```
for (i = 0; i < no_tasks; i++) {
    recv(P1, request_tag);          /* request for task */
    send(&task, Pi, task_tag);      /* send tasks into queue */
}
recv(P1, request_tag);          /* request for task */
send(&empty, Pi, task_tag);      /* end of tasks */
```

Process  $P_i$  ( $1 < i < n$ )

```
if (buffer == empty) {
    send(Pi-1, request_tag);      /* request new task */
    recv(&buffer, Pi-1, task_tag);/* task from left proc */
}
if ((buffer == full) && (!busy)) { * get next task *
    task = buffer;                /* get task*/
    buffer = empty;               /* set buffer empty */
    busy = TRUE;                  /* set process busy */
}
nrecv(Pi+1, request_tag, request); /* check msg from right */
if (request && (buffer == full)) {
    send(&buffer, Pi+1);          /* shift task forward */
    buffer = empty;
}
if (busy) {                      /* continue on current task */
    Do some work on task.
    If task finished, set busy to false.
}
```

Nonblocking `nrecv()` is necessary to check for a request being received from the right.

# Nonblocking Receive Routines

## PVM

Nonblocking receive, `pvm_nrecv( )`, returned a value that is zero if no message has been received.

A probe routine, `pvm_probe( )`, could be used to check whether a message has been received without actual reading the message

Subsequently, a normal `recv( )` routine is needed to accept and unpack the message.

# Nonblocking Receive Routines

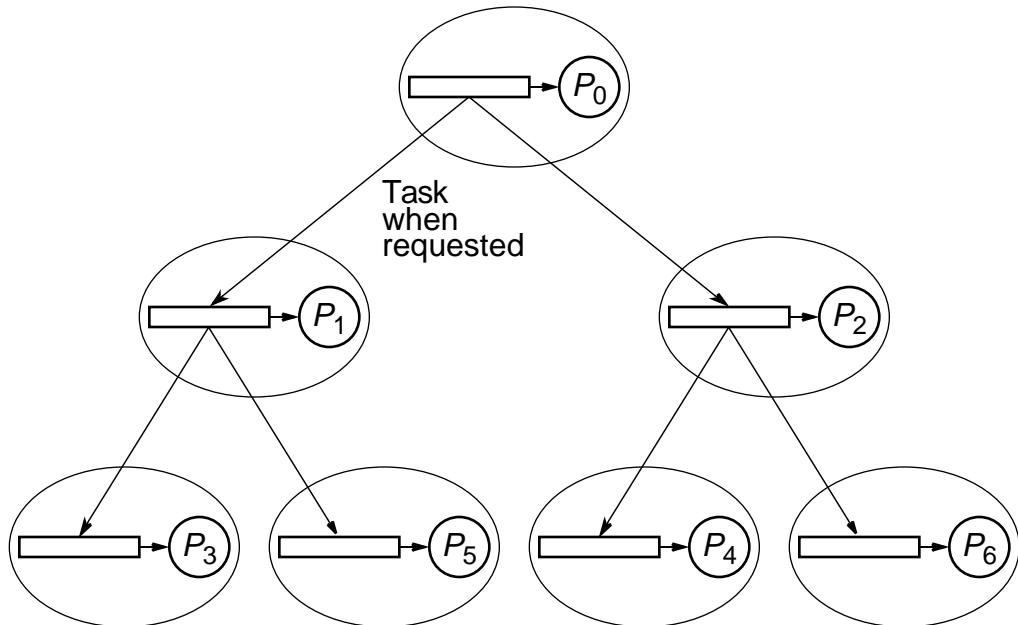
## MPI

Nonblocking receive, `MPI_Irecv()`, returns a request “handle,” which is used in subsequent completion routines to wait for the message or to establish whether the message has actually been received at that point (`MPI_Wait()` and `MPI_Test()`, respectively).

In effect, the nonblocking receive, `MPI_Irecv()`, posts a request for message and returns immediately.

# Load balancing using a tree

Tasks passed from node into one of the two nodes below it when node buffer empty.



# Distributed Termination Detection Algorithms

## Termination Conditions

At time  $t$  requires the following conditions to be satisfied:

- Application-specific local termination conditions exist throughout the collection of processes, at time  $t$ .
- There are no messages in transit between processes at time  $t$ .

Subtle difference between these termination conditions and those given for a centralized load-balancing system is having to take into account messages in transit.

Second condition necessary because a message in transit might restart a terminated process. More difficult to recognize. The time that it takes for messages to travel between processes will not be known in advance.

## **One very general distributed termination algorithm**

Each process in one of two states:

1. Inactive - without any task to perform
2. Active

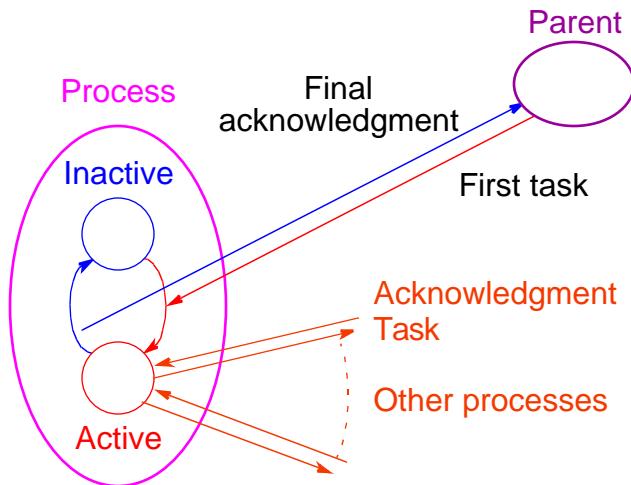
Process that sent task to make it enter the active state becomes its “parent.”

When process receives a task, it immediately sends an acknowledgment message, **except if the process it receives the task from is its parent process.** Only sends an acknowledgment message to its parent when it is ready to become inactive, i.e. when

- Its local termination condition exists (all tasks are completed, *and*)
- It has transmitted all its acknowledgments for tasks it has received, *and*
- It has received all its acknowledgments for tasks it has sent out.

A process must become inactive before its parent process. When first process becomes idle, the computation can terminate.

# Termination using message acknowledgments



Other termination algorithms in textbook.

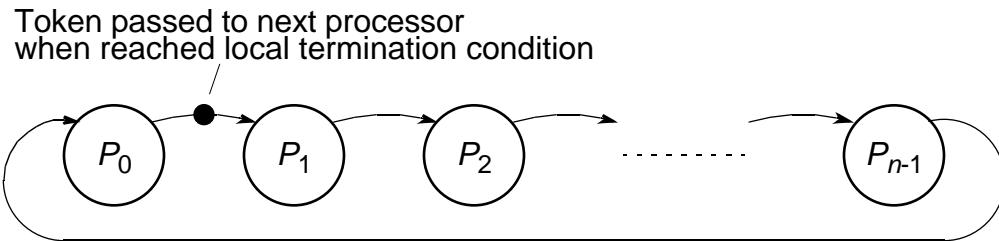
# Ring Termination Algorithms

## Single-pass ring termination algorithm

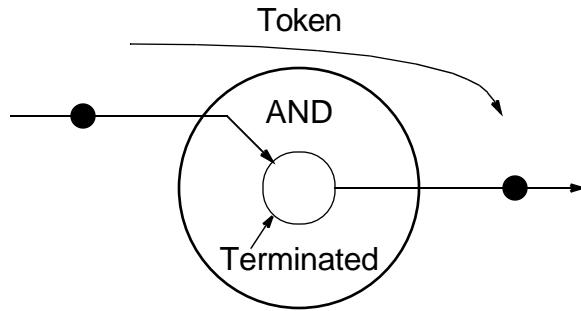
1. When  $P_0$  has terminated, it generates a token that is passed to  $P_1$ .
2. When  $P_i$  ( $1 \leq i < n$ ) receives the token and has already terminated, it passes the token onward to  $P_{i+1}$ . Otherwise, it waits for its local termination condition and then passes the token onward.  $P_{n-1}$  passes the token to  $P_0$ .
3. When  $P_0$  receives a token, it knows that all processes in the ring have terminated. A message can then be sent to all processes informing them of global termination, if necessary.

The algorithm assumes that a process cannot be reactivated after reaching its local termination condition. Does not apply to work pool problems in which a process can pass a new task to an idle process

# Ring termination detection algorithm



# Process algorithm for local termination



## Dual-Pass Ring Termination Algorithm

Can handle processes being reactivated but requires two passes around the ring. The reason for reactivation is for process  $P_j$ , to pass a task to  $P_j$  where  $j < i$  and after a token has passed  $P_j$ . If this occurs, the token must recirculate through the ring a second time.

To differentiate these circumstances, tokens colored white or black.  
Processes are also colored white or black.

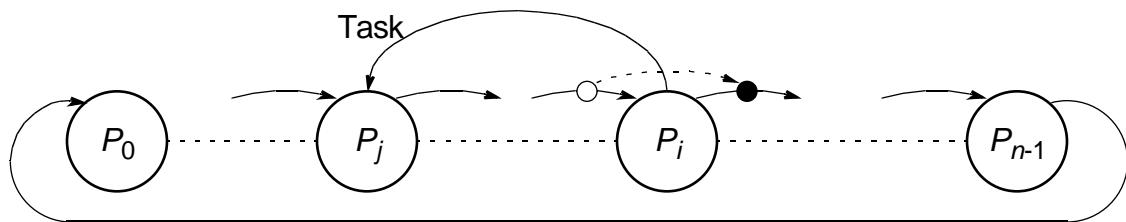
Receiving a black token means that global termination may not have occurred and token must be recirculated around ring again.

The algorithm is as follows, again starting at  $P_0$ :

1.  $P_0$  becomes white when it has terminated and generates a white token to  $P_1$ .
2. The token is passed through the ring from one process to the next when each process has terminated, but the color of the token may be changed. If  $P_i$  passes a task to  $P_j$  where  $j < i$  (that is, before this process in the ring), it becomes a *black process*; otherwise it is a *white process*. A black process will color a token black and pass it on. A white process will pass on the token in its original color (either black or white). After  $P_i$  has passed on a token, it becomes a white process.  $P_{n-1}$  passes the token to  $P_0$ .
3. When  $P_0$  receives a black token, it passes on a white token; if it receives a white token, all processes have terminated.

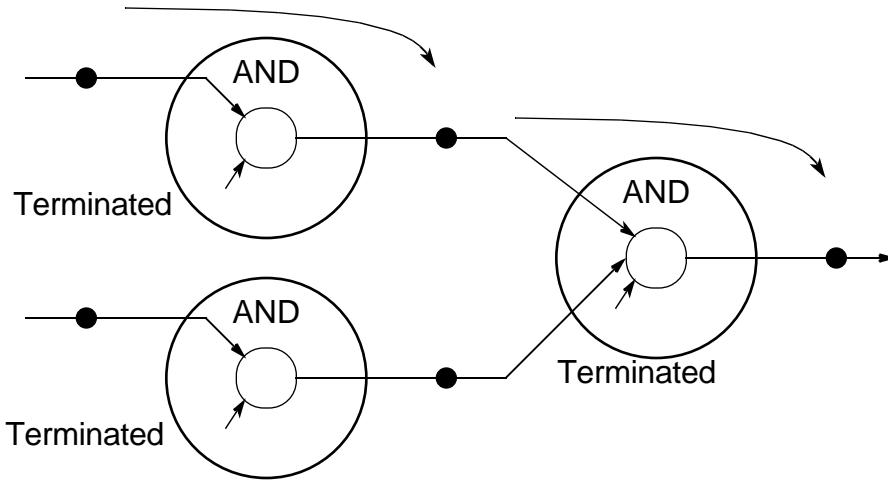
Notice that in both ring algorithms,  $P_0$  becomes the central point for global termination. Also, assumed that an acknowledge signal is generated to each request.

# Passing task to previous processes



# Tree Algorithm

Local actions described can be applied to various structures, notably a tree structure, to indicate that processes up to that point have terminated.



# Fixed Energy Distributed Termination Algorithm

A fixed quantity within system, colorfully termed “energy.”

- System starts with all the energy being held by one process, the root process.
- Root process passes out portions of energy with tasks to processes making requests for tasks.
- If these processes receive requests for tasks, the energy is divided further and passed to these processes.
- When a process becomes idle, it passes the energy it holds back before requesting a new task.
- A process will not hand back its energy until all the energy it handed out is returned and combined to the total energy held.
- When all the energy returned to root and the root becomes idle, all the processes must be idle and the computation can terminate.

Significant disadvantage - dividing energy will be of finite precision and adding partial energies may not equate to original energy. In addition, can only divide energy so far before it becomes essentially zero.

# Load balancing/termination detection Example

## Shortest Path Problem

Finding the shortest distance between two points on a graph.  
It can be stated as follows:

Given a set of interconnected nodes where the links between the nodes are marked with “weights,” find the path from one specific node to another specific node that has the smallest accumulated weights.

The interconnected nodes can be described by a *graph*.

The nodes are called *vertices*, and the links are called *edges*.

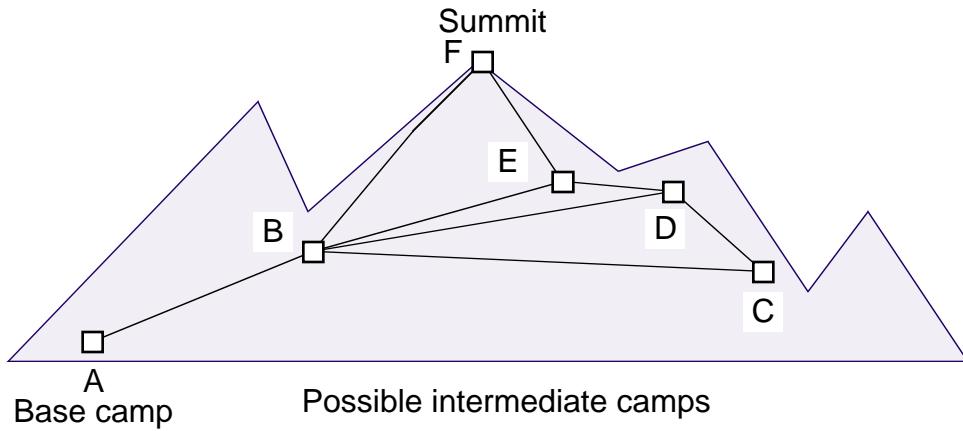
If the edges have implied directions (that is, an edge can only be traversed in one direction, the graph is a *directed graph*.

Graph could be used to find solution to many different problems; eg:

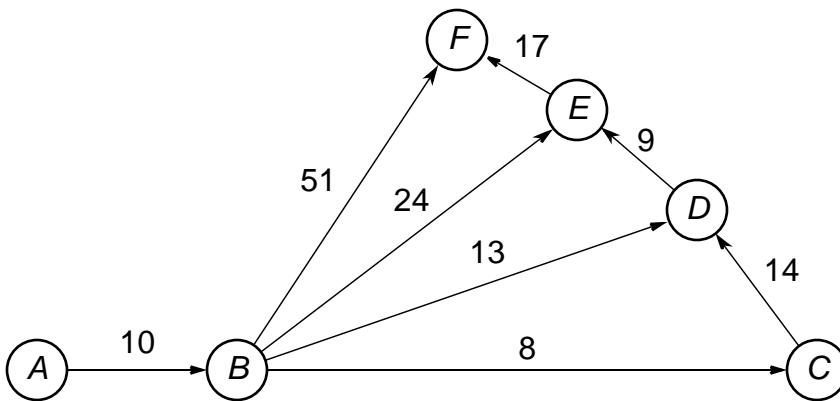
1. The shortest distance between two towns or other points on a map, where the weights represent distance
2. The quickest route to travel, where the weights represent time (the quickest route may not be the shortest route if different modes of travel are available; for example, flying to certain towns)
3. The least expensive way to travel by air, where the weights represent the cost of the flights between cities (the vertices)
4. The best way to climb a mountain given a terrain map with contours
5. The best route through a computer network for minimum message delay (the vertices represent computers, and the weights represent the delay between two computers)
6. The most efficient manufacturing system, where the weights represent hours of work

“The best way to climb a mountain” will be used as an example.

# Example: The Best Way to Climb a Mountain



## Graph of mountain climb



Weights in graph indicate amount of effort that would be expended in traversing the route between two connected camp sites.

The effort in one direction may be different from the effort in the opposite direction (downhill instead of uphill!). (*directed graph*)

# Graph Representation

Two basic ways that a graph can be represented in a program:

1. **Adjacency matrix** — a two-dimensional array,  $a$ , in which  $a[i][j]$  holds the weight associated with the edge between vertex  $i$  and vertex  $j$  if one exists
2. **Adjacency list** — for each vertex, a list of vertices directly connected to the vertex by an edge and the corresponding weights associated with the edges

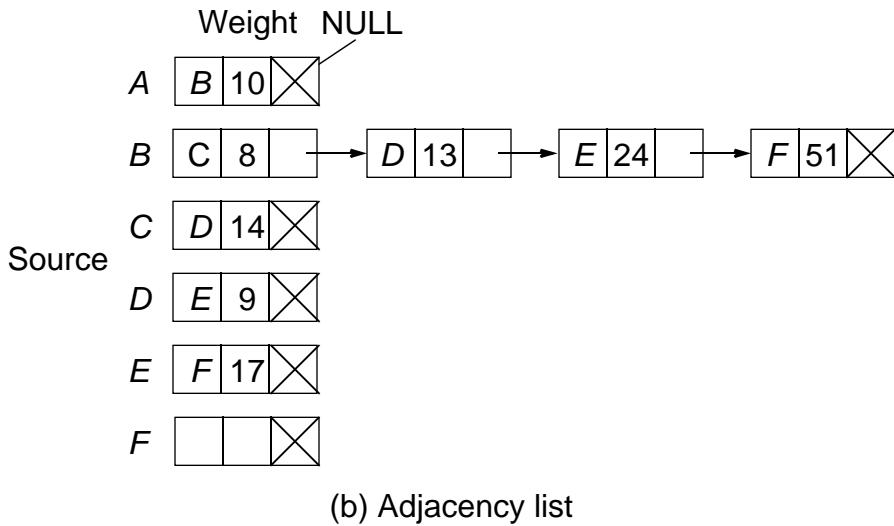
Adjacency matrix used for dense graphs. The adjacency list is used for sparse graphs.

Difference based upon space (storage) requirements. Accessing the adjacency list is slower than accessing the adjacency matrix.

# Representing the graph

		Destination					
		A	B	C	D	E	F
Source	A	•	10	•	•	•	•
	B	•	•	8	13	24	51
	C	•	•	•	14	•	•
	D	•	•	•	•	9	•
	E	•	•	•	•	•	17
	F	•	•	•	•	•	•

(a) Adjacency matrix



# Searching a Graph

Two well-known single-source shortest-path algorithms:

- Moore's single-source shortest-path algorithm (Moore, 1957)
- Dijkstra's single-source shortest-path algorithm (Dijkstra, 1959)

which are similar.

Moore's algorithm is chosen because it is more amenable to parallel implementation although it may do more work.

The weights must all be positive values for the algorithm to work.

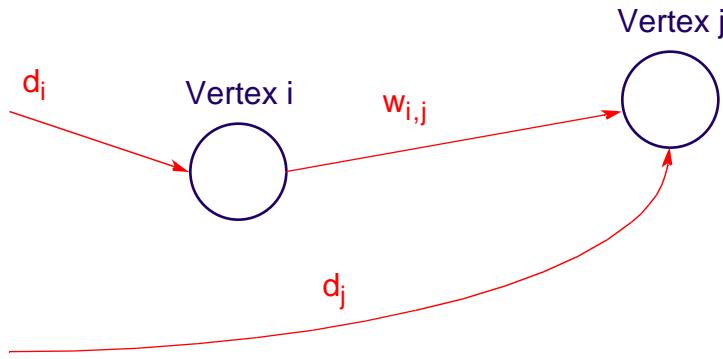
## Moore's Algorithm

Starting with the source vertex, the basic algorithm implemented when vertex  $i$  is being considered as follows.

Find the distance to vertex  $j$  through vertex  $i$  and compare with the current minimum distance to vertex  $j$ . Change the minimum distance if the distance through vertex  $i$  is shorter. If  $d_i$  is the current minimum distance from the source vertex to vertex  $i$  and  $w_{i,j}$  is the weight of the edge from vertex  $i$  to vertex  $j$ :

$$d_j = \min(d_j, d_i + w_{i,j})$$

# Moore's Shortest-path Algorithm



## Date Structures

First-in-first-out vertex queue created to hold a list of vertices to examine. Initially, only source vertex is in queue.

Current shortest distance from source vertex to vertex  $i$  stored in array `dist[i]`. At first, none of these distances known and array elements are initialized to infinity.

## Code

Suppose  $w[i][j]$  holds the weight of the edge from vertex  $i$  and vertex  $j$  (infinity if no edge). The code could be of the form

```
newdist_j = dist[i] + w[i][j];  
if (newdist_j < dist[j]) dist[j] = newdist_j;
```

When a shorter distance is found to vertex  $j$ , vertex  $j$  is added to the queue (if not already in the queue), which will cause vertex  $j$  to be examined again - **Important aspect of this algorithm, which is not present in Dijkstra's algorithm.**

# Stages in Searching a Graph

## Example

The initial values of the two key data structures are

Vertices to consider

A					
---	--	--	--	--	--

vertex\_queue

Current minimum distances

0	•	•	•	•	•	
vertex	A	B	C	D	E	F

dist[]

After examining  $A$  to

Vertices to consider

$B$					
-----	--	--	--	--	--

←

vertex\_queue

Current minimum distances

vertex	$A$	$B$	$C$	$D$	$E$	$F$
	0	10	•	•	•	•
dist[]						

After examining  $B$  to  $F, E, D$ , and  $C$ :

Vertices to consider

$E$	$D$	$C$			
-----	-----	-----	--	--	--



vertex\_queue

Current minimum distances

	0	10	18	23	34	61
vertex	$A$	$B$	$C$	$D$	$E$	$F$

dist[]

After examining  $E$  to  $F$

Vertices to consider

$D$	$C$				
-----	-----	--	--	--	--

←

vertex\_queue

Current minimum distances

	0	10	18	23	34	50
vertex	$A$	$B$	$C$	$D$	$E$	$F$

dist[]

After examining  $D$  to  $E$ :

Vertices to consider

C	E				
---	---	--	--	--	--



vertex\_queue

Current minimum distances

vertex	A	B	C	D	E	F
dist[]	0	10	18	23	32	50

After examining  $C$  to  $D$ : No changes.

After examining  $E$  (again) to  $F$ :

Vertices to consider

--	--	--	--	--	--

vertex\_queue

Current minimum distances

0	10	18	23	32	49
---	----	----	----	----	----

vertex    A    B    C    D    E    F

dist[]

No more vertices to consider. We have the minimum distance from vertex  $A$  to each of the other vertices, including the destination vertex,  $F$ .

Usually, the actual path is also required in addition to the distance. Then the path needs to be stored as distances are recorded. The path in our case is  $A \quad B \quad D \quad E \quad F$

## Sequential Code

Let `next_vertex()` return the next vertex from the vertex queue or `no_vertex` if none.

Assume that adjacency matrix used, named `w[ ][ ]`.

```
while ((i = next_vertex()) != no_vertex)* while a vertex */
for (j = 1; j < n; j++)           /* get next edge */
    if (w[i][j] != infinity) {      /* if an edge */
        newdist_j = dist[i] + w[i][j];
        if (newdist_j < dist[j]) {
            dist[j] = newdist_j;
            append_queue(j);        /* add to queue if not there */
        }
    }                                /*no more to consider*/
```

# Parallel Implementations

## Centralized Work Pool

Centralized work pool holds vertex queue, `vertex_queue[ ]` as tasks.

Each slave takes vertices from the vertex queue and returns new vertices.

Since the structure holding the graph weights is fixed, this structure could be copied into each slave, say a copied adjacency matrix.

## Master

```
while (vertex_queue() != empty) {
    recv(PANY, source = Pi);      /* request task from slave */
    v = get_vertex_queue();
    send(&v, Pi);                  /* send next vertex and */
    send(&dist, &n, Pi);          /* current dist array */
    .
    recv(&j, &dist[j], PANY, source = Pi);/* new distance */
    append_queue(j, dist[j]); /* append vertex to queue */
    /* and update distance array */
};

recv(PANY, source = Pi);      /* request task from slave */
send(Pi, termination_tag); /* termination message*/
```

## Slave (process $i$ )

```
send(Pmaster);           /* send request for task */
recv(&v, Pmaster, tag);    /* get vertex number */
if (tag != termination_tag) {
    recv(&dist, &n, Pmaster);    /* and dist array */
    for (j = 1; j < n; j++) /* get next edge */
        if (w[v][j] != infinity) /* if an edge */
            newdist_j = dist[v] + w[v][j];
            if (newdist_j < dist[j]) {
                dist[j] = newdist_j;
                send(&j, &dist[j], Pmaster);/* add vertex to queue */
                /* send updated distance */
            }
    }
}
```

## Decentralized Work Pool

Convenient approach is to assign slave process  $i$  to search around vertex  $i$  only and for it to have the vertex queue entry for vertex  $i$  if this exists in the queue.

The array `dist[ ]` will also be distributed among the processes so that process  $i$  maintains the current minimum distance to vertex  $i$ .

Process also stores an adjacency matrix/list for vertex  $i$ , for the purpose of identifying the edges from vertex  $i$ .

## Search Algorithm

Vertex A is the first vertex to search. The process assigned to vertex A is activated.

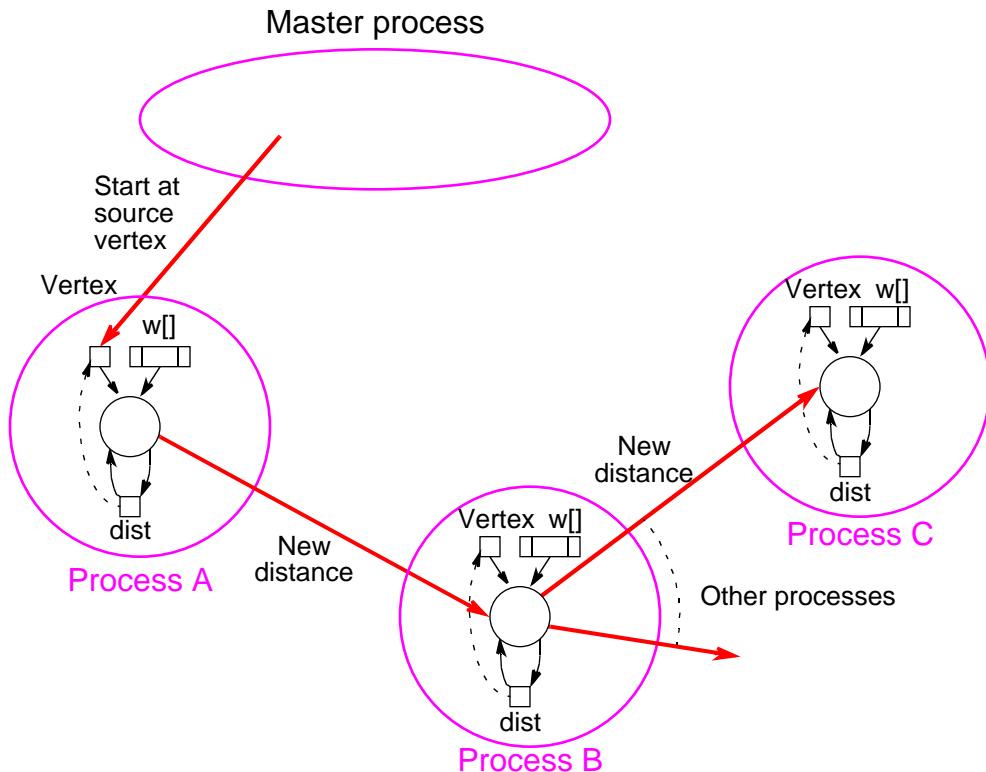
This process will search around its vertex to find distances to connected vertices.

Distance to process  $j$  will be sent to process  $j$  for it to compare with its currently stored value and replace if the currently stored value is larger.

In this fashion, all minimum distances will be updated during the search.

If the contents of  $d[i]$  changes, process  $i$  will be reactivated to search again.

# Distributed graph search



## Slave (process $i$ )

```
recv(newdist, PANY);
if (newdist < dist) {
    dist = newdist;
    vertex_queue = TRUE;           /* add to queue */
} else vertex_queue == FALSE;
if (vertex_queue == TRUE)/*start searching around vertex*/
    for (j = 1; j < n; j++)      /* get next edge */
        if (w[j] != infinity) {
            d = dist + w[j];
            send(&d, Pj);          /* send distance to proc j */
        }
}
```

## Simplified slave (process $i$ )

```
recv(newdist, PANY);
if (newdist < dist)
    dist = newdist;          /* start searching around vertex */
for (j = 1; j < n; j++) /* get next edge */
    if (w[j] != infinity) {
        d = dist + w[j];
        send(&d, Pj);      /* send distance to proc j */
    }
```

Mechanism necessary to repeat actions and terminate when all processes idle - must cope with messages in transit.

## Simplest solution

Use synchronous message passing, in which a process cannot proceed until the destination has received the message.

Process only active after its vertex is placed on queue. Possible for many processes to be inactive, leading to an inefficient solution.

Method also impractical for a large graph if one vertex is allocated to each processor. Group of vertices could be allocated to each processor.

## Chapter 6

# Synchronous Computations

# Synchronous Computations

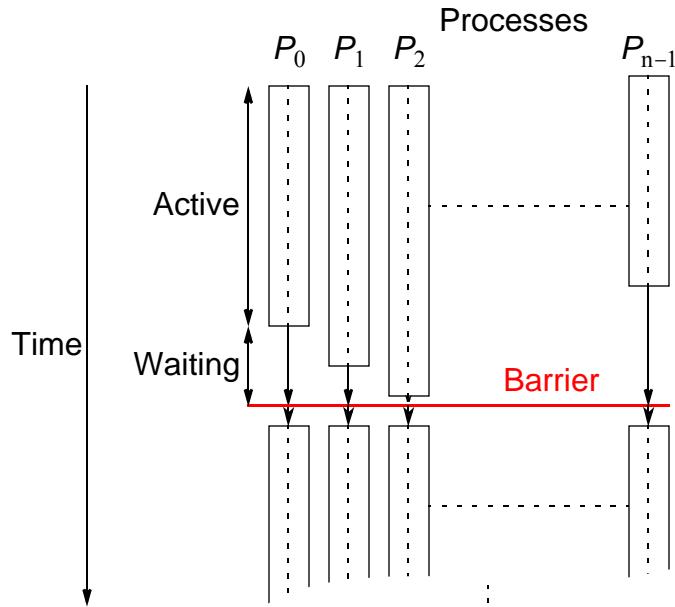
In a (fully) synchronous application, all the processes synchronized at regular points.

## Barrier

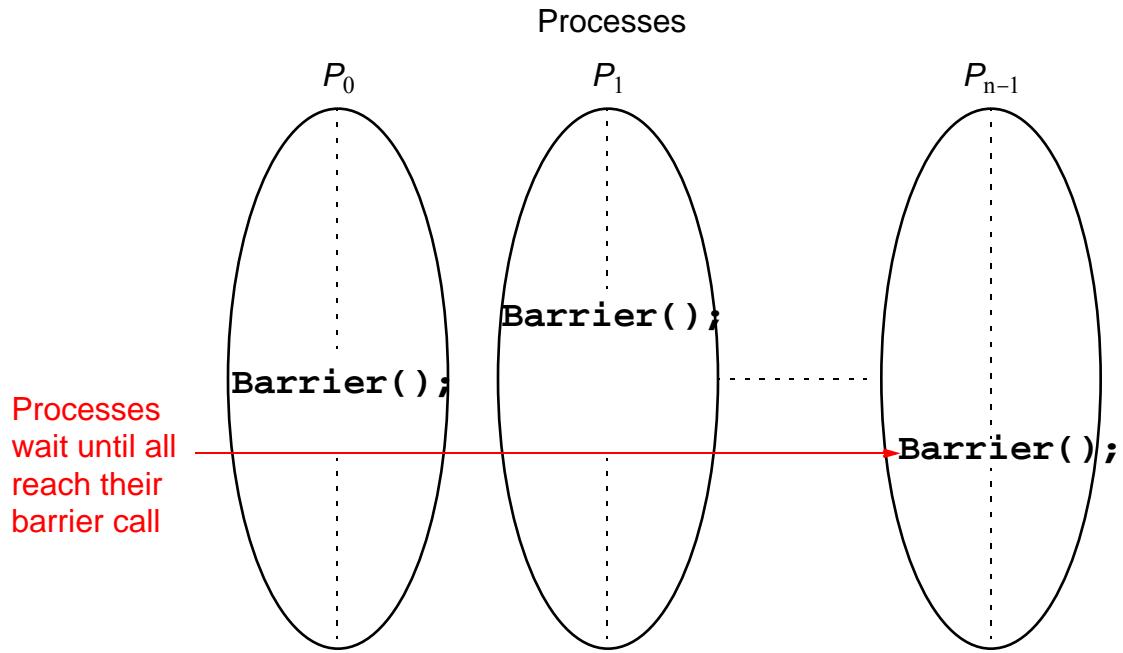
A basic mechanism for synchronizing processes - inserted at the point in each process where it must wait.

All processes can continue from this point when all the processes have reached it (or, in some implementations, when a stated number of processes have reached this point).

# Processes reaching barrier at different times



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



# MPI

## `MPI_Barrier()`

Barrier with a named communicator being the only parameter.  
called by each process in the group, blocking until all members of  
the group have reached the barrier call and only returning then.

# PVM

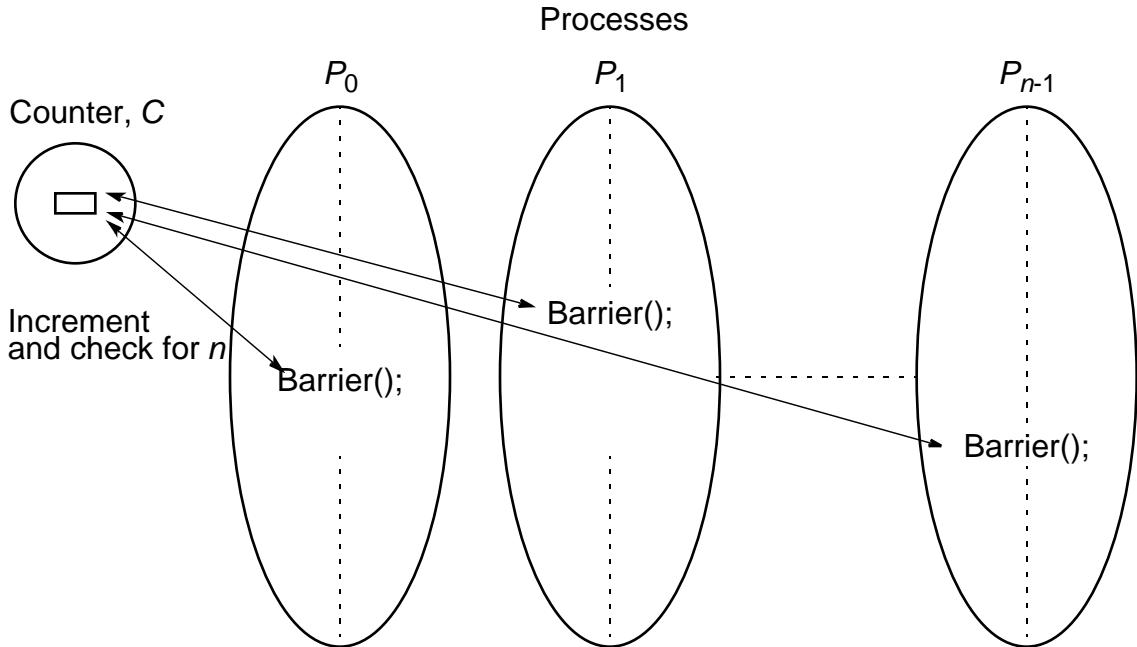
`pvm_barrier()`

similar barrier routine used with a named group of processes.

PVM has the unusual feature of specifying the number of processes that must reach the barrier to release the processes.

# Barrier Implementation

Centralized counter implementation (*linear barrier*):



Counter-based barriers often have two phases:

- A process enters arrival phase and does not leave this phase until all processes have arrived in this phase.
- Then processes move to departure phase and are released.

Good implementations must take into account that a barrier might be used more than once in a process. Might be possible for a process to enter barrier for a second time before previous processes have left barrier for the first time. Two-phase handles this scenario.

## Example code:

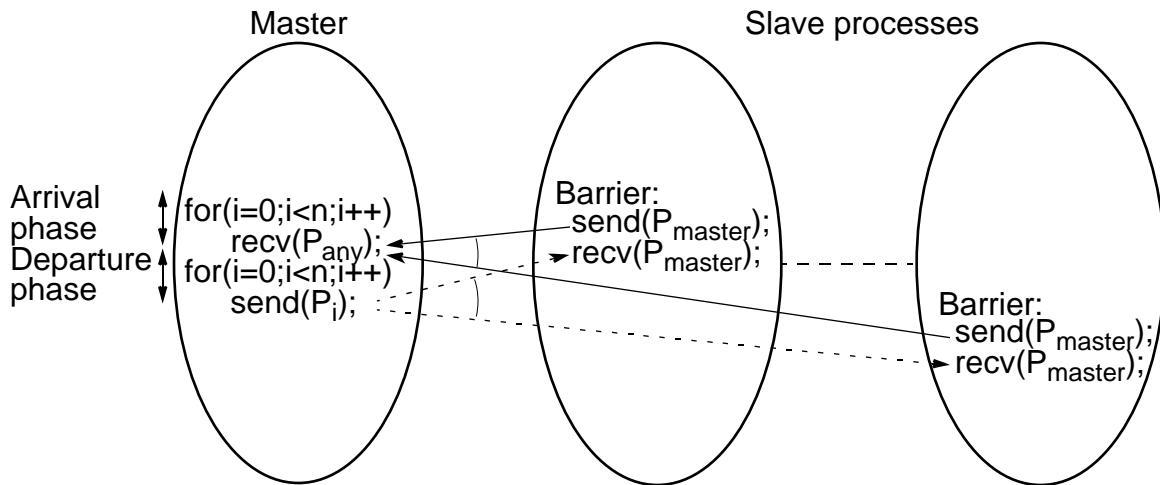
### Master:

```
for (i = 0; i < n; i++)/*count slaves as they reach  
barrier*/  
    recv(Pany);  
for (i = 0; i < n; i++)/* release slaves */  
    send(Pi);
```

### Slave processes:

```
send(Pmaster);  
recv(Pmaster);
```

# Barrier implementation in a message-passing system



# Tree Implementation

More efficient. Suppose 8 processes,  $P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7$ :

First stage:  $P_1$  sends message to  $P_0$ ; (when  $P_1$  reaches its barrier)

$P_3$  sends message to  $P_2$ ; (when  $P_3$  reaches its barrier)

$P_5$  sends message to  $P_4$ ; (when  $P_5$  reaches its barrier)

$P_7$  sends message to  $P_6$ ; (when  $P_7$  reaches its barrier)

Second stage:  $P_2$  sends message to  $P_0$ ; ( $P_2$  &  $P_3$  reached their barrier)

$P_6$  sends message to  $P_4$ ; ( $P_6$  &  $P_7$  reached their barrier)

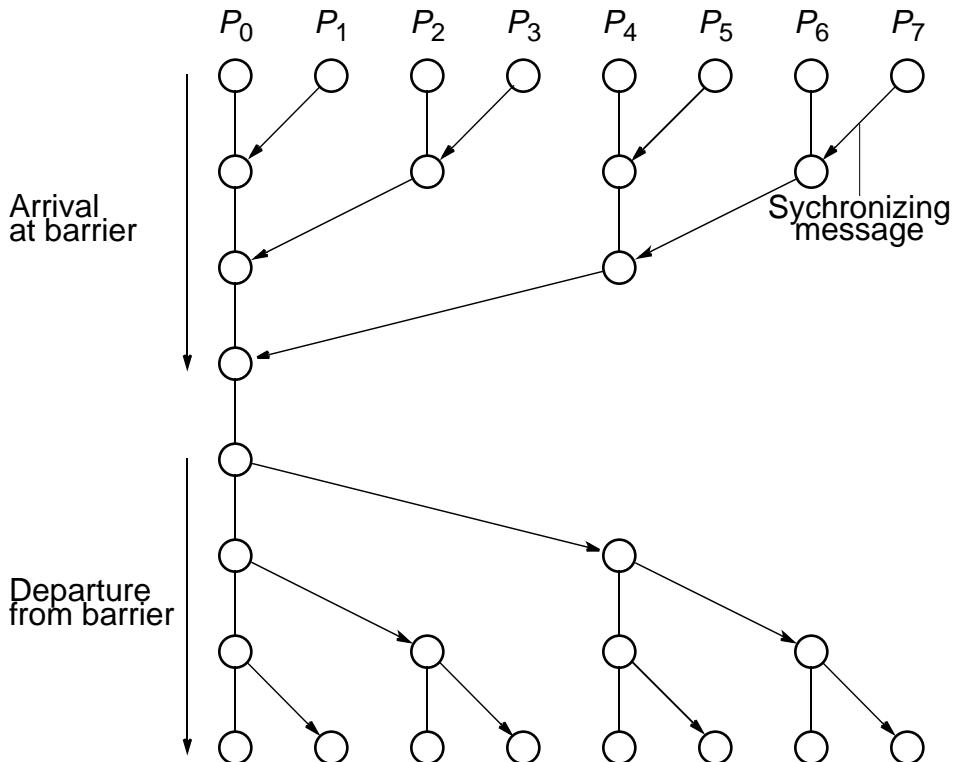
Third stage:  $P_4$  sends message to  $P_0$ ; ( $P_4, P_5, P_6$ , &  $P_7$  reached barrier)

$P_0$  terminates arrival phase;

(when  $P_0$  reaches barrier & received message from  $P_4$ )

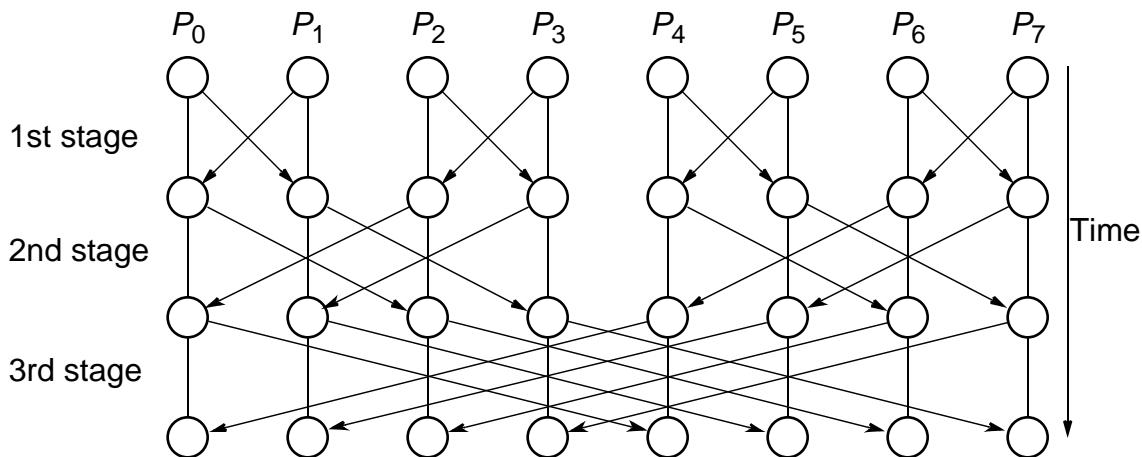
Release with a reverse tree construction.

## Tree Barrier



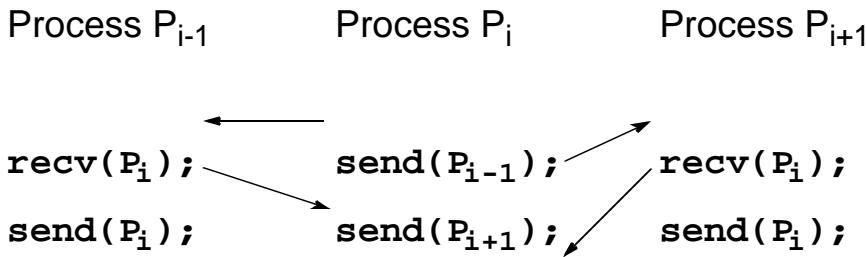
# Butterfly Barrier

First stage	$P_0$	$P_1, P_2$	$P_3, P_4$	$P_5, P_6$	$P_7$
Second stage	$P_0$	$P_2, P_1$	$P_3, P_4$	$P_6, P_5$	$P_7$
Third stage	$P_0$	$P_4, P_1$	$P_5, P_2$	$P_6, P_3$	$P_7$



## Local Synchronization

Suppose a process  $P_i$  needs to be synchronized and to exchange data with process  $P_{i-1}$  and process  $P_{i+1}$  before continuing:



Not a perfect three-process barrier because process  $P_{i-1}$  will only synchronize with  $P_i$  and continue as soon as  $P_i$  allows. Similarly, process  $P_{i+1}$  only synchronizes with  $P_i$ .

## Deadlock

When a pair of processes each send and receive from each other, deadlock may occur.

Deadlock will occur if both processes perform the send, using synchronous routines first (or blocking routines without sufficient buffering). This is because neither will return; they will wait for matching receives that are never reached.

## A Solution

Arrange for one process to receive first and then send and the other process to send first and then receive.

### Example

Linear pipeline, deadlock can be avoided by arranging so the even-numbered processes perform their sends first and the odd-numbered processes perform their receives first.

# Combined deadlock-free blocking sendrecv( ) routines

MPI provides **`MPI_Sendrecv()`** and **`MPI_Sendrecv_replace()`**.

## Example

Process  $P_{i-1}$

Process  $P_i$

Process  $P_{i+1}$

```
sendrecv( $P_i$ );  $\leftrightarrow$  sendrecv( $P_{i-1}$ );  
          sendrecv( $P_{i+1}$ );  $\leftrightarrow$  sendrecv( $P_i$ );
```

**sendrev()s have 12 parameters!**

# Synchronized Computations

Can be classified as:

- Fully synchronous

or

- Locally synchronous

In fully synchronous, all processes involved in the computation must be synchronized.

In locally synchronous, processes only need to synchronize with a set of logically nearby processes, not all processes involved in the computation

# Fully Synchronized Computation Examples

## Data Parallel Computations

Same operation performed on different data elements simultaneously; i.e., in parallel.

Particularly convenient because:

- Ease of programming (essentially only one program).
- Can scale easily to larger problem sizes.
- Many numeric and some non-numeric problems can be cast in a data parallel form.

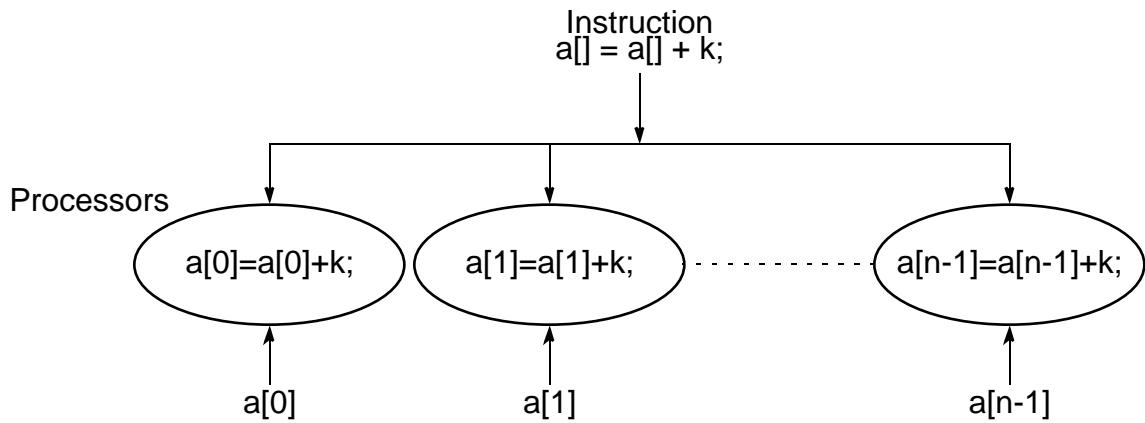
## Example

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

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

The statement `a[i] = a[i] + k` could be executed simultaneously by multiple processors, each using a different index `i (0 < i n)`.

# Data Parallel Computation



## Forall construct

Special “parallel” construct in parallel programming languages to specify data parallel operations

### Example

```
forall (i = 0; i < n; i++) {  
    body  
}
```

states that  $n$  instances of the statements of the body can be executed simultaneously.

One value of the loop variable  $i$  is valid in each instance of the body, the first instance has  $i = 0$ , the next  $i = 1$ , and so on.

To add **k** to each element of an array, **a**, we can write

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

Data parallel technique applied to multiprocessors and multiccomputers.

## Example

To add **k** to the elements of an array:

```
i = myrank;
a[i] = a[i] + k; /* body */
barrier(mygroup);
```

where **myrank** is a process rank between 0 and  $n - 1$ .

## Data Parallel Example - Prefix Sum Problem

Given a list of numbers,  $x_0, \dots, x_{n-1}$ , compute all the partial summations (i.e.,  $x_0 + x_1; x_0 + x_1 + x_2; x_0 + x_1 + x_2 + x_3; \dots$  ).

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.

# Data parallel method of adding all partial sums of 16 numbers

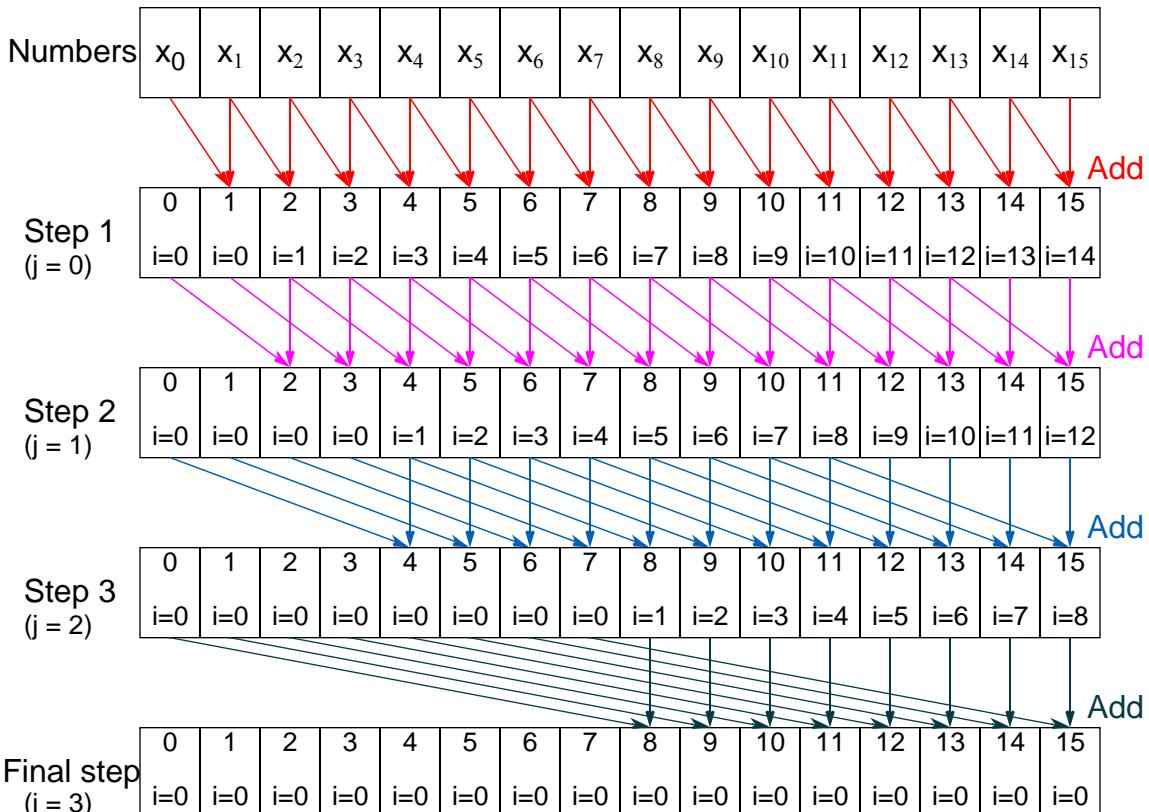
## Sequential code

```
for (j = 0; j < log(n); j++)/* at each step, add*/
    for (i = 2j; i < n; i++)/* to accumulating sum */
        x[i] = x[i] + x[i - 2j];
```

## Parallel code

```
for (j = 0; j < log(n); j++) /* at each step, add */
    forall (i = 0; i < n; i++)/*to sum */
        if (i >= 2j) x[i] = x[i] + x[i - 2j];
```

# Data parallel prefix sum operation



## Synchronous Iteration (Synchronous Parallelism)

Each iteration composed of several processes that start together at beginning of iteration. Next iteration cannot begin until all processes have finished previous iteration. Using **forall**:

```
for (j = 0; j < n; j++) /*for each synch. iteration */
    forall (i = 0; i < N; i++) /*N procs each using*/
        body(i);                /* specific value of i */
    }
```

or:

```
for (j = 0; j < n; j++) /*for each synchr. iteration */
    i = myrank;           /*find value of i to be used */
    body(i);
    barrier(mygroup);
}
```

# 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

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \dots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

.

.

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \dots + a_{2,n-1}x_{n-1} = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \dots + a_{1,n-1}x_{n-1} = b_1$$

$$a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \dots + a_{0,n-1}x_{n-1} = b_0$$

where the unknowns are  $x_0, x_1, x_2, \dots, x_{n-1}$  ( $0 \leq i < n$ ).

By rearranging the  $i$ th equation:

$$a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 \dots + a_{i,n-1}x_{n-1} = b_i$$

to

$$x_i = (1/a_{i,i})[b_i - (a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 \dots a_{i,i-1}x_{i-1} + a_{i,i+1}x_{i+1} \dots + a_{i,n-1}x_{n-1})]$$

or

$$x_i = \frac{1}{a_{i,i}} \left[ b_i - \sum_{j \neq i} a_{i,j} x_j \right]$$

This equation gives  $x_i$  in terms of the other unknowns and can be used as an iteration formula for each of the unknowns to obtain better approximations.

## Jacobi Iteration

All values of  $x$  are updated **together**.

Can be proven that the Jacobi method will converge if the diagonal values of  $a$  have an absolute value greater than the sum of the absolute values of the other  $a$ 's on the row (the array of  $a$ 's is *diagonally dominant*) i.e. if

$$\sum_{j \neq i} |a_{i,j}| < |a_{i,i}|$$

This condition is a sufficient but not a necessary condition.

## Termination

A simple, common approach. Compare values computed in one iteration to values obtained from the previous iteration.

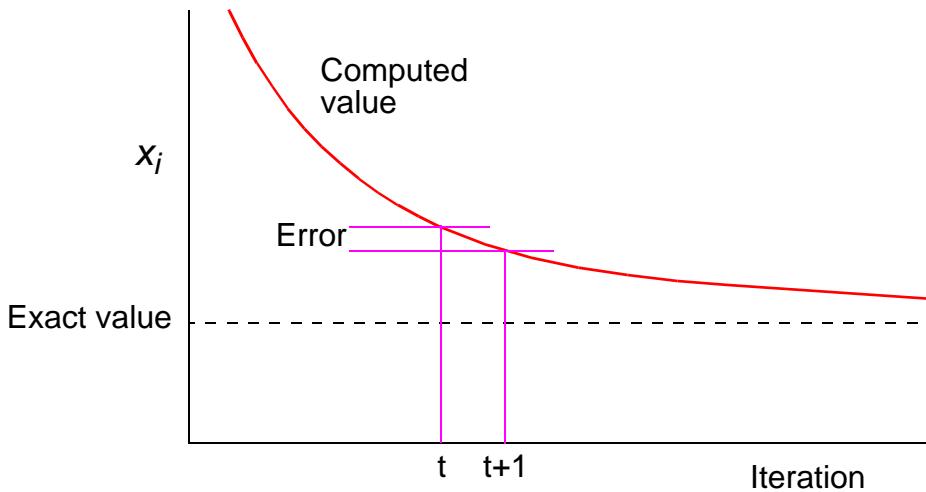
Terminate computation when all values are within given tolerance;  
i.e., when

$$\left| x_i^t - x_i^{t-1} \right| < \text{error tolerance}$$

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.

# Convergence Rate



# Parallel Code

Process  $P_i$  could be of the form

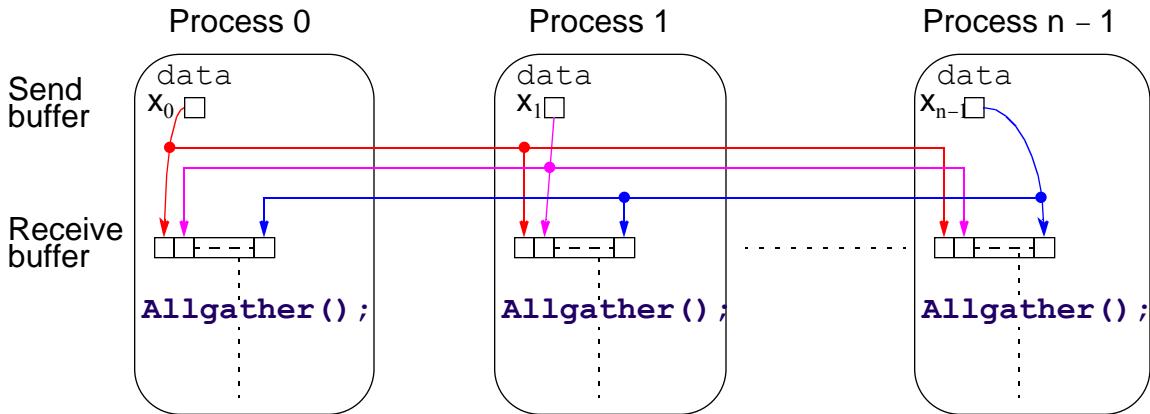
```
x[i] = b[i];                                /*initialize unknown*/
for (iteration = 0; iteration < limit; iteration++) {
    sum = -a[i][i] * x[i];
    for (j = 0; j < n; j++)          /* compute summation */
        sum = sum + a[i][j] * x[j];
    new_x[i] = (b[i] - sum) / a[i][i];/* compute unknown */
    allgather(&new_x[i]);           /*bcast/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.

Introduce a new message-passing operation - Allgather.

## Allgather

Broadcast and gather values in one composite construction.



# Partitioning

Usually number of processors much fewer than number of data items to be processed. Partition the problem so that processors take on more than one data item.

*block allocation* – allocate groups of consecutive unknowns to processors in increasing order.

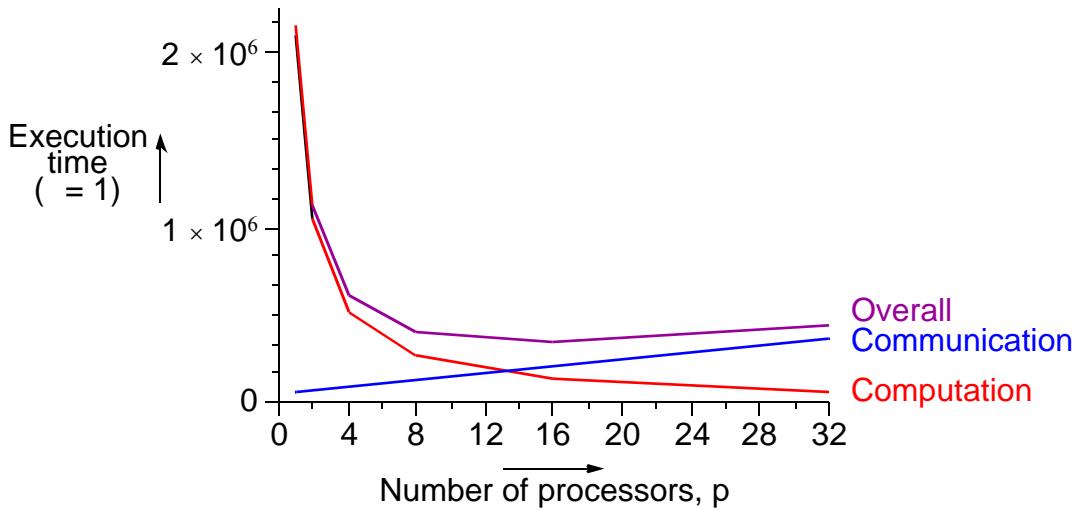
*cyclic allocation* – processors are allocated one unknown in order; i.e., processor  $P_0$  is allocated  $x_0, x_p, x_{2p}, \dots, x_{((n/p)-1)p}$ , processor  $P_1$  is allocated  $x_1, x_{p+1}, x_{2p+1}, \dots, x_{((n/p)-1)p+1}$ , and so on.

Cyclic allocation no particular advantage here (Indeed, may be disadvantageous because the indices of unknowns have to be computed in a more complex way).

# Effects of computation and communication in Jacobi iteration

Consequences of different numbers of processors done in textbook.

Get:



# **Locally Synchronous Computation**

## **Heat Distribution Problem**

An area has known temperatures along each of its edges. Find the temperature distribution within.

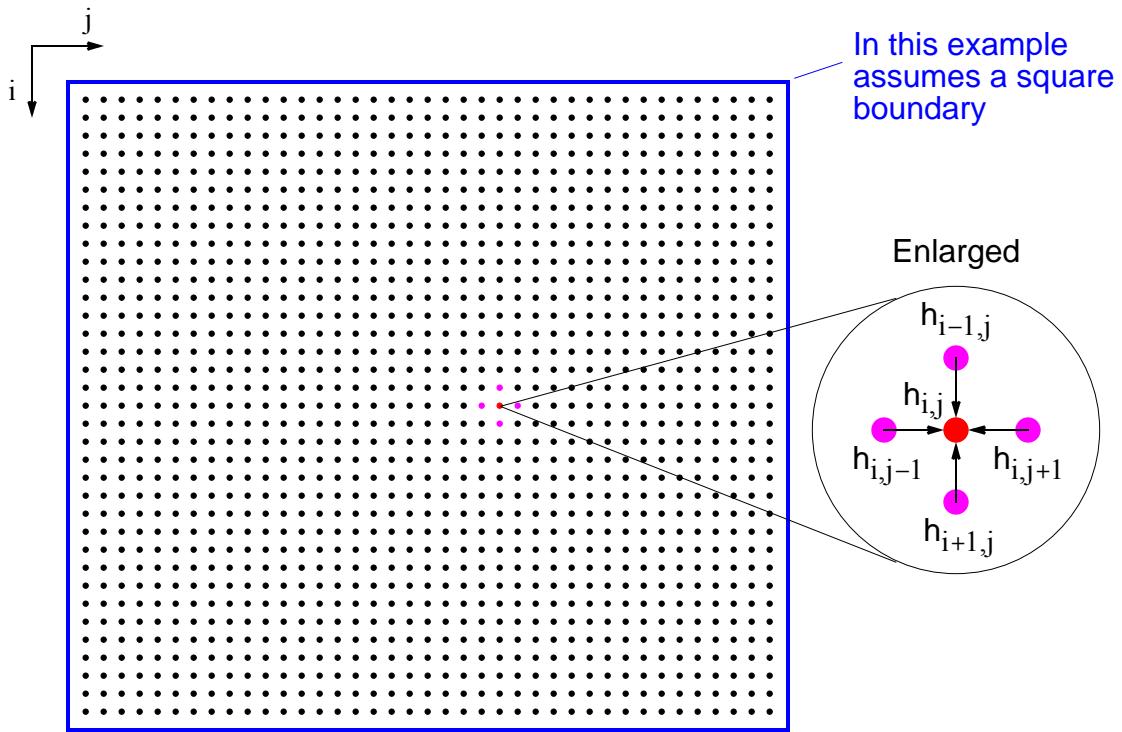
Divide area into fine mesh of points,  $h_{i,j}$ . Temperature at an inside point taken to be average of temperatures of four neighboring points. Convenient to describe edges by points.

Temperature of each point by iterating the equation:

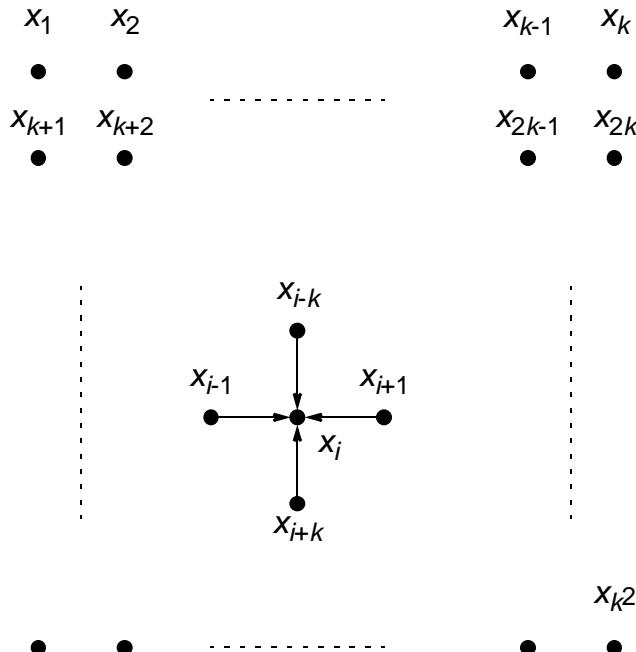
$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

( $0 < i < n$ ,  $0 < j < n$ ) for a fixed number of iterations or until the difference between iterations less than some very small amount.

# Heat Distribution Problem



# Natural ordering of heat distribution problem



Number points from 1 for convenience and include those representing the edges. Each point will then use the equation

$$x_i = \frac{x_{i-1} + x_{i+1} + x_{i-k} + x_{i+k}}{4}$$

Could be written as a linear equation containing the unknowns  $x_{i-k}$ ,  $x_{i-1}$ ,  $x_{i+1}$ , and  $x_{i+k}$ :

$$x_{i-k} + x_{i-1} - 4x_i + x_{i+1} + x_{i+k} = 0$$

Notice: solving a (sparse) system of linear equations.

Also solving Laplace's equation.

# Sequential Code

Using a fixed number of iterations

```
for (iteration = 0; iteration < limit; iteration++) {  
    for (i = 1; i < n; i++)  
        for (j = 1; j < n; j++)  
            g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-  
1]+h[i][j+1]);  
        for (i = 1; i < n; i++)/* update points */  
            for (j = 1; j < n; j++)  
                h[i][j] = g[i][j];  
}
```

To stop at some precision:

```
do {
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);

    for (i = 1; i < n; i++)/* update points */
        for (j = 1; j < n; j++)
            h[i][j] = g[i][j];

    continue = FALSE;      /* indicates whether to continue */
    for (i = 1; i < n; i++)/* check each pt for convergence */
        for (j = 1; j < n; j++)
            if (!converged(i,j) /* point found not converged */
                continue = TRUE;
                break;
    }

} while (continue == TRUE);
```

## Parallel Code

With fixed number of iterations,  $P_{i,j}$  (except for the boundary points):

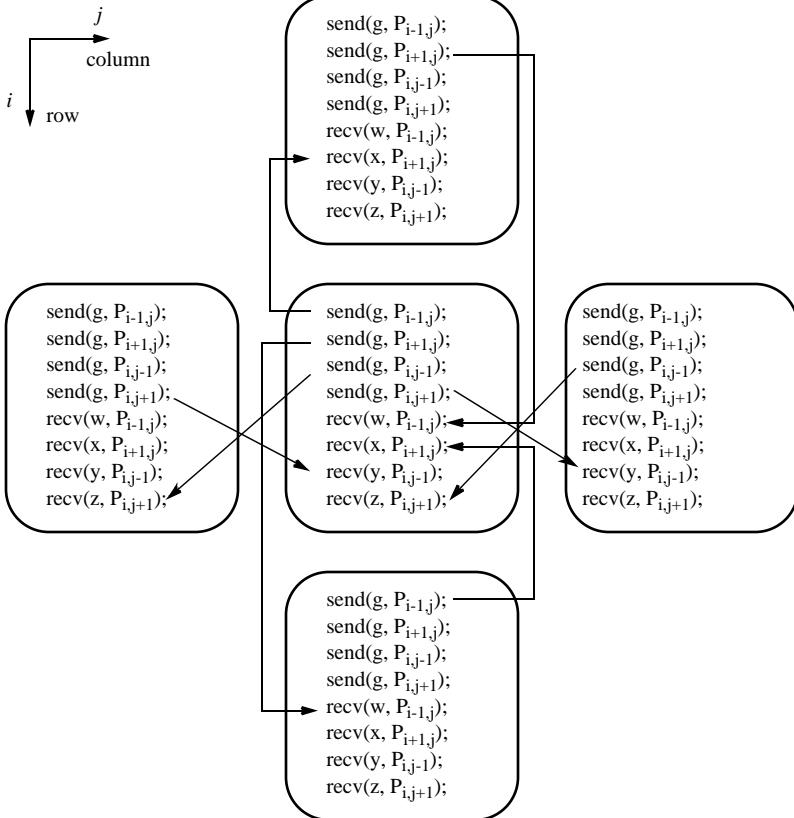
```
for (iteration = 0; iteration < limit; iteration++) {  
    g = 0.25 * (w + x + y + z);  
    send(&g, Pi-1,j); /* non-blocking sends */  
    send(&g, Pi+1,j);  
    send(&g, Pi,j-1);  
    send(&g, Pi,j+1);  
    recv(&w, Pi-1,j); /* synchronous receives */  
    recv(&x, Pi+1,j);  
    recv(&y, Pi,j-1);  
    recv(&z, Pi,j+1);  
}
```

Local barrier



Important to use **send( )**s that do not block while waiting for the **recv( )**s; otherwise the processes would deadlock, each waiting for a **recv( )** before moving on - **recv( )**s must be synchronous and wait for the **send( )**s.

## Message passing for heat distribution problem



Version where processes stop when they reach their required precision:

```
iteration = 0;
do {
    iteration++;
    g = 0.25 * (w + x + y + z);
    send(&g, Pi-1,j);           /* locally blocking sends */
    send(&g, Pi+1,j);
    send(&g, Pi,j-1);
    send(&g, Pi,j+1);
    recv(&w, Pi-1,j);           /* locally blocking receives */
    recv(&x, Pi+1,j);
    recv(&y, Pi,j-1);
    recv(&z, Pi,j+1);
} while((!converged(i, j)) || (iteration < limit));
send(&g, &i, &j, &iteration, Pmaster);
```

To handle the processes operating at the edges:

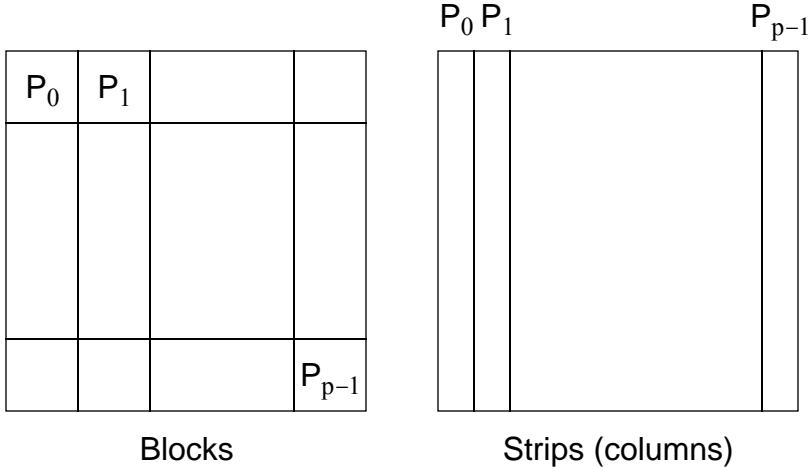
MPI has a construct to help here

```
if (last_row) w = bottom_value;
if (first_row) x = top_value;
if (first_column) y = left_value;
if (last_column) z = right_value;
iteration = 0;
do {
    iteration++;
    g = 0.25 * (w + x + y + z);
    if !(first_row) send(&g, P_{i-1,j});
    if !(last_row) send(&g, P_{i+1,j});
    if !(first_column) send(&g, P_{i,j-1});
    if !(last_column) send(&g, P_{i,j+1});
    if !(last_row) recv(&w, P_{i-1,j});
    if !(first_row) recv(&x, P_{i+1,j});
    if !(first_column) recv(&y, P_{i,j-1});
    if !(last_column) recv(&z, P_{i,j+1});
} while((!converged) || (iteration < limit));
send(&g, &i, &j, iteration, P_master);
```

# Partitioning

Normally allocate more than one point to each processor, because many more points than processors.

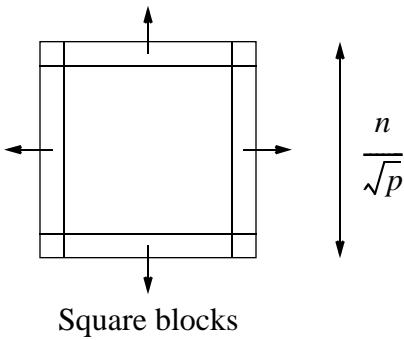
Points could be partitioned into square blocks or strips:



## Block partition

Four edges where data points exchanged. Communication time given by

$$t_{\text{commsq}} = 8 t_{\text{startup}} + \frac{n}{\sqrt{p}} t_{\text{data}}$$

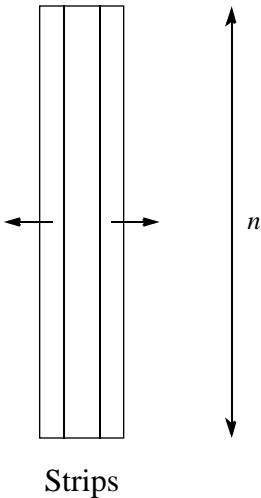


## Communication consequences of partitioning

## Strip partition

Two edges where data points are exchanged. Communication time is given by

$$t_{\text{commcol}} = 4(t_{\text{startup}} + nt_{\text{data}})$$



Strips

## Optimum

In general, the strip partition is best for a large startup time, and a block partition is best for a small startup time.

With the previous equations, the block partition has a larger communication time than the strip partition if

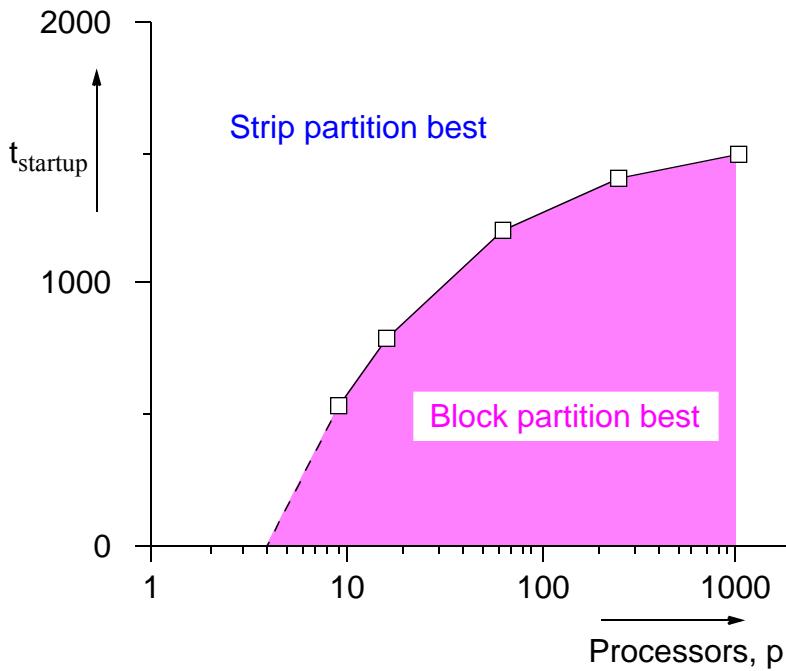
$$8t_{\text{startup}} + \frac{n}{\sqrt{p}} t_{\text{data}} > 4(t_{\text{startup}} + nt_{\text{data}})$$

or

$$t_{\text{startup}} > n - \frac{2}{\sqrt{p}} t_{\text{data}}$$

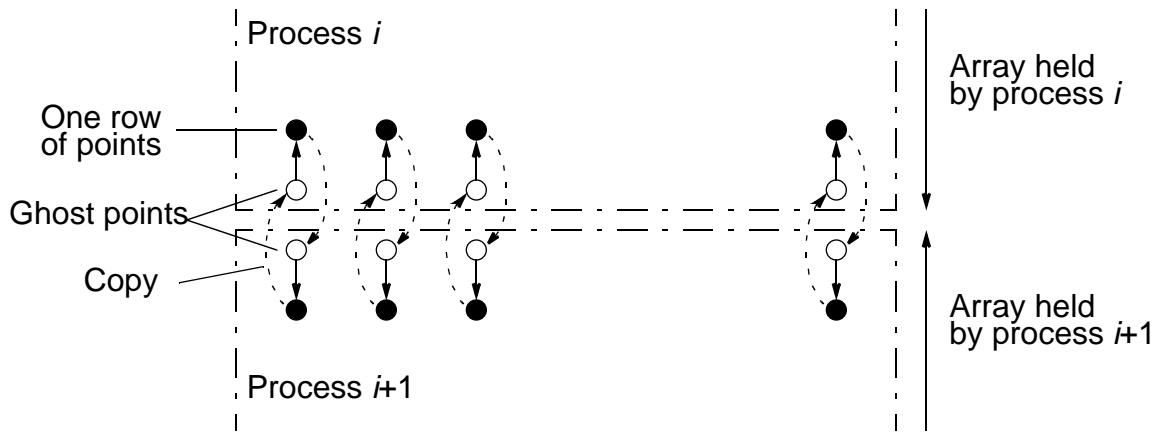
(p 9).

# Startup times for block and strip partitions



# Ghost Points

An additional row of points at each edge that hold the values from the adjacent edge. Each array of points is increased to accommodate the ghost rows.



## Safety and Deadlock

When all processes send their messages **first** and then receive all of their messages is “**unsafe**” because it relies upon buffering in the **send( )**s. The amount of buffering is not specified in MPI.

If insufficient storage available, send routine may be delayed from returning until storage becomes available or until message can be sent without buffering.

Then, a locally blocking **send()** could behave as a synchronous **send()**, only returning when the matching **recv()** is executed.

Since a matching **recv()** would never be executed if all the **send()**s are synchronous, **deadlock would occur**.

## Making the code safe

Alternate the order of the `send()`s and `recv()`s in adjacent processes so that only one process performs the `send()`s first:

```
}
```

Then even synchronous `send()`s would not cause deadlock.

Good way you can test for safety is to replace message-passing routines in a program with synchronous versions.

# MPI Safe Message Passing Routines

MPI offers several alternative methods for safe communication:

- Combined send and receive routines:

`MPI_Sendrecv()`

which is guaranteed not to deadlock

- Buffered send(s):

`MPI_Bsend()`

here the user provides explicit storage space

- Nonblocking routines:

`MPI_Isend()` and `MPI_Irecv()`

which return immediately. Separate routine used to establish whether message has been received - `MPI_Wait()`,  
`MPI_Waitall()`,`MPI_Waitany()`,`MPI_Test()`,`MPI_Testall()`,  
or `MPI_Testany()`

# Other fully synchronous problems

## Cellular Automata

The problem space is divided into cells.

Each cell can be in one of a finite number of states.

Cells affected by their neighbors according to certain rules, and all cells are affected simultaneously in a “generation.”

Rules re-applied in subsequent generations so that cells evolve, or change state, from generation to generation.

Most famous cellular automata is the “Game of Life” devised by John Horton Conway, a Cambridge mathematician.

Also good assignment for graphical output.

# The Game of Life

Board game - theoretically infinite two-dimensional array of cells. Each cell can hold one “organism” and has eight neighboring cells, including those diagonally adjacent. Initially, some cells occupied.

The following rules apply:

1. Every organism with two or three neighboring organisms survives for the next generation.
2. Every organism with four or more neighbors dies from overpopulation.
3. Every organism with one neighbor or none dies from isolation.
4. Each empty cell adjacent to exactly three occupied neighbors will give birth to an organism.

These rules were derived by Conway “after a long period of experimentation.”

# Simple Fun Examples of Cellular Automata

## “Sharks and Fishes”

An ocean could be modeled as a three-dimensional array of cells.

Each cell can hold one fish or one shark (but not both).

Fish and sharks follow “rules.”

# Fish

Might move around according to these rules:

1. If there is one empty adjacent cell, the fish moves to this cell.
2. If there is more than one empty adjacent cell, the fish moves to one cell chosen at random.
3. If there are no empty adjacent cells, the fish stays where it is.
4. If the fish moves and has reached its breeding age, it gives birth to a baby fish, which is left in the vacating cell.
5. Fish die after  $x$  generations.

# Sharks

Might be governed by the following rules:

1. If one adjacent cell is occupied by a fish, the shark moves to this cell and eats the fish.
2. If more than one adjacent cell is occupied by a fish, the shark chooses one fish at random, moves to the cell occupied by the fish, and eats the fish.
3. If no fish in adjacent cells, the shark chooses an unoccupied adjacent cell to move to in a similar manner as fish move.
4. If the shark moves and has reached its breeding age, it gives birth to a baby shark, which is left in the vacating cell.
5. If a shark has not eaten for  $y$  generations, it dies.

Similar examples: “foxes and rabbits” -Behavior of rabbits to move around happily whereas behavior of foxes is to eat any rabbits they come across.

# Sample Student Output



# Serious Applications for Cellular Automata

## Examples

- fluid/gas dynamics
- the movement of fluids and gases around objects
- diffusion of gases
- biological growth
- airflow across an airplane wing
- erosion/movement of sand at a beach or riverbank.

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/232907330>

# Parallelizing apriori on dual core using openMP

Article · April 2012

---

CITATIONS

4

READS

61

3 authors, including:



T. AnuRadha  
Velagapudi Ramakrishna Siddhartha Engineering College

20 PUBLICATIONS 23 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



frequent itemset mining on multicore processors [View project](#)



malware detection using deep learning [View project](#)

# Parallelizing Apriori on Dual Core using OpenMP

Anuradha.T  
ANU,Guntur  
A.P.,INDIA

Satya Pasad R  
ANU,Guntur  
A.P.,INDIA

S.N.Tirumalarao  
NEC,Guntur  
A.P.,INDIA

## ABSTRACT

Accumulation of abundant data from different sources of the society but a little knowledge situation has lead to Knowledge Discovery from Databases or Data Mining. Data Mining techniques use the existing data and retrieve the useful knowledge from it which is not directly visible in the original data. As Data Mining algorithms deal with huge data, the primary concerns are how to store the data in the main memory at run time and how to improve the run time performance. Sequential algorithms cannot provide scalability, in terms of the data dimension, size, or runtime performance, for such large databases. Because the data sizes are increasing to a larger quantity, we must use high-performance parallel and distributed computing to get the advantage of more than one processor to handle these large quantities of data. The recent advancements in computer hardware for parallel processing is multi core or Chip Multiprocessor (CMP) systems. In this paper we present an efficient and easy technique for parallelization of apriori on dual-core using openMP wih perfect load balancing between the two cores. We present the performance evaluation of apriori for different support counts with different sized databases on dual core compared to sequential implementation

## General Terms

Data Mining, Parallel Processing

## Keywords

Data Mining, Parallel Processing

## 1. INTRODUCTION

Data Mining deals with large volumes of data to extract the previously unseen and useful knowledge (1,2).Association Rule mining (ARM) or frequent itemset mining is an important functionality of Data Mining(3). The apriori algorithm (4) is one of the best algorithms for finding frequent itemsets from a transactional database. It requires scanning the entire database more number of times. As Data Mining mainly deals with large volumes of data, the main concern should be how to improve the performance of the algorithm. One way of improving the performance of apriori is parallelizing the algorithm (5, 6). The recent advancement in computer hardware for parallel processing is Multi-Core architectures(7,8).In this paper we present the performance evaluation of parallelization of apriori for different sized databases with different support counts on a dual core system compared to its sequential implementation using an efficient and easy technique with perfect load balancing between the processors.

## 2. Related Work

Apriori algorithm is the first algorithm proposed for frequent itemset mining which depends on candidate generation (4, 9). Han et al. proposed FP-growth method for frequent itemset mining without candidate generation (10). Apriori and FP-growth methods use horizontal data format to represent the transactional database. Zaki proposed Eclat algorithm for

frequent itemset mining using vertical data format(11). To handle the

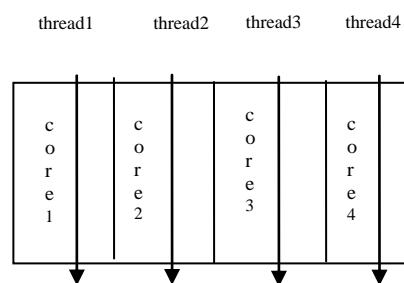
scalability problem of sequential algorithms, parallel and distributed algorithms are proposed(12).Rakesh Agrawal, John C.Shafer proposed the first two parallel association algorithms count distribution and data distribution (13). David W. Cheung et al. proposed a distributed association rule mining algorithm FDM, which reduces the number of messages to be passed at mining association rules (14). Mohammed J. Zaki presented a survey on parallel and distributed association rule mining (6).Zaiane et al. proposed a parallel algorithm for mining frequent patterns using FP-Growth mining (15).Wenbin Fang et al. presented two implementations of frequent itemset mining on new generation Graphics processing units (16).Shrirish Tatikonda et al. discussed mining subtrees from a tree structured data by considering various memory optimizations in (17). Li Liu et al. proposed a cache-conscious FP-array (frequent pattern array) and a lock-free dataset tiling parallelization mechanism to parallelize frequent itemset mining on multi core using FP-tree based mining(18).

## 3. LITERATURE

### 3.1 Multi core

Multi core refers two or more processors. But they differ from independent parallel processors as they are integrated on the same chip circuit (7,8).A multi core processor implement message passing or shared memory inter core communication methods for multiprocessing. If the number of threads are less than or equal to the number of cores, separate core is allocated to each thread and threads run independently on multiple cores. (Figure 1) If the number of threads are more than the number of cores, the cores are shared among the threads.

Any application that can be threaded can be mapped efficiently to multi-core, but the improvement in performance gained by the use of multi core processors depends on the fraction of the program that can be parallelized.[ Amdahl's law] (19)



**Figure 1: Independent threads on the cores**

### 3.2 OpenMP

OpenMP (Open Multi Processing) is an application program interface (API) that supports multi-platform shared memory multi processing programming in C/C++ and Fortran on many architectures(20,21,22).It consists of a set of compiler directives. OpenMP uses Fork-Join Parallelism to implement multi threading.

### 3.3 Fork-Join Parallelism

Initially programs begin as a single process: master thread. We can make some part of the program to work in parallel by creating child threads. Master thread executes in serial mode until the parallel region construct is encountered. Master thread creates a team of parallel child threads (fork) that simultaneously execute statements in the parallel region. The work sharing construct divides the work among all the threads. After executing the statements in the parallel region, team threads synchronize and enumerate (join) but master continues (Figure 2).

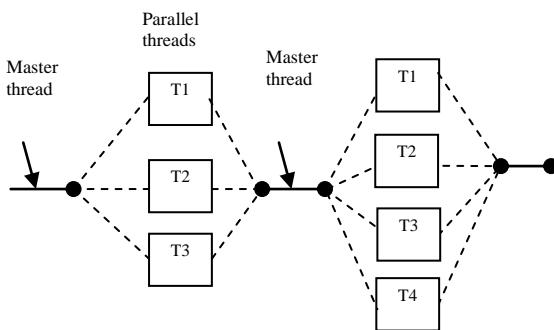


Figure 2: Fork-Join parallelism

## 4 .ASSOCIATION RULE MINING

The concept of Association rule mining was originated from the market basket analysis (3). It considers a transactional database D which consists of the transactional records of the customers {T1,T2,...,Tn}. Each transaction T consists of the items purchased by the customers in one visit of the super market. The items are the subset of the set of whole items I in the super market we are considering for analysis. We represent I as the set {I1, I2,..., Im}.An itemset consists of some combination of items which occur together or a single item from I. Association rule mining  $X \Rightarrow Y$ , represents the dependency relationship between two different itemsets X and Y in the database. The relationship is whenever X is occurring in any transaction, there is a probability that Y may also occur in the same transaction. This occurrence is based on two interesting measures.

1.Support= percentage of transactions in D that contain XUY  

$$support(X) = P(X \cup Y)$$

2.Confidence=percentage of transactions in D containing X that also contain Y.

$$confidence(X \Rightarrow Y) = P(Y/X)$$

Finding the association rules for any given transactional database consists of two parts:

1. Finding the frequent itemsets—Frequent itemsets are the itemsets which are having a frequency more than a predefined minimum support count (min\_sup).
2. Generation of Association rules— Rules generated from the subsets of a frequent itemset.

The major part of this rule mining is finding frequent itemsets. Apriori algorithm is the popular algorithm for finding the frequent itemsets.

### 4.1 The Apriori algorithm

It is based on the apriori property that all nonempty subsets of a frequent itemset must also be frequent (4). It is a two step process.

Step 1: The Prune Step: The entire database is scanned to find the count of each candidate in Ck. Ck represents candidate k-itemset. The count each itemset in Ck is compared with a predefined minimum support count to find whether that itemset can be placed in frequent k-itemset Lk.

Step 2: The join step: Lk is natural joined with itself to get the next candidate k+1- itemset Ck+1.

The major step here is the prune step which requires scanning the entire database for finding the count of each itemset in every candidate k-itemset. So to find all the frequent itemsets in the database, it requires more time if the database size is more.

### 4.2 Parallelizing apriori on dual- core using openmp

In parallelizing apriori on dual-core, we use the fork-join concept of OpenMP for finding frequent k- itemset.

Pseudocode for apriori on dualcore

//All the items in the database will be in candidate 1-itemset , C1.For each item in C1, the following procedure will be followed.

1. divide the database into 2 partitions.
2. select a minimum support count ,min\_sup.
3. SET\_ OMP\_NUM\_THREADS =2
4. /\* start of parallel code
- #pragma omp parallel
- #pragma omp sections
- {
- omp section
- {
- //partition1
- find\_count1(i)
- }
- omp section
- {
- //partition2
- find\_count2(i)
- }
- /\* implicit barrier \*/
- //The first section calculate the count of items in partition1 and second section calculate the count of items in partition 2 .
5. sum up the counts of each partition separately
- for each item[i] in C1 and if count[i]>min\_sup ,place item[i] in frequent 1-itemset,L1.
6. Join L1 with itself to get C2 where the items are of the type (i,j).
- 7.Go to step 4 for finding count of items in C2.
- 8.Repeat this process until L k=  $\emptyset$ .

## 5. EXPERIMENTAL WORK

For our experimentation, we have used Intel Pentium Dual-Core 1.60 GHz processors with 3GB RAM. We have used Fedora 9 Linux (Kernel 2.6.25-14, Red Hat nash version 6.0.52) equipped with GNU C++ (gcc version 4.3) in our study. We have used OpenMP threads to study their performance on dual-Core processors. In our experiments randomly generated data sets are used. Random data sets are generated separately to have 1 to 10 lakh records with 10 different items,I1 to I10.Our algorithm is tested with different support counts 5%,10%,15%, 20%,30%,

35% 40%,45% and 50% for each dataset. First the program is implemented sequentially and then it is implemented on dual core processors using openMP threads. Perfect load balancing will be done as each thread will take only 50% of the data. To know the time consumed by the program in different environments, we have used time command of Linux .This command gives the real time, user time and system time. These timings are noted by varying datasets and for each dataset by varying different support counts for single core and dual core implementations.

### 5.1 Comparison with related Work

We have parallelized apriori in our work using the concept similar to count distribution (CD) algorithm discussed in (13). In count distribution also database is partitioned and each processor is responsible for only locally stored transactions. But in count distribution algorithm a hash-tree corresponds to all the candidates is built and it is partitioned among processors but our implementation does not build any hash tree. Because of the overhead in constructing the hash tree, CD algorithm does not perform well with respect to increasing the number of candidates that is by lowering minimum support count(23) But our implementation perform well at lower minimum support counts. In Data distribution (DD) algorithm the candidate itemsets are partitioned among the processors. And both CD and DD algorithms are implemented on individual parallel processors. But our implementation is done on dual core processor. Shirish Tatikonda et al. (17) discussed parallelizing on multi core but they have worked with mining subtrees from a tree structured data and our implementation deals with frequent itemset mining. Li Liu et al.(18) proposed frequent itemset mining on multi core using FP-tree based mining and our implementation uses apriori algorithm.

### 5.2 Experimental Results

Our experiments gave the following observations:

1. There is a run time performance improvement by parallelizing the apriori algorithm on dual core compared to sequential implementation.
2. For any dataset with different support counts, real time consumed by the algorithm on dual core is less than that of the time consumed for sequential execution (Figure 3 to Figure 12). The benefit of dual core in real times can be observed more at lower support counts than at higher support counts that means when the frequent itemsets generated are more. (Figure 13 to Figure 22).
3. But we can observe only a slight reduction in the user time on dual core compared to sequential execution at lower support counts and user time on dual core is slightly more than sequential execution at higher support counts.(Figure 25 to Figure 28)
4. For any dataset , when the support count is increasing, the real time and user time consumed will be decreasing on both sequential and parallel implementations if the frequent itemsets generated are different for those support counts. (Figure 3 to Figure 12,Figure 23,Figure 24).
5. For any support count , when the dataset size is increasing, the real and user time consumed will be increasing on both sequential and parallel implementations.(Figure 13 to Figure 22,Figure 25 to Figure 28)

### 5.3 Real time Observations for different support counts with fixed data sizes

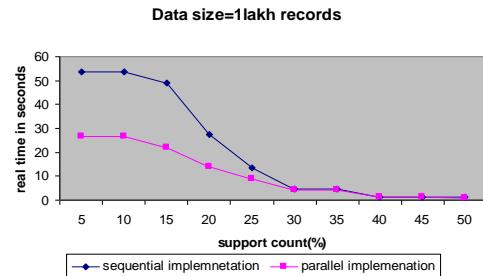


Figure 3 : One lakh data with varying support counts

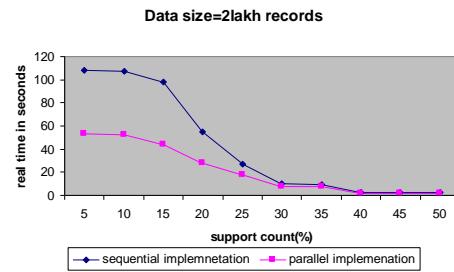


Figure 4: Two lakh data with varying support counts

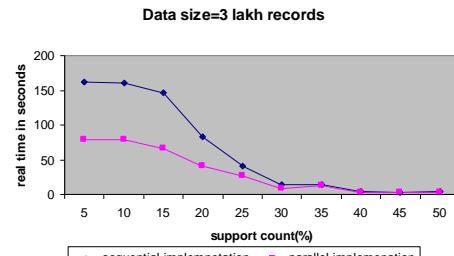


Figure 5: Three lakh data with varying support counts

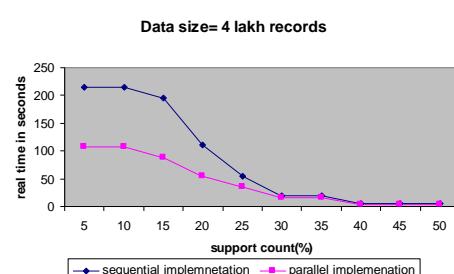


Figure 6: Four lakh data with varying support counts

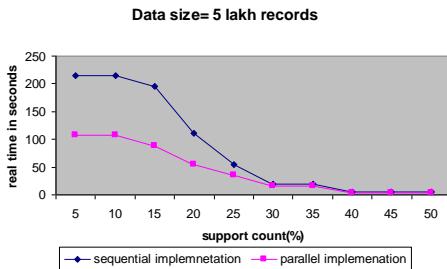


Figure 7: Five lakh data with varying support counts

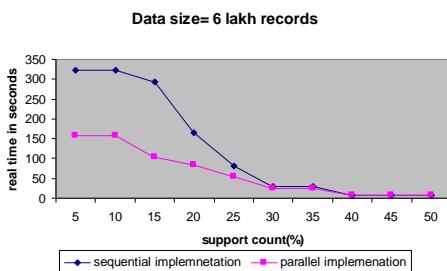


Figure 8: Six lakh data with varying support counts

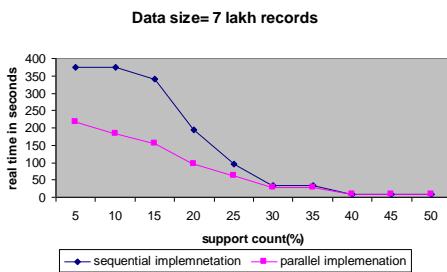


Figure 9: Seven lakh data with varying support counts

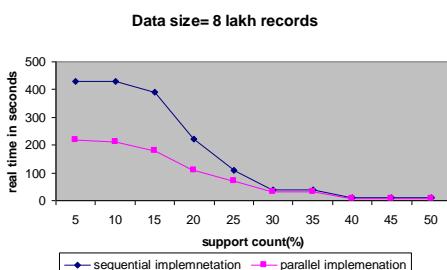


Figure 10: Eight lakh data with varying support counts

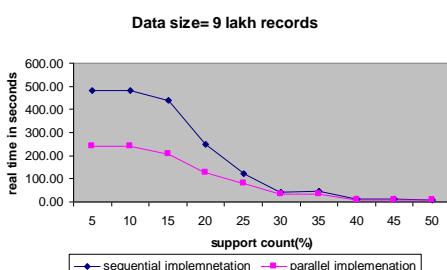


Figure 11: Nine lakh data with varying support counts

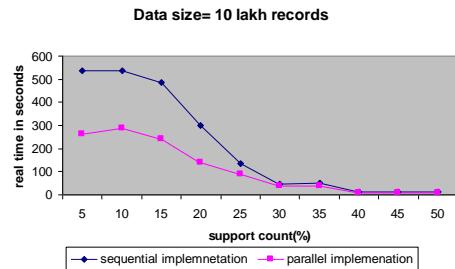


Figure 12: Ten lakh data with varying support counts

#### 5.4 Real time observations for different data sizes with fixed support counts

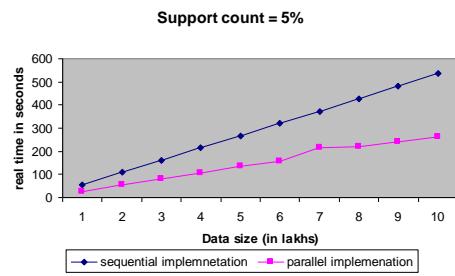


Figure 13: Sup\_count 5% with varying datasizes

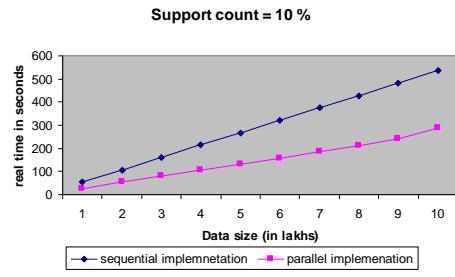


Figure 14: Sup\_count 10% with varying datasizes

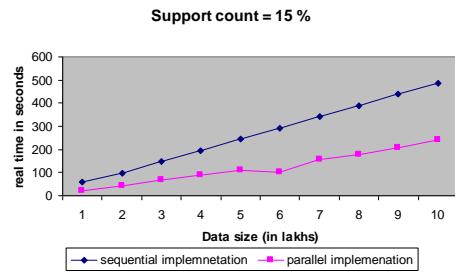


Figure 15: Sup\_count 15% with varying datasizes

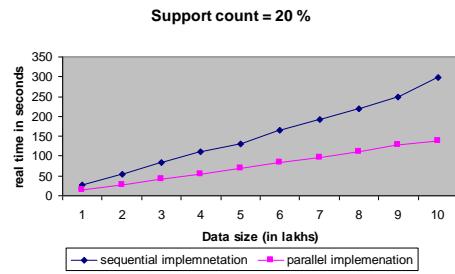


Figure 16: Sup\_count 20% with varying datasizes

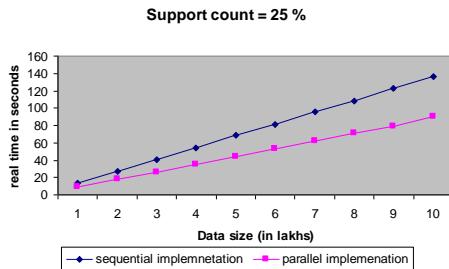


Figure 17: Sup\_count 25% with varying datasizes

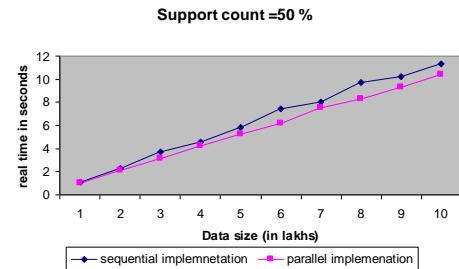


Figure 22: Sup\_count 50% with varying datasizes

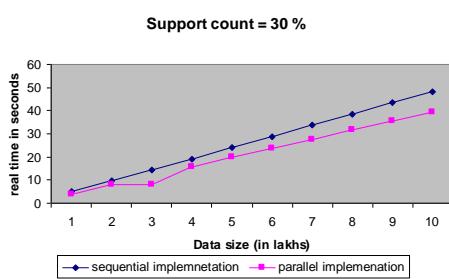


Figure 18: Sup\_count 30% with varying datasizes

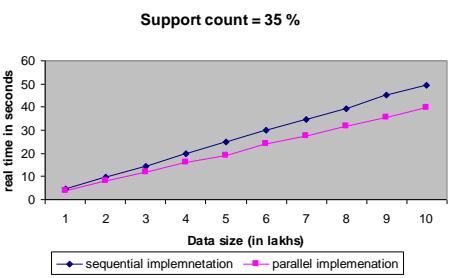


Figure 19: Sup\_count 35% with varying datasizes

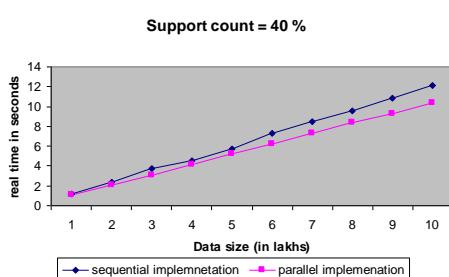


Figure 20: Sup\_count 40% with varying datasizes

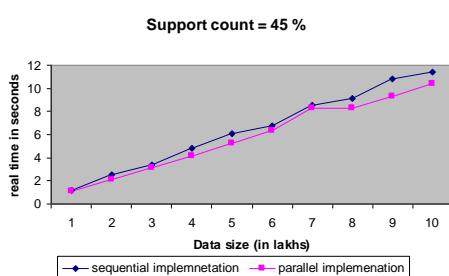


Figure 21: Sup\_count 45% with varying datasizes

## 5.6 User time Observations for different support counts with fixed data sizes

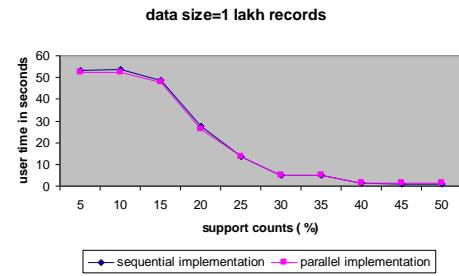


Figure 23: One lakh data with varying support counts

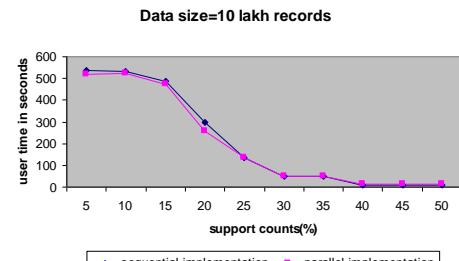


Figure 24: Ten lakh data with varying support counts

## 5.7 User time observations for different datasizes with fixed support counts

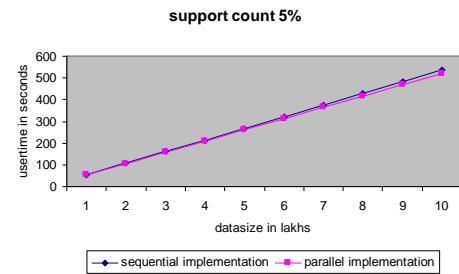


Figure 25: sup\_count 5% with varying datasizes

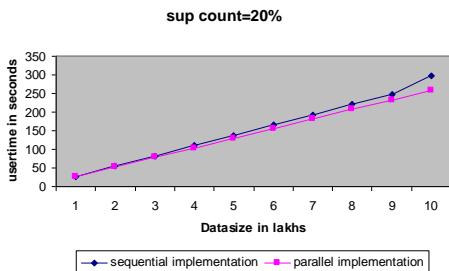


Figure 26: sup\_count 20% with varying datasizes

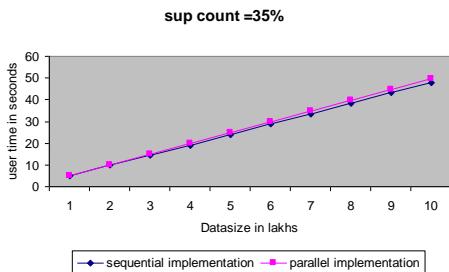


Figure 27: sup\_count 35% with varying datasizes

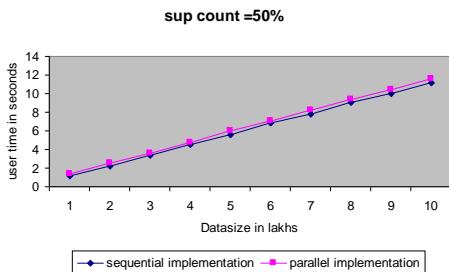


Figure 28: sup\_count 50% with varying datasizes

## 7. CONCLUSIONS

Apriori algorithm is parallelized on dual core using a simple and efficient technique with perfect load balancing between the cores. The run time performance of parallelization of apriori on dual core is compared to sequential execution with different support counts for different databases. There is a clear run time performance improvement of parallelizing the algorithm on dual core in terms of real time compared to sequential implementation on single core. In our future work we study the performance of apriori on dual core by changing the number of threads.

## 8. REFERENCES

- [1] Jiawei Han and Micheline Kamber 2006 "Data Mining concepts and Techniques", 2nd edition Morgan Kaufmann Publishers, San Francisco.
- [2] Fayyad, Usama, Gregory Piatetsky-Shapiro, and Padhraic Smyth 1996 "From Data Mining to Knowledge Discovery in Databases". AI Magazine Volume 17 Number 3(1996)
- [3] Agrawal R, Imielinski T, Swami A 1993 "Mining association rules between sets of items in large databases" In: Proceedings of the 1993 ACM-SIGMOD international conference on management of data (SIGMOD'93), Washington, DC, pp 207–216
- [4] Agrawal R, Srikant R 1994 "Fast algorithms for mining association rules" In: Proceedings of the 1994 international conference on very large data bases (VLDB'94), Santiago, Chile, pp 487–499
- [5] R. Agrawal and J. Shafer 1996 "Parallel mining of association rules" *IEEE Trans. Knowl. Data Eng.*, vol. 8, pp. 962–969, Dec. 1996.
- [6] M.J. Zaki 1997 "parallel and distributed association mining:A survey" *IEEEConcur*, vol. 7, pp. 14–25, Dec. 1997.
- [7] Herb Sutter 2005 "The Free Lunch Is Over A Fundamental Turn Toward Concurrency in Software" *This article appeared in Dr. Dobb's Journal*, 30(3), March 2005.
- [8] N.Karmakar 2011 "The New Trend in processor Making Multi-Core Architecture" www.scribd.com 15<sup>th</sup> may2011
- [9] Jiawei Han, Hong Cheng,Dong Xin, Xifeng Yan 2007 "Frequent pattern mining: current status and future directions" In the *Journal of Data Min Knowl Disc* (2007) 15:55–86, Springer Science+Business Media, LLC 2007.
- [10] Han J, Pei J, Yin Y 2000 "Mining frequent patterns without candidate generation" In: Proceeding of the 2000 ACM-SIGMOD international conference on management of data (SIGMOD'00),Dallas, TX, pp 1–12
- [11] ZakiMJ 2000 "Scalable algorithms for association mining" *IEEETransKnowl Data Eng* 12:372–390
- [12] Park JS, Chen MS, Yu PS 1995 "Efficient parallel mining for association rules" In: Proceeding of the 4th international conference on information and knowledge management, Baltimore, MD,pp 31–36
- [13] Agrawal R, Shafer JC 1996 "Parallel mining of association rules: design, implementation, and experience" *IEEE Trans Knowl Data Eng* 8:962–969
- [14] Cheung DW, Han J, Ng V, Fu A, Fu Y 1996 "A fast distributed algorithm for mining association rules" In: Proceeding of the 1996 international conference on parallel and distributed information systems, Miami Beach, FL, pp 31–44
- [15] O. R Zaiane,M. El-Hajj, and P. Lu 2001 "Fast parallel association rule mining without candidacy generation" in *Proc. ICDM*, 2001, [Online].Available: citeseer.ist.psu.edu/474621.html, pp. 665–668.
- [16] Wenbin Fang, Mian Lu, Xiangye Xiao, Bingsheng He, Qiong Luo 2009 "Frequent itemset mining on graphics processors" *Proceedings of the Fifth International Workshop on Data Management onNew Hardware (DaMoN 2009)* June 28, 2009, Providence, Rhode-Island
- [17] Shirish Tatikonda, Srinivasan Parthasarathy "Mining TreeStructured Data on Multicore Systems", *VLDB '08*, August 2430,2008, Auckland, New Zealand
- [18] Li Liu2, 1, Eric Li1, Yimin Zhang1, Zhizhong Tang 2007 "Optimization of Frequent Itemset Mining on Multiple-Core Processor" *VLDB '07*, September 23–28, 2007, Vienna, Austria.
- [19] Amdahl, Gene 1967 "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities". *AFIPS Conference Proceedings* (30): 483–485.
- [20] OpenMP Architecture , "OpenMP C and C++ ApplicationProgramInterface", Copyright © 1997–2002

- |   |        |   |
|---|--------|---|
| OpenMP<br>Architecture<br>Board. <a href="http://www.openmp.org/">http://www.openmp.org/</a>  | Review | [22] Ruud van der pas 2009 “An Overview of OpenMP” NTU<br>Talk January 14 2009  |
| [21] Kent Milfeld 2011 “Introduction to Programming with<br>OpenMP” September 12th 2011, TACC |        | [23] S N sivanandam, S Sumathi 2006 “DataMining<br>concepts,tasks and Techniques” First print 2006 by<br>Thomson Business Information Pvt. Ltd., India. |

# MapReduce

**Thoai Nam**

Faculty of Computer Science and Engineering  
HCMC University of Technology

# Ref

- MapReduce algorithm design, Jimmy Lin



processes 20 PB a day (2008)  
crawls 20B web pages a day (2012)



>10 PB data, 75B DB  
calls per day (6/2012)

>100 PB of user data +  
500 TB/day (8/2012)



S3: 449B objects, peak 290k  
request/second (7/2011)  
IT objects (6/2012)



640K ought to be  
enough for anybody.



150 PB on 50k+ servers  
running 15k apps (6/2011)



Wayback Machine: 240B web  
pages archived, 5 PB (1/2013)

LHC: ~15 PB a year



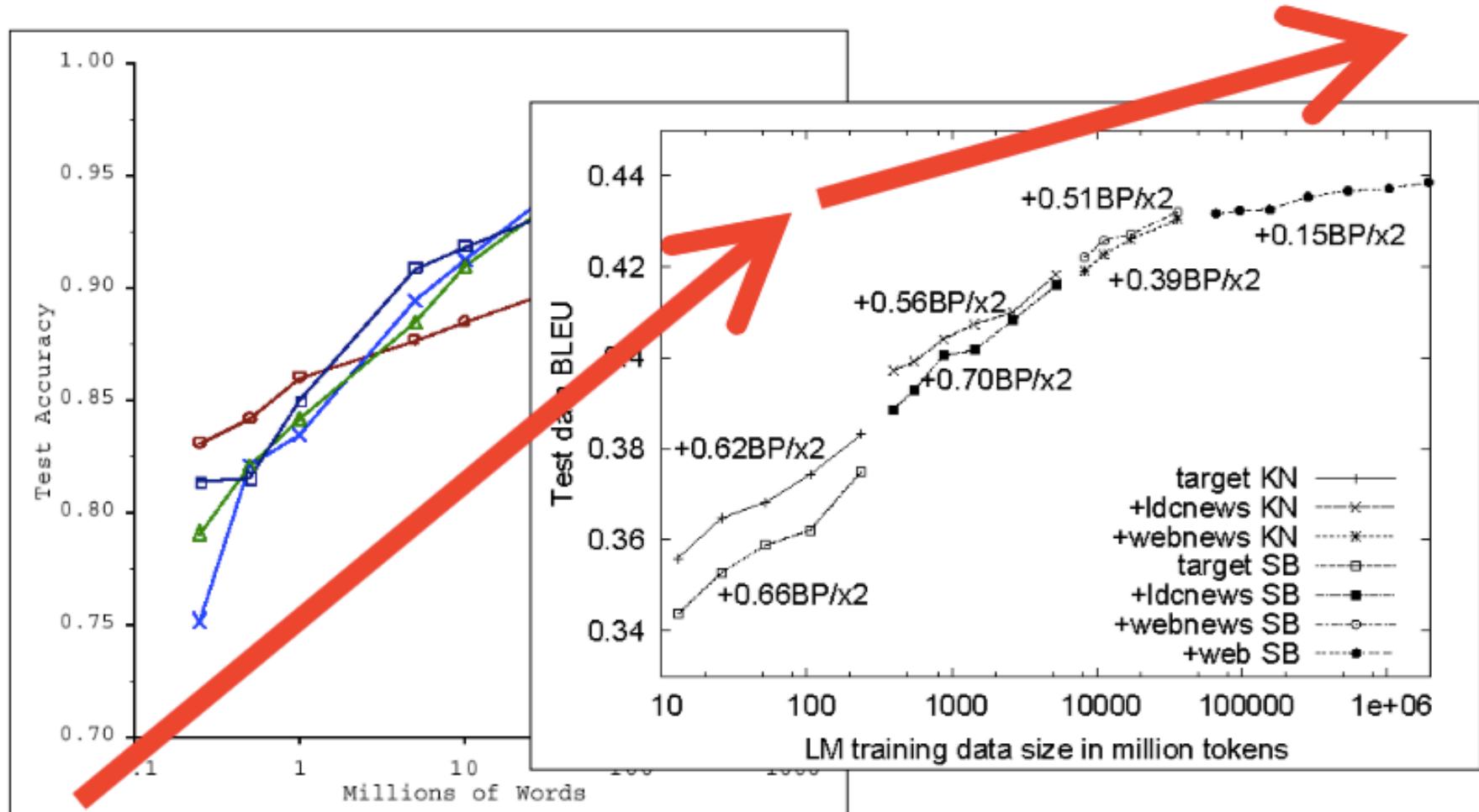
LSST: 6-10 PB a year  
(~2015)



**How much data?**

# No data like more data!

s/knowledge/data/g;



A wide-angle photograph of a massive server room. The space is filled with floor-to-ceiling server racks, their front panels glowing with various colors like blue, green, and yellow. The room has a high ceiling supported by a complex steel truss structure. Light from the overhead fixtures creates a dramatic play of shadows and highlights on the metallic surfaces of the racks and the ceiling.

# MapReduce

CSE-HCMUT

5

# Typical Big Data Problem

- Iterate over a large number of records

**Map** Extract something of interest from each

- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

**Reduce**

**Key idea: provide a functional abstraction for these two operations**

# MapReduce: A Real World Analogy

## Coins Deposit



# MapReduce: A Real World Analogy

## Coins Deposit



## Coins Counting Machine

# MapReduce: A Real World Analogy

## Coins Deposit

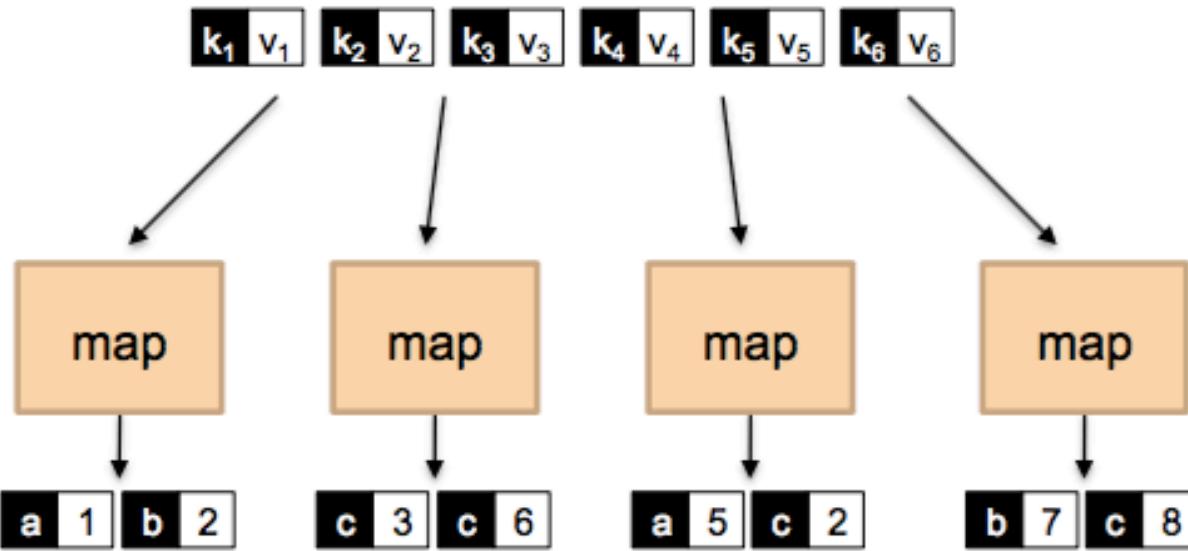


**Mapper:** Categorize coins by their face values

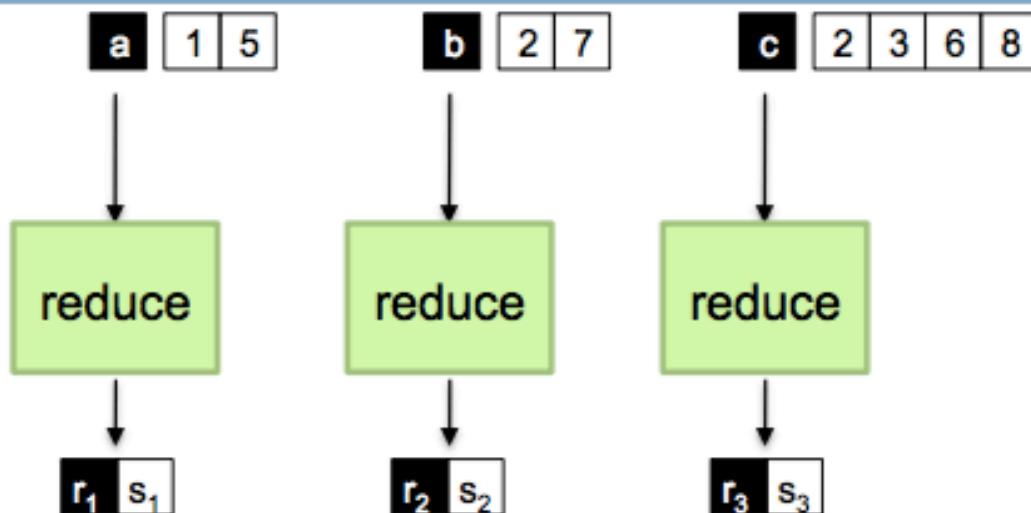
**Reducer:** Count the coins in each face value *in parallel*

# MapReduce

- Programmers specify two functions:  
**map** ( $k_1, v_1$ )  $\rightarrow$  [ $k_2, v_2$ ]  
**reduce** ( $k_2, [v_2]$ )  $\rightarrow$  [ $k_3, v_3$ ]  
(All values with the **same key** are sent to the same reducer)
- The execution framework handles everything else...



**Shuffle and Sort: aggregate values by keys**



# MapReduce

- Programmers specify two functions:

**Map** (k<sub>1</sub>, v<sub>1</sub>) → <k<sub>2</sub>, v<sub>2</sub>>\*

**Reduce** (k<sub>2</sub>, list (v<sub>2</sub>)) → list (v<sub>3</sub>)

(All values with the **same key** are sent to the same reducer)

- The execution framework handles everything else...

# MapReduce “runtime”

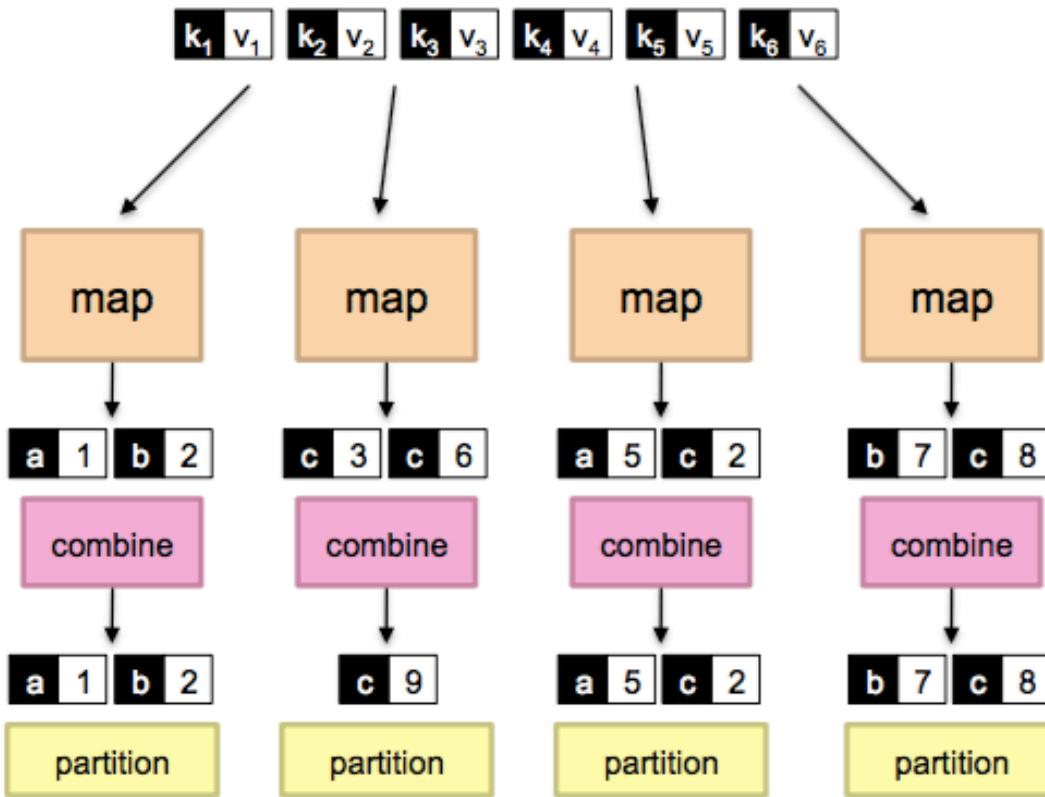
- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed file system

# Synchronization & ordering

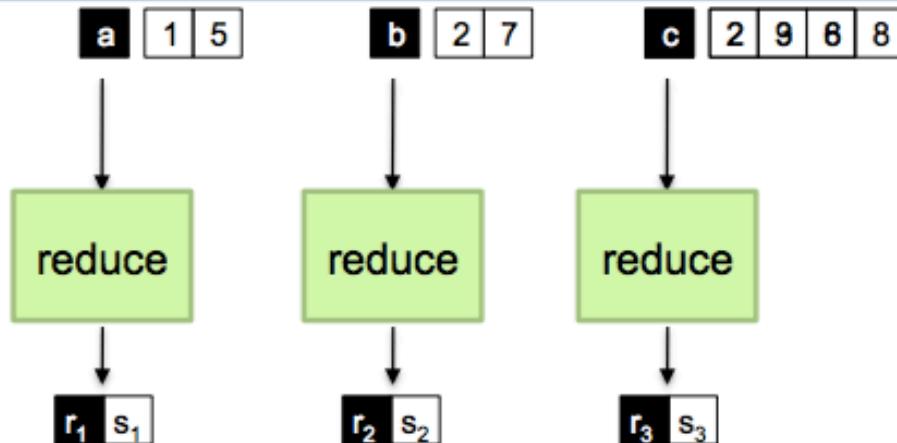
- Barrier between map and reduce phases
  - But intermediate data can be copied over as soon as mappers finish
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

# MapReduce

- Programmers specify two functions:  
**Map** ( $k_1, v_1 \rightarrow \langle k_2, v_2 \rangle^*$ )  
**Reduce** ( $k_2, \text{list } (v_2) \rightarrow \text{list } (v_3)$ )  
(All values with the **same key** are sent to the same reducer)
- The execution framework handles everything else...
- Not quite...usually, programmers also specify:  
**partition** ( $k_2, \text{number of partitions} \rightarrow \text{partition for } k_2$ 
  - Often a simple hash of the key, e.g.,  $\text{hash}(k') \bmod n$
  - Divides up key space for parallel reduce operations  
**combine** ( $k_2, v_2 \rightarrow \langle k_2, v_2 \rangle^*$ 
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic

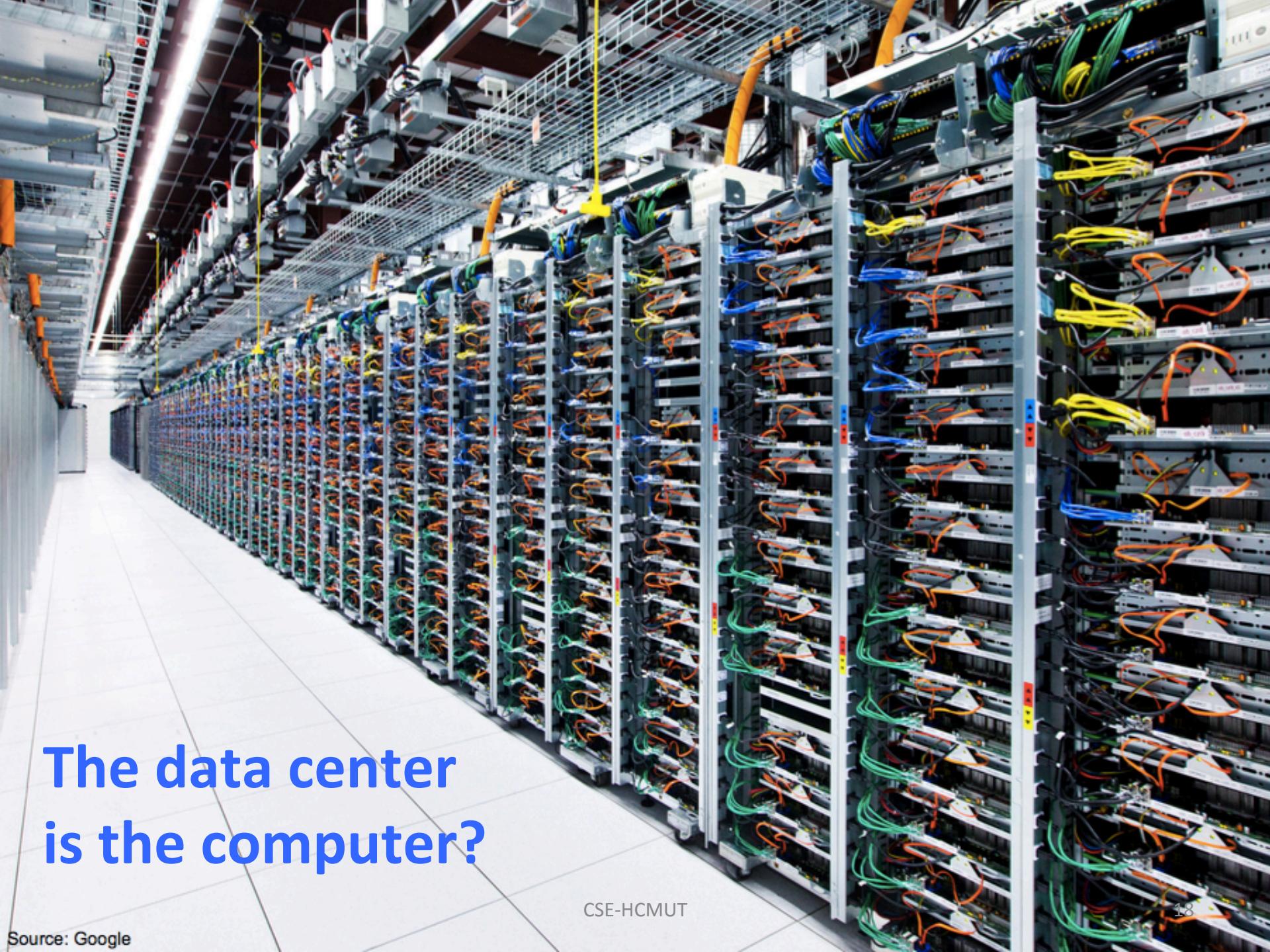


### Shuffle and Sort: aggregate values by keys



# What's the big deal?

- Developers need the right level of abstraction
  - Moving beyond the von Neumann architecture
  - We need better programming models
- Abstractions hide low-level details from the developers
  - No more race conditions, lock contention, etc.
- MapReduce separating the *what* from *how*
  - Developer specifies the computation that need to be performed
  - Execution framework (“runtime”) handles actual execution



The data center  
is the computer?

# MapReduce can refers to...

- The programming model
- The execution framework (aka “runtime”)
- The specific implementation

# MapReduce Implementations

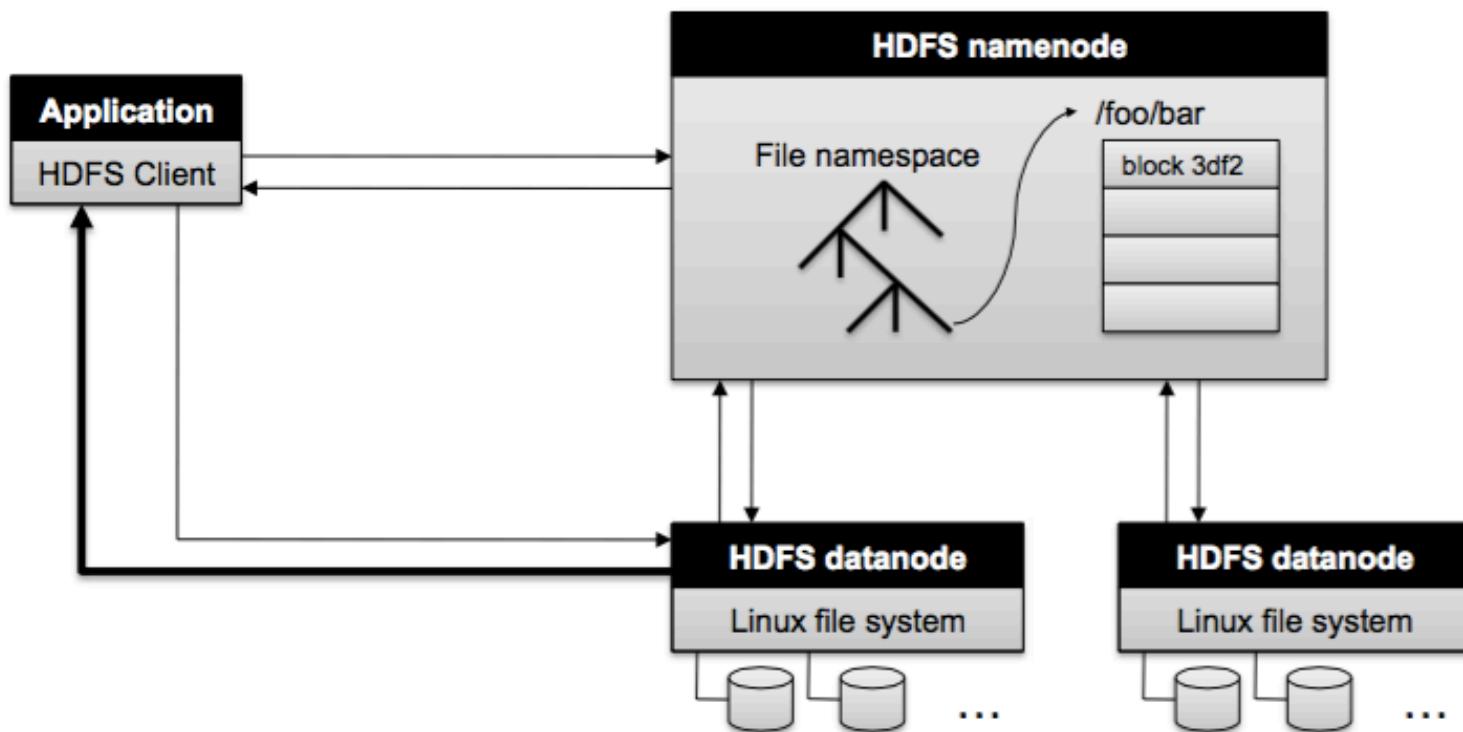
- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, now an Apache project
  - Used in production at Yahoo, Facebook, Twitter, LinkedIn, Netflix, etc.
  - The *de facto* big data processing platform
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.



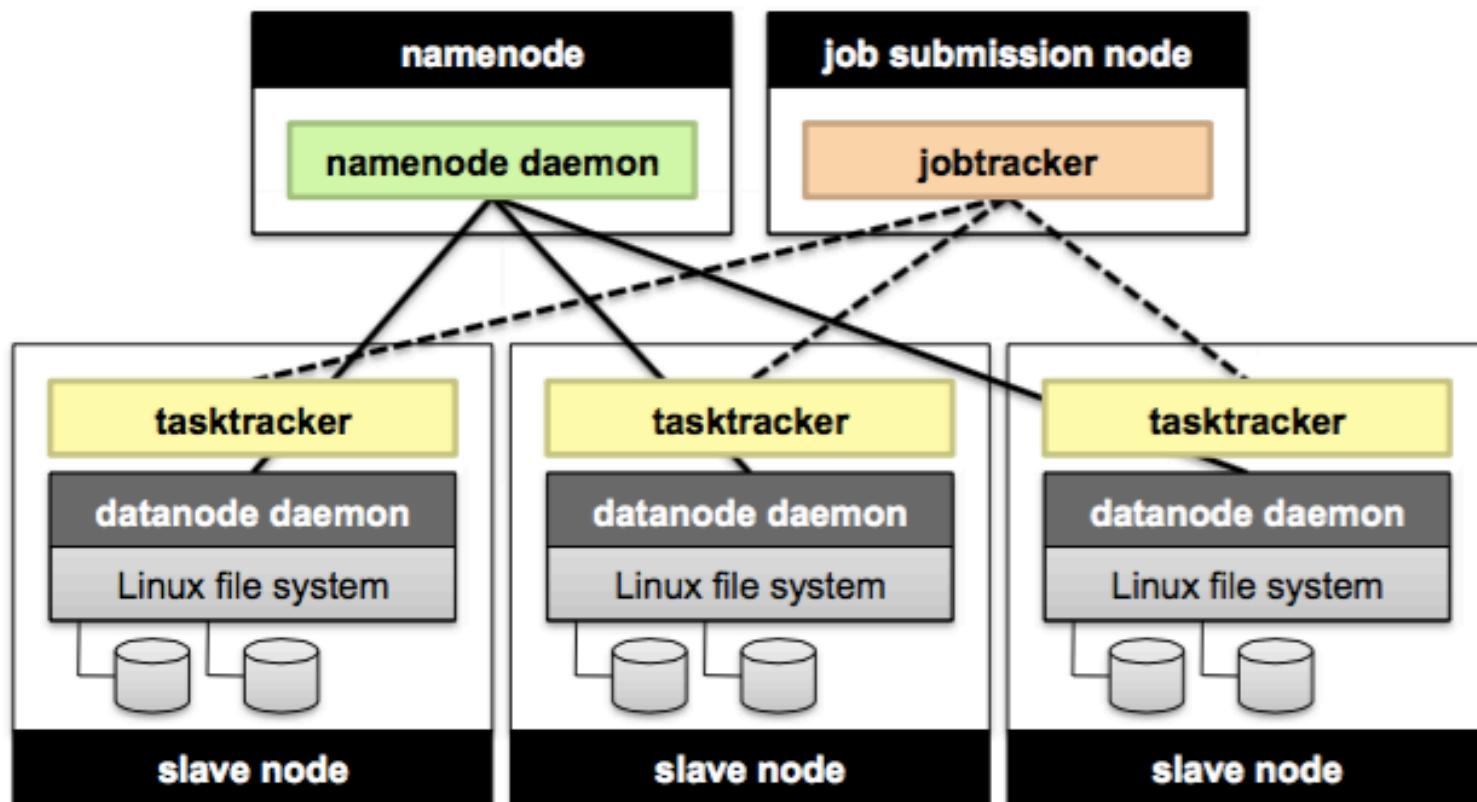
# MapReduce algorithm design

- The execution framework handles “everything else”...
  - Scheduling: assigns workers to map and reduce tasks
  - “Data distribution”: moves processes to data
  - Synchronization: gathers, sorts, and shuffles intermediate data
  - Errors and faults: detects worker failures and restarts
- Limited control over data and execution flow
  - All algorithms must expressed in m,r,c,p
- You don’t know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

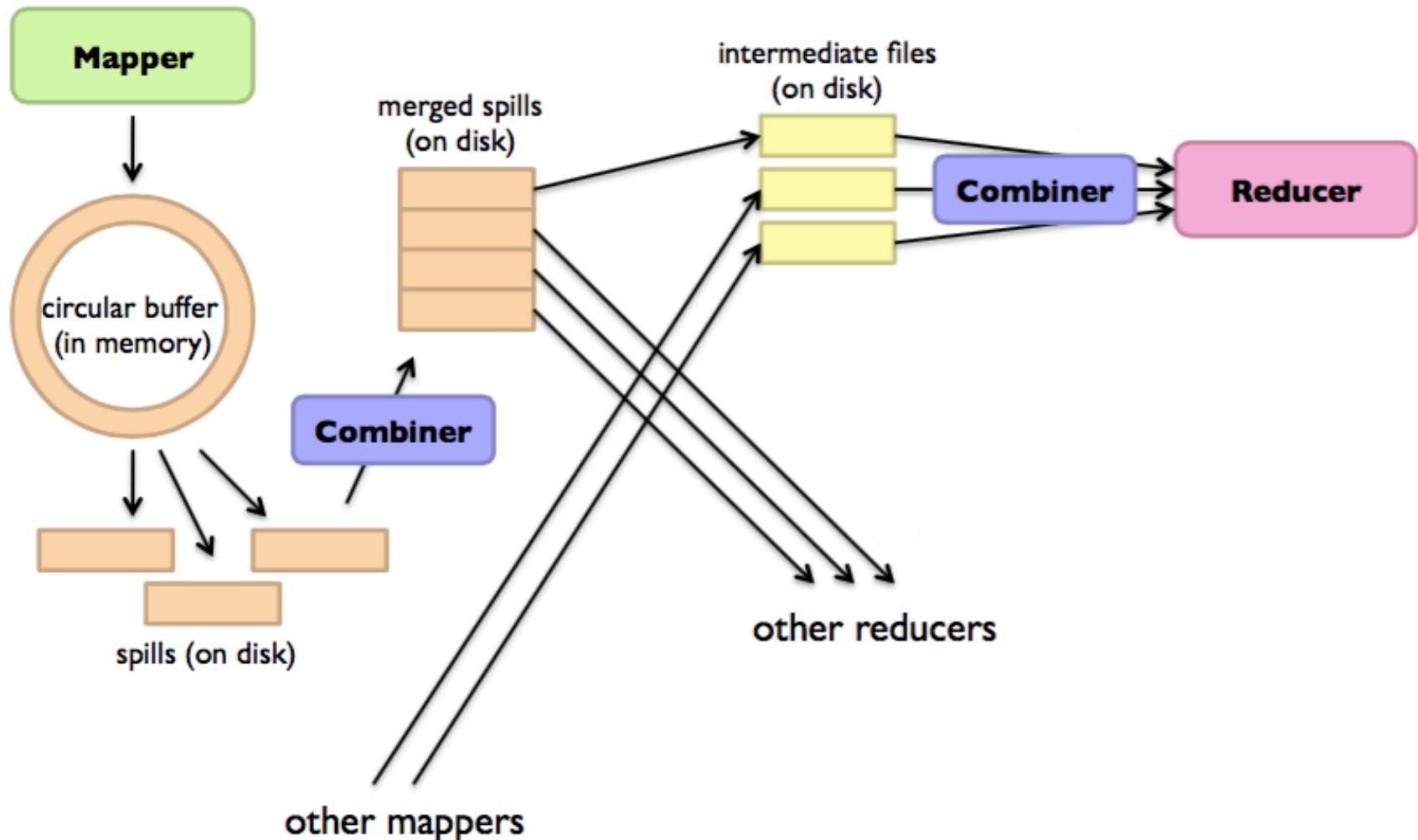
# HDFS architecture



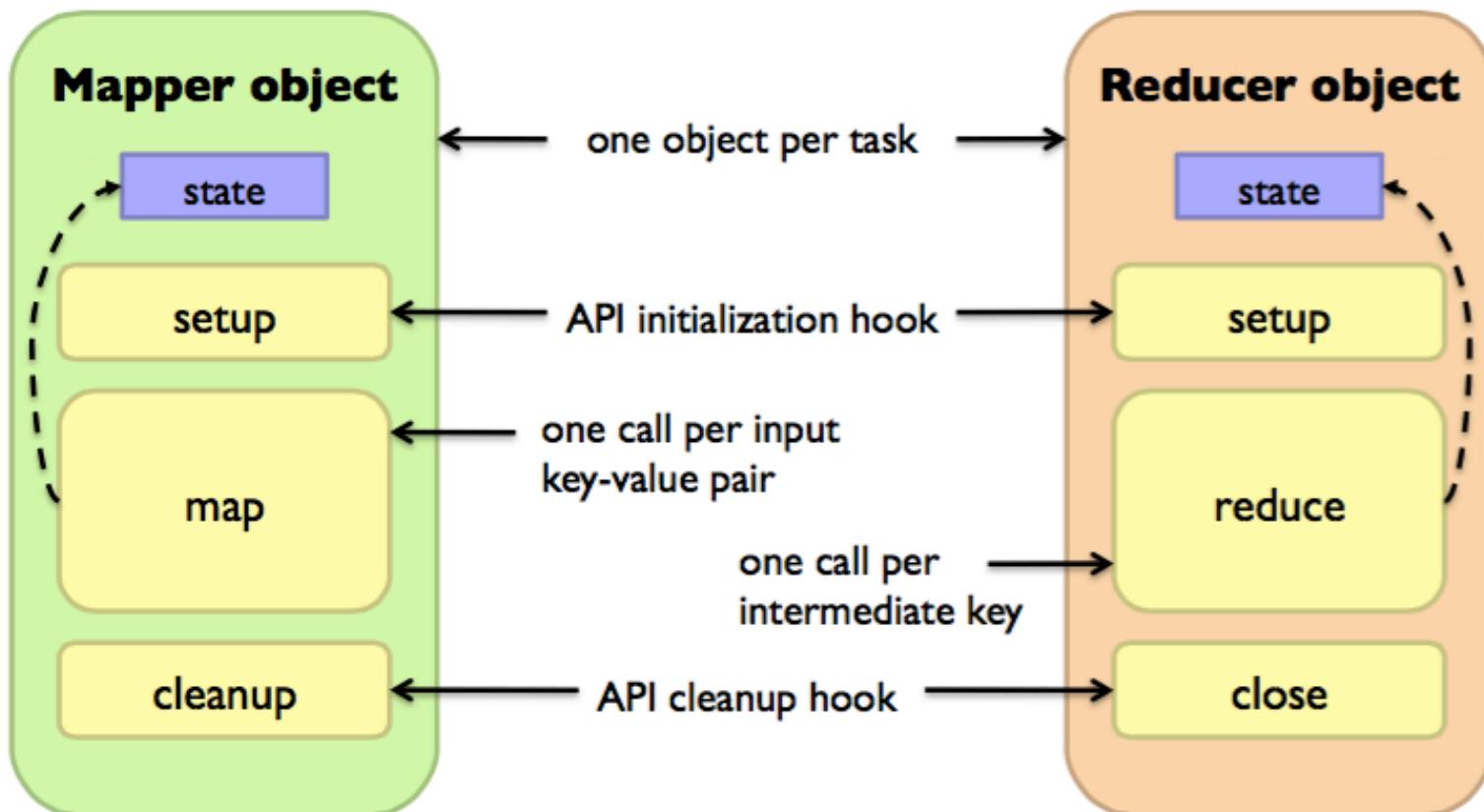
# MapReduce & HDFS



# Shuffle and sort



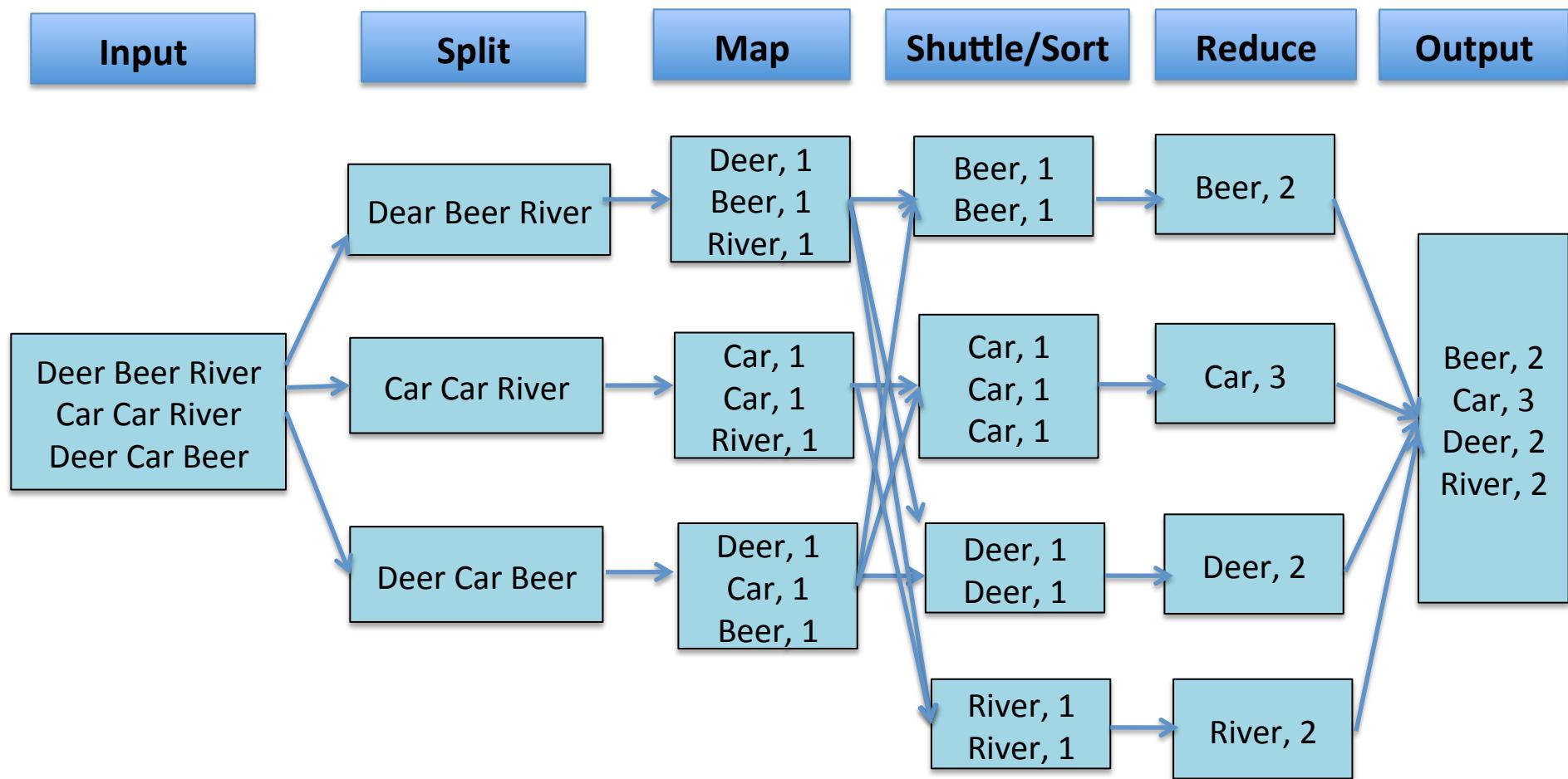
# Preserving state



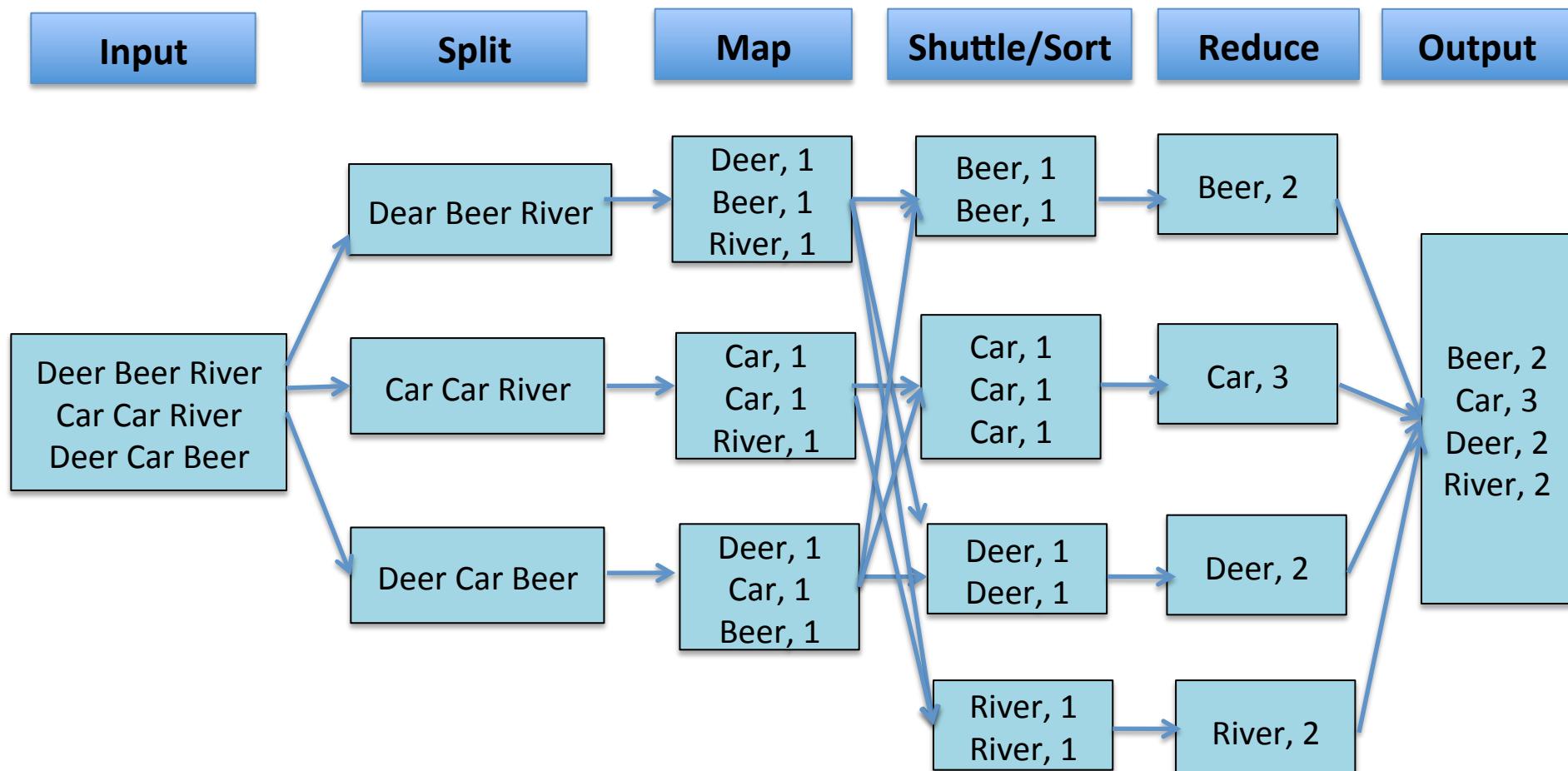
# Implementation don't

- Don't unnecessarily create objects
  - Object creation is costly
  - Garbage collection is costly
- Don't buffer objects
  - Processes have limited heap size (remember, commodity machines)
  - May work for small datasets, but won't scale!

# MapReduce Example: Word Count



# MapReduce Example: Word Count



**Q: What are the Key and Value Pairs of Map and Reduce?**

**Map:** Key=word, Value=1

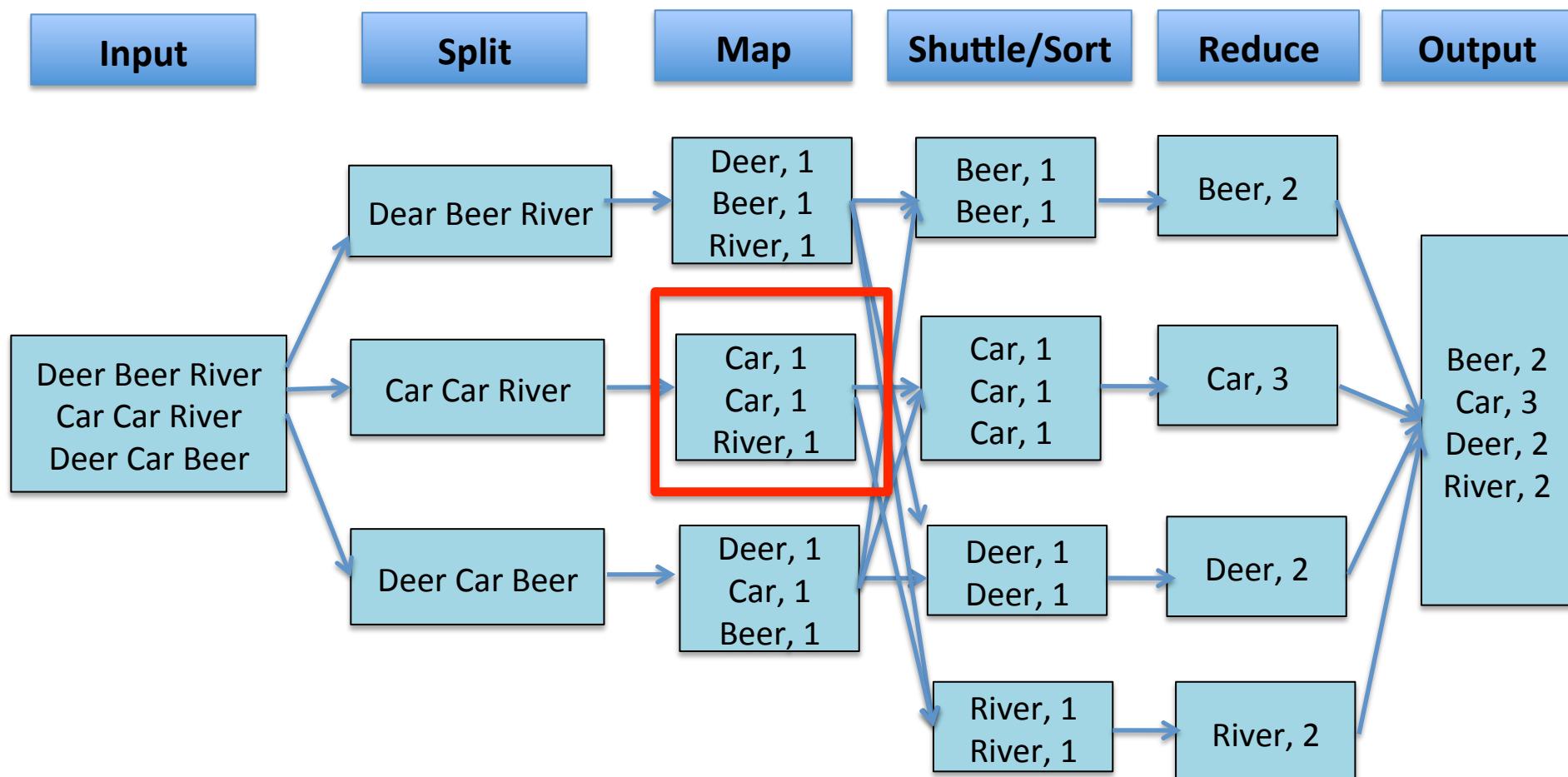
**Reduce:** Key=word, Value=aggregated count

# Word Count: baseline

```
1: class MAPPER
2:   method MAP(docid a, doc d)
3:     for all term t  $\in$  doc d do
4:       EMIT(term t, count 1)

1: class REDUCER
2:   method REDUCE(term t, counts [c1, c2, ...])
3:     sum  $\leftarrow$  0
4:     for all count c  $\in$  counts [c1, c2, ...] do
5:       sum  $\leftarrow$  sum + c
6:     EMIT(term t, count s)
```

# MapReduce Example: Word Count

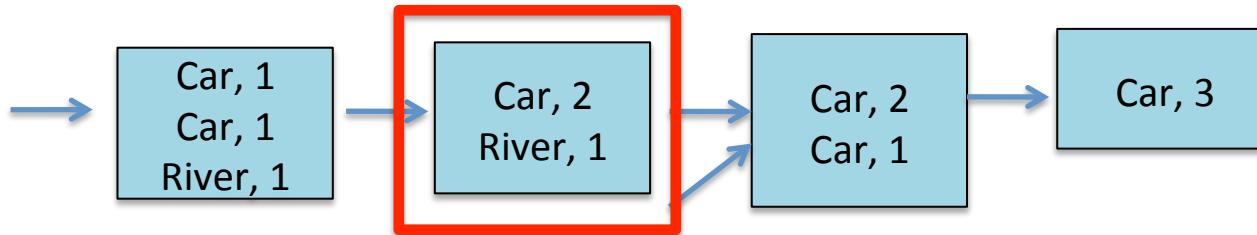


Q: Do you see any place we can improve the efficiency?

Local aggregation at mapper will be able to improve  
MapReduce efficiency.

# MapReduce: Combiner

- **Combiner:** do local aggregation/combine task at mapper



- **Q: What are the benefits of using combiner:**
  - Reduce memory/disk requirement of Map tasks
  - Reduce network traffic
- **Q: Can we remove the reduce function?**
  - No, reducer still needs to process records with same key but from *different mappers*
- **Q: How would you implement combiner?**
  - It is the same as Reducer!

# Word Count: version 1

```
1: class MAPPER
2:     method MAP(docid a, doc d)
3:         H ← new ASSOCIATIVEARRAY
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1                      ▷ Tally counts for entire document
6:         for all term t ∈ H do
7:             EMIT(term t, count H{t})
```

# Word Count: version 2

```
1: class MAPPER
2:   method INITIALIZE
3:      $H \leftarrow$  new ASSOCIATIVEARRAY
4:   method MAP(docid  $a$ , doc  $d$ )
5:     for all term  $t \in$  doc  $d$  do
6:        $H\{t\} \leftarrow H\{t\} + 1$ 
7:   method CLOSE
8:     for all term  $t \in H$  do
9:       EMIT(term  $t$ , count  $H\{t\}$ )
```

Key idea: preserve state across  
input key-value pairs!

▷ Tally counts *across* documents

Are combiners still need?

# Design pattern for local aggregation

- “In-mapper combining”
  - Fold the functionality of the combiner into the mapper by preserving state across multiple map calls
- Advantages
  - Speed
  - Why is this faster than actual combiners?
- Disadvantages
  - Explicit memory management required
  - Potential for order-dependent bugs

# Combiner design

- Combiners and reducers share same method signature
  - Sometimes, reducers can serve as combiners
  - Often, not...
- Remember: combiner are optional optimizations
  - Should not affect algorithm correctness
  - May be run 0, 1, or multiple times
- Example: find average of integers associated with the same key

# Computing the Mean: version 1

```
1: class MAPPER
2:     method MAP(string t, integer r)
3:         EMIT(string t, integer r)
4:
5: class REDUCER
6:     method REDUCE(string t, integers [r1, r2, ...])
7:         sum ← 0
8:         cnt ← 0
9:         for all integer r ∈ integers [r1, r2, ...] do
10:             sum ← sum + r
11:             cnt ← cnt + 1
12:             ravg ← sum/cnt
13:             EMIT(string t, integer ravg)
```

Why can't we use Reducer as Combiner?

# Computing the Mean: version 2

```
1: class MAPPER
2:     method MAP(string  $t$ , integer  $r$ )
3:         EMIT(string  $t$ , integer  $r$ )
4:
5: class COMBINER
6:     method COMBINE(string  $t$ , integers [ $r_1, r_2, \dots$ ])
7:          $sum \leftarrow 0$ 
8:          $cnt \leftarrow 0$ 
9:         for all integer  $r \in$  integers [ $r_1, r_2, \dots$ ] do
10:              $sum \leftarrow sum + r$ 
11:              $cnt \leftarrow cnt + 1$ 
12:         EMIT(string  $t$ , pair ( $sum, cnt$ ))           ▷ Separate sum and count
13:
14: class REDUCER
15:     method REDUCE(string  $t$ , pairs [( $s_1, c_1$ ), ( $s_2, c_2$ ) ...])
16:          $sum \leftarrow 0$ 
17:          $cnt \leftarrow 0$ 
18:         for all pair ( $s, c$ )  $\in$  pairs [( $s_1, c_1$ ), ( $s_2, c_2$ ) ...] do
19:              $sum \leftarrow sum + s$ 
20:              $cnt \leftarrow cnt + c$ 
21:          $r_{avg} \leftarrow sum/cnt$ 
22:         EMIT(string  $t$ , integer  $r_{avg}$ )
```

**Why doesn't this work?**

# Computing the Mean: version 3

```
1: class MAPPER
2:     method MAP(string t, integer r)
3:         EMIT(string t, pair (r, 1))
4:
5: class COMBINER
6:     method COMBINE(string t, pairs [(s1, c1), (s2, c2) ...])
7:         sum ← 0
8:         cnt ← 0
9:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
10:             sum ← sum + s
11:             cnt ← cnt + c
12:         EMIT(string t, pair (sum, cnt))
13:
14: class REDUCER
15:     method REDUCE(string t, pairs [(s1, c1), (s2, c2) ...])
16:         sum ← 0
17:         cnt ← 0
18:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
19:             sum ← sum + s
20:             cnt ← cnt + c
21:         ravg ← sum/cnt
22:         EMIT(string t, pair (ravg, cnt))
```

# Computing the Mean: version 4

```
1: class MAPPER
2:     method INITIALIZE
3:          $S \leftarrow$  new ASSOCIATIVEARRAY
4:          $C \leftarrow$  new ASSOCIATIVEARRAY
5:     method MAP(string  $t$ , integer  $r$ )
6:          $S\{t\} \leftarrow S\{t\} + r$ 
7:          $C\{t\} \leftarrow C\{t\} + 1$ 
8:     method CLOSE
9:         for all term  $t \in S$  do
10:             EMIT(term  $t$ , pair ( $S\{t\}$ ,  $C\{t\}$ ))
```

Are combiners still need?

# Word Count & sorting

- **New Goal:** output all words sorted by their frequencies (total counts) in a document.
- **Question:** How would you adopt the basic word count program to solve it?
- **Solution:**
  - Sort words by their counts in the reducer
  - Problem: what happens if we have more than one reducer?

# Word Count & sorting

- **New Goal:** output all words sorted by their frequencies (total counts) in a document.
- **Question:** How would you adopt the basic word count program to solve it?
- **Solution:**
  - Do two rounds of MapReduce
  - In the 2<sup>nd</sup> round, take the output of WordCount as input but switch key and value pair!
  - Leverage the sorting capability of *shuffle/sort* to do the global sorting!

# Word Count & top K words

- **New Goal:** output the top K words sorted by their frequencies (total counts) in a document.
- **Question:** How would you adopt the basic word count program to solve it?
- **Solution:**
  - Use the solution of previous problem and only grab the top K in the final output
  - Problem: is there a more efficient way to do it?

# Word Count & top K words

- **New Goal:** output the top K words sorted by their frequencies (total counts) in a document.
- **Question:** How would you adopt the basic word count program to solve it?
- **Solution:**
  - Add a sort function to the *reducer* in the first round and only output the top K words
  - Intuition: the global top K must be a local top K in any reducer!

# MapReduce In-class Exercise

- **Problem:** Find the maximum monthly temperature for each year from weather reports
- **Input:** A set of records with format as:  
 $\langle \text{Year/Month}, \text{Average Temperature of that month} \rangle$ 
  - **(200707,100), (200706,90)**
  - **(200508, 90), (200607,100)**
  - **(200708, 80), (200606,80)**
- **Question:** write down the Map and Reduce function to solve this problem
  - Assume we split the input by line

# Mapper and Reducer of Max Temperature

- Map(key, value){  
    // key: line number  
    // value: tuples in a line  
    for each tuple t in value:  
        Emit(t->year, t->temperature);}
- Reduce(key, list of values){  
    // key: year  
    //list of values: a list of monthly temperature  
    int max\_temp = -100;  
    for each v in values:  
        max\_temp= max(v, max\_temp);  
    Emit(key, max\_temp);}

Combiner is the same as Reducer

# MapReduce Example: Max Temperature

Input

(200707,100), (200706,90)  
(200508, 90), (200607,100)  
(200708, 80), (200606,80)

Map

(2007,100), (2007,90)

(2005, 90), (2006,100)

(2007, 80), (2006, 80)

Combine

**(2007,100)**

(2005, 90), (2006,100)

(2007, 80), (2006, 80)

Shuttle/Sort

(2005,[90])

(2006,[100, 80])

(2007,[100, 80])

Reduce

**(2005,90)**

**(2006,100)**  
CSE-HCMUT

**(2007,100)**

# MapReduce In-class Exercise

- **Key-Value Pair of Map and Reduce:**
  - **Map:** (year, temperature)
  - **Reduce:** (year, maximum temperature of the year)
- **Question: How to use the above Map Reduce program (*that contains the combiner*) with slight changes to find the average monthly temperature of the year?**

# Mapper and Reducer of Average Temperature

- Map(key, value){  
    // key: line number  
    // value: tuples in a line  
    for each tuple t in value:  
        Emit(t->year, t->temperature);}
  - Reduce(key, list of values){  
    // key: year  
    // list of values: a list of monthly temperatures  
    int total\_temp = 0;  
    for each v in values:  
        total\_temp= total\_temp+v;  
    Emit(key, total\_temp/size\_of(values));}
- Combiner is the same as Reducer

# MapReduce Example: Average Temperature

Input

(200707,100), (200706,90)  
(200508, 90), (200607,100)  
(200708, 80), (200606,80)

Real average of  
2007: 90

Map

(2007,100), (2007,90)

(2005, 90), (2006,100)

(2007, 80), (2006,80)

Combine

(2007,95)

(2005, 90), (2006,100)

(2007, 80), (2006,80)

Shuttle/Sort

(2005,[90])

(2006,[100, 80])

(2007,[95, 80])

Reduce

(2005,90)

(2006,90)  
CSE-HCMUT

(2007,87.5)

# MapReduce In-class Exercise

- The problem is with the combiner!
- Here is a simple counterexample:
  - $(2007, 100), (2007, 90) \rightarrow (2007, 95)$   
 $(2007, 80) \rightarrow (2007, 80)$
  - Average of the above is:  $(2007, 87.5)$
  - However, the real average is:  $(2007, 90)$
- However, we can do a small trick to get around this
  - Mapper:  $(2007, 100), (2007, 90) \rightarrow (2007, <190, 2>)$   
 $(2007, 80) \rightarrow (2007, <80, 1>)$
  - Reducer:  $(2007, <270, 3>) \rightarrow (2007, 90)$

# MapReduce Example: Average Temperature

Input

(200707,100), (200706,90)  
(200508, 90), (200607,100)  
(200708, 80), (200606,80)

Map

(2007,100), (2007,90)

(2005, 90), (2006,100)

(2007, 80), (2006,80)

Combine

(2007,<190,2>)

(2005, <90,1>),  
(2006, <100,1>)

(2007, <80,1>),  
(2006,<80,1>)

Shuttle/Sort

(2005,[<90,1>])

(2006,[<100,1>, <80,1>])

(2007,[<190,2>, <80,1>])

Reduce

(2005,90)

(2006,90)  
CSE-HCMUT

(2007,90)

# Mapper and Reducer of Average Temperature

- **Map(key, value){**  
    // key: line number  
    // value: tuples in a line  
    for each tuple t in value:  
        Emit(t->year, t->temperature);**}**
- **Reduce (key, list of values){**  
    // key: year  
    // list of values: a list of <temperature sums, counts> tuples  
  
    int total\_temp = 0;  
    int total\_count=0;  
    for each v in values:  
        total\_temp= total\_temp+v->sum;  
        total\_count=total\_count+v->count;  
  
    **Emit(key,total\_temp/total\_count);}**
- **Combine(key, list of values){**  
    // key: year  
    // list of values: a list of monthly temperature  
  
    int total\_temp = 0;  
    for each v in values:  
        total\_temp= total\_temp+v;  
  
    **Emit(key,<total\_temp,size\_of(values)>);**

# MapReduce In-class Exercise

- Functions that can use combiner are called *distributive*:
  - Distributive: Min/Max(), Sum(), Count(), TopK()
  - Non-distributive: Mean(), Median(), Rank()

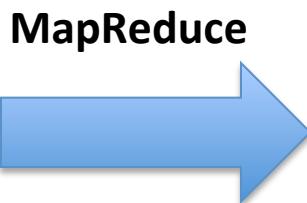
Gray, Jim\*, et al. "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals." Data Mining and Knowledge Discovery 1.1 (1997): 29-53.

\*Jim Gray received Turing Award in 1998

# Map Reduce Problems Discussion

- **Problem 1:** Find Word Length Distribution
- **Statement:** Given a set of documents, use MapReduce to find the length distribution of all words contained in the documents
- **Question:**
  - What are the Mapper and Reducer Functions?

This is a test data for  
the word length  
distribution problem



12: 1  
7: 1  
6: 1  
4: 4  
3: 2  
2: 1  
1: 1

# Mapper and Reducer of Word Length Distribution

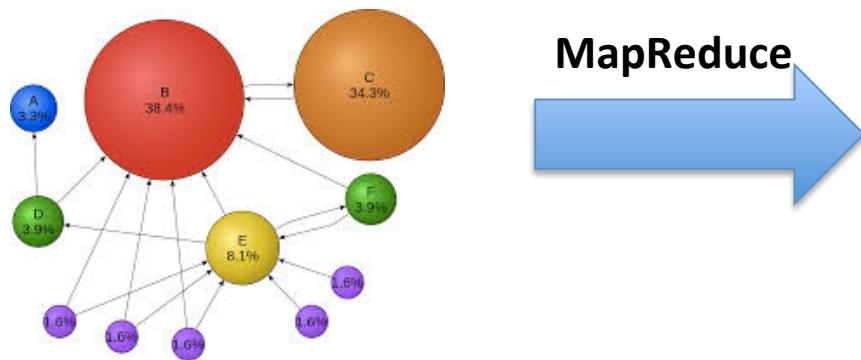
- Map(key, value){  
    // key: document name  
    // value: words in a document  
    for each word w in value:  
        Emit(length(w), w);}
- Reduce(key, list of values){  
    // key: length of a word  
    // list of values: a list of words with the same length  
    Emit(key, size\_of(values));}

# Map Reduce Problems Discussion

- **Problem 1:** Find Word Length Distribution
- **Mapper and Reducer:**
  - Mapper(document)  
  { Emit (**Length(word)**, **word**) }
  - Reducer(output of map)  
  { Emit (**Length(word)**, **Size of (List of words at a particular length)**)}}

# Map Reduce Problems Discussion

- **Problem 2:** Indexing & Page Rank
- **Statement:** Given a set of web pages, each page has a **page rank** associated with it, use Map-Reduce to find, for each word, a list of pages (sorted by rank) that contains that word
- **Question:**
  - What are the Mapper and Reducer Functions?



Word 1: [page x1,  
page x2, ...]

Word 2: [page y1,  
page y2, ...]

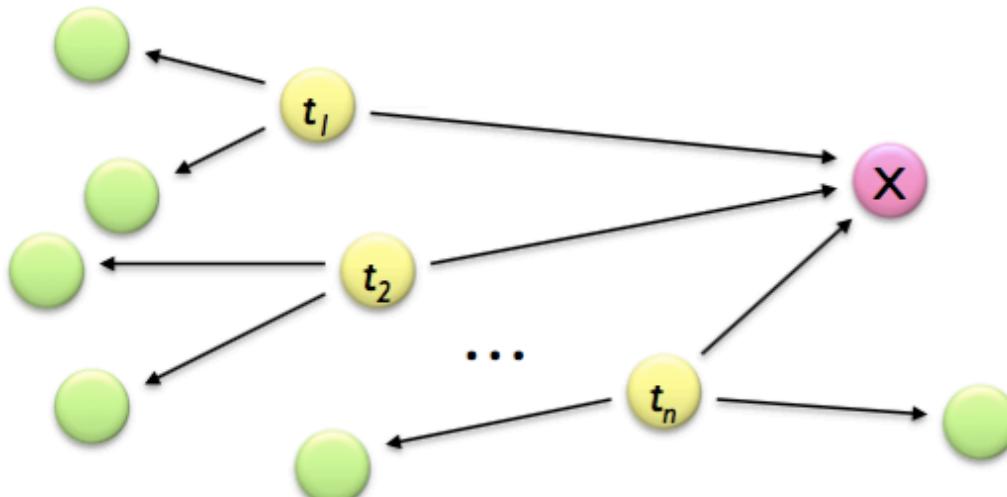
...

# Page Rank

Given page  $x$  with inlinks  $t_1, \dots, t_n$ , where

- $C(t)$  is the out-degree of  $t$
- $\alpha$  is probability of random jump
- $N$  is the total number of nodes in the graph

$$PR(x) = \alpha \left( \frac{1}{N} \right) + (1 - \alpha) \sum_{i=1}^n \frac{PR(t_i)}{C(t_i)}$$



# Mapper and Reducer of Indexing and PageRank

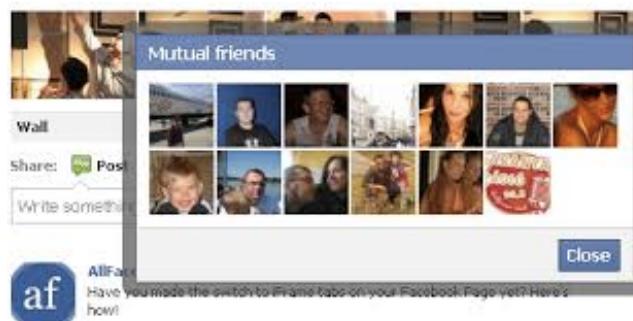
- Map(key, value){  
    // key: a page  
    // value: words in a page  
    for each word w in value:  
        Emit(w, <page\_id, page\_rank>);}
- Reduce(key, list of values){  
    // key: a word  
    // list of values: a list of pages containing that word  
    sorted\_pages=sort(values, page\_rank)  
    Emit(key, sorted\_pages);}

# Map Reduce Problems Discussion

- Problem 2: Indexing and Page Rank
- **Mapper and Reducer:**
  - Mapper(page\_id, <page\_text, page\_rank>)  
  { Emit (word, <page\_id, page\_rank>) }
  - Reducer(output of map)  
  { Emit (word, List of pages contains the word sorted by their page\_ranks)}

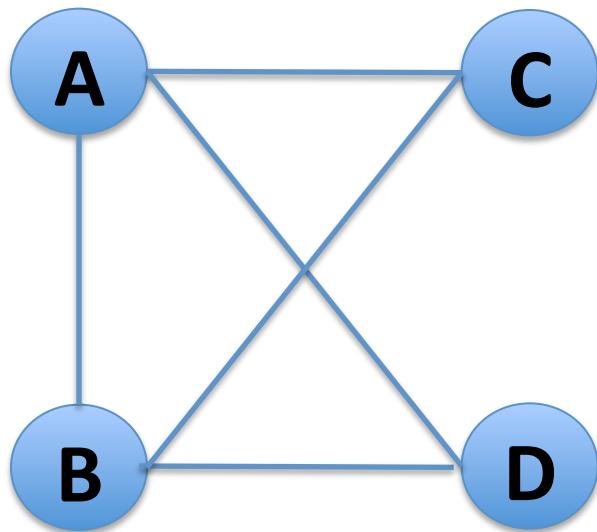
# Map Reduce Problems Discussion

- **Problem 3: Find Common Friends**
- **Statement:** Given a group of people on online social media (e.g., Facebook), each has a list of friends, use Map-Reduce to find common friends of any two persons who are friends
- **Question:**
  - What are the Mapper and Reducer Functions?



# Map Reduce Problems Discussion

- Problem 3: Find Common Friends
- Simple example:



Input:

A -> B,C,D

B-> A,C,D

C-> A,B

D->A,B

MapReduce

Output:

(A ,B) -> C,D

(A,C) -> B

(A,D) -> ..

....

# Mapper and Reducer of Common Friends

- Map(key, value){  
    // key: person\_id  
    // value: the list of friends of the person  
    for each friend f\_id in value:  
        Emit(<person\_id, f\_id>, value);}
- Reduce(key, list of values){  
    // key: <friend pair>  
    // list of values: a set of friend lists related with the friend pair  
    for v1, v2 in values:  
        common\_friends = v1 intersects v2;  
    Emit(key, common\_friends);}

# Map Reduce Problems Discussion

- **Problem 3:** Find Common Friends
- **Mapper and Reducer:**
  - Mapper(friend list of a person)
    - { for each person in the friend list:  
    Emit (<friend pair>, <list of friends>) }
  - Reducer(output of map)
    - { Emit (<friend pair>, **Intersection of two (i.e, the one in friend pair) friend lists**) }

# Map Reduce Problems Discussion

- Problem 3: Find Common Friends
- Mapper and Reducer:

Input:

A -> B,C,D  
B-> A,C,D  
C-> A,B  
D->A,B

Map:

(A,B) -> B,C,D  
(A,C) -> B,C,D  
(A,D) -> B,C,D  
**(A,B) -> A,C,D**  
(B,C) -> A,C,D  
(B,D) -> A,C,D  
(A,C) -> A,B  
(B,C) -> A,B  
(A,D) -> A,B  
(B,D) -> A,B

Reduce:

**(A,B)** -> C,D  
(A,C) -> B  
(A,D) -> B  
(B,C) -> A  
(B,D) -> A

Suggest  
Friends ☺

*Enjoy MR and Hadoop* ☺

