



北京航空航天大学
COLLEGE OF SOFTWARE 软件学院
BEIHANG UNIVERSITY

Chapter 2

Parallel Hardware and Parallel Software

软件学院 邵兵

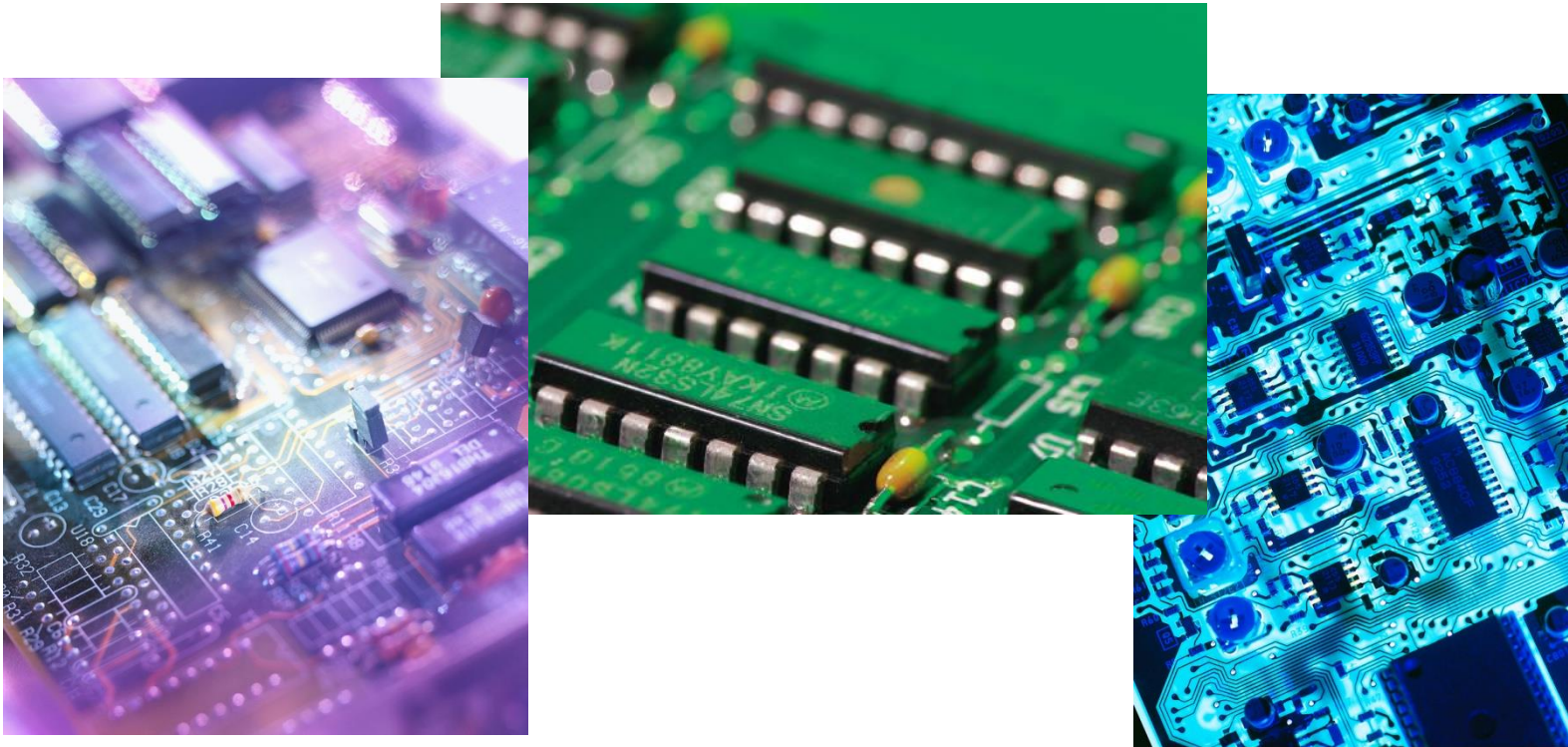
2022年3月11日

Contents

- 1. Some background**
- 2. Modifications to the von Neumann model**
- 3. Interconnection networks**
- 4. Parallel software**
- 5. Performance**
- 6. Parallel program design**



1. SOME BACKGROUND

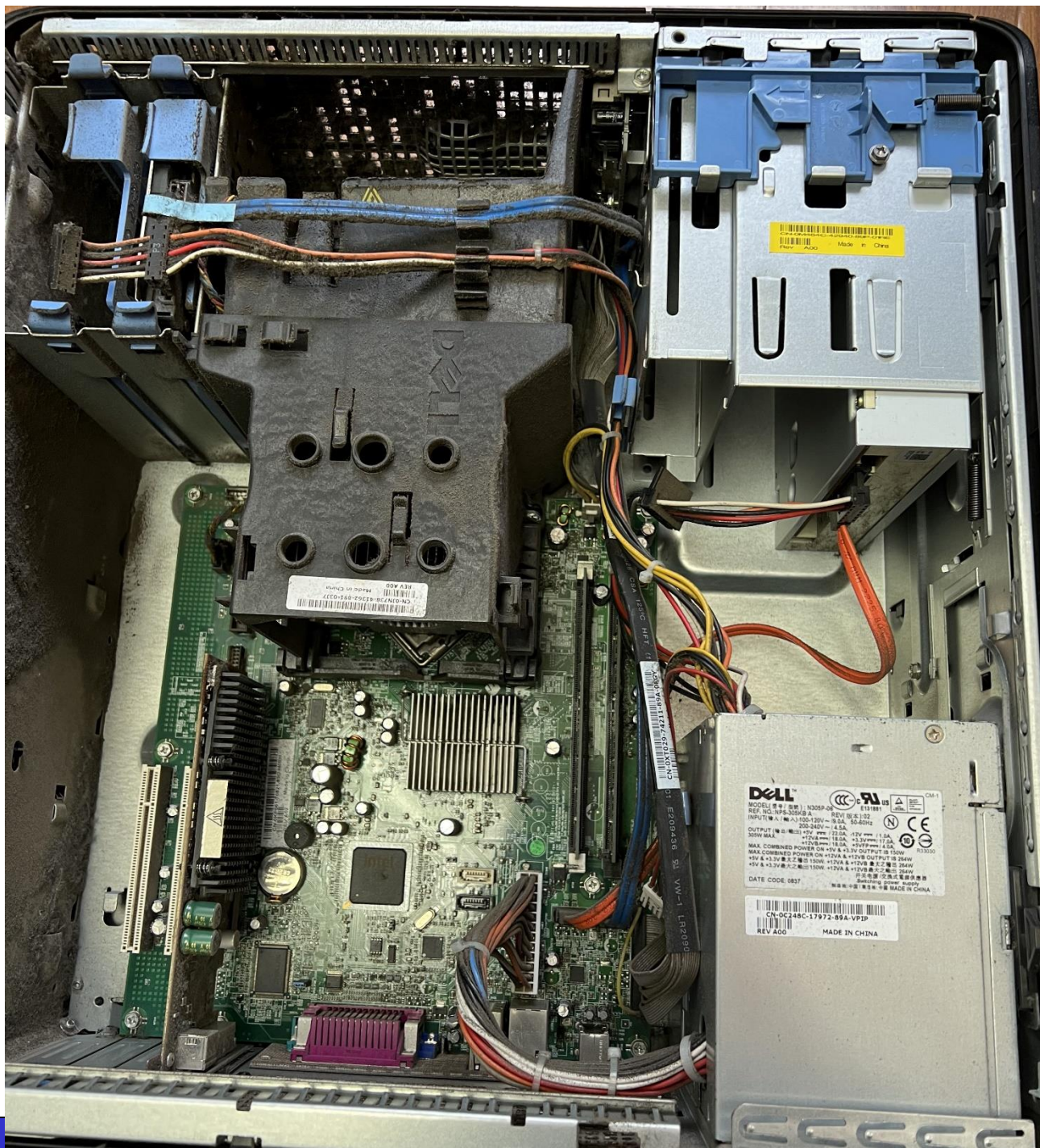


北京航空航天大学
COLLEGE OF SOFTWARE
BEIHANG UNIVERSITY 软件学院

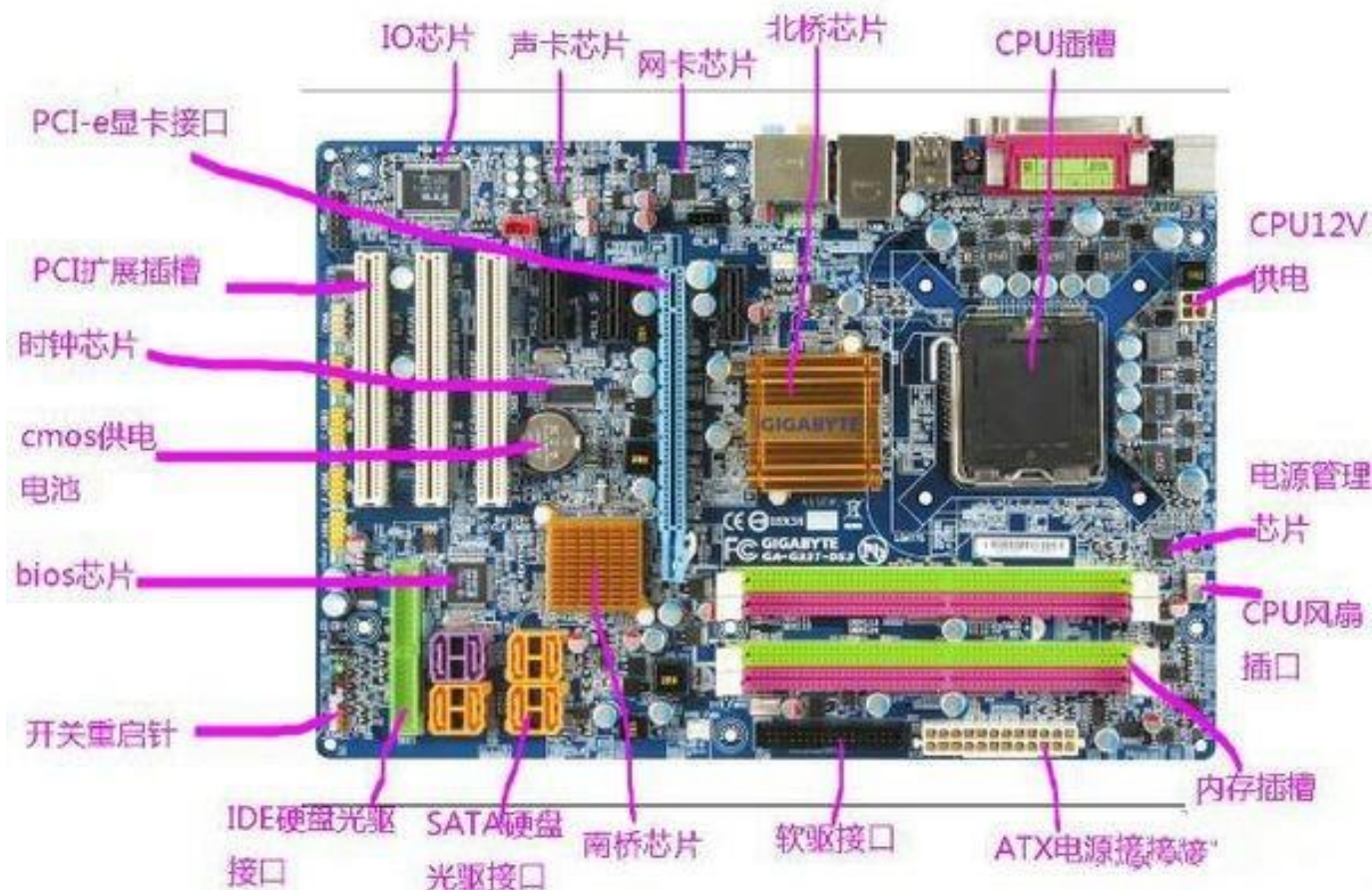
计算机主板

问题:

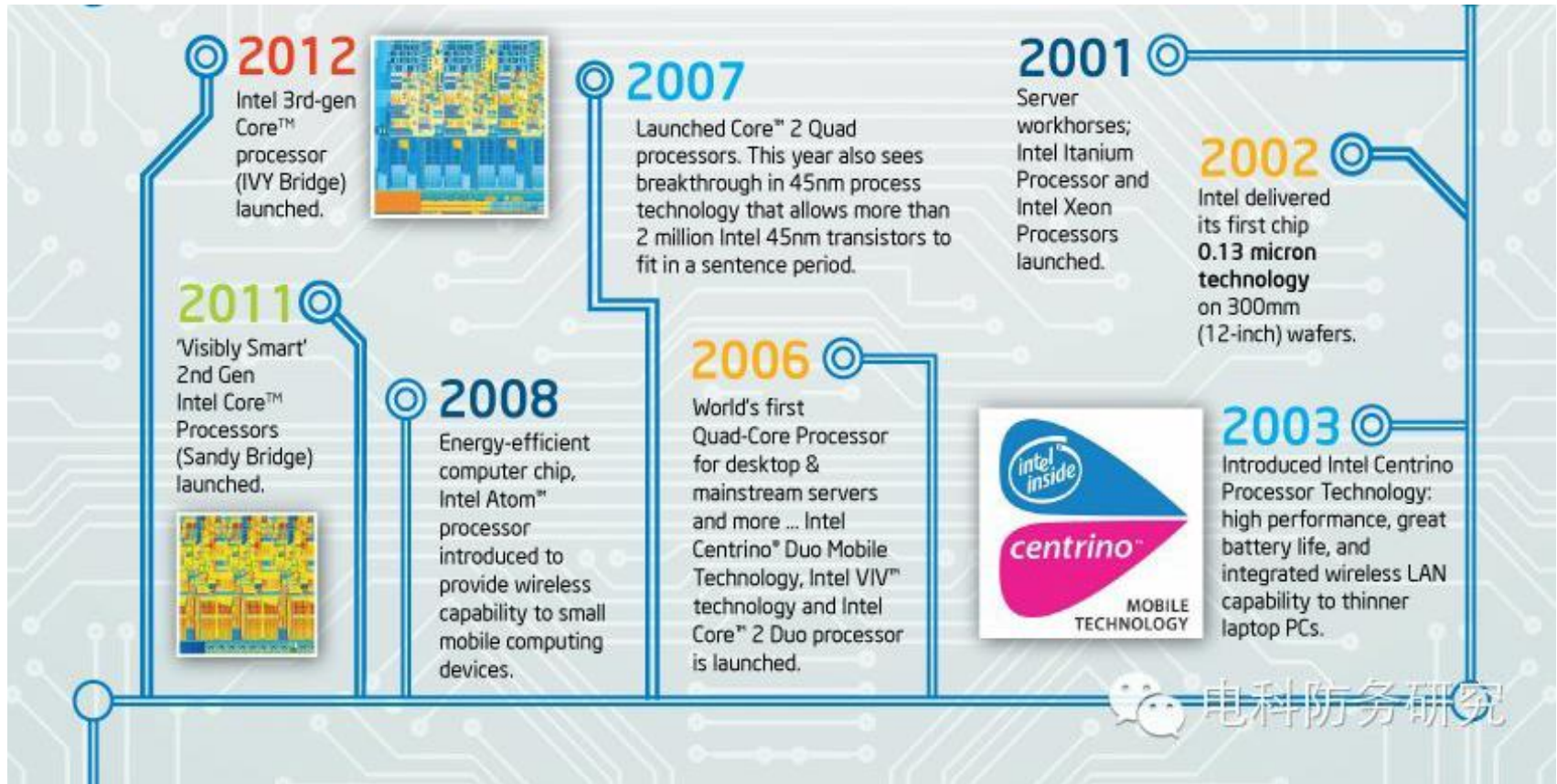
- 计算机里为什么没有A、B盘?
- 硬盘不够用了怎么办?
- 如果嫌内存小了怎么办?
- 如果计算机时钟一直显示某个过去的时间, 哪里出问题了?
- 如果买了GPU卡, 应当往哪里插?

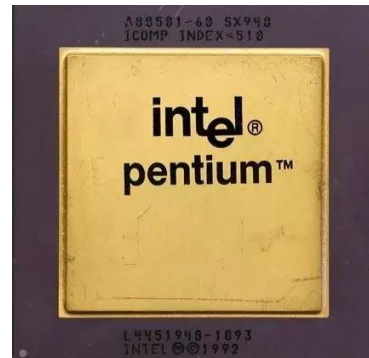
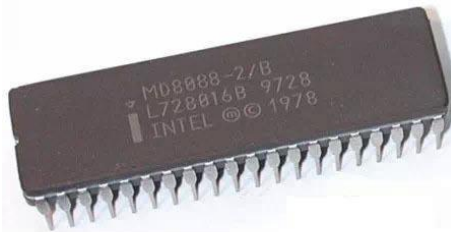
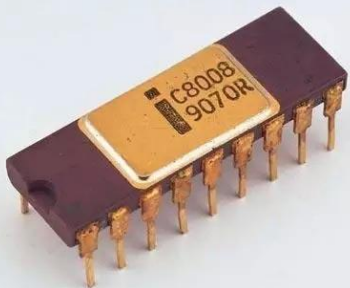
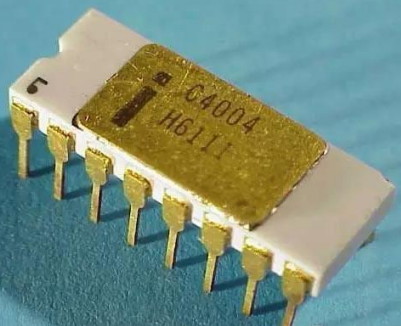


计算机主板



Evolution of the processor





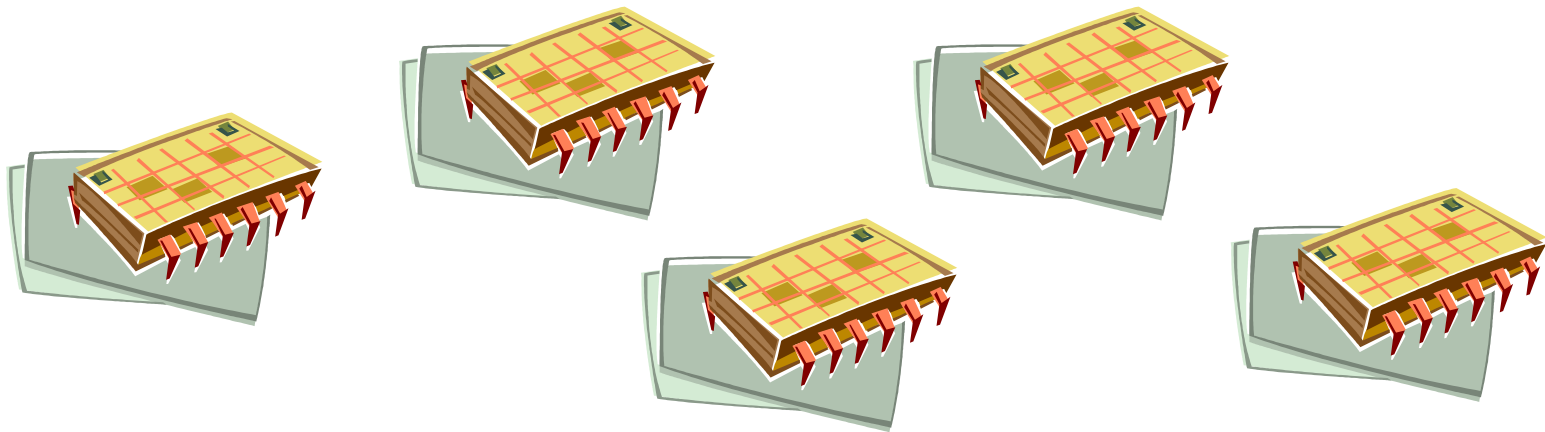
Changing times

- From 1986 – 2002, microprocessors were speeding like a rocket, increasing in performance an average of 50% per year.
- Since then, it's dropped to about 20% increase per year.

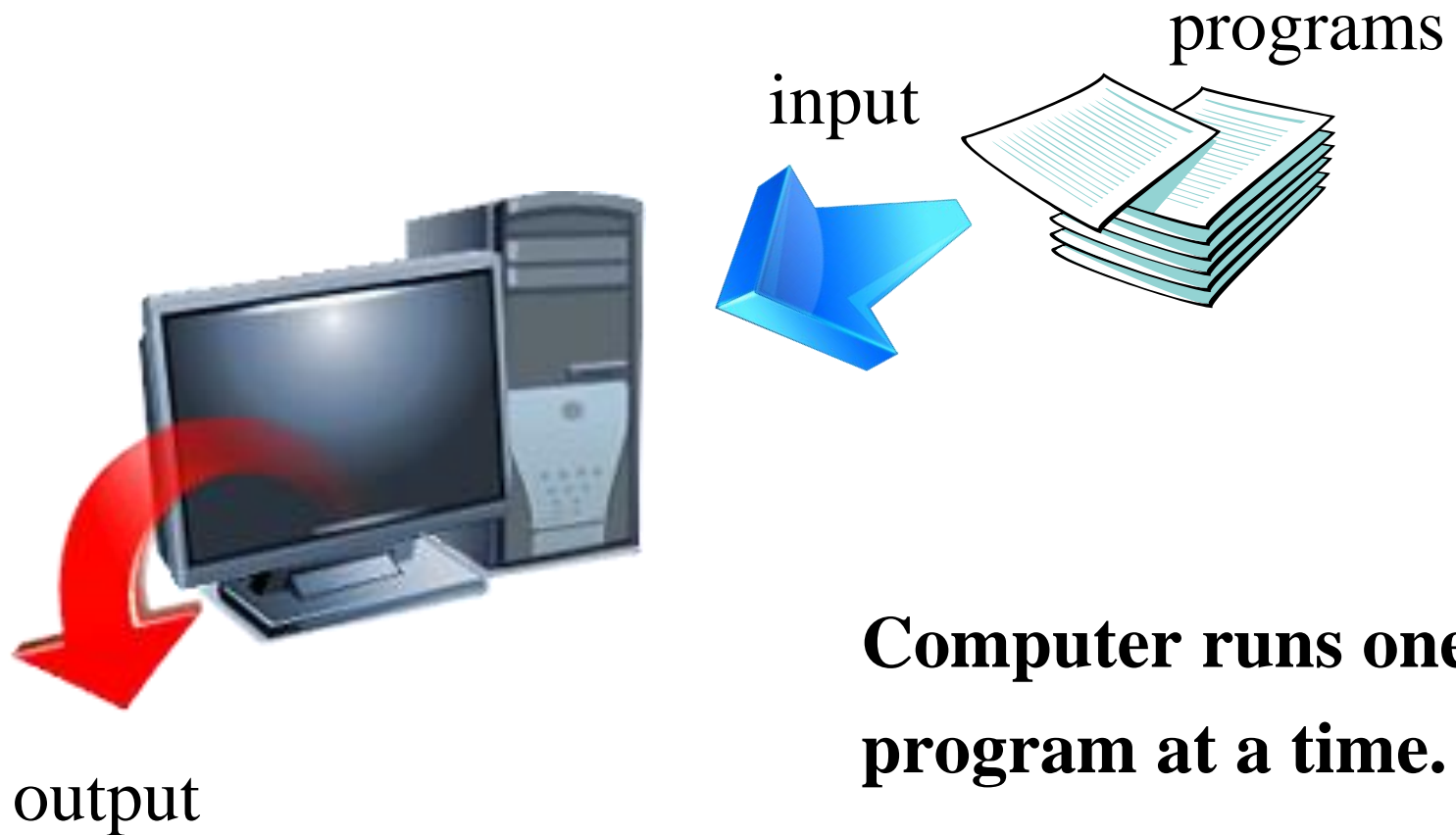


An intelligent solution

- Instead of designing and building faster microprocessors, put multiple processors on a single integrated circuit.



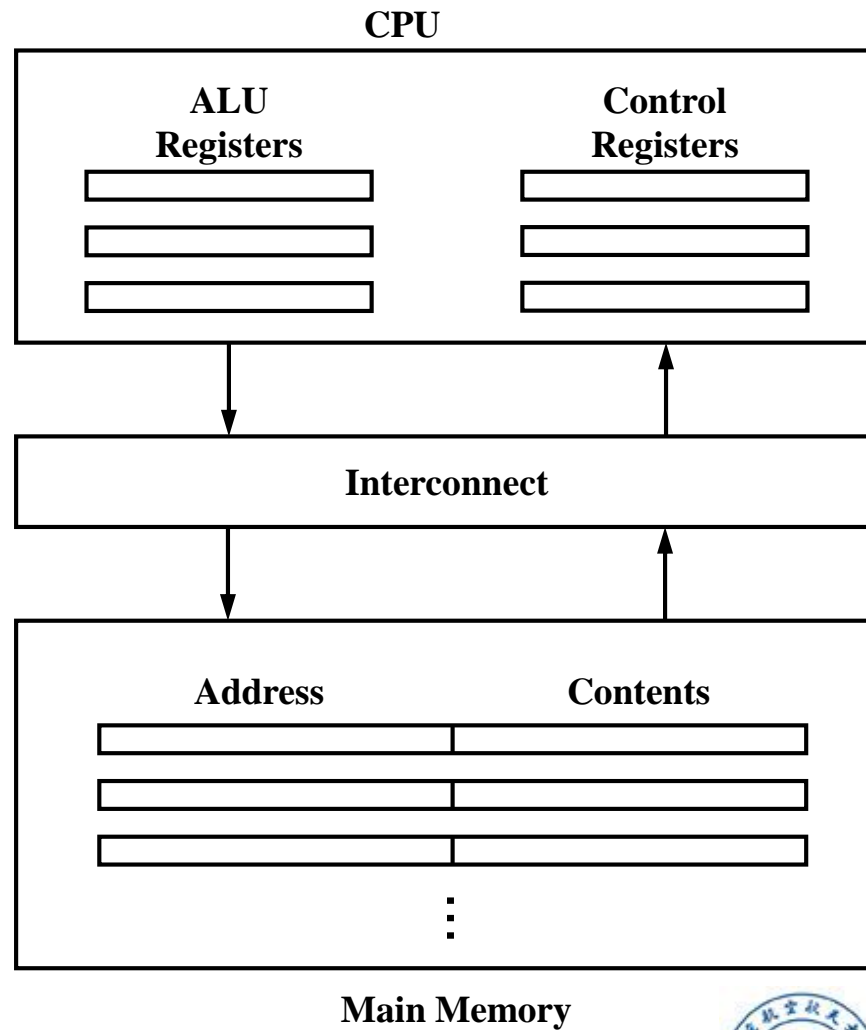
Serial hardware and software



Computer runs one program at a time.

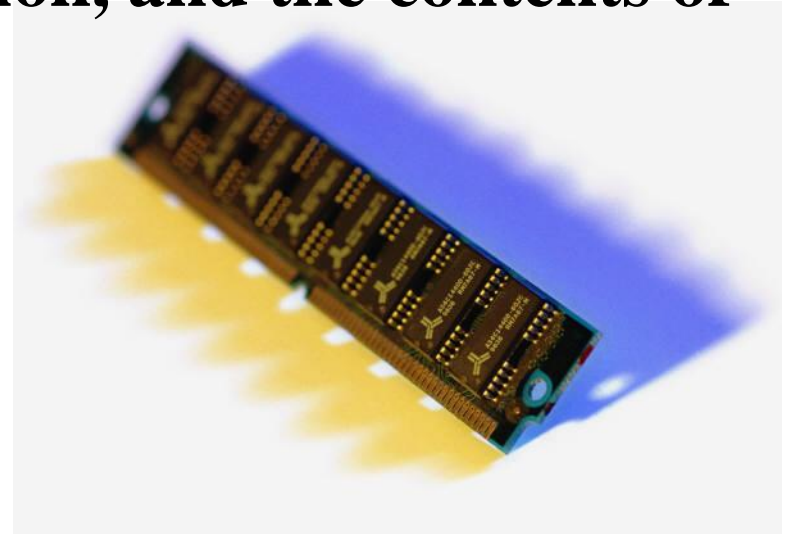


The Von Neumann Architecture



Main memory

- This is a collection of locations, each of which is capable of storing both instructions and data.
- Every location consists of an address, which is used to access the location, and the contents of the location.



Central processing unit (CPU)

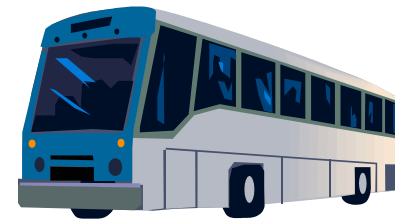
Divided into two parts.

- **Control unit**
 - responsible for deciding which instruction in a program should be executed. (*the boss*)
- **Arithmetic and logic unit (ALU)**
 - responsible for executing the actual instructions. (*the worker*)

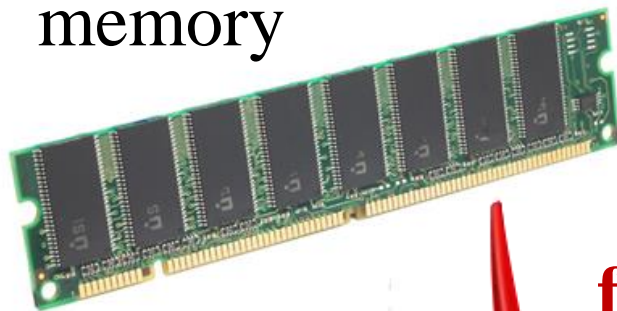


Key terms

- **Register – very fast storage, part of the CPU.**
- **Program counter – stores address of the next instruction to be executed.**
- **Bus – wires and hardware that connects the CPU and memory.**



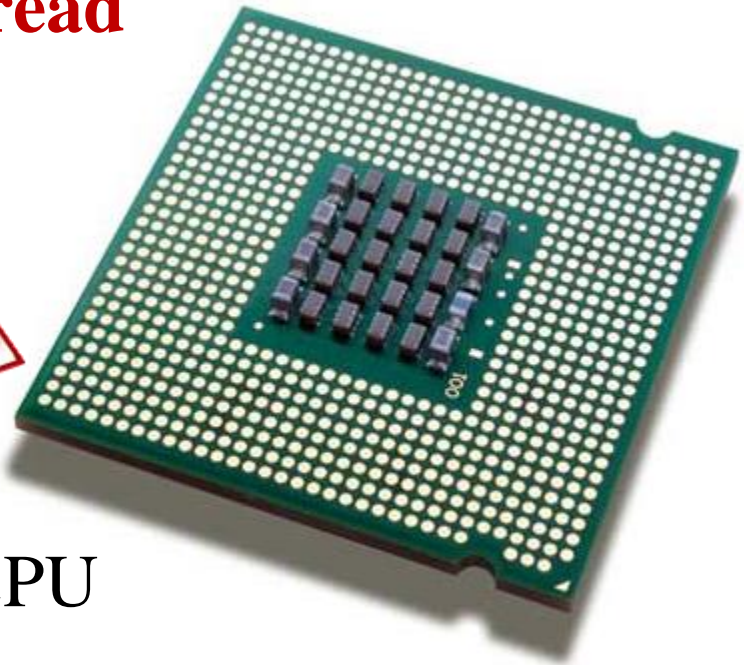
memory



fetch/read

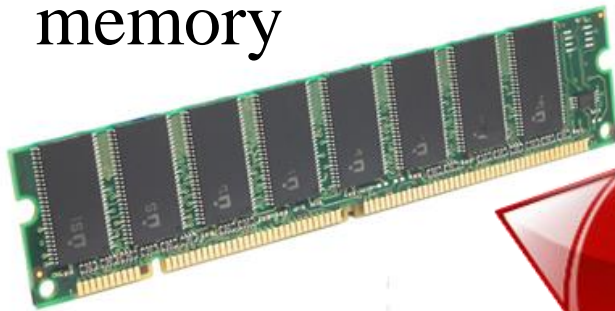


CPU



北京航空航天大学
COLLEGE OF SOFTWARE
BEIHANG UNIVERSITY 软件学院

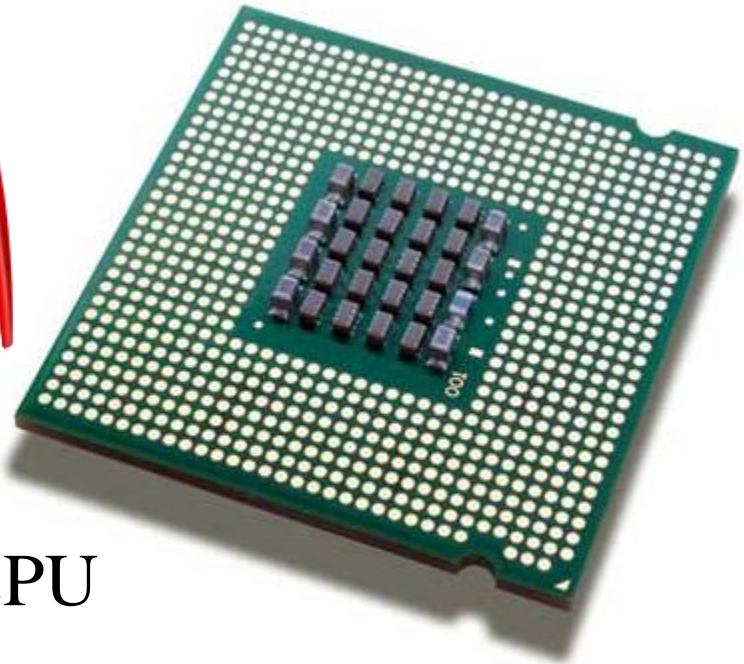
memory



write/store

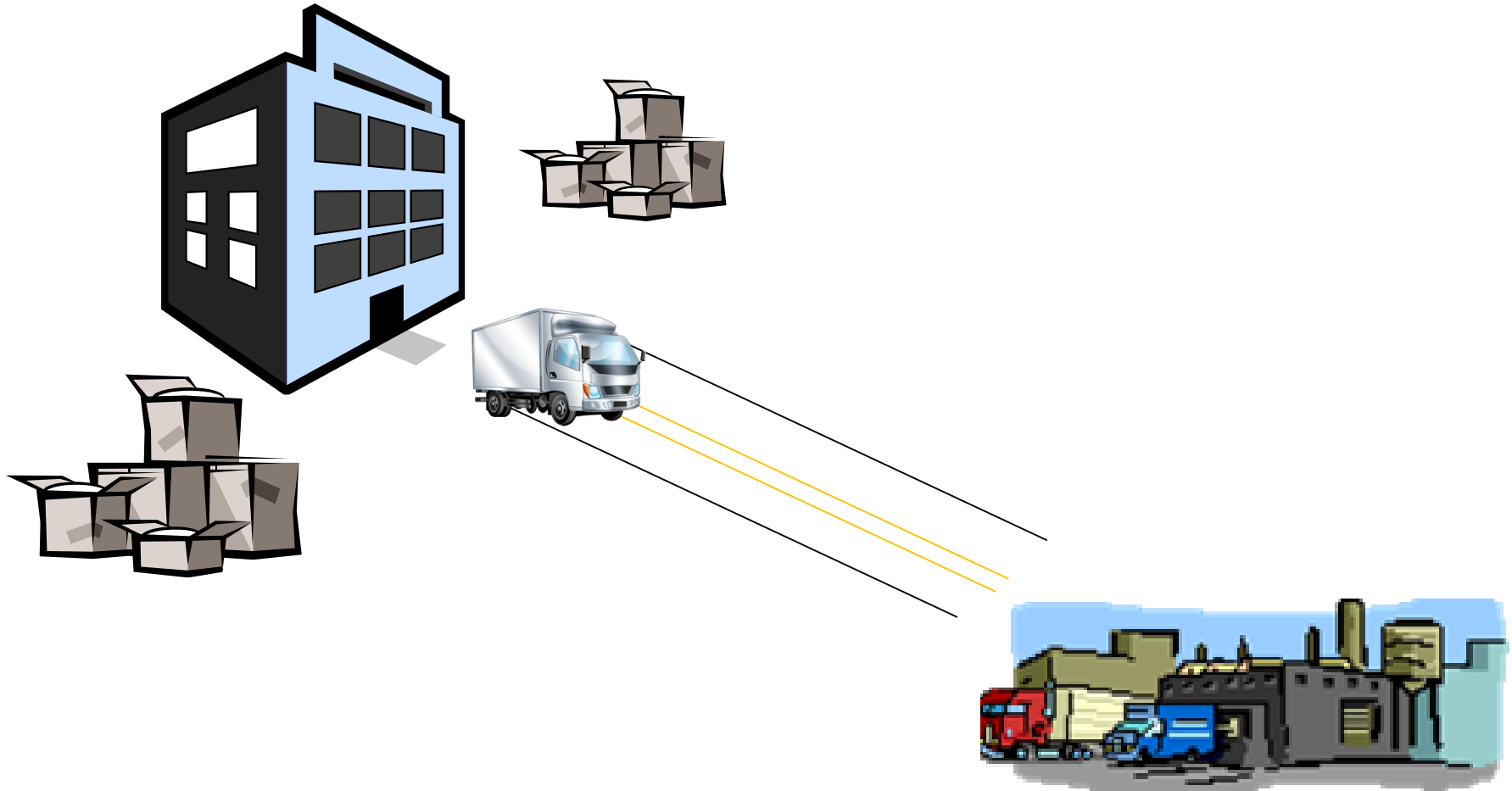


CPU



北京航空航天大学
COLLEGE OF SOFTWARE
BEIHANG UNIVERSITY 软件学院

von Neumann bottleneck



An operating system “process”

- An instance of a computer program that is being executed.
- Components of a process:
 - The executable machine language program.
 - A block of memory.
 - Descriptors of resources the OS has allocated to the process.
 - Security information.
 - Information about the state of the process.



Multitasking

- Gives the illusion that a single processor system is running multiple programs simultaneously.
- Each process takes turns running. (time slice)
- After its time is up, it waits until it has a turn again. (blocks)

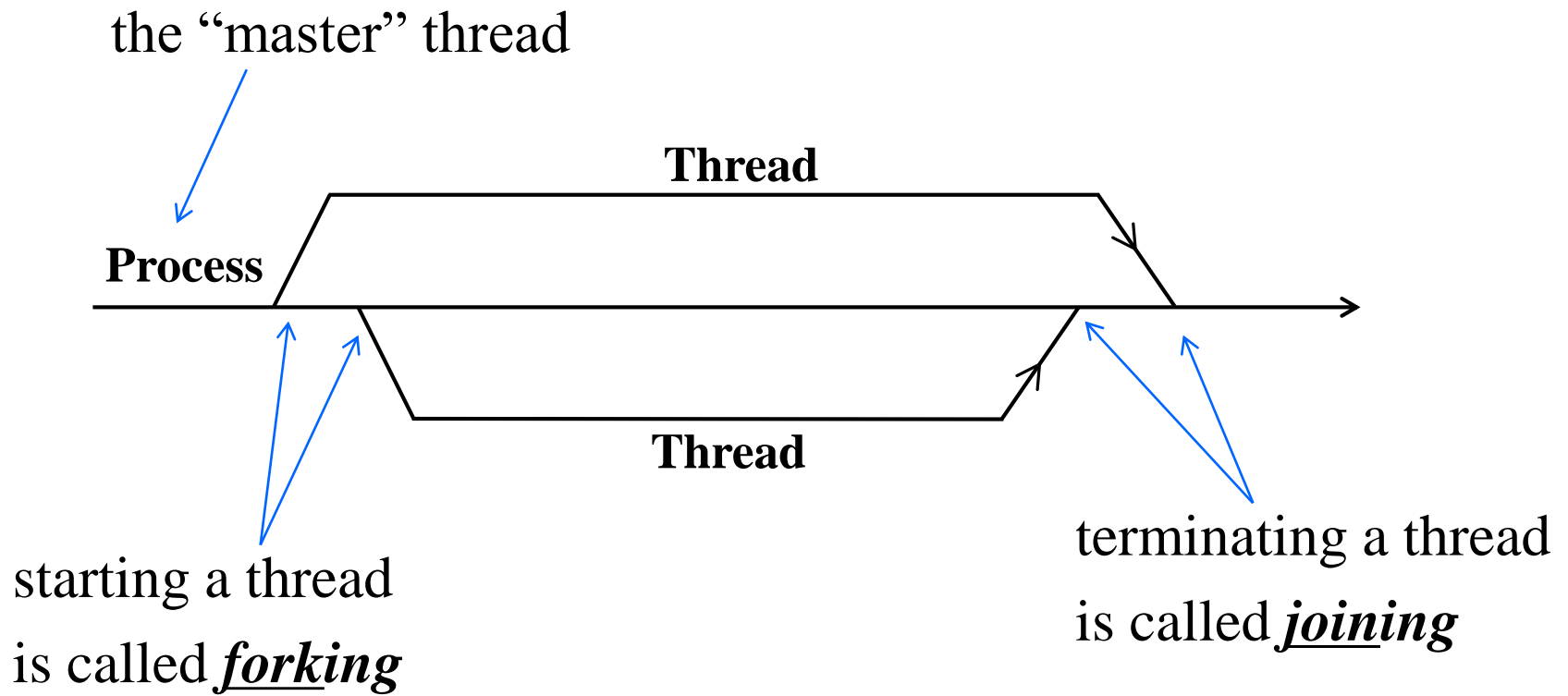


Threading

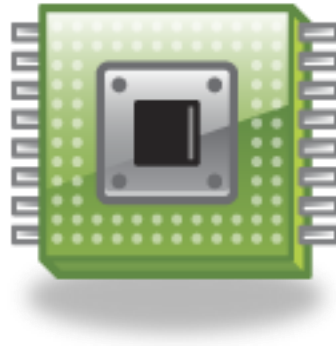
- **Threads are contained within processes.**
- **They allow programmers to divide their programs into (more or less) independent tasks.**
- **The hope is that when one thread blocks because it is waiting on a resource, another will have work to do and can run.**



A process and two threads

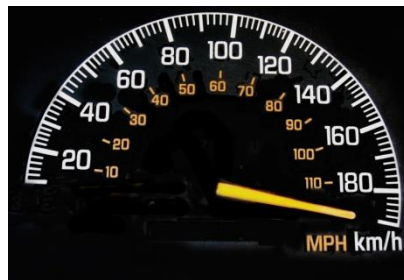


2. MODIFICATIONS TO THE VON NEUMANN MODEL



Basics of caching

- A collection of memory locations that can be accessed in less time than some other memory locations.
- A CPU cache is typically located on the same chip, or one that can be accessed much faster than ordinary memory.



Principle of locality

- Accessing one location is followed by an access of a nearby location.
- **Spatial locality** – accessing a nearby location.
- **Temporal locality** – accessing in the near future.



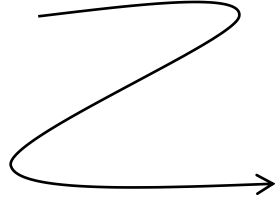
Principle of locality

```
float z[1000];  
...  
sum = 0.0;  
for (i = 0; i < 1000; i++)  
    sum += z[i];
```



Levels of Cache

smallest & fastest

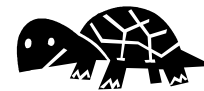


L1



L2

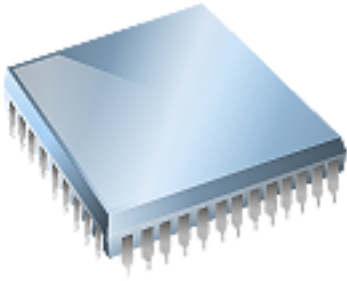
L3



largest & slowest



Cache hit



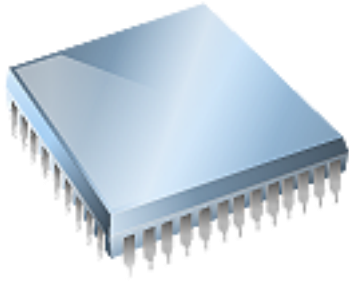
fetch **x**

L1 **x** **sum**

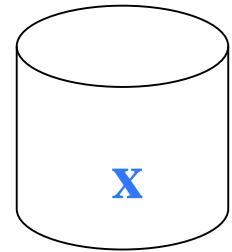
L2 **y** **z** **total**

L3 **A[]** **radius** **r1** **center**

Cache miss



fetch **x**



**main
memory**

L1 **y** **sum**

L2 **r1** **z** **total**

L3 **A[]** **radius** **center**



北京航空航天大学
COLLEGE OF SOFTWARE **软件学院**
BEIHANG UNIVERSITY

Issues with cache

- When a CPU **writes** data to cache, the value in cache may be inconsistent with the value in main memory.
- **Write-through** caches handle this by updating the data in main memory at the time it is written to cache.
- **Write-back** caches mark data in the cache as **dirty**. When the cache line is replaced by a new cache line from memory, the **dirty** line is written to memory.



Cache mappings

- **Full associative** – a new line can be placed at any location in the cache.
- **Direct mapped** – each cache line has a unique location in the cache to which it will be assigned.
- **n -way set associative** – each cache line can be place in one of n different locations in the cache.



n-way set associative

- When more than one line in memory can be mapped to several different locations in cache we also need to be able to decide which line should be replaced or **evicted** (赶走).



Example

Assignments of a 16-line main memory to a 4-line cache

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2 or 3	0	0 or 1
1	0, 1, 2 or 3	1	2 or 3
2	0, 1, 2 or 3	2	0 or 1
3	0, 1, 2 or 3	3	2 or 3
4	0, 1, 2 or 3	0	0 or 1
5	0, 1, 2 or 3	1	2 or 3
6	0, 1, 2 or 3	2	0 or 1
7	0, 1, 2 or 3	3	2 or 3
8	0, 1, 2 or 3	0	0 or 1
9	0, 1, 2 or 3	1	2 or 3
10	0, 1, 2 or 3	2	0 or 1
11	0, 1, 2 or 3	3	2 or 3
12	0, 1, 2 or 3	0	0 or 1
13	0, 1, 2 or 3	1	2 or 3
14	0, 1, 2 or 3	2	0 or 1
15	0, 1, 2 or 3	3	2 or 3

Caches and programs

```
double A[MAX][MAX], x[MAX],
. . .
/* Initialize A and x, assign
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
        y[i] += A[i][j]*x[j];

. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
    for (i = 0; i < MAX; i++)
        y[i] += A[i][j]*x[j];
```

Cache line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]



Instruction Level Parallelism (ILP)

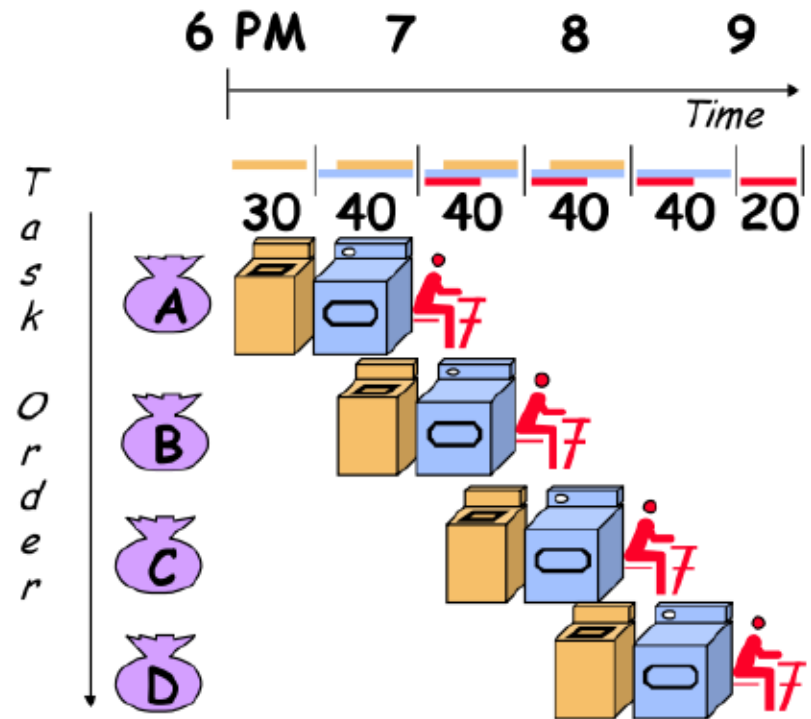
- Attempts to improve processor performance by having multiple processor components or functional units simultaneously executing instructions.
- Pipelining - functional units are arranged in stages.
- Multiple issue - multiple instructions can be simultaneously initiated.



Pipelining

Dave Patterson's Laundry example: 4 people doing laundry
wash (30 min) + dry (40 min) + fold (20 min) = 90 min **Latency**

- In this example:
 - Sequential execution takes $4 * 90\text{min} = 6 \text{ hours}$
 - Pipelined execution takes $30 + 4 * 40 + 20 = 3.5 \text{ hours}$
- Bandwidth = loads/hour
 - BW = 4/6 Non-pipelining
 - BW = 4/3.5 pipelining
 - BW ≤ 1.5 pipelining, when more total loads
- Pipelining helps **bandwidth** but not **latency** (90 min)
- Bandwidth limited by **slowest** pipeline stage
- Potential speedup = **Number of pipe stages**



Pipelining example (1)

Add the floating point numbers 9.87×10^4 and 6.54×10^3

Time	Operation	Operand 1	Operand 2	Result
0	Fetch operands	9.87×10^4	6.54×10^3	
1	Compare exponents	9.87×10^4	6.54×10^3	
2	Shift on operand	9.87×10^4	0.654×10^4	
3	Add	9.87×10^4	0.654×10^4	10.524×10^4
4	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
5	Round result	9.87×10^4	0.654×10^4	1.05×10^5
6	Store result	9.87×10^4	0.654×10^4	1.05×10^5



Pipelining example (2)

```
float x[1000], y[1000], z[1000];
```

```
. . .
```

```
for (i = 0; i < 1000; i++)  
    z[i] = x[i] + y[i];
```

- Assume each operation takes one nanosecond (10^{-9} seconds).
- This for loop takes about 7000 nanoseconds.



Pipelining example (3)

- Divide the floating point adder into 7 separate pieces of hardware or functional units.
- First unit fetches two operands, second unit compares exponents, etc.
- Output of one functional unit is input to the next.



Pipelining example (4)

Pipelined Addition. Numbers in the table are subscripts of operands/results.

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999



Pipelining example (5)

- One floating point addition still takes 7 nanoseconds.
- But 1000 floating point additions now takes 1006 nanoseconds!

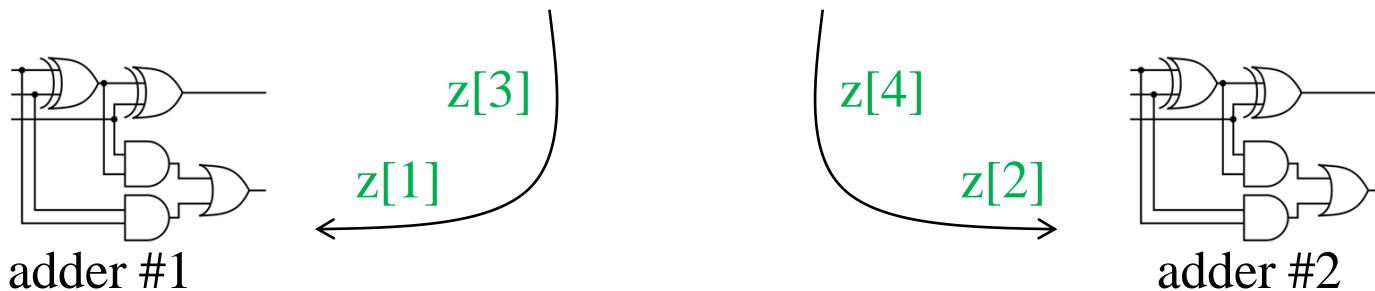


北京航空航天大学
COLLEGE OF SOFTWARE
BEIHANG UNIVERSITY 软件学院

Multiple Issue (1)

- Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program.

```
for (i = 0; i < 1000; i++)  
    z[i] = x[i] + y[i];
```



Multiple Issue (2)

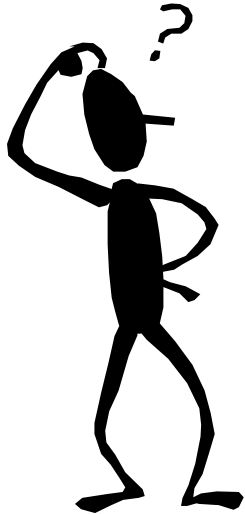
- **static multiple issue - functional units are scheduled at compile time.**
- **dynamic multiple issue – functional units are scheduled at run-time.**

superscalar



Speculation (1)

- In order to make use of multiple issue, the system must find instructions that can be executed simultaneously.

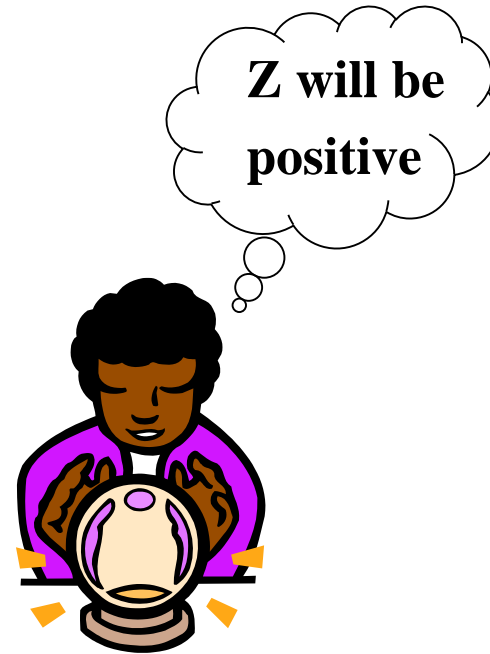


- In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess.



Speculation (2)

```
z = x + y ;  
if (z > 0)  
    w = x ;  
else  
    w = y ;
```



**If the system speculates incorrectly,
it must go back and recalculate $w = y$.**



Hardware multithreading (1)

- **There aren't always good opportunities for simultaneous execution of different threads.**
- **Hardware multithreading provides a means for systems to continue doing useful work when the task being currently executed has stalled.**
 - **Ex., the current task has to wait for data to be loaded from memory.**



Hardware multithreading (2)

- **Fine-grained** - the processor switches between threads after each instruction, skipping threads that are stalled.
 - Pros: potential to avoid wasted machine time due to stalls.
 - Cons: a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction.



Hardware multithreading (3)

- **Coarse-grained** - only switches threads that are stalled waiting for a time-consuming operation to complete.
 - Pros: switching threads doesn't need to be nearly instantaneous.
 - Cons: the processor can be idled on shorter stalls, and thread switching will also cause delays.

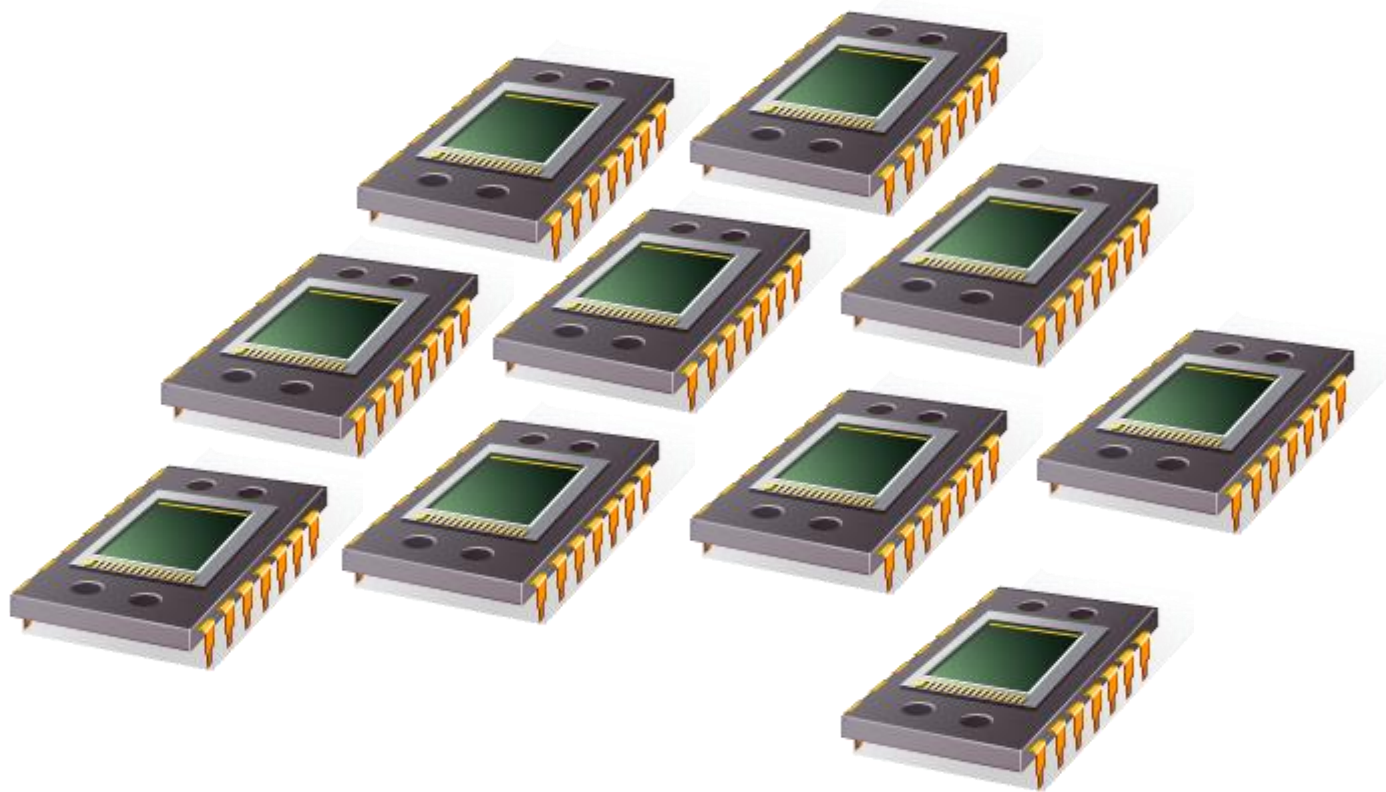


Hardware multithreading (4)

- **Simultaneous multithreading (SMT 同步多线程)**
 - a variation on fine-grained multithreading. It allows multiple threads to make use of the multiple functional units.



3. INTERCONNECTION NETWORKS



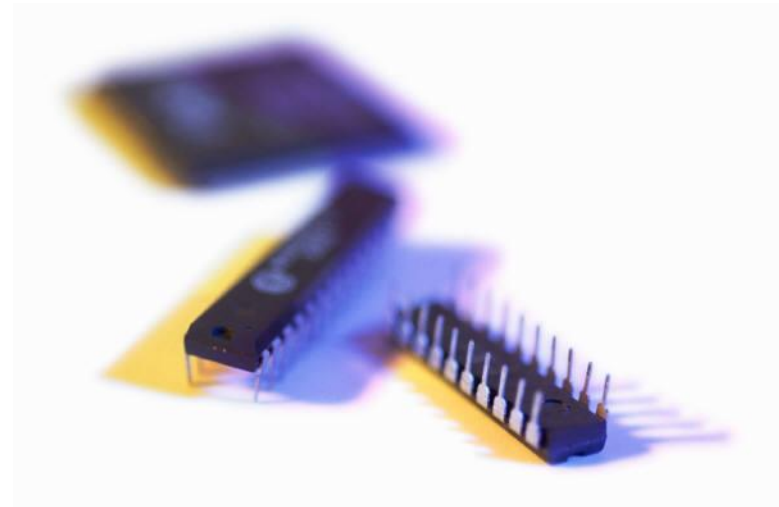
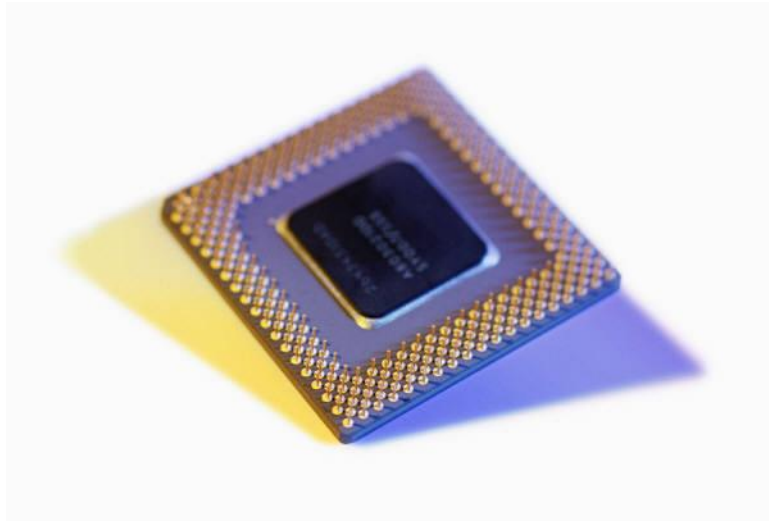
Shared Memory System (1)

- A collection of autonomous processors is connected to a memory system via an interconnection network.
- Each processor can access each memory location.
- The processors usually communicate implicitly by accessing shared data structures.

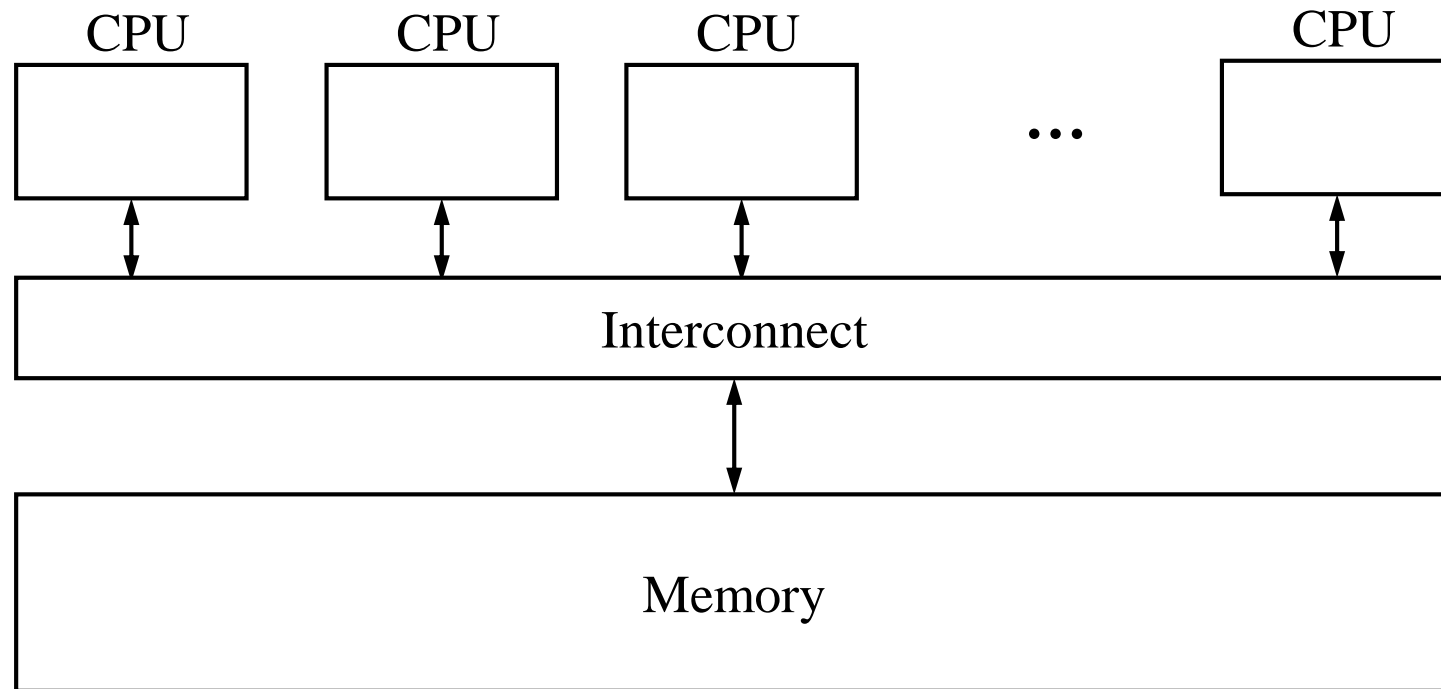


Shared Memory System (2)

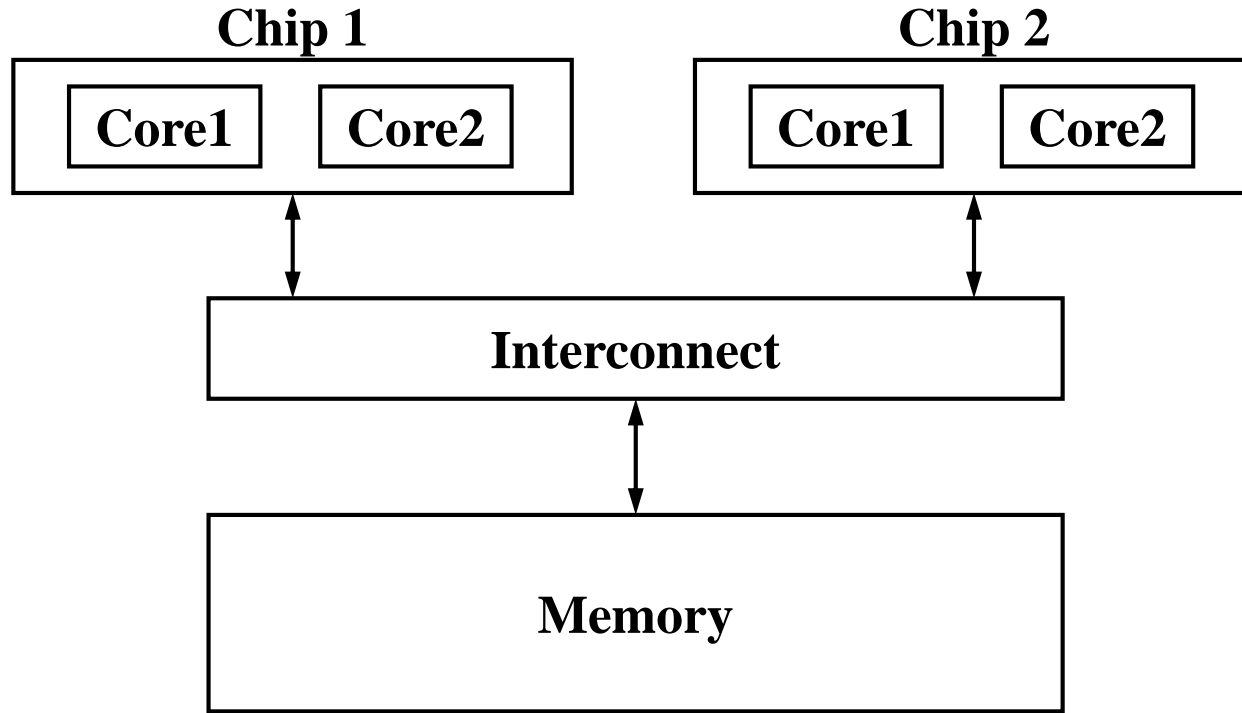
- **Most widely available shared memory systems use one or more multicore processors.**
 - (multiple CPU's or cores on a single chip)



Shared Memory System (3)



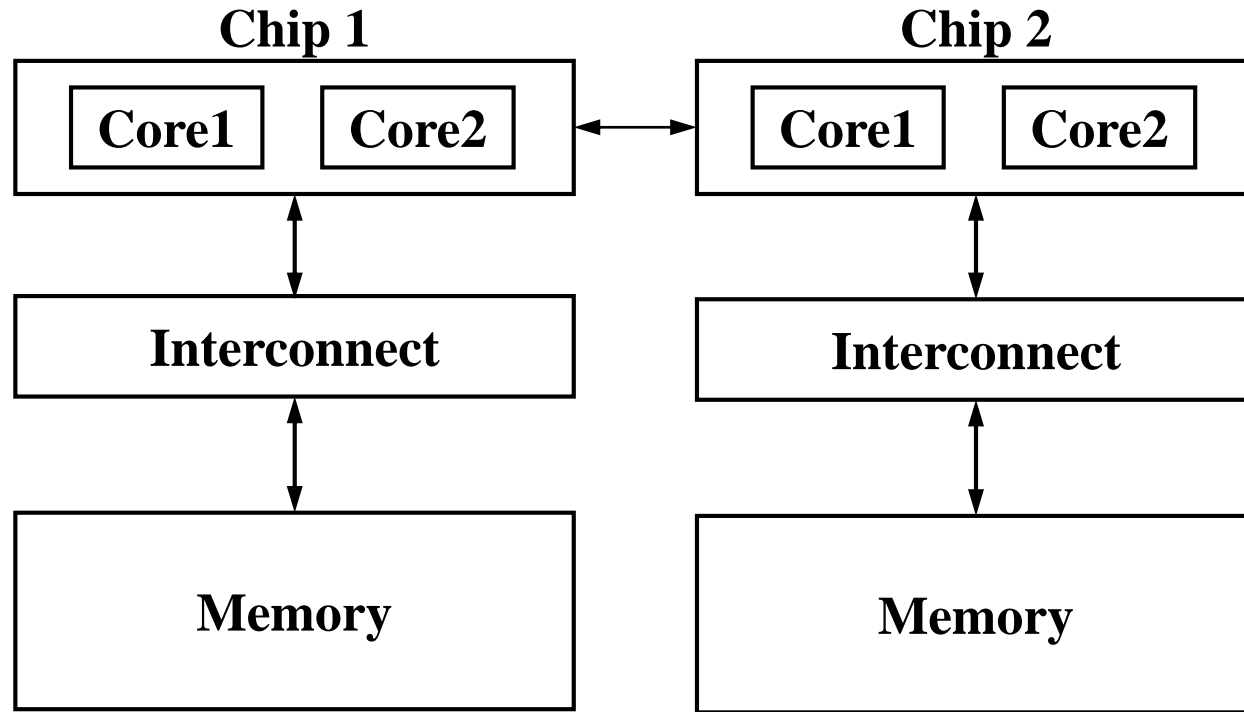
UMA multicore system



Time to access all the memory locations will be the same for all the cores.



NUMA multicore system



A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

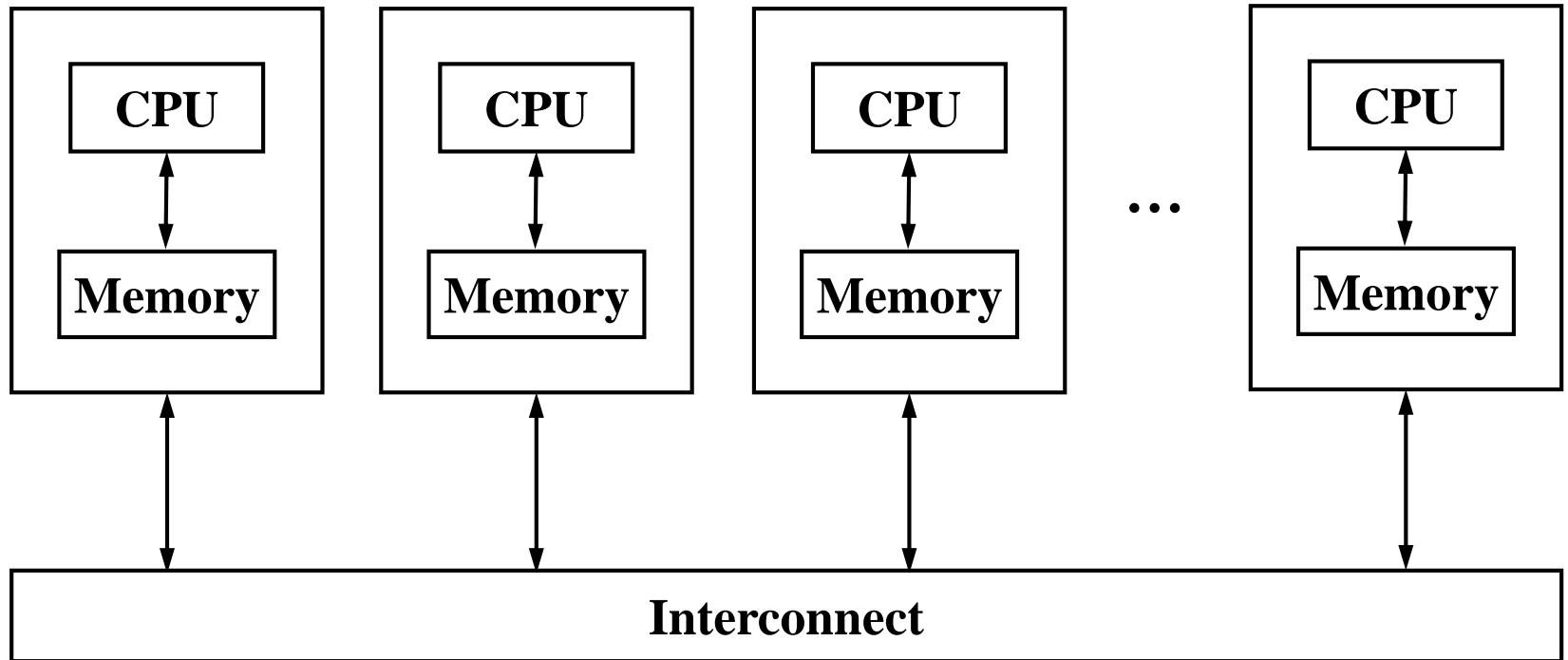


Distributed Memory System

- **Clusters** (most popular)
 - A collection of commodity systems.
 - Connected by a commodity interconnection network.
- **Nodes** of a cluster are individual computations units joined by a communication network.



Distributed Memory System



Interconnection networks

- **Affects performance of both distributed and shared memory systems.**
- **Two categories:**
 - **Shared memory interconnects**
 - **Distributed memory interconnects**



Shared memory interconnects

■ **Bus** interconnect

- A collection of parallel communication wires together with some hardware that controls access to the bus.
- Communication wires are shared by the devices that are connected to it.
- As the number of devices connected to the bus increases, contention for use of the bus increases, and performance decreases.

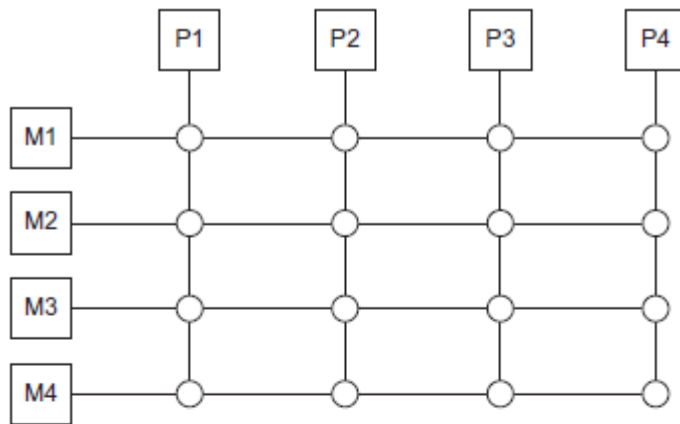


Shared memory interconnects

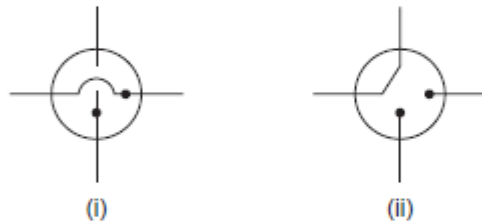
■ Switched interconnect

- Uses switches to control the routing of data among the connected devices.
- **Crossbar** –
 - Allows simultaneous communication among different devices.
 - Faster than buses.
 - But the cost of the switches and links is relatively high.

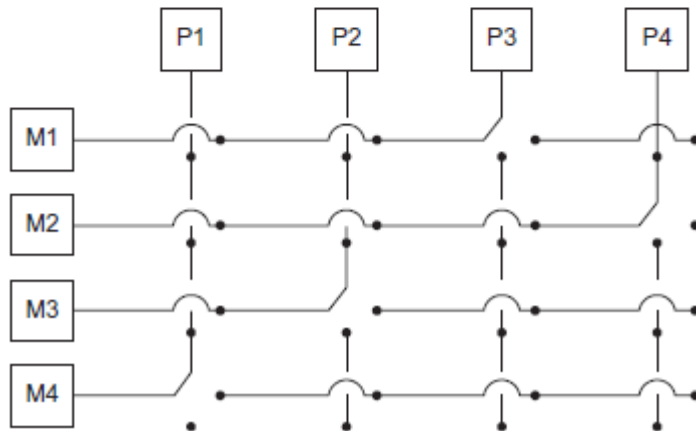




(a)



(b)



(c)

(a)

A crossbar switch connecting 4 processors (P_i) and 4 memory modules (M_j)

(b)

Configuration of internal switches in a crossbar

(c) Simultaneous memory accesses by the processors



北京航空航天大学
COLLEGE OF SOFTWARE
BEIHANG UNIVERSITY 软件学院

Distributed memory interconnects

- **Two groups**

- **Direct interconnect**

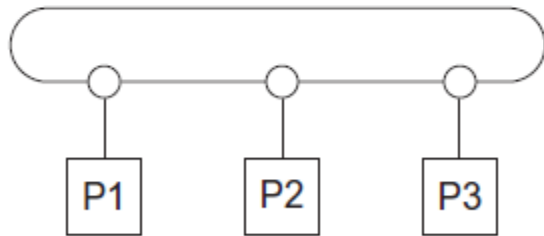
Each switch is directly connected to a processor memory pair, and the switches are connected to each other.

- **Indirect interconnect**

Switches may not be directly connected to a processor.

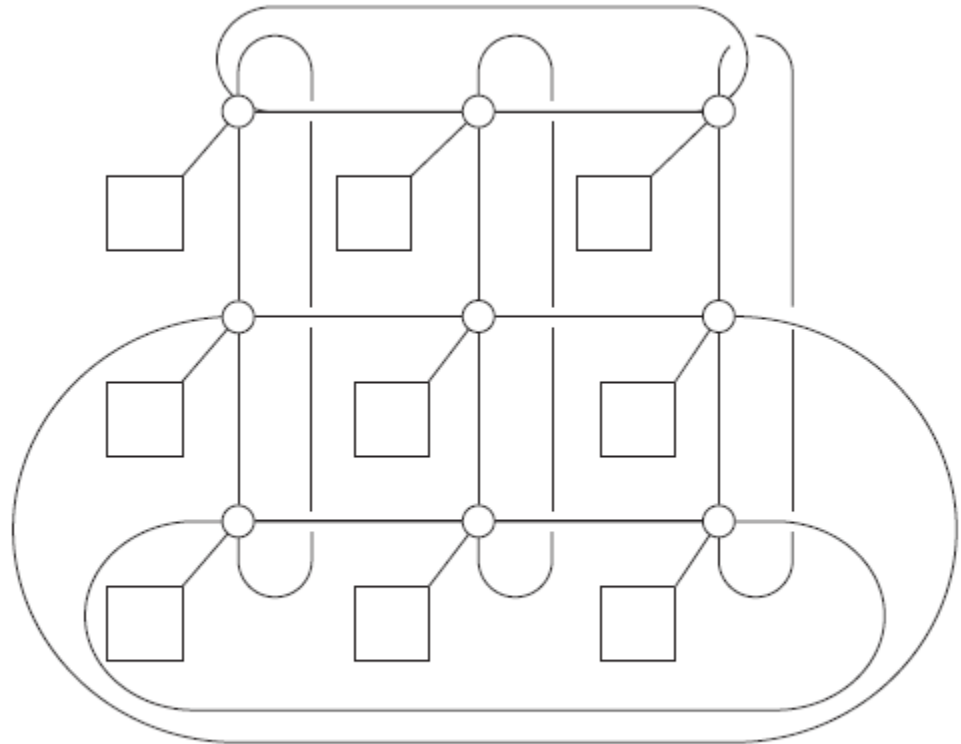


Direct interconnect



(a)

ring



(b)

toroidal mesh

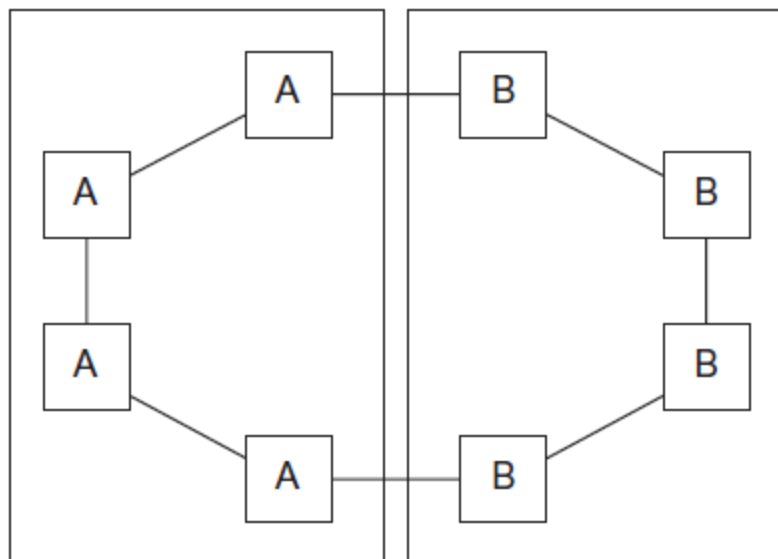


Bisection width

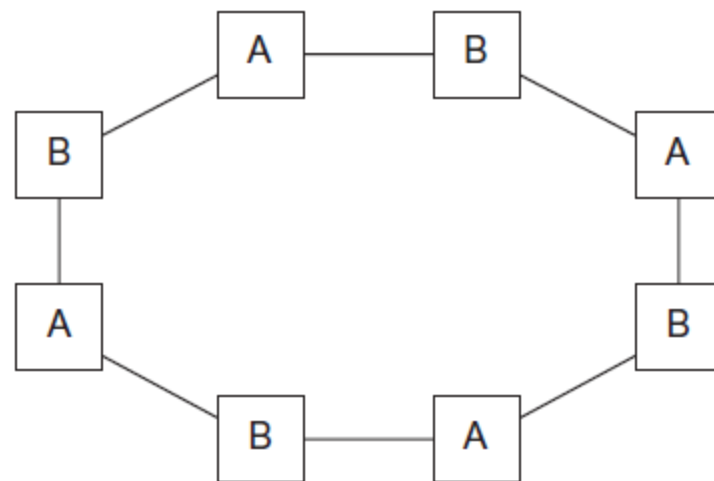
- A measure of “number of simultaneous communications” or “connectivity”.
对分网络各半所必须移去的最少边数
(对剖宽度或等分宽度)
- How many simultaneous communications can take place “across the divide” between the halves?



Two bisections of a ring



(a)

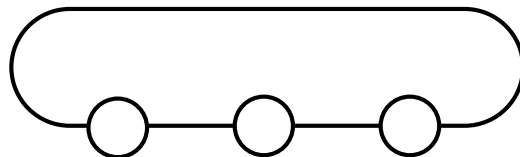
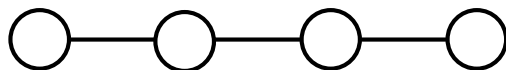


(b)



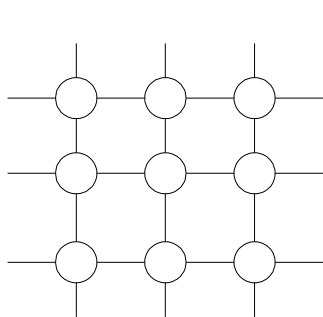
静态互连网络（1）

- 一维线性阵列（1-D Linear Array）：
 - 并行机中最简单、最基本的互连方式
 - 每个节点只与其左、右近邻相连，也叫二近邻连接
 - N 个节点用 $N-1$ 条边串接之，内节点度为 2，直径为 $N-1$ ，对剖宽度为 1
 - 当首、尾节点相连时可构成循环移位器，在拓扑结构上等同于环。环可以是单向的或双向的，节点度恒为 2，直径或为 $\lfloor N/2 \rfloor$ （双向环）或为 $N-1$ （单向环），对剖宽度为 2

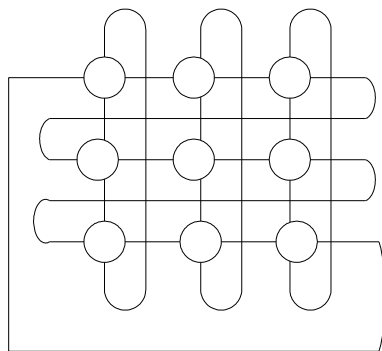


静态互连网络（2）

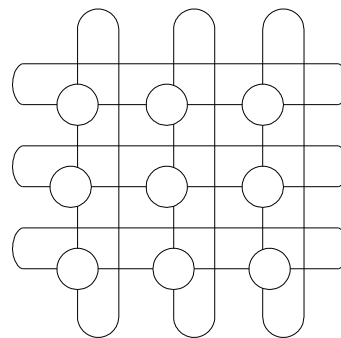
- $\sqrt{N} \times \sqrt{N}$ 二维网孔 (2-D Mesh) :
 - 每个节点只与其上、下、左、右的近邻相连（边界节点除外），节点度为 4，网络直径为 $2(\sqrt{N}-1)$ ，对剖宽度为 \sqrt{N}
 - 在垂直方向上带环绕，水平方向呈蛇状，就变成Illiac网孔了，节点度恒为 4，网络直径为 $\sqrt{N}-1$ ，而对剖宽度为 $2\sqrt{N}$
 - 垂直和水平方向均带环绕，则变成了2-D环绕 (2-D Torus)，节点度恒为 4，网络直径为 $2\lfloor\sqrt{N}/2\rfloor$ ，对剖宽度为 $2\sqrt{N}$



(a) 2-D网孔



(b) Illiac网孔



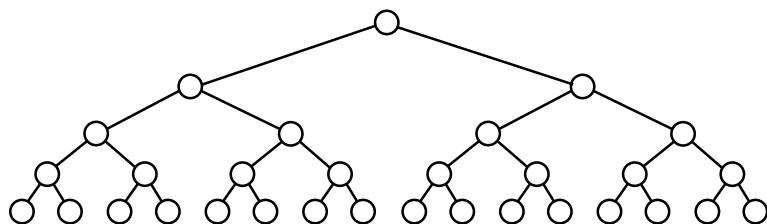
(c) 2-D环绕



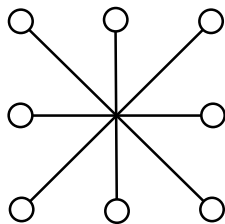
静态互连网络 (3)

■ 二叉树:

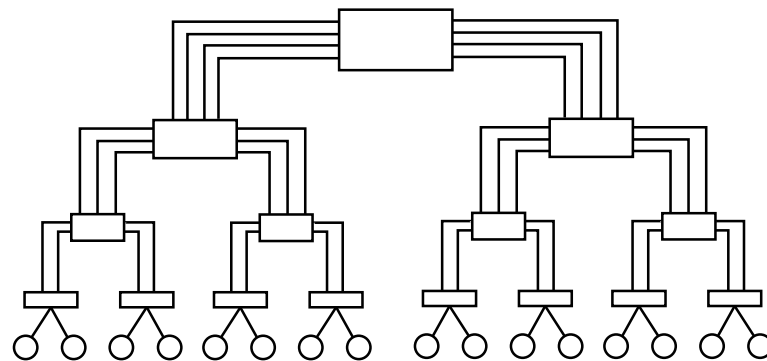
- 除了根、叶节点, 每个内节点只与其父节点和两个子节点相连。
- 节点度为3, 对剖宽度为1, 而树的直径为 $2(\lceil \log N \rceil - 1)$
- 如果尽量增大节点度, 则直径缩小为2, 此时就变成了星形网络, 其对剖宽度为 $\lfloor N/2 \rfloor$
- 传统二叉树的主要问题是根易成为通信瓶颈。胖树节点间的通路自叶向根逐渐变宽。



(a) 二叉树



(b) 星形连接



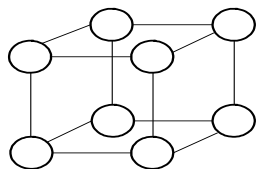
(c) 二叉胖树



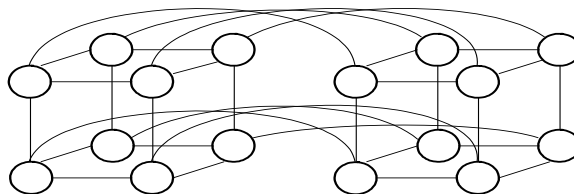
静态互连网络（4）

■ 超立方：

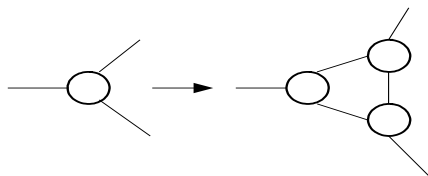
- 一个 n -立方由 $N = 2^n$ 个顶点组成，3-立方如图(a)所示，4-立方如图(b)所示，由两个3-立方的对应顶点连接而成。
- n -立方的节点度为 n ，网络直径也是 n ，而对剖宽度为 $N/2$ 。
- 如果将3-立方的每个顶点代之以一个环就构成了如图(d)所示的3-立方环，此时每个顶点的度为3，而不像超立方那样节点度为 n 。



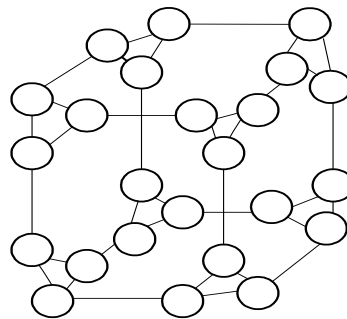
(a) 3-立方



(b) 4-立方

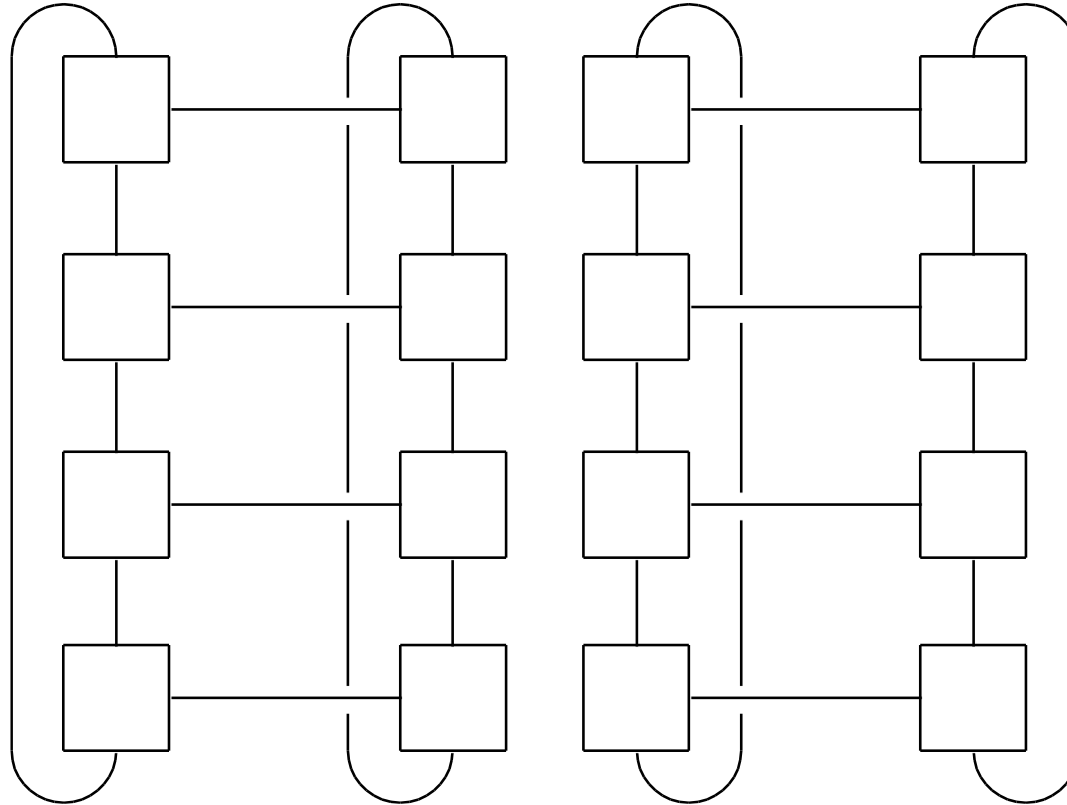


(c) 顶点代之以环



(d) 3-立方环

A bisection of a toroidal mesh (环面网格)



Definitions

■ Bandwidth

- The rate at which a link can transmit data.
- Usually given in megabits or megabytes per second.

■ Bisection bandwidth

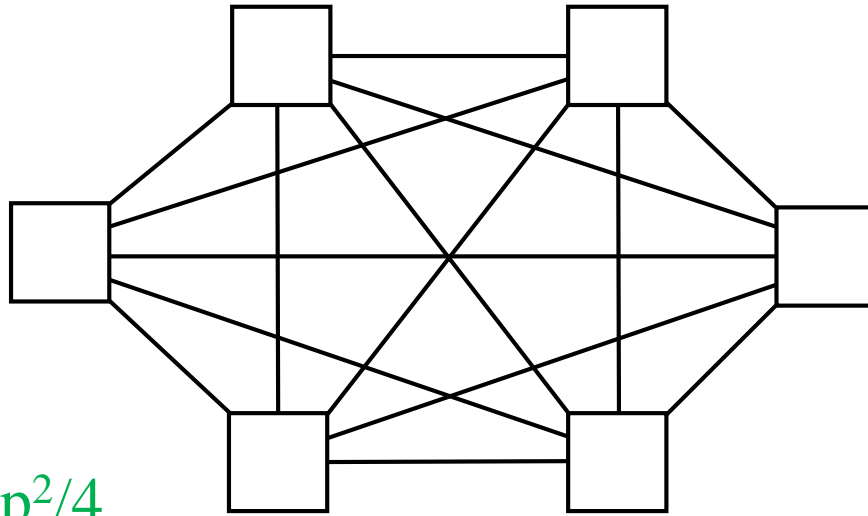
- A measure of network quality.
- Instead of counting the number of links joining the halves, it sums the bandwidth of the links.



Fully connected network

- Each switch is directly connected to every other switch.

impractical



bisection width = $p^2/4$

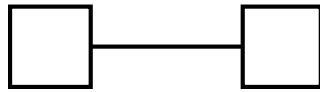


Hypercube

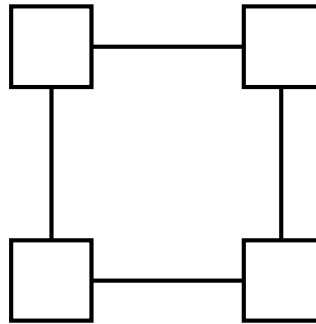
- Highly connected direct interconnect.
- Built inductively:
 - A **one-dimensional hypercube** is a fully-connected system with two processors.
 - A **two-dimensional hypercube** is built from two one-dimensional hypercubes by joining “corresponding” switches.
 - Similarly a **three-dimensional hypercube** is built from two two-dimensional hypercubes.



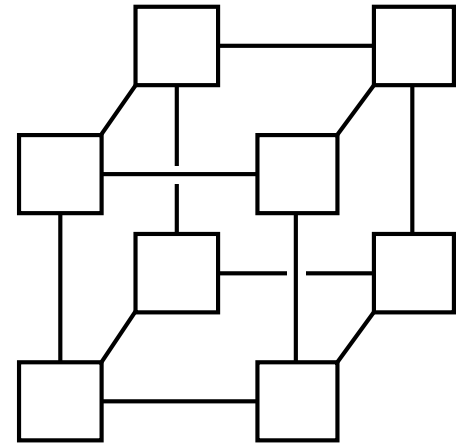
Hypercubes



(a)
one-



(b)
two-



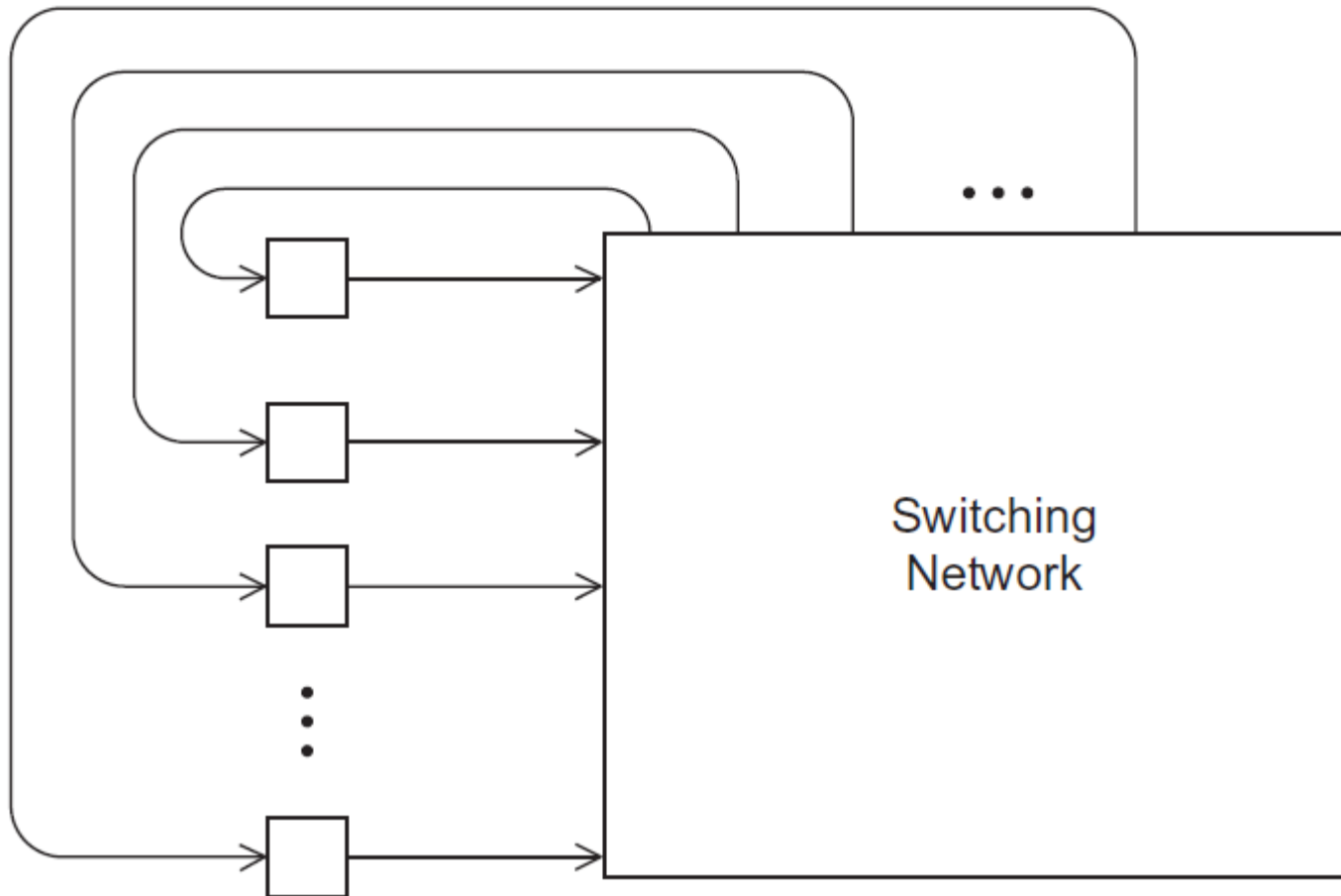
(c)
three-dimensional

Indirect interconnects

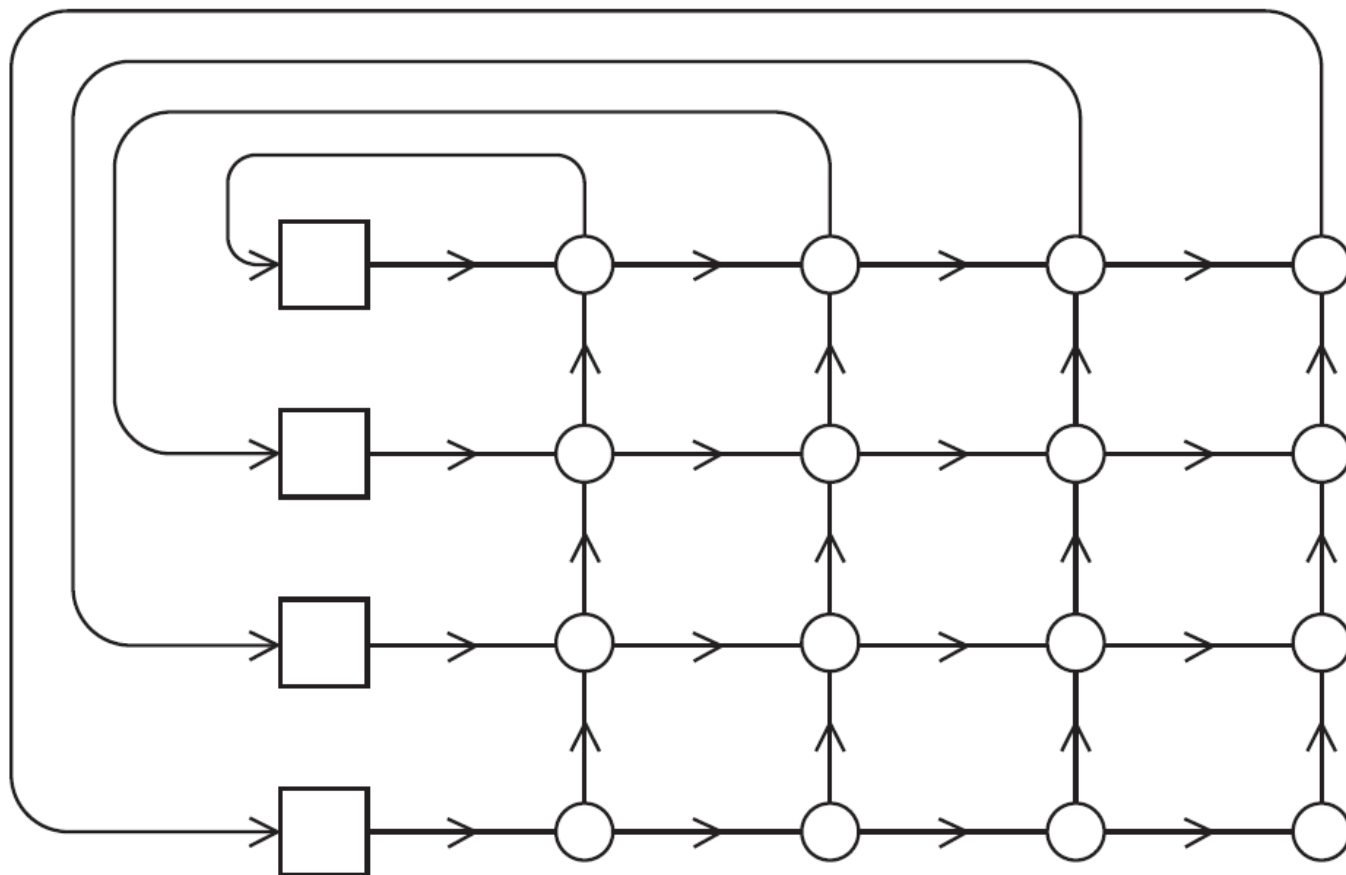
- **Simple examples of indirect networks:**
 - **Crossbar**
 - **Omega network**
- **Often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network.**



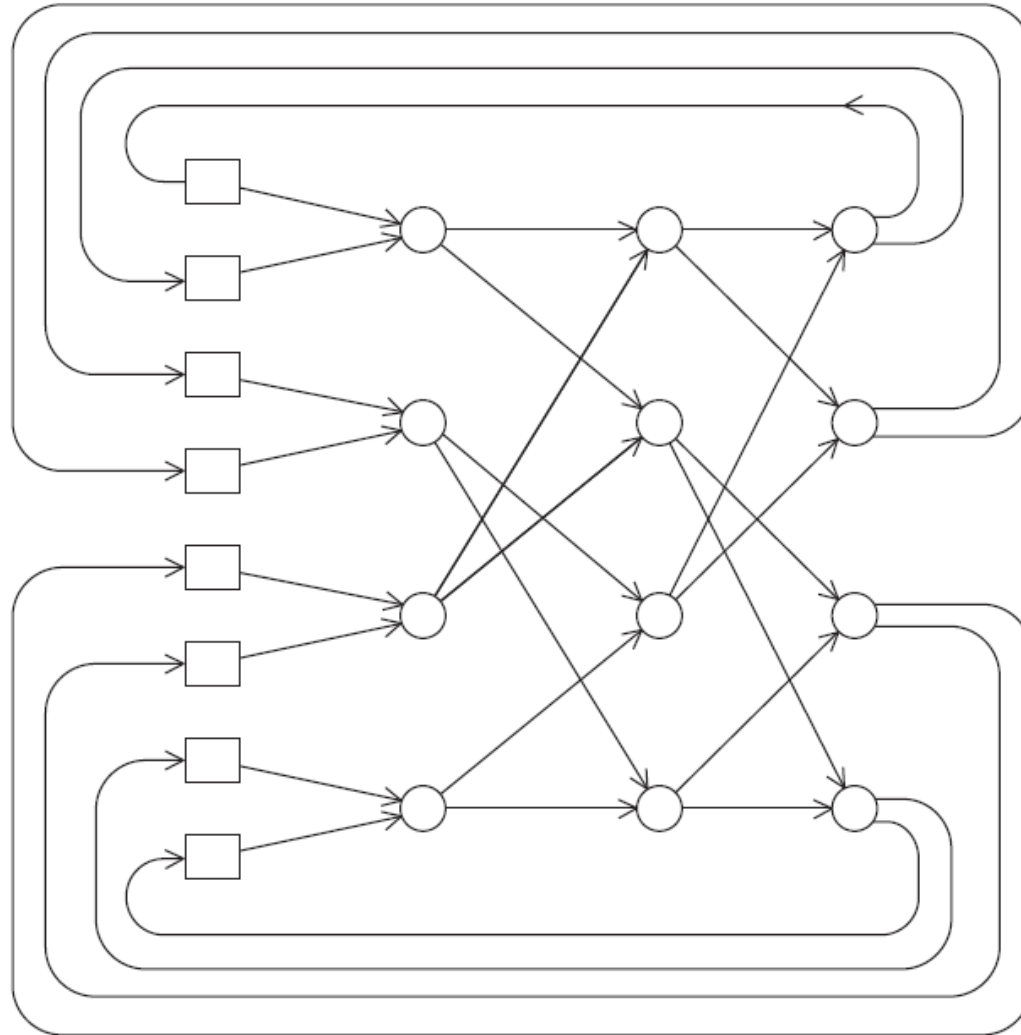
A generic indirect network



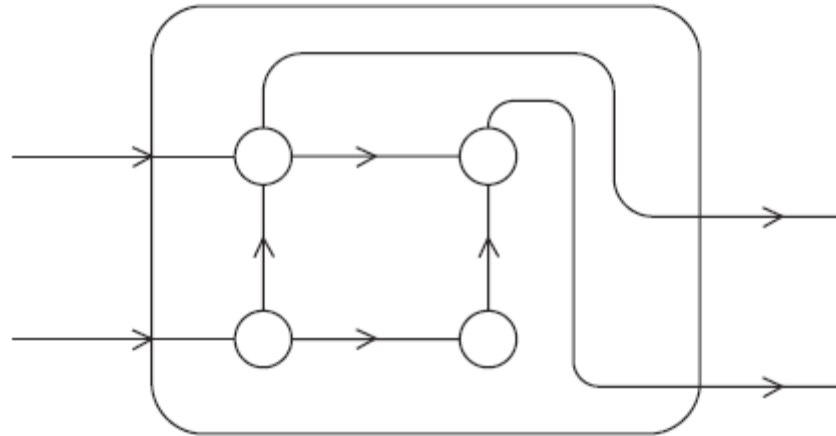
Crossbar interconnect for distributed memory



An omega network



A switch in an omega network



More definitions

- Any time data is transmitted, we're interested in how long it will take for the data to reach its destination.
- **Latency**
 - The time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte.
- **Bandwidth**
 - The rate at which the destination receives data after it has started to receive the first byte.



$$\text{Message transmission time} = l + n / b$$

latency (seconds)

length of message (bytes)

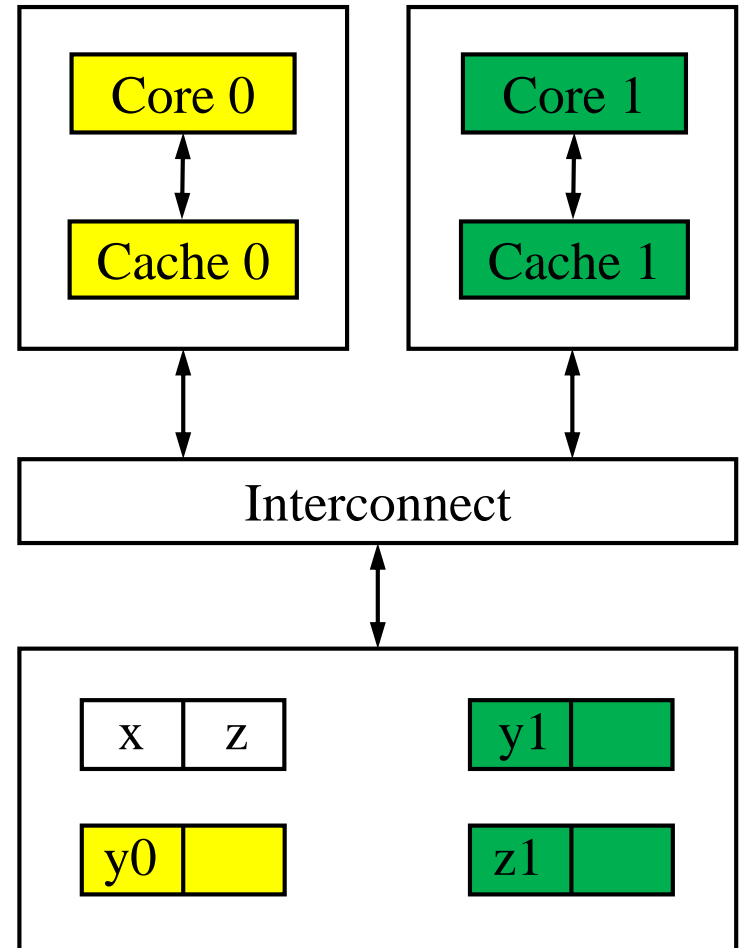
bandwidth (bytes per second)



Cache coherence

- **Programmers have no control over caches and when they get updated.**

A shared memory system with two cores and two caches

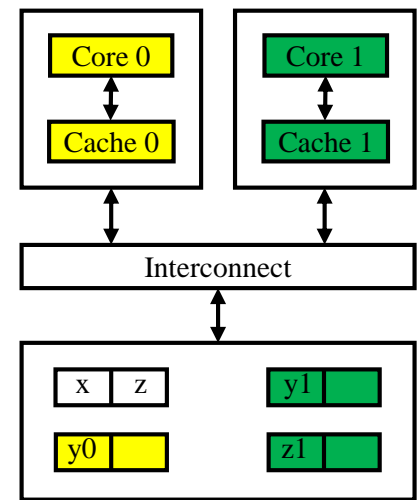


Cache coherence

y0 privately owned by Core 0

y1 and z1 privately owned by Core 1

x = 2; /* shared variable */



Time	Core 0	Core 1
0	y0 = x	y1 = 3 * x
1	x = 7	Statement(s) not involving x
2	Statement(s) not involving x	z1 = 4 * x

y0 eventually ends up = 2

y1 eventually ends up = 6

z1 = ???



Snooping Cache Coherence

- The cores share a bus .
- Any signal transmitted on the bus can be “seen” by all cores connected to the bus.
- When core 0 updates the copy of x stored in its cache it also broadcasts this information across the bus.
- If core 1 is “snooping” the bus, it will see that x has been updated and it can mark its copy of x as invalid.



Directory Based Cache Coherence

- Uses a data structure called a **directory** that stores the status of each cache line.
- When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.





4. PARALLEL SOFTWARE



The burden is on software

- **Hardware and compilers can keep up the pace needed.**
- **From now on...**
 - **In shared memory programs:**
 - **Start a single process and fork threads.**
 - **Threads carry out tasks.**
 - **In distributed memory programs:**
 - **Start multiple processes.**
 - **Processes carry out tasks.**



SPMD – single program multiple data

- A SPMD programs consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches.

```
if (I'm thread/process 0)
    do this;
else
    do that;
```

task-parallelism



data-parallelism



```
if (I'm thread/process 0)
    operate on the first half of the array;
else /* I'm thread/process 1*/
    operate on the second half of the array;
```



Writing Parallel Programs

1. Divide the work among the processes/threads
 - (a) so each process/thread gets roughly the same amount of work
 - (b) and communication is minimized.
2. Arrange for the processes/threads to synchronize.
3. Arrange for communication among processes/threads.

```
double x[n], y[n];  
...  
for (i = 0; i < n; i++)  
    x[i] += y[i];
```

$[0 \sim (n/p)-1]$ $[(n/p) \sim 2(n/p)-1]$... $[(p-1)(n/p) \sim n-1]$



Shared Memory

■ Dynamic threads

- Master thread waits for work, forks new threads, and when threads are done, they terminate.
- Efficient use of resources, but thread creation and termination is time consuming.

■ Static threads

- Pool of threads created and are allocated work, but do not terminate until cleanup.
- Better performance, but potential waste of system resources.

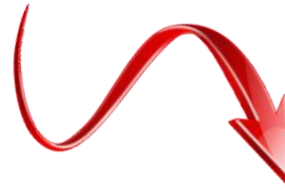


Nondeterminism

```
. . .  
printf("Thread %d > my_val = %d\n", my_rank, my_x) ;  
. . .
```



```
Thread 1 > my_val = 19  
Thread 0 > my_val = 7
```



```
Thread 0 > my_val = 7  
Thread 1 > my_val = 19
```



Nondeterminism

```
my_val = Compute_val(my_rank) ;  
x += my_val;
```

Time	Core 0	Core 1
0	Finish assignment to my_val	In call to Compute_val
1	Load x = 0 into register	Finish assignment to my_val
2	Load my_val = 7 into register	Load x = 0 into register
3	Add my_val = 7 to x	Load my_val = 19 into register
4	Store x = 7	Add my_val to x
5	Start other work	Store x = 19



Nondeterminism

- Race condition
- Critical section
- Mutually exclusive
- Mutual exclusion lock (mutex, or simply lock)

```
my_val = Compute_val(my_rank);
```

```
Lock(&add_my_val_lock);
```

```
    x += my_val;
```

```
Unlock(&add_my_val_lock);
```

Mutual exclusion lock



busy-waiting

```
ok_for_1 = false;
...
my_val = Compute_val(my_rank);

if (my_rank == 1)
    while (!ok_for_1);    /* Busy-wait loop */
x += my_val;              /* Critical section */

if (my_rank == 0)
    ok_for_1 = true;      /* Let thread 1 update x */
```

Busy waiting



Distributed Memory

```
char message [100];
```

```
. . .
```

```
my_rank = Get_rank();
```

Message passing

```
if (my_rank == 1) {
```

```
    sprintf(message, "Greetings from process 1");
```

```
    Send(message, MSG_CHAR, 100, 0);
```

```
} else if (my_rank == 0) {
```

```
    Receive(message, MSG_CHAR, 100, 1);
```

```
    printf("Process 0 > Received:%s\n", message);
```

```
}
```



Partitioned Global Address Space Languages

```
shared int n =...;

shared double x[n], y[n];

private int i, my_first_element, my_last_element;

my_first_element =...;

my_last_element =...;

/* Initialize x and y */
...

for (i = my_first_element; i <= my_last_element; i++)
    x[i] += y[i];
```



Input and Output

- In distributed memory programs, only process 0 will access *stdin*. In shared memory programs, only the master thread or thread 0 will access *stdin*.
- In both distributed memory and shared memory programs all the processes/threads can access *stdout* and *stderr*.



Input and Output

- However, because of the indeterminacy of the order of output to *stdout*, in most cases only a single process/thread will be used for all output to *stdout* other than debugging output.
- Debug output should always include the rank or id of the process/thread that's generating the output.



Input and Output files

- Only a single process/thread will attempt to access any single file other than *stdin*, *stdout*, or *stderr*. So, for example, each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.





5. PERFORMANCE



Speedup

- Number of cores = p
- Serial run-time = T_{serial}
- Parallel run-time = T_{parallel}



linear speedup

$$T_{\text{parallel}} = T_{\text{serial}} / p$$



Speedup of a parallel program

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$



Efficiency of a parallel program

$$E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}} \right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}$$



Speedups and efficiencies of a parallel program

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
$E = S / p$	1.0	0.95	0.90	0.81	0.68

一个和尚挑水吃，两个和尚抬水吃，三个和尚没水吃。

——通信的开销增大了！

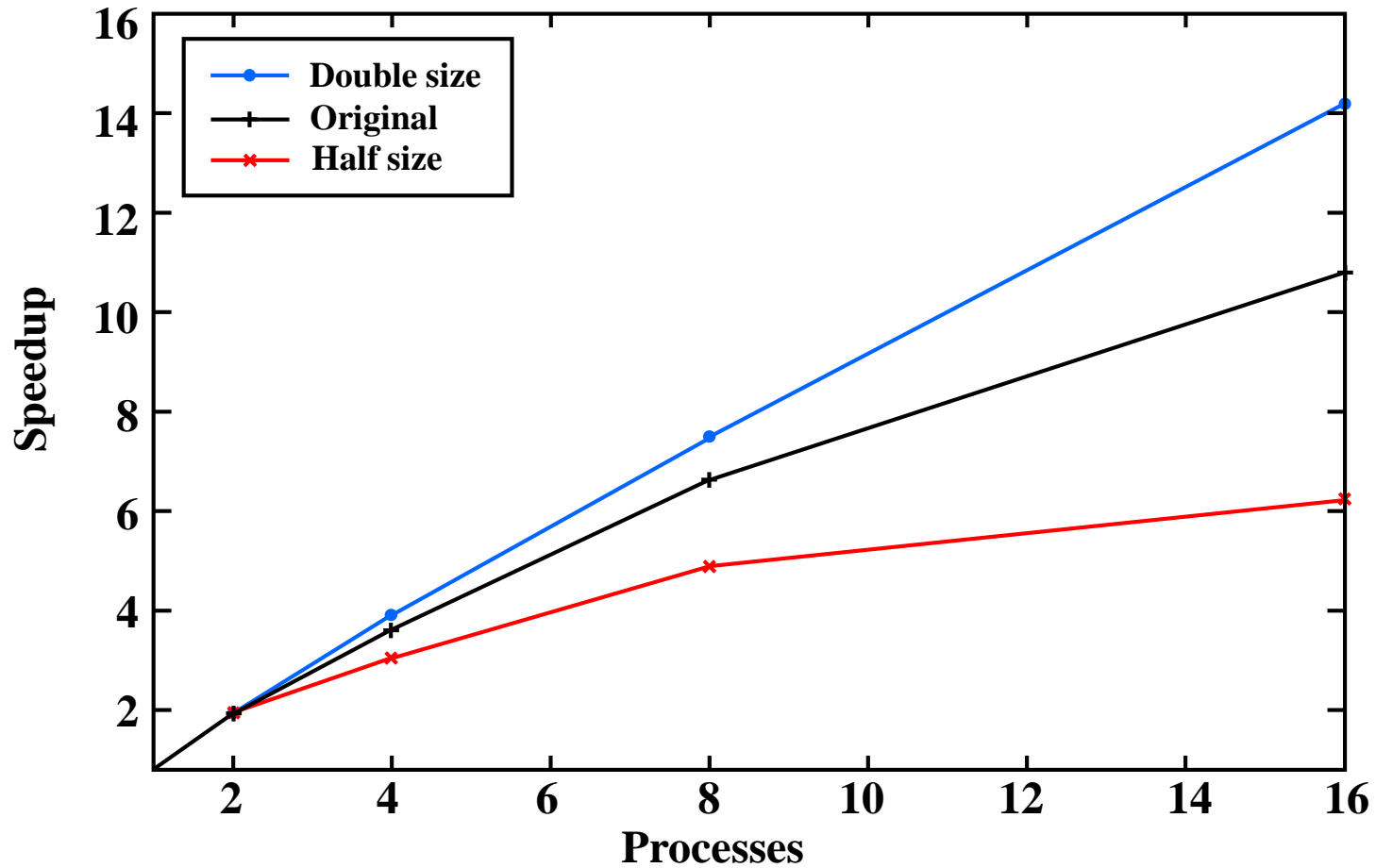


Speedups and efficiencies of parallel program on different problem sizes

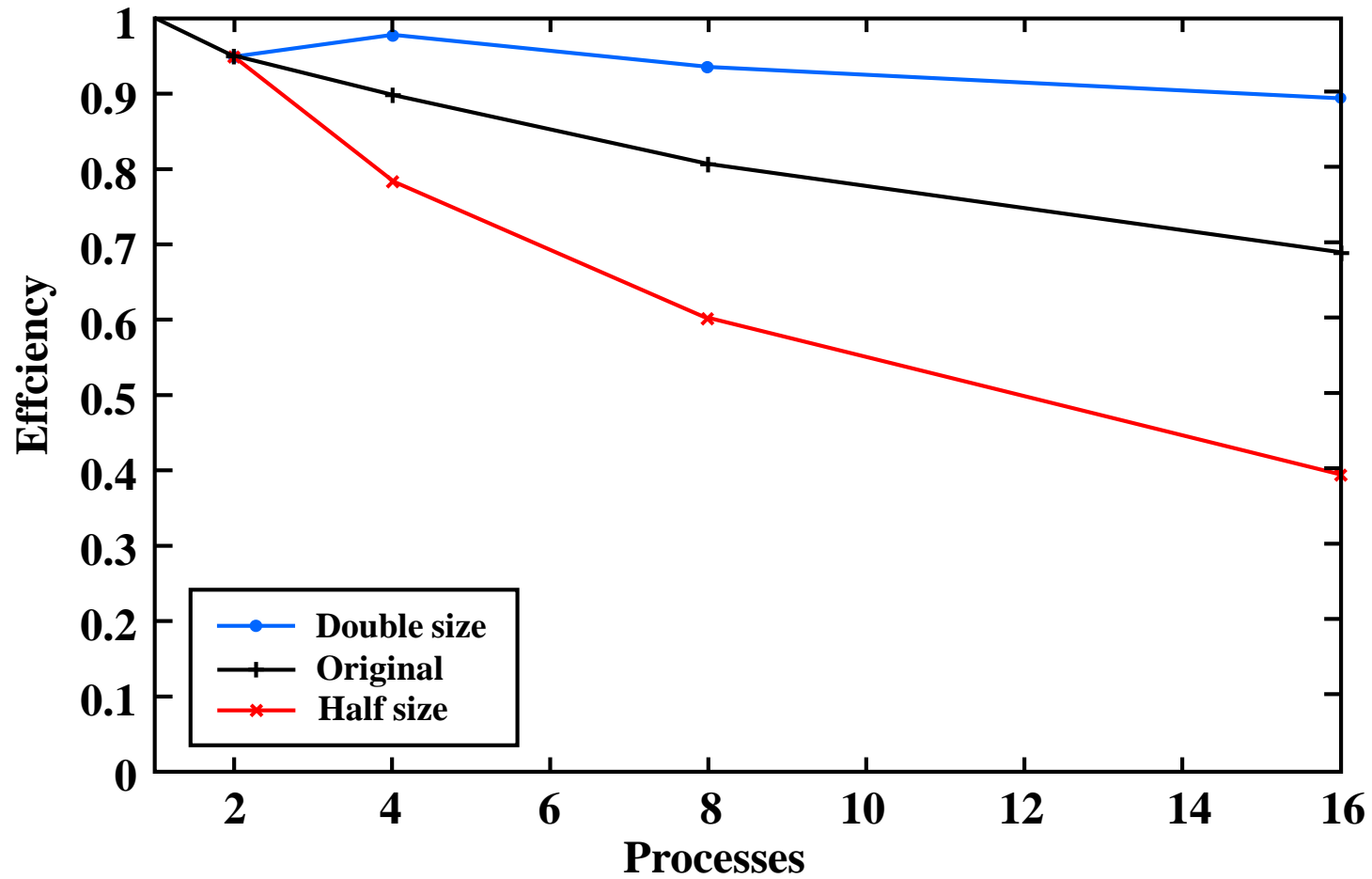
	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89



Speedup



Efficiency



Effect of overhead

$$T_{\text{parallel}} = T_{\text{serial}} / p + T_{\text{overhead}}$$

并行工作时间 = 串行工作时间 / 进程（或线程）个数
+ 通信开销时间



Amdahl's Law (1967)

- Unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited — regardless of the number of cores available.



Example

- We can parallelize 90% of a serial program.
- Parallelization is “perfect” regardless of the number of cores p we use.
- $T_{\text{serial}} = 20$ seconds
- Runtime of parallelizable part is $0.9 \times T_{\text{serial}} / p = 18 / p$
- Runtime of “unparallelizable” part is $0.1 \times T_{\text{serial}} = 2$
- Overall parallel run-time is

$$T_{\text{parallel}} = 0.9 \times T_{\text{serial}} / p + 0.1 \times T_{\text{serial}} = 18 / p + 2$$



Example (cont.)

■ Speedup

$$S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}} / p + 0.1 \times T_{\text{serial}}} = \frac{20}{18 / p + 2}$$

$$P \rightarrow \infty$$

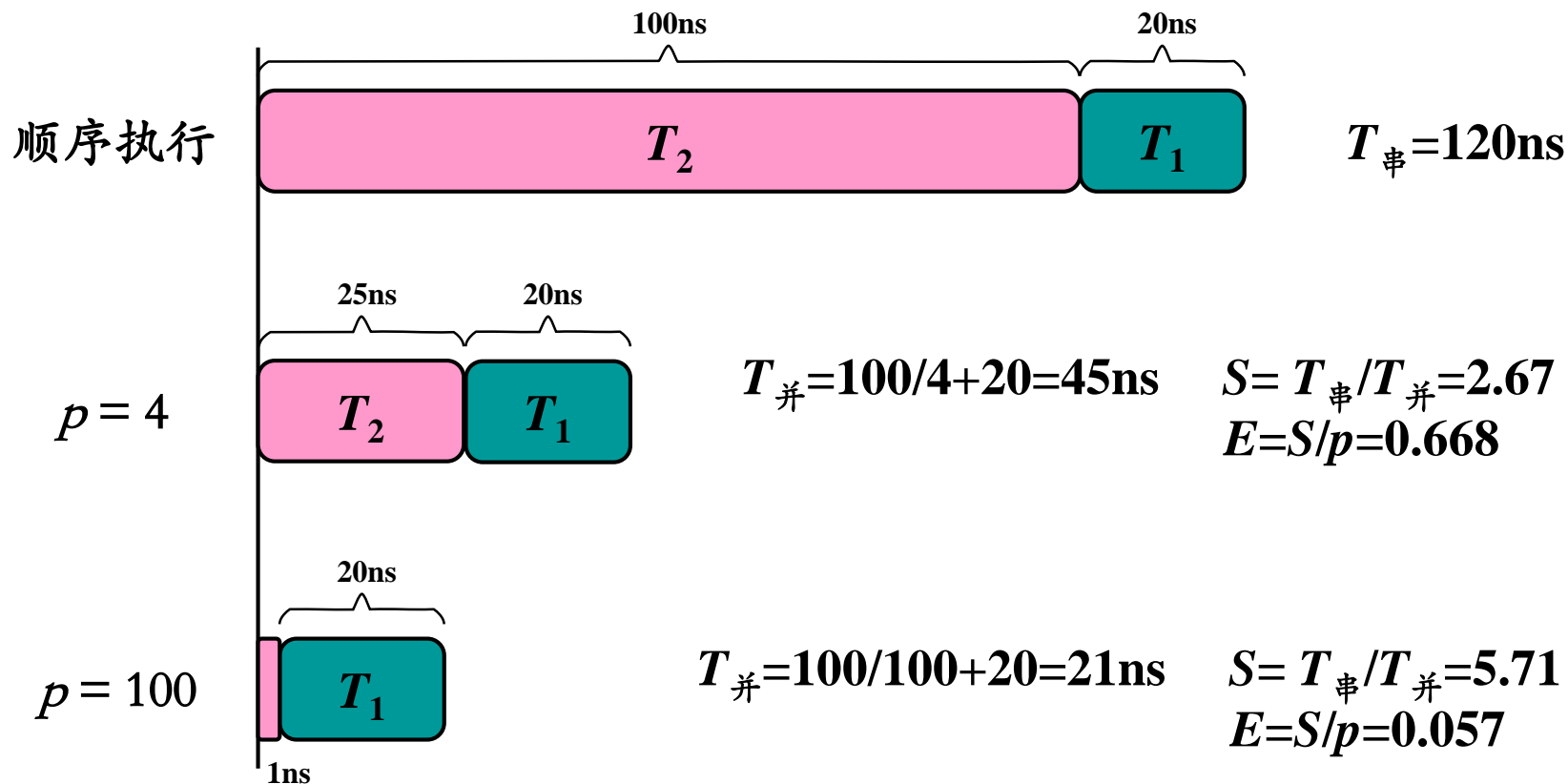
$$S \rightarrow 10$$



另一个例子:

T_1 =不受影响的执行时间



T_2 =受优化影响的执行时间



按照阿姆达尔定律, 极限加速比为6 ($S=1/\text{串行部分占比}$)



Scalability

- In general, a problem is *scalable* if it can handle ever increasing problem sizes.
- If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is *strongly scalable*. 
- If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, the problem is *weakly scalable*. 



Taking Timings

- What is time?
- Start to finish?
- A program segment of interest?
- CPU time?
- Wall clock time?



Taking Timings

```
double start, finish;  
...  
start = Get_current_time();  
/* Code that we want to time */  
...  
finish = Get_current_time();  
printf("The elapsed time = %e second\n", finish-start);
```

theoretical
function

MPI_Wtime()

omp_get_wtime()



Taking Timings

```
private double start, finish;  
...  
start = Get_current_time();  
/* Code that we want to time */  
...  
finish = Get_current_time();  
printf("The elapsed time = %e second\n", finish-start);
```



Taking Timings

```
shared double global_elapseded;  
private double my_start, my_finish, my_elapseded;  
...  
/* Synchronize all processes/threads */  
Barrier();  
my_start = Get_current_time();  
  
/* Code that we want to time */  
...  
  
my_finish = Get_current_time();  
my_elapseded = my_finish - my_start;  
  
/* Find the max across all processes/threads */  
global_elapseded = Global_max(my_elapseded);  
if (my_rank == 0)  
    printf("The elapsed time = %e second\n", global_elapseded );
```





6. PARALLEL PROGRAM DESIGN



北京航空航天大学
COLLEGE OF SOFTWARE
BEIHANG UNIVERSITY 软件学院

Foster's methodology (PCAM)

- ① **Partitioning**: divide the computation to be performed and the data operated on by the computation into small tasks.

The focus here should be on identifying tasks that can be executed in parallel.



Foster's methodology (PCAM)

- ② **Communication**: determine what communication needs to be carried out among the tasks identified in the previous step.



Foster's methodology (PCAM)

- ③ **Agglomeration or aggregation:** combine tasks and communications identified in the first step into larger tasks.

For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.



Foster's methodology

- ④ **Mapping**: assign the composite tasks identified in the previous step to processes/threads.

This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.



第八章 并行算法一般设计过程

➤ 8.1 PCAM设计方法学

8.2 划分

8.3 通信

8.4 组合

8.5 映射

8.6 小结



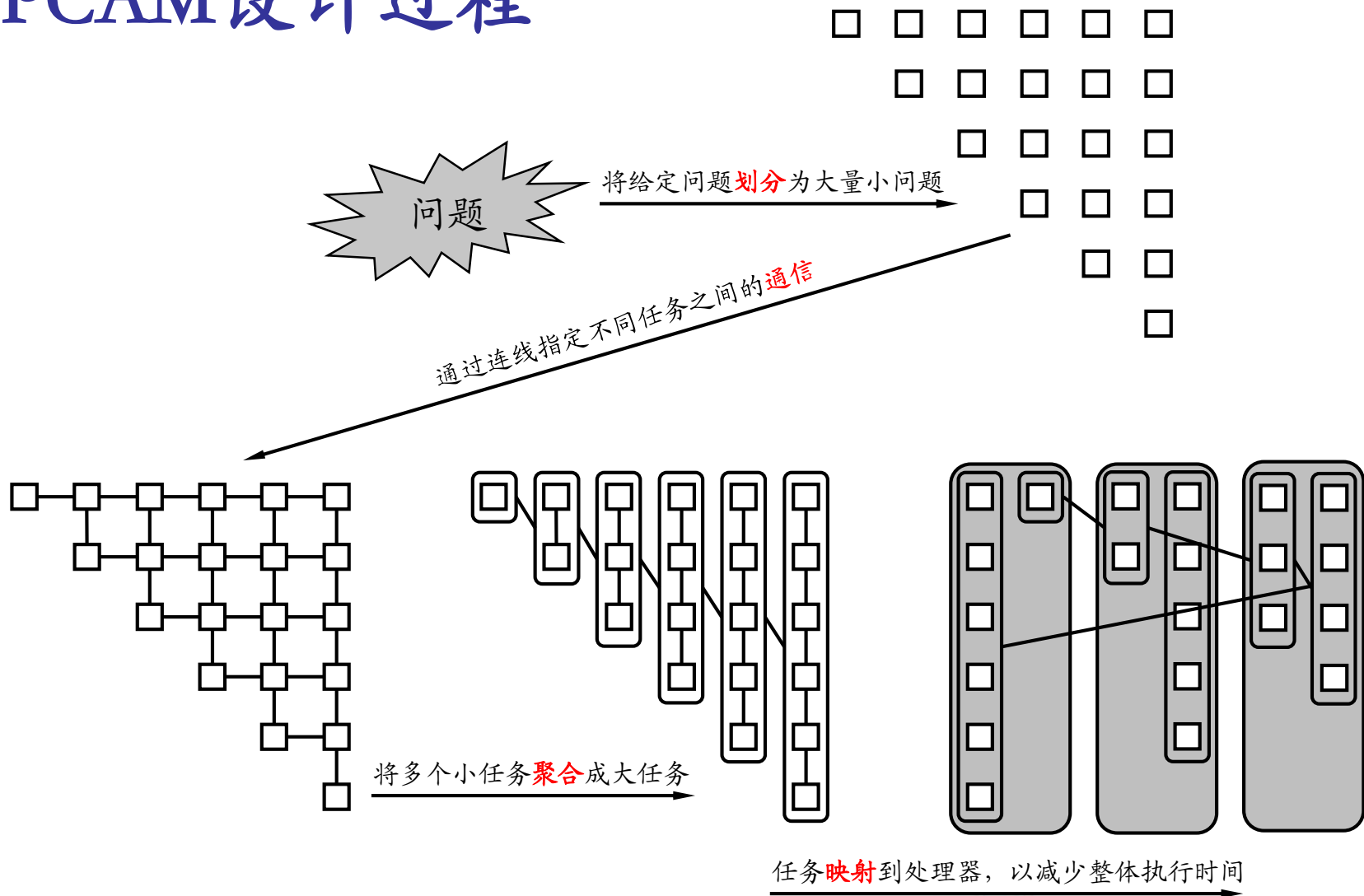
8.1 PCAM设计方法学

设计并行算法的四个阶段

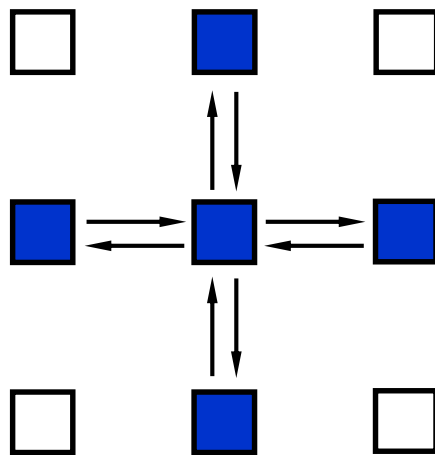
- 划分(Partitioning): 分解成小的任务, 开拓并行性;
- 通信(Communication): 确定诸任务间的数据交换, 监测划分的合理性;
- 组合(Agglomeration): 依据任务的局部性, 组合成更大的任务;
- 映射(Mapping): 将每个任务分配到处理器上, 提高算法的性能。



PCAM设计过程



例：二维数组的Jacobi迭代

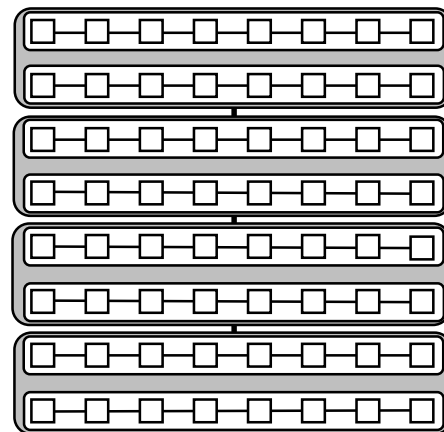
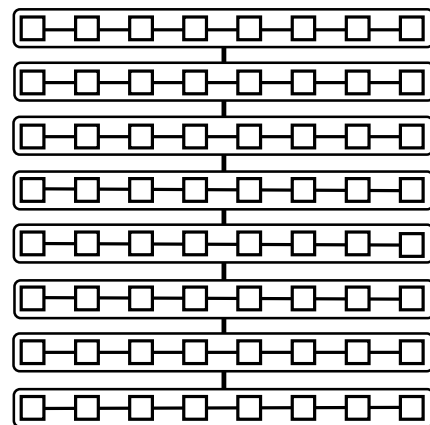
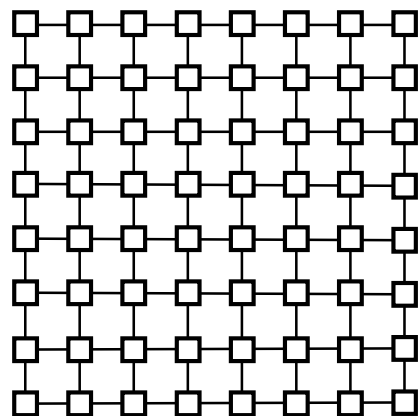


$$data_{t+1}(i,j) \leftarrow \frac{data_t(i-1,j) + data_t(i+1,j) + data_t(i,j-1) + data_t(i,j+1)}{4}$$

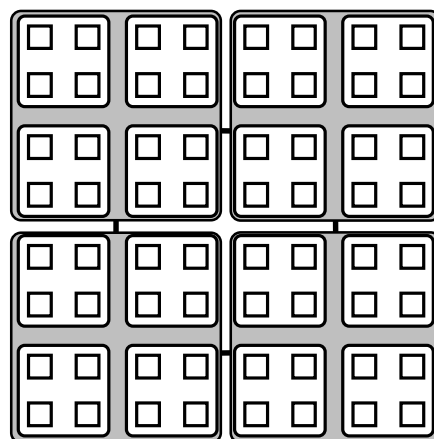
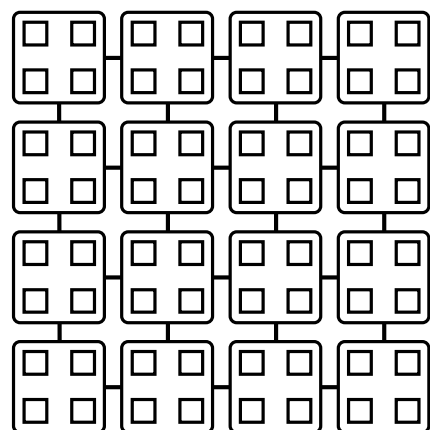


例：

$n \times n$ $n = 8$



$$T_{comm}(n) = 2(s + rn)$$



$$T_{comm}(n) = 4(s + r(n/\sqrt{p})) \quad p = 4$$

当 p 增大时，下面的方案比上面的好！



北京航空航天大学
COLLEGE OF SOFTWARE
BEIHANG UNIVERSITY 软件学院

第八章 并行算法一般设计过程

8.1 PCAM设计方法学

➤ 8.2 划分

➤ 8.2.1 方法描述

8.3 通信

8.2.2 域分解

8.4 组合

8.2.3 功能分解

8.5 映射

8.2.4 划分判据

8.6 小结



划分方法描述

- 充分开拓算法的并发性和可扩放性;
- 先进行数据分解(称域分解), 再进行计算功能的分解(称功能分解);
- 使数据集和计算集互不相交;
- 划分阶段忽略处理器数目和目标机器的体系结构;
- 能分为两类划分:
 - 域分解(domain decomposition)
 - 功能分解(functional decomposition)



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

➤ 8.2 划分

8.2.1 方法描述

8.3 通信

➤ 8.2.2 域分解

8.4 组合

8.2.3 功能分解

8.5 映射

8.2.4 划分判据

8.6 小结



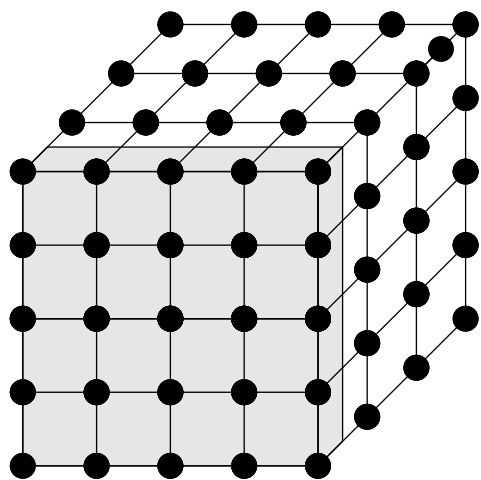
域分解

- 划分的对象是数据，可以是算法的输入数据、中间处理数据和输出数据；
- 将数据分解成大致相等的小数据片；
- 划分时考虑数据上的相应操作；
- 如果一个任务需要别的任务中的数据，则会产生任务间的通信；

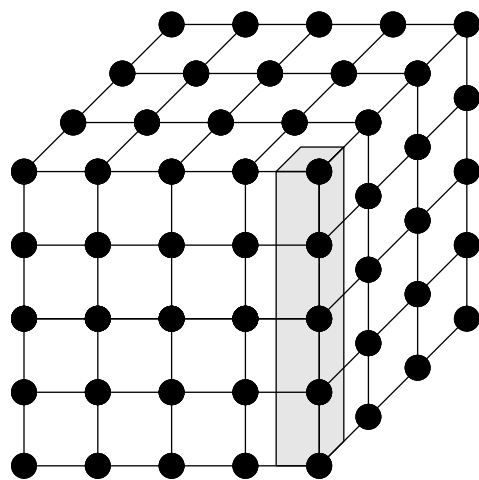


域分解

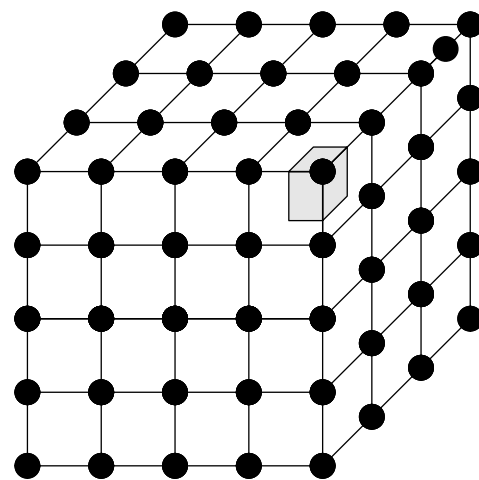
- 示例：三维网格的域分解，各格点上计算都是重复的。下图是三种分解方法：



1-D



2-D



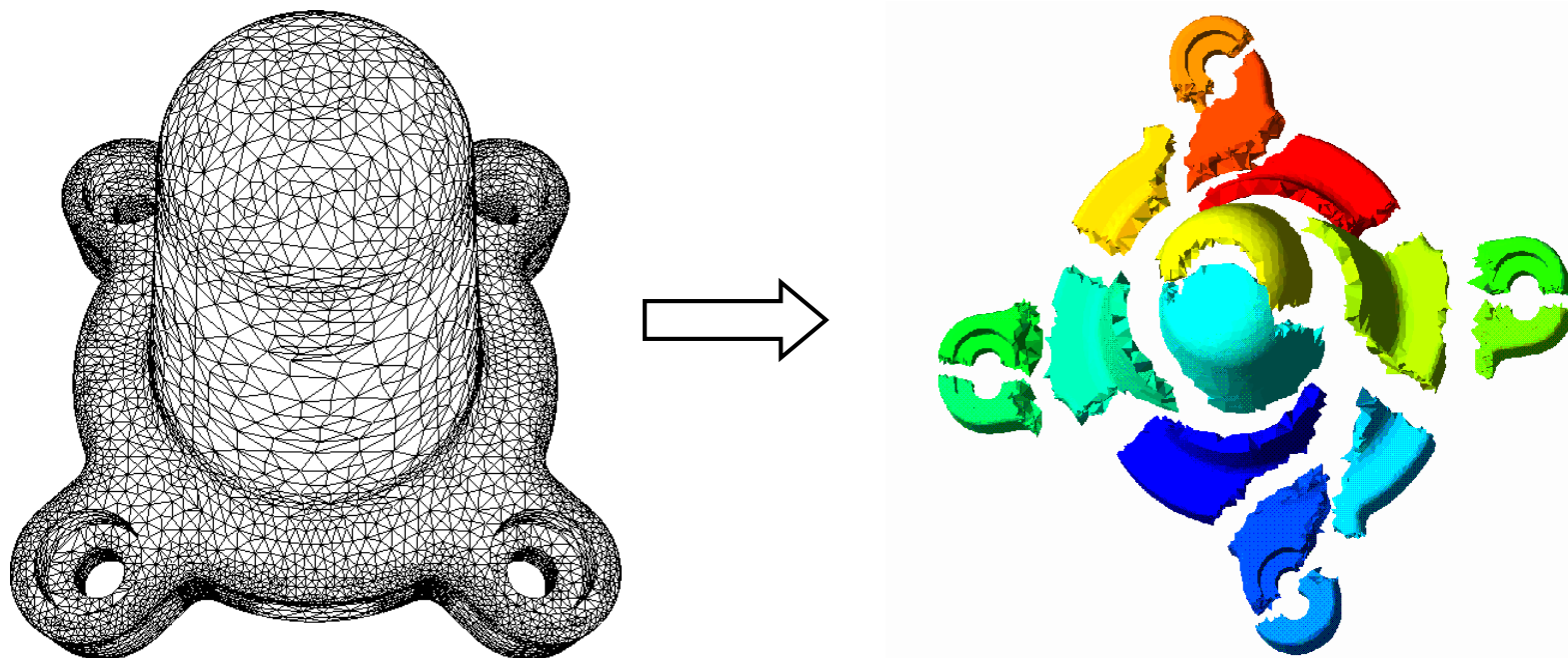
3-D

图7.2



域分解

- 不规则区域的分解示例：



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

➤ 8.2 划分

8.2.1 方法描述

8.3 通信

8.2.2 域分解

8.4 组合

➤ 8.2.3 功能分解

8.5 映射

8.2.4 划分判据

8.6 小结



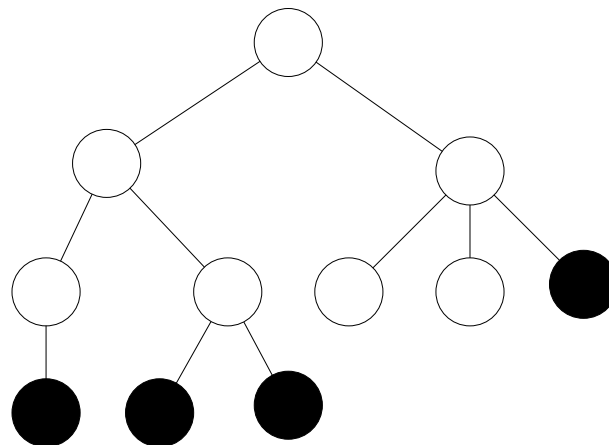
功能分解

- 划分的对象是计算，将计算划分为不同的任务，其出发点不同于域分解；
- 划分后，研究不同任务所需的数据。如果这些数据不相交的，则划分是成功的；如果数据有相当的重叠，意味着要重新进行域分解和功能分解；
- 功能分解是一种更深层次的分解。

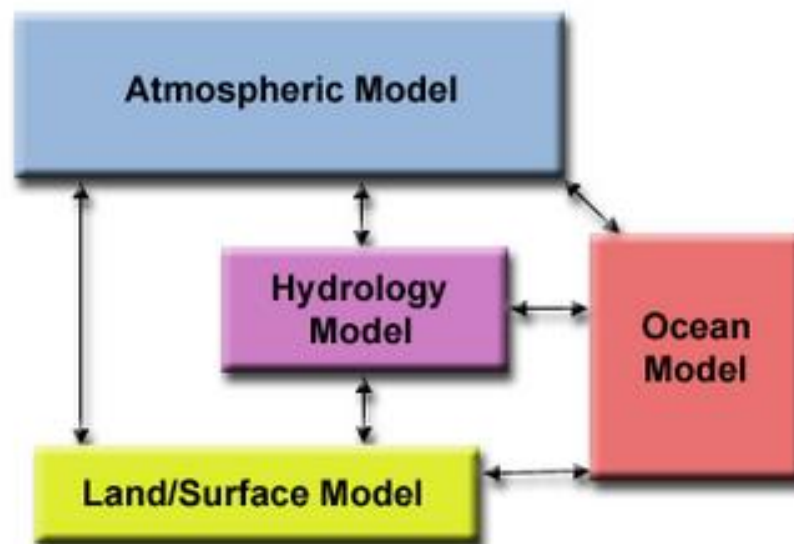


功能分解

■ 示例1：搜索树



■ 示例2：气候模型



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

➤ 8.2 划分

8.2.1 方法描述

8.3 通信

8.2.2 域分解

8.4 组合

8.2.3 功能分解

8.5 映射

➤ 8.2.4 划分判据

8.6 小结



划分判据

- 划分是否具有灵活性？
- 划分是否避免了冗余计算和存储？
- 划分任务尺寸是否大致相当？
- 任务数与问题尺寸是否成比例？
- 功能分解是一种更深层次的分解，是否合理？



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

8.2 划分

➤ 8.3 通信

➤ 8.3.1 方法描述

8.4 组合

8.3.2 四种通信模式

8.5 映射

8.3.3 通信判据

8.6 小结



通信方法描述

- 通信是PCAM设计过程的重要阶段;
- 划分产生的诸任务,一般不能完全独立执行,需要在任务间进行数据交流;从而产生了通信;
- 功能分解确定了诸任务之间的数据流;
- 诸任务是并发执行的,通信则限制了这种并发性;



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

8.2 划分

➤ 8.3 通信

8.3.1 方法描述

8.4 组合

➤ 8.3.2 四种通信模式

8.5 映射

8.3.3 通信判据

8.6 小结



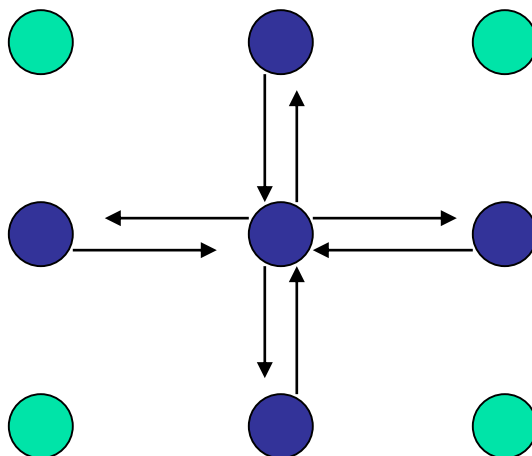
四种通信模式

- 局部/全局通信
- 结构化/非结构化通信
- 静态/动态通信
- 同步/异步通信



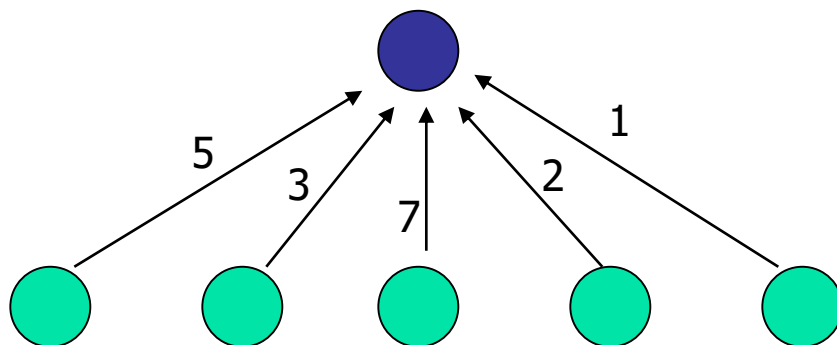
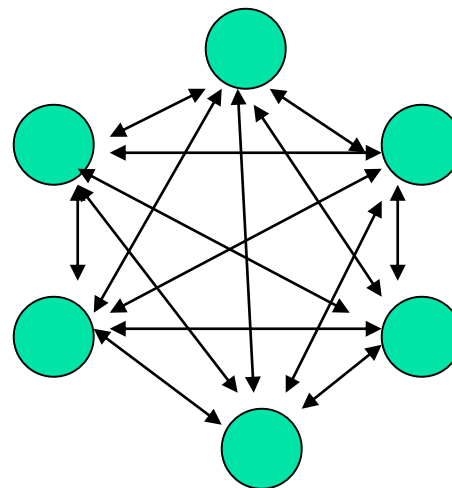
局部通信

- 通信限制在一个邻域内



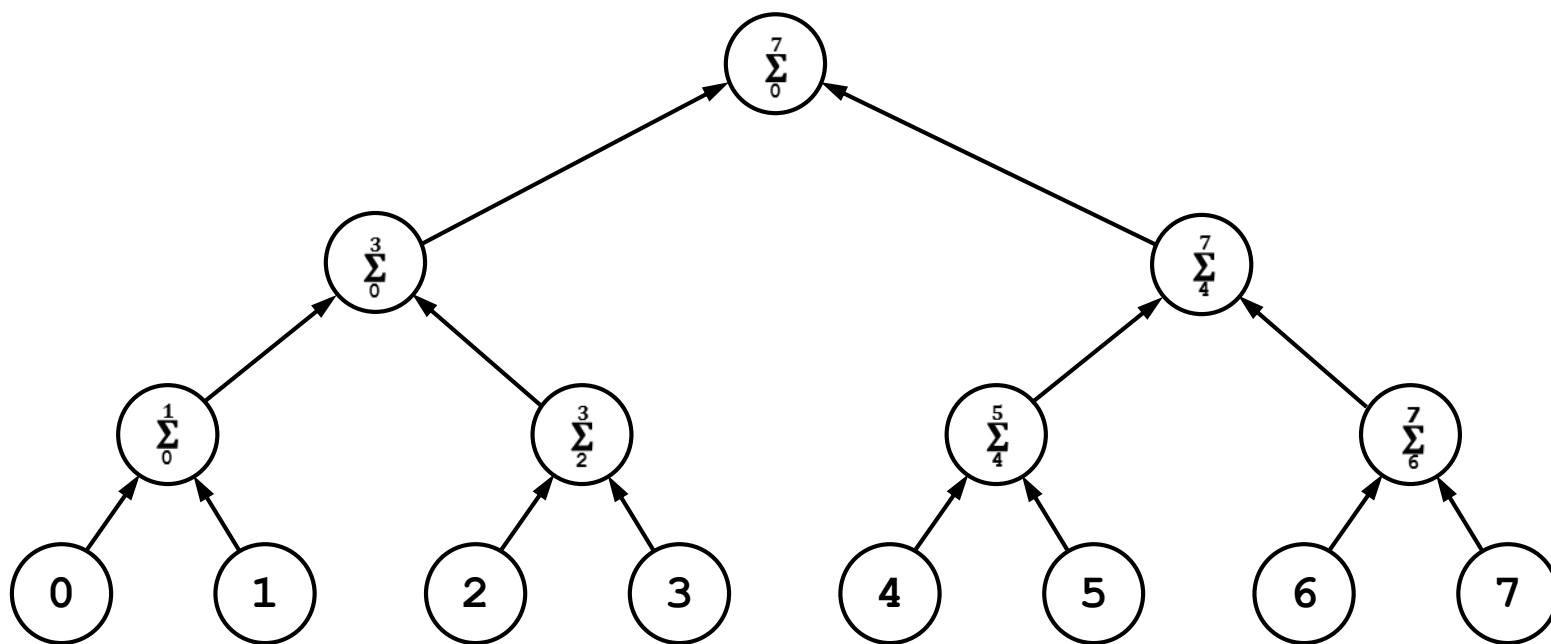
全局通信

- 通信非局部的
- 例如:
 - All to All
 - Master-Worker



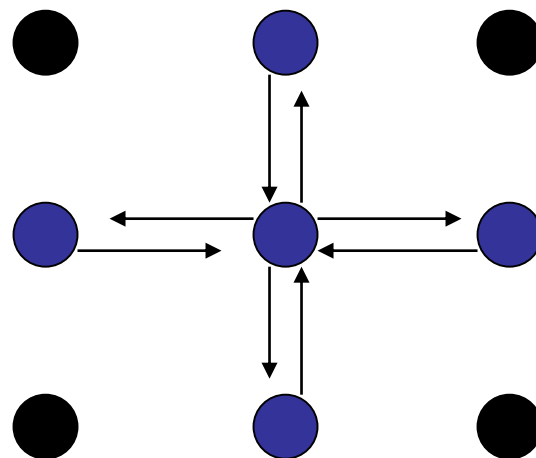
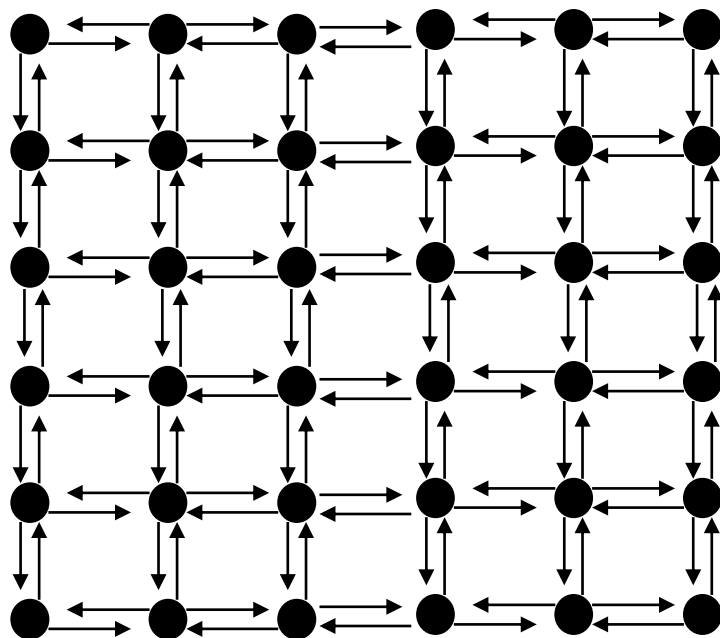
将全局通信化为局部通信

■ N=8时的分治求和树



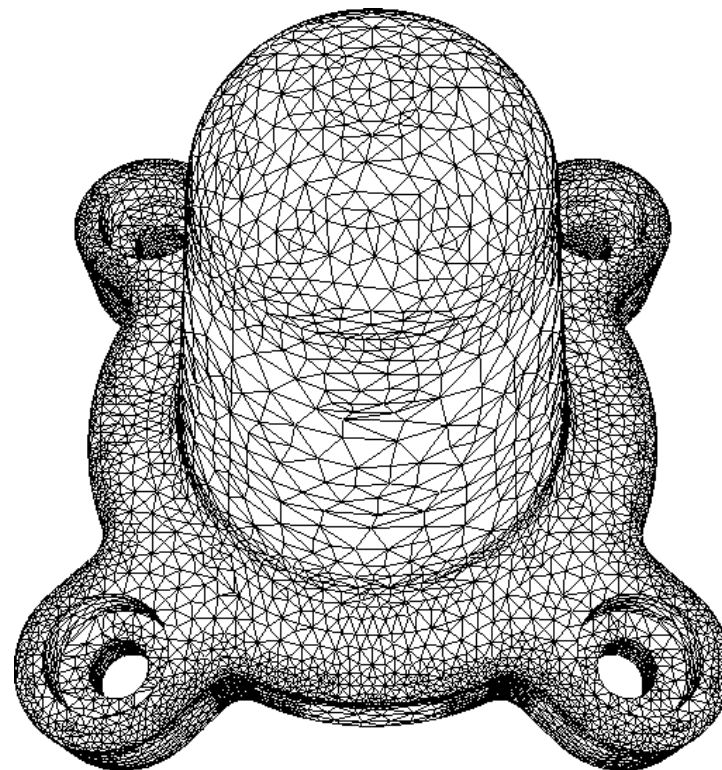
结构化通信

- 每个任务的通信模式是相同的;
- 下面是否存在一个相同通信模式?



非结构化通信

- 没有一个统一的通信模式
- 例如：无结构化网格



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

8.2 划分

➤ 8.3 通信

8.3.1 方法描述

8.4 组合

8.3.2 四种通信模式

8.5 映射

➤ 8.3.3 通信判据

8.6 小结



通信判据

- 所有任务是否执行大致相当的通信?
- 是否尽可能地局部通信?
- 通信操作是否能并行执行?
- 同步任务的计算能否并行执行?



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

8.2 划分

8.3 通信

➤ 8.4 组合

➤ 8.4.1 方法描述

8.5 映射

8.4.2 表面-容积效应

8.6 小结

8.4.3 重复计算

8.4.4 组合判据



方法描述

- 组合是由抽象到具体的过程，是将组合的任务能在一类并行机上有效的执行；
- 合并小尺寸任务，减少任务数。如果任务数恰好等于处理器数，则也完成了映射过程；
- 通过增加任务的粒度和重复计算，可以减少通信成本；
- 保持映射和扩展的灵活性，降低软件工程成本；



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

8.2 划分

8.3 通信

➤ 8.4 组合

8.4.1 方法描述

8.5 映射

➤ 8.4.2 表面-容积效应

8.6 小结

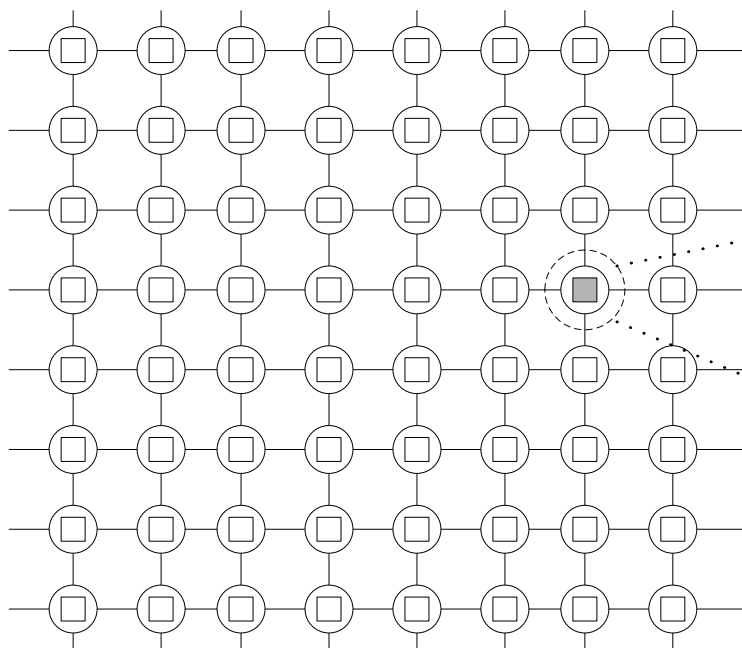
8.4.3 重复计算

8.4.4 组合判据



表面-容积效应

- 通信量与任务子集的表面积成正比，计算量与任务子集的体积成正比（对二维而言，通信量与周长成正比，计算量则与面积成正比）；
- 增加重复计算有可能减少通信量；

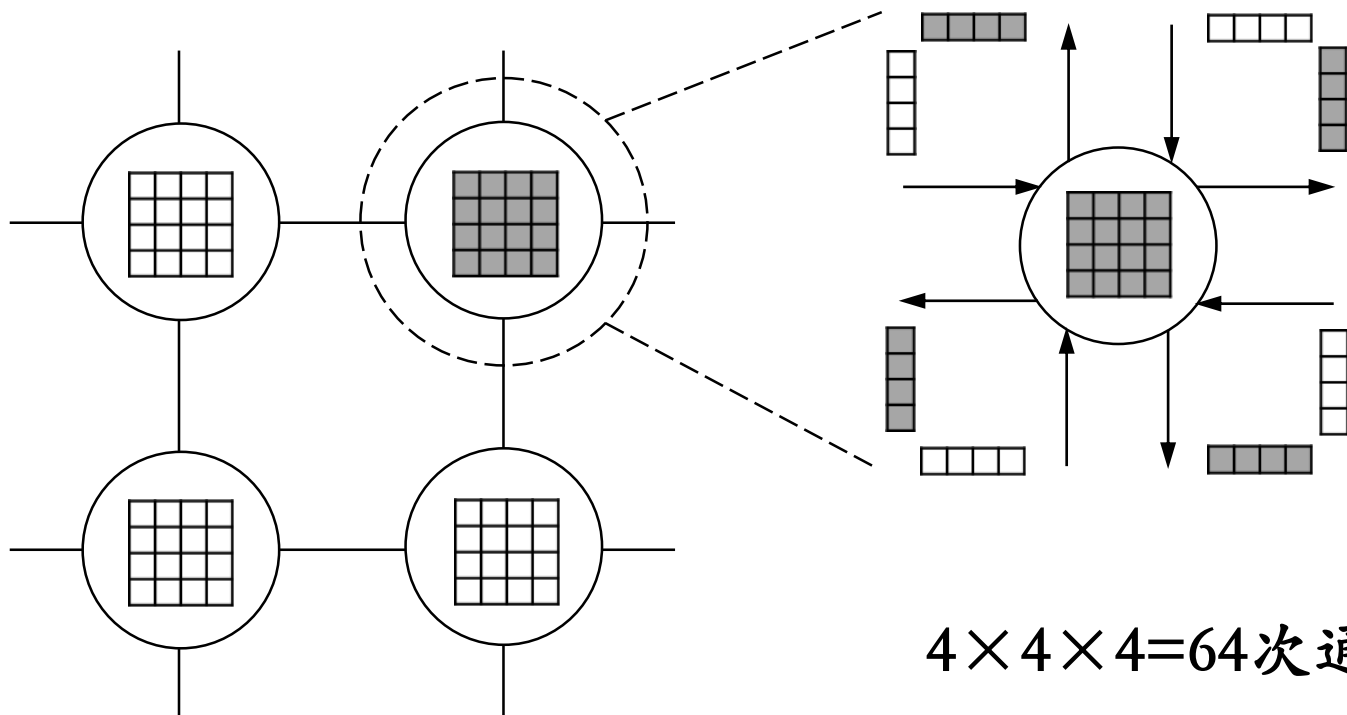


$$8 \times 8 \times 4 = 256 \text{ 次通信}$$



表面-容积效应

- 二维网格5点差分格式中增加粒度对通信成本的影响:



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

8.2 划分

8.3 通信

➤ 8.4 组合

8.4.1 方法描述

8.5 映射

8.4.2 表面-容积效应

8.6 小结

➤ 8.4.3 重复计算

8.4.4 组合判据



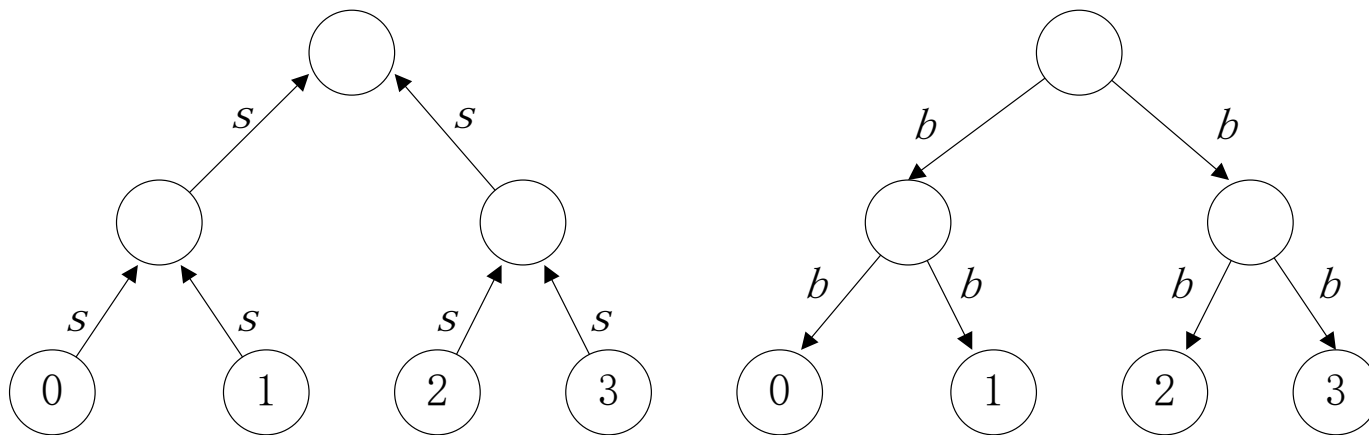
重复计算

- 重复计算减少通信量，但增加了计算量，应保持恰当的平衡；
- 重复计算的目标应减少算法的总运算时间；



重复计算

- 示例：二叉树上N个处理器求N个数的全和，要求每个处理器均保持全和。

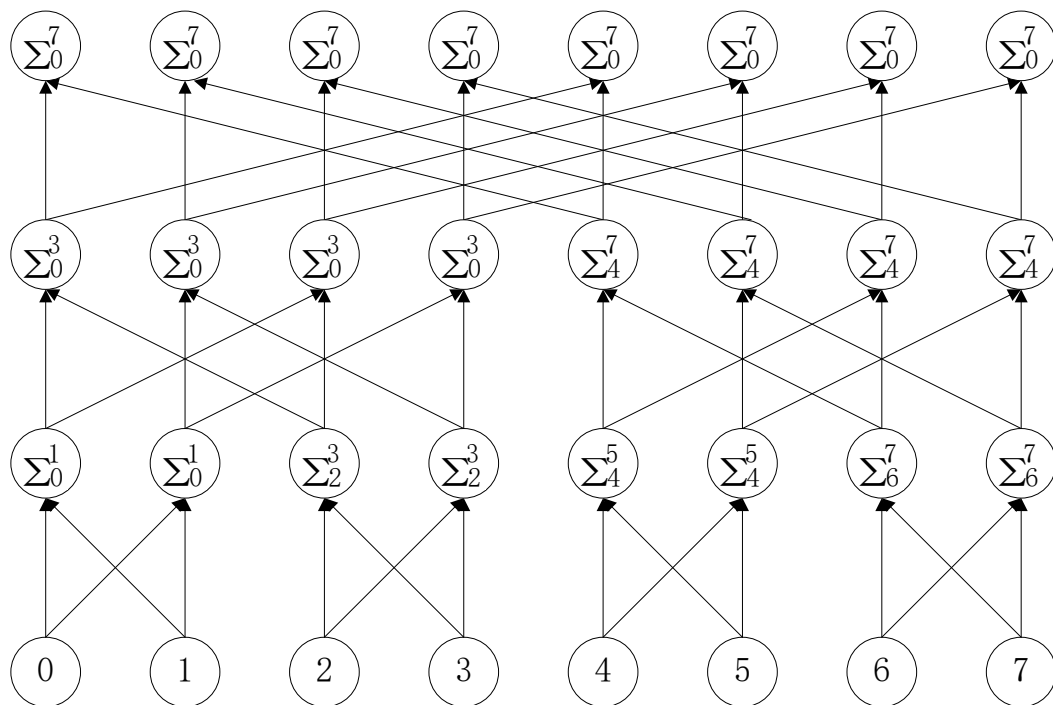


二叉树上求和，共需 $2\log_2 N$ 步



重复计算

- 示例：二叉树上N个处理器求N个数的全和，要求每个处理器均保持全和。



蝶式结构求和，使用了重复计算，共需 $\log_2 N$ 步。



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

8.2 划分

8.3 通信

➤ 8.4 组合

8.4.1 方法描述

8.5 映射

8.4.2 表面-容积效应

8.6 小结

8.4.3 重复计算

➤ 8.4.4 组合判据



组合判据

- 增加粒度是否减少了通信成本?
- 重复计算是否已权衡了其得益?
- 是否保持了灵活性和可扩放性?
- 组合的任务数是否与问题尺寸成比例?
- 是否保持了类似的计算和通信?
- 有没有减少并行执行的机会?



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

8.2 划分

8.3 通信

8.4 组合

➤ 8.5 映射

➤ 8.5.1 方法描述

8.6 小结

8.5.2 负载均衡算法

8.5.3 任务调度算法

8.5.4 映射判据



方法描述

- 每个任务要映射到具体的处理器，定位到运行机器上；
- 任务数大于处理器数时，存在负载平衡和任务调度问题；
- 映射的目标：减少算法的执行时间
 - 并发的任务 → 不同的处理器
 - 任务之间存在高通信的 → 同一处理器
- 映射实际是一种权衡，属于NP完全问题；



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

8.2 划分

8.3 通信

8.4 组合

➤ 8.5 映射

8.5.1 方法描述

➤ 8.5.2 负载均衡算法

8.6 小结

8.5.3 任务调度算法

8.5.4 映射判据



负载均衡算法

- 静态的：事先确定；
- 概率的：随机确定；
- 动态的：执行期间动态负载；
- 基于域分解的：
 - 递归对剖
 - 局部算法
 - 概率方法
 - 循环映射



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

8.2 划分

8.3 通信

8.4 组合

➤ 8.5 映射

8.5.1 方法描述

8.5.2 负载平衡算法

➤ 8.5.3 任务调度算法

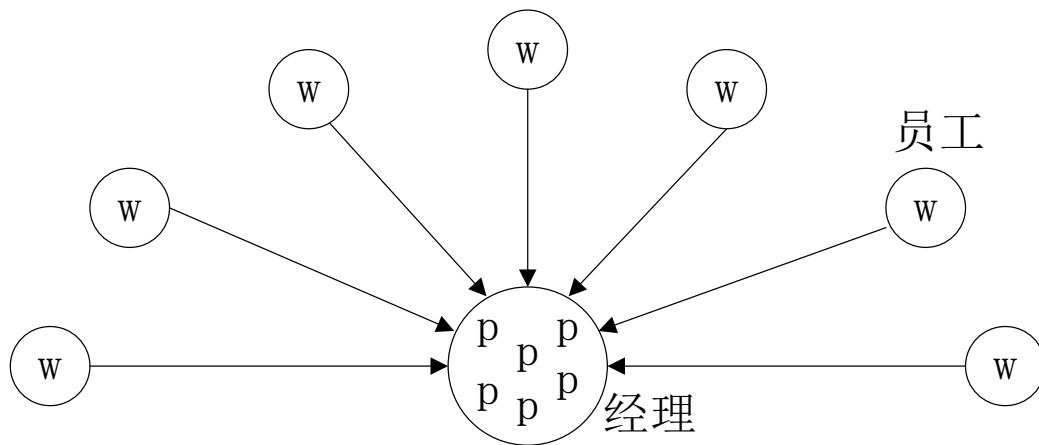
8.5.4 映射判据

8.6 小结



任务调度算法

- 任务放在集中的或分散的任务池中，使用任务调度算法将池中的任务分配给特定的处理器。下面是两种常用调度模式：
- 经理/雇员模式



- 非集中模式



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

8.2 划分

8.3 通信

8.4 组合

➤ 8.5 映射

8.5.1 方法描述

8.5.2 负载平衡算法

8.5.3 任务调度算法

➤ 8.5.4 映射判据



映射判据

- 采用集中式负载均衡方案，是否存在通信瓶颈？
- 采用动态负载均衡方案，调度策略的成本如何？



第八章 并行算法一般设计过程

8.1 PCAM设计方法学

8.2 划分

8.3 通信

8.4 组合

8.5 映射

➤ 8.6 小结



北京航空航天大学
COLLEGE OF SOFTWARE
BEIHANG UNIVERSITY 软件学院

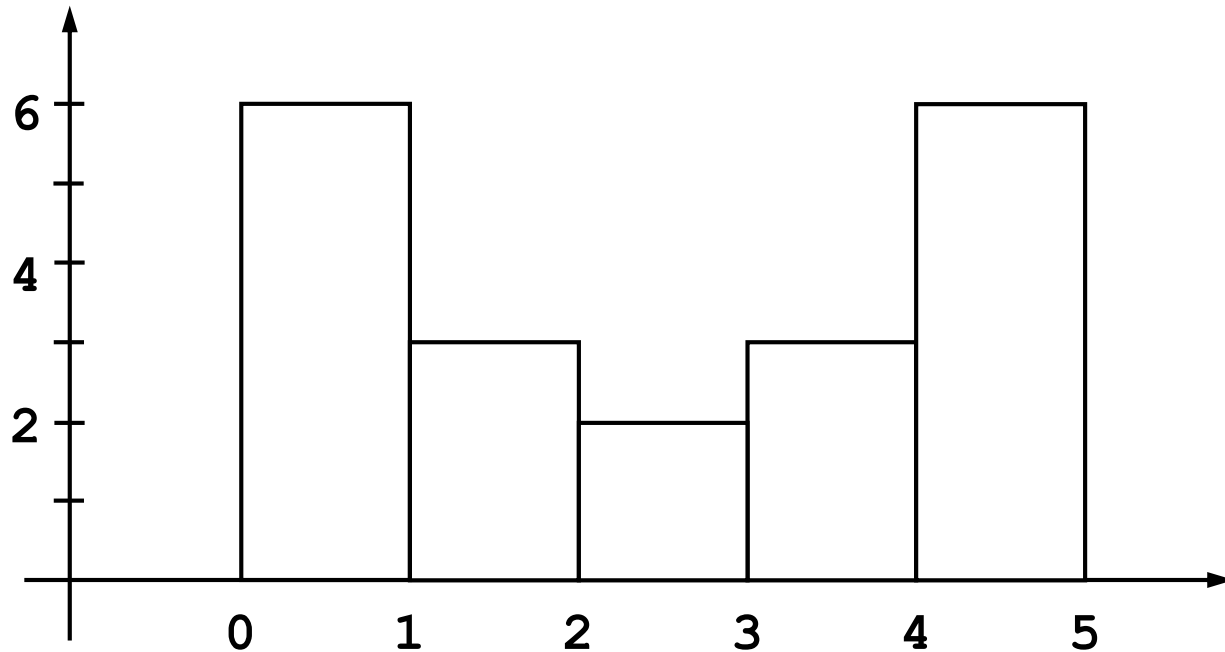
小结

- 划分
 - 域分解和功能分解
- 通信
 - 任务间的数据交换
- 组合
 - 任务的合并使得算法更有效
- 映射
 - 将任务分配到处理器，并保持负载平衡



Example – histogram (直方图)

- 1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9



Serial program - input

1. The number of measurements: **data_count**
2. An array of data_count floats: **data**
3. The minimum value for the bin containing the smallest values: **min_meas**
4. The maximum value for the bin containing the largest values: **max_meas**
5. The number of bins: **bin_count**

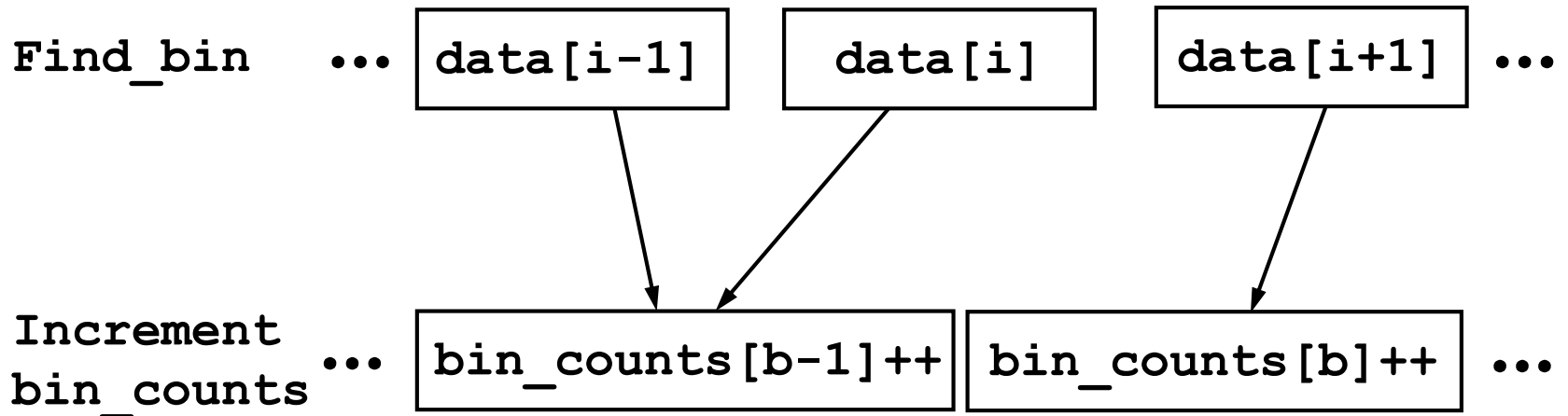


Serial program - output

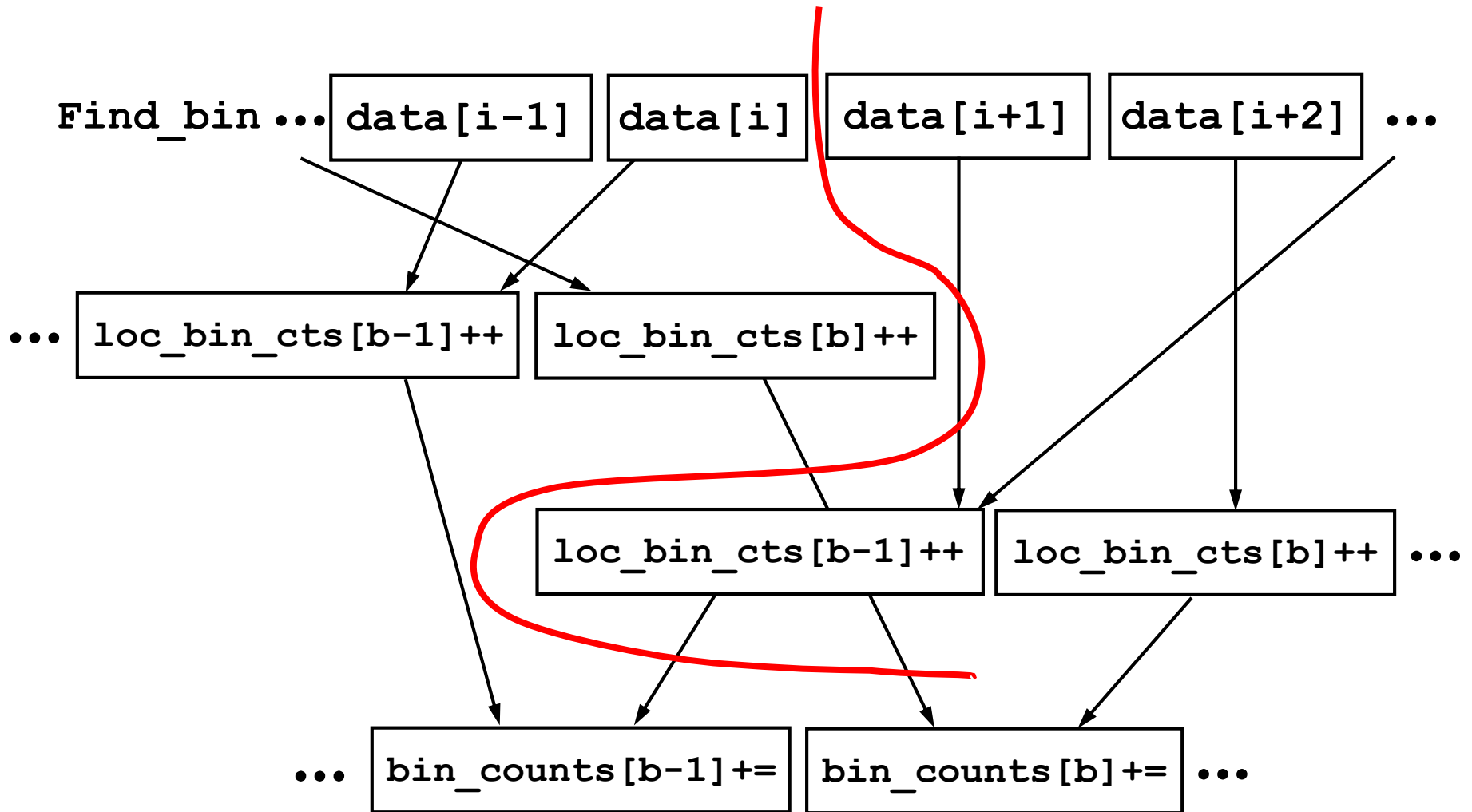
1. **bin_maxes** : an array of bin_count floats
2. **bin_counts** : an array of bin_count ints



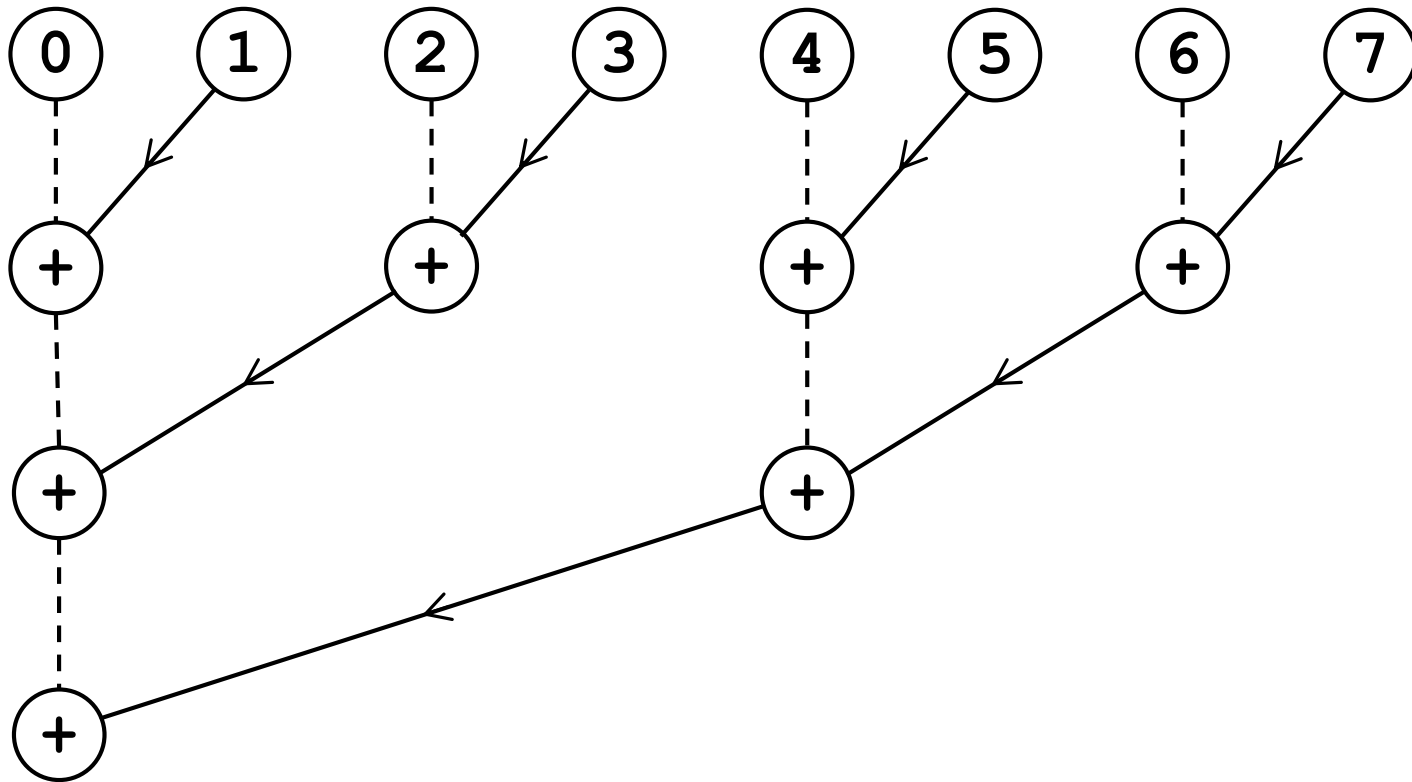
First two stages of Foster's Methodology



Alternative definition of tasks and communication



Adding the local arrays



Concluding Remarks (1)

- **Serial systems**

- The standard model of computer hardware has been the von Neumann architecture.

- **Parallel hardware**

- Flynn's taxonomy.

- **Parallel software**

- We focus on software for homogeneous MIMD systems, consisting of a single program that obtains parallelism by branching.
 - SPMD programs.



Concluding Remarks (2)

■ Input and Output

- We'll write programs in which one process or thread can access **stdin**, and all processes can access **stdout** and **stderr**.
- However, because of nondeterminism, except for debug output we'll usually have a single process or thread accessing **stdout**.



Concluding Remarks (3)

- **Performance**
 - Speedup
 - Efficiency
 - Amdahl's law
 - Scalability
- **Parallel Program Design**
 - Foster's methodology

