

# 第一章 并行计算导论

哈尔滨工业大学

张伟哲

2025, Fall Semester

# 目录

- 什么是并行计算
- 为什么需要并行计算
- 并行计算发展
- 并行计算面临的挑战

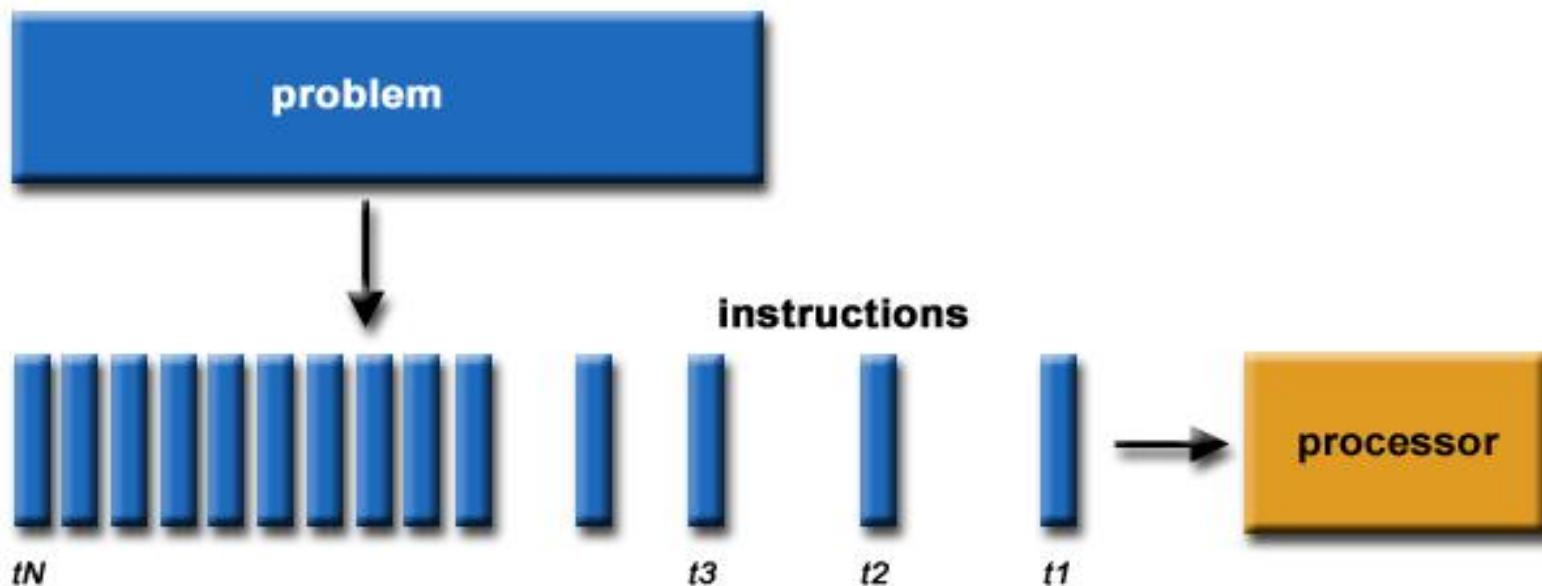
# 目录

- 什么是并行计算
- 为什么需要并行计算
- 并行计算发展
- 并行计算面临的挑战

# 什么是并行计算?

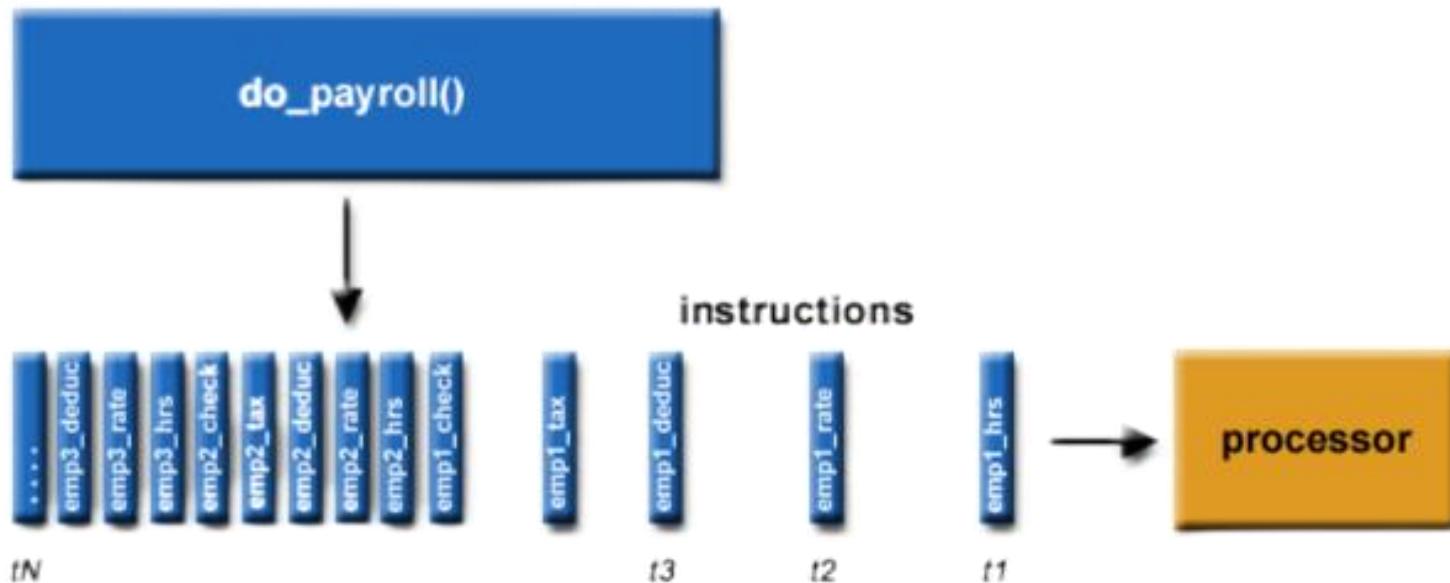
## ■ 传统情况下，程序是串行的(Serial Computing)

- 问题被分解成一系列离散的指令
- 这些指令顺序执行
- 单个处理器上执行
- 任意时刻只能有一条指令再执行



# 什么是并行计算?

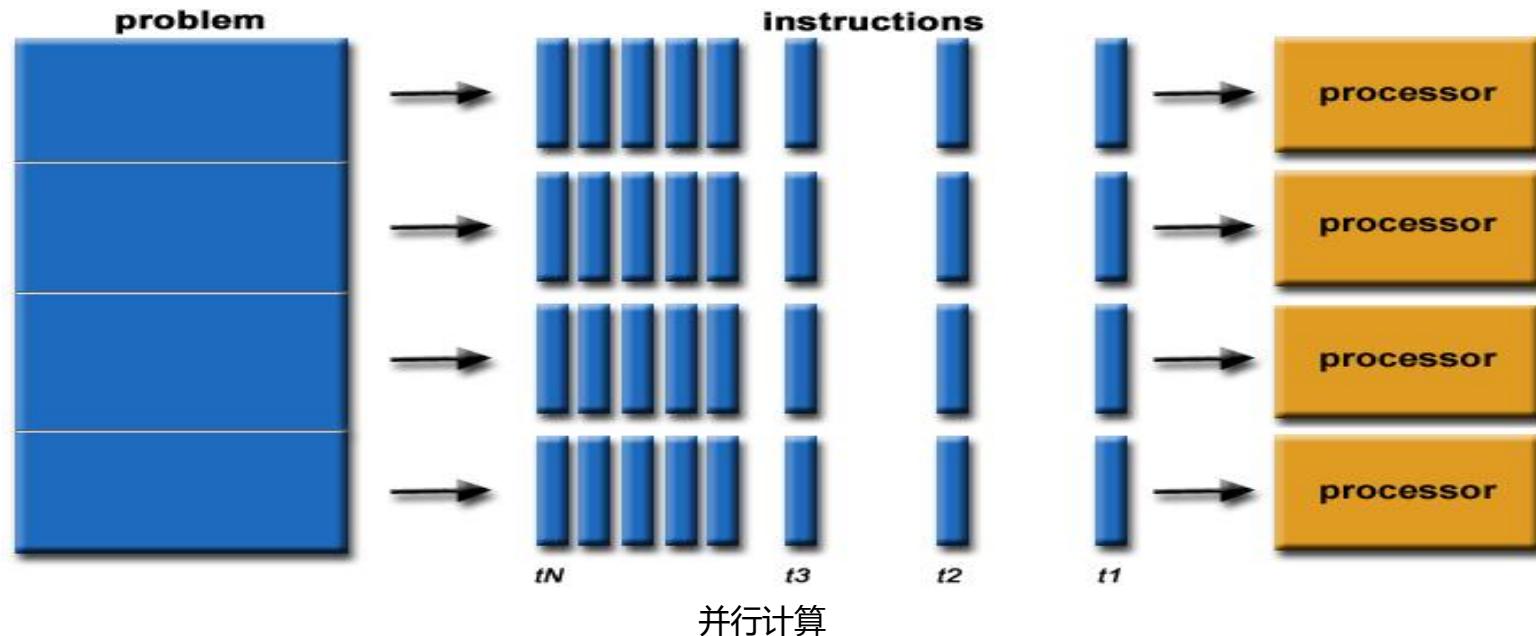
- 传统情况下，程序是串行的(Serial Computing)
  - 问题被分解成一系列离散的指令
  - 这些指令顺序执行
  - 单个处理器上执行
  - 任意时刻只能有一条指令再执行



# 什么是并行计算?

- 并行计算(Parallel Computing): 同时使用多种计算资源解决计算问题

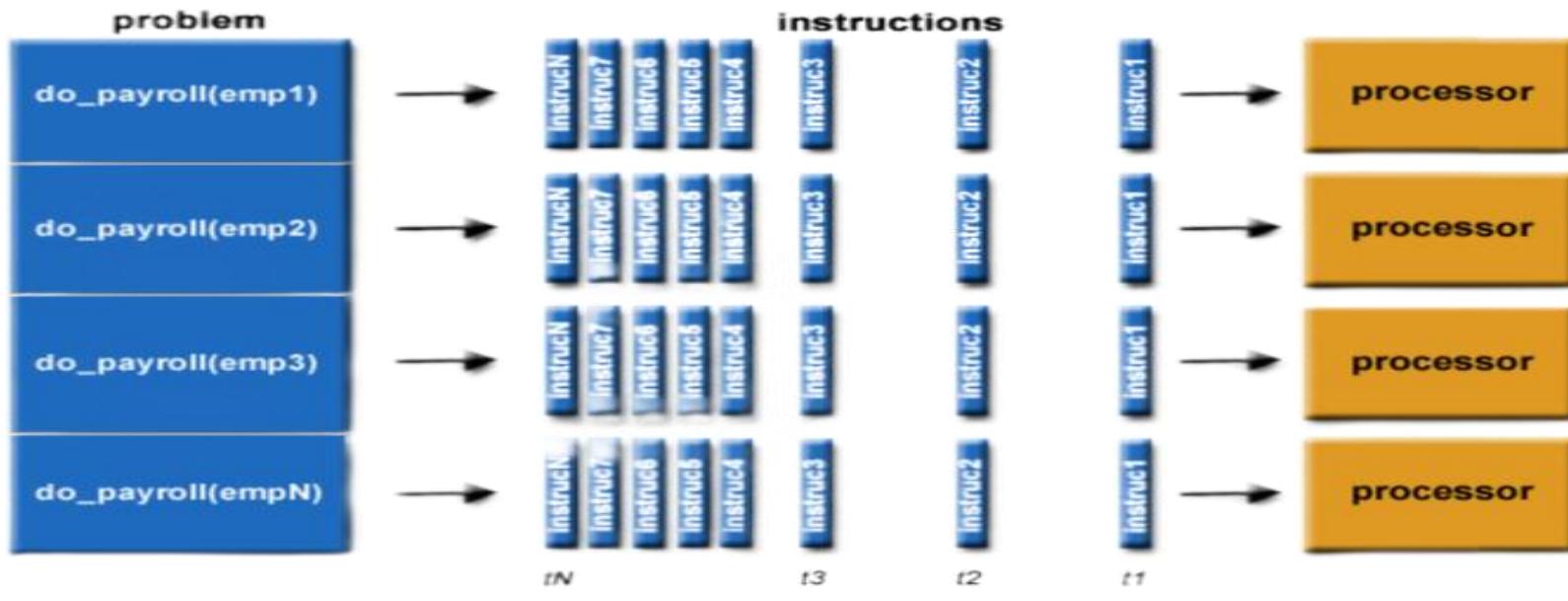
- 问题被分为离散的部分
- 每个部分进一步分解为离散的指令
- 每个部分指令同时在不同处理器执行
- 总体控制和协调机制



# 什么是并行计算?

- 并行计算(Parallel Computing): 同时使用多种计算资源解决计算问题

- 问题被分为离散的部分
- 每个部分进一步分解为离散的指令
- 每个部分指令同时在不同处理器执行
- 总体控制和协调机制



# 什么是并行计算？

## ■ 身边的并行计算

华为



小米



魅族



苹果



骁龙888, 麒麟9000,  
Exynos 8895, A14

... 2核、4核

英特尔® 酷睿，苹果M1

2核、4核、8核

...

华为



小米



苹果



神威26010众核处理器  
、Matrix-2000、A64FX

...

1064万核、498万  
核、763万核



神威



天河



富岳

# 目录

- 什么是并行计算
- 为什么需要并行计算**
- 并行计算发展
- 并行计算面临的挑战

# 为什么需要并行计算？

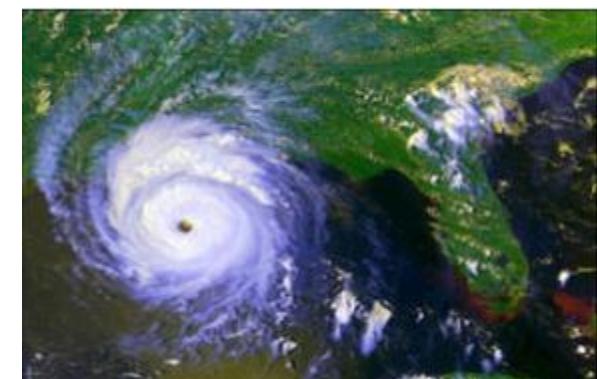
■ 并行计算更适合模拟和理解复杂的现实世界现象



星系的形成



行星的运动



气候的变化



高峰期交通



板块运动

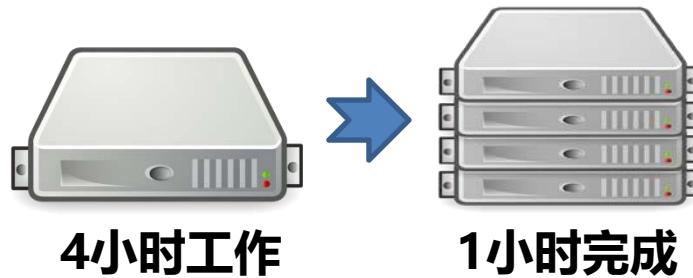


天气

# 为什么需要并行计算?

## ■ 节约大量的时间&金钱

- 为一个任务分配更多的资源 → 节约时间
- 并行计算机用更廉价的部件组合而成 → 节约金钱



	DUAL XEON CPU server	DGX-1 GPU server (8 GPUs)
FLOPS	3TF	170TF
Mem BW	76GB/s	768GB/s
Alexnet Train Time	150 Hr	2Hr
Train in 2Hr	>250Nodes	1Node
Price	>\$774000	\$129,000

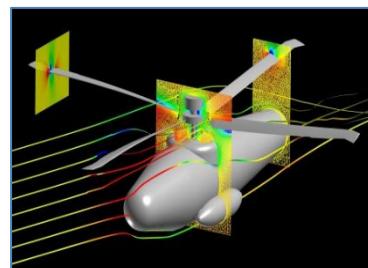


# 为什么需要并行计算?

## ■ 解决更大规模/更复杂问题

- 计算机内存限制，串行处理大规模问题不切实际
- 需要petaflops和petabytes计算资源，甚至“E级计算”
- 示例：石油勘探

- 数学问题： $Ax=b$
- 工区： $20\text{km} \times 20\text{km} \times 10\text{km}$
- 网格： $50\text{m} \times 50\text{m} \times 20\text{m}$
- 网格数量：8千万
- 数据量：TB级
- 100台服务器（节点）
- 执行时间：>10小时



# 为什么需要并行计算?

## ■ 训练更大的AI模型

- 单机/普通数据中心算力和存储不足以训练大模型
- 需要petaflops和petabytes计算资源,甚至“E级计算”
- 示例: 清华大学“八卦炉”模型

- 超算平台: 新一代神威超级计算机
- 模型参数: 174万亿
- 训练性能: > 1 EFLOPS (混合精度)

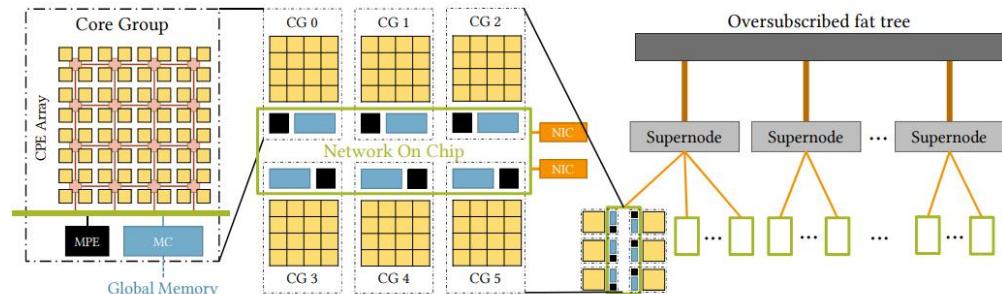
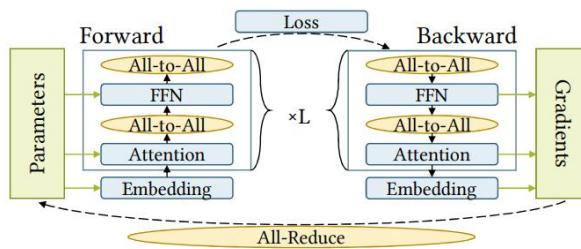


Figure 1. Simplified computing process of the proposed model.

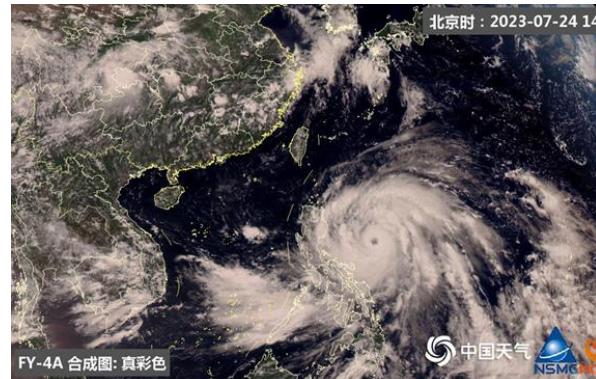
Figure 2. Architecture of the *New Generation Sunway Supercomputer*.

# 为什么需要并行计算?

## ■ 典型并行计算应用



天体物理仿真



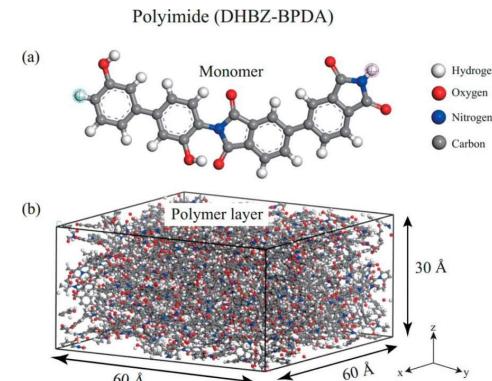
天气预测



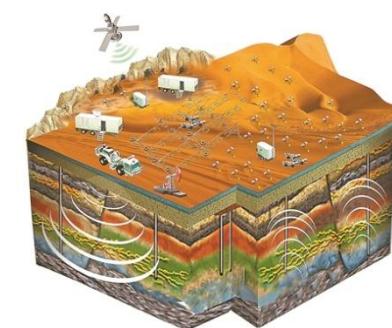
人工智能大模型



核爆模拟



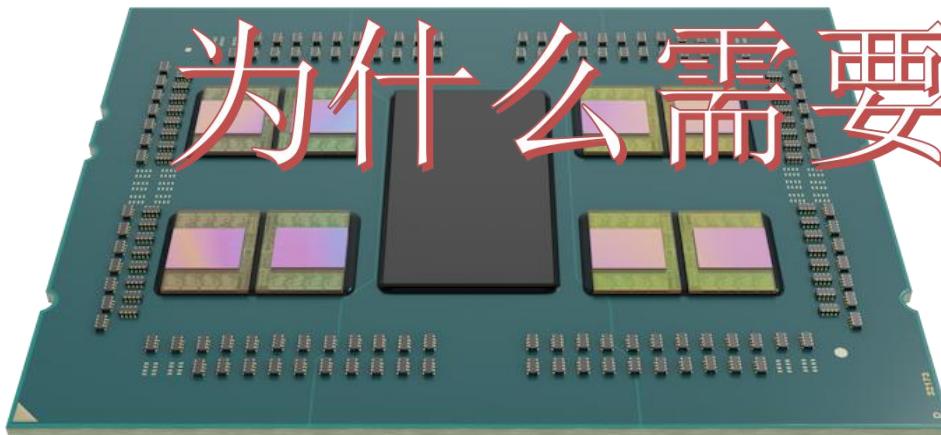
分子动力学仿真



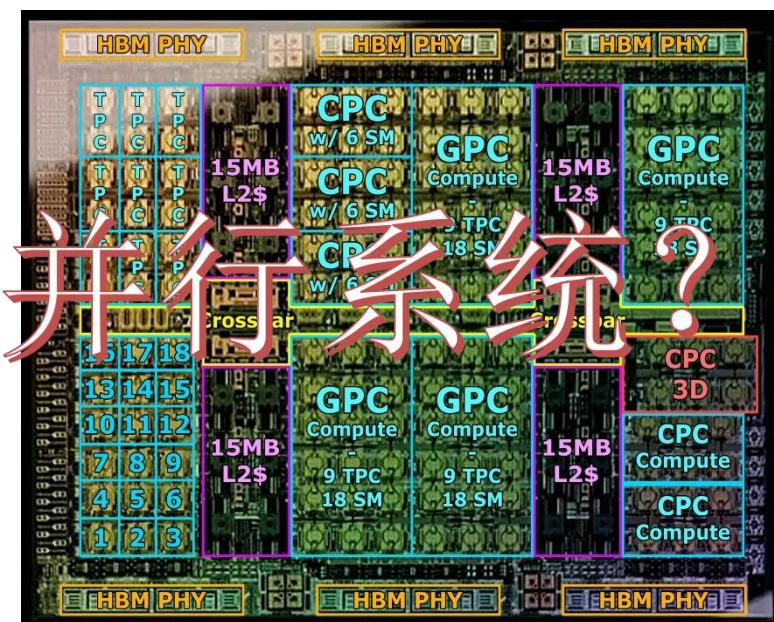
物探(矿物勘探)

# 为什么需要并行计算?

- 理解底层硬件架构，充分利用硬件性能



64 Cores AMD EPYC 7003系列CPU

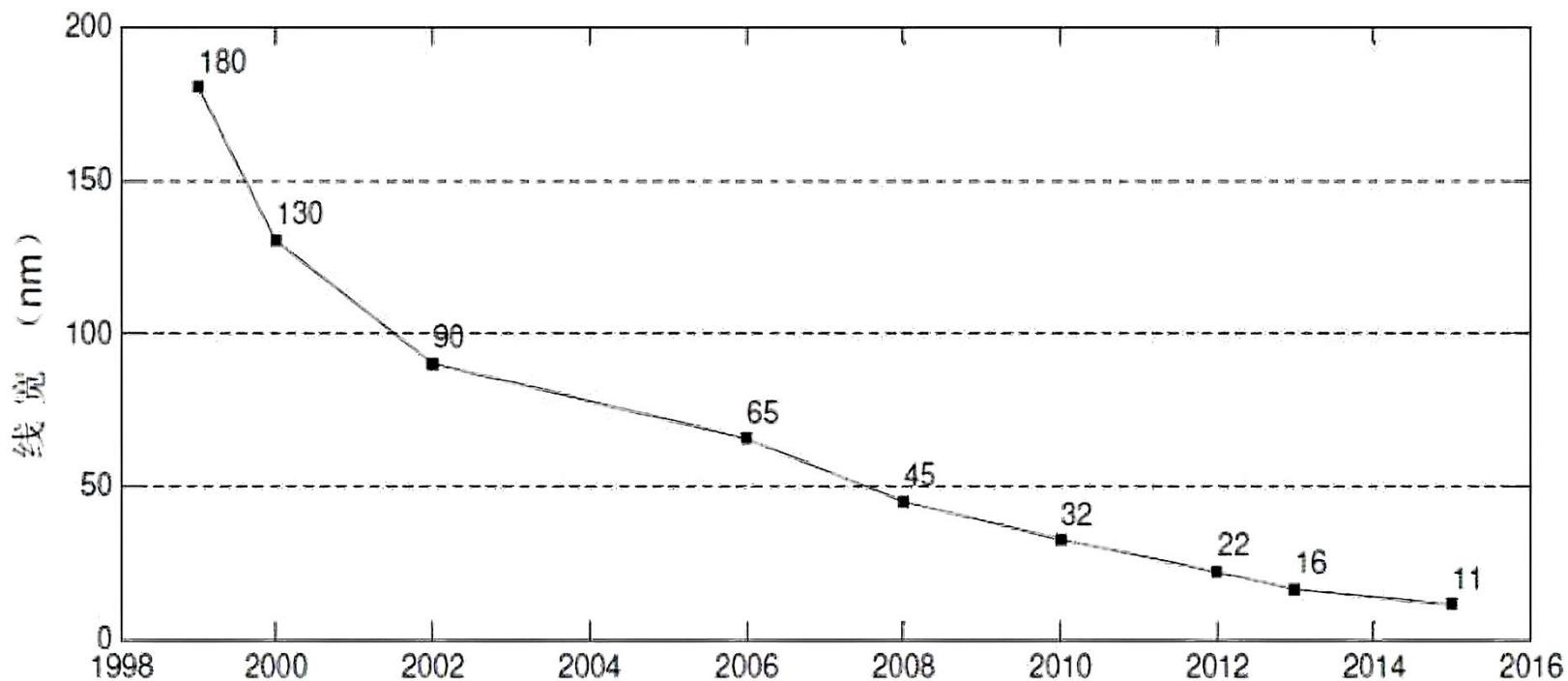


18432 Cores NVIDIA H100 GPU

# 为什么需要并行计算？

## ■ 晶体管线宽不断降低

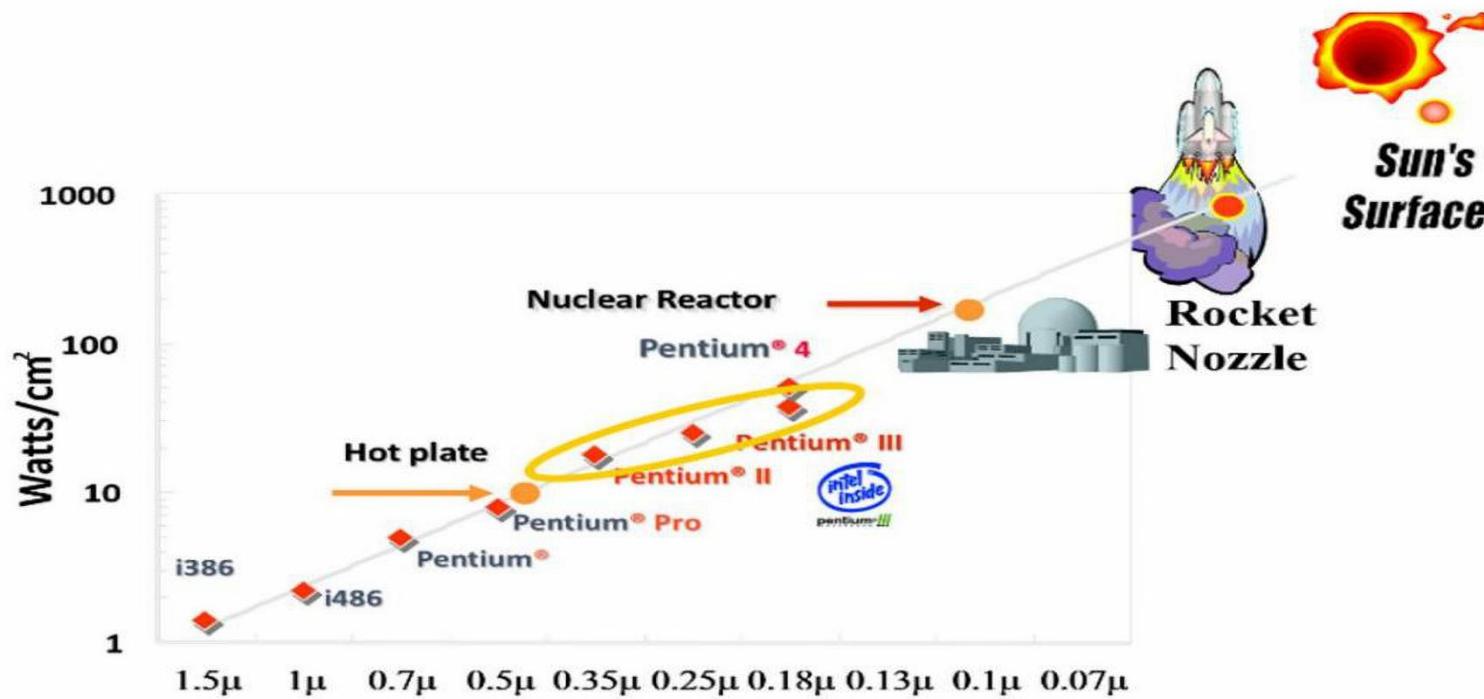
- 晶体管密度增加，频率增加，芯片性能提高
- 台积电、三星：5nm
- 中芯国际：14nm



# 为什么需要并行计算?

## ■ 芯片功耗密度不断增加

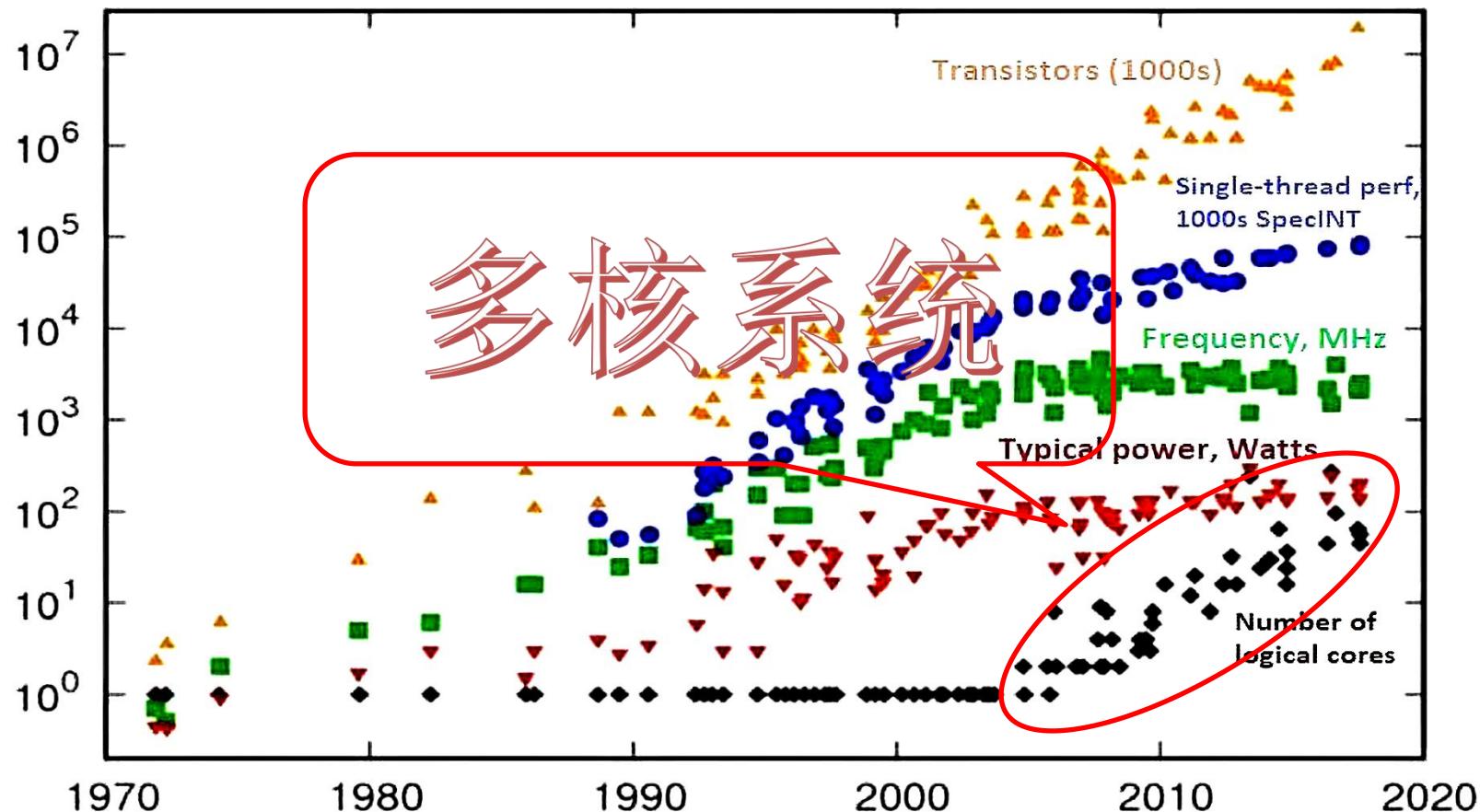
- 温度上升影响芯片的可靠性
- 45nm工艺，每百个芯片每月发生一次故障
- 16nm工艺，每百个芯片每天发生一次故障



# 为什么需要并行计算?

## ■ 单处理器的性能提升变缓

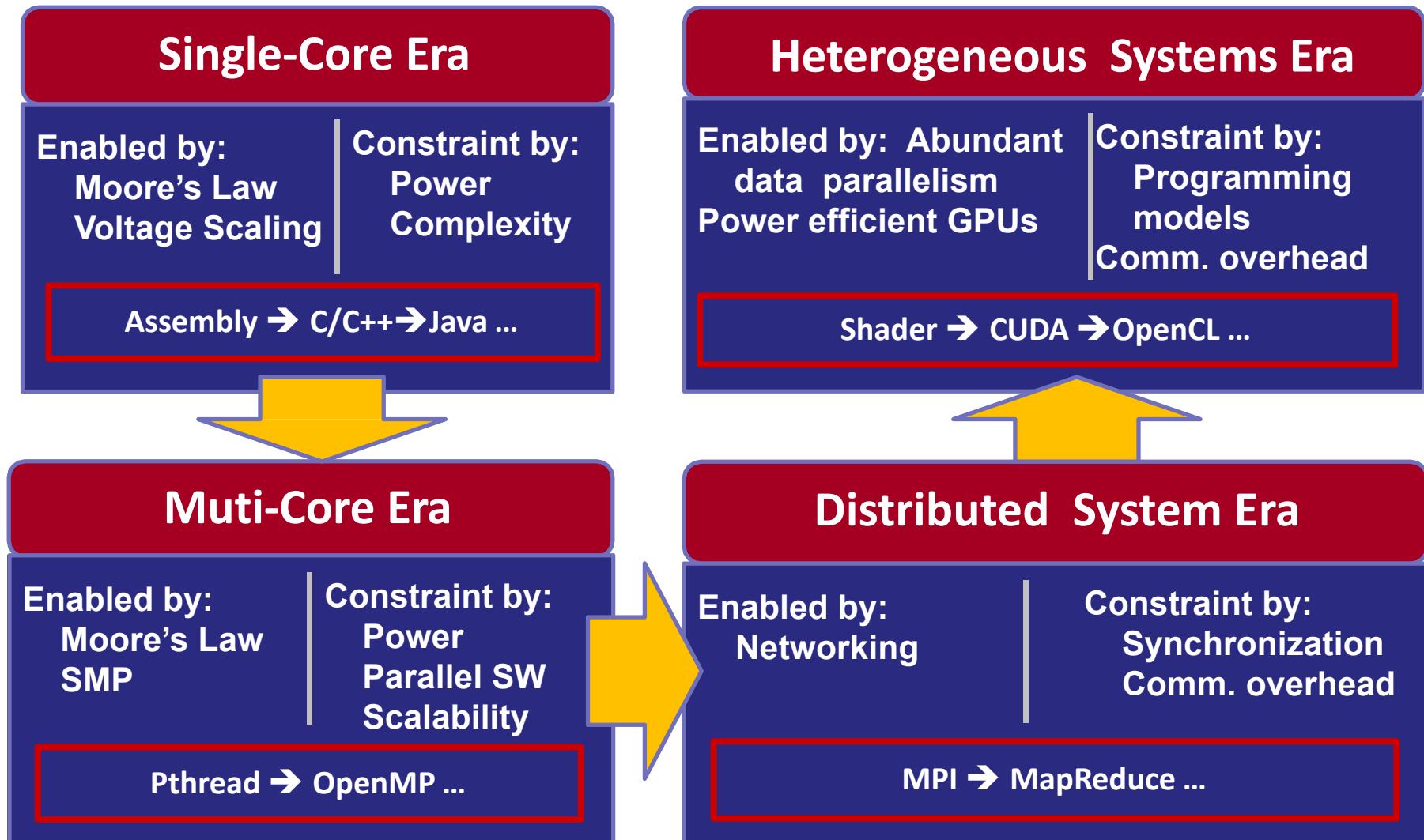
➤ 2002之前: 50%/Year; 2002之后: <20%/Year



# 目录

- 什么是并行计算
- 为什么需要并行计算
- 并行计算发展**
- 并行计算面临的挑战

# 并行计算发展

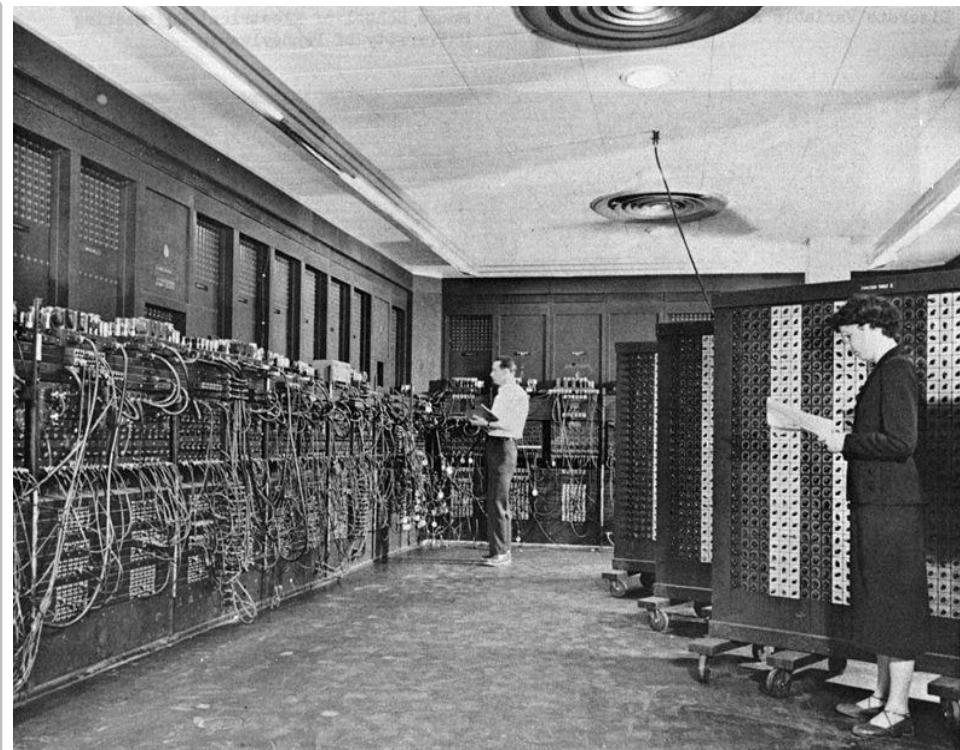


# 并行计算发展

## ■ 始于70年代

➤ 1946年第一台计算机 ENIAC  
(Electronic Numerical Integrator And Computer)

- ①、占地170平方
- ②、重约 30 吨
- ③、5000 次加法/秒或  
500次乘法/秒
- ④、15分钟换一个零件
- ⑤、主要用于弹道计算  
和氢弹研制

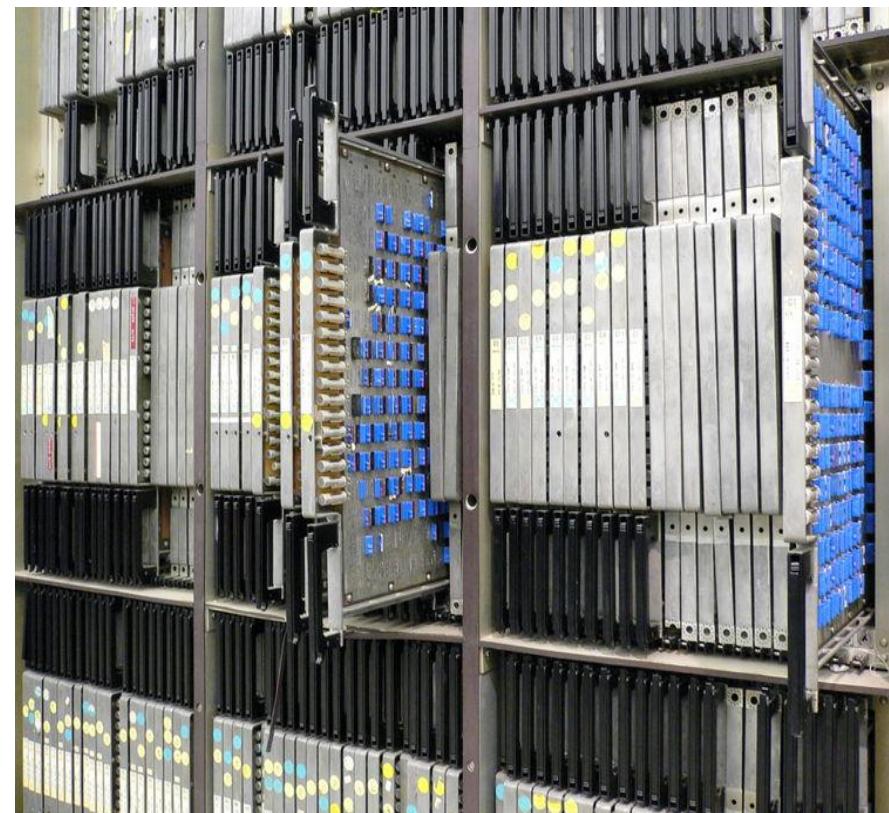


# 并行计算发展

## ■ 始于70年代

➤ 1972年第一台并行计算机 ILLIAC IV (伊利诺依大学)

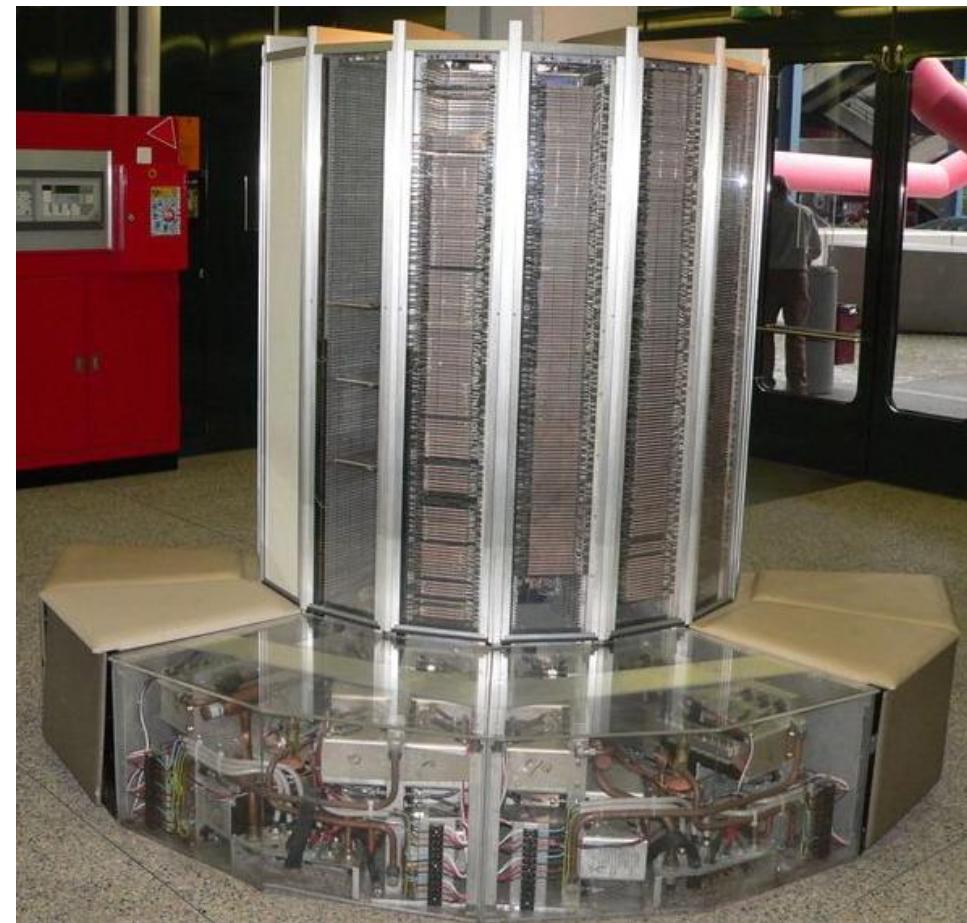
- ①、60年代末开始建造
- ②、72年建成, 74年运行第一个完整程序, 76年运行第一个应用程序
- ③、64个处理器, 是当时性能最高CDC7600机器的2-6倍
- ④、公认的1981年前最快
- ⑤、1982年退役
- ⑥、可扩展性好, 可编程性差



# 并行计算发展

- 始于70年代
  - 向量机 Cray-1

- ①、一般将 Cray-1 投入运行的 1976 年称为“超级计算元年”
- ②、编程方便，但可扩展性差
- ③、以 Cray 为代表的向量机称雄超级计算机界十几载



收藏于 Deutsches Museum 德意志博物馆的 Cray-1原型

# 并行计算发展

## ■ 80年代：百家争鸣

### ➤ 早期：以 MIMD 并行计算机的研制为主

- Denelcor HEP (1982年) 第一台商用 MIMD 并行计算机
- IBM 3090 80 年代普遍为银行所采用
- Cray X-MP Cray 研究公司第一台 MIMD 并行计算机



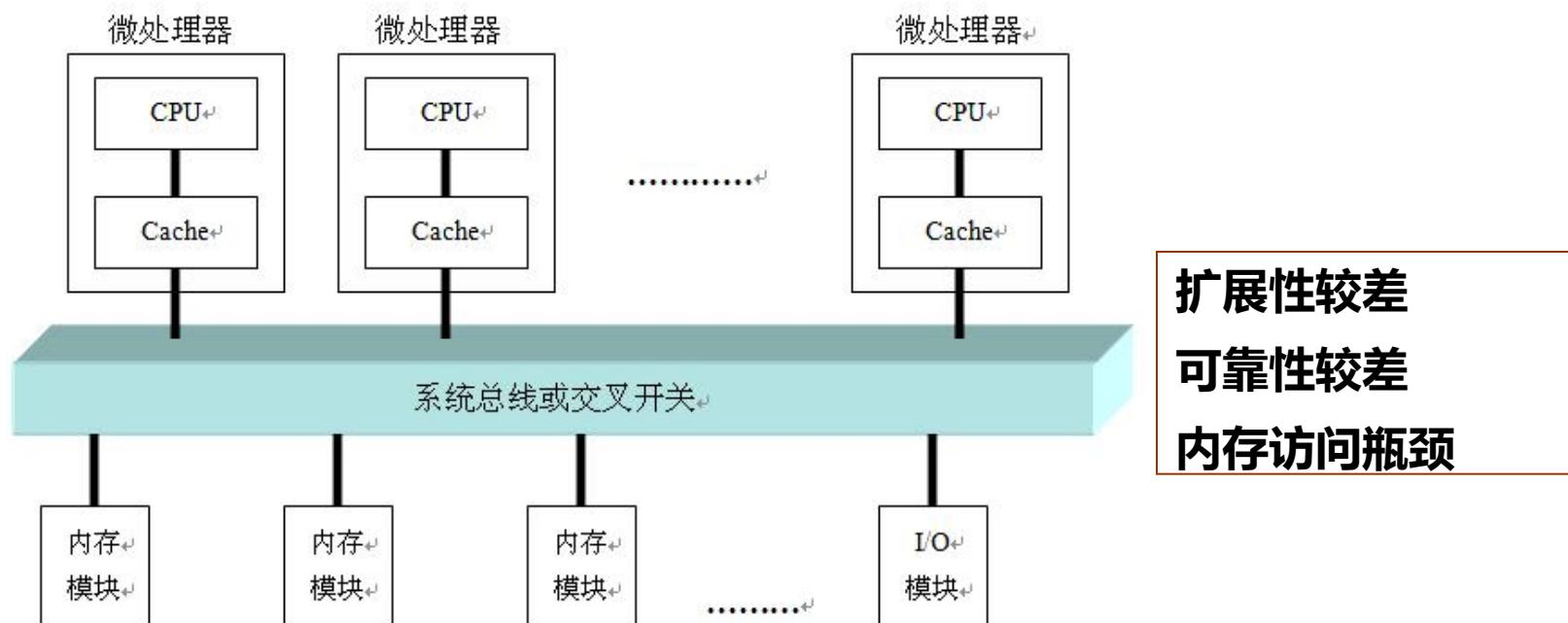
西摩·克雷 Seymour Cray (1925-1996)，  
电子工程学学士，应用数学硕士，  
超级计算机之父，Cray研究公司的创始人，  
亲手设计了Cray机型的全部硬件与操作系统，  
作业系统由他用机器码编写完成。1984年时，  
公司占据了超级计算机市场 70% 的份额。  
1996年Cray研究公司被SGI收购，2000年被  
出售给Tera计算机公司，成立Cray公司。

# 并行计算发展

## ■ 80年代：百家争鸣

### ➤ 中期：共享存储多处理机 SMP

➤ SMP：在一个计算机上汇集一组处理器，各处理器对称共享内存及计算机的其他资源，由单一操作系统管理，极大提高整个系统的数据处理能力

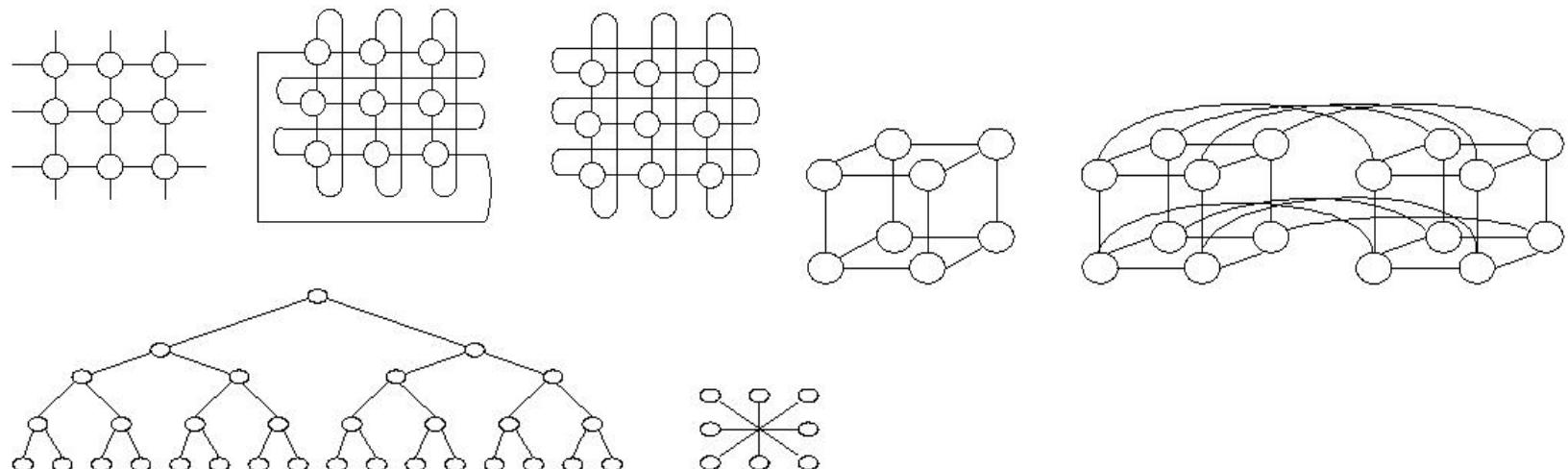


# 并行计算发展

## ■ 80年代：百家争鸣

### ➤ 后期：具有强大计算能力的并行机

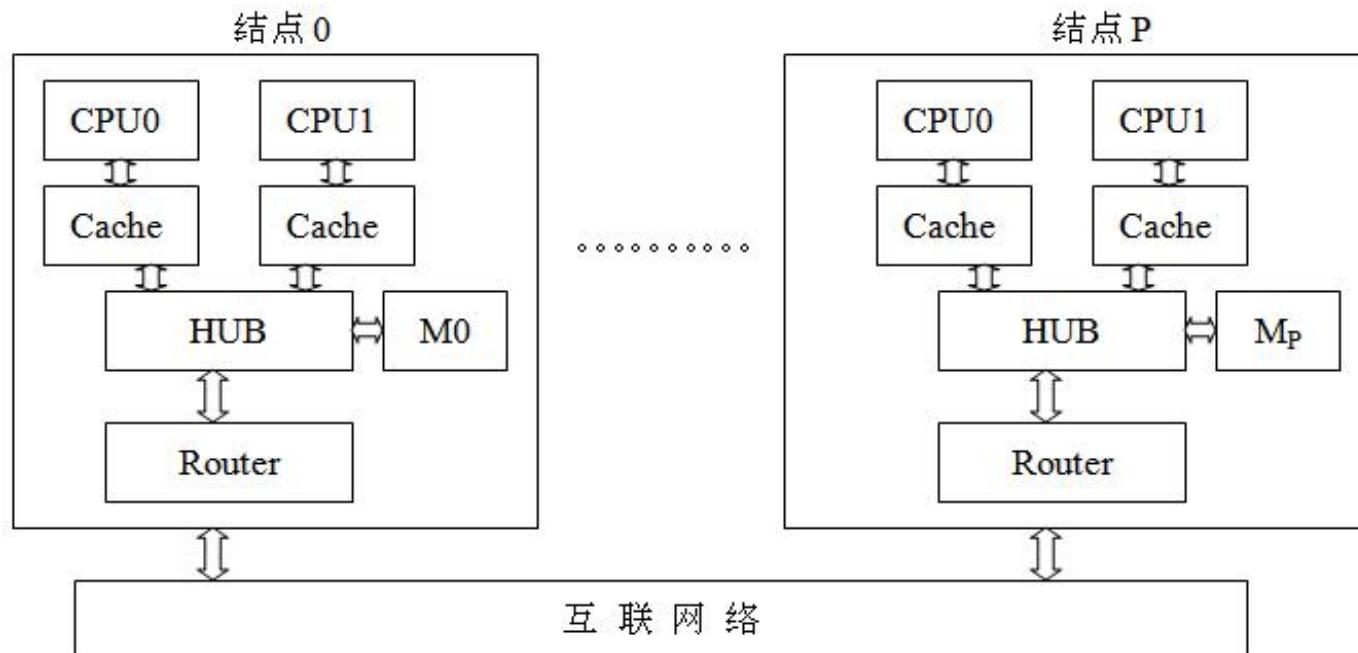
- 通过二维Mesh连接的Meiko (Sun) 系统
- 超立方体连接的 MIMD 并行机：nCUBE-2、iPSC/80
- 共享存储向量多处理机 Cray Y-MP
- .....



# 并行计算发展

## ■ 90 年代：体系结构框架趋于统一

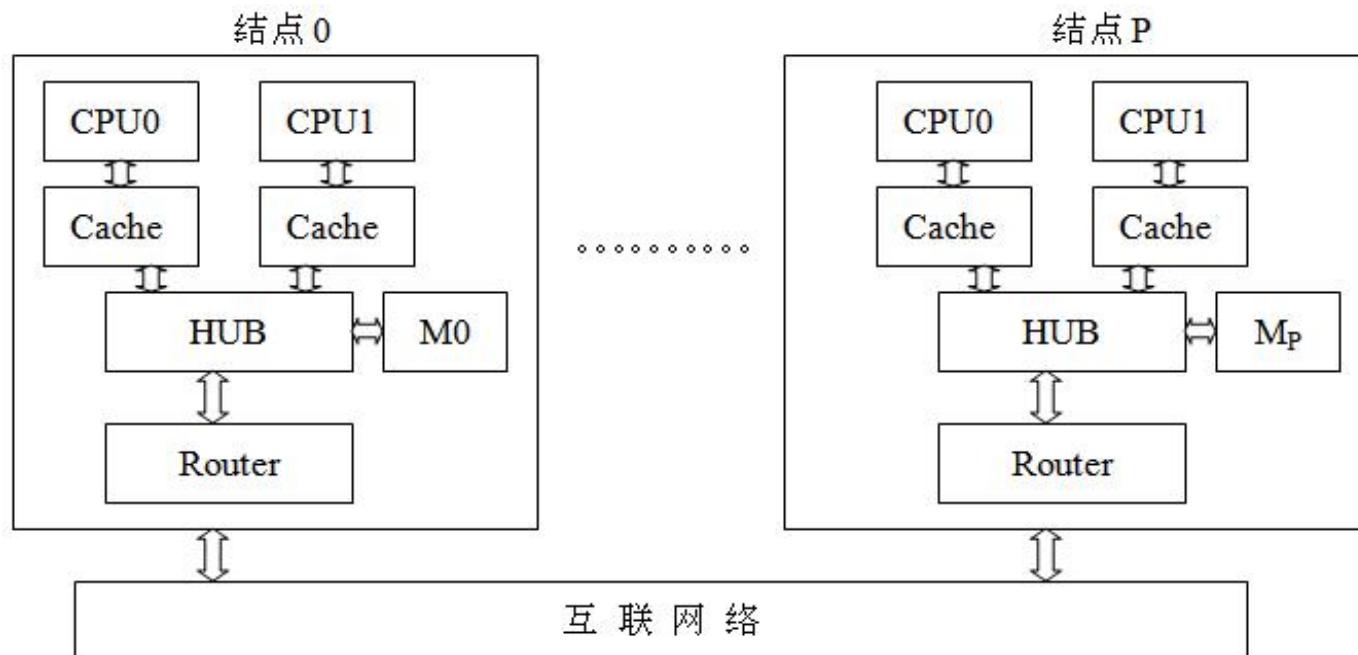
- DSM (Distributed Shared Memory) 分布式共享存储
  - 以结点为单位，每个结点有一个或多个CPU
  - 专用的高性能互联网络连接 (Myrinet, Infiniband, ... )



# 并行计算发展

## ■ 90 年代：体系结构框架趋于统一

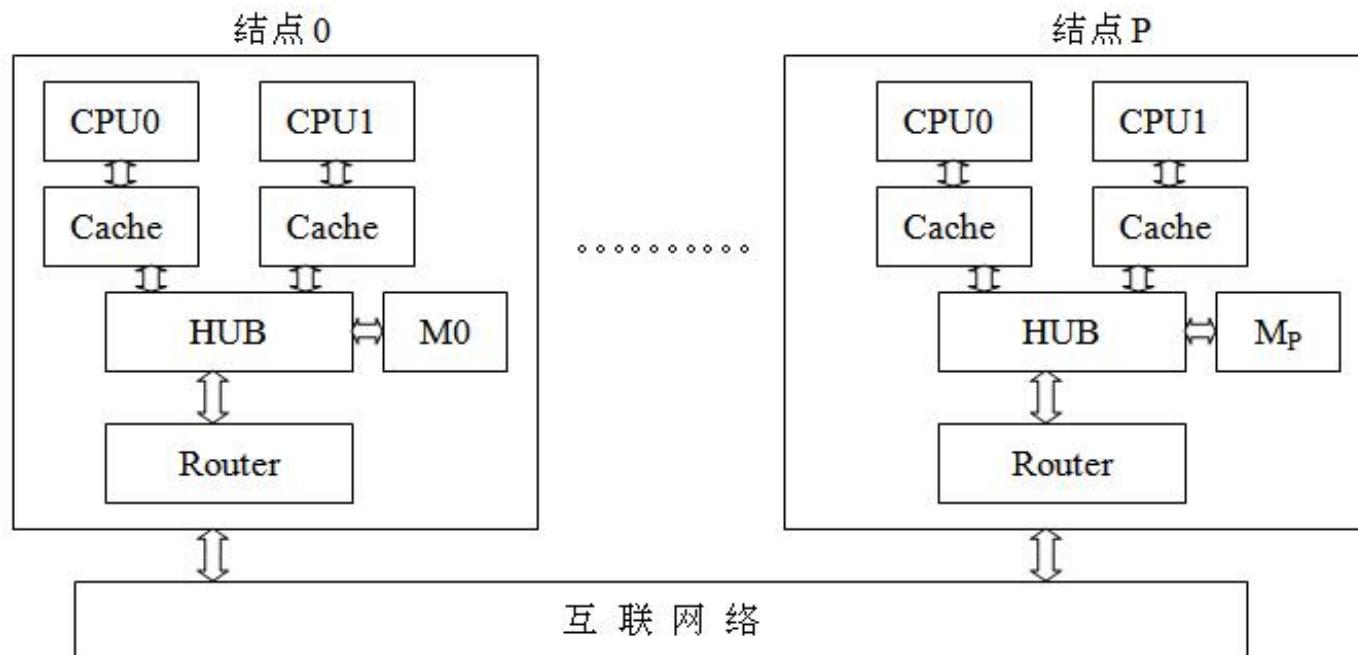
- DSM (Distributed Shared Memory) 分布式共享存储
  - 分布式存储：内存模块局部在每个结点中
  - 单一的操作系统



# 并行计算发展

## ■ 90 年代：体系结构框架趋于统一

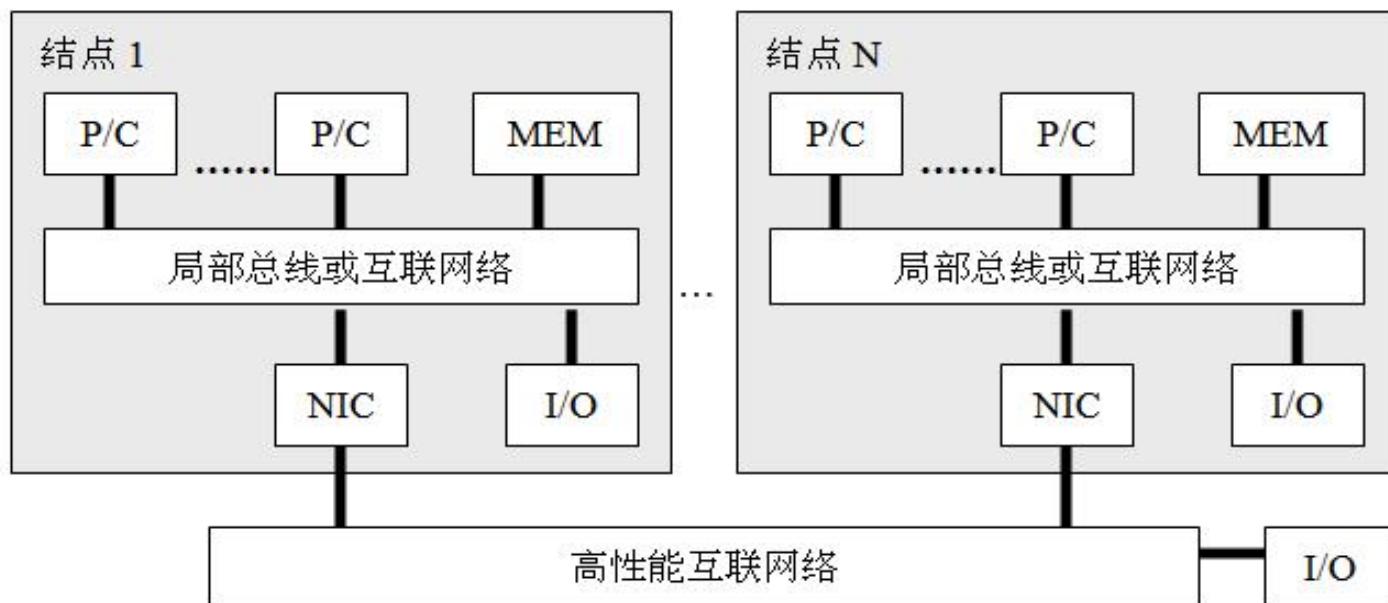
- DSM (Distributed Shared Memory) 分布式共享存储
  - 单一的内存地址空间
  - 可扩展到上百个结点



# 并行计算发展

## ■ 90 年代：体系结构框架趋于统一

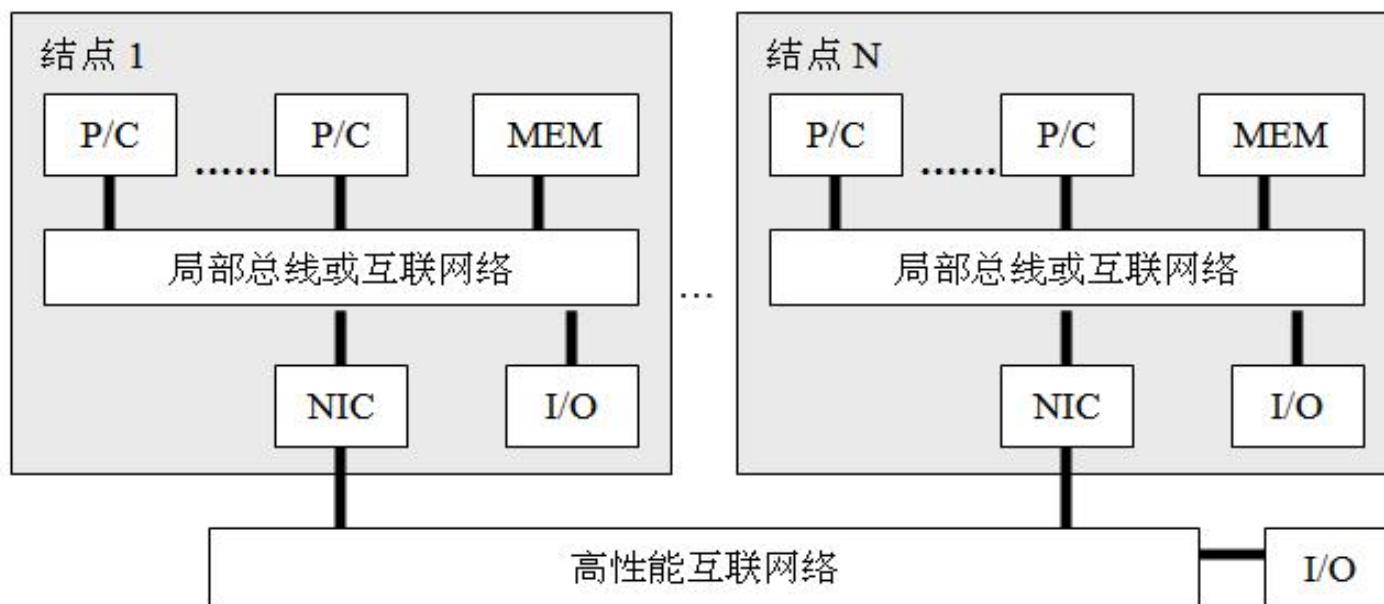
- MPP (Massively Parallel Processing) 大规模并行处理结构
  - 每个结点相对独立，有一个或多个微处理器
  - 每个结点均有自己的操作系统



# 并行计算发展

## ■ 90 年代：体系结构框架趋于统一

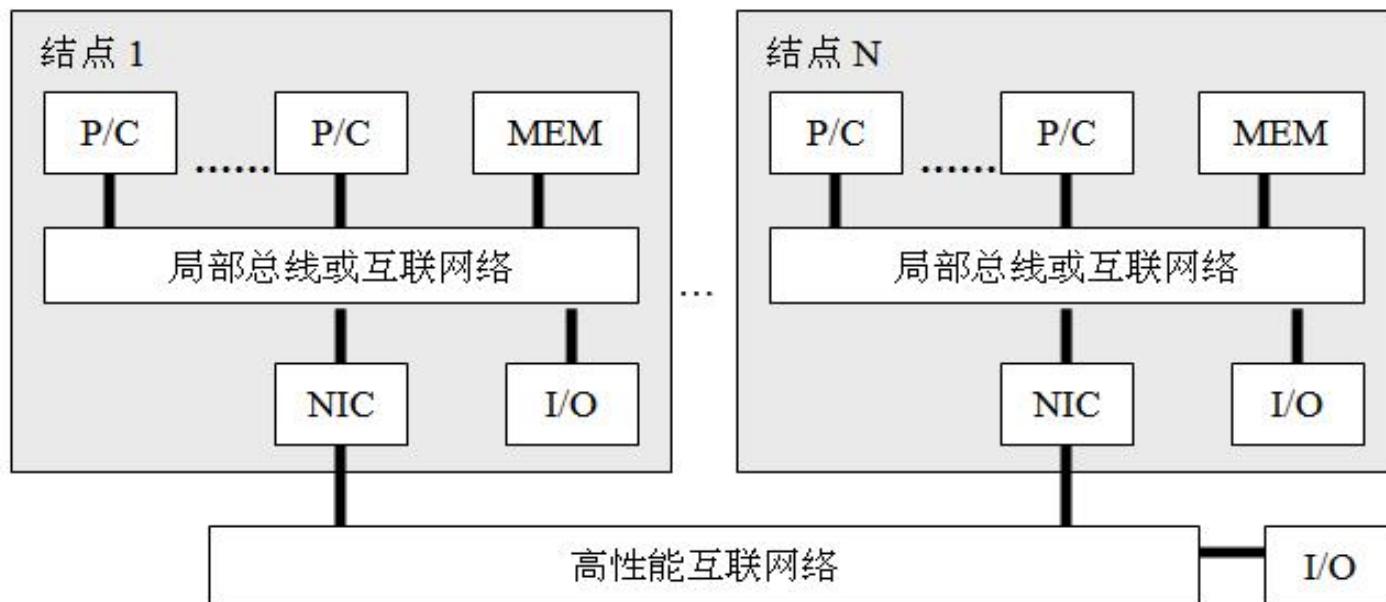
- MPP (Massively Parallel Processing) 大规模并行处理结构
  - 各个结点自己独立的内存，避免内存访问瓶颈
  - 各个结点只能访问自己的内存模块



# 并行计算发展

## ■ 90 年代：体系结构框架趋于统一

- MPP (Massively Parallel Processing) 大规模并行处理结构
- 扩展性较好



# 并行计算发展

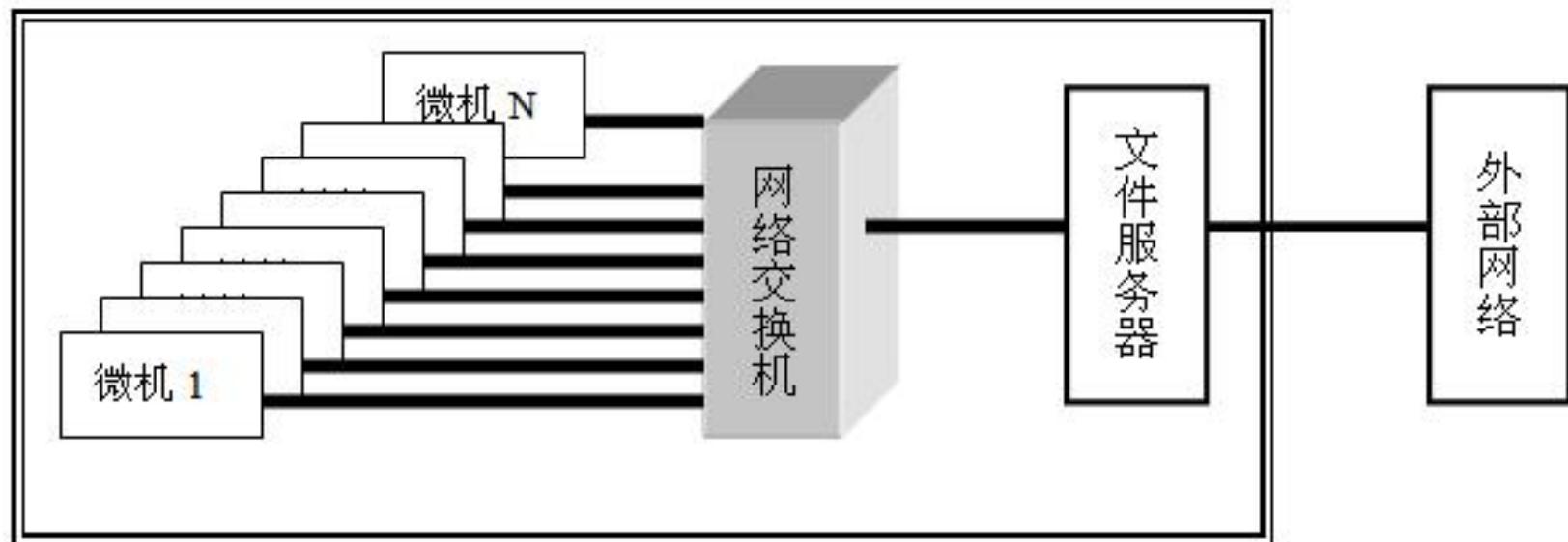
## ■ 90 年代：体系结构框架趋于统一

- NOW (Network of Workstations) 工作站机群
  - 每个结点都是一个完整的工作站，有独立的硬盘与UNIX系统
  - 结点间通过低成本的网络（如千兆以太网）连接
  - 每个结点安装消息传递并行程序设计软件，实现通信、负载平衡等
  - 投资风险小、结构灵活、可扩展性强、通用性好、异构能力强，被大量中小型计算用户和科研院校所采用
- 也称为 COW (Cluster of Workstations)
- NOW (COW) 与 MPP 之间的界线越来越模糊

# 并行计算发展

## ■ 90 年代：体系结构框架趋于统一

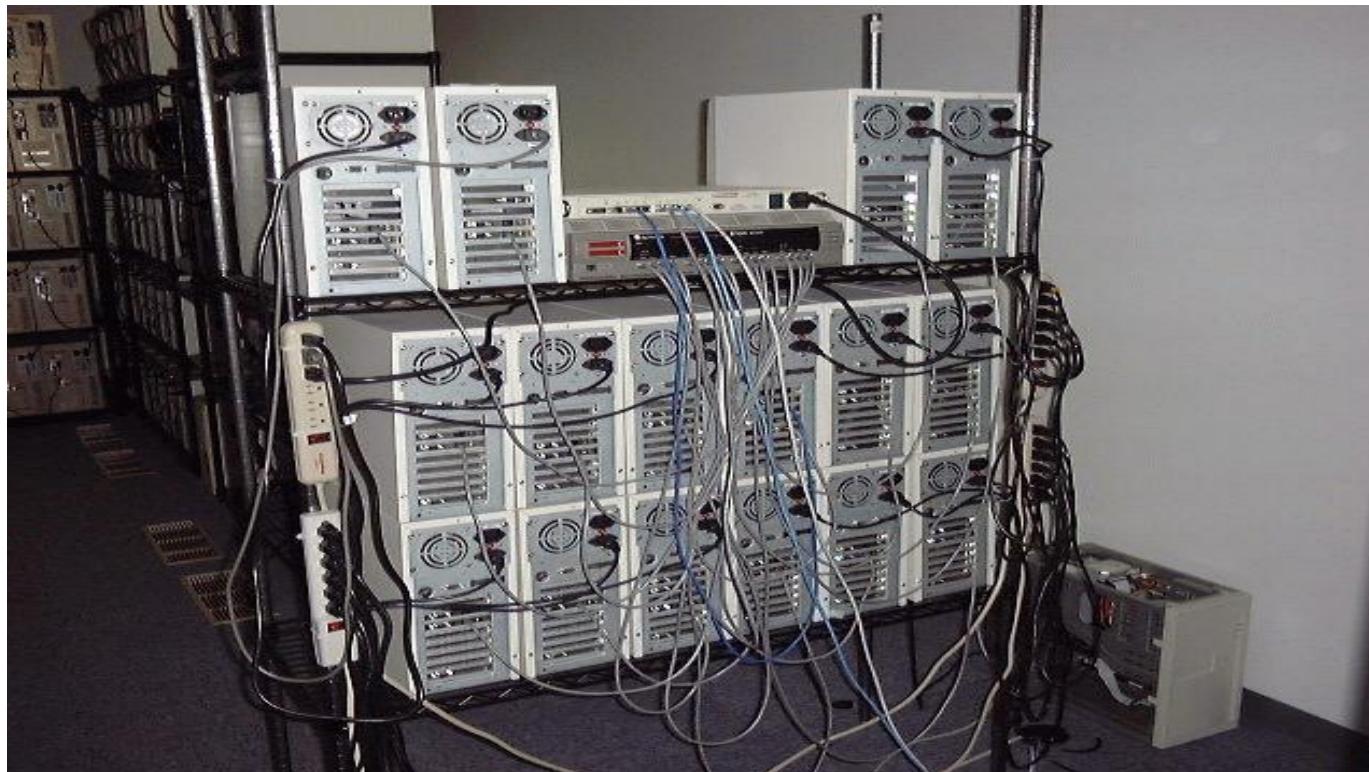
- NOW (Network of Workstations) 工作站机群
- NOW的典型代表：Beowulf cluster 微机机群
- 性能价格比极高



# 并行计算发展

## ■ 90 年代：体系结构框架趋于统一

- NOW (Network of Workstations) 工作站机群
- 第一台 Beowulf 机群



# 并行计算发展

## ■ 2000 年至今：前所未有的大踏步发展

### ➤ Cluster 机群

- 每个结点含多个商用处理器，结点内部共享存储
- 采用商用机群交换机通过总线连接结点，结点分布存储
- 各结点采用Linux操作系统、GNU编译系统和作业管理系统

### ➤ Constellation 星群

- 每个结点是一台子并行机
- 采用商用机群交换机通过总线连接结点，结点分布存储
- 各个结点运行专用的操作系统、编译系统和作业管理系统

### ➤ MPP

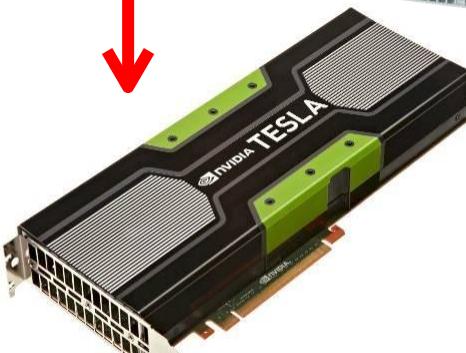
- 专用高性能网络，大多为政府直接支持

# 并行计算发展

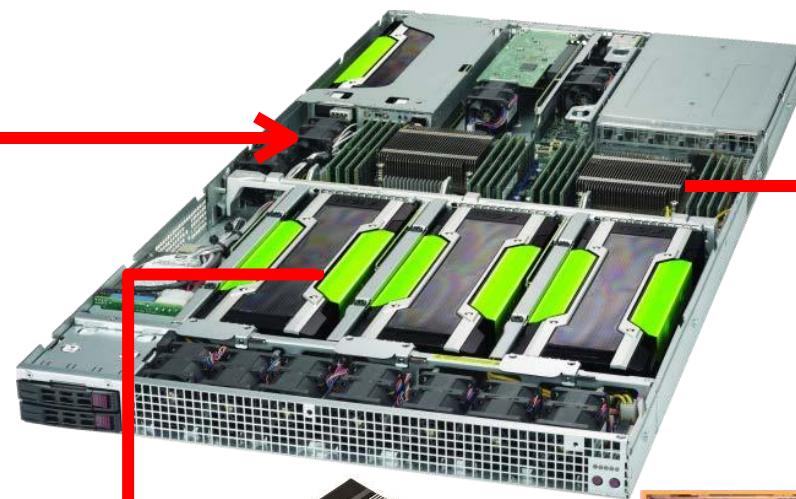
## ■ 目前典型的并行系统组成（引入异构架构）



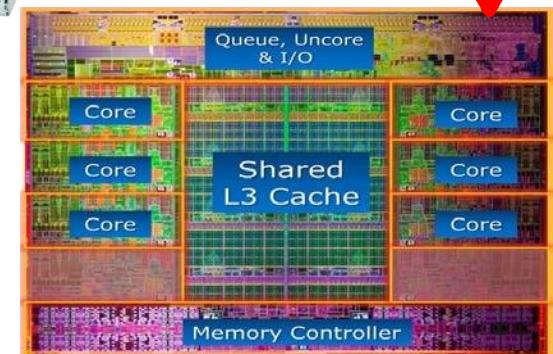
Racks: 16~42U



Co-Processor:  
100x cores/1000x threads



Node/Server: 1~4U



Multi-core: 4~12 cores

# 并行计算发展

## ■ 并行计算发展水平的标志：超级计算机

- 体量巨大、造价高昂的设备，拥有数以万计的处理器，旨在执行专业性强、计算密集型的任务
- 它的性能是以每秒浮点运算（FLOPS）来衡量的，而不是以每秒百万条指令（MIPS）来衡量的
- Top500排名（1993年开始），每年发布两次

### 前沿(Frontier)

- ① Top500排名第一， $1.194 \text{ EFlop/s}$  ( $1\text{E}=10^{18}$ )
- ② 9402个AMD EPYC CPU
- ③ 437608个AMD MI250X GPU
- ④ 37.608PB总内存，700PB存储



美国前沿（Frontier）超级计算机

# 并行计算发展

## ■ 并行计算发展水平的标志：超级计算机

- 最新的硬件技术
- 定制的系统配置
- 优化的软件和函数库
- 巨大的资金投入和能耗成本

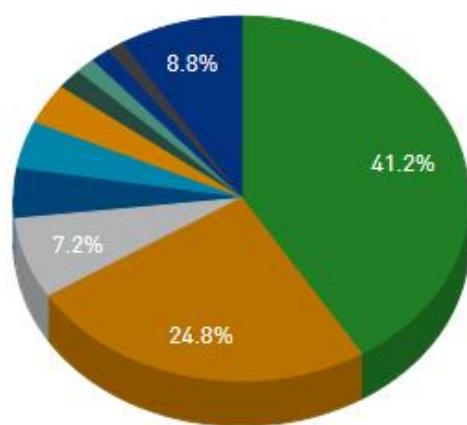
科技军备竞赛：  
国家高新技术  
发展水平的重  
要标志



# 并行计算发展

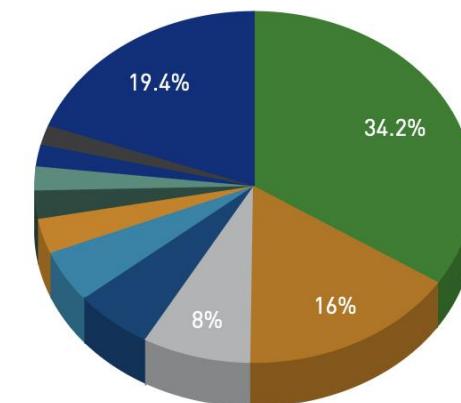
- Top500趋势：国家对比
  - 美国超算数量第一，中国超算数量第二

Countries System Share



2018

Countries System Share



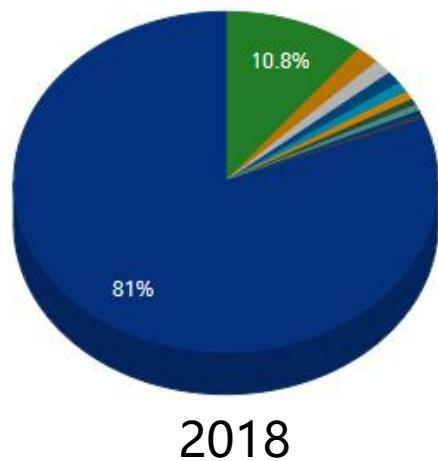
2024

# 并行计算发展

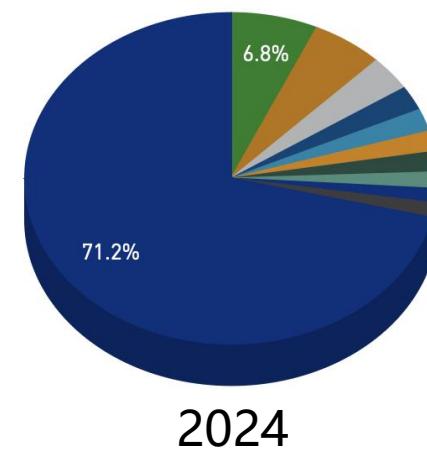
## ■ Top500趋势：算力架构对比

- 伴随AI等应用的强势崛起，异构算力成最火爆的概念
- NVIDIA GPU、Intel协处理器、国防科大Matrix-2000、曙光DCU

Accelerator/Co-Processor System Share



Accelerator/Co-Processor System Share

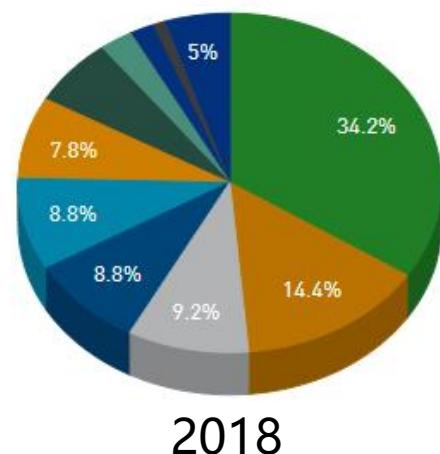


# 并行计算发展

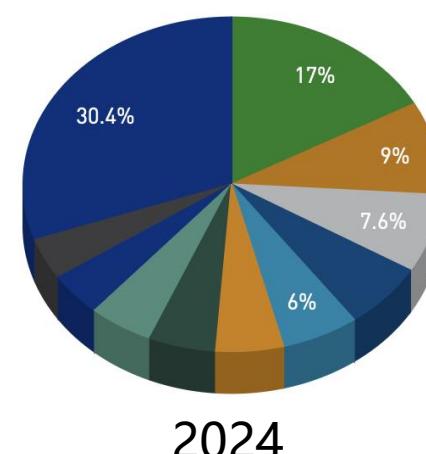
## ■ Top500趋势：互联网络对比

- 以太网交换机和网卡包含RDMA、智能网络编排
- 以太网相对于Infiniband和各类定制网络，性价比更高

Interconnect System Share



Interconnect System Share

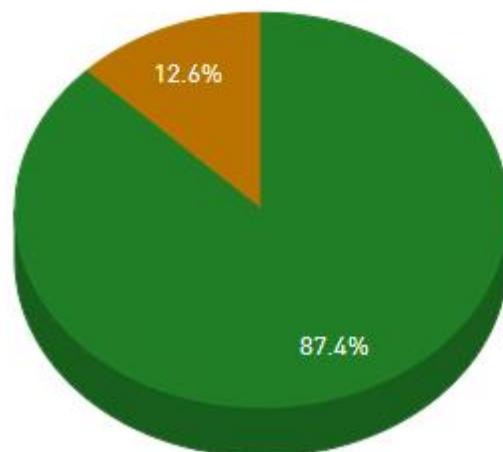


# 并行计算发展

## ■ Top500趋势：超算架构对比

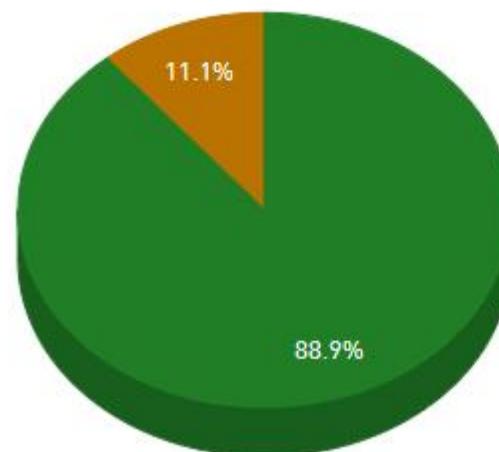
- Cluster架构占据主导地位
- 功能和架构限制之下，MPP架构超算占比逐渐降低

Architecture System Share



2018

Architecture System Share



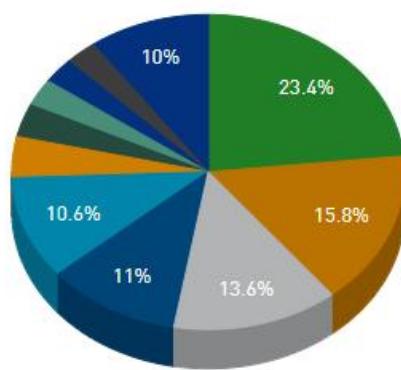
2023

# 并行计算发展

## ■ Top500趋势：制造商对比

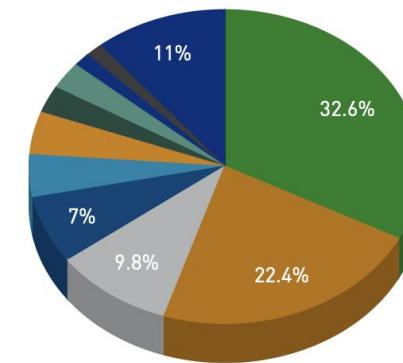
- 承接来自国家、科研机构和顶尖企业的需求
- 榜单中超算制造商，中国品牌市场份额快速提升

Vendors System Share



2018

Vendors System Share



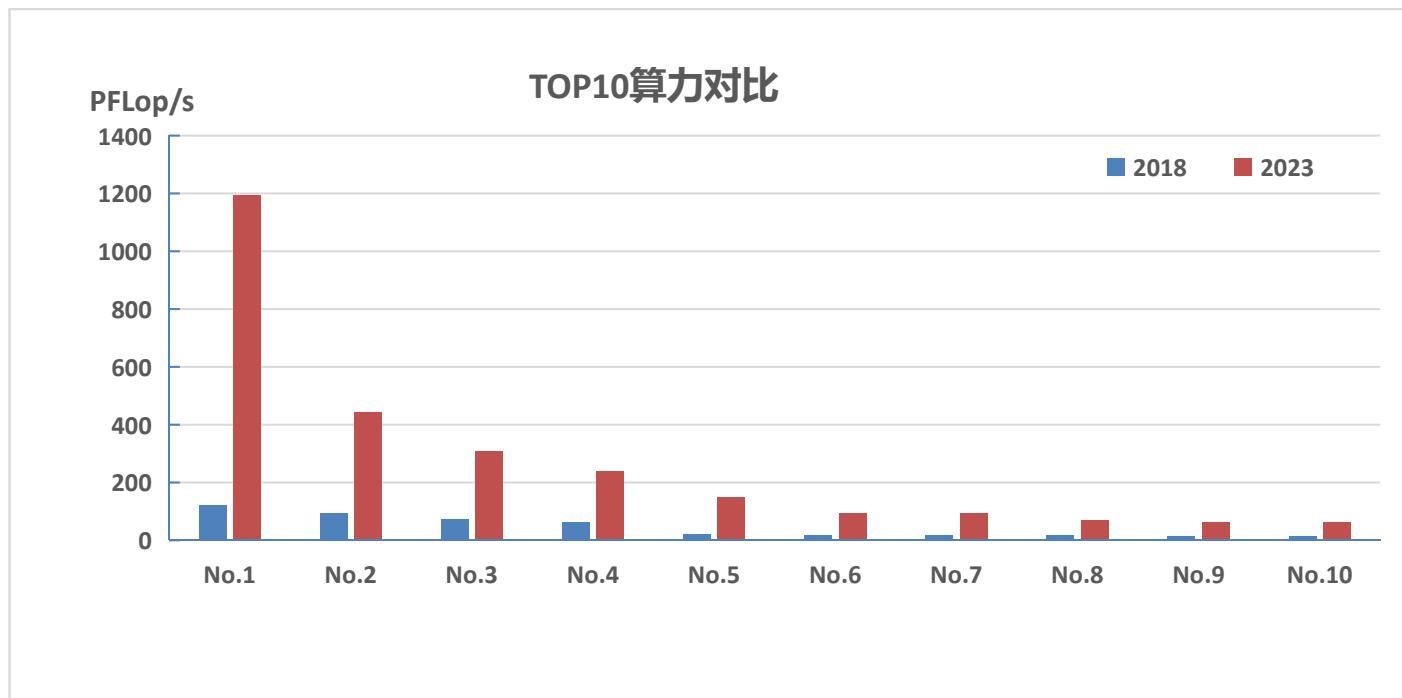
2024



# 并行计算发展

## ■ Top500趋势：百亿亿次 (EFlop)

- 5年跨度的两张TOP10榜单对比，4-10倍的算力增长
- E级计算时代已经到来



# 并行计算发展

## ■ 全球各大经济体，E级超算计划

**天河三号**：采用Matrix 3000片上异构众核处理器，已投入使用，FP64算力>1EFLop/s

**神威E级**：采用SW39000片上异构众核处理器，已投入使用，FP64算力>1EFLop/s

**曙光E级**：采用x86架构的海光处理器和曙光DCU

**JUPITER**：欧盟的超算机构EuroHPC JU计划在德国Jülich建造基于GPU的E级超算，还在招标中

欧洲第二台E级超算(还未命名)计划部署在法国

**Post-K (后“京”)**：作为日本超算“京”的后续产品，Post-K将采用目前已经成功部署的富士通A64FX处理器。Post-K计算节点原型已经开发完成，I/O及计算节点有48个核心外加4个辅助核心

**Frontier**：由AMD和HPE CRAY共同研发制造，用户同样为美国能源部，已投入使用

**Aurora**：由Intel和HPE CRAY共同研发制造，用户为美国能源部阿贡实验室

**El Capitan**：由AMD和HPE CRAY共同研发制造，用户为美国能源部劳伦斯利弗莫尔实验

# 目录

- 什么是并行计算
- 为什么需要并行计算
- 并行计算发展
- 并行计算面临的挑战

# 并行计算面临的挑战

## ■ 功耗 (Power)

- 超算耗电功率巨大, 如何降低功率提高能效, 是算力增长的主要挑战

10th  
**SuperPOD**  
121PF  
-- MW  
-- GF/W

8th  
**MareNostrum 5 ACC**  
175PF  
4.2MW  
41.67GF/W

6th  
**Alps**  
270PF  
5.2MW  
51.92GF/W

4th  
**Fugaku**  
442PF  
29.9MW  
14.78GF/W

2th  
**Aurora**  
1,012PF  
38.7MW  
26.15GF/W

Exascale

9th  
**Summit**  
148PF  
10.1MW  
14.72GF/W

7th  
**Leonardo**  
175PF  
7.5MW  
23.33GF/W

5th  
**LUMI**  
379PF  
7.1MW  
53.38GF/W

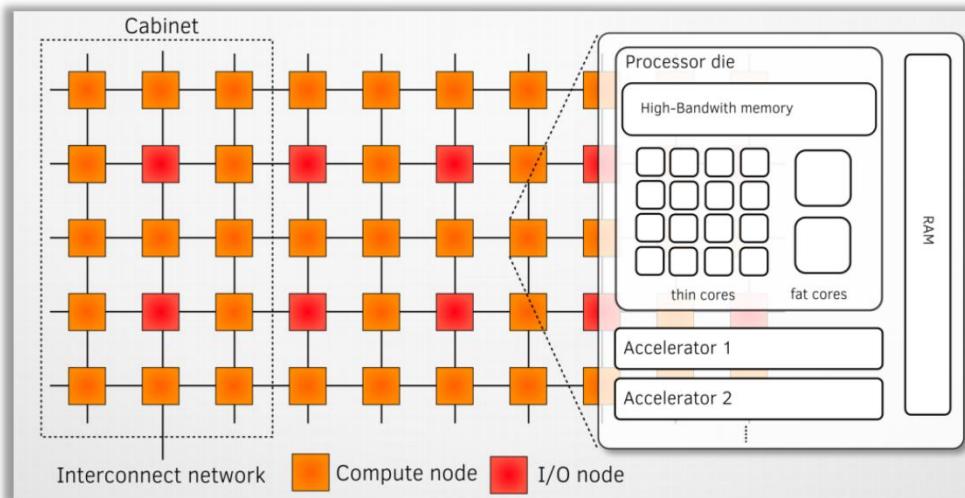
3th  
**Eagle**  
561PF  
-- MW  
-- GF/W

1th  
**Frontier**  
1,206PF  
22.7MW  
53.13GF/W

# 并行计算面临的挑战

## ■ 应用性能 (Performance)

- 追求应用可获得的性能而不是峰值性能
- 实际应用性能经常在10%甚至5%的峰值之下



**节点间并行**: Complex software stack required to handle huge traffic and failure (lossless compression, resilience, huge pages, RMA...)

**节点内并行**: Heterogeneous computing with different accelerators, memory space and bandwidth

For i from 1 to N :  
If (condition on i) :  
 $d[i] = a[i] + b[i]*c[i]$

mask	1	1	0	1	0	1	0	1	
a[i]	a[i+7]	a[i+6]	a[i+5]	a[i+4]	a[i+3]	a[i+2]	a[i+1]	a[i]	
+									
b[i]	b[i+7]	b[i+6]	b[i+5]	b[i+4]	b[i+3]	b[i+2]	b[i+1]	b[i]	
X									
c[i]	c[i+7]	c[i+6]	c[i+5]	c[i+4]	c[i+3]	c[i+2]	c[i+1]	c[i]	
=									
d[i]	d	c[i+7]	c[i+6]	c[i+5]	c[i+4]	c[i+3]	c[i+2]	c[i+1]	c[i]

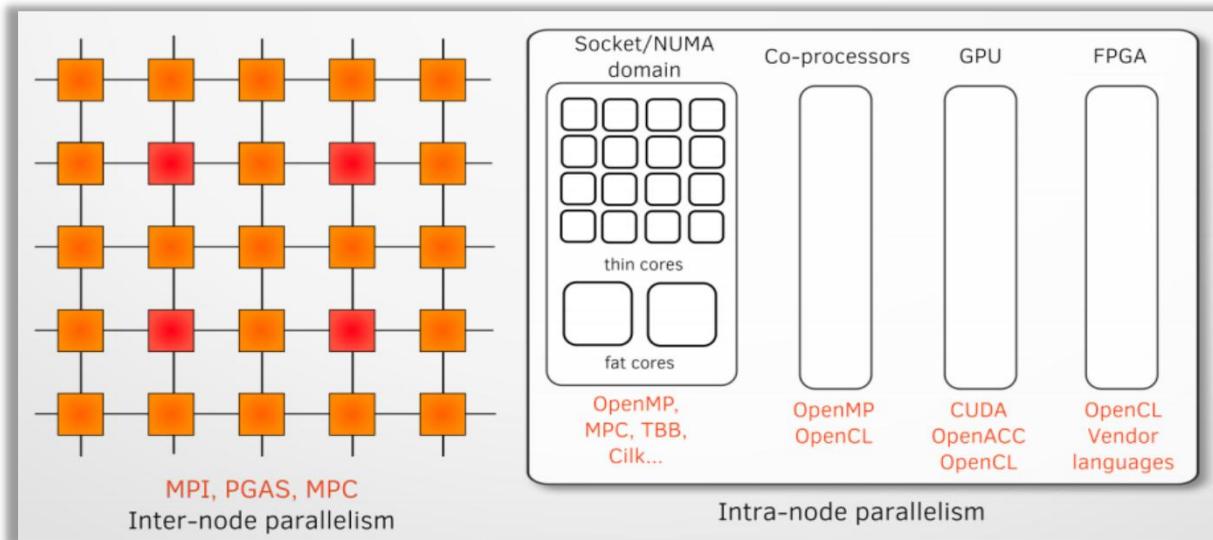
Single Instruction Multiple Data SIMD vision as in x86 Intel

**向量化**: Capability to compute simultaneously a full vector of data with the same set of operations

# 并行计算面临的挑战

## ■ 可编程性 (Programmability)

- 大规模并行和异构体系结构给并行编程带来巨大困难
- 并行程序编程难，调试难，性能不确定



- ①、MPI + X  
(OpenMP, Open ACC...)
- ②、Limit data movements, avoid global communications, use shared memory within a node

- ①、Use non-blocking communication with computation
- ②、Dedicate cores to the communications, diagnostic processing, I/O
- ③、Use accelerator in symmetric mode to exploit the full node computational power

# 并行计算面临的挑战

## ■ 可靠性 (Resilience)

- 巨大的系统规模使得系统的平均无故障时间大大缩短，甚至在1小时以下
- 如何完成长时间不间断运行的应用？

卡内基梅隆大学根据美国洛斯阿拉莫斯国家实验室 (Los Alamos National Labs, LANL) 22 台超级计算机长达 9 年的故障数进行统计

Category	Hardware	Software	Network	Environment	Human	Unknown
Failure Percentage	30-60%	5-24%	<3%	<3%	<3%	20-30%
Downtime Percentage	40-80%	3-25%	<2%	<5%	<3%	<10%

硬件故障和软件故障比例较大，由网络、环境因素和人为因素故障比例较小  
由于超级计算机本身结构复杂，故障成因较多，同时故障类别本身鉴别起来较为困难，因此，有相当一部分故障没有给出明确故障原因

# Reference

- Parallel Programming course slides from Prof. Jerry Chou, National Tsing Hua University. [http://lms.nthu.edu.tw/sys/read\\_attach.php?id=1530893](http://lms.nthu.edu.tw/sys/read_attach.php?id=1530893)
- TOP500: <https://www.top500.org/>
- Blaise Barney, Lawrence Livermore National Laboratory, Introduction to Parallel Computing, [https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)



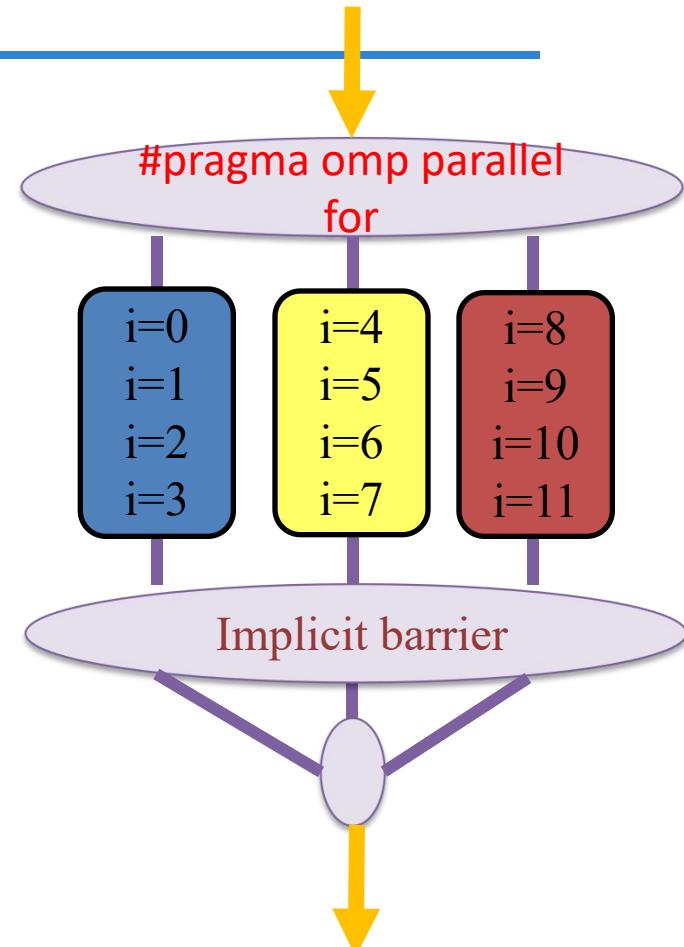
# 并行编程实践

## ■ OpenMP示例

```
for (int i = 0; i < 12; i++)  
    c[i] = a[i] + b[i];
```



```
#pragma omp parallel for  
for (int i = 0; i < 12; i++)  
    c[i] = a[i] + b[i];
```





# 并行编程实践

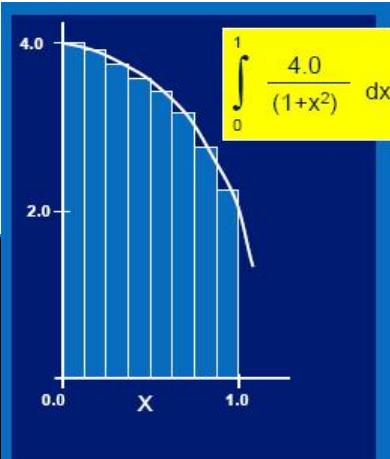
## ■ OpenMP示例

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

static long num_steps=200000000;
double step, pi;

void main()
{
    clock_t start, stop;
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    start = clock();
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    stop = clock();
    printf("Pi = %15.12f\n",pi);
    printf("The time to calculate PI was %f seconds\n",((double)(stop - start)/CLOCKS_PER_SEC));
}
```



```
[zqybegin@ecs-zqy openmp]$ vim serial_pi.c
[zqybegin@ecs-zqy openmp]$ gcc serial_pi.c -o serial_pi.bin
[zqybegin@ecs-zqy openmp]$ ./serial_pi.bin
The value of PI is 3.141592653590
The time to calculate PI was 7.006011 seconds
[zqybegin@ecs-zqy openmp]$ ./openmp_pi.bin
The value of PI is 3.141592653590
The time to calculate PI was 1.768647 seconds
```

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

static long num_steps=200000000;
double step, pi;

void main()
{
    double start, stop;
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    start = omp_get_wtime();
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    stop = omp_get_wtime();
    printf("Pi = %15.12f\n",pi);
    printf("The time to calculate PI was %f seconds\n",((double)(stop - start)));
}
```

```
gcc serial_pi.c -o serial_pi.bin
```

```
gcc -fopenmp parallel_pi.c -o
openmp_pi.bin
```



# 并行编程实践

## ■ MPI示例

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <time.h>
4 #include <mpi.h>
5
6 int prime_part ( int id, int p, int n );
7
8 int main ( int argc, char *argv[] )
9 {
10     MPI_Init(&argc, &argv);
11     int id, p, n;
12     MPI_Comm_rank(MPI_COMM_WORLD, &id);
13     MPI_Comm_size(MPI_COMM_WORLD, &p);
14     if(id == 0)
15     {
16         printf("Input N Value: ");
17         fflush(stdout);
18         scanf("%d", &n);
19     }
20     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
21
22     int total = 0;
23     int total_part = 0;
24
25     double time1 = MPI_Wtime();
26     total_part = prime_part(id, p, n);
27     MPI_Reduce(&total_part, &total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
28     double time2 = MPI_Wtime();
29     if(id == 0)
30     {
31         printf ( "Between 2 and %d, there are %d primes\n", n, total );
32         printf ( "Computation Time %.10lf\n", time2 - time1 );
33     }
34     MPI_Finalize();
35
36     return 0;
37 }
```

```
38 int prime_part(int id, int p, int n)
39 {
40     int i;
41     int j;
42     int prime;
43     int total_part;
44
45     total_part = 0;
46
47     for ( i = id * 2 + 1; i <= n; i += p*2 )
48     {
49         prime = 1;
50
51         for ( j = 2; j < i; j++ )
52         {
53             if ( i % j == 0 )
54             {
55                 prime = 0;
56                 break;
57             }
58         }
59         if ( prime )
60         {
61             total_part = total_part + 1;
62         }
63     }
64
65     return total_part;
66 }
```

```
mpicc prime-mpi.c -o prime-mpi.bin
```

```
mpirun -np 6 ./prime-mpi.bin
```



# 并行编程实践

## MPI示例

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <time.h>
4 #include <mpi.h>
5
6 int prime_part ( int id, int p, int
7
8 int main ( int argc, char *argv[] ) {
9 {
10     MPI_Init(&argc, &argv);
11     int id, p, n;
12     MPI_Comm_rank(MPI_COMM_WORLD,
13     MPI_Comm_size(MPI_COMM_WORLD));
14     if(id == 0)
15     {
16         printf("Input N Value\n");
17         fflush(stdout);
18         scanf("%d", &n);
19     }
20     MPI_Bcast(&n, 1, MPI_INT, 0,
21
22     int total = 0;
23     int total_part = 0;
24
25     double time1 = MPI_Wtime();
26     total_part = prime_part(id,
27     MPI_Reduce(&total_part, &tot
28     double time2 = MPI_Wtime();
29     if(id == 0)
30     {
31         printf( "Between 2
32         printf( "Computatio
33     }
34     MPI_Finalize();
35
36     return 0;
37 }
```

运行时间 (T/s) 统计表

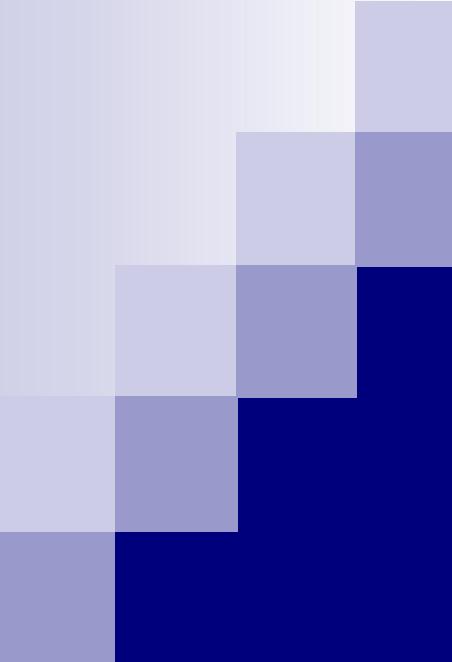
进程数 N \	1	2	4	6
100000	1.053	0.528	0.270	0.179
200000	3.959	1.996	1.008	0.678
400000	14.951	7.504	3.778	2.534
800000	56.615	28.434	14.268	9.661

加速比 ( $S_p$ ) 统计表

进程数 N \	1	2	4	6
100000	1.00	1.99	3.90	5.88
200000	1.00	1.98	3.93	5.84
400000	1.00	1.99	3.96	5.90
800000	1.00	1.99	3.97	5.86

prime-mpi.bin

mpirun -np 6 ./prime-mpi.bin



# 第二章

# 并行计算机体系结构

哈尔滨工业大学

张伟哲

2025, Fall Semester

# 目录

- 并行系统分类
- 共享内存系统
- 分布式内存系统
- 异构系统架构
- 互连网络

# 目录

- 并行系统分类
- 共享内存系统
- 分布式内存系统
- 异构系统架构
- 互连网络

# 并行计算机

- 由一组**处理单元**组成
- 各处理单元之间相互**通信与协作**
- 以**更快的速度**共同完成一项**大规模计算任务**

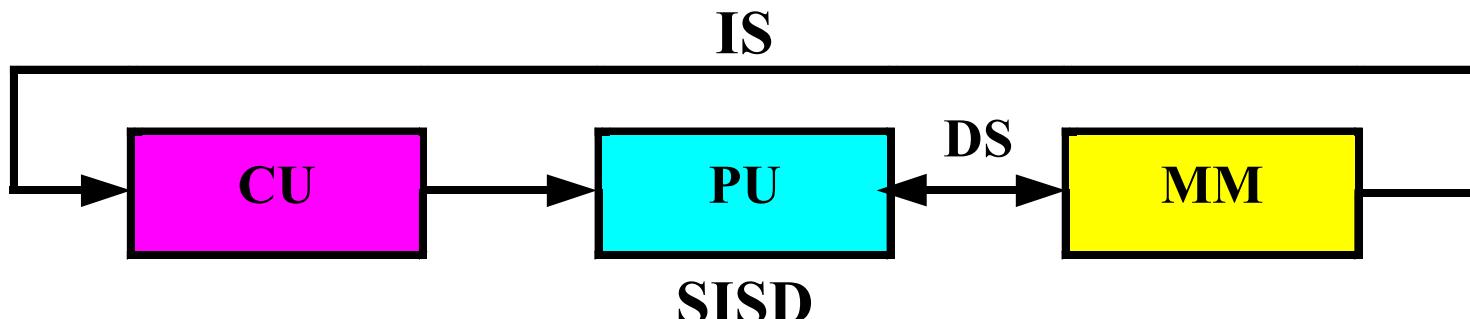


# Flynn分类法

<p><b>S I S D</b> Single Instruction, Single Data  串行计算机(von Neumann计算机)</p>	<p><b>S I M D</b> Single Instruction, Multiple Data  适用性很有限(如MPEG类计算、字符串匹配计算)</p>
<p><b>M I S D</b> Multiple Instruction, Single Data  为完美分类而设置，意义不大</p>	<p><b>M I M D</b> Multiple Instruction, Multiple Data  常见的并行计算机都可归入此类 MPP/Cluster/SMP/当前基于Cache的 Multi-core (Intel、AMD)</p>

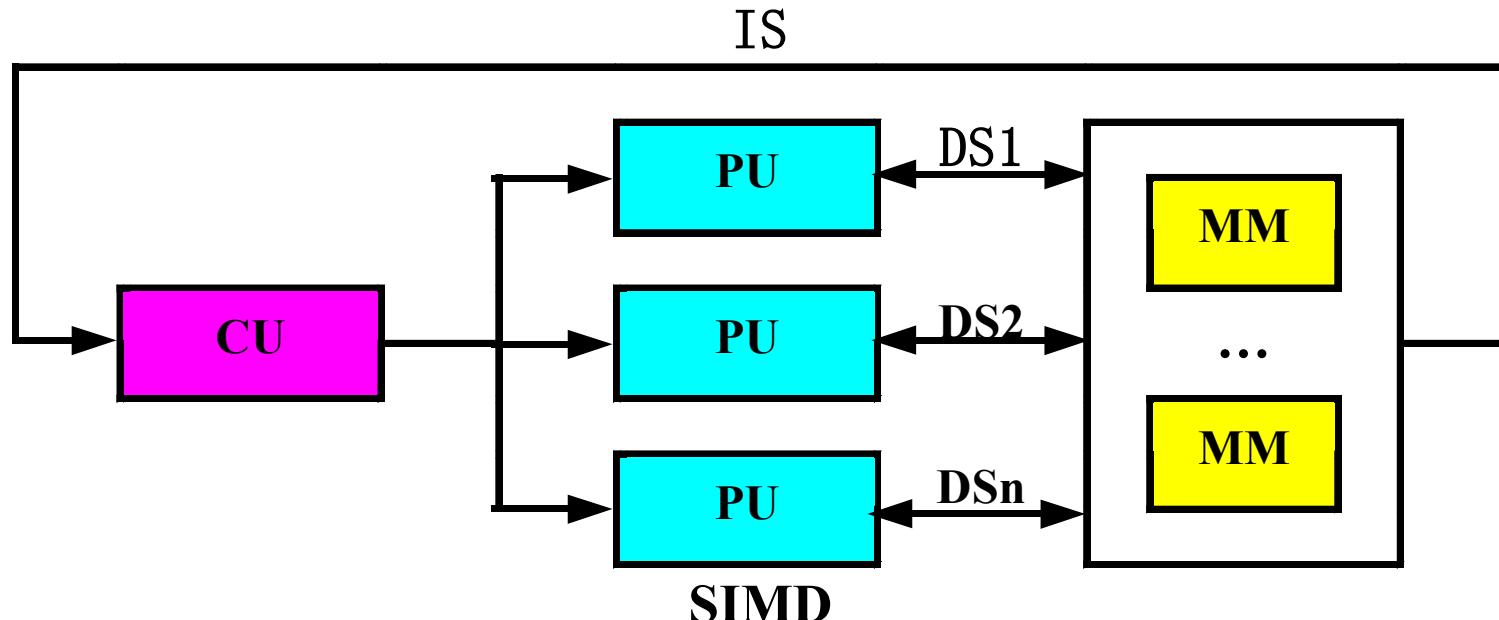
# Flynn分类法——SISD

- 处理器串行执行指令
- 或者处理器内采用指令流水线，以时间重叠技术实现了一定程度的指令并行执行
- 甚至处理器是超标量处理器，内有几条指令流水线实现了更大程度上的指令并行执行
- 都是以单一的指令流从存储器取指令，以单一的数据流从存储器取操作数和将结果写回存储器



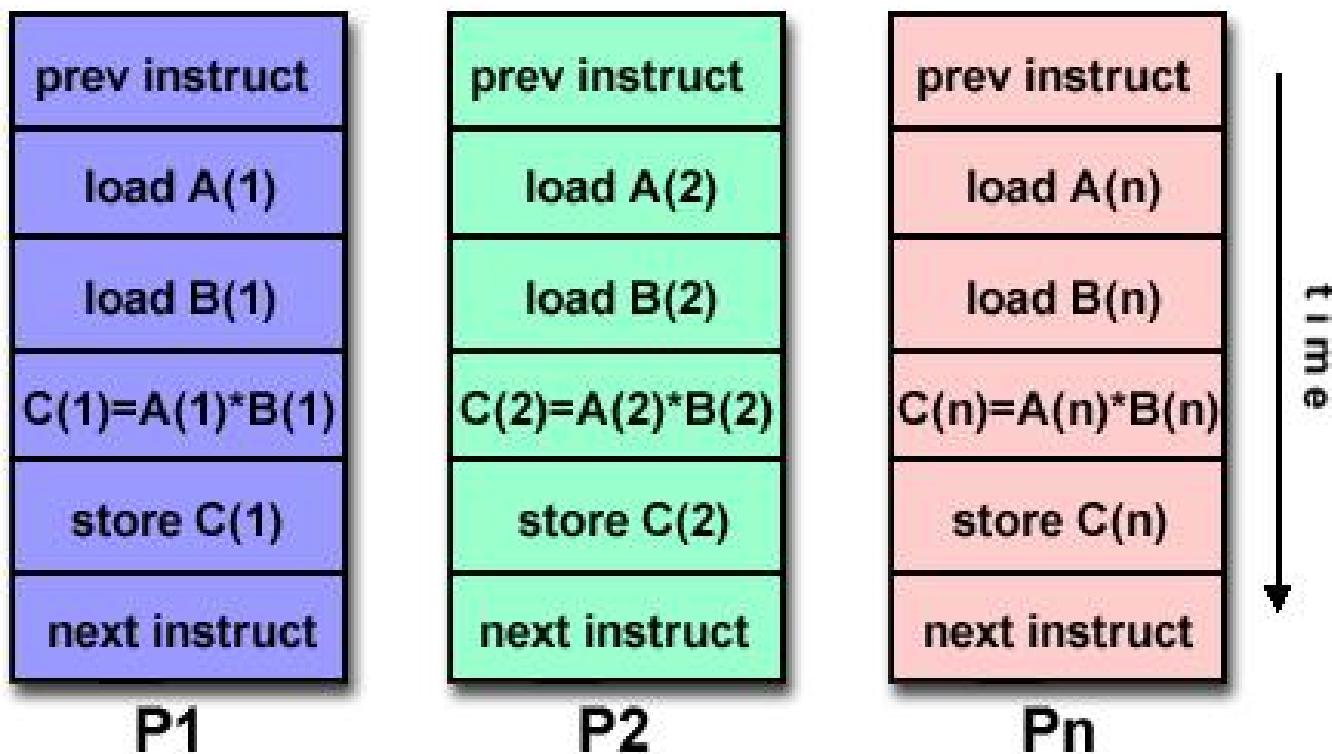
# Flynn分类法——SIMD

- 有单一的控制部件，但是有多个处理部件
- 计算机以一个控制单元从存储器取单一的指令流，一条指令同时作用到各个处理单元，控制各个处理单元对来自不同数据流的数据组进行操作



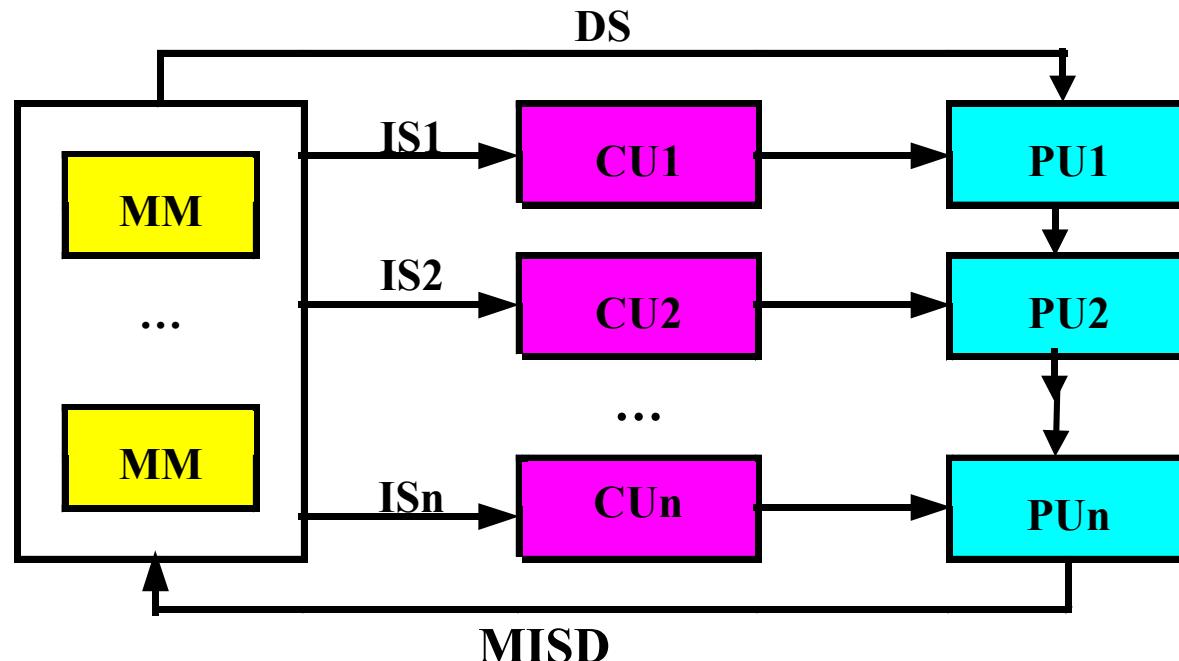
# Flynn分类法——SIMD

- 多用于**矢量运算中**，加速指令运算，比如矩阵乘
- 适用于**非常规则的计算**，例如：视频、音频处理的MPEG算法；密集矩阵的运算



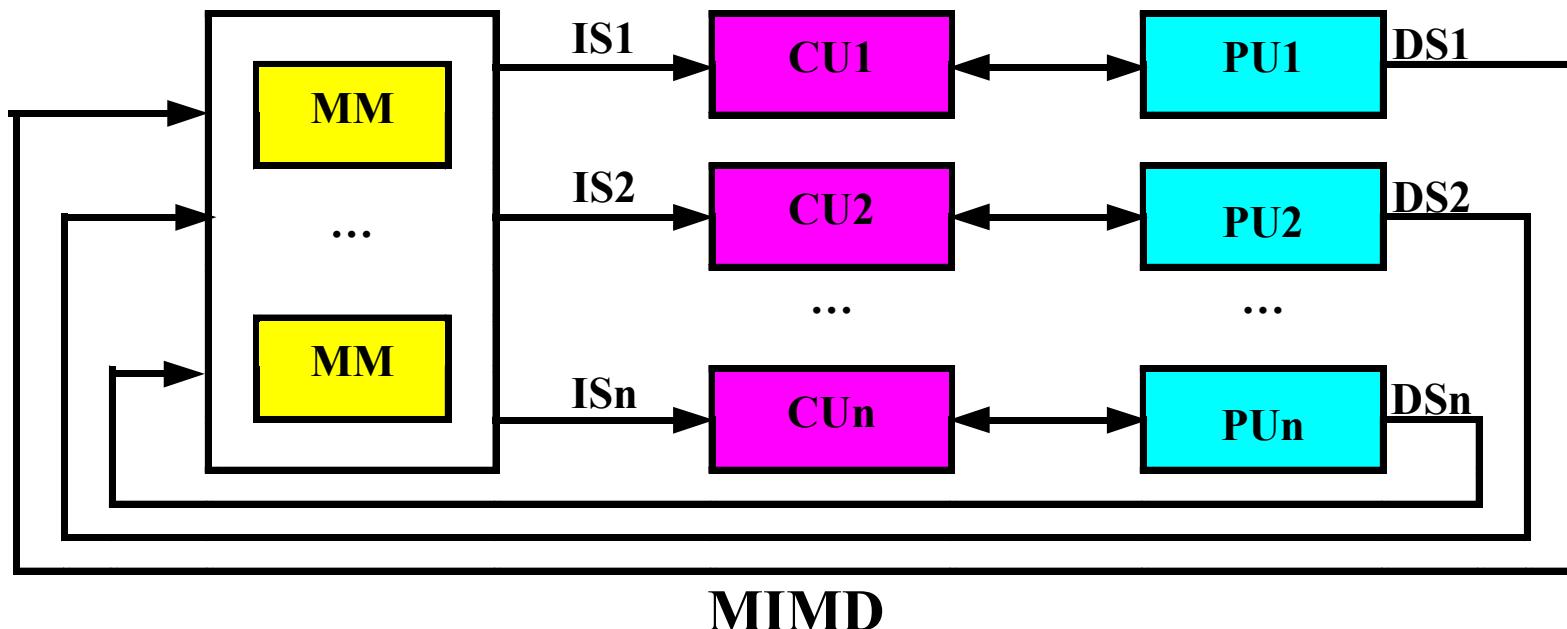
# Flynn分类法——MISD

- 多个处理单元，各配有相应的控制单元
- 各个处理单元接收不同的指令，多条指令同时在一份数据上进行操作
- 不常见的结构，通常用于容错



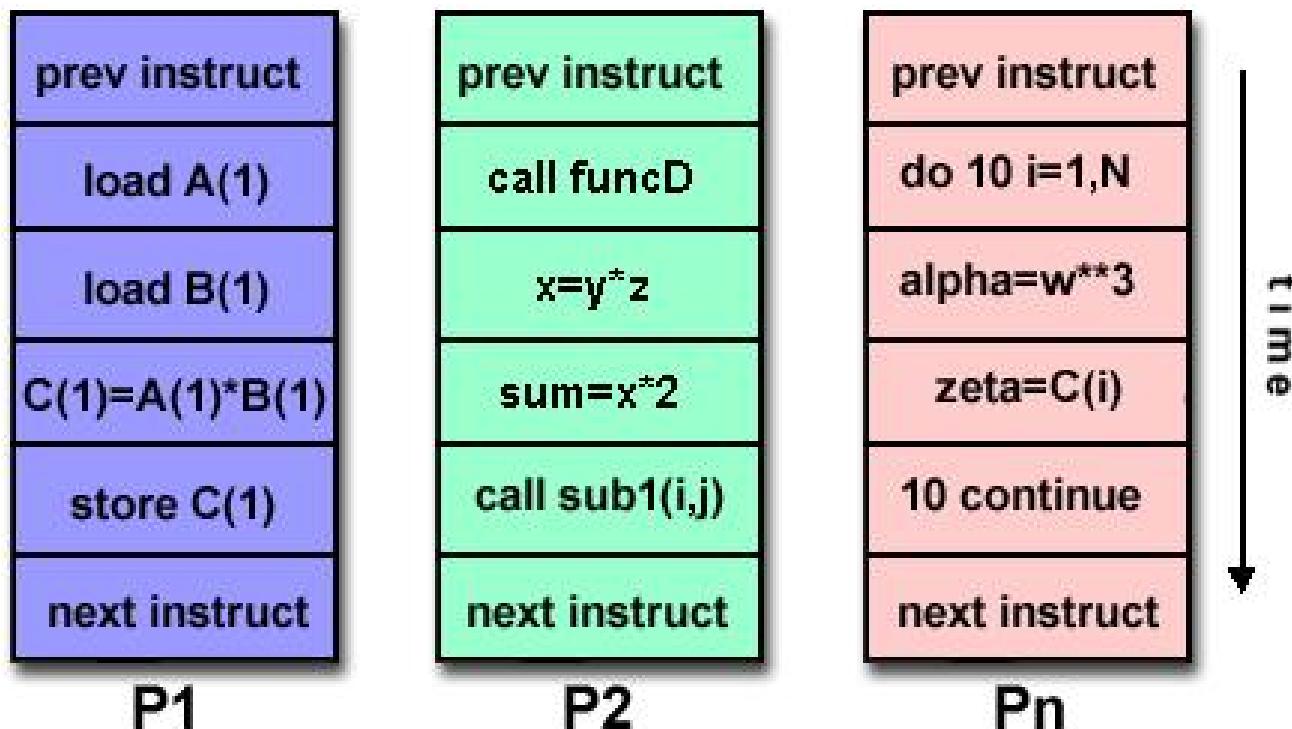
# Flynn分类法——MIMD

- 有多个处理单元，每个处理单元有相应控制单元
- 各个处理单元可以接收不同的指令并对不同的数据流进行操作

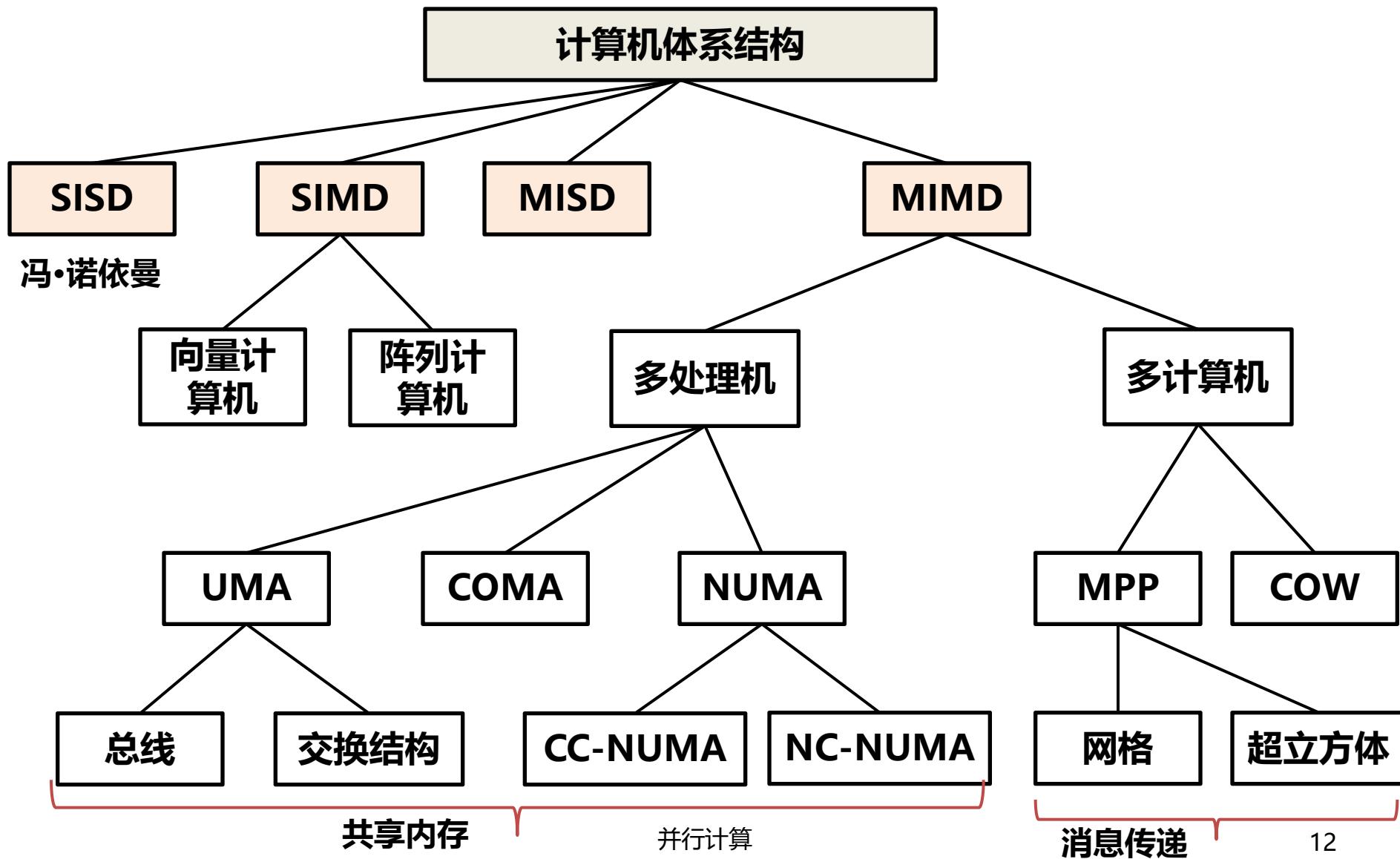


# Flynn分类法——MIMD

- 大多数现代并行计算机都属于这一类
- 对称多处理机 **SMP**, 大规模并行处理机 **MPP**,  
工作站机群 **COW**, 分布式共享存储系统 **DSW**



# 并行计算机体系结构图谱



# MIMD体系结构

## ■ 多处理机系统——基于共享内存

- 系统中只有唯一的地址空间，所有的处理器共享该地址空间
- 唯一的地址空间并不意味着在物理上只有一个存储器。共享地址空间可以通过一个物理上共享的存储器来实现，也可以通过分布式存储器并在硬件和软件的支持下实现

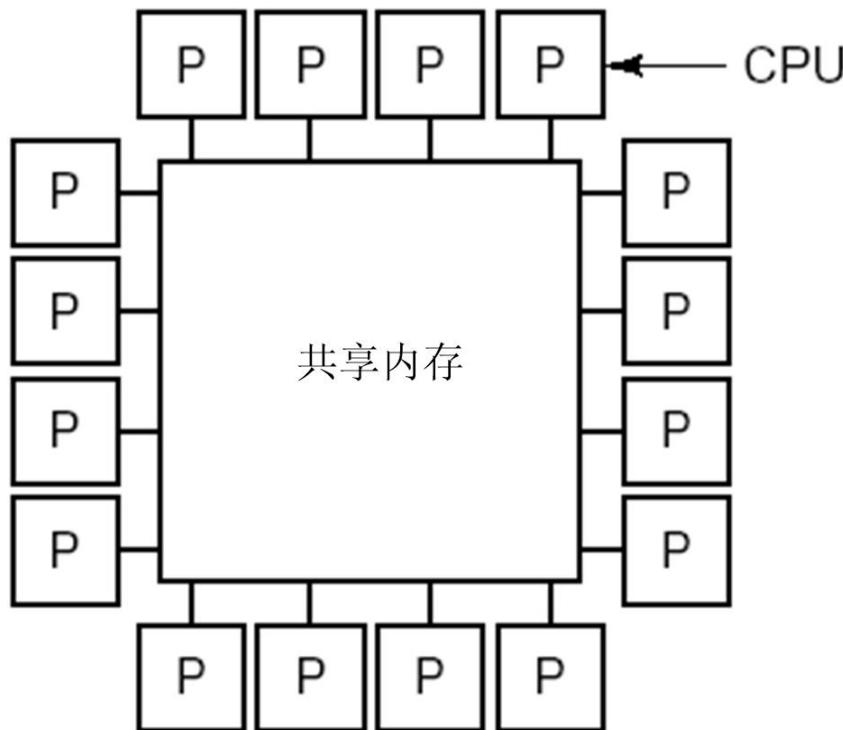
## ■ 多计算机系统——基于消息传递(分布式内存)

- 每个处理器有自己的存储器，该存储器只能被该处理器访问而不能被其他处理器直接访问，这种存储器称为局部存储器或私有存储器
- 当处理器A需要向处理器B传送数据时，A把数据以消息的形式发送给B

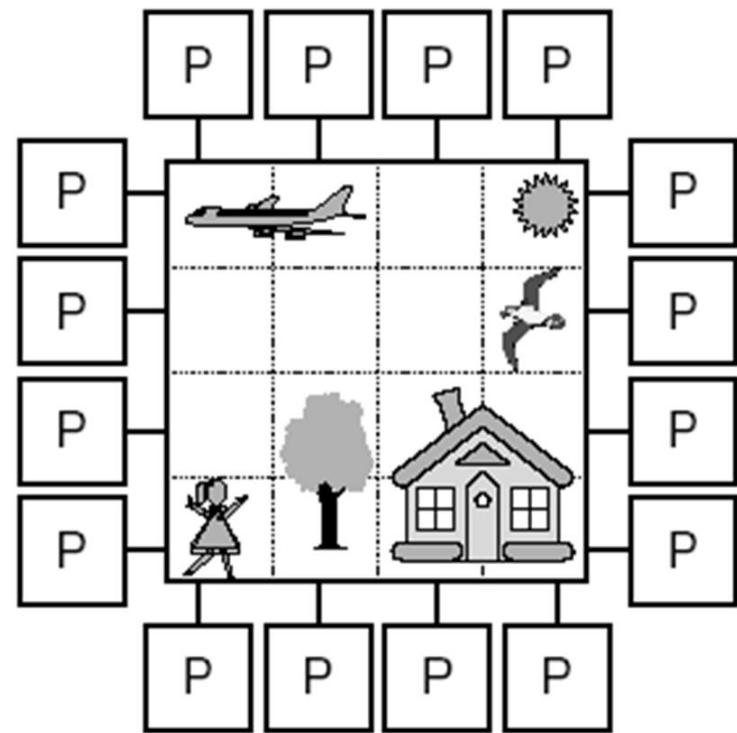
# 目录

- 并行系统分类
- **共享内存系统**
- 分布式内存系统
- 异构系统架构
- 互连网络

# 共享内存的多处理器



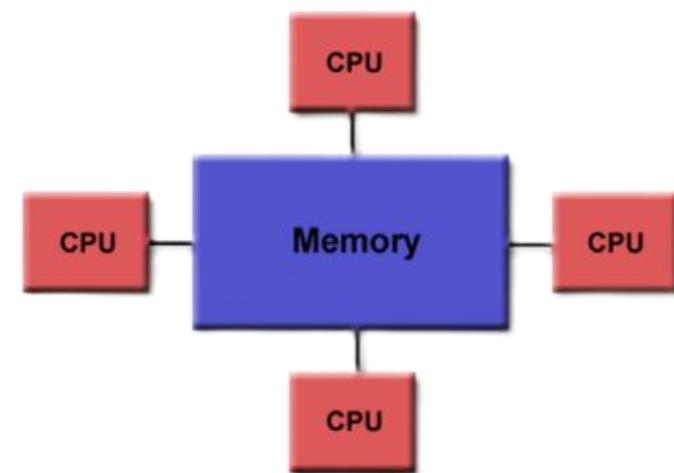
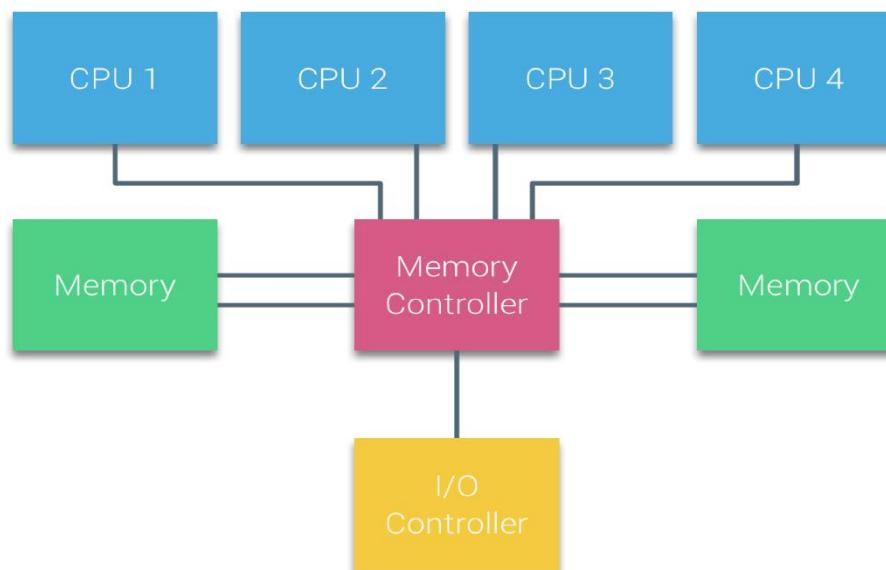
(a) 16个CPU共享一个公共内存的多处理器系统



(b) 一个图像分成16块，每块都由不同的CPU分析

# 共享内存系统——UMA

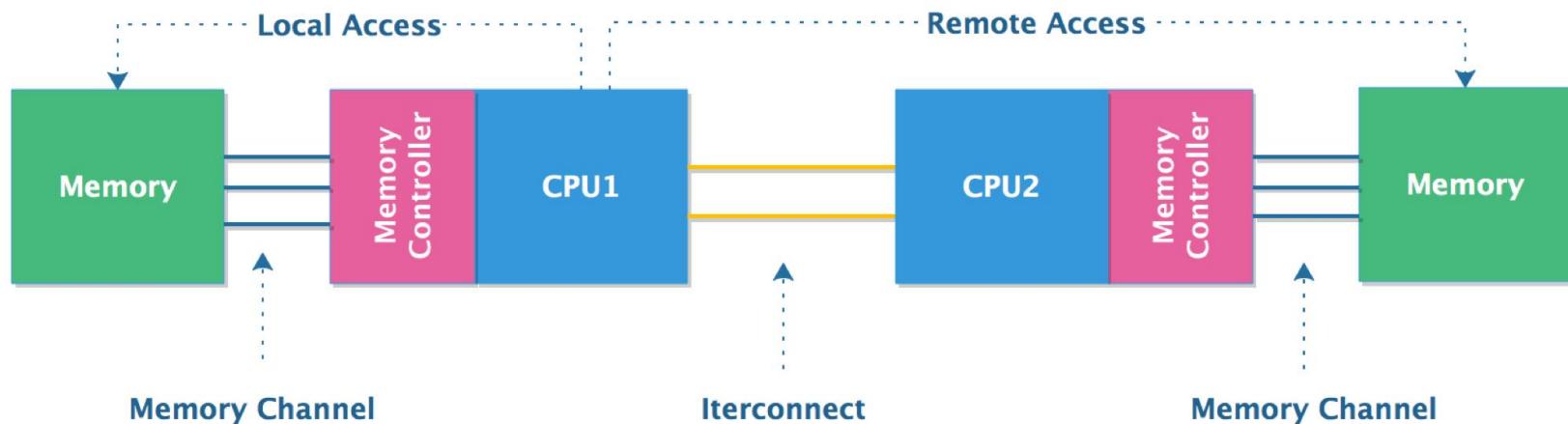
- UMA(Uniform Memory Access) 均匀存储器访问
  - 物理存储器被所有处理器均匀的共享
  - 所有处理器访问任何存储的时间相等
  - 每台处理器可带私有高速缓存
  - 外围设备可以一定形式共享



# 共享内存系统——NUMA

## ■ NUMA (Non-Uniform Memory Access) 非均匀存储器访问

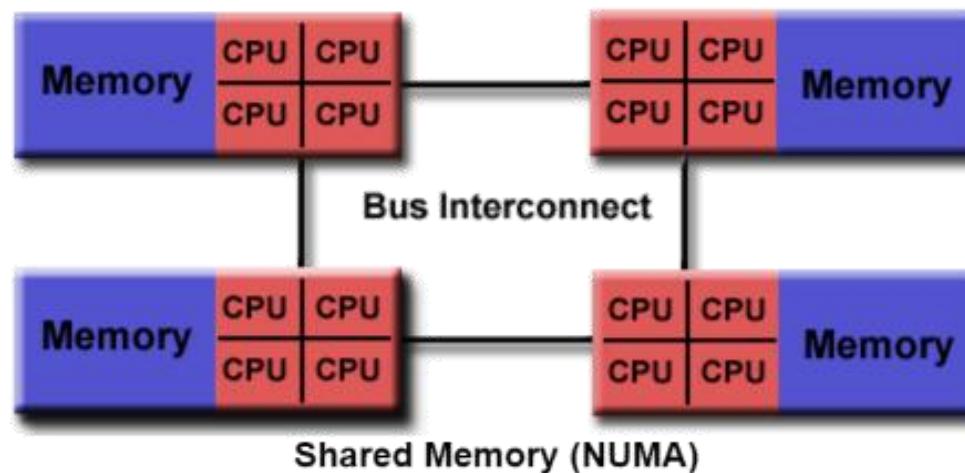
- 共享存储器分布在所有的处理器中
- 处理器访问存储器的时间不均匀
- 每台处理器可带私有高速缓存
- 外设可以一定形式共享



# 共享内存系统——NUMA

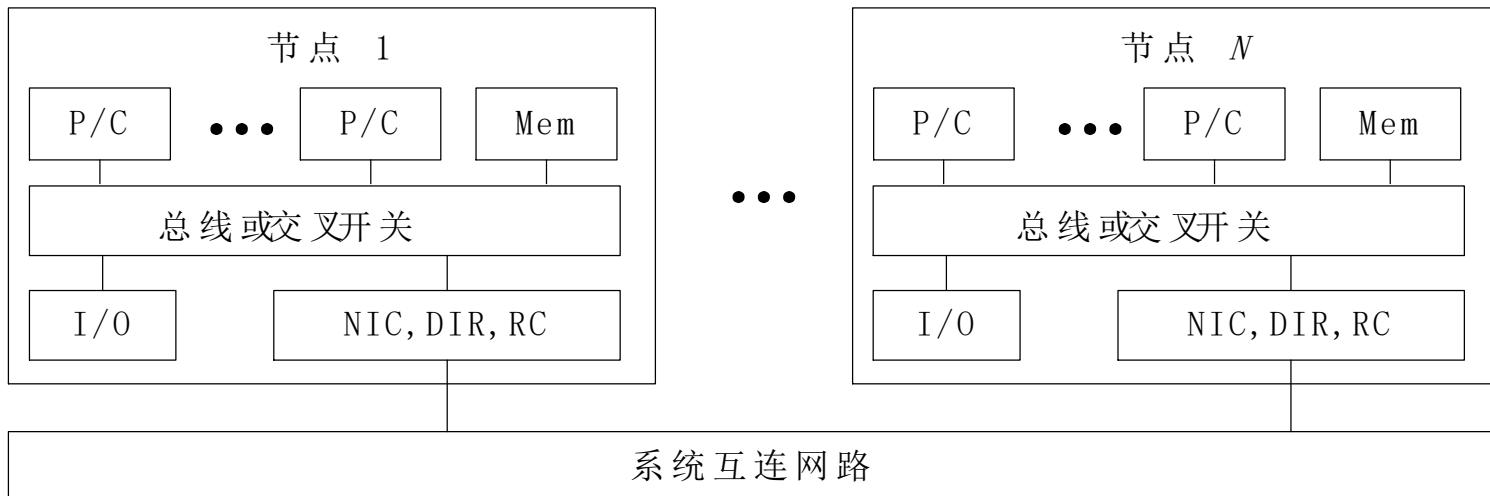
## ■ NUMA (Non-Uniform Memory Access) 非均匀存储器访问

- 共享存储器分布在所有的处理器中
- 处理器访问存储器的时间不均匀
- 每台处理器可带私有高速缓存
- 外设可以一定形式共享



# 共享内存系统——CC-NUMA

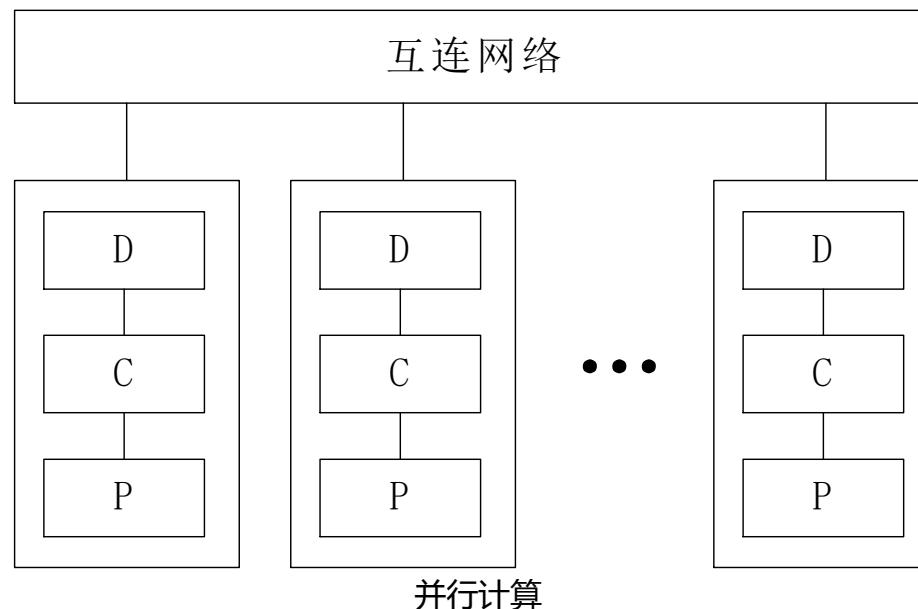
- CC-NUMA (Coherent-Cache Nonuniform Memory Access) 高速缓存一致性非均匀存储访问
  - 大多数使用基于目录的高速缓存一致性协议
  - 保留SMP易于编程的优点，改善常规SMP的可扩放性
  - 分布共享存储的DSM多处理机系统
  - 程序员无需明确地在节点上分配数据



# 共享内存系统——COMA

## ■ COMA(Cache-Only Memory Access) 全高速缓存存取

- 没有存储层次结构，全部高速缓存组成全局地址空间
- 利用分布的高速缓存目录进行远程高速缓存的访问
- COMA中的高速缓存容量一般大于2 级高速缓存容量
- 数据开始时可任意分配



# 目录

- 并行系统分类
- 共享内存系统
- 分布式内存系统**
- 异构系统架构
- 互连网络

# 分布式内存系统

- 连接多台服务器形成一个不同享内存的计算平台



**集群:** 数十台服务器

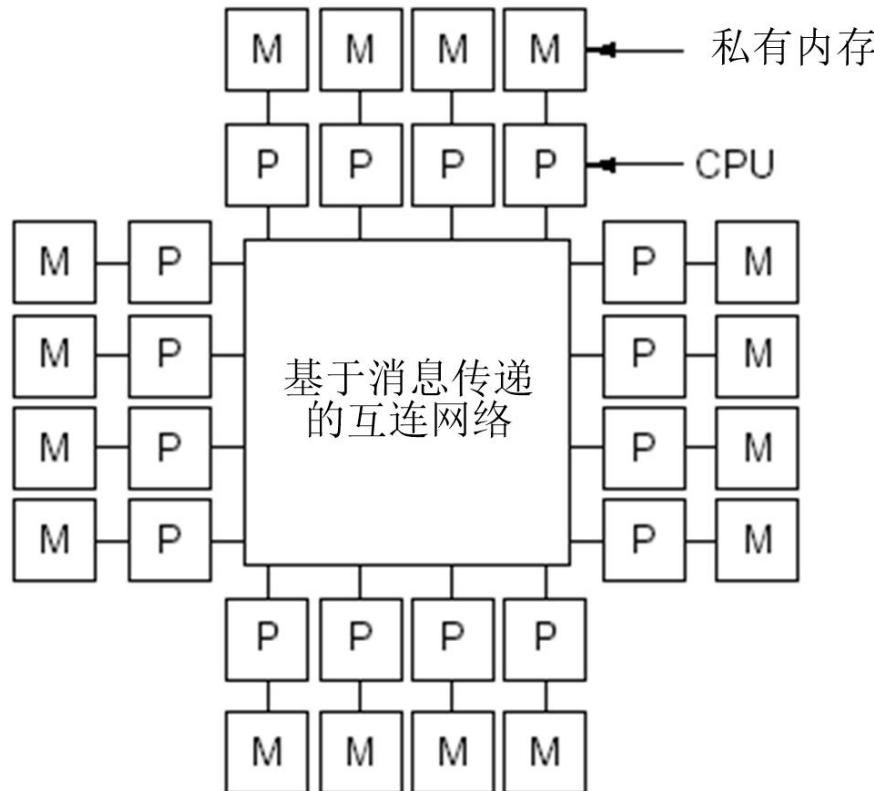


**超级计算机:** 数百台  
服务器

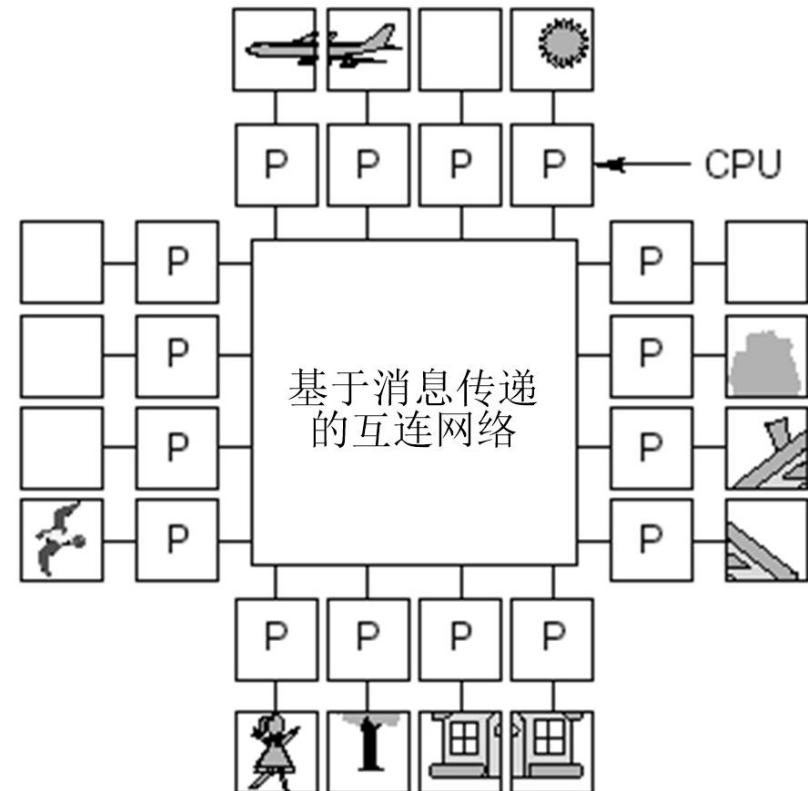


**数据中心:** 数千台服务器

# 分布式内存系统



(a) 16个CPU的多计算机系统(每个CPU都有私有内存)

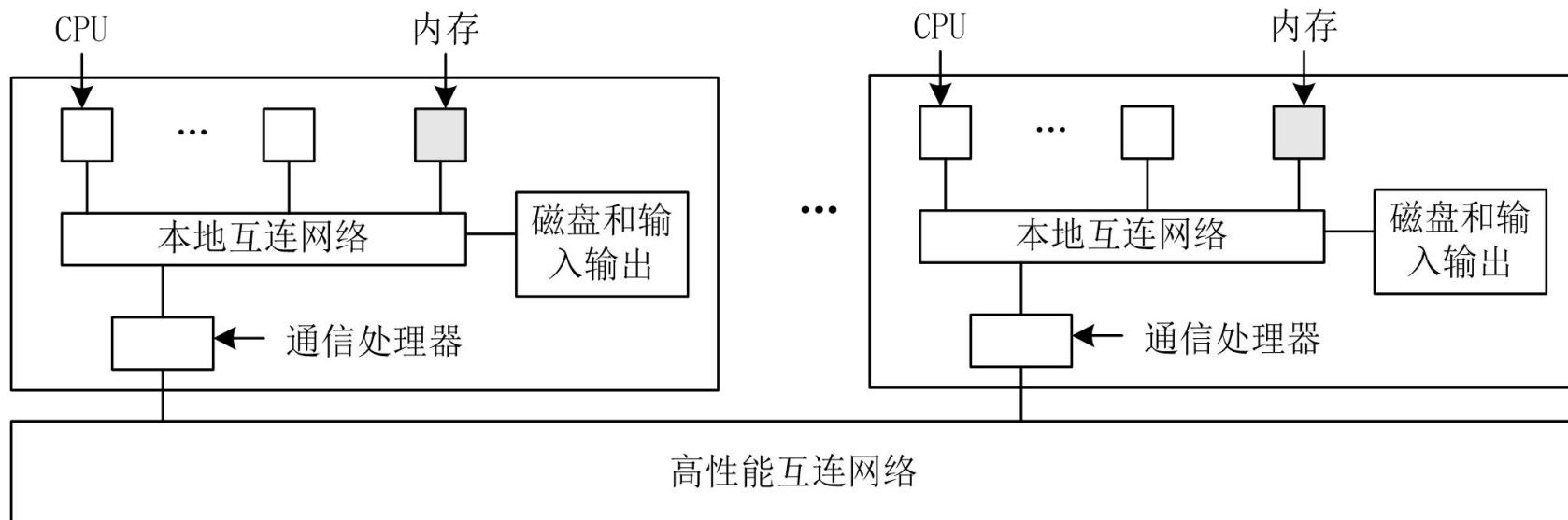


(b) 图13.1中的图像分布在16块内存中

# 分布式内存系统

## ■ 通用多计算机体系结构

- 每个节点都由一个或者多个CPU、RAM、磁盘以及其它的输入/输出设备和通信处理器组成
- 通信处理器通过互连网络相互连接起来。可以使用多种不同的拓扑结构，交换策略和寻径算法



# 分布式内存系统——MPP

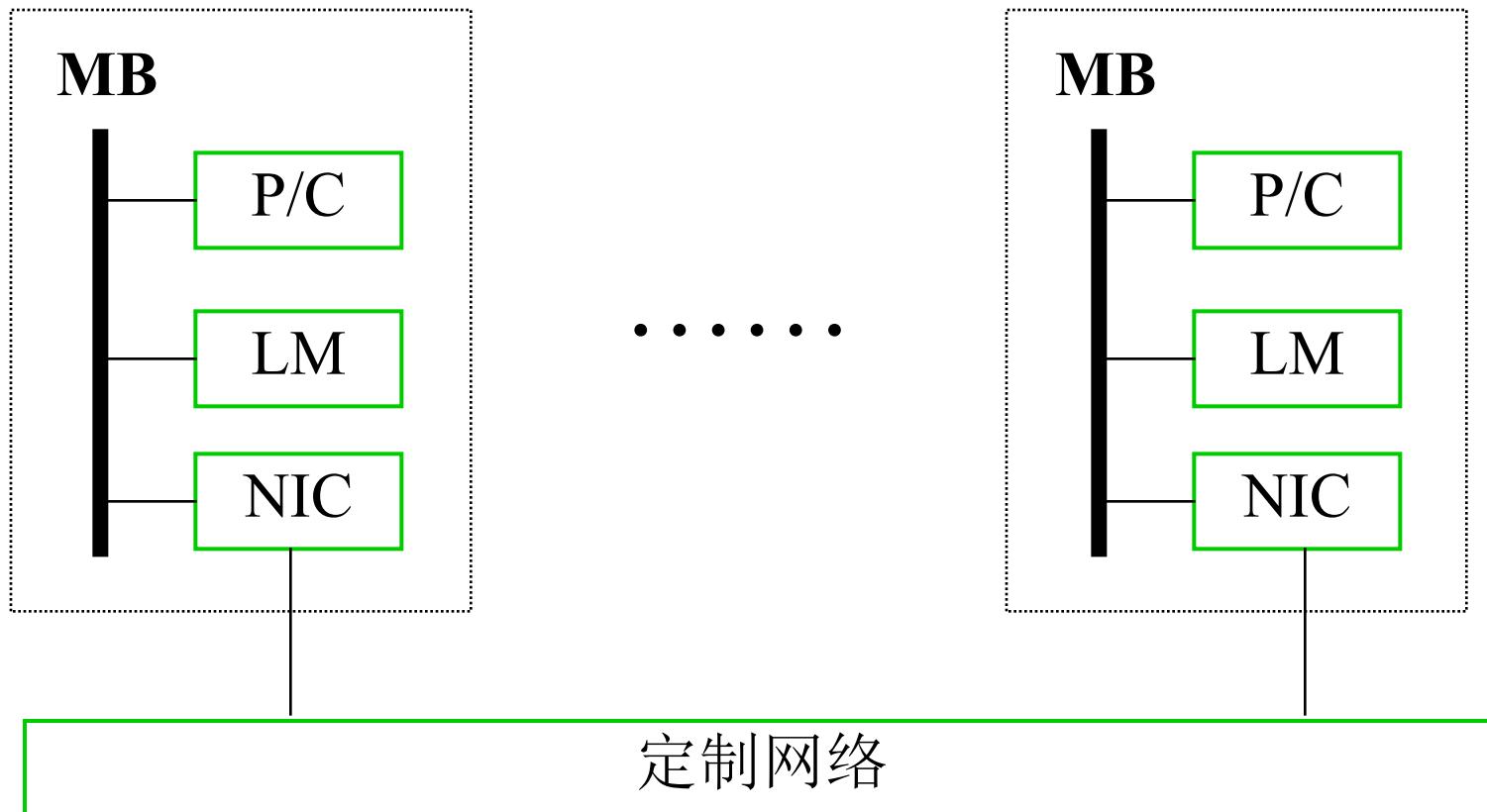
- MPP (Massively Parallel Processor) 大规模并行处理机
  - 由成百上千台处理机组成的大规模并行计算机系统
  - 过去主要用于科学计算、工程模拟等以计算为主的场合，目前也广泛应用于商业和网络应用中
  - 开发困难，价格高，市场有限。是国家综合实力的象征
- 系统特点
  - 一般使用标准的商用CPU作为它们的处理器
  - 使用了高性能的私用的互连网络，可以在低延时和高带宽的条件下传递消息
  - 具有强大的输入/输出能力
  - 能够进行特殊的容错处理

# 分布式内存系统——MPP

LM: 本地存储器

NIC: 网络接口电路

MB: 存储器总线

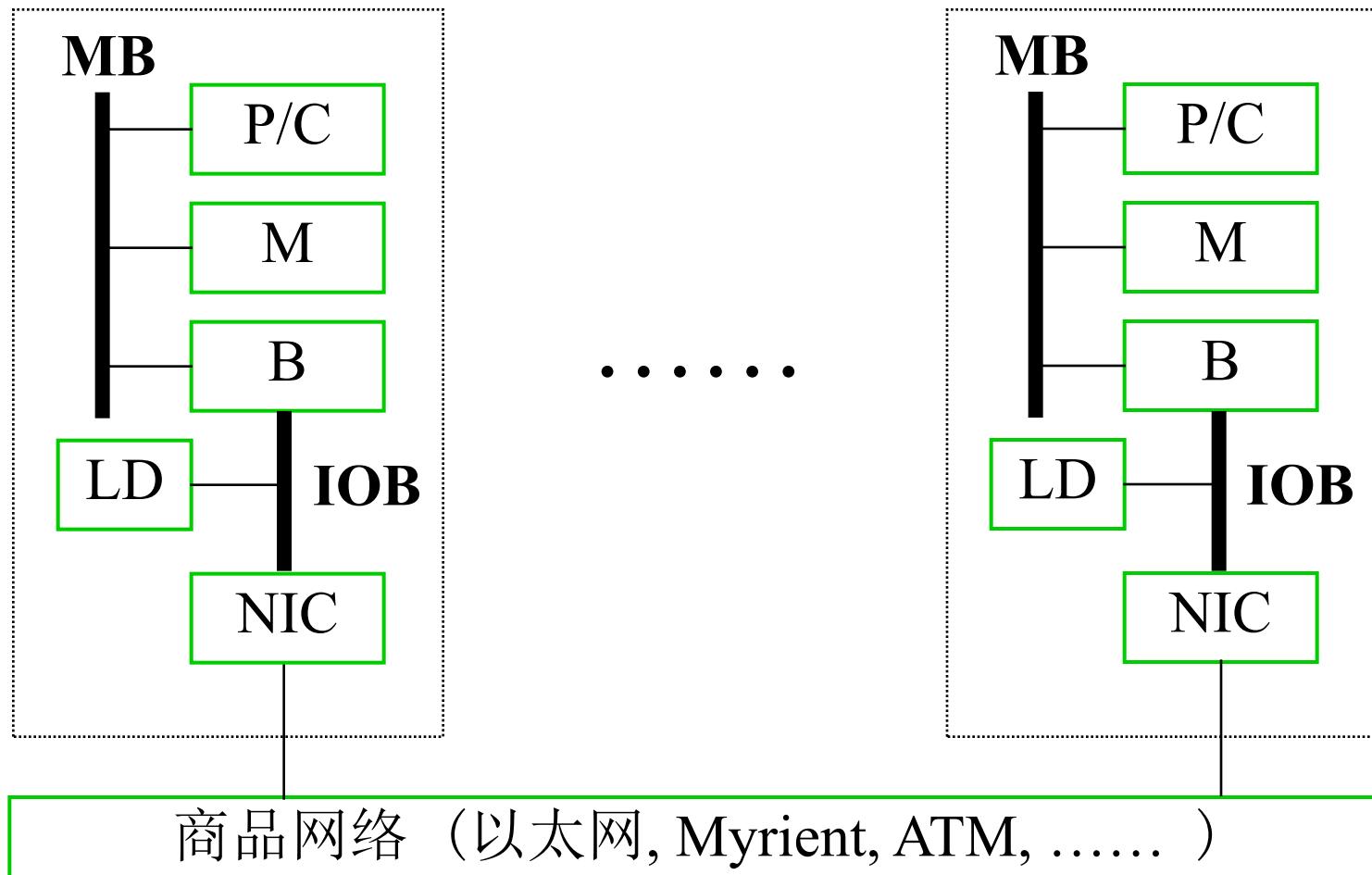


# 分布式内存系统——Cluster

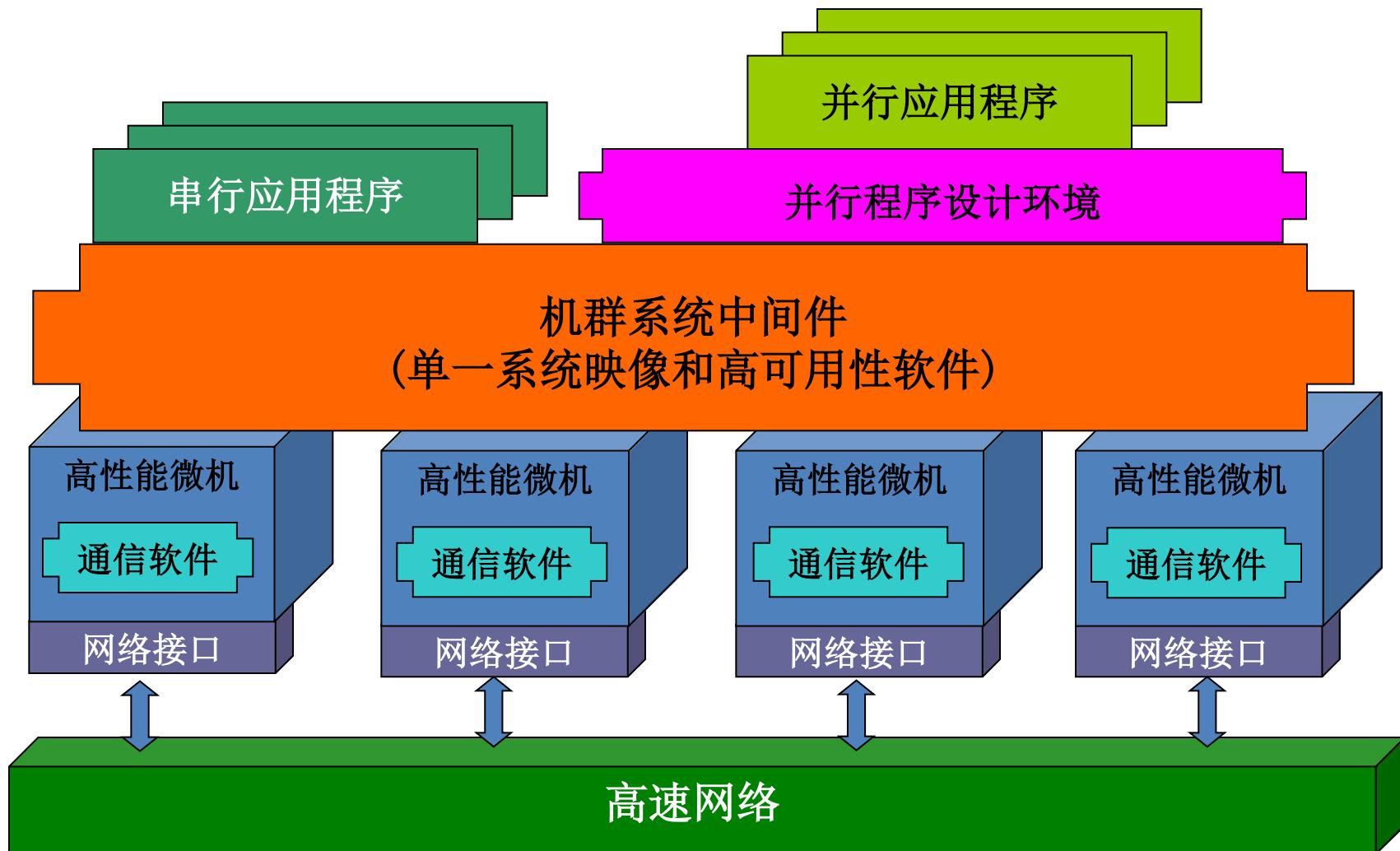
- Cluster 集群 (COW机群)
  - 由大量的PC机或者工作站通过商用网络连接在一起构成
  - 可以完全使用可以买到的商用组件装配而成，这些商用组件都是大规模生产的产品，能够获得较高的性价比
- 与MPP的区别 (体系结构方面)
  - 节点是更完整的计算机，计算机可以是同构或异构
  - 节点都有自己的磁盘，驻留有自己的完整的操作系统；而MPP系统结点一般没有磁盘，只驻留操作系统内核
  - MPP使用制造厂商专有的高速通信网络；COW一般采用公开销售的标准高速局域网或系统域网，网络通常是与节点计算机的I/O总线相连（松散耦合），而MPP的网络接口是连到处理节点的存储总线上（紧耦合）

# 分布式内存系统——Cluster

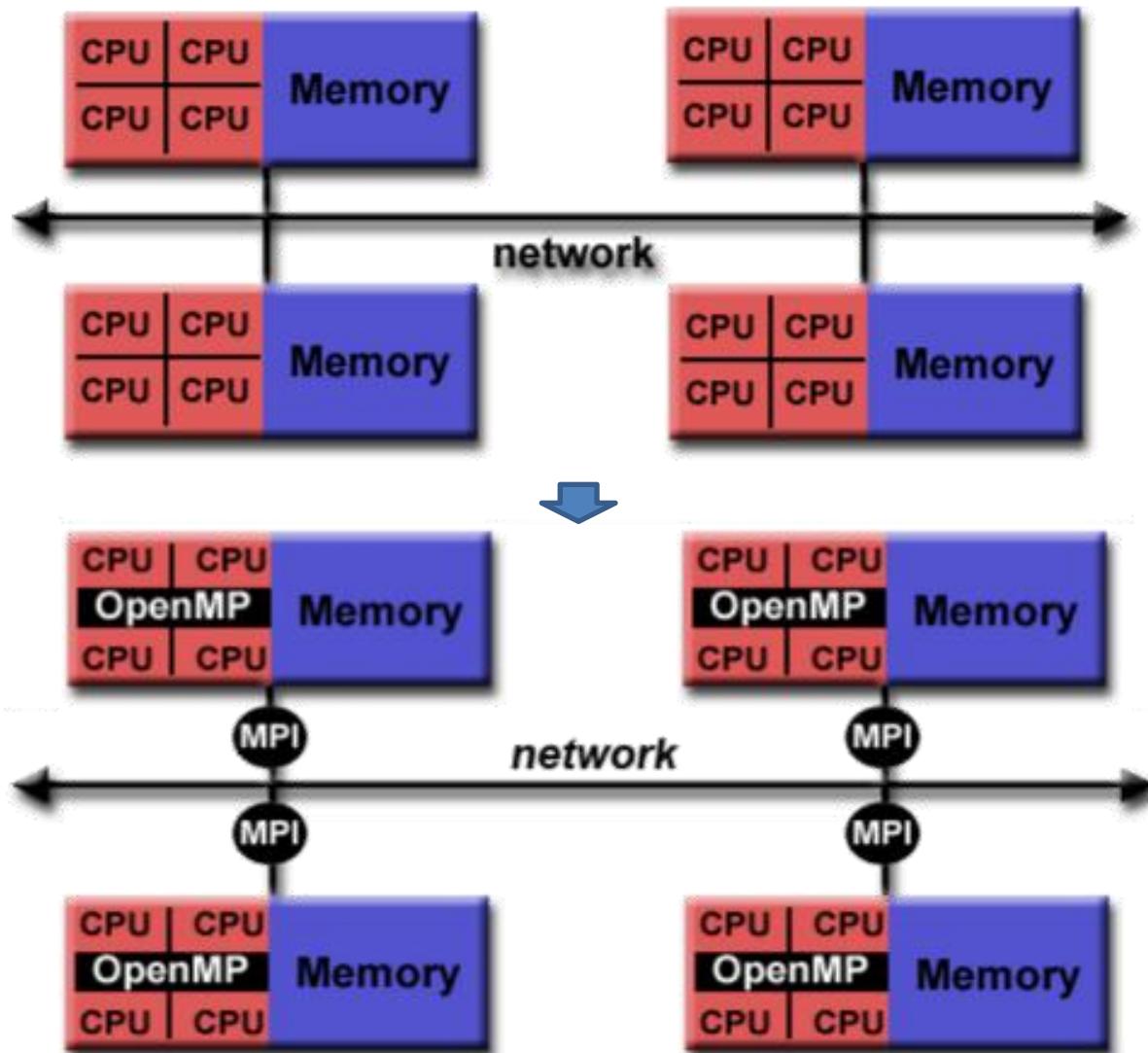
B: 存储总线与I/O总线的接口  
LD: 本地磁盘  
IOB: I/O总线



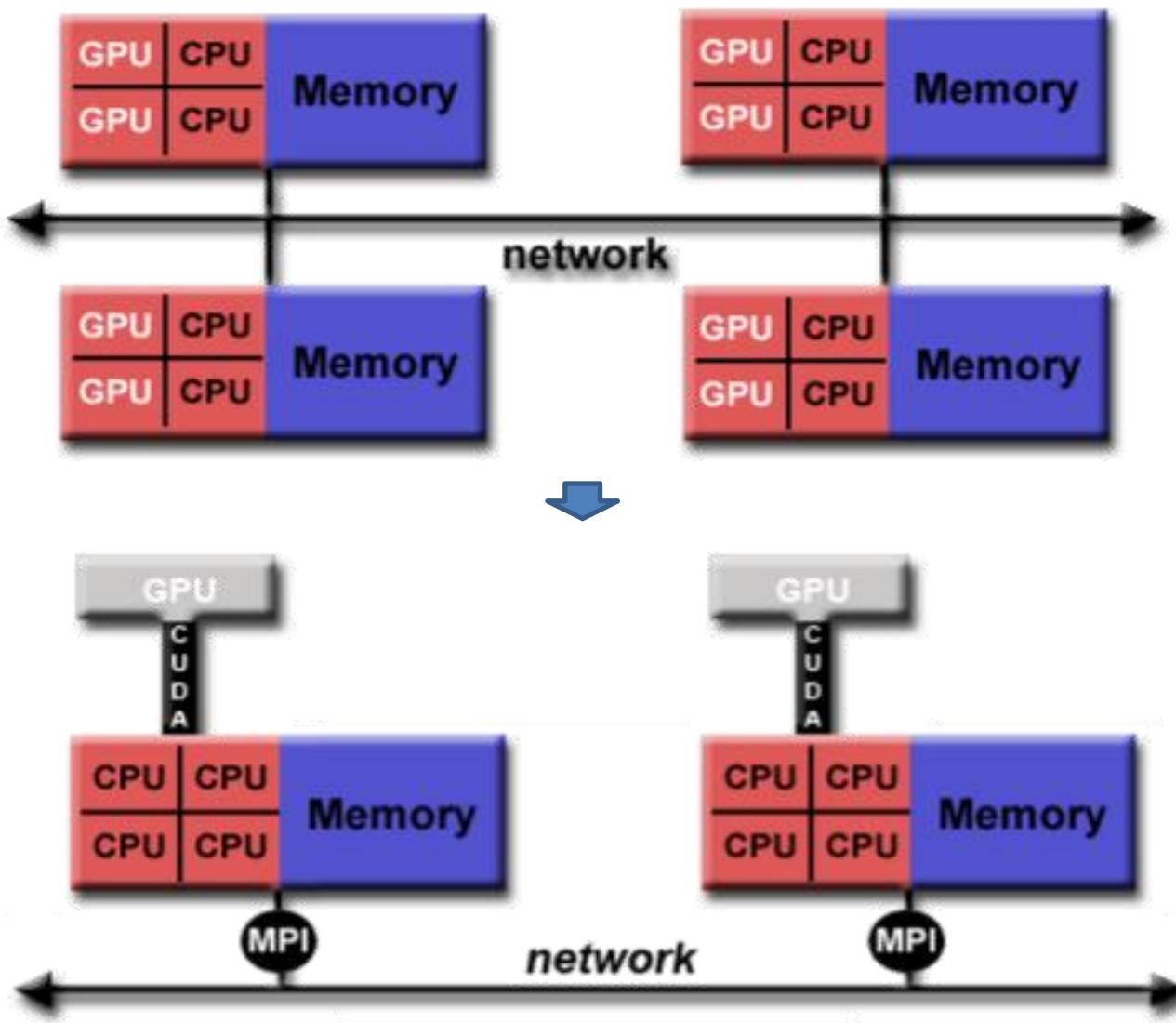
# 分布式内存系统——Cluster



# 分布式内存系统



# 分布式内存系统

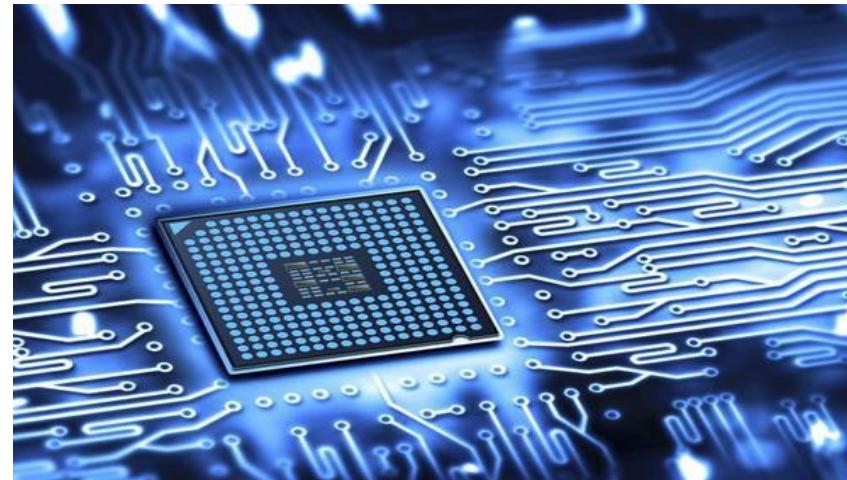


# 目录

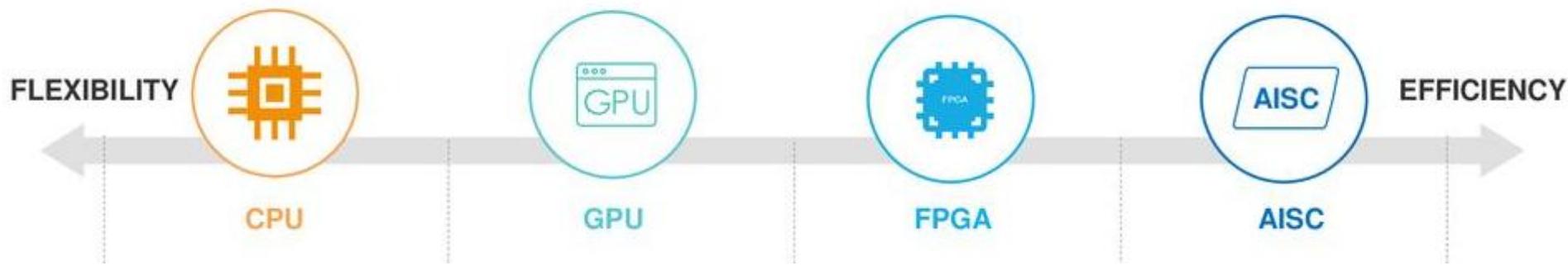
- 并行系统分类
- 共享内存系统
- 分布式内存系统
- 异构系统架构
- 互连网络

# 异构系统架构

- 多种处理器或核心
- 使用加速器来提高性能或能源效率，通常结合专门的处理能力来处理特定任务
- 适合执行single instruction, multiple data (SIMD) 和 single instruction, multiple threads (SIMT)



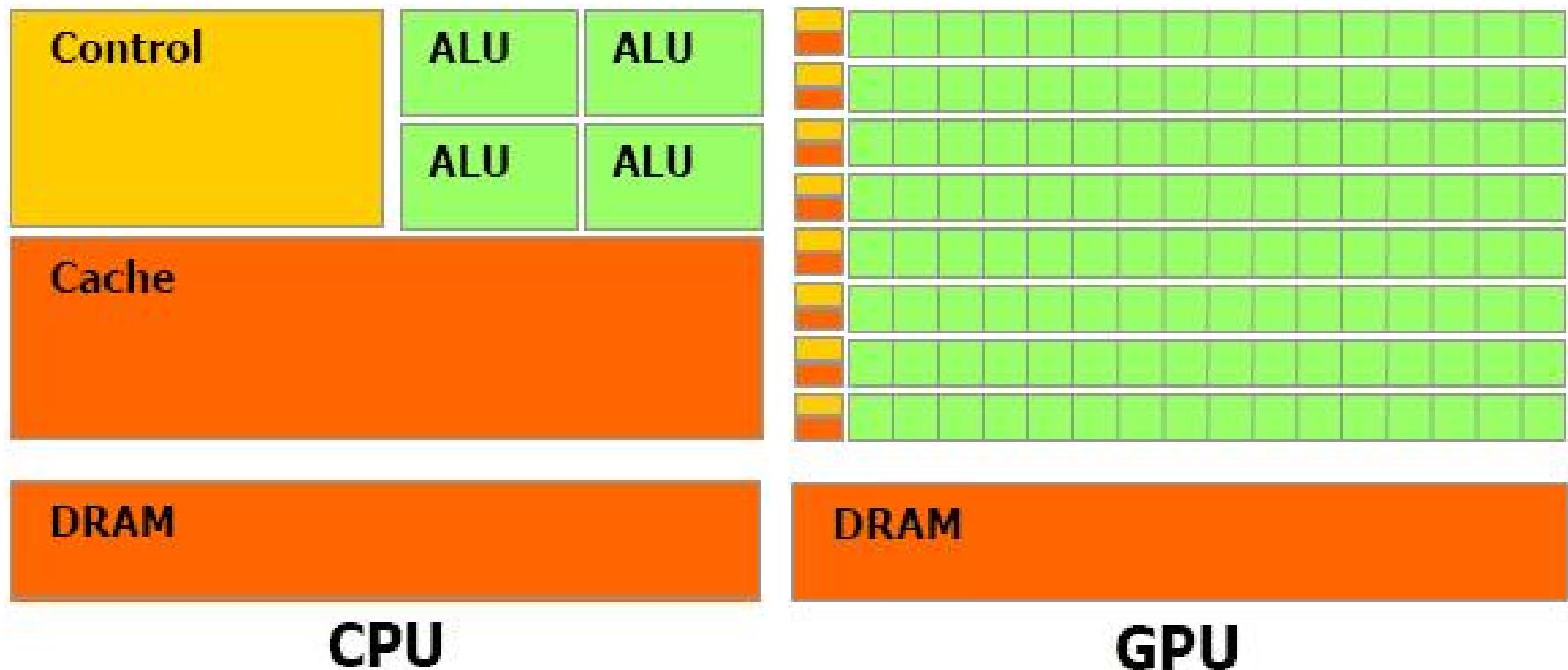
# 异构系统架构



- |   |  |   |   |
|---|--|---|---|
| <ul style="list-style-type: none"><li>➤ 通用型, 复杂计算</li><li>➤ 灵活、易用、通用</li><li>➤ 性能较低</li></ul> | <ul style="list-style-type: none"><li>➤ 批量数据并行计算</li><li>➤ 高性能</li><li>➤ 高功耗</li></ul> | <ul style="list-style-type: none"><li>➤ 不规则数据并行计算</li><li>➤ 性能好</li><li>➤ 能效比高</li><li>➤ 灵活</li></ul> | <ul style="list-style-type: none"><li>➤ 适用数据并行计算</li><li>➤ 高性能</li><li>➤ 低功耗</li><li>➤ 专用电路, 不可修改</li></ul> |
|---|--|---|---|

# 异构系统架构

## ■ NVIDIA GPU , CPU和GPU比较



# 异构系统架构

## ■ NVIDIA GPU发展

Micro-architecture	Launch	GPU
Tesla	2008年	S1070
Fermi	2010年	M2090、S2070
Kepler	2012年	K40/K80
Maxwell	2014年	GeForce GTX 750Ti、GeForce GTX TITAN X
Pascal	2016年	Tesla P40、GTX 1080TI/Titan XP、Quadro GP100/P6000/P5000
Volta	2017年	Tesla V100、GeForce TITAN V
Turing	2018年	RTX 2080Ti /2080/2070 Quadro RTX6000
Ampere	2020年	A100、RTX 3080Ti/3080/3070/3060、RTX 3090
Hopper	2022年	H100
Ada Lovelace	2022年	L40、L4、RTX 40 系列

# 异构系统架构

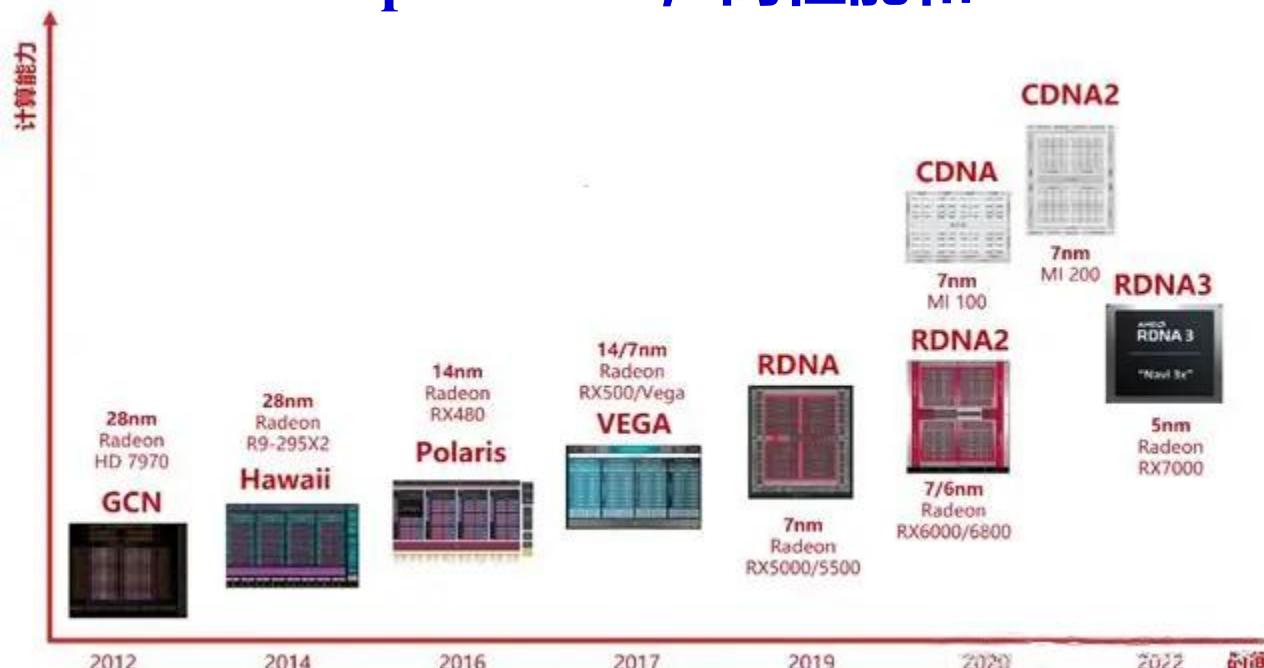
## ■ NVIDIA GPU发展



# 异构系统架构

## ■ AMD GPU 架构：

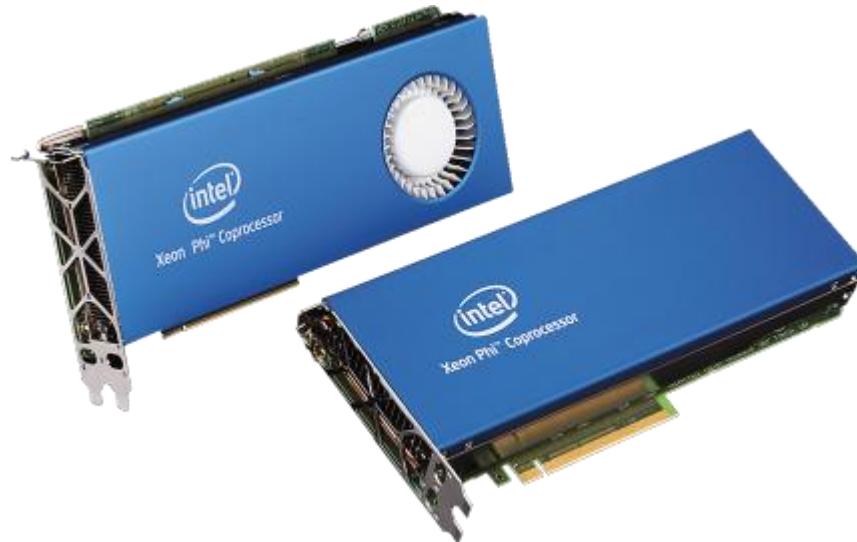
- TeraScale
- GCN: Graphics Core Next
- RDNA: Radeon DNA, 游戏
- CDNA: Compute DNA, 高性能和AI



# 异构系统架构

## ■ Intel MIC (Many Integrated Core)

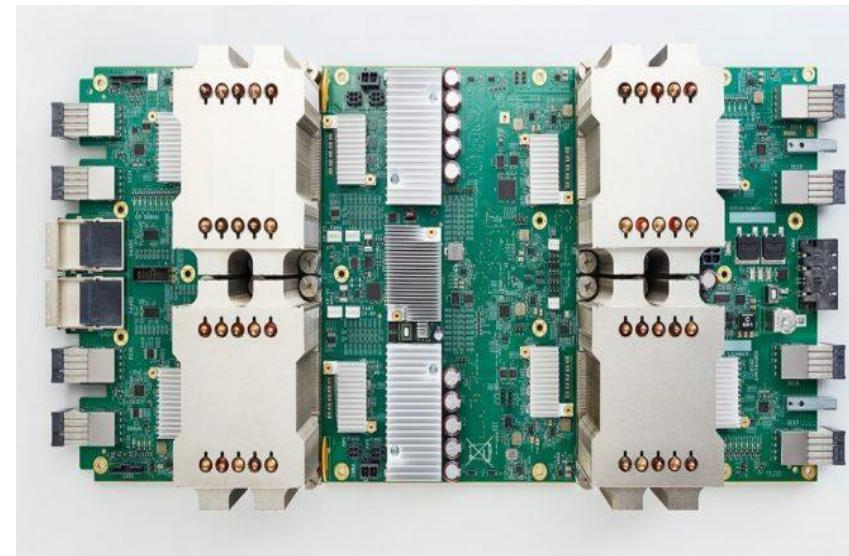
- 由许多x86核集成在一起的加速设备
- 核心设计与奔腾处理器相似
- 添加64位硬件支持，更多的硬件线程（每四个核4个硬件线程），512位的SIMD向量支持



# 异构系统架构

## ■ TPU (Tensor Processing Unit)

- 人工智能加速器**专用集成电路 (ASIC)**
- 谷歌开发
- 主要任务是**矩阵处理，乘法和累加运算的组合**
- TPU 包含数千个乘法累加器，这些累加器彼此直接连接以形成大型物理矩阵



# 异构系统架构

## ■ TPU 架构

	TPUv1	TPUv2	TPUv3	TPUv4
<b>Date Introduced</b>	2016	2017	2018	2021
<b>Process Node</b>	8 nm	16 nm	16 nm	7 nm
<b>Die Size (mm<sup>2</sup>)</b>	331	< 625	< 700	< 400
<b>On chip memory (MiB)</b>	28	32	32	144
<b>Clock Speed (MHz)</b>	700	700	940	1050
<b>Memory (GB)</b>	8GB DDR3	16GB HBM	32GB HBM	8GB
<b>TDP(W)</b>	75	280	450	175

# 异构系统架构

- **FPGA (Field Programmable Gate Array)**
  - 专用集成电路中的一种**半定制电路**, 是可编程的逻辑列阵
  - **低功耗和高能量效率**
  - **重新编程以加速不同的应用**
  - **采用硬件的方式来实现逻辑和算法, 不需发射和解析指令**
  - **针对需求设计多种计算部件来同时实现数据并行和流水线并行**
  - **实例: 微软 Catapult**

# 异构系统架构

## ■ CPU+FPGA异构架构种类：

- FPGA 作为**外部独立的计算模块**，通过网络、数据总线、I/O 接口等机制与处理器进行连接；
- FPGA 作为**共享内存的计算模块**，被用于可重构计算器件置于Cache高速缓存和内存之间；
- FPGA 作为**协处理器**，与 CPU**共享缓存**；
- FPGA **集成处理器架构**，将处理器高度嵌入到FPGA 可编程器件中，实现了CPU与FPGA的紧耦合。随着FPGA处理单元与CPU之间耦合度的增加，通信代价逐渐降低，系统设计的复杂度也相应提高。

# 目录

- 并行系统分类
- 共享内存系统
- 分布式内存系统
- 异构系统架构
- **互连网络**

# 互连网络概述

## ■ 并行计算机的通信体系结构是系统核心

- 两个层次：底层的互连网络；上层的语言、软件工具包、编译器、操作系统等提供的通信支持

## ■ 互连网络是并行计算机系统内部的互连网络

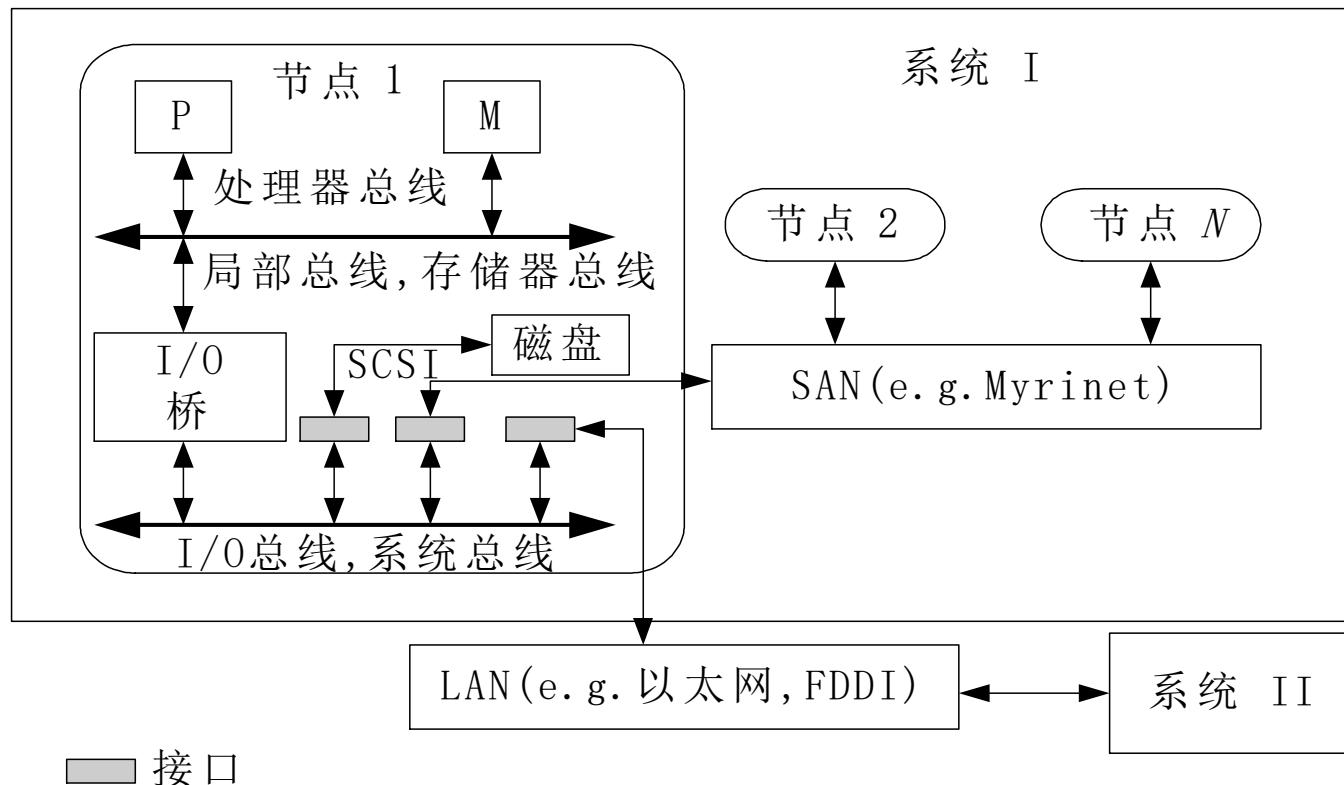
- 定义：由开关元件按一定拓扑结构和控制方式构成的网络以实现计算机系统内部多个处理机或多个功能部件间的相互连接
- 与计算机网络在工作原理、概念以及术语上有许多相同或相似之处；并且某些并行计算机系统中的互连网络就是高速以太网和ATM网络

## ■ 互连网络一般由以下五个部分组成

- CPU、内存模块、接口、链路和交换节点

# 互连网络

## ■ 局部总线、I/O总线、SAN和LAN



# 互连网络拓扑结构

- 互连网络的拓扑结构描述了链路和交换节点是如何组织安排的。拓扑结构可以用图来表示，链路用边表示，交换节点用节点表示
- 静态网络
  - 静态网络(Static Networks)是指节点间有着固定连接通路且在程序执行期间，这种连接保持不变的网络
- 动态网络
  - 动态网络(Dynamic Networks)由开关单元构成，可按应用程序的要求动态地改变连接状态。如总线、交叉开关，多级交换网络等

# 互连网络参数

- **节点度** (Node Degree) : 与节点相连接的边数, 表示节点所需要的端口数, 根据链路到节点的方向, 节点度可以进一步表示为: **节点度 = 入度 + 出度**, 其中**入度**是进入节点的链路数, **出度**是从节点出来的链路数
- **网络直径** (Network Diameter) : 网络中任何两个节点之间的最长距离, 即最大路径数
- **对剖宽度** (Bisection Width) : 对分网络各半所必须移去的最少边数
- **对剖带宽** (Bisection Bandwidth) : 每秒钟内, 在最小的对剖平面上通过所有连线的最大信息位(或字节)数
- 如果从任一节点观看网络都一样, 则称网络为**对称的** (Symmetry)

# 静态互连网络

## ■ 一维线性阵列 (1-D Linear Array)

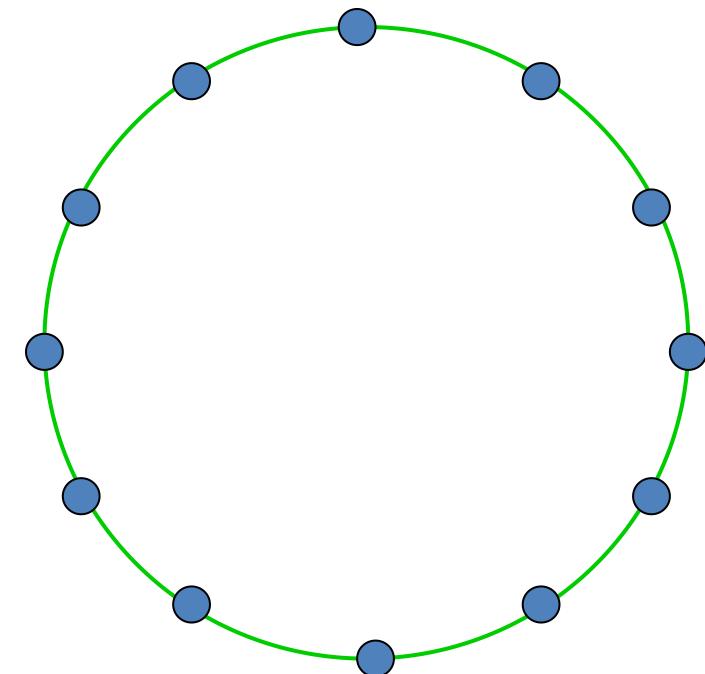
- 并行机中最简单、最基本的互连方式
- 每个节点仅与其左右近邻相连，也叫二近邻连接
- 对 $N$ 个结点的线性阵列，有 $N-1$ 条链路，直径为 $N-1$   
(任意两点之间距离的最大值) 度为2不对称，对剖宽度为1。 $N$ 很大时，通信效率很低



# 静态互连网络

## ■ 环形

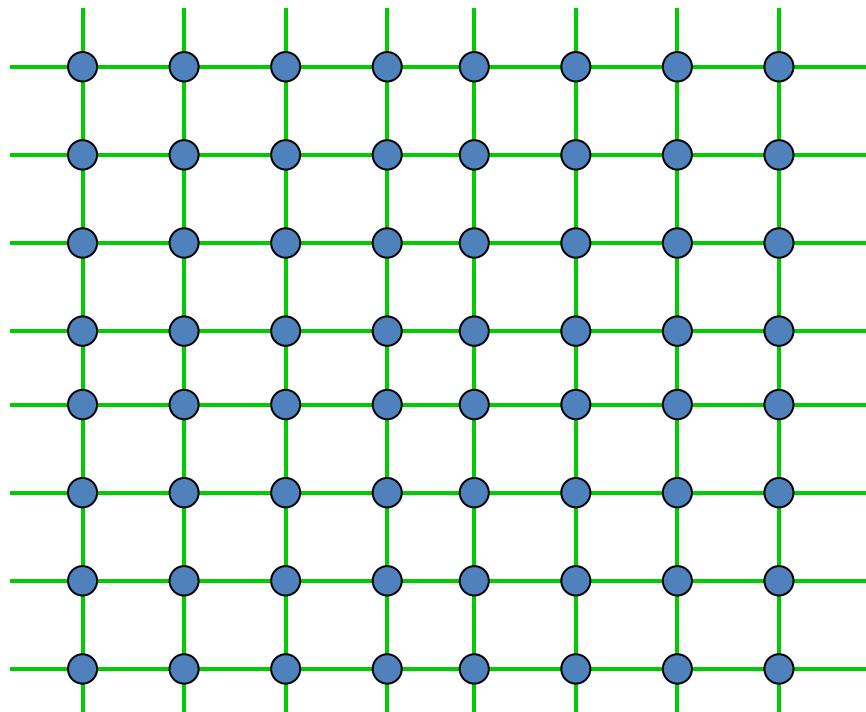
- 对 $N$ 个节点的环，考虑相邻节点数据传送方向：
- 双向环：链路数为 $N$ ，直径 $\lfloor N/2 \rfloor$ ，度为2，对称，对剖宽度为2
- 单向环：链路数为 $N$ ，直径 $N-1$ ，度为2，对称，对剖宽度为2



# 静态互连网络

## ■ 二维网孔 (2-D Mesh)

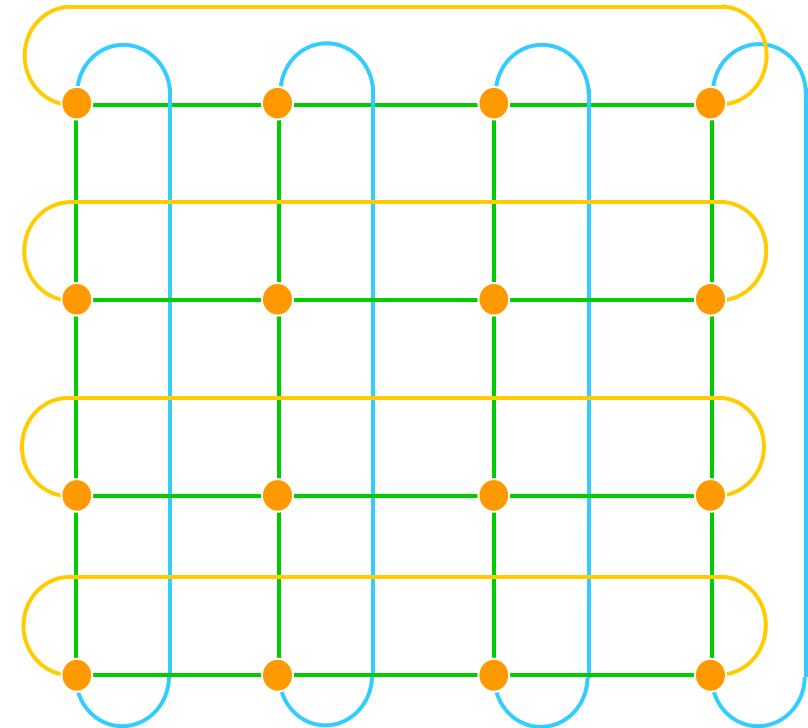
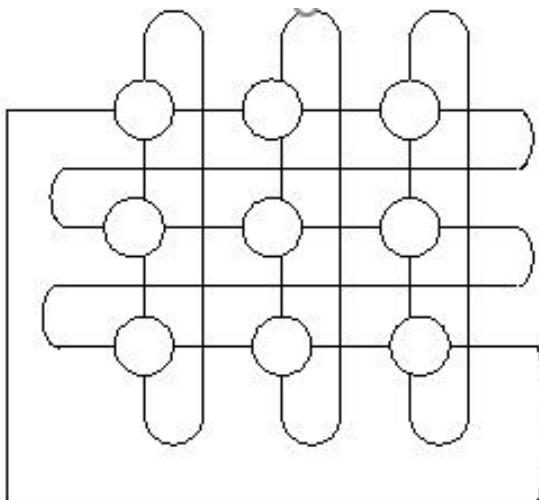
- $N$ 个节点的  $\sqrt{N} \times \sqrt{N}$  网格
- 每个节点只与其上、下、左、右的近邻相连 (边界节点除外)
- 链路数量为  $2(N - \sqrt{N})$ , 节点度为4, 网络直径为  $2(\sqrt{N} - 1)$ , 非对称, 对剖宽度为  $\sqrt{N}$



# 静态互连网络

## ■ 二维环绕 (2-D Torus)

- $N$ 个节点的  $\sqrt{N} \times \sqrt{N}$  网格
- 垂直和水平方向带环绕
- 链路数量为 $2N$ , 节点度恒为4, 网络直径  $2\lfloor\sqrt{N}/2\rfloor$ , 对称, 对剖宽度为 $2\sqrt{N}$

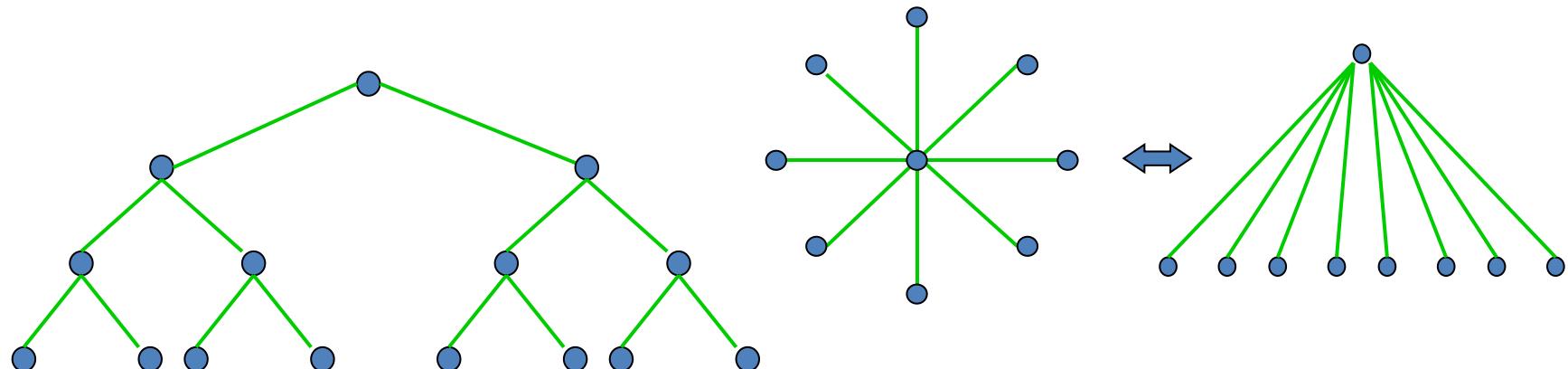


垂直方向带环绕, 水平方向呈蛇状, 网络直径  $\sqrt{N} - 1$   
Illiad网孔

# 静态互连网络

## ■ 二叉树

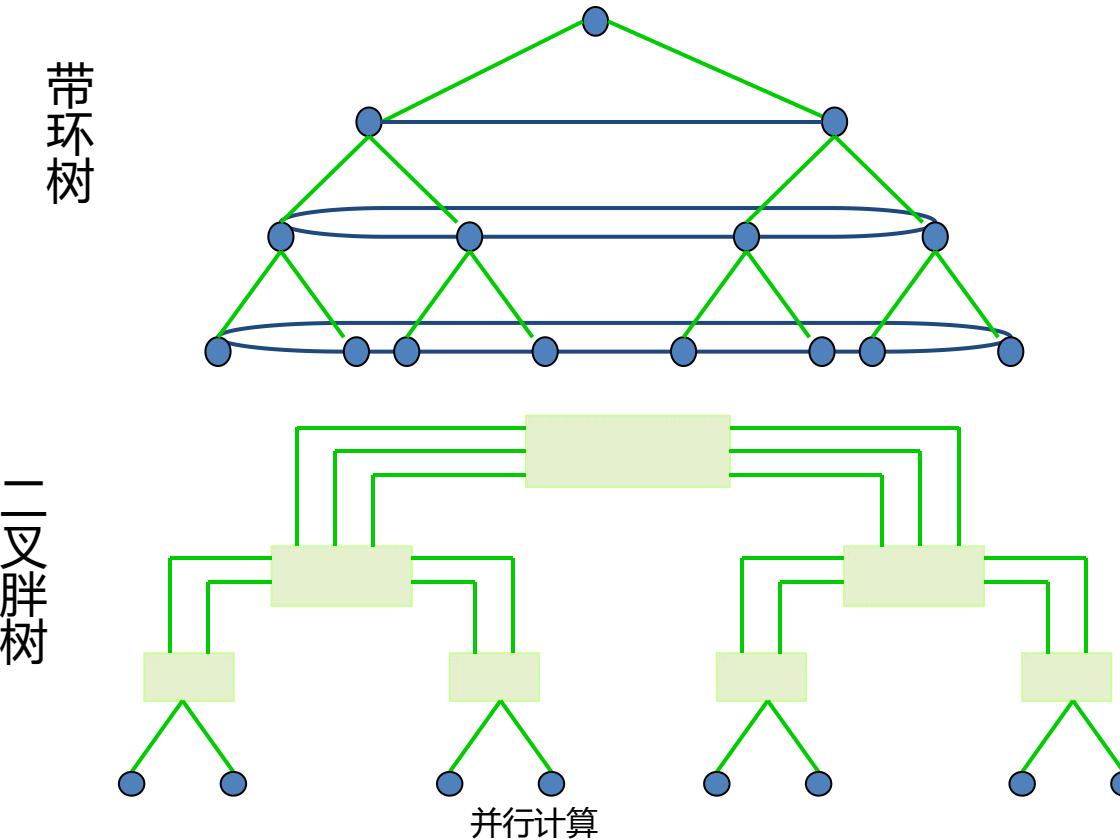
- 除了根、叶节点，每个内节点只与其父节点和两个子节点相连
- 节点度为3，对剖宽度为1，直径为  $2(\lceil \log N \rceil - 1)$
- 尽量增大节点度为  $N-1$ ，直径缩小为2，此时就变成星形网络，其对剖宽度为  $\lfloor N/2 \rfloor$



# 静态互连网络

## ■ 二叉树

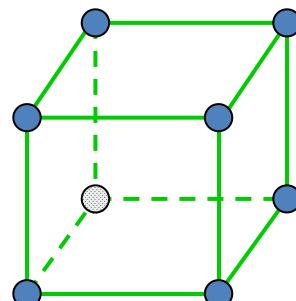
- 传统二叉树的主要问题是，根节点易成为通信瓶颈
- 这两种结构都可以缓解根节点的瓶颈问题



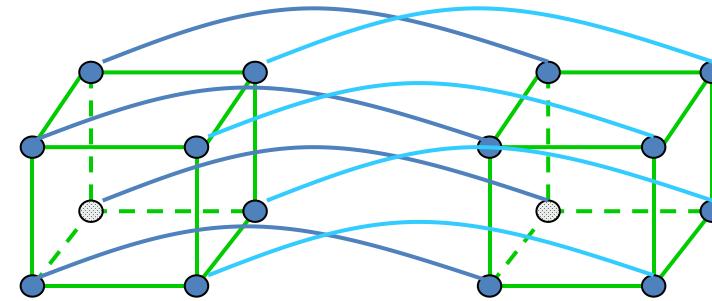
# 静态互连网络

## ■ 超立方

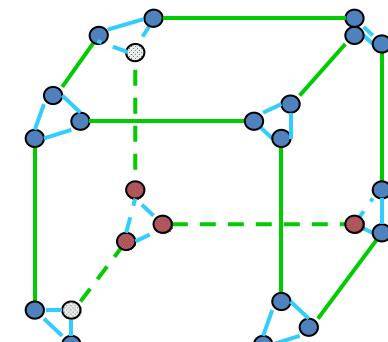
- 一个 $n$ -立方由  $N = 2^n$  个顶点组成，4-立方是由3-立方的对应顶点连接而成
- $n$ -立方的节点度为 $n$ ，网络直径为 $n$ ，对剖宽度为 $N/2$
- 如果将 $n$ -立方的每个顶点代之以一个 $n$ 个节点的环，节点总数  $n2^n$ ，就构成了 $n$ -立方环，每个顶点度为3



3-立方体



4-立方体



3-立方环

# 静态互连网络

网络名称	网络规模	节点度	网络直径	对剖宽度	对称	链路数
线性阵列	$N$	2	$N-1$	1	非	$N-1$
环形	$N$	2	$\lfloor N/2 \rfloor$ (双向)	2	是	$N$
2-D 网孔	$(\sqrt{N} \times \sqrt{N})$	4	$2(\sqrt{N} - 1)$	$\sqrt{N}$	非	$2(N - \sqrt{N})$
Illiac 网孔	$(\sqrt{N} \times \sqrt{N})$	4	$(\sqrt{N} - 1)$	$2\sqrt{N}$	非	$2N$
2-D 环绕	$(\sqrt{N} \times \sqrt{N})$	4	$2\lfloor \sqrt{N}/2 \rfloor$	$2\sqrt{N}$	是	$2N$
二叉树	$N$	3	$2(\lceil \log N \rceil - 1)$	1	非	$N-1$
星形	$N$	$N-1$	2	$\lfloor N/2 \rfloor$	非	$N-1$
超立方	$N = 2^n$	n	n	$N/2$	是	$n \cdot N/2$
立方环	$N = k \cdot 2^k$	3	$2k - 1 + \lfloor k/2 \rfloor$	$N/(2k)$	是	$3N/2$

# 动态互连网络

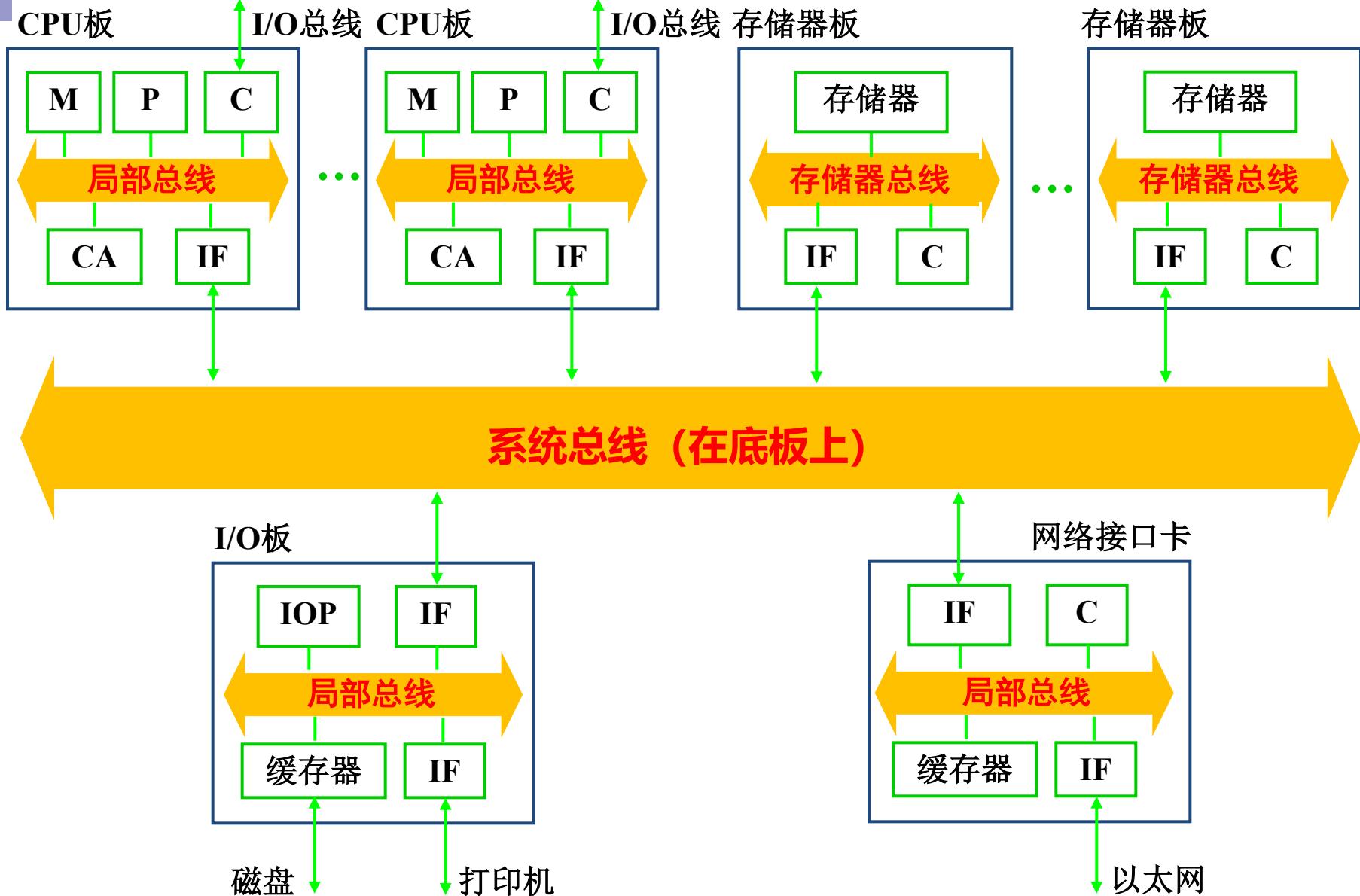
## ■ 网络特点

- 动态网络中的连接不固定，在程序执行过程中可根据需要改变
- 网络的开关元件，链路可通过设置这些开关的状态来重构
- 动态网络主要有总线、交叉开关、多级交换网络

# 动态互连网络

## ■ 总线

- 总线实际上是连接处理器、存储器和I/O等外围设备的一组导线和插座
- 它在某一时刻只能用于一对源和目的之间传输数据。
- 当有多对源和目的请求使用总线时，要进行总线仲裁。  
当CPU数目较多时对总线争用严重 (<=32个)



IF:专用逻辑接口 C:专用控制器 P:处理器 M:局部存储器 CA:高速缓存 IOP:I/O处理器

并行计算

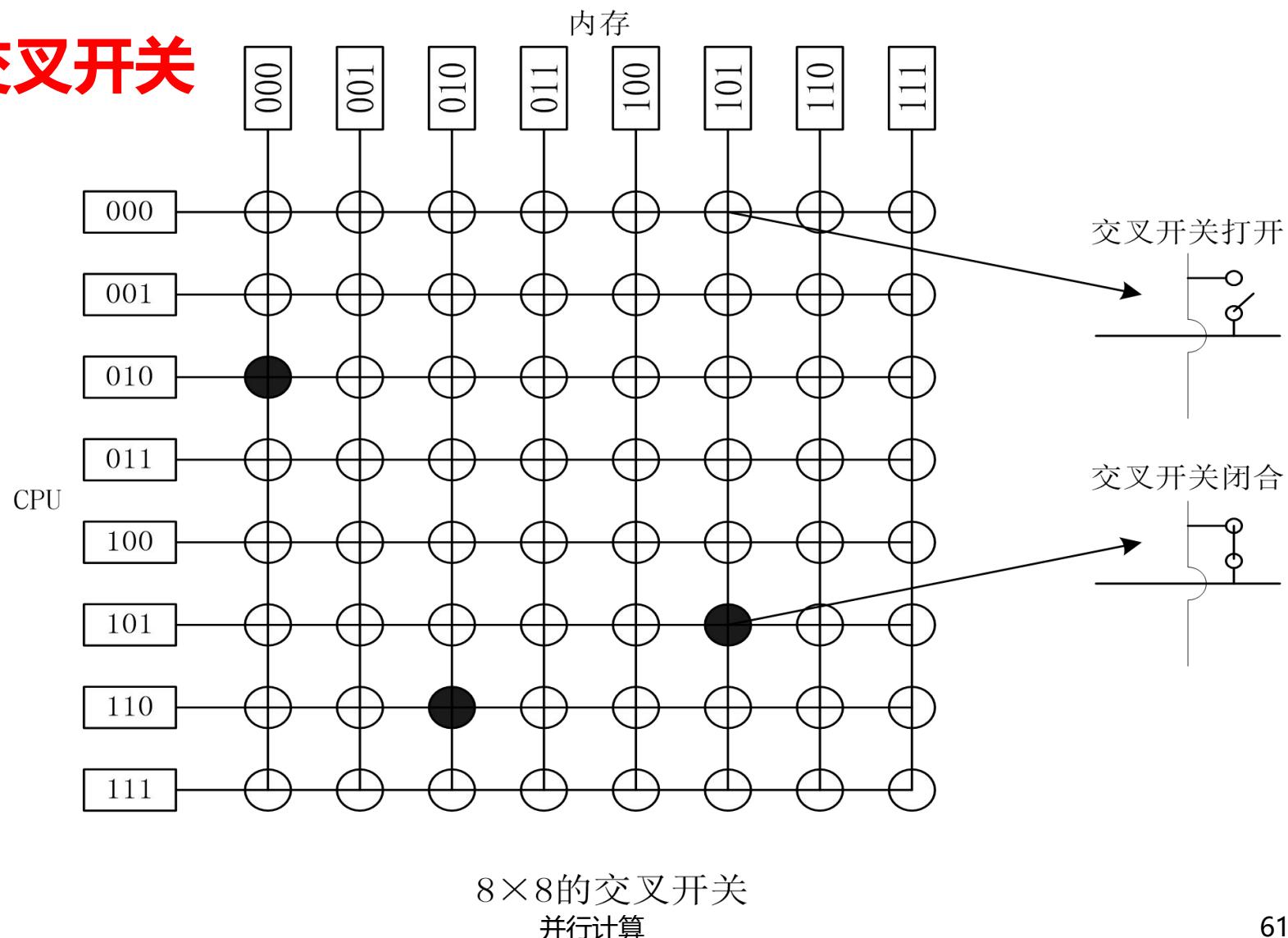
# 动态互连网络

## ■ 交叉开关

- 单级交换网络，可为每个端口提供更高的带宽。像电话交换机一样，交叉点开关可由程序控制动态设置其处于“开”或“关”状态，从而能提供所有（源、目的）对之间的动态连接
- 交叉开关一般有两种使用方式：一种是用于对称的多处理机或多计算机机群中的处理器间的通信；另一种是用于SMP服务器或向量超级计算机中处理器和存储器之间的存取

# 动态互连网络

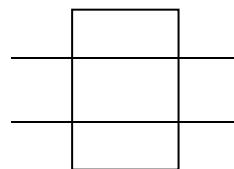
## ■ 交叉开关



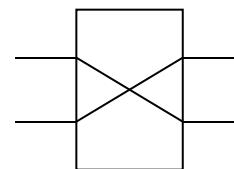
# 动态互连网络

## ■ 多级交换网络

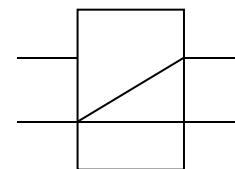
- MIN (Multistage Interconnection Network) : 单级交叉开关级联起来形成多级互连网络
- 交换开关模块：一个交换开关模块有n个输入和n个输出，每个输入可连接到任意输出端口，但只允许一对一或一对多的映射
- 级间互联模式：均匀洗牌、蝶式、多路洗牌、交叉开关及立方体连结等



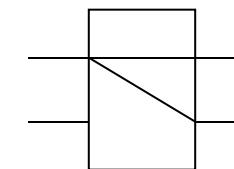
直通



交换



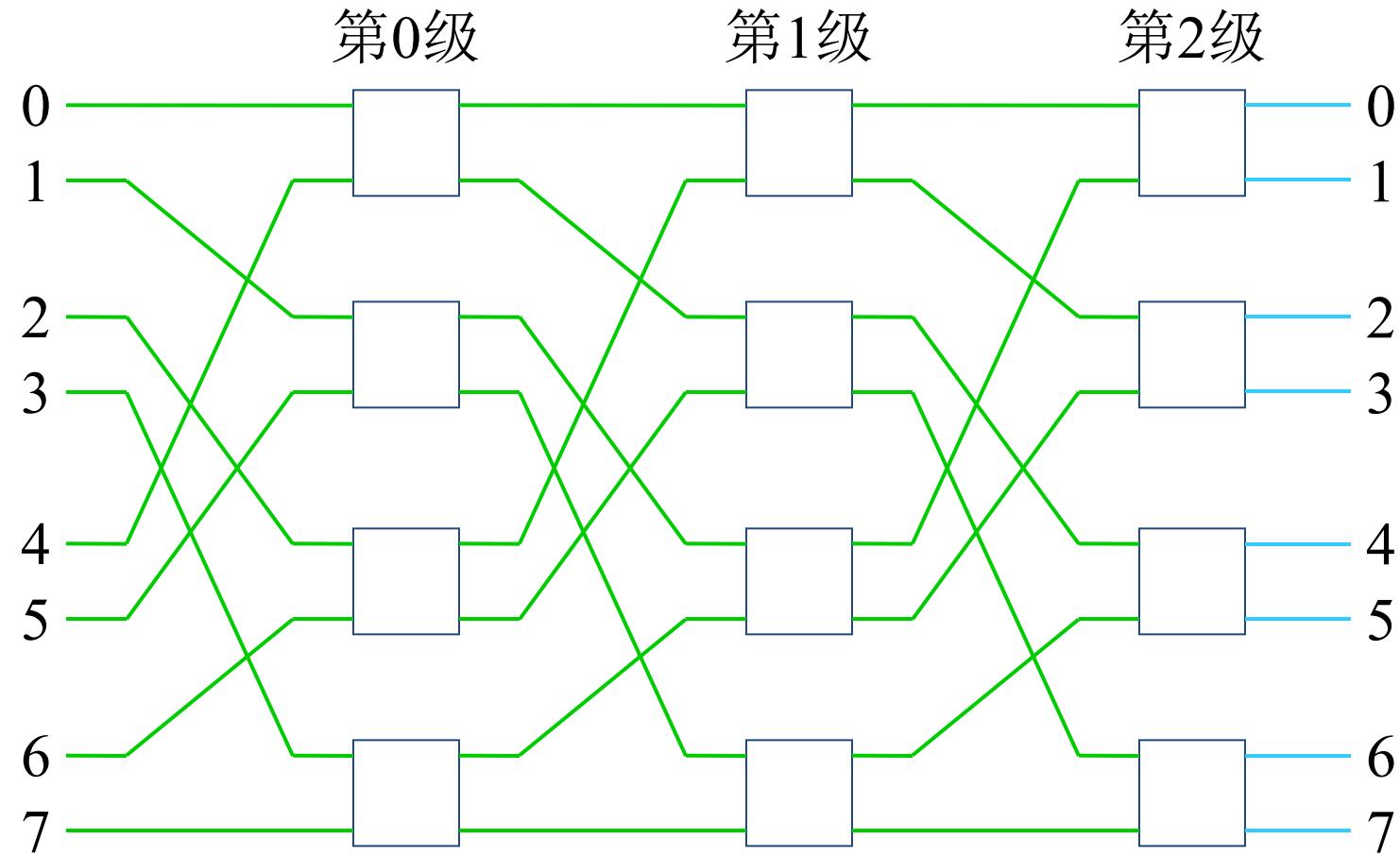
上播



下播

# 动态互连网络

## ■ 多级交换网络



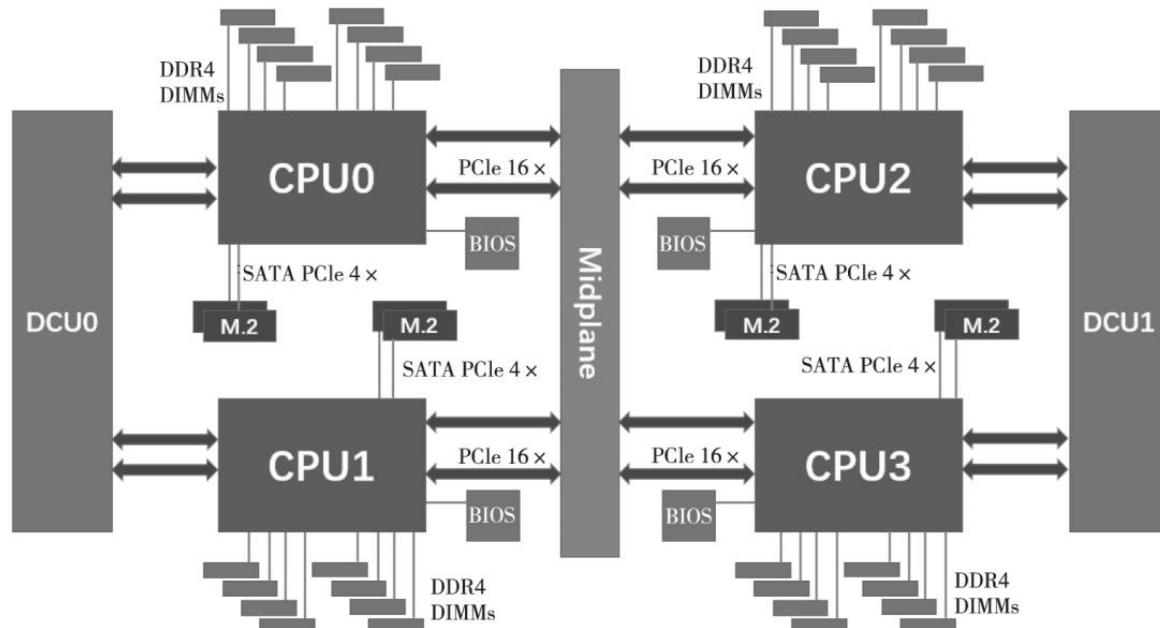
# Reference

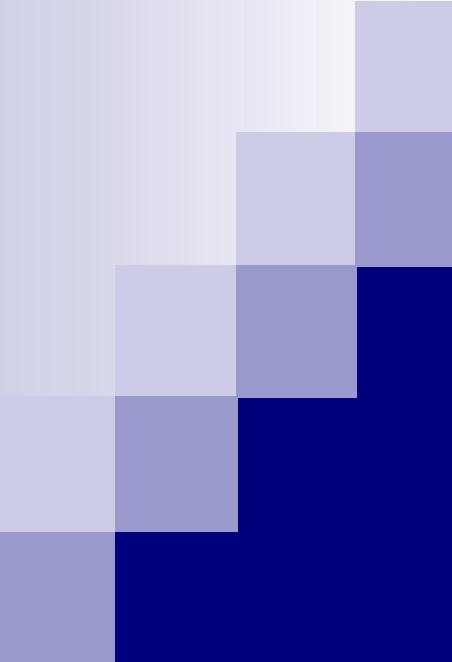
- 并行计算课程, 陈国良, 中国科学技术大学
- <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>
- <https://medium.com/high-tech-accessible/an-overview-of-amds-gpu-architectures-884432a717a6>
- 中国科学技术大学超级计算中心 Intel MIC高性能计算服务器使用指南  
<https://max.book118.com/html/2017/0422/101705121.shtml>
- [https://en.wikipedia.org/wiki/Tensor\\_Processing\\_Unit](https://en.wikipedia.org/wiki/Tensor_Processing_Unit)
- <https://cloud.google.com/tpu/docs/system-architecture-tpu-vm?hl=zh-cn>
- 阳王东, 王昊天, 张宇峰,等. 异构混合并行计算综述. 计算机科学, 2020.

# 国产超算发展

## ■ 曙光E级原型机

- 共有512个节点，1024颗Hygon处理器和512块DCU加速卡。
- 各节点之间使用200Gbps的高速网络，采用6D-Torus的方式实现高维互连。
- 每个节点有2颗Hygon 7185处理器和1块DCU加速卡，256GB的DDR4内存，240GB的M.2 SSD硬盘。





# 第三章

# 并行程序设计基础

哈尔滨工业大学

张伟哲

2025, Fall Semester

# 目录

- PCAM设计原理
- 并行程序性能评价
- 并行程序编程模型

# 目录

- PCAM设计原理
- 并行程序性能评价
- 并行程序编程模型

# 并行算法的一般设计方法

- 第一类：串行算法的直接并行化
- 第二类：从问题描述开始设计并行算法
- 第三类：借用已有算法求解新问题

# 第一类：串行算法的直接并行化

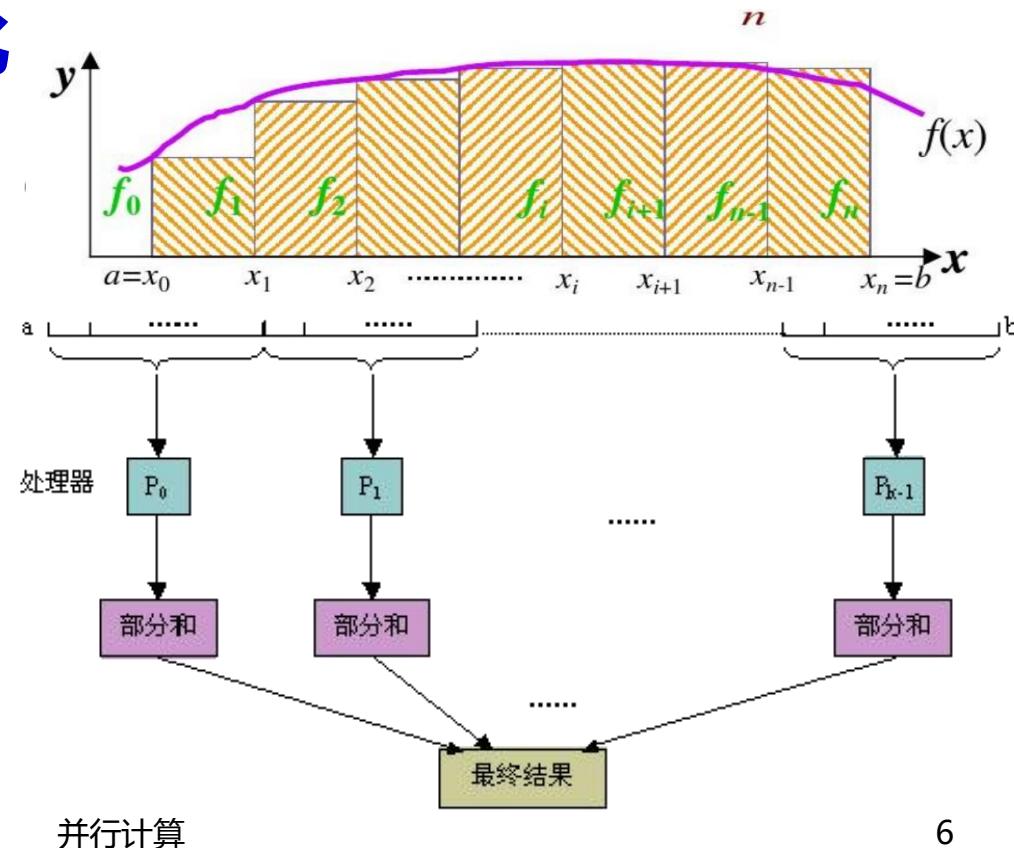
## ■ 方法描述

- 发掘和利用现有串行算法中的并行性，直接将串行算法改造为并行算法
- 许多并行编程语言都支持通过在原有的串行程序中加入并行原语（例如某些通信命令等）的方法将串行程序并行化
- 由串行算法直接并行化的方法是并行算法设计的最常用方法之一
- 不是所有的串行算法都可以直接并行化的（模拟退火算法）
- 一个好的串行算法并不能并行化为好的并行算法
- 许多数值串行算法可以并行化为有效的并行算法

# 第一类：串行算法的直接并行化

- **示例：用求和的方法进行数值积分；设被积函数为 $f(x)$ ，积分区间为 $[a,b]$**
- **串行算法，可以用循环和叠加完成上述求和**
- **可以直接进行并行化**

科学和工程中有大量数值计算问题。针对这些问题，人们已经设计出了许多串行数值计算方法。在设计这些问题的并行算法时，大多采用串行算法直接并行化的方法。这样做的一个显著优点是，算法的稳定性、收敛性等问题在串行算法中已有结论，不必再考虑。



# 第二类：从问题描述开始设计并行算法

- 方法描述

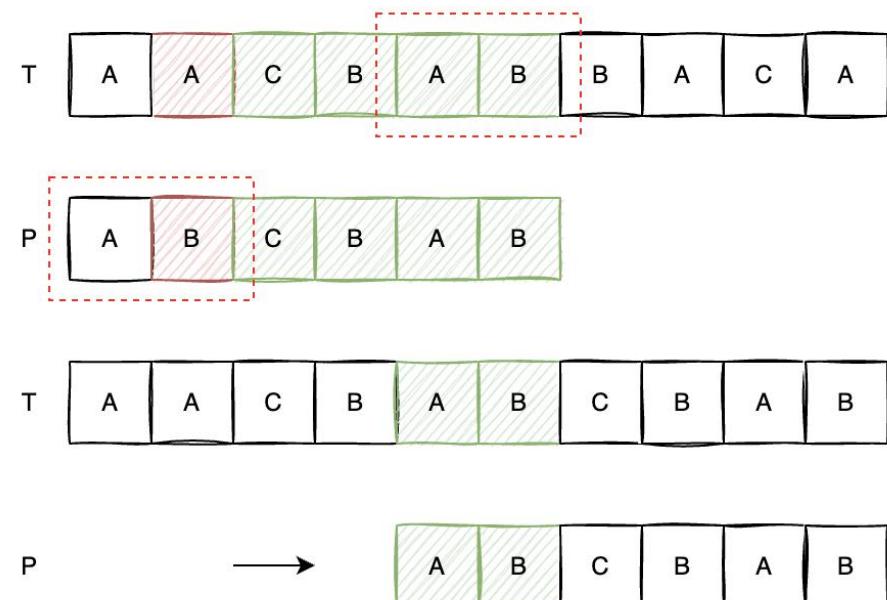
- 从问题本身描述出发，不考虑相应的串行算法，设计一个全新的并行算法

- 挖掘问题的固有特性与并行的关系

- 设计全新的并行算法是挑战性和创造性的工作

# 第二类：从问题描述开始设计并行算法

- **示例：**串中包含的字符的个数称为串的长度。给定长度为 $n$ 的正文串T和长度为 $m$ 的模式串P，找出P在T中所有出现的位置称为**串匹配问题**
- 目前已知的有效的串匹配算法均不易直接并行化，需要重新设计并行算法



# 第三类：借用已有算法求解新问题

## ■ 方法描述

- 找出求解问题和某个已解决问题之间的联系
- 改造或利用已知算法应用到求解问题上

## ■ 这是一项创造性的工作

## ■ 需要很高的技巧，同时算法设计者要有敏锐的观察力并且在并行算法方面有丰富的经验

# 第三类：借用已有算法求解新问题

- **示例：使用矩阵乘法算法求解所有点对间最短路径是一个很好的范例**
- **问题描述：**设在一有向图中，各弧都赋予了非负整数权。图中一条路径的长度定义为该路径上所有的弧的权的和。图中两结点之间的最短路径是指它们之间长度最短的路径

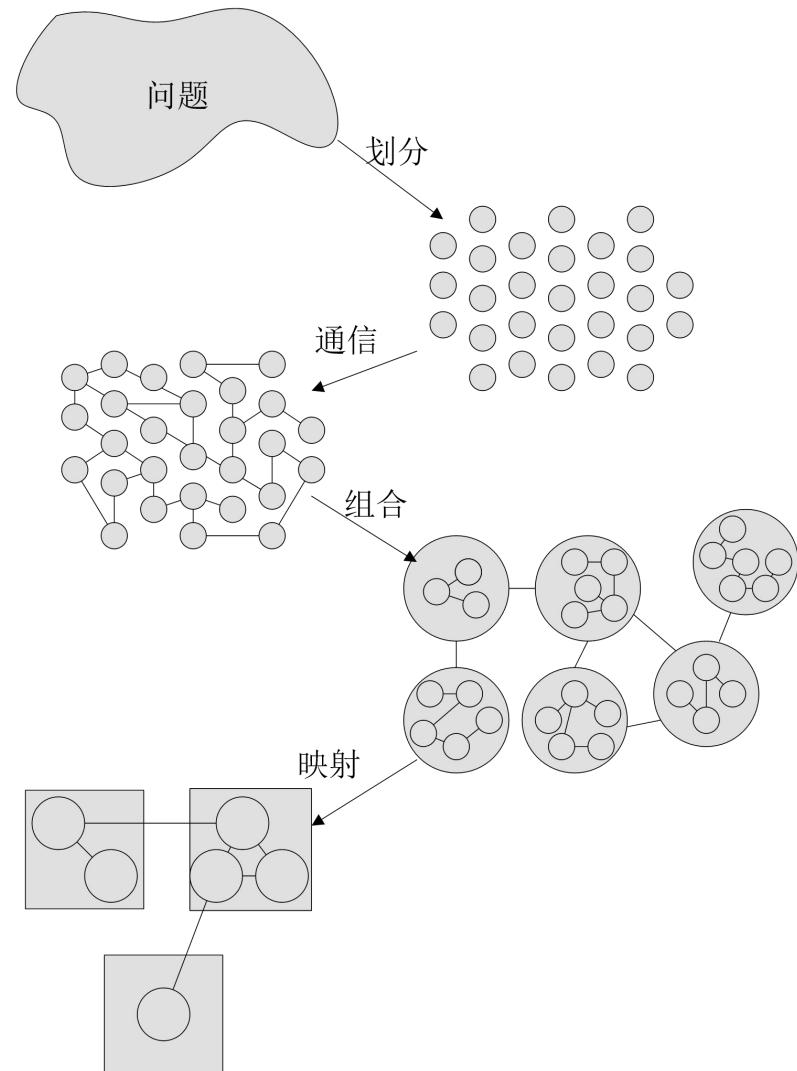


# 并行算法的设计过程

- 目的：设计出一个具有并发性、可扩展性、局部性和模块性的并行算法
- PCAM设计方法
  - 任务划分 (Partitioning)
  - 通信分析 (Communication)
  - 任务组合 (Agglomeration)
  - 处理器映射 (Mapping)

# PCAM设计方法

- **划分:** 将整个计算任务分解成一些小任务，其目的是尽量开拓并行执行的可能性
- **通信:** 确定小任务执行中需要进行的通信，为组合做准备
- **组合:** 按性能要求和实现的代价来考察前两阶段的结果，适当地将一些小任务组合成更大的任务以提高性能、减少通信开销
- **映射:** 将组合后的任务分配到处理器上，其目标是使全局执行时间和通信开销尽量小，使处理器的利用率尽量高



# 划分方法描述

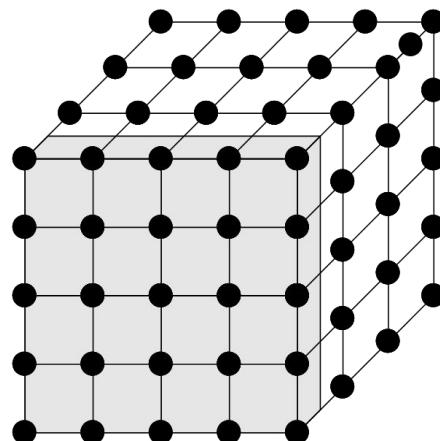
- 充分开拓算法的并发性和可扩展性
- 先进行数据分解（称域分解），再进行计算功能的分解（称功能分解）
- 划分阶段忽略处理器数目和目标机器的体系结构
- 能分为两类划分
  - 域分解 (domain decomposition)
  - 功能分解 (functional decomposition)

# 划分：域分解

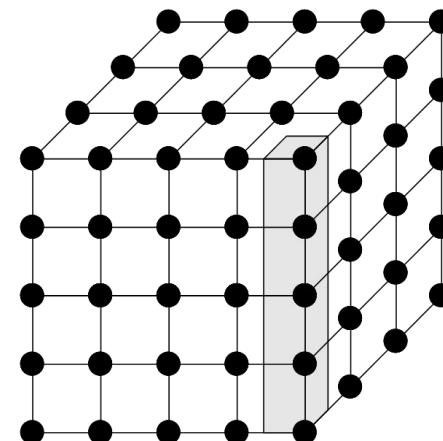
- 域分解也叫**数据划分**，划分的对象是数据。这些数据可以是算法（或程序）的输入数据、计算的中间结果或计算的输出数据
- 域分解的步骤
  - 分解与问题相关的数据，如果可能的话，应使每份数据的数据量大体相等
  - 再将每个计算关联到它所操作的数据上
  - 由此就产生出一些任务，每个任务包括一些数据及其上的操作
  - 当一个操作需要别的任务中的数据时，就会产生通信要求

# 划分：域分解

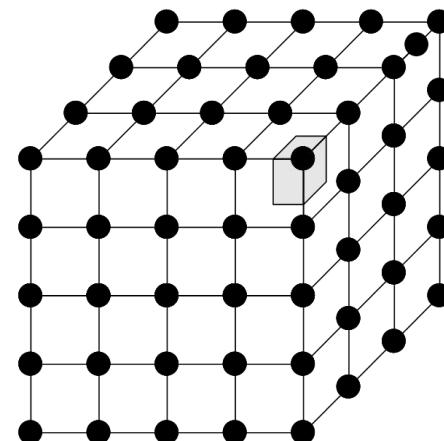
- 域分解示例：三维网格的域分解，各格点上计算都是重复的



1 - D



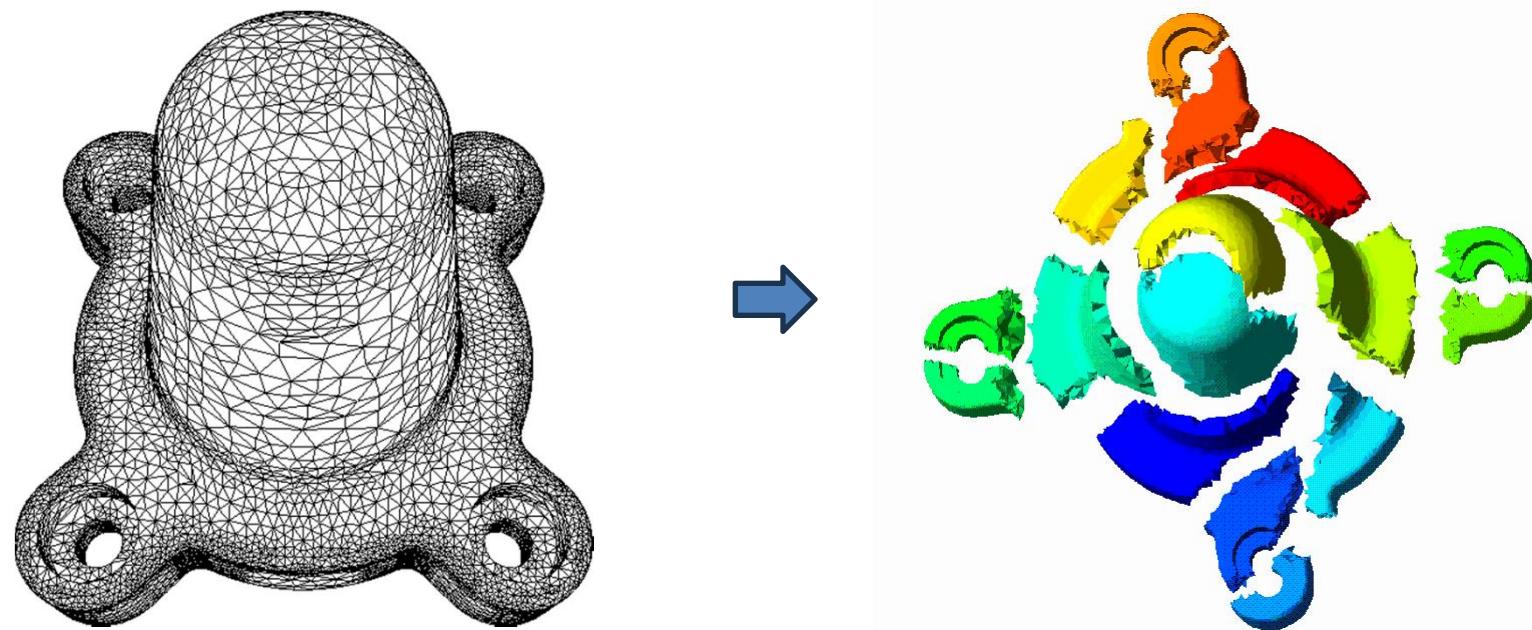
2 - D



3 - D

# 划分：域分解

## ■ 不规则区域的分解示例

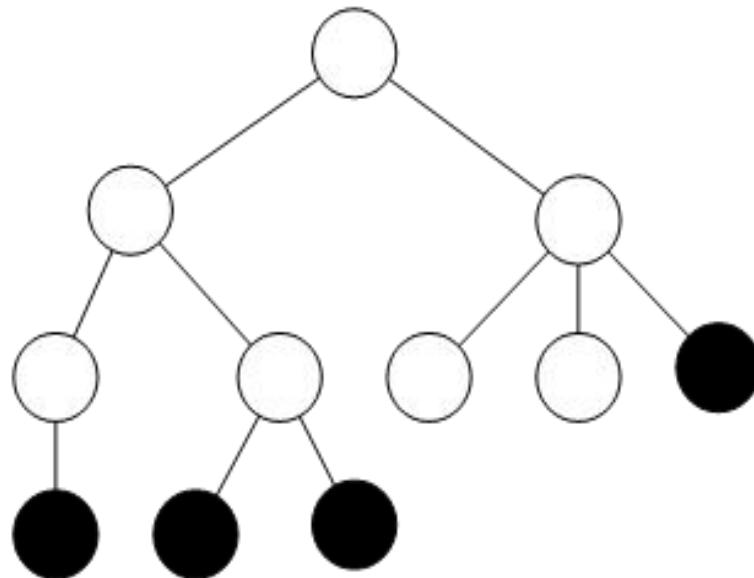


# 划分：功能分解

- 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
- 功能分解的步骤
  - 首先关注被执行的计算的分解，而不是计算所需数据
  - 如果所作的计算划分是成功的，再继续研究计算所需的数据
  - 如果这些数据是不相交或相交很少的，就意味着划分是成功的；如果这些数据有相当的重叠，就会产生大量的通信，此时就暗示应考虑数据分解

# 划分：功能分解

- 功能分解示例：搜索树
- 搜索树没有明显的可分解的数据结构，但易于进行细粒度的功能分解



- 开始时根结点生成一个任务
- 对其评价后，如果它不是一个解，就生成若干叶结点
- 这些叶结点可以分到各个处理器上并行地继续搜索

# 划分：判断依据

## ■ 下面的问题可以帮助检查所作的划分是否合理

检查项	问题
✓ 所划分的任务数是否高于目标机上处理器数目一个量级？	若不是，在后面的设计步骤中将缺少灵活性
✓ 划分是否避免了冗余的计算和存储要求？	若不是，则产生的算法对大型问题可能不是可扩展的
✓ 各任务的尺寸是否大致相当？	若不是，则分配处理器时很难做到负载平衡
✓ 划分的任务数是否与问题尺寸成比例？	理想情况下，问题尺寸的增加应引起任务数的增加而不是任务尺寸的增加。若不是这样，算法可能不能求解更大的问题，尽管有更多的处理器
✓ 是否采用了几种不同的划分法？	多考虑几种选择可以提高灵活性。同时既要考虑域分解又要考虑功能分解

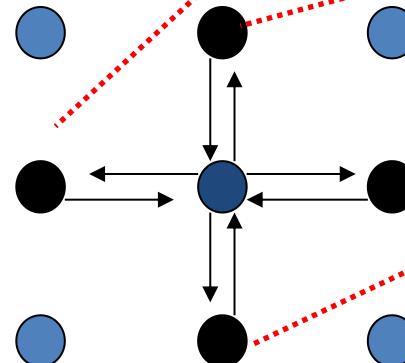
# 通信方法描述

- 由划分产生的各任务一般都不能完全独立地执行
- 各任务之间需要交换数据和信息，这就产生了通信的要求
- 通信就是为了进行并行计算，诸任务之间所需进行的数据传输
- 诸任务是并发执行的，通信则限制了这种并发性
- 通信的四种模式
  - 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

# 通信：局部通信

- 局部通信中，每个任务只与少数的几个近邻任务通信

$$x_{i,j}(k) = \frac{4x_{i,j}(k-1) + x_{i-1,j}(k-1) + x_{i+1,j}(k-1) + x_{i,j-1}(k-1) + x_{i,j+1}(k-1)}{8}$$



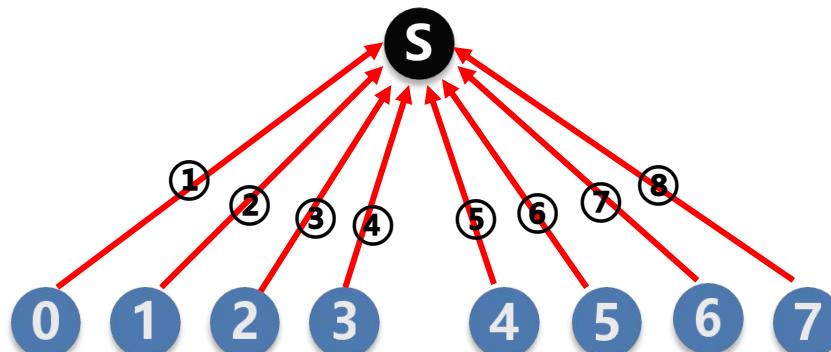
- 例：数值计算中的雅可比有限差分法
- 处于 $(i, j)$ 位置上的处理器负责计算  $x_{i,j}$
- 计算每个 $x_{i,j}(k)$ 时， $(i, j)$ 位置上的处理器只需与其上、下、左、右的邻居处理器通信
- 在获得数据的同时把自己的 $x_{i,j}(k-1)$ 发送给邻居处理器

# 通信：全局通信

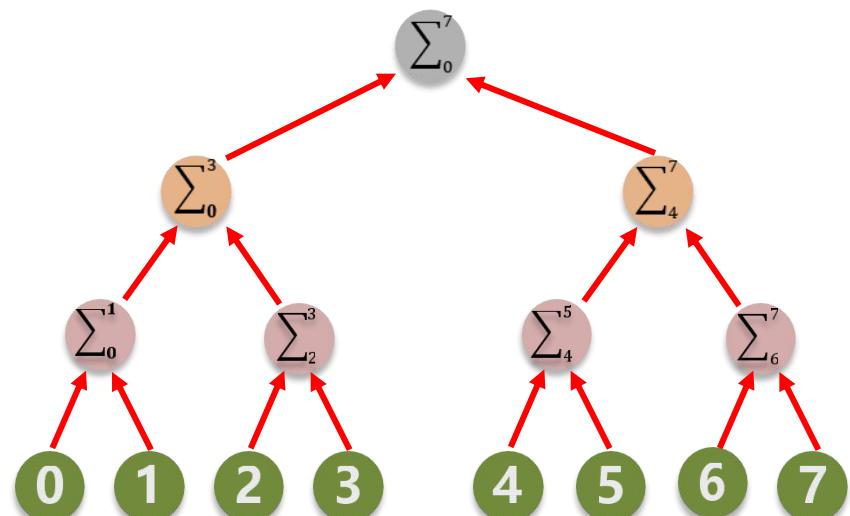
- 全局通信中，每个任务要与很多别的任务通信

$$S = \sum_{i=0}^{N-1} x_i$$

使用一个根进程负责从各个进程一次接收一个值( $x_i$ )，并进行累加，这时出现全局通信的局面



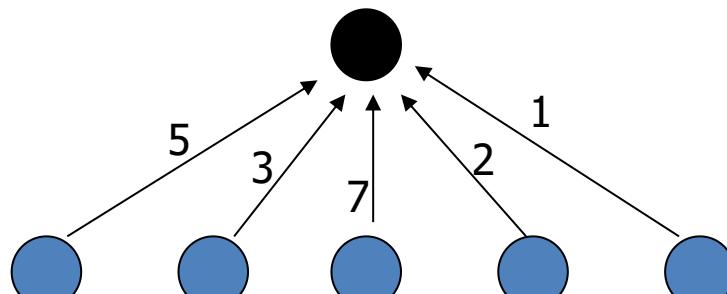
全局通讯  
Master-Worker



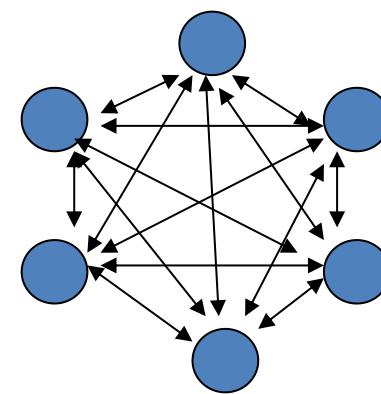
$$S = \sum_{i=0}^{2^n-1} x_i = \sum_{i=0}^{2^{n-1}-1} x_i + \sum_{i=2^{n-1}}^{2^n-1} x_i$$

# 通信：全局通信

- 全局通信中，每个任务要与很多别的任务通信



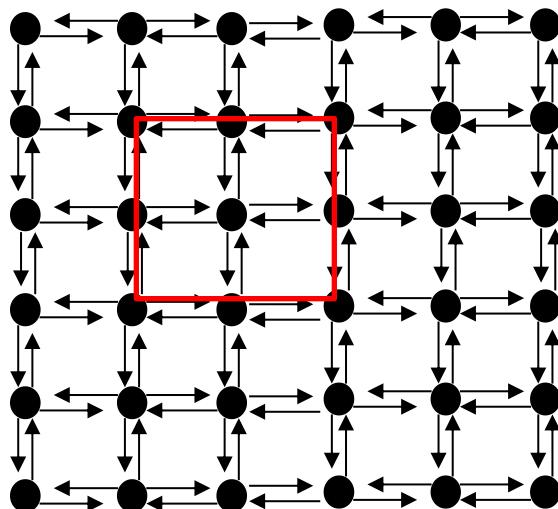
全局通讯  
Master-Worker



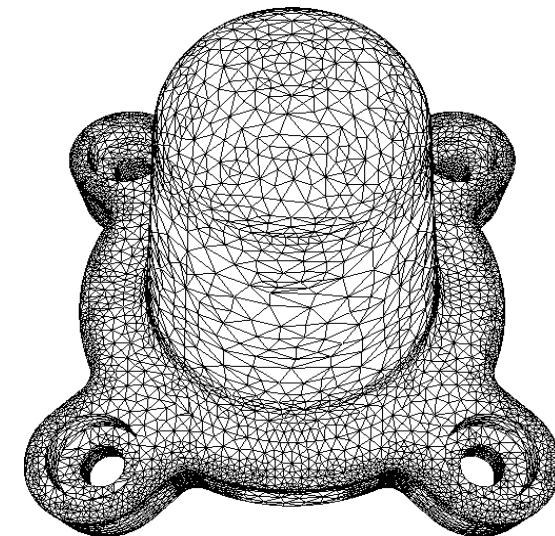
全局通讯  
All-to-All

# 通信：结构化/非结构化通信

- 结构化通信中，一个任务和其近邻形成规则的结构（如树、网格等）
- 非结构化通信中，通信网可能是任意图



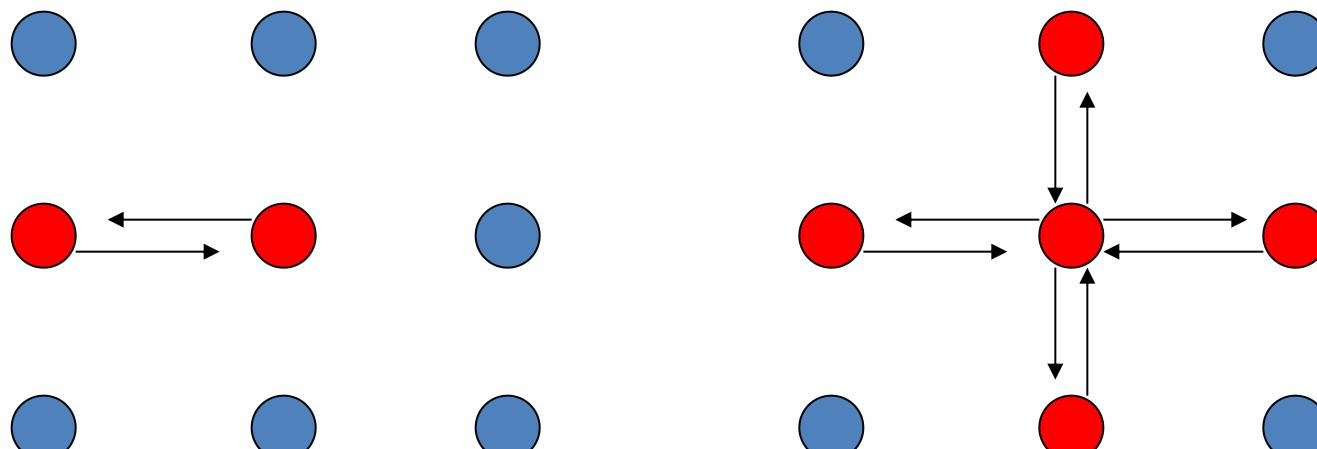
结构化通讯



非结构化通讯

# 通信：静态/动态通信

- 静态通信中，通信伙伴不随时间变化
- 动态通信中，通信伙伴可能动态变化，可以由运行时计算的数据确定

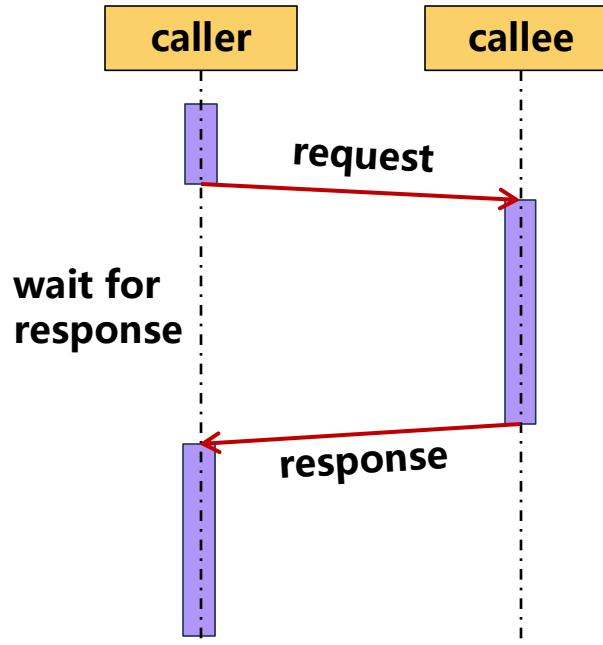


静态通讯

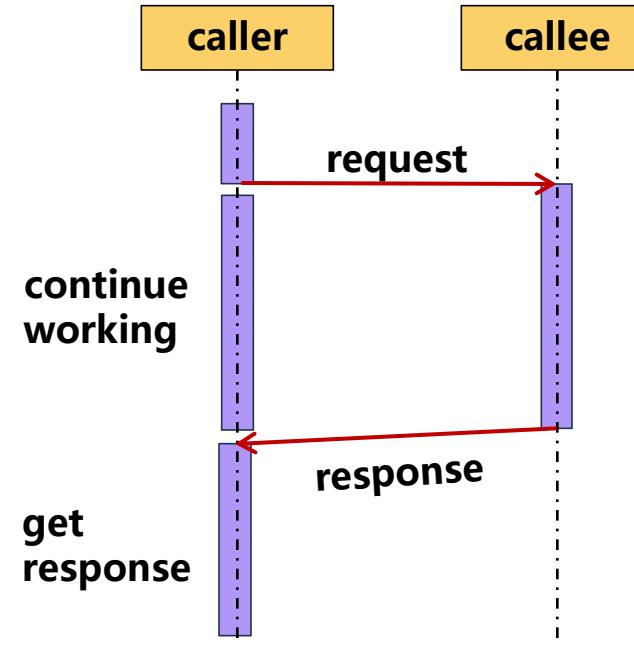
动态通讯

# 通信：同步/异步通信

- 同步通信中，接收方和发送方协同操作
- 异步通信中，接收方获取数据无需与发送方协同



同步通讯



异步通讯

# 通信：判断依据

## ■ 下面的问题可以帮助检查通信设计是否合理

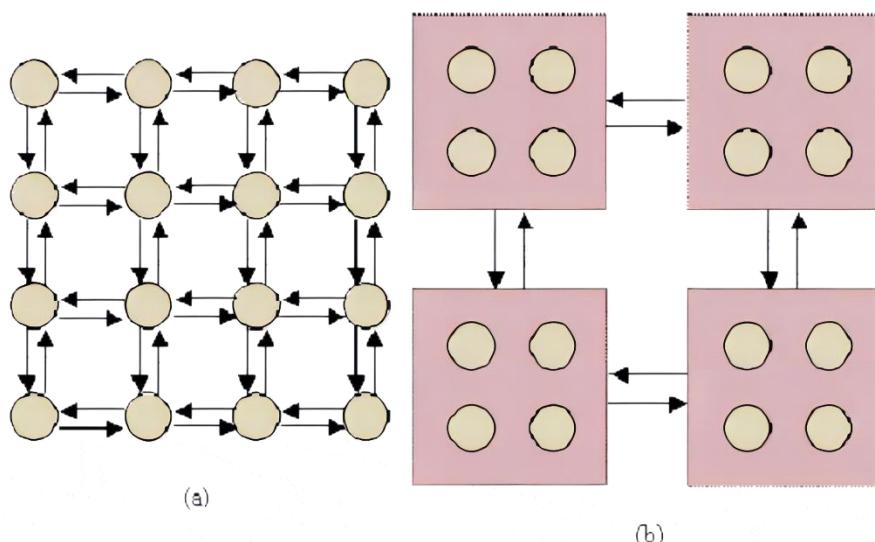
检查项	问题
✓ 所有任务是否执行大致同样多的通信？	若不是，所设计的算法的可扩展性可能会不好
✓ 每个任务是否只与少数的近邻通信？	若不是，则可能导致全局通信。此时应设法将全局通信换成局部通信
✓ 诸通信操作能否并行执行？	若不能，所设计的算法可能是低效的和不具可扩展性的。此时可试用分治策略来开发并行性
✓ 不同任务的计算能否并行执行？	若不能并行执行，所设计的算法可能是低效的和不具可扩展性的。此时可考虑重新安排通信和计算的顺序以改善这种情况
✓ 是否会因为等待数据而降低并行度？	

# 组合方法描述

- 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
- 目的：合并小尺寸任务，减少任务数量和通信开销
- 通过增加任务的粒度和重复计算，减少通讯成本
- 保持映射和扩展的灵活性，降低软件工程成本
- 组合需要考虑两个因素
  - 表面-容积效应
  - 重复计算

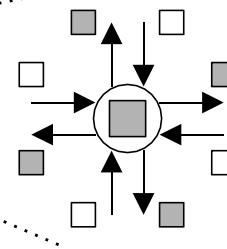
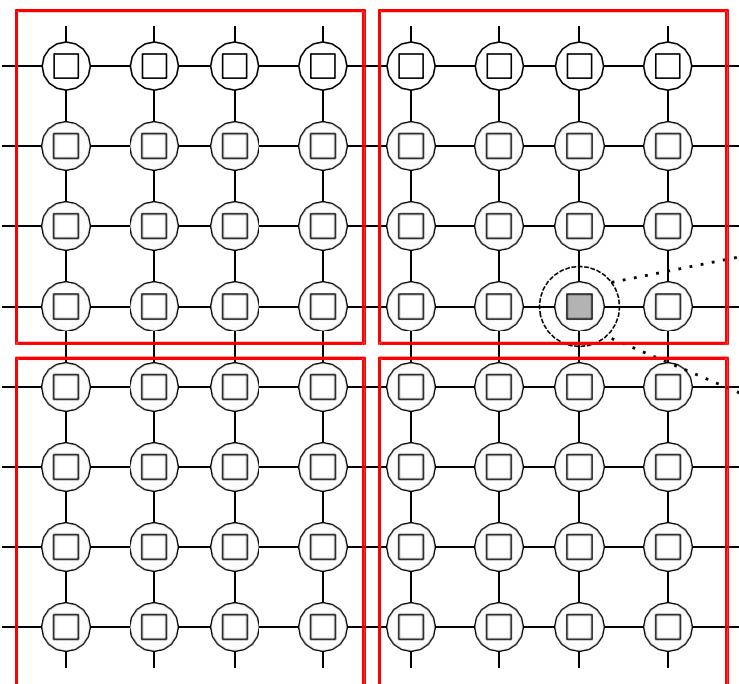
# 组合：表面-容积效应

- 任务的通信需求正比于它所操作的数据域的表面积，计算需求正比于它所操作的数据域的容积
- 计算单元的通信与计算之比随任务尺寸的增加而减小



- 假设需要计算的数据是 $4 \times 4$ 矩阵。如果把计算每个元素算作一个任务，则有16个任务。每轮迭代中，每个任务都需要与其上下左右的任务通信，共需48次通信
- 将相邻的四个元素的计算作为一个任务则只需8次通信

# 组合：表面-容积效应



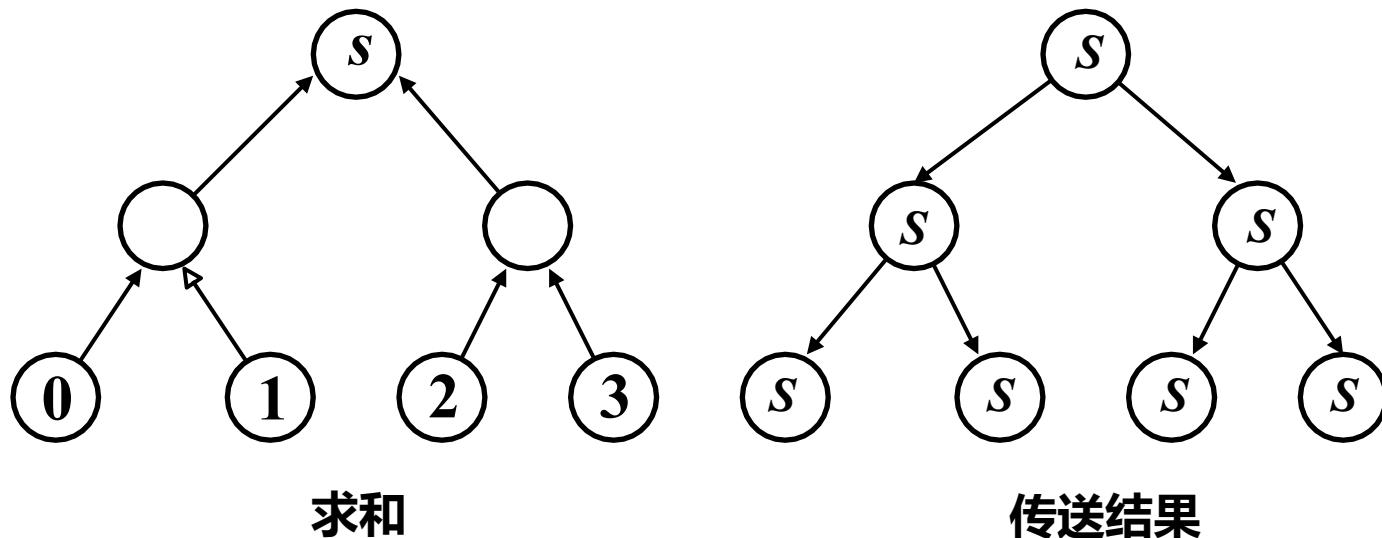
- **细粒度二维网 (8\*8网络)**
  - $8*8=64$ 个任务
  - $64*4=256$ 次通信
- **每个任务通信4次，共传输256个数据**
  
- **粗粒度二维网**
  - $2*2=4$ 个任务
  - $4*4=16$ 次通信
- **每个任务通信4次，共传输16个数据**

# 组合：重复计算

- 采用冗余的计算来减少通信和/或整个计算时间
- 重复计算减少通讯量，但增加了计算量，应保持恰当的平衡，最终目标应减少算法的总运算时间

# 组合：重复计算

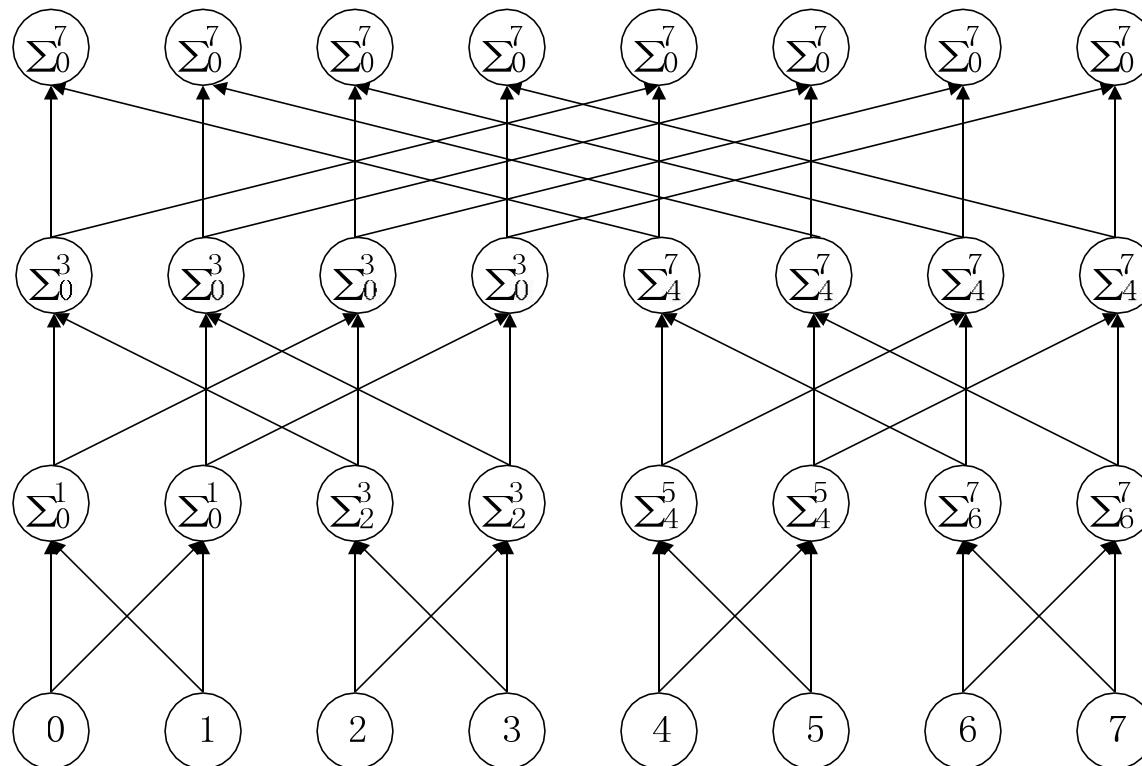
- 示例：假定在二叉树上求 $N$ 个数的和，且要求最终在每个处理器上都有该结果



先自叶向根求和，得到结果后再自根向叶播送，  
共需 $2\log N$ 步

# 组合：重复计算

- 示例：假定在二叉树上求N个数的和，且要求最终在每个处理器上都有该结果



蝶式结构求和，使用了重复计算，共需 $\log N$ 步

# 组合：判断依据

## ■ 下面的问题可以帮助检查所进行的组合是否合理

检查项
✓ 增加粒度是否减少了通讯开销?
✓ 重复计算是否已权衡了其得益?
✓ 是否保持了灵活性和可扩展性?
✓ 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讯?
✓ 有没有减少并行执行的机会?

# 映射方法描述

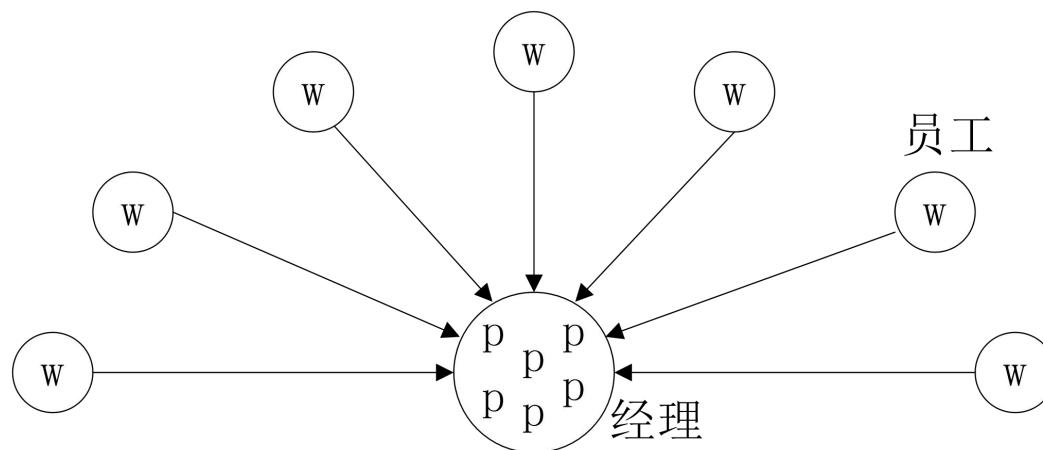
- 任务要映射到具体的处理器，定位到运行机器上
- 目标：最小化全局执行时间和通信成本，最大化处理器的利用率，减少算法的总执行时间
  - 把并发执行的任务放在不同的处理器上以增加并行度
  - 把需频繁通信的任务置于同一处理上以提高局部性
- 映射实际是一种权衡，属于NP完全问题
- 映射过程所用到的两类算法
  - 负载平衡算法
  - 任务调度算法

# 映射：负载均衡算法

- 任务的工作量不同，通信是非结构化的，可采用负载平衡算法
- 局部算法：通过从近邻迁入任务和向近邻迁出任务来达到负载平衡
- 概率方法：将任务随机地分配给处理器，如果任务足够多，则每个处理器预计能分到大致等量的任务
- 循环映射：轮流地给处理器分配计算任务

# 映射：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
- **经理/雇员模式**：一个进程（经理）负责分配任务，每个雇员向经理请求任务，得到任务后执行任务
- **非集中模式**：无中心管理者的分布式调度法



# 映射：判断依据

## ■ 下面的问题可以帮助检查映射设计得是否合理

### 检查项

- ✓ 如果采用集中式负载均衡方案，是否检查了中央管理者不会成为瓶颈？
- ✓ 如果采用动态负载均衡方案，是否衡量过不同策略的成本？
- ✓ 如果采用概率或循环映射法，是否有足够多的任务？一般地，任务数应不少于处理器数的10倍。

# PCAM设计原理小结

- **划分**: 域分解和功能分解
- **通讯**: 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，并保持负载均衡

# 目录

- PCAM设计原理
- 并行程序性能评价
- 并行程序编程模型

# 加速比 (Speedup)

## ■ 程序加速比计算

- $T_s$  : 使用最佳串行算法的执行时间
- $T_p$  : 使用p个处理器时的执行时间

$$S(p) = \frac{T_s}{T_p}$$

## ■ 线性加速比

- 理论上的最大加速比 (理想)  $S(p) = p$

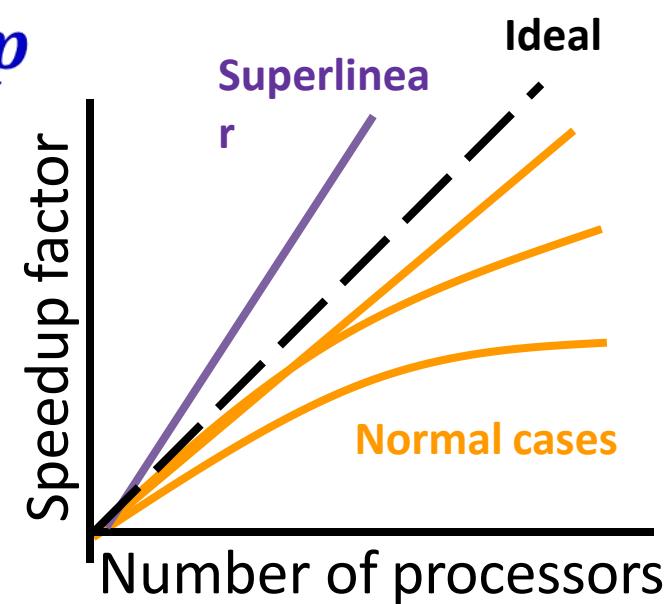
## ■ 超线性加速比

- 实践中偶尔会发生
- 额外的硬件资源 (如内存)
- 软/硬件深度优化 (如缓存)

$$S(p) > p$$

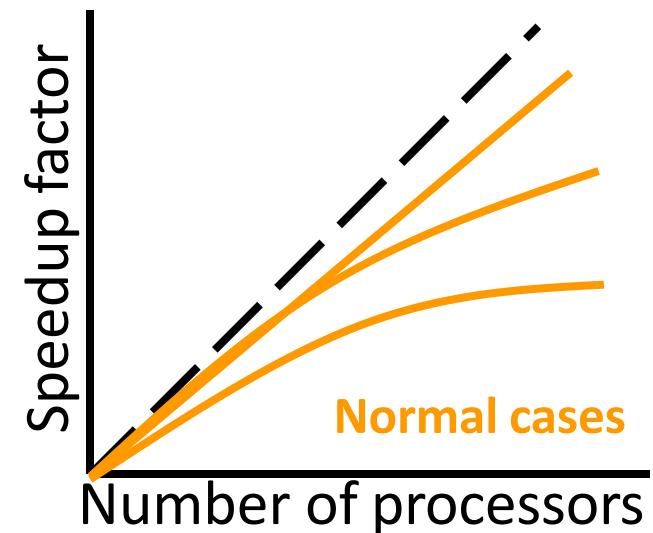
## ■ 系统效率 (Efficiency)

$$E(p) = \frac{T_s}{T_p \times p} = \frac{S(p)}{p} \times 100\%$$



# 加速比 (Speedup)

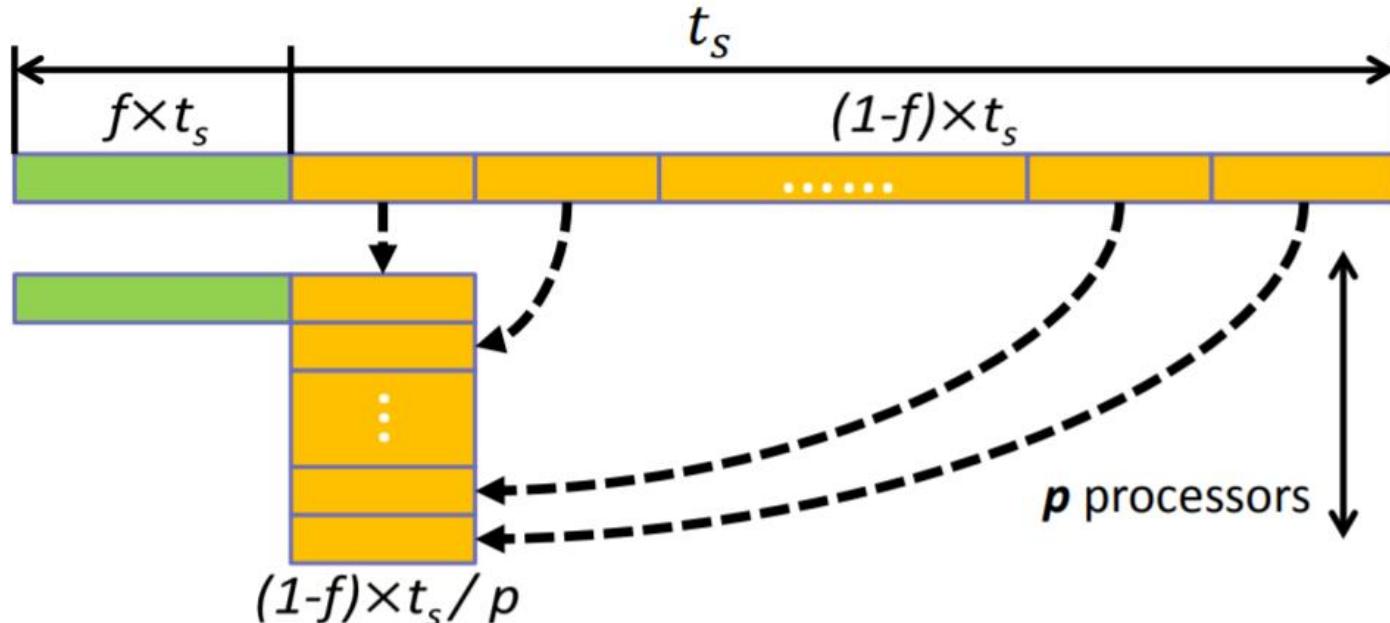
- 理想（最大）加速比很难达到  $S(p) = p$
- 程序中并非每个部分的计算都可以并行化，使得处理器空闲
- 并行版本中需要额外的计算（即用于的同步成本）
- 进程之间需要通讯（通讯成本，通常是主要因素）



# Amdahl定律

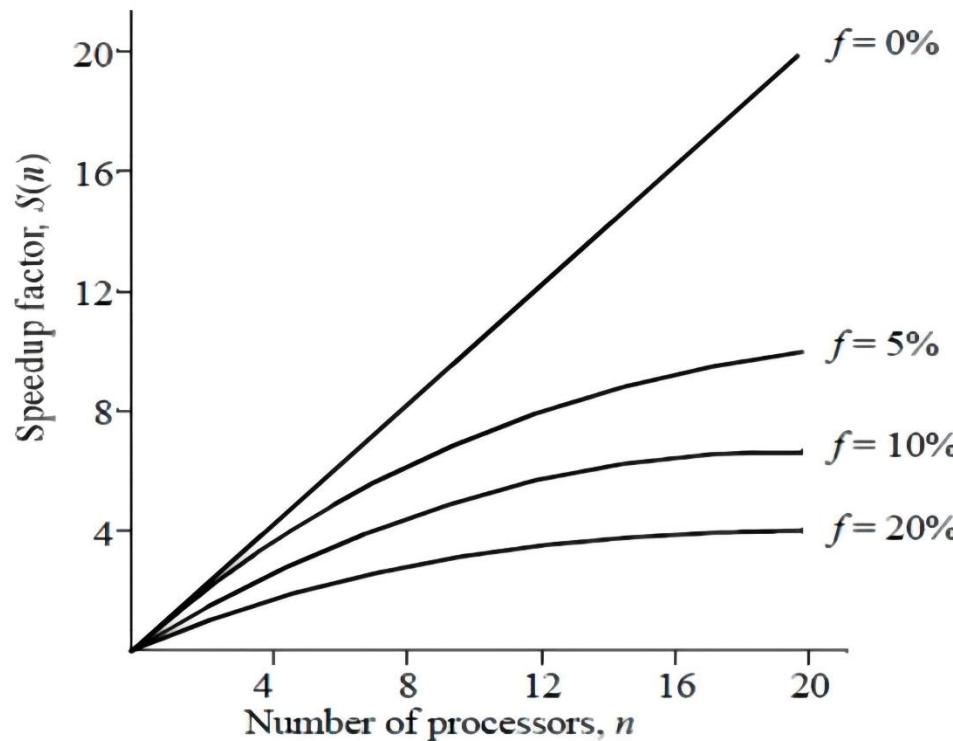
- Amdahl定律定义了串行程序并行化后加速比计算公式与理论上限
- $f$  表示程序中不可以被并行化的部分所占的比例

$$S(p) = \frac{t_s}{f \times t_s + (1 - f) \times t_s/p} = \frac{p}{1 + (p - 1) \times f}$$



# Amdahl定律

- 处理器数量无穷大时,  $S(p)_{p \rightarrow \infty} \frac{p}{1+(p-1) \times f} = \frac{1}{f}$
- 并行代码所占的百分比固定的情况下, 随着处理器数量的增加, 对并行效率的提升会固定在一定比例



# Amdahl定律

- 固定不变的计算负载
- 固定的计算负载分布在多个处理器上的
- 增加处理器加快执行速度，从而达到加速的目的

# Gustafson定律

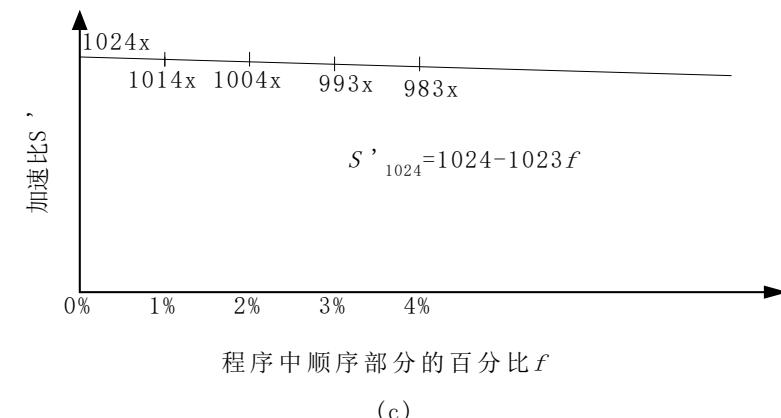
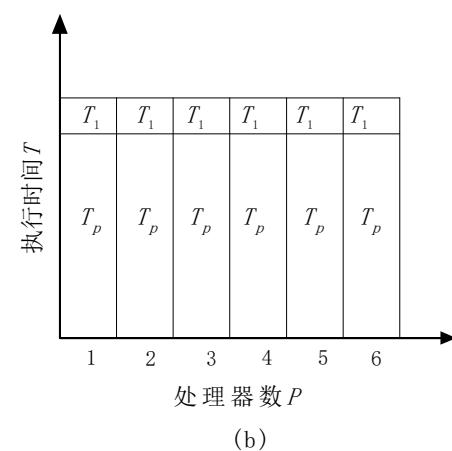
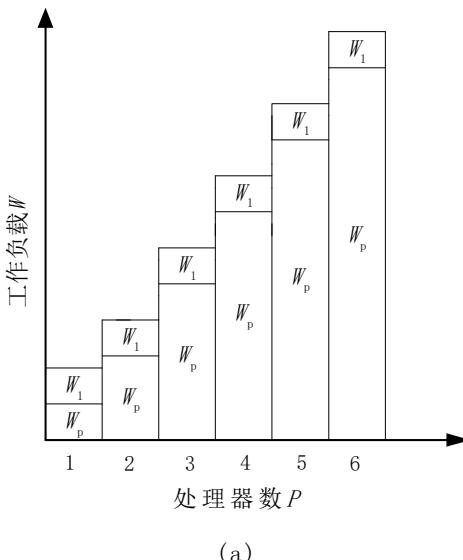
- Amdahl 定律有一个重要前提，就是处理的数据集大小是固定的，但是这在大数据计算的领域里，这个假设并不经常能达到，因为人们总是会为了在短时间内处理更多的数据
- 很多大型计算，精度要求很高，即在此类应用中精度是一个关键因素，而计算时间是固定不变的。此时为了提高精度，必须加大计算量，相应的也必须增加处理器的数目来完成这部分计算，以保持计算时间不变
- 研究在给定的时间内用不同数目的处理器能够完成多大的计算量是并行计算中一个很实际的问题

# Gustafson定律

- $t_s, t_p$  表示串行部分和并行部分执行的时间
- $f$  表示程序中不可以被并行化的部分的所占的比例

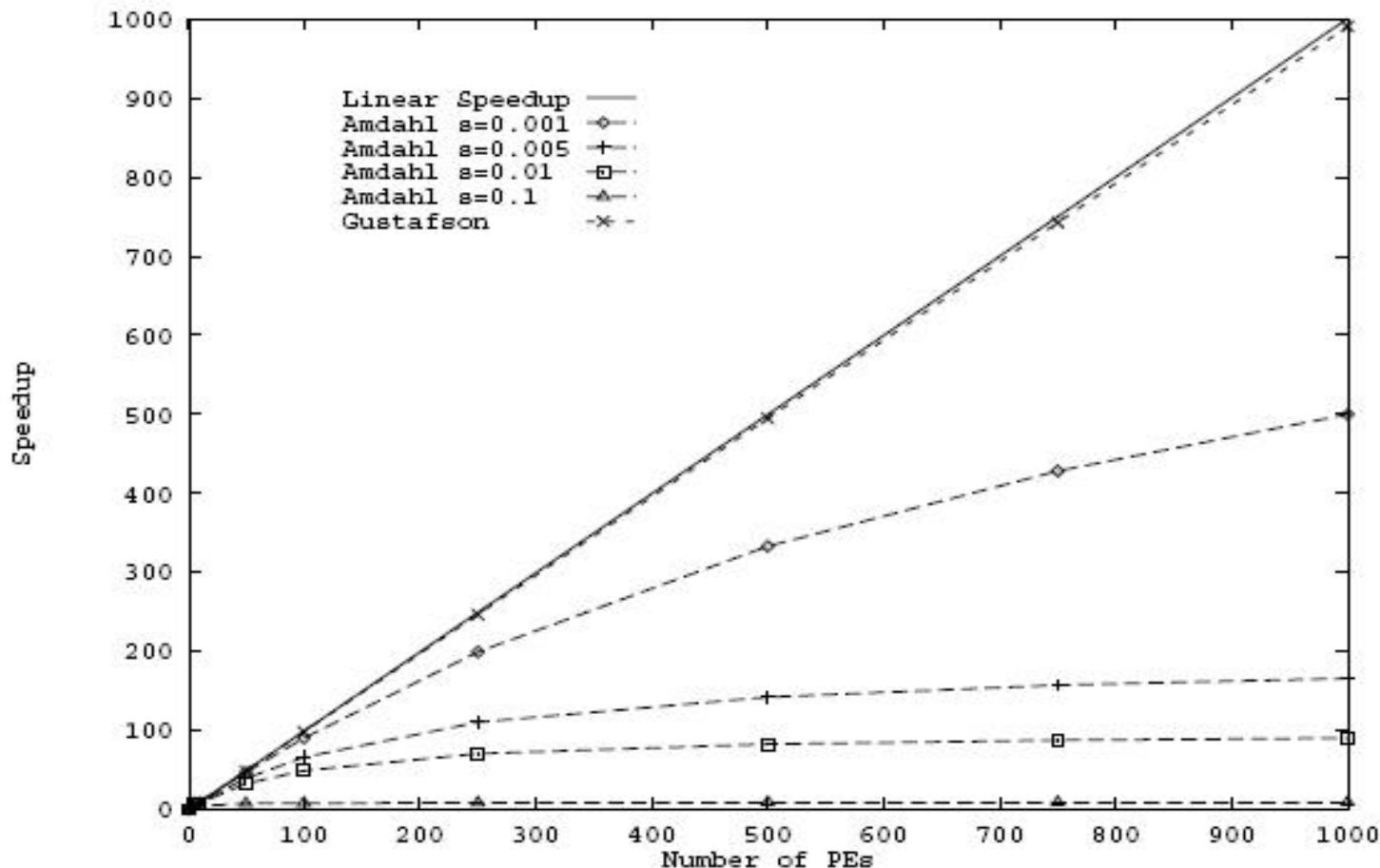
$$f = \frac{t_s}{t_s + t_p}$$

$$\blacksquare S(p) = \frac{t_s + p \times t_p}{t_s + p \times t_p / p} = \frac{t_s + p \times t_p}{t_s + t_p} = f + p \times (1 - f)$$



# Amdahl定律 vs Gustafson定律

■ 刻画的内容等价，可相互推导

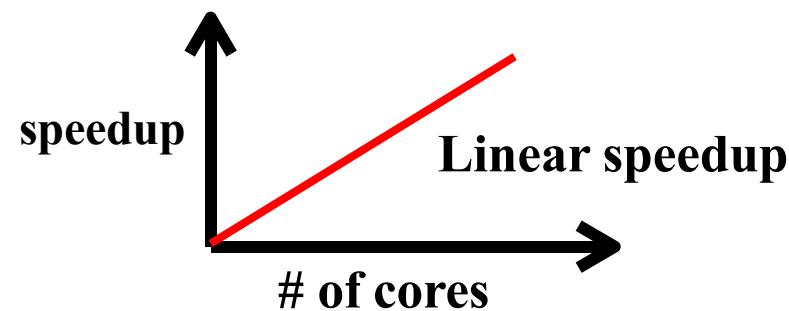
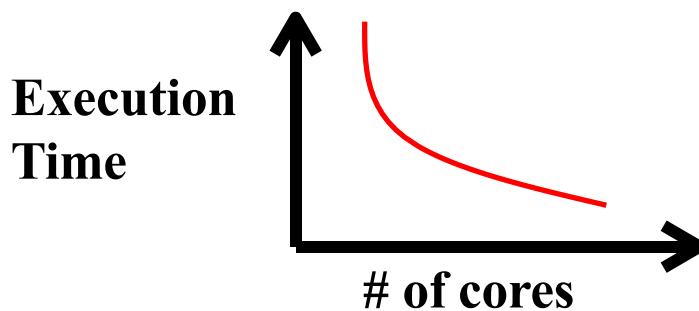


# 可扩展性 (Scalability)

- 确定的应用背景下，计算机系统（或算法或程序等）性能随处理器数的增加而按比例提高的能力
- 可扩展性分类
  - 强可扩展性 (Strong Scalability)
  - 弱可扩展性 (Weak Scalability)

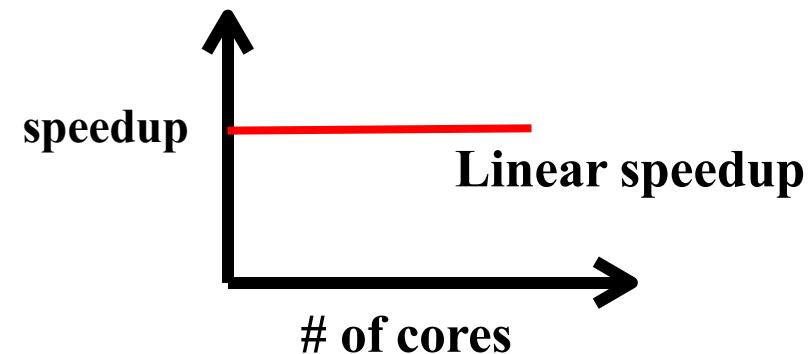
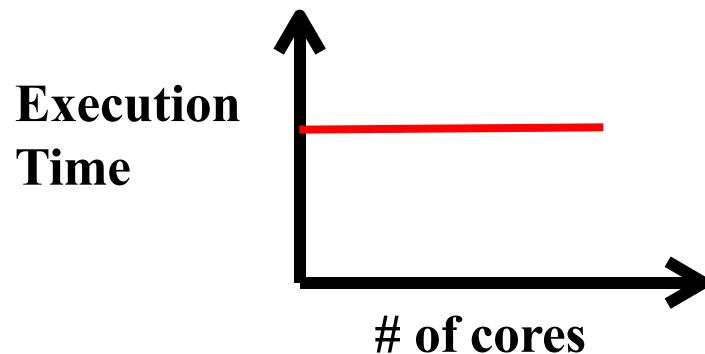
# 强可扩展性 (Strong Scalability)

- 问题规模不变，处理器的数量增加
- 用于寻找一个平滑点，可以在可接受的时间内完成计算（用多少台机器解决这个问题是最好的）
- 如果加速比等于处理器的数量，则称为线性扩展 (Linear Scaling)



# 弱可扩展性 (Weak Scalability)

- 随着处理器的增加，问题的规模同比增加
- 目标是在相同的时间内解决更大规模的问题
- 随着负载的增加，执行时间保持不变，则称为线性扩展

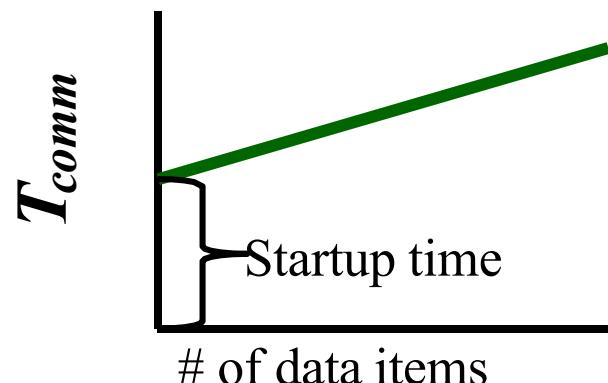


# 强可扩展性 vs 弱可扩展性

- 强可扩展性：线性扩展难实现，因为通讯开销可能与系统规模（节点数、处理器数）同比例增加
- 弱可扩展性：线性扩展更容易实现，因为程序通常采用最近邻通信模式，其中通信开销相对恒定，而与使用的进程数无关

# 时间复杂度 (Time Complexity)

- $T_p = T_{comp} + T_{comm}$ 
  - $T_p$  并行算法的总体执行时间
  - $T_{comp}$  计算部分时间
  - $T_{comm}$  通讯部分时间
  
- $T_{comm} = q(T_{startup} + nT_{data})$ 
  - $T_{startup}$  信息延迟 (假设固定值)
  - $T_{data}$  单位数据的传输时间
  - $n$ : 信息中的数据量
  - $q$ : 信息的数量



# 时间复杂度——示例1

## ■ 算法步骤

- Step 1: 节点1发送 $n/2$ 个数字给节点2
- Step 2: 两个节点同时对 $n/2$ 个数字进行计算求和
- Step 3: 节点2发送部分和的计算结果给节点1
- Step 4: 节点1将部分和相加得出最终结果

## ■ 时间复杂度分析

- Computation(for step 2 & 4)

$$T_{comp} = n/2 + 1 = O(n)$$

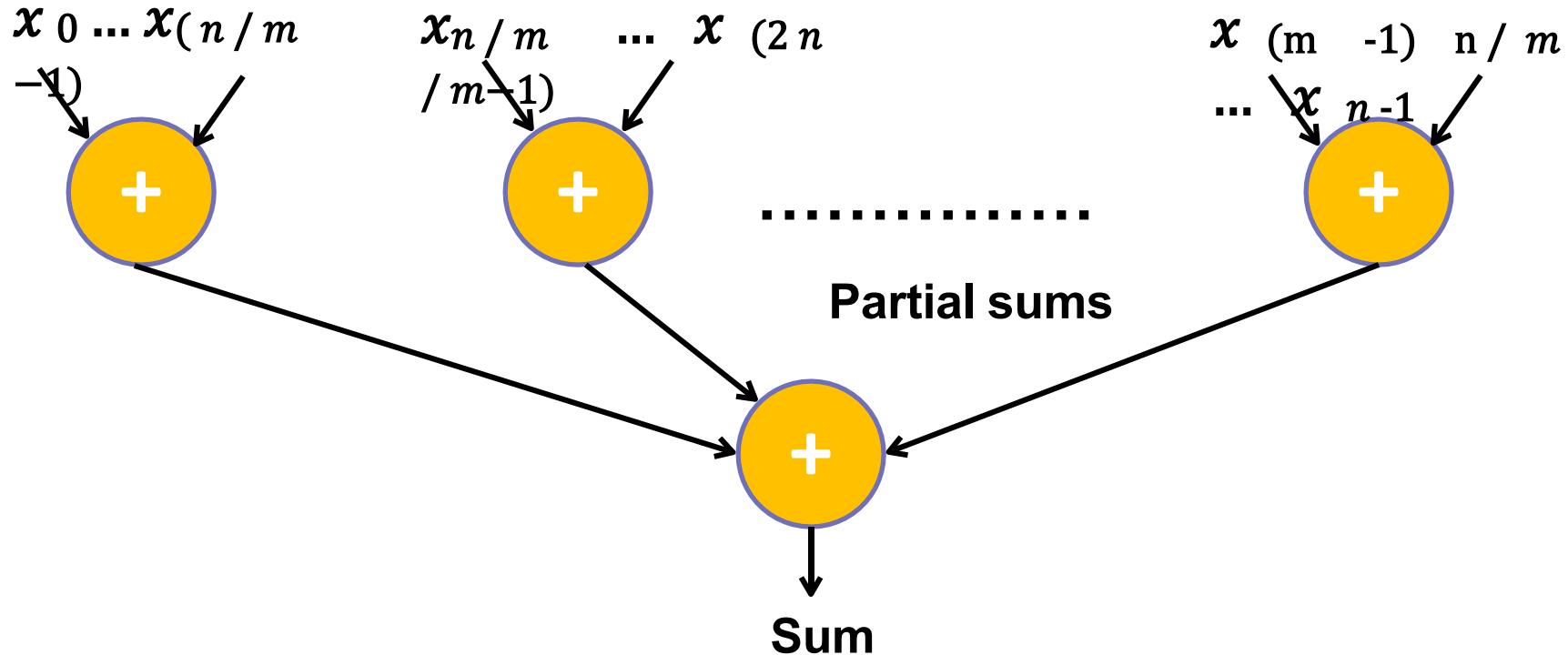
- Communication(for step 1 & 3)

$$\begin{aligned} T_{comm} &= (T_{startup} + n/2 \times T_{data}) + (T_{startup} + T_{data}) \\ &= 2T_{startup} + (n/2 + 1) \times T_{data} = O(n) \end{aligned}$$

- 算法整体复杂度:  $O(n)$

# 时间复杂度——示例2

- Adding  $n$  numbers using  $m$  processes
  - Evenly partition numbers to processes



# 时间复杂度——示例2

## ■ Adding $n$ numbers using $m$ processes

- Sequential:  $O(n)$
- Parallel:

- Phase1: Send numbers to slaves

$$t_{comm1} = m(t_{startup} + (n/m)t_{data})$$

- Phase2: Compute partial sum

$$t_{comp1} = n/m - 1$$

- Phase3: Send results to master

$$t_{comm2} = m(t_{startup} + t_{data})$$

- Phase4: Compute final accumulation

$$t_{comp2} = m - 1$$

- Overall:  $t_p = 2mt_{startup} + (n + m)t_{data} + m + \frac{n}{m} - 2$

并行计算

Tradeoff  
between  
**computation & communication**

# 目录

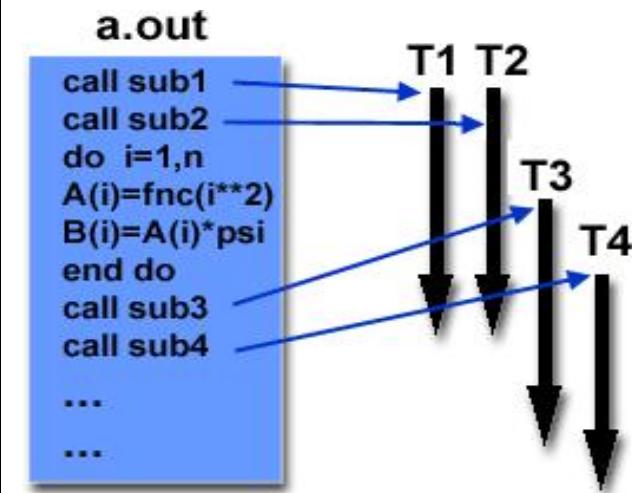
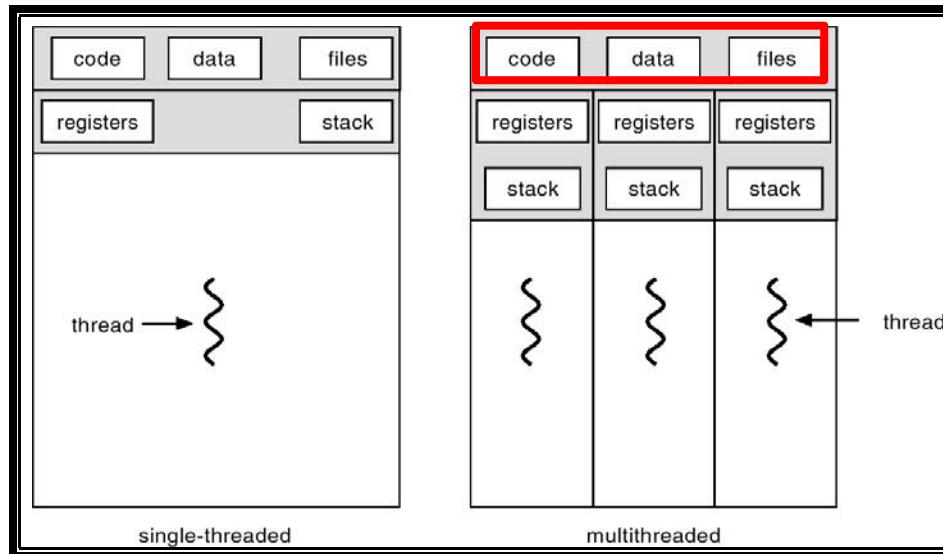
- PCAM设计原理
- 并行程序性能评价
- 并行程序编程模型

# 并行编程模型

- 硬件和内存架构之上的一种抽象
- 一般来说，编程模型的设计与计算机体系结构相匹配
  - 共享内存编程模型 → 共享内存系统
  - 消息传递编程模型 → 分布式内存系统
- 编程模型不受机器或内存体系结构的限制
  - 共享内存系统上可以支持消息传递编程模型：例如，单个服务器上的MPI
  - 分布式内存系统上支持共享内存编程模型：例如，Partitioned Global Address Space (PGAS)

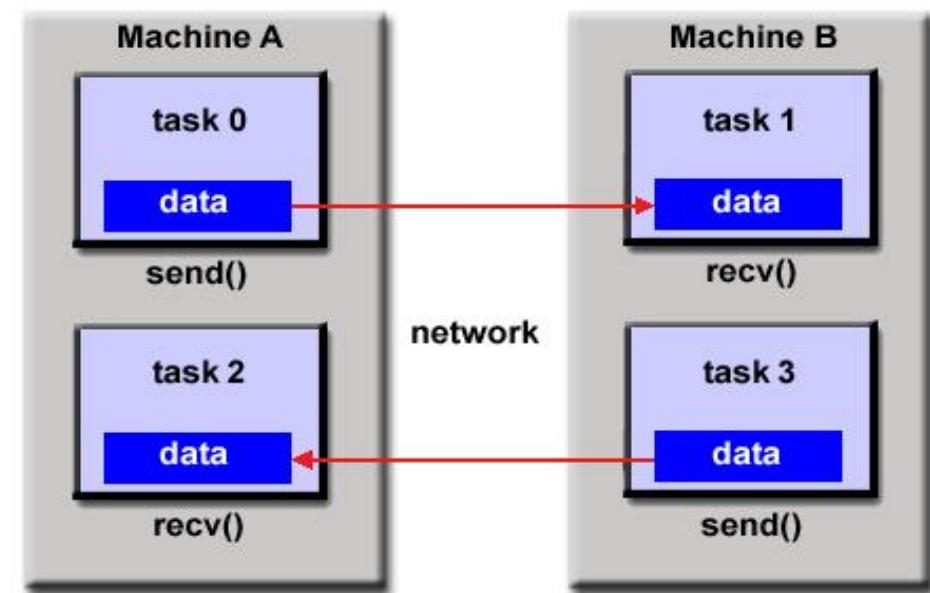
# 共享内存编程模型

- 单个进程可以有多个并发执行路径
- 线程有本地数据，但也共享资源
- 线程通过全局内存（Global Memory）相互通信
- 线程可以产生和消失，但主进程始终存在
  - 提供必要的共享资源，直到应用程序完成



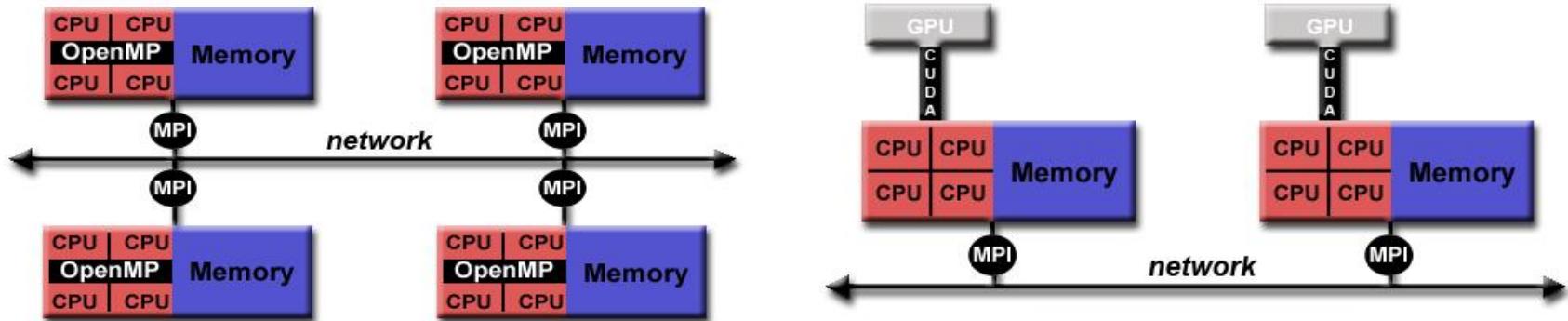
# 消息传递编程模型

- 在计算过程中使用自己的本地内存的一组任务
  - 多个任务可以驻留在同一台物理机器上和/或跨任意数量的机器
- 任务通过发送和接收消息（内存副本）的通信过程来交换数据
- 典型代表：MPI
  - Send, Recv, Broadcast
  - Gather, Scatter等



# 混合编程模型

- 混合模型结合了前面描述的多个编程模型
  - 消息传递模型（MPI）与共享内存模型（OpenMP）的组合
  - 消息传递模型（MPI）与CPU-GPU混合编程
- 混合模型适用于目前流行的多核/众核集群环境

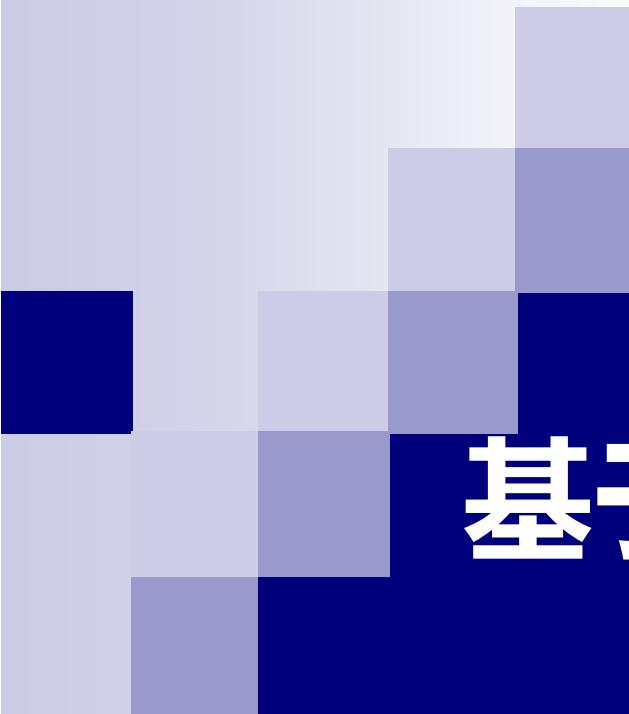


# 并行程序编程模型小结

- 编程模型和并行系统的设计和流行是相互影响的
- OpenMP、MPI、Pthreads、CUDA等只是一些供用户进行并行编程的并行语言
- 理解什么是并行计算甚至比如何进行并行编程更重要，因为理解并行计算，可以
  - 快速学习新的并行编程方式
  - 了解程序的性能和瓶颈
  - 优化程序的性能

# Reference

- 并行算法设计的基本原理， 并行计算课程， 清华大学  
<https://pop0726.github.io/bxjs/text/catalog/content1.htm>
- 并行计算课程， 陈国良， 中国科学技术大学
- **Introduction to Parallel Computing, Plamen Krastev, San Diego State University,**  
[http://sci.sdsu.edu/johnson/phys580/Parallel\\_computing1.pdf](http://sci.sdsu.edu/johnson/phys580/Parallel_computing1.pdf)
- **Parallel Programming course slides from Prof. Jerry Chou, National Tsing Hua University.**
- **Blaise Barney, Lawrence Livermore National Laboratory, Introduction to Parallel Computing,**  
[https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)



# 第四章

# 基于共享内存的并行计算

哈尔滨工业大学

孙新越

2025, Fall Semester

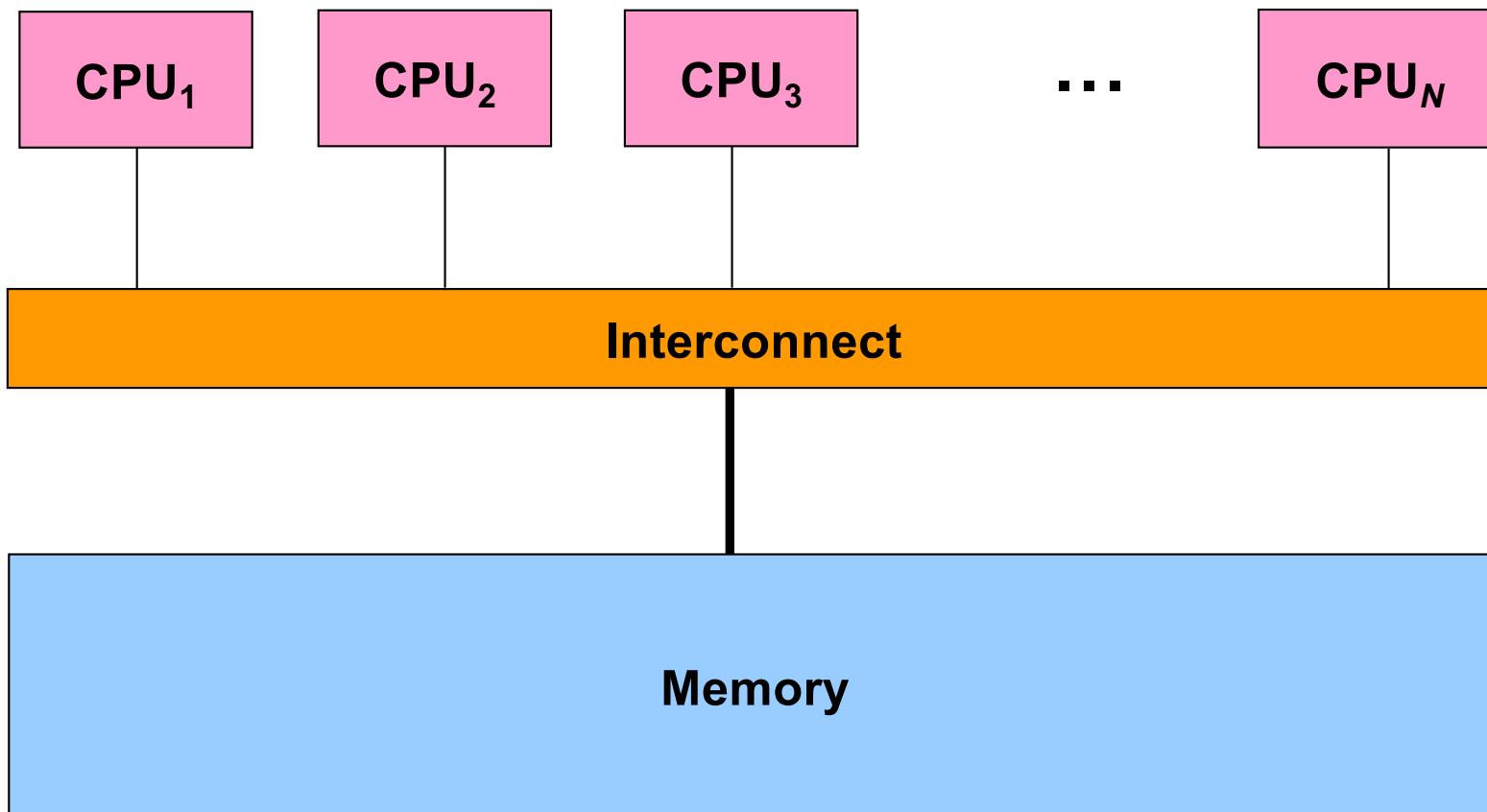
# 目录

- 进程与线程
- 基于Pthread的多线程并行
- 基于OpenMP的多线程并行

# 目录

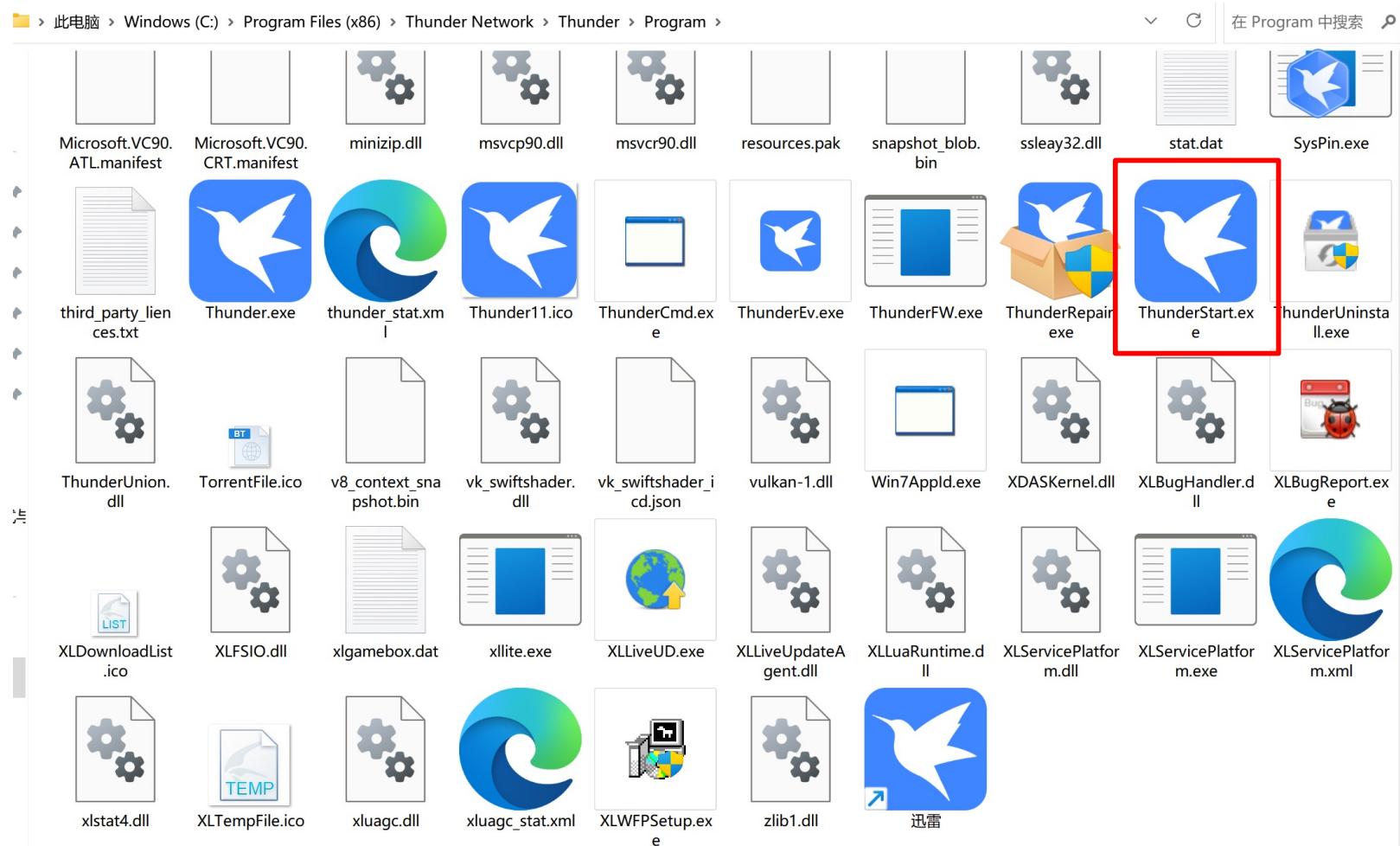
- **进程与线程**
- 基于Pthread的多线程并行
- 基于OpenMP的多线程并行

# 共享内存系统



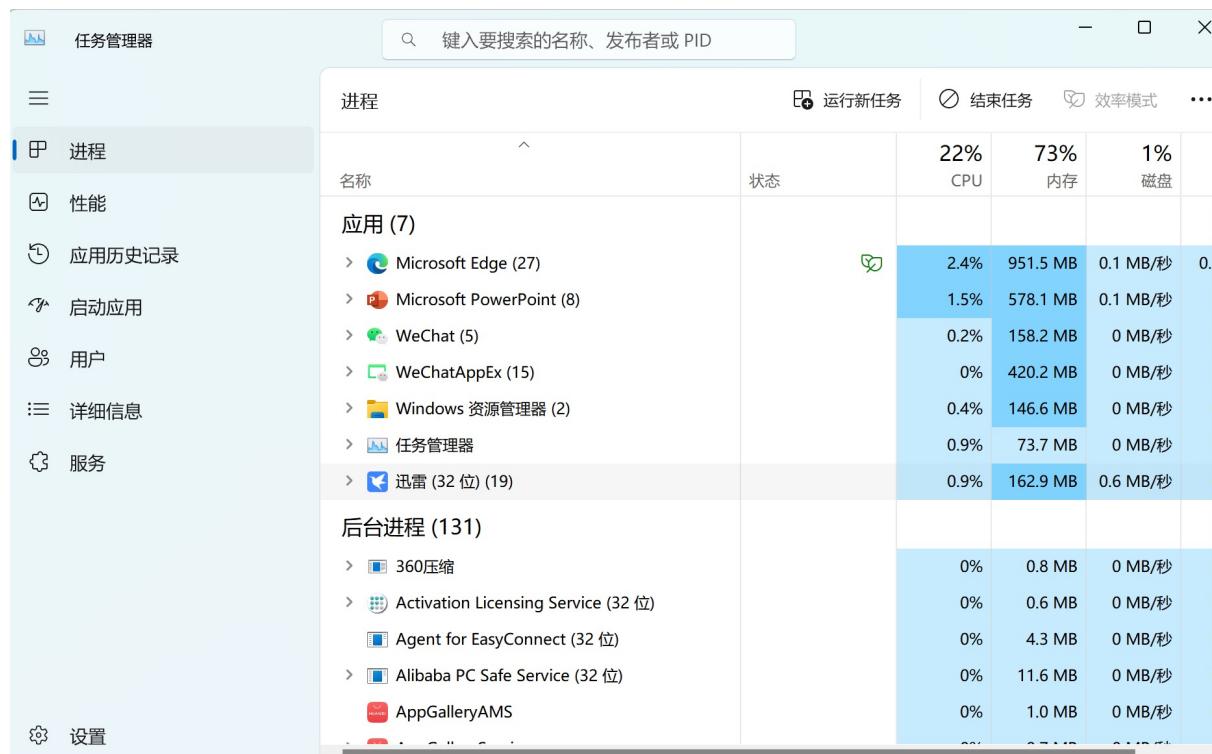
# 程序

## ■ 程序是静态文件



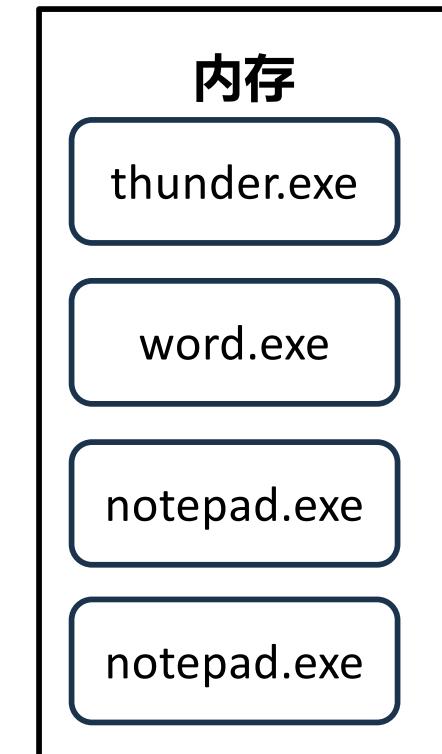
# 进程

- 进程是程序运行的实例，当一个进程执行进入内存运行时，即变成一个进程
- 进程的资源是彼此隔离的，其他进程不允许访问

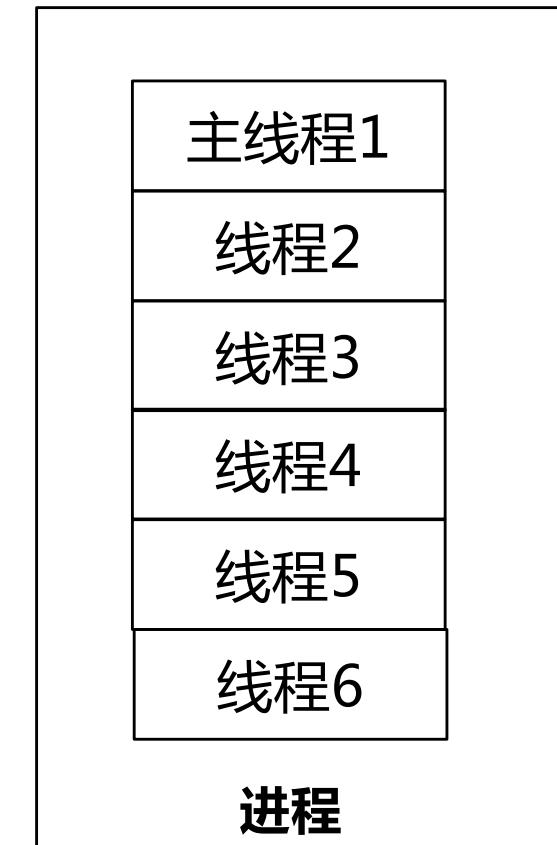
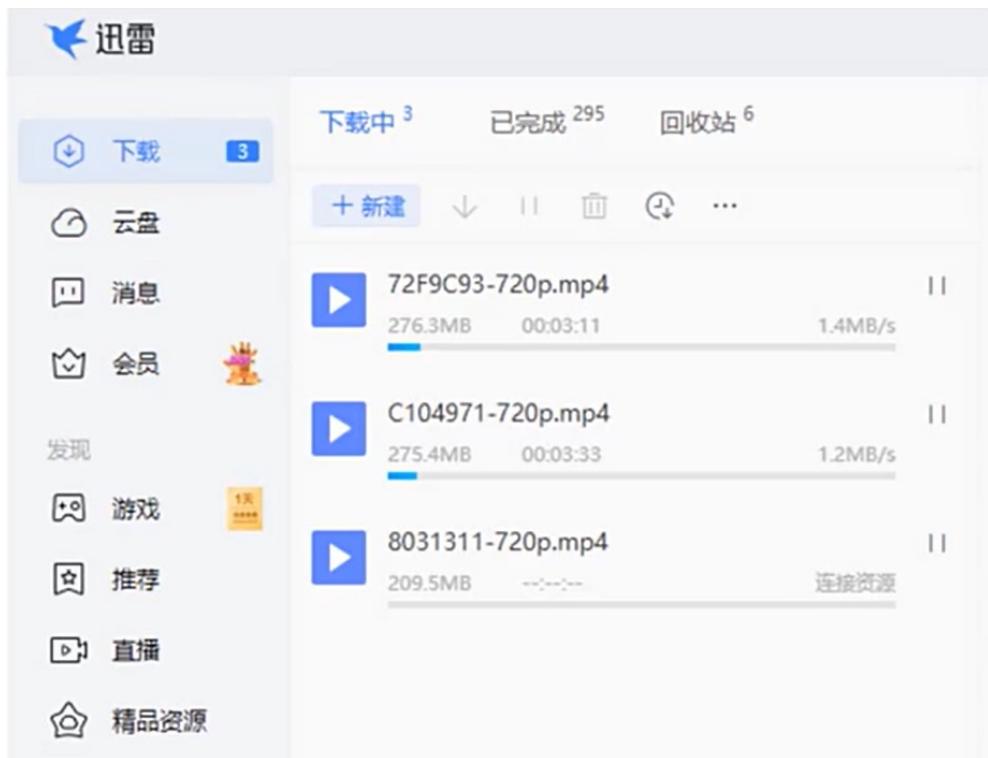


任务管理器

并行计算



# 线程

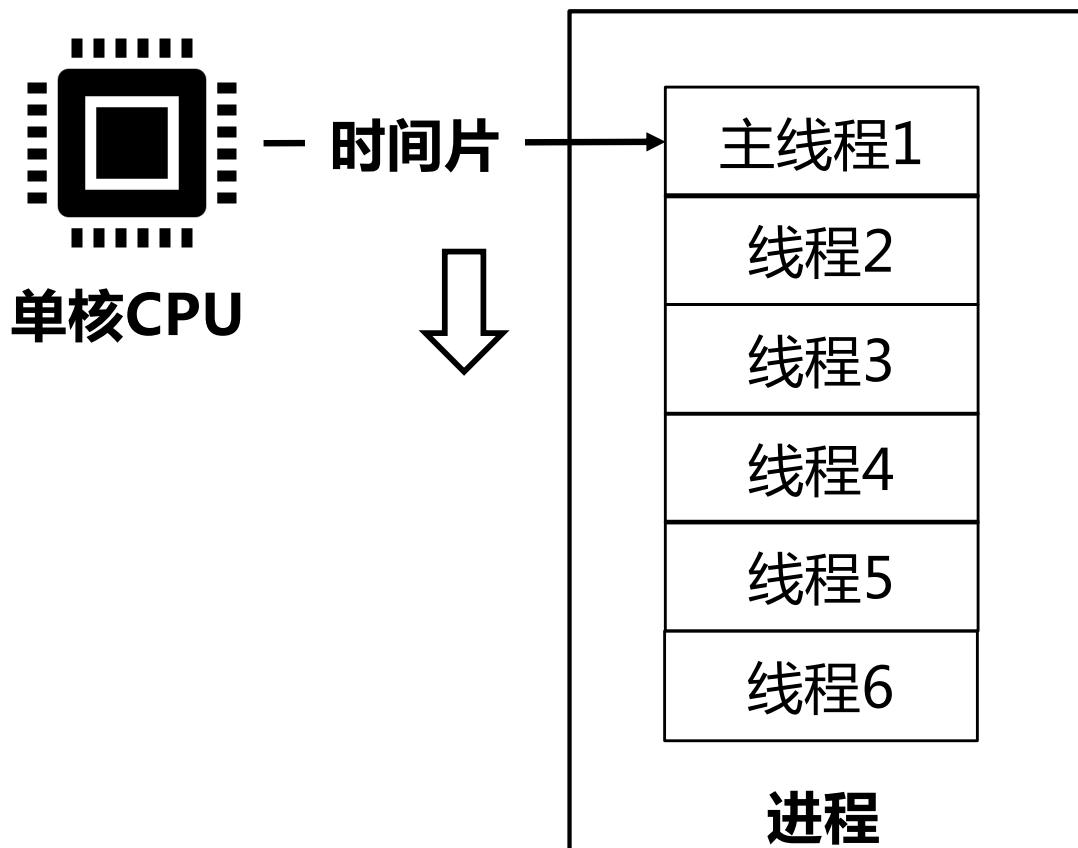


# 线程

- 线程是进程执行的“基本任务”，每个线程都有自己的功能，是CPU分配与调度的基本单位
- 一个进程中可以包含多个线程，反之，一个线程只能隶属于某一个进程
- 进程内至少拥有一个“线程”，这个线程叫“主线程”，主线程消亡则进程结束
- 轻量级“进程”

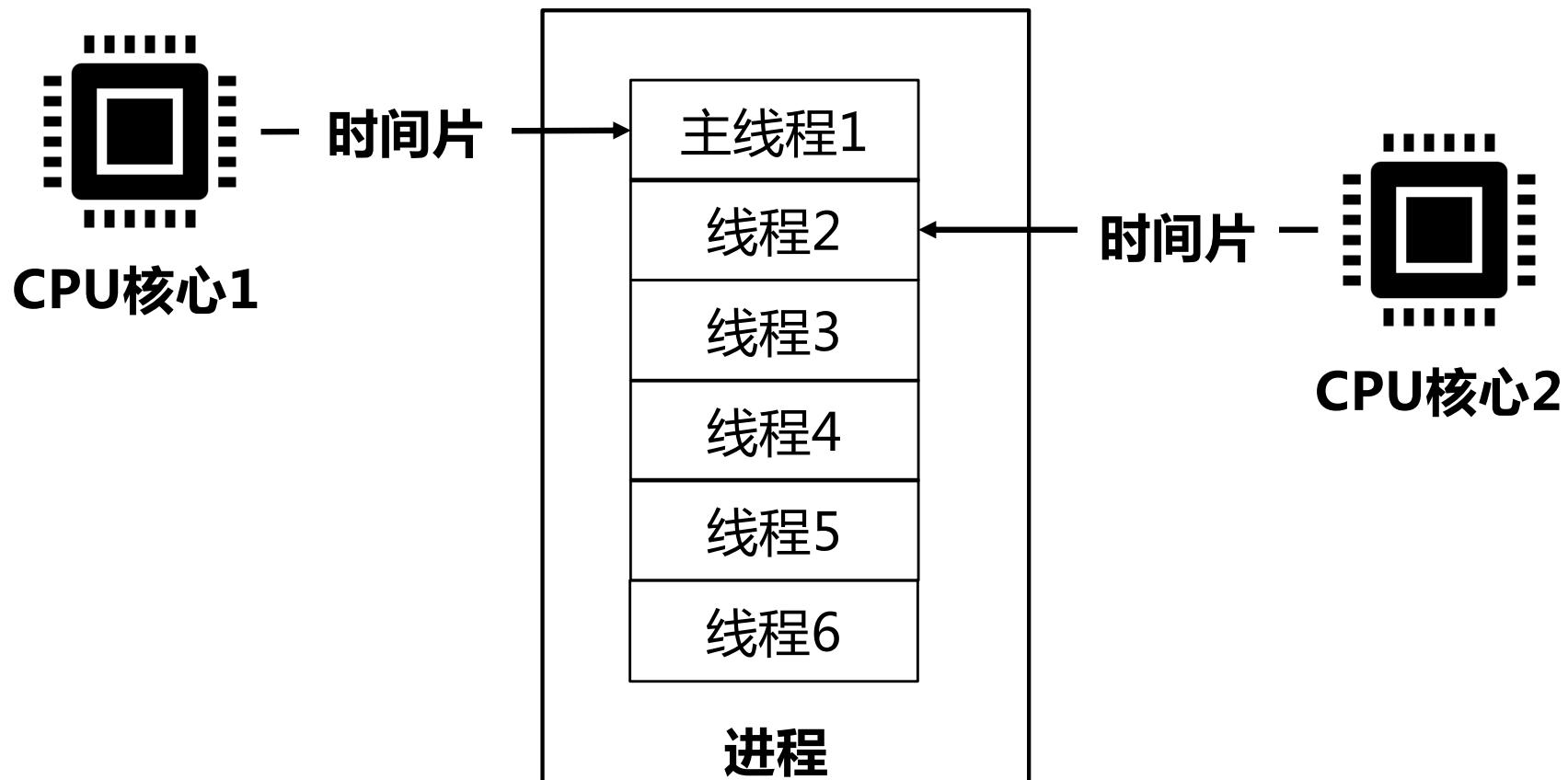


# CPU、进程、线程的关系



并发执行

# CPU、进程、线程的关系



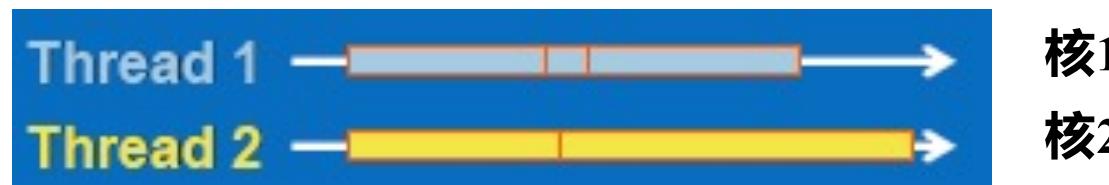
并行执行

# CPU、进程、线程的关系

- 单核平台，线程并发执行



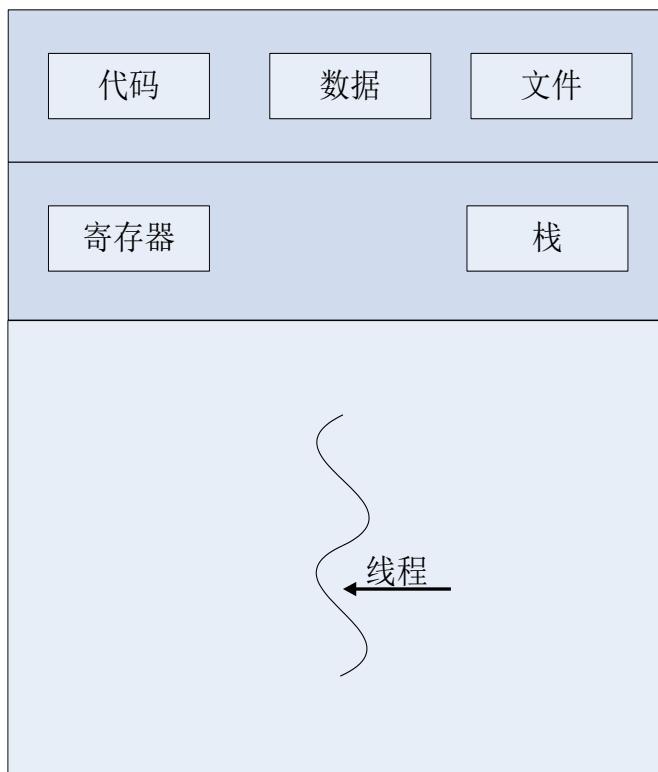
- 多核平台，各个核上可以实现线程并行



# 多线程

- 在单个程序中同时运行多个线程完成不同的工作，  
称为多线程

进程

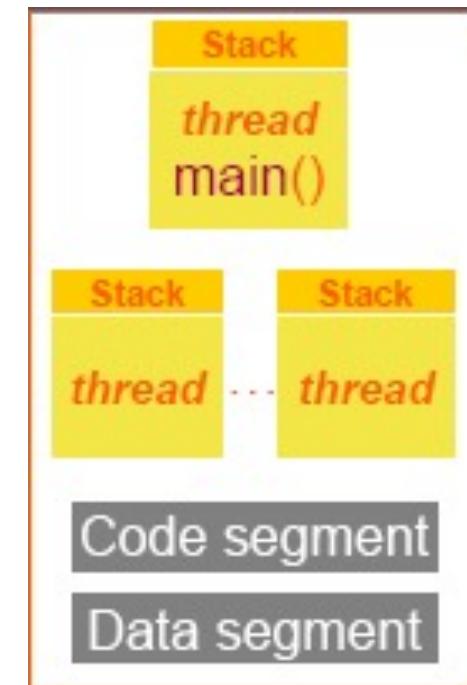


进程



# 多线程

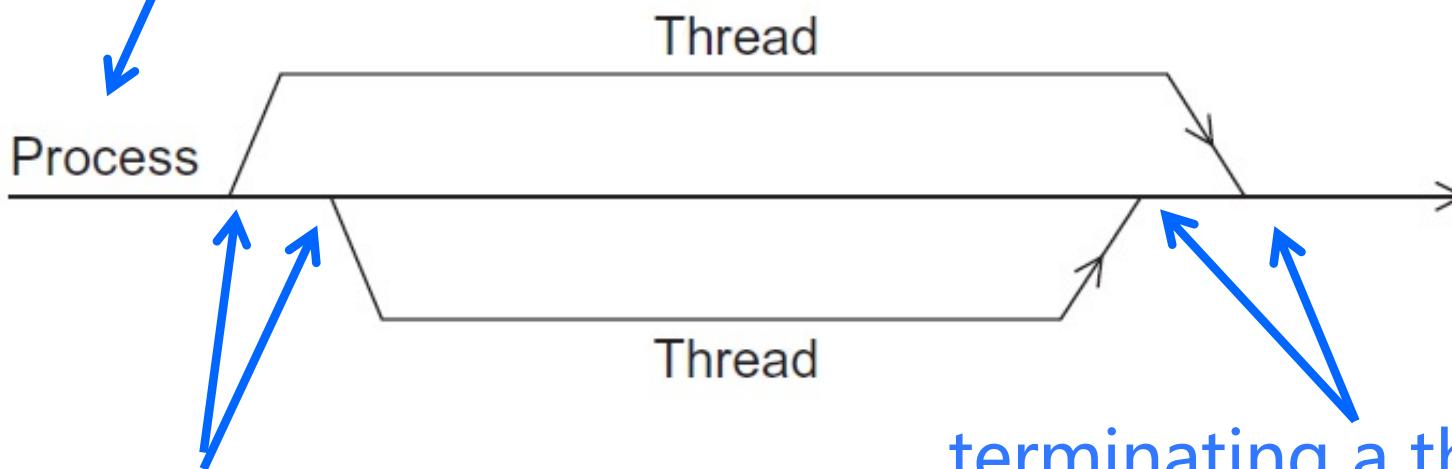
- 创建一个线程比创建一个进程的代价要小
- 线程的切换比进程间的切换代价小
- 充分利用多处理器
- 数据共享，使线程间的通信比进程间通信更高效



进程

# 多线程

the “master” thread (主线程)



starting a thread  
Is called forking (派生)

terminating a thread  
Is called joining (合并)

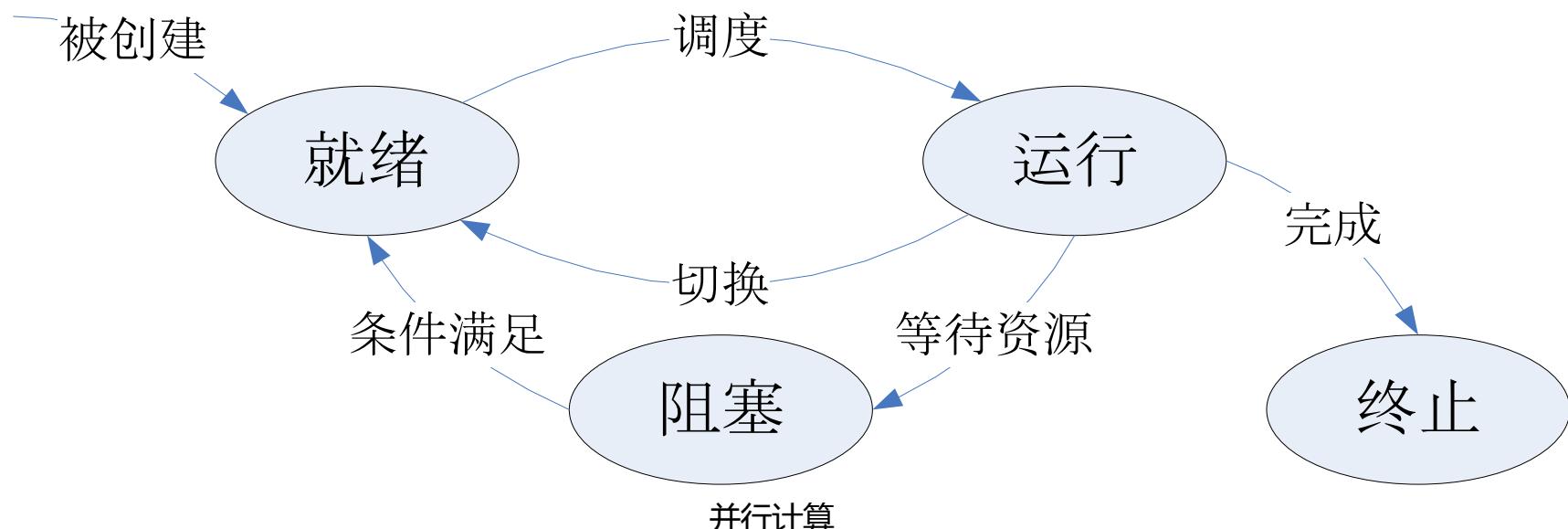
# 多线程

- 线程的生命周期
- 线程的标识：通常用一个整数来标识一个线程
- 线程的创建：
  - 自动创建，从main函数开始的主线程
  - 调用函数库接口创建新线程（`pthread_create`）
- 线程的终止
  - 执行完毕，或者调用了`pthread_exit`
  - 主线程退出导致整个进程终止

# 多线程

## ■ 线程的状态

- 就绪 ( ready ) : 线程等待可用的处理器。
- 运行 ( running ) : 线程正在被执行。
- 阻塞 ( blocked ) : 线程正在等待某个事件的发生 ( 比如 I/O 的完成 , 试图加锁一个被上锁的互斥量 )。
- 终止 ( terminated ) : 线程从起始函数中返回或者调用 pthread\_exit 。



# 目录

- 进程与线程
- 基于Pthreads的多线程并行
- 基于OpenMP的多线程并行

# Pthreads库简介

- **POSIX为支持多线程应用而提供的一组程序设计标准接口（API）**
- **一个类Unix操作系统（如Linux、Mac OS等）上的标准库**
- **Pthread API提供了编写并行程序必须的函数，主要分为如下几类：**
  - **线程管理（Thread management）**
  - **互斥量（Mutexes）**
  - **条件变量（Condition variables）**
  - **同步（Synchronization）**

# “Hello world” (1)

- Pthreads程序示例：主函数启动了多个线程，每个线程打印一条消息，然后退出

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

/* Global variable: accessible to all threads */
int thread_count;

void * Hello(void* rank); /* Thread function */

int main(int argc, char* argv[]){
    long thread; /* Use long in case of a 64-bit system */
    pthread_t* thread_handles;

/* Get number of threads from command line */
    thread_count = strtol(argv[1], NULL, 10);
    thread_handles = malloc(thread_count * sizeof(pthread_t));
```

# “Hello world” (1)

- Pthreads程序示例：主函数启动了多个线程，每个线程打印一条消息，然后退出

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

**pthread\_t**数据结构用来存储线程的专有信息，由**pthread.h**声明。**pthread\_t**对象是一个不透明对象。对象中存储的数据都是系统绑定的，用户级代码无法直接访问到里面的数据

**pthread.h**，这是Pthreads线程库的头文件，用来声明Pthreads的函数、常量和类型等

*ble to all threads \*/*

全局变量，线程共享

*/\* Thread handles \*/*

从命令行读取需要生成的线程数量。**strtol**函数的功能是将字符串转化为**long int**（长整型），在**stdlib.h**中声明

```
int main(int argc, char* argv[]){
    long thread; /* Use long in case of overflow */
    pthread_t* thread_handles;
```

为每个线程的**pthread\_t**对象分配内存

*/\* Get number of threads from command line \*/*

```
thread_count = strtol(argv[1], NULL, 10);
thread_handles = malloc(thread_count * sizeof(pthread_t));
```

# “Hello world” (2)

- Pthreads程序示例：主函数启动了多个线程，每个线程打印一条消息，然后退出

```
for (thread = 0; thread < thread_count ; thread++)
    pthread_create(&thread_handles[thread], NULL,
                   Hello, (void*) thread);

printf("Hello from the main thread\n");

for (thread = 0; thread < thread_count; thread++)
    pthread_join(thread_handles[thread], NULL);

free(thread_handles);

return 0;
} /*main*/
```

# “Hello world” (2)

- Pthreads程序示例：主函数启动了多个线程，每个线程打印一条消息，然后退出

```
for (thread = 0; thread < thread_count ; thread++)
    pthread_create(&thread_handles[thread], NULL,
                   Hello, (void*) thread);
printf("Hello from the ma
```

```
int pthread_join(pthread_t *thread,
                 void ** ret_val_p);
```

第二个参数可以接收任意由pthread\_t对象所关联的那个线程产生的返回值

每个线程调用一次pthread\_join函数。  
调用一次pthread\_join将等待pthread\_t对象所关联的那个线程结束

```
pthread_create(pthread_t *thread,
               pthread_attr_t *attr,
               void *(*start_fun)(void*),
               void * arg);
```

thread 操作系统返回的线程ID  
attr 程序指定的线程属性 (或NULL)  
start\_fun 线程的开始函数  
arg 线程开始函数的参数

# “Hello world” (3)

- Pthreads程序示例：主函数启动了多个线程，每个线程打印一条消息，然后退出

```
void* Hello(void* rank){  
    /* Use long in case of 64-bit system */  
    long my_rank = (long) rank;  
  
    printf("Hello from thread %ld of %d\n", my_rank,  
          thread_count);  
  
    return NULL;  
} /* Hello Function */
```

# “Hello world” (3)

- Pthreads程序示例：主函数启动了多个线程，每个线程打印一条消息，然后退出

```
void* Hello(void* rank){  
    /* Use long in case of 64-bit integers */  
    long my_rank = (long)  
        *( (long*)rank );  
  
    printf("Hello from thread %ld\n", my_rank );  
  
    return NULL;  
} /* Hello Function */
```

由pthread\_create生成并运行的函数

void\* thread\_function ( void\* args\_p )

void\* 可以转换成C语言中任意指针类型  
args\_p 可以指向一个列表，该列表包含一个或者多个该函数需要的数值  
thread\_function返回的值 可以是一个包含一个或者多个值的列表

# “Hello world” (4)

## ■ 编译和运行Pthreads程序

**编译:** `gcc pth_hello.c -o pth_hello -lpthread`

**运行:** `./pth_hello <number of threads>`

```
./pth_hello 1 ↵
```

```
Hello from the main thread
```

```
Hello from thread 0 of 1
```

```
./pth_hello 4 ↵
```

```
Hello from the main thread
```

```
Hello from thread 0 of 4
```

```
Hello from thread 1 of 4
```

```
Hello from thread 2 of 4
```

```
Hello from thread 3 of 4
```

# 线程创建

向调用者传递子线程的线程号

```
int pthread_create (  
    pthread_t *thread_handle,  
    pthread_attr_t *attribute,  
    void * (*thread_function)(void *),  
    void *arg );
```

typedef struct  
{

**控制线程的各种属性**

    int  
    int  
    struct sched\_param  
    int  
    int  
    size\_t  
    int  
    void\*  
    size\_t  
} pthread\_attr\_t;

**指定子线程需要执行的函数**

    detachstate; //线程的分离状态  
    schedpolicy; //线程调度策略  
        schedparam; //线程的调度参数  
    inheritsched; //线程的继承性  
    scope; //线程的作用域  
    guardsize; //线程栈末尾的警戒缓冲区大小  
    stackaddr\_set; //线程堆栈的地址集  
    stackaddr; //线程栈的位置  
    stacksize; //线程栈的大小

# 线程终止

- 阻塞并等待**thread**参数指定的线程的终止。通常由“**master thread**”调用

```
int pthread_join ( pthread_t thread, void **ptr );
```

- 使线程正常终止，**value\_ptr**是数据指针，可以储存线程的返回值，可以被**pthread\_join()**函数的第二个参数接收，通常由“**worker thread**”调用

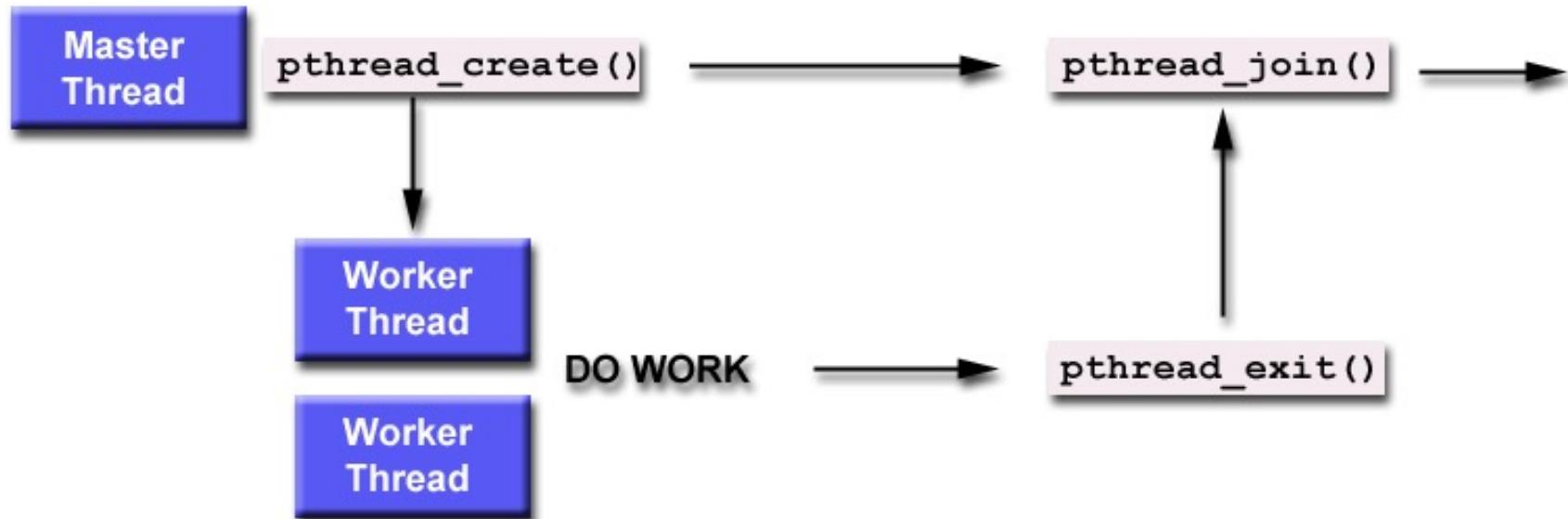
```
void pthread_exit ( void *value_ptr );
```

- 被同进程中的其它线程（包括主线程）强制终止

```
int pthread_cancel ( pthread_t thread );
```

# 线程终止

- `pthread_join`实现线程间同步的一种方式，子线程合入主线程，主线程阻塞等待子线程结束，然后回收子线程资源



- 主线程与子线程分离，子线程结束资源自动回收

```
int pthread_detach( pthread_t thread );
```

# 矩阵-向量乘法

- 如果  $A = (a_{ij})$  是一个  $m \times n$  的矩阵， $x$  是一个  $n$  维列向量，矩阵-向量的乘积  $Ax = y$  是一个  $m$  维的列向量

$$\begin{array}{|c|c|c|c|}\hline a_{00} & a_{01} & \cdots & a_{0,n-1} \\ \hline a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\ \hline\end{array} \quad \begin{array}{c} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{array} = \begin{array}{|c|c|c|c|}\hline y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1} \\ \hline \vdots \\ \hline y_{m-1} \\ \hline\end{array}$$

# 串行伪代码

```
/*For each row of A*/
for(i = 0; i < m; i++) {
    y[i] = 0.0;
    /*For each element of the row and each element of x*/
    for(j = 0; j < n; j++)
        y[i] += A[i][j] * x[j];
}
```

$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j$$

# 并行化代码

- 通过把工作分配给各个线程将程序并行化
- 一种分配方法是将线程外层的循环分块，每个线程计算y的一部分
- 例如，假设 $m=n=6$ ，线程数thread\_count（或t）为3，则计算可以按下列情况分配

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

# 并行化代码

- 为了计算 $y[0]$ ，线程0将执行代码

```
y[0] = 0.0;  
for(j = 0; j < n; j++) {  
    y[0] += A[0][j] * x[j];  
}
```

- 线程0需要访问矩阵A的第0行以及向量x中的每一个元素
- 一般地，被分配给 $y[i]$ 的线程将执行代码

```
y[i] = 0.0;  
for(j = 0; j < n; j++) {  
    y[i] += A[i][j] * x[j];  
}
```

# 使用3个线程

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

每个线程除了访问各自分配到的矩阵A的第i行以及y分量外，还要访问X中的每个元素。这意味着最低限度下，要共享向量x

如果矩阵A和y是全局变量，主函数就可以简单地通过读取标准输入stdin来初始化矩阵A，乘积向量y也可以很容易被主线程打印输出

thread 0

```
y[0] = 0.0;  
for(j = 0; j < n; j++) {  
    y[0] += A[0][j] * x[j];  
}
```

general case

```
y[i] = 0.0;  
for(j = 0; j < n; j++) {  
    y[i] += A[i][j] * x[j];  
}
```

# 并行化代码

- 编写每一个线程的代码，确定每个线程计算哪一部分的y
- 假设，m与n都能被t整除，在例子中， $m=6$ ， $t=3$ ，每个线程能分配到 $m/t=3$ 行的运算数据，而且，线程0处理第一部分的 $m/t$ 行，线程1处理第二部分的 $m/t$ 行...
- 所以线程q处理的矩阵行是

$$\text{第一行 : } q \times \frac{m}{t}$$

$$\text{最后一行 : } (q+1) \times \frac{m}{t} - 1$$

$a_{00}$	$a_{01}$	$\cdots$	$a_{0,n-1}$
$a_{10}$	$a_{11}$	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{i0}$	$a_{i1}$	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$

$x_0$	=	$y_0$
$x_1$		$y_1$
$\vdots$		$\vdots$
		$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$
		$\vdots$
		$y_{m-1}$

# Pthreads多线程实现

```
void* Path_mat_vec(void* rank) {
    y[i] = 0.0;
    long my_rank = (long) rank;
    int i, j;
    int local_m = m/thread_count;
    int my_first_row = my_rank * local_m;
    int my_last_row = (my_rank + 1) * local_m - 1;

    for(i = my_first_row; j <= my_last_row; i++) {
        y[i] = 0.0;
        for(j = 0; j < n; j++)
            y[i] += A[i][j] * x[j];
    }
    return NULL;
}/* Pth_mat_vect function*/
```

# 练习

## ■ 编写完整的Pthreads矩阵-向量程序

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

/*Global variables*/
int      thread_count;
Int      m, n;
double* A;
double* x;
double* y;

/*Serial functions*/
void Usage(char* prog_name);
void Read_matrix(char* prompt, double A[], int m, int n);
void Read_vector(char* prompt, double x[], int n);
void Print_matrix(char* title, double A[], int m, int n);
void Print_vector(char* title, double y[], double m);

/*Parallel function*/
void* Pth_mat_vect(void* rank);
```

提示

# 共享变量被改写？

- 共享内存区域是较理想的存储访问方式，所以矩阵-向量乘法的代码很容易编写
- 在主函数创建线程后，除了y以外，**没有任何共享变量被改写**
- 即使是y，也是每个线程各自改变属于自己运算的那一部分，没有两个或两个以上线程共同处理同一部分y的情况
- **如果多个线程需要更新同一内存单元的数据会怎样？**

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

# 估计 $\pi$ 值

$$\pi = 4 \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots \right)$$

```
double factor = 1.0;
double sum = 0.0;

for(i = 0; j < n; i++, factor = -factor) {
    sum += factor/(2 * I + 1);
}
pi = 4.0 * sum;
```

# 估计 $\pi$ 值的并行化

- 用并行化矩阵-向量乘法的方法并行化该程序：  
将for循环分块后交给各个线程处理，并将sum设为全局变量
- 为了简化计算，假设线程数`thread_count`，简称t能够整除项目总数n。一般地，对于线程q，循环变量的范围是

$$\bar{n} = n/t$$

$$q\bar{n}, q\bar{n} + 1, q\bar{n} + 2, \dots, (q + 1)\bar{n} - 1$$

# 使用线程函数来计算π

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n * my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0) /*my_first_i is even*/
        factor = 1.0;
    else                      /*my_first_i is odd*/
        factor = -1.0;

    for(i = my_first_i; j < my_last_i; i++, factor = -factor) {
        sum += factor/(2 * i + 1);
    }
    return NULL;
}/* Thread_sum function*/
```

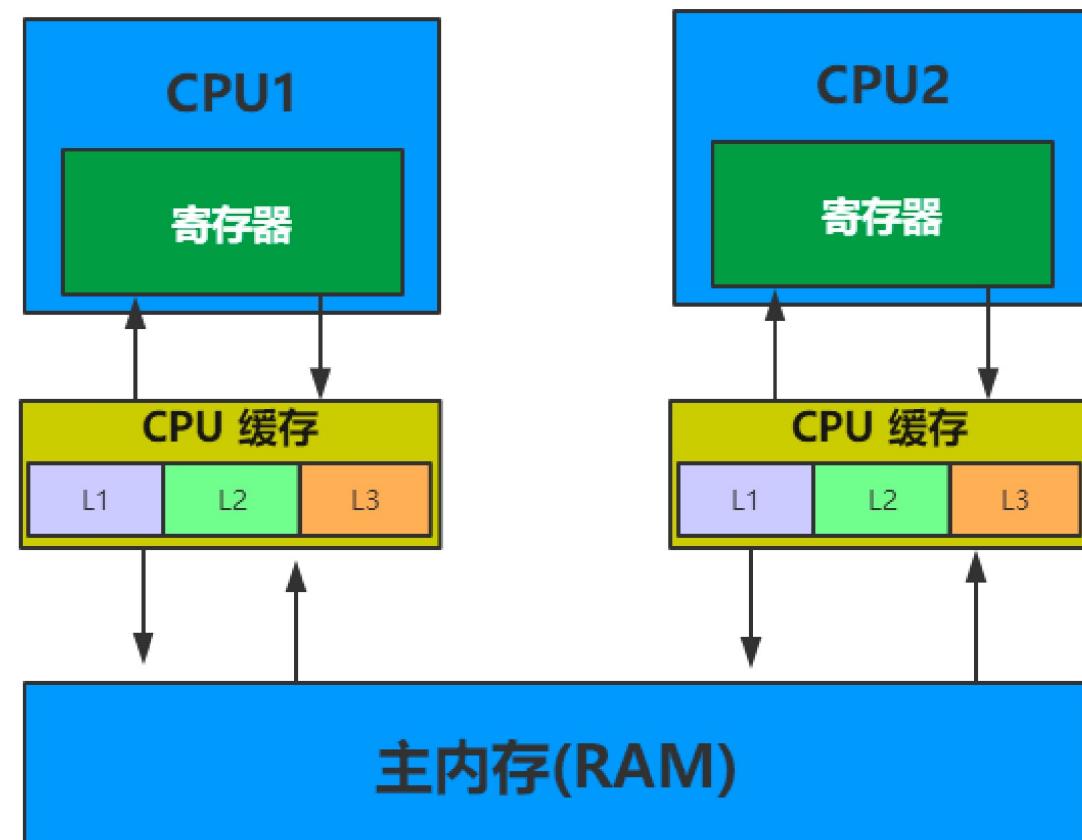
# 使用双线程测试

- 随着n的增加，单线程的估算结果也越来越准确，  
n每增大十倍就能获得更加精确的结果
- 双线程的计算结果反而变糟。事实上，多次运行  
双线程程序，尽管n未变，但每次的结果都不同
- 结论：当多个线程尝试更新同一个共享变量时，  
会出现问题

	n			
	$10^5$	$10^6$	$10^7$	$10^8$
$\pi$	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

# 可能的竞争条件

- 用一条C语言语句将存储单元y的内容加到存储单元x中去： $x=x+y$
- 机器执行：从主存中加载到CPU的寄存器，才能进行加法运算。当运算完成后，必须将结果再从寄存器重新存储到主存中



# 可能的竞争条件

- 假设有2个线程，每个线程对并存储在自己私有变量y中的值进行计算。还假设将这些私有变量加到共享变量x中，主线程将x的初始值置为0
- 每个线程执行以下代码

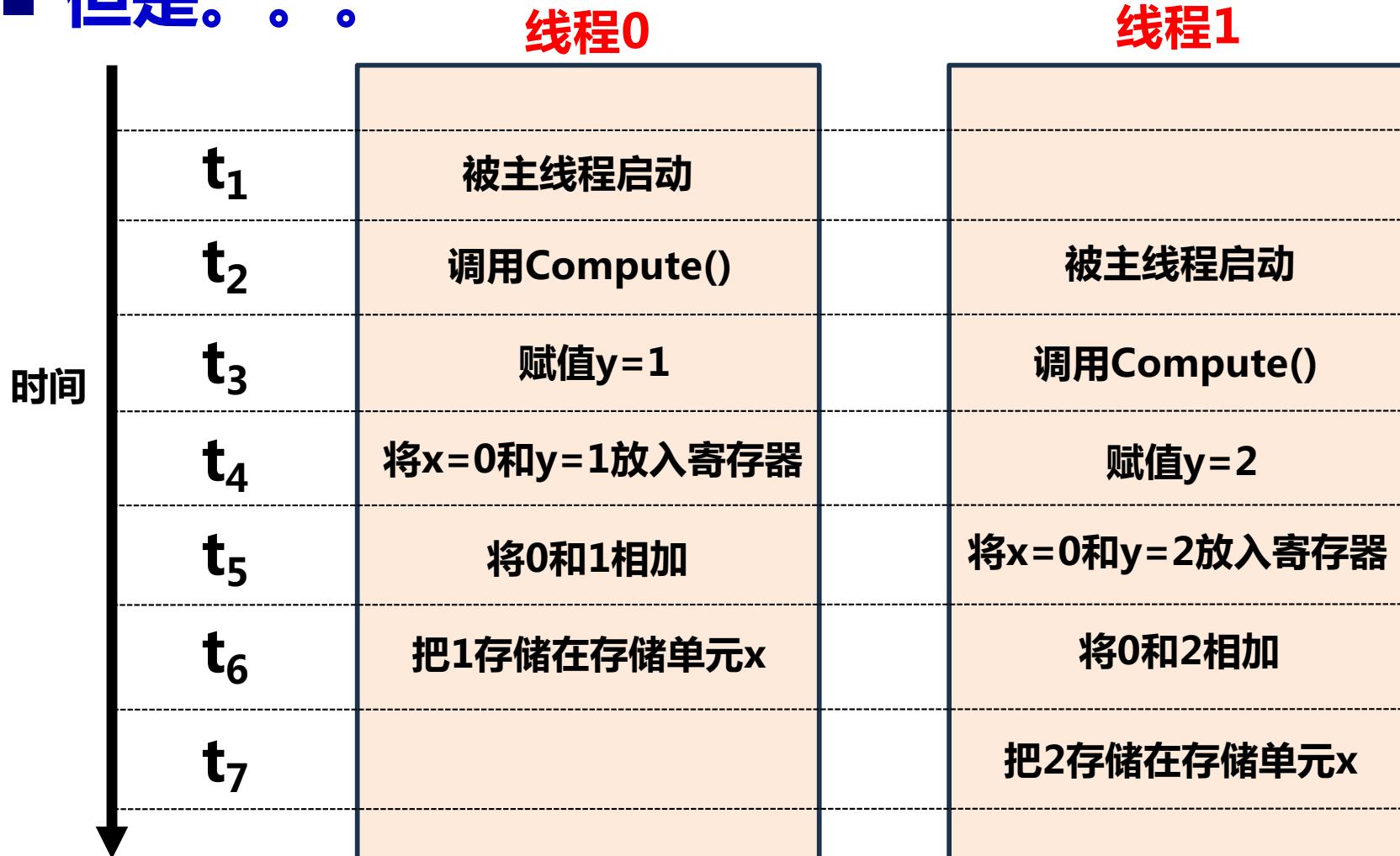
```
y= Compute(my_rank);  
x=x+y;
```

- 假设线程0计算出的结果为y=1，线程1计算出的结果为y=2，则“正确”结果就应该是x=3

# 可能的竞争条件

$y = \text{Compute}(\text{my\_rank});$   
 $x = x + y;$

- 但是。 . .



# 竞争条件

- 当多个线程尝试更新一个共享资源（共享变量）时，结果可能是无法预测的
- 当多个线程都要访问共享变量或共享文件这样的共享资源时，如果至少其中一个访问是更新操作，那么这些访问就可能会导致某种错误，我们称之为竞争条件( race condition)

# 临界区

- **临界区**：一个更新共享资源的代码段，一次只允许一个线程执行该代码段
- **目的**：当多个线程访问共享数据时，为保证数据的完整性，将共享数据保护起来
- **访问临界区的原则**：
  - 一次最多只能一个线程停留在临界区内
  - 不能让一个线程无限地停留在临界区内



并行计算

# 临界区：忙等待

```
y = Compute(my_rank);  
x = x + y;
```

临界区

```
flag = 0; // 主线程初始化
```

.....

```
y = Compute(my_rank);  
while (flag != my_rank);  
x = x + y;  
flag++;
```

这里的while循环只有判断语句，没有执行语句  
执行临界区代码的顺序：线程0，线程1，线程2...

# 使用忙等待求全局和（计算 $\pi$ ）

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n * my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0) /*my_first_i is even*/
        factor = 1.0;
    else                      /*my_first_i is odd*/
        factor = -1.0;

    for(i = my_first_i; j < my_last_i; i++, factor = -factor) {
        while (flag != my_rank);
        sum += factor/(2 * i + 1);
        flag = (flag+1) % thread_count;
    }
    return NULL;
}/*Thread_sum function*/
```

问题？

# 使用忙等待求全局和（计算 $\pi$ ）

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n * my_rank;
    long long my_last_i = my_first_i + my_n;
    double my_sum = 0.0;

    if (my_first_i % 2 == 0) /*my_first_i is even*/
        factor = 1.0;
    else                      /*my_first_i is odd*/
        factor = -1.0;

    for(i = my_first_i; j < my_last_i; i++, factor = -factor) {
        my_sum += factor/(2 * i + 1);
    }
    while (flag != my_rank);
    sum += my_sum;
    flag = (flag+1) % thread_count;
    return NULL;
}/* Thread_sum function*/
```

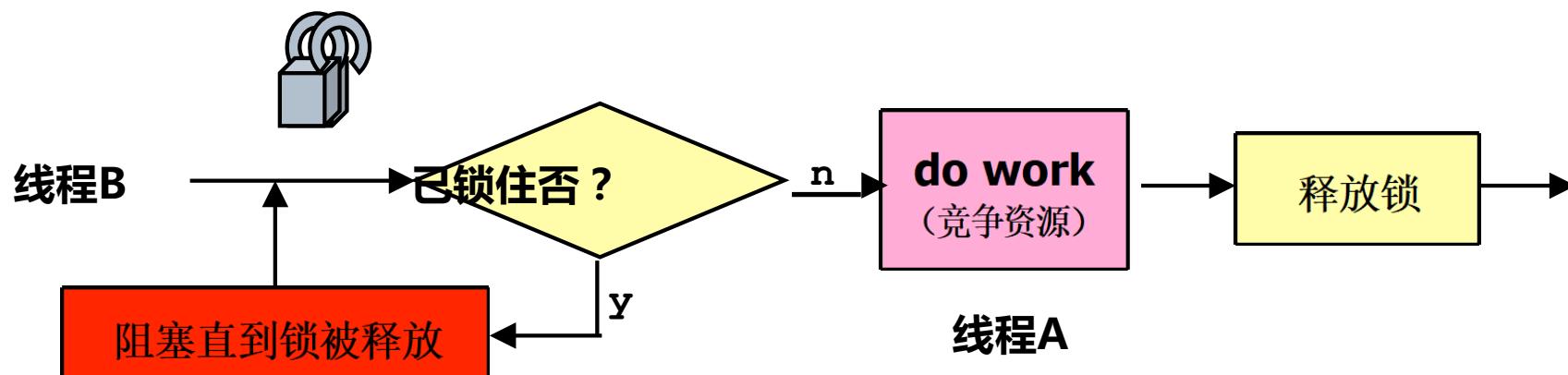
循环后用忙等  
待求全局和

# 临界区：互斥量

- 互斥量是互斥锁的简称，特殊类型的变量，通过某些特殊类型的函数，互斥量可以用来限制每次只有一个线程能进入临界区
- 互斥量可以保证了一个线程独享临界区，其他线程在有线程以及进入该临界区的情况下，不能同时进入

# 临界区：互斥量

- 互斥量有两种状态：
  - **lock** 互斥量一旦被某个线程锁住，其它申请该锁的线程将被阻塞，直至锁被解开
  - **unlock** 未锁住的互斥锁允许线程通过，然后自动变为**lock**状态。已锁住的互斥锁必须由得到此锁的线程释放



# 临界区：互斥量

- Pthread标准为互斥量提供了一个特殊类型 **pthread\_mutex\_t**
- 创建一个互斥量，其状态为**unlock**

```
pthread_mutex_t mtx;  
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;  
或  
mtx = pthread_mutex_init(&mtx, NULL);
```

# 临界区：互斥量

## ■ 加锁（阻塞和非阻塞）

如果状态为unlock，则得到此锁并改变状态为lock，然后返回；否则阻塞线程

```
pthread_mutex_lock(pthread_mutex_t *mtx);
```

```
pthread_mutex_trylock(pthread_mutex_t *mtx);
```

## ■ 解锁

改变锁状态为unlock，然后返回

```
pthread_mutex_unlock(pthread_mutex_t *mtx);
```

## ■ 销毁互斥量

程序使用完互斥量后，进行销毁

```
pthread_mutex_destroy(pthread_mutex_t *mtx);
```

# 使用互斥量求全局和（计算 $\pi$ ）

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n * my_rank;
    long long my_last_i = my_first_i + my_n;
    double my_sum = 0.0;

    if (my_first_i % 2 == 0) /*my_first_i is even*/
        factor = 1.0;
    else                      /*my_first_i is odd*/
        factor = -1.0;

    for(i = my_first_i; j < my_last_i; i++, factor = -factor) {
        my_sum += factor/(2 * i + 1);
    }
    pthread_mutex_lock(&mutex);
    sum += my_sum;
    pthread_mutex_unlock(&mutex);
    return NULL;
}/* Thread_sum function*/
```

注意：在使用互斥量的多线程程序中，  
多个线程进入临界区的顺序是随机的

# 练习

## ■ 利用互斥量编写求π值的Pthread程序

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <math.h>

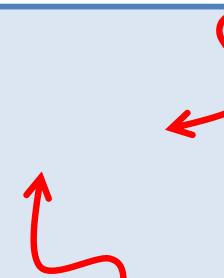
/*Global variables*/
long    thread_count;
double  sum;
pthread_mutex_t mutex;

/*Serial functions*/
double Serial_pi(long long n);

/*Parallel function*/
void* Thread_sum(void* rank);
```

提示

main函数中对mutex进行初始化  
pthread\_mutex\_init(&mutex, NULL);



main函数结束之前对mutex进行销毁  
pthread\_mutex\_destroy(&mutex);

# 比较忙等待和互斥量

Programs Using  $n = 10^8$  Terms on a System  
with Two Four-Core Processors

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \text{thread\_count}$$

当线程数多于核数时候，忙等待效率降低

- ✓ 忙等待：强调访问临界区的顺序
- ✓ 互斥量：访问临界区的顺序随机的

# 互斥量的问题

- 忙等待：能事先确定线程执行临界区代码的顺序
- 互斥量：哪个线程先进入临界区以及此后的顺序由系统随机选取
- 有一些应用，需要控制线程进入临界区的顺序

# 互斥量的问题

```
/*n and product_matrix are shared and initialized by the main  
thread*/  
/* product_matrix is initialized to be the identity matrix */  
void* Thread_work(void* rank) {  
    long my_rank = (long) rank;  
    matrix_t my_mat = Allocate_matrix(n);  
    Generate_matrix(my_mat);  
    pthread_mutex_lock(&mutex);  
    Multiply_matrix(product_mat, my_mat);  
    pthread_mutex_unlock(&mutex);  
    Free_matrix(&my_mat);  
    return NULL;  
}
```

矩阵乘法没有交换律  
所以不能使用互斥量

# 信号量

- 信号量（Semaphore）可以用于临界区，保护共享资源
- 信号量的特性如下：
  - 信号量是一种特殊类型的unsigned int无符号整型变量，可以赋值为0、1、2、...
  - 要访问共享资源的线程必须获取一个信号量，则信号量减1
  - 当信号量为0时，试图访问共享资源的线程将处于等待状态
  - 离开共享资源的线程释放信号量，信号量加1

# 信号量



12个停车位，空3个位置，来了5辆车

# 信号量

```
include <semaphore.h>
```

## ■ 信号量初始化

```
int sem_init(sem_t* semaphore_p, int shared,  
             unsigned initial_val);
```

参数1：信号量指针

参数2：信号量类型（一般设置为0）

参数3：信号量初始值

## ■ 销毁信号量

```
int sem_destroy(sem_t* semaphore_p);
```

释放一个信号量，信号量的值加1

## ■ 发信号

```
int sem_post(sem_t* semaphore_p);
```

申请一个信号量，当前无可用信号量则等待，有可用信号量时占用一个信号量，对信号量的值减1

## ■ 等待信号

```
int sem_wait(sem_t* semaphore_p);
```

# 例：线程发送消息

- 假设t个线程，线程0向线程1发消息，线程1向线程2发消息，。。。, 线程t-2向线程t-1发消息，线程t-1向线程0发消息
- 一个线程“接收”一条消息后，打印消息并终止
- 为实现消息传递，分配一个char\*类型的共享数组，每个线程初始化消息后，共享数组的指针指向要发送的消息
- 主线程把共享数组messages初始化为NULL
- 主线程把信号量初始化为0

# 例：线程发送消息

main函数中对信号量进行初始化

```
semaphores = (sem_t*)malloc(thread_count*sizeof(sem_t));
for(thread=0; thread < thread_count; ++thread) {
    sem_init(&semaphores[thread],0,0);
}
```

```
void* Send_msg(void* rank) {
    long my_rank = (long) rank;
    long dest = (my_rank + 1) % thread_count;
    char* my_msg = malloc( MSG_MAX * sizeof(char) );
    sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
    messages[dest] = my_msg;
    sem_post(&semaphores[dest]);

    sem_wait(&semaphores[my_rank]);
    printf("Thread %ld > %s\n", my_rank, messages[my_rank]);

    return NULL;
} /* Send_msg */
```

# 信号量

- 信号量与互斥量最大的区别在于信号量是没有个体拥有权的，主线程将所有的信号量初始化为0，即“加锁”，其他线程都能对任何信号量调用sem\_post和sem\_wait函数

# 路障

- 同步点 ( Barrier , 屏障/路障 ) : 线程到达路障进入阻塞状态 , 等待所有线程都到达 , 才能继续执行
- 屏障的应用 :
  - 计时
  - 调试程序
  - . . .



# 利用路障计算“最慢”线程的时间

```
/* Shared */  
double elapsed_time;  
...  
/* Private */  
double my_start, my_finish, my_elapsed;  
...  
Synchronize threads; // 所有线程在此同步  
Store current time in my_start;  
/* Execute timed code */  
...  
Store current time in my_finish; // 结束计时  
my_elapsed = my_finish - my_start;  
elapsed_time = Maximum of my_elapsed values;  
// 取所有线程中最大的时间
```

# 利用路障调试程序

```
point in program we want to reach;  
barrier; // 等待所有线程到达该点  
if (my_rank == 0) {  
    printf("All threads reached this point\n");  
    fflush(stdout);  
}
```



# 路障

```
include <pthread.h>
```

参数1 : barrier对象地址  
参数2 : barrier的属性对象  
参数3 : 需要等待的线程数量

## ■ 路障初始化

```
int pthread_barrier_init(pthread_barrier_t *restrict barrier,  
                         pthread_barrierattr_t *restrict attr, unsigned count);
```

## ■ 等待路障

要等待的barrier

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

## ■ 销毁路障

要销毁的barrier

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

# 路障示例

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
pthread_barrier_t barrier;

void* init(void* args)
{
    printf("----thread init work(%d)----\n", time(NULL));
    // 模拟初始化工作。
    sleep(10);
    // 到达路障
    pthread_barrier_wait(&barrier);
    printf("----thread start work(%d) ----\n", time(NULL));
    sleep(10);
    printf("----thread stop work(%d) ----\n", time(NULL));
    return NULL;
}
```

# 路障示例

```
int main(int argc, char* argv[])
{
    // 初始化路障，该路障等待两个线程到达时放行
    pthread_barrier_init(&barrier, NULL, 2);

    printf("****main thread barrier init done****\n");
    pthread_t pid;
    pthread_create(&pid, NULL, &initor, NULL);
    printf("****main waiting(%d)****\n", time(NULL));

    // 主线程到达，被阻塞，当初始化子线程到达路障时才放行。
    pthread_barrier_wait(&barrier);

    pthread_barrier_destroy(&barrier);
    printf("****main start to work(%d)****\n", time(NULL));
    sleep(30);
    pthread_join(pid, NULL);
    printf("****thread complete(%d)****\n", time(NULL));
    return 0;
}
```

```
****main thread barrier init done****
****main waiting(1696299436)****
----thread init work(1696299436)----
----thread start work(1696299446)----
****main start to work(1696299446)****
----thread stop work(1696299456)----
****thread complete(1696299476)****
```

# 条件变量

- 条件变量 ( Conditional Variable ) 使线程在某个特定条件或事件发生前处于挂起状态
- 条件变量数据类型

`pthread_cond_t`

- 条件变量初始化

```
int pthread_cond_init(pthread_cond_t* cond_var_p,  
                      const pthread_condattr_t* attr)
```

- 销毁条件变量

```
int pthread_cond_destroy (pthread_cond_t* cond_var_p)
```

# 条件变量

- 解锁一个阻塞的线程

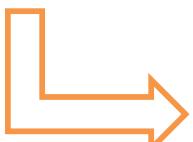
```
int pthread_cond_signal(pthread_cond_t* cond_var_p);
```

- 解锁所有被阻塞的线程

```
int pthread_cond_broadcast(pthread_cond_t* cond_var_p);
```

- 通过互斥量阻塞线程

```
int pthread_cond_wait( pthread_cond_t* cond_var_p,  
                      pthread_mutex_t* mutex_p);
```



```
pthread_mutex_unlock(&mutex_p);  
pthread_cond_wait(&cond_var_p, &mutex_p);  
pthread_mutex_lock(&mutex_p);
```

# 条件变量示例

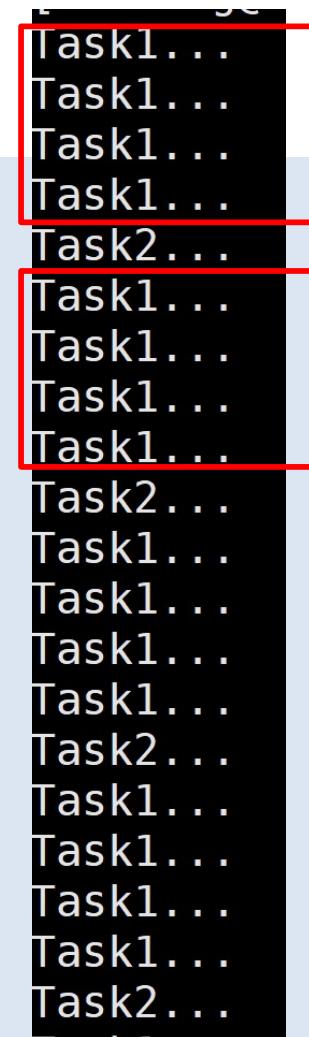
```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t Mutex1,Mutex2;
pthread_cond_t cond;

void *pthread_Task1(void *arg) {
    int cnt = 1;
    while(1) {
        pthread_mutex_lock(&Mutex1);
        printf("Task1...\r\n");
        if(cnt++ % 4 == 0)
            //pthread_cond_signal(&cond);
            pthread_cond_broadcast(&cond);
        pthread_mutex_unlock(&Mutex1);
        sleep(1);
    }
}
```

# 条件变量示例

```
void *pthread_Task2(void *arg) {  
    while(1) {  
        pthread_cond_wait(&cond,&Mutex2);  
        printf("Task2...\r\n");  
    }  
}  
int main(int argc,char **argv) {  
    pthread_t pthread_id1,pthread_id2;  
  
    pthread_mutex_init(&Mutex1,NULL);  
    pthread_mutex_init(&Mutex2,NULL);  
    pthread_cond_init(&cond,NULL);  
  
    pthread_create(&pthread_id1,NULL(pthread_Task1,NULL);  
    pthread_create(&pthread_id2,NULL(pthread_Task2,NULL);  
  
    pthread_join(pthread_id1,NULL);  
    pthread_join(pthread_id2,NULL);  
    return 0;  
}
```



Task1...  
Task1...  
Task1...  
Task1...  
Task1...  
Task2...  
Task1...  
Task1...  
Task1...  
Task1...  
Task1...  
Task2...  
Task1...  
Task1...  
Task1...  
Task1...  
Task1...  
Task2...  
Task1...  
Task1...  
Task1...  
Task1...  
Task2...

# 读写锁

- 读写锁 (Read Write Lock) 把对共享资源的访问者分为读者和写者，读者仅仅对共享资源进行读访问，写者仅仅对共享资源进行写操作
- 读写锁数据类型

`pthread_rwlock_t`

- 初始化

```
int pthread_rwlock_init(pthread_rwlock_t* rwlock_p,  
                      const pthread_rwlockattr_t* attr_p );
```

- 读加锁

```
int pthread_rwlock_rdlock(pthread_rwlock_t* rwlock_p);
```

# 读写锁

## ■ 写加锁

```
int pthread_rwlock_wrlock(pthread_rwlock_t* rwlock_p);
```

## ■ 解锁

```
int pthread_rwlock_unlock(pthread_rwlock_t* rwlock_p);
```

## ■ 销毁

```
int pthread_rwlock_destroy(pthread_rwlock_t* rwlock_p );
```

# 读写锁示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>

#define MAXDATA    1024
#define MAXREDER   100
#define MAXWRITER  100
struct {
    pthread_rwlock_t  rwlock;
    char datas[MAXDATA];
} shared = {
    PTHREAD_RWLOCK_INITIALIZER
};

void *reader(void *arg);
void *writer(void *arg);
```

# 读写锁示例

```
int main(int argc,char *argv[]) {  
    int i,readercount,writercount;  
    pthread_t tid_reader[MAXREDER],tid_writer[MAXWRITER];  
    if(argc != 3) {  
        printf("usage : <reader_writer> #<readercount> #<writercount>\n");  
        return 0;  
    }  
    readercount = atoi(argv[1]);  
    writercount = atoi(argv[2]);  
    for(i = 0; i < writercount; ++i)  
        pthread_create(&tid_writer[i], NULL, writer, NULL);  
    sleep(10);  
    for(i = 0; i < readercount; ++i)  
        pthread_create(&tid_reader[i], NULL, reader, NULL);  
    for(i = 0; i < writercount; ++i)  
        pthread_join(tid_writer[i], NULL);  
    for(i = 0; i < readercount; ++i)  
        pthread_join(tid_reader[i], NULL);  
    return 0;  
}
```

# 读写锁示例

```
void *reader(void *arg) {
    pthread_rwlock_rdlock(&shared.rwlock);
    printf("Reader begins read message.\n");
    sleep(1);
    printf("Read message is: %s\n",shared.datas);
    pthread_rwlock_unlock(&shared.rwlock);
    return NULL;
}

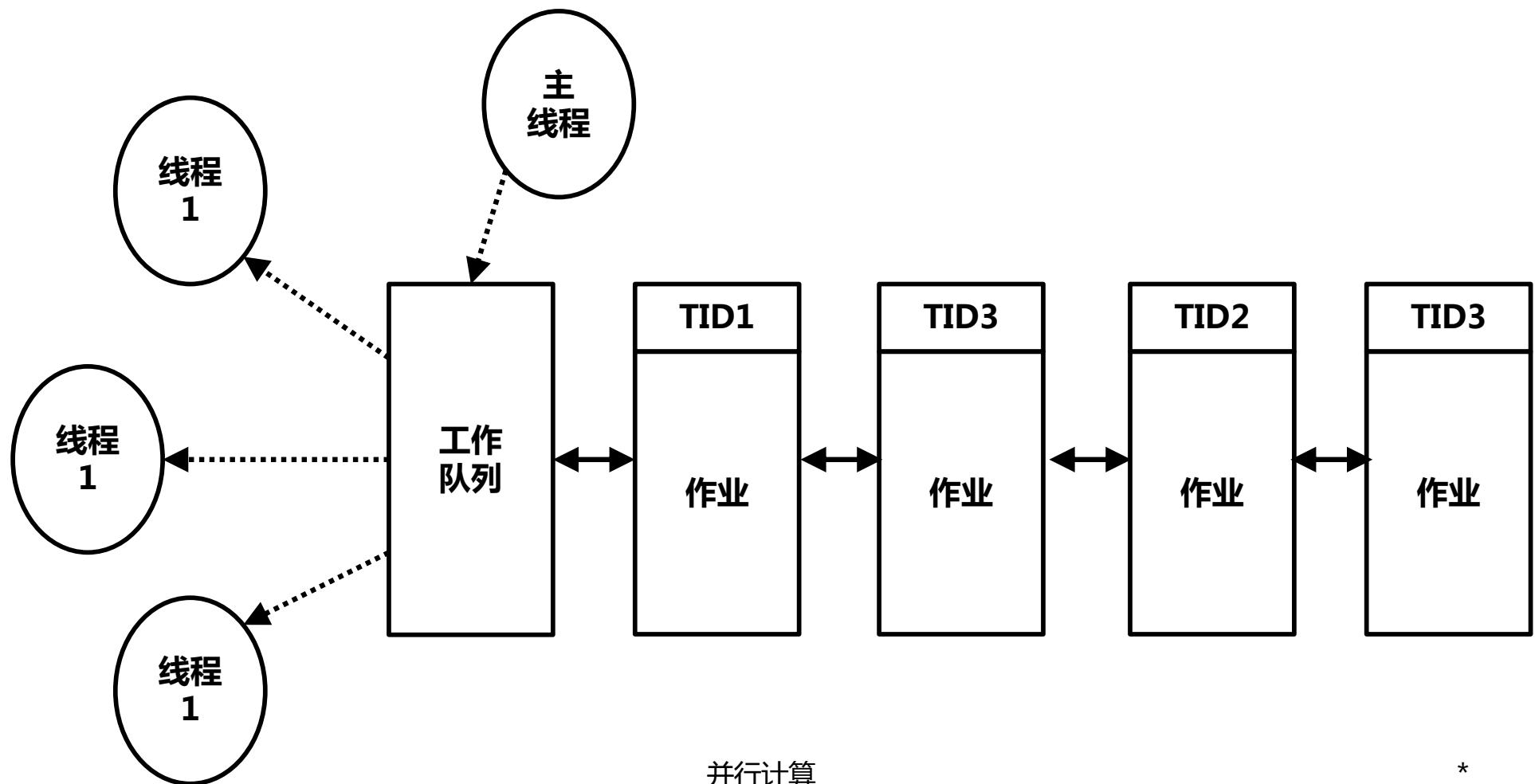
void *writer(void *arg) {
    char datas[MAXDATA];
    pthread_rwlock_wrlock(&shared.rwlock);
    printf("Writers begins write message.\n");
    printf("Enter the write message: \n");
    gets(datas);
    strcat(shared.datas,datas);
    pthread_rwlock_unlock(&shared.rwlock);
    return NULL;
}
```

# 读写锁示例

```
./pth_rwlock_test 2 2↙  
Writers begins write message.  
Enter the write message:  
hello↙  
Writers begins write message.  
Enter the write message:  
world↙  
Reader begins read message.  
Reader begins read message.  
Read message is: helloworld  
Read message is: helloworld
```

# 读写锁应用场景

- 作业请求队列由单个读写锁保护，多个工作线程  
获取单个主线程分配给它们的作业



# Pthreads小结

- 线程管理，创建、终止、分离、同步
- 临界区，互斥量，信号量
- 路障，条件变量，读写锁
- 扩展阅读：
  - **POSIX Threads Programming, by Blaise Barney,  
Lawrence Livermore National Laboratory:  
<https://computing.llnl.gov/tutorials/pthreads/>**

# 目录

- 进程与线程
- 基于Pthread的多线程并行
- 基于OpenMP的多线程并行

# OpenMP简介



<https://www.openmp.org>

- 通过在源代码（串行）中添加 OpenMP 指令和调用 OpenMP 库函数来实现在共享内存系统上的并行
- 为共享内存并行程序员提供了一种简单灵活的开发并行应用的接口模型，使程序既可以在台式机执行，也可以在超级计算机执行，具有良好的可移植性

Jointly defined by a group of major computer hardware and software vendors and major parallel computing user facilities, the OpenMP API is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications on platforms ranging from embedded systems and accelerator devices to multicore systems and shared-memory systems.

<https://www.openmp.org>

API: Application Programming Interface

# OpenMP使用说明

- FORTRAN/C/C++ 自带程序库，无需另外安装
- 编译时不打开OpenMP编译选项，则编译器将忽略 OpenMP指令，从而生成串行可执行程序（串行等价性）
- 打开OpenMP编译选项，编译器将OpenMP指令进行处理，编译生成OpenMP并行执行程序
- 并行线程数可以在程序启动时利用环境变量等方法进行动态设置
- 编程方式：增量并行
- 支持与 MPI 混合编程

# OpenMP发展历史

- 起源于ANSI X3H5 ( 1994 ) 草案，1997年，部分设备商和编译器开发商组成ARB ( 架构审查委员会 )，着手制定OpenMP标准化规范
- 目标：编程简单，增量化并行，移植性好，扩展性好，支持主流编译器
- 支持 Unix , Linux , Windows 等操作系统
- ARB成员：AMD , ARM , Intel , IBM , Cray , HP , NVIDIA , ...

► FORTRAN version 1.0 (1997) , C/C++ version 1.0 (1998)

► FORTRAN version 2.0 (2000) , C/C++ version 2.0 (2002)

► OpenMP 3.0 – (May 2008)

► OpenMP 3.1 – (July 2011)

► OpenMP 4.0 – (July 2013)

► OpenMP 4.5 – (Nov 2015)

► OpenMP 5.0 – (Nov 2018)

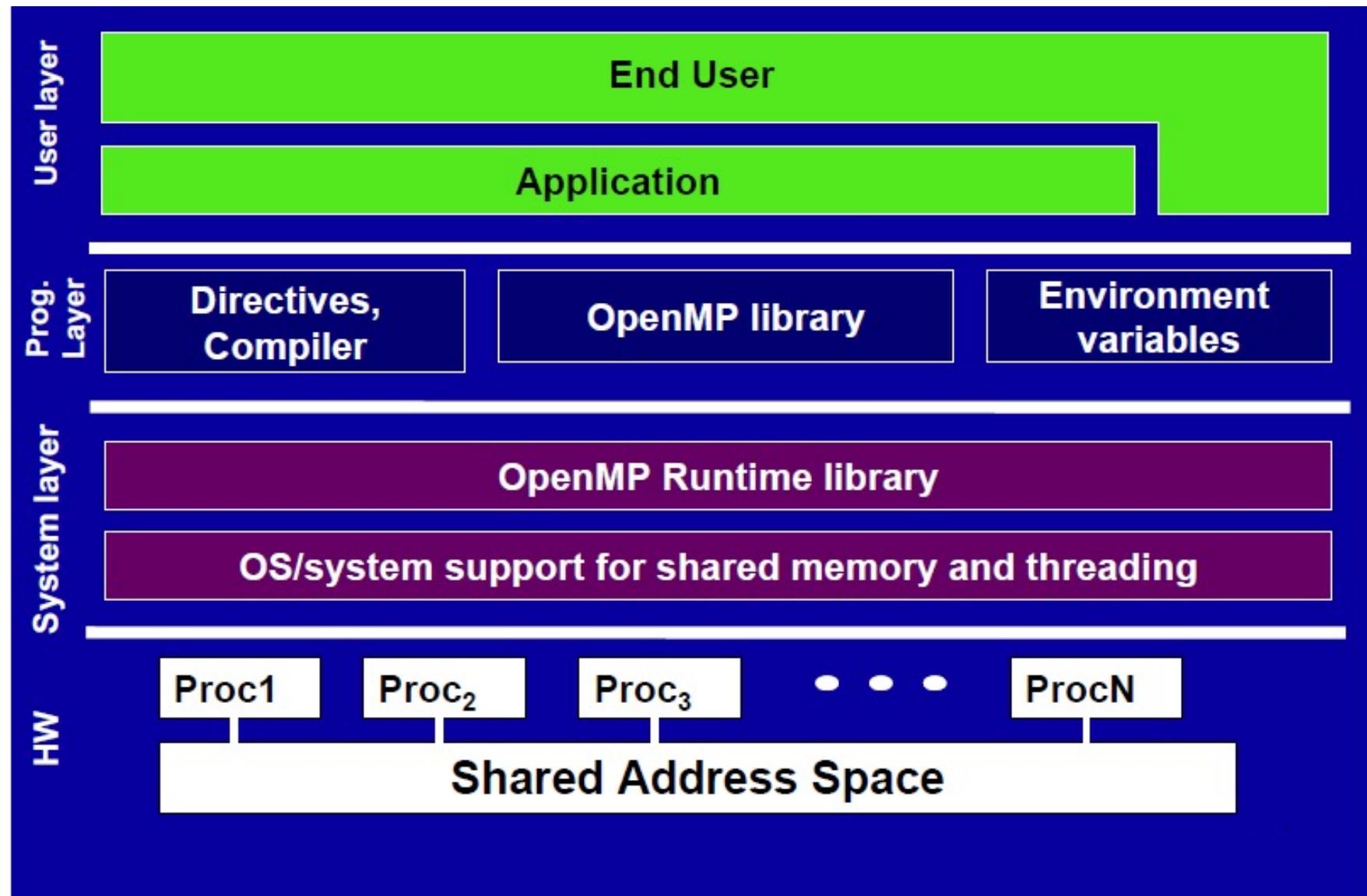
► OpenMP 5.2 – (Nov 2021)



OpenMP  
Application Programming  
Interface

Version 5.2 November 2021

# OpenMP组成



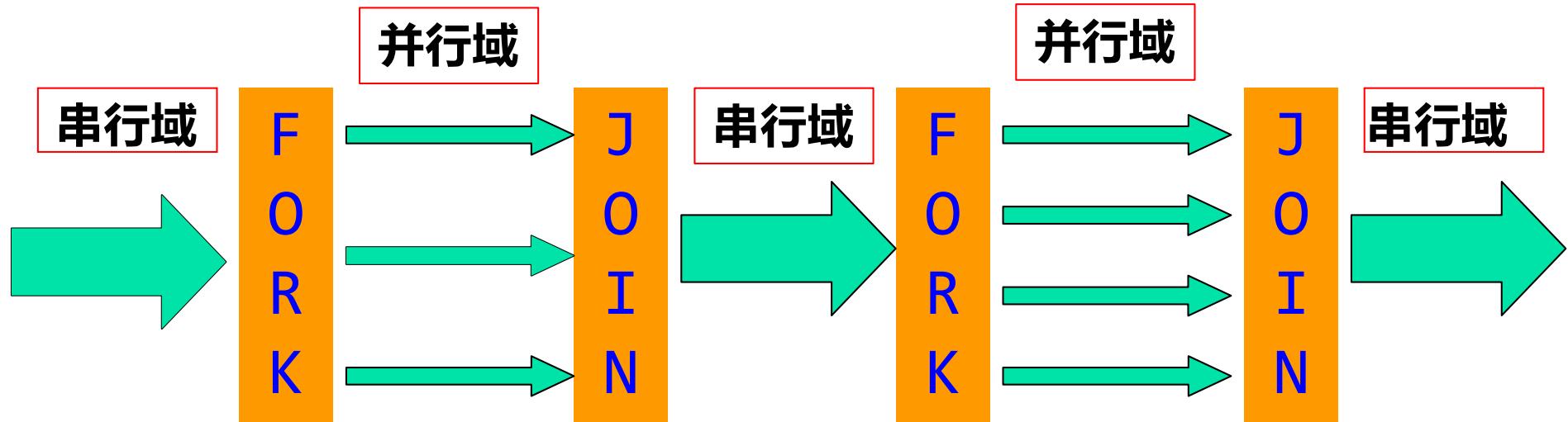
# OpenMP并行方式

OpenMP是基于线程的并行编程模型

OpenMP采用Fork-Join并行执行方式

- ① OpenMP程序开始于一个单独的主线程（Master Thread），然后主线程一直串行执行（串行域）
- ② 直到遇见第一个并行域（Parallel Region），然后开始并行执行并行域
- ③ 并行域代码执行完后再回到主线程，执行串行域，直到遇到下一个并行域
- ④ 以此类推，直至程序运行结束。

# Fork-Join



- **Fork** : 主线程创建一个并行线程队列，然后并行域中的代码在不同的线程上并行执行
- **Join** : 当并行域执行完之后，它们或被同步，或被中断，最后只有主线程继续执行

† 并行域可以嵌套

# “Hello world”

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
void Hello(void); /*Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

#pragma omp parallel num_threads(thread_count)
Hello();

return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /* Hello */
```

- 头文件: `omp.h`
- OpenMP 指令标识符`#pragma omp`

- 编译
  - `gcc -fopenmp omp_hello.c -o omp_hello`

# “Hello world”

## ■ 编译和运行OpenMP程序

**编译:**    `gcc -fopenmp omp_hello.c -o omp_hello`

**运行 :**    `./omp_hello <number of threads>`

```
./omp_hello 4 ↵
```

```
Hello from thread 1 of 4  
Hello from thread 0 of 4  
Hello from thread 3 of 4  
Hello from thread 2 of 4  
Hello from thread 0 of 4
```

```
./omp_hello 4 ↵
```

```
Hello from thread 0 of 4  
Hello from thread 1 of 4  
Hello from thread 2 of 4  
Hello from thread 3 of 4  
Hello from thread 0 of 4
```

**注意 : 线程正在竞争访问标准输出 , 因此不保证输出会按线程编号的顺序出现**

# 几点说明

## ■ OpenMP程序编写

- 通常采用**增量并行方法**：逐步改造现有的串行程序，每次只对部分代码进行并行化，这样可以逐步改造，逐步调试
- C/C++ 的 OpenMP 指令标识符为 **#pragma omp**
- C/C++ 程序中，OpenMP指令**区分大小写**
- OpenMP 指令后是一个结构块（用大括号括起来）

## ■ 源程序编译

```
gcc -fopenmp omp_hello.c -o omp_hello
```

```
icc -openmp omp_hello.c -o omp_hello // Intel C
```

# OpenMP编程三要素

- 编译制导指令 ( Compiler Directive )
- 运行时库函数 ( Runtime Library Routines )
- 环境变量 ( Environment Variables )

# 要素1：编译制导指令

- OpenMP的大部分语句都是编译制导指令

Fortran	<code>!\$OMP construct [clause [clause]...]</code>
C/C++	<code>#pragma omp construct [clause [clause]...]</code>

#pragma omp	construct	clause
制导指令前缀 所有的OpenMP制导 指令语句都有前缀	制导指令 具体的指令，指导编 译器并行化	子句 制导指令之后，负责 添加一些补充设置

- 不支持OpenMP的编译器也能编译OpenMP程序，只是会忽略OpenMP语句

# 编译制导指令

OpenMP通过对串行程序添加**编译制导指令**实现并行化

## 编译制导指令分类

- **并行域指令**：创建并行域，即产生多个线程以并行方式执行任务，所有并行任务必须放在并行域中才能被并行执行
- **工作共享指令**：负责任务划分，并分发给各个线程，工作共享指令不能产生新线程，因此必须位于并行域中
- **同步指令**：负责并行线程之间的同步
- **数据环境**：负责并行域内的变量的属性（共享或私有），以及边界上（串行域与并行域）的数据传递

# 并行域：PARALLEL结构

## ■ 并行域的创建

```
#pragma omp parallel // 创建并行域
```

- 产生多个线程，即生成一个并行域
- 并行域中的所有代码默认都将被所有线程并行执行
- 可以通过线程 id 给不同线程手工分配不同的任务
- 也可以利用**工作共享指令**给每个线程分配任务
- 并行域可以嵌套
- 并行域结束后，将回到主线程

# 并行域：PARALLEL结构

<b>Fortran</b>	<pre>!\$omp parallel [clause clause ...] <i>structured-block</i>  !\$omp end parallel</pre>
<b>C/C++</b>	<pre>#pragma omp parallel [clause clause ...] { <i>structured-block</i> }</pre>

## ■ 结尾处有隐式同步，可用的子句包括：

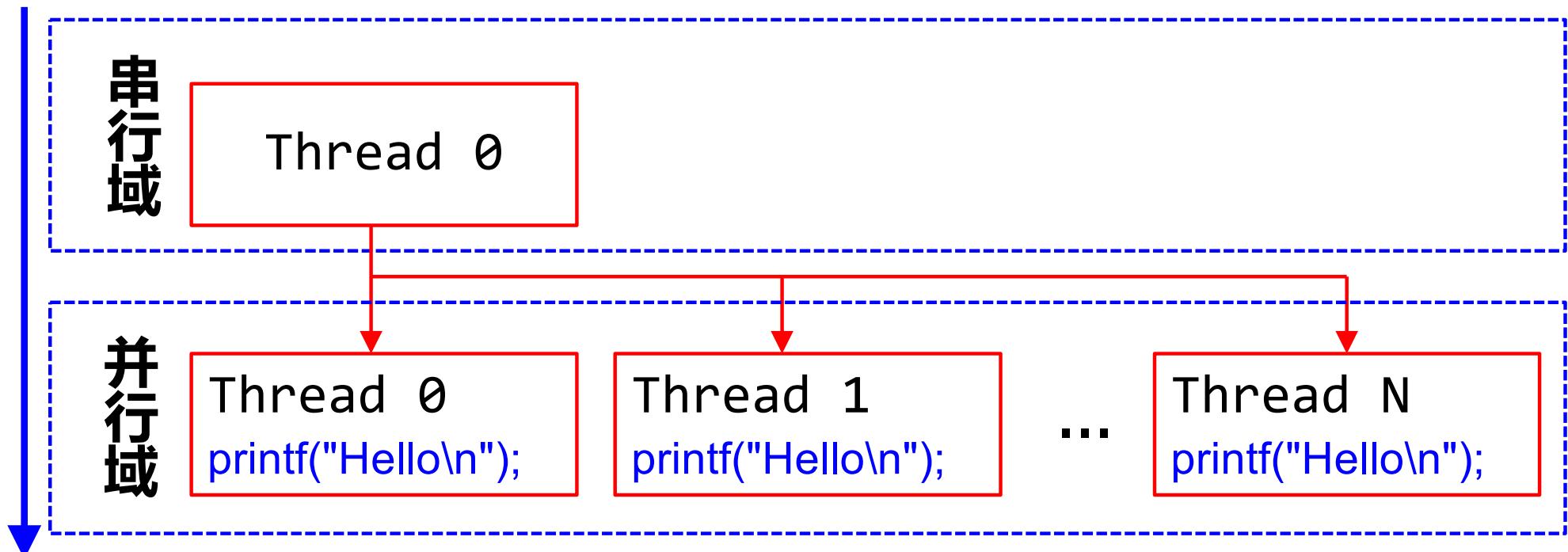
**if** (*scalar-logical-expression*)  
**num\_threads**(*scalar-integer-expression*)  
**default**(*shared* | *none*)  
**private**(*list*), **firstprivate**(*list*)  
**shared**(*list*)  
**copyin**(*list*)  
**reduction**(*op* : *list*)

子句用来添加一些补充信息。  
若多个，则用空格隔开。

若没有指定线程个数，则产  
生最大可能的线程个数。

# 并行域示例

```
#pragma omp parallel  
{  
    printf("Hello\n");  
}
```



# 并行域示例

## ■ 创建一个4-线程的并行域

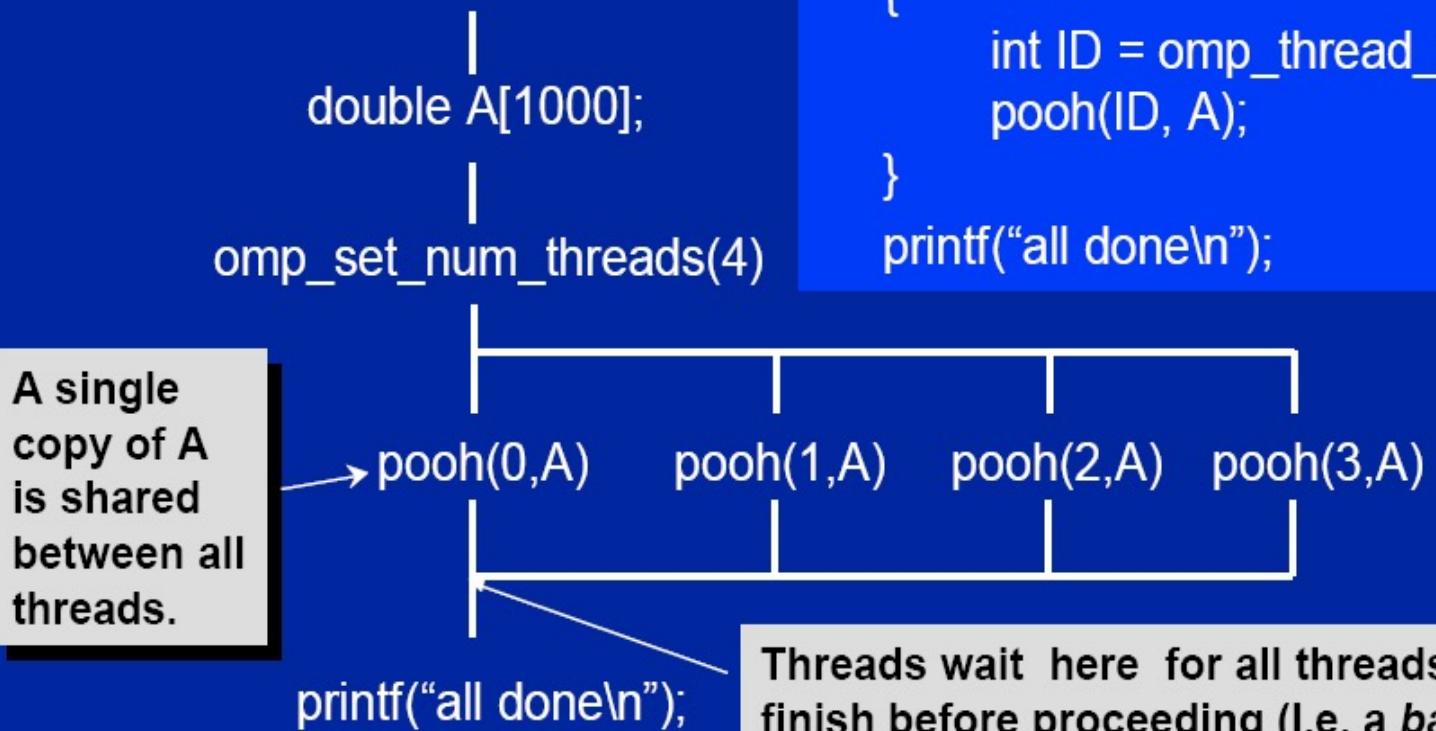
每个线程都执行omp语句  
后面的代码块

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID =omp_get_thread_num();
    pooh(ID,A);
}
printf("all done\n");
```

每个线程调用函数pooh(ID,A), ID = 0 to 3

# 并行域示例

- Each thread executes the same code redundantly.



# 并行域线程数设置

- 通过环境变量设置线程数

```
export OMP_NUM_THREADS=4
```

- 在代码中设置线程数

```
omp_set_num_threads(number_of_threads);
```

或

```
#pragma omp parallel num_threads(number_of_threads)
```

# 工作共享

## ■ 工作共享指令 Work-Sharing Constructs

- 负责任务的划分和分配，
- 在每个工作分享结构入口处无需同步
- 每个工作分享结构结束处会隐含路障同步

- **for** 指令：自动划分和分配循环任务
- **sections** 指令：手动划分任务
- **single** 指令：指定并行域中的串行任务
- **master** 指令：指定仅由主线程执行的串行任务
- **task** 指令：任务并行

# 工作共享：FOR结构

<b>Fortran</b>	<pre>!\$omp do [clause clause ...]       <i>do-loops</i> !\$omp end do</pre>
<b>C/C++</b>	<pre>#pragma omp for [clause clause ...] {     <i>for-loops</i> }</pre>

- 只负责工作分享，不负责并行域的产生和管理，一般需放在并行域中
- 如果不放在并行域内，则只能串行执行
- 结尾处有隐式同步，可用的子句（clause）包括：

**private(*list*)**, **firstprivate(*list*)**, **lastprivate(*list*)**  
**reduction(*op* : *list*)**  
**schedule(*kind*[, *chunk\_size*])**  
**ordered**  
**nowait**

# 工作共享：FOR结构

## 串行循环

- 将循环变量从初始值开始，逐次递增或递减，直至满足结束条件，其间对每个循环变量的取值都将执行一次循环体内的代码，且循环体内的代码是依次串行执行的

## OpenMP并行循环

- 假定总共有N次循环，OpenMP对循环的任务分配就是将这N次循环进行划分，然后让每个并发线程各自负责其中的一部分循环工作，因此必须确保每次循环之间的数据的相互独立性

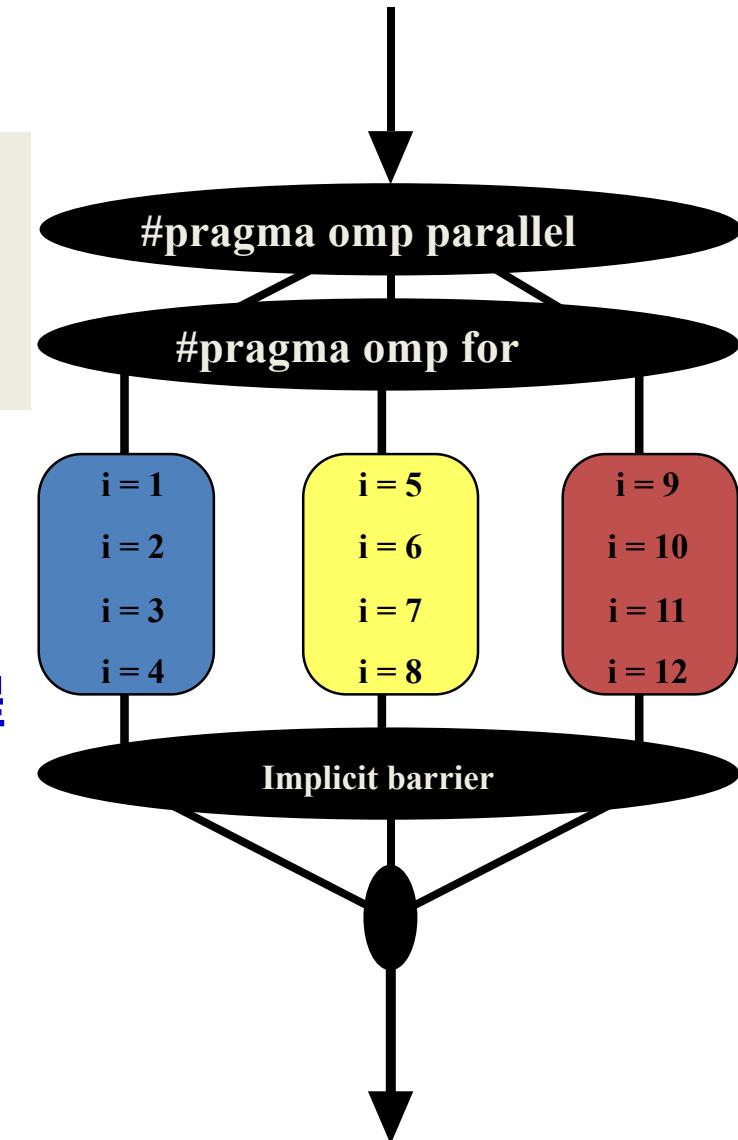
† 循环变量只能是整型或指针

† 将任务划分后分发给并发线程称为“调度”（schedule）

# 工作共享：FOR结构

```
#pragma omp parallel num_threads(4)
#pragma omp for
for(i = 1, i < 13, i++)
    c[i] = a[i] + b[i]
```

- 每个线程分配到一组迭代
- 线程在omp for语句结束后会等待其它线程



# FOR举例

```
#include <omp.h>
#include <stdio.h>
#define N 10000
int main()
{
    int A[N], B[N], i;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp for
        for(i=0; i<N; i++)
        {
            B[i]=i; A[i]=2*B[i];
        }
    }
    printf("A[%d]=%d, B[%d]=%d\n", A[N-1], B[N-1]);
    return 0;
}
```

**for (循环变量赋初值; 循环条件; 循环变量增量)**

**循环体 // 循环体中不能修改循环变量的值**

# 并行域与FOR合并

- 如果并行域中只有循环共享结构，则可以合写在一起

Fortran	<pre>!\$omp parallel do [clause clause ...]     <i>do-loops</i> !\$omp end parallel do</pre>
C/C++	<pre>#pragma omp parallel for [clause clause ...] {     <i>for-loops</i> }</pre>

# FOR举例

```
#define N 100000000
int main()
{
    static int A[N], B[N], i;  clock_t t0, t1;
    t0 = clock();
#pragma omp parallel for num_threads(4)
    for(i=0; i<N; i++){
        B[i]=i;
        A[i]=2*B[i];
    }
    t1 = clock();
    printf("A[%d]=%d, B[%d]=%d\n", A[N-1], B[N-1]);
    printf("Elapsed time: %.2e\n", (double)(t1-t0)/CLOCKS_PER_SEC);
    return 0;
}
```

# 负载均衡问题

```
...
int main()
{
    ...
    sum = 0.0;
    #pragma omp parallel for num_threads(4)
    for(i = 0; i <= n; i++) {
        sum += f(i);
    }
    ...
}
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;
    for(j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
}
```

f函数调用所需要的时间，与参数i的大小成正比

# SCHEDULE

- 在循环共享结构中，将任务划分后分发给各个线程称为调度(schedule)
- 任务调度的方式直接影响程序的效率：(1) 任务的均衡程度；(2) 循环体内数据访问顺序与相应的 cache 冲突情况。

## 循环体任务的调度基本原则

- 分解代价低：分解方法要快速，尽量减少分解任务而产生的额外开销
- 任务计算量要均衡
- 尽量避免高速缓存（cache）冲突，提高 cache 命中率

# 高速缓存cache

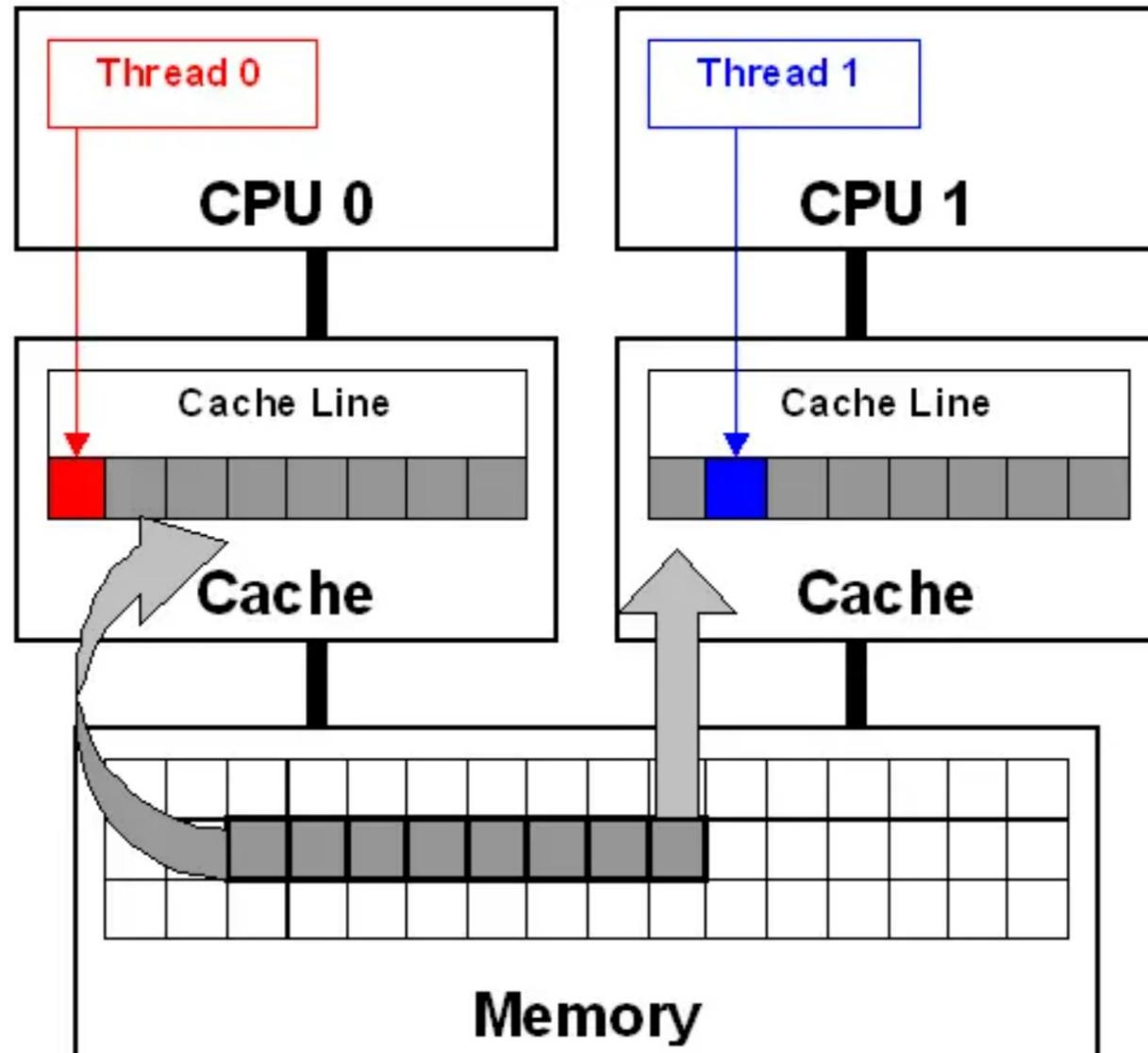
高速缓存 ( cache ) 的关键特性是以连续单元的数据块的形式组成的，当处理器需要引用某个数据块的一个或几个字节时，这个块的所有数据就会被传送到高速缓存中。因此，如果接下来需要引用这个块中的其他数据，则不必再从主存中调用它，这样就可以提高执行效率。

在多处理机系统中，不同的处理器可能需要同一个数据块的不同部分（不是相同的字节），尽管实际数据不共享（处理器有各自的高速缓存），但如果一个处理器对该块的其他部分写入，由于高速缓存的一致性协议，这个块在其他高速缓存上的拷贝就要全部进行更新或者使无效，这就是所谓的“假共享”，它对系统的性能有负面的影响。

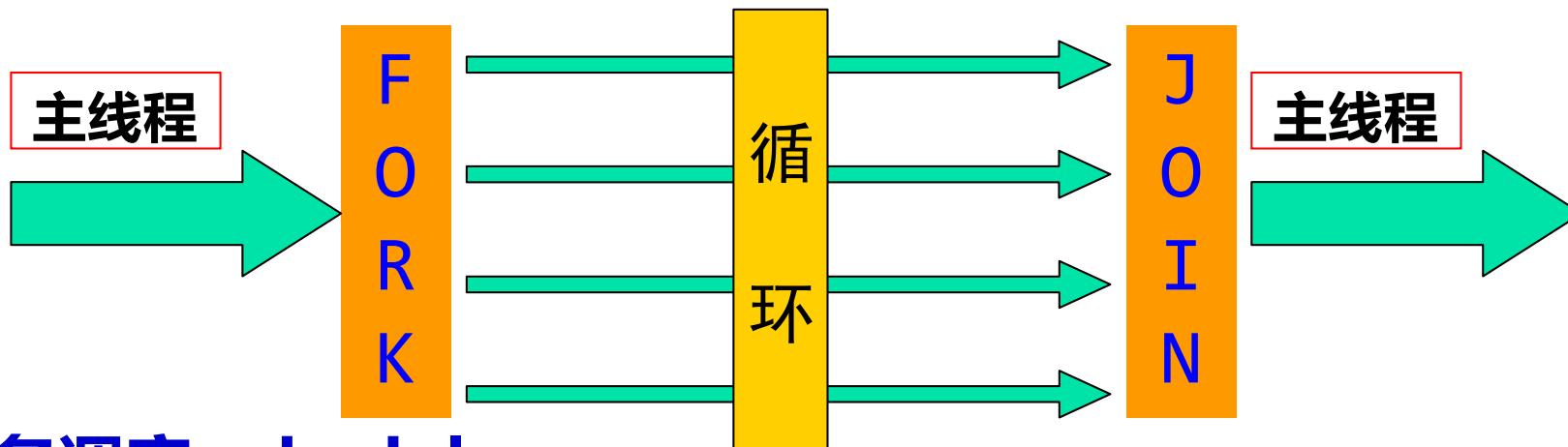
比如：两个处理器A和B访问同一个数据块的不同部分，如果处理器A修改了数据，则高速缓存一致协议将更新或者使处理器B 中的高速缓存块无效。而在此时处理器B可能也修改了数据，则高速缓存一致协议反过来又要将处理器A 中的高速缓存块进行更新或者使无效。如此往复，就会导致高速缓存块的乒乓效应 ( ping-pong effect )。

# 伪共享问题

- 每个CPU对应的Cache把内存中同一段地址区域中（Memory中的灰色区域）的值拷贝过去，并且会保持这段地址中值的一致性
- 当CPU0中修改了这段区域的第一个值（红色所示），那么为了保持一致，会先通过某些机制，更新CPU1的Cache中对应区域的值，然后才能执行其他任务
- CPU1修改了这段地址区域的第二个值（蓝色所示），同样为了保持一致，也会更新CPU0中的Cache中的内容
- 程序会来回地不停更新另一个Cache，导致程序执行变慢



# SCHEDULE



## ■ 任务调度 schedule

- schudule(static, chunk)

静态分配，chunk 为任务块的大小，每个任务块被轮转分配给各线程

- schudule(dynamic, chunk)

动态分配，chunk 为任务块的大小，按先来先服务原则分配

- schudule(guided, chunk)

动态分配，任务块大小可变，先大后小，chunk 指定最小任务的大小

- schudule(runtime)

具体调度方式到运行时才进行，由环境变量OMP\_SCHEDULE 确定  
并行计算

# SCHEDULE方式

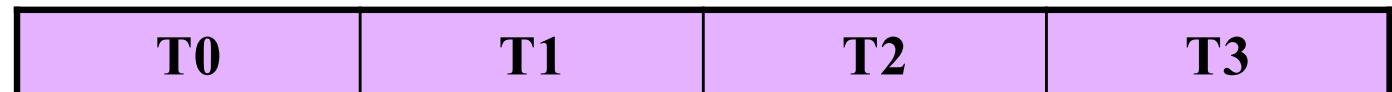
**schudule (static, chunk)**

**schudule (static)**

- 循环任务被划分为chunk 大小的子任务，然后被轮转的分配给各个线程
- 省略chunk，则循环任务被划分成（近似）相同大小的子任务，每个线程被分配一个子任务

**例：**假如线程数为 4，总任务量为 40，则

**schudule(static)**



**schudule(static, 4)**



# SCHEDULE方式

**schudule(dynamic, chunk)**

**schudule(dynamic)**

- 基于先来先服务方式分配给各线程
- 当省略chunk时，默认值为1

**schudule(guided, chunk)**

**schudule(guided)**

- 类似dynamic，但任务块开始较大，然后变小，划分方式取决于编译器
- chunk指定最小任务的大小，省略时默认值为1

**schudule(runtime)**

- 调度延迟到运行时，调度方式取决于环境变量**OMP\_SCHEDULE**的值

# SCHEDULE方式

## ■ 调度策略适用场景

调度策略	适用场景
STATIC	每次迭代的工作是可预测的并且相似的
DYNAMIC	每次迭代的工作是不可预测的，高度不规则的
GUIDED	Dynamic调度策略的一种特例，能够减少dynamic调度策略的负载

# 估计 $\pi$ 值-串行

$$\pi = 4 \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots \right)$$

```
int n = 100000;
double factor = 1.0;
double sum = 0.0;

for(k = 0; k < n; k++) {
    sum += factor/(2 * k + 1);
    factor = -factor;
}
pi = 4.0 * sum;
```

# 估计 $\pi$ 值-OpenMP版本1

```
int n = 100000;
double factor = 1.0;
double sum = 0.0;

#pragma omp parallel for num_threads(thread_count) \
reduction(+:sum)
for(k = 0; k < n; k++) {
    sum += factor/(2 * k + 1);
    factor = -factor;
}
pi = 4.0 * sum;
```

循环依赖或数据依赖问题

# 估计 $\pi$ 值-OpenMP版本2

```
int n = 100000;
double factor = 1.0;
double sum = 0.0;

#pragma omp parallel for num_threads(thread_count) \
reduction(+:sum)
for(k = 0; k < n; k++) {
    factor=(k%2==0)?1.0:-1.0
    sum+=factor/(2*k+1);
}
pi = 4.0 * sum;
```

共享变量问题

# 估计 $\pi$ 值-OpenMP版本3

```
int n = 100000;
double factor = 1.0;
double sum = 0.0;

#pragma omp parallel for num_threads(thread_count) \
reduction(+:sum) private(factor)
for(k = 0; k < n; k++) {
    factor=(k%2==0)?1.0:-1.0
    sum+=factor/(2*k+1);
}
pi = 4.0 * sum;
```

# 估计 $\pi$ 值-OpenMP版本4

```
int n = 100000;
double factor = 1.0;
double sum = 0.0;

#pragma omp parallel for num_threads(thread_count) \
    default.none reduction(+:sum) private(k,factor) shared(n)
for(k = 0; k < n; k++) {
    factor=(k%2==0)?1.0:-1.0
    sum+=factor/(2*k+1);
}
pi = 4.0 * sum;
```

# 矩阵-向量乘法

- 如果  $A = (a_{ij})$  是一个  $m \times n$  的矩阵， $x$  是一个  $n$  维列向量，矩阵-向量的乘积  $Ax = y$  是一个  $m$  维的列向量

$a_{00}$	$a_{01}$	$\cdots$	$a_{0,n-1}$
$a_{10}$	$a_{11}$	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{i0}$	$a_{i1}$	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$

$x_0$	=	$y_0$
$x_1$		$y_1$
$\vdots$		$\vdots$
$x_{n-1}$		$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$
		$\vdots$
		$y_{m-1}$

# 串行伪代码

```
/*For each row of A*/
for(i = 0; i < m; i++) {
    y[i] = 0.0;
    /*For each element of the row and each element of x*/
    for(j = 0; j < n; j++)
        y[i] += A[i][j] * x[j];
}
```

$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j$$

# 矩阵-向量乘法 OpenMP版本

```
/* Spawn a parallel region explicitly scoping all variables */
#pragma omp parallel shared(A,x,y,chunk) private(i,j)
{
    #pragma omp for schedule (static, chunk)
    for(i = 0; i < m; i++) {
        y[i] = 0.0;
        /*For each element of the row and each element of x*/
        for(j = 0; j < n; j++)
            y[i] += A[i][j] * x[j];
    }
}
```

# 工作共享 : SECTIONS 结构

Fortran

`!$omp sections [clause clause ...]`

`!$omp section`

*structured-block*

`!$omp section`

*structured-block*

`!$omp end sections [nowait]`

C/C++

`#pragma omp sections [clause clause ...]`

{

`#pragma omp section`

*structured-block*

`#pragma omp section`

*structured-block*

}

- **sections** 也可以与 **parallel** 合并 , 即 `#pragma omp parallel sections`

# SECTIONS结构

- 指令**sections**创建一个工作共享域
- 域中的子任务由指令**section**创建，必须是一个相对独立的完整代码块
- 每个子任务都将只被一个线程执行
- 结束处隐含障碍同步，除非显式指明**nowait**
- **sections**可用的子句有

**private**(*list*)

**firstprivate**(*list*)

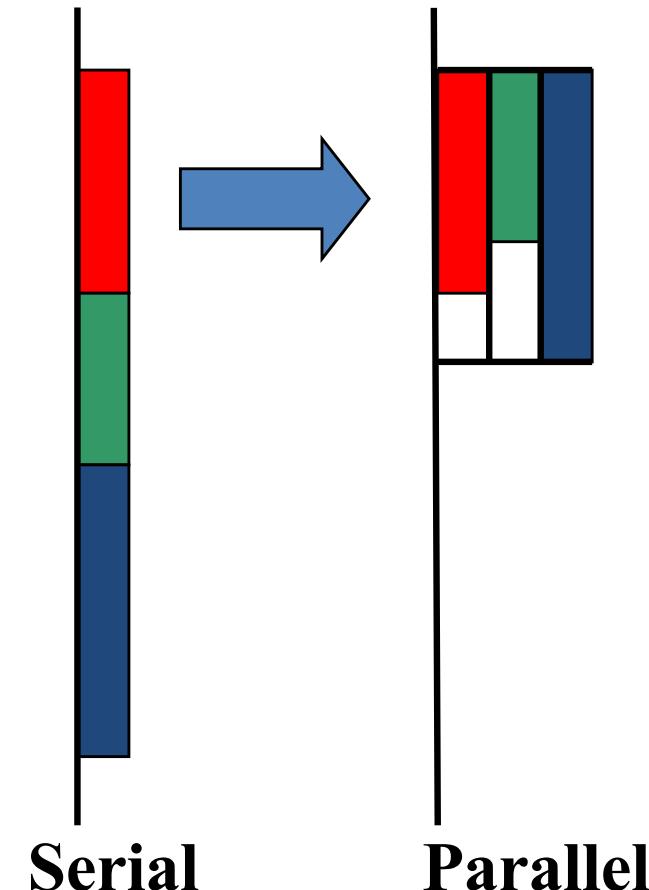
**lastprivate**(*list*)

**reduction**(*op* : *list*)

**nowait**

# SECTIONS结构

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```



- 注意 **sections** 和 **section** 的区别
- **#pragma omp section** 必须在 **sections** 域中
- 共享域中的每个子任务必须由 **section** 指令创建
- 第一个子任务前面的 **section** 指令可以省略
- 子任务个数小于线程个数时，多余的线程空闲等待
- 子任务个数大于线程个数时，任务分配由编译器指定，尽量负载平衡

# 工作共享：SINGLE结构

<b>Fortran</b>	<pre>!\$omp single [clause clause ...]       <i>structured-block</i> \$omp end single [nowait copyprivate(<i>list</i>)]</pre>
<b>C/C++</b>	<pre>#pragma omp single [clause clause ...] {     <i>structured-block</i> }</pre>

- 用在并行域中，指定代码块只能由一个线程执行（不一定是主线程）
- 第一个遇到 **single** 指令的线程执行相应的代码，其它线程则在 **single** 结尾处等待，除非显式指明**nowait**
- 可用的子句有

**private(*list*)**  
**firstprivate(*list*)**  
**copyprivate(*list*)**  
**nowait**

# SINGLE示例

```
#pragma omp parallel private (tmp)
{
    do_many_things();
    #pragma omp single
    {
        exchange_boundaries();
    }
    do_many_other_things();
}
```

# 工作共享：MASTER结构

<b>Fortran</b>	<pre>!\$omp master     <i>structured-block</i> !\$omp end master</pre>
<b>C/C++</b>	<pre>#pragma omp master {     <i>structured-block</i> }</pre>

- **master** 块仅由线程组中的主线程执行
- 其它线程跳过并继续执行下面的代码，即结尾处**没有隐式同步**
- 通常用于 I/O
- 与 **single [nowait]** 的区别：  
**master** 指定由主线程执行，而 **single** 由最先到达的线程执行

# MASTER示例

```
#pragma omp parallel private (tmp)
{
    do_many_things();
    #pragma omp master
    {
        exchange_boundaries();
    }
    #pragma barrier
    do_many_other_things();
}
```

# 工作共享：TASK结构

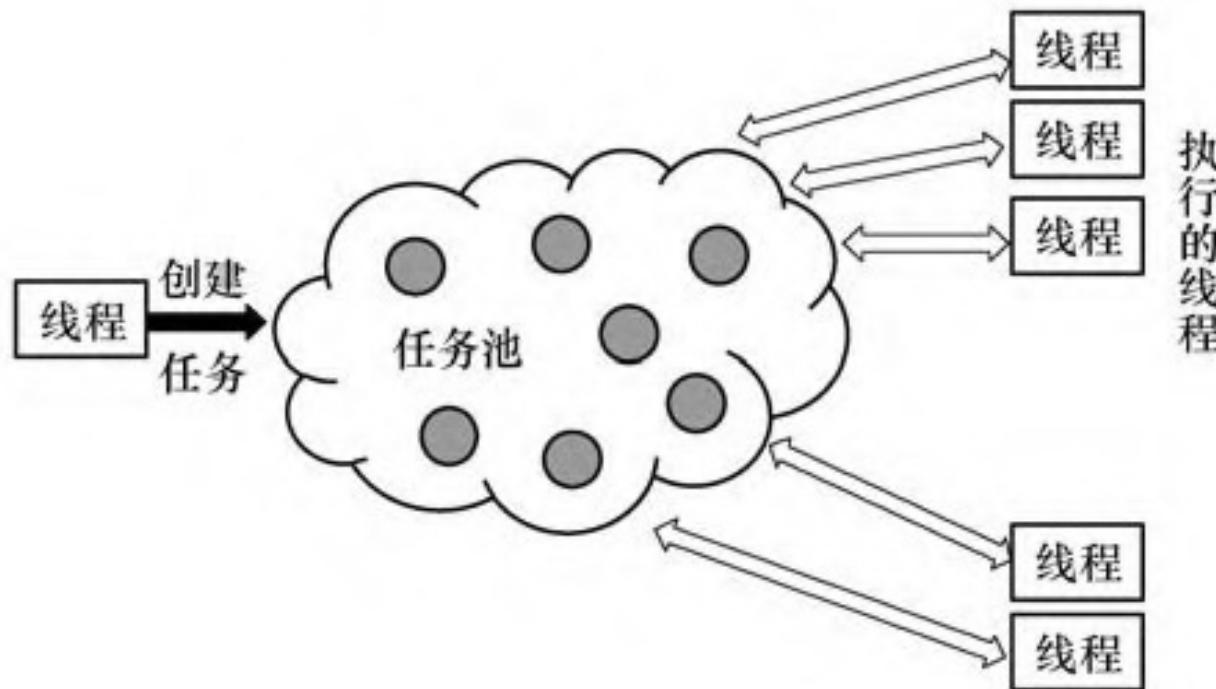
<b>Fortran</b>	<pre>!\$omp task [clause clause ...]       <i>structured-block</i> !\$omp end task</pre>
<b>C/C++</b>	<pre>#pragma omp task [clause clause ...] {     <i>structured-block</i> }</pre>

- **task**定义了一个任务，该任务可能被当前线程立即执行，也可能被加入任务队列中等待条件满足时由线程组中某线程执行
- 任务并行比数据并行粒度更大，且更加灵活，能够处理递归等任务数量不确定的情况

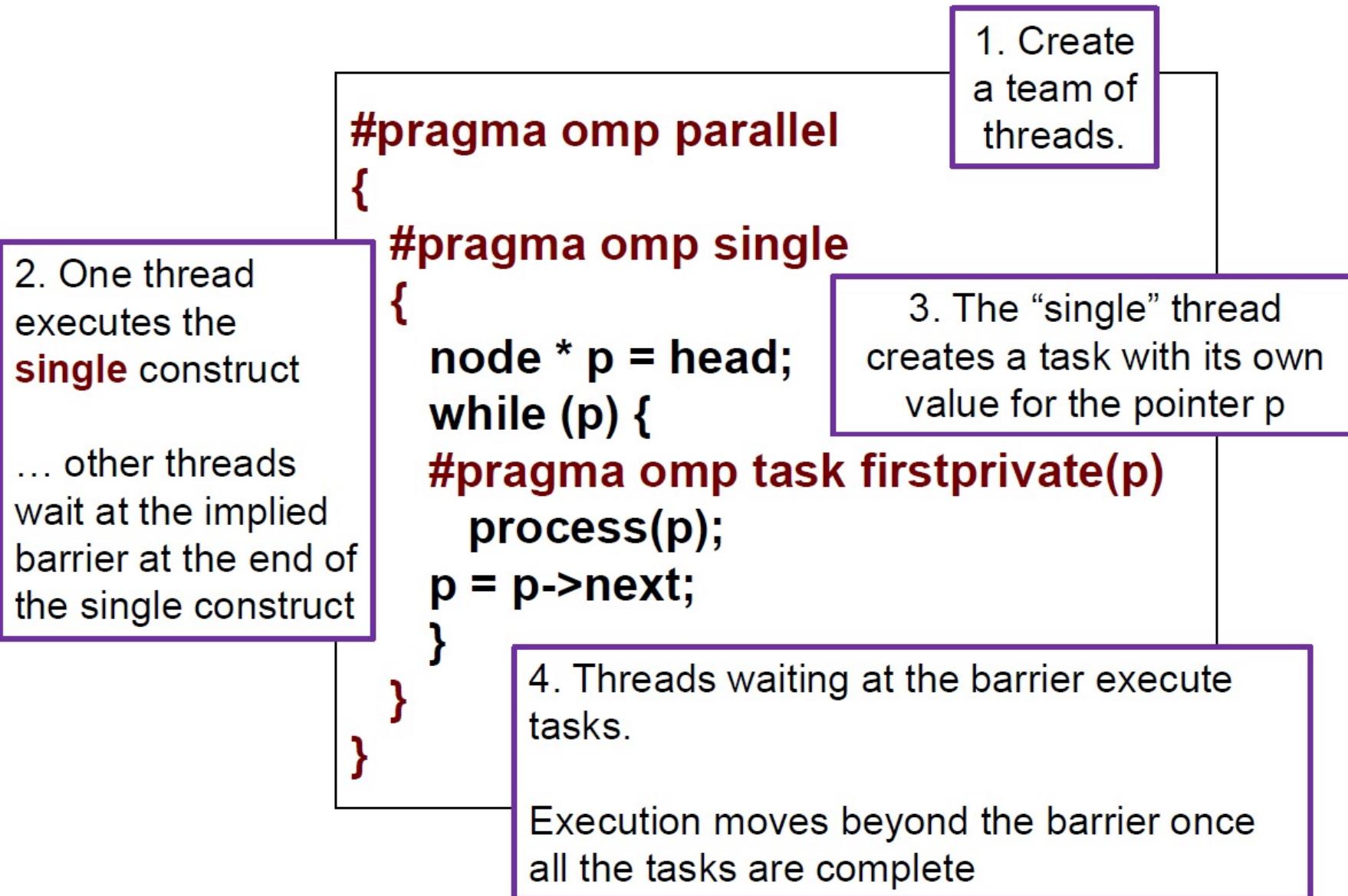
**if (*scalar-expression*)**  
**final(*scalar-expression*)**  
**default(shared | none)**  
**private(*list*), firstprivate(*list*)**  
**shared(*list*)**  
**untied**  
**mergeable**

# 工作共享：TASK结构

- 指令task主要适用于不规则的循环迭代（如do-while）和递归的函数调用，这些都是无法利用指令for完成的情况

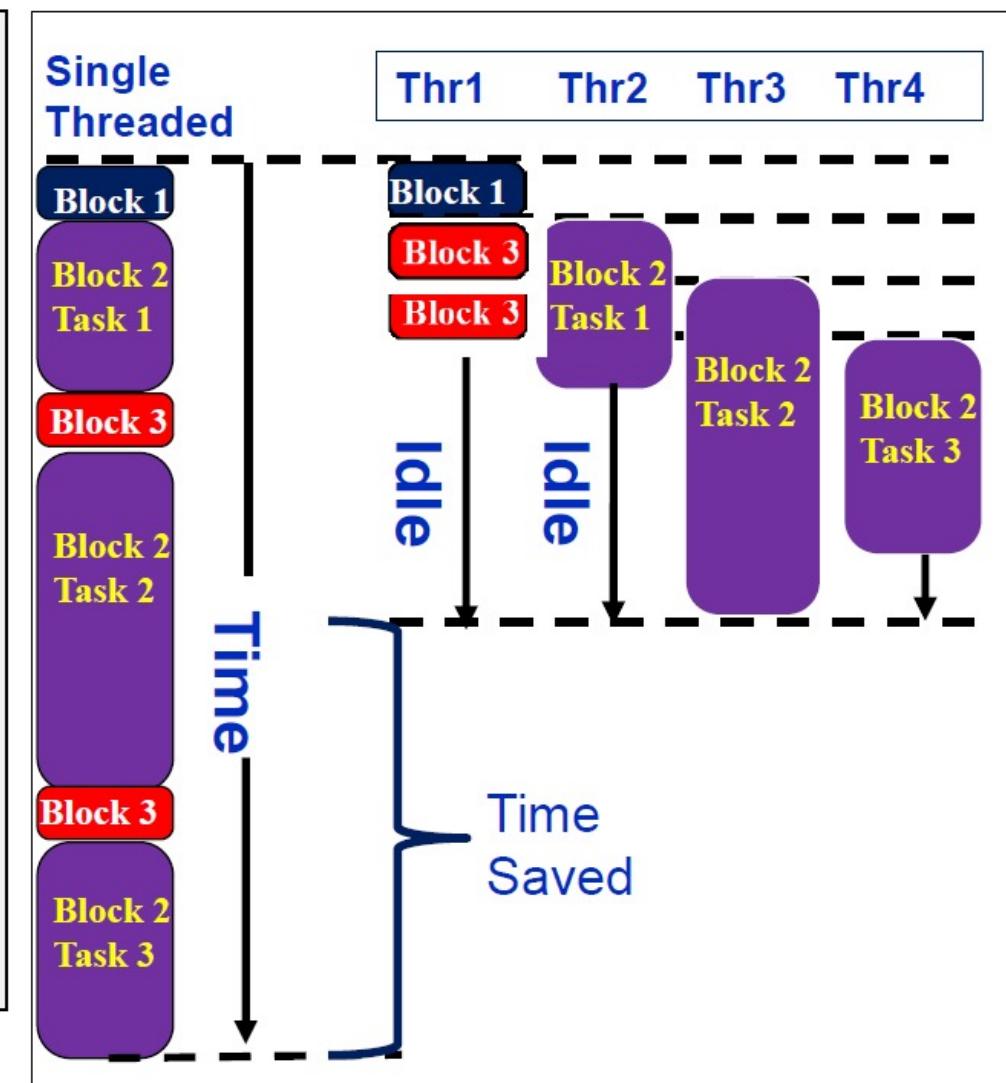


# 工作共享 : TASK结构



# 工作共享：TASK结构

```
#pragma omp parallel
{
    #pragma omp single
    { //block 1
        node * p = head;
        while (p) { // block 2
            #pragma omp task
            process(p);
            p = p->next; //block 3
        }
    }
}
```



# 变量的作用域

- 变量作用域的属性可以通过如下方式修改
  - **default (shared | none)**
- 作用域属性语句
  - **shared(varname,...)**
  - **private(varname,...)**

# 变量的作用域

- 私有变量在每个线程中都有一份存储
  - 私有变量在被创建时是未初始化状态
  - 即使私有变量在进入并行代码块之前已经被初始化，但是在进入并行代码块时仍然是未初始化的

```
void* work(float* c, int N) {  
    float x, y;  
    int i;  
    x = y = 100;  
    #pragma omp parallel for private(x,y)  
    for(i=0; i<N; i++) {  
        x = a[i];  
        y = b[i];  
        c[i] = x + y;  
    }  
    printf("x = %d, y = %d\n", x, y);  
}
```

# 变量的属性

## ■ 如何决定哪些变量是共享哪些是私有？

- 循环变量、临时变量、写变量一般应设置成私有的；
- 数组变量、仅用于读的变量通常是共享的；
- 能设置成共享的变量建议设置成共享的。
- **default(None)**：所有变量必须显式指定是私有或共享

† 紧跟for 结构后面的循环变量默认是私有的，其他循环的  
循环变量需显式声明成私有的

# 数据共享属性子句

## ■ 数据作用域属性子句 Data Sharing Attribute Clauses

<b>private(list)</b>	创建一个或多个变量的私有拷贝，即在每个线程中都创建一个同名局部变量，但没有初始值； 列表中的变量必须已定义，且不能是常量和引用； 列表中的多个变量用逗号隔开。
<b>firstprivate(list)</b>	<b>private</b> 的扩展， 创建私有拷贝的同时，将主线程中的同名变量的值作为初值。
<b>lastprivate(list)</b>	退出并行域时，将指定的私有拷贝的“最后”值复制到主线程中的同名变量中； “最后”：循环的最后一次迭代（按串行方式），或 <b>sections</b> 的最后一个 <b>section</b> （代码中）； 可能会增加额外开销，一般不建议使用，可以用共享变量等方式实现。

# 数据共享属性子句

## ■ 数据作用域属性子句 Data Sharing Attribute Clauses

<b>shared(list)</b>	指定一个或多个变量为共享变量，即所有线程都可以访问这些变量
<b>default(...)</b>	指定并行域内的变量的缺省属性，C语言支持 <b>shared</b> 和 <b>none</b>

# 数据共享属性子句

## ■ 数据作用域属性子句 Data Sharing Attribute Clauses

<code>copyin(list)</code>	配合 <code>threadprivate</code> ，用主线程同名变量的值对 <code>threadprivate</code> 的私有拷贝进行初始化
<code>copyprivate(list)</code>	配合 <code>single</code> ，将 <code>single</code> 块中串行计算得到的变量 值广播到并行域中其它线程的同名变量中
<code>reduction(op:list)</code>	创建一个或多个变量的私有拷贝，在并行结束后 对这些变量执行指定的归约操作（如求和），并将结果返回给主线程中的同名变量

# REDUCTION子句

## ■ 规约操作 reduction

```
void main (){
    int i;
    double ZZ, sum=0.0;
    omp_set_num_threads(NUM_THREADS)
    #pragma omp parallel for reduction(+:sum) private(ZZ)
    for (i=0; i< 1000; i++){
        ZZ = func(i);
        sum = sum + ZZ;
    }
}
```

reduction (op : list)

“list” 中的变量在reduction语句所处的  
并行代码块中必须是共享的

- 在 **reduction** 子句中，编译器为每个线程创建变量 **sum** 的私有拷贝
- 退出并行域时，将这些值加在一起并把结果加到原始变量 **sum** 中
- **reduction** 中的 **op** 操作可以是：**+, -, \*, &&, ||, &, |, ^** 和内置函数 **max, min**

# REDUCTION

## ■ 规约操作时私有变量的初始值

+ * - <b>&amp;&amp;</b> <b>  </b> &   ^	0 1 0 1 0 ~0 0 0
<b>max</b> <b>min</b>	相应变量类型的最小值 相应变量类型的最大值

# OpenMP子句

## ■ 其它字句

<code>if(log_expr)</code>	条件并行，满足指定条件时才执行相关操作
<code>num_threads(int)</code>	指定并行域内线程的个数
<code>nowait</code>	忽略并行线程或其它制导指令中暗含的障碍同步，使用时需小心
<code>schedule(type,chunk)</code>	指定循环任务的分配规则
<code>collapse(int)</code>	将多重循环转换为一层循环，然后进行任务划分

# COLLAPSE子句

```
for(int i = 0; i < 5; i++){
    for(int j = 0; j < num; j++){
        y[i][j] = ...;//computing
    }
}
```



```
#pragma omp parallel for collapse(2)
for(int i = 0; i < 5; i++){
    for(int j = 0; j < num; j++){
        y[i][j] = ...;//computing
    }
}
```



改善负载均衡性，增加每个线程的计算量

# 同步

## ■ 同步结构 Synchronization Constructs

<b>critical</b>	避免线程竞争，其包含的代码同一时刻只能有一个线程执行
<b>barrier</b>	障碍同步：用在并行域内，所有线程执行到 <b>barrier</b> 都要停下等待，直到所有线程都执行到 <b>barrier</b> ，然后再继续往下执行
<b>atomic</b>	确保一个特殊存储单元只能原子更新，即不允许许多线程同时去写，只能用于单一赋值语句等特殊情况
<b>flush</b>	确保线程存储的临时视图与共享存储中的数据一致，并且保证一个变量在共享存储中的读/写顺序
<b>ordered</b>	指定并行域的循环按迭代顺序执行
<b>taskwait</b>	可配合 <b>task</b> 结构使用，创建任务调度点

# 同步 : CRITICAL 结构

Fortran	<pre>!\$omp critical [(name)]     <i>structured-block</i>  !\$omp end critical [(name)]</pre>
C/C++	<pre>#pragma omp critical [(name)] {     <i>structured-block</i> }</pre>

- **critical 块（临界区）限定同一时间只能有一个线程执行**
- **所有线程将依次执行 critical 块**
- **主要用于共享变量的更新，写文件等，避免数据竞争**
- **并行域中可以包含多个critical块**
- **线程到达临界区入口时等待，直到没有其它线程执行同名 critical 块**
- **可以给每个critical块起名字**
- **同名的临界区被看作是一个整体：同一时间，同名块中只能有一个线程**
- **所有没有名字的 critical 块被看作是一个整体**

# CRITICAL示例

```
#pragma omp parallel num_threads(4) private(i,tid,mysum) shared(a,h,mypi)
{
    tid = omp_get_thread_num();

    #pragma omp for
    for(i=1; i<n; i++)
    {
        mysum= mysum + f(a+i*h);
    }
    #pragma omp critical
    mypi = mypi + mysum;
}

mypi = mypi + (f(a) + f(b))/2;
mypi = h*mypi;
```

# 同步 : BARRIER结构

Fortran	<code>!\$omp barrier</code>
C/C++	<code>#pragma omp barrier</code>

- 路障同步：线程遇到该指令都停下来等待，直到同组所有线程都到达该点，然后再继续执行后面的代码
- **barrier** 指令必须放在同组所有线程都能到达或都不能到达的地方，否则可能会引起死锁！比如不能放在 **single** , **critical** , **master** , **section** 中！在循环或选择结构中使用时也需要注意，如：

```
if (myid<5) {  
    #pragma omp barrier  
}
```

# BARRIER示例

- Barrier: Each thread waits until all threads arrive.

```
#pragma omp parallel shared (A, B, C) private(id)
```

```
{
```

```
    id=omp_get_thread_num();
```

```
    A[id] = big_calc1(id);
```

```
#pragma omp barrier
```

```
#pragma omp for
```

```
    for(i=0;i<N;i++){C[i]=big_calc3(i,A);} ←
```

```
#pragma omp for nowait
```

```
    for(i=0;i<N;i++){ B[i]=big_calc2(C, i); } ←
```

```
    A[id] = big_calc3(id);
```

```
}
```

implicit barrier at the  
end of a for work-  
sharing construct

implicit barrier at the end  
of a parallel region

no implicit barrier  
due to nowait

# 同步 : ATOMIC结构

Fortran	<code>!\$omp atomic [clause]</code>
C/C++	<code>#pragma omp atomic [clause]</code>

- **atomic**指令保证对某个存储地址做原子改写，不允许多个线程同时改写
- 该指令仅作用于紧随其后的一条语句，所支持的运算参见OpenMP手册  
支持的子句：**read | write | update | capture**
- 使用 **atomic** 和 **critical** 都可以避免数据竞争
- 使用 **atomic** 可能比 **critical** 更高效：  
使用 **atomic** 指令可以并行地更新数组内的不同元素  
而使用**critical**的话，则只能串行执行

原子改写的含义是指：读取该存储地址的内容、做所需运算、然后把新 值写回该存储地址这一连串操作不会被其它线程间断，它保证所有操作 要么全部完成，要么保持原封不动。

# ATOMIC示例

```
#pragma omp parallel for shared(x, y, index, n)
for (i = 0; i < n; i++) {
    #pragma omp atomic
    x[index[i]] += work1(i);
    y[i] += work2(i);
}
```



atomic语句是critical语句的一个特例  
只应用在对某个内存地址进行简单的更新操作，可以并行地更新数组内的不同元素

# 同步 : FLUSH 结构

Fortran	<code>!\$omp flush [(list)]</code>
C/C++	<code>#pragma omp flush [(list)]</code>

- **flush** 标识数据同步点，在这些点上，系统将提供一致的内存视图，在该指令出现的点上，共享变量被写回内存
- 不带 **list**，则表示针对所有变量
- 下列指令隐含不带 **list** 的 **flush** 指令：  
**barrier** , **critical** , **parallel** , **ordered**

注：若有 **nowait** 子句，则不再隐含 **flush** 指令

# 同步 : ORDERD 结构

<b>Fortran</b>	<pre>!\$omp ordered     <i>structured-block</i>  !\$omp end ordered</pre>
<b>C/C++</b>	<pre>#pragma omp ordered {     <i>structured-block</i> }</pre>

- 指定循环必须按串行时的顺序执行
- **ordered** 只能出现在 **for** 或 **parallel for** 指定的循环区
- 一个循环体中最多只能有一个 **ordered** 块

# ORDERD示例

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

int main()
{
    const int niter = 10;
    #pragma omp parallel for ordered
    for (int i = 0; i < niter; i++) {
        int thr = omp_get_thread_num();
        printf("unordered iter %d of %d on thread %d\n", i, niter, thr);
        #pragma omp ordered           // 这里是需要顺序执行的部分
        printf("ordered iter %d of %d on thread %d\n", i, niter, thr);
    }
    return 0;
}
```

unordered iter 0 of 10 on thread 0  
ordered iter 0 of 10 on thread 0  
unordered iter 1 of 10 on thread 0  
ordered iter 1 of 10 on thread 0  
unordered iter 2 of 10 on thread 0  
unordered iter 8 of 10 on thread 3  
unordered iter 3 of 10 on thread 1  
ordered iter 2 of 10 on thread 0  
ordered iter 3 of 10 on thread 1  
unordered iter 4 of 10 on thread 1  
ordered iter 4 of 10 on thread 1  
unordered iter 5 of 10 on thread 1  
ordered iter 5 of 10 on thread 1  
unordered iter 6 of 10 on thread 2  
ordered iter 6 of 10 on thread 2  
unordered iter 7 of 10 on thread 2  
ordered iter 7 of 10 on thread 2  
ordered iter 8 of 10 on thread 3  
unordered iter 9 of 10 on thread 3  
ordered iter 9 of 10 on thread 3

# 数据环境

## ■ 数据环境指令 Data Environment Constructs

**threadprivate(list)**

将一个或多个私有变量声明为全局的，即在多个并行域中使用时，保留私有变量在上次并行域中的值；  
可以与**copyin**子句联合使用，将主线程的值广播给其他线程。

**Fortran**    `!$omp threadprivate(list)`

**C/C++**    `#pragma omp threadprivate(list)`

- 指定变量或公共数据块是线程私有的，且在同一个线程内是全局的（多个并行域）
- **threadprivate** 需出现在变量声明之后，其它语句之前
- 数据在同一个线程内是全局的

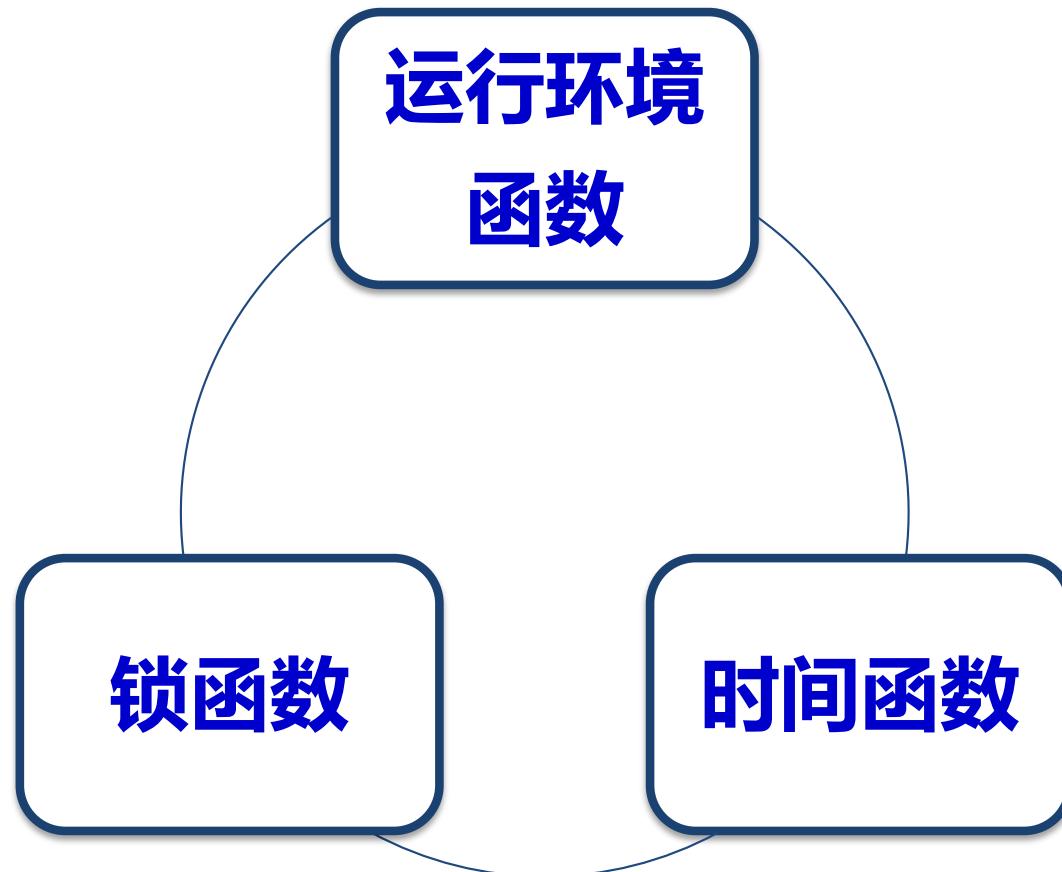
# THREADPRIVATE示例

```
int a;
#pragma omp threadprivate(a)
#pragma omp parallel // 第一个并行域
{
    a=omp_get_thread_num();
}

. . .
#pragma omp parallel // 第二个并行域
{
    . . .
}
```

- 变量 a 在第一个并行域中获取不同的值
- 在第二个并行域内，a保持原有的值

# 要素2：运行时库函数



# 运行环境函数

<code>omp_set_num_threads(int)</code>	设置并行域中的线程个数（用在串行域中）
<code>omp_get_num_threads()</code>	返回当前并行域中的线程个数
<code>omp_get_max_threads()</code>	返回并行域中缺省可用的最大线程个数
<code>omp_get_thread_num()</code>	返回当前线程的线程号，0号为主线程
<code>omp_get_num_procs()</code>	返回系统中处理器的个数
<code>omp_in_parallel()</code>	判断是否在并行域中
<code>omp_set_dynamic(int)</code>	启用或关闭线程数目动态改变功能 （用在串行域中）
<code>omp_get_dynamic()</code>	判断系统是否支持动态改变线程数目
<code>omp_set_nested(int)</code>	启用或关闭并行域嵌套功能（缺省为关闭）
<code>omp_get_nested()</code>	判断系统是否支持并行域的嵌套

# 锁函数

## 什么是锁

- OpenMP提供了一些锁函数用于避免任务竞争，作用类似于atomic和critical
- OpenMP锁是一个特殊的变量，称为锁变量
- 锁变量的取值有：uninitialized, unlocked, locked
- 如果一个锁处于 unlocked状态，则任务就可以设置该锁，并拥有这个锁，只有拥有该锁的任务才能解锁，其他任务只有等解锁后才能使用这个锁
- 锁变量有两类：简单锁和嵌套锁，后者的区别在于可以设置多次而不阻塞，分别是omp\_lock\_t 和omp\_nest\_lock\_t

# 锁函数

锁操作：初始化，销毁，上锁，解锁，测试

■ 锁函数一般按如下顺序进行调用

- ① 用 `omp_init_lock/ omp_init_nest_lock` 初始化锁变量
- ② 调用 `omp_set_lock/ omp_set_nest_lock` 上锁，是阻塞型函数
- ③ 线程须调用 `omp_unset_lock/ omp_unset_nest_lock` 来解锁
- ④ 当不再需要该锁，调用 `omp_destroy_lock/ omp_destroy_nest_lock` 来释放锁资源，即设为为初始化
- ⑤ 函数 `omp_test_lock/ omp_test_nest_lock` 尝试去上锁，非阻塞型，若上锁成功，返回 1（简单锁）或嵌套层数（嵌套锁），否则返回 0

# 锁函数

<code>omp_init_lock(omp_lock_t * lock)</code>	初始化一个简单锁
<code>omp_destroy_lock(omp_lock_t * lock)</code>	销毁一个简单锁
<code>omp_set_lock(omp_lock_t * lock)</code>	上锁操作
<code>omp_unset_lock(omp_lock_t * lock)</code>	解锁操作
<code>omp_test_lock(omp_lock_t * lock)</code>	非阻塞上锁操作
<code>omp_init_nest_lock(omp_nest_lock_t * lock)</code>	初始化一个嵌套锁
<code>omp_destroy_nest_lock(omp_nest_lock_t * lock)</code>	销毁一个嵌套锁
<code>omp_set_nest_lock(omp_nest_lock_t * lock)</code>	上锁操作
<code>omp_unset_nest_lock(omp_nest_lock_t * lock)</code>	解锁操作
<code>omp_test_nest_lock(omp_nest_lock_t * lock)</code>	非阻塞上锁操作

# 锁函数举例

```
int main()
{
    int k;    omp_lock_t lock;

    omp_init_lock(&lock); // 初始化锁
    #pragma omp parallel num_threads(4) private(k)
    {
        omp_set_lock(&lock); // 上锁
        for(k=0; k<4; k++)
            printf("myid=%d, k=%d\n", omp_get_thread_num(), k);
        omp_unset_lock(&lock); // 解锁
    }
    omp_destroy_lock(&lock); // 释放锁
    return 0;
}
```

```
myid=1, k=0
myid=1, k=1
myid=1, k=2
myid=1, k=3
myid=0, k=0
myid=0, k=1
myid=0, k=2
myid=0, k=3
myid=2, k=0
myid=2, k=1
myid=2, k=2
myid=2, k=3
myid=3, k=0
myid=3, k=1
myid=3, k=2
myid=3, k=3
```

# 时间函数

<code>omp_get_wtime()</code>	获取 wall time, 以秒为单位, 双精度型的实数
<code>omp_get_wtick()</code>	获取每个时钟周期的秒数, 即 <code>omp_get_wtime</code> 的精度

## ● `omp_get_wtime`

**returns elapsed wall clock time in seconds**

<b>Fortran</b>	<code>double precision function omp_get_wtime()</code>
<b>C/C++</b>	<code>double omp_get_wtime(void);</code>

## ● `omp_get_wtick`

**returns the number of seconds between successive clock ticks**

<b>Fortran</b>	<code>double precision function omp_get_wtick()</code>
<b>C/C++</b>	<code>double omp_get_wtick(void);</code>

# 时间函数举例

```
double t0, t1;  
  
t0=omp_get_wtime();  
... work to be timed ...  
t1=omp_get_wtime();  
printf("Work took %f seconds\n", t1-t0);
```

C/C++

# 要素3：环境变量

OpenMP 提供了一环境变量来控制并行代码的执行

<b>OMP_SCHEDULE</b>	设置循环任务的调度模式
<b>OMP_NUM_THREADS</b>	设置线程个数
<b>OMP_DYNAMIC</b>	设置是否开启线程数的动态变化功能
<b>OMP_PROC_BIND</b>	设置线程是否与处理器绑定
<b>OMP_NESTED</b>	设置是否开启并行域的嵌套功能
<b>OMP_STACKSIZE</b>	线程的栈的大小，缺省单位是K
<b>OMP_WAIT_POLICY</b>	设置线程等待时是否占用处理器资源
<b>OMP_MAX_ACTIVE_LEVELS</b>	设置并行域嵌套最大层数
<b>OMP_THREAD_LIMIT</b>	设置整个程序所能使用的最大线程数

# 环境变量

## ■ 在 Linux 下修改环境变量的一般格式

```
export 环境变量名 = 值
```

```
echo $环境变量名 // 查看变量的值
```

```
export OMP_SCHEDULE=dynamic
```

```
export OMP_SCHEDULE="guided,4"
```

```
export OMP_NUM_THREADS=4
```

```
export OMP_PROC_BIND=true
```

```
export OMP_PROC_BIND=false
```

# Reference

- OpenMP Application Programming Interface,  
<https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf>
- Pthread, <https://hpc-tutorials.llnl.gov posix/>
- Intel® oneAPI Threading Building Blocks,  
<https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html#gs.sxzdka>
- Blaise Barney, Lawrence Livermore National Laboratory, Introduction to Parallel Computing, [https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)



# 第五章

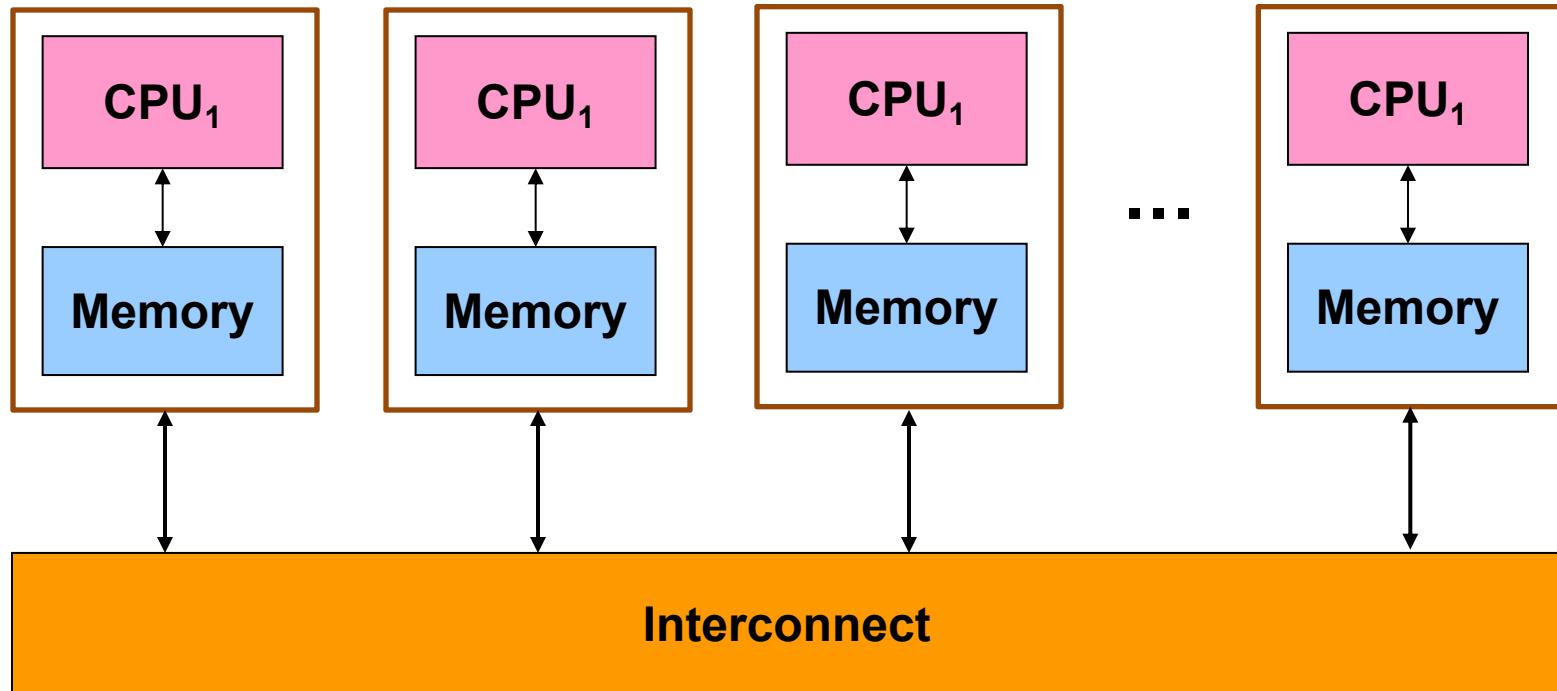
# 基于分布式内存的并行计算

哈尔滨工业大学

张伟哲

2025, Fall Semester

# 分布式内存系统



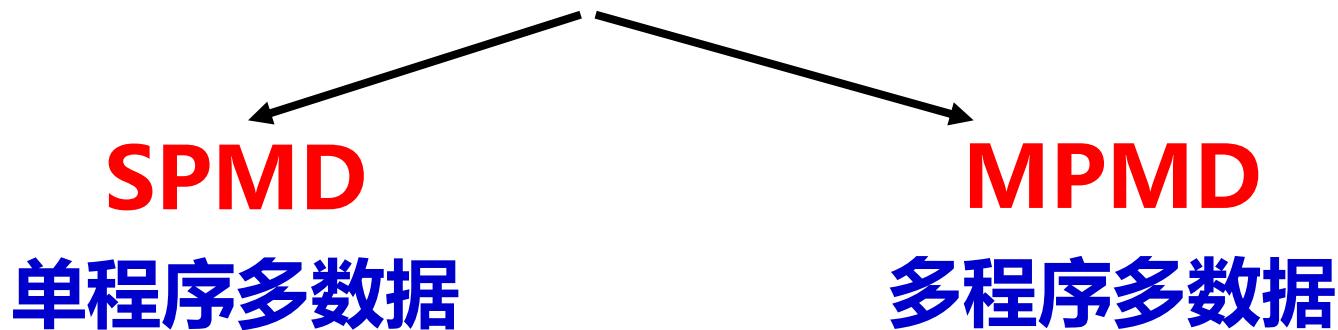
集群

节点可以通过互联网络发送和接收信息来  
与其他节点进行通信

# 消息传递编程的原理

- 在消息传递程序中，每个处理器运行一个单独的进程（子程序、任务）
  - 支持消息传递范式机器的逻辑视图是由P个进程组成，每个进程都有自己的**专用地址空间**
- 所有变量都是**私有的**
  - 每个数据元素必须属于空间的一个分区；因此，必须对于数据进行显示分区和放置
- 通过特殊的**子函数调用**进行通信
  - 所有交互（只读或读/写）都需要两个进程的协作——拥有数据的进程和想要访问数据的进程

# 消息传递编程的原理



- 同一个程序
- 每个进程只知道/操作一小部分数据
- 每个进程执行不同的功能（输入、问题设置、解决方案、输出、显示）

# MPI介绍

## MPI: Message Passing Interface

- <https://www mpi-forum.org/>
- 消息传递编程标准，提供一个高效、可扩展、统一的并行编程环境，是目前最为通用的分布式并行编程方式。
- MPI是一种消息传递编程模型，是一种标准或规范，MPI实现通过提供库函数实现进程间通信，从而进行并行计算，目前所有并行机制制造商都提供对MPI的支持。
- MPI是一个库，不是一门语言，最终目的是服务于进程间通信

The goal of the Message-Passing Interface, simply stated, is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

# MPI介绍

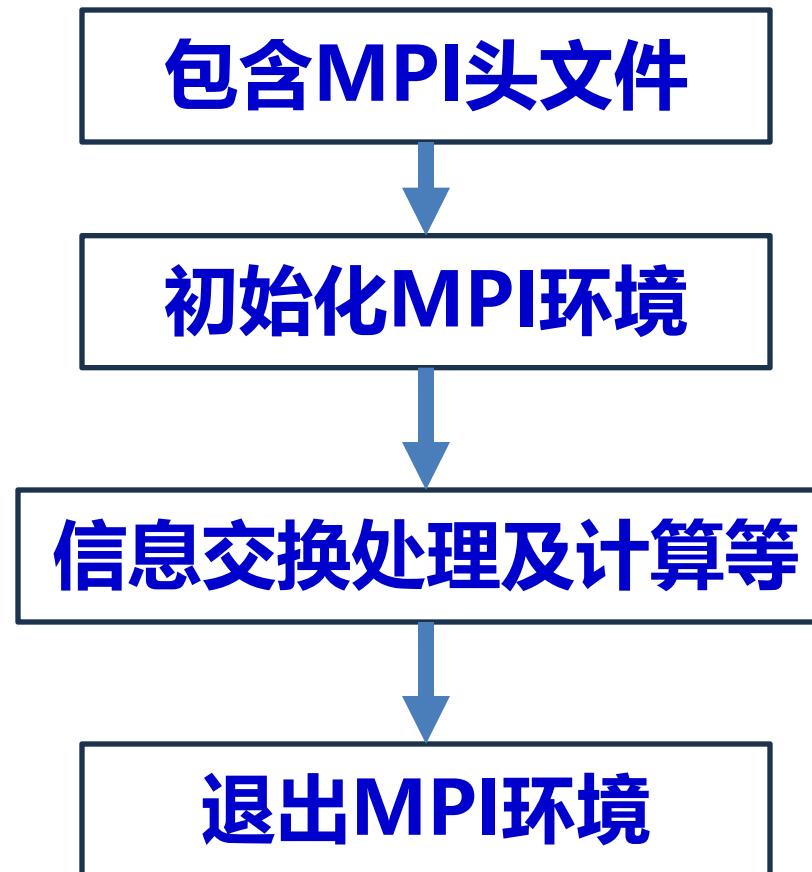
## MPI 的目标

- 高通信性能，高可移植性，强大的功能
- Practical, Portable, Efficient, Flexible

## MPI 标准和 MPI 实现

- 1994 年 MPI-1.0; 1998 年 MPI-2.0; 2012 年 MPI-3.0; MPI-3.1 (2015); MPI-4.0 (2021) ; MPI-5.0
- 支持 C/C++ 和 Fortran (目前以 Fortran 90 为主)
- MPI 实现 (免费版) : MPICH 和 OpenMPI
- MPI 实现 (商业版) : Intel MPI, IBM MPI, HP-MPI, ...
- 所有版本都遵循 MPI 标准，可以不加修改地运行

# MPI程序的一般结构



# 第一个MPI程序

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

int main(int argc, char * argv[])
{
    MPI_Init(&argc, &argv);

    printf("Hello World!\n");

    MPI_Finalize();
    return 0;
}
```

```
mpicc -O2 -o hello mpi_hello.c
mpirun -np 4 ./hello
```

# 程序分析

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

MPI相对于C 语言的头文件

```
int MPI_Init(int *argc, char ***argv)
```

MPI程序的开始，在调用其他MPI函数之前被调用。其目的是初始化 MPI 环境

```
MPI_Finalize()
```

MPI 程序的结束，在计算结束时被调用，它执行各种清理任务以终止 MPI 环境

# Hello程序如何执行

## ■ SPMD: Single Program Multiple Data

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

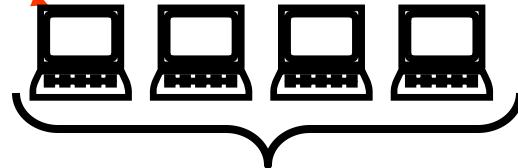
main(
    int argc,
    char *argv[])
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
}
```



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

main(
    int argc,
    char *argv[])
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
}
```

rsh\ssh



```
Hello World!
Hello World!
Hello World!
Hello World!
```

# **MPI\_Comm\_size 和 MPI\_Comm\_rank**

■ 在写MPI程序时，我们通常需要知道以下两个问题的答案：

- 任务由多少个进程来进行并行计算？
- 我是哪一个进程？

# 第二个MPI程序

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
int main(int argc, char * argv[])
{
    int myid, np;    int namelen;
    char proc_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Get_processor_name(proc_name,&namelen);
    fprintf(stderr,"Hello, I am proc. %d of %d on %s\n",
    myid, np, proc_name);
    MPI_Finalize();
}
```

```
int MPI_Get_processor_name( char *name, int *resultlen )
```

# 程序分析

**MPI\_MAX\_PROCESSOR\_NAME**

预定义的宏，即 MPI 所允许的机器名字的最大长度

**int MPI\_Comm\_rank(MPI\_Comm comm, int \*rank)**

返回本进程的进程号，是一个整数，范围从零到通信器的大小减一

**int MPI\_Comm\_size(MPI\_Comm comm, int \*size)**

函数，返回所有参加运算的进程的个数

**int MPI\_Get\_processor\_name( char \*name, int \*resultlen )**

函数，返回运行本进程所在的结点的主机名

# MPI通信器

## 通信器/通信子 (Communicator)

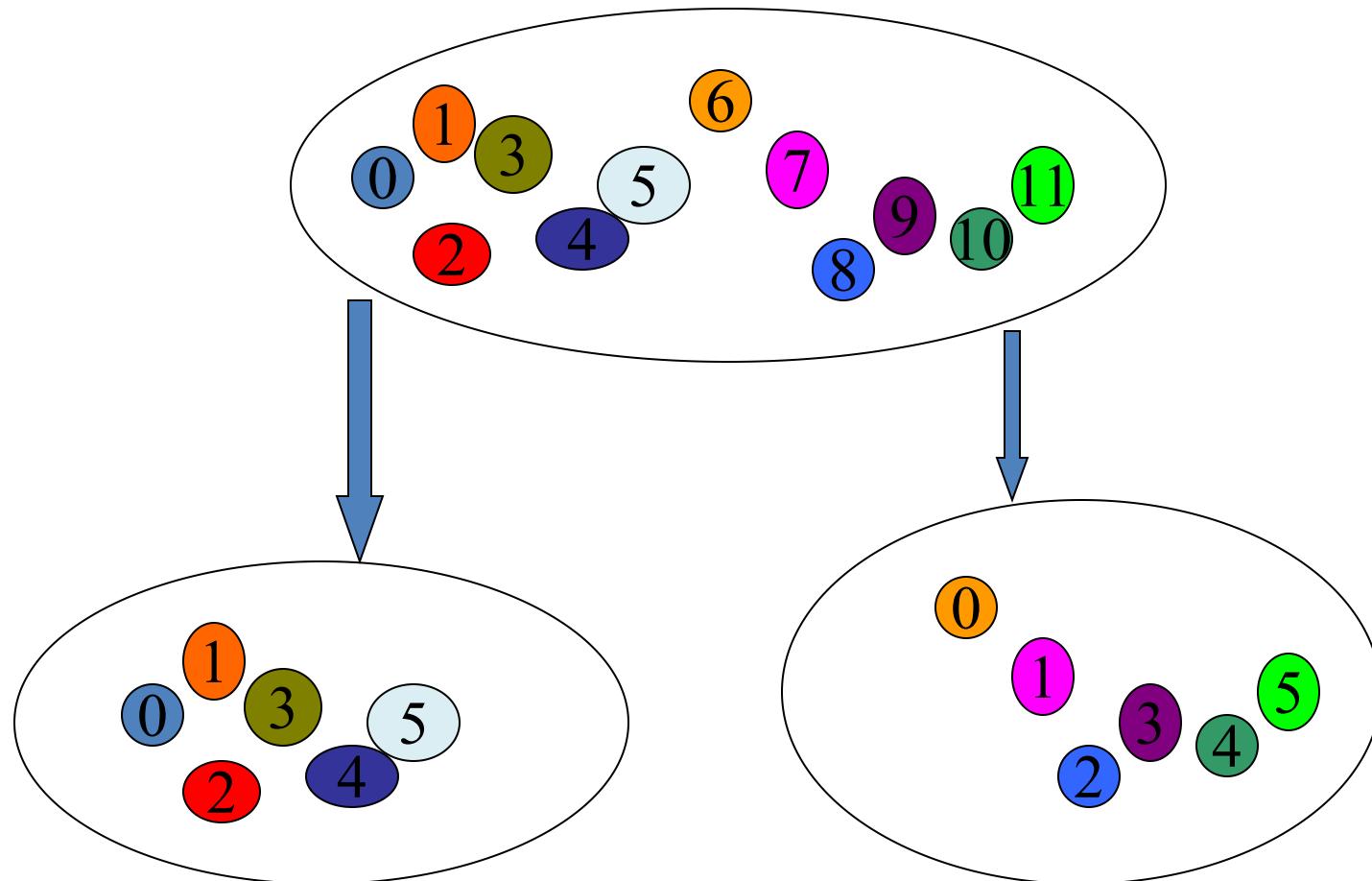
- 一个通信器定义一个通信域 (communication domain) —— 一组允许相互通信的进程
- 有关通信域的信息存储在 MPI\_Comm类型的变量中
- MPI程序中进程间的通信必须通过通信器进行
- 在执行函数MPI\_Init之后，一个MPI程序的所有进程形成一个缺省的组,这个组的通信域即被写作MPI\_COMM\_WORLD
- MPI通信操作函数中必不可少的参数，用于限定参加通信的进程的范围

# MPI通信器

## MPI进程

- MPI 程序中一个独立参与通信的个体
- MPI 进程组：由部分或全部进程构成的有序集合
- 进程号是相对进程组或通信器而言的，同一进程在不同的进程组或通信器中可以有不同的进程号
- 进程号是在进程组或通信器被创建时赋予的
- 进程号取值范围为  $0, \dots, np-1$

# MPI通信器



# 程序运行结果

## ■ 在单个节点上，开 4 个进程的运行结果

Hello, I am proc. 1 of 4 on node1

Hello, I am proc. 0 of 4 on node1

Hello, I am proc. 2 of 4 on node1

Hello, I am proc. 3 of 4 on node1

## ■ 在四个节点上，开 4 个进程的运行结果

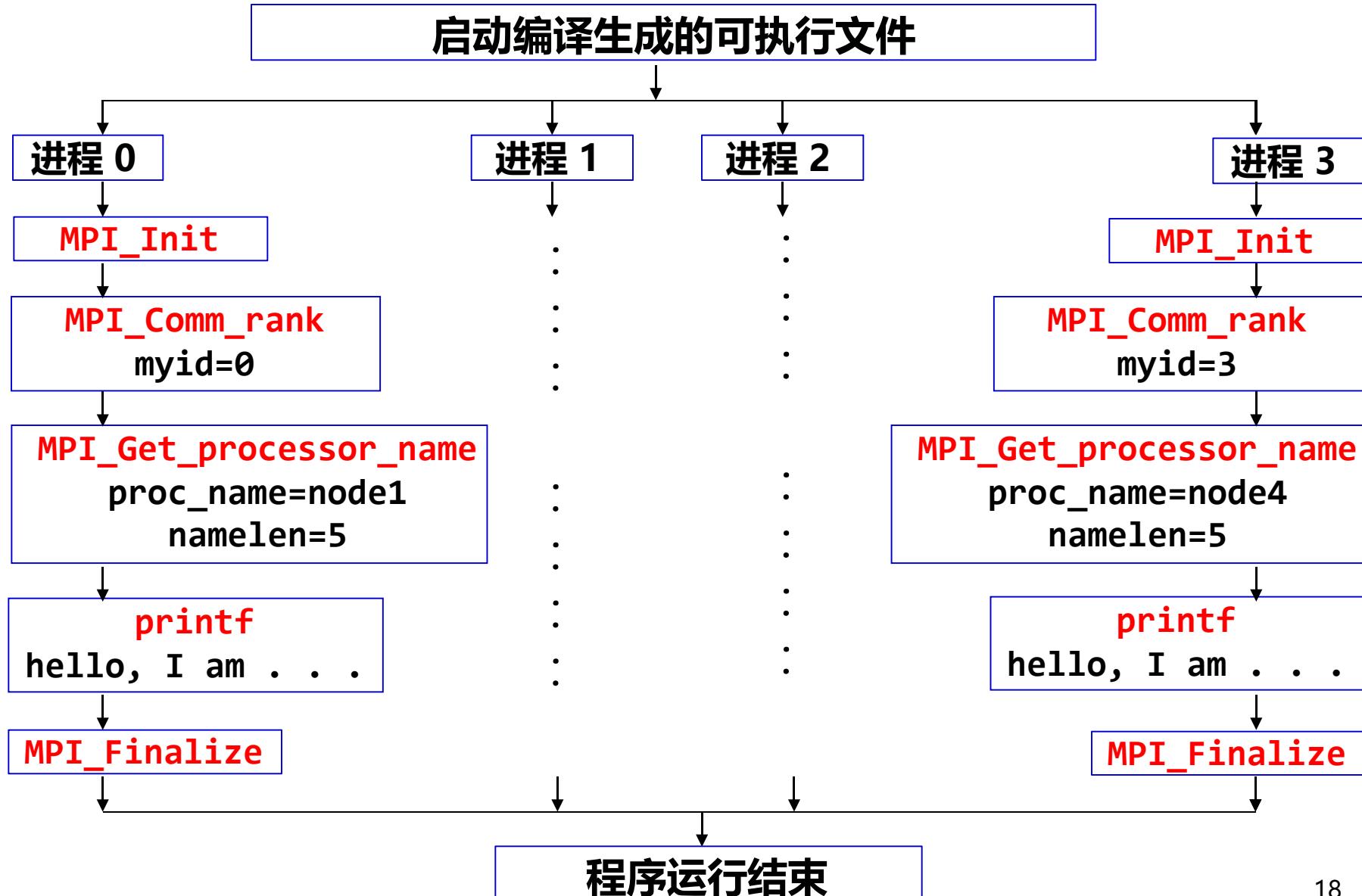
Hello, I am proc. 1 of 4 on node1

Hello, I am proc. 0 of 4 on node2

Hello, I am proc. 2 of 4 on node3

Hello, I am proc. 3 of 4 on node4

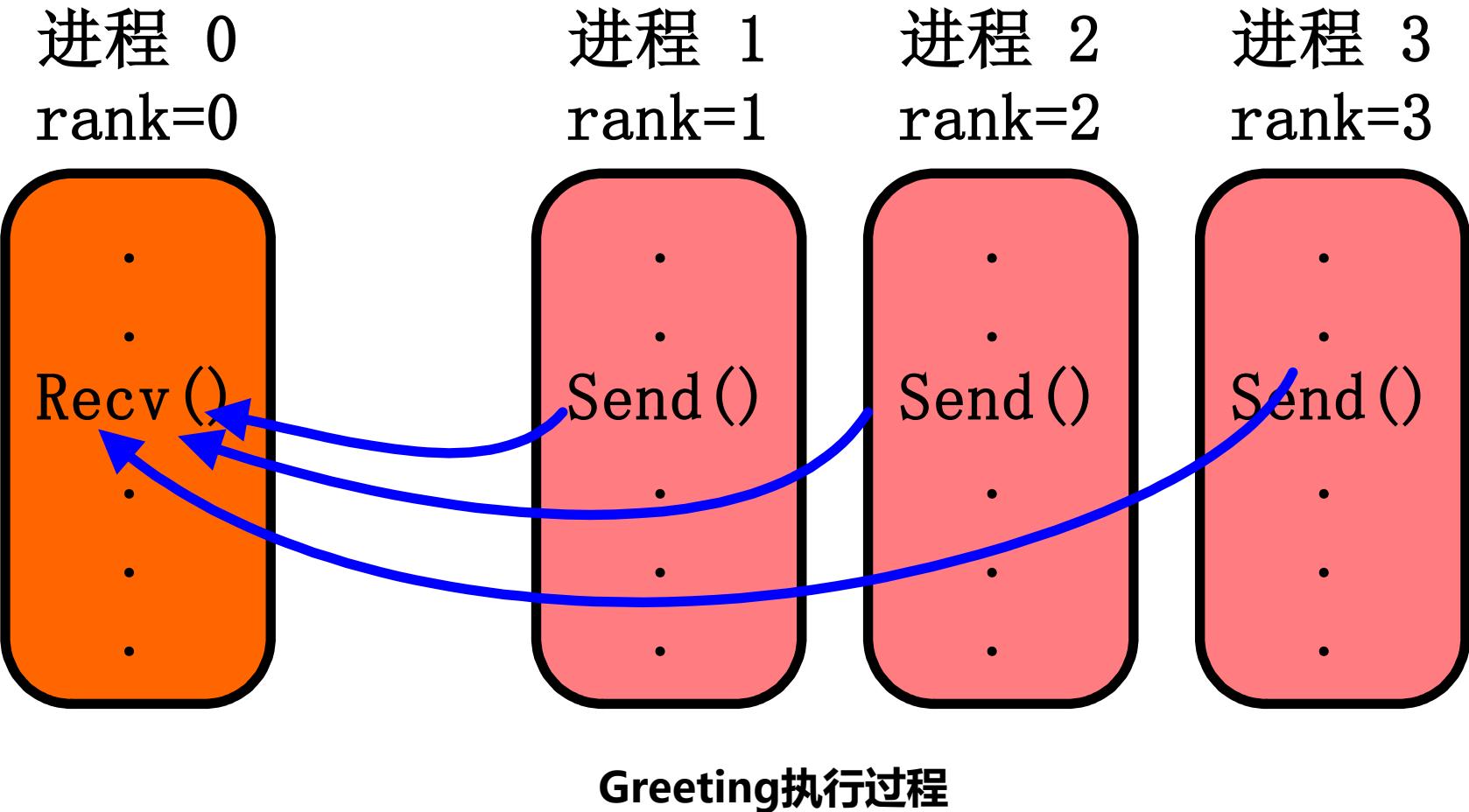
# 程序运行过程



# MPI编程的一些惯例

- MPI 的所有常量、变量与函数均以MPI\_ 开头
- 在C 程序中，所有常数的定义除下划线外一律由大写字母组成，在函数和数据类型定义中，接MPI\_ 之后的第一个字母大写，其余全部为小写字母，即MPI\_Xxxx\_xxx 形式
- 除MPI\_Wtime 和MPI\_Wtick 外，所有C函数调用之后都将返回一个错误信息码
- 由于C语言的函数调用机制是值传递，所以MPI的所有C函数中的输出参数用的都是指针
- MPI 是按进程组(Process Group) 方式工作：
  - 所有 MPI 进程在开始时均是在通信器MPI\_COMM\_WORLD 所对应的进程组中工作，之后用户可以根据自己的需要，建立其它的进程组
- 所有 MPI 的通信一定要在通信器中进行

# MPI并行通信程序



# Greetings程序

```
1 #include<stdio.h>
2 #include<string.h>
3 #include<mpi.h>
4
5 const int MAX_STRING=100;
6
7 int main(){
8     char greeting[MAX_STRING];
9     int comm_sz;
10    int my_rank;
11
12    MPI_Init(NULL,NULL);
13    MPI_Comm_size(MPI_COMM_WORLD,&comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
15
16    if(my_rank!=0){
17        sprintf(greeting,"Greetings from process %d of %d!",my rank,comm sz);
18        MPI_Send(greeting,strlen(greeting)+1,MPI_CHAR,0,0,MPI_COMM_WORLD);
19    }else{
20        for(int q = 1;q < comm sz;q++){
21            MPI_Recv(greeting,MAX_STRING,MPI_CHAR,q,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
22            printf("%s\n",greeting);
23        }
24    }
25    MPI_Finalize();
26    return 0;
27 }
```

# 消息发送和接收

- MPI 中发送和接收消息的基本函数分别是**MPI\_Send**和**MPI\_Recv**, 也称为**点对点通信**

```
int MPI_Send(void *buf, int count, MPI_Datatype  
            datatype, int dest, int tag, MPI_Comm comm)
```

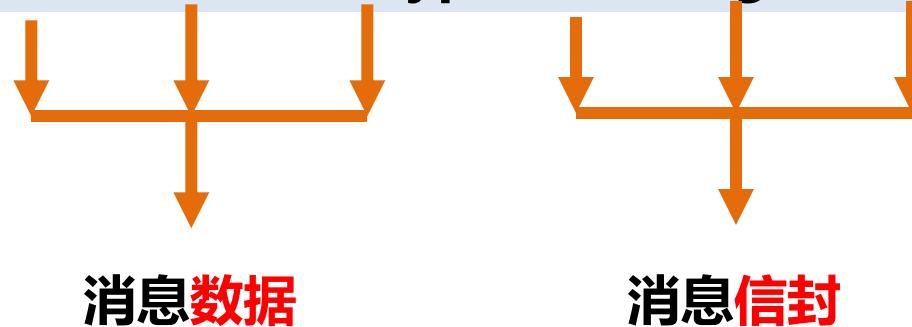
```
int MPI_Recv(void *buf, int count, MPI_Datatype  
            datatype, int source, int tag, MPI_Comm comm,  
            MPI_Status *status)
```

# 消息发送和接收

## MPI 消息 (message)

- MPI消息包括信封和数据两部分
- 信封: <源/目, 标识, 通信域>
- 数据: <起始地址, 数据个数, 数据类型>

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

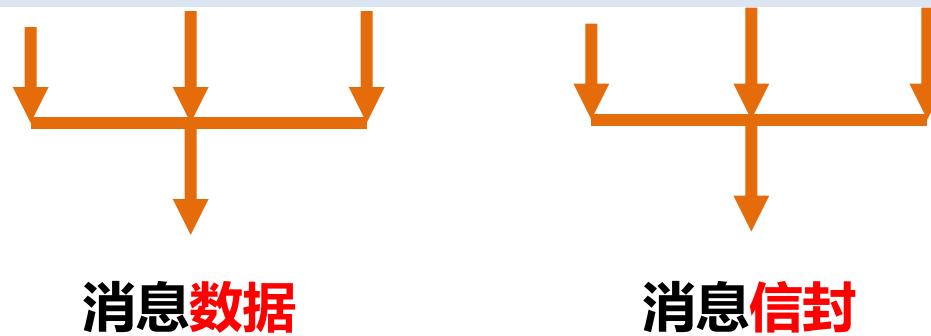


# 消息发送和接收

## MPI 消息 (message)

- MPI消息包括信封和数据两部分
- 信封: <源/目, 标识, 通信域>
- 数据: <起始地址, 数据个数, 数据类型>

```
int MPI_Recv(buf, count, datatype, source, tag, comm, status)
```



# MPI消息数据

**buf, count, datatype**

**buf:** **发送(接收)缓冲区的起始地址**

**发送哪些连续数据? 从内存的哪个位置开始发送?  
接收到的数据存放在内存的什么位置?**



**count:** **发送 (接收) 数据的个数**

**发送 (接收) 多少个数?**

**datatype:** **发送 (接收) 的数据类型**

**发送 (接收) 什么样的数据?**

# MPI消息信封

**发送:** dest, tag, comm

**接收:** source, tag, comm

**dest:** 目标进程 (将消息发送给哪个进程? )

**source:** 源进程 (从哪个进程接收消息? )

**tag:** 消息标签 (当发送者发送两个相同类型的数据给同一个接收者时, 如果没有消息标识, 接收者将如何区别这两个消息)

# MPI数据类型

MPI 除了提供函数外，还定义了一组常量和数据类型

- MPI 常量命名规则：全部大写
- MPI 变量数据类型命名规则：
  - 以MPI\_ 开头，后面跟C语言原始数据类型名称
- MPI 数据类型包含原始数据类型和自定义数据类型

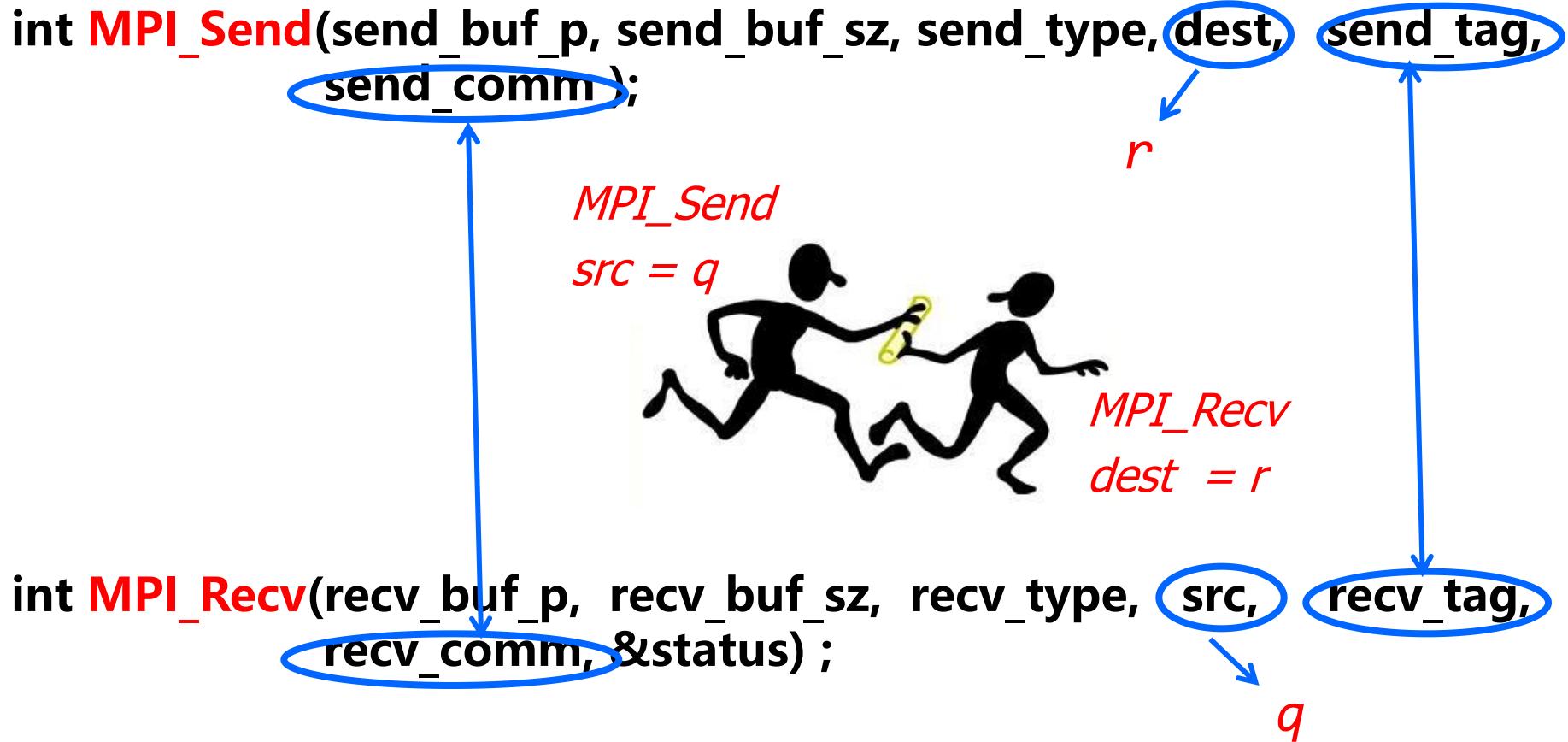
† 注意：MPI 的数据类型只用于消息传递！

# MPI数据类型

MPI datatype	C datatype
MPI_CHAR	<code>signed char</code>
MPI_SHORT	<code>signed short int</code>
MPI_INT	<code>signed int</code>
MPI_LONG	<code>signed long int</code>
MPI_UNSIGNED_CHAR	<code>unsigned char</code>
MPI_UNSIGNED_SHORT	<code>unsigned short int</code>
MPI_UNSIGNED_INT	<code>unsigned int</code>
MPI_UNSIGNED_LONG	<code>unsigned long int</code>
MPI_FLOAT	<code>float</code>
MPI_DOUBLE	<code>double</code>
MPI_LONG_DOUBLE	<code>long double</code>
MPI_BYTE	无对应类型
MPI_PACKED	无对应类型

MPI 数据类型主要  
用于数据的传递

# 消息匹配



# MPI\_Send点对点通信

```
int MPI_Send(void *buf, int count,  
                MPI_Datatype datatype,  
                int dest, int tag, MPI_Comm comm)
```

参数	含义
<b>buf</b>	所发送消息的内存首地址
<b>count</b>	传递的数据的长度
<b>datatype</b>	变量类型，MPI预定义的变量类型
<b>dest</b>	接收消息的进程的标识号
<b>tag</b>	消息标签, 用于识别消息
<b>comm</b>	通信器, 数据在其中进程间传输

# MPI\_Send点对点通信

**MPI\_Send(buf, count, datatype, dest, tag, comm)**

- 阻塞型消息发送接口
- MPI\_SEND 将缓冲区中count个datatype类型的数据发给进程号为 dest 的目的进程。这里 count 是元素个数，即指定数据类型的个数，不是字节数，数据的起始地址为 buf。本次发送的消息标签是 tag，使用标签的目的是把本次发送的消息和本进程向同一目的进程发送的其它消息区别开来。其中 dest 的取值范围为 0~np-1 ( np 表示通信器 comm 中的进程数) 或 MPI\_PROC\_NULL, tag 的取值为 0~ MPI\_TAG\_UB
- 该函数可以发送各种类型的数据，如整型、实型、字符等
- 点对点通信是 MPI 通信机制的基础

# MPI\_Recv点对点通信

```
int MPI_Recv(void *buf, int count,  
            MPI_Datatype datatype,  
            int source, int tag,  
            MPI_Comm comm, MPI_Status *status)
```

参数	含义
buf	接收消息数据的首地址
count	接收数据的最大个数
datatype	接收数据的数据类型
source	发送消息的进程的标识号
tag	消息标签, 用于识别消息
comm	通信器, 数据在其中进程间传输
status	返回状态

# MPI\_Recv点对点通信

**MPI\_Recv(buf, count, datatype, source, tag, comm, status)**

- 阻塞型消息接收接口
- 从指定的进程source接收不超过count个datatype类型的数据，并把它放到缓冲区中，起始位置为buf，本次消息的标识为tag。这里source的取值范围为0~np-1，或MPI\_ANY\_SOURCE，或MPI\_PROC\_NULL，tag的取值为0~MPI\_TAG\_UB或MPI\_ANY\_TAG
- 接收消息时返回的状态 STATUS，在 C 语言中是用结构定义的，在 FORTRAN 中是用数组定义的，其中包括 MPI\_SOURCE，MPI\_TAG 和 MPI\_ERROR。此外 STATUS 还包含接收消息元素的个数，但它不是显式给出的，需要用到后面给出的函数 MPI\_Get\_Count

# 示例

.....

```
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

if(myid==0){
    MPI_Recv( buf1,10,MPI_INT,1,1,MPI_COMM_WORLD,&status);
    MPI_Recv (buf2,10,MPI_INT,2,1,MPI_COMM_WORLD,&status);
}

if(myid==1)
    MPI_Send (buf1,10 ,MPI_INT,0,1,MPI_COMM_WORLD );
if (myid ==2)
    MPI_Send(buf2,10 ,MPI_INT,0,1,MPI_COMM_WORLD );
```

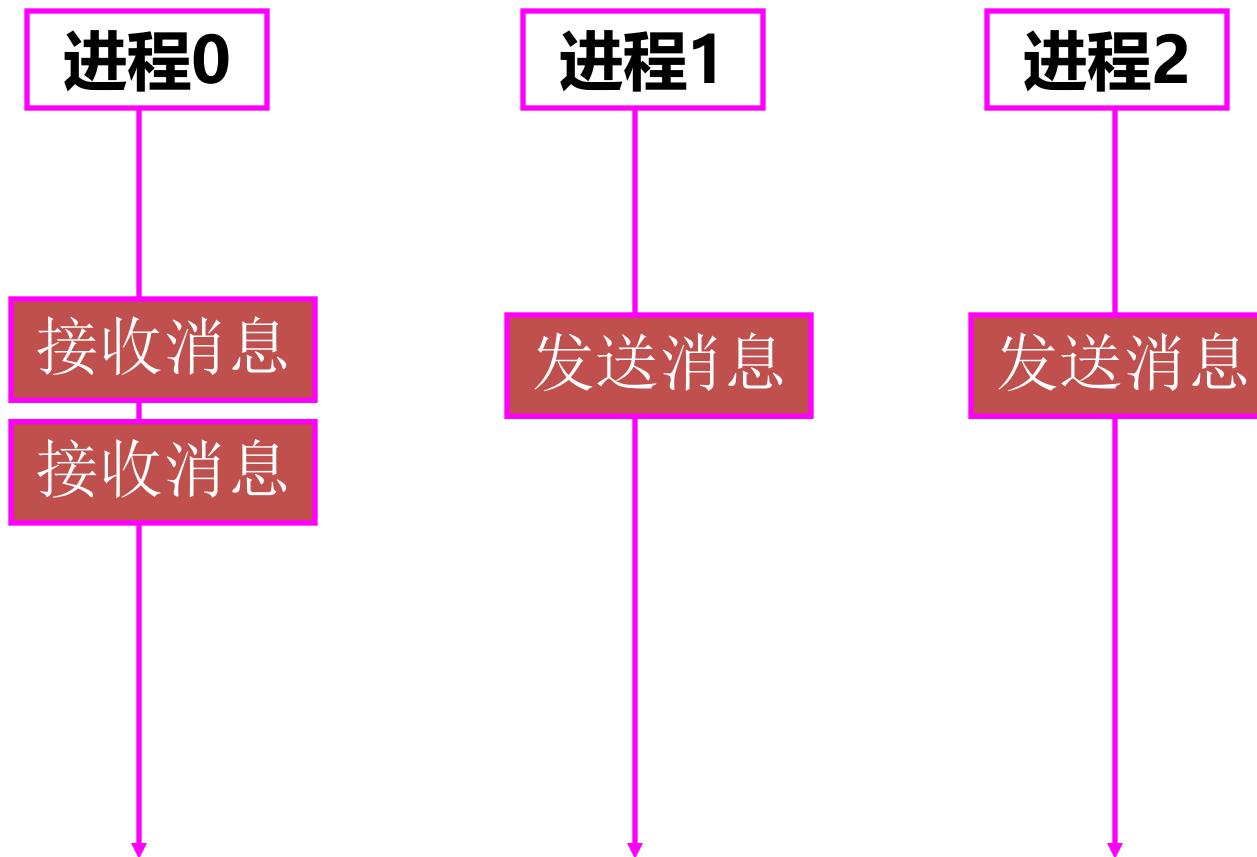
消息来源于进程1

消息来源于进程2

消息发送给进程0

消息发送给进程0

# 程序执行过程



# 任意源和任意标识

## **MPI\_ANY\_SOURCE, MPI\_ANY\_TAG**

- **MPI\_ANY\_SOURCE:**

**任何进程发送的消息都可以接收，但其它要求必须满足**

- **MPI\_ANY\_TAG:**

**任何标签的消息都可以接收，但其它要求必须满足**

- **两者可同时使用，也可单独使用**

- **不能给通信域comm指定任意值**

- **发送操作与接收操作不对称**

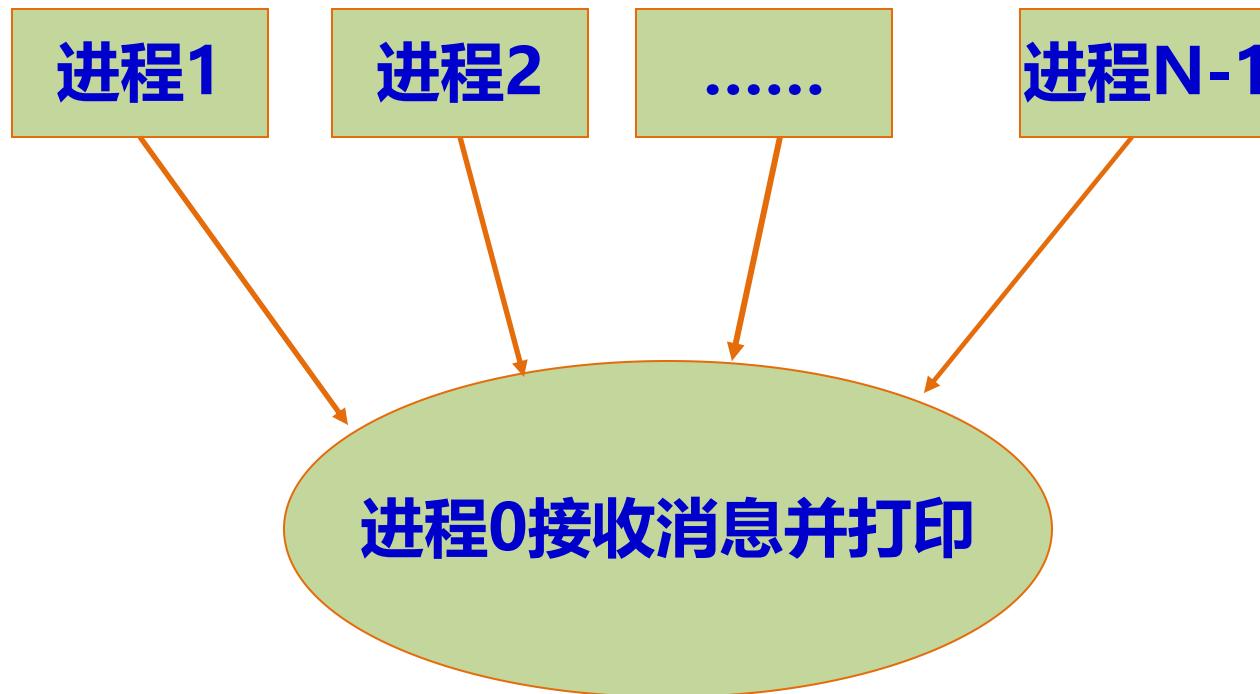
# 示例

.....

```
MPI_Comm_rank(MPI_COMM_WORLD,&myid);  
if(myid == 0){ 消息来源：可能是进程1，也可能是进程2  
    MPI_Recv(buf1, 10, MPI_INT, MPI_ANY_SOURCE, 1,  
             MPI_COMM_WORLD, &status);  
  
    MPI_Recv (buf2,10,MPI_INT, MPI_ANY_SOURCE, 1,  
              MPI_COMM_WORLD, &status);  
}  
  
if(myid == 1) 消息发送给进程0  
    MPI_Send (buf1,10 ,MPI_INT,0,1,MPI_COMM_WORLD );  
if (myid == 2) 消息发送给进程0  
    MPI_Send(buf2,10 ,MPI_INT,0,1,MPI_COMM_WORLD )
```

.....

# 进程0接收任意源和任意标签消息



# 分析greetings

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
    int numprocs;          /*进程数,该变量为各处理器中的同名变量,存储是分布的*/
    int myid;              /*进程ID,存储也是分布的*/ */
    MPI_Status status;     /*消息接收状态变量,存储也是分布的*/ */
    char message[100];     /*消息buffer,存储也是分布的*/ */
    /*初始化MPI*/
    MPI_Init( &argc, &argv );
    /*该函数被各进程各调用一次,得到自己的进程rank值*/
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    /*该函数被各进程各调用一次,得到进程数*/
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
```

# 分析greetings

```
if (myid != 0) {  
    /*建立消息*/  
    sprintf(message, "Greetings from process %d!", myid);  
    /* 发送长度取strlen(message)+1,使\0也一同发送出去*/  
    MPI_Send(message, strlen(message)+1, MPI_CHAR,  
             0, 99, MPI_COMM_WORLD);  
}  
else{ /*myrank == 0*/  
    for (source = 1; source < numprocs; source++) {  
        MPI_Recv(message, 100, MPI_CHAR, source, 99,  
                 MPI_COMM_WORLD, &status);  
        printf("%s\n", message);  
    }  
}  
/*关闭MPI,标志并行代码段的结束*/  
MPI_Finalize();  
} /* end main */
```

# Greetings执行过程

假设进程数为3

(进程0)  
(rank=0)

```
•  
•  
Recv();  
•  
•  
Recv();  
•
```

(进程1)  
(rank=1)

```
•  
•  
Send();  
•  
•
```

(进程2)  
(rank=2)

```
•  
•  
Send()  
•  
•
```

?



问题:进程1和2谁先开始发送消息?谁先完成发送?

# 运行greetings

```
[haomeng@ubuntu ~]$ mpicc -o greeting greeting.c  
[haomeng@ubuntu ~]$ mpirun -np 4 ./greeting
```

Greetings from process 1 of 4!

Greetings from process 2 of 4!

Greetings from process 3 of 4!

```
[haomeng@ubuntu ~]$
```

- 计算机打印字符
- 我们输入的命令

# 利用MPI\_ANY\_SOURCE

```
1 #include<stdio.h>
2 #include<string.h>
3 #include<mpi.h>
4
5 const int MAX_STRING=100;
6
7 int main(){
8     char greeting[MAX_STRING];
9     int comm_sz; //线程的个数
10    int my_rank;
11
12    MPI_Init(NULL,NULL);
13    MPI_Comm_size(MPI_COMM_WORLD,&comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
15
16    if(my_rank!=0){
17        sprintf(greeting,"Greetings from process %d of %d!",my_rank,comm_sz);
18        MPI_Send(greeting,strlen(greeting)+1,MPI_CHAR,0,0,MPI_COMM_WORLD);
19    }else{
20        for(int q=1;q<comm_sz;q++){
21            MPI_Recv(greeting,MAX_STRING,MPI_CHAR,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
22            printf("%s\n",greeting);
23        }
24    }
25
26    MPI_Finalize();
27    return 0;
28 }
```

# 运行greetings

```
[haomeng@ubuntu ~]$ mpicc -o greeting greeting.c
```

```
[haomeng@ubuntu ~]$ mpirun -np 4 ./greeting
```

Greetings from process 3 of 4!

Greetings from process 1 of 4!

Greetings from process 2 of 4!

```
[haomeng@ubuntu ~]$
```

- 计算机打印字符
- 我们输入的命令

# 最基本的MPI程序

- MPI函数的总数虽然庞大，但根据实际编写MPI的经验，常用的MPI调用的个数确实有限
- 下面是6个最基本也是最常用的MPI函数
  - MPI\_Init(...)
  - MPI\_Comm\_size(...)
  - MPI\_Comm\_rank(...)
  - MPI\_Send(...)
  - MPI\_Recv(...)
  - MPI\_Finalize()
- 下面更多的了解MPI

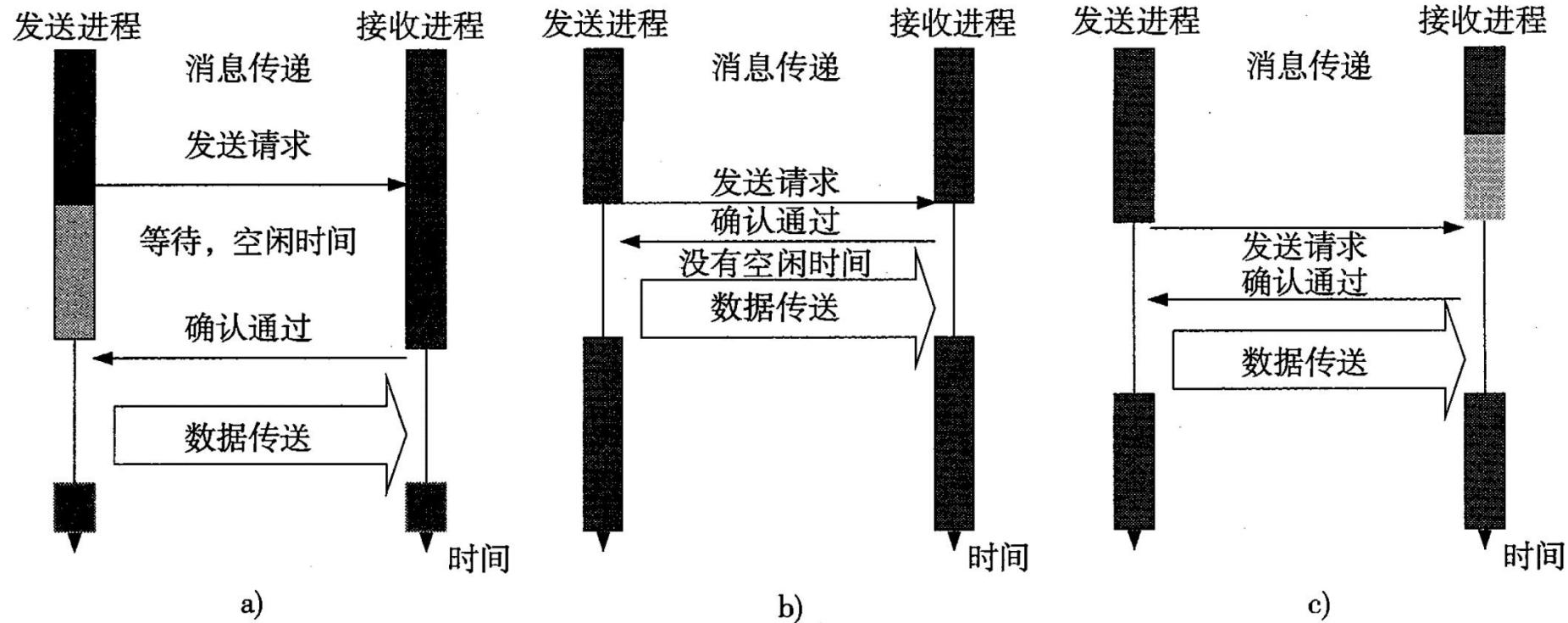
# 阻塞与非阻塞通信

- MPI的点对点通信(Point-to-Point Communication)同时提供了阻塞和非阻塞两种通信机制，也支持多种通信模式
- 不同通信模式和不同通信机制的结合，便产生了非常丰富的点对点通信函数

# 阻塞与非阻塞通信

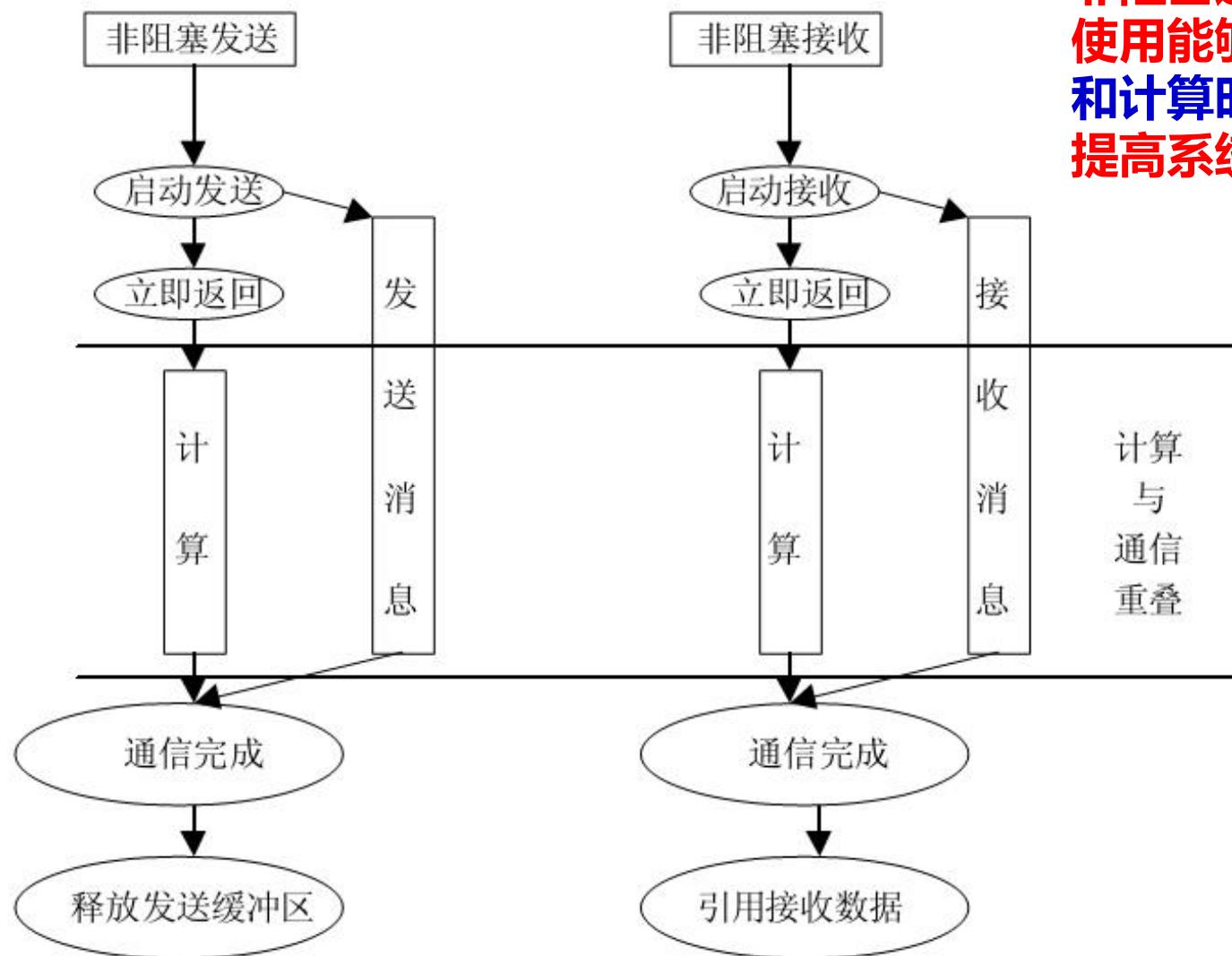
- 阻塞和非阻塞通信的主要区别在于返回后的资源可用性
- 阻塞通信返回的条件：
  - 通信操作已经完成。即消息已经发送或接收
  - 调用的缓冲区可用。若是发送操作，则该缓冲区已经可以被其它的操作更新；若是接收操作，该缓冲区的数据已经完整，可以被正确引用
- 非阻塞通信返回后并不意味着通信操作的完成，MPI还提供了对非阻塞通信完成的检测，主要的有两种：MPI\_Wait函数和MPI\_Test函数

# 阻塞通讯：握手模式



# 非阻塞标准发送和接收

非阻塞通信机制的  
使用能够重叠通信  
和计算时间，从而  
提高系统的性能



# MPI\_Isend点对点通信

```
int MPI_Isend(void *buf, int count,  
              MPI_Datatype datatype,  
              int dest, int tag, MPI_Comm comm,  
              MPI_Request *request)
```

参数	含义
buf	所发送消息的内存首地址
count	传递的数据的长度
datatype	变量类型, MPI预定义的变量类型
dest	接收消息的进程的标识号
tag	消息标签, 用于识别消息
comm	通信器, 数据在其中进程间传输
request	非阻塞通信完成对象

# MPI\_Irecv点对点通信

```
int MPI_Irecv(void *buf, int count,  
              MPI_Datatype datatype,  
              int source, int tag, MPI_Comm comm,  
              MPI_Request *request)
```

参数	含义
buf	接收消息数据的首地址
count	接收数据的最大个数
datatype	接收数据的数据类型
source	发送消息的进程的标识号
tag	消息标签, 用于识别消息
comm	通信器, 数据在其中进程间传输
request	非阻塞通信完成对象

# 判断非阻塞通信完成

- **发送的完成**: 代表发送缓冲区中的数据已送出，发送缓冲区可以重用。它并不代表数据已被接收方接收。数据有可能被缓冲
- **接收的完成**: 代表数据已经写入接收缓冲区。接收者可访问接收缓冲区

阻塞型函数，必须等待指定的通信请求完成后才能返回

```
int MPI_Wait(MPI_Request* request,  
             MPI_Status * status)
```

```
int MPI_Test(MPI_Request * request, int * flag,  
             MPI_Status * status)
```

检测指定的通信请求，不论通信是否完成都立刻返回。若通信已经完成则返回flag=true，如果通信还未完成则返回 flag=false

# MPI\_Wait应用示例

```
MPI_Request request;
MPI_Status status;
int x,y;
if(rank == 0){
    x = func1();
    MPI_Isend(&x, 1, MPI_INT, 1, 99, comm, &request)
    ...
    MPI_Wait(&request,&status);
    x = func2();
    ...
}
else{
    MPI_Irecv(&y, 1, MPI_INT, 0, 99, comm, &request)
    ...
    MPI_Wait(&request, &status);
    func3(y);
}
```

# MPI\_Test应用示例

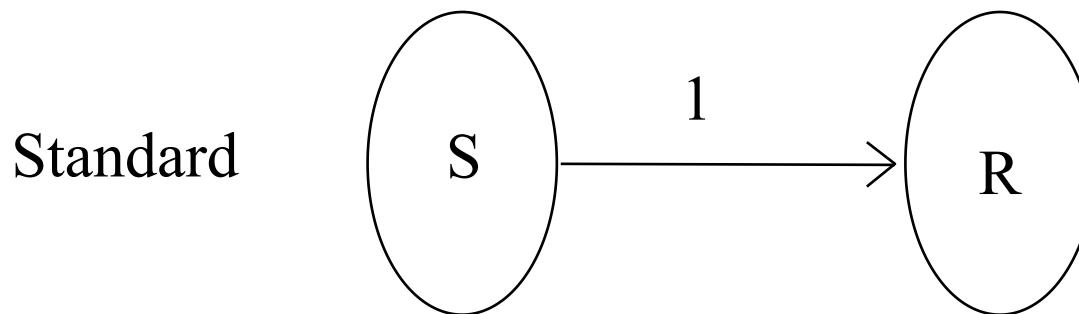
```
MPI_Request request;
MPI_Status status;
int x,y,flag;
if(rank == 0){
    x = func1();
    MPI_Isend(&x, 1, MPI_INT, 1, 99, comm, &request);
    ... //computing
    while(!flag)
        MPI_Test(&request, &flag, &status);
    x = func2();
}
else{
    MPI_Irecv(&y, 1, MPI_INT, 0, 99, comm, &request);
    ... //computing
    while(!flag)
        MPI_Test(&request, &flag, &status);
    func3(y);
}
```

# 通信模式

- 通信模式指的是缓冲管理，以及发送方和接收方之间的同步方式
- 共有下面四种通信模式：
  - 标准(standard)通信模式
  - 缓冲(buffered)通信模式
  - 同步(synchronous)通信模式
  - 就绪(ready)通信模式

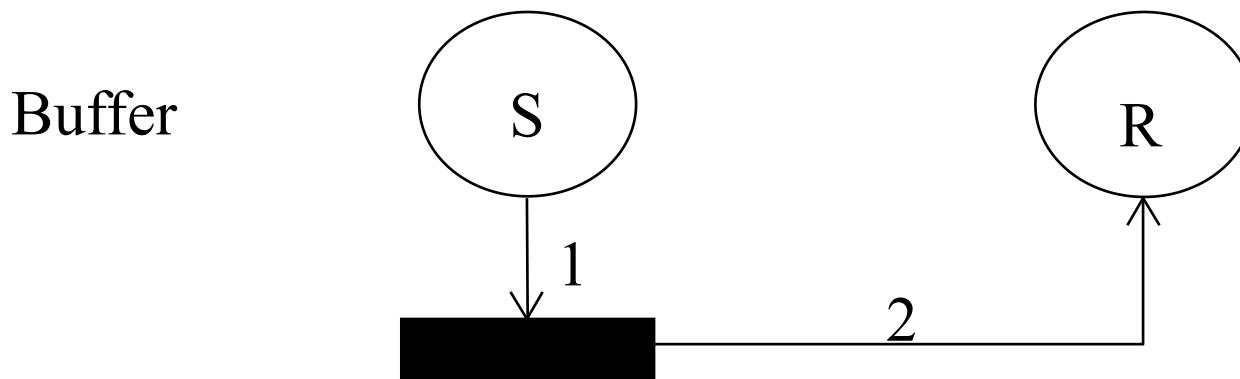
# 标准通信模式

- 是否对发送的数据进行缓冲由MPI的实现来决定，而不是由用户程序来控制
- 发送可以是同步的或缓冲的，取决于实现
- 对应MPI\_Send



# 缓冲通信模式

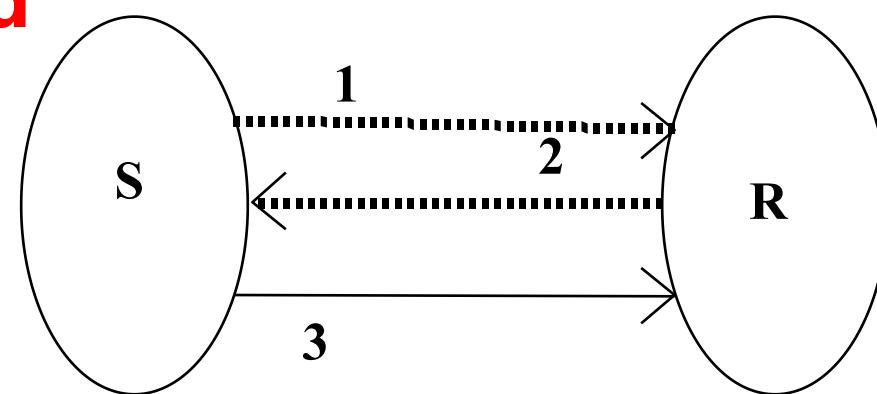
- 缓冲通信模式的发送不管接收操作是否已经启动都可以执行
- 需要用户程序事先申请一块足够大的缓冲区，通过**MPI\_Buffer\_attach**实现，通过**MPI\_Buffer\_detach**来回收申请的缓冲区
- 对应**MPI\_Bsend**



# 同步通信模式

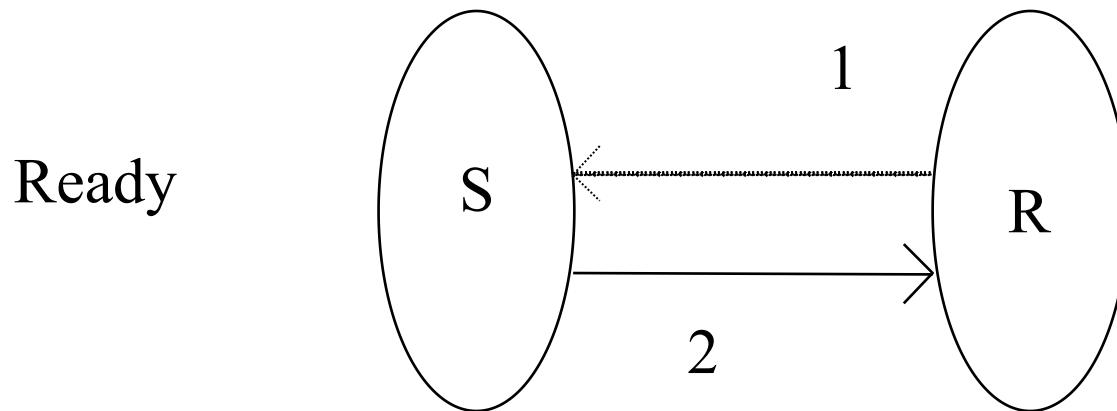
- 只有相应的接收过程已经启动，发送过程才正确返回
- 同步发送返回后，表示发送缓冲区中的数据已经全部被系统缓冲区缓存，并且已经开始发送
- 当同步发送返回后，发送缓冲区可以被释放或者重新使用
- 对应MPI\_Ssend

Synchronous



# 就绪通信模式

- 发送操作只有在接收进程相应的接收操作已经开始才进行发送
- 当发送操作启动而相应的接收还没有启动，发送操作将出错。就绪通信模式的特殊之处就是接收操作必须先于发送操作启动
- 对应MPI\_Rsend



# 通信模式

- MPI的发送操作支持四种通信模式，它们与阻塞属性一起产生了MPI中的8种发送操作
- MPI的接收操作有两种：阻塞接收和非阻塞接收

MPI 原语	阻塞	非阻塞
Standard Send	<b>MPI_Send</b>	<b>MPI_Irecv</b>
Synchronous Send	<b>MPI_Ssend</b>	<b>MPI_Issend</b>
Buffered Send	<b>MPI_Bsend</b>	<b>MPI_Ibsend</b>
Ready Send	<b>MPI_Rsend</b>	<b>MPI_Irsend</b>
Receive	<b>MPI_Recv</b>	<b>MPI_Irecv</b>
Completion Check	<b>MPI_Wait</b>	<b>MPI_Test</b>

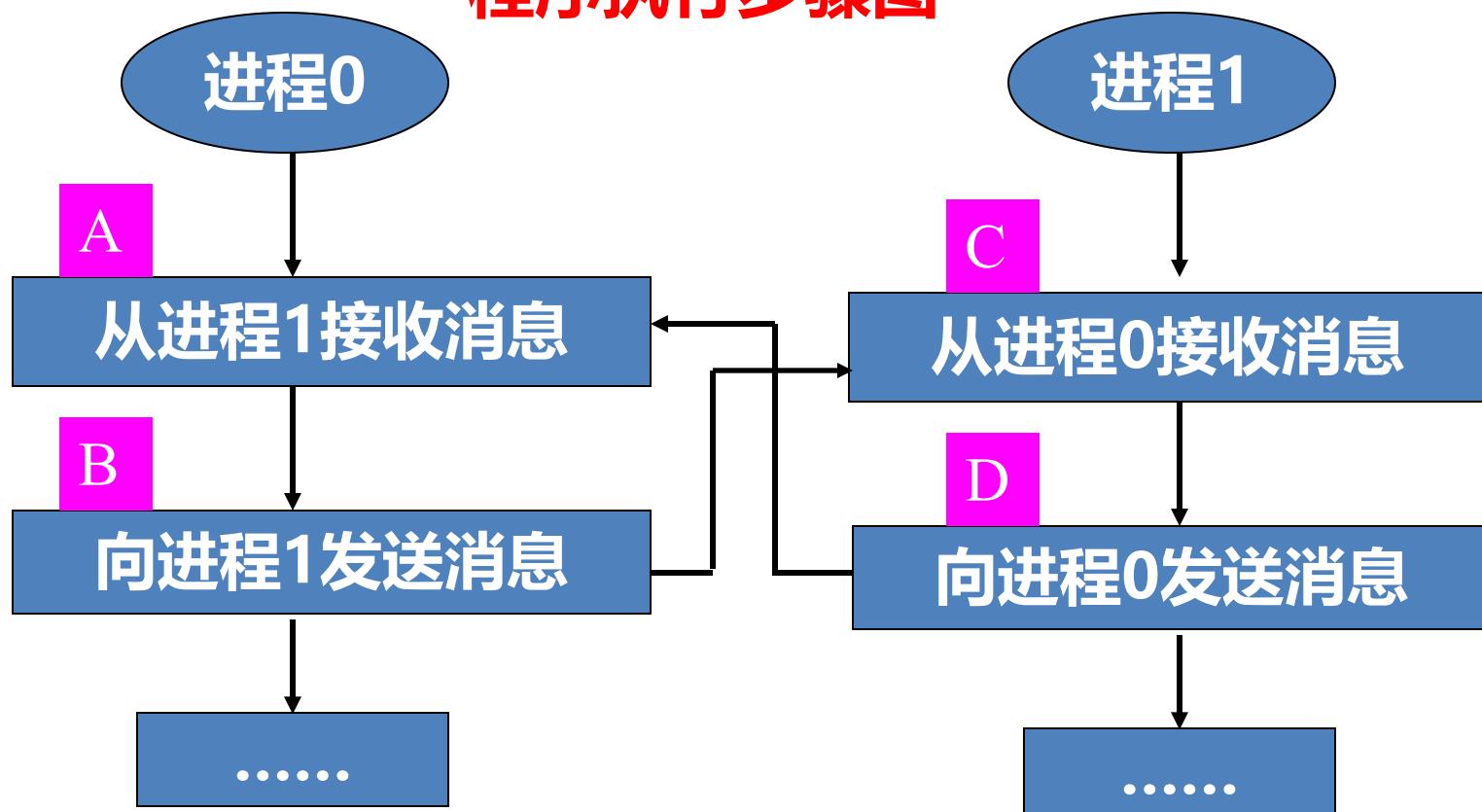
# 编写安全的MPI程序

假设程序启动时共产生两个进程

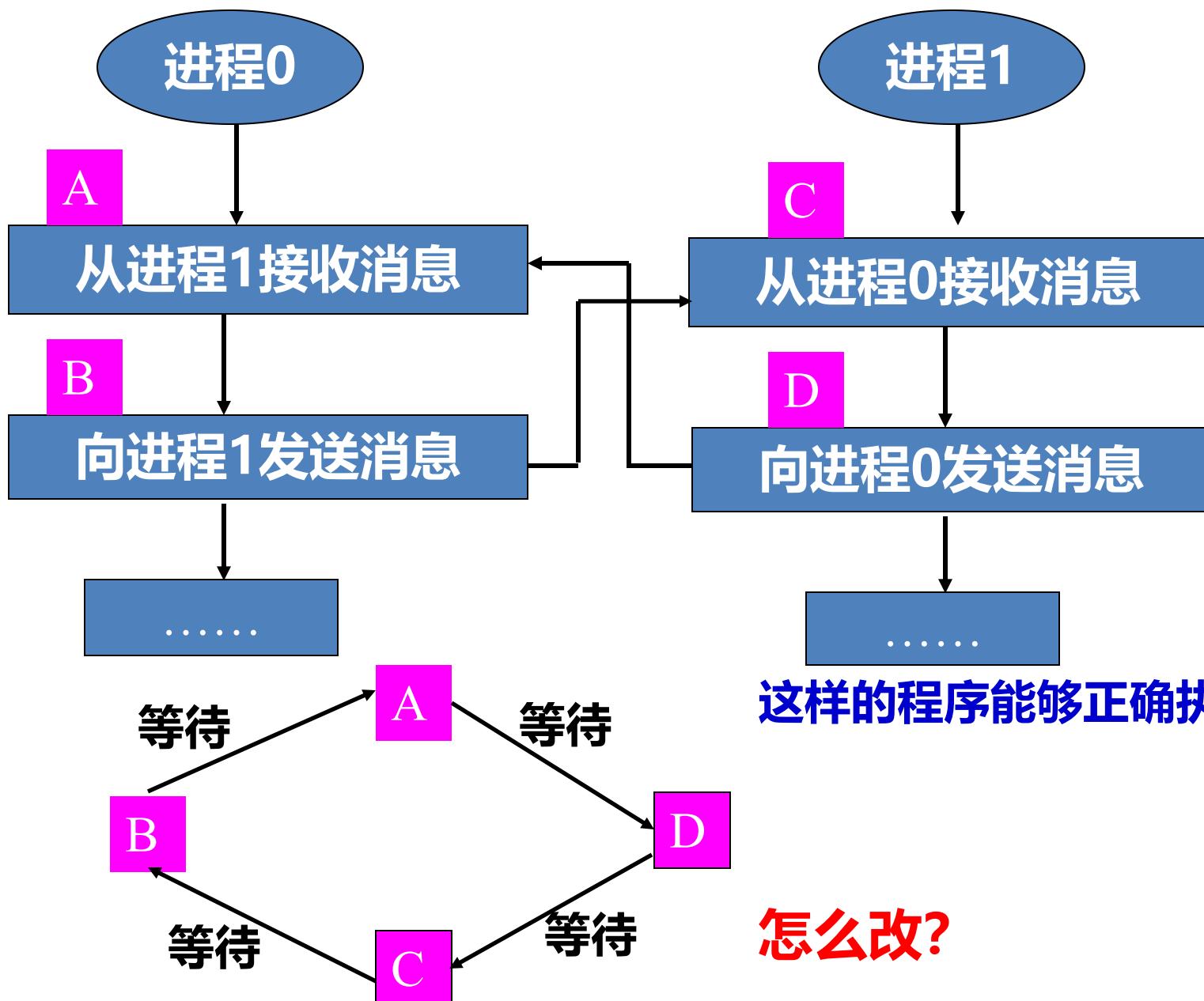
```
comm=MPI_COMM_WORLD;  
  
MPI_Comm_rank(MPI_COMM_WORLD,&rank);  
  
if(rank==0)  
{    MPI_Recv(x2 , 3 , MPI_INT,1,tag,comm,&status);  
    MPI_Send(x1 , 3 , MPI_INT,1,tag,comm);    }  
  
if(rank==1)  
{    MPI_Recv(x1, 3 ,MPI_INT,0,tag,comm,&status);  
    MPI_Send(x2, 3 ,MPI_INT,0,tag,comm);    }  
  
.....
```

# 编写安全的MPI程序

## 程序执行步骤图



程序的预期目标：1、两个进程分别从对方接收一个消息，  
2、向对方发送一个消息。



```
comm=MPI_COMM_WORLD;  
  
MPI_Comm_rank(MPI_COMM_WORLD,&rank);  
  
if(rank==0)  
{    MPI_Recv(x2 , 3 , MPI_INT,1,tag,comm,&status);  
    MPI_Send(x1 , 3 , MPI_INT,1,tag,comm);    }  
  
if(rank==1)  
{    MPI_Recv(x1, 3 ,MPI_INT,0,tag,comm,&status);  
    MPI_Send(x2, 3 ,MPI_INT,0,tag,comm);    }  
  
.....
```

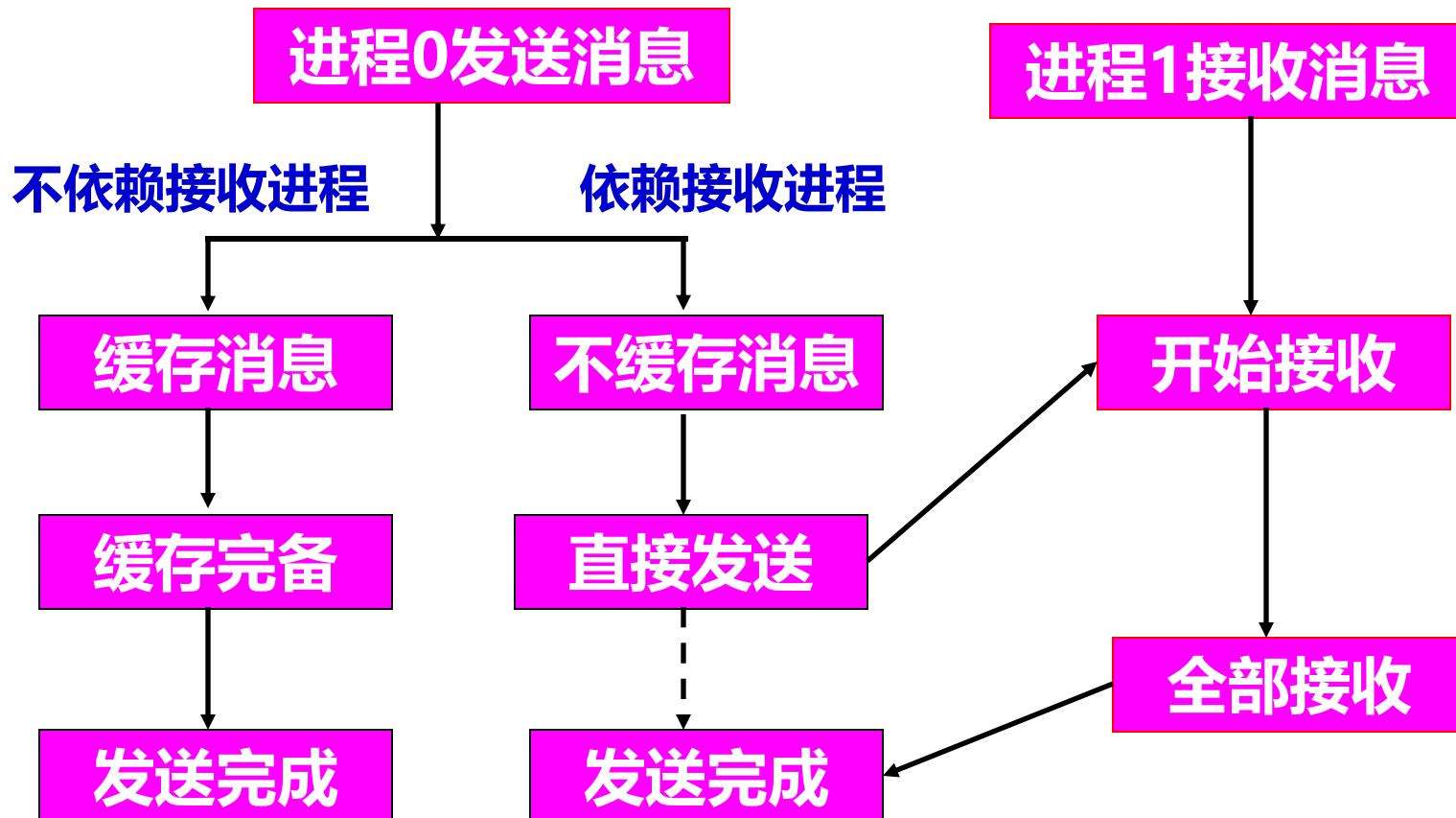
两个进程都是先接收，后发送。  
没有发送，何谈接收？ 不合逻辑！

## 改成：先发送，后接收

```
comm=MPI_COMM_WORLD  
  
MPI_Comm_rank(MPI_COMM_WORLD,&rank)  
  
if(rank==0)  
{   MPI_Send(sendbuf,3,MPI_INT,1,tag,comm);  
    MPI_Recv(recvbuf,3,MPI_INT,1,tag,comm, &status);  
}  
  
if(rank==1)  
{   MPI_Send(sendbuf,3,MPI_INT,0,tag,comm);  
    MPI_Recv(recvbuf,3,MPI_INT,0,tag,comm, &status);  
}
```

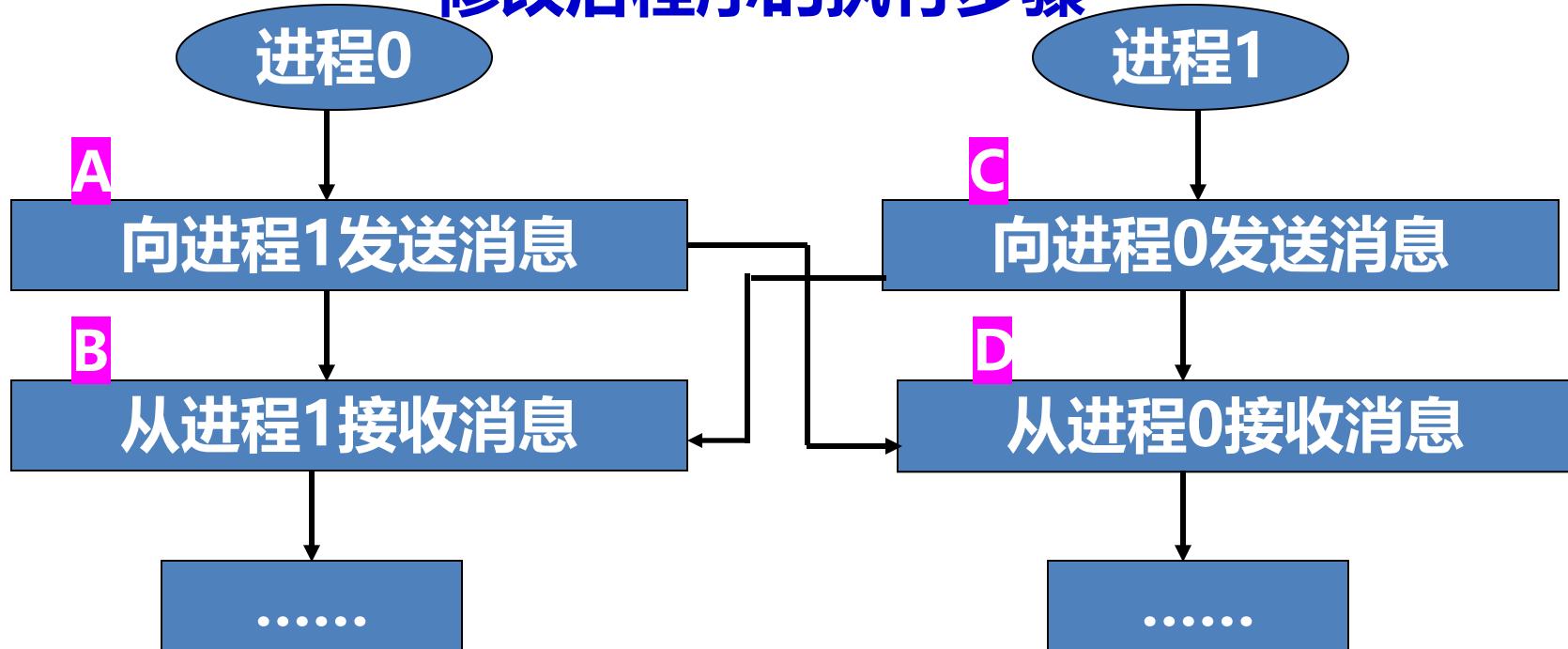
这个程序的性能如何？

欲分析该程序的性能，应先了解上述通信模式中发送与接收操作的执行过程



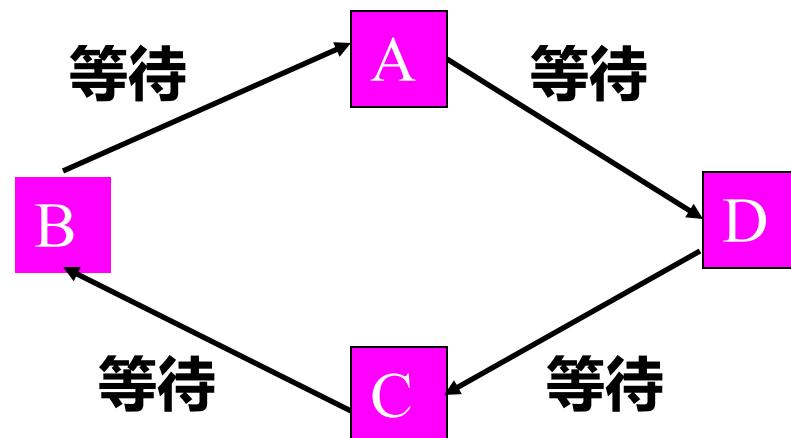
(1) 是否缓存数据由MPI决定；(2) 如果MPI决定缓存该数据，则发送操作可正确返回而不要求接收操作收到发送的数据；(3) 缓存数据要付出代价，缓冲区并非总是可以得到的。

## 修改后程序的执行步骤

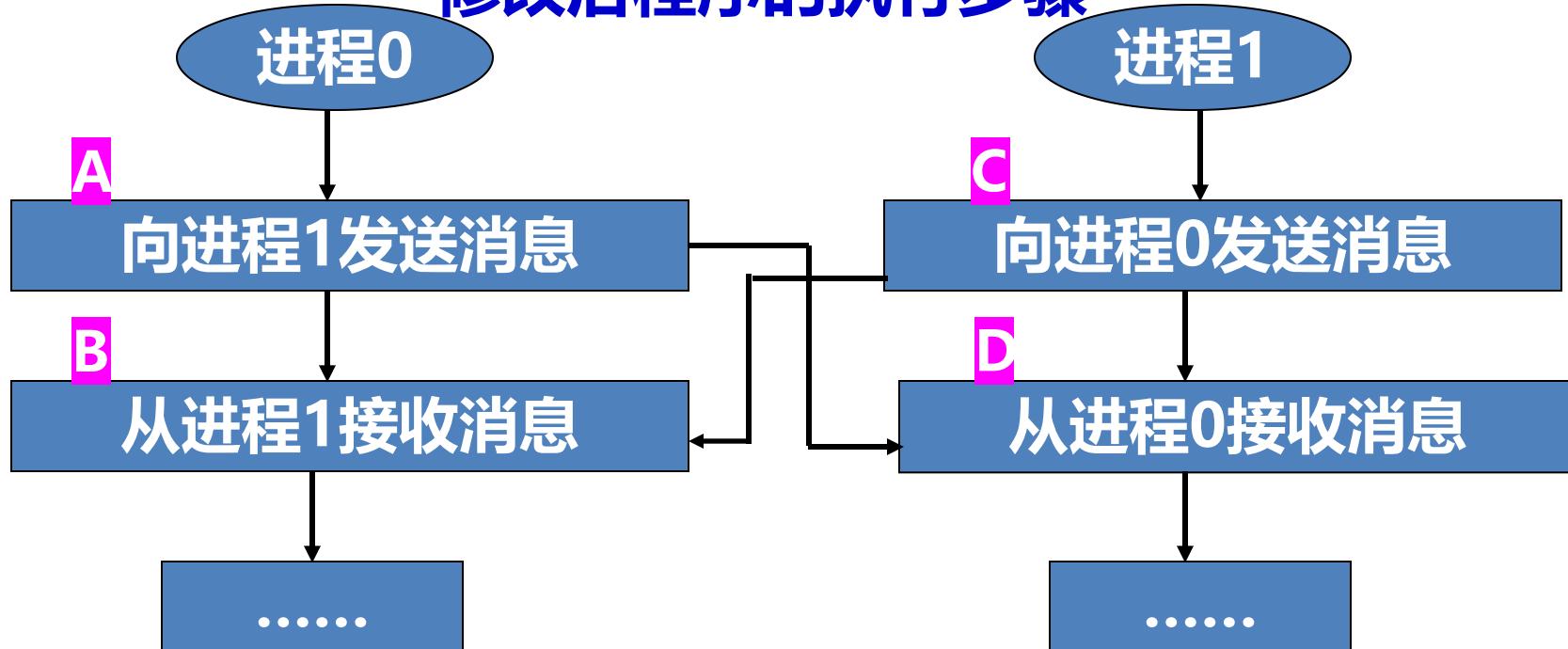


如果系统不缓存消息：

**消息死锁!**

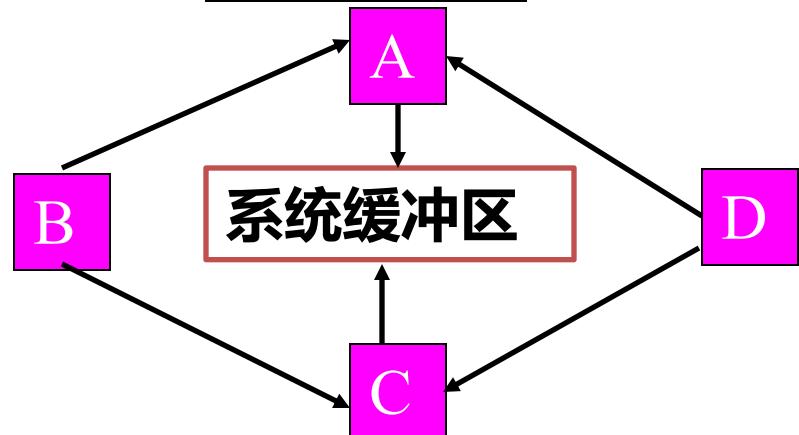


## 修改后程序的执行步骤



如果系统缓存消息：

A, C两个发送操作都需要系统缓冲区，如果系统缓冲不足，消息传递将无法完成。



**不安全的通信调用次序!**

## 再一次修改：发送与接收操作按次序进行匹配

```
comm=MPI_COMM_WORLD  
  
MPI_Comm_rank(MPI_COMM_WORLD,&rank)  
  
if(rank==0)  
{    MPI_Send(sendbuf,count,MPI_INT,1,tag,comm);  
    MPI_Recv(recvbuf,count,MPI_INT,1,tag,comm,&status);  
}  
  
if(rank==1)  
{    MPI_Recv(recvbuf,count,MPI_INT,0,tag,comm,&status);  
    MPI_Send(sendbuf,count,MPI_INT,0,tag,comm);  
}
```

进程0

安全通信!

进程1

A

向进程1发送消息

C

从进程0接收消息

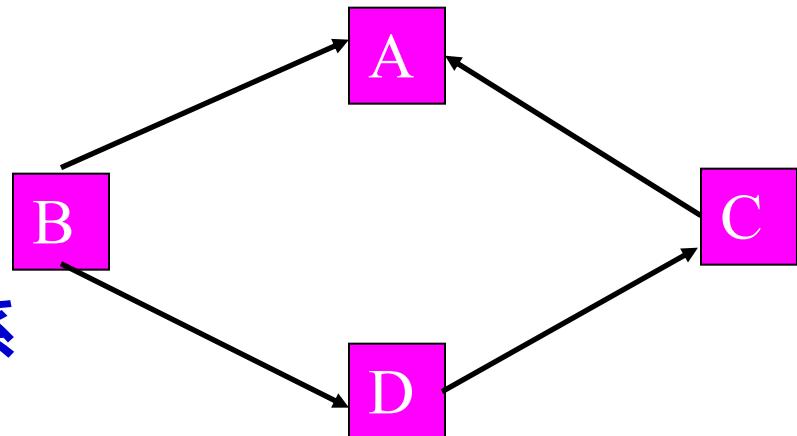
B

从进程1接收消息

D

向进程0发送消息

1. 只要C存在，则系统不提供缓冲区A也能够执行；
2. C能够执行；
3. A, C完成后，只要B存在，系统不提供缓冲区D也能够执行；
4. B能够执行。



# 编写安全的MPI程序

- 将发送与接收操作按照次序进行匹配
  - 一个进程的发送操作在前，接收操作在后；
  - 另一个进程的接收操作在前，发送操作在后；
- 若将两个进程的发送与接收操作次序互换，其消息传递过程仍是安全的

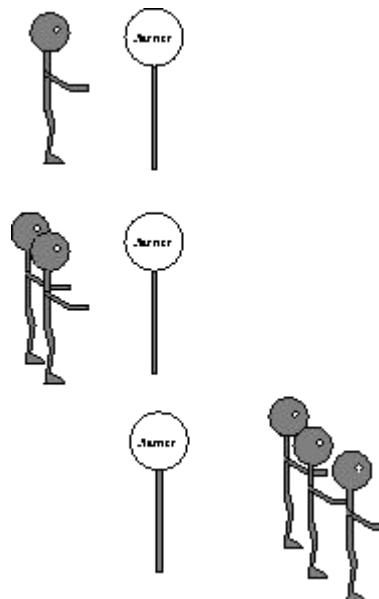
# 聚合通信

- MPI的聚合 (Collective) 通信，即向一组进程发送消息或从一组进程接收消息
  - 一对多通信：一个进程向其他所有的进程发送消息，这个负责发送消息的进程称为**root**进程
  - 多对一通信：一个进程负责从其他所有的进程接收消息，这个负责接收消息的进程也称为**root**进程
  - 多对多通信：每个进程都向其他所有的进程发送或接收消息
- 聚合通信包括：**路障(Barrier)**、**广播通信(Broadcast)**、**收集通信(Gather)**、**分散通信(Scatter)**、**归约通信(Reduction)**、**全到全广播通信(All-to-all)** . . .

# 聚合通信：路障

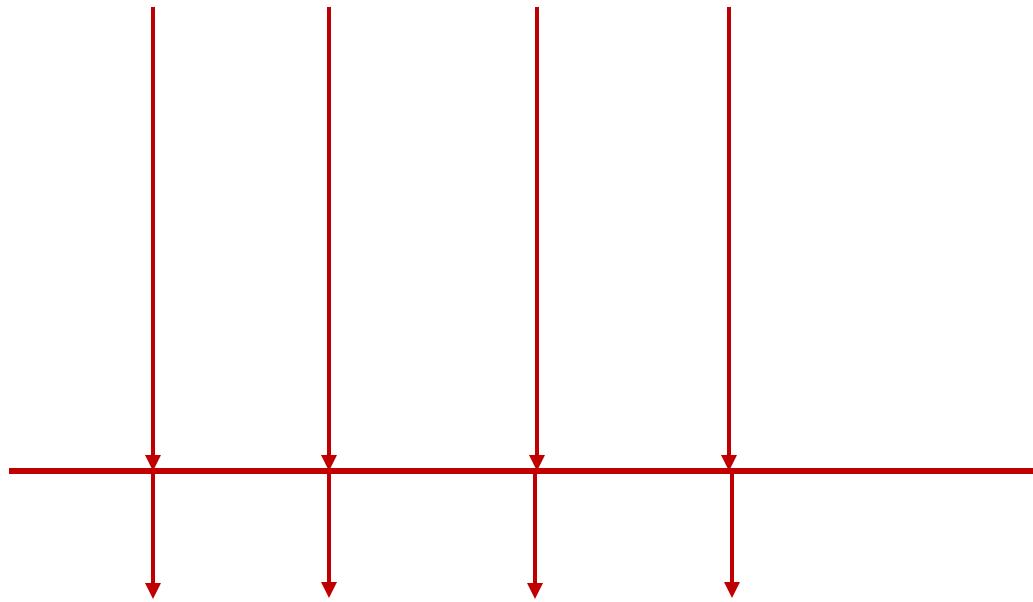
- **MPI\_Barrier** 阻塞通信器中所有进程，直到所有的进程组成员都调用了它。仅当进程组所有的成员都进入了这个调用后，各个进程中这个调用才可以返回

```
int MPI_Barrier ( MPI_Comm comm )
```



路障操作会同步多个进程

# 聚合通信：路障



快的进程必须等待慢的进程，直到所有进程都执行到该语句后才可以向下进行

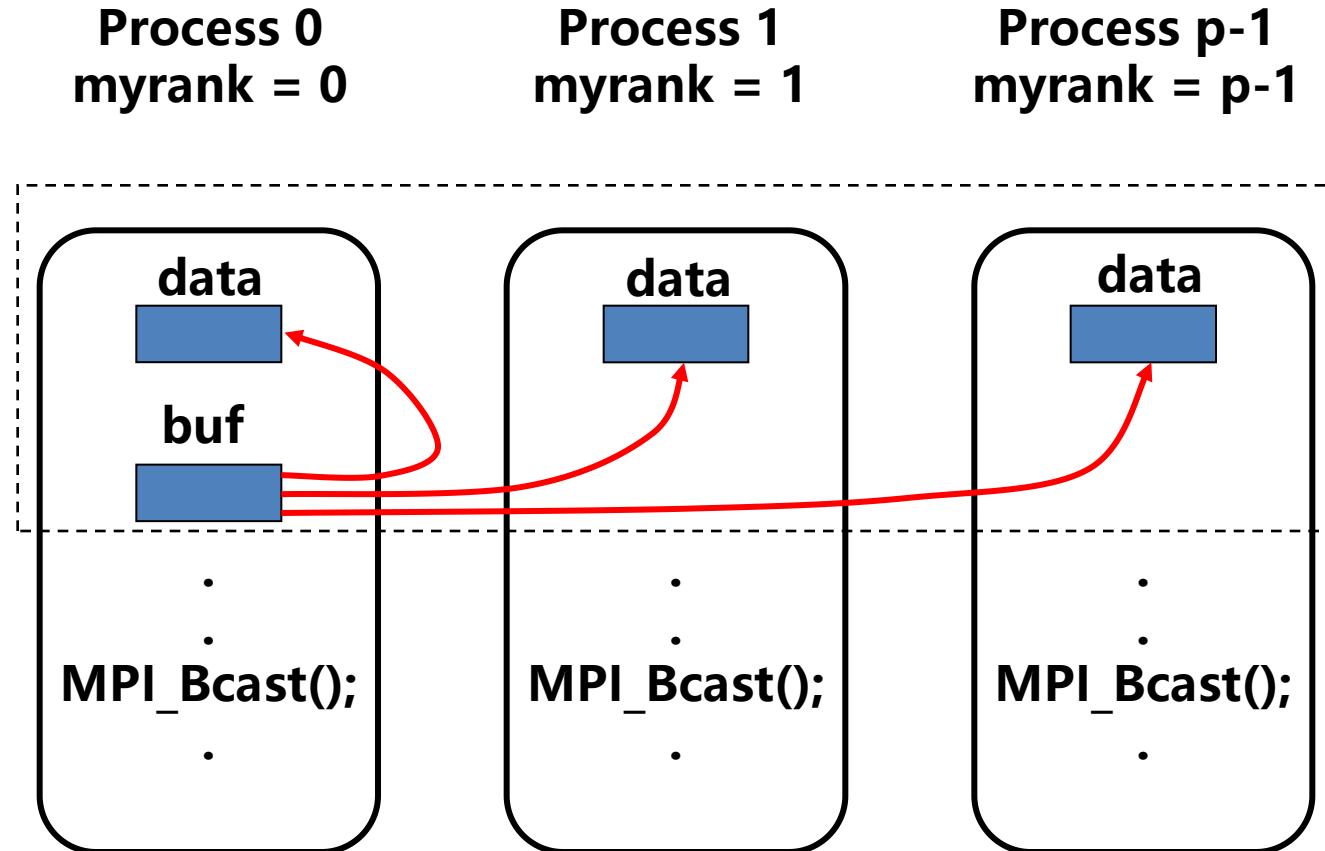
# 聚合通信：广播通信

- 一到全广播通信 (one-to-all broadcast)
- 从一个进程向一组中的其他进程发送数据

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

参数	含义
buf	通信消息缓冲区的起始地址
count	将广播出去/或接收的数据个数
datatype	广播/接收数据的数据类型
root	广播数据的根进程的标识号
comm	通信器，广播通信在通讯器的进程组内进行

# 聚合通信：广播通信



# 聚合通信：广播通信

```
int p, myrank;
float buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
/*得进程编号*/
MPI_Comm_rank(comm, &my_rank);
/* 得进程总数 */
MPI_Comm_size(comm, &p);

if(myrank==0)
    buf = 1.0;
MPI_Bcast(&buf, 1, MPI_FLOAT, 0, comm);
```

# 聚合通信：广播通信

```
int main(int argc, char* argv[]) {  
    int myid, size;  
    MPI_Init(&argc, &argv);  
    char message[124] = "my name is rank0";  
    MPI_Status status;  
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);  
    MPI_Comm_size(MPI_COMM_WORLD, &size);  
  
    if (myid == 0){ //0进程作为根进程  
        MPI_Bcast(message, 124, MPI_CHAR, 0, MPI_COMM_WORLD);  
    }  
    else{  
        //各进程打印接收到的消息  
        MPI_Bcast(message, 124, MPI_CHAR, 0, MPI_COMM_WORLD);  
        printf("rank %d received message is: %s\n", myid, message);  
    }  
    MPI_Finalize();  
    return 0;  
}
```

# 聚合通信：广播通信

```
int main(int argc, char* argv[]) {  
    int myid, size;  
    MPI_Init(&argc, &argv);  
    char message[124] = "my name is rank0";  
    MPI_Status status;  
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);  
    MPI_Comm_size(MPI_COMM_WORLD, &size);  
}
```

```
(base) haomeng@ubuntu:~/projects/mytest/mpi$ mpirun -np 4 ./a.out  
rank 2 received message is: my name is rank0  
rank 3 received message is: my name is rank0  
rank 1 received message is: my name is rank0
```

//各进程打印接收到的消息

```
    MPI_Bcast(message, 124, MPI_CHAR, 0, MPI_COMM_WORLD);  
    printf("rank %d received message is: %s\n", myid, message);  
}  
MPI_Finalize();  
return 0;  
}
```

# 聚合通信：收集通信

- 每个进程(包括根进程)将其发送缓冲区中的内容发送到根进程，根进程根据发送这些数据的进程的序列号将它们依次存放到自己的消息缓冲区中

```
int MPI_Gather(void *sendbuf, int sendcount,  
               MPI_Datatype senddatatype, void *recvbuf, int recvcount,  
               MPI_Datatype recvdatatype, int root, MPI_Comm comm)
```

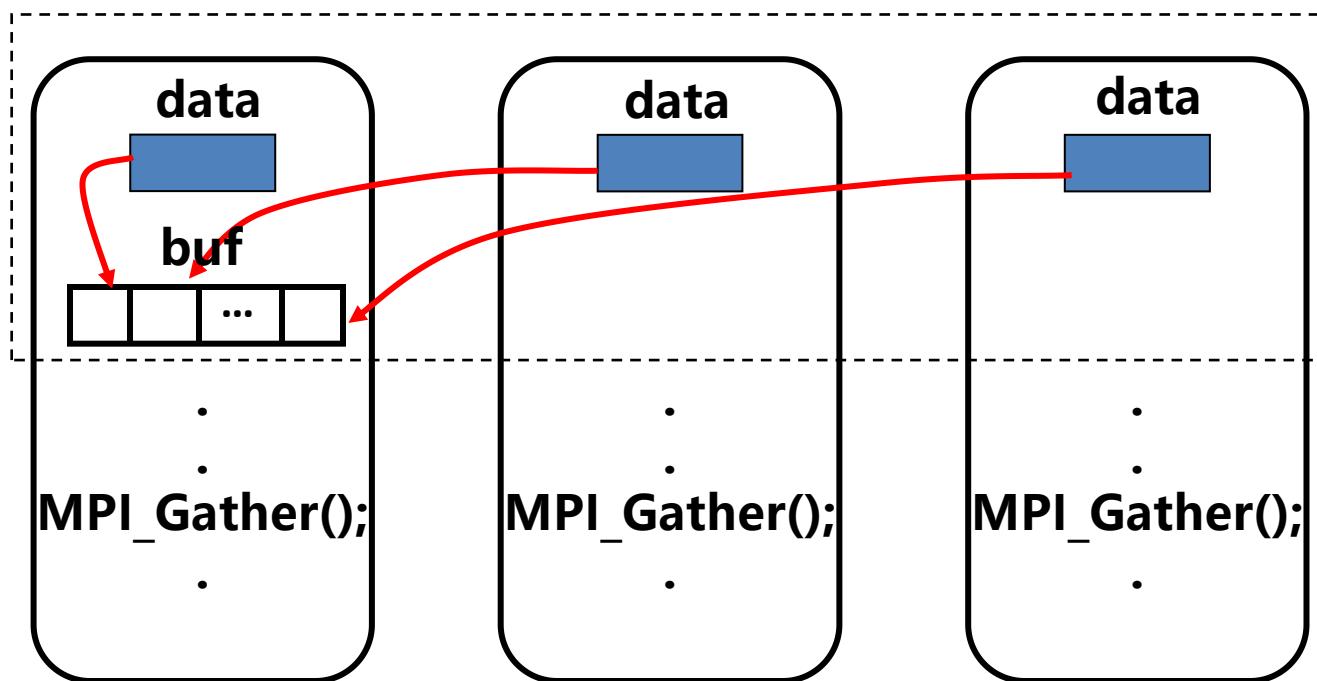
参数	含义
sendbuf	发送消息缓冲区的起始地址
sendcount	发送消息缓冲区中的数据个数
senddatatype	发送消息缓冲区中的数据类型
recvbuf	接收消息缓冲区的起始地址(仅对于根进程有意义)
recvcount	待接收的数据个数(仅对于根进程有意义)
recvdatatype	接收数据的数据类型(仅对于根进程有意义)
root	接收进程的标识号
comm	通信器

# 聚合通信：收集通信

Process 0  
myrank = 0

Process 1  
myrank = 1

Process p-1  
myrank = p-1



# 聚合通信：收集通信

```
int p, myrank;
float data[10]; /*分布变量*/
float* buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
/*得进程编号*/
MPI_Comm_rank(comm,&my_rank);
/* 得进程总数 */
MPI_Comm_size(comm, &p);
...
if(myrank==0){
    /*开辟接收缓冲区*/
    buf=(float*)malloc(p*10*sizeof(float));
}
MPI_Gather(data,10,MPI_FLOAT,buf,10,MPI_FLOAT,0,comm);
```

# 聚合通信：分散通信

- 源（根）进程一个数组中的每个元素发送到不同目标进程。分散（Scatter）操作是收集（Gather）操作的逆操作

```
int MPI_Scatter(void *sendbuf, int sendcount,  
                MPI_Datatype senddatatype, void *recvbuf, int recvcount,  
                MPI_Datatype recvdatatype, int root, MPI_Comm comm)
```

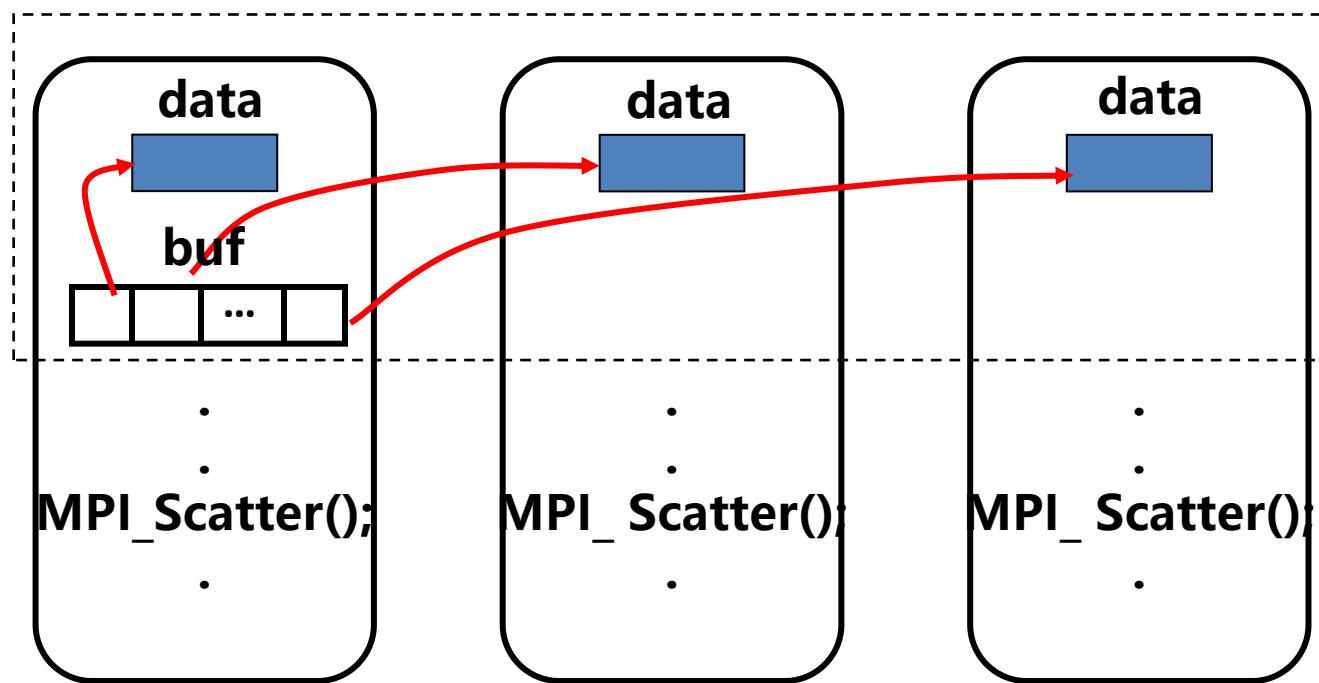
参数	含义
sendbuf	发送消息缓冲区的起始地址
sendcount	发送到各个进程的数据个数
senddatatype	发送消息缓冲区中的数据类型
recvbuf	接收消息缓冲区的起始地址
recvcount	待接收的数据个数
recvdatatype	接收数据的数据类型
root	发送进程的标识号
comm	通信器

# 聚合通信：分散通信

Process 0  
myrank = 0

Process 1  
myrank = 1

Process p-1  
myrank = p-1



# 聚合通信：分散通信

```
int p, myrank;
float data[10];
float* buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
/*得进程编号*/
MPI_Comm_rank(comm,&my_rank);
/* 得进程总数 */
MPI_Comm_size(comm, &p);
...
if(myrank==0){
    /*开辟发送缓冲区*/
    buf = (float*)malloc(p*10*sizeof(float));
    ...
}
MPI_Scatter(buf,10,MPI_FLOAT,data,10,MPI_FLOAT,0,comm);
```

# 聚合通信：规约通信

- 对组中所有进程的发送缓冲区中的数据用OP参数指定的操作进行运算，并将结果送回到根进程的接收缓冲区中

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

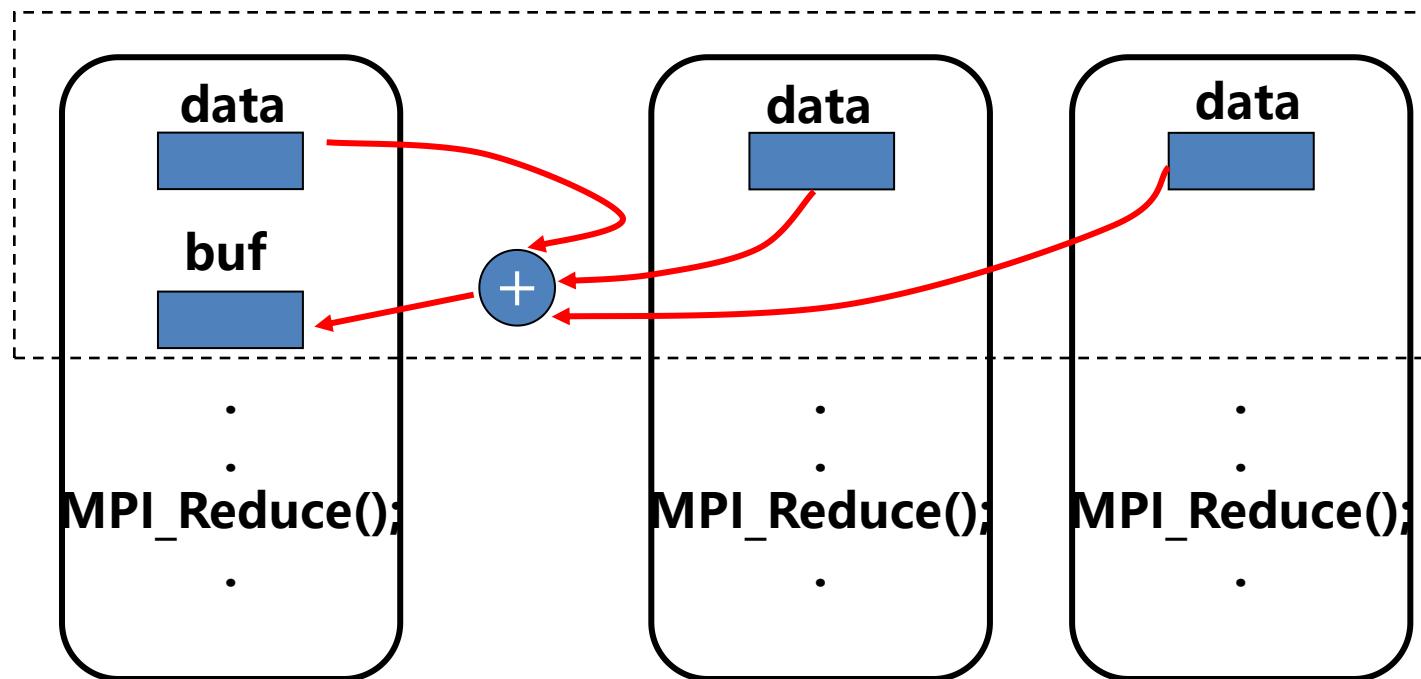
参数	含义
sendbuf	发送消息缓冲区的起始地址
recvbuf	接收消息缓冲区中的地址
count	发送消息缓冲区中的数据个数
datatype	发送消息缓冲区的数据类型
op	归约操作符
root	根进程的标识号
comm	通信器

# 聚合通信：规约通信

Process 0  
myrank = 0

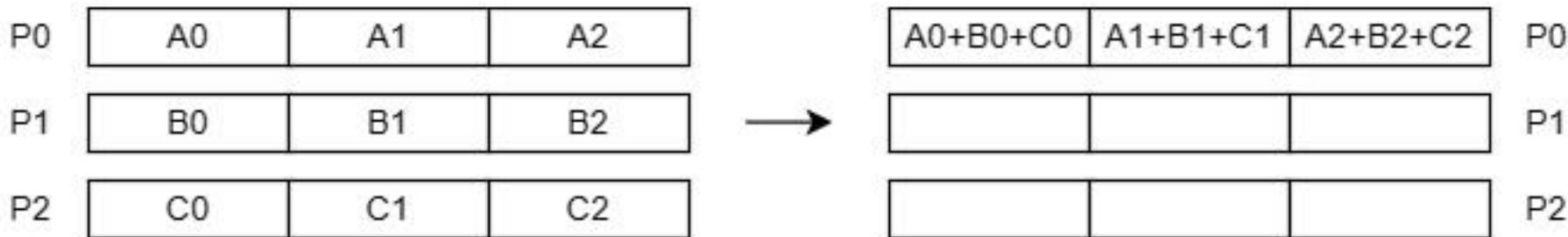
Process 1  
myrank = 1

Process p-1  
myrank = p-1



# 聚合通信：规约通信

- 所有进程所提供的数据长度相同、类型相同



# 聚合通信：规约通信

## ■ MPI 中预定义的运算操作 (op)

操作名	含义
<b>MPI_MAX</b>	求最大值
<b>MPI_MIN</b>	求最小值
<b>MPI_SUM</b>	求和
<b>MPI_PROD</b>	求积
<b>MPI_BAND</b>	逻辑与
<b>MPI_BOR</b>	二进制按位与

操作名	含义
<b>MPI_LOR</b>	逻辑或
<b>MPI_BOR</b>	二进制按位或
<b>MPI_LXOR</b>	逻辑异或
<b>MPI_BXOR</b>	二进制按位异或
<b>MPI_MAXLOC</b>	求最大值和所在位置
<b>MPI_MINLOC</b>	求最小值和所在位置

## ■ 每种归约运算操作所允许的数据类型

运算操作 OP	允许的数据类型
<b>MPI_MAX, MPI_MIN</b>	整型和实型
<b>MPI_SUM, MPI_PROD</b>	整型、实型和复型
<b>MPI_BAND, MPI_LOR, MPI_LXOR</b>	C的整型
<b>MPI_BAND, MPI_BOR, MPI_BXOR</b>	整型和二进制型(MPI_BYTE)

# 聚合通信：规约通信

- 归约操作可以使用MPI预定义的运算操作，也可以使用用户自定义的运算操作，必须满足结合律
  - 创建自定义归约运算操作: **MPI\_Op\_create**
  - 释放自定义的归约操作: **MPI\_Op\_free**

```
int MPI_Op_create(MPI_User_function * func, int commute,  
                      MPI_Op *op)
```

参数	含义
<b>func</b>	用户自定义的函数
<b>commute</b>	如果用户自定义的运算可交换则为true, 否则为false
<b>op</b>	运算操作符

```
int MPI_Op_free(MPI_Op * op)
```

# 聚合通信：规约通信

```
int p, myrank;
float data = 0.0;
float buf;
MPI_Comm comm;
MPI_Init(&argc, &argv);
/*得进程编号*/
MPI_Comm_rank(comm,&my_rank);

/*各进程对data进行不同的操作*/
data = data + myrank * 10;

/*将各进程中的data数相加并存入根进程的buf中 */
MPI_Reduce(&data,&buf,1,MPI_FLOAT,MPI_SUM,0,comm);
```

# 聚合通信：其他

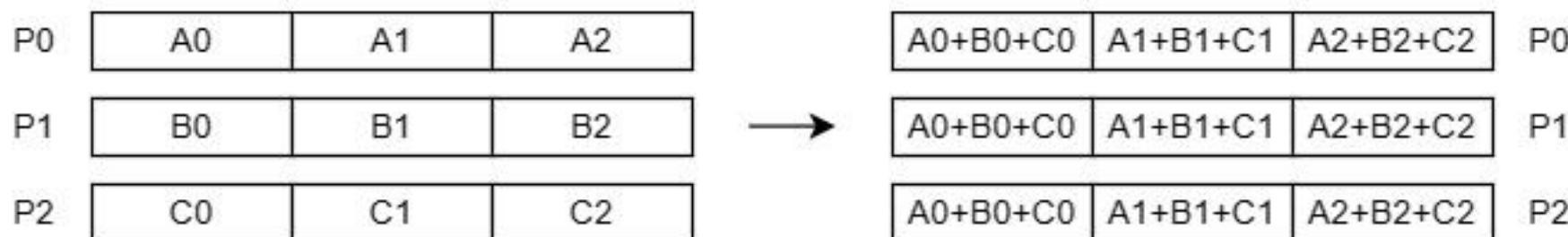
- **MPI\_Allreduce**: 全规约函数，归约并将结果发送到所有进程

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

参数	含义
sendbuf	发送消息缓冲区的起始地址
recvbuf	接收消息缓冲区的起始地址
count	发送消息缓冲区中的数据个数
datatype	发送消息缓冲区中的数据类型
op	归约操作符
comm	通信器

# 聚合通信：其他

- **MPI\_Allreduce**: 全规约函数，归约并将结果发送到所有进程
- 所有进程的recvbuf将同时获得归约运算的结果
- 相当于**MPI\_Reduce**后再将结果进行一次广播  
**MPI\_Bcast**



# 聚合通信：其他

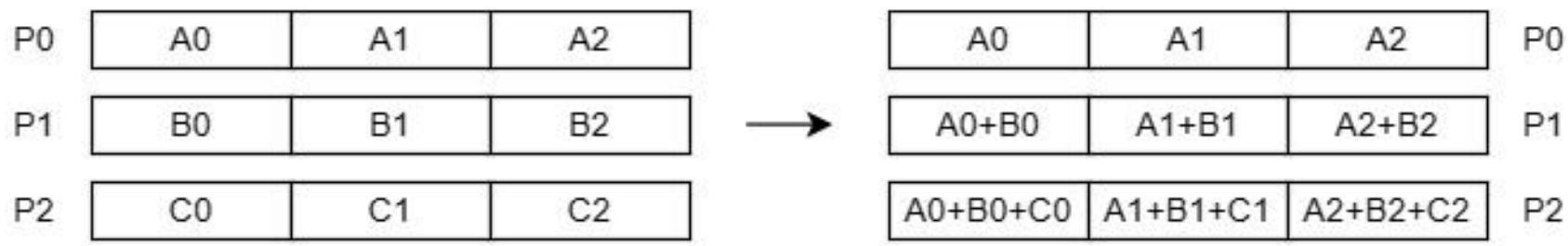
- **MPI\_Scan**: 扫描函数，特殊的规约，每一个进程都对排在它前面的进程进行归约操作

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

参数	含义
sendbuf	发送消息缓冲区的起始地址
recvbuf	接收消息缓冲区的起始地址
count	输入缓冲区中元素的个数
datatype	输入缓冲区中元素的类型
op	归约操作符
comm	通信器

# 聚合通信：其他

- **MPI\_Scan**: 扫描函数
- 每一个进程都对排在它前面的进程进行归约操作，操作结束后，第*i*个进程中的recvbuf中将包含前*i*个进程的归约结果
- 0号进程接收缓冲区中的数据就是其发送缓冲区的数据



# 聚合通信：其他

- **MPI\_Allgather**: 全收集函数，从所有进程收集数据到所有进程（每个进程最后获得相同的所有数据）

```
int MPI_Allgather (void * sendbuf, int sendcount,  
                   MPI_Datatype sendtype, void * recvbuf,  
                   int recvcount, MPI_Datatype recvtype,  
                   MPI_Comm comm)
```

参数	含义
<b>sendbuf</b>	发送消息缓冲区的起始地址
<b>sendcount</b>	发送消息缓冲区中的数据个数
<b>sendtype</b>	发送消息缓冲区中的数据类型
<b>recvbuf</b>	接收消息缓冲区的起始地址
<b>recvcount</b>	从其它进程中接收的数据个数
<b>recvtype</b>	接收消息缓冲区的数据类型
<b>comm</b>	通信器

# 聚合通信：其他

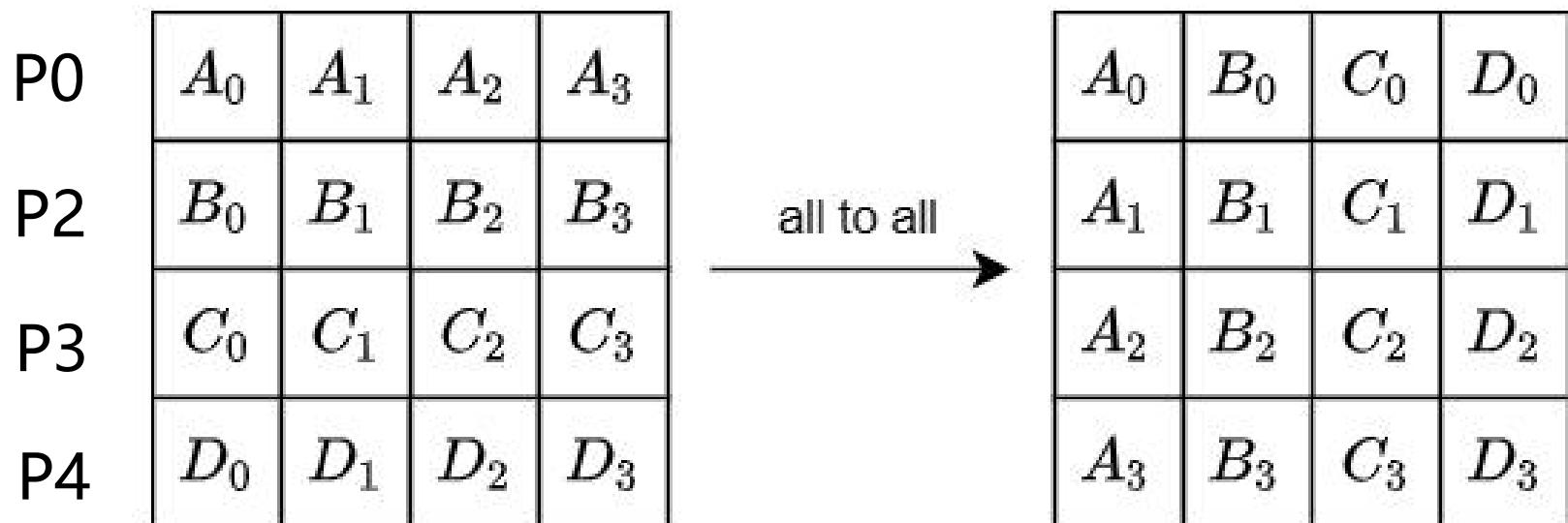
## ■ MPI\_Alltoall: 全收集分散函数

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

参数	含义
sendbuf	发送消息缓冲区的起始地址
sendcount	发送到每个进程的数据个数
sendtype	发送消息缓冲区中的数据类型
recvbuf	接收消息缓冲区的起始地址
recvcount	从每个进程中接收的元素个数
recvtype	接收消息缓冲区的数据类型
comm	通信器

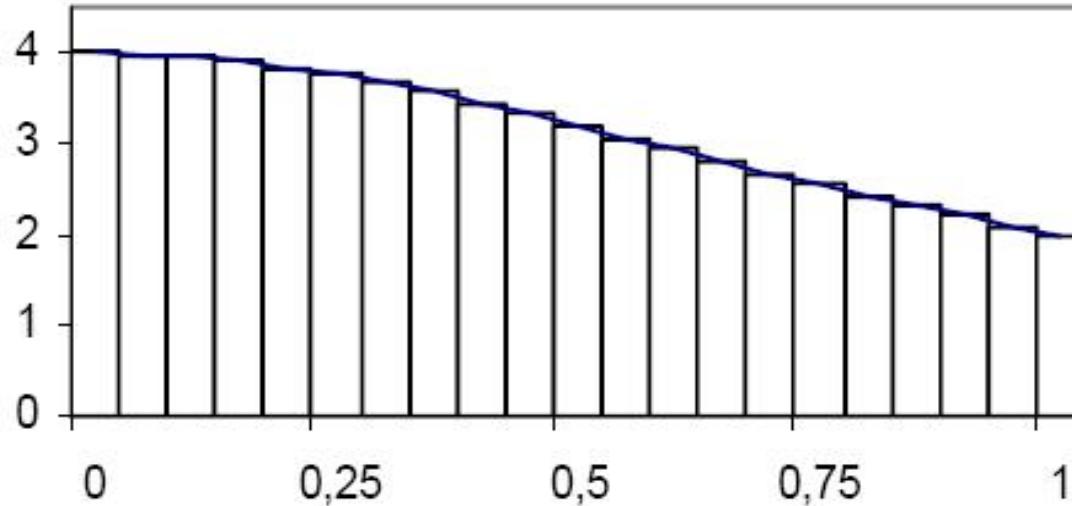
# 聚合通信：其他

- **MPI\_Alltoall**: 全收集分散函数
- 每个进程发送一个消息给n个进程，包括它自己，这n个消息的发送缓冲区中以标号的顺序有序地存放
- 一次全局交换中共有 $n^2$ 个消息进行通信



# 示例：计算 $\pi$

## ■ 通过积分的方法计算 $\pi$

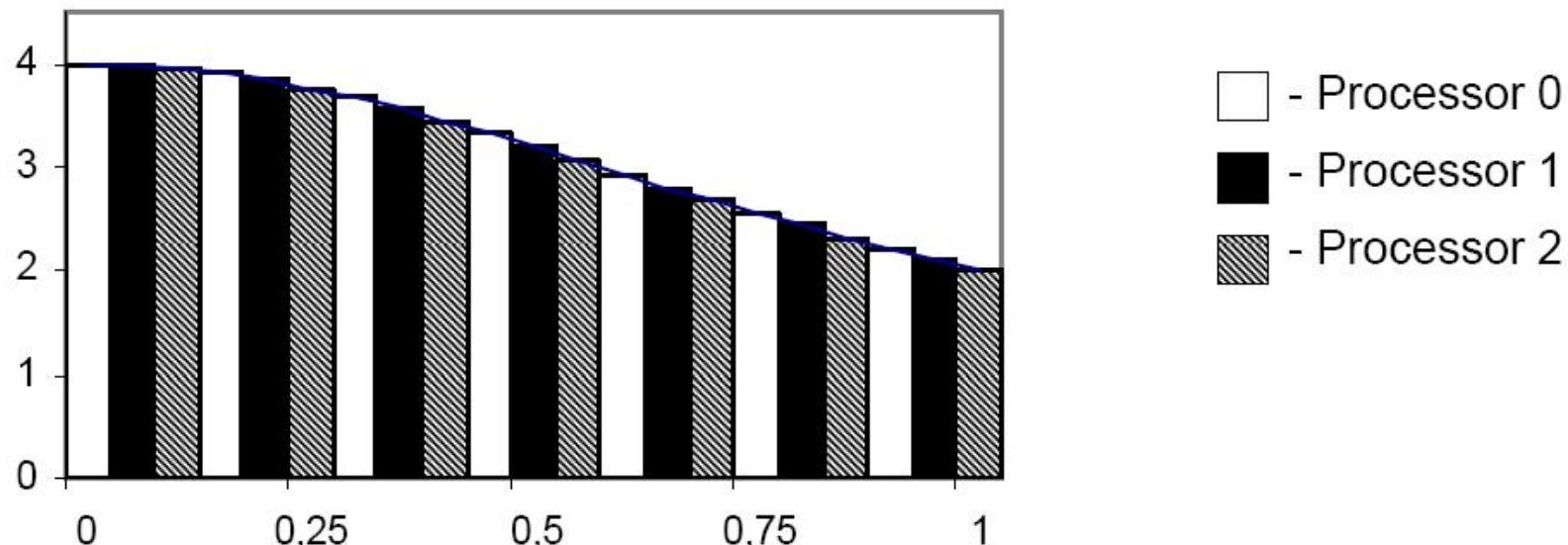


## ■ 使用矩形法来计算数值积分

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

# 计算π

- 在处理器之间循环分配计算负载
- 在不同处理器上计算的部分和最终必须相加



# 计算π：串行程序

```
int num_steps = 1000;
double width;
int main ()
{
    int i;
    double x, pi, sum = 0.0;
    width = 1.0 / (double) num_steps;
    for (i=1; i <= num_steps; i++) {
        x = (i-0.5)* width;
        sum = sum + 4.0 / (1.0+x*x);
    }
    pi = sum * width;
    return 0;
}
```

# 计算π：MPI并行程序（点对点通信）

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
int main(int argc, char *argv[]){
    int n, myid, numprocs, i;
    double mypi, pi, h, sum, x;
    MPI_Status st;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if(myid == 0){
        printf("Enter the number of intervals: \n");
        scanf("%d", &n);
        for(i = 1; i < numprocs; i++)
            MPI_Send(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    } else {
        MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &st);
    }
```

# 计算π：MPI并行程序（点对点通信）

```
h = 1.0/(double)n;
sum = 0.0;
for(i = myid + 1; i <= n; i+= numprocs){
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x * x);
}
mypi = h * sum;
if(myid != 0)
    MPI_Send(&mypi, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
else {
    pi = mypi;
    for(i = 1; i < numprocs; i++){
        MPI_Recv(&mypi,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD,&st);
        pi += mypi;
    }
    printf("pi is approximately %.16f\n", pi);
}
MPI_Finalize();
return 0;
}
```

# 计算π：MPI并行程序（聚合通信）

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
int main(int argc, char *argv[]){
    int n, myid, numprocs, i;
    double mypi, pi, h, sum, x;
    MPI_Status st;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if(myid == 0){
        printf("Enter the number of intervals: \n");
        scanf("%d", &n);
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

# 计算π：MPI并行程序（聚合通信）

```
h = 1.0/(double)n;
sum = 0.0;
for(i = myid + 1; i <= n; i+= numprocs){
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x * x);
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if(myid == 0)
    printf("pi is approximately %.16f\n", pi);
MPI_Finalize();
return 0;
}
```

# 计时：MPI\_Wtime

- 返回从过去某一时刻开始所经过的秒数

```
double MPI_Wtime(void)
```

- 对一个MPI代码块进行计时：

```
double start, finish;  
...  
start = MPI_Wtime();  
/*Code to be timed*/  
...  
finish = MPI_Wtime();  
print("Proc %d > Elapsed time = %e seconds\n", my_rank,  
      finish - start);
```

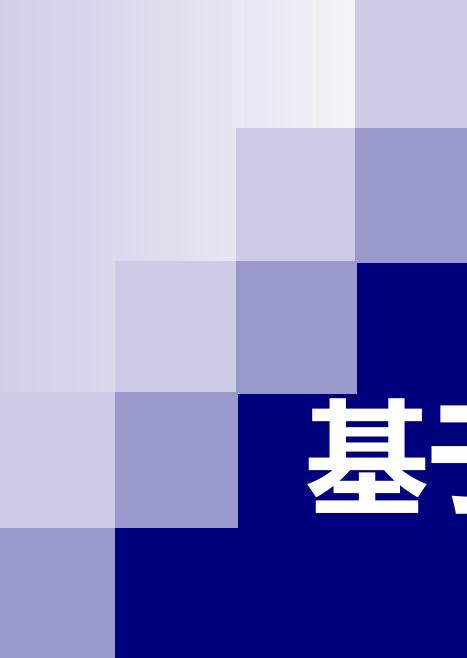
# 计时：MPI\_Wtime

- 并行程序会为每个进程报告一次时间，但我们需要获得一个总的单独时间
- 理想情况是，所有的进程同时开始运算，当最后一个进程完成运算时，能获取从开始到最后一个进程结束之间的时间开销。换句话讲，**并行时间取决于“最慢”进程花费的时间**

```
double local_start, local_finish, local_elapsed, elapsed;  
...  
MPI_Barrier(comm);  
local_start = MPI_Wtime();  
/*Code to be timed*/  
...  
local_finish = MPI_Wtime();  
local_elapsed = local_finish - local_start;  
MPI_Reduce(&local_elapsed,&elapsed,1,MPI_DOUBLE,MPI_MAX,0,comm);  
if(my_rank == 0)  
    print("Elapsed time = %e seconds\n",elapsed);
```

# Reference

- **Message Passing Interface (MPI)**, <https://hpc-tutorials.llnl.gov/mpi/>
- **并行计算课程, 陈国良, 中国科学技术大学**
- **MPI: A Message-Passing Interface Standard**,  
<https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>
- **Blaise Barney, Lawrence Livermore National Laboratory, Introduction to Parallel Computing**,  
[https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)
- **The Message Passing Interface (MPI) standard**,  
<http://www-unix.mcs.anl.gov/mpi/>



# 第六章

# 基于异构系统的并行计算

哈尔滨工业大学

张伟哲

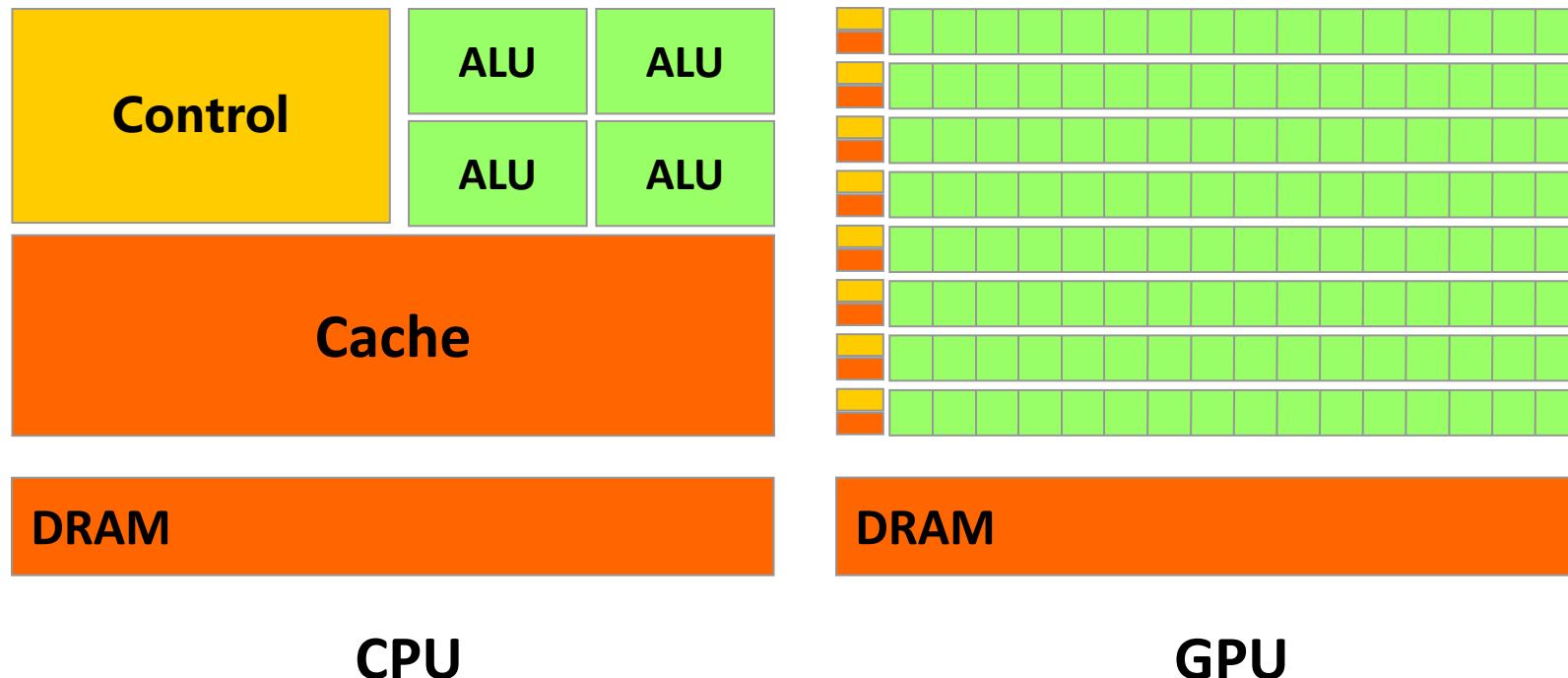
2025, Fall Semester

# 什么是GPU计算

- GPU (Graphic Processing Unit) : 从图形处理到通用计算
- GPU为高度并行的实时3D渲染计算而设计, 高 GFLOPS , 高带宽
- 3D渲染技术及3D API的发展, 促进GPU向通用计算处理器发展
- NVIDIA GPU为通用计算专门优化设计, 于2007年推出CUDA (Compute Unified Device Architecture)

# GPU与CPU硬件架构的对比

- 与CPU相比，GPU具有更多的计算单元，但对于复杂控制过程的处理能力则比CPU差
- GPU更适用于同时进行大量简单的统一操作



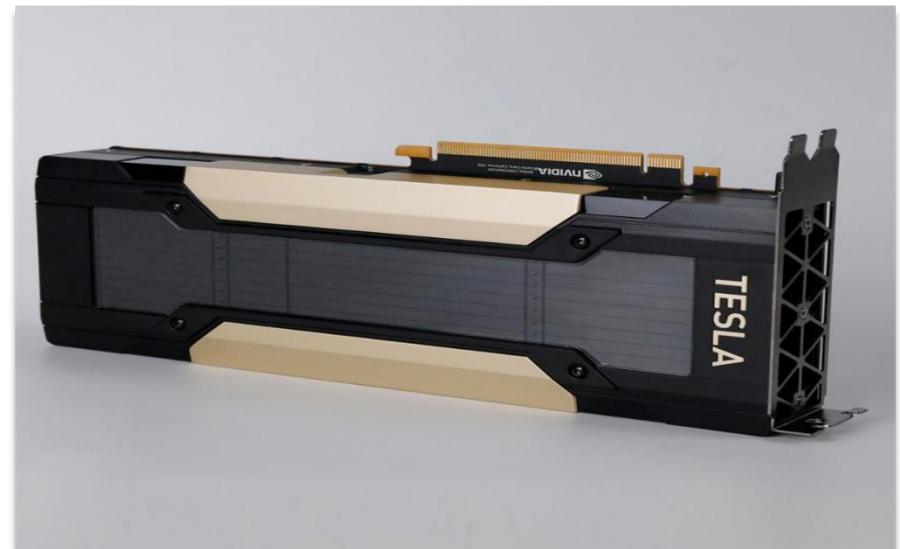
# GPU与CPU硬件架构的对比

## CPU (Skylake)

- 40 cores (2 20-core chips)
- 2 threads each
- 2xAVX-512, so 2x8 in double precision
- ~640 way parallelism (40\*2\*8)

## GPU (V100)

- 80 SM
- 64 warps per SM
- 32 threads per warp in double precision
- ~150,000+ way parallelism (80\*64\*32)



# GPU与CPU应用范围对比

## ■ CPU: control processor

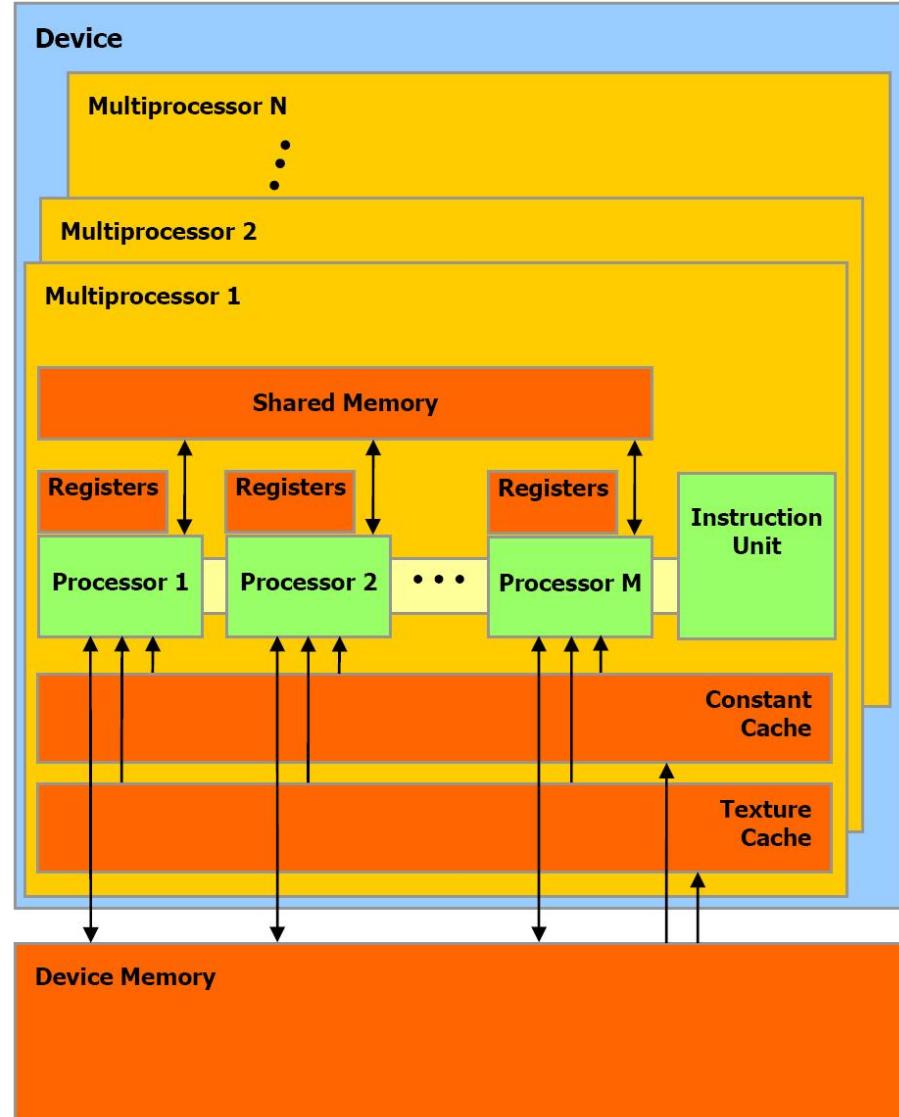
- 不规则数据结构
- 不可预测存取模式
- 递归算法
- 分支密集型算法
- 单线程程序

## ■ GPU: data processor

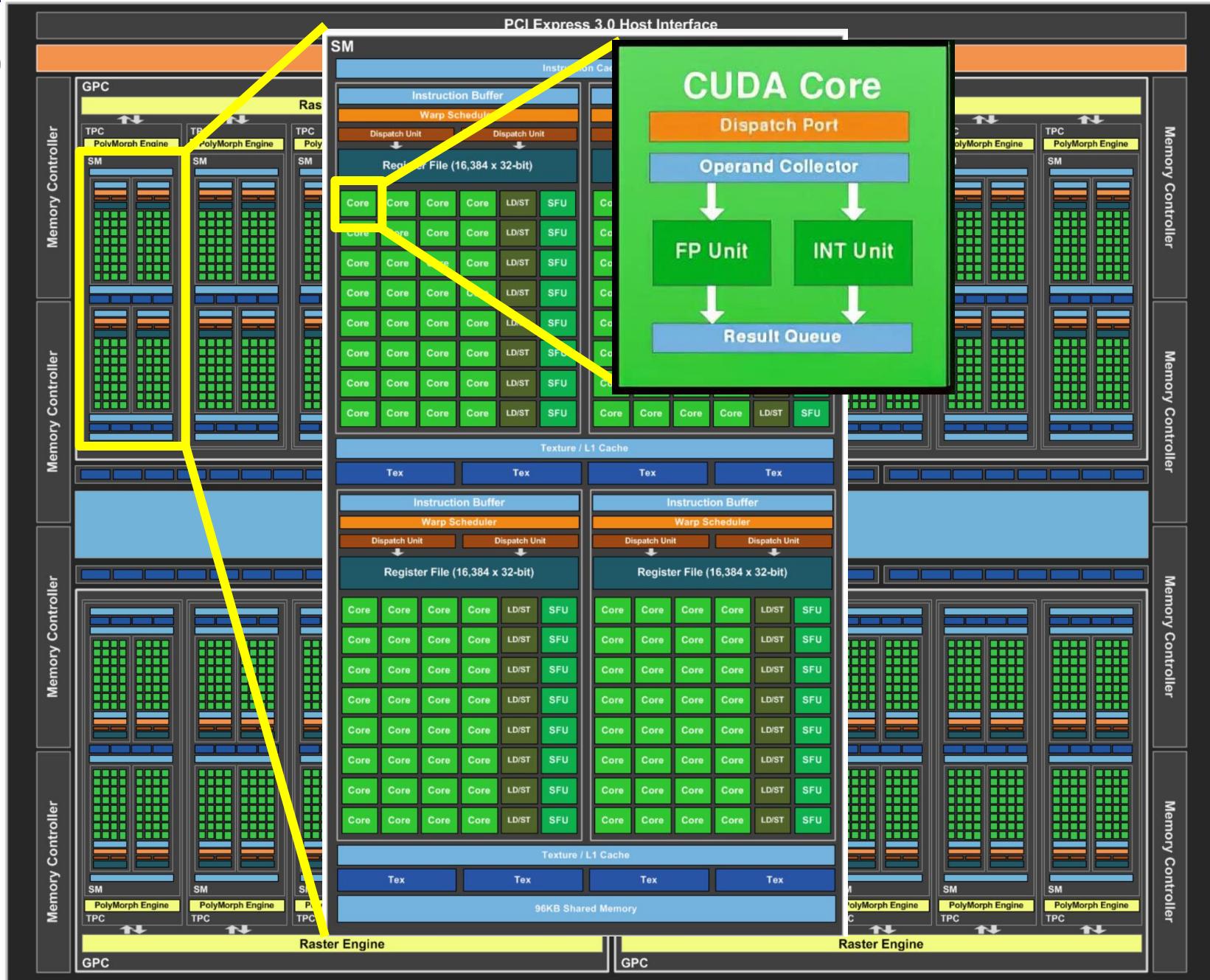
- 规则数据结构
- 可预测存取模式
- 油气勘探、金融分析、医疗成像、有限元、基因分析、地理信息系统、深度学习...

# GPU基本硬件架构

- 流多处理器 (SM, Stream Multiprocessor)
- 流处理器 (SP, Stream Processor)
- 共享内存 (Shared Memory)
- 板载显存 (Device Memory)

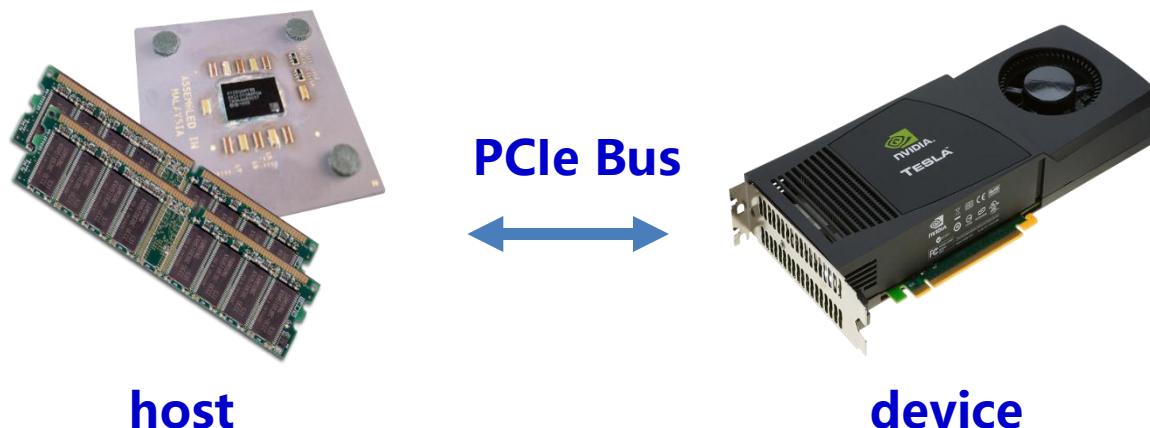






# 异构计算

- GPU并不是独立运行的计算平台，需要与CPU协同工作，可以看成CPU的协处理器
- GPU计算，实际是基于CPU+GPU的异构计算
- 在异构计算架构中，GPU与CPU通过PCIe总线连接，CPU所在位置是主机端（host），GPU所在位置称为设备端（device）



# 异构计算

- GPU包括更多运算核心，适合数据并行的计算密集型任务，如大型矩阵运算
- CPU的运算核心较少，但是其可以实现复杂的逻辑运算，适合控制密集型任务
- CPU上的线程是重量级的，上下文切换开销大
- GPU由于存在众多核心，线程是轻量级的
- CPU+GPU的异构系统可以优势互补，CPU负责处理逻辑复杂的串行任务，GPU重点处理数据密集型的并行计算任务

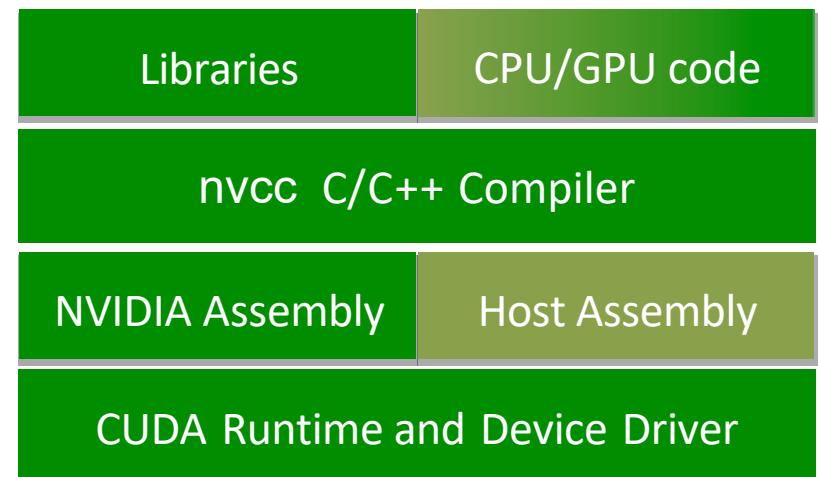
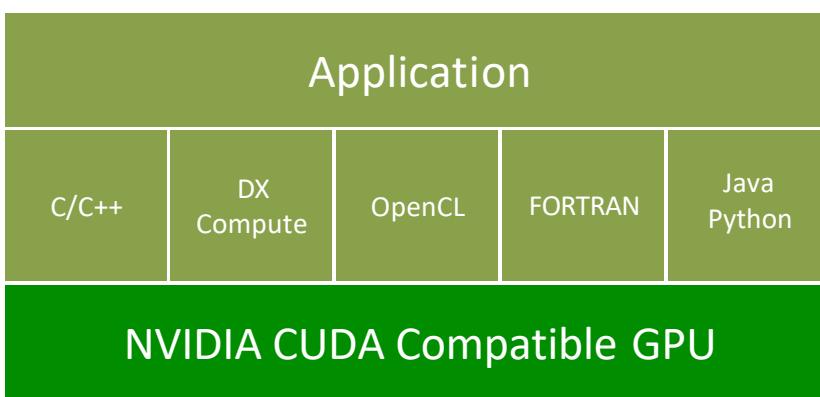
# CUDA

- CUDA是NVIDIA公司所开发的GPU编程模型
- 提供了**GPU编程的简易接口**，基于CUDA编程可以构建基于GPU计算的应用程序
- CUDA提供了对其它编程语言的支持，如C/C++，Python，Fortran等语言
- 硬件：NVIDIA GPU，可以是**专门的科学计算卡**，比如Tesla A100），也可以是**普通的游戏显卡**，比如GTX或者RTX
- 软件：**CUDA Toolkit**：Windows、Mac 和大多数标准 Linux 发行版都支持  
<https://developer.nvidia.com/cuda-toolkit>

# CUDA

CUDA Device Driver  
CUDA Toolkit (**compiler, debugger, profiler, lib**)  
CUDA SDK (examples)  
Windows, Mac OS, Linux

## Parallel Computing Architecture



Libraries – FFT, Sparse Matrix, BLAS, RNG, CUSP, Thrust...

# 异构计算

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    shared _int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out; // host copies of a, b, c
    int *d_in, *d_out; // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **)&d_in, size);
    cudaMalloc((void **)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

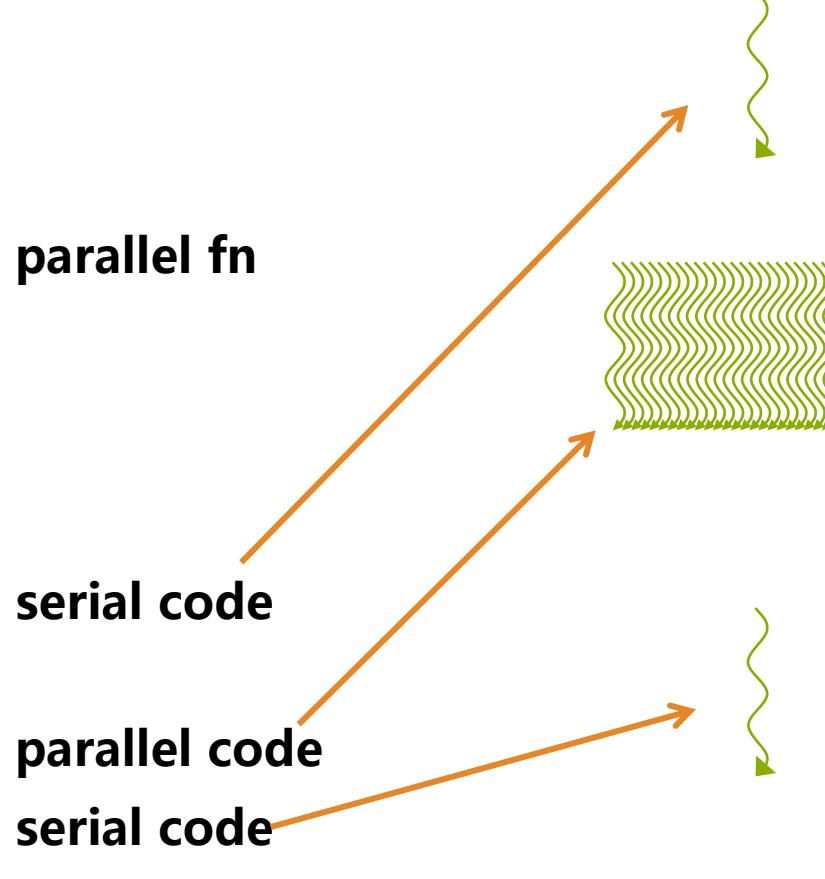
    // Clean up
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel fn

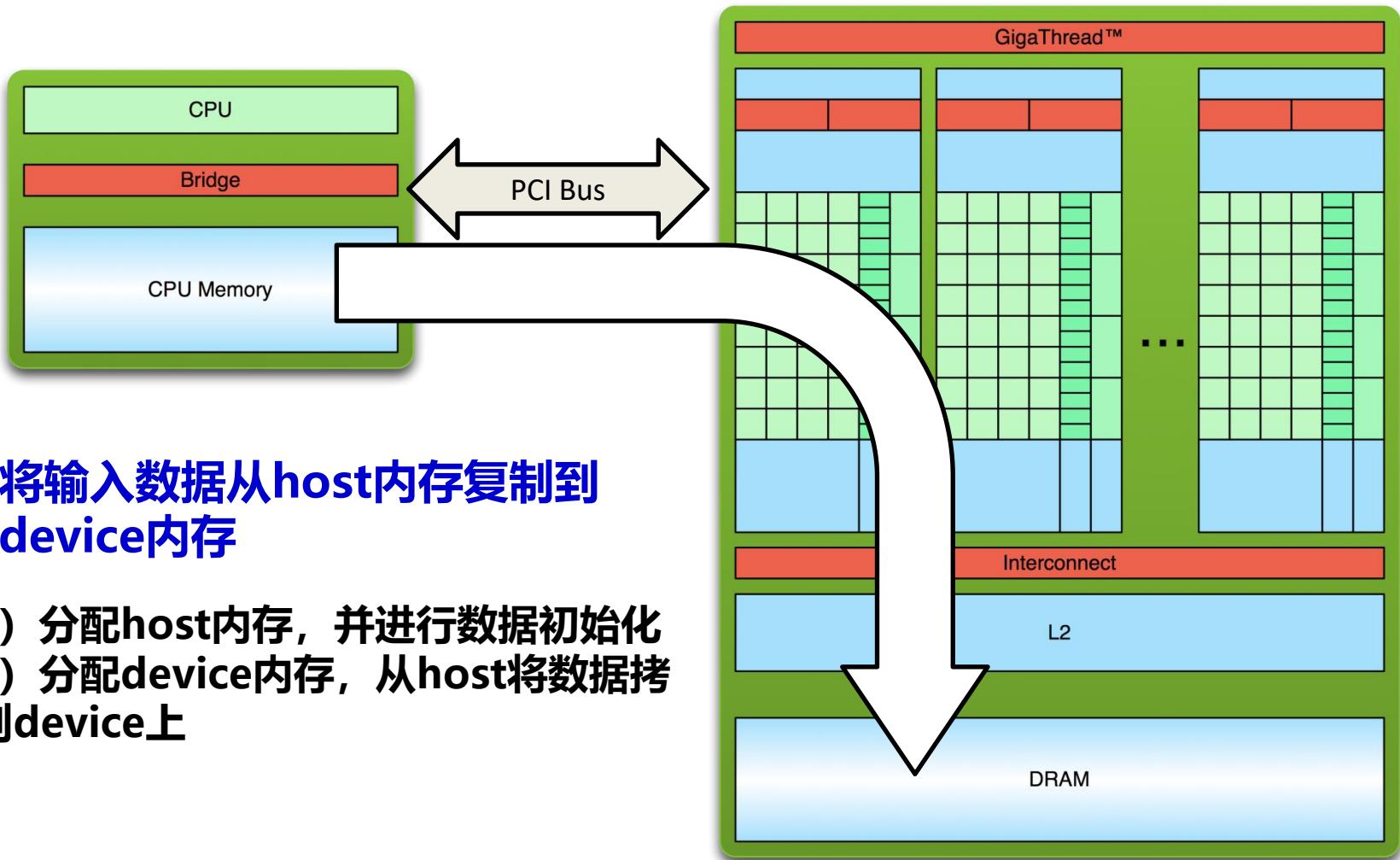
serial code

parallel code

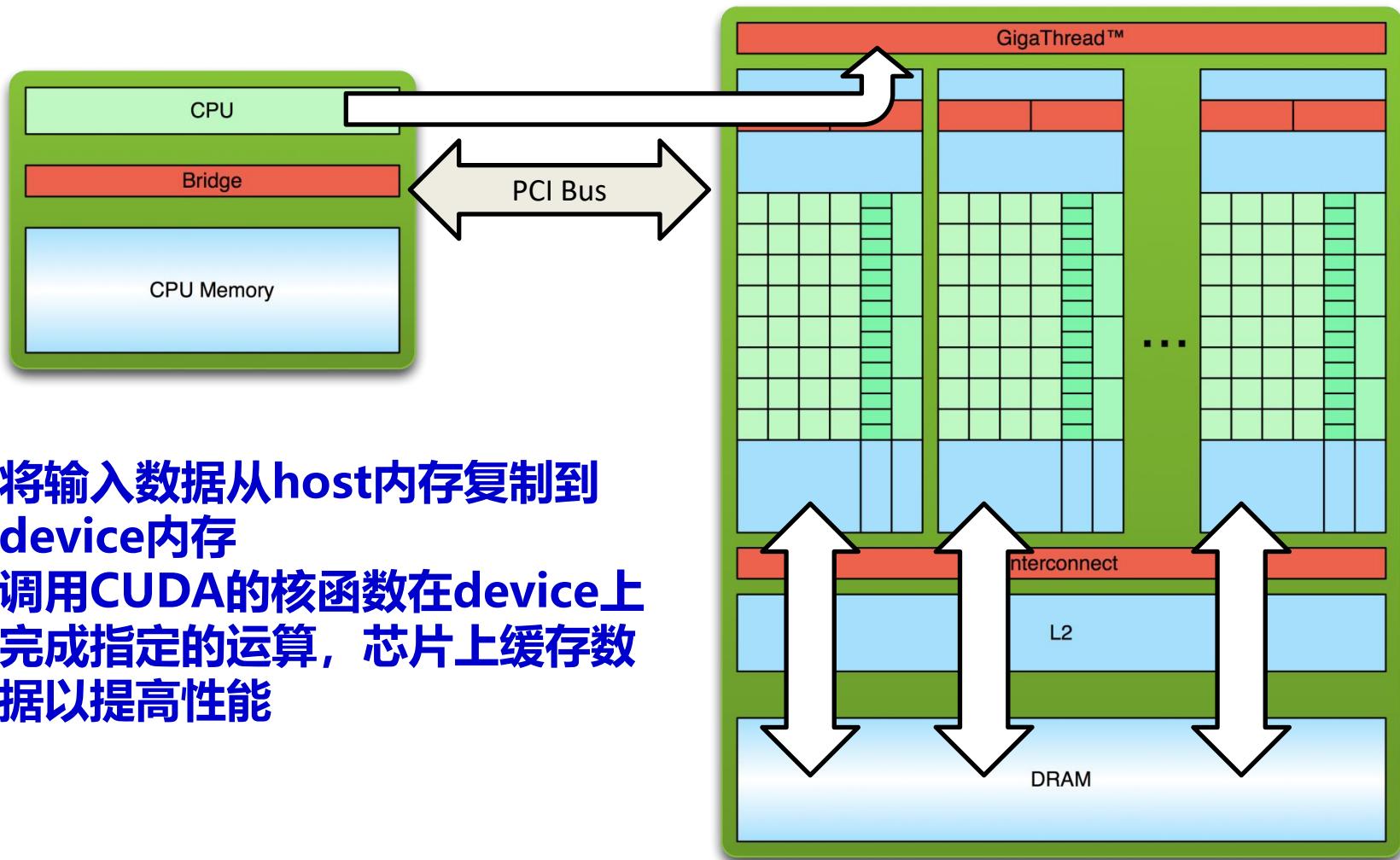
serial code



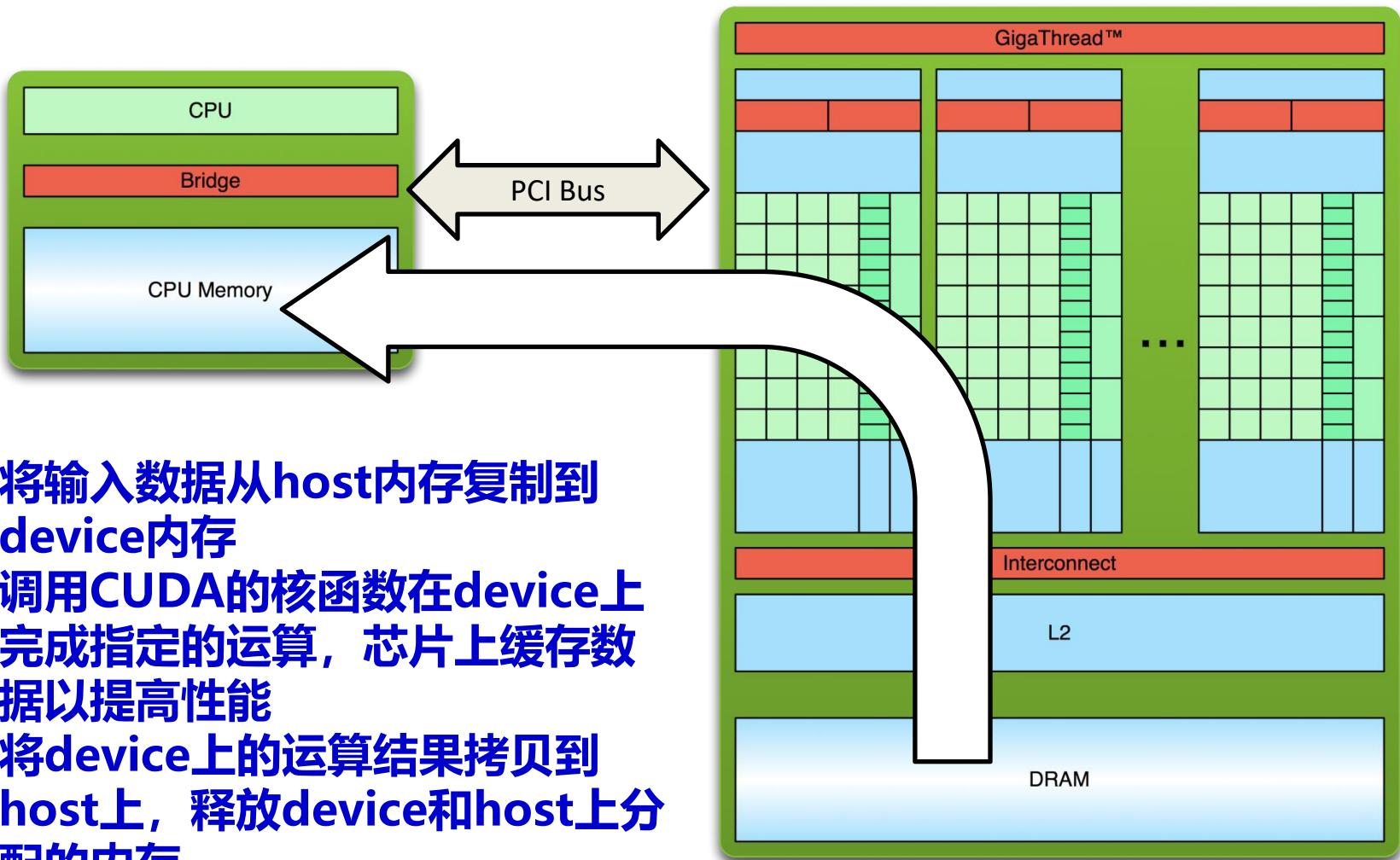
# CPU-GPU协同处理流程



# CPU-GPU协同处理流程



# CPU-GPU协同处理流程



1. 将输入数据从host内存复制到device内存
2. 调用CUDA的核函数在device上完成指定的运算，芯片上缓存数据以提高性能
3. 将device上的运算结果拷贝到host上，释放device和host上分配的内存

# Hello World!

```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

Output:

```
$ nvcc hello_world.cu  
$ ./a.out  
Hello World!  
$
```

■ 标准C代码运行在host上

■ NVIDIA编译器 (nvcc) 可用于编译没有device代码的程序

# Hello World! with Device Code

```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- 包含两个新的语法元素...

# CUDA函数定义

```
__global__ void mykernel(void) {  
}
```

■CUDA C/C++关键字 **\_\_global\_\_** 表示一个**kernel**函数：

- 在**device**上运行
- 从**host**代码调用

■nvcc将源代码分为**host**部分和**device**部分

- **device**函数（例如**mykernel()**）由**NVIDIA**编译器编译
- **host**函数（例如**main()**）标准主机编译器编译，如**gcc**

# CUDA函数定义

	执行	调用
<code>_device_ float DeviceFunc()</code>	Device	Device
<code>_global_ void KernelFunc()</code>	Device	Host
<code>_host_ float HostFunc()</code>	Host	Host

- **\_global\_ 定义kernel函数**
  - 必须返回void
  - kernel 函数都是异步执行
- **\_device\_ 和 \_host\_ 可以组合使用**
  - 则被定义的函数在CPU和GPU上都被编译
- **不添加限定词时，函数默认为 \_host\_**

# Hello World! with Device Code

```
mykernel<<<1,1>>>();
```

- <<< >>> 尖括号表示从 host 代码到 device 代码的调用
  - 也称为 “**kernel启动**”，异步
  - 里面的参数(1,1)，分别表示线程块的个数、每个线程块中线程的个数，乘积就是启动的线程数

这就是在 GPU 上执行函数所需的全部内容！

# Hello World! with Device Code

```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

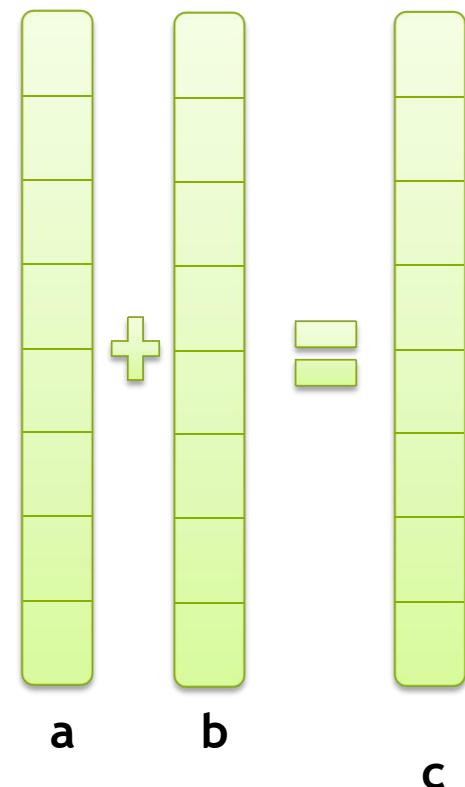
## Output:

```
$ nvcc hello.cu  
$ a.out  
Hello World!  
$
```

mykernel()什么都没做

# Vector Addition

- GPU是适合大规模并行的
- 我们从两个整数相加开始，然后构建向量加法



# Addition on the Device

## ■ 将两个整数相加的简单kernel

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- **\_\_global\_\_** 是CUDA C/C++关键字
  - **add()** 将在device端执行
  - **add()** 将从host端调用

# Addition on the Device

- kernel函数中变量是指针类型

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- add()在device上运行，a、b和c必须指向device内存
- 因此，需要在GPU上分配内存

# 内存管理

- host内存和device内存是独立的实体

- device指针指向GPU内存
- host指针指向CPU内存



- 用于处理device内存的CUDA API

- `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- 类似C语言的`malloc()`, `free()`, `memcpy()`

```
cudaError_t cudaMalloc(void** devPtr, size_t size)
```

```
cudaError_t cudaFree ( void* devPtr )
```

# 内存管理

```
cudaError_t cudaMemcpy(void *dist, const void* src,  
size_t count, cudaMemcpyKind kind)
```

- 在 host 和device之间复制数据， cudaMemcpyKind 指定复制的方向

kind	含义
cudaMemcpyDeviceToHost	从设备向主机拷贝
cudaMemcpyHostToDevice	从主机向设备拷贝
cudaMemcpyHostToHost	从主机向主机拷贝
cudaMemcpyDeviceToDevice	从设备向设备拷贝

```
cudaError_t cudaMemcpyAsync(void *dist, const void* src,  
size_t count, cudaMemcpyKind kind,  
cudaStream_t stream = 0)
```

- 异步数据传输，可以在调用函数后立即返回，无需等待数据传输完成

# Addition on the Device

## ■ add() 核函数

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

## ■ main() 函数

```
int main(void) {  
    int a, b, c;                      // host copies of a, b, c  
    int *d_a, *d_b, *d_c;   // device copies of a, b, c  
    int size = sizeof(int);  
  
    // Allocate space for device copies of a, b, c  
    cudaMalloc((void **) &d_a, size);  
    cudaMalloc((void **) &d_b, size);  
    cudaMalloc((void **) &d_c, size);  
  
    // Setup input values  
    a = 2;  
    b = 7;
```

# Addition on the Device

## ■ main() 函数

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice) ;
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice) ;

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c) ;

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost) ;

printf("%d\n", c) ;
// Cleanup
cudaFree(d_a) ; cudaFree(d_b) ; cudaFree(d_c) ;
return 0;
}
```

# 开始并行化

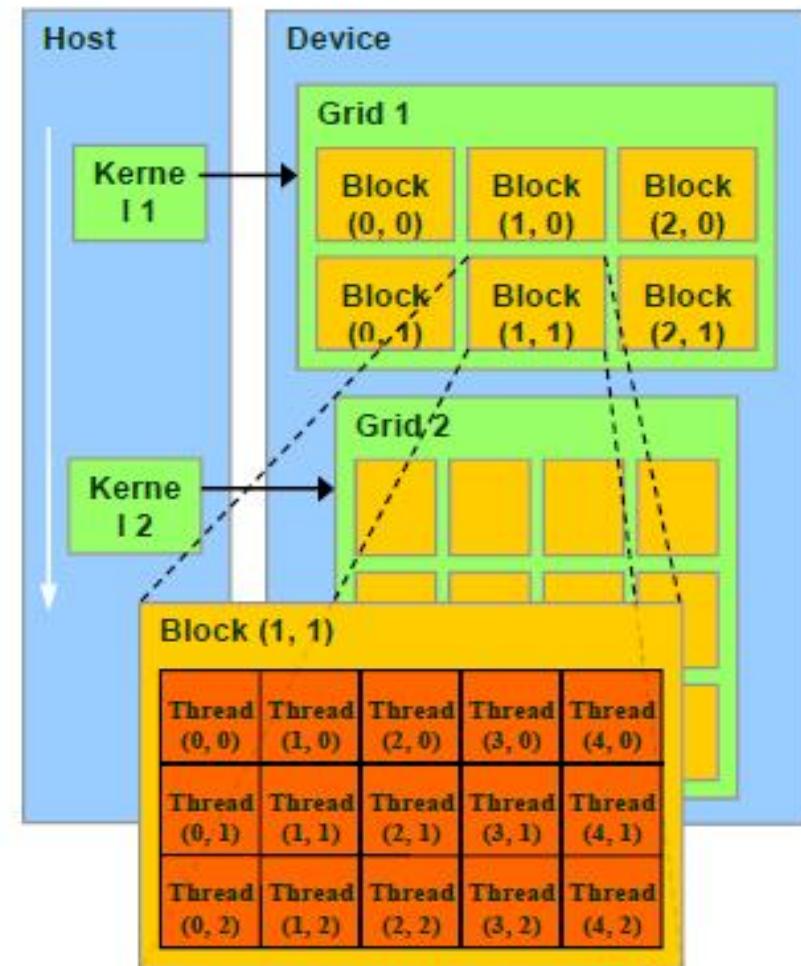
- 如何在device上并行运行代码

```
add<<< 1, 1 >>>();  
          |  
          N  
add<<< N, 1 >>>();
```

- 从执行add()函数1次，变为并行执行N次
- 通过并行运行add()，可以进行向量加法

# 线程组织模型

- 每个Kernel 函数包含非常多的线程，对应一个Grid
- 所有Thread执行相同顺序程序
- Thread被划分成线程块 Block
- 同一Block内的线程可以通过共享SM资源或同步相互协作
- 每个Thread和Block拥有唯一ID，通过内置变量读取
- 层次结构：  
**Grid → Block → Thread**

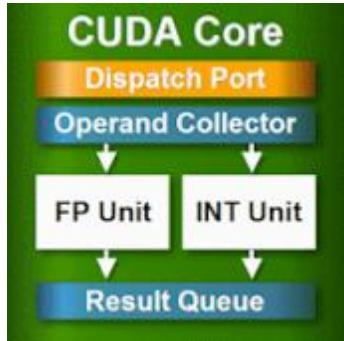


# 线程与硬件的关系

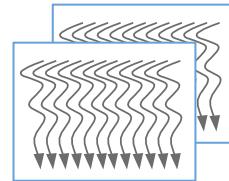
Threads



Kernel is executed by threads  
processed by CUDA Core



Blocks

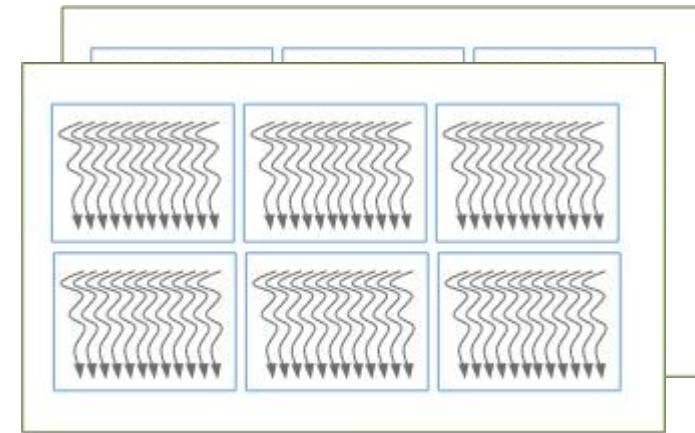


512-1024 threads / block

Maximum 8 blocks per SM  
32 parallel threads are  
executed at the same time  
in a **Warp**



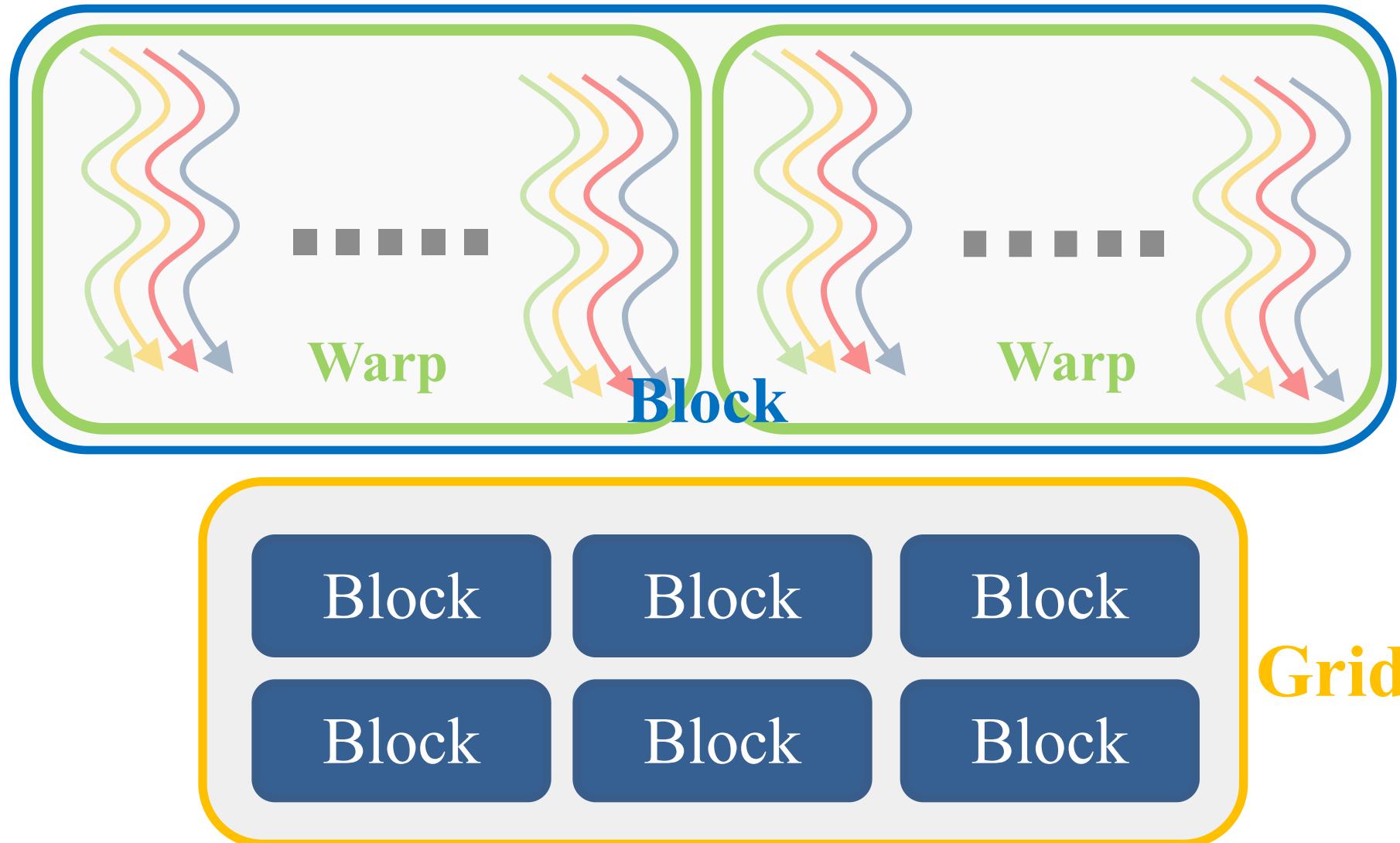
Grids



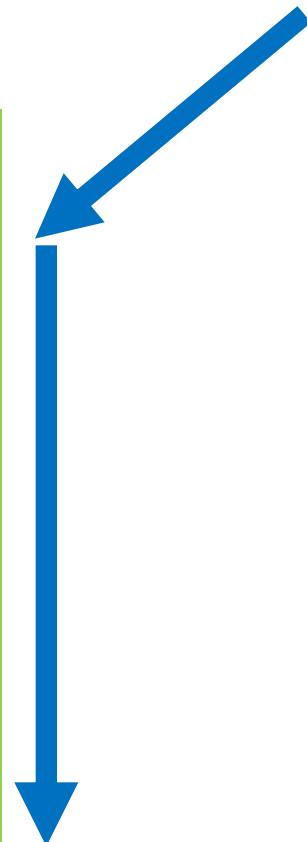
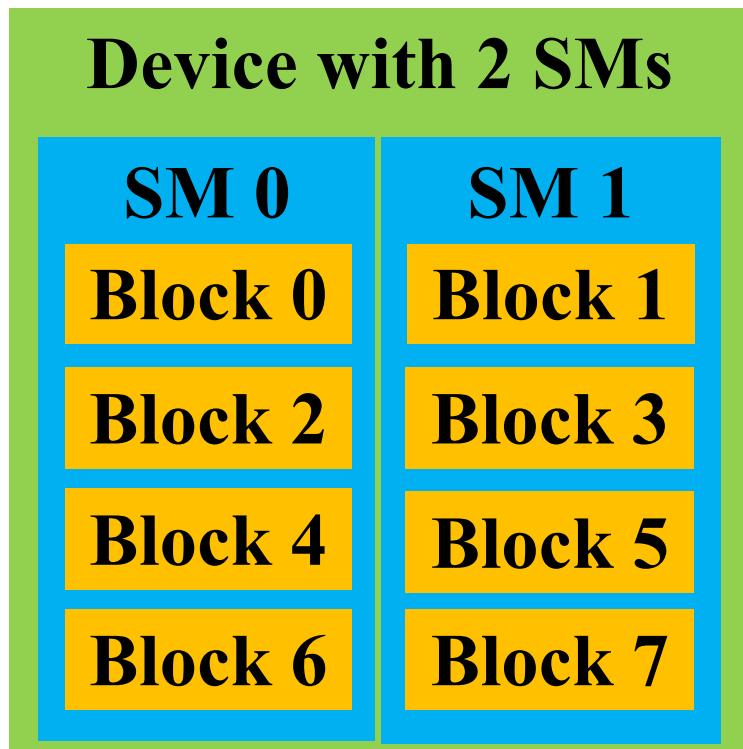
One grid per kernel with  
multiple concurrent kernels

并行计算

# 线程与硬件的关系

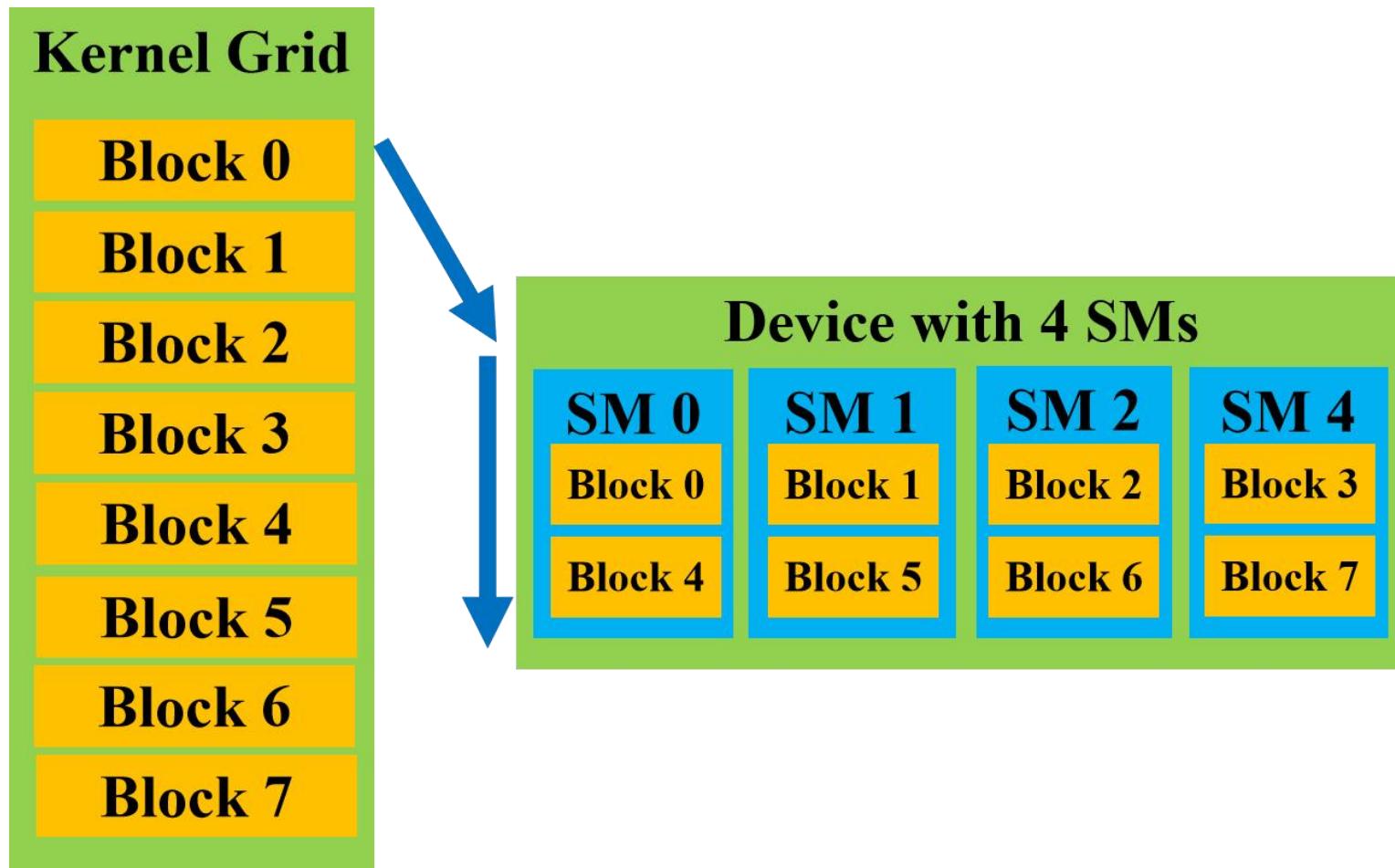


# 线程与硬件的关系

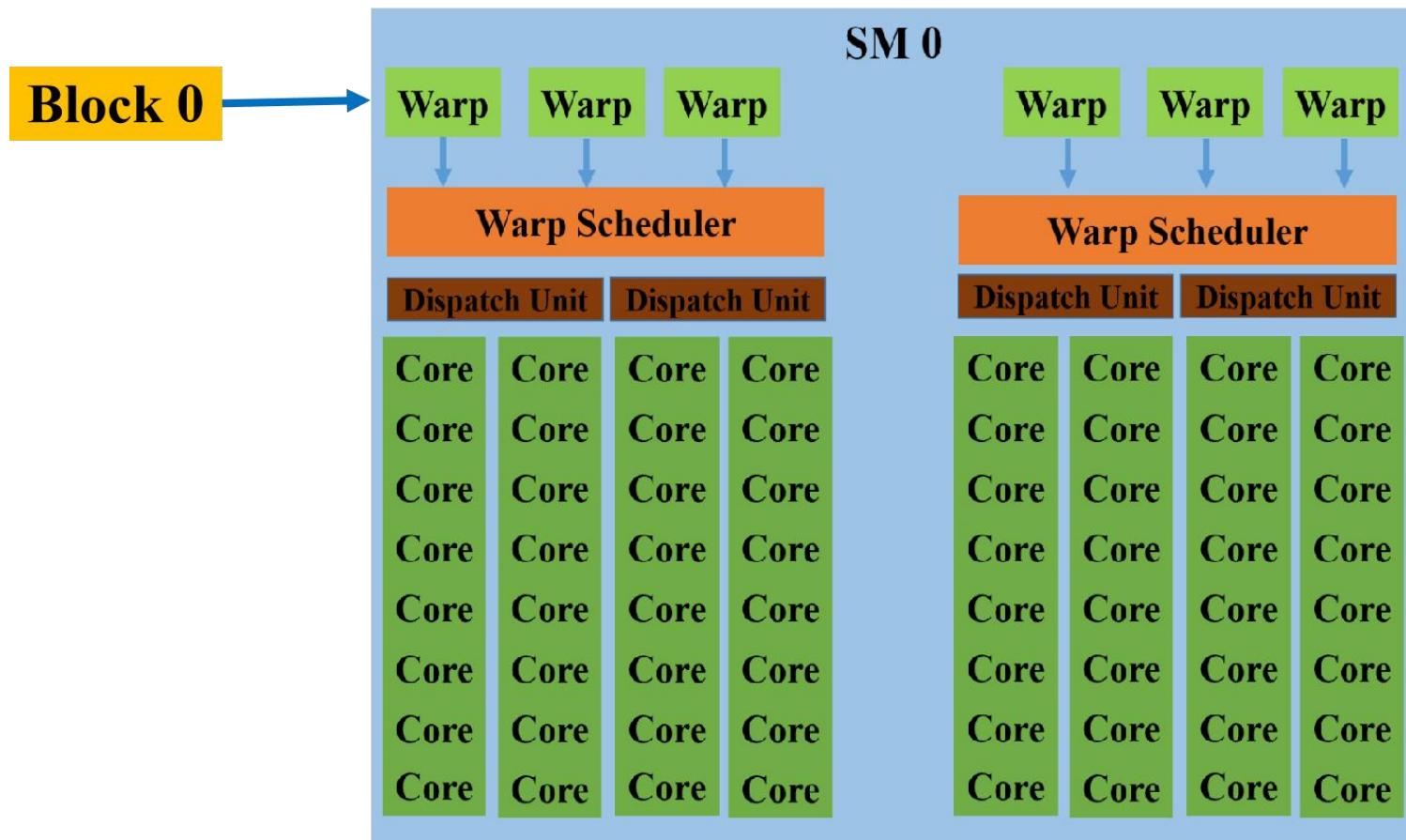


Kernel Grid
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7

# 线程与硬件的关系



# 线程与硬件的关系



# 线程与硬件的关系

GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6	7
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	2551
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
Ratio of SM Registers to FP32 Cores	341	512	1024	1024
Shared Memory Size / SM	16 KB/32 KB/ 48 KB	96 KB	64 KB	Configurable up to 96 KB

<https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>

# 线程与硬件的关系

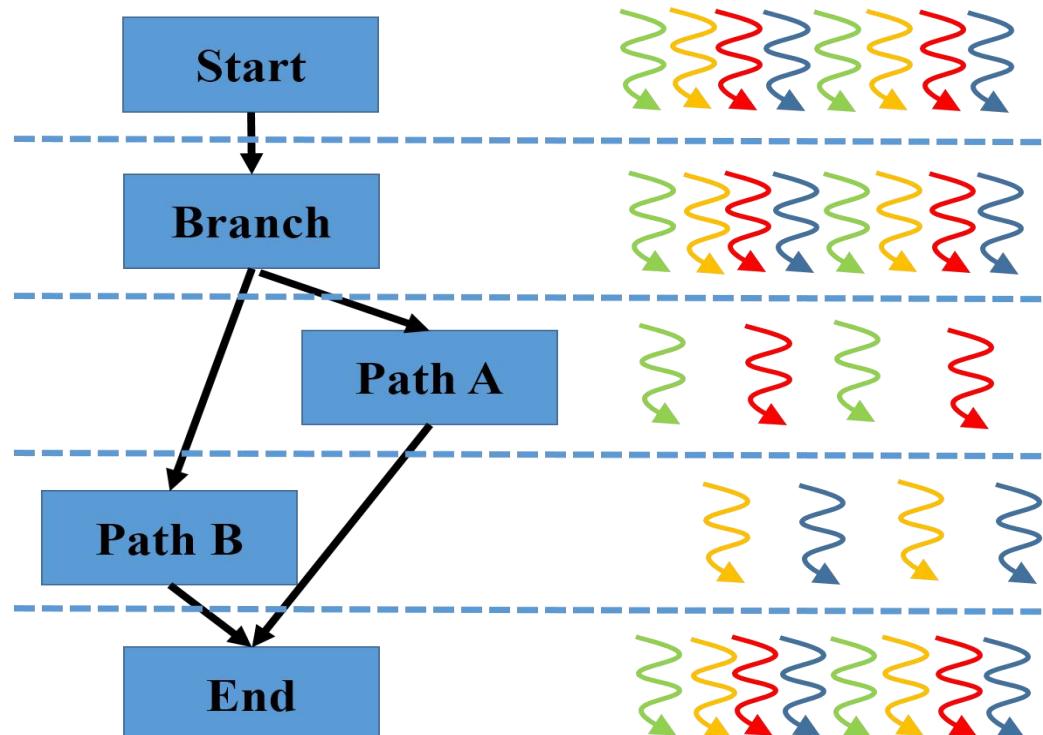
- 在任何给定时间，warp 中的所有线程都必须执行相同的指令，但这会产生分支分歧 (branch divergence)
- divergence 示例：

```
if (threadIdx.x % 2==0) { ... }  
else { ... }
```

- 为block中的线程创建两个不同的控制路径

# 线程与硬件的关系

```
if (threadIdx.x % 2==0) { ... }  
else { ... }
```



# 线程与硬件的关系

## ■ 如何避免branch divergence?

```
if (threadIdx.x / WARP_SIZE ==0) { ... }  
else { ... }
```

- 还为一个 block 中的线程创建两个不同的控制路径，但粒度是warp size的整数倍，整个 warp 做同样的工作
- 将具有相同分支行为（即执行相同分支路径）的线程放在同一个warp中，这样warp中的线程就可以并行执行，从而达到减少Branch Divergence、提高性能的目的

# Vector Addition 版本1

```
add<<< N, 1 >>>();
```

- 尖括号内部参数1代表Block数量，参数2代表每个Block中Thread数量，可以是1D、2D或3D
- 利用内置变量blockIdx.x获取Block一维索引

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- 通过使用blockIdx.x对数组进行索引，每个Block并行处理不同的索引

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```

并行计算

# Vector Addition 版本1

## ■ main() 函数

```
#define N 512

int main(void) {
    int *a, *b, *c;           // host copies of a, b, c
    int *d_a, *d_b, *d_c;     // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Vector Addition 版本1

## ■ main() 函数

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Vector Addition 版本2

```
add<<< 1, N >>>();
```

- 也可以将Block并行替换为Thread并行
- 利用内置变量`threadIdx.x`代替`blockIdx.x`

```
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

# Vector Addition 版本2

## ■ main() 函数

```
#define N 512

int main(void) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                            // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Vector Addition 版本2

## ■ main() 函数

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

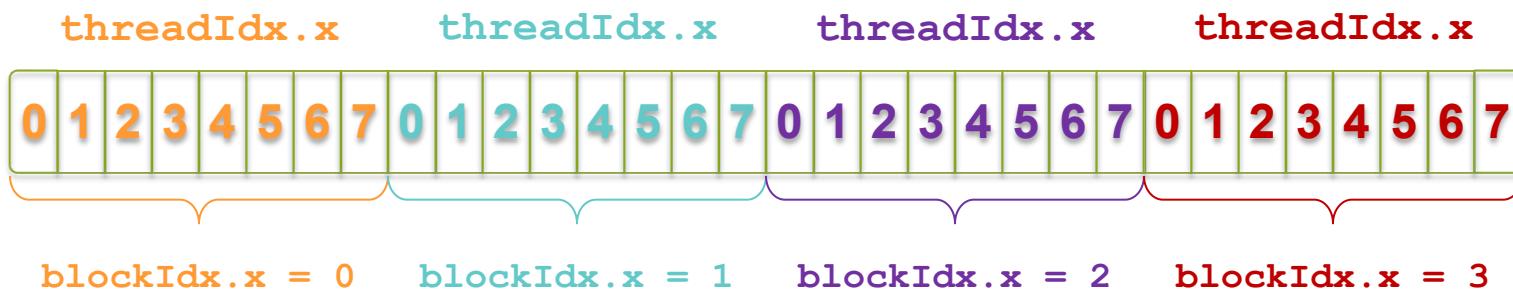
// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Vector Addition 版本3

- 硬件限制，每个Block中线程数有上限（一般为1024）
- 当N的规模变的很大时，就需要联合设置多Block和多Thread
- 不能再简单使用`blockIdx.x`或`threadIdx.x`

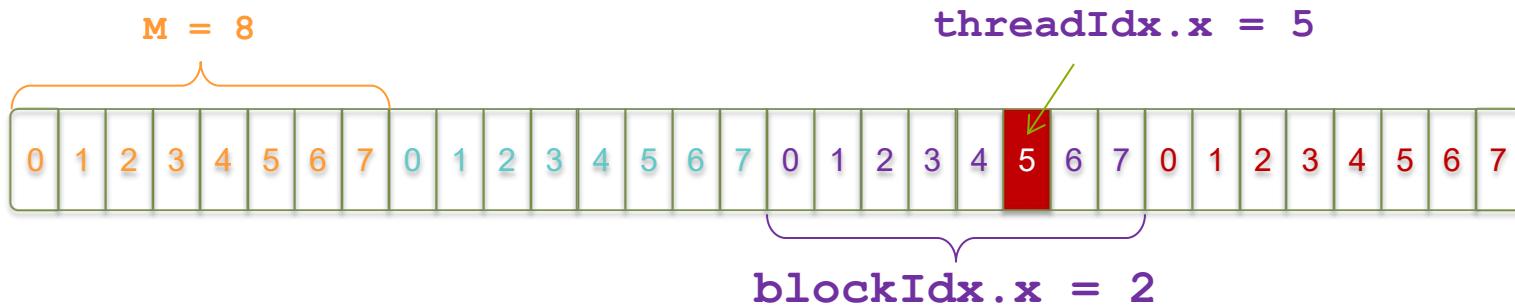


- 对于M个线程/Block，每个线程对应的数组唯一索引由下式给出：

```
int index = threadIdx.x + blockIdx.x * M;
```

# Vector Addition 版本3

■ 问题：哪个线程对红色标注的数组元素进行操作？



```
int index = threadIdx.x + blockIdx.x * M;  
= 5 + 2 * 8;  
= 21;
```

# Vector Addition 版本3

- 使用内置变量**blockDim.x**获取每个Block中的线程数

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- add() 核函数

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

# Vector Addition 版本3

## ■ main() 函数

```
#define N (1024*1024)
#define THREADS_PER_BLOCK 256

int main(void) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                            // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Vector Addition 版本3

## ■ main() 函数

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Vector Addition 版本4

- 处理任意向量大小，怎么改进？
- 避免线程的访问超出数组的界限

```
__global__ void add(int *a, int *b, int *c, int n) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    if (index < n)  
        c[index] = a[index] + b[index];  
}
```

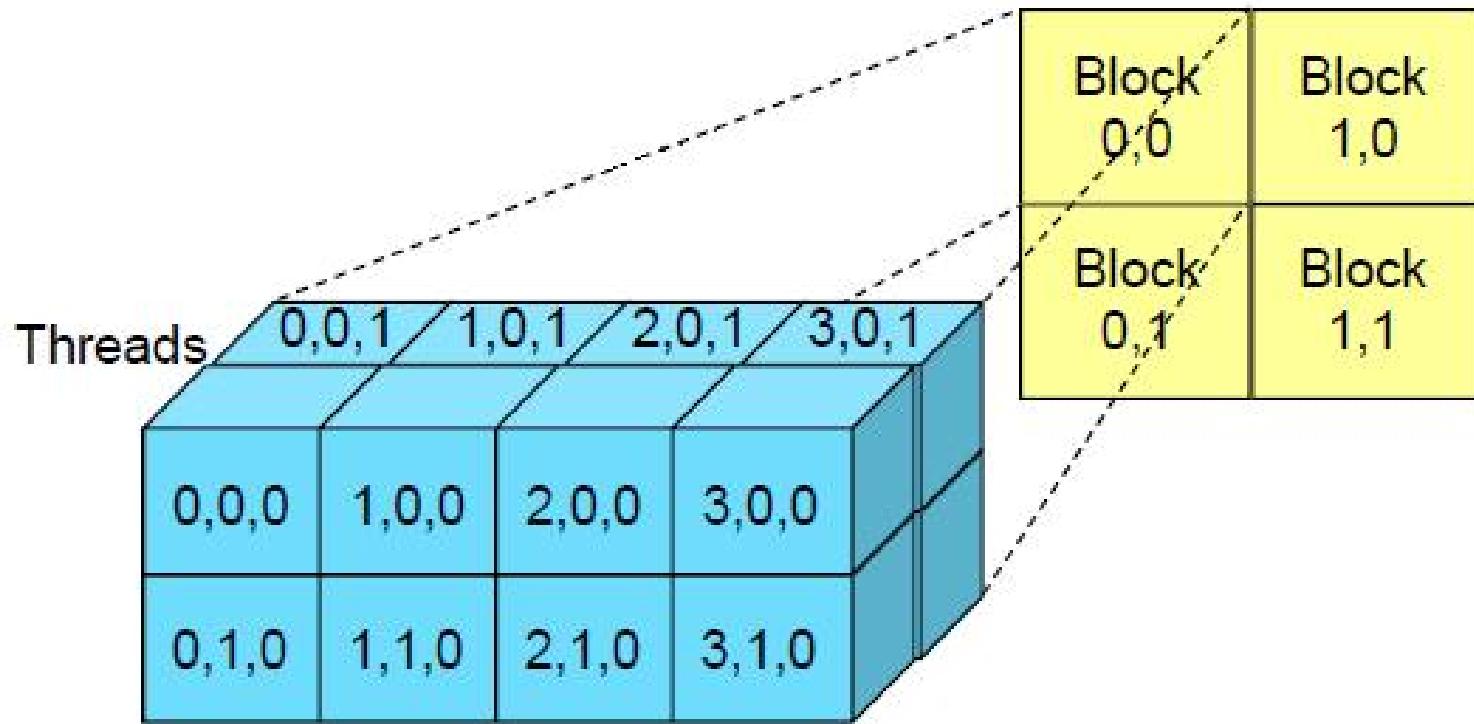
- 更新kernel启动语句

```
add<<< (N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

# 内置dim3类型

## ■ 定义grid和thread block的组织

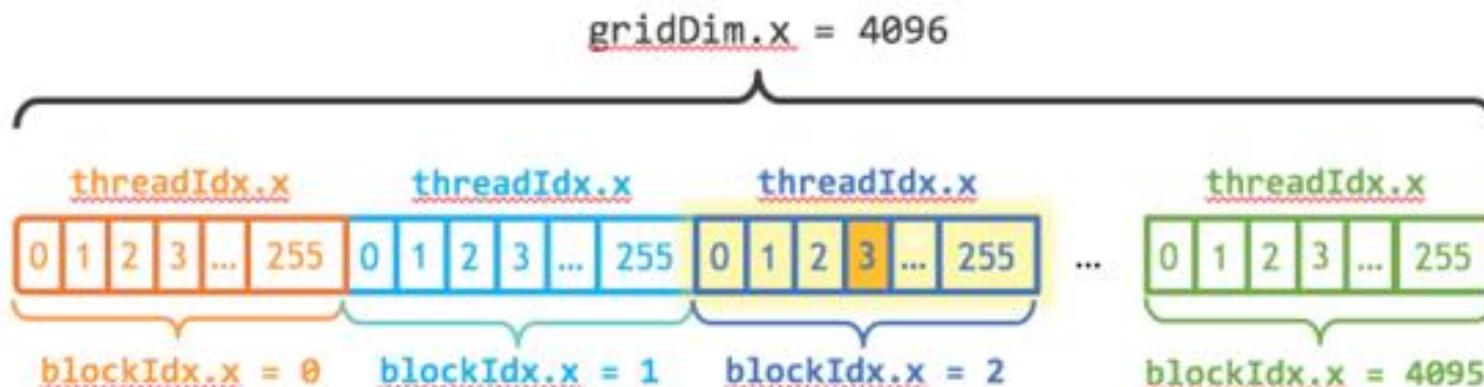
- `dim3 dimGrid(2, 2);`
- `dim3 dimBlock(4, 2, 2);`
- `kernelFunction<<< dimGrid, dimBlock>>>(...);`



# 内置索引变量

- 利用**threadIdx**确定线程块中线程的索引，即线程所在**block**中各个维度上的线程号
  - **threadIdx.x** **threadIdx.y**  
**threadIdx.z**
- 利用**blockIdx**确定网格中线程块的索引，即**block**所在**grid**中各个维度上的块号
  - **blockIdx.x** **blockIdx.y**  
**blockIdx.z**
- 利用**gridDim**确定网格的维度
  - **gridDim.x** **gridDim.y** **gridDim.z**
- 利用**blockDim**确定线程块的维度
  - **blockDim.x** **blockDim.y** **blockDim.z**

# Grids and Thread Blocks

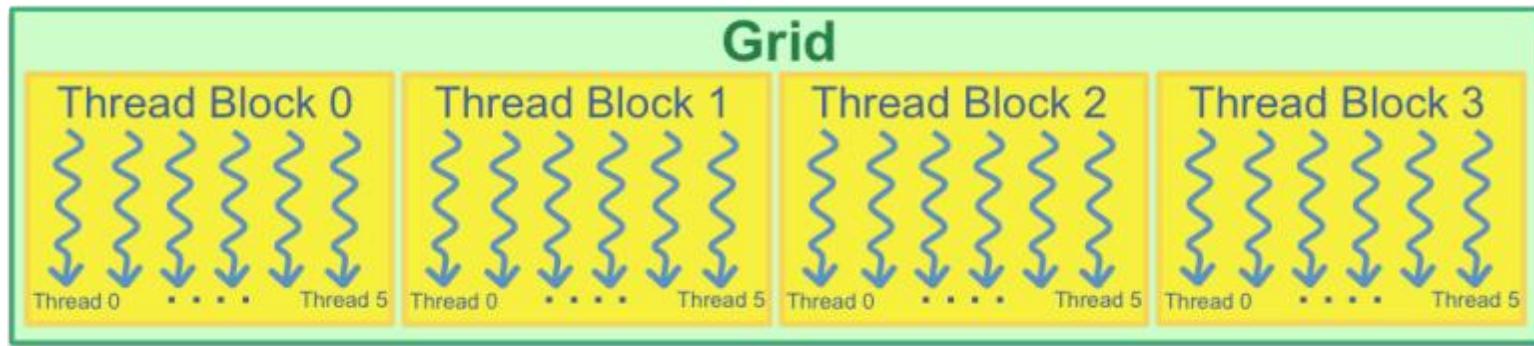


`index = blockIdx.x * blockDim.x + threadIdx.x`

$$\text{index} = (2) * (256) + (3) = 515$$

# Grids and Thread Blocks

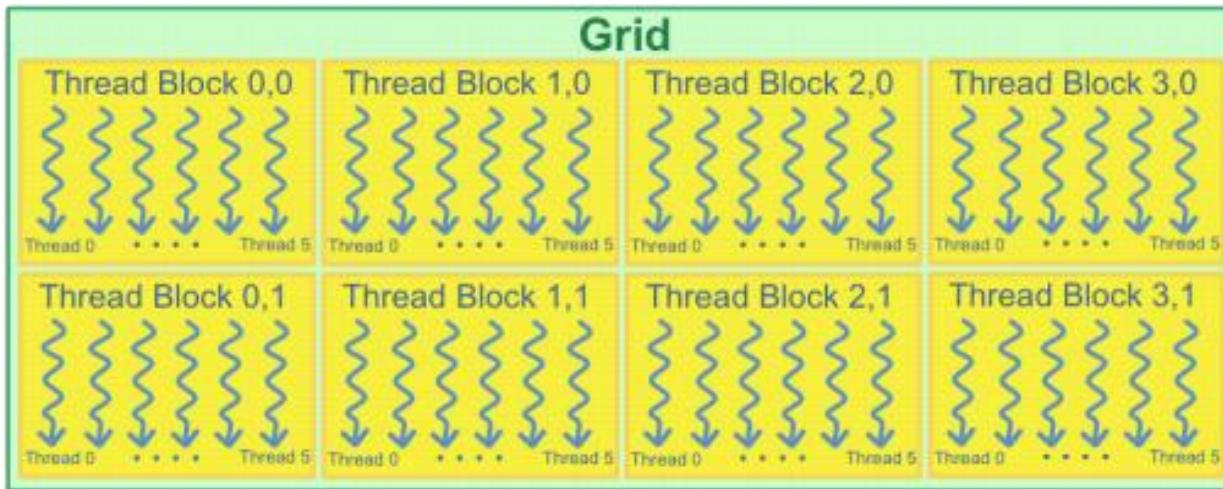
## ■ A 1D Grid of 1D Blocks



```
int threadid = blockIdx.x * blockDim.x + threadIdx.x;
```

# Grids and Thread Blocks

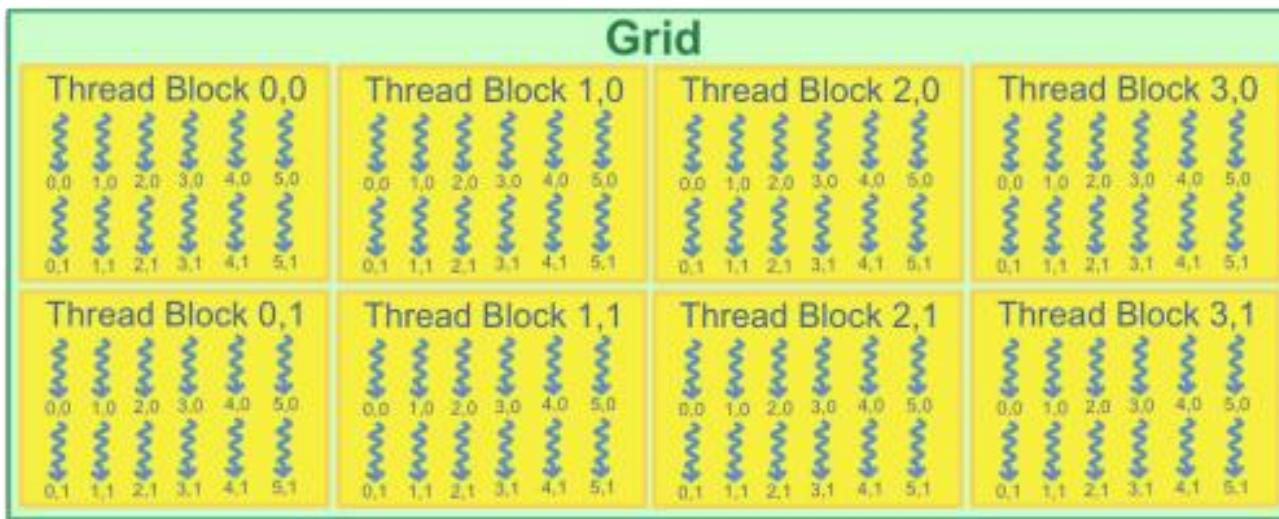
## ■ A 2D Grid of 1D Blocks



```
int blockIdx = blockIdx.y * blockDim.x + blockIdx.x;  
int threadIdx = blockIdx * blockDim.x + threadIdx.x;
```

# Grids and Thread Blocks

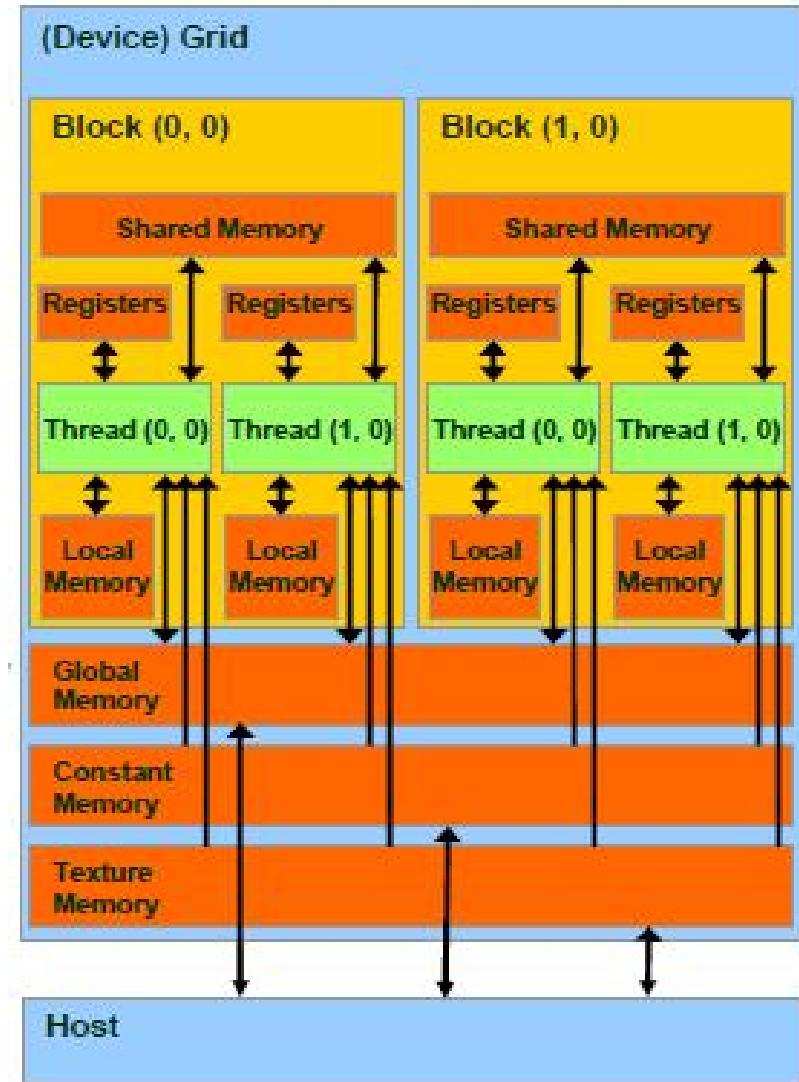
## ■ A 2D Grid of 2D Blocks



```
int blockIdx = blockIdx.x + blockIdx.y * blockDim.x;  
int threadIdx = blockIdx * (blockDim.x * blockDim.y)  
+ (threadIdx.y * blockDim.x) + threadIdx.x;
```

# GPU内存的类型

- R/W per-thread **registers**
  - 1-cycle latency
- R/W per-thread **local memory**
  - Slow – register spilling to global memory
- R/W per-block **shared memory**
  - 1-cycle latency
  - But bank conflicts may drag down
- R/W per-grid **global memory**
  - ~500-cycle latency
  - But coalescing accessing could hide latency
- Read only per-grid **constant and texture memories**
  - ~500-cycle latency
  - But cached

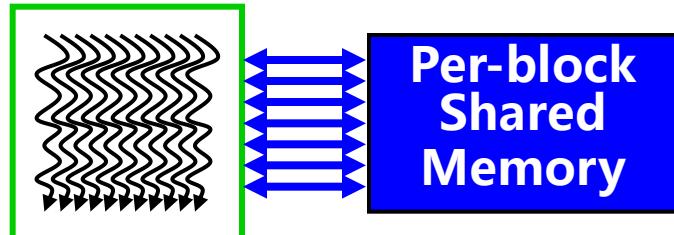


# 线程与内存的关系

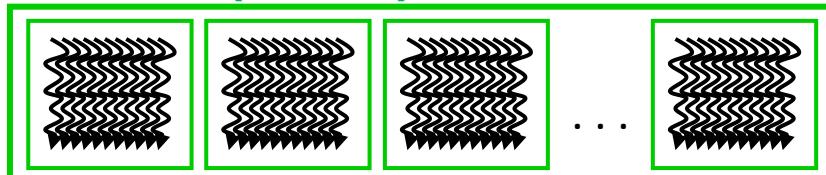
Thread



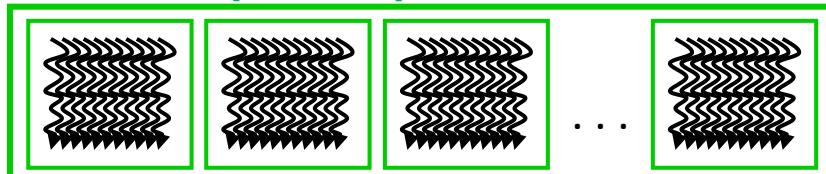
Block



Grid (kernel) 0



Grid (kernel) 1



(Device) Grid

Block (0, 0)

Shared Memory

Registers

Thread (0, 0)

Local  
Memory

Registers

Thread (1, 0)

Local  
Memory

Block (1, 0)

Shared Memory

Registers

Thread (0, 0)

Local  
Memory

Registers

Thread (1, 0)

Local  
Memory

Per-device  
Global  
Memory

Sequential  
Kernels

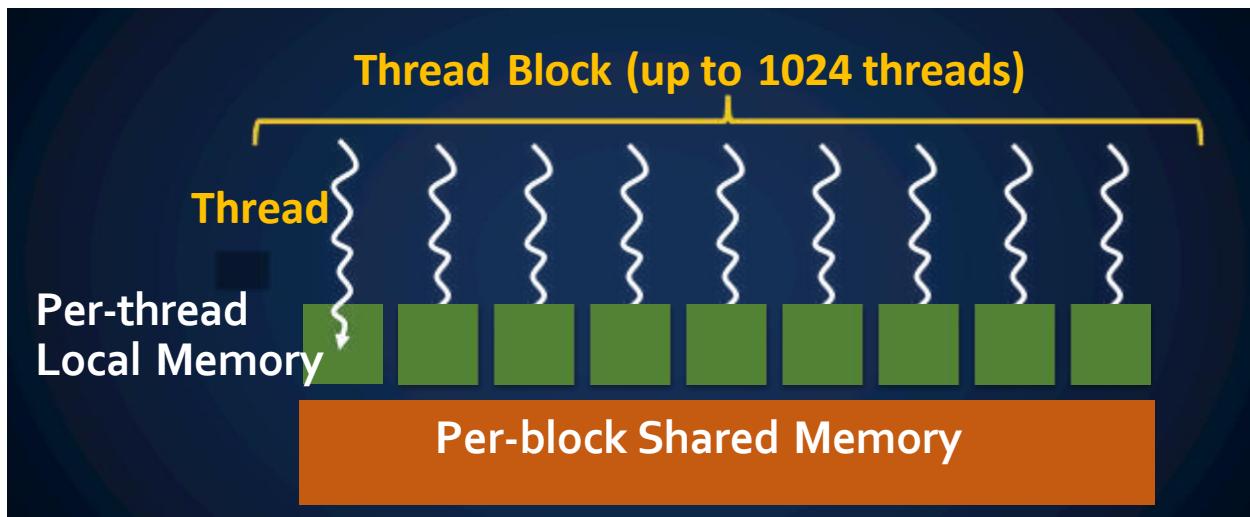
# 多层次并行策略

- 为什么同时使用线程块（blocks）和线程（threads），这种多层次并行策略？

```
kernel_call<<<blocks, threads>>>(...);
```

- 每个block的线程最大数量有限制
  - 使用单个块里面的线程无法适用于大规模的数组处理
- 同一线程块内的线程可以共享高速的shared memory
  - 线程块之间的数据共享很慢

# Shared Memory

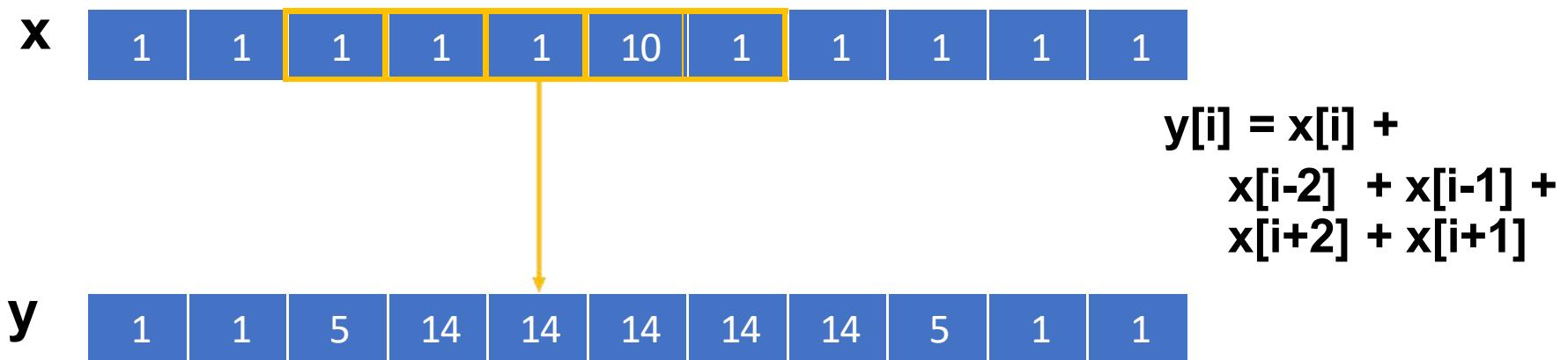


- 使用 `__shared__` 进行声明，按线程块分配
- 是快速片上存储器，用户管理
- 对其他线程块中的线程不可见，同一线程块内的线程共享

# 变量内存空间说明符

- shared， 变量储存于GPU上thread block 内的Shared Memory空间，只能被thread block内的线程存取
- device， 变量储存于GPU上的Global Memory空间，可被grid中所有线程存取
- constant， 变量储存于GPU上的Constant Memory空间，可被grid中所有线程存取
- 无修饰 (Local变量) ， 储存于SM内的寄存器和 Local Memory, Thread私有

# 1D Stencil



**1D 5-point stencil (with a “radius” of 2)**

# 1D Stencil

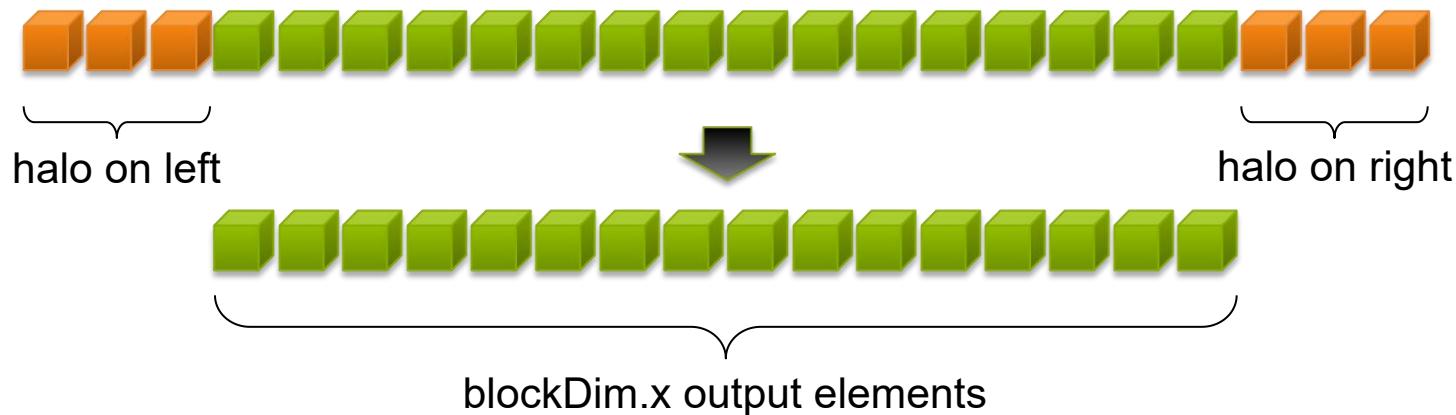


- 每个线程处理一个输出元素
  - 每个block处理blockDim.x个元素
- 输入元素被读取多次
- 输入数据的重复使用情况：
  - radius为2， 每个输入元素读取5次
  - radius为3， 每个输入元素读取7次



# 1D Stencil

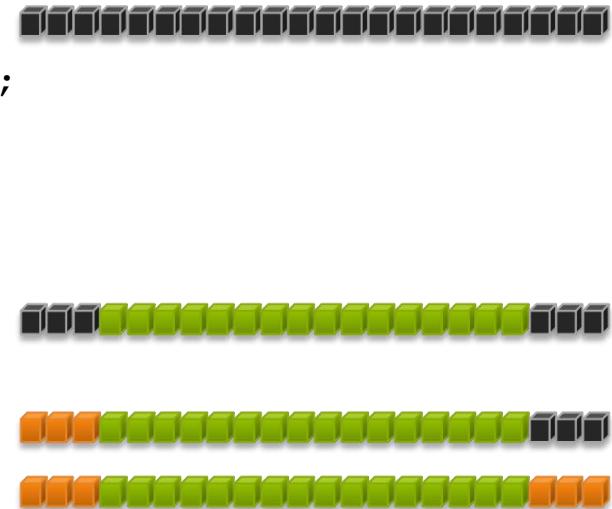
- 策略：利用shared memory缓存数据
- 从global memory 读取  
(`blockDim.x+2*radius`) 个输入元素到  
shared memory
- 计算`blockDim.x`输出元素
- 将`blockDim.x`输出元素写入global memory



# 1D Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] =
            in[gindex + BLOCK_SIZE];
    }
}
```



# 1D Stencil Kernel

```
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
```

# 1D Stencil

- 上述Stencil Kernel有什么问题？
- 假设线程15在线程0获取halo之前读取halo...
- 数据竞争 (Data Race) !

```
temp[lindex] = in[gindex];           Store at temp[18]   
if (threadIdx.x < RADIUS) {  
    temp[lindex - RADIUS] = in[gindex - RADIUS];   Skipped, threadIdx > RADIUS  
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
}  
  
int result = 0;  
result += temp[lindex + 1];           Load from temp[19] 
```

# 线程同步

```
void __syncthreads();
```

- 同步一个线程块（Block）中的所有线程
- 用于防止RAW/WAR/WAW问题
- 保证所有线程必须到达barrier
  - 在条件代码中，条件在整个块中必须是一致的

```
if (threadIdx.x < 10)
{
    __syncthreads();
}
```



# 线程同步

void \_\_syncthreads();

- 同步一个线程块（Block）中的所有线程
- 用于防止RAW/WAR/WAW问题
- 保证所有线程必须到达barrier
  - 在条件代码中，条件在整个块中必须是一致的
- kernel与kernel之间有隐式barrier

---

vec\_minus<<<nblocks, blksize>>>(a, b, c);

---

vec\_dot<<<nblocks, blksize>>>(c, c);

# 1D Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
}
```

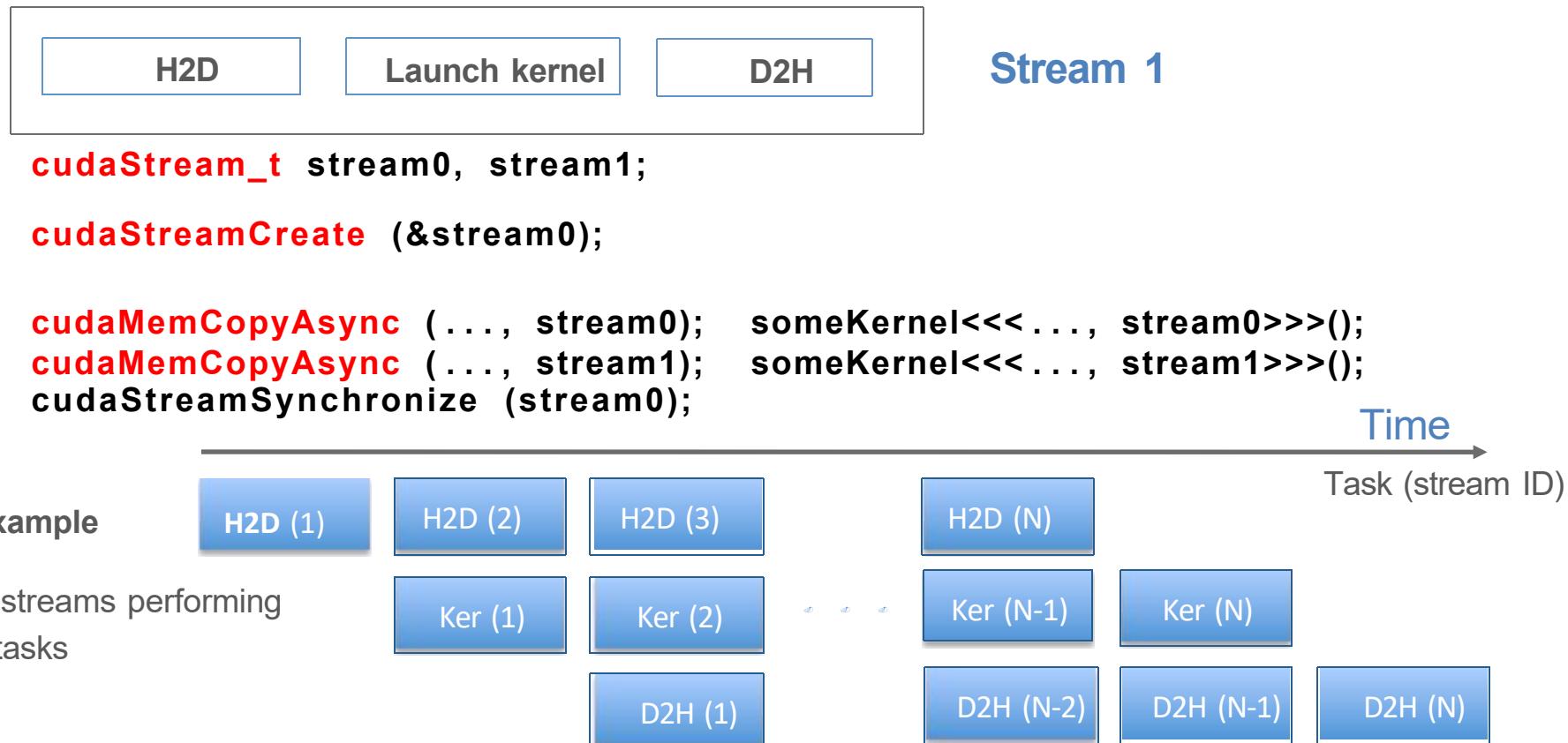
# 1D Stencil Kernel

```
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
```

# CUDA Stream

- Stream, 定义为按顺序执行的设备操作序列，包括 kernel 执行和数据传输



# CUDA Stream

- Stream, 定义为按顺序执行的设备操作序列，包括 kernel 执行和数据传输

- **Serial**

串行

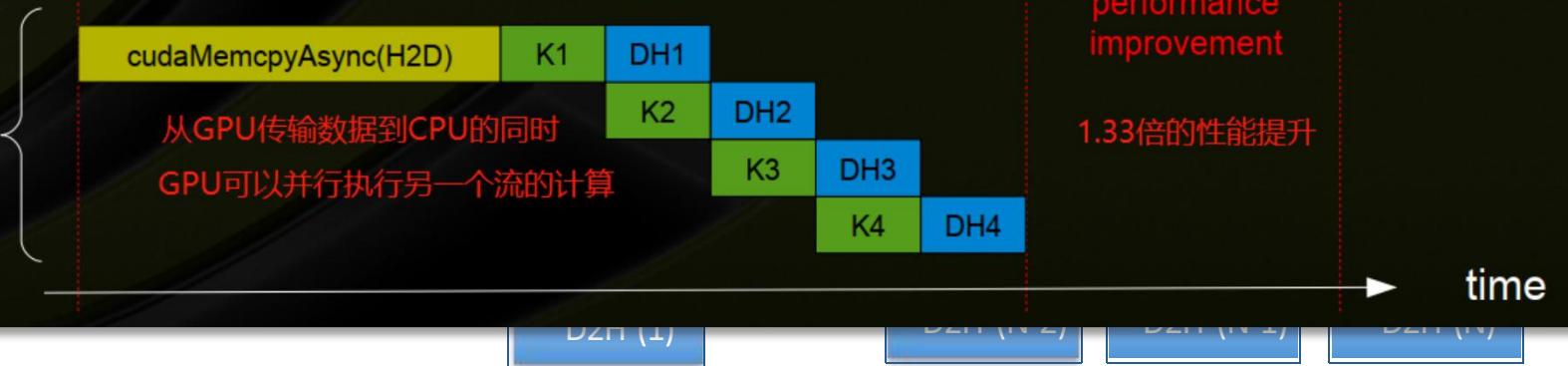


- **Concurrent – overlap kernel and D2H copy**

并行

**streams**

流



# 获取GPU信息

```
#include <stdio.h>
int main() {
    int noOfDevices;
    /*get no. of device*/
    cudaGetDeviceCount(&noOfDevices);
    cudaDeviceProp prop;
    for(int i = 0;i<noOfDevices;i++) {
        /*get device properties */
        cudaGetDeviceProperties(&prop,i);
        printf("Device Name:\t%s\n",prop.name);
        printf("Total global memory:\t%ld\n",
               prop.totalGlobalMem);
        printf ("No . of SMs:\t %d\n",
               prop.multiProcessorCount);
        printf ("Shared memory / SM:\t %ld\n",
               prop.sharedMemPerBlock);
        printf("Registers / SM:\t %d\n",
               prop.regsPerBlock);
    }
    return 0;
}
```

**CUDA函数:**  
**cudaGetDeviceCount**  
**cudaGetDeviceProperties**

编译

```
nvcc whatDevice.cu -o whatDevice
```

输出

Device Name:	Tesla C2050
Total global memory:	2817720320
No. of SMs:	14
Shared memory / SM:	49152
Registers / SM:	32768

For more properties see  
**struct cudaDeviceProp**

For details see CUDA Reference Manual

# CUDA计时

```
int main() {  
    cudaEvent_t start, stop;  
    float time;  
    cudaEventCreate(&start);  
    cudaEventCreate(&stop);  
    cudaEventRecord(start,0);  
  
    someKernel<<<grid,blocks,0,0>>>(...);  
  
    cudaEventRecord(stop,0);  
    cudaEventSynchronize(stop); ← Ensures kernel execution has completed  
  
    cudaEventElapsedTime(&time,start,stop);  
    cudaEventDestroy(start);  
    cudaEventDestroy(stop);  
    printf("Elapsed time %f sec\n",time*.001);  
    return 0;  
}
```

**CUDA Event API Timer are,**  
**- OS independent**  
**- High resolution**  
**- Useful for timing asynchronous calls**

**Standard CPU timers will not measure the timing information of the device.**

# 其他

- Constant Memory
- Texture Memory
- Graphics Interoperability
- Atomics
- Streams
- CUDA C On Multiple GPUs

官方指南: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-interface>

# Reference

- CUDA C++ Programming Guide, <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-interface>
- CUDA Toolkit Documentation, <https://docs.nvidia.com/cuda/index.html>
- Blaise Barney, Lawrence Livermore National Laboratory, Introduction to Parallel Computing, [https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)



# 第七章 并行算法及性能分析

哈尔滨工业大学

张伟哲

2025, Fall Semester

# 目录

- 通信开销模型
- 不同拓扑的通信开销
- 矩阵-向量相乘

# 目录

- 通信开销模型
- 不同拓扑的通信开销
- 矩阵-向量相乘

# 通信开销

- 在并行程序中，除了空闲 (idling) 和争用 (contention) 之外，通信 (communication) 也是主要的开销
- 通信成本取决于各种特征，包括编程模型语义 (programming model semantics)、网络拓扑 (network topology)、数据处理 (data handling) 和路由 (routing) 以及相关的软件协议 (software protocols)

# 消息传递开销

- 通过网络传输信息的总时间包含以下时间：
- 启动时间 (Startup time)  $t_s$ : 在发送和接收节点所花费的时间 (执行路由算法、编程路由器等)
- 每跳时间 (Per-hop time)  $t_h$ : 跳数的函数，包括交换机延迟、网络延迟等因素
- 每字传输时间 (Per-word transfer time)  $t_w$ : 由消息长度决定的所有开销，包括链路带宽、错误检查和纠正等

# 存储转发路由

- 存储转发路由 (Store-and-Forward Routing)
- 信息被发送到一个中间站，被保存起来，并在以后发送到最终目的地或另一个中间站
- 一条大小为  $m$  个字的消息通过  $l$  个通信链路 (communication links) 的总通信成本为

$$t_{comm} = t_s + (mt_w + t_h)l.$$

- 在大多数平台上， $t_h$  很小，上面的表达式可以近似为

$$t_{comm} = t_s + mlt_w.$$

# 直通路由

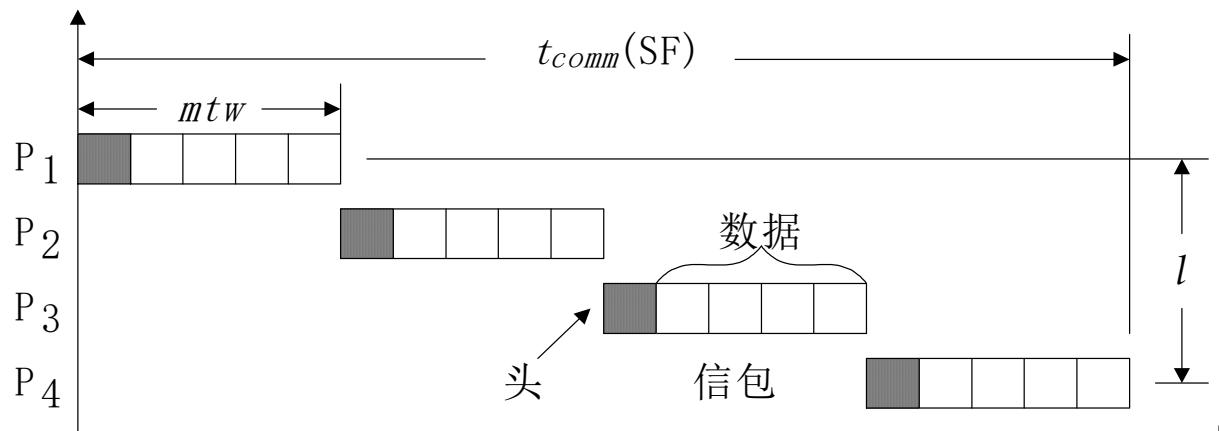
- 直通路由 (Cut-Through Routing)
- 在传递一个消息之前，就为它建立一条从源结点到目的结点的物理通道。在传递的全部过程中，线路的每一段都被占用，当消息的尾部经过网络后，整条物理链路才被废弃
- 总通信时间近似为：

$$t_{comm} = t_s + t_h l + t_w m.$$

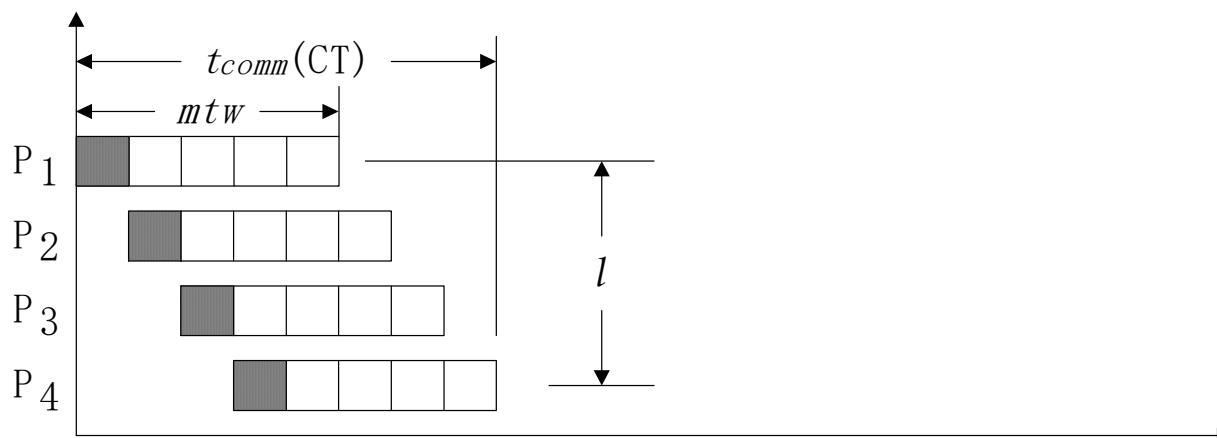
- $t_h$ 通常远小于 $t_s$ 和 $t_w$ 的，总通信时间进一步简化

$$t_{comm} = t_s + t_w m.$$

# SF和CT模式的时空图



(a)



(b)

# 目录

- 通信开销模型
- 不同拓扑的通信开销
- 矩阵-向量相乘

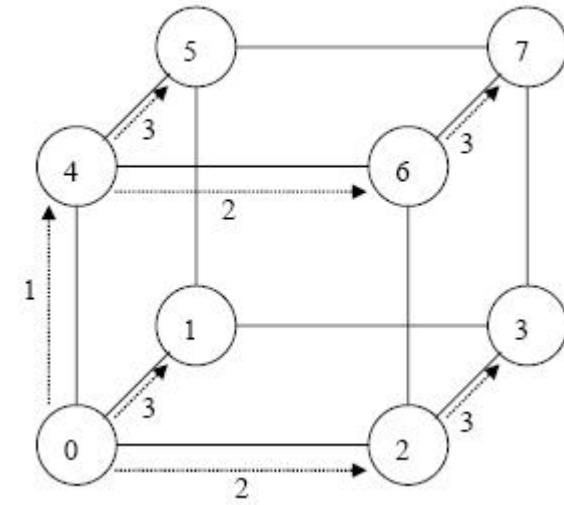
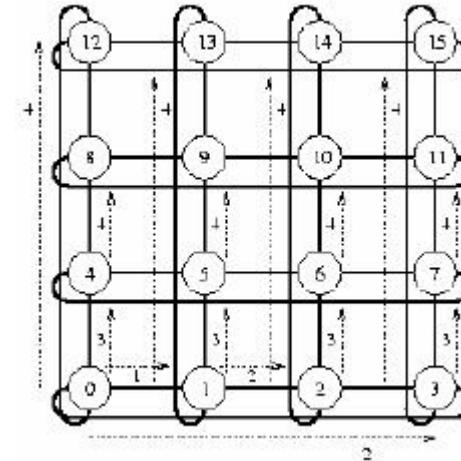
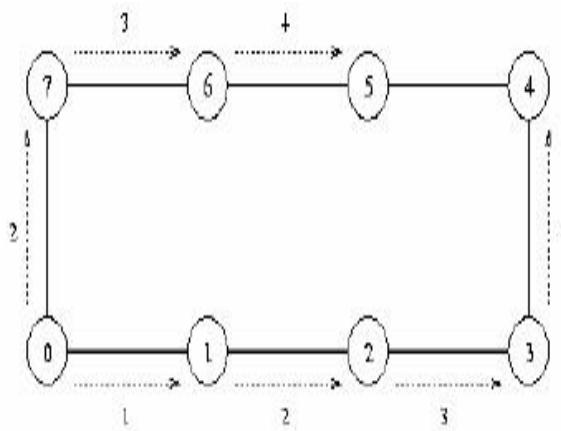
# 点对点通信

## ■ 距离 $l$ 的计算：对于 $p$ 个处理器

■ 一维环形： $l \leq \lfloor p/2 \rfloor$

■ 带环绕Mesh： $l \leq 2\lfloor \sqrt{p}/2 \rfloor$

■ 超立方： $l \leq \log p$



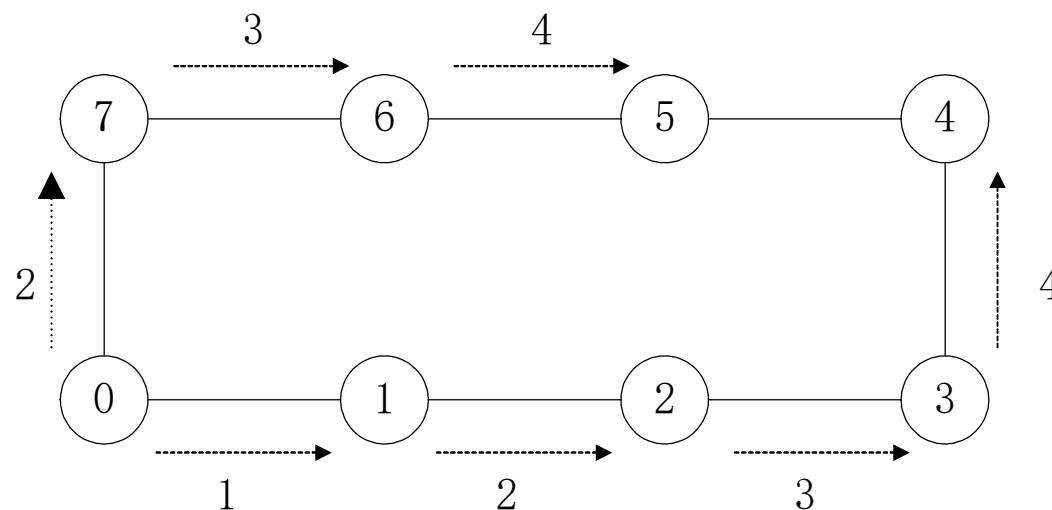
# 点对点通信

## ■ 两个网络处理器之间的数据通信

拓扑	存储转发路由 Store-and-Forward Routing	直通路由 Cutting-Through
环 (Ring)	$t_s + mt_w \lfloor p/2 \rfloor$	$t_s + mt_w$
网格 (Grid—torus)	$t_s + 2mt_w \lfloor \sqrt{p}/2 \rfloor$	$t_s + mt_w$
超立方 (Hypercube)	$t_s + mt_w \log_2 p$	$t_s + mt_w$

# One-to-All Broadcast

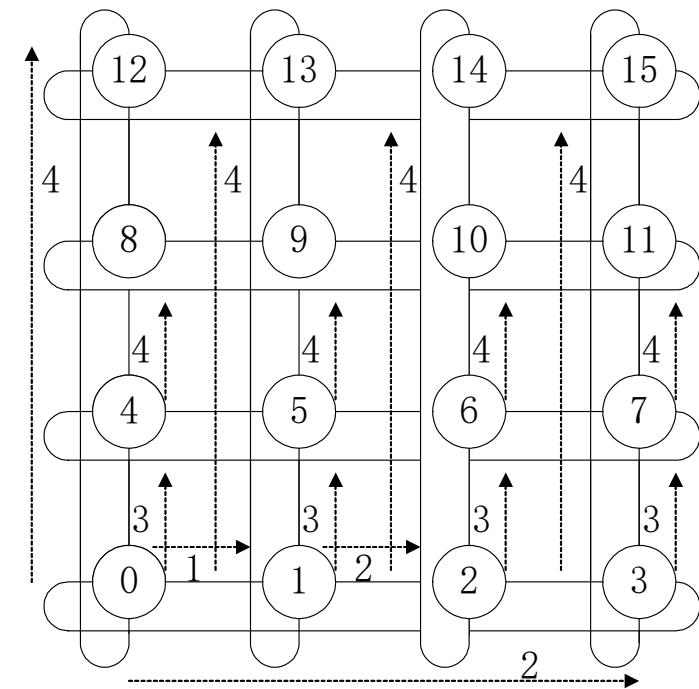
- 一到全广播，SF模式，环
- 步骤：①先左右邻近传送；②再左右二个方向同时播送
- 示例：



- 通信时间： $t_{one-to-all}(SF) = (t_s + mt_w)\lceil p / 2 \rceil$

# One-to-All Broadcast

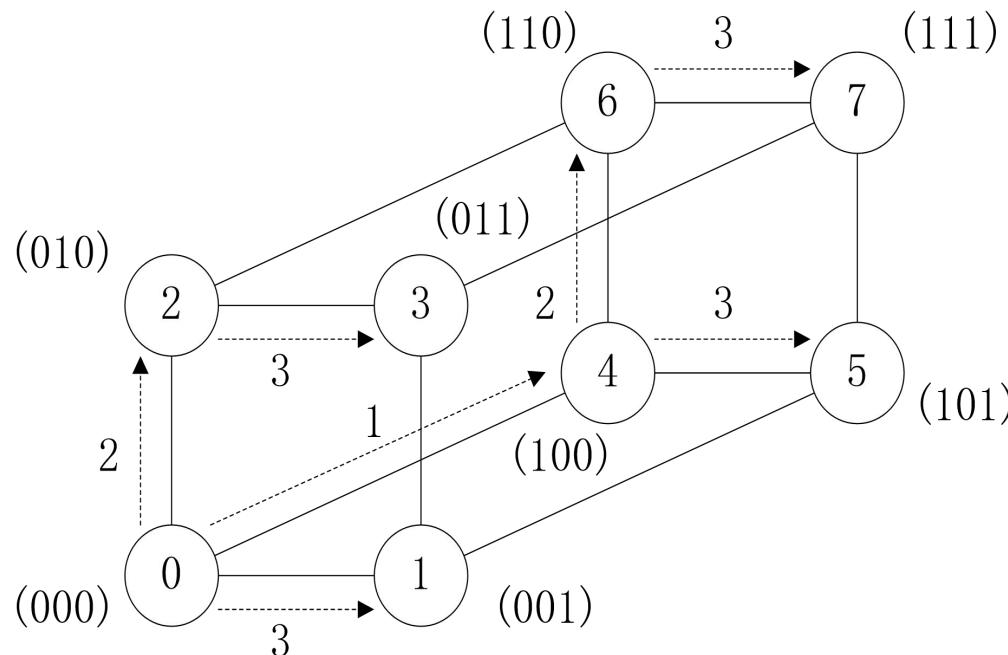
- 一到全广播，SF模式，环绕网孔
- 步骤：①先完成一行中的播送；②再同时进行各列的播送
- 示例：共4步(2步行、2步列)



- 通信时间： $t_{one-to-all}(SF) = 2(t_s + mt_w) \left\lceil \frac{\sqrt{p}}{2} \right\rceil$

# One-to-All Broadcast

- 一到全广播，SF模式，超立方
- 步骤：从低维到高维，依次进行播送；
- 示例：



- 通信时间： $t_{one-to-all}(SF) = (t_s + mt_w) \log p$

# One-to-All Broadcast

■ 一到全广播，CT模式，  
环

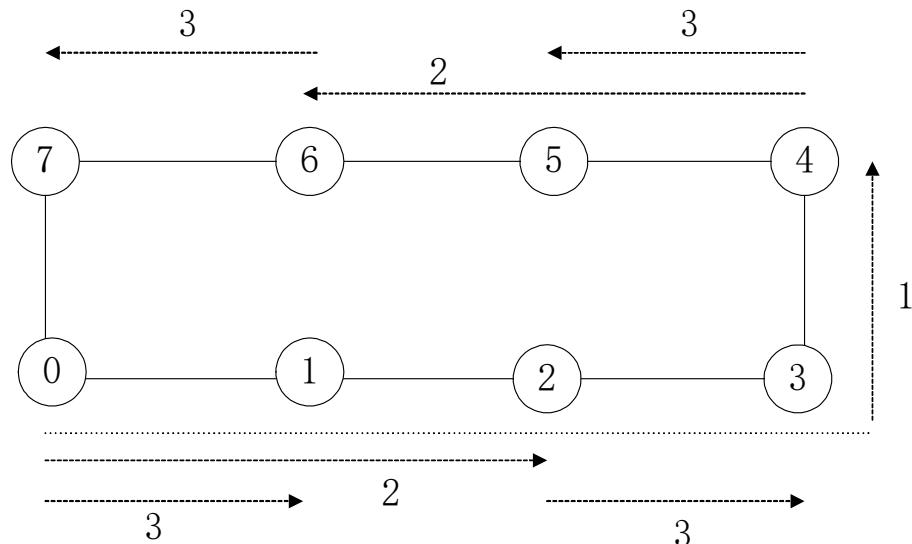
■ 步骤：

- (1) 先发送至  $p/2$  远的处理器；
- (2) 再同时发送至  $p/2^2$  远的处理器；

.....

- (i) 再同时发送至  $p/2^i$  远的处理器；

■ 通讯时间： 
$$\begin{aligned} t_{one-to-all}(CT) &= \sum_{i=1}^{\log p} (t_s + mt_w + t_h p / 2^i) \\ &= t_s \log p + mt_w \log p + t_h(p-1) \\ &\approx (t_s + mt_w) \log p \quad (t_h \text{ 可忽略时 }) \end{aligned}$$



# One-to-All Broadcast

- 一到全广播，CT模式，网孔

- 步骤：

- (1) 先进行行播送；

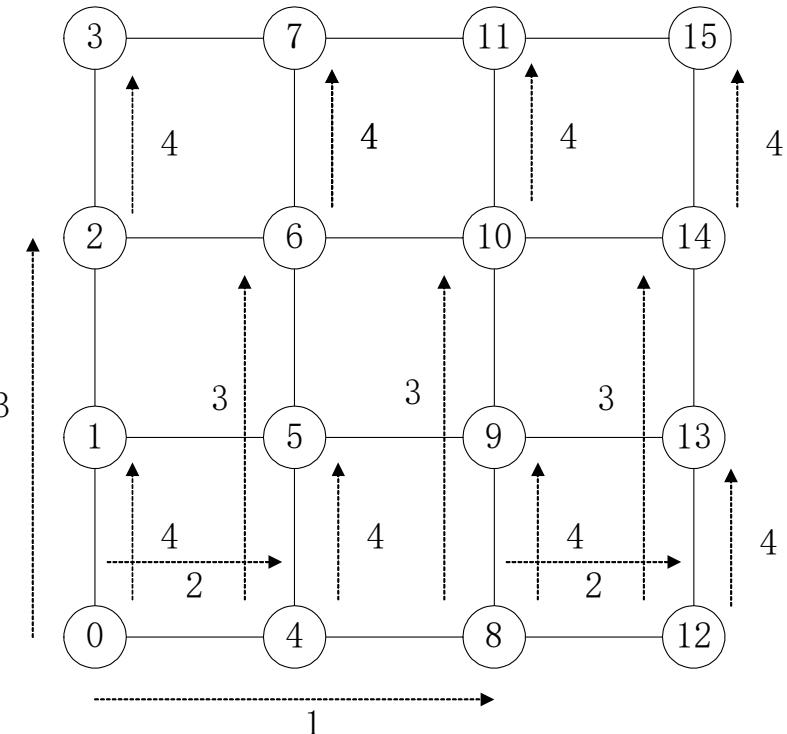
$$\text{// } t_s \log \sqrt{p} + mt_w \log \sqrt{p} + t_h(\sqrt{p} - 1)$$

- (2) 再同时进行列播送；

$$\text{// } t_s \log \sqrt{p} + mt_w \log \sqrt{p} + t_h(\sqrt{p} - 1)$$

- 通信时间：

$$\begin{aligned} t_{one-to-all}(CT) &= 2(t_s \log \sqrt{p} + mt_w \log \sqrt{p} + t_h(\sqrt{p} - 1)) \\ &= (t_s + mt_w) \log p + 2t_h(\sqrt{p} - 1) \end{aligned}$$



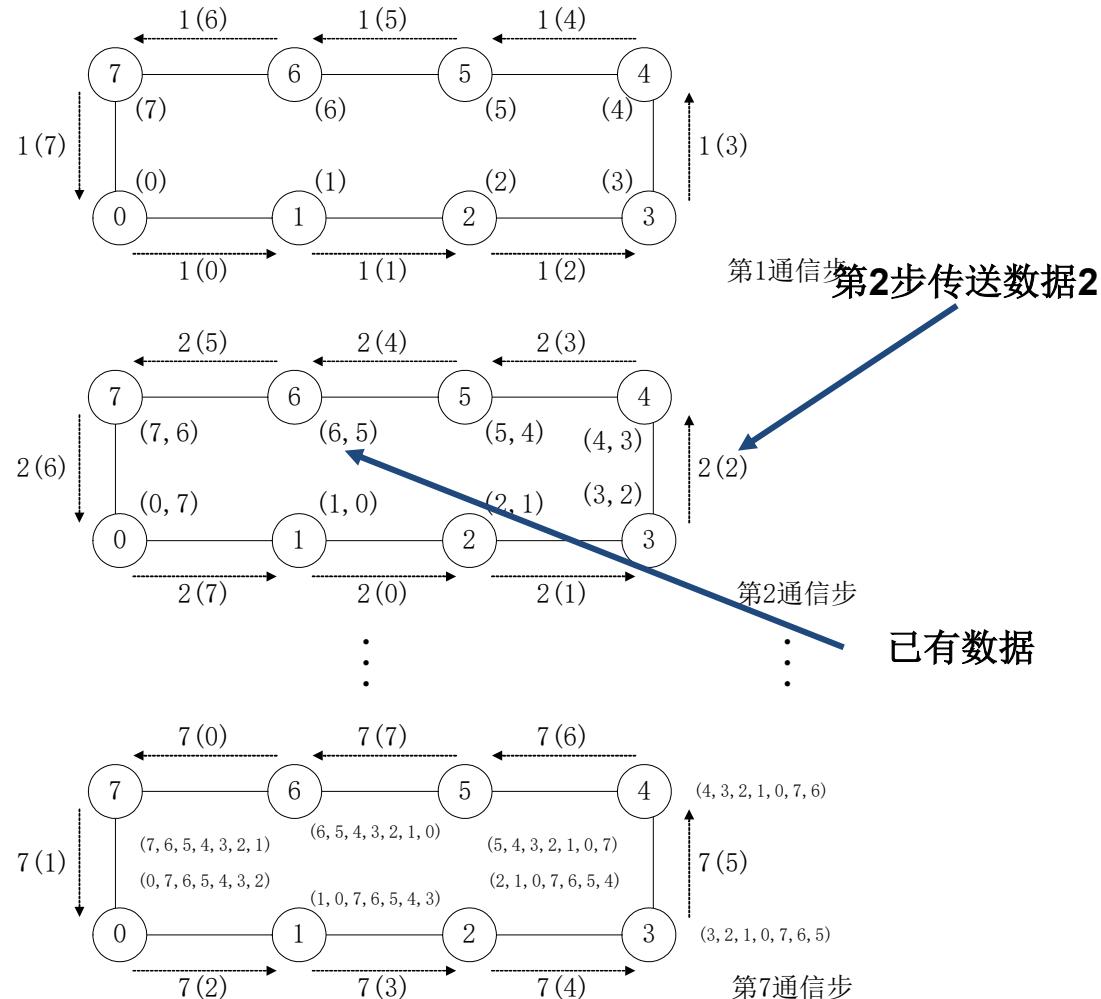
# One-to-All Broadcast

- 一到全广播, CT模式, 超立方
- 步骤: 依次从低维到高维播送, d-立方,  
 $d=0,1,2,3,4\dots$ ;
- 通信时间:

$$t_{one-to-all}(CT) = (t_s + mt_w) \log p$$

# All-to-All Broadcast

- 全到全广播，SF模式，环
- 步骤：同时向右(或左)播送刚接收到的信包



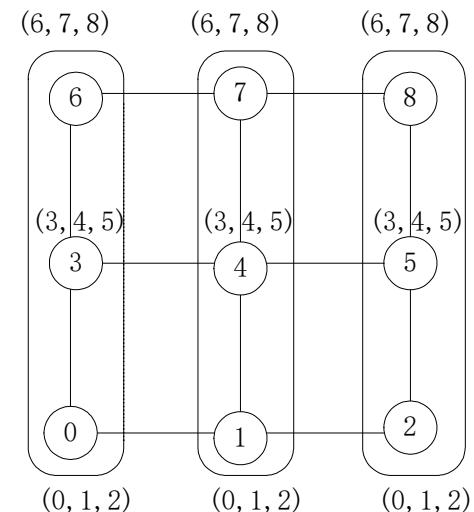
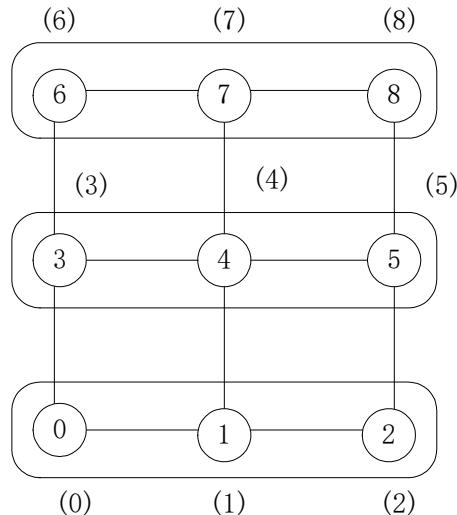
# All-to-All Broadcast

■ 全到全广播，SF模式，环绕网孔

■ 步骤：

- (1) 先进行行的播送；
- (2) 再进行列的播送；

■ 通信时间：



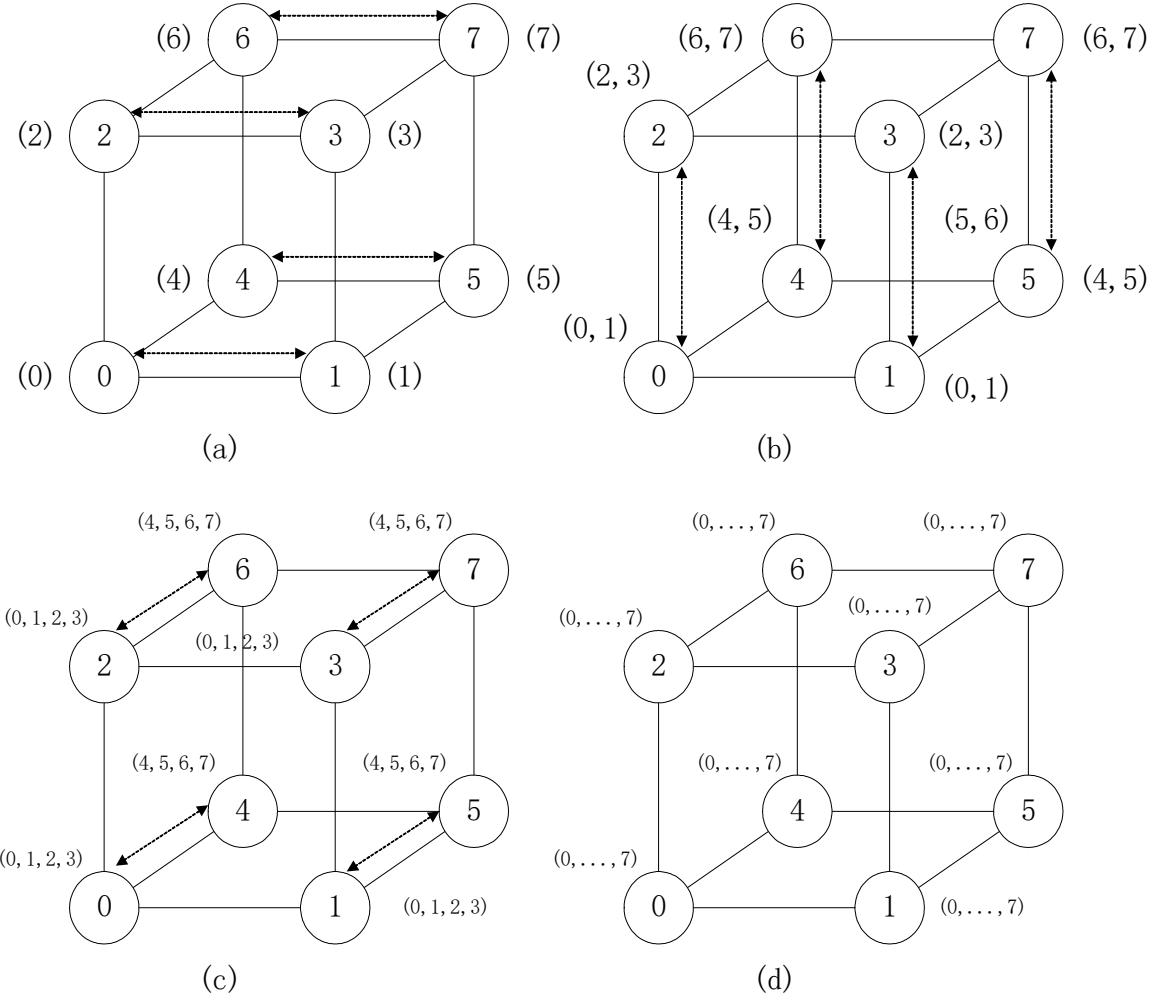
$$\begin{aligned}t_{all-to-all}(SF) &= (t_s + mt_w)(\sqrt{p} - 1) + (t_s + m\sqrt{p} \cdot t_w)(\sqrt{p} - 1) \\&= 2t_s(\sqrt{p} - 1) + mt_w(p - 1)\end{aligned}$$

# All-to-All Broadcast

- 全到全广播，SF模式，超立方体
- 步骤：依次按维进行，多到多的播送；

## ■ 通信时间：

$$\begin{aligned} t_{all-to-all}(SF) &= \sum_{i=1}^{\log p} (t_s + 2^{i-1} m t_w) \\ &= t_s \log p + m t_w (p-1) \end{aligned}$$



并行计算

# All-to-All Broadcast

- 全到全广播，CT模式

$$t_{all-to-all}(CT) = t_{all-to-all}(SF)$$

# 目录

- 通信开销模型
- 不同拓扑的通信开销
- 矩阵-向量相乘

# 矩阵的划分——带状划分

■  $16 \times 16$ 阶矩阵,  $p=4$

P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

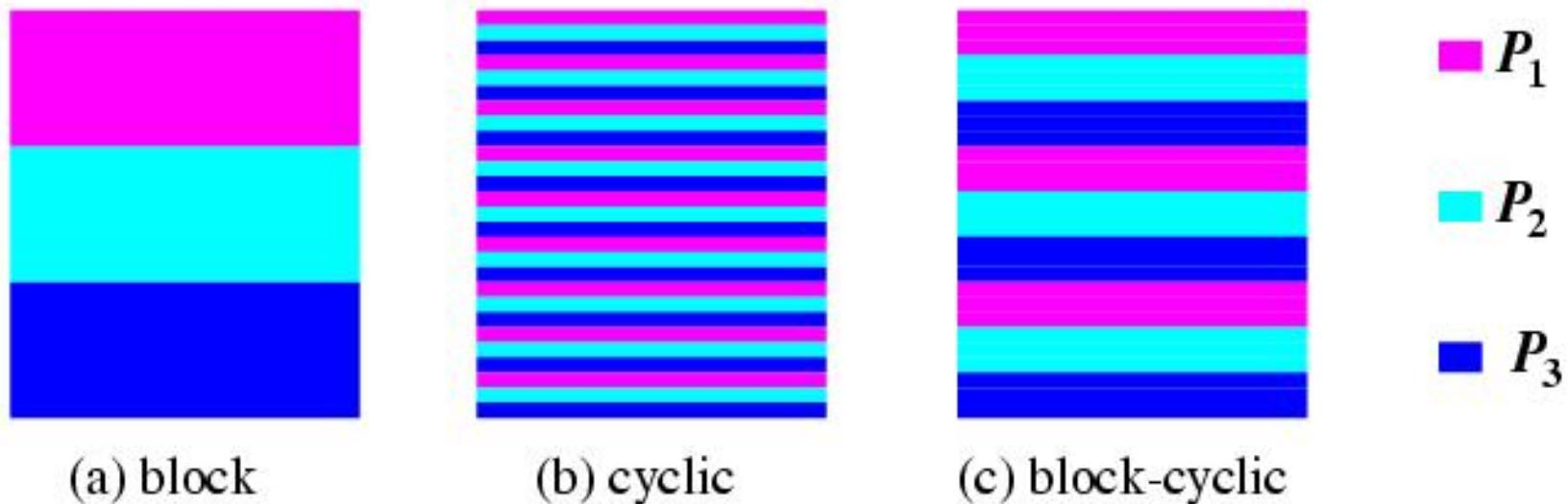
( a )

0	4	8	12	P <sub>0</sub>
1	5	9	13	P <sub>1</sub>
2	6	10	14	P <sub>2</sub>
3	7	11	15	P <sub>3</sub>

( b )

# 矩阵的划分——带状划分

■ 示例:  $p = 3$ ,  $27 \times 27$ 矩阵的3种带状划分



Striped row-major mapping of a  $27 \times 27$  matrix on  $p = 3$  processors.

# 矩阵的划分——棋盘划分

## ■ 8×8阶矩阵, p=16

(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)	(0, 5)	(0, 6)	(0, 7)
P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>				
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)	(1, 6)	(1, 7)
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)	(2, 6)	(2, 7)
P <sub>4</sub>	P <sub>5</sub>	P <sub>6</sub>	P <sub>7</sub>				
(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(3, 6)	(3, 7)
(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)	(4, 6)	(4, 7)
P <sub>8</sub>	P <sub>9</sub>	P <sub>10</sub>	P <sub>11</sub>				
(5, 0)	(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)	(5, 6)	(5, 7)
(6, 0)	(6, 1)	(6, 2)	(6, 3)	(6, 4)	(6, 5)	(6, 6)	(6, 7)
P <sub>12</sub>	P <sub>13</sub>	P <sub>14</sub>	P <sub>15</sub>				
(7, 0)	(7, 1)	(7, 2)	(7, 3)	(7, 4)	(7, 5)	(7, 6)	(7, 7)

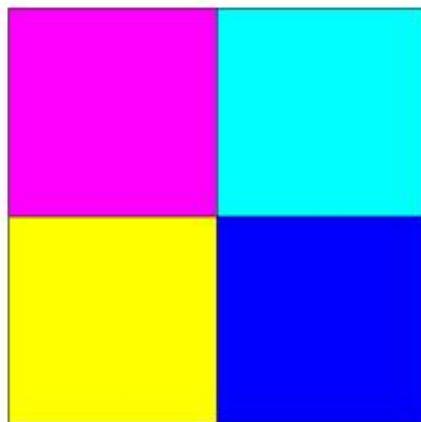
( a )

(0, 0)	(0, 4)	(0, 1)	(0, 5)	(0, 2)	(0, 6)	(0, 3)	(0, 7)
P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>				
(4, 0)	(4, 4)	(4, 1)	(4, 5)	(4, 2)	(4, 6)	(4, 3)	(4, 7)
(1, 0)	(1, 4)	(1, 1)	(1, 5)	(1, 2)	(1, 6)	(1, 3)	(1, 7)
P <sub>4</sub>	P <sub>5</sub>	P <sub>6</sub>	P <sub>7</sub>				
(5, 0)	(5, 4)	(5, 1)	(5, 5)	(5, 2)	(5, 6)	(5, 3)	(5, 7)
(2, 0)	(2, 4)	(2, 1)	(2, 5)	(2, 2)	(2, 6)	(2, 3)	(2, 7)
P <sub>8</sub>	P <sub>9</sub>	P <sub>10</sub>	P <sub>11</sub>				
(6, 0)	(6, 4)	(6, 1)	(6, 5)	(6, 2)	(6, 6)	(6, 3)	(6, 7)
(3, 0)	(3, 4)	(3, 1)	(3, 5)	(3, 2)	(3, 6)	(3, 3)	(3, 7)
P <sub>12</sub>	P <sub>13</sub>	P <sub>14</sub>	P <sub>15</sub>				
(7, 0)	(7, 4)	(7, 1)	(7, 5)	(7, 2)	(7, 6)	(7, 3)	(7, 7)

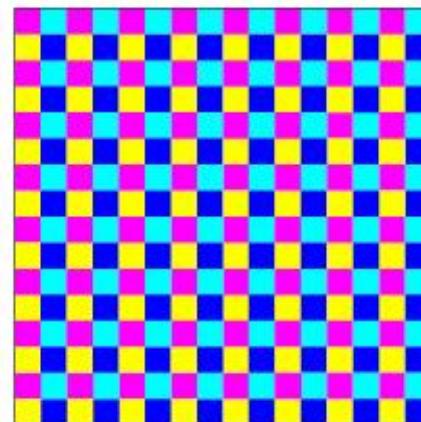
( b )

# 矩阵的划分——棋盘划分

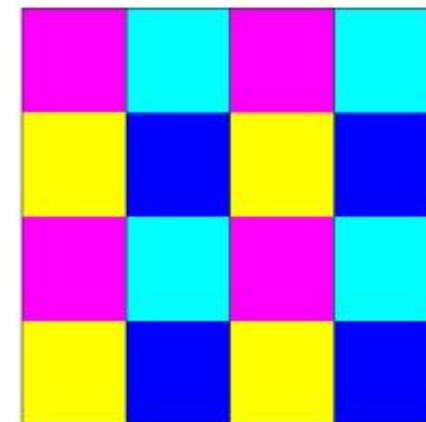
■ 示例： $p = 4$ ,  $16 \times 16$ 矩阵的3种棋盘划分



(a) block



(b) cyclic



(c) block cyclic

- $P_1$
- $P_2$
- $P_3$
- $P_4$

Checkerboard mapping of a  $16 \times 16$  matrix on  $p = 2 \times 2$  processors.

# 矩阵-向量相乘

## ■ 矩阵-向量相乘 (Matrix-Vector Multiplication)

$$c = A \cdot b$$



$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} = \begin{pmatrix} a_{0,0}, a_{0,1}, \dots, a_{0,n-1} \\ \vdots \\ a_{m-1,0}, a_{m-1,1}, \dots, a_{m-1,n-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{pmatrix}$$

矩阵-向量乘法可以转化m个A矩阵行向量和列向量b的内积

$$c_i = (a_i, b) = \sum_{j=0}^{n-1} a_{ij} b_j \quad 0 \leq i < m$$

# 矩阵-向量相乘

- 若 $m=n$ , 将 $n \times n$ 矩阵A与 $n \times 1$ 向量相乘, 得到 $n \times 1$ 的结果向量y
- 串行算法需要 $n^2$ 次乘法和加法运算

$$W = n^2.$$

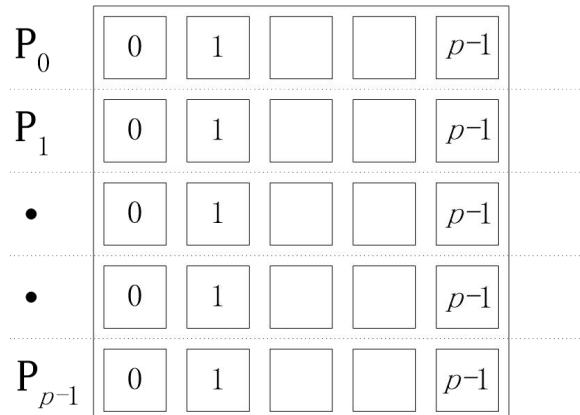
# 矩阵-向量相乘：带状划分

- $n \times n$  矩阵被分布到  $p$  个进程上，每个进程存储矩阵的完整行
- $n \times 1$  列向量也被分布到  $p$  个进程上，每个进程上存储列向量的部分元素 ( $n/p$  个元素)

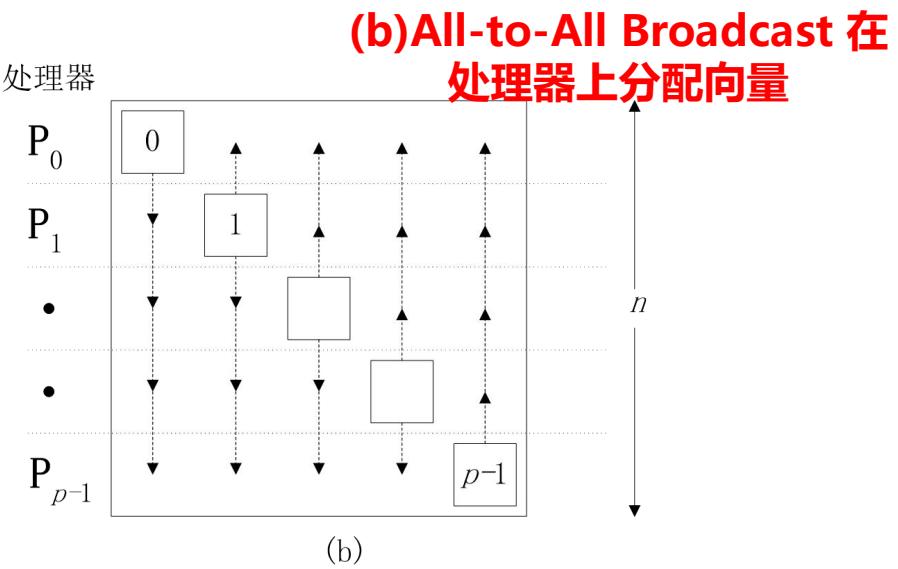
# 矩阵-向量相乘：带状划分



(a) 矩阵A和列向量x的初始化划分



(c) 广播通讯之后，每个进程分配完整向量



(b)



(d) 矩阵和结果向量y的最终分布情况

# 矩阵-向量相乘：带状划分

- 每进程初始时拥有向量x的部分元素 ( $n/p$ 个元素)，之后使用All-to-All广播通信将所有元素分布到所有进程
- 则进程 $P_i$ 计算

$$y[i] = \sum_{j=0}^{n-1} (A[i, j] \times x[j])$$

# 矩阵-向量相乘：带状划分

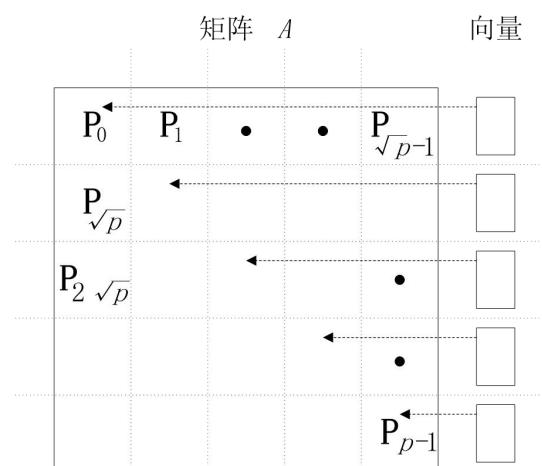
- 现在考虑  $P < n$  的情况
- 每个进程最初存储  $n/p$  矩阵的完整行和大小为  $n/p$  的向量的一部分
- 在所有进程上利用 All-to-All 广播通讯，所传递消息的大小为  $n/p$
- 然后进行  $n/p$  次的局部向量点积 (dot product)
- 超立方连接，并行运行时间为：

$$\begin{aligned} T_p &= \frac{n^2}{p} + t_s \log p + \frac{n}{p} t_w (p - 1) \\ &= \frac{n^2}{p} + t_s \log p + nt_w \end{aligned}$$

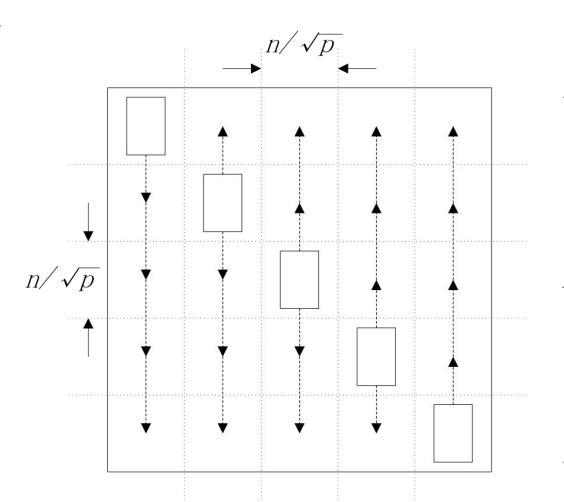
# 矩阵-向量相乘：棋盘划分

- $n \times n$  矩阵在  $p = n^2$  个进程分布，每个进程处理一个元素
- $n \times 1$  向量只分布在最后一列的  $n$  个进程上

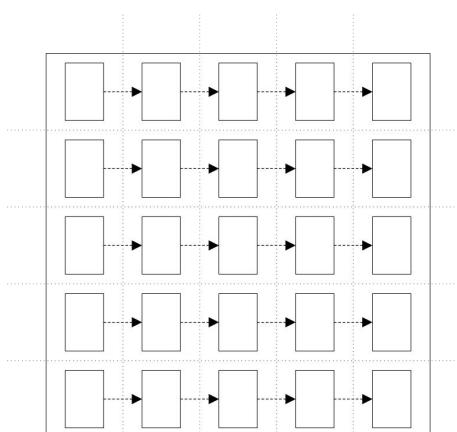
# 矩阵-向量相乘：棋盘划分



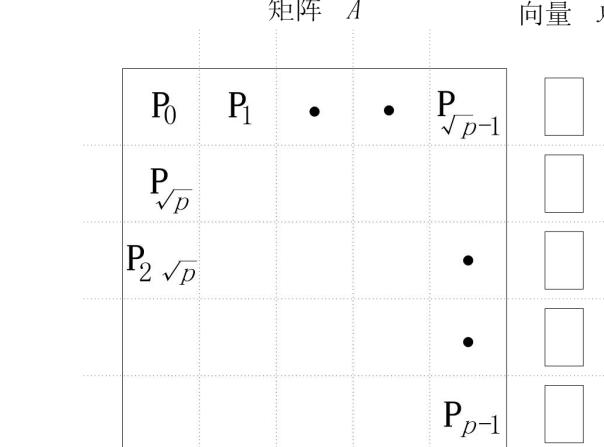
(a) 数据初始化分布及通信，将列向量x分布到对角线上的进程



(b) one-to-all广播通信，将列向量x的部分在每个进程列内广播



(c) all-to-one规约通信，在每个进程行内规约得到部分结果



并行计算

(d) 计算结果的最终分布情况

# 矩阵-向量相乘：棋盘划分

- 当进程数量  $P < n^2$  的时，每个进程处理的元素的个数为： $(n/\sqrt{p}) \times (n/\sqrt{p})$
- 列向量在进程列上分布，每个部分包含元素的个数为： $n/\sqrt{p}$
- 对齐操作、广播通信、规约通信的消息大小的元素数为： $n/\sqrt{p}$
- 计算操作是  $(n/\sqrt{p}) \times (n/\sqrt{p})$  子矩阵和  $n/\sqrt{p}$  子向量的乘积

# 矩阵-向量相乘：棋盘划分

- 对齐操作所需时间

$$t_s + t_w n / \sqrt{p}$$

- 广播和规约通信所需时间

$$(t_s + t_w n / \sqrt{p}) \log(\sqrt{p})$$

- 局部子矩阵-子向量乘法所需时间

$$t_c n^2 / p$$

- 总运行时间

$$T_P \approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p$$

# Reference

- 并行计算课程, 陈国良, 中国科学技术大学
- **Introduction to Parallel Computing, Ananth Grama, Purdue University,**  
[https://www.cs.purdue.edu/homes/ayg/CS525\\_SPR17/chap2\\_slides.pdf](https://www.cs.purdue.edu/homes/ayg/CS525_SPR17/chap2_slides.pdf)
- **Introduction to Parallel Computing, Ananth Grama, Purdue University,**  
[https://www.cs.purdue.edu/homes/ayg/CS525\\_SPR17/chap8\\_slides.pdf](https://www.cs.purdue.edu/homes/ayg/CS525_SPR17/chap8_slides.pdf)

# 附录：矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication)

- 考虑  $n \times n$  稠密方阵乘法，记作  $C = A \times B$
- 串行计算的复杂度的是  $O(n^3)$
- 分块操作： $n \times n$  矩阵可以视作  $q \times q$  个分块子矩阵的矩阵，每块记作  $A_{i,j}$  ( $0 \leq i, j \leq q$ )，维度是  $(n/q) \times (n/q)$
- 需要执行  $q^3$  次矩阵乘法，每个矩阵乘法都是两个  $(n/q) \times (n/q)$  矩阵相乘

# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication)

- 再考虑将  $n \times n$  矩阵  $A$ 、 $B$  划分为  $p$  块,  $A_{i,j}$  and  $B_{i,j}$   
( $0 \leq i, j < \sqrt{p}$ ), 每块维度是  $(n/\sqrt{p}) \times (n/\sqrt{p})$
- 进程  $P_{i,j}$  初始存储  $A_{i,j}$  和  $B_{i,j}$ , 并计算结果子矩阵  $C_{i,j}$
- 计算结果子矩阵  $C_{i,j}$  需要所有子矩阵  $A_{i,j}$  和  $B_{i,j}$  ( $0 \leq k < \sqrt{p}$ )
- 使用 All-to-All 广播通信, 在行方向广播  $A$  的块, 在列方向广播  $B$  的块
- 最后执行局部子矩阵乘法

# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication)

- 两个广播通信所需时间:

$$2(t_s \log(\sqrt{p}) + t_w(n^2/p)(\sqrt{p} - 1))$$

- 需要  $\sqrt{p}$  次子矩阵惩罚，子矩阵维度为  $(n/\sqrt{p}) \times (n/\sqrt{p})$
- 并行运行时间为:

$$T_P = \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}.$$

- 这种算法的主要缺点是没有访存优化

# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication)

### ➤ Cannon算法

- 在该算法中，对第 $i$ 行中的 $\sqrt{p}$ 个进程进行调度，即在同一时间每个进程在使用不同的 $A_{i,k}$ 分块
- 在每次子矩阵乘法之后，这些分块子矩阵在进程之间循环移动，以使每个进程都可以得到一个新的 $A_{i,k}$

# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication) ➤ Cannon算法

(a) A矩阵初始对齐

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

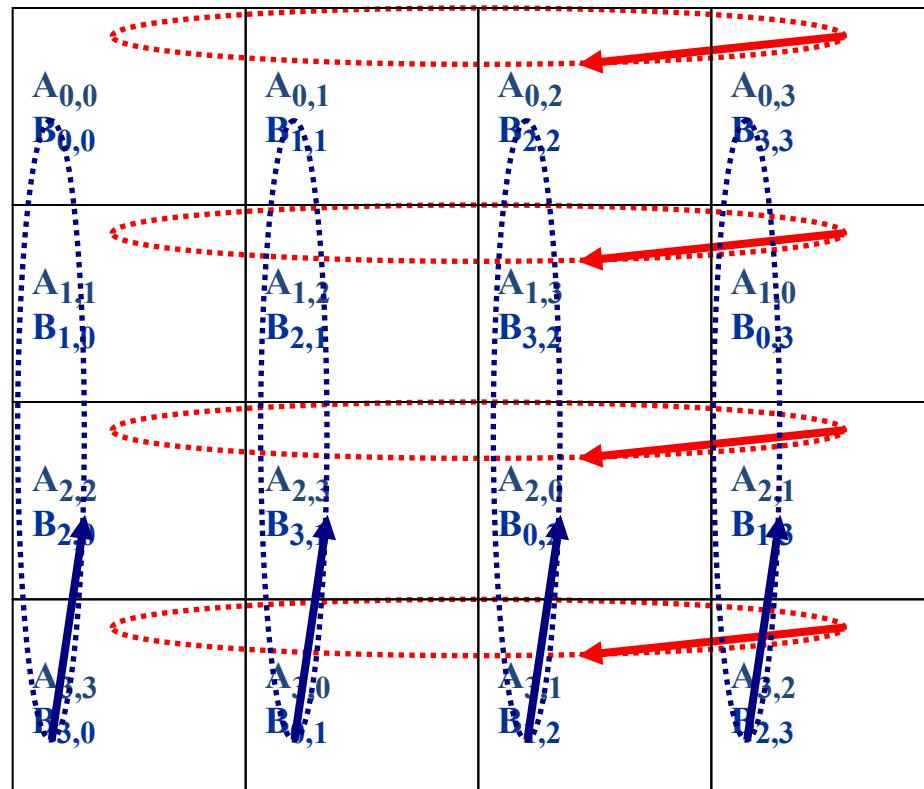
(b) B矩阵初始对齐操作

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication) ➤ Cannon算法

(c) 初始对齐操作之后的A、B矩阵



# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication) ➤ Cannon算法

第一次循环移位后

$A_{0,1}$ $B_{1,0}$	$A_{0,2}$ $B_{2,1}$	$A_{0,3}$ $B_{3,2}$	$A_{0,0}$ $B_{0,3}$
$A_{1,2}$ $B_{2,0}$	$A_{1,3}$ $B_{3,1}$	$A_{1,0}$ $B_{0,2}$	$A_{1,1}$ $B_{1,3}$
$A_{2,3}$ $B_{3,0}$	$A_{2,0}$ $B_{0,1}$	$A_{2,1}$ $B_{1,2}$	$A_{2,2}$ $B_{2,3}$
$A_{3,0}$ $B_{0,0}$	$A_{3,1}$ $B_{3,1}$	$A_{3,2}$ $B_{2,2}$	$A_{3,3}$ $B_{3,3}$

第二次循环移位后

$A_{0,2}$ $B_{2,0}$	$A_{0,3}$ $B_{3,1}$	$A_{0,0}$ $B_{0,2}$	$A_{0,1}$ $B_{1,3}$
$A_{1,3}$ $B_{3,0}$	$A_{1,0}$ $B_{0,1}$	$A_{1,1}$ $B_{1,2}$	$A_{1,2}$ $B_{2,3}$
$A_{2,0}$ $B_{0,0}$	$A_{2,1}$ $B_{1,1}$	$A_{2,2}$ $B_{2,2}$	$A_{2,3}$ $B_{3,3}$
$A_{3,1}$ $B_{1,0}$	$A_{3,2}$ $B_{2,1}$	$A_{3,3}$ $B_{3,2}$	$A_{3,0}$ $B_{0,3}$

第三次循环移位后

$A_{0,3}$ $B_{3,0}$	$A_{0,0}$ $B_{0,1}$	$A_{0,1}$ $B_{1,2}$	$A_{0,2}$ $B_{2,3}$
$A_{1,0}$ $B_{0,0}$	$A_{1,1}$ $B_{1,1}$	$A_{1,2}$ $B_{2,2}$	$A_{1,3}$ $B_{3,3}$
$A_{2,1}$ $B_{1,0}$	$A_{2,2}$ $B_{2,1}$	$A_{2,3}$ $B_{3,2}$	$A_{2,0}$ $B_{0,3}$
$A_{3,2}$ $B_{2,0}$	$A_{3,3}$ $B_{3,1}$	$A_{3,0}$ $B_{0,2}$	$A_{3,1}$ $B_{1,3}$

# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication)

### ➤ Cannon算法

- 执行局部分块子矩阵乘法
- $A$  的分块向左循环移位,  $B$  的分块向上循环移位
- 执行下一次局部分块子矩阵乘法, 并累加到部分结果;  
重复上述步骤, 直到所有 $\sqrt{p}$ 个分块都被计算

# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication)

### ➤ Cannon算法

- 每个循环移位操作所需时间是 $t_s + t_w n^2 / p$ , 一共有 $2\sqrt{p}$ 次循环移位
- 计算 $\sqrt{p}$ 个 $(n/\sqrt{p}) \times (n/\sqrt{p})$ 维矩阵乘法所需时间为 $n^3/p$
- 并行运行时间为:

$$T_P = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w \frac{n^2}{\sqrt{p}}.$$

# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication)

### ➤ DNS算法

- 使用三维划分 (3-D partitioning)
- 将矩阵乘法视作立方体，矩阵  $A$  和  $B$  来自两个正交面，结果  $C$  来自另一个正交面
- 立方体中的每个内部节点代表一个加乘 (Add-Multiply) 运算
- DNS算法使用三维分块方案来划分立方体

# 矩阵-矩阵相乘

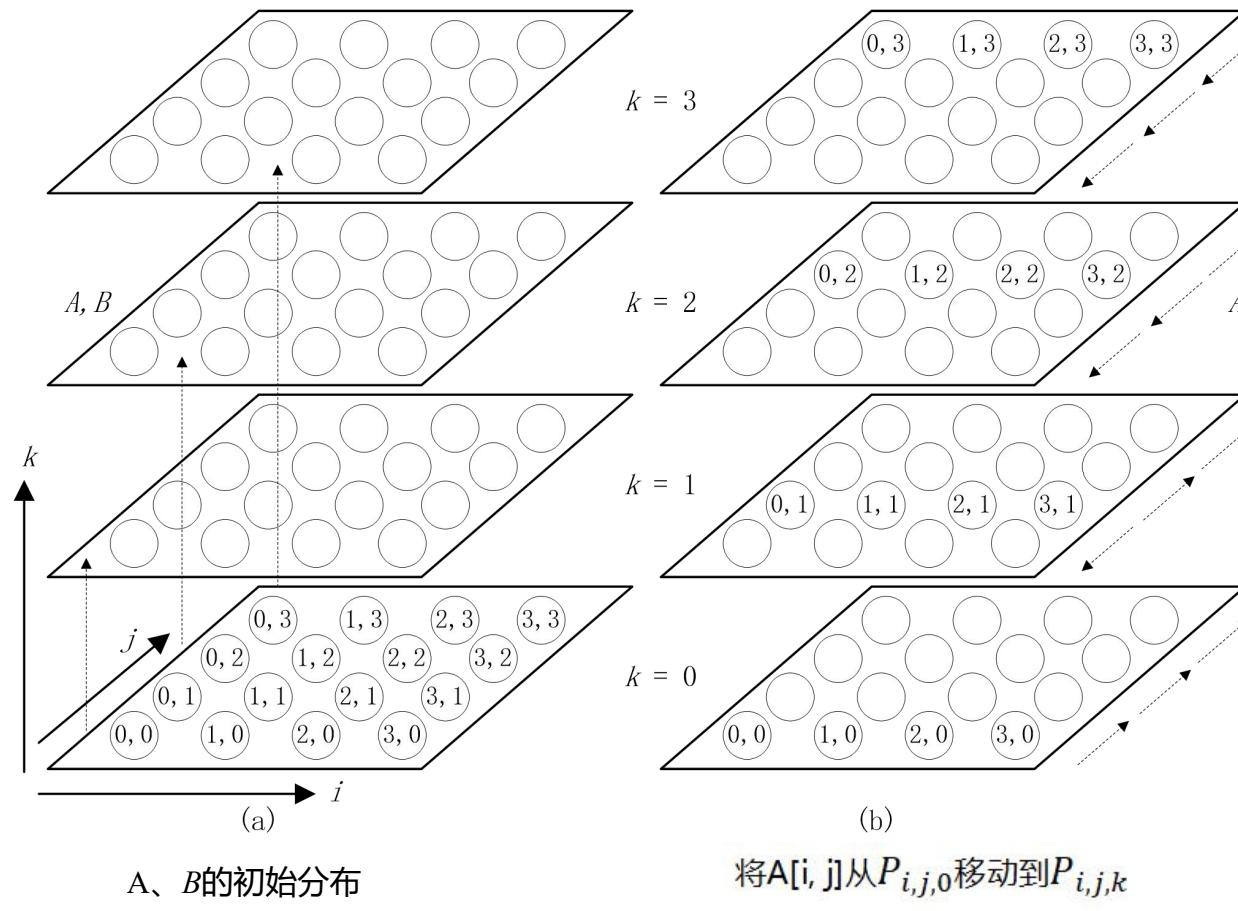
## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication)

### ➤ DNS算法

- 假设处理器组成  $n \times n \times n$  网格
- 移动  $A$  的列和  $B$  的行，并执行广播操作
- 每个处理器计算一个加乘运算
- 之后沿  $C$  方向累加加乘的结果
- 由于每个加乘操作花费常数时间，累加和广播操作花费  $\log n$  时间，因此总的时间复杂度是  $\log n$
- 上述不是开销最优的，在累加方向使用  $n / \log n$  个处理器可使开销最优

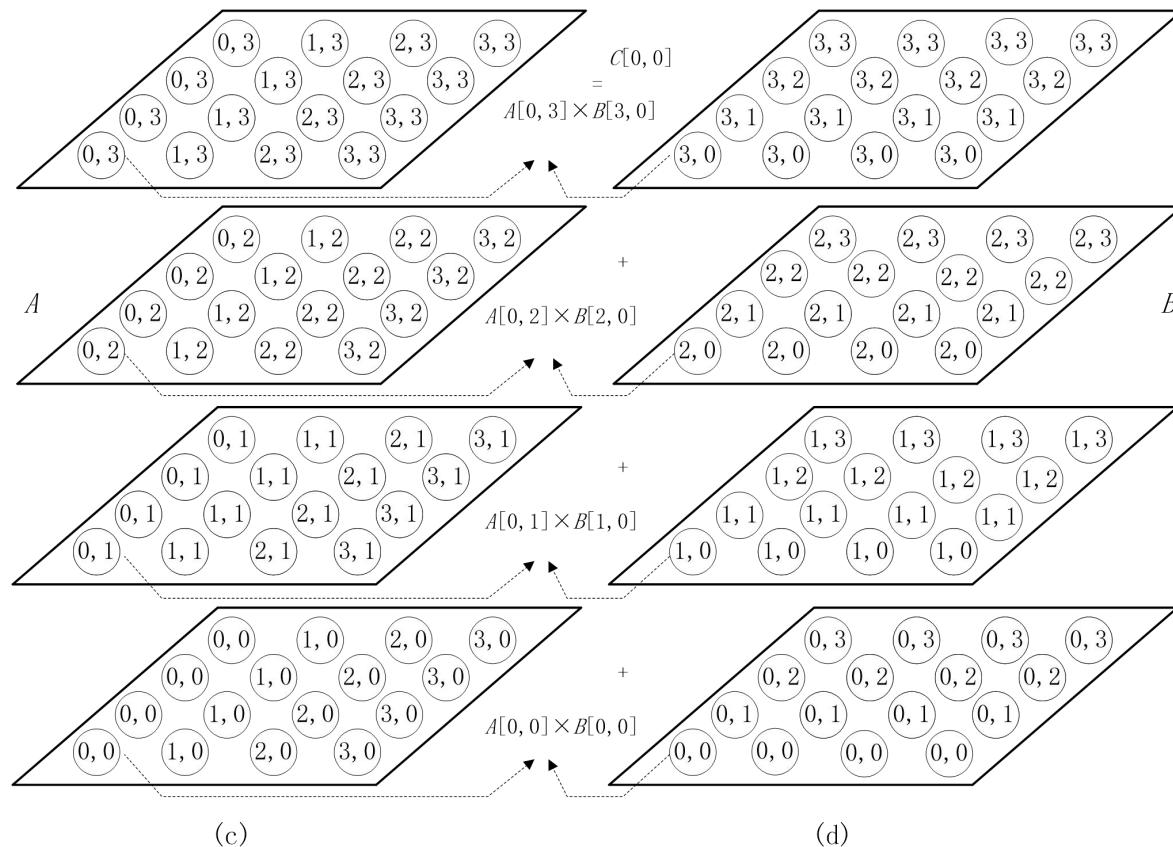
# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication) ➤ DNS算法



# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication) ➤ DNS算法



沿j方向广播 $A[i, j]$ , 得到A的最终分布

B的最终分布情况

# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication)

### ➤ DNS算法

- 当处理器数量小于 $n^3$ 时，即假设进程数量  $p = q^3$  ,  $q < n$
- 则两矩阵分块的子矩阵维度是  $(n/q) \times (n/q)$
- 每个矩阵可以被视作  $q \times q$  的二维方阵，其中每个元素是一个子矩阵
- 该算法与前一个算法相同，只是在这种情况下，操作的基本单位是子块而不是单个元素

# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication)

### ➤ DNS算法

- 当处理器数量小于 $n^3$ 时
- 第一个One-to-One通讯被应用与A和B，每个矩阵上花费的时间是

$$t_s + t_w(n/q)^2$$

- 两个One-to-All广播通讯，每个矩阵上花费的时间为

$$2(t_s \log q + t_w(n/q)^2 \log q)$$

- 规约操作花费的时间为

$$t_s \log q + t_w(n/q)^2 \log q$$

- 计算 $(n/q) \times (n/q)$ 个子矩阵乘法花费的时间为  $(n/q)^3$

- 并行运行时间约为

$$T_P = \frac{n^3}{p} + t_s \log p + t_w \frac{n^2}{p^{2/3}} \log p.$$

# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication)

### ➤ 基本定义

- 内存 (M) — 对于给定问题，所需的存储空间 (例如：字节数)
- 工作负载 (W) — 对于给定问题，所需的操作数 (例如：浮点操作数)，包括数据加载和存储操作
- 速度 (V) — 单个处理器上，单位时间操作数 (例如：浮点数/秒)
- 时间 (T) — 经过的墙上时钟时间，从计算开始到结束(例如：秒)
- 开销 (C) — 处理器数量与执行时间的乘积 (例如：处理器-秒)

# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication)

### ➤ 基本定义

- 下标表示使用的处理器数量 (例如:  $T_1$  表示串行时间,  $W_p$  表示  $p$  个处理器的工作负载)
- 一般假设  $M_p \geq M_1$ , 如果没有数据复制, 假设  $M_p = M_1$  ( $p \geq 1$ ) 是合理的, 此时不再使用下标只记作  $M$
- 如果串行算法是最优的, 且忽略偶然情况, 那么  $W_p \geq W_1$ ; 通常情况下  $W_p > W_1$  ( $p > 1$ )
- 并行开销:  $O_p = W_p - W_1$

# 矩阵-矩阵相乘

## ■ 矩阵-矩阵相乘 (Matrix-Matrix Multiplication)

### ➤ 基本定义

- 数据量通常决定计算量，在这种情况下，我们可以用  $W(M)$  来表示计算复杂度对存储复杂度的依赖
- 例如：计算两个满秩矩阵 ( $n$  维) 乘法
  - $M = (n^2)$ ,  $W = (n^3)$ , 可以推导出  $W(M) = (M^{3/2})$
  - 由于每个数据项都可能用于至少一个操作，因此假设工作  $W$  至少随内存  $M$  线性增长是合理的