中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

SSE316：云计算技术
Cloud Computing Technology

陈壮彬

软件工程学院

https://zbchern.github.io/sse316.html

云的并行计算

- 阿姆达尔定律
- 并行计算机架构
- 传统并行程序设计模型
- 并行程序设计例子
- 消息传递接口

云的并行计算

- 阿姆达尔定律
- 并行计算机架构
- 传统并行程序设计模型
- 并行程序设计例子
- 消息传递接口

阿姆达尔定律

- 我们将程序并行化，以便更快地运行它们
- 并行程序的运行速度会快多少？
  - ✓假设一个程序顺序执行需要T_1时间，而将其并行执行在p个处理器中需要的时间为 T_p
  - ✓假设对于该程序，其不可并行执行的部分占比为s（则其可并行执行的部分为$1-s$）
  - ✓则理论加速比为 $$\frac{T_1}{T_p}=\frac{T_1}{T_1\times s+T_1\times\frac{1-s}{p}}=\frac{1}{s+\frac{1-s}{p}}$$

阿姆达尔定律 (Amdahl 's Law)

阿姆达尔定律 (cont'd)

- 具体例子
  - ✓假设你有4个处理器且你的程序有80%的部分可并行，即：

$$p = 4, s = 80\%$$

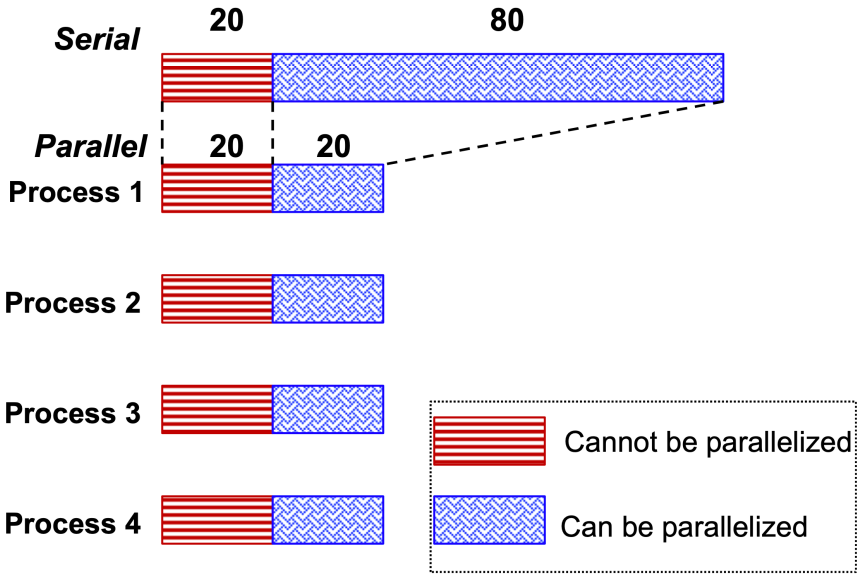
  - ✓根据阿姆达尔定律，加速比为

$$\frac{1}{s + \frac{1-s}{p}} = \frac{1}{0.2 + \frac{0.8}{4}} = 2.5 \text{ times}$$

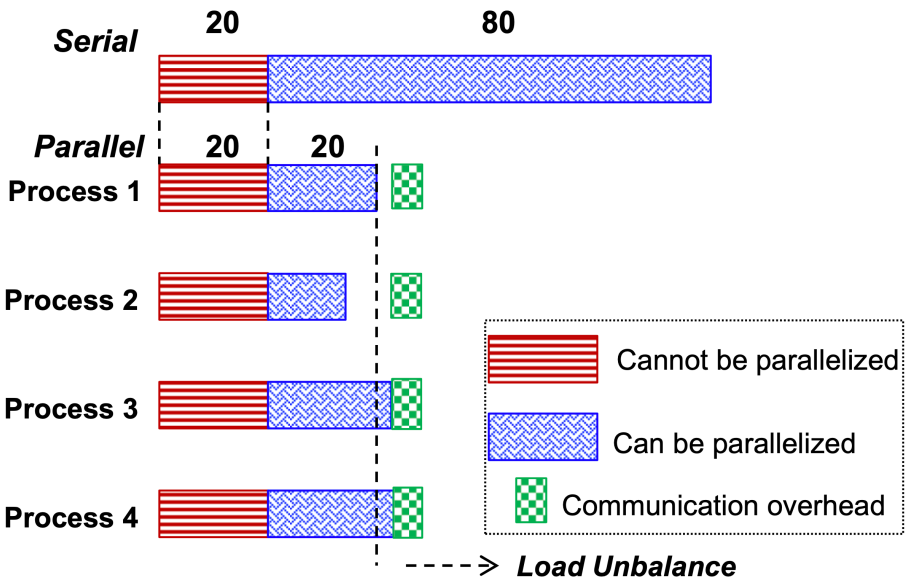
  - ✓然而，实际情况下，2.5倍的加速比是无法达到的

阿姆达尔定律 (cont'd)

- 理想与真实情况
  - ✓阿尔达姆定律太过简单了，无法被应用到实际中
  - ✓当我们运行一个并行程序时，一般来说，进程之间存在通信开销和工作负载不平衡



1. Parallel Speed-up: An Ideal Case



2. Parallel Speed-up: An Actual Case

阿姆达尔定律 (cont'd)

- 为了有效地从并行化中获益，我们应该遵循以下准则

  ✓最大化我们程序中可并行化的部分（增大s）

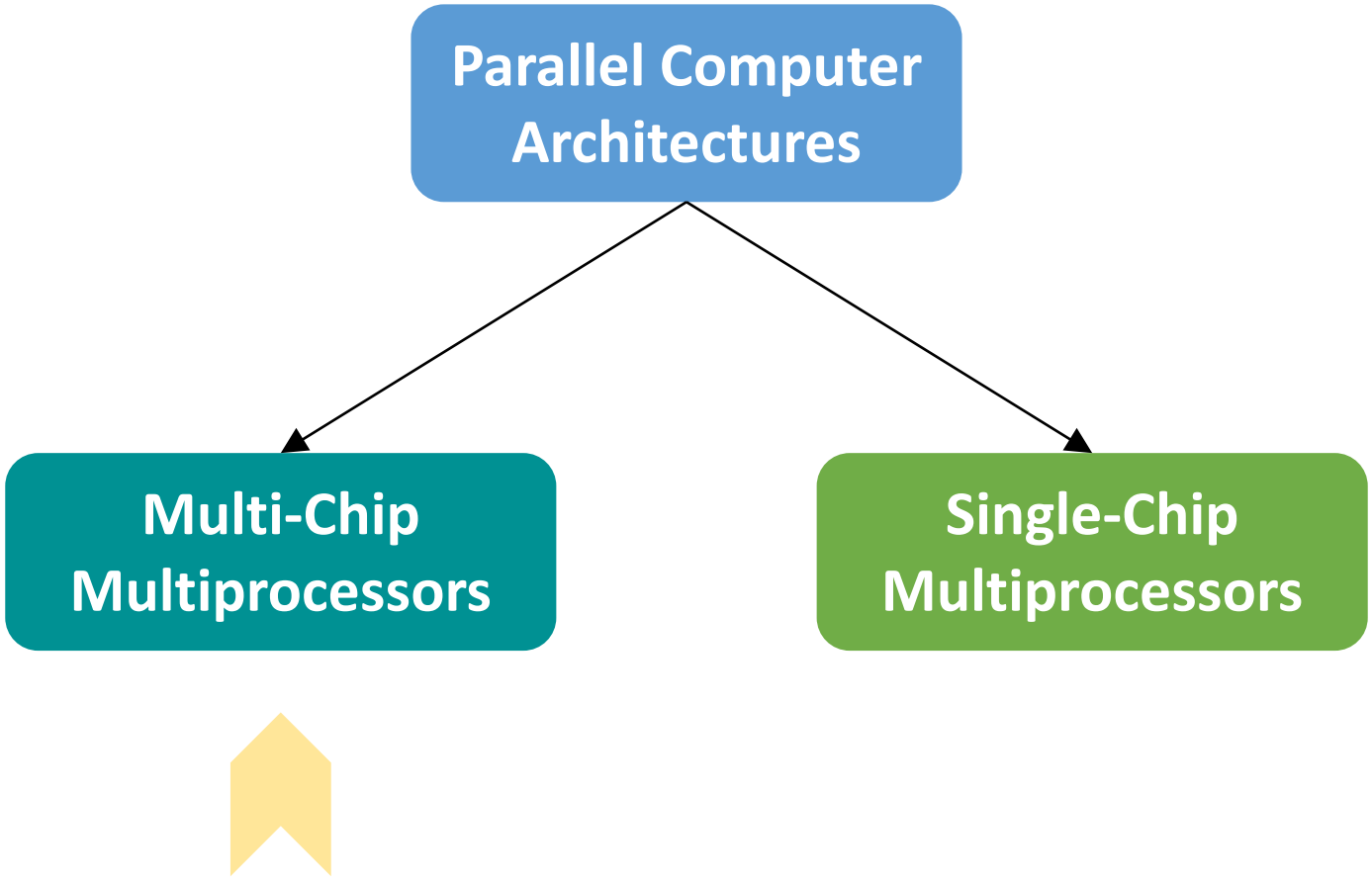
  ✓平衡处理器之间的工作负载

  ✓最小化通信时间

云的并行计算

- 阿姆达尔定律
- 并行计算机架构
- 传统并行程序设计模型
- 并行程序设计例子
- 消息传递接口

并行计算机架构类型


Parallel Computer Architectures
Multi-Chip Multiprocessors
Single-Chip Multiprocessors

多芯片多处理器架构

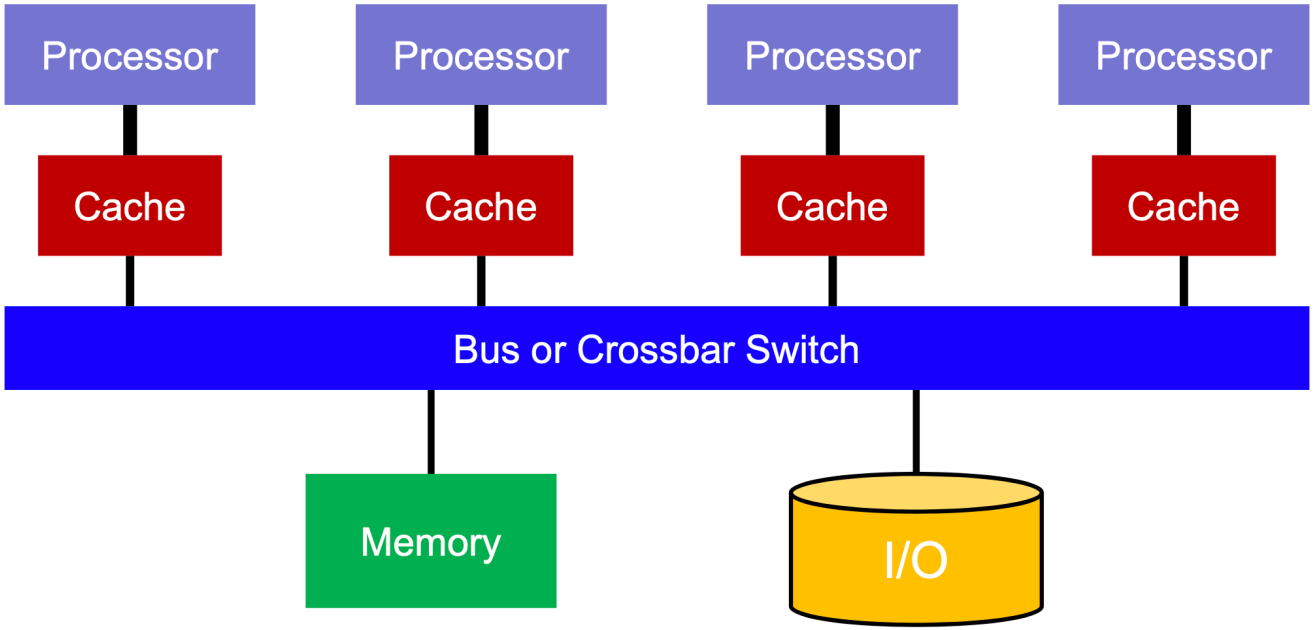
- 我们可以从以下两个方面对多芯片多处理器计算机的体系结构进行分类
  - ✓内存是集中式的还是分布式的
  - ✓地址空间是否共享

		地址空间 Address Space	
		共享 Shared	不共享 Individual
内存 Memory	集中式 Centralized	对称式多处理器 均匀内存访问架构	N/A
	分布式 Distributed	分布式共享内存 非均匀内存访问架构	大规模并行处理器 均匀内存访问架构

对称式多处理器：SMP (Symmetric Multiprocessor)
均匀内存访问架构：UMA (Uniform Memory Access) Architecture
分布式共享内存：DSM (Distributed Shared Memory)
非均匀内存访问架构：NUMA (Non-Uniform Memory Access) Architecture
大规模并行处理器：MPP (Massively Parallel Processors)

对称式多处理器

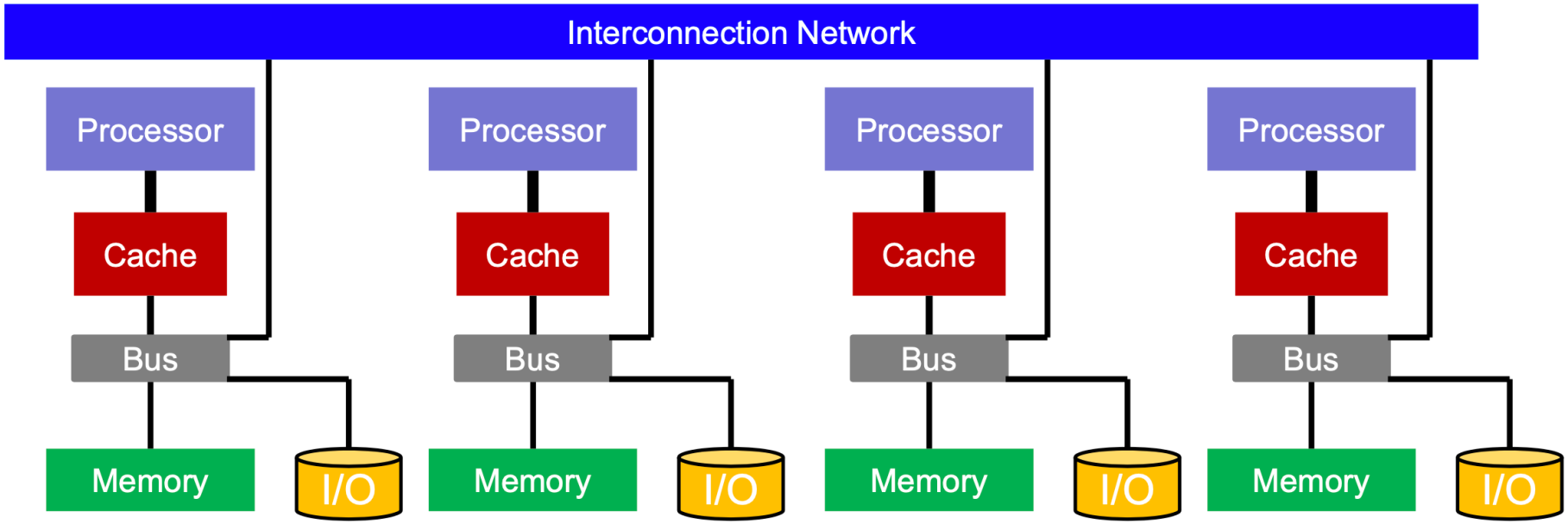
- 使用共享内存，所有处理器均可访问



- 通常，一个操作系统控制整个系统

大规模并行处理器

- 每个节点拥有自己的处理器，内存和I/O子系统

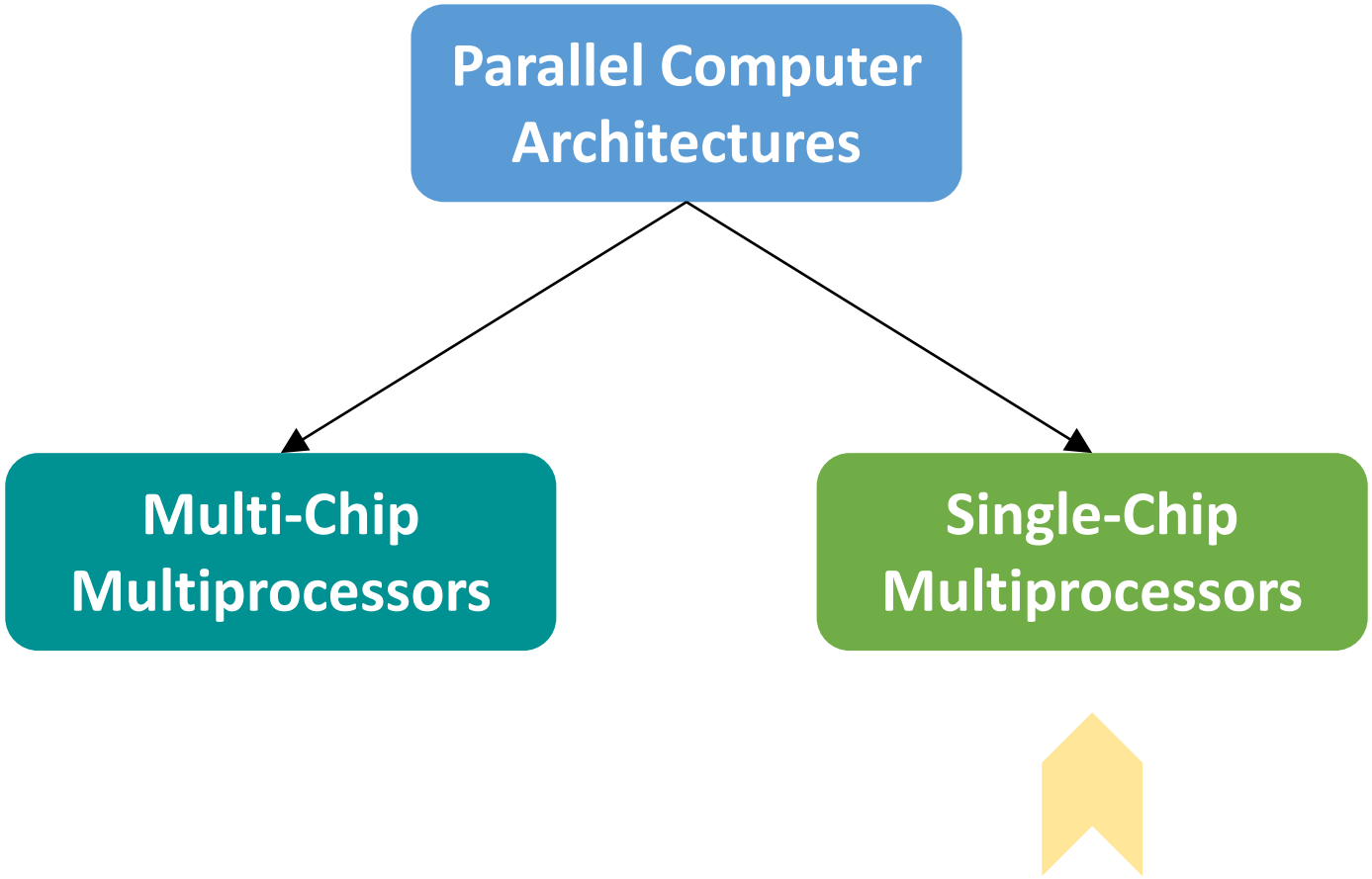


- 通常，每个节点由独立的操作系统控制

分布式共享内存

- “分布式共享内存”的硬件模型与“大规模并行处理器”类似
- DSM使用基于硬件/软件目录的一致性协议为应用程序提供共享地址空间
- 内存延迟取决于内存是本地访问还是远程访问（远程访问则为非均匀内存访问架构）
- 通常，一个操作系统控制整个系统

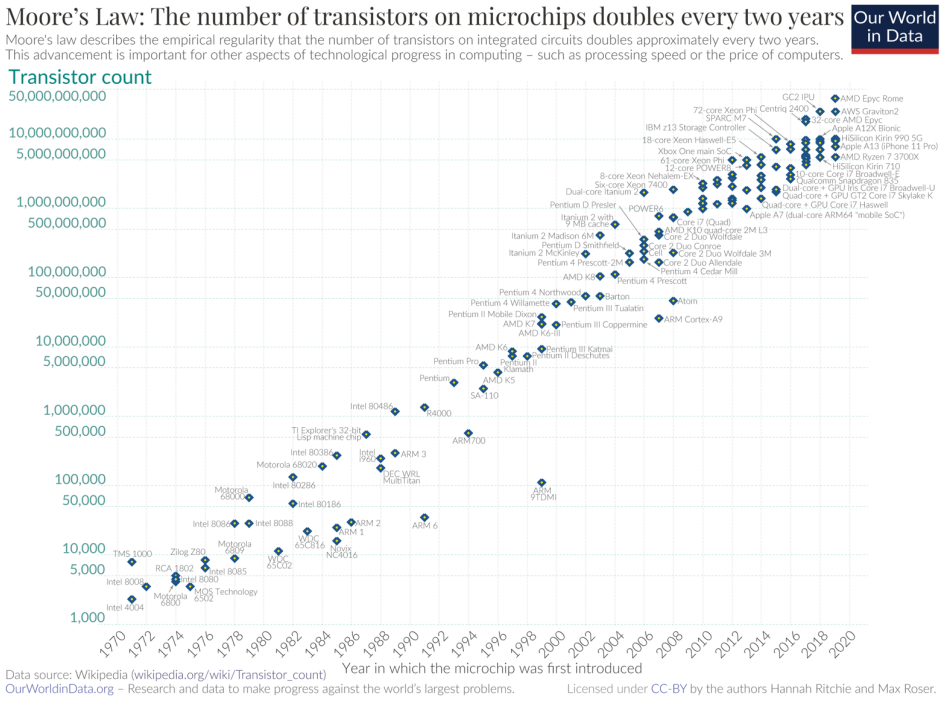
并行计算机架构类型


Parallel Computer Architectures
Multi-Chip Multiprocessors
Single-Chip Multiprocessors

摩尔定律

- 随着芯片制造技术的进步，晶体管越来越小，有可能将更多的晶体管放在芯片上

- 晶体管数量每18到24个月翻一番（此经验观察被称为摩尔定律， Moore's Law ）



我们要怎么利用这些晶体管呢？

如何利用更多的晶体管？

选择1：增加更多缓存到芯片上

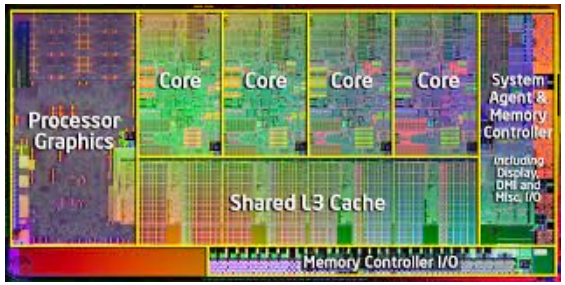
当到达某个点时，增加缓存可能只会将命中率从99%提高到99.5%，应用程序的性能提升有限

选择2：增加更多处理器(核)到芯片上

降低复杂性，能耗，同时应用程序的性能提升明显

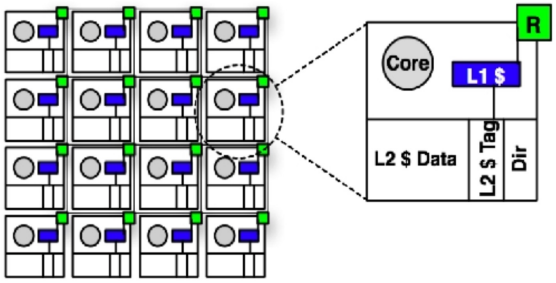
多处理器芯片

- 产物即为多处理器芯片 (Chip Multiprocessors)



- CMP目前被认为是首选的架构

- CMP中的核可能紧密耦合或松散耦合
  - ✓内核可共享缓存，也可以不共享
  - ✓内核可通过消息传递或共享内存的方式实现交互



- 常见的CMP互连（即片上网络）包括总线(bus)、环(ring)、2D网格(2D mesh)和交叉(crossbar)

- CMP可以是同质的(homogeneous)或异质的(heterogeneous)
  - ✓同质CMP拥有相同类型的核
  - ✓异质CMP拥有不同类型的核

摩尔定律是否还成立？

‘Moore’s Law’s dead,’ Nvidia CEO Jensen Huang says in justifying gaming-card price hike

Last Updated: Sept. 22, 2022 at 7:43 a.m. ET
First Published: Sept. 21, 2022 at 6:16 p.m. ET

By Wallace Witkowski Follow

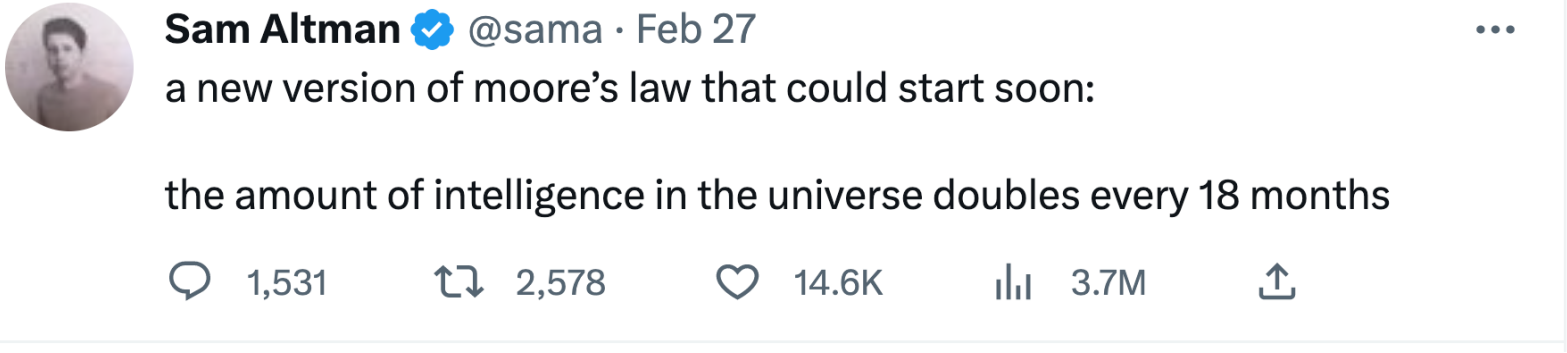
6

Silicon wafers used to make chips ‘not a little bit more expensive, it is a ton more expensive,’ Huang says

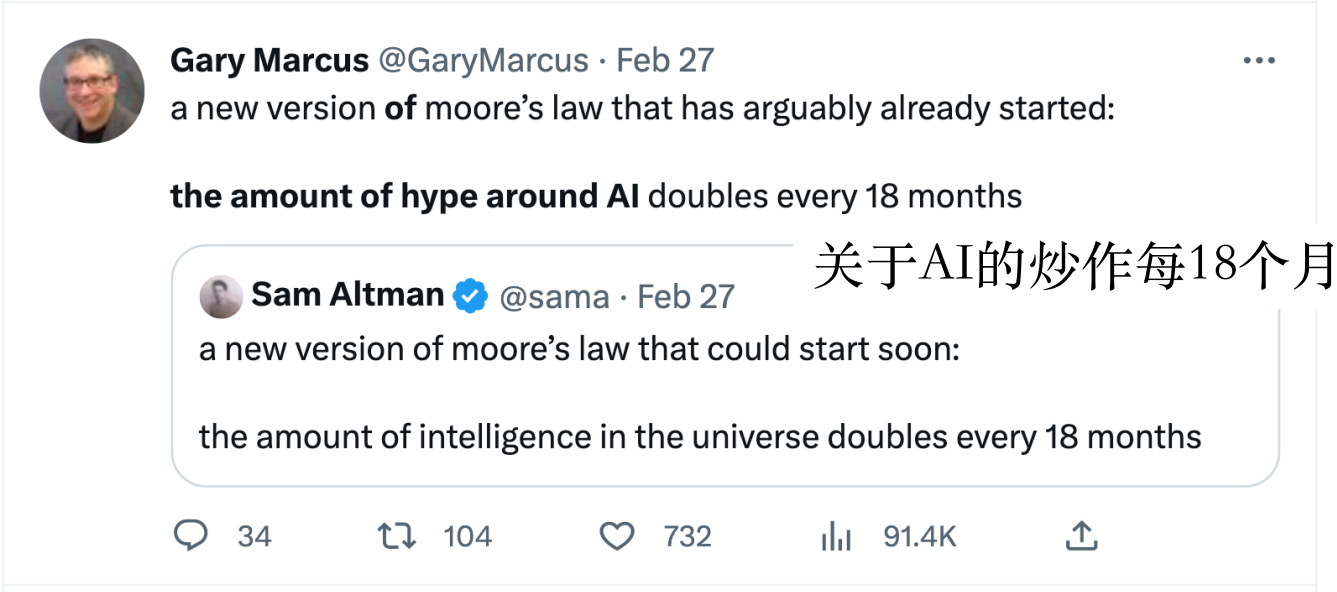
Nvidia CEO Jensen Huang gives the keynote speech Tuesday at GTC. NVIDIA

“Computing is a not a chip problem, it’s a software and chip problem,” Huang said.

新摩尔定律？



ChatGPT之父Sam Altman



关于AI的炒作每18个月翻一番

演员，Rebooting AI作者

云的并行计算

- 阿姆达尔定律
- 并行计算机架构
- 传统并行程序设计模型
- 并行程序设计例子
- 消息传递接口

并行程序设计模型

- 什么是并行程序设计模型

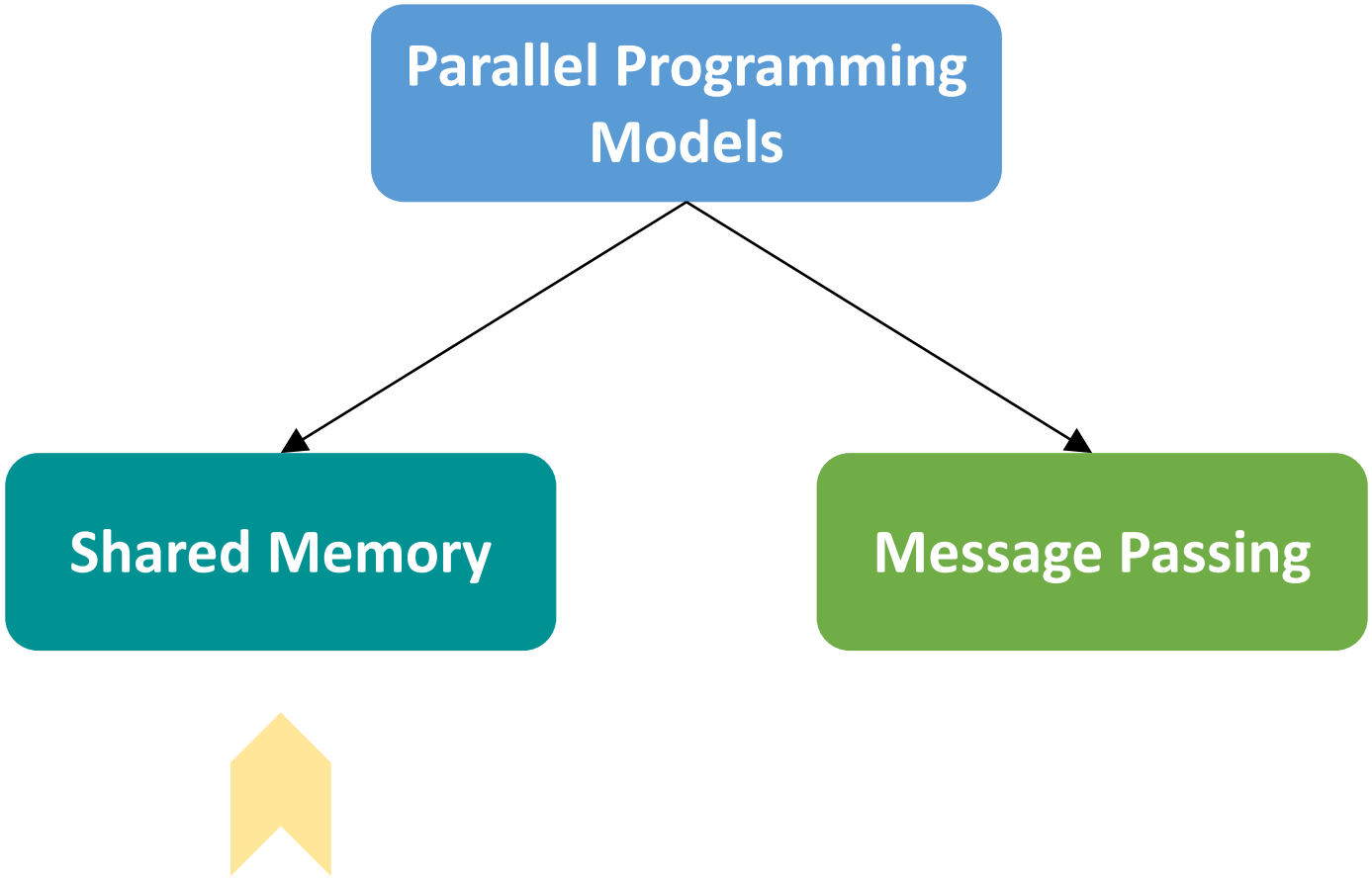
  ✓编程模型是硬件向程序员提供的抽象

  ✓决定了程序员如何将其算法指定为硬件能够理解的并行计算单元（即任务）

  ✓决定了硬件执行并行任务的效率

✓目标：利用底层体系结构的所有处理器（如SMP、MPP、CMP），将程序的运行时间减到最小

传统并行程序设计模型

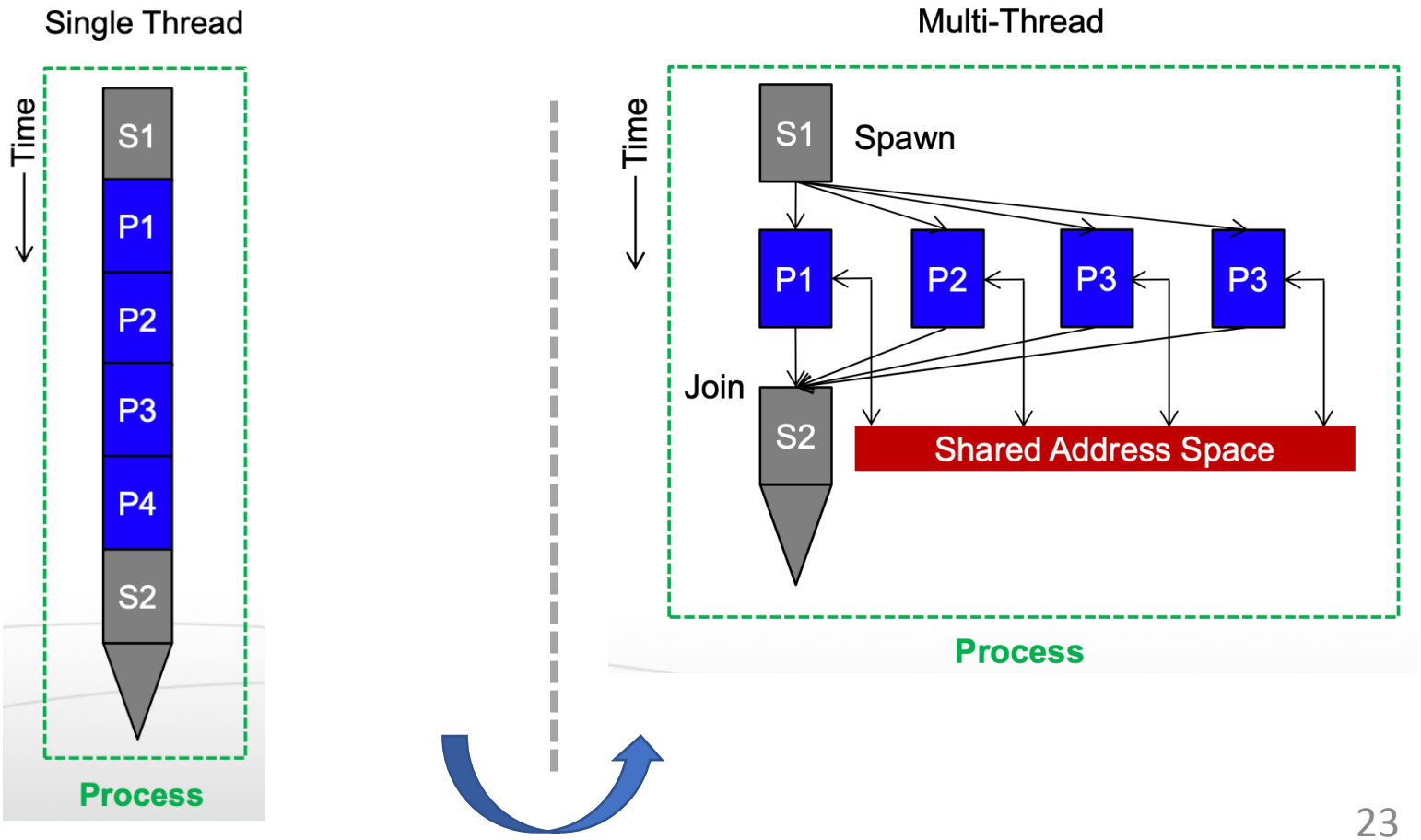

Parallel Programming Models
Shared Memory
Message Passing

共享内存模型

- 在共享内存编程模型中，抽象是并行任务可以访问内存的任何位置
- 并行任务可以通过读写公共内存位置进行通信
- 这类似于来自共享单个地址空间的单个进程的线程
- 多线程程序最适合共享内存编程模型

共享内存模型

S_i = *Serial*
P_j = *Parallel*



共享内存例子

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
sum = 0;
for (i=0; i<8; i++)
  if (a[i] > 0)
    sum = sum + a[i];
Print sum;
```

Sequential

```
begin parallel // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter=4, sum=0;
shared double sum=0.0, a[], b[], c[];
shared lock_type mylock;

start_iter = getid() * local_iter;
end_iter = start_iter + local_iter;
for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];
barrier;

for (i=start_iter; i<end_iter; i++)
  if (a[i] > 0) {
    lock(mylock);
      sum = sum + a[i];
    unlock(mylock);
  }
barrier;     // necessary

end parallel // kill the child thread
Print sum;
```

Parallel

为什么需要锁？

- 共享内存的线程需要同步
- 这通常由互斥实现 (mutual exclusion)
- 互斥要求当有多个线程时，任何时候只允许一个线程写入共享内存的部分（或临界区，critical section）
- 如何在线程之间保持临界区的互斥呢？

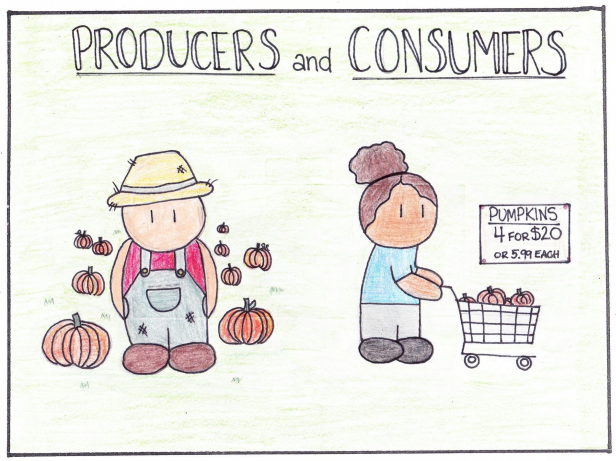
生产者和消费者

生产者

```
while (true) {
     /* produce an item in next_produced */

     while (count == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```



消费者

```
while (true) {
     while (count == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     count--;

     /* consume the item in next_consumed */
}
```

生产者和消费者 (cont'd)

- 假设count的值为5
- count++和count--可能通过机器语言实现

$register_1$ = `count`
$register_1 = register_1 + 1$
`count` = $register_1$

$register_2$ = `count`
$register_2 = register_2 - 1$
`count` = $register_2$

T_0:	*producer*	execute	$register_1$ = `count`	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2$ = `count`	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	`count` = $register_1$	$\{count = 6\}$
T_5:	*consumer*	execute	`count` = $register_2$	$\{count = 4\}$

producer生产了一项，consumer消费了一项，最终count=4（应为5）！

临界区问题

- 因为允许两个进程并发操作count，所以得到了不正确的结果

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

典型进程的通用结构

- 临界区(critical section)：进程执行该区时可修改公共变量
- 进入区(entry section)：进入临界区前，进程请求许可
- 退出区(exit section)：退出临界区，进程做相应设置
- 剩余区(remainder section)：代码剩余区域

皮特森算法 (Peterson's Algorithm)

- 实现进程互斥访问临界区的一种算法
- 算法要求进程共享两个数据项

```
int turn;
boolean flag[2];
```

变量turn表示哪个进程可以进入临界区
如turn=i，表示进程P_i可进入

数据flag表示哪个进程准备进入临界区
如flag[i]=true表示进程P_i准备进入

当 flag[i]=true且turn=i时，进程P_i可以进入临界区

皮特森算法 (cont'd)

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

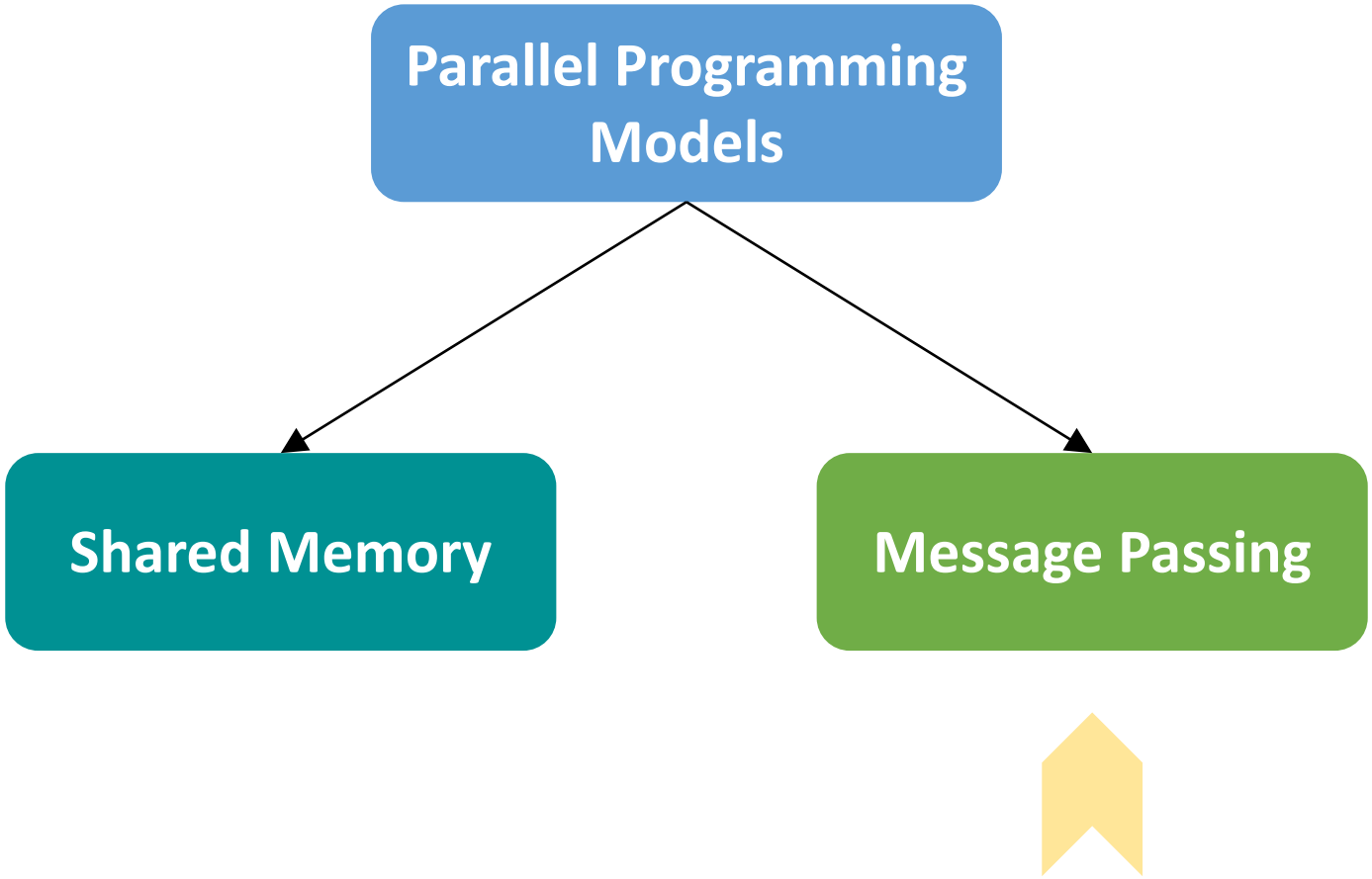
P_i为了进入临界区，将flag[i]设置成true

同时将turn设置成j，表示P_j可以进入临界区

等待另一个进入临界区的进程
（仅有一个进程会进入）

进程离开临界区，此设置允许
另一个进程进入临界区

传统并行程序设计模型

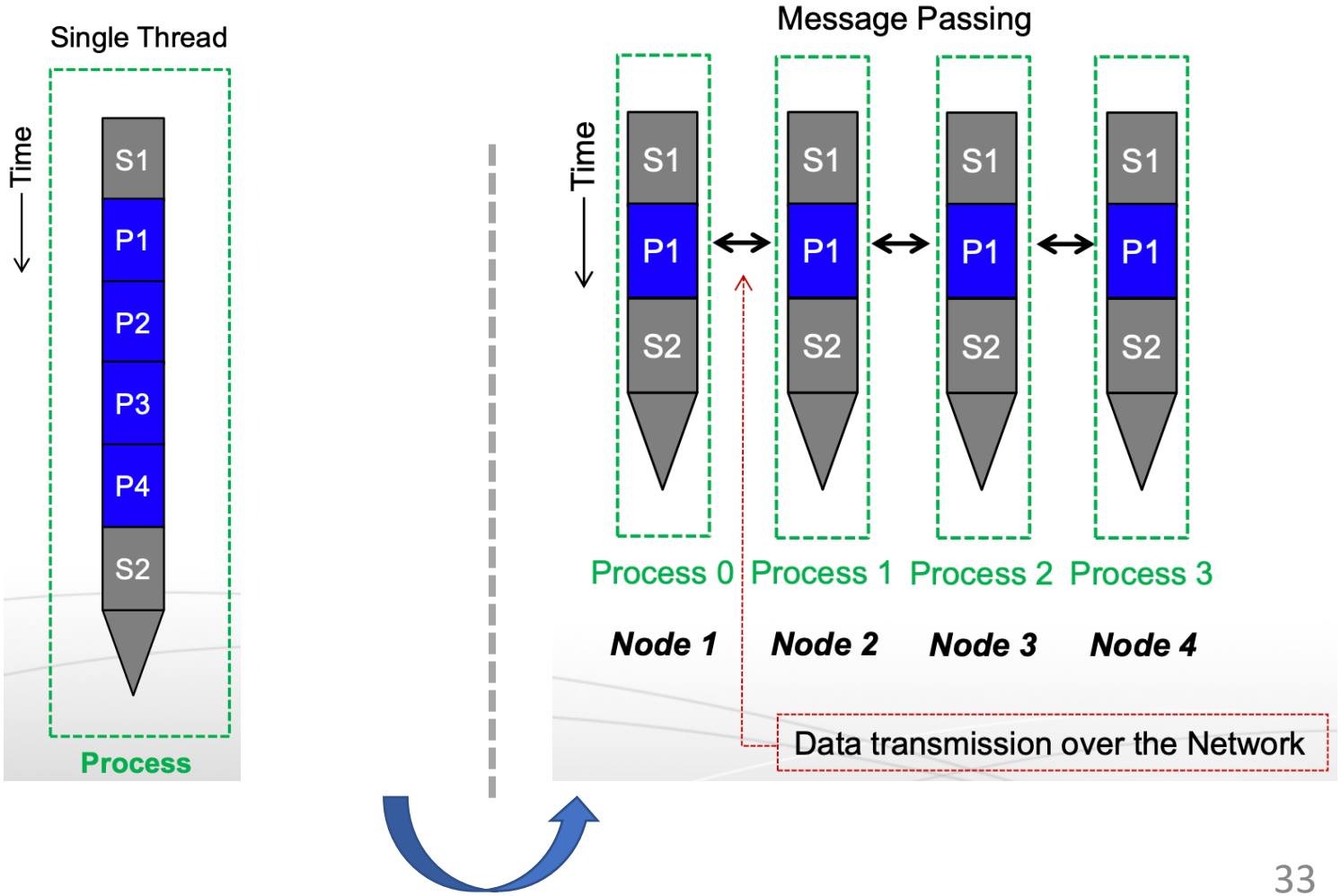

Parallel Programming Models
Shared Memory
Message Passing

消息传递模型

- 在消息传递模型中，并行的任务拥有本地内存
- 一个任务不可访问其他任务的内存
- 因此，它们必须通过消息传递进行数据交换
- 类似于不共享地址空间的进程
- 消息传递接口（MPI）程序最适合消息传递编程模型

消息传递模型 (cont'd)

S_i = *Serial*
P_j = *Parallel*



消息传递模型例子

```
for (i=0; i<8; i++)
      a[i] = b[i] + c[i]
sum=0
for (i=0; i<8; i++)
      if (a[i] > 0)
            sum = sum + a[i];
print sum;
```

Sequential

```
//发送或接受数据数据
if (id == 0) send_msg (P1, [data]);
else recv_msg(P0, [data]);
//本地计算
for (i=start_iter; i<end_iter; i++)
      a[i] = b[i] + c[i];
local_sum = 0;
for (i=start_iter; i<end_iter; i++)
      if (a[i] > 0) local_sum = local_sum + a[i]
//发送或接受数据结果
if (id == 0) { //合并结果
      recv_msg (P1, &local_sum1);
      sum = local_sum + local_sum1;}
else
      send_msg (P0, local_sum);
```

Parallel

消息传递模型例子

```
for (i=0; i<8; i++)
      a[i] = b[i] + c[i]
sum=0
for (i=0; i<8; i++)
      if (a[i] > 0)
            sum = sum + a[i];
print sum;
```

Sequential

```
//发送或接受数据数据
if (id == 0) send_msg (P1, [data]);
else recv_msg(P0, [data]);
//本地计算
for (i=start_iter; i<end_iter; i++)
      a[i] = b[i] + c[i];
local
for (i
      i                                  + a[i]
//发送或接受数据结果
if (id == 0) { //合并结果
      recv_msg (P1, &local_sum1);
      sum = local_sum + local_sum1;}
else
      send_msg (P0, local_sum);
```

不需要互斥操作

Parallel

云的并行计算

- 阿姆达尔定律
- 并行计算机架构
- 传统并行程序设计模型
- 并行程序设计例子
- 消息传递接口

单程序多数据和多程序多数据

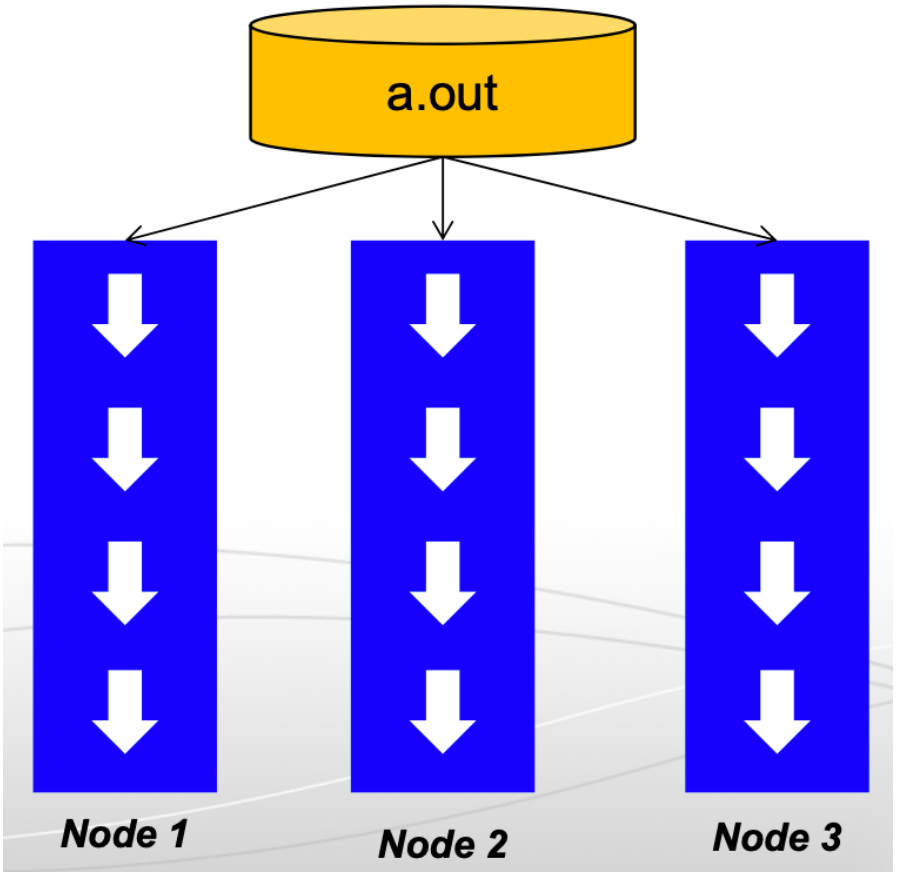
- 使用消息传递来运行多个进程时，可根据并行执行中协作的程序数量做进一步分类

单程序多数据模型
Single Program Multiple Data (SPMD) model

多程序多数据模型
Multiple Programs Multiple Data (MPMD) model

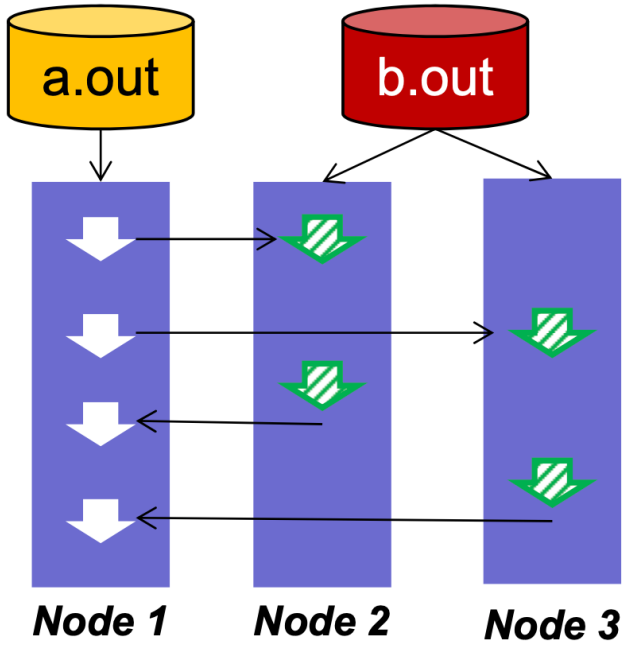
单程序多数据模型

- 在单程序多数据模型中，只有一个程序，每个进程使用相同的可执行文件处理不同的数据集

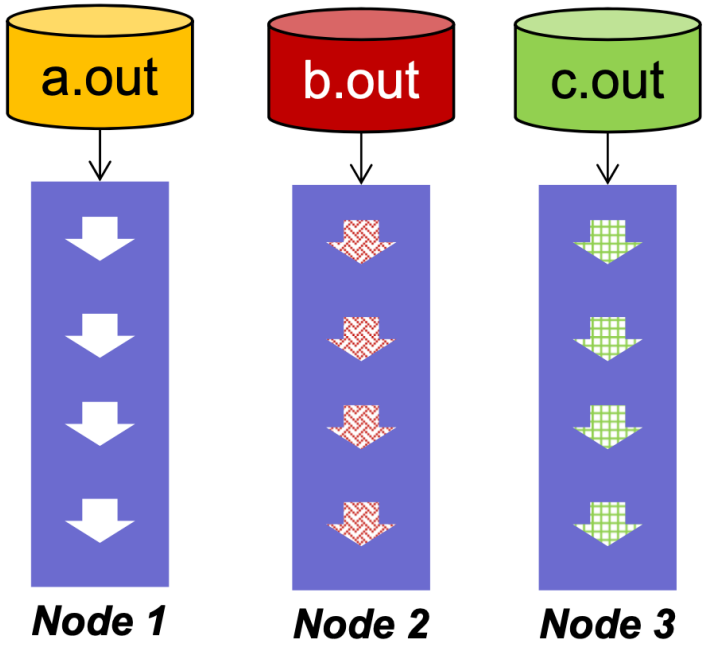


多程序多数据模型

- 在多程序多数据模型中， 每个节点执行各自程序并处理相应数据，进程间相互协作以解决共同的问题
- MPMD有两种风格，master/slave和coupled analysis



1. MPMD: Master/Slave



a.out= Structural Analysis,
b.out = fluid analysis and
c.out = thermal analysis

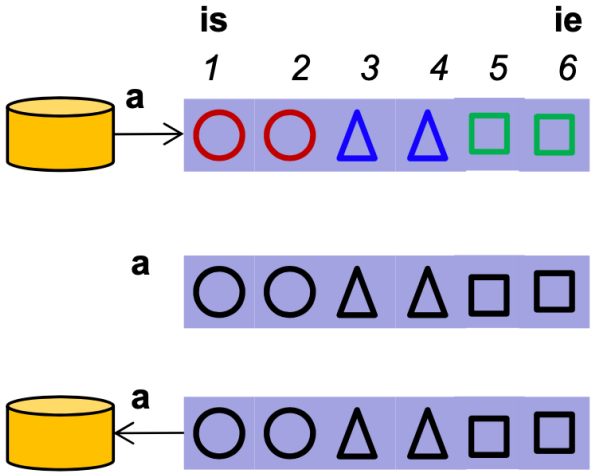
Example

2. MPMD: Coupled Analysis

多程序多数据模型例子

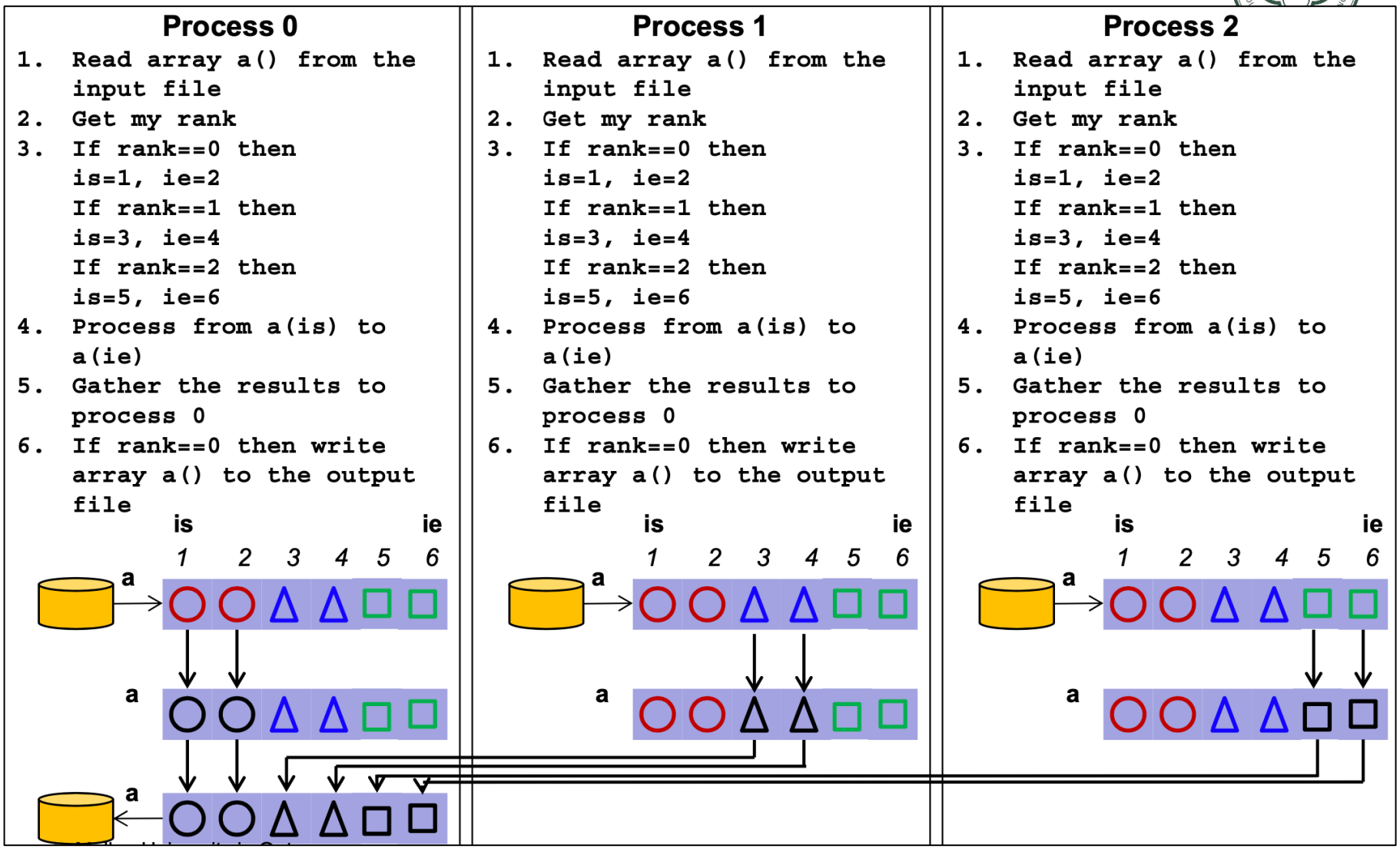
A Sequential Program

```
1. Read array a() from the input file
2. Set is=1 and ie=6  //is = index start and ie = index end
3. Process from a(is) to a(ie)
4. Write array a() to the output file
```



- Colored shapes indicate the initial values of the elements
- Black shapes indicate the values after they are processed

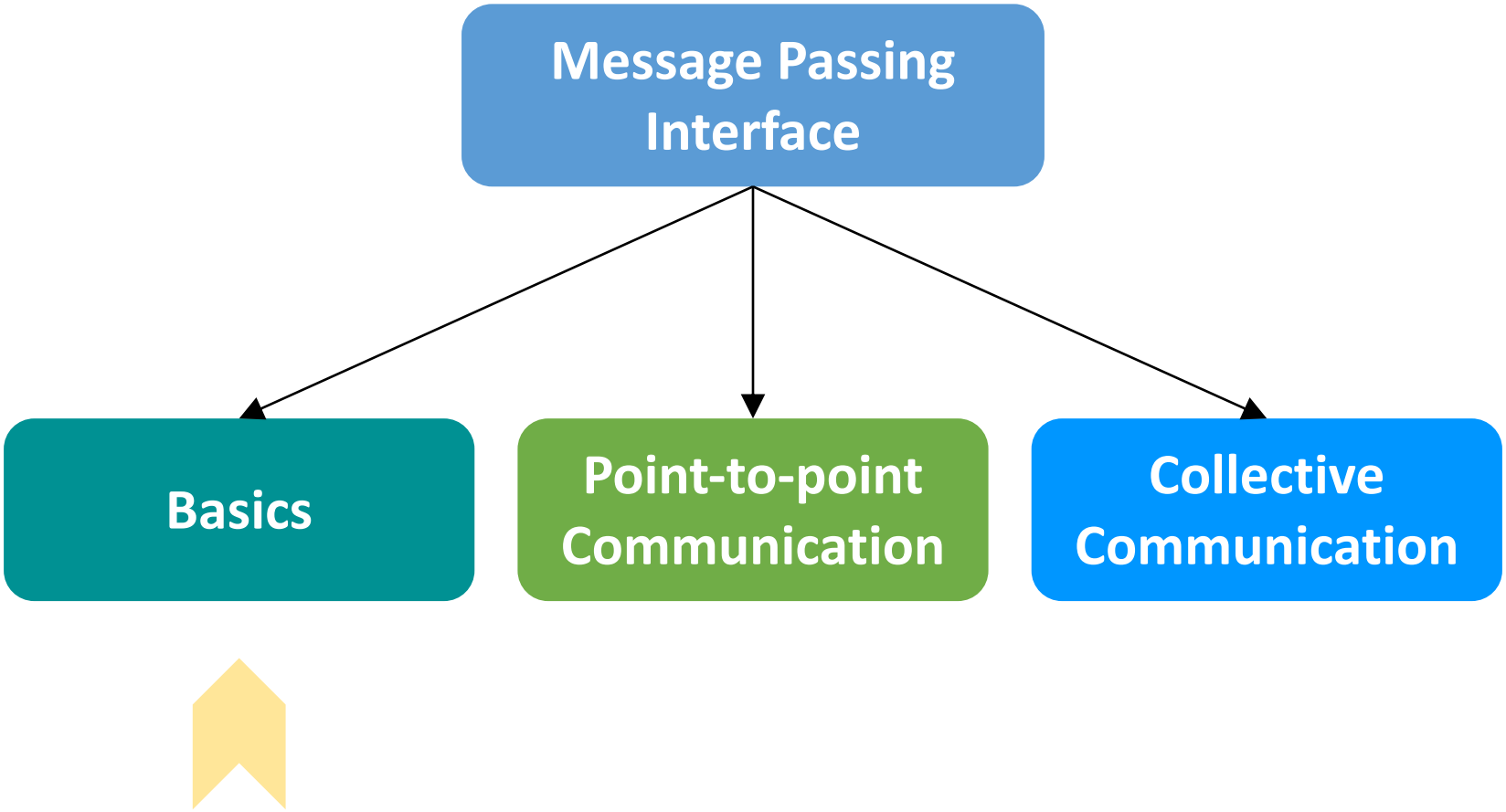
多程序多数据模型例子 (cont'd)



云的并行计算

- 阿姆达尔定律
- 并行计算机架构
- 传统并行程序设计模型
- 并行程序设计例子
- 消息传递接口

消息传递接口


Message Passing Interface
Basics
Point-to-point Communication
Collective Communication

什么是消息传递接口

- 消息传递接口（Message Passing Interface）
  - ✓用于编写消息传递程序的库标准
  - ✓MPI的目标是为消息传递建立一个可移植、高效和灵活的标准
  - ✓MPI本身并不是一个库，而是一个库的规范
  - ✓MPI不是IEEE或ISO标准，但实际上已成为在HPC平台上编写消息传递程序的行业标准

MPI的优点

Reason	Description
Standardization	MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms
Portability	There is no need to modify your source code when you port your application to a different platform that supports the MPI standard
Performance Opportunities	Vendor implementations should be able to exploit native hardware features to optimize performance
Functionality	Over 115 routines are defined
Availability	A variety of implementations are available, both vendor and public domain

MPI编程模型

- MPI是消息传递编程模型的一个示例
- MPI现在几乎用于任何常见的并行架构，包括MPP、SMP集群、工作站集群和异构网络
- 使用MPI，程序员负责正确识别程序的并行性，并使用MPI实现并行算法

通信器和组

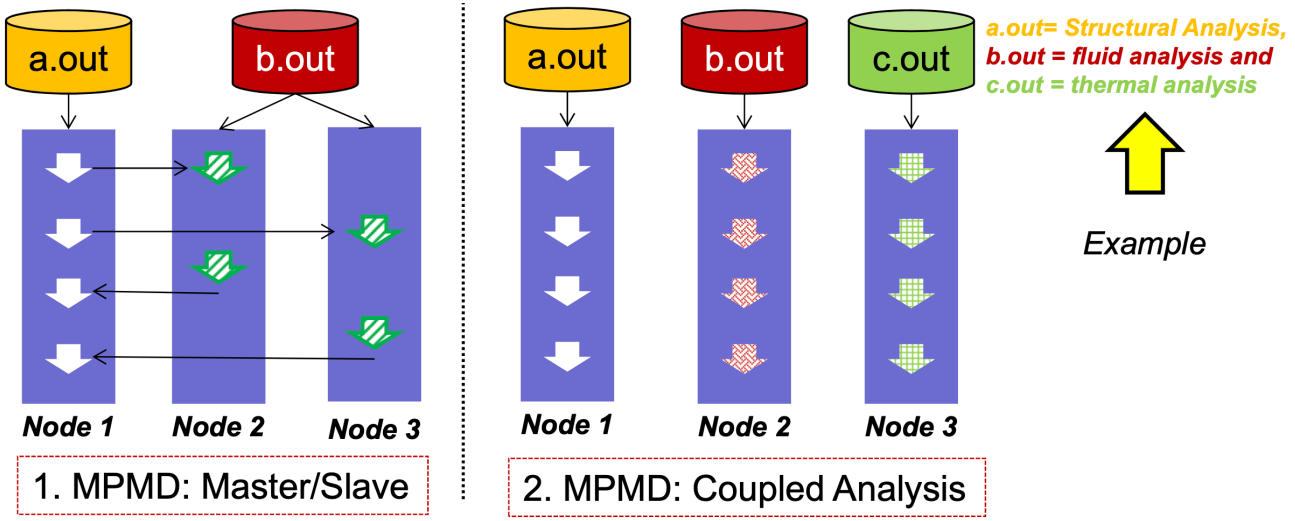
- MPI使用称为通信器和组（communicators and groups）的对象来定义哪些进程集合可以相互通信以解决某个问题
- 大多数MPI子例程（subroutine）要求将通信器指定为参数
- 通信器MPI_COMM_WORLD通常用于调用通信子例程
- MPI_COMM_WORLD是一个预定义的通信器，包含所有MPI进程

Ranks

- 在通信器中，每个进程都有自己的唯一整数标识符，称为rank，在进程初始化时由系统分配
- rank有时称为task ID，rank是连续的，从零开始
- 程序员使用rank来指定消息的来源和目的地
- 应用程序也经常有条件地使用rank来控制程序执行（例如，如果rank=0，做这个/如果rank=1，做那个）

多个通信器

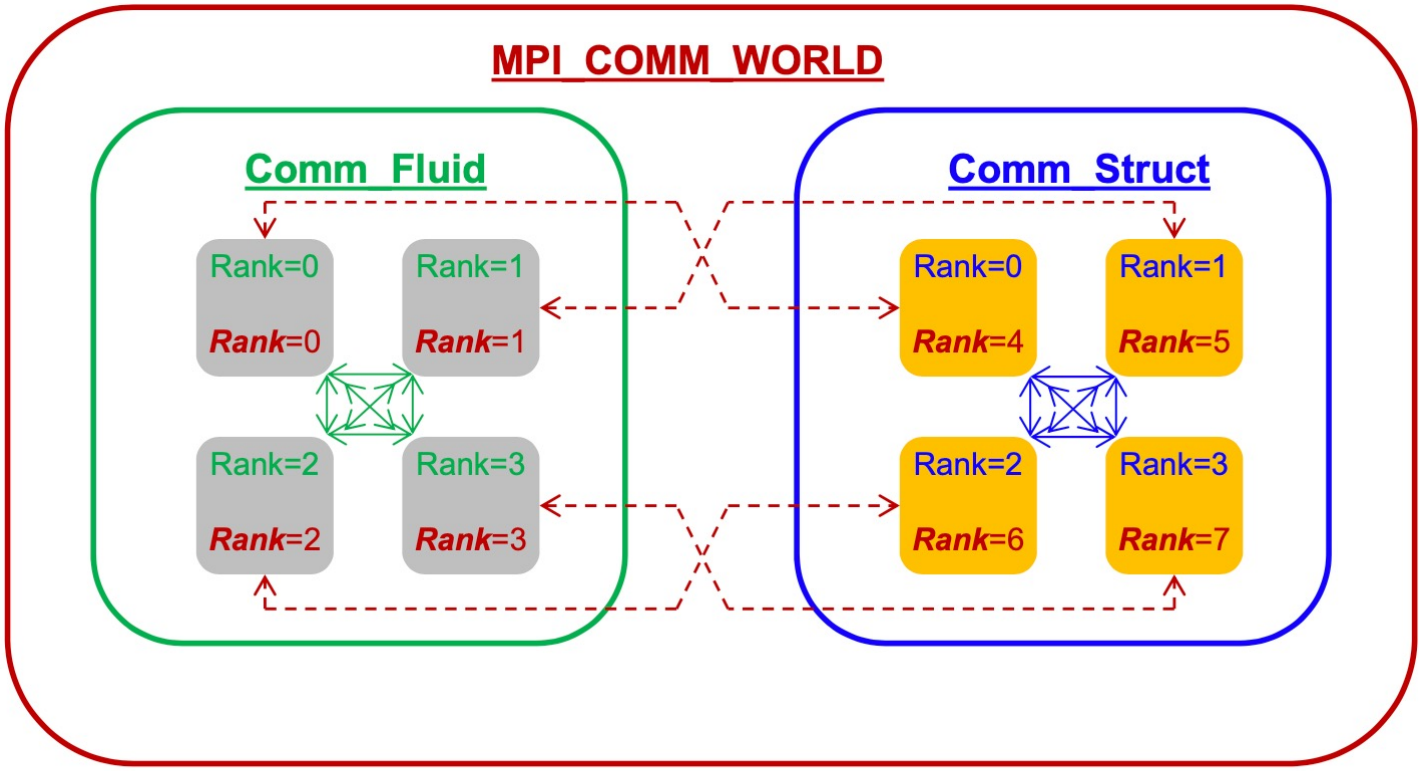
- 一个问题可能由几个子问题组成，每个子问题都可以独立解决
- 这种类型的应用通常出现在MPMD coupled analysis类别中



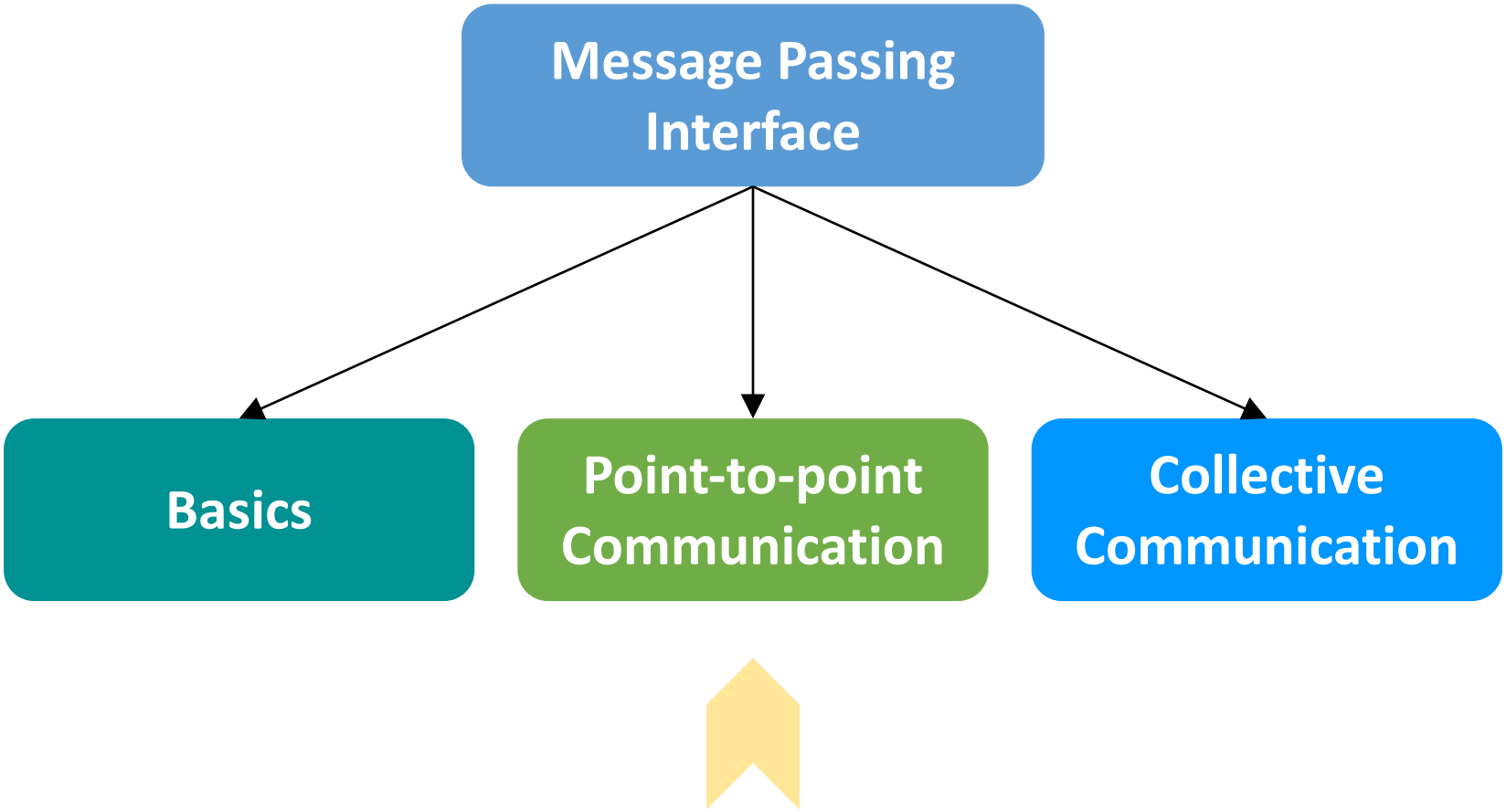
- 我们可以为每个子问题创建一个新的通信器，作为现有通信器的子集
- MPI允许您通过使用MPI_COMM_SPLIT实现这一点

多通信器的例子

- 考虑一个包含流体动力学分析和结构分析的问题，其中每个部分都可以并行计算



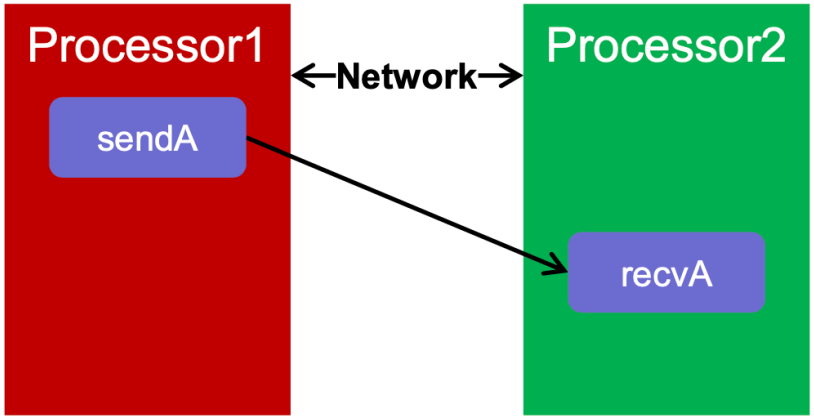
消息传递接口


Message Passing Interface
Basics
Point-to-point Communication
Collective Communication

点对点通信

- MPI点对点通信通常仅涉及两个不同的MPI任务之间的消息传递

一个任务执行发送操作，另一个执行匹配的接收操作



- 理想情况下，每个发送操作都将与其匹配的接收完全同步

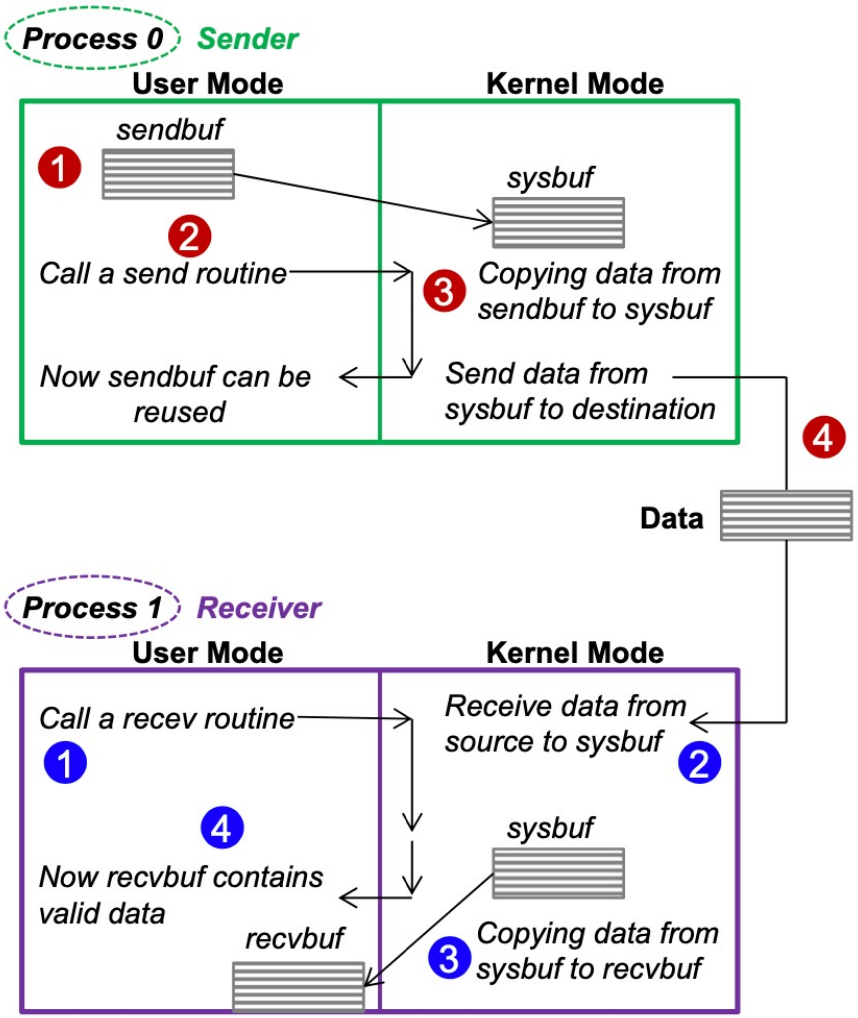
- 虽然较少发生，单MPI实现必须支持能够在两个任务不同步时处理数据存储

两个例子

- 考虑如下点对点通信的两个例子
  - ✓一个send operation在receive operation准备好之前5秒就发生了，在这5秒内消息存放在哪儿？
  - ✓一个receive operation一次仅能接受一个消息，那么当有多个消息同时发送时，其他消息存放在哪儿？

点对点通信的具体步骤

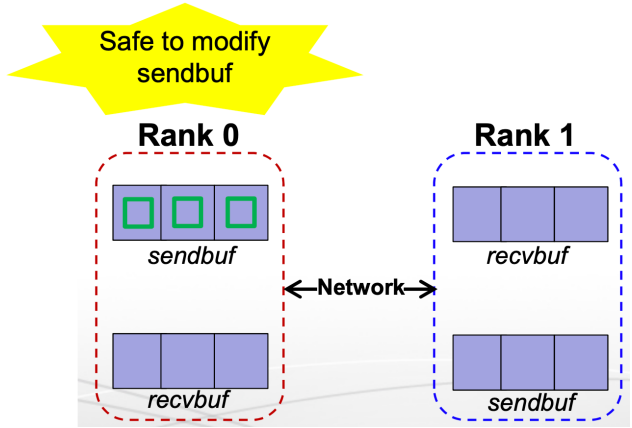
1. The data is copied to the user buffer by the user
2. The user calls one of the MPI send routines
3. The system copies the data from the user buffer to the system buffer
4. The system sends the data from the system buffer to the destination process



1. The user calls one of the MPI receive routines
2. The system receives the data from the source process and copies it to the system buffer
3. The system copies data from the system buffer to the user buffer
4. The user uses data in the user buffer

阻塞发送和接收

- 在点对点通信中，我们通常区分如下两种方式：
  - ✓阻塞通信（blocking communication）
  - ✓非阻塞通信（non-blocking communication）



- 阻塞发送例程只有在应用程序缓冲区能被安全修改以供重用后才会返回
  - ✓安全意味着修改不会影响用于接收任务的数据
  - ✓这并不意味着数据已由接收方接收，它可能位于发送方的系统缓冲区中

阻塞发送和接收

- 一个阻塞发送操作可以是
  - ✓同步的：意味着发送与接收任务发生握手以确认安全发送
  - ✓异步的：意味着发送方的系统缓冲区用于保存数据，以便最终交付给接收方
- 阻塞接收操作仅在数据到达（即存储在应用程序recvbuf中）并准备好由程序使用后返回

非阻塞发送和接收

- 非阻塞发送和非阻塞接收操作类似

  ✓几乎立即返回

  ✓他们不等待任何通信事件完成，例如：
    - 把消息从用户缓存复制到系统缓存
    - 或者消息是否实际到达

非阻塞发送和接收 (cont'd)

- 然而，在确保库实际执行了请求的非阻塞操作之前，修改应用程序缓冲区是不安全的

- 如果在复制完成之前使用应用程序缓冲区
  - ✓不正确的数据可能被复制到系统缓冲区（在非阻塞发送的情况下）
  - ✓或者接收缓冲区不包含想要的内容（在非阻塞接收的情况下）

可以在发送或接收操作后使
用MPI_WAIT()确保复制完成

为什么要使用非阻塞操作？

- 非阻塞通信负责度高且容易出错，为什么还要使用？

  ✓非阻塞操作通常比阻塞操作更快

  ✓当数据在应用和系统缓存之间来回拷贝时，我们可以进行其他操作

MPI点对点通信子例程

Routine	Signature
Blocking send	int *MPI_Send*(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
Non-blocking send	int *MPI_Isend*(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
Blocking receive	int *MPI_Recv*(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
Non-blocking receive	int *MPI_Irecv*(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

MPI例子：数组加和

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

#define array_size 20
#define num_elem_pp 10
#define tag1 1
#define tag2 2

int array_send[array_size];
int array_recv[num_elem_pp];

int main(int argc, char **argv){

        int myPID;
        int num_procs;
        int sum = 0;
        int partial_sum = 0;
        double startTime = 0.0;
        double endTime;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myPID);
        MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

        if(myPID == 0 )
        {
                startTime = MPI_Wtime();
```

Initialize MPI environment

The Master

MPI例子：数组加和 (cont'd)

The Master allocates equal portions of the array to each process

```
int i;
for(i= 0; i < array_size; i++)
        array_send[i] = i;
```

```
int j;
for(j = 1; j < num_procs; j++){
        int start_elem = j * num_elem_pp;
        int end_elem = start_elem + num_elem_pp;

        MPI_Send(&array_send[start_elem], num_elem_pp, MPI_INT, j, tag1,
                MPI_COMM_WORLD);
}
```

The Master calculates its partial sum

```
int k;
for(k=0; k < num_elem_pp; k++)
        sum += array_send[k];
```

The Master collects all partial sums from all processes and calculates a grand total

```
int l;
for(l = 1; l < num_procs; l++){

        MPI_Recv(&partial_sum, 1, MPI_INT, MPI_ANY_SOURCE, tag2,
                MPI_COMM_WORLD, &status);

        printf("Partial sum received from process %d = %d\n", l, status.MPI_SOURCE);
        sum += partial_sum;
}
```

MPI例子：数组加和 (cont'd)

The Slave

```
endTime = MPI_Wtime();
printf("Grand Sum = %d and time taken = %f\n", sum, (endTime-startTime));
}else{
```

The Slave receives its array share

```
MPI_Recv(&array_recv, num_elem_pp, MPI_INT, 0, tag1, MPI_COMM_WORLD, &status);
```

The Slave calculates its partial sum

```
int i;
for(i = 0; i < num_elem_pp; i++)
        partial_sum += array_recv[i];
```

```
MPI_Send(&partial_sum, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD);
```

The Slave sends to the Master its partial sum

```
}
MPI_Finalize();
```

Terminate the MPI execution environment

```
}
```

单向通信

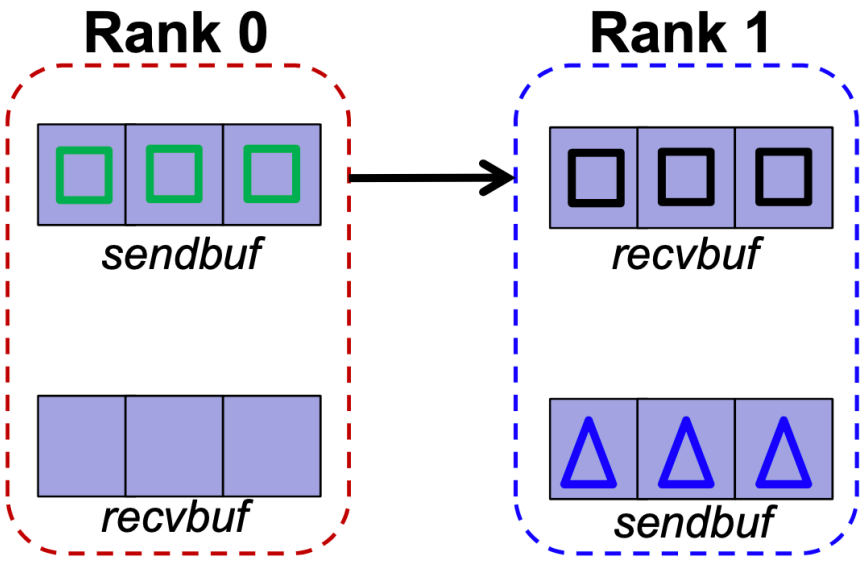
- 从进程0向进程1发送消息时，有四种MPI子例程组合可供选择

✓阻塞发送和阻塞接收

✓非阻塞发送和阻塞接收

✓阻塞发送和非阻塞接收

✓非阻塞发送和非阻塞接收



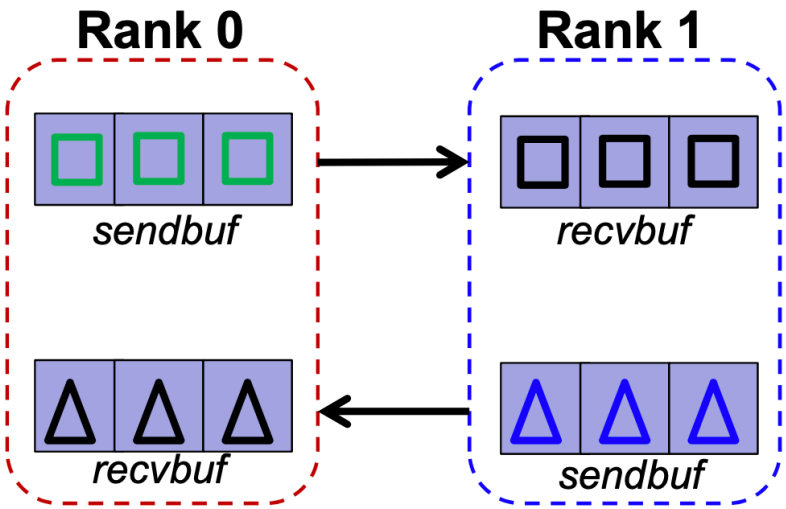
双向通信

- 当两个进程相互交换数据时，主要有3种情况

✓Case 1：两个进程调用send，然后receive

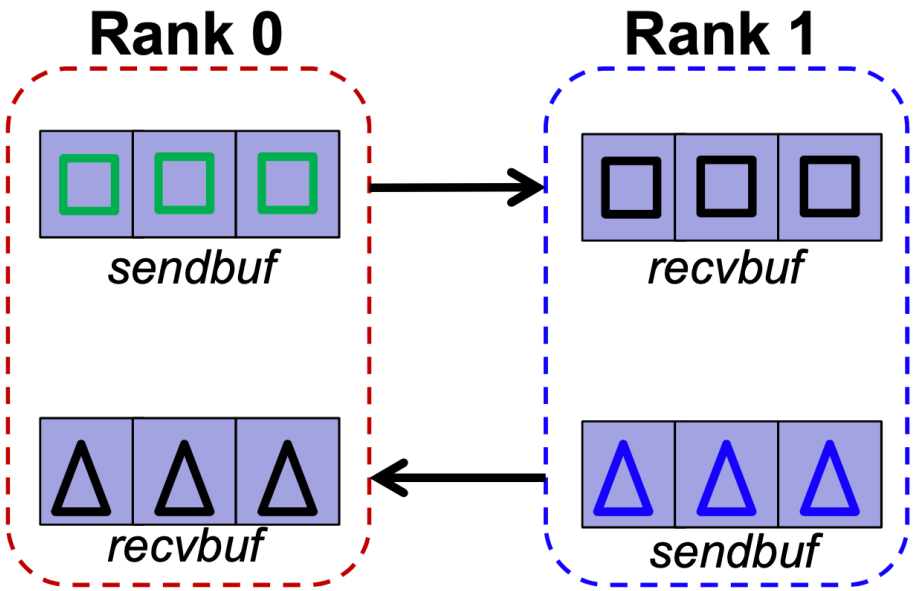
✓Case 2：两个进程调用receive，然后send

✓Case 3：一个进程调用send然后receive，另一个进程则相反



双向通信 – 死锁

- 在双向通信中，我们需要关注死锁问题
- 当死锁发生时，所有进程会阻塞不前
- 死锁发生的原因：
  - ✓顺序错误的发送和接收调用
  - ✓系统缓存太小



情况1：先发送后接收

```
IF (myrank==0) THEN
  CALL MPI_SEND(sendbuf, …)
  CALL MPI_RECV(recvbuf, …)
ELSEIF (myrank==1) THEN
  CALL MPI_SEND(sendbuf, …)
  CALL MPI_RECV(recvbuf, …)
ENDIF
```

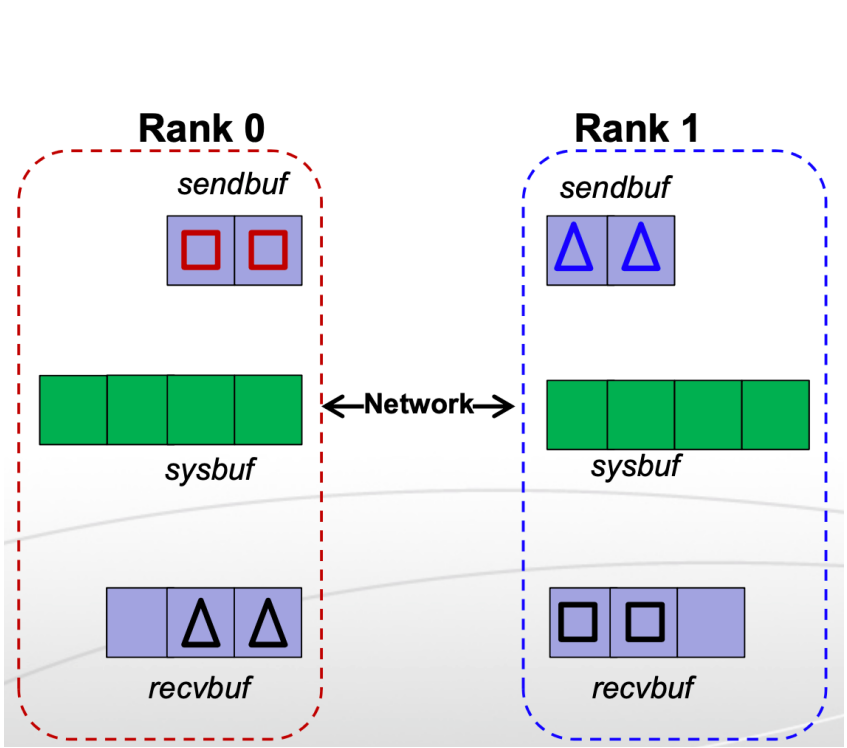
```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, …, ireq, …)
  CALL MPI_WAIT(ireq, …)
  CALL MPI_RECV(recvbuf, …)
ELSEIF (myrank==1) THEN
  CALL MPI_ISEND(sendbuf, …, ireq, …)
  CALL MPI_WAIT(ireq, …)
  CALL MPI_RECV(recvbuf, …)
ENDIF
```

MPI_ISEND后面紧跟MPI_WAIT在逻辑上等同于MPI_SEND

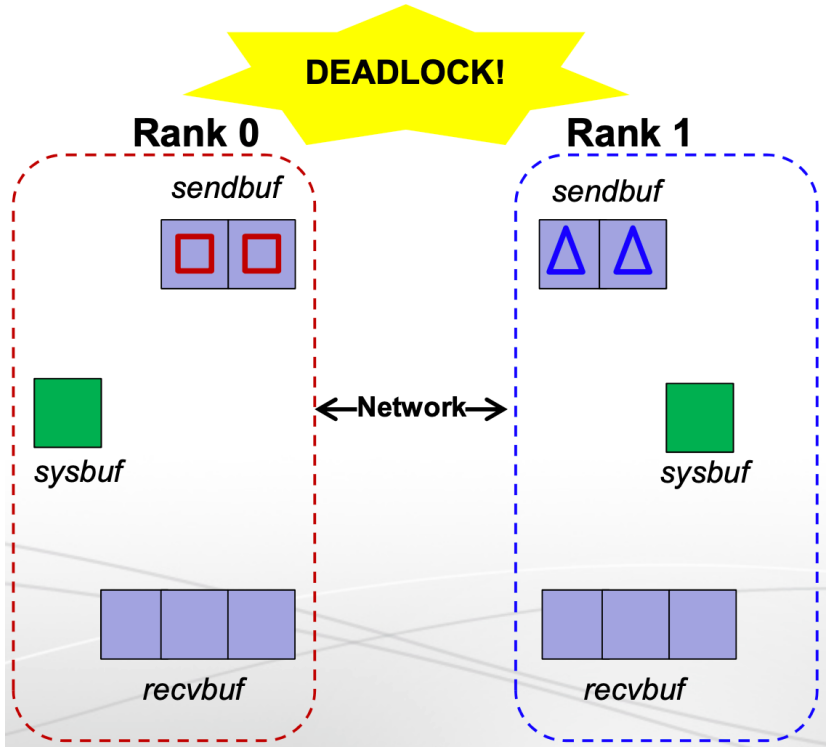
情况1：先发送后接收 (cont'd)

- 如果系统缓冲区大于发送缓冲区，会发生什么？

- 如果系统缓冲区小于发送缓冲区，会发生什么？



系统缓冲区大于发送缓冲区



系统缓冲区小于发送缓冲区

情况1：先发送后接收 (cont'd)

- 下面的程序是否会出现死锁？

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, …, ireq, …)
  CALL MPI_RECV(recvbuf, …)
  CALL MPI_WAIT(ireq, …)
ELSEIF (myrank==1) THEN
  CALL MPI_ISEND(sendbuf, …, ireq, …)
  CALL MPI_RECV(recvbuf, …)
  CALL MPI_WAIT(ireq, …)
ENDIF
```

- 不会，因为
  - ✓程序立即从MPI_ISEND返回并开始从其他进程接收数据
  - ✓同时，数据传输已完成，为MPI_RECV设置MPI_WAIT并不会导致死锁

情况2：先接收后发送

- 下面的程序是否会出现死锁？

```
IF (myrank==0) THEN
  CALL MPI_RECV(recvbuf, …)
  CALL MPI_SEND(sendbuf, …)
ELSEIF (myrank==1) THEN
  CALL MPI_RECV(recvbuf, …)
  CALL MPI_ISEND(sendbuf, …)
ENDIF
```

会，不管系统缓存有多大

- 如果我们用MPI_ISEND替换MPI_SEND呢？

死锁依旧会发生

情况2：先接收后发送 (cont'd)

- 下面的程序是否会出现死锁？

```
IF (myrank==0) THEN
  CALL MPI_IRECV(recvbuf, …, ireq, …)
  CALL MPI_SEND(sendbuf, …)
  CALL MPI_WAIT(ireq, …)
ELSEIF (myrank==1) THEN
  CALL MPI_IRECV(recvbuf, …, ireq, …)
  CALL MPI_SEND(sendbuf, …)
  CALL MPI_WAIT(ireq, …)
ENDIF
```

不会，程序可以安全执行

情况3：两个进程操作相反

- 下面的程序是否会出现死锁？

```
IF (myrank==0) THEN
   CALL MPI_SEND(sendbuf, …)
   CALL MPI_RECV(recvbuf, …)
ELSEIF (myrank==1) THEN
   CALL MPI_RECV(recvbuf, …)
   CALL MPI_SEND(sendbuf, …)
ENDIF
```

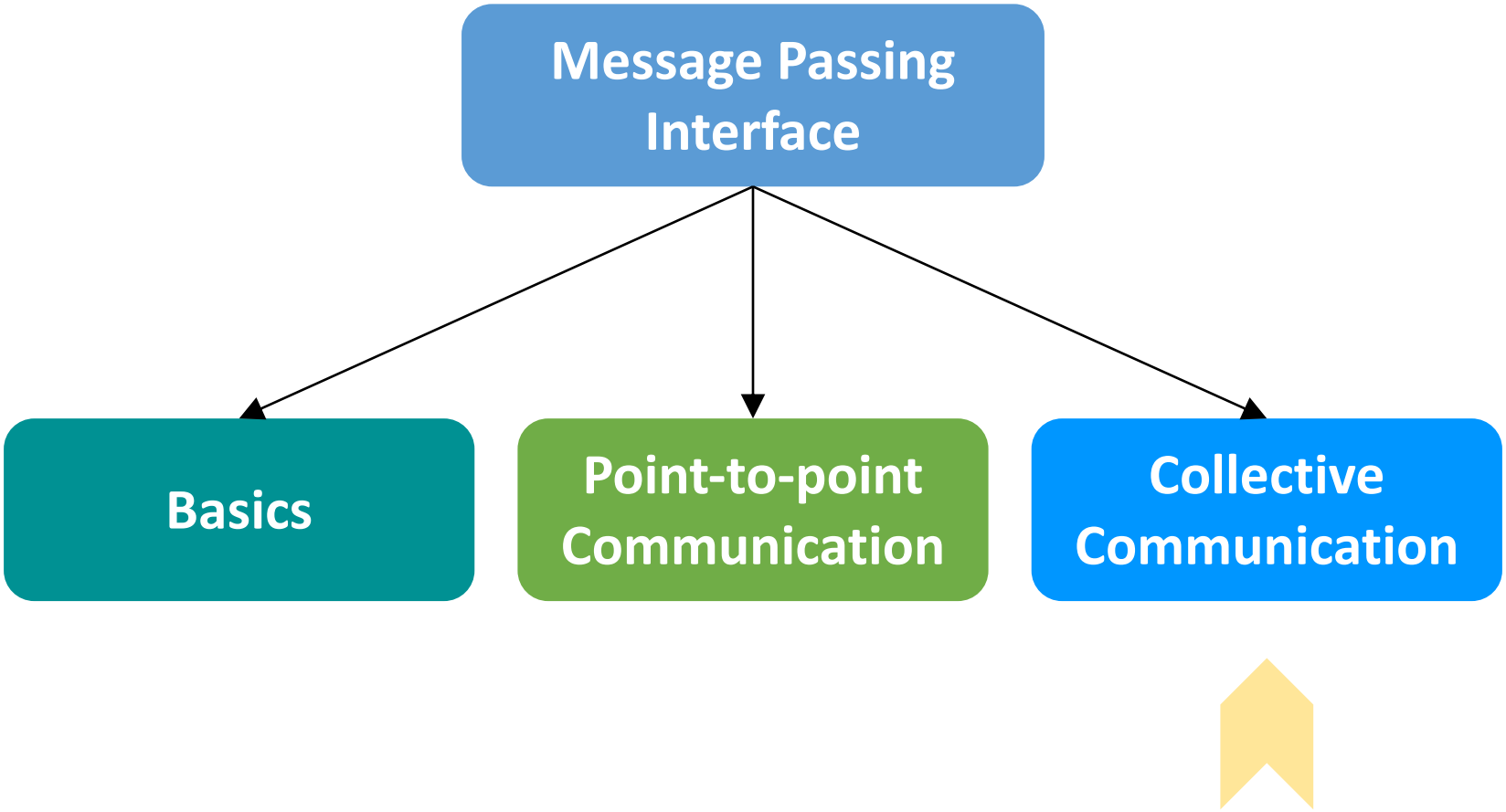
若两个程序以相反的顺序调用发送和接收，
则总是安全的，不管是否为阻塞操作

推荐调用顺序

- 考虑到前面的情况、性能和死锁问题，建议使用以下代码

```
IF (myrank==0) THEN
  CALL MPI_ISEND(sendbuf, …, ireq1, …)
  CALL MPI_IRECV(recvbuf, …, ireq2, …)
ELSEIF (myrank==1) THEN
  CALL MPI_ISEND(sendbuf, …, ireq1, …)
  CALL MPI_IRECV(recvbuf, …, ireq2, …)
ENDIF
  CALL MPI_WAIT(ireq1, …)
  CALL MPI_WAIT(ireq2, …)
```

消息传递接口


Message Passing Interface
Basics
Point-to-point Communication
Collective Communication

集体通信 (collective communication)

- 集体通信允许一组进程之间交换数据
- 它必须涉及通信器范围内的所有进程
- 集体通信例程中的通信器参数应指定通信中涉及的进程
- 因此，程序员有责任确保通信器内的所有进程参与集体操作

集体通信的模式

- 集体通信通常包含下列模式
  1. Broadcast
  2. Scatter
  3. Gather
  4. Allgather
  5. Alltoall
  6. Reduce
  7. Allreduce
  8. Scan
  9. Reducescatter

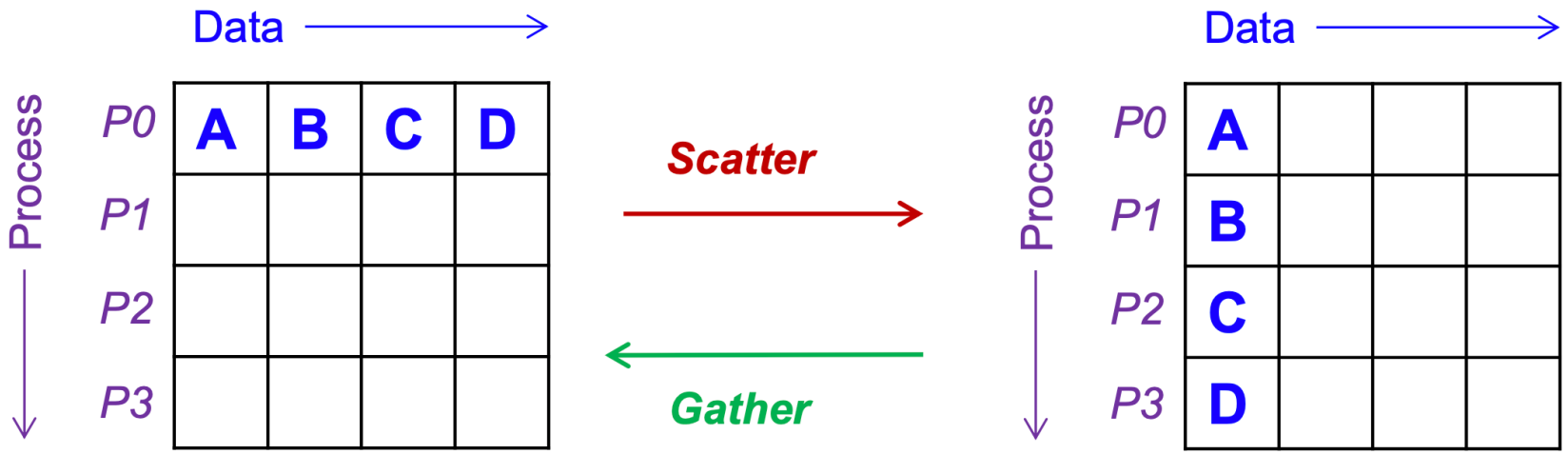
Broadcast

- Broadcast从rank为root的进程向组中的所有其他进程发送消息

Scatter和Gather

- Scatter将不同的消息从单个源进程分发到组中的每个进程

- Gather将组中每个进程的不同消息收集到单个目标进程



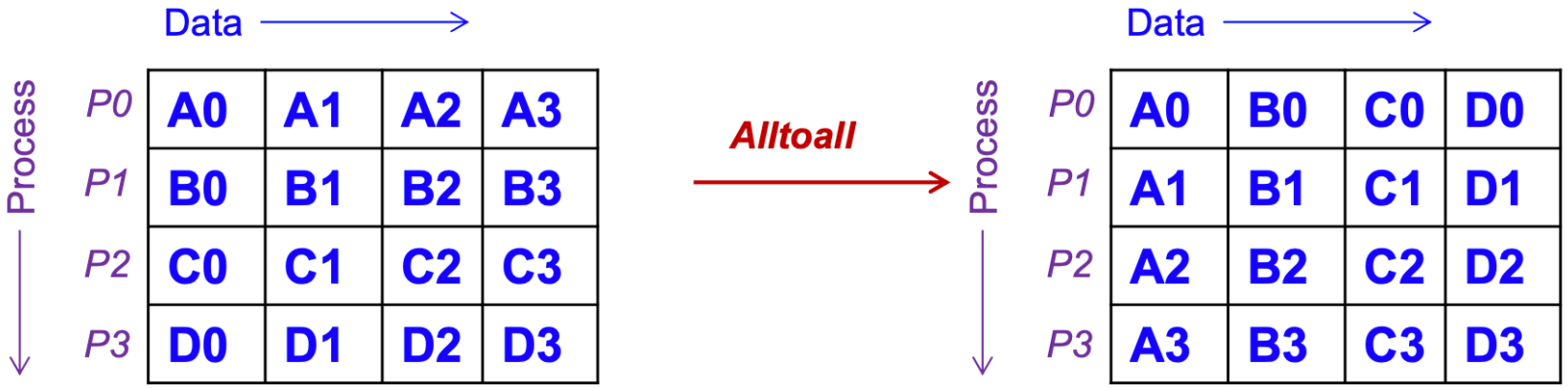
All Gather

- Allgather收集所有进程的数据并将其分发给所有进程

- 等同于组中的每个任务在组中执行一对一的广播操作

All to All

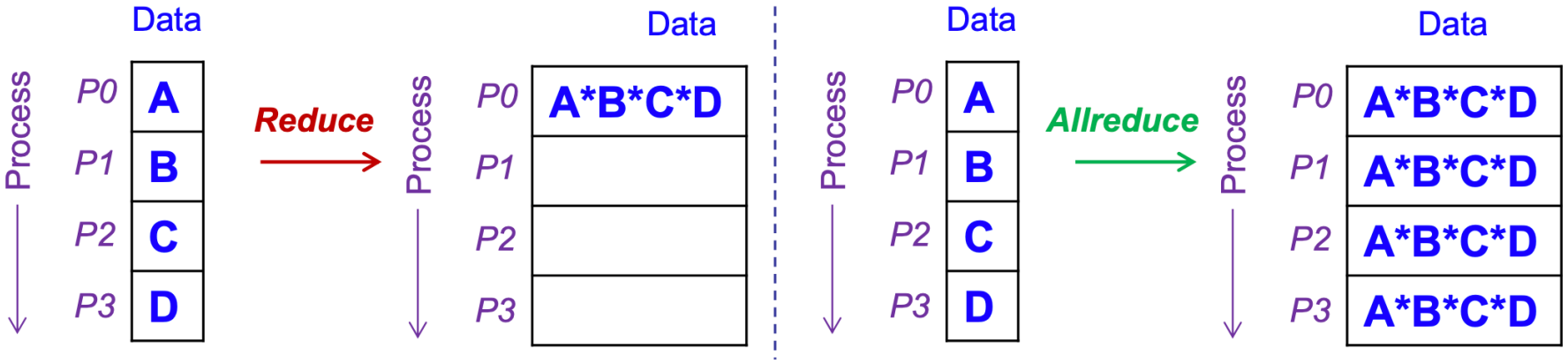
- 组中的每个进程都执行Scatter操作，按索引顺序向组中的所有进程发送不同的消息



Reduce and All Reduce

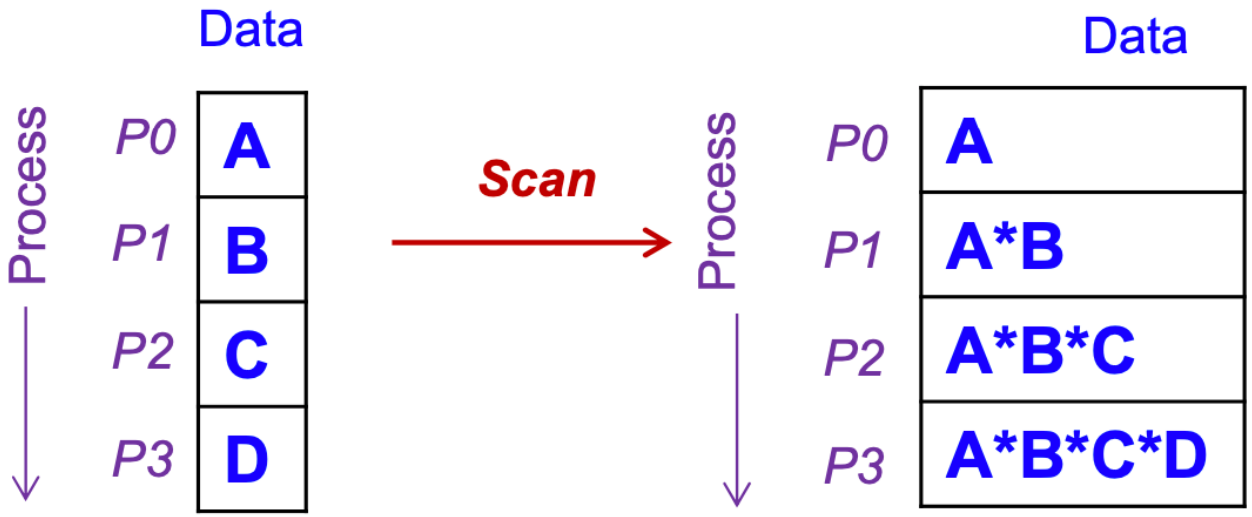
- Reduce对组中的所有任务应用reduce操作，并将结果放在一个任务中

- Allreduce应用reduce操作，并将结果放置在组中的所有任务中。这相当于Reduce后跟Broadcast



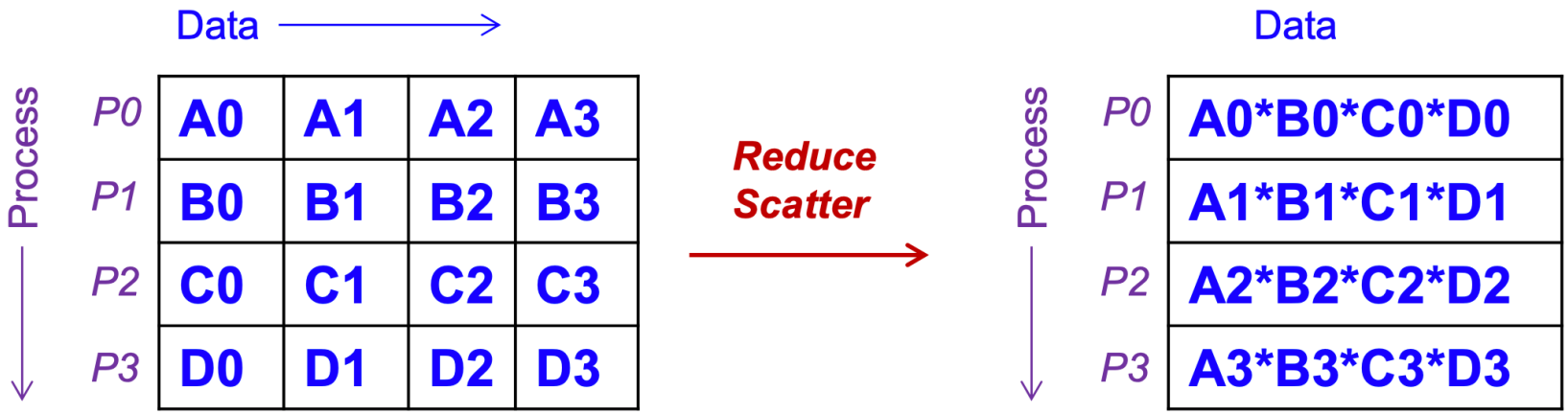
Scan

- Scan逐步对集合中的进程使用reduce操作



Reduce Scatter

- Reduce Scatter首先应用reduce操作然后使用scatter将结果分散到不同进程



云计算中的集体通信应用 (cont'd)

- Broadcast

Broadcast的一个常见应用是将软件更新或配置更改传播到集群中的多个虚拟机。一个中央控制器可以使用广播消息，将更新或更改同时发送给集群中的所有虚拟机，而无需向每个虚拟机发送单独的消息。

云计算中的集体通信应用 (cont'd)

- Gather

在数据分析应用中，Gather可以用于从多个来源收集和聚合数据，例如传感器或数据库。例如，一个基于云的天气监测系统可能从多个天气站收集温度、气压和湿度数据，然后聚合数据以产生天气预报。

云计算中的集体通信应用 (cont'd)

- Scatter

Scatter可以用于在云计算中将数据或任务分发给多个工作节点进行并行处理。例如，一个机器学习算法可以将大型数据集分散到多个工作节点上，每个节点处理数据子集，并将结果发送回主节点以进行聚合。

云计算中的集体通信应用 (cont'd)

- Reduce

在许多科学计算应用中，Reduce可以用于将多个模拟或实验的部分结果组合成最终结果。例如，计算流体力学模拟可以使用多个具有不同初始条件的模拟，然后将结果归约以产生最终解。

云计算中的集体通信应用 (cont'd)

- All to All

全对全通信在云计算中的一个常见用例是分布式数据挖掘。多个工作节点可以分别分析大型数据集的子集，然后与所有其他节点交换信息以提高整体分析的准确性。

中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

谢谢

陈壮彬

软件工程学院

https://zbchern.github.io/sse316.html