



1/248

# 高性能计算系统

迟学斌(081201M05004H, chi@sccas.cn)

中国科学院计算机网络信息中心

2025年2-6月



Back

Close

# 目 录



2/248

## 一、并行计算综述 10

1.1	什么是并行计算 . . . . .	12
1.2	并行计算机的发展 . . . . .	16
1.3	为什么需要并行计算 . . . . .	29
1.4	高性能计算环境 . . . . .	31
1.5	国际上千万亿次计算的应用问题 . . . . .	36
1.6	习题 . . . . .	45

## 二、并行计算机体系结构 46

2.1	网络的分类 . . . . .	47
2.2	网络的基本概念 . . . . .	48



Back

Close



2.3	间接网络 . . . . .	50
2.4	直接网络 . . . . .	55
2.5	习题 . . . . .	58

### 三、并行计算的基本概念 **59**

3.1	并行计算机系统-MPP . . . . .	60
3.2	并行计算机系统-SMP . . . . .	61
3.3	并行计算机系统-Cluster . . . . .	62
3.4	并行计算机系统-Hybrid . . . . .	63
3.5	并行计算机系统的分类 . . . . .	64
3.6	并行计算的程序结构 . . . . .	65
3.7	并行计算的基本定义 . . . . .	66
3.8	习题 . . . . .	68



## 四、矩阵乘并行计算

69

4.1	矩阵卷帘 (wrap) 存储方式 . . . . .	71
4.2	串行矩阵乘法 . . . . .	73
4.3	行列分块算法 . . . . .	74
4.4	行行分块算法 . . . . .	76
4.5	列行分块算法 . . . . .	78
4.6	列列分块算法 . . . . .	80
4.7	Cannon算法 . . . . .	82
4.8	修正Cannon算法 . . . . .	85
4.9	习题 . . . . .	89

## 五、线性代数方程组的并行求解

90

5.1	串行 $LU$ 分解算法 . . . . .	91
5.2	分布式系统的并行 $LU$ 分解算法 . . . . .	94



4/248





5.3	三角方程组的并行解法 . . . . .	96
5.4	经典迭代法-Jacobi . . . . .	99
5.5	经典迭代法-Gauss-Seidel . . . . .	101
5.6	习题 . . . . .	104

## 六、FFT并行算法 105

6.1	一维串行FFT算法 . . . . .	109
6.2	二维串行FFT算法 . . . . .	113
6.3	FFT实现技术 . . . . .	114
6.4	习题 . . . . .	125

## 七、MPI并行程序设计 126

7.1	并行程序类型 . . . . .	127
7.2	MPI并行程序的基本结构 . . . . .	128





7.3	MPI并行环境管理函数	130
7.4	MPI通信子操作	132

## 八、点到点通信函数 **136**

8.1	阻塞式SEND和RECV	137
8.2	合成函数SENDRECV	152
8.3	消息查询函数	156
8.4	非阻塞式ISEND和IRECV	158
8.5	消息请求完成函数	160
8.6	消息请求检查函数	162
8.7	持久通讯函数SEND_INIT和RECV_INIT	164
8.8	高维进程	168
8.9	习题	172



Back

Close

## 九、自定义数据类型 173

9.1	CONTIGUOUS数据类型	174
9.2	数据类型辅助函数	176
9.3	VECTOR数据类型	178
9.4	INDEX数据类型	181
9.5	STRUCT数据类型	184
9.6	特殊数据类型与绝对原点	186
9.7	MPI的数据打包与拆包	190
9.8	习题	192

## 十、MPI聚合通信 194

10.1	障碍同步MPI_Barrier	195
10.2	广播MPI_Bcast	196
10.3	收集MPI_Gather	197



7/248



Back

Close



10.4	散播MPI_Scatter . . . . .	201
10.5	全交换MPI_Alltoall . . . . .	204
10.6	习题 . . . . .	208

## 十一MPI归约操作 209

11.1	归约MPI_Reduce . . . . .	210
11.2	运算种类与可用数据类型 . . . . .	212
11.3	前缀MPI_Scan . . . . .	215
11.4	归约散播MPI_Reduce_scatter . . . . .	216
11.5	自定义运算 . . . . .	218

## 十二MPI组操作 220

12.1	进程组的创建 . . . . .	221
12.2	进程组管理 . . . . .	224





十三并行程序实例227

13.1	$\pi$ 值近似计算程序 . . . . .	228
13.2	Cannon算法实现程序 . . . . .	235
13.3	数据广播并程序 . . . . .	239
13.4	数据分散并程序 . . . . .	244

参考文献248





10/248

# 一、并行计算综述

需要掌握三方面的知识:

- 高性能计算机的最新发展状况
- 并行计算的基本概念与并行算法
- 并行程序实现技术与方法

考核方式: 出勤率、课堂发言、口头报告、课堂开卷考试

课后习题: 需认真完成, 以便检验对于相关知识的掌握情况

If you don't study hard, you'll bite yourself in the ass and fail the exam. Are you game?



Back

Close

科学技术是第一生产力，人才是第一资源，创新是引领发展的第一动力。

1.1、什么是并行计算

1.2、并行计算机的发展

1.3、为什么需要并行计算

1.4、中科院高性能计算环境

1.5、国际上千万亿次计算的应用问题



11/248



Back

Close



# 什么是并行计算

并行计算（parallel computing）是指，在并行机上，将一个应用问题分解成多个子任务，将每个子任务分配给不同的处理器，各个处理器之间相互协同，并行地执行这些子任务，从而达到提高求解速度，或者完成求解大规模应用问题的目的。

开展并行计算，必须具备三个基本条件：

1. 并行机。并行机至少包含两台或两台以上处理机，这些处理机通过互连网络相互连接，相互通信。
2. 应用问题必须具有并行度。也就是说，应用可以分解为多个子任务，这些子任务可以并行地执行。将一个应用分解为多个子任务的过程，

[Back](#)[Close](#)



称为并行算法的设计。

3. 并行编程。在并行机提供的并行编程环境上，具体实现并行算法，编制并行程序，并运行该程序，从而达到并行求解应用问题的目的。

### 例子 1 并行计算求和问题

$$S = \sum_{i=0}^{n-1} a_i \quad (1.1)$$

假设  $n = 4 \times m$ ，记  $S_0 = \sum_{i=0}^{m-1} a_i$ ， $S_1 = \sum_{i=m}^{2m-1} a_i$ ， $S_2 = \sum_{i=2m}^{3m-1} a_i$ ， $S_3 = \sum_{i=3m}^{n-1} a_i$ ，则有  $S = S_0 + S_1 + S_2 + S_3$ 。

因此，计算  $S$  可以并行执行，亦即  $S_i$ ， $i = 0, 1, 2, 3$  可以同时计算出。

进一步可以同时计算  $S_{00} = S_0 + S_1$ ，和  $S_{01} = S_2 + S_3$ 。最后再计算  $S = S_{00} + S_{01}$



Back

Close



## 例子 2 流水线并行

汽车生产线，自助餐等。所有这些并行计算问题，其根本是如何设计并行计算方法，下面的示例告诉我们算法的重要性。

## 例子 3 计算 $x^n$ 的复杂性。

首先我们需要给出计算 $x^n$ 的算法，针对算法给出计算复杂性。对于一个问题来说，通常其计算复杂性是确定的，因此我们在算法设计时就应该寻找最佳的计算复杂性算法，从而达到快速计算之目的。那么，怎样计算 $x^n$ ？比如 $n = 13$ ，可以计算 $x^2 = x \times x$ ， $x^4 = x^2 \times x^2$ ， $x^8 = x^4 \times x^4$ ， $x^5 = x^4 \times x$ ，最后计算 $x^{13} = x^8 \times x^5$ ，一共需要5次计算。我们的算法就是基于这种方式，按照2的幂次进行计算。对于给定的一个整数 $n$ ，其2进制表示可以记为： $n = 2^k + a_{k-1}2^{k-1} + \cdots + a_12 + a_0$ ，其中 $a_i = 0$ 或者 $a_i = 1$ ， $k = \log_2 n$ 。据此我们可以给出计算方法如下：

## 算法 1 快速计算 $x^n$ 的算法



Back

Close



15/248

```
y=1;
for i=0 to k-1 do
    if a[i] = 1 do
        y=y*x;
    end{if}
    x = x*x;
end{for}
y=y*x;
```

因此我们可以得出, 当所有的 $a_i$ 均非零时, 这个算法的计算复杂性为 $2k + 1$ , 即 $2\log_2 n + 1$ , 通常亦称之为计算复杂性为 $O(\log n)$ 。通过此算法我们可以看到, 一个简单的问题, 其计算方法可以是复杂的, 并且此问题不适合并行计算。



Back

Close



# 并行计算机的发展

并行计算机从70年代开始，到80年代蓬勃发展和百家争鸣，90年代体系结构框架趋于统一，近10年来机群技术的快速发展，并行机技术日趋成熟。本节以时间为线索，简介并行计算机发展的推动力和各个阶段，以及各类并行机的典型代表和它们的主要特征。

1972年，世界上诞生了第一台并行计算机*ILLIAC IV*，它含32个处理单元，环型拓扑连接，每台处理机拥有局部内存，为SIMD类型机器。对大量流体力学程序，*ILLIAC IV*获得了2-6倍于当时性能最高的*CDC 7600*机器的速度。70年代中后期，出现了*Cray-1*为代表的向量计算机，现在这种计算机也仍在使用，世界上先进的高性能计算机系统日本的地球模拟器的处理器就是采用向量机。



Back

Close





进入80年代, *MPP*并行计算机开始大量涌现, 这种类型的计算机早期的典型代表有 *iPSC/860*, *nCUBE-2*, *Meiko*。我国也在这个时期, 研制成功了向量计算机银河-1。

从90年代开始, 主流的计算机仍然是*MPP*, 例如: *IBM SP2*, *Cray T3D*, *Cray T3E*, *Intel Paragon XP/S*, 中科院计算所“曙光1000”等。

目前主要采用的计算机系统结构为:

1. 机群系统 (参见 3.3)
2. *DSM*, *SMP*系统 (参见 3.2)
3. *MPP*系统 (参见 3.1)
4. 星群系统
5. *GPU*与*CPU*混合集群系统 (天河一号)



Back

Close



这里的星群实际上也是一种机群，它包含了异构机群，每个计算结点可以是不同结构的SMP、MPP等构成。在近年的高性能计算机系统中，以混合结构完成的系统取得了飞速发展。这主要得益于混合结构能够取得很高的峰值速度，若在实际应用中能够发挥作用，需要进行大量艰苦的努力。以下表1.10、1.9、1.8、1.7、1.6、1.5、1.4、1.3、1.2分别是2009年11月份、2011年11月份、2013年11月份、2014年11月份、2015年11月份、2016年11月份、2018年11月份、2020年11月份、2022年11月份世界500强计算机系统的前10名情况。表1.1是最新世界500强前10名的计算机系统。



Back

Close



19/248



Back

Close

**表 1.1 2023年11月TOP500计算机系统的前10名(TFLOPS)**

计算机系统	核数	性能	功耗
USA Frontier - HPE Cray EX235a, AMD 3rd EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	8,699,904	1,194.00	22,703
USA Aurora - HPE Cray EX - Xeon CPU Max 9470 52C 2.4GHz, Intel GPU Max, Slingshot-11	4,742,808	585.34	24,687
USA Eagle - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, IB NDR	1,123,200	561.20	
Japan Sc Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	7,630,848	442.01	29,899
Finland LUMI - HPE Cray EX235a, AMD 3rd EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	2,752,704	379.70	7,107
Italy Leonardo - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, A100, HDR100 Infiniband	1,824,768	238.70	7,404
USA Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100	2,414,592	148,600	10,096
Spain MareNostrum 5 ACC - Xeon Platinum 8460Y+ 40C 2.3GHz, NVIDIA H100, IB NDR200	680,960	138.20	2,560
USA Eos NVIDIA DGX SuperPOD - NVIDIA DGX H100, Xeon Platinum 8480C 56C 3.8GHz, IB NDR400	485,888	121.40	
USA Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100	1,572,480	94,640	7,438



20/248



# 表 1.2 2022年11月TOP500计算机系统的前10名(TFLOPS)

计算机系统	核数	性能	功耗
USA Frontier - HPE Cray EX235a, AMD 3rd EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	8,730,112	1,102,000	21,100
Japan Sc Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	7,299,072	415,530	28,334.5
Finland LUMI - HPE Cray EX235a, AMD 3rd EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	2,220,288	309,100	6,015.77
Italy Leonardo - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, A100, HDR100 Infiniband	1,463,616	174,700	5,610
USA Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100	2,414,592	148,600	10,096
USA Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100	1,572,480	94,640	7,438.28
China Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz	10,649,600	93,014.6	15,371
Perlmutter - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10	761,856	70,870	2,589
USA Selene - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband	555,520	63,460	2,646
China Tianhe-2 - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000	4,981,760	61,444.5	18,482



21/248



Back

Close

表 1.3 2020年11月TOP500计算机系统的前10名(TFLOPS)

计算机系统	核数	性能	功耗
Japan Sc Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	7,299,072	415,530	28,334.5
USA Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100	2,414,592	148,600	10,096
USA Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100	1,572,480	94,640	7,438.28
China Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz	10,649,600	93,014.6	15,371
USA Selene - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband	555,520	63,460	2,646
China Tianhe-2 - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000	4,981,760	61,444.5	18,482
Germany JUWELS Booster Module - Bull Sequana X-H2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation Cluster-Suite	449,280	44,120	1,764
Italy HPC5 - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband	669,760	35,450	2,252
USA Frontera - Dell C6420, Xeon Platinum 8280 28C 2.7GHz	448,448	23,516.4	
Canada ALCOR - Dell C6420, Xeon Gold 6145 24C 2.3GHz, NVIDIA V100, Mellanox HDR Infiniband	672,528	33,433	



22/248



Back

Close

表 1.4 2018年TOP500计算机系统的前10名(TFLOPS)

计算机系统	核数	性能	功耗
Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM	2,397,824	143,500.0	9,783
Sierra - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM	1,572,480	94,640.0	7,438
China Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	15,371
China Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT	4,981,760	61,444.5	18,482
Switzerland Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100	387,872	21,230.0	2,384
Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc.	979,072	20,158.7	7,578
Japan AI Bridging Cloud Infrastructure (ABCI) - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR , Fujitsu	391,680	19,880.0	1,649
Germany SuperMUC-NG - ThinkSystem SD530, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path , Lenovo	305,856	19,476.6	



23/248



Back

Close

# 表 1.5 2016年TOP500计算机系统的前10名(TFLOPS)

计算机系统	核数	性能	功耗
China Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	15,371
China Tianhe-2 (MilkyWay-2), Intel Xeon E5-2692 12C 2.20GHz, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	17,808
Titan - Cray XK7 , Opteron 6274 16C 2.20GHz, NVIDIA K20x Cray Inc.	560,640	17,590.0	8,209
Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, IBM	1,572,864	17,173.2	7,890
Cori - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect, Cray Inc.	622,336	14,014.7	3,939
Japan Oakforest-PACS - PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path, Fujitsu	556,104	13,554.6	2,719
Japan K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	12,660
Switzerland Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100	206,720	9,779.0	1,312
Mira - BlueGene/Q, Power BQC 16C 1.60GHz,IBM	786,432	8,586.6	3,945
Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect	301,056	8,100.9	4,233



# 表 1.6 2015年TOP500计算机系统的前10名(TFLOPS)

计算机系统	核数	性能	功耗
China Tianhe-2 (MilkyWay-2), Intel Xeon E5-2692 12C 2.20GHz, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	17,808
Titan - Cray XK7 , Opteron 6274 16C 2.20GHz, NVIDIA K20x Cray Inc.	560,640	17,590.0	8,209
Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, IBM	1,572,864	17,173.2	7,890
Japan K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	12,660
Mira - BlueGene/Q, Power BQC 16C 1.60GHz,IBM	786,432	8,586.6	3,945
Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect	301,056	8,100.9	4,233
Switzerland Piz Daint - Cray XC30, Xeon E5-2670 8C 2.60GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	2,325
Germany Hazel Hen - Cray XC40, Xeon E5-2680v3 12C 2.5GHz, Aries interconnect, Cray Inc.	185,088	5,640.2	3,615
Saudi Arabia Shaheen II - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect, Cray Inc.	196,608	5,537.0	2,834
Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.70GHz, Infiniband FDR, Intel Xeon Phi SE10P Del-1	462,462	5,168.1	4,510





25/248



# 表 1.7 2014年TOP500计算机系统的前10名(TFLOPS)

计算机系统	核数	性能	功耗
China Tianhe-2 (MilkyWay-2), Intel Xeon E5-2692 12C 2.20GHz, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	17,808
Titan - Cray XK7 , Opteron 6274 16C 2.20GHz, NVIDIA K20x Cray Inc.	560,640	17,590.0	8,209
Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, IBM	1,572,864	17,173.2	7,890
Japan K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	12,660
Mira - BlueGene/Q, Power BQC 16C 1.60GHz,IBM	786,432	8,586.6	3,945
Switzerland Piz Daint - Cray XC30, Xeon E5-2670 8C 2.60GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	2,325
Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.70GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	4,510
Germany JUQUEEN - BlueGene/Q, Power BQC 16C 1.60GHz, Custom Interconnect IBM	458,752	5,008.9	2,301
Vulcan - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393,216	4,293.3	1,972
Cray CS-Storm, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, Nvidia K40 Cray Inc.	72,800	3,577.0	1,499



26/248



Back

Close

**表 1.8 2013年TOP500计算机系统的前10名(TFLOPS)**

计算机系统	核数	性能	功耗
Tianhe-2 (MilkyWay-2), Intel Xeon E5-2692 12C 2.20GHz, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	17,808
Titan - Cray XK7 , Opteron 6274 16C 2.20GHz, NVIDIA K20x Cray Inc.	560,640	17,590.0	8,209
Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, IBM	1,572,864	17,173.2	7,890
K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	12,660
Mira - BlueGene/Q, Power BQC 16C 1.60GHz,IBM	786,432	8,586.6	3,945
Switzerland Piz Daint - Cray XC30, Xeon E5-2670 8C 2.60GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	2,325
Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.70GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	4,510
Germany JUQUEEN - BlueGene/Q, Power BQC 16C 1.60GHz, Custom Interconnect IBM	458,752	5,008.9	2,301
Vulcan - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393,216	4,293.3	1,972
Germany SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM	147,456	2,897.0	3,423



表 1.9 2011年TOP500计算机系统的前10名(TFLOPS)

计算机系统	国家	年份	核数	性能
K computer, SPARC64 VIIIfx 2.0GHz Tofu interconnect	Japan	2011	705024	10510
NUDT TH-1A, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C	China	2010	186368	2566
Cray XT5-HE Opteron Six Core 2.6 GHz	USA	2009	224162	1759
Dawning TC3600 Blade, X5650 Nvidia Tesla C2050 GPU	China	2010	120640	1271
HP ProLiant SL390s G7 X5670 Nvidia GPU, Linux/Win	Japan	2010	73278	1192
Cray XE6, Opteron 6136 8C	USA	2011	142272	1110
SGI Altix ICE 8200/8400EX	USA	2011	111104	1088
Cray XE6 12-core 2.1 GHz	USA	2010	153408	1054
Bull bullx super-node S6010/S6030	France	2010	138368	1050
BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2Ghz/Opteron DC 1.8GHz, Voltaire Infiniband	USA	2009	122400	1042



28/248



表 1.10 2009年TOP500计算机系统的前10名(GFLOPS)

计算机系统	国家	年份	核数	性能
Cray XT5-HE Opteron Six Core 2.6 GHz	USA	2009	224162	1759000
BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2Ghz/Opteron DC 1.8GHz, Voltaire Infiniband	USA	2009	122400	1042000
Cray XT5-HE 6 Core 2.6 GHz	USA	2009	98928	831700
Blue Gene/P Solution	Germany	2009	294912	825500
NUDT TH-1 Cluster, Xeon E5540/E5450, ATI Radeon HD 4870 2,Infiniband	China	2009	71680	563100
SGI Altix ICE 8200EX, Xeon QC 3.0/2.93 GHz	USA	2009	56320	544300
eServer Blue Gene Solution	USA	2007	212992	478200
Blue Gene/P Solution	USA	2007	163840	450300
SunBlade x6420, Opteron QC 2.3 Ghz, Infiniband	USA	2008	62976	433200
Sun Blade x6275, Xeon X55xx 2.93 Ghz, Infiniband	USA	2009	41616	423900



# 为什么需要并行计算

全球气象预报中天气预报模式要求在24小时内完成48小时天气预报数值模拟,此时,至少需要计算635万个网格点,内存需求大于1TB,计算性能要求高达25万亿次/秒。又如,美国在1996年开始实施ASCI计划,要求分四个阶段,逐步实现万亿次、十万亿次、30万亿次和100万亿次的大规模并行数值模拟,实现全三维、全物理过程、高分辨率的核武器数值模拟。除此之外,在天体物理、流体力学、密码破译、海洋大气环境、石油勘探、地震数据处理、生物信息处理、新药研制、湍流直接数值模拟、燃料燃烧、工业制造、图像处理等领域,以及大量的基础理论研究领域,存在计算挑战性问题,均需要并行计算的技术支持。

上个世纪90年代,HPCC计划提出的背景是一大批巨大挑战性问题



Back

Close



30/248

题需要解决，其中包括：天气与气候预报，分子、原子与核结构，大气污染，燃料与燃烧，生物学中的大分子结构，新型材料特性，国家安全等有关问题。而近期内要解决的问题包含：磁记录技术，新药研制，高速城市交通，催化剂设计，燃料燃烧原理，海洋模型模拟，臭氧层空洞，数字解剖，空气污染，蛋白质结构设计，金星图象分析和密码破译技术。



Back

Close



# 高性能计算环境

## 国际高性能计算环境现状



图 1.1 主要代表



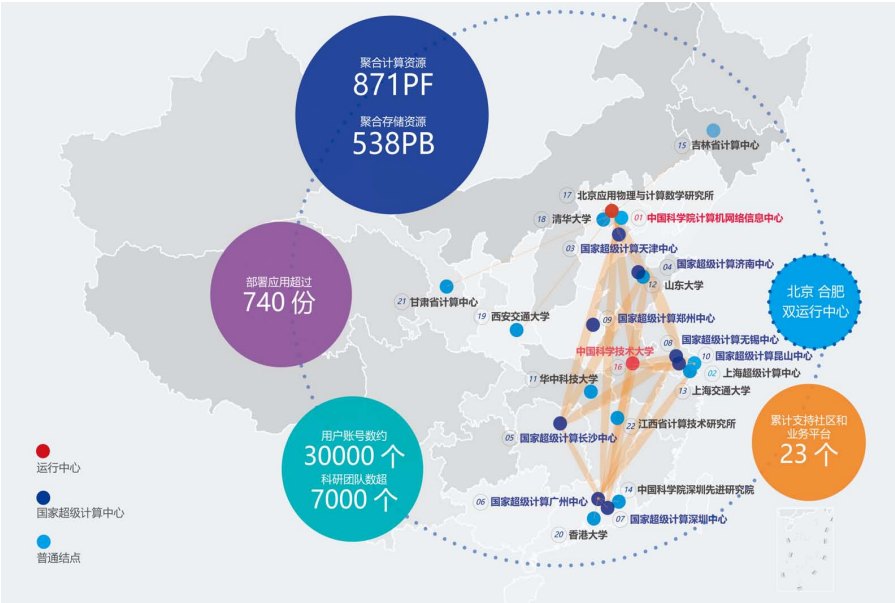
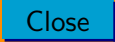


图 1.2 区域分布情况





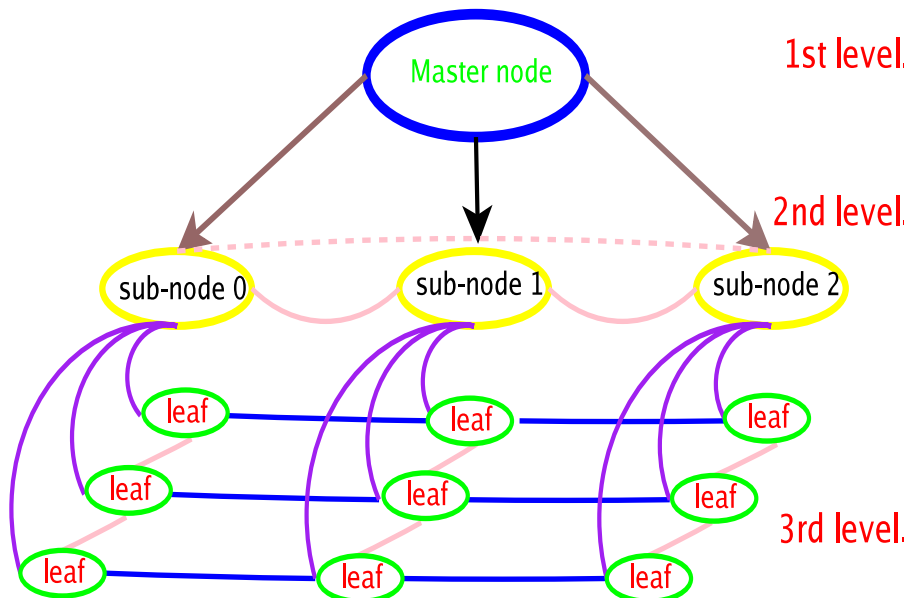


图 1.3 三层网络系统结构



# 总中心千万亿次计算机系统

元超级计算机系统是混合结构高性能机群系统, 其峰值性能 $2PFLOPS$ , 采用 $CPU+GPU$ , 以及 $CPU+MIC$ ; 中科先导1号系统, 类似 $CPU+GPU$ 的异构系统, 峰值性能 $200PFLOPS$ 。

## 分中心计算环境

每个分中心计算能力 $\geq 10TFLOPS$ , 分布如下:

- 兰州分中心: 寒区旱区环境与工程研究所
- 大连分中心: 大连化学物理研究所
- 合肥分中心: 合肥物质科学研究院和中国科技大学
- 昆明分中心: 昆明植物研究所/昆明动物研究所
- 青岛分中心: 海洋研究所/生物能源与过程研究所



34/248



Back

Close



35/248

- 深圳分中心：深圳先进技术研究院
- 沈阳分中心：沈阳金属研究所
- 武汉分中心：武汉水生生物研究所
- 广州分中心：生物医药与健康研究院



Back

Close



36/248

# 国际上千万亿次计算的应用问题

## 美国千万亿次应用问题

### 1. 天体物理

- 恒星的辐射、动力学和核物理；
- 超新星物理、伽玛射线爆发、双黑洞系统和 neutron 星之间的碰撞；
- 地球、巨型气体行星 (*gas giants*)、恒星的电磁场如何产生和演化？
- 冠状物质抛射 (*coronal mass ejections*) 及其对地球电磁场的影响，包括磁场的重结 (*magnetic reconnection*) 和地磁场次暴 (*geo-magnetic sub-storms*) 的模拟；



Back

Close



- 银河系的形成与演化;
- 低马赫数天体物质流 (*Low Mach-number astrophysical flows*), 例如形成行星云 (*planetary nebula*) 的恒星外壳的爆炸;
- 早期宇宙结构的演化;
- 分子云 (*molecular clouds*) 和前恒星核 (*pre-stellar cores*) 的形成

## 2. 流体力学

- 在经典电磁流体、化学反应物中的分层与不分层、有旋涡与无漩涡湍流中的详细结构和性质;
- 在复杂系统中化学反应过程与流体动力学间的相互作用, 例如燃烧、大气化学以及化学反应过程;
- 可压缩多相流的性质





38/248

### 3. 环境科学

- 大气系统、气象系统和地球气候之间的非线性相互作用;
- 地球碳、氮、水的耦合循环动力学;
- 通过高分辨率、宽带、全球的地震探测研究地球的内部构造;
- 十年期间的大型江河流域水文动力学;
- 海洋与陆地以及海洋与大气的耦合动力学

### 4. 生物科学

- 具有大生物分子和生物分子团的反应机理, 例如酶、核糖体和细胞膜;
- 在只给定基本氨基酸序列的条件下预测蛋白质的三维结构;
- 病毒衣壳组装 (*assembly of capsids*) 的理解

### 5. 材料科学



Back

Close



39/248

- 利用第一原理模拟极端条件下物质的块体性质 (*bulk properties*);
- 适应特殊性和高效地运用第一原理设计催化剂、药物和其它分子材料;
- 材料设计;
- 对摩擦和润滑分子层面的理解;
- 精确到1卡/摩尔的任何化学反应界面势能, 以及在此界面的分子相应的动力学行为的研究;
- 分子电子器件设计;
- 纳米尺度的工程结构的性质;
- 半导体和金属表面的多相催化作用

## 6. 高能物理



Back

Close



40/248

- 强关联系统 (*Strongly correlated systems*);
- 阿秒脉冲激光与多原子分子 (*polyatomic molecules*) 的相互作用;
- 强相互作用主导的高能物理过程;
- 燃烧等离子体的性质和不稳定性, 以及主动磁约束技术研究

## 7. 工业应用

- 工程系统健壮和优化设计;
- 复杂系统的控制;
- 特别巨大的 (天文数字的) 数据集合的分析



Back

Close



# 欧盟千万亿次应用问题



41/248

## 1. 气象、气候学和地球科学领域

- 气候改变，气象学，水文学和空气质量
- 海洋学和海洋业预报
- 地球科学

## 2. 天体物理学，高能物理和等离子体物理领域

- 天体物理学，包括从天体的形成，到整个宇宙的起源和演化问题
- 基本粒子物理学
- 等离子体物理，包括建造*ITER*提出的科学和技术问题

## 3. 材料科学、化学和纳米科学领域

- 复杂材料的理解，包括对各类材料的成核现象、生长、自组织和



Back

Close



42/248

聚合的模拟, 确定工艺过程、使用条件和构成之间关系的材料力学性质的多尺度描述

- 复杂化学的理解, 包括大气化学、软物质化学(例如聚合物)、燃烧的原子层次的描述、超分子装配技术、生物化学
- 纳米科学, 包括纳电子学, 纳米尺度的机械属性仿真, 纳米尺度的流变学、应用流体学和摩擦学的基于原子作用的描述

#### 4. 生命科学领域

- 系统生物学, 在未来4年内, 欧洲将实现世界上第一个“硅片中的”细胞
- 染色体动力学
- 大尺度蛋白质动力学
- 蛋白质联结和聚合



Back

Close



- 超分子系统
- 医学，例如，确定触发多基因疾病、预测在某些人群中与药物异常代谢相关的次级作用或药物与异于其原始靶点的大分子的交互作用的仿真

## 5. 工程学领域

- 直升机的完全仿真
- 生物医学流体力学
- 燃气轮机和内燃烧引擎
- 森林火灾
- 绿色飞行器
- 虚拟发电厂



Back

Close

# 日本千万亿次应用问题

## 主要目标

1. 瞄准未来的知识发现与创造
2. 在先进科学与技术上的若干突破
3. 在经济与环境上的长久可持续发展
4. 强化经济与工业领域，构建创新型日本
5. 保障贯穿整个生命周期的良好健康状况
6. 建设安全的国家



44/248



Back

Close



## 习题

1. 在主程序中定义了矩阵 $A$ 和 $B$  (比如 $C$ 语言,  $A[50][67]$ ,  $B[73][49]$ ), 请写一个通用子程序完成矩阵相加, 使得对于任何 $m \times n$ 的子矩阵 $A$ 与子矩阵 $B$ 相加都能调用该子程序, 用 $C$ 或 $Fortran$  语言 (比如,  
 $matradd(m, n, ..., A, ..., B, ..., C)$ ).
2. 设多项式 $P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ , 请给出计算 $P_n(x_0)$ 的方法并写出计算它的子程序。
3. 对任意给定的 $x \neq 0$ 和 $n \geq 0$ , 用算法1 写一个程序计算 $x^n$ 。



Back

Close



46/248

## 二、并行计算机体系结构

这章主要以互连网络的结构为介绍内容，它是并行计算机体系结构的核心

2.1、网络的分类

2.2、网络的基本概念

2.3、间接网络

2.4、直接网络

本章图片来源于中科院计算技术研究所唐志敏研究员



Back

Close



# 网络的分类

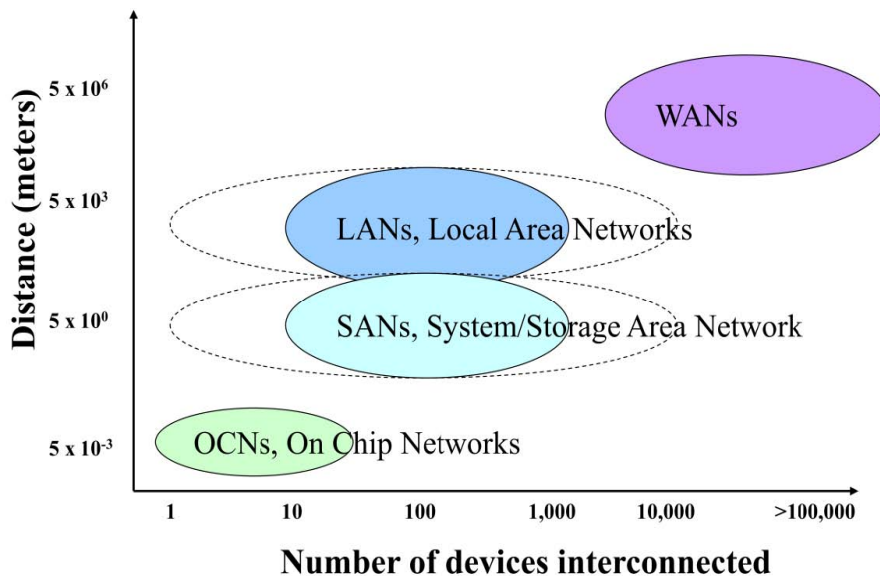


图 2.1 网络的分类





# 网络的基本概念

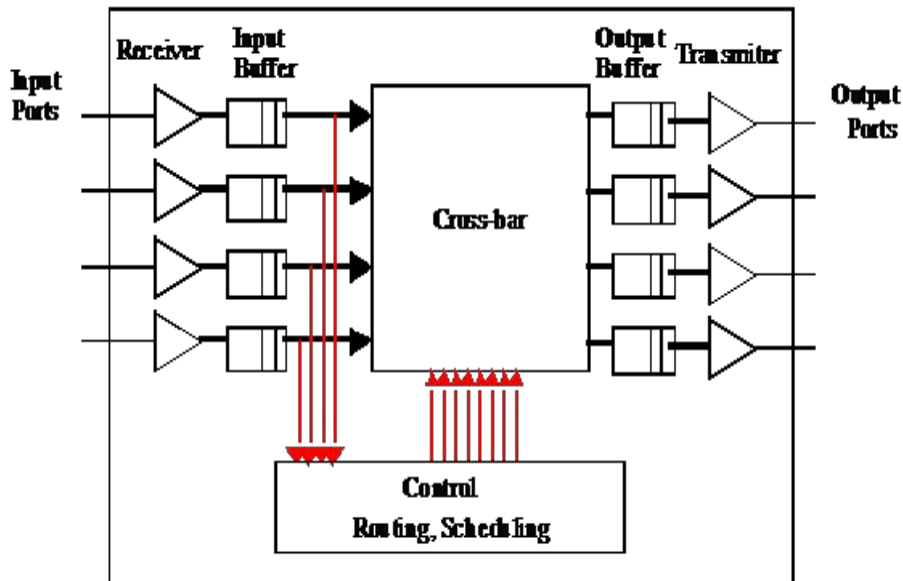


图 2.2 开关







49/248

定义 1 链路 (*link*) 是节点或者开关与节点或开关之间的连接线

定义 2 网络是由链路和开关构成的一个图

如果记所有开关为  $V$ , 所有链路为  $E$ , 则显然有  $E \subset V \times V$ ; 进一步对于全连接网络, 则有  $E = V \times V$ 。

网络连接有直接连接和间接连接两种, 接下来就分别介绍一些典型的网络代表。



Back

Close

# 间接网络

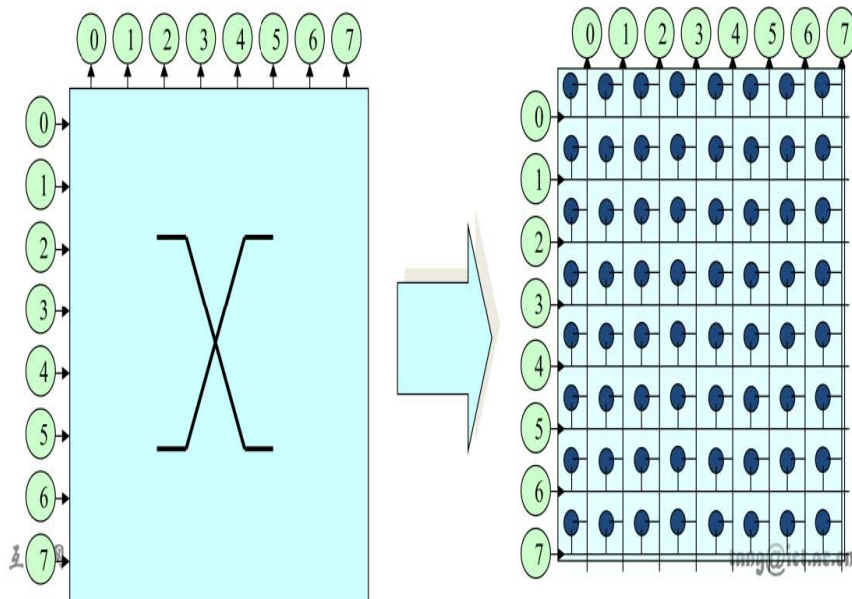


图 2.3 交叉开关



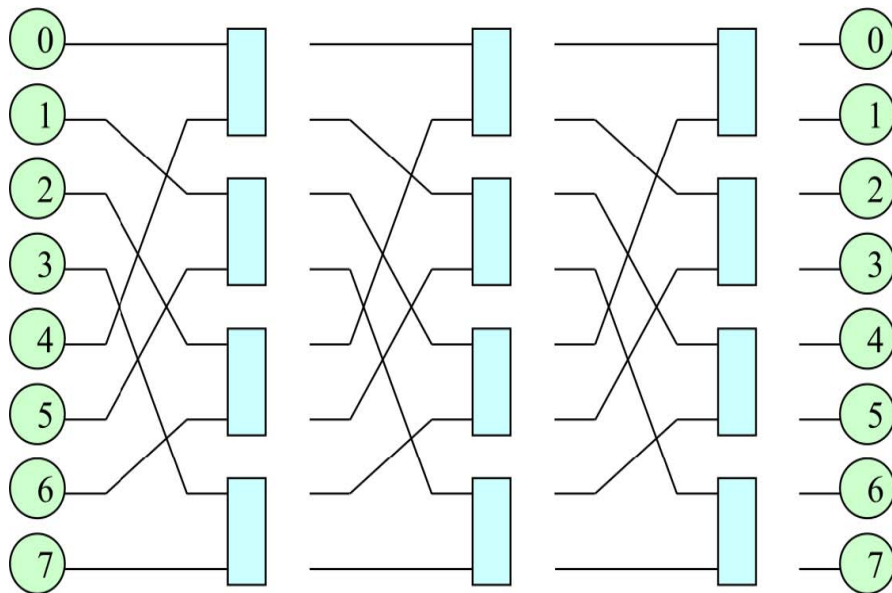


图 2.4 8口 $\Omega$ 网络

连接方式:  $x_{n-1}x_{n-2}\cdots x_0 \Rightarrow x_{n-2}\cdots x_0x_{n-1}$ , 亦称为完美洗牌。

这样构造的互连网络与交叉开关网络在一般情况之下是可以媲美



Back

Close



52/248

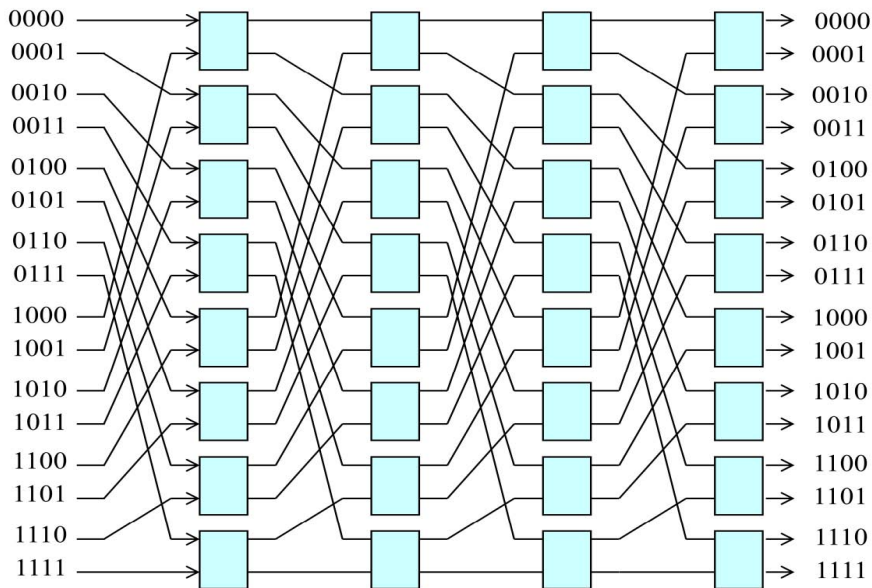


图 2.5 16口 $\Omega$ 网络

的，能够完成任意两点之间的信息通讯



Back

Close

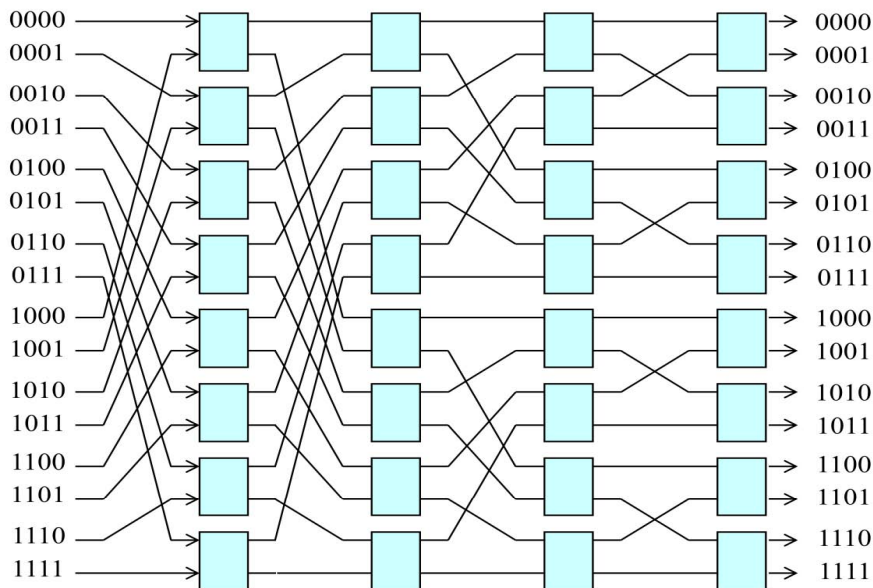


图 2.6 16口baseline网络

连接方式:

$$x_{n-1}x_{n-2} \cdots x_0 \Rightarrow x_{n-2} \cdots x_0 x_{n-1}$$

$$x_{n-1}x_{n-2} \cdots x_0 \Rightarrow x_0 x_{n-1} \cdots x_1$$

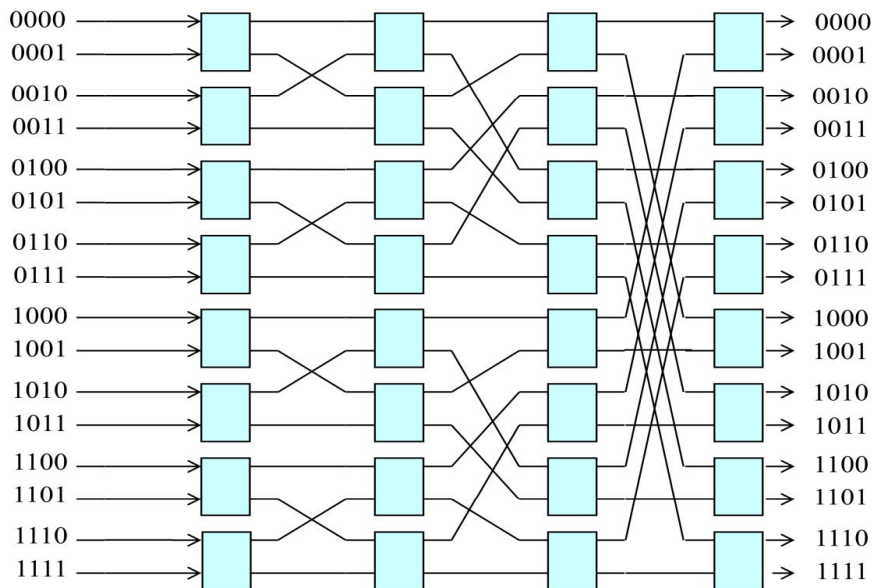



图 2.7 16口butterfly网络

连接方式:  $x_{n-1}x_{n-2} \cdots x_0 \Rightarrow x_0x_{n-1} \cdots x_1$ , 蝶式置换  
 $x_{n-1}x_{n-2} \cdots x_1x_0 \Rightarrow x_0x_{n-2} \cdots x_1x_{n-1}$



Back

Close

# 直接网络

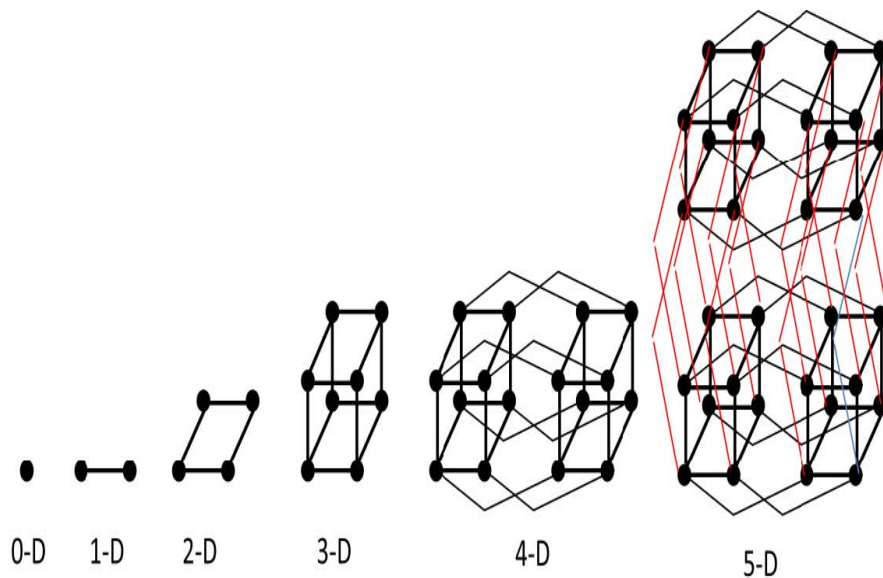


图 2.8 超立方互连



55/248



Back

Close

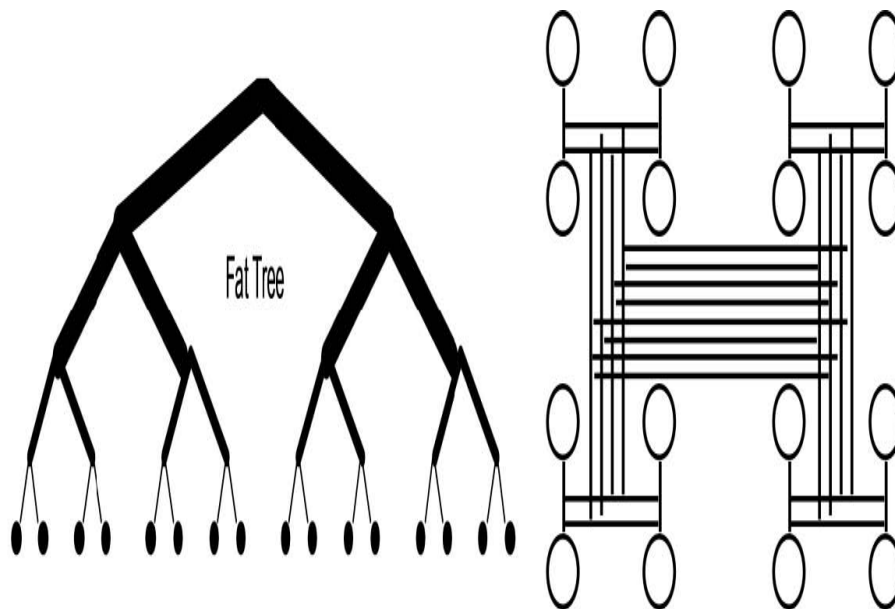


图 2.9 胖树互连

- 树的自然形态
- 叶子是节点
- 环形互连



Back

Close





57/248

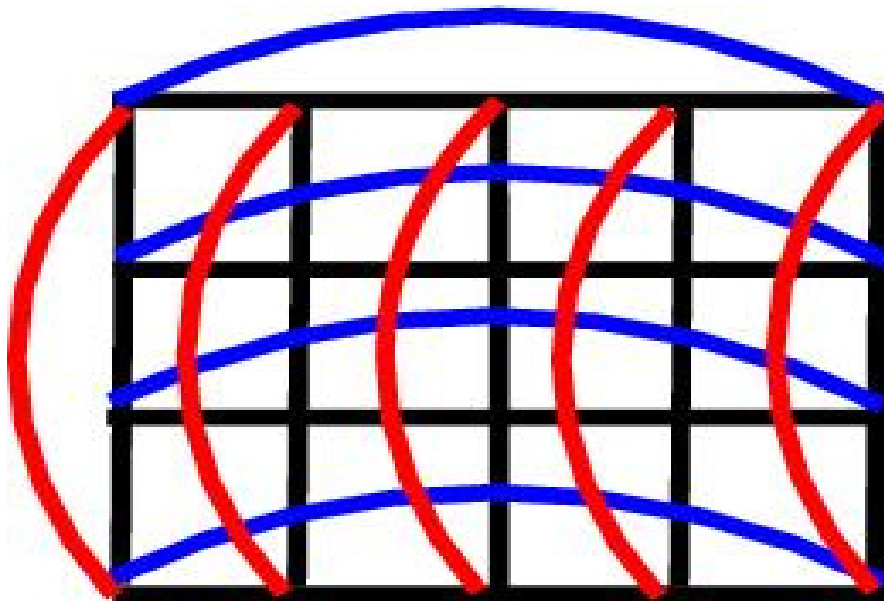


图 2.10 2D Torus

- 2D网格互连



Back

Close



# 习题

1. 设 $n$ 是环形结构的节点数, 记节点之间消息通讯的长度为 $l$ , 证明 $l < n/2$
2. 如果是 $n \times n$  的 $2D$ 环形互连结构, 则 $l < n$



Back

Close



# 三、并行计算的基本概念

3.1、并行计算机系统-*MPP*

3.2、并行计算机系统-*SMP*

3.3、并行计算机系统-*Cluster*

3.5、并行计算机系统的分类

3.6、并行计算的程序结构

3.7、并行计算的基本定义

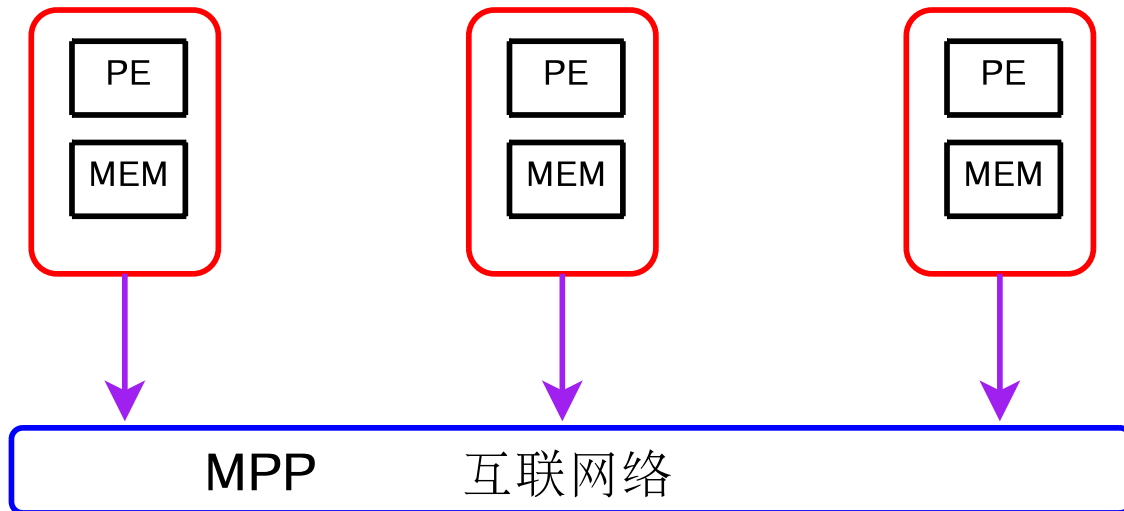


Back

Close

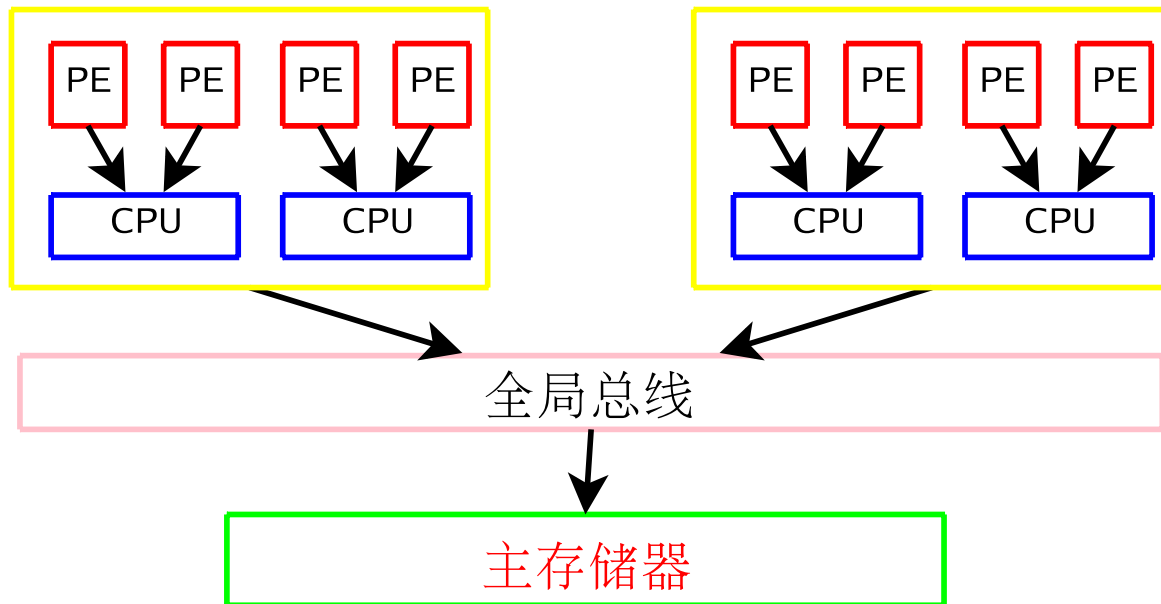


# 并行计算机系统-MPP





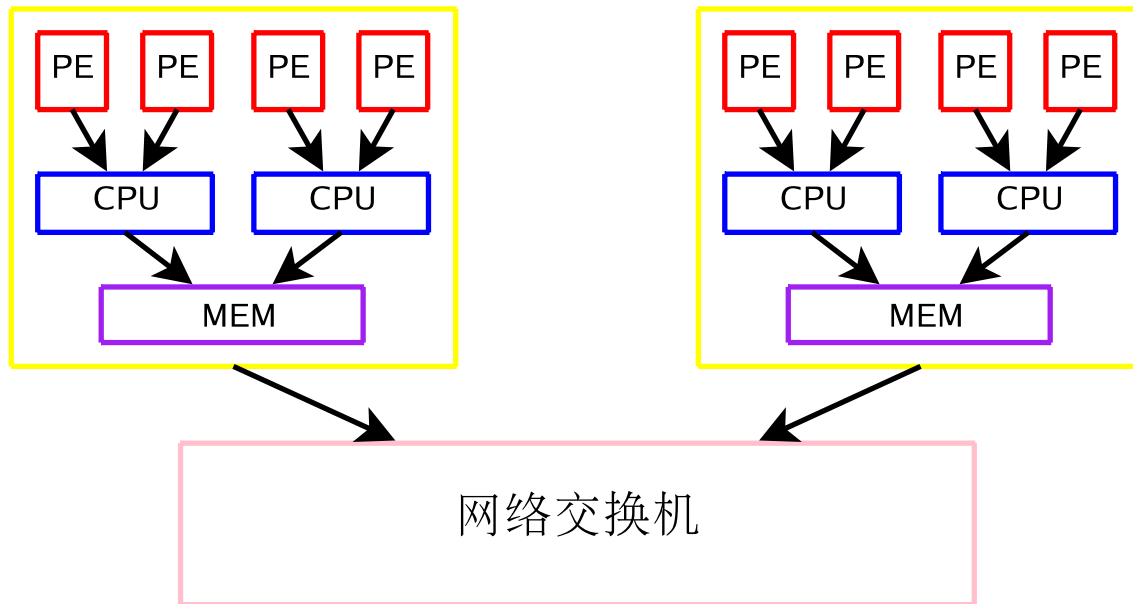
# 并行计算机系统-SMP





62/248

# 并行计算机系统-Cluster

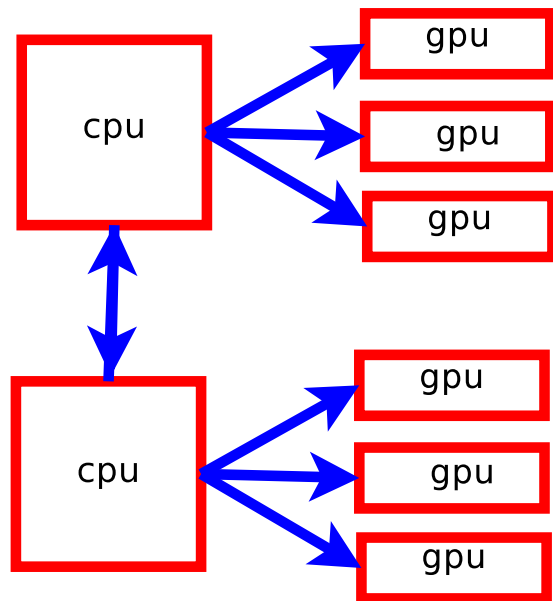


Back

Close



## 并行计算机系统-*Hybrid*



此类系统需要使用混合编程, *MPI+OpenMP+Cuda*。





64/248

# 并行计算机系统的分类

1. 单指令流单数据流(*SISD*), 现今普通计算机
2. 多指令流单数据流(*MISD*), 没有实际的计算机
3. 单指令流多数据流(*SIMD*), 向量计算机、共享存储计算机(参见 3.2)
4. 多指令流多数据流(*MIMD*), 大规模并行处理系统、机群(参见 3.1、3.3)

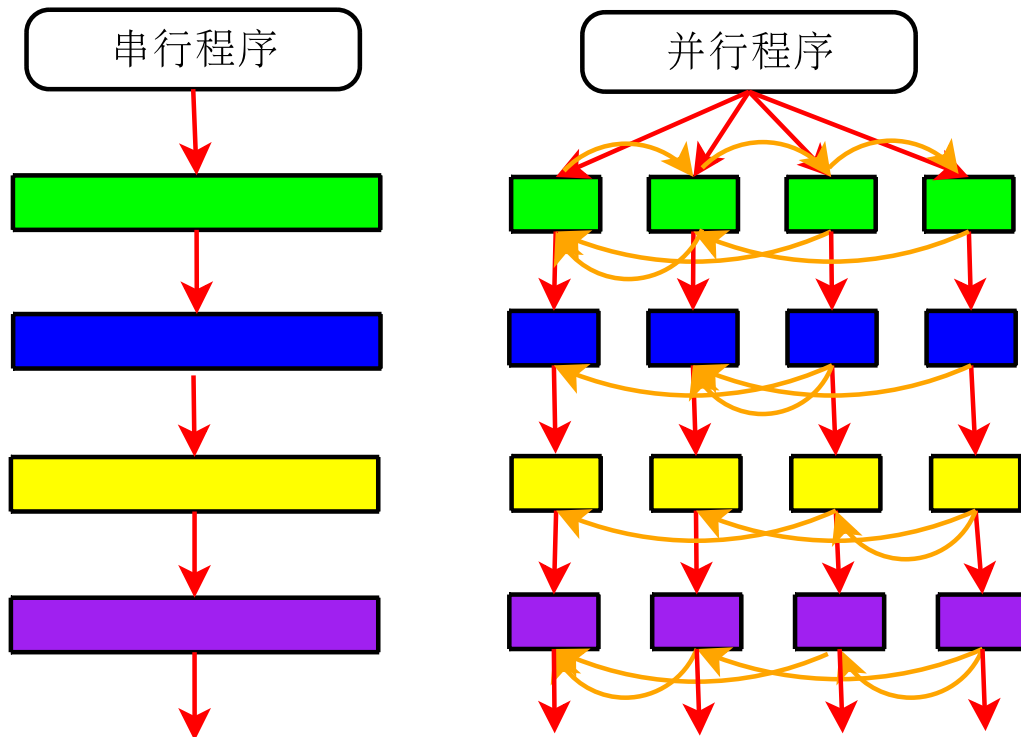


Back

Close



# 并行计算的程序结构





# 并行计算的基本定义

粒度：在并行执行过程中，二次通讯之间每个处理机计算工作量大小的一个粗略描述。分为粗粒度、细粒度。

复杂性：在不考虑通讯开销的前提下，每个处理机上的计算量最大者，即为并行计算复杂性。 $\log_2 n$

并行度：算法可以并行的程度。

加速比：

$$S_p(q) = \frac{T_s}{T_p(q)} \quad (3.1)$$

效率：

$$E_p(q) = \frac{S_p(q)}{q} \quad (3.2)$$



Back

Close



67/248

*Amdahl*定律：假设串行计算所需要的时间 $T_s = 1$ ， $\alpha$ 是执行该计算所必需的串行部分所占的百分比，则有

$$S_p(q) = \frac{1}{\alpha + (1 - \alpha)/q} \quad (3.3)$$

$$\lim_{q \rightarrow \infty} S_p(q) = \frac{1}{\alpha} \quad (3.4)$$

*Gustafson*定律：假设并行计算所需要的时间 $T_p = 1$ ， $\alpha$ 是执行该并行计算所需的串行部分所占的百分比，则有

$$S_p(q) = \frac{\alpha + (1 - \alpha) \times q}{1} \quad (3.5)$$



Back

Close



68/248

# 习题

1. 从加速比的基本定义出发, 证明 *Amdahl* 定律和 *Gustafson* 定律;
2. 你所了解的并行计算的基本方法 (在学习的过程中查阅资料, 了解一些基本并行计算方法)。



Back

Close



## 四、矩阵乘并行计算

4.1、矩阵卷帘存储方式

4.2、串行矩阵乘法

4.3、行列分块算法

4.4、行行分块算法

4.5、列行分块算法

4.6、列列分块算法

4.7、*Cannon*算法

矩阵乘是矩阵计算的基础，这里给大家介绍的并行计算方法，是为了让大家了解如何设计并行计算方法，这是并行计算方法设计的基



Back

Close



70/248

础, 掌握了这里的方法, 对今后开展并行计算将起到至关重要的作用。除了我们将要介绍的矩阵乘并行计算方法外, 矩阵计算主要考虑如下问题:

$$Ax = b, \quad \min_{x \in R^n} \|Ax - b\|, \quad Ax = \lambda x, \quad A = U\Sigma V^T$$

所有这些问题的求解, 都需要对矩阵 $A$ 进行分解。针对最小二乘问题, 通常采用 $QR$ 分解方法, 亦即 $A = QR$ , 其中 $Q$ 是正交矩阵,  $R$ 是上三角矩阵。我们这里只介绍 $Ax = b$ 的 $LU$ 分解计算方法(第五章)。



Back

Close



## 矩阵卷帘 (*wrap*) 存储方式

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} & A_{05} & A_{06} & A_{07} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & A_{16} & A_{17} \\ A_{20} & A_{21} & A_{22} & A_{23} & A_{24} & A_{25} & A_{26} & A_{27} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & A_{36} & A_{37} \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & A_{46} & A_{47} \\ A_{50} & A_{51} & A_{52} & A_{53} & A_{54} & A_{55} & A_{56} & A_{57} \\ A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} & A_{67} \\ A_{70} & A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{pmatrix} \quad (4.1)$$



Back

Close



72/248

在一个 $3 \times 2$ 的处理机网格上,  $8 \times 8$ 矩阵的存储方式如下:

$$\left( \begin{array}{cccc|cccc} A_{00} & A_{02} & A_{04} & A_{06} & A_{01} & A_{03} & A_{05} & A_{07} \\ A_{30} & A_{32} & A_{34} & A_{36} & A_{31} & A_{33} & A_{35} & A_{37} \\ A_{60} & A_{62} & A_{64} & A_{66} & A_{61} & A_{63} & A_{65} & A_{67} \\ \hline A_{10} & A_{12} & A_{14} & A_{16} & A_{11} & A_{13} & A_{15} & A_{17} \\ A_{40} & A_{42} & A_{44} & A_{46} & A_{41} & A_{43} & A_{45} & A_{47} \\ A_{70} & A_{72} & A_{74} & A_{76} & A_{71} & A_{73} & A_{75} & A_{77} \\ \hline A_{20} & A_{22} & A_{24} & A_{26} & A_{21} & A_{23} & A_{25} & A_{27} \\ A_{50} & A_{52} & A_{54} & A_{56} & A_{51} & A_{53} & A_{55} & A_{57} \end{array} \right) \quad (4.2)$$

对于一般 $m \times n$ 分块矩阵和一般的处理机阵列 $p \times q$ , 小块 $A_{ij}$ 存放在处理机 $P_{kl}$  ( $k = i \bmod p, l = j \bmod q$ ) 中。



Back

Close





# 串行矩阵乘法

串行矩阵乘积子程序(*i-j-k*形式)

```
do i=1, M
  do j=1, L
    do k=1, N
       $c(i, j) = c(i, j) + a(i, k) * b(k, j)$ 
    enddo
  enddo
enddo
```

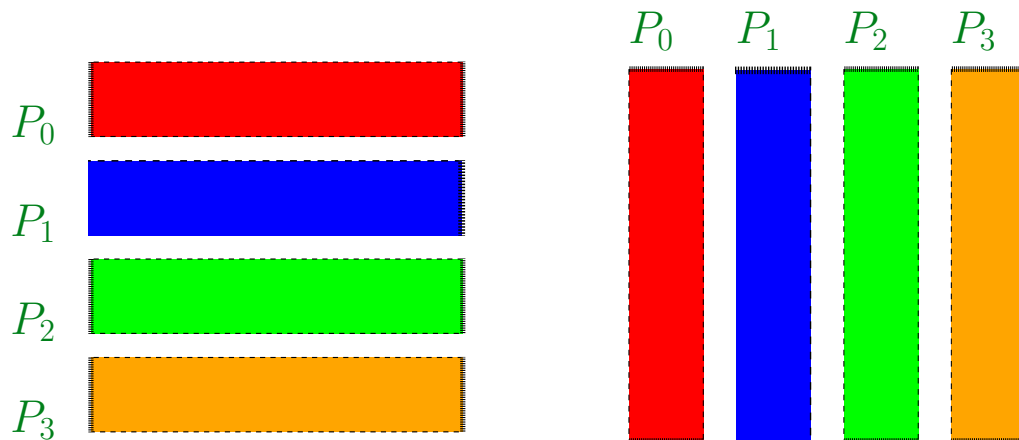


Back

Close



# 行列分块算法



$$A = \begin{bmatrix} A_0^T & A_1^T & \cdots & A_{p-1}^T \end{bmatrix}^T, \quad B = \begin{bmatrix} B_0 & B_1 & \cdots & B_{p-1} \end{bmatrix} \quad (4.3)$$

亦即  $A_i$ 、 $B_i$  存放在处理机  $P_i$  中。  $C_{ij} = A_i \times B_j$



Back

Close

## 算法 2 行列分块算法

$\text{mp1} \equiv (\text{myid}+1) \bmod p, \text{mm1} \equiv (\text{myid}-1 + p) \bmod p$

for  $i = 0$  to  $p - 1$  do

$l \equiv (i+\text{myid}) \bmod p$

$C_l = A \times B$

if  $i \neq p - 1$ , send( $B$ , mm1), recv( $B$ , mp1)

end{for}



75/248



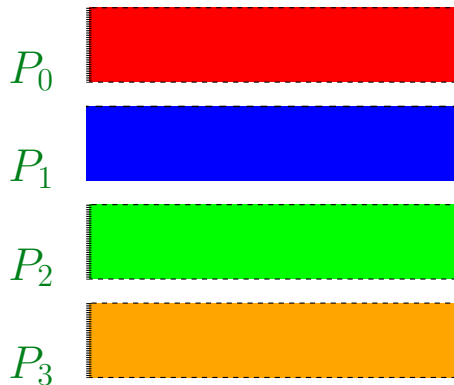
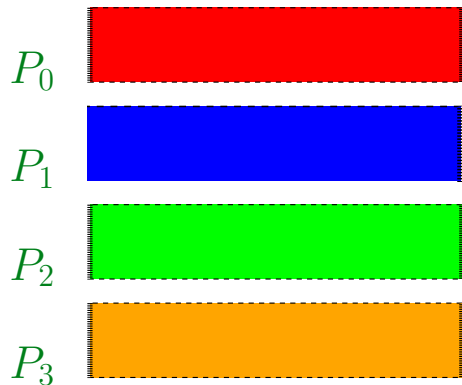
Back

Close



76/248

# 行行分块算法



$$B = \begin{bmatrix} B_0^T & B_1^T & \cdots & B_{p-1}^T \end{bmatrix}^T, \quad A_i = \begin{bmatrix} A_{i0} & A_{i1} & \cdots & A_{i,p-1} \end{bmatrix} \quad (4.4)$$



Back

Close

### 算法 3 行行分块算法

$\text{mp1} \equiv (\text{myid}+1) \bmod p$ ,  $\text{mm1} \equiv (\text{myid}-1 + p) \bmod p$

for  $i = 0$  to  $p - 1$  do

$l \equiv (i+\text{myid}) \bmod p$

$C = C + A_l \times B$

if  $i \neq p - 1$ ,  $\text{send}(B, \text{mm1})$ ,  $\text{recv}(B, \text{mp1})$

end{for}



77/248



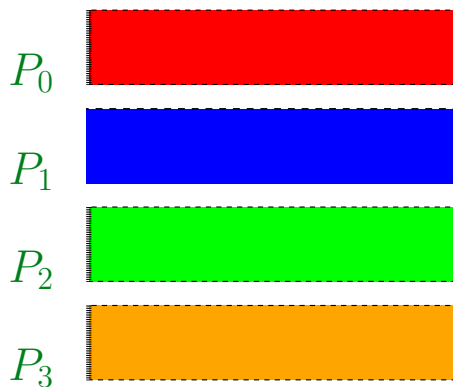
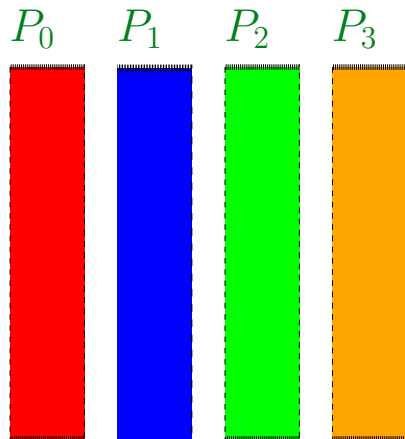
Back

Close



78/248

# 列行分块算法



$$A = \begin{bmatrix} A_0 & A_1 & \cdots & A_{p-1} \end{bmatrix}, \quad B_i = \begin{bmatrix} B_{i0} & B_{i1} & \cdots & B_{i,p-1} \end{bmatrix} \quad (4.5)$$

从而有  $C = \sum_{i=0}^{p-1} A_i \times B_i$ 。



Back

Close

## 算法 4 列行分块算法

$$C = A \times B_{\text{myid}}$$

for  $i = 1$  to  $p - 1$  do

$$l \equiv (i + \text{myid}) \bmod p, k \equiv (p - i + \text{myid}) \bmod p$$

$$T = A \times B_l$$

send( $T, l$ ), recv( $T, k$ )

$$C = C + T$$

end{for}



79/248

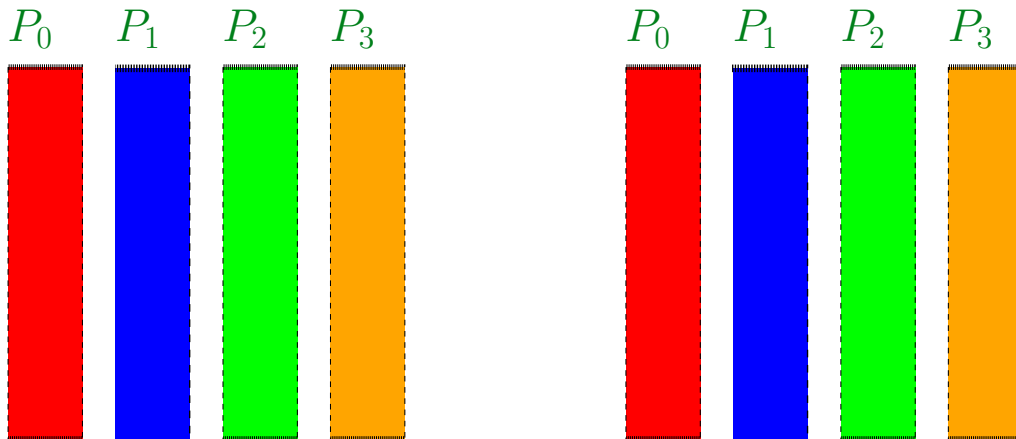


Back

Close



# 列列分块算法



$$B_i = \begin{bmatrix} B_{i0}^T & B_{i1}^T & \dots & B_{i,p-1}^T \end{bmatrix}^T$$





## 算法 5 列列分块算法

$\text{mp1} \equiv (\text{myid}+1) \bmod p, \text{mm1} \equiv (\text{myid}-1 + p) \bmod p$

for  $i = 0$  to  $p - 1$  do

$l \equiv (i+\text{myid}) \bmod p$

$C = C + A \times B_l$

if  $i \neq p - 1$ ,  $\text{send}(A, \text{mm1}), \text{recv}(A, \text{mp1})$

end{for}



81/248



Back

Close



82/248

# Cannon算法

$$\begin{array}{ccc|ccc} A_{00} & A_{01} & A_{02} & B_{00} & B_{01} & B_{02} \\ A_{10} & A_{11} & A_{12} & B_{10} & B_{11} & B_{12} \\ A_{20} & A_{21} & A_{22} & B_{20} & B_{21} & B_{22} \end{array}$$

$$\begin{array}{ccc|ccc} A_{00} & A_{00} & A_{00} & B_{00} & B_{01} & B_{02} \\ A_{11} & A_{11} & A_{11} & B_{10} & B_{11} & B_{12} \\ A_{22} & A_{22} & A_{22} & B_{20} & B_{21} & B_{22} \end{array}$$

$$\begin{array}{ccc|ccc} A_{01} & A_{01} & A_{01} & B_{10} & B_{11} & B_{12} \\ A_{12} & A_{12} & A_{12} & B_{20} & B_{21} & B_{22} \\ A_{20} & A_{20} & A_{20} & B_{00} & B_{01} & B_{02} \end{array}$$



Back

Close



$$\begin{array}{ccc|ccc} A_{02} & A_{02} & A_{02} & B_{20} & B_{21} & B_{22} \\ A_{10} & A_{10} & A_{10} & B_{00} & B_{01} & B_{02} \\ A_{21} & A_{21} & A_{21} & B_{10} & B_{11} & B_{12} \end{array}$$

## 算法 6 *Cannon*算法

$$C = 0$$

$$\text{mpc1} \equiv (\text{mycol}+1) \bmod m; \text{mmc1} \equiv (m+\text{mycol}-1) \bmod m;$$

$$\text{mpr1} \equiv (\text{myrow}+1) \bmod m; \text{mmr1} \equiv (m+\text{myrow}-1) \bmod m;$$

for  $i = 0$  to  $m - 1$  do

$$k \equiv (\text{myrow}+i) \bmod m;$$

$$r \equiv (m + k - i) \bmod m;$$

if  $\text{mycol}=k$  &  $\text{myrow}=r$  then



Back

Close



84/248

```
    send( $A$ , (myrow, mpc1)); copy( $A$ , tmpA);  
else if myrow= $r$   
    recv(tmpA, (myrow, mmc1));  
    if  $k \neq \text{mpc1}$ , send(tmpA, (myrow, mpc1));  
end{if}  
 $C = C + \text{tmpA} \times B$ ;  
if  $i \neq m - 1$  then  
    send( $B$ , (mmr1, mycol)); recv( $B$ , (mpr1, mycol));  
end{if}
```

```
end{for}
```



Back

Close



## 修正Cannon算法

这个算法针对矩阵分块乘  $C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$  的要求，需要矩阵  $A$  的列指标和矩阵  $B$  的行指标相同。如何实现？在前面介绍的Cannon算法，我们选择了一种方案，现在我们将给大家一个新的方法，实现并行计算矩阵乘。

与Cannon算法相同，假设使用  $p \times p$  个处理。这里对矩阵  $A$  的列进行调整，对矩阵  $B$  的行进行调整，使得每一个处理机上的  $A$  和  $B$  是满足要求的。



Back

Close



86/248

$A_{00}$   $A_{01}$   $A_{02}$   $A_{03}$

$A_{10}$   $A_{11}$   $A_{12}$   $A_{13}$

$A_{20}$   $A_{21}$   $A_{22}$   $A_{23}$

$A_{30}$   $A_{31}$   $A_{32}$   $A_{33}$

$A_{00}$   $A_{01}$   $A_{02}$   $A_{03}$

$A_{11}$   $A_{12}$   $A_{13}$   $A_{10}$

$A_{22}$   $A_{23}$   $A_{20}$   $A_{21}$

$A_{33}$   $A_{30}$   $A_{31}$   $A_{32}$

$B_{00}$   $B_{01}$   $B_{02}$   $B_{03}$

$B_{10}$   $B_{11}$   $B_{12}$   $B_{13}$

$B_{20}$   $B_{21}$   $B_{22}$   $B_{23}$

$B_{30}$   $B_{31}$   $B_{32}$   $B_{33}$

$B_{00}$

$B_{10}$   $B_{11}$

$B_{20}$   $B_{21}$   $B_{22}$

$B_{30}$   $B_{31}$   $B_{32}$   $B_{33}$

$B_{01}$   $B_{02}$   $B_{03}$

$B_{12}$   $B_{13}$

$B_{23}$



Back

Close

 $A_{00} \quad A_{01} \quad A_{02} \quad A_{03}$  $A_{11} \quad A_{12} \quad A_{13} \quad A_{10}$  $A_{22} \quad A_{23} \quad A_{20} \quad A_{21}$  $A_{33} \quad A_{30} \quad A_{31} \quad A_{32}$  $B_{00} \quad B_{11} \quad B_{22} \quad B_{33}$  $B_{10} \quad B_{21} \quad B_{32} \quad B_{03}$  $B_{20} \quad B_{31} \quad B_{02} \quad B_{13}$  $B_{30} \quad B_{01} \quad B_{12} \quad B_{23}$ 

形成这个变形之后，现在每个进程中的矩阵 $A$ 和 $B$ 既是计算 $C$ 的一个可以相乘的数据对。接下来的计算每次只需要对 $A$ 进行列左移一



Back

Close

格,  $B$ 行上移一格。

$$A_{01} \quad A_{02} \quad A_{03} \quad A_{00}$$

$$A_{12} \quad A_{13} \quad A_{10} \quad A_{11}$$

$$A_{23} \quad A_{20} \quad A_{21} \quad A_{22}$$

$$A_{30} \quad A_{31} \quad A_{32} \quad A_{33}$$

$$B_{10} \quad B_{21} \quad B_{32} \quad B_{03}$$

$$B_{20} \quad B_{31} \quad B_{02} \quad B_{13}$$

$$B_{30} \quad B_{01} \quad B_{12} \quad B_{23}$$

$$B_{00} \quad B_{11} \quad B_{22} \quad B_{33}$$



88/248



Back

Close





## 习题

1. 假设若把 $m \times n$ 的矩阵 $A$ 按照循环方式存放在 $p \times q$ 的处理机系统中, 在每个处理机 $P_{kl}$ 上所得到的矩阵为 $A^{kl}$ , 请建立矩阵 $A$ 与矩阵 $A^{kl}$ 的关系式, 即 $\forall a_{ij}, \exists k, l, s, t$ , 使得 $a_{ij} = a_{st}^{kl}$ ;
2. 假设矩阵 $A$ 和向量 $b$ 是按行分块存储在 $p$ 个处理机中的, 请给出并行计算 $x^+ = Ax + b$ 的方法;
3. 请分析矩阵乘法的列列分块算法和Cannon算法的复杂性 (假设传输长度为 $N$ 的数据的复杂性为 $\alpha + N\beta$ , 广播数据按照树型方式进行);
4. 请用自己的理解给出Cannon算法的描述。



Back

Close



90/248

# 五、 线性代数方程组的并行求解

5.1、串行 $LU$ 分解算法

5.2、分布式系统的并行 $LU$ 分解算法

5.3、三角方程组的并行求解

5.4、经典迭代法-Jacobi

5.5、经典迭代法-Gauss-Seidel



Back

Close



# 串行 $LU$ 分解算法

$$Ax = b \quad (5.1)$$

$PA = LU$ ,  $Ly = Pb$ ,  $Ux = y$ 。

$P(i, j)$ 是由单位矩阵交换第 $i$ 行和第 $j$ 行得到的矩阵, 记单位矩阵 $I$ 的第 $i$ 列为 $e_i$ , 则有

$$\begin{aligned} P(i, j) &= \begin{pmatrix} e_1 & \cdots & e_{i-1} & e_j & e_{i+1} & \cdots & e_{j-1} & e_i & e_{j+1} & \cdots & e_n \end{pmatrix} \\ &= I + e_i(e_j - e_i)^T + e_j(e_i - e_j)^T \end{aligned} \quad (5.2)$$



Back

Close



容易验证矩阵 $P(i, j)$ 是对称矩阵, 并且有

$$\begin{aligned}
 P(i, j)^2 &= I + e_i(e_j - e_i)^T + e_j(e_i - e_j)^T + e_i(e_j - e_i)^T \\
 &\quad + e_i(e_j - e_i)^T e_i(e_j - e_i)^T + e_j(e_i - e_j)^T e_i(e_j - e_i)^T \\
 &\quad + e_j(e_i - e_j)^T + e_i(e_j - e_i)^T e_j(e_i - e_j)^T \\
 &\quad + e_j(e_i - e_j)^T e_j(e_i - e_j)^T \\
 &= I + 2e_i(e_j - e_i)^T + 2e_j(e_i - e_j)^T \\
 &\quad - e_i(e_j - e_i)^T + e_j(e_j - e_i)^T \\
 &\quad + e_i(e_i - e_j)^T - e_j(e_i - e_j)^T = I
 \end{aligned} \tag{5.3}$$

记 $\tilde{l}_i$ 是分量小于等于 $i$ 均为0的一个 $n$ 维向量,  $l_i$ 是 $n - i$ 维向量, 亦即

$$\tilde{l}_i^T = (0, \dots, 0, l_{i+1,i}, \dots, l_{n,i})^T = (0, \dots, 0, l_i^T)^T$$

如果矩阵 $A$ 的元素 $a_{11} \neq 0$ , 一定存在一个矩阵 $L_1 = I + \tilde{l}_1 e_1^T$ , 使得矩阵 $L_1 A$ 的第1列为 $a_{11} e_1$ 。



## 算法 7 部分选主元的 *Guass* 消去算法

for  $j = 0$  to  $n - 2$  do

find  $l$ :  $|a_{lj}| = \max\{|a_{ij}|, i = j, \dots, n - 1\}$

if  $l \neq j$ , swap  $A_j$  and  $A_l$

if  $a_{jj} = 0$ ,  $A$  is singular and return

$a_{ij} = a_{ij}/a_{jj}$ ,  $i = j + 1, \dots, n - 1$

for  $k = j + 1$  to  $n - 1$  do

$a_{ik} = a_{ik} - a_{ij} \times a_{jk}$ ,  $i = j + 1, \dots, n - 1$

end{for}

end{for}



93/248



Back

Close



94/248

# 分布式系统的并行 $LU$ 分解算法

icol= 0

for  $j = 0$  to  $n - 2$  do

    if myid= $j \bmod p$  then

        find  $l$ :  $|a_{l,icol}| = \max\{|a_{i,icol}|, i = j, \dots, n - 1\}$

        if  $l \neq j$ , swap  $a_{j,icol}$  and  $a_{l,icol}$

        if  $a_{j,icol} = 0$ ,  $A$  is singular and kill all processes

$a_{i,icol} = a_{i,icol}/a_{j,icol}$ ,  $f_{i-j-1} = a_{i,icol}$ ,  $i = j + 1, \dots, n - 1$

        send( $l$ , myid+1) and send( $f$ , myid+1), icol+1  $\rightarrow$  icol

    else



Back

Close



95/248

recv( $l$ , myid-1) and recv( $f$ , myid+1)

if myid+1  $\neq j \bmod p$  then

send( $l$ , myid+1) and send( $f$ , myid+1)

end{if}

end{if}

if  $l \neq j$ , swap  $A_j$  and  $A_l$

for  $k=\text{icol}$  to  $m-1$  do

$a_{ik} = a_{ik} - f_i \times a_{jk}, i = j+1, \dots, n-1$

end{for}

end{for}



Back

Close



96/248

# 三角方程组的并行解法

$k = 0$

if  $myid=0$ , then

$$u_i = b_i, i = 0, \dots, n-1, v_i = 0, i = 0, \dots, p-2$$

else

$$u_i = 0, i = 0, \dots, n-1$$

for  $i = myid$  step  $p$  to  $n-1$  do

if  $i > 0$ ,  $recv(v, i-1 \bmod p)$

$$x_k = (u_i + v_0)/l_{ik}$$



Back

Close





97/248

$$v_j = v_{j+1} + u_{i+1+j} - l_{i+1+j,k} \times x_k, j = 0, \dots, p-3$$

$$v_{p-2} = u_{i+p-1} - l_{i+p-1,k} \times x_k$$

send( $v, i + 1 \bmod p$ )

$$u_j = u_j - l_{jk} \times x_k, j = i + p, \dots, n-1$$

$$k + 1 \rightarrow k$$

end{for}



Back

Close



98/248

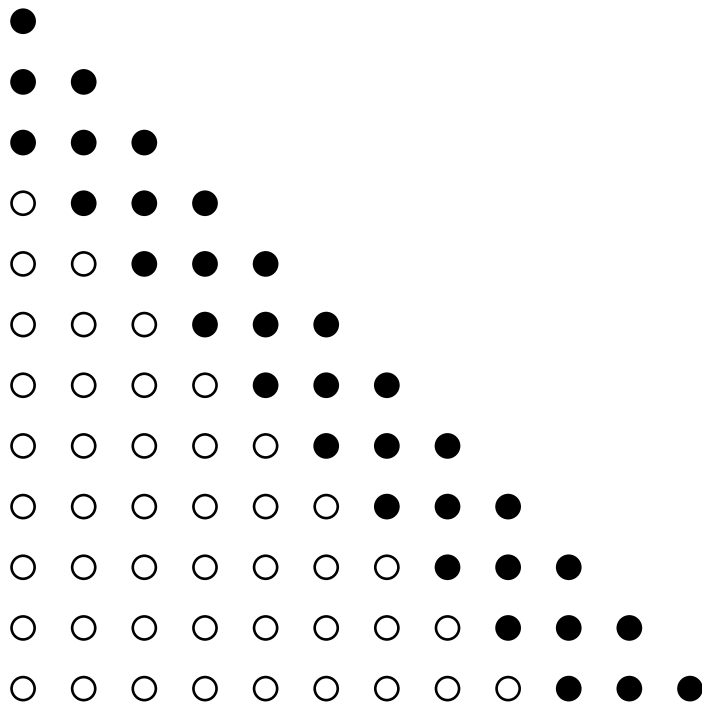


图 5.1 使用3个处理机求解下三角线性代数方程组



Back

Close



99/248

# 经典迭代法-Jacobi

考虑求解线性代数方程组

$$Ax = b \quad (5.4)$$

其中 $A$ 是 $m \times m$ 矩阵, 记 $D$ 、 $-L$ 、 $-U$ 分别是 $A$ 的对角、严格下三角、严格上三角部分构成的矩阵, 即 $A = D - L - U$ 。这时方程组(5.4)可以变为

$$Dx = b + (L + U)x \quad (5.5)$$

如果方程组(5.5)右边的 $x$ 已知, 由于 $D$ 是对角矩阵, 可以很容易求得左边的 $x$ , 这就是Jacobi迭代法的出发点。因此, 对于给定的初值 $x^{(0)}$ , Jacobi



Back

Close

迭代法如下：

$$x^{(k+1)} = D^{-1}(L + U)x^{(k)} + D^{-1}b \quad (5.6)$$

记  $G = D^{-1}(L + U) = I - D^{-1}A$ ,  $g = D^{-1}b$ 。则每次迭代就是做矩阵向量乘，然后是向量加。亦即：

$$x^+ = Gx + g \quad (5.7)$$

关于这个迭代法的并行计算问题，在习题4.9中对一种情况进行考虑，其它矩阵存储方式下的并行也同样可以完成，这里不再赘述。



100/248



Back

Close



# 经典迭代法-Gauss-Seidel

Gauss-Seidel迭代法是逐个分量进行计算的一种方法，考虑线性代数方程组(5.4)的分量表示

$$\sum_{j=1}^n a_{ij}x_j = b_i, \quad i = 1, \dots, n \quad (5.8)$$

对于给定的初值 $x^{(0)}$ ，Gauss-Seidel迭代法如下：

**算法 8** *Gauss-Seidel*迭代算法

$$k = 0$$

$$x_1^{(k+1)} = (b_1 - \sum_{j=2}^n a_{1j}x_j^{(k)})/a_{11}$$

$$x_2^{(k+1)} = (b_2 - a_{21}x_1^{(k+1)} - \sum_{j=3}^n a_{2j}x_j^{(k)})/a_{22}$$



Back

Close



...

$$x_{n-1}^{(k+1)} = (b_{n-1} - \sum_{j=1}^{n-2} a_{n-1,j} x_j^{(k+1)} - a_{n-1,n} x_n^{(k)}) / a_{n-1,n-1}$$

$$x_n^{(k+1)} = (b_n - \sum_{j=1}^{n-1} a_{nj} x_j^{(k+1)}) / a_{nn}$$

$$\|x^{(k+1)} - x^{(k)}\|_2 < \epsilon \|x^{(k+1)} - x^{(0)}\|_2? \quad k = k + 1$$

## 经典迭代法-Gauss-Seidel的并行化

记  $s_i = \sum_{j=i+1}^n a_{ij} x_j^{(0)}$ ,  $i = 1, \dots, n-1$ ,  $s_n = 0$ 。并行计算方法如下:

### 算法 9 并行 Gauss-Seidel 迭代算法

$k = 0$

for  $i = 1, n$  do



Back

Close



$$x_i^{(k+1)} = (b_i - s_i)/a_{ii}, s_i = 0$$

for  $j = 1, n, j \neq i$  do

$$s_j = s_j + a_{ji}x_i^{(k+1)}$$

end{for}

end{for}

$$\|x^{(k+1)} - x^{(k)}\|_2 < \epsilon \|x^{(k+1)} - x^{(0)}\|_2? k = k + 1$$

在算法9中，每次并行计算 $s_j$ ，之后可以并行计算截止条件是否满足。



Back

Close



## 习题

1. 如果矩阵是按照列循环方式存放在处理机中的，请问是否可以设计一个并行计算Gauss-Seidal的方法；
2. 请用自己的理解，对三角矩阵方程组的并行求解方法进行描述和解释。



Back

Close





## 六、FFT并行算法

在科学与工程计算中快速富氏变换应用非常广泛,自上个世纪60年代提出以来,已经被认为是上个世纪最伟大的算法之一。这里我们考虑FFT的并行算法以及如何实现。FFT考虑快速计算

$$y_k = \sum_{j=0}^{n-1} x_j e^{-\frac{2\pi i j k}{n}}, \quad k = 0, 1, \dots, n-1. \quad (6.1)$$

的问题,其中 $i^2 = -1$ 。记 $\omega(n) = e^{-\frac{2\pi i}{n}}$ ,则 $\omega(n)^k$ 是方程 $x^n = 1$ 的根。亦即有:

$$y_k = \sum_{j=0}^{n-1} x_j \omega^{jk}, \quad k = 0, 1, \dots, n-1. \quad (6.2)$$

这里 $e^{i\theta} = \cos(\theta) + i \sin(\theta)$ ,因此有如下Euler 方程:

$$e^{i\pi} + 1 = 0 \quad (6.3)$$



Back

Close



下面的性质显然成立:

1.  $(\omega(n)^k)^n = 1$
2.  $\omega(n)^2 = \omega(n/2)$
3.  $\omega(n)^{n/2} = -1$

记  $Y = (y_0, y_1, \dots, y_{n-1})^T$ ,  $X = (x_0, x_1, \dots, x_{n-1})^T$ ,  $\Omega_{kj} = \omega(n)^{kj}$ , 则式(6.1)的计算过程可以写成矩阵乘向量的形式  $Y = \Omega X$ 。因此, 直接计算式(6.1)需要  $O(n^2)$  个浮点运算。由于  $\omega(n)$  具有特殊性, 假设  $n = 2m$ , 式(6.2)可以分成如下的计算过程

$$\begin{cases} y_k = \sum_{j=0}^{m-1} x_{2j} \omega(m)^{kj} + \omega(n)^k \sum_{j=0}^{m-1} x_{2j+1} \omega(m)^{kj} \\ y_{k+m} = \sum_{j=0}^{m-1} x_{2j} \omega(m)^{kj} - \omega(n)^k \sum_{j=0}^{m-1} x_{2j+1} \omega(m)^{kj} \\ k = 0, 1, \dots, m-1 \end{cases} \quad (6.4)$$



Back

Close



107/248

假设 $T_n$ 是计算所有 $y_k$ 的计算量，由式(6.4)有 $T_n = 2T_{n/2} + 3/2n$ ，也即 $T_n = 3/2n \log_2 n + n$ 。在这里讨论的计算方法中，假定数据的长度是 $n = 2^m$ 。

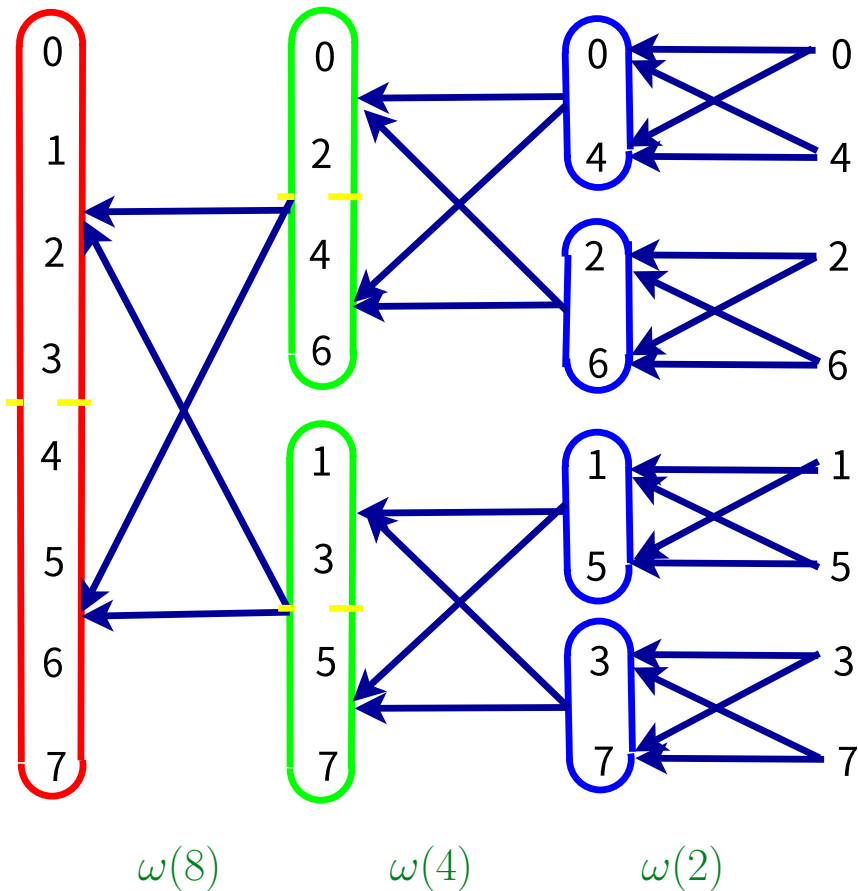


Back

Close



108/248



$\omega(8)$

$\omega(4)$

$\omega(2)$



Back

Close



# 一维串行FFT算法

公式(6.4)是FFT的一种计算方法基础，可以通过递推的方式来完成其计算任务。在进行FFT算法的执行过程中，需要对原始数据进行重排序，以利于计算。数据重新排列的规则是按位倒置 (bit reverse) 方式进行的，以 $n = 16$ 为例，按位倒置变换如表6.1所示（每次左移一位）。设2进制数为 $x_0x_1 \cdots x_{n-1}x_n$ ，其中 $x_i = 0, 1$ ，则按位倒置变换将其转变为 $x_nx_{n-1} \cdots x_1x_0$ 。

假设 $n = 2^m$ ，所有 $x_j$ 按照按位倒置规则进行重新排序，记为 $y_j$ 。令 $D(2k)$ 是 $k$ 阶对角矩阵，对角线上的元素为 $\omega(2k)^j$ ， $j = 0, \dots, k-1$ 。按照公式(6.4)，在时间上进行大幅度减少（decimation in time, DIT）的FFT算法如下：



Back

Close



表 6.1 按位倒置变换

原始顺序	第一次	第二次	第三次	十进制
0000	0000	0000	0000	0
0001	0010	0100	1000	8
0010	0100	1000	0100	4
0011	0110	1100	1100	12
0100	1000	0010	0010	2
0101	1010	0110	1010	10
0110	1100	1010	0110	6
0111	1110	1110	1110	14
1000	0001	0001	0001	1
1001	0011	0101	1001	9
1010	0101	1001	0101	5
1011	0111	1101	1101	13
1100	1001	0011	0011	3
1101	1011	0111	1011	11
1110	1101	1011	0111	7
1111	1111	1111	1111	15



Back

Close

## 算法 10 FFT计算方法

1. 置  $s = 1, t = 1, l = n/2$

2. 计算所有长度为  $2t$  的变换  $l$  个, 其中每个变换的形式为

$$Y = \begin{pmatrix} I & D(2t) \\ I & -D(2t) \end{pmatrix} Y$$

3. 如果  $s < m$ , 置  $s = s + 1, t = 2t, l = l/2$ , 重复上一步计算。

在这个算法中, 其计算过程是从计算长度为2的一些变换开始, 然后是长度为4 的一些变换, 最后得到长度为 $2^m$ 的变换。每次的计算是非常简单的, 比如已经有了2个长度为2的FFT的序列分别记为 $u, v$ , 则由



111/248



Back

Close

它们产生的长度为4的序列 $z$ 的计算公式如下：

$$\begin{cases} z_0 &= u_0 + \omega(4)^0 v_0 \\ z_1 &= u_1 + \omega(4)^1 v_1 \\ z_2 &= u_0 - \omega(4)^0 v_0 \\ z_3 &= u_1 - \omega(4)^1 v_1 \end{cases} \quad (6.5)$$



112/248



Back

Close





## 二维串行FFT算法

$$\left\{ \begin{aligned} y_{k_x k_y} &= \sum_{j_x=0}^{n_x-1} \sum_{j_y=0}^{n_y-1} x_{j_x j_y} e^{-\frac{2\pi i j_x k_x}{n_x}} e^{-\frac{2\pi i j_y k_y}{n_y}} \\ &= \sum_{j_y=0}^{n_y-1} \left( \sum_{j_x=0}^{n_x-1} x_{j_x j_y} e^{-\frac{2\pi i j_x k_x}{n_x}} \right) e^{-\frac{2\pi i j_y k_y}{n_y}} \\ &= \sum_{j_y=0}^{n_y-1} z_{k_x j_y} e^{-\frac{2\pi i j_y k_y}{n_y}} \\ k_x &= 0, 1, \dots, n_x - 1, k_y = 0, 1, \dots, n_y - 1 \end{aligned} \right. \quad (6.6)$$

从公式(6.6)不难得出,  $y_{k_x k_y}$  的计算过程可以由两个方向的一维FFT来完成。



Back

Close



# FFT实现技术

从计算公式 (6.4) 可以得出, 对于每个  $k$  需要计算  $\omega(n)^k$ , 其中  $0 \leq k < n/2$ 。因此计算过程需要计算一系列的型如  $\omega(m)^k$  的值, 假设  $m \times 2^l = n$ , 由  $\omega(m)$  的性质可知,  $\omega(n)^{2^l} = \omega(m)$ , 且不难得出  $2^l k < n/2$ , 从而有  $\omega(m)^k = \omega(n)^{2^l k}$ 。据此可知  $\omega(m)^k$  也是方程  $x^n = 1$  的一个根。

至此, 是否清楚上述讨论的目的? 在实现算法 10 的时候, 我们只需要开始时计算出  $\omega(n)^k$ , 其中  $0 \leq k < n/2$ , 亦即方程  $x^n = 1$  的一半根。另一半根在计算过程中是不需要的, 即使需要, 也可以由  $\omega(n)^{(n/2 + k)} = -\omega(n)^k$  获得。

如何得到按位倒置顺序 (表 6.1) 是实现中的难点? 下面将给出一种方法 (参见 6.4) 的两种实现方式, 供大家参考。



Back

Close

## 算法 11 按位倒置实现方式I

```
void oddeven( n, narray )
int n, *narray;
{
    int nh, i, j, mid;

    if( n == 2 ) return;

    nh = n>>1;
    for(i=1; i<nh; i++)
    {
        mid = narray[2*i];
        for(j=2*i; j>i; j--) narray[j] = narray[j-1];
```



115/248



Back

Close



116/248

```
        narray[i]=mid;
    }
    oddeven(nh, narray);
    oddeven(nh, &narray[nh]);

    return;
}
```

### 算法 12 按位倒置实现方式II

```
void icopy(n, x, ix, y, iy)
int n, ix, iy; int *x, *y;
{
    int i, inx=0, iny=0;
    for(i=0; i<n; i++) {
```



Back

Close



117/248

```
y[iny] = x[inx];  
inx += ix; iny +=iy;  
}
```

```
return;  
}
```

```
void moddeven( n, narray, nwork )  
int n, *narray, *nwork;  
{
```

```
    int nh;
```

```
    if( n == 2 ) return;
```



Back

Close



118/248

```
nh = n>>1;
icopy( nh, narray, 2, nwork, 1 );
icopy( nh, &narray[1], 2, &nwork[nh], 1 );
icopy( n, nwork, 1, narray, 1 );
moddeven(nh, narray, nwork);
moddeven(nh, &narray[nh], &nwork[nh]);

return;
}
```

接下来我们给出一种只是要得到按位倒置排序的方法，这种方法  
是前面方法的逆，亦即是相反的操作，此方法简单有效。

### 算法 13 按位倒置实现方式III

```
void bitreverse( n, narray )
```



Back

Close



119/248

```
int n, *narray;
{
    int n2, i;

    if( n < 2 ) { narray[0]=0; return;}
    n2 = 2;
    narray[0]=0; narray[1]=1;
    while( n2 < n ) {
        for(i=0; i<n2; i++) {
            narray[i] *=2; narray[i+n2] = narray[i]+1;}
        n2 *=2; }

    return;
```



Back

Close

```
}
```

## 算法 14 FFT 实现程序

```
/*given the input x, y=Ax, where  $a_{ij}=\exp(2\pi i \cdot I_{ij}/n)$ ,  
  A is FT matrix  
*/  
void cxbfft(n, x, y) int n; double complex *x, *y; {  
    int m, l2, i, j, *bitr, mh, sw, stp;  
    double complex *roots, *wroots;  
  
    mh = 1;  
    m = 2;  
    l2 = n >> 1;  
    bitr =(int *)malloc(n*sizeof(int));
```



120/248



Back

Close





121/248

```
roots=(double complex *)malloc(l2*sizeof(double complex));
wroots=(double complex *)malloc(l2*sizeof(double complex));

oneroots( n, roots );
bitreverse( n, bitr );
reorder( n, bitr, x );
free( bitr );

sw = 0;
while( mh < n ) {
    getwmk(n, m, l2, roots, wroots);
/*here compute the transform of length m*/
    if( sw == 0 ) {
        sw = 1;
```



Back

Close



122/248

```
    stp = 0;
    for(i=0; i<l2; i++) {
        for(j=0; j<mh; j++) x[stp+j+mh] *= wroots[j];
        for(j=0; j<mh; j++) {
            y[stp+j] = x[stp+j]+x[stp+j+mh];
            y[stp+j+mh] = x[stp+j]-x[stp+j+mh];
        }
        stp += m;
    }
}

else {
    sw = 0;
    stp = 0;
```



Back

Close



123/248

```
for(i=0; i<12; i++) {  
    for(j=0; j<mh; j++) y[stp+j+mh] *= wroots[j];  
    for(j=0; j<mh; j++) {  
        x[stp+j] = y[stp+j]+y[stp+j+mh];  
        x[stp+j+mh] = y[stp+j]-y[stp+j+mh];  
    }  
    stp += m;  
}  
}  
  
mh = m;  
m = m * 2;  
12 = 12 / 2;  
}
```



Back

Close

```
if( sw == 0)
    for(i=0; i<n; i++)
        y[i] = x[i];
free(roots);
free(wroots);
return;
}
```



124/248



Back

Close



## 习题

1. 令 $n = 2^m$ , 请写一个子程序计算 $x^n = 1$ 的所有根;
2. 证明按位倒置排序中需要调整的数据是对调;
3. 按照算法 13给出的按位倒置排序, 请写一个子程序对数组 $x$  进行重新排序。



Back

Close



126/248

## 七、*MPI*并行程序设计

这里我们将重点介绍*SPMD* (*single program multi data*) 并行程序, 它是其它并行程序如主从式等的基础。因此, 熟练掌握*SPMD*并行程序的设计与实现是学习的核心。

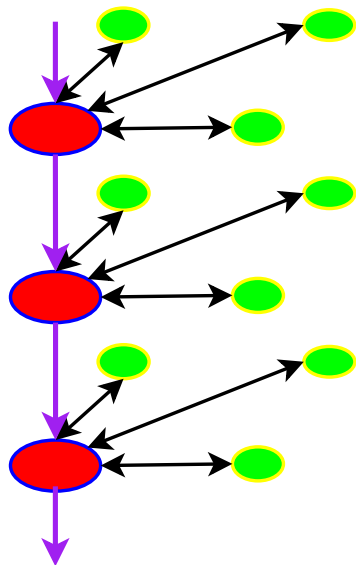
以*SPMD*并行程序为目标, 介绍*MPI*并行程序是如何实现的。



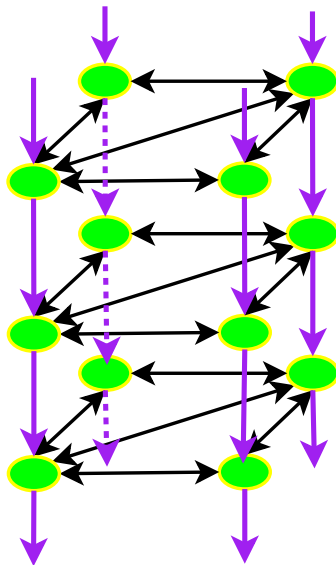
Back

Close

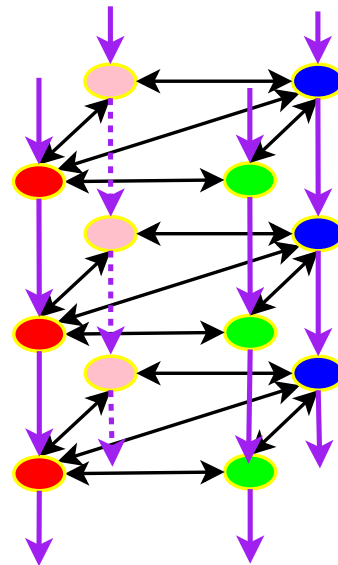
# 并程序类型



主从式M-S



对称式SPMD



自主式MPMD





# MPI并行政程的基本结构

1. 进入MPI环境。产生通讯子(*communicator*) (进程序号、进程数);
2. 程序主体。实现计算的全部内容;
3. 退出MPI环境。不能再使用MPI环境

记*iam*表示进程序号, 那么如下一段FORTRAN程序:

```
m = iam + 2  
if ( iam .eq. 0 ) then  
    m = 10  
endif
```

或者是:



Back

Close





129/248

```
if ( iam .eq. 0 ) then
    m = 10
else
    m = iam + 2
endif
```

产生了 $m$ 的值, 0进程是10, 1进程是3, 2进程是4,  $\dots$ 。

*Alan Perlis said "You think you know when you can learn, are more sure when you can write, even more when you can teach, but certain when you can program."*

不积跬步, 无以至千里; 不积小流, 无以成江海; 道阻且长, 行则将至; 路虽迢, 不行不至; 事虽小, 不为不成。



Back

Close



# MPI并行环境管理函数

## MPI\_INIT

<i>C</i>	<code>int MPI_Init( int *argc, char ***argv )</code>
<i>Fortran</i>	<code>MPI_INIT( IERROR )</code> <code>INTEGER IERROR</code>

调用MPI\_INIT函数之后，系统产生一个通讯子（*communicator*）称之为MPI\_COMM\_WORLD，这个通讯子将贯穿MPI并行政程序设计的始终。除MPI\_INITIALIZED之外，使用MPI的其它函数必须在此函数之后。

## MPI\_FINALIZE

<i>C</i>	<code>int MPI_Finalize( void )</code>
<i>Fortran</i>	<code>MPI_FINALIZE( IERROR )</code> <code>INTEGER IERROR</code>



Back

Close

使用MPI\_FINALIZE函数之后，任何MPI函数都不能再调用。

## MPI\_INITIALIZED

---

<i>C</i>	int MPI_Initialized( int flag )
----------	---------------------------------

---

<i>Fortran</i>	MPI_INITIALIZED( FLAG, IERROR )
----------------	---------------------------------

---

LOGICAL FLAG

INTEGER IERROR

---



131/248



Back

Close



# MPI通信子操作

## MPI\_COMM\_SIZE

---

*C*     `int MPI_Comm_size( MPI_Comm comm, int *size )`

---

*Fortran*   `MPI_COMM_SIZE( COMM, SIZE, IERROR )`  
`INTEGER COMM, SIZE, IERROR`

---

## MPI\_COMM\_RANK

---

*C*     `int MPI_Comm_rank( MPI_Comm comm, int *rank )`

---

*Fortran*   `MPI_COMM_RANK( COMM, RANK, IERROR )`  
`INTEGER COMM, RANK, IERROR`

---

## MPI\_COMM\_DUP

---

*C*     `int MPI_Comm_dup( MPI_Comm comm, MPI_Comm *newcomm )`

---

*Fortran*   `MPI_COMM_DUP( COMM, NEWCOMM, IERROR )`  
`INTEGER COMM, NEWCOMM, IERROR`

---



Back

Close



此函数产生的通讯子与原通讯子完全一致，方便用户自定义通讯子。

## MPI\_COMM\_SPLIT

---

<i>C</i>	<code>int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)</code>
----------	---

---

<i>Fortran</i>	<code>MPI_COMM_SPLIT( COMM, COLOR, KEY, NEWCOMM, IERROR ) INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR</code>
----------------	--

---

在MPI\_COMM\_SPLIT函数中，COLOR和KEY都是非负整数。KEY是确定新通信子中成员的顺序，COLOR相同的构成一个通信子，如果其取值为MPI\_UNDEFINED，所得到的通信子为MPI\_COMM\_NULL。

目前我们介绍的这几个MPI函数是并程序中极其重要的，有了第 7.3和第 7.4 节的函数，我们就可以编写简单的并程序。

### 例子 4 简单并程序

```
#include "myhead.h"
```

[Back](#)[Close](#)



134/248

```
/*Start the main program*/  
  
void main(argc, argv)  
  
int argc; char **argv;  
  
{  
  
    int iam, np;  MPI_Comm comm;  
  
  
  
  
  
  
  
  
  
/*Start MPI environment*/  
  
    MPI_Init( &argc, &argv );  
  
    MPI_Comm_dup( MPI_COMM_WORLD, &comm );  
  
    MPI_Comm_rank( comm, &iam );  
  
    MPI_Comm_size( comm, &np );  
  
/*MAIN work here */  
  
    printf("\nThe process %d of %d is running!\n", \
```



Back

Close

```
    iam, np);
```

```
/*finished, terminate the MPI*/
```

```
    MPI_Finalize( );
```

```
}
```



135/248



Back

Close



## 八、点到点通信函数

这部分内容是MPI通讯的核心，对阻塞式和非阻塞式发送（send）和接收（recv）进行阐述，同时结合具体实例，理解发送与接收函数的作用。

[Back](#)[Close](#)





# 阻塞式SEND和RECV

这2个函数是MPI通讯函数的基础，也是编写并行计算程序的实质性函数。可以说，了解了通讯的要求，几乎所有并行算法的实现都可以由这2个最基本的通讯函数来完成。

## MPI\_SEND

---

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

---

```
Fortran  MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR )
          <type> BUF(*)
          INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

---



Back

Close



138/248

## MPI\_RECV

---

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

---

```
Fortran  MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS,
                IERROR )
```

```
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
STATUS(MPI_STATUS_SIZE), IERROR
```

---

MPI\_STATUS的三个成员是MPI\_SOURCE, MPI\_TAG和MPI\_ERROR。

## SEND和RECV函数的基本数据类型

Fortran程序可以使用的数据类型：

---

```
MPI_INTEGER  MPI_REAL      MPI_DOUBLE_PRECISION
MPI_COMPLEX  MPI_LOGICAL  MPI_CHARACTER
MPI_PACKED   MPI_BYTE
```

---



Back

Close



## C程序可以使用的数据类型:

MPI_CHAR	MPI_SHORT	MPI_INT
MPI_LONG	MPI_UNSIGNED_CHAR	MPI_UNSIGNED_SHORT
MPI_UNSIGNED_INT	MPI_UNSIGNED_LONG	MPI_FLOAT
MPI_DOUBLE	MPI_LONG_DOUBLE	MPI_PACKED
MPI_BYTE		

在使用SEND和RECV函数时, 需要注意发送和接收的数据类型的一致性, 避免运行错误。

## 应用发送和接收函数实例

### 例子 5 传送数据至下一个进程

```
void ring_s_r( m, iam, np, comm, n )  
  
int m, iam, np *n;  
  
MPI_Comm comm;
```



Back

Close



140/248

```
{  
    int next=(iam+1) % np, front=(np+iam-1) % np, \  
        tag = 1;  
    MPI_Status st;  
  
    if( iam == 0 ) {  
        MPI_Send( &m, 1, MPI_INT, next, tag, comm );  
        MPI_Recv( n, 1, MPI_INT, front, tag, comm, &st );  
    }  
    else {  
        MPI_Recv( n, 1, MPI_INT, front, tag, comm, &st );  
        MPI_Send( &m, 1, MPI_INT, next, tag, comm );  
    }  
}
```



Back

Close

```
return;  
}
```

此程序可以完成从本进程向下一个进程传送数据的作用，同时，接收从前面进程发来的数据。然而，我们不难发现，如此做法是顺序执行的，没有并行执行消息传递，因此可以说不是一个好的并行程序设计方案。是否存在一个优秀设计方案，使得实现过程尽可能优化？

### 例子 6 数值求解二维泊松方程

$$\begin{aligned} -\Delta u(x, y) &= f(x, y), (x, y) \in \Omega = [0, \pi] \times [0, \pi] \\ u(x, y)|_{\partial\Omega} &= 0 \end{aligned} \quad (8.1)$$

这里我们采用5点中心差分格式，记  $h = \pi/(n+1)$ ,  $x_0 = 0$ ,  $y_0 = 0$ ,  $x_i = x_0 + ih$ ,  $y_i = y_0 + ih$ ,  $u_{ij} = u(x_i, y_j)$ ,  $f_{ij} = f(x_i, y_j)$ 。则有





如下方程组:

$$4u_{ij} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = h^2 f_{ij}, i, j = 1, n \quad (8.2)$$

根据 (8.1) 提供的边界条件, 进一步可以有:

$$\begin{aligned} 4u_{11} - u_{2,1} - u_{1,2} &= h^2 f_{11} \\ 4u_{1j} - u_{2,j} - u_{1,j-1} - u_{1,j+1} &= h^2 f_{1j}, j = 2, n-1 \\ 4u_{1n} - u_{2,n} - u_{1,n-1} &= h^2 f_{1n} \end{aligned} \quad (8.3)$$

如果采用Jacobi (参见 5.4) 迭代法求解, 使残量相对误差小于 $\epsilon$ 。  
对于给定的初始值 $x^0$ , 记 $r^k = b - Ax^k$ , 迭代终止条件为:

$$\|r^k\|_2 / \|r^0\|_2 < \epsilon \quad (8.4)$$

从而有:

$$x^{k+1} = D^{-1}(b + (L + U)x^k) = D^{-1}r^k + x^k \quad (8.5)$$

使用Jacobi迭代法的并程序如下:



Back

Close

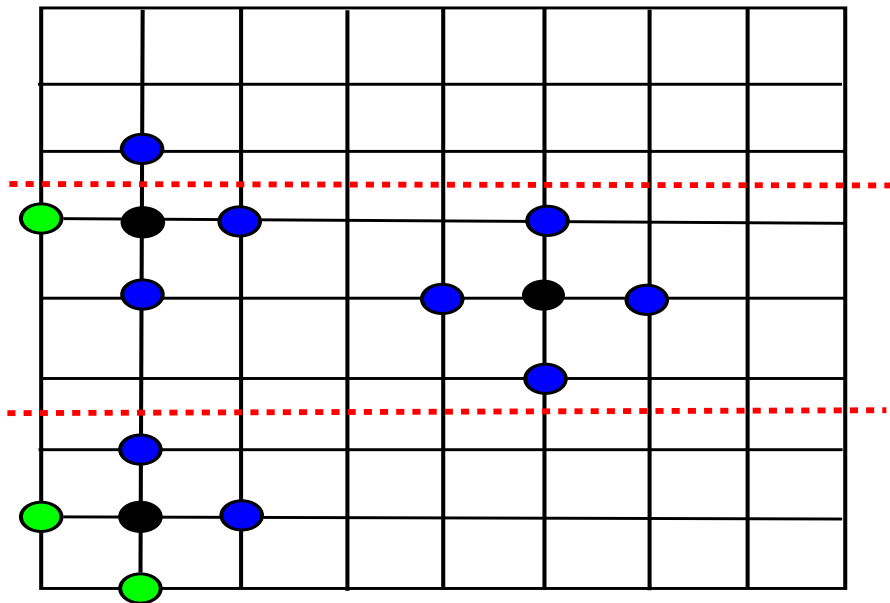


图 8.1 strip partitioning domain



Back

Close



144/248

```
#include "myhead.h"

#include <math.h>

/* solving Poisson equations with zero boundary values,
 * partitioning in row direction */

float sqnorm2(a, lda, m, n)
int lda, m, n;
float *a;
{
    int i, j;
    float r=0.0;
```



Back

Close





145/248

```
for(i=0; i<m; i++)
    for(j=0;j<n; j++)
        r += a[i*lda+j]*a[i*lda+j];

return r;
}

/* f = 2*sin(x)*sin(y) */
void righthts( h, frhs, lds, m, n, iam, np)
int lds, m, n, iam, np;
float h, *frhs;
{
    int i, j, ioff, ir;
```



Back

Close



146/248

```
ir = n % np;
ioff = n / np;
ioff = iam*ioff;
if( iam < ir ) ioff += iam;
else ioff += ir;

for(i=1; i<=m; i++)
    for(j=1; j<n; j++)
        frhs[i*lds+j]= 2.0*sinf((i+ioff)*h)*sinf(j*h);

return;
}
```



Back

Close



147/248

```
void laplace(comm, iam, np, h, work, lds, m, n, frhs, sol)
MPI_Comm comm;

int iam, np, lds, m, n;
float h, *work, *frhs, *sol;
{
    int i, j, k, front, next, max_iter=200;
    float sh=h*h, epsilon=1.0e-3, residual, r0, rt;
    MPI_Status st;

    front = (np+iam-1)%np;
    next = (iam+1)%np;
    if( iam == 0 ) front = MPI_PROC_NULL;
```



Back

Close



148/248

```
if( iam == np-1 ) next = MPI_PROC_NULL;
/* initialize the solver values */
for(i=0; i<m+2; i++)
    for(j=0; j<n+2; j++)
        sol[i*lds+j] = 0.0;
/* initial residual value */
rt = sqnorm2( &frhs[lds+1], lds, m, n );
MPI_Allreduce( &rt, &r0, 1, MPI_FLOAT, MPI_SUM, comm );
r0 = sqrtf( r0 )*sh;

/* iterate for solving discretized equations */
for(k=0; k<max_iter; k++) {
```



Back

Close



149/248

```
/* calculate the residual in 5 point DS*/  
for(i=1; i<=m; i++)  
    for(j=1; j<=n; j++)  
        work[i*lds+j]=sh*frhs[i*lds+j]+sol[i*lds+j-1]+ \  
            sol[i*lds+j+1]+sol[(i-1)*lds+j]+ \  
            sol[(i+1)*lds+j]-4.0*sol[i*lds+j];  
  
rt = sqnorm2( &work[lds+1], lds, m, n );  
MPI_Allreduce( &rt, &residual, 1, MPI_FLOAT,\br/>               MPI_SUM, comm );  
  
residual = sqrtf( residual )/r0;  
if( epsilon < residual ) break;
```



Back

Close



150/248

```
/* update the new solution */  
for(i=1; i<=m; i++)  
    for(j=1; j<=n; j++)  
        sol[i*lds+j]+=0.25*work[i*lds+j];  
  
/* send the sol 1st row to front process */  
MPI_Sendrecv( &sol[lds+1], n, MPI_FLOAT, front, 1, \  
              &sol[(m+1)*lds+1], n, MPI_FLOAT, \  
              next, 1, comm, &st );  
  
/* send the sol last row to next process */  
MPI_Sendrecv( &sol[m*lds+1], n, MPI_FLOAT, next, 2, \  
              &sol[1], n, MPI_FLOAT, front, 2, \  
              comm, &st );
```



Back

Close

```
}
```

```
return;
```

```
}
```



151/248



Back

Close



# 合成函数SENDRECV

## MPI\_SENDRECV

---

```
C    int MPI_Sendrecv( void *sendbuf, int sendcount,
MPI_Datatype sendtype, int dest, int sendtag,
void *recvbuf, int recvcount, MPI_Datatype recvtpe,
int source,int recvtag,MPI_Comm comm,MPI_Status *status)
```

---

```
Fortran MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG,
RECVBUF, REVCOUNT, RECVTPE, SOURCE, RECVTAG, COMM,
STATUS, IERROR )

<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, REVCOUNT,
      RECVTPE, SOURCE, RECVTAG, COMM,
      STATUS(MPI_STATUS_SIZE), IERROR
```

---

值得注意的是，函数SEND、RECV、SENDRECV互相兼容，即SEND发送的消息可以用SENDRECV来接收，而SENDRECV发送的消息也可以用RECV接

[Back](#)[Close](#)



收。

## 合成函数SENDRECV\_Replace

### MPI\_SENDRECV\_REPLACE

---

C	<pre>int MPI_Sendrecv_replace( void *buf, int count, MPI_Datatype type, int dest, int sendtag, int source,int recvtag,MPI_Comm comm,MPI_Status *status)</pre>
Fortran	<pre>MPI_SENDRECV_REPLACE(BUF, COUNT, TYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR )  &lt;type&gt; BUF(*)  INTEGER COUNT, TYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR</pre>

---

在使用此函数时，最好与8.2节定义的函数进行比较，通常来说，MPI\_SENDRECV\_REPLACE效率低下。

在MPI接收消息时，不需要指明消息是从哪个进程来的，可以采用通配符的方式，即`source = MPI_ANY_SOURCE`，`tag = MPI_ANY_TAG`。同时为方



153/248



Back

Close

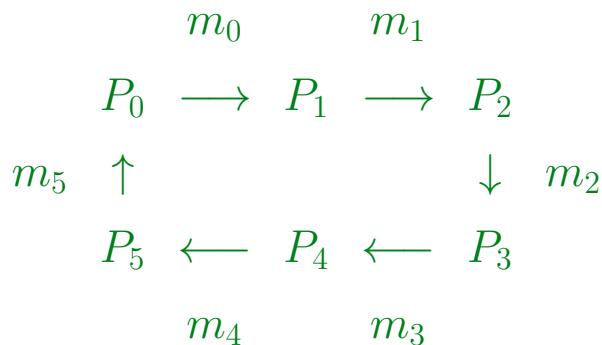


便使用MPISENDRECV函数, MPI还定义了一个空进程, 用MPIPROC\_NULL来表示。

## 复合函数的作用

以上是点到点阻塞式通讯的一些基本函数, 是MPI程序的基础。现在我们来了解一下例子 5 的通讯部分, 可以改写成如下:

```
MPI_Sendrecv( &m, 1, MPI_INT, next, tag, n, 1, \
               MPI_INT, front, tag, comm, &st );
```

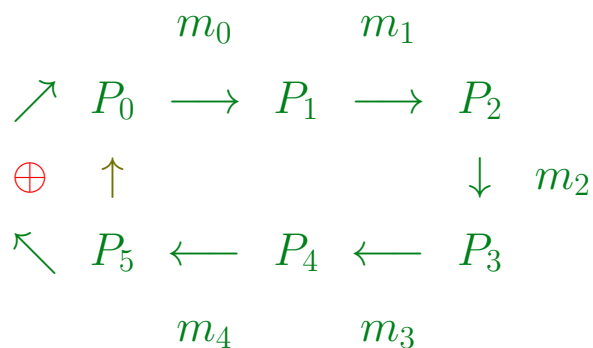


Back

Close



如果在通讯过程中，这里 $P_5$ 不发送消息给 $P_0$ ，例子 5将如何适应之。



Back

Close



# 消息查询函数

## MPI\_PROBE

---

```
C      int MPI_Probe(int source, int tag, MPI_Comm comm,  
                    MPI_Status *status )
```

---

```
Fortran  MPI_PROBE( SOURCE, TAG, COMM, STATUS, IERROR )  
          INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),  
          IERROR
```

---

## MPI\_IPROBE

---

```
C      int MPI_Iprobe(int source, int tag, MPI_Comm comm,  
                     int* flag, MPI_Status *status )
```

---

```
Fortran  MPI_IPROBE( SOURCE, TAG, COMM, FLAG, STATUS, IERROR )  
          LOGICAL FLAG  
          INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),  
          IERROR
```

---



Back

Close



## MPI\_GET\_COUNT

---

```
C      int MPI_Get_count( MPI_Status *status, MPI_Datatype type,  
                        int *count )
```

---

```
Fortran  MPI_GET_COUNT( STATUS, TYPE, COUNT, IERROR )  
          INTEGER STATUS(MPI_STATUS_SIZE), TYPE, COUNT, IERROR
```

---

在MPI中，还有一个与之类似的函数MPI\_GET\_ELEMENT，一般情况下，二者相同，只有在特殊的情况下，只能使用MPI\_GET\_ELEMENT才能得到所需要的数据类型的个数。这二个函数的参数是完全一致的，意义相同。



Back

Close



# 非阻塞式ISEND和IRECV

## MPI\_ISEND

---

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

---

```
Fortran  MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,  
                  IERROR )  
  
         <type> BUF(*)  
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

---



Back

Close

## MPI\_IRecv

---

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

---

```
Fortran  MPI_IRecv( BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST,  
                 IERROR )  
  
          <type> BUF(*)  
          INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

---



159/248



Back

Close



# 消息请求完成函数

## MPI\_WAIT

---

C	<code>int MPI_Wait( MPI_Request *request, MPI_Status *status )</code>
Fortran	<code>MPI_WAIT(REQUEST, STATUS, IERROR )</code> <code>INTEGER REQUEST, STATUS(*), IERROR</code>

---

## MPI\_WAITANY

---

C	<code>int MPI_Waitany( int count, MPI_Request *requests,</code> <code>int *index, MPI_Status *status )</code>
Fortran	<code>MPI_WAITANY( COUNT, REQUESTS, INDEX, STATUS, IERROR )</code> <code>INTEGER COUNT, REQUESTS(*), INDEX, STATUS(*), IERROR</code>

---



Back

Close





## MPI\_WAITALL

---

```
C      int MPI_Waitall( int count, MPI_Request *requests,
                      MPI_Status *statuses )
```

---

```
Fortran  MPI_WAITALL( COUNT, REQUESTS, STATUSES, IERROR )
          INTEGER COUNT, REQUESTS(*), STATUS(*, *), IERROR
```

---

## MPI\_WAITSOME

---

```
C      int MPI_Waitsome( int incount, MPI_Request *requests,
                       int *outcount,int *indices, MPI_Status *statuses )
```

---

```
Fortran  MPI_WAITSOME( INCOUNT, REQUESTS, OUTCOUNT, INDICES,
                     STATUSES, IERROR )
          INTEGER INCOUNT, REQUESTS(*), OUTCOUNT, INDICES(*),
                     STATUS(MPI_STATUS_SIZE, *), IERROR
```

---

以上这些请求完成函数需要与非阻塞式通讯函数相结合使用，保障所请求的消息在一定时候必须完成。



Back

Close



# 消息请求检查函数

## MPI\_TEST

---

C	<code>int MPI_Test( MPI_Request *request, int *flag, MPI_Status *status )</code>
Fortran	<code>MPI_TEST(REQUEST, FLAG, STATUS, IERROR )</code> <code>INTEGER REQUEST, STATUS(*), IERROR</code> <code>LOGICAL FLAG</code>

---

## MPI\_TESTANY

---

C	<code>int MPI_Testany( int count, MPI_Request *requests,</code> <code>int *index, int *flag, MPI_Status *status )</code>
Fortran	<code>MPI_TESTANY( COUNT, REQUESTS, INDEX, FLAG, STATUS, IERROR )</code> <code>INTEGER COUNT, REQUESTS(*), INDEX, STATUS(*), IERROR</code> <code>LOGICAL FLAG</code>

---



Back

Close



163/248

## MPI\_TESTALL

---

```
C      int MPI_Testall( int count, MPI_Request *requests,
                        int *flag, MPI_Status *statuses )
```

---

```
Fortran  MPI_TESTALL( COUNT, REQUESTS, FLAG, STATUSES, IERROR )
          INTEGER COUNT, REQUESTS(*), STATUS(*, *), IERROR
          LOGICAL FLAG
```

---

## MPI\_TESTSOME

---

```
C      int MPI_Testsome( int incount, MPI_Request *requests,
                        int *outcount,int *indices, MPI_Status *statuses )
```

---

```
Fortran  MPI_TESTSOME( INCOUNT, REQUESTS, OUTCOUNT, INDICES,
                        STATUSES, IERROR )
          INTEGER INCOUNT, REQUESTS(*), OUTCOUNT, INDICES(*),
                        STATUS(MPI_STATUS_SIZE, *), IERROR
```

---



Back

Close



# 持久通讯函数SEND\_INIT和RECV\_INIT

## MPI\_SEND\_INIT

---

```
C      int MPI_Send_Init( void* buf, int count, MPI_Datatype type,
                          int dest, int tag, MPI_Comm comm, MPI_Request *request )
```

---

```
Fortran MPI_SEND_INIT(BUF,COUNT,TYPE,DEST,TAG,COMM,REQUEST,IERROR)
        <type> BUF(*)
        INTEGER COUNT, TYPE, DEST, TAG, COMM, REQUEST, IERROR
```

---

## MPI\_RECV\_INIT

---

```
C      int MPI_Recv_init( void* buf, int count, MPI_Datatype type,
                          int src, int tag, MPI_Comm comm, MPI_Request *request )
```

---

```
Fortran MPI_RECV_INIT(BUF,COUNT,TYPE,SRC,TAG,COMM,REQUEST,IERROR)
        <type> BUF(*)
        INTEGER COUNT, TYPE, SRC, TAG, COMM, REQUEST, IERROR
```

---



Back

Close



## MPI\_START

C	<code>int MPI_Start( MPI_Request *request )</code>
Fortran	<code>MPI_START(REQUEST, IERROR )</code> <code>INTEGER REQUEST, IERROR</code>

## MPI\_STARTALL

C	<code>int MPI_Startall( int count, MPI_Request *requests )</code>
Fortran	<code>MPI_STARTALL(COUNT, REQUESTS, IERROR )</code> <code>INTEGER COUNT, REQUESTS(*), IERROR</code>

## 消息请求的释放与取消

在调用MPI\_Send\_init或者MPI\_Recv\_init完成执行过程之后，消息请求句柄还停留在系统中，为避免引起系统资源枯竭，需要将请求句柄赋值为MPI\_REQUEST\_NULL。这时需要使用如下函数：



Back

Close



## MPI\_REQUEST\_FREE

C	<code>int MPI_Request_free( MPI_Request *request )</code>
Fortran	<code>MPI_REQUEST_FREE(REQUEST, IERROR )</code> <code>INTEGER REQUEST, IERROR</code>

对于使用MPI\_Isend和MPI\_Irecv 产生的请求句柄，在调用MPI\_Wait 系列函数之后，**将不能再调用此函数！** 切记！

当想撤销MPI\_Isend或者MPI\_Irecv 的消息时，MPI提供了如下函数：

## MPI\_CANCEL

C	<code>int MPI_Cancel( MPI_Request *request )</code>
Fortran	<code>MPI_CANCEL(REQUEST, IERROR )</code> <code>INTEGER REQUEST, IERROR</code>

取消一个消息的操作是局部性的，它将立即返回。因此，需要结合MPI\_Wait或者MPI\_Test。这2个函数返回的STATUS 将被用在下面的函数中。

[Back](#)[Close](#)

## MPI\_TEST\_CANCELLED

---

C	int MPI_Test_cancelled( MPI_Status *status, int *flag )
---	---

---

Fortran	MPI_TEST_CANCELLED(STATUS, FLAG, IERROR )
---------	---

LOGICAL FLAG

INTEGER STATUS(\*), IERROR

---



167/248



Back

Close



# 高维进程

我们知道在进入MPI环境之后，系统对每个进程进行编号，想知道自己是哪个进程，可以通过MPI\_Comm\_rank获得。由于这样产生的进程是1维的，对于绝大部分MPI并行程序是能够满足需要的，然而还有些并行程序只用1维进程就非常不方便，比如我们之前讲到的Cannon 4.7算法。因此，在实际并行程序中，对于进程的高维表示是不可或缺的。如何能够做到这一点？下面仅以怎样获得2维进程表示为例加以阐述。

假设我们需要 $p$ 行 $q$ 列的进程表示，那么，在进入MPI并行环境时所产生的进程数 $n_p$ 必须大于等于 $p \times q$ ，不然无法产生所需要的2维进程。按照行优先的方式 $P_{iam}$ 与 $P_{ij}$ 之间的映射为：

$$i = iam / q \quad j = iam \mod q$$

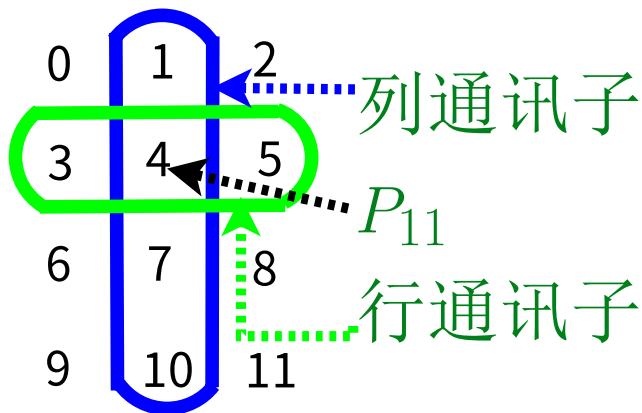


Back

Close



这个映射是一对一的。



仅有这样的映射关系难以满足程序实现的需求，通常需要利用2维表示的优势，亦即在行方向和列方向进行通信，这就是每个 $P_{ij}$ 进程具备行和列通讯的便利条件，为此需要行通讯子以及列通讯子。哪里可以获得这2个通讯子？在前面第 7.4节中，我们介绍了MPI\_Comm\_split，现在正是发挥此函数作用的时机，是否已经清楚和知道怎么实现？

```
void mesh( iam, np, comm, p, q, myrow, mycol, \
```



169/248



Back

Close



170/248

```
        rowcom, colcom )

int iam, np, p, q, *myrow, *mycol;
MPI_Comm comm, *rowcom, *colcom;
{
    int color, key;

    if( np < p*q ) return;

    if( iam < p*q ) color = iam / q;
    else color = MPI_UNDEFINED;
    key = iam;
    MPI_Comm_split( comm, color, key, rowcom );
    /*column communicator*/
}
```



Back

Close



171/248

```
if( iam < p*q ) color = iam % q;
else color = MPI_UNDEFINED;
key = iam;
MPI_Comm_split( comm, color, key, colcom );

if( iam < p*q ) {
    MPI_Comm_rank( *colcom, myrow );
    MPI_Comm_rank( *rowcom, mycol );
}
return;
}
```



Back

Close



## 习题

假设  $n = m \times p$ ,  $A$  是  $n \times n$  的矩阵, 并行计算  $y = Ax + b$ 。其中  $A$  按列分块存放在处理机中, 亦即处理机  $P_i$  中存放  $A$  的第  $i$  个列块, 仍记为  $A$ , 这时  $A$  是  $n \times m$  矩阵。在每个处理机中的  $x$  是能够与  $A$  相乘的部分, 也就是一个  $m$  维向量。  $b$  存放在  $P_0$  中,

1. 给出并行计算  $y$  的方法;
2. 依据给出的方法, 写出计算  $y$  的子程序。如果把计算结果最终放到  $x$  中, 程序是怎样的?



Back

Close



## 九、自定义数据类型

数据类型的基本结构:

$$\text{Typemap} = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$$

$$\text{Typemap} = \{(type_0, disp_0, len_0), \dots, (type_{n-1}, disp_{n-1}, len_{n-1})\}$$

自定义数据类型是MPI的重要组成部分，也是最能发挥用户的主观能动性。同时对于发挥MPI程序的性能至关重要，能够充分体现设计水平。



Back

Close



# CONTIGUOUS数据类型

考虑计算机中存放的一个数据为：

$a_0$	$a_1$	$a_2$	$a_3$	$a_4$	$\dots$	$\dots$	$a_n$
-------	-------	-------	-------	-------	---------	---------	-------

将红色部分定义成一个新的数据类型可由如下函数完成。

## MPI\_TYPE\_CONTIGUOUS

---

```
C      int MPI_Type_contiguous(int count,MPI_Datatype oldtype,  
                               MPI_Datatype *newtype )
```

---

```
Fortran  MPI_TYPE_CONTIGUOUS( COUNT, OLDTYPE, NEWTYPE, IERROR )  
          INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

---

**例子 7** 如果我们定义连续的2个实数为一个新的数据类型NEWTYPE, FORTRAN程序如下：

[Back](#)[Close](#)



175/248

```
call mpi_type_contiguous(2,mpi_real,newtype,ierr)
```

因此, 如果原程序是:

```
call mpi_send(a, 10, mpi_real, 1, 99, comm, ierr)
```

则可以用如下程序替换:

```
call mpi_send( a, 5, newtype, 1, 99, comm, ierr )
```

可以用这个函数构造双精度复数。



Back

Close



# 数据类型辅助函数

## MPI\_TYPE\_COMMIT

---

C	<code>int MPI_Type_commit( MPI_Datatype *datatype )</code>
Fortran	<code>MPI_TYPE_COMMIT( DATATYPE, IERROR )</code> <code>INTEGER DATATYPE, IERROR</code>

---

## MPI\_TYPE\_FREE

---

C	<code>int MPI_Type_free( MPI_Datatype *datatype )</code>
Fortran	<code>MPI_TYPE_FREE( DATATYPE, IERROR )</code> <code>INTEGER DATATYPE, IERROR</code>

---

## MPI\_TYPE\_EXTENT

---

C	<code>int MPI_Type_extent(MPI_Datatype datatype,MPI_Aint *extent)</code>
Fortran	<code>MPI_TYPE_EXTENT( DATATYPE, EXTENT, IERROR )</code> <code>INTEGER DATATYPE, EXTENT, IERROR</code>

---



Back

Close





## MPI\_ADDRESS

C	<code>int MPI_Address( void* location, MPI_Aint *address )</code>
Fortran	<code>MPI_ADDRESS( LOCATION, ADDRESS, IERROR )</code> <code>&lt;type&gt; LOCATION(*)</code> <code>INTEGER ADDRESS, IERROR</code>

这里需要注意的是MPI\_Aint，目前它是一个长整数，不正确定义变量，可能引起程序错误。



Back

Close



# VECTOR数据类型

如果将计算机中存放的下面标红的数据构造成一个新的数据类型，

$a_0$	$a_1$	$a_2$	$a_3$	$a_4$	$a_5$	$a_6$	$a_7$	$a_8$	$a_9$	$a_A$	$a_B$	$a_C$	...	...	$a_n$
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-----	-----	-------

怎样实现？

## MPI\_TYPE\_VECTOR

---

```
C      int MPI_Type_vector( int count, int blocklength,
                           int stride, MPI_Datatype oldtype,
                           MPI_Datatype *newtype )
```

---

```
Fortran  MPI_TYPE_VECTOR( COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,
                           NEWTYPE, IERROR )
          INTEGER COUNT,BLOCKLENGTH,STRIDE,OLDTYPE,NEWTYPE,IERROR
```

---



Back

Close



## MPI\_TYPE\_HVECTOR

---

```
C      int MPI_Type_hvector(int count, int blocklength,  
                           int stride, MPI_Datatype oldtype,  
                           MPI_Datatype *newtype )
```

---

```
Fortran  MPI_TYPE_HVECTOR( COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,  
                          NEWTYPE, IERROR )  
  
INTEGER COUNT,BLOCKLENGTH,STRIDE,OLDTYPE,NEWTYPE,IERROR
```

---

这2个函数的参数是一样的，只是在HVECTOR中的STRIDE是按照字节数为单位。这个数据类型可以用来定义一个新的传输矩阵的数据类，如果没有这个数据类型的支持，传输矩阵将带来许多麻烦。下面是一个矩阵 9.1数据类型的定义示例：

```
void typevector(m, n, lda, vector)  
  
int m, n, lda; MPI_Datatype *vector;  
  
{  
  
    MPI_Type_vector( m, n, lda, MPI_INT, vector );
```



Back

Close

```
}
```

此函数定义了一个传输 $m \times n$ 的矩阵数据类型，lda是矩阵的主维数的大小。如果考虑型如 9.1的两块矩阵定义成一个新类型，而这两块矩阵之间的距离没有办法在用矩阵数据类型的个数表示，这时就需要用HVECTOR。



180/248



Back

Close



# INDEX数据类型

对于如下红色的数据需要传输，

$a_0$	$a_1$	$a_2$	$a_3$	$a_4$	$a_5$	$a_6$	$a_7$	$a_8$	$a_9$	$a_A$	$a_B$	$a_C$	...	...	$a_n$
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-----	-----	-------

这时数据块的大小以及他们之间的距离都不是常数，不能再用VECTOR定义新的数据类型，这时需要如下函数。



Back

Close

## MPI\_TYPE\_INDEXED

---

```
C      int MPI_Type_indexed(int count,int *array_of_blocklengths,  
                           int *array_of_displacements,  
                           MPI_Datatype oldtype,  
                           MPI_Datatype *newtype )
```

---

```
Fortran  MPI_TYPE_INDEXED( COUNT, ARRAY_OF_BLOCKLENGTHS,  
                          ARRAY_OF_DISPLACEMENTS, OLDTYPE,  
                          NEWTYPE, IERROR )  
  
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),  
          ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR
```

---



182/248



Back

Close

## MPI\_TYPE\_HINDEXED

---

```
C      int MPI_Type_hindexed(int count,int *array_of_blocklengths,  
                             int *array_of_displacements,  
                             MPI_Datatype oldtype,  
                             MPI_Datatype *newtype )
```

---

```
Fortran  MPI_TYPE_HINDEXED( COUNT, ARRAY_OF_BLOCKLENGTHS,  
                             ARRAY_OF_DISPLACEMENTS, OLDTYPE,  
                             NEWTYPE, IERROR )  
  
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),  
          ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR
```

---

这2个函数的参数是一样的,只是ARRAY\_OF\_DISPLACEMENTS在HINDEX中是按照字节数为单位的。



183/248



Back

Close



# STRUCT数据类型

如果希望传输下面标红的几块数据，

$a_0$	$a_1$	$a_2$	$a_3$	$a_4$	...	...	$a_n$
-------	-------	-------	-------	-------	-----	-----	-------

$b_0$	$b_1$	$b_2$	$b_3$	$b_4$	...	...	$b_m$
-------	-------	-------	-------	-------	-----	-----	-------

$c_0$	$c_1$	$c_2$	$c_3$	$c_4$	...	...	$c_k$
-------	-------	-------	-------	-------	-----	-----	-------

前面介绍的构造数据类型的方法都不能完成此项任务。在MPI中，给出了构造此类需求的函数如下。



Back

Close





## MPI\_TYPE\_STRUCT

---

```
C      int MPI_Type_struct(int count,int *array_of_blocklengths,
                           int *array_of_displacements,
                           MPI_Datatype *array_of_types,
                           MPI_Datatype *newtype)
```

---

```
Fortran  MPI_TYPE_STRUCT( COUNT, ARRAY_OF_BLOCKLENGTHS,
                           ARRAY_OF_DISPLACEMENTS,
                           ARRAY_OF_TYPES, NEWTYPE, IERROR )

      INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
      ARRAY_OF_DISPLACEMENTS(*),
      ARRAY_OF_TYPES(*), NEWTYPE, IERROR
```

---

在自定义数据类型中，MPI\_TYPE\_STRUCT是一个万能函数，其它几个函数都可以通过MPI\_TYPE\_STRUCT 来产生。比如，

```
ARRAY_OF_BLOCKLENGTHS(i)  =BLOCKLENGTH
ARRAY_OF_DISPLACEMENTS(i) =i*STRIDE*SIZEOF(OLDTYPE)
ARRAY_OF_TYPES(i)         =OLDTYPE
```

则由MPI\_TYPE\_STRUCT构造的数据类型与MPI\_TYPE\_VECTOR是相同的。

此外，MPI\_TYPE\_STRUCT 主要是用于C程序中的结构类型数据传输。



Back

Close



# 特殊数据类型与绝对原点

为丰富自定义数据类型的使用，MPI提供了MPI\_LB和MPI\_UB2个数据类型，同时也提供了一个绝对原点MPI\_BOTTOM，以下用具体实例进行说明。在自定义数据类型中，矩阵应用是最广泛的，比如我们希望传输下面的红色矩阵块，其中矩阵元素是实型的

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} & a_{06} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \end{pmatrix}$$

我们可以用MPI\_Type\_vector定义一个由红色元素组成的小块矩阵的





187/248

数据类型`submat`，具体为：

```
MPI_Type_vector( 2, 3, 7, MPI_FLOAT, &submat )
```

如果在进程0和进程1中都是这样定义的矩阵数据类型，则可以将进程0 中红色小块数据发送给进程1，即在进程0中有：

```
MPI_Send( A, 1, submat, 1, 3, comm )
```

进程1中相应有：

```
MPI_Recv( A, 1, submat, 0, 3, comm, &st )
```

假如发送的不是1块，而是2块，那么送出去的是哪些数据？

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} & a_{06} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \end{pmatrix}$$



Back

Close



为什么会是如此，这是因为数据类型`submat`的延伸尺寸（`extent`）为40，下一个数据块就是从这里开始的。那如果我们希望传送的是对角块矩阵，就是下面的红兰块，怎么处理？

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} & a_{06} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \end{pmatrix} \quad (9.1)$$

其实也容易理解，只需要把`submat`的延伸尺寸放到这里的 $a_{23}$ 之前即可。正是因为有这样的需求，MPI 提供了二个0字节的数据类型，一个是`MPI_LB`，另一个是`MPI_UB`，此即为我们所称之的特殊数据类型。使用这二个数据类型构造的新类型不增加数据的真实尺寸，只是起到改变延伸尺寸的作用。



Back

Close



189/248

对于前述(9.1)的二块数据，我们可以对`submat`进行改造，从而用

```
MPI_Send( A, 2, submat2, 1, 3, comm )
```

实现传送所需数据的目的。如何能够做到？就是用数据类型`MPI_UB`和`submat`一起构成一个新的数据类型`submat2`，其中`submat`的位移是0，`MPI_UB`的位移是 $(2 \times 7 + 3) \times \text{sizeof}(\text{float})$ 。如此构造的数据类型`submat2`在传送1 块数据时，与数据类型`submat`是一致的。

在`MPI_Address`中返回的地址是相对于`MPI_BOTTOM`，如果数据类型定义中不使用相对位移，则可以使用`MPI_BOTTOM`作为发送和接收数据的首地址，此即为`MPI_BOTTOM`的作用。



Back

Close



# MPI的数据打包与拆包

## MPI\_PACK

---

```
C      int MPI_Pack(void* inbuf,int incount,MPI_Datatype datatype,
                  void* outbuf, int outsize, int *position,
                  MPI_Comm comm )
```

---

```
Fortran  MPI_PACK( INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
                  POSITION, COMM, IERROR )
          <type> INBUF(*), OUTBUF(*)
          INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
```

---

[Back](#)[Close](#)



## MPI\_UNPACK

---

```
C      int MPI_Unpack( void* inbuf, int insize, int *position,
                      void* outbuf, int outcount,
                      MPI_Datatype datatype, MPI_Comm comm )
```

---

```
Fortran MPI_UNPACK( INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
                   DATATYPE, COMM, IERROR )

<type> INBUF(*), OUTBUF(*)

INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
```

---

## MPI\_PACK\_SIZE

---

```
C      int MPI_Pack_size( int incount, MPI_Datatype datatype,
                        MPI_Comm comm, int *size )
```

---

```
Fortran MPI_PACK_SIZE( INCOUNT, DATATYPE, COMM, SIZE, IERROR )

INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

---

[Back](#)[Close](#)



# 习题

## 1. 分块矩阵

$$A = \begin{pmatrix} A_{00} & A_{01} & \dots & \dots \\ A_{10} & A_{11} & \dots & \dots \\ A_{20} & A_{21} & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} \quad (9.2)$$

其中  $A_{ij}$  是  $m \times m$  阶矩阵,

- (a) 请写出构造一个数据类型的子程序, 使得新数据类型可以发送和接收小块矩阵  $A_{00}$ , 并且也可以一次性发送和接收  $A_{00}$  和  $A_{20}$ ;
- (b) 如果只是发送  $A_{00}$  和  $A_{20}$ , 是否还有其它构造数据类型的方法? 如果有怎么构造?



Back

Close





193/248

2. 设结构`{int m[3]; float a[2]; char c[5];}`定义的数组为`x[10]`, 如果将进程0长度为10的数组`x`发送给进程1, 请写一个程序予以验证。
3. 假设分块下三角矩阵的对角块矩阵都是 $m \times m$ 阶的下三角矩阵, 构造一个下三角矩阵的数据类型, 使得这个数据类型可以用于传送这个矩阵的连续多个对角块矩阵。



Back

Close



194/248

## 十、MPI聚合通信

这部分包括MPI\_Barrier, MPI\_Bcast, MPI\_Gather, MPI\_Scatter, MPI\_Alltoall。通讯的特点是通讯子内所有进程均参与通讯,是MPI 为用户提供方便而研制的函数,从而可以提高编程效率。



Back

Close



# 障碍同步MPI\_Barrier

## MPI\_BARRIER

C	int MPI_Barrier( MPI_Comm comm )
Fortran	MPI_BARRIER( COMM, IERROR ) INTEGER COMM, IERROR

所有进程都执行这个函数后，立即返回，不然就等待。

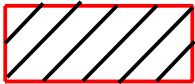
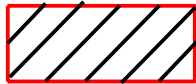
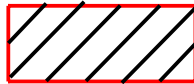
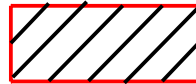


Back

Close



# 广播MPI\_Bcast

 $P_0$  $P_0$  $P_1$  $P_2$  $P_3$ 

## MPI\_BCAST

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

```
Fortran MPI_BCAST( BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR )  
          <type> BUFFER(*)  
          INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

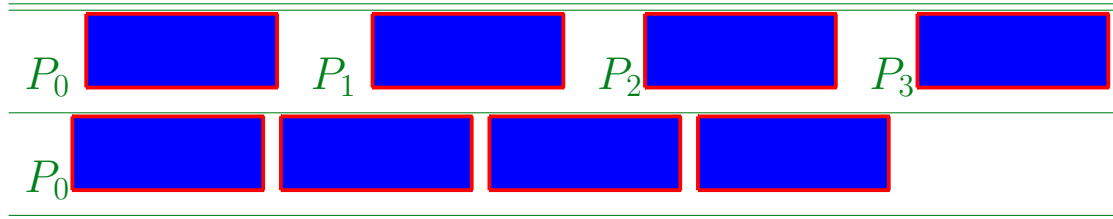


Back

Close



# 收集MPI\_Gather



## MPI\_GATHER

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

```
Fortran  MPI_GATHER( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,  
                   RECVCOUNT, RECVTYPER, ROOT, COMM, IERROR )  
  
<type> SENDBUF(*), RECVBUF(*)  
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPER, ROOT,  
        COMM, IERROR
```



Back

Close

## MPI\_ALLGATHER

---

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

---

```
Fortran  MPI_ALLGATHER( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,  
                    RECVCOUNT, RECVTYPE, COMM, IERROR)  
  
<type> SENDBUF(*), RECVBUF(*)  
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE,  
        COMM, IERROR
```

---



198/248



Back

Close



199/248

## MPI\_GATHERV

---

```
C      int MPI_Gatherv( void* sendbuf, int sendcount,
                        MPI_Datatype sendtype, void* recvbuf,
                        int *recvcounts, int *displs,
                        MPI_Datatype recvttype, int root,
                        MPI_Comm comm )
```

---

```
Fortran  MPI_GATHERV( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                     RECVCOUNTS, DISPLS, RECVTTYPE, ROOT, COMM,
                     IERROR )

<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*),
                     RECVTTYPE, ROOT, COMM, IERROR
```

---



Back

Close

## MPI\_ALLGATHERV

---

```
C      int MPI_Allgatherv( void* sendbuf, int sendcount,  
                          MPI_Datatype sendtype, void* recvbuf,  
                          int *recvcounts, int *displs,  
                          MPI_Datatype recvtype, MPI_Comm comm)
```

---

```
Fortran  MPI_ALLGATHERV( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,  
                      RECVCOUNTS, DISPLS, RECVTYPE, COMM, IERROR)
```

```
<type> SENDBUF(*), RECVBUF(*)  
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*),  
      RECVTYPE, COMM, IERROR
```

---



200/248



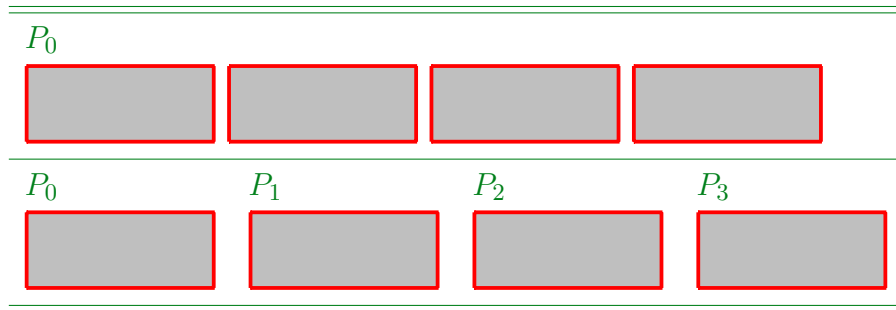
Back

Close





# 散播MPI\_Scatter



Back

Close



202/248

## MPI\_SCATTER

---

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

---

```
Fortran  MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE,
                    RECVBUF, RECVCOUNT, RECVTYPE, ROOT,
                    COMM, IERROR )

<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, REVCOUNT,
                    RECVTYPE, ROOT, COMM, IERROR
```

---



Back

Close

## MPI\_SCATTERV

---

```
C      int MPI_Scatterv(void* sendbuf, int *sendcounts,
                        int *displs, MPI_Datatype sendtype,
                        void* recvbuf, int recvcoun,
                        MPI_Datatype recvtype, int root,
                        MPI_Comm comm )
```

---

```
Fortran  MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE,
                      RECVBUF, RECVCOUNT, RECVTYPE, ROOT,
                      COMM, IERROR )

<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT,
                      RECVTYPE, ROOT, COMM, IERROR
```

---



203/248

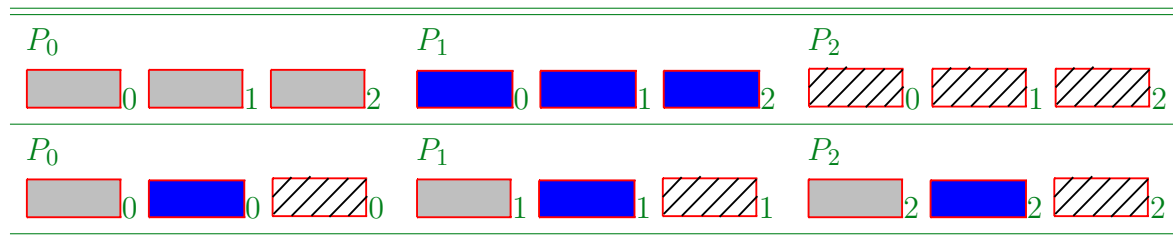


Back

Close



# 全交换MPI\_Alltoall



$$\begin{array}{c|ccc} P_0 & A_0 & A_1 & A_2 & \Rightarrow & A_0 & B_0 & C_0 \\ P_1 & B_0 & B_1 & B_2 & & A_1 & B_1 & C_1 \\ P_2 & C_0 & C_1 & C_2 & & A_2 & B_2 & C_2 \end{array}$$



Back

Close

## MPI\_ALLTOALL

---

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

---

```
Fortran  MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE,
                      RECVBUF, RECVCOUNT, RECVTYPE,
                      COMM, IERROR )

<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT,
                      RECVTYPE, COMM, IERROR
```

---



205/248



Back

Close



206/248

## MPI\_ALLTOALLV

---

```
C      int MPI_Alltoallv(void* sendbuf, int *sendcounts,
                        int *sdispls, MPI_Datatype sendtype,
                        void* recvbuf, int *recvcounts,
                        int *rdispls, MPI_Datatype recvtype,
                        MPI_Comm comm )
```

---

```
Fortran  MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
                      RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE,
                      COMM, IERROR )

<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*),
                      RDISPLS(*), RECVTYPE, COMM, IERROR
```

---

### 算法 15 *all2all*

1. 假设有 $p$ 个进程，第 $i$ 个进程上的数据划分为： $A_{i0} \ A_{i1} \ \cdots \ A_{i,p-1}$ ，最后第 $i$ 个进程上的结果是： $A_{0i} \ A_{1i} \ \cdots \ A_{p-1,i}$ ；
2. 记 $f = iam$ ,  $n = iam$ ,  $B_{iam} = A_{iam}$ ;



Back

Close

3.  $n = (n + 1) \bmod p$ ,  $f = (f + p - 1) \bmod p$ , 发送  $A_n$  给进程  $n$ ,  
从  $f$  进程接收  $B_f$



207/248



Back

Close



## 习题

1. 用MPI\_SEND和MPI\_RECV实现MPI\_ALLTOALL的并行数据传输方法与程序;
2. 假设数组 $x$ 的长度 $n = mp$ , 证明:  
 $\text{MPI\_Bcast} = \text{MPI\_Scatter} + \text{MPI\_Allgather}$



Back

Close





209/248

# 十一、MPI归约操作

主要函数有：MPI\_Reduce, MPI\_Scan, MPI\_Reduce\_scatter。所有进程都参与，亦即在通讯子内的各个进程按照运算要求进行操作，得到一个结果。使用者可以不用考虑这方面的并行实现，直接调用这些函数即可。

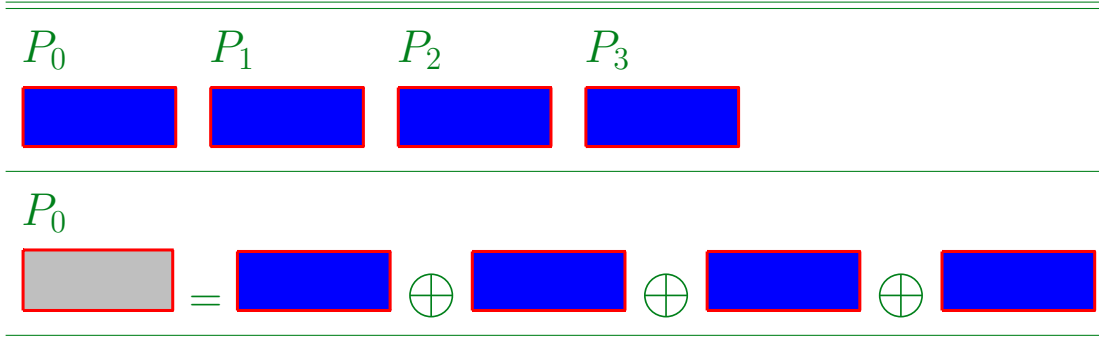


Back

Close



# 归约MPI\_Reduce



## MPI\_REDUCE

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

```
Fortran  MPI_REDUCE( SENDBUF, RECVBUF, COUNT, DATATYPE,  
                   OP, ROOT, COMM, IERROR )  
  
         <type> SENDBUF(*), RECVBUF(*)  
         INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```



Back

Close

## MPI\_ALLREDUCE

---

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

---

```
Fortran  MPI_ALLREDUCE( SENDBUF, RECVBUF, COUNT, DATATYPE,  
                    OP, COMM, IERROR )  
  
          <type> SENDBUF(*), RECVBUF(*)  
          INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

---



211/248



Back

Close



表 11.1 运算种类

操作名	意义	操作名	意义
MPI_MAX	求最大	MPI_LOR	逻辑或
MPI_MIN	求最小	MPI_BOR	按位或
MPI_SUM	求和	MPI_LXOR	逻辑与或
MPI_PROD	求积	MPI_BXOR	按位与或
MPI_LAND	逻辑与	MPI_MAXLOC	求最大和位置
MPI_BAND	按位与	MPI_MINLOC	求最小和位置

## 运算种类与可用数据类型

常用的运算是MPI\_MAX、MPI\_MIN和MPI\_SUM，有时也可能用到



Back

Close

MPI\_MAXLOC和MPI\_MINLOC。



操作名	允许的数据类型
MPI_MAX, MPI_MIN	integer, Floating point
MPI_SUM, MPI_PROD	integer, Floating point, Complex
MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical
MPI_BAND, MPI_BOR, MPI_BXOR	integer, Byte

复合数据类型

对于MPI\_MAXLOC和MPI\_MINLOC两种运算, MPI对Fortran程序和C程序使用的复合数据类型规定如下:



214/248

## Fortran 程序

复合数据类型	类型描述
MPI_2REAL	pair of REALs
MPI_2DOUBLE_PRECISION	pair of DOUBLE PRECISIONs
MPI_2INTEGER	pair of INTEGERs

## C 程序

复合数据类型	类型描述
MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int
MPI_LONG_INT	long and int
MPI_SHORT_INT	short and int
MPI_LONG_DOUBLE_INT	long double and int
MPI_2INT	pair of ints

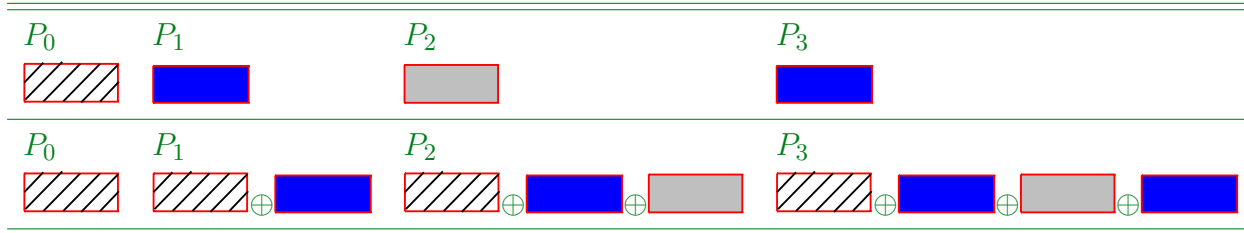


Back

Close



# 前缀MPI\_Scan



## MPI\_SCAN

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

```
Fortran  MPI_SCAN( SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM,
                  IERROR )

         <type> SENDBUF(*), RECVBUF(*)
         INTEGER COUNT(*), DATATYPE, OP, COMM, IERROR
```

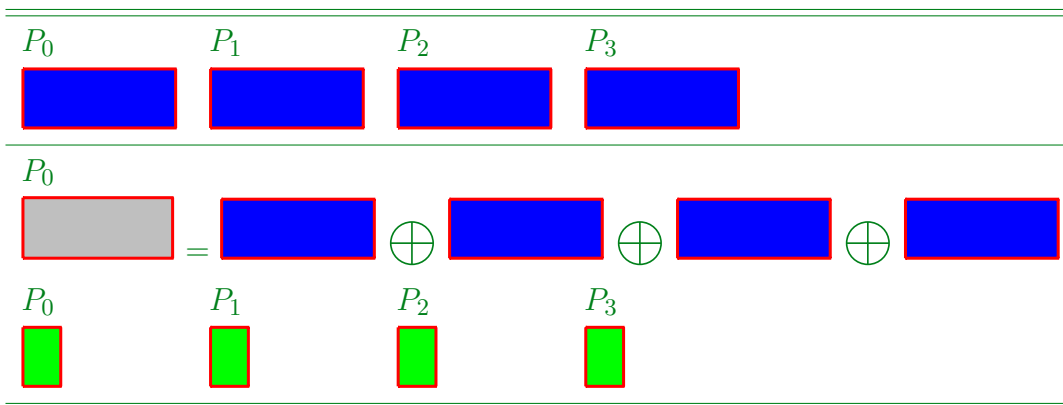


Back

Close



# 归约散播MPI\_Reduce\_scatter



Back

Close



## MPI\_REDUCE\_SCATTER

---

```
C      int MPI_Reduce_scatter(void* sendbuf, void* recvbuf,  
                             int *recvcounts,  
                             MPI_Datatype datatype, MPI_Op op,  
                             MPI_Comm comm )
```

---

```
Fortran  MPI_REDUCE_SCATTER( SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE,  
                             OP, COMM, IERROR )  
  
<type> SENDBUF(*), RECVBUF(*)  
INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
```

---



217/248



Back

Close



# 自定义运算

对于归约操作函数中的运算，用户根据自己的需要，可以自行定义满足要求的算子。为达到此目的，需要借助以下的一些函数来完成。

## MPI\_OP\_CREATE

---

```
C      int MPI_Op_create( MPI_User_fuction *function,
                        int commute, MPI_Op *op )
```

---

```
Fortran  MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR )
          EXTERNAL FUNCTION
          LOGICAL COMMUTE
          INTEGER OP, IERROR
```

---

## MPI\_OP\_FREE

---

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

---

```
Fortran  MPI_OP_FREE( OP, IERROR )
          INTEGER OP, IERROR
```

---



Back

Close



用户自定义函数是有严格要求的，具有如下形式：

使用C语言时，用户自定义函数为：

```
typedef void MPI_User_function( void *invec, void *inoutvec,  
                                int *len, MPI_Datatype *datatype);
```

使用FORTRAN语言时，其函数定义为：

```
FUNCTION USER_FUNCTION( INVEC(*), INOUTVEC(*), LEN, TYPE)
```



Back

Close



220/248

## 十二、MPI组操作

组操作类似于数学中的集合运算，主要目的是构建不同用途的通讯子，这些通讯子如果用`Comm_split`将难以实现，因此MPI为编程方便，提供了构建任意通讯子的方法。



Back

Close



# 进程组的创建

## MPI\_COMM\_GROUP

---

C	int MPI_Comm_group( MPI_Comm comm, MPI_Group *group )
---	---

---

Fortran	MPI_COMM_GROUP( COMM, GROUP, IERROR )
	INTEGER COMM, GROUP, IERROR

---

## MPI\_GROUP\_UNION

---

C	int MPI_Group_union( MPI_Group group1, MPI_Group group2, MPI_Group *group )
---	--

---

Fortran	MPI_GROUP_UNION( GROUP1, GROUP2, GROUP, IERROR )
	INTEGER GROUP1, GROUP2, GROUP, IERROR

---



Back

Close



222/248

## MPI\_GROUP\_INTERSECTION

---

```
C      int MPI_Group_intersection( MPI_Group group1,  
                                MPI_Group group2, MPI_Group *group )
```

---

```
Fortran MPI_GROUP_INTERSECTION( GROUP1, GROUP2, GROUP, IERROR )  
        INTEGER GROUP1, GROUP2, GROUP, IERROR
```

---

## MPI\_GROUP\_DIFFERENCE

---

```
C      int MPI_Group_difference( MPI_Group group1,  
                                MPI_Group group2, MPI_Group *group )
```

---

```
Fortran MPI_GROUP_DIFFERENCE( GROUP1, GROUP2, GROUP, IERROR )  
        INTEGER GROUP1, GROUP2, GROUP, IERROR
```

---

## MPI\_GROUP\_INCL

---

```
C      int MPI_Group_incl( MPI_Group group1, int n, int *ranks,  
                           MPI_Group *group )
```

---

```
Fortran MPI_GROUP_INCL( GROUP1, N, RANKS, GROUP, IERROR )  
        INTEGER GROUP1, N, RANKS(*), GROUP, IERROR
```

---



Back

Close



## MPI\_GROUP\_EXCL

---

```
C      int MPI_Group_excl( MPI_Group group1, int n, int *ranks,  
                          MPI_Group *group )
```

---

```
Fortran  MPI_GROUP_EXCL( GROUP1, N, RANKS, GROUP, IERROR )  
          INTEGER GROUP1, N, RANKS(*), GROUP, IERROR
```

---

## MPI\_GROUP\_RANGE\_INCL

---

```
C      int MPI_Group_range_incl( MPI_Group group1, int n,  
                                int ranges[] [3], MPI_Group *group )
```

---

```
Fortran  MPI_GROUP_RANGE_INCL( GROUP1, N, RANGES, GROUP, IERROR )  
          INTEGER GROUP1, N, RANGES(3, *), GROUP, IERROR
```

---

## MPI\_GROUP\_RANGE\_EXCL

---

```
C      int MPI_Group_range_excl( MPI_Group group1, int n,  
                                int ranges[] [3], MPI_Group *group )
```

---

```
Fortran  MPI_GROUP_RANGE_EXCL( GROUP1, N, RANGES, GROUP, IERROR )  
          INTEGER GROUP1, N, RANGES(3, *), GROUP, IERROR
```

---



Back

Close



# 进程组管理

## MPI\_GROUP\_SIZE

C	<code>int MPI_Group_size( MPI_Group group, int *size )</code>
Fortran	<code>MPI_GROUP_SIZE( GROUP, SIZE, IERROR )</code> <code>INTEGER GROUP, SIZE, IERROR</code>

## MPI\_GROUP\_RANK

C	<code>int MPI_Group_rank( MPI_Group group, int *rank )</code>
Fortran	<code>MPI_GROUP_RANK( GROUP, RANK, IERROR )</code> <code>INTEGER GROUP, RANK, IERROR</code>



Back

Close





## MPI\_GROUP\_TRANSLATE\_RANKS

---

```
C      int MPI_Group_translate_ranks( MPI_Group group1, int n,  
                                     int *ranks1, MPI_Group group2, int *ranks2 )
```

---

```
Fortran  MPI_GROUP_TRANSLATE_RANKS( GROUP1, N, RANKS1, GROUP2,  
                                     RANKS2, IERROR )  
  
INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
```

---

## MPI\_COMM\_CREATE

---

```
C      int MPI_Comm_create( MPI_Comm comm, MPI_Group group,  
                           MPI_comm *newcomm )
```

---

```
Fortran  MPI_COMM_CREATE( COMM, GROUP, NEWCOMM, IERROR )  
  
INTEGER COMM, GROUP, NEWCOMM, IERROR
```

---

## MPI\_GROUP\_FREE

---

```
C      int MPI_Group_free( MPI_Group group )
```

---

```
Fortran  MPI_GROUP_FREE( GROUP, IERROR )  
  
INTEGER GROUP, IERROR
```

---



Back

Close



226/248

## MPI\_GROUP\_COMPARE

---

```
C      int MPI_Group_compare( MPI_Group group1,  
                             MPI_Group group2, int *result )
```

---

```
Fortran  MPI_GROUP_COMPARE( GROUP1, GROUP2, RESULT, IERROR )
```

---

```
INTEGER GROUP1, GROUP2, RESULT, IERROR
```

---

RESULT的值为: MPI\_IDENT, MPI\_SIMILAR, MPI\_UNEQUAL。



Back

Close

# 十三、并行程序实例

13.1、 $\pi$ 值近似计算程序

13.2、Cannon算法实现程序

13.3、数据广播并行程序

13.4、数据分散并行程序



227/248



Back

Close



## $\pi$ 值近似计算程序

例子 8  $\pi$ 值近似计算:

由三角函数和定积分公式可知:

$$\frac{\pi}{4} = \arctan(1) = \int_0^1 \frac{1}{1+x^2} dx \quad (13.1)$$

假设将区间 $[0, 1]$ 分成 $n$ 等份, 记 $h = 1/n$ ,  $x_i = ih$ ,  $i = 0, 1, \dots, n$ , 则采用梯形积分公式计算积分(13.1)如下:

$$\int_0^1 \frac{1}{1+x^2} dx = \sum_{i=0}^{n-1} \left[ \frac{h}{2} \left( \frac{1}{1+x_i^2} + \frac{1}{1+x_{i+1}^2} \right) - \frac{h^3}{12} \left( \frac{1}{1+\xi_i^2} \right)'' \right] \quad (13.2)$$



Back

Close



229/248

对于给定的精度 $\varepsilon$ ，可以确定一个 $N = \lceil \sqrt{4/(3\varepsilon)} \rceil$ ，使得当 $n \geq N$ 时，有：

$$\left| \int_0^1 \frac{4}{1+x^2} dx - \sum_{i=0}^{n-1} \left[ \frac{h}{2} \left( \frac{4}{1+x_i^2} + \frac{4}{1+x_{i+1}^2} \right) \right] \right| < \varepsilon \quad (13.3)$$

可以将

$$\sum_{i=0}^{n-1} \left[ \frac{h}{2} \left( \frac{4}{1+x_i^2} + \frac{4}{1+x_{i+1}^2} \right) \right] \quad (13.4)$$

作为计算 $\pi$ 的近似值。

```
program computing_pi
```

```
*The header file for using MPI parallel environment,  
* which must be included for all mpi programs.
```

```
include 'mpif.h'
```



Back

Close



230/248

\*Variables declaration

```
integer iam, np, comm, ierr  
integer n, i, num, is, ie  
real*8 pi, h, eps, xi, s
```

\*Enroll in MPI environment and get the MPI parameters

```
call mpi_init(ierr)  
call mpi_comm_dup(mpi_comm_world, comm, ierr)  
call mpi_comm_rank(comm, iam, ierr)  
call mpi_comm_size(comm, np, ierr)
```

\*Read the number of digits you want for value of Pi.

```
if(iam .eq. 0) then
```



Back

Close



231/248

```
        write(*, *) 'Number of digits(1-16)= '  
        read(*, *) num  
endif  
call mpi_bcast(num,1,mpi_integer,0,comm,ierr)  
  
eps = 1  
do 10 i=1, num  
    eps = eps * 0.1  
10 continue  
  
n = sqrt(4.0/(3.0*eps))  
h = 1.0/n  
num = n/np
```



Back

Close



232/248

```
if(iam .eq. 0) then
    s = 3.0
    xi = 0
    is = 0
    ie = num
elseif(iam .eq. np-1) then
    s = 0.0
    is = iam*num
    ie = n - 1
    xi = is * h
else
    s = 0.0
    is = iam*num
```



Back

Close





233/248

```
    ie = is + num  
    xi = is * h  
endif
```

```
if(np .eq. 1) ie = ie - 1  
do 20 i=is+1, ie
```

```
    xi = xi + h  
    s = s + 4.0/(1.0+xi*xi)
```

```
20  continue  
    call mpi_reduce(s, pi, 1, mpi_double_precision,  
&                  mpi_sum, 0, comm, ierr)  
    if(iam .eq. 0) then  
        pi = h*pi
```



Back

Close



234/248

```
        write(*, 99) pi
    endif
    call mpi_finalize(ierr)
99  format('The pi= ', f16.13)
    end
```



Back

Close



# Cannon算法实现程序

这里我们给出Cannon算法4.7的实现过程, 从中可以体会一下我们的所学。

```
#include "myhead.h"

/*Cannon algorithm, processors are p*p*/
void cannon(m, k, n, a, lda, b, ldb, c, ldc, \
            rcom, ccom, work)
int m, k, n, lda, ldb, ldc;
float *a, *b, *c, *work;
MPI_Comm rcom, ccom;
```



Back

Close

```
{
```

```
int p, rid, cid, front, next, i, j, l, kb, \  
    kk, root;
```

```
MPI_Datatype smatrb1;
```

```
MPI_Status st;
```

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

```
MPI_Comm_rank( rcom, &cid );
```

```
MPI_Comm_rank( ccom, &rid );
```

```
front = (rid-1+p)%p;
```

```
next = (rid+1)%p;
```

```
for(i=0; i<m; i++)
```



236/248



Back

Close



237/248

```
for(j=0; j<n; j++)
    c[i*ldc+j] = 0.0;

l= 0;
kb = k;
for(i=0; i<p; i++) {
    root = (rid+l) % p;
    if( i != 0 ) {
        MPI_Sendrecv(&kb, 1, MPI_INT, front, 1, \
                     &kk, 1, MPI_INT, next, 1, ccom,\
                     &st);

        MPI_Type_vector(kb, n, ldb, MPI_FLOAT, \
                        &smatrb1);

        kb = kk;
```



Back

Close



238/248

```
    MPI_Type_commit( &smatrb1 );
    MPI_Sendrecv(b, 1, smatrb1, front, 1, work, kb*n, \
                 MPI_FLOAT, next, 1, ccom, &st);
    MPI_Type_free( &smatrb1 );
    scopym(kb, n, work, n, b, ldb);
}

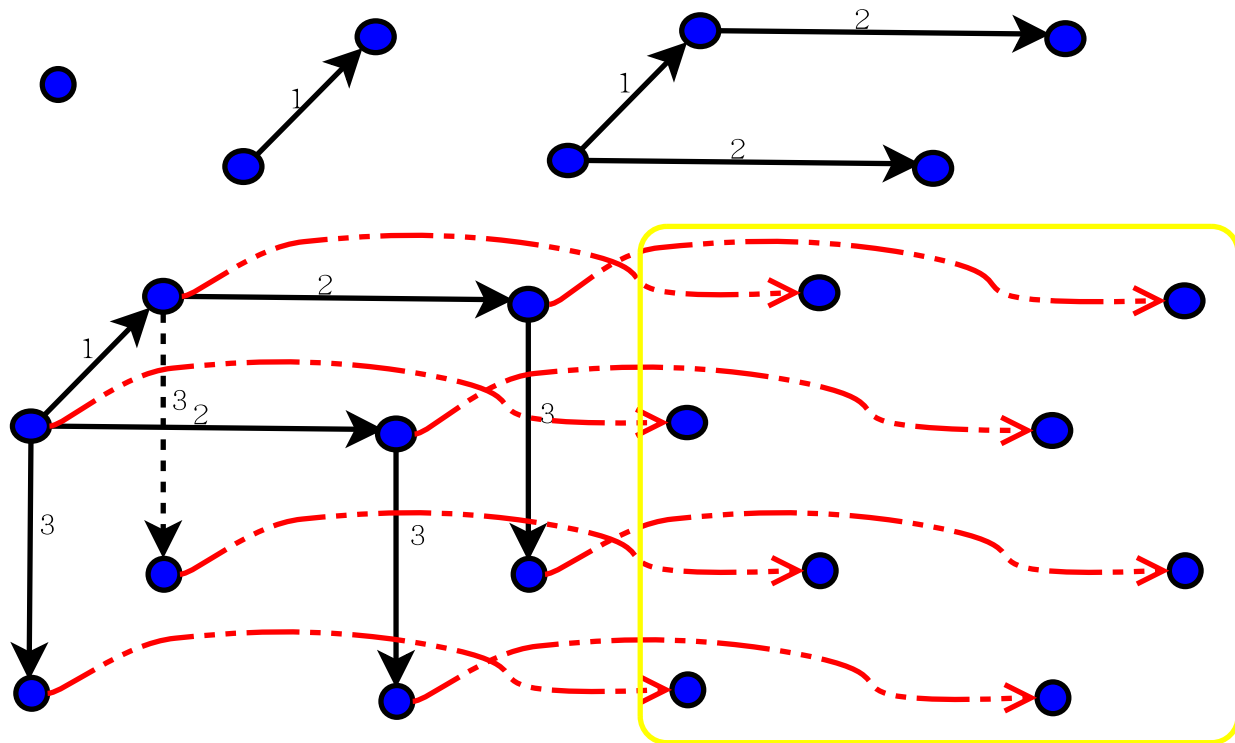
if(cid == root)
    scopym(m, kb, a, lda, work, kb);
MPI_Bcast(work, m*kb, MPI_FLOAT, root, rcom);
sgemm(m, kb, n, work, kb, b, ldb, c, ldc);
l++;
}
}
```



Back

Close

# 数据广播并程序序



239/248



## 例子 9 数据广播

在使用 $q$ 个处理机的系统上，对数据进行广播，则其MPI程序如何实现？

```
subroutine mpibcastr( b, n, root, comm, iam, np )
include 'mpif.h'
integer n, root, comm, iam, np
real b(*)

integer ierr, newid, i, des, src, left,
&      status(mpi_status_size), mlen, iter

newid = mod( np+iam-root, np )
iter = alog(real(np))/alog(2.0)+1.0e-16
mlen = 1
```



240/248



Back

Close





241/248

```
do 20 i=1, iter
    des = mod( iam + mlen, np )
    src = mod( np + iam - mlen, np )
    if( newid .lt. mlen) then
        call mpi_send( b, n, mpi_real, des, 1, comm,
&                        ierr )
    elseif( newid .lt. 2*mlen ) then
        call mpi_recv( b, n, mpi_real, src, 1, comm,
&                        status, ierr )
    endif
    mlen = 2*mlen
20 continue
left = np - mlen
```



Back

Close



242/248

```
if ( left .le. 0 ) return
des = mod( iam + mlen, np )
src = mod( np + iam - mlen, np )
if( newid .lt. left) then
    call mpi_send( b, n, mpi_real, des, 1, comm,
&                  ierr )
elseif( newid .ge. mlen .and.
&        newid .lt. mlen+left ) then
    call mpi_recv( b, n, mpi_real, src, 1, comm,
&                 status, ierr )
endif

return
```



Back

Close

end



243/248

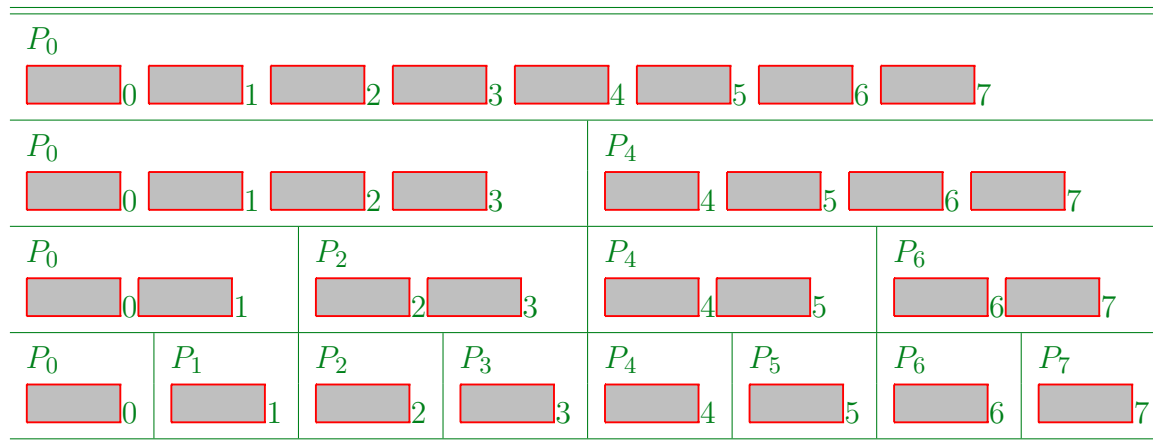


Back

Close



# 数据分散并程序序



## 例子 10 数据分散

这里我们考虑如何实现MPI\_Scatter，类似于13.3节关于广播函数的实现，在此也可以看出MPI\_Comm\_dup以及MPI\_Comm\_split的作用。



Back

Close



245/248

```
void scatter( comm, blk, a )  
MPI_Comm comm; int blk; float *a;  
{  
    int iam, np;  
    MPI_Comm tcom, scom;  
    MPI_Status st;  
    int len, color, key;  
  
    MPI_Comm_size( comm, &np );  
    MPI_Comm_rank( comm, &iam );  
    MPI_Comm_dup( comm, &tcom );  
  
    while ( np > 1 ) {
```



Back

Close



246/248

```
len = np / 2;
if( iam == 0 ) MPI_Send( &a[len*blk], \
    (np-len)*blk, MPI_FLOAT, len, 1, tcom );
if( iam == len ) MPI_Recv( &a[0], \
    (np-len)*blk, MPI_FLOAT, 0, 1, tcom, &st );
if ( iam < len ) color = 0;
else color = 1;
key = iam;
MPI_Comm_split( tcom, color, key, &scom );

MPI_Comm_dup( scom, &tcom );
MPI_Comm_size( tcom, &np );
MPI_Comm_rank( tcom, &iam );
```



Back

Close

```
}  
  
MPI_Comm_free( &tcom );  
  
MPI_Comm_free( &scom );  
  
return;  
  
}
```



247/248



Back

Close



248/248

# 参考文献

- [1] 迟学斌、王彦桐、王珏、刘芳,《并行计算与实现技术》,科学出版社, 2015
- [2] 张林波、迟学斌、莫则尧、李若,《并行计算导论》,清华大学出版社, 2006
- [3] 陈国良,《并行计算——结构●算法●编程》,高等教育出版社, 2003
- [4] 莫则尧、袁国兴,《消息传递并行编程环境MPI》,科学出版社, 2001
- [5] Brian W. Kernighan, Dennis M. Ritchie,《The C Programming Language》, Prentice-Hall, 1988
- [6] **基本要求:** 熟悉Fortran或C程序设计语言,了解数值计算方法



Back

Close