中国科学院研究生院2008-2009学年课程



# 高性能并行计算

迟学斌(chi@sccas.cn, www.sccas.cn)
中国科学院计算机网络信息中心

2009年2-6月




# 目　录












# 一、 绪论

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、中科院可能需要千万亿次计算的应用研究

1.6、国际上千万亿次计算的应用问题



# 什么是并行计算



并行计算（parallel computing）是指，在并行机上，将一个应用分解成多个子任务，分配给不同的处理器，各个处理器之间相互协同，并行地执行子任务，从而达到加速求解速度，或者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，

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

$$S = \sum_{i=0}^{n-1} a_i \tag{1.1}$$

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$S = S_{00} + S_{01}$



## 例子 2 流水线并行

汽车生产线，自助餐等。



# 并行计算机的发展

并行计算机从70年代的开始，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。



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（参见 2.3）
2. DSM, SMP（参见 2.2）
3. MPP（参见 2.1）
4. 星群

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。下面是国际上2008年TOP500计






算机系统的前10名情况：

| 计算机系统 | 国家 | 年份 | 核数 | 性能 |
|---|---|---|---|---|
| BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2Ghz/Opteron DC 1.8GHz, Voltaire Infiniband | USA | 2008 | 129600 | 1105000 |
| Cray XT5 QC 2.3 GHz | USA | 2008 | 150152 | 1059000 |
| SGI Altix ICE 8200EX, Xeon QC 3.0/2.66 GHz | USA | 2008 | 51200 | 487005 |
| 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 |
| Cray XT4 QuadCore 2.3 GHz | USA | 2008 | 38642 | 266300 |
| Cray XT4 QuadCore 2.1 GHz | USA | 2008 | 30976 | 205000 |
| Sandia/Cray Red Storm, XT 3/4, 2.4/2.2GHz 2/4 cores | USA | 2008 | 38208 | 204200 |
| Dawning 5000A, QC Opteron 1.9 Ghz, IB, Win HPC 2008 | China | 2008 | 30720 | 180600 |



# 为什么需要并行计算

全球气象预报中期天气预报模式要求在24小时内完成48小时天气预测数值模拟，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。

HPCC计划提出的背景是一大批巨大挑战性问题需要解决，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。



# 中科院高性能计算环境

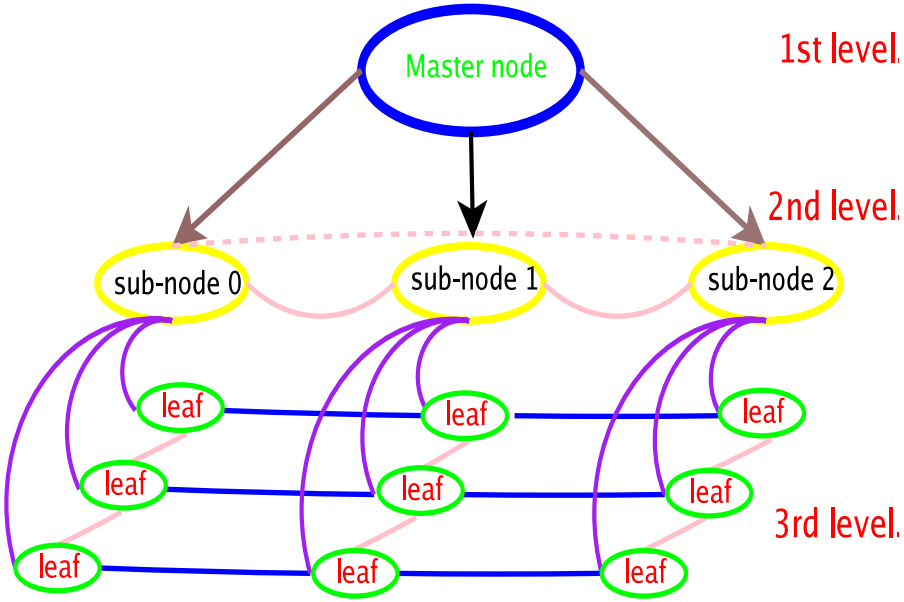



图 1.1 三层网格系统结构

## 总中心百万亿次计算机系统

深腾7000系统是混合结构高性能机群系统，其峰值性能142TFLOPS，实测LINPACK性能100TFLOPS，TOP500位列19位（2008年11月），主要配置如下：

- 胖结点：2台SGI Altix 4700，768个Itanium II(1.67GHz)处理器核
- 厚结点：38台IBM 3950 M2，2,432 个Xeon(2.93GHz)处理器核
- 刀片结点：1140台IBM刀片服务器，9,120个Xeon(3.0GHz)处理器核
- 可视化结点：12台IBM服务器，96个Xeon(3.0GHz)处理器核
- IO结点、登录结点、启动结点...
- 内存50TB，磁盘350TB，虚拟带库60TB，智能磁带库1000TB






## 分中心计算环境

每个分中心计算能力≥10TFLOPS，分布如下：

- 上海分中心：上海生命科学研究院
- 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
- 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
- 深圳分中心：深圳先进技术研究院
- 沈阳分中心：沈阳计算技术研究所有限公司
- 北京分中心：物理研究所、数学与系统科学研究院等



# 中科院可能需要千万亿次计算的应用研究



1. 环境科学

- 曾庆存、王斌、延晓冬等，大气所，“气候学理论研究、模式发展及模式应用：IAP-DCSM,LASG,RIEMS等”
- 周广庆等，王凡等，大气所、海洋所，“海洋模式发展及应用”
- 于强、莫兴国、田静、陶波等，地理科学与资源研究所，“中国生态系统研究网络：分布式生态水文模拟和数据同化、高分辨率陆地表面水热过程、土壤－植物－大气系统的水热过程和作物生长模型”
- 石耀霖，张怀等，研究生院计算地球实验室，“地球动力学定量化

模拟研究：区域应力演化历史、区域和全球地震波数值模拟”

2. 生命科学

- 陈润生，生物物理所，“理论生物学和生物信息学”
- 韩敬东，遗传所，“分子系统生物学”
- 于坤千，上海药物所，“大规模药物筛选、生物大分子体系的分子动力学模拟”
- 于军，北京基因组所，“基因组学与生物信息”
- 朱维良，上海药物所，“药物发现与设计”
- 吴佳妍，北京基因组所，“转录组信息挖掘”
- 熊兵，上海药物所，“计算机辅助药物设计及药物化学合成；生物大分子的构象变化，包括蛋白质折叠；蛋白－蛋白相互作用；药物－大分子相互作用；蛋白质构象变化疾病”






- 雷红星，北京基因组所，“蛋白质动力学模拟”

3. 化工、材料、纳米科学

- 葛蔚，过程所，“化工复杂过程的多尺度模拟分析与控制”
- 韩克利，大连化物所，“分子反应动力学实验与理论研究”
- 杨小震，孔滨等，化学所，“高分子化学与物理实验室”
- 史庭云，武汉物数所，“原子分子物理：量子少体结构与碰撞动力学，原子、分子的强外场效应”
- 杨明晖，武汉物数所，“分子光谱,反应动力学和量子化学”
- 杨锐、成会明、徐东生等，金属所，“材料设计/纳米尺度工程结构性质”
- 杜世萱、方忠等，物理所，“超高密度信息存储材料与技术/分子电子器件设计”






- 张文清，上海硅酸盐所，“多尺度材料模拟程序的发展及应用”

4. 高能物理与等离子体物理、流体

- 陈莹，高能所，“格点规范理论和格点QCD数值模拟研究”
- 马建平，理论物理所，“强相互作用的研究”
- 李建刚，等离子体物理研究所，“高温等离子体与受控热核聚变领域”
- 李新亮、傅德熏等，力学所，“高速飞行器复杂流动的直接数值模拟”

5. 空间天气、天文

- 冯学尚，空间中心，“太阳风暴传播”
- 王赤等，空间中心，”地球磁层空间天气模拟”






Back
Close

- 景益鹏，上海天文台，“宇宙大尺度结构的数值模拟和宇宙学研究：宇宙的起源和演化、宇宙大尺度结构、星系的形成和演化、暗物质、早期宇宙结构的演化”
- 冯珑珑，紫金山天文台，“星系和宇宙学研究:宇宙大尺度结构的形成、演化和统计特征，高红移宇宙的物理学，天体物理问题的大规模数值模拟研究早期宇宙结构的演化”
- 赵永恒，国家天文台，“活动天体的理论研究、高能天体的观测分析、多波段观测以及LAMOST 项目的科学研究和管理”



# 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、双黑洞系统和中子星之间的碰撞；
- 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（geomagnetic sub-storms）的模拟；

- 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



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



- 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

- 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

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






## 欧盟千万亿次应用问题

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

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






## 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

### 应用问题





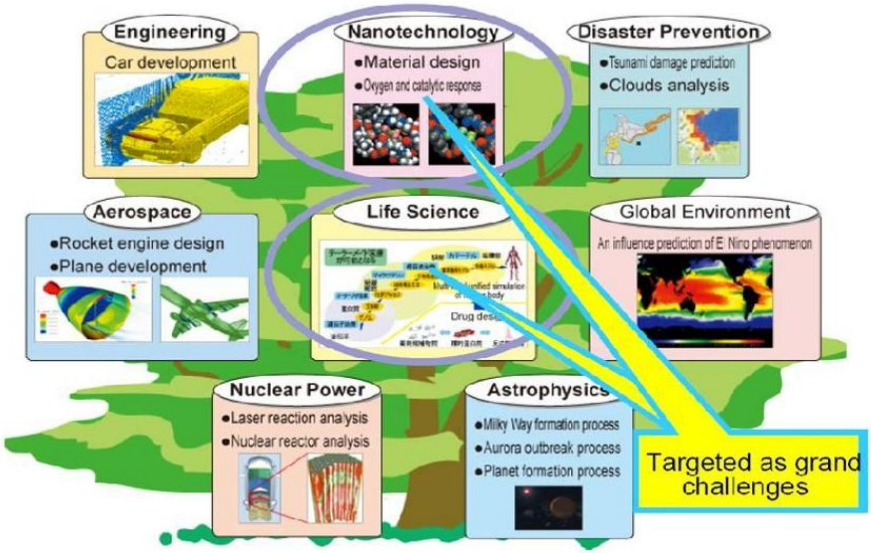




图 1.2 千万亿次计算机的应用领域

# 二、 并行计算的基本概念

2.1、并行计算机系统-MPP

2.2、并行计算机系统-SMP

2.3、并行计算机系统-Cluster

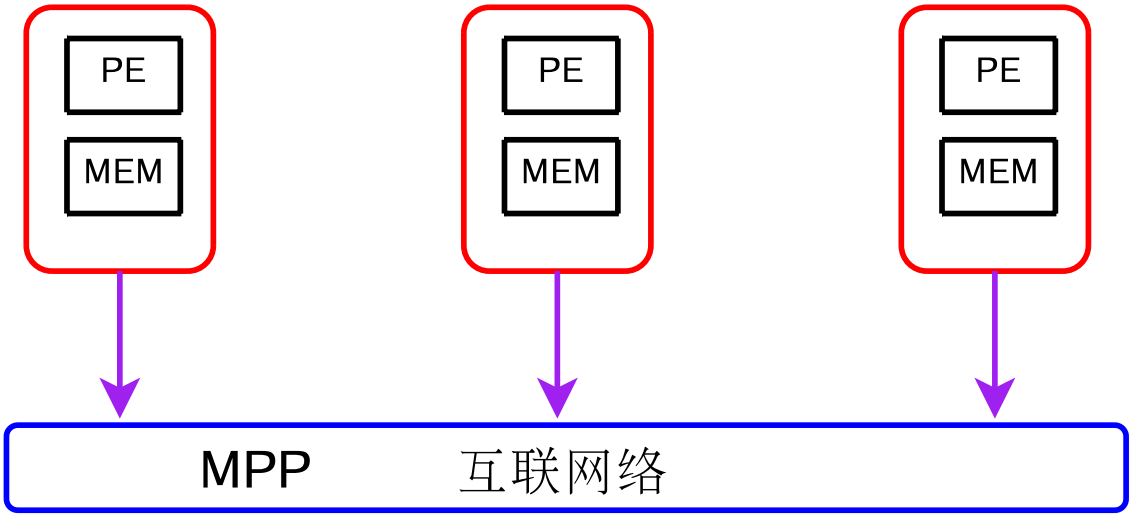
2.4、并行计算机系统的分类

2.5、并行计算的程序结构

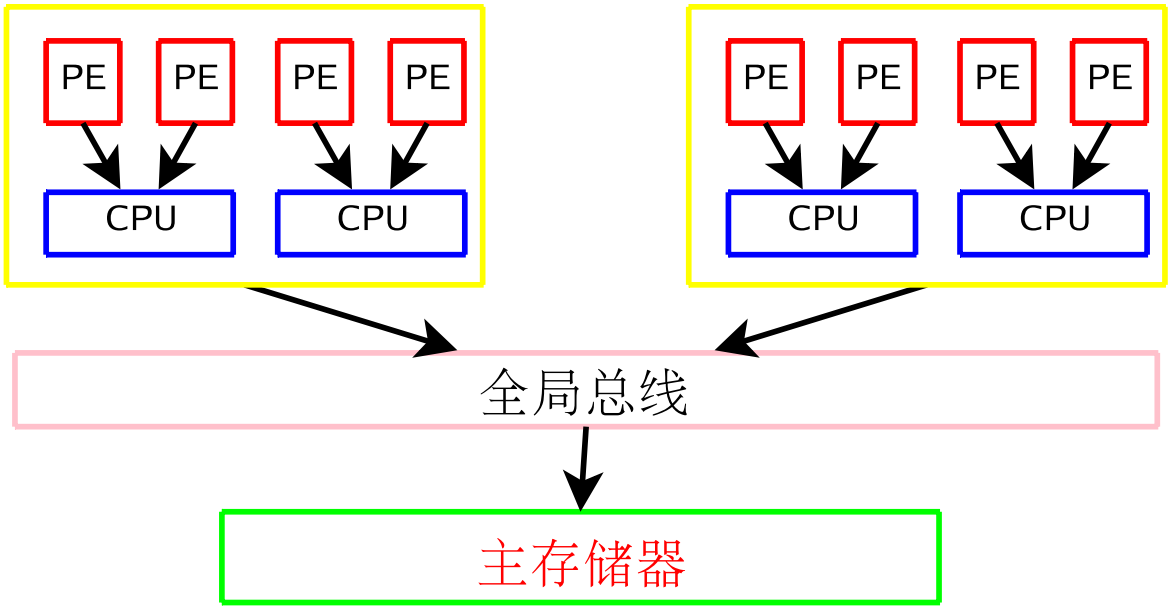
2.6、并行计算的基本定义



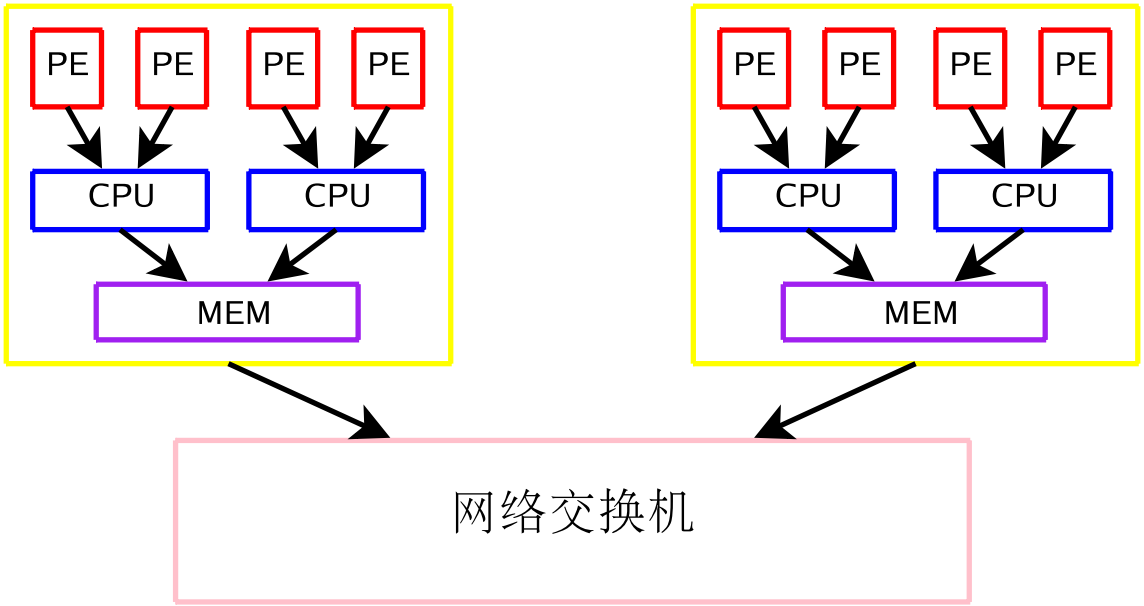
# 并行计算机系统-MPP



# 并行计算机系统-SMP



# 并行计算机系统-Cluster



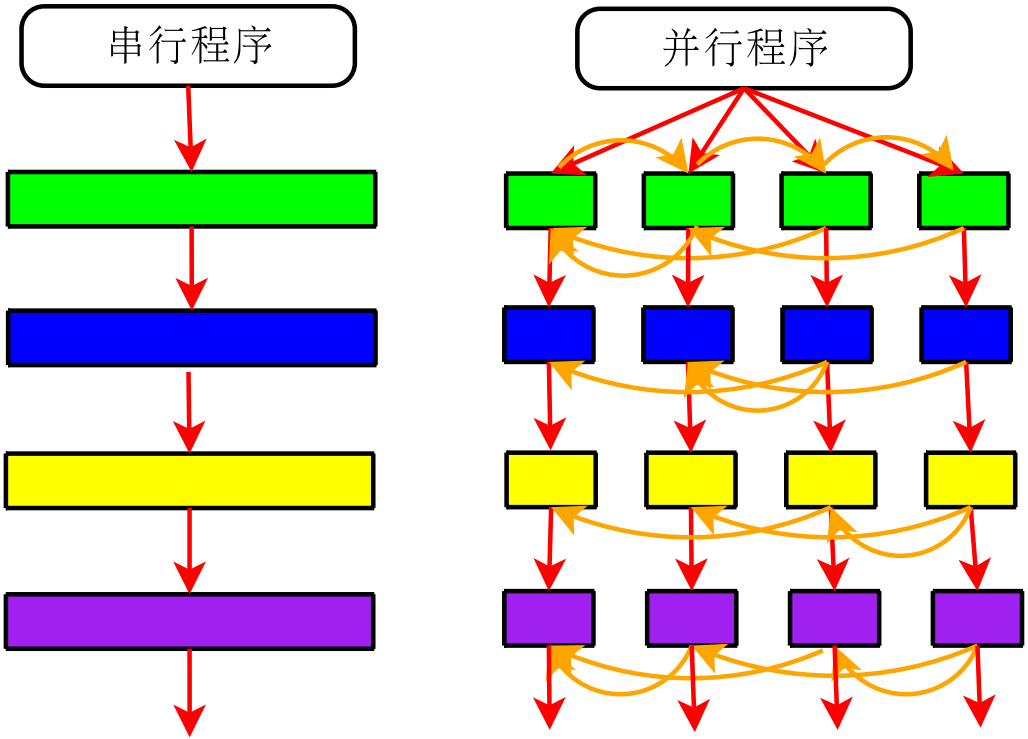
# 并行计算机系统的分类

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 2.2）
4. 多指令流多数据流(MIMD)，大规模并行处理系统、机群（参见 2.1、2.3）






# 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。

并行度：算法可以并行的程度。

加速比：

$$S_p(q) = \frac{T_s}{T_p(q)} \tag{2.1}$$

效率：

$$E_p(q) = \frac{S_p(q)}{q} \tag{2.2}$$



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

$$S_p(q) = \frac{1}{\alpha + (1-\alpha)/q} \tag{2.3}$$

$$\lim_{q\to\infty} S_p(q) = \frac{1}{\alpha} \tag{2.4}$$

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

$$S_p(q) = \frac{\alpha + (1-\alpha) \times q}{1} \tag{2.5}$$



# 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）。






# 三、 矩阵乘并行计算





# 矩阵卷帘（wrap）存储方式

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} & A_{05} & A_{06} & A_{07} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & A_{16} & A_{17} \\ A_{20} & A_{21} & A_{22} & A_{23} & A_{24} & A_{25} & A_{26} & A_{27} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & A_{36} & A_{37} \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & A_{46} & A_{47} \\ A_{50} & A_{51} & A_{52} & A_{53} & A_{54} & A_{55} & A_{56} & A_{57} \\ A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} & A_{67} \\ A_{70} & A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{pmatrix} \tag{3.1}$$






Back
Close

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：

$$\left(\begin{array}{cccc|cccc} A_{00} & A_{02} & A_{04} & A_{06} & A_{01} & A_{03} & A_{05} & A_{07} \\ A_{30} & A_{32} & A_{34} & A_{36} & A_{31} & A_{33} & A_{35} & A_{37} \\ A_{60} & A_{62} & A_{64} & A_{66} & A_{61} & A_{63} & A_{65} & A_{67} \\ \hline A_{10} & A_{12} & A_{14} & A_{16} & A_{11} & A_{13} & A_{15} & A_{17} \\ A_{40} & A_{42} & A_{44} & A_{46} & A_{41} & A_{43} & A_{45} & A_{47} \\ A_{70} & A_{72} & A_{74} & A_{76} & A_{71} & A_{73} & A_{75} & A_{77} \\ \hline A_{20} & A_{22} & A_{24} & A_{26} & A_{21} & A_{23} & A_{25} & A_{27} \\ A_{50} & A_{52} & A_{54} & A_{56} & A_{51} & A_{53} & A_{55} & A_{57} \end{array}\right) \tag{3.2}$$

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$P_{kl}$ $(k = i \mod p,\ l = j \mod q)$中。



# 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

$$A = \left[ A_0^{\mathrm{T}} \;\; A_1^{\mathrm{T}} \;\; \cdots \;\; A_{p-1}^{\mathrm{T}} \right]^{\mathrm{T}}, \quad B = \left[ B_0 \;\; B_1 \;\; \cdots \;\; B_{p-1} \right] \tag{3.3}$$

**算法 1** 行列分块算法

mp1 $\equiv$ myid+1 mod $p$, mm1 $\equiv$ myid$-$1 mod $p$

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

$l \equiv i$+myid mod $p$

$C_l = A \times B$

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

end$\{for\}$

# 行行分块算法

$$B = \left[ \begin{array}{cccc} B_0^{\mathrm{T}} & B_1^{\mathrm{T}} & \cdots & B_{p-1}^{\mathrm{T}} \end{array} \right]^{\mathrm{T}}, \quad A_i = \left[ \begin{array}{cccc} A_{i0} & A_{i1} & \ldots & A_{i,p-1} \end{array} \right] \quad (3.4)$$

**算法 2** 行行分块算法

mp1 $\equiv$ myid+1 mod $p$, mm1 $\equiv$ myid-1 mod $p$

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

$l \equiv i$+myid mod $p$

$C = C + A_l \times B$

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

end$\{for\}$

# 列行分块算法

$$A = \left[ A_0 \;\; A_1 \;\; \cdots \;\; A_{p-1} \right], \quad B_i = \left[ B_{i0} \;\; B_{i1} \;\; \ldots \;\; B_{i,p-1} \right] \tag{3.5}$$

**算法 3** 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$\{for\}$

# 列列分块算法

$$B_i = \left[ \begin{array}{cccc} B_{i0}^{\mathrm{T}} & B_{i1}^{\mathrm{T}} & \cdots & B_{i,p-1}^{\mathrm{T}} \end{array} \right]^{\mathrm{T}}$$

**算法 4** 列列分块算法

mp1 $\equiv$ myid+1 mod $p$, mm1 $\equiv$ myid-1 mod $p$

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

$l \equiv i$+myid mod $p$

$C = C + A \times B_l$

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

end{$for$}



# Cannon算法

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

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

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



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



**算法 5** *Cannon*算法

$C = 0$

mpc1 $\equiv$ mycol+1 mod $m$; mmc1 $\equiv$ mycol-1 mod $m$;

mpr1 $\equiv$ myrow+1 mod $m$; mmr1 $\equiv$ myrow-1 mod $m$;

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

$k \equiv$ myrow+$i$ mod $m$;

$r \equiv k - i$ mod $m$;

if mycol=$k$ & myrow=$r$ then

send($A$, (myrow, mpc1)); copy($A$, tmpA);

else if myrow=$r$

send(tmpA, (myrow, mmc1));

if k $\neq$ mpc1, send(tmpA, (myrow, mpc1));

end$\{if\}$

$C = C$+tmpA $\times B$;

if $i \neq m - 1$ then

send($B$, (mmr1, mycol)); recv($B$, (mpr1, mycol));

end$\{if\}$

end$\{for\}$



# 四、 线性代数方程组的并行求解

4.1、串行$LU$分解算法

4.2、分布式系统的并行$LU$分解算法

4.3、三角方程组的并行求解

4.4、经典迭代法-Jacobi

4.5、经典迭代法-Gauss-Seidel



# 串行$LU$分解算法



$$Ax = b \tag{4.1}$$

$PA = LU$，$Ly = Pb$，$Ux = y$。

**算法 6** 部分选主元的$Guass$消去算法

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

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

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

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

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

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

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

end$\{for\}$

end$\{for\}$



# 分布式系统的并行$LU$分解算法

icol= 0

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

if myid=$j$ mod $p$ then

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

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

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

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

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

else

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

if myid+1 $\neq j$ mod $p$i 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 $k$=icol to $m-1$ do

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

end$\{for\}$

end$\{for\}$



# 三角方程组的并行解法

$k = 0$

if myid=0, then

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

else

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

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

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

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



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

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

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

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

$k+1 \rightarrow k$

$\text{end}\{for\}$



# 经典迭代法-Jacobi

考虑求解线性代数方程组

$$Ax = b \tag{4.2}$$

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(4.2)可以变为

$$Dx = b + (L + U)x \tag{4.3}$$

如果方程组(4.3)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

迭代法如下：

$$x^{(k+1)} = D^{-1}(L+U)x^{(k)} + D^{-1}b \tag{4.4}$$

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：

$$x^{+} = Gx + g \tag{4.5}$$

关于这个迭代法的并行计算问题，在习题4.6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。



# 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(4.2)的分量表示

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, \qquad i = 1, \ldots, n \tag{4.6}$$

对于给定的初值$x^{(0)}$，Gauss-Seidel迭代法如下：

**算法 7** *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

$\cdots$

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

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

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

## 经典迭代法-Gauss-Seidel的并行化

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

**算法 8** *并行Gauss-Seidel迭代算法*

$k = 0$

for $i = 1, n$ do

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

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8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。



# 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$\bar{A}$，请建立矩阵$A$与矩阵$\bar{A}$和$P_{kl}$的关系式；
2. 假设矩阵$A$和向量$b$是按行分块的，请给出并行计算$x^+ = Ax + b$的方法；
3. 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。






# 五、 MPI并行程序设计

5.1并行程序类型、MPI-SPMD并行程序结构

5.2MPI并行环境管理函数

5.3MPI通信子操作

5.4点到点通信函数

5.5自定义数据类型

5.6MPI的数据打包与拆包

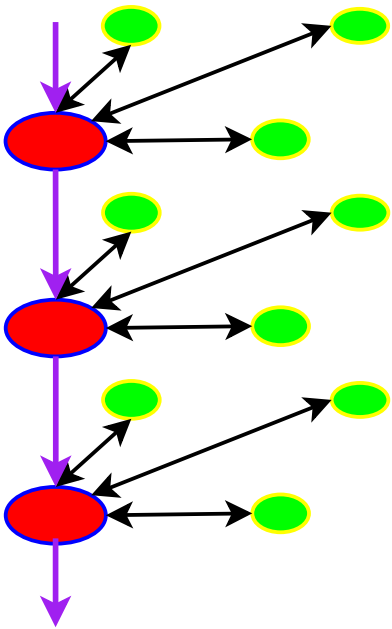
5.7MPI聚合通信

5.8MPI全局归约操作

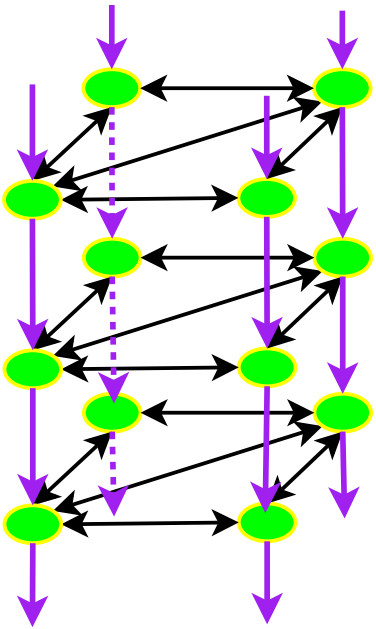
5.9MPI组操作



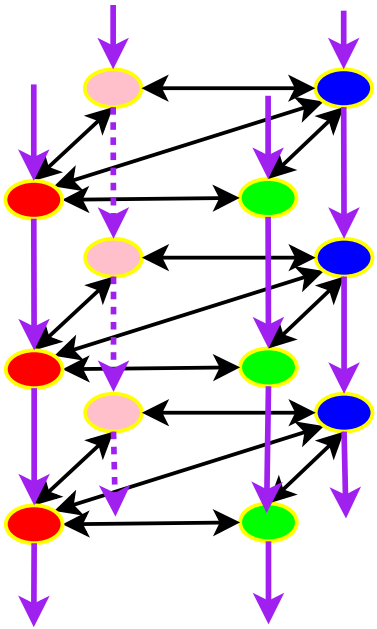
# 并行程序类型、MPI-SPMD并行程序结构



主从式M-S



对称式SPMD



自主式MPMD

## MPI并行程序的基本结构

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：

```
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，···。






# MPI并行环境管理函数



## MPI_INIT

| | |
|---|---|
| C | `int MPI_Init( int *argc, char ***argv )` |
| Fortran | `MPI_INIT( IERROR )` |
| | `INTEGER IERROR` |

## MPI_FINALIZE

| | |
|---|---|
| C | `int MPI_Finalize( void )` |
| Fortran | `MPI_FINALIZE( IERROR )` |
| | `INTEGER IERROR` |

## MPI_INITIALIZED

| | |
|---|---|
| C | `int MPI_Initialized( int flag )` |
| Fortran | `MPI_INITIALIZED( FLAG, IERROR )` |
| | `LOGICAL FLAG` |
| | `INTEGER IERROR` |




# 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 )`<br>`INTEGER COMM, SIZE, IERROR` |

## MPI_COMM_RANK

| | |
|---|---|
| C | `int MPI_Comm_rank( MPI_Comm comm, int *rank )` |
| Fortran | `MPI_COMM_RANK( COMM, RANK, IERROR )`<br>`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 )`<br>`INTEGER COMM, NEWCOMM, IERROR` |




## MPI_COMM_SPLIT

| | |
|---|---|
| C | `int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)` |
| Fortran | `MPI_COMM_SPLIT( COMM, COLOR, KEY, NEWCOMM, IERROR )`<br>`INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR` |

目前我们介绍的这几个MPI函数是并行程序中极其重要的，有了第 5.2和第 5.3 节的函数，我们就可以编写简单的并行程序。






# 点到点通信函数



## 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 )`<br>`<type> BUF(*)`<br>`INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR` |




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

MPI_STATUS的三个成员是MPI_SOURCE，MPI_TAG和MPI_ERROR。

## SNED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。






# 合成函数SENDRECV

MPI_SENDRECV

| | |
|---|---|
| C | int MPI_Sendrecv( void *sendbuf, int sendcount,<br>MPI_Datatype sendtype, int dest, int sendtag,<br>void *recvbuf, int recvcount, MPI_Datatype recvtype,<br>int source,int recvtag,MPI_Comm comm,MPI_Status *status) |
| Fortran | MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG,<br>RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM,<br>STATUS, IERROR )<br><type> SENDBUF(*), RECVBUF(*)<br>INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT,<br>RECVTYPE, SOURCE, RECVTAG, COMM,<br>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接收。



## 合成函数SENDRECV_Replace

MPI_SENDRECV_REPLACE

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

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。同时为方便使用MPI_SENDRECV函数，MPI还定义了一个空进程，用MPI_PROC_NULL来表示。

## 消息查询函数






Back
Close

## MPI_PROBE

| | |
|---|---|
| C | `int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status )` |
| Fortran | `MPI_PROBE( SOURCE, TAG, COMM, STATUS, IERROR )`<br>`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 )`<br>`LOGICAL FLAG`<br>`INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR` |

## MPI_GET_COUNT

| | |
|---|---|
| C | `int MPI_Get_count( MPI_Status *status, MPI_Datatype type, int *count )` |
| Fortran | `MPI_GET_COUNT( STATUS, TYPE, COUNT, IERROR )`<br>`INTEGER STATUS(MPI_STATUS_SIZE), TYPE, COUNT, IERROR` |






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

## 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 )`<br>`<type> BUF(*)`<br>`INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR` |






Back
Close

# 消息请求完成函数

## MPI_WAIT

| | |
|---|---|
| C | `int MPI_Wait( MPI_Request *request, MPI_Status *status )` |
| Fortran | `MPI_WAIT(REQUEST, STATUS, IERROR )`<br>`INTEGER REQUEST, STATUS(*), IERROR` |

## MPI_WAITANY

| | |
|---|---|
| C | `int MPI_Waitany( int count, MPI_Request *requests,`<br>`int *index, MPI_Status *status )` |
| Fortran | `MPI_WAITANY( COUNT, REQUESTS, INDEX, STATUS, IERROR )`<br>`INTEGER COUNT, REQUESTS(*), INDEX, STATUS(*), IERROR` |

## MPI_WAITALL

| | |
|---|---|
| C | `int MPI_Waitall( int count, MPI_Request *requests,`<br>`MPI_Status *statuses )` |
| Fortran | `MPI_WAITALL( COUNT, REQUESTS, STATUSES, IERROR )`<br>`INTEGER COUNT, REQUESTS(*), STATUS(*, *), IERROR` |






Back
Close

### 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 )`<br>`INTEGER INCOUNT, REQUESTS(*), OUTCOUNT, INDICES, STATUS(MPI_STATUS_SIZE, *), IERROR` |

## 消息请求检查函数

### MPI_TEST

| | |
|---|---|
| C | `int MPI_Test( MPI_Request *request, int *flag, MPI_Status *status )` |
| Fortran | `MPI_TEST(REQUEST, FLAG, STATUS, IERROR )`<br>`INTEGER REQUEST, FLAG, STATUS(*), IERROR` |

### MPI_TESTANY

| | |
|---|---|
| C | `int MPI_Testany( int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status )` |
| Fortran | `MPI_TESTANY( COUNT, REQUESTS, INDEX, FLAG, STATUS, IERROR )`<br>`INTEGER COUNT, REQUESTS(*), INDEX, FLAG, STATUS(*), IERROR` |






Back
Close

## MPI_TESTALL

| | |
|---|---|
| C | `int MPI_Testall( int count, MPI_Request *requests, int *flag, MPI_Status *statuses )` |
| Fortran | `MPI_TESTALL( COUNT, REQUESTS, FLAG, STATUSES, IERROR )`<br>`INTEGER COUNT, REQUESTS(*), FLAG, STATUS(*, *), IERROR` |

## 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 )`<br>`INTEGER INCOUNT, REQUESTS(*), OUTCOUNT, INDICES, STATUS(MPI_STATUS_SIZE, *), IERROR` |

## 持久通讯函数SEND_INIT和RECV_INIT






Back
Close

## 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)`<br>`<type> BUF(*)`<br>`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)`<br>`<type> BUF(*)`<br>`INTEGER COUNT, TYPE, SRC, TAG, COMM, REQUEST, IERROR` |

## MPI_START

| | |
|---|---|
| C | `int MPI_Start( MPI_Request *request )` |
| Fortran | `MPI_START(REQUEST, IERROR )`<br>`INTEGER REQUEST, IERROR` |



Back

Close

### MPI_STARTALL

| | |
|---|---|
| C | `int MPI_Startall( int count, MPI_Request *request )` |
| Fortran | `MPI_STARTALL(COUNT, REQUESTS, IERROR )`<br>`INTEGER COUNT, REQUESTS(*), IERROR` |

## 消息请求的释放与取消

消息请求完成之后，需要将请求句柄赋值为MPI_REQUEST_NULL。

### MPI_REQUEST_FREE

| | |
|---|---|
| C | `int MPI_Request_free( MPI_Request *request )` |
| Fortran | `MPI_REQUEST_FREE(REQUEST, IERROR )`<br>`INTEGER REQUEST, IERROR` |

### MPI_CANCEL

| | |
|---|---|
| C | `int MPI_Cancel( MPI_Request *request )` |
| Fortran | `MPI_CANCEL(REQUEST, IERROR )`<br>`INTEGER REQUEST, IERROR` |






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` |





# 自定义数据类型

数据类型的构成:

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

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

## CONTIGUOUS数据类型

MPI_TYPE_CONTIGUOUS

| | |
|---|---|
| C | `int MPI_Type_contiguous(int count,MPI_Datatype oldtype, MPI_Datatype *newtype )` |
| Fortran | `MPI_TYPE_CONTIGUOUS( COUNT, OLDTYPE, NEWTYPE, IERROR )`<br>`INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR` |




Back

Close

**例子 3** 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：

```
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 )
```

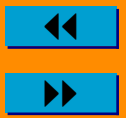
可以用这个函数构造双精度复数。

## 数据类型辅助函数

MPI_TYPE_COMMIT

| | |
|---|---|
| C | `int MPI_Type_commit( MPI_Datatype *datatype )` |
| Fortran | `MPI_TYPE_COMMIT( DATATYPE, IERROR )`<br>`INTEGER DATATYPE, IERROR` |







## MPI_TYPE_FREE

| | |
|---|---|
| C | `int MPI_Type_free( MPI_Datatype *datatype )` |
| Fortran | `MPI_TYPE_FREE( DATATYPE, IERROR )`<br>`INTEGER DATATYPE, IERROR` |

## MPI_TYPE_EXTENT

| | |
|---|---|
| C | `int MPI_Type_extent(MPI_Datatype datatype,MPI_Aint *extent)` |
| Fortran | `MPI_TYPE_EXTENT( DATATYPE, EXTENT, IERROR )`<br>`INTEGER DATATYPE, EXTENT, IERROR` |

## MPI_ADDRESS

| | |
|---|---|
| C | `int MPI_Address( void* location, MPI_Aint *address )` |
| Fortran | `MPI_ADDRESS( LOCATION, ADDRESS, IERROR )`<br>`<type> LOCATION(*)`<br>`INTEGER ADDRESS, IERROR` |

## 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 )`<br>`INTEGER COUNT,BLOCKLENGTH,STRIDE,OLDTYPE,NEWTYPE,IERROR` |

## 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 )`<br>`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。

# INDEX数据类型



## 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
```




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。

## STRUCT数据类型






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。



# 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
```

## 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

# MPI聚合通信

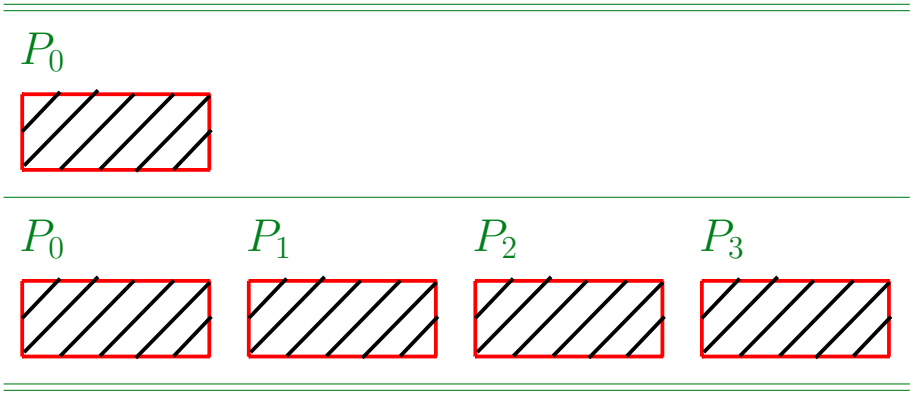
## 障碍同步MPI_Barrier

| MPI_BARRIER | |
|---|---|
| C | `int MPI_Barrier( MPI_Comm comm )` |
| Fortran | `MPI_BARRIER( COMM, IERROR )`<br>`INTEGER COMM, IERROR` |





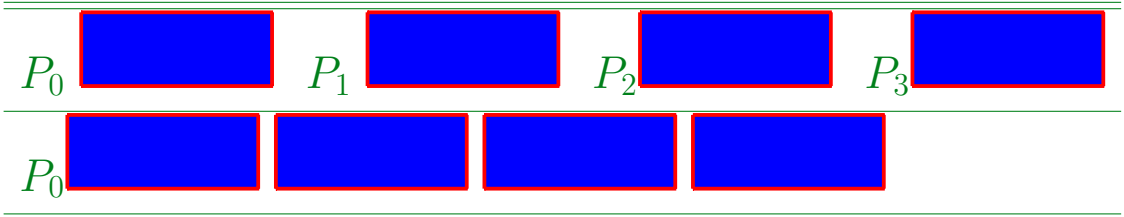

# 广播MPI_Bcast




## 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 )`<br>`<type> BUFFER(*)`<br>`INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR` |

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






Back

Close

# 收集MPI_Gatherv



## MPI_GATHERV

C

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

Fortran

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

## MPI_ALLGATHERV

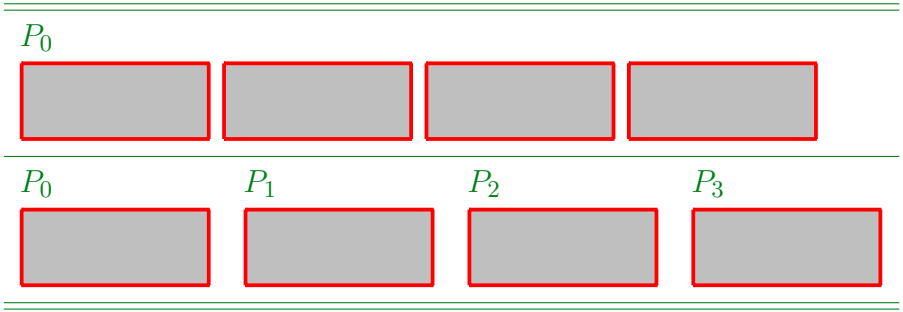
C

```
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
```

## 散播MPI_Scatter



## MPI_SCATTER(V)

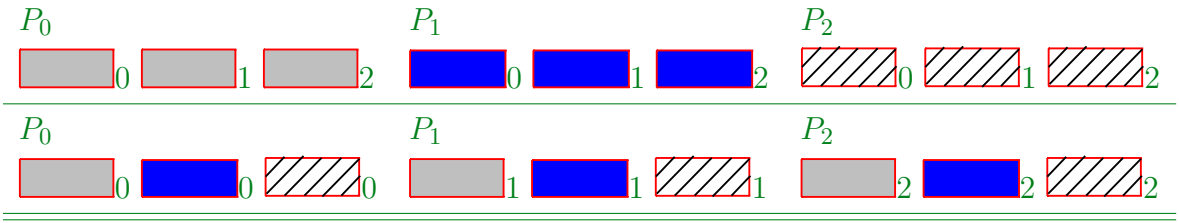
C

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

Fortran

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

## 全交换MPI_Alltoall



# MPI_ALLTOALL(V)

C

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

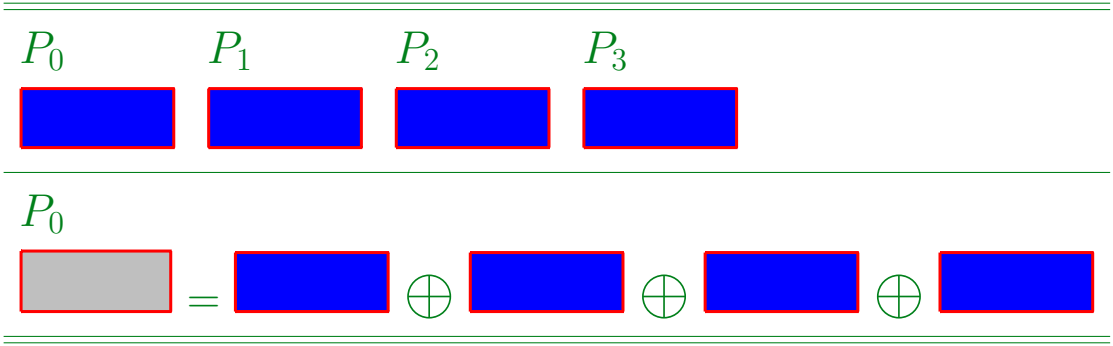
Fortran

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



# MPI全局归约操作

## 归约MPI_Reduce



# MPI_ALLREDUCE(REDUCE)

| | |
|---|---|
| C | `int MPI_Allreduce(Reduce)(void* sendbuf, void* recvbuf, int count,MPI_Datatype datatype, MPI_Op op, (int root,) MPI_Comm comm)` |
| Fortran | `MPI_ALLREDUCE(REDUCE)( SENDBUF, RECVBUF, COUNT, DATATYPE, OP, (ROOT,) COMM, IERROR )`<br>`<type> SENDBUF(*), RECVBUF(*)`<br>`INTEGER COUNT, DATATYPE, OP, (ROOT,) COMM, IERROR` |






Back
Close

# 运算种类与可用数据类型



| 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 |




Back
Close

| 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：




## Fortran 程序

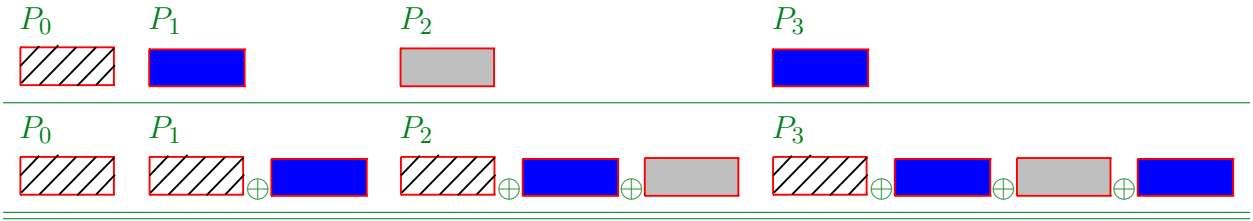
| 复合数据类型 | 类型描述 |
|---|---|
| MPI_2REAL | pair of REALs |
| MPI_2DOUBLE_PRECISION | pair of DOUBLE PRECISIONs |
| MPI_2INTEGER | pair of INTEGERs |

## 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 |



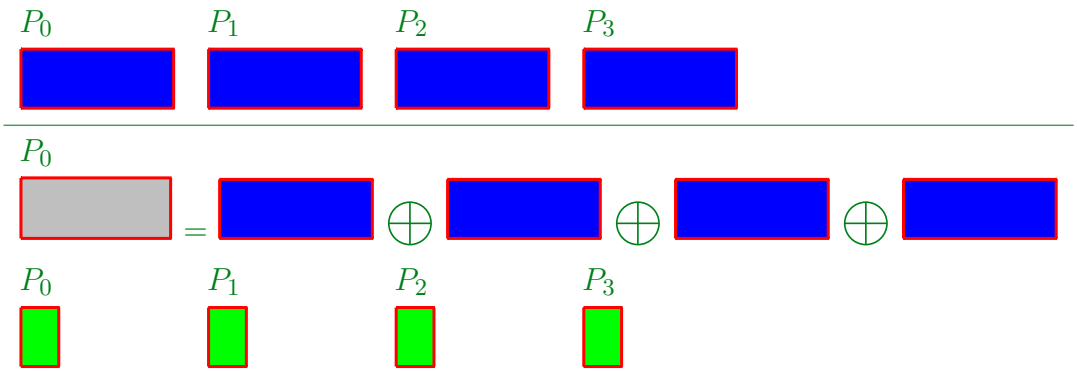
# 前缀MPI_Scan




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

# 归约散播MPI_Reduce_scatter




## 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
```

# MPI组操作

## 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 )`<br>`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 )`<br>`INTEGER GROUP1, GROUP2, GROUP, IERROR` |






Back
Close

## 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 )`<br>`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 )`<br>`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 )`<br>`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 )`<br>`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 )`<br>`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 )`<br>`INTEGER GROUP1, N, RANGES(3, *), GROUP, IERROR` |



Back

Close



## MPI_GROUP_SIZE

| | |
|---|---|
| C | `int MPI_Group_size( MPI_Group group, int *size )` |
| Fortran | `MPI_GROUP_SIZE( GROUP, SIZE, IERROR )`<br>`INTEGER GROUP, SIZE, IERROR` |

## MPI_GROUP_RANK

| | |
|---|---|
| C | `int MPI_Group_rank( MPI_Group group, int *rank )` |
| Fortran | `MPI_GROUP_RANK( GROUP, RANK, IERROR )`<br>`INTEGER GROUP, RANK, IERROR` |

## 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 )`<br>`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 )`<br>`INTEGER COMM, GROUP, NEWCOMM, IERROR` |

## MPI_GROUP_FREE

| | |
|---|---|
| C | `int MPI_Group_free( MPI_Group group )` |
| Fortran | `MPI_GROUP_FREE( GROUP, IERROR )`<br>`INTEGER GROUP, IERROR` |

## MPI_GROUP_COMPARE

| | |
|---|---|
| C | `int MPI_Group_compare( MPI_Group group1, MPI_Group group2, int *result )` |
| Fortran | `MPI_GROUP_COMPARE( GROUP1, GROUP2, RESULT, IERROR )`<br>`INTEGER GROUP1, GROUP2, RESULT, IERROR` |

RESULT的值为：MPI_IDENT，MPI_SIMILAR，MPI_UNEQUAL。






# 习题

1. 用MPI_SEND和MPI_RECV实现MPI_ALLTOALL的方法与程序；
2. 假设分块下三角矩阵的对角块矩阵都是$m \times m$阶的，给出一个下三角矩阵的数据类型，使得可以用于传送这个矩阵的多个对角块矩阵。






# 六、 并行程序实例

6.1 $\pi$值近似计算程序

6.2 数据广播并行程序

6.3 Cannon算法实现程序



# $\pi$值近似计算程序



**例子 4** $\pi$**值近似计算:**

由三角函数和定积分公式可知:

$$\frac{\pi}{4} = \arctan(1) = \int_0^1 \frac{1}{1+x^2} dx \tag{6.1}$$

假设将区间$[0,\ 1]$分成$n$等份，记$h = 1/n$，$x_i = ih$，$i = 0, 1, \ldots, n$，则采用梯形积分公式计算积分**(6.1)**如下:

$$\int_0^1 \frac{1}{1+x^2} dx = \sum_{i=0}^{n-1} \left[ \frac{h}{2} \left( \frac{1}{1+x_i^2} + \frac{1}{1+x_{i+1}^2} \right) - \frac{h^3}{12} \left( \frac{1}{1+\xi_i^2} \right)'' \right] \tag{6.2}$$

Back

Close

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：

$$|\int_0^1 \frac{4}{1+x^2}dx - \sum_{i=0}^{n-1}\left[\frac{h}{2}(\frac{4}{1+x_i^2}+\frac{4}{1+x_{i+1}^2})\right]| < \varepsilon \tag{6.3}$$

可以将

$$\sum_{i=0}^{n-1}\left[\frac{h}{2}(\frac{4}{1+x_i^2}+\frac{4}{1+x_{i+1}^2})\right] \tag{6.4}$$

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'
```



```
*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
```



```
      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
```






```
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
```



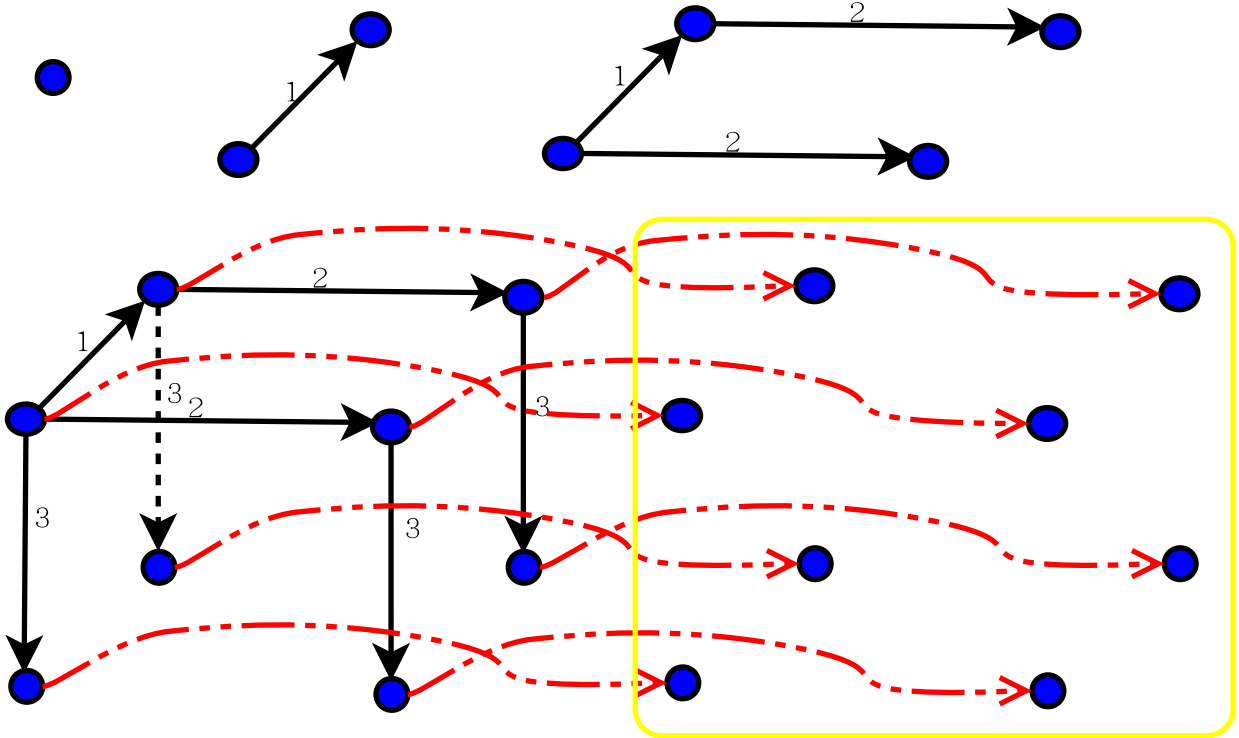
```
        ie = is + num
        xi = is * h
      endif

      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
```



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



# 数据广播并行程序



## 例子 5 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？

```
      program mpibcast
*The header file for using MPI parallel environment,
*which must be included for all mpi programs.
      include 'mpif.h'
*Variables declaration
      integer iam, np, comm, ierr
      real s(10)
*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)
*Main task begins
      s(1) = real(iam)
      s(2) = real(iam+1)
      call mpibcastr( s, 2, 1, comm, iam, np )
      write(*, *) 'Value on ', iam, ' is ', s(1), s(2)
      call mpi_finalize( ierr )
      end

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






Back
Close

```
 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
 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
```






Back
Close

```
          call mpi_recv( b, n, mpi_real, src, 1, comm,
     &                   status, ierr )
        endif
        mlen = 2*mlen
20     continue
       left = np - mlen
       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.
```



Back

Close

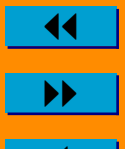
```
&         newid .lt. mlen+left ) then
    call mpi_recv( b, n, mpi_real, src, 1, comm,
&                  status, ierr )
 endif

 return
 end
```



# Cannon算法实现程序

```
      program computing_mat_mult
*The header file for using MPI parallel environment,
*which must be included for all mpi programs.
      implicit none
      include 'mpif.h'
*Variables declaration
      integer iam, np, comm, ierr
      integer n,i,j,m,k,npsr,status(mpi_status_size)
      integer lda, ldb, ldc, ldt
      parameter( lda=50, ldb=lda, ldc=lda, ldt=lda )
```





```
      real*8 a(lda, lda), b(ldb, ldb), c(ldc, ldc),
     &       t(ldt, ldt)
      integer myid, mycomm, rowcomm, colcomm, color,
     &        rcolor, rst, ccolor, key, ma, ka, kb,
     &        nb, mr, nr, kr, rowid, colid, kt, next,
     &        front, newtypea, newtypeb, blklen(5),
     &        displs(5), sizedble, blks
*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 order of matrix
      if(iam .eq. 0) then
        write(*, *) 'The order of matrix '
        read(*, *) m
      endif
      call mpi_bcast(m, 1, mpi_integer, 0, comm, ierr)

      k = m-1
      n = m
      npsr = sqrt( real(np) + 0.5 )
*Create the demanding processes
      if ( iam .lt. npsr*npsr ) then
        color = 0
```



```
      else
        color = 1
      endif
      key = iam
      call mpi_comm_split( comm, color, key, mycomm,
     &                     ierr )
      call mpi_comm_rank( mycomm, myid, ierr )
*The algorithm runs on the effective communicator
      if( color .eq. 0 ) then
*Create the row communicator for a sqare process array
        key = myid
        rcolor = myid / npsr
        call mpi_comm_split( mycomm, rcolor, key,
```



Back

Close

```
     &                          rowcomm, ierr )
        call mpi_comm_rank( rowcomm, colid, ierr )
*Create the column
        ccolor = mod( myid, npsr )
        call mpi_comm_split( mycomm, ccolor, key,
     &                          colcomm, ierr )
        call mpi_comm_rank( colcomm, rowid, ierr )
*The matrix a is m by k, b is k by n.
*Determine how many rows of matrix A in each process
        ma = m / npsr
        mr = mod( m, npsr )
        if( rowid .lt. mr ) then
          ma = ma + 1
```






Back
Close

```
        endif
*Determine how many columns of matrix A in each process
        ka = k / npsr
        kr = mod( k, npsr )
        if( colid .lt. kr ) then
          ka = ka + 1
        endif
*Determine how many rows of matrix B in each process
        kb = k / npsr
        kr = mod( k, npsr )
        if( rowid .lt. kr ) then
          kb = kb + 1
        endif
```



Back

Close

```
*Determine how many columns of matrix B in each process
        nb = n / npsr
        nr = mod( n, npsr )
        if( colid .lt. nr ) then
          nb = nb + 1
        endif
*Initialize the matrix A, B, and C
        call mat_init_a( npsr, a, lda, m, ma, k, ka,
     &                   rowid, colid )
        call mat_init_b( npsr, b, ldb, k, kb, n, nb,
     &                   rowid, colid )
        call mat_init_c(c, ldc, ma, nb)
        rst = rcolor
```



```
        front = mod(rowid-1+npsr, npsr)
        next = mod(rowid+1, npsr)
*Begin the main Cannon's algorithm
        do 10 i=1, npsr
*Copy the broadcasting matrix A into a matrix T
          if(colid .eq. rst) then
            call mat_copy(a, lda, ma, ka, t, ldt)
            kt = ka
          endif
*Broadcasting the matrix A in each row
          call mpi_bcast( kt, 1, mpi_integer, rst,
     &                    rowcomm, ierr)
*Define a new data type for matrix
```






Back
Close

```
            call mpi_type_vector( kt, ma, lda,
     &                            mpi_double_precision,
     &                            newtypea, ierr )
            call mpi_type_commit( newtypea, ierr )
            call mpi_bcast( t,1,newtypea,rst,rowcomm,
     &                      ierr)
            call mpi_type_free( newtypea, ierr )
*Do matrix multiplication
            call dgemm(t,ldt,b,ldb,c,ldc,ma,kt,nb)
*Move matrix B in each column communicator
            if(i .ne. npsr) then
              call mat_copy(b, ldb, kt, nb, t, ldt)
              call mpi_sendrecv(t, ldt*nb,
```



```
     &                          mpi_double_precision,
     &                          front,1,b,ldb*nb,
     &                          mpi_double_precision,
     &                          next,1,colcomm,status,
     &                          ierr)
            rst = mod(rst+1, npsr)
          endif
10      continue

        print *, 'the end on process ', iam
        write(6, 99) ((c(i, j), i=1, 4), j=1, 4)
        call mpi_comm_free( rowcomm, ierr )
        call mpi_comm_free( colcomm, ierr )
```



Back

Close

```
      else
        print *, 'The process ', iam, ' is idle!'
      endif

99    format(1x, 4f15.9)
      call mpi_comm_free( mycomm, ierr )

*leave the MPI environment
      call mpi_finalize(ierr)

      end
*************************************************
* A(i, j) = i+j
```



```
* B(i, j) = 1.0 -2.0*mod(i+j, 2)

      subroutine mat_init_a( npsr, a, lda, m, ma, k,
     &                       ka, rowid, colid )
      integer lda, ma, ka, rowid, colid, m, k, npsr
      real*8 a(lda, *)
      integer i, j, mt, mr, ist, jst, kt, kr, offside

      mt = m / npsr
      mr = mod(m, npsr)
      if(rowid .lt. mr ) then
        ist = (mt+1)*rowid
      else
```



```
  ist = mt*rowid+mr
endif
kt = k / npsr
kr = mod(k, npsr)
if(colid .lt. kr ) then
  jst = (kt+1)*colid
else
  jst = kt*colid+kr
endif
offside = ist + jst
do 10 j=1, ka
  do 10 i=1, ma
    a(i, j) = offside + i + j
```



```
10     continue
       return
       end

       subroutine mat_init_b( npsr, b, ldb, k, kb, n,
      &                       nb, rowid, colid )
       integer ldb, nb, kb, rowid, colid, n, k, npsr
       real*8 b(ldb, *)
       integer i, j, kt, kr, nt, nr, ist, jst, offside

       kt = k / npsr
       kr = mod(k, npsr)
       if(rowid .lt. kr ) then
```



```
  ist = (kt+1)*rowid
else
  ist = kt*rowid+kr
endif
nt = n / npsr
nr = mod(n, npsr)
if(colid .lt. nr ) then
  jst = (nt+1)*colid
else
  jst = nt*colid+nr
endif
offside = ist + jst
do 10 j=1, nb
```



```
        do 10 i=1, kb
          b(i, j) = 1.0-2.0*mod(offside + i + j, 2)
10      continue

        return
        end

        subroutine mat_init_c(c, ldc, ma, nb)
        integer ldc, ma, nb
        real*8 c(ldc, *)
        integer i, j

        do 10 j=1, nb
```



```
      do 10 i=1, ma
        c(i, j) = 0.0
10    continue
      return
      end

      subroutine mat_copy(a, lda, ma, ka, t, ldt)
      integer lda, ma, ka, ldt
      real*8 a(lda, *), t(ldt, *)

      integer i, j

      do 10 j=1, ka
```



```
        do 10 i=1, ma
          t(i, j) = a(i, j)
10    continue

      return
      end

      subroutine dgemm( t, ldt, b, ldb, c, ldc, ma,
     &                  kt, nb)
      integer ldt, ldb, ldc, ma, kt, nb
      real*8 t(ldt, *), b(ldb, *), c(ldc, *)

      integer i, j, k
```






Back
Close

```
      do 10 j=1, nb
        do 10 k=1, kt
          do 10 i=1, ma
            c(i, j)=c(i, j)+t(i, k)*b(k, j)
10    continue

      return
      end
```



# 参考文献

[1] 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

[2] 陈国良，≪并行计算－－结构•算法•编程≫，高等教育出版社，2003

[3] 莫则尧、 袁国兴， ≪消息传递并行编程环境MPI≫， 科学出版社，2001

[4] 基本要求： 熟悉Fortran或C程序设计语言，了解数值计算方法