MPI 并行编程讲稿

张林波
中国科学院数学与系统科学研究院
科学与工程计算国家重点实验室

2006 年 12 月 1 日

目 录

参考材料 **1**

第一章 预备知识 **2**

1.1 高性能并行计算机系统简介 2

1.1.1 微处理器的存储结构 2

1.1.2 Cache 结构对程序性能的影响 2

1.1.3 共享内存 SMP 型并行计算机 2

1.1.4 分布式内存 MPP 型并行计算机 3

1.1.5 DSM 型并行计算机 3

1.1.6 SMP/DSM 机群 3

1.1.7 微机/工作站机群 4

1.1.8 TOP500 4

1.2 并行编程模式 4

1.2.1 自动并行与手工并行 4

1.2.2 OpenMP 5

1.2.3 DSM 编程模式 5

1.2.4 高性能 Fortran: HPF 5

1.2.5 消息传递并行编程模式 5

1.3 Unix 程序开发简介 6

1.3.1 Unix 中常用的编译系统 6

1.3.2 实用工具 `make` 7

1.4 消息传递编程平台 MPI 8

1.4.1 MPI 程序的编译与运行 8

1.4.2 利用 MPICH 建立 MPI 程序开发与调试环境 8

第二章 MPI 基础知识 **10**

2.1 下载 MPI 标准的 PS 文档 10

2.2 一些名词与概念 10

2.3 编程模式 10

2.4 MPI 函数的一般形式 11

2.5 MPI 的原始数据类型 11

2.5.1 Fortran 77 原始数据类型 . . . 11
2.5.2 C 原始数据类型 . . . 12
2.6 MPI 的几个基本函数 . . . 12
2.6.1 初始化 MPI 系统 . . . 12
2.6.2 检测 MPI 系统是否已经初始化 . . . 12
2.6.3 得到通信器的进程数及进程在通信器中的序号 . . . 13
2.6.4 退出 MPI 系统 . . . 13
2.6.5 异常终止 MPI 程序的执行 . . . 13
2.6.6 查询处理器名称 . . . 13
2.6.7 获取墙上时间及时钟精度 . . . 14
2.7 MPI 程序的基本结构 . . . 14
2.7.1 Fortran 77 程序 . . . 14
2.7.2 C 程序 . . . 15

第三章 点对点通信 **16**
3.1 标准阻塞型点对点通信函数 . . . 16
3.1.1 标准阻塞发送 . . . 16
3.1.2 阻塞接收 . . . 16
3.1.3 阻塞型消息传递实例 . . . 17
3.1.4 其它一些阻塞型消息传递函数 . . . 18
3.2 消息发送模式 . . . 19
3.2.1 阻塞型缓冲模式消息发送函数 . . . 20
3.3 阻塞型与非阻塞型函数 . . . 20
3.4 非阻塞型点对点通信函数 . . . 21
3.4.1 非阻塞发送 . . . 21
3.4.2 非阻塞接收 . . . 21
3.4.3 通信请求的完成与检测 . . . 22
3.4.4 通信请求的释放 . . . 24
3.5 消息探测与通信请求的取消 . . . 24
3.5.1 消息探测 . . . 24
3.5.2 通信请求的取消 . . . 25
3.6 点对点通信函数汇总 . . . 26
3.7 持久通信请求 . . . 26
3.7.1 创建持久消息发送请求 . . . 26
3.7.2 创建持久消息接收请求 . . . 27

3.7.3 开始基于持久通信请求的通信 . . . 27
3.7.4 持久通信请求的完成与释放 . . . 27

第四章 数据类型 28

4.1 与数据类型有关的一些定义 . . . 28
4.1.1 数据类型定义 . . . 28
4.1.2 数据类型的大小 . . . 29
4.1.3 数据类型的下界、上界与域 . . . 29
4.1.4 MPI_LB 和 MPI_UB . . . 30
4.1.5 数据类型查询函数 . . . 31
4.2 数据类型创建函数 . . . 32
4.2.1 MPI_Type_contiguous . . . 32
4.2.2 MPI_Type_vector . . . 32
4.2.3 MPI_Type_hvector . . . 33
4.2.4 MPI_Type_indexed . . . 34
4.2.5 MPI_Type_hindexed . . . 35
4.2.6 MPI_Type_struct . . . 35
4.2.7 地址函数 MPI_Address . . . 36
4.3 数据类型的使用 . . . 37
4.3.1 数据类型的提交 . . . 37
4.3.2 数据类型的释放 . . . 37
4.3.3 MPI_Get_elements . . . 39
4.4 数据的打包与拆包 . . . 39
4.4.1 数据打包 . . . 39
4.4.2 数据拆包 . . . 39
4.4.3 得到打包后的数据大小 . . . 40
4.5 MPI 1.1 中位移与数据大小的限制 . . . 41

第五章 聚合通信 (Collective Communications) 42

5.1 障碍同步 . . . 42
5.2 广播 . . . 42
5.3 数据收集 . . . 43
5.3.1 收集相同长度数据块 MPI_Gather . . . 43
5.3.2 收集不同长度数据块 MPI_Gatherv . . . 44
5.3.3 全收集 MPI_Allgather . . . 44
5.3.4 不同长度数据块的全收集 MPI_Allgatherv . . . 45

5.4 数据散发 . . . 46
5.4.1 散发相同长度数据块 MPI_Scatter . . . 46
5.4.2 散发不同长度数据块 MPI_Scatterv . . . 47
5.5 全部进程对全部进程的数据散发收集 . . . 48
5.5.1 相同数据长度的全收集散发 MPI_Alltoall . . . 48
5.5.2 不同数据长度的全收集散发 MPI_Alltoallv . . . 49
5.6 归约 . . . 50
5.6.1 归约函数 MPI_Reduce . . . 50
5.6.2 全归约 MPI_Allreduce . . . 52
5.6.3 归约散发 MPI_Reduce_scatter . . . 52
5.6.4 前缀归约 MPI_Scan . . . 53
5.6.5 归约与前缀归约中用户自定义的运算 . . . 54
5.7 两个程序实例 . . . 55
5.7.1 π 值计算 . . . 55
5.7.2 Jacobi 迭代求解二维 Poisson 方程 . . . 55

第六章 进程组与通信器 59

6.1 基本概念 . . . 59
6.1.1 进程组 . . . 59
6.1.2 上下文 (Context) . . . 59
6.1.3 域内通信器 (Intracommunicator) . . . 59
6.1.4 域间通信器 (Intercommunicator) . . . 59
6.2 进程组操作函数 . . . 59
6.2.1 查询进程组大小和进程在组中的序号 . . . 59
6.2.2 两个进程组间进程序号的映射 . . . 60
6.2.3 比较两个进程组 . . . 60
6.2.4 进程组的创建与释放 . . . 60
6.3 域内通信器操作函数 . . . 63
6.3.1 比较两个通信器 . . . 63
6.3.2 通信器的创建与释放 . . . 63
6.4 通信器的附加属性 (Caching) . . . 65
6.5 域间通信器 (Intercommunicator) . . . 65
6.6 进程拓扑结构 . . . 65
6.6.1 迪卡尔拓扑结构 . . . 65
6.6.2 一般拓扑结构 . . . 68

6.6.3 底层支持函数 70

第七章 文件输入输出 **71**

7.1 基本术语 71
7.2 基本文件操作 72
7.2.1 打开 MPI 文件 72
7.2.2 关闭 MPI 文件 73
7.2.3 删除文件 73
7.2.4 设定文件长度 73
7.2.5 为文件预留空间 74
7.2.6 查询文件长度 74
7.3 查询文件参数 74
7.3.1 查询打开文件的进程组 74
7.3.2 查询文件访问模式 74
7.4 设定文件视窗 75
7.4.1 文件中的数据表示格式 75
7.4.2 可移植数据类型 76
7.4.3 查询数据类型相应于文件数据表示格式的域 76
7.5 文件读写操作 76
7.5.1 使用显式位移的阻塞型文件读写 77
7.5.2 使用独立文件指针的阻塞型文件读写 78
7.5.3 使用共享文件指针的阻塞型文件读写 78
7.5.4 非阻塞型文件读写函数 79
7.5.5 分裂型文件读写函数 79
7.6 文件指针操作 80
7.6.1 独立文件指针操作 80
7.6.2 共享文件指针操作 80
7.6.3 文件位移在文件中的绝对地址 81
7.7 不同进程对同一文件读写操作的相容性 81
7.7.1 设定文件访问的原子性 81
7.7.2 查询 atomicity 的当前值 82
7.7.3 文件读写与存储设备间的同步 82
7.8 子数组数据类型创建函数 83

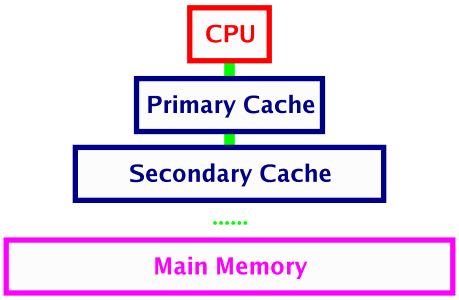
参考材料

- 莫则尧等,《消息传递并行编程环境 MPI》, 科学出版社, 2001.
- 都志辉等,《高性能并行计算编程技术 MPI 并行程序设计》，清华大学出版社，2001
- 张林波等,《并行计算导论》，清华大学出版社，2006.
- http://www.mpi-forum.org: 【*】http://www.mpi-forum.org
- ftp://ftp.cc.ac.cn/pub/home/zlb/bxjs/bxjs.pdf:
  【*】ftp://ftp.cc.ac.cn/pub/home/zlb/bxjs/bxjs.pdf
- 本讲义中的程序实例的下载地址：
  ftp://ftp.cc.ac.cn/pub/home/zlb/bxjs/: 【*】ftp://ftp.cc.ac.cn/pub/home/zlb/bxjs/

第一章　预备知识

§1.1　高性能并行计算机系统简介

§1.1.1　微处理器的存储结构



§1.1.2　Cache 结构对程序性能的影响

例 1.1 矩阵乘法中不同循环顺序对程序性能的影响.

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j} \qquad i,j = 1,\ldots,n$$

```
DO J=1,N
   DO I=1,N
      C(I,J) = 0.DO
   ENDDO
ENDDO
DO I=1,N
   DO J=1,N
      DO K=1,N
         C(I,J) = C(I,J) + A(I,K) * B(K,J)
      ENDDO
   ENDDO
ENDDO
```

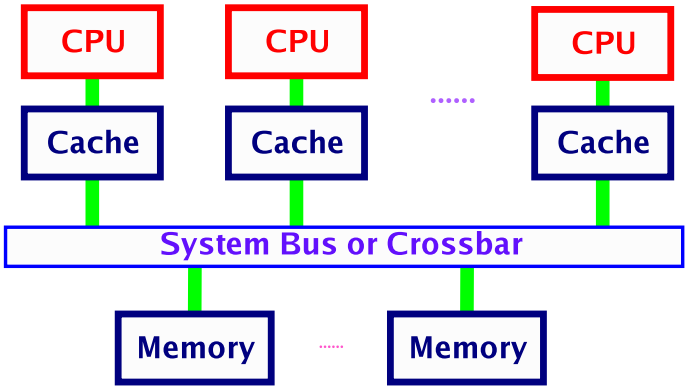
矩阵乘法实例 1: 【matmul/matmul0.m4】

矩阵乘法实例 2: 【matmul/matmul.f】

§1.1.3　共享内存 SMP 型并行计算机

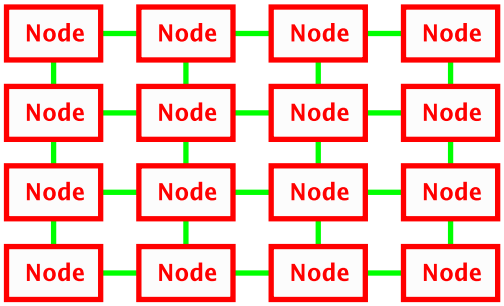
- 对称多处理器 (Symmetric Multi–Processors), 或共享内存处理器 (Shared Memory Processors).
- 多个处理器通过系统总线或交叉开关共享一个或多个内存模块.
- 优点: 使用简单, 维护方便.
- 缺点: 受系统总线带宽限制, 只能支持少量处理器 (一般十几个).
- 并行编程方式: 通常采用 OpenMP, 也可使用消息传递 (MPI/PVM) 及 HPF.

- 代表机型: `SGI Power Challenge:【01-chall.sh】`, Sun E10000, 等.



§1.1.4 分布式内存 MPP 型并行计算机

- Massively Parallel Processors 的简称
- 指由大量具有局部内存的计算结点通过高速系统网络联接而构成的并行处理系统.
- MPP 系统的系统网络通常具有某种拓扑结构 (如 tree, mesh, torus, hypercube).



§1.1.5 DSM 型并行计算机

分布共享内存: Distributed Shared Memory

- 多个物理上具有独立内存的处理单元, 通过高速网络联接在一起.
- 逻辑上作为共享内存并行机使用.
- 也称为 NUMA 结构 (NonUniform Memory Access).
- 不同处理单元间内存的共享通过特殊的硬件/软件实现.
- 具有比 SMP 型并行机更好的可扩展性 (超过 100 个 CPU).
- 代表机型: SGI Origin 2000/3000.

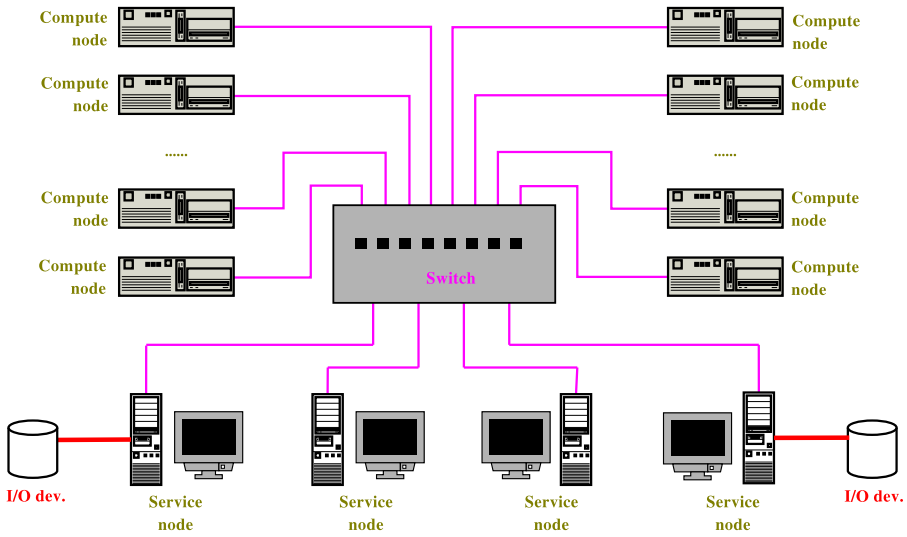
§1.1.6 SMP/DSM 机群

- 将多台 SMP 或 DSM 并行机通过互联网络连接而成.
- 目前国内外最高性能的并行机大多是这种结构.

§1.1.7 微机/工作站机群

- 微机机群 (PC cluster, 又称为 Beowulf cluster), 工作站机群 (NOW, Network Of Workstations): 将联网的多台微机或工作站组织成一台并行计算机. 目前常用的网络有以太网 (100Mbps), ATM (155/622Mbps), Myrinet (1.2Gbps, `http://www.myricom.com`).
- 适合于构造中等规模的并行系统 (多达数百个处理器).
- 根据机群中所使用的机型可为同构型和异构型两种.
- 根据机群的使用方式又可分为专用型和兼用型. 前者指该机群专门用于并行计算
- 微机/工作站机群的优点是价格便宜、配置灵活. 但其规模及并行效率受网络设备的制约.

配以适当的系统管理工具及作业调度、性能监控、并行程序调试开发环境、及外设等, 微机/工作站机群系统可以达到与商用 MPP 系统一样的使用效果.



§1.1.8 TOP500

`TOP500 Supercomputer Sites:` 【`http://netlib.amss.ac.cn/benchmark/top500.html`】

§1.2 并行编程模式

§1.2.1 自动并行与手工并行

- 在 SMP 及 DSM 并行机上编译系统通常具有一定的对用户程序 (C/Fortran) 进行自动并行化的能力, 但经常需要人工干预 (通过指导语句、命令行选项等) 以达到理想的并行效率. 并行主要针对循环进行 (细粒度并行).
- 在分布式内存并行机上尚无通用高效的自动并行工具, 主要依靠人工编写并行程序.

- 并行算法的设计及并行程序的编制已成为目前制约大规模并行计算机应用的主要障碍.

§1.2.2 OpenMP

在串行程序的循环语句前插入特定的指导语句, 告知编译系统一些有助于对该循环进行并行的信息以及/或是强制编译系统按指定的方式将该循环并行化.

- 主要限于 SMP 及 DSM 型的并行系统. 现在也已发展到一些 MPP 系统.
- 通常结合编译系统的自动并行化功能使用.
- 也有一些自动并行化工具, 它们对程序结构及数据流进行分析, 然后自动插入适当的 OpenMP 语句.
- OpenMP 的优点是学习及使用相对简单, 很多情况下不需要对原有算法及程序做大的改动. 缺点是只适合一类特定的机型, 并且程序的可扩展性通常不如用消息传递方式编写的并行程序.
- 对一些具有强数据相关性的计算过程需要通过改变计算顺序或修改算法甚至设计新的算法来实现并行计算.
- OpenMP 程序实例: 矩阵乘: 【matmul/matmul-omp.f】

§1.2.3 DSM 编程模式

- 建立在某种内存一致性协议之上, 通过软件或软硬件结合的方式实现处理器间内存的共享.
- 通过要求 DSM 并行程序中对内存的读写遵循一定的规则来减小维护内存一致性的开销.
- 参看: http://lssc.cc.ac.cn/Parallel-programming/唐志敏/jiajia.ppt
- 软件 DSM 系统实例: JiaJia: 【http://www.ict.ac.cn/chpc/chinese/field1.htm】

§1.2.4 高性能 Fortran: HPF

- 基于数据并行的编程语言, 即用户通过指导语句指示编译系统将数据分布到各处理器上, 编译系统根据数据分布的情况生成并行程序.
- 主要面向 Fortran 90/95.
- 优点: 编程相对简单, 串并行程序一致.
- 适合于 SMP/DSM/MPP/cluster 系统.
- 主要缺点是并行过程对用户的透明度不高, 程序的性能很大程度上依赖于所用的编译系统及用户对编译系统的了解, 需要针对不同的 HPF 编译器做不同的优化, 影响了程序的可移植性.
- HPF 程序实例: 矩阵乘: 【matmul/matmul-hpf.f】

§1.2.5 消息传递并行编程模式

- 并行程序由一组独立运行的进程构成. 进程间通过相互发送消息来实现数据交换.
- 可以说消息传递并行编程是并行应用程序开发的最底层编程方式之一. 很多其它并行开发语言或工具 (如一些 HPF 编译器) 将程序转化成消息传递型并行程序来实现并行.
- 常用的消息传递平台有 PVM (Parallel Virtual Machine) 和 MPI (Message Passing Interface).
- 程序通用性好, 用 PVM 或 MPI 编写的程序可以在任何并行计算机上运行.
- 能够达到很高的并行效率, 具有很好的可扩展性.
- 缺点: 程序的编制与调试比较困难, 许多情况下要对程序甚至算法做大的改动.

§1.3 Unix 程序开发简介

§1.3.1 Unix 中常用的编译系统

- 编译器由前端和后端组成. 通常用户只需使用前端命令即可完成编译、链接.
- C 编译器: `cc`, `gcc` (GNU C) 等.
- Fortran 编译器: `f77`, `fc`, `g77` (GNU Fortran), `f90` (Fortran 90) 等.
- 可用 `man` 查看使用手册, 如: `man cc`, `man f77` 等等.
- 命令行形式:

  `% cc` [options] files [options]

  `% f77` [options] files [options]
- 文件的类型由文件的扩展名决定:
  - C 源代码:`.c`;
  - Fortran 77 源代码:`.f`;
  - 带预处理的 Fortran 源代码:`.F`;
  - Fortran 90 源代码:`.f90`;
  - C++ 源代码:`.c++`, `.C`, `.cpp`, `.cc`, `.cxx`;
  - 汇编代码:`.s`, `.S`;
  - 目标文件:`.o`;
  - 库文件:`.a`;
  - 共享库:`.so`;
- 命令行选项:
  - `-c`: 只编译, 不链接, 即只生成 `.o` 文件.
  - `-o` *filename*: 指定输出文件名, 缺省为 `*.o`, `a.out` 等.
  - `-I`*path*: 指定 (增加) 包含文件 (如 `*.h`) 的搜索目录.
  - `-L`*path*: 指定 (增加) 库文件的搜索目录.
  - `-l`*name*: 与库文件 `lib`*name*`.a` 链接.
  - 优化开关: `-O`, `-O1`, `-O2`, `-O3`, 等等.
  - 目标码中包含源文件名、行号等信息 (用于程序调试): `-g`.
- 例:
  - `f77 -O2 -o prog file1.f file2.c file3.o file4.a`
  - `f77 -c file.f`
    `f77 -o out file.o`
  - `f77 -c -I/usr/local/mpi/include file.f`
    `f77 -o prog -L/usr/local/mpi/lib file.o -lmpi` (等价于:
    `f77 -o prog file.o /usr/local/mpi/lib/libmpi.a`)

§1.3.2 实用工具 make

- 命令形式:

  ```
  make [-f Makefile] [options] [target [target ...]]
  ```

  其中 -f 选项给出定义规则的文件名 (简称 Makefile 文件), 缺省使用当前目录下的 Makefile 或 makefile 文件. target 指明要求生成的目标 (在 Makefile 中定义), 当命令行中不给出 target 时 make 只生成 Makefile 中定义的第一个目标. 比较有用也较通用的命令行选项有下面一些:

  -f 文件名: 指定 Makefile 文件名.

  -n: 只显示将要执行的命令而并不执行它们.

  -p: 显示定义的全部规则及宏, 用于对 Makefile 的调试.

- 通过 Makefile 文件定义一组文件之间的依赖关系及处理命令, 方便程序开发过程中的编译与维护.
- 处理规则的建立以特定的文件扩展名及文件修改时间为基础. 缺省支持常用的程序扩展名: .c, .f, .F, .o, .a, .h, 等等. 用户可以通过 .SUFFIXES: 目标定义新的文件扩展名.
- 基本规则:

  ```
  目标: 依赖对象
  <tab> 处理命令
  <tab> ... ...
  ```

  例:

  ```
  prog: file1.f file2.f file3.c
          f77 -O2 -o prog file1.f file2.f file3.c
  ```

  其含义为: 如果目标 (prog) 不存在, 或者任何一个依赖对象 (file1.f, file2.f, file3.c) 比目标新, 则执行指定的命令.

- 宏定义:

  ```
  SRC=file1.f file2.f file3.c
  prog: $(SRC)
          f77 -O2 -o prog $(SRC)
  ```

  环境变量可以在 Makefile 中作为宏使用, 如 $(HOME).

- 常用预定义的宏:
  - $@: 代表目标名 (上例中为 prog)
  - $<: 第一个依赖对象名 (上例中为 file1.f)
  - $^: 全部依赖对象名 (上例中为 file1.f file2.f file3.c)
  - $?: 全部比目标新的依赖对象名
  - $*: 用在隐式规则中, 代表不含扩展名的依赖对象
- 隐式规则:

  ```
  prog: file1.o file2.o file3.o
          f77 -O2 -o prog $?
  .c.o:
          cc -O2 -c $*.c
  .f.o:
          f77 -O2 -c $*.f
  ```

Makefile 实例: 【Makefile.dummy】

§1.4 消息传递编程平台 MPI

- 由全世界工业、科研和政府部门联合建立的一个消息传递编程标准, 以便为并行应用程序的设计提供一个高效、统一的编程环境. 它是目前最为通用的并行编程方式, 也是分布式并行系统的主要编程环境.
- MPI 标准中定义了一组函数接口用于进程间的消息传递. 这些函数的具体实现由各计算机厂商或科研部门来完成.
- 除各厂商提供的 MPI 系统外, 一些高校、科研部门也在开发免费的通用 MPI 系统, 其中最著名的有:
  - MPICH (http://www.mcs.anl.gov/mpi/mpich)
  - LAM MPI (http://www.lam-mpi.org/)

  它们均提供源代码, 并支持目前绝大部分的并行计算系统 (包括微机和工作站机群). 实际上许多厂商提供的 MPI 系统是在 MPICH 的基础上优化产生的.
- MPI 的第一个标准 MPI 1.0 于 1994 年推出. 最新的标准为 2.0 版, 于 1998 年推出.

§1.4.1 MPI 程序的编译与运行

- MPI 程序的编译:

```
% mpicc -O2 -o mpiprog mpisrc.c
% mpif77 -O2 -o mpiprog mpisrc1.c mpisrc2.f
```

- MPI 程序的运行:

```
% mpirun -np 4 mpiprog
```

MPI 程序实例: 【/usr/local/mpi/examples/pi3.f】

§1.4.2 利用 MPICH 建立 MPI 程序开发与调试环境

MPICH 是目前最为流行的免费 MPI 系统, 它支持几乎所有的 Unix 及 Windows 9x/NT 环境, 很适合于用来建立自己的 MPI 程序调试环境. 建议在 Linux 环境下安装 MPICH 从而形成一个 MPI 程序的开发调试环境

- mpich.tar.gz 的下载: http://www.mcs.anl.gov/mpi/mpich/ 或 ftp://lsec.cc.ac.cn/pub/mpich, 目前最新版本为 MPICH2-1.0.5.
- 解开 MPICH 源代码:

```
% tar xzpvf mpich.tar.gz
```

  或:

```
% gzip -dc mpich2-1.0.5p4.tar.gz | tar xpvf -
```

  或:

```
% gzip -d mpich2-1.0.5p4.tar.gz
% tar xpvf mpich2-1.0.5p4.tar
```

  展开后, doc 子目录中含有 MPICH 的安装说明及使用说明.
- 配置、编译 MPICH:

```
% cd mpich2-1.0.5p4     (或cd mpich2-1.0.5p4)
% ./configure -prefix=/usr/local/mpi
% make
```

- 在单机上编译运行简单测试程序:

```
% cd examples/basic
% ../../bin/mpdboot
% ../../bin/mpicc -o cpi cpi.c
% ../../bin/mpirun -np 1 cpi
% ../../bin/mpirun -np 2 cpi
```

在运行上述命令之前应该开启系统中的 rsh 服务，并检查文件 ~/.rhosts 或 /etc/hosts.equiv 是否存在并且其中包含本机的主机名, 当文件不存在或没有包含本机主机名的话需要创建一个文件并且 (或) 在其中加入本机主机名. 否则上述最后一条命令将产生 permission denied 错误.

- 以超级用户身份在目录 mpich2-1.0.5p4 下运行:

```
# make install
```

来将 MPICH 安装到目录 /usr/local/mpi 下, 以便其它用户也能使用. 然后可以删除 mpich2-1.0.5p4 目录.

- 使用 C shell 的用户应该在文件 ~/.cshrc 中加入:

```
setenv PATH ${PATH}:/usr/local/mpi/bin
setenv MANPATH ${MANPATH}:/usr/local/mpi/man
```

使用 Bourne shell 的用户应该在文件 ~/.profile 中加入:

```
PATH=${PATH}:/usr/local/mpi/bin
export PATH
MANPATH=${MANPATH}:/usr/local/mpi/man
export MANPATH
```

以便可以直接使用 /usr/local/mpi/bin 中的命令 (mpif77, mpicc, mpirun 等等), 以及用 man 命令来得到 MPI 函数的使用手册.

第二章　MPI 基础知识

§2.1　下载 MPI 标准的 PS 文档

`MPI 1.1/2.0 文档:` 【`http://www.mpi-forum.org/`】

MPI 2.0 的 PostScript 文档中只有 MPI 1.2/2.0 版新增加的内容, 应该结合 MPI 1.1 的文档阅读.

§2.2　一些名词与概念

程序与代码 我们这里说的程序不是指以文件形式存在的源代码、可执行代码等, 而是指为了完成一个计算任务而进行的一次运行过程.

进程 (process) 一个 MPI 并行程序由一组运行在相同或不同计算机/计算结点上的进程或线程构成. 为统一起见, 我们将 MPI 程序中一个独立参与通信的个体称为一个进程. 在 Unix 系统中, MPI 的进程通常是一个 Unix 进程. 在共享内存/消息传递混合编程模式中, 一个 MPI 进程可能代表一组 Unix 线程.

进程组 (process group) 指一个 MPI 程序的全部进程集合的一个有序子集. 进程组中每个进程被赋于一个在该组中唯一的序号 (rank), 用于在该组中标识该进程. 序号的取值范围是 0 到 进程数 – 1.

通信器 (communicator) 通信器 (也有译成通信子的) 是完成进程间通信的基本环境, 它描述了一组可以互相通信的进程以及它们之间的联接关系等信息. MPI 的所有通信必须在某个通信器中进行. 通信器分域内通信器 (intracommunicator) 和域间通信器 (intercommunicator) 两类, 前者用于属于同一进程组的进程间的通信, 后者用于分属两个不同进程组的进程间的通信. 域内通信器由一个进程组和有关该进程组的进程间的拓扑联接关系构成.

MPI 系统在一个 MPI 程序运行时会自动创建两个通信器, 一个称为 `MPI_COMM_WORLD`, 它包含该 MPI 程序中的所有进程, 另一个称为 `MPI_COMM_SELF`, 它指单个进程自己所构成的通信器.

序号 (rank) 序号用来在一个进程组或通信器中标识一个进程. MPI 程序中的进程由进程组/序号或通信器/序号所唯一确定. 序号是相对于进程组或通信器而言的: 同一个进程在不同的进程组或通信器中可以有不同的序号. 进程的序号是在进程组或通信器被创建时赋予的.

MPI 系统提供了一个特殊的进程序号 `MPI_PROC_NULL`, 它代表空进程 (不存在的进程). 与 `MPI_PROC_NULL` 间的通信实际上没有任何作用.

消息 (message) MPI 程序中在进程间传送的数据称为消息. 一个消息由通信器、源地址、目的地址、消息标签和数据构成.

通信 (communication) 通信指在进程之间进行消息的收发、同步等操作.

§2.3　编程模式

SPMD 编程模式 Single Program Multiple Data 的缩写. 指构成一个程序的所有进程运行的是同一份可执行代码. 不同进程根据自己的序号可能执行该代码中的不同分支. 这是 MPI 编程中最常用的编程方式. 用户只需要编写、维护一份源代码.

MPMD 编程模式 Multiple Program Multiple Data 的缩写. 指构成一个程序的不同进程运行不同的可执行代码. 用户需要编写、维护多份源代码.

主/从编程模式 英文为 Master/Slave. 它是 MPMD 编程模式的一个特例, 也是 MPMD 编程模式中最常见的方式. 构成一个程序的进程之一负责所有进程间的协调及任务调度, 该进程称为主进程 (Master), 其余进程称为从进程 (Slave). 通常用户需要维护两份源代码.

§2.4 MPI 函数的一般形式

C: 一般形式为:

```
int MPI_Xxxxxx(...)
```

MPI 的 C 函数名中下划线后第一个字母大写, 其余字母小写. 除 `MPI_Wtime()` 和 `MPI_Wtick()` 外, 所有 MPI 的 C 函数均返回一个整型错误码, 当它等于 `MPI_SUCCESS` (0) 时表示调用成功, 否则表示调用中产生了某类错误.

Fortran 77: 一般形式为:

```
SUBROUTINE MPI_XXXXXX(...., IERR)
```

除 `MPI_WTIME` 和 `MPI_WTICK` 外, Fortran 77 的 MPI 过程全部是 Fortran 77 子程序 (`SUBROUTINE`), 它们与对应的 MPI 的 C 函数同名 (但不区分大小写), 参数表除最后多出一个整型参数 `IERR` 用于返回调用错误码及参数的类型不同外, 也和 MPI 的 C 函数一样.

§2.5 MPI 的原始数据类型

MPI 系统中数据的发送与接收都是基于数据类型进行的. 数据类型可以是 MPI 系统预定义的, 称为原始数据类型, 也可以是用户在原始数据类型的基础上自己定义的数据类型.

MPI 为 Fortran 77 和 C 定义的原始的数据类型在下面两个表格中给出. 除表中列出的外, 有的 MPI 实现可能还提供了更多的原始数据类型, 如 `MPI_INTEGER2`, `MPI_LONG_LONG_INT`, 等等.

§2.5.1 Fortran 77 原始数据类型

MPI 数据类型	对应的 Fortran 77 数据类型
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER*1
MPI_BYTE	
MPI_PACKED	

§2.5.2 C 原始数据类型

MPI 数据类型	对应的 C 数据类型
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_SHORT	short
MPI_LONG	long
MPI_CHAR	char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned
MPI_UNSIGNED_LONG	unsigned long
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

§2.6 MPI 的几个基本函数

本节介绍 MPI 系统的几个基本函数.

§2.6.1 初始化 MPI 系统

C

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

Fortran 77

```
MPI_INIT(IERR)
INTEGER IERR
```

初始化 MPI 系统. 通常它应该是第一个被调用的 MPI 函数. 除 `MPI_Initialized` 外, 其它所有 MPI 函数仅在调用了该函数后才可以被调用.

在 C 接口中, MPI 系统通过 `argc` 和 `argv` 得到命令行参数, 并且会将其中 MPI 系统专用的参数删除而仅留下供用户程序解释的参数.

§2.6.2 检测 MPI 系统是否已经初始化

C

```
int MPI_Initialized(int *flag)
```

Fortran 77

```
MPI_INITIALIZED(FLAG, IERR)
LOGICAL FLAG
INTEGER IERR
```

如果已经调用过 `MPI_Init` 则返回 `flag = true`, 否则返回 `flag = false`. 这是唯一可以在 `MPI_Init` 之前调用的函数.

§2.6.3 得到通信器的进程数及进程在通信器中的序号

C

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

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

Fortran 77

```
MPI_COMM_SIZE(COMM, SIZE, IERR)
INTEGER COMM, SIZE, IERR

MPI_COMM_RANK(COMM, RANK, IERR)
INTEGER COMM, RANK, IERR
```

上述两个函数分别返回指定通信器中的进程数及本进程在该通信器中的序号.

§2.6.4 退出 MPI 系统

C

```
int MPI_Finalize(void)
```

Fortran 77

```
MPI_FINALIZE(IERR)
INTEGER IER
```

退出 MPI 系统. 所有 MPI 进程在正常退出前都必须调用该函数. 它是 MPI 程序中最后一个被调用的 MPI 函数: 调用 `MPI_Finalize` 后不允许再调用任何 MPI 函数.

用户在调用该函数前应该确认所有的 (非阻塞) 通信均已完成.

§2.6.5 异常终止 MPI 程序的执行

C

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

Fortran 77

```
MPI_ABORT(COMM, ERRORCODE, IERR)
INTEGER COMM, ERRORCODE, IERR
```

调用该函数时表明因为出现了某种致命错误而希望异常终止 MPI 程序的执行. MPI 系统会尽量设法终止通信器 `comm` 中的所有进程. 在 Unix 系统环境中, `errcode` 被作为进程的退出码 (exit code) 返回给操作系统.

§2.6.6 查询处理器名称

C

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

Fortran 77

```
MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERR)
CHARACTER*(*) NAME
INTEGER RESULTLEN, IERR
```

该函数在 name 中返回进程所在的处理器的名称. 参数 name 应该提供不少于 MPI_MAX_PROCESSOR_NAME 个字节的存储空间用于存放处理器名称

§2.6.7 获取墙上时间及时钟精度

C

```
double MPI_Wtime(void)

double MPI_Wtick(void)
```

Fortran 77

```
DOUBLE PRECISION FUNCTION MPI_WTIME()

DOUBLE PRECISION MPI_WTICK()
```

MPI_WTIME 返回当前墙上时间, 以从某一固定时刻算起的秒数为单位. 而 MPI_WTICK 则返回 MPI_WTIME 函数的时钟精度, 也是以秒为单位. 例如, 假设 MPI_WTIME 使用的硬件时钟计数器每 1/1000 秒增加 1, 则 MPI_WTICK 的返回值为 10^{-3}, 表示用函数 MPI_WTIME 得到的时间精度为千分之一秒.

在 C 中, 这是唯一两个返回双精度值而非整型错误码的 MPI 接口函数. 在 Fortran 77 中, 这是唯一两个 FUNCTION 形式的接口 (其它 Fortran 77 MPI 的接口一律以 SUBROUTINE 的形式定义).

§2.7 MPI 程序的基本结构

§2.7.1 Fortran 77 程序

```
      PROGRAM MPIPRG
      INCLUDE 'mpif.h'
      INTEGER MYRANK, NPROCS, IERR
*
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_RANK( MPI_COMM_WORLD, MYRANK, IERR)
      CALL MPI_COMM_SIZE( MPI_COMM_WORLD, NPROCS, IERR)
      ... ...
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```

程序实例: 【02-mpi.f】

§2.7.2 C 程序

```
#include "mpi.h"
... ...
int main(int argc, char *argv[])
{
    int myrank, nprocs;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    ... ...
    MPI_Finalize();
}
```

程序实例:【02-mpi.c】

第三章　点对点通信

点对点通信指在一对进程之间进行的消息收发操作: 一个进程发送消息, 另一个进程接收消息.

§3.1　标准阻塞型点对点通信函数

§3.1.1　标准阻塞发送

C

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

Fortran 77

```
MPI_SEND(BUFF, COUNT, DATATYPE, DEST, TAG, COMM, IERR)
<type>  BUFF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERR
```

tag 的取值范围为 0 - MPI_TAG_UB.

dest 的取值范围为 0 - np-1 (np 为通信器 comm 中的进程数) 或 MPI_PROC_NULL.

count 是指定数据类型的个数, 而不是字节数.

§3.1.2　阻塞接收

C

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

Fortran 77

```
MPI_RECV(BUFF, COUNT, DATATYPE, SOURCE, TAG, COMM,
+       STATUS, IERR)
<type>  BUFF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, IERR
INTEGER STATUS(MPI_STATUS_SIZE)
```

tag 的取值范围为 0 - MPI_TAG_UB, 或 MPI_ANY_TAG.

source 的取值范围为 0 - np-1 (np 为通信器 comm 中的进程数), 或 MPI_ANY_SOURCE, 或 MPI_PROC_NULL.

count 给出接收缓冲区的大小 (指定数据类型的个数), 它是接收数据长度的上界. 具体接收到的数据长度可通过调用 MPI_Get_count 函数得到.

§3.1.2.1 status 中的内容

status 返回有关接收到的消息的信息, 它的结构如下:

- 在 C 中 status 是一个结构, 它包含下面三个用户可以使用的域:

```
typedef struct {
    ... ...
    int MPI_SOURCE;        消息源地址
    int MPI_TAG;           消息标签
    int MPI_ERROR;         错误码
    ... ...
} MPI_Status;
```

- 在 Fortran 77 中 status 各个元素的含义如下:

```
STATUS(MPI_SOURCE)         消息源地址
STATUS(MPI_TAG)            消息标签
STATUS(MPI_ERROR)          错误码
```

§3.1.2.2 查询接收到的消息长度

C

```
int MPI_Get_count(MPI_Status status,
        MPI_Datatype datatype, int *count)
```

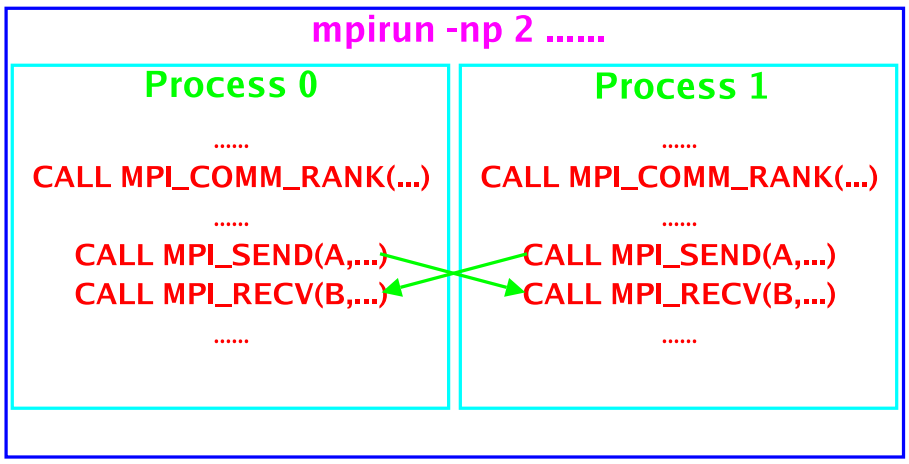
Fortran 77

```
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERR)
INTEGER DATATYPE, COUNT, IERR
INTEGER STATUS(MPI_STATUS_SIZE)
```

该函数在 count 中返回消息的长度 (数据类型个数).

§3.1.3 阻塞型消息传递实例

例 3.1 阻塞型消息传递: 【03-ex1.f】



例 3.2 消息收发顺序导致的程序死锁：【03-ex2.f】

例 3.3 长消息导致的程序死锁：【03-ex3.f】

例 3.4 *0* 号进程先发，其它进程先收：【03-ex4.f】：通信过程是串行的.

例 3.5 偶数号的进程先发，奇数号的进程先收：【03-ex5.f】：通信过程是并发的. 当 NPROCS=1 时仍有可能发生死锁.

如果将接收及求和中的数组 B 改成 A, 会得到什么结果?

§3.1.4 其它一些阻塞型消息传递函数

§3.1.4.1 发送与接收组合进行

MPI_SENDRECV 函数将一次发送调用和一次接收调用合并进行. 它使得 MPI 程序更为简洁. 更重要的是, MPI 的实现通常能够保证使用 MPI_SENDRECV 函数的程序不会出现上节例子中出现的由于消息收发配对不好而引起的程序死锁.

C

```
int MPI_Sendrecv(void *sendbuff, int sendcount,
        MPI_Datatype sendtype, int dest, int sendtag,
        void *recvbuff, int recvcount,
        MPI_Datatype recvtype, int source, int recvtag,
        MPI_Comm comm, MPI_Status *status)
```

Fortran 77

```
 MPI_SENDRECV(SENDBUFF, SENDCOUNT, SENDTYPE,
+        DEST, SENDTAG,
+        RECVBUFF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG,
+        COMM, STATUS, IERR)
 <type>  SENDBUFF(*), RECVBUFF(*)
 INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG,
+        RECVCOUNT, RECVTYPE, SOURCE, RECVTAG,
+        COMM, IERR
 INTEGER STATUS(MPI_STATUS_SIZE)
```

例 3.6 用 *MPI_SENDRECV* 替换 *MPI_SEND* 和 *MPI_RECV:*【03-ex6.f】

§3.1.4.2 发送与接收组合进行, 收发使用同一缓冲区

C

```
int MPI_Sendrecv_replace(void *buff, int count,
        MPI_Datatype datatype,
        int dest, int sendtag,
        int source, int recvtag,
        MPI_Comm comm, MPI_Status *status)
```

Fortran 77

```
 MPI_SENDRECV_REPLACE(BUFF, COUNT, DATATYPE,
+        DEST, SENDTAG, SOURCE, RECVTAG,
+        COMM, STATUS, IERR)
 <type>  BUFF(*)
 INTEGER COUNT, DATATYPE, DEST, SENDTAG,
+        SOURCE, RECVTAG, COMM, IERR
 INTEGER STATUS(MPI_STATUS_SIZE)
```

§3.2 消息发送模式

MPI 共有四种模式的消息发送函数. 这些函数具有完全一样的入口参数, 但它们发送消息的方式或对接收方的状态要求不同.

标准模式 (standard mode) 由 MPI 系统来决定是否将消息拷贝至一个缓冲区然后立即返回 (此时消息的发送由 MPI 系统在后台进行), 还是等待将数据发送出去后再返回. 大部分 MPI 系统预留了一定大小的缓冲区, 当发送的消息长度小于缓冲区大小时会将消息缓冲然后立即返回, 否则则当部分或全部消息发送完成后才返回. 标准模式发送被称为是非局部的, 因为它的完成可能需要与接收方联络. 标准模式的阻塞发送函数为 `MPI_Send`.

缓冲模式 (buffered mode) MPI 系统将消息拷贝至一个用户提供的缓冲区然后立即返回, 消息的发送由 MPI 系统在后台进行. 用户必须确保所提供的缓冲区大小足以容下采用缓冲模式发送的消息. 缓冲模式发送操作被称为是局部的, 因为它不需要与接收方联络即可立即完成 (返回). 缓冲模式的阻塞发送函数为 `MPI_Bsend`.

同步模式 (synchronous mode) 实际上就是在标准模式的基础上还要求确认接收方已经开始接收数据后函数调用才返回. 显然, 同步模式的发送是非局部的. 同步模式的阻塞发送函数为 `MPI_Ssend`.

就绪模式 (ready mode) 调用就绪模式发送时必须确保接收方已经处于就绪状态 (正在等待接收该消息), 否则该调用将产生一个错误. 该模式设立的目的是在一些以同步方式工作的并行系统上由于发送时可以假设接收方已经在接收而减少一些消息发送的开销. 如果一个使用就绪模式的 MPI 程序是正确的, 则将其中所有就绪模式的消息发送改为标准模式后也应该是正确的. 就绪模式的阻塞发送函数为 `MPI_Rsend`.

后三种模式的消息发送函数名分别在标准模式消息发送函数 `MPI_Send` 中加上 `B`, `S` 和 `R` 得到, 并且四个函数具有完全一样的参数.

§3.2.1 阻塞型缓冲模式消息发送函数

C

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

Fortran 77

```
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERR)
<type>  BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERR
```

提交阻塞型缓冲模式的消息发送, 各参数的含义与 MPI_Send 同. 该函数将 buf 中的数据拷贝到一个用户指定的缓冲区中然后立即返回. 实际发送由 MPI 系统在后台进行.

用户在调用该函数前必须调用 MPI_Buffer_attach 来为它提供一个发送缓冲区:

C

```
int MPI_Buffer_attach( void* buffer, int size)
```

Fortran 77

```
MPI_BUFFER_ATTACH( BUFFER, SIZE, IERR)
<type>  BUFFER(*)
INTEGER SIZE, IERR
```

将 buffer 提交为供 MPI_Bsend 使用的缓冲区, 该缓冲区的长度为 size 字节. MPI 只允许提交一个供 MPI_Bsend 使用的缓冲区, 用户应该保证该缓冲区足够容下所有可能同时用 MPI_Bsend 发送的消息 (每个消息所需的缓冲区长度通常等于消息中数据的长度加上一个常量, 用户可以调用函数 MPI_Type_size (见 31 页) 来查询一个消息实际需要的缓冲区长度).

MPI_Buffer_detach 撤消用 MPI_Buffer_attach 提交的缓冲区. 在调用 MPI_Buffer_detach 前用户不应该修改缓冲区中的内容, 以免破坏正在发送的消息. 该函数的返回亦表明所有缓冲模式的消息发送均已完成.

C

```
int MPI_Buffer_detach( void *buffer_addr, int *size)
```

Fortran 77

```
MPI_BUFFER_DETACH( BUFFER_ADDR, SIZE, IERR)
<type>  BUFFER_ADDR(*)
INTEGER SIZE, IERR
```

注意, 上述函数中 buffer_addr 和 size 均为输出参数. 其中 buffer_addr 返回所撤消的缓冲区的地址, 而 size 则返回所撤消的缓冲区长度.

§3.3 阻塞型与非阻塞型函数

阻塞型 (blocking) 阻塞型函数需要等待指定操作的实际完成, 或至少所涉及的数据已被 MPI 系统安全地备份后才返回. 如 MPI_Send 和 MPI_Recv 都是阻塞型的. MPI_Send 调用返回时表明数据已经发出或被 MPI 系统复制, 随后对发送缓冲区的修改不会改变所发送的数据. 而 MPI_Recv 返回时, 则表明数据接收已经完成. 阻塞型函数的操作是非局部的, 它的完成可能需要与其它进程

进行通信. 阻塞型函数使用不当很容易引起程序的死锁.

非阻塞型 (non-blocking) 非阻塞型函数的调用总是立即返回, 而实际操作则由 MPI 系统在后台进行. 用户必须随后调用其它函数来等待或查询操作的完成情况. 在操作完成之前对相关数据区的操作是不安全的, 因为随时可能与正在后台进行的操作发生冲突. 非阻塞型函数调用是局部的, 因为它的完成不需要与其它进程进行通信. 在有些并行系统上, 通过非阻塞型函数的使用可以实现计算与通信的重叠进行.

§3.4 非阻塞型点对点通信函数

§3.4.1 非阻塞发送

C

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

Fortran 77

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

该函数递交一个消息发送请求, 要求 MPI 系统在后台完成消息的发送. `MPI_Isend` 函数为该发送创建一个请求并将请求的句柄通过 `request` 变量返回给 MPI 进程, 供随后查询/等待消息发送的完成用.

与阻塞消息发送一样, 非阻塞消息发送也有四种模式: 标准模式 (`MPI_Isend`)、缓冲模式 (`MPI_Ibsend`)、同步模式 (`MPI_Issend`) 和就绪模式 (`MPI_Irsend`), 后三种模式较少使用, 我们不在此介绍.

§3.4.2 非阻塞接收

C

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

Fortran 77

```
      MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
     +        REQUEST, IERR)
      <type>  BUF(*)
      INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST,
     +        IERR
```

该函数递交一个消息接收请求, 要求 MPI 系统在后台完成消息的接收. `MPI_Isend` 函数为该接收创建一个请求并将请求的句柄通过 `request` 变量返回给 MPI 进程, 供随后查询/等待消息接收的完

成用.

§3.4.3 通信请求的完成与检测

§3.4.3.1 等待、检测一个通信请求的完成

C

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

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

Fortran 77

```
MPI_WAIT(REQUEST, STATUS, IERR)
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERR

MPI_TEST(REQUEST, FLAG, STATUS, IERR)
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERR
LOGICAL FLAG
```

MPI_Wait 是阻塞型函数, 它必须等待通信的完成才返回. MPI_Test 是与之相对应的非阻塞函数.

MPI_Wait 等待指定的通信请求完成然后返回. 成功返回时, status 中包含关于所完成的通信的信息, 相应的通信请求被释放, request 被置成 MPI_REQUEST_NULL.

MPI_Test 检测指定的通信请求, 不论通信是否完成都立即返回. 如果通信已经完成则返回 flag = true, status 中包含关于所完成的通信的信息, 相应的通信请求被释放, request 被置成 MPI_REQUEST_NULL. 如果通信尚未完成则返回 flag = false.

对于接收请求, status 返回的内容与 MPI_Recv 返回的一样. 对于发送请求, status 的返回的值不确定.

§3.4.3.2 等待、检测一组通信请求中某一个的完成

C

```
int MPI_Waitany(int count,
        MPI_Request *array_of_requests,
        int *index, MPI_Status *status)

int MPI_Testany(int count,
        MPI_Request *array_of_requests,
        int *index, int *flag, MPI_Status *status)
```

Fortran 77

```
MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS,
+        IERR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, IERR
INTEGER STATUS(MPI_STATUS_SIZE)

MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG,
```

```
     +        STATUS, IERR)
      INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, IERR
      INTEGER STATUS(MPI_STATUS_SIZE)
      LOGICAL FLAG
```

等待或检测 array_of_requests 给出的通信请求中任何一个的完成. 成功返回时, index 中包含所完成的通信请求在数组 array_of_requests 中的位置. 其它参数的含义与 MPI_Wait 和 MPI_Test 同.

§3.4.3.3 等待、检测一组通信请求的全部完成

C

```
int MPI_Waitall(int count,
        MPI_Request *array_of_requests,
        MPI_Status *array_of_statuses)

int MPI_Testall(int count,
        MPI_Request *array_of_requests,
        int *flag, MPI_Status *array_of_statuses)
```

Fortran 77

```
      MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES,
     +        IERR)
      INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERR
      INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*)

      MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG,
     +        ARRAY_OF_STATUSES, IERR)
      INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERR
      INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*)
      LOGICAL FLAG
```

当函数返回值等于 MPI_ERR_IN_STATUS 时表明部分通信请求的处理出错, 此时用户应检查 array_of_statuses 的每个元素中 MPI_ERROR 域的值来得到这些通信请求的错误码.

§3.4.3.4 等待、检测一组通信请求中一部分的完成

C

```
int MPI_Waitsome(int incount,
        MPI_Request *array_of_requests,
        int outcount, int *array_of_indices,
        MPI_Status *array_of_statuses)

int MPI_Testsome(int incount,
        MPI_Request *array_of_requests,
        int outcount, int *array_of_indices,
        MPI_Status *array_of_statuses)
```

Fortran 77

```
      MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,
     +        ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERR)
      INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT,
     +        ARRAY_OF_INDICES(*), IERR
      INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*)

      MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,
     +        ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERR)
      INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT,
     +        ARRAY_OF_INDICES(*), IERR
      INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*)
```

outcount 中返回的是成功完成的通信请求个数. array_of_indices 的前 outcount 个元素给出已完成的通信请求在数组 array_of_requests 中的位置. 当指定的通信请求中至少有一个已完成时或发生错误时 MPI_Waitsome 才返回, 而 MPI_Testsome 当所有指定的通信请求都未完成时在 outcount 中返回 0.

当函数返回值等于 MPI_ERR_IN_STATUS 时表明部分通信请求的处理出错, 此时用户应检查 array_of_statuses 的每个元素中 MPI_ERROR 域的值来得到这些通信请求的错误码.

§3.4.4 通信请求的释放

C

```
int MPI_Request_free(MPI_Request *request)
```

Fortran 77

```
MPI_REQUEST_FREE(REQUEST, IERR)
INTEGER REQUEST, IERR
```

MPI_Request_free 释放指定的通信请求 (及所占用的内存). 如果与该通信请求相关联的通信尚未完成, 则 MPI_Request_free 会等待通信的完成. 函数成功返回后 request 的值将被置成 MPI_REQUEST_NULL.

例 3.7 非阻塞消息传递：【03-ex7.f】

§3.5 消息探测与通信请求的取消

§3.5.1 消息探测

C

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

Fortran 77

```
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERR)
INTEGER SOURCE, TAG, COMM, IERR
```

```
INTEGER STATUS(MPI_STATUS_SIZE)
```

例: 在接收未知长度的消息时, 可先用 MPI_Probe 和 MPI_Get_count 得到消息的长度.

MPI_Probe 属于阻塞型函数, 它等待直到一个符合条件的消息到达后才返回. 对应的非阻塞函数为 MPI_Iprobe, 它不论是否有符合条件的消息都立即返回: 如果探测到符合条件的消息 flag = true, 否则 flag = false.

C

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

Fortran 77

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

§3.5.2 通信请求的取消

C

```
int MPI_Cancel(MPI_Request *request)

int MPI_Test_cancelled(MPI_Status *status, int *flag)
```

Fortran 77

```
MPI_CANCEL(REQUEST, IERR)
INTEGER REQUEST, IERR

MPI_TEST_CANCELLED(STATUS, FLAG, IERR)
INTEGER STATUS(MPI_STATUS_SIZE), IERR
LOGICAL FLAG
```

MPI_Cancel 用于取消一个尚未完成的通信请求, 它在 MPI 系统中设置一个取消该通信请求的标志然后立即返回, 具体的取消操作由 MPI 系统在后台完成. 调用 MPI_Cancel 后, 仍需调用 MPI_Wait, MPI_Test, 或 MPI_Request_free 等函数来完成并释放该通信请求.

MPI_Test_cancelled 可用来检测一个通信请求是否已被取消, 返回 flag = true 如果该通信请求已被取消.

§3.6 点对点通信函数汇总

函数类型	通信模式	阻塞型	非阻塞型
消息发送函数	标准模式	MPI_Send	MPI_Isend
	缓冲模式	MPI_Bsend	MPI_Ibsend
	同步模式	MPI_Ssend	MPI_Issend
	就绪模式	MPI_Rsend	MPI_Irsend
消息接收函数		MPI_Recv	MPI_Irecv
消息检测函数		MPI_Probe	MPI_Iprobe
等待/查询函数		MPI_Wait	MPI_Test
		MPI_Waitall	MPI_Testall
		MPI_Waitany	MPI_Testany
		MPI_Waitsome	MPI_Testsome
释放通信请求		MPI_Request_free	
取消通信			MPI_Cancel
			MPI_Test_cancelled

§3.7 持久通信请求

持久通信请求 (persistent communication request) 可用于以完全相同的方式 (相同的通信器、收发缓冲区、数据类型与长度、源/目的地址和消息标签) 重复收发的消息. 目的是减少处理消息的开销及简化 MPI 程序.

§3.7.1 创建持久消息发送请求

C

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

Fortran 77

```
 MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
+        REQUEST, IERR)
 <type>  BUF(*)
 INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM,
+        REQUEST, IERR
```

创建一个持久消息发送请求, 该函数并不开始实际消息的发送, 而只是创建一个请求句柄并返回给用户程序, 留待以后实际消息发送时用.

`MPI_Send_init` 对应于标准模式的非阻塞发送. 相应地还有三个函数 `MPI_Bsend_init`, `MPI_Ssend_init` 和 `MPI_Rsend_init`, 分别对应于缓冲模式、同步模式和就绪模式的非阻塞发送, 我们不在此介绍.

§3.7.2 创建持久消息接收请求

C

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

Fortran 77

```
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
        REQUEST, IERR)
<type>  BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST,
+       IERR
```

创建一个持久消息接收请求, 该函数并不开始实际消息的接收, 而只是创建一个请求句柄并返回给用户程序, 留待以后实际消息接收时用.

§3.7.3 开始基于持久通信请求的通信

C

```
int MPI_Start(MPI_Request *request)

int MPI_Startall(int count,
        MPI_Request *array_of_requests)
```

Fortran 77

```
MPI_START(REQUEST, IERR)
INTEGER REQUEST, IERR

MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERR
```

每次调用 `MPI_Start` 相当于调用一次与 `MPI_xxxx_init` 相对应的非阻塞型通信函数 (如 `MPI_Isend`, `MPI_Irecv` 等). `MPI_Startall` 的作用与 `MPI_Start` 类似, 但它可一次启动多个通信. 用一个 `MPI_xxxx_init` 创建的持久通信请求可反复调用 `MPI_Start` 或 `MPI_Startall` 来完成多次通信.

§3.7.4 持久通信请求的完成与释放

与普通非阻塞型通信函数一样, 通过持久通信请求启动的通信也需要调用 §3.4.3, §3.5.2 等节中介绍的函数来等待或检测通信的完成情况或取消通信, 或用 `MPI_Request_free` 来释放一个持久通信请求.

例 3.8 使用持久通信函数: 【`03-ex8.f`】

第四章　数据类型

MPI 的消息收发函数只能处理连续存储的同一类型的数据. 当需要在一个消息中发送或接收具有复杂结构的数据时, 可以通过定义数据类型 (datatype) 来实现. 数据类型是 MPI 的一个重要特征, 它的使用可有效地减少消息传递的次数, 增大通信粒度, 并且在收/发消息时避免或减少数据在内存中的拷贝、复制.

§4.1　与数据类型有关的一些定义

§4.1.1　数据类型定义

一个 MPI 数据类型由两个 n 元序列构成, n 为正整数. 第一个序列包含一组数据类型, 称为*类型序列* (type signature):

$$\texttt{Typesig} = \{\texttt{type}_0, \texttt{type}_1, \ldots, \texttt{type}_{n-1}\}.$$

第二个序列包含一组整数位移, 称为*位移序列* (type displacements):

$$\texttt{Typedisp} = \{\texttt{disp}_0, \texttt{disp}_1, \ldots, \texttt{disp}_{n-1}\}.$$

注意, 位移序列中位移总是以字节为单位计算的.

构成类型序列的数据类型称为基本数据类型, 它们可以是原始数据类型, 也可以是任何已定义的数据类型, 因此 MPI 的数据类型是嵌套定义的. 为了以后叙述方便, 我们称非原始数据类型为*复合数据类型*.

类型序列刻划了数据的类型特征. 位移序列则刻划了数据的位置特征. 类型序列和位移序列的元素一一配对构成的序列

$$\texttt{Typemap} = \{(\texttt{type}_0, \texttt{disp}_0), (\texttt{type}_1, \texttt{disp}_1), \ldots, (\texttt{type}_{n-1}, \texttt{disp}_{n-1})\}.$$

称为*类型图* (type map). 假设数据缓冲区的起始地址为 `buff`, 则由上述类型图所定义的数据类型包含 n 块数据, 第 i 块数据的地址为 $\texttt{buff} + \texttt{disp}_i$, 类型为 $\texttt{type}_i$, $i = 0, 1, \ldots, n-1$.

MPI 的原始数据类型的类型图可以写成 {(类型, 0)}. 如 `MPI_INTEGER` 的类型图为 {(`INTEGER`, 0)}.

位移序列中的位移不必是单调上升的, 表明数据类型中的数据块不要求按顺序排放. 位移也可以是负的, 即数据类型中的数据可以位于缓冲区起始地址之前.

例 4.1 *假设数据类型 `TYPE` 类型图为*

$$\{(\texttt{REAL}, 4), (\texttt{REAL}, 12), (\texttt{REAL}, 0)\}$$

则语句:

```
REAL A(100)
... ...
CALL MPI_SEND(A, 1, TYPE, ...)
```

将发送 `A(2)`, `A(4)`, `A(1)`.

例 4.2 *假设数据类型 TYPE 类型图为*

$$\{(\texttt{REAL}, -4), (\texttt{REAL}, 0), (\texttt{REAL}, 4)\}$$

则语句:

```
REAL A(3)
... ...
CALL MPI_SEND(A(2), 1, TYPE, ...)
```

将发送 A(1), A(2), A(3).

§4.1.2 数据类型的大小

数据类型的大小 (size) 指该数据类型中包含的数据长度 (字节数), 它等于类型序列中所有基本数据类型的大小之和. 数据类型的大小就是消息传递时需要发送或接收的数据长度. 假设数据类型 type 的类型图为:

$$\{(\texttt{type}_0, \texttt{disp}_0), (\texttt{type}_1, \texttt{disp}_1), \ldots, (\texttt{type}_{n-1}, \texttt{disp}_{n-1})\}$$

则该数据类型的大小为:

$$\texttt{sizeof}(\texttt{type}) = \sum_{i=0}^{n-1} \texttt{sizeof}(\texttt{type}_i)$$

§4.1.3 数据类型的下界、上界与域

数据类型的下界 (lower bound) 定义为数据的最小位移. 数据类型的上界 (upper bound) 定义为数据的最大位移加 1, 再加上一个使得数据类型满足操作系统地址对界要求 (alignment) 的修正量 ε. 数据类型的域 (extent) 定义为上界与下界之差:

$$\begin{aligned} \texttt{lb}(\texttt{type}) &= \min_i \{\texttt{disp}_i\} \\ \texttt{ub}(\texttt{type}) &= \max_i \{\texttt{disp}_i + \texttt{sizeof}(\texttt{type}_i)\} + \varepsilon \\ \texttt{extent}(\texttt{type}) &= \texttt{ub}(\texttt{type}) - \texttt{lb}(\texttt{type}) \end{aligned}$$

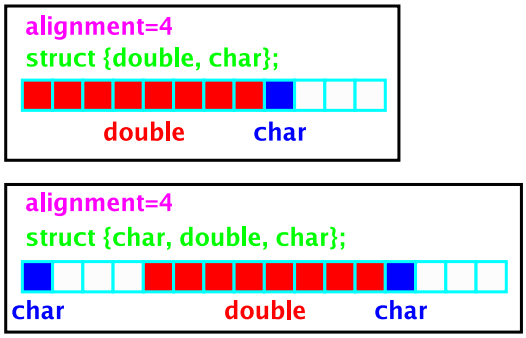
其中, 地址对界修正量 ε 是使得 extent 能被该数据类型的对界量整除的最小非负整数.

一个数据类型的对界量定义如下: 原始数据类型的对界量由编译系统决定, 而复合数据类型的对界量则定义为它的所有基本数据类型的对界量的最大值. 地址对界要求指一个数据类型在内存中的 (字节) 地址必须是它的对界量的整数倍.

例 4.3 *C 语言中的地址对界*

```
#include <stdio.h>

typedef struct {
    double d;
    char   c;
} CS;

typedef struct {
    char   c1;
    double d;
```

```
    char   c2;
} CS1;

main()
{
  CS  a;
  CS1 b;

  printf("sizeof(CS)=%d\n", sizeof(CS));
  printf("offset(a.d)=%d, offset(a.c)=%d\n",
(char *)&a.d - (char *)&a, (char *)&a.c - (char *)&a);

  printf( "sizeof(CS1)=%d\n", sizeof(CS1));
  printf( "offset(b.c1)=%d, offset(b.d)=%d, offset(b.c2)=%d\n",
(char *)&b.c1 - (char *)&b, (char *)&b.d  - (char *)&b,
(char *)&b.c2 - (char *)&b);
}
```

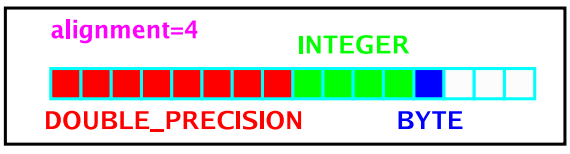


例 4.4 假设 MPI_DOUBLE_PRECISION 和 MPI_INTEGER 的对界量均为 4, MPI_BYTE 的对界量为 1, 则类型图:

$$\{(\texttt{MPI\_DOUBLE\_PRECISION}, 0), (\texttt{MPI\_INTEGER}, 8), (\texttt{MPI\_BYTE}, 12)\}$$

的对界量为 4, 下界为 0, 上界为 16, 域为 16, $\varepsilon = 3$.

数据类型的下界、上界、域和大小:【04-ex1.f】



§4.1.4 MPI_LB 和 MPI_UB

MPI 系统提供了两个特殊的数据类型 MPI_LB 和 MPI_UB, 称为伪数据类型 (pseudo datatype). 它们的大小是 0, 并且当它们出现在一个数据类型的类型图中时对该数据类型的实际数据内容不起任何

作用. 它们的作用是让用户可以人工指定一个数据类型的上下界. MPI 规定: 如果一个数据类型的基本类型中含有 MPI_LB, 则它的下界定义为:

$$\texttt{lb}(\texttt{type}) = \min_i\{\texttt{disp}_i \,|\, \texttt{type}_i = \texttt{MPI\_LB}\};$$

类似地, 如果一个数据类型的基本类型中含有 MPI_UB, 则它的上界定义为:

$$\texttt{ub}(\texttt{type}) = \max_i\{\texttt{disp}_i \,|\, \texttt{type}_i = \texttt{MPI\_UB}\}.$$

例 4.5 类型图

$$\begin{aligned}\{&(\texttt{MPI\_LB}, -4),\\ &(\texttt{MPI\_UB}, 20),\\ &(\texttt{MPI\_DOUBLE\_PRECISION}, 0),\\ &(\texttt{MPI\_INTEGER}, 8),\\ &(\texttt{MPI\_BYTE}, 12)\}\end{aligned}$$

的下界为 -4, 上界为 20, 域为 24.

MPI_LB 和 MPI_UB 的作用: 【04-ex2.f】

§4.1.5 数据类型查询函数

下面四个函数分别返回指定数据类型的大小、域和上下界:

C

```
int MPI_Type_size(MPI_Datatype datatype, int *size)

int MPI_Type_extent(MPI_Datatype datatype,
        MPI_Aint *extent)

int MPI_Type_ub(MPI_Datatype datatype,
        MPI_Aint *displacement)

int MPI_Type_lb(MPI_Datatype datatype,
        MPI_Aint *displacement)
```

Fortran 77

```
MPI_TYPE_SIZE(DATATYPE, SIZE, IERR)
INTEGER DATATYPE, SIZE, IERR

MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERR)
INTEGER DATATYPE, EXTENT, IERR

MPI_TYPE_UB(DATATYPE, DISPLACEMENT, IERR)
INTEGER DATATYPE, DISPLACEMENT, IERR

MPI_TYPE_LB(DATATYPE, DISPLACEMENT, IERR)
INTEGER DATATYPE, DISPLACEMENT, IERR
```

§4.2 数据类型创建函数

§4.2.1 MPI_Type_contiguous

C

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

Fortran 77

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERR)
INTEGER COUNT, OLDTYPE, NEWTYPE, IERR
```

新数据类型 newtype 由 count 个老数据类型 oldtype 按域 (extent) 连续存放构成.

例 4.6 *Fortran 77 语句:*

```
CALL MPI_SEND(BUFF, COUNT, TYPE, ...)
```

与:

```
CALL MPI_TYPE_CONTIGUOUS(COUNT, TYPE, NEWTYPE, IERR)
CALL MPI_TYPE_COMMIT(NEWTYPE, IERR)
CALL MPI_SEND(BUFF, 1, NEWTYPE, ...)
```

是等效的.

§4.2.2 MPI_Type_vector

C

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

Fortran 77

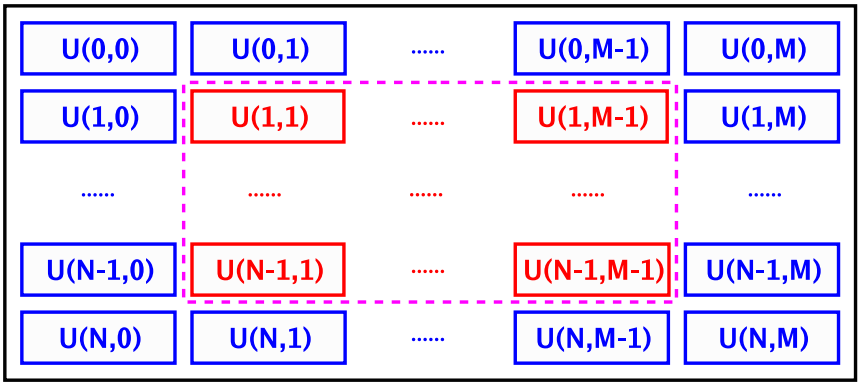
```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,
+       NEWTYPE, IERR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
+       IERR
```

新数据类型 newtype 由 count 个数据块构成, 每个数据块由 blocklength 个连续存放的 oldtype 构成, 相邻两个数据块的位移相差 stride × extent(oldtype) 个字节.

例 4.7

```
DOUBLE PRECISION U(0:N, 0:M)
... ...
CALL MPI_TYPE_VECTOR(M-1, N-1, N+1,
+       MPI_DOUBLE_PRECISION, NEWTYPE, IERR)
CALL MPI_TYPE_COMMIT(NEWTYPE, IERR)
CALL MPI_SEND(U(1,1), 1, NEWTYPE, ...)
```

将发送 U(1:N-1,1:M-1).



思考: 如何发送 U 的一列? 如何发送 U 的一行? 如何发送 U 的对角线?

§4.2.3 MPI_Type_hvector

C

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

Fortran 77

```
MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,
+        NEWTYPE, IERR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
+        IERR
```

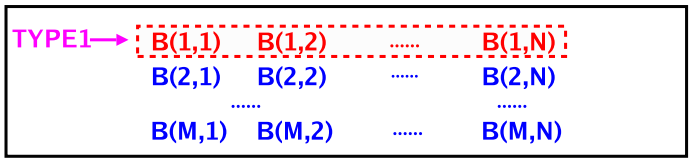
新数据类型 newtype 由 count 个数据块构成, 每个数据块由 blocklength 个连续存放的 oldtype 构成, 相邻两个数据块的位移相差 stride 个字节.

函数 MPI_Type_hvector 与 MPI_Type_vector 的唯一区别在于: stride 在 MPI_Type_vector 中以 oldtype 的域为单位, 而在 MPI_Type_hvector 中以字节为单位.

例 4.8

```
DOUBLE PRECISION A(N,M), B(M,N)
INTEGER TYPE1, TYPE2, STATUS(MPI_STATUS_SIZE)
... ...
CALL MPI_TYPE_VECTOR(N, 1, M, MPI_DOUBLE_PRECISION,
+                    TYPE1, IERR)
CALL MPI_TYPE_EXTENT(MPI_DOUBLE_PRECISION, I, IERR)
CALL MPI_TYPE_HVECTOR(M, 1, I, TYPE1, TYPE2, IERR)
CALL MPI_TYPE_FREE(TYPE1, IERR)
CALL MPI_TYPE_COMMIT(TYPE2, IERR)
CALL MPI_SENDRECV(A, N*M, MPI_DOUBLE_PRECISION, 0, 111,
+                 B, 1, TYPE2, 0, 111,
+                 MPI_COMM_SELF, STATUS, IERR)
CALL MPI_TYPE_FREE(TYPE2, IERR)
```

将 A 的转置拷贝到 B 中. 程序实例: 【04-ex3.f】



§4.2.4 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 77

```
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
+        ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE,
+        IERR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
+        ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE,
+        IERR
```

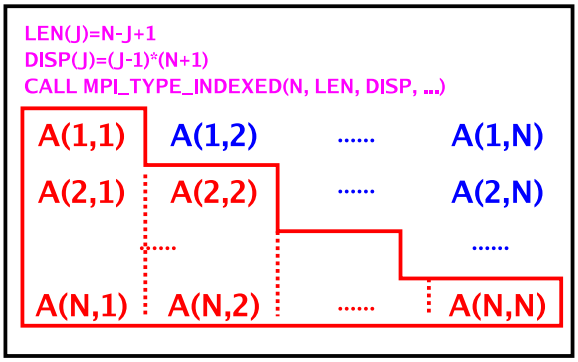
新数据类型 newtype 由 count 个数据块构成. 第 i 个数据块包含 array_of_blocklengths(i) 个连续存放的 oldtype, 字节位移为 array_of_displacements(i) × extent(oldtype).

MPI_Type_indexed 与 MPI_Type_vector 的区别在于每个数据块的长度可以不同, 数据块间也可以不等距.

例 4.9

```
REAL A(N,N)
INTEGER LEN(N), DISP(N), TYPE
... ...
LEN(1)=N
DISP(1)=0
DO J=2,N
   LEN(J)=LEN(J-1)-1
   DISP(J)=(J-1)*(N+1)
ENDDO
CALL MPI_TYPE_INDEXED(N,LEN,DISP,MPI_REAL,TYPE,IERR)
CALL MPI_TYPE_COMMIT(TYPE, IERR)
CALL MPI_SEND(A, 1, TYPE, ...)
... ...
```

发送矩阵 A 的下三角部分.



§4.2.5 MPI_Type_hindexed

C

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

Fortran 77

```
MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
+        ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
+        ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE,
+        IERR
```

新数据类型 newtype 由 count 个数据块构成. 第 i 个数据块包含 array_of_blocklengths(i) 个连续存放的 oldtype, 字节位移为 array_of_displacements(i).

函数 MPI_Type_hindexed 与函数 MPI_Type_indexed 的唯一区别在于 MPI_Type_hindexed 中 array_of_displacements 以字节为单位.

§4.2.6 MPI_Type_struct

C

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

Fortran 77

```
MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
+        ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES,
+        NEWTYPE, IERR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
+        ARRAY_OF_DISPLACEMENTS(*), ARRAY_OF_TYPES(*),
```

```
+        NEWTYPE, IERR
```

新数据类型 newtype 由 count 个数据块构成. 第 i 个数据块包含 array_of_blocklengths(i) 个连续存放、类型为 array_of_types(i) 的数据, 字节位移为 array_of_displacements(i).

函数 MPI_Type_struct 与 MPI_Type_hindexed 的区别在于各数据块可以由不同的数据类型构成.

§4.2.7 地址函数 MPI_Address

函数 MPI_Address 返回指定变量的“绝对”地址, 可用于在 Fortran 77 中计算变量的位移量.

C

```
int MPI_Address(void *buff, MPI_Aint *address)
```

Fortran 77

```
MPI_ADDRESS(BUFF, ADDRESS, IERR)
<type>  BUFF(*)
INTEGER ADDRESS, IERR
```

C 语言中通过指针可以实现对地址的任何操作, 而 Fortran 77 语言中则难以用通用的方式直接使用一个地址. 为便于 Fortran 77 代码使用 MPI_ADDRESS 函数返回的地址, MPI 定义了一个变量 MPI_BOTTOM, 它相当于绝对地址 0, 因此, 调用:

```
CALL MPI_ADDRESS(BUFF, ADDRESS, IERR)
```

后, MPI_BOTTOM(ADDRESS) 与 BUFF 代表着同一个内存地址.

例 4.10

```
REAL A(N), B(N)
INTEGER TYPE, IA, IB
......
CALL MPI_ADDRESS(A, IA, IERR)
CALL MPI_ADDRESS(B, IB, IERR)
CALL MPI_TYPE_HVECTOR(2, N, IB-IA, MPI_REAL, TYPE, IERR)
......
CALL MPI_SEND(A, 1, TYPE, ...)
... ...
```

同时发送两个同类型数组 A 和 B.

例 4.11

```
PARAMETER (N=1024, M=512, L=256)
REAL    A(N)
INTEGER B(M)
COMPLEX C(L)
INTEGER LEN(3), DISP(3), TYPE(3), NEWTYPE, IA, IB, IC
DATA TYPE /MPI_REAL, MPI_INTEGER, MPI_COMPLEX/,
+     LEN  /N, M, L/
... ...
CALL MPI_ADDRESS(A, IA, IERR)
CALL MPI_ADDRESS(B, IB, IERR)
CALL MPI_ADDRESS(C, IC, IERR)
DISPS(1)=0
```

```
DISPS(2)=IB-IA
DISPS(3)=IC-IA
CALL MPI_TYPE_STRUCT(3, LEN, DISP, TYPE, NEWTYPE, IER)
... ...
CALL MPI_SEND(A, 1, NEWTYPE, ...)
... ...
```

同时发送三个不同类型数组 A, B 和 C.

例 4.12 例 *4.11* 中最后几行可以改成:

```
... ...
DISPS(1)=IA
DISPS(2)=IB
DISPS(3)=IC
CALL MPI_TYPE_STRUCT(3, LEN, DISP, TYPE, NEWTYPE, IER)
... ...
CALL MPI_SEND(MPI_BOTTOM, 1, NEWTYPE, ...)
... ...
```

§4.3 数据类型的使用

除原始数据类型外, 其它数据类型在首次用于消息传递之前必须通过 MPI_Type_commit 函数进行提交. 一个 (非原始) 数据类型在不再需要时应该调用 MPI_Type_free 函数进行释放, 以便释放它所占用的系统资源.

§4.3.1 数据类型的提交

C

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

Fortran 77

```
MPI_TYPE_COMMIT(DATATYPE, IERR)
INTEGER DATATYPE, IERR
```

一个数据类型在被提交后就可以和 MPI 原始数据类型完全一样地在消息传递中使用.

如果一个数据类型仅仅用于创建其它数据类型的中间步骤而并不直接在消息传递中使用, 则不必将它提交, 一旦基于它的其它数据类型创建完毕即可立即将它释放.

§4.3.2 数据类型的释放

C

```
int MPI_Type_free(MPI_Datatype *datatype)
```

Fortran 77

```
MPI_TYPE_FREE(DATATYPE, IERR)
INTEGER DATATYPE, IERR
```

MPI_Type_free 释放指定的数据类型. 函数返回后, datatype 将被置成 MPI_DATATYPE_NULL. 正在进行的使用该数据类型的通信将会正常完成. 一个数据类型的释放对在它的基础上创建的其它数据类型不产生影响.

例 4.13

```
      REAL U(N,M), V(N,M)
      INTEGER IU, IV, CTYPE, LTYPE
      ... ...
      CALL MPI_ADDRESS(U, IU, IERR)
      CALL MPI_ADDRESS(V, IV, IERR)
      CALL MPI_TYPE_VECTOR(M, 1, N, MPI_REAL, CTYPE, IERR)
      CALL MPI_TYPE_HVECTOR(2, 1, IV-IU, CTYPE, LTYPE, IERR)
      CALL MPI_TYPE_FREE(CTYPE, IERR)
      CALL MPI_TYPE_HVECTOR(2, N, IV-IU, MPI_REAL, CTYPE,IERR)
      ... ...
!     发送 U(1:N,1),V(1:N,1)
      CALL MPI_SEND(U(1,1), 1, CTYPE, ...)
!     发送 U(1,1:M),V(1,1:M)
      CALL MPI_SEND(U(1,1), 1, LTYPE, ...)
!     发送 U(1:N,M),V(1:N,M)
      CALL MPI_SEND(U(1,M), 1, CTYPE, ...)
!     发送 U(N,1:M),V(N,1:M)
      CALL MPI_SEND(U(N,1), 1, LTYPE, ...)
      ... ...
```

例 4.14

```
      REAL U(N,M), V(N,M)
      INTEGER IU, IV, EX, CTYPE, LTYPE
      INTEGER LENS(3), DISPS(3), TYPES(3)
      DATA LENS/1, 1, 1/, TYPES/MPI_REAL, MPI_REAL, MPI_UB/
      ... ...
      CALL MPI_ADDRESS(U, IU, IERR)
      CALL MPI_ADDRESS(V, IV, IERR)
      CALL MPI_TYPE_EXTENT(MPI_REAL, EX, IERR)
      DISPS(1) = 0
      DISPS(2) = IV - IU
      DISPS(3) = EX
      CALL MPI_TYPE_STRUCT(3, LENS, DISPS, TYPES, CTYPE, IERR)
      DISPS(3) = EX*N
      CALL MPI_TYPE_STRUCT(3, LENS, DISPS, TYPES, LTYPE, IERR)
      ... ...
!     发送 U(1:N,1),V(1:N,1)
      CALL MPI_SEND(U(1,1), N, CTYPE, ...)
!     发送 U(1,1:M),V(1,1:M)
      CALL MPI_SEND(U(1,1), M, LTYPE, ...)
!     发送 U(1:N,M),V(1:N,M)
      CALL MPI_SEND(U(1,M), N, CTYPE, ...)
!     发送 U(N,1:M),V(N,1:M)
      CALL MPI_SEND(U(N,1), M, LTYPE, ...)
      ... ...
```

§4.3.3 MPI_Get_elements

函数 MPI_Get_elements 与 MPI_Get_count 类似, 但它返回的是消息中所包含的 MPI 原始数据类型的个数. MPI_Get_elements 返回的 count 值如果不等于 MPI_UNDEFINED 的话, 则必然是 MPI_Get_count 返回的 count 值的倍数.

C

```
int MPI_Get_elements(MPI_Status *status,
        MPI_Datatype datatype, int *count)
```

Fortran 77

```
MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERR)
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERR
```

例 4.15 *MPI_Get_elements:* 【04-ex4.f】

§4.4 数据的打包与拆包

在 MPI 中, 通过使用特殊数据类型 MPI_PACKED, 用户可以以类似于 PVM 中的方式将不同的数据进行打包后再一次发送出去, 接收方在收到消息后再进行拆包.

§4.4.1 数据打包

C

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

Fortran 77

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

该函数将缓冲区 inbuf 中的 incount 个类型为 datatype 的数据进行打包. 打包后的数据放在缓冲区 outbuf 中. outsize 给出的是 outbuf 的总长度 (字节数, 供函数检查打包缓冲区是否越界用). comm 是发送打包数据将使用的通信器.

position 是打包缓冲区中的位移, 第一次调用 MPI_Pack 前用户程序应将 position 设为 0, 随后 MPI_Pack 将自动修改它, 使得它总是指向打包缓冲区中尚未使用部分的起始位置. 每次调用 MPI_Pack 后的 position 实际上就是已打包的数据的总长度.

§4.4.2 数据拆包

C

```
int MPI_Unpack(void *inbuf, int insize, int *position,
```

```
        void *outbuf, int outcount,
        MPI_Datatype datatype, MPI_Comm comm)
```

Fortran 77

```
 MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
+        DATATYPE, COMM, IERR)
 <type>  INBUF(*), OUTBUF(*)
 INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM,
+        IERR
```

MPI_Unpack 进行数据拆包操作, 它正好是 MPI_Pack 的逆操作: 它从 inbuf 中拆包 outcount 个类型为 datatype 的数据到 outbuf 中. 函数中各参数的含义与 MPI_Pack 类似, 只不过这里的 inbuf 和 insize 对应于 MPI_Pack 中的 outbuf 和 outsize, 而 outbuf 和 outcount 则对应于 MPI_Pack 中的 inbuf 和 incount.

§4.4.3 得到打包后的数据大小

由于 MPI 打包时会在用户数据中加入一些附加信息, 因此打包后的数据大小不等于用户数据的大小. 函数 MPI_Pack_size 返回一个数据打包后的大小, 可被用来计算所需的打包缓冲区长度.

C

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

Fortran 77

```
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERR)
INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERR
```

size 返回 incount 个类型为 datatype 的数据为了在通信器 comm 中进行发送或接收而打包后的数据长度.

例 4.16

```
INTEGER*4 BUFF(64)
INTEGER N, M
REAL    A(5), B(5)
INTEGER POSITION
... ...
IF ( MYRANK .EQ. 0 ) THEN
   POSITION = 0
   CALL MPI_PACK(N, 1, MPI_INTEGER, BUFF, 256, POSITION,
+                MPI_COMM_WORLD, IERR)
   CALL MPI_PACK(M, 1, MPI_INTEGER, BUFF, 256, POSITION,
+                MPI_COMM_WORLD, IERR)
   CALL MPI_PACK(A, 5, MPI_REAL,    BUFF, 256, POSITION,
+                MPI_COMM_WORLD, IERR)
   CALL MPI_PACK(B, 5, MPI_REAL,    BUFF, 256, POSITION,
+                MPI_COMM_WORLD, IERR)
   CALL MPI_SEND(BUFF, 256, MPI_PACKED, 1, 111,
+                MPI_COMM_WORLD, IERR)
ELSE IF ( MYRANK .EQ. 1 ) THEN
   CALL MPI_RECV(BUFF, 256, MPI_PACKED, 0, 111,
```

```
+                 MPI_COMM_WORLD, IERR)
    POSITION = 0
    CALL MPI_UNPACK(BUFF, 256, POSITION, N, 1,
+                   MPI_INTEGER, MPI_COMM_WORLD, IERR)
    CALL MPI_UNPACK(BUFF, 256, POSITION, M, 1,
+                   MPI_INTEGER, MPI_COMM_WORLD, IERR)
    CALL MPI_UNPACK(BUFF, 256, POSITION, A, 5,
+                   MPI_REAL,    MPI_COMM_WORLD, IERR)
    CALL MPI_UNPACK(BUFF, 256, POSITION, B, 5,
+                   MPI_REAL,    MPI_COMM_WORLD, IERR)
 ENDIF
```

§4.5 MPI 1.1 中位移与数据大小的限制

本章介绍的许多函数中 (如 `MPI_Address`, `MPI_Type_hvector` 等), C 与 Fortran 77 对一些参数使用了不同的变量类型: 它们在 C 函数中的类型为 `MPI_Aint`, 而在 Fortran 77 中的类型为 `MPI_INTEGER`.

`MPI_Aint` 是 MPI 定义的一个用于与位移及数据大小有关的操作的 C 变量类型, 因为在 64 位系统中用 `int` 存储地址或数据大小可能不够, 在这些系统上 `MPI_Aint` 通常被定义成 `long`, 而在 32 位系统中 `MPI_Aint` 则通常定义为 `int`. 用 `MPI_Aint` 来说明存储地址的整型变量便于保证 MPI 代码中的地址操作在 32 位与 64 位操作系统间的可移植性.

MPI 1.1 没有为 Fortran 77 提供相对应的变量类型, 而是统一使用 `INTEGER`. 因此如果一个 Fortran 77 程序使用了这些函数的话, 它在 32 与 64 位操作系统间的可移植性可能会受到影响, 并且用 Fortran 77 接口处理超过 2GB(或 4GB) 的位移、数据大小等参数也会有困难.

此外, 一些 C 接口的定义也有问题, 如 `MPI_Type_size` 中 `size` 说明为 `int *`, 这就隐含限制了它返回的大小不能超过 2GB.

MPI 2 中定义了一组新的函数, 如 `MPI_Get_address`, `MPI_Type_create_hvector` 等等, 目的就是澄清部分这类问题并且为 C 和 Fortran (90) 提供统一的接口参数. 有兴趣者可参看 MPI 2.0 标准第 2.6 节.

第五章　聚合通信 (Collective Communications)

本章介绍 MPI 提供的聚合通信函数, 它们包括障碍同步 (barrier synchronization)、广播 (broadcast)、数据收集 (gather) 与散发 (scatter)、归约 (reduction) 等等. 这些函数均要求属于同一进程组 (通信器) 的进程共同参与, 协同完成.

聚合通信函数根据数据的流向可分为一对多 (一个进程对多个进程, 如广播、数据散发)、多对一 (多个进程对一个进程, 如数据收集、归约) 和多对多 (多个进程对多个进程) 三种类型的操作. 在一对多和多对一操作中, 有一个进程扮演着特殊的角色, 称为该操作的根进程 (root).

§5.1　障碍同步

C

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

Fortran 77

```
MPI_BARRIER(COMM, IERR)
INTEGER COMM, IERR
```

该函数用于进程间的同步. 一个进程调用该函数后将等待直到通信器 comm 中的所有进程都调用了该函数才返回.

§5.2　广播

C

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

Fortran 77

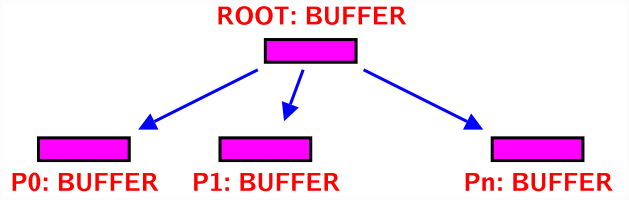
```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERR)
<type>  BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERR
```

通信器 comm 中进程号为 root 的进程 (根进程) 将自己 buffer 中的内容同时发送给通信器中所有其它进程.

直观上说, 假设 NPROCS 为通信器 comm 中的进程数, MYRANK 为进程在通信器 comm 中的进程号, 则 MPI_Bcast 相当于:

```
IF ( MYRANK .EQ. ROOT ) THEN
   DO I=0, NPROCS-1
      IF ( I .NE. ROOT ) THEN
         CALL MPI_SEND(BUFFER, COUNT, DATATYPE, I, ...
      ENDIF
   ENDDO
```

```
ELSE
   CALL MPI_RECV(BUFFER, COUNT, DATATYPE, ROOT, ...
ENDIF
```



§5.3 数据收集

数据收集指各个进程 (包括根进程) 将自己的一块数据发送给根进程, 根进程将这些数据合并成一个大的数据块.

§5.3.1 收集相同长度数据块 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)
```

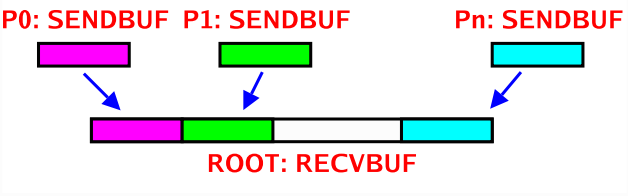
Fortran 77

```
 MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
+        RECVCOUNT, RECVTYPE, ROOT, COMM, IERR)
 <type>  SENDBUF(*), RECVBUF(*)
 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
+        COMM, IERR
```

所有进程 (包括根进程) 将 sendbuf 中的数据发送给根进程, 根进程将这些数据按进程号的顺序依次接收到 recvbuf 中. 发送和接收的数据类型与长度必须相配, 即发送和接收使用的数据类型必须具有相同的类型序列. 参数 recvbuf, recvcount 和 recvtype 仅对根进程有意义.

假设 NPROCS 为通信器中的进程数, MYRANK 为进程号, 则 MPI_Gather 相当于:

```
CALL MPI_SEND(SENDBUF, SENDCOUNT, SENDTYPE, ROOT, ...)
IF ( MYRANK .EQ. ROOT ) THEN
   DO I=0, NPROCS-1
      CALL MPI_RECV( RECVBUF + I*RECVCOUNT*extent(RECVTYPE),
+                    RECVCOUNT, RECVTYPE, I, ...)
   ENDDO
ENDIF
```



§5.3.2 收集不同长度数据块 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)
```

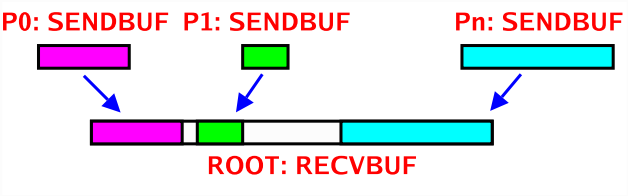
Fortran 77

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

与 MPI_Gather 类似, 但允许每个进程发送的数据长度不同, 并且根进程接收时也不一定将它们连续存放. recvbuf, recvtype, recvcounts 和 displs 仅对根进程有意义. 数组 recvcounts 和 displs 的元素个数等于进程数, 分别给出从每个进程接收的数据长度和位移 (以 recvtype 为单位).

假设 NPROCS 为通信器中的进程数, MYRANK 为进程号, 则 MPI_Gatherv 相当于:

```
INTEGER DISPLS(0:NPROCS-1), RECVCOUNTS(0:NPROCS-1)
... ...
CALL MPI_SEND(SENDBUF, SENDCOUNT, SENDTYPE, ROOT, ...)
IF ( MYRANK .EQ. ROOT ) THEN
   DO I=0, NPROCS-1
      CALL MPI_RECV( RECVBUF + DISPLS(I)*extent(RECVTYPE),
+                    RECVCOUNTS(I), RECVTYPE, I, ...)
   ENDDO
ENDIF
```



§5.3.3 全收集 MPI_Allgather

C

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

Fortran 77

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

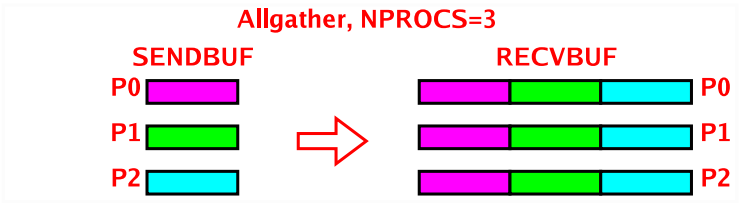
MPI_Allgather 与 MPI_Gather 类似, 区别是所有进程同时将数据收集到 recvbuf 中, 因此称为数据全收集.

MPI_Allgather 等价于依次以每个进程为根进程调用 NPROCS 次普通数据收集函数 MPI_Gather:

```
 DO I=0, NPROCS-1
    CALL MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
+                   RECVCOUNT, RECVTYPE, I, COMM, IERR)
 ENDDO
```

也可以认为 MPI_Allgather 相当于以任一进程为根进程调用一次普通收集, 紧接着再对收集到的数据进行一次广播, 例如:

```
 ROOT=0
 CALL MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
+                RECVCOUNT, RECVTYPE, ROOT, COMM, IERR)
 CALL MPI_BCAST(RECVBUF, RECVCOUNT*NPROCS, RECVTYPE, ROOT,
+               COMM, IERR)
```



§5.3.4 不同长度数据块的全收集 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 77

```
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
```

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

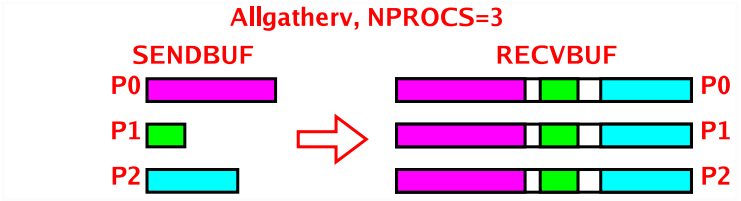
MPI_Allgatherv 用于不同长度数据块的全收集. 它的参数与 MPI_Gatherv 类似.

MPI_Allgatherv 等价于依次以每个进程为根进程调用 NPROCS 次 MPI_Gatherv:

```
 DO I=0, NPROCS-1
    CALL MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
+           RECVCOUNTS, DISPLS, RECVTYPE, I, COMM, IERR)
 ENDDO
```

也可以认为 MPI_Allgatherv 相当于以任一进程为根进程调用一次普通收集, 紧接着再对收集到的数据进行一次广播, 例如:

```
 ROOT=0
 CALL MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
+        RECVCOUNTS, DISPLS, RECVTYPE, ROOT, COMM, IERR)
 CALL MPI_TYPE_INDEXED(NPROCS, RECVCOUNTS, DISPLS, RECVTYPE,
+        NEWTYPE, IERR)
 CALL MPI_TYPE_COMMIT(NEWTYPE, IERR)
 CALL MPI_BCAST(RECVBUF, 1, NEWTYPE, ROOT, COMM, IERR)
 CALL MPI_TYPE_FREE(NEWTYPE, IERR)
```



§5.4 数据散发

数据散发指根进程将一个大的数据块分成小块分别散发给各个进程 (包括根进程自己). 它是数据收集的逆操作.

§5.4.1 散发相同长度数据块 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)
```

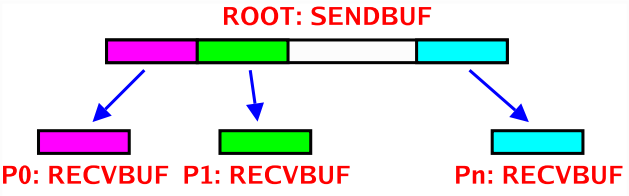
Fortran 77

```
      MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
     +        RECVCOUNT, RECVTYPE, ROOT, COMM, IERR)
      <type>  SENDBUF(*), RECVBUF(*)
      INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
     +        COMM, IERR
```

根进程 root 的 sendbuf 中包含有 NPROCS 个连续存放的数据块, 每个数据块包含 sendcount 个类型为 sendtype 的数据, NPROCS 为通信器中的进程数. 根进程将这些数据块按进程的序号依次分发给各个进程 (包括根进程自己). 参数 sendbuf, sendcount 和 sendtype 仅对根进程有意义.

假设 MYRANK 为进程号, 则 MPI_Scatter 相当于:

```
IF ( MYRANK .EQ. ROOT ) THEN
   DO I=0, NPROCS-1
      CALL MPI_SEND( SENDBUF + I*SENDCOUNT*extent(SENDTYPE),
+                    SENDCOUNT, SENDTYPE, I, ...)
   ENDDO
ENDIF
CALL MPI_RECV(RECVBUF, RECVCOUNT, RECVTYPE, ROOT, ...)
```



§5.4.2 散发不同长度数据块 MPI_Scatterv

C

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

Fortran 77

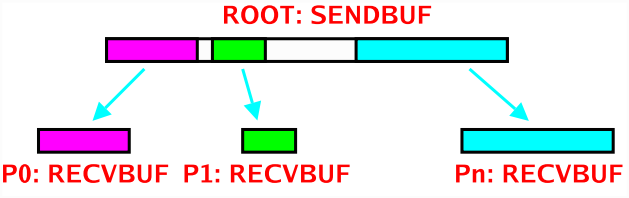
```
      MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE,
     +        RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERR)
      <type>  SENDBUF(*), RECVBUF(*)
      INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT,
     +        RECVTYPE, ROOT, COMM, IERR
```

与 MPI_Scatter 类似, 但允许发送的每个数据块的长度不同并且在 sendbuf 中不一定连续存放. sendbuf, sendtype, sendcounts 和 displs 仅对根进程有意义. 数组 sendcounts 和 displs 的元素个数等于进程数, 它们分别给出发送给每个进程的数据长度和位移 (以 sendtype 为单位).

假设 NPROCS 为通信器中的进程数, MYRANK 为进程号, 则 MPI_Scatterv 相当于:

```
INTEGER DISPLS(0:NPROCS-1), SENDCOUNTS(0:NPROCS-1)
```

```
... ...
IF ( MYRANK .EQ. ROOT ) THEN
   DO I=0, NPROCS-1
      CALL MPI_SEND( SENDBUF + DISPLS(I)*extent(SENDTYPE),
+                    SENDCOUNTS(I), SENDTYPE, I, ...)
   ENDDO
ENDIF
CALL MPI_RECV(RECVBUF, RECVCOUNT, RECVTYPE, ROOT, ...)
```



§5.5 全部进程对全部进程的数据散发收集

每个进程散发自己的一个数据块, 并且收集拼装所有进程散发过来的数据块. 我们称该操作为数据的“全散发收集”. 它既可以被认为是数据全收集的扩展, 也可以被认为是数据散发的扩展.

§5.5.1 相同数据长度的全收集散发 MPI_Alltoall

C

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

Fortran 77

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

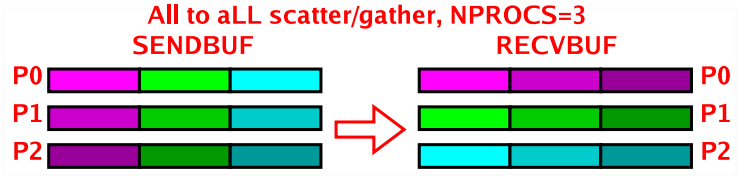
第 i 个进程将 sendbuf 中的第 j 块数据发送至第 j 个进程的 recvbuf 的第 i 个位置, $i, j = 0, \ldots,$ NPROCS-1 (NPROCS 为进程数). sendbuf 和 recvbuf 均由 NPROCS 个连续存放的数据块构成, 但它们每个数据块的长度/类型分别为 sendcount/sendtype 和 recvcount/recvtype.

该操作相当于将数据/进程进行一次转置. 例如, 假设一个二维数组按行分块存储在各进程中, 则调用该函数可很容易地将它变成按列分块存储在各进程中.

假设 MYRANK 为进程号, 则 MPI_Alltoall 相当于:

```
DO I=0, NPROCS-1
   CALL MPI_SEND( SENDBUF + I*SENDCOUNT*extent(SENDTYPE),
```

```
     +                  SENDCOUNT, SENDTYPE, I, ...)
      ENDDO
      DO I=0, NPROCS-1
         CALL MPI_RECV( RECVBUF + I*RECVCOUNT*extent(RECVTYPE),
     +                  RECVCOUNT, RECVTYPE, I, ...)
      ENDDO
```



§5.5.2 不同数据长度的全收集散发 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)
```

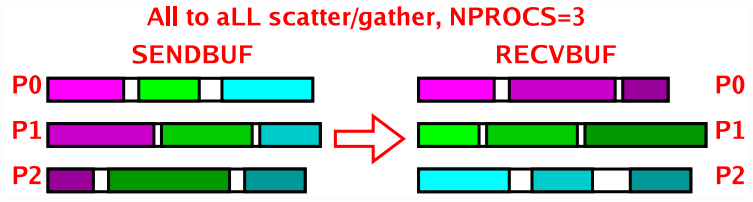
Fortran 77

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

与 MPI_Alltoall 类似, 但每个数据块的长度可以不等, 并且不要求连续存放. 各个参数的含义很容易从 MPI_Alltoall, MPI_Scatterv 和 MPI_Gatherv 中参数的含义得出.

假设 MYRANK 为进程号, NPROCS 为进程数, 则 MPI_Alltoallv 相当于:

```
      DO I=0, NPROCS-1
         CALL MPI_SEND( SENDBUF + SDISPLS(I)*extent(SENDTYPE),
     +                  SENDCOUNTS(I), SENDTYPE, I, ...)
      ENDDO
      DO I=0, NPROCS-1
         CALL MPI_RECV( RECVBUF + RDISPLS(I)*extent(RECVTYPE),
     +                  RECVCOUNTS(I), RECVTYPE, I, ...)
      ENDDO
```



§5.6 归约

假设一个通信器中有 p 个进程, 每个进程均有一个 n 个元素的数组. 设 $\{a_{k,i}, k = 1, \ldots, n\}$ 为第 i 个进程中的数组, $i = 0, \ldots, p-1$. 又设 $\oplus$ 为关于这些数组元素的二目运算, 则相应的归约操作 (reduction) 结果定义为数组 $\{\mathrm{res}_k, k = 1, \ldots, n\}$, 其中:

$$\mathrm{res}_k = a_{k,0} \oplus a_{k,1} \cdots \oplus a_{k,p-1},$$

MPI 的归约函数要求运算 $\oplus$ 满足结合律, 但可以不满足交换律. 运算可以是 MPI 预定义的, 也可以是用户自行定义的.

§5.6.1 归约函数 MPI_Reduce

C

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

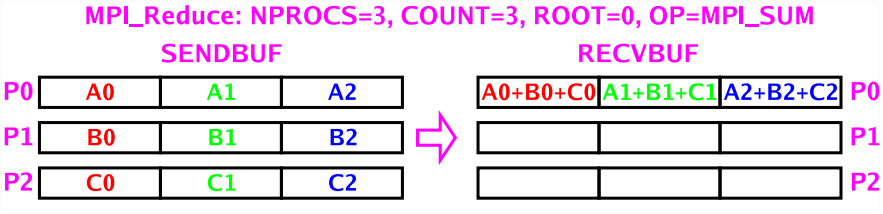
Fortran 77

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

设进程数为 NPROCS, 则 MPI_Reduce 相当于在根进程 (root) 中计算:

```
DO K=1, COUNT
   RECVBUF(K) = SENDBUF(K) of process 0
   DO I=1, NPROCS-1
      RECVBUF(K) = RECVBUF(K) op ( SENDBUF(K) of process I )
   ENDDO
ENDDO
```

函数中除 op 参数外, 其它参数的含义是显而易见的. op 指定归约使用的运算, op 在 C 中的类型为 MPI_Op, 而在 Fortran 77 中则是一个句柄. op 可以是 MPI 预定义的, 也可以是用户定义的. 我们将在 §5.6.5 中 (54 页) 介绍如何自定义归约中的运算.



下表给出 MPI 预定义的归约运算以及它们对数据类型的要求. 表中“字节”指 MPI_BYTE.

MPI 预定义的运算及对数据类型的限制

MPI 运算符	含义	允许的数据类型	
		C	Fortran 77
MPI_MAX	求最大	整型, 实型	整型, 实型
MPI_MIN	求最小	整型, 实型	整型, 实型
MPI_SUM	求和	整型, 实型, 复型	整型, 实型, 复型
MPI_PROD	求积	整型, 实型, 复型	整型, 实型, 复型
MPI_LAND	逻辑与	整型	逻辑型
MPI_BAND	二进制按位与	整型, 字节	整型, 字节
MPI_LOR	逻辑或	整型	逻辑型
MPI_BOR	二进制按位或	整型, 字节	整型, 字节
MPI_LXOR	逻辑异或	整型	逻辑型
MPI_BXOR	二进制按位异或	整型, 字节	整型, 字节
MPI_MAXLOC	最大值及位置	*	*
MPI_MINLOC	最小值及位置	*	*

MPI_MINLOC 和 MPI_MAXLOC 是两个特殊的运算, 它们要求由数对 (连续存放的两个数) 构成的一类特殊数据类型. MPI 为它们定义了下面一些数据类型:

C

$$\texttt{MPI\_FLOAT\_INT} = \{\texttt{float}, \texttt{int}\}$$

$$\texttt{MPI\_DOUBLE\_INT} = \{\texttt{double}, \texttt{int}\}$$

$$\texttt{MPI\_LONG\_INT} = \{\texttt{long}, \texttt{int}\}$$

$$\texttt{MPI\_2INT} = \{\texttt{int}, \texttt{int}\}$$

$$\texttt{MPI\_SHORT\_INT} = \{\texttt{short}, \texttt{int}\}$$

$$\texttt{MPI\_LONG\_DOUBLE\_INT} = \{\texttt{long double}, \texttt{int}\}$$

Fortran 77

$$\texttt{MPI\_2REAL} = \{\texttt{REAL}, \texttt{REAL}\}$$

$$\texttt{MPI\_2DOUBLE\_PRECISION} = \{\texttt{DOUBLE PRECISION}, \texttt{DOUBLE PRECISION}\}$$

$$\texttt{MPI\_2INTEGER} = \{\texttt{INTEGER}, \texttt{INTEGER}\}$$

设 $x=(u,i)$, $y=(v,j)$, 则 `MPI_MINLOC`$(x,y)=(w,k)$, 其中:

$$w=\min(u,v),\qquad k=\begin{cases} i & \text{if } u<v \\ \min(i,j) & \text{if } u=v \\ j & \text{if } u>v \end{cases}$$

设 $x=(u,i)$, $y=(v,j)$, 则 `MPI_MAXLOC`$(x,y)=(w,k)$, 其中:

$$w=\max(u,v),\qquad k=\begin{cases} i & \text{if } u>v \\ \min(i,j) & \text{if } u=v \\ j & \text{if } u<v \end{cases}$$

§5.6.2 全归约 MPI_Allreduce

C

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

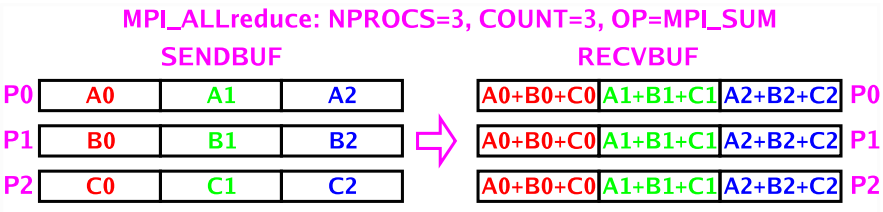
Fortran 77

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

全归约函数与普通归约函数的操作类似, 但所有进程将同时获得归约运算的结果. `MPI_Allreduce` 除了比 `MPI_Reduce` 少了一个 `root` 参数外, 其它参数及含义与后者一样.

`MPI_Allreduce` 相当于在 `MPI_Reduce` 后马上再将结果进行一次广播, 因此它等价于:

```
 ROOT=0
 CALL MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT,
+        COMM, IERR)
 CALL MPI_BCAST(RECVBUF, COUNT, DATATYPE, ROOT, COMM, IERR)
```



§5.6.3 归约散发 MPI_Reduce_scatter

C

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

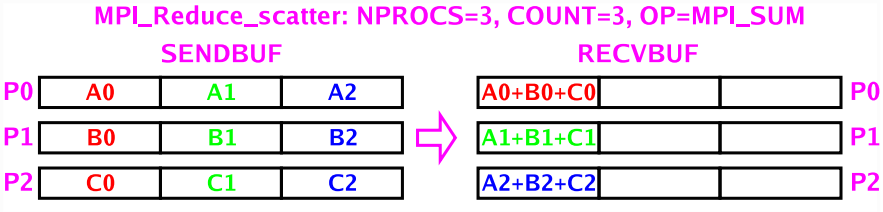
Fortran 77

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

归约散发函数首先进行一次 COUNT $= \sum_{i=0}^{\text{NPROCS}-1}$ recvcounts(i) 的归约操作, 然后再对归约结果进行散发操作, 散发给第 i 个进程的数据块长度为 recvcounts(i). 其余参数的含义与 MPI_Reduce 一样.

设 NPROCS 为进程数, MYRANK 为进程号, 则 MPI_Reduce_scatter 相当于:

```
 INTEGER ROOT,COUNT,DISPLS(0:NPROCS-1),RECVCOUNTS(0:NPROCS-1)
 <type>  TMPBUF(*)

 COUNT = RECVCOUNTS(0)
 DISPLS(0) = 0
 DO I=1, NPROCS-1
    COUNT = COUNT + RECVCOUNTS(I)
    DISPLS(I) = DISPLS(I-1) + RECVCOUNTS(I-1)
 ENDDO
 ROOT=0
 CALL MPI_REDUCE(SENDBUF, TMPBUF, COUNT, DATATYPE, OP, ROOT,
+        COMM, IERR)
 CALL MPI_SCATTERV(TMPBUF, RECVCOUNTS, DISPLS, DATATYPE,
+        RECVBUF, RECVCOUNTS(MYRANK), DATATYPE, ROOT, COMM,
+        IERR)
```



§5.6.4 前缀归约 MPI_Scan

C

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

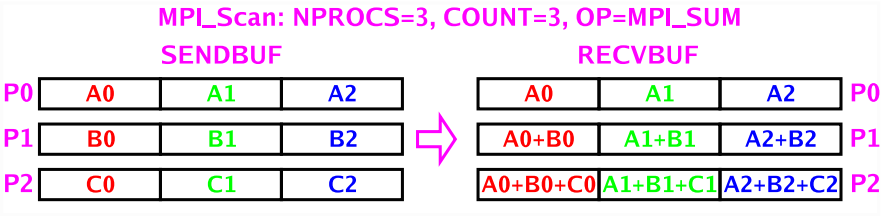
Fortran 77

```
 MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM,
+         IERR)
 <type>  SENDBUF(*), RECVBUF(*)
 INTEGER COUNT, DATATYPE, OP, COMM, IERR
```

前缀归约, 或前缀扫描, 与归约操作类似, 但各处理器依次得到部分归约结果. 确切地说, 操作结束后第 i 个处理器的 recvbuf 中将包含前 i 个处理器的归约运算结果. 各参数的含义与 MPI_Allreduce 基本相同.

设进程号为 MYRANK, 则 MPI_Scan 相当于每个进程分别计算:

```
DO K=1, COUNT
   RECVBUF(K) = SENDBUF(K) of process 0
   DO I=1, MYRANK
      RECVBUF(K) = RECVBUF(K) op ( SENDBUF(K) of process I )
   ENDDO
ENDDO
```



§5.6.5 归约与前缀归约中用户自定义的运算

除 MPI 提供的运算外, 用户可以自行定义归约和前缀归约中使用的运算.

C

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

Fortran 77

```
MPI_OP_CREATE(FUNC, COMMUTE, OP, IERR)
EXTERNAL FUNC
LOGICAL COMMUTE
INTEGER OP, IERR
```

MPI_Op_create 创建 (定义) 一个新的运算. 参数中 func 是用户提供的用于完成该运算的外部函数名, commute 用来指明所定义的运算是否满足交换律 (commute=true 表示满足). C 接口中 op 返回所创建的运算指针, Fortran 77 接口则返回所创建的运算句柄. 一个运算创建后便和 MPI 预定义的运算一样, 可以用在本节介绍的所有归约和前缀归约函数中.

负责完成二目运算的外部函数 func 应该具有如下形式的接口:

C

```
void func(void *invec, void *inoutvec, int *len,
        MPI_Datatype *datatype)
```

Fortran 77

```
FUNCTION FUNC(INVEC, INOUTVEC, LEN, DATATYPE)
<type>  INVEC(LEN), INOUTVEC(LEN)
INTEGER LEN, DATATYPE
```

进入函数 func 时, invec 和 inoutvec 包含参与运算的操作数 (operand). 函数返回时 inoutvec 中应该包含运算的结果. len 给出 invec 和 inoutvec 中包含的元素个数 (相当于归约函数中的 count).

datatype 给出操作数的数据类型 (即归约函数中的 datatype). 直观地, 函数 func 必须完成如下操作:

```
DO I=1,LEN
   INOUTVEC(I) = INVEC(I) op INOUTVEC(I)
ENDDO
```

当一个用户定义的运算不再需要时, 可以调用 MPI_Op_free 将其释放, 以便释放它所占用的系统资源.

C

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

Fortran 77

```
MPI_OP_FREE(OP, IERR)
INTEGER OP, IERR
```

§5.7 两个程序实例

§5.7.1 π 值计算

例 5.1 这是 *MPICH* 中名为 pi3.f 的程序实例. 该程序利用积分:

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

计算 π 值. 令 $f(x) = 4/(1+x^2)$. 将区间 $[0,1]$ 分成 n 等分, 并令 $x_i = (i-0.5)/n$, 则:

$$\pi \approx \frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

假设并行程序中共有 P 个进程参与计算, 则第 k 个进程负责计算:

$$\sum_{1\le i\le n,(i-1)\bmod P=k} f(x_i)$$

π 值计算:【/usr/local/mpi/examples/pi3.f】

§5.7.2 Jacobi 迭代求解二维 Poisson 方程

§5.7.2.1 串行算法与程序

考虑长方形区域 $\Omega = (0,a)\times(0,b)$ 上的 Poisson 方程:

$$\begin{cases} -\Delta u = f, & x \in \Omega \\ u = g, & x \in \partial\Omega \end{cases}$$

采用均匀网格及 5 点差分格式离散上述方程. 设 $h_x = a/n$, $h_y = b/m$ 为网格步长, $x_i = ih_x$, $y_j = jh_y$, $u_{i,j} = u(x_i, y_j)$, $f_{i,j} = f(x_i, y_j)$, $g_{i,j} = g(x_i, y_j)$, $i = 0, \ldots, n$, $j = 0, \ldots, m$. 则差分方程为:

$$\begin{cases} \dfrac{2u_{i,j} - u_{i+1,j} - u_{i-1,j}}{h_x^2} + \dfrac{2u_{i,j} - u_{i,j+1} - u_{i,j-1}}{h_y^2} = f_{i,j} \\ u_{i,0} = g_{i,0}, u_{i,m} = g_{i,m}, u_{0,j} = g_{0,j}, u_{n,j} = g_{n,j} \\ \qquad i = 1, \ldots, n-1, \quad j = 1, \ldots, m-1 \end{cases}$$

令 $d = 1/(2/h_x^2 + 2/h_y^2)$, $d_x = d/h_x^2$, $d_y = d/h_y^2$, 则差分方程可写成:

$$\begin{cases} u_{i,j} - d_x \cdot (u_{i+1,j} + u_{i-1,j}) - d_y \cdot (u_{i,j+1} + u_{i,j-1}) = d \cdot f_{i,j} \\ u_{i,0} = g_{i,0},\ u_{i,m} = g_{i,m},\ u_{0,j} = g_{0,j},\ u_{n,j} = g_{n,j} \\ \qquad i = 1, \ldots, n-1, \quad j = 1, \ldots, m-1 \end{cases}$$

Jacobi 迭代的公式为:

$$u_{i,j}^{\text{new}} = d \cdot f_{i,j} + d_x \cdot (u_{i+1,j}^{\text{old}} + u_{i-1,j}^{\text{old}}) + d_y \cdot (u_{i,j+1}^{\text{old}} + u_{i,j-1}^{\text{old}})$$
$$i = 1, \ldots, n-1, \quad j = 1, \ldots, m-1$$

程序实例中取 $g(x, y) = -(x^2 + y^2)/4$, $f(x, y) = 1$. 显然解析解为 $u_a(x, y) = -(x^2 + y^2)/4$, 并且 $\forall n, m > 0$, 差分方程的解等于解析解.

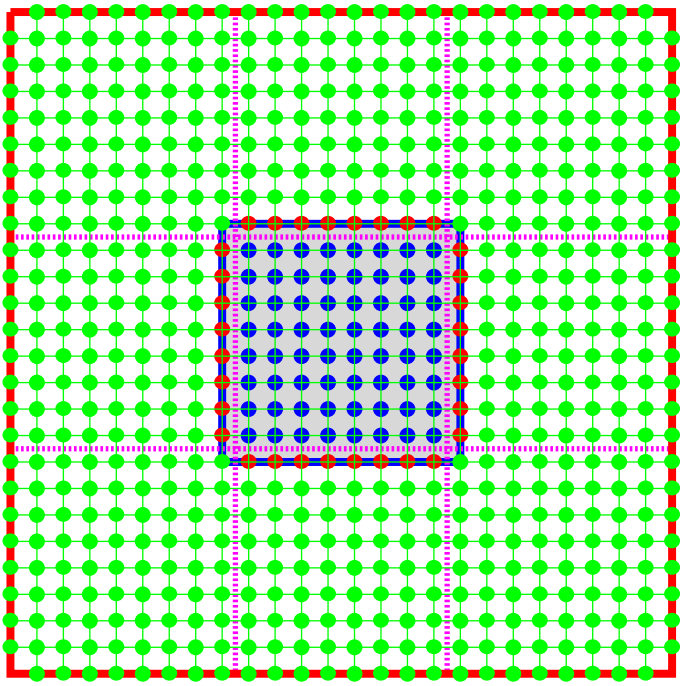
串行程序: 【jacobi0.f】

§5.7.2.2 并行算法与程序

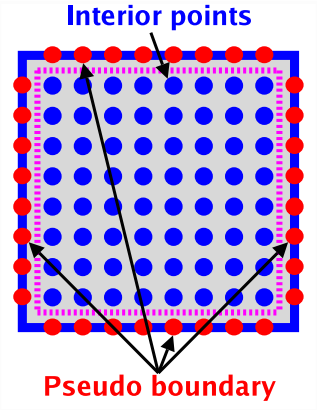
设处理器数为 NPROCS, 将计算区域按二维方式划分成 NP × NQ 个子区域, NP × NQ = NPROCS. 相邻子区域间有一个网格步长的重叠, 以便于子区域间的数据交换. 每个子区域包含大致相等的网格内点数. 每个进程负责一个子区域的计算.

文件 jacobi1.cmm 中定义了一个公共块 /PARMS/, 它包含一些重要的全局变量以及包含 mpif.h 的语句. 并行程序每个子程序的开始都包含 jacobi1.cmm, 以避免通过参数表在子程序间传递大量的参数. 这些全局变量如下:

NPROCS	MPI_COMM_WORLD 中的进程数.
NP, NQ	x 方向和 y 方向的进程数.
MYRANK	进程在 MPI_COMM_WORLD 中的进程号.
NGLOB, MGLOB	整个区域的网格点数.
NLOCA, MLOCA	子区域的网格点数.
RANK_X, RANK_Y	x 方向和 y 方向的进程坐标.
IOFFSET, JOFFSET	子区域原点 (左下角) 在整个网格中的坐标.



NGLOB 和 MGLOB 表示整个网格规模. NLOCA 和 MLOCA 表示子区域的网格规模. 子区域上的网格点可表示为 (0:NLOCA, 0:MLOCA), 其中 (1:NLOCA-1, 1:MLOCA-1) 为子区域的内点. 并行程序中每个进程仅负责子区域内点的计算. 网格点 (0, 1:MLOCA-1), (NLOCA, 1:MLOCA-1), (1:NLOCA-1, 0), 和 (1:NLOCA-1, MLOCA) 为子区域的“边界点”, 它们可能是整个计算区域的边界点 (物理边界), 也可能是相邻子区域的内点, 后者通常称为“内边界点” (inner boundary points), “拟/伪边界点” (pseudo boundary points), 或“幽灵点” (ghost points).



进行子区域划分后, 每个子区域内点的迭代公式与串行程序完全一样, 但每次迭代后必须通过消息传递来从相邻子区域获取非物理边界点上的新的函数近似值.

子区域划分按如下原则进行:

- 如果用户指定了处理器的划分方式 (即用户给出了 NP, NQ 并且它们满足 NP × NQ = NPROCS), 则采用用户指定的处理器的划分. 否则根据总的网格点数 NGLOB 和 MGLOB 调用子程序 PARTITION 计算出适当的 NP, NQ, 使得子区域中两个方向上的网格点数尽量接近, 从而使得总的通信量达到最小. 子程序 PARTITION 是用 C 语言编写的, 源代码在文件 partition.c 中.
- 为方便子区域间边界数据的交换, 用二维坐标 (RANK_X, RANK_Y) 来标识子区域和进程, 它们与进

程序号 MYRANK 的关系是:

$$\begin{cases} \texttt{RANK\_X} = \mathrm{mod}(\texttt{MYRANK}, \texttt{NP}) \\ \texttt{RANK\_Y} = \texttt{MYRANK}/\texttt{NP} \\ \texttt{MYRANK} = \texttt{RANK\_X} + \texttt{RANK\_Y} \times \texttt{NP} \end{cases}$$

- 给定处理器划分后, 子区域的大小通过将整个计算网格的内点尽量均匀地分配给各子区域来确定. 以 x 方向为例, NLOCA 的计算公式如下:

$$\texttt{NLOCA} = \begin{cases} (\texttt{NGLOB} - 1)/\texttt{NP} + 1, & \text{如果 NGLOB-1 能被 NP 整除} \\ (\texttt{NGLOB} - 1)/\texttt{NP} + 1 + \delta, & \text{如果 NGLOB-1 不能被 NP 整除} \end{cases}$$

令 $r = \mathrm{mod}(\texttt{NGLOB} - 1, \texttt{NP})$, 则 δ 定义如下:

$$\delta = \begin{cases} 1, & \text{if } \texttt{RANK\_X} < r \\ 0, & \text{if } \texttt{RANK\_X} \geq r \end{cases}$$

- 按上式确定子区域大小后, 子区域的原点坐标 (在全局网格中的网格点编号) 很容易直接计算出来 (如果定义了适当的通信器, 也可调用 MPI_SCAN 函数计算). 仍以 x 方向为例, 我们有:

$$\texttt{IOFFSET} = \begin{cases} \texttt{(NGLOB-1)/NP} \cdot \texttt{RANK\_X} + \texttt{RANK\_X}, & \text{if } \texttt{RANK\_X} < r \\ \texttt{(NGLOB-1)/NP} \cdot \texttt{RANK\_X} + r, & \text{if } \texttt{RANK\_X} \geq r \end{cases}$$

其中 $r = \mathrm{mod}(\texttt{NGLOB} - 1, \texttt{NP})$.

以上参数的计算在子程序 INIT0 中完成.

子程序 JACOBI 中定义了两个数据类型 T1 和 T2, 分别对应于数组的一列 (x 方向) 和一行 (y 方向), 用于子区域内边界的消息传递.

下面给出两个 MPI 代码实例. 它们的区别在于, 第一个代码采用函数 MPI_Sendrecv 进行消息传递, 而第二个代码则采用持久通信函数进行消息传递.

COMMON 块定义文件 jacobi1.cmm: 【jacobi1.cmm】

Jacobi 迭代: 数据交换方式: 【jacobi1.f】

Jacobi 迭代的改进: 数据平移方式: 【jacobi1a.f】

COMMON 块定义文件 jacobi2.cmm: 【jacobi2.cmm】

使用持久通信函数的 Jacobi 迭代: 【jacobi2.f】

第六章　进程组与通信器

§6.1　基本概念

§6.1.1　进程组

进程组是一组进程的有序集合, 它定义了通信器中进程的集合及进程的序号. MPI 中进程组与通信器类似通过句柄来进行操作. 一个 MPI 程序创建时预定义了两个进程组句柄: 一个是 `MPI_GROUP_EMPTY`, 它代表由空进程组集合构成的进程组, 另一个是, `MPI_GROUP_NULL`, 表示非法进程组.

需要注意的是, 进程组的句柄是进程所固有的, 只对本进程有意义. 因此进程组的句柄通过通信在进程间进行传递是无意义的.

§6.1.2　上下文 (Context)

上下文是通信器的一个固有性质, 它为通信器划分出特定的通信空间. 消息都是在一个给定的上下文中进行传递. 不同上下文间不允许进行消息的收发, 这样可以确保不同通信器中的消息不会混淆. 此外, MPI 通常也要求点对点通信与聚合通信是独立的, 它们间的消息不会互相干扰.

上下文对用户是不可见的. 它是 MPI 实现中的一个内部概念.

§6.1.3　域内通信器 (Intracommunicator)

域内通信器由进程组和上下文构成. 一个通信器的进程组中必须包含定义该通信器的进程作为其成员. 此外, 为了优化通信以及支持处理器实际的拓扑联接方式, 通信器中还可以定义一些附加属性, 如进程间的拓扑联接方式等. 域内通信器可以用于点对点通信, 也可以用于聚合通信.

MPI 预定义的域内通信器包括 `MPI_COMM_WORLD`, `MPI_COMM_SELF` 和 `MPI_COMM_NULL`, 其中 `MPI_COMM_NULL` 代表非法通信器.

§6.1.4　域间通信器 (Intercommunicator)

域间通信器用于分属于不同进程组的进程间的点对点通信. 一个域间通信器由两个进程组构成. 域间通信器不能定义进程的拓扑联接信息, 也不能用于聚合通信.

§6.2　进程组操作函数

§6.2.1　查询进程组大小和进程在组中的序号

C

```
int MPI_Group_size(MPI_Group group, int *size)

int MPI_Group_rank(MPI_Group group, int *rank)
```

Fortran 77

```
MPI_GROUP_SIZE(GROUP, SIZE, IERR)
INTEGER GROUP, SIZE, IERR

MPI_GROUP_RANK(GROUP, RANK, IERR)
INTEGER GROUP, RANK, IERR
```

上述两个函数分别返回给定进程组的大小 (包含的进程个数) 及进程在其中的序号. 它们与 MPI_COMM_SIZE 和 MPI_COMM_RANK 完全类似.

§6.2.2 两个进程组间进程序号的映射

C

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

Fortran 77

```
MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1,
+        GROUP2, RANKS2, IERR)
INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERR
```

该函数给出进程组 group1 中的一组进程序号 ranks1 所对应的进程在进程组 group2 中的序号 ranks2. n 给出数组 ranks1 和 ranks2 中序号的个数.

§6.2.3 比较两个进程组

C

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

Fortran 77

```
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERR)
INTEGER GROUP1, GROUP2, RESULT, IERR
```

比较两个进程组. 返回时, 如果两个进程组包含的进程及它们的序号完全一样则 result = MPI_IDENT, 如果两个进程组包含的进程一样但序号不同, 则 result = MPI_SIMILAR, 否则 result = MPI_UNEQUAL.

§6.2.4 进程组的创建与释放

§6.2.4.1 获取通信器中的进程组

C

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

Fortran 77

```
MPI_COMM_GROUP(COMM, GROUP, IERR)
INTEGER COMM, GROUP, IERR
```

在 group 参数中返回指定通信器的进程组.

§6.2.4.2 进程组的并集

C

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
        MPI_Group *newgroup)
```

Fortran 77

```
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERR
```

newgroup 返回 group1 与 group2 的并集构成的进程组. 新进程组中进程序号的分配原则是先对属于 group1 的进程按 group1 的序号编号, 再对属于 group2 但不属于 group1 的进程按 group2 的序号编号.

§6.2.4.3 进程组的交集

C

```
int MPI_Group_intersection(MPI_Group group1,
        MPI_Group group2, MPI_Group *newgroup)
```

Fortran 77

```
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERR
```

newgroup 返回 group1 与 group2 的交集构成的进程组. 新进程组中进程的序号按 group1 的序号进行编排.

§6.2.4.4 进程组的差集

C

```
int MPI_Group_difference(MPI_Group group1,
        MPI_Group group2, MPI_Group *newgroup)
```

Fortran 77

```
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERR
```

newgroup 返回的新进程组由属于 group1 但不属于 group2 的进程构成, 序号按 group1 中的序号进行编排.

§6.2.4.5 进程组的子集

C

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks,
        MPI_Group *newgroup)
```

Fortran 77

```
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERR
```

新进程组 newgroup 由老进程组 group 中用数组 ranks 给出序号的进程组成, n 是数组 ranks 的元素个数. 新进程组中进程的序号亦由 ranks 数组决定, 即进程组 newgroup 中序号为 i 的进程是老进程组 group 中序号为 ranks[i] (Fortran 77 中应为 RANKS($i+1$)) 的进程.

该函数也可用来对进程组中的进程进行重新排序.

当参数 n = 0 时将创建一个空进程组 MPI_GROUP_EMPTY.

§6.2.4.6 进程组减去一个子集

C

```
int MPI_Group_excl(MPI_Group group, int n, int *ranks,
        MPI_Group *newgroup)
```

Fortran 77

```
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERR
```

该函数将一个进程组进程的集合减去其中的一个子集而得到一个新进程组. 减去的进程序号由数组 ranks 给出, n 给出减去的进程数. 新进程组的序号按进程在老进程组中的序号顺序排列得出.

§6.2.4.7 进程序号范围构成的子集

C

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

Fortran 77

```
MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERR)
INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERR
```

将由一组进程序号范围构成的子集组成一个新进程组. 每个范围由一个三元数对 (起始序号, 终止序号, 步长) 描述, n 为范围个数. 即构成新进程组的进程集合由老进程组中序号属于下述集合的进程组成:

$$\{r \mid r = \texttt{ranges}[i]\texttt{[0]} + k \cdot \texttt{ranges}[i]\texttt{[2]},$$
$$k = 0, \ldots, \lfloor \frac{\texttt{ranges}[i]\texttt{[1]} - \texttt{ranges}[i]\texttt{[0]}}{\texttt{ranges}[i]\texttt{[2]}} \rfloor, \quad i = 0, \ldots, \texttt{n-1}\}$$

上式中所有计算出的 r 必须互不相同, 否则调用出错. 新进程组中进程的序号按上式中先 k 再 i 的顺序编排 (Fortran 77 中数组标号应该加 1).

§6.2.4.8 进程序号范围构成的子集的补集

C

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

Fortran 77

```
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERR)
INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERR
```

将老进程组减去由一组进程序号范围给出的进程而得到一个新的进程组, 它相当于取函数 MPI_Group_range_incl 给出的子集的补集. 各项参数的含义与 MPI_Group_range_incl 类似.

§6.2.4.9 进程组的释放

当一个进程组不再需要时可调用 MPI_GROUP_FREE 来将其释放.

C

```
int MPI_Group_free(MPI_Group *group)
```

Fortran 77

```
MPI_GROUP_FREE(GROUP, IERR)
INTEGER GROUP, IERR
```

函数返回时会将 group 置成 MPI_GROUP_NULL 以防止以后被误用. 实际上, 函数只是将该进程组加上释放标志. 只有基于该进程组的所有通信器均已被释放后才会真的将其释放.

§6.3 域内通信器操作函数

本节介绍用于对通信器进行操作的函数. 其中 MPI_COMM_SIZE 和 MPI_COMM_RANK 在前面已做过介绍, 这里不再重复.

§6.3.1 比较两个通信器

C

```
int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2,
     int *result)
```

Fortran 77

```
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERR)
INTEGER COMM1, COMM2, RESULT, IERR
```

比较两个 (域内) 通信器. 返回时, 如果两个通信器代表同一个通信域 (即两个通信器中的进程可以互相通信) 则 result = MPI_IDENT; 如果两个通信器不代表同一通信域, 但它们的进程组相同, 即它们包含的进程相同并且进程的序号也相同, 则 result = MPI_CONGRUENT; 如果两个通信器包含的进程相同但进程的序号不同, 则 result = MPI_SIMILAR; 否则 result = MPI_UNEQUAL.

§6.3.2 通信器的创建与释放

本节中的函数用于创建新通信器. 新通信器必须在一个已有通信器 (老通信器) 的基础上创建. 调用一个通信器创建函数时, 所有属于老通信器的进程必须同时调用该函数. 返回时, 属于新通信器的进程得到新通信器的句柄, 而不属于新通信器的进程则得到 MPI_COMM_NULL.

§6.3.2.1 通信器的复制

C

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

Fortran 77

```
MPI_COMM_DUP(COMM, NEWCOMM, IERR)
INTEGER COMM, NEWCOMM, IERR
```

生成一个与 comm 具有完全相同属性的新通信器 newcomm. 注意, 新通信器 newcomm 与老通信器 comm 代表着不同的通信域, 因此在它们之间不能进行通信操作 (见下面例 6.1), 即用 comm 发送的消息不能用 newcomm 来接收, 反之亦然.

该函数通常用在并行库函数中: 在库函数的开头将用户提供的通信器参数复制产生一个新通信

器, 并库函数中使用新通信器进行通信, 而在库函数返回前将新通信器释放, 这样可以确保库函数中的通信不会与用户的其它通信相互干扰.

例 6.1 复制的通信器与原通信器的比较：【06-ex1.f】

§6.3.2.2 创建通信器

C

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

Fortran 77

```
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERR)
INTEGER COMM, GROUP, NEWCOMM, IERR
```

创建一个包含指定进程组 group 的新通信器 newcomm. 这个函数并不将 comm 的其它属性传递给 newcomm, 而是为 newcomm 建立一个新的上下文. 返回时, 属于进程组 group 的进程中 newcomm 等于新通信器的句柄, 而不属于进程组 group 的进程中 newcomm 则等于 MPI_COMM_NULL.

§6.3.2.3 分裂通信器

C

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
        MPI_Comm *newcomm)
```

Fortran 77

```
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERR)
INTEGER COMM, COLOR, KEY, NEWCOMM, IERR
```

该函数按照由参数 color 给出的颜色将通信器中的进程分组, 所有具有相同颜色的进程构成一个新通信器. 新通信器中进程的序号按参数 key 的值排序, 两个进程的 key 值相同时则按它们在老通信器 comm 中的序号排序. 返回时, newcomm 等于进程所属的新通信器的句柄. color 必须是非负整数或 MPI_UNDEFINED. 如果 color = MPI_UNDEFINED, 则表示进程不属于任何新通信器, 返回时 newcomm = MPI_COMM_NULL.

§6.3.2.4 通信器的释放

当一个通信器不再需要时可调用 MPI_COMM_FREE 来将其释放.

C

```
int MPI_Comm_free(MPI_Comm *comm)
```

Fortran 77

```
MPI_COMM_FREE(COMM, IERR)
INTEGER COMM, IERR
```

函数返回时会将 comm 置成 MPI_COMM_NULL 以防止以后被误用. 实际上, 该函数只是将通信器加上释放标志. 只有所有引用该通信的操作均已完成后才会真的将其释放.

§6.3.2.5 例: Jacobi 迭代求解二维 Poisson 方程

在 §5.7.2 (第 55 页) Jacobi 迭代求解二维 Poisson 方程的实例中, 我们可以在每个进程中定义两个新通信器, 一个由 x 方向的一行处理器构成, 称为 COMM_X, 另一个由 y 方向的一列处理器构成, 称

为 COMM_Y. 进程的坐标 RANK_X 和 RANK_Y 可分别取成进程在 COMM_X 和 COMM_Y 中的序号. 子区域拟边界值交换的通信利用新通信器来进行.

Jacobi 迭代, COMMON 文件:【jacobi3.cmm】

Jacobi 迭代:【jacobi3.f】

§6.4 通信器的附加属性 (Caching)

略.

§6.5 域间通信器 (Intercommunicator)

略.

§6.6 进程拓扑结构

进程拓扑结构是 (域内) 通信器的一个附加属性, 它描述一个进程组各进程间的逻辑联接关系. 进程拓扑结构的使用一方面可以方便、简化一些并行程序的编制, 另一方面可以帮助 MPI 系统更好地将进程映射到处理器以及组织通信的流向, 从而获得更好的并行性能.

MPI 的进程拓扑结构定义为一个无向图, 图中结点 (node) 代表进程, 而边 (edge) 代表进程间的联接. MPI 进程拓扑结构也被称为虚拟拓扑结构, 因为它不一定对应处理器的物理联接.

MPI 提供了一组函数用于创建各种进程拓扑结构. 应用问题中较为常见、也是较为简单的一类进程拓扑结构具有网格形式, 这类结构中进程可以用迪卡尔坐标来标识, MPI 中称这类拓扑结构为迪卡尔 (Cartesian) 拓扑结构, 并且专门提供了一组函数对它们进行操作.

§6.6.1 迪卡尔拓扑结构

§6.6.1.1 创建迪卡尔拓扑结构

C

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
        int *dims, int *periods, int reorder,
        MPI_Comm *comm_cart)
```

Fortran 77

```
      MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER,
     +        COMM_CART, IERR)
      INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERR
      LOGICAL PERIODS(*), REORDER
```

该函数从一个通信器 comm_old 出发, 创建一个具有迪卡尔拓扑结构的新通信器 comm_cart.

ndims 给出进程网格的维数. 数组 dims 给出每维中的进程数. 数组 periods 则说明进程在各个维上的联接是否具有周期性, 即该维中第一个进程与最后一个进程是否相联, 周期的迪卡尔拓扑结构也称为环面 (torus) 结构, periods[i] = true 表明第 i 维是周期的, 否则则是非周期的. reorder 指明

是否允许在新通信器 comm_cart 中对进程进行重新排序. 在某些并行机上, 根据处理器的物理联接方式及所要求的进程拓扑结构对进程重新排序有助于提高并行程序的性能.

comm_cart 中各维的进程数之积必须不大于 comm_old 中的进程数, 即:

$$\prod_{i=0}^{\texttt{ndims}-1} \texttt{dims}[i] \leq \texttt{NPROCS}$$

其中 NPROCS 为 comm_old 的进程数. 当 $\prod_{i=0}^{\texttt{ndims}-1} \texttt{dims}[i] < \texttt{NPROCS}$ 时一些进程将不属于 comm_cart, 这些进程的 comm_cart 参数将返回 MPI_COMM_NULL.

§6.6.1.2 辅助函数

C

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

Fortran 77

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERR)
INTEGER NNODES, NDIMS, DIMS(*), IERR
```

该函数当给定总进程数及维数时自动计算各维的进程数, 使得它们的乘积等于总进程数, 并且各维上的进程数尽量接近. 确切地说, 给定 nnodes 和 ndims, 函数计算正整数 dims[i], $i = 0, \ldots, \texttt{ndims} - 1$, 使得 $\prod_{i=0}^{\texttt{ndims}-1} \texttt{dims}[i] = \texttt{nnodes}$ 并且各 dims[i] 的值尽量接近.

该函数要求输入时 dims 的中元素的值为非负整数, 并且它仅修改 dims 中输入值为 0 的元素. 因此用户可以指定一些维的进程数而仅要求计算其它维的进程数.

局限: 没有考虑实际数据在各维上的大小. 例如, 在我们的 Jacobi 迭代实例中, 假如进程数为 4, 差分网络为 100×400, 则理想的进程拓扑结构应为 1×4, 此时无法调用 MPI_DIMS_CREATE 自动计算最优进程划分.

§6.6.1.3 将迪卡尔拓扑结构分割成低维子结构

C

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims,
        MPI_Comm *newcomm)
```

Fortran 77

```
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERR)
INTEGER COMM, NEWCOMM, IERR
LOGICAL REMAIN_DIMS(*)
```

该函数将一个具有迪卡尔拓扑结构的通信器 comm 中指定的维抽取出来, 构成一个具有低维迪卡尔结构 (子网格) 的新通信器 newcomm. 数组 remain_dims 的元素指定相应的维是否被包含在新通信器中, 如果 remain_dims[i] = true 表示子网格包含第 i 维, 否则表示子网格不包含第 i 维.

§6.6.1.4 查询迪卡尔拓扑结构的维数

C

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

Fortran 77

```
MPI_CARTDIM_GET(COMM, NDIMS, IERR)
```

```
INTEGER COMM, NDIMS, IERR
```

MPI_Cartdim_get 在 ndims 中返回通信器 comm 的迪卡尔拓扑结构的维数.

§6.6.1.5 查询迪卡尔拓扑结构的详细信息

C

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims,
        int *periods, int *coords)
```

Fortran 77

```
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERR)
INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERR
LOGICAL PERIODS(*)
```

MPI_Cart_get 返回通信器 comm 的迪卡尔拓扑结构的详细信息. 数组 dims, periods 和 coords 分别返回各维的进程数、是否周期及当前进程的迪卡尔坐标. 参数 maxdims 给出数组 dims, periods 和 coords 的长度的上界.

§6.6.1.6 迪卡尔坐标到进程序号的映射

C

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

Fortran 77

```
MPI_CART_RANK(COMM, COORDS, RANK, IERR)
INTEGER COMM, COORDS(*), RANK, IERR
```

给定一个进程在通信器 comm 中的迪卡尔坐标 coords, 该函数在 rank 中返回进程在 comm 中的进程号. 如果某维具有周期性, 则 coords 中对应的坐标值允许“越界”, 即小于 0 或大于等于该维的进程数.

§6.6.1.7 进程序号到迪卡尔坐标的映射

C

```
int MPI_Cart_coords(MPI_Comm comm, int rank,
   int maxdims, int *coords)
```

Fortran 77

```
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERR)
INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERR
```

给定一个进程在通信器 comm 中的进程号 rank, 该函数在 coords 中返回进程在 comm 中的迪卡尔坐标. maxdims 给出数组 coords 的最大长度.

§6.6.1.8 数据平移 (shift) 操作中源地址与目的地址的计算

在一个具有迪卡尔拓扑结构的通信器中经常在一个给定维上对数据进行平移 (shift), 如用 MPI_SENDRECV 将一块数据发送给该维上后面一个进程, 同时接收从该维上前面一个进程发送来的数据. MPI 提供了一个函数来方便这种情况下目的地址和源地址的计算.

C

```
int MPI_Cart_shift(MPI_Comm comm, int direction,
```

```
              int disp, int *rank_source, int *rank_dest)
```

Fortran 77

```
      MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE,
     +        RANK_DEST, IERR)
      INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST,
     +        IERR
```

输入参数 direction 是进行数据平移的维号 ($0 \le$ direction $<$ ndims), disp 给出数据移动的“步长”(绝对值) 和“方向”(正负号). 输出参数 rank_source 和 rank_dest 分别是平移操作的源地址和目的地址.

假设指定维上的进程数为 d, 当前进程该维的坐标为 i, 源进程 rank_source 该维的坐标为 i_s, 目的进程 rank_dest 该维的坐标为 i_d, 如果该维是周期的, 则:

$$i_s = i - \texttt{disp} \mod d$$
$$i_d = i + \texttt{disp} \mod d$$

否则:

$$i_s = \begin{cases} i - \texttt{disp}, & \texttt{if } 0 \le i - \texttt{disp} < d \\ \texttt{MPI\_PROC\_NULL}, & \text{otherwise} \end{cases}$$

$$i_d = \begin{cases} i + \texttt{disp}, & \texttt{if } 0 \le i + \texttt{disp} < d \\ \texttt{MPI\_PROC\_NULL}, & \text{otherwise} \end{cases}$$

例 6.2
使用迪卡尔拓扑结构的 *Jacobi* 迭代, *COMMON* 文件: 【jacobi4.cmm】
使用迪卡尔拓扑结构的 *Jacobi* 迭代: 【jacobi4.f】

§6.6.2 一般拓扑结构

§6.6.2.1 创建图 (graph) 拓扑结构

C

```
      int MPI_Graph_create(MPI_Comm comm_old, int nnodes,
              int *index, int *edges, int reorder,
              MPI_Comm *comm_graph)
```

Fortran 77

```
      MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES,
     +        REORDER, COMM_GRAPH, IERR)
      INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*),
     +        COMM_GRAPH, IERR
      LOGICAL REORDER
```

从通信器 comm_old 出发, 创建一个具有给定的图结构的新通信器 comm_graph.

新通信器的拓扑结构图由参数 nnodes, index 和 edges 描述: nnodes 是图的结点数 (如果 nnodes 小于通信器 comm_old 的进程数, 则一些进程将不属于新通信器 comm_graph, 这些进程中参数 comm_graph 的返回值将为 MPI_COMM_NULL), index[i] ($i = 0, \ldots,$ nnodes $- 1$) 给出结点 $0, \ldots, i$ 的邻居数之和, edges 则顺序给出所有结点的邻居的序号. 用 Neighbor(i) 表示第 i 个结点的邻居的序号集合, 则:

$$\text{Neighbor}(0) = \{\text{edges}[j] \mid 0 \le j < \text{index}[0]\}$$

$$\text{Neighbor}(i) = \{\text{edges}[j] \mid \text{index}[i-1] \le j < \text{index}[i]\}$$

$$i = 1, \ldots, \text{nnodes} - 1$$

(注意: Fortran 77 中所有数组下标应该加 1).

参数 reorder 指明是否允许在新通信器中对进程重新编号 (与函数 MPI_Cart_create 中类似).

§6.6.2.2 查询拓扑结构类型

C

```
int MPI_Topo_test(MPI_Comm comm, int *status)
```

Fortran 77

```
MPI_TOPO_TEST(COMM, STATUS, IERR)
INTEGER COMM, STATUS, IERR
```

查询拓扑结构类型. 返回时, 如果 comm 具有迪卡尔拓扑结构则 status = MPI_CART, 如果 comm 具有图结构则 status = MPI_GRAPH, 否则 status = MPI_UNDEFINED.

§6.6.2.3 查询拓扑结构的结点数与边数

C

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes,
        int *nedges)
```

Fortran 77

```
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERR)
INTEGER COMM, NNODES, NEDGES, IERR
```

该函数在参数 nnodes 中返回通信器 comm 的拓扑结构图的结点数 (等于 comm 中的进程数), 在参数 nedges 中返回通信器 comm 的拓扑结构图的边数.

§6.6.2.4 查询拓扑结构的详细参数

C

```
int MPI_Graph_get(MPI_Comm comm, int maxindex,
        int maxedges, int *index, int *edges)
```

Fortran 77

```
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES,
+        IERR)
INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*),
+        IERR
```

该函数返回通信器 comm 的拓扑结构图中的 index 和 edges 数组 (见 §6.6.2.1, 68 页). 参数

maxindex 和 maxedges 分别限定数组 index 和 edges 的最大长度.

§6.6.2.5 查询指定进程的邻居

C

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank,
        int maxneighbors, int *neighbors)
```

Fortran 77

```
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS,
+        IERR)
INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERR
```

该函数在数组 neighbors 中返回通信器 comm 中序号为 rank 的进程的所有邻居的序号. maxneighbors 限定数组 neighbors 的最大长度.

§6.6.3 底层支持函数

本节的两个函数是为方便 MPI 拓扑结构的实现而设, 普通用户通常不会用到.

§6.6.3.1 查询指定迪卡尔结构下理想的进程编号方式

C

```
int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims,
        int *periods, int *newrank)
```

Fortran 77

```
MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERR)
INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERR
LOGICAL PERIODS(*)
```

该函数在 newrank 中返回给定迪卡尔拓扑结构下当前进程的建议编号. 参数 ndims, dims 和 periods 的含义与函数 MPI_Cart_create 中相同 (参看 §6.6.1.1, 65 页).

§6.6.3.2 查询指定图结构下理想的进程编号方式

C

```
int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index,
        int *edges, int *newrank)
```

Fortran 77

```
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERR)
INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERR
```

该函数在 newrank 中返回给定图结构下当前进程的建议编号. 参数 nnodes, index 和 edges 的含义与函数 MPI_Graph_create 中相同 (参看 §6.6.2.1, 68 页).

第七章　文件输入输出

MPI 的输入输出 (I/O) 函数属于 MPI 2.0. 在 MPI 2.0 中, 函数接口定义包含 C, C++, 和 Fortran 三种. 出于严谨性考虑, Fortran 接口中一些参数使用了 Fortran 90 类型 (如 `INTEGER(MPI_OFFSET_KIND)`), 在 Fortran 77 代码中这些参数在不同的平台上可能需要采用不同的写法 (如 `INTEGER*4`, `INTEGER*8` 等等), 因而会影响到代码的可移植性.

本节介绍的函数在 MPICH 1.2.1 中已经全部实现, 但在其它 MPI 版本中可能目前还不能使用. 因此, 在使用本节中介绍的函数前应先确认所使用的 MPI 系统是否支持它们. 如果使用 MPICH 1.2.1, 在用到本节函数的 Fortran 子程序中除了需要包含头文件 `'mpif.h'` 外, 还要包含头文件 `'mpiof.h'`. (本节介绍的函数 MPICH 1.2.0 中也已实现, 但有些函数似乎有问题).

§7.1　基本术语

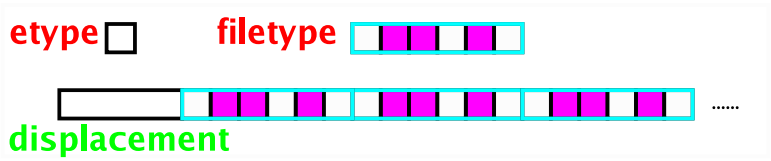
文件 (file) MPI 的“文件”可以看成由具有相同或不同类型的数据项构成的序列. MPI 支持对文件的顺序和随机访问. MPI 的文件是和进程组相关联的: MPI 打开文件的函数 (`MPI_File_open`) 中要求指定一个通信器, 并且该通信器中所有进程必须同时对文件进行打开或关闭操作.

起始位置 (displacement) 一个文件的起始位置指相对于文件开头以字节为单位的一个绝对地址, 它用来定义一个“文件视窗”的起始位置.

基本单元类型 (etype) 基本单元类型 (elementary type) 是定义一个文件最小访问单元的 MPI 数据类型. 一个文件的基本单元类型可以是任何预定义或用户构造的并已经递交的 MPI 数据类型, 但其类型图中的位移必须非负并且位移序列是 (非严格) 单调上升的. MPI 的文件操作完全以基本单元类型为单位: 文件中的位移 (offset) 以基本单元类型的个数而非字节数为单位, 文件指针总是指向一个基本单元的起始地址.

文件单元类型 (filetype) 文件单元类型也是一个 MPI 数据类型, 它定义了对一个文件的存取图案. 文件单元类型可以等于基本单元类型, 也可以是在基本单元类型基础上构造并已递交的任意 MPI 数据类型. 文件单元类型的域必须是基本单元类型的域的倍数, 并且文件单元类型中间的“洞”的大小也必须是基本单元类型的域的倍数.

视窗 (view) 文件视窗指一个文件中目前可以访问的数据集. 文件视窗由三个参数定义: 起始位置, 基本单元类型, 文件单元类型. 文件视窗指从起始位置开始将文件单元类型连续重复排列构成的图案, MPI 对文件进行存取操作时将“跳过”图案中的“空洞”, 见下图.



位移 (offset) MPI 的 I/O 函数中位移总是相对于文件起始位置 (当前视窗) 计算, 并且以基本单元类型的域为单位.

文件大小 (file size) 文件大小指从文件开头到文件结尾的总字节数.

文件指针 (file pointer) 文件指针是 MPI 管理的两个内部位移 (隐式位移). MPI 在每个进程中为每个打开的文件定义了两个文件指针, 一个供本进程独立使用, 称为独立文件指针 (individual file pointer), 另一个供打开文件的进程组中所有进程共同使用, 称为共享文件指针 (shared file pointer).

文件句柄 (file handle) MPI 打开一个文件后, 返回给调用程序一个文件句柄, 供以后访问及关闭该文件时用. MPI 的文件句柄在文件关闭时被释放.

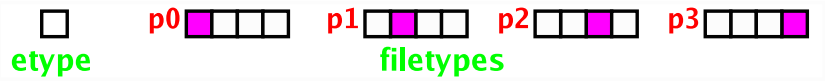
例 7.1 假设 ext(MPI_REAL) = 4, *etype* = MPI_REAL, 打开文件 fh 的进程组包括 4 个进程 p_i, $i = 0, 1, 2, 3$, 四个进程中文件单元类型分别定义如下:

$$
\begin{aligned}
p_0: &\quad \mathit{filetype} = \{(\mathtt{REAL},\ 0), (\mathtt{LB}, 0), (\mathtt{UB}, 16)\} \\
p_1: &\quad \mathit{filetype} = \{(\mathtt{REAL},\ 4), (\mathtt{LB}, 0), (\mathtt{UB}, 16)\} \\
p_2: &\quad \mathit{filetype} = \{(\mathtt{REAL},\ 8), (\mathtt{LB}, 0), (\mathtt{UB}, 16)\} \\
p_3: &\quad \mathit{filetype} = \{(\mathtt{REAL}, 12), (\mathtt{LB}, 0), (\mathtt{UB}, 16)\}
\end{aligned}
$$

如果四个进程中独立文件指针均为 0, 则调用:

```
CALL MPI_FILE_READ(FH, A, 1, MPI_REAL, STATUS, IERR)
```

将文件开头的四个数依次赋给四个进程中的变量 A.



§7.2 基本文件操作

§7.2.1 打开 MPI 文件

C

```
int MPI_File_open(MPI_Comm comm, char *filename,
        int amode, MPI_Info info, MPI_File *fh)
```

Fortran

```
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERR)
CHARACTER*(*) FILENAME
INTEGER COMM, AMODE, INFO, FH, IERR
```

打开一个 MPI 文件. 文件成功打开后, 在参数 fh 中返回该文件的句柄, 供以后对该文件进行操作用. comm 指定打开文件的通信器, 所有属于 comm 的进程必须同时调用该函数. filename 是打开的文件名, comm 中所有进程提供的文件名必须代表同一个文件. amode 给出文件的打开模式 (细节见下文), comm 中所有进程必须提供同样的 amode 参数.

输入参数 INFO 提供给 MPI 系统一些附加提示信息, 它由 MPI 的实现具体定义, 我们不在此介绍. 用户可用常数 MPI_INFO_NULL 代替它, 表示没有提示信息.

amode 参数与普通操作系统中的文件访问模式类似. MPI 为其定义的值有:

- MPI_MODE_RDONLY — 只进行读操作

- MPI_MODE_RDWR — 同时进行读操作和写操作
- MPI_MODE_WRONLY — 只进行写操作
- MPI_MODE_CREATE — 如果文件不存在则创建一个新文件
- MPI_MODE_EXCL — 创建文件时若文件存在则打开失败
- MPI_MODE_DELETE_ON_CLOSE — 关闭文件后将其删除
- MPI_MODE_UNIQUE_OPEN — 用户可以确保只有当前程序访问该文件
- MPI_MODE_SEQUENTIAL — 只能对文件进行顺序读写
- MPI_MODE_APPEND — 打开后将文件指针置于文件结尾处

上述各模式可以用二进制“或”运算进行迭加 (C 中为 “|”, Fortran 77 中可以用 “+” 代替“或”, 只要同一模式不出现两次以上).

§7.2.2 关闭 MPI 文件

C

```
int MPI_File_close(MPI_File *fh)
```

Fortran

```
MPI_FILE_CLOSE(FH, IERR)
INTEGER FH, IERR
```

关闭文件. 文件关闭完成后, 文件句柄被释放, fh 被置成 MPI_FILE_NULL. 用户应该确保调用该函数前所有与该文件有关的操作请求均已完成.

MPI_FILE_CLOSE 是聚合型函数, 进程组中所有进程必须同时调用并且提供同样的参数.

§7.2.3 删除文件

C

```
int MPI_File_delete(char *filename, MPI_Info info)
```

Fortran

```
MPI_FILE_DELETE(FILENAME, INFO, IERR)
CHARACTER*(*) FILENAME
INTEGER INFO, IERR
```

删除指定文件. 如果文件不存在, 则返回 MPI_ERR_NO_SUCH_FILE 错误. 要删除的文件通常应该是没打开的或已关闭的.

§7.2.4 设定文件长度

C

```
int MPI_File_set_size(MPI_File fh, MPI_Offset size)
```

Fortran

```
MPI_FILE_SET_SIZE(FH, SIZE, IERR)
INTEGER FH, IERR
INTEGER(KIND=MPI_OFFSET_KIND) SIZE
```

将指定文件的长度 (指从文件开头到文件结尾的字节数) 设成 size. 如果当前文件长度大于 size, 则文件将被截断成 size 字节. 如果当前文件长度小于 size, 则文件大小被设为指定长度, 此时

操作系统不一定为该文件实际分配存储空间.

MPI_FILE_SET_SIZE 是聚合型函数, 进程组中所有进程必须同时调用并且提供同样的参数.

§7.2.5 为文件预留空间

C

```
int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
```

Fortran

```
MPI_FILE_PREALLOCATE(FH, SIZE, IERR)
INTEGER FH, IERR
INTEGER(KIND=MPI_OFFSET_KIND) SIZE
```

如果当前文件长度大于等于 size, 则该函数不起任何作用. 否则它将文件长度调整到 size 指定的大小, 并且强制操作系统为文件分配好存储空间.

MPI_FILE_PREALLOCATE 是聚合型函数, 进程组中所有进程必须同时调用并且提供相同的参数.

§7.2.6 查询文件长度

C

```
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
```

Fortran

```
MPI_FILE_GET_SIZE(FH, SIZE, IERR)
INTEGER FH, IERR
INTEGER(KIND=MPI_OFFSET_KIND) SIZE
```

在参数 size 中返回指定文件的当前长度.

§7.3 查询文件参数

§7.3.1 查询打开文件的进程组

C

```
int MPI_File_get_group(MPI_File fh, MPI_Group *group)
```

Fortran

```
MPI_FILE_GET_GROUP(FH, GROUP, IERR)
INTEGER FH, GROUP, IERR
```

该函数在参数 group 中返回与文件句柄 fh 相关联 (即打开该文件) 的进程组句柄. 用户应该负责在不再需要该句柄时将其释放.

§7.3.2 查询文件访问模式

C

```
int MPI_File_get_amode(MPI_File fh, int *amode)
```

Fortran

```
MPI_FILE_GET_AMODE(FH, AMODE, IERR)
INTEGER FH, AMODE, IERR
```

该函数在参数 amode 中返回文件句柄 fh 所对应的文件的访问模式.

§7.4 设定文件视窗

C

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
        MPI_Datatype etype, MPI_Datatype filetype,
        char *datarep, MPI_Info info)
```

Fortran

```
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP,
+        INFO, IERR)
INTEGER FH, ETYPE, FILETYPE, INFO, IERR
CHARACTER*(*) DATAREP
INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

将文件视窗的起始位置设为 disp (从文件开头以字节为单位计算), 基本单元类型设为 etype, 文件单元类型设为 filetype. 参数 datarep 给出文件中的数据表示格式. 参数 info 用来重新指定附加提示信息.

MPI_FILE_SET_VIEW 是聚合型函数, 进程组中所有进程必须同时调用. 不同进程可以提供不同的 disp, filetype 和 info 参数, 但必须提供相同的 datarep 参数和具有相同域的 etype 参数.

如果文件打开时使用了 MPI_MODE_SEQUENTIAL 模式, 则 disp 参数必须写成 MPI_DISPLACEMENT_CURRENT (代表文件的当前位置).

§7.4.1 文件中的数据表示格式

参数 datarep 是一个字符串, 给出文件中使用的数据表示格式. 它有下面一些可能值:

"native" 文件中数据完全按其在内存中的表示形式存放. 使用该数据表示的文件不能在数据格式不兼容的计算机间交换使用.

"internal" 指 MPI 内部格式, 具体由 MPI 的实现定义. 使用该数据表示的文件可以确保能在使用同一 MPI 系统的计算机间进行交换使用, 即使这些计算机的数据格式不兼容.

"external32" 使用 IEEE 定义的一种通用数据表示格式, external data representation (简称 XDR). 使用该数据表示的文件可以在所有支持 MPI 的计算机间交换使用. 该格式可用于在数据表示不兼容的计算机间交换数据.

许多 MPI 系统目前尚未全部实现上述三种格式 (它们通常只支持 "native" 格式).

除上述数据表示外, 用户还可以通过函数 MPI_REGISTER_DATAREP 定义自己的数据表示形式, 我们不在此介绍. 感兴趣者请参看 MPI 标准文档.

MPI 不将有关数据表示格式的信息写在文件中, 因此用户须保证在设定文件窗口时指定的数据表示格式与文件中的实际数据表示格式相符.

特别需要注意的是, 当 datarep 不等于 "native" 时, 基本单元类型 (etype) 和文件单元类型 (filetype) 在文件中的形式有可能与它们在内存中的形式不一样. 此时, 如果用作单元类型的数据类型是“可移植的” (portable datatype, 定义见 §7.4.2, 76 页), 则 MPI 在函数 MPI_FILE_SET_VIEW 中会自动对其进行调整 (缩放) 以便与文件中的数据表示格式相匹配. 如果用作单元类型的数据类型不是

“可移植”的, 则用户必须保证它们与文件中的数据表示格式相符, 必要时使用 MPI_TYPE_LB 和 MPI_TYPE_UB 来进行调整.

§7.4.2 可移植数据类型

MPI 中一个数据类型称为是可移植的, 如果它是一个预定义数据类型, 或者是在一个可移植的数据类型的基础上使用下述函数之一创建的:

MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_INDEXED,
MPI_TYPE_DUP, MPI_TYPE_CREATE_SUBARRAY,
MPI_TYPE_INDEXED_BLOCK, MPI_TYPE_CREATE_DARRAY

(其中后四个函数是 MPI 2.0 中新增加的数据类型创建函数). 因此, 可移植数据类型的位移和上下界都是以某一预定义数据类型为单位的. 换言之, 可移植的数据类型在其构造过程中不能使用下述函数:

MPI_TYPE_HINDEXED, MPI_TYPE_HVECTOR, MPI_TYPE_STRUCT

(即不能直接以字节为单位来设定数据类型的位移和上下界).

§7.4.3 查询数据类型相应于文件数据表示格式的域

MPI 提供了一个函数用来查询一个 (内存中的) 数据类型在文件中的域 (当文件的数据表示格式不等于 "native" 时, 数据类型在文件中的域可能与它在内存中的域不同).

C

```
int MPI_File_get_type_extent(MPI_File fh,
        MPI_Datatype datatype, MPI_Aint *extent)
```

Fortran

```
MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERR)
INTEGER FH, DATATYPE, IERR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT
```

§7.5 文件读写操作

MPI 进行文件读写操作的函数由下表给出, 其中可分别 xxxx 代表 READ 和 WRITE, 分别对应于读操作和写操作的函数.

定位方式	同步方式	进程组进程间的协同方式	
		非聚合式	聚合式
显式位移	阻塞型	MPI_xxxx_AT	MPI_xxxx_AT_ALL
	非阻塞或分裂型	MPI_Ixxxx_AT	MPI_xxxx_AT_ALL_BEGIN MPI_xxxx_AT_ALL_END
独立文件指针	阻塞型	MPI_xxxx	MPI_xxxx_ALL
	非阻塞或分裂型	MPI_Ixxxx	MPI_xxxx_ALL_BEGIN MPI_xxxx_ALL_END
共享文件指针	阻塞型	MPI_xxxx_SHARED	MPI_xxxx_ORDERED
	非阻塞或分裂型	MPI_Ixxxx_SHARED	MPI_xxxx_ORDERED_BEGIN MPI_xxxx_ORDERED_END

MPI 文件读写操作函数按指定数据在文件中的位置的方式分为使用显式位移 (直接在函数中指定位移量, 以基本单元类型的域为单位)、使用独立文件指针和使用共享文件指针三类. 每种类型的操作不会对其它类型操作的位置产生影响, 如使用显式位移的操作不会改变独立文件指针或共享文件指针, 使用独立文件指针的操作不会改变共享文件指针, 而使用共享文件指针的操作也不会改变独立文件指针.

当一个进程使用独立文件指针或共享文件指针对文件进行操作时, 文件中的位移由文件指针的当前值决定. 一个使用文件指针的操作完成后, 该文件指针的值被自动刷新, 指向文件中操作读或写的最后一个数据之后. 独立文件指针是各进程私有的, 它的刷新仅依赖于本进程, 不受其它进程读写操作的影响. 而共享文件指针则被进程组中所有进程共享, 当多个进程同时使用共享文件指针进行读写时, 每个进程的读写操作都会移动共享文件指针, 文件指针总的移动量相当于所有读写操作的迭加.

文件读写操作函数按进程组中进程间的协同方式分为非聚合式 (noncollective) 和聚合式 (collective) 两种. 非聚合式函数的完成只依赖于本进程, 它们不要求进程组中的所有进程同时调用, 而由各进程分别独立地调用, 当多个进程同时调用非聚合式函数时, 不同进程间对数据读写的先后顺序是不确定的. 而聚合式函数的完成依赖于同组所有进程间的协调, 它们要求进程组中全部进程同时调用, 各进程对数据读写的先后顺序由进程的序号确定.

按照函数调用是否阻塞, 即在文件操作的同时是否还能执行其它任务, MPI 的文件读写函数又分为阻塞型 (blocking)、非阻塞型 (nonblocking) 和分裂型 (split) 三种. 阻塞型函数返回后即表明读写操作已经“完成”, 进程马上可以对读写缓冲区进行后续操作或关闭文件. 非阻塞型文件读写函数与非阻塞型消息传递函数类似, 只向系统发出一个读或写请求, 随后 (特别是在关闭文件前) 进程需要调用 MPI_WAIT 或 MPI_TEST 等函数来等待操作的完成. 分裂型函数将文件的读写操作分解成开始 (BEGIN) 和结束 (END) 两步, 以便允许进程在读写开始和结束之间进行一些其它的计算或通信.

§7.5.1 使用显式位移的阻塞型文件读写

所有使用显式位移的阻塞型文件读写函数 (*_AT, *_AT_ALL) 的接口参数完全一样, 这里仅列出函数 MPI_FILE_READ_AT 的接口参数供参考.

C

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset,
        void *buf, int count, MPI_Datatype datatype,
```

```
            MPI_Status *status)
```

Fortran

```
      MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE,
     +        STATUS, IERR)
      <type>  BUF(*)
      INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE),
     +        IERR
      INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

fh 为文件句柄, offset 为位移. buf, count 和 datatype 分别为数据的缓冲区地址、个数和类型. status 返回操作结果状态 (与通信函数类似).

§7.5.2 使用独立文件指针的阻塞型文件读写

使用独立文件指针的阻塞型文件读写函数与使用显式位移的阻塞型文件读写函数的功能完全一样, 只是文件位移由独立文件指针隐式设定. 这些函数的接口参数中比使用显式位移的函数少了一个 offset 参数, 其它参数完全一样. 作为例子, 我们给出 MPI_FILE_READ 的接口参数.

C

```
int MPI_File_read(MPI_File fh, void *buf, int count,
        MPI_Datatype datatype, MPI_Status *status)
```

Fortran

```
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERR)
<type>  BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE),
+       IERR
```

§7.5.3 使用共享文件指针的阻塞型文件读写

使用共享文件指针的阻塞型文件读写函数的接口参数与使用独立文件指针的阻塞型文件读写函数的接口参数完全一样. 作为例子, 这里给出 MPI_FILE_READ_ORDERED 的接口参数.

C

```
int MPI_File_read_ordered(MPI_File fh, void *buf,
        int count, MPI_Datatype datatype,
        MPI_Status *status)
```

Fortran

```
MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS,
+       IERR)
<type>  BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE),
+       IERR
```

由于使用共享文件指针的文件操作函数中进程组的全部进程共同使用和修改同一个文件指针, 因此这类操作非常类似于以文件为“根进程”的数据收集和散发, 即它们相当于将进程组中各进程的数据块合并写入文件 (收集) 或读取文件中的数据并分发给各进程 (散发). 当使用非聚合式函数

MPI_FILE_READ_SHARED 和 MPI_FILE_WRITE_SHARED 时, 各进程从文件中读取或写入文件的数据块在文件中的相对位置是不确定的, 而聚合式函数 MPI_FILE_READ_ORDERED 和 MPI_FILE_WRITE_ORDERED 则可确保这些数据块在文件中严格按进程序号排列.

§7.5.4 非阻塞型文件读写函数

每个阻塞型非聚合式文件读写函数都有一个对应的非阻塞型函数, 由阻塞型函数的函数名中在 READ 或 WRITE 前面加 I 构成, 如 MPI_FILE_READ 的非阻塞型函数为 MPI_FILE_IREAD. 非阻塞型函数的接口参数中只需将对应的阻塞型函数的参数表中的 status 参数换成 request, 其它参数完全一样. 非阻塞型函数递交文件读或写的请求, 在 request 中返回一个请求句柄, 实际的读或写操作在后台进行. 非阻塞型文件读写函数返回的请求句柄与非阻塞型消息传递函数所返回的句柄的操作方式完全一样, 即用户需在关闭文件前调用 MPI_WAIT, MPI_TEST 等函数来检查、等待操作的完成.

做为例子, 下面列出 MPI_IREAD_AT 的参数.

C

```
int MPI_File_iread_at(MPI_File fh, MPI_Offset offset,
        void *buf, int count, MPI_Datatype datatype,
        MPI_Request *request)
```

Fortran

```
MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE,
+        REQUEST, IERR)
<type>  BUF(*)
INTEGER FH, COUNT, DATATYPE, REQUEST, IERR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

§7.5.5 分裂型文件读写函数

MPI 为每个阻塞型聚合式文件读写函数定义了一对分裂型函数, 分别在阻塞型函数的函数名后面加 _BEGIN 和 _END 构成. 分裂型函数将文件读写操作分解成开始和结束两步, 用户可以在开始和结束之间插入其它通信或计算, 从而实现计算或通信与文件输入输出重叠进行. 这些函数的函数名及包含的接口参数如下:

```
MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
MPI_FILE_READ_AT_ALL_END(fh, buf, status)

MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
MPI_FILE_WRITE_AT_ALL_END(fh, buf, status)

MPI_FILE_READ_ALL_BEGIN(fh, buf, count, datatype)
MPI_FILE_READ_ALL_END(fh, buf, status)

MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)
MPI_FILE_WRITE_ALL_END(fh, buf, status)

MPI_FILE_READ_ORDERED_BEGIN(fh, buf, count, datatype)
```

```
MPI_FILE_READ_ORDERED_END(fh, buf, status)

MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)
MPI_FILE_WRITE_ORDERED_END(fh, buf, status)
```

§7.6 文件指针操作

§7.6.1 独立文件指针操作

§7.6.1.1 移动独立文件指针

C

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset,
        int whence)
```

Fortran

```
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERR)
INTEGER FH, WHENCE, IERR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

改变独立文件指针的位移. 参数 `whence` 可取为下列值:

- `MPI_SEEK_SET` — 将指针的位移设为 `offset`
- `MPI_SEEK_CUR` — 将指针的位移设为当前位移加上 `offset`
- `MPI_SEEK_END` — 将指针的位移设为文件结尾加上 `offset`

§7.6.1.2 查询独立文件指针的当前位移

C

```
int MPI_File_get_position(MPI_File fh,
        MPI_Offset *offset)
```

Fortran

```
MPI_FILE_GET_POSITION(FH, OFFSET, IERR)
INTEGER FH, IERR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

在参数 `offset` 中返回独立文件指针的位移.

§7.6.2 共享文件指针操作

§7.6.2.1 移动共享文件指针

C

```
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset,
        int whence)
```

Fortran

```
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERR)
INTEGER FH, WHENCE, IERR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

改变共享文件指针的位移. 参数 whence 可取为下列值:

- MPI_SEEK_SET — 将指针的位移设为 offset
- MPI_SEEK_CUR — 将指针的位移设为当前位移加上 offset
- MPI_SEEK_END — 将指针的位移设为文件结尾加上 offset

MPI_FILE_SEEK_SHARED 是聚合型函数, 进程组中所有进程必须同时调用并且提供相同的参数.

§7.6.2.2 查询共享文件指针的当前位移

C

```
int MPI_File_get_position_shared(MPI_File fh,
        MPI_Offset *offset)
```

Fortran

```
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERR)
INTEGER FH, IERR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

在参数 offset 返回共享文件指针的位移.

§7.6.3 文件位移在文件中的绝对地址

C

```
int MPI_File_get_byte_offset(MPI_File fh,
        MPI_Offset offset, MPI_Offset *disp)
```

Fortran

```
MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERR)
INTEGER FH, IERR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
```

该函数将以 etype 为单位相对于当前文件视窗的位移 (offset) 换算成以字节为单位从文件开头计算的绝对地址 (disp).

§7.7 不同进程对同一文件读写操作的相容性

当单个或多个进程同时对同一个文件进行访问时, MPI 称这些访问是相容的, 如果这些访问可以等效地被看成是以某种顺序依次进行的, 即便它们的先后顺序是不确定的. 换言之, 对同一个文件的多个访问是相容的, 如果它们中的任一访问都不会在操作过程中间由于被另一个访问打断或干扰而影响到访问的结果. MPI 系统允许用户将对一个文件的访问设置成具有“原子性”(atomicity, 意即“不可分的”) 来保证属于与该文件关联的进程组中的进程对该文件的访问的相容性. 但如果同一个文件分别被不同的进程组打开, 并且两个进程组中对该文件的访问存在冲突, 则用户必须通过在程序中调用 MPI_FILE_SYNC 函数以及同步函数 (MPI_BARRIER) 等来保证对文件访问的相容性与访问顺序.

§7.7.1 设定文件访问的原子性

C

```
int MPI_File_set_atomicity(MPI_File fh, int flag)
```

Fortran

```
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERR)
INTEGER FH, IERR
LOGICAL FLAG
```

该函数设定是否需要保证打开文件的进程组中进程对该文件的访问的原子性. 当 flag 为 true 时, MPI 系统将保证文件访问的原子性从而保证属于 (与该文件相关联的) 同一进程组的进程对该文件的访问的相容性. 而当 flag 为 false 时, MPI 不保证对文件访问的原子性, 而需要用户通过其它途径来保证对文件的不同访问间的相容性.

MPI_FILE_SET_ATOMICITY 是聚合型函数, 进程组中所有进程必须同时调用并且提供相同的参数.

例 7.2 在文件的同一位置上一个进程写、另一个进程读.

```
INTEGER STATUS(MPI_STATUS_SIZE), FH, A(10)
... ...
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'myfile',
+                  MPI_MODE_RDWR + MPI_MODE_CREATE,
+                  MPI_INFO_NULL, FH, IERR)
CALL MPI_FILE_SET_VIEW(FH, 0, MPI_INTEGER, MPI_INTEGER,
+                      'native', MPI_INFO_NULL, IERR)
CALL MPI_SET_ATOMICITY(FH, .TRUE., IERR)
IF ( MYRANK .EQ. 0 ) THEN
   DO I=1, 10
      A(I)= 5
   ENDDO
   CALL MPI_FILE_WRITE_AT(FH, 0, A, 10, MPI_INTEGER,
+                         STATUS, IERR)
ELSE IF ( MYRANK .EQ. 1 ) THEN
   CALL MPI_FILE_READ_AT(FH, 0, A, 10, MPI_INTEGER,
+                        STATUS, IERR)
ENDIF
```

在该例中, 因为 *atomicity* 被设为 true, 因此进程 *1* 将总是读到 *0* 个数或 *10* 个 *5*. 如果改变上面的程序将 *atomicity* 设为 false, 则进程 *1* 读到的结果是不确定的, 它与具体的 *MPI* 实现和程序运行过程有关.

§7.7.2 查询 atomicity 的当前值

C

```
int MPI_File_get_atomicity(MPI_File fh, int *flag)
```

Fortran

```
MPI_FILE_GET_ATOMICITY(FH, FLAG, IERR)
INTEGER FH, IERR
LOGICAL FLAG
```

该函数在参数 flag 中返回 atomicity 的当前值.

§7.7.3 文件读写与存储设备间的同步

C

```
int MPI_File_sync(MPI_File fh)
```

Fortran

```
MPI_FILE_SYNC(FH, IERR)
INTEGER FH, IERR
```

该函数确保将调用它的进程新近写入文件的数据写入存储设备. 如果文件在存储设备中的内容已被其它进程改变, 则调用该函数可确保调用它的进程随后读该文件时得到的是改变后的数据. 调用该函数时不能有尚未完成的对该文件的非阻塞型或分裂型读写操作.

注意, 如果打开文件的进程组的两个进程中一个进程往文件中写入一组数据, 另一个进程希望从文件的同一位置读到这组数据, 则各进程可能需要调用两次 MPI_FILE_SYNC, 并在两次调用间进行一次同步 (MPI_BARRIER). 第一次对 MPI_FILE_SYNC 的调用确保第一个进程写的数据被写入存储设备, 而第二次调用则可确保新写入存储设备的数据被另一个进程读到.

MPI_FILE_SYNC 是聚合型函数, 进程组中所有进程必须同时调用并且提供相同的参数.

§7.8 子数组数据类型创建函数

C

```
int MPI_Type_create_subarray(int ndims,
      int array_of_sizes[], int array_of_subsizes[],
      int array_of_starts[], int order,
      MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Fortran

```
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES,
+       ARRAY_OF_SUBSIZES, ARRAY_OF_STARTS, ORDER,
+       OLDTYPE, NEWTYPE, IERR)
INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
+       ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERR
```

这是一个辅助数据类型构造函数, 可用于对分布式数组的读写操作.

该函数创建一个“子数组”数据类型, 即描述一个 n 维 (全局) 数组中的一个 n 维子数组. 创建的新数据类型的域对应于全局数组, 类型中数据的位移由子数组的元素在全局数组中的位移确定.

参数 ndims 给出数组的维数. array_of_sizes[i] 给出全局数组第 i 维的大小. array_of_subsizes[i] 给出子数组第 i 维的大小. array_of_starts[i] 给出子数组第 i 维在全局数组中的起始位置 (不论 C 还是 Fortran 均用 0 表示从全局数组的第一个元素开始). 参数 order 给出数组元素的排列顺序, order = MPI_ORDER_C 表示数组元素按 C 的数组顺序排列, order = MPI_ORDER_FORTRAN 表示数组元素按 Fortran 的数组顺序排列. oldtype 给出数组元素的数据类型. newtype 返回所创建的子数组数据类型的句柄.

子数组各维的大小必须大于 0 并且小于或等于全局数组相应维的大小. 子数组的起始位置可以是全局数组中的任何位置, 但必须确保子数组被包含在全局数组中, 否则函数调用报错.

如果数据类型 oldtype 是可移植数据类型, 则新数据类型 newtype 也是可移植数据类型.

例 7.3 *我们在 Jacobi 迭代中增加两个用户输入的参数:* NSAVE *(整型) 和* RST *(逻辑型). 如果* NSAVE > 0*, 则每迭代* NSAVE *次和最后一次迭代后程序会将差分网格、方程的右端项以及当前近似解等数据写入文件* jacobi.dat *中 (如果* NSAVE ≤ 0 *则不存盘). 当* RST=.true. *时, 程序将从*

jacobi.dat 中读入的近似解作为初始值继续进行迭代, 而当 RST = .false. 时程序以 *0* 为初始值进行迭代.

如果使用的 *MPI* 系统不是 *MPICH-1.2.1,* 则可能需要将 jacobi5.cmm 中包含 mpiof.h 的一行说明掉. 在一些 *64* 位的系统上可能需要将子程序 READ_DATA 和 WRITE_DATA 中变量 OFFSET 的类型说明成 INTEGER*8 或 INTEGER(KIND=MPI_OFFSET_KIND) *(Fortran 90/95).*

COMMON 块文件*:*【jacobi5.cmm】

Fortran 77 程序*:*【jacobi5.f】