# 2004 年高性能计算培训班材料*

迟学斌　中国科学院计算机网络信息中心
张林波　中国科学院数学与系统科学研究院
莫则尧　北京应用物理与计算数学研究所

2005 年 7 月 24 日

*本文档及程序示例可从 *http://lssc.cc.ac.cn/training2004* 下载

# 目 录

# 第一部分

# 并行计算基础

# 第一章　并行计算基础知识

参见附件 1。

# 第二章　并行计算机体系结构

参见附件 2。

# 第三章　Linux 机群

本章介绍在微机环境中配置、安装 MPI 并行环境。为方便叙述，这里仅给出最简单的配置方法而暂不考虑其它问题，如网络性能、网络安全等等。

本章叙述的方法以 RedHat–9 (或 Fedora Core) 为例，如果使用其它 Linux 系统在一些细节上会略有差异。

## §3.1　引言

- Linux 机群系统已成为最流行的高性能计算平台，在高性能计算机中占有越来越大的比重
- 系统规模可从单机、少数几台联网的微机直到包括上千个结点的大规模并行系统
- 既可作为廉价的并行程序调试环境，也可设计成真正的高性能并行机
- 普及并行计算必不可少的工具
- 用于高性能计算的机群系统在结构上、使用的软件工具上通常有别于用于提供网络、数据库服务的机群 (后者亦称为服务器集群)
- *TOP500 Supercomputer Sites*
- *Cluster@TOP500*
- 参考资料: 用关键字 “cluster howto” 在网上搜索相关材料 ...

## §3.2　构建 Linux 机群的要素

- 单台或联网的多台微机或服务器
- Linux 系统: RedHat, Debian, SuSE, Mandrake, ...
- (可选) 高速内联网络: 千兆以太网，*Myrinet*, *QsNet*, *Dolphin SCI*, *Infiniband*, ...
- 编译系统: gcc/g++/g77, *PGI*, *Intel*, ...
- MPI 系统: *MPICH*, *LAM-MPI*, ...
- 网络文件系统: NFS, *PVFS*, *Lustre* ...
- 资源管理与作业调度: *PBS*, *Torque* (OpenPBS 改进版本), *Condor*, *LSF*, ...
- 数学库:

  **BLAS** *MKL*, *ATLAS*, *Kazushige Goto's BLAS* (推荐)

  **FFTW** *http://www.fftw.org/*

  **LAPACK** *http://www.netlib.org/lapack/*

  **ScaLAPACK** *http://www.netlib.org/scalapack/*

...

- 其它工具:

  **PETSc** *http://www-unix.mcs.anl.gov/petsc/petsc-2/*

  **UG** *http://cox.iwr.uni-heidelberg.de/~ug/*

  ...

## §3.3 几种典型的 Linux 机群结构

### §3.3.1 单台微机

- 可以有一个或多个处理器
- 安装 Linux 系统, C、Fortran 编译器, 以及 MPICH 或 LAM-MPI
- 模拟并行: 一个处理器上运行多个进程或线程
- 真实并行: 多个处理器上运行多个进程或线程
- 通过共享内存或 TCP/IP 进行通信

### §3.3.2 由几台日常使用的微机构成的机群

- 通常连接在同一个局域网, 通信通过网络进行
- 安装 Linux, 编译环境, MPICH 或 LAM-MPI
- 为方便使用, 最好设立一个共享的目录 (NFS), 以及一个 NIS 或 LDAP 服务器

### §3.3.3 专用并行机群

指专门建造的用于并行计算的机群。

- 通常单独形成一个局域网，再通过一个网关连接到 Internet
- 通常使用内部 IP 地址 (如 192.168.0.x), 对外部而言只有网关是可见的
- 可根据需要设立一至数台服务器，分别承担网关、时钟同步 (NTP)、NIS/LDAP、网络文件系统 (NFS)、资源管理、用户登录、作业调度等服务
- 通过 IP 伪装 (*IP Masquerading*) 或网络地址转换 (*NAT*, Network Address Translation) 使得内部结点能够直接访问 Internet (ipchains 或 iptables)
- 利用*LVS* (Linux Virtual Server) 将外部用户分配到登录结点
- 大型机群通常采用安装在机柜中的机架式或刀片式服务器，并有专门配备的 UPS 及空调
- 几个机群实例: CRack, *LSSC–I*, *LSSC–II*

## §3.4 在单机上安装、配置 MPI 并行环境

### §3.4.1 Linux 的安装

- 可以安装任何 Linux 发布版, 推荐 RedHat 的 Fedora Core
- 一些必须安装的包 (关于如何用 RedHat 的 `rpm` 命令安装软件包可参看*Maximum RPM*)

  `gcc` **包** GNU C, 用于 C 程序的编译

  `gcc-g77` **包** GNU Fortran 77, 用于 Fortran 程序的编译

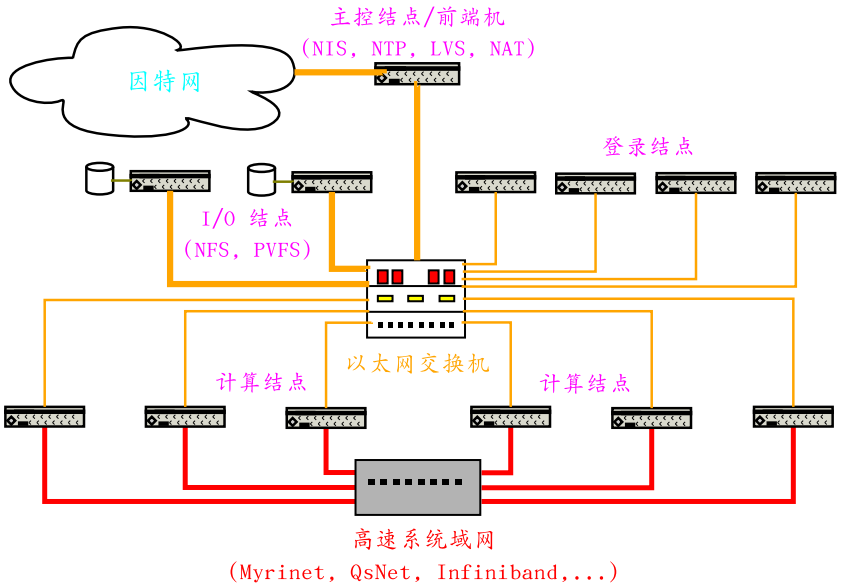



图 3.1: 典型 Linux 机群结构示意图

rsh 包和 rsh-server 包 用于启动 MPI 进程

- 配置 rsh-server, 下面的操作必须以 root 身份执行。
  - 编辑文件/etc/hosts.equiv, 在其中加上本机主机名 (单独占一行)。如果该文件不存在则创建一个。
  - 开启 rsh 服务:
    ```
    /sbin/chkconfig rsh on
    ```
  - 测试 rshd 的配置。以普通用户 (非 root) 登录并运行命令:
    ```
    rsh 主机名 /bin/hostname
    ```
    如果配置正确, 该命令应该显示本机主机名。如果出错，可以查看 /var/log/messages 文件中的错误信息。
  - 注意保证 shell 初始化文件 (.cshrc, .profile, .bashrc 等) 不要往 stdout 和 stderr 输出任何信息，即上述命令的输出除了主机名外不应该有任何其它内容。否则可能影响 MPI 进程的启动。

### §3.4.2 MPICH 的安装

MPICH 的手册在源程序的 doc 目录中。

- 从网址 *http://www-unix.mcs.anl.gov/mpi/mpich/* 处下载 MPICH 最新版本的源程序, 文件名为 mpich-1.x.x.tar.gz, 其中 1.x.x 为 MPICH 的版本号 (目前的最新版本为 1.2.5)。在下面描述的过程中需用 MPICH 的实际版本号替换 1.x.x
- 展开 MPICH 源程序:
  ```
  tar xzpvf mpich-1.x.x.tar.gz
  ```

- 配置、编译 MPICH:

```
cd mpich-1.x.x
./configure -prefix=/usr/local/mpi --disable-weak-symbols
make
```

上述命令生成的 MPICH 库应该使用 `ch_p4` 进行底层通信。选项 `--disable-weak-symbols` 的使用是为避免 MPICH 1.2.5 的一个 bug, 否则产生的库中将没有 `MPI_File_xxxx` 等函数 (只有 `PMPI_File_xxx` 等函数)

- 安装 MPICH:

```
make install
```

这一步必须以 `root` 身份执行, 它将 MPICH 的文件安装到目录 `/usr/local/mpi` 中。安装完毕后可以删除 MPICH 源文件目录。

- 分别将 "`/usr/local/mpi/bin`" 和 "`/usr/local/mpi/man`" 加入到环境变量 `PATH` 和 `MANPATH` 中。只需在目录 `/etc/profile.d` 中创建两个文件 `mpich.sh` 和 `mpich.csh`, 它们分别对 Bourne shell 和 C shell 起作用, 这两个文件的内容如下:

  – */etc/profile.d/mpich.sh*:

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

  – */etc/profile.d/mpich.csh*:

```
#!/bin/csh
if ( $?MANPATH ==  0 ) then
    setenv MANPATH :/usr/local/mpi/man
else
    setenv MANPATH ${MANPATH}:/usr/local/mpi/man
endif
setenv PATH ${PATH}:/usr/local/mpi/bin
```

- 测试 MPICH:

**C 程序**

```
cp /usr/local/mpi/examples/cpi.c .
mpicc -o cpi cpi.c .
mpirun -np 1 cpi
mpirun -np 2 cpi
```

**Fortran 程序**

```
cp /usr/local/mpi/examples/pi3.f .
mpif77 -o pi3 pi3.f .
mpirun -np 1 pi3
```

```
mpirun -np 2 pi3
```

(程序运行过程中依次输入 10000 和 0)。

注: 如果使用 MPICH–1.2.5, 则在单机上运行多个进程时需要用 '-all-local' 选项，即:

```
mpirun -all-local -np ...
```

## §3.5 在联网的多台机器上安装、配置 MPI 并行环境

Linux 系统的安装要求与前一节单机环境一样。此外，在开始下面的步骤之前还应该先配置好 TCP/IP 网络连接。为避免额外麻烦, 在配置网络时请不要开启任何防火墙设置。

此外，为方便相互访问，将所有机器的主机名放在 /etc/hosts 文件中。可以在所有机器上使用同样的 /etc/hosts 文件，它包含如下形式的内容:

```
127.0.0.1    localhost.localdomain   localhost
10.10.10.1   node1.mydomain  node1
10.10.10.2   node2.mydomain  node2
...          ...             ...
10.10.10.n   noden.mydomain  noden
```

(按真实情况替换其中的主机名和 IP 地址)。

下面介绍的方法使用*NIS* (Network Information Service, 也叫做 SUN Yellow Pages) 管理用户帐号，使用*NFS* (Network File System) 共享用户目录。

首先根据情况选择一台机器作为 NIS 和 NFS 服务器，我们将该机器称为服务结点或主结点机，而其它机器称为从结点机。主结点机和从结点机的配置方式不同，下面将分别介绍。

### §3.5.1 设置 NFS

**主结点机**

- 创建目录:

  ```
  mkdir -p /home/local
  ```

- 将 /usr/local 链接到 /home/local:

  ```
  /bin/rm -rf /usr/local
  ln -s /home/local /usr/.
  ```

  (注: 如果以前的 /usr/local 中安装有有用的文件, 则在执行上述命令前应该将其拷贝或移走)。

- 确认在主结点机上安装了 nfs-utils 包。开启 NFS 服务:

  ```
  /sbin/chkconfig nfs on
  /sbin/chkconfig nfslock on
  /etc/init.d/nfslock restart
  /etc/init.d/nfs restart
  ```

- 在文件 /etc/exports 中加入下面一行:

  ```
  /home  *(rw,no_root_squash)
  ```

  它将 /home 目录输出给所有机器。(注: 出于安全考虑可以限制仅将目录输出给指定的一些结点机, 及将 no_root_squash 改成 root_squash。请用命令 man 5 exports 查看相关参数的写法)。

- 输出指定目录 (/home):
  `exportfs -a`
  (也可重新启动系统)。

**从结点机**

- 创建目录:
  `mkdir -p /home`
- 在文件 /etc/fstab 中加入下面一行:
  `<主结点机名>:/home  /home  nfs  defaults 0 0`
  (将 <主结点机名> 替换成主结点机的主机名或 IP 地址)。
- 运行命令:
  `/sbin/chkconfig netfs on`
  它使得系统启动时自动挂接主结点机上的 /home 目录。
- 运行命令:
  `mount /home`
  (也可重新启动系统)。
- 将 /usr/local 链接到 /home/local:
  `/bin/rm -rf /usr/local`
  `ln -s /home/local /usr/.`
  (注: 如果以前的 /usr/local 中有有用的文件, 则在执行上述命令前应该将其拷贝或移走)。

上述所有操作必须以 root 身份执行。本步骤完成后，所有结点机上的 /home 和 /usr/local 目录的内容应该是一样的。在从结点机上可用命令 df 检查挂接情况。

有关 NFS 的进一步材料可在网上搜索 “NFS” 等关键字。

### §3.5.2 设置 NIS

以下描述中假设以 ‘CLUSTER’ 作为 NIS 域名。

**主结点机**

- 确认安装了下述包:
  ypserv, ypbind, yp-tools
- 在文件 /etc/sysconfig/network 中加入下面一行:
  `NISDOMAIN=CLUSTER`
- 开启 NIS 服务:
  `/sbin/chkconfig ypserv on`
  `/etc/init.d/ypserv start`
- 初始化 NIS 数据库:
  `/usr/lib/yp/ypinit -m`
  程序运行时按 Ctrl-D, 然后按 “y” 和回车。该命令将生成 NIS 数据库。可以忽略
  `No rule to make target ...`
  之类的错误信息。

- 开启 NIS 客户程序:
  `/sbin/chkconfig ypbind on`
  `/etc/init.d/ypbind start`
- 验证 NIS 设置
  - 命令 "`ypwhich`" 应该显示出主结点机主机名。
  - 命令 "`ypcat passwd`" 应该显示出 (主结点机上的) 用户帐号。

**从结点机**

- 确认安装了下述包:
  `ypbind`, `yp-tools`
- 在文件 `/etc/sysconfig/network` 中加入下面一行:
  `NISDOMAIN=CLUSTER`
- 开启 NIS 客户程序:
  `/sbin/chkconfig ypbind on`
  `/etc/init.d/ypbind start`
- 验证 NIS 设置
  - 命令 "`ypwhich`" 应该显示出主结点机主机名。
  - 命令 "`ypcat passwd`" 应该显示出主结点机上的用户帐号。
- 为了能够使用 NIS 用户登录，还需要修改 `/etc/nsswitch.conf` 文件，使其包含下述设置:

```
passwd: files nis
shadow: files nis
group:  files nis
hosts:  files nis dns
```

注意在从结点机上不要重复定义用户信息，即在从结点机上应该将所有 NIS 用户 (可用命令 "`ypcat passwd`" 显示出来) 从文件 `/etc/passwd` 和 `/etc/shadow` 中删除，将所有 NIS 组 (可用命令 "`ypcat group`" 显示出来) 从文件 `/etc/group` 中删除。

完成 NIS 配置后，创建新的用户帐号只需在主结点机上进行 (注意将用户的主目录放到 `/home` 下), 然后运行命令 "`cd /var/yp; make`" 即可。

如果在主结点机上修改了一个用户帐号信息，也应该运行一次上述命令以刷新 NIS 数据库。

NIS 用户在从结点机上不能用 "`passwd`" 命令修改用户口令，而必须用 "`yppasswd`" 命令来修改。

上述所有操作必须以 `root` 身份执行。

有关 NIS 的进一步材料可在网上搜索 "NIS" 等关键字。

### §3.5.3 设置 `rsh`

- 确认安装了 3.4.1 中列出的包。
- 将所主机名加到文件 `/etc/hosts.equiv` 中。
- 开启 `rsh` 服务:
  `/sbin/chkconfig rsh on`

注: 为使 root 用户能够用 rsh 执行远程命令，需将 /etc/hosts.equiv 文件拷贝到 /root/.rhosts, 并在文件 /etc/securetty 中加入 “rsh”。

上述操作必须在所有结点机上以 root 身份进行。

完成上述设置后, 在任何一台结点机上应该可以在所有结点机 (包括自己) 上执行远程命令。可按如下方法测试:

- 以一个 NIS 用户登录到一个结点机上并运行命令:
  rsh 另一台结点机名 /bin/hostname
  如果配置正确, 该命令应该显示对方的主机名。如果出错，可以查看对方 /var/log/messages 文件中的错误信息。

- 注意保证 shell 初始化文件 (.cshrc, .profile, .bashrc 等) 不要往 stdout 和 stderr 输出任何信息，即上述命令的输出除了主机名外不应该有任何其它内容。否则可能影响到 MPI 进程的启动。

### §3.5.4 MPICH 的安装

MPICH 的安装与单机下的安装完全一样，只需在主结点机上进行，因为所有结点机 /usr/local 目录是共享的。此外，需要将文件 /etc/profile.d/mpich.sh 和 /etc/profile.d/mpich.csh 拷贝到所有结点机上。

### §3.5.5 MPICH 程序的编译、运行

MPICH 程序的编译可以在任何一台结点机上用 mpicc (C), mpif77 (Fortran), mpiCC (C++) 等命令进行。它们是 MPICH 提供的 shell 脚本，用法与普通的 C/C++/Fortran 编译器一样。

MPICH 程序的运行方式取决于编译 MPICH 系统时选择的底层 driver。这里介绍的编译方式使用 ch_p4 作为底层 driver, 这种情况下有两种方法选择运行一个 MPI 程序使用的结点机和进程数, 即:

- mpirun -machinefile 文件名 -np 4 MPI程序名 [MPI程序参数]
  文件 文件名 中列出希望使用的结点机名，一行一个。mpirun 将在给定的结点机上启动指定数目的进程 (这里是 4)。当进程数目大于结点机数目时 mpirun 会在一些结点机上启动两个或更多进程。命令 “mpirun -help” 可以显示一个 mpirun 的简要使用说明。

- ./MPI程序名 -p4pg 文件名 [MPI程序参数]
  这种方式可以精确控制在每台结点机上启动的 MPI 进程数与进程序号，并且允许在不同结点机上启动不同的可执行文件 (适用于 Master/Slave 模式的并行程序)。文件 文件名 中按下列格式列出各结点机上启动的程序名:

```
结点机名1  0  可执行文件名1
结点机名2  1  可执行文件名2
结点机名3  1  可执行文件名3
...  ...
结点机名n  1  可执行文件名n
```

  其中 结点机名1 必须是运行该命令时所在的结点机, 可执行文件名1 必须与命令行上的 MPI程序名 为同一文件。所有 可执行文件名 必须使用绝对路径 (如 /home/zlb/test/cpi)。通常情况下，所有 可执行文件名 是一样的。而当同一 结点机名 出现多次时表示在该结点机上启动多个进程。

例如，假设用户在结点机 node1 的 /home/zlb/test 目录下，该目录中有已经编译好的 MPI 程序 cpi。在该目录下创建一个名为 p4file 的文件，它包含如下内容:

```
node1  0  /home/zlb/test/cpi
node2  1  /home/zlb/test/cpi
node1  1  /home/zlb/test/cpi
node2  1  /home/zlb/test/cpi
```

则命令 "./cpi -p4pg p4file" 将在 node1, node2 上运行四个进程, 其中进程 0 和进程 2 在 node1 上, 进程 1 和进程 3 在 node2 上。

## §3.6 专用并行机群系统

指专门用于并行计算的机群系统。这类系统的特点是结点机针对机群的用途经过专门选型，结点机配置比较统一。

- 结点机按照所负责的功能分成几类, 如 I/O 结点、计算结点、登录结点、服务结点等，同一类结点机通常具有相同的硬件配置和文件系统结构
- 一般单独构成一个局域网，使用内部网络地址
- 配备资源调度和作业管理系统
- 系统规模可以从数个结点到上千个结点

图 3.2 中给出一个包含 5 个结点的机群示例，它有 1 个主控结点，4 个计算结点。

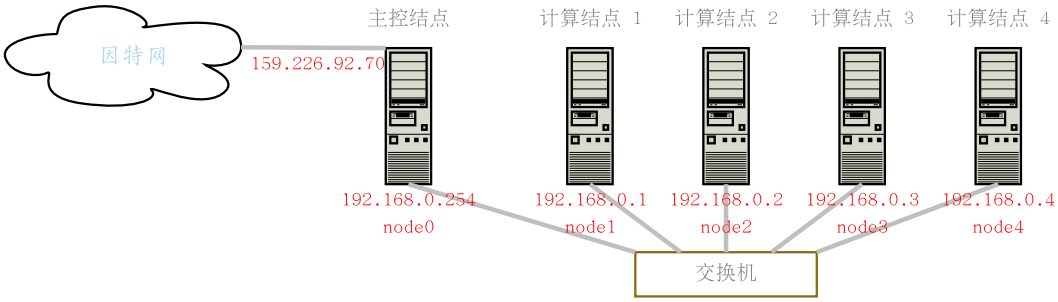



图 3.2: 小型专用机群示例

以图 3.2 的机群为例，假设 4 台计算结点的硬件配置完全一样，则它们安装的系统文件也基本一样，唯一区别是主机名和 IP 地址。在 RedHat Linux 中, 主机名由文件

/etc/sysconfig/network

设定，而假设以太网卡设备名为 eth0 的话则 IP 地址由文件

/etc/sysconfig/network-scripts/ifcfg-eth0

设定，这是计算结点文件系统间的唯一区别。为方便操作系统的安装与管理，我们希望所有计算结点上这两个文件也完全一样。为达到这一目的，一个办法是利用 DHCP 协议从主控结点获取 IP 地址，具体作法可参看, 例如, *http://www.linux.org/docs/ldp/howto/mini/DHCP/*。我们在这里介绍另一个简单方法，它将所有 MAC 地址放在文件 /etc/hosts 中, 计算结点启动时通过 MAC 地址得出自己的主机名和 IP 地址。下面是具体做法。

- /etc/hosts 文件:

```
127.0.0.1       localhost.localdomain localhost
192.168.0.254   node0   node0.cluster  00:90:27:57:42:64
192.168.0.1     node1   node1.cluster  00:90:27:57:45:F2
192.168.0.2     node2   node2.cluster  00:90:27:57:42:B4
192.168.0.3     node3   node3.cluster  00:90:27:57:45:E1
192.168.0.4     node4   node4.cluster  00:90:27:57:3F:7E
```

(最后一列应换成真实 MAC 地址，MAC 地址可在结点机上用命令 /sbin/ifconfig eth0 获得)

- /etc/sysconfig/network 文件:

```
NETWORKING=yes
hwaddr=‘/sbin/ifconfig eth0 | /bin/grep -i HWaddr | awk ’{print $5}’‘
HOSTNAME=‘grep ’^[0-9]’ /etc/hosts | grep -w $hwaddr | awk ’{print $2}’‘
unset hwaddr
GATEWAY=192.168.0.254
NISDOMAIN=CLUSTER
```

- /etc/sysconfig/network-scripts/ifcfg-eth0 文件:

```
DEVICE=eth0
BOOTPROTO=static
IPADDR=‘grep ’^[0-9]’ /etc/hosts | grep -w "$HOSTNAME" | awk ’{print $1}’‘
BROADCAST=192.168.0.255
NETMASK=255.255.255.0
NETWORK=192.168.0.0
ONBOOT=yes
```

按上述方法配置, 只需要安装好一台计算结点，其它计算结点可以简单地通过硬盘拷贝进行复制，也容易利用远程引导 (PXE boot) 实现自动安装。

## §3.7 组建大型机群系统需要考虑的一些问题

- 环境考虑: 布局、电源、空调等
- 结点机操作系统自动安装、恢复
- 结点机状态监控: CPU 负载、用户作业、内存使用等
- 结点机硬件状态监控: 电压、风扇、温度等 (*http://secure.netroedge.com/~lm78/*)
- MPI 系统进程数的限制
- rsh 端口数限制
- 并行程序的快速启动
- 文件系统、并行 I/O

# 第四章　矩阵并行计算

在科学与工程计算的许多问题中, 矩阵运算, 特别是矩阵相乘、求解线性方程组和矩阵特征值问题是最基本的内核. 随着MMP 并行计算机和网络并行环境的不断发展, 为了充分发挥分布式并行计算机的功效, 有必要对计算方法进行深入的研究. 这里着重考虑矩阵乘积和求解线性方程组的多种并行算法, 其向量化算法 [2, 3] 本书将不作介绍. 代数特征值问题及其相关问题的计算方法在文献 [4] 中有较详细的讨论, 这里将不作介绍. 为了讨论方便起见, 先作一些约定, 假设有 $p$ 个处理机, $P_j$ 表示第 $j$ 个处理机, $P_{\text{myid}}$ 表示当前运行程序的处理机, send($x$, $j$) 和recv($x$, $j$) 分别表示在 $P_{\text{myid}}$ 中把 $x$ 传送到 $P_j$ 和从 $P_j$ 中接收 $x$, 本章给出的算法都是在 $P_{\text{myid}}$ 处理机上的. 此外, 用 $i \bmod p$ 表示 $i$ 对 $p$ 取模运算.

程序设计与机器实现是密不可分的, 计算结果的好坏与编程技术有很大的关系, 尤其是在并行计算机环境下, 研制高质量的程序对发挥计算机的性能起着至关重要的作用. 我们结合算法研究给出在不同并行机上的一些重要例子来表明程序设计思想, 并采用程序方式描述算法, 以利于机器实现.

在并行算法的研究中, 经常要用到加速比的概念, 它是评价并行算法好坏的一个重要标准. 关于加速比的多种定义在第三章中已经给出, 这里把要用到的再强调一下, 用 $S_p$ 表示 $p$ 个处理机的加速比, 其定义如下:

$$S_p = \text{单机求解问题的执行时间}/p \text{ 个处理机求解问题的执行时间}$$

这个定义在应用中经常受到单机解题规模的限制, 也就是说, 在多处理机上能求解的问题在单机上是做不了的, 通常是问题规模增大时, 单机内存不够的缘故. 另一个容易改进的加速比的定义为

$$S_p = p \text{ 个处理机求解问题的运算速度}/\text{单机求解问题的运算速度}$$

如此定义的加速比对工作量确定的问题是容易得到的, 对于这里所考虑的问题如此定义的加速比是合适的, 但对工作量不确定, 单机又无法求解的问题这两种定义都不适合.

关于通信模式, 数据传输方式有许多不同的假设, 这样的假设在理论上讨论并行算法的加速比及效率是非常重要的. 对于消息传递型并行系统, 通常考虑其为: $T_c = \alpha + \beta \times N$, 其中 $\alpha$ 是启动时间, $\beta$ 是传输单位数据所需的时间, $N$ 是数据的传输量.

## §4.1　矩阵相乘的若干并行算法

矩阵乘积在实际应用中是经常要用到的, 许多先进的计算机上都配有高效的串行程序库. 为了在并行计算环境上实现矩阵乘积, 研究并行算法是非常必要的. 本节要考虑的计算问题是

$$C = A \times B \tag{4.1}$$

其中 $A$ 和 $B$ 分别是 $m \times k$ 和 $k \times n$ 矩阵, $C$ 是 $m \times n$ 矩阵. 不失一般性, 假设 $m = \bar{m} \times p$, $k = \bar{k} \times p$ 和 $n = \bar{n} \times p$, 下面考虑基于对矩阵 $A$ 和 $B$ 的不同划分的并行计算方法.

### §4.1.1　行列划分算法

这里将矩阵 $A$ 和 $B$ 分别划分为如下的行块子矩阵和列块子矩阵:

$$A = \begin{bmatrix} A_0^T & A_1^T & \ldots & A_{p-1}^T \end{bmatrix}^T, \quad B = \begin{bmatrix} B_0 & B_1 & \ldots & B_{p-1} \end{bmatrix} \tag{4.2}$$

这时 $C=(C_{i,j})=(A_i\times B_j)$, 其中 $C_{i,j}$ 是 $\bar{m}\times\bar{n}$ 矩阵. $A_i$, $B_i$ 和 $C_{i,j}$, $j=0,\ldots,p-1$ 存放在 $P_i$ 中, 这种存放方式使数据在处理机中不重复. 在本节所考虑的算法中, 都使用这种数据的存放方式. 由于使用 $p$ 个处理机, 每次每台处理机计算出一个 $C_{i,j}$, 计算 $C$ 需要 $p$ 次来完成. $C_{i,j}$ 的计算是按对角线进行的, 计算方法如下:

**算法 4.1.1.**

*for* $i=0$ *to* $p-1$ *do*
  $l\equiv i$+*myid mod p*
  $C_l=A\times B$, *mp1* $\equiv$ *myid*+*1 mod p, mm1* $\equiv$ *myid*-*1 mod p*
  *if* $i\neq p-1$, *send*($B$, *mm1*), *recv*($B$, *mp1*)
*end*{*for*}

在这个算法中, $C_l=C_{\mathrm{myid},l}$, $A=A_{\mathrm{myid}}$, $B$ 在处理机中每次循环向前移动一个处理机, 每次交换数据为 $k\times\bar{n}$ 矩阵, 交换次数为 $p-1$ 次. 如果用 $\mathrm{DTA}_1$ 表示在算法 4.1.1 中数据的交换量, $\mathrm{CA}_1$ 表示在算法 4.1.1 中的计算量, 则有 $\mathrm{DTA}_1=2\times k\times(n-\bar{n})$, $\mathrm{CA}_1=m\times k\times n/p$.

### §4.1.2 行行划分算法

这里将矩阵 $A$ 和 $B$ 均划分为行块子矩阵, 矩阵 $A$ 的划分同式 (4.2), $B$ 的划分如下:

$$B=\left[\begin{array}{cccc} B_0^T & B_1^T & \ldots & B_{p-1}^T \end{array}\right]^T \tag{4.3}$$

$C_i$ 为和 $A_i$ 相对应的 $C$ 的第 $i$ 个块, 进一步把 $A_i$ 按列分块与 $B$ 的行分块相对应, 记

$$A_i=\left[\begin{array}{cccc} A_{i0} & A_{i1} & \ldots & A_{i,p-1} \end{array}\right]$$

从而有

$$C_i=A_i\times B=\sum_{j=0}^{p-1}A_{i,j}\times B_j$$

初始数据 $A$, $B$ 和 $C$ 的存放方式与 4.1.1 相同, 在结点 $P_{\mathrm{myid}}$ 上的计算过程可归纳为算法 4.1.2.

**算法 4.1.2.**

*for* $i=0$ *to* $p-1$ *do*
  $l\equiv i$+*myid mod p*
  $C=C+A_l\times B$, *mp1* $\equiv$ *myid*+*1 mod p, mm1* $\equiv$ *myid*-*1 mod p*
  *if* $i\neq p-1$, *send*($B$, *mm1*), *recv*($B$, *mp1*)
*end*{*for*}

这个算法中的数据交换量和计算量与算法 4.1.1 相同, 所不同的只是计算 $C$ 的方式, 其中 $A_l=A_{myid,l}$.

### §4.1.3 列列划分算法

这里将矩阵 $A$ 和 $B$ 均划分为列块子矩阵, $B$ 的划分与式 (4.2) 相同, $A$ 划分为如下形式:

$$A=\left[\begin{array}{cccc} A_0 & A_1 & \ldots & A_{p-1} \end{array}\right] \tag{4.4}$$

这时 $C$ 的划分与 $B$ 的划分相对应, 也是按列分块的, 进一步把 $B_i$ 按行分成与 $A$ 的列分块相对应的行分块, 记 $B_i = \left[\begin{array}{cccc} B_{i0}^T & B_{i1}^T & \dots & B_{i,p-1}^T \end{array}\right]^T$, 从而有下面计算 $C$ 的方法.

$$C_j = A \times B_j = \sum_{i=0}^{p-1} A_i \times B_{i,j}$$

这时的计算过程是传送矩阵 $A$ 而不是 $B$, 具体的算法描述如下:

**算法 4.1.3.**

*for* $i = 0$ *to* $p-1$ *do*
  $l \equiv i$+*myid mod p*
  $C = C + A \times B_l$, *mp1* $\equiv$ *myid*+*1 mod p, mm1* $\equiv$ *myid*-*1 mod p*
  *if* $i \neq p-1$, *send(A, mm1), recv(A, mp1)*
*end*$\{for\}$

算法 4.1.3 的计算量与算法 4.1.1 和算法 4.1.2 是相同的, 算法 4.1.3 的数据交换量是 $\mathrm{DTA}_3 = 2 \times m \times (k - \bar{k})$. 当 $m \neq n$ 时, $\mathrm{DTA}_1 \neq \mathrm{DTA}_3$. 两种算法数据交换量的大小是由 $m$ 和 $n$ 决定的, 即当 $m \leq n$ 时, $\mathrm{DTA}_3 \leq \mathrm{DTA}_1$. 由于它们的计算量是相同的, 因此只要按通信量大小选择算法就可以得到好的并行效率.

### §4.1.4 列行划分算法

这里将矩阵 $A$ 和 $B$ 分别划分为列和行块子矩阵, $A$ 的划分与式 (4.4) 相同, $B$ 的划分与式 (4.3) 相同. 由此得到

$$C = A \times B = \sum_{i=0}^{p-1} A_i \times B_i$$

$C$ 的计算是通过 $p$ 个规模和 $C$ 相同的矩阵之和得到的, 从对问题的划分可以看出, 并行算法的关键是计算矩阵的和, 设计有效地计算矩阵和的算法, 对发挥分布式并行系统的效率起着重要作用. 为了使负载达到平衡, 我们给出如下的一种简单计算方法, 其中这里的 $C$ 与其它算法中的 $C$ 不同, 它不是分块的, 而是整个存放在所有的处理机中, 但在计算 $C$ 的和时采用与 4.1.1 相同的按行分块.

**算法 4.1.4.**

$C = A \times B$
*for* $i = 1$ *to* $p-1$ *do*
  $l \equiv i$+*myid mod p* $k \equiv$ *myid*-$i$ *mod p*
  *send*($C_l$, $l$), *recv*($T$, $k$)
  $C_{myid} = C_{myid} + T$
*end*$\{for\}$

当然在求和时有许多实现方法, 这里给出的算法简洁易懂, 其通信量 $\mathrm{DTA}_4 = 2 \times (m - \bar{m}) \times n$. 如果采用按列分块方式计算 $C$, 算法 4.1.4 也同样适合, 这时的通信量为 $\mathrm{DTA}_4 = 2 \times m \times (n - \bar{n})$, 由于 $m \times \bar{n}$ 和 $\bar{m} \times n$ 是相等的, 因此选择何种方式计算 $C$ 可根据需要而定.

### §4.1.5 Cannon 算法

假设矩阵 $A$, $B$ 和 $C$ 可以分成 $m \times m$ 块矩阵, 也即, $A = (A_{ij})_{m \times m}$, $B = (B_{ij})_{m \times m}$, 和 $C =$

$(C_{ij})_{m\times m}$, 其中 $A_{ij}$, $B_{ij}$ 和 $C_{ij}$ 是 $n\times n$ 矩阵, 进一步假定有 $p=m\times m$ 个处理机. 为了讨论 Cannon 算法, 引入块置换矩阵 $Q=(Q_{ij})$ 使得

$$Q_{ij}=\begin{cases} I_n, & j\equiv i+1 \bmod m \\ 0_n, & \text{其它情况} \end{cases}$$

其中 $I_n$ 和 $0_n$ 分别是 $n$ 阶单位矩阵和零矩阵. 定义对角块矩阵 $D_A^{(l)}=\mathrm{diag}(D_i^{(l)})=\mathrm{diag}(A_{i,i+l \bmod m})$, 容易推导出 $A=\sum_{l=0}^{m-1} D_A^{(l)}\times Q^l$. 因此

$$C=A\times B=\sum_{l=0}^{m-1} D_A^{(l)}\times Q^l\times B=\sum_{l=0}^{m-1} D_A^{(l)}\times B^{(l)}$$

其中 $B^{(l)}=Q^l\times B=Q\times B^{(l-1)}$. 利用这个递推关系式, 并把处理机结点编号从一维映射到二维, 即有 $P_{\mathrm{myid}}=P_{\mathrm{myrow,mycol}}$, 数据 $A_{ij}$,$B_{ij}$, 和 $C_{ij}$ 存放在 $P_{ij}$ 中, 容易得到下面的在处理机 $P_{\mathrm{myid}}$ 结点上的算法.

**算法 4.1.5.**

$C=0$
*for* $i=0$ *to* $m-1$ *do*
  $k\equiv$ *myrow+i mod m; mp1* $\equiv$ *mycol+1 mod m; mm1* $\equiv$ *mycol-1 mod m;*
  *if mycol=k then*
    *send(A, (myrow, mp1)); copy(A, tmpA);*
  *else*
    *recv(tmpA, (myrow, mm1));*
    *if mycol* $\neq$ *mp1, send(tmpA, (myrow, mp1));*
  *end*$\{if\}$
  $C=C+$*tmpA* $\times B$*; mp1* $\equiv$ *myrow+1 mod m; mm1* $\equiv$ *myrow-1 mod m;*
  *if* $i\neq m-1$ *then*
    *send(B, (mm1, mycol)); recv(B, (mp1, mycol));*
  *end*$\{if\}$
*end*$\{for\}$

该算法具有很好的负载平衡, 其特点是在同一行中广播 $A$, 计算出 $C$ 的部分值之后, 在同列中滚动 $B$. 数据交换量 $\mathrm{DTA}_5=m\times 2\times n^2+(m-1)\times 2\times n^2=2(2m-1)n^2=4m^2n^2/\sqrt{p}-2m^2n^2/p$. 由于计算量对每个处理机来说是相同的, 因此在选择算法时只需考虑通信量.

从所给出的这五个并行计算矩阵乘积的算法可以看到, 对于方阵的乘积, 当 $p\geq 4$ 时, Cannon 算法具有优越性. 在理论上这些算法的加速比当问题规模增大时都趋向于处理机个数, 它们可以由本章开始时给出的加速比的定义以及数据传送时间公式导出, 感兴趣的读者可把它们当成练习来做.

## §4.2 线性方程组的解法

线性方程组是许多重要问题的核心, 因此有效地求解线性方程组在科学与工程计算中是非常重要的. 并行计算机的问世, 使求解问题的速度和解题规模大幅度地提高, 同时也使计算方法产生了变化. 在传统的串行机上, Linpack 是求解线性方程组的有效软件包. 然而在并行机上求解此问题, 就需要设

计出适合于该机的并行算法, 算法的优劣会对并行机的效率产生很大的影响. 这里考虑的问题是

$$Ax = b$$

这里的任务可以分为两方面, 一方面是并行计算矩阵 $A$ 的$LU$ 分解, 其中 $L$,$U$ 分别是下三角和上三角矩阵, 也即存在一排列矩阵 $Q$, 使 $AQ = LU$. 另一方面是并行求解三角形方程组, 即, 求解方程组 $Ly = b$ 和 $Ux = y$. 下面我们描述有关的算法.

### §4.2.1 分布式系统的并行 $LU$ 分解算法

首先考虑 $n \times n$ 矩阵 $A = (a_{ij})$ 的串行$LU$ 分解法. 根据求解线性方程组的需要, 采用部分选主元的 Gauss 消去法, 进行列消去, 使得 $L$ 是单位下三角矩阵. 在算法中 $a_k$ 表示 $A$ 的第 $k$ 行.

**算法 4.2.1.**

*for* $j = 0$ *to* $n-2$ *do*
  *find* $l$: $|a_{lj}| = \max\{|a_{ij}|, i = j, \ldots, n-1\}$
  *if* $l \neq j$, *swap* $A_j$ *and* $A_l$
  *if* $a_{jj} = 0$, $A$ *is singular and return*
  $a_{ij} = a_{ij}/a_{jj}$, $i = j+1, \ldots, n-1$
  *for* $k = j+1$ *to* $n-1$ *do*
    $a_{ik} = a_{ik} - a_{ij} \times a_{jk}$, $i = j+1, \ldots, n-1$
  *end*$\{for\}$
*end*$\{for\}$

在算法 4.2.1 中, 主要计算工作量是修正矩阵 $A$, 即做 $a_{ik} - a_{ij} \times a_{jk}$, 因此并行计算的主要任务就是在多处理机上同时对矩阵 $A$ 的不同部分做修正. 在多处理机上 $LU$ 分解的重要工作是使载荷尽可能的平衡, 我们采用卷帘 (wrap) 存储方式在各处理机上分配矩阵 $A$, 把矩阵 $A$ 的第 $i$ 列存放在 $P_{i \bmod p}$ 中. 假设 $n = p \times m$, 在下面算法中 $A$ 的第 $i$ 列为原来 $A$ 的第 $i \times p + \text{myid}$ 列, 下面给出在 $P_{\text{myid}}$ 上的结点描述.

**算法 4.2.2.**

*icol*$= 0$
*for* $j = 0$ *to* $n-2$ *do*
  *if myid*$=j \bmod p$ *then*
    *find* $l$: $|a_{l,icol}| = \max\{|a_{i,icol}|, i = j, \ldots, n-1\}$
    *if* $l \neq j$, *swap* $a_{j,icol}$ *and* $a_{l,icol}$
    *if* $a_{j,icol} = 0$, $A$ *is singular and kill all processes*
    $a_{i,icol} = a_{i,icol}/a_{j,icol}$, $i = j+1, \ldots, n-1$
    $f_{i-j-1} = a_{i,icol}$, $i = j+1, \ldots, n-1$
    *send*$(l,$ *myid+1) and send*$(f,$ *myid+1)*
    *icol+1* $\rightarrow$ *icol*
  *else*
    *recv*$(l,$ *myid-1) and recv*$(f,$ *myid+1)*
    *if myid+1* $\neq j \bmod p$, *send*$(l,$ *myid+1) and send*$(f,$ *myid+1)*

*end*{*if*}
*if* $l \neq j$, *swap* $A_j$ *and* $A_l$
*for* $k$=*icol to* $m-1$ *do*
$a_{ik} = a_{ik} - f_i \times a_{jk}$, $i = j+1, \ldots, n-1$
*end*{*for*}
*end*{*for*}

算法 4.2.2 是在分布式并行计算机上做 $LU$ 分解的有效方法之一, 我们在国产并行机上做了许多实验, 效果很好. 如果采用分块卷帘方式存储, 增大了算法的粒度, 效果更好. 为减少在一个处理机上计算分解因子时其它处理机出现的等待, 可以采用提前计算下一次要用到的因子. 也即, 算法 4.2.2 中的条件判断由 if myid=$j \bmod p$ 变成 if myid=$j+1 \bmod p$, 首先只修正计算下一个因子要用到的列, 然后就计算分解因子, 这对于具有发送消息不需要等待的并行系统来说是非常适合的. 如果并行系统不具备这样的性质, 那就不要用这种改进的算法, 否则会事倍功半.

### §4.2.2 具有共享存储系统的并行 $LU$ 分解算法

共享存储系统是并行计算机的另一代表, 其上的算法设计与分布式系统的算法设计有很大的不同. 主要体现在通信方面, 共享存储系统数据的交换是通过共享变量实现的. 在做 $LU$ 分解时, 我们已经知道计算工作分为两个部分, 一是计算分解因子, 另一是修正矩阵 $A$. 由于需要交换的数据是每步的分解因子, 因此我们把分解因子定义成共享变量, 并用 gfactor 表示, 与此相对应的选主元的变量记为 gl, 它们的局部复制品用 lfactor 和 $l$ 表示, 同时用 flag 变量作为 gfactor 被用过与否的标志. 算法如下:

**算法 4.2.3.**

*icol*= 0
*for* $j = 0$ *to* $n-2$ *do*
*if myid*=$j \bmod p$ *then*
*find* $l$: $|a_{l,icol}| = \max\{|a_{i,icol}|, i = j, \ldots, n-1\}$
*if* $l \neq j$, *swap* $a_{j,icol}$ *and* $a_{l,icol}$
*if* $a_{j,icol} = 0$, $A$ *is singular and kill all processes*
$a_{i,icol} = a_{i,icol}/a_{j,icol}$, $i = j+1, \ldots, n-1$
$lfactor_{i-j-1} = a_{i,icol}$, $i = j+1, \ldots, n-1$
*if* $\sum_{k=0}^{p-1} flag_k \neq p$, *wait*
*copy(lfactor, gfactor), copy(l, gl),* $flag_k = 0$, $k = 0, \ldots, p-1$
*icol+1* $\rightarrow$ *icol*
*else*
*if* $flag_{myid} \neq 0$, *wait*
*copy(gfactor, lfactor), copy(gl, l),* $flag_{myid} = 1$
*end*{*if*}
*if* $l \neq j$, *swap* $A_j$ *and* $A_l$
*for* $k$ =*icol to* $m-1$ *do*
$a_{ik} = a_{ik} - lfactor_i \times a_{jk}$, $i = j+1, \ldots, n-1$
*end*{*for*}
*end*{*for*}

算法 4.2.3 与算法 4.2.2 在许多方面是相同的, 这里的 copy 代替了算法 4.2.2 中的 send 和 recv. 在这个算法中, 同样需要等待. 在这类系统上, 可以设计出几乎不需要等待的算法. 在算法 4.2.3 的基础上, 对于第 0 个处理机要首先计算出分解因子, 之后的计算过程由如下的修改算法给出:

**算法 4.2.4.**

*icol=0*
*if myid=0, icol=1*
*for* $j = 0$ *to* $n-2$ *do*
  *cid=*$j \bmod 2p$, *next=*$j+1 \bmod p$
  *if* $flag_{cid} \neq 0$, *wait*
  *copy(gfactor, lfactor), copy(gl, l)*
  *if myid=next then*
    *if* $l \neq j$, *swap* $a_{l,icol}$ *and* $a_{j,icol}$
    $a_{i,icol} = a_{i,icol} - lfactor_i \times a_{j,icol}$, $i = j+1, \ldots, n-1$
    *find gl:* $|a_{gl,icol}| = \max\{|a_{i,icol}|, i = j+1, \ldots, n-1\}$
    *if* $gl \neq j+1$, *swap* $a_{j+1,icol}$ *and* $a_{gl,icol}$
    *if* $a_{j+1,icol} = 0$, $A$ *is singular and kill all processes*
    $a_{i,icol} = a_{i,icol}/a_{j,icol}$, $i = j+2, \ldots, n-1$
    $gfactor_{i-j-2} = a_{i,icol}$, $i = j+2, \ldots, n-1$
    $flag_{cid+1 \bmod 2p} = 0$, $flag_{cid+2 \bmod 2p} = 1$
    *icol+1* $\rightarrow$ *icol*
  *end*$\{if\}$
  *if* $l \neq j$, *swap* $A_j$ *and* $A_l$
  *for* $k =$*icol to* $m-1$ *do*
    $a_{ik} = a_{ik} - lfactor_i \times a_{jk}$, $i = j+1, \ldots, n-1$
  *end*$\{for\}$
*end*$\{for\}$

这个算法的特点是计算分解因子时不是在修正全部矩阵之后进行的, 而是把当前要计算因子的列做修正就计算分解因子, 这样在下一次其它处理机做修正时就不需要等待, 实现了计算分解因子和修正矩阵在时间上的重叠, 这是在这类机器上做$LU$ 分解的一种理想算法, 详见文献 [5].

### §4.2.3 三角方程组的并行解法

本节考虑三角方程组的并行计算方法, 我们不妨仅讨论解下三角方程组 $Lx = b$. 三角方程组的并行求解对有效的并行求解线性方程组是不可缺少的, 它的并行效果的好坏对求解整个问题有直接的影响, 这里给出两种并行实现方法. 首先给出一个串行算法.

**算法 4.2.5.**

*for* $i = 0$ *to* $n-1$ *do*
  $x_i = b_i/l_{ii}$
  *for* $j = i+1$ *to* $n-1$ *do*
    $b_j = b_j - l_{ji} \times x_i$

*end*{*for*}
*end*{*for*}

在这个算法中每次对 $b$ 进行修正时用到 $L$ 的一列, 如果按这种方式并行修正 $b$, 则称之为列扫描方法. 对于列扫描算法, 原始数据 $L$ 适合于按行存放, 当修正 $b$ 的值时, 就可以并行计算. 同时为使每个处理机的工作量尽可能均衡, 要采取卷帘方式存放数据. 正如我们所描述的, 为了实现并行计算, 需要将每步计算出来的解的一个分量传送到所有其它处理机中, 其通信次数是很多的, 这对于消息传递型并行系统是不太适合的. 但是对于有共享存储的系统是可以采用这种计算方案的. 像前一节中做 $LU$ 分解时一样, 需要引进共享变量 flag 和 gb, 这时在共享存储系统上的算法可描述成如下形式:

**算法 4.2.6.**

$k = 0$
*for* $i = 0$ *to* $n - 1$ *do*
  *if* $myid \equiv i \ mod \ p$ *then*
    $gb_i = b_k / l_{ki}$, $flag_i = 1$, $k + 1 \to k$
  *end*{*if*}
  *if* $flag_i \neq 1$, *wait*
  *for* $j = k$ *to* $m - 1$ *do*
    $b_j = b_j - l_{ji} \times gb_i$
  *end*{*for*}
*end*{*for*}

这是一种非常自然的并行实现方法, 但是为减少等待, 可以采用修正 $b$ 的一个分量就计算下一个解的方式, 这样其它的处理机在做下一次修正时就已经有了要用的新值, 而不需要等待. 这些都是算法与程序设计时应该注意的技巧, 反复试验就会有所体会. 在大部分现有的共享存储并行系统上, 都提供了同步机制, 在算法中条件判断的等待也可以由系统提供的库函数来实现, 即在此同步. 但对采用异步方式计算下一个新解的算法就不适合了, 这时就应用我们在算法中给出的条件判断的等待来实现.

下面介绍一种在分布式并行机上的下三角方程组的求解方法, 该方法采用按列卷帘方式存放数据, 每次传递的是部分修正的右端项, 而不是新求出的解, 通过叠加的方式计算下次的新解. 这个算法在分布式系统上被广泛应用, 是非常有效的并行算法, 其形式如下:

**算法 4.2.7.**

$k = 0$
*if myid=0, then*
  $u_i = b_i$, $i = 0, \ldots, n - 1$, $v_i = 0$, $i = 0, \ldots, p - 2$
*else*
  $u_i = 0$, $i = 0, \ldots, n - 1$
*for* $i = 0$ *to* $n - 1$ *do*
  *if* $i > 0$, *recv*($v$, $i - 1 \ mod \ p$)
  $x_k = (u_i + v_0) / l_{ik}$
  $v_j = v_{j+1} + u_{i+1+j} - l_{i+1+j,k} \times x_k$, $j = 0, \ldots, p - 3$

$v_{p-2} = u_{i+p-1} - l_{i+p-1,k} \times x_k$
*send*$(v,\ i+1\ mod\ p)$
$u_j = u_j - l_{jk} \times x_k,\ j = i+p, \ldots, n-1$
$k+1 \to k$
*end*$\{for\}$

关于这个算法的详细讨论可参见文献 [6].

## §4.3 对称正定线性方程组的并行解法

对于对称正定矩阵 $A$ 的 $LU$ 分解, 我们采用Cholesky 分解, 也即 $A = R^T R$, 其中 $R$ 是上三角矩阵. 由于三角方程组的求解在前一节中已经给出, 因此这里着重考虑对称正定矩阵的Cholesky 分解. 一个方法是在传统的 Cholesky 分解列格式算法的基础上, 对于发送数据不需等待的并行系统, 提出的并行算法. 另一个方法是用双曲旋转变换的方式来做 Cholesky 分解. 下面就分别介绍这两种算法.

### §4.3.1 Cholesky 分解列格式的并行计算

这里给出的并行 Cholesky 分解算法, 是在传统的 Cholesky 分解列格式算法的基础上, 结合分布式并行计算机系统的特点给出的 [7]. 它是在分布式多处理机系统上求 Cholesky 分解的有效算法. 首先从串行算法出发

**算法 4.3.1.**

*for* $j = 0$ *to* $n-1$ *do*
$a_{jj} = a_{jj} - \sum_{k=0}^{j-1} a_{jk} \times a_{jk},\ a_{jj} = \sqrt{a_{jj}}$
*for* $i = j+1$ *to* $n-1$
$a_{ij} = (a_{ij} - \sum_{k=0}^{j-1} a_{jk} \times a_{ik})/a_{jj}$
*end*$\{for\}$
*end*$\{for\}$

在这个算法中, 矩阵 $R^T$ 存放在与矩阵 $A$ 相对应的下三角位置, 它对于 $j$ 循环来说, 每次计算出 $R^T$ 的一列, 故称之为列格式算法. 由于第 $j$ 列的计算用到前面的 $j$ 列的值, 因此在并行计算 $R$ 时就要把它之前的列的信息传送到该列所在的结点上. 在该串行算法的基础上, 引入分解因子变量 $F$, 它记录当前处理机之前的 $p-1$ 个处理机上的分解因子, 是 $(p-1) \times n$ 矩阵, 原始矩阵 $A$ 按行卷帘方式存放在处理机中, 则在结点 $P_{\text{myid}}$ 上的算法如下:

**算法** 2

for $i = 0$ to $m-1$ do
$k = i \times p + \text{myid},\ l = k - p + 1$
if $k > 0$ then, recieve $G$ from $P_{\text{myid}-1}$
for $j = 0$ to $p-2$ do
$a_{i,j+l} = (a_{i,j+l} - \sum_{t=0}^{j+l-1} a_{it} \times g_{jt})/g_{j,j+l}$
$F_j = G_{j+1}$

end{$for$}
$a_{ik} = a_{ik} - \sum_{t=0}^{k-1} a_{it} \times a_{it}$
$a_{ik} = \sqrt{a_{ik}}$, $F_{p-2} = A_i$
Send $F$ to $P_{\text{myid}+1}$
for $e = i+1$ to $m-1$ do
  for $j = 0$ to $p-2$ do
    $a_{e,j+l} = (a_{e,j+l} - \sum_{t=0}^{j+l-1} a_{et} \times g_{jt})/g_{j,j+l}$
  $a_{ek} = (a_{ek} - \sum_{t=0}^{k-1} a_{et} \times a_{it})/a_{ik}$
  end{$for$}
end{$for$}

这个并行算法的特点是每步计算出 $R^T$ 的 $p$ 列, 在同一循环中, 各处理机计算出的 $R^T$ 的 $p$ 列是不相同的, 从而实现计算与通信的异步进行, 减少处理机的等待. 理论分析与数值计算结果可参见文献 [7].

### §4.3.2 双曲变换 Cholesky 分解

双曲变换Cholesky 分解是指做 Cholesky 分解时使用如下的双曲变换:

$$H = \begin{bmatrix} \cosh\phi & \sinh\phi \\ \sinh\phi & \cosh\phi \end{bmatrix}$$

在我们的算法中, 使用下面的形式:

$$H = (1-\rho^2)^{-\frac{1}{2}} \begin{bmatrix} 1 & -\rho \\ -\rho & 1 \end{bmatrix}$$

其中 $\rho = \tanh(-\phi)$. 通过这个变换把矩阵 $A$ 化成 $R^T R$, 首先我们从它的基本理论开始.

假设 $A = D + U^T + U$, 其中 $D$ 是对角矩阵,$U$ 是严格上三角矩阵, 记 $W = D^{-1/2}U$, 和 $V = D^{1/2} + W$. 通过简单的推导有

$$A = V^T V - W^T W$$

从而有下面的关系式成立:

$$\begin{bmatrix} R^T & 0 \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix} = \begin{bmatrix} V^T & W^T \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \begin{bmatrix} V \\ W \end{bmatrix}$$

其中 $I$ 是 $n \times n$ 单位矩阵, 为做双曲变换 Cholesky 分解, 要用到下面的一些定义与引理.

**定义 4.3.1.** 如果一个 $2m \times 2m$ 矩阵 $\Theta$ 满足下面的关系式:

$$\Theta^T \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \Theta = \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix}$$

其中 $I$ 是 $m \times m$ 单位矩阵, 则称之为是伪正交的 *(pseudo-orthogonal)* 矩阵.

从这个定义可以看到, 如果存在一个伪正交矩阵 $Q$ 使得

$$Q\begin{bmatrix} V \\ W \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

那么从前面的关系式中就可得出 $A = R^T R$, 因此主要任务就是寻找伪正交矩阵 $Q$. 下面不加证明地列出文献 [8] 中的引理.

**引理 4.3.1.** *如果 $R$ 和 $S$ 都是 $n \times n$ 上三角矩阵, 使得 $R^T R - S^T S$ 对称正定, 则 $R$ 是可逆的, 并满足:*

$$| s_{kk} r_{kk}^{-1} | < 1, \qquad 1 \le k \le n$$

**引理 4.3.2.** *如果 $R$ 和 $S$ 都是 $n \times n$ 上三角矩阵, 使得 $R^T R - S^T S$ 对称正定, 并令 $\rho_k \equiv s_{kk} r_{kk}^{-1}$, $1 \le k \le n$, $\hat{Q} = \widetilde{Q}^{(n)} \widetilde{Q}^{(n-1)} \cdots \widetilde{Q}^{(1)}$, 其中 $\widetilde{Q}^{(k)}$ 的元素定义如下:*

$$\tilde{q}_{ij}^{(k)} = \begin{cases} 1, & i = j \ne k \text{ 或 } i = j \ne n+k \\ (1-\rho_k^2)^{-\frac{1}{2}}, & i = j = k \text{ 或 } i = j = n+k \\ -(1-\rho_k^2)^{-\frac{1}{2}} \rho_k, & (i,j) = (k, n+k) \text{ 或 } (i,j) = (n+k, k) \\ 0, & \text{其它} \end{cases}$$

*此外, 如果 $\begin{bmatrix} \widetilde{R} \\ \widetilde{S} \end{bmatrix} = \hat{Q} \begin{bmatrix} R \\ S \end{bmatrix}$, 则 $\hat{Q}$ 是伪正交矩阵, 并且 $\widetilde{R}$ 是上三角矩阵, $\widetilde{S}$ 是严格上三角矩阵.*

从 $\widetilde{Q}^{(k)}$ 的定义可以看到, 它是作用在 $R$ 和 $S$ 的第 $k$ 行的一个变换, 如果

$$H_k = (1-\rho_k^2)^{-\frac{1}{2}} \begin{bmatrix} 1 & -\rho_k \\ -\rho_k & 1 \end{bmatrix}$$

则 $H_k \begin{bmatrix} R_k \\ S_k \end{bmatrix}$ 与 $\widetilde{Q}^{(k)} \begin{bmatrix} R \\ S \end{bmatrix}$ 的第 $k$ 行和第 $k+n$ 行是相同的. 由此我们说 $\widetilde{Q}^{(k)}$ 是一个旋转变换.

假设 $Q$ 是 $n \times n$ 循环置换矩阵, 其中 $p_{1,n} = 1$ 和 $p_{i,i-1} = 1$, $2 \le i \le n$. 根据引理 2, 双曲变换 Cholesky 分解的算法可描述成如下:

**算法 4.3.2.**

$V^0 = V$, $W^0 = W$, $A = V^T V - W^T W$

*for* $i = 0$ *to* $n-1$ *do*

$$\begin{bmatrix} V^{i+1} \\ W^{i+1} \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & Q \end{bmatrix} \hat{Q}^{(i)} \begin{bmatrix} V^i \\ W^i \end{bmatrix}$$

*end* { *for* }

算法中 $\hat{Q}^{(i)}$ 与引理 4.3.2 中 $\hat{Q}$ 的定义相同, 这时的 $\hat{Q}^{(i)}$ 是把 $W_i$ 的对角线上的元素消为 0. 如果矩阵 $A$ 的逆是半带宽为 $\beta$ 的带状矩阵, 则算法 4.3.2 中的循环变量 $i$ 只需到 $\beta$ 就可计算出 $A$ 的 Cholesky 分解, 从而减少了计算时间, 详细的讨论请参见文献 [8]. 由于计算每个 $H_k$ 是相互独立的, 因此易于并行计算, 这里就不再详述了.

### §4.3.3 修正的双曲变换 Cholesky 分解

在算法 3 中 $\hat{Q}^{(i)}$ 是 $\widetilde{Q}^{(k)}$ 的乘积, 而每个 $\widetilde{Q}^{(k)}$ 只影响 $V^i$ 和 $W^i$ 的第 $k$ 行, 实际上就是双曲变换作用到一个 $2\times l$ 矩阵上, 假设 $M$ 是 $2\times l$ 矩阵, $H$ 是双曲变换. 我们的目的是在计算 $\overline{M}=HM$ 使得 $\bar{m}_{21}=0$ 的一系列计算过程中减少计算量.

为使 $\bar{m}_{21}=0$, $H$ 是容易计算的, 为此可以选择 $\rho=m_{21}m_{11}^{-1}$. 但是这需要开方运算和 $6\times l$ 次算术运算. 为达到不开方和减少算术运算的目的, 假设 $M=KB$, 其中 $K=\mathrm{diag}(K_1,K_2)$ 是 $2\times 2$ 正对角矩阵, 即 $K_1>0$ 和 $K_2>0$. 令 $G=\bar{K}^{-1}HK$, 其中 $\bar{K}$ 是 $2\times 2$ 对角矩阵. 如果 $\bar{B}=GB$, 则 $\bar{M}=\bar{K}\bar{B}$. 这里通过适当选取 $\bar{K}$, 以达到减少运算次数和开方运算的目的. 在这里我们并不直接计算 $\bar{K}$, 而是用它的平方形式. 假设 $L=K^2$, $\bar{L}=\bar{K}^2$, 则 $\bar{L}$ 的计算由下面的引理给出.

**引理 4.3.3.** 假设 $\alpha=\frac{L_2}{L_1}$, $\beta=\frac{b_{21}}{b_{11}}$. 如果选取 $\bar{L}=(1-\alpha\beta^2)^{-1}L$, 则

$$G=\begin{bmatrix}1 & -\alpha\beta\\ -\beta & 1\end{bmatrix}$$

*Proof.* 从 $H$ 的定义我们知道 $\rho=m_{21}m_{11}^{-1}=\frac{K_2b_{21}}{K_1b_{11}}$. 因此就有

$$HK=(1-\rho^2)^{-\frac{1}{2}}\begin{bmatrix}K_1 & -\frac{K_2^2b_{21}}{K_1b_{11}}\\ -\frac{K_2b_{21}}{b_{11}} & K_2\end{bmatrix}=(1-\alpha\beta^2)^{-\frac{1}{2}}K\begin{bmatrix}1 & -\alpha\beta\\ -\beta & 1\end{bmatrix}$$

引理得证. □

**引理 4.3.4.** 如果 $R$ 和 $S$ 都是 $n\times n$ 上三角矩阵, 并且 $E$ 和 $F$ 是对角矩阵, 使得 $R^TER-S^TFS$ 是正定的, 并假设 $\alpha_k=\frac{F_k}{E_k}$, $\beta_k=\frac{s_{kk}}{r_{kk}}$, 如果

$$\begin{bmatrix}\widetilde{R}\\ \widetilde{S}\end{bmatrix}=\hat{Q}\begin{bmatrix}R\\ S\end{bmatrix}$$

其中 $\hat{Q}=\widetilde{Q}^{(n)}\widetilde{Q}^{(n-1)}\cdots\widetilde{Q}^{(1)}$,$\widetilde{Q}^{(k)}$, $E$ 和 $F$ 的元素是如下定义的:

$$\tilde{q}_{ij}^{(k)}=\begin{cases}1, & i=j\\ -\alpha_k\beta_k, & i=k,j=n+k\\ -\beta_k, & i=n+k,j=k\\ 0, & \text{其它}\end{cases}$$

和

$$\widetilde{E}_k=\frac{E_k}{1-\alpha_k\beta_k^2},\qquad \widetilde{F}_k=\frac{F_k}{1-\alpha_k\beta_k^2},\qquad 1\le k\le n$$

则 $\widetilde{R}^T\widetilde{E}\widetilde{R}-\widetilde{S}^T\widetilde{F}\widetilde{S}=R^TER-S^TFS$, 并且 $\widetilde{R}$ 是上三角矩阵, $\widetilde{S}$ 是严格上三角矩阵.

这个引理的证明是容易的, 故此略去.

假设 $A=V^TEV-W^TFW$, 则修正的双曲 Cholesky 分解算法可被描述成如下形式:

**算法 4.3.3.**

$V^0=V$, $W^0=W$, $E^0=E$, $F^0=F$

*for* $i=0$ *to* $n-1$ *do*

$$\begin{bmatrix} V^{i+1} \\ W^{i+1} \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & P \end{bmatrix} \hat{Q}^{(i)} \begin{bmatrix} V^{i} \\ W^{i} \end{bmatrix}$$

$$E_k^{i+1} = \frac{E_k^i}{1-\alpha_k^i \beta_k^{i\,2}}, \qquad \widetilde{F}_k^{i+1} = \frac{F_k^i}{1-\alpha_k^i \beta_k^{i\,2}}, \qquad 1 \le k \le n$$

$$F_1^{i+1} = \widetilde{F}_n^{i+1}, \quad F_{k+1}^{i+1} = \widetilde{F}_k^{i+1}, \qquad 1 \le k \le n-1$$

*end* { *for* }

这里 $\hat{Q}^{(i)}$ 的定义与引理 4.3.4 中 $\hat{Q}$ 的定义相同. 这个算法与算法 4.3.2 一样都是易于并行实现的.

## §4.4 三对角方程组的并行解法

解三对角线性方程组在偏微分方程数值解中起着非常重要的作用, 因此已经有了很多关于它的并行算法, 这方面的工作可参见文献 [9]–[10]. 这里着重介绍几个重要的方法, 为不太熟悉并行计算的读者提供一些实例, 以便对进一步的并行计算打下基础. 在这里要求解的问题是 $Ax = d$, 其中

$$A = T(c,a,b) = \begin{bmatrix} a_0 & b_0 & & & \\ c_1 & a_1 & b_1 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-2} & a_{n-2} & b_{n-2} \\ & & & c_{n-1} & a_{n-1} \end{bmatrix}$$

### §4.4.1 递推法

在本章的第二节中, 我们已经介绍了如何做矩阵 $A$ 的 $LU$ 分解, 这里由于 $A$ 是三对角矩阵, 其 $LU$ 分解可表达成如下形式:

$$A = \begin{bmatrix} 1 & & & & \\ e_1 & 1 & & & \\ & \ddots & \ddots & & \\ & & e_{n-2} & 1 & \\ & & & e_{n-1} & 1 \end{bmatrix} \times \begin{bmatrix} f_0 & b_0 & & & \\ & f_1 & b_1 & & \\ & & \ddots & \ddots & \\ & & & f_{n-2} & b_{n-2} \\ & & & & f_{n-1} \end{bmatrix}$$

由此可以得出:

$$f_0 = a_0, \quad f_{i-1}e_i = c_i, \quad f_i = a_i - e_i b_{i-1}, \quad i = 1, \ldots, n-1$$

归纳整理有如下的非线性递推关系式:

$$f_i = a_i - c_i \times b_{i-1}/f_{i-1}, \quad i = 1, \ldots, n-1$$

如果令 $f_i = u_i/u_{i-1}$, $u_0 = f_0$, $u_{-1} = 1$, 则有下面的二阶线性递推关系:

$$\begin{bmatrix} u_i \\ u_{i-1} \end{bmatrix} = \begin{bmatrix} a_i & t_i \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u_{i-1} \\ u_{i-2} \end{bmatrix}, \quad t_i = c_i b_{i-1}, \quad i = 1, \ldots, n-1$$

我们从一阶线性递推关系式开始进行线性递推关系式的并行计算, 它具有如下形式:

$$x_0 = 0, \qquad x_i = a_i x_{i-1} + b_i, \qquad 1 \le i \le n$$

为了并行计算的需要 [11], 引入函数

$$Q(t,s)=\sum_{j=s}^{t}(\prod_{r=j+1}^{t}a_r)b_j$$

其中假定 $\prod_{r=t+1}^{t}a_r=1,\ 1\le s,t\le n,\ s\le t$. 如此定义的函数 $Q(t,s)$ 具有以下的性质:

(1) $Q(i,i)=b_i,\ 1\le i\le n$

(2) $\forall l\ge 0$, 使得 $s+l<t$, 不难证明下式成立:

$$Q(t,s)=(\prod_{r=s+l+1}^{t}a_r)Q(s+l,s)+Q(t,s+l+1)$$

(3) 由前面两个性质容易推出下面的关系式:

$$x_i=Q(i,1),\qquad 1\le i\le n$$

以上性质表明, 可以用递推倍增的方式并行计算一阶线性递推关系式. 以 $n=2^m$ 并行计算 $x_n=Q(n,1)$ 为例, 根据前述的性质不难得到

$$x_n=Q(n,1)=(\prod_{r=2^{m-1}+1}^{2^m}a_r)Q(2^{m-1},1)+Q(2^m,2^{m-1}+1)$$

在此分解中,$Q(2^{m-1},1)$ 和 $Q(2^m,2^{m-1}+1)$ 结构完全相同可同时计算, 而它们又可分别递推地继续按分解方式计算, 同时在计算它们的时间内还可以计算 $\prod_{r=2^{m-1}+1}^{2^m}a_r$. 因此并行计算 $x_1,\ldots,x_n$ 一般来说分别需要 $\lceil\log n\rceil$ 个乘法步和加法步, 也即, 并行计算所需的时间为 $O(\log n)$, 由于串行计算所需的时间为 $O(n)$, 故加速比为 $O(n/\log n)$. 这时使用的处理机个数为 $n$, 实际当中能使用的处理机个数远小于这个阶数, 因此需要适当的处理. 图 4.1 给出了在 $n=8$ 时并行计算 $x_1,\ldots,x_8$ 的流程图, 其中 $Q(i,i)=b_i,\ i=1,\ldots,8$.

前面给出的二阶线性递推关系式的做法与这里的做法完全相同, 只是这时的运算是二阶矩阵乘积, 因此我们知道如何用这种递推的方法求解三对角矩阵的 $LU$ 分解. 求解线性方程组需要再做二次一阶线性递推, 故可用同一种方法来求解此方程组, 有关的详细计算过程在此略去.

### §4.4.2 分裂法

本小节考虑对称正定三对角线性方程组的解法, 它基于对矩阵的分块, 故此称之为分裂法 [12, 10]. 其分解形式为如下的块三对角矩阵:

$$A=\begin{bmatrix}A_0 & B_0 & & & \\ B_0^T & A_1 & B_1 & & \\ & \ddots & \ddots & \ddots & \\ & & B_{p-3}^T & A_{p-2} & B_{p-2}\\ & & & B_{p-2}^T & A_{p-1}\end{bmatrix}$$

其中 $B_i$ 只有左下角的一个元素不为零,$A_i$ 是对称正定三对角矩阵. 因此可以对 $A_i$ 做 $LDL^T$ 分解, 这

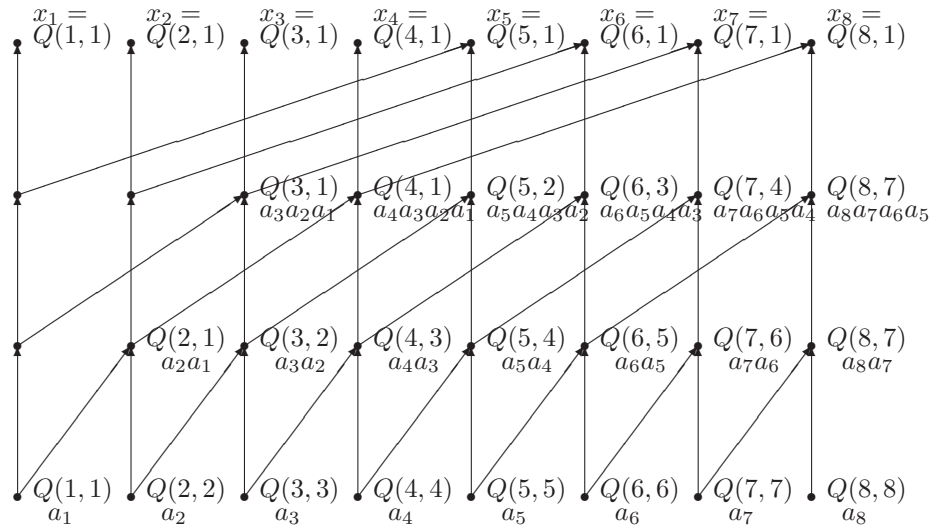



图 4.1: 递推法流程图

里 $D$ 是对角矩阵,$L$ 是单位下三角阵. 假设 $A_i = L_iD_iL_i^T$, 令 $L = \mathrm{diag}(L_i)$, 则有

$$L^{-1}AL^{-T} = \begin{bmatrix} D_0 & \bar{B}_0 & & & \\ \bar{B}_0^T & D_1 & \bar{B}_1 & & \\ & \ddots & \ddots & \ddots & \\ & & \bar{B}_{p-3}^T & D_{p-2} & \bar{B}_{p-2} \\ & & & \bar{B}_{p-2}^T & D_{p-1} \end{bmatrix}$$

其中 $\bar{B}_i = L_i^{-1}B_iL_{i+1}^{-T} = B_iL_{i+1}^{-T}$, 它是除最后一行外均为 0 的矩阵. 由于 $D_i$ 是对角矩阵, 可以分别把 $\bar{B}_i$ 的最后一行除最后一个元素外均消为 0, 记消去后的矩阵为 $\widetilde{D}_i$ 和 $\widetilde{B}_i$, 从而 $\widetilde{D}_i$ 和 $\widetilde{B}_i$ 以及 $\widetilde{B}_i^T$ 的最后一个元素构成一个新的小的三对角线性方程组. 对于这个小的线性方程组可在一台机器上来求解, 把这个解传送到所有的处理机中, 就可求出原问题的解. 在这个方法中, 首先要用到的是 $LDL^T$ 分解, 这个分解可由类似于上一节中的 $R^TR$ 分解的方法得到, 这里只需用串行的分解方法, 故不再具体列出 $LDL^T$ 的串行分解方法. 综上所述, 我们可给出下述算法:

**算法 4.4.1.**

*(1)* 计算 $L_i$, $D_i$, 使 $A_i = L_iD_iL_i^T$;

*(2)* 对矩阵做变换, 并对 $B_i$ 做前述的消去;

*(3)* 形成小的三对角线性方程组, 并求解之;

*(4)* 求解整个问题.

这个算法具有很好的并行性, 是目前被认为求解这类问题最有效的算法. 通过简单的计算可以得出, 该算法的并行计算复杂性比串行计算复杂性增加了一倍, 虽然增加了并行性, 但计算复杂性的增加降低了算法的效率. 目前由于缺少更有效的三对角方程组的并行计算方法, 该算法在求解此问题时仍被广泛采用. 由于它是分块在每个处理机上独立进行大量的运算, 因此这个算法不难应用到块三对角线性方程组.

## §4.5 异步并行迭代法

异步迭代算法在并行计算中起着重要的作用, 因为此类算法不需要处理机之间的等待, 使处理机的工作效率能够得到充分的发挥. 这方面的研究工作早在 60 年代就已经开始, 文献 [13] 中给出了线性迭代 $x = Bx + c$ 的收敛定理, 当谱半径 $\rho(|B|) < 1$ 时, 此迭代过程是异步迭代收敛的. 继此之后, 文献 [14] 给出了非线性迭代 $x = Fx$ 的收敛定理, 当 $F$ 是 $P$- 收缩映射时, 此迭代过程是异步迭代收敛的. $P$- 收缩映射 (P-contraction) 的定义参见文献 [15].

### §4.5.1 异步并行迭代法基础

首先引入关于 $x = Fx$ 的异步迭代算法的定义, 最后给出文献 [14] 中的一个结论. 记 $|A|$ 为 $A$ 的每个元素取绝对值的矩阵, $|x|$ 表示对向量 $x$ 的分量取绝对值的向量, $A \geq 0$ 表示 $A$ 的元素均大于等于 0. $Fx$ 的第 $i$ 个分量记为 $f_i(x)$ 或 $f_i(x_1, \ldots, x_n)$, 向量序列记为 $x^{(j)}$, $j = 0, 1, \ldots$, 所有非负整数的集合记为 $N$.

**定义 4.5.1.** *设 $F$ 是 $R^n \to R^n$ 的映射, 则关于算子 $F$ 和初始点 $x^{(0)}$ 的异步迭代是由下述递推关系定义的向量序列 $x^{(j)} \in R^n$, $j = 1, 2, \ldots$,*

$$J = \{J_j | j = 1, 2, \ldots\}$$

$$S = \{(s_1^{(j)}, \ldots, s_n^{(j)}) \,|\, j = 1, 2, \ldots\}$$

$$x_i^{(j)} = \begin{cases} x_i^{(j-1)}, i \notin J_j \\ f_i(x_1^{(s_1^{(j)})}, \ldots, x_n^{(s_n^{(j)})}), i \in J_j \end{cases}$$

*其中, $J$ 是 $\{1, 2, \ldots, n\}$ 的非空子集构成的序列, $S$ 是 $N^n$ 中的一个序列. 此外, 对每个 $i = 1, \ldots, n$, $J$ 和 $S$ 满足如下三个条件:*

*(1) $s_i^{(j)} \leq j - 1$, $j = 1, 2, \ldots$;*

*(2) $s_i^{(j)}$ 作为 $j$ 的函数趋于无穷大;*

*(3) $i$ 在集合 $J_j$ $(j = 1, 2, \ldots)$ 中出现无穷多次.*

下边要用到文献 [14] 中的一个重要结果, 故此我们以引理的形式给出.

**引理 4.5.1.** *假设 $|Fx - Fy| \leq A|x - y|$, 其中 $A$ 是非负矩阵并且 $\rho(A) < 1$. 则迭代 $x = Fx$ 是异步收敛的.*

### §4.5.2 线性迭代的一般收敛性结果

在这一小节中, 考虑求解线性方程组 $Ax = b$ 的一些迭代法的收敛性. 对于线性迭代法, 通常采用 $A$ 的分裂形式, $A = B - C$, 这时的迭代形式如下:

$$x = B^{-1}Cx - B^{-1}b \tag{4.5}$$

其中 $B$ 是可逆的. 由引理 4.5.1 可知, 当 $\rho(|B^{-1}C|) < 1$ 时, 上述的迭代过程是异步迭代收敛的. 下面我们就对 $A$ 是 $M$ 矩阵或对角占优矩阵的情况讨论其收敛性, 这些结果容易推广到分块 $M$ 矩阵或 $H$ 矩阵, 关于这些矩阵的定义可参见文献 [15]. 下面不加证明地给出文献 [15] 中的一些结论. 首先给出弱正则分裂的定义.

**定义 4.5.2.** 设 $A$, $B$ 和 $C$ 是实矩阵, 如果 $A = B - C$, $B^{-1} \geq 0$ 和 $B^{-1}C \geq 0$, 则称 $A = B - C$ 是 $A$ 的弱正则分裂.

**引理 4.5.2.** 设 $A = B - C$ 是一弱正则分裂, 则 $\rho(B^{-1}C) < 1$ 当且仅当 $A^{-1}$ 存在并且 $A^{-1} \geq 0$.

**引理 4.5.3.** 设 $B$ 和 $C$ 是 $n \times n$ 阶矩阵, 如果 $|C| \leq B$, 则 $\rho(C) \leq \rho(B)$.

**引理 4.5.4.** 设 $A$ 是严格或不可约对角占优矩阵, 如果 $a_{ij} \leq 0$ $(i \neq j)$, 且 $a_{ii} > 0$, 则 $A$ 是 $M$ 矩阵.

下面我们不加证明地引用文献 [16] 中的两个结论.

**引理 4.5.1.** 设 $A = B - C$ 是 $A$ 的弱正则分裂且 $A$ 是 $M$ 矩阵, 则迭代形式 *(4.5)* 是异步迭代收敛的.

这个定理可应用到许多迭代法中, 比如 *Jacobi* 和 *Gauss-Seidel* 型迭代法等. 对于这种类型的迭代法, 其矩阵 $A$ 分解成 $A = D - L - U$, 其中 $D$ 是对角矩阵,$L$ 和 $U$ 分别是严格下和上三角矩阵. 基于这种分裂形式的迭代过程的异步迭代收敛性可用下面的定理给出.

**引理 4.5.2.** 设 $A$ 是严格或不可约对角占优矩阵,$A = B - C$, 其中 $B = D - \alpha L$, $C = (1 - \alpha)L + U$, $0 \leq \alpha \leq 1$, 则迭代形式 *(4.5)* 是异步迭代收敛的.

这些结果在求解偏微分方程的差分离散的方程中有很大的作用, 因为在采用区域分解法的时候, 信息的交换可以不需要等待, 所以可以提高并行处理机的效率.

# 第二部分

# MPI 并行程序设计

# 第五章　消息传递并行程序设计平台 MPI

MPI 是英文 Message Passing Interface 的缩写, 是基于消息传递编写并行程序的一种用户界面. 消息传递是目前并行计算机上广泛使用的一种程序设计模式, 特别是对分布式存储的可扩展的并行计算机 SPCs (Scalable Parallel Computers) 和工作站机群 NOWs (Networks of Workstations) 或 COWs (Clusters of Workstations). 尽管还有很多其它的程序实现方式, 但是过程之间的通信采用消息传递已经是一种共识. 在 MPI 和 PVM 问世以前, 并行程序设计与并行计算机系统是密切相关的, 对不同的并行计算机就要编写不同的并行程序, 给并行程序设计和应用带来了许多麻烦, 广大并行计算机的用户迫切需要一些通用的消息传递用户界面, 使并行程序具有和串行程序一样的可移植性.

在过去的 4 年中, 国际上确定了 MPI 为消息传递用户界面标准, 自从 1994 年 6 月推出 MPI 以来, 它已被广泛接受和使用, 目前国际上推出的所有并行计算机都支持 MPI 和 PVM. 对于使用 SPCs 的用户来说, 编写 SPMD 并行程序使用 MPI 可能更为方便. 在使用 MPI 函数与常数时要注意, MPI 的所有函数与常数均以 `MPI_` 开头. 在 C 程序中, 所有常数的定义除下划线外一律由大写字母组成, 在函数和数据类型定义中, 接 `MPI_` 之后的第一个字母大写, 其余全部为小写字母, 在所有函数调用之后都将返回一个错误信息码, 对于 Fortran 程序, MPI 函数全部以过程方式调用, 其错误码以哑元参数返回. 此外, MPI 是按进程组 (Process Group) 方式工作的, 所有 MPI 程序在开始时均被认为是在通信子 `MPI_COMM_WORLD` 所拥有的进程组中工作, 之后用户可以根据自己的需要, 建立其它的进程组. 此外还需注意的是, 所有 MPI 的通信一定要在通信子 (communicator) 中进行. 此处我们选择了使用 MPI 编程中常用的函数作介绍, 并结合并行计算的实际需要, 给出了大量的 Fortran 程序示例, 以便增强对 MPI 函数的理解, 关于 MPI 的详细介绍可参见文献 [1].

## §5.1　MPI 并行环境管理函数

在编写 MPI 并行程序中, 必不可少的二个函数是 `MPI_Init( )` 和 `MPI_Finalize( )`. 它们的详细定义如下:

MPI_INIT

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

由于 ANSI C 的主程序 main 接受参数 `argc` 和 `argv`, 如此使用 `MPI_Init` 可将这些参数传送到每个进程中. 在 Fortran 程序中, `MPI_INIT` 只返回一个错误码. 这个函数初始化 MPI 并行程序的执行环境, 它必须在调用所有其它 MPI 函数 (除 `MPI_INITIALIZED`) 之前被调用, 并且在一个 MPI 程序中, 只能被调用一次.

MPI_FINALIZE

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

这个函数清除 MPI 环境的所有状态. 即一但它被调用, 所有 MPI 函数都不能再调用, 其中包括 MPI_INIT.

在一个程序中, 如果不清楚是否已经调用了 MPI_INIT, 可以使用 MPI_INITIALIZED 来检查, 它是唯一的可以在调用 MPI_INIT 之前使用的函数.

MPI_INITIALIZED

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

参数说明

| | | |
|---|---|---|
| OUT | FLAG, | 如果 MPI_INIT 被调用, 返回值为 TRUE, 否则为 FALSE. |

另外在 MPI 中还提供了一个用来检查出何种错误的函数 MPI_ERROR_STRING, 它将给出错误信息.

MPI_ERROR_STRING

| | |
|---|---|
| C | `int MPI_Error_string( int errorcode, char *string, int *len )` |
| Fortran | `MPI_ERROR_STRING( ERRORCODE, STRING, LEN, IERROR )`<br>`INTEGER ERRORCODE, LEN, IERROR`<br>`CHARACTER*(*) STRING` |

参数说明

| | | |
|---|---|---|
| IN | ERRORCODE, | 由 MPI 函数返回的错误码. |
| OUT | STRING, | 对应 ERRORCODE 的错误信息. |
| OUT | LEN, | 错误信息 STRING 的长度. |

在使用此函数时, 应注意 STRING 的长度最小应为 MPI_MAX_ERROR_STRING.

## §5.2 进程控制函数

在这一章中, 介绍与进程组有关的一些基本函数, 其中包括如何建立进程组和通信子, 灵活使用这些函数在实际程序设计中会带来巨大方便.

## §5.3 MPI 进程组操作函数

MPI_COMM_GROUP

| | |
|---|---|
| C | `int MPI_Comm_group( MPI_Comm comm, MPI_Group *group )` |
| Fortran | `MPI_COMM_GROUP( COMM, GROUP, IERROR )`<br>`INTEGER COMM, GROUP, IERROR` |

| 参数说明 | | |
|---|---|---|
| IN | COMM, | 通信子. |
| OUT | GROUP, | 对应 COMM 的进程组. |

MPI_COMM_GROUP 这个函数是用来建立一个通信子对应的新进程组, 之后就可以对此进程组进行需要的操作.

MPI_GROUP_FREE

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

| 参数说明 | | |
|---|---|---|
| INOUT | GROUP, | 释放进程组并返回 MPI_GROUP_NULL. |

当 MPI_GROUP_FREE 被调用之后, 任何关于此进程组的操作将视为无效.

MPI_GROUP_SIZE

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

| 参数说明 | | |
|---|---|---|
| IN | GROUP, | 进程组. |
| OUT | SIZE, | 进程组中的进程个数. |

如果进程组是 MPI_GROUP_EMPTY, 则返回值 SIZE 为 0.

MPI_GROUP_RANK

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

| 参数说明 | | |
|---|---|---|
| IN | GROUP, | 进程组. |
| OUT | RANK, | 进程在进程组中的编号. |

如果进程不是进程组中的成员, 则返回值 RANK 为 MPI_UNDEFINED.

MPI_GROUP_TRANSLATE_RANKS

| | |
|---|---|
| C | int MPI_Group_translate_ranks( MPI_Group group1, int n, int *ranks1,<br>MPI_Group group2, int *ranks2) |
| Fortran | MPI_GROUP_TRANSLATE_RANKS( GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR )<br>INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR |

| 参数说明 | | |
|---|---|---|
| IN | GROUP1, | 进程组 1. |
| IN | N, | RANKS1 和 RANKS2 中数组元素个数. |
| IN | RANKS1, | 进程组 1 中有效编号组成的数组. |
| IN | GROUP2, | 进程组 2. |
| OUT | RANKS2, | RANKS1 中的元素在进程组 2 中的对应编号. |

如果属于进程组 1 的某个进程可以在 RANKS1 中找到, 而这个进程不属于进程组 2, 则在 RANKS2 中对应 RANKS1 的位置返回值为 MPI_UNDEFINED.

MPI_GROUP_INCL

| | |
|---|---|
| C | int MPI_Group_incl( MPI_Group group, int n, int *ranks, MPI_Group newgroup) |
| Fortran | MPI_GROUP_INCL( GROUP, N, RANKS, NEWGROUP, IERROR )<br>INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR |

| 参数说明 | | |
|---|---|---|
| IN | GROUP, | 进程组. |
| IN | N, | RANKS 数组中元素的个数和新进程组的大小. |
| IN | RANKS, | 将在新进程组中出现的旧进程组中的编号. |
| OUT | NEWGROUP, | 由 RANKS 定义的顺序导出的新进程组. |

MPI_GROUP_EXCL

| | |
|---|---|
| C | int MPI_Group_excl( MPI_Group group, int n, int *ranks, MPI_Group newgroup) |
| Fortran | MPI_GROUP_EXCL( GROUP, N, RANKS, NEWGROUP, IERROR )<br>INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR |

| 参数说明 | | |
|---|---|---|
| IN | GROUP, | 进程组. |
| IN | N, | RANKS 数组中元素的个数. |
| IN | RANKS, | 在新进程组中不出现的旧进程组中的编号. |
| OUT | NEWGROUP, | 旧进程组中不在 RANKS 里的元素组成的新进程组. |

MPI_GROUP_UNION

| | |
|---|---|
| C | int MPI_Group_union( MPI_Group group1, MPI_Group group2, MPI_Group newgroup) |
| Fortran | MPI_GROUP_UNION( GROUP1, GROUP2, NEWGROUP, IERROR )<br>INTEGER GROUP1, GROUP2, NEWGROUP, IERROR |

| 参数说明 | | |
|---|---|---|
| IN | GROUP1, | 进程组 1. |
| IN | GROUP2, | 进程组 2. |
| OUT | NEWGROUP, | 进程组 1 和进程组 2 的并. |

MPI_GROUP_INTERSECTION

| | |
|---|---|
| C | int MPI_Group_intersection( MPI_Group group1, MPI_Group group2, MPI_Group newgroup) |
| Fortran | MPI_GROUP_INTERSECTION( GROUP1, GROUP2, NEWGROUP, IERROR )<br>INTEGER GROUP1, GROUP2, NEWGROUP, IERROR |

| 参数说明 | | |
|---|---|---|
| IN | GROUP1, | 进程组 1. |
| IN | GROUP2, | 进程组 2. |
| OUT | NEWGROUP, | 进程组 1 和进程组 2 的交. |

MPI_GROUP_DIFFERENCE

| | |
|---|---|
| C | int MPI_Group_difference( MPI_Group group1, MPI_Group group2, MPI_Group newgroup) |
| Fortran | MPI_GROUP_DIFFERENCE( GROUP1, GROUP2, NEWGROUP, IERROR )<br>INTEGER GROUP1, GROUP2, NEWGROUP, IERROR |

| 参数说明 | | |
|---|---|---|
| IN | GROUP1, | 进程组 1. |
| IN | GROUP2, | 进程组 2. |
| OUT | NEWGROUP, | 进程组 1 和进程组 2 的差. |

以上关于进程组操作的函数功能, 和数学中集合运算相同.

## §5.4 MPI 通信子操作

下面介绍的关于通信子的操作和上述关于进程组操作十分类似.

MPI_COMM_SIZE

| | |
|---|---|
| C | int MPI_Comm_size( MPI_Comm comm, int *size ) |
| Fortran | MPI_COMM_SIZE( COMM, SIZE, IERROR )<br>INTEGER COMM, SIZE, IERROR |

| 参数说明 | | |
|---|---|---|
| IN | COMM, | 通信子. |
| OUT | SIZE, | 通信子中的进程个数. |

MPI_COMM_RANK

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

参数说明

| | | |
|---|---|---|
| IN | `COMM`, | 通信子. |
| OUT | `RANK`, | 通信子中的进程编号. |

MPI_COMM_DUP

| | |
|---|---|
| C | `int MPI_Comm_dup( MPI_Comm comm, MPI_Comm *newcomm )` |
| Fortran | `MPI_COMM_DUP( COMM, NEWCOMM, IERROR )`<br>`INTEGER COMM, NEWCOMM, IERROR` |

参数说明

| | | |
|---|---|---|
| IN | `COMM`, | 通信子. |
| OUT | `NEWCOMM`, | `COMM` 通信子的复制. |

MPI_COMM_CREATE

| | |
|---|---|
| C | `int MPI_Comm_create( MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm )` |
| Fortran | `MPI_COMM_CREATE( COMM, GROUP, NEWCOMM, IERROR )`<br>`INTEGER COMM, GROUP, NEWCOMM, IERROR` |

参数说明

| | | |
|---|---|---|
| IN | `COMM`, | 通信子. |
| IN | `GROUP`, | 通信子 `COMM` 的一个子集. |
| OUT | `NEWCOMM`, | 对应 `GROUP` 的新通信子. |

MPI_COMM_SPLIT

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

参数说明

| | | |
|---|---|---|
| IN | `COMM`, | 通信子. |
| IN | `COLOR`, | 子集控制值. |
| IN | `KEY`, | 子集中进程编号的顺序. |
| OUT | `NEWCOMM`, | 由此产生的新通信子. |

这个函数划分 COMM 所对应的进程组为不相交的子进程组, 每个子进程组有一个共同的值 COLOR, 就是说, 每个子进程组中包含 COLOR 相同的所有进程.

MPI_COMM_FREE

| | |
|---|---|
| C | `int MPI_Comm_free( MPI_Comm *comm )` |
| Fortran | `MPI_COMM_FREE( COMM, IERROR )`<br>`INTEGER COMM, IERROR` |

参数说明

| | | |
|---|---|---|
| INOUT | COMM, | 通信子. |

## §5.5 点到点通信函数

点到点通信是指一对进程之间的数据转换, 也就是说, 一边发送数据另一边接收数据. 点到点通信是 MPI 通信机制的基础, 它分为同步通信和异步通信二种机制.

## §5.6 阻塞式通信函数

MPI_SEND

| | |
|---|---|
| C | `int MPI_Send( void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )` |
| Fortran | `MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR )`<br>`<type> BUF(*)`<br>`INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR` |

参数说明

| | | |
|---|---|---|
| IN | BUF, | 所要发送消息数据的首地址. |
| IN | COUNT, | 发送消息数组元素的个数. |
| IN | DATATYPE, | 发送消息的数据类型. |
| IN | DEST, | 接收消息的进程编号. |
| IN | TAG, | 消息标签. |
| IN | COMM, | 通信子. |

这里 COUNT 是 BUF 的元素个数而不是字节数.

MPI_RECV

| | |
|---|---|
| C | `int MPI_Recv( void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status )` |
| Fortran | `MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR )`<br>`<type> BUF(*)`<br>`INTEGER COUNT, DATATYPE, SOURCE, TAG,COMM,STATUS(MPI_STATUS_SIZE), IERROR` |

| 参数说明 | | |
|---|---|---|
| OUT | BUF, | 接收消息数据的首地址. |
| IN | COUNT, | 接收消息数组元素的最大个数. |
| IN | DATATYPE, | 接收消息的数据类型. |
| IN | SOURCE, | 发送消息的进程编号. |
| IN | TAG, | 消息标签. |
| IN | COMM, | 通信子. |
| OUT | STATUS, | 接收消息时返回的状态. |

在这个接收函数中, 可以不指定 SOURCE 和 TAG, 而分别用 MPI_ANY_SOURCE 和 MPI_ANY_TAG 来代替, 用于接收任何进程发送的消息或任何编号的消息. 接收消息时返回的状态 STATUS 在 C 语言中是用结构定义的, 在 Fortran 中是用数组定义的, 其中包括 MPI_SOURCE, MPI_TAG 和 MPI_ERROR. 此外 STATUS 还包含接收消息元素的个数, 但它不是显式给出的, 需要用到后面给出的函数 MPI_GET_COUNT.

MPI 的基本数据类型定义与相应的 Fortran 和 C 的数据类型对照关系如下:

Fortran 程序中可使用的 MPI 数据类型

| MPI datatype | Fortran datatype |
|---|---|
| MPI_INTEGER | INTEGER |
| MPI_REAL | REAL |
| MPI_DOUBLE_PRECISION | DOUBLE PRECISION |
| MPI_COMPLEX | COMPLEX |
| MPI_LOGICAL | LOGICAL |
| MPI_CHARACTER | CHARACTER(1) |
| MPI_BYTE | |
| MPI_PACKED | |

C 程序中可使用的 MPI 数据类型

| MPI datatype | Fortran datatype |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED_INT | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | |
| MPI_PACKED | |

这里 MPI_BYTE 和 MPI_PACKED 不对应 Fortran 或 C 中的任何数据类型, MPI_BYTE 是由一个字节组成的, 而 MPI_PACKED 将在后面介绍. 到目前为止, 我们已经可以编写一些简单的 MPI 程序, 下面给出一

个 MPI 程序的例子, 我们结合这个例子说明一个 MPI 程序的组成部分. 重点需要掌握的 MPI 函数有初始化并行环境的函数 MPI_INIT, 得到本进程的编号函数 MPI_COMM_RANK, 得到全部进程个数的函数 MPI_COMM_SIZE, 退出 MPI 并行环境的函数 MPI_FINALIZE, 以及进行消息传递的二个函数 MPI_SEND 和 MPI_RECV.

**例 5.6.1.** 假设一共有 $p$ 个进程, 在进程编号为 myid ($\texttt{myid} = 0, \ldots, p-1$) 的进程中有一个整数 $m$, 我们要把 $m$ 传送到进程 $(myid+1)\ mod\ p$ 中.

```
      program ring
c
c The header file mpif.h must be included when you use MPI fuctions.
c
      include 'mpif.h'
      integer myid, p, mycomm, ierr, m, status(mpi_status_size),
     &        next, front, mod, n
c
c Create MPI parallel environment and get the necessary data.
c
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
c
c Beginning the main parallel work on each process.
c
      m = myid
      front = mod(p+myid-1, p)
      next = mod(myid+1, p)
c
c Communication with each other.
c
      if(myid .eq. 0) then
        call mpi_recv(n, 1, mpi_integer, front, 1, mycomm, status, ierr)
        call mpi_send(m, 1, mpi_integer, next, 1, mycomm, ierr)
        m = n
      else
        call mpi_send(m, 1, mpi_integer, next, 1, mycomm, ierr)
        call mpi_recv(m, 1, mpi_integer, front, 1, mycomm, status, ierr)
      endif
c
c Ending of parallel work.
c
      print *, 'The value of m is ', m, ' on Process ', myid
      call mpi_comm_free(mycomm, ierr)
c
c Remove MPI parallel environment.
c
      call mpi_finalize(ierr)
      end
```

MPI_GET_COUNT

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

参数说明

| | | |
|---|---|---|
| IN | STATUS, | 接收消息时返回的状态. |
| IN | DATATYPE, | 接收消息的数据类型. |
| OUT | COUNT, | 接收消息数组元素的个数. |

由于在接收消息时使用的是最大个数, 为了准确知道接收消息的个数, 就要使用此函数.

MPI_SENDRECV

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

参数说明

| | | |
|---|---|---|
| IN | SENDBUF, | 所要发送消息数据的首地址. |
| IN | SENDCOUNT, | 发送消息数组元素的个数. |
| IN | SENDTYPE, | 发送消息的数据类型. |
| IN | DEST, | 接收消息的进程编号. |
| IN | SENDTAG, | 发送消息标签. |
| OUT | RECVBUF, | 接收消息数据的首地址. |
| IN | RECVCOUNT, | 接收消息数组元素的最大个数. |
| IN | RECVTYPE, | 接收消息的数据类型. |
| IN | SOURCE, | 发送消息的进程编号. |
| IN | RECVTAG, | 接收消息标签. |
| IN | COMM, | 通信子. |
| OUT | STATUS, | 接收消息时返回的状态. |

这是发送消息和接收消息组合在一起的一个函数, 好处是不用考虑先发送还是先接收消息. 在示例 5.6.1 中从通信开始到通信结束部分可用如下的一个函数来完成:

```
      call mpi_sendrecv(m, 1, mpi_integer, next, 1, n, 1, mpi_integer,
     &                  front, 1, mycomm, status, ierr)
      m = n
```

由此可见使用 MPI_SENDRECV 可以使程序简化, 更重要的是在通信过程中不需要考虑哪个进程先发送还是先接收, 从而可以避免消息传递过程中的死锁.

MPI_SENDRECV_REPLACE

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

参数说明

| | | |
|---|---|---|
| OUT | BUF, | 发送和接收消息数据的首地址. |
| IN | COUNT, | 发送和接收消息数组元素的个数. |
| IN | DATATYPE, | 发送和接收消息的数据类型. |
| IN | DEST, | 接收消息的进程编号. |
| IN | SENDTAG, | 发送消息标签. |
| IN | SOURCE, | 发送消息的进程编号. |
| IN | RECVTAG, | 接收消息标签. |
| IN | COMM, | 通信子. |
| OUT | STATUS, | 接收消息时返回的状态. |

在这个函数中, 发送和接收使用同一个消息缓冲区 BUF. 因此示例 5.6.1 的通信部分可进一步改写成如下的形式:

```
      call mpi_sendrecv_replace(m, 1, mpi_integer, next, 1,
     &                          front, 1, mycomm, status, ierr)
```

在进行环形通信时, 使用 MPI_SENDRECV_REPLACE 是非常方便的. 为了更方便使用此函数, MPI 定义了一个空进程 MPI_PROC_NULL, 如果通信采用这个空进程将不起任何作用. 比如在示例 5.6.1 中, 如果我们不需要从第 $p-1$ 个进程向第 0 个进程传送 $m$, 则可用如下方式来实现:

```
      ......
      if(myid .eq. 0) front = MPI_PROC_NULL
      if(myid .eq. p-1) next = MPI_PROC_NULL
      call mpi_sendrecv_replace(m, 1, mpi_integer, next, 1,
     &                          front, 1, mycomm, status, ierr)
```

MPI_PROBE

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

| 参数说明 | | |
|---|---|---|
| IN | SOURCE, | 发送消息进程的编号. |
| IN | TAG, | 接收消息的标签. |
| IN | COMM, | 通信子. |
| OUT | STATUS, | 返回到达消息的状态. |

此函数与下面给出的 MPI_IPROBE 的功能基本类似, 但是 MPI_PROBE 一定要等到消息才返回.

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

| 参数说明 | | |
|---|---|---|
| IN | SOURCE, | 发送消息进程的编号. |
| IN | TAG, | 接收消息的标签. |
| IN | COMM, | 通信子. |
| OUT | FLAG, | 如果指定消息已经达到, FLAG 返回值为 TRUE. |
| OUT | STATUS, | 返回到达消息的状态. |

这个函数对由 SOURCE, TAG 和 COMM 确定的消息是否到达, 它都将立即返回. 这里 SOURCE 和 TAG 同 MPI_RECV 中一样, 可以用通配符 (wildcard) MPI_ANY_SOURCE 和 MPI_ANY_TAG 来代替.

## §5.7 非阻塞式通信函数

非阻塞式通信函数是指在通信过程中, 不需要等待通信结束就返回, 通常这种通信过程交由计算机的后台来处理. 如果计算机系统提供硬件支持非阻塞式通信函数, 就可以使计算与通信在时间上的重叠, 从而提高并行计算的效率.

| MPI_ISEND | |
|---|---|
| C | int MPI_Isend( void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request ) |
| Fortran | MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR )<br><type> BUF(*)<br>INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR |

参数说明

| | | |
|---|---|---|
| IN | BUF, | 所要发送消息数据的首地址. |
| IN | COUNT, | 发送消息数组元素的个数. |
| IN | DATATYPE, | 发送消息的数据类型. |
| IN | DEST, | 接收消息的进程编号. |
| IN | TAG, | 消息标签. |
| IN | COMM, | 通信子. |
| OUT | REQUEST, | 请求句柄以备将来查询. |

MPI_IRECV

| | |
|---|---|
| C | `int MPI_Irecv( void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request )` |
| Fortran | `MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR )`<br>`<type> BUF(*)`<br>`INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR` |

参数说明

| | | |
|---|---|---|
| OUT | BUF, | 接收消息数据的首地址. |
| IN | COUNT, | 接收消息数组元素的个数. |
| IN | DATATYPE, | 接收消息的数据类型. |
| IN | SOURCE, | 发送消息的进程编号. |
| IN | TAG, | 消息标签. |
| IN | COMM, | 通信子. |
| OUT | REQUEST, | 请求句柄以备将来查询. |

MPI_ISEND 和 MPI_IRECV 不需要等待发送或接收消息完成就可执行其余的任务, 看发送或接收过程是否结束, 有下面的一些函数:

MPI_WAIT

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

参数说明

| | | |
|---|---|---|
| INOUT | REQUEST, | 请求句柄. |
| OUT | STATUS, | 发送或接收消息的状态. |

如果 REQUEST 所指的操作已经完成, MPI_WAIT 将结束等待状态.

MPI_TEST

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

| 参数说明 | | |
|---|---|---|
| INOUT | REQUEST, | 请求句柄. |
| OUT | FLAG, | REQUEST 所指的操作已经完成返回值为 TRUE. |
| OUT | STATUS, | 发送或接收消息的状态. |

以上这二个函数中 STATUS 对发送消息操作是没有定义的, 唯一可在发送消息操作中使用 STATUS 的是查询函数 MPI_TEST_CANCELLED.

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

| 参数说明 | | |
|---|---|---|
| INOUT | REQUEST, | 请求句柄, 返回值为 MPI_REQUEST_NULL. |

以上是对单个 REQUEST 进行查询的函数, 如果要对多个请求句柄进行查询, 有下面的一些函数可以使用.

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

| 参数说明 | | |
|---|---|---|
| IN | COUNT, | 请求句柄的个数. |
| INOUT | ARRAY_OF_REQUESTS, | 请求句柄数组. |
| OUT | INDEX, | 已经完成通信操作的句柄指标. |
| OUT | STATUS, | 消息的状态. |

这个函数当所有请求句柄中至少有一个已经完成通信操作, 就返回, 如果有多于一个请求句柄已经完成, MPI_WAITANY 将随机选择其中的一个并立即返回.

| MPI_TESTANY | |
|---|---|
| C | int MPI_Testany( int count, MPI_Request *array_of_requestS,<br>int *index, int *flag, MPI_Status *status ) |
| Fortran | MPI_TESTANY( COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR )<br>LOGICAL FLAG<br>INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX,<br>STATUS(MPI_STATUS_SIZE), IERROR |

| 参数说明 | | |
|---|---|---|
| IN | COUNT, | 请求句柄的个数. |
| INOUT | ARRAY_OF_REQUESTS, | 请求句柄数组. |
| OUT | INDEX, | 已经完成通信操作的句柄指标. |
| OUT | FLAG, | 如果有一个已经完成, 则 FLAG=TRUE. |
| OUT | STATUS, 消息的状态. | |

这个函数无论有没有通信操作完成都将立即返回.

MPI_WAITALL

```
C       int MPI_Waitall( int count, MPI_Request *array_of_requests,
                         MPI_Status *array_of_statuses )
Fortran MPI_WAITALL( COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR )
        INTEGER COUNT, ARRAY_OF_REQUESTS(*),
                ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR
```

| 参数说明 | | |
|---|---|---|
| IN | COUNT, | 请求句柄的个数. |
| INOUT | ARRAY_OF_REQUESTS, | 请求句柄数组. |
| INOUT | ARRAY_OF_STATUSES, | 所有消息的状态数组. |

这个函数当所有通信操作完成之后才返回, 否则将一直等待.

MPI_TESTALL

```
C       int MPI_Testall( int count, MPI_Request *array_of_requests,
                         int *flag, MPI_Status *array_of_statuses )
Fortran MPI_TESTALL( COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR )
        LOGICAL FLAG
        INTEGER COUNT, ARRAY_OF_REQUESTS(*),
                ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR
```

| 参数说明 | | |
|---|---|---|
| IN | COUNT, | 请求句柄的个数. |
| INOUT | ARRAY_OF_REQUESTS, | 请求句柄数组. |
| OUT | FLAG, | 如果有一个没完成, 则 FLAG=FALSE. |
| INOUT | ARRAY_OF_STATUSES, | 所有消息的状态数组. |

这个函数无论所有通信操作是否完成都将立即返回.

MPI_CANCEL

```
C       int MPI_Cancel( MPI_Request *request )
Fortran MPI_CANCEL( REQUEST, IERROR )
        INTEGER REQUEST, IERROR
```

| 参数说明 | | |
|---|---|---|
| INOUT | REQUEST, | 请求句柄. |

这个函数用来取消一个发送或接收操作, 其后要使用 MPI_WAIT 或 MPI_TEST 等函数, 因此使用此函数会损失许多效率. 但是要知道关于 REQUEST 请求句柄的操作是否已经被取消, 还要使用下面的检测函数:

MPI_TEST_CANCELLED

| | |
|---|---|
| C | `int MPI_Test_cancelled( MPI_Status *status, int *flag )` |
| Fortran | `MPI_TEST_CANCELLED( STATUS, FLAG, IERROR )`<br>`LOGICAL FLAG`<br>`INTEGER REQUEST, IERROR` |

| 参数说明 | | |
|---|---|---|
| IN | STATUS, | 消息的状态. |
| OUT | FLAG, | 如果已经取消, 则 FLAG=TRUE. |

## §5.8 特殊的点到点通信函数

MPI_SEND_INIT

| | |
|---|---|
| C | `int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request )` |
| Fortran | `MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR )`<br>`<type> BUF(*)`<br>`INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR` |

| 参数说明 | | |
|---|---|---|
| IN | BUF, | 所要发送消息数据的首地址. |
| IN | COUNT, | 发送消息数组元素的个数. |
| IN | DATATYPE, | 发送消息的数据类型. |
| IN | DEST, | 接收消息的进程编号. |
| IN | TAG, | 消息标签. |
| IN | COMM, | 通信子. |
| OUT | REQUEST, | 请求句柄以备将来查询. |

MPI_RECV_INIT

| | |
|---|---|
| C | `int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype,int source, int tag, MPI_Comm comm, MPI_Request *request )` |
| Fortran | `MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR )`<br>`<type> BUF(*)`<br>`INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR` |

| 参数说明 | | |
|---|---|---|
| OUT | BUF, | 接收消息数据的首地址. |
| IN | COUNT, | 接收消息数组元素的个数. |
| IN | DATATYPE, | 接收消息的数据类型. |
| IN | SOURCE, | 发送消息的进程编号. |
| IN | TAG, | 消息标签. |
| IN | COMM, | 通信子. |
| OUT | REQUEST, | 请求句柄以备将来查询. |

MPI_SEND_INIT 和 MPI_RECV_INIT 是二个持久性通信函数, 它们的参数和前面讲到的异步通信中的发送与接收相同, 由此可以把通信以通道方式建立起来对应关系, 从而提高通信效率. 但是仅有这二个函数还不能够达到通信的目的, 它们还需要用 MPI_START 或 MPI_STARTALL 来激活.

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

| 参数说明 | | |
|---|---|---|
| INOUT | REQUEST, | 请求句柄. |

在使用 MPI_START 之后, MPI_SEND_INIT 和 MPI_RECV_INIT 就同异步发送与接收.

| MPI_STARTALL | |
|---|---|
| C | int MPI_Startall( int count, MPI_Request *array_of_request ) |
| Fortran | MPI_STARTALL( COUNT, ARRAY_OF_REQUEST, IERROR )<br>INTEGER COUNT, ARRAY_OF_REQUEST, IERROR |

| 参数说明 | | |
|---|---|---|
| IN | COUNT, | 需要激活的请求句柄个数. |
| INOUT | ARRAY_OF_REQUEST, | 请求句柄数组. |

**例 5.8.1.** *在示例 5.6.1 中, 整数 $m$ 只是在进程环中向前移动一次, 如果我们要移动多次, 可以使用下面的程序实现:*

```
      program mring
      include 'mpif.h'
c
      integer iter
      parameter(iter = 3)
      integer myid, p, mycomm, ierr, m, status(mpi_status_size, 2),
     &        next, front, mod, n, i, requests(2)
c
      call mpi_init( ierr )
```

```
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
c
      m = myid
      front = mod(p+myid-1, p)
      next = mod(myid+1, p)
      call mpi_send_init(m, 1, mpi_integer, next, 1, mycomm,
     &                   requests(1), ierr)
      call mpi_recv_init(n, 1, mpi_integer, front, 1, mycomm,
     &                   requests(2), ierr)
      do 10 i=1, iter
        call mpi_startall(2, requests, ierr)
        call mpi_waitall(2, requests, status, ierr)
        m = n
10    continue
      print *, 'The value of m is ', m, ' on Process ', myid
c
      call mpi_comm_free(mycomm, ierr)
      call mpi_finalize(ierr)
      stop
      end
```

## §5.9 MPI 的通信模式

前面所提到的通信模式是标准通信模式, 它是由 MPI 实现中确定的, 可能选择的是缓冲区模式 (buffered-mode) 或同步模式 (synchronous-mode), 此外 MPI 还提供了就绪模式 (ready-mode). 这些模式仅对发送消息起作用, 无论用任何模式发送的消息都可用一个接收函数来完成接收消息. 在下面给出的与发送消息有关的函数, 其参数与对应标准通信模式时是完全一致的, 因此这里我们仅给出函数名字及简要说明.

- MPI_BSEND, 使用缓冲区发送消息, 只要缓冲区足够大, 而不管接收进程是否开始接收它都将立即返回, 也就是说, 在发送消息时是不需要等待的.
- MPI_SSEND, 它可以在没有接收信号时就开始发送, 但要等到接收完成之后才能结束通信操作, 好处是无需用户自己申请缓冲区.
- MPI_RSEND, 只当已经有接收信号时才开始发送消息, 否则将出现错误.
- MPI_IBSEND, 异步通信的 MPI_BSEND.
- MPI_ISSEND, 异步通信的 MPI_SSEND.
- MPI_IRSEND, 异步通信的 MPI_RSEND.
- MPI_BSEND_INIT, 建立自己缓冲区的 MPI_SEND_INIT.
- MPI_SSEND_INIT, 同步模式的 MPI_SEND_INIT.

- MPI_RSEND_INIT, 就绪模式的 MPI_SEND_INIT.

有关消息缓冲区的管理, 可以使用如下的二个函数:

MPI_BUFFER_ATTACH

| | |
|---|---|
| C | `int MPI_Buffer_attach( void *buffer, int size )` |
| Fortran | `MPI_BUFFER_ATTACH( BUFFER, SIZE, IERROR )`<br>`<TYPE> BUFFER(*)`<br>`INTEGER SIZE, IERROR` |

参数说明

| | | |
|---|---|---|
| IN | BUFFER, | 初始化缓冲区的首地址. |
| IN | SIZE, | 缓冲区按字节的大小. |

这个函数指明使用消息缓冲区模式通信时用哪一个缓冲区.

MPI_BUFFER_DETACH

| | |
|---|---|
| C | `int MPI_Buffer_detach( void *buffer, int *size )` |
| Fortran | `MPI_BUFFER_DETACH( BUFFER, SIZE, IERROR )`<br>`<TYPE> BUFFER(*)`<br>`INTEGER SIZE, IERROR` |

参数说明

| | | |
|---|---|---|
| OUT | BUFFER, | 要拆卸的缓冲区. |
| OUT | SIZE, | 返回缓冲区的字节数. |

## §5.10 用户定义的数据类型与打包

## §5.11 用户定义的数据类型

在 MPI 中用户定义的数据类型和 C 语言中的结构是非常类似的, 它包括数据类型、数据长度和数据起始位置, 这个新定义的数据类型同其它 MPI 定义的标准数据类型一样, 可以用来作为发送和接收消息的数据类型. 对于这个新定义的数据类型, 具体地说它的组成可以概括为如下的形象描述:

- 它包含了一系列的原始数据类型;
- 和一系列的称之为位移 (displacement) 的整数.

数据类型与位移是成对出现的, 这一系列的数据对就是称之为类型映射 (type map). 用类型映射来表示一个新的数据类型为:

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

在这个新的数据类型中, 包括位移是 $disp_i$ 的原始数据类型 $type_i$, 其中 $i = 0, \ldots, n-1$. 如何构造新的数据类型我们将在以下进行详细介绍.

MPI_TYPE_CONTIGUOUS

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

参数说明

| | | |
|---|---|---|
| IN | COUNT, | 重复放在一起的 OLDTYPE 的元素个数. |
| IN | OLDTYPE, | 原始数据类型. |
| OUT | NEWTYPE, | 新构造的数据类型. |

这是构造数据类型中最简单的一种, 就是把原始数据类型重复 COUNT 次, 我们举个简单的例子来说明其含义. 假设 COUNT=5, OLDTYPE=INTEGER, 则调用此函数之后产生的新的数据类型 NEWTYPE 为 5 个整数.

MPI_TYPE_VECTOR

| | |
|---|---|
| C | `int MPI_Type_vector( int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype )` |
| Fortran | `MPI_TYPE_VECTOR( COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR )`<br>`INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR` |

参数说明

| | | |
|---|---|---|
| IN | COUNT, | OLDTYPE 的块的个数. |
| IN | BLOCKLENGTH, | 每块的长度. |
| IN | STRIDE, | 用 OLDTYPE 的元素度量的每块起始的间距. |
| IN | OLDTYPE, | 原始数据类型. |
| OUT | NEWTYPE, | 新构造的数据类型. |

假设 COUNT=3, BLOCKLENGTH=2, STRIDE=3, OLDTYPE=INTEGER, 则调用此函数得到的新数据类型 NEWTYPE 为 6 个整数组成. 这 6 个整数是在原始数组中从开始取 2 个, 隔一个再取 2 个, 再隔一个再取 2 个构成的.

MPI_TYPE_HVECTOR

| | |
|---|---|
| C | `int MPI_Type_hvector( int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype )` |
| Fortran | `MPI_TYPE_HVECTOR( COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR )`<br>`INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR` |

参数说明

| | | |
|---|---|---|
| IN | COUNT, | OLDTYPE 的块的个数. |
| IN | BLOCKLENGTH, | 每块的长度. |
| IN | STRIDE, | 用字节数度量的每块起始的间距. |
| IN | OLDTYPE, | 原始数据类型. |
| OUT | NEWTYPE, | 新构造的数据类型. |

这个函数要比 MPI_TYPE_VECTOR 更具普遍性, 但是它的每一块是固定不变的, 下面将介绍变块长度的数据类型如何构造.

MPI_TYPE_INDEXED

| | |
|---|---|
| C | int MPI_Type_indexed( int count, int *array_of_blocklengths,<br>int *array_of_displacements,<br>MPI_Datatype oldtype, MPI_Datatype *newtype ) |
| Fortran | MPI_TYPE_INDEXED( COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,<br>OLDTYPE, NEWTYPE, IERROR )<br>INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),<br>OLDTYPE, NEWTYPE, IERROR |

参数说明

| | | |
|---|---|---|
| IN | COUNT, | OLDTYPE 的块的个数. |
| IN | ARRAY_OF_BLOCKLENGTHS, | 每块长度数组. |
| IN | ARRAY_OF_DISPLACEMENTS, | 用元素个数定义的每块的起始位移量数组. |
| IN | OLDTYPE, | 原始数据类型. |
| OUT | NEWTYPE, | 新构造的数据类型. |

这里 ARRAY_OF_BLOCKLENGTHS 和 ARRAY_OF_DISPLACEMENTS 是由 OLDTYPE 定义的元素个数来度量的, 而且 ARRAY_OF_DISPLACEMENTS 是一个相对值. 前面给出的 MPI_TYPE_VECTOR 函数是此函数的特例, 也即下面的二个调用产生相同的数据类型.

MPI_TYPE_VECTOR( COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR ) 和

MPI_TYPE_INDEXED( COUNT, LENGTHS, DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR )

其中 LENGTHS(i)=BLOCKLENGTH, DISPLACEMENTS(i)=i*STRIDE, i=0, ..., COUNT-1. 在这里我们给出的函数 MPI_TYPE_INDEXED 的位移量是由 OLDTYPE 定义的元素个数确定的, 下面将要给出的函数的位移量是由字节数确定的.

MPI_TYPE_HINDEXED

| | |
|---|---|
| C | int MPI_Type_hindexed( int count, int *array_of_blocklengths,<br>int *array_of_displacements,<br>MPI_Datatype oldtype, MPI_Datatype *newtype ) |
| Fortran | MPI_TYPE_HINDEXED( COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,<br>OLDTYPE, NEWTYPE, IERROR )<br>INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),<br>OLDTYPE, NEWTYPE, IERROR |

参数说明

| | | |
|---|---|---|
| IN | COUNT, | OLDTYPE 的块的个数. |
| IN | ARRAY_OF_BLOCKLENGTHS, | 每块长度数组. |
| IN | ARRAY_OF_DISPLACEMENTS, | 用字节数定义的每块的起始位移量数组. |
| IN | OLDTYPE, | 原始数据类型. |
| OUT | NEWTYPE, | 新构造的数据类型. |

以上给出的关于数据类型构造函数都是单一的原始数据类型, 下面将给出一个最一般的构造新数据类型的函数.

MPI_TYPE_STRUCT

| | |
|---|---|
| C | `int MPI_Type_struct( int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)` |
| Fortran | `MPI_TYPE_STRUCT( COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR )`<br>`INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*), ARRAY_OF_TYPES(*), NEWTYPE, IERROR` |

参数说明

| | | |
|---|---|---|
| IN | `COUNT`, | 块的个数. |
| IN | `ARRAY_OF_BLOCKLENGTHS`, | 每块长度数组. |
| IN | `ARRAY_OF_DISPLACEMENTS`, | 用字节数定义的每块的起始位移量数组. |
| IN | `ARRAY_OF_TYPES`, | 原始数据类型数组. |
| OUT | `NEWTYPE`, | 新构造的数据类型. |

在所有 MPI 的构造数据类型中, `MPI_TYPE_STRUCT` 是使用最广泛的一个函数, 正确使用此函数在实际应用中是非常重要的. 到目前为此, 所有构造数据类型的函数都已经介绍了, 但是要使新构造的数据类型真正能够使用, 还需要有以下的一些函数配合才行, 因此关于如何使用这些数据类型构造函数的例子将在其它一些有关的函数介绍完之后给出.

MPI_TYPE_COMMIT

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

参数说明

| | | |
|---|---|---|
| INOUT | `DATATYPE`, | 准备提交的数据类型. |

在使用自定义的 MPI 数据类型之前, 必须调用 `MPI_TYPE_COMMIT`.

MPI_TYPE_FREE

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

参数说明

| | | |
|---|---|---|
| INOUT | `DATATYPE`, | 释放不用的数据类型. |

MPI_TYPE_EXTENT

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

参数说明

| | | |
|---|---|---|
| IN | DATATYPE, | 数据类型. |
| OUT | EXTENT, | 数据类型拓展后的字节数. |

MPI_ADDRESS

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

参数说明

| | | |
|---|---|---|
| IN | LOCATION, | 在存储器中的变量表示. |
| OUT | ADDRESS, | 变量在存储器中的地址. |

为了说明如何构造数据类型和 MPI_TYPE_EXTENT 函数的含义以及 MPI_ADDRESS 的使用, 我们给出一个具体的例子加以解释.

**例 5.11.1.** 假设在 *C* 语言中定义了一个结构, 其中包括一个双精度数和一个字符, 我们要定义这样结构的 *MPI* 数据类型.

```
#include <stdio.h>
#include "mpi.h"
typedef struct {
  double value; char str; } data;

void main(argc, argv)
int argc; char **argv;
{
  int p, myid, lens[3]={1, 1, 1}, i;
  MPI_Comm mycomm;  data tst[3];
  MPI_Datatype new, type[3] = {MPI_DOUBLE, MPI_CHAR, MPI_UB};
  MPI_Aint disp[3], size;  MPI_Status status;

  MPI_Init( &argc, &argv );
  MPI_Comm_dup( MPI_COMM_WORLD, &mycomm);
  MPI_Comm_rank( mycomm, &myid);  MPI_Comm_size( mycomm, &p);

  if (myid == 0)
    for(i=0; i<3; i++) { tst[i].value = 1.0; tst[i].str='a'; }
  printf("\nThe Process %1d of %1d is running.\n", myid, p);
  MPI_Address(&tst[0], &disp[0]);  MPI_Address(&tst[0].str, &disp[1]);
```

```
  for (i=2; i>=0; i--)  disp[i] -= disp[0];
  MPI_Type_struct(3, lens, disp, type, &new);
  MPI_Type_commit(&new);
  MPI_Type_extent(new, &size);

  if(myid == 0) printf("\nThe Extent of Struct Data Type is %d.\n", size);
  if(myid == 0) MPI_Ssend(tst, 3, new, 1, 1, mycomm);
  else if (myid == 1) MPI_Recv(tst, 3, new, 0, 1, mycomm, &status);
  MPI_Type_free(&new);
  if(myid == 1)
    printf("\nThe values are %f and %c\n", tst[1].value, tst[1].str);
  MPI_Comm_free(&mycomm);
  MPI_Finalize();
}
```

在这个 C 程序中, 我们定义了一个新的 MPI 数据类型, 而且还使用它进行消息传递. 这里我们使用了 MPI 中提供的伪数据类型 (pseudo datatype) MPI_UB, 在 MPI 中还有一个伪数据类型 MPI_LB. 这样定义的 MPI 的新数据类型 +new+ 在调用函数 MPI_TYPE_EXTENT 之后返回值与机器关于双精度数是按 4 位还是 8 位对齐有关, 如果是按 4 位对齐, 返回值为 12, 否则为 16. 但是如果我们使用下面的函数 MPI_TYPE_SIZE, 它的返回值是 9.

MPI_TYPE_SIZE

| | |
|---|---|
| C | int MPI_Type_size( MPI_Datatype datatype, int *size ) |
| Fortran | MPI_TYPE_SIZE( DATATYPE, SIZE, IERROR )<br>INTEGER DATATYPE, SIZE, IERROR |

参数说明

| | | |
|---|---|---|
| IN | DATATYPE, | 数据类型. |
| OUT | SIZE, | 数据类型的字节数. |

MPI_GET_ELEMENTS

| | |
|---|---|
| C | int MPI_Get_elements(MPI_Status *status,MPI_Datatype datatype,int *count) |
| Fortran | MPI_GET_ELEMENTS( STATUS, DATATYPE, COUNT, IERROR )<br>INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR |

参数说明

| | | |
|---|---|---|
| IN | STATUS, | 接收消息的状态. |
| IN | DATATYPE, | 数据类型. |
| OUT | COUNT, | MPI 基本数据类型的个数. |

该函数和 MPI_GET_COUNT 的含意是不同的, MPI_GET_COUNT 是接收到的数据类型个数, 而这里给出的 MPI_GET_ELEMENTS 得到的是 MPI 基本数据类型的个数. 如果这里的数据类型是 MPI 基本数据类型, 如 C 语言中的 MPI_INT (Fortran 中 MPI_INTEGER) 等, 则 MPI_GET_COUNT 和 MPI_GET_ELEMENTS 等价. 我们使用下面的一段程序来说明它们的差别.

```
......
call mpi_type_contiguous(2, mpi_integer, type, ierr)
call mpi_type_commit(type, ierr)
if(myid .eq. 0 .and. p .gt. 1) then
  call mpi_send(ia, 3, mpi_integer, 1, 1, mycomm, ierr)
elseif(myid .eq. 1) then
  call mpi_recv(ia, 2, type, 0, 1, mycomm, status, ierr)
  call mpi_get_count(status, type, count1, ierr)
  call mpi_get_elements(status, type, count2, ierr)
endif
......
```

一般来说, 在我们自己写的程序中很少会出现发送与接收时数据类型是不同的, 因此在大部分实际应用中, 只需会使用 MPI_GET_COUNT 就不会遇到任何困难. 在这一节中, 我们已经介绍了如何构造自定义的 MPI 数据类型, 通常在定义这些数据类型时使用的是相对位置移动量, 如果使用绝对地址来构造新的 MPI 数据类型, 在使用这个新的数据类型进行消息传递时, MPI 定义了一个称之为 MPI_BOTTOM 的消息缓冲区常数, 以供 MPI 的发送与接收函数使用, 具体的使用方式将在以后的例子中给出.

## §5.12 MPI 的数据打包与拆包

这里的数据打包与拆包和 PVM (Parallel Virtual Machine) 有相同的性质, 在 MPI 的绝大多数应用中使用用户自定义的 MPI 数据类型就可以完成许多数据一起发送与接收的目的. 但是对于复杂的情况使用数据打包可能会收到好处.

MPI_PACK

| | |
|---|---|
| C | int MPI_Pack( void* inbuf, int incount, MPI_Datatype datatype,<br>void* outbuf, int outsize, int *position, MPI_Comm comm ) |
| Fortran | MPI_PACK( INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM,<br>IERROR )<br><type> INBUF(*), OUTBUF(*)<br>INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR |

参数说明

| | | |
|---|---|---|
| IN | INBUF, | 准备打在包中的输入缓冲区. |
| IN | INCOUNT, | 输入元素的个数. |
| IN | DATATYPE, | 数据类型. |
| OUT | OUTBUF, | 打包缓冲区. |
| IN | OUTSIZE, | 用字节数定义的打包缓冲区的大小. |
| INOUT | POSITION, | 打包缓冲区的当前位置. |
| IN | COMM, | 打包消息的通信子. |

MPI_UNPACK

| | |
|---|---|
| C | `int MPI_Unpack( void* inbuf, int insize, int *position, void* outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm )` |
| Fortran | `MPI_UNPACK( INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM, IERROR )`<br>`<type> INBUF(*), OUTBUF(*)`<br>`INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR` |

参数说明

| | | |
|---|---|---|
| IN | INBUF, | 拆包缓冲区. |
| IN | INSIZE, | 用字节数定义的拆包缓冲区的大小. |
| INOUT | POSITION, | 打包缓冲区的当前位置. |
| OUT | OUTBUF, | 输出数据缓冲区. |
| IN | OUTCOUNT, | 输出数据缓冲区元素的个数. |
| IN | DATATYPE, | 数据类型. |
| IN | COMM, | 通信子. |

采用这种打包方式进行消息传递时, 数据类型一定要使用 MPI_PACKED. 为了能够知道一组数据在打包时需要多大的缓冲区, MPI 提供了如下的函数:

MPI_PACK_SIZE

| | |
|---|---|
| C | `int MPI_Pack_size( int incount, MPI_Datatype datatype, MPI_Comm comm, int *size )` |
| Fortran | `MPI_PACK_SIZE( INCOUNT, DATATYPE, COMM, SIZE, IERROR )`<br>`INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR` |

参数说明

| | | |
|---|---|---|
| IN | INCOUNT, | 准备打包的元素个数. |
| IN | DATATYPE, | 数据类型. |
| IN | COMM, | 通信子. |
| OUT | SIZE, | 按字节数定义的打包需要的缓冲区大小. |

此函数返回的值 SIZE 要比原始数据本身所占有的字节数要大, 这是因为在包中还有一些其它的信息. 以下我们给出使用 MPI_PACK 和 MPI_UNPACK 的一个例子, 来结束关于用户自定义的数据类型部分.

**例 5.12.1.** 假设我们要把一个整数数组和一个双精度数组从第 0 个进程传送到第 1 个进程, 尽管我们可以使用用户自定义数据类型来达到一次传送的目的, 但我们也可用打包和拆包的方式实现上述要求. 具体实现如下:

```
      program pack
      include 'mpif.h'
c
      integer maxbuf, len
```

```
      parameter (maxbuf = 200, len = 10)
      integer myid, p, mycomm, ierr, status(mpi_status_size, 2),
     &        ia(len), count1, count2, i, pos
      real*8 a(len)
      character buf(maxbuf)
c
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
c
      if(myid .eq. 0) then
        do 10 i=1, len
          ia(i) = i
          a(i) = dble(i)
10      continue
      endif
      call mpi_pack_size(len, mpi_integer, mycomm, count1, ierr)
      call mpi_pack_size(len, mpi_double_precision, mycomm, count2, i)
      print *, 'The pack size of 10 integer and real8 are:',
     &         count1, count2
      pos = 1
      if(myid .eq. 0 .and. p .gt. 1) then
        call mpi_pack(ia, len, mpi_integer, buf, maxbuf, pos,
     &                mycomm, ierr)
        call mpi_pack(a, len, mpi_double_precision, buf, maxbuf, pos,
     &                mycomm, ierr)
        call mpi_send(buf, pos-1, mpi_packed, 1, 1, mycomm, ierr)
      elseif(myid .eq. 1) then
        call mpi_recv(buf, maxbuf, mpi_packed, 0, 1, mycomm,
     &                status, ierr)
        call mpi_unpack(buf, maxbuf, pos, ia, len, mpi_integer,
     &                  mycomm, ierr)
        call mpi_unpack(buf, maxbuf, pos, a, len, mpi_double_precision,
     &                  mycomm, ierr)
        print *, 'The received values are: ', a(1), a(2), a(3)
      endif
      call mpi_comm_free(mycomm, ierr)
      call mpi_finalize(ierr)
      stop
      end
```

## §5.13 聚合通信

聚合通信 (collective communication) 是指多个进程 (通常大于 2) 之间的通信. 在这一节中, 介绍三方面的内容, 障碍同步 (barrier synchronization)、全局通信函数 (global communication functions) 和全局归约操作 (global reduction operations).

## §5.14 障碍同步

MPI_BARRIER

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

参数说明

| | | |
|---|---|---|
| IN | COMM, | 通信子. |

这是 MPI 提供的唯一的一个同步函数, 当 COMM 中的所有进程都执行这个函数时才返回, 如果有一个进程没有执行此函数, 其余进程将处于等待状态. 在执行完这个函数之后, 所有进程将同时执行其后的任务.

## §5.15 单点与多点通信函数

MPI_BCAST

| | |
|---|---|
| C | `int MPI_Bcast( void* buffer, int count, MPI_Datatype datatype,`<br>`int root, MPI_Comm comm )` |
| Fortran | `MPI_BCAST( BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR )`<br>`<type> BUFFER(*)`<br>`INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR` |

参数说明

| | | |
|---|---|---|
| INOUT | BUFFER, | 缓冲区的首地址. |
| IN | COUNT, | 缓冲区中元素的个数. |
| IN | DATATYPE, | 缓冲区的数据类型. |
| IN | ROOT, | 以它为源进行广播的进程编号. |
| IN | COMM, | 通信子. |

使用此函数时必须注意在 COMM 中的所有进程都执行此函数, 如果进程编号 ROOT, 则表示把此进程中的 BUFFER 的内容广播到 COMM 所有其它的进程中. 此函数是并行程序中经常出现的, 因此是个必须很好掌握的 MPI 通信函数.

MPI_GATHER

| | |
|---|---|
| C | `int MPI_Gather( void* sendbuf, int sendcount, MPI_Datatype sendtype,`<br>`void* recvbuf, int recvcount, MPI_Datatype recvtype,`<br>`int root, MPI_Comm comm )` |
| Fortran | `MPI_GATHER( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,`<br>`RECVTYPE, ROOT, COMM, IERROR )`<br>`<type> SENDBUF(*), RECVBUF(*)`<br>`INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR` |

| 参数说明 | | |
|---|---|---|
| IN | SENDBUF, | 发送缓冲区的首地址. |
| IN | SENDCOUNT, | 要发送的元素的个数. |
| IN | SENDTYPE, | 发送缓冲区的数据类型. |
| OUT | RECVBUF, | 接收缓冲区的首地址. |
| IN | RECVCOUNT, | 接收每个进程中数据元素的个数. |
| IN | RECVTYPE, | 接收缓冲区的数据类型. |
| IN | ROOT, | 接收进程的编号. |
| IN | COMM, | 通信子. |

这是 MPI 提供的聚集 (gather) 函数, 其作用是把 COMM 中所有进程 (包括 ROOT) 的数据聚集到 ROOT 进程中, 并且自动按进程编号顺序存放在接收缓冲区 RECVBUF 中. 对于非 ROOT 进程, 忽略接收缓冲区 RECVBUF.

MPI_GATHERV

| | |
|---|---|
| C | int MPI_Gatherv( void* sendbuf, int sendcount, MPI_Datatype sendtype,<br>void* recvbuf, int *recvcounts, int *displs,<br>MPI_Datatype recvtype, int root, MPI_Comm comm ) |
| Fortran | MPI_GATHERV( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,<br>DISPLS, RECVTYPE, ROOT, COMM, IERROR )<br><type> SENDBUF(*), RECVBUF(*)<br>INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE,<br>ROOT, COMM, IERROR |

| 参数说明 | | |
|---|---|---|
| IN | SENDBUF, | 发送缓冲区的首地址. |
| IN | SENDCOUNT, | 要发送的元素的个数. |
| IN | SENDTYPE, | 发送缓冲区的数据类型. |
| OUT | RECVBUF, | 接收缓冲区的首地址. |
| IN | RECVCOUNTS, | 接收每个进程数据元素的个数整数数组. |
| IN | DISPLS, | 接收数据存放的位置数组. |
| IN | RECVTYPE, | 接收缓冲区的数据类型. |
| IN | ROOT, | 接收进程的编号. |
| IN | COMM, | 通信子. |

这个函数是 MPI_GATHER 的扩充, 它允许从不同的进程中接收不同长度的消息, 而且接收到的消息可以存放在接收缓冲区的不同位置, 因此使用要比 MPI_GATHER 更灵活. 下面我们给出一个例子来说明如何使用 MPI 的聚集函数:

**例 5.15.1.** 假设在每个进程上都有一组数据, 现要将它们收集到进程编号为 ROOT 的进程中, 并按进程编号的顺序存放, 则有如下的程序实现此功能.

```
      program gather
      include 'mpif.h'
```

```
c
      integer maxbuf, len, mp
      parameter (maxbuf = 200, len = 10, mp = 5)
      integer myid, p, mycomm, ierr, root, ia(len), iga(maxbuf), i,
     &        displs(mp), counts(mp)
c
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
c
      do 10 i=1, len
        ia(i) = i+myid*len
10    continue
c
      do 20 i=1, p
        displs(i) = 20*i-20
        counts(i) = len
20    continue
      root = 0
c
      call mpi_gather(ia, len, mpi_integer, iga, len, mpi_integer,
     &                root, mycomm, ierr)
c
      if(myid .eq. root) then
        print *, 'The gather values are: ',
     &           iga(1), iga(len+1), iga(2*len+1)
      endif
c
      call mpi_gatherv(ia, len, mpi_integer, iga, counts, displs,
     &                 mpi_integer, root, mycomm, ierr)
c
      if(myid .eq. root) then
        print *, 'The gatherv values are: ',
     &           iga(2), iga(2*len+2), iga(4*len+2)
      endif
c
      call mpi_comm_free(mycomm, ierr)
      call mpi_finalize(ierr)
      stop
      end
```

这个程序的输出值应分别为: *1, 11, 21* 和 *2, 12, 22*. MPI_GATHER 接收到的数据在接收缓冲区中是连续存放的, 而 MPI_GATHERV 接收到的数据在接收缓冲区中是不连续存放的, 在我们这里给出的例子中, 由于 COUNTS 的所有分量是相同的, 可以看成是每隔步长为 *20* 的位置存放一个新接收的数据.

MPI_SCATTER

| | |
|---|---|
| C | `int MPI_Scatter( void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm )` |
| Fortran | `MPI_SCATTER( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR )`<br>`<type> SENDBUF(*), RECVBUF(*)`<br>`INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR` |

参数说明

| | | |
|---|---|---|
| IN | SENDBUF, | 发送缓冲区的首地址. |
| IN | SENDCOUNT, | 发送到每个进程中的元素的个数. |
| IN | SENDTYPE, | 发送缓冲区的数据类型. |
| OUT | RECVBUF, | 接收缓冲区的首地址. |
| IN | RECVCOUNT, | 接收数据元素的个数. |
| IN | RECVTYPE, | 接收缓冲区的数据类型. |
| IN | ROOT, | 发送进程的编号. |
| IN | COMM, | 通信子. |

这是 MPI 提供的散布 (scatter) 函数, 此函数是 MPI_GATHER 的逆操作.

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

参数说明

| | | |
|---|---|---|
| IN | SENDBUF, | 发送缓冲区的首地址. |
| IN | SENDCOUNTS, | 发送到每个进程中的元素的个数数组. |
| IN | DISPLS, | 发送到每个进程中的数据起始位移. |
| IN | SENDTYPE, | 发送缓冲区的数据类型. |
| OUT | RECVBUF, | 接收缓冲区的首地址. |
| IN | RECVCOUNT, | 接收数据元素的个数. |
| IN | RECVTYPE, | 接收缓冲区的数据类型. |
| IN | ROOT, | 发送进程的编号. |
| IN | COMM, | 通信子. |

## §5.16 多点与多点通信函数

MPI_ALLGATHER

| | |
|---|---|
| C | `int MPI_Allgather( void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm )` |
| Fortran | `MPI_ALLGATHER( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR )`<br>`<type> SENDBUF(*), RECVBUF(*)`<br>`INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR` |

**参数说明**

| | | |
|---|---|---|
| IN | SENDBUF, | 发送缓冲区的首地址. |
| IN | SENDCOUNT, | 要发送的元素的个数. |
| IN | SENDTYPE, | 发送缓冲区的数据类型. |
| OUT | RECVBUF, | 接收缓冲区的首地址. |
| IN | RECVCOUNT, | 接收每个进程中数据元素的个数. |
| IN | RECVTYPE, | 接收缓冲区的数据类型. |
| IN | COMM, | 通信子. |

MPI_ALLGATHERV

| | |
|---|---|
| C | `int MPI_Allgatherv( void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm )` |
| Fortran | `MPI_ALLGATHERV( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR )`<br>`<type> SENDBUF(*), RECVBUF(*)`<br>`INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR` |

**参数说明**

| | | |
|---|---|---|
| IN | SENDBUF, | 发送缓冲区的首地址. |
| IN | SENDCOUNT, | 要发送的元素的个数. |
| IN | SENDTYPE, | 发送缓冲区的数据类型. |
| OUT | RECVBUF, | 接收缓冲区的首地址. |
| IN | RECVCOUNT, | 接收每个进程中数据元素的个数. |
| IN | RECVTYPE, | 接收缓冲区的数据类型. |
| IN | COMM, | 通信子. |

此二函数是每个进程都进行数据聚集操作, 而 MPI_GATHER 和 MPI_GATHERV 只是其中的一个进程进行数据的聚集. 这里每个进程都发送消息也都接收消息.

MPI_ALLTOALL

| | |
|---|---|
| C | int MPI_Alltoall( void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm ) |
| Fortran | MPI_ALLTOALL( SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR )<br><type> SENDBUF(*), RECVBUF(*)<br>INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR |

参数说明

| | | |
|---|---|---|
| IN | SENDBUF, | 发送缓冲区的首地址. |
| IN | SENDCOUNT, | 要发送的元素的个数. |
| IN | SENDTYPE, | 发送缓冲区的数据类型. |
| OUT | RECVBUF, | 接收缓冲区的首地址. |
| IN | RECVCOUNT, | 接收每个进程中数据元素的个数. |
| IN | RECVTYPE, | 接收缓冲区的数据类型. |
| IN | COMM, | 通信子. |

MPI_ALLTOALLV

| | |
|---|---|
| C | int MPI_Alltoallv( void* sendbuf, int *sendcounts, int *sdispls, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm ) |
| Fortran | MPI_ALLTOALLV( SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR )<br><type> SENDBUF(*), RECVBUF(*)<br>INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, IERROR |

参数说明

| | | |
|---|---|---|
| IN | SENDBUF, | 发送缓冲区的首地址. |
| IN | SENDCOUNTS, | 发送到每个进程中的元素的个数数组. |
| IN | SDISPLS, | 发送到每个进程中的数据起始位移. |
| IN | SENDTYPE, | 发送缓冲区的数据类型. |
| OUT | RECVBUF, | 接收缓冲区的首地址. |
| IN | RECVCOUNTS, | 接收数据元素的个数数组. |
| IN | RDISPLS, | 接收进程中存放数据的起始位移. |
| IN | RECVTYPE, | 接收缓冲区的数据类型. |
| IN | COMM, | 通信子. |

这二个函数等价于在所有进程中的数据进行散布. 为了便于理解和正确使用此函数, 我们还是给出一个具体例子.

```
program allall
```

```
      include 'mpif.h'
      integer maxbuf, len, mp
      parameter (maxbuf = 200, len = 10, mp = 5)
      integer myid, p, mycomm, ierr, igb(maxbuf), iga(maxbuf), i,
     &        sdispls(mp), scounts(mp), rdispls(mp), rcounts(mp)
c
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
c
      do 10 i=1, maxbuf
        iga(i) = i+maxbuf*myid
10    continue
      do 20 i=1, p
        sdispls(i) = 20*i-20
        rdispls(i) = 15*i-15
        scounts(i) = len
        rcounts(i) = len
20    continue
      call mpi_alltoall(iga, len, mpi_integer, igb, len, mpi_integer,
     &                  mycomm, ierr)
      print *, 'The alltoall values are: ',
     &         igb(1), igb(len+1), igb(2*len+1), ' on Proc. ', myid
c
      call mpi_alltoallv(iga, scounts, sdispls, mpi_integer, igb,
     &                   rcounts, rdispls, mpi_integer, mycomm, ierr)
      print *, 'The alltoallv values are: ',
     &         igb(1), igb(16), igb(31), ' on Proc. ', myid
c
      call mpi_comm_free(mycomm, ierr)
      call mpi_finalize(ierr)
      stop
      end
```

## §5.17 全局归约操作

MPI_REDUCE

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

| 参数说明 | | |
|---|---|---|
| IN | SENDBUF, | 发送缓冲区的首地址. |
| OUT | RECVBUF, | 接收缓冲区的首地址. |
| IN | COUNT, | 缓冲区中元素的个数. |
| IN | DATATYPE, | 缓冲区的数据类型. |
| IN | OP, | 何种归约操作. |
| IN | ROOT, | 接收进程编号. |
| IN | COMM, | 通信子. |

这个函数是按分量进行归约操作的, 其运算由 OP 来确定, 最后的结果放在 ROOT 进程中, 其它进程中的 RECVBUF 不起作用. 在 MPI 中规定了一些允许的操作如下:

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

这些运算是有数据类型要求的, 首先对数据类型进行分类:

| | |
|---|---|
| C integer: | MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG |
| Fortran integer: | MPI_INTEGER |
| Floating point: | MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE |
| Logical: | MPI_LOGICAL |
| Complex: | MPI_COMPLEX |
| Byte : | MPI_BYTE |

现在对每种操作允许的数据类型规定如下:

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

关于 MPI_MAXLOC 和 MPI_MINLOC, MPI 对 Fortran 程序和 C 程序使用的复合数据类型规定如下:

**Fortran 程序**

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

**C 程序**

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

**例 5.17.1.** 假设在每个处理机中有一个数，我们要在这些数中找一个最大的，并确定这个最大数在哪个处理机中，则可用如下的程序：

```
      program reduce
      include 'mpif.h'
c
      integer myid, p, mycomm, ierr, m, n, root, pair(2), answer(2)
c
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
      root = 0
      m = myid
      call mpi_reduce(m, n, 1, mpi_integer, mpi_max, root, mycomm, ierr)
c
      if(myid .eq. root) print *, 'The maxmum value is ', n
      pair(1) = mod(myid + 1, p)
      pair(2) = myid
      call mpi_reduce(pair, answer, 1, mpi_2integer, mpi_maxloc, root,
     &                mycomm, ierr)
      if(myid .eq. root) print *, 'The maxmum value is ', answer(1),
     &                           ' on process ', answer(2)
c
      call mpi_comm_free(mycomm, ierr)
      call mpi_finalize(ierr)
      stop
      end
```

MPI_ALLREDUCE

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

参数说明

| | | |
|---|---|---|
| IN | SENDBUF, | 发送缓冲区的首地址. |
| OUT | RECVBUF, | 接收缓冲区的首地址. |
| IN | COUNT, | 缓冲区中元素的个数. |
| IN | DATATYPE, | 缓冲区的数据类型. |
| IN | OP, | 何种归约操作. |
| IN | COMM, | 通信子. |

此函数和 MPI_REDUCE 的意思是相同的, 只是最后结果在所有的进程中.

MPI_REDUCE_SCATTER

| | |
|---|---|
| C | `int MPI_Reduce_scatter( void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )` |
| Fortran | `MPI_REDUCE_SCATTER( SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, IERROR )`<br>`<type> SENDBUF(*), RECVBUF(*)`<br>`INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR` |

参数说明

| | | |
|---|---|---|
| IN | SENDBUF, | 发送缓冲区的首地址. |
| OUT | RECVBUF, | 接收缓冲区的首地址. |
| IN | RECVCOUNTS, | 元素个数的数组. |
| IN | DATATYPE, | 缓冲区的数据类型. |
| IN | OP, | 何种归约操作. |
| IN | COMM, | 通信子. |

此函数相当于先做 MPI_REDUCE, 然后在做 MPI_SCATTER.

**例 5.17.2.** 设我们使用 $p$ 个进程计算 $c = Ab$, 其中 $A$ 是 $m$ 阶矩阵, $b$ 是 $m$- 维向量. 假设 $m = np$ (如不能整除把剩余部分放在最后一个进程中), 在每个进程中存放 $A$ 的 $n$ 列和对应的 $b$ 的 $n$ 个分量, 现要在每个进程中得到与 $b$ 对应的 $c$ 或是 $c$ 的全部, 则我们可以用下面的程序实现:

```
      program redsct
      include 'mpif.h'
c
      integer lda, cols, maxnp
      parameter (lda = 100, cols = 100, maxnp = 5)
```

```
      integer myid, p, mycomm, ierr, m, n, counts(maxnp)
      real a(lda, cols), b(cols), c(lda), sum(lda)
      logical sct
c
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
      if (p .gt. maxnp) goto 999
      m = 100
      sct = .false.
c      sct = .true.
      call initab(m, myid, p, n, a, lda, b, sum)
      call locmv(m, n, a, lda, b, sum)
c if sct = true, call reduce_scatter, otherwize call allreduce
      if(sct) then
        call mpi_allgather(n, 1, mpi_integer, counts, 1,
     &                     mpi_integer, mycomm, ierr)
        call mpi_reduce_scatter(sum, c, counts, mpi_real, mpi_sum,
     &                          mycomm, ierr)
      else
        call mpi_allreduce(sum, c, m, mpi_real, mpi_sum, mycomm, ierr)
      endif
c
      print *, 'The values of c are ', c(1), c(2)
c
999   call mpi_comm_free(mycomm, ierr)
      call mpi_finalize(ierr)
      stop
      end

      subroutine initab(m, myid, np, k, a, lda, b, sum)
      integer m, myid, np, k, lda, i, j, l
      real a(lda, *), b(*), sum(*)
      do 10 i=1, m
10      sum(i) = 0.0
      k = m/np
      l = myid * k
      if(myid .eq. np-1) k = m-l
      do 20 i=1, k
20      b(i) = 1.0
      do 40 j=1, k
        do 40 i=1, m
40        a(i, j) = real(i+l+j)
      return
      end

      subroutine locmv(m, n, a, lda, b, c)
      integer m, n, lda, i, j
      real a(lda, *), b(*), c(*)
      do 20 j=1, n
```

```
        do 20 i=1, m
20        c(i) = c(i)+a(i, j)*b(j)
      return
      end
```

MPI_SCAN

| | |
|---|---|
| C | int MPI_Scan( void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm ) |
| Fortran | MPI_SCAN( SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR )<br><type> SENDBUF(*), RECVBUF(*)<br>INTEGER COUNT(*), DATATYPE, OP, COMM, IERROR |

参数说明

| | | |
|---|---|---|
| IN | SENDBUF, | 发送缓冲区的首地址. |
| OUT | RECVBUF, | 接收缓冲区的首地址. |
| IN | COUNT, | 元素个数. |
| IN | DATATYPE, | 缓冲区的数据类型. |
| IN | OP, | 何种归约操作. |
| IN | COMM, | 通信子. |

此函数是 MPI_REDUCE 在不同进程中的重复使用, 进程 $i$ 中得到的是在由编号为 $\{0,1,\ldots,i\}$ 的进程构成的进程组中使用 MPI_REDUCE 结果, 是一个并不常用的函数.

在 MPI 中提供了用户自定义的操作函数以满足不同需要, 它由如下的几个函数来实现:

MPI_OP_CREATE

| | |
|---|---|
| C | int MPI_Op_create( MPI_User_function *function, int commute, MPI_Op *op ) |
| Fortran | MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR )<br>EXTERNAL FUNCTION<br>LOGICAL COMMUTE<br>INTEGER OP, IERROR |

参数说明

| | | |
|---|---|---|
| IN | FUNCTION, | 用户定义的函数. |
| IN | COMMUTE, | 等于 TRUE 是可交换的, 否则只是可结合的. |
| OUT | OP, | 新的归约操作. |

此函数定义了一个新的归约操作 OP.

MPI_OP_FREE

| | |
|---|---|
| C | int MPI_Op_free( MPI_Op *op ) |
| Fortran | MPI_OP_FREE( OP, IERROR )<br>INTEGER OP, IERROR |

| 参数说明 | | |
|---|---|---|
| IN | OP, | 归约操作. |

此函数释放归约操作 OP. 在 MPI 中, 用户自定义操作的函数是有严格要求的, 其形式如下:

| | |
|---|---|
| C | void user_function( void* invec, void* inoutvec, int *len,<br>MPI_Datatype *datatype ) |
| Fortran | FUNCTION USER_FUNCTION( INVEC, INOUTVEC, LEN, DATATYPE )<br><datatype> INVEC(LEN), INOUTVEC(LEN)<br>INTEGER LEN, DATATYPE |

下面的例子是用自定义的求和函数来实现全局操作的:

```
      program userdef
      include 'mpif.h'
      integer len
      parameter (len = 100)
      integer myid, p, mycomm, ierr, m, root, myop
      real x(len), y(len)
      external userfunc
      call mpi_init( ierr )
      call mpi_comm_dup( mpi_comm_world, mycomm, ierr)
      call mpi_comm_rank( mycomm, myid, ierr )
      call mpi_comm_size( mycomm, p, ierr )
      print *, 'Process ', myid, ' of ', p, ' is running'
      m = 100
      root = 0
      call initx(m, myid, x)
      call mpi_op_create(userfunc, .true., myop, ierr)
      call mpi_reduce(x, y, m, mpi_real, myop, root, mycomm, ierr)
      call mpi_op_free(myop, ierr)
      if(myid .eq. root)
     &   print *, 'The values of answer are ', y(1), y(2), y(3)
      call mpi_comm_free(mycomm, ierr)
      call mpi_finalize(ierr)
      stop
      end

      subroutine initx(m, myid, x)
      integer m, myid, i
      real x(*)
      do 10 i=1, m
10      x(i) = real(i+myid)
      return
      end

      subroutine userfunc(x, y, m, mpi_real)
      integer m
      real x(*), y(*)
      do 10 i=1, m
10      y(i) = y(i)+x(i)
```

```
    return
    end
```

## §5.18 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.

### §5.18.1 迪卡尔拓扑结构

#### §5.18.1.1 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:

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

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`.

#### §5.18.1.2 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[`$i$`]`, $i = 0, \ldots,$ `ndims` $-1$, 使得 $\prod_{i=0}^{\texttt{ndims}-1}$ `dims[`$i$`]` $=$ `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, 在例 7.2.4 的程序示例中, 假如进程数为 4, 差分网络为 $100 \times 400$, 则理想的进程拓扑结构应为 $1 \times 4$, 此时无法调用 `MPI_DIMS_CREATE` 自动计算最优进程划分，后者会返回 $2 \times 2$。

#### §5.18.1.3 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.

#### §5.18.1.4 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.

#### §5.18.1.5 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.

#### §5.18.1.6 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.

#### §5.18.1.7 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.

#### §5.18.1.8 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指定方向 (维) 上在处理器间进行数据平移 (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:

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

进程拓扑结构使用的程序示例参见例 7.2.4.

### §5.18.2 一般拓扑结构

#### §5.18.2.1 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 给出图中的结点数 (如果 nnodes 小于通信器 comm_old 中的进程数, 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:

$$\texttt{Neighbor}(0) = \{\texttt{edges}[j] \mid 0 \le j < \texttt{index}[0]\}$$
$$\texttt{Neighbor}(i) = \{\texttt{edges}[j] \mid \texttt{index}[i-1] \le j < \texttt{index}[i]\}$$
$$i = 1, \ldots, \texttt{nnodes} - 1$$

(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).

#### §5.18.2.2 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.

#### §5.18.2.3 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.

#### §5.18.2.4 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 §5.18.2.1, 112 页). 参数 maxindex 和 maxedges 分别限定数组 index 和 edges 的最大长度.

#### §5.18.2.5 查询指定进程的邻居数

**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
```

数组 neighbors 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.

### §5.18.3 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.

#### §5.18.3.1 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 §5.18.1.1, 109 页).

#### §5.18.3.2 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 §5.18.2.1, 112 页).

# 第六章　文件输入输出 (MPI–IO)

MPI 的并行输入输出函数 (简称 MPI–IO 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.

本章介绍的函数在 MPICH 1.2.2 及后续版本中已经全部实现, 但在其它 MPI 版本中目前还不一定能用. 因此, 在使用本章中介绍的函数前应先确认所使用的 MPI 系统是否支持它们.

## §6.1　基本术语

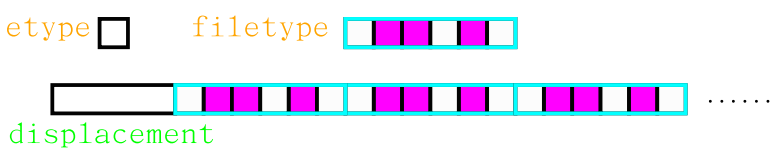
**文件 (file)** MPI 的“文件”可以看成由具有相同或不同类型的数据项构成的序列. MPI 支持对文件的顺序和随机访问. MPI 的文件是和进程组相关联的: MPI 打开文件的函数 (`MPI_File_open`) 中要求指定一个通信器, 并且该通信器中所有进程必须同时对文件进行打开或关闭操作.

**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 6.1.1.** 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:

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

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.


## §6.2 基本文件操作

### §6.2.1 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).

### §6.2.2 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.

### §6.2.3 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.

### §6.2.4 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.

### §6.2.5 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.

### §6.2.6 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.

## §6.3 查询文件参数

### §6.3.1 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.

### §6.3.2 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.

## §6.4 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.

如果打开文件时使用的是 MPI_MODE_SEQUENTIAL 模式, 则 disp 参数的值只能是文件当前位置, 即 MPI_DISPLACEMENT_CURRENT.

### §6.4.1 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 §6.4.2), 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.

### §6.4.2 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 新增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).

### §6.4.3 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
```

## §6.5 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<br>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<br>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<br>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.

### §6.5.1 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).

### §6.5.2 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
```

### §6.5.3 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.

### §6.5.4 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
```

#### §6.5.5 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)
```

### §6.6 文件指针操作

#### §6.6.1 独立文件指针操作

##### §6.6.1.1 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`

#### §6.6.1.2 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` 返回独立文件指针的位移.

### §6.6.2 共享文件指针操作

#### §6.6.2.1 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.

#### §6.6.2.2 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.

### §6.6.3 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).

## §6.7 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.

### §6.7.1 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 MPI 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, 由 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 6.7.1. *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.

### §6.7.2 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.

### §6.7.3 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.

## §6.8 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.

`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 7.2.5 中给出了一个使用 MPI–IO 函数的程序示例.

# 第七章　MPI 程序示例

## §7.1　矩阵乘积

$A$ 为 M × N 阶矩阵, $B$ 为 N × L 阶矩阵, $C$ 为 M × L 阶矩阵。计算矩阵乘积 $C = AB$。

### §7.1.1　算法描述

假设：使用 NPROCS 个 MPI 进程，为简单起见假定 M 和 L 均为 NPROCS 的倍数。$A$ 和 $C$ 按行等分成子块分别存储在不同的进程中，而 $B$ 则按列等分成子块分别存储在不同的进程中。$A$, $B$ 和 $C$ 的子块大小分别为 MLOC × N, N × LLOC 和 MLOC × L 其中 MLOC = M/NPROCS, LLOC = L/NPROCS。具体存储方式为:

$\{\ a_{i,j} \mid k \times \mathtt{MLOC} < i \le (k+1) \times \mathtt{MLOC},\quad 1 \le j \le \mathtt{N}\ \}$ 存储在进程 $k$ 的数组 A 中

$\{\ b_{i,j} \mid 1 \le i \le \mathtt{N},\quad k \times \mathtt{LLOC} < j \le (k+1) \times \mathtt{LLOC}\ \}$ 存储在进程 $k$ 的数组 B 中

$\{\ c_{i,j} \mid k \times \mathtt{MLOC} < i \le (k+1) \times \mathtt{MLOC},\quad 1 \le j \le \mathtt{L}\ \}$ 存储在进程 $k$ 的数组 C 中

$k = 0, \ldots, \mathtt{NPROCS} - 1$

算法: 矩阵 $A$ 和 $C$ 的子块不动, 矩阵 $B$ 的子块在各个进程间循环移动。图 7.1 是当 NPROCS = 3 时的计算流程示意图。

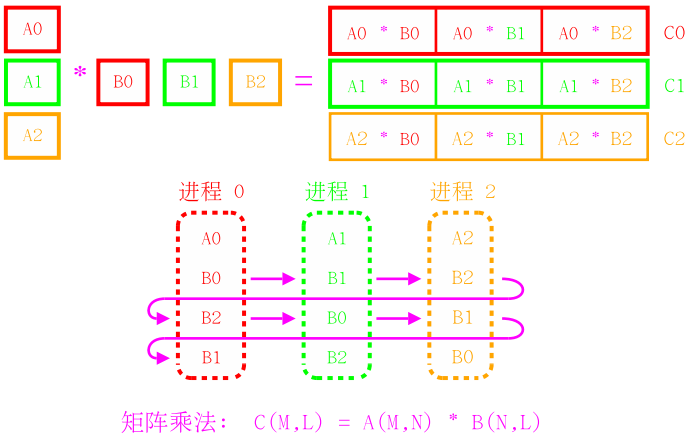



图 7.1: 矩阵乘法计算流程

### §7.1.2　MPI 并行程序

我们将主程序单独放在一个文件中，它负责分配存储单元并生成矩阵 $A$ 和 $B$ 的子块，然后调用子程序 MATMUL 完成矩阵的乘法运算。

子程序 MATMUL 的参数形式如下:

```
      SUBROUTINE MATMUL(M, N, L, MYRANK, NPROCS, A, B, C, WORK)
      DOUBLE PRECISION A(M/NPROCS, N), B(N, L/NPROCS),
     &                 C(M/NPROCS, L), WORK(N, L/NPROCS)
```

其中 NPROCS 为 MPI 进程数, MYRANK 为当前进程的 MPI 进程号。数组 A, B 和 C 分别存储矩阵 $A$, $B$ 和 $C$ 的子块。WORK 为工作数组, 它的大小与数组 B 一样。

**例 7.1.1.** 矩阵乘积主程序 *(源程序在文件 matrix-main.f 中)*。

```
*
* Parallel matrix multiplication: main program
*
      PROGRAM MATMUL_MAIN
      IMPLICIT DOUBLE PRECISION (A-H, O-Z)
      INCLUDE 'mpif.h'
      PARAMETER (NBUFFER=128*1024*1024/8)
      DIMENSION BUF(NBUFFER)
      DOUBLE PRECISION TIME_START, TIME_END
      EXTERNAL INIT, MATMUL, CHECK
*
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYRANK, IERR)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROCS, IERR)
*
      IF (MYRANK.EQ.0) THEN
         PRINT *, 'Enter M, N, L: '
         CALL FLUSH(6)
         READ(*,*) M, N, L
      ENDIF
      CALL MPI_BCAST(M, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, IERR)
      CALL MPI_BCAST(N, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, IERR)
      CALL MPI_BCAST(L, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, IERR)
*
      IF ( MOD(M,NPROCS).NE.0 .OR. MOD(L,NPROCS).NE.0 ) THEN
         IF (MYRANK.EQ.0) PRINT *, 'M OR L Cannot be divided by nprocs!'
         CALL MPI_FINALIZE(IERR)
         STOP
      ENDIF
*
      IA   = 1
      IB   = IA  + M/NPROCS   * N
      IC   = IB  + N          * L/NPROCS
      IWK  = IC  + M/NPROCS   * L
      IEND = IWK + N          * L/NPROCS
      IF ( IEND .GT. NBUFFER+1 ) THEN
         IF (MYRANK.EQ.0) PRINT *, 'Insufficient buffer size!'
         CALL MPI_FINALIZE(IERR)
         STOP
      ENDIF
```

```
*
      CALL INIT( M, N, L, MYRANK, NPROCS, BUF(IA), BUF(IB), BUF(IC) )

      TIME_START = MPI_WTIME()
      CALL MATMUL( M, N, L, MYRANK, NPROCS, BUF(IA), BUF(IB), BUF(IC),
     &          BUF(IWK) )
      TIME_END = MPI_WTIME()

      CALL CHECK( M, N, L, MYRANK, NPROCS, BUF(IA), BUF(IB), BUF(IC) )
*
      IF ( MYRANK .EQ. 0 ) THEN
         PRINT *, 'TIME   = ', TIME_END-TIME_START
         PRINT *, 'MFLOPS = ', M*(N+N-1.0)*L/(TIME_END-TIME_START)*1D-6
      ENDIF
*
      CALL MPI_FINALIZE(IERR)
      STOP
      END
*
*---------------------------------------------------------------------
*
      SUBROUTINE INIT(M, N, L, MYRANK, NPROCS, A, B, C)
      IMPLICIT DOUBLE PRECISION (A-H, O-Z)
      INCLUDE 'mpif.h'
      DIMENSION A(M/NPROCS, N), B(N, L/NPROCS), C(M/NPROCS, L)
*
      MLOC = M/NPROCS
      LLOC = L/NPROCS
*
* INIT. A, B
      DO J=1, N
         DO I=1, MLOC
            A(I,J) = I+MYRANK*MLOC
         ENDDO
      ENDDO
*
      DO J=1, LLOC
         DO I=1, N
            B(I,J) = J+MYRANK*LLOC
         ENDDO
```

```
      ENDDO
*
      RETURN
      END
*
*--------------------------------------------------------------------
*
      SUBROUTINE CHECK(M, N, L, MYRANK, NPROCS, A, B, C)
      IMPLICIT DOUBLE PRECISION (A-H, O-Z)
      INCLUDE 'mpif.h'
      DIMENSION A(M/NPROCS, N), B(N, L/NPROCS), C(M/NPROCS, L)
      INTEGER LOCAL_CODE, CODE
*
      MLOC = M/NPROCS
      LLOC = L/NPROCS
*
* CHECK THE RESULTS
      LOCAL_CODE = 0
      DO J=1, L
         DO I=1, MLOC
            IF ( ABS(C(I,J) - N*DBLE(I+MYRANK*MLOC)*J) .GT. 1D-10 ) THEN
               LOCAL_CODE = 1
               GOTO 10
            ENDIF
         ENDDO
      ENDDO
*
10    CALL MPI_REDUCE( LOCAL_CODE, CODE, 1, MPI_INTEGER, MPI_SUM, 0,
     &                 MPI_COMM_WORLD, IERR)
*
      IF ( MYRANK .EQ. 0 ) THEN
         PRINT *, 'Code   =  ', CODE
      ENDIF
*
      RETURN
      END
```

**例 7.1.2.** 矩阵乘子程序：使用 MPI_Sendrecv_replace *(源程序在文件 matrix1.f 中)*。

```
*
* Parallel multiplication of matrices using MPI_Sendrecv
```

```
*
      SUBROUTINE MATMUL(M, N, L, MYRANK, NPROCS, A, B, C, WORK)
      IMPLICIT DOUBLE PRECISION (A-H, O-Z)
      INCLUDE 'mpif.h'
      DIMENSION A(M/NPROCS, N), B(N, L/NPROCS), C(M/NPROCS, L),
     &          WORK(N, L/NPROCS)
      INTEGER SRC, DEST, TAG
      INTEGER STATUS(MPI_STATUS_SIZE)
*
      MLOC = M/NPROCS
      LLOC = L/NPROCS
*
      DEST = MOD( MYRANK-1+NPROCS, NPROCS )
      SRC  = MOD( MYRANK+1,        NPROCS )
*
      JPOS=MYRANK*LLOC
*
      DO IP=1, NPROCS - 1
         TAG = 10000 + IP
*
         DO J=1, LLOC
            DO I=1, MLOC
               SUM=0.DO
               DO K=1, N
                  SUM = SUM + A(I,K) * B(K,J)
               ENDDO
               C(I, J+JPOS) = SUM
            ENDDO
         ENDDO
*
         CALL MPI_SENDRECV_REPLACE(B, N*LLOC, MPI_DOUBLE_PRECISION,
     &             DEST, TAG, SRC, TAG, MPI_COMM_WORLD, STATUS, IERR)
*
         JPOS = JPOS + LLOC
         IF ( JPOS .GE. L ) JPOS = 0
*
      ENDDO
*
      DO J=1, LLOC
         DO I=1, MLOC
```

```
43              SUM=0.DO
44              DO K=1, N
45                 SUM = SUM + A(I,K) * B(K,J)
46              ENDDO
47              C(I, J+JPOS) = SUM
48           ENDDO
49        ENDDO
50  *
51        RETURN
52        END
```

可以用下述命令编译、运行例 7.1.1–7.1.2 给出的程序:

```
mpif77 -O3 -o matrix matrix-main.f matrix1.f
mpirun -np 4 matrix
```

### §7.1.3 MPI 并行程序的改进

**例 7.1.3.** 使用异步通信函数 MPI_Isend/MPI_Irecv，在适当的硬件环境下它可使得计算与通信重叠进行。该程序在文件 *matrix2.f* 中。

```
1   *
2   * Parallel multiplication of matrices using MPI_Isend/MPI_Irecv
3   *
4         SUBROUTINE MATMUL(M, N, L, MYRANK, NPROCS, A, B, C, WORK)
5         IMPLICIT DOUBLE PRECISION (A-H, O-Z)
6         INCLUDE 'mpif.h'
7         DIMENSION A(M/NPROCS, N), B(N, L/NPROCS), C(M/NPROCS, L),
8        &          WORK(N, L/NPROCS)
9         INTEGER SRC, DEST, TAG
10        INTEGER STATUS(MPI_STATUS_SIZE, 2), REQUEST(2)
11  *
12        MLOC = M/NPROCS
13        LLOC = L/NPROCS
14  *
15        DEST = MOD( MYRANK-1+NPROCS, NPROCS )
16        SRC  = MOD( MYRANK+1,        NPROCS )
17  *
18        JPOS=MYRANK*LLOC
19  *
20        DO IP=1, NPROCS - 1
21           TAG = 10000 + IP
22  *
```

```
23          CALL MPI_ISEND( B, N*LLOC, MPI_DOUBLE_PRECISION, DEST, TAG,
24     &                    MPI_COMM_WORLD, REQUEST(1), IERR )
25          CALL MPI_IRECV( WORK, N*LLOC, MPI_DOUBLE_PRECISION, SRC, TAG,
26     &                    MPI_COMM_WORLD, REQUEST(2), IERR )
27 *
28          DO J=1, LLOC
29             DO I=1, MLOC
30                SUM=0.DO
31                DO K=1, N
32                   SUM = SUM + A(I,K) * B(K,J)
33                ENDDO
34                C(I, J+JPOS) = SUM
35             ENDDO
36          ENDDO
37 *
38          CALL MPI_WAITALL(2, REQUEST, STATUS, IERR)
39 *
40 * 拷贝 WORK -> B (可以通过在计算/通信中交替使用 B/WORK 来避免该操作)
41          DO J=1, LLOC
42             DO I=1, N
43                B(I,J) = WORK(I,J)
44             ENDDO
45          ENDDO
46 *
47          JPOS = JPOS + LLOC
48          IF ( JPOS .GE. L ) JPOS = 0
49 *
50       ENDDO
51 *
52       DO J=1, LLOC
53          DO I=1, MLOC
54             SUM=0.DO
55             DO K=1, N
56                SUM = SUM + A(I,K) * B(K,J)
57             ENDDO
58             C(I, J+JPOS) = SUM
59          ENDDO
60       ENDDO
61 *
62       RETURN
```

```
63        END
```

**例 7.1.4.** 调用 *BLAS* 库函数完成矩阵子块的乘积。选用适当的 *BLAS* 库可以大幅度提高程序的实际运行性能。源程序在文件 *matrix-blas.f* 中。注意，编译该程序时必须与 *BLAS* 库链接。

```
1  *
2  * Parallel multiplication of matrices using MPI_Isend/MPI_Irecv and BLAS
3  *
4        SUBROUTINE MATMUL(M, N, L, MYRANK, NPROCS, A, B, C, WORK)
5        IMPLICIT DOUBLE PRECISION (A-H, O-Z)
6        INCLUDE 'mpif.h'
7        DIMENSION A(M/NPROCS, N), B(N, L/NPROCS), C(M/NPROCS, L),
8       &          WORK(N, L/NPROCS)
9        INTEGER SRC, DEST, TAG
10       INTEGER STATUS(MPI_STATUS_SIZE, 2), REQUEST(2)
11 *
12       MLOC = M/NPROCS
13       LLOC = L/NPROCS
14 *
15       DEST = MOD( MYRANK-1+NPROCS, NPROCS )
16       SRC  = MOD( MYRANK+1,        NPROCS )
17 *
18       JPOS=MYRANK*LLOC
19 *
20       DO IP=1, NPROCS - 1
21          TAG = 10000 + IP
22 *
23          CALL MPI_ISEND( B, N*LLOC, MPI_DOUBLE_PRECISION, DEST, TAG,
24      &                   MPI_COMM_WORLD, REQUEST(1), IERR )
25          CALL MPI_IRECV( WORK, N*LLOC, MPI_DOUBLE_PRECISION, SRC, TAG,
26      &                   MPI_COMM_WORLD, REQUEST(2), IERR )
27 *
28          CALL DGEMM('N', 'N', MLOC, LLOC, N, 1.D0, A, MLOC, B, N, 0.D0,
29      &              C(1,1+JPOS), MLOC)
30 *
31          CALL MPI_WAITALL(2, REQUEST, STATUS, IERR)
32 *
33 * 拷贝 WORK -> B (可以通过在计算/通信中交替使用 B/WORK 来避免该操作)
34          CALL DCOPY(N*LLOC, WORK, 1, B, 1)
35 *
36          JPOS = JPOS + LLOC
```

```
37          IF ( JPOS .GE. L ) JPOS = 0
38  *
39        ENDDO
40  *
41        CALL DGEMM('N', 'N', MLOC, LLOC, N, 1.D0, A, MLOC, B, N, 0.D0,
42       &           C(1,1+JPOS), MLOC)
43  *
44        RETURN
45        END
```

**作业 7.1.1.** 修改例 *7.1.4* 中的程序，以避免第 *34* 行的矩阵拷贝 *(DCOPY)* 操作。

**作业 7.1.2.** 修改例 *7.1.4* 中的程序，要求在不使用中间数组 *work* 的情况下实现计算与通信的 *(*部分*)* 重叠。修改后的程序对性能有什么影响？*(*提示：将数组 *B* 分成两半，一半通信，一半计算*)*。

## §7.2 Poisson 方程求解

设计 MPI 程序，求解定义在二维规则区域上的 Poisson 方程:

$$\begin{cases} -\Delta u(x,y) = f(x,y) & (x,y) \in \Omega = (0,a) \times (0,b) \\ u(x,y)|_{\partial\Omega} = g(x,y) \end{cases} \tag{7.1}$$

其中，$f(x,y)$ 和 $g(x,y)$ 为已知函数，分别定义在区域 $\Omega$ 的内部和边界。

### §7.2.1 并行算法设计

沿坐标轴 $x$ 和 $y$ 方向，分别取步长

$$h_x = \frac{a}{\texttt{IM}}, \qquad h_y = \frac{b}{\texttt{JM}} \tag{7.2}$$

将区域 $\Omega$ 离散成规模为 $\texttt{IM} \times \texttt{JM}$ 的网格，其中 IM 和 JM 分别为沿坐标轴 $x$ 和 $y$ 方向的网格单元个数。

不妨设方程 (7.1) 的近似解 $u(x,y)$ 定义在所有网格结点上，且用如下未知量表示

$$\begin{cases} u_{i,j} = u(i \times h_x, j \times h_y) & 1 \le i \le \texttt{IM}-1, \quad 1 \le j \le \texttt{JM}-1 \\ u_{i,j} = g_{i,j} = g(i \times h_x, j \times h_y) & i = 0 \text{ 或 } i = \texttt{IM} \text{ 或 } j = 0 \text{ 或 } j = \texttt{JM} \end{cases} \tag{7.3}$$

用二阶中心差商近似导数:

$$\begin{cases} u_{xx}(i \times h_x, j \times h_y) \approx \dfrac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h_x^2}, \\ u_{yy}(i \times h_x, j \times h_y) \approx \dfrac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_y^2} \end{cases} \tag{7.4}$$

并记

$$f_{i,j} = f(i \times h_x, j \times h_y) \tag{7.5}$$

将以上程式代入式 (7.1), 则问题转化为求解稀疏线性代数方程组：

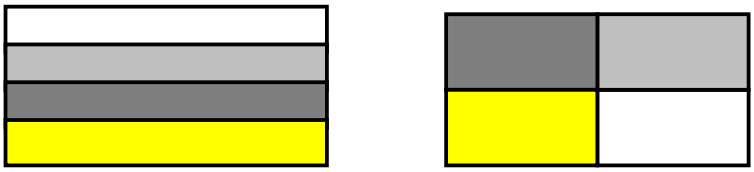
$$2(h_x^2+h_y^2)u_{i,j}-h_y^2(u_{i-1,j}+u_{i+1,j})-h_x^2(u_{i,j-1}+u_{i,j+1})=h_x^2h_y^2f_{i,j},\\ 1\le i\le \mathtt{IM}-1,\quad 1\le j\le \mathtt{JM}-1 \tag{7.6}$$

具体地，我们将采用众所周知的 Jacobi 点迭代算法求解方程 (7.6)。

### §7.2.2 MPI 并行程序设计

设计求解方程 (7.1) 的 MPI 并行程序必须考虑两个关键问题：

第一，选择正确的区域分解策略。将区域 $\Omega$ 分解成多个子区域，分配给不同的进程，并保证进程间的负载平衡和最小的消息传递通信开销。通常有两种方式：1）沿 $y$ 方向的一维条分解策略，如图 7.2(a) 所示；2）沿两个方向的二维块分解策略，如图 7.2(b) 所示。显然，如果取 $x$ 方向的进程个数等于 1，则二维块分解策略就退化为一维条分解策略。无论哪种方式，都应该尽量保证每个子区域包含的网格结点个数相等，因为这样才能保证进程间的负载平衡。



(a) 一维条分解　　(b) 二维块分解

图 7.2: 两种区域分解策略

第二，选择合适的通信数据结构。由式 (7.6) 可知，在任意网格结点上，执行 Jacobi 点迭代需要知道该结点上、下、左、右四个相邻结点上的近似解。因此，在每次 Jacobi 迭代之前，每个进程必须与其相邻的进程交换边界结点上的近似解。

为了描述通信数据结构，不妨设方程近似解定义在网格单元的中心。图 7.3 给出了一个 3×3 的二维块区域分解，其中，各个子区域被分配给不同的进程，各个进程负责求解该子区域的近似解。具体地，进程 5 将向其相邻的四个进程 (进程 2、进程 4、进程 6 和进程 8) 输出 “●” 标示的网格单元的近似解；同时，从这四个进程接收用 “○” 标示的网格单元上的近似解。因此，如何管理这些沿区域分解边界交换的网格单元上的量，将会直接影响到 MPI 程序的并行性能。图 7.4 给出了一个比较有效的办法，就是沿各个子区域的边界，向外增加一个宽度为 1 的辅助网格单元，用于存储相邻子区域在这些网格单元上的近似解。

类似地，同样的数据结构也适应于近似解定义在网格结点上的情形，这里不再讨论。

下面，基于以上二维块区域分解策略和通信数据结构，给出求解差分方程 (7.6) 的 MPI 并行程序。其中，近似解定义在网格结点上。为了简单，这里假设网格单元个数 IM 和 JM 能分别被沿 $x$ 方向和 $y$ 方向的进程个数 NPX 和 NPY 整除, 进程的序号按自然序排列 (先沿 $x$ 方向，后沿 $y$ 方向)。

**例 7.2.1.** *并行 Jacobi 点迭代 MPI 程序：二维块分解策略 (源程序见文件 poisson0.f)。*

```
1 ! Poisson 方程求解: 使用阻塞通信(可能死锁)。作者: 莫则尧
2       INCLUDE  "mpif.h"
3       PARAMETER(DA=2.0, DB=3.0)  ! 问题求解区域沿X、Y方向的大小
```

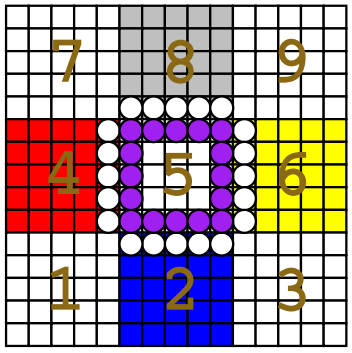



图 7.3: 3 × 3 二维区域分解

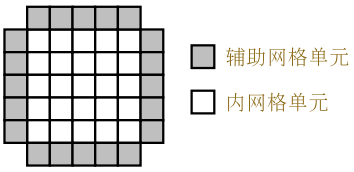



图 7.4: 辅助网格单元示意图

```
      PARAMETER(IM=30, JM=60)     ! 沿X、Y方向的全局网格规模
      PARAMETER(NPX=1,  NPY=1)    ! 沿X、Y方向的进程个数
      PARAMETER(IML=IM/NPX, JML=JM/NPY)
         ! 各进程沿X、Y方向的局部网格规模，仅为全局网格规模的1/(NPX*NPY)
      REAL  U(0:IML+1, 0:JML+1)  ! 定义在网格结点的近似解
      REAL  US(0:IML+1, 0:JML+1) ! 定义在网格结点的精确解
      REAL  U0(IML, JML)          ! Jacobi迭代辅助变量
      REAL  F(IML, JML)           ! 函数f(x, y)在网格结点上的值
      INTEGER  NPROC              ! mpirun启动的进程个数，必须等于NPX*NPY
      INTEGER  MYRANK,MYLEFT,MYRIGHT,MYUPPER,MYLOWER
                                  ! 各进程自身的序号，4个相邻进程的序号
      INTEGER  MEPX,MEPY          ! 各进程自身的序号沿X,Y方向的坐标
      REAL  XST,YST               ! 各进程拥有的子区域沿X,Y方向的起始坐标
      REAL  HX, HY                ! 沿X,Y方向的网格离散步长
      REAL  HX2,HY2,HXY2,RHXY
      INTEGER  IST,IEND,JST,JEND
                     ! 各进程沿X,Y方向的内部网格结点的起始和终止坐标
      INTEGER  HTYPE, VTYPE
                     ! MPI用户自定义数据类型，表示各进程沿X,Y方向
                     ! 与相邻进程交换的数据单元
      INTEGER  STATUS(MPI_STATUS_SIZE)
! In-line functions
      solution(x,y)=x**2+y**2     ! 解析解
      rhs(x,y)=-4.0               ! Poisson 方程源项(右端项)
!
!     程序可执行语句开始
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROC,IERR)
      IF (NPROC.NE.NPX*NPY.OR.MOD(IM,NPX).NE.0.OR.MOD(JM,NPY).NE.0) THEN
```

```
33          PRINT *, '+++ mpirun -np xxx error OR grid scale error, ',
34     &            'exit out +++'
35          CALL MPI_FINALIZE(IERR)
36          STOP
37        ENDIF
38  !
39  !     按自然序(先沿X方向, 后沿Y方向)确定各进程自身及其4个相邻进程的序号
40        CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)
41        MYLEFT  = MYRANK - 1
42        IF (MOD(MYRANK,NPX).EQ.0)   MYLEFT=MPI_PROC_NULL
43        MYRIGHT = MYRANK + 1
44        IF (MOD(MYRIGHT,NPX).EQ.0)  MYRIGHT=MPI_PROC_NULL
45        MYUPPER = MYRANK + NPX
46        IF (MYUPPER.GE.NPROC)       MYUPPER=MPI_PROC_NULL
47        MYLOWER = MYRANK - NPX
48        IF (MYLOWER.LT.0)           MYLOWER=MPI_PROC_NULL
49        MEPY=MYRANK/NPX
50        MEPX=MYRANK-MEPY*NPX
51  !                     对应二维NPY(NPX Cartesian行主序坐标为(MEPY,MEPX).
52  !
53  !     基本变量赋值, 确定各进程负责的子区域
54        HX =DA/IM
55        HX2=HX*HX
56        HY =DB/JM
57        HY2=HY*HY
58        HXY2=HX2*HY2
59        RHXY=0.5/(HX2+HY2)
60        XST=MEPX*DA/NPX
61        YST=MEPY*DB/NPY
62        IST=1
63        IEND=IML
64        IF (MEPX.EQ.NPX-1) IEND=IEND-1   ! 最右边的区域X方向少一个点
65        JST=1
66        JEND=JML
67        IF (MEPY.EQ.NPY-1) JEND=JEND-1   ! 最上边的区域Y方向少一个点
68  !
69  !     数据类型定义
70        CALL MPI_TYPE_CONTIGUOUS(IEND-IST+1, MPI_REAL, HTYPE, IERR)
71        CALL MPI_TYPE_COMMIT(HTYPE, IERR)
72                  ! 沿X方向的连续IEND-IST+1个MPI_REAL数据单元,
```

```
73                ! 可用于表示该进程与其上、下进程交换的数据单元
74       CALL MPI_TYPE_VECTOR(JEND-JST+1, 1, IML+2, MPI_REAL, VTYPE, IERR)
75       CALL MPI_TYPE_COMMIT(VTYPE, IERR)
76                ! 沿Y方向的连续JEND-JST+1个MPI_REAL数据单元,
77                ! 可用于表示该进程与其左、右进程交换的数据单元
78 !
79       DO J=JST-1, JEND+1
80       DO I=IST-1, IEND+1
81          xx=(I+MEPX*IML)*HX      ! xx=XST+I*HX
82          yy=(J+MEPY*JML)*HY      ! yy=YST+J*HY
83          IF (I.GE.IST.AND.I.LE.IEND .AND. J.GE.JST.AND.J.LE.JEND) THEN
84             U(I,J)  = 0.0                ! 近似解赋初值
85             US(I,J) = solution(xx,yy)    ! 解析解
86             F(I,J)  = rhs(xx,yy)         ! 右端项
87          ELSE IF ((I.EQ.IST-1  .AND. MEPX.EQ.0) .OR.
88      &            (J.EQ.JST-1  .AND. MEPY.EQ.0) .OR.
89      &            (I.EQ.IEND+1 .AND. MEPX.EQ.NPX-1) .OR.
90      &            (J.EQ.JEND+1 .AND. MEPY.EQ.NPY-1)) THEN
91             U(I,J) = solution(xx,yy)     ! 边界值
92          ENDIF
93       ENDDO
94       ENDDO
95 !
96 !     Jacobi迭代求解
97       NITER=0
98 100   CONTINUE
99       NITER=NITER+1
100 !
101 !     交换定义在辅助网格结点上的近似解
102      CALL MPI_SEND(U(1,1),      1, VTYPE, MYLEFT,  NITER+100,
103     &              MPI_COMM_WORLD,IERR)                    ! 发送左边界
104      CALL MPI_SEND(U(IEND,1),   1, VTYPE, MYRIGHT, NITER+100,
105     &              MPI_COMM_WORLD,IERR)                    ! 发送右边界
106      CALL MPI_SEND(U(1,1),      1, HTYPE, MYLOWER, NITER+100,
107     &              MPI_COMM_WORLD,IERR)                    ! 发送下边界
108      CALL MPI_SEND(U(1,JEND),   1, HTYPE, MYUPPER, NITER+100,
109     &              MPI_COMM_WORLD,IERR)                    ! 发送上边界
110      CALL MPI_RECV(U(IEND+1,1), 1, VTYPE, MYRIGHT, NITER+100,
111     &              MPI_COMM_WORLD,STATUS,IERR)             ! 接收右边界
112      CALL MPI_RECV(U(0,1),      1, VTYPE, MYLEFT,  NITER+100,
```

```
113       &              MPI_COMM_WORLD, STATUS,IERR)            ! 接收左边界
114        CALL MPI_RECV(U(1,JEND+1), 1, HTYPE, MYUPPER, NITER+100,
115       &              MPI_COMM_WORLD, STATUS,IERR)            ! 接收上边界
116        CALL MPI_RECV(U(1,0),      1, HTYPE, MYLOWER, NITER+100,
117       &              MPI_COMM_WORLD, STATUS, IERR)           ! 接收下边界
118        DO J=JST,JEND
119        DO I=IST,IEND
120           U0(I,J)=RHXY*(HXY2*F(I,J)+HX2*(U(I,J-1)+U(I,J+1))
121       &                           +HY2*(U(I-1,J)+U(I+1,J)))
122        ENDDO
123        ENDDO
124 !
125 !     计算与精确解间的误差
126        ERR=0.0
127        DO J=JST,JEND
128        DO I=IST,IEND
129           U(I,J)=U0(I,J)
130           ERR=MAX(ERR, ABS(U(I,J)-US(I,J))) ! 用L\infty模以使误差与NP无关
131        ENDDO
132        ENDDO
133        ERR0=ERR
134        CALL MPI_ALLREDUCE(ERR0,ERR,1,MPI_REAL,MPI_MAX,
135       &                   MPI_COMM_WORLD,IERR)
136        IF (MYRANK.EQ.0 .AND. MOD(NITER,100).EQ.0) THEN
137           PRINT *, 'NITER = ', NITER, '    ERR = ', ERR
138        ENDIF
139 !
140        IF (ERR.GT.1.E-3)  THEN     ! 收敛性判断
141           GOTO 100                 ! 没有收敛, 进入下次迭代
142        ENDIF
143 !
144        IF (MYRANK.EQ.0) THEN
145           PRINT *, ' !!! Successfully converged after ',
146       &           NITER, ' iterations'
147           PRINT *, ' !!! error = ', ERR
148        ENDIF
149 !     输出近似解(略)
150 !
151        CALL MPI_FINALIZE(IERR)
152        END
```

在上例中，我们固定了该 MPI 程序产生的进程个数 NP = NPY*NPX，这样做的一个主要目的是为了使各进程的内存空间仅为相应串行程序的 1/(NPY*NPX)，从而使得原来串行程序在单机上由于内存不够而无法计算的问题通过多机并行计算而成为可能。当然，这样做也带来一些不便，例如它要求 MPI 运行命令 "mpirun -np xxx" 中的参数 xxx 等于 NPY*NPX，且如果参数 NPY 和 NPX 被改变后，必须重新编译该程序。

### §7.2.3 MPI 并行程序的改进

在例 7.2.1 中，交换定义在辅助网格结点近似解的 8 条消息传递语句 (102–117 行) 是该 MPI 程序的关键。但是，由 MPI 标准可知, 它们是不安全的，因为在某些并行机上，当消息较长时 (例如大于 16KB)，可能由于 MPI 系统缓存区大小的限制，而导致执行该 MPI 程序的进程的死锁。因此，我们将它们替换成如下的非阻塞通信函数。

**例 7.2.2.** 改进一: 非阻塞通信 *(源程序见文件 poisson1.f)*

```
      ... ... ... ... ... ... (略)
! 用下一行替换 poisson0.f 第24行
      INTEGER  REQ(8), STATUS(MPI_STATUS_SIZE,8)
      ... ... ... ... ... ... (略)
! 用下述内容替换 poisson0.f 第102-117行
      CALL MPI_ISEND(U(1,1),      1, VTYPE, MYLEFT,  NITER+100,
     &               MPI_COMM_WORLD,REQ(1),IERR)           ! 发送左边界
      CALL MPI_ISEND(U(IEND,1),   1, VTYPE, MYRIGHT, NITER+100,
     &               MPI_COMM_WORLD,REQ(2),IERR)           ! 发送右边界
      CALL MPI_ISEND(U(1,1),      1, HTYPE, MYLOWER, NITER+100,
     &               MPI_COMM_WORLD,REQ(3),IERR)           ! 发送下边界
      CALL MPI_ISEND(U(1,JEND),   1, HTYPE, MYUPPER, NITER+100,
     &               MPI_COMM_WORLD,REQ(4),IERR)           ! 发送上边界
      CALL MPI_IRECV(U(IEND+1,1), 1, VTYPE, MYRIGHT, NITER+100,
     &               MPI_COMM_WORLD, REQ(5),IERR)          ! 接收右边界
      CALL MPI_IRECV(U(0,1),      1, VTYPE, MYLEFT,  NITER+100,
     &               MPI_COMM_WORLD, REQ(6),IERR)          ! 接收左边界
      CALL MPI_IRECV(U(1,JEND+1), 1, HTYPE, MYUPPER, NITER+100,
     &               MPI_COMM_WORLD, REQ(7),IERR)          ! 接收上边界
      CALL MPI_IRECV(U(1,0),      1, HTYPE, MYLOWER, NITER+100,
     &               MPI_COMM_WORLD, REQ(8),IERR)          ! 接收下边界
      CALL MPI_WAITALL(8,REQ,STATUS,IERR)     ! 阻塞式等待消息传递的结束
      ... ... ... ... ... ... (略)
```

在例 7.2.1 中，我们可以将 118–123 行的循环分裂成两个部分，其中一个部分需要辅助网格点上的近似解，而另一个部分不需要辅助网格点上的近似解。这样，为了改进该 MPI 程序的并行性能，我们可以将后一个部分的计算与例 7.2.2 的非阻塞消息传递重叠起来，从而达到屏蔽网络延迟的目的。具体改进如下。

**例 7.2.3.** 改进二: 重叠通信与计算 *(源程序见文件 poisson2.f)*

```
      ... ... ... ... ... ... (略)
! 用下述内容替换 poisson1.f 第119-124行
      DO J=JST+1,JEND-1
      DO I=IST+1,IEND-1
```

```
         U0(I,J)=RHXY*(HXY2*F(I,J)+HX2*(U(I,J-1)+U(I,J+1))
     &                            +HY2*(U(I-1,J)+U(I+1,J)))
      ENDDO
      ENDDO
      CALL MPI_WAITALL(8,REQ,STATUS,IERR)     ! 阻塞式等待消息传递的结束
      DO J=JST, JEND, JEND-JST
      DO I=IST, IEND
         U0(I,J)=RHXY*(HXY2*F(I,J)+HX2*(U(I,J-1)+U(I,J+1))
     &                            +HY2*(U(I-1,J)+U(I+1,J)))
      ENDDO
      ENDDO
      DO J=JST, JEND
      DO I=IST, IEND, IEND-IST
         U0(I,J)=RHXY*(HXY2*F(I,J)+HX2*(U(I,J-1)+U(I,J+1))
     &                            +HY2*(U(I-1,J)+U(I+1,J)))
      ENDDO
      ENDDO
      ... ... ... ... ... ... (略)
```

在例 7.2.1–7.2.3 中，各进程按自然序 (先沿 $x$ 方向，后沿 $y$ 方向) 确定与它相邻的 4 个进程的序号 (MYLEFT, MYRIGHT, MYLOWER, MYUPPER), 以及它自己所处的行主序位置 (MEPY，MEPX)。实际上，这些进程按区域分解策略可以很自然地映射到 NPY*NPX 的二维 Cartesian 拓扑结构 (参看 §5.18.1)，而 (MEPY，MEPX) 就是各进程在该拓扑结构中的坐标。因此，我们可以从通信器 MPI_COMM_WORLD 出发, 建立二维 Cartesian 拓扑结构，从而方便地确定各进程的相邻关系，并使得之后的所有 MPI 消息传递均基于该拓扑结构进行。

**例 7.2.4.** *改进三：二维 Cartesian 拓扑结构 (源程序见文件 poisson3.f)*

```
      ... ... ... ... ... ... (略)
! 在 poisson[012].f 程序头(变量声明部分)加入下面二行
      INTEGER COMM, DIMS(2),COORD(2)
      LOGICAL PERIOD(2),REORDER
      ... ... ... ... ... ... (略)
! 用下述内容替换 poisson[012].f 第39-50行
      DIMS(1)=NPY             ! 拓扑结构中Y方向的进程个数
      DIMS(2)=NPX             ! 拓扑结构中X方向的进程个数
      PERIOD(1)=.FALSE.       ! 沿Y方向, 拓扑结构非周期连接
      PERIOD(2)=.FALSE.       ! 沿X方向, 拓扑结构非周期连接
      REORDER=.TRUE.          ! 在新通信器中, 进程重新排列序号
      CALL MPI_CART_CREATE(MPI_COMM_WORLD, 2, DIMS, PERIOD, REORDER,
     &                     COMM, IERR)
      CALL MPI_COMM_RANK(COMM,MYRANK,IERR)
      CALL MPI_CART_COORDS(COMM,MYRANK,2,COORD,IERR)
      MEPY=COORD(1)
      MEPX=COORD(2)
      CALL MPI_CART_SHIFT(COMM, 0, 1, MYLOWER, MYUPPER, IERR)   ! Y方向
      CALL MPI_CART_SHIFT(COMM, 1, 1, MYLEFT,  MYRIGHT, IERR)   ! X方向
      ... ... ... ... ... ... (略)
```

在例 7.2.1–7.2.4 中，我们忽略了近似解的输出，这里，我们用第 第六章 章中介绍的 MPI 并行 I/O 函数实现近似解的并行输出。特别地, 我们要求输出的近似解按自然序排列，且包含物理边界结点。

**例 7.2.5.** 改进四：并行 *I/O* (源程序见文件 *poisson4.f*)。该程序使用了独立文件指针、聚合型函数 MPI_FILE_WRITE_ALL (参看表 6.5)。

```
      ... ... ... ... ... ... (略)
! 在程序头(变量声明部分)加入下面内容
      INTEGER FH, FILETYPE, MEMTYPE, GSIZE(2), LSIZE(2), START(2)
! 注意: 从下面三种形式的变量声明中根据所使用的MPI系统选择一个正确的
! (可以参考文件 mpiof.h 或 mpif.h 中 MPI_OFFSET_KIND 的定义)
!     INTEGER(kind=MPI_OFFSET_KIND) OFFSET    ! 适用于 Fortran 90
!     INTEGER*8 OFFSET                        ! 适用于 64 位系统
      INTEGER*4 OFFSET                        ! 适用于某些 32 位系统
      ... ... ... ... ... ... (略)
! 在标有 “输出近似解(略)” 处 (倒数第四行) 加入下述内容
      GSIZE(1)=IM+1
      GSIZE(2)=JM+1
      LSIZE(1)=IEND-IST+1
      IF (MEPX.EQ.0) LSIZE(1)=LSIZE(1)+1
      IF (MEPX.EQ.NPX-1) LSIZE(1)=LSIZE(1)+1
      LSIZE(2)=JEND-JST+1
      IF (MEPY.EQ.0) LSIZE(2)=LSIZE(2)+1
      IF (MEPY.EQ.NPY-1) LSIZE(2)=LSIZE(2)+1
      START(1)=IML*MEPX
      IF (MEPX.NE.0) START(1)=START(1)+1
      START(2)=JML*MEPY
      IF (MEPY.NE.0) START(2)=START(2)+1
!
!  定义局部子数组数据类型
      CALL MPI_TYPE_CREATE_SUBARRAY(2, GSIZE, LSIZE, START,
     &          MPI_ORDER_FORTRAN, MPI_REAL, FILETYPE, IERR)
      CALL MPI_TYPE_COMMIT(FILETYPE, IERR)
!
! 打开二进制文件
      CALL MPI_FILE_OPEN(COMM,'result.dat',
     &          MPI_MODE_CREATE + MPI_MODE_WRONLY,
     &          MPI_INFO_NULL, FH, IERR)
       OFFSET=0         ! 注意使用正确的变量类型 (INTEGER*4 或 INTEGER*8)
                        ! (参考文件 mpif.h 中 MPI_OFFSET_KIND 的定义)
      CALL MPI_FILE_SET_VIEW(FH, OFFSET, MPI_REAL, FILETYPE,
     &          'native', MPI_INFO_NULL, IERR)
!
! 定义数据类型, 描述子数组在内存中的分布
      GSIZE(1)=IML+2
      GSIZE(2)=JML+2
      START(1)=1
      IF (MEPX.EQ.0) START(1)=0
      START(2)=1
      IF (MEPY.EQ.0) START(2)=0
      CALL MPI_TYPE_CREATE_SUBARRAY(2, GSIZE, LSIZE, START,
```

```
     &          MPI_ORDER_FORTRAN, MPI_REAL, MEMTYPE, IERR)
      CALL MPI_TYPE_COMMIT(MEMTYPE, IERR)
!
! 输出近似解(含物理边界结点)
      CALL MPI_FILE_WRITE_ALL(FH, U, 1, MEMTYPE, STATUS, IERR)
      CALL MPI_FILE_CLOSE(FH, IERR)
!
      ... ... ... ... ... ... (略)
```

至止，例 7.2.2∼例 7.2.5 分别从非阻塞通信、重叠通信与计算、拓扑结构和并行 I/O 四个方面，利用相应的 MPI 函数，依次改进了例 7.2.1 中 MPI 程序的功能和并行性能。通过该应用示例，读者可以较好地将所介绍的 MPI 函数联系在一起，解决实际问题。

**作业 7.2.1.** *在例 7.2.1–7.2.5 中用了一排辅助网格单元。通过增加虚拟网格点的宽度可以提高通信粒度。例如，如果使用两排辅助网格单元，则可以每两次迭代交换一次子区域边界附近的近似解，代价是子区域间增加了少量的重复计算。试修改例 7.2.4 或例 7.2.5 中的程序，在程序中增加一个参数* BW*，它代表辅助网格单元的宽度 (*$bw \geq 1$*)，并比较不同* BW *的值对程序性能的影响。*

**作业 7.2.2.** *修改例 7.2.5 中的程序，将 Jacobi 迭代改为红黑顺序的 Gauss–Seidel 迭代。*

# 参考文献


[1] 莫则尧，袁国兴，《消息传递并行编程环境 MPI》，科学出版社，2001。

[2] J. M. Ortega, Introduction to Parallel and Vector Solution of Linear Systems, Plenum Press, 1988.

[3] J. Dongarra, I. Duff, D. Soresen and H. van der Vorst, Solving Linear Systems on Vector and Shared Memory Computers, SIAM, 1991.

[4] G. Golub and C. van Loan, Matrix Computation, The Johns Hopkins University Press, 1983. (中译本: 矩阵计算, 廉庆荣、邓健新、刘秀兰译, 大连理工大学出版社, 1988 年)

[5] 迟学斌, 在具有局部内存与共享主存的并行机上并行求解线性方程组, 计算数学, 1995 年第 17 卷第 2 期.

[6] G. Y. Li and T. F. Coleman, A Parallel Triangular Solver for a Hypercube Multiprocessor, TR 86–787, Cornell University, 1986.

[7] 迟学斌, Transputer 上 Cholesky 分解的并行实现, 计算数学, 1993 年第 15 卷第 3 期.

[8] J. M. Delosme and I. C. F. Ipsen, Positive Definite Systems with Hyperbolic Rotations, Linear Algebra Appl., 77 (1986), 75–111.

[9] D. H. Lawrie and A. H. Sameh, The Computation and Communication Complexity of a Parallel Banded System Solver, ACM Trans. Math. Soft., 10 (1984), 185–195.

[10] 迟学斌, Transputer 上线性系统的并行求解, 中国计算机用户, 1991 年第 10 期.

[11] 陈景良, 并行数值方法, 清华大学出版社, 1983 年.

[12] 张宝琳、袁国兴、刘兴平、陈劲, 偏微分方程并行有限差分方法, 科学出版社, 1994 年.

[13] D. Chazan and W. Miranker, Chaotic Relazation, J. Lin. Alg. Appl., 2 (1969), 199–222.

[14] G.Baudet, Asynchronous Iterative Methods for Multiprocessors, J. ACM, 25 (1978), 226–244.

[15] J. M. Ortega and W. C. Rheinboldt, Iterative Solution of Nonlinear Equations in Several Variables, Academic Press, 1970.

[16] 迟学斌, 线性方程组的异步迭代法, 计算数学, 1992 年第 14 卷第 3 期.

[17] S. Balay, W. Gropp, L. Curfman McInnes, and B. Smith, PETSc homepage, *http://www.mcs.anl.gov/petsc*.

[18] P. Bridge, N. Doss, W. Gropp, etc., User's Guide to MPICH, a Portable Implementation of MPI, ANL/MCS-TM-000，1994

[19] P. Bridge, N. Doss, W. Gropp, etc., Installation Guide to MPICH, a Portable Implementation of MPI, ANL/MCS-TM-000，1994

[20] D. E. Culler, J. P. Singh, A. Gupta, Parallel Computer Architecture: a Hardware/Software Approach, Morgan Kaufmann publisher, 1999.

[21] 陈国良，并行计算：结构、算法与编程，北京：高等教育出版社，1999。

[22] J. Dongarra, B. Tourancheau, editors, Prodceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, SIAM publications, 1994.

[23] W. Gropp, E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message–Passing Interface, MIT Press，Cambridge, MA, 1994.

[24] W. Gropp, E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message–Passing Interface, 2nd edition, MIT Press，Cambridge, MA, 1999.

[25] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, A High–Performance Portable Implementation of the MPI Message–Passing Interface Standard, Parallel Computing, Vol.22, No.6,1996, pp.789–828.

[26] W. Gropp, E. Lusk, R. Thakur, Using MPI–2: Advanced Features of the Message–Passing Interface, MIT Press, 1999.

[27] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam, PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Network Parallel Computing, MIT Press, Cambridge, MA, 1994.

[28] R. Hempel, D. W. Walker, The Emergence of The MPI Message Passing Standard for Parallel Computing, Computer Standard and Interface, Vol.21, 1999, pp.51–62.

[29] IMPI Steering Committee. IMPI — Interoperable Message–Passing Interface, 1998, *http://impi.nist.gov/IMPI*.

[30] 李晓梅，莫则尧等，可扩展并行算法设计与分析，北京：国防工业出版社，2000。

[31] Message Passing Interface Forum, MPI: A Message–Passing Interface Standard, *http://www.mpi-forum.org*.

[32] Message Passing Interface Forum, MPI: A Message–Passing Interface Standard, International Journal of Supercomputer Applications, Vol.8, No.3/4, 1994.

[33] Message Passing Interface Forum, MPI–2: A Message–Passing Interface Standard, International Journal of Supercomputer Applications, Vol.12, No.1/2, 1994

[34] MPI–2 C++ Bindings, 1999, *http://www.mpi.nd.edu/research/mpi2c++*.

[35] N. Nieuwejaar, D. Kotz, The Galley Parallel File System, Parallel Computing, Vol.23, No.4, 1997, pp.447–476.

[36] OpenMP Fortran Application Program Interface, Version 1.0, 1997, *http://www.openmp.org*.

[37] OpenMP C and C++ Application Program Interface, Version 1.0, 1997, *htpp://www.openmp.org*.

[38] Real–Time Message Passing Interface (MPI/RT) Forum, Document for the Real–Time Message Passing Interface (MPI/RT–1.0) Draft Standard, *http://www.mpirt.org/drafts.html*, 1999.

[39] W. Stalling, Operating Systems Internals and Design Principles, 3rd edition, Prentice Hall Publisher, 1998.

[40] 沈志宇等，并行程序设计，长沙：国防科技大学出版社，1997.

[41] P. S. Pacheco, Programming Parallel Processors Using MPI, San Francisco, CA, Morgan Kaufmann, 1995.

[42] M. Snir, S. Otto, S. Huss-lederman, etc., MPI: The Complete Reference, Cambridge, MA, MIT Press，1994.

[43] 孙家昶，张林波，迟学斌等，网络并行计算与分布式编程环境, 科学出版社，1996.

[44] The MPI–IO Committee, MPI-IO: A Parallel File I/O Interface for MPI, Version 0.5, 1996, *http://parallel.nas.nasa.gov/MPI-IO*.

[45] TOP500 list, *http://www.top500.org*

[46] R. Treumann, Experiences in the Implementation of a Thread Safe, Threads Based MPI for the IBM RS/6000 SP, 1998, *http://www.reserch.ibm.com/actc/Tools/MPI˙Threads.html*

[47] R. A. Van de Geijn, Using PLAPACK: Parallel Linear Algebra Package, MIT Press, Cambridge, MA, 1997.

[48] VI Architecture, *http://www.viarch.org*.

# 附 录

## §A.1 Linpack 性能测试

### §A.1.1 相关链接

- 中国高性能计算机 TOP100 排行榜
  - *http://www.samss.org.cn/*
- Top500 Supercomputer Sites
  - *http://www.top500.org*
- Clusters@TOP500
  - *http://clusters.top500.org/*
- Frequently Asked Questions on the Linpack Benchmark and Top500
  - *http://www.netlib.org/utk/people/JackDongarra/faq-linpack.html*
  - *http://netlib.amss.ac.cn/utk/people/JackDongarra/faq-linpack.html*

### §A.1.2 HPL 简介

- A Portable Implementation of High Performance Linpack Benckmark, 其前身称为 HPC (Linpack's Highly Parallel Computing benchmark).
- 作者: A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary, Innovative Computing Laboratory, Computer Science Department, University of Tennessee.
- 下载:

  *http://www.netlib.org/benchmark/hpl*

  或:

  *http://netlib.amss.ac.cn/benchmark/hpl* (国内镜像)
- 算法、功能与特点:
  - 行主元 *LU* 分解求解大型稠密线性代数方程组.
  - 消息传递并行, 适用于 MPP 和机群系统 (也可用于其它并行系统, 如 SMP 等).
  - 给出误差估计和程序的实际计算速度 (Gflops).
  - 高性能、高并行效率、高可扩展性.
  - 适用于大规模并行 Linpack 性能测试 (Highly Parallel Linpack benchmark)

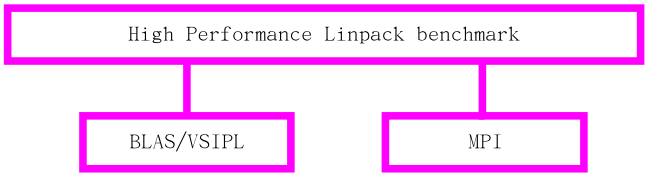



图 A.1: HPL 软件结构

### §A.1.3 适用于 Linux 的 BLAS 库

- 标准 Fortran/C 源程序:
  - *http://www.netlib.org/blas/*
  - *http://netlib.amss.ac.cn/blas/* (国内镜像)
  - 缺点: 性能低, 通常 $<\sim 10\%$ 峰值.
- ATLAS: Automatically Tuned Linear Algebra Software.
  - *http://www.netlib.org/atlas/*
  - *http://netlib.amss.ac.cn/atlas/*
  - 性能较高
- ASCI–Red BLAS: *http://www.cs.utk.edu/ ghenry/distrib*
- Intel Math Kernel Library: *http://developer.intel.com/software/products/mkl/*
- Kazushige Goto's BLAS: *http://www.cs.utexas.edu/users/flame/goto/*

### §A.1.4 HPL 的编译、运行与性能优化

- 下载源代码:
  - *http://www.netlib.org/benchmark/hpl/hpl.tgz*
  - *http://netlib.amss.ac.cn/benchmark/hpl/hpl.tgz* (国内镜像)
- 展开 HPL 源代码:

```
% tar xpvf hpl.tgz
% cd hpl
```

- 注意: 应该在用户主目录下展开, 否则 `make` 时会出错.
- 仔细阅读文件 `INSTALL` 中的安装说明.
- 在 `setup` 目录中选择一个文件 `Make.xxxx`, 将其拷贝至顶层当前目录, 如:

```
% cp setup/Make.Linux_PII_FBLAS .
```

- 根据编译器、MPI 与 BLAS 的情况对 `Make.xxxx` 做相应修改, 然后键入 "`make arch=xxxx`", 如:

```
% make arch=Linux_PII_FBLAS
```

- 编译完成后将在 `bin/xxxx/` (上例中为 `bin/Linux_PII_FBLAS/`) 下产生文件 `xhpl` 和 `HPL.dat`

- 进入可执行程序子目录 `bin/xxxx`, 如:

  ```
  % cd bin/Linux_PII_FBLAS
  ```

- 编辑、修改 `HPL.dat` 中的参数. 关于如何寻找最优的参数可阅读文件 `TUNING` 中的说明.

- 运行 `xhpl`, 如:

  ```
  % mpirun -np 4 xhpl
  ```

  (注意 MPI 进程数要对应 `HPL.dat` 中的参数 `P` 和 `Q`).

- 如何获得好性能?

  – 选用适当的 BLAS 或 VSIPL 库.

  – 寻找最优分块大小 `NB`, 以及最优处理器划分方式 `P` 和 `Q`.

  – 在内存容量许可的前题下尽量加大测试问题的规模.

  – 其它参数的选取.

以下是 HPL 程序输出示例，其中 `T/V` 列出了 `HPL.dat` 中定义的主要参数, `N` 为方程组阶数, `NB` 为分块大小, `P` 和 `Q` 为处理器划分方式, `Time` 为计算时间 (墙上时间), `Gflops` 为程序所达到的实际性能 (即 Linpack 性能, 以十亿次/秒为单位)。

```
1  ================================================================================
2  HPLinpack 1.1  --  High-Performance Linpack benchmark  --  January 1, 2001
3  Written by A. Petitet and R. Clint Whaley,  Innovative Computing Labs.,  UTK
4  ================================================================================
5  ... ... (略)
6  ================================================================================
7  T/V                N    NB     P     Q               Time             Gflops
8  --------------------------------------------------------------------------------
9  W02C2R1         2000    64     2     2               6.33          8.434e-01
10 --------------------------------------------------------------------------------
11 ||Ax-b||_oo / ( eps * ||A||_1  * N        ) =        0.0147384 ...... PASSED
12 ||Ax-b||_oo / ( eps * ||A||_1  * ||x||_1  ) =        0.0035825 ...... PASSED
13 ||Ax-b||_oo / ( eps * ||A||_oo * ||x||_oo ) =        0.0007784 ...... PASSED
14 ================================================================================
15 ... ... (略)
```