

# 第七章 MPI 编程及性能优化

与多线程、OpenMP 并程序不同，MPI 是一种基于消息传递的并行编程技术。本章将详细介绍-MPI 并行编程技术，以及 MPI 程序性能分析与优化方法。

MPI 编程是基于消息传递的并行编程技术，是如今应用最为广泛的并程序开发方法。到底消息传递并程序与基于共享存储的多线程、OpenMP 程序有何不同，如何进行消息传递，以及如何进行 MPI 程序性能分析，这些内容将在本章一一展开介绍。

希望通过本章的学习，读者能够了解 MPI 并程序的特点，掌握简单的 MPI 函数使用方法和程序优化手段，可以自己动手编写和分析、优化简单的 MPI 并程序。

## 第 1 节 MPI 简介

### 1.1 MPI 及其历史

与 OpenMP 相似，消息传递接口（Message Passing Interface，简称 MPI）是一种编程接口标准，而不是一种具体的编程语言。该标准是由消息传递接口论坛（Message Passing Interface Forum，简称 MPiF）发起讨论并进行规范化的。

细细算来，MPI 标准从 1992 年开始起草，1994 年发布第一个版本 MPI-1（MPI v1.0，进而发展出 1.1 和 1.2 版），到 1997 年发布第二个版本 MPI-2（MPI v2.0），直至今天，已经有十五年的历史了。经过这十五年的改进和完善，也伴随着高性能计算技术的普及，尤其是集群系统的普及，MPI 标准如今已经成为事实意义上的消息传递并行编程标准，也是最为流行的并行编程接口。

简而言之，MPI 标准定义了一组具有可移植性的编程接口。各个厂商或组织遵循这些标准接口实现自己的 MPI 软件包，典型的实现包括开源的 MPiCH、LAM MPI 以及不开源的 INTEL MPI。而对于程序员来说，设计好应用程序并行算法，调用这些接口，链接相应平台上的 MPI 库，就可以实现基于消息传递的并行计算。也正是由于 MPI 提供了统一的接口，该标准受到各种并行平台上的广泛支持，这也使得 MPI 程序具有良好的可移植性。目前，MPI 支持多种编程语言，包括 Fortran77，Fortran90 以及 C/C++；同时，MPI 支持多种操作系统，包括大多数的类 UNIX 系统以及 Windows 系统（Windows 2000、Windows XP 等）；MPI 还支持多核(Multicore)、对称多处理机(SMP)、集群(Cluster)等各种硬件平台。。

### 1.2 典型 MPI 实现简介

MPI 是一个标准。它不属于任何一个厂商，不依赖于某个操作系统，也不是一种并行编程语言。不同的厂商和组织遵循着这个标准推出各自的实现，而不同的实现也会有其不同的特点。

在本节中，我们将简要介绍两种典型的 MPI 实现：MPiCH 和 Intel MPI，而 7.2.1 节和 7.2.2 节我们将进一步教大家如何在 Linux 和 Windows 操作系统上来安装、配置 MPiCH 软件包。

## 1. MPICH

MPICH 是影响最大、用户最多的 MPI 实现。它是由美国的 Argonne 国家实验室开发的开放源码的 MPI 软件包。它与 MPI 标准同步发展，方便易用。目前可下载的最新的 MPICH 软件包为 MPICH 1.2.7p1 和 MPICH 2-1.0.5p4，分别遵循 MPI-1.2 和 MPI-2 两个版本的 MPI 标准。如果你想要使用这些软件，可以到如下网址下载：

<http://www-unix.mcs.anl.gov/mpi/mpich1/download.html>

<http://www-unix.mcs.anl.gov/mpi/mpich2/index.htm#download>

MPICH 的特点在于：

- 开放源码；
- 与 MPI 标准同步发展；
- 支持多程序多数据（Multiple Program Multiple Data, MPMD）编程和异构集群系统；
- 支持 C/C++、Fortran 77 和 Fortran 90 的绑定；对 Fortran 的支持提供了头文件 `mpif.h` 和模块两种方式；
- 支持类 Unix 和 Windows NT 平台；
- 支持环境非常广泛，包括多核、SMP、集群和大规模并行计算系统；

除此之外，MPICH 软件包中还集成了并程序序设计环境组件，包括并行性能可视化分析工具和性能测试工具等。

## 2. Intel MPI

Intel MPI 是由 Intel 公司推出的符合 MPI-2 标准的 MPI 实现。其最新版本是 3.0 版，突出的特色在于提供了灵活的多架构支持。

Intel MPI 提供了名为 Direct Access Programming Library（DAPL）的中间层来支持多架构，兼容多种网络硬件及协议，优化网络互联。Intel MPI 库及 DAPL 互联结构可用图 7.1.1 清晰地表示出来。

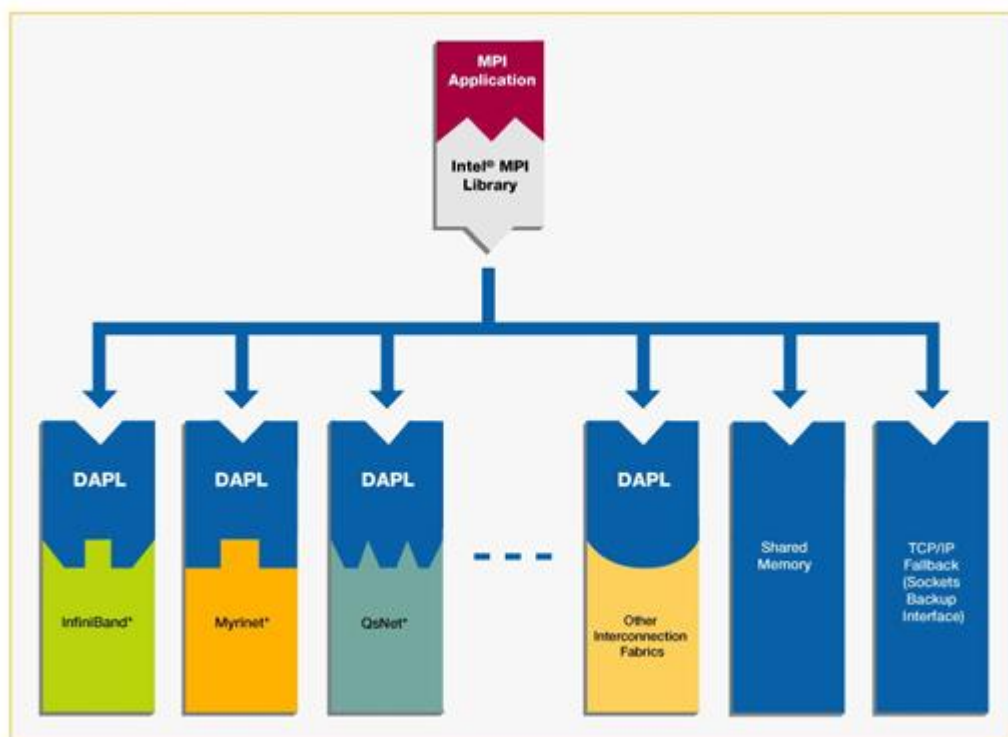


图 7.1 Intel MPI 库及其基于 DAPL 的互联结构

从图 7.1 可以看出，Intel MPI 透明地支持 TCP/IP、共享内存，并基于 DAPL 有效支持

多种高性能互联系统,例如,如今迅速普及的 InfiniBand,在并行计算领域广泛使用的 Myrinet。基于此, Intel MPI 在通信协议的选择上无需进行额外设置,可自动选择 MPI 进程间最快的传输协议,使用更加简单和有效。而这点也对于同时具备多核、多处理器和集群架构的并行系统尤为重要,通过自动选择通信方式, Intel MPI 将尽可能充分利用多核、多处理器和互联网络不同层次的通信能力。

此外, Intel MPI 提供更好的线程安全机制,多线程的 MPI 程序并不限制 MPI 的使用。

大家可以访问 <http://www3.intel.com/cd/software/products/apac/zho/329245.htm> 进一步了解 Intel MPI 或者下载使用其最新的评估版。

## 1.3 MPI 程序特点

MPI 程序是基于消息传递的并行程序。消息传递指的是并行执行的各个进程具有自己独立的堆栈和代码段,作为互不相关的多个程序独立执行,进程之间的信息交互完全通过显式地调用通信函数来完成。这与多线程、OpenMP 程序共享同一内存空间有着显著的不同,而且有利有弊。好处在于带来了更多的灵活性,除了支持多核并行、SMP 并行之外,消息传递更容易实现多个节点间的并行处理;而另一方面,也正是源于这种进程独立性和显式消息传递的特点, MPI 标准更加繁复,基于其开发并行程序也更加复杂。

基于消息传递的并行程序可以划分为单程序多数据 (Single Program Multiple Data, 简称 SPMD) 和多程序多数据 MPMD 两种形式。SPMD 使用一个程序来处理多个不同的数据集以达到并行的目的。并行执行的不同程序实例处于完全对等的位置。相应的, MPMD 程序使用不同的程序处理多个数据集,合作求解同一个问题。

SPMD 是 MPI 程序中最常用的并行模型。图 7.2 为 SPMD 执行模型的示意图,其中表示了一个典型的 SPMD 程序,同样的程序 `prog_a` 运行在不同的处理核上,处理了不同的数据集。

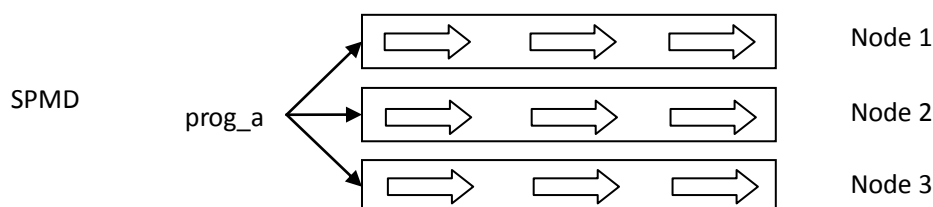


图 7.2 SPMD 执行模型

为了更好地说明 SPMD 程序是如何工作的,让我们来看看图 7.3 演示的一个串行程序的执行过程。程序先在外存中读取一个数组,并对数组中的每一个元素进行相同的处理,最后将处理完的数组中的每一项数据写入到外存中。

由于 SPMD 程序实际上是运行同样的一个程序,只不过处理不同的数据集,因此,在 SPMD 程序中,每一个进程会有一个进程号(在 MPI 中被称为进程号 `rank`)用来相互区别。在 SPMD 中,就用这个进程号来确定不同的任务。图 7.3 说明了用 `rank` 值来区别不同进程,进而处理不同数据的情况。在这个例子中,每一个进程只需要处理数据总量的三分之一。

上图中,每一个进程首先都读入数组数据,并且根据不同的 `rank` 值来处理不同的数据。由于并行处理的过程中,数据被分割到不同的节点上,因此需要通过某一种手段最后将数据

集合在一起。在上述并行政序的数据处理完毕之后，通过消息传递的办法来收集数据处理的结果，最后由 0 号进程将结果写入到磁盘。如何有效地做到这点？就需要消息传递，来将多个由于并行化而分开的进程有效联系在一起，协同地完成工作。而 **MPI** 标准和实现正式提供了标准的通信接口及其相应的底层软件。

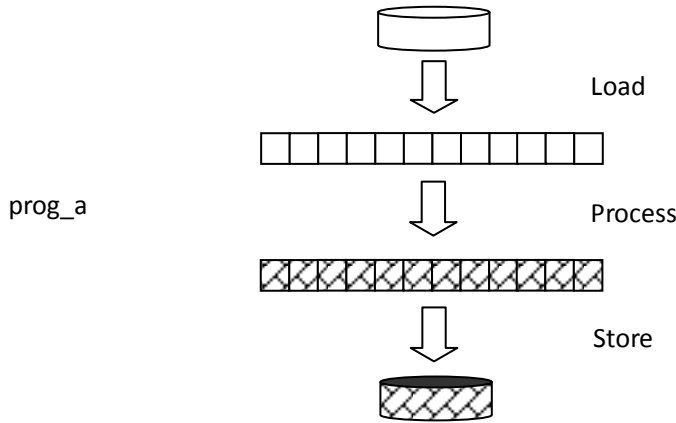


图 7.3 串行处理程序的运行过程

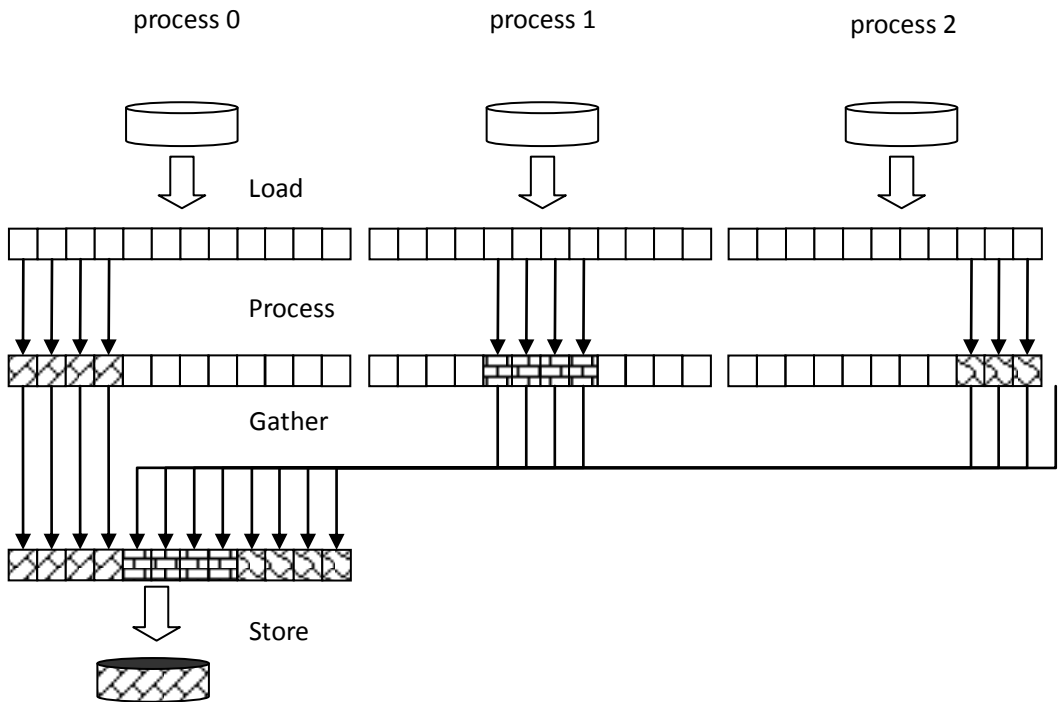


图 7.4 SPMD 程序的并行执行过程

图 7.5 则给出了三种典型 **MPMD** 程序的执行模型。

(a) 是一个管理者(Master)/工人(Worker)类型的 **MPMD** 程序，由一个管理者(Master)程序 `prog_a` 来控制整个程序的执行，并将不同的任务分配给多个工人(Worker)程序 `prog_b` 来完成工作。

(b) 为另外一种类型的 **MPMD** 程序：联合数据分析程序。在大部分的时间内，不同的程序各自独立的完成自己的任务，并在特定的时候交换数据。一般来说，这种方式的并行程序进程间的耦合性最少，通信也少，更容易获得好的并行加速效果。

(c) 是流式的 MPMD 程序，程序运行由 prog\_a、prog\_b 和 prog\_c 组成，这三个程序的执行过程就好像工厂里的流水线一样。对于一个任务而言，prog\_a 处理后的输出给 prog\_b 作为输入，prog\_b 的输出再给 prog\_c 作为输入，这三个程序之间是典型的串行执行。在这种情况下，并行性的取得依赖于执行大量的任务，通过这种流水线获得性能加速。在多核环境中，一个处理器内的多个处理核之间通信带宽高而延迟低，能够有效利用流式模型提高计算性能。

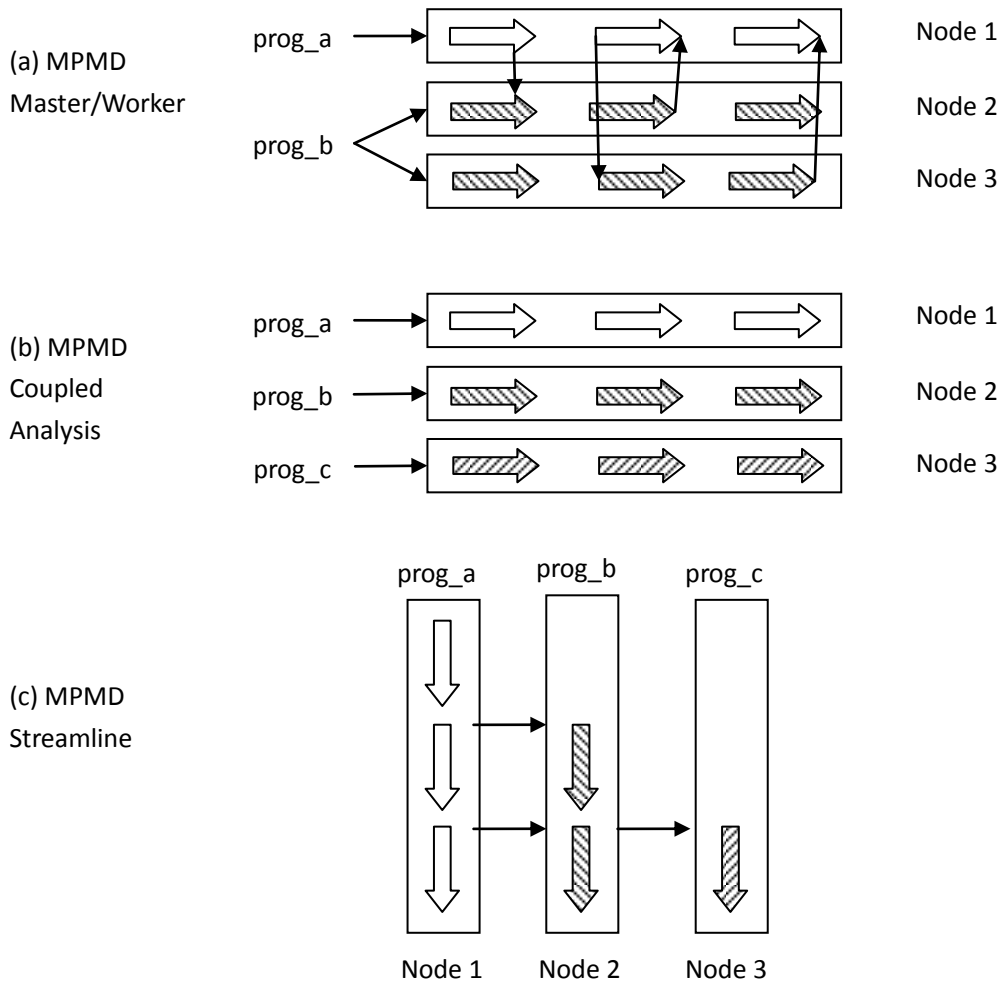


图 7.5 MPMD 执行模型

记住本小节介绍的 MPI 程序的特点，以及 SPMD 和 MPMD 的多种执行模型，读者就可以在以后的开发过程中灵活地设计不同的 MPI 并程序。

## 1.4 本章内容组织

本小节（第 7.1 节）主要介绍 MPI 编程技术的一些背景知识，包括：MPI 标准的历史和发展，MPI 程序的特点。希望读者能够了解 MPI 程序（消息传递程序）与多线程、OpenMP 程序在程序视图（SPMD 和 MPMD）、进程视图（多个进程视图完全分离）和通信管理（显式管理各种通信）等方面的不同，为后续章节更好地理解 and 掌握 MPI 编程技术打下基础。

MPI 实现多种多样，且都是与编译器、具体编程语言相独立的软件包，这为 MPI 软件包的安装和使用带来了问题。第 7.2 节我们将详细讲解应用最广泛的 MPICH 软件包在 Linux 和 Windows 上的具体安装和配置过程。读者可以参照我们的提示自己动手安装 MPI 软件，快速开启自己的 MPI 之旅。

第 7.3 节将重点讲解 MPI 编程最基础，也是最重要的部分。这一节会通过实例来帮助读者由浅入深开始 MPI 并行程序的开发。而第 7.4 节则对另一种 MPI 通信机制—群集通信进行简要介绍。

编写并行程序是为了利用冗余硬件（例如多核、多处理器或多机）提高应用性能。但往往事与愿违，程序并行后性能提升不明显，甚至是性能降低了该怎么办。第 7.5 节我们给出了性能分析和性能优化的一些方法和技巧。

最后，值得一提的是，MPI 标准并不是一成不变，而是仍在不断完善和发展当中。本章介绍的内容集中于 MPI-1 的核心部分。而对于 MPI-2 提出的高级功能将在第 7.6 节中将做简要说明。

## 第2节 MPICH 的安装和配置

正如 MPICH-2 主页 (<http://www-unix.mcs.anl.gov/mpi/mpich2/>) 上所说的, MPICH-2 将要取代 MPICH-1 成为最流行的 MPI。因此,在这一节中,我们将叙述如何在 Linux 和 Windows 上安装和配置自己的 MPICH-2 软件包。而象 Intel MPI 这样的商业软件则有更简便的安装界面和完善的配置文档可供参阅。

### 2.1 在 LINUX 上安装和配置 MPICH-2

#### Step 1: MPICH-2 的下载

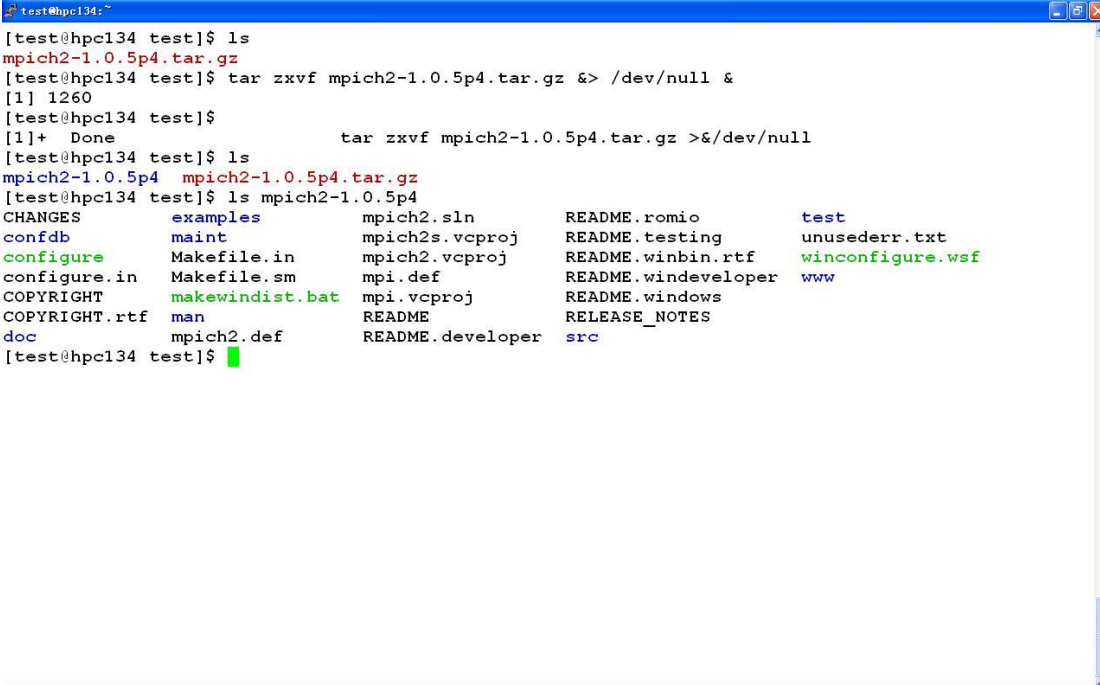
MPICH 采用了 GNU Bin 工具进行维护和发布。软件的稳定版本可在 <http://www-unix.mcs.anl.gov/mpi/mpich2/index.htm#download> 下载。目前其最新版本为 1.0.5p4, 因此下载到的源码安装包是 mpich2-1.0.5p4.tar.gz。

为方便说明,特将笔者采用的系统环境配置描述如下:双路(即双处理器),每处理器 4 核 2.66GHz 高性能服务器,主机名为 hpc134,操作系统为 Fedaro 2, gcc 的版本为 3.3.3。所有操作均以用户 test 执行,用户主目录为/home/test,使用 bash 环境。

将 mpich2-1.0.5p4.tar.gz 文件拷贝到/home/test/目录下,使用

**tar zxvf mpich2-1.0.5p4.tar.gz &> /dev/null &**

解压缩,如图 1-1 所示。



```
test@hpc134:~  
[test@hpc134 test]$ ls  
mpich2-1.0.5p4.tar.gz  
[test@hpc134 test]$ tar zxvf mpich2-1.0.5p4.tar.gz &> /dev/null &  
[1] 1260  
[test@hpc134 test]$  
[1]+  Done                  tar zxvf mpich2-1.0.5p4.tar.gz >&/dev/null  
[test@hpc134 test]$ ls  
mpich2-1.0.5p4  mpich2-1.0.5p4.tar.gz  
[test@hpc134 test]$ ls mpich2-1.0.5p4  
CHANGES      examples      mpich2.sln    README.romio  test  
conftdb       maint         mpich2s.vcproj README.testing unusederr.txt  
configure     Makefile.in  mpich2.vcproj README.winbin.rtf winconfigure.wsf  
configure.in  Makefile.sm  mpi.def       README.windeveloper www  
COPYRIGHT     makewindist.bat mpi.vcproj    README.windows  
COPYRIGHT.rtf man          README       RELEASE_NOTES  
doc           mpich2.def   README.developer src
```

图 7.2.1 解压缩 MPICH 源程序文件

## Step 2: MPICH2 的安装

首先进入 mpich2-1.0.5p4 目录（命令行为 `cd mpich2-1.0.5p4`）。

安装之前先利用 autoconf 机制进行配置。最本地，通过 configure 脚本指定安装目标路径即可。我们这里的安装路径设为/home/test/mpich2-1。相应的命令行为：

**`./configure --prefix=/home/test/mpich2-1`**

如果配置成功，使用 **`make;make install`** 命令即可完成软件包安装。

## Step 3: MPICH-2 的配置与验证

MPI 进程的创建、启动和管理需借助进程管理器（PM）来完成。直观地讲，PM 就是 MPI 环境与操作系统的接口。MPICH 提供了多种进程管理器，本章的测试环境均使用 mpd，因此这里仅以 mpd 为例，说明环境的建立和配置过程。

安装成功后，还需要进一步配置 MPICH 的执行环境。可能的工作有以下几部分。

### 1. 需要 Python 软件

MPD 由 python 实现的一组工具构成，因此首先需确保机器上已经安装了正确版本的 python 解释器，而大部分 linux 操作系统实现都会内置 python。需要注意的是 2-1.0.5p4 要求 2.2 版本或以上的 python 解释器，如果你的系统上没有，可在 <http://www.python.org> 下载并安装。

### 2. 设置主目录下的 mpd.hosts 文件

mpdboot 在主机上启动 mpd 进程，形成 MPI 运行时环境，目标主机列表由 \$HOME/mpd.hosts 文件指定。我们在其中加入了一行：

**hpc134**

### 3. 配置 SSH 无密码登录

在利用 mpdboot 启动之前，需配置集群环境支持节点间的无密码登录，该功能需要 ssh 的支持，不过一般 linux 系统中都会安装 ssh 软件。配置的具体做法是：

#### 1. 在各个节点上运行命令

**`ssh-keygen -t rsa`**

如果希望使用 dsa 加密算法，则以 dsa 替换上述命令中的 rsa 即可。

#### 2. 拷贝上述产生的文件（.ssh/id\_rsa.pub）生成.ssh/authorized\_keys2，并设置相应文件访问权限

**`chmod go-rwx /home/test/.ssh/authorized_keys2`**

以上利用了 ssh 提供的密钥加密算法进行身份认证，也可采取 host based 等机制配置无密码登录，详细可参考 ssh 手册。

此外，出于安全考虑，需在 \$HOME/目录下提供“.mpd.conf”文件，简单地，该文件可以仅包含一行，如“MPD\_SECRETWORD=mypasswd”即可。注意，.mpd.conf 的文件访问权限必须设置为“600”。



4. 命令和手册页路径的配置

为方便 MPI 的使用，可将如下语句添加到主目录（/home/test/）下的.bashrc 文件中，从而方便地定位执行文件和通过 man 手册页查阅文档：

```
export PATH=/home/test/mpich-2.1/bin:$PATH
export MANPATH=/home/test/mpich-2.1/man:$MANPATH
```

存盘退出后，运行 `source /home/test/.bashrc` 使上述设置生效。

Step 4: 启动 MPD

在安装和上述配置结束后，即可启动 mpd 进程，创建 mpi 运行时环境了。具体的命令为：


**mpiboot**

表 7.2.1 中列出了与 mpd 有关的命令。

| 表 7.2.1 mpd 相关命令说明 |                                   |
|--------------------|-----------------------------------|
| 命 令                | 说 明                               |
| mpd                | 启动 mpd 守护进程                       |
| mpdtrace           | 打印 mpi 运行时环境内所有 mpd 守护进程的信息       |
| mpdboot            | 启动一组 mpd 进程                       |
| mpdringtest        | 测试消息在环境环行一周的时间                    |
| mpdallexit         | 停止运行时环境的所有进程（mpd 进程）              |
| mpdcleanup         | 运行时环境崩溃情况下，可用该命令清除本地的 Unix socket |
| mpdlistjobs        | 列出作业的进程信息                         |
| mpdkilljob         | 停止某个作业的所有进程                       |
| mpdsigjob          | 对某个作业的所有进程发送信号                    |
| mpiexec            | 启动一个作业                            |

mpd 是环境的守护进程，而其它工具则通过与 mpd 进行通信来实现其功能。关于命令行选项的详细说明可通过（--help）察看帮助选项。

通过 mpdboot 命令启动环境后，可通过 mpitrace 和 mpiringtest 验证环境是否正确运行。mpdboot 默认地通过 ssh 在远程节点上启动 mpd 进程。

 **指点迷津：**

上述的启动方式有一个缺点，即每个用户都需分别启动自己的 mpi 环境，通过 mpdboot 启动属于自己的守护进程 mpd。此举会带来较多的资源开销。可通过 root 用户进行配置，以使得可能运行作业的用户共享一个 mpi 运行时环境。

为此，首先需要以 root 用户身份安装（make install）MPI 环境，并在在/etc/mpd.conf 给出 MPD\_SECRETWORD 信息；其次在每个用户自己的\${HOME}/.mpd.conf 中增加一行

"MPD\_USE\_ROOT\_MPD=1"，或设置为环境变量即可。

## Step 5: 应用程序的编译、链接

源程序中使用了 MPI 调用之后，则需在编译和链接时链接到 MPI 库。MPI 库在 Linux 和 Mac 上可以静态链接库和动态链接库两种形式存在。为简化链接过程，MPICH 环境提供了形如 mpicc, mpif77, mpif90 等编译脚本。可以在源码目录中，通过如下命令来完成编译和链接：

**mpicc -o mypi cpi.c**

其中 mpicc 是 MPICH2 的编译脚本；-o mypi 是让编译器生成名为 mypi 的可执行文件；而 cpi.c 则是源码文件。



**指点迷津：**

- 编译应用程序时，应尽量采用 MPICH 安装时所使用的编译器。可通过环境变量 MPICH\_CC，MPICH\_CXX，MPICH\_F77，MPICH\_F90 指定其编译器。通过命令 mpich2version 可察看安装 MPICH 过程所使用的命令，配置参数，编译器及编译选项等信息。
- MPICH 在各种平台上均支持静态链接，在 lib/目录下保存了编译应用程序时可用的所有库，“\*.a”文件。
- C++/Fortran 90 等程序的编译如果出错，则可考虑检查 mpif.h，mpi.h 等，察看语法是否符合。例如 MPICH 同时支持 Fortran 77 和 Fortran 90，MPI 常量在 Fortran 77 环境中通过 mpif.h 提供定义，但在 Fortran 90 种则需要以 use 方式链接模块。但需注意，MPI 的 Fortran 90 模块提供的调用并不完整，如不支持可变参数列表。

## Step 6: MPI 程序的运行

MPI-2 标准建议使用 mpiexec 代替 mpirun 来启动应用程序。MPICH2 实现了 mpiexec 标准并通过 mpd 对标准作了适当扩展。实际上，mpiexec 提供了 MPI 环境与外界的强大接

口，构成了 MPI 进程管理环境。

类似 `mpirun` 命令，`mpiexec` 启动应用程序的标准命令可写作：`mpiexec -n 4 ./mympi`。该命令为 `mympi` 程序启动 4 个进程。

可通过 `--help` 参数看看还有哪些其它选项可供设置。常用的参数可以用来指定 `host` 列表、指定可执行程序的搜索路径、工作目录等，也可在多个不同进程组启动多个应用，并分别制定参数等等。

值得注意的是，MPD 进程管理器对标准的 `mpiexec` 进行了适当扩展。`mpiexec` 提供如下基本参数：`mpiexec -n <num_nodes> <executable>`。其中 `<num_nodes>` 表示程序需要运行的节点个数；`<executable>` 为可执行程序名字，可能为 `mpi` 程序也可非 `mpi` 程序。如：`mpiexec -n 9 ./mpi_app` 将在 9 个节点上运行 `mpi_app` 程序，`mpd` 负责在其运行时环境中选取足够数量的节点。如 `mpiexec -n 4 ls -a`，将在 4 个节点上列出“当前”路径。

## 2.2 在 Windows 上安装 MPICH-2

MPICH 为不同硬件平台提供了不同的 windows 安装程序，也可以方便地到 <http://www-unix.mcs.anl.gov/mpi/mpich2/index.htm#download> 下载。

表 7.2.2 当前不同硬件平台下 Windows MPICH 下载包信息

| 平 台               | 下载包名称                           |
|-------------------|---------------------------------|
| Win32 IA32        | mpich2-1.0.5p2-win32-ia32.msi   |
| Win64 EM64T/AMD64 | mpich2-1.0.5p2-win64-x86-64.zip |
| Win64 IA64        | mpich2-1.0.3-1-win64-ia64.zip   |

在 Windows 下面安装 MPICH-2 非常简单，只需要执行相应的安装程序即可。例如在 32 位的 Windows XP 平台下，运行安装程序 `mpich2-1.0.5p2-win32-ia32.msi`。值得注意的是，安装这个程序需要 .net framework version 2.0 的支持。下面是安装开始之后的信息界面：

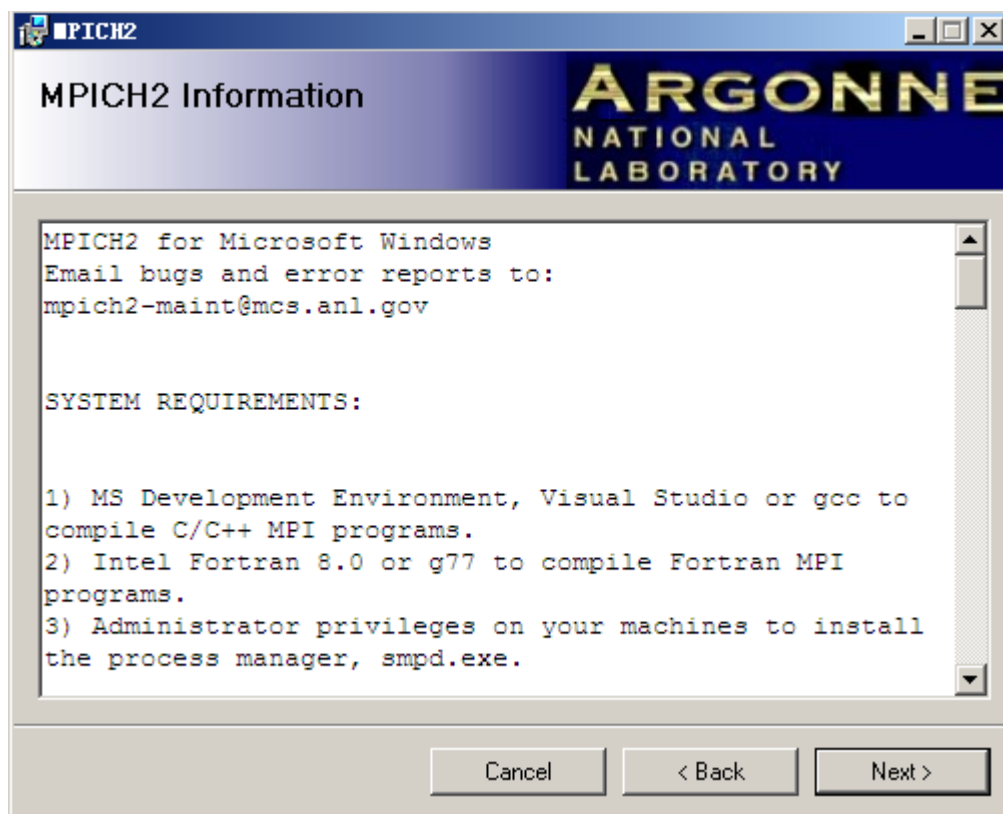


图 7.2.2 在 Windows 环境下安装 MPICH2 编程环境

在安装的过程，基本上只需要点击下一步（Next）即可，无需作其它的配置。安装过程需要注意的是进程管理器的密码配置，这个密码被用来访问所有的 smpd 服务程序，因此需要长期保存下来。另外，安装程序提示只能通过管理员才能够安装相应的程序，如果不是管理员组的成员是无法安装 MPICH2 的。下图给出了相应的进程管理器访问密码配置界面，默认的访问密码是 behappy。

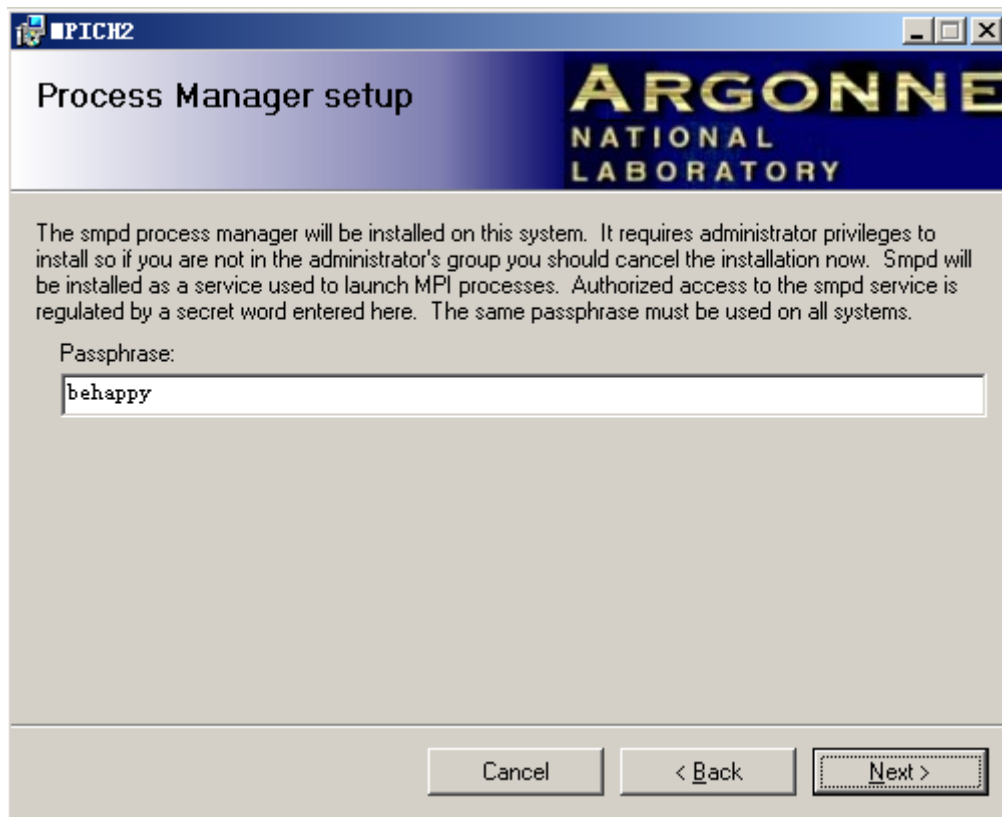


图 7.2.3 配置进程管理器访问密码

使用默认的安装选项完成安装会在系统盘（一般为 C:）中创建 MPICH2 的安装目录，比如 C:\Program Files\MPICH2。在这个目录下面，子目录 include 包含了编程所需要的头文件，子目录 lib 包含了相应的程序库，而 bin 则包含了 MPI 在 Windows 下面必需的运行程序，如 smpd.exe 进程管理程序，以及 mpiexec.exe 用来启动 MPI 程序的运行。另外，在运行的时候需要的动态链接库（dll）在安装的时候被默认地拷贝到 Windows 的系统目录中（默认为 C:\Windows\System32），因此无需考虑运行路径问题。为了避免不必要的麻烦，最好在所有的运行节点上都进行安装，最小安装的问题可以参考联机文档。

为了编写 MPI 并行程序，在 Windows 下面，一般使用 Microsoft 提供的 Visual Studio 来编写。下面以 Microsoft Visual Studio .Net 2003 为例说明编写 MPI 程序的步骤。

第一步：建立 Visual Studio .Net 2003 项目，创建命令行界面的应用程序即可。

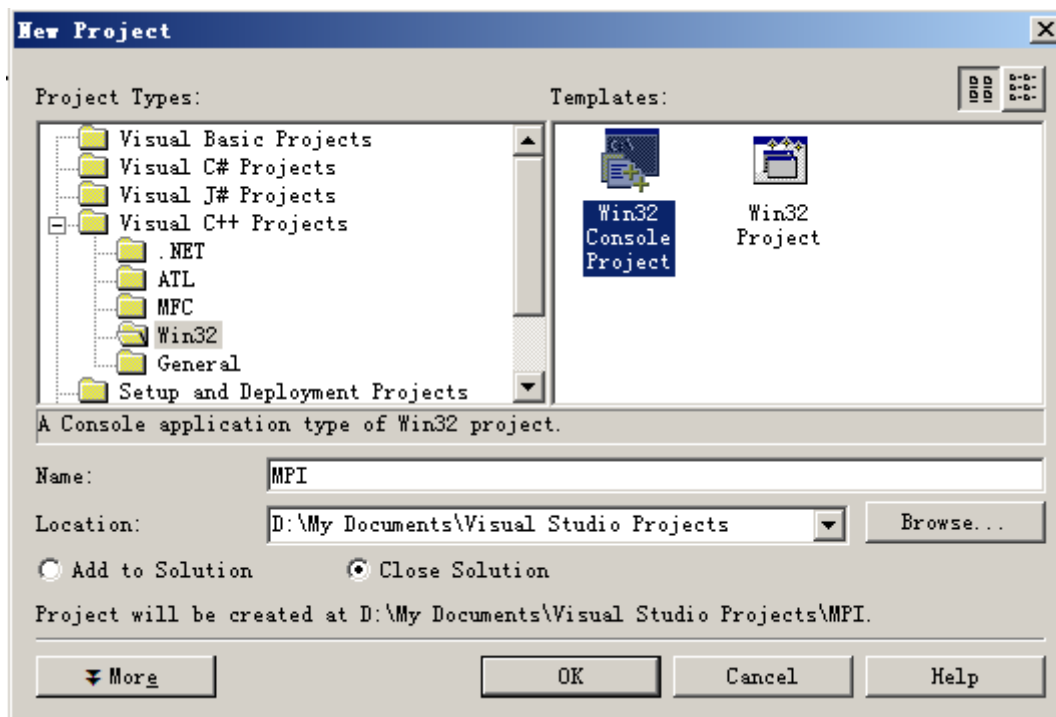


图 7.2.4 生成默认的 Win32 Cosole 项目

第二步：将 mpich2\include 加入到头文件目录中，配置选项在菜单 Tools→Options→Projects→VC++ Directories 对话框中。

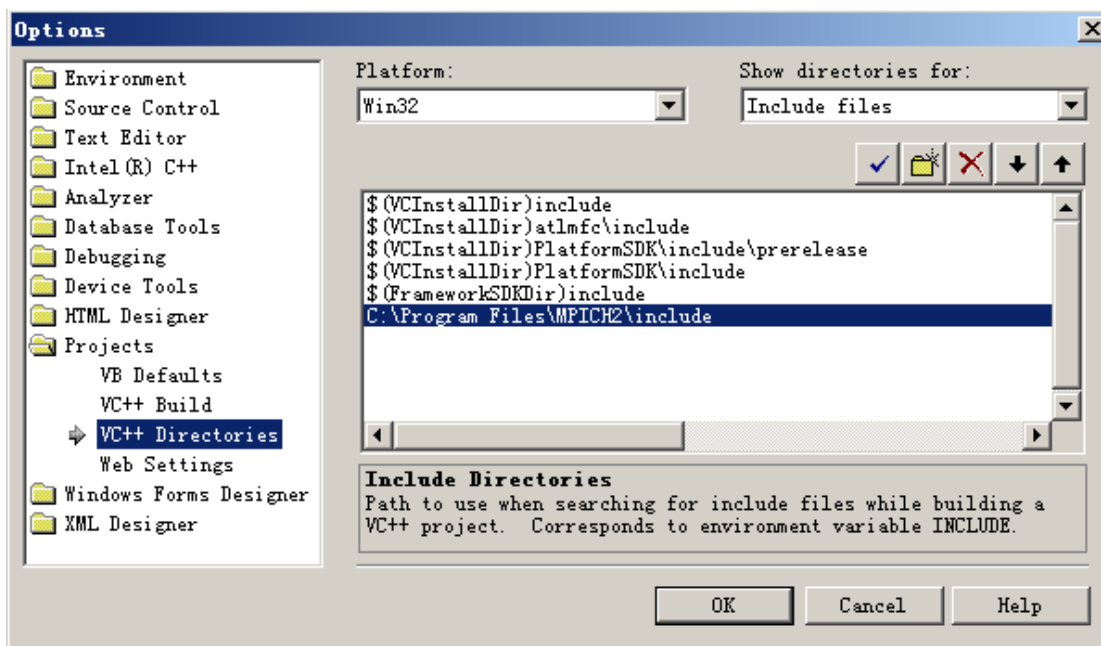


图 7.2.5 配置头文件目录

第三步：将 mpich2\lib 加入到库文件目录中，配置选项在菜单 Tools→Options→Projects→VC++ Directories 对话框中。

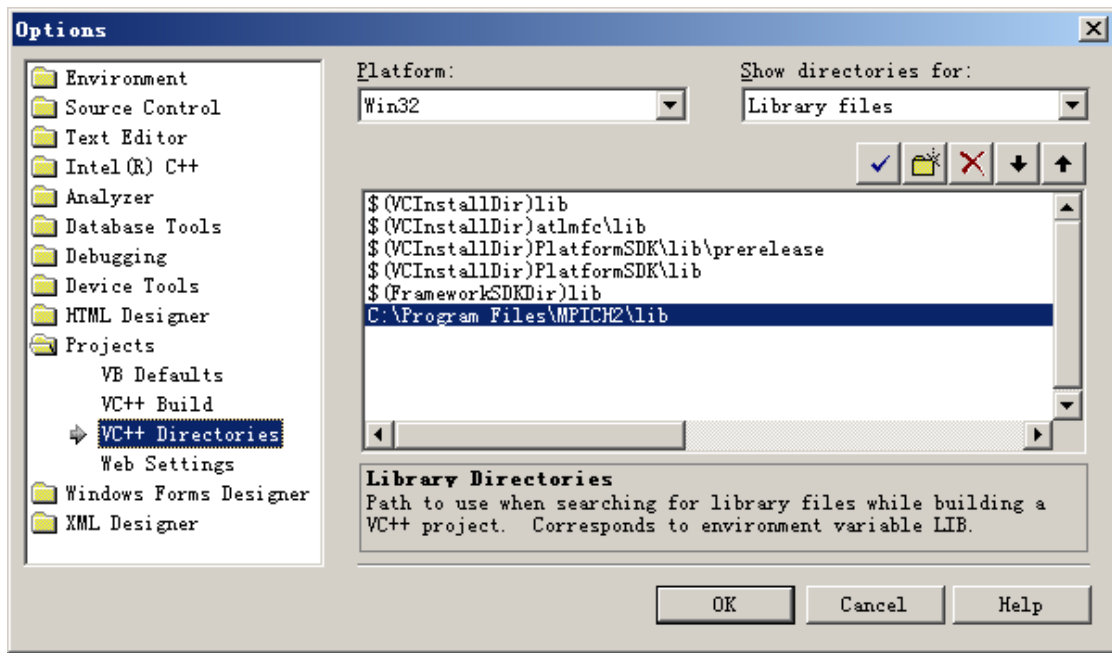


图 7.2.6 配置库文件目录

第四步：设置项目的属性，将 mpi.lib 加入到链接库中。

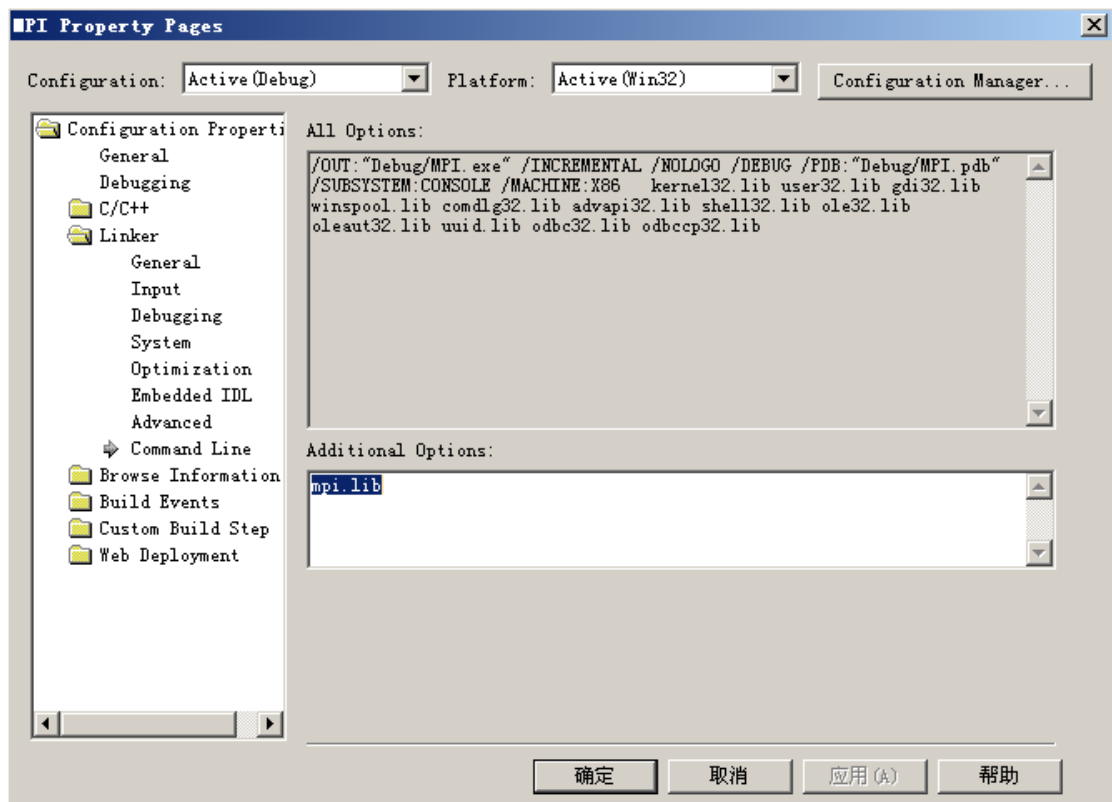


图 7.2.7 配置程序链接库

如果是 C 程序的话，此时就可以编译程序了。但是由于 Visual Studio.Net 2003 默认生成的是 C++ 的应用程序，并且有预编译头（precompiled header）的支持。所以，在编写程序的时候需要采取一些特殊的手段来避免名字冲突。在这里，我们可以将 MPI 的头文件 mpi.h 放在预编译头文件 stdafx.h 的第一句，下面是一个支持 MPI 编译的预编头文件的例子：

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#pragma once

#include "mpi.h"

#include <iostream>
#include <tchar.h>
#include <math.h>

// TODO: reference additional headers your program requires here
```

至此，就可以在主程序文件中编写 MPI 程序，编译和链接生成可运行的应用程序。将生成的程序以及相应的动态链接库（dll，在默认安装下被放置到 Windows 的系统目录中）拷贝到所有的运行节点机或者放在一个共享的目录中，然后就可以启动 mpiexec.exe 来运行 MPI 程序了。

## 第 3 节 MPI 编程基础

在这一小节中，我们将对 MPI 的基本用法进行讲解。本章所采用的例子都是用 C 语言编写，在 linux 平台上运行。我们假定读者具有基本的 C/C++ 语言的基础，这样就可以按照本书的次序，由易到难逐步掌握 MPI 并行程序的编程方法和技巧了。另外，读者也完全可以方便地将本章提供的示例程序移植到 Windows 平台上执行。

### 3.1 简单的 MPI 程序示例

首先，我们来看一个简单的 MPI 程序实例。如同我们学习各种语言的第一个程序一样，对于 MPI 的第一个程序同样是"Hello Word"。

```
/* Case 1 hellow.c */
#include <stdio.h>
#include "mpi.h"

int main ( int argc, char *argv[] ) {
int rank;
int size;

    MPI_Init ( argc, argv );
    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
```



```

MPI_Comm_size (MPI_COMM_WORLD, &size) ;
printf ( "Hello world from process %d of %d\n", rank, size ) ;
MPI_Finalize ( ) ;
return 0;
}

```

根据上一节的介绍，我们使用如下命令编译和链接这个程序：

**mpicc -o hellow hellow.c**

运行这个例子可以在可执行文件的目录中执行 **mpiexec -np 4 ./hellow**。运行结果如下：

```

Hello world from process 0 of 4
Hello world from process 1 of 4
Hello world from process 2 of 4
Hello world from process 3 of 4

```

这个程序在 MPI 程序运行的每个进程中分别打印各自的 MPI 进程号（0~3）和总进程数（4）。



**指点迷津：**

由于四个进程是并行执行，所以即使输出的顺序有变化也是正常的，程序中并

没有限制哪个进程在前，哪个进程在后。

## 3.2 MPI 程序的四个基本函数

现在让我们进一步分析上面的 hellow.c。

首先，我们可以看出，要使用 MPI 函数库，首先要在前面包含 mpi.h。当然如果是 Fortran 的程序则需要使用 mpif.h 或者应用相应的 module。

在这个例子程序的主函数中，先是变量声明，然后调用了四个 MPI 函数，和一个 printf 函数。这四个函数都是 MPI 最重要和最常用的函数。下面让我们一一作说明。

### 1. MPI\_Init 和 MPI\_Finalize

MPI\_Init 用来初始化 MPI 执行环境，建立多个 MPI 进程之间的联系，为后续通信做准备。而 MPI\_Finalize 则是结束 MPI 执行环境。

如同 OpenMP 定义并行区一样，这两个函数就是用来定义 MPI 程序的并行区的。也就是说，除了检测是否初始化的函数之外，不应该在这两个函数定义的区域之外调用其他 MPI 函数。一般的 MPI 程序都会在主函数开始和结束分别调用这两个函数。

当然，这并不是说 MPI\_Init 之前和 MPI\_Finalize 之后一定不能有其他程序语句。MPI 标准没有定义在 MPI\_Init 调用之前和 MPI\_Finalize 之后的行为。与 OpenMP 并行区之外为单线程执行不同，大多数 MPI 实现都会在各个并行进程中独立地执行相应的代码。

C 语言的 MPI\_Init 接口需要提供 argc 和 argv 参数；MPI\_Finalize 函数则不需要提供任何参数。MPI\_Init 和 MPI\_Finalize 都返回整型值，标识函数是否调用成功。

## 2. MPI\_Comm\_rank

第 7.1 节介绍过 SPMD 的程序形式, 给出的例子中需要通过进程标识和总数来分配数据。MPI\_Comm\_rank 就是来标识各个 MPI 进程的, 告诉调用该函数的进程“我是谁?”。MPI\_Comm\_rank 返回整型的错误值, 需要提供两个函数参数:

- MPI\_Comm 类型的通信域, 标识参与计算的 MPI 进程组。上面的例子中是 MPI\_COMM\_WORLD, 这个进程组是 MPI 实现预先定义好的进程组, 指的是所有 MPI 进程所在的进程组。如果你想要申请自己的特殊的进程组, 则需要通过 MPI\_Comm 定义并通过其他 MPI 函数生成。



**多学两招:**

MPI 实现还会预先定义另外一个进程组 MPI\_COMM\_SELF, 只包含各个进程自己的进程组。

- 整型指针, 返回进程在相应进程组中的进程号。进程号从 0 开始编号。

## 3. MPI\_Comm\_size

本函数则用来标识相应进程组中有多少个进程。MPI\_Comm\_size 也返回整型的错误值, 同时有两个函数参数:

- MPI\_Comm 类型的通信域, 标识参与计算的 MPI 进程组。上面的例子中是 MPI\_COMM\_WORLD。
- 整型指针, 返回相应进程组中的进程数。

## 3.3 MPI 的点对点通信

点对点通信是 MPI 编程的基础。本节我们将重点介绍其中两个最重要的 MPI 函数 MPI\_Send 和 MPI\_Recv。

还是先来看一个完整的例子。

```
/* Case 2 srtest.c */

#include "mpi.h"
#include <stdio.h>
#include <string.h>

#define BUFLLEN 512

int main (int argc, char *argv[])
{
    int myid, numprocs, next, namelen;
    char buffer[BUFLLEN], processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;
```

```

MPI_Init (&argc,&argv) ;
MPI_Comm_size (MPI_COMM_WORLD,&numprocs) ;
MPI_Comm_rank (MPI_COMM_WORLD,&myid) ;
MPI_Get_processor_name (processor_name,&namelen) ;

printf ("Process %d on %s\n", myid, processor_name) ;
printf ("Process %d of %d\n", myid, numprocs) ;
memset (buffer, 0, BUFLen*sizeof (char)) ;
if (myid == numprocs-1)
    next = 0;
else
    next = myid+1;

if (myid == 0)
{
    strcpy (buffer,"hello there") ;
    printf ("%d sending '%s' \n",myid,buffer) ;fflush (stdout) ;
    MPI_Send (buffer, strlen (buffer) +1, MPI_CHAR, next, 99, MPI_COMM_WORLD) ;
    printf ("%d receiving \n",myid) ;fflush (stdout) ;
    MPI_Recv ( buffer, BUFLen, MPI_CHAR, MPI_ANY_SOURCE, 99,
MPI_COMM_WORLD, &status) ;
    printf ("%d received '%s' \n",myid,buffer) ;fflush (stdout) ;
}
else
{
    printf ("%d receiving \n",myid) ;fflush (stdout) ;
    MPI_Recv ( buffer, BUFLen, MPI_CHAR, MPI_ANY_SOURCE, 99,
MPI_COMM_WORLD, &status) ;
    printf ("%d received '%s' \n",myid,buffer) ;fflush (stdout) ;
    MPI_Send (buffer, strlen (buffer) +1, MPI_CHAR, next, 99, MPI_COMM_WORLD) ;
    printf ("%d sent '%s' \n",myid,buffer) ;fflush (stdout) ;
}
MPI_Finalize () ;
return (0) ;
}

```

用 mpicc -o test srtest.c 编译上面的程序， mpiexec -np 2 ./test 运行得到如下结果：

```

Process 0 on hpc134
Process 0 of 2
0 sending 'hello there'
Process 1 on hpc134
Process 1 of 2
1 receiving

```

```
0 receiving
1 received 'hello there'
0 received 'hello there'
1 sent 'hello there'
```

分析上面的程序可以看到：

```
MPI_Init (&argc,&argv) ;
MPI_Comm_size (MPI_COMM_WORLD,&numprocs) ;
MPI_Comm_rank (MPI_COMM_WORLD,&myid) ;
MPI_Get_processor_name (processor_name,&namelen) ;
```

上述语句是初始化的工作。这里有一个新的函数 `MPI_Get_processor_name`，它是用来取得运行本进程的机器的名称，该名称放在字符串 `processor_name` 中，该字符串的长度为 `namelen`。宏定义 `MPI_MAX_PROCESSOR_NAME` 是机器名的最大长度。

```
printf ("Process %d on %s\n", myid, processor_name) ;
printf ("Process %d of %d\n", myid, numprocs) ;
```

这两句程序用来输出当前运行进程号 `myid`，机器名 `processor_name`，和总进程数 `numprocs`。在输出结果中可以看见 0 号进程对应的是 `Process 0 on hpc134`，`Process 0 of 2`。

```
memset (buffer, 0, BUFLen*sizeof (char)) ;
```

将 `buffer` 清空。

```
if (myid == numprocs-1)
    next = 0;
else
    next = myid+1;
```

告诉每一个进程号它们后一个进程号 `next` 是多少。最后一个进程号 `numprocs-1` 的下一个是 0。

有了上述准备，假设 `myid` 为 0，我们进入下面的 `if` 分支：

```
if (myid == 0)
{
    strcpy (buffer,"hello there") ;
    printf ("%d sending '%s'\n",myid,buffer) ;fflush (stdout) ;
    MPI_Send (buffer, strlen (buffer) +1, MPI_CHAR, next, 99, MPI_COMM_WORLD) ;
    printf ("%d receiving \n",myid) ;fflush (stdout) ;
    MPI_Recv ( buffer, BUFLen, MPI_CHAR, MPI_ANY_SOURCE, 99, MPI_COMM_WORLD,
&status) ;
    printf ("%d received '%s'\n",myid,buffer) ;fflush (stdout) ;
}
```

首先将字符串“hello there”拷贝到 `buffer` 中，然后输出 `buffer` 中的内容：0 sending 'hello there'，并用 `fflush` 刷新输出流。然后调用了这节我们要分析的关键函数 `MPI_Send`。

```
MPI_Send (buffer, strlen (buffer) +1, MPI_CHAR, next, 99, MPI_COMM_WORLD) ;
```

`MPI_Send` 函数的标准形式是：`int MPI_SEND (buf, count, datatype, dest, tag, comm)`。

其中，输入参数包括：

- buf，发送缓冲区的起始地址，可以是各种数组或结构的指针。
- count，整型，发送的数据个数，应为非负整数。
- datatype，发送数据的数据类型，这个参数将在后续节中详细介绍。
- dest，整型，目的进程号。
- tag，整型，消息标志，后续节中会做进一步介绍。
- comm，MPI 进程组所在的通信域，留待后续节进一步介绍。

该函数没有输出参数，返回错误码。

这个函数的含义是向通信域 comm 中的 dest 进程发送数据。消息数据存放在 buf 中，类型是 datatype，个数是 count 个。这个消息的标志是 tag，用以和本进程向同一目的进程发送的其它消息区别开来。

本程序中的含义就是在通讯域 MPI\_COMM\_WORLD 中，向 next 号进程（对于 0 号进程来说就是 1 号进程），发送 buffer 中的全部数据也就是 hello there，类型是 MPI\_CHAR，标签是 99。注意到我们在程序前面声明的是 char 型的 buffer，这里用的 datatype 要转化为 MPI\_CHAR。MPI\_CHAR 是 MPI 的预定义数据类型，它是和常用的 C 数据类型 char 有一一对应的关系。

接着 `printf ("%d receiving \n",myid);fflush (stdout);`，输出字符串：0 receiving。

然后是我们要介绍的第二个重要函数 MPI\_Recv。

`MPI_Recv (buffer, BUFLen, MPI_CHAR, MPI_ANY_SOURCE, 99, MPI_COMM_WORLD, &status);`

MPI\_Recv 的标准形式是：`int MPI_RECV (buf,count,datatype,source,tag,comm,status)`。

其中，输出参数包括：

- buf，接收缓冲区的起始地址，可以是各种数组或结构的指针。
- status，MPI\_Status 结构指针，返回状态信息。

输入参数有：

- count，整型，最多可接收的数据的个数。
- datatype，接收数据的数据类型。
- source，整型，接收数据的来源即发送数据进程的进程号。
- tag，整型，消息标识，应与相应的发送操作消息标识相同。
- comm，本进程（即消息接收进程）和消息发送进程所在的通信域。

函数返回错误码。

本程序语句的意思是 0 号进程从 MPI\_COMM\_WORLD 域中任意进程（MPI\_ANY\_SOURCE 表示接收任意源进程发来的消息）接收标签号为 99，而且不超过 512（前面定义了 `#define BUFLen 512`）个 MPI\_CHAR 类型的数据，保存到 buffer 中。



### 脚下留心：

接收缓冲区 buf 的大小不能小于发送过来的有效消息长度。否则可能由于数组越界产生导致程序错误。

MPI\_Recv 绝大多数的参数和 MPI\_Send 相对应，有相同的意义，很好理解。唯一的区别就是 MPI\_Recv 里面多了一个参数 status。MPI\_Status 是 MPI 中一个特殊的，也是比较有用的结构。MPI\_Status 的结构定义在 mpi.h 当中可以找到。

```
/* The order of these elements must match that in mpif.h */
```

```
typedef struct MPI_Status {
    int count;
    int cancelled;
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status;
```

status 主要显示接收函数的各种错误状态。我们通过访问 status.MPI\_SOURCE、status.MPI\_TAG 和 status.MPI\_ERROR 就可以得到发送数据进程号、发送数据使用的 tag 以及本接收操作返回的错误代码。另外，更为有用的是实际接收到的数据项数可以由 MPI 函数 MPI\_Get\_count 获得。



### 脚下留心：

这里我们并没有简单使用 status.count 来获取数据项数。

MPI\_Get\_count 的标准定义为：

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

其中，前两个参数为输入参数，status 是 MPI\_Recv 返回的状态结构的指针，datatype 指定数据类型；最后一个参数是输出参数，是实际接收到的给定数据类型的数据项数。与大多数 MPI 函数一样，这个函数返回错误码。

大家可以尝试将下面的语句替换进程 0 的条件块，观察一下输出，想想为什么？

```
if (myid == 0)
{
    strcpy (buffer,"hello there");
    printf ("%d sending '%s'\n",myid,buffer);fflush (stdout);
    MPI_Send (buffer, strlen (buffer) +1, MPI_CHAR, next, 99, MPI_COMM_WORLD);
    printf ("%d receiving \n",myid);fflush (stdout);
    MPI_Recv ( buffer, BUFLen, MPI_CHAR, MPI_ANY_SOURCE, 99, MPI_COMM_WORLD,
    &status);
    printf ("%d received '%s'\n",myid,buffer);fflush (stdout);
    {int num=0;
    printf ("buffer has %d length\n",strlen (buffer));
    MPI_Get_count (&status, MPI_CHAR, &num);
    printf ("I get %d of CHAR\n",num);
```

```

    MPI_Get_count (&status, MPI_INT, &num);
    printf ("I get %d of INT\n",num);
    num=-1;
    MPI_Get_count (&status, MPI_DOUBLE, &num);
    printf ("I get %d of INT\n",num);
    printf ("%d in status.count\n",status.count);
}
}

```

例程后面的语句与前面类似，这里就不再赘述了。大家可以使用更多进程数运行这个程序，比如：`mpiexec -np 4 ./test`，来看看会发生什么？



### 指点迷津：

如果读者仔细分析点对点通信的例子，相信不难发现这是一个典型的流式

MPMD 的 MPI 并行程序。进程  $i$  等待进程  $i-1$  传递过来的字符串“hello there”，并将

其传递给进程  $i+1$ ，最后一个进程则传递字符串给进程 0。

## 消息管理 7 要素

MPI 最重要的功能莫过于消息传递。正如我们先前看到一样，`MPI\_Send` 和 `MPI\_Recv` 负责在两个进程间发送和接收消息。总结起来，点对点消息通信的参数主要是由以下 7 个参数组成：

- (1) 发送或者接收缓冲区 `buf`;
- (2) 数据数量 `count`;
- (3) 数据类型 `datatype`;
- (4) 目标进程或者源进程 `destination/source`;
- (5) 消息标签 `tag`;
- (6) 通信域 `comm`;
- (7) 消息状态 `status`，只在接收的函数中出现。

MPI 程序中的消息传递可以很形象地比喻成我们日常的邮件发送和接收。自然地，`buf`, `count`, `datatype` 可以被称作是信件的内容，而 `source/destination`, `tag`, `comm` 则好比是邮件的信封，因此我们称之为消息信封。这里我们对其中四个参数做进一步的探讨。

### 1. 消息数据类型

在消息缓冲的三个变量中，最值得注意的是 `datatype`，消息数据类型。

为什么需要定义消息数据类型？主要的理由有两个：一是支持异构平台计算的互操作性，二是允许方便地将非连续内存区中的数据，具有不同数据类型的内容组成消息。MPI 程序有严格的数据类型匹配要求。类型匹配包涵了两个层面的内容：一是宿主语言的类型（C 或者 Fortran 数据类型）和通信操作所指定的类型相匹配；二是发送方和接收方的类型匹配。MPI 用预定义的基本数据类型和导出数据类型来满足上述要求。

### (1) 基本数据类型

如前所述，我们需要发送和接收连续的数据，MPI 提供了预定义的数据类型供程序员使用。

表 7.3.1 MPI 预定义数据类型与 C 数据类型的对应关系

| MPI 预定义数据类型        | 相应的 C 数据类型         |
|--------------------|--------------------|
| 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       | unsigned int       |
| MPI_UNSIGNED_LONG  | unsigned long int  |
| MPI_FLOAT          | float              |
| MPI_DOUBLE         | double             |
| MPI_LONG_DOUBLE    | long double        |
| MPI_BYTE           | 无对应类型              |
| MPI_PACKED         | 无对应类型              |

表 7.3.2 MPI 预定义数据类型与 FORTRAN77 数据类型的对应关系

| MPI 预定义数据类型          | 相应的 FORTRAN77 数据类型 |
|----------------------|--------------------|
| MPI_INTEGER          | INTEGER            |
| MPI_REAL             | REAL               |
| MPI_DOUBLE_PRECISION | DOUBLE PRECISION   |
| MPI_COMPLEX          | COMPLEX            |
| MPI_LOGICAL          | LOGICAL            |
| MPI_CHARACTER        | CHARACTER          |
| MPI_BYTE             | 无对应类型              |
| MPI_PACKED           | 无对应类型              |

对于初学者来说，应尽可能保证发送和接收的数据类型完全一致。

MPI\_BYTE 与 MPI\_PACKED 两个数据类型并没有具体的 C 或者 Fortran 类型与其对应。实际上，他们可以与任何以字节为单位的消息相匹配。MPI\_BYTE 是将消息内容不加修改地通过二进制字节流来传递的一种方法。而 MPI\_PACKED 是为了将非连续的数据进行打包发送而提出的。经常与函数 MPI\_Pack\_size 和 MPI\_Pack 联合使用。下面给出了一个使用 MPI\_PACKED 数据类型的一个简单例子。

```
double A[100];
MPI_Pack_size (50,MPI_DOUBLE,comm,&BufferSize) ;
TempBuffer = malloc (BufferSize) ;
j = sizeof (MPI_DOUBLE) ;
Position = 0;
for (i=0;i<50;i++)
```



```

MPI_Pack (A+i*2,1,MPI_DOUBLE,TempBuffer,BufferSize,&Position,comm) ;
MPI_Send (TempBuffer,Position,MPI_PACKED,destination,tag,comm) ;

```

表 7.3.3 MPI\_PACKED 例程的说明

| 例 程  | 说 明   |
|--|---|
| MPI_Pack_size  | 决定需要一个多大的缓冲区来存放 50 个 MPI_DOUBLE 数据项。<br>函数参数除 comm（通信域）外全是整型。   |
| malloc (BufferSize)  | 为缓冲区 TempBuffer 分配内存  |
| for 循环   | 将数组 A 的 50 个偶序数元素打成一个包，放在 TempBuffer 中  |
| MPI_Pack<br>(A+i*2, 1, MPI_DOUBLE,<br>TempBuffer,BufferSize,<br>&Position, comm) ; | 第一个参数是被打包的数组元素的地址<br>第三个参数是被打包的数组元素的数据类型<br>Position，整型，用于跟踪已经有多少个数据项被打包；而 position 的最后值在接下来的 MPI_Send 中被用作消息计数 |

## (2) 导出数据类型

除了这些基本数据类型之外，MPI 还允许通过导出数据类型，将不连续的，甚至是不同类型的数据元素组合在一起形成新的数据类型。我们称这种由用户定义的数据类型为导出数据类型。这种数据类型需要用户使用 MPI 提供的构造函数来构造。限于篇幅，本章不再对其做进一步介绍。

归纳起来类型匹配规则可以概括为：

- 有类型数据的通信，发送方和接收方均使用相同的数据类型；
- 无类型数据的通信，发送方和接收方均以 MPI\_BYTE 作为数据类型；
- 打包数据的通信，发送方和接收方均使用 MPI\_PACKED。

## 2. 消息标签 TAG

TAG 是消息信封中的一项，是程序在同一接收者的情况下，用于标识不同类型消息的一个整数。我们来看下面这个例子，就可以明白 TAG 的重要性了。

未使用标签：

```

Process P: send (A,32,Q) ; send (B,16,Q) ;
Process Q: recv (X, 32, P) ; recv (Y, 16, P) ;

```

使用了标签：

```

Process P: send (A,32,Q,tag1) ; send (B,16,Q,tag2) ;
Process Q: recv (X, 32, P, tag1) ; recv (Y, 16, P, tag2)

```

这段代码打算传送 A 的前 32 个字节进入 X，传送 B 的前 16 个字节进入 Y。但是如果消息 B 尽管后发送但先到达进程 Q，就会被第一个 recv 接收在 X 中。使用标签 TAG 则可以有效避免这个错误。

## 3. 通信域

消息的发送和接收方必须使用相同的消息标签才能实施通信。而在具有复杂通信模式的 MPI 程序当中，特别是将一个并行程序模块插入到另一个并行程序当中时，维护 TAG 来匹配消息可能是十分烦琐的事情。为此，MPI 中提出了消息信封的另一项：通信域。

一个通信域（comm）包含一个进程组（process group）及其上下文（context）。进程组是进程的有限有序集。有限意味着，在一个进程组中，进程的个数 n 是有限的。这里的 n 称为进程组的大小（group size）。有序意味着 n 个进程是按整数 0, 1, ..., n-1 进行编号的。

通信域限定了消息传递的进程范围。

一个进程在一个通信子（组）中用它的编号进行标识。组的大小和进程号可以通过调用 MPI 例程 MPI\_Comm\_size 和 int MPI\_Comm\_rank 获得。

MPI 实现已经预先定义了两个进程组：MPI\_COMM\_SELF，只包含各个进程自己的进程组；MPI\_COMM\_WORLD，包含本次启动的所有 MPI 进程的进程组。同时，MPI 还为通信子提供了各种管理函数，其中包括：

- (1) 通信域比较 int MPI\_Comm\_compare (comm1, comm2, result)：如 comm1, comm2 为相同句柄，则 result 为 MPI\_Ident；如果仅仅是各进程组的成员和序列号都相同，则 result 为 MPI\_Congruent；如果二者的组成员相同但序号不同则结果为 MPI\_Similar；否则，结果为 MPI\_Unequal。
- (2) 通信域拷贝 int MPI\_Comm\_dup (comm, newcomm)：对 comm 进行复制得到新的通信域 newcomm。
- (3) 通信域分裂 int MPI\_Comm\_split (comm, color, key, newcomm)：本函数要求 comm 进程组中的每个进程都要执行，每个进程指定一个 color（整型），此调用首先将具有相同 color 值的进程形成一个新的进程组，新产生的通信域与这些进程组一一对应。新通信域中各个进程的顺序编号根据 key（整型）的大小决定，即 key 越小，则相应进程在新通信域中的顺序编号也越小，若一个进程中的 key 相同，则根据这两个进程在原来通信域中顺序号决定新的进程号。一个进程可能提供 color 值为 MPI\_Undefined，此种情况下，其 newcomm 返回 MPI\_COMM\_NULL。
- (4) 通信域销毁 int MPI\_Comm\_free (comm)：释放给定通信域。

上述函数都返回错误码。

#### 4. 状态字（status）

前面已经讨论过，状态字的主要功能是保存接收到的消息的状态。再来看一个简单的例子。

```
while (true) {  
    MPI_Recv (received_request, 100, MPI_BYTE, MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &Status);  
    switch (Status.MPI_TAG) {  
        case tag_0: perform service type0;  
        case tag_1: perform service type1;  
        case tag_2: perform service type2;  
    }  
}
```

这个例子中，MPI\_Recv 并没有指明具体接收哪里发来的消息，它可以接收任意源（MPI\_ANY\_SOURCE），任意标签的消息（MPI\_ANY\_TAG）。我们通过检查 Status 中的 MPI\_TAG 可以有效把消息区分开来。当一个接收者能从不同进程接收不同大小和标签的信息时，比如服务器进程，查阅状态信息就会很有用。



**指点迷津：牢记通信匹配**

MPI 最主要的功能是消息通信。

为了保证程序的性能和正确性，我们在编写 MPI 程序时首先应关注上述介绍的

消息管理要素的匹配关系。具体的说,在相互通信的进程中,通信数据类型应匹配;消息标签、通信域应完全相同;发送进程号与接收进程号应一一对应;接收消息的缓冲区应不小于发送过来的消息的大小;在数据类型相同的条件下,接收数据数量应不小于发送数据数量。

当然光关注消息要素匹配,也不能保证 MPI 程序正确运行。接下来我们对上面介绍的 MPI 程序例子作进一步讨论。正如前面分析的一样,0 号进程和非 0 号进程走的分支不一样。如果大家同等对待,都是先收后发行吗?为了验证这个想法,我们首先去掉 else 那个分支,使所有的进程都走进程 0 的执行语句,且都是先收后发。

这样,四个进程的结果如下:

```
Process 0 on hpc134
Process 0 of 4
0 receiving
Process 1 on hpc134
Process 1 of 4
1 receiving
Process 2 on hpc134
Process 2 of 4
2 receiving
Process 3 on hpc134
Process 3 of 4
3 receiving
```

可以看到,程序进入了停滞状态。之所以先收后发出现问题是因为所有的进程都在收到消息之后才能继续后面操作,也就是说在没有收到正确消息之前他们不会进行后面的发送。我们看见 0, 1, 2, 3 都是在 receiving 的状态,而这时候没有进程发送消息,并且所有的进程都在等待其他进程发送的消息。这种“大家都在等待”的状态称作“死锁”。死锁现象是多进程、多线程编程中经常发生的现象。死锁出现的原因在于,从程序员的角度看,当 MPI\_Send 或者 MPI\_Recv 正确返回,其结果是该调用要求的通信操作已正确完成,即消息已成功发出或成功接收;该调用的缓

缓冲区可用，若是发送操作则该缓冲区可以被其它的操作更新，若是接收操作该缓冲区中的数据已经可以被完整地使用。我们称这样的消息发送或者接收为阻塞通信，这个例子中的标准的 MPI\_Recv 和 MPI\_Send 就属于阻塞操作。

基于此，除了要求发送接收消息的消息信封匹配外，阻塞通信中点对点消息顺序的匹配也对正确通信有着至关重要的影响。

进一步，大家思考一下每个进程都是先发后收又会是怎样？为什么会这样哪？

### 3.4 统计时间

编写并行程序的目的是提高程序运行性能。为了检验并行化的效果，我们经常会用到统计时间的函数。在此，MPI 提供了两个时间函数 MPI\_Wtime 和 MPI\_Wtick。其中，MPI\_Wtime 函数返回一个双精度数，标识从过去的某点的时间到当前时间所消耗的时间秒数。而函数 MPI\_Wtick 则返回 MPI\_Wtime 结果的精度。

我们把前面最初的 srtest.c 稍作修改。

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

#define BUFLLEN 512

int main (int argc, char *argv[])
{
    int myid, numprocs, next, namelen;
    char buffer[BUFLLEN], processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;
    double t1,t2,t3,tick;

    MPI_Init (&argc,&argv) ;
    MPI_Comm_size (MPI_COMM_WORLD,&numprocs) ;
    MPI_Comm_rank (MPI_COMM_WORLD,&myid) ;
    MPI_Get_processor_name (processor_name,&namelen) ;

    t1=MPI_Wtime () ;

    printf ("Process %d on %s\n", myid, processor_name) ;
    printf ("Process %d of %d\n", myid, numprocs) ;
```

```

memset (buffer, 0, BUFLen*sizeof (char)) ;
if (myid == numprocs-1)
    next = 0;
else
    next = myid+1;

if (myid == 0)
{
    strcpy (buffer,"hello there") ;
    printf ("%d sending '%s' \n",myid,buffer) ;fflush (stdout) ;
    MPI_Send (buffer, strlen (buffer) +1, MPI_CHAR, next, 99, MPI_COMM_WORLD) ;
    printf ("%d receiving \n",myid) ;fflush (stdout) ;
    MPI_Recv ( buffer, BUFLen, MPI_CHAR, MPI_ANY_SOURCE, 99,
MPI_COMM_WORLD, &status) ;
    printf ("%d received '%s' \n",myid,buffer) ;fflush (stdout) ;
}
else
{
    printf ("%d receiving \n",myid) ;fflush (stdout) ;
    MPI_Recv ( buffer, BUFLen, MPI_CHAR, MPI_ANY_SOURCE, 99,
MPI_COMM_WORLD, &status) ;
    printf ("%d received '%s' \n",myid,buffer) ;fflush (stdout) ;
    MPI_Send (buffer, strlen (buffer) +1, MPI_CHAR, next, 99, MPI_COMM_WORLD) ;
    printf ("%d sent '%s' \n",myid,buffer) ;fflush (stdout) ;
}
t2=MPI_Wtime () ;
t3=t2-t1;
tick = MPI_Wtick () ;
printf ("%d process time is '%f' \n", myid,t3) ;
printf ("%d process tick is '%f' \n", myid,tick) ;
MPI_Finalize () ;
return (0) ;
}

```

两个进程的运行结果是：

```

Process 0 on hpc134
Process 0 of 2
0 sending 'hello there'
Process 1 on hpc134
Process 1 of 2
1 receiving
0 receiving
1 received 'hello there'
1 sent 'hello there'

```

```
0 received 'hello there'
1 process time is '0.000699'
1 process tick is '0.000004'
0 process time is '0.000870'
0 process tick is '0.000004'
```

从上述的程序中，我们可以看出使用 `MPI_WTime` 将计时点 `t2` 和 `t1` 之间的时间统计出来（进程 0 运行时间 0.000870s，进程 1 运行时间 0.000699s），同时使用 `MPI_Wtick` 可以得到计时的精度（计时精度为 0.000004s）。

## 3.5 错误管理

MPI 在错误管理方面提供了丰富的接口函数，这里我们介绍其中最简单的部分接口。

- 用 `status.MPI_ERROR` 来获取错误码。
- MPI 终止 MPI 程序执行的函数 `MPI_Abort`。

`int MPI_Abort (MPI_Comm comm, int errorcode)`

它使 `comm` 通信域的所有进程退出，返回 `errorcode` 给调用的环境。通信域 `comm` 中的任一进程调用此函数都能够使该通信域内所有的进程结束运行。

## 3.6 小结

本节通过两个实例（`hellow.c` 和 `srtest.c`），向大家介绍了 MPI 编程的初步知识。其中核心部分是 MPI 的点对点通信，我们讲解了 `MPI_Send` 和 `MPI_Recv` 的用法。

对于点对点通信，需要强调的是正确的消息匹配。这包括调用顺序和消息要素的匹配。匹配是正确实施通信的关键。

在 linux 平台上，按照第 7.2 节正确设置了 `MANPATH` 环境变量后，大家就可以同 `man` 手册页方便地查询各种 MPI 接口。例如，可以用 `man MPI_Send` 或者 `man MPI_Recv` 来查看函数的功能、输入输出参数和返回值，以及 Fortran 语言接口信息等等。

## 第 4 节 MPI 群集通信

除了前面介绍的点到点通信之外，MPI 还提供了群集通信。所谓群集通讯，包含了一对多，多对一和多对多的进程通信模式。它的最大的特点就是多个进程参与通信，如果说点到点就像生活中的打电话或单独交谈，那么群集通信更像是小组讨论，广播等多人参与的交流活动。下面我们将要介绍在 MPI 中常用的几个群集通信函数。

### 4.1 同步

本函数接口是：`int MPI_Barrier (MPI_Comm comm)`。

这个函数像一道路障。在操作中，通信子 `comm` 中的所有进程相互同步，即它们相互等待，直到所有进程都执行了他们各自的 `MPI_Barrier` 函数，然后再各自接着开始执行后续的

代码。同步函数是并程序中控制执行顺序的有效手段。

## 4.2 广播

广播顾名思义，就是一对多的传送消息。它的作用是从一个 root 进程向组内所有其他的进程发送一条消息。它的接口形式是：

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

图 7.4.1 给出了广播操作的示意。

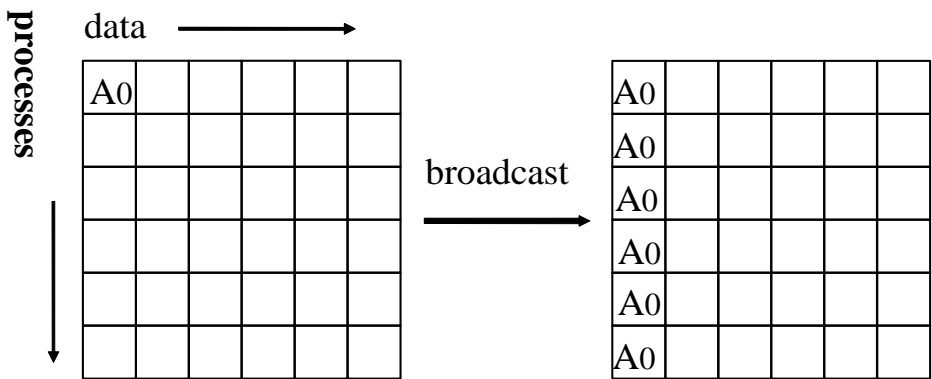


图 7.4.1 广播操作示意图

## 4.3 聚集

聚集函数 MPI\_Gather 是一个多对一的通信函数。其接口为：

```
int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm )
```

root 进程接收该通信组每一个成员进程（包括 root 自己）发送的消息。这  $n$  个消息的连接按进程号排列存放在 root 进程的接收缓冲中。每个发送缓冲由三元组（sendbuf, sendcnt, sendtype）标识。所有非 root 进程忽略接收缓冲，对 root 进程发送缓冲由三元组（recvbuf, recvcnt, recvtype）标识。图 7.4.2 给出聚集操作的示意。

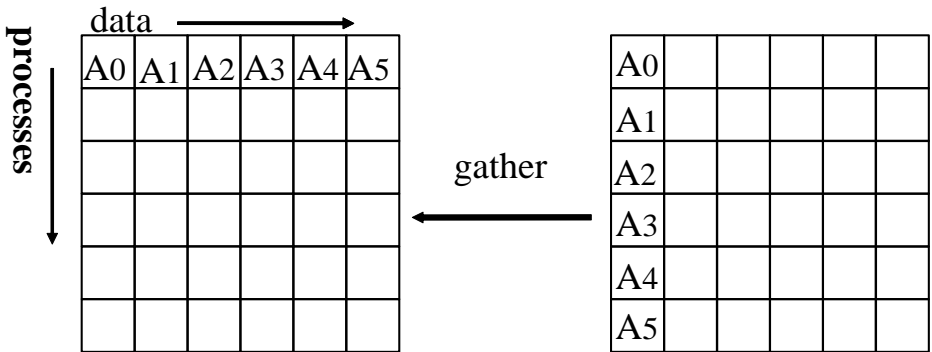


图 7.4.2 聚集操作示意图

## 4.4 播撒

```
int MPI_Scatter (void *sendbuf, int sendcnt, MPI_Datatype sendtype,  
                void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root,  
                MPI_Comm comm)
```

`MPI_Scatter` 是一对多的传递消息。但是它和广播不同，`root` 进程向各个进程传递的消息是可以不同的。`Scatter` 实际上执行的是与 `Gather` 相反的操作。

## 4.5 扩展的聚集和播撒操作

`MPI_Allgather` 的作用是每一个进程都收集到其他所有进程的消息，它相当于每一个进程都执行了 `MPI_Gather` 执行完了 `MPI_Gather` 之后，所有的进程的接收缓冲区的内容都是相同的，也就是说每个进程给所有进程都发送了一个相同的消息，所以名为 `allgather`。本函数的接口是：

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

图 7.4.3 给出了扩展的聚集和播撒操作的示意。

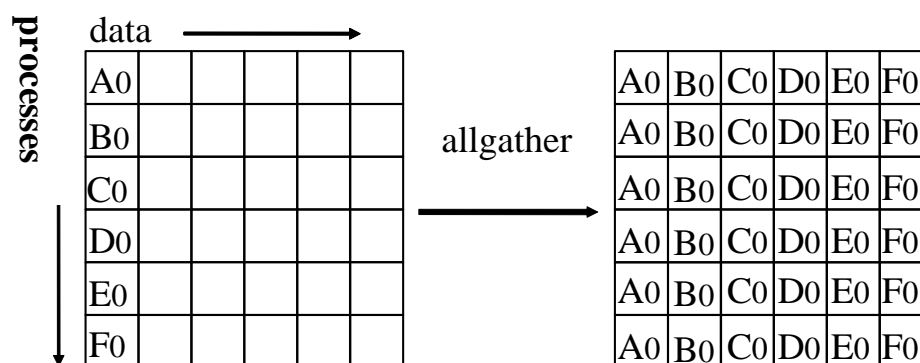


图 7.4.3 扩展的聚集和播撒操作示意图

## 4.6 全局交换

`MPI_Allgather` 每个进程发一个相同的消息给所有的进程，而 `MPI_Alltoall` 散发给不同进程的消息是不同的。因此，它的发送缓冲区也是一个数组。`MPI_Alltoall` 的每个进程可以向每个接收者发送数目不同的数据，第 `i` 个进程发送的第 `j` 块数据将被第 `j` 个进程接收并存放在其接收消息缓冲区 `recvbuf` 的第 `i` 块，每个进程的 `sendcount` 和 `sendtype` 的类型必须和所有其他进程的 `recvcount` 和 `recvtype` 相同，这也意味着在每个进程和根进程之间发送的数据量必须和接收的数据量相等。函数接口为：

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



全局交换的操作示意图为图 7.4.4。

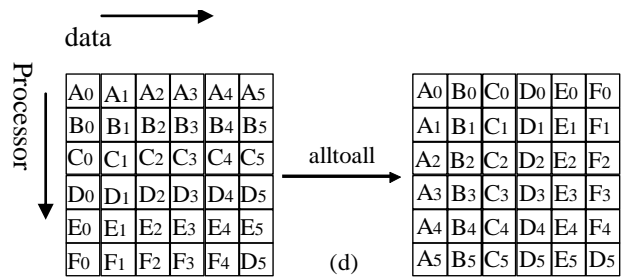


图 7.4.4 全局交换操作示意图

## 4.7 规约与扫描

MPI 提供了两种类型的聚合操作：归约（reduction）和扫描（scan）。

### 1. 归约

int MPI\_Reduce (void \*sendbuf, void \*recvbuf, int count, MPI\_Datatype datatype,  
MPI\_Op op, int root, MPI\_Comm comm)

这里每个进程的待处理数据存放在 sendbuf 中，可以是标量也可以是向量。所有进程将这些值通过输入的操作子 op 计算为最终结果并将它存入 root 进程的 recvbuf。数据项的数据类型在 Datatype 域中定义。具体的归约操作包括：

- MPI\_MAX 求最大值
- MPI\_MIN 求最小值
- 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 逻辑异或
- MPI\_BXOR 按位异或
- MPI\_MAXLOC 最大值且相应位置
- MPI\_MINLOC 最小值且相应位置

规约操作的数据类型组合如表 7.4.1 所示。

表 7.4.1 规约操作与相应类型的对应关系

| 操作                        | 允许的数据类型                   |
|---------------------------|---------------------------|
| MPI_MAX,MPI_MIN           | C 整数, Fortran 整数, 浮点数     |
| MPI_SUM,MPI_PROD          | C 整数, Fortran 整数, 浮点数, 复数 |
| MPI_LAND,MPI_LOR,MPI_XLOR | C 整数, 逻辑型                 |
| MPI_BAND,MPI_BOR,MPI_BXOR | C 整数, Fortran 整数, 字节型     |

在 MPI 中，针对规约操作，所有的 MPI 预定义的操作都是可结合的，也是可交换的。同时，用户可以指定自定义的函数操作，这些操作是也要可结合的，但可以不是可交换的。

### 2. 扫描

int MPI\_Scan (void \*sendbuf, void \*recvbuf, int count, MPI\_Datatype datatype,

MPI\_Op op, MPI\_Comm comm)

MPI\_Scan 常用于对分布于组中的数据作前置归约操作。此操作将序列号为 0, ...,  $i$  (包括  $i$ ) 的进程发送缓冲区的归约结果存入序列号为  $i$  的进程接收消息缓冲区中。这种操作支持的数据类型、操作以及对发送及接收缓冲区的限制和规约相同。与规约相比, 扫描 Scan 操作省去了 Root 域, 因为扫描是将部分值组合成  $n$  个最终值, 并存放在  $n$  个进程的 recvbuf 中。具体的扫描操作由 Op 域定义。

MPI 的归约和扫描操作允许每个进程贡献向量值, 而不只是标量值。向量的长度由 Count 定义。MPI 也支持用户自定义的归约操作。

## 4.8 简单示例

前面第 7.1 节曾经分析过 MPI 程序的特点, MPI 程序可以是对等模式的 SPMD 程序也可以是主从、联合以及流式模式的 MPMD 程序。其中, SPMD 是最简单、最常用的 MPI 程序形式, 第 7.1 节已经给出了一个清楚的例子。本节针对 MPMD 程序, 介绍一个主从模式的例子给大家: 积分法求解  $\pi$  的例子 cpi.c。在这个例子中应用了两个 MPI 群集函数: 广播和规约。

```
/* Case 3 cpi.c */
#include "mpi.h"
#include <stdio.h>
#include <math.h>

double f (double) ;

double f (double a)
{
    return (4.0 / (1.0 + a*a)) ;
}

int main (int argc, char *argv[])
{
    int    n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    double startwtime = 0.0, endwtime;
    int    namelen;
    char   processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init (&argc, &argv) ;
    MPI_Comm_size (MPI_COMM_WORLD, &numprocs) ;
    MPI_Comm_rank (MPI_COMM_WORLD, &myid) ;
    MPI_Get_processor_name (processor_name, &namelen) ;

    fprintf (stdout, "Process %d of %d is on %s\n",
```

```

        myid, numprocs, processor_name) ;
fflush (stdout) ;

n = 10000;                /* default # of rectangles */
if (myid == 0)
    startwtime = MPI_Wtime () ;

MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD) ;

h = 1.0 / ((double) n);
sum = 0.0;
/* A slightly better approach starts from large i and works back */
for (i = myid + 1; i <= n; i += numprocs)
{
    x = h * ((double) i - 0.5) ;
    sum += f (x) ;
}
mypi = h * sum;

MPI_Reduce (&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD) ;

if (myid == 0) {
    endwtime = MPI_Wtime () ;
    printf ("pi is approximately %.16f, Error is %.16f\n",
            pi, fabs (pi - PI25DT)) ;
    printf ("wall clock time = %f\n", endwtime-startwtime) ;
    fflush (stdout) ;
}

MPI_Finalize () ;
return 0;
}

```

进程 0 是主进程，其他的都是从进程。程序首先初始化，输出一些消息和步长  $n$ 。

```

MPI_Init (&argc,&argv) ;
MPI_Comm_size (MPI_COMM_WORLD,&numprocs) ;
MPI_Comm_rank (MPI_COMM_WORLD,&myid) ;
MPI_Get_processor_name (processor_name,&namelen) ;

```

```

fprintf (stdout,"Process %d of %d is on %s\n",
        myid, numprocs, processor_name) ;
fflush (stdout) ;
n=10000;

```

然后由零号进程负责计时。

```

if (myid == 0)
    startwtime = MPI_Wtime ();

```

主进程给各个从进程派发任务：n，整个计算的上界 n。

```

MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

```

```

h= 1.0 / (double) n;
sum = 0.0;
/* A slightly better approach starts from large i and works back */
for (i = myid + 1; i <= n; i += numprocs)
{
    x = h * ((double) i - 0.5);
    sum += f(x);
}
mypi = h * sum;

```

上面各个从进程计算自己部分的 mypi。为了充分利用系统资源，让各个进程任务尽量均匀，0 号进程也是从进程的一份子，自己也参与计算。其中，语句 for (i = myid + 1; i <= n; i += numprocs) 中 i 每次增加总进程的个数。这样，每个从进程计算的次数大体相当。计算完成后各个从进程的 mypi 被传回 0 号进程。

```

if (myid == 0) {
    endwtime = MPI_Wtime ();
    printf ("pi is approximately %.16f, Error is %.16f\n",
           pi, fabs (pi - PI25DT));
    printf ("wall clock time = %f\n", endwtime-startwtime);
    fflush (stdout);
}

```

```

MPI_Finalize ();
return 0;

```

最后这一部分是由主进程计算并输出执行时间，和计算得出的  $\pi$  与我们预设标准  $\pi$  的误差，结束 MPI。让我们进一步看一下运行结果：

```

[test@hpc134 test]$ mpiexec -np 1 ./a.out
Process 0 of 1 is on hpc134
pi is approximately 3.1415926544231341, Error is 0.0000000008333410
wall clock time = 0.000342

```

```

[test@hpc134 test]$ mpiexec -np 2 ./a.out
Process 0 of 2 is on hpc134
Process 1 of 2 is on hpc134
pi is approximately 3.1415926544231318, Error is 0.0000000008333387
wall clock time = 0.000973

```

```

[test@hpc134 test]$ mpiexec -np 4 ./a.out
Process 0 of 4 is on hpc134

```

Process 1 of 4 is on hpc134  
Process 2 of 4 is on hpc134  
Process 3 of 4 is on hpc134  
pi is approximately 3.1415926544231239, Error is 0.0000000008333307  
wall clock time = 0.002232

## 4.9 小结

本节对 MPI-1 的主要群集通信函数作了简要介绍。群集通信共同的特点如下。了解这些群集通信的特点，为正确使用群集通信打下了基础。

- 通信子中的所有进程必须调用群集通信例程。如果代码中只有通信子中的一部分成员调用了群集例程而其它没有调用，则是错误的。这个错误代码的行为是不确定的，意味着它可能发生任何事情，包括死锁或产生错误的结果。
  - 一个进程一旦结束了它所参与的群集操作就从群集例程中返回。
  - 除了 MPI\_Barrier 以外，每个群集例程使用类似于点对点通信中的标准（standard）和阻塞的通信模式。例如，当 Root 进程从 MPI\_Bcast 中返回时，它就意味着发送缓冲可以被安全地再次使用，这是其它进程可能还没有启动它们相应的 MPI\_Bcast。
  - 一个群集例程是否同步取决于 MPI 的具体实现。MPI 要求用户负责保证他的代码无论实现是否是同步的都应正确。
  - count 和 datatype 在所包含的所有进程中必须是一致的。
  - 在群集例程中没有 tag 参数。消息信封由通信子参数和源/目的进程定义。例如，在 MPI\_Bcast 中，消息的源是 root，目的是所有进程（包括 Root）。
  - 在 MPI-1 中，只支持阻塞和通信子内（intra-communicator）群集通信。
- 了解这些群集通信的特点，为正确使用群集通信打下了基础。