**MPI程序设计入门教程**

版本历史：

2019-7-28 朱良君 第二版，基于微软的MPI实现[MS-MPI](https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi)，修复Bug

2011-5-17 吴辉 第一版，基于[MPICH2](http://www.mpich.org/downloads/)的MPI实现

前言——MPI简介

## 什么是MPI？

MPI（Message Passing Interface）是消息传递并行程序设计的标准之一，其中1.3版本于2008年5月发布，2.2版本于2009年9月发布，3.1版本于2015年6月发布，最新版的4.0规范也正在开发中。MPI正成为并行程序设计事实上的工业标准。

对于MPI的定义概括起来有三个方面：

第一，**MPI是一个库，而不是一门语言**。许多人认为MPI就是一种并行语言，这是不准确的。但是按照并行语言的分类，可以把FORTRAN+MPI或C+MPI 看作是一种在原来串行语言基础之上扩展后得到的并行语言。MPI库可以被FORTRAN77/C/Fortran90/C++调用，从语法上说，它遵守所有对库函数/过程的调用规则，和一般的函数/过程没有什么区别。

第二，**MPI是一种标准或规范的代表**，而不是特指某一个对它的具体实现。迄今为止，所有的并行计算机制造商都提供对MPI的支持，可以在网上免费得到MPI在不同并行计算机上的实现，一个正确的MPI程序，可以不加修改地在所有的并行机上运行。

第三，**MPI是一种消息传递编程模型**，并成为这种编程模型的代表和事实上的标准。MPI虽然很庞大，但是它的最终目的是服务于进程间通信这一目标的。

MPI的一个最重要的特点就是免费和源代码开放，MPI可以被迅速接受和它为自己定下的高效率、方便移植和功能强大三个主要目标密不可分。采用广为使用的语言FORTRAN和C/C++进行绑定，也是它成功的一个重要因素，当然，MPI的成功，还因为它总结和吸收了前期大量消息传递系统的经验，一个成功的标准是需要大量的实践和艰苦的努力的，MPI就是这种实践和努力的结果。

## 目前MPI的主要实现

MPICH是一种最重要的MPI实现之一，它可以免费从**http://www.mpich.org**下载。主要研制单位是Argonne国家实验室和MSU。

此外，常见的MPI实现还有[Open-MPI](https://www.open-mpi.org/)、[Intel MPI](https://software.intel.com/en-us/mpi-library)、[IBM Spectrum MPI](https://www.ibm.com/us-en/marketplace/spectrum-mpi)以及微软的[MS-MPI](https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi)等。

本教程以MS-MPI为例，开发环境为Microsoft Visual Studio 2010 sp1。

第一节 VS2010下配置MPI开发环境

## MS-MPI的安装

从微软官网下载[MS-MPI v6](https://www.microsoft.com/en-us/download/details.aspx?id=47259)或更新版本（VS2010最高支持v8版本），以默认路径安装msmpisdk.msi和MSMpiSetup.exe。

安装完成后查看系统环境变量是否已自动设置（右键我的电脑->属性->高级系统设置->环境变量->系统变量）：

MSMPI\_BIN=C:\Program Files\Microsoft MPI\Bin\

MSMPI\_INC=C:\Program Files (x86)\Microsoft SDKs\MPI\Include\

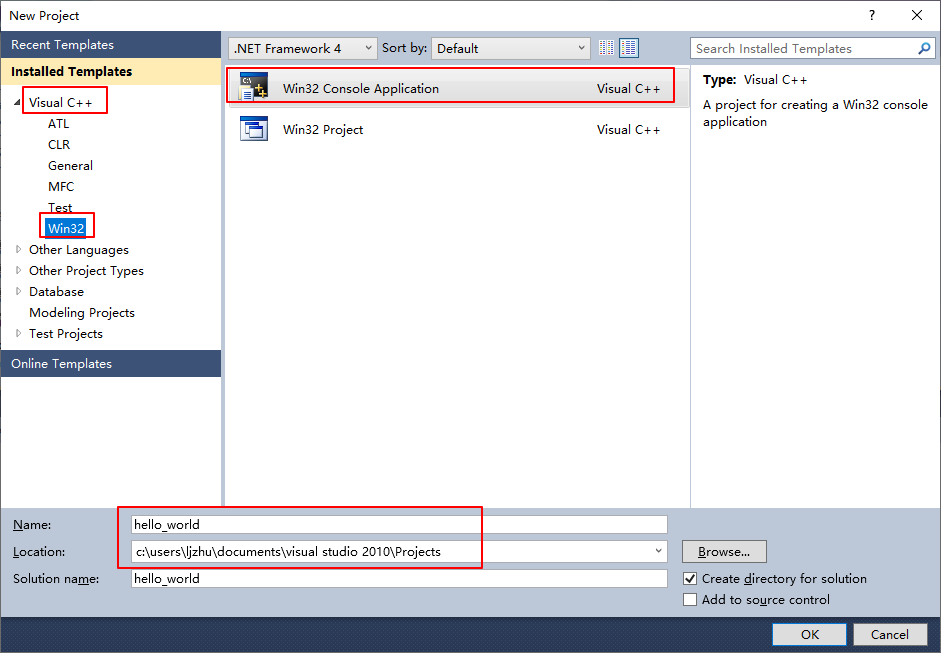
MSMPI\_LIB32=C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x86\

MSMPI\_LIB64=C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64\

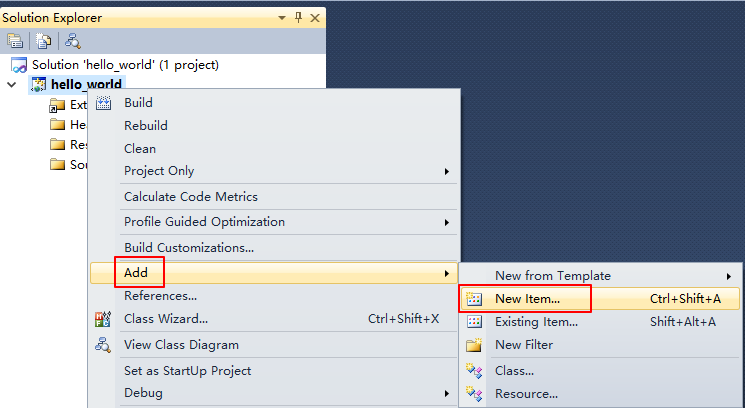
## Hello World

按照惯例，我们以一个Hello World程序为例讲解VS2010下MPI开发环境的配置。

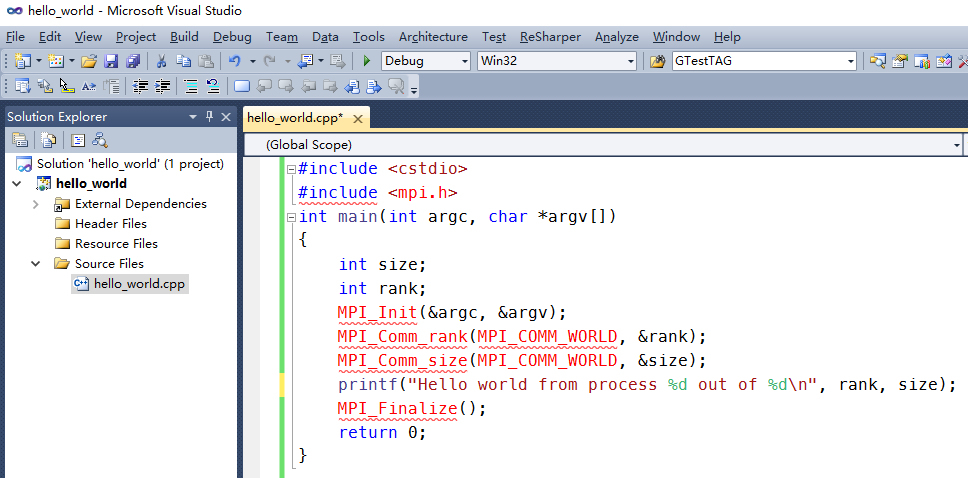
1. 打开VS2010中，点击“新建项目”新建一个Visual C++的Win32控制台应用，设置保存路径和项目名，并在下一步中选择“空项目”；



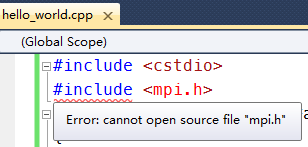
1. 右键“hello\_world”工程，选择“Add”->“New Item”，添加一个新的hello\_world.cpp源码文件；



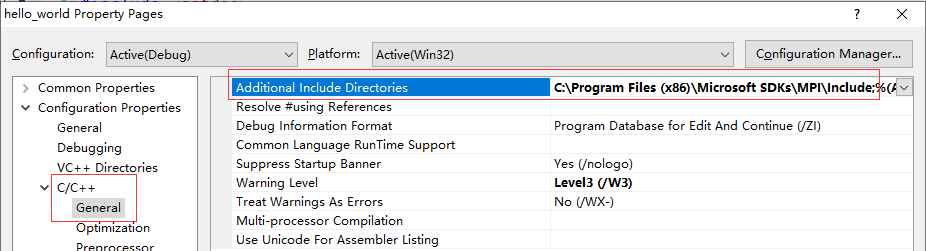
1. 在源码文件下双击打开hello\_world.cpp进行源码编辑：



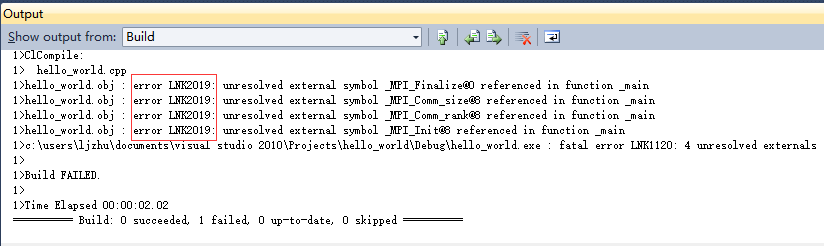
1. 鼠标指向代码编辑器中第一个红色下划线，显示无法打开源文件“mpi.h”，说明本项目尚未设置MPI环境，因此无法编译、运行；



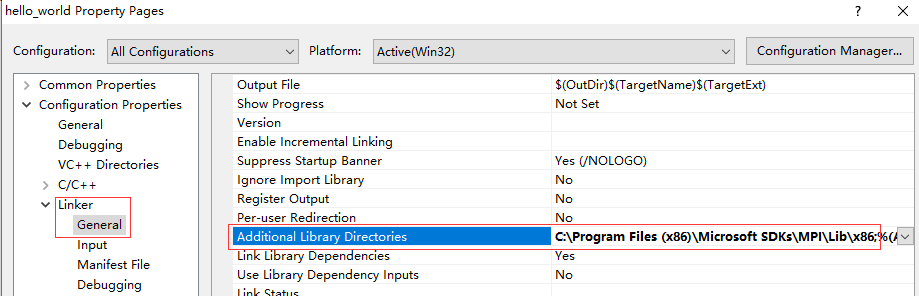
1. 右键“hello\_world”工程，选择“Properties”打开项目属性页进行设置：Configuration Properties -> C/C++ -> General -> Additional Include Directories，添加MPI库的头文件地址：C:\Program Files (x86)\Microsoft SDKs\MPI\Include



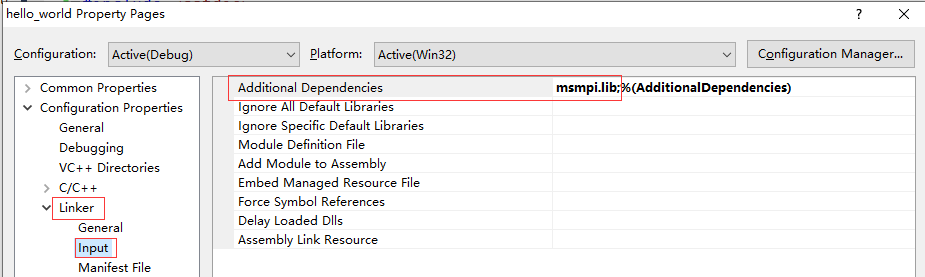
1. 点击确定之后，便发现代码中的红色下划线消失了，此时，我们便可以右键“hello\_world”工程，选择“Build”进行编译了，但是，随即便发现了失败信息，即error LNK2019: unresolved external symbol …，无法解析外部符号的链接错误，该错误为C/C++编程中最常见的错误之一，说明编译器无法找到代码中所引用（依赖）的库；



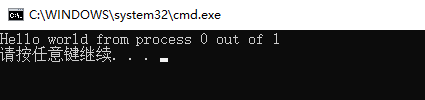
1. 右键“hello\_world”工程，选择“Properties”打开项目属性页进行设置：
   1. Configuration Properties -> Linker -> General -> Additional Library Directories，添加MPI库的静态链接库地址：C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x86



* 1. 在Input -> Additional Dependencies中添加msmpi.lib

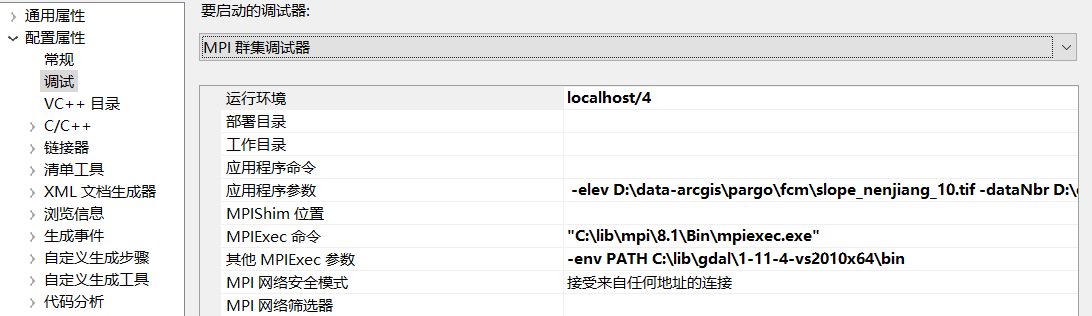


1. 再次编译即可成功，按Ctrl+F5键即可运行程序如下，这和正常的Hello World程序似乎无异：**只有一个进程输出了一行信息**；

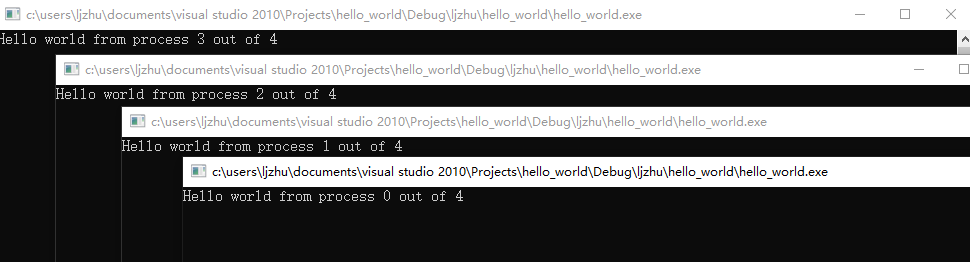


1. 如何运行MPI并行程序呢？
   1. 在开发阶段，可通过设置VS2010工程的调试信息，右键“hello\_world”工程，选择“Properties”打开项目属性页进行设置：

Configuration Properties -> Debugging，Debugger to launch选择MPI Cluster Debugger，Run Environment中输入localhost/4，即在本地计算机上采用4个进程运行该程序。应用程序参数

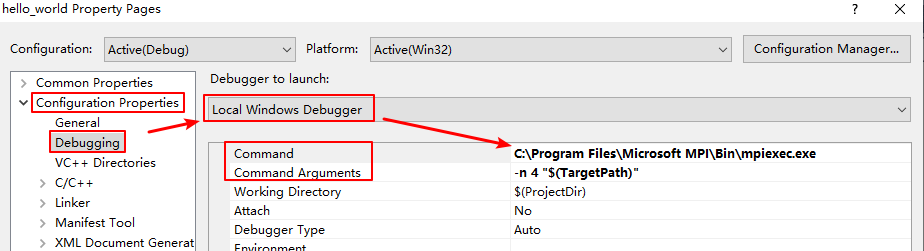


避免程序运行后直接结束，可按Ctrl+F5打开调试，同时弹出4个调试窗口，每个窗口的Hello world是从4个不同的进程（编号0~3）输出的

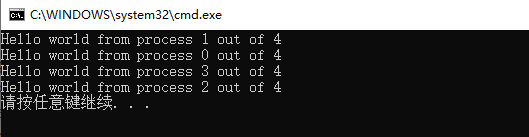


* 1. 如果不希望每个进程都开启一个调试窗口，VS2010同样支持采用本地调试，右键“hello\_world”工程，选择“Properties”打开项目属性页进行设置：

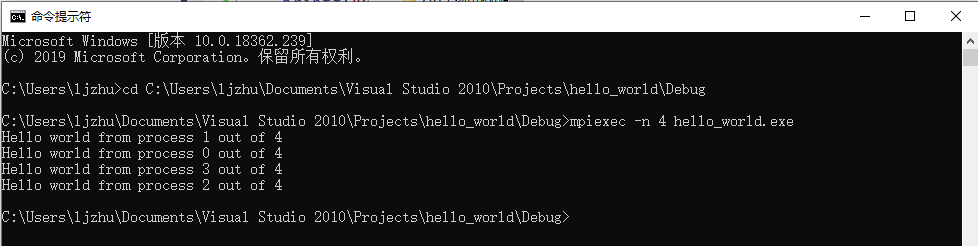
Configuration Properties -> Debugging，Debugger to launch选择：Local Windows Debugger，Command中输入mpiexec的完整路径，如C:\Program Files\Microsoft MPI\Bin\mpiexec.exe，在Command Arguments中输入-n 4 "$(TargetPath)"，调试时Command和Command Arguments即拼接成完整的程序调用命令，即mpiexec -n 4 hello\_world.exe



按Ctrl+F5打开调试，显示类似如下窗口（需要注意的是，由于4个进程各自独立执行打印语句，因此，该调试窗口显示的进程顺序并不是固定的，而是随机的，且每次运行均可能是不相同的）：



* 1. 开发完成后，可通过**命令行形式**调用程序，打开cmd.exe，通过cd命令进入hello\_world.exe程序所在文件夹，然后通过mpiexec命令运行MPI程序：mpiexec -n 4 hello\_world.exe，命令中-n是参数，后面的4表示有4个CPU核（或进程）进行并行计算



1. 程序说明



第二节 MPI基本编程函数介绍

在MPI-1中，共有128个调用接口，在MPI-2中有287个，应该说MPI是比较庞大的，完全掌握这么多的调用对于初学者来说是比较困难的。但是，从理论上说，MPI所有的通信功能可以用它的6个基本的函数来实现，掌握了这6个函数，就可以实现所有的消息传递并行程序的功能。

**注意：** 本教程中所有关于C++定义的MPI函数形式均不适用于MS-MPI的实现，因此，使用MS-MPI进行并行编程时，应当使用C定义格式。本教程示例将仅使用C格式。

## MPI的6个基本函数

1. **MPI初始化函数**

C定义：int MPI\_Init(int \*argc, char \*\*\*argv)

C++定义：void MPI::Init(int argc, char\* argv[])

功能：用命令行参数初始化MPI环境

输入：argc、argv—表示命令行参数，同C语言的main()函数参数格式，argv中包含欲并行运行的进程数

输出：在C语言中，返回值—非零/零表示初始化是否成功，在C++中返回void

说明：该函数必须为程序中第一个调用的MPI函数，它完成MPI程序所有的初始化工作，所有MPI程序的第一条可执行语句都是这条语句

C示例：MPI\_Init(&argc, &argv); // **argc、argv引用的是mian()函数的参数**

C++示例: MPI::Init(argc, argv);

1. **MPI终止函数**

C定义：int MPI\_Finalize (void)

C++定义：void MPI::Finalize(void)

功能：结束MPI程序的运行，指结束MPI环境的使用

输入：无

输出：在C语言中，返回值—非零/零表示结束MPI环境是否成功，在C++中返回void

说明：该函数必须为程序中最后一个调用的MPI函数，它结束MPI程序的运行，如果不调用它或者在它后面还有其它MPI函数，那么程序的运行结果将不可预知。

C示例：MPI\_ Finalize ();

C++示例：MPI::Finalize();

1. **MPI\_Comm\_size 函数**

C定义：int MPI\_Comm\_size(MPI\_Comm comm, int \*size)

C++定义：int MPI::COMM\_WORLD.Get\_size(void)

功能：获取总进程数

输入：在C语言中，comm 通信域句柄（系统默认的为MPI\_COMM\_WORLD，也可自己定义）；在C++语言中，COMM\_WORLD是communicator类的一个实例，Get\_size是它的方法

输出：size，即通信域comm内包括的进程数整数

说明：这一调用返回给定的通信域中所包括的进程的个数，不同的进程通过这一调用得知在给定的通信域中一共有多少个进程在并行执行

1. **MPI\_Comm\_rank 函数**

C定义：int MPI\_Comm\_rank(MPI\_Comm comm, int \*rank)

C++定义：int MPI::COMM\_WORLD.Get\_rank(void)

功能：获取本进程的进程号

输入：在C语言中，comm是该进程所在的通信域句柄；在C++语言中，COMM\_WORLD是communicator类的一个实例，Get\_rank是它的方法

输出：rank，调用进程在comm中的标识号

说明：这一调用返回调用进程在给定的通信域中的进程标识号，有了这一标识号，不同的进程就可以将自身和其它的进程区别开来，实现各进程的并行和协作

1. **MPI\_Send** 函数

C定义：int MPI\_Send(void\* buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)

C++定义：MPI::COMM\_WORLD.Send(const void\* buf, int count, MPI\_Datatype datatype, int dest, int tag)

功能：发送消息给特定的进程

输入：buf 发送缓冲区的起始地址(可选类型)

count 将发送的数据的个数(非负整数)

datatype 发送数据的数据类型(句柄)

dest 目的进程标识号(整型)

tag 消息标志(整型)

在C语言中，comm是通信域(句柄)；在C++语言中，COMM\_WORLD是communicator类的一个实例，Send是它的方法

输出：无

说明：MPI\_SEND将发送缓冲区中的count个datatype数据类型的数据发送到目的进程，其中datatype数据类型可以是MPI的预定义类型，**也可以是用户自定义的类型**，目的进程在通信域中的标识号是dest，本次发送的消息标志是tag，使用这一标志，就可以把本次发送的消息和本进程向同一目的进程发送的其它消息区别开来

1. **MPI\_Recv** 函数

C定义：int MPI\_Recv(void\* buf, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Status \*status)

C++定义：MPI::COMM\_WORLD.Recv(void\* buf, int count, MPI\_Datatype datatype, int source, int tag)

功能：接受别的进程发过来的消息

输入：count 最多可接收的数据的个数(整型)

datatype 接收数据的数据类型(句柄)

source 接收数据的来源即发送数据的进程标识号(整型)

tag 消息标识与相应的发送操作的表示相匹配相同(整型)

在C语言中，comm是通信域(句柄)；在C++语言中，COMM\_WORLD是communicator类的一个实例，Recv是它的方法

输出：buf 接收缓冲区的起始地址(可选数据类型)

Status：返回状态 (状态类型MPI\_Status)，返回状态变量status用途很广，它是MPI定义的一个数据类型，使用之前需要用户为它分配空间。在C实现中，状态变量是由至少三个域组成的结构类型，这三个域分别是：MPI\_SOURCE，MPI\_TAG和MPI\_ERROR，它还可以包括其它的附加域，这样通过对status.MPI\_SOURCE，status.MPI\_TAG和status.MPI\_ERROR的引用，就可以得到返回状态中所包含的发送数据进程的标识，发送数据使用的tag标识和本接收操作返回的错误代码

说明：MPI\_Recv从指定的进程source接收消息，并且该消息的数据类型和消息标识和本接收进程指定的datatype和tag相一致，接收到的消息所包含的数据元素的个数最多不能超过count，接收缓冲区是由count个类型为datatype的连续元素空间组成，由datatype指定其类型（MPI预定义数据类型见表2.1），起始地址为buf，接收到消息的长度必须小于或等于接收缓冲区的长度，这是因为如果接收到的数据过大，MPI没有截断，接收缓冲区会发生溢出错误，因此编程者要保证接收缓冲区的长度不小于发送数据的长度

**表2.1 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 |

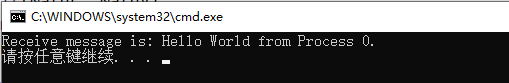
MPI\_BYTE和MPI\_PACKED数据类型没有相应于一个C的数据类型，类型MPI\_BYTE的一个值是由一个字节组成(8个二进制位)。一个字节不同于一个字符，因为对于字符的表示，不同机器可以用一个以上的字节表示字符，另一方面，在所有的机器上，一个字节有相同的二进制值。MPI\_PACKED相对复杂一些，将会在后面的学习中详细介绍。

一个简单的发送和接收的例子：

* 程序：



* 运行结果：



## MPI消息的传递

MPI消息传递过程：

数据发送缓冲区

数据接收缓冲区

消息传递

MPI消息传递过程示意图

MPI的消息传递过程可以分为三个阶段：

①消息装配，将发送数据从发送缓冲区中取出，加上消息信封等形成一个完整的消息。

②消息传递，将装配好的消息从发送端传递到接收端。

③消息拆卸，从接收到的消息中取出数据送入接收缓冲区。

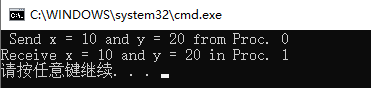
在这三个阶段，都需要**类型匹配**：①在消息装配时，发送缓冲区中变量的类型必须和相应的发送操作指定的类型相匹配；②在消息传递时，发送操作指定的类型必须和相应的接收操作指定的类型相互匹配；③在消息拆卸时，接收缓冲区中变量的类型必须和接收操作指定的类型相匹配。

在MPI消息传递过程中，**tag标识**也起了很重要的作用。当发送者发送两个相同类型的数据给同一个接收者时，如果没有消息标识，接收者将无法区别这两个消息。下面以程序示例来说明tag在MPI消息传递中的作用。

* 程序：



* 运行结果：



程序说明：标识为tag2的消息2到达前，由于没有匹配的接收操作，先等待，就是说进程0的消息1先到，x也不会被y接收，因为tag标识不匹配，进程1将等待，当标识为tag2的消息2到达，类型、标识、源完全匹配，进程1接收消息2，然后再匹配消息tag1的消息1，当接收操作相匹配，进程1接收消息1。

## MPI通信域（通信器）

MPI通信域包括两部分：**进程组**和**通信上下文**。进程组即所有参加通信的进程的集合，如果一共有N个进程参加通信，则进程的编号从0到N-1；通信上下文提供一个相对独立的通信区域，不同的消息在不同的上下文中进行传递，不同上下文的消息互不干涉，通信上下文可以将不同的通信区别开来。

在C语言中，MPI提供了一个预定义的通信域MPI\_COMM\_WORLD。MPI初始化后，便会产生这样一个通信域，它包括了初始化时可得的全部进程（进程组），这些进程是通过它们在MPI\_COMM\_WORLD中的进程号来进行标识的。在C++中，通信上下文和进程组被封装在一个类中，叫做通信器（communicator）。

用户可以在原有的通信域的基础上，定义新的通信域。通信域实际上为库和通信模式提供一种重要的封装机制。他们允许各模式有其自己的独立的通信域，以及它们自己的进程计数方案。

## 小结

MPI调用接口的总数虽然庞大，但是根据我们实际编写MPI程序的经验，最常使用的MPI调用的个数却是十分有限的。因此本节专门就最常使用的MPI调用拿出来进行讲解，目的就是希望读者能够较容易的进入MPI 并快速掌握MPI的基本程序设计方法。

参考文献或MPI学习网站：

1. <https://computing.llnl.gov/tutorials/mpi/>
2. <https://www.mcs.anl.gov/research/projects/mpi/>
3. <http://www.netlib.org/utk/papers/mpi-book/node1.html>
4. 都志辉. 高性能计算并行编程技术——MPI并行程序设计. 北京：清华大学出版社，2001
5. William Gropp, Eving Lusk, Anthony Skjellum. 1999. Using MPI portable Parallel Programming with the Message-passing Interface

## 作业

开4个进程，在0号进程中，在屏幕显示要求输入2个数，输入后把这2个数分别传到线程1和线程2；在线程1中，把接收到的2个数相加，并把相加结果传到3号线程；在线程2中，把接收到的2个数相乘，并把乘积的结果传到3号线程；在线程3中，把分别从1号、2号线程接收的计算结果打印到屏幕。

线程0：输入2个数

线程1：2个数相加

线程2：2个数相乘

线程3：计算结果打印

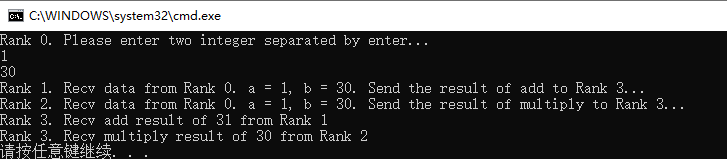
传递这2个数

传递这2个数

传递计算结果

传递计算结果

* 运行结果示例：



第三节 更多MPI函数与程序实例

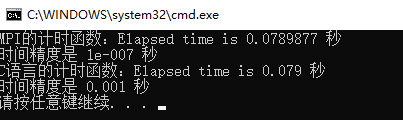
## 用MPI实现计时功能

在MPI程序中，经常会用到时间函数，比如用来统计程序运行的时间，或根据时间的不同选取不同的随机数种子，或根据时间的不同对程序的执行进行控制等。通过下面一个例子，介绍MPI时间函数，并与C++的“time.h”中的计时函数做了对比。

* 代码：



* 运行结果：



从实验结果来看，两者的主要的区别是：MPI 中由函数 double MPI\_Wtime(void) 取得当前时间，计时的精度由double MPI\_Wtick(void) 取得；C/C++由clock\_t clock(void) 取得当前时间，计时的精度由常数 CLOCKS\_PER\_SEC 定义；MPI\_Wtime 计时精度更高。

## MPI任意进程之间相互问候

在许多情况下需要任意两个进程之间都进行数据的交换，下面给出一个例子，任意进程都向其它的进程问好：

任意进程间相互问候

hello

hello

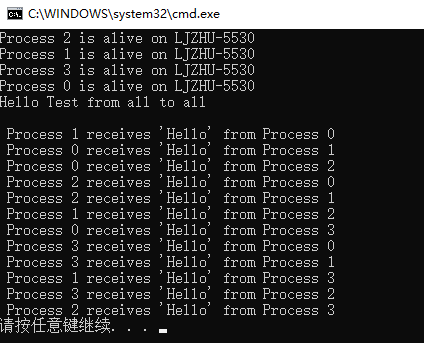
hello

hello

hello

hello

* 运行结果：



* 程序：



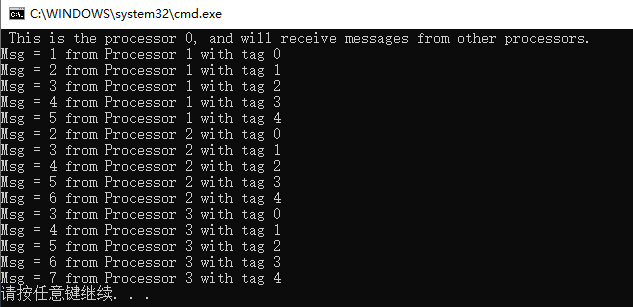
## 接收任意源和任意标识的消息

在接收操作中，通过使用**任意源**和**任意tag标识**，使得该接收操作可以接收任何进程以任何标识发送给本进程的数据，但是**该消息的数据类型必须和接收操作的数据类型相一致**。下面给出一个使用任意源和任意标识（**MPI\_ANY\_SOURCE**和**MPI\_ANY\_TAG**）的例子，其中进程0接收来自其它所有进程的消息，然后将各消息的内容、消息来源和消息标识打印出来。

* 程序：



* 运行结果：



## MPI 程序实例——数值积分 (π 值的计算)

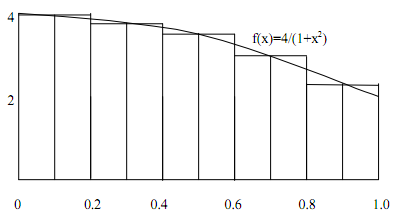
摘自MPICH实例，数值积分计算π。



令函数f(x) = 4/(1+x2)，则有



而*f*(*x*)的图像为



函数f(x) = 4/(1+x2)图像

计算*f*(*x*)图像下面从0到1之间的面积即为*π*的值。而该面积可以用图示的5个小矩形面积的和来近似，矩形的高度取函数在矩形中间点的取值，当用更多的矩形来划分时，该近似值就越接近于真实的*π*值，设将0到1的区间划分为*N*个矩形，则近似公式为

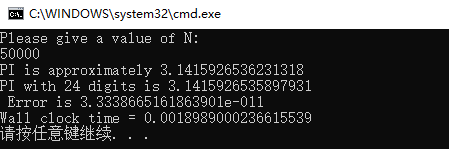


如果总共有*p*个进程，每个进程独立计算上式中的一部分，然后再调用Reduce函数，将各个进程的结果加起来，即为*π*的近似值。由此，可得求*π*的程序如下：

* 程序：



* 运行结果：



## 自定义通信器（通信域）

MPI是按**进程组**方式工作，所有MPI程序在开始时均被认为是在通信器MPI\_COMM\_WORLD所拥有的进程组中工作。用户也可以根据自己的需要，建立其它进程组和通信域。前面的示例都用预定义的MPI\_COMM\_WORLD通信域来实现进程间的通信，下面通过利用Monte Carlo 方法求*π*的例子来介绍如何创建进程组和新的通信器（域）。

虽然这不是一个很好的方法求*π*，但这是一个通信器创建的很简单的例子。如图3.5所示，如果一个圆的半径是1，面积是*π*，圆的外接正方形的面积为2\*2=4，圆面积与正方形面积的比*r* 为*π*/4。我们通过在这个正方形上产生随机样点(*x*, *y*)，并计算有多少个随机样点在圆内（通过判断点坐标是否在圆内，即*x*2 + *y*2 < 1，*π*= 4*r*），来计算*r*的值。这样我们就有，*π*= 4\**r*。

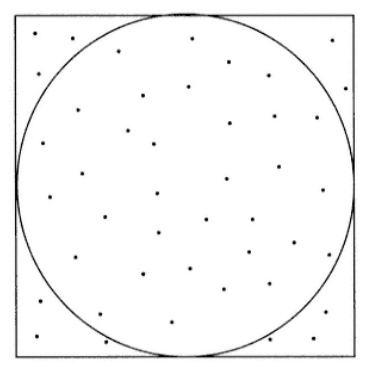


图3.5 Monte Carlo 方法计算π

本例中的所用到的MPI函数有：

int **MPI\_Comm\_group**(MPI\_Comm comm, MPI\_Group \* group)

作用：在group参数中返回指定包含的进程组。

int **MPI\_Group\_free**(MPI\_Group \*group)

作用：释放指定的进程组。函数返回时会将 group 置成 MPI\_GROUP\_NULL以防止释放后被误用。实际上，该函数只是将该进程组加上释放标志。只有基于该进程组的所有均被释放后才会实际将其释放。

int **MPI\_Group\_excl**(MPI\_Group group, int n, int \*ranks, MPI\_Group \*newgroup)

作用：该函数将进程组 group 的进程集合减去一个子集而得到一个新进程组 newgroup，减去的进程号由数组 ranks 给出，*n* 是 ranks 中的进程数。新进程组的进程号保持进程在老进程组中顺序。

int **MPI\_Comm\_create**(MPI\_Comm comm, MPI\_Group group, MPI\_Comm \*comm\_out)

作用：创建一个包含指定进程组 group 的新 comm\_out。这个函数并不将 comm 的属性传递给 comm\_out，而是为 comm\_out 建立一个新的上下文。返回时，属于进程组 group 的进程中 comm\_out 等于新的句柄，而不属于进程组 group 的进程中 comm\_out 则等于 MPI\_COMM\_NULL。

int **MPI\_Comm\_free**(MPI\_Comm \*comm)

作用：释放指定的。函数返回时会将 comm 置成 MPI\_COMM\_NULL 以防止释放后被误用。实际上该函数只是将加上释放标志。当所有引用该通信的操作全部完成后才会实际将其释放。

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

作用：全归约。它与普通归约函数 MPI\_Reduce 的操作类似，但所有进程将同时获得归约运算的结果，op 是归约操作的二目运算。它除了比 MPI\_Reduce 少一个 root 参数外，其余参数及含义与后者一样。 **MPI\_Allreduce 相当于在 MPI\_Reduce 后马上再调用 MPI\_Bcast 广播归约结果**。

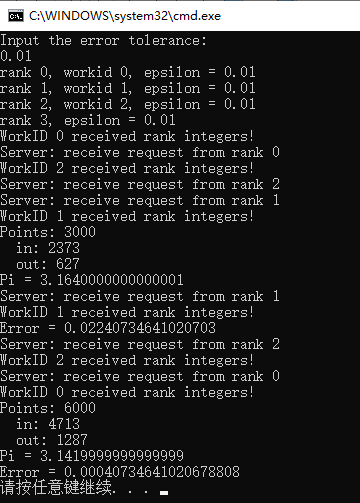
程序说明：本例是根据用户输入容差的大小，计算符合精度的π值。并行随机数生成器问题比较复杂[1,2]，因此本示例考虑将随机数生成进程（server）和计算进程（worker）分开，需要将server进程从原来的进程组中排除，而得到一个新的进程组，worker\_group进程组（语句：MPI\_Group\_excl ( world\_group, 1, ranks, &worker\_group )）。由于需要进行worker\_group进程间的通信，因此需要创建一个新的workers（语句：MPI\_Comm\_create( world, worker\_group, &workers )）。这样做的目的是不同的进程在不同的（world和works）中有不同的通信上下文，通信互不干扰。

* 程序：





* 运行结果（mpiexec.exe -n 4，**注意：每次运行结果是不同的**）：



## 小结

本节主要介绍了**MPI\_Wtime()**函数，功能是获得当前时间，单位是秒，MPI\_Wtick()函数，返回MPI\_Wtime的精度，单位是秒；**MPI\_Abort()** 语句的功能是结束通信域comm中的所有进程；**MPI\_Get\_processor\_name ()** 函数的作用是获取当前机器名字和名字的长度；**MPI\_Barrier()**进程同步函数；使用**MPI\_ANY\_SOURCE, MPI\_ANY\_TAG**接收任何进程以任何标识发送给本进程的数据。还介绍了MPI定义的一种数据类型，MPI\_Status，MPI消息传递中非常有用；比较常用的广播功能，**MPI\_Bcast()**函数；以及**MPI\_Reduce()**归约函数的使用。最后通过Monte Carlo方法计算*π*示例，介绍如果**创建进程组和通信器**。

第四节 并行I/O

本节介绍对文件各种方式的并行输入/输出（I/O）操作，主要包括直接对文件指定位置数据的读写、独立文件指针的读写数据以及共享文件指针的读写。

## 基本术语

**基本数据类型 (elementary type，etype)** 是定义一个文件最小访问单元的 MPI 数据类型。一个文件的基本数据类型可以是任何预定义或用户**构造的并已递交**的 MPI 数据类型（详见MPI标准文档MPI\_Type\_commit）。MPI 的文件操作完全以基本数据类型为单位：文件中的**位移 (oﬀset)** 以基本数据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。

**文件类型 (ﬁletype)** 文件类型也是一个 MPI 数据类型，它定义了对一个文件的存取模板。文件类型可以等于基本数据类型，也可以是在基本数据类型基础上**构造并已递交**的任意MPI 数据类型。文件类型的大小必须是基本数据类型大小的倍数，并且文件类型中间的“空洞”的大小也必须是基本数据类型大小的倍数。

**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。

**绝对偏移 (displacement)** 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“视图”的起始位置。

**视图 (view)** 每个进程对打开的文件都有其自己的视图，视图定义了处于打开的文件中的以有序etype为类型的可访问的数据集合。文件视窗由三个参数定义：绝对偏移，基本数据类型（etype），文件类型（filetype）。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”，见图4-1。

**etype**

**filetype**

**……**

**绝对偏移**

**filetype**

**filetype**

**filetype**

**空洞**

图4-1以文件类型为单位的对文件条块化

一组进程之间可通过互补视图以散发/收集等方式实现数据分布。如图4-2所示。实际上，视图是利用抽象数据结构对文件的一个**虚拟化描述**，或者称为文件的逻辑描述。

**etype**

**……**

**进程0的filetype**

**进程1的filetype**

**进程2的filetype**

**绝对偏移**

**filetype**

**filetype**

**……**

**……**

**……**

**文件开始**

图4-2 以文件类型为单位在进程间实现文件条块化分割

**相对偏移 (oﬀset)** 指相对当前视图的文件位置，以基本数据类型为计数单位。在计数时，不考虑视图文件类型（filetype）中的空洞。如果offest=0指向视图中第一个基本数据类型（etype），即文件开始跳过定义视图的绝对偏移和视图内部空洞之后的基本数据类型的总数。例如在视图定义的图4-2中，**对进程1，如果offset=2则指向定义视图的绝对偏移后第8个基本数据类型处**。

**文件指针(ﬁle pointer)** 是指MPI维护文件的指针，主要包括两个含义：一个供**本进程独立**使用，称为独立文件指针 (individual ﬁlepointer)；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 ﬁlepointer)。

**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。

**文件大小 (ﬁle size)** 指从文件开头到文件结尾的总字节数。

## 基本文件操作

int **MPI\_File\_open**(MPI\_Comm comm, char \* filename, int amode, MPI\_Info info,

MPI\_File \* fh)

MPI\_File\_open是一个组调用，即通信域comm中的所有进程都必须按给定的文件名filename来执行它，各进程使用的文件名filename要相同，给定的打开方式amode也要相同，但是各进程传递给运行时的信息info可以互不相同。该调用返回一个文件句柄fh，以后各进程对文件的具体操作都是通过文件句柄fh来实现的。

根据用途的不同，打开方式可以是表格4-1所示的任何9种方式之一。

表4-1 文件打开方式

|  |  |
| --- | --- |
| **打开方式** | **含义** |
| MPI\_MODE\_RDONLY | 只读 |
| MPI\_MODE\_RDWR | 读写 |
| MPI\_MODE\_WRONLY | 只写 |
| MPI\_MODE\_CREATE | 若文件不存在则创建 |
| MPI\_MODE\_EXCL | 创建不存在的新文件，若存在则报错 |
| MPI\_MODE\_DELETE\_ON\_CLOSE | 关闭时删除文件 |
| MPI\_MODE\_UNIQUE\_OPEN | 不能并发打开，确保只有当前程序访问该文件 |
| MPI\_MODE\_SEQUENTIAL | 文件只能顺序存取 |
| MPI\_MODE\_APPEND | 追加方式打开，初始文件指针指向文件尾 |

int **MPI\_File\_close**(MPI\_File \* fh)

MPI\_File\_close关闭前面已经打开的与句柄fh相联系的文件，它也是一个组调用，即所有打开该文件的进程，也必须都执行关闭操作，这里虽然没有明确指出通信域或进程组，但是，文件句柄fh已包含了进程组的信息。

int **MPI\_File\_delete**(char \* filename, MPI\_Info info)

MPI\_File\_delete删除指定的文件filename，其中info参数是传递给运行时的信息。

int **MPI\_File\_set\_size**(MPI\_File fh, MPI\_Offset size)

MPI\_File\_set\_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。它也是一个组调用，进程组中所有进程必须同时调用并且提供同样的参数。

int **MPI\_File\_preallocate**(MPI\_File fh, MPI\_Offset 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。它也是一个组调用，进程组中所有进程必须同时调用并且提供同样的参数。

int **MPI\_File\_get\_size**(MPI\_File fh, MPI\_Offset \*size)

在参数 size 中返回指定文件的当前长度。

int **MPI\_File\_get\_group**(MPI\_File fh, MPI\_Group \*group)

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。

int **MPI\_File\_get\_amode**(MPI\_File fh, int \*amode)

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。

## 文件视图

不同进程对应的文件指针可以是互不相同的，它们可以分别指向同一文件的不同位置。而视图是相对于某一进程来说的，它是特定进程所能看到的文件，某一进程的文件视图可以是整个文件，但多数情况下，文件视图只是整个文件的一个或几个部分，文件视图在整个文件中对应的部分可以是不连续的，但各个进程看到的其文件视图中的数据却是连续的。如图4-3所示，文件与视图的关系。**不同的进程，通过在相同的文件上定义互不交叉的文件视图，就可以实现对文件的并行访问。**

**文件不同部分向视图的映射**

**视图1**

**视图2**

**视图3**

**数据**

图4-3 文件与视图的关系

int **MPI\_File\_set\_view**(MPI\_File fh, MPI\_Offset disp, MPI\_Datatype etype,

MPI\_Datatype filetype, char \*datarep, MPI\_Info info)

MPI\_File\_set\_view设置文件视图，它是一个组调用，所有与fh相联系的进程组中的进程都执行这一调用。调用进程在fh对应的文件中设置本进程的文件视图，该视图相对于文件头的偏移是disp，即视图首先从文件中跳过disp个字节，然后给出了视图数据的基本数据单位etype，以后所有对该视图的访问必须以etype为单位来进行，而filetype则在etype的基础上，通过以etype为单位定义数据类型filetype，将视图不需要的数据排斥在外，即一般filetype定义的数据类型是有“空洞”的，这些空洞是视图无需访问的数据。

"datarep"参数用于指定视图数据的表示方法，共有三种：native、internal和external32，定义数据表示是为了高效解决MPI的一致性问题，因为不同类型的计算机，其数据的表示方法是不同的。

"native"，文件中数据完全按其在内存中的表示形式存放。优点是不需要进行类型转换，可避免数据精度损失以及减少转换所耗时间。缺点是无法实现异构环境下跨平台。

"internal"，指 MPI 内部格式，具体由 MPI 的实现定义。可用于同构和异构环境中，MPI环境负责相应的数据类型转换。

"external32"，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。缺点是由于数据格式转换可能会导致精度丢失并增加时间开销。

"native"和"internal"两种表示方式取决于具体的MPI版本，而"external32"则在所有MPI环境中提供完全一致的解析协议。

MPI\_File\_set\_view调用完成后，原来的文件句柄fh就不再代表该文件，而是代表本调用产生的文件视图，以后使用fh对文件的所有操作都是对其视图的操作。

int **MPI\_File\_get\_view**(MPI\_File fh, MPI\_Offset \* disp, MPI\_Datatype \* etype,

MPI\_Datatype \* filetype, char \* datarep)

MPI\_File\_get\_view是一个查询调用，它返回文件视图的各种参数，fh是给定的文件视图句柄，disp是该视图在文件中的起始偏移位置，etype是文件视图的基本数据单位类型，filetype是文件视图的文件类型，datarep是文件视图的数据表示方法，以上各个参数对应于MPI\_File\_set\_view调用时所给出的各种参数。

int **MPI\_File\_seek**(MPI\_File fh, MPI\_Offset offset, int whence)

MPI\_File\_seek将文件的指针移动到给定的位置。其中fh是文件句柄，offset是相对于whence的偏移位置，它的值可正可负。其中whence的取值可以为MPI\_SEEK\_SET、MPI\_SEEK\_CUR和MPI\_SEEK\_END。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；

## 使用绝对偏移指针读写文件

int **MPI\_File\_read\_at**(MPI\_File fh, MPI\_Offset offset, void \* buf, int count,

MPI\_Datatype datatype, MPI\_Status \* status)

MPI\_File\_read\_at在文件fh中，从指定的偏移位置offset开始，读取count个数据类型为datatype的数据，存放到数据缓冲区buf之中，status是该读写操作完成后返回的状态参数，如图4-4所示。

**buf（输出缓冲区）**

**数据类型type1**

**文件头**

**读取数据起始位置offset=100**

**MPI\_FILE\_READ\_AT(fh,100,buf,5,type1,status)**

图4-4 MPI\_FILE\_READ\_AT示意图

int **MPI\_File\_write\_at**(MPI\_File fh, MPI\_Offset offset, void \* buf, int count,

MPI\_Datatype datatype, MPI\_Status \* status)

MPI\_File\_write\_at与MPI\_File\_read\_at相对应，它向文件句柄fh对应中，从指定的位置offset开始，将数据缓冲区buf中count个类型为datatype的数据，写入到该文件中，其中status是返回的状态参数。如图4-5所示。

**buf（输出缓冲区）**

**数据类型type1**

**文件头**

**写入数据起始位置offset=100**

**MPI\_FILE\_WRITE\_AT(fh,100,buf,5,type1,status)**

图4-5 MPI\_FILE\_WRITE\_AT示意图

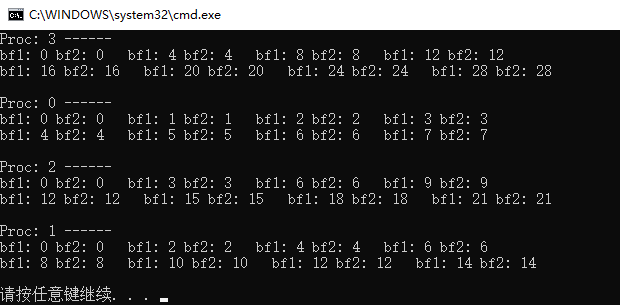
下面程序演示绝对偏移指针读写文件。

* 程序：





* 运行结果：



## 使用独立文件指针读写文件

int **MPI\_File\_read**(MPI\_file fh, void \* buf, int count, MPI\_Datatype datatype,

MPI\_Status \* status)

MPI\_File\_read从视图文件fh中读取数据，读取数据的类型是datatype，读取数据的个数是count，读取的数据放到buf缓冲区中。这里没有指明读取的位置，该位置是隐含指定的，它就是当前视图文件句柄指针的位置，该读取操作完成后，视图文件句柄指针自动指向下一个视图内的基本数据类型的位置。

int **MPI\_File\_write**(MPI\_file fh, void \* buf, int count, MPI\_Datatype datatype,

MPI\_Status \* status)

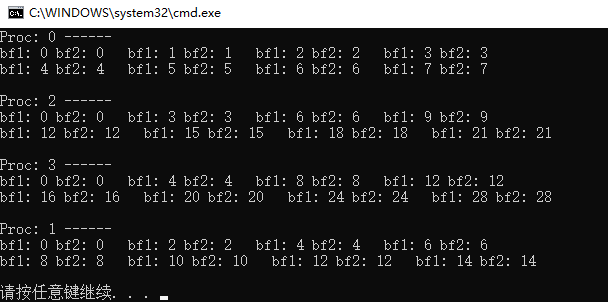
MPI\_File\_write将buf中的count个数据类型为datatype的数据写入到视图文件中。MPI\_File\_write和MPI\_File\_read是对应的，写入位置也是由当前视图文件句柄的指针隐含指定的，写入完成后，当前视图文件句柄指针指向视图中下一个基本数据单元的位置。

下面程序示例演示使用独立文件指针读写数据。

* 程序：



* 运行结果：



## 使用共享文件指针读写文件

int **MPI\_File\_read\_shared**(MPI\_File fh, void \* buf, int count, MPI\_Datatype datatype,

MPI\_Status \* status)

int **MPI\_File\_write\_shared**(MPI\_File fh, void \* buf, int count, MPI\_Datatype datatype,

MPI\_Status \* status)

int **MPI\_File\_read\_ordered**(MPI\_File fh, void \* buf, int count, MPI\_Datatype datatype,

MPI\_Status \* status)

int **MPI\_File\_write\_ordered**(MPI\_File fh, void \* buf, int count, MPI\_Datatype datatype,

MPI\_Status \* status)

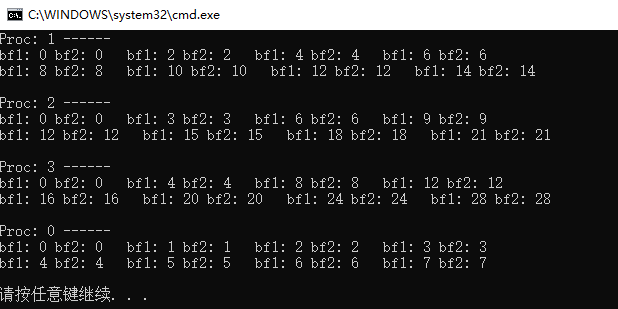
由于使用共享文件指针的文件操作函数中进程组的全部进程共同使用和修改同一个文件指针，因此这类操作非常类似于为“根进程”的数据收集和散发，即它们相当于将进程组中各进程的数据块合并写入文件 (收集) 或读取文件中的数据并分发给各进程 (散发)，当使用函数MPI\_File\_read\_shared 和 MPI\_File\_write\_shared时，各进程从文件中读取或写入文件的数据块在文件中的相对位置是不确定的，而调用函数 MPI\_File\_read\_ordered 和 MPI\_File\_write\_ordered则可确保这些数据块在文件中严格按进程序号排列。因此使用函数MPI\_File\_read\_shared和 MPI\_File\_write\_shared 时，要求程序自行保证读写操作的确定性（位置确定和顺序确定）。

下面程序示例演示共享文件指针读写文件

* 程序：



* 运行结果：



而用上面示例用 MPI\_File\_write\_shared和MPI\_File\_read\_shared函数分别替换MPI\_File\_write\_ordered和MPI\_File\_read\_ordered函数后，**可能会产生不确定的结果**。

## 作业

1、准备2个二进制文件：输出0~99999999范围的100000000个浮点型数的二进制文件data1.dat, 复制一下改名为data2.dat；

2、并行读取文件data1.dat、data2.dat，并将其对应的序列相加后，并行输出到二进制文件out.dat；

3、要求对步骤2计时，与串行程序比较。

作业提交要求，代码和屏幕打印截图。