
Parallel Programming with MPI

Binding with Fortran, C, C++

KONG, Tao
School of Mathematics
Shandong University

2010/10/11

目录

- [并行简介](#)
- [MPI概述](#)
- 消息传递
- [MPI程序结构](#)
- 传递成功
- [点对点通信](#)
- [群集通信](#)
- MPI扩展

并行简介

- 何谓并行
- 如何创建并行程序
- 并行结构
- 并行编程模型

何谓并行

- 多个线程同时进行工作。
- 就像电路的并联可以起到分流的作用一样。当你遇到下面的问题时可以考虑使用并行
 - 降低解决问题的运行时间
 - 增大要解决的问题的尺度

如何创建并行程序

- 把要解决的问题分解到一些子任务
 - 理想的状态是各个子任务可以互不影响的工作
- 把这些任务映射到不同的进程
- 进程有共享或本地数据
 - 共享：被多于一个线程使用
 - 本地：对每个进程私有
- 利用一些并行编程环境写源码

并行结构

- 分布存储
 - 每个处理器使用本地存储
 - 不能直接访问其他处理器的内存
- 共享存储
 - 处理器可以直接访问属于其他处理器的存储空间
 - 不同的处理器可以在同一个内存总线上
- 混合存储

并行编程模型

- 分布存储系统

- 处理器之间为了共享数据，必须由程序员显式的安排如何通信——所谓“消息传递”
- 消息传递的库
 - MPI (Message Passing Interface)
 - PVM (Parallel Virtual Machine)

- 共享存储系统

- 基于“进程”的编程。不同进程可以指定拥有共同的存储空间
- 当然也可以显示的指定消息传递。



MPI

- 什么是MPI

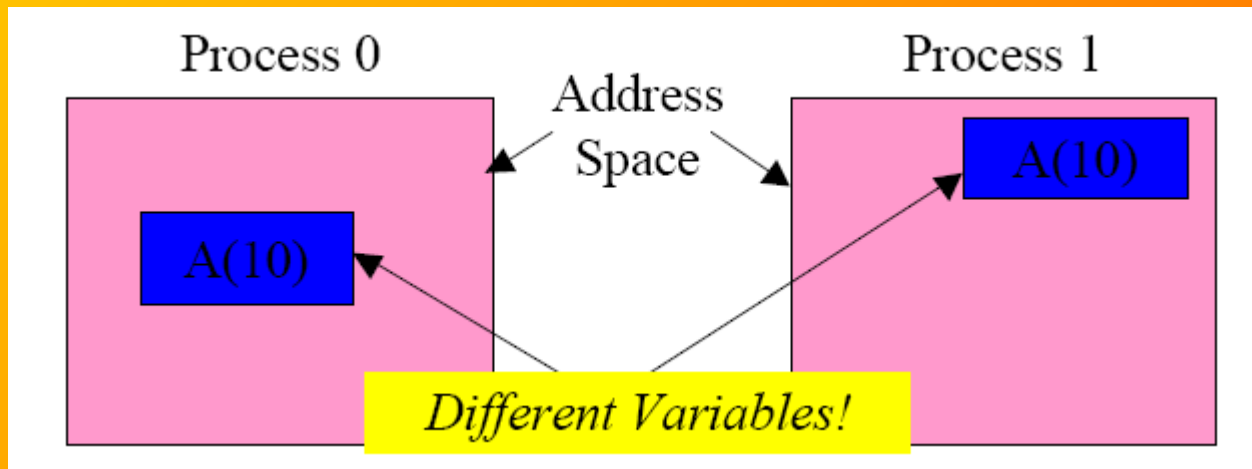
- Message Passing Interface
- MPI不是一种语言，他只是一个库，是通过绑定到Fortran, C, C++来实现的
- 所有的并行操作均是通过调用预定义的一些方法来实现的。这些方法均是一些函数。对于C++，引入面向对象机制，把这些函数抽象成类和对象来实现。
- 这些预定的操作定义在
 - C/C++: Mpi.h
 - Fortran: Mpif.h, 或者使用模板 mpi(对于fortran90之后)

- 实现软件

- MPICH
- Open MPI

传递的消息是什么

- 数据
 - 可以是数，数组，字符，等等



MPI的语言绑定

- 前面提到MPI是一个库。
windows/linux下使用tree 命令可以看到MPICH不过是安装了一些库，和对这些库说明的头文件。
 - 说明：更深入的剖析发现，linux下mpicc/mpicxx/mpif90/mpif77等mpich自带的编译命令一些shell文件。默认使用gcc/c++/ifort编译器
- MPI这个库的实现有3中语言格式C，c++，Fortran。因此对MPI的使用必须和特定的语言结合起来使用。
- 因为C和C++的相识性，MPI的C语言实现完全可以和C++绑定起来使用。这也是目前网上大部分的所谓“MPI+C++”的实现方式。

```
D:\MPICH2
├--include
|   clog_commset.h
|   clog_const.h
|   clog_inttypes.h
|   clog_uuid.h
|   mpe.h
|   mpe_log.h
|   mpe_logf.h
|   mpe_misc.h
|   mpi.h
|   mpi.mod
|   mpicxx.h
|   mpif.h
|   mpio.h
|   mpi_base.mod
|   mpi_constants.mod
|   mpi_sizeofs.mod
└--lib
    cxx.lib
    fmpich2.lib
    fmpich2g.lib
    fmpich2s.lib
    libfmpich2g.a
    libmpi.a
    libmpicxx.a
    mpe.lib
    mpi.lib
```



MPI程序结构

- Handles: 句柄,控制项 , etc
- 头文件
- MPI函数格式
- 一个简单的例子
- 如何编译和运行
- 含有通信的例子
- 常用基本函数
- 消息
- 通信域
- 返回上级

Handle 句柄

- C实现

- 句柄为声明的数据类型

- Fortran实现

- 缓冲区，可以是任意类型的空间。
 - 在本文中，用 “<type> buff(*)” 表示
 - 其他全部是整形

- C++实现

- 多为类;整个MPI为一个命名空间。
 - 如果在文件开头加上
 - using namespace MPI;
- 则在后面的调用中，可以省略MPI::.

头文件

- MPI 的常数和预定义操作都定义在这

- C/C++

```
#include <mpi.h>
```

- Fortran

```
include 'mpif.h'  
use mpi
```

- 注意各大编译器之间的mod文件并不兼容。MPICH默认使用intel的fortran编译器。欲使用gfortran需从新编译生成mpi.mod文件。

MPI函数格式

➤ C/C++:

```
error=MPI_Aaaa_aaaa (parameter,...) ;
```

```
MPI_Aaaa_aaaa (parameter,...) ;
```

➤ Fortran:

```
CALL MPI_AAAA_AAA (parameter,..., IERROR)
```

➤ C++

```
returnvalue=MPI::AAAA;
```

```
returnvalue=MPI::AAAA.Aaaa_aaaa (parameter,...) ;
```

第一个MPI程序

- 最基本的问题是：
 - 怎么进入MPI环境，怎么退出
 - 我怎么确定我用了几个进程，到底哪个进程是当前进程？
- MPI提供函数来解决这些问题
 - *MPI_Init* 进入并行环境
 - *MPI_Finalize* 退出并行环境
 - *MPI_Comm_size* 报告有多少进程
 - *MPI_Comm_rank* 报告进程编号， $0 \sim size-1$.

Hello(Fortran)

```
program main
```

```
    use mpi    ! or !include 'mpif.h'
```

```
    implicit none
```

```
    integer ierr, rank, size,msg
```

```
    call MPI_INIT( ierr )
```

```
    call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
```

```
    call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
```

```
    print *, 'I am ', rank, ' of ', size
```

```
    call MPI_FINALIZE( ierr )
```

```
end
```


Hello(C)

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Hello(C++)

```
#include <iostream>
#include <mpi.h>
using namespace std;

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI::Init(argc, argv );
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    cout << "I am " << rank << " of " << size << endl;
    MPI::Finalize();
    return 0;
}
```

Hello的一些说明

- MPI_COMM_WORLD是全局通信域，包含所有的进程，在MPI启动时自动产生
- 各个进程之间没有任何影响，每个语句独立的在各进程中执行。即使是输出语句
 - 所以如果输出很多，输出结果可能很乱

如何编译运行MPI程序

- MPICH-2提供了专门的命令用来编译运行mpi程序。

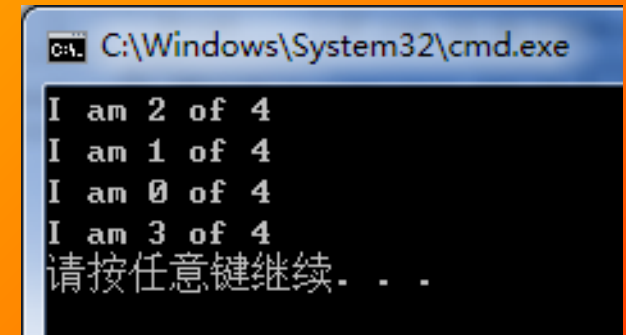
- 编译

- F77: mpif77
 - F90: mpif90
 - C: mpicc
 - C++: mpicxx

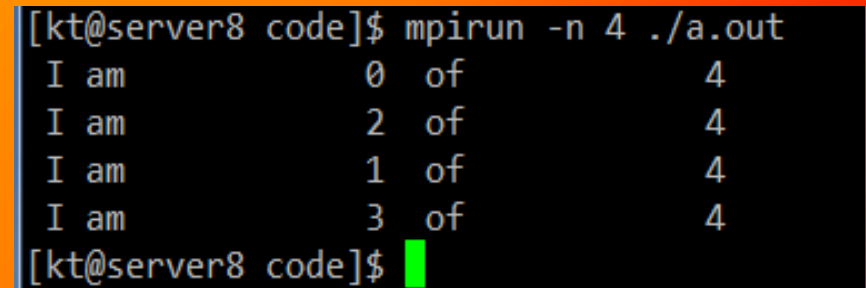
- 运行

- mpiexec -np 10 ./hello
 - mpirun -np 10 ./hello

- 运行结果



```
C:\Windows\System32\cmd.exe
I am 2 of 4
I am 1 of 4
I am 0 of 4
I am 3 of 4
请按任意键继续...
```



```
[kt@server8 code]$ mpirun -n 4 ./a.out
I am 0 of 4
I am 2 of 4
I am 1 of 4
I am 3 of 4
[kt@server8 code]$
```

一个最基本的带通信的实例

- MPI中最基本的就是消息传递机制。
- 下面的这个例子可以说明如何通信的。
 - 进程0创建一随机数数组，并发给进程1
 - 进程1输出接收前后缓冲区的值

Program

```
program mpicomu  
  use mpi  
  implicit none
```

mpicomu1/2(Fortran)

```
  integer,parameter :: num=10  
  real,dimension(num):: buf  
  integer :: i  
  
  integer ::rank,ierror  
  
  call MPI_Init(ierror)  
  call MPI_Comm_rank(MPI_COMM_WORLD,rank,ierror)  
  
  select case (rank)  
  case (0)  
    !call Initial_rand()  
    do i=1,num  
      call random_number(buf(i))  
      buf(i)=buf(i)*100  
    end do  
  
    call MPI_Send(buf,num,MPI_REAL,1,1,MPI_COMM_WORLD,ierror)
```

case(1)

```
print *, "Before receiving..."
print "(5(F4.1,2X))", (buf(i), i=1, num)
```

```
call MPI_Recv(buf, num, MPI_REAL, 0, 1, MPI_COMM_WORLD, ierror)
print *, "After receiving..."
print "(5(F4.1,2X))", (buf(i), i=1, num)
```

```
case default
```

```
print "(l2,': No operation)", rank
```

```
end select
```

```
call MPI_Finalize(ierror)
```

```
end program
```

```
[kt@server8 code]$ mpirun -n 2 ./a.out
Before receiving...
0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0
After receiving...
0.0  2.5  35.3  66.7  96.3
83.8  33.5  91.5  79.6  83.3
[kt@server8 code]$
```

1. MPI初始化

- 通常是MPI的第一个函数调用，通过它进入MPI环境

- C/C++

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

- Fortran

```
integer ierror  
call MPI_Init(ierror)
```

- C++

```
MPI::Init( int argc, char** argv);  
MPI::Init();
```


2. MPI结束

- MPI程序的最后一条MPI语句，从MPI环境退出

- C/C++

```
int MPI_Finalize();
```

- Fortran

```
integer ierror  
call MPI_Finalize(ierror)
```

- C++

```
MPI::Finalize();
```

3. MPI中止

- 出现特殊情况，需要中途停止MPI

- C/C++

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

- Fortran

```
integer comm, errorcode, ierror
```

```
MPI_ABORT(COMM, ERRORCODE, IERROR)
```

- C++

```
void MPI::Comm::Abort(int errorcode);
```

```
eg. MPI::MPI::COMM_WORLD.Abort(errcode);
```

4. 获取通信域大小

- 在一个通信域中有多少个进程

- C/C++

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

- Fortran

```
integer comm, size, ierror  
call MPI_Comm_size(comm, size, ierror)
```

- C++

```
int MPI::COMM::Get_size() const;  
eg. size=MPI::COMM_WORLD.Get_size();
```

5. 获取进程编号

- 取的进程编号，取值在 $[0, \text{size}-1]$, 其中size是进程总数。

- C/C++

```
int MPI_Comm_rank(MPI_Comm comm, int *size);
```

- Fortran

```
integer comm, size, ierror  
call MPI_Comm_rank(comm, size, ierror)
```

- C++

```
int MPI::COMM::Get_rank() const;  
eg. size=MPI::COMM_WORLD.Get_rank();
```

6. 消息发送

- C++版的特点是将通信域看成一个类，面向对象。

➤ C/C++

```
int MPI_Send(void*buf,int count,MPI_Datatype dp,  
             int dest,int tag, MPI_Comm comm);
```

➤ Fortran

```
integer count,dp,dest,tag,comm,ierror  
call MPI_Send(buf,count,dp,dest,tag,comm,ierror)
```

➤ C++

```
void MPI::Comm::Send(const void* buf, int count,  
                     const MPI::Datatype& datatype,  
                     int dest, int tag) const;
```

```
eg. MPI::COMM_WORLD.Send(buf,2,MPI::CHAR,0,1);
```

7. 消息接收

➤ C/C++

```
int MPI_Recv(void *buf,int count,MPI_Datatype dp,  
             int source,int tag,MPI_Comm comm,MPI_Status  
             *status);
```

➤ Fortran

```
integer count,dp,source,tag,comm,sts(status_size),err  
call MPI_Recv(buf,count,dp,source,tag,comm,sts,err)
```

➤ C++

```
void MPI::Comm::Recv(void* buf, int count, MPI::Datatype&  
datatype,  
int source, int tag, MPI::Status& status) const  
eg. MPI::COMM_WORLD.Recv(&myval, 1, MPI::INT, my_rank-1, 0,  
&status);
```



消息

- 消息构成
- 消息标签
- 消息数据类型
 - 预定义数据类型
 - 新加的预定义数据类型 (Packed)
 - 派生数据类型
- 消息状态
 - 返回上级

消息构成

- 消息缓冲

- <起始地址, 数据个数, 数据类型>

- 消息地址

- <源/目标地址, 消息标签, 通信域>

MPI_Send(buf,count,datatype,dest,tag,comm)

- 消息标签

- 可以往同一个地址发送多封信, 这时候就要由tag来区分

- 常用的预定义常量: MPI_ANY_SOURCE,
MPI_ANY_TAG,

消息数据类型

- 消息传递时的两个问题
 - 异构性问题
 - 不同节点使用不同处理器和操作系统，如何保证通信双方的互操作性
 - 数据不连续问题
- MPI的消息数据类型
 - 预定义数据类型
 - 因为消息传递时应用的是MPI自己的类型，这个是相同的。因而支持异构计算。
 - 派生数据类型
 - 提供了强大的派生数据构造函数，跟预定义数据类型中的MPI_Pack类型，共同可解决消息数据的不连续问题。

MPI预定义数据类型(C)

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	Signed log 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	
MPI_PACKED	

MPI预定义数据类型(Fortran)

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	

MPI预定义数据类型(C++)

MPI datatype	C datatype	C++ datatype
MPI::CHAR	char	char
MPI::WCHAR	wchar_t	wchar_t
MPI::SHORT	signed short	signed short
MPI::INT	signed int	signed int
MPI::LONG	signed long	signed long
MPI::SIGNED_CHAR	signed char	signed char
MPI::UNSIGNED_CHAR	unsigned char	unsigned char
MPI::UNSIGNED_SHORT	unsigned short	unsigned short
MPI::UNSIGNED	unsigned int	unsigned int
MPI::UNSIGNED_LONG	unsigned long	unsigned long int
MPI::FLOAT	float	float
MPI::DOUBLE	double	double
MPI::LONG_DOUBLE	long double	long double
MPI::BOOL		bool
MPI::COMPLEX		Complex<float>
MPI::DOUBLE_COMPLEX		Complex<double>
MPI::LONG_DOUBLE_COMPLEX		Complex<long double>
MPI::BYTE		
MPI::PACKED		

消息状态

- 可以看到在MPI的接收函数中，多了消息状态参数
 - C实现中它是一个结构体
 - `status.MPI_SOURCE`
 - `status.MPI_TAG`
 - Fortran实现中是一个大小为`MPI_STATUS_SIZE`的整数数组
 - `status(MPI_SOURCE)`
 - `status(MPI_TAG)`
 - C++实现中是一个类
 - `status.Get_source();`
 - `status.Get_tag();`
 - 接收到的数据数量都通过`MPI_Get_count`函数获得。

接收数据的数量

- 接收到的数据数量，可能并没有填满buffer。我们来确定实际收到的数量

➤ C/C++

```
int MPI_Get_count(MPI_Status * stat,  
                  MPI_Datatype datatype, int * count);
```

➤ Fortran

```
integer status(MPI_Status_size),datatype  
integer count, ierror  
call MPI_Get_count(status,datatype,count,ierror)
```

➤ C++

```
int MPI::Status::Get_count(  
                           const MPI::Datatype& datatype) const  
eg. MPI::Status stat  
    count=staus.Get_count(MPI::INT);
```

如何使用打包数据类型

1. 确定要打包的数据总共占用多少空间
 - 调用MPI函数
 - 注意单位(Bytes)
2. 开辟所需要数量的空间，作为缓冲区
3. 将这些数据打包，存储到开辟的空间中。
 - 调用MPI函数
4. 发送，此时选用的消息数据类型是
`MPI_PACKED || MPI::PACKED`

如何使用打包数据类型

1. 确定要打包的数据总共占用多少空间

➤ C/C++

```
int MPI_Pack_size(int n, MPI_Datatype dt, MPI_Comm comm, int * size);
```

➤ Fortran

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

➤ C++

```
int MPI::Datatype::Pack_size(int incount, const MPI::Comm& comm) const;
```

eg. `int`

```
size=MPI::DOUBLE.Pack_size(50,MPI::COMM_WORLD)
```


如何使用打包数据类型

3. 将数据打包

➤ C/C++

```
int MPI_Pack(void * inbuf,int incount,MPI_Datatype dt,
void * outbuf, int outsz, int * position,MPI_Comm comm);
```

➤ Fortran

```
MPI_PACK(CHOICE INBUF,INTEGER INCOUNT,INTEGER
DATATYPE,CHOICE OUTBUF, INTEGER OUTSIZE,INTEGER
POSITION,INTEGER COMM,INTEGER IERROR)
```

➤ C++

```
void MPI::Datatype::Pack(const void* inbuf, int
incount, void* outbuf, int outsize, int& position, const
MPI::Comm& comm) const;
eg. MPI::DOUBLE.Pack(a+i,1,b,50,position,MPI::COMM_WORLD);
double a[100],b[50];
```

打包函数的说明

- 这个函数打包由 $\langle \text{inbuf}, \text{incount}, \text{datatype} \rangle$ (消息缓冲3元体)指定的消息到由 $\langle \text{outbuf}, \text{outsize} \rangle$ 指定的空间。输入缓冲可以是任意可用的通信缓冲，输出缓冲是任意由 outbuf 开始， outsize 个字节的连续的空间
- 下面看一个实例

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "mpi.h"
using namespace std;

const int NUM=100;

int main(int argc, char *argv[])
{
    //begin the main function
    double A[NUM];
    int size,rank;
    int n;
    double * Tempbuffer;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    MPI_Pack_size(NUM/2,MPI_DOUBLE,MPI_COMM_WORLD,&n); //the return value n is measured by byte
    Tempbuffer=new double[n/sizeof(double)];
    //printf("%2d: n=%d. It's measured by bytes. Equals to %d double space.\n",rank,n,n/sizeof(double));
```

Program

打包示例 -2/2

```
switch (rank) {
case 0: { //If there 's some declaration in the switch construction,
        //you should add curly bracket to delimit the scope of the declaration
        printf("Rank %d\n",rank);

        srand((int)time(NULL));

        for(int i=0;i<NUM;i++){    A[i]=(rand()%100)/10.0;
                                    cout<<A[i]<<" ";

        }
        cout<<endl;
        int position=0;
        for (int i=0;i<NUM/2;i++)
MPI_Pack(A+i*2,1,MPI_DOUBLE,Tempbuffer,n,&position,MPI_COMM_WORLD);
        printf("position=%d.It's measured by bytes,too. Equals to %d double space.\n“
                                   ,position,position/sizeof(double));
        MPI_Send(Tempbuffer,position,MPI_PACKED,1,1,MPI_COMM_WORLD);
        cout<<"Success Send!"<<endl;
        } break;
case 1:  MPI_Recv(Tempbuffer,n,MPI_PACKED,0,1,MPI_COMM_WORLD, & status);
        printf("Rank %d\n",rank);
        for (int i=0;i<n/sizeof(double);i++)
            cout<<Tempbuffer[i]<<" ";
        cout<<endl;

    }
MPI_Finalize();
return 0;
} Maths. S.D.U.
```

```
C:\Windows\System32\cmd.exe

Rank 0
0.8 7.2 6.6 0.3 6.7 5.2 0.6 4 6 2.4 9.3 4.5 3.4 9.2 2.6 9.1 5.3
2.3 2.2 8.6 6.1 9.9 6.5 4.6 9.7 9 7.5 8.9 0 4.8 0.1 8.9 5.7 5
0 9.5 2 0.3 8.5 6 4.1 8.5 2.6 8.3 8.7 1.4 6.4 7.2 2.7 9.6 9.3 7
.2 9.8 4.9 7.6 2.8 5.1 1.5 0.6 6.6 2.2 4.2 9.6 4.1 1.8 2.3 5.8 2
.8 2.2 9.4 3 0.7 4.6 9.1 3.3 1.3 8.6 3.8 8.1 7.7 1.2 8.3 9.6 8.6
1 3.1 3.5 3.3 1 2.4 5.1 3.4 2.1 8.6 5.6 3.7 6.1 7.1 2.8 3.1
position=400.It's measured by bytes,too. Equals to 50 double space.
Success Send!
Rank 1
0.8 6.6 6.7 0.6 6 9.3 3.4 2.6 5.3 2.2 6.1 6.5 9.7 7.5 0 0.1 5.7
0 2 8.5 4.1 2.6 8.7 6.4 2.7 9.3 9.8 7.6 5.1 0.6 2.2 9.6 1.8 5.8
2.2 3 4.6 3.3 8.6 8.1 1.2 9.6 1 3.5 1 5.1 2.1 5.6 6.1 2.8
请按任意键继续. . .
```

ISOLING, I do

派生数据类型

- MPI提供了几种方法来构造派生数据类型
 - contiguous
 - vector
 - indexed
 - struct
- 定义完一种类型后要将这种类型提交，才可以使用，如果不用的话可以释放
 - MPI_Type_commit
 - MPI_Type_free

➤ MPI Type contiguous

- 最简单的构造函数，构造count个oldtype的连续数据作为新的newtype返回。
MPI_Type_contiguous (count,oldtype,&newtype)
MPI_TYPE_CONTIGUOUS (count,oldtype,newtype,ierr)

➤ MPI Type vector MPI Type hvector

- 跟contiguous类似，但是允许想同的偏移量。由count个块构成，每个块包含blocklength个类型位oldtype的数据。相邻两块间相距stride个oldtype的空间。
MPI_Type_hvector 跟MPI_Type_vector 相同，除了stride是以byte计算的。
MPI_Type_vector (count,blocklength,stride,oldtype,&newtype)
MPI_TYPE_VECTOR (count,blocklength,stride,oldtype,newtype,ierr)

➤ MPI Type indexed MPI Type hindexed

- 功能和区别同上。所不同的是每个块的数量和相邻两块的偏移量由数组给出。
数组大小跟count一样。
MPI_Type_indexed (count,blocklens[],offsets[],old_type,&newtype)
MPI_TYPE_INDEXED (count,blocklens(),offsets(),old_type,newtype,ierr)

➤ MPI Type struct

- 连每个块的消息数据类型也可以不一样，由数组给出。

`MPI_Type_struct (count, blocklens[], offsets[], old_types(), &newtype)`

`MPI_Type_struct (count, blocklens(), offsets(), old_types(), newtype, ierr)`

➤ MPI Type extent

- 返回指定数据类型占用空间的大小，byte为单位。类似于C++中的sizeof

`MPI_Type_extent (datatype &extent)`

`MPI_Type_extent (datatype extent, ierr)`

➤ MPI Type commit

- 把新的数据类型提交给系统。任何用户数据类型（派生数据类型）都必须得提交后才能使用。

`MPI_Type_commit (&datatype)`

`MPI_Type_commit (datatype, ierr)`

➤ MPI Type free

- 释放指定的数据类型。以释放空间。

`MPI_Type_free (&datatype)`

`MPI_Type_free (datatype, ierr)`

C++实现的派生数据类型构造方法

- 因为在C++实现中，所有的数据类型均是一种Datatype类的对象。所有的构造函数均该类中的一种方法。由旧数据类型派生一种新的数据类型，就是该旧数据类型对象调用相应的构造函数。
 - using namespace MPI;
 - virtual Datatype Create_contiguous(int count) const
 - virtual Datatype Create_vector(int count, int blocklength, int tride) const
 - virtual Datatype Create_indexed(int count, const int * blocklength, const int * tride) const
 - static Datatype Create_struct(int v1, int v2[], MPI_Aint v3[], const Datatype v4[])
 - virtual int Get_size(void) const
 - virtual void Commit(void)
 - virtual void Free(void)

C++ 派生例子

```
MPI::Datatype newtype[4];
newtype[0]=MPI::DOUBLE.Create_contiguous(50);//创建包含连续50个double
//类型的新类型
newtype[1]=MPI::INT.Create_vector(50,2,3);//创建包含50个块，每个块由2个
连续INT型数据组成，相邻两块的起始位置相隔3个int型数据，因而实际
相隔3-2=1个数据
int a[3]={1,2,3},b[3]={2,3,4};
Datatype c[3]={MPI::DOUBLE,MPI::FLOAT,MPI::COMPLEX};
newtype[2]=MPI::CHAR.Create_indexed(3,a,b);
newtype[3]=Datatype::Create_struct(3,a,b,c);

for(int i=0;i<4;i++) newtype[i].Commit();
```

C++封装

- 其实C++不过是做了个封装，使其面向对象，方便编程。
- 下面的代码是从头文件mpicxx.h中的datatype类中复制来的一段代码。从中可以看出，C++封装中的实现借用了经典C实现中的代码。

```
virtual Datatype Create_vector( int v1, int v2, int v3 ) const {  
    Datatype v5;  
    MPIX_CALL( MPI_Type_vector( v1, v2, v3, (MPI_Datatype)  
the_real_datatype, &(v5.the_real_datatype) ));  
    return v5; }
```

Program

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <mpi.h>
```

取奇数位的类型——派生实现

```
using namespace std;
```

```
const int num=100;
```

```
int main(int argc, char *argv[])
{
```

```
    //begin the main function
```

```
    //double A[num];
```

```
    double *A=new double[num];
```

```
    int rank,size;
```

```
    MPI::Init();
```

```
    rank=MPI::COMM_WORLD.Get_rank();
```

```
    //define new datatype
```

```
    MPI::Datatype Odd;
```

```
    Odd=MPI::DOUBLE.Create_vector(num/2,1,2);
```

```
    Odd.Commit();
```

Program

```

switch(rank) { case 0: { srand((int)time(NULL));
                        cout<<"Rank 0\n";
                        for (int i=0;i<num;i++) {
                                A[i]=rand()%100/10.0;
                                cout<<A[i]<<" ";
                                }
                        cout<<endl;

                        MPI::COMM_WORLD.Send(A,1,Odd,1,1);
                        MPI::COMM_WORLD.Send(A,1,Odd,1,2);

}break;
case 1: { MPI::Status stat;

          MPI::COMM_WORLD.Recv(A,1,Odd,0,1);
          cout<<"\n*****\nRank 1, tag 1\n";
          for(int i=0;i<num;i++)
                  cout<<A[i]<<" ";
          cout<<endl;

          double * B=new double[num];
          MPI::COMM_WORLD.Recv(B,num/2,MPI::DOUBLE,0,2,stat);
          cout<<"\n*****\nRank 1, tag 2. \n";
          for(int i=0;i<num;i++)
                  cout<<B[i]<<" ";
          cout<<endl;

}break;
}

MPI::Finalize();
return 0;
}

```

KONG, Tao

Maths. S.D.U.

C:\Windows\System32\cmd.exe

Rank 0

4.5	9.9	4.6	0.9	6.3	8.9	9.8	0.8	1.3	5.3	0.3	2.5	4.4	6.
3	5.3	6.4	7.6	9.4	1.7	0.5	0	3.8	4.4	5.9	2.8	6.4	6.8
7.8	1.6	0.8	0.4	2.5	9.5	1.8	5.2	3.1	0.2	4.2	0.6	8.2	5
	8.8	1.9	4	8.7	6	4.1	7.4	3.4	6.5	4.4	1	9	0
	7	9.1	2.9	2.5	0.6	3.5	9.2	0.7	3.6	4.3	2.6	1.3	0.8
2	2.9	3	7.1	3.9	6.7	5.8	8.7	5.1	5.5	9.5	0.7	1.4	5
	7.8	5.3	4.5	0.5	6.9	9.1	1.4	3.9	5.8	8.7	7	3.3	9.4
4	3.4	4.7											

Rank 1, tag 1

4.5	0	4.6	0	6.3	0	9.8	0	1.3	0	0.3	0	4.4	0	5.3	0
7.6	0	1.7	0	0	0	4.4	0	2.8	0	6.8	0	1.6	0	0.4	0
5	0	5.2	0	0.2	0	0.6	0	5	0	1.9	0	8.7	0	4.1	0
	0	4.4	0	9	0	5.1	0	7	0	2.9	0	0.6	0	9.2	0
	2.6	0	0.8	0	2.9	0	7.1	0	6.7	0	8.7	0	5.5	0	0.7
	5	0	7.8	0	4.5	0	6.9	0	1.4	0	5.8	0	7	0	9.4
4	0														

Rank 1, tag 2

4.5	4.6	6.3	9.8	1.3	0.3	4.4	5.3	7.6	1.7	0	4.4	2.8	6.8
	1.6	0.4	9.5	5.2	0.2	0.6	5	1.9	8.7	4.1	3.4	4.4	9
	7	2.9	0.6	9.2	3.6	2.6	0.8	2.9	7.1	6.7	8.7	5.5	0.7
	7.8	4.5	6.9	1.4	5.8	7	9.4	3.4	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0												

请按任意键继续. . .

通信域

- 进程组
- 通信上下文
 - 每个通信域都有唯一的一个通信上下文，每个通信上下文唯一的识别一个通信域
- 分为组内通信和组间通信
 - 一般只用到组内通信
- MPI包括几个预定义的通信域
 - MPI_COMM_WORLD: 所有进程的集合，在执行MPI_Init之后产生
 - MPI_COMM_SELF: 只包含使用它的进程

通信域

- 对通信域的操作

函数名	含义
MPI_Comm_size	
MPI_Comm_rank	
MPI_Comm_compare	
MPI_Comm_dup	
MPI_Comm_create	
MPI_Comm_split	
MPI_Comm_free	

简单功能实现

- 每个进程把自己的进程号传给0进程，0进程按接收到的先后顺序将他们存到自己的内存空间中
- 当接收完所有的进程的消息后，把收到的消息输出。

Program

```
#include <iostream>
#include <cstring>
#include <mpi.h>
using namespace std;
int main(int argc, char *argv[])
{
```

ranknum (C++)

```
    static char buf1[2],buf2[2],result[20];
    int myrank,size;
    MPI_Status status;
    cout<<"Now begin to start MPI"<<endl;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, & size);
    MPI_Comm_rank(MPI_COMM_WORLD, & myrank);
    //cout<<"This is a message from process "<<myrank<<" of "<<size<<" processes!"<<endl;
    printf("This is a message from process %2d of %2d processes\n",myrank,size);

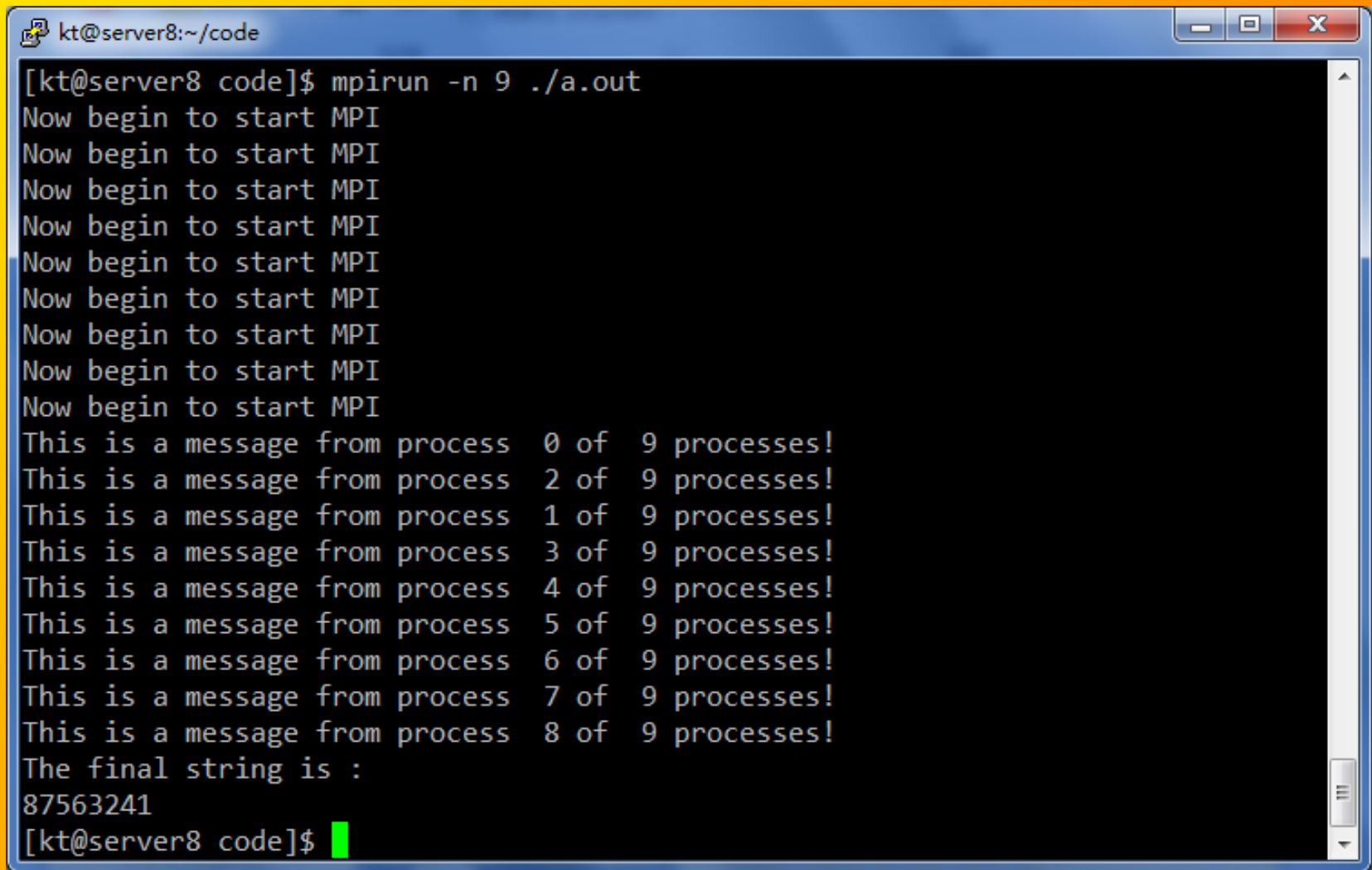
    int ii=size;
    switch (myrank){
    case 0:      while(--ii){
                  MPI_Recv( buf1,1,MPI_CHAR,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,& status);
                  strcat(result,buf1);
                }
                cout<<"The final string is :\n"<<result<<endl;
                break;
    default:    buf2[0]=char(myrank+48);
                MPI_Send( buf2,1,MPI_CHAR,0,1,MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

KONG, Tao

Maths. S.D.U.

运行结果

A terminal window titled 'kt@server8:~/code' with standard window controls. The terminal shows the execution of 'mpirun -n 9 ./a.out'. It displays nine lines of 'Now begin to start MPI', followed by nine lines of 'This is a message from process X of 9 processes!' where X ranges from 0 to 8. Then it shows 'The final string is : 87563241' and ends with a prompt '[kt@server8 code]\$' and a green cursor.

```
kt@server8:~/code
[kt@server8 code]$ mpirun -n 9 ./a.out
Now begin to start MPI
Now begin to start MPI
Now begin to start MPI
Now begin to start MPI
Now begin to start MPI
Now begin to start MPI
Now begin to start MPI
Now begin to start MPI
Now begin to start MPI
This is a message from process 0 of 9 processes!
This is a message from process 2 of 9 processes!
This is a message from process 1 of 9 processes!
This is a message from process 3 of 9 processes!
This is a message from process 4 of 9 processes!
This is a message from process 5 of 9 processes!
This is a message from process 6 of 9 processes!
This is a message from process 7 of 9 processes!
This is a message from process 8 of 9 processes!
The final string is :
87563241
[kt@server8 code]$
```

KONG, Tao

Maths. S.D.U.

点对点通信(P2P Communication)

- MPI通信模式

- 指的是缓冲管理以及发送方和接受方的同步方式
 - 标准
 - 缓冲
 - 同步
 - 就绪

- 阻塞和非阻塞通信

- 通信完成的标志: *消息已经成功发出或接受*
- 成功是指
 - 若是发送, 则发送缓冲区可以被更新
 - 若是接收, 则接收缓冲区可以使用
- 根据是否等到通信完成后才返回, 分为阻塞通信和非阻塞通信
 - 前者成功后才返回, 后则无论是否成功均返回

- MPI发送接收类别

- 发送支持4种通信模式, 与阻塞共同组成8中发送操作
- 而接收只有两种: 阻塞和非阻塞

Mode	Completion Condition
Synchronous send	Only completes when the receive has completed
Buffered send	Always completes (unless an error occurs), irrespective of receiver
Standard send	Message sent (receive state unknown)
Ready send	Always completes (unless an error occurs), irrespective of whether the receive has completed
Receive	Completes when a message has arrived

阻塞通信

MODE	MPI CALL
Standard send	MPI_SEND
Synchronous send	MPI_SSEND
Buffered send	MPI_BSEND
Ready send	MPI_RSEND
Receive	MPI_RECV

非阻塞通信

Non-Blocking Operation	MPI Call
Standard send	MPI_ISEND
Synchronous send	MPI_ISSEND
Buffered send	MPI_IBSEND
Ready send	MPI_IRSEND
Receive	MPI_IRECV

标准非阻塞通信

➤ C/C++

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

➤ Fortran:

```
call MPI_ISEND(CHOICE BUF,INTEGER COUNT,INTEGER  
DATATYPE,INTEGER DEST, INTEGER TAG,INTEGER COMM,INTEGER  
REQUEST,INTEGER IERROR)
```

➤ C++:

```
MPI::Request MPI::Comm::Isend(const void *buf, int count,  
const MPI::Datatype& datatype, int dest, int tag) const;  
eg. MPI::Request rq=MPI::COMM_WORLD.Isend(a,50,MPI::CHAR,0,1)  
rq.Wait();
```


非阻塞通信的完成测试

- 等待还是测试

- 等待-这是阻塞的，等到request所指的操作完成

- C/C++: `int MPI_Wait(MPI_Request *request, MPI_Status *status);`
 - Fortran: `integer :: request, ierror`
`integer, dimension(MPI_STATUS_SIZE) :: status`
`call MPI_WAIT(request, status, ierror)`
 - C++: `void MPI::Request::Wait(MPI::Status & status);`
`void MPI::Request::Wait();`

- 测试—这是非阻塞的，返回true or false

- C/C++: `int MPI_Test(MPI_Request * request, int * flag, MPI_Status * status);`
 - Fortran: `integer :: request, ierror, flag`
`integer, dimension(MPI_STATUS_SIZE) :: status`
`call MPI_WAIT(request, flag, status, ierror)`
 - C++: `bool MPI::Request::Test(MPI::Status & status);`
`bool MPI::Request::Test();`

避免死锁

通信和计算的重叠

群集通信(Collective Communication)

- 特点
- 汇总
- 详解

特点

- 通信域中的所有进程都必须调用群集通信函数
- 除MPI_Barrier外，所有群集通信都相当于标准阻塞的通信模式
 - 因此一个进程一旦结束了它所参与的群集操作就返回，并不保证其他进程执行该群集操作已经完成。
- 所有参与群集操作的进程中，count和Datatype必须是兼容的。
- 群集通信中的消息没有标签，消息标签由通信域和源/目标进程定义。
 - 所有这些保证了在C++实现中，所有群集函数均作为通信域这个类的方法。

群集通信汇总

类型	函数名	含义
通信	MPI_Bcast	一对多广播同样的消息
	MPI_Gather	多对一收集各进程信息
	MPI_Gatherv	一般化
	MPI_Allgather	全局收集，每个进程执行Gather
	MPI_Allgatherv	一般化
	MPI_Scatter	一对多散播不同消息
	MPI_Scatterv	一般化
	MPI_Alltoall	多对多全局交换，每个进程执行Scatter
	MPI_Alltoallv	一般化
聚集	MPI_Reduce	多对以归约
	MPI_Allreduce	一般化
	MPI_Reduce_scatter	归约并散播
	MPI_Scan	扫描。每个进程对自己前面的进程归约
同步	MPI_Barrier	路障同步

注：红色的为常用的，将详细介绍

C++的群集通信汇总

- 群集通信属于一个通信域所有。而在C++实现中，通信域是一个类，因此所有的群集通信函数，均是这个类中的成员函数。
-

- `void MPI::Comm::Bcast(
 void* buff, int count, const MPI::Datatype& datatype, int root)
const=0`
- `void MPI::Comm::Gather(
 const void* sendbuf, int sendcount, const MPI::Datatype& sendtype,
 void* recvbuf, int recvcount, const MPI::Datatype& recvttype,
 int root) const`
- `void MPI::Comm::Scatter(
 const void* sendbuf, int sendcount, MPI::Datatype& sendtype,
 void* recvbuf, int recvcount, MPI::Datatype& recvttype,
 int root) const`
- `void MPI::Comm::Allgather(
 const void* sendbuf, int sendcount, const MPI::Datatype& sendtype,
 void* recvbuf, int recvcount, const MPI::Datatype& recvttype)
const`
- `void MPI::Comm::Alltoall(
 const void* sendbuf, int sendcount, const MPI::Datatype& sendtype,
 void* recvbuf, int recvcount, const MPI::Datatype& recvttype)
const`
- `void MPI::Comm::Reduce(const void* sendbuf, void* recvbuf,
 int count, const MPI::Datatype& datatype,
 const MPI::Op& op, int root) const=0`
- `void MPI::Comm::Scan(const void* sendbuf, void* recvbuf,
 int count, const MPI::Datatype& datatype,
 const MPI::Op& op) const=0`
- `void MPI::Comm::Barrier() const=0`

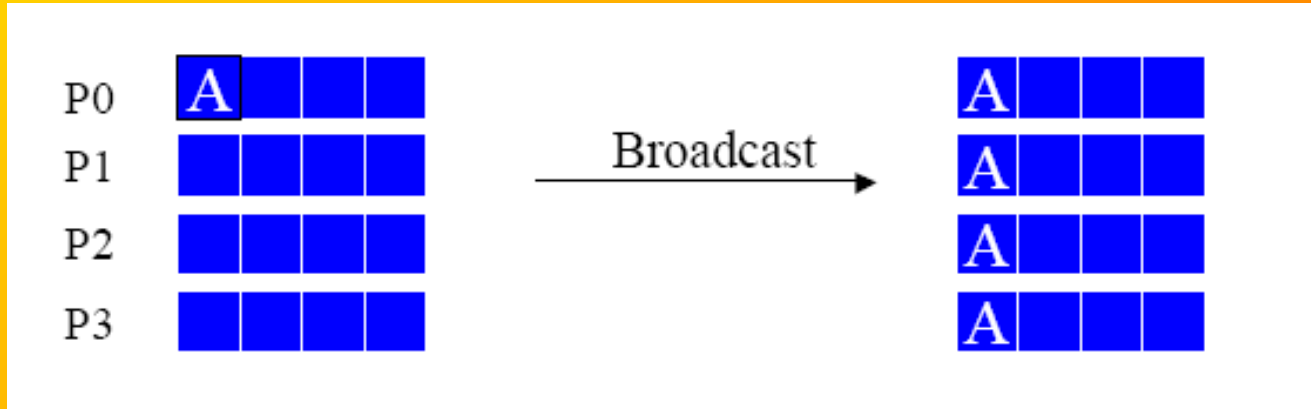
C++的群集通信汇总(示例)

群集通信的通信功能

- 多对一
 - 一对多
 - 多对多
-
- 除了群集通信必须要指明通信域外，前两者还需指明root进程，多对多只需指明通信域。

1. 广播(MPI_Bcast)

- 一对多，消息相同。有一个root进程，由它向所有进程发送消息。对于root本身，缓冲区即是发送缓冲又是接收缓冲。



➤ C/C++

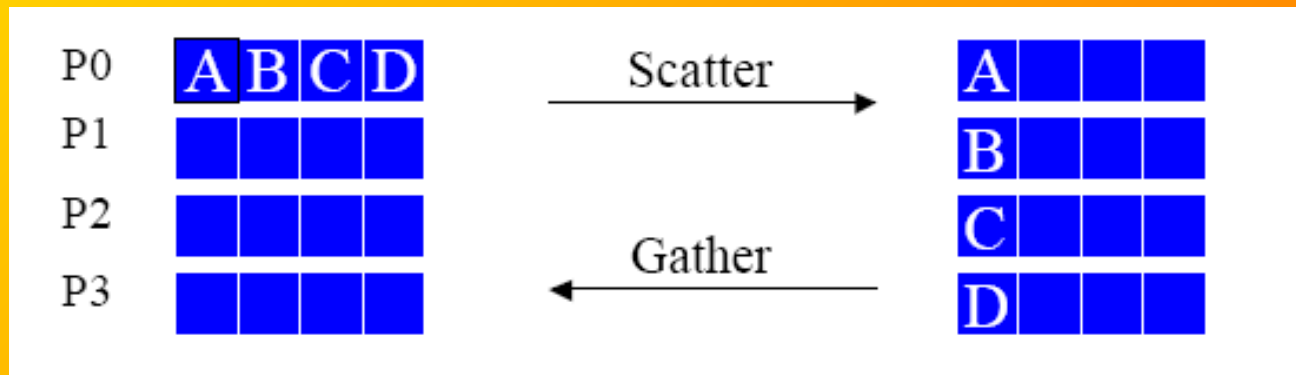
```
int MPI_Bcast(void* Addr, int count, MPI_Datatype dp,  
              int root, MPI_Comm comm);
```

➤ Fortran

```
<type> buff(*)  
integer count, dp, root, comm, ierror  
call MPI_Bcast(buff, count, dp, root, comm, ierror)
```

2. 收集 (MPI_Gather)

- 多对一，消息相同。有一个root进程用来接收，其他包含root发送。这n个消息按进程的标号进行拼接。所有非root进程忽略接收缓冲。



➤ C/C++

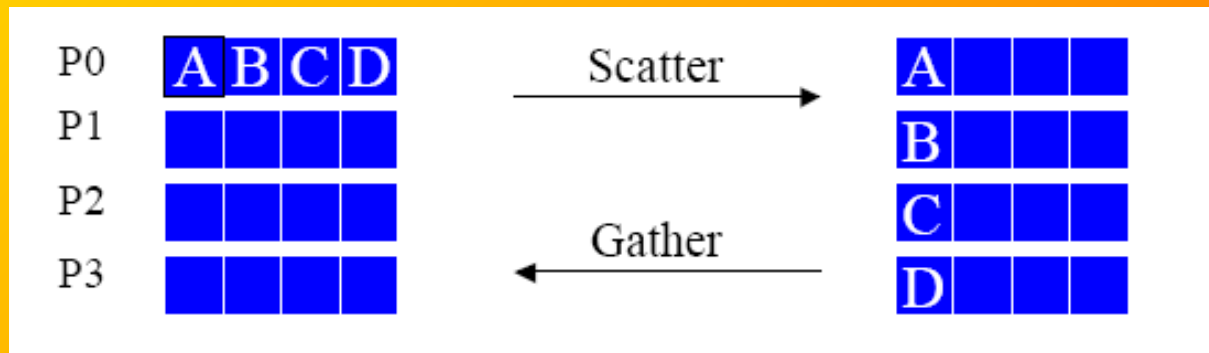
```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype  
senddp, void* recvbuff, int recvcount, MPI_Datatype senddp, int root,  
MPI_Comm comm);
```

➤ Fortran

```
<type> sendbuff(*) recvbuf(*)  
integer sendcount, sendtype, recvcount, recvtpe, root, comm, ierror  
call MPI_Gather(sendbuff, sendcount, sendtype,  
                & recvbuff, recvcount, recvtpe, root, comm, ierror)
```

3. 散播 (MPI_Scatter)

- 一对多，消息不同，顺序存储。有一个root进程，将发送缓冲中的数据按进程编号，有序的发送。非root进程忽略发送缓冲。



➤ C/C++

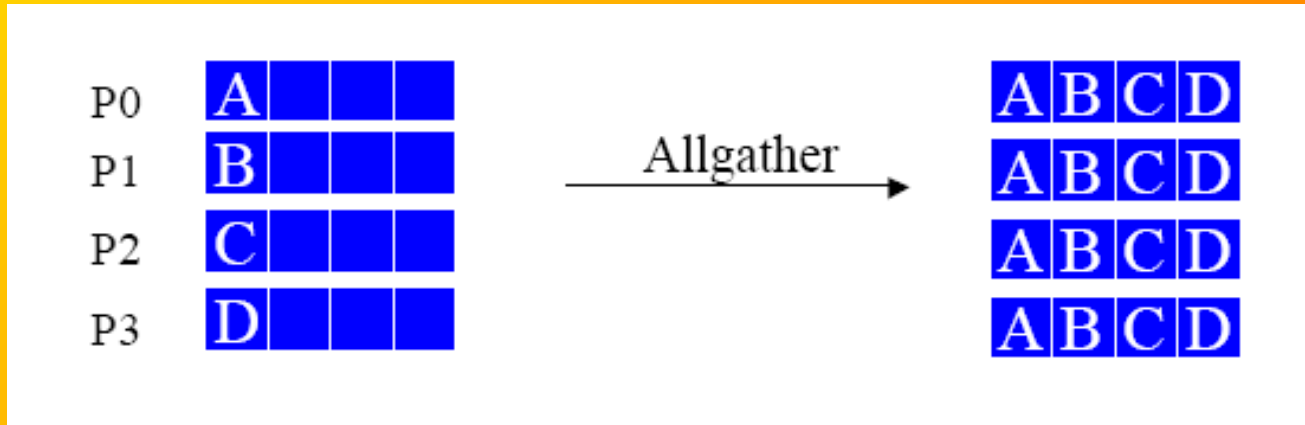
```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype  
senddatatype, void *recvbuf, int recvcount, MPI_Datatype  
recvdatatype, int root, MPI_Comm comm);
```

➤ Fortran

```
<type> sendbuff(*) recvbuf(*)  
integer sendcount, sendtype, recvcount, recvtpe, root, comm, ierror  
call MPI_Scatter(sendbuff, sendcount, sendtype,  
& recvbuff, recvcount, recvtpe, root, comm, ierror)
```

4. 全局收集 (MPI_Allgather)

- 相当于每个进程都作为root进程执行了一次Gather操作。属于整个通信域，因此只需指明通信域，不需要指定root。



➤ C/C++

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype  
senddatatype, void *recvbuf, int recvcount, MPI_Datatype  
recvdatatype, MPI_Comm comm);
```

➤ Fortran

```
<type> sendbuff(*) recvbuf(*)  
integer sendcount, sendtype, recvcount, recvtpe, comm, ierror  
call MPI_Allgather(sendbuff, sendcount, sendtype,  
    & recvbuff, recvcount, recvtpe, comm, ierror)
```

5. 全局交换(MPI_Alltoall)

- 相当于每个进程作为root进程执行了一次Scatter操作。属于整个通信域，因此只需指明通信域，不需要指定root。



➤ C/C++

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype  
senddatatype, void *recvbuf, int recvcount, MPI_Datatype  
recvdatatype, MPI_Comm comm);
```

➤ Fortran

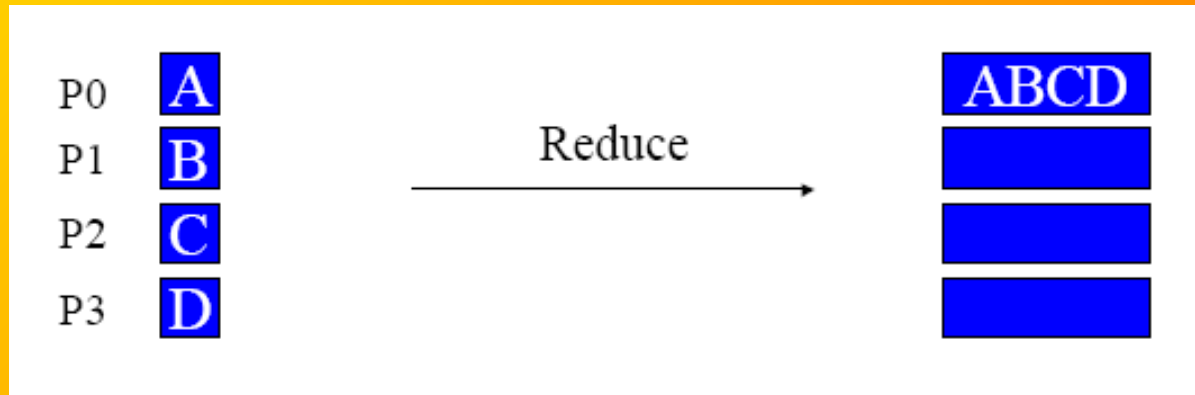
```
<type> sendbuff(*) recvbuff(*)  
integer sendcount, sendtype, recvcount, recvtpe, comm, ierror  
call MPI_Alltoall(sendbuff, sendcount, sendtype,  
                  & recvbuff, recvcount, recvtpe, comm, ierror)
```

群集通信的聚合功能

- 聚合功能，使得各进程间在进行通信的同时还完成一定的计算。
- 分3步
 - 通信。消息根据要求发送到目标进程。
 - 处理。目标进程对接收到的消息进行处理。
 - 放入指定缓冲区。

1. 归约 (MPI_Reduce)

- 多对一，且完成计算。每个进程将发送缓冲区中数据发到root进程，root完成指定的操作，并将结果放到接收缓冲。



➤ C/C++

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

➤ Fortran

```
<type> sendbuf(*),recvbuf(*)
integer count,datatype,op,root,comm,ierror
call MPI_Reduce(sendbuf,recvbuf,count,datatype,op,
&               root,comm,ierror)
```


归约操作（OP）汇总

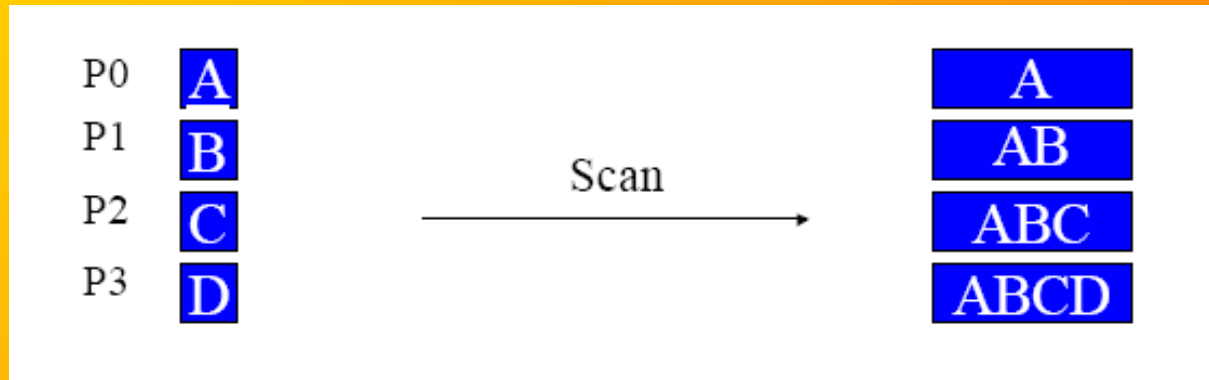
MPI归约操作		C 数据类型	Fortran数据类型
MPI_MAX	最大值	integer, float	integer, real, complex
MPI_MIN	最小值	integer, float	integer, real, complex
MPI_SUM	求和	integer, float	integer, real, complex
MPI_PROD	乘积	integer, float	integer, real, complex
MPI LAND	逻辑与	integer	logical
MPI_BAND	按位与	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LOR	逻辑或	integer	logical
MPI_BOR	按位或	integer, MPI_BYTE	integer, MPI_BYTE
MPI_LXOR	逻辑异或	integer	logical
MPI_BXOR	按位异或	integer, MPI_BYTE	integer, MPI_BYTE
MPI_MAXLOC	最大值且相应位置	float, double and long double	real, complex, double precision
MPI_MINLOC	最小值且相应位置	float, double and long double	real, complex, double precision

KONG, Tao

Maths. S.D.U.

2. 扫描 (MPI_Scan)

- 多对多。一种特殊的Reduce：每一个进程多作为root对排在他前面的进程执行归约操作。



➤ C/C++

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

➤ Fortran

```
<type> sendbuf(*), recvbuf(*)  
integer count, datatype, op, root, comm, ierror  
call MPI_Scan(sendbuf, recvbuf, count, datatype, op,  
              & root, comm, ierror)
```

群集通信的同步功能

- 路障 (MPI_Barrier)
- 为了协调各个进程间的进度和步伐。
- 保证通信域内，所有的进程都已经执行完了调用之前的所有操作。

➤ C/C++

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

➤ Fortran

```
integer :: comm, ierror  
call MPI_Barrier(comm, ierror)
```