MPI 并行编程基础

李会民
lihm@qibebt.ac.cn

中国科学院青岛生物能源与过程研究所 超级计算中心

2010 年 06 月

MPI 编程培训内容

并行计算简介

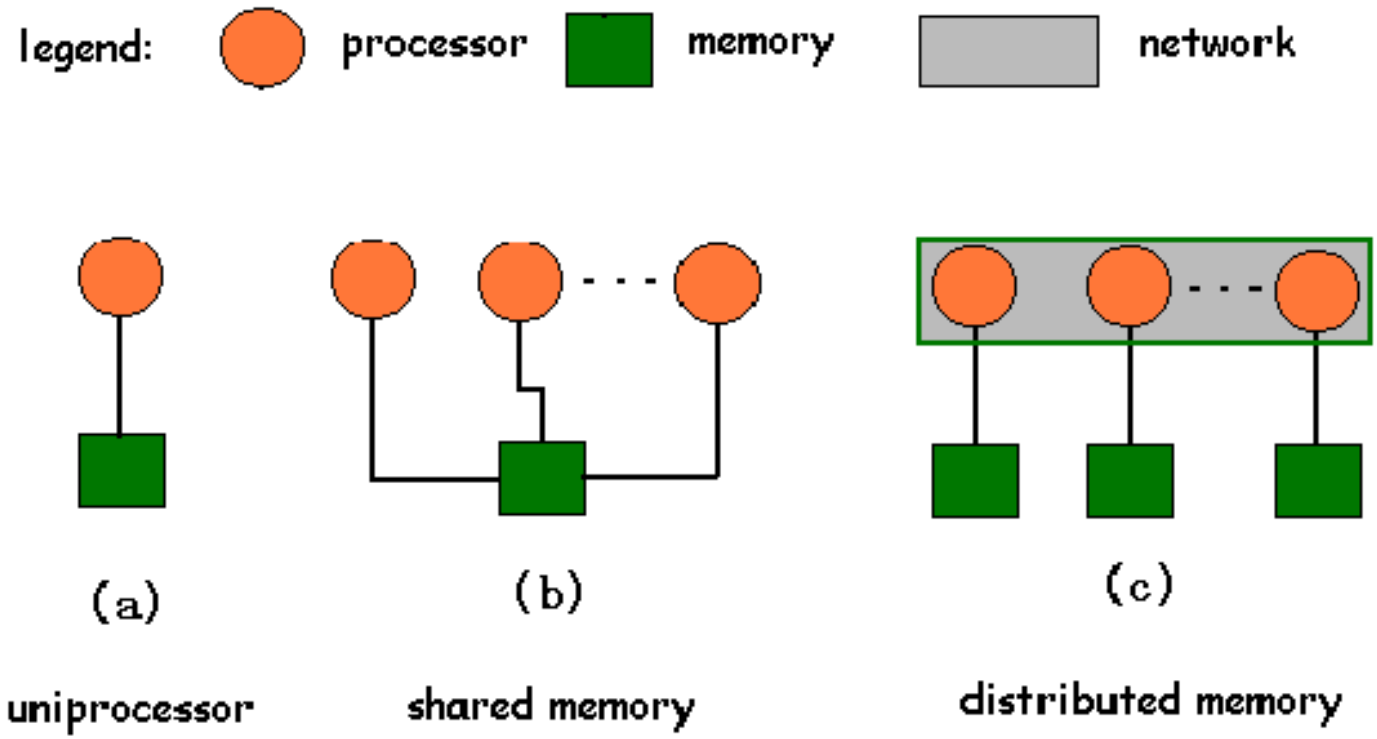
为什么要采用并行计算？

- 串行程序速度提升缓慢
- 可以加快速度
- 可以加大规模

并行计算设计的分类

- 共享内存：OpenMP
  - 多个处理器共享同一内存
  - ccNUMA(Cache-Coherent Non-Uniform Memory Access)、SMP(Symmetric MultiProcessing)
- 分布式内存：MPI
  - 每个处理器都有自己的内存
  - 处理器之间通过传递消息交换信息
  - Cluster、MPP(Massively Parallel Processing)

三种计算模型


legend:
processor
memory
network
(a)
(b)
(c)
uniprocessor
shared memory
distributed memory

并行化分解方法

- 任务分解：多任务并发执行
- 功能分解：分解被执行的计算
- 区域分解：分解被执行的数据

分布式内存并行方式

- 任务并行：不同参数的大量工况计算
- 区域分解并行：大规模多节点分块并行计算

MPI 简介

MPI(Message Passing Interface)是 1994 年 5 月发布的一种消息传递接口标准。它实际上是一个消息传递函数库的标准说明，以语言独立的形式来定义这个接口库，并提供了与 C/C++ 和 Fortran 语言的绑定。

主要站点：http://www.mpi-forum.org/

MPI 的历史

- MPI 初稿：美国并行计算中心工作会议（1992 年 4月）
- MPI-1 提出讨论：第一届 MPI 大会（1993 年 1 月）
- MPI-1.0 公布：MPI 论坛（1994 年 5 月）
- MPI-1.1 公布：MPI 论坛（1995 年 6 月）
- MPI-1.2 公布：MPI 论坛（1997 年 7 月）
- MPI-2 公布：MPI 论坛（1997 年 7 月）
- MPI-1.3 公布：MPI 论坛（2008 年 5 月）
- MPI-2.1 公布：MPI 论坛（2008 年 9 月）
- MPI-2.2 公布：MPI 论坛（2009 年 9 月）

MPI 的实现

- MPICH：最广泛的 MPI 实现
  http://www.mcs.anl.gov/research/projects/mpi/mpich1/
- MPICH2：MPICH的新分支
  http://www.mcs.anl.gov/research/projects/mpich2/
- MVAPICH 和 MVAPICH2：针对 InfiniBand 网络的 MPICH 版本
  http://mvapich.cse.ohio-state.edu/
- MPICH-MX 和MPICH-MX2：针对 Myrinet 网络的 MPICH 版本
  http://www.myri.com/scs/download-mpichmx.html
- Open-MPI：建立在 FT-MPI、LA-MPI、LAM/MPI 等之上
  http://www.open-mpi.org/
- Intel MPI：建立在 MPICH2 和 MVAPICH2 之上
  http://software.intel.com/en-us/intel-mpi-library/
- LAM(Local Area Multicomputer)：Ohio State University 开发
  http://www.lam-mpi.org

MPI 编程

MPI 为程序员提供一个并行环境库，程序员通过调用 MPI 的库函数来达到程序员所要的并行目的，只使用其中的六个最基本的函数就能编写一个完整的 MPI 程序去求解很多问题。这六个基本函数，包括启动和结束 MPI 环境，识别进程以及发送和接收消息：

MPI_INIT	启动 MPI 环境
MPI_COMM_SIZE	确定进程数
MPI_COMM_RANK	确定自己的进程标识符
MPI_SEND	发送一条消息
MPI_RECV	接收一条消息
MPI_FINALIZE	结束 MPI 环境

Fortran 版本简单例子

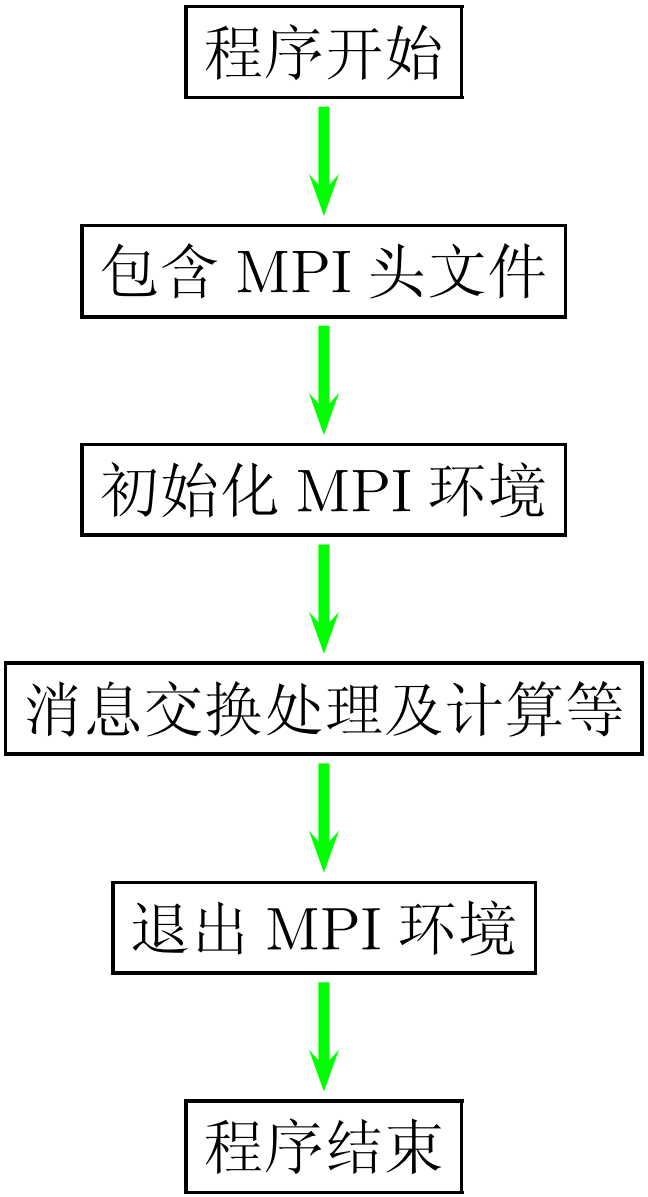
```
PROGRAM MAIN
USE MPI
CHARACTER * (MPI_MAX_PROCESSOR_NAME) ProcessorName
INTEGER MyID,NumProcs,NameLen,iErr

CALL MPI_INIT(iErr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,MyID,iErr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NumProcs,iErr)
CALL MPI_GET_PROCESSOR_NAME(ProcessoRname,NameLen,iErr)
WRITE (*,10) MyID,NumProcs,ProcessorName
10  FORMAT('Hello␣world,␣process', I2, '␣of␣', I1, '␣on␣', 20A)
CALL MPI_FINALIZE(iErr)
END
```

C 版本简单例子

```
#include "mpi.h"
main(int argc, char **argv)
{
    int NumProcs,MyID,i,j,k;
    MPI_Status status;
    char msg[20];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&NumProcs);
    MPI_Comm_rank(MPI_COMM_WORLD,&MyID);
    if(MyID == 0){
        strcpy(msg,"Hello World");
        MPI_Send(msg,strlen(msg) + 1,MPI_CHAR,1,99,MPI_COMM_WORLD);
    }else if(MyID ==1){
        MPI_Recv(msg,20,MPI_CHAR,0,99,MPI_COMM_WORLD,&status);
        printf("Receive message = %s\n",msg);
    }
    MPI_Finalize();
}
```

MPI 程序的一般结构



头文件

MPI 程序要求所有包含 MPI 调用的程序应加入 MPI 文件头：

- C：#include "mpi.h"
- Fortran：
  - F77：include 'mpif.h'
  - F90 及之后：use mpi（建议）或 include 'mpif.h'
  - Fortran 虽然不区分大小写，但 Linux 系统区分大小写，因此这里的 mpif.h 必须为小写，如果用 use mpi 则不存在此问题

通信子和组

- MPI 通过指定通信子（又称通信因子、通信域）和组来对进程进行一种逻辑上的划分，通信子定义了进程组内或组间通讯的上下文（具体就是指明通讯链路的数据结构指针）
- MPI_COMM_WORLD 通信子在 MPI 环境初始化过程中创建

进程号（rank）

在一个通信子中，每个进程都有一个唯一的整数标识符，称作“进程号”，进程号是从 0 开始的连续整数，用进程号可以控制程序的不同部分在不同的进程中并行运行

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYID, IERR)
IF (MYID == 0) THEN
    0 号进程运行的程序段
ELSE IF (MYID == 1) THEN
    1 号进程运行的程序段
ENDIF
```

MPI 消息

- MPI 消息包括信封和数据两部分，信封指出了发送或接收消息的对象及相关信息，数据是本消息将要传递的内容
- 数据：<起始地址，数据个数，数据类型>
- 信封：<源/目，标识，通信子>

隐含约定

- MPI 函数的命名规则
  - 函数名形式为 MPI_Class_action_subset、MPI_Class_action、MPI_Action_subset 或 MPI_Action
  - C 语言：MPI 和第一个_后的首字符为大写，其余为小写
  - Fortran 语言：函数名不区分大小写。Fortran 函数除了 MPI_Wtime、MPI_Wtick 外，比 C 函数多一个参数 IERROR
- 在 MPI 标准中 MPI 函数采用语言独立的方式说明，参数用 IN、OUT 或 INOUT 标记
  - IN：变量为输入给函数的，调用后其值不变
  - OUT：变量不是输入给函数的，其值被函数调用后更新
  - INOUT：变量既是输入给函数的，调用后也会被更新

MPI 的基本函数

利用以下六条基本函数即可完成 MPI 并行程序

- MPI_Init
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Send
- MPI_Recv
- MPI_Finalize

MPI_Init：初始化 MPI 环境

- 初始化 MPI 运行环境
- 必须调用；首先调用；调用一次
- 每个进程都有一个参数表

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

MPI_INIT( IERROR )
    INTEGER IERROR
```

注：

- 绿色表示 C 语法
- 蓝色表示 Fortran 语法，第一行为函数，后面的行为变量声明

MPI_Comm_size：取得进程数

返回与该组通信子相关的进程数，通信子必须是组内通信子

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

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

MPI_Comm_rank：取得进程号

返回该进程在指定通信子中的进程号（0～进程数-1），一个进程在不同通信子中的进程号可能不同

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

```
MPI_COMM_RANK( COMM, RANK, IERROR )
    INTEGER COMM, SIZE, IERROR
```

MPI_Send：发送消息

发送缓冲区中的 count 个 datatype 数据类型的数据发送到目的进程

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

IN buf 发送缓存的起始地址（选择型）
IN count 发送缓存的元素的个数（非负整数）
IN datatype 每个发送缓存元素的数据类型（句柄）
IN dest 目的地进程号（整型）
IN tag 消息标志（整型）
IN comm 通信子（句柄）

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

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

MPI_Recv：接收消息

从指定的进程 source 接收消息，收到的消息所包含的数据元素的个数最多不能超过 count

MPI_RECV(buf, count, datatype, source, tag, comm, status)

OUT buf 接收缓存的起始地址（选择型）
IN count 接收缓存中元素的个数（整型）
IN datatype 每个接收缓存元素的数据类型（句柄）
IN source 发送操作的进程号（整型），可为 MPI_ANY_SOURCE
IN tag 消息的标识（整型），可为 MPI_ANY_TAG
IN comm 通信组（句柄）
OUT status 状态对象（状态）

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

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type>BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

注意避免死锁

下面例子各进程都在等待对方进程先发来再发送，导致死锁

```
if (MyID == 0) then
  CALL MPI_RECV(MSG_1,11,MPI_CHARACTER,1,0,MPI_COMM_WORLD,STATUS,iErr)
  CALL MPI_SEND(MSG_0,11,MPI_CHARACTER,1,0,MPI_COMM_WORLD,iErr)
elseif (MyID == 1) then
  CALL MPI_RECV(MSG_1,11,MPI_CHARACTER,0,0,MPI_COMM_WORLD,STATUS,iErr)
  CALL MPI_SEND(MSG_0,11,MPI_CHARACTER,0,0,MPI_COMM_WORLD,iErr)
endif
```

下面两个例子不存在死锁

```
if (MyID == 0) then
  CALL MPI_SEND(MSG_0,11,MPI_CHARACTER,1,0,MPI_COMM_WORLD,iErr)
  CALL MPI_RECV(MSG_1,11,MPI_CHARACTER,1,0,MPI_COMM_WORLD,STATUS,iErr)
elseif (MyID == 1) then
  CALL MPI_SEND(MSG_0,11,MPI_CHARACTER,0,0,MPI_COMM_WORLD,iErr)
  CALL MPI_RECV(MSG_1,11,MPI_CHARACTER,0,0,MPI_COMM_WORLD,STATUS,iErr)
endif
```

```
if (MyID == 0) then
  CALL MPI_RECV(MSG_1,11,MPI_CHARACTER,1,0,MPI_COMM_WORLD,STATUS,iErr)
  CALL MPI_SEND(MSG_0,11,MPI_CHARACTER,1,0,MPI_COMM_WORLD,iErr)
elseif (MyID == 1) then
  CALL MPI_SEND(MSG_0,11,MPI_CHARACTER,0,0,MPI_COMM_WORLD,iErr)
  CALL MPI_RECV(MSG_1,11,MPI_CHARACTER,0,0,MPI_COMM_WORLD,STATUS,iErr)
endif
```

MPI_Finalize：结束 MPI 环境

结束 MPI 执行环境。该函数一旦被程序调用时，就不能调用 MPI 的其它例行函数（包括 MPI_Init），用户须保证在进程调用 MPI_Finalize 之前完成与进程有关的所有通信

```
int MPI_Finalize( void )

MPI_FINALIZE( IERROR )
    INTEGER IERROR
```

参数说明

- 缓冲区（buffer）
- 数据个数（count）
- 数据类型（type）
- 目的地（dest）
- 源（source）
- 标识符（tag）
- 通信子（comm）
- 状态（status）

缓冲区（buffer）

应用程序定义的用于发送或接收数据的消息缓冲区

数据个数（count）

- 发送或接收指定数据类型的个数
- 数据类型的长度 * 数据个数的值为用户实际传递的消息长度

数据类型（type）

MPI 定义了一些缺省的数据类型，用户也可根据需要建立自己的类型

MPI 定义的与 C 数据类型的对应关系

MPI 数据类型	C 数据类型	MPI 数据类型	C 数据类型
MPI_CHAR	char	MPI_C_BOOL	_Bool
MPI_SHORT	signed short int	MPI_INT8_T	int8_t
MPI_INT	signed int	MPI_INT16_T	int16_t
MPI_LONG	signed long int	MPI_INT32_T	int32_t
MPI_LONG_LONG_INT	signed long long int	MPI_INT64_T	int64_t
MPI_LONG_LONG (as a synonym)	signed long long int	MPI_UINT8_T	uint8_t
MPI_SIGNED_CHAR	signed char	MPI_UINT16_T	uint16_t
MPI_UNSIGNED_CHAR	unsigned char	MPI_UINT32_T	uint32_t
MPI_UNSIGNED_SHORT	unsigned short int	MPI_UINT64_T	uint64_t
MPI_UNSIGNED	unsigned int	MPI_C_COMPLEX	float _Complex
MPI_UNSIGNED_LONG	unsigned long int	MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
MPI_UNSIGNED_LONG_LONG	unsigned long long int	MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_FLOAT	float	MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_DOUBLE	double	MPI_BYTE	8 binary digits
MPI_LONG_DOUBLE	long double	MPI_PACKED	data packed or unpacked with MPI_Pack MPI_Unpack
MPI_WCHAR	wchar_t (dened in <stddef.h>)		

MPI 定义的与 Fortran 数据类型的对应关系

MPI 数据类型	Fortran 数据类型
MPI_CHARACTER	character(1)
MPI_INTEGER	integer
MPI_REAL	real
MPI_DOUBLE_PRECISION	double precision
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack MPI_Unpack

目的地（dest）

发送进程指定的接收该消息的目的进程，也就是接收进程的进程号

源（source）

- 接收进程指定的发送该消息的源进程，也就是发送进程的进程号
- 如该值为 MPI_ANY_SOURCE 表示接收任意源进程发来的消息

标识符（tag）

- 由程序员指定的为标识一个消息的唯一非负整数值（0-32767）
- 发送操作和接收操作的标识符一定要匹配
- 对于接收操作来说，如 tag 指定为 MPI_ANY_TAG 则可与任何发送操作的 tag 相匹配

通信子（comm）

- 包含源与目的进程的一组上下文相关的进程集合
- 除非用户自己定义（创建）了新的通信子，否则一般使用系统预先定义的全局通信子 MPI_COMM_WORLD

状态（status）

- 对接收操作，包含接收消息的源进程（source）和标识符（tag）
- 在 C 程序中，是个包含三个成员的结构体：
  - status.MPI_SOURCE：发送数据的进程标识
  - status.MPI_TAG：发送数据使用的 tag 标识
  - status.MPI_ERROR：接收操作返回的错误代码
- 在 Fortran 程序中，是包含 MPI_STATUS_SIZE 个整数的数组：
  - status(MPI_SOURCE)：发送数据的进程标识
  - status(MPI_TAG)：发送数据使用的 tag 标识
  - status(MPI_ERROR)：接收操作返回的错误代码
- 相当于一种接收方对消息的监测机制，并且以其为依据对消息作出不同的处理（当用通配符接受消息时）

Hello World：F90 实例

```
USE MPI
INTEGER MyID, NumProcs, iErr
INTEGER STATUS(MPI_STATUS_SIZE)
CHARACTER*11 MSG_0,MSG_1
MSG_0='Hello world'

CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MyID, iErr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NumProcs, iErr)
if (MyID == 0) then
  CALL MPI_SEND(MSG_0,11,MPI_CHARACTER,1,0,MPI_COMM_WORLD,iErr)
  PRINT *, 'ID', MyID, 'send ', MSG_0,' to ID 1'
elseif (MyID == 1) then
  CALL MPI_RECV(MSG_1,11,MPI_CHARACTER,0,0,MPI_COMM_WORLD,STATUS,iErr)
  PRINT *, 'ID', MyID, 'recieve ', MSG_1,' from ID 0'
endif
CALL MPI_FINALIZE(iErr)

END
```

Hello World：C 实例

```c
#include "mpi.h"
int main( int argc, char **argv )
{
  char message[20];
  int myrank;
  MPI_Status status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
  if (myrank == 0)  /* code for process zero */
  {
      strcpy(message,"Hello, there");
      MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else if (myrank == 1) /* code for process one */
  {
      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
      printf("received :%s:\n", message);
  }
  MPI_Finalize();
}
```

并行计算 π 原理

π 计算公式：

$$
\begin{aligned}
h &= \frac{1.0}{n} \\
x &= h * (i - 0.5) \\
\pi &= \sum_{i=1}^{n} \frac{4.0}{1.0 + x * x}
\end{aligned}
$$

原理：将相邻的项依次分配到相邻的进程进行计算，最后求和。

并行计算 π：C 实例

```c
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{ int n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264 3;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (1) {
        if (myid == 0) {printf("Enter the number of intervals: (0 quits) "); scanf("%d",&n);}
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
        else {
            h   = 1.0 / (double) n;
            sum = 0.0;
            for (i=myid+1; i <= n; i += numprocs) {x=h*((double)i-0.5); sum+=(4.0/(1.0+x*x));}
            mypi = h * sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
            if (myid == 0) printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi-PI25DT));
        }
    }
    MPI_Finalize();
    return 0;
}
```

并行计算 π：F90 实例

```
use mpi
double precision PI25DT
parameter       (PI25DT = 3.14159265358979323846264 3d0)
double precision mypi, pi, h, sum, x, f, a
integer n, myid, numprocs, i, ierr
f(a) = 4.d0 / (1.d0 + a*a) ! function to integrate
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
10 if ( myid .eq. 0 ) then
   print *, 'Enter the number of intervals: (0 quits) '
   read(*,*) n
endif
call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr) ! broadcast n
if ( n .le. 0 ) goto 30 ! check for quit signal
h = 1.0d0/n ! calculate the interval size
sum = 0.0d0
do i = myid+1, n, numprocs
   x = h * (dble(i) - 0.5d0)
   sum = sum + f(x)
enddo
mypi = h * sum ! collect all the partial sums
call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,MPI_COMM_WORLD,ierr)
if (myid .eq. 0) print *, 'pi is ', pi, ' Error is', abs(pi - PI25DT) ! node 0 prints the answer.
goto 10
30 call MPI_FINALIZE(ierr)
end
```

并行计算向量点乘原理

向量点乘计算公式：

$$\mathbf{A} \cdot \mathbf{B} = \sum_{i=0}^{n-1} A[i] * B[i]$$

原理：将全体项进行等分，连续项依次分配到相邻的进程进行计算，最后求和。

并行计算向量点乘：C 实例

```
#include "mpi.h"
int main(argc,argv)
int argc;
char *argv[];
{
  double sum, sum_local;
  double a [256], b [256];
  int i, n, numprocs, myid, my_first, my_last;
  n = 256;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  my_first = myid * n/numprocs;
  my_last = (myid + 1) * n/numprocs;
  for (i = 0; i < n; i++) {
     a [i] = i * 0.5;
     b [i] = i * 2.0;
  }
  sum_local = 0;
  for (i = my_first; i < my_last; i++) {
       sum_local = sum_local + a[i]*b[i];
  }
  MPI_Allreduce(&sum_local, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  if (myid==0) printf ("sum = %f\n", sum);
  MPI_Finalize();
}
```

点对点通信：阻塞发送

- MPI_SEND(buf, count, datatype, dest, tag, comm)：标准模式
- MPI_RECV(buf, count, datatype, source, tag, comm, status)

- MPI_BSEND(buf, count, datatype, dest, tag, comm)：缓存模式
- MPI_SSEND(buf, count, datatype, dest, tag, comm)：同步模式
- MPI_RSEND(buf, count, datatype, dest, tag, comm)：准备好模式

三种附件通信模式

- 无论一个匹配接收是否已登入，能开始一个“缓存模式”的发送操作。它可以在一个匹配接收登入以前完成。但是，不象标准发送，这个操作是局部的，它的完成不依赖一个匹配接收的发生。因此，如果执行一个发送而没有匹配接收登入，那么 MPI 必须缓存正出发的消息, 以便允许发送调用完成。如果没有充足的缓存空间，一个错误将发生。可得到的缓存空间数量由用户控制。为使缓存模式有效可以要求用户分配缓存。

  只要一可能, 消息就被发送。

- 无论一个匹配接收是否已登入，能开始“同步模式”的一个发送操作。但是，只有一个匹配接收登入，接收操作已开始接收同步发送的消息时，发送操作将成功完成。所以，一个同步发送的完成不表示发送缓存能被再使用，但是表明接收者已到达执行的某一点，即它已开始执行匹配接收。如果发送和接收都是阻塞操作，那么同步模式的使用提供同步语义：两个进程在通信时聚会以前，一个通信不能完成。在这个模式下执行的一个发送是非局部的。

  发送者发一个“请求发送”的消息。接收者存储这个请求。当一个匹配接收登入时，接收者发回一个“允许发送”的消息，这时接收者发送消息。

- 标准发送：第一个协议用于短的消息, 第二个协议用于长的消息。

- 只有匹配接收已登入，可以开始一个“准备好通信”模式的发送。否则，这个操作是错误的，其结果是无定义的。在某些系统，这允许移出一个信号交换式操作，并导致提高性能。发送操作的完成不依赖一个匹配接收的状态，只表明发送缓存能被再使用。准备好模式的一个发送操作，与一个标准发送操作或一个同步发送操作有相同的语义；只是发送者给系统提供附加的信息（即：一个匹配接收已登入），能节省一些额外开销。因此，在一个正确的程序中，一个准备好发送能被一个标准发送替代，对程序的动作无影响而对性能有影响。

  发送者把消息拷贝到一个缓存, 然后以非阻塞方式发送它（使用与标准发送同样的协议）。

非阻塞通信

- 通过重叠通信和计算在许多系统能提高性能
- 当数据已被从发送缓存拷出时，发送完成调用返回

- MPI_ISEND(buf, count, datatype, dest, tag, comm, request)：标准模式
- MPI_IRECV(buf, count, datatype, source, tag, comm, request)：接收只有一种标准模式
- MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)：缓存模式
- MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)：同步模式
- MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)：准备好模式

注：多了个请求句柄（request 参数），此参数在 C 语言中为 MPI_Request 类型，在 Fortran 中为 INTEGER 类型。

等待通信完成和检测通信是否完成

- MPI_WAIT(request, status)：等待检测的通信完成
- MPI_WAITANY(count, array_of_requests, index, status)：等待任意一个检测的通信完成
- MPI_WAITALL(count, array_of_requests, array_of_statuses)：等待所有检测的通信都完成
- MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)：等待有些检测的通信完成
- MPI_TEST(request, flag, status)：测试检测的通信是否完成
- MPI_TESTANY(count, array_of_requests, index, ag, status)：测试是否任意一个检测的通信完成
- MPI_TESTALL(count, array_of_requests, ag, array_of_statuses)：测试是否所有要检测的通信都完成
- MPI_TESTSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)：测试是否有些检测的通信完成
- MPI_REQUEST_GET_STATUS(request, ag, status)：获取检测的通信的状态，与 WAIT 和 TEST 不同，并不释放句柄
- MPI_REQUEST_FREE(request)：释放某个通信

WAIT 是要等待满足某个条件后完成，TEST 是检测以后即完成

探测和取消

- MPI_IPROBE(source, tag, comm, flag, status)：非阻塞探测进程 source 是否发标记为 tag 的消息来，状态存储在 flag
- MPI_PROBE(source, tag, comm, status)：阻塞探测进程 source 是否发标记为 tag 的消息来
- MPI_CANCEL(request)：取消通信
- MPI_TEST_CANCELLED(status, flag)：探测通信是否已取消

坚持式通信请求

当重复发送接收同样参数的信息是，可以利用下面坚持式通信方式得到更高的通信性能

- 初始化发送和接收
  - MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)
  - MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request)
  - MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)
  - MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request)
  - MPI_RECV_INIT(buf, count, datatype, dest, tag, comm, request)
- MPI_START(request)：开始某个通信
- MPI_STARTALL(count, array_of_requests)：开始所有通信
- MPI_REQUEST_FREE(request)：释放某个通信

发送接收和空进程

- MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)：发送接收使用不同的缓冲区
- MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)：发送接收使用同样缓冲区
- MPI_PROC_NULL：空进程（虚拟进程），对此进程的发送或接收立即返回，在某些情况下编程采用空进程会更加方便

集合通信

- 所有组成员间的栅障同步（barrier synchronization）
- 一个成员到组内所有成员的广播（broadcast）通信
- 一个成员从所有组成员收集（gather）数据
- 一个成员向组内所有成员分散（scatter）数据
- 组内所有成员都接收结果
- 组内所有成员到所有成员间的分散/收集数据操作
- 全局归约（global reduction）操作如求和（sum），求极大值（max），或用户自定义的操作，这里结果返回给所有的组成员或仅返回给其中的一个成员
- 组合归约（combined reduction）和分散操作
- 组内所有成员上的搜索（scan）操作（也称前置操作）

集合通信函数列表

- MPI_BARRIER(comm)：同步
- MPI_BCAST(buffer, count, datatype, root, comm)：广播
- MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)：收集
- MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)：带有偏移的收集
- MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)：散发
- MPI_SCATTERV(sendbuf,sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)：带有偏移的散发
- MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)：所有进程都收集
- MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,recvtype,comm)：带有偏移的所有进程都收集
- MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)：全交换
- MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)：规约
- MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)：全部进程都进行规约
- MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)：规约后散发
- MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)：扫描，又称前置规约，对排在此进程前面的进程和此进程的数据进行规约
- MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm)：排它扫描，对排在此进程前面的进程的数据进行规约

MPI_REDUCE、MPI_ALLREDUCE、MPI_REDUCE_SCATTER 和 MPI_SCAN 的规约运算 op 可以为：

- MPI_MAX：最大值
- MPI_MIN：最小值
- MPI_SUM：求和
- MPI_PROD：乘积
- MPI_LAND：逻辑与
- MPI_BAND：位与
- MPI_LOR：逻辑或
- MPI_BOR：位或
- MPI_LXOR：逻辑异或
- MPI_BXOR：位异或

数据的打包与解包

一些通信库为发送不连续数据提供打包/解包函数，用户在发送前显式地把数据包装到一个连续的缓冲区，在接收之后从连续缓冲区中解包。在多数情况下允许有一个派生数据类型不显式打包和解包，用户指明要发送的和接收的数据的分布，通信库直接访问一个不连续的缓冲区。

- MPI_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm)：打包
- MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)：解包
- MPI_PACK_SIZE(incount, datatype, comm, size)：获取打包数据占用空间的上界

一个打包单元可用 MPI_PACKED 类型发送。任何点到点或收集式通信功能可被用来从一个进程移动构成打包单元的字节序列到另一个进程中。该打包单元这时可用任何接收操作使用任意数据类型来接收：类型匹配的规定对于以 MPI_PACKED 类型发送的消息不再起作用。

以任何类型发送的消息（包括 MPI_PACKED 类型）都可用 MPI_PACKED 类型接收后被调用 MPI_UNPACK 来解包。

打包解包例子

将浮点数数组和字符数组打包发送与接收后解包

```
USE MPI
INTEGER MYID,IERR,MYSIZE,STATUS(400),POSTION
REAL BUF1(10)
CHARACTER*20 BUF2
CHARACTER PSEND,PRECV

CALL MPI_INIT(IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYID, IERR)
IF (MYID==0) THEN
  BUF1=1.0
  BUF2="ABCDEFGHIGKLMNOPQRST"
  POSTION=0
  CALL MPI_PACK(BUF1, 10, MPI_REAL, PSEND, 100, POSTION, MPI_COMM_WORLD, IERR)
  CALL MPI_PACK(BUF2, 20, MPI_CHARACTER, PSEND, 100, POSTION, MPI_COMM_WORLD, IE
  CALL MPI_SEND(PSEND, POSTION, MPI_PACKED, 1, 1, MPI_COMM_WORLD, IERR)
ELSE IF (MYID==1) THEN
  CALL MPI_RECV(PRECV, 100, MPI_PACKED, 0, 1, MPI_COMM_WORLD, STATUS, IERR)
  CALL MPI_UNPACK(PRECV, 100, POSTION, BUF1, 10, MPI_REAL, MPI_COMM_WORLD, IERR)
  CALL MPI_UNPACK(PRECV, 100, POSTION, BUF2, 20, MPI_CHARACTER, MPI_COMM_WORLD
  PRINT*,'BUF1:␣',BUF1
  PRINT*,'BUF2:␣',BUF2
END IF
CALL MPI_FINALIZE(IERR)
END
```

MPI 高级功能

- 组，上下文及通信子
- 进程拓扑
- MPI 环境管理
- 派生数据类型
- 动态进程管理
- 远程存储访问
- 并行 I/O

MPI 组，上下文及通信子

- 通信子：将所有这些观点都封装起来以便为MPI中的所有操作提供适当的机会。通信子可分为两种：组内通信子用于一组进程内的操作；组间通信子用于两组进程间的点对点通信。
- 缓冲区：通信子提供了一个缓冲机制允许人们将与 MPI 固有特征等价的新的属性联系到通信子上。高级用户可利用这一机制进一步修饰通信子并通过 MPI 实现一些通信子函数。
- 组：组定义了一个进程的有序集合，每一进程具有一个序列号，而且为组间进程通信定义低级名字也是由组完成的。这样组在点对点通信中为进程名字定义了一个范围。另外组还定义了集合操作的范围。在 MPI 中可从通信子中对组进行分别维护，但是只有通信子才能用于通信操作。
- 上下文：为 MPI 提供了拥有分离安全的消息传送空间的能力。

MPI 进程拓扑

拓扑是加在内部通信子上的额外、可选的属性，它不能被加在组间通信子上。对于一组进程（通信子内部），拓扑能够提供一种方便的命名机制，另外，可以辅助运行时间系统，将进程映射到硬件上。

MPI 中的进程组是 n 个进程的集合，组中的每一进程被赋予一个从 0 到 n-1 的标识数。在许多并行应用程序中，进程的线性排列不能充分地反映进程间在逻辑上的通信模型（通常由基本问题几何和所用的数字算法所决定），进程经常被排列成二维或三维网格形式的拓扑模型，而且，通常用一个图来描述逻辑进程排列，我们指这种逻辑进程排列为“虚拟拓扑”。

在虚拟进程拓扑和底层的物理硬件拓扑之间有一个清晰的差别。进程分配到物理处理器上时，虚拟拓扑可以由系统开发，假设这会帮助提高所给机器的通信性能，然而，这种映射是如何来完成的，已超出了 MPI 的范围。另外，虚拟拓扑的描述仅依赖于应用，并且独立于机器。

MPI 环境管理

用于获取和在合适的地方设置相关于 MPI 实现及执行环境（例如错误处理）的不同参数的例程。这里描述了用于进入和离开 MPI 执行环境的过程。

MPI 派生数据类型

到此为止，所有的点对点通信只牵涉含有相同数据类型的相邻缓冲区，这对两种用户限制太大。一种是经常想传送含有不同数据类型值的消息的用户；另一种是经常发送非连续数据的用户。一种解决的办法是在发送端把非连续的数据打包到一个连续的缓冲区，在接收端再解包。这样做的缺点在于在两端都需要额外的内存到内存拷贝操作，甚至当通信子系统具有收集分散数据功能的时候也是如此。而 MPI 提供说明更通用的，混合的非连续通信缓冲区的机制。直到执行（implementation）时再决定数据应该在发送之前打包到连续缓冲中，还是直接从数据存储区收集。

这里提供的通用机制允许不需拷贝，而是直接传送各种形式和大小的目标。我们并没有假设 MPI 库是用本地语言描述的连续目标。因此，如果用户想要传送一个结构或一个数组部分，则需要向 MPI 提供一个通信缓冲区的定义，该定义用问题模仿那个结构和数组部分的定义。这些工具可以用于使库设计者定义能够传送用本地语言定义的目标的通信函数：通过对可获得的符号表或虚拟向量（dope vector）的定义解码即可。这种高级通信功能不是 MPI 的部分。

MPI 动态进程管理

- 组间通信子
- 动态创建新的 MPI 进程
- 独立进程间的通信
- 基于 socket 的通信

MPI 远程存储访问

- 远程存储访问即直接对非本地的存储空间进行访问
- 一个进程对另外一个进程的存储区域进行直接访问

MPI 并行 I/O

- 显式偏移的并行文件读写
- 多视口的并行文件并行读写
- 共享文件读写
- 分布式数组文件的存取

OpenMP 与 MPI 混合并行编程

MPI 并行进程内部的可以再执行 OpenMP 并行线程以联合进行并行计算，一般用于节点内部为共享内存，节点间为分布式内存的超算系统

```
program main
use omp_lib
use mpi
implicit none
integer, save :: iErr, MyID, NumNodes
integer i, j

call mpi_init(ierr)
call mpi_comm_rank(mpi_comm_world, MyID ,iErr)
call mpi_comm_size(mpi_comm_world, NumNodes, iErr)
print*,'MPI: My ID: ', MyID, ', Number of Processes: ', NumNodes
!$omp parallel
i=omp_get_num_threads()
j=omp_get_thread_num()
print*, 'OMP: Thread Number:', j, ', Number of Threads: ', i
!$omp end parallel
call mpi_finalize(ierr)
end
```

MPI 程序编译

在 MPICH/OpenMPI 平台下的编译：

- 编译 C 程序：
  mpicc -o prog-mpi prog-mpi.c
- 编译 C++ 程序，MPICH1、2、OpenMPI：
  mpiCC -o prog-mpi prog-mpi.cpp
- 编译 C++ 程序，MPICH2、OpenMPI：
  mpicxx -o prog-mpi prog-mpi.cpp
- 编译 F77 程序：
  mpif77 -o prog-mpi prog-mpi.f
- 编译 F90 程序：
  mpif90 -o prog-mpi prog-mpi.f90

对 MPI 与 OpenMP 混合编程的程序，只要将编译命令换成打开支持 OpenMP 的选项即可，比如：

- GCC：**-fopenmp**
- Intel：**-openmp**
- PGI：**-mp**

MPI 程序运行

加载 n 个进程运行：

- MPICH1、2、OpenMPI：
  mpirun -np n prog-mpi
- MPICH2、OpenMPI：
  mpiexec -np n prog-mpi

- 对 MPI 与 OpenMP 混合编程的程序的执行，只要设置 OpenMP 环境变量后用 MPI 程序运行方式运行即可
- 很多超算系统采用作业调度系统管理用户作业，此时需通过作业调度系统来运行作业，需参看对应的作业调度系统手册

并行效率

$$S = \frac{1}{f_{par}/P + (1 - f_{par})}$$

Amdahl 定律，其中 P 是并行的核数，f_{par} 为可并行的部分串行时占用的计算时间的百分比

并行效率

$$S = \frac{1}{f_{par}/P + (1 - f_{par})}$$

Amdahl 定律，其中 P 是并行的核数，f_{par} 为可并行的部分串行时占用的计算时间的百分比

假设 $f_{par} = 80\%$，那么 16 和 32 核时最大的并行效率是多少？

并行效率

$$S = \frac{1}{f_{par}/P + (1 - f_{par})}$$

Amdahl 定律，其中 P 是并行的核数，f_{par} 为可并行的部分串行时占用的计算时间的百分比
假设 $f_{par} = 80\%$，那么 16 和 32 核时最大的并行效率是多少？

- 16 核：4

并行效率

$$S = \frac{1}{f_{par}/P + (1 - f_{par})}$$

Amdahl 定律，其中 P 是并行的核数，f_{par} 为可并行的部分串行时占用的计算时间的百分比
假设 $f_{par} = 80\%$，那么 16 和 32 核时最大的并行效率是多少？

- 16 核：4
- 32 核：4.4

并行效率

$$S = \frac{1}{f_{par}/P + (1 - f_{par})}$$

Amdahl 定律，其中 P 是并行的核数，f_{par} 为可并行的部分串行时占用的计算时间的百分比
假设 $f_{par} = 80\%$，那么 16 和 32 核时最大的并行效率是多少？

- 16 核：4
- 32 核：4.4

尽可能多地并行化代码，特别是耗时部分

辅助软件工具

- 程序性能分析：
  - Intel VTune：
    http://software.intel.com/en-us/intel-vtune/
  - AMD CodeAnalyst：
    http://developer.amd.com/cpu/CodeAnalyst/
- 程序源码结构分析：
  - Understand：
    http://www.scitools.com/products/understand/

联系方式

- 中国科学院青岛生物能源与过程研究所：
  - http://www.qibebt.cas.cn
- 青能所超级计算中心：
  - http://scc.qibebt.cas.cn
- 李会民：
  - http://staff.ustc.edu.cn/~hmli/
  - lihm@qibebt.ac.cn、hmli@ustc.edu.cn、li.huimin@gmail.com
  - 0532-80662795