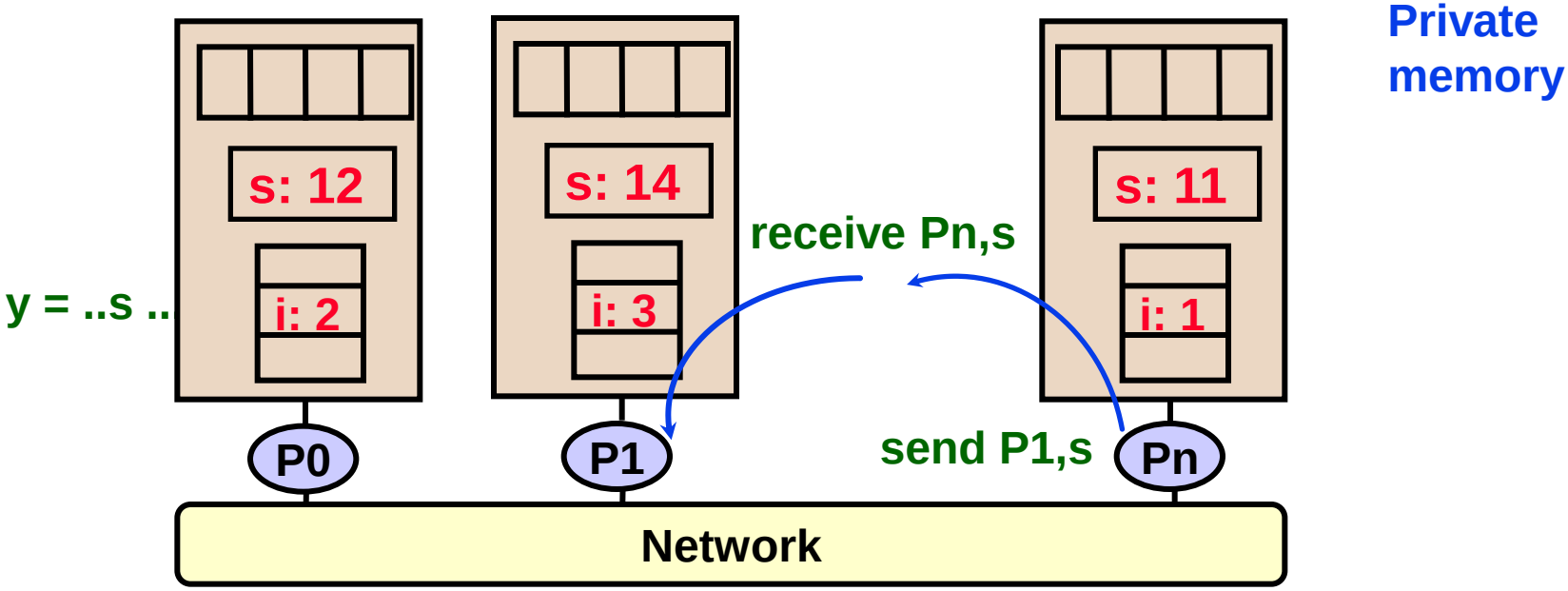
基于消息传递的并行编程(MPI)

内容安排

1. 回顾消息传递编程模型
2. MPI编程初步
3. 了解MPI基本的通信机制和MPI-2

主流并行程序模型—消息传递(Message Passing)

- 并行程序由一组命名进程组成
  - 分开的地址空间：没有共享数据，数据和负载显式分配
  - 进程间通信采用显式的send/receive组
  - 显式同步以确保执行顺序
  - 多进程，异步并行
  - 进程数多在程序启动时确定



消息传递

消息传递的特点：

在消息传递模型中，一个并行应用由一组进程组成，每个进程的代码是本地的，只能访问私有数据，进程之间通过传递消息实现数据共享和进程同步.

优点： 用户可以对并行性的开发、数据分布和通信实现完全控制.

缺点：

(1) 要求程序员显式地处理通信问题，如，消息传递调用的位置，数据移动，数据复制，数据操作，数据的一致性等等.

(2) 对大多数科学计算程序来说，消息传递模型的真正困难还在于显式的域分解，也就是说，将对相应数据的操作限定在指定的处理器上进行，在每个处理器上只能看见整个分布数据的一部分.

(3) 无法以渐进的方式、通过逐步将串行代码转换成并行代码而开发出来. 大量的散布在程序各处的域分解要求整个程序由串行到并行的转换一次性实现，而共享存储方法允许在现有的串行代码中插入并行说明从而实现逐步转换. 与之相比，这是消息传递的一个明显的缺点.

消息传递功能需求

- 共享存储编程(OpenMP)
  - 定义并行区 (omp parallel)
  - 设置并行度
  - 并行结构 (omp for; omp (parallel) sections)
  - 任务分配(schedule)
  - 数据管理/变量分类 (omp private/shared)
  - 同步控制 (omp critical…)
- 消息传递编程
  - 定义并行区
  - 设置并行度(静态、动态)
  - 程序员完成任务分配
  - 通信管理(消息管理、通信模式)
  - 同步控制(BARRIER)

消息传递接口(Message-Passing Interface)

在当前所有的消息传递软件中, 最重要最流行的是MPI, 它能运行在所有的并行平台上, 包括SMP. 已经在Windows这样的非Unix平台上实现. 程序设计语言支持C/C++, Fortran, Java.

消息传递库(Message-Passing Libraries)

1.1 MPI(Message Passing Interface) 简介

1992年4月	组建了一个制定消息传递接口标准的工作组
1992年10月	初稿形成，主要定义了点对点通信接口
1993年1月	第一届MPI会议在Dallas举行
1993年2月	公布了MPI-1修定版本
1993年11月	MPI的草稿和概述发表在Supercomputing‘93的会议论文集中
1994年5月	MPI标准正式发布
1994年7月	发布了MPI标准的勘误表
1997年	MPI论坛发布了一个修订的标准，叫做MPI-2，同时，原来的MPI更名为MPI-1

目标：是提供一个实际可用的、可移植的、高效的和灵活的消息传递接口标准. MPI以语言独立的形式来定义这个接口库，并提供了与C、Fortran和Java语言的绑定. 这个定义不包含任何专用于某个特别的制造商、操作系统或硬件的特性. 由于这个原因，MPI在并行计算界被广泛地接受.

消息传递库(Message-Passing Libraries)

MPI的实现

- 建立在厂家专用的环境之上

  IBM SP2的POE/MPL,

  Intel Paragon的OSF/Nx

- 公共的MPI环境:

  LAM(Local Area Multicomputer) Ohio超级计算中心

  MPICH　　Argonne国家实验室与Mississippi州立大学

  Intel MPI　　Intel

  MVAPICH　　Ohio

  OpenMPI　　Indiana Univ., etc

MPICH是MPI在各种机器上的可移植实现,可以安装在几乎所有的平台上:

- ✓ PC
- ✓ 工作站
- ✓ SMP
- ✓ MPP
- ✓ COW

基于Linux系统安装INTEL MPI

- INTEL MPICH包的获取
  - 166.111.143.18/19: /tmp/l_mpi_p_4.0.1.007.tgz

- 安装
  - 解压： tar zxvf l_mpi_p_4.0.2.003.tgz
  - Linux完全安装 cd l_mpi_p_4.0.2.003; ./install.sh
    - 详见intelmpi install.txt
  - 目录结构(4.0.1.007)
    - intel64/bin 可执行脚本
    - intel64/etc 配置文件
    - intel64/include 头文件
    - intel64/lib 库
  - 不同版本的目录结构可能是不尽相同的

基于Linux系统的INTELMPI

- 配置mpi环境
  - source /apps/intel/impi/4.0.2.003/intel64/bin/mpivars.sh
  - 将上述语句加入~/.bashrc

- 启动mpi环境
  - 单机：mpd &
  - 多机：mpdboot --totalnum=4 --file=./hosts
    查看配置 mpdtrace

基于Linux系统的INTELMPI

- MPI程序的编译
  - mpiicpc/mpiicc/mpiifort/mpiifort
  - C++/C/F77/F90

基于Linux系统的INTELMPI (自有的多机环境需要考虑)

- 多进程的MPI程序运行
  - 通常多机有相同的用户
  - 必需建立信任机制
    - 为什么要建立信任机制？
    - 建议采用ssh，安装时需要指定[与MPI的安装配合]
      - ssh-keygen -t rsa
      - cd ~/ssh
      - cp -p id_rsa.pub authorized_keys2
      - chmod go-rwx authorized_keys2
      - ssh-agent $SHELL; ssh-add
  - 其他机器如何设置？

基于Linux系统的INTELMPI (没有共享目录多机环境)

- 复制可执行程序和相关数据
  - **将可执行程序和所需要的数据文件拷贝到指定机器的指定位置**
  - **scp cpi tp1:/home/mpi/mpich/examples/basic/.**
  - **scp 源 目**

    源/目 [机器名：]路径

基于Linux系统的INTELMPI (设置并行度)

- MPI程序的执行
  - 使用缺省配置文件

    mpiexec -n N executive_file

    缺省配置文件，使用缺省路径
  - **mpiexec–machinefile ./hosts –n N executive_file**

    指定配置文件，使用缺省路径
  - **mpiexec –host XXX –n N executive_file : -host XXX –n M executive_file2**

    指定主机名称，使用缺省路径；多可执行文件运行

MPI的6个基本函数

功能	含义
MPI_Init	启动MPI
MPI_Finalize	结束MPI计算
MPI_Comm_size	确定进程数
MPI_Comm_rank	返回进程ID
MPI_Send	发送一条消息
MPI_Recv	接收一条消息

设置并行区

- 不要忘了”mpi.h”
- 怎么没有具体的参数列表？

  man！如何设置man？
- 注意命名规则

MPI其他有用的功能函数

功能	含义
MPI_Wtime()	从过去某时刻到当前消耗的时间(秒)
MPI_Wtick()	**MPI_Wtime**返回结果的精度
MPI_Initialized()	检查是否**MPI_Init**被调用过
MPI_Abort()	中止**MPI**程序，用于出错处理

实验：MPI计时工具

一个计算∑foo(i)的MPI SPMD消息传递程序, 存放在文件 “myprog.c”中

```
#include "stdio.h"
#include "mpi.h"
int foo(i)
int i;
{return i;}
int main(argc, argv)
int argc;
char* argv[];
{
int i, tmp, sum=0, group_size, my_rank, N;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &group_size);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank==0) {
      printf("Enter N:"); scanf("%d",&N);
      for (i=1;i<group_size;i++)
            MPI_Send(&N, 1, MPI_INT, i, i, MPI_COMM_WORLD);
      for (i=my_rank;i<N;i=i+group_size) sum=sum+foo(i);
      for (i=1;i<group_size;i++) {
            MPI_Recv(&tmp,1,MPI_INT,i,i,MPI_COMM_WORLD,&status);
      sum=sum+tmp;
      }
      printf("\n The result = %d\n", sum);
}
else {
      MPI_Recv(&N,1,MPI_INT,0,my_rank,MPI_COMM_WORLD,&status);
      for (i=my_rank;i<N;i=i+group_size) sum=sum+foo(i);
      MPI_Send(&sum,1,MPI_INT,0,my_rank,MPI_COMM_WORLD);
}
MPI_Finalize(); return 0;
}
```

初始化MPI环境

得到缺省的进程组大小

得到每个进程在组中的编号

发送消息

接收消息

终止MPI环境

- 实验
  - ✓ 目的：启动mpi, 获取运行信息
  - ✓ 任务
    - ✓了解并行区概念
    - ✓打印进程号和进程总数
    - ✓控制进程数量
  - ✓ 位置：c01b02:/tmp/lec05/env.c

MPI中的消息

为什么MPI中的发送和接收操作要做得这么复杂呢?

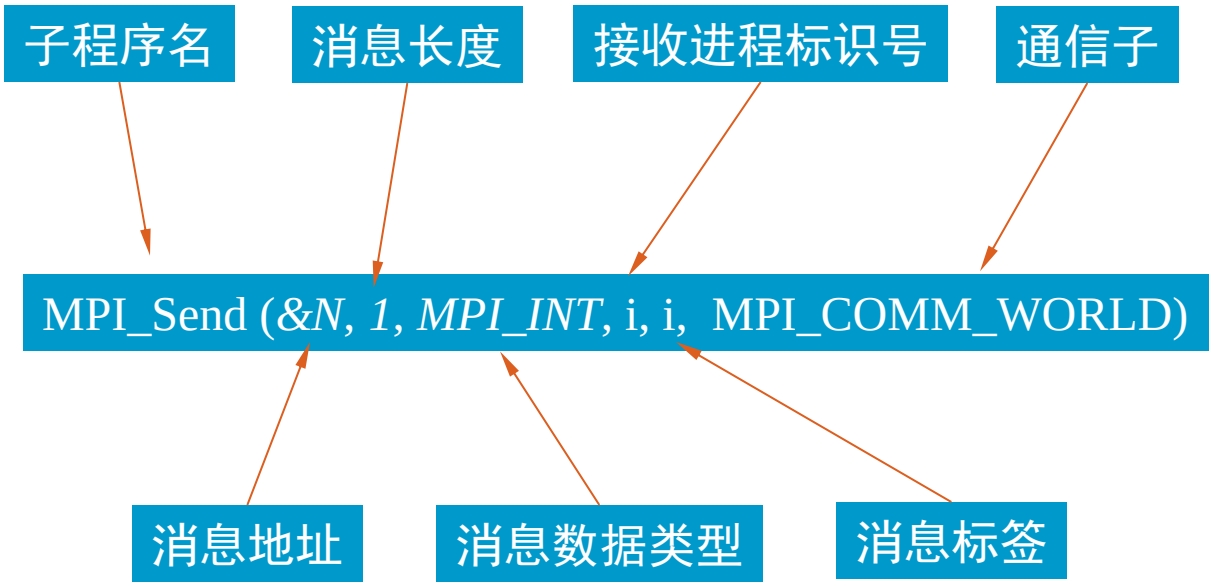
```
MPI_Send(&N,1,MPI_INT,i,i,MPI_COMM_WORLD)        send M to Q;
MPI_Recv(&N,1,MPI_INT,0,i,MPI_COMM_WORLD,&status) receive S from P
```

MPI消息的组成:

消息的内容 (即, 信的内容), 称为消息缓冲(message buffer)

消息的接收者 (即, 写在信封上面的东西), 称为消息信封(message envelop)

MPI中的消息



MPI_Send (buffer, count, datatype, destination, tag, communicator)

(buffer, count, datatype) 消息缓冲

(destination, tag, communicator) 消息信封

MPI中的消息

MPI的 四个重要概念:

- 消息数据类型 (message data types)
- 通信子 (communicators)
- 通信操作 (communication operations)
- 虚拟拓扑 (virtual topology)

MPI中的消息

为什么需要定义消息数据类型?

理由有两个:

- 一是支持异构计算
- 另一是允许非连续，非均匀内存区中的消息.

异构计算(heterogeneous computing): 指的是在由不同计算机，如工作站网络，组成的系统上运行应用程序. 系统中的每台计算机可能由不同的厂商生产、使用不同的处理器和操作系统当这些计算机使用不同的数据表示时如何保证互操作性.

MPI中的消息

表 1　MPI 中预定义的数据类型

MPI(C 语言绑定)	C	MPI(Fortran 语言绑定)	Fortran
MPI_BYTE		MPI_ BYTE	
MPI_CHAR	signed char	MPI_CHARACTER	CHARACTER
		MPI_COMPLEX	COMPLEX
MPI_DOUBLE	double	MPI_DOUBLE_PRECISION	DOUBLE_PRECISION
MPI_FLOAT	float	MPI_REAL	REAL
MPI_INT	int	MPI_INTEGER	INTEGER
		MPI_LOGICAL	LOGICAL
MPI_LONG	long		
MPI_LONG_DOUBLE	long double		
MPI_PACKED		MPI_PACKED	
MPI_SHORT	short		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long		
MPI_UNSIGNED_SHORT	unsigned short		

- 实验
  - ✓ 目的：mpi send/recv
  - ✓ 任务
    - ✓实现进程间标准消息通信
    - ✓进一步认识死锁
  - ✓ 位置：c01b02:/tmp/lec05/sendrecv.c

发送非连续数据

1. 导出数据类型

a） 连续MPI_Type_contiguous
b） 向量MPI_Type_vector
c） 索引MPI_Type_indexed
d） 结构MPI_Type_struct

Step1：定义新数据类型变量（MPI_Data_type）
Step2：建立导出数据类型（选取a－d及其组合 ）
Step3：提交新数据类型（MPI_Type_commit）

2. 打包

在消息传递中发送一个导出数据类型

- MPI_Type_contiguous (count,oldtype,&newtype)

MPI_ Type_contiguous

count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);

9.0	10.0	11.0	12.0

1 element of rowtype

- 实验
  - ✓ 目的：导出数据类型
  - ✓ 任务
    - ✓实现导出数据类型MPI_Type_contiguous
    - ✓位置：
      c01b02:/tmp/lec05/contiguous.c

在消息传递中发送一个导出数据类型

发送一数组的所有偶序数元素

```
double A[100];
MPI_Data_type EvenElements;
... ...
MPI_Type_vector(50,1,2,MPI_DOUBLE,&EvenElements);
MPI_Type_commit(&EvenElements);
MPI_Send(A,1,EvenElements,destination, ...);
```

例　　程	说　　明
MPI_Data_type	声明一个新的数据类型 EvenElements.
MPI_Type_vector(50,1,2,MPI_DOUBLE,&EvenElements)	产生一个导出数据类型 EvenElements，它由 50 个块组成. 每个块的组成是一个双精度数，后跟一个 8 字节的间隔，接在后面的是下一块. stride 是两个双精度数的大小，即 16 字节. 8 字节的间隔用于跳过数组 A 的奇序数元素.
MPI_Type_commit(&EvenElements)	这个新类型必须在被发送例程使用前交付使用.
MPI_Send(A,1,EvenElements,destination, ...)	注意: EvenElements 的一个元素包含 A 的所有 50 个偶序数元素. 因此, MPI_Send 的 count 域值为 1.

MPI_Type_hvector is identical to MPI_Type_vector except that stride is specified in bytes.

MPI中的消息

说明:

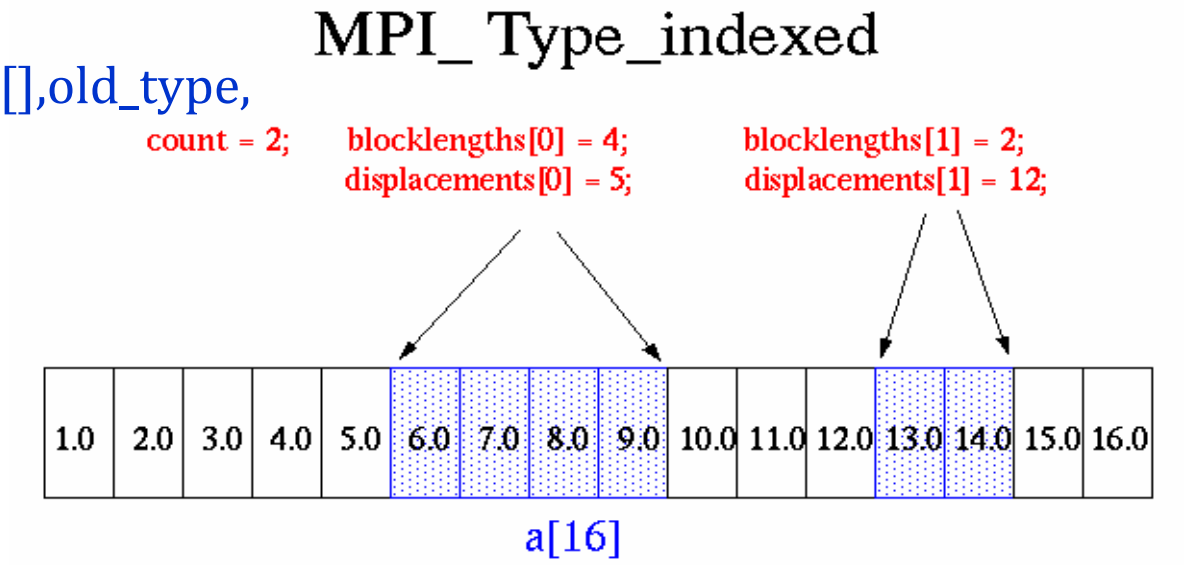
MPI_Type_vector(count, blocklength, stride, oldtype, &newtype)是构造导出数据类型的MPI例程.

导出类型newtype由blocks的拷贝count份组成. 每块(blocks)由已有的数据类型oldtype的blocklength份连续项的拷贝组成.

stride定义每两个连续的块之间的oldtype元素个数. 因此, (stride-blocklength)即是两个块之间的间隔.

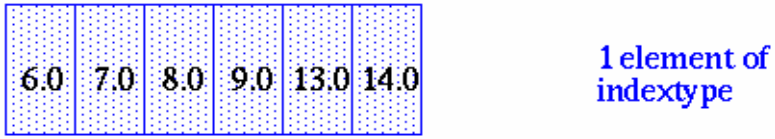
在消息传递中发送一个导出数据类型

- MPI_Type_indexed (count,blocklens[],offsets[],old_type, &newtype)




```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);
```

```
MPI_Send(&a, 1, indextype, dest, tag, comm);
```



在消息传递中发送一个导出数据类型

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

MPI_ Type_struct

```
typedef struct { float x, y, z, velocity; int n, type; } Particle;
Particle particles[NELEM];

MPI_Type_extent(MPI_FLOAT, &extent);

count = 2; oldtypes[0] = MPI_FLOAT;   oldtypes[1] = MPI_INT
           offsets[0] = 0;            offsets[1] = 4 * extent;
           blockcounts[0] = 4;        blockcounts[1] = 2;
```

particles[NELEM]

```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);

MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.

打包

```
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);
```

例 程	说 明
MPI_Pack_size	决定需要一个多大的临时缓冲区来存放50个MPI_DOUBLE数据项
malloc (BufferSize)	为缓冲区 TempBuffer 动态分配内存
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 (A+i*2, 1, MPI_DOUBLE, TempBuffe,BufferSize, &Position, comm);	第一个参数是被打包的数组元素的地址. 第三个参数是被打包的数组元素的数据类型, position 用于跟踪已经有多少个数据项被打包. position的最后值在接下来的MPI_Send中被用作消息计数.

MPI中的消息信封

用户如何来定义消息的接收者呢? 在下面列出的MPI发送例程中, 消息信封由三项组成.

destination 域是一个整数, 标识消息的接收进程.

MPI_Send (address, count, datatype, *destination, tag, communicator*)

消息标签(message tag), 也称为消息类型(message type), 是程序员用于标识不同类型消息、限制消息接收者的一个整数.

MPI中的消息信封

为什么要使用消息标签(Tag)?

为了说明为什么要用标签, 我们先来看右面一段没有使用标签的代码:

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.

使用标签可以避免这个错误.

未使用标签

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

使用了标签

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

在消息传递中使用标签

未使用标签

```
Process P:
send (request1,32, Q)

Process R:
send (request2, 32, Q)

Process Q:
while (true) {
      recv (received_request, 32, Any_Process);
      process received_request;
      }
```

使用标签的另一个原因是可以简化对下列情形的处理.
假定有两个客户进程P和R, 每个发送一个服务请求消息给服务进程Q.

使用了标签

```
Process P:
send(request1, 32, Q, tag1)

Process R:
send(request2, 32, Q, tag2)

Process Q:
while (true){
      recv(received_request, 32, Any_Process, Any_Tag, Status);
      if (Status.MPI_TAG==tag1) process received_request in one way;
      if (Status.MPI_TAG==tag2) process received_request in another way;
      }
```

MPI中的消息信封

通信子

MPI_Send (address, count, datatype, destination, tag, *communicator*)

通信子(communicator):

一个进程组(process group)+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.

一个进程在一个通信子(组)中用它的编号进行标识. 组的大小和进程编号可以通过调用以下的MPI例程获得:

```
MPI_Comm_size(communicator, &group_size)
MPI_Comm_rank(communicator, &my_rank)
```

MPI中的消息信封

MPI-1被设计成使不同通信子中的通信是相互分开的, 以及任何群集通信是与任何点对点通信分开的, 即使它们是在相同的通信子内. 通信子概念尤其方便了并行库的开发.

MPI-1只支持组内通信 (intra-communication)
MPI-2支持组间通信 (inter-communication)

MPI中的消息

```
typedef struct {
        int count;
        int MPI_SOURCE;
        int MPI_TAG;
        int MPI_ERROR;
#if (MPI_STATUS_SIZE > 4)
        int extra[MPI_STATUS_SIZE - 4];
#endif
} MPI_Status;
```

消息传递中的状态(Status)字

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, 状态信息就很有用.

```
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;
            }
}
```

MPI消息特性的总结

发送者进程总结如下

MPI_Send(buffer, count, datatype, destination, tag, communicator)

例子:

```
MPI_Send(&N, 1, MPI_INT, i, i , MPI_COMM_WORLD);
```

- 第一个参数指明消息缓存的起始地址, 即存放要发送的数据信息.
- 第二个参数指明消息中给定的数据类型有多少项, 这个数据类型由第三个参数给定.
- 数据类型要么是基本数据类型, 要么是导出数据类型, 后者由用户生成指定一个可能是由混合数据类型组成的非连续数据项.
- 第四个参数是目的进程的标识符(进程编号)
- 第五个是消息标签
- 第六个参数标识进程组和上下文, 即, 通信子. 通常, 消息只在同组的进程间传送. 但是, MPI允许通过intercommunicators在组间通信.

发送者进程总结如下

MPI_Recv(address, count, datatype,source, tag, communicator, status)

例:

MPI_Recv(&tmp, 1, MPI_INT, i, i, MPI_COMM_WORLD,&Status)

第一个参数指明接收消息缓冲的起始地址, 即存放接收消息的内存地址
第二个参数指明给定数据类型的最大项数, 它存放在第三个参数内, 可以被接收. 接收到的实际项数可能少一些
第四个参数是源进程标识符 (编号)
第五个是消息标签
第六个参数标识一个通信子
第七个参数是一个指针, 指向一个结构

这两个域可以是wildcard MPI_Any_source和 MPI_Any_tag.

MPI_Status Status

存放了各种有关接收消息的各种信息.

Status.MPI_SOURCE 实际的源进程编号

Status.MPI_TAG 实际的消息标签

实际接收到的数据项数由MPI例程

MPI_Get_count(&Status, MPI_INT, &C)

读出. 这个例程使用Status中的信息来决定给定数据类型(在这里是MPI_INT)中的实际项数, 将这个数放在变量C中.

小结

- 消息传递编程
  - **定义并行区(MPI_Init/MPI_Finalize)**
  - **设置并行度(静态, mpirun -np N)**
  - **通信管理(消息管理、通信模式)**
    - **数据类型(基本数据类型、导出数据类型)**
    - **TAG**
    - **通信子**
  - 同步控制

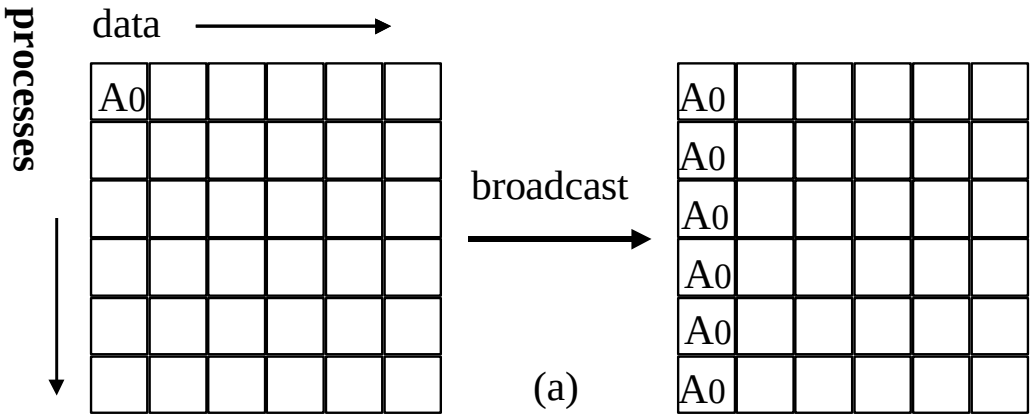
群集通信

广播(Broadcast)

MPI_Bcast(Address, Count, Datatype, *Root, Comm*)

在下列broadcast操作中，标号为Root的进程发送相同的消息给标记为Comm的通信子中的所有进程（包括root自身，为什么？）.

消息的内容如同点对点通信一样由三元组(Address, Count, Datatype)标识．对Root进程来说，这个三元组既定义了发送缓冲也定义了接收缓冲．对其它进程来说，这个三元组只定义了接收缓冲.



群集通信

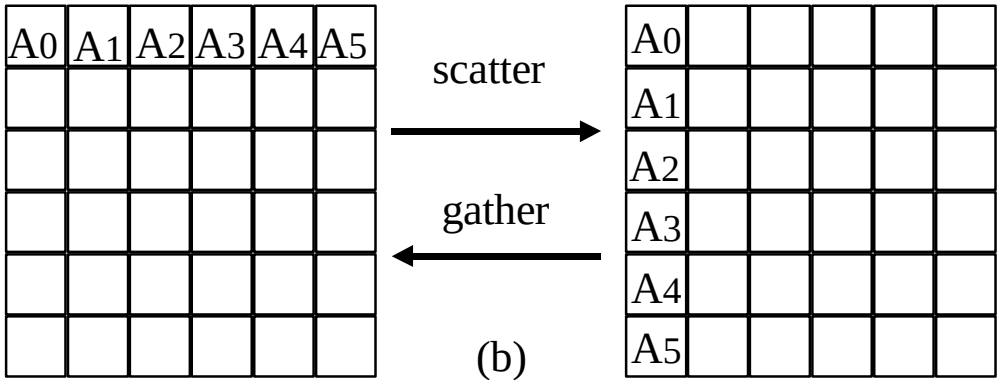
播撒(Scatter)

MPI_Scatter (SendAddress,SendCount,SendDatatype,
***RecvAddress,RecvCount,RecvDatatype,Root,Comm*)**

聚集(Gather)

MPI_Gather (SendAddress,SendCount,SendDatatype,
***RecvAddress,RecvCount,RecvDatatype,Root,Comm*)**

群集通信



MPI_Scatter

Scatter只执行与Gather相反的操作. Root进程发送给所有n个进程发送一个不同的消息, 包括它自已. 这n个消息在Root进程的发送缓冲区中按标号的顺序有序地存放. 每个接收缓冲由三元组(RecvAddress, RecvCount, RecvDatatype)标识. 所有的非Root进程忽略发送缓冲. 对Root进程, 发送缓冲由三元组(SendAddress, SendCount, SendDatatype)标识.

MPI_ Gather

Root进程从n个进程的每一个接收各个进程(包括它自已)的消息. 这n个消息的连接按序号rank进行, 存放在Root进程的接收缓冲中. 每个发送缓冲由三元组(SendAddress, SendCount, SendDatatype)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(RecvAddress, RecvCount, RecvDatatype)标识.

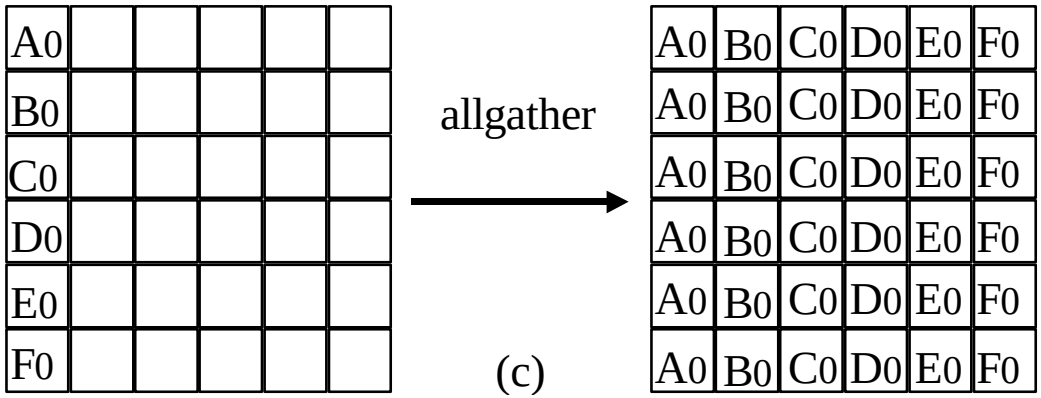
接收缓冲区足够大

- 实验
  - ✓ 目的：mpi 群集通信
  - ✓ 任务
    - ✓实现MPI_Scatter
    - ✓位置：
      c01b02:/tmp/lec05/collective.c

群集通信

扩展的聚集和播撒操作Allgather

MPI_Allgather (SendAddress, SendCount, SendDatatype,
RecvAddress, RecvCount, RecvDatatype, Comm **)**



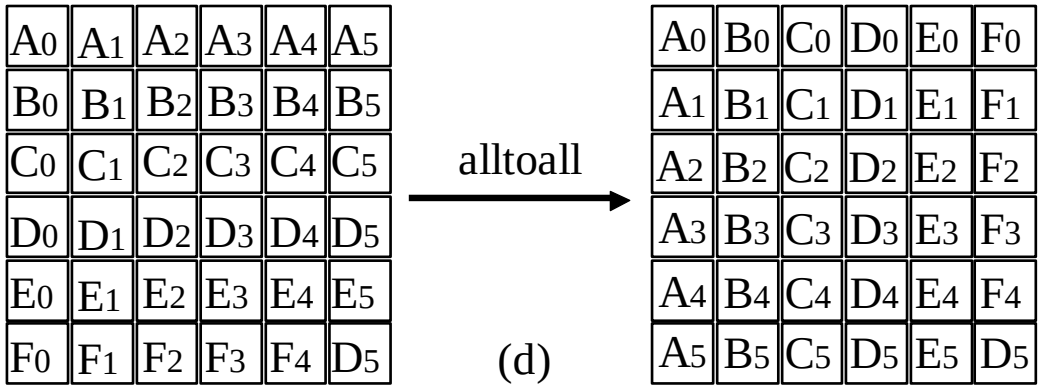
群集通信

全局交换(Tatal Exchange)

MPI_Alltoall(SendAddress, SendCount, SendDatatype,

***RecvAddress, RecvCount, RecvDatatype, Comm*)**

每个进程发送一个消息给所有n个进程, 包括它自已. 这n个消息在它的发送缓冲中以标号的顺序有序地存放. 从另一个角度来看这个通信, 每个进程都从n个进程接收一个消息. 这n个消息以标号的顺序被连接起来, 存放在接收缓冲中. 注意, 全局交换等于由n个不同进程做的n次Gather操作. 因此, 不再需要Root参数. 所有对所有, 在一次全局交换中共有n^2个消息进行通信.



(d)

群集通信

聚合(Aggregation)

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

归约(reduction)

MPI_Reduce(SendAddress, RecvAddress, Count, Datatype, Op, Root, Comm)

MPI_AllReduce(SendAddress, RecvAddress, Count, Datatype, Op, Comm)

这里每个进程的部分值存放在SendAddress中. 所有进程将这些值归约为最终结果并将它存入Root进程的RecvAddress. 数据项的数据类型在Datatype域中定义. 归约操作由Op域定义.

群集通信

- MPI定义的归约操作
  - MPI_MAX　　最大值
  - MPI_MIN　　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

群集通信

- 规约操作的数据类型组合
  - 基本数据类型组:
  - C语言中的整型　MPI_INT MPI_LONG MPI_SHORT
  - MPI_UNSIGNED_SHORT MPI_UNSIGNED
  - MPI_UNSIGNED_LONG
  - Fortran语言中的整型　MPI_INTEGER
  - 浮点数　MPI_FLOAT MPI_DOUBLE MPI_REAL
  - MPI_DOUBLE_PRECISION MPI_LONG_DOUBLE
  - 逻辑型　MPI_LOGICAL
  - 复数型　MPI_COMPLEX
  - 字节型　MPI_BYTE

群集通信

- 每种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,复数
  - MPI_LAND, MPI_LOR, MPI_LXOR　　C整数,逻辑型
  - MPI_BAND, MPI_BOR, MPI_BXOR　　C整数,Fortran整数,字节型

- 实验
  - ✓ 目的：mpi 并行pi
  - ✓ 任务
    - ✓用MPI_Bcast和MPI_Reduce实现pi计算
    - ✓位置：c01b02:/tmp/lec05/pi_mpi.c

群集通信

路障(Barrier)

MPI_Barrier(Comm)

在路障操作中, 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.

群集通信

群集例程的共同特点

- 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一个群集例程而其它没有调用, 则是错误的. 一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. [All or None]
- 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)、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的Address可以被安全地再次使用. 其它进程可能还没有启动它们相应的MPI_Bcast!

- 一个群集例程一般是同步的, 取决于实现. MPI要求用户负责保证他的代码无论实现是否是同步的都是正确的.
- 支持预定义数据类型而非导出数据类型.
- 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.

群集通信

表　MPI 中的群集通信

Type	Routine	Functionality
Data movement	MPI_Bcast	一对多播送相同的信息
	MPI_Gather	多对一收集个人信息
	MPI_Gatherv	通用的 MPI_Gather
	MPI_Allgather	全局收集操作
	MPI_Allgatherv	通用的 MPI_Allgather
	MPI_Scatter	一对多播撒个人信息
	MPI_Scatterv	通用的 MPI_Scatter
	MPI_Alltoall	多对多全交换个人信息
	MPI_Alltoallv	通用的 MPI_Alltoall
Aggregation	MPI_Reduce	多对一归约
	MPI_Allreduce	通用的 MPI_Reduce
	MPI_Reduce_scatter	通用的 MPI_Reduce
	MPI_Scan	多对多并行 prefix
Synchronization	MPI_Barrier	路障同步

MPI接口卡(1)

Message Passing Interface
Quick Reference in C

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

Blocking Point-to-Point

Send a message to one process. (§3.2.1)
```
int MPI_Send (void *buf, int count,
   MPI_Datatype datatype, int dest, int
   tag, MPI_Comm comm)
```
Receive a message from one process. (§3.2.4)
```
int MPI_Recv (void *buf, int count,
   MPI_Datatype datatype, int source, int
   tag, MPI_Comm comm, MPI_Status *status)
```
Count received data elements. (§3.2.5)
```
int MPI_Get_count (MPI_Status *status,
   MPI_Datatype datatype, int *count)
```
Wait for message arrival. (§3.8)
```
int MPI_Probe (int source, int tag,
   MPI_Comm comm, MPI_Status *status)
```
Related Functions: MPI_Bsend, MPI_Ssend, MPI_Rsend, MPI_Buffer_attach, MPI_Buffer_detach, MPI_Sendrecv, MPI_Sendrecv_replace, MPI_Get_elements

Non-blocking Point-to-Point

Begin to receive a message. (§3.7.2)
```
int MPI_Irecv (void *buf, int count,
   MPI_Datatype, int source, int tag,
   MPI_Comm comm, MPI_Request *request)
```
Complete a non-blocking operation. (§3.7.3)
```
int MPI_Wait (MPI_Request *request,
   MPI_Status *status)
```
Check or complete a non-blocking operation. (§3.7.3)
```
int MPI_Test (MPI_Request *request, int
   *flag, MPI_Status *status)
```
Check message arrival. (§3.8)
```
int MPI_Iprobe (int source, int tag,
   MPI_Comm comm, int *flag, MPI_Status
   *status)
```
Related Functions: MPI_Isend, MPI_Ibsend, MPI_Issend, MPI_Irsend, MPI_Request_free, MPI_Waitany, MPI_Testany, MPI_Waitall, MPI_Testall, MPI_Waitsome, MPI_Testsome, MPI_Cancel, MPI_Test_cancelled

Persistent Requests

Related Functions: MPI_Send_init, MPI_Bsend_init, MPI_Ssend_init, MPI_Rsend_init, MPI_Recv_init, MPI_Start, MPI_Startall

Derived Datatypes

Create a strided homogeneous vector. (§3.12.1)
```
int MPI_Type_vector (int count, int
   blocklength, int stride, MPI_Datatype
   oldtype, MPI_Datatype *newtype)
```
Save a derived datatype (§3.12.4)
```
int MPI_Type_commit (MPI_Datatype
   *datatype)
```
Pack data into a message buffer. (§3.13)
```
int MPI_Pack (void *inbuf, int incount,
   MPI_Datatype datatype, void *outbuf,
   int outsize, int *position, MPI_Comm
   comm)
```
Unpack data from a message buffer. (§3.13)
```
int MPI_Unpack (void *inbuf, int insize,
   int *position, void *outbuf, int
   outcount, MPI_Datatype datatype,
   MPI_Comm comm)
```
Determine buffer size for packed data. (§3.13)
```
int MPI_Pack_size (int incount,
   MPI_Datatype datatype, MPI_Comm comm,
   int *size)
```
Related Functions: MPI_Type_contiguous, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_hindexed, MPI_Type_struct, MPI_Address, MPI_Type_extent, MPI_Type_size, MPI_Type_lb, MPI_Type_ub, MPI_Type_free

Collective

Send one message to all group members. (§4.4)
```
int MPI_Bcast (void *buf, int count,
   MPI_Datatype datatype, int root,
   MPI_Comm comm)
```
Receive from all group members. (§4.5)
```
int MPI_Gather (void *sendbuf, int
   sendcount, MPI_Datatype sendtype, void
   *recvbuf, int recvcount, MPI_Datatype
   recvtype, int root, MPI_Comm comm)
```
Send separate messages to all group members. (§4.6)
```
int MPI_Scatter (void *sendbuf, int
   sendcount, MPI_Datatype sendtype, void
   *recvbuf, int recvcount, MPI_Datatype
   recvtype, int root, MPI_Comm comm)
```
Combine messages from all group members. (§4.9.1)
```
int MPI_Reduce (void *sendbuf, void
   *recvbuf, int count, MPI_Datatype
   datatype, MPI_Op op, int root, MPI_Comm
   comm)
```
Related Functions: MPI_Barrier, MPI_Gatherv, MPI_Scatterv, MPI_Allgather, MPI_Allgatherv, MPI_Alltoall, MPI_Alltoallv, MPI_Op_create, MPI_Op_free, MPI_Allreduce, MPI_Reduce_scatter, MPI_Scan

Groups

Related Functions: MPI_Group_size, MPI_Group_rank, MPI_Group_translate_ranks, MPI_Group_compare, MPI_Comm_group, MPI_Group_union, MPI_Group_intersection, MPI_Group_difference, MPI_Group_incl, MPI_Group_excl, MPI_Group_range_incl, MPI_Group_range_excl, MPI_Group_free

Basic Communicators

Count group members in communicator. (§5.4.1)
```
int MPI_Comm_size (MPI_Comm comm, int
   *size)
```
Determine group rank of self. (§5.4.1)
```
int MPI_Comm_rank (MPI_Comm comm, int
   *rank)
```
Duplicate with new context. (§5.4.2)
```
int MPI_Comm_dup (MPI_Comm comm, MPI_Comm
   *newcomm)
```
Split into categorized sub-groups. (§5.4.2)
```
int MPI_Comm_split (MPI_Comm comm, int
   color, int key, MPI_Comm *newcomm)
```
Related Functions: MPI_Comm_compare, MPI_Comm_create, MPI_Comm_free,

MPI接口卡(2)

MPI_Comm_test_inter, MPI_Comm_remote_size, MPI_Comm_remote_group, MPI_Intercomm_create, MPI_Intercomm_merge

Communicators with Topology

Create with cartesian topology. (§6.5.1)
```
int MPI_Cart_create (MPI_Comm comm_old,
    int ndims, int *dims, int *periods, int
    reorder, MPI_Comm *comm_cart)
```
Suggest balanced dimension ranges. (§6.5.2)
```
int MPI_Dims_create (int nnodes, int
    ndims, int *dims)
```
Determine rank from cartesian coordinates. (§6.5.4)
```
int MPI_Cart_rank (MPI_Comm comm, int
    *coords, int *rank)
```
Determine cartesian coordinates from rank. (§6.5.4)
```
int MPI_Cart_coords (MPI_Comm comm, int
    rank, int maxdims, int *coords)
```
Determine ranks for cartesian shift. (§6.5.5)
```
int MPI_Cart_shift (MPI_Comm comm, int
    direction, int disp, int *rank_source,
    int *rank_dest)
```
Split into lower dimensional sub-grids. (§6.5.6)
```
int MPI_Cart_sub (MPI_Comm comm, int
    *remain_dims, MPI_Comm *newcomm)
```
Related Functions: MPI_Graph_create, MPI_Topo_test, MPI_Graphdims_get, MPI_Graph_get, MPI_Cartdim_get, MPI_Cart_get, MPI_Graph_neighbors_count, MPI_Graph_neighbors, MPI_Cart_map, MPI_Graph_map

Communicator Caches

Related Functions: MPI_Keyval_create, MPI_Keyval_free, MPI_Attr_put, MPI_Attr_get, MPI_Attr_delete

---

LAM & MPI Information

1224 Kinnear Rd.
Columbus, Ohio 43212
614-292-8492

lam@tbag.osc.edu

http://www.osc.edu/lam.html
ftp://tbag.osc.edu/pub/lam

Error Handling

Related Functions: MPI_Errhandler_create, MPI_Errhandler_set, MPI_Errhandler_get, MPI_Errhandler_free, MPI_Error_string, MPI_Error_class

Environmental

Determine wall clock time. (§7.4)
```
double MPI_Wtime (void)
```
Initialize MPI. (§7.5)
```
int MPI_Init (int *argc, char ***argv)
```
Cleanup MPI. (§7.5)
```
int MPI_Finalize (void)
```
Related Functions: MPI_Get_processor_name, MPI_Wtick, MPI_Initialized, MPI_Abort, MPI_Pcontrol

Constants

Wildcards (§3.2.4)
```
MPI_ANY_TAG, MPI_ANY_SOURCE
```
Elementary Datatypes (§3.2.2)
```
MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG,
MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT,
MPI_UNSIGNED, MPI_UNSIGNED_LONG,
MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE,
MPI_BYTE, MPI_PACKED
```
Reserved Communicators (§5.2.4)
```
MPI_COMM_WORLD, MPI_COMM_SELF
```
Reduction Operations (§4.9.2)
```
MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD,
    MPI_BAND, MPI_BOR, MPI_BXOR, MPI_LAND,
    MPI_LOR, MPI_LXOR
```

LAM Quick Reference

LAM / MPI Extensions

Spawn processes.
```
int MPIL_Spawn (MPI_Comm comm, char *app,
    int root, MPI_Comm *child_comm);
```
Get communicator ID.
```
int MPIL_Comm_id (MPI_Comm comm, int *id);
```
Deliver an asynchronous signal.
```
int MPIL_Signal (MPI_Comm comm, int rank,
    int signo);
```
Enable trace collection.
```
int MPIL_Trace_on (void);
```
Related Functions: MPIL_Comm_parent, MPIL_Universe_size, MPIL_Type_id, MPIL_Comm_gps, MPIL_Trace_off

Session Management

Confirm a group of hosts.
```
recon -v <hostfile>
```
Start LAM on a group of hosts.
```
lamboot -v <hostfile>
```
Terminate LAM.
```
wipe -v <hostfile>
```
Hostfile Syntax
```
# comment
<hostname> <userid>
<hostname> <userid>
...etc...
```

Compilation

Compile a program for LAM / MPI.
```
hcc -o <binary> <source> -I<incdir>
    -L<libdir> -l<lib> -lmpi
```

Processes and Messages

Start an SPMD application.
```
mpirun -v -s <src_node> -c <copies>
    <nodes> <program> -- <args>
```
Start a MIMD application.
```
mpirun -v <appfile>
```
Appfile Syntax
```
# comment
<program> -s <src_node> <nodes> -- <args>
<program> -s <src_node> <nodes> -- <args>
...etc...
```
Examine the state of processes.
```
mpitask
```
Examine the state of messages.
```
mpimsg
```
Cleanup all processes and messages.
```
lamclean -v
```

- 实验
  - ✓ 目的：了解mpi通信性能
  - ✓ 任务
    - ✓用IMB-MPI1
    - ✓位置：
      /apps/intel/impi/4.0.2.003/intel64/
      bin/IMB-MPI1

1 https://computing.llnl.gov/tutorials/mpi/

- **Skip LLNL MPI Implementations and Compilers Section**
- **Learn General Concepts and Non-Blocking Message Passing Routines sections by yourself**
- **Any question let me know**

2 Intel MPI getting started, Reference_manual

3 IMB-MPI userguide

Backup

MPI中的通信子管理

- MPI_Comm_dup(comm，newcom),对comm进行复制得到新的通信域newcomm，复制包括组以及缓冲区等信息

- MPI_Comm_split(comm, color, key, &newcomm)
  - 对于comm中的进程都要执行
  - 每个进程指定一个color，此调用首先将具有相同color的值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的大小决定，即key越小，则相应进程在原来通信域中的顺序编号也越小，若两个进程中的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编号。
  - 一个进程可能提供color值MPI_Undefined，此种情况下，newcomm返回MPI_Comm_null

- MPI_Comm_free(comm),释放给定通信域

MPI中的消息信封

MPI中的新通信子

考虑如下由10个进程执行的代码:

```
MPI_Comm MyWorld, SplitWorld;
int my_rank, group_size, Color, Key;

        MPI_Init(&argc, &argv);
        MPI_Comm_dup(MPI_COMM_WORLD,&MyWorld);
        MPI_Comm_rank(MyWorld,&my_rank);
        MPI_Comm_size(MyWorld,&group_size);

        Color=my_rank%3;
        Key=my_rank/3;

        MPI_Comm_split(MyWorld,Color,Key,&SplitWorld);
```

MPI中的消息信封

MPI_Comm_dup(MPI_COMM_WORLD,&MyWorld)

将创建一个新的通信子MyWorld, 它是包含与原始的MPI_COMM_WORLD相同的10个进程的进程组,但有不同的上下文.

表 分裂一个通信子 MyWorld

Rank in MyWorld										
Color										
Key										
Rank in SplitWorld(Color=0)										
Rank in SplitWorld(Color=1)										
Rank in SplitWorld(Color=2)										

MPI中的消息信封

MPI_Comm_dup(MPI_COMM_WORLD,&MyWorld)

将创建一个新的通信子MyWorld, 它是包含与原始的MPI_COMM_WORLD相同的10个进程的进程组,但有不同的上下文.

表 分裂一个通信子 MyWorld

Rank in MyWorld	0	1	2	3	4	5	6	7	8	9
Color	0	1	2	0	1	2	0	1	2	0
Key	0	0	0	1	1	1	2	2	2	3
Rank in SplitWorld(Color=0)	0			1			2			3
Rank in SplitWorld(Color=1)		0			1			2		
Rank in SplitWorld(Color=2)			0			1			2	