MPI并行程序设计（C版）

法律条款

1. 对本文档的任何使用都被视为完全理解并接受本文档列举的所有法律条款。
2. 本文档的所有权利归作者所有，作者保留所有权利。
3. 未经作者书面同意，禁止任何形式的商业使用。商业使用形式包括但不限于出版、复制、传播、展示、引用、编辑。
4. 本文档允许以学术研究、技术交流为目的使用。复制、传播过程中不得对本文档作任何增减编辑，引用时需注明出处。
5. 实施任何侵权行为的法人或自然人都必须向作者支付赔偿金，赔偿金计算方法为：

   赔偿金 = 涉案人次 × 涉案时长（天）× 涉案文档份数 × 受众人次 × 100元人民币，

   涉案人次、涉案时长、涉案文档份数、受众人次小于1时，按1计算。
6. 对举报侵权行为、提供有价值证据的自然人或法人，作者承诺奖励案件实际赔偿金的50%.
7. 涉及本文档的法律纠纷由作者所在地法院裁决。
8. 本文档所列举法律条款的最终解释权归作者所有。

并行计算简介

MPI并行编程

- MPI简介
- MPI程序基础架构
- MPI基本函数
  - 4个基本函数
- MPI基本通信函数
- MPI程序编译、运行
- 集合通信
  - 数据广播、收集、散发、转置、归约
  - 综合实例：并行计算π 值

MPI软件包的安装、使用

- OpenMPI, Mpich2, mvapich2

目录

并行计算简介

■ 并行计算定义

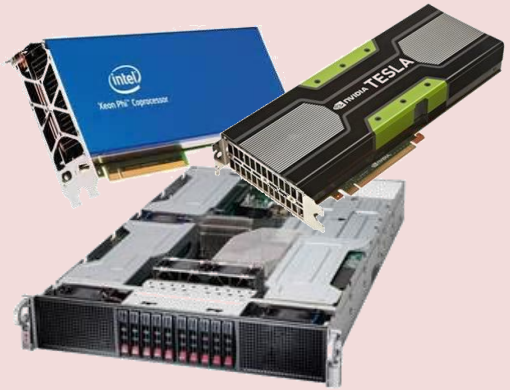
并行计算是指，在并行机上，将一个应用分解成多个子任务，分配给不同的处理器，各个处理器之间相互协同，并行的执行子任务，从而达到加速求解速度的目的。

■ 并行计算条件

- □ 开展并行计算必须具备三个基本条件：
  1、并行机（硬件）
  2、应用问题必须具有并行度（应用）
  3、并行程序（软件）

并行层次

CPU单核并行
指令级并行
SIMD、超线程
多核并行
共享内存OpenMP



加速器
nVidia GPU
AMD GPU
Intel Xeon Phi
专为并行计算设计

NVIDIA CUDA
OpenACC

超级计算机
多节点、异构
成千上万核心
成熟的并行技术

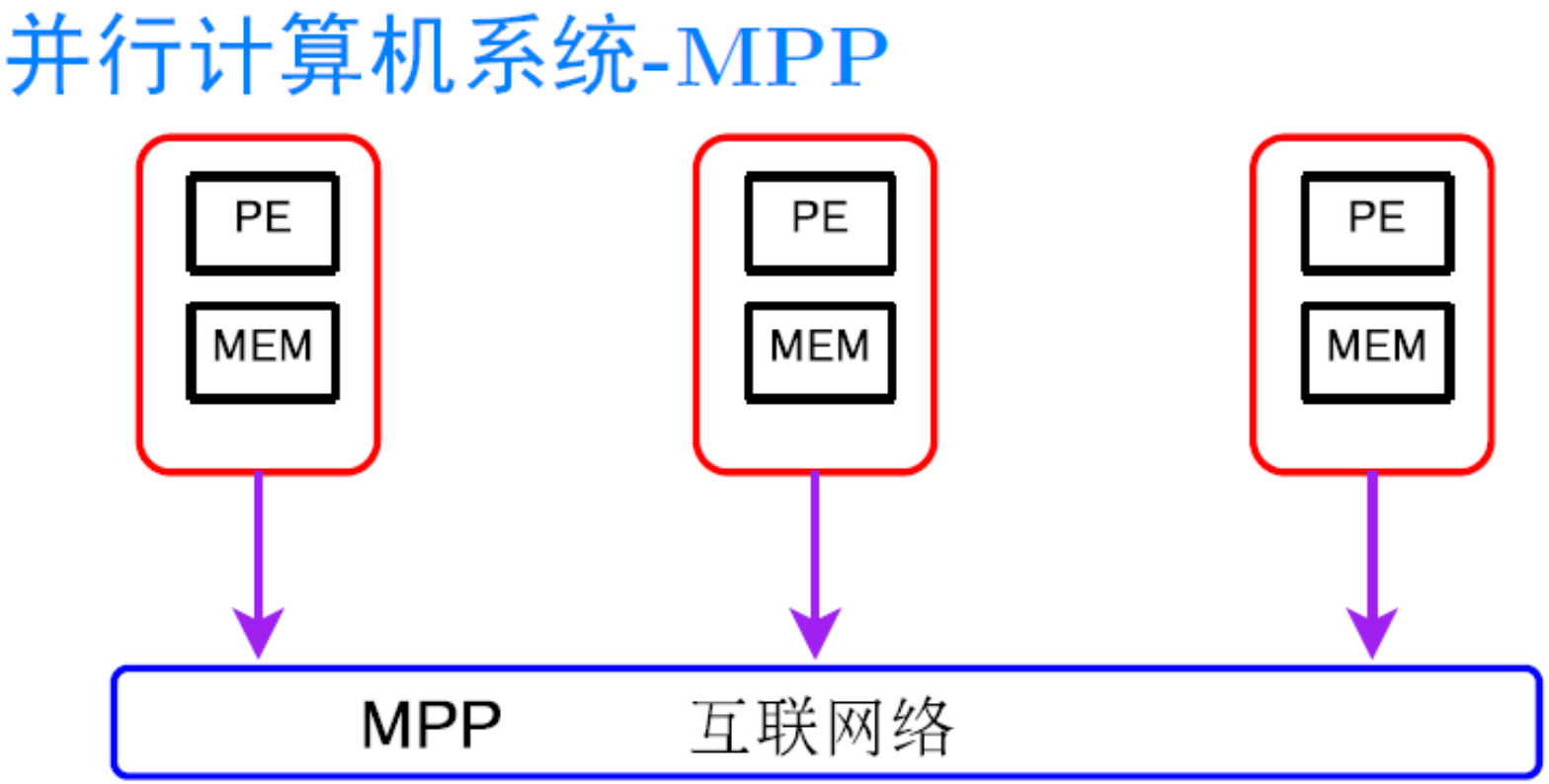
ANSYS
CST
FEKO
Comprehensive Electromagnetic Solutions

MPI

并行计算简介

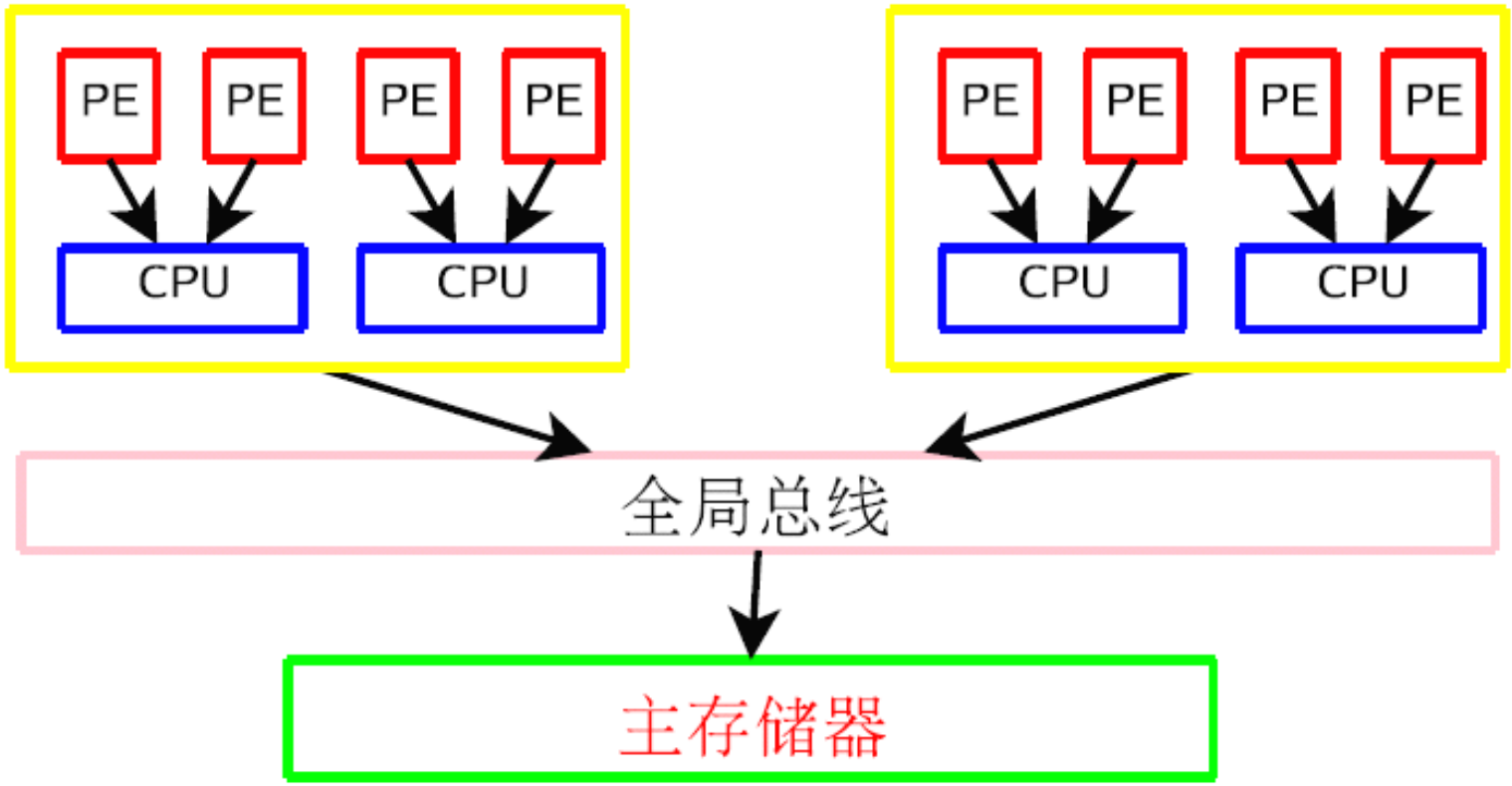
- 并行计算系统-分类
  - MPP
  - SMP
  - **Cluster**
  - CPU和GPU的混合系统
  - 星群系统

并行计算简介



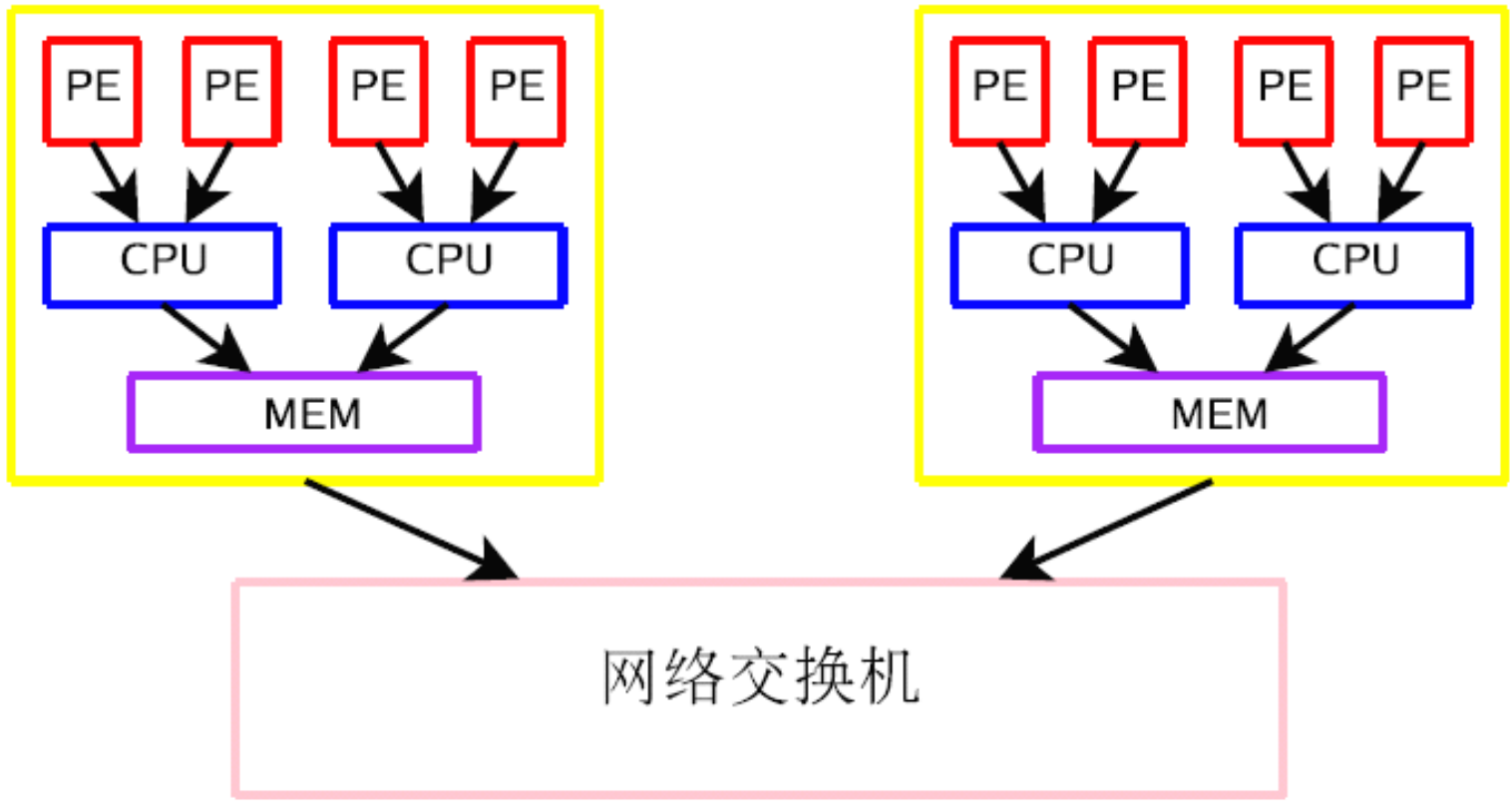
并行计算简介

并行计算机系统-SMP



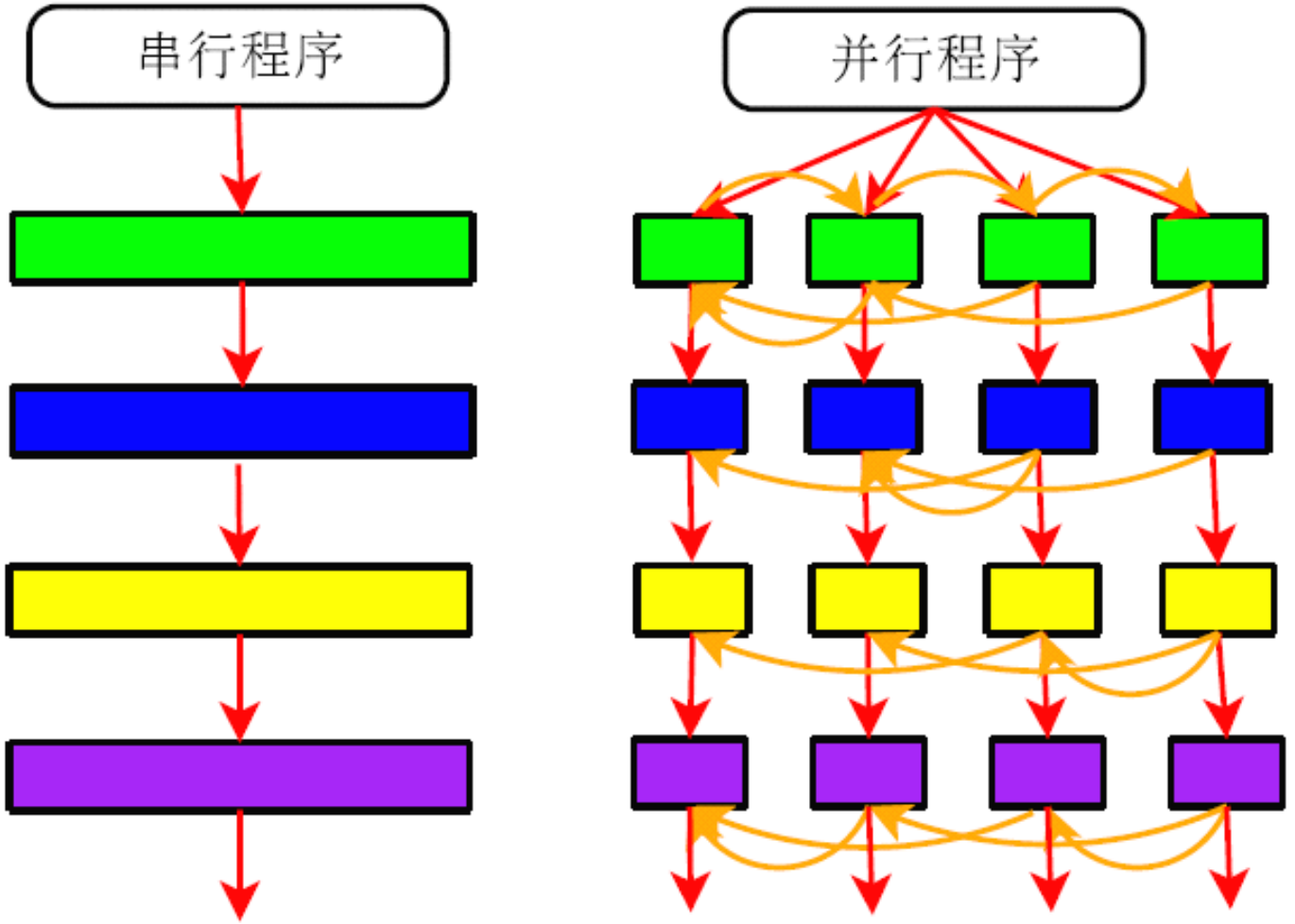
并行计算简介

并行计算机系统-Cluster



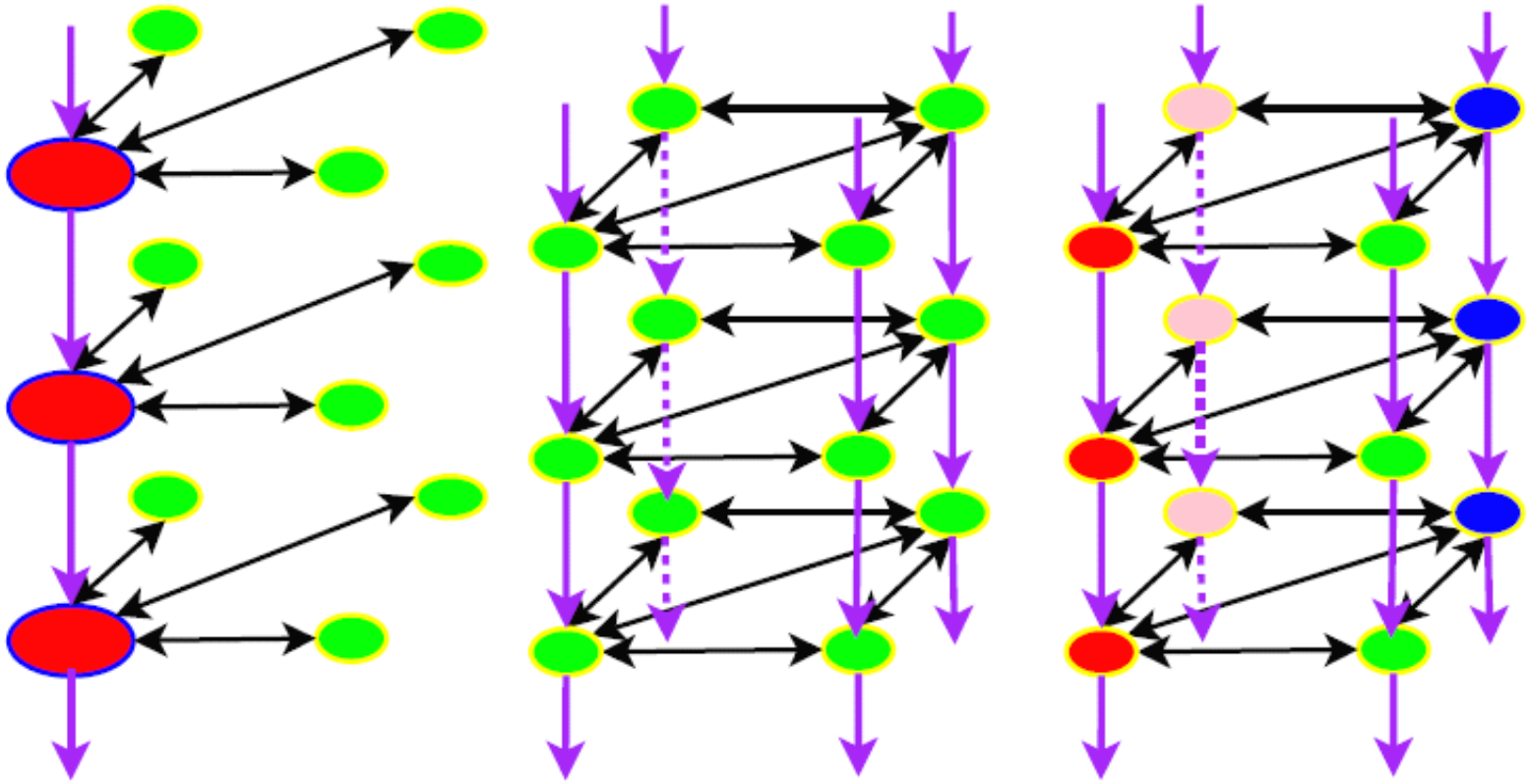
并行计算简介

并行计算的程序结构



并行计算简介

并行程序类型、MPI-SPMD并行程序结构



目录

MPI并行编程

■ MPI简介-什么是MPI：

MPI（Message-Passing Interface）是一种标准，最初是1991年在奥地利的学术和工业界的一些研究人员一块商议制定的。

■ 标准的实现：

- MPICH argonne natiaonal laboratory and Mississippi State University
- LAM/MPI Ohio Supercomputer Center
- Openmpi LAM/MPI和其他一些早期MPI产品的整合
- Mvapich MPI over infiniband

Ohio Supercomputer Center

- 其他MPI产品

HP、Intel和Microsoft（MPICH和LAM/MPI的衍生产品）

MPI并行编程

■ MPI简介-MPI发展历程

- MPI 1.1: 1995

  MPICH:是MPI最流行的非专利实现,由Argonne国家实验室和密西西比州立大学联合开发,具有更好的可移植性.

- MPI 1.2-2.0

  动态进程、并行I/O、远程存储访问、支持F90和C++(1997).

- MPI 2.2　　2009年
- MPI 3.0　　2012年9月21日

MPI并行编程

■ MPI简介-标准的实现

	MPICH	MVAPICH	OpenMPI	Intel MPI
开发者	Argonne National Lab	Ohio State University	OpenMPI development Team	Intel
是否开源	是	是	是	否
支持的网络	以太网	InfiniBand/以太网	InfiniBand/以太网	InfiniBand/以太网
MPI标准	2.2 3.0	2.2	2.2	2.2
前身	MPICH	MVAPICH	LAM-MPI	/

MPI并行编程

■ MPI简介-MPI分布式并行：

各节点内存相互独立

- 一个任务分成多个子任务，
- 每个节点启动若干进程，各自承担一个子任务
- 内存总容量无上限、计算时间无限压缩

子任务间需要通信

- 适应多种网络介质、协议、操作系统、编译器
- 硬件环境（CPU、数据类型的长度、大端小端、不同公司网卡）

并行计算简介

MPI并行编程

- MPI简介
- MPI程序基础架构
- MPI基本函数
  - 4个基本函数
- MPI基本通信函数
- MPI程序编译、运行
- 集合通信
  - 数据广播、收集、散发、转置、归约
  - 综合实例：并行计算π 值

MPI软件包的安装、使用

- OpenMPI, Mpich2, mvapich2

MPI并行编程

- MPI程序基础架构
  - 1、进入mpi环境。产生通讯子、进程号、进程数。
  - 2、程序主体。实现计算的全部内容。
  - 3、退出mpi环境。不能再使用mpi环境。

MPI并行编程

MPI程序基础架构

C和Fortran中MPI函数约定

- C
  - 必须包含mpi.h.
  - MPI 函数返回出错代码或 MPI_SUCCESS成功标志.
  - MPI_前缀，且只有MPI以及MPI_标志后的第一个字母大写，其余小写.
- Fortran
  - 必须包含mpif.h（fortran77），use mpi (fortran95)
  - 通过子函数形式调用MPI，函数最后一个参数为返回值

MPI并行编程

■ MPI程序基础架构

```
#include<stdio.h>
#include<mpi.h>
int main(int argc,char* argv[])
{
   int myid;
   int numprocs;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

   printf("myid is %d\n",myid);

   MPI_Finalize();
   return 0;
}
```

MPI并行编程

- MPI程序中，一个独立参与通信的个体称为一个进程(process)
  - 一个MPI进程通常对应于一个普通的进程或线程
  - 在共享内存/消息传递混合模式程序中，一个MPI进程可能代表一组UNIX线程
- 部分或全部进程构成的一个有序集合称为一个进程组。
  - 进程组中的每个进程都被赋于一个唯一的序号（rank）,称为进程号，从0开始编号
- 一个进程组及其相关属性（进程拓扑连接关系等），称为一个通信器(communicator)
  - 两个内置通信器，MPI_COMM_WORLD包含所有进程，MPI_COMM_SELF只包含进程自身

并行计算简介

MPI并行编程

- MPI简介
- MPI程序基础架构
- MPI基本函数
  - 4个基本函数
- MPI基本通信函数
- MPI程序编译、运行
- 集合通信
  - 数据广播、收集、散发、转置、归约
  - 综合实例：并行计算π 值

MPI软件包的安装、使用

- OpenMPI, Mpich2, mvapich2

MPI并行编程

- MPI基础函数
  1. MPI_Init(…);
  2. MPI_Comm_rank(…);
  3. MPI_Comm_size(…);
  4. MPI_Finalize();

MPI并行编程

- 启动、结束MPI环境

MPI_INIT	
C	int MPI_Init(int *argc, char ***argv)
Fortran	MPI_INIT(IERROR) INTEGER IERROR

- 最先被调用的MPI函数，完成MPI程序的所有初始化工作。启动MPI环境,标志并行代码的开始.
- 参数argc和argv遇main函数传来，从而main必须带参数运行
- 参数用来初始化并行环境（进程数、计算结点名、通信协议等）

MPI_FINALIZE	
C	int MPI_Finalize(void)
Fortran	MPI_FINALIZE(IERROR) INTEGER IERROR

- 最后调后的MPI函数，清除[illegible]
- 此后，MPI语句的运行结果是不可预知的
- 之后串行代码仍可在主进程(rank = 0)上运行(如果必须)

并行计算简介

MPI并行编程

- MPI简介
- MPI程序基础架构
- MPI基本函数
  - 4个基本函数
- MPI基本通信函数
- MPI程序编译、运行
- 集合通信
  - 数据广播、收集、散发、转置、归约
  - 综合实例：并行计算π 值

MPI软件包的安装、使用

- OpenMPI, Mpich2, mvapich2

MPI并行编程

- 基本的MPI通信域函数

MPI_COMM_SIZE	
C	int MPI_Comm_size(MPI_Comm comm, int *size)
Fortran	MPI_COMM_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR

获得通信域comm/COMM内的进程数目，保存于size/SIZE中

MPI_COMM_RANK	
C	int MPI_Comm_rank(MPI_Comm comm, int *rank)
Fortran	MPI_COMM_RANK(COMM, RANK, IERROR) INTEGER COMM, RANK, IERROR

获得该进程在通信域comm/COMM内的进程序号，保存于rank/RANK中

MPI并行编程

- 基本的MPI点对点通信函数

MPI_SEND	
C	int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
Fortran	MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

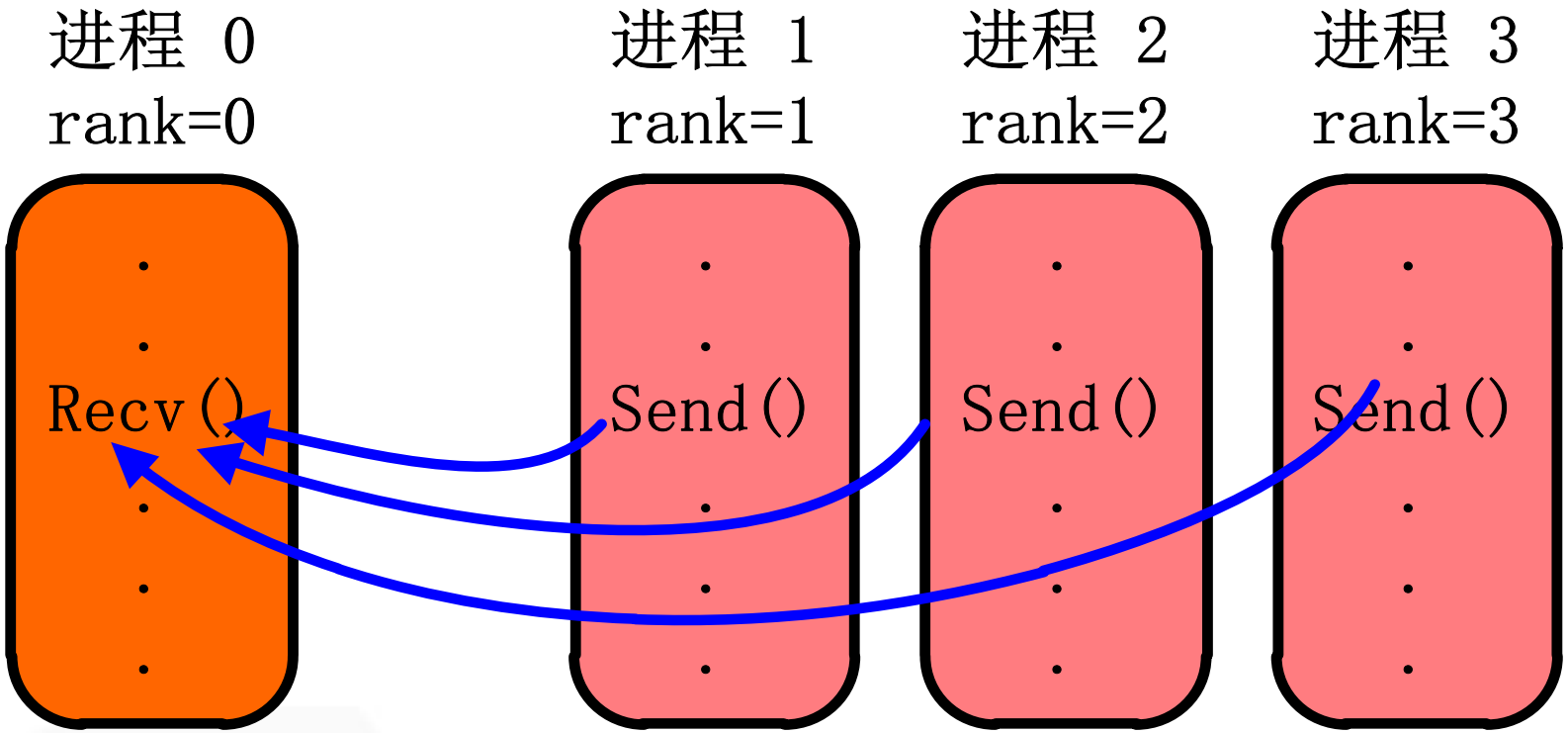
将发送缓冲区中的count个datatype数据类型的数据发送到目的进程

MPI_RECV	
C	int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
Fortran	MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR) <type> BUF(*) INTEGER COUNT, DATATYPE, SOURCE, TAG,COMM, STATUS(MPI_STATUS_SIZE), IERROR

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数据类型和消息标识相一致，收到的消息所包含的数据元素的个数最多不能超过count.

MPI并行编程

- 基本的MPI通信域函数-hello world



MPI并行编程　Hello World(c)


```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
  int numprocs; /*进程数,该变量为各处理器中的同名变量*/
  int myid;     /*本进程ID,存储也是分布的*/
  MPI_Status status;/*消息接收状态变量,存储也是分布的*/
  char message[100];/*消息buffer,存储也是分布的*/
  int source

  MPI_Init(&argc, &argv); /*初始化MPI*/
  /*该函数被各进程各调用一次,得到自己的进程rank值*/
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  /*该函数被各进程各调用一次,得到进程数*/
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
```

MPI并行编程　Hello World(c)

```
    if(myid != 0){/*建立消息*/
      sprintf(message, "\"Hello World\" from process
  %d/%d!", myid, numprocs);
      /* 发送长度取strlen(message)+1,使\0也一同发送出去*/
      MPI_Send(message, 100, MPI_CHAR, 0, 99,
  MPI_COMM_WORLD);
    }
    else{ /* myid == 0 */
      for(source = 1; source < numprocs; source++){
        MPI_Recv(message, 100, MPI_CHAR, source, 99,
  MPI_COMM_WORLD,&status);
        printf("%s\n", message);
      }
    }
    MPI_Finalize(); /*关闭MPI,标志并行代码段的结束*/
    return 0;
} /* End main */
```

并行计算简介

MPI并行编程

- MPI简介
- MPI程序基础架构
- MPI基本函数
  - 4个基本函数
- MPI基本通信函数
- MPI程序编译、运行
- 集合通信
  - 数据广播、收集、散发、转置、归约
  - 综合实例：并行计算π 值

MPI软件包的安装、使用

- OpenMPI, Mpich2, mvapich2

MPI并行编程

```
$ source /public/software/profile.d/ompi-1.6-gnu.sh
$ mpicc -o hello hello.c
$ ./hello                    (x)
[0]  Aborting program! Could not create p4
procgroup.  Possible missing fileor program
started without mpirun.
$ mpirun -np 4 hello                    (✓)
   "Hello World" from process 1/4!
   "Hello World" from process 2/4!
   "Hello World" from process 3/4!
$
```

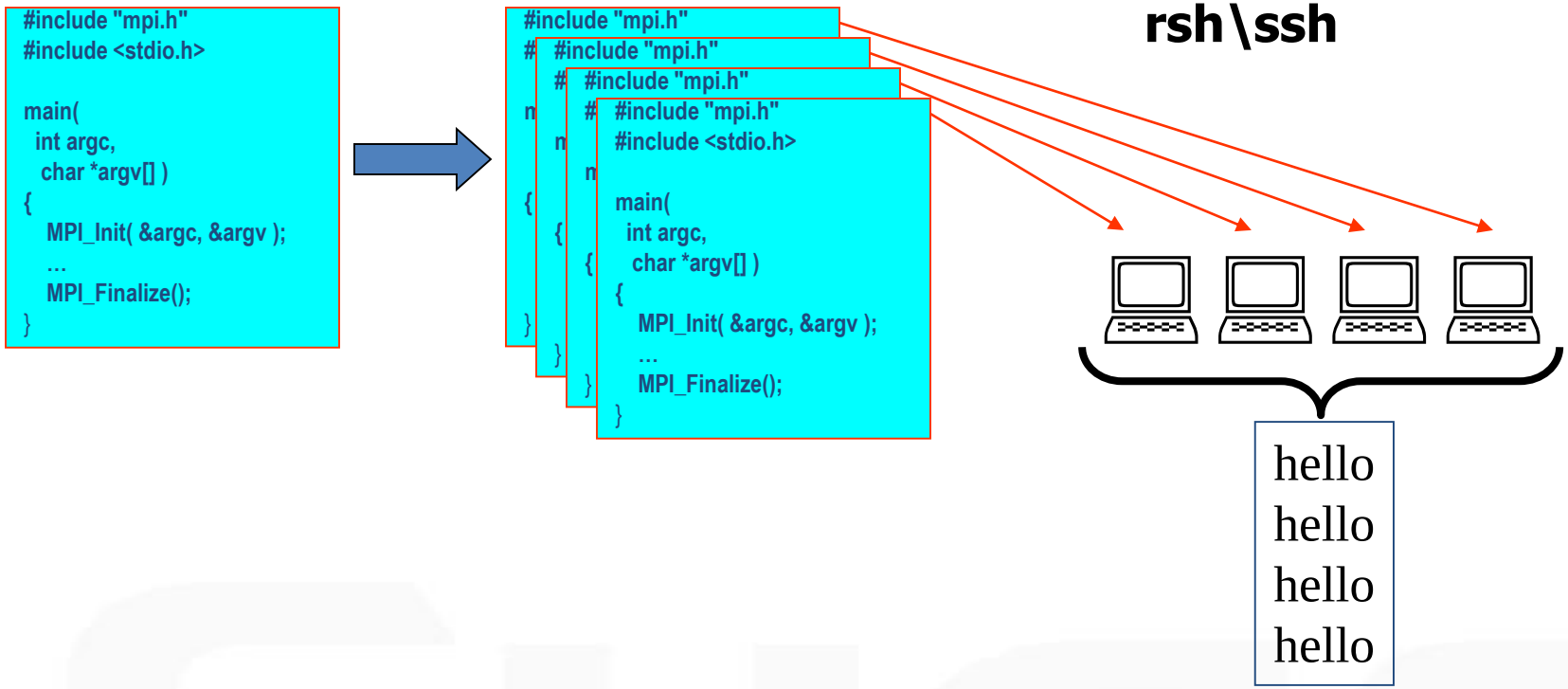
计算机打印字符

我们输入的命令

MPI并行编程

hello是如何被执行的?

- SPMD: Single Program Multiple Data 单程序多数据



MPI原生数据类型

MPI C语言数据类型		MPI Fortran语言数据类型	
MPI_CHAR	signed char	MPI_CHARACTER	character(1)
MPI_SHORT	signed short int		
MPI_INT	signed int	MPI_INTEGER	integer
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_REAL	real
MPI_DOUBLE	double	MPI_DOUBLE_PRECISION	double precision
MPI_LONG_DOUBLE	long double		
		MPI_COMPLEX	complex
		MPI_LOGICAL	logical
MPI_BYTE	8 binary digits	MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack	MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

并行计算简介

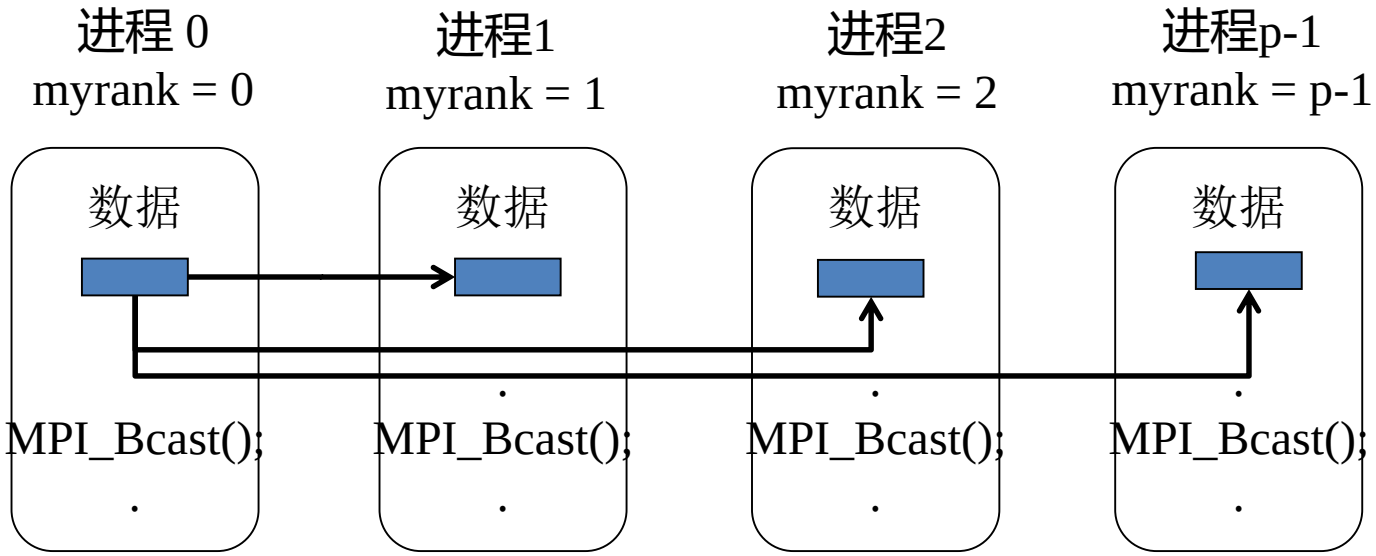
MPI并行编程

- MPI简介
- MPI程序基础架构
- MPI基本函数
  - 4个基本函数
- MPI基本通信函数
- MPI程序编译、运行
- 集合通信
  - 数据广播、收集、散发、转置、归约
  - 综合实例：并行计算π 值

MPI软件包的安装、使用

- OpenMPI, Mpich2, mvapich2

Broadcast -- 数据广播




```
int MPI_Bcast (
            void *buffer,/*发送/接收buf*/
            int  count,  /*元素个数*/
            MPI_Datatype datatype,
            int root,    /*指定根进程*/
            MPI_Comm comm)
```

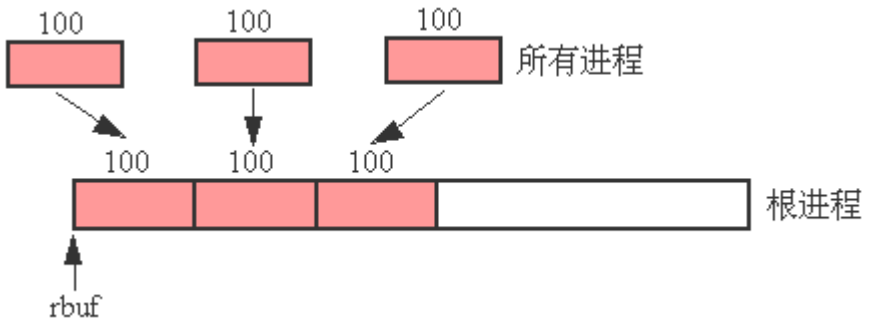
根进程既是发送缓冲区也是接收缓冲区

bcast示例

```
#include<iostream>
#include"mpi.h"
using namespace std;
int main(int argc, char **argv)
{
    int array[2];
    int myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    array[0] = array[1] = myrank;
    cout << "b rank " << myrank << ":  " ;
    cout << array[0] << "  " << array[1] << endl;
    int root = 0;
    MPI_Bcast(array,2,MPI_INT, root, MPI_COMM_WORLD);
    cout << "a rank " << myrank << ":  " ;
    cout << array[0] << "  " << array[1] << endl;
    MPI_Finalize();
    return 0;
}
```

```
$ mpirun -np 4 ./bcast.exe
b rank 0:  0  0
a rank 0:  0  0
b rank 3:  3  3
b rank 1:  1  1
a rank 1:  0  0
b rank 2:  2  2
a rank 2:  0  0
a rank 3:  0  0
```

数据收集-gather




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

Gather示例

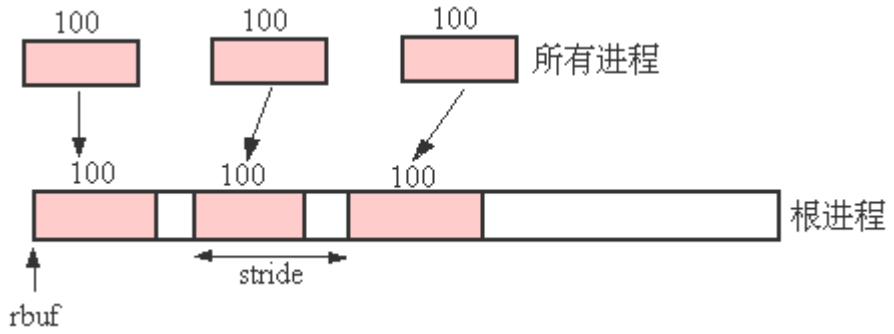
```
#include<iostream>
#include"mpi.h"
using namespace std;
int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  int myrank, gsize;
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &gsize);

  int sendarray[2];
  sendarray[0] = sendarray[1] = myrank;
  cout << “rank " << myrank << ":  " ;
  cout << sendarray[0] << "  " << sendarray[1] << endl;
```

```
 int root = 0;
  int *rbuf;
  if(myrank == root) rbuf = new int[gsize*2];
  MPI_Gather(sendarray, 2, MPI_INT, rbuf, 2, MPI_INT, root,
MPI_COMM_WORLD);

  if(myrank == root)
  {
    cout << "sendarray = " << endl;
    for(int i=0; i < gsize; i++)
      cout << rbuf[i*2] << "  " << rbuf[i*2 +1] <<  "  ";
    cout << endl;
    delete [] rbuf;
  }
  MPI_Finalize();
 return 0;
}
```

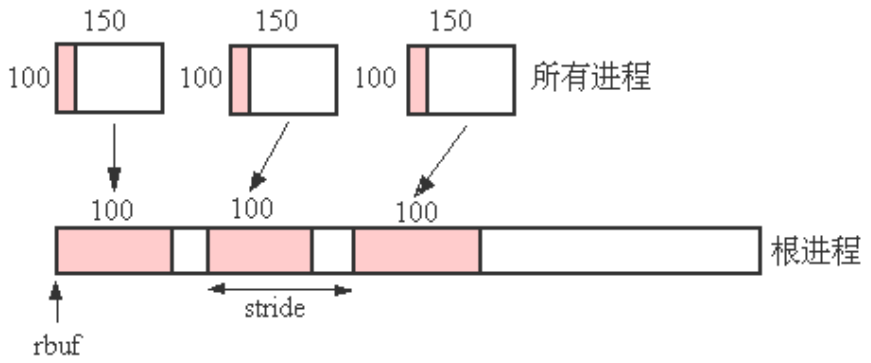
Gather算例输出

```
$ mpirun -np 4 ./gather.exe
rank 0:  0  0
rank 2:  2  2
rank 1:  1  1
rank 3:  3  3
sendarray =
0  0  1  1  2  2  3  3
```

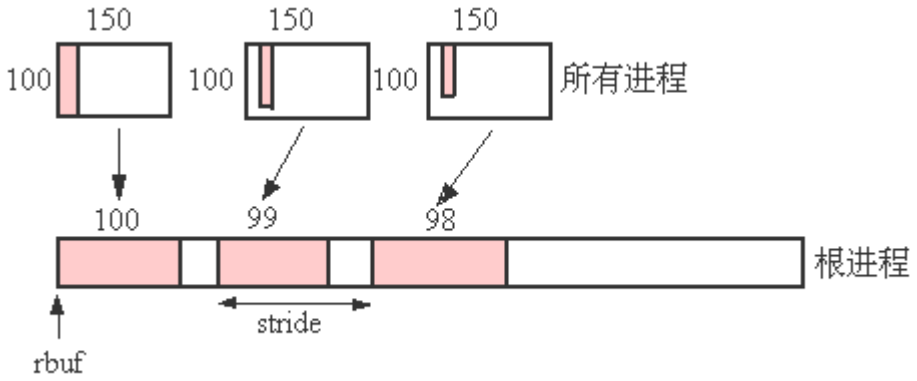
更强收集 MPI_Gatherv



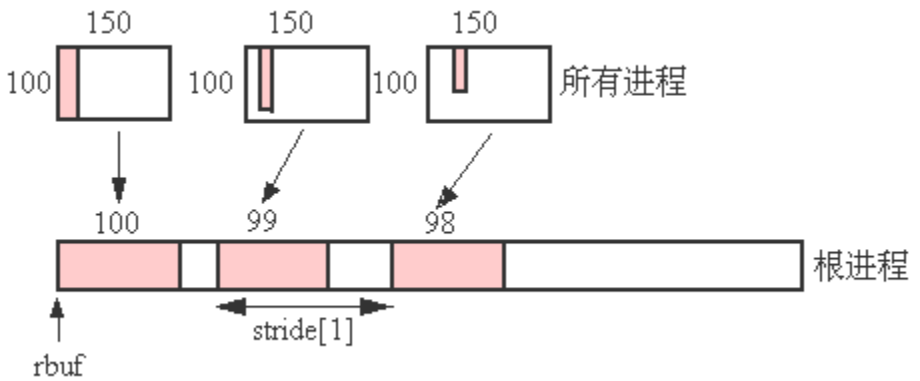
根进程从组中的每个其他进程收集100个整型数据,每个数据集之间按一定步长分开存放



根进程从100*150的数组中收集第0列,每个数据集之间按一定步长分开存放

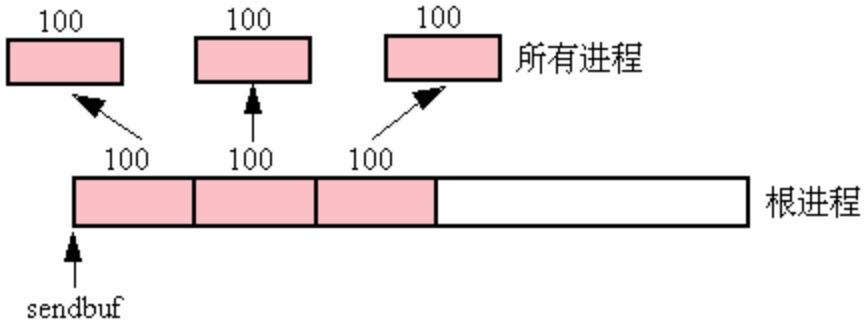


根进程从100*150的数组中收集第i列的100-i个整型数据,每个数据集之间按一定步长分开存放



根进程从100*150的数组中收集第i列的100-i个整型数据,每个数据集之间按步长stride[i]分开存放

数据散发-scatter



MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

sendbuf	待发送数据首地址
sendcounts	待发送数据数量
sendtype	待发送数据的类型
recvbuf	接收到数据的存放位置
recvcount	接收的数据数量
recvtype	接收数据的类型
root	根进程号
comm	通信器名字

scatter C接口

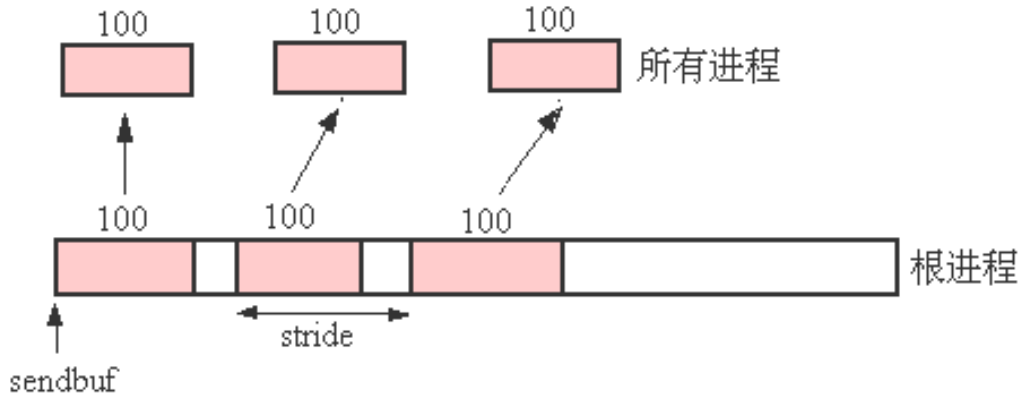
```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm
comm)
```

```
#include<iostream>
#include"mpi.h"
using namespace std;
int main(int argc, char **argv)
{
    int myrank, gsize, *sendbuf;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &gsize);
    int rbuf[2];
    rbuf[0] = rbuf[1] = myrank;
    cout<<"rank"<<myrank<<":"<<rbuf[0]<<"  "<<rbuf[1]<<endl;
    int root = 0;
    if(myrank ==root)
    {
        sendbuf = new int[gsize*2];
        for(int i=0; i<gsize*2; i++) sendbuf[i] = 100;
    }
```

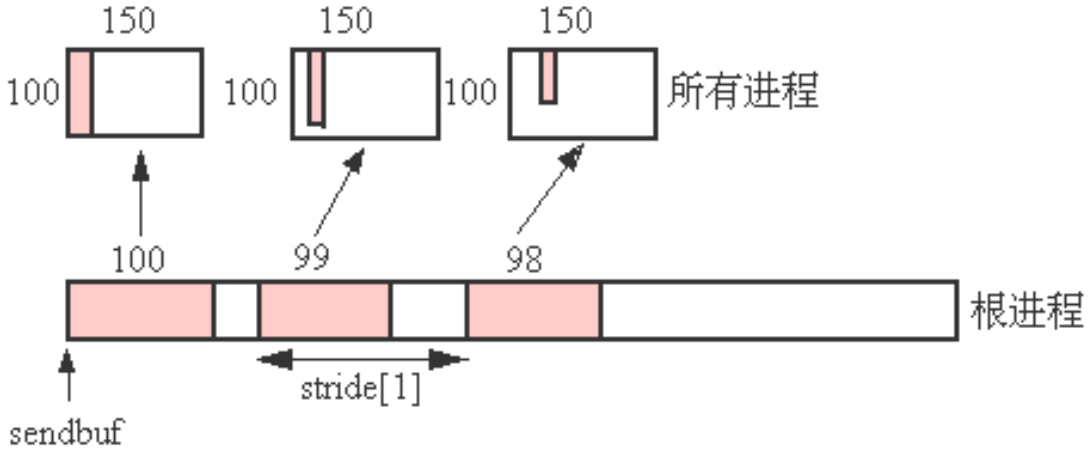
```
    MPI_Scatter(sendbuf, 2, MPI_INT, rbuf, 2,
MPI_INT, root, MPI_COMM_WORLD);

    cout << "Arank" << myrank << ":" << rbuf[0] << "
" << rbuf[1] << endl;
    if(myrank == root) delete [] sendbuf;
    MPI_Finalize();
    return 0;
}
```

```
$ mpirun -np 4 ./scatter.exe
rank0:0  0
Arank0:100  100
rank1:1  1
Arank1:100  100
rank2:2  2
Arank2:100  100
rank3:3  3
Arank3:100  100
```

更强分发-scatterv

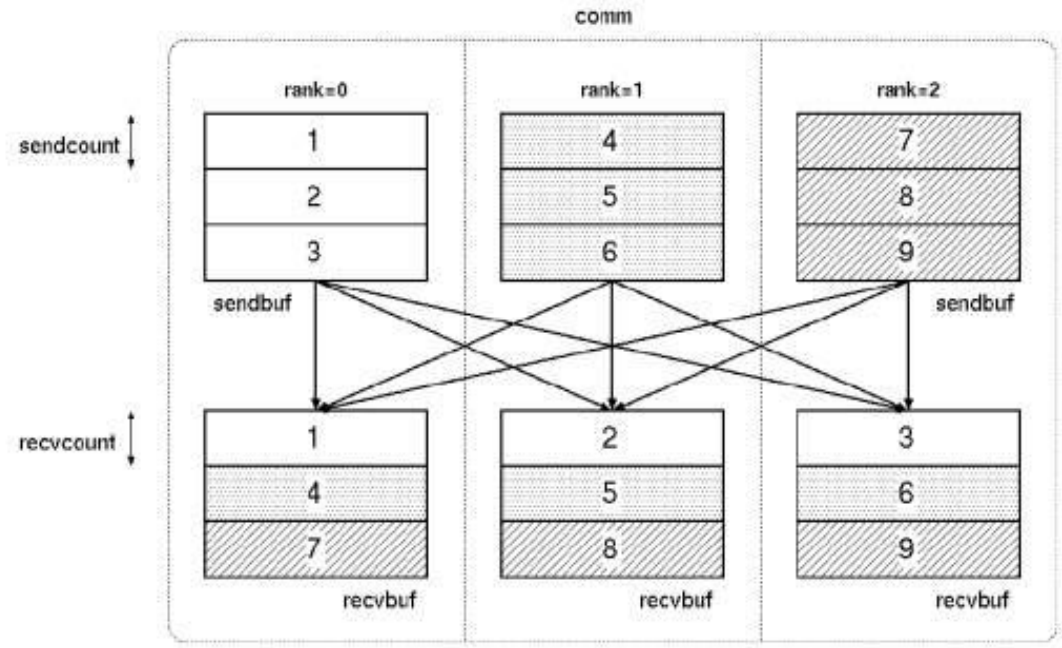


根进程分散100个整型数据,在分散时按步长stride取数据



根进程分散100*150数组的第i列中100-i个整型数据块.发送端的步长分别为stride[i]

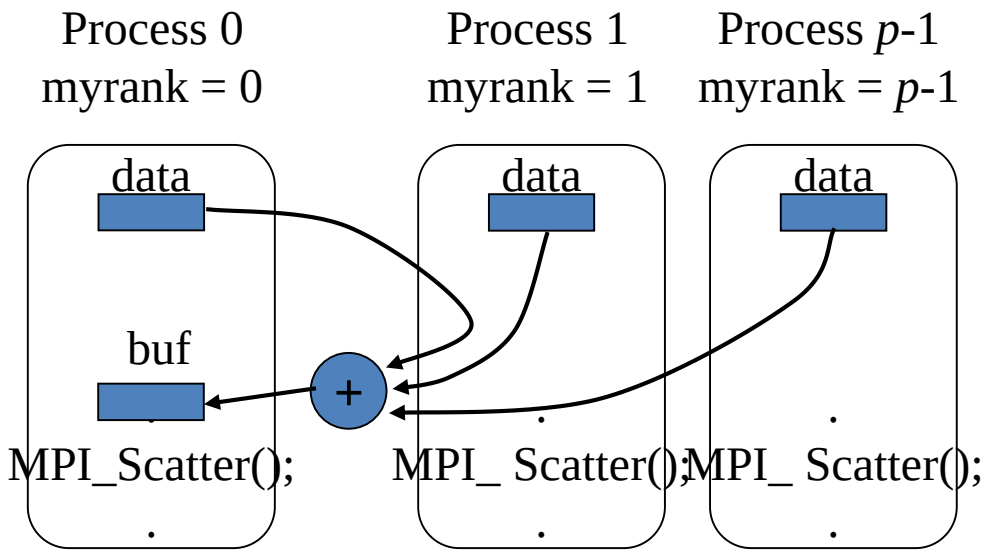
Alltoall -- 数据转置



通信域中所有进程从其他进程收集数据，同时将自己的数据散发给其他进程.

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

全局归约reduce



对组中所有进程的发送缓冲区中的数据用OP参数指定的操作进行运算,并将结果送回到根进程的接收缓冲区中.

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

支持的归约操作运算

操作符	含义
MPI_MAX	最大
MPI_MIN	最小
MPI_SUM	求合
MPI_PROD	乘积
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	max value and location
MPI_MINLOC	min value and location
用户自定义	自定义

示例：向量内积

两个长度为N的向量a[i]、b[i] 作内积:
S=a[0]*b[0] + a[1]*b[1] + … + a[N-1]*b[N-1]

向量a[:]和b[:]分布在多个进程上

```
#include<iostream>
#include <ctime>
#include <cstdlib>
#include"mpi.h"
using namespace std;
int main(int argc, char **argv)
{
    int myrank, gsize, *sendbuf;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &gsize);
    int a[10], b[10];
    double sum_part, sum;
    srand(unsigned(time(0)+myrank));
    sum_part = 0.0;
```

```
for(int i=0; i<10; i++)
{
    a[i] = (random()%10);
    b[i] = (random()%4);
    sum_part += a[i]*b[i];
}
cout << "sum_part = " << sum_part << endl;
int root =0;
MPI_Reduce(&sum_part, &sum, 1, MPI_DOUBLE,  MPI_SUM,
root, MPI_COMM_WORLD);
if(myrank==root) cout<<"sum="<<sum<<endl;
MPI_Finalize();
return 0;
}
```

```
$ mpirun -np 4 ./reduce.exe
sum_part = 50
sum_part = 49
sum_part = 76
sum_part = 93
sum = 268
```

实例: 求π

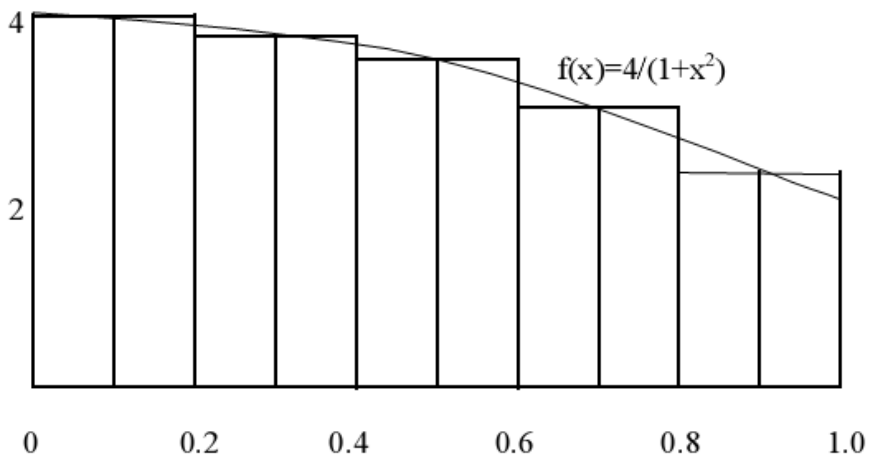
$$\int_0^1 \frac{1}{1+x^2}dx = \arctan(x)\Big|_0^1 = \arctan(1) - \arctan(0) = \arctan(1) = \frac{\pi}{4}$$

令函数 $f(x) = 4/(1+x^2)$

则有 $\int_0^1 f(x)dx = \pi$

$$\pi \approx \sum_{i=1}^{n} f(\frac{2\times i-1}{2\times N})\times\frac{1}{N}$$

$$= \frac{1}{N}\times\sum_{i=1}^{n} f(\frac{i-0.5}{N})$$



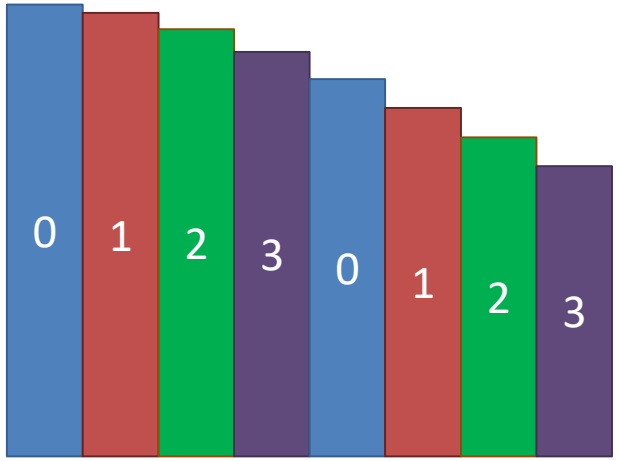
串行代码

```
#include <iostream>
using namespace std;
double f(double a){
        return (4.0/(1.0+a*a));
}
int main(void)
{
        int n = 100;
        double h, x, pi, sum=0.0;
        h=1.0/(double)n;
        for(int i=1; i<=n; i++)
        {
                x=h*((double)i - 0.5);
                sum += f(x);
        }
        pi=h*sum;
        cout << "pi = " <<  pi << endl;
 return 0;
}
```

并行代码

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

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

int main(int argc, char *argv[])
{
    int n, myrank, nprocs;
    double PI25DT = 3.14159265358979238;
    double mypi, pi, h, sum, x;
    double startwtime=0.0, endwtime;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    n = 100;
    if(0 == myrank) startwtime = MPI_Wtime();
```

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    h = 1.0 / (double)n;
    sum = 0.0;
    for(int i = myrank + 1; i <= n; i += nprocs)
    {
        x = h * ((double)i - 0.5);
        sum += f(x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
    if(0 == myrank)
    {
        cout << "mypi-pi =" << pi-PI25DT << endl;
        endwtime = MPI_Wtime();
        cout << "wtime =" << endwtime-startwtime << endl;
    }
    MPI_Finalize();
    return 0;
}
```



并行计算简介

MPI并行编程

- MPI简介
- MPI程序基础架构
- MPI基本函数
  - 4个基本函数
- MPI基本通信函数
- MPI程序编译、运行
- 集合通信
  - 数据广播、收集、散发、转置、归约
  - 综合实例：并行计算π 值

MPI软件包的安装、使用

- OpenMPI, Mpich2, mvapich2

MPI编译工具安装-OpenMPI

稳定版1.6，http://www.open-mpi.org/software/ompi/v1.6/

```
$ tar zxvf openmpi-1.6.tar.gz
$ cd openmpi-1.6
$ ./configure --prefix=/public/software/mpi/ompi-1.6-
gnu CC=gcc CXX=g++ FC=gfortran F77=gfortran
$ make -j 8
$ make install
```

intel编译器版

```
$ ./configure CC=icc CXX=icc F77=ifort FC=ifort …
```

创建环境变量文件/public/software/profile.d/ompi-1.6-gnu.sh

```
#!/bin/bash

MPI_HOME=/public/software/mpi/openmpi-1.6-gnu
export PATH=${MPI_HOME}/bin:$PATH
export LD_LIBRARY_PATH=${MPI_HOME}/lib:$LD_LIBRARY_PATH
export MANPATH=${MPI_HOME}/share/man:$MANPATH
unset MPI_HOME
```

OpenMPI编译命令

代码语言	命令
C, .c文件	mpicc
C++, .cpp文件	mpicxx, mpic++
Fortran77, .f .for文件	mpif77
Fortran90, .f90文件	mpif90
运行作业	mpirun, mpiexec

mpirun用法

```
$ mpirun -np 4 -hostfile hostfile program
```

-np X：运行X个进程

--hostfile：指定计算节点

hostfile形式一：每行一个结点

```
node1
node1
node2
node2
```

hostfile形式二：指定重复次数

```
node1 slots=2
node2 slots=2
```

选择通信网络

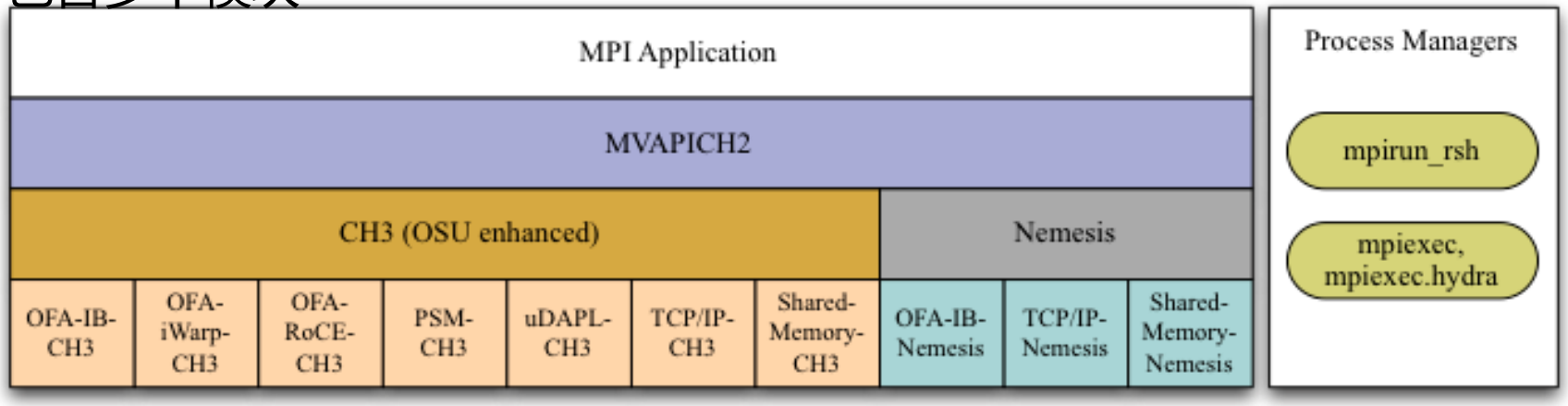
节点内使用共享内存，节点间使用InfiniBand通信

```
$ mpirun -np 12 -hostfile hosts --mca btl
self,sm,openib ./program
```

选项	含义
--mca orte_rsh_agent rsh	指定节点间通信使用rsh，默认为ssh
--mca btl self,tcp	以太网使用以太网TCP/IP协议通信
--mca btl self,sm	单节点运行时使用共享内存，效率较高
--mca btl self,openib	有Infiniband设备时，使用IB通信
--mca btl_tcp_if_include eth0	以太网通信时使用eth0接口，默认使用所有接口

mvapich2简介

包装mpich2，兼容性好
添加支持InfiniBand
最新版本1.8，支持MPI2.2标准
支持GPU通信，结点内、跨结点
包含多个模块



Mvapich2安装

同时编译IB版和TCP/IP版

```
$ tar zxvf mvapich2-1.7rc1.tgz
$ cd mvapich2-1.7rc1
$ ./configure --prefix=/public/software/mpi/mvapich2-17rc1-gnu CC=gcc CXX=g++ FC=gfortran F77=gfortran --with-device=ch3:nemesis:ib,tcp
$ make && make install
```

编译IB版（显式选择）

```
$ ./configure --with-device=ch3:mrail --with-rdma=gen2
```

编译IB版（默认选择）

```
$ ./configure
```

编译IB+GPU版

```
$ ./configure --enable-cuda
```

编译TCP/IP版

```
$ ./configure --with-device=ch3:sock
```

Mvapich2安装(2)

编译IB+GPU+PGI编译器

```
$ ./configure CC=pgcc CXX=pgCC FC=pgf90 F77=pgf77  --
enable-cuda CPPFLAGS="-D__x86_64
-D__align__\(n\)=__attribute__\(\(aligned\(n\)\)\)-
D__location__\(a\)=__annotate__\(a\)
-DCUDARTAPI="
```

Mvapich2编译命令

代码语言	命令
C, .c文件	mpicc
C++, .cpp文件	mpicxx, mpic++
Fortran77, .f .for文件	mpif77
Fortran90, .f90文件	mpif90
运行作业	mpirun_rsh(首选)，mpirun

mpirun_rsh命令

在n0,n1两个结点上运行4个进程，默认SSH通信

```
$ mpirun_rsh -np 4 n0 n0 n1 n1 ./cpi
```

在n0,n1两个结点上运行4个进程，改用RSH通信

```
$ mpirun_rsh -rsh -np 4 n0 n0 n1 n1 ./cpi
```

使用hostfile

```
$ mpirun_rsh -np 4 -hostfile hosts ./cpi
```

hostfile格式

```
# 允许使用注释
host1       # rank 0将放在host1
host2:2     # rank 1和2将放在host2
host3       # rank 3放在host3
host4:4     # ranks 4至7放在host4
# 若进程数大于8，剩作的进程将循环分配给这些结点。例如rank 8放在host1
# rand 9和rank 10放在host2.
```

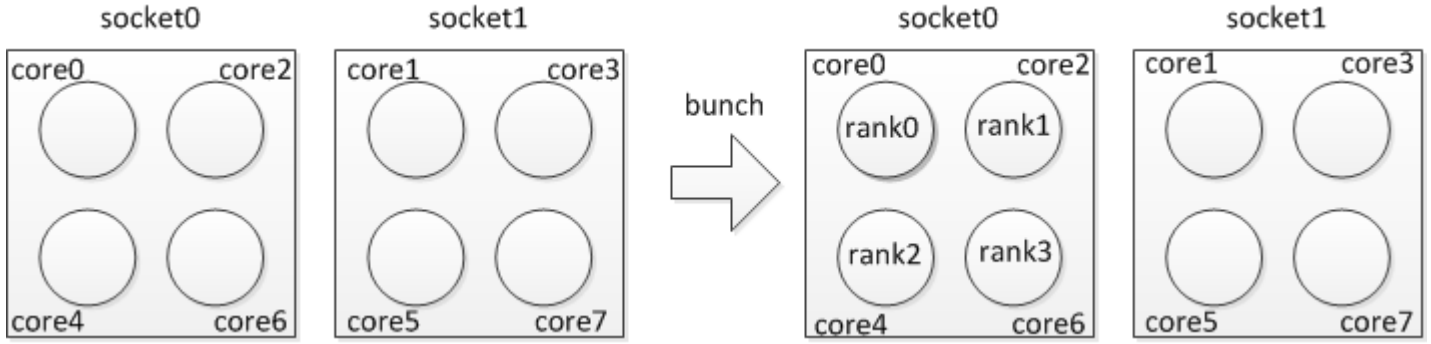
指定环境变量（需紧邻可执行文件）

```
$ mpirun_rsh -np 4 -hostfile hosts ENV1=value
ENV2=value ./cpi
```

按核进程绑定

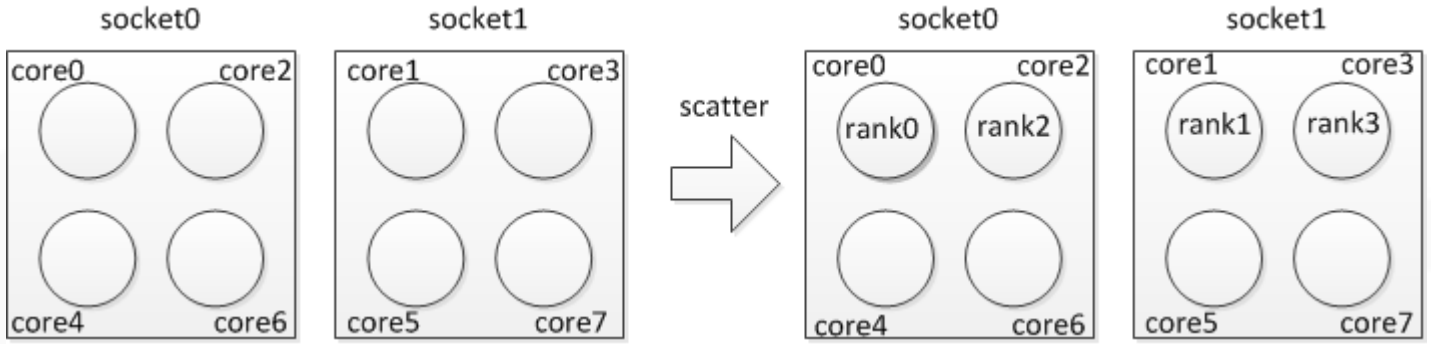
bunch绑定

```
$ mpirun_rsh -np 4 -hostfile hosts
MV2_CPU_BINDING_POLICY=bunch ./a.out
```



scatter绑定

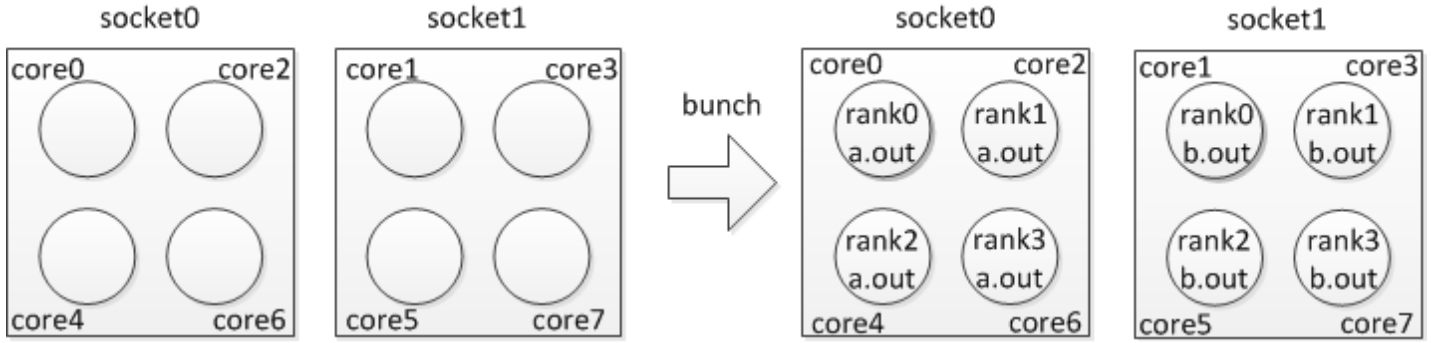
```
$ mpirun_rsh -np 4 -hostfile hosts
MV2_CPU_BINDING_POLICY=scatter ./a.out
```



按核进程绑定（续）

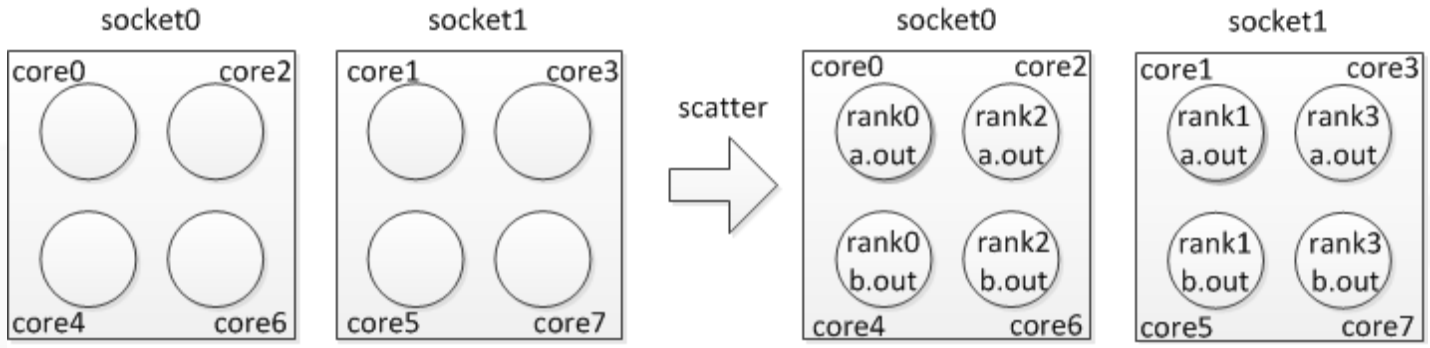
bunch绑定(多作业)

```
$ mpirun_rsh -np 4 -hostfile hosts MV2_CPU_BINDING_POLICY=bunch ./a.out
$ mpirun_rsh -np 4 -hostfile hosts MV2_CPU_BINDING_POLICY=bunch ./b.out
```



scatter绑定(多作业)

```
$ mpirun_rsh -np 4 -hostfile hosts MV2_CPU_BINDING_POLICY=scatter ./a.out
$ mpirun_rsh -np 4 -hostfile hosts MV2_CPU_BINDING_POLICY=scatter ./a.out
```



计算 决定未来

THANKS