消息传递接口MPI:聚合通信(Collective Communication)

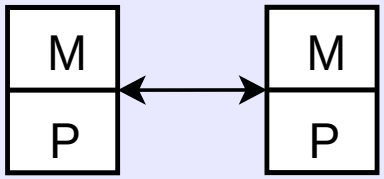
陈俊清

jqchen@math.tsinghua.edu.cn

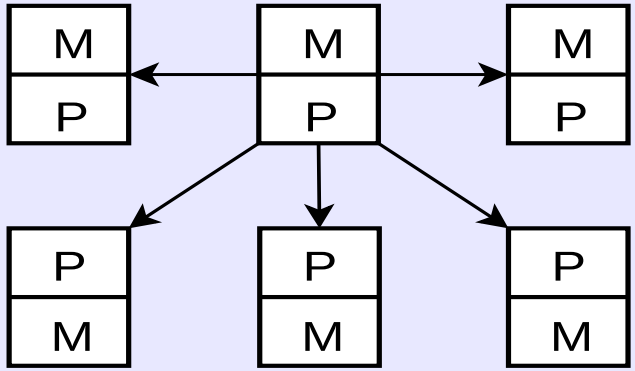
清华大学数学科学系

March 29, 2013

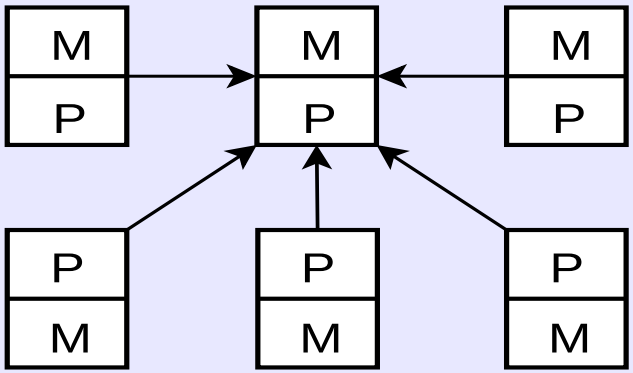
常见消息传递模式



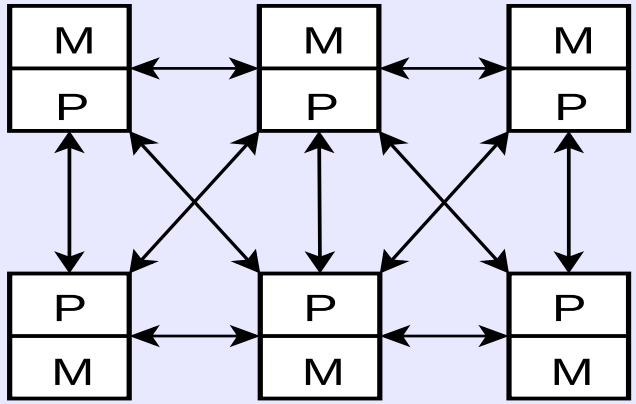
(a) 点对点通信



(b) 一对多通信（广播、散发）



(c) 多对一通信（收集、归约）



(d) 多对多通信（全收集、全交换、全归约、归约分发）

聚合通信

聚合通信函数包括障碍同步（barrier synchronization）、广播（broadcast）、数据收集（gather）与散发（scatter）、归约(reduce)等等。 这些函数均要求属于同一进程组（通信器）的进程共同参与、协同完成。
聚合通信函数根据数据的流向可分为一对多（一个进程对多个进程，如广播散发）、多对一（多个进程对一个进程，如数据收集、归约） 和多对多（多个进程对多个进程）三种类型的操作。 在一对多和多对一操作中，有一个进程扮演着特殊的角色，称为该操作的根进程（root）。

聚合通信

特点:

- 全局性，即同一通信器内所有进程都参与;
- 所有进程的函数调用形式相同，但部分参数意义不同;
- 阻塞通信方式;
- 不需要"tag“ 参数;
- 实现功能：通信、同步、计算;
- 通信模式：一对多、多对一、多对多。

MPI 聚合通信（II）

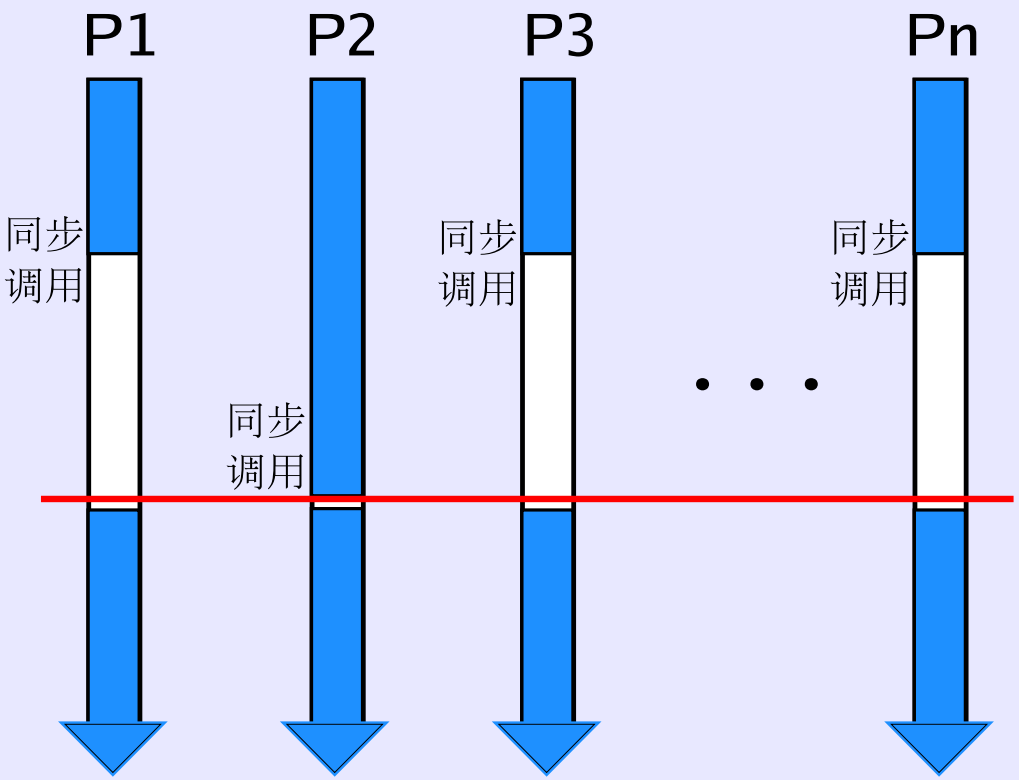
- 全局通信函数
  广播：MPI_Bcast()
  散发：MPI_Scatter()
  收集：MPI_Gather()
  全收集：MPI_Allgather()
  全交换：MPI_Alltoall()
- 全局归约
  归约：MPI_Reduce()
  全规约：MPI_Allreduce()
  归约散发：MPI_Reduce_scatter()
  前缀归约：MPI_Scan()
- 同步函数：MPI_Barrier()

Outline

障碍同步

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

该函数用于进程间的同步。一个进程调用该函数后将等待直到通信器comm中的所有进程都调用了 该函数才返回（see ex5-00.c）。



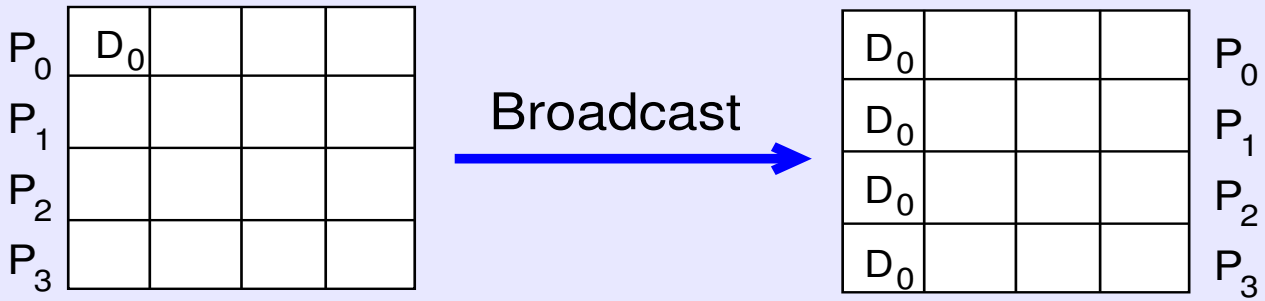
Outline

广播

```
int MPI_Bcast(void *buffer, int count,
        MPI_Datatype datatype, int root,
        MPI_Comm comm)
```

通信器comm中进程号为root的进程（根进程）将自己buffer中的内容同时发送给通信器中所有其它进程。(参看例子程序：ex5-0.c)

广播



直观上说，假设nprocs为通信器comm中的进程数，myrank为进程在通信器comm中的进程号，则MPI_Bcast相当于：

```
if(myrank  == root){
    for(i=0;i<nprocs;i++)
        if(i!=root)
            MPI_Send(buffer, count, datatype, i,..);
}
else
    MPI_Recv(buffer, count, datatype, root,...)
```

Outline

数据收集

数据收集是指各个进程（包括根进程）将自己的一块数据发给根进程，根进程将这些数据合并成一个大的数据块。
收集相同长度数据块MPI_Gather:

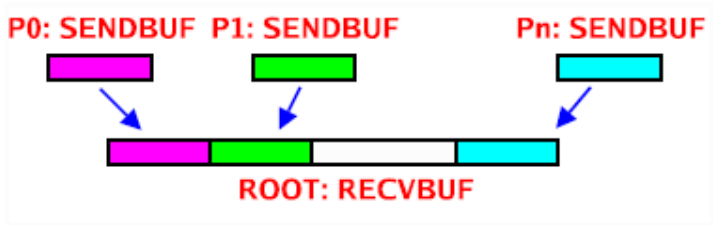
```
int MPI_Gather(void *sendbuf, int sendcount,
        MPI_Datatype sendtype,void *recvbuf,
        int recvcount, MPI_Datatype recvtype, int root,
        MPI_Comm comm)
```

所有进程（包括根进程）将sendbuf中的数据发送给根进程，根进程将这些数据按进程号的顺序依次收到recvbuf中。发送和接收的数据类型与长度必须相配，即发送和接收用的数据类型必须有相同的类型序列。参数recvbuf，recvcount和recvtype仅对根进程有意义。(参看例子程序ex5-1.c, ex5-2.c, ex5-3.c)

收集相同长度数据块MPI_Gather

假设nprocs为通信器中的进程数，myrank为进程号，则MPI_Gather相当于:

```
MPI_Send(sendbuf, sendcount, sendtype, root, ...);
if(myrank == root){
    for(i=0;i<nprocs;i++)
        MPI_Recv(recvbuf+I*recvcount*extent(recvtype),
                 recvcount, recvtype,i,...)
}
```



收集不同长度的数据

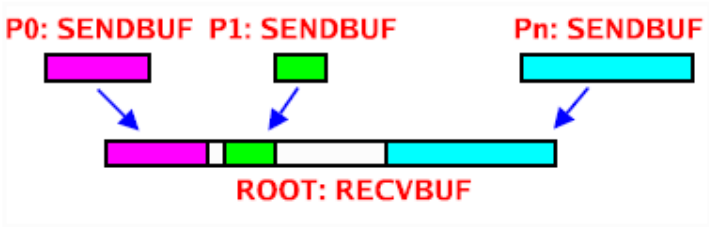
```
int MPI_Gatherv(void *sendbuf,int sendcount,
        MPI_Datatype sendtype, void *recvbuf,
        int *recvcounts,int *displs,
        MPI_Datatype recvtype, int root, MPI_Comm comm)
```

与MPI_Gather类似，但允许每个进程发送的数据长度不同，并且根进程接收时也不一定将他们连续存放。 recvbuf, recvtype, recvcounts和displs仅对根进程有意义。数组recvcounts和displs的元素个数等于进程数， 分别给出从每个进程接收的数据长度和位移（以recvtype的域为单位）。(参看例子程序 ex5-4.c,ex5-5.c,ex5-6.c)

收集不同长度的数据

假设nprocs为通信器中的进程数，myrank为进程号，则MPI_Gatherv相当于:

```
int displs[nprocs],recvcounts[nprocs];
MPI_Send(sendbuf, sendcount, sendtype, root, ...);
if (myrank == root){
    for(i=0;i<nprocs;i++)
        MPI_Recv(recvbuf+displs[i]*extent(recvtype),
                 recvcounts[i],recvtype,i,...)
}
```



全收集MPI_Allgather

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

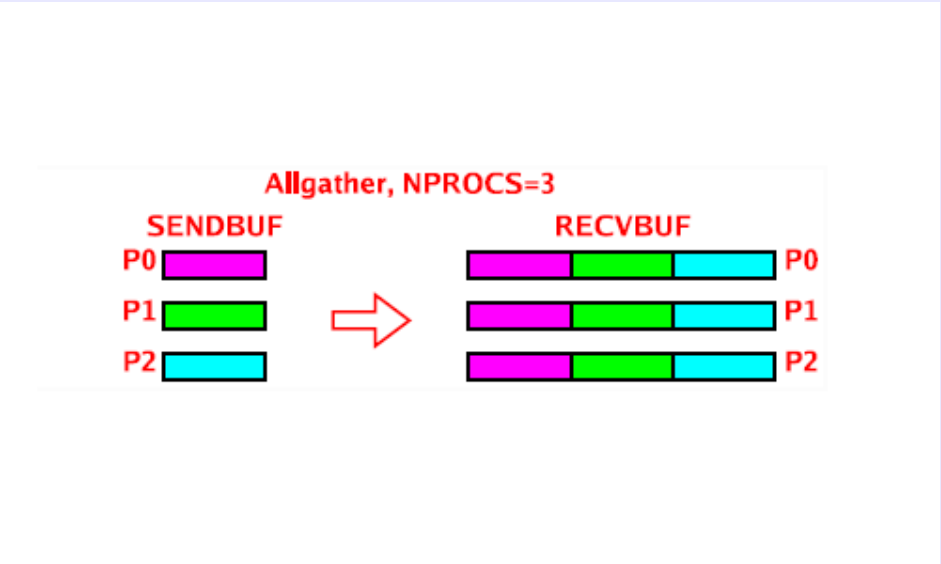
MPI_Allgather与MPI_Gather类似，区别是所有进程同时将数据收集到recvbuf中，因此称为数据全收集。 MPI_Allgather等价于依次以每个进程为根进程调用nprocs次普通数据收集函数MPI_Gather:

```
for(i=0;i<nprocs;i++)
    MPI_Gather(sendbuf, sendcount, sendtype,recvbuf,
            recvcount,recvtype, i, comm);
```

全收集MPI_Allgather

也可以认为MPI_Allgather相当于以任一进程为根进程调用一个普通收集，紧接着再对收集到的数据进行一次广播，例如：

```
root = 0;
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf,
        recvcount, recvtype, root, comm);
MPI_Bcast(recvbuf, recvcount*nprocs, recvtype, root,
        comm);
```



不同长度数据块的全收集MPI_Allgatherv

```
int MPI_Allgatherv(void *sendbuf, int sendcout,
        MPI_Datatype sendtype,void *recvbuf,
        int *recvcounts, int *displs,
        MPI_Datatype,recvtype, MPI_Comm comm)
```

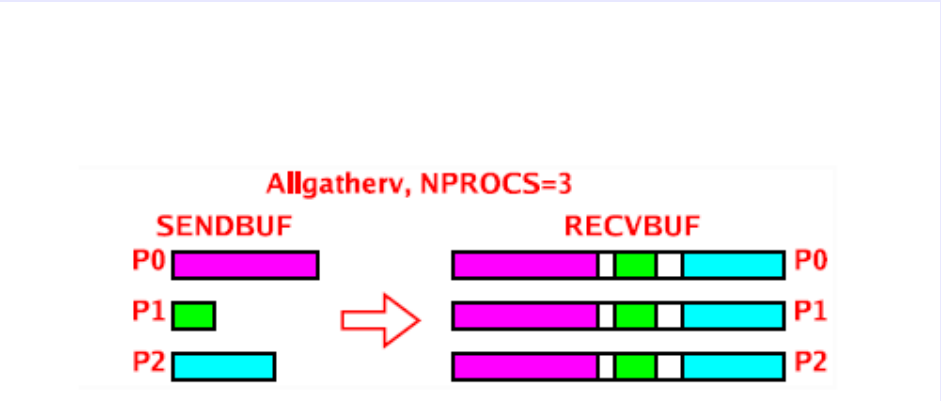
MPI_Allgatherv用于不同长度数据块的全收集。它的参数与MPI_Gatherv类似。
MPI_Allgatherv等价于一次以每个进程为根进程调用nprocs次MPI_Gatherv:

```
for(i=0;i<nprocs;i++)
    MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf,
            recvcounts,displs,recvtype,i,comm)
```

不同长度数据块的全收集MPI_Allgatherv

也可认为MPI_Allgatherv相当于以任一进程为根进程调用一次普通收集，紧接着再对收集到的数据进行一次广播，例如：

```
root == 0;
MPI_Gatherv(sendbuf, sendcout, sendtype, recvbuf,
        recvcounts, displs,recvtype, root, comm);
MPI_Type_indexed(nprocs, recvcounts, displs,recvtype,
        newtype);
MPI_Type_commit(newtype);
MPI_Bcast(recvbuf,1,newtype,root,comm);
MPI_Type_free(newtype)
```



Outline

数据散发

数据散发指根进程将一个大的数据块分成小块分别散发给各个进程（包括根进程自己）。它是数据收集的逆操作。
散发相同长度的数据块MPI_Scatter:

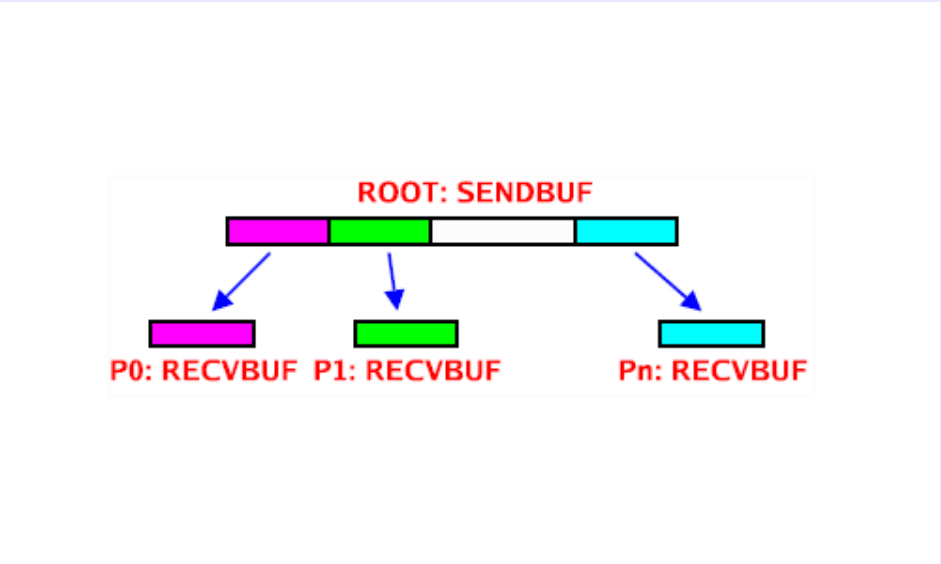
```
int MPI_Scatter(void *sendbuf, int sendcount,
        MPI_Datatype sendtype, void *recvbuf,
        int recvcount, MPI_Datatype recvtype, int root,
        MPI_Comm comm)
```

根进程root的sendbuf中包含有nprocs个连续存放的数据块，每个数据块包含sendcount个类型为sendtype的数据， nprocs为通信器中的进程数。根进程将这些数据块按进程的序号依次分发给各个进程（包括根进程自己）。 参数sendbuf，sendcount和sendtype仅对根进程有意义。(参看例子程序：ex5-7.c)

数据散发MPI_Scatter

假设myrank为进程号，则MPI_Scatter相当于：

```
if(myrank == root){
    for(i=0;i<nprocs;i++)
        MPI_Send(sendbuf+I*sendcount*extent(sendtype),
                 sendcount,senttype,i,...);
}
MPI_Recv(recvbuf, recvcount, recvtype,root,...)
```



散发不同长度的数据块MPI_Scatterv

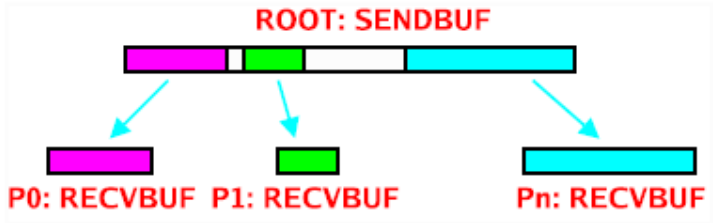
```
int MPI_Scatterv(void *sendbuf, int *sendcounts,
        int *displs,MPI_Datatype sendtype,
        void *recvbuf, int recvcount,
        MPI_Datatype recvtype, int root, MPI_Comm comm)
```

与MPI_Scatter类似，但允许发送的每个数据块的长度不同，并且在sendbuf中不一定连续存放。 sendbuf，sendtype，sendcounts和displs仅对根进程有意义。数组sendcounts和displs的元素个数等于进程个数， 他们分别给出发送给每个进程的数据长度和位移（以sendtype为单位）。

散发不同长度的数据块MPI_Scatterv

假设nprocs为通信器中的进程数，myrank为进程号，则MPI_Scatterv 相当于:

```
displs[nprocs],sendcounts[nprocs];
   ... ...
if(myrank == root)
    for(i=0;i<nprocs;i++)
        MPI_Send(sendbuf+displs[i]*extent(sendtype),
                 sendcounts[i],sendtype,i,...);
MPI_Recv(recvbuf, recvcount, recvtype, root, ...);
```



全部进程对全部进程的数据散发收集

每个进程散发自己的一个数据块，并且收集拼装所有进程散发过来的数据块。我们称该操作为数据的“ 全散发收集”。 它既可以被认为是数据全收集的扩展，也可以被认为数据散发的扩展。
相同数据长度的全收集散发MPI_Alltoall:

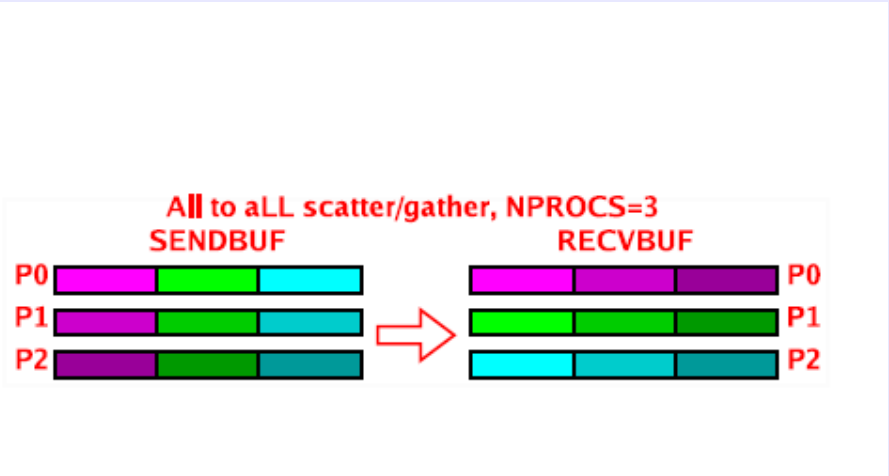
```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype sendtype, void *recvbuf,
        int recvcount, MPI_Datatype recvtype,
        MPI_Comm comm)
```

第i个进程将sendbuf中的第j块数据发送至第j个进程的recvbuf的第i个位置，i,j=0,. . . ,nprocs-1(nprocs为进程数)。 sendbuf和recvbuf均由nprocs个连续存放的数据块构成， 但他们每个数据块的长度/类型分别为sendbuf/sendtype和recvcount/recvtype。(参看例子程序ex5-8.c)

相同数据长度的全收集散发MPI_Alltoall

该操作相当于将数据/进程进行一次转置。例如，假设一个二维数组按行分块存储在各个进程中，则调用该函数可很容易地将它变成按列分块存储在各进程中。假设myrank为进程号，则MPI_Alltoall相当于：

```
for(i=0;i<nprocs;i++)
    MPI_Send(sendbuf + i*sendcount*extent(sendtype),
             sendcount, sendtype,i,...);
for(i=0;i<nprocs;i++)
    MPI_Recv(recvbuf+i*recvcount*extent(recvtype),
             recvcount,recvtype,i,...);
```



不同数据长度的全收集散发MPI_Alltoallv

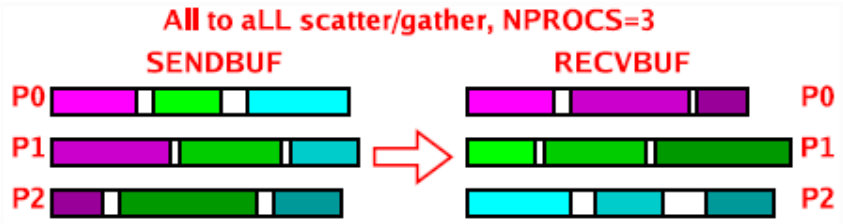
```
int MPI_Alltoallv(void *sendbuf, int *sendcounts,
        int *sdispls, MPI_Datatype sendtype,
        void *recvbuf, int *recvcounts, int *rdispls,
        MPI_Datatype recvtype, MPI_Comm comm)
```

与MPI_Alltoall类似，但每个数据块的长度可以不等，并且不要求连续存放。各个参数的含义很容易从MPI_Alltoall, MPI_Scatterv和MPI_Gatherv中的参数的含义得出。

不同数据长度的全收集散发MPI_Alltoallv

假设myrank为进程号，nprocs为进程数，则MPI_Alltoallv相当于：

```
for(i=0;i<nprocs;i++)
    MPI_Send(sendbuf + sdispls[i]*extent(sendtype),
             sendcounts[i],sendtype,i,...);
for(i=0;i<nprocs;i++)
    MPI_Recv(recv+rdispls[i]*extent(recvtype),
             recvcounts[i],recvtype,i,...)
```



Outline

归约

假设一个通信器中有p个进程，每个进程均有一个n个元素的数组。设$\{a_{i,k}, k = 1, \ldots, n\}$为第i个进程中的数组，$i = 0, \ldots, p-1$。又设$\oplus$为这些数组的二元运算，则相应的归约操作(reduction)结果定义为数组$\{res_k, k = 1, \ldots, n\}$，其中：

$$res_k = a_{0,k} \oplus a_{1,k} \ldots \oplus a_{p-1,k}.$$

MPI的归约函数要求$\oplus$满足结合律，但可以不满足交换律。运算可以是MPI预定义的，也可以是用户自行定义的。

归约函数MPI_Reduce

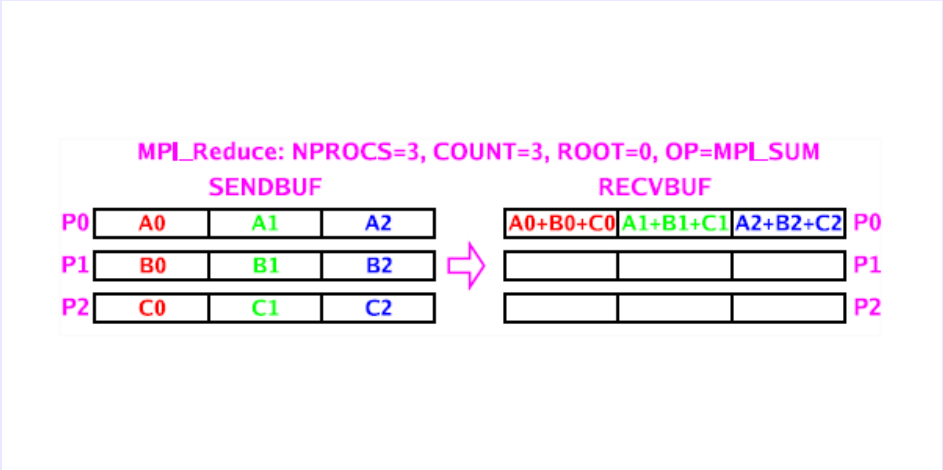
```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, int root,
        MPI_Comm comm)
```

设进程数为nprocs，则MPI_Reduce相当于在根进程(root)中计算：

```
for(k=1;k<count;k++){
    recv(k) = sendbuf(k) of process 0;
    for(i=1;i<nprocs;i++)
        recvbuf(k)=recvbuf(k) op(sendbuf(k)of process i);
}
```

函数中除op参数外，其它参数的含义是显而易见的。op指归约使用的运算， op在C语言中的类型为MPI_Op，可以是MPI预定义的，也可以是用户自定义的。(参看例子程序ex5-9.c)

归约函数MPI_Reduce



归约函数MPI_Reduce

下表给出MPI预定义的归约运算以及他们对数据类型的要求。表中的字节表示MPI_BYTE.

MPI运算符	含义	允许的数据类型
MPI_MAX	求最大	整型，实型
MPI_MIN	求最小	整型，实型
MPI_SUM	求和	整型，实型
MPI_PROD	求积	整型，实型
MPI_LAND	逻辑与	整型
MPI_BAND	二进制按位与	整型，字节
MPI_LOR	逻辑或	整型
MPI_BOR	二进制逻辑或	整型，字节
MPI_LXOR	逻辑异或	整型
MPI_BXOR	二进制按位异或	整型，字节
MPI_MAXLOC	最大值及位置	*
MPI_MINLOC	最小值及位置	*

归约函数MPI_Reduce

MPI_MINLOC和MPI_MAXLOC是两个特殊的运算，他们要求由数对（连续存放的两个数）构成一类特殊数据类型。MPI为它们定义了下面一些数据类型：

```
MPI_FLOAT_INT={float, int}
MPI_DOUBLE_INT={double, int}
MPI_LONG_INT={long, int}
MPI_2INT={int, int}
MPI_SHORT_INT={short, int}
MPI_LONG_DOUBLE_INT={long double, int}
```

设$x = (u, i), y = (v, j)$，则MPI_MINLOC$(x, y) = (w, k)$,其中：

$$w = \min(u, v), k = \begin{cases} i & if\ u < v; \\ min(i, j) & if\ u = v; \\ j & if\ u > v; \end{cases}$$

归约函数MPI_Reduce

MPI_MAXLOC:
设$x = (u, i), y = (v, j)$, 则MPI_MAXLOC$(x, y) = (w, k)$,其中:

$$w = \max(u, v), k = \begin{cases} i & if\ u > v; \\ min(i, j) & if\ u = v; \\ j & if\ u < v; \end{cases}$$

全归约MPI_Allreduce

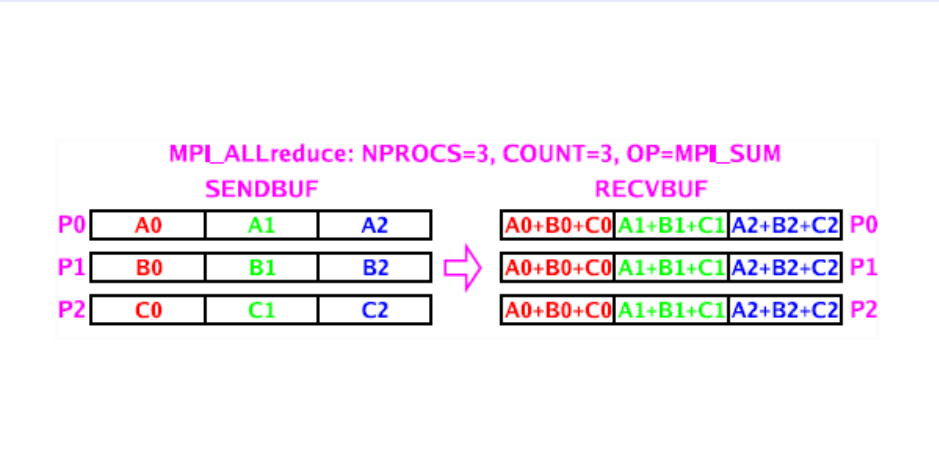
```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
        int count, MPI_Datatype datatype, MPI_Op op,
        MPI_Comm comm)
```

全归约函数与普通归约函数的操作类似，但所有进程将同时获得归约运算的结果。MPI_Allreduce除了比MPI_Reduce少了一个root参数外，其他参数及含义与后者一样。

全归约MPI_Allreduce

MPI_Allreduce相当于在MPI_Reduce后马上再将结果进行一次广播，因此它等价于：

```
root = 0;
MPI_Reduce(sendbuf, recvbuf, count, datatype, op,
        root, comm);
MPI_Bcast(recvbuf, count, datatype, root, comm)
```



归约散发MPI_Reduce_scatter

```
int MPI_Reduce_scatter(void *sendbuf, void *recvbuf,
        int *recvcounts, MPI_Datatype datatype,
        MPI_Op, op, MPI_Comm comm)
```

归约散发函数首先进行一次COUNT $= \sum_{i=0}^{nprocs-1} recvcounts[i]$的归约操作， 然后再对归约结果进行散发操作，散发给第i个进程的数据块长度为recvcounts[i]. 其余参数的含义与MPI_Reduce一样。

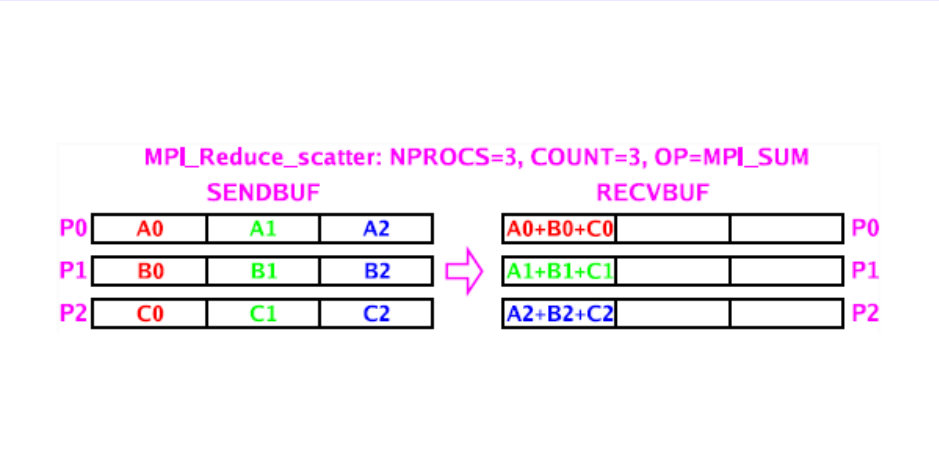
归约散发MPI_Reduce_scatter

设nprocs为进程数，myrank为进程号，则MPI_Reduce_scatter相当于：

```
int root,count,displs[nprocs],recvcounts[nprocs];
<type> *tmpbuf;
count = recvcount[0];
displs[0]=0;
for(i=0;i<nprocs;i++){
    count = count +recvcounts[i];
    displs[i]=displs[i]+recvcounts[i-1];
}
```

归约散发MPI_Reduce_scatter

```
root = 0;
MPI_Reduce(sendbuf, tmpbuf, count, datatype, op, root,
        comm);
MPI_Scatterv(tmpbuf,recvcounts, displs,datatype,
        recvbuf, recvcounts(myrank),datatype,root,comm)
```



前缀归约MPI_Scan

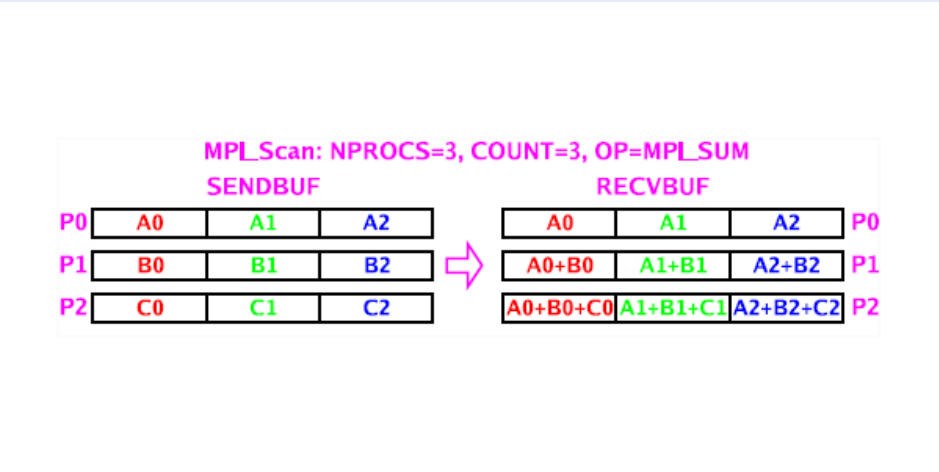
```
int MPI_Scan(void *sendbuf, void *recvbuf,int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

前缀归约，或前缀扫描，与归约操作类似，但各个处理器依次得到部分归约结果。确切地说，操作结束后第i个处理器的recvbuf中将包含前i个处理器的归约运算结果。各参数的含义与MPI_Allreduce基本相同。

前缀归约MPI_Scan

设进程号为myrank，则MPI_Scan相当于每个进程分别计算：

```
for(k=1;k<=count;k++){
    recvbuf(k)=sendbuf(k) of process 0;
    for(i=1;i<=myrank;i++)
        recvbuf(k)=recvbuf(k) op (sendbuf(k) of process i)
}
```



自定义归约运算

除MPI定义的运算外，用户也可以自己定义归约与前缀归约中运算。

```
int MPI_Op_create(MPI_User_function *func, int commute,
        MPI_Op *op)
```

MPI_Op_create创建（定义）一个新的运算。参数中func是用户提供的用于完成该运算的外部函数名， commute用来指明所定义的运算是否满足交换律（commute＝true表示满足）。C接口中op返回所创建的运算指针。一个运算创建后便和MPI预定义的运算一样，可以用在前面介绍的所有归约和前缀归约函数中。

自定义归约运算

负责完成二元运算的外部函数func应该具有如下形式的接口：

```
void func(void *invec, void *inoutvec, int *len,
          MPI_Datatype *datatype)
```

进入函数func时，invec和inoutvec包含参与运算的操作数(operand)。函数返回时inoutvec中应该包含运算的结果。len给出invec和inoutvec中包含的元素个数（相当于归约函数中的count）.datatype给出操作数的数据类型（即归约函数中的datatype）。直观地，函数func必须完成如下操作：

```
for(i=0;i<len;i++)
    inoutvec[i]=invec[i] op inoutvec[i];
```

当一个用户定义的运算不再需要时，可以调用MPI_Op_free将其释放，以便释放它所占用的系统资源。

```
int MPI_Op_free(MPI_Op *op)
```

Outline

正确使用聚合通信

一个正确的，可扩展的包含聚合通信的程序应该避免出现死锁。下面的语句是错误的：

```
switch(rank){
    case 0:
        MPI_Bcast(buf1,count,type,0, comm);
        MPI_Bcast(buf2,count,type,1, comm);
        break;
    case 1:
        MPI_Bcast(buf2,count, type, 1, comm);
        MPI_Bcast(buf1,count, type, 0, comm);
        break;
}
```

假设通信组包含0，1两个进程，如果操作同步进行，则会发生死锁。

正确使用聚合通信

下面的程序也是危险的:

```
switch(rank){
    case 0:
        MPI_Bcast(buf1,count, type, 0, comm0);
        MPI_Bcast(buf2,count, type, 2, comm2);
        break;
    case 1:
        MPI_Bcast(buf1,count, type, 1, comm1);
        MPI_Bcast(buf2,count, type, 0, comm0);
        break;
    case 2:
        MPI_Bcast(buf1,count, type, 2, comm2);
        MPI_Bcast(buf2,count, type, 1, comm1);
        break;
}
```

正确使用聚合通信(续)

假设comm0包含{0，1}，comm1包含{1，2}，comm2包含{2，0}。
如果广播是同步进行的，则出现循环依赖：comm0完成后comm2才能完成，comm2完成后comm1才能完成，comm1完成后comm0才能完成。
于是出现死锁。

正确使用聚合通信（续)

下面的语句是错误的:

```
switch(rank) {
      case 0:
          MPI_Bcast(buf1, count, type, 0, comm);
          MPI_Send(buf2, count, type, 1, tag, comm);
          break;
      case 1:
          MPI_Recv(buf2, count, type, 0, tag, comm, status);
          MPI_Bcast(buf1, count, type, 0, comm);
          break;
  }
```

0进程广播，然后阻塞发送给1进程；而1进程阻塞接收0进程发来所的数据，并执行0进程的广播。这也是一种循环依赖。

程序实例

π计算： MPICH中cpi.c的程序实例。该程序利用积分：

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

计算π值。令$f(x) = 4/(1+x^2)$。将区间$[0,1]$分成n等分，并令$x_i = (i-0.5)/n$,则：

$$\pi \approx \frac{1}{n} \sum_{i=1}^{n} f(x_i)$$

假设并行程序中共有P个进程参与计算，则第k个进程负责计算：

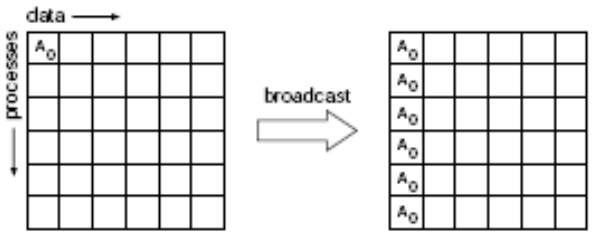
$$\sum_{1 \le i \le n, (i-1) mod P = k} f(x_i)$$

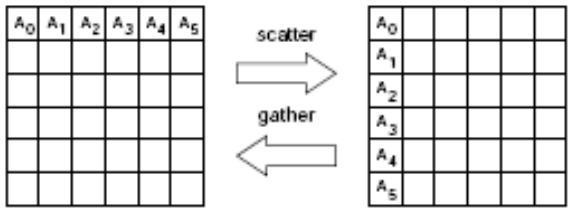
参看程序:cpi.c

聚合通信的总结

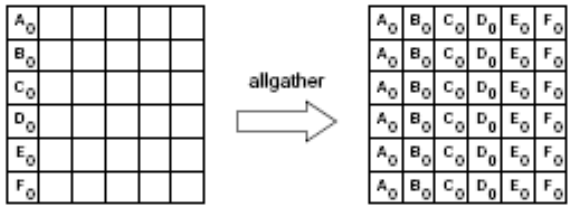
- 障碍同步：MPI_Barrier,一个组内所有进程都调用后才返回;
- 广播：MPI_Bcast;
- 数据(全)收集：MPI_Gather,MPI_Gatherv,MPI_Allgather,MPI_Allgatherv;
- 数据(全)发散：MPI_Scatter,MPI_Scatterv,MPI_Alltoall,MPI_Alltoallv;
- 数据(全)归约：MPI_Reduce,MPI_Allreduce;
- 数据归约发散：MPI_Reduce_scatter;
- 数据前缀归约：MPI_Scan.

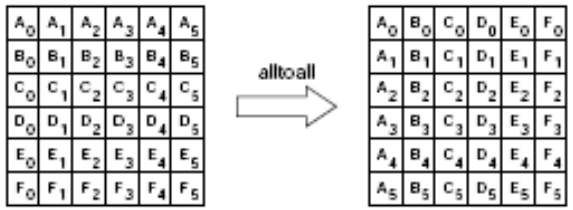
聚合通信的总结





聚合通信的总结





作业

- 编写程序，把一个长度为1000的数组的所有分量求和，要求对任意多个进程都对。
- 用阻塞型点对点通信函数实现MPI_Scan的功能，其中二元运算为例子程序ex5-10.c中的复数乘积。