Parallel Programming with MPI - Day 3

Science & Technology Support
High Performance Computing

Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212-1163
Table of Contents

• Collective Communication

• Problem Set
Collective Communication

- Collective Communication
- Barrier Synchronization
- Broadcast*
- Scatter*
- Gather
- Gather/Scatter Variations
- Summary Illustration
- Global Reduction Operations
- Predefined Reduction Operations
- MPI_Reduce
- Minloc and Maxloc*
- User-defined Reduction Operators
- Reduction Operator Functions
- Registering a User-defined Reduction Operator*
- Variants of MPI_Reduce

*includes sample C and Fortran programs
Collective Communication

- Communications involving a group of processes
- Called by *all* processes in a communicator
- Examples:
  - Broadcast, scatter, gather (Data Distribution)
  - Global sum, global maximum, etc. (Collective Operations)
  - Barrier synchronization
Characteristics of Collective Communication

• Collective communication will not interfere with point-to-point communication and vice-versa

• All processes must call the collective routine

• Synchronization not guaranteed (except for barrier)

• No non-blocking collective communication

• No tags

• Receive buffers must be exactly the right size
Barrier Synchronization

- **Red light for each processor:** turns **green** when all processors have arrived
- **Slower than hardware barriers** (example: Cray T3E)

**C:**

```c
int MPI_Barrier (MPI_Comm comm)
```

**Fortran:**

```fortran
INTEGER COMM,IERROR

CALL MPI_BARRIER (COMM,IERROR)
```
Broadcast

- One-to-all communication: same data sent from root process to all the others in the communicator

- C:

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

- Fortran:

```fortran
 type BUFFER (*)
 INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM IERROR)
```

- All processes must specify same root rank and communicator
# Sample Program #5 - C

```c
#include<mpi.h>
void main (int argc, char *argv[]) {
    int rank;
    double param;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank==5) param=23.0;
    MPI_Bcast(&param,1,MPI_DOUBLE,5,MPI_COMM_WORLD);
    printf("P:%d after broadcast parameter is %f\n",rank,param);
    MPI_Finalize();
}
```

```
P:0 after broadcast parameter is 23.000000
P:6 after broadcast parameter is 23.000000
P:5 after broadcast parameter is 23.000000
P:2 after broadcast parameter is 23.000000
P:3 after broadcast parameter is 23.000000
P:7 after broadcast parameter is 23.000000
P:1 after broadcast parameter is 23.000000
P:4 after broadcast parameter is 23.000000
```
Sample Program #5 - Fortran

```fortran
PROGRAM broadcast
INCLUDE 'mpif.h'
INTEGER err, rank, size
real param
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
if(rank.eq.5) param=23.0
call MPI_BCAST(param,1,MPI_REAL,5,MPI_COMM_WORLD,err)
print *,"P:",rank," after broadcast param is ",param
CALL MPI_FINALIZE(err)
END
```

P:1 after broadcast parameter is 23.
P:3 after broadcast parameter is 23.
P:4 after broadcast parameter is 23
P:0 after broadcast parameter is 23
P:5 after broadcast parameter is 23.
P:6 after broadcast parameter is 23.
P:7 after broadcast parameter is 23.
P:2 after broadcast parameter is 23.
Scatter

- One-to-all communication: different data sent to each process in the communicator (in rank order)

C:

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

Fortran:

```fortran
&lt;type&gt; SENDBUF(*), RECVBUF(*)

CALL MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                 RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
```

- `sendcount` is the number of elements sent to each process, not the “total” number sent
  - send arguments are significant only at the root process
Scatter Example
Sample Program #6 - C

```c
#include <mpi.h>
void main (int argc, char *argv[]) {
    int rank,size,i,j;
    double param[4],mine;
    int sndcnt,revcnt;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    revcnt=1;
    if(rank==3){
        for(i=0;i<4;i++) param[i]=23.0+i;
        sndcnt=1;
    }
    MPI_Scatter(param,sndcnt,MPI_DOUBLE,&mine,revcnt,MPI_DOUBLE,3,MPI_COMM_WORLD);
    printf("P:%d mine is %f\n",rank,mine);
    MPI_Finalize();
}
```

P:0 mine is 23.000000
P:1 mine is 24.000000
P:2 mine is 25.000000
P:3 mine is 26.000000
Sample Program #6 - Fortran

```fortran
PROGRAM scatter
INCLUDE 'mpif.h'
INTEGER err, rank, size
real param(4), mine
integer sndcnt,rcvcnt
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
rcvcnt=1
if(rank.eq.3) then
  do i=1,4
    param(i)=23.0+i
  end do
  sndcnt=1
end if
call MPI_SCATTER(param,sndcnt,MPI_REAL,mine,rcvcnt,MPI_REAL,
               &                           3,MPI_COMM_WORLD,err)
print *,"P:",rank," mine is ",mine
CALL MPI_FINALIZE(err)
END
```

P:1 mine is 25.
P:3 mine is 27.
P:0 mine is 24.
P:2 mine is 26.
Gather

- All-to-one communication: different data collected by root process
  - Collection done in rank order

- MPI_GATHER & MPI_Gather have same arguments as matching scatter routines

- Receive arguments only meaningful at the root process
Gather Example

<table>
<thead>
<tr>
<th>rank</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

1. Rank 0: A
2. Rank 1: A, B
3. Rank 2: A, B, C
4. Rank 3: A, B, C, D
Gather/Scatter Variations

- **MPI_Allgather**

- **MPI_Alltoall**

- **No root process specified:** all processes get gathered or scattered data

- **Send and receive arguments significant for all processes**
Summary

- **BCAST**: BROADCAST
- **SCATTER**: SCATTER
- **GATHER**: GATHER
- **ALLGATHER**: ALLGATHER

Rank 0
Rank 1
Rank 2
Global Reduction Operations

• Used to compute a result involving data distributed over a group of processes

• Examples:
  – Global sum or product
  – Global maximum or minimum
  – Global user-defined operation
Example of a Global Sum

- Sum of all the $x$ values is placed in result only on processor 0

C:

```c
MPI_Reduce(&x,&result,1, MPI_INTEGER, MPI_SUM, 0, MPI_COMM_WORLD)
```

Fortran:

```fortran
CALL MPI_REDUCE(x,result,1,MPI_INTEGER,MPI_SUM,0,
             MPI_COMM_WORLD,IERROR)
```
## Predefined Reduction Operations

<table>
<thead>
<tr>
<th>MPI Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise exclusive OR</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum and location</td>
</tr>
</tbody>
</table>
General Form

- **count** is the number of “ops” done on consecutive elements of **sendbuf** (it is also size of **recvbuf**)

- **op** is an associative operator that takes two operands of type **datatype** and returns a result of the same type

C:

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

Fortran:

```fortran
?type> SENDBUF(*), RECVBUF(*)
CALL MPI_REDUCE(SENDBUF,RECVBUF,COUNT,DATATYPE,OP,ROOT,COMM,IERROR)
```
MPI_Reduce

Rank

0

1

2

3

MPI_REDUCE

AoDoGoJ
Minloc and Maxloc

- Designed to compute a global minimum/maximum and an index associated with the extreme value
  - Common application: index is the processor rank (see sample program)

- If more than one extreme, get the first

- Designed to work on operands that consist of a value and index pair

- MPI Datatypes include:

  C:
  
  MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_2INT, MPI_SHORT_INT, MPI_LONG_DOUBLE_INT

  Fortran:
  
  MPI_2REAL, MPI_2DOUBLEPRECISION, MPI_2INTEGER
#include <mpi.h>
/* Run with 16 processes */
void main (int argc, char *argv[]) {
    int rank;
    struct {
        double value;
        int rank;
    } in, out;
    int root;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    in.value=rank+1;
in.rank=rank;
    root=7;
    MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MAXLOC,root,MPI_COMM_WORLD);
    if(rank==root) printf("PE:%d max=%lf at rank %d\n",rank,out.value,out.rank);
    MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MINLOC,root,MPI_COMM_WORLD);
    if(rank==root) printf("PE:%d min=%lf at rank %d\n",rank,out.value,out.rank);
    MPI_Finalize();
}

P:7 max=16.000000 at rank 15
P:7 min=1.000000 at rank 0
PROGRAM MaxMin
C
C Run with 8 processes
C
INCLUDE 'mpif.h'
INTEGER err, rank, size
classic in(2), out(2)
call MPI_INIT(err)
call MPI_COMM_RANK(MPI_WORLD_COMM, rank, err)
call MPI_COMM_SIZE(MPI_WORLD_COMM, size, err)
in(1) = rank + 1
in(2) = rank
call MPI_REDUCE(in, out, 1, MPI_2INTEGER, MPI_MAXLOC,
& 7, MPI_COMM_WORLD, err)
if (rank.eq.7) print *, "P: ", rank, " max = ", out(1), " at rank ", out(2)

call MPI_REDUCE(in, out, 1, MPI_2INTEGER, MPI_MINLOC,
& 2, MPI_COMM_WORLD, err)
if (rank.eq.2) print *, "P: ", rank, " min = ", out(1), " at rank ", out(2)
call MPI_FINALIZE(err)
END

P:2 min=1 at rank 0
P:7 max=8 at rank 7
User-Defined Reduction Operators

• Reducing using an arbitrary operator \( \chi \)

C -- function of type `MPI_User_function`:

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

Fortran -- function of type:

```fortran
$type$ INVEC(LEN), INOUTVEC(LEN)
INTEGER LEN, DATATYPE

FUNCTION MY_OPERATOR (INVEC(*), INOUTVEC(*), LEN, DATATYPE)
```
Reduction Operator Functions

- Operator function for $\chi$ must have syntax:

  \[
  \text{for (i=1 to len)} \\
  \quad \text{inoutvec}(i) = \text{inoutvec}(i) \; \chi \; \text{invec}(i)
  \]

- Operator $\chi$ need not commute

- `inoutvec` argument acts as both a second input operand as well as the output of the function
Registering a User-Defined Reduction Operator

• Operator handles have type `MPI_Op` or `INTEGER`

• If `commute` is `TRUE`, reduction may be performed faster

C:

```c
int MPI_Op_create (MPI_User_function *function,
                    int commute, MPI_Op *op)
```

Fortran:

```fortran
EXTERNAL FUNC
INTEGER OP,IERROR
LOGICAL COMMUTE

MPI_OP_CREATE (FUNC, COMMUTE, OP, IERROR)
```
Sample Program #8 - C

```c
#include <mpi.h>
typedef struct {
    double real, imag;
} complex;

void cprod(complex *in, complex *inout, int *len, MPI_Datatype *dptr) {
    int i;
    complex c;
    for (i=0; i<*len; ++i) {
        *inout=c;
        in++;
        inout++;    
    }
}

void main (int argc, char *argv[]) {
    int rank;
    int root;
    complex source, result;
```
Sample Program #8 - C (cont.)

```c
MPI_Op myop;
MPI_Datatype ctype;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

MPI_Type_contiguous(2,MPI_DOUBLE,&ctype);
MPI_Type_commit(&ctype);
MPI_Op_create(cprod,TRUE,&myop);
root=2;
source.real=rank+1;
source.imag=rank+2;
MPI_Reduce(&source,&result,1,ctype,myop,root,MPI_COMM_WORLD);
if(rank==root) printf("PE:%d result is %lf + %lfi\n",rank,
result.real, result.imag);
MPI_Finalize();
}
```

P:2 result is -185.000000 + -180.000000i
Sample Program #8 - Fortran

```fortran
PROGRAM UserOP
  INCLUDE 'mpif.h'
  INTEGER err, rank, size
  integer source, reslt
  external digit
  logical commute
  integer myop
  CALL MPI_INIT(err)
  CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
  CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
  commute=.true.
  call MPI_OP_CREATE(digit,commute,myop,err)
  source=(rank+1)**2
  call MPI_BARRIER(MPI_COM_WORLD,err)
  call MPI_SCAN(source,reslt,1,MPI_INTEGER,myop,MPI_COMM_WORLD,err)
  print *,"P:",rank," my result is ",reslt
  CALL MPI_FINALIZE(err)
END

integer function digit(in,inout,len,type)
  integer in(len),inout(len)
  integer len,type
  do i=1,len
    inout(i)=mod((in(i)+inout(i)),10)
  end do
  digit = 5
end
```

P:6 my result is 0
P:5 my result is 1
P:7 my result is 4
P:1 my result is 5
P:3 my result is 0
P:2 my result is 4
P:4 my result is 5
P:0 my result is 1
Variants of MPI_REDUCE

- **MPI_ALLREDUCE** -- no root process (all get results)
- **MPI_REDUCE_SCATTER** -- multiple results are scattered
- **MPI_SCAN** -- “parallel prefix”
MPI_ALLREDUCE

Rank

0

A B C

1

D E F

2

G H I

3

J K L

AoDoGoJ

MPI_ALLREDUCE

A B C
MPI_REDUCE_SCATTER

<table>
<thead>
<tr>
<th>Rank</th>
<th>recvcounts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

MPI_REDUCE_SCATTER

AoDoGoJ
MPI_SCAN

Rank

0

MPI_SCAN

A

AoD

AoDoG

AoDoGoJ

A

B

C

D

E

F

G

H

I

J

K

L
Problem Set

1) Write a program in which four processors search an array in parallel (each gets a fourth of the elements to search). All the processors are searching the integer array for the element whose value is 11. There is only one 11 in the entire array of 400 integers.

By using the non-blocking MPI commands you have learned, have each processor continue searching until one of them has found the 11. Then they all should stop and print out the index they stopped their own search at.

You have been given a file called data which contains the integer array (ASCII, one element per line). Before the searching begins have ONLY P0 read in the array elements from the data file and distribute one fourth to each of the other processors and keep one fourth for its own search.

2) Rewrite your solution program to Problem 1 so that the MPI broadcast command is used.

3) Rewrite your solution program to Problem 1 so that the MPI scatter command is use.
4) In this problem each of eight processors used will contain an integer value in its memory that will be the operand in a collective reduction operation. The operand value for each processor is -27, -4, 31, 16, 20, 13, 49, and 1 respectively.

Write a program in which the maximum value of the integer operands is determined. The result should be stored on P5. P5 should then transfer the maximum value to all the other processors. All eight processors will then normalize their operands by dividing by the maximum value. (EXTRA CREDIT: Consider using MPI_ALL_REDUCE)

Finally, the program should calculate the sum of all the normalized values and put the result on P2. P2 should then output the normalized global sum.