MPI by Example

Excerpts from:
Ohio Supercomputer Center &
Argonne National Lab
Peer-to-Peer

• Write a program that takes data from process \( i \) and sends it to the next process according to rank.

• The last process will send its data to the first.

• Assume that the data consists of a single integer.

• Each process reads the data from the user.
Peer-To-Peer Implementation

#include <stdio.h>
#include "mpi.h"
int main( int argc, char** argv )
{
    int rank, value, size;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    scanf( "%d", &value );
    MPI_Send( &value, 1, MPI_INT, (rank + 1) % size, 0, MPI_COMM_WORLD );
    MPI_Recv( &value, 1, MPI_INT, (rank – 1 + size) % size, 0, MPI_COMM_WORLD, &status );
    printf( "Process %d got %d\n", rank, value );
    MPI_Finalize( );
    return 0;
}
User-Defined Operators (1)

C -- function of type MPI_User_function:

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

- Operator function for * must have syntax:
  ```c
  for (i=1 to len)
    inoutvec(i) = inoutvec(i) * invec(i)
  ```
- Operator * does not need to commute
- `inoutvec` argument acts as both a second input operand as well as the output of the function
User-Defined Operators (2)

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

- Operator handles have type `MPI_Op`
- If `commute` is `TRUE`, reduction may be performed faster
#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;
Example – Complex Product (2)

```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();
```
Calculating PI

• This exercise presents a simple program to determine the value of pi.
• The algorithm suggested here is chosen for its simplicity. The method evaluates the integral of 4/(1+x*x) between 0 and 1.
• The method is simple: the integral is approximated by a sum of n intervals; the approximation to the integral in each interval is (1/n)*4/(1+x*x).
• The master process (rank 0) asks the user for the number of intervals; the master should then broadcast this number to all of the other processes.
• Each process then adds up every n'th interval (x = rank/n, rank/n+size/n,...).
• Finally, the sums computed by each process are added together using a reduction.
Calculating PI

```c
int main(int argc, char* argv) {
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643; double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits ) ");
        scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    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;
}
```
2D-Array

• Process $i$ sends $num$ ints from the $i_{th}$ column of a 100x150 int array.

• The complicating factor is that the various values of $num$ are not known to root, so a separate gather must first be run to find these out.

• The data is placed contiguously at the receiving end.
MPI_Comm comm = MPI_COMM_WORLD;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, stride, myrank, disp[2], blocklen[2];
MPI_Datatype stype,types[2];
int *displs,i,*rcounts, num = DATA;
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );

/* First, gather nums to root */
rcounts = (int *)malloc(gsize*sizeof(int));
MPI_Gather( &num, 1, MPI_INT, rcounts, 1, MPI_INT, root, comm);

if (myrank==0) {

    /* root now has correct rcounts, using these we set displs[] so
    that data is placed contiguously (or concatenated) at receive end */
displs = (int *)malloc(gsize*sizeof(int));
displs[0] = 0;
for (i=1; i<gsize; ++i)
displs[i] = displs[i-1]+rcounts[i-1];

Why not scan?
Why do we calculate on every processor?
And, create receive buffer

```
rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1]) *sizeof(int));
```

Create datatype for one int, with extent of entire row

```
disp[0] = 0;
disp[1] = 150*sizeof(int);
type[0] = MPI_INT;
type[1] = MPI_UB;
blocklen[0] = 1;
blocklen[1] = 1;
MPI_Type_struct( 2, blocklen, disp, type, &stype );
MPI_Type_commit( &stype );
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, num, stype, rbuf, rcounts, displs, MPI_INT, root, comm);
```