Introduction to MPI

- MPI is a standard for message passing interfaces
- MPI-1 covers point-to-point and collective communication
- MPI-2 covers connection based communication and I/O
- Typical implementations include MPICH (Used by Scyld), and LAMMPI
MPI Basics

- Most used MPI commands
  
  MPI_Init - start using MPI
  MPI_Comm_size - get the number of tasks
  MPI_Comm_rank - the unique index of this task
  MPI_Send - send a message
  MPI_Recv - receive a message
  MPI_Finalize - stop using MPI
Initialize and Finalize

- The first MPI call must be to `MPI_Init`.
- The last MPI call must be to `MPI_Finalize`.

```c
#include <mpi.h>

int main(int argc, int **argv)
{
    MPI_Init(&argc, &argv);
    // put program here
    MPI_Finalize();
}
```
Initialize and Finalize

int MPI_Init(int *argc, char ***argv);

int MPI_Finalize();
Size and Rank

- MPI_Comm_size returns the number of tasks in the job
  
  ```c
  int size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  ```

- MPI_Comm_rank returns the number of the current task (0 .. size-1)
  
  ```c
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ```
A Simple Example

#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {

    int rank, size;
    MPI_Init( &argc,&argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);
    MPI_Comm_size( MPI_COMM_WORLD, &size);
    printf("Hello world from process %d of %d \n", rank,size);
    MPI_Finalize();
    return 0;
}
Mpirun

• Runs one or more tasks on nodes:

  mpirun [options] prog-name arg1 arg2 ...
  -np <int>       run <int> tasks
  --all_cpus      run a task on each cpu
  --all_local     run all tasks on master node
  --no_local      run no tasks on master node
  --exclude <int>[::<int>]          run on any nodes not listed
  --map <int>[::<int>]             run nodes listed
Send and Recv

- **MPI_Send** to send a message
  
  ```c
  char sbuf[COUNT];
  MPI_Send(sbuf, COUNT, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  ```

- **MPI_Recv** to receive a message
  
  ```c
  char rbuf[COUNT];
  MPI_Status status;
  MPI_Recv(rbuf, COUNT, MPI_CHAR, 1, 99, MPI_COMM_WORLD, &status);
  ```
Anatomy of MPI_Recv

- MPI_Recv(rbuf, COUNT, MPI_CHAR, 1, 99, MPI_COMM_WORLD, &status);
  rbuf : pointer to receive buffer
  COUNT : items in receive buffer
  MPI_CHAR : MPI datatype
  1 : source task number (rank)
  99 : message tag
  MPI_COMM_WORLD : communicator
  Status : pointer to status struct
MPI Datatypes

- Encodes type of data sent and received
- Built-in types
  
  ```
  MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG
  MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE
  MPI_BYTE, MPI_PACKED
  ```

- User defined types (covered later)
  
  ```
  MPI_Type_contiguous, MPI_Type_vector, 
  MPI_Type_indexed, MPI_Type_struct 
  MPI_Pack, MPI_Unpack
  ```
MPI Communicators

- Abstract structure represents a group of MPI tasks that can communicate
  - `MPI_COMM_WORLD` represents all of the tasks in a given job
- Programmer can create new communicators to subset `MPI_COMM_WORLD`
  - `RANK` or task number is relative to a given communicator
- Messages from different communicators do not interfere
Each task in a job has a unique rank or task number.

Numbers run from 0 to size-1, where size is the number of tasks.

- `MPI_Comm_size(MPI_COMM_WORLD, &size)`
- `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`

Send and `Recv` specify destination or source task by rank.

`Recv` can specify source of `MPI_ANY_SOURCE` to receive from any task.
Message Tags

- All messages are sent with an integer message tag
- `MPI_Recv` will only receive a message with the tag specified
- `MPI_ANY_TAG` can be used to receive messages with any tag
Matching Send andRecv

- When \texttt{MPI\_Send} called, send is "posted"
- When \texttt{MPI\_Recv} called, receive is "posted"
- Posted \texttt{MPI\_Recv} matches posted \texttt{MPI\_Send} if
  - \textit{Destination of} \texttt{MPI\_Send} matches receiving task
  - \textit{Source of} \texttt{MPI\_Recv} matches sending task or source is \texttt{MPI\_SOURCE\_ANY}
  - \textit{Tag of} \texttt{MPI\_Send} matches tag of \texttt{MPI\_Recv} or tag of \texttt{MPI\_Recv} is \texttt{MPI\_TAG\_ANY}
  - \textit{Communicator or} \texttt{MPI\_Send} matches communicator of \texttt{MPI\_Recv}
Receive Buffer Size

• Receive buffer must be big enough for the message being received
• If message is smaller, only part of buffer filled in
• If message is too big, overflow error
• MPI_Probe allows programmer to check the next message before receiving it

  // src, tag, comm, stat
  MPI_Probe(1, 99, MPI_COMM_WORLD, &status);
MPI Status Struct

- Allows user to query the return status of MPI call
  
  ```c
  status(MPI_SOURCE)
  status(MPI_TAG)
  status(MPI_ERROR)
  ```

- Allows user to query number of items received
  
  ```c
  int count;

  MPI_Get_count(&status, MPI_CHAR, &count)
  ```
Send and Receive Example

#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {

    int numprocs, myrank, namelen, i;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    char greeting[MPI_MAX_PROCESSOR_NAME + 80];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank);
    MPI_Comm_size( MPI_COMM_WORLD, &numprocs);
    MPI_Get_Processor_name( processor_name, &namelen);

    sprintf(greeting,"Hello world from process %d of %d on %s \n",
            myrank,numprocs, processor_name);
Send and Receive Example

if (myrank == 0) {
    printf("%s\n", greeting);
    for (i=1;i<numprocs;i++) {
        MPI_Recv(greeting,sizeof(greeting), MPI_CHAR, i, 1, MPI_COMM_WORLD);
        printf("%s\n", greeting);
    }
}
else {
    MPI_Send(greeting, strlen(greeting) +1, MPI_CHAR, 0,1,MPI_COMM_WORLD);
}

MPI_Finalize();
return( 0);
}
Communication Modes

- Convered on next pages ...
  - Standard Mode
  - Buffered Mode
    
    MPI_Bsend(buf, count, datatype, dest, tag, MPI_COMM_WORLD);
  
  - Synchronous Mode
    
    MPI_Ssend(buf, count, datatype, dest, tag, MPI_COMM_WORLD);
  
  - Ready Mode
    
    MPI_Rsend(buf, count, datatype, dest, tag, MPI_COMM_WORLD);
Standard Mode

- Message is buffered if system space is available
- Otherwise call will block until message delivered or system space becomes available
- Completion of send does not imply message delivered
Buffered Mode

- Program provides buffer space for messages
  
  ```
  MPI_Buffer_attach(buffer, size);
  MPI_Buffer_detach(buffer_addr, size);
  ```

- Send returns as soon as message is copied into buffer

- Completion of send does not imply message delivered
Synchronous Mode

• Send blocks until matching receive is posted and data either buffered by system or delivered

• Completion of send does not imply completion of receive, but DOES imply start of receive
Ready Mode

• Send can only be correctly called if the matching receive has already been posted
• Send completes when data is either received or copied to system buffer
• Used primarily on systems where data flow can be optimized if receiver is ready for message
Semantics

• Covered on next pages ...
  – Messages are non-overtaking
  – Progress is guaranteed
  – Fairness is not guaranteed
  – System resources may be limited
Non-Overtaking Messages

- If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending.

- If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending.

Quoted from: MPI: A Message-Passing Interface Standard

(c) 1993, 1994, 1995 University of Tennessee, Knoxville, Tennessee. Permission to copy without fee all or part of this material is granted, provided the University of Tennessee copyright notice and the title of this document appear, and notice is given that copying is by permission of the University of Tennessee.
Fairness

- MPI does not guarantee fairness in matching messages
- Messages from different sources may overtake one another
- Programmer's responsibility to prevent starvation
Limited System Resources

- MPI does not guarantee system resources exist for buffering messages.
- Programs that assume system resources available can deadlock if resources become busy.
- Properly coded programs usually exist that will complete regardless of available buffer space.
- Buffered mode allows programmer to provide adequate buffer space.
Combined Send/Recv

- Single call both sends and receives a message
- Can send and receive to same task, or different tasks
- Guarantees that buffering and blocking semantics will not result in deadlock

    MPI_Sendrecv();
    MPI_Sendrecv_replace();
Non-blocking Recv Details

• Added argument for MPI_Request structure
  
  ```c
  MPI_Request request;
  MPI_Irecv(buffer, count, datatype, dest, tag, MPI_COMM_WORLD, &request);
  ```

• Test for completion
  
  ```c
  MPI_Status status;
  int flag;
  MPI_Test(&request, &flag, &status);
  ```

• Wait for completion
  
  ```c
  MPI_Wait(&request, &status);
  ```
Collective Communication

- A communication pattern that involves all processes in a communicator.

- MPI contains a rich set of collective communication operations:
  - Broadcast, Reduce, Allreduce
  - Scatter, Gather, Allgather, Alltoall
  - Scan, Barrier, Scatter_reduce
Broadcast

- Broadcast to all tasks of a communicator
  ```c
  int root; // rank of the source
  MPI_Bcast(buffer, count, datatype,
             root, MPI_COMM_WORLD);
  ```
- Must be called by all tasks with the same arguments
Barrier

• Barrier synchronizes all tasks of a communicator
  
  \[
  \text{MPI\_Barrier(MPI\_COMM\_WORLD);}
  \]

• Each task calling \text{MPI\_Barrier} will stop until all tasks in the communicator have called \text{MPI\_Barrier}
Barrier

- Barrier synchronizes all tasks of a communicator
  - MPI_BARRIER(MPI_Comm comm);
  - MPI::Comm::Barrier()
  - MPI_BARRIER(COMM, IERROR)
  - INTEGER COMM, IERROR

- Each task calling MPI_BARRIER will stop until all tasks in the communicator have called MPI_BARRIER
Reduction

- Perform collective operations on a group of tasks

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

<table>
<thead>
<tr>
<th>T0</th>
<th>A0</th>
<th>B0</th>
<th>C0</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>T2</td>
<td>A2</td>
<td>B2</td>
<td>C2</td>
<td>D2</td>
</tr>
<tr>
<td>T3</td>
<td>A3</td>
<td>B3</td>
<td>C3</td>
<td>D3</td>
</tr>
</tbody>
</table>

Reduce

<table>
<thead>
<tr>
<th>ΣA</th>
<th>ΣB</th>
<th>ΣC</th>
<th>ΣD</th>
</tr>
</thead>
</table>
Reduction Operators

- The MPI_Op argument can be user-defined or one of a number of predefined operation.
  - The operation is performed on each node on the send buffer and stored in the destination node in the receive buffer
  - Predefined Operations include:
    - Sum, Product
    - Max, Min
    - Logical and bitwise AND, OR, XOR
- User defined functions are done with MPI_OP_CREATE and MPI_OP_FREE
- Many other collective ops exist in MPI and MPI-2; wait for the advanced course!
Example

• Find PI by numerically integrating
  \[-f(x) = \frac{4}{1 + x^2}\]
  – over range 0-1
• Implement with broadcast and reduce
• Number of intervals is broadcast
• Processors each pick a range of intervals to compute in parallel
• Reduction is used to produce a global sum
• Implemented in C, Fortran, and C++
Example

```c
#include "mpi.h"
#include <stdio.h>
#include <math.h>
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);
    while (1) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```
Example

```c
if (n == 0)
    break;
else {
    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;
```
An Example - Integration with the Trapezoidal Rule

```c
main(int argc, char **argv) {
    MPI_Init(&argc, &argv);

    /* Get my process rank and size*/
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process' interval of
     * integration = local_n*h. So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

    /* Add up the integrals calculated by each process */
    if (my_rank == 0) {
        total = integral;
        for (source = 1; source < p; source++) {
            MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                      MPI_COMM_WORLD, &status);
            total = total + integral;
        }
    } else {
        MPI_Send(&integral, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
    }

    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n", n);
        printf("of the integral from %f to %f = %f\n", a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();
} /* main */
```
An Example - Integration with the Trapezoidal Rule

float Trap(
    float local_a /* in */,
    float local_b /* in */,
    int local_n /* in */,
    float h /* in */) {

    float integral; /* Store result in integral */
    float x;
    int i;

    float f(float x); /* function we're integrating */

    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;
    return integral;
} /* Trap */

float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    return_val = x*x;
    return return_val;
} /* f */
void Get_data(float* a_ptr, float* b_ptr, 
        int* n_ptr, int my_rank, int p) {

    int source = 0;   /* All local variables used by */
    int dest;        /* MPI_Send and MPI_Recv */
    int tag;
    MPI_Status status;

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%f%f%d", a_ptr, b_ptr, n_ptr);
        for (dest = 1; dest < p; dest++) {
            tag = 0;
            MPI_Send(a_ptr, 1, MPI_FLOAT, dest, tag, 
                    MPI_COMM_WORLD);
            tag = 1;
            MPI_Send(b_ptr, 1, MPI_FLOAT, dest, tag, 
                    MPI_COMM_WORLD);
            tag = 2;
            MPI_Send(n_ptr, 1, MPI_INT, dest, tag, 
                    MPI_COMM_WORLD);
        }
    } else {
        tag = 0;
        MPI_Recv(a_ptr, 1, MPI_FLOAT, source, tag, 
                  MPI_COMM_WORLD, &status);
        tag = 1;
        MPI_Recv(b_ptr, 1, MPI_FLOAT, source, tag, 
                  MPI_COMM_WORLD, &status);
        tag = 2;
        MPI_Recv(n_ptr, 1, MPI_INT, source, tag, 
                  MPI_COMM_WORLD, &status);
    }
} /* Get_data */
Communication Pattern in Trapezoid Program

For initial send, assuming 8 nodes

Last worker starts after process 0 sends 7 messages!
Tree structured Communication

Last worker starts after process 0 sends 3 messages!
Modified Trapezoid

- Adapt to add tree-based computation using send and receive.

```c
void Get_data1(...

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
    }

    for (stage = 0; stage < Ceiling_log2(p); stage++)
        if (I_receive(stage, my_rank, &source))
            Receive(a_ptr, b_ptr, n_ptr, source);
        else if (I_send(stage, my_rank, p, &dest))
            Send(*a_ptr, *b_ptr, *n_ptr, dest);
} /* Get_data1*/
```

- Creates a log(p) stage tree to exchange data
Modified Trapezoid

- Adapt to add tree-based computation using send and receive.

    int I_receive(int stage, int my_rank,
                  int* source_ptr /* out */) {
        int power_2_stage;

        /* 2^stage = 1 << stage */
        power_2_stage = 1 << stage;
        if ((power_2_stage <= my_rank) &&
            (my_rank < 2*power_2_stage)) {
            *source_ptr = my_rank - power_2_stage;
            return 1;
        } else return 0;
    } /* I_receive */

    int I_send(int stage, int my_rank,
               int p, int* dest_ptr ) {
        int power_2_stage;

        /* 2^stage = 1 << stage */
        power_2_stage = 1 << stage;
        if (my_rank < power_2_stage) {
            *dest_ptr = my_rank + power_2_stage;
            if (*dest_ptr >= p) return 0;
            else return 1;
        } else return 0;
    } /* I_send */

voices

void Send(float a, float b, int n, int dest) {

    MPI_Send(&a, 1, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
    MPI_Send(&b, 1, MPI_FLOAT, dest, 1, MPI_COMM_WORLD);
    MPI_Send(&n, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
} /* Send */
Tree Communication

- OK, that **should** go faster
- Wouldn't it be nice if MPI created the tree for us?
- But on, certain systems, it may not be faster....
Tree Communication

Nodes 0-3

Nodes 4-7

Slow link
Tree Communication

This tree minimizes the number of messages across the slow link!
Collective Operations

- MPI collective operations protect programmer from details of cluster topology
- MPI definition does not specify how broadcast is implemented
- MPICH implements broadcast as a spanning tree as shown
- New versions embed topology aware operations to deal with "slow link" problem
  - Programmer calls broadcast, MPI Implementation deals with details
- Our trapezoid integration program can be modified to use broadcast as follows:
Trapezoid with Broadcast

```c
void Get_data2(float* a_ptr, float* b_ptr, int* n_ptr, int my_rank) {
    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
    }
    MPI_Bcast(a_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);
} /* Get_data2 */
```

- Cleaner, simpler program
- More portable between clusters
- Note, process 0 has broadcast the same as all other ranks...
Reduction Revisited

• Perform collective operations on a group of tasks

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

    sendbuf
    recvbuf
    count
    datatype
    op
    root
    comm
Reduction Operators

- MPI_Op argument can be user-defined or predefined operation
  - Values in send buffer on each node combined using operator
  - Result stored on root in receive buffer
- Predefined Operations:

  - MPI_MAX Maximum
  - MPI_MIN Minimum
  - MPI_SUM Sum
  - MPI_PROD Product
  - MPI_LAND Logical And
  - MPI_LOR Logical Or
  - MPI_BOR Bitwise Or
  - MPI_LXOR Logical Exclusive OR
  - MPI_BXOR Logical Bitwise OR
  - MPI_MAXLOC Maximum and it's location
  - MPI_MINLOC Minimum and it's location
Dot-Products with Reduction

float Serial_dot(float x[], float y[], int n) {
    int i;
    float sum = 0.0;
    for (i = 0; i < n; i++)
        sum = sum + x[i]*y[i];
    return sum;
} /* Serial_dot */

float Parallel_dot(float local_x[], float local_y[], int n_bar) {
    float local_dot;
    float dot = 0.0;
    float Serial_dot(float x[], float y[], int m);

    local_dot = Serial_dot(local_x, local_y, n_bar);
    MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,
                MPI_SUM, 0, MPI_COMM_WORLD);
    return dot;
} /* Parallel_dot */

Note: you may not make the first and second arguments to reduce the same!
Creating Reduction Operators

• Reduce supports user defined “plug in” functions for reduce (and scan):

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

• Function – user supplied routine meeting specification
• Commute – is the operation commutative (T/F)?
• Op – handle (pointer) to give to reduce
Creating Reduction Operators

- User-supplied functions must have 4 arguments as described below:

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

- Datatype is the type to perform the operation on
- len is the length of both invec and inoutvec
- The operation is performed on each element of invec with the corresponding element of inoutvec
- Results are returned in inoutvec
Scatter and Gather

- **Scatter** takes data stored on a single node and sends a piece of it to every node in a communicator.
- **Gather** merges a data set scattered across multiple tasks and collects it on a single task.
- Extremely useful in any kind of matrix operations.
Scatter

- Distinct data sent from root to each task
  
  ```c
  int MPI_Scatter(void* sndbuf, int sndcount, MPI_Datatype sndtype,
                  void* rcvbuf, int rcvcount, MPI_Datatype rcvtype,
                  int root, MPI_Comm comm);
  ```
Gather

- Data is sent from all tasks to the root
- Each task's data is stored in rank order

```c
MPI_Gather(void* sndbuf, int sndcount, MPI_Datatype sndtype,
           void* rcvbuf, int rcvcount, MPI_Datatype rcvtype, int root,
           MPI_Comm comm);
```
Example - Dot product

- Take the dot product of all rows of matrix \( A \) with vector \( x \)
- Assume \( A \) and \( x \) are stored in a block-row distribution on tasks.
- Need to **gather** all elements of \( x \) onto a single task

```c
float local_x[]; /* Storage for local part of vector */
float global_x[]; /* Storage for global vector */
/* Assumes n is divisible by p */
MPI_Gather(local_x, n/p, MPI_FLOAT, global_x, n/p,
    MPI_FLOAT, 0, MPI_COMM_WORLD);
```

- \( X \) stored on task with rank 0 in communicator `comm_world`
Example, continued...

- Scatter vector read from user across tasks to begin with:

```c
void Read_vector(
    char* prompt    /* in */,
    float local_x[] /* out */,
    int local_n    /* in */,  
    int my_rank    /* in */,
    int p          /* in */) {

    int i;
    float temp[MAX_ORDER];
    if (my_rank == 0) {
        printf("%s\n", prompt);
        for (i = 0; i < p*local_n; i++)
            scanf("%f", &temp[i]);
    }
    MPI_Scatter(temp, local_n, MPI_FLOAT, local_x, local_n,
                MPI_FLOAT, 0, MPI_COMM_WORLD);
} /* Read_vector */
```
Example, continued...

- Gather matrix to print results:

```c
void Print_matrix(
    char*            title       /* in */ ,
    LOCAL_MATRIX_T   local_A     /* in */ ,
    int              local_m     /* in */ ,
    int              n           /* in */ ,
    int              my_rank     /* in */ ,
    int              p           /* in */) { 
    int    i ,  j ;
    float temp[MAX_ORDER][MAX_ORDER];
    MPI_Gather(local_A, local_m*MAX_ORDER, MPI_FLOAT, temp, 
               local_m*MAX_ORDER, MPI_FLOAT, 0, MPI_COMM_WORLD);
    if  (my_rank == 0) {
        printf("%s\n", title);
        for  (i = 0; i < p*local_m; i++) {
            for  (j = 0; j < n; j++)
                printf("%4.1f ", temp[i][j]);
            printf("\n");
        }
    }
} /* Print_matrix */
```
Gather to All

- Gather, but all tasks get result

  ```c
  MPI_Allgather(void* sndbuf, 
                  int sndcount, MPI_Datatype sndtype, 
                  void* rcvbuf, 
                  int rcvcount, MPI_Datatype rcvtype, 
                  int root, MPI_Comp comm);
  ```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>A0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>A2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>A3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
</tr>
<tr>
<td></td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
</tr>
<tr>
<td></td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
</tr>
<tr>
<td></td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
</tr>
</tbody>
</table>

Allgather
All-to-All

- Scatter combined with gather-to-all

\[
\text{MPI\_Alltoall}(\text{void* } \text{sndbuf, int sndcount, MPI\_Datatype sndtype, void* rcvbuf, int rcvcount, MPI\_Datatype rcvtype, MPI\_Comm comm});
\]

<table>
<thead>
<tr>
<th>T0</th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>B0</td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
</tr>
<tr>
<td>T2</td>
<td>C0</td>
<td>C1</td>
<td>C2</td>
<td>C3</td>
</tr>
<tr>
<td>T3</td>
<td>D0</td>
<td>D1</td>
<td>D2</td>
<td>D3</td>
</tr>
</tbody>
</table>

Alltoall

<table>
<thead>
<tr>
<th>A0</th>
<th>B0</th>
<th>C0</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
<td>D2</td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td>C3</td>
<td>D3</td>
</tr>
</tbody>
</table>
GatherV

• Like gather, but with a stride:
  
  int MPI_Gatherv(void *sendbuf, int sendcount,
                  MPI_Datatype sendtype, void *recvbuf, int
                  *recvcounts, int *displs, recvtype, root, comm);

• Allows a varying amount of data to be gathered from each process
  – recvcounts now an array

• Allows spacing between the blocks:
  – the “displs” argument lists displacements between start of received blocks
All Reduce

- Reduce, but all tasks receive result

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

• Same syntax as reduce; however, result is left on all tasks not just root task
• Equivalent result to calling reduce then broadcast
• On a switched network, should be same performance as reduce
Reduce Scatter

- Reduce, results scattered to tasks
  
  ```c
  MPI_Reduce_scatter(void* sndbuf, 
                      void* rcvbuf, int *rcvcounts, 
                      MPI_Datatype datatype, 
                      MPI_Op op, MPI_Comm comm);
  ```

```
   T0  A0  B0  C0  D0
   T1  A1  B1  C1  D1
   T2  A2  B2  C2  D2
   T3  A3  B3  C3  D3

   ΣA
   ΣB
   ΣC
   ΣD
```

Reduce_Scatter
Scan

• Performs prefix reduction
  \[
  \text{MPI\_Scan} (\text{void*} \ \text{sndbuf},
  \text{void*} \ \text{rcvbuf}, \text{int} \ \text{count},
  \text{MPI\_Datatype} \ \text{datatype},
  \text{MPI\_Op} \ \text{op}, \text{MPI\_Comm} \ \text{comm})
  \]

• User defined functions are the same as with reduce

• \text{MPI\_IN\_PLACE} option as second argument leaves result in the input buffer

• Inclusive scan, exclusive scan is available with \text{MPI\_Exscan()}
User Defined Datatypes

- **Methods for creating data types**
  
  ```
  MPI_Type_contiguous();
  MPI_Type_vector();
  MPI_Type_indexed();
  MPI_Type_struct();
  MPI_Pack();
  MPI_Unpack();
  ```

- **MPI datatypes defined similar to modern programming languages (C,C++,F90)**

- **Allows communication and I/O operations to use the same datatypes as rest of program**

- **Makes expressing the partitioning of datasets easier**
Some datatype terminology

• Every MPI datatype has a few characteristics
  – type signature
    • list of the basic datatypes (in order) that make up the derived type
  – type map
    • basic datatypes
    • lower bound of each type
    • extent of the type (size + buffering)

• Some of this information is available about MPI datatypes through:
  MPI_Get_extent
  MPI_Get_size
Contiguous Array

- Creates an array of counts elements:

  ```c
  MPI_Type_contiguous(int count,
                      MPI_Datatype oldtype,
                      MPI_Datatype *newtype)
  ```
Strided Vector

- Constructs a cyclic set of elements

\[
\text{MPI\_Type\_vector}(\text{int count, int blocklength, int stride, MPI\_Datatype oldtype, MPI\_Datatype *newtype});
\]

- Stride specified in number of elements
- Stride can be specified in bytes
  \[
  \text{MPI\_Type\_hvector}();
  \]
- Stride counts from start of block
Indexed Vector

- Allows an irregular pattern of elements
  
  \[
  \text{MPI	extunderscore Type	extunderscore indexed}(\text{int\ count,} \\
  \quad \text{int \ast array	extunderscore of	extunderscore blocklengths,} \\
  \quad \text{int \ast array	extunderscore of	extunderscore displacements,} \\
  \quad \text{MPI	extunderscore Datatype\ oldtype,} \\
  \quad \text{MPI	extunderscore Datatype \ast newtype});
  \]

- Displacements specified in number of elements
  
  – Displacements can be specified in bytes
    
    \[
    \text{MPI	extunderscore Type	extunderscore hindexed}();
    \]

- A shortcut if all blocks are the same length:
  
  \[
  \text{MPI	extunderscore Type	extunderscore create	extunderscore indexed	extunderscore block}()
  \]
Structured Records

- Allows different types to be combined
  
  ```c
  MPI_Type_struct(int count,
                  int *array_of_blocklengths,
                  MPI_Aint *array_of_displacements,
                  MPI_Datatype *array_of_types,
                  MPI_Datatype *newtype);
  ```

- Blocklengths specified in number of elements
- Displacements specified in bytes
Committing types

• In order for a user-defined derived datatype to be used as an argument to other MPI calls, the type must be “committed”.

  MPI_Type_commit(type);
  MPI_Type_free(type);

• Use commit after calling the type constructor, but before using the type anywhere else.

• Call free after the type is no longer in use (no one actually does this, but it makes computer scientists happy...)


Pack and Unpack

• Packs sparse structures into contiguous memory

```c
MPI_Pack(void* inbuf, int incount,
    MPI_Datatype datatype,
    void *outbuf, int outsize,
    int *position, MPI_Comm comm);

MPI_Unpack(void* inbuf, int insize,
    int *position, void *outbuf,
    int outcount,
    MPI_Datatype datatype,
    MPI_Comm comm);
```
Dealing with Communicators

- Many MPI operations deal with all the processes in a communicator
- MPI_COMM_WORLD by default contains every task in your MPI job
- Other communicators can be defined for more complex operations; for different parts of the task, to add topology, to segregate different kinds of messaging
- The next few slides cover the MPI functions that deal with:
  - Accessing communicators
  - Constructing communicators
  - Destroying communicators
Accessing Communicators

• int MPI_Comm_size(MPI_Comm comm, int *size)
  – Returns the size of the group in comm

• int MPI_Comm_rank(MPI_Comm comm, int *rank)
  – Returns the rank of the caller in that communicator

• int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
  – Returns if two communicators are the same, similar(same tasks but different ranks), or different
Creating Communicators

\[
\text{int MPI\_Comm\_dup(MPI\_Comm comm, MPI\_Comm *newcomm)} \\
\text{MPI::Intracomm MPI::Intracomm::Dup() const} \\
\text{MPI::Intercomm MPI::Intercomm::Dup() const} \\
\text{MPI::Cartcomm MPI::Cartcomm::Dup() const} \\
\text{MPI::Graph comm MPI::Graphcomm::Dup() const}
\]

- Creates an exact copy of the communicator
Creating Communicators

```c
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
MPI::Intracomm MPI::Intracomm::create(...) const
MPI::Intercomm MPI::Intercomm::create(...) const
```

- Creates a new communicator with the contents of group
  - Group must be a subset of Comm

```c
int MPI_Comm_split(comm, color, key, newcomm)
```
MPI_DIMS_CREATE

- A helper function for specifying a likely dimension decomposition.

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
```

```
void MPI::Compute_dims(int nnodes,int ndims, int dims[])
```

- `nnodes` - total nodes in grid
- `ndims` - number of dimensions
- `dims` - array returned with dimensions

Example:

```
MPI_Dims_create(6, 2, dims)
```

will return `(3,2)` in `dims`

```
MPI_Dims_create(6, 3, dims)
```

will return `(3,2,1)` in `dims`

No rounding or ceiling function provided
Destroying Communicators

\[ \text{int } \text{MPI\_Comm\_free(} \text{MPI\_Comm } \text{comm) } \]
\[ \text{void } \text{MPI::Comm::Free()} \]

- Destroys the named communicator
Topologies and Communicators

- MPI allows processes to be grouped in logical topologies
- Topologies can aid the programmer
  - Convenient naming methods for processes in a group
  - Naming can match communication patterns
  - A standard mechanism for representing common algorithmic concepts (i.e. 2D grids)
- Topologies can aid the runtime environment
  - Better mappings of MPI tasks to hardware nodes
  - Not really useful in the Clemson cluster environment....
Cartesian Topologies

int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)

MPI::Cartcomm MPI::Intracomm::CreateCart(...)  
MPI_CART_CREAT(…)

• comm_old - input communicator
• ndims - # of dimensions in cartesian grid
• dims - integer array of size ndims specifying the number of processes in each dimension
• periods - true/false specifying whether each dimension is periodic
• reorder - ranks may be reordered or not
• comm_cart - new communicator containing new topology.
Cartesian Inquiry Functions

- **MPI_Cartdim_get** will return the number of dimensions in a cartesian structure
  
  ```c
  int MPI_Cartdim_get(MPI_Comm comm, int *ndims);
  ```

- **MPI_Cart_get** provides information on an existing topology
  - Arguments roughly mirror the create call
    ```c
    int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods, int *coords);
    ```
  - Maxdims keeps a given communicator from overflowing your arguments
Cartesian Translator Functions

• Task IDs in a cartesian coordinate system correspond to ranks in a "normal" communicator.
  – point-to-point communication routines (send/receive) rely on ranks
    int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
    int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)

• Coords - cartesian coordinates
• rank - ranks
Cartesian Shift function

```c
int MPI_Cart_Shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)
```

- **direction** - coordinate dimension of shift
- **disp** - displacement (can be positive or negative)
- **rank_source and rank_dest** are return values
  - Use that source and dest to call MPI_Sendrecv
Cartesian Shift Example

MPI_Comm ICOMM;
int idims, reorder, myid, nxplus, nxminus,
int *periods, *icoords;
void *plane1, *plane2;

MPI_Cart_create(MPI_COMM_WORLD, 2, idims, periods, reorder, &ICOMM);
MPI_Comm_rank(ICOMM, &myid);

//Find nearest neighbor ranks

MPI_Cart_shift(ICOMM, 0, 1, &nxminus, &nxplus);
MPI_Cart_shift(ICOMM, 1, 1, &nyminus, &nyplus);
MPI_Cart_get(ICOMM, 2, &idims, periods, icoords);

MPI_Sendrecv(plane1, NY, MPI.DOUBLE_PRECISION, nxplus, isend,
plane2, NY, MPI.DOUBLE_PRECISION, nxminus, irecv,
ICOMM, istatus);