Parallel Programming with MPI

Matthew Pratola

mp00aa@sandcastle.cosc.brocku.ca

mpratola@hotmail.com

2003.

Contents

- Introduction
- Communicators
- Blocking and Non-Blocking P2P Communication
- Global Communication
- Datatypes
- Conclusion

Introduction

- The Message Passing Interface (MPI) is an open standard.
- Freely available implementations include MPICH and LAM.
- Designed for portability, flexibility, consistency and performance.

MPI*: The Standard

- Covers:
  - Point to point (P2P) communication
  - Collective operations
  - Process groups
  - Communication domains
  - Process topologies
  - Profiling interface
  - F77 and C bindings

*ver1.x

Hello World

```
#include <stdio.h>
#include “mpi.h”
main(int argc, char **argv){
    int myrank;    //rank of process
    int p;         //number of processes
    int source;    //rank of sender
    int dest;      //rank of receiver
    int tag=50;    //tag for messages
    char message[100];
    MPI_Status status;
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    if(myrank!=0){
        sprintf(message,”Hello from %d!”,myrank);
        dest=0;
        MPI_Send(message,strlen(message)+1,MPI_CHAR,dest,tag,MPI_COMM_WORLD);
    } else {
        for(source=1;source<p;source++) {
        MPI_Recv(message,100,MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);
        printf(“%s\n”,message);
        }
    }
    MPI_Finalize();
}
```

Hello World - Output

mpirun -np 4 ./helloworld

Hello from 1!

Hello from 2!

Hello from 3!

Communicators

- Allow one to divide up processes so that they may perform (relatively) independent work.
- A communicator specifies a communication domain made up of a group of processes.

Intracommunicator

- An intracommunicator is used for communicating within a single group of processes.
- Default intracommunicator MPI_COMM_WORLD includes all available processes at runtime.
- Can also specify the topology of an intracommunicator.

Intercommunicator

- Used for communication between two disjoint groups of processes.
- Limits on the type of communication that can occur with intercommunicators.

P2P: The Basics

- Send and receive functions allow the communication of *typed* data with an associated *tag*
- *typing* is required for portability
- *tagging* enables a form of message selection or identification

P2P: Blocking Send/Recv

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

- Perform a standard mode blocking send and standard mode blocking receive, respectively

P2P: Blocking Call Semantics

- The receive can be started before the corresponding send is initiated.
- Receive will only return after message data is stored in receive buffer.
- The send can be started whether or not the corresponding receive has been posted.

P2P: Blocking Call Semantics

- The send will not return until the message data and envelope have been stored such that the sender is free to access and alter the send buffer.
- The MPI implementation may buffer your send allowing it to return almost immediately.

P2P: Blocking Call Semantics

- If the implementation does not buffer the send, the send will not complete until the matching receive occurs.
- This dual-mode behaviour should have no effect on well constructed programs, but could be the source of deadlock in poorly constructed programs.

Example

- Always succeeds, even if no buffering is done.

```
if(rank==0)
{
   MPI_Send(...);
   MPI_Recv(...);
}
else if(rank==1)
{
   MPI_Recv(...);
   MPI_Send(...);
}
```

Example

- Will always deadlock, no matter the buffering mode.

```
if(rank==0)
{
   MPI_Recv(...);
   MPI_Send(...);
}
else if(rank==1)
{
   MPI_Recv(...);
   MPI_Send(...);
}
```

Example

- Only succeeds if sufficient buffering is present -- strictly unsafe!

```
if(rank==0)
{
   MPI_Send(...);
   MPI_Recv(...);
}
else if(rank==1)
{
   MPI_Send(...);
   MPI_Recv(...);
}
```

P2P: Order Property

- Successive messages sent by a process to another process in the same communication domain are received in the order they are sent.
- Exception is the *MPI_ANYSOURCE* wildcard which allows a receive to be filled from any sender.
- Messages received from many processes have no "global" order.

P2P: Other common calls

int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status);

int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status);

P2P: Non-blocking comm.

- Often performance can be improved by overlapping communication and computation.
- A non-blocking send posts the send and returns immediately.
- A complete-send operation finishes the non-blocking send.

P2P: Non-blocking cnt’d

- A non-blocking receive posts the receive and returns immediately.
- A complete-receive operation finishes the non-blocking receive.
- There is compatibility between blocking and non-blocking calls.

P2P: N-B send/recv (post)

int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,MPI_Comm comm, MPI_Request *request)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

P2P: N-B send/recv (completion)

int MPI_Wait(MPI_Request *request, MPI_Status *status)
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

- MPI_Wait will return when the operation identified by *request* is complete.
- MPI_Test will return *flag==true* when the operation identified by *request* is complete.

MPI: Collective Communications

- Collective communications transmit (and possibly operate on) data among all processes in a given communications group.
- Barrier (synchronization), global communications, global reduction operations.

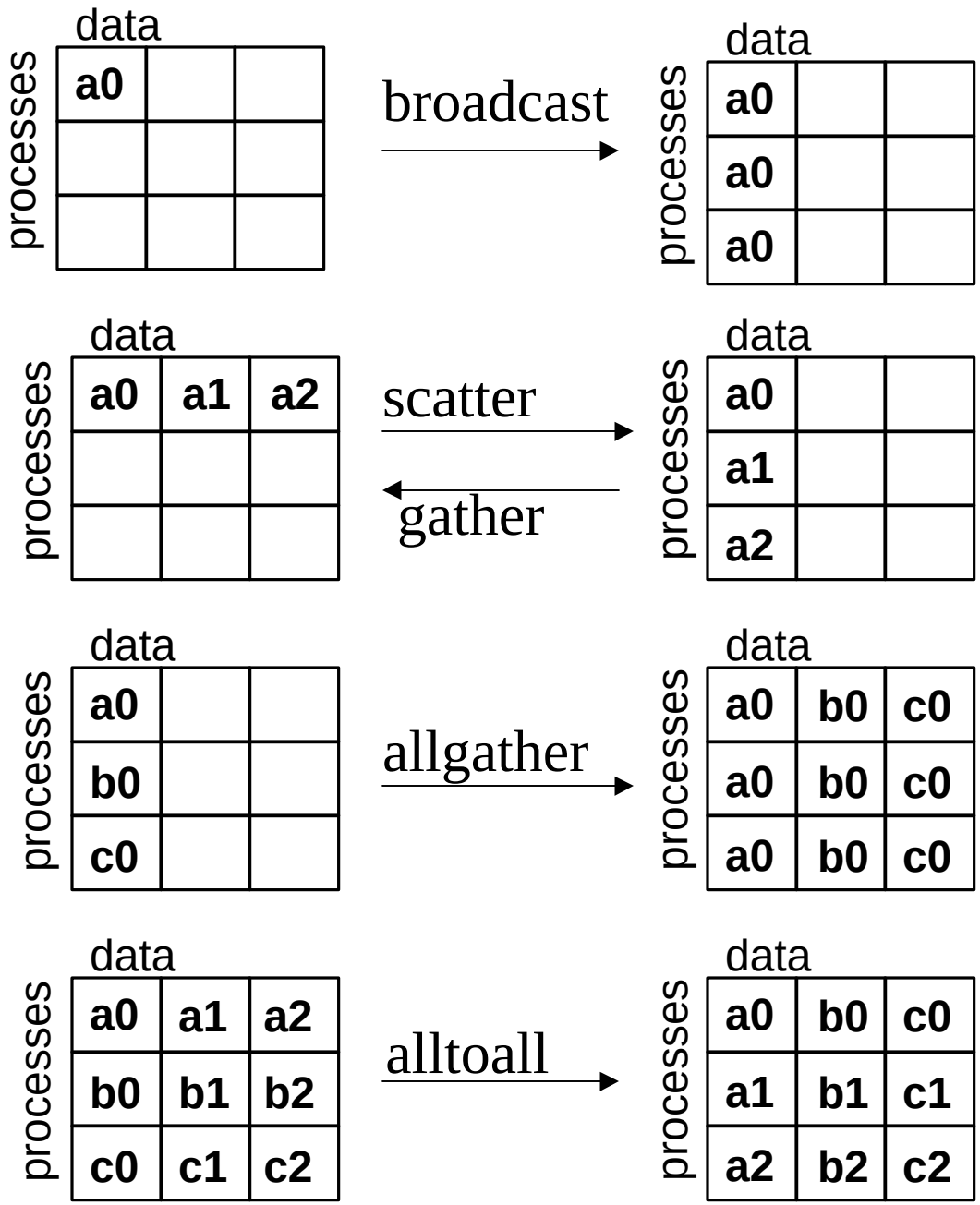
MPI: Barrier

int MPI_Barrier(MPI_Comm comm)

- blocks the caller until all group members have called it.
- syncs all processes in a group to some known point.

MPI: Global Communications

- only come in blocking mode calls .
- no tag provided, messages are matched by order of execution within the group.
- intercommunicators are not allowed.
- you cannot match these calls with P2P receives.


data
processes
a0
broadcast
data
processes
a0
a0
a0
data
processes
a0
a1
a2
scatter
gather
data
processes
a0
a1
a2
data
processes
a0
b0
c0
allgather
data
processes
a0
b0
c0
a0
b0
c0
a0
b0
c0
data
processes
a0
a1
a2
b0
b1
b2
c0
c1
c2
alltoall
data
processes
a0
b0
c0
a1
b1
c1
a2
b2
c2

MPI: Broadcast

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

- broadcasts a message from process with rank *root* in *comm* to all other processes in *comm*.

MPI: Gather

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

- each process in *comm* (including *root* itself) sends its *sendbuf* to *root*.
- the *root* process receives the messages in *recvbuf* in rank order.

MPI: Scatter

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

- inverse to MPI_Gather.
- *sendbuf* is ignored by all non-*root* processes.

MPI: Allgather

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

- similar to MPI_Gather except now all processes receive the result.
- *recvbuf* is NOT ignored.
- j^{th} block of data sent from each process is received by every process and placed in the j^{th} block of *recvbuf*.

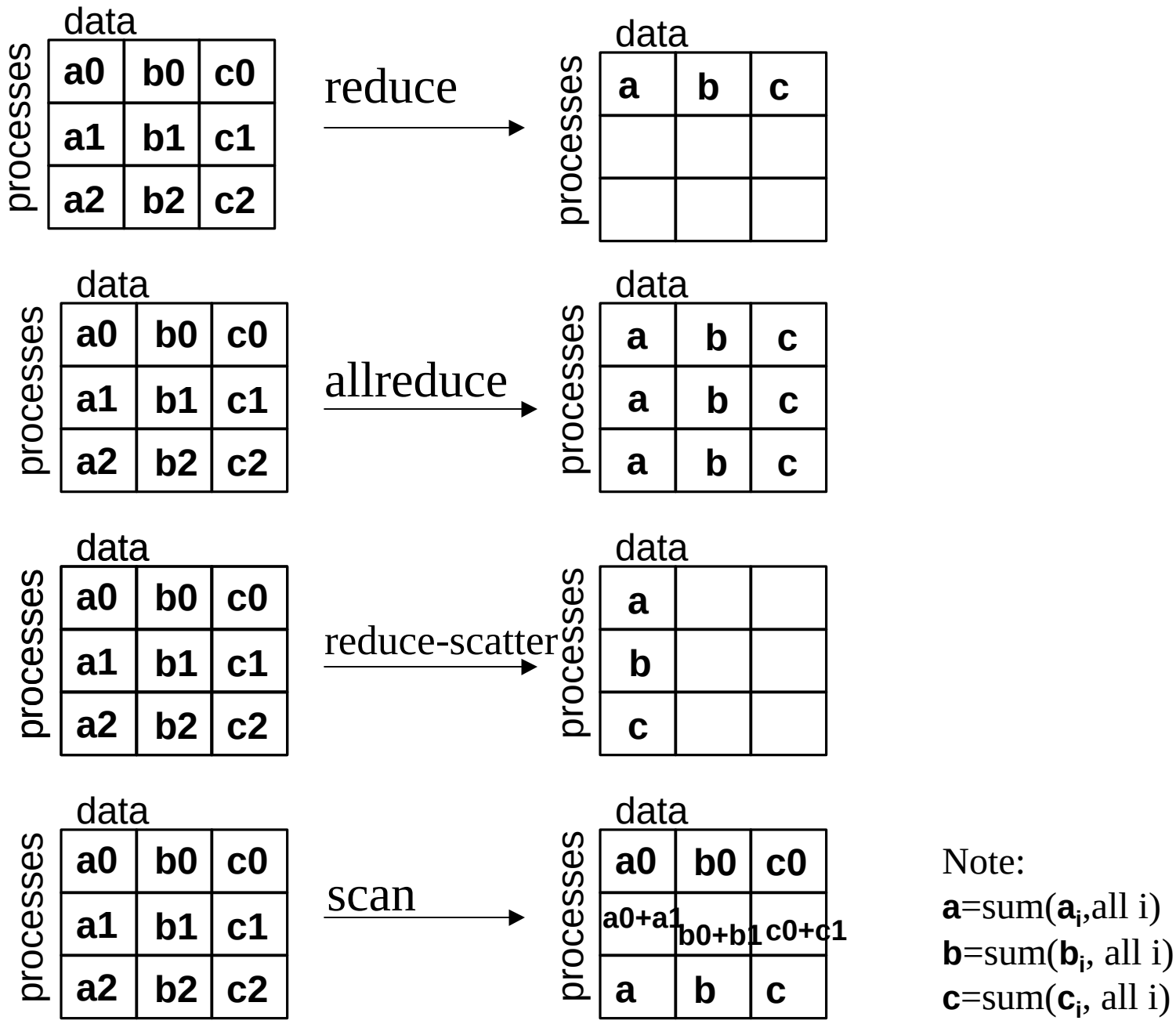
MPI: Alltoall

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

- similar to MPI_Allgather except each process sends distinct data to each of the receivers.
- the j^{th} block sent from process i is received by process j and placed in the i^{th} block of *recvbuf*.

MPI: Reduction Operations

- Perform global reduce operations across all members of a group.
- Many predefined operations come with MPI.
- Ability to define your own operations.



Note:
a=sum($\mathbf{a}_i$,all i)
b=sum($\mathbf{b}_i$, all i)
c=sum($\mathbf{c}_i$, all i)
(here i=0..2)

MPI: Predefined Reduction Operators

MPI_MAX - maximum

MPI_MIN - minimum

MPI_SUM - sum

MPI_PROD - product

MPI_LAND - logical and

MPI_BAND - bitwise and

MPI_LOR - logical or

MPI_BOR - bitwise or

MPI_LXOR - logical xor

MPI_BXOR - bitwise xor

MPI_MAXLOC - max value and location

MPI_MINLOC - min value and location

MPI: Reduce

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

- combines elements provided by input buffer of each process in the group using operation *op*.
- returns combined value in the output buffer of process with rank *root.*

MPI: Allreduce

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

- same as MPI_Reduce except the result appears in the *recvbuf* of all processes in the group.

MPI: Reduce-Scatter

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

- results of reduction operation are distributed among all processes in the group.
- that is, each process contains a unique result.

MPI: Scan

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

- process i will contain the results of reduction of data from processes 0..i.

MPI: Datatypes and data passing

- Passing data becomes more complicated when its not a single variable or a contiguous array of values.
- When the data to be passed is not contiguous in memory, there are two options: derived datatypes and packing.

MPI: Derived Datatypes

- Derived datatypes specify the sequence of primitive datatypes (MPI_INT, MPI_DOUBLE, etc...).
- and, the sequence of (byte) displacements.
- This forms the type map: {(type0,disp0),(type1,disp1),...}

MPI: Constructing Derived Datatypes

int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

- Constructs a typemap consisting of *count* replications of *oldtype* in contiguous memory locations.

eg: oldtype=[], count=4, newtype=[][][][]

MPI: Constructing Derived Datatypes

int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

- Allows replication of a datatype consisting of equally spaced blocks.

eg: oldtype=[xxx], count=3, blocklength=2, stride=3, newtype=[xxx][xxx]xxx[xxx][xxx]xxx[xxx][xxx](xxx)

MPI: Constructing Derived Datatypes

int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

- similar to vector, except *stride* is now calculated in byte-multiples instead of *oldtype*-multiples.

eg: oldtype=[xxx], count=3, blocklength=2, stride=7, newtype=[xxx][xxx]x[xxx][xxx]x[xxx][xxx](x)

Example

```
struct Partstruct
{
    char class; //particle class
    double d[6]; //particle coordinates
    char b[7]; //some additional information
}
struct Partstruct particle[1000];
int i, dest, rank;
MPI_Comm comm;
MPI_Datatype locationtype; //datatype for locations
MPI_Type_hvector(1000,3,sizeof(Partstruct),MPI_DOUBLE,&locationtype);
MPI_Type_commit(&locationtype);
MPI_Send(particle[0].d,1,locationtype,dest,tag,comm);
```

MPI: Constructing Derived Datatypes

int MPI_Type_indexed(int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

- displacement between successive blocks need not be equal

int MPI_Type_hindexed(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

- similar, except counting with bytes

MPI: Constructing Derived Datatypes

int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)

- here we now also allow multiple old-datatypes

MPI: Constructing Derived Datatypes

- Must always call
  MPI_Type_commit(MPI_Datatype *datatype)
  before using a constructed datatype.
- Constructed datatypes deallocated using
  MPI_Type_free(MPI_Datatype *datatype)

MPI: Pack and Unpack

- We explicitly pack non-contiguous data into a contiguous buffer for transmission, then unpack it at the other end.
- When sending/receiving packed messages, must use MPI_PACKED datatype in send/receive calls.

MPI: Packing

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

- will pack the information specified by *inbuf* and *incount* into the buffer space provided by *outbuf* and *outsize*
- the current packing call will pack this data starting at offset *position* in the *outbuf*

MPI: Unpacking

int MPI_Unpack(void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)

- unpacks message to *outbuf*.
- updates the *position* argument so it can be used in a subsequent call to MPI_Unpack.

Example

```
int i;
char c[100];
char buffer[110];
int position=0;

//pack
MPI_Pack(&i,1,MPI_INT,buffer,110,&position,MPI_COMM_WORLD);
MPI_Pack(c,100,MPI_CHAR,buffer,110,&position,MPI_COMM_WORLD);
//send
MPI_Send(buffer,position,MPI_PACKED,1,0,MPI_COMM_WORLD);
...
```

Example cnt’d

```
...
//correspoding receive
//position=0
MPI_Recv(buffer,110,MPI_PACKED,1,0,MPI_COMM_WORLD,&status);
//and unpack
MPI_Unpack(buffer,110,&position,&i,1,MPI_INT,MPI_COMM_WORLD);
MPI_Unpack(buffer,110,&position,c,100,MPI_CHAR,MPI_COMM_WORLD
    );
...
```

MPI: Other Pack/Unpack Calls

```
int MPI_Pack_size(int incount, MPI_Datatype datatype,
    MPI_Comm comm, int *size)
```

- allows you to find out how much space (bytes) is required to pack a message.
- enables user to manage buffers for packing.

MPI: Derived Datatypes vs. Pack/Unpack

- Pack/Unpack is quicker/easier to program.
- Derived datatypes are more flexible in allowing complex derived datatypes to be defined.
- Pack/Unpack has higher overhead.
- Derived datatypes are better if the datatype is regularly reused.

Conclusion

- Much can be accomplished with a small subset of MPI's functionality.
- But there is also a lot of depth to MPI's design to allow more complicated functionality.
- We haven't even touched MPI-2!

Useful Links:

http://www-unix.mcs.anl.gov/dbpp/text/node1.html
http://www.phy.duke.edu/brahma/beowulf_online_book/
http://www-unix.mcs.anl.gov/dbpp/web-tours/mpi.html
http://www-unix.mcs.anl.gov/mpi/tutorial
http://www.google.com

This document is hosted at:
http://lie.math.brocku.ca/twolf/htdocs/MPI