Parallel Hardware/Software (review)

Hardware
- Uniform memory access; cache coherent (cc-UMA)
  - SMPs
- Non-uniform memory access; cache coherent (cc-NUMA)
  - SGI Origin 2000; HP Exemplar
- Non-uniform memory access; not cache coherent (ncc-NUMA)
  - MPPs; clusters

Note: locality is good even on ccUMA machines (cache)

Software models
- Single thread of control
  - Automatic parallelization (autotasking); requires cc
  - Data parallel (HPF); no cc required; NUMA implicit
- Multiple threads of control
  - Message passing; no cc required; NUMA explicit
  - “Threads”; requires cc
Message passing programs

- Separate processes
- Separate address spaces (distributed memory model)
- Processes execute independently and concurrently
- **Processes transfer data cooperatively**

**Single Program Multiple Data (SPMD)**
- All processes are the same program, but act on different data

**Multiple Program Multiple Data (MPMD)**
- Each process may be a different program.

MPI Supports both of these. Not all computers support MPMD.

Cooperative Data Transfer

**Send** operation in process 1 is matched by **receive** operation in process 2:
Models related to message passing

Active messages
- Message contains address of handler that processes incoming data
- No receive operations
- Separate bulk transfer mechanism

Remote memory operations (get/put, 1-sided communication)
- Process may directly access memory of another process with get and put operations
- Other synchronization mechanisms to coordinate access

Common features
- Separate processes
- Separate address spaces (distributed memory model)
- Processes execute independently and concurrently

MPI History

History
- MPI Forum: government, industry and academia. All major players represented.
- Formal process began November 1992
- Draft presented at Supercomputing 1993
- Final standard (1.0) published May 1994
- Clarifications (1.1) published June 1995
- MPI-2 process began April, 1995
- MPI-1.2 finalized July 1997
- MPI-2 finalized July 1997

Current status
- Public domain versions available from ANL/MSU (MPICH), OSC (LAM)
- Proprietary versions available from all parallel computer vendors

This is why MPI is important.
**MPI Overview**

**MPI covers**
- Point-to-point communication (send/receive)
- Collective communication
- Support for library development

**MPI design goals**
- Portable
- Provides access to fast hardware (user space/zero copy)
- Based on existing practice (MPI-1)

**MPI does not cover**
- Fault tolerance
- Parallel/distributed operating system

---

**An MPI Application**

An MPI application

![Diagram of communication paths between processes](image)

The elements of the application are:
- **4 processes**, numbered zero through three
- Communication paths between them

The set of processes plus the communication channels is called “**MPI_COMM_WORLD**”. More on the name later.
“Hello World” — C

```
#include <mpi.h>
main(int argc, char *argv[])
{
    int me, nprocs
    MPI_Init(&argc, &argv)
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs)
    MPI_Comm_rank(MPI_COMM_WORLD, &me)
    printf("Hi from node %d of %d\n", me, nprocs)
    MPI_Finalize()
}
```

Compiling and Running

Different on every machine.

Compile:
```
mpicc -o hello hello.c
mpif77 -o hello hello.c
```

Start four processes (somewhere):
```
mpirun -np 4 ./hello
```
“Hello world” output

Run with 4 processes:

\[
\begin{align*}
\text{Hi from node 2 of 4} \\
\text{Hi from node 1 of 4} \\
\text{Hi from node 3 of 4} \\
\text{Hi from node 0 of 4}
\end{align*}
\]

Note:
- Order of output is not specified by MPI
- Ability to use `stdout` is not even guaranteed by MPI!

Point-to-point communication in MPI

![Diagram showing point-to-point communication between processes 1 and 2 using `MPI_Send(data, ...)` and `MPI_Recv(data, ...)`.](image)
Point-to-point Example

Process 0 sends array “A” to process 1 which receives it as “B”

1:
```c
#define TAG 123
double A[10];
MPI_Send(A, 10, MPI_DOUBLE, 1, TAG, MPI_COMM_WORLD)
```

2:
```c
#define TAG 123
double B[10];
MPI_Recv(B, 10, MPI_DOUBLE, 0, TAG, MPI_COMM_WORLD, &status)
```
or
```c
MPI_Recv(B, 10, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status)
```

Some Predefined datatypes

C:
```c
MPI_INT
MPI_FLOAT
MPI_DOUBLE
MPI_CHAR
MPI_LONG
MPI_UNSIGNED
```

Fortran:
```c
MPI_INTEGER
MPI_REAL
MPI_DOUBLE_PRECISION
MPI_CHARACTER
MPI_COMPLEX
MPI_LOGICAL
```

Language-independent
```c
MPI_BYTE
```
**Source/Destination/Tag**

**src/dest**

- **dest**
  - Rank of process message is being sent to (destination)
  - Must be a valid rank (0...N-1) in communicator
- **src**
  - Rank of process message is being received from (source)
  - “Wildcard” **MPI_ANY_SOURCE** matches any source

**tag**

- On the sending side, specifies a label for a message
- On the receiving side, must match incoming message
- On receiving side, **MPI_ANY_TAG** matches any tag

**Status argument**

In C: MPI_Status is a structure

- **status.MPI_TAG** is tag of incoming message
  (useful if **MPI_ANY_TAG** was specified)
- **status.MPI_SOURCE** is source of incoming message
  (useful if **MPI_ANY_SOURCE** was specified)
- How many elements of given datatype were received
  
  MPI_Get_count(IN status, IN datatype, OUT count)

In Fortran: status is an array of integer

  integer status(MPI_STATUS_SIZE)
  status(MPI_SOURCE)
  status(MPI_TAG)

In MPI-2: Will be able to specify **MPI_STATUS_IGNORE**
Guidelines for using wildcards

Unless there is a good reason to do so, do not use wildcards

Good reasons to use wildcards:

- Receiving messages from several sources into the same buffer but don’t care about the order (use `MPI_ANY_SOURCE`)

- Receiving several messages from the same source into the same buffer, and don’t care about the order (use `MPI_ANY_TAG`)

Exchanging Data

- Example with two processes: 0 and 1
- General data exchange is very similar

```
This is wrong! (for MPI)
```
**Deadlock**

The MPI specification is wishy-washy about deadlock.

- A **safe** program does not rely on system buffering.
- An **unsafe** program may rely on buffering but is not as portable.

Ignore this. MPI is all about writing portable programs.

Better:

- A **correct** program does not rely on buffering
- A program that relies on buffering to avoid deadlock is **incorrect**.

In other words, it is your fault if your program deadlocks.

---

**Non-blocking operations**

Split communication operations into two parts.

- First part initiates the operation. It does not block.
- Second part waits for the operation to complete.

```c
MPI_Request request;

MPI_Recv(buf, count, type, dest, tag, comm, status) =
MPI_Irecv(buf, count, type, dest, tag, comm, &request) +
MPI_Wait(&request, &status)

MPI_Send(buf, count, type, dest, tag, comm) =
MPI_Isend(buf, count, type, dest, tag, comm, &request) +
MPI_Wait(&request, &status)
```
Using non-blocking operations

```c
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;

Process 0:
MPI_Irecv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &request)
MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD)
MPI_Wait(&request, &status)

Process 1:
MPI_Irecv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &request)
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD)
MPI_Wait(&request, &status)
```

- No deadlock
- Data may be transferred concurrently

Using non-blocking operations (II)

Also possible to use nonblocking send:

```c
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
MPI_Request request;
MPI_Status status;
p=1-me; /* calculates partner in 2 process exchange */

Process 0 and 1:
MPI_Isend(A, 100, MPI_DOUBLE, p, MYTAG, WORLD, &request)
MPI_Recv(B, 100, MPI_DOUBLE, p, MYTAG, WORLD, &status)
MPI_Wait(&request, &status)
```

- No deadlock
- “status” argument to MPI_Wait doesn’t return useful info here.
- Better to use Irecv instead of Isend if only using one.
**Overlapping communication and computation**

On some computers it may be possible to do useful work while data is being transferred.

```c
MPI_Request requests[2];
MPI_Status statuses[2];

MPI_Irecv(B, 100, MPI_DOUBLE, p, 0, WORLD, &request[1])
MPI_Isend(A, 100, MPI_DOUBLE, p, 0, WORLD, &request[0])

.... do some useful work here ....

MPI_Waitall(2, requests, statuses)
```

- **Irecv/Isend** initiate communication
- Communication proceeds “behind the scenes” while processor is doing useful work
- Need both **Isend** and **Irecv** for real overlap (not just one)
- Hardware support necessary for true overlap
- This is why “o” in “LogP” is interesting.

---

**Operations on MPI_Request**

- **MPI_Wait** *(INOUT request, OUT status)*
  - Waits for operation to complete
  - Returns information (if applicable) in status
  - Frees request object (and sets to MPI_REQUEST_NULL)
- **MPI_Test** *(INOUT request, OUT flag, OUT status)*
  - Tests to see if operation is complete
  - Returns information in status if complete
  - Frees request object if complete
- **MPI_Request_free** *(INOUT request)*
  - Frees request object but does not wait for operation to complete
- **MPI_Waitall** *(..., INOUT array_of_requests, ...)*
- **MPI_Testall** *(..., INOUT array_of_requests, ...)*
- **MPI_Waitany/MPI_Testany/MPI_Waitsome/MPI_Testsome**

**MPI_Cancel** cancels or completes a request. Problematic.
Non-blocking communication gotchas

Obvious caveats:

1. You may not modify the buffer between `Isend()` and the corresponding `Wait()`. Results are undefined.

2. You may not look at or modify the buffer between `Irecv()` and the corresponding `Wait()`. Results are undefined.

3. You may not have two pending `Irecv()`s for the same buffer.

Less obvious gotchas:

4. You may not look at the buffer between `Isend()` and the corresponding `Wait()`.

5. You may not have two pending `Isend()`s for the same buffer.

Why the `isend()` restrictions?

- Everyone agrees they are user-unfriendly.
- Restrictions give implementations more freedom

Situation:
- Heterogeneous computer
- Byte order is different in process 1 and process 2

Implementation (example):
- Swap bytes in the original buffer
- Send the buffer
- Swap bytes back to original order

Comments:
- Implementation does not have to allocate any additional space.
- No implementations that currently do this (but there was)
- There are other scenarios that have the same restrictions
**Semantics vs. Implementation**

Distinguish between *semantics* and *implementation* of a routine.

**Semantics**

What you have to know about a routine in order to use it correctly.

**Implementation**

Low-level details of how a library routine is constructed in order to implement a certain semantics.

Ideal world: only semantics important  
Real world: implementation may be important for performance

---

** MPI_Send semantics**

Most important:

- Buffer may be reused after MPI_Send() returns  
- May or may not block until a matching receive is called (non-local)

Others:

- Messages are non-overtaking  
- Progress happens  
- Fairness not guaranteed

**MPI_Send does not require a particular implementation, as long as it obeys these semantics.**
Review of Implementation from Lecture 6

2 protocols

**Eager**: send data immediately; use pre-allocated or dynamically allocated remote buffer space.

- One-way communication (fast)
- Requires buffer management
- Requires buffer copy
- Does not synchronize processes (good)

**Rendezvous**: send request to send; wait for ready message to send

- Three-way communication (slow)
- No buffer management
- No buffer copy
- Synchronizes processes (bad)

Point-to-point Performance (review)

How do you model and measure point-to-point communication performance?

\[
data \text{ transfer time} = f(\text{message size})
\]

Often a linear model is a good approximation

\[
data \text{ transfer time} = \text{latency} + \frac{\text{message size}}{\text{bandwidth}}
\]

- **latency** is startup time, independent of message size
- **bandwidth** is number of bytes per second

- linear is often a good approximation
- piecewise linear is sometimes better
- the latency/bandwidth model helps understand performance issues
Latency and bandwidth

- for short messages, latency dominates transfer time
- for long messages, the bandwidth term dominates transfer time

What are short and long?

\[ \text{latency term} = \frac{\text{bandwidth term}}{\text{latency}} \quad \text{when} \quad \text{latency} = \frac{\text{message size}}{\text{bandwidth}} \]

Critical message size = \text{latency} \times \text{bandwidth}

Example: \text{50 us} \times \text{50 MB/s} = \text{2500 bytes}

- messages longer than 2500 bytes are bandwidth dominated
- messages shorter than 2500 bytes are latency dominated

Effect of buffering on performance

Copying to/from a buffer is like sending a message

\[ \text{copy time} = \text{copy latency} + \frac{\text{message size}}{\text{copy bandwidth}} \]

For a single-buffered message:

\[ \text{total time} = \text{buffer copy time} + \text{network transfer time} \]
\[ = \text{copy latency} + \frac{\text{message size}}{\text{copy bandwidth}} + \frac{\text{message size}}{\text{network bandwidth}} \]

Copy latency is sometimes trivial compared to effective network latency

\[ \frac{1}{\text{effective bandwidth}} = \frac{1}{\text{copy bandwidth}} + \frac{1}{\text{network bandwidth}} \]

Lesson: \textbf{Buffering hurts bandwidth}
Mixing protocols for high performance of MPI_Send

Description

- **Eager** for short messages
- **Rendezvous** for long messages
- Switch protocols near latency-bandwidth product

Features

- Low latency for latency-dominated (short) messages
- High bandwidth for bandwidth-dominated (long) messages
- Reasonable memory management (upper limit on size of message that may be buffered)
- Non-ideal performance for some messages near critical size

Send Modes

**Standard**

- Send may not complete until matching receive is posted
  - `MPI_Send`, `MPI_Isend`

**Synchronous**

- Send does not complete until matching receive is posted
  - `MPI_Ssend`, `MPI_Issend`

**Ready**

- Matching receive must already have been posted
  - `MPI_Rsend`, `MPI_Irsend`

**Buffered**

- Buffers data in user-supplied buffer
  - `MPI_Bsend`, `MPI_Ibsend`
Communicators

What is **MPI_COMM_WORLD**?

A **communicator** consists of:

- **A group of processes**
  - Numbered 0 ... N-1
  - Never changes membership
- **A set of private communication channels between them**
  - Message sent with one communicator cannot be received by another.
  - Implemented using hidden message tags

Why?

- Enables development of safe libraries
- Restricting communication to subgroups is useful

Safe Libraries

User code may interact with library code.

- User code may send message received by library
- Library may send message received by user code

Triggers:

- Wildcard receives
- Non BSP communication

```c
start_communication();
library_call(); /* library communicates internally */
wait();
```
**Communicators**

Solution: library uses private communication domain

A communicator includes private virtual communication domain:
- All communication performed w.r.t a communicator
- Source/destination ranks with respect to communicator
- Message sent on one communicator cannot be received on another.

```c
MPI_Send(buffer, len, type, dest, tag, comm)
MPI_Recv(buffer, len, type, source, tag, comm, status)
```

**MPI_COMM_WORLD**

**MPI_COMM_WORLD** is
- A group of all initial MPI processes
- Communication channels between them

```
MPI_Send(buf, len, type, dest, tag, MPI_COMM_WORLD)
```

*dest* is a rank in **MPI_COMM_WORLD**
Creating and manipulating communicators

Create a communicator with same group as MPI_COMM_WORLD but different communication channels:

```c
MPI_Comm mycomm;
MPI_Comm_dup(MPI_COMM_WORLD, &mycomm);
```

This is a collective routine.
- Must be called on all processes in MPI_COMM_WORLD
- May not complete until all processes have called it

General principle:
All routines for creating and manipulating communicators are collective.

---

MPI_COMM_SPLIT

Partition a communicator into several sub-groups

```c
MPI_Comm_split( IN oldcomm, IN color, IN key,
                OUT newcomm)
```

`color`
- Partitions the original communicator
- All processes with the same color get same `newcomm`

`key`
- determines rank within new communicator
- higher key means higher rank
Example: rows and columns of matrix

### MPI_COMM_WORLD

```
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
```

Example: rows and columns of a matrix (II)

```c
MPI_Comm row, col;
int nnodes, me, len, myrow, mycol;

MPI_Comm_size(MPI_COMM_WORLD, &nnodes);
MPI_Comm_rank(MPI_COMM_WORLD, &me);

/* compute my row/column coordinates */
len = isqrt(nnodes);
myrow = me/len;
mycol = me%len;

/* create row and column communicators */
MPI_Comm_split(MPI_COMM_WORLD, myrow, me, &row);
MPI_Comm_split(MPI_COMM_WORLD, mycol, me, &col);
```
**Intercommunicators**

An intercommunicator is:
- Two non-overlapping groups
  - **local group** (includes the local process)
  - **remote group** (does not include the local process)
- Communication channels between processes in one group and processes in the other group (but not within a group!)
- Note: “local” and “remote” are logical, not necessarily physical

An intercommunicator can be used instead of a regular (intra) communicator in Point-to-point operations:
- **dest** or **src** argument is a rank in the remote group

In MPI-1, intercommunicators are rare
(MPI-2 dynamic process management makes use of intercommunicators)

**Collective Operations**

Collective communication is communication among a group of processes:
- Broadcast
- Synchronization (barrier)
- Global operations (reductions)
- Scatter/gather
- Parallel prefix (scan)
Barrier

**MPI_Barrier(communicator)**

No process leaves the barrier until all processes have entered it.

Model for collective communication:
- All processes in communicator must participate
- Process might not finish until have all have started.

Broadcast

**MPI_Bcast(buf, len, type, root, comm)**

- Process with rank = root is source of data (in buf)
- Other processes receive data

```c
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
if (myid == 0) {
    /* read data from file */
}
MPI_Bcast(data, len, type, 0, MPI_COMM_WORLD);
```

Note:
- All processes must participate
- MPI has no “multicast” that is matched by a receive
**Reduction**

Combine elements in input buffer from each process, placing result in output buffer.

\[
\text{MPI\_Reduce}(\text{indata}, \text{outdata}, \text{count}, \text{type}, \text{op}, \text{root}, \text{comm})
\]

\[
\text{MPI\_Allreduce}(\text{indata}, \text{outdata}, \text{count}, \text{type}, \text{op}, \text{comm})
\]

- Reduce: output appears only in buffer on root
- Allreduce: output appears on all processes

Operation types:
- MPI\_SUM
- MPI\_PROD
- MPI\_MAX
- MPI\_MIN
- MPI\_BAND
- arbitrary user-defined operations on arbitrary user-defined datatypes

**Reduction example: dot product**

```c
/* distribute two vectors over all processes such that
 processor 0 has elements 0...99
 processor 1 has elements 100...199
 processor 2 has elements 200...299
 etc. */

double dotprod(double a[100], double b[100])
{
    double gresult = lresult = 0.0;
    integer i;
    /* compute local dot product */
    for (i = 0; i < 100; i++) lresult += a[i]*b[i];
    MPI_Allreduce(lresult, gresult, 1, MPI_DOUBLE,
                  MPI\_SUM, MPI\_COMM\_WORLD);
    return(gresult);
}
```
Data movement: all-to-all

All processes send and receive data from all other processes.

\[
\text{MPI\_Alltoall}(\text{sendbuf}, \text{sendcount}, \text{sendtype}, \\
\text{recvbuf}, \text{recvcount}, \text{recvtype}, \\
\text{comm})
\]

For a communicator with N processes:
- \text{sendbuf} contains N blocks of \text{sendcount} elements each
- \text{recvbuf} receives N blocks of \text{recvcount} elements each
- Each process sends block \(i\) of \text{sendbuf} to process \(i\)
- Each process receives block \(i\) of \text{recvbuf} from process \(i\)

Example: multidimensional FFT (matrix transpose)

Other collective operations

There are many more collective operations provided by MPI:

\text{MPI\_Gather/Gatherv/Allgather/Allgatherv}
- each process contributes local data that is gathered into a larger array

\text{MPI\_Scatter/Scatterv}
- subparts of a single large array are distributed to processes

\text{MPI\_Reduce\_scatter}
- same as Reduce + Scatter

\text{Scan}
- prefix reduction

The “v” versions allow processes to contribute different amounts of data
Semantics of collective operations

For all collective operations:

- Must be called by all processes in a communicator

Some collective operations also have the “barrier” property:

- Will not return until all processes have started the operation
- **MPI_Barrier, MPI_Allreduce, MPI_Alltoall**, etc.

Others have the weaker property:

- May not return until all processes have started the operation
- **MPI_Bcast, MPI_Reduce, MPI_Comm_dup**, etc.

Performance of collective operations

Consider the following implementation if **MPI_Bcast**:

```c
if (me == root) {
    for (i = 0; i < N; i++) {
        if (i != me) MPI_Send(buf, ..., dest=i, ...);
    }
} else {
    MPI_Recv(buf, ..., src=i, ...);
}
```

**Non-scalable**: time to execute grows linearly with number of processes.

High-quality implementations of collective operations use algorithms with better scaling properties if the network supports multiple simultaneous data transfers.

- Algorithm may depend on size of data
- Algorithm may depend on topology of network
An implementation of MPI_Bcast

Broadcast to N nodes can be done in log(N) steps.

Why datatypes?

Motivation for basic datatypes:

- Automatic data conversion on heterogeneous systems
  - different sizes
  - different formats
- Automatic size calculation on any system
  - useful in Fortran (no sizeof)
- More natural
  - Specify count, not length in bytes

Heterogeneous?

- Many applications are hype
- Calculation on Cray plus Visualization on SGI is example of a possibly good reason to support heterogeneity
User-defined datatypes

Applications can define arbitrary composite datatypes

Motivation

• Naturalness
  • Row or column of a matrix
  • Complex data structure

• New functionality
  • Reduction functions on complex data types
  • Ability to send different types of data in same message

• Convenience
  • Automatic local gather/scatter of data

• Performance
  • Possibly

But:

• Can be difficult to understand
• Can hurt performance if not careful
• Not appropriate for dynamic types

User-defined datatypes: Contiguous

New datatype: 5 contiguous integers

```c
MPI_Datatype mp_type;
MPI_Type_contiguous(5, MPI_INT, &mp_type);
MPI_Type_commit(&mp_type);
/* ... use datatype ... */
MPI_Send(buf, 3, mp_type, dest, tag, comm);
/* ... */
MPI_Type_free(&mp_type);
```

- `MPI_TYPE_CONTIGUOUS` creates the new datatype
- `MPI_TYPE_COMMIT` makes it available for use
- New datatype can be used anywhere a basic datatype can be used
- `MPI_TYPE_FREE` deallocates storage
Contiguous datatype example

typedef struct {
    int a[5];
} multi_precision_real;

multi_precision_real x[100], y[100];
MPI_Datatype mp_type;
MPI_Op MP_ADD
void mp_add(void *a, void *b, MPI_Datatype type);
...
MPI_Op_create(mp_add, 1, &MP_ADD);
MPI_Type_contiguous(5, MPI_INT, &mp_type);
MPI_Type_commit(&mp_type);
...
MPI_Reduce(x, y, 100, mp_type, MP_ADD, 0, comm);

Vector datatypes

Common situation: column of a matrix (C) or row of a matrix (Fortran)
Strided data
Other type constructors

**Vector, Hvector**
- Strided arrays, stride specified in elements or bytes

**Struct**
- Arbitrary data at arbitrary displacements

**Indexed**
- Like vector but displacements, blocks may be different lengths
- Like struct, but single type and displacements in elements

**Hindexed**
- Like Indexed, but displacements in bytes

**Other:**
- Absolute addresses possible using **MPI_Address** and **MPI_BOTTOM**.
- “holes” in top, bottom or middle of datatypes possible.

When to use user-defined datatypes

What’s the catch?

**Complex datatypes can kill performance**

- Most implementations pack data into a contiguous buffer and send
- Implementation packing is much slower than user packing
- Hidden holes in apparently contiguous datatype can dramatically reduce performance
Datatype recommendation

For contiguous data: use datatypes.

For non-contiguous data:

- Structure code so that there is a clean interface to communication
- Write two versions of the communication module
  - quick and dirty
  - “the MPI Way”

Quick and dirty means:

- Pack the data into your own buffer
- Send as a contiguous MPI datatype

Really quick and dirty (not recommended):

- Use use MPI_BYTE for everything
- Only use if alignment prevents tight packing

MPI_PACKED

PVM style: pack+send ... receive+unpack

```c
int bigbuf[1000];
int a, b, pos;
double c;
position = 0;
MPI_Pack(&a, 1, MPI_INT, bigbuf, 1000, &pos, comm);
MPI_Pack(&b, 1, MPI_INT, bigbuf, 1000, &pos, comm);
MPI_Pack(&c, 1, MPI_DOUBLE, bigbuf, 1000, &pos, comm);
MPI_Send(bigbuf, pos, MPI_PACKED, dest, tag, comm);

MPI_Recv(bigbuf, 1000, MPI_PACKED, src, tag, comm);
MPI_Unpack(bigbuf, 1000, &pos, &a, 1, MPI_INT, comm);
MPI_Unpack(bigbuf, 1000, &pos, &b, 1, MPI_INT, comm);
MPI_Unpack(bigbuf, 1000, &pos, &c, 1, MPI_DOUBLE, comm);
```
When to use MPI_PACKED

Having MPI pack the data for you is guaranteed to be slower than packing it yourself. No pipelining possible.

Bad reasons:
- Porting a PVM code that uses pvm_pack/pvm_unpack
- Don’t want to learn about datatypes

Good reasons:
- Need to unpack data incrementally because data is self-describing
- Need to pack data incrementally because data gathering code is separate from data sending code
- Datatypes impractical
  - Used once
  - Too complex

Other MPI features

- **Timing**
- Persistent communication
- Combined send/receive
- Attributes
- **Topologies**
- **Profiling Interface**
- Thread safety

- **MPI-2**
**Timing**

Double precision wallclock time, in seconds.

```c
double t1, t2;
t1  = MPI_Wtime();
    .... do some work ...

t2  = MPI_Wtime();
printf("Elapsed time is %f seconds\n", t2-t1);
```

Notes:
- Time starts at some arbitrary point in the past
- Note times not synchronized unless `MPI_WTIME_IS_GLOBAL`

---

**Accurate timing is not simple**

Three standard problems

- Processes are unsynchronized to start
- Load imbalance shows up in collective and point-to-point operations
- Extra synchronization to avoid problems 1+2 causes network contention
**Communicator Topologies**

Many applications have logical communication topology. E.g.:

![Diagram of a 4x4 grid connecting 16 processes](image)

- Processes communicate only with connected processes

---

**Communicator Topologies (II)**

MPI can understand logical topology information.

**Uses:**
- Reorder processes to map effectively to hardware topology
- Convenience

**Implementation:**
- Communicators with topologies are regular intracommunicators with extra information associated with them.
- Topologies can be implemented with attributes caching.

**Recommendation/Opinion:**
- Topologies do no harm
- Performance improvement rare but may become important on clusters of SMPs
**Topology functions**

Cartesian topologies

- `MPI_CART_CREATE`
- `MPI_DIMS_CREATE`
- `MPI_CARTDIMS_GET`
- `MPI_CART_GET`
- `MPI_CART_RANK`
- `MPI_CART_COORDS`
- `MPI_CARTSHIFT`
- `MPI_CART_SUB`
- `MPI_CART_MAP`

Graph topologies

- `MPI_GRAPH_CREATE`
- `MPI_GRAPHDIMS_GET`
- `MPI_GRAPH_GET`
- `MPI_GRAPH_NEIGHBORS_COUNT`
- `MPI_GRAPH_NEIGHBORS`
- `MPI_GRAPH_MAP`

**Profiling**

MPI provides a profiling interface

- Mechanism to make MPI functions available with name `PMPI_` as well as `MPI_`
- Program can be linked with `PMPI_` and `MPI_` “libraries” and replace specific `MPI_` routines with its own.
- `MPI_PCONTROL` function (no-op)

This allows

- User can implement specific `MPI_` functions as wrappers around the `PMPI_` functions
- Wrapper functions can record information about what was called and when.
- Wrapper functions may slightly modify functionality (e.g. replace regular mode send with synchronous send).
**MPI-2**

Dynamic process management
- Spawn new processes
- Client/server
- Peer-to-peer

One-sided communication
- Remote Get/Put/Accumulate
- Locking and synchronization mechanisms
- NOT “shared memory”

I/O
- Allows MPI processes to write cooperatively to a single file
- Makes extensive use of MPI datatypes to express distribution of file data among processes
- Allow optimizations such as collective buffering
- Actually implemented!

---

**Freely available MPI Implementations (I)**

**MPICH**
Developed at Argonne National Lab and Mississippi State Univ.
- Runs on
  - Networks of workstations (IBM, DEC, HP, IRIX, Solaris, SunOS, Linux, Win 95/NT)
  - MPPs (Paragon, CM-5, Meiko, T3D) using native M.P.
  - SMPs using shared memory
- Strengths
  - Free, with source
  - Easy to port to new machines and get good performance (ADI)
  - Easy to configure, build
  - Basis for many vendor implementations
- Weaknesses
  - Large
  - No virtual machine abstraction NOWs
Freely available MPI implementations (II)

**LAM** (Local Area Multicomputer)
Developed at the Ohio Supercomputer Center
- [http://www.mpi.nd.edu/lam](http://www.mpi.nd.edu/lam)
- Runs on
  - SGI, IBM, DEC, HP, SUN, LINUX
- Strengths
  - Free, with source
  - Virtual machine model for networks of workstations
  - Lots of debugging tools and features
  - Has early implementation of MPI-2 dynamic process management
- Weaknesses
  - Does not run on MPPs

Where to get more information

Home pages
- [http://www.mpi-forum.org](http://www.mpi-forum.org)
- [http://www.mcs.anl.gov/mpi](http://www.mcs.anl.gov/mpi)

Newsgroups
- comp.parallel.mpi

Books
- *Using MPI*, by Gropp, Lusk, Skjellum. The MIT Press
- *Parallel Programming with MPI*, by Pacheco. Morgan Kauffman