An Introduction to the Message Passing Interface

2/98

William Saphir

Lawrence Berkeley National Laboratory
NERSC Division
Phone: 510-486-5442 Fax: 510-495-2998
wcs@nersc.gov

Topics

Focus is on MPI, not on parallel algorithms.

- Background
- Basic MPI
- Example: heat equation
- Advanced MPI
- PVM vs. MPI
- Tools

Note: views expressed are those of the speaker, not necessarily (and often not) those of the MPI Forum.

Parallel Hardware/Software

Computer scientists deal with:

- **Address space models**
  - Shared address space (usually cache coherent)
  - Separate address spaces
- **Memory access models**
  - Uniform access to all of memory
  - Non-uniform access to all of memory (NUMA)

Computational scientists deal with:

- **Software models**
  - Single thread of control
    - Automatic parallelization (autotasking)
    - Data parallel (HPF)
  - Multiple threads of control
    - Message passing (separate address spaces)
    - "Threads" (shared address space)

Message passing programs

The "message passing" paradigm:

- 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

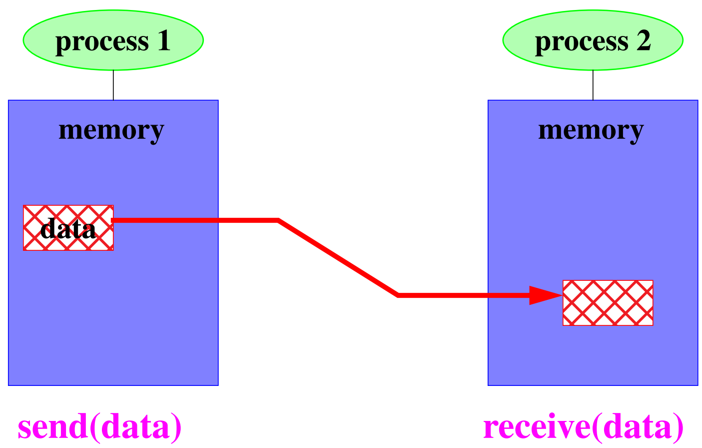
Multiple Program Multiple Data (MPMD)

- Each process may be a different program.

Note: The T3E does not support MPMD programs. This is a T3E problem, not an MPI problem.

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
- These are available on the T3E

Common features

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

Other models

Data Parallel (distributed memory)
- Compiler automatically distributes data among processes
- Parallelism is transparent to the user (in theory)
- Standard: High Performance Fortran (HPF)

Threads (shared memory single address space)
- Separate threads of control in (mostly) the same address space
- Explicit user control of threads, locking and synchronization
- Standard: Posix threads
- Not available on the T3E. Rarely used in scientific apps.

Work sharing (shared memory single address space)
- Single copy of the data in a single memory
- Each process does part of the work in a loop
- Transparent to the user (implemented by compiler)
- “Autotasking” on Cray vector systems (and others)

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 June1995
- 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

Other Libraries

APPL	Developed at NASA Lewis
Chameleon	Lightweight portable library from ANL
CHIMP	Developed in England
CMMD	Native CM-5 library, from TMC
Express	Proprietary (ParaSoft), portable
LAM	A version of MPI is written on top of this
MPL (EUI-H)	1st native message passing library on the IBM SP2 (SP1)
NX	Native library on Intel machines (Paragon, iPSC/860)
P4	Developed at ANL, successor to PARMACS
PARMACS	Developed at ANL, used widely in Europe
PICL	Emphasis on tracing
PVM	From ORNL/UTK
TCGMSG	Modeled on PARMACS
Zipcode	Developed at Livermore

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
- Based on existing practice

MPI does not cover

- Fault tolerance
- Parallel/distributed operating system

MPI-2 Overview

Major topics

- Dynamic process management
- One-sided communication (get/put)
- I/O
- Extended collective operations
- Fortran 90 and C++ bindings

Status

- Standard finalized July 1997
- I/O implementations from Sun, ROMIO

Sidetrack 1:
Parallel Applications

Approaches to Parallelism

Basic idea: each process gets a part of the problem

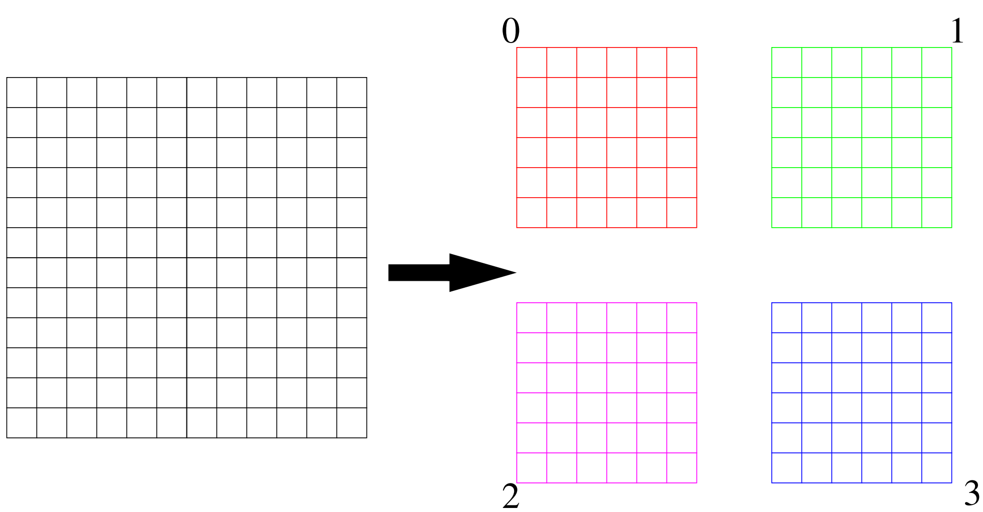
Data/domain decomposition

- Each process gets a different physical domain
- Typical iteration involves computation on interior of domain followed by exchange of boundary data

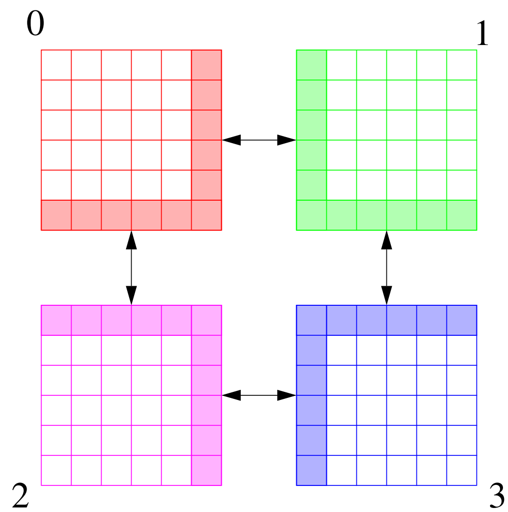
Functional decomposition

- Each process gets "all" the data
- Each process does different work on that data

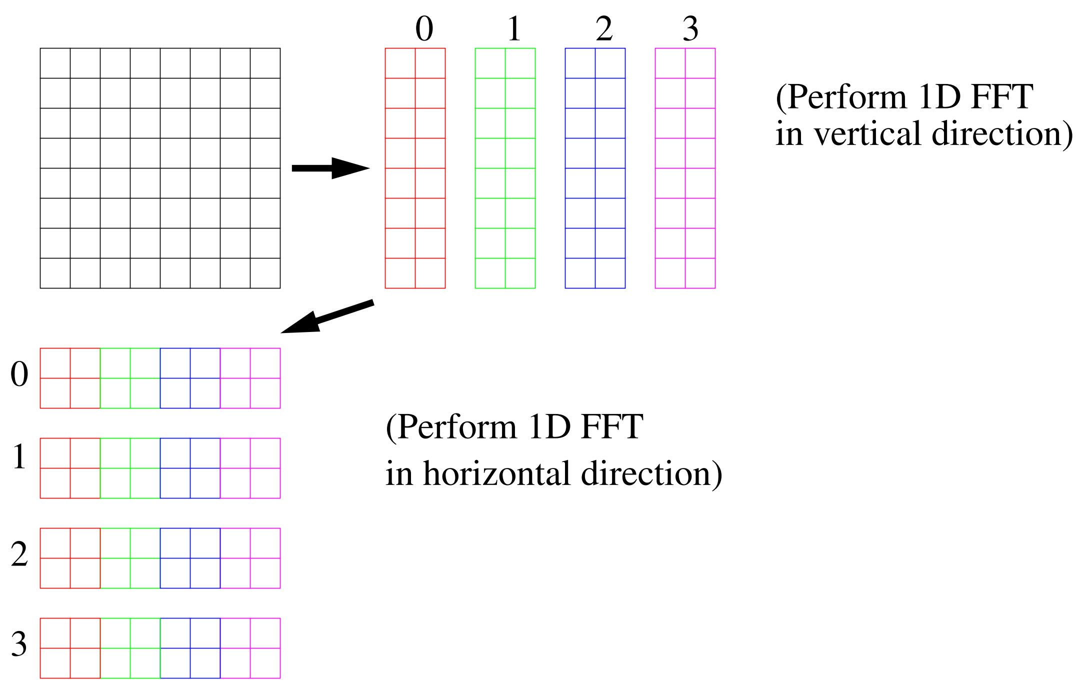
Decomposing a Single Grid or Array



Boundary exchange



All-to-all exchange: 2D FFT



Writing a parallel code

1. In most cases: **choose best serial algorithm**, not algorithm that is easiest to parallelize.

2. Three easy rules for a parallel implementation
- Load balance - keep all processors busy
- Large computation/communication ratio
- Few large messages rather than many small messages

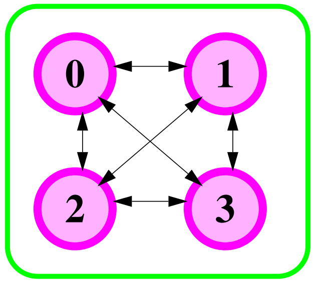
3. Finally, optimize MPI stuff.

MPI is a small part of the code and not all that important

End of sidetrack

An MPI Application

An MPI application



The elements of the application are:

- **4 processes**, numbered zero through three
- **Communication channels** between them

The set of processes plus the communication channels is called "**MPI_COMM_WORLD**". More on the name later.

"Hello World" — Fortran

```
program hello
implicit none
include 'mpif.h'

integer me, nprocs, ierr

call MPI_INIT(ierr)

call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, me, ierr)

print *, ' hello from proc ', me, ' of ', nprocs

call MPI_FINALIZE(ierr)
end
```

“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 on the T3E

Compile and run on 4 processors.

Non-malleable:

```
% cc -o hello hello.c
% f90 -o hello hello.f
% mpprun -n 4 ./hello
```

Malleable:

```
% cc -o hello -X 4 hello.c
% f90 -o hello -X 4 hello.f
% mpprun ./hello
% ./hello
% mpprun -n 4 ./hello # illegal
```

"Hello world" output

Run with 4 processes:

```
Hi from node 2 of 4
Hi from node 1 of 4
Hi from node 3 of 4
Hi from node 0 of 4
```

Note:

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

Critical MPI routines

`MPI_Init`

- Must always be called.
- State of program (e.g., number of processes) is undefined before
- Pass addresses of **`main()`** arguments in C.

`MPI_Finalize`

- Must always be called at end of program

`<mpi.h>` and **'mpif.h'**

- Define all predefined constants
- Declare functions (prototypes in C)
- Define all MPI-specific types (**`<mpi.h>`** only)

C and F ortran differences

Names

- In C, routines names start with **`MPI_`**
- Next letter is capitalized, rest are lower case
- In Fortran, case does not matter (technically, upped case only)

Return value vs. ierr argument

- In C, almost all routines are functions that return an integer error code
- In Fortran, almost all routines are subroutines that return an error code in an integer “ierr” argument

C and F ortran differences (II)

MPI objects

In C, all MPI objects are represented by new datatypes, which are handles.

- **`MPI_Comm`**
- **`MPI_Datatype`**
- **`MPI_Status`**
- **`etc`**

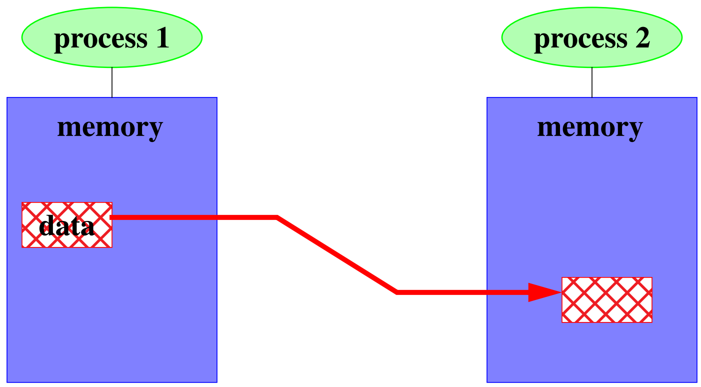
These are typedefs, so that declarations look like:

```
MPI_Comm mycomm;
```

In Fortran, MPI objects are represented by integers
Except for the status object, which is

```
integer status(MPI_STATUS_SIZE)
```

Point-to-point communication in MPI



MPI_Send(data, ...)   MPI_Recv(data, ...)

Point-to-point Example

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

1:

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

2:

```
#define TAG 123
double B[10];
MPI_Recv(B, 10, MPI_DOUBLE, 0, TAG,
         MPI_COMM_WORLD, &status)
```

or

```
MPI_Recv(B, 10, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status)
```

MPI_Send in detail

```
MPI_Send(buffer, count, datatype, dest, tag, comm)
```

`void *buffer`

- Address of data (any type)

`int count`

- Number of items to send (not necessarily full size of buffer)
- Number of items of given datatype, not number of bytes

`MPI_Datatype datatype`

- Describes type of data to be sent

`int dest`

- Rank of destination process in **`comm`**

`int tag`

- Integer label for message

`MPI_Comm comm`

- Communicator containing group of processes and communication channels

Use of datatypes

Predefined types offer:

- Automatic data conversion on heterogeneous machines
- No need to calculate size of data in bytes

Later on: user-defined datatypes provide more functionality.

Some Predefined datatypes

C:

```
MPI_INT
MPI_FLOAT
MPI_DOUBLE
MPI_CHAR
MPI_LONG
MPI_UNSIGNED
```

Fortran:

```
MPI_INTEGER
MPI_REAL
MPI_DOUBLE_PRECISION
MPI_CHARACTER
MPI_COMPLEX
MPI_LOGICAL
```

Language-independent

```
MPI_BYTE
```


MPI_Recv in detail

MPI_Recv(buf, count, datatype, src, tag, comm, status)

void *buffer
- Location to place incoming data. Must have enough space.

int count
- Maximum number of items of given datatype to receive
- Actual number may be less but more is erroneous

MPI_Datatype datatype
- Describes type of data to be received

int src
- Rank of source process in comm

int tag
- Integer label for message

MPI_Comm comm
- Communicator containing group of processes and communication channels

MPI_Status status
- Returns extra information about the incoming message

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

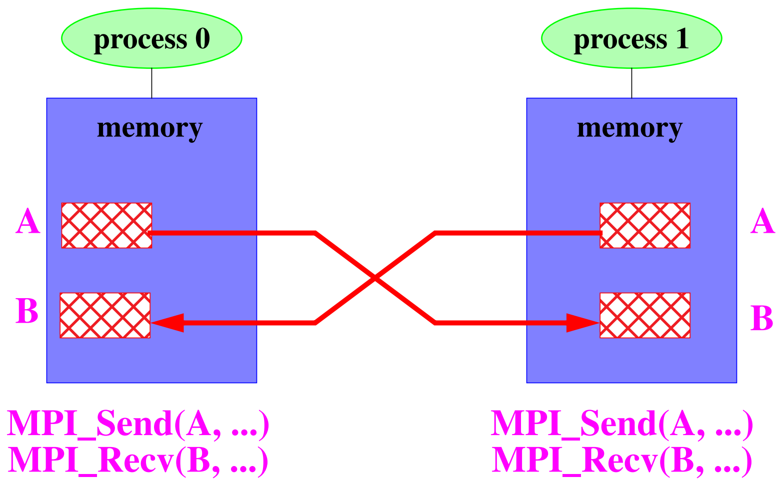
Unless there is a good reason to do so, do not use wildcards

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!

Exchanging data (II)

```
#define MYTAG 123
#define WORLD MPI_COMM_WORLD
```

Process 0:

```
MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD)
MPI_Recv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &status)
```

Process 1:

```
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD)
MPI_Recv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &status)
```

The problem:

- **`MPI_Send`** is a non-local operation — may not complete until a matching receive is posted.
- Both processes may block in **`MPI_Send`**, waiting for corresponding receive. This is called **deadlock**.
- Note: the above code will probably work with today's version of MPI on the T3E. Don't do it!

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. It is not helpful.

Better (practical wisdom):

- 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 it your program deadlocks.

One solution to deadlock

Switch the order of Send/Recv on one process.
Process 0:

```
MPI_Send(A, 100, MPI_DOUBLE, 1, MYTAG, WORLD)
MPI_Recv(B, 100, MPI_DOUBLE, 1, MYTAG, WORLD, &status)
```

Process 1:

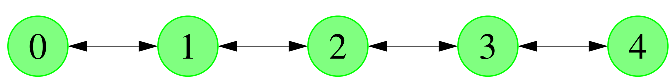
```
MPI_Recv(B, 100, MPI_DOUBLE, 0, MYTAG, WORLD, &status)
MPI_Send(A, 100, MPI_DOUBLE, 0, MYTAG, WORLD)
```

This seems to solve the problem, but:

- It works only in simple cases
- Even in this case, does not allow use of bidirectional hardware.
- In more complicated cases, it may avoid deadlock but serialize communication.
- Makes SPMD code non-symmetric and harder to follow.

Recommendation: **Use another approach**

Example of serialized communication



Process 0:

```
MPI_Recv(..., src = 1, ...)
MPI_Send(..., dest = 1, ...)
```

Process 1, 2, 3, (4):

```
MPI_Send(..., dest = me-1, ...)
MPI_Recv(..., src = me-1, ...)
MPI_Recv(..., src = me+1, ...)
MPI_Send(..., dest = me+1, ...)
```

Problem: communication happens one message at a time.

Note: while this is bad practice, with large messages it might be the fastest approach on an easily-congestible network.

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.

```
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

```
#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:

```
#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.

•

Sidetrack II: Example

Model Problem

Heat Equation:

$$\frac{\partial u}{\partial t} = \sigma \frac{\partial^2 u}{\partial x^2}$$

Straightforward discretization in 1-D, explicit (!) time stepping:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \sigma \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h}$$

Rearranging:

$$u_i^{n+1} = u_i^n + \frac{\sigma \Delta t}{h}(u_{i+1}^n - 2u_i^n + u_{i-1}^n) = u_i^n + \Delta u_i^n$$

Disclaimer: numerical algorithm is for illustration only

Model Problem: Fortran 77+

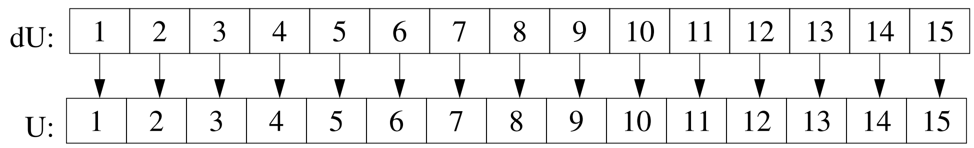
```
      parameter (N = 1000)
      real*8 U(N), dU(N)
c     calculate dU
      do i = 2, N-1
         dU(i) = (sigma * dt / h) * ( U(i+1) - 2*U(i)+ U(i-1))
      end do

c     update U
      do i = 2, N-1
         U(i) = U(i) + dU(i)
      end do
```

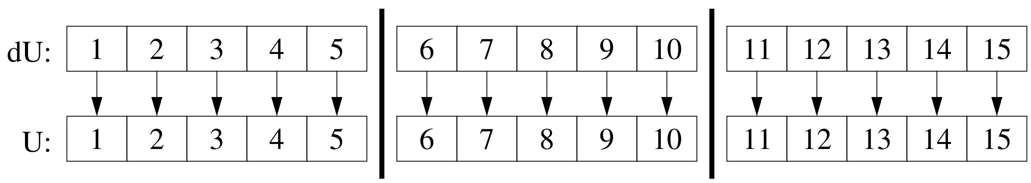
- go from 2 to N-1 to implement fixed temperature at boundary
- U initialized to an arbitrary temperature distribution
- dU(1) = dU(N) = 0

Model Problem: Data Distribution

Focus attention on **U = U + dU** (N = 15)
Layout of U and dU in serial memory:



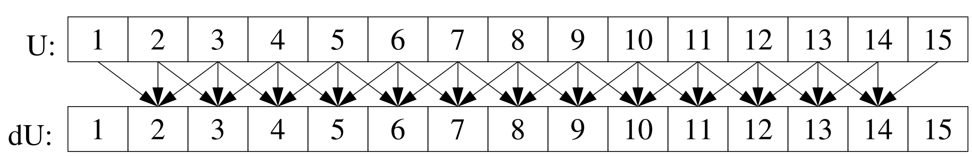
Possible layout of U and dU in distributed memory (3 processors):



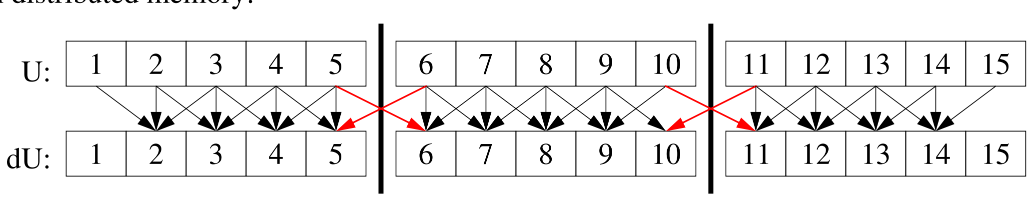
- Computation can be done entirely in parallel.
- U and dU must be “aligned”
- U and dU must be distributed evenly

Model Problem: Communication

Focus attention on **computation of dU** (3-point stencil)
In serial memory:



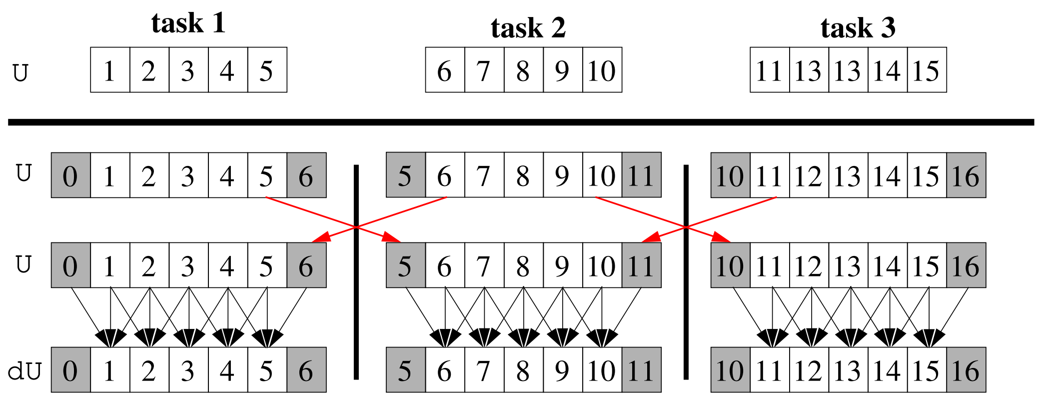
In distributed memory:



Computation can be done in parallel but requires some communication.

Model Problem Data Layout

- Add “shadow entries” to local arrays
- Break calculation into two parts: communication + computation
- Computation is entirely local



Model Problem Implementation: Setup

```
      program heat
      implicit none
      include 'mpif.h'

      integer N, MyN, ierr, me, nprocs, left, right
      logical is_right, is_left
      parameter (N = 1000)
c     It's impossible in F77 to determine the size of arrays
c     at runtime so use F90 instead.
      real, allocatable :: U(:), dU(:)

      call mpi_init(ierr)
      call mpi_comm_rank(MPI_COMM_WORLD, me)
      call mpi_comm_size(MPI_COMM_WORLD, nprocs)

      MyN = N/nprocs ! assume N is a multiple of nprocs for now
      allocate(U(0:MyN+1), dU(0:MyN+1))
```

Model Problem Implementation: Data Exchange

```
c   set up topology
    left = me - 1
    right = me + 1
    if (left .ge. 0) is_left = .TRUE.
    else is_left = .FALSE.
    if (right .lt. nprocs) is_right = .TRUE.
    else is_right = .FALSE.
c   ...
c   exchange data
    if (is_left) call mpi_irecv(U(0), 1, MPI_REAL,
   $    left, 0, MPI_COMM_WORLD, lrequest, ierr)
    if (is_right)call mpi_irecv(U(MyN+1), 1, MPI_REAL,
   $    right, 0, MPI_COMM_WORLD, rrequest, ierr)
    if (is_left) call mpi_send(U(1), 1, MPI_REAL,
   $    left, 0,  MPI_COMM_WORLD, ierr)
    if (is_right)call mpi_send(U(MyN), 1, MPI_REAL,
   $    right, 0, MPI_COMM_WORLD, ierr)
    if (is_left) call mpi_wait(lrequest, status)
    if (is_right)call mpi_wait(rrequest, status)
```


Model Problem Implementation: Data Update

```
c calculate dU
    do i = 1, MyN
       dU(i) = (sigma * dt / h) * ( U(i+1) - 2*U(i)+ U(i-1))
    end do

c update U
    do i = 1, MyN
       U(i) = U(i) + dU(i)
    end do
```

•

End of example

Overlapping communication and computation

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

```
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

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 after an **`Isend()`** and before the corresponding **`Wait()`**. Results are undefined.

2. You may not look at or modify the buffer after an **`Irecv()`** and before 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 after and **`Isend()`** and before 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

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`**

Send mode recommendations

1. Always use regular mode unless you have a good reason not to

2. Synchronous mode adds latency and increases synchronization. Don't use unless you have a good reason.

3. Ready mode is almost always a loser

- Few implementations give performance advantage
- Intermittent code failure due to race conditions hard to diagnose
- Extra synchronization necessary to use ready mode often outweighs any possible performance advantage

4. Buffered mode is rarely useful

- Hurts performance
- Difficult to use

Communicators

What is **`MPI_COMM_WORLD`**?

A **communicator** consists of:

- **A group of processes**
  - Numbered 0 ... N-1
  - Never changes membership
- **A set of communication channels between these processes**
  - Each communicator has its own set of channels.
  - Message sent with one communicator cannot be received by another.
  - Channel in the sense of a separate communication medium, not a serialized one.

Why?

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

Safe Libraries

User code may interact with user code.

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

Triggers:

- Wildcard receives
- Non BSP communication

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


Communicators

Solution: library uses private communication domain

A communicator includes private virtual communication domain:

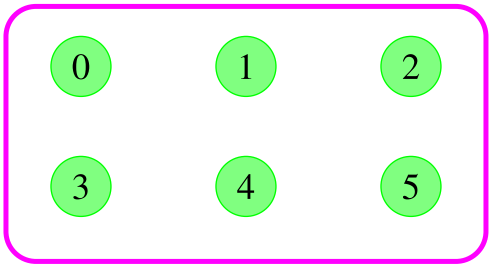
- All communication performed w.r.t a communicator
- Source/destination ranks with respect to communicator
- Message sent on one communicator cannot be received on another.

```
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_COMM_WORLD

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

dest is a rank in **MPI_COMM_WORLD**

Groups vs. Communicators

Communicator contains a group and communication channels

Group is a set of processes, with no communication channels

```
MPI_Comm MPI_COMM_WORLD;
MPI_Group worldgroup;
MPI_Comm_group(MPI_COMM_WORLD, &worldgroup);
```

Fortran:

```
integer MPI_COMM_WORLD, worldgroup, ierr
call MPI_COMM_GROUP(MPI_COMM_WORLD, worldgroup, ierr)
```

Creating and manipulating communicators

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

```
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

```
MPI_Comm_split( IN oldcomm, IN color, IN key,
                OUT newcomm)
MPI_Comm oldcomm, *newcomm;
int color, key;
```

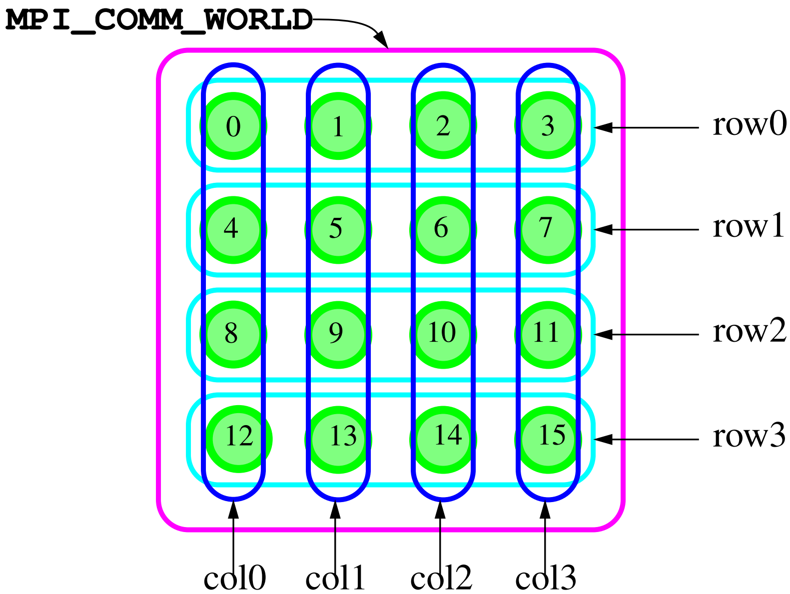
color

- 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



Example: rows and columns of a matrix (II)

```
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);
```

Group Management

Arbitrary manipulations on groups are possible:

- **`MPI_Group_union`**
- **`MPI_Group_intersection`**
- **`MPI_Group_difference`**
- **`MPI_Group_incl`**
- **`MPI_Group_excl`**
- **`MPI_Group_range_incl`**
- **`MPI_Group_range_excl`**

Once you have a the right group, you can create a communicator using:

```
MPI_Comm_create(communicator, group, &newcomm)
```

Read the spec.

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:

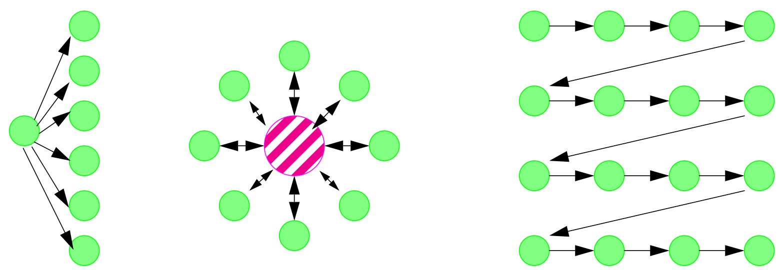
- **`dest`** or **`src`** argument is a rank in the remote group

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 started.

Broadcast

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

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

```
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.

```
MPI_Reduce(indata, outdata, count, type, op, root, comm)
MPI_Allreduce(indata, outdata, count, type, op, 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**
- ...

Reduction example: dot product

```
/* 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);
}
```


User-defined reduction operators

Consider the “maximum absolute value” operation, for which there is no predefined MPI operator. We’d like to do:

```
int a;
...
compute: max over all processes of |a|
```

Solution: User-defined reduction operator

```
MPI_Op MAXABS;
MPI_Op_create(<user function>, 1, &MAXABS);
MPI_Reduce(&a, &b, 1, MPI_INT, MAXABS, comm);
```

- First arg of **`Op_create`** is user-defined function
- Second arg of **`Op_create`** says whether function is commutative
- Function is assumed to be associative

Data movement: all-to-all

All processes send and receive data from all other processes.

```
MPI_Alltoall(sendbuf, sendcount, sendtype,
             recvbuf, recvcount, recvtype,
             comm)
```

For a communicator with N processes:

- **sendbuf** contains N blocks of **sendcount** elements each
- **recvbuf** receives N blocks of **recvcount** elements each
- Each process sends block **i** of **sendbuf** to process **i**
- Each process receives block **i** of **recvbuf** from process **i**

Example: multidimensional FFT (matrix transpose)

Other collective operations

There are many more collective operations provided by MPI:

MPI_Gather/Gatherv/Allgather/Allgatherv

- each process contributes local data that is gathered into a larger array

MPI_Scatter/Scatterv

- subparts of a single large array are distributed to processes

MPI_Reduce_scatter

- same as Reduce + Scatter

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`**:

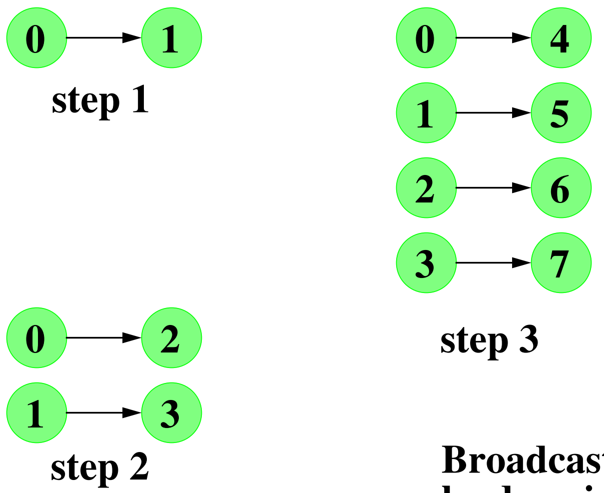
```
if (me == root) {
    for (i = 0; i < N; i++) {
        if (i != me) MPI_Send(buf, ..., dest=i, ...);
    }
} else {
    MPI_Recv(buf, ..., src=i, ...);
}
```

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.

Datatypes

- C
  - **`MPI_INT`**
  - **`MPI_FLOAT`**
  - **`MPI_DOUBLE`**
  - **`MPI_CHAR`**
  - **`MPI_LONG`**
  - etc
- Fortran
  - **`MPI_INTEGER`**
  - **`MPI_REAL`**
  - **`MPI_DOUBLE_PRECISION`**
  - **`MPI_CHARACTER`**
  - **`MPI_LOGICAL`**
  - etc
- Language-independent
  - **`MPI_BYTE`**

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

User-defined datatypes: Contiguous

New datatype: 5 contiguous integers

```
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

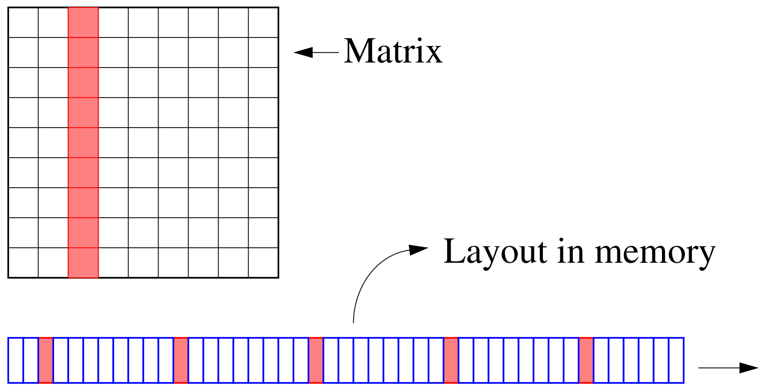
Contiguous datatype example

```
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



Sending a column (C) of a matrix (I)

Solution 1: one at a time

```
/* send 12th column of this matrix */
int a[49][103];
int i;
for (i = 0; i < 49; i++) {
   MPI_Send(&a[i][11], 1, MPI_INT, ...);
}
```

Solution 2: pack it up and send it

```
int col[49];
for (i = 0; i < 49; i++) col[i] = a[i][11];
MPI_Send(col, 49, MPI_INT, ...);
```

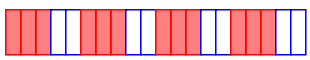
MPI_Type_vector

```
MPI_Type_vector(count, blocklength, stride, oldtype,
newtype)

int a[49][103];
MPI_Datatype columntype;
MPI_Type_vector(49, 1, 103, MPI_INT, &columntype);
MPI_Type_commit(&columntype);

/* send 12th column*/
MPI_Send(&a[0][12], 1, columntype, ...);
```

`Type_vector` can be used for arbitrary (fixed) blocklength/stride, e.g.:



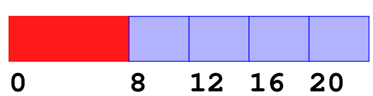
count = 4
blocklength = 3
stride = 5

Type_struct

```
MPI_Type_struct(count, array_of_blocklengths,
    array_of_displacements, array_of_types, newtype);
```

Allows arbitrary types/displacements

```
struct {
    double x;
    int a[4];
} mystruct;
```




```
int blocklengths[]={1, 4};
int displacements[]={0, 8};
MPI_Datatype types[] = {MPI_DOUBLE, MPI_INT};
MPI_type_struct(2, blocklengths, displacements, types,
                &mystruct_type);
```

Other type constructors

`Hvector`

- Like Vector but stride specified in bytes

`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:

- Possible to have absolute addresses in datatypes by using address for displacement (compute address using **`MPI_Address`**) and **`MPI_BOTTOM`** for location of buffer.
- Possible to put "holes" in datatypes so that **`contiguous`** arrays are aligned properly. See **`MPI_TYPE_UB`** and **`MPI_TYPE_LB`**. These can be very confusing. Avoid if possible. The MPI-2 function **`MPI_TYPE_RESIZED`** may make things easier.

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

BUT: vector datatypes on T3E are faster than hand packing!

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_P ACKED

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

```
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

Timing

Double precision wallclock time, in seconds.

```
double t1, t2;
t1 = MPI_Wtime();

.... do some work ...

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

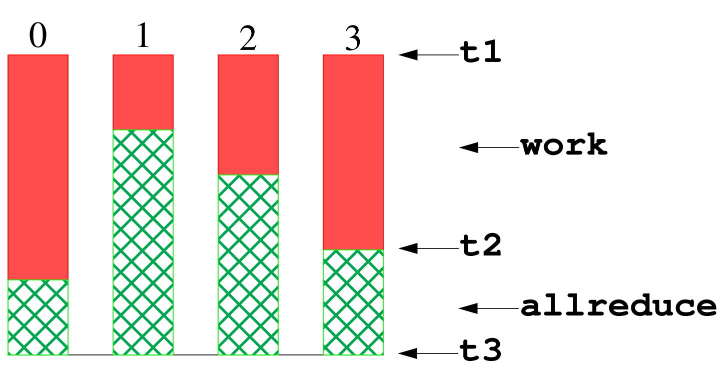
Notes:

- 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



Persistent communication

```
MPI_Send_init(buf, count, type, dest, tag, comm,
&request)
MPI_Start(request);
MPI_Wait(&request, &status);
...
MPI_Start(request);
MPI_Wait(&request, &status);
...
MPI_Start(request);
MPI_Wait(&request, &status);
```

- Send_init similar to isend, but doesn't start communication.
- Request object can be reused
- One-time startup cost

Persistent communication (II)

Related routines:

```
MPI_Recv_init
MPI_Ssend_init
MPI_Bsend_init
MPI_Rsend_init

MPI_Startall
```

The verdict on persistent communication: **don't bother**

- They provide no extra functionality.
- They generally do not improve performance because they don't nail down both sides of the communication.
- They introduce a new kind of object, a "persistent request", which is slightly different from a regular request and therefore confusing.

Combined Send/Receive

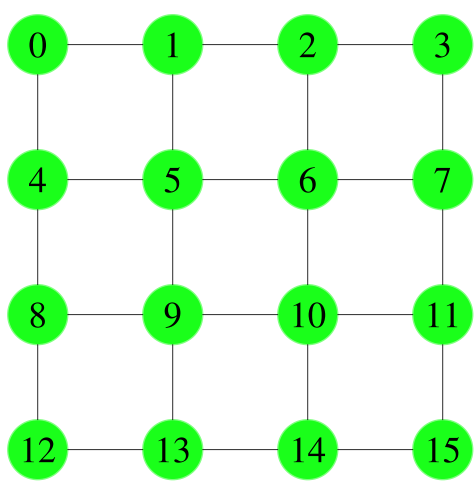
```
MPI_Sendrecv(sndbuf, sndcount, sndtype, dst, sndtag,
             rcvbuf, rcvcount, rcvtype, src, rcvtag,
             comm, status)
```

Recommendation: don't use this routine

- No performance advantage over Irecv+Send+Wait
- Easy to make mistakes since not pairwise (dst != src)

Communicator Topologies

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



- 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 possible
- MPI could do fine without

Topology functions

Cartesian topologies

```
MPI_CART_CREATE          convenience
MPI_DIMS_CREATE          hardware mapping
MPI_CARTDIM_GET
MPI_CART_GET
MPI_CART_RANK
MPI_CART_COORDS
MPI_CART_SHIFT
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
```

Profi ling

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).

PVM and MPI

PVM and MPI are different types of systems

- PVM is a distributed operating system with some message passing functionality. It is also a research project (which inspired MPI).
- MPI is a message passing library. It is a standard.

PVM problems

- Performance is bad
- Implementation is "non-standard" on the T3E
- No parallel libraries available

Recommendation:

- Use MPI unless you require special functionality available only in PVM.

Tools/libraries

NERSC has an effort to support tools from the DOE "ACTS" Toolkit. The following are some of the ACTS tools under consideration for support. Let us know if you might be interested in using them!

Aztec: solution of large sparse linear systems. Variety of krylov methods and preconditioners

Overture: support for adaptive mesh refinement

PETSc: infrastructure for solving PDEs using implicit methods

POOMA: templated class libraries for particle simulation

PVODE: parallel ODE solvers

SCALAPACK: dense linear algebra

Freely available MPI Implementation

MPICH

Developed at Argonne National Lab and Mississippi State Univ.

- `http://www.mcs.anl.gov/mpi/mpich`
- 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

Where to get more information

Home pages

- **`http://www.mpi-forum.org`**
- **`http://www.mcs.anl.gov/mpi`**

Newsgroups

- comp.parallel.mpi

Books

- **Using MPI**, by Gropp, Lusk, Skjellum. The MIT Press
- **MPI: The Complete Reference**, by Snir, Otto, Huss-Lederman, Walker, Dongarra. The MIT Press
- **Parallel Programming with MPI**, by Pacheco. Morgan Kauffman.
- Coming in 1998: **MPI-2: The Complete Reference**, by Gropp, Lederman, Lusk, Nitzberg, Saphir, Snir. The MIT Press

Additional material

Point-To-Point Communication Details

Why go into great detail on the semantics of point-to-point routines?

- Point-to-point is the core of a communication library
- Documentation not too clear
- Detailed understanding necessary for optimization
- Semantics are subtle

Note: Some applications use no point-to-point communication! (FFTs)

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

(sorry, English majors)

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 provides a great deal of flexibility for implementations

Implementation

- Buffering
- Protocols
- Performance model

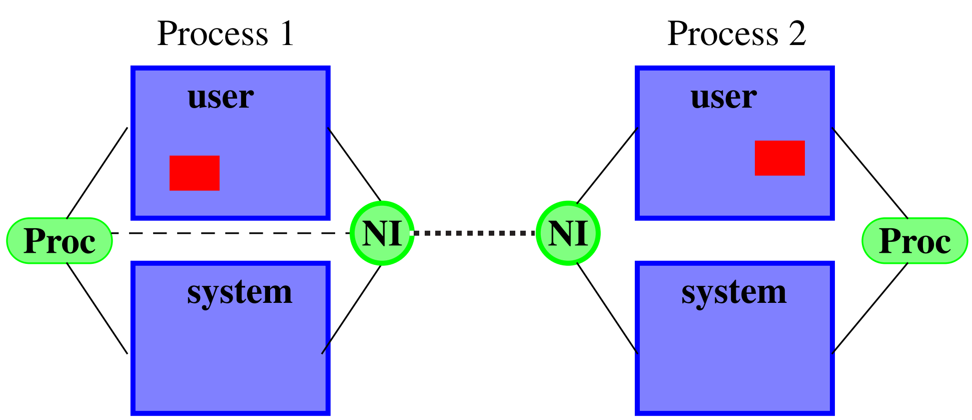
Buffering

Buffering: MPI may makes a temporary copy in system-managed memory

- Buffering allows **`MPI_Send`** to return before data has actually been transferred
- Buffering introduces extra data movement

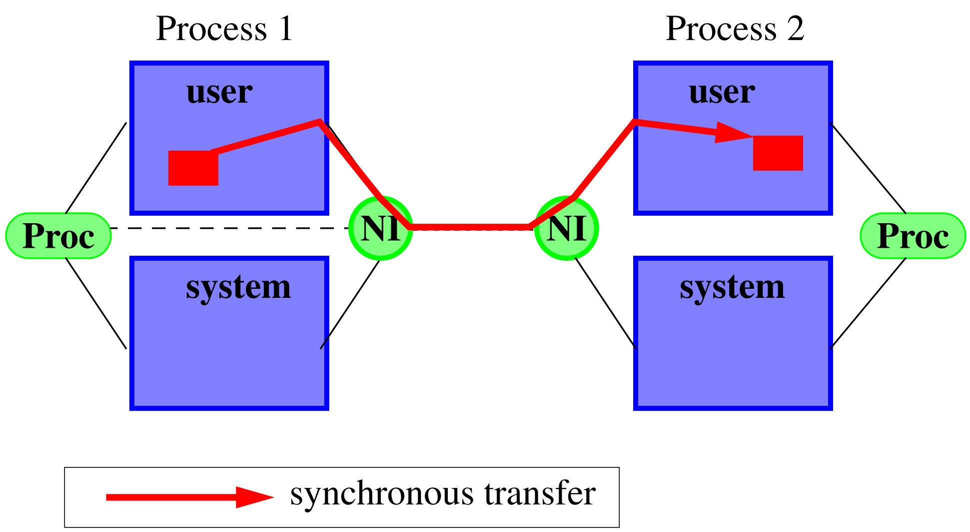
Buffering

Conceptual model:

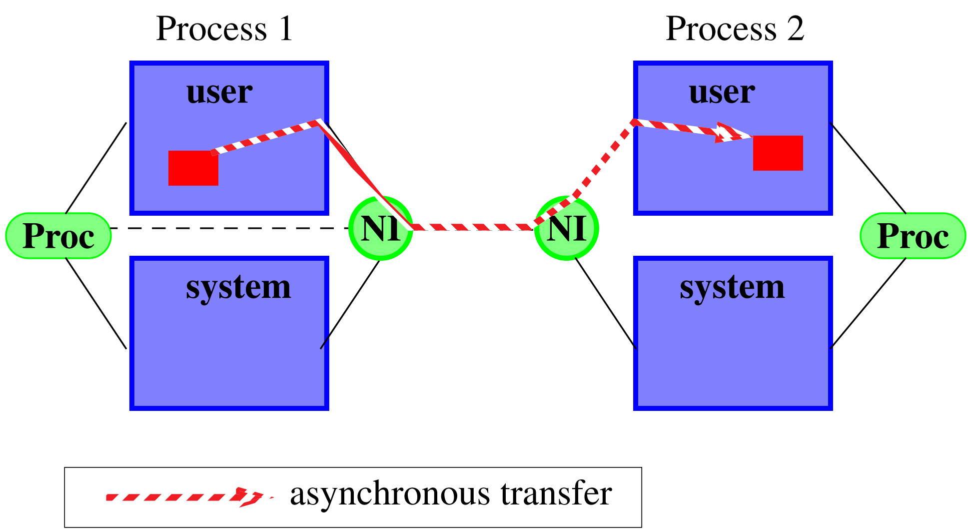


NI: Network interface
Proc: Processor

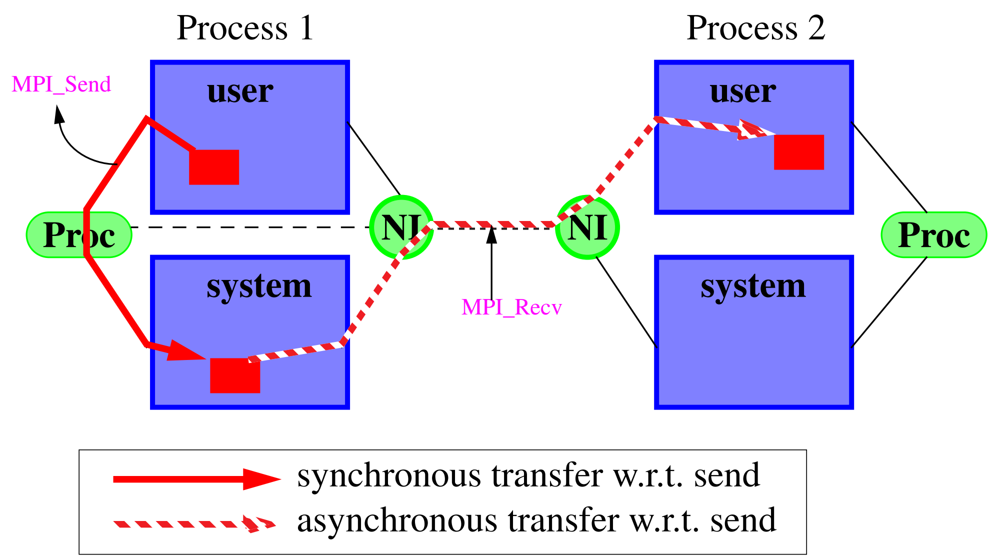
MPI_Send: No buffering



MPI_Isend: no buffering

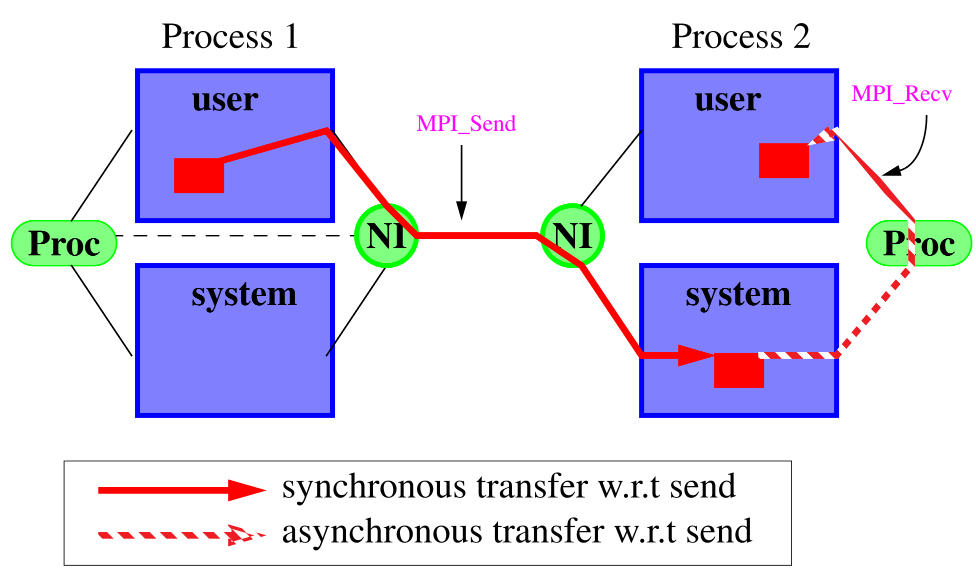


MPI_Send: local buffering



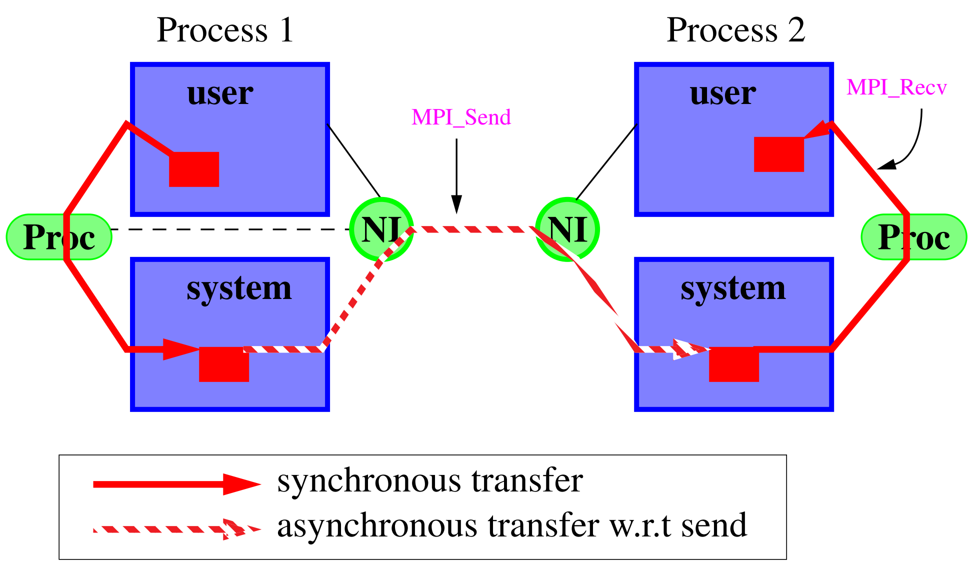
- Sender does not have to wait for receiver

MPI_Send: remote buffering



- Sender does not have to wait for receiver

MPI_Send: local and remote buffering



- This is the default communication strategy in PVM

Point-to-point Performance (sidetrack)

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

```
data transfer time = f(message size)
```

Often a linear model is a good approximation

```
data transfer time = latency + message size / 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?

```
latency term = bandwidth term
```

when

```
latency = message_size/bandwidth
```

Critical message size = `latency * bandwidth`

Example: `50 us * 50 MB/s = 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

```
copy time = copy latency + message_size / copy bandwidth
```

For a single-buffered message:

```
total time    = buffer copy time + network transfer time
              = copy latency + network latency
                  + message_size *
            (1/copy bandwidth + 1/network bandwidth)
```

copy latency is sometimes trivial compared to effective network latency (for message passing, not other types of communication)

```
1/effective bandwidth = 1/copy_bandwidth +
                        1/network_bandwidth
```

Lesson: **Buffering hurts bandwidth**

Protocols

Buffering all messages isn’t practical because buffering

- hurts bandwidth
- requires complicated/expensive memory management
- requires extra memory use

However buffering

- reduces synchronization

Buffering issues addressed through high-level protocols

Rendezvous Protocol

Description

- Sender sends request-to-send
- Receiver acknowledges
- Sender sends data

Implementation

- No buffering required (except for initial request)

Properties

- High latency (three-steps)
- High bandwidth (no extra buffer copy)

Eager Protocol

Description

- Sender sends data immediately

Implementation

- Usually must be buffered
- May be directly transferred if receive already posted

Features

- Low latency
- Low bandwidth (buffer copy)

Mixed protocol

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