

Message passing programs

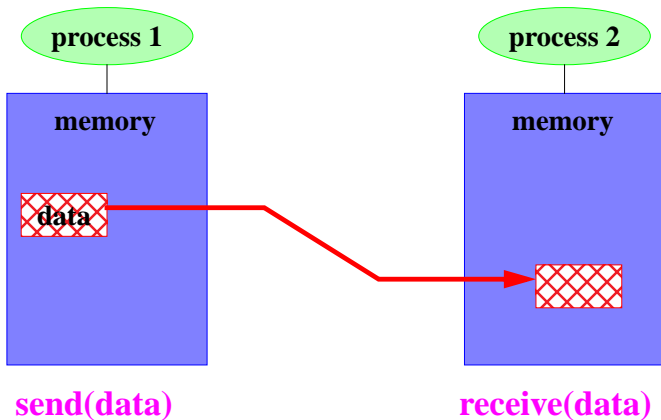
- Separate processes
- Separate address spaces (distributed memory model)
- Processes execute independently and concurrently
- **Processes transfer data cooperatively**

Single Program Multiple Data (SPMD)

- All processes are the same program, but act on different data

Cooperative Data Transfer

Send operation in process 1 is matched by **receive** operation in process 2:



Examples of message passing approaches

Basic idea: each process gets a part of the problem

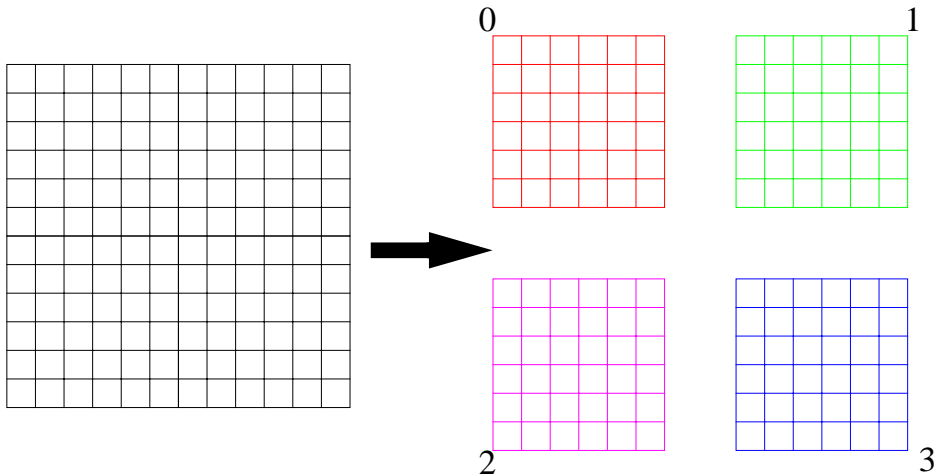
Data/domain decomposition

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

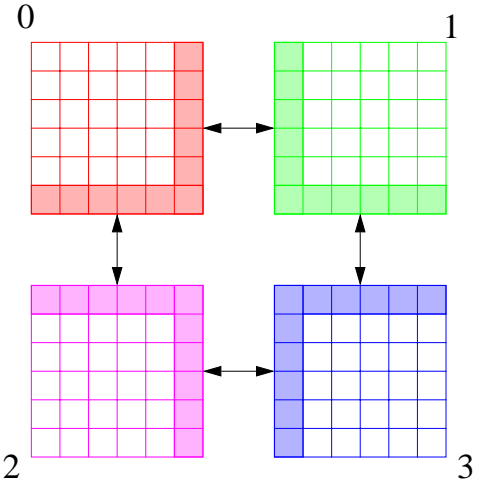
Functional decomposition

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

Decomposing a Single Grid or Array



Boundary exchange



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-2 finalized June 1997 (planned)

Current status

- Public domain versions available from ANL/MSU (MPICH), OSC (LAM), University of Edinburgh (CHIMP), MSU (Unify), etc.
- Proprietary versions available from Cray, Fujitsu, HP, IBM, Intel, NEC, SGI, TMC, etc.

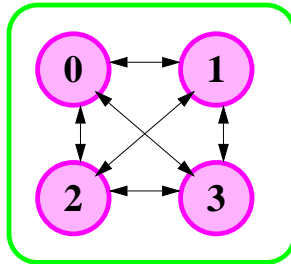
Other Libraries

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.

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” — 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()
}
```

“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 Fortran 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 Fortran differences (II)

MPI objects

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

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

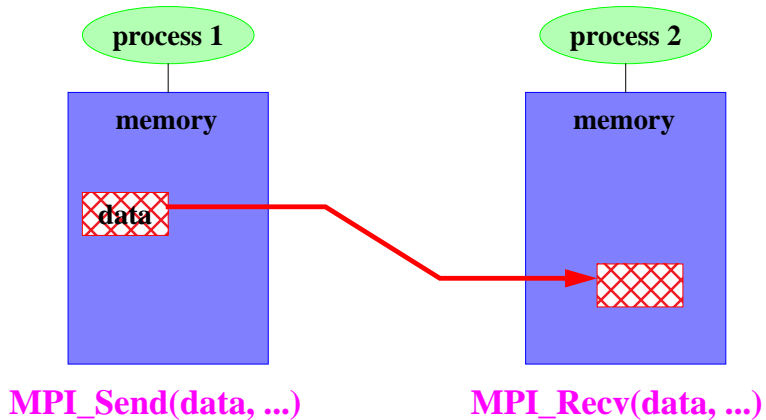
These are typedefs, so that declarations look like:

```
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



Point-to-point Example

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

A:

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

B:

```
#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

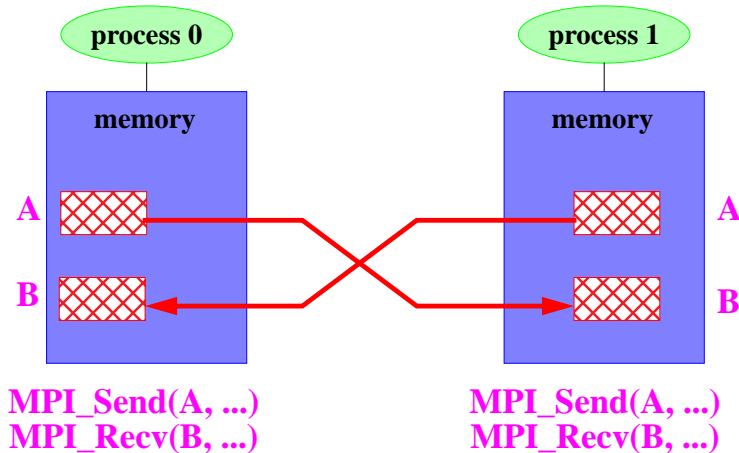
Unless there is a good reason to do so, do not use wildcards

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**.

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 if your program deadlocks. Do not blame the vendor (unless there is a true bug in the implementation, of course.)

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 “simultaneously”

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.
- 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.

```
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

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 messages
- Progress happens
- Fairness not guaranteed

MPI_Send provides a great deal of flexibility for implementations

Other MPI point-to-point routines

MPI provides 4 communication codes, with slightly different semantics

- Standard MPI_Send/MPI_Isend
- Synchronous MPI_Ssend/MPI_Issend
- Ready MPI_Rsend/MPI_Irsend
- Buffered MPI_Bsend/MPI_Ibsend

Recommendation: **Synchronous/Ready/Buffered are a waste of time. They rarely help performance, often hurt, and can be difficult to use.**

MPI provides “persistent communication:

- MPI_Send_init
- MPI_Ssend_init
- MPI_Rsend_init
- MPI_Bsend_init

These are even more of a waste of time.

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:

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

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

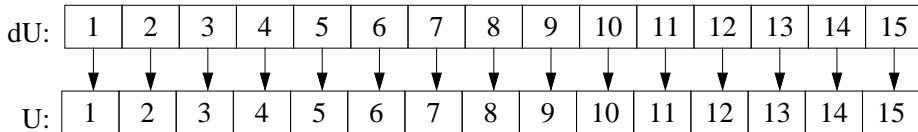
c update U
do i = 2, N-1
    U(i) = U(i) + dU(i)
end do
```

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

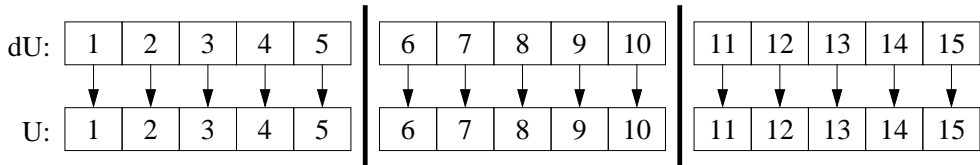
Model Problem: Data Distribution

Focus attention on $\mathbf{U} = \mathbf{U} + d\mathbf{U}$ ($N = 15$)

Layout of \mathbf{U} and $d\mathbf{U}$ in serial memory:



Possible layout of \mathbf{U} and $d\mathbf{U}$ in distributed memory (3 processors):

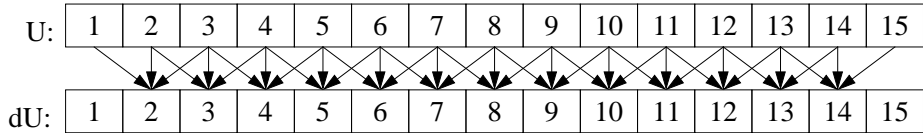


- Computation can be done entirely in parallel.
- \mathbf{U} and $d\mathbf{U}$ must be “aligned”
- \mathbf{U} and $d\mathbf{U}$ must be distributed evenly

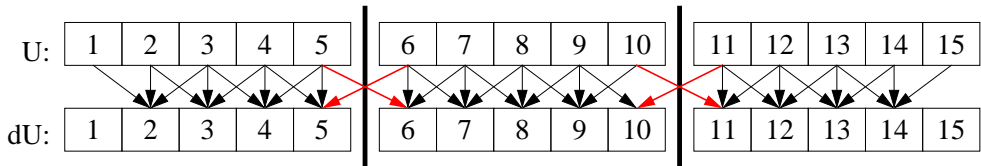
Model Problem: Communication

Focus attention on **computation of dU** (3-point stencil)

In serial memory:



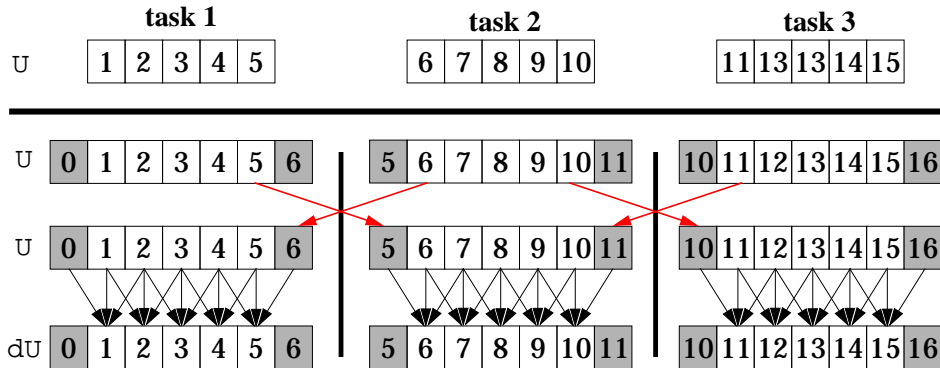
In distributed memory:



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

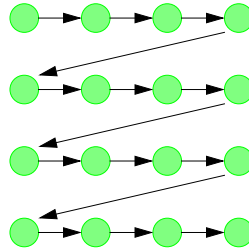
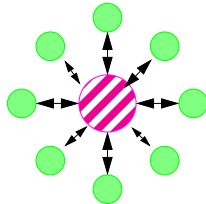
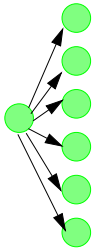
```
c calculate dU
  do i = 1, MyN
    dU(i) = (sigma * dt / h) * ( U(i+1) - 2*U(i)+ U(i-1))
  end do

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

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

```
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 sends to a subset of a communicator

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**
- ...

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: FFT (usually)

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 of **MPI_Bcast**:

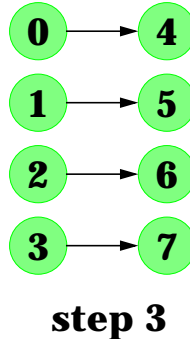
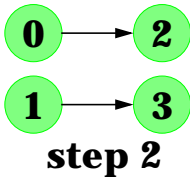
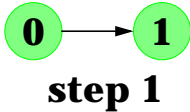
```
if (me == root) {
    for (i = 0; i < N; i++) {
        if (i != me) MPI_Send(buf, ..., dest=i, ...);
    }
} else {
    MPI_Recv(buf, ..., src=i, ...);
}
```

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
 - In your dreams

But:

- Can be difficult to understand
- Can hurt performance if not careful

User-defined datatypes: Contiguous

New datatype: 5 contiguous integers

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

- **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

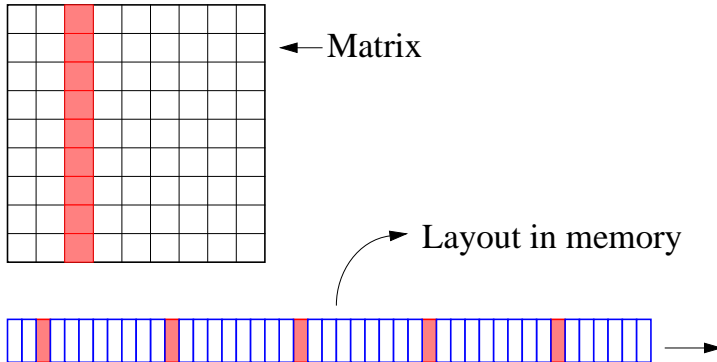
Contiguous datatype example

```
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

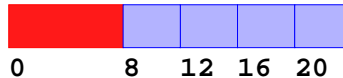
stride = 5

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[]={0, 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?

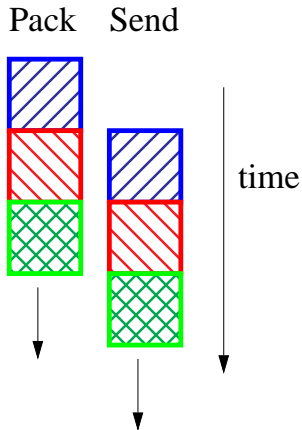
Complex datatypes can kill performance

- 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

The theoretical performance win

Consider sending a strided datatype.

Pipeline



Pack a block of data
Send it
While sending, pack next
etc.

- Nice idea
- No current implementations do this

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

More information

Books

- **Using MPI**, by William Gropp, Ewing Lusk, and Anthony Skjellum, published by MIT Press ISBN 0-262-57104-8
- **MPI: The Complete Reference**, by Snir, Otto, Huss-Lederman, Walker, Dongarra. The MIT Press
- **Parallel Programming with MPI**, by Peter Pacheco. Morgan Kaufman Publishers Inc..

The standard

- <http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html>

On-line Tutorials

- <http://www.mcs.anl.gov/mpi/tutorial/>

WWW

- <http://www.mcs.anl.gov/mpi/index.html>