Parallel Computing

Programming with MPI

Kevin Dunseath

Computational Science and Applications

2017–2018

- Books:
  - ”Parallel Programming with MPI”, Peter Pacheco, Morgan Kaufmann Publishers (October 1996)
  - ”An Introduction to Parallel Programming”, Peter Pacheco, Morgan Kaufmann Publishers (January 2011) *(covers MPI and OpenMP)*
- Websites:
  - http://www.open-mpi.org/
  - http://www.idris.fr/formations/mpi/

MPI

Message Passing Interface:

> a standard specifying the API (Application Programming Interface) for a library of routines implementing communications between processors

Available in C, C++, Fortran, Python, . . . ,
and on many different types of machine

Programs written using MPI are portable from one type of machine to another

MPI uses a distributed memory programming model, but exists on both distributed memory and shared memory systems

Parallelism is the responsibility of the programmer, who must

- identify the parts of the code that can be run in parallel
- decide how to distribute data over the available processors
- insert calls to the MPI library to share data amongst processors

Blocks of data are sent from one processor to one or more processors as *messages*

The programmer must form the message, address it by specifying the sender and the receiver, and call the appropriate communication routine

Communication environment

Communicator

An object defining a set of processes which may communicate with each other
The set of all processes associated with the current program is denoted by `MPI_COMM_WORLD`

Rank

For a particular communicator, each process is identified by a integer assigned by the system
The rank is contiguous and starts from 0

The rank is used to specify the source and destination of messages to be sent

Messages

In MPI, a message consists of the data to be transferred plus an *envelope*, containing

- the rank of the process receiving the message
- the rank of the process sending the message
- a *tag*, which may be used to distinguish messages arriving from the same source but requiring different actions
- a communicator specifying the group of processes that can communicate with each other

The data to be sent is stored in a contiguous array specified by the address of the first element, the number of elements to be sent, and their type (integers, real numbers, . . .)

Basic syntax

- Header file: must be included in all programs or routines that make calls to the MPI library

C	Fortran
`#include "mpi.h"`	`include 'mpif.h'`

- Calling MPI routines in C:

```
err = MPI_Xxxxx(.....)

err = MPI_Send(&buf, len, type, dest, tag, comm)
```

- Calling MPI routines in Fortran:

```
call MPI_XXXXX(....., ierr)

call MPI_SEND(buf, len, type, dest, tag, comm, ierr)
```

Routine names are case-sensitive in C but not in Fortran

Interacting with the environment

`MPI_Init` initializes the MPI execution environment, and must be called once in any MPI program, before any other calls to MPI routines

```
MPI_Init(&argc,&argv)                call MPI_INIT(ierr)
```

`MPI_Finalize` terminates the MPI execution environment, and must be the last call to an MPI routine in the program

```
MPI_Finalize()                  call MPI_FINALIZE(ierr)
```

`MPI_Abort` terminates all MPI processes associated with the communicator. Usually used when an error has occurred

```
MPI_Abort(comm,errorcode)
call MPI_ABORT(comm,errorcode,ierr)
```

Interacting with the environment (2)

`MPI_Comm_size` returns the number of processes associated with a given communicator. If `comm=MPI_COMM_WORLD`, this is the number of processes being used by the application

```
MPI_Comm_size(comm,&size)
call MPI_COMM_SIZE(comm,size,ierr)
```

`MPI_Comm_rank` returns the rank of the current process in the given communicator

```
MPI_Comm_rank(comm,&rank)
call MPI_COMM_RANK(comm,rank,ierr)
```

Interacting with the environment (3)

`MPI_Get_processor_name` returns the processor name and the length of the name. The buffer receiving the name must be at least `MPI_MAX_PROCESSOR_NAME` in size

```
MPI_Get_processor_name(&name,&lenname)
call MPI_GET_PROCESSOR_NAME(name,lenname,ierr)
```

`MPI_Wtime` is a function returning a double precision value representing the elapsed wall-clock time in seconds for the calling process, measured from some point in the past

```
double t0 = MPI_Wtime()
```

```
double precision ::  t0
t0 = MPI_WTIME()
```

A first MPI program

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

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

  int len_name,nprocs,rank;

  char *my_name = (char *) malloc(MPI_MAX_PROCESSOR_NAME
                                  *sizeof(char));

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  MPI_Get_processor_name(my_name,&len_name);

  printf("I am processor %d out of %d, my name is %s \n",
                                   rank,nprocs,my_name);
  MPI_Finalize();
}
```

A first MPI program

```
program hello

use mpi
implicit none
!include 'mpif.h'

integer :: ierr, len_name, nprocs, rank
character(len=MPI_MAX_PROCESSOR_NAME) :: my_name

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_GET_PROCESSOR_NAME(my_name,len_name,ierr)

write (6,'("I am processor ",i2," out of ",i2,", my name is ",
   rank, nprocs, my_name

call MPI_FINALIZE(ierr)

stop
end program hello
```

```
mac-kevin> mpicc hello.c
mac-kevin> mpirun -np 2 ./a.out

I am processor 0 out of 2, my name is mac-kevin.local
I am processor 1 out of 2, my name is mac-kevin.local
```

```
mac-kevin> mpif90 hello.f90
mac-kevin> mpirun -np 2 ./a.out

I am processor 0 out of 2, my name is mac-kevin.local
I am processor 1 out of 2, my name is mac-kevin.local
```

MPI data types

MPI defines its own set of data types to ensure portability between systems:

C: MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MP_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE, MPI_PACKED

Fortran: MPI_CHARACTER, MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX, MPI_DOUBLE_COMPLEX, MPI_LOGICAL, MPI_BYTE, MPI_PACKED

Users can also define their own data types

MPI predefined constants

As well as data types, MPI also defines a large number of constants (in `mpi.h`). For example:

`MPI_COMM_WORLD`: default communicator containing all processes

`MPI_MAX_PROCESSOR_NAME`: minimum size for buffer containing the result of `MPI_Get_processor_name`

`MPI_SUCCESS`: result returned by MPI routines on successful completion

`MPI_ANY_SOURCE`: wildcard indicating that the message to be received can come from any source

`MPI_ANY_TAG` wildcard indicating that the label or tag on the message to be received can have any value

Point-to-Point communications

Point-to-point communication involves passing a message between two processors: one processor sends the message, the other processor receives the message

Different types of point-to-point routines:

- Blocking send / blocking receive
- Non-blocking send / non-blocking receive
- Buffered send (for very large messages)
- Synchronous send

When a call to an MPI send or receive function is made, a request for communication is said to have been ”posted”

Point-to-Point communications (2)

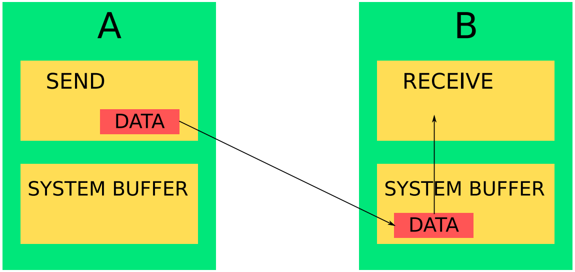
Send operations are not usually perfectly synchronised with their matching receive operations:

- the send may occur before the receive is ready - where is the message stored while the receive is pending?
- multiple sends may arrive at the same processor, which can only receive messages one at a time - where are the other messages stored while waiting their turn?

The standard does not specify what to do in these cases - the answer is implementation dependent

Point-to-Point communications: buffering

- Typically, the message is held in a system buffer managed by the MPI Library and not visible to the programmer



- The space of user variables and data is called the application buffer

Point-to-Point communications: buffering (2)

- The system buffer is of finite size and can easily be filled up
- Exceeding the buffer size will cause a program to give erroneous results, or to crash
- To send very large messages, the user must
    - allocate a sufficiently large buffer
    - tell the system to use this buffer to transfer the message
    - call *buffered* versions of the send/receive operations

Point-to-Point communications: blocking

Blocking

- a blocking send will only return control to the program when it is sure it can modify the application buffer without changing the data in the message. There is no guarantee that the message has been received.
- a blocking send can be synchronous (hand-shaking with receive processor to confirm send) or asynchronous (message is stored in system buffer while waiting to be received)
- a blocking receive only returns control to the program after the data has arrived safely

Point-to-Point communications: blocking (2)

Non-blocking

- non-blocking send and receive return control to the program almost immediately, without waiting for the communication to be completed
- non-blocking operations ask the MPI library to perform the communication whenever it can. It is unsafe to modify the application buffer until it is certain that the data has been transferred.

MPI provides routines to check if a communication has been completed successfully

MPI guarantees that messages will not overtake each other, but will arrive in the order they were sent

Point-to-Point communications: routines

Blocking send

```
MPI_Send(buffer,len,type,dest,tag,comm)
```

Blocking receive

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

Non-blocking send

```
MPI_Isend(buffer,len,type,dest,tag,comm,request)
```

Non-blocking receive

```
MPI_Irecv(buffer,len,type,source,tag,comm,
                                    request)
```

Point-to-Point communications: routines (2)

buffer: variable or array containing data to be sent or received. In C, this is passed by reference and so must be a pointer or the address of a variable (`&var`)

len: the number of data items to be transferred

type: the type of data to be transferred (`MPI_INT`, `MPI_FLOAT, ...`)

source/destination: the rank of the sending or receiving process. `MPI_ANY_SOURCE` may be used in a receive operation to listen for messages from any other processor

tag: a non-negative integer defined by the programmer to label a message. The tags on a send-receive pair must be identical. `MPI_ANY_TAG` may be used to receive a message regardless of its tag

Point-to-Point communications: routines (3)

communicator: defines the set of processes involved in the communication, usually `MPI_COMM_WORLD`

status: in a receive operation, this contains the source of the message `status.MPI_SOURCE` and its tag `status.MPI_TAG`. It is a structure of type `MPI_Status`

In Fortran, status is an array of size `MPI_STATUS_SIZE`, and the source and tag are given by `status(MPI_SOURCE)` and `status(MPI_TAG)`

request: used by non-blocking routines to label a particular communication. It can be used later on to determine in the communication has completed

Point-to-Point communications: blocking mode

Standard or blocking mode: `Send, Recv` :

- will not return control to user program until the message buffers used in the argument list of the function call can be safely modified by subsequent statements in the program
- `Send`: the data in the message buffer has been sent or copied into the system buffer
- `Recv`: the data has ben copied into the message buffer and can be used

```
if (rank == source) then
   call MPI_SEND (a,len,MPI_DOUBLE_PRECISON,dest,tag,  &
                 comm,ierr)
!  The array a can be safely updated here.
   ...
else if (rank == dest) then
   call MPI_RECV (b,len,MPI_DOUBLE_PRECISON,source,tag, &
                 comm,status,ierr)
!  The array b can be safely used here.
   ...
end if
```

```
double *a, *b;
int ierr;

...

if (rank == source) {
   ierr = MPI_Send(a,len,MPI_DOUBLE,dest,tag,MPI_COMM_WORLD);
// The array a can be safely updated here.
   ...
} else if (rank == dest) {
   ierr = MPI_Recv(b,len,MPI_DOUBLE,source,tag,
                   MPI_COMM_WORLD,&status);
// The array b can be safely used here.
   ...
}
```

Point-to-Point communications: non-blocking mode

Non-blocking mode: `Isend, Irecv` :

- starts or ”posts” a send/receive operation
- the user program must explicitly complete the communication later on $\Rightarrow$ requires at least two function calls
- `Isend`: system starts to copy meassge to system buffer or to its destination
- `IRecv`: system is informed that it can start to copy data in to message buffer
- Completion of communication is signaled by the `request` variable in the function call

Point-to-Point communications: synchronous mode

Synchronous send `Ssend`:

- does not complete (i.e. return control to user program) until a matching `Recv` has begun to receive the message
- completion indicates that send buffer can be reused and that the receiver has reached a certain point in the execution (it has started executing the matching receive)
- does not require system buffering

```
if (rank == 0) {
  MPI_Ssend(&x,1,MPI_DOUBLE,dest,0,MPI_COMM_WORLD);
} else if (rank == dest) {
  MPI_Recv(&x,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD,&status);
}
```

Point-to-Point communications: buffered mode

Buffered send `Bsend`:

- `Bsend` is local: completion does not depend on the existence of a matching `Recv`
- if there is no matching `Recv` posted, message is copied into a buffer or zone of memory that is explicitly created (allocated) by the user
- `MPI_Buffer_attach` and `MPI_Buffer_detach` are used to associate the buffer with `Bsend`
- the buffer must be sufficiently large to store the message
- only one buffer can be attached at any time

A second MPI program

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

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

  int dest,i,nprocs,rank;
  double x;
  MPI_Status status;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
```

A second MPI program (2)

```
  if (rank != 0) {
    dest = 0;
    srand((rank+1)*time(NULL));
    x = (double) rank * rand()/RAND_MAX;
    printf("Processsor %d has x = %f \n", rank, x);
    MPI_Send(&x,1,MPI_DOUBLE,dest,0,MPI_COMM_WORLD);
  } else {
    for (i=1; i < nprocs; i++) {
      MPI_Recv(&x,1,MPI_DOUBLE,MPI_ANY_SOURCE,0,
                                 MPI_COMM_WORLD,&status);
      printf("Processor 0 received %f from processor %d \n",
                  x, status.MPI_SOURCE);
    }
  }
  MPI_Finalize();
}
```

```
mac-kevin> mpicc communicate.c

mac-kevin> mpirun -np 4 ./a.out

Processsor 1 has x = 0.636852
Processsor 3 has x = 0.821065
Processsor 2 has x = 1.910556
Processor 0 received 0.636852 from processor 1
Processor 0 received 1.910556 from processor 2
Processor 0 received 0.821065 from processor 3

mac-kevin>
```

Deadlocks

```
int a[10], b[10], rank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank == 0) {
   MPI_Send(a,10,MPI_INT,1,1,MPI_COMM_WORLD);
   MPI_Send(b,10,MPI_INT,1,2,MPI_COMM_WORLD);
} else if (rank == 1) {
   MPI_Recv(b,10,MPI_INT,0,2,MPI_COMM_WORLD);
   MPI_Recv(a,10,MPI_INT,0,1,MPI_COMM_WORLD);
}
```

- If `MPI_Send` uses buffering, this will work if there is sufficient buffer space
- If `MPI_Send` does not use buffering (blocking until matching receive is posted), neither process can proceed
- Matching the order of the send and receive operations corrects this problem

Deadlocks (2)

In an exchange, trying to receive data before sending

Trying to receive data when there has been no send

```
if (rank == A) {
   MPI_Recv(&x,lenx,MPI_INT,B,tag1,MPI_COMM_WORLD,&status);
   MPI_Send(&y,leny,MPI_INT,B,tag2,MPI_COMM_WORLD);
} else {
   MPI_Recv(&y,leny,MPI_INT,A,tag2,MPI_COMM_WORLD,&status);
   MPI_Send(&x,lenx,MPI_INT,B,tag1,MPI_COMM_WORLD);
}
```

Since `MPI_Recv` is blocking, this code will certainly hang, since `A` and `B` will wait for messages that will not be sent

The program will also hang or crash if there are errors in the parameters, for example a mismatch in the tag, or sending to the wrong destination

MPI_Sendrecv

`MPI_Sendrecv` combines seperate send and receive operations in one function call

A message can be sent to one process (destination) and a different message received from another process (source). The source and destination processes may be the same

The sent and received messages must be stored in different, non-overlapping arrays

```
int err =  MPI_Sendrecv(sbuffer, slen, stype, dest, stag,
                        rbuffer, rlen, rtype, src,  rtag,
                        comm, &status)
```

The send and receive operations are blocking, and can help avoid deadlocks

Message probing

In MPI_Recv, the message length must be known beforehand, in order to allocate enough space in the receive buffer

The length/dimensions of an array can be sent in an earlier message, via MPI_Send:

```
if (rank==0) {
// Set up message in data array
    MPI_Send(&len,1,MPI_INT,1,0,MPI_COMM_WORLD);
    MPI_Send(data,len,MPI_DOUBLE,1,0,MPI_COMM_WORLD);
} else if (rank==1) {
    int len;
    MPI_Status status;
    MPI_Recv(&len,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
    double* data  = new double [len];
    MPI_Recv(data,len,MPI_DOUBLE,0,0,MPI_COMM_WORLD,&status);
}
```

Message probing (2)

`MPI_Probe` (blocking) or `MPI_Iprobe` (non-blocking) allow the user to check an incoming message without actually receiving it

```
int err =  MPI_Probe(src, tag, comm, &status)
int err = MPI_Iprobe(src, tag, comm, &flag, &status)

int err = MPI_Get_count(&status, type, &count)
```

`MPI_Get_count` returns the number of values of the specified data type that have been received

Message probing (3)

Example, determining the length of the message in order to allocate the ressources necessary to receive it

```
if (rank==0) {
// Set up message in data array
    MPI_Send(data,len,MPI_DOUBLE,1,0,MPI_COMM_WORLD);
} else if (rank==1) {
    int len;
    MPI_Status status;
    MPI_Probe(0,0,MPI_COMM_WORLD,&status);
    MPI_Get_count(&status,MPI_DOUBLE,&len);
    double* data  = new double [len];
    MPI_Recv(data,len,MPI_DOUBLE,0,0,MPI_COMM_WORLD,&status);
}
```

`MPI_Probe` blocks until a matching message (source, tag) has been found.

Collective communications

Collective communications involve all processors associated with a given communicator (`MPI_COMM_WORLD`)

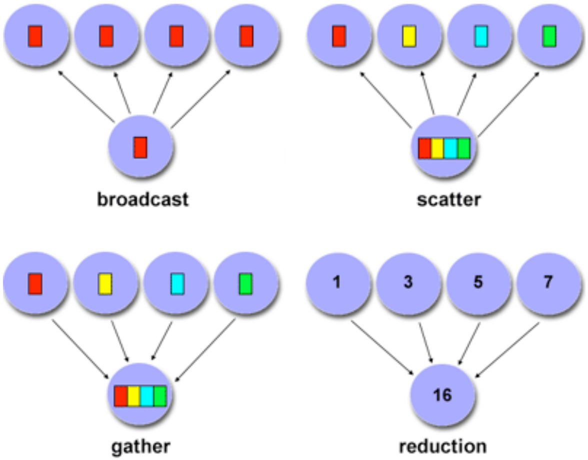
Different types of collective communication routines:

synchronisation: forcing processors to wait until all members of the communicator group have reached a given point in the execution

data sharing: distributing data from one processor to all the others, or gathering data from all processors onto one processor

computation: one processor collects data from all the others, and performs a simple operation (addition, multiplication) on this data. Also known as *reduction*

Collective communications are blocking



https://computing.llnl.gov/tutorials/mpi/images/collective_comm.gif

Collective communications routines

Barrier

Creates a barrier, at which each task waits until all processors arrive at the same call

```
MPI_Barrier(comm)
```

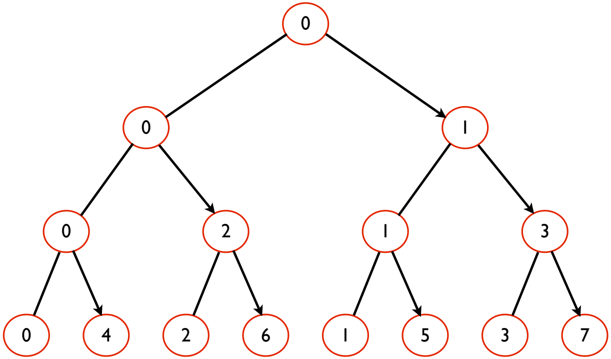
Broadcast

Sends a message from the processor with rank ”root” to all other processes in the same communicator

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

Collective communications routines (2)

Example: Broadcast using a tree structure



`MPI_Bcast` may be optimized for topology of machine
It is more efficient then several calls to `MPI_Send` and `MPI_recv`

Collective communications routines (3)

Scatter

Distributes different messages (data) of identical lengths from a single source to all other processes in the same communicator

```
MPI_Scatter(&sendbuf,sendlen,sendtype,&recvbuf,
                     recvlen,recvtype,root,comm)
```

- Data stored in `sendbuf` are split into segments, each of which has `sendlen` elements of type `sendtype`. The first segment is sent to process 0, the second to process 1, . . .
- `recvlen` and `recvtype` specify the number and type of elements received by each process. They will usually be the same as `sendlen` and `sendtype`

```
int nprocs,rank,root;
int rbuff[100];
int* sbuff;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&nprocs);

if (rank==root)
   sbuff = (int *)malloc(nprocs*100*sizeof(int));

...

MPI_Scatter(sbuff, 100, MPI_INT, rbuff, 100, MPI_INT,
            root, MPI_COMM_WORLD);
```

Collective communications routines (4)

Gather

Gathers different messages (data) of identical lengths from each process in the communicator onto a single process ”root”

```
MPI_Gather(&sendbuf,sendlen,sendtype,&recvbuf,
                    recvlen,recvtype,root,comm)
```

- Data referenced by `sendbuf` on each process consists of `sendlen` elements of type `sendtype`
- `recvlen` and `recvtype` specify the number and type of elements received from each process, not the total amount of data received. They will usually be the same as `sendlen` and `sendtype`
- Data is stored in `recvbuf` on the process `root` in process order: data from processs 0 followed by data from process 1 followed by data from process 2 . . .

```
int nprocs,rank,root;
int sbuff[100];
int* rbuff;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&nprocs);

if (rank==root)
   rbuff = (int *)malloc(nprocs*100*sizeof(int));

...

MPI_Gather(sbuff, 100, MPI_INT, rbuff, 100, MPI_INT,
           root, MPI_COMM_WORLD);
```

Collective communications routines (5)

Reduce

Applies a reduction to the data on all processes, storing the result on the process with rank "root"

```
MPI_Reduce(&sendbuf,&recvbuf,len,type,op,root,
                                          comm)
```

Reduction operations `op` include

`MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_MAXLOC, MPI_MINLOC, MPI_LAND, MPI_LOR`

as well as a number of bit-wise operations

Collective communications routines (6)

AllGather

As for `MPI_Gather`, except that the result is available to all processes

```
MPI_Allgather(&sendbuf,sendlen,sendtype,
                &recvbuf,recvlen,recvtype,comm)
```

AllReduce

As for `MPI_Reduce`, except that the result is available to all processes. Equivalent to `MPI_Reduce` followed by `MPI_Bcast`

```
MPI_Allreduce(&sendbuf,&recvbuf,len,type,op,
                                        comm)
```

Collective communications routines (7)

Collective communication routines are executed by all processes:

```
int nprocs,rank;
float x;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&nprocs);

if (rank == 0) {
  printf("Enter a real number");
  scanf("%f",&x);
  printf("\n");
}

MPI_Bcast(&x,1,MPI_FLOAT,0,MPI_COMM_WORLD);
printf("Process %d has x = %f \n",rank,x);

MPI_Finalize();
```

Domain decomposition

A common situation: points on a grid, stored in a 2D matrix
Updating values at one point requires values at neighbouring points

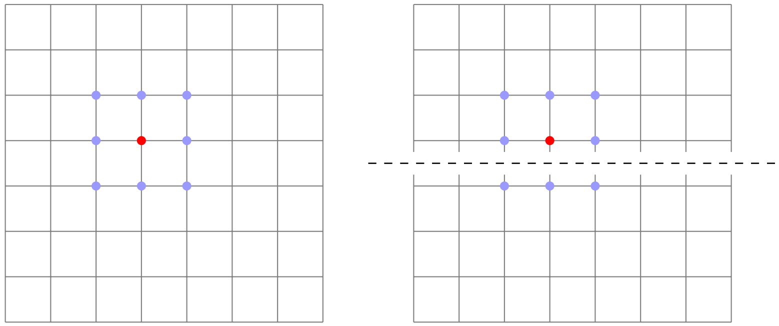
Example: finite difference approximation to second-order derivatives in a 2D system:

$$\begin{aligned}\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) f(x,y) &= \frac{1}{(\Delta x)^2}\left(f_{i+1,j} - 2f_{ij} + f_{i-1,j}\right) \\ &\quad + \frac{1}{(\Delta y)^2}\left(f_{i,j+1} - 2f_{ij} + f_{i,j-1}\right)\end{aligned}$$

with

$$\begin{aligned} f_{ij} &\equiv f(x_i, y_j) \\ x_i &= x_{\mathrm{min}} + i\Delta x \\ y_j &= y_{\mathrm{min}} + j\Delta y \end{aligned}$$

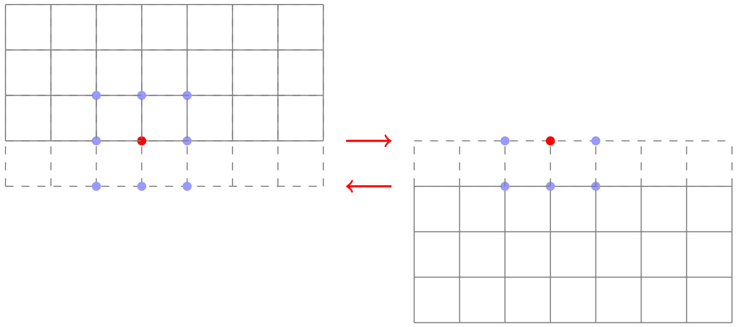
Domain decomposition (2)



No problem if all points are held by the same process, but what if some neighbouring points are held on a different process?

Domain decomposition (3)

One common solution: ghost (or halo) cells



Add an extra row (coulmn) at the start and/or end of the actual data storing the values at points held by the other processes

Update these ”ghost” cells with new values when needed

Running MPI programs

To compile a program using MPI:

```
mpicc prog.c  mpic++ prog.cpp  mpif90 prog.f90
```

To run the program with N processes:

```
mpirun -np N ./a.out
```

The machines to be used can be specified in a file:

```
mpirun -np N --hostfile hosts ./a.out
```

where each line in the file `hosts` has the form of a machine name or IP address and the number of processes to be executed on that particular machine.

The current machine (on which `mpirun` has been typed) must be in the list contained in `hosts`

Running MPI programs (2)

When running the program on N processors, N copies of the program are executed

Each process is distinguished by its rank. A process on the current machine is assigned the rank 0

Only the process with rank 0 can read data from the keyboard. If necessary, data can then be sent or broadcast to the other processes

A process receiving a message must know the length of the message. If necessary, this length must be sent first

All data are local: each process must allocate and free any arrays that may be required