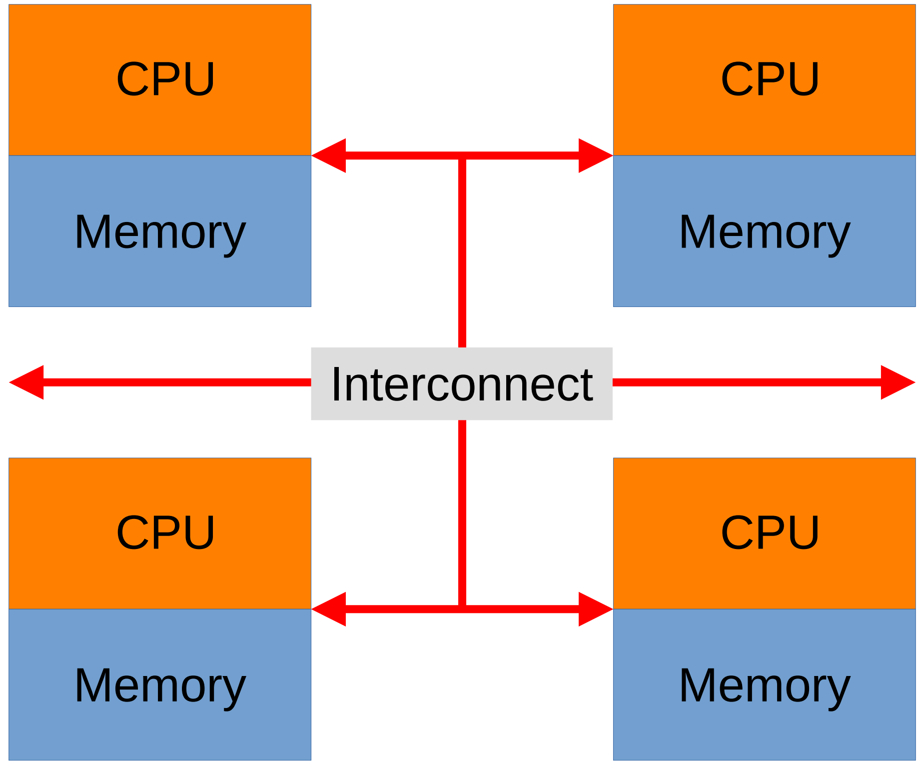
Distributed Memory Programming using MPI

Kameswararao Anupindi

Department of Mechanical Engineering,
IIT Madras
February 2024

Example of a distributed memory machine



Credit: https://www.nvidia.com/en-in/networking/infiniband-switching/

Interconnects ~ 100 Gb/s

1. InfiniBand

2. Omni-Path

About MPI (Message Passing Interface)

- MPI 4.1 standard ***https://www.mpi-forum.org/docs/***
- MPI Implementations:
  - OpenMPI, MPICH, MVAPICH, Intel MPI etc.
- MPI – not a new language, rather a library that provides functions for C/C++/FORTRAN.
- What is a process?
- Send - Recv function calls
- Point-to-point or collective communication
- Incremental parallelization not possible!
- SIMD/SPMD model

Compilation and Execution

- **mpicc** -g -Wall -o mpi_hello mpi_hello.c
- **mpif90** -g -Wall -o mpi_hello mpi_hello.f90

- mpicc/ mpif90 is a wrapper script (?)

- **mpiexec** -n <no. of processes> <executable>
- **mpiexec** -n 4 ./mpi_hello

- How do we get from mpiexec to execution of code?

Hello World - A serial program

```
/* Hello world program */
#include <stdio.h>

int main(void)
{
  printf("\n Hello, world!\n");

  return 0;
}
```

Hello World – Parallel version in C

```
#include<stdio.h>
#include<string.h>
#include<mpi.h>

int main(int argc, char** argv)
{
  int i, myid, size, tag=100;
  char message_send[50], message_recv[50];
  MPI_Status status;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  /* printf("%d", size); */
```

Hello World – Parallel version in C contd.

```
  if (myid != 0)
    {
      sprintf(message_send, " Hello from process id: %d \n ", myid);
      MPI_Send(message_send, 50, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
    }
  else
    {
      for (i = 1; i < size; i++)
        {
          MPI_Recv(message_recv, 50, MPI_CHAR, i, tag, MPI_COMM_WORLD, &status);
          printf("\n %s", message_recv);
        }

      sprintf(message_send, " Hello from process id: %d \n ", myid);
      printf("\n %s", message_send);

    }

  MPI_Finalize();

  return 0;
}
```

Hello World – Parallel version in FORTRAN

```
program main
  implicit none
  include "mpif.h"
  integer :: i, myid, size, mpierror, tag, status(MPI_STATUS_SIZE)
  character(len=50) :: message_send, message_recv

  call MPI_INIT(mpierror)

  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, mpierror)
  call MPI_COMM_RANK(MPI_COMM_WORLD, myid, mpierror)

  tag = 100

  if (myid /= 0) then
     write(message_send, *), 'Hello from process id:', myid
     call MPI_SEND(message_send, 50, MPI_CHARACTER, 0, tag, MPI_COMM_WORLD, mpierror)
  else
     do i = 1, size-1
        call MPI_RECV(message_recv, 50, MPI_CHARACTER, i, tag, MPI_COMM_WORLD, status, mpierror)
        write(*, *), message_recv
     end do
     write(message_send, *), 'Hello from process id:', myid
     write(*, *), message_send
  end if

  call MPI_FINALIZE(mpierror)

end program main
```

Initialization and Finalization

```c
int MPI_Init(int * argc_p, char *** argv_p)

int MPI_Init( NULL, NULL)



int MPI_Finalize(void)
```

```fortran
subroutine MPI_INIT(mpierror)
    integer, intent(out) :: mpierror



subroutine MPI_FINALIZE(mpierror)
    integer, intent(out) :: mpierror
```

Communication

```
int MPI_Comm_size(
                MPI_Comm  comm,
                int* comm_size_p);

int MPI_Comm_rank(
                MPI_Comm  comm,
                int* myid_p);
```

```
subroutine MPI_COMM_SIZE(
                          MPI_Comm comm,
                        integer size,
                        integer mpierror)

subroutine MPI_COMM_RANK(
                        MPI_Comm comm,
                        integer myid,
                        integer mpierror)
```

Communication – C functions

```
int MPI_Send(
        void*           message_buf_p,
        int             message_size,
        MPI_Datatype    message_type,
        int             dest,
        int             tag,
        MPI_Comm        communicator);


int MPI_Recv(
        void*           message_buf_p,
        int             buf_size,
        MPI_Datatype    buf_type,
        int             source,
        int             tag,
        MPI_Comm        communicator,
        MPI_Status*         status_p);
```

Communication – FORTRAN routines

```
int MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)
        type(*), dimension(:), intent(in) :: buf
        integer, intent(in) :: count
        type(mpi_datatype), intent(in) :: datatype
        integer, intent(in) :: dest, tag
        type(mpi_comm), intent(in) :: comm
        integer, optional, intent(out) :: ierror


int MPI_RECV(buf, count, dataype, source, tag, comm, status,
             ierror)
        type(*), dimension(:) :: buf
        integer, intent(in) :: count
        type(mpi_datatype), intent(in) :: datatype
        integer, intent(in) :: source, tag
        type(mpi_comm), intent(in) :: comm
        type(mpi_status) :: status
        integer, optional, intent(out) :: ierror
```

Message Matching

Process a:

```
MPI_Send(send_buf_p, send_buf_size, send_type,
         dest, send_tag, send_comm);
```

Process b:

```
MPI_Recv(recv_buf_p, recv_buf_size, recv_type,
         src, recv_tag, recv_comm, &status)
```

```
recv_comm   ==   send_comm
recv_tag    ==   send_tag
dest        ==   b
src         ==   a
```

Message Matching contd.

Must specify compatible buffers:

```
send_buf_p       --  recv_buf_p
send_buf_size    --  recv_buf_size
send_type        --  recv_type
```

Most of the time, it is sufficient to have:

```
recv_type        ==  send_type
recv_buf_size    >=  send_buf_size
```

Then the message sent by Process a can be successfully received by Process b.

The case of unknown message order

```
for ( i = 1; i < comm_size ; i++)
{
    MPI_Recv(recv_buf_p, recv_buf_size, recv_type,
    MPI_ANY_SOURCE, recv_tag, comm, status_p);

    function(recv_buf_p);

}
```

The case of multiple messages from a sender

```
for ( i = 1; i < n_messages ; i++)
{
    MPI_Recv(recv_buf_p, recv_buf_size, recv_type,
    source, MPI_ANY_TAG, comm, status_p);

    function(recv_buf_p);

}
```

- MPI_ANY_SOURCE and MPI_ANY_TAG are “wildcard” arguments
- Only a receiver can use a wildcard argument.
- Senders must specify a **process rank** and a non-negative **tag**
- No wildcard argument for communicator argument.

The status argument in MPI_Recv

A receiver can receive a message without knowing

1) The amount of data in a message
2) The sender of the message
3) The tag of the message

How can receiver find out these values?

The status argument in MPI_Recv

```
MPI_Status
{
MPI_SOURCE
MPI_TAG
MPI_ERROR
   …..
}


status.MPI_SOURCE,
status.MPI_TAG
MPI_Get_count(&status, recv_type, &count);

MPI_STATUS_IGNORE
```

Functioning of MPI_Send and MPI_Recv

Sending process will assemble the message "envelope"

1. The sending process will copy the message from send_buf_p to system buffer and attaches the header information containing: sender, receiver, tag, communicator, size

2. Send the message via network switch from the sending process to the receiving process

3. At the receiving end, copy the message from the system buffer to recv_buf_p prescribed by MPI_Recv.

MPI_Send and MPI_Recv are *blocking* and *asynchronous!*

MPI_Send and MPI_Recv behavior

MPI_Send could (1) buffer (2) block

MPI_Recv always blocks

- MPI run time system ensures that the messages be non-overtaking when **two processes** are involved

- However, MPI can't impose performance on a network! when **more than two processes** are involved.

Behavior of Receive Order

```
if (my rank == 0)
{
MPI Send (sendbuf1, count, MPI_INT, 2, tag, comm);
MPI Send (sendbuf2, count, MPI_INT, 1, tag, comm);
}
else if (my rank == 1)
{
MPI Recv (recvbuf1, count, MPI_INT, 0, tag, comm, &status);
MPI Send (recvbuf1, count, MPI_INT, 2, tag, comm);
}
else if (my rank == 2)
{
MPI Recv (recvbuf1, count, MPI_INT, MPI_ANY_SOURCE,
              tag, comm, &status);
MPI Recv (recvbuf2, count, MPI_INT, MPI_ANY_SOURCE,
              tag, comm,&status);
}
```

A few potential pitfalls

Process a	Process b
X	MPI_Recv
MPI_Send	X

Other examples:

1. Non-matching tags

2. Rank of the destination process is the same as the rank of the source process