MANCHESTER 1824
The University of Manchester

COMP35112 Chip Multiprocessors

Message Passing Interface

Pierre Olivier

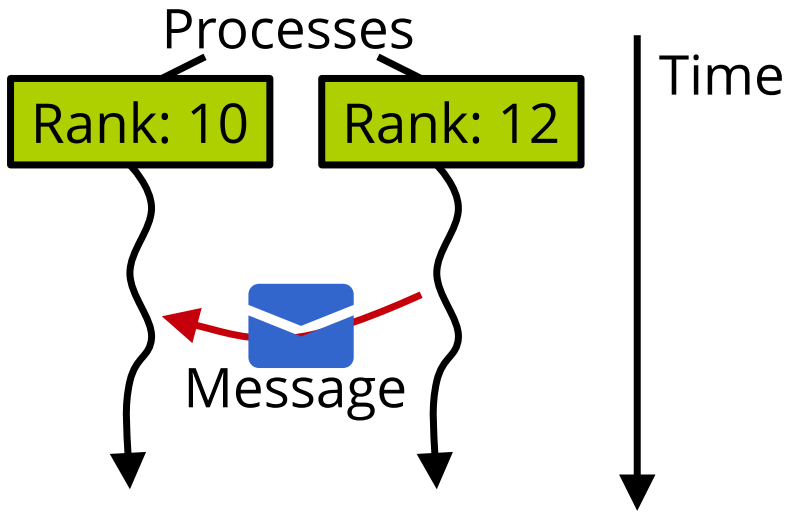
Threads vs. OpenMP and MPI

- Multithreading as we saw it is not a panacea
  1. Shared-memory limits us to the machine boundary
  2. Lots of effort to divide parallel work
- To address these problems, two standard parallel programming frameworks:
  - **MPI: Message Passing Interface** (addresses 2., this lecture)
  - **OpenMP: Open MultiProcessing** (addresses 1., seen in a future lecture)

MPI

- **Message Passing Interface**:
  - A **standard**: set of basic functions used to create portable parallel applications
- Used to program a wide range of parallel architectures:
  - Shared memory machines, **supercomputers**, **clusters**
- Relies on **message passing** for communication
- **It is the most widely used API for parallel programming**
- Downside: an existing application needs to be redesigned with MPI in mind to be ported

Messages, Processes, Communicators



- Messages: rank of sending/receiving processes + tag for type
- Process groups + communicators (e.g. `MPI_COMM_WORLD`) to restrict communications

MPI Hello World

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    MPI_Init(NULL, NULL); /* Initialise MPI */

    /* Retrieve the world size i.e. the total number of processes */
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    /* Retrieve the curent process rank (id) */
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Retreive the machine name */
    char machine_name[MPI_MAX_PROCESSOR_NAME];
    int nlen;
    MPI_Get_processor_name(machine_name, &nlen);

    printf("Hello world from process of rank %d, out of %d processes on machine"
           " %s\n", my_rank, world_size, machine_name);

    /* Exit */
    MPI_Finalize();
}
```

13-mpi/mpi-hello.c

MPI Hello World

- To install the MPICH implementation on Ubuntu/Debian:

```
sudo apt install mpich
```

- To compile the aforementioned example, use the provided gcc wrapper:

```
mpicc listing1.c -o listing1
```

- To run:

```
mpirun listing1  # Default: will run a single process
mpirun -n 2 listing1 # Manually specify the number of processes to run
```

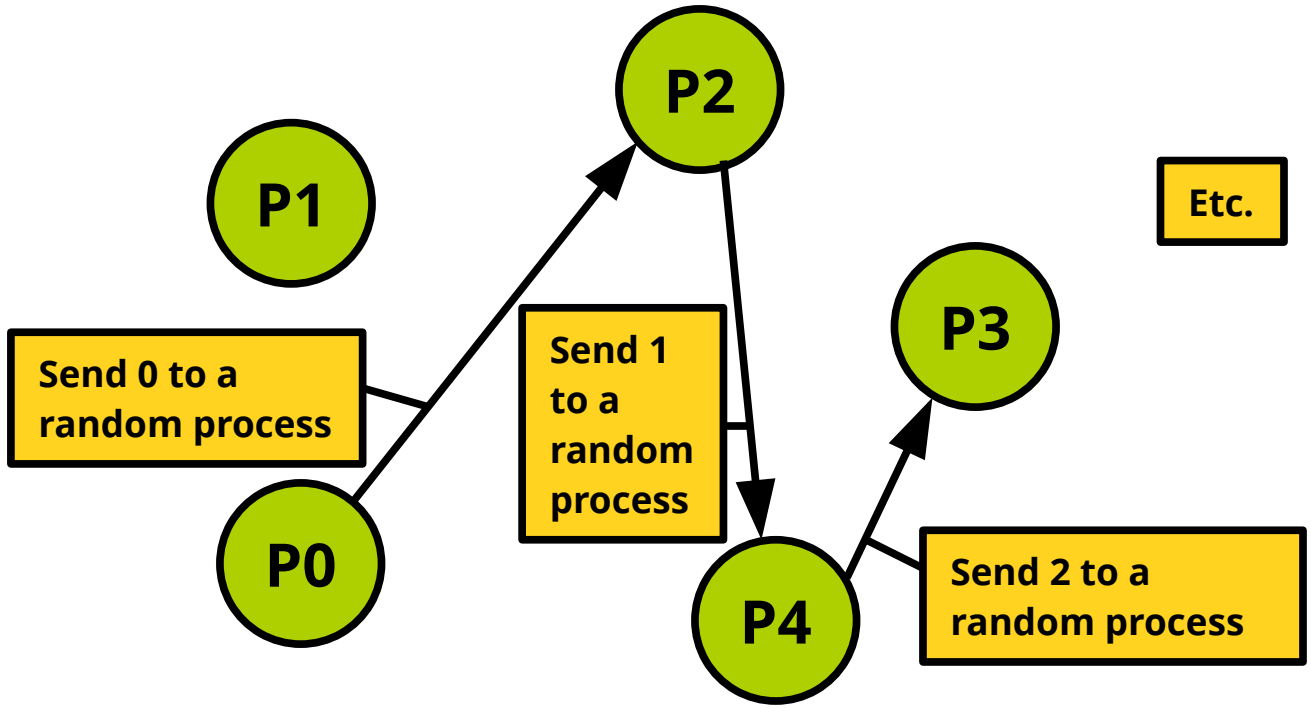
Blocking `send` and `recv`

```
MPI_Send(void* data, int cnt, MPI_Datatype t, int dst, int tag, MPI_Comm com);
MPI_Recv(void* data, int cnt, MPI_Datatype t, int src, int tag, MPI_Comm com, MPI_Status* st);
```

- `data`: pointer to the data to send/receive
- `cnt`: number of items to send/receive
- `t`: type of data - e.g. `MPI_INT`, `MPI_DOUBLE`, etc.
- `dst`: rank of the receiving process
- `src`: rank of the sending process
- `tag`: integer, in `recv` can be `MPI_ANY_TAG`
- `com`: the communicator, e.g. `MPI_COMM_WORLD`
- `st`: status information

MPI `send`/`recv` Example

- A counter "bounces" randomly between processes, gets incremented at each hop



MPI `send`/`recv` Example

```c
#define BOUNCE_LIMIT    20
int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);
    int my_rank, world_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    if (world_size < 2) {
        fprintf(stderr, "World need to be >= 2 processes\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    if(my_rank == 0) {
        int payload = 0;
        int target_rank = 0;
        while(target_rank == my_rank)
            target_rank = rand()%world_size;

        printf("[%d] sent int payload %d to %d\n", my_rank, payload, target_rank);
        MPI_Send(&payload, 1, MPI_INT, target_rank, 0, MPI_COMM_WORLD);
    }
```

13-mpi/send-recv.c

The rest on the code follows in the next slide

```
    while(1) {
        int payload;
        MPI_Recv(&payload, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);

        if(payload == -1) {
            printf("[%d] received stop signal, exiting\n", my_rank);
            break;
        }

        if(payload == BOUNCE_LIMIT) {
            int stop_payload = -1;
            printf("[%d] broadcasting stop signal\n", my_rank, payload);
            for(int i=0; i<world_size; i++)
                if(i != my_rank)
                    MPI_Send(&stop_payload, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            break;
        }

        payload++;
        int target_rank = my_rank;
        while(target_rank == my_rank) target_rank = rand()%world_size;

        printf("[%d] received payload %d, sending %d to %d\n", my_rank,
                payload-1, payload, target_rank);

        MPI_Send(&payload, 1, MPI_INT, target_rank, 0, MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

13-mpi/send-recv.c 

Collective Operations

- `MPI_Barrier`: barrier synchronisation
- `MPI_Bcast`: broadcast
- `MPI_Scatter`: broadcast parts of a single piece of data to different processes
- `MPI_Gather`: inverse of ‘scatter’
- `MPI_Reduce`: a reduction operation
- etc.

Other MPI Features

- Asynchronous (non-blocking) communication: `MPI_Isend()` and `MPI_Irecv()`
  - Combined with `MPI_Wait()` and/or `MPI_Test()`
- Persistent communication
- Modes of communication
  - Standard
  - Synchronous
  - Buffered
- "One-sided" messaging
  - `MPI_Put()`, `MPI_Get()`

Summary

- MPI: standard to develop parallel applications
  - Exploit parallelism **beyond the boundaries of a single physical machine**
  - Widely used
  - Cost: significant effort to rewrite legacy serial applications with MPI
  - Guide to run applications on a cluster: https://mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan/
- Coming up: **OpenMP**
  - A framework to parallelise shared memory applications with as few effort as possible