Exercise 1

Performance Evaluation of MPI Collective OPerations

Fu Ziyang
SM3800011

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Architecture

Experimental Setup

2 AMD EPYC Nodes: 256 cores

Fixed Message size

Map by Core policy

EPYC Rome node with two sockets

	Hierarchy Level	Components
1	CCX (Core complex)	4 Cores
2	CCD (Core complex Die)	2 Core complexes
3	Socket	8 CCDs: 64 cores
4	Node	2 Sockets: 128 cores

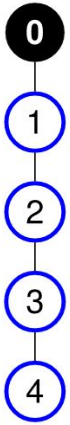
Latencies between different Processor Regions

	Region	Latency
1	Same CCX	0.14 μs
2	Same CCD	0.31 μs
3	Same Socket	0.44 μs
4	Same Node	0.66 μs
5	Different Nodes	1.82 μs

Architecture

MPI_Broadcast

MPI_Broadcast: basic Linear



The root sends the data to the first process in the communicator, which then sends it to the next process, and so on, until all processes have received the data

Estimating communication time using naive model

$(T_{p2p} * (P\text{-}1) + L)$

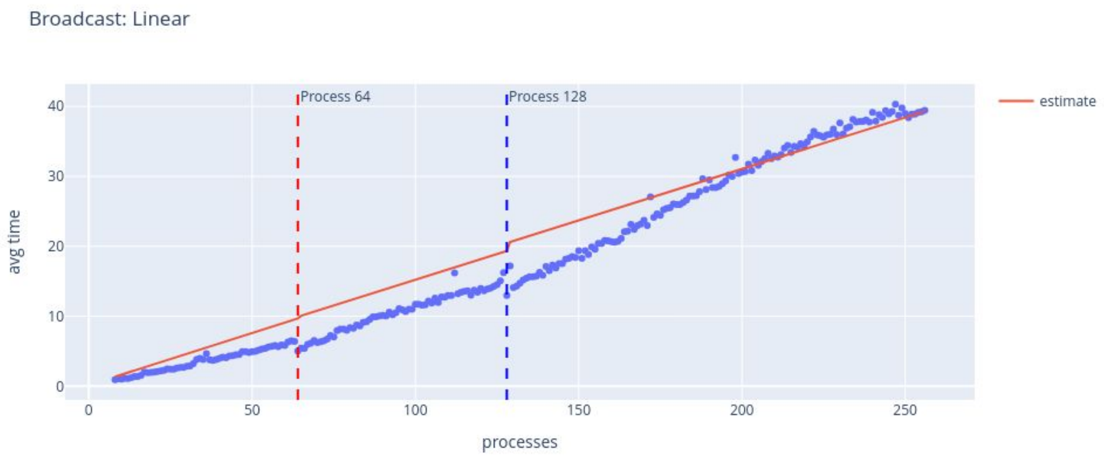
T_{p2p} = point to point communication;

p = number of processes

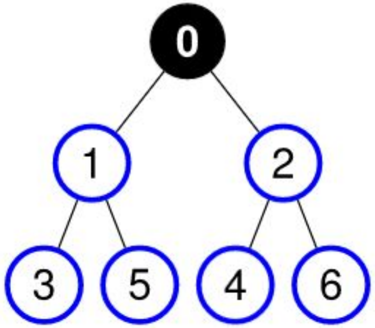
L = estimate latency between the root and the receiver

Basically the sum of P point to point communications

Plotting the estimated communication vs effective time



MPI_Broadcast: Binary Tree



The messages sent from the root traverse the tree starting from the root itself and going towards the leaf nodes through intermediate nodes

Each internal process has two children, and hence data is transmitted from each node to both children

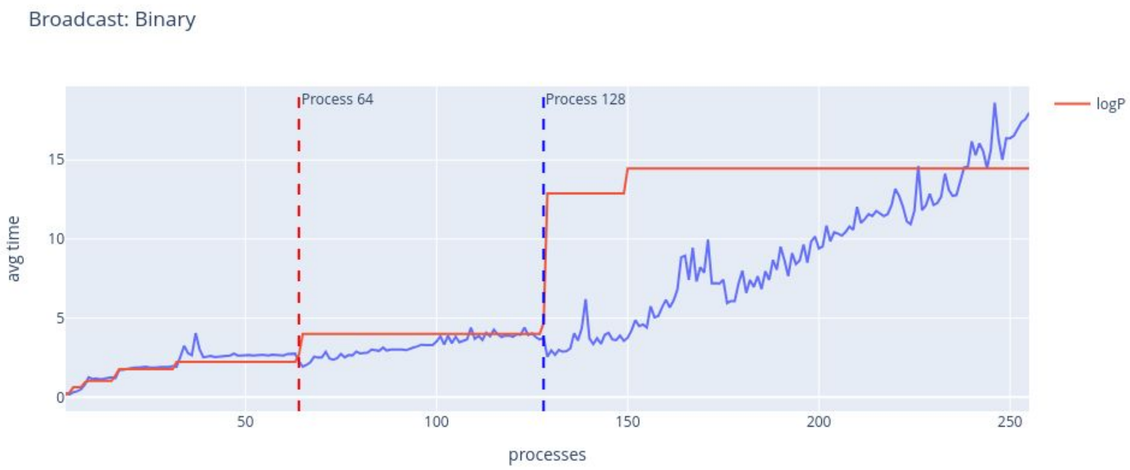
Estimating communication time using LogP model

$\lceil log_2\,(P+1)-1\rceil(L+2o)$

o = the increase in the cost of $P-1$ overlapping non-blocking send operations

It was estimated as the slope of the linear trend: 2 different os
- 0-128
- 128-256

Plotting the estimated communication vs effective time

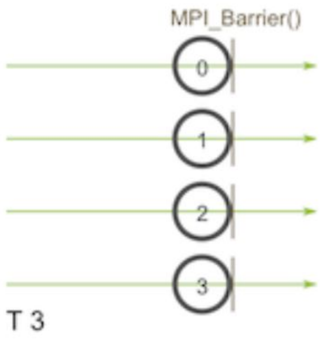


Architecture

MPI_Broadcast

MPI_Barrier

Barrier: Linear



All the processes report to a pre-selected root.

Once every process has done the reporting, the root sends a releasing message to all the processes involved in the communicator.

A node can leave the barrier only after the reception of the pre-said message.

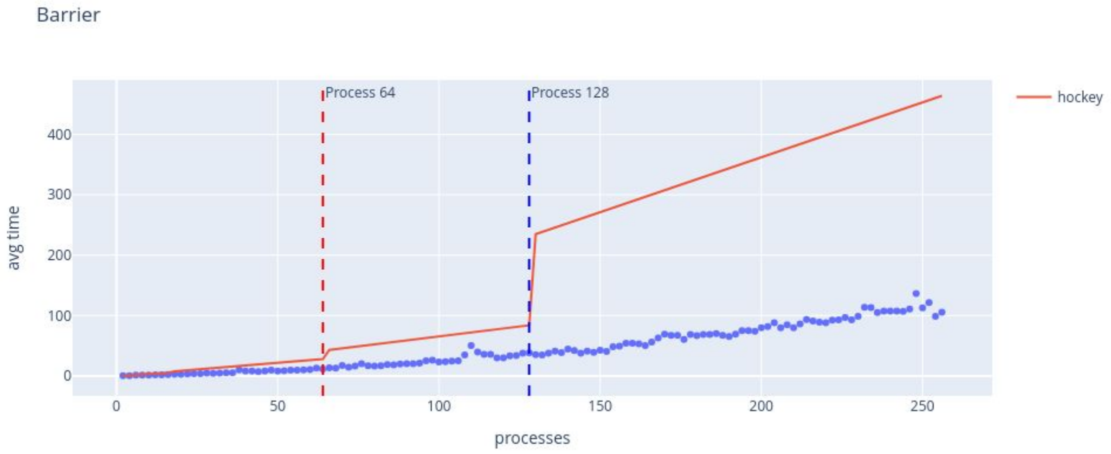
Estimating communication time using Hockney model

$Tlinear\ (P) = (P - 1) \cdot \alpha$

α = latency. We took a proxy of it using the latencies between the different processor regions.

Quite poor results.

Plotting the estimated communication vs effective time



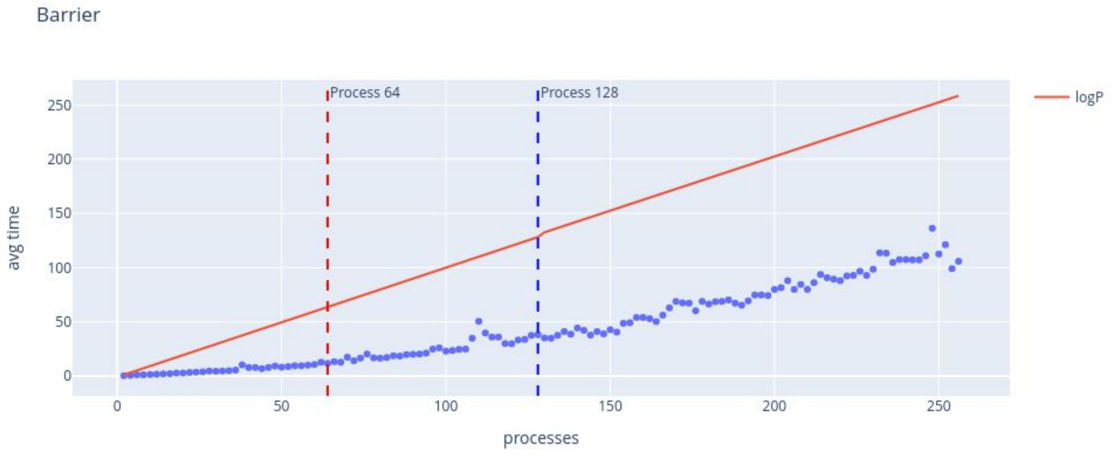
Estimating communication time using LogP model

$Tlinear\ (P) = (P - 2) + 2 \cdot (L + 2 \cdot o)$

o = the increase in the cost of $P - 1$ overlapping non-blocking send operations

It was estimated as the slope of the linear trend

Plotting the estimated communication vs effective time



Exercise 2

Hybrid MPI + OpenMP: the Mandelbrot Set

Fu Ziyang
SM3800011

Partition scheme

First Come First Serve: Row major

Master

Availability and Rows assignment:

```
mb_t *master (int nx, int ny, int size) {
int next_row = 0;
  while (next_row < ny) {
    int available_p;
    MPI_Recv(&available_p, 1, MPI_INT, MPI_ANY_SOURCE, TAG_TASK_REQUEST,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&next_row, 1, MPI_INT, available_p, TAG_TASK_DATA,
    MPI_COMM_WORLD);
    next_row++;
  }
```

Sending the termination signal:

```
for (int i = 1; i < size; i++) {
  int termination_signal = -1;
  MPI_Send(&termination_signal, 1, MPI_INT, i, TAG_TASK_DATA, MPI_COMM_WORLD);
 }
```

Worker

Signaling availability and receiving the rows:

```
void worker (int nx, int ny, double xL, double yL, double xR, double yR, int
Imax, int rank) {
  mb_t **big_array = NULL;
  int *big_index_arrays = (int *)malloc(sizeof(int))
  while (1) {
    MPI_Send(&rank, 1, MPI_INT, 0, TAG_TASK_REQUEST, MPI_COMM_WORLD);
    MPI_Recv(&row_index, 1, MPI_INT, 0, TAG_TASK_DATA, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
    if (row_index == -1) { break;}
    mb_t *row = (mb_t *)malloc(nx * sizeof(mb_t));
    double dy = (yR - yL) / (double)(ny - 1);
    double y = yL + row_index * dy;
```

Computing the rows and storing them:

```
    compute_row(row, nx, xL, xR, y, Imax);
    size++;
    big_array = (mb_t **)realloc(big_array, size * sizeof(mb_t *));
    if (big_array == NULL) { exit(1);}
    big_index_arrays = (int *)realloc(big_index_arrays, size * sizeof(int));
    big_index_arrays[size -1] = row_index;
    big_array[size - 1] = row;
    received_rows++;
    }
```

Master

Receiving back the computed rows and map them into the global matrix

```
mb_t *master (int nx, int ny, int size) {
// ... code from before
  MPI_Status status;
  mb_t *Mandelbrot_1D = (mb_t *)malloc(nx * ny * sizeof(mb_t));
  for (int i = 0; i < ny; i++) {
    int row_index;
    MPI_Recv(&row_index, 1, MPI_INT, MPI_ANY_SOURCE, TAG_TASK_ROW,
    MPI_COMM_WORLD, &status);
    mb_t *row = (mb_t *)malloc(nx * sizeof(mb_t));
    MPI_Recv(row, nx, MPI_UNSIGNED_SHORT, status.MPI_SOURCE, TAG_MATRIX_ROW,
    MPI_COMM_WORLD, &status);
    memcpy(&Mandelbrot_1D[row_index * nx], row, nx * sizeof(mb_t));
  }
  return Mandelbrot_1D;
 }
```

Worker

Sending back the computed rows

```
void worker (int nx, int ny, double xL, double yL, double xR, double yR, int
Imax, int rank) {
  //... code from before
  for (int i = 0; i < received_rows; i++) {
    MPI_Send(&big_index_arrays[i], 1, MPI_INT, 0, TAG_TASK_ROW,
    MPI_COMM_WORLD);
    MPI_Send(big_array[i], nx, MPI_UNSIGNED_SHORT, 0, TAG_MATRIX_ROW,
    MPI_COMM_WORLD);
    }
  // freeing memory...
  return;
  }
```

Setup

Inputs

n_x and n_y are set to 3096
I_max is set to 65535, short int was employed

MPI Scaling

mapped processes using --map-by core
ranging the number of processes from 2 up to 256

OMP Scaling

threads affinity: OMP_PLACES=cores
ranging the number of threads from 2 up to 64

Partition scheme

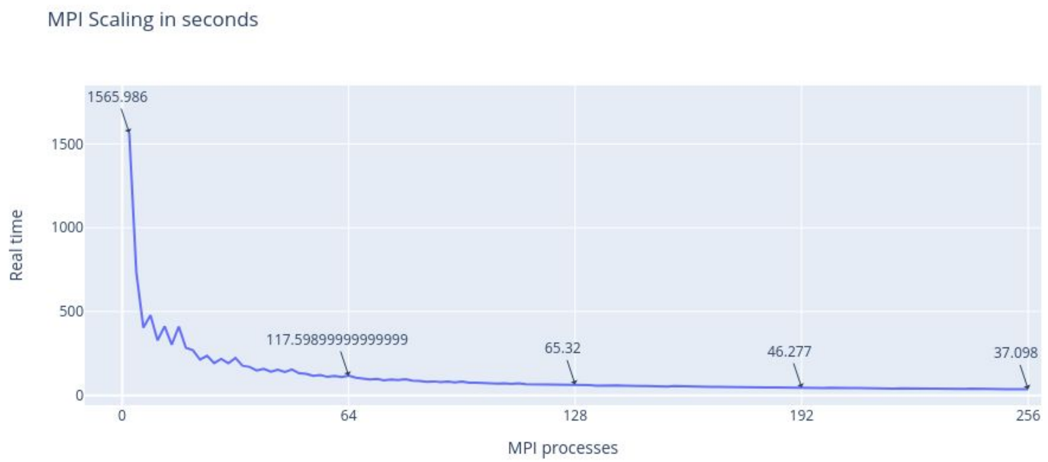
MPI Scaling

MPI Scaling visualizing the execution times

The first highlighted number is the execution time spent by a serial implementation of the code:

master + one worker

first a steep drop in the execution time, but progressive diminishing decrease in the execution time



MPI Scaling visualizing the Speedups

The theoretical Speedup is computed using Amdahl's law:

$$\text{Speedup}(s) = 1 / [(1-p) + p/s]$$

p is the portion of the program that can be parallelized
s in the number of employed processes

Issue

progressive diminishing return, probably due to increasing communication overhead

Solutions

Use collective operations
Use non blocking operations
Reduce the communications

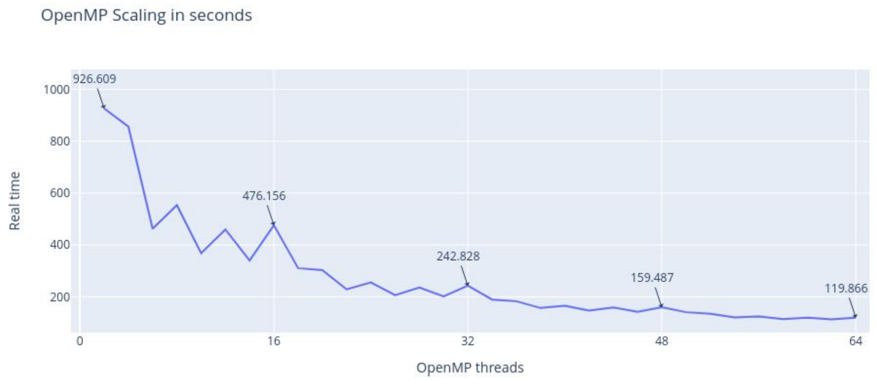
Partition scheme

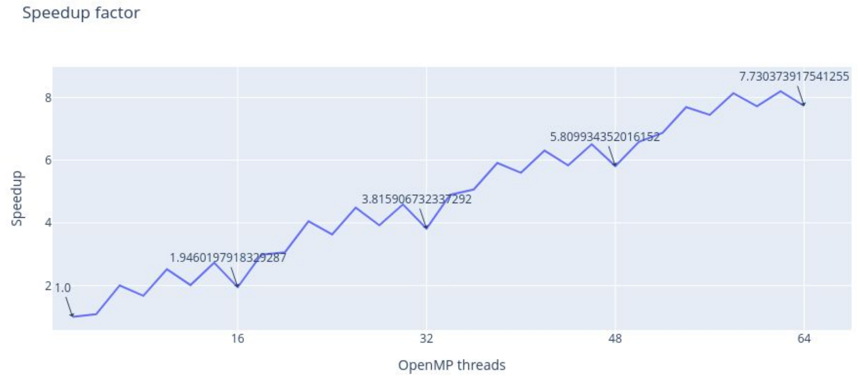
MPI Scaling

OMP scaling

OMP Scaling visualizing the execution times and the Speedups

initial steep decline passing, but as the number of computational units gets increased we observe a progressive diminishing return on the additional threads





Issue: progressive diminishing return,
probably due to **False sharing**

Shared Variables	Mandelbrot array and the variable next_row are shared among all the threads
Invalid Cache line	one thread modifies → other threads need to invalidate their copies
Loading from DRam	Threads are forced to continuously "load" the updated variables from the main memory

Thank You for Your attention