High-Performance Computing Lab for CSE **2023**

Student: Simon Aaron Menzi Discussed with: Lukas Bühler

Solution for Project 4

Due date: 17 April 2023 (midnight)

1. Ring maximum using MPI [10 Points]

This task is a short warm up to the harder tasks in this project. The problem is solved by going through a for-loop and sum up the received number and let the send buffer be equal to the receive buffer afterwards. This is done with Isend and Irecv and Wait but could be accomplished in a different way as well, for example with Isendrecv and Wait. The "left" and "right" ranks can be calculated with some modulo-arithmetic.

```
right = (my_rank + 1) % size;/* get rank of neighbor to your right */
left  = (my_rank - 1) % size;/* get rank of neighbor to your left */

snd_buf = my_rank;
for(i = 0; i < size; i++){
    MPI_Isend(&snd_buf, 1, MPI_INT, right, 0, MPI_COMM_WORLD, &sendRequest);
    MPI_Irecv(&rcv_buf, 1, MPI_INT, left, 0, MPI_COMM_WORLD, &recvRequest);

    MPI_Wait(&sendRequest, &status);
    MPI_Wait(&recvRequest, &status);

    sum += rcv_buf;
    snd_buf = rcv_buf;
}
```

2. Ghost cells exchange between neighboring processes [20 Points]

The Cartesian coordinate system is created with the following code (with 16 processes):

```
dims[0]= 4;
dims[1]= 4;
periods[0]= 1;
periods[1]= 1;

MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm_cart);
int cart_rank;
MPI_Comm_rank(comm_cart, &cart_rank);
```

The derived data type is created with the following C++-code:

```
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm_cart);
int cart_rank;
MPI_Comm_rank(comm_cart, &cart_rank);
```

Last but not least, the communication in a cartesian communication system can be done in numerous ways, with Isend, recv, Isendrecv. The following code snippet shows how it is done with Isend, Irecv and Wait in the task.

```
// top
MPI_Irecv(&data[DOMAINSIZE*(DOMAINSIZE -1)  +1], SUBDOMAIN, MPI_DOUBLE, rank_bottom,
     0, comm_cart, &recv_request[0]);
MPI_Isend(&data[DOMAINSIZE +1], SUBDOMAIN, MPI_DOUBLE, rank_top, 0, comm_cart, &
    send_request[0]);

// bottom
MPI_Irecv(&data[1], SUBDOMAIN, MPI_DOUBLE, rank_top, 0, comm_cart, &recv_request[1])
    ;
MPI_Isend(&data[DOMAINSIZE*(DOMAINSIZE -2)  +1], SUBDOMAIN, MPI_DOUBLE, rank_bottom,
     0, comm_cart, &send_request[1]);

// left
MPI_Irecv(&data[2*DOMAINSIZE -1], 1, data_ghost, rank_right, 0, comm_cart, &
    recv_request[2]);
MPI_Isend(&data[DOMAINSIZE +1], 1, data_ghost, rank_left, 0, comm_cart, &
    send_request[2]);

// right
MPI_Irecv(&data[DOMAINSIZE], 1, data_ghost, rank_left, 0, comm_cart, &recv_request
    [3]);
MPI_Isend(&data[2*DOMAINSIZE -2], 1, data_ghost, rank_right, 0, comm_cart, &
    send_request[3]);


MPI_Waitall(4, send_request, MPI_STATUS_IGNORE);
MPI_Waitall(4, recv_request, MPI_STATUS_IGNORE);
```

3. Parallelizing the Mandelbrot set using MPI [25 Points]

Thanks to the given hints and the provided structures, the coding of the partition and domain is done quickly, fast and with nearly no boiler plate code. The MPI-functions MPI_Cart_create(...) and MPI_Dims_create(...) take over dimension computation and placement. For safety, add-guards were added to the consts.h. The consts.h-file was added to the make-file and the mpi-header file was included in the mandel_mpi.c-file.

As the dimensions do not need to be divisible by the square root of number of processes, the remainder is added to the domains with the highest possible x-rank and/or y-rank.

```
Partition createPartition(int mpi_rank, int mpi_size)
{
Partition p;
int dims[] = {0, 0};
MPI_Dims_create(mpi_size, 2, dims);
p.ny = dims[0];
p.nx = dims[1];

int periods[] = {0, 0};
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &p.comm);

MPI_Cart_coords(p.comm, mpi_rank, 2, &p.x);

return p;
}
```

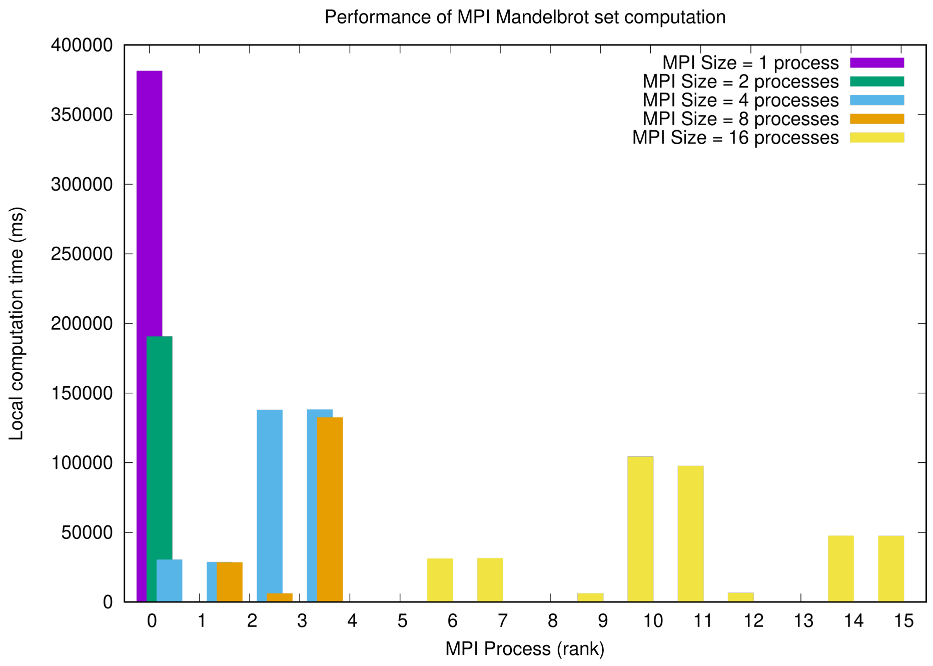
```
Partition updatePartition(Partition p_old, int mpi_rank)
{
.
.
.

MPI_Cart_coords(p.comm, mpi_rank, 2, &p.x);
}
```

```
Domain createDomain(Partition p)
{
.
.
.
d.startx = p.x * d.nx;
d.starty = p.y * d.ny;

if(p.x == p.nx - 1)
    d.nx += IMAGE_WIDTH % p.nx;
if(p.y == p.ny - 1)
    d.ny += IMAGE_HEIGHT % p.ny;

d.endx = d.startx + d.nx;
d.endy = d.starty + d.ny;
.
.
.
}
```

The communication was handled with the following code:

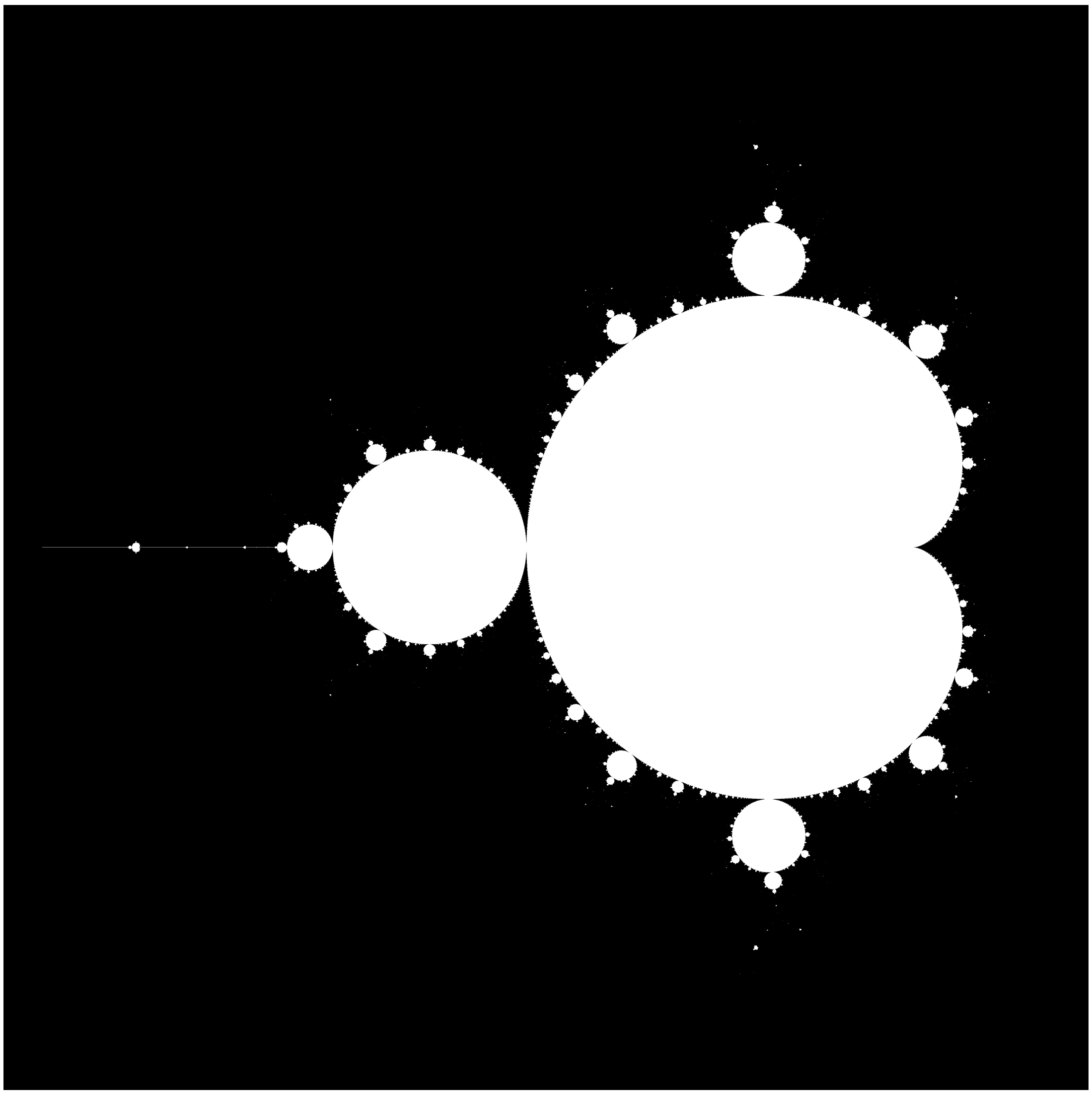
```
. . .
if (mpi_rank != 0)
{
    MPI_Send(c, d.nx*d.ny, MPI_INT, 0, 0, p.comm);
}
.
.
.
for (int proc = 1; proc < mpi_size; proc++){
    . . .
    MPI_Recv(c, d1.nx*d1.ny, MPI_INT, proc, 0, p1.comm, MPI_STATUS_IGNORE);
    . . .
}
```

Every rank sends its data to the master rank (0 rank) which receives the data rank per rank.



In this graphic we see that the workload is not evenly distributed across the ranks. This seems logical as some parts of a Mandelbrot-set-image are totally black, therefore exceeding the circle $|z| = 2$ quickly and thus not needing a lot of calculation. Improvements could be made by changing the partition such that it takes this load distribution information into consideration.

Nonetheless, improvements are made with using multiple compute nodes, as the work is distributed across some of the nodes. But because the workload is not evenly distributed across all the ranks, it will not scale as well as it could.



4. Option A: Parallel matrix-vector multiplication and the power method [45 Points]

This option was chosen as it seemed more familiar. The task is straight forward, as the power method is well explained. An Allgather-approach was chosen for distribution of the vector that is created by the multiplication with the matrix in question. The norm is calculated and spread across all ranks with Allreduce. This approach was used as it seemed the most simple and clear.

```
MPI_Allreduce(&localNorm, &globalNorm, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
globalNorm = sqrt(globalNorm);
localNorm = 0.;
MPI_Allgather(nextX, numRows, MPI_DOUBLE, x, numRows, MPI_DOUBLE, MPI_COMM_WORLD);
```

Due to problems in running this task on Euler, a rather simple approach was taken in benchmarking, which was allocating enough nodes by hand and running the program by hand. This way of benchmarking has reached its limits when wanting to use more than 16 nodes.

Because of bad time management, benchmarking was not conducted further than timing for a single problem size (8192, as it is a multiple of 64) and is included down below. Plots were not generated for the same reason.

Times (s)	processors
63.241510	1
34.297940	2
16.877922	4
11.468892	8
6.692392	16