SPP_Lab2

Task 1 & Task 2

For communicaion using first version(0.470s) not second version(0.559s), and for floating-point add, first version may have better precision.

version 1

```
for(int stride = 1; stride < world_size; stride*=2)
    {
        if(world_rank%(2*stride)==0) {
            double rcv;
            MPI_Recv(&rcv, 1, MPI_DOUBLE, world_rank + stride, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            temp += rcv;
        }
        if((world_rank - stride) % (2 * stride) == 0) {
            MPI_Send(&temp, 1, MPI_DOUBLE, world_rank - stride, 1,
MPI_COMM_WORLD);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
```

version 2

```
  if(world_rank!=0){
        int error = MPI_Send(&temp, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
        if(error != MPI_SUCCESS){
            printf("process %d fail to send \n", world_rank);
        }
  }
  if(world_rank == 0){
      result+=temp;
      for(int i = 1; i<world_size;i++){
          double rcv;
```

```
        MPI_Recv(&rcv,1,MPI_DOUBLE,i,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        result+=rcv;
    }
}
```

mpirun -np 8 ./integral 100000000

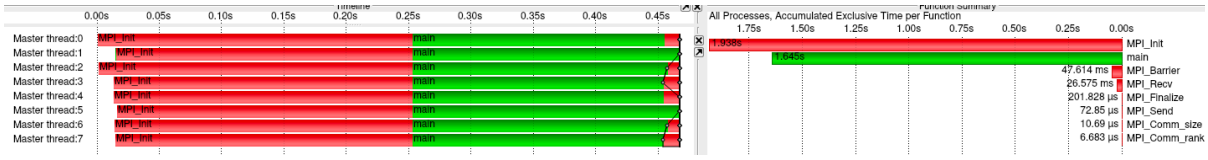
result:3.141531

```
[wy17hosa@logc0001 wy17hosa]$ cd Task2/
[wy17hosa@logc0001 Task2]$ ls
MANIFEST.md  scorep.cfg  traces  traces.def  traces.otf2
```

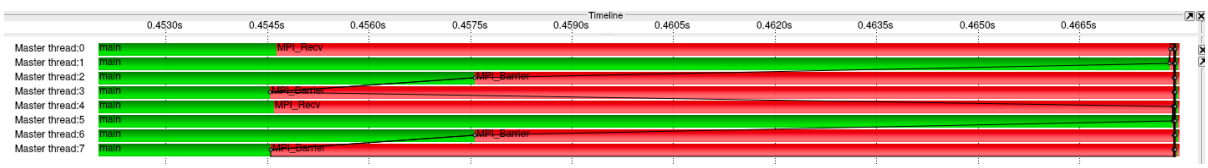
use X11

```
brew install --cask xquartz

Run Applications > Utilities > XQuartz.app

export DISPLAY=:0

ssh -X wy17hosa@lcluster1.hrz.tu-darmstadt.de

cd $HPC_SCRATCH/Task2

module load vampir

vampir traces.otf2
```

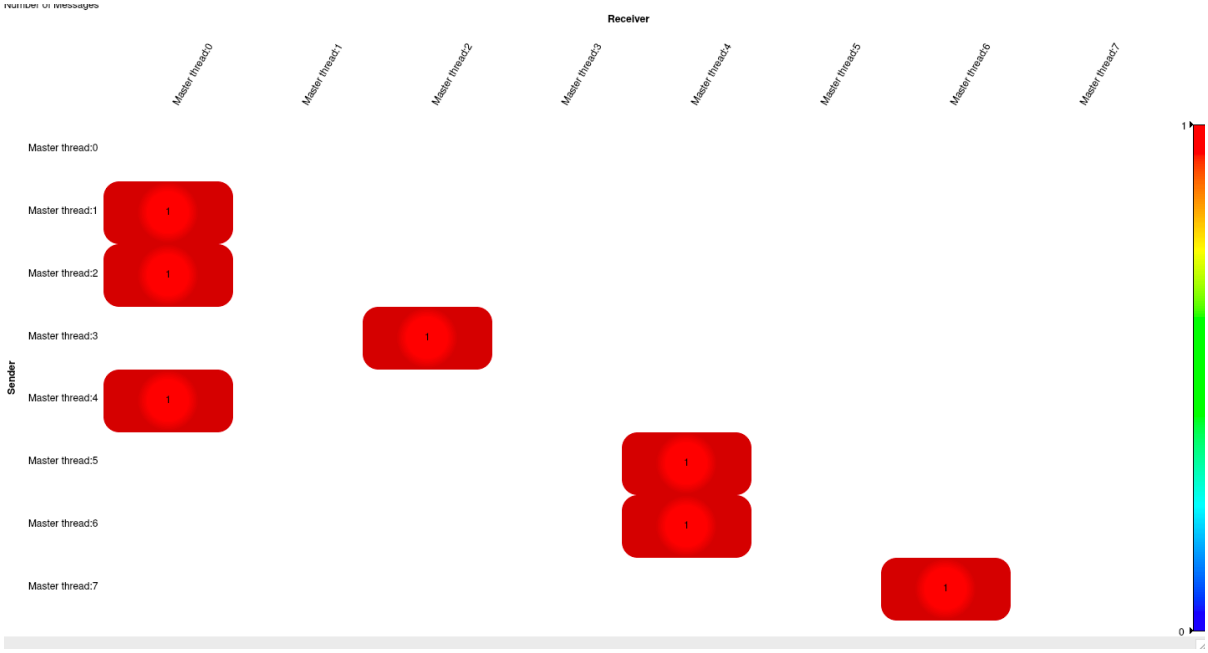
Timeline & Function Summary



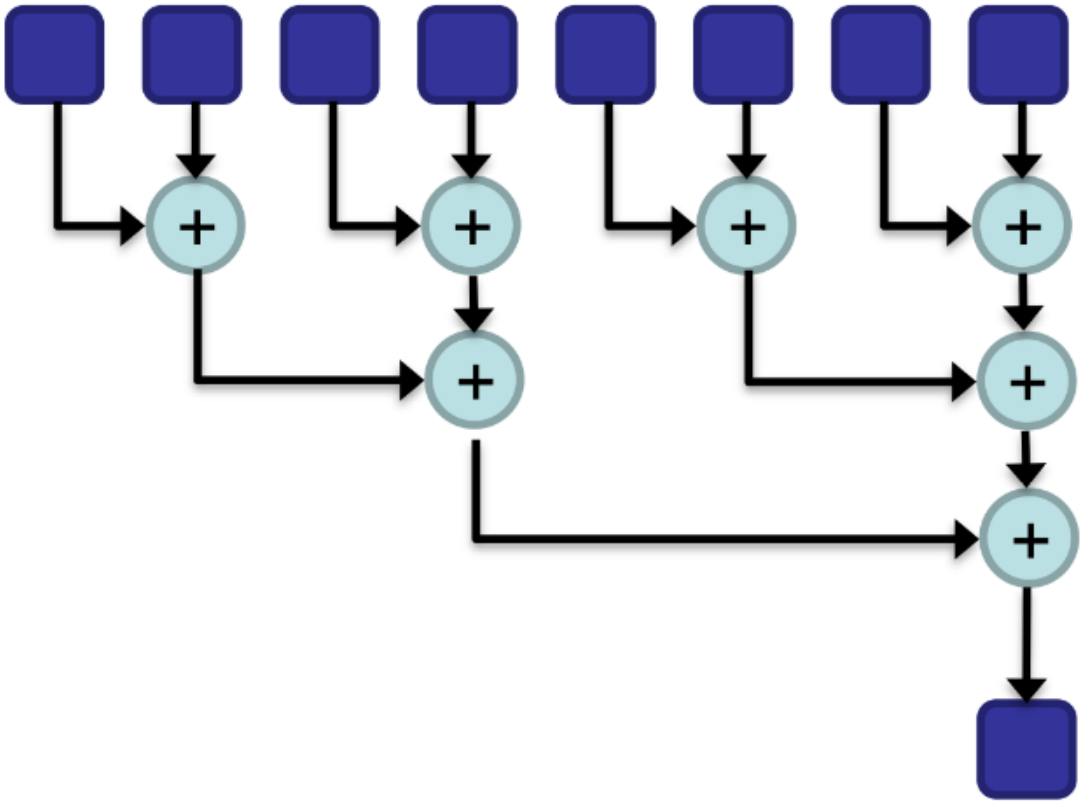
0.4530s ~ 0.4665s



Communication Matrix View for 1.version(what we use)



Use screenshots to describe how the various partial results are brought together.
We follow the thinking in this picture, First add the result of each odd number to the largest even number less than it, then add the result of each even number that is not divisible by four, to the largest number that is less than it and divisible by four, and so on. We can see that on the timeline, every odd-numbered process sends a message to the largest even-numbered process smaller than it, and every even-numbered process that is not divisible by four sends a message to the largest even-numbered process smaller than it is, which is divisible by 4.



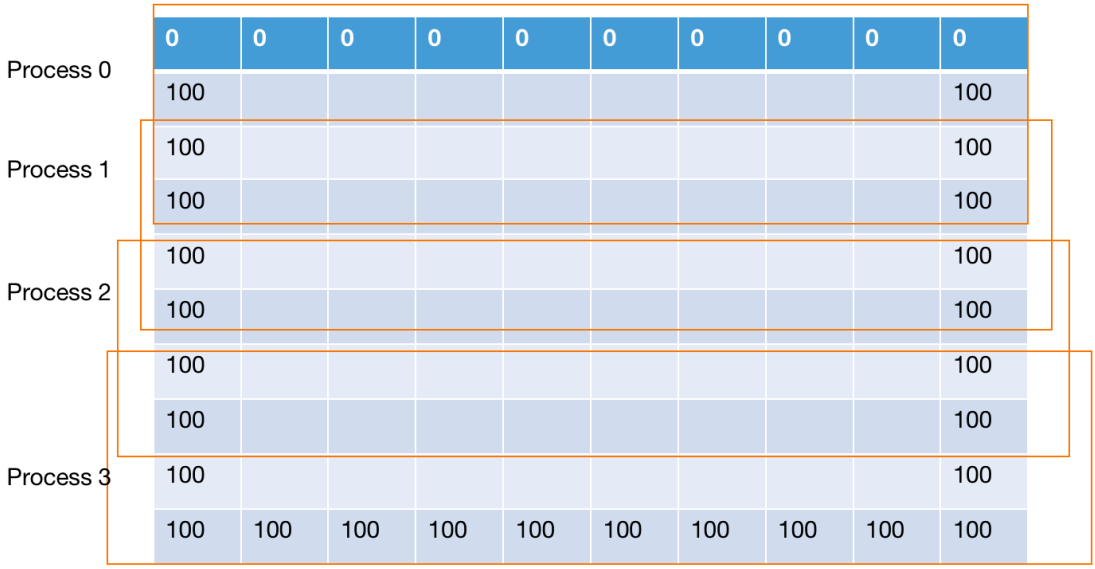
Task 3

a) Skizzieren Sie das Kommunikationsschema.

Example: 10 x 10 matrix and 4 processes.
Calculates the number of valid lines for each process except the first and last line based on the number of processes provided

```
int effectivRow = ceil((double)(M-2)/(double)size);
```

for our example
effectivRow = 2



And to calculate the elements in these rows, we still need up and down 2 row,so

```
int rowProProcess = effectivRow+2;
```

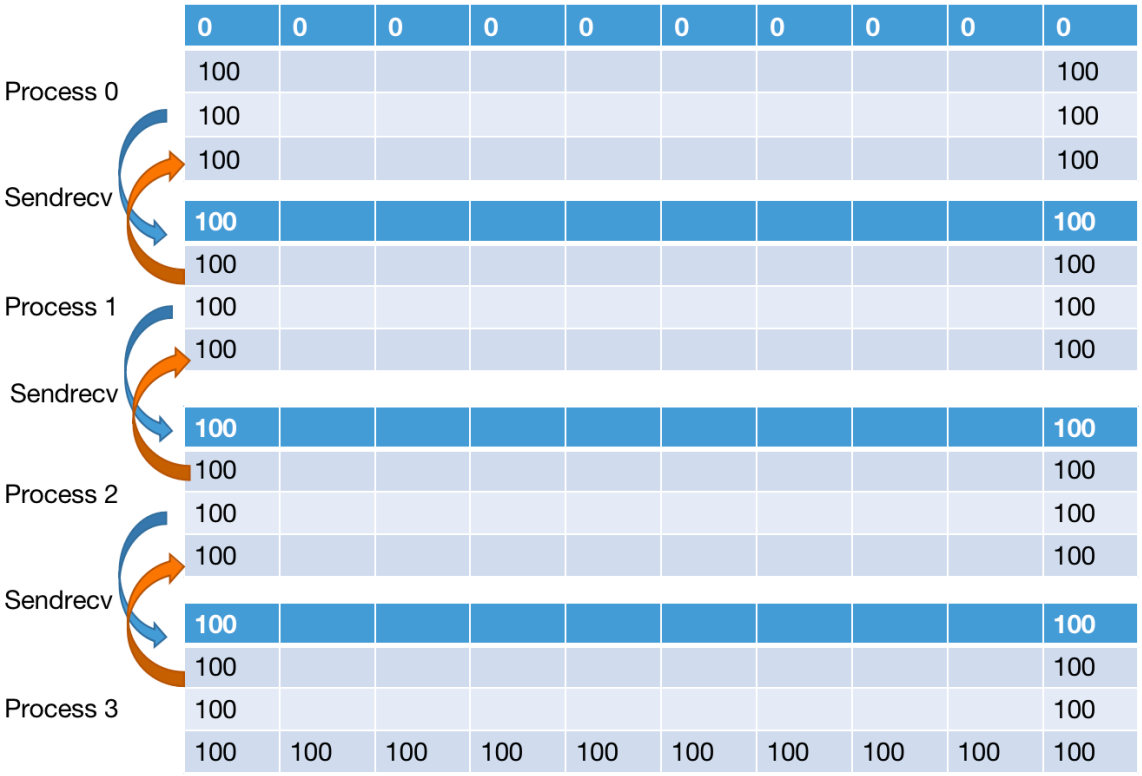
1 process has coverRow row covered by the next process

```
int coverRow = (size * rowProProcess - M)/(size-1);
```

And for the last process, it may be not be filled.The number of row for last process is

```
int end = M-1-effectivRow*(size-1);
```

So the communication method is: each process in each iteration, get the first and the last row from its next process.



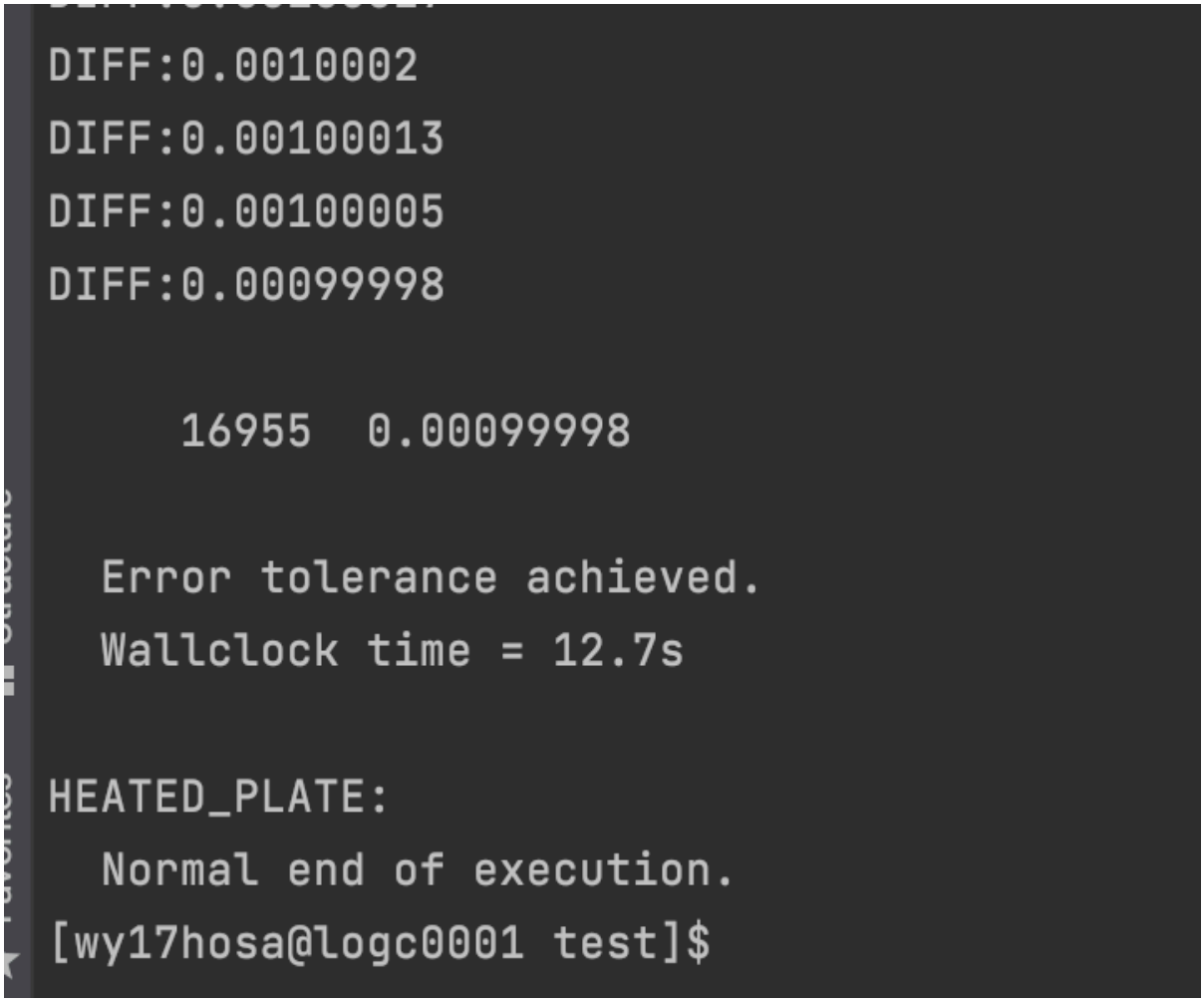
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
100									100
100									100
100									100
100									**100**
100									100
100									100
100									100
100									**100**
100									100
100									100
100									100
100									**100**
100									100
100									100
100	100	100	100	100	100	100	100	100	100

b)

with MPI 8 processes


DIFF:0.0010002
DIFF:0.00100013
DIFF:0.00100005
DIFF:0.00099998
16955 0.00099998
Error tolerance achieved.
Wallclock time = 1.05s
HEATED_PLATE:
Normal end of execution.
[wy17hosa@logc0001 Task3]$

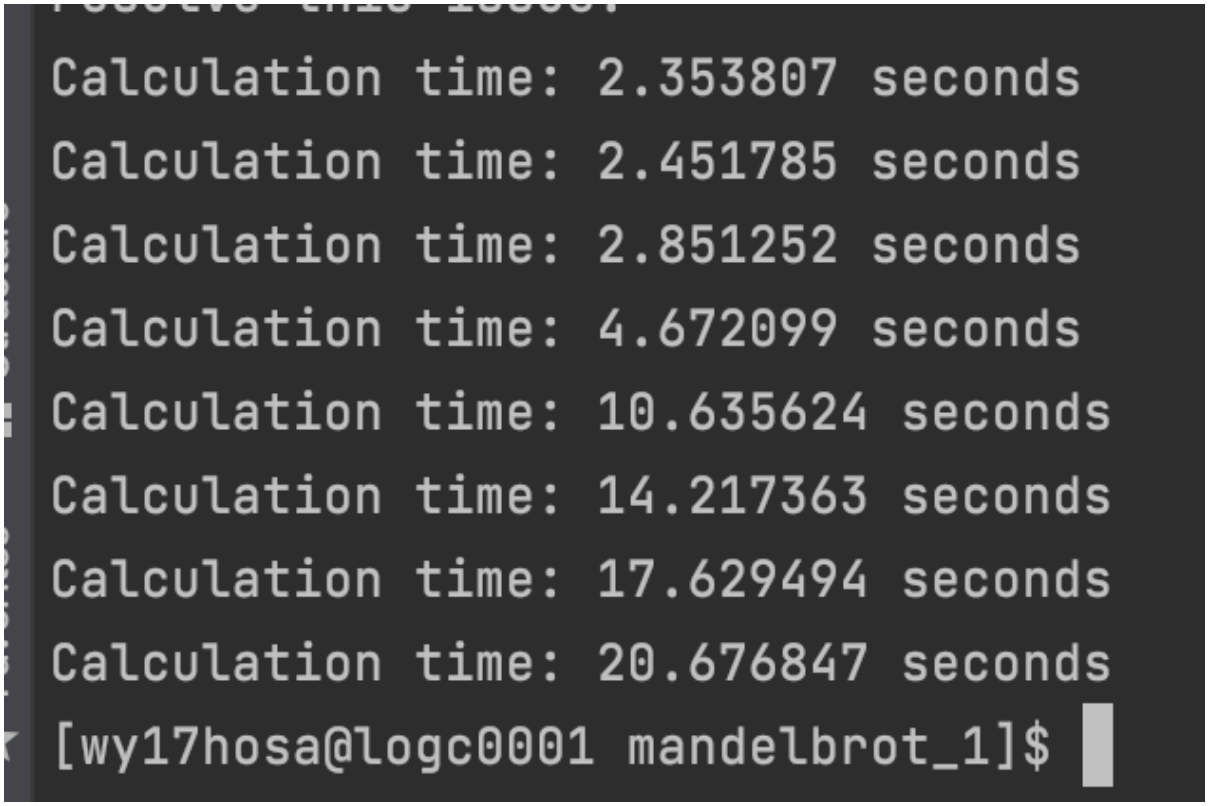
without MPI



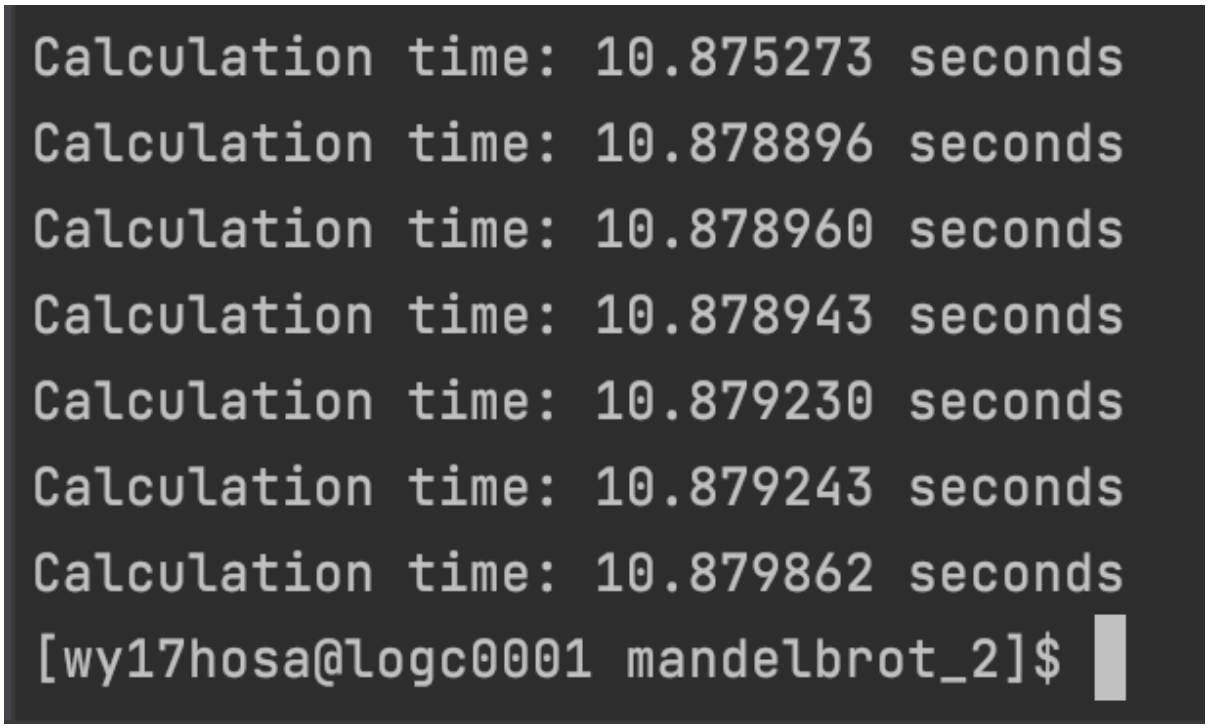
c)
Because any communication of this task do not depends on other communication. And from the communication part to the begin of the next iteration, there is no write or read of the exchanged data.
Advantage: In this part blocking is not necessary, so the non-blocking communication can save time and increase the communication bandwidth of the program

Task 4

Output of Implementation 1:



Output of Implementation 2:



Wie werden in Implementation 1 die verschiedenen Rechenaufgaben (Pixel) verteilt?
Static scheduling, firstly use global height and the number of process to caculate the number of local height, and offset for each process, then each process base on the local

height and offset to do own task.

Wie werden in Implementation 2 die verschiedenen Rechenaufgaben (Pixel) verteilt?
Dynamic scheduling & Master-worker pattern. In n processes, 1 master and n-1 worker, all processes can write in the file. Each task for worker is given by the serial number of row, and worker caculates for maximal MAX_ITER iteration or stoped by condition, then save the result.

Firstly, workers send 0 to master and request their first task(row to caculation).

Every time that master receives a message from k. process, it will give the next task(next row to caculation) to k.process.

Worker process receive the message with next task, then it works. And write the result to the file(with caculating the offset). Then it require next task.

When all task finished, master send -1 to worker, caculation stop.

So we can see in output of this implementation 7 worker processes have almost same caculation time, because they begin and stop at almost same time.

Wie wird die Ausgabedatei geschrieben, gibt es hierbei einen Unterscheid zwischen den Implementationen?

For implementation 1, all process can write in the file. First process write the PPM header. And then all process do their own task, after their own task finished, it will write all results it has finished one by one into the right position of file.

```
int file_offset = image->y_offset * image->global_width * 3;
MPI_File_seek(file, file_offset, MPI_SEEK_END);
for (y = 0; y < image->local_height; ++y)
    for (x = 0; x < image->local_width; ++x) {

      MPI_File_write(file, &(image->data[x][y]), 3, MPI_CHAR,
                     MPI_STATUS_IGNORE);
    }
```

For implementation 2, master write the PPM header, and workers write the content. One different between 1 and 2 is that, in 2. implementation, once one process finish one row, it will write it into file. With MPI_File_write_at(), in 1. implementation is MPI_File_write()

```
MPI_File_write_at(data->file, offset, local_img_row, data->columns * 3,
                  MPI_CHAR, MPI_STATUS_IGNORE);
```

Difference between these 2 functions is:
In 1. : MPI_File_seek change the file pointer with offset and MPI_File_write() write with the file pointer.
In 2. : MPI_File_write_at() can directly write with a specified offset.

Wie viel Kommunikation erfordern die beiden Implementationen?

1. version: no communication
2. version: for n rows image and m process(m-1 workers), require 2n+2m-2 communication

Erlautern Sie mindestens je einen Vor- und einen Nachteil der beiden Implementationen.

1. Implementation
   Advantage: No communication overhead.
   Disadvantage: Static scheduling leads to load imbalance that may cause delay at the end of MPI part, which is waste of computing resources . And each process must save all results that it has computed until writing to file. So each process may need more memory than 2. Implementation.

2. Implementation
   Advantage: Avoid load imbalance. Each process need memory for only one row instead of loc_height * row .
   Disadvantage: Communication overhead. Requires more concurrent access to the output file.

**Unter welchen Umstanden wurden Sie welche Implementation vorziehen? **

If we have enough computing and memory:
{
If the computing for each process is almost the same or the computing is not so big(in this problem means image have not so many height), then the 1. implementation is better. It can save communication overhead.

If the computing for each process is different and the problem is big enough, then the 2.

implementation.
}

if we have limited memory:
2.implementation is better, because it require less memory.