Assignment III

Foundations of High Performance Computing

January 20, 2020

Eros Fabrici

Exercise 1 - Mandelbrot Set (Hybrid, MPI and OpenMP versions with load balancing)

Approach to the Solution and Design

Let N_c be the number of cores, n_x and n_y the variables specified in the requirements.

In order to guarantee the load balancing, instead of dividing in chunks the data-set of dimension $(n_x * n_y)/N_c$, we divide them in a fixed smaller size, namely $chuck\_size = n_x * 20$. In this way, the *master* process will distribute the first $N_c - 1$ chunks to the slaves and as soon as a slave finishes, the master will re-assign a new chuck to it, until there is no work to be done. In addition, more parallelism will be added inside each slave by using threads and in this case load balancing is guaranteed thanks to the pragma parameters *scheduling* and *collapse*.

The program then works in the following way: first we assign the first chucks to the MPI slave processes as mentioned above, then the computation, namely computing the pixels of the assigned chuck, inside each slave is parallelized by using threads.

According to how many MPI processes and threads are assigned to the execution, the program behaves could behave as an MPI implementation, an OpenMP one or an hybrid one. For instance, if we compile without the `-fopenmp` flag (mpiicc compiler), the program will behave an MPI version, on the other hand, if we execute with 1 MPI process and more than one thread, it will work as an OpenMP version (except for writing the file, which in all cases is performed with MPI I/O), while if we execute both with some MPI processes and threads it will be hybrid.

The language used is C, and two programs were made: the serial version and the parallel one. The main difference between the two is the redistribution mechanism used by the master process in the parallel program.

Master

The master, as introduced before, will act as a manager which distributes the work if the `world_size` is grater than 1, otherwise he will perform the whole work. Following the basic algorithm.

1. Open the file and writes the header on the .pgm file, then do and collective open on the file.
2. If the `world_size==1`
   (a) the master will compute the entire Mandelbrot set, write it in the file and terminate.
3. Otherwise, the master will start sending the row offsets to the slaves.
4. The master waits for the acknowledgements from the slaves. When the master receives an ACK, it will assign a new job to the slave which sent the ACK. When there are no more jobs left, it will send an message to the slaves with a tag `TERMINATE`.
5. The master terminates.

For the implementation details, see the method `void master()` in the code `mandelbrot_set.c`.

Slave

The slave algorithm is the following.

1. The slave opens the file with a collective open.
2. It waits for the row offset to be sent by the master.
3. If the tag of this message is equal to `TERMINATE`
   (a) the slave will terminate.
4. Otherwise, according to whether the tag is equal to `DATA` or `REMAINDER`, the size of the chunk to be computed by the slave varies. If `tag==DATA` then the size is the default one, i.e. $n_x * 20$, otherwise it is $n_x * (n_y\%20)$. Subsequently, the slave will compute the assigned chunk, write the results in the .pgm file by using MPI I/O and send and ACK to the master.
5. go to 2.

For the implementation details, see the method `void slave()` in the code `mandelbrot_set.c`.

compute__mandelbrot and compute__pixel methods

From the below method it is possible to analyse the OPENMP parallelization which guarantees load balancing thanks to the *dynamic* scheduling and collapse, which can be checked thanks to the matrix `timer_threads[][]` in which we store the total time for each thread. The argument `row_offset` is used for calculating correctly the imaginary part of $c \in \mathbb{C}$ and the correct offset to be used when writing in the file.

```c
/**
 * method that compute and write the mandelbrot set slice.
 * row_offset represent the from which line of the matrix to start
 * work_amount indicates how many lines to compute
 */
void compute_mandelbrot(unsigned int row_offset, unsigned int work_amount, unsigned short* buffer)
{
    struct complex c;
    unsigned int i, j;
    #ifdef _OPENMP
    #pragma omp parallel for schedule(dynamic, 10) private(c) collapse(2)
    #endif
    for(i = 0; i < work_amount; i++) {
        for(j = 0; j < N_x; j++) {
            #if defined(_OPENMP) && defined(LOAD_BALANCE)
                double s = omp_get_wtime();
            #endif
            c.imag = y_L + (i + row_offset) * d_y;
            c.real = x_L + j * d_x;
            *(buffer + (i * N_x + j)) = compute_pixel(c, I_max);
            #if defined(_OPENMP) && defined(LOAD_BALANCE)
                timer_threads[pid][omp_get_thread_num()] += omp_get_wtime() - s;
            #endif
        }
    }

    //write the results using offset
    MPI_File_write_at(output_file,
                      header_size * sizeof(char) + row_offset * N_x * sizeof(unsigned short),
                      buffer, N_y * work_amount, MPI_UNSIGNED_SHORT, &file_stat);
}
```

The pixel are computed as showed below, as example of the result (mind that it is a screenshot of the pgm image) can be viewed in Figure 1.

```c
/**
 * Function that given a complex number c and and integer,
 * computes if c belongs to the Mandelbrot Set and returns
 * the counter used in the loop
 */
unsigned short compute_pixel(struct complex c, unsigned short max_iter)
{
    unsigned short count = 0;
    struct complex z;
    z.real = 0.0; z.imag = 0.0;
    double temp;

    do
    {
        temp = z.real * z.real - z.imag * z.imag + c.real;
        z.imag = 2 * z.real * z.imag + c.imag;
        z.real = temp;
    } while ((z.real * z.real + z.imag * z.imag < 4.0) && (count++ < max_iter));

    return count;
}
```

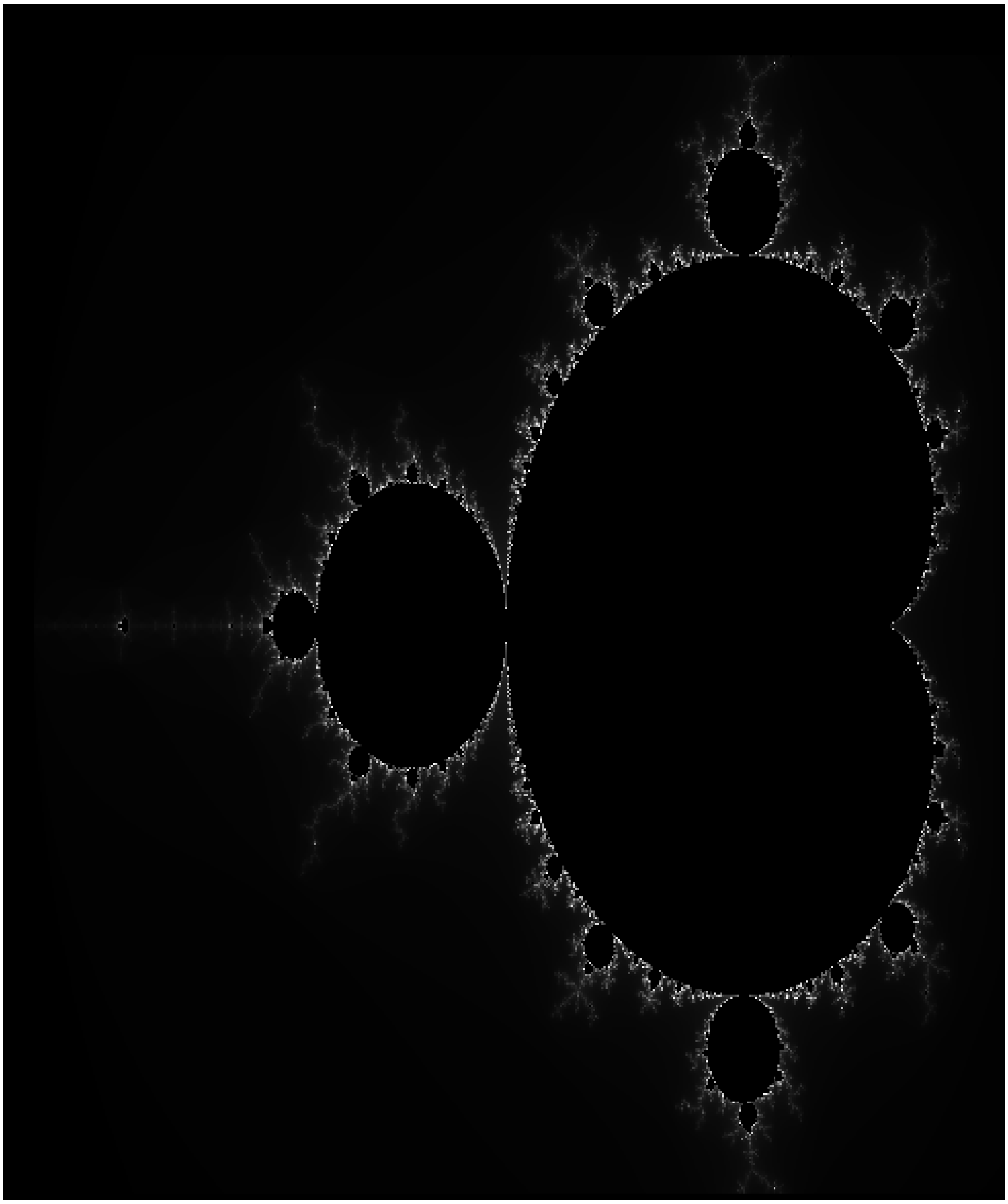

Figure 1: An screenshot of the mandelbrot set 1000x1000 pixel

Analysis

Strong and weak scaling

The tests for both strong and weak scaling were made on a single Ulysses node and with just MPI and OPENMP.

As expected, OPENMP performs slightly better that MPI withing the single node, even though they both scale perfectly (Fig. 2 and 3.

The input for the strong scaling was: $n_x = 2000$ $n_y = 2000$ $x_r = -2$ $x_r = 1$ $y_l = -1$ $y_r - 1.0$ $I_max = 65535$, while for the weak scaling test $n_x = n_y \in 500, 707, 1000, 1225, 1421, 1579$ with respectively 2, 4, 8, 12, 16 and 20 cores (or threads).

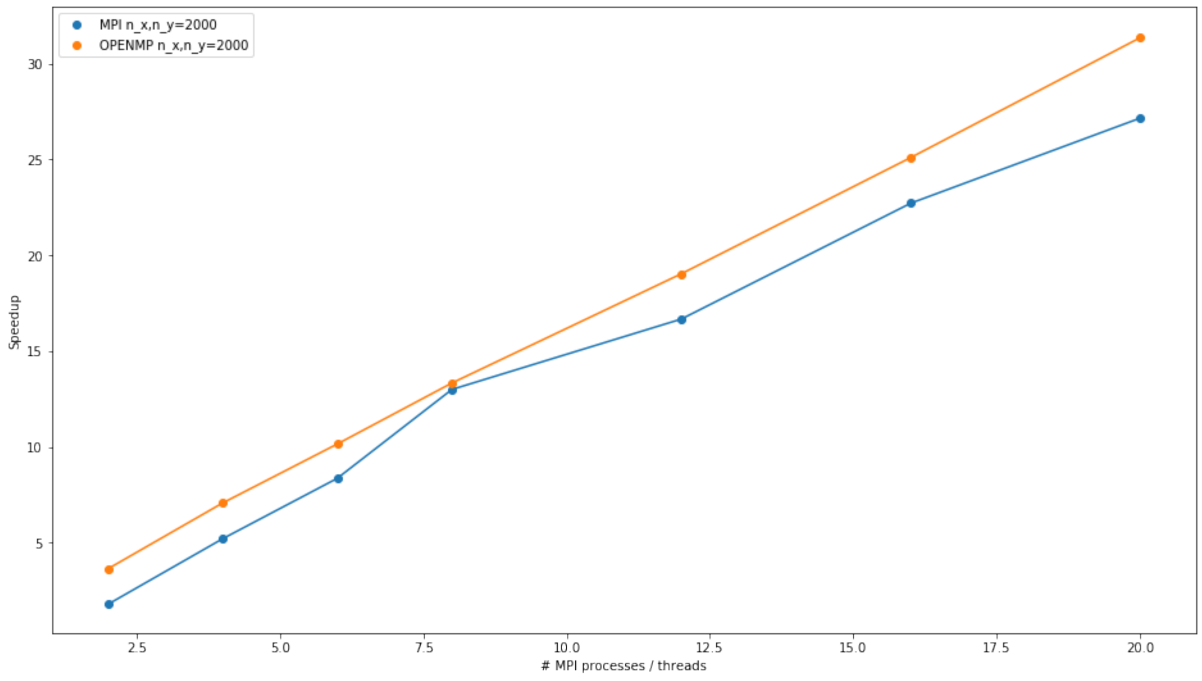

Figure 2: MPI vs OPENMP strong scaling

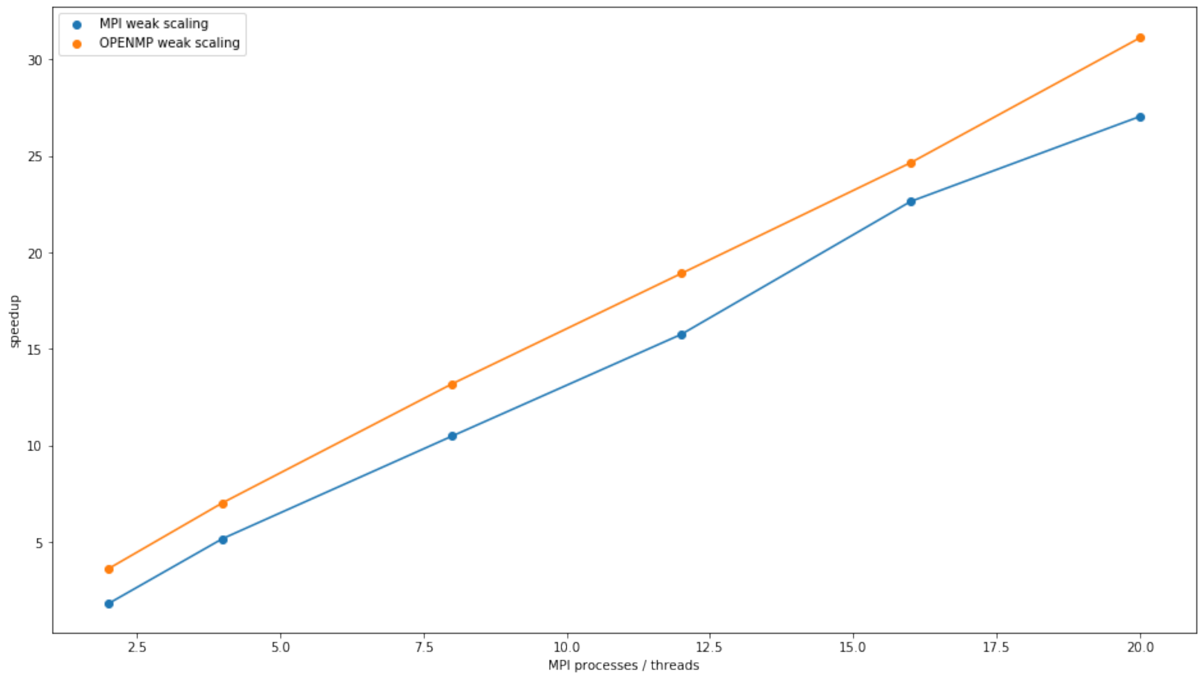

Figure 3: MPI vs OPENMP weak scaling

Further tests

I tested the hybrid on a single node and the results are shown in Figure 4.

However, the best thing to test hybrid is to create a set of MPI processes, each of one located in a different node, and use 20 threads for each of the node. Unfortunately, I was not able to test this as when I was using the following script

```
echo "Hybrid version"
module purge
module load openmpi/1.8.3/intel/14.0
module load impi-trial/5.0.1.035
cd mandelbrotSet/src
mpiicc -fopenmp mandelbrot_set.c -o mandelbrot_set_hybrid_mult_nodes.x
export I_MPI_JOB_RESPECT_PROCESS_PLACEMENT=off
export OMP_PLACES=cores
export OMP_PROC_BIND=spread

OLDIFS=$IFS; IFS=','; for procs in 1,20 2,39 3,29 ; do set -- $procs;
    export OMP_NUM_THREADS=$2
    mpiexec.hydra -n $1 -ppn 1 ./mandelbrot_set_hybrid_mult_nodes.x 4000 4000 -2.0 -1.0 1.0 1.0 65535;
done; IFS=$OLDIFS
```

I am getting the following errors:

```
libibverbs: Warning: no userspace device-specific driver found for
/sys/class/infiniband_verbs/uverbs0
librdmacm: Fatal: no RDMA devices found
librdmacm: Fatal: no RDMA devices found
librdmacm: Fatal: no RDMA devices found
librdmacm: Fatal: no RDMA devices found
librdmacm: Fatal: no RDMA devices found
librdmacm: Fatal: no RDMA devices found
librdmacm: Fatal: no RDMA devices found
librdmacm: Fatal: no RDMA devices found
librdmacm: Fatal: no RDMA devices found
librdmacm: Fatal: no RDMA devices found
```

I tried to run the MPI version in multiple nodes, but it returned errors. Following the script and error.

```
echo "MPI version"
module purge
module load openmpi/1.8.3/intel/14.0
module load impi-trial/5.0.1.035
cd mandelbrotSet/src
mpiicc mandelbrot_set.c -o mandelbrot_set_MPI_mult_nodes.x

for procs in 20 40 60; do
    mpiexec.hydra -n ${procs} ./mandelbrot_set_MPI_mult_nodes.x 4000 4000 -2.0 -1.0 1.0 1.0 65535
done
```

```
[71:cn08-19] unexpected DAPL connection event 0x4008 from 39
Assertion failed in file ../../src/mpid/ch3/channels/nemesis/netmod/dapl/dapl_poll_rc.c
at line 1679: 0
internal ABORT - process 71
[55:cn08-18] unexpected disconnect completion event from [71:cn08-19]
Assertion failed in file ../../src/mpid/ch3/channels/nemesis/netmod/dapl/dapl_conn_rc.c
at line 1179: 0
internal ABORT - process 55
[15:cn01-19] unexpected disconnect completion event from [55:cn08-18]
Assertion failed in file ../../src/mpid/ch3/channels/nemesis/netmod/dapl/dapl_conn_rc.c
```

```
at line 1179: 0
internal ABORT - process 15
[31:cn04-04] unexpected disconnect completion event from [15:cn01-19]
Assertion failed in file ../../src/mpid/ch3/channels/nemesis/netmod/dapl/dapl_conn_rc.c
at line 1179: 0
internal ABORT - process 31
[47:cn08-18] unexpected disconnect completion event from [31:cn04-04]
Assertion failed in file ../../src/mpid/ch3/channels/nemesis/netmod/dapl/dapl_conn_rc.c
at line 1179: 0
internal ABORT - process 47
```

I tried to do the same thing in Ulysses v2 Beta, but I was not able to find Intel MPI, therefore I tried with Open MPI, but I ran into problem of threads not being properly pinned to the cores (even by setting `OMP\_PLACES=cores OMP\_PROC\_BIND=spread` and other slurm options).

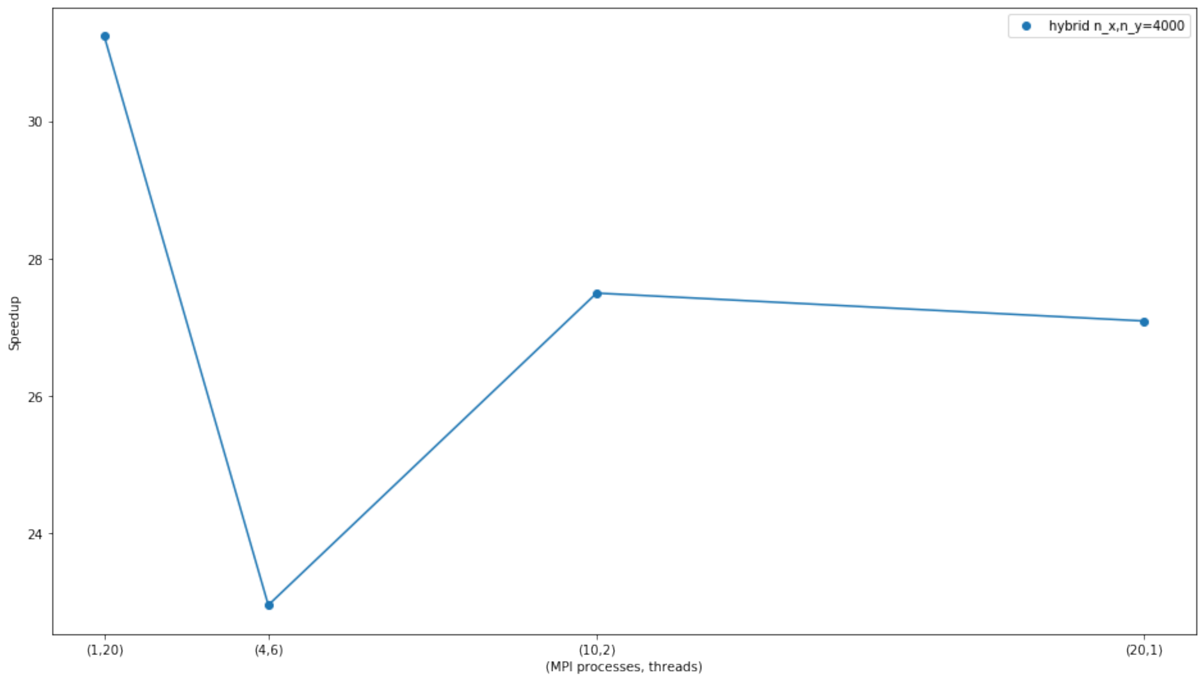

Figure 4: Hybrid strong scaling

MPI strong and weak scaling (single node) - Ulysses v2

Despite the problems described before, I managed to compute weak (Fig. 6) and strong scaling (Fig. 5) in a single node of Ulysses v2, up to 32 MPI processes and with $n_x = n_y = 4000$.

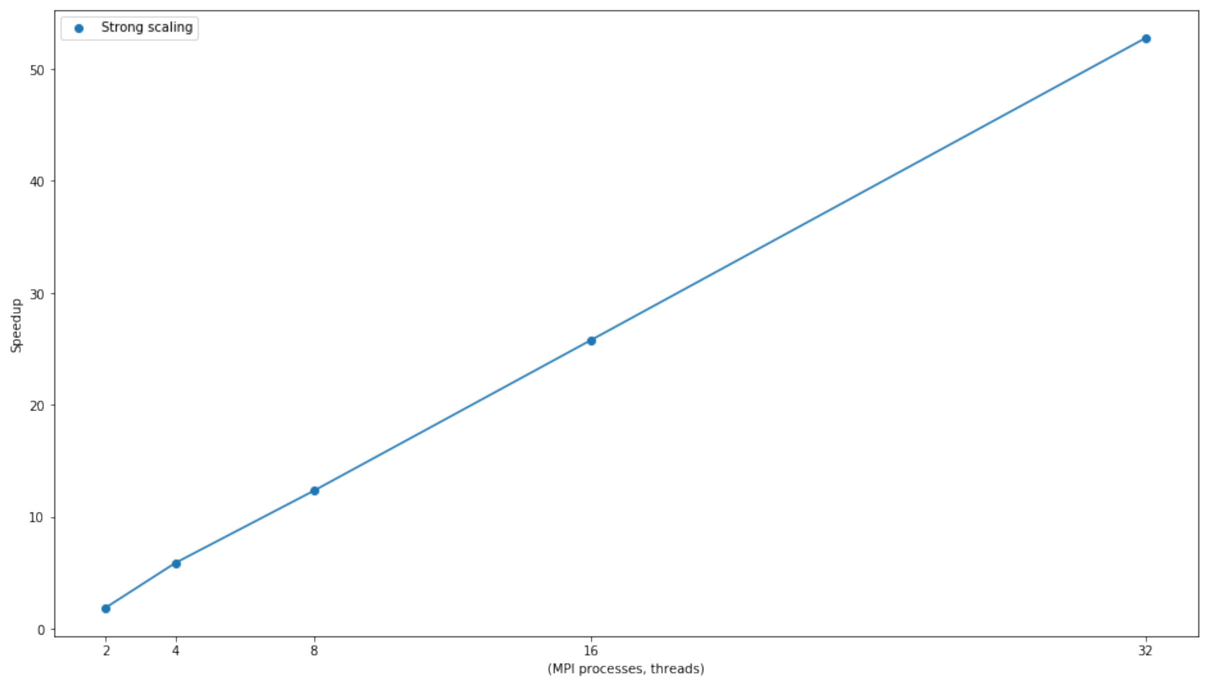

Figure 5: MPI Strong scaling on Ulysses v2

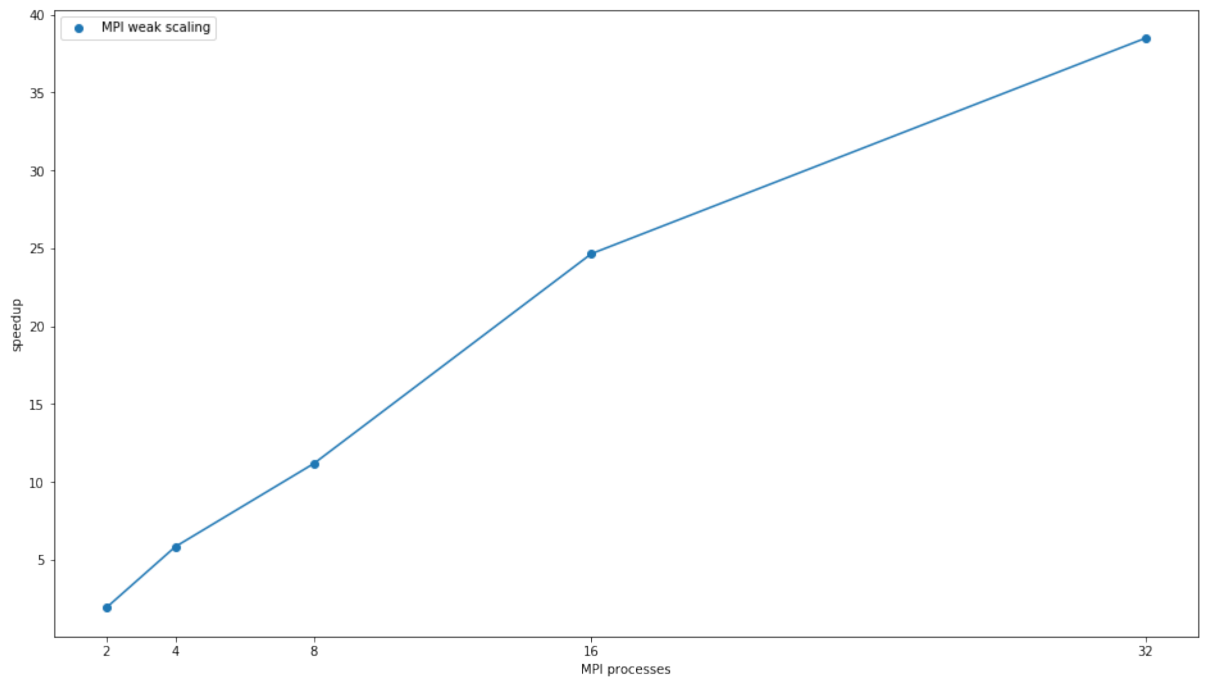

Figure 6: MPI weak scaling on Ulysses v2

Appendix

For compilation I used `mpiicc` or `mpicc`. The modules used in the cluster can be seen in the .sh files in this project folder. The program accepts the same input as specified in the requirements. For compiling the serial program, I used gcc 4.9.2, which requires to add the flag -lrt at the end.

Note that `I_max` has to be 65535.