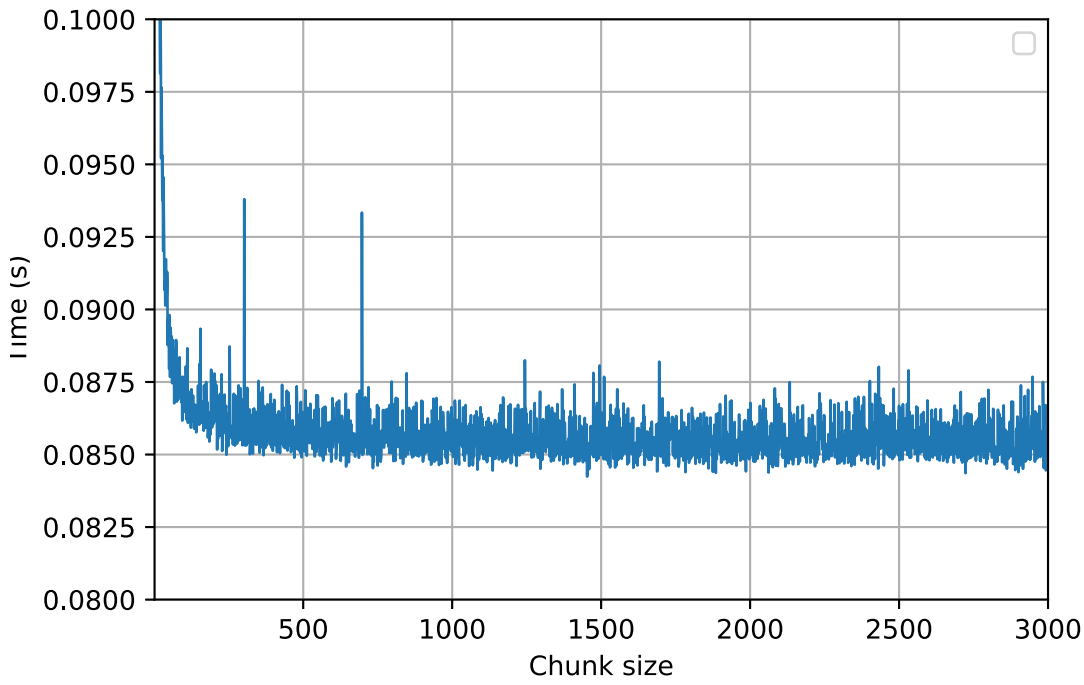
Assignment III

Programs

Serial version of the program was written using `write_pgm_image()` giving a grayscale image as output.

OpenMP

The OpenMp version of the program was simply achieved using a `#pragma omp parallel for schedule(dynamic, 100)` in which the scheduling ensure an optimize division of the work between each thread, preventing unbalance cause by different computation time required by different parts of the image. In order to decide the most appropriate chunk size, total times required by the program was plotted at increasing sizes. Using 10 threads the results are plotted below:



Starting from a maximum value of about 0.38 seconds, the computation time quickly set (at about chunk size 10) at more or less 0.085 seconds. If the chunk size is too big, the time required is expected to grow again because of the work imbalance. In our cases to the chunk size was given the secure number 100.

MPI

The MPI version of the program reserve the use of the Master process for communications between processes. For this reason all the data collected from this program starts from -np 2. Every slave process, whenever available, send a message to the Master. The Master send back an index, telling the slave process which part of the image he should analyses (a line at a time). Then using MPI I/O, when it has finished, every process write the result on a common file, and then became available again, sending again a message to the Master. When the Master acknowledge

that the whole image has been processed, it send to all processes a massage to stop them. The details about the communications and writing command are written as comment in the code.

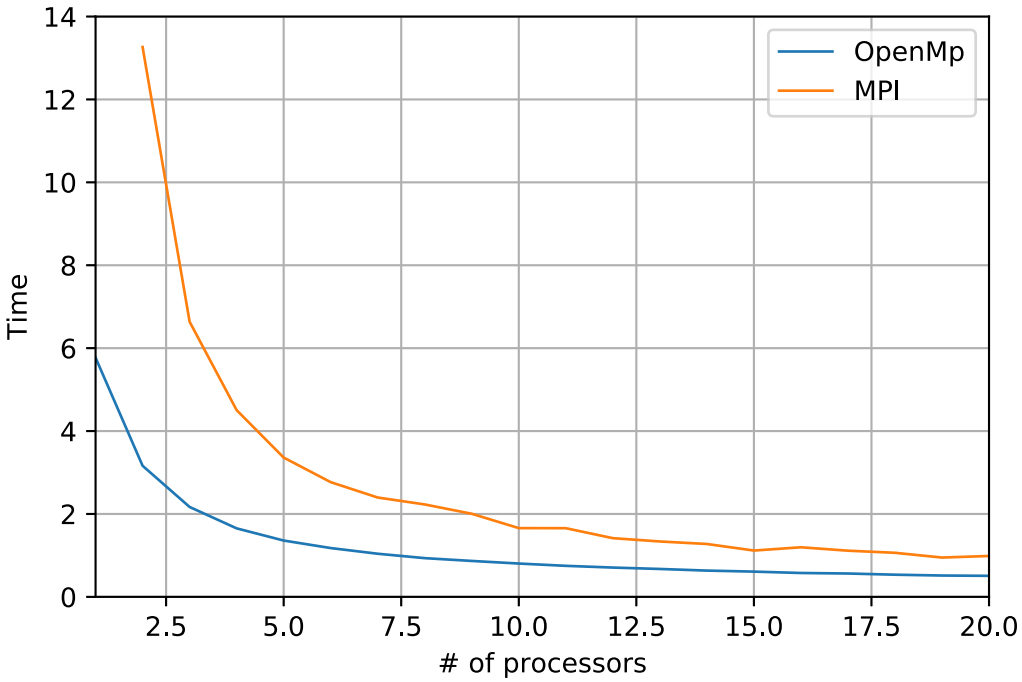
As it is, only one line is assigned to every process as fast as it became available . Considering that in the default image there are 4000 of them, while we used at most 19 working threads, it is safe to say that workload imbalance can be neglected. Work imbalance is expected to start slowing down the code when the part processed by thread is at least comparable to the total work load. In the case considered the total work load is 32000000, four order of magnitude bigger than what is assigned by default to every process.

Strong scaling

Strong scaling measures was recorded using the following bash scrip (for the OpenMP version):

```
export KMP_AFFINITY=granularity=fine,scatter
module load intel/18.4
icc  mandelbrot_openMP.c -o mandelbrot_openMP-x -DOUTPUT -std=c99 -lrt -fopenmp
for procs in {1..20}
do
export OMP_NUM_THREADS=${procs}
/usr/bin/time ./mandelbrot_openMP.x
done
```

With minor changes for the MPI versions, the dimension of the image was kept constant, while the number of threads was constantly increasingly up to 20. Results for time are plotted below:

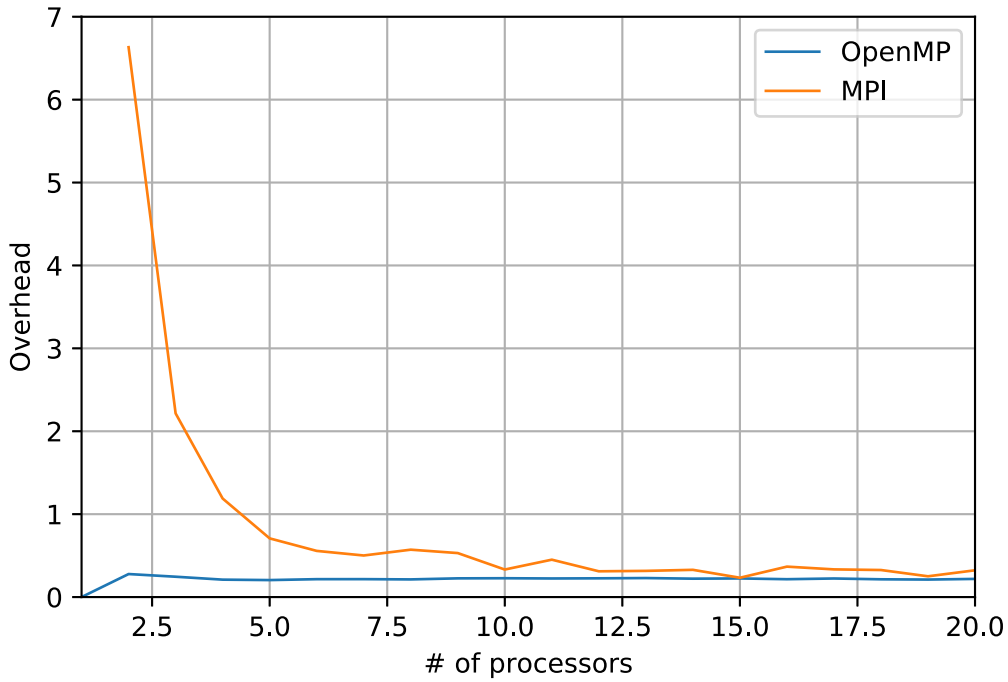


The results aren't surprising, considering that OpenMP is already optimized, so attempting to manually organize communications between threads leads to slower computation times.

The overhead was estimated using the following formula:

$$O = T_p - \frac{T_s}{p}$$

Where T_p is the time taken by the parallel program, while T_s the time taken by the serial one.

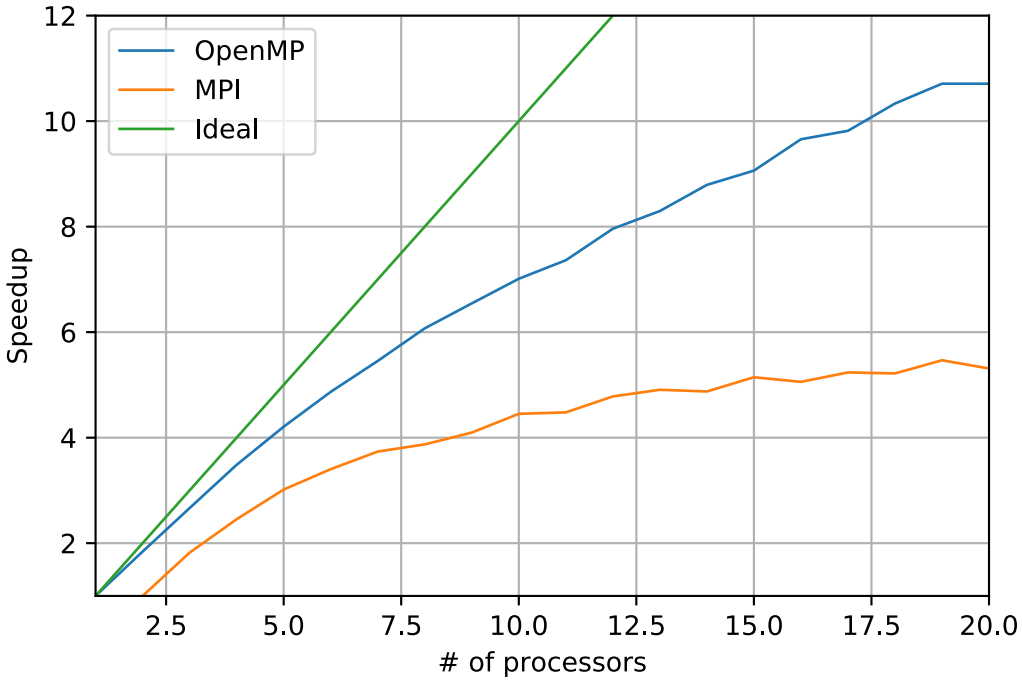


Overhead, in the MPI program, became progressively less important as the number of processes used increase. That is caused by the fact that communication time is overwhelmed by the speed gained by the parallelization.

The speedup of a parallel program is defined as

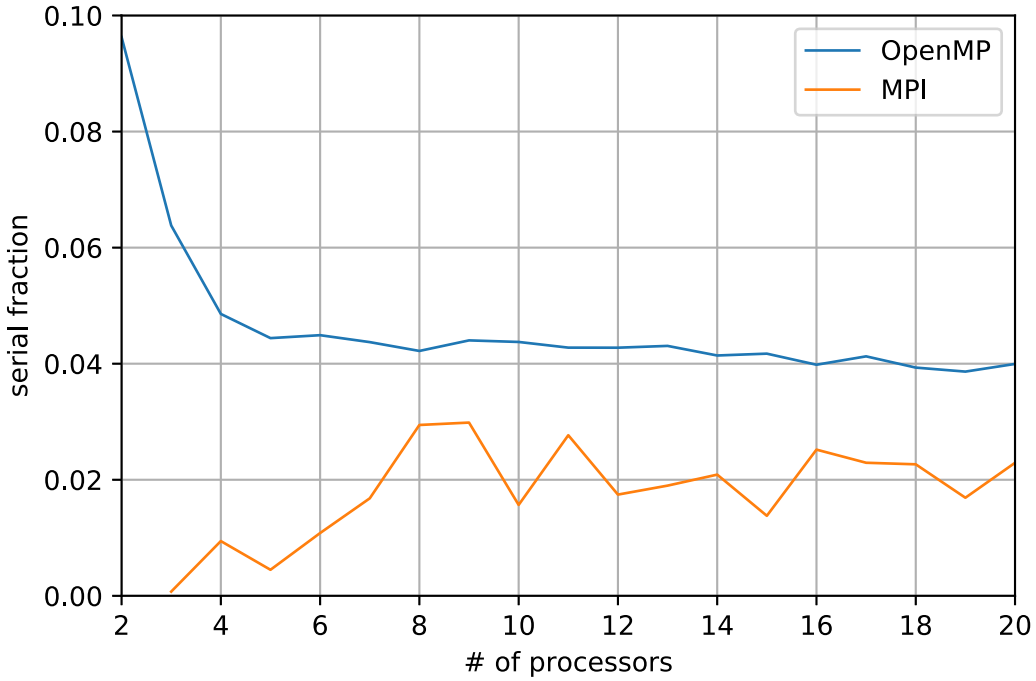
$$S_p(P) = \frac{T_1}{T_P}$$

where T_1 denotes the execution time of the serial algorithm, while T_P the execution time of a parallel code solving the same problem on P processor. Considering that the two programs also write the image in different ways, for the speedup plot the elapsed time was used, giving a more fair comparison:



Values of the serial fraction was estimated using the definition formula:

$$e(n,p) = \frac{\frac{1}{S_p(n,t)} - \frac{1}{p}}{1 - \frac{1}{p}}$$



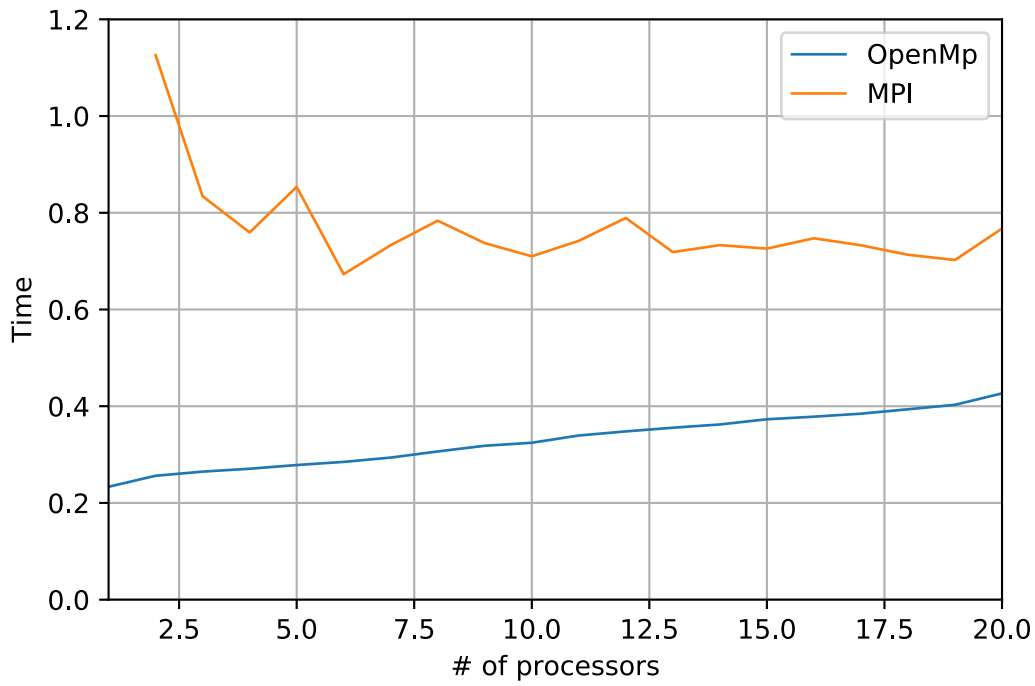
Considering that the serial fraction for the OpenMP program is directly given by the chunk size, it is not surprising that the serial fraction quickly set to steady value of about 0.04.
MPI on the other end is heavily dependent on how the portion of the complex plane is divided among the process, therefore it shows an erratic behavior.

Weak scaling

Weak scaling measures was recorded using the following bash scrip (for the OpenMP version):

```
export KMP_AFFINITY=granularity=fine,scatter
module load intel/18.4
icc  mandelbrot_openMP.c -o mandelbrot_openMP.x -DOUTPUT -std=c99 -lrt -fopenmp
for procs in {1..20}
do
c=$(awk -v x=${procs} 'BEGIN{print sqrt(x)}')
i=1600
ii=800
a=$(echo ${c} ${i} | awk '{printf "%4.3f\n",$1*$2}')
a=$(echo ${a} | awk '{print int($1)}')
b=$(echo ${c} ${ii} | awk '{printf "%4.3f\n",$1*$2}')
b=$(echo ${b} | awk '{print int($1)}')
export OMP_NUM_THREADS=${procs}
/usr/bin/time ./mandelbrot_openMP.x ${a} ${b} -2.5 1 1.5 -1 255
done
```

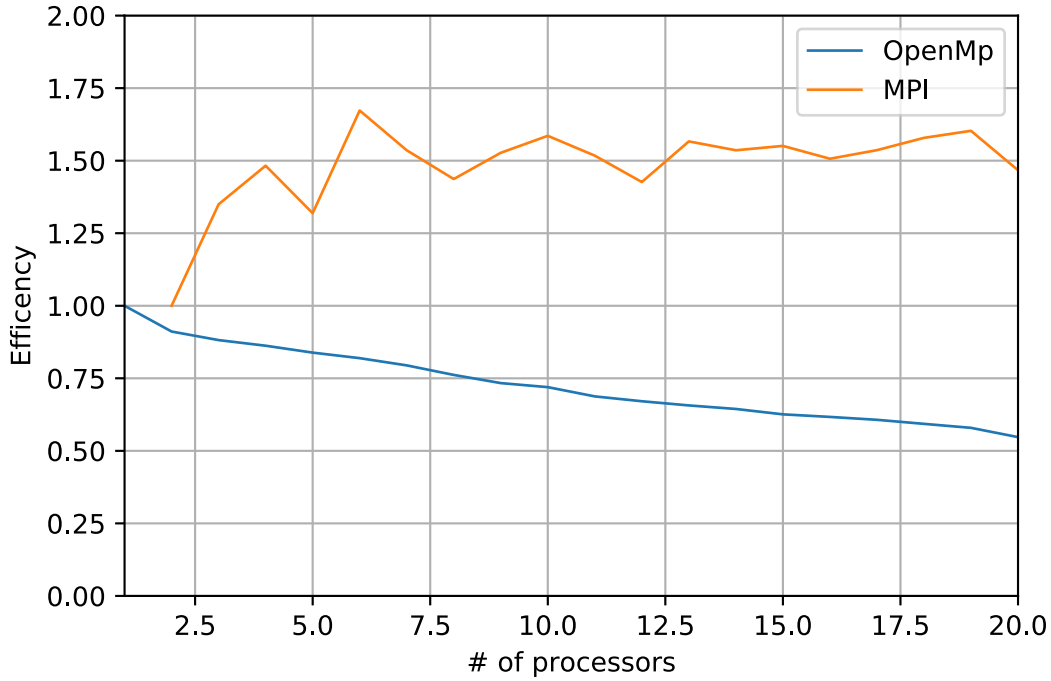
In order to ensure the linearity of the scaling, each of the dimensions of the image was multiplicative with the square root of the number of process(also there are few differences in the MPI version concerning the different compiler and different way to run the executable). The results are plotted below using times recorded in the code:



The efficiency for the weak scalability was calculated the usual way following the formula:

$$ES(P) = \frac{T(1)}{T(P)}$$

The results were:



The decrease in efficiency as well as increasing time in the weak scalability test could be explained considering that, rising the resolution of the image, along side the increase in the total amount of points that have to be evaluated, the points themselves require more steps in order to understand if they are or not in the Mandelbrot Set. That is caused by the fact that we picked the region of the complex plane with the highest density of point in the Mandelbrot Set.