Report

January 20, 2020

1 Exercise 1

1.1 The Mandelbrot set

1.2 Openmp

First of all, for the given problem serial code is written. After that this code is openmpized in a straightforward way. In this implementation of openmp, it is observed (by measuring minimum, maximum and average thread times) that there is work unbalance since some points diverge rapidly as expected. Therefore different ways were searched to get balanced parallel algorithm.

First idea is to use schedule directives with different scheduling types and chunk-sizes to make threads have same amount of workload. Although this implementation is better than straightforward one, it doesn't scale. Second idea which balanced the workload is using collapse directives with schedule directive. By using collapse directive nested for loops collapsed into one large iteration. This implementation gives balanced (minimum, maximum and average threads times very close to each other) and scaling parallel algorithm.

Below plots shows the performance (strong and weak scaling) of algorithm (with collapse and schedule directives) with comparison to straightforward implementation

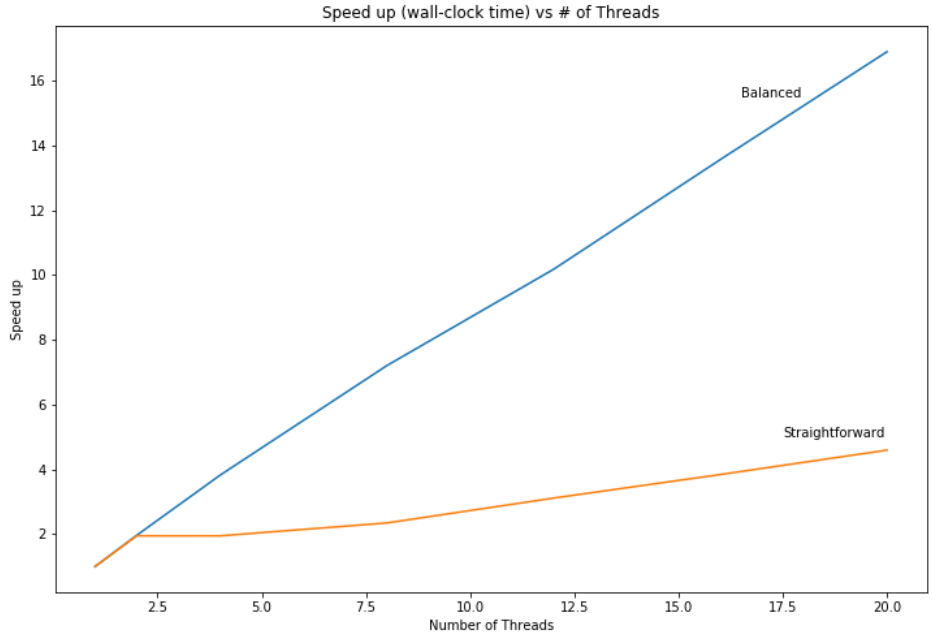
1.2.1 Strong Scaling

Since elapsed time and wall time about same, for scaling tests only one of it used.

For strong scaling test below parameters are used:

$nx = 1024$, $ny = 1024$, $xL = -2$, $yL = -2$, $xR = 2$, $yR = 2$, $Imax = 40000$

Speed Up / # Threads	1	2	4	8	12	16	20
Balanced	1	1.965	3.822	7.206	10.180	13.570	16.902
Straight	1	1.948	1.947	2.345	3.114	3.841	4.595



Below table you can find the comparison between straightforward and balanced version in terms of average, minimum and maximum thread times (just for 4 threads as an example but this situaiton valid for different number of threads).

Average, Minimum and Maximum thread run times (4 threads)		
	Straightforward	Balanced
Average	21.5	21.7
Min	0.16	21.7
Max	43.09	21.7

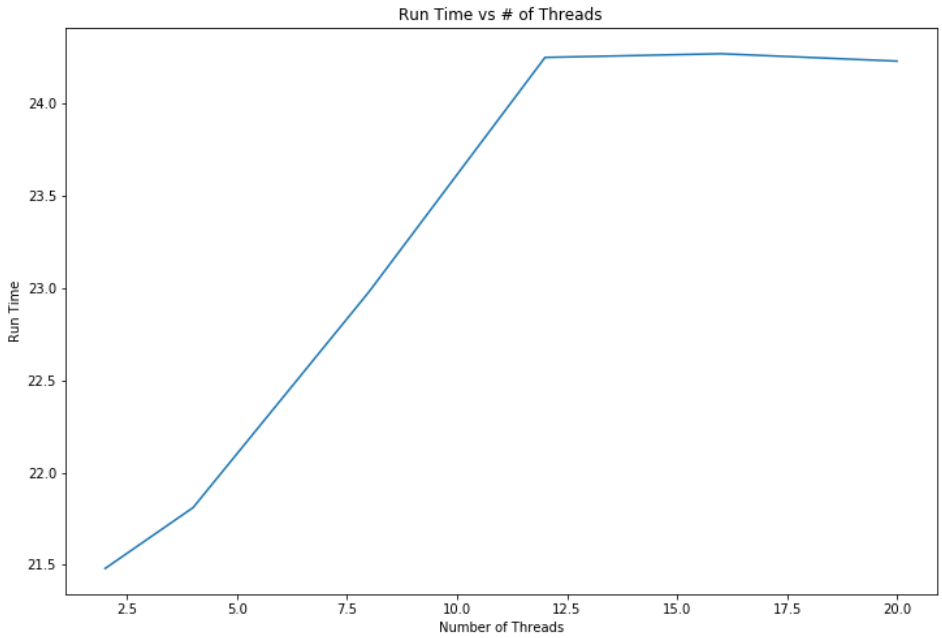
It can be observed that straightforward implementation doesn't scale and unbalanced. However with the directives schedule and dynamic program is balanced and scales good.

1.2.2 Weak Scaling

For weak scaling test below parameters are used:

$nx = 1024$, $ny = 1024$, $xL = -2$, $yL = -2$, $xR = 2$, $yR = 2$, $procs \times 10000$ where $procs = threadnumber$

Elapsed Time (Weak Scaling) / # Threads	2	4	8	12	16	20
Balanced	21.48	21.81	22.98	24.25	24.26	24.26



According to logic of weak scalability when we increase the number of threads with the same amount of work per thread, runtimes supposed to be same. However for this example there are small differences. There might be some room for optimization of parallelization part. But after 12 threads run time became more or less constant. This happens most probably because of Ullyses architecture (two socket - 10 cores each socket).

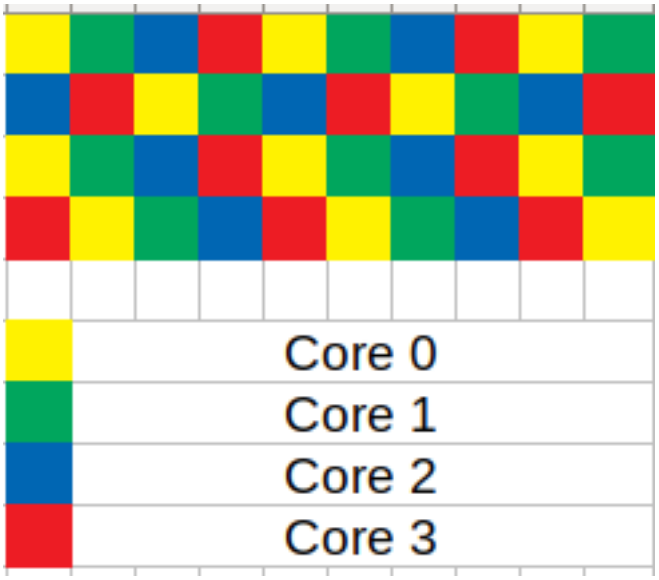
1.3 MPI

1.3.1 Strong Scaling

Balancing MPI process is different than openmp paradigm. Several methods considered by me, for example dividing matrix (image matrix) row by row and sending to available core, or dividing matrix to submatrix and assigning to cores, these methods weren't implemented because I thought that sending row by row may spawn more overhead due to the communication time, submatrix regions can be unbalanced naturally (points in some submatrices may diverge rapidly with high probably on the other hand some of them may not).

The algorithm which is used for the solution of problem, each core takes part of matrix index by index starting from their rank up to end of matrix with the increment of size.

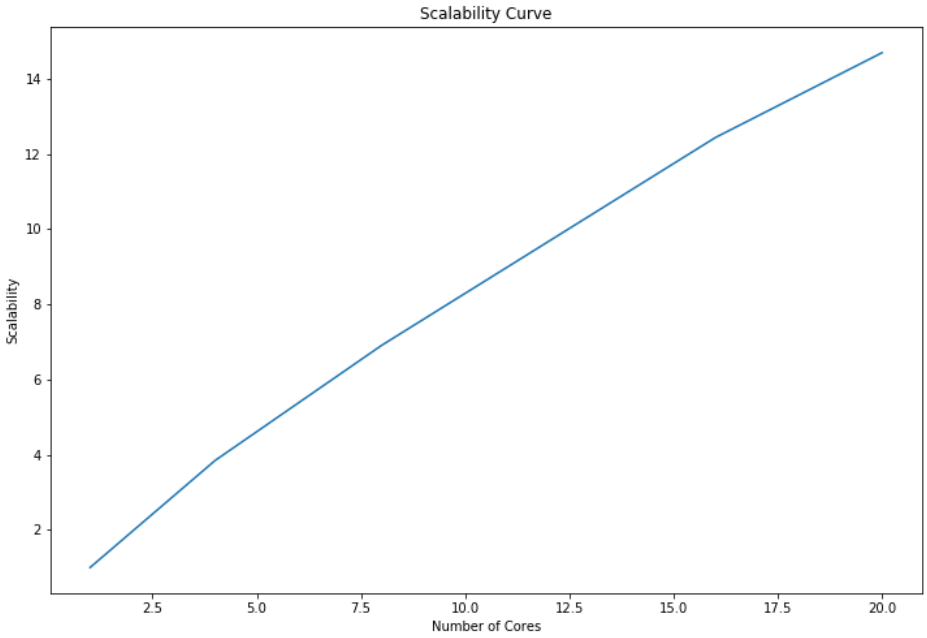
Image below represents the work sharing between cores. 4x10 matrix is divided for 4 cores. By implementing this, master core didn't create big matrix and assign the work for each core. Instead of this, amount of work (called N), coordinates (xL, yL, xR, yR), pixels and iteration (nx, ny, Imax) are broadcasted. Indices of big matrix and mandelbrot results are gathered from each core.



For strong scaling test below parameters are used:

$nx = 1024$, $ny = 1024$, $xL = -2$, $yL = -2$, $xR = 2$, $yR = 2$, $Imax = 3000$

Speed Up/ # Cores	1	2	4	8	16	20
MPI	1	1.95	3.85	6.91	12.43	14.69



According to strong scaling results algorithm scales. However after 5 cores most probably because of communication overhead (2 sockets) it doesn't scale as good as in 1 socket.

To understand the balance of workload, each cores run time printed. It is observed that work load is balanced among the cores. Below table shows the runtimes for 8 cores as an example.

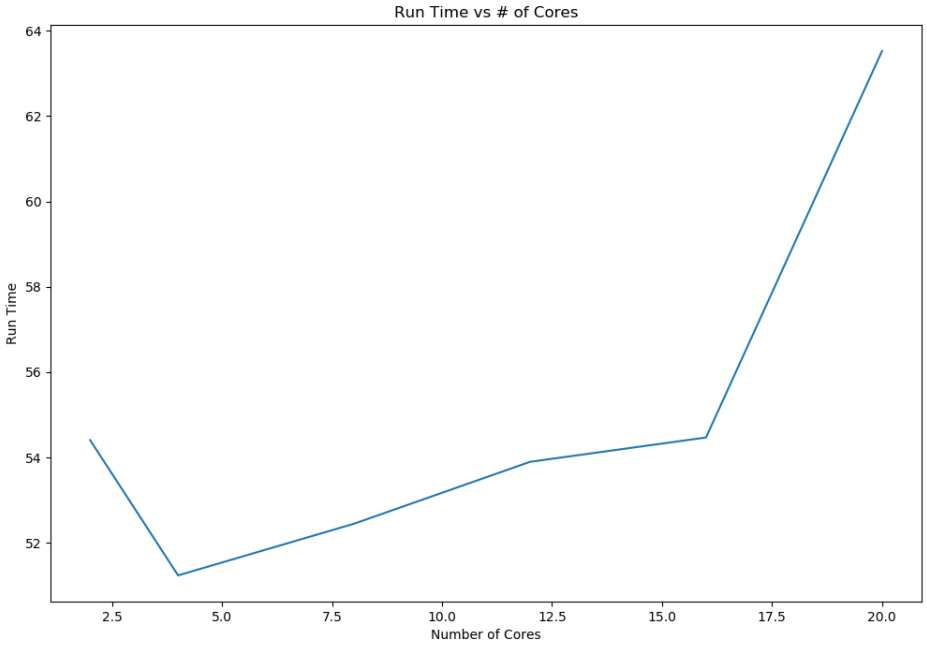
Core ID	0	1	2	3	4	5	6	7
Run Time	38.83	38.82	38.82	38.82	38.82	38.82	38.82	38.87

1.3.2 Weak Scaling

For weak scaling test below parameters are used:

$nx = 1024$, $ny = 1024$, $xL = -2$, $yL = -2$, $xR = 2$, $yR = 2$, $procs \times 500$ where $procs = corenumber$

Elapsed Time (Weak Scaling) / # Core	2	4	8	12	16	20
Run Time	54.41	51.24	52.45	53.90	54.47	63.52



Weak scaling test of mpi code is done several times but every time different results are observed, most probably taken node effected the test. Hence last trial will be evaluated.

Up to 16 cores, run times are more or less same with small differences. However for 20 cores it is bit more than other run times. This may happen because when iteration increase, the data which is transferred increase also, this situation may cause a slow communication between cores.

1.3.3 MPI IO

For mpi io part, plane coordinates (as doubles), pixels and max iteration (as integers) and lastly calculated matrix is written to byte file. To control the file there is another python script with mpi io, which shows the written file and save the image of mandelbrotset. Plane coordinates and pixels and max iteration is written by using collective opperation on the other hand matrix is written by creating contigious block.