FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

CAC assignment 2

Christian Møllgaard

dpj482

15. marts 2016

1 Introduction

The task of this assignment was to convert the given program to run in parallel on the same generated data. To do this the multiprocessing library is used together with the provided shared memory array shmarray class.

2 What have changed

I decided to go with a dynamic model instead of a static model. This choice was due to some hints dropped by other students in class, and I have unfortunately not had the time to test this my self as i went straight for the dynamic model.

2.1 Data generation

In the data generation, I converted all returned Numpy arrays to shmarrays, to make them shared memory. To handle an error caused by pickling the shmarrays, I also made U and V global, so I am able to access them from within the other processes. Apart from the changed to U and V I also made result arrays, where i store the result till all processes are done. This way there was no need for a barrier, or the wait for all processes to complete worked as a barrier.

2.2 The handler

Before any processing can be done, the data needs to be split up into a few pieces. I split the images based on the amount of desired cpus and a constant, that i have not had the time to optimize, so it is hard coded to 2. At this point i only find the block size, and do not actually split the data yet. I add 1 to the block size to handle when the block size and size of data does not match, and could lead to some rows not getting processed.

The handler then starts the workers with a slight modified version of the original RD function. The part where it updated the values and the edges have been taken out, and it uses the global

arrays instead of passed arrays. It gets as argument the block size, and it's index. It then creates views of global arrays, that it uses locally before storing the result in the result array. Because the RD function is dependent on data that is outside of it's current "scope", I increase the size of the local view, so it also contains data from the next process. This way i create some ghost padding, that both the current process, and the next will work with.

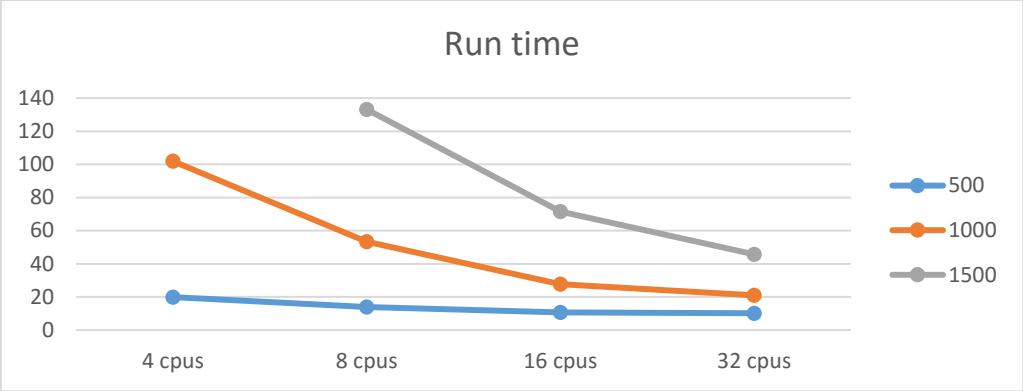
The handler wait for all processes to finish this process before moving on, to copying the result back into the original matrices. This is also done in parallel. If I did not have to have global arrays, i could spare a lot of time here, by just returning a modified version of the result array as U and V, instead of copying the data back into U and V, but the shmarrays does not really work for me in this way.

After all this is done, I fix the edges.

3 Result

I was unable to get all my test data due to bad timing.

To see how the function scales I had planes to find the average with different amount of cpus. There is a clear performance increase, but it does not scale very well with the amount of cpus. Looking at table and figues below, it is easy to see that it does perform faster with more cpus, but it does not scale very well. Some of these bad numbers may be due to the hardcoded cpus*2 workers size.



As shown in the speedup tables below, there is a significant speedup when increasing the cpu count. The speedup drops very quickly, and the speed up graph would probably stagnate quite soon, if tested with a higher amount of cpus.

Tabel 1: Speedup

size	4 cpus	8 cpus	16 cpus	32 cpus
500	4.286675	6.149203	8.071036	8.3977
1000	3.015844	5.762412	11.11917	14.63639
1500	-	4.964749	9.243043	14.46534

Tabel 2: Speedup divided by amount of cpus

size	4 cpus	8 cpus	16 cpus	32 cpus
500	1.071669	0.76865	0.50444	0.262428
1000	0.753961	0.720302	0.694948	0.457387
1500	-	0.620594	0.57769	0.452042

4 conclusion

The program does run faster with more cpus, but there are some clear scaling issues. Some of these, could be handled, by testing more different constants for the amount of workers. Being able to pickle the shmarrays would speed up the process to somewhat enabling the program to just reassign U and V between the original and a copy.