DAT470/DIT065 Assignment 2

Mirco Ghadri
mircog@chalmers.se

Ali Alkhaled
alialk@student.chalmers.se

April 22, 2024

Problem 1

(a) Make the simulation reproducible by implementing the –seed parameter. Every time the same seed is used, the same results should be produced. (2pt)

Answer: See *assignment2_problem1.py*.

(b) Look at the function simulate. Describe in your report, which parts of the function need to be executed serially, and which could be parallelized. (2pt)

Answer: The function simulate has a for loop which calls the play_blackjack() function n times:

```
for i in range(n):
    results[i] = play_blackjack(basic_strategy)
```

This for loop can be parallelized, since each call to play_blackjack() is independent and can thus run as a separate process. All other parts of the simulate function need to be executed serially.

(c) Using sufficiently large (but constant-sized) input, measure the fraction of time spent in the sequential and parallelizable part of the program. Report the size of input, the fraction and absolute time spent in sequential part of the program, the fraction and absolute time spent in the parallelizable part, and the total running time. (2 pt)

Answer: For the command `/opt/local/bin/run_job.sh -s assignment2_problem1.py 500000 -- --seed 5`, where n is 500000 and seed is set to 5, we get the following table of results

Component	Time (seconds)	Percentage of Total Running Time
Sequential Part	0.00041413307189941406	0.0005636226256630309%
Parallel Part	73.47659921646118	99.99943637737434%
Total Running Time	73.47701334953308	100%

Table 1: Simulation Timings

(d) Using Amdahl's law, what is the upper bound on speedup that can be achieved?

(1 pt)

Answer: Amdahl's Law states that

$$\text{Speedup} = \frac{1}{(1-P) + \frac{P}{s}} \tag{1}$$

In the equation, P stands for the fraction of the programs total running time which comes from code that can be parallelized. In our case, we saw that 99.999% of the total running time came from a parallelizable component in the code, more specifically, the for loop. This means that P = 0.99999 in our case. S stands for the speedup of the parallelizable part of the code, i.e. how much we speed it up. This could be arbitrarily large, since we can use infinitely many processes. If we let the speedup of this part go towards infinity, we get the following:

$$\text{Speedup} = \lim_{s \to \infty} \frac{1}{(1-P) + \frac{P}{s}} = \frac{1}{1-P} \tag{2}$$

If we plug in the value for P, we get that the theoretical speedup upper bound is equal to

$$\text{Speedup} = \frac{1}{1-P} = \frac{1}{1-0.99999} = \frac{1}{0.00001} = 100000 \tag{3}$$

This theoretical speedup factor of 100000 happens if we speed up the parallel part of the code so much that it occurs instantly, but the remaining sequential part of the code will still have to be executed regularly.

(e) Use the multiprocessing module to parallelize the simulate function, that is, implement the -w command-line parameter. (4 pt)

Answer: See *assignment2_problem1.py*.

(f) Evaluate your parallel implementation on the cluster empirically. Produce a plot that contains the following:

- *The observed speedup as a function of the number of CPU cores*

  To get the observed speedup, we use the formula

  $$S(n) = \frac{t_1}{t(n)} \tag{4}$$

  Here, t_1 stands for the time it takes to run the program with 1 CPU core, while $t(n)$ is the time it takes to run the program with n CPU cores. This gives us the speedup as a function of the number of CPU cores. We know that the time it took to run the program with 1 CPU core was ≈ 73 seconds. For this experiment, we will run the program on Bayes with the number of workers ranging from 1 to 30. The reason we will run it on Bayes is because Bayes will automatically allocate the number of CPU cores that are specified in workers. This allows us to create a for loop in our program and get all the results in 1 run.

n	t_n (seconds)	$\frac{t_1}{t_n}$
1	73.82490348815918	1
2	36.794870376586914	2.003452528
3	24.578934907913208	3.002993386
4	18.382068395614624	4.013798162
5	14.766664028167725	5.003992958
6	12.326732397079468	5.986118261
7	10.810151815414429	6.826465925
8	9.713636636734009	7.595810636
9	8.774367094039917	8.400399755
10	7.533107280731201	9.801365398
11	7.0179502964019775	10.518032986
12	6.495280027389526	11.36301487
13	6.082304239273071	12.147398871
14	5.430260181427002	13.579458754
15	5.02759051322937	14.656716289
16	4.765574216842651	15.488202027
17	4.443081855773926	16.616073038
18	4.2226996421813965	17.458051657
19	4.0043699741363525	18.420504283
20	3.8259005546569824	19.300800664
21	3.6390202045440674	20.282898564
22	3.4268763065338135	21.549529517
23	3.3624017238616943	21.930295849
24	3.1849663257598877	23.169485922
25	3.0709052085876465	24.02722599
26	2.9869656562805176	24.640636781
27	2.887920379638672	25.556925449
28	2.8504865169525146	25.866470495
29	2.695115804672241	27.37573493
30	2.650547742843628	27.863965032

Table 2: Speedup factor, i.e. ratio of t_1 to t_n

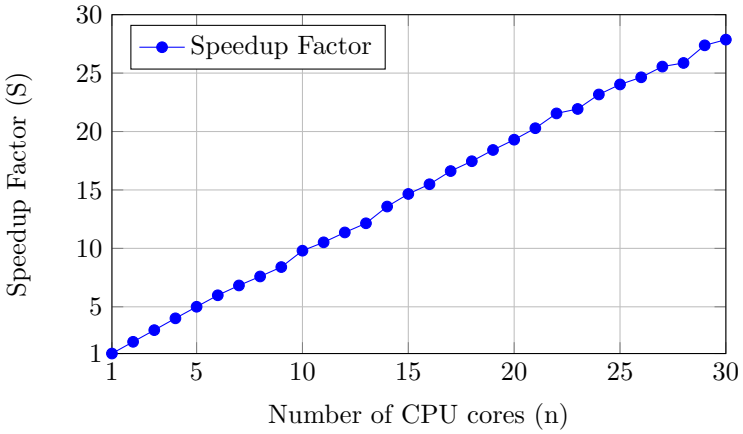

Figure 1: Speedup factor S as a function of n

- *The theoretical speedup as predicted by Amdahl's law as a function of the number of CPU cores*

  The theoretical speedup can be written as

$$\text{Speedup}(s) = \frac{1}{(1-P) + \frac{P}{s}} \tag{5}$$

  Small s is the speedup factor of the parallel part of the program. We know that this speedup of this part depends on the number of CPU cores, so we can rewrite Amdahl's law as a function of n(the number of CPU cores) such that

$$\text{Speedup}(n) = \frac{1}{(1-P) + \frac{P}{s(n)}} \tag{6}$$

  $s(n)$ does not tell us more than s, so we need to substitute it with the actual effect that n has on the speedup. In a perfect scenario, the speedup factor s is directly proportional to the number of CPU cores, i.e. $s(n) = n$. The reason is that if we use n CPU's, we can not achieve a higher speedup factor of the parallel part than n. The reason is that in a perfect scenario the running time is divided evenly between each process/CPU core and we get that speedup factor is $\frac{1}{\frac{1}{n}} = n$. Hence, we can write the theoretical speedup S as a function of the number of CPU cores as

$$\text{Speedup}(n) = \frac{1}{(1-P) + \frac{P}{n}} \tag{7}$$

  To make the plot, we let n go from 1 to 30, as in the previous plot. From previous results, we know that $P = 0.99999$.

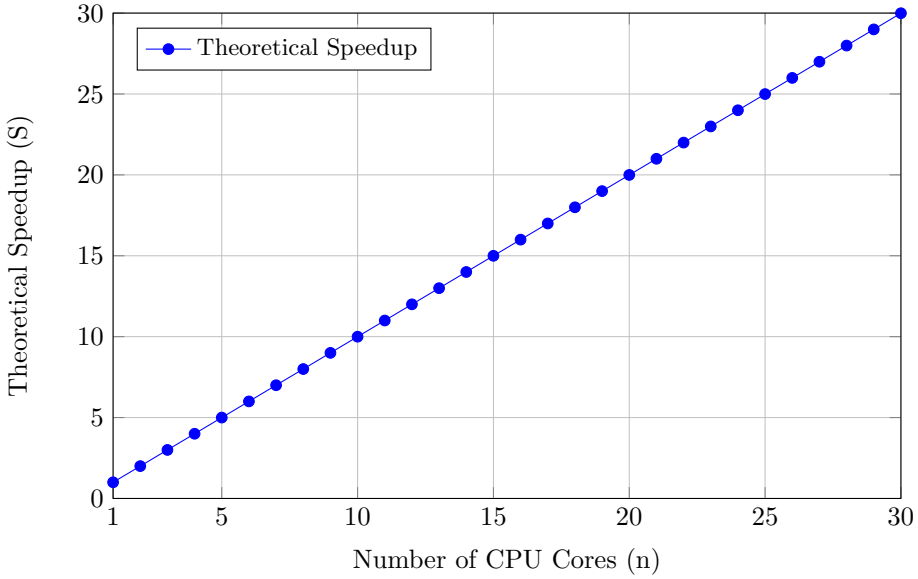

Figure 2: Theoretical Speedup S as a function of n

We can see that the 2 plots(Figure 1 and Figure 2) are almost identical. This means that the speedup S that we got empirically corresponds well with the theoretical speedup factor S. Hence, we can say that the theory of Amdahl's law is valid.

- *The upper bound on the speedup as predicted by Amdahl's law*

  -

$$\text{Speedup} = \lim_{n\to\infty} \frac{1}{(1-P) + \frac{P}{s(n)}} = \frac{1}{1-P} \tag{8}$$

(3 pt)

Problem 2

(a) Change the queues into multiprocessing queues, and adjust the structure of the program such that you parallelize the program by creating a desired number of processes. (4 pts)

Answer: See *assignment2_problem2.py*.

(b) Furthermore, change the way the dataset is handled: if you pass the NumPy array as an argument to the process, it is copied. We want to avoid this, so use a multiprocessing array to share memory between the processes; you must not make a copy for the processes, but the processes must access the original data. (2 pts)

Answer: See *assignment2_problem2.py*.

(c) Like in Problem 1, determine the fraction of sequential and parallelizable code in the main function. Report the fractions and the maximal speedup as predicted by Amdahl's law in your report. (2 pts)

When running the query python ufo.py < queries_large.txt we got the following running time for the sequential and parallelizable part of the code:

	Running Time (s)	**Fraction of Total Running Time (%)**
Sequential Part	0.177	1.73
Parallelizable Part	10.101	98.27

Table 3: Running Time and Fraction of Total Running Time

The maximal speedup predicted by Amdahl's law is given by the equation:

$$S = \frac{1}{1-P} \tag{9}$$

We have that P is equal to 0.9827, so we get:

$$S = \frac{1}{1-0.9827} = \frac{1}{0.0173} \approx 57.80 \tag{10}$$

This means that if we manage to execute the parallelizable part of the code instantly by means of multiprocessing, the speedup factor of the running time of the entire program will be equal to ≈ 58.

(d) Evaluate the speedup of the program empirically on the cluster, and produce a similar plot that shows the observed speedup, the speedup predicted by Amdahl's law, and the theoretical upper bound for the speedup. (2 pts)

Number of Workers/CPU cores	**Total Running Time (Seconds)**
1	10.329352855682373
2	5.2570672035217285
3	3.5630152225494385
4	2.7485134601593018
8	1.5172829627990723
16	0.9042048454284668
32	0.578439474105835

Table 4: Total running time for different numbers of workers

To produce a plot of the observed speedup, we calculate it as

$$S = \frac{t_1}{t_n} \tag{11}$$

where t_1 is equal to the time it takes to run the program with 1 CPU core(worker) and t_n is the time it takes to run the program with n CPU cores. We get the following plot.

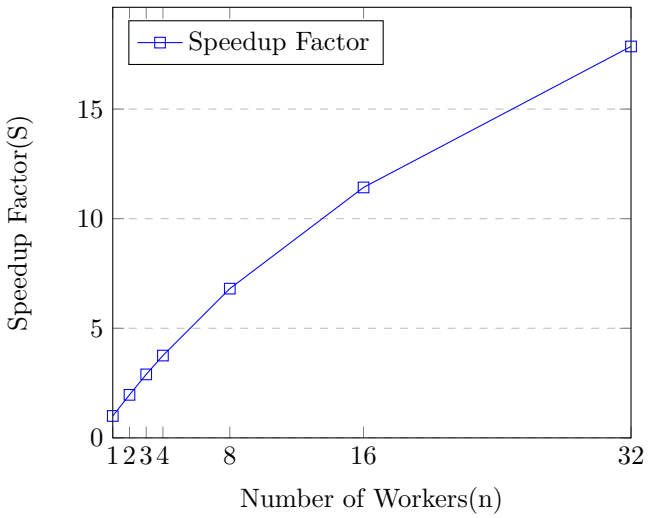

Figure 3: Speedup factor as a function of number of workers(CPU cores)

To calculate the theoretical speedup predicted by Amdahl's law, we assume the speedup of the parallel part(small s) is directly proportional to the number of workers/CPU cores(n). We know that proportion of parallelizable code P is

equal to 0.9827. We get that:

$$S(n) = \frac{1}{(1-P) + \frac{P}{n}} = \frac{1}{(1-0.9827) + \frac{0.9827}{n}} = \frac{1}{(0.0173) + \frac{0.9827}{n}} \tag{12}$$

To make the plot, we calculate the value of $S(n)$ for n going from 1 to 32. We get the following plot:

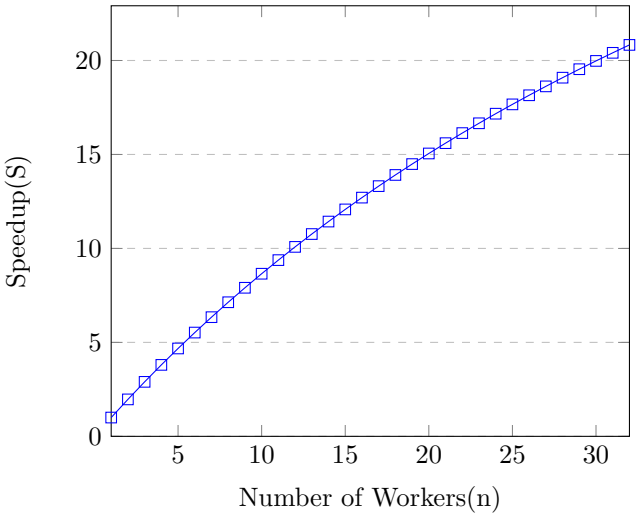

Figure 4: Theoretical Speedup as a function of number of workers

We can see that the theoretical speedup is well in accordance with the empirical speedup factor that we got. This tells us that Amdahl's law is working.