David Pérez Carrasco NIA:241375
Eduard Rodríguez López NIA:242710

Universitat Pompeu Fabra Barcelona

Lab1

1. Dot product (50%)

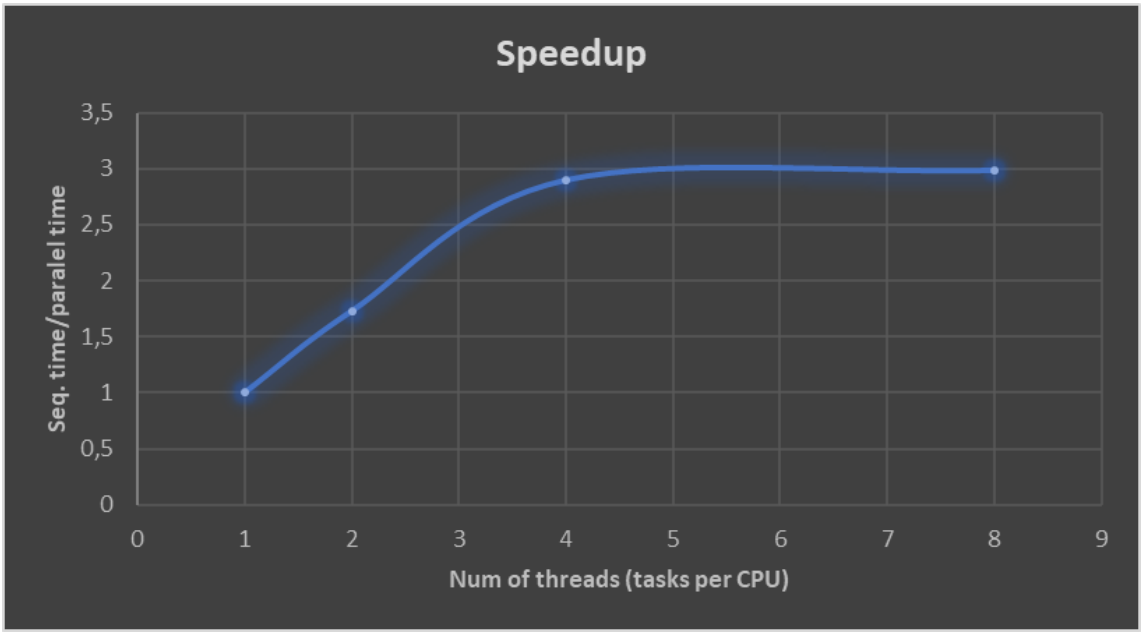
Exercise 2

2. Submit a job executing the code with a vector size of 100M integers and obtain results for 1, 2, 4 and 8 threads (you can use the sample slurm script dotp job.cmd as a base for your job files). How does it scale? Plot the strong speedup.

First of all, we have executed the code 10 times with each number of threads, in order to compute an accurate average to determine a realistic and objective time value. Moreover, we have rejected all the values that seemed outliers because as we wanted the most objective representation of our results we know that sometimes the cluster returns a slightly higher time value because of the high traffic, which happened to us, for example with several values that overweight the mean by 4 or 5 seconds. So we used the 10 realistic values to compute the average, as we can see in the following table where we collect all the results.

Tries	1 Thread	2 Threads	4 Threads	8 Threads
Try 1	0,060908	0,035949	0,022251	0,021099
Try 2	0,063993	0,035991	0,021812	0,021139
Try 3	0,063551	0,036404	0,02227	0,020924
Try 4	0,062631	0,037159	0,021958	0,021159
Try 5	0,063234	0,036642	0,021426	0,021158
Try 6	0,06165	0,036298	0,021946	0,021217
Try 7	0,065548	0,036708	0,021603	0,02124
Try 8	0,062614	0,036532	0,021571	0,021024
Try 9	0,062957	0,036815	0,021373	0,021166
Try 10	0,063357	0,036209	0,021345	0,021091
Average	**0,0630443**	**0,0364707**	**0,0217555**	**0,0211217**

After that, we have computed the strong speedup, by dividing the sequential time (1 thread) by the parallel time corresponding to each num of threads, which creates the following plot.



This plot shows that our results do not seem to be wrong, as we can see a increasing similar to the logarithmic one, with a each time less increasing tendency but never decreasing, in a way we can appreciate that the difference between the sequential and the parallel is notorious, but since the 4 threads the difference is almost impossible to appreciate.

3. Using the time spent in the dot product function, compute throughput in GBytes/s and GFLOP/s for each configuration (present them in a table).

Once we have the different times, we just need to multiply the number of floats involved, in this case 100M by the float's size in bytes, which is 4 bytes, and divide all of that by the corresponding time. About the GFLOPS, we need to count all the operations, which is 200M, as there are 2 operations (multiplication and sum) for each value at the vector, and divide all of that by the corresponding time of each case. This gives us a table which shows us the correlation between the GBytes/s and GFLOPS/s, as the seconds are exactly the half of the firsts, which is reasonable as there are 2 operations, the half of the float's size. We have to recall that the time is in seconds.

Nthreads	Time	Gbytes/s	GFLOPS/s	Speedup
1	0,0630443	6,34474489	3,17237244	1
2	0,0364707	10,9677083	5,48385416	1,72862873
4	0,0217555	18,3861552	9,19307761	2,89785571
8	0,0211217	18,9378696	9,46893479	2,98481183

We have to make a subsection here about the 1 thread time. We actually got 0,072 seconds of the average time when executing with one thread, which surprised us because the average time of the sequential execution was just 0,063s, so we realized that it could be because the overhead of creating just one thread, which actually does not create any parallelization but lowers the execution because of this overhead so the time is a little bit slower. Knowing that, we decided to take the sequential times as the 1 thread execution time, because they could objectively be the same, but subtracting the overhead we obtain the sequential time which is obtained at the exercise 1.1.

Exercise 3

2. Submit a job executing the code with a vector size of 100M integers and obtain results for 1, 2, 4 and 8 cores with the option of vectorization inside the parallel region. How does it scale? Add a table to your report comparing the values with the simple parallelization.

At first, we can see at the following table the values of the times and the corresponding GBytes/s, GFLOPS/s and even the speedup values of the vectorized executions.

NThreads	Time	Gbytes/s	GFLOPS/s	Speedup
1	0,0313247	12,7694758	6,38473792	1
2	0,0216162	18,50464	9,25232002	1,44913074
4	0,020967	19,0775981	9,53879907	1,4940001
8	0,0209419	19,1004637	9,55023183	1,49579074

Moreover, we have created a table that directly compares just the time values of both the vectorized and the simple parallelization for each number of threads.

NThreads	Time Simple Parallelization	Time Vectorized Parallelization
1	0,0630443	0,0313247
2	0,0364707	0,0216162
4	0,0217555	0,020967
8	0,0211217	0,0209419

At this table we can see that the time of the simple parallelization is practically the double of the vectorized one, and the difference with 2 threads is also really considerable but when reaching the 4 threads it starts being almost negligible. Anyway, this shows that there is actually an appreciable difference between the vectorized and the simple parallelization with a small number of threads, but for higher ones it may be insignificant.

3. Change the flag -O2 in the Makefile for -O3. Submit a job executing the code with a vector size of 100M integers and obtain results for 1, 2, 4 and 8 cores with the options of Exercise 2 and Exercise 3. Compare the results. What does -O3 do?

The flags -O2 and -O3 are a GCC optimization flag, whose goal is to improve the overall performance of the code. With -O2 apply almost all optimization, therefore, the compiling time will increase. On the other hand, -O3 is the highest optimization level, so consequently it'll raise even more compiling time, which is appreciable both at Exercise 2 and 3, as we can see in the following comparing tables.

NThreads	Time Simple Parallelization -O2	Time Simple Parallelization -O3
1	0,0630443	0,0312604
2	0,0364707	0,02159
4	0,0217555	0,0208767
8	0,0211217	0,0208687

NThreads	Time Vectorized Parallelization -O2	Time Vectorized Parallelization -O3
1	0,0313247	0,0311887
2	0,0216162	0,0215697
4	0,020967	0,0208702
8	0,0209419	0,0208616

As we can see, the -O3 effect is noticeable mainly at the simple parallelization. We appreciate almost half of the time at simple parallelization with 1 thread, and a bit less the same with 2 threads but the difference decreases as the number of threads increases.

On the other hand, at the vectorized parallelization we almost do not appreciate any difference, maybe one or two ms in each case, and like the simple parallelization this difference increases as the number of threads increases.

This leads us to make some conclusions. At first, it is obvious that -O3 is more optimal than -O2, as the theory claims, but this optimization could be reached otherwise by vectorization, as we can see almost the same results when the parallelization is vectorized and when it is simple but with -O3 flag, and the flag changing does not affect practically to the vectorized ones.

Moreover, in this exercise we have noticed more frequently something that we said before, the overhead and the time error due to the cluster traffic, which we have noticed is higher when we increase the number of threads. When we tested the 8 threads parallelization we obtained practically the same result as with 4 threads, but we had to discard a lot more of outliers, as we obtained 0,05s or 0,04s values from time to time, which happened with less frequency when we tested less number of threads. Furthermore, the final value could be even a little greater than the 4 threads one, which shows us that when we reach a certain number of threads it starts not being as optimal as it does not actually increase the efficiency of the parallelization but it remains increasing the overhead error time, so there has to be a value when is not worth to keep increasing the number of threads.

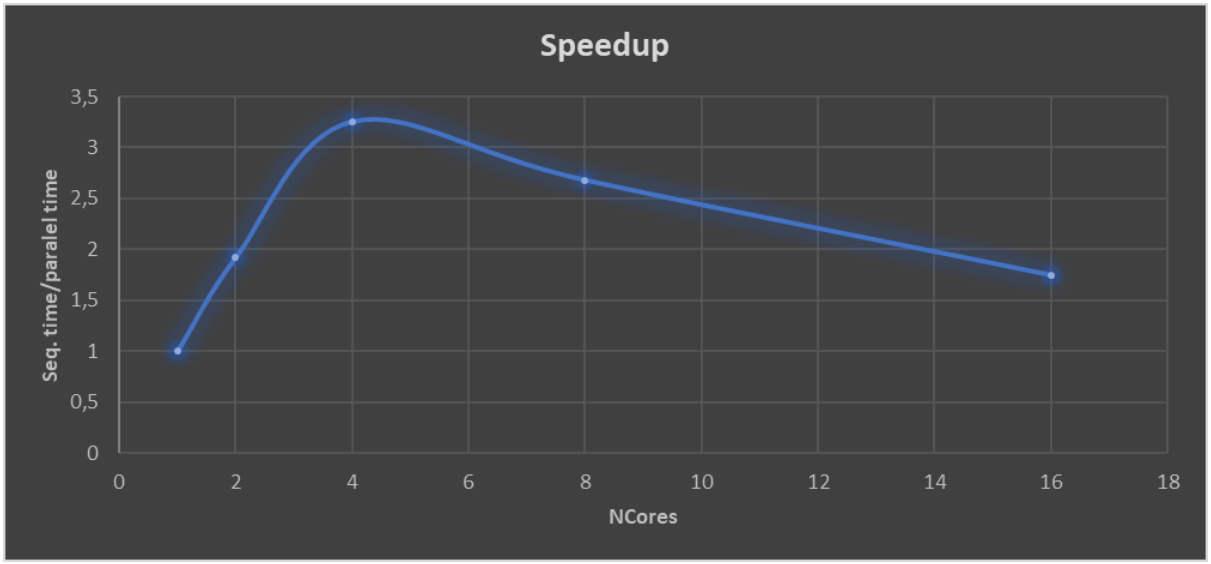
2. Quicksort (40%)

Exercise 4

2. Submit a job executing the code with a vector size of 1M doubles and obtain results for 1, 2, 4, 8 and 16 cores (you can use the sample job file dotp sort.cmd as a base for your job files). How does it scale? Plot the speedup.

In this exercise we have done the same as in the previous, we have created a table in which we noted 10 different attempts for each NCores, and we have computed the average of each one. Then, we have computed the Strong Speedup with the previous formula and we have obtained the following table and plot.

Ncores	Time (s)	Speedup
1	0,0925087	1
2	0,0481359	1,921823421
4	0,0283953	3,257887749
8	0,0344976	2,68159814
16	0,0529409	1,747395681



In this plot is even more appreciable the phenomenon observed before, that as NCores increases the Speedup increases until reached some point, in this case NCores=4, when it is no more efficient and the speedup starts decreasing due to the overhead effect.

3. Submit a job executing the code with a vector size of 100M doubles and obtain results for 1, 2, 4, 8 and 16 cores (you can use the sample job file dotp sort.cmd as a base for your job files). Does it scale better or worse than before? Why?

Ncores	Time 1M (s)	Time 100M (s)	Speedup 1M	Speedup 100M
1	0,0925087	11,5042107	1	1
2	0,0481359	5,8111458	1,92182342	1,979680272
4	0,0283953	3,0557992	3,25788775	3,764714219
8	0,0344976	2,07129	2,68159814	5,554128442
16	0,0529409	2,26585	1,74739568	5,077216365

As we can see, the time considerably increases when dealing with 100M doubles, especially with the higher numbers of threads, and thereby the speedup does the same. This makes complete sense because now we have more doubles to deal with, so we'll need more cores to do that efficiently. This is the reason why with 2 or 4 threads the speedup is a little bit similar but when reaching the 8 and 16 threads, when the 1M speedup starts to decrease, the 100M one just starts increasing more, it is reasonable that a 100M double array would be most efficiently dealt with 8 Cores than a 1M double array would be.

Exercise 5

1. Add the clause if(hi − lo ≥ (X)) to the pragmas of the recursive calls to Quicksort and repeat the tests in Exercises 4.2 and 4.3. Try different values for (X) (5, 10 and 1000) and different sizes for the problem. What do you observe? What does it do? Add a table to your report with the different values you have obtained.

After increasing the value X, it is quite evident that the running time is decreased, which makes sense since the divide and conquer process will finish before.

1M:

Ncores	Time X=5 (s)	Time X=10 (s)	Time X=1000 (s)
1	0,090985	0,092856	0,091456248
2	0,049854	0,04812	0,046987
4	0,031456	0,03957654	0,0258997
8	0,0310232	0,028031	0,01498679
16	0,045321	0,03297868	0,010986797

100M:

Ncores	Time X=5 (s)	Time X=10 (s)	Time X=1000 (s)
1	11,343	11,432053	11,40412321
2	5,75013402	5,73901231	5,725913421
4	3,05097808	3,08746	3,045214321
8	1,92856779	1,89523421	1,70634534
16	1,8063454	1,81654634	1,205634634

3. N-Queens (10%)

Exercise 6

1. Use gprof to get a profile of the code. What is the function that takes more time? You may need to modify the Makefile and the job script.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.86      7.28     7.28   380575     0.02     0.02  Fitness
  0.14      7.29     0.01       73     0.14     0.14  copy_pop
  0.00      7.29     0.00     7798     0.00     0.00  Tournament
  0.00      7.29     0.00     7300     0.00     0.98  Mutation
  0.00      7.29     0.00     3650     0.00     0.00  OnePointCrossover
  0.00      7.29     0.00       74     0.00     1.91  population_fitness
  0.00      7.29     0.00        1     0.00     0.00  create_population
```

From the following table, it is self-evident that the Fitness function is much more time-consuming than the rest. In particular, it amounts to 99.86 % of the total time of the whole task.

2. Parallelize the most time consuming function and run the code requesting 4 cores. Is the new code faster? Note: the result should always be found with 73 iterations.

We parallelize using a reduction to the variable attack and making k be a private variable to avoid its misuse.