Task 4: Exploiting the potential of modern architectures

Group 13
Pablo Almarza Marques
Jorge González Gómez

Exercise 0: Information about the system topology.

We will be using our own computer due to likely probable long queue times and slowdowns in the cluster labomat. We were going to use either Digital Ocean's trial $100 or Azure's free students credit, but in the last practice we didn't even bother with the cluster, we just went straight ahead for Azure's services and we found out times were highly inconsistent and had weird downward spikes.

The frequency of the processors available in Jorge's computer is 3504 MHz, while the number of cores is 4. The hyperthreading option is active because the number of siblings is 4. The processor is an Intel® Core™ i5-7600 CPU @ 3.50GHz, with a total cache size of 6144 KB.

Exercise 1: Basic OpenMP programs.

1.1.
It is possible to run more threads than cores since OpenMP allows parallel threads. It makes sense if we are executing a program with multiple tasks, but they will likely wait for other threads to stop before the OS can assign a core to them.

1.2
The number of threads in the lab should be the number of cores that the computer has, since it does not have multithreading, in the cluster, we can use up to 8 threads (4 cores with hyperthreading enabled), the same as on our own computer.

1.3
The environment variable OMP_NUM_THREADS has the lowest priority (will be used as a last resort), while the function `omp_set_num_threads` has more priority over the environment variable, but we discovered that the sentence inside of the "pragma" has the highest priority and this value will be taken over every other one
1.4
If we declare a private variable, only the thread where it is declared will see it while the rest will have it as a non initialized variable.

1.5
When the parallel region starts executing, the private variable takes the value 0.

1.6
At the end of the parallel region, the value is computed so the program gets the value of initialization.

1.7
When the variable is public, every change made is seen by all the threads, and so everyone works with the value modified. When the parallel region ends, the value of the variable will be the one when the last modification was made.

Exercise 2: Parallelize the dot product.

2.1.

By running the serial version, we see this program just multiplies two vectors value by value (let i be an incremental integer number and N the size of the vectors, it would do a sum of A[i] * B[i] from i = 0 up to i = N), and then just keeps summing these values onto a variable. These vectors in particular are filled with 1, so the result of the multiplication is just 1 every single time, which means the sum is just sum = sum + 1, which in turn means the result will always be N. By default, it's set to the `#define`'d variable M, so the result will always be 100000. Replacing this M value by any other number just changes the result to this new M number.

2.2.

The result isn't correct due to race conditions. This means different threads will keep their internal "sum" variable which isn't truly global, so the different threads sum their numbers and then replace the actual global sum when they end being executed, which means the last thread will just output their result and overwrite every other thread's result.

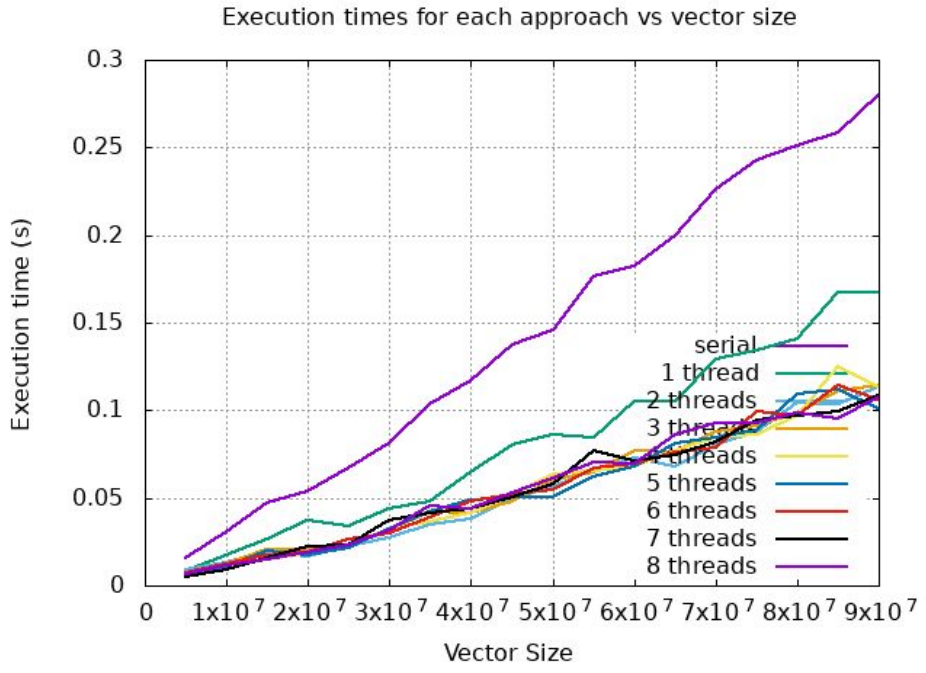
2.3.

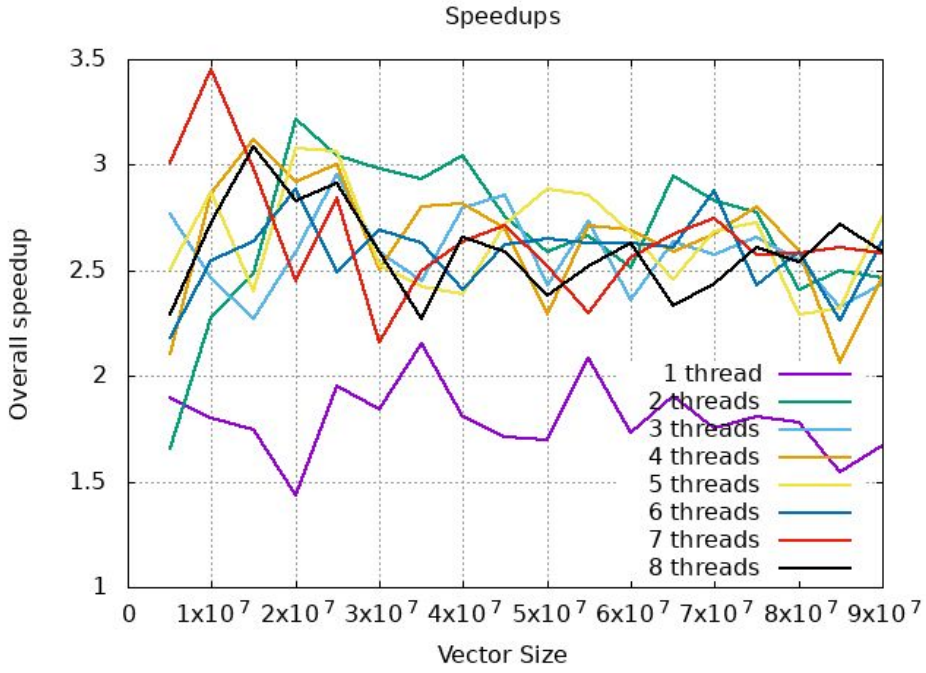
If we replace our old pragma with "`#pragma omp critical`" we can get the correct value, we assume this is due to the nature of critical code and means the other threads are waiting as in sempahors for the currently executing thread to go (which means this would be the same as a serial program, just with added overheads). If we place the "`omp atomic`" pragma on top of the "sum" operation, it would also solve this issue since the omp atomic pragma prevents race conditions without an excessive need of locks and semaphors.
The chosen option is critical just for one simple reason: execution time. Through running this program, we discovered that the "`omp atomic`" approach takes around 4 to 5 times more to execute than our "`omp critical`" approach.

2.4.

Now that we added "`#pragma omp parallel for reduction(+:sum)`", we see that the time is inconsistent after running it some times, since some of the times it runs as poorly as the “atomic” approach, but other times it’s slightly better than the “serial” approach where we just execute it in one thread. This is due to the size of the vector: since it is pretty much a relatively small vector, doing this work in parallel doesn’t really provide a significant difference

2.5.





We can see a lot of spikes due to us working in real time in Visual Studio Code Live Share, having an ongoing Discord call; but we can still observe that the bigger the number of threads, the bigger the speedup. Weirdly enough, we see how the speedup decreases as the vector size increases; this is maybe due to a lower usage of CPU resources on Jorge's computer.

In our specific cases (in this graphs), it always compensates using multiple threads, but for values less than 80000 approximately we can see how it improves in parallel, so we would just have to add an if (M>80000) in the omp pragma.
This is because for values that are small, the overhead of creating new threads and letting the O.S. planifier decide what to run first does impact performance more than it would help in this case.

Exercise 3: Parallelize the dot product.

Execution time (s): Size 1000.

Version \ #Threads	1	2	3	4
Serie	10.424923	10.424923	10.424923	10.424923
Parallel - loop1	14.340552	8.486952	8.558855	10.585156
Parallel - loop2	11.744158	8.009758	6.023851	6.388840
Parallel - loop3	10.804795	5.883614	3.973516	4.055701

Speedup: Size 1000.
Speedup = Serie/Loop

Version \ #Threads	1	2	3	4
Serie	1	1	1	1
Parallel - loop1	0,726954	1,228347	1,218027	0,984862
Parallel - loop2	0,887668	1,301527	1,730607	1,631739
Parallel - loop3	0,964842	1,771857	2,623601	2,570436

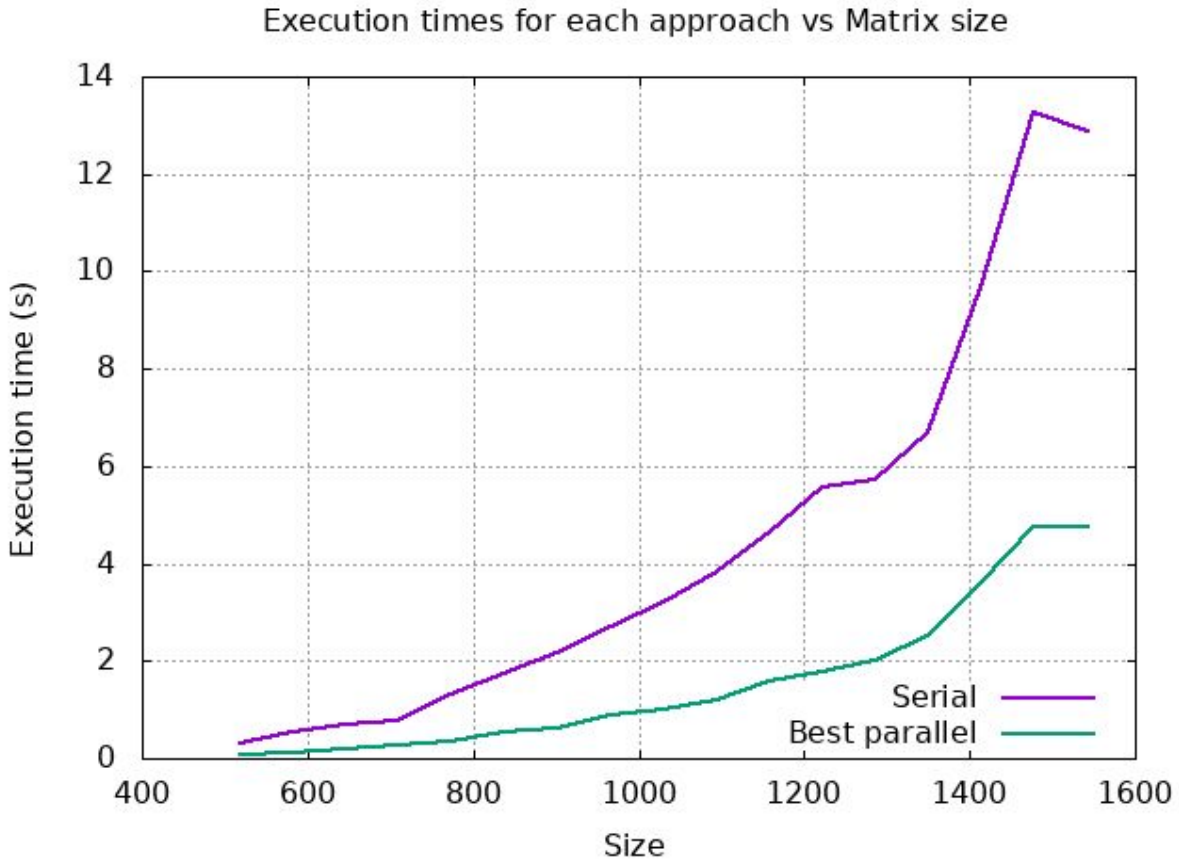
3.1
The worst version is the one that the parallel loop is on the most innerloop, this is because the parallel region is only one loop, so it will need to prepare the threads and the parallel area, wasting more time.

The best version is the one that the parallel loop is on the most outer loop, this is because this is the best optimization since it parallelizes all the work and also the preparation of the threads is only done one time.

3.2

Comparing the results, it is obvious that the coarse-grained parallelization is a preference for big algorithms because the tasks will be parallelized therefore getting a really optimal time.

3.3



In the last graphic we were executing things simultaneously like in the other graphics so it is adulterated, but the average speed up is around 2.5 to 3.5 seconds.