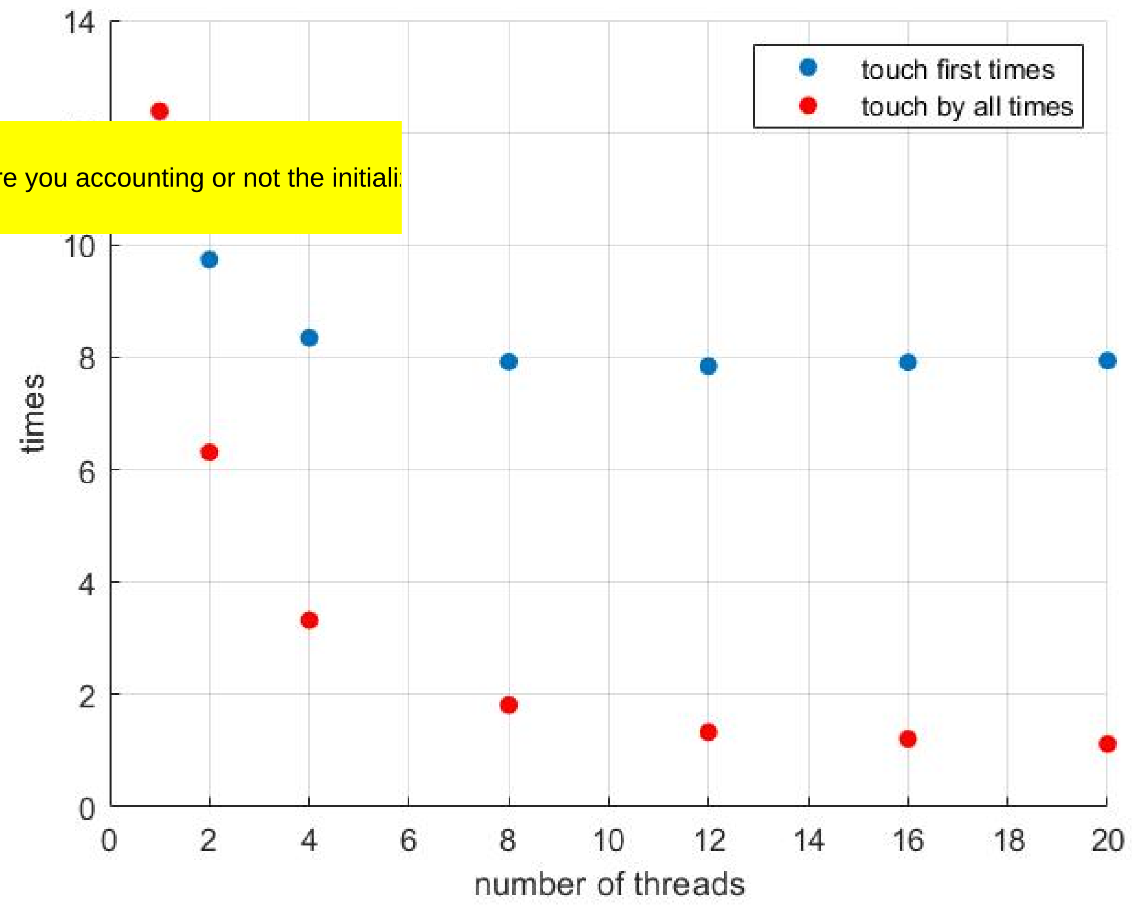
# Second Assignment Report

## Exercise 0

### Comparison between touch first and touch by all approaches in the parallel sum of N numbers.

First of all, the testing was done with N=2.000.000.000. Both approaches, as one would expect, are better in term of execution times when compared to the serial algorithm, which took approximately 12.4 seconds. Looking at the plot below, we see that the touch first approach takes two to four times more than the touch by all one.

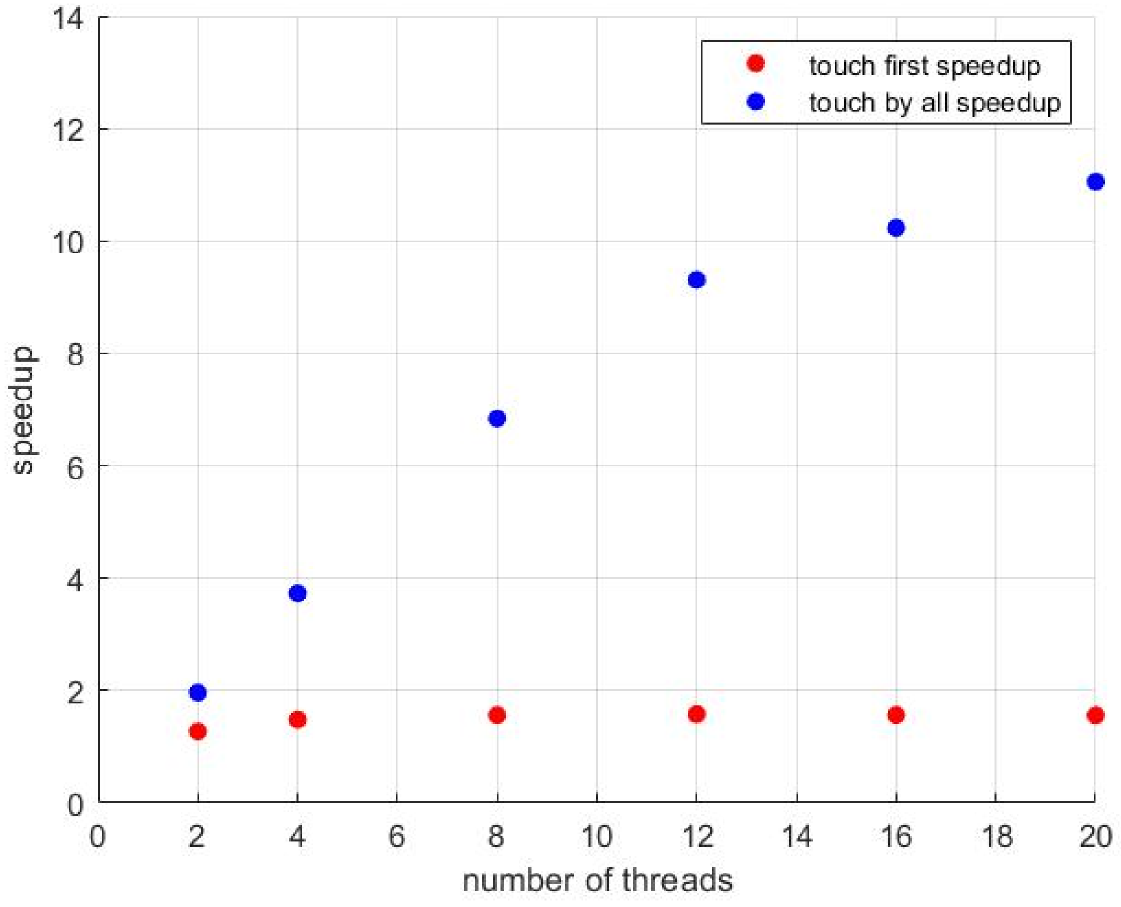
### Execution times for N=2.000.000.000



[*]which time?Are you accounting or not the initiali

Therefore, we get the following plot for the speedup of the two codes:

**Strong scaling for N=2.000.000.000**



Notice that the touch first speedup basically remains the same while increasing the number of threads, and actually starts to decrease when having a larger number of threads due to parallel overhead. The touch by all speedup resembles more the behaviour predicted by Amdahl's law, and in particular for 8 threads or less the speedup is almost linear, while using more threads, as before, increases the overhead.

and how do you define the overhead? is it constant with the number of threads for both codes??

To estimate such overhead, I ran the serial version of the codes and the parallel one with only one thread. For both versions the overhead associated with OpenMP is between 0.2 and 0.4 seconds. On the other hand the overhead due to communication times and waiting times in this case is very small, at least for the number of threads used. Therefore by taking into account the time needed to spawn the threads we can estimate the parallel overhead to be less than 0.5 seconds even when using 20 threads.

communication in an openmp code?

Out of the two, the touch by all approach gives the best result. This is due to the fact that, as seen during the course, for each thread the data is placed in the most convenient place.

Another approach would be that each thread first allocates the memory it needs and then initializes it. A way to do so is to declare a global pointer, then pass it to each thread with a firstprivate clause so to be able to modify it. Then, after having divided the problem in equal parts, each thread carefully allocates the memory it needs starting from the memory location pointed to by the global pointer. This way the array is cor
each thread allocates and initialises the part it needs.

[*]well.. more or less.and the impler

## Exercise 2

### Binary search with OpenMP

First of all, the testing was done with $N_{search}$=20.000.000, $N_{data}$=100.000.000. The serial time was 8.70 seconds, whereas the parallel ones were:

| Threads | 1 | 2 | 4 | 8 | 12 |
|---|---|---|---|---|---|
| Times | 8.91 s | 5.5 s | 3.7 s | 2.9 s | 2.7 |

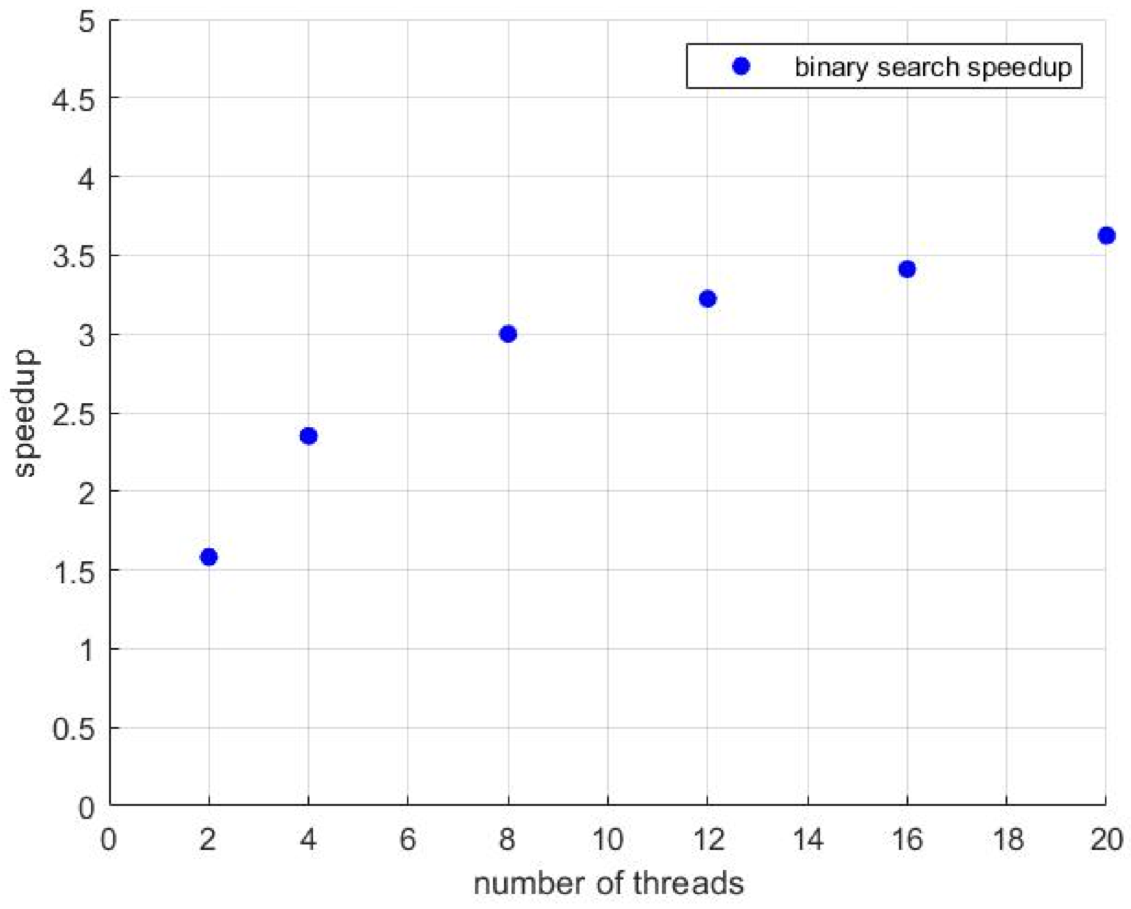
[*]...and HOW you do parallelize ?

Running the code with only one threads indicates the overhead of the OpenMP constructions, which in this case amounts to 0.2 seconds more or less. There is a decent, albeit not great, reduction in the execution times, which scale well only with a small number of threads. By increasing the number of iterations the code would probably scale better.

In the code each thread allocates and initialises the subset of the data it needs for the computations, rather than having it allocated by the master thread. This reduces the execution time by approximately 30%.

In the following plot we see a strong scaling test for algorithm.

**Strong Scaling for $N_{search}$=20.000.000, $N_{data}$=100.000.000**


binary search speedup
speedup
number of threads