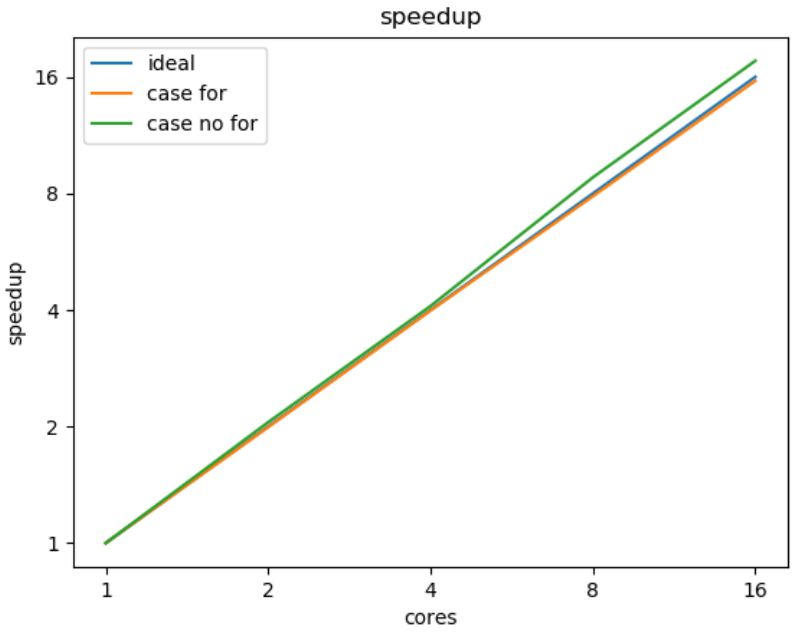
Sergi Solís 230306
Hong-ming Xiang 229517

LAB 1 REPORT

Q1.1 Explain which strategy and scheduler you choose to parallelize axpby and why. List the variables used and justify the selected data scope used (e.g. private, shared, etc).

We are not specifying any scheduler premise, that means that we are using the default (static) scheduler for this exercise. In this case we think that this will be optimal since all the iterations have to perform the same operation and we consider that the time taken by each iteration is the same. The variables used are x and y that represent the two vectors and the iterator i. Also we have a and b that are constant scalars and n that is the constant size of the vectors. In this case we are also using the default data scope since we want the vectors and constants to be shared because all the threads need to access this data and the iterator private.

Q1.2 Plot the speedup and explain the results. Include in the plot the scalability for axpby_openmp() and axpby_openmp_nofor(). Save the "speedup.py" file for the final report.



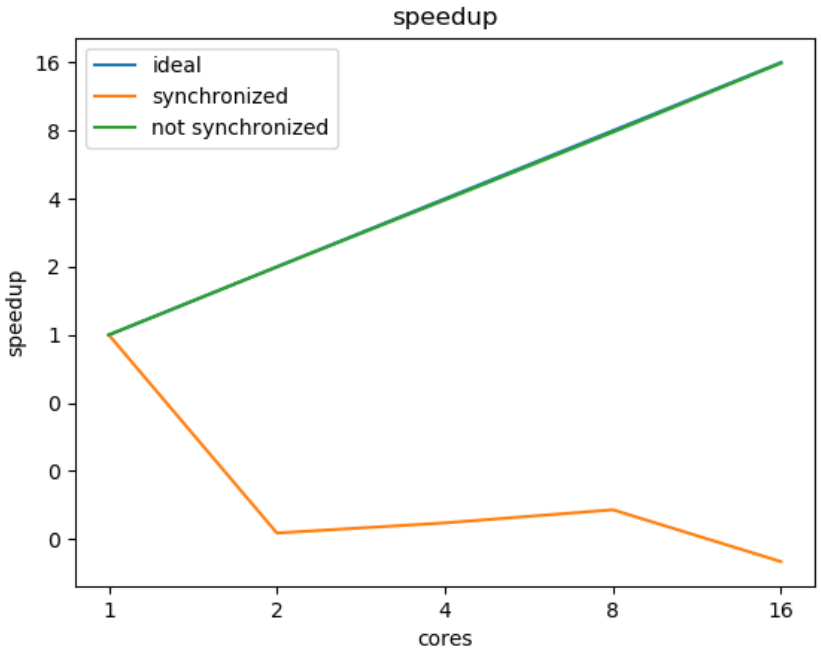
As we could see both methods have a very similar slope compared with the ideal one, so our parallelization is working as expected. However our "no for" case is above the ideal one that could have been explained by some variance in the execution time.

Q1.3 Introduce the formula that you have used to calculate the throughput and explain it.

Given the example provided, we deduced that the throughput calculations are the following: *Data(Megabytes)/time(seconds) = throughput(Megabytes)* since we need the throughput result on Gigabytes we will need to divide this result by 1000 (since 1 Gb = 1000Mb).

Q2.1 Do different iterations have different workloads? Why?
They don't have different workloads since we are performing the same operations without any iteration dependence, that means that we are not adding any operation that depends on "i".

Q2.2 Plot the speedup and explain the results. Save the "speedup.py" file for the final report.



As we can see in this picture the code implemented with synchronization has a very bad performance (we can deduce it since we have seen in class that is common that code related to synchronization causes overhead). Otherwise, the code made without synchronization (using reduction to optimize it even more) gives very good results, a very similar slope compared with the ideal one.

Q2.3 Propose two strategies to solve the problem of all threads writing on the same variable the total amount of points inside the circle:

- 1st strategy. Implying synchronization.
On this strategy we used the parallel for directive and the atomic one to provide a mutual exclusion section for the read and update part.

```
#pragma omp parallel for
for(i=0;i<steps; i++) {

    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;
    x = ((double)random_next/(double)PMOD)*(random_hi-random_low)+random_low;

    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;
    y = ((double)random_next/(double)PMOD)*(random_hi-random_low)+random_low;

    dist = x*x + y*y;

    if (dist <= r*r) {
        #pragma omp atomic
        total++;
    }
}
```

- 2nd strategy. Not implying synchronization but needing an extra step.
On this strategy we used the reduction method for doing the sum on each thread (in

```
#pragma omp parallel for reduction(+:total)
for(i=0;i<steps; i++) {

    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;
    x = ((double)random_next/(double)PMOD)*(random_hi-random_low)+random_low;

    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;
    y = ((double)random_next/(double)PMOD)*(random_hi-random_low)+random_low;

    dist = x*x + y*y;

    if (dist <= r*r) {
        total++;
    }
}
```

parallel) and sum up all of them in the final variable total.

What strategy do you prefer? Why?
We can use a synchronization mechanism as atomic in order to avoid race conditions, however in this case it will serialise the code and we think that it will be much more efficient to use reduction.

Q3.1 Which loop did you choose to parallelise? Why? Is there any dependency between iterations?

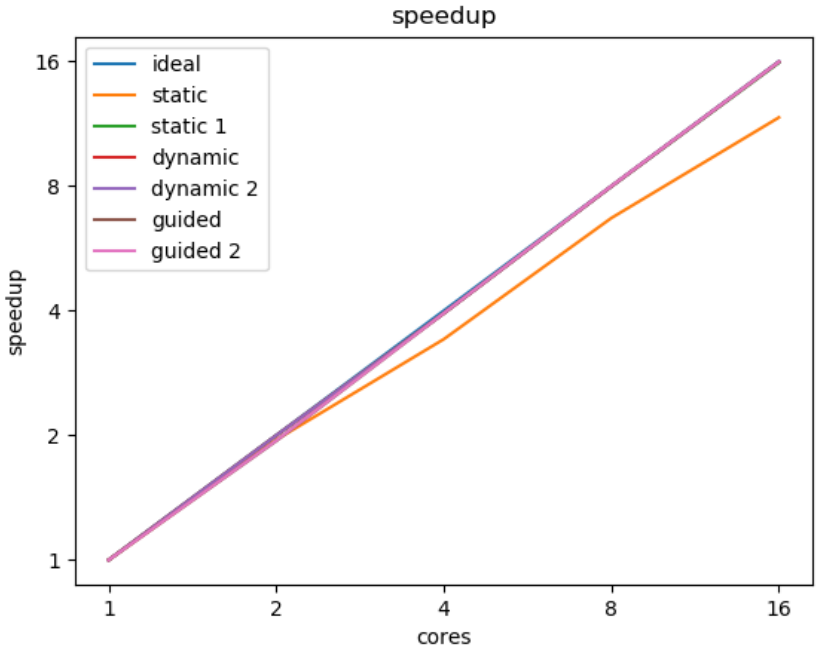
The loop that we have to parallelise is the first of all in the section in which the mandelbrot is generated, since we have tried to parallelise other sections and this case is the one that gave us the best result.

There is dependency between iterations of the different loops since iterators or variables that depend on it are used outside of its own for loop.

Q3.2 Use at least 6 different scheduling configurations (combinations of scheduling strategies and chunk size) and test them. Plot the speedup for all the scheduling combinations and discuss the results. Save the "speedup.py" file for the final report.

	A	B	C	D	E	F	G
1		static	static 1	dynamic	dynamic 2	guided	guided 2
2	1 core	1,195749	1,204177	1,215422	1,203511	1,204272	1,20645
3	2 core	0,612853	0,602798	0,608129	0,602032	0,621136	0,621145
4	4 core	0,35	0,305378	0,308467	0,305212	0,305287	0,305748
5	8 core	0,178575	0,150893	0,15208	0,150697	0,150923	0,150894
6	16 core	0,102071	0,075828	0,076178	0,07544	0,07547	0,075576

we collect the runtime of each of the scheduling configurations for each number of cores as you could see in the table above.



Then, as we could see on the results, almost all of the scheduling configurations run with the ideal shape except the default one (static,[chunk]) where it takes the number of threads as the chunk. The reason for that loss of performance might be some type of imbalance for this specific case since we could see that we started losing performance on the 4th core test and it seems that is getting worse on the 16th core test. Given the results, we decided to use the dynamic approach in our code.

Q3.3 List all the variables that you chose to put under each "data scope". Please, describe what is doing each "data scope" with the variables.

After trying multiple combinations of datascope we decided to let it as default since it guarantees us the desired image result (on some tests trying to redefine private/shared some variables the result of the image was wrong) and also it has the lower runtime.

Looking at the code we can deduce that we should define as private the i ,j and iter variables (iterators) since all the threads will be using these variables in order to write the content in

the Mandelbrot matrix and then avoid race conditions. This private data is put on a separate stack per thread (each thread reserves his own memory location for this variable).

All the other data that has been declared outside the local scope will be treated as shared data, that means that any thread using a shared variable will access the same memory location.