L2. Multiprocessors

Introduction

After a heuristic approach we have decide to gather our final data with problem sizes 100, 500 and 1000; with 1,2,3,4,5,6,7,8,9,10,11 and 12 as core numbers and for each test repeat the execution at least three times and get the lowest value.

Three problem sizes were chosen as they are enough to represent the changes in that variable.
The core range was chosen because the test was performed on an Intel i5-8250U (4 cores / 8 threads) that was plugged in to the charger to avoid power throttling.
The reason for executing three times and choosing the lowest value was due to that the CPU being used for this experiments is not fully idle, as Excel and the operative system are being executing at the same time, that's why by choosing the lowest value for each execution we assure that we don't pick high values caused by other processes. Another approach would have been setting the CPU priority of the experiment to the highest.

After doing some research on the performance of the CPU being used, the conclusions are that we should expect the four first application threads (numCore) to perform as independent CPUs -so that for every thread we divide the time of the execution of an ideal parallelizable software- but as it has only four cores, when we increase the number of application threads we are expected to earn an additional ~25% of speed (4 app threads vs 8 app threads) as Intel uses Hyperthreading and then, once we reach 8 application threads no further improvement is expected.

With the time results of your experiments:

a. For each size of the problem, estimate the *Execution time affected by improvement* (t_{aff}) and the *Execution time affected by improvement* (t_{unaff}).

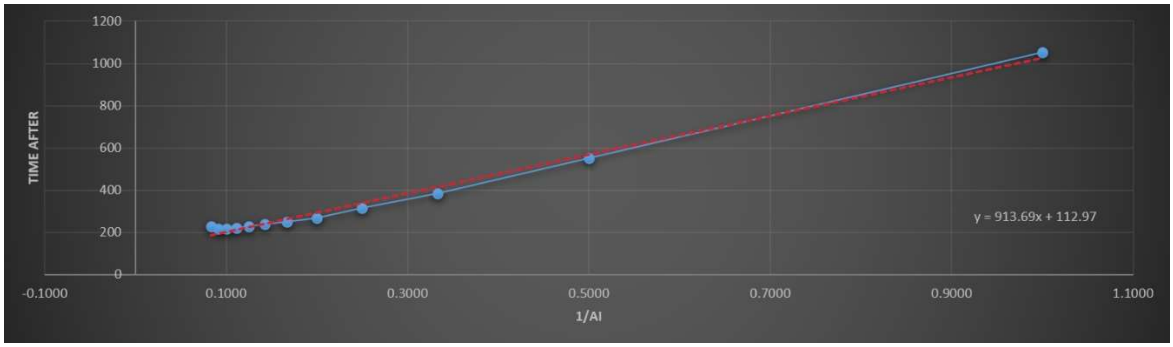
Remember the Amdahl's law:

$$t_{after} = \frac{t_{aff}}{A_i} + t_{unaff}$$

You can use the method of least squares to calculate both constants. To do this, you have to consider $x = 1/A_i$, so:

$$t_{after} = t_{aff} \cdot x + t_{unaff}$$

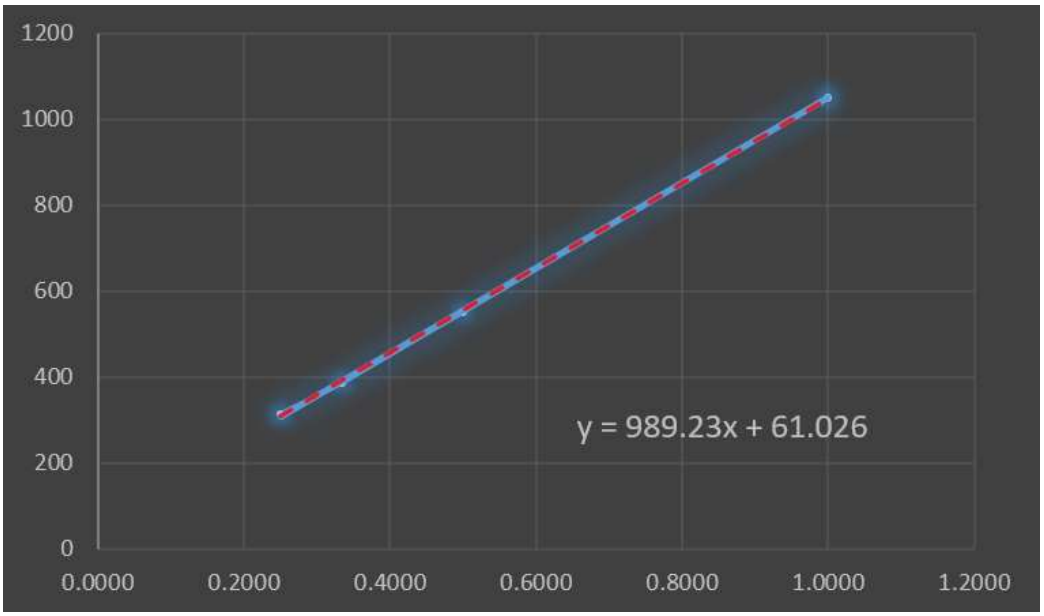
For problem size 100:



Relation between time and the inverse of the improvement. ProblemSize 100.

We can appreciate that for a small number of numCores (Ai=1 to Ai=0.25), the relation seems to be linear, but from them, if we add more cores (Ai gets lower) the time finds the bottom. This is due to that our system has only 4 real CPU cores, so adding more numCores will not give us the same performance beyond the real number of physical ones.

Therefore, we have decided to produce another figure that only has numCores from 1 to 4, as this are the real cores available. This will be useful to find a real trendline.



Relation between time and the inverse of the improvement. ProblemSize 100, numCores 1 to 4.

Here we can see that the relation is perfectly linear, and that the trend line function is the following:

$$f(x) = 989.23x + 61.026$$

Regarding to the time affected by the improvement and time unaffected, we have reached to these equations derived from the previous ones:

$$T_{affected} = \frac{(T_{before} * Ai) - (T_{after} * Ai)}{(Ai - 1)}$$

$$T_{unaffected} = T_{before} - T_{affected}$$

We can then apply these formulas to our data to find the values:

Time affected and unaffected by the improvement depending on the numCores.

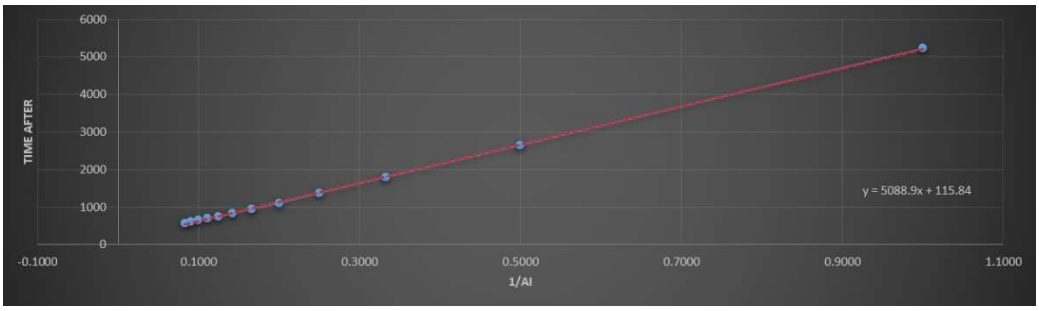
As seen before, we should only consider the values for numCores 2 to 4 to make conclusions about the problem, but others are shown additionally. Obviously numCore 1 is not displayed as it cannot have any improvement because is the baseline.

The conclusion here would be that the time unaffected (and tAffected as they are related) is only dependent on the problem size, but due to variance in data we can see that they are little discrepancies depending on the number of cores (always below or equal to 4).

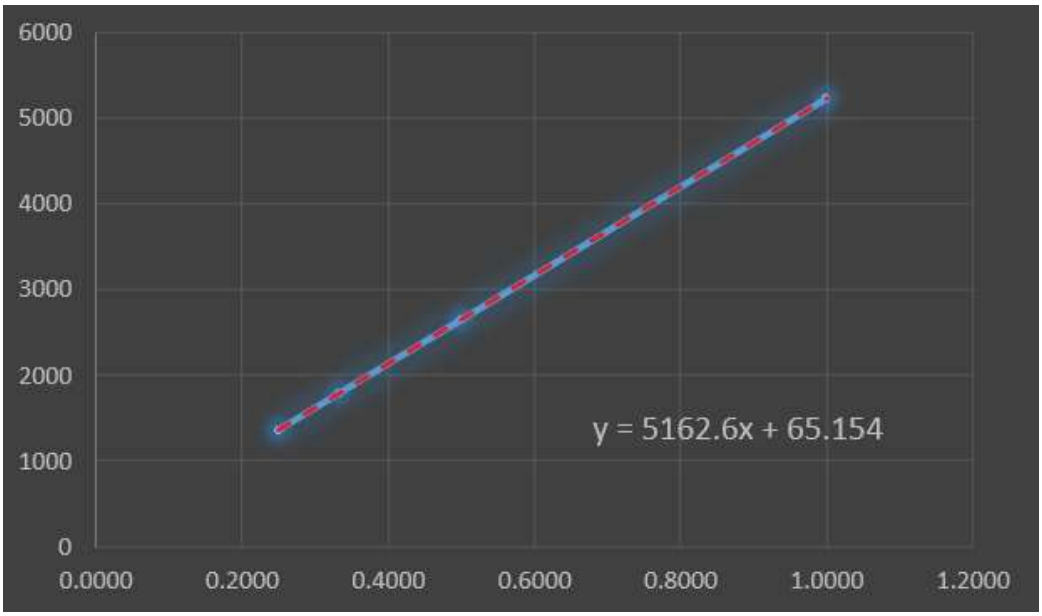
tUnaffected = ~58ms
tAffected = ~994ms
tTotal = 1052ms

We will do the same calculations for the other problem sizes.

For problem size 500:



Relation between time and the inverse of the improvement. ProblemSize 500.

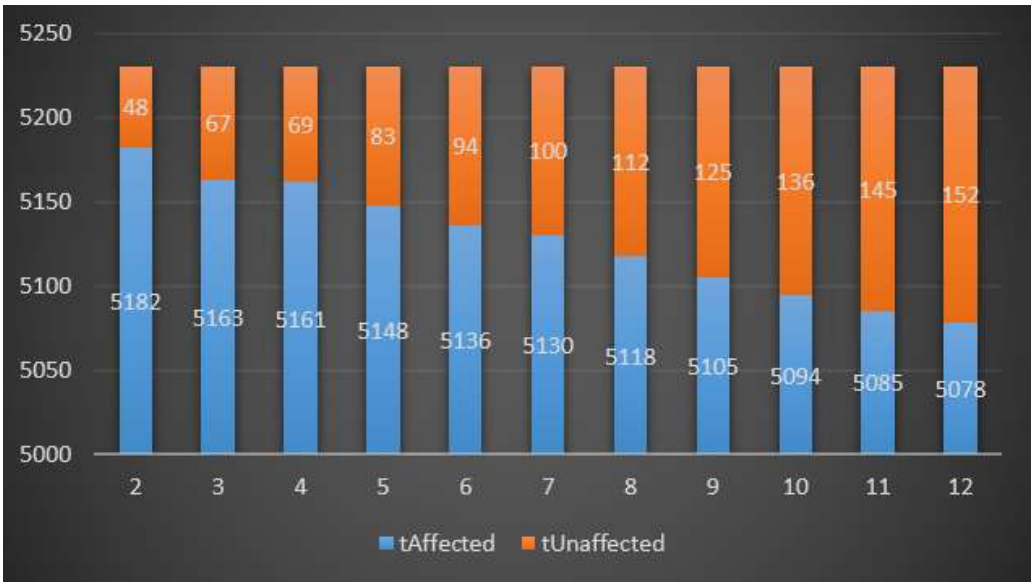


Relation between time and the inverse of the improvement. ProblemSize 500, numCores 1 to 4.

Here we can see that the relation is perfectly linear, and that the trend line function is the following:

$$f(x) = 5162.6x + 65.154$$

We can then apply the formulas to our data to find the values of time affected:



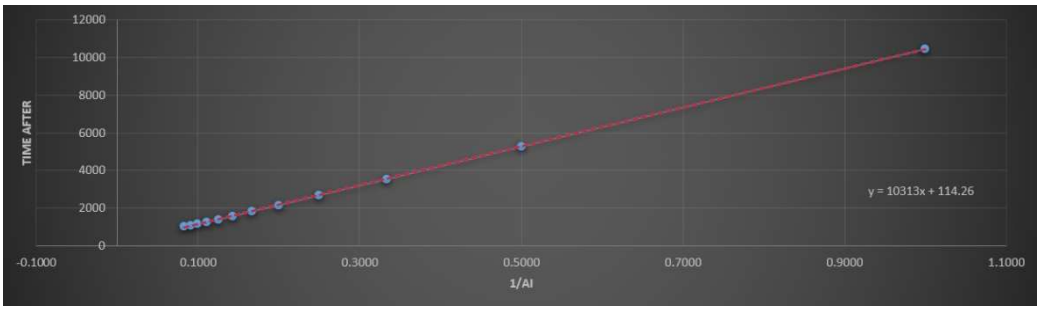
Time affected and unaffected by the improvement depending on the numCores.

We average numCores 2, 3, 4...

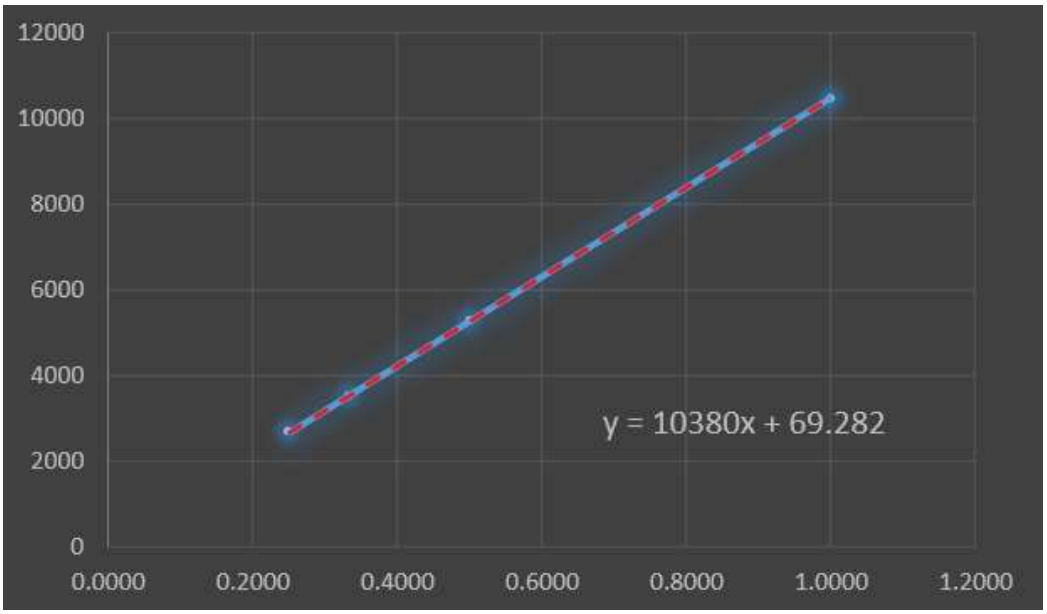
tUnaffected = ~61ms
tAffected = ~5169ms
tTotal = 5230ms

It seems we got a similar tAffected, let's see on the last problem size.

For problem size 1000:



Relation between time and the inverse of the improvement. ProblemSize 1000.



Relation between time and the inverse of the improvement. ProblemSize 1000, numCores 1 to 4.

Here we can see that the relation is perfectly linear, and that the trend line function is the following:

$$f(x) = 10380x + 69.282$$

We can then apply the formulas to our data to find the values of time affected:

Time affected and unaffected by the improvement depending on the numCores.

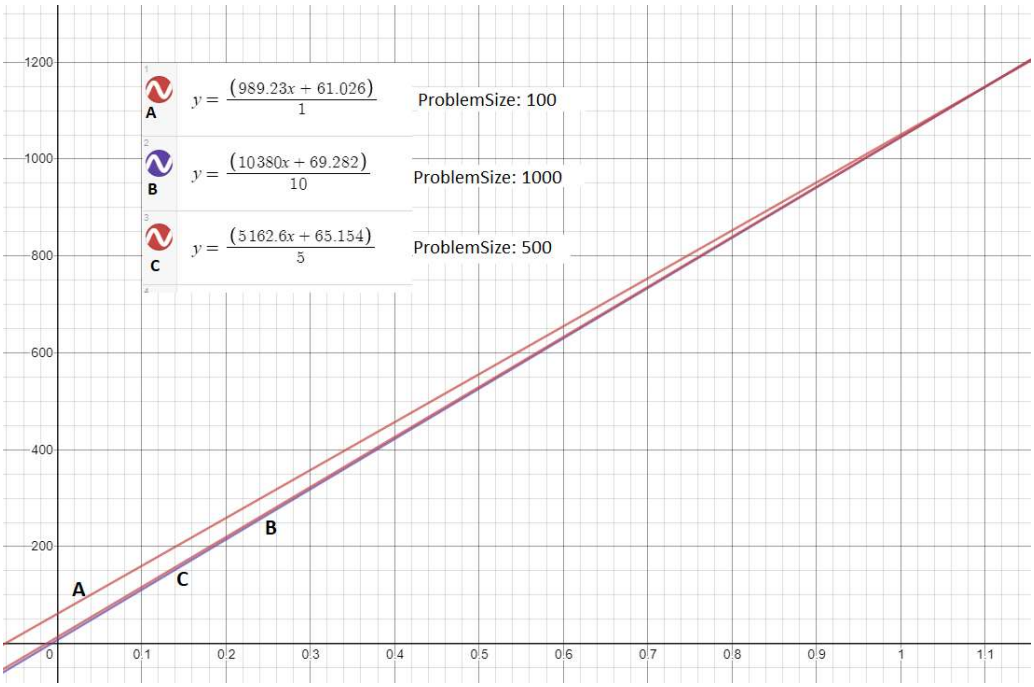
We average numCores 2, 3, 4...

tUnaffected = ~68ms
tAffected = ~ 10382ms
tTotal = 10450ms

Conclusion:

Through all the problem sizes, tAffected remains the same while tUnaffected grows as it is just tTotal minus tAffected.

We can also compare the three trendline functions we obtained.

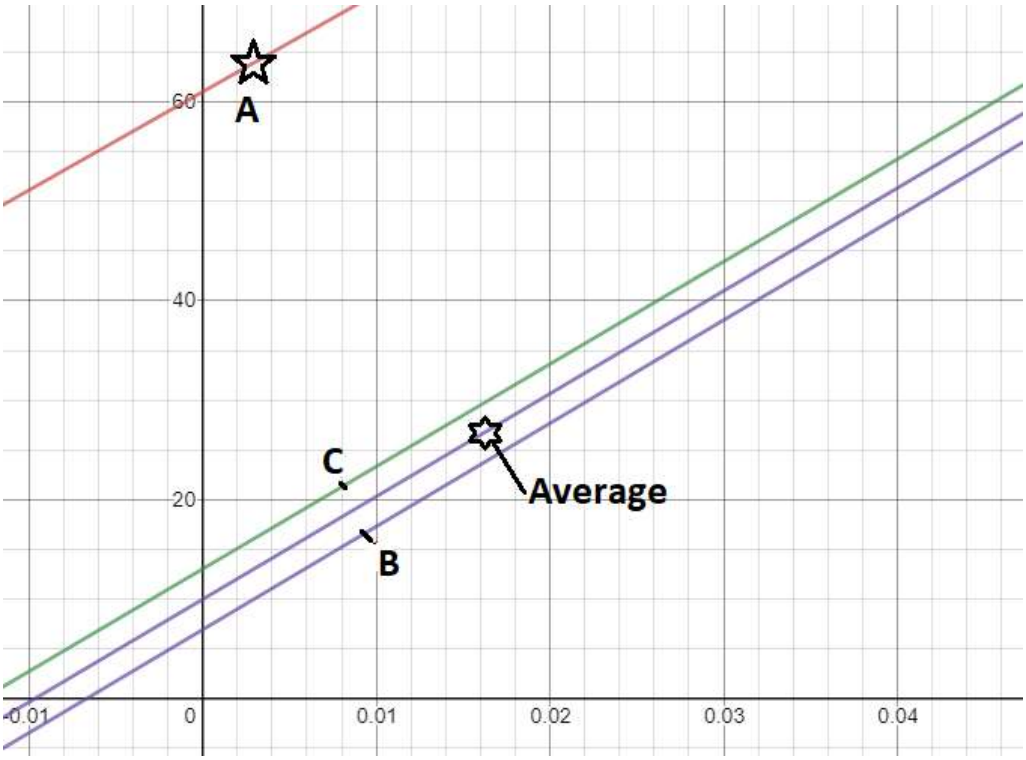


Comparison of the three trendlines

To compare them we have used and adjustment factor in as they come from different problem sizes, this factor is 1 for problem size 100, 5 for problem size 500 and 10 for problem size 1000. This adjustment factor allows us to account for the different problem sizes.

We can see clearly that all three functions are similar, but that B and C are almost the same. The difference in line A could be due that is the smallest problem size, so the measurements do not take into account the overhead of launching the experiment. In B and C, with bigger problem sizes this overhead is diluted.

So, by seeing this comparison we can see that there is a common function for all problem sizes. To estimate this function, we have averaged function B and C.



Average of trendline B and C

Relation between inverse of improvement (x) and time (y).

$$f(x) = 1034.26x + 9.9795$$

If we force it to pass through the origin, we get the following:

$$f(x) = 1034.26x$$

b. Estimate the relationship between the size of the problem and t_{aff}. To do this, consider that this relationship is a linear one and use the method of least squares:

$$t_{aff} = A_{aff} \cdot N + B_{aff}$$

As concluded on the previous section, the time affected depends on the size of the problem, with that results we obtain the following:

$$tAff = pSize * z - tUnaff$$

Being z a constant that relates the problem size with the total time like the following:

$$z = \frac{tTotal}{pSize}$$

So, for our experiments in our machine we obtain these values:

$$tAff = pSize * 10.46 - 62$$

(time units are expressed as milliseconds)

Z was calculated as the average of all three problem sizes with numCore=1

c. Estimate the relationship between the size of the problem and t_{unaff}. To do this, consider that this relationship is a linear one and use the method of least squares:

$$t_{unaff} = A_{unaff} \cdot N + B_{unaff}$$

As we said previously,

$$T_{unaffected} = T_{total} - T_{affected}$$

Also, we have found tUnaffected to be constant between problem sizes we can conclude that the relationship between size of the problem and tUnaffected is the following:

$$T_{unaffected} = 62 + P_{size} * 0$$

(time units are expressed as milliseconds)

This can be simplified as:

$$T_{unaffected} = 62$$

(time units are expressed as milliseconds)

So, we can conclude that tUnaffected is not correlated with the size of the problem.

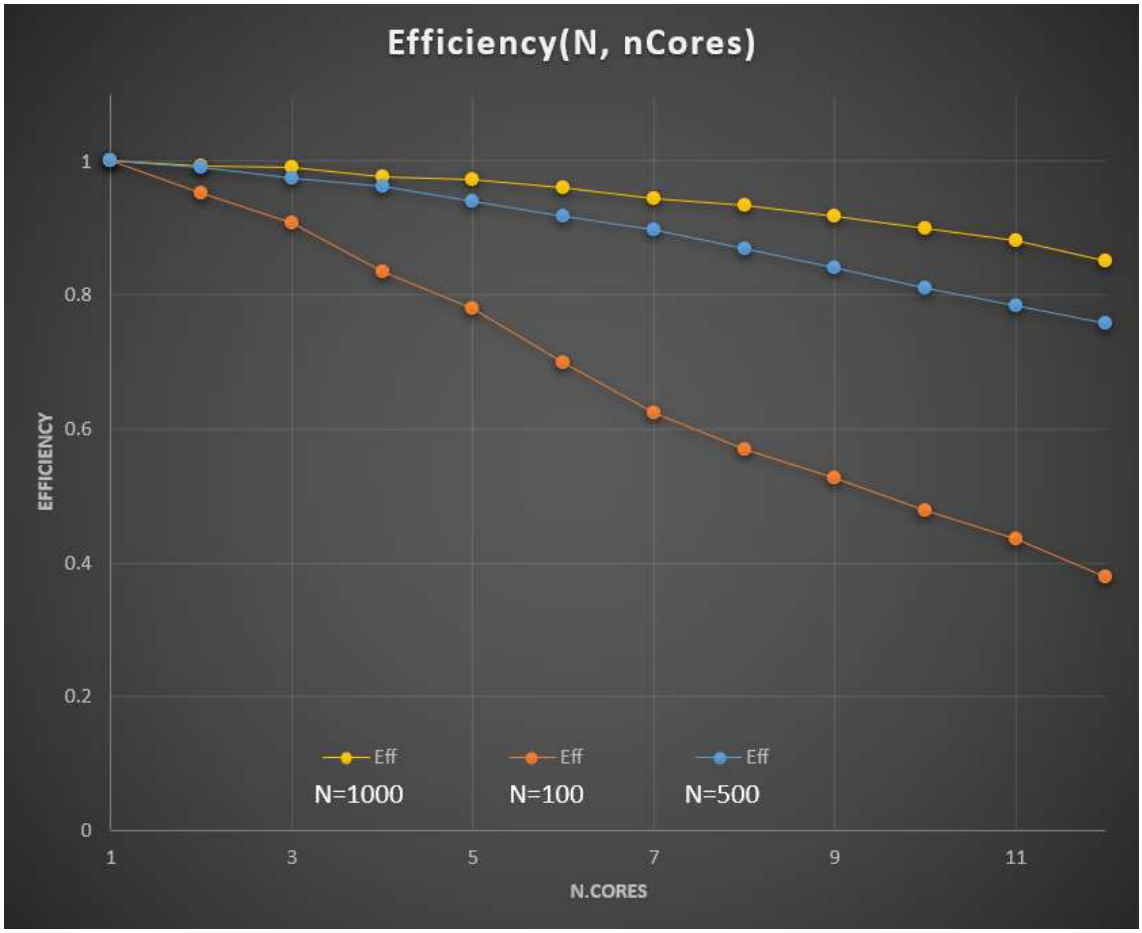
d. Calculate the efficiency of this program as a function of N and the number of cores.

To calculate the efficiency, we have defined it as follows:

$$Efficiency = \frac{\left(\frac{T_{before}}{Ai}\right)}{T_{after}}$$

(tBefore is time before improvement, a.k.a numCores=1)
(tAfter is time after improvement, a.k.a execution time)

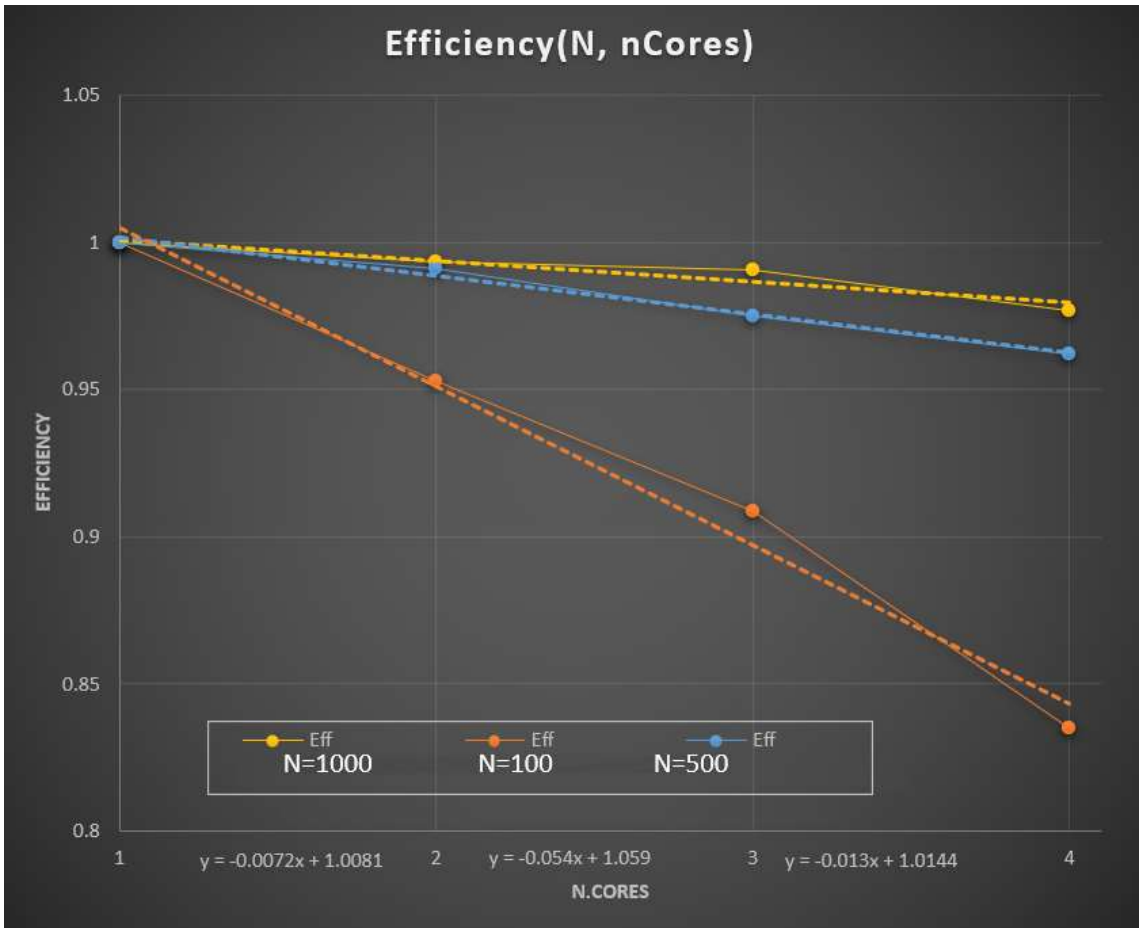
With our data we have plotted this chart:



Comparison of efficiency depending on problem size and numCores

As stated previously, we should divide the calculation in two parts, one for numCores from 1 to 4, and other from numCores 5 and above, as if the

computer had more cores, we should be looking on the first calculation to extrapolate, and not in the second one.



If we look to the coefficient that accompanies the X (0.0072; 0.054; 0.013) in each equation we can find the relation with N.

With the average of the three equations we reach to this approximation for efficiency depending on N and the number of Cores. The 6.37 constant comes from the average of these coefficients.

$$Efficiency = -\frac{6.37}{N} * nCores + \frac{6.37}{N} + 1$$

e. Explain the different results that you get.

In each sections the results were explained but we will provide a brief summary.

a) The time unaffected is constant in each problem size and the time affected varies with the problem size. This is expected if the affected code depends on problem size and unaffected code is almost constant.

b) The relation between time affected and time unaffected is the following.

$$t_{affected} = 1034.26\,N + 9.9795$$

$$t_{unaffected} = 62$$

(time units are expressed as milliseconds)

The explanation is the same as section a), that the unaffected code remains constant while the code affected depends on problem size.

c) The time unaffected does not depend on the problem size, so:

$$T_{unaffected} = 62 + P_{size} * 0$$

(time units are expressed as milliseconds)

$$t_{unaffected} = 62$$

(time units are expressed as milliseconds)

d) The efficiency it's just the ratio of the real improvement to the usage of additional cores, so if our software has no time unaffected, the efficiency should be 100%.

This remains true while the additional cores are available on a real machine. This is the equation of which we have less confidence as it has much variance depending on the nCores and if they surpass greatly the number of physical cores. Take it as a grain of salt.

$$Efficiency = -\frac{6.37}{N} * nCores + \frac{6.37}{N} + 1$$

2.2. SMachine Simulator

Create a program in the SMachine Simulator to calculate the minimum of a vector with 8 elements. Your program must do the following actions:

1. Load the numbers in the eight first registers.
2. Find the minimum.
3. Store the minimum in the first memory address.

Try to make the program as fast as possible.

The first approach has been to make a single thread that works as expected but reducing the number of instructions as maximum as possible.

The first strategy has been to remove the loops and just write the instructions in order so we save some instructions jumping through the loop.

The software first loads the eight values, in the example DEADBEEF is loaded in registers 01234567 (lines 1 to 8), then it assumes that the first value is the current minimum (line 9), then it just checks if the other values are lower than that update the current minimum with new values (lines 10 to 23), then it just moves the best minimum to the first memory address (line 24).

```
1   TRANSFRE    0,D     //loading 8 values
2   TRANSFRE    1,E
3   TRANSFRE    2,A
4   TRANSFRE    3,D
5   TRANSFRE    4,B
6   TRANSFRE    5,E
7   TRANSFRE    6,E
8   TRANSFRE    7,F     //finished loading 8 values
9   TRANSFRR    8,0     //R0 is our best minimum yet, store it on R8
10  CMPMENOR    8,1     //if R1 > R8 skip the next instruction
11  TRANSFRR    8,1     //R1 is our best minimum yet, store it on R8
12  CMPMENOR    8,2     //if R2 > R8 skip the next instruction
13  TRANSFRR    8,2     //R2 is our best minimum yet, store it on R8
14  CMPMENOR    8,3     // ...
15  TRANSFRR    8,3     // ...
16  CMPMENOR    8,4     // ...
17  TRANSFRR    8,4     // ...
18  CMPMENOR    8,5     // ...
19  TRANSFRR    8,5     // ...
20  CMPMENOR    8,6     // ...
21  TRANSFRR    8,6     // ...
22  CMPMENOR    8,7     // ...
23  TRANSFRR    8,7     // ...
24  TRANSFMR    0,8     // Store our best minimum, R8, on fist memory address
```

As it has been proved, some instructions like CMPMENOR and CMPMAYOR took more than one cycle to complete, so we can speed up our software by launching some threads.

This software takes around 300 seconds to end, that is 60 cycles (if they last 5 seconds). This time can vary depending on the input.

Now we will try to parallelize this software to improve its performance.

To implement multithreading, it's just needed to divide your program in more than one file and feed them to the emulator.

We believe that the order in the files is important, that's why the files are named in the order they have to be feed into the simulator to maintain the software consistency.

As the simulator has four cores, we've decided to launch four threads with the following code.

(1).sgb

```
1   TRANSFRE 0,D
2   TRANSFRE 1,E
3   TRANSFRR 8,0
4   CMPMENOR 8,1
5   TRANSFRR 8,1
6   CMPMENOR 8,9
7   TRANSFRR 8,9
8   CMPMENOR 8,A
9   TRANSFRR 8,A
10  TRANSFMR 0,8
```

(2).sgb

```
1   TRANSFRE 2,A
2   TRANSFRE 3,D
3   TRANSFRR 9,2
4   CMPMENOR 9,3
5   TRANSFRR 9,3
```

(3).sgb

```
1   TRANSFRE 4,B
2   TRANSFRE 5,E
3   TRANSFRR A,4
4   CMPMENOR A,5
5   TRANSFRR A,5
6   CMPMENOR A,B
7   TRANSFRR A,B
```

(4).sgb

```
1   TRANSFRE 6,E
2   TRANSFRE 7,F
3   TRANSFRR B,6
4   CMPMENOR B,7
5   TRANSFRR B,7
```

The instructions are basically the same ones as the single threaded software but distributed among the four files.

Each thread loads two input values at the start (lines 1 to 2 in each file), then they store the smallest one in they own register (lines 3 to 5 in each file).

Once this is completed, we have four relative minimums, as each thread has calculated its own, they are stored on registers 8, 9, A and B.

Now, thread (1) and (3) will compare these values to find new minimums (lines 6 to 7).

Finally, thread (1) will transfer the absolute minimum to register 8 and then from them to memory ending the program.

This software takes around 130 seconds to end, that is 26 cycles (if they last 5 seconds). This time can vary depending on the input.

This is 230% faster than our single threaded approach.