F2B304 - TP Parallelism

ALVAREZ Paulina
BENCSIK Andrei

Caution: there are very funny jokes ahead!

1. Personal data leaks

Exercises :

2) For every person there are :
Last Name, Telephone Number, City, First Name, Address, Landline, Password, Postal Code, Email, Mobile, Comment.
The people are from France based on the postal codes. The only information that is hashed is the password, everything else is in plain text.

3) The passwords are all of length 32 characters. When you do not use salted hashing and the passwords repeat it is easy to decrypt by using a list of common words.

4) Using *count_passwords.py*, here the top five most used passwords (hashed).

Hash	Password	Count
7cf2db5ec261a0fa27a502d3196a6f60	*pizza*	1377
Ab4f63f9ac65152575886860dde480a1	*azerty*	1055
E10adc3949ba59abbe56e057f20f883e	*123456*	950
Be5d9ba998a9412a49a6e9d4fcedf931	*doudou*	546
199d2cdff476357ae65c3b6291e91a45	*dominos*	521

**) Anybody surprised ? Might just be the first ones we would try ..*

1.1 Get the words right

1.1.1 Dictionaries

Script *dump_aspell*, gets the words from the french dictionary, sorts and saves them in the file “french.txt”.

aspell -d fr dump master | sort -u > french.txt

1.1.2 RTFM

Exercises :

2) Script *ungroffman*, using groffer command to create a file with the list of words in the man page given as an argument.
3) Some scripts take longer because of the use of temporary files, which involves disk activity when creating, writing and destroying files.

1.1.3 Data Parallelism

Exercises :

1) 1893 man pages found with command : find /usr/share/man/fr -type f | wc -l
2) Script *ungroff_all_serial*, writes a file with a sorted list of unique words from all the man (french) pages.
3) Using command *time* : Execution time of 20.50s with 234 man pages.
4) File processing is the most time consuming part of the script.
Script *ungroff_all_parallel*, parallelize the file processing of the *ungroff_all_serial* script.

1.2 Crack it

Preliminary exercises

a) MD5 hashes computation time using one core.

Computing time using python script *crackit.py*

0.6 s to compute 629574 hashes.
9.53025379066e-07 s for hashing 1 word in average.

Otherwise, using the command ***openssl*** ***speed*** ***md5***, we have the following results on an i7 5500u, dual core, multi threading enabled

Doing md5 for 3s on 16 size blocks: 8743239 md5's in 2.99s
Doing md5 for 3s on 64 size blocks: 6556709 md5's in 3.00s
Doing md5 for 3s on 256 size blocks: 3851550 md5's in 3.00s
Doing md5 for 3s on 1024 size blocks: 1454607 md5's in 2.99s
Doing md5 for 3s on 8192 size blocks: 210535 md5's in 3.00s

The 'numbers' are in 1000s of bytes per second processed.
type 16 bytes 64 bytes 256 bytes 1024 bytes 8192 bytes
md5 46786.56k 139876.46k 328665.60k 498166.41k 574900.91k

So.. python seems pretty good at doing MD5 hashing.

b) Same using all the available core/threads.
This time using the script in **hashing_parallel** (using GNU parallel - check in the git repo) and the ***md5sum*** function we get:

/hashing_parallel 13,11s user 12,84s system 74% cpu 34,767 total

That is for 10 000 passwords.

c) Compared to the list you've created, what can you conclude ?
It seems that python is very fast for doing this sort of work. No wonder it's the tool of choice for many hackers...

Exercises

1) Using our python script (since python was very fast and convenient), called crackit_v2.py, we managed to crack more than 5_000 passwords! (We do want those 5_000 beers …!!!!!! Actually, more precise, 5_515). You can see for yourself in the **passwords.txt** file.
You can also check out the most used passwords, in what we call the topXXXpasswords. Check that file out as well!

2) Seems like the program is fast enough for this task. The most time consuming part is the Hashing, but it seems like python does away with that very well.

3) We added more dictionaries for fun! Using only the aspell dump we were able to find about 3_000 passwords. Adding the man pages.. 4_000, adding a new dictionary, 5_400 and last, but not least, adding a list of the most common passwords, we squeezed another 100.

To use our python script you can either call --help or believe us and use it with the dictionaries in the dictionaries folder. (You can add your own, but they must be one word per line)

Usage: ***python crackit_v2 dict1 dict2 dict3*** .. the more the merrier !

Rejoice with some of the most common (and funny, well if you're a geek that is):
('1bbd886460827015e5d605ed44252251', '11111111') -> binary!
('7813d1590d28a7dd372ad54b5d29d033', '6969') -> how does this even work ? two 69s?
('c8837b23ff8aaa8a2dde915473ce0991', '123321') -> better than 12345?
('21b72c0b7adc5c7b4a50ffcb90d92dd6', 'matrix') -> "**N:** I thought it wasn't real.
M: Your mind makes it real." Yaasss
('b59c67bf196a4758191e42f76670ceba', '1111') -> more binary

('7cf2db5ec261a0fa27a502d3196a6f60', 'pizza') -> I am already hungry, wait till you see below..
('acc6f2779b808637d04c71e3d8360eeb', 'pussy') -> this one was unexpected..

Conclusions

1) Because if it is decrypted in one site, there is a strong chance the attacker will try with the same password in other sites. For example, if someone in the database has the same password for the mail it's easy to hack.

2) We would recommend the usage of salted hashes to all the field in the database, not only passwords. They should not permit users to have such simple passwords.
We would also recommend that somebody get fired really quick and be replaced with one of us. We do hope they pay in pizzas.

3) Yes, because we love pizza and they have a decent price cut when you order online. You should definitely try "La reine"! Yummy!

2. Let C : warming up

2.1 Total recall

Reduction loop :

```
int r = 0;
int vec[n];
for (size_t l = 0; l < n; 1++) {
        r += f( vec[i] );
}
```

If *f* has the good properties : it should return integer values in the case presented, so the summatory is actually associative and the order of sums will not alter the result, making parallelization possible.

2.2 Warming up : Min/Max of an array

Exercises

1) The original code for main uses one loop to **initialize** the array with random numbers and one to reduce and find **min** and **max**.
We are not sure if we understood if this is what you wanted, but here goes.

a) Transferring both jobs into one loop is easy. The code for reduce is moved to the **random_init** function (this can be seen in **main_one_read.c**).
First when we check the time elapsed for each execution:

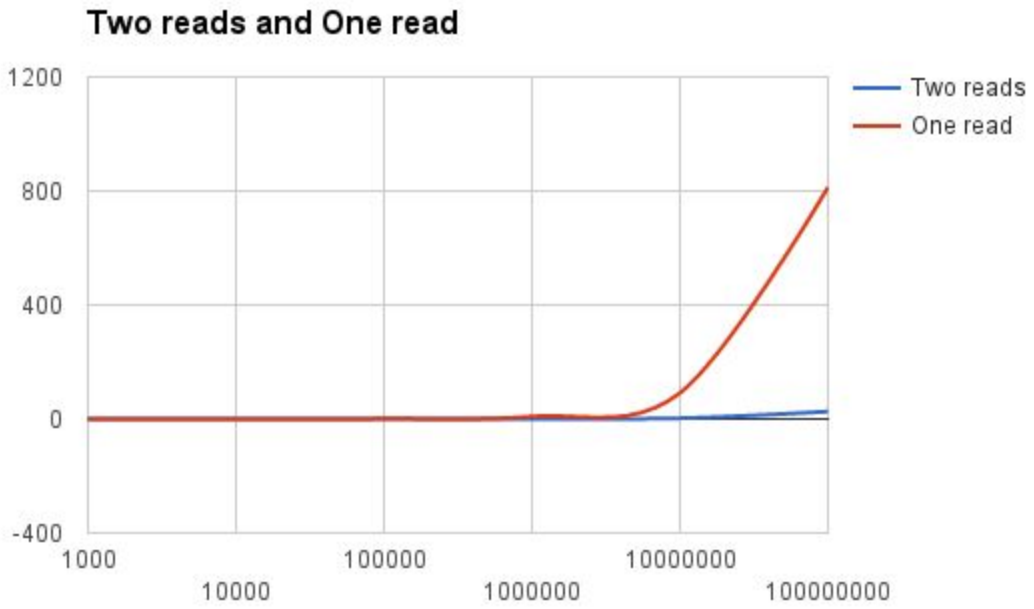


Fig 1) Time elapsed in milliseconds for both cases

SURPRISE!
For low values the difference is not easily noticeable, but when we get to millions, the time difference is clear. Till now, the winner is the version with two array reads.
Let's move to throughput:

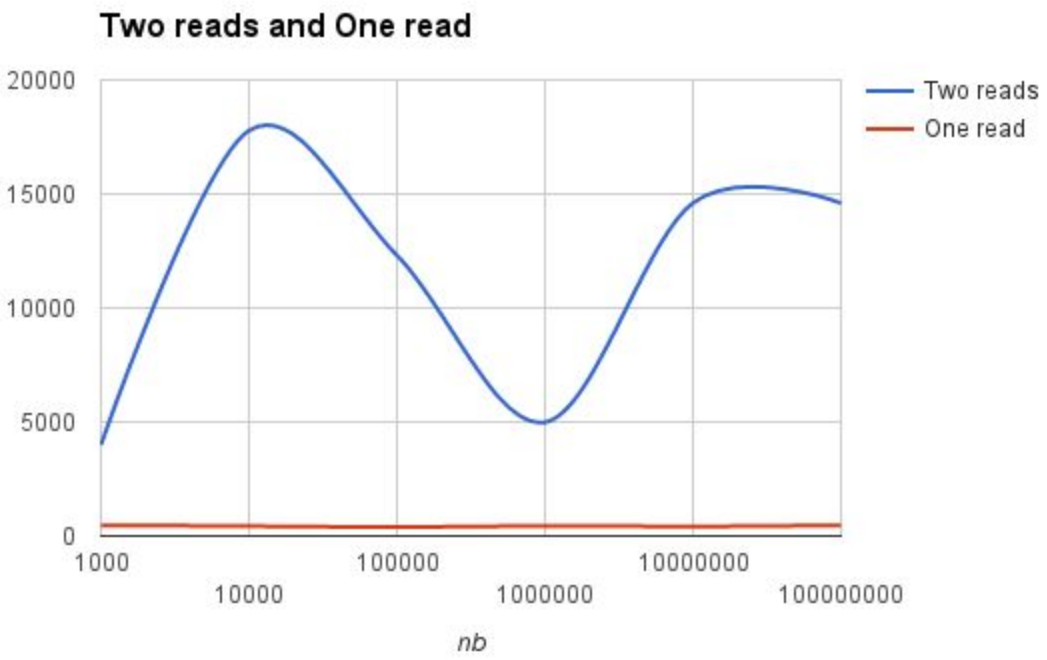


Fig 2) Throughput in MB/s for the same two programs

Not so surprising anymore, the two reads wins again. And by a hefty margin.

A possible explanation for this is that when using the one read we do many operations in the same loop. Thus we would require moving some data around : RAM <-> cache. Whilst this is happening, the cache may get overwritten by the time the loop reiterates, causing a new RAM access.

Using **strace** we can also see more **mmap** calls for the one loop -> more memory operations.

strace ./main 1000

Just an example : Two loops - 1000 elements - **6 mmap**

One loop - 1000 elements - **11 mmap**

Checking the assembly code for both does not, unfortunately, show much of a difference. (gcc -S -c main.c ..)

The Makefile has new rules for trying out and seeing the ASM code.

Note! If this wasn't what you wanted, we are sorry, but that is what we understood! And we have learned something in our effort!

b) We have parallelized the loop using the reduction clause:

```
#pragma omp parallel for reduction(max : local_max), reduction(min : local_min)
for (size_t i = 0; i < n; i++) {
        const uint32_t v = vec[i];
        if (v > local_max) {
                local_max = v;
        }
        if (v < local_min) {
                local_min = v;
        }
```

You can check this out in the 3-openmp folder using the, making the project and running **./main n**

We have also used the correction provided to see how much of a difference it can make.

Using the reduction version, we got better results than a simple **omp for loop**

./main_more_advanced 100000

minmax: in 3.93200 ms. Input (#/size/BW): 100000/0.38147 MB/97.01674 MB/s | Output (#/size/BW): 1/0.00000 MB/0.00097 MB/s

min = 3722 / max = 2147469841

./main 100000

minmax: in 1.16611 ms. Input (#/size/BW): 100000/0.38147 MB/327.13147 MB/s | Output (#/size/BW): 1/0.00000 MB/0.00327 MB/s

min = 3722 / max = 2147469841

c) After changing the Makefile, the TBB program works on our setup. Note that the linker seems to ignore the flags if there is no file requesting it. So in order to make it work we have put the file name (main.cpp) before the compiler flags, by modifying the makefile.

Here we go!

The template provided is made of a class that mandatorily has an **operator**, which is min/max in our case. Then it needs two constructors : one simple constructor and a **splitting constructor** to make division easy. For the rest, we just use the object (*minmax)* and pass it to

tbb::parallel_reduce(tbb::blocked_range<size_t>(0, n), ***minmax****);*

Performance is similar to OpenMP for this case, we do get some milliseconds shaved off the time we had with a simple OpenMP reduce:

minmax: in 19.59705 ms. Input (#/size/BW): 100000000/381.46973 MB/19465.66743 MB/s | Output (#/size/BW): 1/0.00000 MB/0.00019 MB/s

min = 7 / max = 2147483611

The code is too complicated for such a small performance gain. But since it is aimed at being similar to OpenMP we can see where it could be interesting

3 Edge detection

3.1 Sequential Algorithm

Exercises

1) The Makefile has new rules for trying out and seeing the ASM code. Measuring performance of **qt_edge/0-naive-linear** compilation, using the command :

time make all

	Size [KB]				Time [s]			
Optimization	main	helpers.o	main.o	edge_detect.o	user	system	cpu	total
O1	278	278	242	36	1.89	0.16	98%	2.088
O2	288	280	260	37	1.84	0.22	98%	2.090
O3	290	281	261	39	1.87	0.20	98%	2.097

Running the code in **qt_edge/0-naive-linear** with different levels of compiler optimization, using the command :

time ./main ../images/Lenna.png 1

Optimization	User time [s]	System time [s]
-O0	0m0.164	0m0.028
-O1	0m0.112	0m0.028
-O2	0m0.116	0m0.020
-O3	0m0.132	0m0.008
-O3 -ftree-vectorizer-verbose=1	0m0.124	0m0.012

2) The valgrind command outputs a file called callgrind.out.[pid]

valgrind --tool=callgrind ./make ../image/Lenna.jpg 1

3.2 Task based parallelism

Exercises

For the next parts we have changed to the images in the images folder and we move to a size of 2 (as can be seen in the execution commands). We did so so we can have a longer running time so we could actually make a more fine difference.

1) Running both linear and parallel with OpenMP's section with the command

./main ../images/dreama-high.jpg 2

File name	Method	Time [ms]	Input [size/BW]	Output [size/BW]
	linear	299.878	19.7754 MB/65.9448 MB/s	19.7754 MB/65.9448 MB/s
	sections	166.782	19.7754 MB/118.57 MB/s	19.7754 MB/118.57 MB/s
	task	167.416	19.7754 MB/118.121 MB/s	19.7754 MB/118.121 MB/s
	disaster	621.138	19.7754 MB/31.8374 MB/s	19.7754 MB/31.8374 MB/s

Simple #pragma omp parallel sections -> section

```
#pragma omp parallel sections
{
	#pragma omp section
	{
		erode(min_img, gray_img, width, height, edge_width);
	}
	#pragma omp section
	{
		dilate(max_img, gray_img, width, height, edge_width);
	 }
 }
```

The running time is reduced by 50% with sections! But the functions are pretty similar .. so might explain why. Not sure if it's a limitation but the tasks running in each section should be completely independent (but that should be the case with any method we use, be it sections or tasks..). On another hand the tasks should take about the same time to finish, like this we get the sweet 50% time reduction. Also, reading 3 lines ahead (because we always do that) there is another limitation -> sections can't be nested !

Because we are curious .. let's see which of the functions takes longer ..
And the winner is : none.. Because they take the same amount of time, by doing a mean over the results.

2) Using tasks we get almost the same results as with sections, if not a bit slower. Interesting fact is that it seems like the threads wait for eachother if we don't put them in an atomic section (a #pragma single). Without that pragma the results are a disaster!

3.3 Data parallelism

Exercises

1) Using the **#pragma omp parallel for** gives us a good improvement over the single threaded version. It even gives an advantage over the **#pragma omp sections** version. Improvement is around 30%.

This might be due to the fact that the CPU this is running on is a hyperthreaded dual-core.

./main ../images/miranda-high.jpg 2
edge-detect: in 125.133 ms. Input (#/size/BW): 1/14.0625 MB/112.381 MB/s | Output (#/size/BW): 1/14.0625 MB/112.381 MB/s

2)Using different types of scheduler types: dynamic, static, guided yields almost the same results. The running time difference is so small that it's probably due to other stuff going on at the same time.

3.4 Improve locality

Exercises

1) The loop fusion was made in the 2-linear-tasks. (It's true that this folder structure is becoming pretty hard to understand ..) When running the **time** command, we obtained the following :
./main ../images/miranda-high.jpg 2
edge-detect: in 134.846 ms. Input (#/size/BW): 1/14.0625 MB/104.286 MB/s | Output (#/size/BW): 1/14.0625 MB/104.286 MB/s
0,34s user 0,04s system 29% cpu 1,291 total
This is for the serial algorithm. Performance gain ?
There is a very big performance increase using one loop. More precisely a 53.3 % decrease in running time. Now that's a huge win!

Let's see how much we can improve on that by using OpenMP's parallel for:
time ./main ../images/miranda-high.jpg 2
edge-detect: in 73.6511 ms. Input (#/size/BW): 1/14.0625 MB/190.934 MB/s | Output (#/size/BW): 1/14.0625 MB/190.934 MB/s
./main ../images/miranda-high.jpg 2
0,47s user 0,05s system 39% cpu 1,306 total

Not bad at all !!

2) Is it just the loop overhead ? Well, not exactly. But it surely contributes as well. There are 4 more mutex locks/unlocks in the two loop version (157 in the one loop, 161 in the two loop). Much difference. Such wow. (insert Doge here)

3.5 Vectorizing

Exercises

1) The processor we used is an i7 5500U. It has the next vector instructions:

- MMX instructions
- SSE / Streaming SIMD Extensions
- SSE2 / Streaming SIMD Extensions 2
- SSE3 / Streaming SIMD Extensions 3
- SSSE3 / Supplemental Streaming SIMD Extensions 3
- SSE4 / SSE4.1 + SSE4.2 / Streaming SIMD Extensions 4
- AVX / Advanced Vector Extensions
- AVX2 / Advanced Vector Extensions 2.0

2) From the processor documentation the AVX technology uses 16 register. Each one can handle 4 x 64 bit doubles, so it can handle (4 x 8) x 8 bits (since the pixel is an uint8. So 32 pixels at once.

3) Well using the vector instructions we got these results:
./main ../images/miranda-high.jpg 2
edge-detect: in 74.1889 ms. Input (#/size/BW): 1/14.0625 MB/189.55 MB/s | Output (#/size/BW): 1/14.0625 MB/189.55 MB/s
Which is very good! The performance is the same as it was for the openMP version. If running it multiple times we do get values around 73 milliseconds.

run	1	2	3	4	5	6	7	8
time(ms)	78.78	71.98	76.55	72.4	76.9	72.75	75.06	72.47

Average ? **74.61125 milliseconds**

A conclusion after trying to figure out how all these methods work ?
The most simple way to parallelize was #pragma omp parallel for. The TBB version is a bit more complicated to use, but gives a slight performance improvement. As for the vector instructions? Well we either did not understand them well enough (and we probably did not), or they just do not offer a huge improvement over what we had before.

What we would use if we had the choice? We would probably go for the OpenMP version. If it seems that this does not offer sufficient improvement we would try to move to TBB (if the environment offers it - that is also a problem - while openMP is pretty much installed all over, tbb is not yet so famous). Last but not least, we might try to squeeze the last ounce of performance with vector instructions (but at least for now this would be the last resort). What seems a recurring theme is that we get the huge BOOST in the first step of naive parallelization. Then the increase becomes smaller as we construct over that, in the end we just over complicate the code and we get insignificant improvements (for this application at least)

4. Histograms

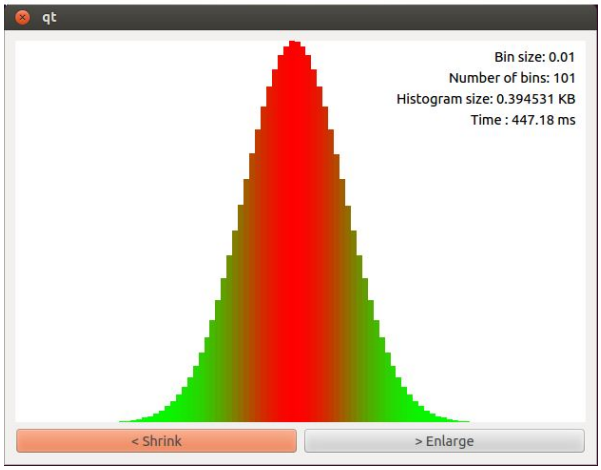
Exercises

1) To compile the code in **/histogram**

```
%>qmake -project hist.pro
%>qmake hist.pro
%>make
```

2) The qt application in **/histogram/qt** running with the command **./qt --gen-[normal/uniform] N**, where N is the size of the interval, with different inputs gives the following results :

Uniform distribution N=100 000 000



Normal distribution N=100 000 000

The time taken to process is around 0.01ms for N=1 000, 0.5ms for N=100 000, 5ms for N=1 000 000, 45ms for N=10 000 000, and finally, around 450ms for N=100 000 000. You can clearly notice the difference starting from N=10mil.

More Exercises

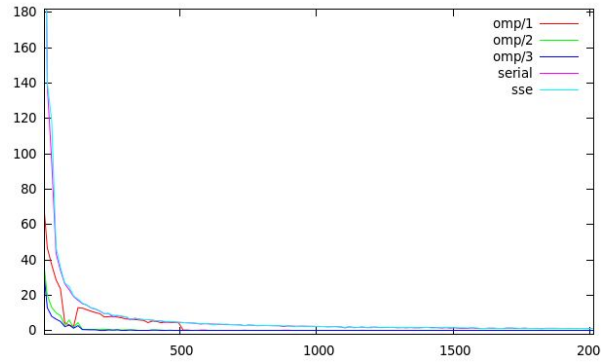
1) Running the code in **/bench** with the command **./benchs --gen-uniform N nbins**, where N is the size of the interval and nbins is the number of bins.

N/nbins	1 000 000	5 000 000	10 000 000
1 000 000	serial: 0.13461ms sse: 0.139728ms omp/1: 2.44841ms omp/2: 3.54979ms omp/3: 4.6613 ms	serial: 0.916979ms sse: 0.905638ms omp/1: 12.205 ms omp/2: 18.3637ms omp/3: 25.5749ms	serial: 2.11399ms sse: 1.91903ms omp/1: 24.8131ms omp/2: 36.6606ms omp/3: 48.0096ms
5 000 000	serial: 0.142132ms sse: 0.148038ms omp/1: 2.46653ms omp/2: 8.51953ms omp/3: 4.88244ms	serial: 1.00314ms sse: 0.957ms omp/1: 12.4441ms omp/2: 18.61ms omp/3: 24.399ms	serial: 2.06828ms sse: 1.98426ms omp/1: 24.8914ms omp/2: 37.4041ms omp/3: 53.051ms
10 000 000	serial: 0.138132ms sse: 0.134724ms omp/1: 2.50084ms omp/2: 3.70412ms omp/3: 4.84247ms	serial: 0.985822ms sse: 1.02092ms omp/1: 12.2265ms omp/2: 18.1669ms omp/3: 24.4175ms	serial: 2.09287ms sse: 1.98684ms omp/1: 24.5924ms omp/2: 36.4125ms omp/3: 48.1161ms

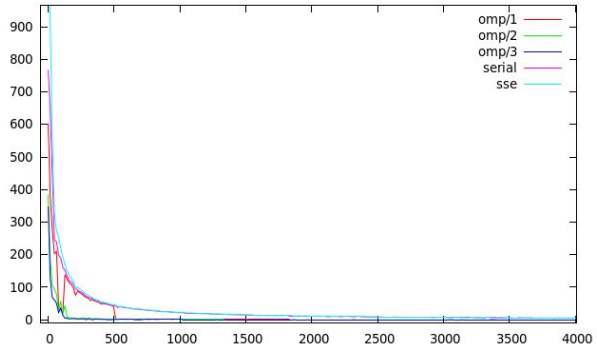
We can tell from these results than increasing N does not make a huge difference in processing time; nevertheless, adding bins slows it greatly, getting around 600% increase between 1 000 000 and 5 000 000 bins.
We also have that serial is actually faster than omp and omp gets slower with more threads; though sse time starts by being similar to serial time, but with 10 000 000 bins sse is faster.

The code **/histogram/core/histogram_bench.cpp** has a function called hist_bench_all which calls all three possible benches : serial, sse and omp; omp does extra because it calls the function max_thread_nb times. Each one of them calls hist_bench with different value for constant f (compute_[serial/sse/omp] which are defined in **/histogram/core/histogram.cpp**), in this function, gets two timestamps NTIMES (defined as 50), before and after calling f, and gets the difference to get an average value of the process time.

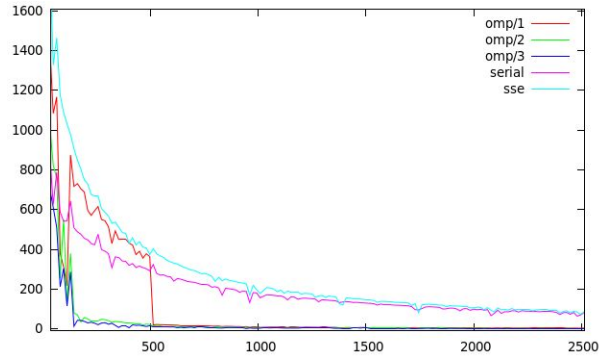
2) Running code in **/histogram/bench_plot** with command **time ./benchs_plot --gen-uniform** N, where N is size of the interval.



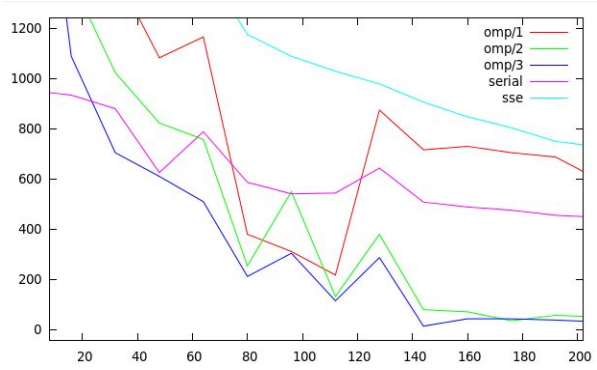
N=10
real 2m23.354s
user 2m0.388s
sys 2m50.051s



N=100
real 2m7.309s
user 1m43.258s
sys 2m36.122s



N=1000
real 2m17.630s
user 1m52.599s
sys 2m43.214s



Zoom of N=1000

THE END
We thank you for the really cool TP ! :)