M2-BIG DATA
High Performance Computing
Profiling

Lab 2 - OpenMP

Réalisé par :
Gaby Maroun

Encadré par :
Dr. Etancelin JM

11 janvier 2021

Exercise 1 : Matrix-matrix product

Part 1 : Sequential program .

The program written and work sequentially in the function productMat_seq while the role of the function initMat is to randomly initialize new matrices as a 1D array to be multiplied.

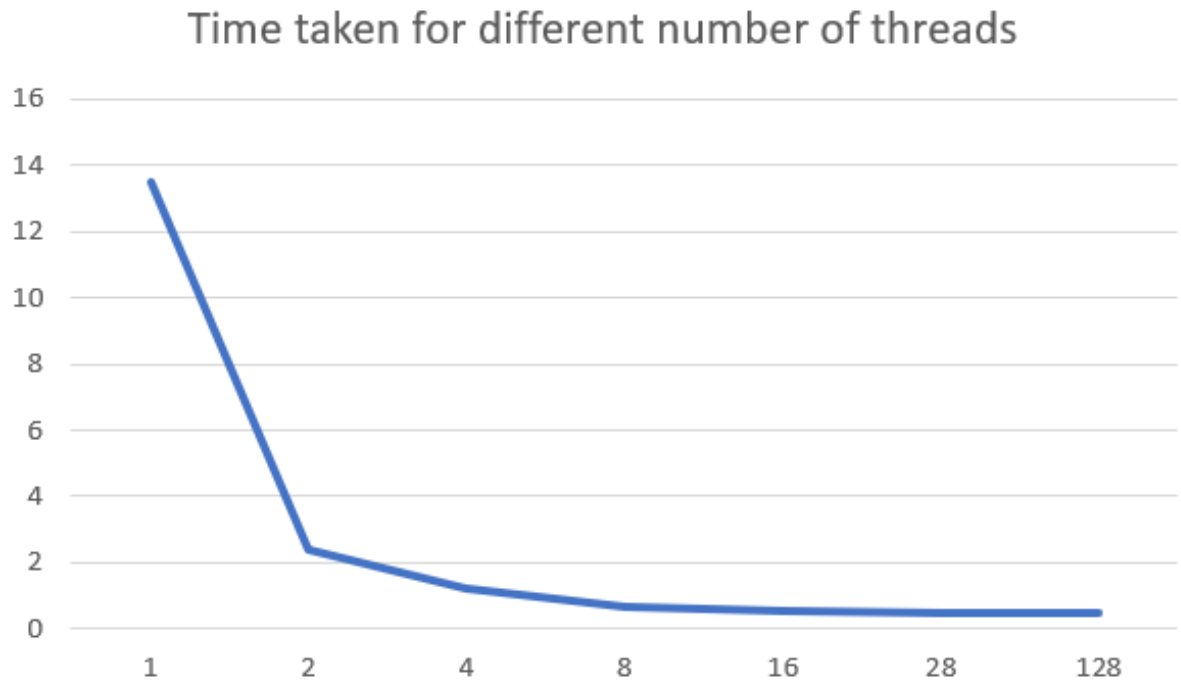
Part 2 : OpenMP parallelization

- The multiplication of the 2 matrices is the function to be parallelized as we can assume that this is the most time and memory consuming part of the program.
  As seen in the courses, I opted to use the pragma omp parallel with these different options :
  - default(none) recommended as it will prevent errors due to behavior.
  - shared(a,b,c,part_rows) a,b and c represent the 2 multiplied matrices and their result respectively and they're shared in the memory to synchronize the calculations between the threads while the part_rows variable is shared to be used in the guided parallelization of the for loop
  - The pragma omp for schedule(guided, part_rows) is used as it decomposes the tasks, as indicated by the part_rows, decreasingly to organize the work between the threads and make them start with the lowest work demanding task.
- The omp_get_wtime() is surrounding the multiplication code as suggested to measure the average computational time.

— I performed a strong scalability study for matrices of size n = 1000 on different number of threads :

Threads	Time
1	13.468
2	2.4
4	1.196
8	0.638
16	0.547
28	0.498
128	0.466

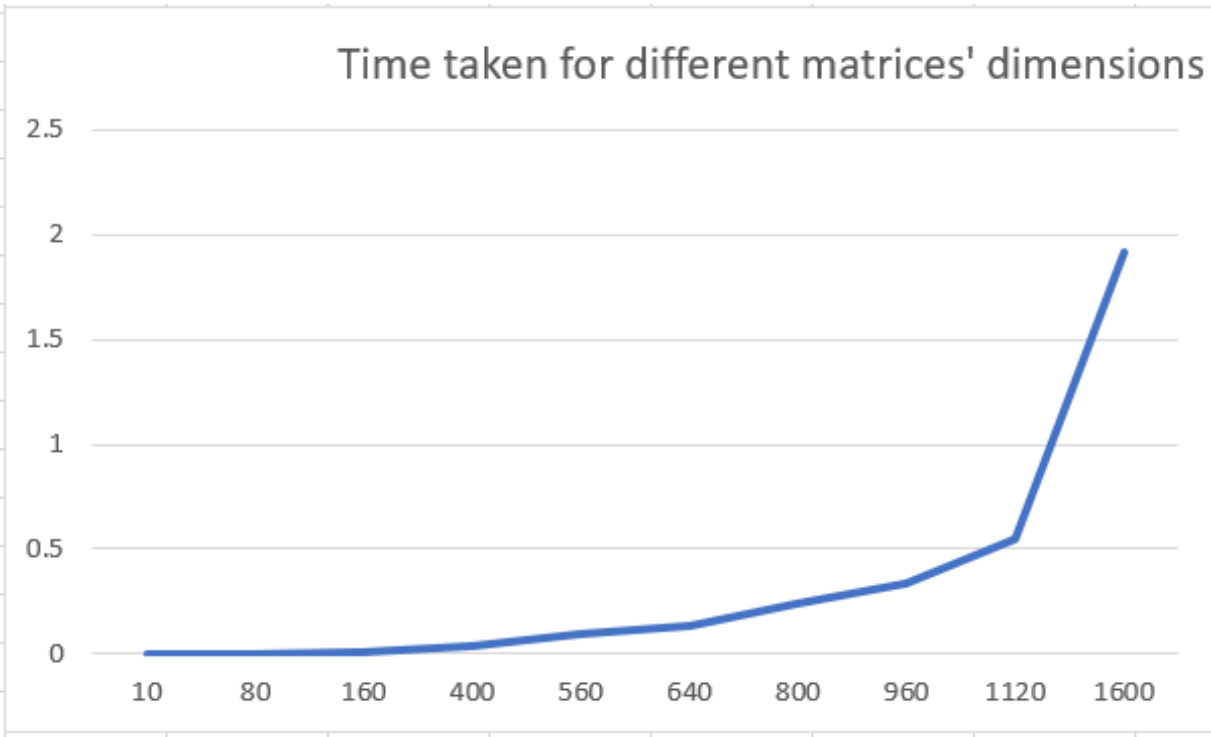
Dim=1000



We can assume that 28 threads is the best choice to be used costing way less time than all the others.

— To study the performance, plotting the computational time against the matrices size and while using 28 threads as deduced from the previous point, for the following sizes came as follow :

Dimension	Time
10	0.0005
80	0.0025
160	0.0053
400	0.0365
560	0.0936
640	0.1313
800	0.2384
960	0.3381
1120	0.5465
1600	1.9183



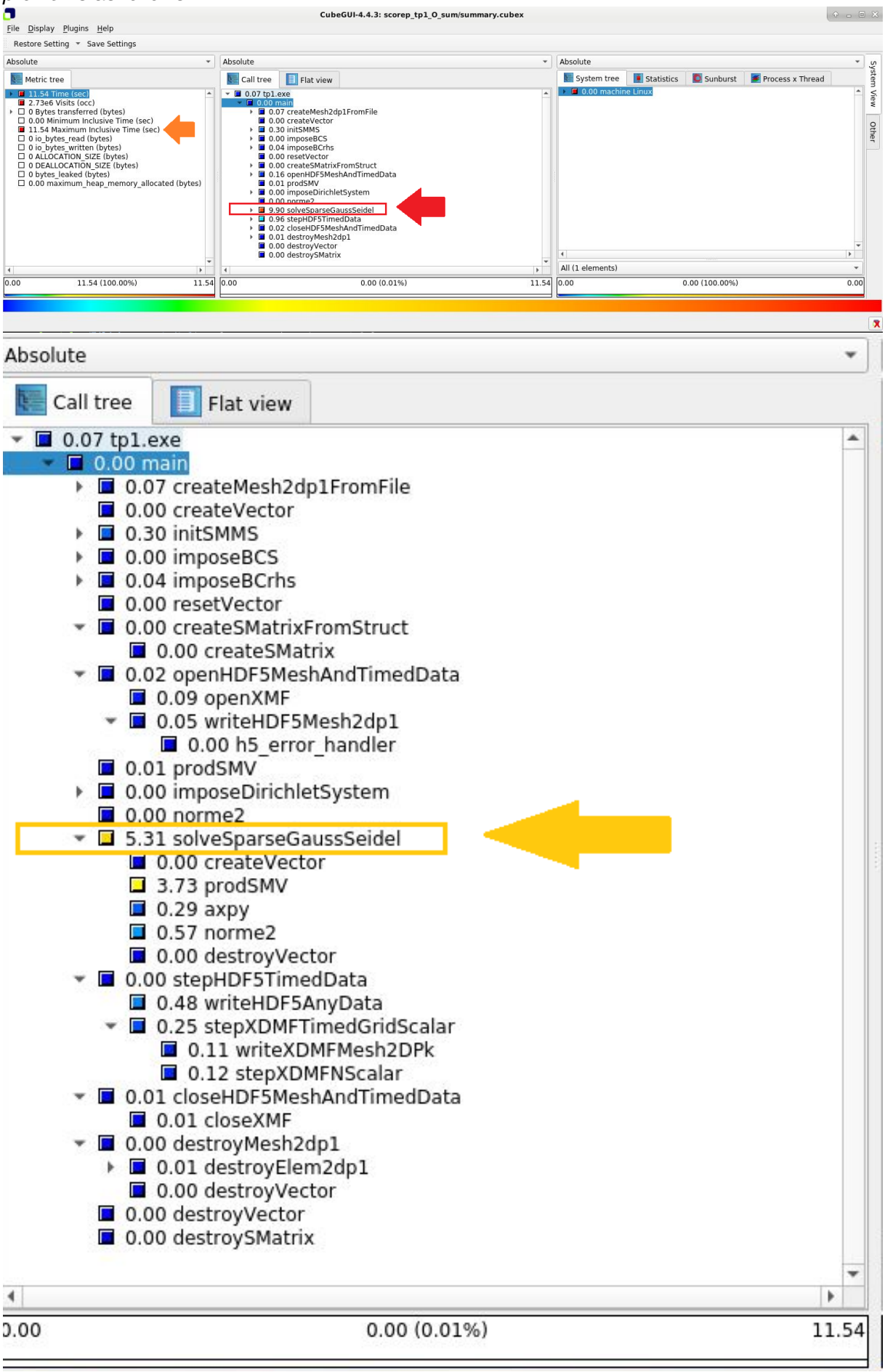
We can assume that by increasing the dimension of the matrice, the time of the multiplication increase that means the computational time is proportional with the theoretical complexity.

Exercice 2 : Parallelization of a real code

1. Recall the computational time for sequential code (1st practical session) from scalasca profiler.

As seen in the previous tp, the computational time for the sequential code from scalasca

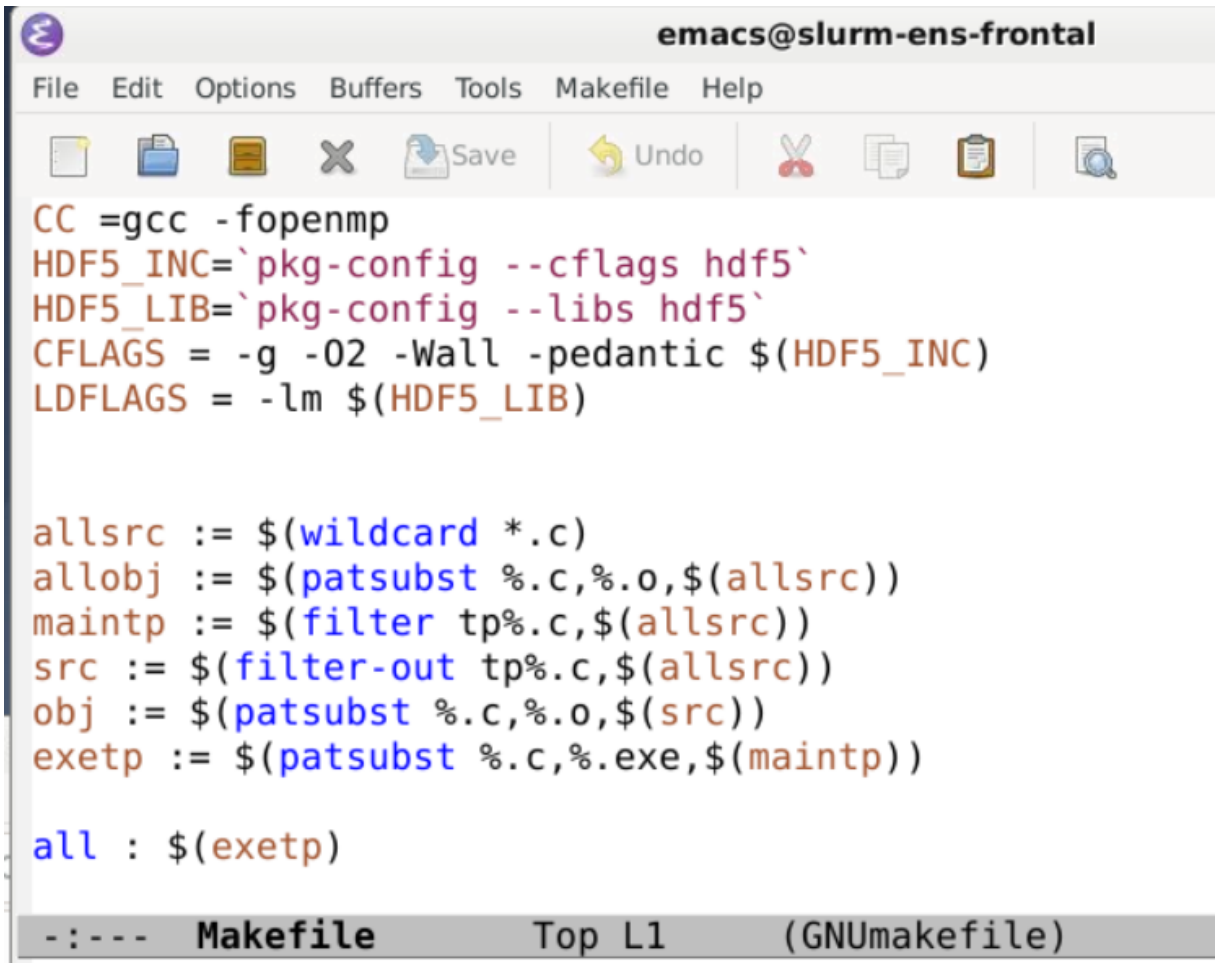
profiler is as follows :



2. Find one or several loops to parallelize, then parallelize with OpenMP. You must justify your parallelization choices.

   As we can deduce, the focus of parallelization should be on the solveSparseGaussSeidel function, specifically the prodSMV.

```
int k=0;
vector r = createVector(A.n);
vector AdiagInv = createVector(A.n);
// A diagonal computation
for (int i = 0; i < A.n; i++) {
  for (int nz = 0; nz < A.nnzi[i+1]-A.nnzi[i]; nz++) {
    int j = A.j[A.nnzi[i]+nz];
    if(i==j) AdiagInv.data[i] = 1.0/A.data[A.nnzi[i]+nz];
  }
}
// r initial
    #pragma omp parallel default(none) shared(A, x, r, b, k,
tol,AdiagInv)
prodSMV(A, x, r);
axpy(-1.0, r, b, r);
while(k<MAXITER && norme2(r)>tol) {
#pragma omp for schedule(static)
  for (int i=0; i < A.n; i++) {
    // Compute new x
    x.data[i] = b.data[i];
    for (int nz = 0; nz < A.nnzi[i+1]-A.nnzi[i]; nz++) {
      int j = A.j[A.nnzi[i]+nz];
      if(i!=j)
        x.data[i] -= A.data[A.nnzi[i]+nz]*x.data[j];
    }
    x.data[i] *= AdiagInv.data[i];
```



To parallelize, I used the famous pragma omp parallel alongside

— default(none) recommended as it will prevent errors due to behavior.

— shared(A, x, r, b, k, tol, AdiagInv) These variables are shared in the memory to synchronize their calculations between the threads.

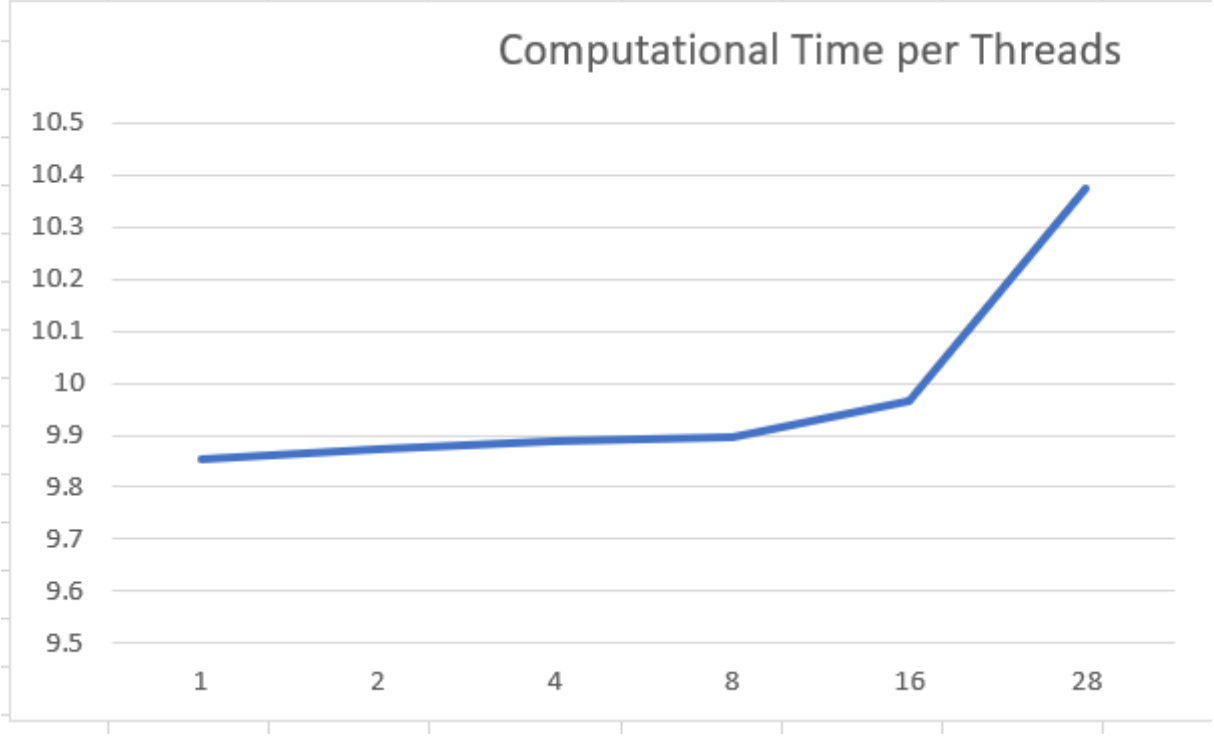
— The pragma omp for schedule(static) is used to parallelize the for loop

I would also like to mention that I tried to parallelize the Jacobi function but that didn't affect the computational time neither.

3. Perform a strong scalability study on Bilbo cluster from 1 to 28 threads for the new version of the code

   The improvement expected was never found. I spent so much time trying to make it work but nothing appeared to be the right way. Increasing the number of threads didn't help, instead it made it worse. I tried to implement many ways to parallelize the code but nothing seemed to be right.

Threads	Real time	User time	System time
1	9.855	9.776s	0.020s
2	9.874	12.856s	0.020s
4	9.887	18.889s	0.008s
8	9.897	30.952s	0.012s
16	9.967	56.178s	0.016s
28	10.373	1m34.693s	0.064s



4. Use scalasca again on your OpenMP code. What are the performances for the function that you changed ? Mind that sum of threads times are displayed on the graphical interface of Scalsca. Observe the statistics pannel (right sub-window).
   As seen in the previous question, the parallelization didn't work, so I couldn't answer this question.
5. Explain what could be further optimized.
   As seen in the previous answers, the parallelization didn't work, so it wasn't clear for me what could've been the next possible function to be optimized.

La fin.