Advanced Computer Architecture

Parallel Programming Project

Optimized matrix multiplication and inversion

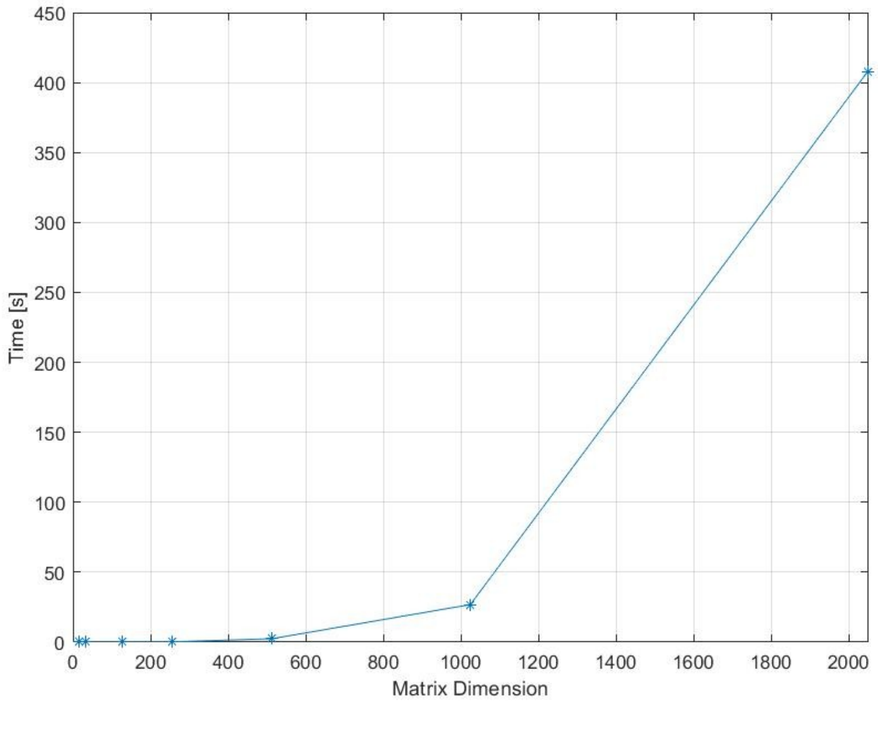
Cotogni Marco 470612

1) Analysis of the serial algorithm

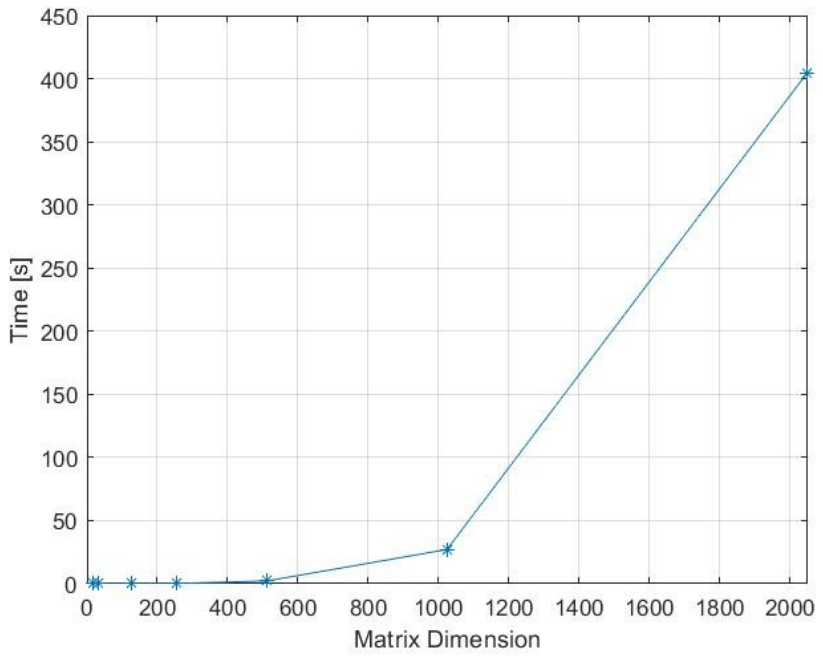
Matrix Multiplication

Matrix multiplication is a very simple problem when the matrices used are small. With the rapid expansion of the memories dimensions and the increase of the processors computing power, the users can be able to treat matrices with big dimensions. This problem is common in fields like Machine Learning, Data Analysis, Differential Equation, Network Theory and so on..

My First approach implements the Naive Product between matrices (row-column product). This method is the most used when the matrices are small and, in this case, it works fast. In fact, when the matrices' dimensions are lesser than 100, it results faster than the other methods. This method allows the users to use matrices of any dimensions, there isn't a limit on the scalability. The only constrain is to respect the dimensions rules: Matrix A: n*m; Matrix B: m*l; Matrix C: n*l. To avoid the problem of inexactly dimensions I considered only square matrices with same dimensions. In the file "Mult.c" I used a contiguous allocation of the array to avoid jump inside the memory because a serial implementation without using a contiguous allocation makes a big number of cache misses and the performances are worse. In this method the heaviest part is the computation of matrix C.



The Second Approach is the Naive product between one matrix and the transposition of the second one (the fundamental hypothesis is the symmetry of matrix B) I chose this method because it allows the vectorization of the inner loop (of the function row_column). Performances are better than the Naive Product results when the dimensions increase. This method allows to reduce the number of the cache miss when the matrices became large. The transpose function is very fast and also for this method the heaviest part is the computation of the matrix C.



The Third Approach is the Strassen Method: this Method works with Square matrices with dimensions 2^n*2^n. There isn’t any constrain on the scalability. If the dimension aren’t 2^n*2^n the method doesn’t work.
The Matrices A,B and C are divided in 4 sub matrices, this method avoid the matrix multiplication between big matrices. The number of multiplications are limited (and reserved to matrix with dimensions (2^n)/4) because it’s better to use sum and subtraction between matrices. This is the explication of the method (https://it.wikipedia.org/wiki/Algoritmo_di_Strassen):

$$\mathbf{A} =: \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix}, \mathbf{B} =: \begin{pmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{pmatrix}, \mathbf{C} =: \begin{pmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{pmatrix}$$

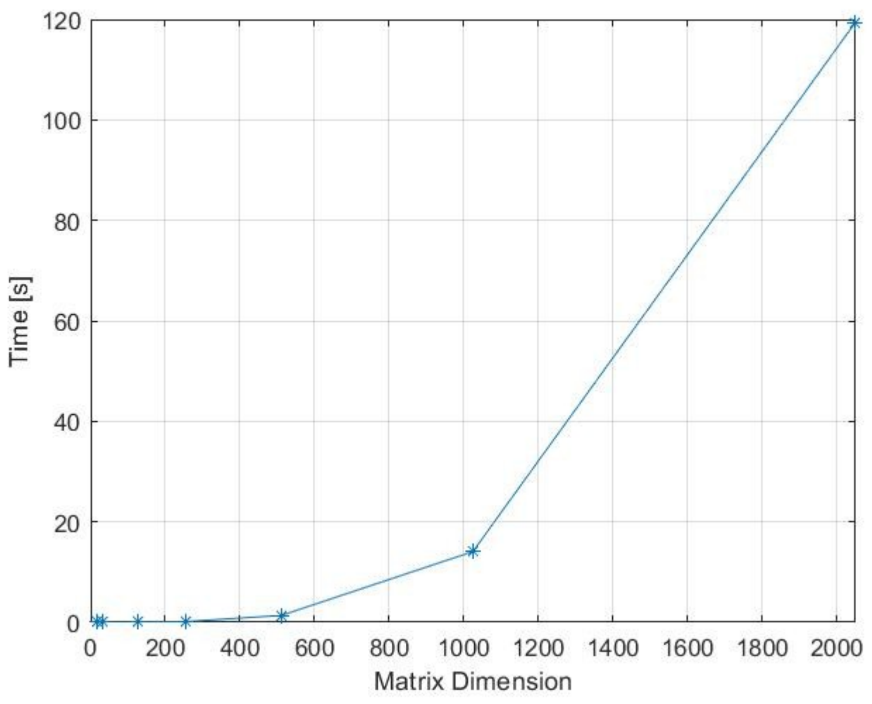
After the division of the matrices in sub matrices, simple operations among matrices with small dimension are computed.

$$\begin{aligned}
\mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\
\mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\
\mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\
\mathbf{M}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
\mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\
\mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\
\mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})
\end{aligned}$$

Now M matrices are used to compute the result Matrix C:

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}$$

Now the matrix C is computed assigning the sub matrices to it.
This Method presents good performances because reducing the number of multiplication(which is the more expensive operation) the code run faster.



Time in seconds

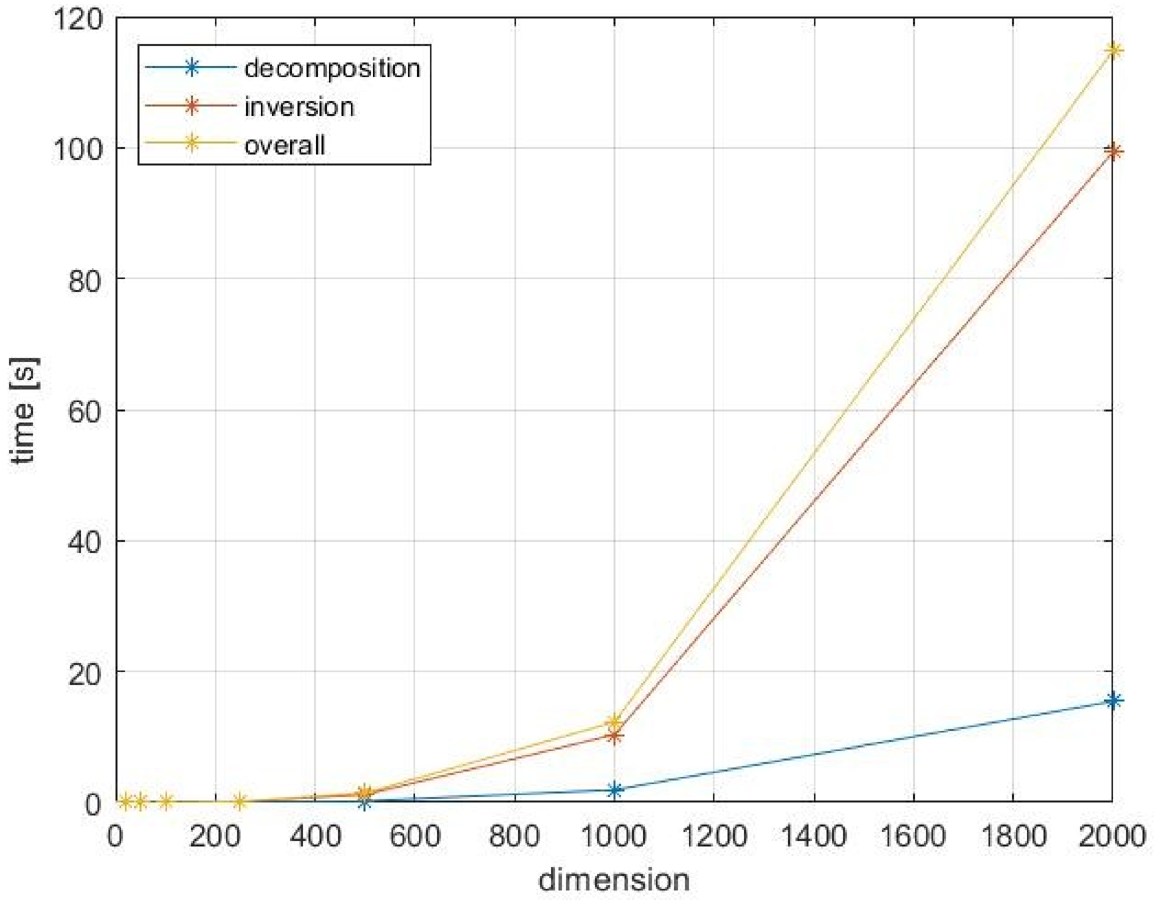
Method/DIM	16	32	128	256	512	1024	2048
Naive	0.00013	0.000777	0.030307	0.20951	2.2421	26.8426	407.8543
Transpose	0.000143	0.000804	0.022266	0.20422	2.2415	27.0876	404.3527
Strassen	0.000237	0.000870	0.019129	0.14692	1.3708	14.0416	119.2447

Analyzing the graphs and the tables is possible to say that the best performances are obtained with the Strassen Method, because the number of multiplications is reduced and making sum and subtraction instead of products makes the code faster (the function which implements sum and subtractions is the heaviest part of the Strassen method: 99% of total time).

Matrix Inversion

The Matrices inversion is a not trivial problem because the risk of working with big matrices is to enter in an infinite loop.
The Method I used is called matrix inversion with LU decomposition. This method, studied during the Numerical Methods course, allows to reduce the computation of the inverse matrix with a trick: make the matrix to invert, simpler. The method is very simple giving a matrix A, decompose it in the Matrix L and U which are Lower triangular and upper triangular matrices. The method I used to implement them, is a variant of the classic LU, it is called LUP because there is a new function called pivoting. The pivoting variant allows to decrease the percentage of error by doing the decomposition. The pivoting consists in checking which is the row with the bigger element, starting from the first column, and making an exchange between rows to bring the selected row on the top of the matrix. The results of the decomposition can be verified making the product P*A = L*U. After making the LUP decomposition, the next step is the inversion of the decomposed Matrix. The inversion can’t be realized with the Cramer method (computing the determinant) because the time already explodes with small matrices (20x20). So working on the L and U matrices allows the inversion in an observable time. For this method I considered a contiguous memory allocation to decrease the number

of cache misses. In the file "LU_Pivoting.c" is possible to find the implementation (setting threads number =1). The biggest fraction of the run time is occupied by the inversion, the decomposition is very fast.
CodeSource:
https://en.wikipedia.org/wiki/LU_decomposition#C_code_examples.



Time in seconds

Fun/DIM	20	50	100	250	500	1000	2000
Decomp	0.000054	0.000807	0.006276	0.037432	0.241714	1.9260	15.44778
Inv	0.000222	0.003625	0.011805	0.143509	1.202079	10.3434	99.3587
Overall	0.000314	0.004461	0.018096	0.180955	1.443806	12.269518	114.80656

From the data we can say that the method is faster with small dimensions and obviously the time increases with bigger matrices but it doesn't diverge (like Cramer method). Tests Made on Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz.

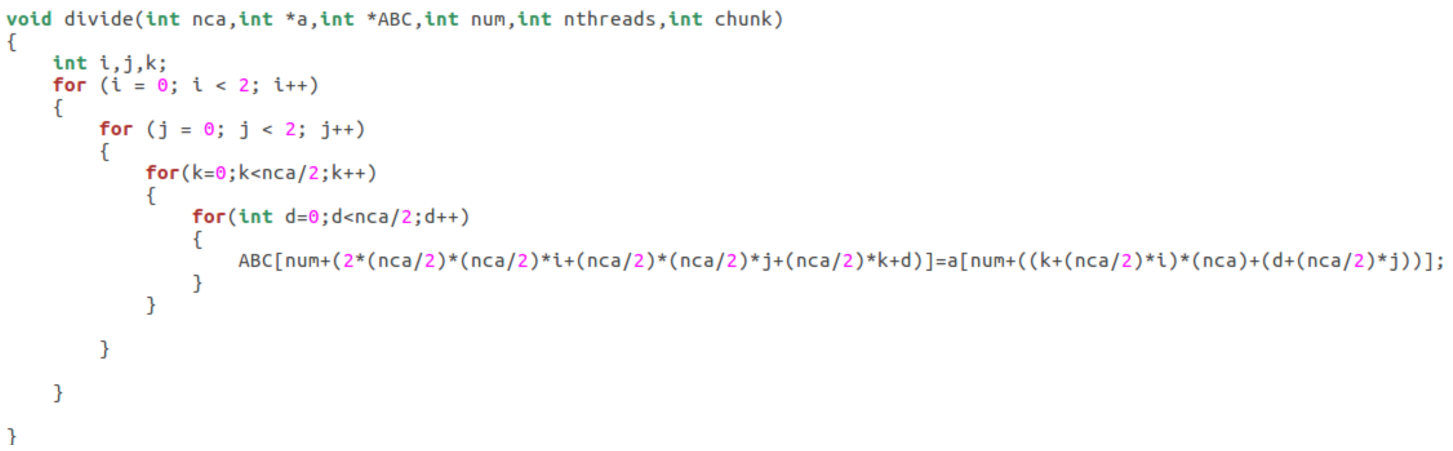
2) A-Priori study of the available parallelism

Matrix Multiplication

For the Naive Product the heaviest part of the computation is the triple for that performs the Matrix C making the Row-column product between the matrices A and B. This is the block to parallelize and to allow a full parallelism, the contiguous allocation of the Matrices is perfect, because the matrices elements are contiguous and in this way the “ jumps” inside the memory to find the elements are avoided. In fact these “Jumps” kill parallelism so it is impossible to improve the speedup without considering this problem.

The Second method allows a double parallelization because it uses the row-column function explained before but it is also possible to parallelize the Matrix Transposition. To do that I used a Temp matrix called bT in order to not lose matrix elements while the operation is running. This matrix is not allocated with matrix A because it exists only in the function, so I chose to avoid a Realloc and I preferred to allocate the matrix bT alone.

The Strassen Method is full of functions which can be parallelized, starting from the function division (which divide in sub matrices A,B,C). The Arithmetic functions like sum, subtraction and product are the major fraction of the code and they require most of the execution time. In fact, make a parallel version of the math function allows the speedup. The functions Divide and Assign take less than 1% of the execution time, but I decided to parallelize them,too.

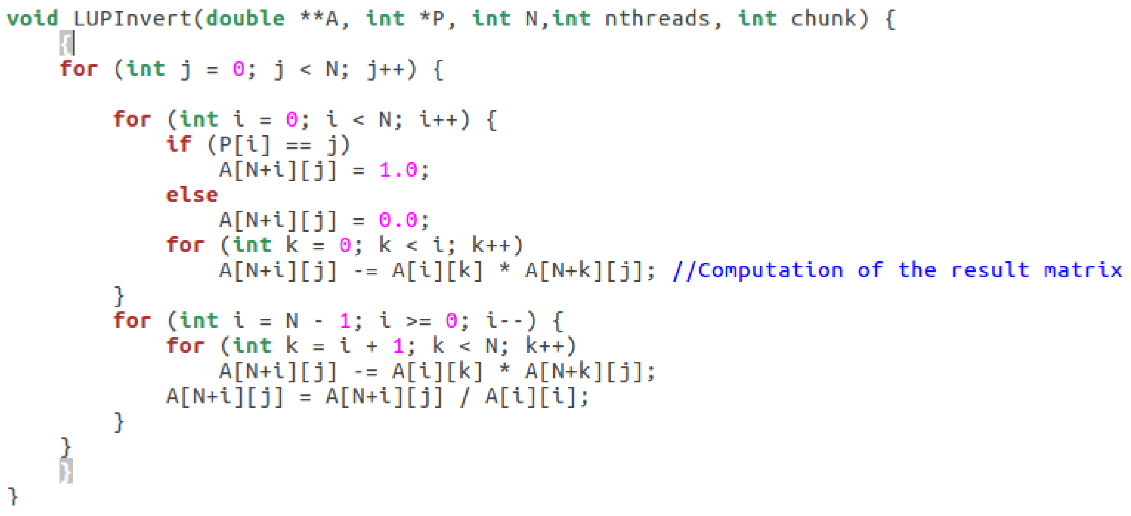


```
void divide(int nca,int *a,int *ABC,int num,int nthreads,int chunk)
{
    int i,j,k;
    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
        {
            for(k=0;k<nca/2;k++)
            {
                for(int d=0;d<nca/2;d++)
                {
                    ABC[num+(2*(nca/2)*(nca/2)*i+(nca/2)*(nca/2)*j+(nca/2)*k+d)]=a[num+((k+(nca/2)*i)*(nca)+(d+(nca/2)*j))];
                }
            }

        }

    }

}
```

So the most important thing in the matrix multiplication, is not only the way we apply the parallelism but is also the organization of the matrices in the memory. That because with large matrices, having a long array is better than looking for all the elements allocated randomly.

Matrix Inversion

For the Matrix inversion the method used is fundamental. The Matrices can be allocated in any way but if the the method have an high computational cost, the time diverges with the increasing of the matrices dimensions. So, in the inversion I tried to parallelize in the best way paying attention to the critical part of the code. So ,I decided to not parallelize the matrix Decomposition (LUP) because the method is already optimized and using parallel region and tasks would have filled the code with critical region and taskwait which introduce only overhead and have worse performances than the serial implementation (of course on the decomposition part). This decision was taken analyzing the code: it's simple to see how there are a lot of loop carried dependences and a lot of "swap": these blocks of code must be executed by one thread/task at time so it's difficult to parallelize this part and having correct decomposition at the same time. Adding taskwait and critical region is like making a barrier and all the tasks must wait the others to go on in the code. I decided to parallelize the Matrix inversion (the second function in the code)



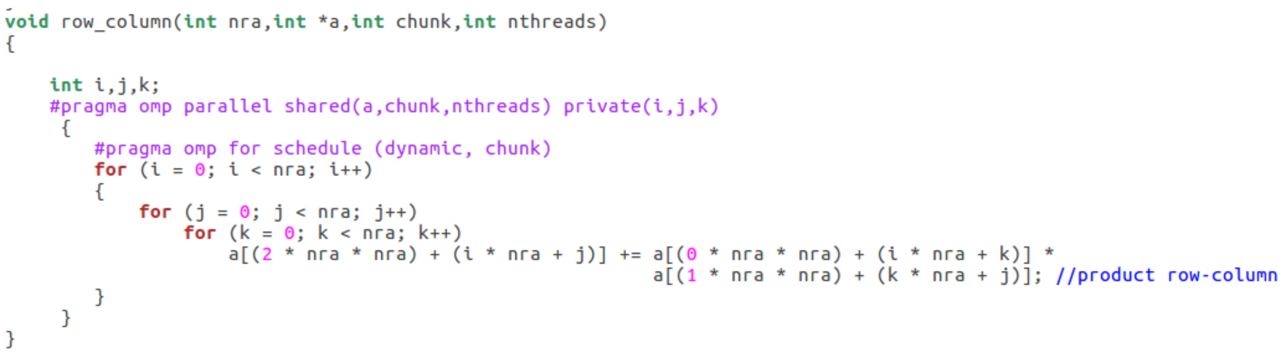
```
void LUPInvert(double **A, int *P, int N,int nthreads, int chunk) {
    {
    for (int j = 0; j < N; j++) {

        for (int i = 0; i < N; i++) {
            if (P[i] == j)
                A[N+i][j] = 1.0;
            else
                A[N+i][j] = 0.0;
            for (int k = 0; k < i; k++)
                A[N+i][j] -= A[i][k] * A[N+k][j]; //Computation of the result matrix
        }
        for (int i = N - 1; i >= 0; i--) {
            for (int k = i + 1; k < N; k++)
                A[N+i][j] -= A[i][k] * A[N+k][j];
            A[N+i][j] = A[N+i][j] / A[i][i];
        }
    }
    }
}
```

This function required the 84% of the total runtime to be executed, so parallelizing them there is an increase of the performances. I developed two versions of the parallel code: the first one using the parallel for and the second one experimenting tasks. The problem of matrix inversion is not easy to solve and I didn't expect "super" results because the matrices are quite regular and don't allow to work with the maximum of the potentiality on the granularity (it's the same for each thread or task).

3) OpenMp parallel implementation

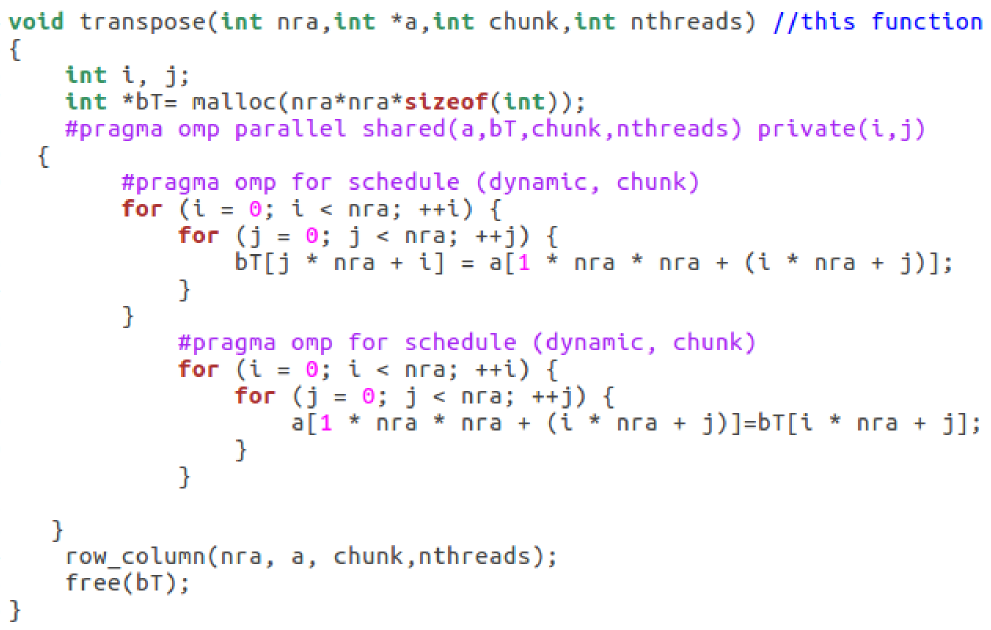
Matrix Multiplication

For the Naive implementation the parallelization algorithm is:



```
void row_column(int nra,int *a,int chunk,int nthreads)
{

    int i,j,k;
    #pragma omp parallel shared(a,chunk,nthreads) private(i,j,k)
     {
        #pragma omp for schedule (dynamic, chunk)
        for (i = 0; i < nra; i++)
        {
            for (j = 0; j < nra; j++)
                for (k = 0; k < nra; k++)
                    a[(2 * nra * nra) + (i * nra + j)] += a[(0 * nra * nra) + (i * nra + k)] *
                                                         a[(1 * nra * nra) + (k * nra + j)]; //product row-column
        }
     }
}
```

It's possible to notice the shared region and the for but also the implementation of the contiguous array, in fact, the matrices A,B and C are allocated with the name A with a distance n*n in terms of elements.
The Transpose method use this kind of parallelization:

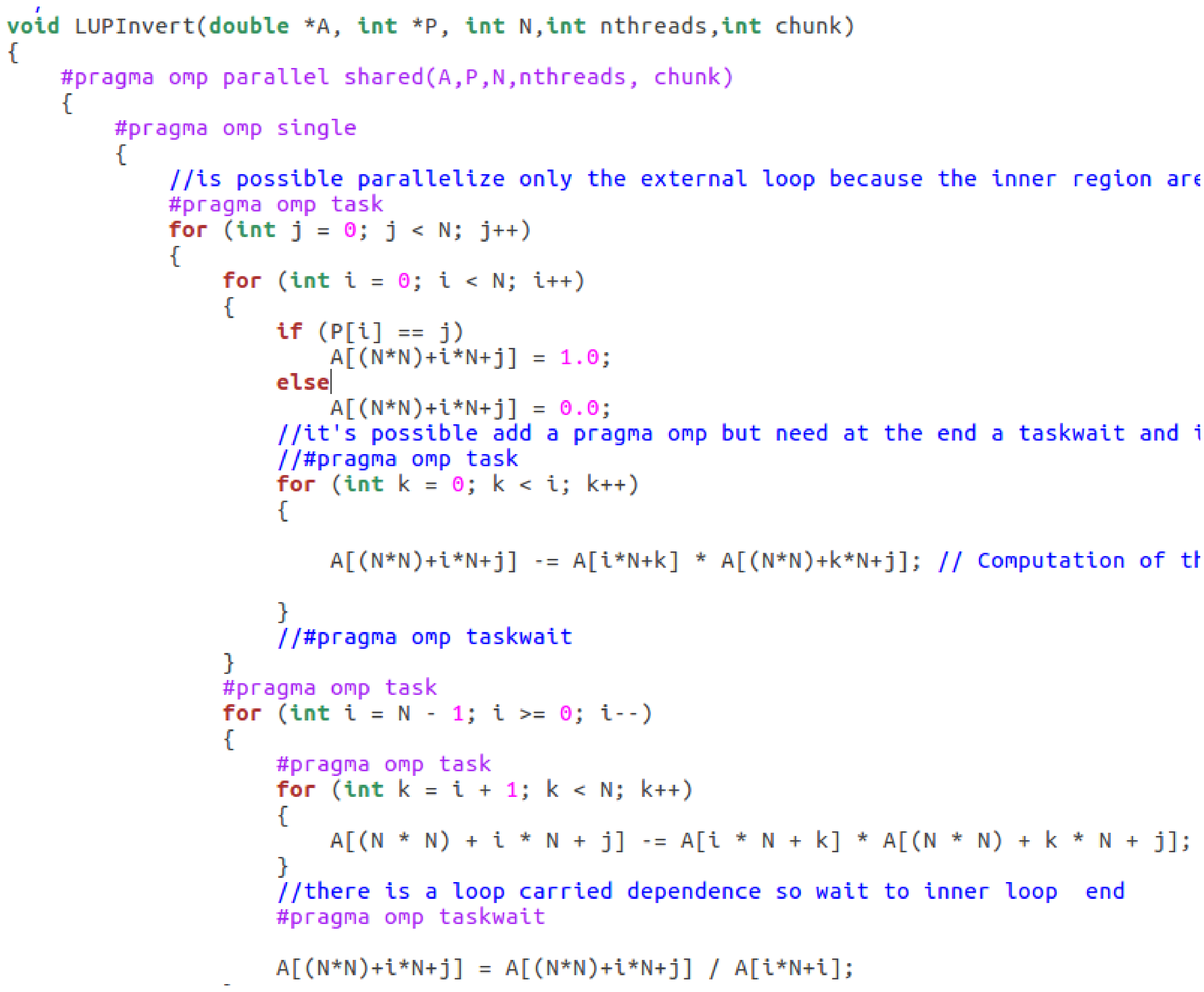


```
void transpose(int nra,int *a,int chunk,int nthreads) //this function
{
    int i, j;
    int *bT= malloc(nra*nra*sizeof(int));
    #pragma omp parallel shared(a,bT,chunk,nthreads) private(i,j)
  {
        #pragma omp for schedule (dynamic, chunk)
        for (i = 0; i < nra; ++i) {
            for (j = 0; j < nra; ++j) {
                bT[j * nra + i] = a[1 * nra * nra + (i * nra + j)];
            }
        }
            #pragma omp for schedule (dynamic, chunk)
            for (i = 0; i < nra; ++i) {
                for (j = 0; j < nra; ++j) {
                    a[1 * nra * nra + (i * nra + j)]=bT[i * nra + j];
                }
            }

   }
    row_column(nra, a, chunk,nthreads);
    free(bT);
}
```

For the Strassen method, all the functions are parallelized to avoid inserting useless list of code (please check the file “Mult.c” and the Strassen function in particular). All the algorithm is parallelized with parallel region and parallel for.

Matrix Inversion

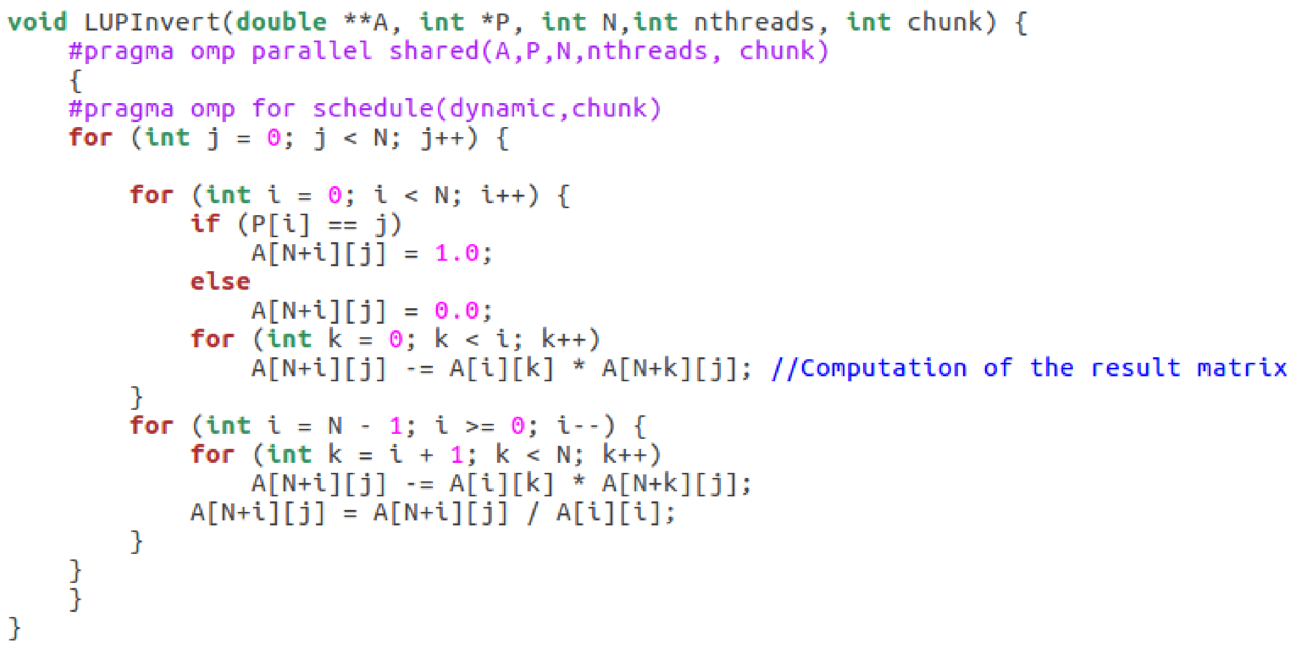
In the previous section I explained the reason why the decomposition region is not parallelizable, now I want to analyze the Task approach and the Parallel for approach.
For the first one:



```
void LUPInvert(double *A, int *P, int N,int nthreads,int chunk)
{
    #pragma omp parallel shared(A,P,N,nthreads, chunk)
    {
        #pragma omp single
        {
            //is possible parallelize only the external loop because the inner region are
            #pragma omp task
            for (int j = 0; j < N; j++)
            {
                for (int i = 0; i < N; i++)
                {
                    if (P[i] == j)
                        A[(N*N)+i*N+j] = 1.0;
                    else
                        A[(N*N)+i*N+j] = 0.0;
                    //it's possible add a pragma omp but need at the end a taskwait and i
                    //#pragma omp task
                    for (int k = 0; k < i; k++)
                    {

                        A[(N*N)+i*N+j] -= A[i*N+k] * A[(N*N)+k*N+j]; // Computation of th

                    }
                    //#pragma omp taskwait
                }
                #pragma omp task
                for (int i = N - 1; i >= 0; i--)
                {
                    #pragma omp task
                    for (int k = i + 1; k < N; k++)
                    {
                        A[(N * N) + i * N + j] -= A[i * N + k] * A[(N * N) + k * N + j];
                    }
                    //there is a loop carried dependence so wait to inner loop  end
                    #pragma omp taskwait

                    A[(N*N)+i*N+j] = A[(N*N)+i*N+j] / A[i*N+i];
```

I defined the parallel region, then, to apply the tasks, I needed to define a single region. Inside this section there are 3 task regions. I created tasks for the outer big loop then, inside it, there is a loop which can be parallelized but it needs a taskwait (to make correct computation) at the end and this wastes time. So, I didn't parallelize this loop. The other loop can be easily parallelized and the inner too but it requests a task wait, if I didn't use it in the computation it would have been wrong. I have also tried to put conditions on the task dimension with the directive "#pragma omp task if" but it increases only the execution time because the matrix is a regular structure. Working on the granularity and creating tasks only when the index of the iterations is big or small adds only overhead, in this case.

The second approach is with Parallel for:



```
void LUPInvert(double **A, int *P, int N,int nthreads, int chunk) {
    #pragma omp parallel shared(A,P,N,nthreads, chunk)
    {
    #pragma omp for schedule(dynamic,chunk)
    for (int j = 0; j < N; j++) {

        for (int i = 0; i < N; i++) {
            if (P[i] == j)
                A[N+i][j] = 1.0;
            else
                A[N+i][j] = 0.0;
            for (int k = 0; k < i; k++)
                A[N+i][j] -= A[i][k] * A[N+k][j]; //Computation of the result matrix
        }
        for (int i = N - 1; i >= 0; i--) {
            for (int k = i + 1; k < N; k++)
                A[N+i][j] -= A[i][k] * A[N+k][j];
            A[N+i][j] = A[N+i][j] / A[i][i];
        }
    }
    }
}
```

This Method requires only a big parallel for (the external), it's also possible to do a collapse but the code is not faster because the last internal loop requires a critical region. That because every thread must work alone on this part. It is possible to experiment other parallel inner region but paying attention to do the computations right (adding critical region inside).

4)Test and Debugging

The First part developed was the serial matrix multiplications with no contiguous allocation of memory, then when I saw the speedup didn't increase I decided to make a study of the overall code. I decided to work with the memory allocation and after doing that I tried with bigger matrices, from 2x2 to 2048x2048. It is possible to

go on but the time to compute every matrices product increases with a computational cost of n^3. After the implementation of this part I worked on the transposition, but starting from the Naive product (with a contiguous allocation). For the Strassen method I started with the non contiguous memory allocation, but when the time didn't decrease I chose to create three big arrays in order to have all the elements near to each others. After the implementation of the serial version, I developed the parallel version of the three methods and made tests with matrices having dimensions up to 2048x2048. Then, I used the Google Compute Engine to create a virtual machine with a number of CPU up to 24 (Limit of the region).
For the Matrix inversion I started with the Cramer method but it was inefficient (with matrix 20x20 the time explodes). So I implemented the LUP Method for matrix multiplications (serial version) with non contiguous allocation memory. I passed to the Contiguous allocation version and to the parallel version (with and without tasks). The tests are made with dimensions up to 2000x2000 but it is possible to go on (I decided to stop with 2000x2000 because the time explodes). After tried that, I experimented the Google Compute Engine to make tests with a variable number of cpu (max 24 limit region) and with matrices up to 2000x2000.

5)Performance Analysis

It's possible to analyze the Theoretical speedup that can be obtained applying the Amdhal's law for every method:

$$s=\frac{n}{n+(1-n)f}$$

SPEEDUP

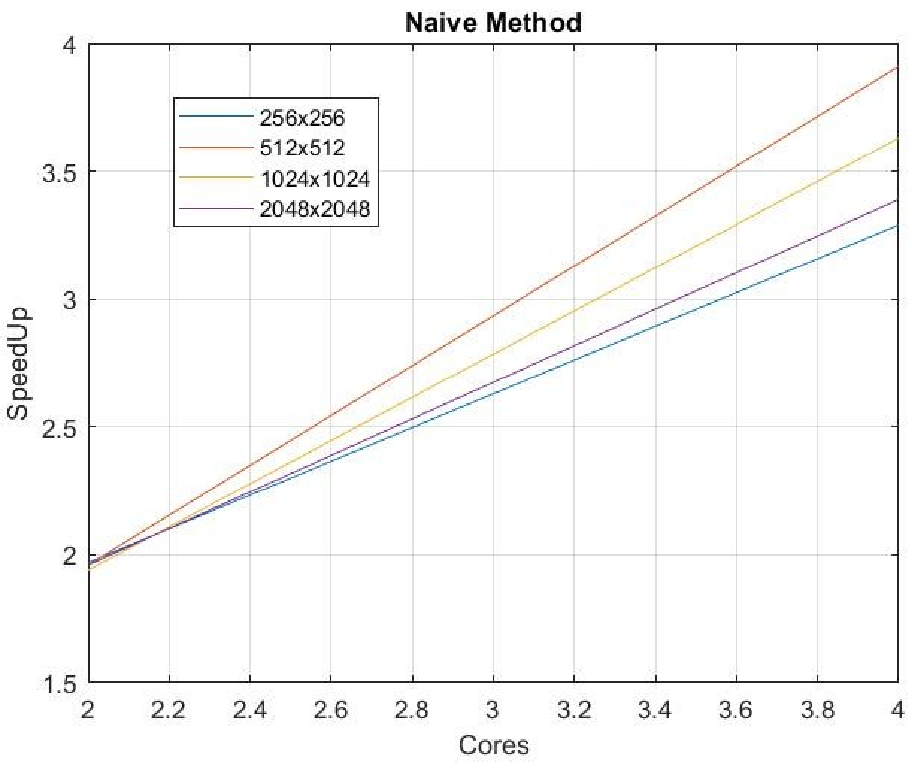
	fraction	n=2	n=4	n=8	n=16	n=24
Naive	99.99%	1.99	3.99	7.99	15.99	23.99
Transpose	99.99%	1.99	3.99	7.99	15.99	23.99
Strassen	99.62%	1.99	3.95	7.79	15.13	22.07
Inverse	84,43%	1.73	2.72	3.82	4.79	5.23

The local tests were made with a Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz with 4 physical cores and 8 logical,but it is useless using 8 threads to parallelize (in a super computer architecture the hyperthreading is switched off because when all the cores aren't underutilized is useless creating more threads to be assigned to a full working core, it adds only overhead. For example using 8 threads with a 512x512 matrix takes 0.3532s (Strassen) while using 4 threads need 0.324s).

Local Tests:

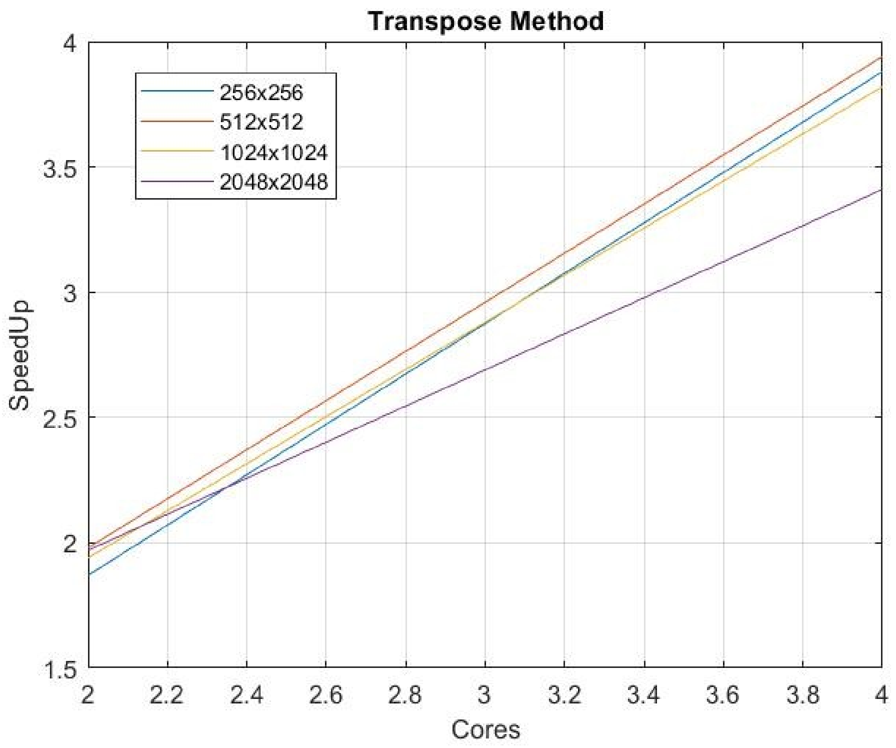
Time in seconds

		Serial	2-Core	4-Core	SpeedUp	
Naive	256x256	0.20951	0.105968	0.063511	1.97	3.29
	512x512	2.2421	1.138391	0.56966	1.96	3.91
	1024x1024	26.8426	13.798627	7.37704	1.94	3.63
	2048x2048	407.8543	206.65641	120.1093	1.96	3.39



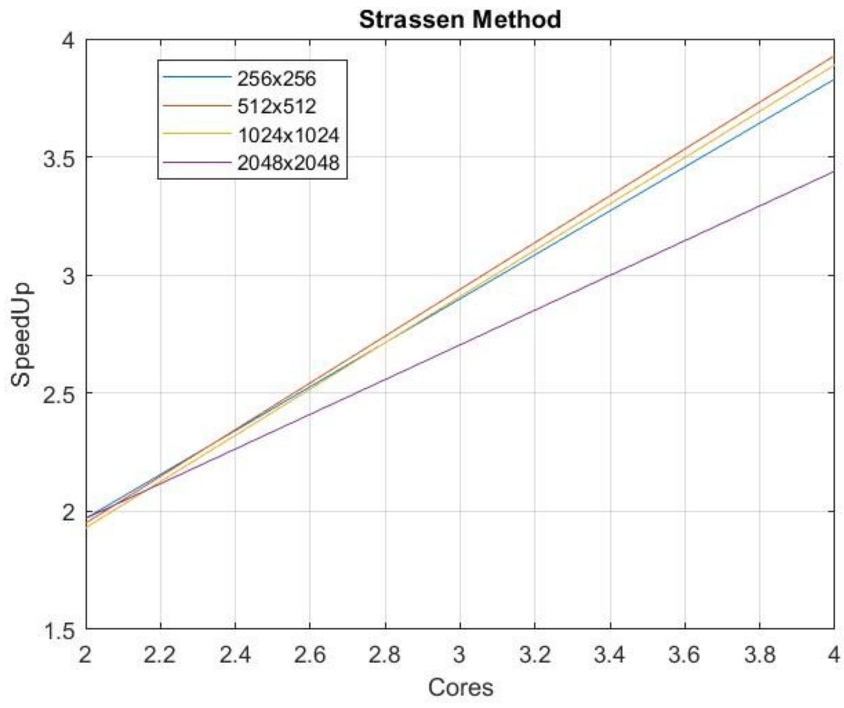
From the analysis of the tables we can observe the Amdhal's law is perfectly respected in fact it's difficult to increase the speed up because the values are very near to the limit.

		Serial	2-Core	4-Core	SpeedUp	
Transpose	256x256	0.20422	0.10867	0.052623	1.87	3.88
	512x512	2.2415	1.127684	0.568045	1.98	3.94
	1024x1024	27.0876	13.9589	7.0876	1.94	3.82
	2048x2048	404.3527	204.40883	118.3484	1.97	3.41



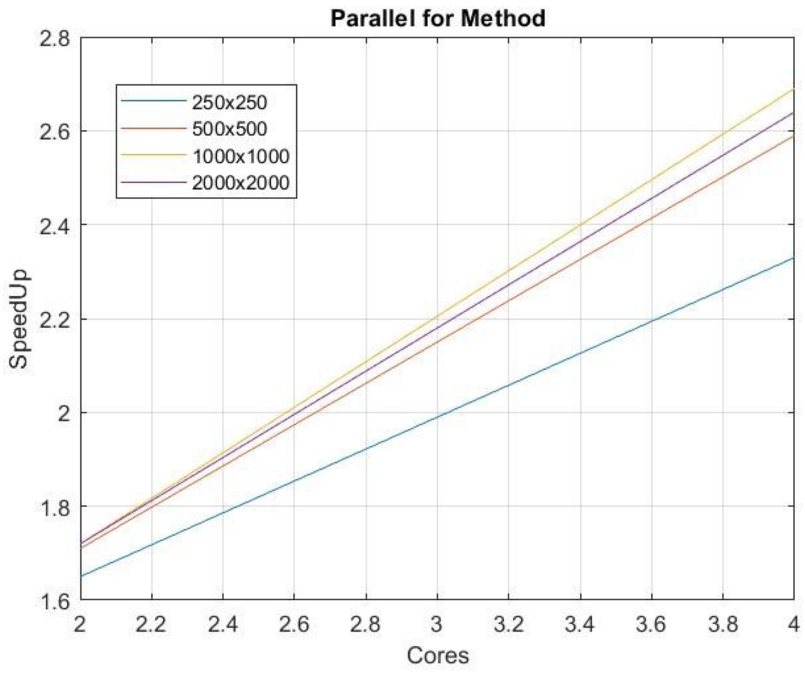
Parallelizing the Matrix Transposition it's possible to observe a linear behavior between the number of cores and the speedup (the code is parallelized for the 99%).

		Serial	2-Core	4-Core	SpeedUp	
Strassen	256x256	0.14692	0.074382	0.03830	1.97	3.83
	512x512	1.3708	0.69996	0.34822	1.95	3.93
	1024x1024	14.0416	7.9274	3.5964	1.93	3.89
	2048x2048	119.2447	60.54072	34.8767	1.97	3.44



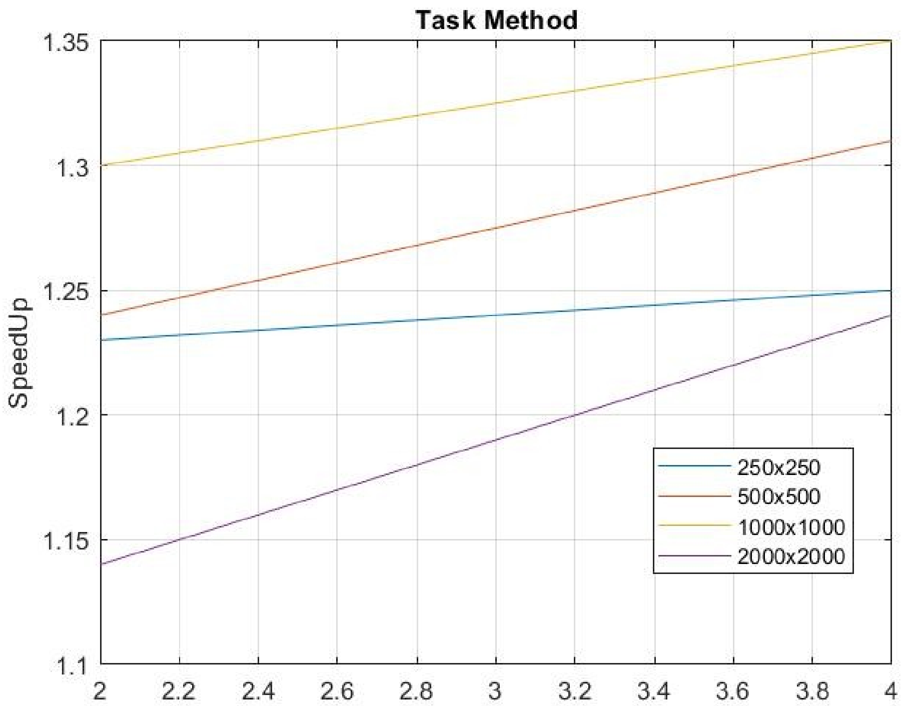
Like for the other methods it's possible to appreciate a linear behavior. Analyzing the time, it's possible to say this method is more efficient than the others because it limits the multiplications in favor of the sums and subtractions and this make the code faster.
In the next tables it is possible to appreciate the local results with the matrix inversion.

		Serial	2-Core	4-Core	speedUp	
Parallel For	250x250	0.180955	0.109575	0.077405	1.65	2.33
	500x500	1.443806	0.842853	0.557092	1.71	2.59
	1000x1000	12.26951	7.111274	4.560809	1.72	2.69
	2000x2000	114.80656	66.16880	43.339968	1.72	2.64



For the Inversion with the Parallel for we have good performances next to the theoretical upper bound. When the matrices become bigger the time for the decomposition rises up (on 2000x2000: 13 seconds): this time is negligible compared to the inverse time, but it starts to weight on the performances. For my purpose I decided to limit the matrix dimensions to 2000x2000 in order to have results in an observable time, but it is also possible to work with matrices dimensions bigger than 2000 (check if the memory capability allows it).

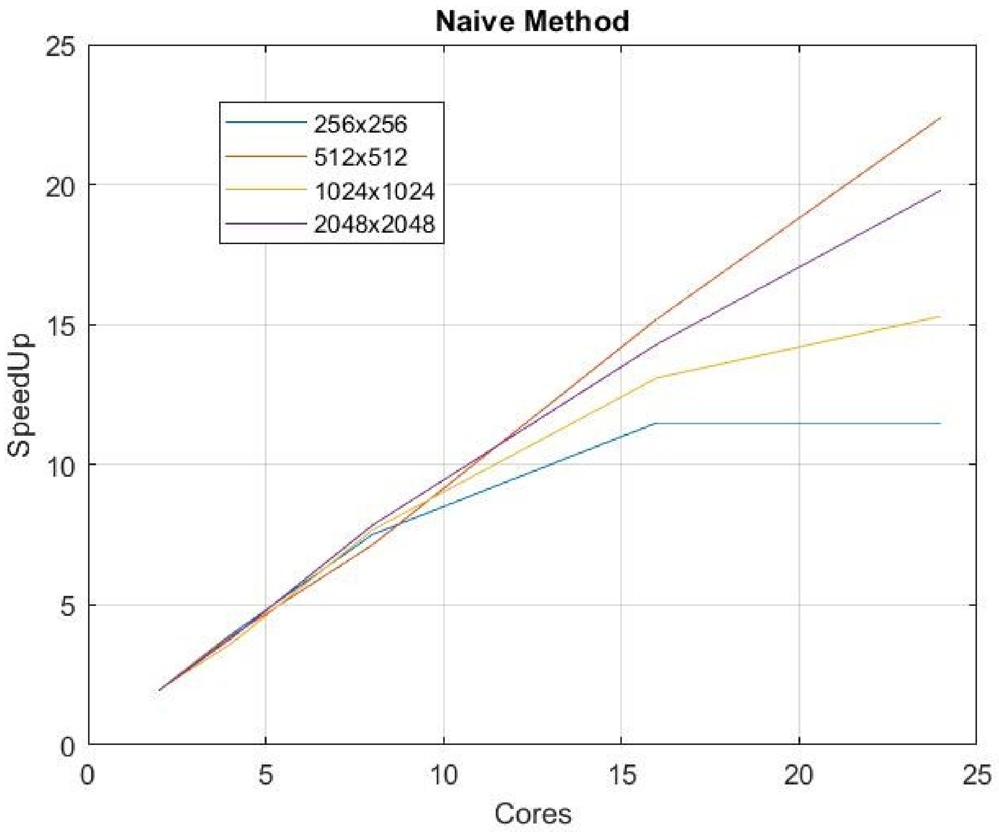
		Serial	2-Core	4-Core	speedUp	
Task	250x250	0.180955	0.146224	0.144319	1.23	1.25
	500x500	1.443806	1.159322	1.096198	1.24	1.31
	1000x1000	12.26951	9.412632	9.086384	1.30	1.35
	2000x2000	114.80656	100.056942	92.469805	1.14	1.24



For the Task Method the behavior is like I have predicted before: the regular shape of the matrix doesn’t allow to explore the parallelization with tasks. I obtained a little increasing of the speedup raising the number of cores.

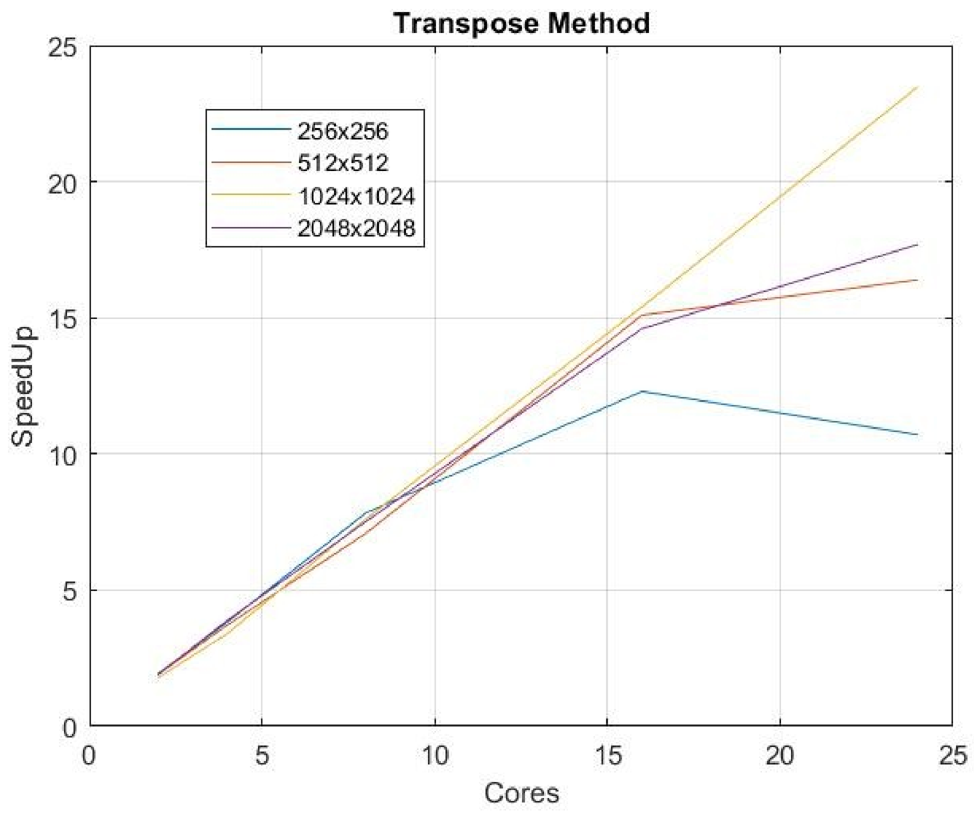
After computing the local tests, I passed to the Google Cloud Platform. With this service I created a virtual machine to explore the parallelism with a variable number of cpu (max 24 cpu) and 90 Gb of memory.

		Serial	2-c	4-c	8-c	16-c	24-c	SpeedUp				
Naive	256x256	0.1727	0.088	0.044	0.023	0.015	0.015	1.96	3.92	7.51	11.5	11.5
	512x512	2.1083	1.069	0.546	0.295	0.138	0.094	1.97	3.86	7.14	15.2	22.4
	1024x1024	27.235	13.75	7.717	3.676	2.081	1.774	1.98	3.59	7.67	13.1	15.3
	2048x2048	259.24	132.90	68.43	33.03	18.11	13.13	1.95	3.78	7.84	14.3	19.8



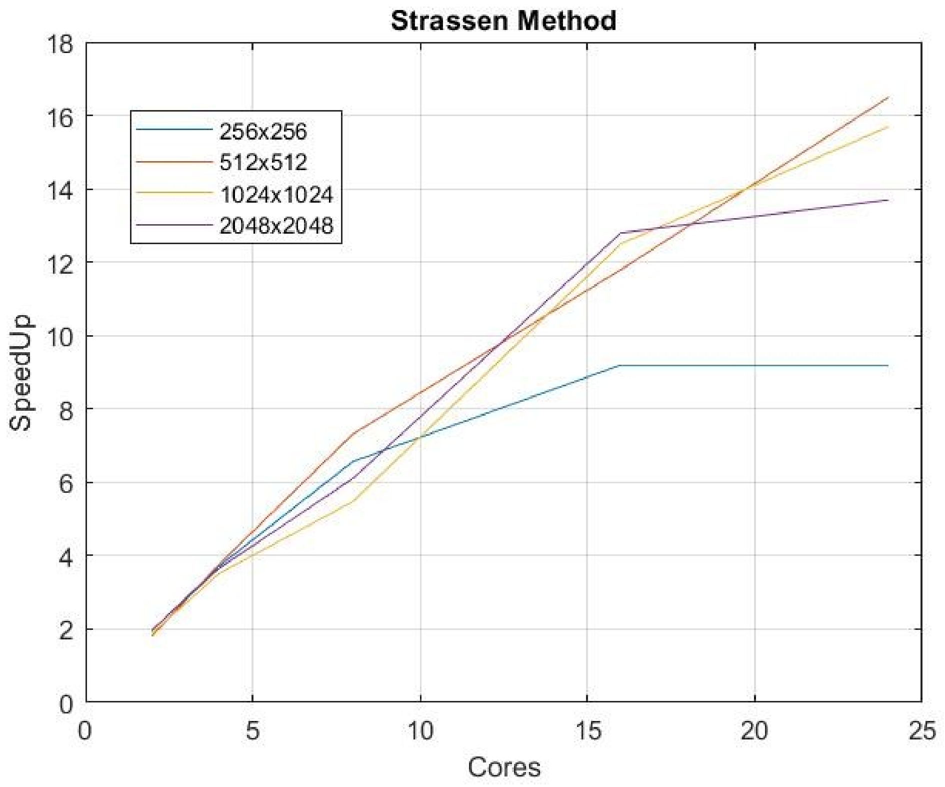
With no surprise the classic naive product between matrices completely exploit the parallelism. In fact we can see an almost linear dependence between cores and speedUp. The best result is obtained with a 512x512 matrices with 24 cpu.

		Serial	2-c	4-c	8-c	16-c	24-c	SpeedUp				
	256x256	0.1724	0.089	0.045	0.022	0.014	0.016	1.93	3.83	7.83	12.3	10.7
Trans.	512x512	2.0910	1.091	0.562	0.295	0.139	0.127	1.91	3.72	7.08	15.1	16.4
	1024x1024	26.582	14.81	7.800	3.489	1.715	1.129	1.79	3.40	7.61	15.4	23.5
	2048x2048	255.63	133.98	65.68	34.02	17.53	14.41	1.90	3.89	7.51	14.6	17.7



The first observable results obtained by watching the graph is the linearity of the 1024x1024 line: it grows respecting the Amdhal's law and reaches the maximum speedup with 24 cpu: 23.5 is close to 23.99 which is the limit. The other lines, when the number of cores increases, lose the linearity. A future study can be done increasing the number of cpu to analyze the behavior (for this region the limit of cpu is 24).

		Serial	2-c	4-c	8-c	16-c	24-c	SpeedUp				
Strass.	256x256	0.1380	0.0700	0.037	0.021	0.015	0.015	1.97	3.72	6.57	9.2	9.2
	512x512	1.1055	0.6130	0.293	0.151	0.093	0.067	1.81	3.77	7.32	11.8	16.5
	1024x1024	14.371	7.5948	4.082	2.619	1.146	0.911	1.89	3.52	5.48	12.5	15.7
	2048x2048	157.01	79.263	42.86	25.70	12.23	11.40	1.98	3.66	6.11	12.8	13.7

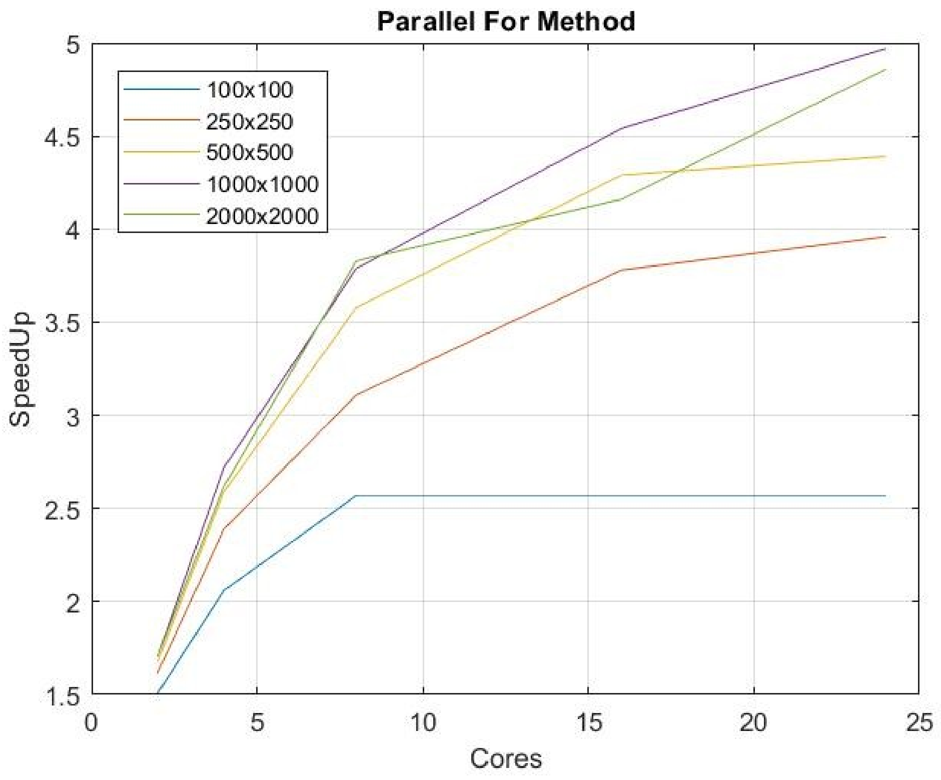


The Strassen Method is faster than the others, but when the number of the cpu became larger, the speedup doesn't grow with linearity.

Excluding one or two cases, all the tests above lose the linearity when the number of cpu grows up to 24. It can be possible because increasing the number of thread in a CPU-mono core environment the overhead caused by the threads synchronization, can weight on the performances. In fact as a support to this thesis, analyzing the local tests (which are done on a CPU-multi core) the linearity is confirmed (that can be verified only with 4 cores, having more cores can allow a deeper study).

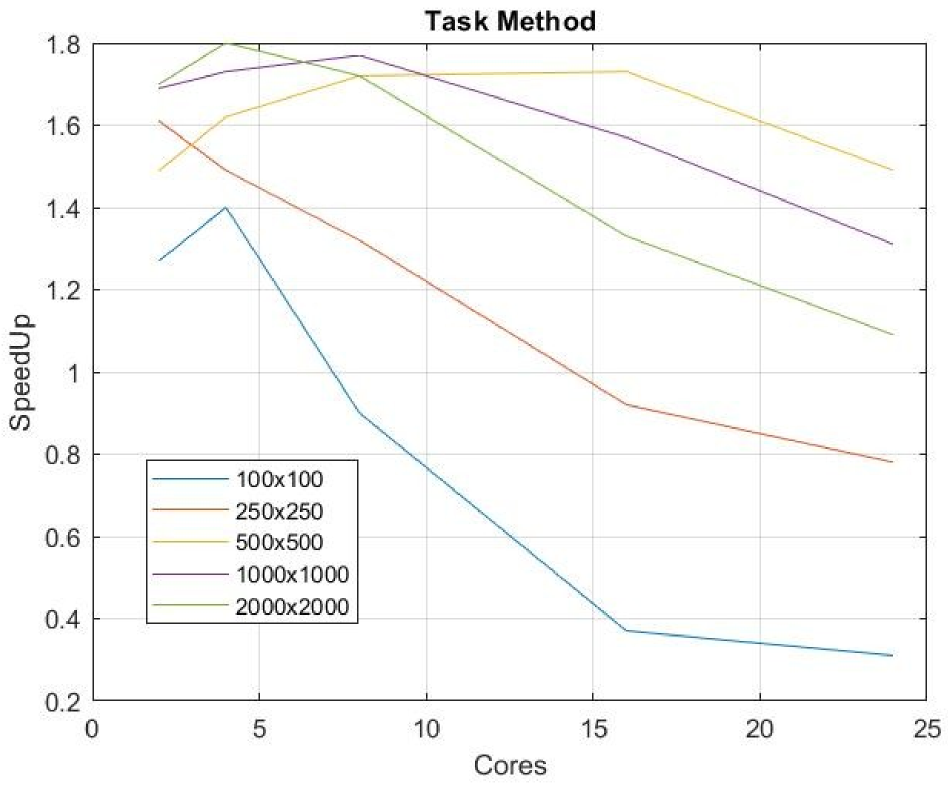
Cloud tests for matrix inversion:

		Serial	2-c	4-c	8-c	16-c	24-c	SpeedUp				
For	100x100	0.0103	0.0068	0.005	0.004	0.004	0.004	1.51	2.06	2.57	2.57	2.57
	250x250	0.1623	0.1004	0.068	0.0522	0.0429	0.041	1.62	2.39	3.11	3.78	3.96
	500x500	1.4095	0.8355	0.544	0.3930	0.3285	0.321	1.68	2.59	3.58	4.29	4.39
	1000x1000	12.030	7.0060	4.416	3.1707	2.6527	2.418	1.71	2.70	3.79	4.54	4.97
	2000x2000	107.18	62.791	40.84	27.937	23.213	22.01	1.71	2.62	3.80	4.16	4.86



From the analysis of the above graph and table it's possible to say that the parallelization algorithm grows up following the number of CPU until 4. Then, the lines, lose the linearity and start to be flatten. That because parallelizing only the 84% of the total code, the maximum speedup that can be reached with 24 cores is 5.23, and my results are close to the Amdhal's law limits. The algorithm works well with the increasing of the matrices dimensions.

		Serial	2-c	4-c	8-c	16-c	24-c	SpeedUp				
Task	100x100	0.0154	0.0121	0.011	0.017	0.041	0.051	1.27	1.4	0.9	0.37	0.31
	250x250	0.2210	0.1379	0.148	0.1664	0.2399	0.282	1.61	1.49	1.32	0.92	0.78
	500x500	1.7825	1.1891	1.013	1.0368	1.0312	1.195	1.49	1.62	1.72	1.73	1.49
	1000x1000	14.507	8.567	8.366	8.1900	9.2251	11.07	1.69	1.73	1.77	1.57	1.31
	2000x2000	117.09	68.84	64.96	68.236	87.582	106.8	1.70	1.80	1.72	1.33	1.09



The results obtained were completely expected: starting with small matrix, the small dimensions don’t allow to exploit the parallel algorithm. In fact using tasks with small matrices inserts only overheads, and it happens also in the local tests. Considering the 2 cpu test, the results are completely in line with the Amdhal’s Law, in fact Parallelinzing only the 84% of the code with 2 cpu, the maximum result that can be obtained is 1.73. For the 4 Cpu version, the maximum speedup that can be obtained is 2.72 and my result is not far from this limit. From 8 cpu and so on the results obtained aren’t good (with 24 cpu the maximum obtainable result is 5.23) and that can happen because the task works well in environments where the granularity can be explored. In my code the tasks repeat every time the same part of code with in the same interval, because an algorithm to invert a matrix is repetitive (iterative method). From my studies using tasks is not the best solution to explore parallelism with matrix inversion.

After a complete analysis of my results, the bests solutions to explore the parallelism with matrix inversion and multiplication can be implemented with LU factorization (parallelizing the inversion with parallel for) and the Strassen Algorithm.