# Report matrices inversion and multiplication

The main idea of the project was about parallelizing a code that operate a large amount of similar operations on matrices, in this case the multiplication and inversion for squared matrices of various size.

# Analysis of the serial algorithm

The first step was to replicate in code form the algorithms that we use day by day for those two operations and identify a structure that needs to be replicate in code form.

## Matrices multiplication: the row by column method

The algorithm of choice for this operation was the row by column method because of its simplicity and loop use in order to calculate the elements in the resulting matrix.

with size

To multiply two matrices those, need to be of size [a x b] and [b x c] in order to create a [a x c] resulting matrix. For simplicity I choose to operate only on squared matrices with increasing dimension N = 2000 from 2000 to 10000.

Implementing the code was simply a matter of translating the algorithm into a c++ function by changing the series operator into a *for* loop.

Here follows the code for the creation of the complete C matrix *(called here r)* and the series is translated into a series of *for* loops in order to create all the elements of the resulting N\*N matrix.

void multiply(float \*\*a, float \*\*b, float \*\*r, int size){

//multiply a\*b = r

for(int i = 0; i < size; i++)

for(int j = 0; j < size; j++)

for(int k = 0; k < size; k++)

r[i][j] = r[i][j] + a[i][k]\*b[k][j];

## Matrix inversion: LU method

For this operation the size of the squared matrices was different, it starts from 500 to 2500 with steps of 500.

The algorithm of choice here was the LU decomposition method. Even if during the bachelor courses of algebra it was taught us that an inversion of any square matrix could be done by using the determinant method, it’s simply waste too much resources of the computer in order to process a complete matrix. This is due to the large number of operations needed in order to calculate the determinant and sub matrices at each step, producing a large amount of operations *(N!)* to the problem. For small matrices like 2x2 or 3x3 it’s still doable but with the increasing size it will became much harder to perform.

So the solution was to use a different approach to the problem and by using the LU decomposition of the matrix A the amount of operations reduces drastically and at the same time there is the possibility of parallelize some part of the process.

First the A matrix is decomposed into two triangular matrices using a pivoting matrix, then are performed the forward and backward substitution that produce as result the inverted matrix.

In order to do that we defined a system with *A* the matrix that needs to be inverted, *I* the identity matrix of size N, and the inverted matrix of A.

That expression could be rewrite as but the problem of the inversion is still present. So, the two substitution are used in order to calculate, element by element, each column of the inverse of A.

for(int i=0; i< size; i++){

float\* y = new float[size]();

forwardSubst(l,p,y,i,size);

backwardSubst(u,y,a1,i,size);

}

The for cycle it’s used in order to calculate all the columns of , and the *size* parameter it’s used to identify the dimension of the matrix to be inverted.

### Forward substitution

The first step is to calculate the elements of an intermediate matrix Y that it’s used in order to simplify the calculation by allowing its creation column-by-column, one element at the time.

void forwardSubst(float \*\*l, float \*\*p, float \*y, int column, int size){

/\*\*

\* solving yi=Pi/ sum of Lrow-i

\*/

    y[0] = p[0][column] / l[0][0];

for(int i = 1; i< size; i++){

y[i] = p[i][column];

for (int j = 0; j < i; j++){

//the P[i] part has been done before

y[i] = y[i] - l[i][j]\*y[j] ;

}

y[i] = y[i]/l[i][i];

}

}

The main focus point here is the recursive for part. It’s used as *sum* in order to update the value y[i] before dividing the partial value of y[i] by the element of the L matrix that’s user full in order to calculate is final value and the complete column it’s calculated by using another for cycle around.

### Backward substitution

This is the last step in order to compute the inverse and it’s a similar to the forward substitution but it starts from the last row of the column and at the last step gives back the first row. Then the same procedure is applied to all the other columns and the inversion is complete.

void backwardSubst(float \*\*u, float \*y, float \*\*a1, int column, int size){

/\*\*

\* solving A^-1 [i] = P[i]/sum of Lrow-i

\*/

a1[size-1][column] = y[size-1] / u[size-1][size-1];

for(int i= size-2; i >= 0; i--){

for (int j = size-1; j > i; j--)

{

a1[i][column] = a1[i][column] - u[i][j]\*a1[j][column] ;

}

a1[i][column] = a1[i][column]/u[i][i];

}

}

As before the sum is calculated trough a for loop and the complete column vector is calculated by using another for loop outside the first one.

# Parallelization of the code

The library chosen for those projects was openMP due to the fact that’s very easy to use and scales very well on a variety of number of threads. Plus, it has a function that can be used to calculate the speedups after the process ended.

Both the programs I decided to parallelize the allocation in memory and initialization of the matrices to reduce the testing time needed but in order to not modify the tests results, the timer for calculating the speedups was set to start an instruction before the starting of the calculation of the multiplication and inversion.

## Matrix multiplication

Due to its simplicity, parallelizing this code it’s very easy and efficient thanks to its structure based around three *for* cycles.

void multiply(float \*\*a, float \*\*b, float \*\*r, int size){

//multiply a\*b

#pragma omp parallel for collapse(2)

for(int i = 0; i < size; i++)

for(int j = 0; j < size; j++)

for(int k = 0; k < size; k++)

r[i][j] = r[i][j] + a[i][k]\*b[k][j];

The “*#pragma omp parallel for collapse(2)”* is used in order to parallelized the operations included into those three loops by **unfolding** those operation and splitting those between the threads available on the virtual machine. By mistake I’ve tried also the *“collapse (3)”* that at first sight gives back better results in terms of time but due to data dependences produces a wrong result at the end.

## Matrix inversion

This process is much complicated than the multiplication in fact some parts cannot be parallelized due to its structure.

For example the pivoting function is one of those section that, even if it is composed by for cycles, even parallelizing just one of those will create conflicts around the shared variables that indicates which row contains the maximum value, that will increase the execution times so I ended up leaving it in serial

void pivoting(float \*\*a, float \*\*p, int size){

for (int k = 0; k < size-1; k++){

int imax = 0;

//foreach column i need to find which row has the maximum (in module) value

for (int j = k; j < size; j++){

//finding the maximum

if (abs(a[j][k]) > abs(a[imax][k])){

imax = j;

}

}

//now i swap the row imax and k

swap(a[k],a[imax]);

swap(p[k],p[imax]);

}

}

For the other section there was a similar problem. The forward and backward substitution rely on each other and also on the previous results in order to compute the next step. So the solution that was left in order to have a faster execution was to parallelize only the columns. Calculating more columns of the inverse at the time will decrease the execution times without having conflict due to the independency of the calculations.

void findInverse(float \*\*a, float \*\*a1, float \*\*l, float \*\*u, float \*\*p, int size){

/\*\*

\* foreach column i solve the system LUai=Pi with the i-th column

\* by using the forward substitution method

\* LYi=Pi

\* Uxi=Yi

\*/

#pragma omp parallel for

for(int i=0; i< size; i++){

float\* y = new float[size]();

forwardSubst(l,p,y,i,size);

backwardSubst(u,y,a1,i,size);

}

}

# Testing and debugging process

## Test platform

The virtual machine that I’ve created to run those tests is a 24 threads machine with 24gb of ram due to the google cloud platform limitations. The operative system of my choice was Ubuntu 18.0 that was controlled trough SSH.

## The testing methodology

Due to time constraints on the VM use, two different type of test were used. For the inverse I’ve chose to do a series of five calculations foreach dimension tested *(from 500 to 3000 with steps of 3000)* then a media was calculated. The same process was applied for different amount of threads *(1,2,4,6,8,12,24)*.

For the multiplication instead, due to the bigger dimension and time needed for a single iteration, I’ve chosen to do a single run *(otherwise would be impossible to do the various test in a feasible time frame)*.

Then the results were written on the terminal in a tabular form and for exporting them more easily on excel for further calculations, they were written also on a txt files.

double time= omp\_get\_wtime();

multiply(a, b, r, size);

time = omp\_get\_wtime()-time;

This process gives me more precise measurements but at the same time I had to increase drastically the execution time of the entire process, especially with a low thread count and high dimension size.

int dimension[]={500,1000,1500,2000,2500,3000};

int threadcount[]={1,2,4,6,8,12,24};

double avgtime;

    ofstream outfile;

    outfile.open("Test\_results\_inverse.txt");

for (int i = 0; i < 7; i++)

{

cout<<"\n\nThreads: "<<threadcount[i]<<"\nSize:\tTime AVG:\n";

        outfile <<"\n\nThreads: " << threadcount[i] << "\nSize:\tTime AVG:\n";

for (int j = 0; j < 6; j++)

{

avgtime = 0; //reinitilize it

cout<<dimension[j]<<"\t";

            outfile << dimension[j] << "\t";

for (int k =1; k <= 5; k++){

                avgtime = avgtime + execution(dimension[j],threadcount[i]);

}

avgtime = avgtime/5.0F;

cout<<avgtime<<"\n";

            outfile << avgtime << "\n";

}

}

    outfile.close();

return 0;

}

Then the *threadcount* value was used in the execution part to be used in the instruction, while the *size* identify the N dimension of the square matrix.

double execution (int size,int threadcount){

omp\_set\_num\_threads(threadcount);

Due to its form, the code will allocate large matrices dynamically during the process and to still have memory during the various test at each iteration of the process of inversion (*or multiplication*), those memory areas were free at the end of the calcualtions.

time = omp\_get\_wtime()-time;

free(\*a);

free(\*a\_p);

free(\*a1);

free(\*l);

free(\*u);

free(\*p);

return time;

}

# Problems

The first one is the time needed for the entire process. It took three days just for the multiplication with the single thread performance test that took fifteen hours for all the multiplications.

The other problem was the approximation that comes with the use of random filled matrices. This creates a few oscillations in the time needed for the operation. For instance, multiplying something for 0 is much faster that do it for 50. This create some outliers in the speedups obtained and in order to better visualize the tendency of the speedups value I’ve included a trendline in the graphs.

Due to probably data dependency, the 3000x3000 inversion with 24 threads resulted always in a “killed” status in the terminal. That’s why all the 3000x3000 inversion are included in the graphs and table but they are considered marginally in the analysis due to the incompletion if the data.

# Performances analysis

## Matrix multiplication analysis

As can be expected from the premises, the multiplication was the operation that mostly gains from the parallelization. That’s because its two outers for loop can be collapsed and there isn’t much else that prevents a flawless elaboration if the dependences are respected in order to calculate the *i-j*th element by calculating an element foreach thread. This means that if with a single core we must do one operation at the time, with twenty-four threads we can calculate ideally the same number of elements at the time, resulting in an ideal speedup of twenty-four times. Unfortunately, this value stays an ideal one because we still have a lot of memory access that must be managed at each operation that brings down the value with higher size.

By looking at the graph in the next page, that reports the speedups with various matrix size and the trend of it, we can clearly see that the difference from a dual core performance to a twenty-four threads one, at its peak, is more than twenty-one times faster. The matrix dimension is surely a component of this gain because for the 2000x2000 matrix multiplication it’s relatively simple to manage and calculate due to its lower component value than the biggest one. We are talking in a huge difference in elements that needs to be elaborate. From 4.000.000 to 1.000.000.000 the difference enormous compared to the smaller number.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| speedup | threads |  |  |  |  |  |
| Size: | 2 | 4 | 6 | 8 | 12 | 24 |
| 2000 | 2,121788 | 4,8540 | 7,301577 | 9,936733 | 21,75091 | 21,75091 |
| 4000 | 1,910194 | 4,2075 | 6,375834 | 8,945286 | 18,12041 | 18,12064 |
| 6000 | 2,193526 | 4,1606 | 6,244504 | 10,55287 | 18,52339 | 18,52294 |
| 8000 | 2,28238 | 4,4227 | 6,590164 | 10,74944 | 15,9402 | 17,76406 |
| 10000 | 2,015734 | 4,0360 | 7,163732 | 9,701283 | 14,51741 | 16,15368 |

From the small pool of test that I have done I can clearly tell from the results that for a twenty-four-thread application, in order to have a bigger benefit, a relative small size matrix is needed, because of the memory access that’s needed and creates a queue between the cores, but if the process in use has 6 threads or less, the speedups are a little higher respect the theoretical results. This can be caused also by the large cache size of the processor used in the virtual machine. The multiplication is quite fully parallelizable, bringing the (*1-Parallelizable code*) part, that represents the amount of code that cannot be parallelized close to 0.

In the data columns we can see that we exceed the theoretical speedup. This can be caused by the fact that the smaller amount of threads is simulated on a 24-core machine with its cache size that is bigger than a smaller processor. Also, the pipeline could have provided an additional increase of performance by being full most of the time due to the program structure.

### Real world applications

In robotics for example we use a lot of 4x4 and 6x6 matrices. Those are multiplied between each other to calculate the position of each joint of a robot respect to its base. So, for a program that calculate trajectories and then has to control a robotic arm even a four-thread CPU could be a good point of start to have better execution times and real time application in industrial plants. Naturally for more complex operation like structure simulations and mathematical problems a good balance between cache size, thread count and processor’s speed are advised.

## Matrix inversion

Due to its data dependences and part, like the pivoting, that cannot be parallelized, as expected the speedups are much lower than the multiplication throughout all the number of cores.

This means that to have a significant speedup for higher thread-count we need to have a bigger matrix to invert, but due to the testing virtual machine limitation in terms of time and memory we can probably saturate resources available due to the bigger number of matrices needed to do the calculations compared to a simply multiplication.

In terms of numbers, from the three theoretical matrices used, we also need some vectors during the calculation to make the parallelization easily and more efficient. This is done for reducing the access to the same part of the memory from each thread. Also, as explained before, we can only do a small amount of column at the time compared to those present in the full matrix. This creates also has to solve data dependences that are present in those operations. So, when parallelized, return a much slower than the theoretical value.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| speedup | threads |  |  |  |  |  |
| Size: | 2 | 4 | 6 | 8 | 12 | 24 |
| 500 | 1,593948 | 2,223975 | 2,563894 | 2,776905 | 3,081744 | 3,0137 |
| 1000 | 1,632681 | 2,325434 | 2,760951 | 2,953497 | 3,371821 | 3,432663 |
| 1500 | 1,802077 | 2,595551 | 3,032508 | 3,317658 | 3,70998 | 3,795603 |
| 2000 | 1,652127 | 2,457734 | 2,900739 | 3,186687 | 3,587114 | 3,757486 |
| 2500 | 1,647175 | 2,475019 | 2,962443 | 3,24115 | 3,756088 | 3,87407 |
| 3000 | 1,657498 | 2,506854 | 2,928824 | 3,395682 | 3,83481 | -killed |

The previous aspect is particularly visible by the 3000x3000 inversion. Each time that was tested with 24 threads the memory access and consumption will drastically increase, causing, probably, the complete process to time out or be killed by the operating system.

As before the speedups are calculated trough the Amdahl law and here we have a bigger portion of the code that cannot be parallelized, increasing the denominator of the formula that will decrease the value then obtained with the test.

Another interesting fact that can be read from the data and particularly from the graph there isn’t too much difference in performance from twelve threads to twenty-four. As before this difference could be more noticeable with larger matrices but, for what I have tested.

A big difference that is present from the multiplication is the pipeline use. We still are able to have some benefits from it, but due to the program composition, the time gains are much lower. The necessity of comparing data and jump to some function, in the case of the pivoting, create a weak use of the pipeline. If we add the amount of data that needs to be fetched in order to do the forward and backward substitutions, we can clearly notice that the number of operation that are required increase drastically compared to the multiplication.

### Real word applications

The matrix inversion, in my experience, could be easily found in control systems, in the LQR and MPC aspect of it. It requires a series of multiplication and inversion of matrices that creates a model, from a differential equation. Then the controller uses it to modify the controlled variables in order to keep the system stable around a required point.

Those matrices increase in size depending from the variables controlled and from the number of inputs used so a multithread approach can be helpful in order to control faster and more precisely a complex system.