PROJECT PROPOSAL PHPC-2019

A High Performance Implementation of the Conjugate Gradient Method (CG)

Principal investigator (PI)	Nicolas Lesimple
Institution	Ecole Polytechnique Fdrale de Lausanne
Address	Avenue des Bains 512, CH-1007 LAUSANNE
Involved researchers	Only PI
Date of submission	June 9 2019
Expected end of project	June 9, 2019
Target machine	deneb1
Proposed acronym	CONJGRAD

Abstract

The Conjugate Gradient Algorithm (CG) is perhaps the best known iterative technique to solve linear systems that are symmetric and positive semi-definite. In this project, we implement and investigate a parallel implementation of the conjugate gradient algorithm using MPI and OpenMP applying this method for solving large systems of linear algebraic equations. The system has the form Ax=b. This type of system is an important problem in many fields of science and engineering. The computational power of such a task increase with the size of the studied matrix and vectors. That's why a parallel implementation is critical to allow the user to be able to compute larger problems. Performance analysis of our implementation will be conducted while using the method to solve linear systems of different sizes on an increasing number of processors.

1 Scientific Background

In mathematics, the conjugate gradient method (CG) is an algorithm for finding the numerical solution of particular systems of linear equations, namely those whose matrix is symmetric and positive-definite. The conjugate gradient method is usually implemented as an iterative algorithm, applicable to systems that are too large to be handled by a direct implementation or other direct methods such as the Cholesky decomposition. It can also be used to solve unconstrained optimization problems such as energy minimization. Therefore, it is very beneficial to apply both shared and distributed memory parallelization techniques. To do that, we use the Message Passage Interface (MPI) (distributed memory) and OpenMP (shared memory). The goal of this project would be to find a High-Performance Computing strategy allowing a substantial speed up while keeping the algorithm user-friendly and easy to understand. This implementation would benefit to a lot of people working with high dimensional linear systems. More specifically, this can benefit to the field of physics where researchers are usually confronted with this type of system. This project will explore a new paradigm at scale. It would be an exploratory project. Thus, the innovation of that kind of project would be to find a new method of parallelization to achieve a significant speed up while using a Hybrid OpenMP and MPI paradigm.

Data: A,b,x_0
Result: x such $Ax = b$
initialization:
usually $x_0 = 0$
$r_0 := b - Ax_0$
$p_0 = r_0$
$k := 0$
while $||r_{k+1}|| > 0$ **do**
  $\alpha_k := \frac{r_k^T r_k}{p_k^T A p_k}$
  $x_{k+1} := x_k + \alpha_k p_k$
  $r_{k+1} := r_k + \alpha_k A p_k$
  $\beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
  $p_{k+1} := -r_{k+1} + \beta_k p_k$
  $k := k + 1$
end

Figure 1: Basic Sequential Version of CG algorithm

This report is organized as follows. First, the Conjugate Gradient method will be described and a sequential implementation will be presented. Then, in the second part, the methods used to parallelize the CG algorithm with the implementation are described. More precisely, the effects of different parallelization approaches are explained. In a third section, analysis of scaling was performed. In addition, in the following part, the resources budget was analyzed. Indeed, the memory storage and the computing power needed were described. Finally, a conclusion summarizing our finding and implementation were done. Potential improvements for future work was also explained.

The chosen version of the sequential Conjugate Gradient algorithm can converge only if the coefficient matrix of the system of linear equations is symmetric and positive semi-definite. The script called *Random_matrix.py* allows the creation of this type of matrix using two for loop. More precisely, we will first create random matrix A and vector b. Then, matrix A will become A*A and vector b will become A*b. This definition of our system assures us the positive semi-definite symmetric property. During our experimentation, we will always use this kind of matrices.

Our CG algorithm is used to search for the minimum value of a multi-dimensional function. In our case, we are searching for the minimum value of the residual $r = b - Ax$. In each iteration, the estimate of the solution x is updated by moving a certain amount α along the search direction vector p. The coefficient α is determined by the current iterations values of A and p. The search direction is adjusted depending on the change in the norm of the residual which is also used as the convergence check. A pseudo-code is shown in Figure 1. We will omit a deep analysis of this algorithm. In fact, our purpose here is to analyze the parallelized implementation of this algorithm and not the mathematical proofs behind it. For more background on the conjugate gradient algorithm, please refer to [2].

2 Implementations

The application is implemented in C++. We have created one implementation of the same CG solver which is a hybrid version using both MPI (distributed memory version) and OpenMP (shared memory version). The code has been fully debugged using the gdb debugger. In addition, the gprof profiling tool has been used to study the behavior of our implemented algorithm. Intel compiler optimization reports was also used to optimize our code and more specifically the matrix vector multiplication.

2.1 Parallelizing the Conjugate Gradient Algorithm

Two main types of parallelism can be achieved in this code: the first one would be task-based, meaning to have each processor perform a certain different task. As we can see in the pseudo-code in Figure 1, (almost) each step depends on the previous step so there is not much opportunity for task-based parallelism. In fact, since CG is an iterative method, the result at the next step depends on the result at the current step, thus it is hard to simultaneously compute the intermediate steps (such as step size, search direction, ...).

The other main type of parallelism is based on exploiting geometric parallelization. This method consists of distributing the data across processors and making each processor responsible for computations on its local data. In fact, the CG algorithm is naturally suited for data-based parallelism. It means that to obtain a speed-up we will divide the data and operations on the among MPI ranks and OpenMP threads. We will study this type of implementation in more depth, for precise steps of the algorithm. In fact, in the sequential algorithm presented above, we can identify basic matrix/vector operations that can be potentially performed in parallel :

- Matrix-Vector Multiplication
- Vector Dot Product
- Scalar-Vector multiplication and more precisely Linear Combination Operation
- Vector Norm Calculation

After some numerical investigations, we decided to focus our parallel interpretation on these operations as they seem to be the most time-consuming operations (matrix-vector multiplication being the more important one). One of our guesses is that the scalar-vector multiplication is a too simple operation to get real speedup when computing in parallel. In the following subsection, we will analyze deeper the methods used to parallelize these operations.

In addition to this operation parallelism, the entire CG process can also be done on a sub-part of data. In fact, using the same principle, the system of equation will be divided into smaller parts, allowing the creation of an upper level of parallelism.

To succeed in this task, Message Passing Interface (MPI) is used by dividing the matrix and vectors into horizontal slices with each rank being responsible for the calculations involving each slice. Additional parallelism is achieved with OpenMP (omp) by multithreading the for loops of the operation described above (matrix-vector multiplication, vector dot product, the linear combination of vectors with scalar, and vector norm).

Data: A,b,x_0
Result: x such Ax = b
initialization :
$x_{sub\_0} = 0$
$r_{sub\_0} := b_{sub} - A_{sub} x_{sub\_0}$
$p_{sub\_0} = r_{sub\_0}$
for *i = 0 : max_iteration* `// we set it to 100000`
do
  $r_{sub} = r_{sub\_old}$
  `// Here 3 MPI and 2 OpenMP implementation were tried :`
  vec_sub_A_multiply_by_p = matrix_multiply_vector_openmp(A_{sub}, p)
  `// Dot products use MPI Allreduce and omp parallel for reduction`
  $\alpha = \frac{dot\_product\_mpi(r_{sub}, r_{sub})}{dot\_product\_mpi(p_{sub}, vec\_sub\_A\_multiply\_by\_p)}$
  `// Linear combination using omp parallel for`
  $x_{sub} = x_{sub} + \alpha * p_{sub}$
  $r_{sub} = r_{sub} - \alpha$*vec_sub_A_multiply_by_p
  `// Check the convergence:`
  if *vector_norm(r_{sub}) < tolerance* **then**
    break
  end
  `// Update of the search direction`
  $\beta = \frac{dot\_product\_mpi(r_{sub}, r_{sub})}{dot\_product\_mpi(r_{sub\_old}, r_{sub\_old})}$
  $p_{sub} = r_{sub} + \beta p_{sub}$
  MPI Allgatherv p_{sub} to p on all Ranks
end
MPI Gatherv of x_{sub} to x on Rank 0
Return x

Figure 2: Parallelized Version of CG algorithm

2.1.1 Global Parallelization : Higher Level

As we just said, we use MPI to divide the initial system given as input into smaller sub-part, and on each of these sub-parts, the CG algorithm is applied. This constitutes the main level of parallelism in our implementation. With this geometric parallelization process, (data-parallel approach), each rank and thread has an approximately equal amount of data and operations to perform. Thus, the load balancing is nicely applied in our implementation and nearly always true. To achieve this parallelism, we used two MPI function : *MPI_Allgatherv()* and *MPI_Gatherv()*. Both commands are used to gathers data from several tasks and deliver the combined data.

However, the 'gathers' of the final solution vector at the end of our Conjugate Gradient function breaks this load balancing rule. In fact, Rank 0 is the only one who is gathering the final x vector. This is not a big deal as this step occurs at the end of our algorithm, which means that there is no additional work to perform. Thus, it does not prevent the other ranks from performing further work. The complete pseudo code of our CG parallelized implementation can be seen in Figure 2.

To achieve the implementation of the hybrid algorithm we used the following MPI and OpenMP methods :

- MPI_Allreduce for dot products
- MPI_Allgatherv for matrix-vector product
- MPI_Gatherv to get solution vector
- omp parallel for and/or omp simd in matrix-vector product
- omp parallel for reduction for sub-vector dot products

In addition, to assess the performance of our implementation, we measured different intervals of time. The goal of this was to compare the effectiveness of our implementations. These time measures always correspond to a mean over 5 independent experiences. It will allow us to obtain more robust and reliable statistics.

To use our implementation, the user needs to verify the comparison between the number of processes with the size of the system n. In fact, to allow a good behavior of our code, we assume that the number of elements n is greater than or equal to the number of processes p.

2.1.2 Input/Output

As said before, the matrix A and b corresponding to the system Ax=b are defined and saved into a file thanks to the python script *Random_matrix.py*. Then, this file is read in ourm *main.cpp* thanks to *ifstream* flow. This reading process is done thanks to rank 0 and store as a variable in our code.

The writing of the solution vector was also done by MPI Rank 0 after it performed a verification with the computation of an error vector. The solution vector is relatively small compared to the matrix, so a parallel output is not necessary.

In addition, each time we use our implementation we check if the results of our CG algorithm was right. To do that, we calculate the result of A*x where A is the matrix given as input and x is the solution we found. This result should be equal to the vector b given as input in the CG algorithm. Thus, for each element of the vector b, we subtract the real value with the one found and we check if the absolute value of the difference, which corresponds to the error, is inferior compared to a chosen tolerance. In our case, the tolerance was set to 10^{-7}. Moreover, solutions found when using different numbers of ranks and threads were also manually checked to ensure that they are nearly the same as the serial case.

2.1.3 Matrix-Vector multiplication Implementation and Optimization

Besides the standard sequential optimizations (loop reordering, vectorization, etc..), we put a special effort to tackle the main bottleneck of our hybrid implementations of the CG algorithm: the Matrix-Vector multiplication part.

In fact, thanks to the usage of *gprof* profiling tool, we are able to observe that this matrix-vector multiplication is the most time-consuming part of the entire algorithm. *Gprof* is a profiling tool based on functions calls counting. In fact, if there are no functions this tool will be useless as nothing interesting will be print. It cannot get a finer grain than the function

calls. This fact makes the implementation of this operation the more crucial part of this level of parallelism. We will analyze the process in details.

In the pseudo code below, the algorithm allowing a non-parallelized vector-matrix multiplication is described. It performs the addition of the partial multiplications of the vector and the matrix rows.

```
1 void matrix_multiply_vector(double **matrix, double *vector, double *result, int M, int N) {
2     for (int i = 0; i < M; i++) {
3         result[i] = 0;
4         for (int k = 0; k < N; k++) {
5             result[i] += matrix[i][k] * vector[k];
6         }
7     }
8 }
```

First, we focus on the memory access pattern in lines 2 to 7. At line 5, the main accesses patterns to array variables result, matrix and vector are done. As we can see due to index, between matrix and vector we have similar memory access in the last dimension indexed by k. As k is the index of the innermost loop, this row-major access pattern is perfect for a C code as it exploits memory locality. The other memory access involved is the writing into result[i], where i is the index of the outermost loop. As in this case, there is row-major access to a one dimension array, we would not have locality problems. Once we know there are no big problems with the memory performance (this is our case), we know we will be able to get good speed-ups on this code if we parallelize it. We decided to implement and study 3 different ways to parallelize this operation with the MPI library. Note that the variables matrix and vector are read-only so they are never written in this algorithm.

MPI Self Scheduling : The first one is called the Self Scheduling algorithm. This method has the goal to complete the matrix-vector multiplication on p processors of message passing cluster. From the literature, it seems that this algorithm is not the best way to parallelize this particular numerical computation. We mainly used the basic MPI send and MPI receives operations. The main idea of this algorithm is the existence of a master process that is coordinating all the other processes. In fact, we assume that the matrix A of size $n * n$ is available with the master process of rank 0 and the vector x of size n is available on all the slave processes, whose rank start from 1 onwards. When the slave finishes its workload, it informs the master which assigns a new workload to the slave. The body of the master loop consists of receiving one entry in the product vector from any slave and sending the next task (row of matrix A) to that slave. In other words, completion of one task by a slave is considered to be a request for the next task. Once all the tasks have been handed out, termination messages are sent. The body of a slave loop consists of receiving a row of matrix A, performing the dot product with vector x, and sending the answer back to the master.

MPI Block Striped Partitioning : The second method used is called block striped partitioning algorithm. In the striped partitioning of a matrix, the matrix is divided into groups of contiguous complete rows or columns, and each processor is assigned one such group. The partitioning is called block-striped if each processor is assigned contiguous rows or columns. Striped partitioning can be block or cyclic. The matrix A of size n x n is striped row-wise among p processes so that each process stores $\frac{n}{p}$ rows of the matrix. Then process P_I computes the dot product of the corresponding rows of the matrix A with the vector x and accumulate the partial result in the array y. Finally, process P_0 collects the dot product of different rows

of the matrix with the vector from all the processes. The main method was used to parallelize the CG algorithm at a higher level.

MPI Block-Checkerboard Partitioning: The third method is called block-checkerboard partitioning. This algorithm was difficult to implement as it required a lot of receiving and sending with the corresponding indexes. The main idea is to arrange the processors in a square grid of q x q processors where $q^2 = p$. We thus assume that the number of processors is a perfect square number. We also need to assume that the size of the matrix is evenly divisible by q.

In checkerboard partitioning, the matrix is divided into smaller square submatrices that are distributed among processes. A checkerboard partitioning splits both the rows and the columns of the matrix, so no process is assigned any complete row or column which is the main difference with the previously described method. Each process stores $\frac{m}{q} * \frac{n}{q}$ blocks of the matrix A and $\frac{n}{q}$ elements of the vector x. Processes broadcast elements of the vector to the other processes in the respective rows of the grid. Each process then performs multiplication of its block matrix with local vector elements and stores the partial result in the vector y to give birth to a resultant vector y of size $\frac{m}{q}$. Finally, process P_0 gathers accumulated partial sum on each process to obtain the resultant vector y.

OpenMP: With OpenMP command line we are exploiting the coarse-grain parallelism by parallelizing the outermost loop of the algorithm. As said previously, the vector and matrice variables are read-only and the result of the operation is written with the help of the index of the outermost loop. So if we distribute the iterations in this loop, making i private and making result shared, the threads will never write in the same positions of result and our parallelized algorithm is created.

In our implementation of this specific operation, the Single Instruction Multiple Data (SIMD) vectorization in combination with *omp parallel for* was used. With the SIMD option, compilers vectorize loops automatically only if a compiler is sure that the data is independent. To check this property the command *pragma omp for simd* was used. In fact, this line of code applied to a loop indicates if the loop can be transformed into a SIMD loop.

With this approach, three ways of doing were tested. First, we just use the command *pragma omp parallel for* which creates a shortcut for specifying a parallel construct containing one or more associated loops and no other statements. The second test was done by replacing the omp parallel for with omp simd. In this case, the result was much slower. However, the SIMD vectorization combined with the parallel for command loop (third try of parallelization) by using the command *pragma omp parallel for simd* resulted in nearly the same CPU times as without the vectorization. We thus decided to keep this last version.

The vectorization did not bring any additional speed up. In fact, it's not nearly as beneficial as multi-threading. This observation could be explained by the fact that most of the time is used to access data meaning that the loop would be memory bound. In this case, using multiple CPU cores would be an effective strategy compared to vectorization which is the same core trying to perform multiple instructions while accessing the data.

We compared these different time measures using our own CPU counting time and the Intel compiler optimization reports.

Finally, we decided to keep the OpenMP version using the SIMD vectorization combined the parallel for command loop. In fact, the MPI Self Scheduling method was not performing well, while the MPI block-checkerboard partitioning had too many inputs requirements to allows the

creation of a user-friendly implementation. The MPI Block Striped Partitioning was performing well and have none negative aspects. However, the OpenMP implementation with the parallel for command loop is applying the same parallelization concept. As the OpenMP version has an additional feature which is the SIMD vectorization we decide to keep this last version even if both methods has the same speed up effect.

2.1.4 Linear Combination Implementation and Optimization

As you can see on the pseudo code of the Conjugate Gradient algorithm, the linear combination operation is needed. By linear combination, we mean to implement the following operation : $result = a * \mathbf{u} + b * \mathbf{v}$ where u and v are some vectors and a and b are some double. Thus, we need to implement the sum of two vector multiply by a constant. The method described in the pseudo-code below was implemented without any high parallel notion. In addition, the same function was implemented using OpenMP command line: *#pragma omp parallel for* which allows a shortcut for specifying a parallel construct containing one or more associated loops and no other statements.

```
void linear_combination(double a, const vec &u, double b, const vec &v, vec &result) {
    size_t n = u.size();
    for (size_t j = 0; j < n; j++)
        result[j] = a * u[j] + b * v[j];
}
```

We experimented our code. By looking at the time it takes in several cases, we are able to conclude that our implementation of this operation was faster without this pragma call. This fact is at least true for the matrices size we are currently working on which is approximately square matrices of size 2000. This observation can be explained by the fact that the overhead of creating threads is larger than operations.

2.1.5 Vector Dot Product Implementation and Optimization

Vectors can be partitioned in different ways and the partitions can be assigned to different processes. Our method is based on the striped partitioning of a vector, where the vector is divided into groups of contiguous elements and each process is assigned to one such group. A serial algorithm for vector-vector multiplication requires n multiplications and (n-1) additions. This serial program is given below in Listing 2:

```
float vector_dot_product (int n, float x[ ], float y[ ]) {
    int i;
    float dot_product;
    for(i=0; i < n; i++) {
        dot_product = dot_product + (x[i] * y[i]);
    }
    return(dot_product);

}
```

Thus, for the dot product, we implement two different methods. The first one, called *dot_product()*, is not using any method of parallelization. The implementation is nearly the one described above in the pseudo code. We implement this dummy method because we need it in another function which is *matrix_multiply_vector_openmp()*. In addition, this method will be used to make a comparison with a parallelized version.

As you may have understood, the second implemented method is called *dot_product_mpi()*. This function, on the opposite of the previous one, is using MPI functions. To implement this operation we used the same principle as for the matrix-vector multiplication using block-striped partitioning approach. There is two different possibilities : - If p divides n evenly, processes P_0 gets the first $\frac{n}{p}$ elements, P_1 the next $\frac{n}{p}$ elements and so on. - If p does not divide n evenly, and r is the remainder, then first r processes get $\frac{n}{p} + 1$ elements and remaining $p - r$ processes get $\frac{n}{p}$ elements. In fact, we choose to use *MPI Allreduce* command combined with the OpenMP command *#pragma omp parallel for reduction*. These both line of code helps to ensure the load balancing by having all ranks perform the reduction. The other option would have been to use MPI Reduce to reduce to Rank 0 and then have Rank 0 broadcast the scalar result to all ranks. This would increase the load on Rank 0 in the same way as explained before in the Main Concept section. In addition, it would create a more complicated code.

We tested both implementations inside our global Conjugate Gradient algorithm. We find out that the CG algorithm using the parallel implementation took 0.00361216 s while the one using the classic implementation took 0.000535268 s per iterations for a square matrix of size 1000. Thus, for this size of matrices, the non-parallelized dot product is 6 times faster than the parallelized one.

2.1.6 Vector Norm Implementation and Optimization

The implemented function allowing this vector norm operation is using the previously described dot product function. In fact, the norm of a vector is just the square root of the dot product of the vector with itself. We thus implemented this function in one line of code. This module was used in the main part of the Conjugate Gradient algorithm to test the overall convergence by looking at the value of the norm of the residual.

2.2 Performance expectations

As said before, the dominating operations during an iteration of the CG algorithm are matrix-vector products. In general, matrix-vector multiplication requires $O(m)$ operations, where m is the number of non-zero entries in the matrix.
Suppose we wish to perform enough iterations to reduce the norm of the error by a factor of $\epsilon = 10^{-8}$. Thanks to mathematical proof and formula, we can show that CG method has a time complexity of $O(m * \sqrt{k})$ while having a space complexity of $O(m)$ where k is the condition number of A.

As we are splitting our input system into several sub-systems to work in a parallel way, the complexity of our algorithm should decrease as with this process, we are decreasing the number m. Thus, the speed up due to this main level of parallelism should be high, even linear for a small number of processors. However, the time taken to execute the gathering of all this sub-system is becoming higher and higher with the increase of the number of processors. Thus, there exists a threshold between these values that will define the magnitude of the speedup. Also, be careful: this implementation would allow a speed up if and only if the size of the problem is enough big to underline the interest of such an algorithm.

Speed-up will be also achieved thanks to the parallelism implemented at the operations level, the matrix vector multiplication being the most important process. However, this speed up should impact less the final result.

3 Prior results

In this section, we will describe the results of scaling studies meaning that we varied the number of OpenMP threads and MPI ranks used for each CG run. The tolerance allowing us to check the convergence of the algorithm was put as 10^{-7} as it seems to be a good compromise between precision and computational intensity. We ran our implemented Conjugate Gradient Algorithm on the Deneb1 cluster at EPFL. This machine presents the following characteristics [1] :

- 376 compute nodes each with 2 Ivy Bridge processors with 8 cores each, CPU running at 2.6 GHz, with 64 GB of memory
- Infiniband QDR 2:1 connectivity
- GPFS filesystem
- Peak performance: 293 TFLOPs (211 in CPUs, 92 in GPUs)

High-performance computing clusters are able to solve big problems using a large number of processors. This is also known as parallel computing, where many processors work simultaneously to produce exceptional computational power and to significantly reduce the total computational time. In such scenarios, scalability or scaling is widely used to indicate the ability of hardware and software to deliver greater computational power when the amount of resources is increased. For HPC clusters, it is important that they are scalable, in other words, that the capacity of the whole system can be proportionally increased by adding more hardware. For software, scalability is sometimes referred to as parallelization efficiency, the ratio between the actual speedup and the ideal speedup obtained when using a certain number of processors.

The speedup in parallel computing can be straightforwardly defined as $speedup = \frac{t_1}{t_N}$ where t_1 is the computational time for running the software using one processor, and t_N is the computational time running the same software with N processors. Ideally, we would like softwares to have a linear speedup that is equal to the number of processors (speedup = N), as that would mean that every processor would be contributing 100% of its computational power. Unfortunately, this is a very challenging goal for real applications to attain.

In this project, we focus on the small/moderately sized matrices because it is more interesting as these are the matrix sizes which the majority of people could frequently encounter in research work.

3.1 Strong scaling

Amdahl's law [5] can be formulated as follows $speedup = \frac{1}{s+\frac{p}{N}}$ where s is the proportion of execution time spent on the serial part, p is the proportion of execution time spent on the part that can be parallelized, and N is the number of processors. Usually by definition, $p = (1 - s)$. Amdahls law states that, for a fixed problem, the upper limit of speedup is determined by the serial fraction of the code. This is called strong scaling. In this part, we will analyze the strong scaling property of our algorithm.

Thus, we are studying the relation between the number of threads/rank given for a run with the speedup it allows. In fact, the plot and tab in Figure 4 and 3 illustrates the speedup for the

Size	OpenMP Threads	MPI rank	Time per CG	Time per iter
5000	1	1	3812.08	0.0823842
5000	1	2	933.78	0.026149
5000	1	4	681.258	0.0192745
5000	1	5	1018.57	0.0211691
5000	2	1	3816.17	0.0824726
5000	2	2	765.292	0.0214308
5000	2	4	687.069	0.0186973
5000	2	5	1127.02	0.023423
5000	4	1	3818.68	0.0825268
5000	4	2	744.149	0.0208387
5000	4	4	635.851	0.0177434
5000	4	5	1056.37	0.0219546

Figure 3: Results of the Strong scaling.

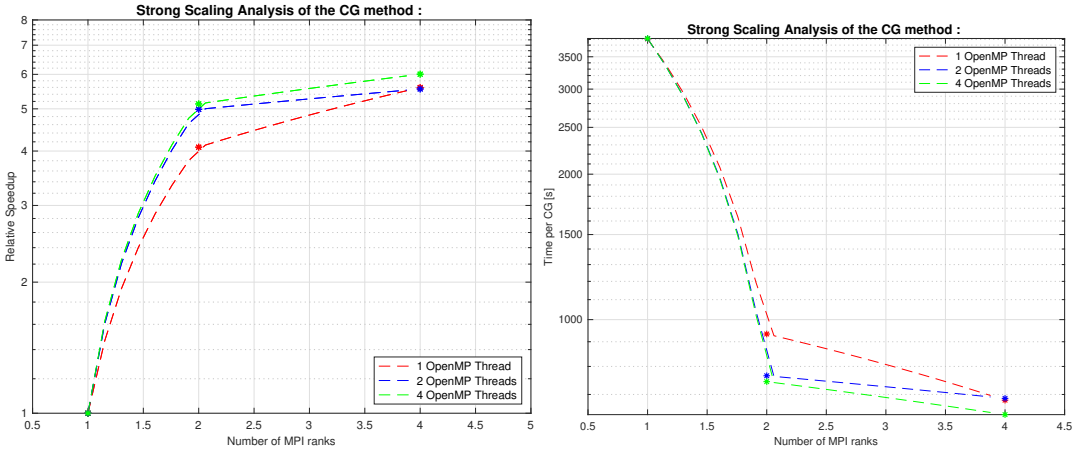

Figure 4: Results of the strong scaling and thread-to-thread scaling studies. Here the matrix size was fixed while the number of ranks and threads was varied as explained above.

Conjugate Gradient solution of a fixed square matrix size of 5000 relative to a single threaded run on 1 processor when using multiple MPI ranks and OpenMP threads. The speedup was determined based on an average of 5 CG solver runs for the same matrix. The different lines represent the strong scaling behavior while following each individual line represents the rank-to-rank scaling.

For our dense input matrices of size 5000, we see that the best speedup is achieved when using the most MPI ranks and threads. The impact of the number of OpenMP threads is visible: the higher this number is the more speed up we get. In addition, the same relation for the number of MPI ranks with the speedup is observed. Here we can conclude about the success of our parallelized implementation: the maximum relative speed-up achieved is around 6 and the increase of both MPI rank and OpenMP threads allows the observation of speed up.

In addition, a decrease of the speedup is observed since a rank number of 5. This decrease in

speedup is probably due to too much overhead of creating so many ranks without enough work available for each thread to perform in parallel. In fact, this decrease can be the illustration of the threshold between the size of the matrix and the number of used ranks that will define the magnitude of the speedup. For example, the time taken to execute the gathering of all the sub-system is becoming higher and higher with the increase of the number of processors. We would expect this kind of behavior in a production scale.

3.2 Weak scaling

Amdahls law gives the upper limit of speedup for a problem of fixed size. This seems to be a bottleneck for parallel computing. However, as Gustafson [6] pointed out that, in practice, the sizes of problems scale with the amount of available resources. If a problem only requires a small number of resources, it is not beneficial to use a large amount of resources to carry out the computation. A more reasonable choice is to use small amounts of resources for small problems and larger quantities of resources for big problems. That's why Gustafsons law is based on the approximations that the parallel part scales linearly with the number of resources, and that the serialized part does not increase with respect to the size of the problem. It provides the formula for scaled speedup: $scaled\_speedup = s + p * N$ where s, p and N have the same meaning as in Amdahls law. In the course we defined it in the following way: given p the number of processors, β the fraction of non-parallelizable code and n the problem size, the speedup is $SpeedUp = p - \beta(n)(p-1)$.

With Gustafsons law, the scaled speedup increases linearly with respect to the number of processors (with a slope smaller than one), and there is no upper limit for the scaled speedup. This is called weak scaling, where the scaled speedup is calculated based on the amount of work done for a scaled problem size (in contrast to Amdahls law which focuses on fixed problem size).

Thus, we are now interested in the study of the time taken for several different OpenMP thread numbers as a function of MPI ranks and problem size such that the problem size per MPI rank remains constant. Indeed, for weak scaling, the matrix size was increased proportionally to the number of MPI Ranks used. Figure and tab 6 and 5 illustrates the weak scaling properties. In this case, we will be focused on the time per iteration as the size and the state of the problem is always changing. It would allow us to have an unbiased comparing variable.

In Figure 6, we can observe a strange scaling behavior. Indeed, we would expect more flat/-linear curves when 4 or less MPI ranks were used. In fact, when the number of MPI rank is set to 2 with a square matrix size of 1414, a non expected huge speedup is observed. It seems that we found optimal conditions for our implementation. In fact, as you can see on Figure 6, even if we increase the size of the system we want to solve, the CPU time per CG iter and the CPU time per CG run is a lot smaller than for the other case. Indeed, we would more expect a linear curve. This effect seems to be due to the following command line : *__gnu_parallel::inner_product()*.

On the opposite, we would have more expected a curve being above a linear regime. In fact, an increase of this kind would be probably due to significant overhead in the communication which was not compensated enough by the performance gain of using more ranks. Recall that we used MPI Allreduce and MPI Allgatherv which would result in more communication CPU time with larger problem size even when the number of math operations per rank remains the

Size	OpenMP Threads	MPI rank	Time per CG	Time per iter
1000	1	1	17.7332	0.00360577
1414	1	2	16.4894	0.00226471
2000	1	4	45.9016	0.00430072
1000	2	1	18.948	0.00360708
1414	2	2	16.9167	0.0023327
2000	2	4	47.8373	0.0047284
1000	4	1	11.0015	0.00360943
1414	4	2	16.5549	0.00228911
2000	4	4	65.4949	0.00487931
1000	8	1	18.0697	0.00360601
1414	8	2	22.8271	0.00319439
2000	8	4	44.3385	0.00480062
1000	16	1	12.2899	0.00361254
1414	16	2	16.42361	0.00230022
2000	16	4	44.0561	0.00451811

Figure 5: Results of the Weak scaling.

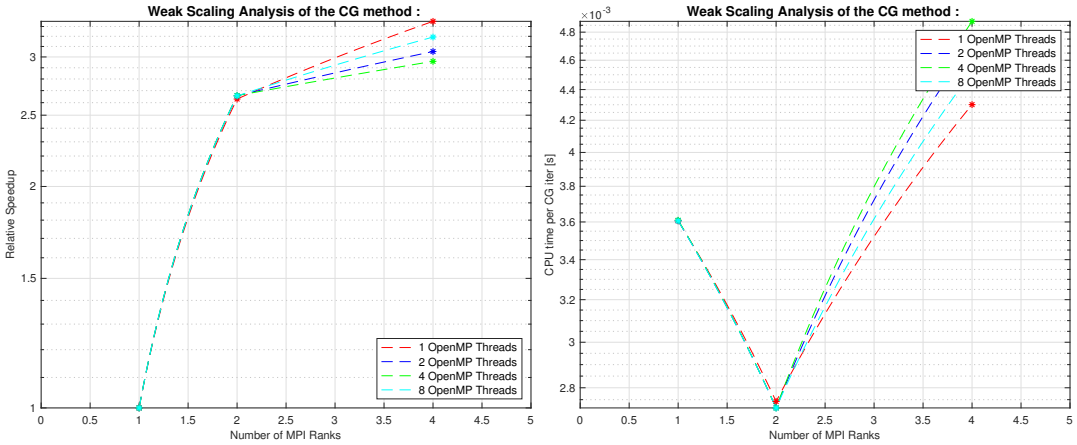

Figure 6: Results of weak scaling studies. For weak scaling, the matrix size was increased proportionally to the number of MPI Ranks used as explained before.

Matrix Size	Raw Storage Needed for the Matrix A and vector b
1000	10.7 Mo
2000	42.6 Mo
5000	266 Mo
10000	1.06 Go

Figure 7: Size of the system and the corresponding storage needed.

same. However, a kind of optimal set of parameters has been found instead of this expected behavior. At a production scale, we would expect this overhead behavior.

4 Resources budget

In order to fulfill the requirements of our project, we present hereafter the resource budget.

4.1 Computing Power

The needed computing power depends on the size of the system we want to focus on. In our case, we would like to study system with a square matrix of size between 1000 and 10000 rows or columns elements. Thanks to our scaling study, we are able to identify several optimal quantities. In fact, in the case of the studied system of matrix size 5000, we would need to have access to 4 total number of cores. We choose this amount as our best relative speedup of 6 was achieved with 4 MPI ranks and 4 OpenMP threads.

4.2 Raw storage

As we explained all along the report, the memory usage is optimized through MPI distributed memory parallelization and the data-parallel approach (we divide the main operation in a block stripped manner for example). Thus, we reduce the data each processor work on. This fact will allow the data to fit into a higher level cache than if we were to operate on the entire matrix and vectors.

In the end, the only issue would be the storage of the raw data. However, as you can see in Figure 7, the type of square matrix we study (with size between 1000 and 10 000) do not take to many storages. However, this raw storage is obviously mandatory for the success of the project. It's nothing as it is a maximum of 1 Giga. The increase in the size of the system would follow the increase of the storage needed.

In addition thanks to Intel report and by looking at command htop on the terminal during one run, we were able to observe the quantity of memory needed per run: a little bit more than the value of the stored input matrix is needed. Thus the temporary disk space for a single run would be near the value of the raw storage needed for the input of the system.

4.3 Grand Total

Let's say that we want to analyze systems with square matrix of size 100 times higher than what we made in this report. We thus would have matrices with a storage value of 100 Go. We will focus on one problem only. This will give us, based on the performance expectations and benchmarks already done, the following resource budget :

Total number of requested cores	256 - 512 [cores]
Minimum total memory	10[GB]
Maximum total memory	256[GB]
Temporary disk space for a single run	100[GB]
Permanent disk space for the entire project	300[GB]
Communications	Hybrid MPI and OpenMP
License	own code (BSD)
Code publicly available ?	Yes
Library requirements	Python2, Numpy, omp.h, mpi.h
Architectures where code ran	Intel 64, BG/Q

5 Scientific outcome

This study demonstrates a straightforward hybrid parallelization of the CG algorithm for solving matrix equations. A substantial speedup over the serial version was achieved. The parallelization was structure in two levels: a higher one which uses a data-based approach by splitting the matrices and vectors into horizontal blocks. Then, the lower level uses both MPI and OpenMP to optimize our implementation at a matrix and vector operations scale. A moderate number of threads provides the best performance. These operations were optimized by trying several ways of parallelizing.

This work is applicable to areas of scientific computing where the model results in a symmetric and positive- definite matrix. Thus, the field of physics (field of solid physic for example) could benefit from this study.

Moreover, this research could be improved with additional features that would allow a wider range of people benefits from it. In fact, Sparse-matrices frequently arise in scientific computing. For example, the equations resulting from finite difference discretization can be written as sparse matrix equations. We could adapt our project to this specific matrix types. This would allow us to achieve really good results. In fact, we can take advantage of the sparse property of these matrices to achieve significant performance improvements by storing and operating on only the non-zero elements.

References

[1] Scientific IT and Application Support, http://scitas.epfl.ch, 2015

[2] Optimization for Machine Learning, EPFL course, https://github.com/epfml/OptML_course

[3] The MPI Forum, *MPI: A Message-Passing Interface Standard*, Technical Report, 1994

[4] Convex Optimization 2, Standford course, https://web.stanford.edu/class/ee364b

[5] Amdahl, Gene M. (1967). AFIPS Conference Proceedings. (30): 483485. doi: 10.1145/1465482.1465560

[6] Gustafson, John L. (1988). Communications of the ACM. 31 (5): 532533. doi: 10.1145/42411.42415

[7] C.C. Paige and M.A. Saunders, Solution of sparse indefinite systems of linear equations, SIAMJ. Numer. Anal, 12 (1975) 617-629.

[8] J.K. Reid, On the method of conjugate gradients for the solution of large sparse systems of linear equations, in: J.K. Reid, ed., Large Sparse Sets of Linear Equations (Academic Press, New York, 1971) 231-254.

6 Nicolas Lesimple, Master – CV

Nicolas Lesimple is a mix of a computer scientist and a mathematician born in Annecy, France. He received his Bachelor degree in Bioengineering from the Ecole Polytechnique Fdrale de Lausanne (Switzerland) in 2017. From 2017 to nowadays, he studies in the Master called Computer Science and Engineering in the Math section. During his master, he did two academic projects. The first one was done in 2018 at EPFL. The goal was to create a tool allowing the Cervical Cancer Image Segmentation. The task was the analysis of sequences of dynamic images (videos) of the cervix under a contrast agent and the development of an Android application. The second project, also done at EPFL in 2019 is about Automated Machine Learning. This project is done in partnership with L2F, a startup based at EPFL. The goals are to compare existing libraries of automated machine learning to several pipelines developed by the company. The ultimate objective is to combine actual knowledge to develop innovative tools for the company. He also works in L2F company as a Data Science Intern and then as a Modeling Expert. Conceptualizing, designing and implementing customized predictive models using advanced machine learning techniques applied to various industries were his goals. This professional experience allows him to interact with professionals from this broad spectrum of industries and to approach complex business problems from both a mathematical and a business perspective, in order to deliver performance-enhancement ad-hoc solutions to corporate clients. His research interests are mainly Machine Learning and all the potential application this field can create. In this context, the HPC field is extremely important as one of our principal limit today is the computational power.

Areas of expertise :

- Computational Science and Engineering
- Machine Learning
- Data Science
- Image Processing, Statistics, Optimization,